**Java**

# 附

## 附：计算机简介

（一）计算机结构：

硬件：CPU，内存，I/O。 （0,1）

软件：系统软件（Windows，Linux，Mac）

应用软件

（二）计算机语言：

机器语言：二进制

汇编语言：指令语句

高级语言：Java，C，C++，Python，C#，

解释型：类似读书，从上而下，一次解释执行，JavaScript

编译型：Java（.java编译成.class字节码文件）

1字节=8位，1位0/1。

## 附：软件开发简介

（一）软件的声明周期：

问题定义（立项），

可行性分析（经济可行性，实用可行性），

总体描述（需求分析），

系统设计（编码设计，数据库设计），

编码，

调试和测试，

验收和运行，

维护升级，

废弃，

（二）软件设计目标：

正确性，健壮性，灵活性，可重用性，高效性。

1. 良好设计的特征：

可扩展性，灵活性，可插入性，

高内聚低耦合

（四）面向对象：继承性，封装性，多态性

（五）软件开发团队：

需求分析师，系统架构师，前端设计师，

Java工程师，测试工程师，运维工程师。

## 附：Eclipse用法

1.Sourse format：代码格式化

2.Sourse Generate：自动生成 构造函数 等；

3.代码调试DeBug模式：DeBug As运行

断点BreakPoint

Resume：执行至下一断点

StepOver：按句执行

StepInto：进入方法内部执行

4.源代码：按住Ctrl 移到代码处；（src.zip）

**JAVA**

# Java编程基础

## Java开发入门

#### 一、Java语言特点

1.简单性：取消了C++的指针和多继承

2.面向对象：封装、继承、多态；

3.编译和解释性：

4.可移植性：跨平台，一次编译，处处运行。（JVM）

跨平台原理：Java语言跨平台，JVM不跨平台。

由于不同平台上有不同的JVM，可以将同一份\*.class执行为不同的机器码。

#### JDK，JRE，JVM

JDK（Java Development Kit）：Java开发工具包，程序开发使用；编译；

JRE（Java Runtime Environment）：Java运行环境；运行；

JVM：解释执行字节码文件（.class→相应OS机器码）

JVM作用：

①执行代码：解释执行字节码文件

②管理内存：将数据加载到内存中去

③垃圾回收机制：空闲时间不定时的动态回收无任何引用的对象所占用的空间。（只需new，不用delete）

编译阶段：语法错误；检查是否存在语法错误，将\*.java编译成\*.class。

运行阶段：内存错误；将\*.class文件加载到内存中，执行程序，传递参数

#### 三、Java开发环境搭建

##### （一）JDK安装

1.从oracle官网下载JDK

2.双击.exe文件安装：安装路径不允许出现中文、空格、@#$等特殊字符。

##### （二）验证JDK安装是否正确

cmd：Windows系统→命令提示符 || win+R→输入cmd。

java -version

##### 配置path环境变量

|  |
| --- |
| Path：可执行文件的路径； |
| JAVA\_HOME：D:\LiziChen\Java\JDK  相对路径Path：%JAVA\_HOME%\bin;（引用JAVA\_HOME）  绝对路径Path：D:\LiziChen\Java\JDK\bin; |
| path 环境变量是系统环境变量中的一种，它用于保存一系列可执行文件的路径，每个路径之间以分号分隔。当在命令行窗口运行一个可执行文件时，操作系统首先会在当前目录下查找是否存在该文件，如果不存在会继续在 path 环境变量中定义的路径下去寻找这个文件，如果仍未找到，系统会报错。 |

##### （四）配置classpath

|  |
| --- |
| classPath：指定class文件的路径； |
| CLASSPATH=.;%JAVA\_HOME%/lib/dt.jar;%JAVA\_HOME%/lib/tools.jar |
| classpath 环境变量也是系统环境变量中的一种，它用于保存一系列类（.class）文件的路径。当 Java 虚拟机需要运行一个类时，会先在 classpath 环境变量中所定义的路径下去寻找所需的 class 文件。如果 classpath 的路径最后有分号，则在 classpath 中找不到时，再到当前路径查找；如果最后无分号，则只在 classpath 设置的路径中查找。 |

标准配置环境变量的方式

JAVA\_HOME = D:\LiziChen\Java\JDK

PATH = %JAVA\_HOME%/bin;%JAVA\_HOME%/jre/bin

CLASSPATH = .;%JAVA\_HOME%/lib/dt.jar;%JAVA\_HOME%/lib/tools.jar

面试题：

classPath与Path的区别？

|  |  |
| --- | --- |
| Path： | 保存一系列可执行文件的路径。 |
| classPath： | 保存一系列类（.class）文件的路径。 |

#### 四、Java程序

##### （一）第一个Java程序：HelloWorld.java

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

1.编译源文件：javac HelloWorld.java

2.运行程序：java HelloWorld

3.输出结果：HelloWorld

##### （二）Java程序的执行

1.编写Java源代码（\*.java）

2.源程序编译成字节码文件（\*.class）

3.JVM执行\*.class文件

##### （三）Java代码规范

1.代码要有层次

2.代码要有缩进，{}内缩进一个table

3.先写结构，再填充内容。（保证符号不出错）

4.运算符两端与操作数之间预留空格。

5.if(){ }、for(){ }、while(){ }、do{ }while()即使只有一条语句，也不要省略{}。

##### （四）Java文件规范

1.文件名=public修饰的类名

2.在一个Java文件中，可以有多个类，只能存在一个public类

3.在一个Java文件中，在public类中，只能存在一个标准main方法

#### 五、Java程序内存逻辑划分

类加载器：将类加载到内存中去

①栈内存：系统自动分配释放；局部变量。

虚拟机栈，本地工作栈；

②堆内存：手动分配释放（new）；

③方法区：

④程序计数器：

#### 六、注释

注释：方便程序阅读而写的一系列说明文字；提高代码可读性。

|  |  |
| --- | --- |
| 1.单行注释： | //单行注释 |
| 2.多行注释： | /\*多行注释  \*多行注释不可嵌套多行注释/\*...\*/  \*多行注释可以嵌套单行注释//  \*/ |
| 3.文档注释： | /\*\*文档注释  @author 开发该类模板的作者  @version 该类模板的版本  @param 对方法中某参数的说明  @return 对方法返回值的说明  @exception 对方法可能异常进行说明  \*/ |
| 使用javadoc命令生成html说明文档  javadoc -d doc HelloWorld.java  -d doc:文件位置 |

#### 附、常用Dos命令：

1.切换盘符：直接键入盘符+“:”，例如d:

2.进入指定目录：cd 要进入的路径

3.cd.. 目录回退一级，cd/ 回退到当前盘符根目录

4.查看IP地址：ipconfig；ipconfig -all；

1. 检查网络连通性：ping IP地址/网址，发送四次

ping IP地址 -t，持续发送数据包

6.结束dos窗口：Ctrl+C

7.清屏：cls

8.查看当前目录下的文件/子文件：dir

9.打开注册表：win+R，输入regedit

DOS命令、快捷键

## Java基本语法

### 数据类型与运算符

#### 标识符

标识符：自定义名称

包名、类名、接口名、方法名、参数名

##### （一）Java标识符命名规则：

|  |
| --- |
| ①只能是字母，数字，下划线\_,美元$  ②严格区分大小写，a与A不同（字母约束）  ③不能使用Java关键字和保留字（字母约束）  ④不能以数字开头（数字约束）  ⑤命名支持中文，但不允许使用（字符约束）  ⑥理论上，命名长度不受限制 |

##### （二）Java标识符命名约定：

|  |
| --- |
| 见名知意  驼峰命名法：首个单词以类名规则命名，其余单词首字母大写  ①类名规则：类名首字母大写，驼峰命名法  ②接口名规则：同类名，默认接口名前加I  ③方法名规则：方法名首字母小写；驼峰命名法  ④参数名规则：同方法名  ⑤包名规则：所有字母小写，用“.”分隔  ⑥常量名规则：所有字母大写，用“\_”分隔 |

#### 关键字

关键字：Java赋予特殊含义的单词，51个

特点：全小写

保留字：准备使用，还未使用（goto，const）

#### 变量和常量

##### （一）常量

1.常量：程序运行中值固定不变的量

2.声明形式：final type CONST\_NAME = value;

（赋值一次，不能改变）

3.常量分类：自定义常量&字面量1,2,3...

4.注意：final修饰的变量为常量，只能赋值一次。

##### （二）变量

1.变量：Java程序中状态可变的值，最基本的存储单元

本质：内存中的一块连续的存储空间。

要素：数据类型，变量名，变量值，变量作用域

2.声明形式：type varName [=value];

3.变量分类：按照变量类型：基本类型变量&引用类型变量

按照变量声明位置：成员变量&局部变量

4.注意：a.Java强类型语言，变量使用前必须先声明。

b.Java局部变量必须初始化，成员变量有默认值。

c.变量的值，修改即被覆盖。

#### 四、数据类型

Java是一种强类型语言

##### （一）常量数据类型

|  |  |
| --- | --- |
| ①整型常量： | 二进制（0b/0B），  十进制（字面量），  八进制（0），容易混淆，建议不要使用。  十六进制（0x/0X），  默认int型 |
| ②浮点数常量： | float单精度浮点数（\*F/\*f）  double双精度浮点数（\*D/\*d）  默认double型 |
| ③字符常量： | 单字符‘a’ |
| ④转义字符： | \n -- 换行 \t -- 制表  \r -- 回车 \b -- 退格  \’-- ‘ \”-- “ \\ --- \ |
| ⑤字符串常量： | “abc” |
| ⑥布尔常量： | true，false。 |
| ⑦null常量： | null（表示对象的引用为空） |

Java7开始，前缀0b表二进制。

Java7开始，数字字面量可以加下划线便于阅读，如1\_000\_000。

##### （二）变量数据类型

基本数据类型：

|  |  |
| --- | --- |
| 1.整型： | ①byte（1B）(-128~127)，补码，数值与符号复合  (127后面是-128，最大的正数加下来是最小的负数)  ②short（2B）  ③int（4B）默认为int，正好超过20亿  ④long（8B）末位加\*L/\*l。 |
| 2.浮点型： | ⑤float（4B）末位加\*F/\*f。  ⑥double（8B）默认为double。  跨平台：固定表数范围和字段长度  不能参与 禁止误差的金融运算， |
| 3.字符型： | ⑦char（2B）一般2B也可能4B，建议用String |
| 4.布尔型： | ⑧boolean（1bit）：true，false。 |

引用数据类型：类(class)，接口(interface)，数组，枚举，注解

面试题：

1.Java中八种基本数据类型？

byte，short，int，long，float，double，char，boolean

2.String是否是一种数据类型？

不是，是String类名。

3.在一个char中能否存储一个汉字？

能，一个char变量2个字节，一个汉字2个字节。

##### （三）数据类型转换

|  |  |
| --- | --- |
| 自动转换： | 容量小的类型→容量大的类型， |
| 强制转换： | 容量大的类型→容量小的类型，精度损失或溢出。 |

1.小容量足够，则正确；超出范围，则精度损失。

2.byte，short，char之间不会相互转换，在计算时会自动转换为int类型进行计算。

##### （四）不同类型数据之间的运算结果的数据类型

|  |
| --- |
| Java运算时，自动提升至最大数据类型。 |
| ①整型byte，short，int运算 --> 运算结果int  ②整型int，long运算 --> 运算结果long  ③整型int，浮点型float运算 --> 运算结果float  ④浮点型之间运算 --> 运算结果double  ⑤字符型char = ASCII表int值，可以跟任何类型进行转换  ⑥布尔型不能与其他任何类型进行运算 |

#### 五、成员变量和局部变量

按照变量声明位置：成员变量&局部变量

|  |  |  |
| --- | --- | --- |
|  | 成员变量 | 局部变量 |
| a.定义 | 声明在类内部，方法外部的变量 | 声明在方法内部的变量（包括形参） |
| b.作用域 | 类内有效。 | 方法内有效。 |
| c.初始化 | 可不进行初始化，有默认值  -整型默认值0  -浮点型默认值0.0  -char默认值‘\u0000’  （空字符，不是空格）  -布尔型默认值false  -String默认值null | 必须初始化，无默认值 |
| d.声明周期 | 随着对象的产生（消失）而产生（消失） | 随着方法的调用（结束）而产生（结束） |
| e.内存分配 | 堆内存（new） | 栈内存 |
| 在方法内，成员变量与局部变量同名，变量名访问的是局部变量； | | |
| 变量作用域：定义的最近大括号中 | | |

#### 六、运算符

|  |  |  |
| --- | --- | --- |
| **※**1.算术运算符：+，-，\*，/ ，% ,++，--   |  | | --- | | ①i++：先执行后自增； | | ②++i：先自增后执行； | |
| **※**2.逻辑运算符：&，|，^，!，&&，||，~，   |  | | --- | | ①逻辑与 &：&前false，&后表达式依旧执行。（按位与） | | ②短路与 &&：&&前false，&&后表达式不执行。 | | ③逻辑或 |：|前true，|后表达式依旧执行。（按位或） | | ④短路或 ||：||前true，||后表达式不执行。 | | ⑤异或 ^：同0，异1；异或三次交换值；  （m=m^n; n=m^n; m=m^n;） |   面试题：交换两个整数的值，不使用中间变量  （1） x=x+y; y=x-y; x=x-y;  （2）x=x\*y; y=x/y; x=x/y;  （3）m=m^n; n=m^n; m=m^n; |
| **※**3.位运算符：<<，>>，>>>，~，&，|，^，（按位运算）  位运算是所有运算中，最快速高效的运算方法。   |  | | --- | | ①左移<<：乘； x移n位，x乘2^n | | ②右移>>：除； x移n位，x除2^n | | ③无符号右移>>>：补0 |   面试题：使用最快速高效的方式，计算2e3。 位左移1<<3；  判断奇偶数：((num&1)==0) ? “偶数”! “奇数” |
| 4.赋值运算符：=，+=，-=，\*=，/=，%=，  a+=b; //a=a+b; |
| 5.条件运算符：a?b:c  条件a==true，返回b；条件a==false，返回c； |
| 6.关系运算符：>，<，>=，<=，==，!=， |

赋值运算符，强转自动完成。

%：忽略负号，求得余数，余数正负号由%前决定；

#### 七、表达式

表达式：符合一定规则的运算符和操作数

表达式的值：运算结果

表达式的类型：表达式的值的数据类型

表达式运算顺序：

#### 附、常用字符集

|  |  |
| --- | --- |
| ASCII码 |  |
| ISO-8859-1： | 欧洲 |
| GBK/GB2313： | 国标 |
| Unicode： | 统一了世界上所有文字的编码  UTF-8，UTF-16，UTF-32。  （Java使用UTF-16,16位） |

### 流程控制

#### 一、顺序控制语句

在Java中的程序执行流程：

从main方法开始执行，从main方法的方法体第一句开始，依次按语句的先后顺序，从上而下，逐句执行。

附：

●获取键入值

java.util.Scanner sc = new java.util.Scanner(System.in);

sc.nextInt(); //键入一个整型数据

sc.next(); //键入一个字符串

●延时输出：

try{

Thread.sleep(10);

}//try

catch(Exception e){

e.printStackTrace();

}//catch

● ； //一条空语句

● long System.currentTimeMillis()；返回1970-1-1 00:00:00至今的毫秒数

#### 二、选择控制语句

在程序执行过程中，在某些时候，我们代码的执行，需要满足一定的条件之后才去执行。

##### **if语句**

if语句的五种类型：

|  |  |
| --- | --- |
| 1、if...  if(条件/布尔表达式){  语句1;  } | 2、if...else...  if(条件/布尔表达式){  语句1;  }else{  语句2;  } |
| 3、if...else if...  if(条件/布尔表达式1){  语句1;  }else if(条件/布尔表达式2){  语句2;  } | 4、if...else if...else...  if(条件/布尔表达式1){  语句1;  }else if(条件/布尔表达式2){  语句2;  }  ......  else{  语句n;  } |
| 5、嵌套  if(条件/布尔表达式1){  if(条件/布尔表达式2){  语句2;  }  } |
| ●if针对的是boolean值进行判断； | |

##### **switch语句**

|  |  |
| --- | --- |
| switch(表达式){  case 常量1:  语句1;  break;  case 常量2:  语句2;  break;  .......  default:  语句N+1;  break;  } | **※**switch后的表达式的类型：  byte，short，int，char，  JDK5引入 枚举；JDK7引入 String； |
| switch关键字：  1.break：跳出当前switch。  -无break会发生case穿透。  2.default：case常量以外的值。  -可以写在switch任意位置，  -一般写在末尾，搭配break。  3.case常量值不可以重复。  4.case连续多个常量值对应相同语句，只需要在最后一个case后写上语句即可。 |
| ●switch针对的是一些特定的，可列举的，有限的值。 | |

面试题：

1.switch后的表达式的值类型可以是什么类型？

2.switch后的可以是int？可以是long？可以是String？ 3.switch中的default必须出现在末位吗？ 不是，通常放在末尾。

|  |  |
| --- | --- |
| if与switch | |
| 区别： | 1.if针对的是boolean值进行判断；  2.switch针对的是一些特定的，可列举的，有限的值。 |
| 联系： | 凡是switch都可用if表示，if不一定能用switch表达。 |

#### 三、循环控制语句

|  |  |  |  |
| --- | --- | --- | --- |
|  | for循环 | while循环 | do...while;循环 |
| 格式： | for(初始化;条件;控制语句){  循环体;  } | 初始化;  while(条件){  循环体;  控制语句;  } | 初始化;  do{  循环体  }while(条件);  先执行后判断 |
| 区别： | 初始化变量作用域：  仅for循环内 | 初始化变量作用域：  不仅限于while循环内 | |

●for( System.out.println("此处执行一次") ; ; System.out.println("此处循环执行") ) { }

●死循环：循环条件永远为true。

死循环利与弊：

1.对于一个真实的应用程序，死循环应该规避；

2.对于我们实际应用，要根据具体情况而定；

有些特殊领域（eg.单片机）会需要死循环；

面试题：

1. for循环的三部分是什么？如果不存在是否可以不存在？不存在程序是否能正常执行？

三部分：初始化语句，条件语句，控制语句

可以不存在，可以写在for语句的外面。

不存在可以正常执行，但是是死循环。

2.利用没有循环体的for循环结构，输出HelloWorld

|  |
| --- |
| for( System.out.println("此处执行一次") ; ; System.out.println("此处循环执行") ) { } |

3.输出九九乘法表

|  |
| --- |
| for(int i = 1;i < 10;i++){//行数  for(int j = 1;j <= i;j++){//列数  System.out.print(j+"\*"+i+"="+(i\*j)+"\t");  }  System.out.println();//一行一换行  } |

4.循环优化

for(int i = 0;i < 1000;i++)

for(int j = 0;j < 100;j++)

for(int k = 0;k < 10;k++)

testFunction(i,j,k);

优化：时间（执行效率）、消耗资源

for(int i = 0;i < 10;i++)

for(int j = 0;j < 100;j++)

for(int k = 0;k < 1000;k++)

testFunction(i,j,k);

#### 四、跳转语句

|  |  |
| --- | --- |
| break： | 跳出当前循环，跳出一层循环； |
| continue： | 跳出本次循环，进行下次循环； |
| return： | 结束整个方法；  **※**return语句必须是最后一句；  return语句后不能再有其他语句； |

面试题：

如何break跳出多重for循环：

|  |  |
| --- | --- |
| 1.使用标签： | break STOP;跳转至STOP:处（任意跳不推荐） |
| 2.使用标识符： | flag=true;（跳出层循环条件 && flag）;  不满足条件，则flag=false;break; |

### 方法

#### 一、方法

1.方法：完成特定功能的代码块

2.方法声明形式：

|  |
| --- |
| [访问修饰符] 返回值类型 方法名(形参列表){  方法体;  } |
| ●访问修饰符：public，protected，default，private，  ●静态修饰符static，最终修饰符final  ●返回值类型：基本数据类型，引用类型，void，  ●参数列表：参数个数，参数类型； |

1. 方法调用：

|  |
| --- |
| 所有方法的执行，都必须在main方法中进行调用；  非静态方法（无static修饰）可调用 静态方法/非静态方法；  静态方法（有static修饰）只能调用 静态方法；  ①调用静态方法：  a. 类对象.方法名()；fun.test();  b. 方法名直接调用；test();  ②调用非静态方法：  a.类对象.方法名()；fun.test(); |

4.方法之间的关系：平行关系，相互调用；

5.return关键字

|  |
| --- |
| **※**return语句必须是最后一句，return;语句后不能再有其他语句； |

面试题：

1.对于返回值为void的方法，能写return吗？

能，return; 即return 空；

2.在return; 之后，能不能写其他语句？

不能，return; 后是 无法访问的语句；

#### 二、方法的重载（Overload）

1.判断方法是否重载：

|  |
| --- |
| ①方法名相同  ②参数列表不同（参数个数不同，参数类型不同）  （参数顺序不同（针对参数列表中存在不同类型数据）） |
| \*③返回值类型不同，不能界定方法是否重载； |

2.在一个类中，不能存在方法的定义形式完全一样的两个方法；

#### 三、方法的递归调用

1.递归调用：方法自己调用自己

2.结束条件：条件判断

3.关键：递归式+结束条件；

#### 四、main方法

1.main方法定义形式固定

2.唯一可改变的是 形参名称args

3.main方法的执行不需要显式调用，由JVM在类加载完成后，主动调用。

### 数组

#### 一、一维数组

1.数组：同种类型数据，有组织的存放于一定的内存空间里

2.声明形式：

数据类型 数组名[] = new 数据类型[长度]; C++，C#

数据类型[] 数组名 = new 数组类型[长度]; Java

int[] array = new int[10];

3.数组访问：

数组名[数据下标]

数组长度属性：arrayName.length

4.数组默认值：

①整型0，浮点型0.0，char型’\u0000’，布尔false；

②引用类型null；

5.数组初始化：

①动态初始化：先定义数组，再通过下标赋值

数组名[下标] = 值；

②静态初始化：定义时就直接赋值

数组类型[] 数组名 = {1,2,3,4,5}; //值之间用,分隔

数组类型[] 数组名 = new 数组类型[]{1,2,3,4,5};

1. 数组内存分配

栈内存：系统自动分配释放；局部变量。

堆内存：手动分配释放（new）；

7.数组使用异常：

①数组下标越界：ArrayIndexOutOfBoundsException

②空指针异常：NullPointerException

array = null，array指针不再指向堆内存中任何一块空间

1. 数组常用操作

遍历：for循环，正序，逆序；

查找：顺序查找

排序：冒泡排序，选择排序，插入排序，快速排序

1. 参数传递

①值传递：基本类型数据传递（传递数据）

②引用传递：引用类型数据传递，参数指针传递；（传递地址）

引用关系：栈内存存放地址，引用堆内存内数据

引用变量值更改：return；

10.数组工具类Arrays

#### 二、二维数组

1.声明形式：int[][] a = new int[row][column];

2.二维数组初始化

①动态初始化：

a[i][j] = value;

②静态初始化：

int[][] a = {{1,2,3,4 } , {1,2,3,4 } ,{1,2,3,4 } ,{1,2,3,4 } };

int[][] a = new int[][]{ { 1,2,3,4 } , {1,2,3,4 } , {1,2,3,4 } };

面试题：

1.杨辉三角

|  |
| --- |
| //杨辉三角  public void pascalTriangle(int row){  int[][] arr = new int[row][row];  for(int i = 0;i < arr.length; i++){  for(int j = 0; j <= i ; j++){  arr[i][0] = 1;  arr[i][i] = 1;  }//for  }//for  //arr[i][j] = arr[i-1][j-1] + arr[i-1][j];  for(int i = 1;i < arr.length;i++){  for(int j = 1; j <= i;j++){  arr[i][j] = arr[i-1][j-1] + arr[i-1][j];  }//for  }//for  //三角输出  for(int i = 0;i < arr.length ;i++ ){  for(int j = 0; j <= i; j++){  System.out.print(arr[i][j]+"\t");  }//for  System.out.println();  }//for  System.out.println();  }//pascalTriangle |

2.矩阵相乘

|  |
| --- |
| //矩阵相乘  public int[][] MatrixMultiplication(int[][] a,int[][] b){  int[][] c = new int [a.length][b[0].length];  for(int i = 0;i < a.length;i++){//行  for(int j = 0;j < b[0].length;j++){//列  for(int k = 0;k < c.length;k++)  c[i][j] = c[i][j] +a[i][k]\*b[k][i];  }  }  return c;  } |

#### 三、排序算法

1.冒泡排序

|  |
| --- |
| // 冒泡排序 ：大的沉底，小的上浮  **public** **void** bubbleSort(**int**[] arr) {  **for** (**int** i = arr.length - 1; i > 0; i--) {  **for** (**int** j = 0; j < i; j++) {  **if** (arr[j] > arr[j + 1]) {  *swap*(arr, j, j + 1);  }  } // for  } // for  }// bubbleSort |

2.选择排序

|  |
| --- |
| // 选择排序：缩小范围，两两比较  **public** **void** selectSort(**int**[] arr) {  **for** (**int** i = 0; i < arr.length - 1; i++) {  **for** (**int** j = i + 1; j < arr.length; j++) {  **if** (arr[i] > arr[j]) {  *swap*(arr, i, j);  } // if  } // for  } // for  }// selectSort |

3.插入排序

|  |
| --- |
| // 插入排序：多次比较，一次插入  **public** **void** insertSort(**int**[] arr) {  **int** j = 0;  **int** key = 0;  **for** (**int** i = 1; i < arr.length; i++) {// 抽取新牌  key = arr[i];// 新牌  **for** (j = i - 1; j >= 0 && key < arr[j]; j--) {  arr[j + 1] = arr[j];  }  arr[j + 1] = key;// 已经j--，必须j+1  } // for  }// insertSort |

4.快速排序

|  |
| --- |
| /\*  \* 快速排序：交换排序  \* pivot：枢轴指针; pivotkey：枢轴值，  \* low：低指针， high：高指针，  \* partition(数组名称,起始位下标,末尾位下标);  \* qSort(数组名称,起始位下标,末尾位下标);  \*/  // 分区函数，返回枢轴指针  **public** **int** partition(**int**[] arr, **int** low, **int** high) {  **int** pivotkey = arr[low];// 设第一个元素为枢轴值  **while** (low < high) {  **while** (low < high && arr[high] >= pivotkey) {  high--;  } // while  arr[low] = arr[high];  **while** (low < high && arr[low] <= pivotkey) {  low++;  } // while  arr[high] = arr[low];  } // while  arr[low] = pivotkey;// 将枢轴值 给到 可覆盖区  **return** low;// 返回枢轴指针  }// partition  // 分区递归  **public** **void** qSort(**int**[] arr, **int** low, **int** high) {  **if** (low < high) {  **int** pivot = partition(arr, low, high);// 枢轴指针  qSort(arr, low, pivot - 1);// 前半部分  qSort(arr, pivot + 1, high);// 后半部分  }  }// qSort  **public** **void** quickSort(**int**[] arr) {  qSort(arr, 0, arr.length - 1);  }// quickSort |

# Java常用API

API：应用程序编程接口；常用的功能的类的封装。

用来实现常见的和通用的功能的类的封装。

类库：API 根据实现的功能不同，划分为不同的集合，每个集合组成一个包，称为类库

### 一、Object类：根类

1.Object是类层次所有类的根类/基类/父类/超类；Object的方法对于其他类都可直接使用。

2.Object常用方法：

|  |
| --- |
| ①boolean equals() 比较两个对象的值是否相等；  对象相等：equals()为true； |
| ②int hashCode() 返回该对象的哈希码值（唯一值）；  对象相等：hashCode()相等； |
| ③String toString() 返回该对象的字符串表示； |
| ④void finalize() 当垃圾回收器确定不存在对该对象的更多引用时（垃圾回收时），由对象的垃圾回收器调用此方法。  a.对象的finalize方法不一定会被调用，即使是进程退出前  b.发生gc()时一个对象的内存是否被释放取决于是否存在该对象的引用，该对象如果包含成员，那么对象成员遵循本条。  c.对象里包含的对象成员，按声明顺序进行释放。 |
| // System.gc()：人为调用垃圾回收器  注意：在实际开发中，不要去重写finalize()方法;也不要主动的去使用System.gc(); |

面试题：

请说出fianl、finalize、finally 三个关键字的使用区别?

①final最终修饰符，

②finalize：

### 二、String字符串

#### （一）String字符串

●String：字符串的抽象数据类型；提供了所有关于字符串的操作；

●String本质：不可变的字符序列，字符数组；字符串常量；

##### 1.创建字符串：

**※**所有字符串都是字符串常量，创建后值不可变；

**※**相同的字符串在字符串常量池只有一份；常量池数据可共享

|  |
| --- |
| ①String str1 = “abc”; //str1指向字符串常量池 |
| ②String str2 = new String(“abc”); //str2指向堆内存地址 |
| ③String str3 = new String(); //str3 先指向堆内存  str3= “abc”; //将字符串常量池的地址赋给str3 |

str1 != str2，str1 == str3;

str1.equals(str2); //true

str2.equals(str3); //true

str1.equals(str3); //true

**※**==比较的是引用地址；equals比较的是对象内容；

面试题：

1.String str = new String(“abc”);

String str = “abc”;

创建了几个对象？

答：2个；堆内存一个，字符串常量区一个

String str = new String(“abc”);

String str = “abcd”;

创建了几个对象？

答：3个；堆内存一个，字符串常量区两个

2.String str1 = “hello”;

String str2 = “world”;

String str3 = “helloworld”;

String str4 = str1+str2;

System.out.println(str3==str4); //false 地址不同

System.out.println(str3.equals(str4)); //true 内容相同

3.Java中equals();与==的区别？

==比较的是地址值；equals比较的是内容；

##### 2.String获取方法：

|  |
| --- |
| ①int [length](mk:@MSITStore:C:\Users\admin\Desktop\蓝桥笔记\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/String.html#length())(); 返回字符串长度。 |
| ②char charAt(int index); 返回指定索引的字符 |
| ③int indexOf(str1/int ch , [int i]); 返回指定字符的索引  从第i位开始，第一次出现str1的索引。（ch为char字符的int值） |
| ④int lastIndexOf(str1/int ch , [int i]); 返回指定字符的索引  从第i位开始，最后一次出现str1的索引。（ch为char字符的int值） |
| ⑤String substring(int start , [int end]); 字符串截取 [ ) |
| ⑥String concat(str1 ); 字符串尾接 |

##### 3.String判断方法：

|  |
| --- |
| ①boolean equalsIgnoreCase(str); 比较字符串内容是否相同（忽略大小写） |
| ②boolean contains(str); 判断是否包含指定字符串str |
| ③boolean startsWith(str); 判断是否指定前缀str |
| ④boolean endsWith(str); 判断是否指定后缀str |
| ⑤boolean isEmpty(); 判断字符串是否为空“”  str = “”; //字符串为空  str = null ; //空指针，非有效引用，不指向内存空间 |

##### 4.String转换方法：

|  |
| --- |
| ①byte[] getBytes(); 把字符串转换为字节数组 |
| ②char[] toCharArray(); 把字符串转换为字符数组 |
| ③static String valueOf(char[] chs); 把字符数组转成字符串 |
| ④static String valueOf(int i); 把int型数据转成字符串  注意：String类的valueOf静态方法可以把任意类型的数据转成字符串。 |
| ⑤String toLowerCase(); 把字符串转成小写 |
| ⑥String toUpperCase(); 把字符串转成大写 |

##### 5.String其他方法：

|  |
| --- |
| ①String replace(oldstr,newstr); newstr替换oldstr  String replace(oldchar,newchar); newchar替换oldchar |
| ②String trim(); 忽略前导空格和尾部空格 |
| ③int [compareTo](mk:@MSITStore:C:\Users\admin\Desktop\蓝桥笔记\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/String.html#compareTo(java.lang.String))(str1); 按字典顺序比较两个字符串 |
| ④int compareToIgnoreCase(str); 按字典顺序比较两个字符串（忽略大小写） |

#### （二）StringBuffer/StringBuilder

●StringBuffer/StringBuilder：可变字符序列，通过某些方法调用可以改变字符序列的长度和内容。

●StringBuffer本质：构造一个其中不带字符的字符串缓冲区。

●StringBuffer()默认容量为 16 个字符。

1.String和StringBuffer/StringBuilder

|  |  |
| --- | --- |
| String | 不可变的字符序列（空间浪费，资源利用率低） |
| StringBuffer/  StringBuilder | 可变的字符序列；通过某些方法调用可以改变字符序列的长度和内容。 |

2.StringBuffer和StringBuilder

|  |  |  |
| --- | --- | --- |
|  | StringBuffer | StringBuilder |
| 区别： | 线程安全，效率低 | 线程不安全，效率高 |
| 使用  场景： | a.数据共享，需要保证线程的安全性和一致性； | 1. 数据独享； 2. 在共享的时候，安全性要求不高； |
| 联系： | StringBuilder是StringBuffer的兼容API。 | |
| \*线程安全：两个线程共享一份数据时，为了保证数据的一致性，往往会在一个线程操作这份数据的时候，为当前数据加锁。 | | |

面试题：

1. [StringBuffer](mk:@MSITStore:C:\Users\admin\Desktop\蓝桥笔记\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/StringBuffer.html#StringBuffer())(); 构造一个其中不带字符的字符串缓冲区，其初始容量为 16 个字符。

2.StringBuffer和StringBuilder的区别？

##### ●StringBuffer常用方法

|  |
| --- |
| ①StringBuilder append(任意类型数据); 末端追加字符串 |
| ②StringBuilder insert(int offset , 任意类型数据); 插入字符串 |
| ③StringBuffer deleteCharAt(int index); 删除指定位置字符 |
| ④StringBuffer delete(int start, int end); 删除子字符串 |
| ⑤StringBuffer replace(int start,int end,str); 替换子字符串 |
| ⑥String substring(int start,[int end]); 字符串截取 |
| ⑦StringBuffer reverse(); 字符串反转（倒序） |

#### （三）正则表达式

1.常用的正则表达式：自行百度

2.正则表达式的使用：

|  |
| --- |
| String str = "lanqiao@126.com";  // Pattern：类是正则表达式的编译表示形式。  Pattern p =Pattern.compile("^\\w+([-+.]\\w+)\*@\\w+([-.]\\w+)\*\\.\\w+([-.]\\w+)\*$");  //Mathcer：类通过解释 Pattern 对字符序列执行匹配操作的引擎。  Matcher m = p.matcher(str);  //进行匹配的结果  boolean flag = m.matches(); |

### 三、基本数据类型的包装类

●为了对基本数据类型操作方便，引入基本数据类型的包装类。

byte -> Byte；short -> Short；int –> Integer；long -> Long

float -> Float；double -> Double；char -> Character；boolean - > Boolean

#### （一）Integer 整型

1.Integer常量

|  |  |
| --- | --- |
| ①public static final int MAX\_VALUE; | 最大值 |
| ②public static final int MIN\_VALUE; | 最小值 |

2.Integer进制转换方法

|  |
| --- |
| ①static String toBinaryString(int i); 十进制转换为二进制 |
| ②static String toOctalString(int i); 十进制转换为八进制 |
| ③static String toHexString(int i); 十进制转换为十六进制 |

3.int与String相互转换

|  |  |
| --- | --- |
| int转String： | ①Integer.toString(int i);  ②String.valueOf(int i); |
| String转int： | ①Integer.parseInt(String str);  ②先用构造方法初始化  [Integer](mk:@MSITStore:C:\Users\admin\Desktop\蓝桥笔记\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/Integer.html#Integer(java.lang.String)) integer = new [Integer](mk:@MSITStore:C:\Users\admin\Desktop\蓝桥笔记\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/Integer.html#Integer(java.lang.String))(str);  再用integer.intValue(); |

jdk1.5后 虚拟机为包装类提供了缓冲池，integer缓冲池的大小-128~127一个字节大小；

#### （二）自动装箱和自动拆箱

（JDK5新特性）

|  |  |
| --- | --- |
| 自动装箱： | 可以将基本数据类型直接赋值给对应包装类对象  Integer integer1 = 100; |
| 自动拆箱： | 可以将基本数据类型包装类对象直接赋值给对应基本数据类型变量  int a = integer1; |

### 四、Math数学运算

●Math 类包含用于执行基本数学运算的方法。

1.Math静态常量

|  |
| --- |
| ①public static final double PI = 3.14159265358979323846; |
| ②public static final double E = 2.7182818284590452354; |

2.random()随机数

|  |
| --- |
| static double random(); 随机产生0.0~1.0的double值 |

随机产生一个a~b的值：double Math.random()\*b+a;

3.round()四舍五入

|  |
| --- |
| static long round(double a); double浮点数转long整型 |
| static int round(float a); float浮点数转int整型 |

### 五、Random伪随机数

●Random(long seed); 以long种子创建随机数生成器

|  |
| --- |
| int nextInt() 返回一个int随机数[0-2^32）； |
| int nextInt(int n) 返回一个int伪随机数[0-n）； |
| int nextDouble() 返回一个int随机数[0.0-1.0）； |

### 六、System类

System类final修饰，不可被继承；

long System.currentTimeMillis()；返回1970-1-1 00:00:00至今的毫秒数

### 七、Date时间

#### （一）Date时间类

●1900年1月1日0时0分0秒00毫秒-->当前时间的毫秒值 ①时间点：1900年1月1日0时0分0秒00毫秒

②date精确到毫秒

#### （二）SimpleDateFormat时间格式化类

进行格式化format()（Date -> String）、解析parse()（文本 -> 日期）；

SimpleDateFormat(String pattern)；用给定的模式和默认语言环境的日期格式符号构造SimpleDateFormat。

**new** SimpleDateFormat("yyyy-MM-dd hh:mm:ss").format(**new** Date())

#### （三）Calendar日历

|  |
| --- |
| static Calendar getInstance(); 获取一个Calendar实例 |

抽象类，不可以被实例化（new）。

|  |
| --- |
| int get(int field); 返回给定日历字段的值。 |
| void set(int field, int value); 将给定的日历字段设置为给定值。 |
| void add(int field, int amount); 添加或减去指定的时间量。 |

月份MONTH（0~11）+1。

# 面向对象

Java是一门纯粹的面向对象的编程语言。

## 面向对象的概念

面向过程：结构化编程思想

面向对象：事物高度抽象化

面向对象的三大特性：封装，继承，多态。

类：具有相同特征，行为和属性的多个对象的抽象。

一组事物相关的属性和行为的集合

类的属性：事物描述信息，成员变量表示。

类的行为：事物功能信息，成员方法表示。

类：抽象的（模型），对事物的抽象描述，属性和行为。

对象：真实的（实例），类的具体实现。

面向对象：从抽象到具体

## 类与对象

### （一）类的定义

|  |
| --- |
| 修饰符 class 类名{  //定义属性：成员变量  变量修饰符 变量类型 变量名称；  //定义行为：成员方法  访问修饰符 返回值类型 方法名（形参列表）{  }  } |

成员变量与局部变量同名，方法内变量名访问的是局部变量；

类的构成：代码块，构造方法，成员变量，成员方法；

类内执行顺序：静态>构造，代码块>构造方法>成员方法；

静态>初始化>构造器>成员方法；

### （二）对象的创建与使用

1.对象的创建（类的实例化）：

①类名 对象名称 = new 类名()；

对象的内存分配：栈区，堆区，方法区（class区）

2.对象的使用：

①访问成员变量：对象引用.对象属性;

②访问成员方法：对象引用.对象方法(实参列表)；

3.匿名对象：

①创建时没有名称，没有变量存储引用地址；

②只能使用一次，栈内存中无匿名对象的引用地址；

### （三）类的封装

封装：对外隐藏实现细节，提供公共访问方法

①属性私有化：private成员属性（类内访问）；

②公开访问方法：public get()、set()；

### （四）标准JavaBean（类）

封装：属性私有化，对外公开get和set访问方法；

无参构造，代参构造，

重写toString( )；equals( )；hashCode( )；

## 构造方法（constructor）

1.构造方法的特点：

①构造方法的方法名与类名相同；

②构造方法无返回值，无法return；

2.构造方法的声明形式：

|  |
| --- |
| 访问修饰符 类名（形参列表）{ } |

3.构造方法的在何时使用：

创建对象时由JVM自动调用；

4.构造方法的重载：

①对于未显式定义构造方法的类，系统会为类自动创建一个默认的无参构造方法；

②对于已显式定义构造方法的类，系统不再提供默认无参构造方法；所以，最好再定义一个无参构造方法。

5.构造方法的作用：

①创建对象

②对象初始化

## 代码块

|  |  |
| --- | --- |
| 代码块作用：变量初始化； | |
| ①普通代码块： | 在方法或语句中的{ }；先出现先执行； |
| ②构造代码块： | 对成员变量初始化；随对象创建而执行； |
| ③静态代码块： | 对静态变量初始化；随类加载而执行； |

## 对象初始化

|  |  |
| --- | --- |
| 创建对象时： | ①使用构造代码块初始化； |
| ②使用构造方法初始化； |
| 创建对象后： | ③调用set()初始化； |

## this关键字

|  |  |
| --- | --- |
|  | this代表本类当前对象； |
| this  作用： | ①区分成员变量和局部变量：this.成员变量 = 局部变量 |
| ②调用成员方法：this.成员方法 |
| ③在构造方法中调用本类的其他构造方法：this(形参列表)  （this( )必须在构造方法的第一行） |

## static关键字

static：创建独立于具体对象的域变量与方法；

static可以修饰成员变量、方法、代码块、内部类；

static特点：

①随类加载而加载；

②

### 静态变量

### 静态方法

### （三）静态代码块

static使用：

为特定域分配单一存储空间。

变量或方法不与任何对象关联，该类所有对象共享。

static随类加载而加载，new之前已存在；

静态无法访问非静态；

static特点：

1. 类相关，与对象无关；

静态变量：类变量；静态方法：类方法；

1. 调用方式：

直接调用：类名.属性; / 类名.方法();

不需要创建对象，可直接使用；

3.存储空间：存储于方法区中静态区；

4.new之前执行

static 减少代码开销

使用static修饰的变量是属于当前类所有，所有对象所共有的

使用static修饰的变量是随着类的加载而加载，成员变量是随着对象的创建而创建。

静态方法注意事项：

1.静态方法中，只能调用静态变量，不能使用非静态变量

2.静态方法只能调用静态方法，不能调用非静态的方法

3.非静态可以访问静态的，而静态的不能访问非静态的

4.静态方法中不能使用this

静态方法一般常用于工具类的方法

思想：对于时间date进行格式化，工具类

对静态变量初始化：

1.声明变量时，

2.静态代码块，

静态代码块（执行一次）>构造代码块（随对象创建而执行）>无参构造方法（）>构造代码块>带参构造方法

执行顺序：

①静态（类加载执行）>非静态（new对象执行），

②代码块>构造方法>成员

调用静态变量：

直接调用：变量名

类名调用：类名.变量名

静态变量，静态方法，静态代码块

类加载

1. 加载main方法所在类，到方法区的class区；
2. 从main方法开始依次顺序执行，
3. 当使用到某一个类时，加载该类；
4. 加载类时扫描类中信息，为静态成员分配空间；
5. 类结束，释放静态成员；

面试题：

读程序，写结果；

静态代码块（执行一次）>构造代码块（随对象创建而执行）>无参构造方法（）>构造代码块（随对象创建而执行）>带参构造方法

静态成员与非静态成员区别：

|  |  |  |
| --- | --- | --- |
|  | 静态成员 | 非静态成员 |
| 1.内存分配： | 随类加载而加载 | 随对象产生而产生 |
| 2.调用方式： | 类名调用 | 对象调用 |
| 3.生命周期： | 与类相关 | 与对象相关 |
| 4.共享方式： | 全类共享 | 当前对象独享 |

静态优点：

1.一个静态属性在堆内存只能存在一个，即大家可以共享该属性值。

2.如果一个对象数组中其中的某个属性大家都是相同的，即可以使用静态，这样可以节省内存空间。

3.静态一旦创建，生命周期就会从创建开始到该程序结束才会死亡。

4.静态依赖于类，而不是对象。

Main方法：

public：访问权限足够大

static：对象产生之前，就要调用；

## 单例模式（Singleton）

单例模式：使类始终保持只有一个实例。

①静态创建对象

②私有化构造方法

③公开提供静态获取实例方法

1.懒汉式：

若对象引用==null：new一个对象；

若对象引用！=null：直接返回已有对象地址；

|  |
| --- |
| **public** **class** Singletonif {  // 静态实例化  **private** **static** Singletonif *s* = **null**;  // 私有化构造方法  **private** Singletonif() {  }  // 公开提供静态获取实例的方法  **public** **static** Singletonif getInstance() {  **if** (*s* == **null**) {  *s* = **new** Singletonif();  }  **return** *s*;  }  } |

2.饿汉式：

|  |
| --- |
| **public** **class** Singletonfinal {  // 静态 final 实例化  **private** **static** **final** Singletonfinal ***s*** = **new** Singletonfinal();  // 私有化构造方法  **private** Singletonfinal() {  }  // 公开提供静态获取实例的方法  **public** **static** Singletonfinal getInstance() {  **return** ***s***;  }  } |

## 继承（extends）

父类/超类

子类/派生类：父类的拓展

Java继承的特点：

Java继承都是单继承，不允许多继承，但可以多层继承。

单继承：一个类只能有一个父类；但是一个父类可以有多个子类

子类可以继承父类那些成员：

1.非私有（private）成员属性；

区分子类和父类的同名的成员属性：

super：代表父类当前对象；

2.构造方法：

不能继承，可以调用

super() 调用父类构造方法；（必须出现在第一句）

3.构造代码块：

父类构造代码块-->父类构造方法-->子类构造代码块-->子类构造方法

4.静态代码块

静态代码块和静态方法在类加载时即执行；

5.成员方法

继承非私有方法；

先调用子类方法，后调用父类方法，

### 方法的重写（Override）

重写：子类重写父类的方法；

重写条件：

1.继承关系

2.方法声明形式完全相同

3.访问修饰符：子类重写方法不能比父类方法严格

重写作用：

方法的扩展或重新实现；

6.静态方法

静态方法形式上的继承，本质不是继承；

静态方法不能重写；

面试题：

父类{静态代码块，构造代码块，构造方法}

子类{静态代码块，构造代码块，构造方法}

创建子类对象：①静态>非静态，②代码块>构造方法>成员

new时，先调用父类构造方法；

继承是两个类

继承的好处

1.提高了代码的复用性；

多个类相同的成员可以放到同一个类中

2.提高了代码的维护性；

如果功能的代码需要修改，修改一处即可

3.使类与类之间产生了关系，继承是多态的前提；

其实这也是继承的一个弊端：类的耦合性很强

子类是父类的扩展，不是简单的复制

super关键字

### final关键字

|  |
| --- |
| final：最终修饰符，终态，不可改变； |
| final类：不能被继承（太监类）； |
| final方法：不能被重写； |
| final变量：只能赋值一次； |
| final引用类型：地址不能被改变； |

类与类之间的关系：

继承，

依赖：一个类作为另一个类的参数出现

类之间存在着整体和部分的关系

聚合：另一个类的存在与否不会影响到当前类的完整性

组合：另一个类的存在与否会直接影响到当前类的完整性

组合模式：树

## 多态

多态：

父类的引用指向子类的对象

多态的条件：

类之间存在继承关系；

必须有方法的重写；

A a = new B();

在A类中检测，在B类中调用；

类型转换：

向上转型：自动转换，父类的引用指向子类的对象

向下转型：强制转换，还原为子类；

（向下转型非原子类会失败）

类型判断：

boolean 对象instanceof类：判断该对象是否属于目标类；

多态访问类的成员：

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

静态变量/方法：编译看左边，运行看左边

## 抽象（abstract）

### 抽象类

意义：被子类继承（extends）；

①不能被实例化；用来被子类继承

②存在构造方法；子类可以访问抽象成员

③不能被final修饰；

### 抽象方法

意义：被非抽象子类重写（@Override）；

①只提供方法声明，不进行方法实现；

②不能被private，static，final修饰；

③只存在于抽象类中（抽象类中不一定存在抽象方法）

非抽象子类必须重写抽象父类中所有抽象方法

抽象类不一定有抽象方法，存在抽象方法的类一定是抽象类；

面试题：

1.抽象方法能用private修饰吗？

不能，private方法不能被重写；

2.抽象方法抽象类能用final修饰吗？

不能，final方法不能被重写，final类不能被继承

3.抽象方法能用static修饰吗？

不能，static方法不能被重写。

## 模板设计模式

由抽象衍生的设计模式；

①把通用的功能写成final的；

②把变化的功能写成abstract的；便于子类重写。

## 接口（interface）

接口（interface）：一种标准和规范；（类似多继承）

接口的意义：被实现（implements）；

接口的组成：

接口常量：默认public static final修饰变量；

接口中只存在静态常量；

接口方法：只能用abstract，public，default，static修饰方法

接口中只存在抽象方法；

接口的使用：

接口名称 引用变量 = new 实现类（）；

多态的体现：接口的引用指向实现类的对象；

接口的特点：

①接口不能实例化；

②接口没有构造方法；

③多继承，多实现；

接口可以继承接口；接口的继承是多继承；

接口的实现类可以多实现

接口和类区别：

1.接口不能实例化；

2.接口中没有构造方法

类与类：继承；单继承，可多层继承；

类与接口：实现；多实现；

接口与接口：继承；多继承；

抽象类：“is a”通用

接口：“like a”扩展

接口与抽象类：

联系：两者都不能被实例化

区别：①抽象类存在构造方法；接口中不存在构造方法；

②抽象类是用来继承的；接口是用来实现的；

## 内部类

内部类：定义在类内部的类；

内部类的意义：服务于外部类；

|  |  |  |
| --- | --- | --- |
|  | 成员内部类 | 方法内部类 |
| ①声明位置： | 类内部，方法外部 | 方法内部 |
| ②作用域： | 类内 | 方法内 |
| ③实例化： | 通过外部类对象访问内部类  外部类.内部类 对象 = new 外部类().new 内部类(); | 方法内实例化 |
| ④class文件： | 外部类.class  外部类$内部类.class | 外部类.class  外部类$1内部类class |
| ⑤类的构成： | 与普通类相同 | |
| ⑥访问限制： | 可直接访问外部类任意成员；  一般使用private修饰，服务于外部类； | 只能使用方法的final局部变量；  jdk8会自动为局部变量添加final修饰； |

静态内部类：

面向对象（接口/服务）编程：

实体：entity；

数据访问层：DAO

业务逻辑层：Service

## 包

对类进行分包管理（文件夹）；

包的命名规则：公司域名的反写+模块名称

包的关键字：package

全类名：包名.类名

Jar包：

1. 使用eclipse的export功能
2. 第三方工具：fatjar
3. Jar文件运行：java -jar 类名.jar

导包：

Java 中的类是分包管理的，你要使用的类和当前类不在同一包下，需要导包。

import 包名.类名；

import 包名.\*；//通配符\*，将包下所有类都导入，慎用。

在Java中lang包不用导包；（language语言包）

访问控制：

public，protected，默认，private，

Java五大包

Java.lang语言包

Java.util多用途工具包

Java.io输入输出流

Java.net网络

Java.sql数据库

# Java高级应用

## 数据结构

（一）基本概念：

|  |
| --- |
| ①数据：描述客观事物的符号； 数值型数据/非数值型数据 |
| ②数据元素：数据的基本单位，（结点/记录）； |
| ③数据项：有独立含义的最小单位；（属性/字段/域）； |
| 数据-->数据元素-->数据项（field） |

（二）数据结构：数据之间的关系；

|  |  |  |
| --- | --- | --- |
| 逻辑结构： | 线性： | 线性：一对一； 线性表，栈，队列，串 |
| 非线性： | 集合： |
| 树：一对多； |
| 图：多对多； |
| 存储结构： | 顺序存储结构：（数组）元素物理地址连续 | |
| 链式存储结构：（指针）元素指针指向下个元素地址 | |

（三）数据运算：

插入，删除，修改，查找，排序

### 线性表

初始化，取值，查找，插入，删除；

顺序表的优缺点：

优点：元素存储密度大；

可随机存取任一元素；

缺点：不易于进行插入删除操作

浪费存储空间

静态存储，无法自动扩充；

单链表：存储的物理地址不一定相邻；

双向链表：

循环链表：

栈：先进后出

队列：先进先出

树：

二叉树：

图：

## 集合与泛型

### 集合概述

集合的产生：为保存数目不确定的对象，解决数组长度固定的问题；

集合类：可以存储任意类型的对象，并且长度可变的类；

集合的分类：

按存储结构分类：

Collection 单列集合

List 有序（插入顺序），集合元素可重复的集合

实现：ArrayList，Vector，LinkedList；

Set 无序（插入顺序），集合元素不可重复的集合

实现：HashSet，TreeSet；

自然排序：Comparable接口的排序规则；

Map 双列集合，具有键值映射关系

实现：HashTable，HashMap，

LinkedHashMap，TreeMap，

集合的特点：

①只能存储引用类型；

②长度可变，不受限制；

数组与集合的区别：

数组：长度固定，可存储基本数据类型和引用类型；

获取长度：数组length属性

遍历：for循环

集合：长度可变，只能存储引用类型；

获取长度：集合size()；

遍历：for循环；迭代器Iterator遍历；增强for循环

### Collection接口

子接口：List，Set，

已知实现类：ArrayList，Vector，LinkedList，HashSet，TreeSet

Collecting接口方法：

#### Iterator迭代器接口

迭代器Iterator：遍历Collection中的元素；

boolean hasNext( ); 判断是否有下一个元素；

（指针指向输出元素的下一个）；

实际使用中，遍历常用while(hasNext()){ }；

next(); 移动指针指向下一个元素，并返回元素的值；

Listiterator：

正向遍历，反向遍历

修改的是迭代器中的集合

注意事项：

不能在迭代的同时，改变集合的结构，不能向集合中插入删除元素。否则会出现ConcurrentModificationException异常；

解决方案：

①使用迭代器进行迭代，并使用迭代器（ListIterator）修改；

②使用for循环、增强for循环进行遍历；

增强for循环：遍历数组与集合；

for(元素类型 变量：容器对象){ }

#### List接口

List接口：有序（插入顺序），元素可重复的Collection；

已知实现类：ArrayList，Vector，LinkedList；

集合遍历修改：

当集合使用迭代器迭代的时候，不允许向原集合添加元素，但可以原集合。

如何解决：

1.使用迭代器get( )元素添加。

2.不使用迭代器，使用for循环遍历。

List常用实现类：

ArrayList：

①基于长度可变的数组实现；

②允许多个null值存在；

③不同步（线程不安全）

数组：元素个数确定的情况下；

ArrayList：元素个数不确定的情况下；

Vector：（已过时）

①基于长度可变的数组实现

②允许多个null值存在；

③同步的（线程安全的）

不常用；

ArrayList与Vector的区别：是否同步；

LinkedList：

①基于双向链表实现

②允许多个null值存在

③不同步（线程不安全）

实现了链表，栈，队列等的操作。

数组：元素个数确定的情况下；

ArrayList：元素个数不确定的情况下；

List在实际开发中的应用：

1.如何选择List接口不同实现？

ArrayList：数组实现（取值查询效率高），不同步

Vector：数组实现（取值查询效率高），同步，不常用；

LinkedList：链表实现（插入删除效率高），

2.使用List接口的多态还是使用具体的对象？

在实际开发中，使用List接口的多态更多；

3.如果不考虑性能问题，三个实现那个使用更多？

在实际开发中，一般使用List list = new ArrayList；

#### Set接口

数学意义上的集合：互异性，无序性，确定性；

Set接口：无序，元素不重复的Collection；

重复判断依据：s1.equals(s2) == true；

当add存入时，即进行重复判断；

Set典型实现类：HashSet，TreeSet；

HashSet：

①基于HashMap实现，使用的是HashMap中的key；

不保证Set迭代顺序恒久不变；

②允许一个null值存在；

③线程不安全；

TreeSet：

①基于TreeMap实现，使用的是TreeMap中的key；

默认使用Comparable进行自然排序

②可进行自然排序、定制排序；

③线程不安全

自然排序：

由元素类型决定排序方式，使用内部比较器：实现Comparable接口，实现compareTo方法；

已实现Comparable接口，重写comareTo方法：

①基本数据类型的包装类：

==：return 0；

>：return 正整数；

<：return 负整数；

②字符串比较：按字典顺序（Unicode）；

定制排序：

由容器决定排序方式，使用容器比较器：创建一个比较器，实现Comparator接口，实现compare方法；

比较器的创建：

①自定义外部类

②写一个内部类

③匿名内部类

### Map接口

Map接口：双列，无序（存取顺序），以键值对形式存储；

实现：HashTable，HashMap，TreeMap

AbstractMap --> HashMap --> LinkedHashMap

SortedMap --> TreeMap

HashTable

①以键值对形式存储

②Key不允许重复

③键值映射关系

Map与Collection的区别：

MapEntry：一对键值对<key，value>；

#### HashMap☆

基于HashTable存储；

基于哈希表、Hash桶数组是实现；

Hash桶数组结构：数组+链表+红黑树；

数组元素存储 一个链表/红黑树 的hashCode；

提高遍历效率：JDK8后引入红黑树；

链表长度>8，转化为红黑树；

HashMap：线程不安全，不支持并发操作；

ConcurrentHashMap：线程安全，支持并发操作；

1.put操作：

流程图：

①判断数组是否为空；

②key是否存在；key存在，直接覆盖value；

1. 扩容机制：

扩容：16\*0.75=12时扩容；

扩容极其耗性能；

扩容优化：估算，初始化容量；

HashMap内部实现：

①

②

③

④

HashMap性能参数：

面试题：

ArrayList的扩容实现和HashMap的扩容的实现的对比？

fail-fast机制：不一定触发；

ConcurrentModificationException；

HashMap最重要，最常用的Map

#### TreeMap

基于红黑树实现；

TreeMap：对key进行自然排序的<key，value>集合；

数据结构：红黑树；查询速率高；

不允许null键，允许null值；

一般在对key排序时，使用TreeMap；

#### HashTable

基于哈希表实现；线程安全的；不允许null键null值；

古老，过时，不常使用；

面试题：

**HashTable与HashMap的区别：**

1. HashTable不支持null-key和null-value；HashMap支持null-key，null-value
2. HashTable默认的初始大小为11，之后每次扩充为原来的2n+1。HashMap默认的初始化大小为16，之后每次扩充为原来的2倍。在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从hash计算的效率上，又是HashMap更胜一筹。
3. HashTable 线程安全（同步）；HashMap 线程不安全（不同步）。HashTable已经被淘汰了，如果你不需要线程安全，使用HashMap；如果你需要线程安全，使用ConcurrentHashMap；

**HashTable与HashMap的区别：**

①Hashtable 是一个古老的 Map 实现类，不建议使用

②Hashtable 是一个线程安全的 Map 实现，但 HashMap 是线程不安全的。

③Hashtable 不允许使用 null 作为 key 和 value，而 HashMap 可以

④与 HashSet 集合不能保证元素的顺序一样，Hashtable 、HashMap 也不能保证其中 key-value 对的顺序

⑤Hashtable 、HashMap 判断两个 Key 相等的标准是：两个 Key 通过 equals 方法返回 true，hashCode 值也相等。

⑥Hashtable 、HashMap 判断两个 Value相等的标准是：两个 Value 通过 equals 方法返回 true

### Collections工具类

### 泛型

#### 泛型

为什么使用泛型：集合元素取出时，自动转化成Object类型；

泛型：不确定集合元素类型时，集合元素类型的参数化；

<参数化类型>，

常用泛型形式：<E>，<T>，<K>，<V>

JDK7泛型的新特点：后面的<>可省略；

List<参数化类型> list = new ArrayList<~~参数化类型~~>();

集合若不定义泛型，默认Object类型；

泛型擦除：泛型只在编译阶段有效，运行阶段不存在了；

泛型类：集合元素类型是泛型；

泛型接口：访问修饰符 interface 接口名<T>{ }

泛型方法：访问修饰符 <T> T 方法名称 （参数列表）{ }

泛型通配符：

<? extends T> 下限，只能使用T和T的子类；

<? super T> 上限，只能使用T和T的父类；

<?> 未知类型，默认Object类型；

#### 可变参数

可变参数：在定义方法时，不确定传入参数的个数

方法形参：（参数类型 ... 容器）；

可变参数的用法：跟数组相同，位于参数列表的末尾；

JDK5新特性：

自动装箱和自动拆箱，增强for循环，

泛型与可变参数，Annotion

JDK7新特性：

①Switch引入String

②泛型后面可以不写<>

③平级异常类型可或

## 异常

### 异常概述

异常：程序编译和运行阶段，发生的不正常现象，称为异常。

已知异常：

①数组下标越界：ArrayIndexOutOfBoundsException

②空指针异常：NullPointerException

③类型转换异常：ClassCastException

④不正当修改异常：ConcurrentModificationException

⑤解析异常：ParseException

### 异常分类

Throwable

Error：指系统发生错误，无法处理

①JVM崩溃VirtualMachineError

②内存溢出OutOfMemoryError

Exception：

编译期异常：Checked Exception

ParseException

运行时异常：Runtime Exception

（Unchecked Exception）

java.lang.IndexOutOfBoundsException

java.lang.NullPointerException

java.lang.ClassCastException

java.lang.ArithmeticException

异常区分：

编译期异常：Exception的子类

运行时异常：Exception-->RuntimeException的子类

### 异常处理

#### 基本异常处理

（一）默认处理：JVM自动处理，输出异常，结束程序运行。

（二）捕获、抛出机制

一般情况下，我们需要对异常做出相应的处理（抓抛模型）；

1.抛出异常：throws 对应异常类型；

方法声明处抛出；

抛给调用者，调用者可以选择 处理/抛出，直至抛给JVM默认处理；

2.捕获异常：

try{

可能发生异常的代码；

}catch( 异常类型 相应类型变量 ){

异常处理；（异常发生时执行）

}finally{

必然执行；（无论异常是否发生，必然执行）

}

try{}catch(){}...

try{}catch(){}finally{}

try{}catch(){}...catch(){}finally{}

try{}finally{}

JDK7新特性：平级异常类型可并列或

try{}catch(异常类型 | 异常类型 | 异常类型 | 变量){}

不能越级并列（子类与父类不可并列）

常与instanceof连用。

#### 多异常处理

①多catch连用

②catch异常类型或并列

低级异常处于catch上层

高级异常处于catch下层

### finally语句

①程序连接数据库时，无论访问是否成功，都应该将链接释放

②在I/O操作中，最终都需要将I/O流关闭；

面试题

try块中有return语句，finally语句还会执行吗？在什么时候执行？

异常未发生，finally一定会执行；finally先于return执行；

异常发生，finally一定会执行，return不执行；

异常发生后，try块内异常之后的语句不执行；

获取异常信息：

void printStackTrace(); 打印异常堆栈信息；

getMessage(); 返回异常信息字符串；

主动抛出异常：

方法内部，throw new 异常类( )；

①new Exception(异常信息); 主动抛出编译期异常

②new RuntimeException(异常信息); 主动抛出运行时异常

throws与throw的区别：

①声明位置：

throws方法声明处抛出；

throw方法内抛出；

②抛出目标：

throws抛出异常类型；

throw抛出异常对象；

③抛出异常数量：

throws抛出多个异常；

throw抛出一个异常；

④异常发生：

throws可能发生异常，不一定发生；

throw一定抛出一个异常对象；

异常处理策略：

编译期异常：必须处理，否则程序会报错，无法执行；

运行时异常：可以不处理，可通过代码逻辑修改避免；

处理策略：

如果清楚异常如何处理，自己处理，无需暴露；

如果不清楚异常如何处理，不作处理，抛给方法调用者，告知异常的存在；

### 自定义异常类

自定义编译期异常：

自定义编译期异常类 extends Exception

无参构造：super();

有参构造：super(str);

|  |
| --- |
| **public** **class** MyException **extends** Exception {  **public** MyException() {  **super**();  }  **public** MyException(String str) {  // 异常处理信息  **super**(str);  }  } |

自定义运行时异常：

自定义异常类 extends RuntimeException

无参构造：super();

有参构造：super(str);

|  |
| --- |
| **public** **class** MyRuntimeException **extends** RuntimeException {  **public** MyRuntimeException() {  **super**();  }  **public** MyRuntimeException(String str) {  // 异常处理信息  **super**(str);  }  } |

## IO与NIO

IO：输入输出（input/output） java.io包

### File类

File类：文件/目录的路径名的抽象表示形式；与OS无关；

路径表示：

|  |
| --- |
| 绝对路径：相对于根目录，从根目录到文件/目录的表示 |
| 相对路径：相对于当前位置，当前工程下的文件/目录 |

●规避不同OS的路径分隔符：File.separator；

#### （一）File类构造方法

|  |
| --- |
| public File(String pathname)； |
| public File(String parent,String child)； |
| public File(File parent,String child)； |

#### （二）File类常用方法

##### 1.创建功能

|  |
| --- |
| public boolean createNewFile()； |
| public boolean mkdir()； |
| public boolean mkdirs()； |

2.删除功能

|  |
| --- |
| public boolean delete() |
| 可删除文件、空目录； （删除目录必须保证目录为空） |

##### 3.重命名功能（另存为）

|  |
| --- |
| public boolean renameTo(File dest)  将当前文件重命名并剪切到目标目录中 |

##### 4.判断功能

|  |
| --- |
| public boolean isDirectory() |
| public boolean isFile() |
| public boolean exists() |
| public boolean canRead() |
| public boolean canWrite() |
| public boolean isHidden() |

##### 5.基本获取功能

|  |
| --- |
| public String getAbsolutePath() |
| public String getPath() |
| public String getName() |
| public long length() |
| public long lastModified() |

##### 6.高级获取功能

|  |
| --- |
| **public** String[] list(); |
| **public** File[] listFiles(); |

### IO流

#### 一、IO流概述

在Java中，对于数据的输入输出操作都是以“流”的方式进行。

流是程序与文件之间数据交换的抽象；

●输入：将数据从磁盘或外界媒质 读入到程序中时；

●输出：将数据从程序中写到外部存储设备时；

#### 二、IO流分类

①按流的方向分：输入流，输出流；

②按文件组成单元分：字节流，字符流；

a.字节文件：文本、图片、音频、视频；

b.字符文件：纯文本文档（.txt）；

③按照流是否直接与特定的地方(如磁盘、内存、设备等)相连分：

a.节点流：可以从或向一个特定的地方（节点）读写数据；例如：InputStream；OutputStream；Reader；Writer；

b.处理流：是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。例如：BufferedInputStrean；

|  |  |  |
| --- | --- | --- |
| java.io | 输入流 | 输出流 |
| 字节流 | InputStream | OutputStream |
| 字符流 | Reader | Writer |

#### 三、字节流

字节流：二进制数据；文本、图片、音频、视频

适用于操作图片、音频、视频；

##### 1.字节输入流InputStream

字节输入流：abstract InputStream

典型实现：FileInputStream(file);

|  |
| --- |
| ①int read( ); //读取一个字节；返回读取的值，-1结束； |
| ②int read(byte[] b); //读入字节数组，返回读取字节数； |
| ③int read(byte[] b,int off,int len);  //将字节读入数组某一部分；返回读取字节数； |
| ●void close()；//关闭流，并释放系统资源 |

##### 2.字节输出流OutputStream

字节输出流：abstract OutputStream

典型实现：FileOutputStream(file,[append]);

参数append=true，追加写append；

|  |
| --- |
| ①void write(int b); //写入一个字节 |
| ②void write(byte[] b); //写入一个字节数组（byte[] 缓冲数组） |
| ③void write(byte[] b,int off,int len); //写入数组的某一部分 |
| ●void flush()；//刷新缓冲，强制写出所有缓冲区数据； |
| ●void close()；//关闭流； |

读取文件时，必须要有文件；

写入文件时，文件可以不存在；创建流对象时新建了文件；

为什么要关闭close()；

IO流操作是资源操作，必须关闭；

①通知系统，该流对象变成垃圾；

②通知垃圾回收期来回收垃圾，释放资源；

创建流对象时，系统做了什么？

1. 调用系统资源，创建目标文件（文件不存在）；
2. 链接到目标文件；

#### 四、字符流

字符流=字节流+字符集； 纯文本文档（.txt）；

适用于操作多国语言（中文）的纯文本文档；

●存在缓冲区关闭流之前，必须先flush()；

##### 1.字符输入流Reader

字符输入流：abstract Reader；

典型实现：转换流InputStreamReader(InputStream is)

|  |
| --- |
| int read(); //读取单个字符；返回读取字符的int值，null结束 |
| int read(char[] c); //将字符读入数组；返回读取字符数； |
| int read(char[] c,int off,int len);  //将字符读入数组的某一部分；返回读取字符数； |
| void close(); //关闭流，并释放系统资源 |

##### 2.字符输出流Writer

字符输出流：Writer；

典型实现：转换流OutputStreamWriter(OutputStream os)

|  |
| --- |
| ①void write(int c); //写入单个字符 |
| ②void write(char[] b); //写入字符数组（char[] 缓冲数组） |
| ③void write(char[] b,int off,int len); //写入字符数组某一部分； |
| ●void flush()；//刷新缓冲，强制写出所有缓冲区数据； |
| ●void close()；//关闭流时，会自动执行flush()； |

字符流=字节流+字符集（编码和解码）；

编码解码必须一致，否则中文乱码；

字符输入转换流InputStreamReader

字符输出转换流OutputStreamWriter

将字节流通过字节流进行编码和解码，转换为字符流；

字节流与字符流区别：

（1）操作单位不同：

字节流是按字节读写（文本图片音频视频）；

字符流是按字符读写（纯文本文档）；

（2）操作对象不同：

字节流读写任意文件；

字符流只能读写纯文本文档.txt；

（3）使用不同：

字节是文件的最小单元；原子级操作；

字符是2个字节，组合时按照字符集编码进行；

流操作文件的一般步骤：

①创建读的目标文件：File

②创建输入流：InputStream

③创建写的目标文件：File

④创建输出流：OutputStream

⑤创建缓冲数组：byte buf = new byte[1024]

⑥while循环边读边写

while((len = is.read()) != -1){

os.write(buf,0,len) ;

}

⑦关闭流close( );

#### 五、缓冲流

##### 1.字节缓冲流

①字节缓冲输入流：BufferedInputStream（被装饰对象）

②字节缓冲输出流：BufferedOutputStream（被装饰对象）

##### 2.字符缓冲流

①字符缓冲输入流BufferedReader（被装饰对象）

|  |
| --- |
| String readLine(); //读取一行字符（不包括换行符）； |

②字符缓冲输出流BufferedWriter（被装饰对象）

|  |
| --- |
| void newLine(); //写入一个跨平台换行符； |

特点：自带一个默认缓冲区byte[8192]；

使用缓冲输出流，必须强制刷新或者关闭；

|  |
| --- |
| ●void flush()；//刷新缓冲，强制写出所有缓冲区数据； |
| ●void close()；  //存在缓冲区关闭流之前，必须先flush()；  //关闭处理流时，只需关闭最外层即可（装饰设计模式）； |

#### 六、装饰设计模式（Decorator）

装饰设计模式：对类进行功能扩展；

①成员变量：被装饰对象

②构造方法：装饰者要对被装饰对象初始化；

③方法扩展：被装饰类的引用.被扩展的方法；

④规范约束：抽象类，接口；

IO中大量使用了装饰设计模式

例如：缓冲流BufferedInputStream

|  |
| --- |
| 问题：继承代码扩展功能的弊端：  1.增强了代码的耦合度  2.单继承限制  解决：装饰设计模式； |

FileReader

Filereader是InputStreamReader的简化类

### 其他处理流

处理流：对节点流进行处理（缓冲流）；

#### 一、缓冲流

在使用处理流时，进行关闭操作时，只需关闭处理流即可；

因为装饰设计模式中，外层

#### 转换流

#### 三、管道流

作用：多用于线程间的通讯；

PipedInputStream

PipedOutputStream([pis])；可直接建立链接；

建立管道链接：pos.connect(pis) ;

（自带内部缓冲区）

#### 四、字节数组流

作用：将字节数组转换成IO流

ByteArrayInputStream

ByteArrayOutputStream

关闭无效

size()返回当前缓冲区的有效字节长度；

#### 五、字符数组流

作用：将字符数组转换成IO流

CharArrayReader

CharArrayWriter

#### 六、合并流

作用：合并2个输入流

SequenceInputStream(InputStream s1,InputStream s2)；

只有合并输入流，没有合并输出流；

#### 七、重定向流

作用：重定向标准输入输出流

1.标准输入输出流

|  |
| --- |
| System.in从键盘输入  System.out：输出到屏幕；  System.err：输出到屏幕； |

2.System下的静态方法

|  |
| --- |
| static void setIn(InputStream in);  static void setOut(PrintStream ps);  static void setErr(PrintStream ps); |

#### 八、随机访问文件流

应用：断点下载，断点续传；

RandomAccessFile

1.参数mode：打开文件的访问模式

“r”只读模式

“rw”读写模式

“rws”

“rwd”

2.存在指针，指向文件进行读写的位置；

getPointFile(); //获取当前指针的位置

通过指针，按字节顺序读取；

3.读取默认ISO-8859-1编码方式；需进行解码再重新编码；

#### 九、对象流

应用：对象的持久化；

ObjectInputStream反序列化

ObjectOutputStream序列化

对象流读写对象必须进行序列化Serializable；

序列化：将一个对象转换成一串二进制的字节数组；

反序列化：将字节数组重新构造成对象；

序列化目的：为了对象传输，保证传输对象的一致性；

如何序列化：使需要序列化的类实现Serializable

#### 十、数据流

DataInputStream

DataOutputStream

#### 十一、打印流

PrintStream

|  |
| --- |
| print(x); //只能有参输出；  printf(); //格式化输出；  println(); //可无参输出； |

IO异常：编译期异常

FileNotFoundExcetion

EOFException

IOException

### NIO流

NIO面向缓冲区的，基于通道的IO操作；（JDK1.4已产生）

|  |
| --- |
| 缓冲区（Buffer）：存储数据； |
| 通道（Channel）：传输数据； |

NIO与IO的区别：

|  |
| --- |
| IO：基于流；阻塞式（每次只能操作一种流）； |
| NIO：面向缓冲区，基于通道，选择器；非阻塞式； |

NIO将以更加高效的方式进行文件的读写操作；

#### 缓冲区（Buffer）

缓冲区：一个用于特定基本类型数据的容器；

缓冲区作用：保存数据；进行数据读写；

Buffer常见实现：

|  |
| --- |
| ByteBuffer，ShortBuffer，IntBuffer，LongBuffer，  FloatBuffer，DoubleBuffer，CharBuffer；无布尔型缓冲区。 |

①建立缓冲区，分配容量：

ByteBuffer b = ByteBuffer.[allocate](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/nio/../../java/nio/ByteBuffer.html#allocate(int))(1024);

清空缓冲区：Clear()

②缓冲区属性

位置：position()；

限制：limit()；

容量：capacity()；

③读/写：get()；put()；

④ 读写模式切换：filp();

⑤标记：mark()；reset()；

非直接缓冲区：[allocate](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/nio/../../java/nio/ByteBuffer.html#allocate(int))(capacity)；

传输方式：拷贝方式；

内存位置：位于堆区；

特点：占用资源较少，容易被释放；But效率低；

直接缓冲区：[allocate](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/nio/../../java/nio/ByteBuffer.html#allocate(int))Direct(capacity)；

传输方式：内存映射；

内存位置：直接位于内存页；

特点：效率高；But分配资源消耗大，不易被回收；

应用：一般分配给易受基础系统的本机IO操作的大型；

#### 通道（Channel）

通道作用：传输数据；

1.Java 为 Channel 接口提供的最主要实现类如下：

•FileChannel：用于读取、写入、映射和操作文件的通道。

•DatagramChannel：通过 UDP 读写网络中的数据通道。

•SocketChannel：通过 TCP 读写网络中的数据。

•ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。

2.获取通道：

本地IO：调用getChannel()；

FileInputStream/FileOutPutStream

RandomAccessFile

网络IO：

Socket

ServerSocket

DatagramSocket

获取通道的其他方式是：

Files类静态方法：newByteChannel() 获取字节通道（JDK1.7）

Channel类静态方法：open(Path path,OpenOpertion ... oo)（JDK1.7）

获取通道：

①本地IO获取通道：本地IO.getChannel()；

②打开通道FileChannle.open(Path path,OpenOpertion ... oo)

使用通道进行数据传输：

①使用通道+非直接缓冲区完成文件复制；

②使用直接缓冲区完成文件复制（内存映射文件）

③直接使用通道完成数据传输；

分散读取和聚集写入

①分散读取：read(ByteBuffer[] bufs)；

②聚集写入：write(Bytebuffer buf1)；

|  |
| --- |
| **while**((inChannel.read(bufs)) != -1) {  **for**(ByteBuffer b : bufs) {  b.flip();  outChannel.write(b);  b.clear();  }  } |

NIO的非阻塞式网络通信：

空闲通道：多路复用；

网络通信的三要素：

IP地址：可以唯一的定位到一台计算机

端口号：可以唯一的定位到一个程序

通信协议：TCP/IP UDP

阻塞式：

客户端：

1. 获取通道

2. 分配指定大小的缓冲区

3. 读取本地文件，并发送到服务端

4. 关闭通道

服务端：

1. 获取通道

2. 绑定连接

3. 获取客户端连接的通道

4. 分配指定大小的缓冲区

5. 接收客户端的数据，并保存到本地

6. 关闭通道

非阻塞式：

客户端

1. 获取通道

2. 切换非阻塞模式

3. 分配指定大小的缓冲区

4. 发送数据给服务端

5. 关闭通道

服务端

1. 获取通道

2. 切换非阻塞模式

3. 绑定连接

4. 获取选择器

5. 将通道注册到选择器上, 并且指定“监听接收事件”

6. 轮询式的获取选择器上已经“准备就绪”的事件

7. 获取当前选择器中所有注册的“选择键(已就绪的监听事件)”

8. 获取准备“就绪”的是事件

9. 判断具体是什么事件准备就绪

10. 若“接收就绪”，获取客户端连接

11. 切换非阻塞模式

12. 将该通道注册到选择器上

13. 获取当前选择器上“读就绪”状态的通道

14. 读取数据

选择器（Selector）

作用：针对非阻塞式IO通信；

打开选择器：Selector.open()；

将通道注册到选择器：

Selector.select()：查看选择器注册的通道数；

键：SelectionKey，状态值

值：Channel，通道

## 反射机制(reflection)

### 反射概述

反射：在程序运行时，可以加载、探知、使用编译期完全未知的类；

反射是Java被当作一门动态语言的基础；

反射：在程序运行时，通过类的Class对象（类名、对象获取）去动态获取类构造器、属性、方法；使代码变的灵活；

#### 类的加载

##### 类的加载：

①加载：将class文件加载到内存中；

②链接：

a）验证：验证.class文件内部结构是否完整；确保不会危害虚拟机安全；

b）准备：为static成员分配内存空间，赋默认值；

c）解析：将符号引用解析为直接引用；

③初始化：类的初始化；为静态成员赋值；

##### 类的初始化时机：

①创建类的实例；

②访问类的静态变量，或者为静态变量赋值；

③调用类的静态方法；

④使用反射方式来强制创建某个类或接口对应的Class对象；

⑤初始化某个类的子类；

⑥直接使用java.exe命令来运行某个主类；

##### 类加载器：

类加载器：将.class文件加载到内存中，为每一个类生成一个唯一的Class对象；

①BootstapClassloader 根类加载器：JDK基本包（核心类库）

②ExtensionClassloader 扩展类加载器：扩展类

③ApplicationClassloader 应用/系统类加载器：classPath配置中的类；

从上而下加载，从下而上检查；

### Class类

Class是final的，无公共构造方法。Class 对象是在加载类时由JVM以及通过调用类加载器中的 defineClass 方法自动构造的。

#### 获取Class对象

①对象.getClass()；

②类.class；

③Class.forName(String className)；全类名；

一个类只有唯一的一个Class对象；

#### 获取类信息

①public构造器：getConstructor(Class<?>... parameterTypes)；

②public属性：getField()；

③public方法：getMethod()；

④private构造器：getDeclaredConstructor()；

⑤private属性：getDeclaredField()；

⑥private方法：getDeclaredMethod()；

setAccessible(true)；//true忽略访问权限；

⑦获取该类的父类：getSuperclass()；

⑧获取该类实现的接口：[getInterfaces](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/Class.html#getInterfaces())()；

#### 反射在实际开发中的应用：

主要应用于框架中；

##### Properties类

Properties类：解析.properties属性配置文件：

.properties属性配置文件特点：

①文件内容以键值对形式存在；

②键和值以字符串形式存在；

ReturnObj

### 工厂设计模式

工厂模式：实例化对象模式；（工厂代替new操作）；

a）工业革命之前，你要一辆车，你需要自己new一个；

b）工业革命之后，你要一辆车，工厂给你造一个；

#### 简单工厂模式

静态工厂模式；

①具体工厂角色：创建产品对象

②抽象产品角色：定义产品的标准和规范

③具体产品角色：具体实现产品

优点：创建对象不使用new，使用工厂创建；

缺点：使工厂与产品产生了高度耦合，不符合类设计的开闭原则；

#### 工厂方法模式

①抽象工厂角色：

②具体工厂角色：

③抽象产品角色：

④具体产品角色：

优点：抽象工厂类的存在，降低了工厂与产品的耦合度；

符合程序的开闭原则，程序扩展性更强；

缺点：在扩展程序的时候，都需要创建具体工厂，程序较复杂；

#### 抽象工厂模式

①抽象工厂角色：

②具体工厂角色：

③抽象产品角色：

④具体产品角色：

优点：分离接口与实现：客户端使用抽象工厂来创建需要的对象；

切换产品族变的容易：

缺点：不易扩展新产品：

### Array类

Array类：动态创建和操作数组的类；

使用反射操作数组：

1.动态创建数组：Array.newInstance(类型.class,长度)；

2.判断是否是数组：arr.getClass().isArray()；

3.返回数组长度：Array.getLength(arr)；

4.设置值：Array.setInt(arr, index, value);

获取值：Array.getInt(arr, 0);

5.返回数组组件的类型：Class<?> arrayType = arr.getClass().getComponentType();

### 代理模式

作用：增强被代理对象；

#### 静态代理：

代理实现完全由程序员自己实现；只能针对特定的对象实现代理

实现步骤：

①定义主业务逻辑接口；

②实现主业务接口；

③代理类实现接口，增强主业务；

a）声明被代理对象：成员变量；

b）绑定被代理对象：有参构造器；

c）代理类增强主业务：方法扩展；

#### 动态代理：

可以代理任意对象；

##### 1.JDK动态代理实现：

InvocationHandler接口和proxy类；

[Object](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/reflect/../../../java/lang/Object.html) [invoke](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/reflect/../../../java/lang/reflect/InvocationHandler.html#invoke(java.lang.Object, java.lang.reflect.Method, java.lang.Object[]))([Object](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/reflect/../../../java/lang/Object.html) proxy, [Method](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/reflect/../../../java/lang/reflect/Method.html) method, [Object](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/reflect/../../../java/lang/Object.html)[] args)；

proxy：代理对象

method：扩展方法

args：扩展方法需要的参数

static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

loader：被代理对象的类加载器；

interfaces：被代理类实现的接口；

h：在代理类中所需要做的处理；被代理对象的处理器；

代理对象必须实现接口，代理的是接口；

实现步骤：

①定义主业务逻辑接口；

②实现主业务接口；

③编写针对代理对象的处理方式；

implements InvocationHandler，实现invoke()；

使用步骤：

①创建代理对象实例

②创建代理的处理器对象并绑定被代理对象；

③创建代理对象（被代理对象的接口）

proxy.newProxyInstance()，返回值是接口对象；

④通过代理对象调用主业务逻辑；

##### 2.cglib动态代理实现：

优点：代理类不需要实现接口；运行速度快；

实现步骤：

①实现主业务逻辑

②代理类implements MethodInterceptor

增强器：Enhancer；

实现interceptor方法

methodProxy.invokeSuper(被代理对象，方法参数)；

使用步骤：

①生成代理对象；

②代理对象调用方法；

## 多线程

### 多线程概述

进程：一个正在运行的程序；在计算机中的运行路径；

一个程序只能有一个进程；

程序是静态的，进程是动态的；

线程：进程的组成单元，在一个进程中有一个或多个线程；

并发：多个任务轮询交替执行；

并行：多任务同时执行；

CPU调度方式：

①时间片轮询；

②抢占式调度方式； Java采用抢占式；

进程与线程共享“堆内存和方法区”；一个线程一个“栈内存”；

Java程序的运行原理：

当使用Java命令启动一个程序的时候，JVM启动，就启动了一个进程；main方法会执行，main方法就是该进程中的一个线程（主线程），同时也会启动GC是后台线程（守护线程）；

### 创建线程

1.创建线程

①继承Thread类；

②实现Runnable接口：本质是[Thread](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/Thread.html#Thread(java.lang.Runnable))([Runnable](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/lang/../../java/lang/Runnable.html) target)；

核心步骤：重写run方法，写入该线程要完成的事情；

2.使用步骤：

a）重写run方法；

b）创建线程对象；

c）调用start()；

3.两种线程实现方式的区别：

|  |  |  |
| --- | --- | --- |
|  | 继承Thread类 | 实现Runnable接口 |
| 优点： | 可直接使用线程所有方法； | 多实现； |
| 缺点： | 单继承； | 不能直接创建线程； |

### 线程控制

1.线程名称

①设置名称：

Thread(String name)；

setName(String name)；

②String getName()；

③long getId()；//线程终生不变的唯一的标识符；

系统会为创建的线程设置默认Name：Thread-01；

主线程默认Name：main；

2.获取当前执行线程

Thread.currentThread()；返回对当前正在执行的线程对象的引用；

3.线程休眠

Thread.sleep(long millis)；在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）；

4.守护线程

void setDaemon(true)；将线程标记为守护线程；

isAlive()；判断线程是否是活动状态（已启动未终止）；

isDaemon()；判断线程是否是守护线程；

5.加入线程

void join()；当前线程暂停执行，直到加入的线程执行完毕；

应用：可以让另一个线程优先执行；

6.礼让线程

Thread.yield()；暂停当前正在执行的线程对象，其他线程并不一定能抢占到。

7.线程优先级

Java抢占式线程调度算法；优先级1-10；默认优先级5；

getPriority()；返回线程的优先级。

setPriority(int newPriority)；更改线程的优先级。

线程异常只能处理，不能抛出，无接受异常者；

### 线程的生命周期

线程状态

1.新建（New）：创建线程对象（new）；JVM为其分配内存，并初始化其成员变量的值

2.就绪（Runnable）：线程对象调用start()；JVM为其创建方法调用栈和程序计数器，等待CPU调度运行；

3.运行（Running）：线程获得CPU使用权，开始执行run()；

4.阻塞（Blocked）：当前正在执行的线程由于某种原因（休眠，礼让，时间到达）暂时失去CPU使用权，等待再次获得CPU使用权；

线程进入阻塞状态：

①wait()；

②执行同步锁时，锁对象被其他线程占用；

③IO操作；

④sleep()，join()；

解除阻塞：

1. 死亡（Dead）：线程执行结束；isAlive()==false；

①run()或call()方法执行完成，线程正常结束。

②线程抛出一个未捕获的Exception或Error。

③直接调用该线程stop()方法来结束该线程——该方法容易导致死锁，通常不推荐使用。

Sleep时间到

IO操作结束

获得同步锁

收到通知notify

start()

获得CPU使用权

yield()；

失去CPU使用权

wait()；sleep()

join()；

等待同步锁；

阻塞式IO操作

新建

就绪

运行

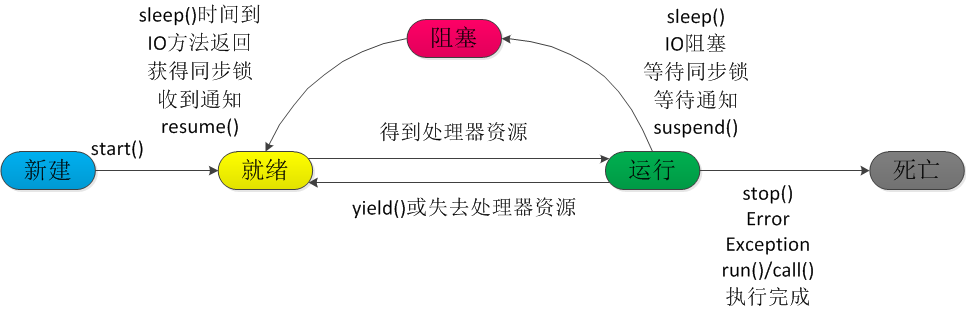
死亡

阻塞

run()执行完毕

stop()

异常



### 线程同步

线程同步：保护共享数据，防止数据不一致；

①同步代码块：

|  |
| --- |
| synchronized(obj){  } |

锁对象：可任意对象；但要保证多线程使用同一锁对象，实现多线程同步；

不能在run()中的同步代码块中使用this作为锁对象；

②同步方法：

同步方法：synchronized关键字修饰的方法；

普通方法锁对象：this；

静态方法锁对象：本类Class对象；

#### 多线程环境中安全使用集合API

Collections.synchronizedXxxx()；

Collections中的静态synchronized方法：

①Collection synchronizedCollention(Collection c)；

②List synchronizedList(list l)；

③Set synchronizedSet(Set s)；

④Map synchronizedMap(Map m)；

|  |
| --- |
| // Synchronized集合是线程安全的，但Iterator不是线程安全的；  List list = Collections.synchronizedList(new ArrayList());  ...  //迭代时，阻塞其他线程调用add()或remove()修改元素；  synchronized(list) {  Iterator i = list.iterator(); // Must be in synchronized block  while (i.hasNext())  foo(i.next());  } |

### 定时任务

Timer类

TimerTask：实现 Runnable接口，重写run()方法；由 Timer 安排为一次执行或重复执行的任务。

void schedule(TimerTask task, Date firstTime, long period)；

\*task 执行的任务

\*firsttime 开始执行的时间

\*period 任务的间隔执行时间

### Java并发编程Executor框架与线程池

Executor框架：基于并发编程的线程池；

Executor的UML图

Executor：

ExecutorService：

##### 创建线程池

Executors的静态方法：

①创建一个固定线程数的线程池：

static ExecutorService Executors.newFixedThreadPool(int nThreads)；

线程池中，如果有空闲线程，则执行任务；如果没有空闲线程，则任务进入阻塞状态，等待空闲线程；

线程池执行任务：void execute(Runnable command)；

②创建一个带有缓存的线程池

static ExecutorService newCachedThreadPool() ；

线程池中，如果有空闲线程，则执行任务；如果没有空闲线程，则创建一个新线程执行任务；

③创建一个单线程线程池

static [ExecutorService](mk:@MSITStore:C:\Users\admin\Desktop\JDK_API_1.6_zh_中文.CHM::/java/util/concurrent/../../../java/util/concurrent/ExecutorService.html) newSingleThreadExecutor()；

线程池中，如果有空闲线程，则执行任务；如果没有空闲线程，则创建一个新线程执行任务；

④创建一个定时执行线程池

static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

类似FixedThreadPool；

定时执行方法：

ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)；

\*command - 要执行的任务

\*delay - 从现在开始延迟执行的时间

\*unit - 延迟参数的时间单位

一般情况下，生命周期短的CachedThreadPool是首选；但是在特殊情况下（线程数>系统负载），生命周期长的会选择FixedThreadPool；

##### 线程池可执行的任务：

Rannable接口：public void run()；//无返回值；

Callable<v>接口：public V call()； //有返回值；

submit()；

get()；

线程池的生命周期

运行状态：

关闭状态：shutdown()；任务提交，不在执行新任务；

终止状态：任务全部提交完毕

### 线程协作

多个线程同时完成一个任务时，多个线程共享任务数据；防止数据不一致，使用线程同步（synchronized）机制；

synchronized：

原子性：在每次执行任务时，能保证只有一个线程在执行任务；

可见性：针对每一个线程执行任务时，所见到的数据都真是有效；

1.用5个售票窗口销售100张火车票；

2.银行账户有1000元，一个人去柜台取钱，另一个人去ATM机取钱；

取款方式：柜台，ATM机；

银行类，一个账户，两个线程模拟取钱，

3.火车过山洞：

火车由A到B，中间有一个隧道，每次只允许一趟火车通过，每趟火车经过隧道需要10s，为防止事故发生，每次只允许一趟火车经过，现有5趟火车经过隧道；

### 线程死锁

多个线程争取锁对象的持有权时的竞争现象；

避免死锁：

①尽量避免锁的嵌套使用；

②让程序执行时，尽可能只获得一个锁；

③超时限制：Lock中的tryLock，在获取锁对象时，可设置时间限制；

### 线程通信

#### Object类的监视器方法（monitor）

等待/通知机制

Object监视器方法：wait()、notify()、notifyAll()；

wait()：使线程进入等待状态，如果没有线程来唤醒，则一直处于等待状态；

wait(long timeout)：使线程进入等待状态；

notify()：唤醒当前处于等待状态的随机一个线程；

notifyAll()：唤醒当前处于等待状态的所有线程；

锁对象执行wait()；和notify()；

#### 互斥锁（Lock接口）

互斥锁：在一次执行中只能有一个线程持有锁对象；（JDK5新特性）

Lock接口+Condition接口；

2个以上线程通信，用到互斥锁；

#### 生产者-消费者模型

基于等待/通知机制；

①仓库：共享数据

②生产者：线程同步；不满足条件，wait；满足条件，生产产品+notify；

③消费者：线程同步；不满足条件，wait；满足条件，消费产品+notify；

## GUI

GUI（Graphical User Interface）：用户图形界面；应用程序提供给用户操作的图形界面，包括窗口、菜单、工具栏及其他屏幕元素；

常用应用：图片验证码；

GUI组件：一些基本的图形元素。

①容器（Container）：Frame，Dialog，Panel（JScrollPanel），

②组件（Component）：Button，Lable，RadioButton，Check，

在Windows中，通过组件与程序交互；在程序中，组件必须装配在容器里；

java.awt：重量级，已过时，调用操作系统底层；

javax.swing：组件名称前加J，调用java，跨平台性强；

①轻量级；

②丰富的控件；

③高度可定制；

④可插拔的外观和感觉；

基于MVC进行架构；

### 容器Container

#### 窗体Frame

顶级容器：可独立存在；Frame

JDialog

#### 面板Panel

中间容器：依赖于其他容器存在，不能单独存在；Panel

Panel被Frame添加add

### 容器布**局管理器LayoutManager**

LayoutManager：组件在面板上的排列方式；

流式布局：FlowLayout，按添加顺序从左向右放置；默认；

边框布局：BorderLayout，将容器划分为东、南、西、北、中；

网格布局：GridLayout(rows,cols)，划分为rows行cols列大小相等的网格；

卡片布局：CardLayout，只有一张卡片可见，切换卡片，切换功能；

空布局：setLayout(null)；通过坐标setBounds()，自定义组件位置；

### 组件Component

#### 文本JTextField

按钮

#### 菜单栏Menu

菜单条MenuBar-->菜单Menu-->菜单项MenuItem

工具栏

状态栏

树形菜单

#### 表格JTable

JTable=columnNames+cellDate；

JTable，TableModel，TableColumn；

①JTable的表头其实也是一个单独的组件TableHeader

②每一列也可以被划分出来作为一个组件TableColumn；

1.创建表格：

①JTable table = new JTable([rows],[cols])；

②以表头和单元格数据创建表格  
JTable table = new JTable(Object[][] cellData , String[] columnNames)；

JTable(Vector rowData, Vector columnNames)；

③通过表格模板创建表格

DefaultTableModel model = DefaultTableModel(Object[][] data, Object[] columnNames)；

JTable table = new JTable(model);

重写isCellEditable()使表格不可编辑

public boolean isCellEidtable() {

return false;

}

2.显示表头：

①添加到滚动面板：panel.add(new JScrollPane(table))；//常用

②获取表头方法：panel.add(table.getTableHeader(),BorderLayout.NORTH)

3.行控制

①行高：table.setRowHeight(20);

②设置行数：tableModel.setRowCount(n);

③获取行数：int rows = table.getRowCount();

④添加表格行

tableModel.addRow(new Object[]{"sitinspring", "35", "Boss"});

⑤删除表格行

model.removeRow(rowIndex); //行序号

4.列控制

① 设置列不可随容器组件大小变化自动调整宽度.

table.setAutoResizeMode(JTable.AUTO\_RESIZE\_OFF);

②限制某列的宽度.

TableColumn firstColumn = table.getColumnModel().getColumn(0);

firstColumn.setPreferredWidth(30);

firstColumn.setMaxWidth(30);

firstColumn.setMinWidth(30);

③设置当前列数.

tableModel.setColumnCount(5);

④取得表格列数

int cols = table.getColumnCount();

⑤添加列

tableModel.addColumn("新列名");

⑥删除列

table.removeColumn(table.getColumnModel().getColumn(columnIndex)); //列序号

5.表格控制

①获取单元格数据

String cellValue=(String) tableModel.getValueAt(row, column);

②获取TableModel：table.getModel();

③添加tableModel.addRow(arr)；

### 事件处理机制

1.事件源（组件）+事件监听器（XXXXListener）;

2.注册事件监听器（addXXXXListener）;

ActionnListener，MouseListener，KeyListener；

WindowListener，MouseMotionListener；

3.事件处理：在监听器接口中的动作方法中处理事件；

在swing事件处理机制中采用的设计模式：适配器模式；

### 适配器模式

有些情况下，无需实现接口所有方法，只需根据需求，实现部分方法即可；

①定义一个接口，在接口中定义规范；

②定义一个抽象类（适配器），抽象类实现接口，空实现接口方法；

③无需实现接口（无需实现所有方法），只需继承适配器（重写部分方法）；

鼠标事件

键盘事件

### JavaGUI的实现

1.窗体JFrame的创建和设置；

2.面板JPanel的创建和设置；

3.组件的创建和设置；

4.组件组装：frame.add(panel)；panel.add(组件)；

5.显示窗体：frame.setVisiable(true)；

6.事件处理机制：

①事件源：组件

②注册事件监听器：addXXXXListener( )；

③事件处理：动作方法的实现；

## 网络编程

IP：唯一的定位到网络中的一台计算机；

TCP：面向连接，保证数据的可靠传输；

UDP：面向无连接，不可靠的；

使用UDP数据报：TFTP，SNMP，DNS，NFS网络文件系统

IPv4：32位，4段\*8位；

域名：一个IP地址对应一个域名，映射关系；

HTTP协议：超文本传输协议;

发送HTTP请求的请求方式：GET，POST，PUT，DELETE；

URL：统一资源定位符；

常用网络命令：

①Ipconfig，

②ping：检测计算机之间的连通性；

### IP地址类：InetAddress

获取InetAddress对象：

①InetAddress[] getAllByName(String host)；

②InetAddress getByAddress(byte[] addr)；

③InetAddress getByAddress( [String host] , byte[] addr )；

④InetAddress getByName(String host)；

⑤InetAddress getLocalHost()；//返回本地主机；

常用方法：

//获取本机名称

ip.getHostName();

//获取本机地址

ip.getHostAddress();

### URL

URL：统一资源定位符；互联网资源指针；

1.URL

2.URLConnection

openConnection()；

3.URL编码与解码

URLEncoder：编码

URLDecoder：解码

### Socket编程

套接字是两台机器间通信的端点。

Socket编程三要素（IP，协议，port）；

①IP：唯一地定位网络中的一台计算机；

②协议：IP、TCP、UDP；

③port：定位计算机中的一个程序；

0-1024：系统端口号；

常用端口：Mysql（3306），oracle（1521），web（80），tomcat（8080）；

#### 基于TCP的Socket编程

TCP：面向连接的，可靠传输；

每进行一次客户端-服务端任务，建立一次连接；

1.客户端：

①Socket client = new Socket(服务端IP地址, 端口号);

②获取client的IO流；

③进行数据传输write&read；

④释放资源close()；

2.服务端：

①ServerSocket server = new ServerSocket(端口号);

Socket socket = server.accept(); //客户端的一次请求

②获取socket的IO流；

③数据传输read&write；

④释放资源close()；

3.优化：

①IO流：BufferedReader的readLine()；PrintWriter的write(String str)；

②多线程/线程池处理Server高并发

#### 基于UDP的Socket编程

UDP：面向无连接，不可靠传输；

发送端与接受端是相对的，不是绝对的；

1.发送端（sender）

①DatagramSocket sender = new DatagramSocket();

②创建一个发送数据报包（DatagramPacket）：

DatagramPacket(byte[] buf, int offset, int length, 目的IP地址, 端口号) ;

③发送数据报包sender.send(发送数据报);

2.接收端（receiver）

①DatagramSocket sender = new DatagramSocket(端口号);

②创建一个接收数据报包（DatagramPacket）：

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

③接收数据报包receiver.receive(接受数据报);

## JUnit单元测试框架

JUnit单元测试框架，第三方框架，XUnit中的一员；

\*JUnit3与JUnit4编写区别：

①JUnit3中的测试方法，必须以testXXX()命名；

②JUnit4中取消了以test前缀命名的规则，可任意命名；

单元测试优点：

①测试不需要去编写main()；

②可以针对不同的类，建立不同的单元测试类；

③可以针对不同方法，建立不同的单元测试方法；

#### JUnit的使用

1.@Test基本测试用例

2.断言测试

断言：判断预期结果与实际结果的差异；

Assert.assertEquals(预测结果,实际结果);

3.JUnit基本常用注解

@Before：方法在每个测试方法执行之前执行Before方法；

@After：方法在每个测试方法执行之后执行After方法；

@BeforeClass：static方法在测试类初始化时执行，只执行一次；

@AfterClass：static方法在测试类结束时执行，只执行一次；

@Test(timeout 1000)：设置方法执行超时时限；

@Test(expected= XX.class)：方法执行的期望结果；

@RunWith：用在类上，表明当前测试环境；

4.JUnit套件测试

@RunWith(Suite.class)

@Suite.SuiteClasses({

TestDemo.class,

Test1.class,

Test2.class,

Test3.class,

Test4.class

})

5.JUnit参数化测试

## Java注解

# GC&JVM

Java内存管理：对象的内存分配，对象的释放；

对象 = null；对象不可达；

当对象引用不可达时，GC将自动回收所有不可达的对象内存空间；

增量式GC就是通过一定的回收算法，把一个长时间的中断，划分为很多个小的中断，通过这种方式减少GC对用户程序的影响。

Hotspot JVM支持增量式GC；

执行FullGC：

①年轻代被写满

②永久代被写满

③System.gc()被显示调用

垃圾回收算法

①复制算法：

②标记清除算法：会产生磁盘碎片

③标记整理算法：

如何让gc运行更高效？

①尽可能尽早地释放无用对象的引用

②尽量少用finalize()；

③尽量及早释放数组、树、图、链表等，不要频繁地建立数据结构和集合；gc回收这些较为复杂；

④当程序有一定的等代时间，程序员可以手动执行苏system.gc()，通知gc进行垃圾回收；

类加载器：

①根类加载器BootStrapClassLoader：

a）由c++编写（所以获取不到，返回null）；

b）在java.lang包（所以lang的类无需导包）；

②扩展类加载器ExtentionClassLoader：java-jdk-ext；

③应用/系统类加载器ApplicationClassLoader：ClassPath路径下的所有类；

类加载器双亲委派机制：根类加载器-->扩展类加载器-->应用类加载器；

类的加载：

①加载：将class文件加载到内存中；

②链接：

a）验证：验证.class文件内部结构是否完整；确保不会危害虚拟机安全；

b）准备：为static成员分配内存空间，赋默认值；

c）解析：将符号引用解析为直接引用；

③初始化：类的初始化；为静态成员赋值；

面试题：

在Java中自定义了一个和Java.String类完全一样的类时，能否使用我自定义String类？

答：不能，类加载机制是双亲委派机制；在BootStrapClassloader中就可以加载到Java的String类，就不会到ApplicationClassLoader去加载String类；

#### 栈

本地方法接口：native；

栈：主要用来执行java程序，线程私有，生命周期与线程相同，

本地方法栈：为执行本地方法分配内存空间；

程序计数器：标记程序执行的跳转和调用；实现选择/循环

每一个线程都拥有独立的栈、本地方法栈、PC；

StackOverflowError：方法不断入栈，栈溢出

（例：方法自己调用自己，没有递归结束条件；）

#### 方法区

#### 堆内存

堆内存：新生代（Eden，from，to），老年代，永久代（不属于堆区）；

Eden-->Survive0（from）-->Survive1（to）-->老年代；

Java1.8之后，将 永久代 更改为 元空间；

OutOfMemoryError：内存溢出；

①JVM堆内存大小设置不够，更改参数可改变内存大小

②

MaxMemory占1/4，total\_Memory占1/64；