

1 Java 简介

Java 流行的原因是许多大公司都用 Java 开发其核心业务。

Java 由 SUN 公司开发，SUN 公司后被 Oracle 收购。

✧ 1.1 Java 技术 3 个发展方向：

- ① Java SE，实现 Java 基础支持，可以进行普通的单机版程序开发；
- ② Java EE，进行企业平台的搭建，现在主要是互联网平台开发；
- ③ Java ME，嵌入式开发，基本被 Android 开发(利用 Java 封装底层 Linux 操作)取代；后来由于 Oracle 和 Google 的撕逼，Google 推出了自己的专属语言 [Kotlin](#) 来进行 Android 开发。

✧ 1.2 Java 特点

- ① 面向对象编程；
- ② 自动内存回收机制；
- ③ 避免复杂的指针，使用简单的引用代替；
- ④ 为数不多支持多线程的语言；
- ⑤ 高效的网络处理能力，基于 NIO 实现更高效的数据传输处理；
- ⑥ 跨平台，良好的可移植性。

✧ 1.3 Java 虚拟机(Java Virtual Machine)

JVM 是一个由软件和硬件模拟出来的计算机，不同操作系统使用不同的 JVM。Java 源文件*.java 经编译得到[字节码文件](#)*.class，再经过解释得到[机器码指令](#)，解释都要求放在 JVM 上处理。

Java 编译器针对 JVM 产生的.class 文件是独立于平台的；Java 解释器负责将 JVM 的代码在特定平台上运行。

✧ 1.4 JDK 简介

Java Development Kit: Java 开发工具包。

JDK 历史：

- JDK 1.0，只提供基础环境，功能不完善；
- JDK 1.2，更名为 Java2，增加了 GUI 改进包、类集框架；
- JDK 1.4 是一个使用较广泛的版本，定义了许多重要的组件，如 NIO；
- JDK 1.5 是 Java 发展的重要里程碑，公布了新的结构化特点，极大简化开发难度；
- JDK 1.7，Oracle 收购 SUN 之后推出的正式版本，修复了很多漏洞；
- JDK 1.8，第一次正式提出 lambda 表达式的使用，支持函数式编程；
- JDK 1.9，提出交互式命令行工具、模块化设计。

然而现在 [JDK 1.10](#) 都已经发布了...

JDK1.8 是经过长期测试稳定的版本，项目中建议使用 JDK1.8。

JRE (Java Runtime Environment)只提供运行环境，不提供开发环境。

如果客户端要运行 Java，只需使用 JRE 就可以。

JDK 包含了 JRE，安装 JDK 也会安装 JRE。

两个重要的命令：

① 编译命令：javac.exe;

② 解释命令：java.exe

都在 JDK 安装目录的 bin 目录下，将该 bin 目录添加到环境变量。

```
$ javac -version
javac 10.0.1

$ java -version
java version "10.0.1" 2018-04-17
Java(TM) SE Runtime Environment 18.3 (build 10.0.1+10)
Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.1+10, mixed mode)
```

2 Java 编程起步

✧ 2.1 第一个程序

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

1) 对源代码进行编译

```
$ javac Hello.java
```

利用 JVM 编译出与平台无关的字节码文件 Hello.class

2) 在 JVM 上进行程序的解释执行：

```
$ java Hello
hello world!
```

① Java 最基础的单元是类，所有程序必须封装在类中执行。

1) **public class** 定义的类名必须与文件名一致，一个.java 文件只能有一个 public class 定义的类；

比如将上面 Hello 改为 HelloWorld，编译会报错：

```
$ javac Hello.java
Hello.java:1: 错误: 类 HelloWorld 是公共的, 应在名为 HelloWorld.java 的文件中声明
```

2) **class** 定义的类可以与文件名不一致，一个.java 文件可以有多个。

编译后生成对应多个.class 字节码文件。

注意：实际开发很少在一个.java 文件里定义多个 class；而是只定义一个 public class 就可以。但学习的时候定义很多.java 文件会很混乱；为了方便学习，还是会一个.java 文件定义多个 class。

② 主方法

主方法是所有程序执行的起点，必须定义在类中。

✧ 2.2 JShell

很多编程语言为了方便使用者代码开发，都会有 Shell 交互式编程环境。

有时只是为了进行简短的程序验证，但在 Java 里必须编写很多结构代码才行。JDK 9 开始提供 JShell，可以只编写代码就能快速验证。类似于 Python Shell，但貌似用处不大。

```
$ jshell
| 欢迎使用 JShell -- 版本 10.0.1
| 要大致了解该版本，请键入: /help intro

jshell> 23 + 45
$1 ==> 68

jshell> "hello" + " " + "world"
$2 ==> "hello world"
```

也可以将语句写入文件保存，再运行：

```
jshell> /open hello.txt
hello, hikari!

jshell> /exit
| 再见
```

✧ 2.3 CLASSPATH 环境属性

如在 D:/learn_java/目录有 Hello.java 文件，javac 编译在该目录生成 Hello.class 字节码文件，可以 java Hello 解释运行；但是切换到其他没有 Hello.class 的目录如 C:/运行会报错：

```
C:\>java Hello
错误: 找不到或无法加载主类 Hello
原因: java.lang.ClassNotFoundException: Hello
```

修改 CLASSPATH 环境属性：

```
C:\>SET CLASSPATH=d:/learn_java
C:\>java Hello
hello world!
```

Java 程序解释的时候 JVM 通过 CLASSPATH 设置的路径进行类的加载。CLASSPATH 默认为当前目录。

修改 CLASSPATH 的坏处是：

```
C:\>javac A.java
C:\>java A
错误: 找不到或无法加载主类 A
原因: java.lang.ClassNotFoundException: A
C:\>SET CLASSPATH=.
C:\>java A
hello world!
```

当想运行 C 盘的字节码文件时，无法运行，因为设置的 CLASSPATH 目录没有 A.class；设为 . 表示当前目录，就可以运行了。
所以 CLASSPATH 一般采用默认设置。

注意：CLASSPATH 是在一个命令行的配置，如果关闭命令行该配置也消失。
最好做法是定义为全局属性：环境变量→新建→变量名为 CLASSPATH，变量值为 .

PATH 和 CLASSPATH 的区别：

- ① PATH 是操作系统提供的路径配置，定义所有可执行程序的路径；
- ② CLASSPATH 是 JRE 提供，用于设置 Java 程序解释时类加载路径，默认为 . 可以通过 SET CLASSPATH=<路径> 设置。

20180527

3 Java 基本概念

✧ 3.1 注释

好的注释使得项目维护更加方便，但很多公司由于管理不善，不写注释。如果作为菜鸟进入这样的公司，维护时面对密密麻麻几万行没有注释的代码，心中定会有几万头草泥马飞奔而过。

编译器在编译时不会对注释内容进行编译处理，Java 有 3 种注释：

- ① 单行注释：//...
- ② 多行注释：/* ... */
- ③ 文档注释：/** ... */，文档注释里需要有很多选项，建议通过开发工具控制使用 IDE 开发时，单行注释比较方便；对于一些重要的类和方法建议使用文档注释。

✧ 3.2 标识符和关键字

- ① 标识符由字母、数字、_、\$ 组成，不能以数字开头，不能是 Java 的关键字。用以标记变量名、方法名、类名等。

JDK1.7 增加了一个神奇的特性：标识符可以用中文！

但基本没人会这么用，对于 JDK 的新特性要保守使用。

- ② 关键字是系统对于一些结构的描述处理，有特殊的含义。一般 IDE 都会显示特殊的颜色，不需要死记硬背啦。

注意：

- 1) JDK 1.4 添加 **assert** 关键字，断言，用于程序调试；
- 2) JDK 1.5 添加 **enum** 关键字，用于枚举定义；
- 3) 未使用到的关键字(保留关键字)：**goto**、**const**；
- 4) 特殊含义的标记：**true**、**false**、**null**，严格来说不算关键字。

4 Java 数据类型

✧ 4.1 Java 数据类型

① 基本数据类型：描述一些具体的数字单元，不牵扯内存分配问题。

1) 数值型：

a) 整型：byte、short、int、long，默认值 0

b) 浮点型：float、double，默认值 0.0

2) 布尔型：boolean，默认值 false

3) 字符型：char，默认值 '\u0000'

不同类型所占大小不一样，保存数据范围也不同。

使用参考：

1) 通常，整数使用 int，小数使用 double；

2) 数据传输或文字编码转换使用 byte 类型(二进制操作)；

3) 处理中文使用 char 较方便；

4) 描述时间、内存或文件大小、描述表的主键列(自动增长)可以使用 long。

② 引用数据类型：牵扯到内存关系的使用。

数组、类、接口，默认值 null

✧ 4.2 整型

Java 任何一个整型常量都是默认 int 类型。

如果变量在处理过程中超过了最大的保存范围，就会数据溢出。

```
int MAX = Integer.MAX_VALUE;
int MIN = Integer.MIN_VALUE;
System.out.println("MAX + 1 = " + (MAX + 1)); // MAX + 1 = -2147483648
System.out.println("MIN - 1 = " + (MIN - 1)); // MIN - 1 = 2147483647
```

$MAX=2^{31}-1$ ，二进制为 0111,1111,1111,1111,1111,1111,1111,1111，加 1 后变为 1000,0000,0000,0000,0000,0000,0000,0000，因为最高位是符号位，所以对应十进制是其补码取反后的十进制+1 再加上负号，即 -2^{31} 。

解决数据溢出就要使用范围更大的数据类型，比如 long：

```
// 可以(long)1、或 1L/1l 转换成 long 类型
System.out.println("MAX + 1 = " + ((long) MAX + 1)); // MAX + 1 = 2147483648
System.out.println("MIN - 1 = " + (MIN - 1L)); // MIN - 1 = -2147483649
```

运算符两边其中一个是 long，另一个是 int，int 会自动转换成 long，因为 long 的范围比 int 大，也就是数据范围小的类型会自动转换为范围大的类型。

long 类型赋值给 int 需要强制类型转换：

```
long a = -2147483649L;
// 范围大的转为范围小的类型需要强制类型转换
int b = (int) a;
System.out.println(b); // 2147483647
```

但会出现数据溢出。不是必须情况，不建议使用强制类型转换。

Java 对 byte 做了特殊处理，如果没超过范围，自动将 int 常量转为 byte；超过范围必须强制类型转换；对于变量也需要强制类型转换。

```
byte n = 20; // 20 在 byte 范围,自动转为 byte
System.out.println(n); // 20
byte m = (byte) 200; // 超过 byte 范围,强制类型转换
System.out.println(m); // -56
int a = 12;
byte s = (byte) a; // 变量,强制类型转换
System.out.println(s); // 12
```

200 转为 byte 为 1100,1000，最高位是符号位，剩余位取反为 110111 对应 55，加 1 为 56，符号位是 1 表示负数，结果是-56。

✧ 4.3 浮点型

① double

```
double a = 12; // 自动类型转换
System.out.println(a); // 12.0
```

自动类型转换都是由范围小的类型向范围大的类型转换。

② float

```
// 默认小数类型为 double,需要强制类型转换为 float
float a = 10.1F;
float b = (float) 10.2;
// 浮点数计算存在精度损失,目前仍未解决
System.out.println(a * b); // 103.020004
```

注意：两个 int 相除得到 int，想要得到 double 需要将其中一个数转为 double：

```
int a = 18;
int b = 4;
// 两个 int 相除,地板除得到 int
System.out.println(a / b); // 4
// 想得到真实结果需要其中一个转换为 double
System.out.println((double) a / b); // 4.5
```

✧ 4.4 字符型

Java 的 char 使用 Unicode 编码，占 2 个字节，包括世界上常用的文字。

```
public class Hello {
    public static void main(String[] args) {
        char c='星';
        int n =c;
        System.out.println(n); // 26143
    }
}
```

由于 Win7 默认 GBK 编码，javac 使用系统默认编码，而此次 java 使用 UTF-8 编码，编译就会报错：编码 GBK 的不可映射字符

在编译需要指定编码为 UTF-8:

```
$ javac -encoding utf-8 Hello.java
$ java Hello
26143
```

✧ 4.4 字符串

任何项目都会用到 **String**，是引用数据类型，一个特殊的类，可以像普通变量采用直接赋值。

可以使用+进行字符串拼接:

```
String s = "result:";
int a = 10;
int b = 24;
// 有 String,所有类型无条件先变为 String
System.out.println(s + a + b); // result:1024
System.out.println(s + (a + b)); // result:34
```

5 运算符

运算符有优先级，但不用记，需要时优先计算直接使用()。

不要编写很复杂的计算，如: $a-- + b++ * --b / a / b * ++a - --b + b++$

一般大学老师或奇葩的面试官才会出这种题。

✧ 5.1 数学运算符

+、-、*、/、%

简化运算符: +=、-=、*=、/=、%=

自增自减: ++、--

```
int a = 23;
int b = 10;
// a++先使用变量后自增; --b 先自减后使用变量
System.out.println(a++ - --b); // 14
System.out.println("a=" + a + ", b=" + b); // a=24, b=9
```

这些代码是以前内存不大时提供的处理方式，但在硬件成本降低的现在，这种计算很繁琐。一般自增自减独立一行写较为直观。

✧ 5.2 关系运算符

>、<、>=、<=、==，关系运算符返回布尔类型

✧ 5.3 三目运算符

简化 if-else 语句，避免无谓的赋值运算处理。

```
int a = 10;
int b = 20;
int max = a > b ? a : b;
System.out.println(max); // 20
int c = 15;
int min = a < b ? (a < c ? a : c) : (b < c ? b : c);
```

```
System.out.println(min); // 10
```

三目运算可以嵌套，但可读性变差，根据实际情况使用。

✧ 5.4 逻辑运算符

&、&&、|、||、!

&& / || 只要前面有一个为 false / true，直接返回 false / true，后面表达式不会判断。应该一直使用短路与和短路或。

✧ 5.5 位运算符

&、|、^、~、>>、<<

```
System.out.println(12 & 7); // 4
System.out.println(12 | 7); // 15
System.out.println(3 << 4); // 48
```

12 对应二进制 1100，7 对应二进制 0111。(前面 28 个 0 省略)

按位与得到 0100 即 4；按位或得 1111 即 15。

$a \ll n$: a 左移 n 位，相当于 $a \times 2^n$ ，比如 $3 \ll 4$ 就是 $3 \times 2^4 = 48$

面试题：&和&&的区别

- ① &和&&都可以作为逻辑运算符：&是普通与，所有条件都要判断；&&是短路与，只要前面判断为 false，后面不再进行判断，直接返回 false。
- ② &可以作为位运算符，&&不可以。

6 程序逻辑结构

三种程序逻辑结构：顺序结构、分支结构、循环结构。

✧ 6.1 分支结构

① if-else

② switch-case

switch 最早只能 int 活 char 判断，JDK 1.5 支持枚举判断，JDK1.7 支持 String 判断；不支持布尔类型。

```
String s = "sun";
switch (s) {
    case "mon":
    case "tue":
    case "wed":
    case "thu":
    case "fri": {
        System.out.println("工作!");
        break;
    }
    case "sat":
    case "sun": {
        System.out.println("休息!");
        break;
    }
}
```



```

    }
    default: {
        System.out.println("输入错误!");
    }
}

```

✧ 6.2 循环结构

① while / do while 循环

do while 不管条件满不满足，循环都要至少执行一次。几乎不用 do while。

② for 循环

知道循环次数优先 for 循环；

不知道循环次数但知道循环结束条件时，使用 while 循环。

③ 循环控制

break 和 continue

④ 循环嵌套

练习：打印三角形

```

int line = 5; // 行数
for (int x = 0; x < line; x++) { // 打印几行
    for (int i = 0; i < line - x; i++) { // 打印空格
        System.out.print(" ");
    }
    for (int i = 0; i <= x; i++) { // 打印*
        System.out.print("* ");
    }
    System.out.println();
}

```

结果：

```

    *
   * *
  * * *
 * * * *
* * * * *

```

这只是程序逻辑的训练，与程序开发关系不大，主要面向应届生。

7 方法

程序中可能需要重复执行某段代码，可以将其封装成方法(method)，也有的叫函数(function)。

定义方法有利于重复调用，所有程序都是通过主方法开始执行的。

✧ 7.1 方法的重载

方法名称相同，参数的个数或类型不同时称为方法的重载。

```

public static void main(String[] args) {
    int a = sum(12, 34);
    int b = sum(12, 34, 56);
    double c = sum(1.2, 5.6);
    System.out.println("a=" + a + ", b=" + b + ", c=" + c); // a=46, b=102, c=6.8
}

public static int sum(int a, int b) {
    return a + b;
}

public static int sum(int a, int b, int c) {
    return a + b + c;
}

public static double sum(double a, double b) {
    return a + b;
}

```

方法重载与返回值类型没有关系，只和参数有关系。

实际开发建议方法重载返回值类型相同。

可以发现 `System.out.println()` 是系统提供的方法重载。

✧ 7.2 递归 (recursion)

需要设置递归结束的条件；每次调用时要修改传递的参数条件。

```

public static void main(String[] args) {
    System.out.println(fib(20)); // 6765
}

public static int fib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}

```

实际开发编写的代码很少出现递归，常用的递归是系统内部提供的。

递归处理不当，容易造成栈内存溢出。

一般面试题常用递归，如汉诺塔、八皇后、二叉树遍历等问题。

8 类与对象

✧ 8.1 面向对象

Java 最大特点就是面向对象编程。也有开发者认为面向过程后函数式编程好。

C 语言已经是面向过程开发的代表，面向过程是面向一个问题的解决方案，

更多情况不会考虑重复利用。面向对象主要设计形式的模块化设计，可以重用配置，面向对象更多考虑的是标准，使用时根据标准拼装。

面向对象的 3 个主要特征：

- ① 封装：内部的操作对外部不可见，安全；
- ② 继承：在已有结构的基础上进行功能扩充；
- ③ 多态：在继承的基础上扩充的概念，类型的转换处理

面向对象开发 3 步骤：

- ① OOA：面向对象分析；
- ② OOD：面向对象设计；
- ③ OOP：面向对象编程。

✧ 8.2 类与对象

类是某一类事物共性的抽象概念；对象是一个具体的产物。
类是一个模板，对象是类创建的实例，先有类后有对象。

类一般有两个组成：

- ① 属性 (Field)：定义对象的水仙
- ② 方法 (Method)：定义对象的行为

示例：

```
class Person {
    String name;
    int age;
    public void show() {
        System.out.println("我的名字是" + name + ", 今年" + age + "岁了!");
    }
}

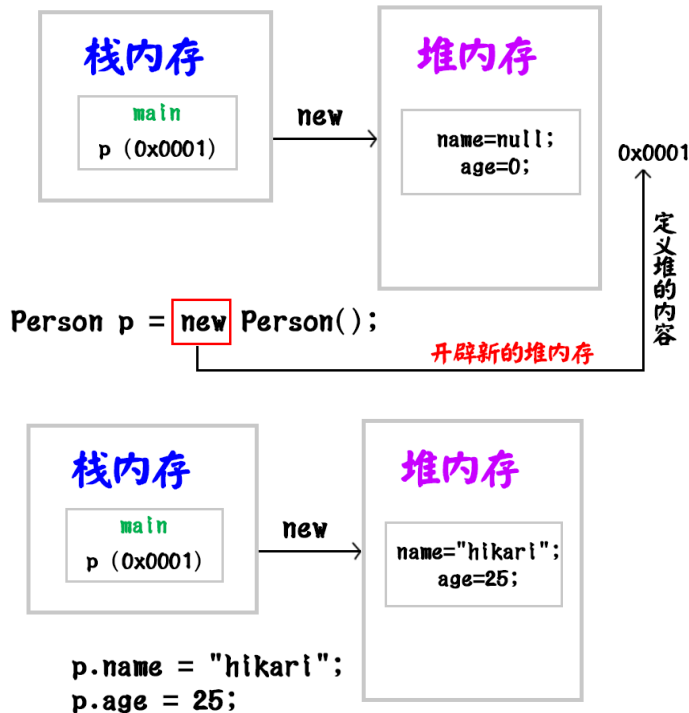
public class Hello {
    public static void main(String[] args) {
        Person p = new Person();
        // 未设置属性,使用默认值
        p.show(); // 我的名字是 null, 今年 0 岁了!
        p.name = "hikari";
        p.age = 25;
        p.show(); // 我的名字是 hikari, 今年 25 岁了!
    }
}
```

类是引用数据类型，其最大困难之处在于内存的管理，在操作时也会发生内存关系的变化。

最常用的内存空间：

- ① 堆内存：保存对象的具体信息；堆内存空间的开辟通过 `new` 完成；
- ② 栈内存：保存一块堆内存的地址，通过地址找到堆内存，找到对象内容

简单内存分析：



对象实例化语句可以是：

- ① 声明并实例化对象：`Person p = new Person();`
- ② 先声明对象，再实例化对象：

```
Person p = null;  
p = new Person();
```

对象调用类的属性或方法必须实例化完成后才能执行。

如果只声明对象，而没有实例化，调用会出错：

```
Exception in thread "main" java.lang.NullPointerException
```

抛出空指针异常，就是堆内存没有开辟时产生的问题，只有引用数据类型存在空指针异常。

✧ 8.3 引用传递

内存的引用传递：同一块堆内存空间可以被不同的栈内存所指向，也可以更换指向。一个栈内存只能保存一个堆内存地址数据。

```
Person p = new Person();  
p.name = "hikari";  
p.age = 25;  
p.show(); // 我的名字是 hikari, 今年 25 岁了!  
Person p1 = p; // 引用传递
```

```
p1.age = 20;
p.show(); // 我的名字是 hikari, 今年 20 岁了!
```

p 和 p1 指向同一个堆内存空间，通过 p1 修改对象属性，p 也会改变。

也可以通过方法实现引用传递：

```
public static void main(String[] args) {
    Person p = new Person();
    p.name = "hikari";
    p.age = 25;
    p.show(); // 我的名字是 hikari, 今年 25 岁了!
    change(p); // 通过方法实现引用传递
    p.show(); // 我的名字是 hikari, 今年 10 岁了!
}

public static void change(Person a) {
    a.age = 10;
}
```

将 Person 的实例化对象 p (本质存的是内存地址)传递到 change()方法，a 也指向与 p 同一块堆内存空间。方法执行完毕，a 断开连接。

没有任何栈内存指向的堆内存空间就是垃圾空间，所有垃圾将被 GC (Garbage Collection)不定期进行回收，释放无用内存空间；如果垃圾过多，将影响 GC 的性能，从而降低整体程序性能。

```
Person p = new Person();
Person p1 = new Person();
p.name = "hikari";
p.age = 25;
p1.name = "maki";
p1.age = 15;
p1 = p; // 引用传递
p1.name = "maki";
p.show(); // 我的名字是 maki, 今年 25 岁了!
```

一开始 p 和 p1 指向各自的堆内存，后来 p1 指向 p 指向的堆内存，原来 p1 指向的堆内存没有任何栈内存指向，因而成为垃圾被回收。

