图文详解 53 道Java基础面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。

Java 概述

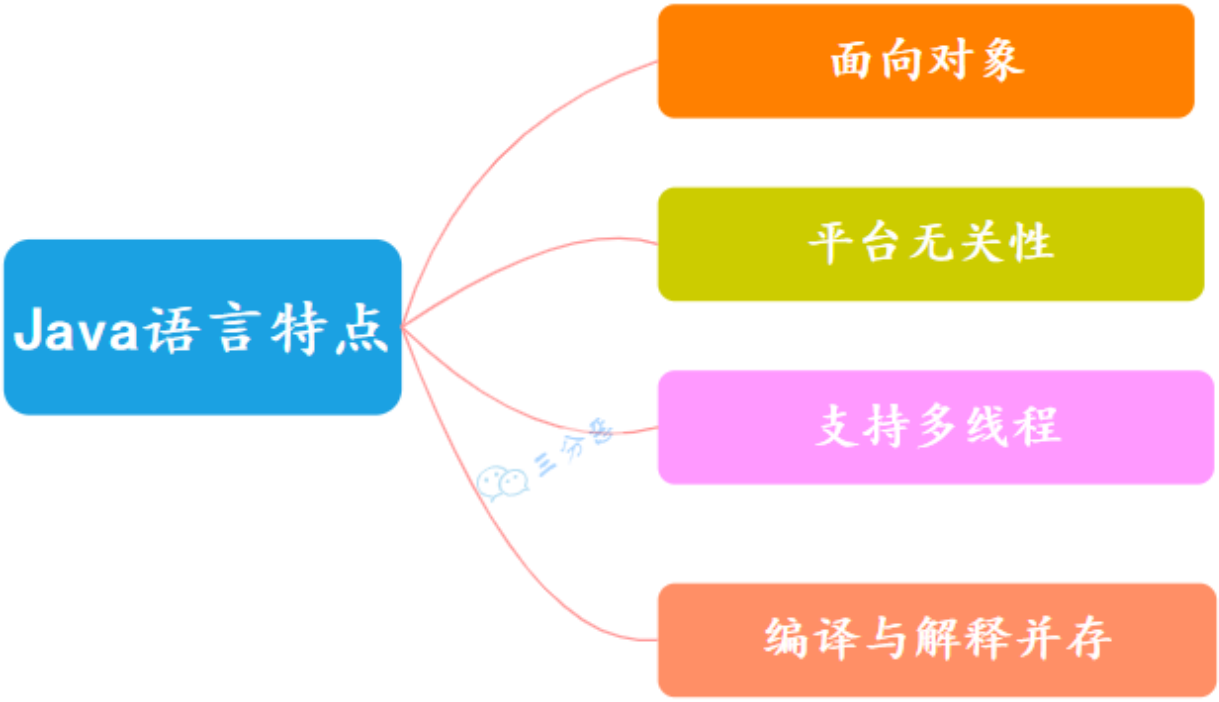
1.什么是 Java?

PS：碎怂 Java，有啥好介绍的。哦，面试啊。

Java 是一门面向对象的编程语言，不仅吸收了 C++语言的各种优点，还摒弃了 C++里难以理解的多继承、指针等概念，因此 Java 语言具有功能强大和简单易用两个特征。Java 语言作为静态面向对象编程语言的优秀代表，极好地实现了面向对象理论，允许程序员以优雅的思维方式进行复杂的编程 。

2.Java 语言有哪些特点?

Java 语言有很多优秀（可吹）的特点，以下几个是比较突出的：



- 面向对象（封装，继承，多态）；
- 平台无关性，平台无关性的具体表现在于，Java 是“一次编写，到处运行（Write Once，Run any Where）”的语言，因此采用 Java 语言编写的程序具有很好的可移植性，而保证这一点的正是 Java 的虚拟机机制。在引入虚拟机之后，Java 语言在不同的平台上运行不需要重新编译。
- 支持多线程。C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持；
- 编译与解释并存；

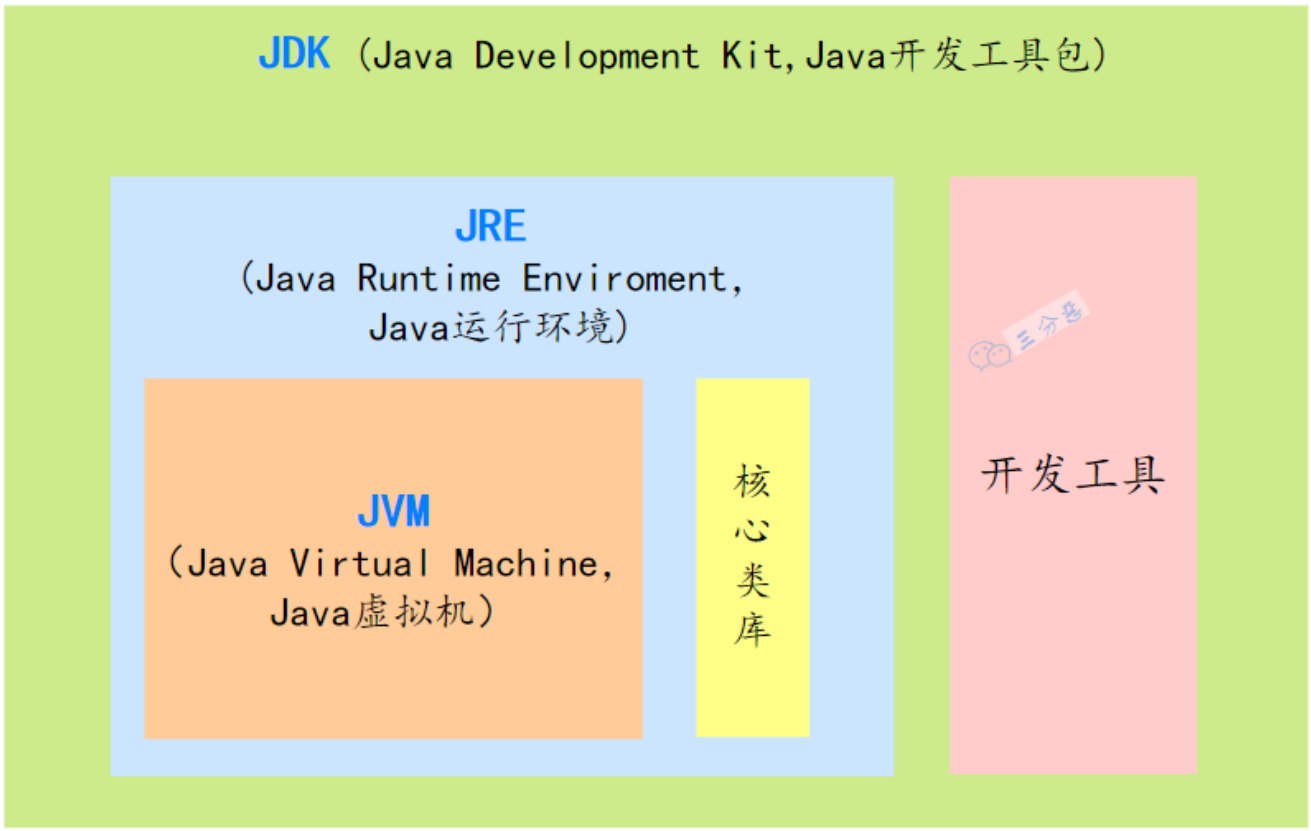
3.JVM、JDK 和 JRE 有什么区别？

JVM：Java Virtual Machine，Java 虚拟机，Java 程序运行在 Java 虚拟机上。针对不同系统的实现（Windows，Linux，macOS）不同的 JVM，因此 Java 语言可以实现跨平台。

JRE：Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，Java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

JDK：Java Development Kit，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

简单来说，JDK 包含 JRE，JRE 包含 JVM。



4.说说什么是跨平台性？原理是什么

所谓跨平台性，是指 Java 语言编写的程序，一次编译后，可以在多个系统平台上运行。

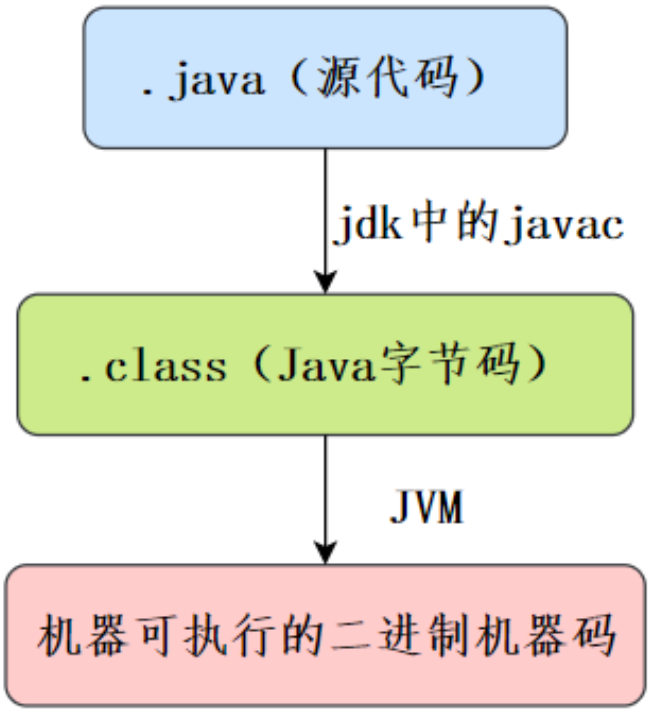
实现原理：Java 程序是通过 Java 虚拟机在系统平台上运行的，只要该系统可以安装相应的 Java 虚拟机，该系统就可以运行 java 程序。

5.什么是字节码？采用字节码的好处是什么？

所谓的字节码，就是 Java 程序经过编译之类产生的.class 文件，字节码能够被虚拟机识别，从而实现 Java 程序的跨平台性。

Java 程序从源代码到运行主要有三步：

- **编译**：将我们的代码（.java）编译成虚拟机可以识别理解的字节码(.class)
- **解释**：虚拟机执行 Java 字节码，将字节码翻译成机器能识别的机器码
- **执行**：对应的机器执行二进制机器码



只需要把 Java 程序编译成 Java 虚拟机能识别的 Java 字节码，不同的平台安装对应的 Java 虚拟机，这样就可以可以实现 Java 语言的平台无关性。

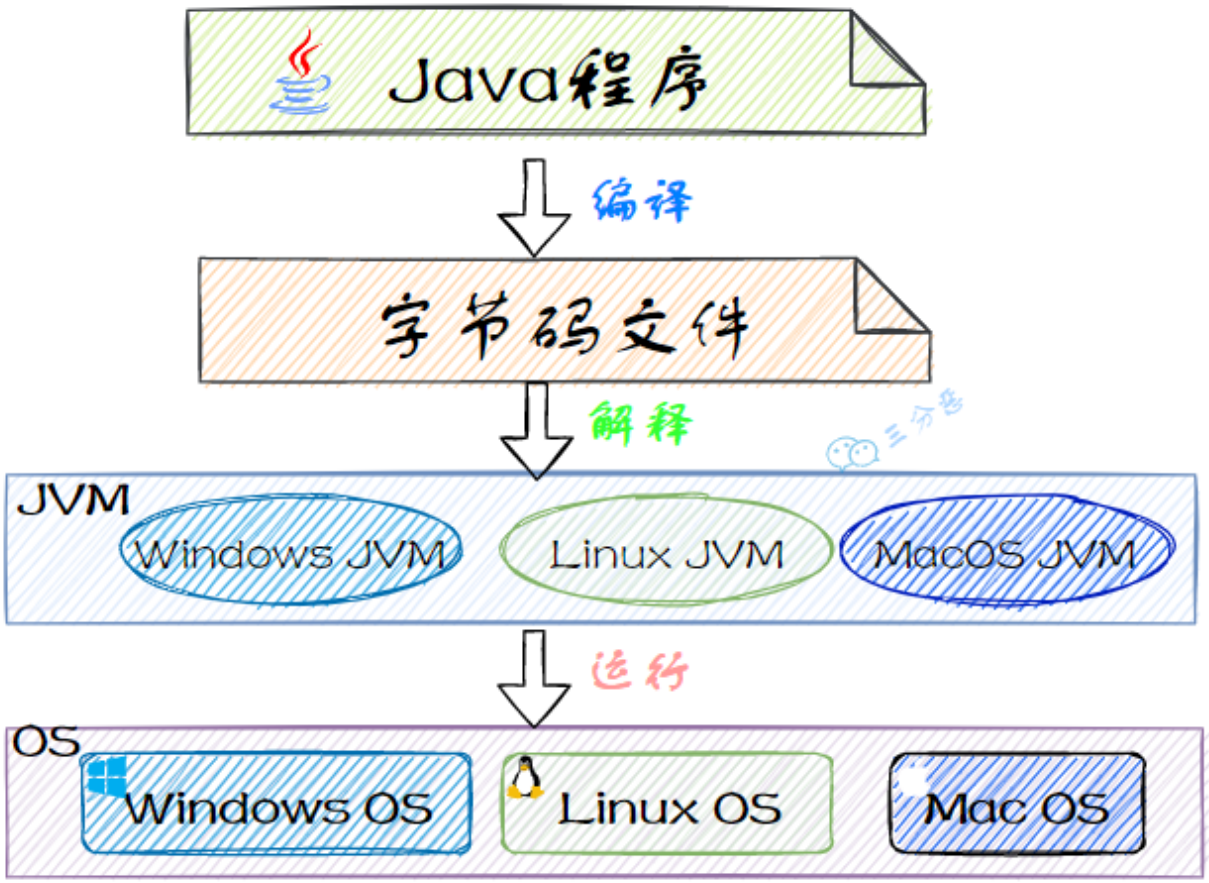
6.为什么说 Java 语言“编译与解释并存”?

高级编程语言按照程序的执行方式分为**编译型**和**解释型**两种。

简单来说，编译型语言是指编译器针对特定的操作系统将源代码一次性翻译成可被该平台执行的机器码；解释型语言是指解释器对源程序逐行解释成特定平台的机器码并立即执行。

比如，你想读一本外国的小说，你可以找一个翻译人员帮助你翻译，有两种选择方式，你可以先等翻译人员将全本的小说（也就是源码）都翻译成汉语，再去阅读，也可以让翻译人员翻译一段，你在旁边阅读一段，慢慢把书读完。

Java 语言既具有编译型语言的特征，也具有解释型语言的特征，因为 Java 程序要经过先编译，后解释两个步骤，由 Java 编写的程序需要先经过编译步骤，生成字节码（`\*.class` 文件），这种字节码必须再经过 JVM，解释成操作系统能识别的机器码，在由操作系统执行。因此，我们可以认为 Java 语言**编译与解释并存**。


Java程序
编译
字节码文件
解释
JVM
Windows JVM
Linux JVM
MacOS JVM
运行
OS
Windows OS
Linux OS
Mac OS


关注沉默王二
学Java不迷路

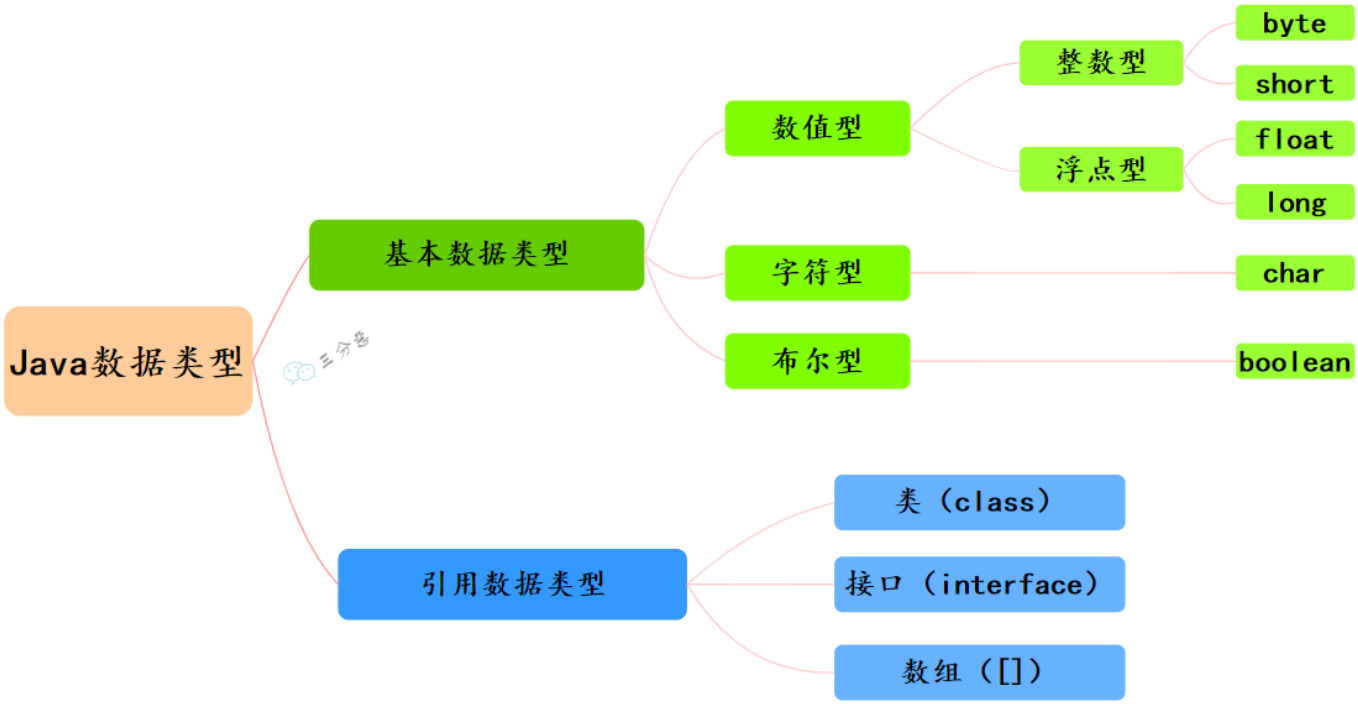

16:58
公众号
沉默王二
发消息
技术文通俗易懂，吹水文风趣幽默。
689篇原创内容 7780个朋友关注
视频号：沉默王二
精彩内容不容错过，加个星标吧
每天一篇硬核Java后端文章，带你
循序渐进学Java！
消息
视频号
服务
1
2
设为星标
推荐给朋友

基础语法

7.Java 有哪些数据类型?

定义：Java 语言是强类型语言，对于每一种数据都定义了明确的具体的数据类型，在内存中分配了不同大小的内存空间。

Java 语言数据类型分为两种：基本数据类型和引用数据类型。



基本数据类型：

- 数值型
  - 整数类型（byte、short、int、long）
  - 浮点类型（float、double）
- 字符型（char）
- 布尔型（boolean）

Java 基本数据类型范围和默认值：

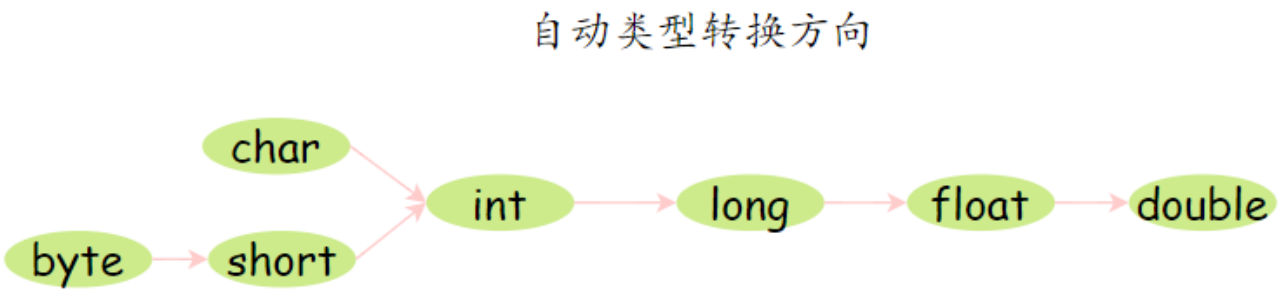
基本类型	位数	字节	默认值
int	32	4	0
short	16	2	0
long	64	8	0L
byte	8	1	0
char	16	2	'u0000'
float	32	4	0f
double	64	8	0d
boolean	1		false

引用数据类型：

- 类（class）
- 接口（interface）
- 数组([])

8.自动类型转换、强制类型转换？看看这几行代码？

Java 所有的数值型变量可以相互转换，当把一个表数范围小的数值或变量直接赋给另一个表数范围大的变量时，可以进行自动类型转换；反之，需要强制转换。



这就好像，小杯里的水倒进大杯没问题，但大杯的水倒进小杯就不行了，可能会溢出。

`float f=3.4` ，对吗？

不正确。3.4 是单精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换`float f =(float)3.4;`或者写成`float f =3.4F`

`short s1 = 1; s1 = s1 + 1;` 对吗？ `short s1 = 1; s1 += 1;` 对吗？

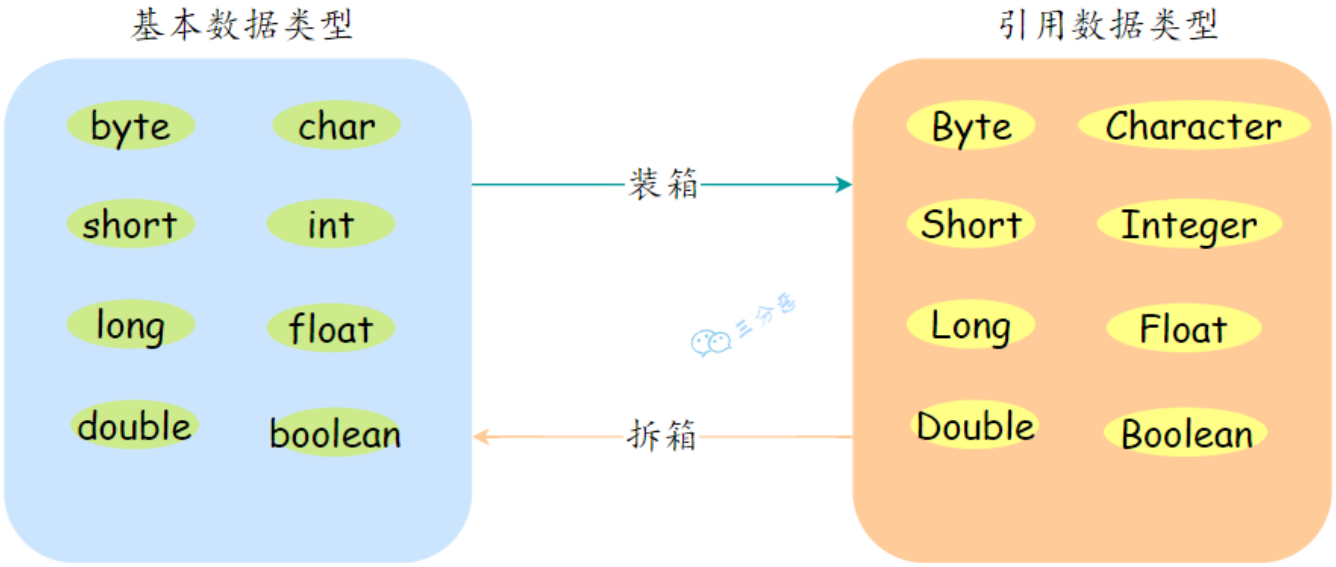
对于 short s1 = 1; s1 = s1 + 1;编译出错，由于 1 是 int 类型，因此 s1+1 运算结果也是 int 型，需要强制转换类型才能赋值给 short 型。

而 short s1 = 1; s1 += 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short(s1 + 1);其中有隐含的强制类型转换。

9.什么是自动拆箱/封箱？

- 装箱：将基本类型用它们对应的引用类型包装起来；
- 拆箱：将包装类型转换为基本数据类型；

Java 可以自动对基本数据类型和它们的包装类进行装箱和拆箱。



举例：

```
Integer i = 10;  //装箱
int n = i;   //拆箱
```

10.&和&&有什么区别？

&运算符有两种用法：`短路与`、`逻辑与`。

&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是 true 整个表达式的值才是 true。

&&之所以称为短路运算是因为，如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&。

例如在验证用户登录时判定用户名不是 null 而且不是空字符串，应当写为`username != null &&!username.equals("")`，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的 equals 比较，否则会产生 NullPointerException 异常。

注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

11.switch 是否能作用在 byte/long/String 上?

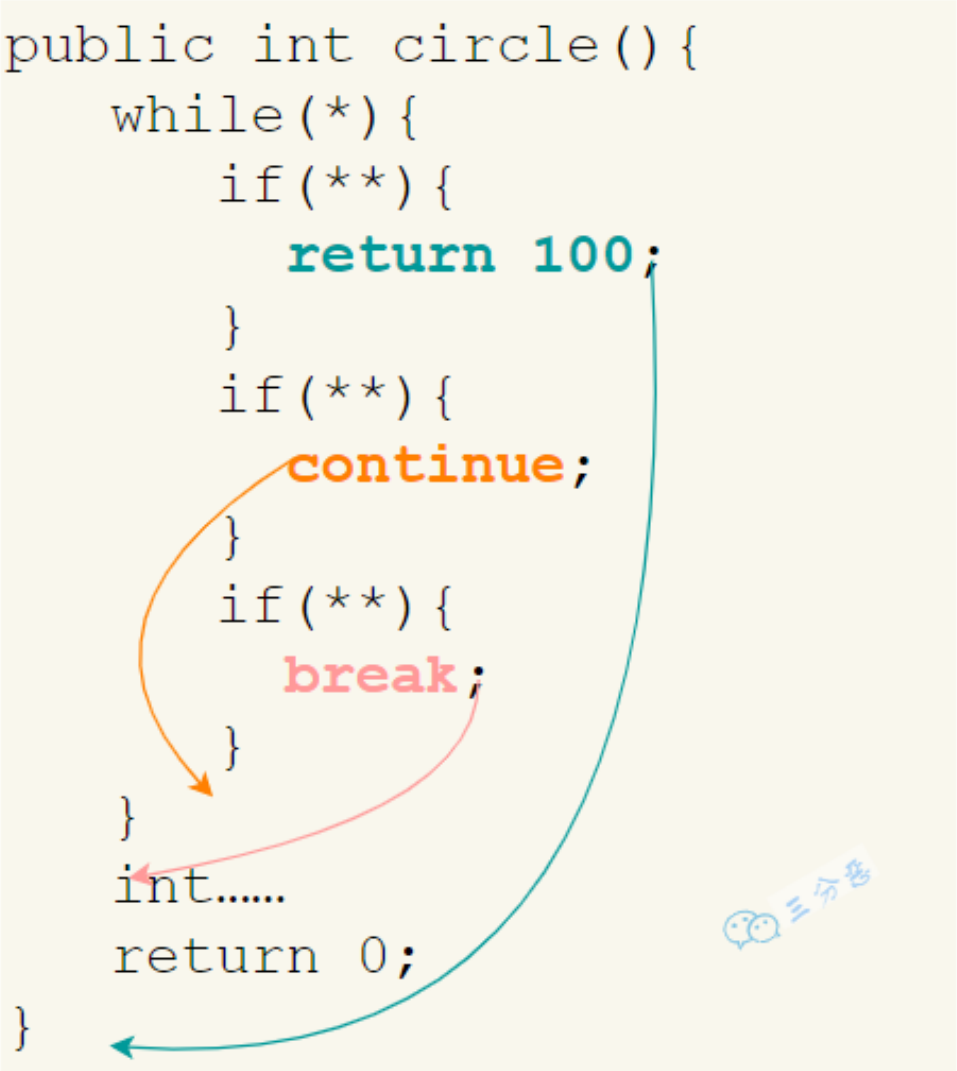
Java5 以前 switch(expr)中，expr 只能是 byte、short、char、int。

从 Java 5 开始，Java 中引入了枚举类型， expr 也可以是 enum 类型。

从 Java 7 开始，expr 还可以是字符串(String)，但是长整型(long)在目前所有的版本中都是不可以的。

12.break ,continue ,return 的区别及作用?

- break 跳出整个循环，不再执行循环(结束当前的循环体)
- continue 跳出本次循环，继续执行下次循环(结束正在执行的循环 进入下一个循环条件)
- return 程序返回，不再执行下面的代码(结束当前的方法 直接返回)



13.用最有效率的方法计算 2 乘以 8?

2 << 3。位运算，数字的二进制位左移三位相当于乘以 2 的三次方。

14.说说自增自减运算？看下这几个代码运行结果?

在写代码的过程中，常见的一种情况是需要某个整数类型变量增加 1 或减少 1，Java 提供了一种特殊的运算符，用于这种表达式，叫做自增运算符（++)和自减运算符（--）。

++和--运算符可以放在变量之前，也可以放在变量之后。

当运算符放在变量之前时(前缀)，先自增/减，再赋值；当运算符放在变量之后时(后缀)，先赋值，再自增/减。

例如，当 `b = ++a` 时，先自增（自己增加 1），再赋值（赋值给 b）；当 `b = a++` 时，先赋值(赋值给 b)，再自增（自己增加 1）。也就是，++a 输出的是 a+1 的值，a++输出的是 a 值。

用一句口诀就是：“符号在前就先加/减，符号在后就后加/减”。

看一下这段代码运行结果？

```
int i  = 1;
i = i++;
System.out.println(i);
```

答案是 1。有点离谱对不对。

对于 JVM 而言，它对自增运算的处理，是会先定义一个临时变量来接收 i 的值，然后进行自增运算，最后又将临时变量赋给了值为 2 的 i，所以最后的结果为 1。

相当于这样的代码：

```
int i = 1;
int temp = i;
i++;
i = temp;
System.out.println(i);
```

这段代码会输出什么？

```
int count = 0;
for(int i = 0;i < 100;i++)
{
   count = count++;
}
System.out.println("count = "+count);
```

答案是 0。

和上面的题目一样的道理，同样是用了临时变量，count 实际是等于临时变量的值。

```
int autoAdd(int count)
{
   int temp = count;
   count = coutn + 1;
   return temp;
}
```

PS：笔试面试可能会碰到的奇葩题，开发这么写，见一次吊一次。

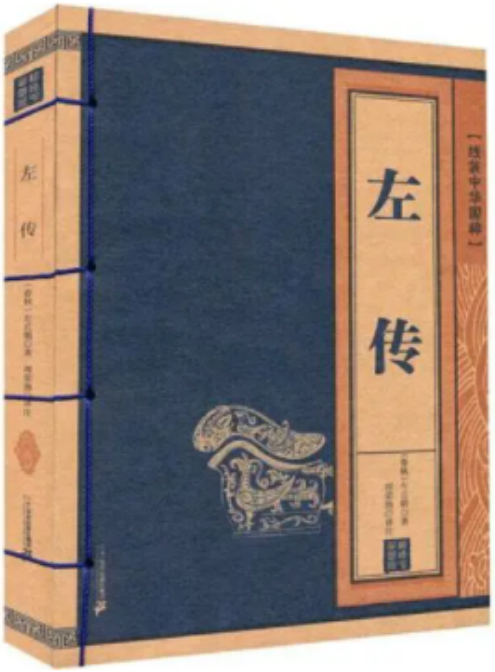
面向对象

15.面向对象和面向过程的区别？

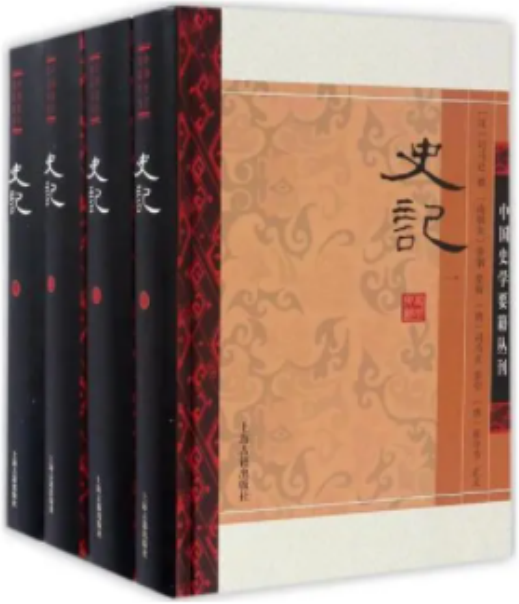
- 面向过程：面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候再一个一个的一次调用就可以。
- 面向对象：面向对象，把构成问题的事务分解成各个对象，而建立对象的目的也不是为了完成一个个步骤，而是为了描述某个事件在解决整个问题的过程所发生的行为。 目的是为了写出通用的代码，加强代码的重用，屏蔽差异性。

用一个比喻：面向过程是编年体；面向对象是纪传体。

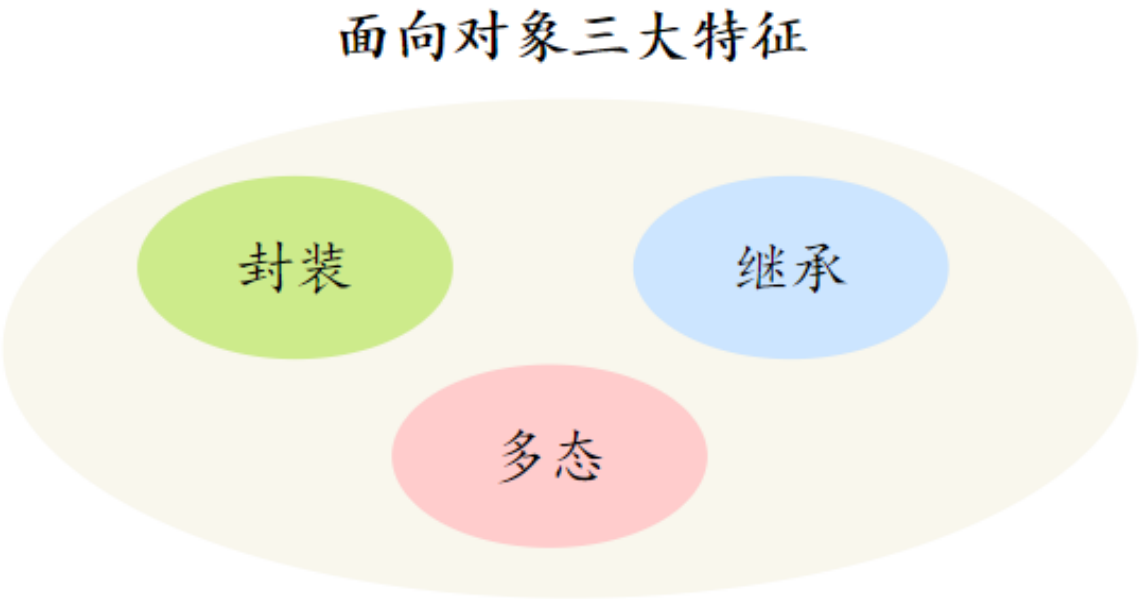
面向过程：
编年体



面向对象：
纪传体



16.面向对象有哪些特性



- 封装

  封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法。

- **继承**

  继承是使用已存在的类的定义作为基础创建新的类，新类的定义可以增加新的属性或新的方法，也可以继承父类的属性和方法。通过继承可以很方便地进行代码复用。

关于继承有以下三个要点：

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。

- **多态**

  所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

  在 Java 中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

17.重载（overload）和重写（override）的区别？

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。

- 重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；
- 重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。

方法重载的规则：

1. 方法名一致，参数列表中参数的顺序，类型，个数不同。
2. 重载与方法的返回值无关，存在于父类和子类，同类中。
3. 可以抛出不同的异常，可以有不同修饰符。

18.访问修饰符 public、private、protected、以及不写（默认）时的区别？

Java 中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

- **default**（即默认，什么也不写）：在同一包内可见，不使用任何修饰符。可以修饰在类、接口、

变量、方法。

- **private**：在同一类内可见。可以修饰变量、方法。注意：不能修饰类（外部类）
- **public**：对所有类可见。可以修饰类、接口、变量、方法
- **protected**：对同一包内的类和所有子类可见。可以修饰变量、方法。注意：不能修饰类（外部类）。

可见性	private	default	protected	public
同一个类中	✓	✓	✓	✓
同一个包中	✗	✓	✓	✓
子类中	✗	✗	✓	✓
全局范围	✗	✗	✗	✓

19.this 关键字有什么作用？

this 是自身的一个对象，代表对象本身，可以理解为：**指向对象本身的一个指针。**

this 的用法在 Java 中大体可以分为 3 种：

1. 普通的直接引用，this 相当于是指向当前对象本身
2. 形参与成员变量名字重名，用 this 来区分：

```
public Person(String name,int age){
    this.name=name;
    this.age=age;
}
```

3. 引用本类的构造函数

20.抽象类(abstract class)和接口(interface)有什么区别？

1. 接口的方法默认是 public ，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法。
2. 接口中除了 static 、 final 变量，不能有其他变量，而抽象类中则不一定。

3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过 extends 关键字扩展多个接口。
4. 接口方法默认修饰符是 public ，抽象方法可以有 public 、 protected 和 default 这些修饰符（抽象方法就是为了被重写所以不能使用 private 关键字修饰！）。
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

1. 在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。
2. jdk9 的接口被允许定义私有方法 。

总结一下 jdk7~jdk9 Java 中接口的变化：

1. 在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。
2. jdk 8 的时候接口可以有默认方法和静态方法功能。
3. jdk 9 在接口中引入了私有方法和私有静态方法。

21.成员变量与局部变量的区别有哪些？

1. **从语法形式上看**：成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public ，private ，static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰。
2. **从变量在内存中的存储方式来看**：如果成员变量是使用 static 修饰的，那么这个成员变量是属于类的，如果没有使用 static 修饰，这个成员变量是属于实例的。对象存于堆内存，如果局部变量类型为基本数据类型，那么存储在栈内存，如果为引用数据类型，那存放的是指向堆内存对象的引用或者是指向常量池中的地址。
3. **从变量在内存中的生存时间上看**：成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. **成员变量如果没有被赋初值**：则会自动以类型的默认值而赋值（一种情况例外:被 final 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值。

22.静态变量和实例变量的区别？静态方法、实例方法呢？

静态变量和实例变量的区别？

静态变量：是被 static 修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个副本。

实例变量：必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

> 静态方法和实例方法有何不同？

类似地。

静态方法：static 修饰的方法，也被称为类方法。在外部调用静态方法时，可以使用"**类名.方法名**"的方式，也可以使用"**对象名.方法名**"的方式。静态方法里不能访问类的非静态成员变量和方法。

实例方法：依存于类的实例，需要使用"**对象名.方法名**"的方式调用；可以访问类的所有成员变量和方法。

24.final 关键字有什么作用？

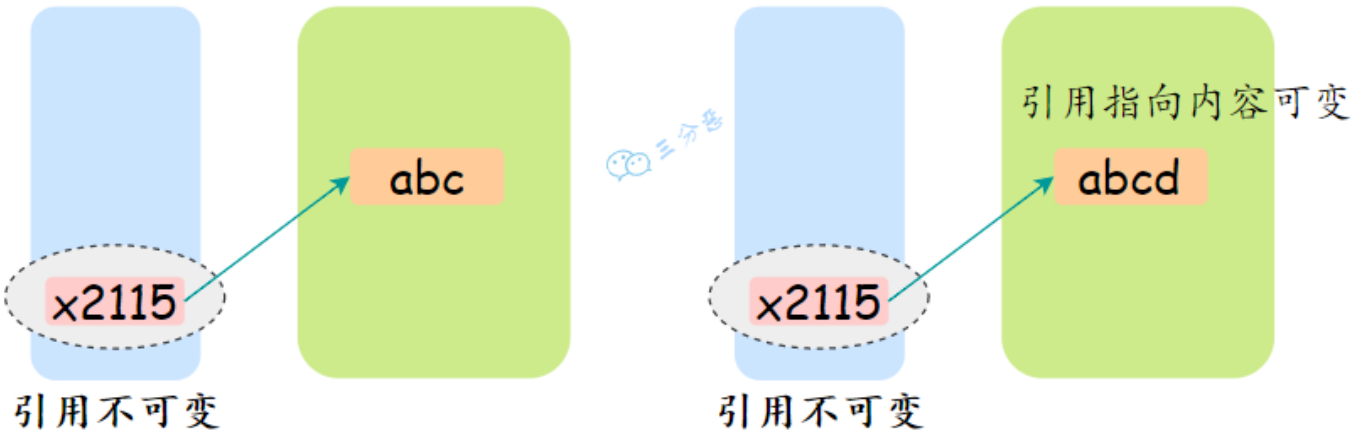
final 表示不可变的意思，可用于修饰类、属性和方法：

- 被 final 修饰的类不可以被继承
- 被 final 修饰的方法不可以被重写
- 被 final 修饰的变量不可变，被 final 修饰的变量必须被显式第指定初始值，还得注意的是，这里的不可变指的是变量的引用不可变，不是引用指向的内容的不可变。

  例如：

```
final StringBuilder sb = new StringBuilder("abc");
sb.append("d");
System.out.println(sb); //abcd
```

一张图说明：



25.final、finally、finalize 的区别？

- final 用于修饰变量、方法和类：final 修饰的类不可被继承；修饰的方法不可被重写；修饰的变量不可变。
- finally 作为异常处理的一部分，它只能在 `try/catch` 语句中，并且附带一个语句块表示这段语句最终一定被执行（无论是否抛出异常），经常被用在需要释放资源的情况下，`System.exit (0)` 可以阻断 finally 执行。
- finalize 是在 `java.lang.Object` 里定义的方法，也就是说每一个对象都有这么个方法，这个方法在 `gc` 启动，该对象被回收的时候被调用。

  一个对象的 finalize 方法只会被调用一次，finalize 被调用不一定会立即回收该对象，所以有可能调用 finalize 后，该对象又不需要被回收了，然后到了真正要被回收的时候，因为前面调用过一次，所以不会再次调用 finalize 了，进而产生问题，因此不推荐使用 finalize 方法。

26.==和 equals 的区别？

==：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象(基本数据类型 == 比较的是值，引用数据类型 == 比较的是内存地址)。

equals()：它的作用也是判断两个对象是否相等。但是这个“相等”一般也分两种情况：

- 默认情况：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象，还是相当于比较内存地址。
- 自定义情况：类覆盖了 equals() 方法。我们平时覆盖的 equals()方法一般是比较两个对象的内容是否相同，自定义了一个相等的标准，也就是两个对象的值是否相等。

举个例子，Person，我们认为两个人的编号和姓名相同，就是一个人：

```
public class Person {
```

```
    private String no;
    private String name;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person)) return false;
        Person person = (Person) o;
        return Objects.equals(no, person.no) &&
                Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(no, name);
    }
}
```

27.hashCode 与 equals?

这个也是面试常问——“你重写过 hashcode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”

什么是 HashCode?

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数，定义在 Object 类中， 是一个本地方法，这个方法通常用来将对象的内存地址转换为整数之后返回。

```
public native int hashCode();
```

哈希码主要在哈希表这类集合映射的时候用到，哈希表存储的是键值对(key-value)，它的特点是：能根据“键”快速的映射到对应的“值”。这其中就利用到了哈希码！

为什么要有 hashCode?

上面已经讲了，主要是在哈希表这种结构中用的到。

例如 HashMap 怎么把 key 映射到对应的 value 上呢？用的就是哈希取余法，也就是拿哈希码和存储元素的数组的长度取余，获取 key 对应的 value 所在的下标位置。

为什么重写 quals 时必须重写 hashCode 方法?

如果两个对象相等，则 hashcode 一定也是相同的。两个对象相等，对两个对象分别调用 equals 方法都返回 true。反之，两个对象有相同的 hashcode 值，它们也不一定是相等的 。因此，**equals** 方法被覆盖过，则 **hashCode** 方法也必须被覆盖。

hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode() ，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

> 为什么两个对象有相同的 hashcode 值，它们也不一定是相等的？

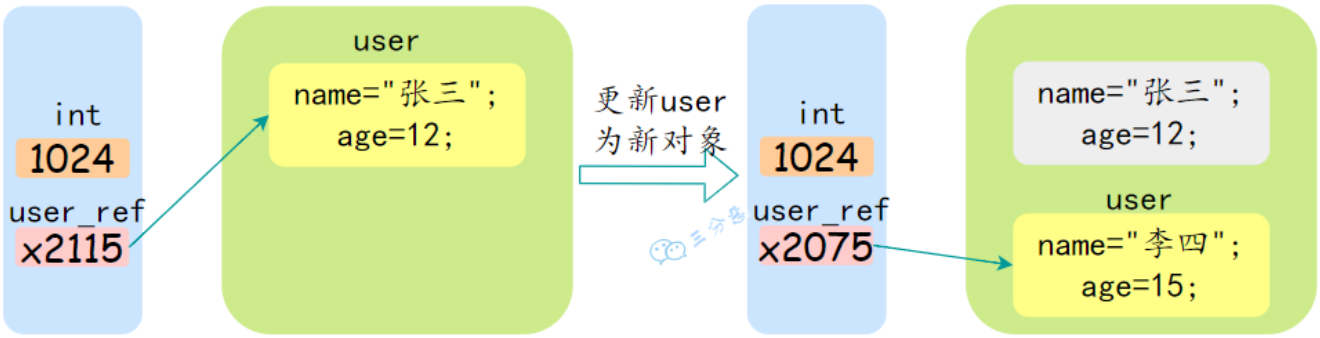
因为可能会碰撞， hashCode() 所使用的散列算法也许刚好会让多个对象传回相同的散列值。越糟糕的散列算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 hashCode ）。

28.Java 是值传递，还是引用传递？

Java 语言是值传递。Java 语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。

JVM 的内存分为堆和栈，其中栈中存储了基本数据类型和引用数据类型实例的地址，也就是对象地址。

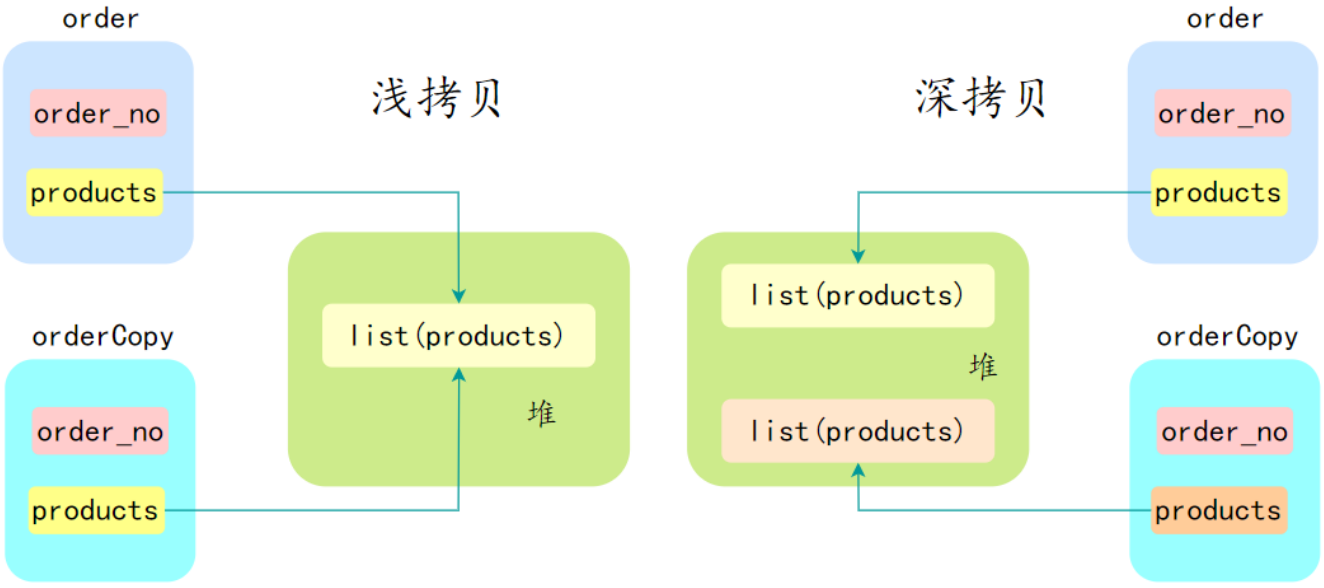
而对象所占的空间是在堆中开辟的，所以传递的时候可以理解为把变量存储的对象地址给传递过去，因此引用类型也是值传递。



29.深拷贝和浅拷贝？

- 浅拷贝：仅拷贝被拷贝对象的成员变量的值，也就是基本数据类型变量的值，和引用数据类型变量的地址值，而对于引用类型变量指向的堆中的对象不会拷贝。
- 深拷贝：完全拷贝一个对象，拷贝被拷贝对象的成员变量的值，堆中的对象也会拷贝一份。

例如现在有一个 order 对象，里面有一个 products 列表，它的浅拷贝和深拷贝的示意图：



因此深拷贝是安全的，浅拷贝的话如果有引用类型，那么拷贝后对象，引用类型变量修改，会影响原对象。

> 浅拷贝如何实现呢？

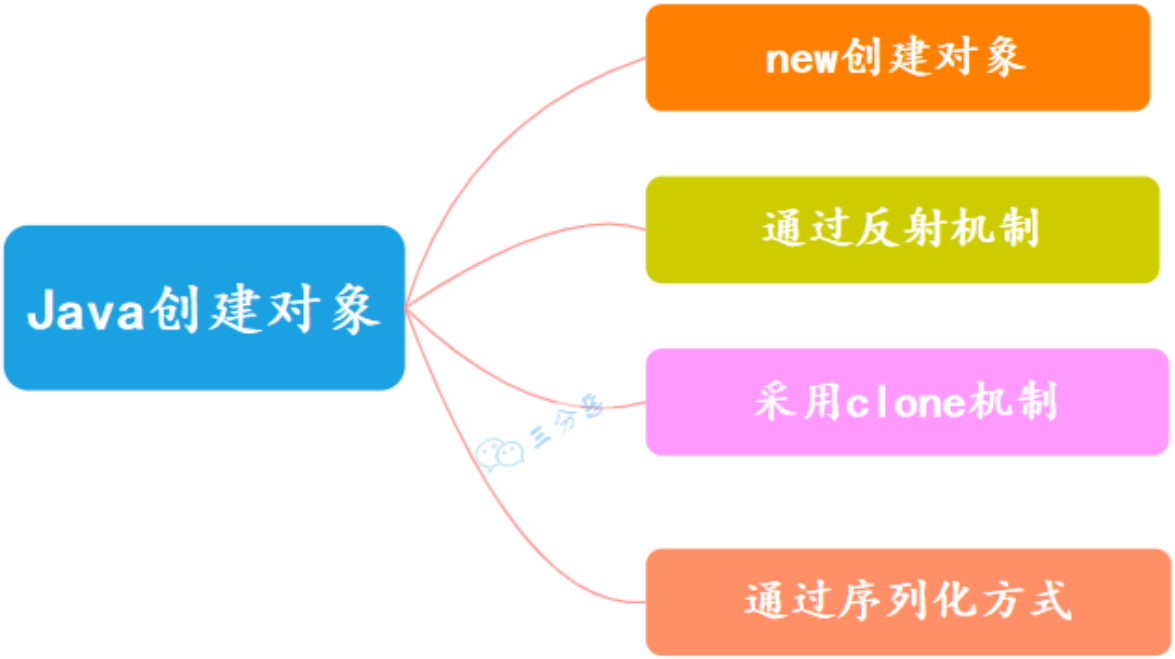
Object 类提供的 clone()方法可以非常简单地实现对象的浅拷贝。

> 深拷贝如何实现呢？

- 重写克隆方法：重写克隆方法，引用类型变量单独克隆，这里可能会涉及多层递归。
- 序列化：可以先将原对象序列化，再反序列化成拷贝对象。

30.Java 创建对象有哪几种方式？

Java 中有以下四种创建对象的方式：



- new 创建新对象
- 通过反射机制
- 采用 clone 机制
- 通过序列化机制

前两者都需要显式地调用构造方法。对于 clone 机制,需要注意浅拷贝和深拷贝的区别，对于序列化机制需要明确其实现原理，在 Java 中序列化可以通过实现 Externalizable 或者 Serializable 来实现。

String

31.String 是 Java 基本数据类型吗？可以被继承吗？

String 是 Java 基本数据类型吗？

不是。Java 中的基本数据类型只有 8 个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type），剩下的都是引用类型（reference type）。

String 是一个比较特殊的引用数据类型。

String 类可以继承吗？

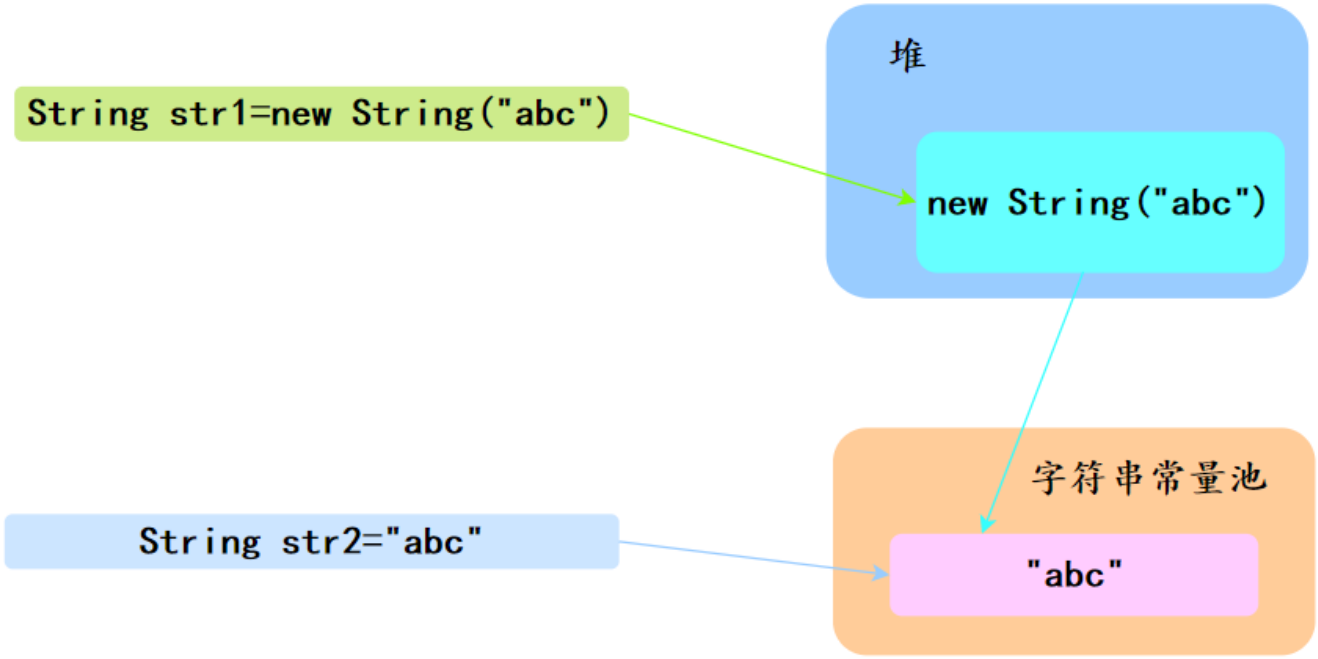
不行。String 类使用 final 修饰，是所谓的不可变类，无法被继承。

32.String 和 StringBuilder、StringBuffer 的区别？

- String：String 的值被创建后不能修改，任何对 String 的修改都会引发新的 String 对象的生成。
- StringBuffer：跟 String 类似，但是值可以被修改，使用 synchronized 来保证线程安全。
- StringBuilder：StringBuffer 的非线程安全版本，性能上更高一些。

33.String str1 = new String("abc")和 String str2 = "abc" 和 区别？

两个语句都会去字符串常量池中检查是否已经存在“abc”，如果有则直接使用，如果没有则会在常量池中创建“abc”对象。



但是不同的是，String str1 = new String("abc") 还会通过 new String() 在堆里创建一个 "abc" 字符串对象实例。所以后者可以理解为被前者包含。

```
String s = new String("abc")创建了几个对象？
```

很明显，一个或两个。如果字符串常量池已经有“abc”，则是一个；否则，两个。

当字符创常量池没有“abc”，此时会创建如下两个对象：

- 一个是字符串字面量 "abc" 所对应的、字符串常量池中的实例
- 另一个是通过 new String() 创建并初始化的，内容与"abc"相同的实例，在堆中。

34.String 不是不可变类吗？字符串拼接是如何实现的？

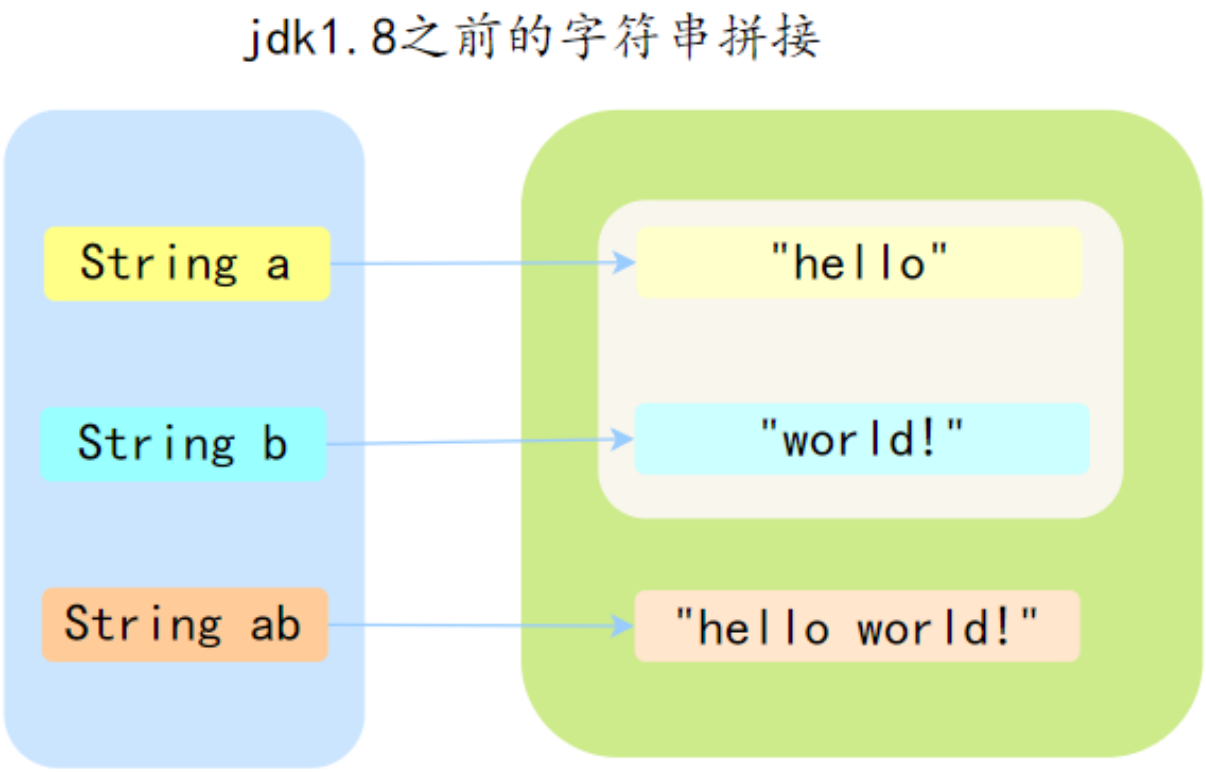
String 的确是不可变的，“+”的拼接操作，其实是会生成新的对象。

例如：

```
String a = "hello ";
String b = "world!";
String ab = a + b;
```

在**jdk1.8** 之前，a 和 b 初始化时位于字符串常量池，ab 拼接后的对象位于堆中。经过拼接新生成了 String 对象。如果拼接多次，那么会生成多个中间对象。

内存如下：



在**Java8** 时JDK 对“+”号拼接进行了优化，上面所写的拼接方式会被优化为基于 StringBuilder 的 append 方法进行处理。Java 会在编译期对“+”号进行处理。

下面是通过 javap -verbose 命令反编译字节码的结果，很显然可以看到 StringBuilder 的创建和 append 方法的调用。

```
stack=2, locals=4, args_size=1
    0: ldc           #2                  // String hello
    2: astore_1
    3: ldc           #3                  // String world!
    5: astore_2
    6: new           #4                  // class java/lang/StringBuilder
    9: dup
   10: invokespecial #5                  // Method java/lang/StringBuilder."<init>":()V
   13: aload_1
   14: invokevirtual #6                  // Method java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
   17: aload_2
```

```
   18: invokevirtual #6                  // Method java/lang/StringBuilder.append:
(Ljava/lang/String;)Ljava/lang/StringBuilder;
   21: invokevirtual #7                  // Method java/lang/StringBuilder.toString:
()Ljava/lang/String;
   24: astore_3
   25: return
```

也就是说其实上面的代码其实相当于：

```
String a = "hello ";
String b = "world!";
StringBuilder sb = new StringBuilder();
sb.append(a);
sb.append(b);
String ab = sb.toString();
```

此时，如果再笼统的回答：通过加号拼接字符串会创建多个 String 对象，因此性能比 StringBuilder 差，就是错误的了。因为本质上加号拼接的效果最终经过编译器处理之后和 StringBuilder 是一致的。

当然，循环里拼接还是建议用 StringBuilder，为什么，因为循环一次就会创建一个新的 StringBuilder 对象，大家可以自行实验。

35.intern 方法有什么作用？

JDK 源码里已经对这个方法进行了说明：

```
     * <p>
     * When the intern method is invoked, if the pool already contains a
     * string equal to this {@code String} object as determined by
     * the {@link #equals(Object)} method, then the string from the pool is
     * returned. Otherwise, this {@code String} object is added to the
     * pool and a reference to this {@code String} object is returned.
     * <p>
```

意思也很好懂：

- 如果当前字符串内容存在于字符串常量池（即 equals()方法为 true，也就是内容一样），直接返回字符串常量池中的字符串
- 否则，将此 String 对象添加到池中，并返回 String 对象的引用

Integer

36.Integer a= 127，Integer b = 127；Integer c= 128，Integer d = 128；，相等吗?

答案是 a 和 b 相等，c 和 d 不相等。

- 对于基本数据类型==比较的值
- 对于引用数据类型==比较的是地址

Integer a= 127 这种赋值，是用到了 Integer 自动装箱的机制。自动装箱的时候会去缓存池里取 Integer 对象，没有取到才会创建新的对象。

如果整型字面量的值在-128 到 127 之间，那么自动装箱时不会 new 新的 Integer 对象，而是直接引用缓存池中的 Integer 对象，超过范围 a1==b1 的结果是 false

```java
public static void main(String[] args) {
    Integer a = 127;
    Integer b = 127;
    Integer b1 = new Integer(127);
    System.out.println(a == b); //true
    System.out.println(b==b1); //false

    Integer c = 128;
    Integer d = 128;
    System.out.println(c == d); //false
}
```

什么是 Integer 缓存?

因为根据实践发现大部分的数据操作都集中在值比较小的范围，因此 Integer 搞了个缓存池，默认范围是 -128 到 127，可以根据通过设置`JVM-XX:AutoBoxCacheMax=`来修改缓存的最大值，最小值改不了。

实现的原理是 int 在自动装箱的时候会调用 Integer.valueOf，进而用到了 IntegerCache。

```java
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

很简单，就是判断下值是否在缓存范围之内，如果是的话去 IntegerCache 中取，不是的话就创建一个新的 Integer 对象。

IntegerCache 是一个静态内部类， 在静态块中会初始化好缓存值。

```
private static class IntegerCache {
    ……
    static {
         //创建Integer对象存储
         for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);
       ……
    }
}
```

37.String 怎么转成 Integer 的？原理？

PS:这道题印象中在一些面经中出场过几次。

String 转成 Integer，主要有两个方法：

- Integer.parseInt(String s)
- Integer.valueOf(String s)

不管哪一种，最终还是会调用 Integer 类内中的`parseInt(String s, int radix)`方法。

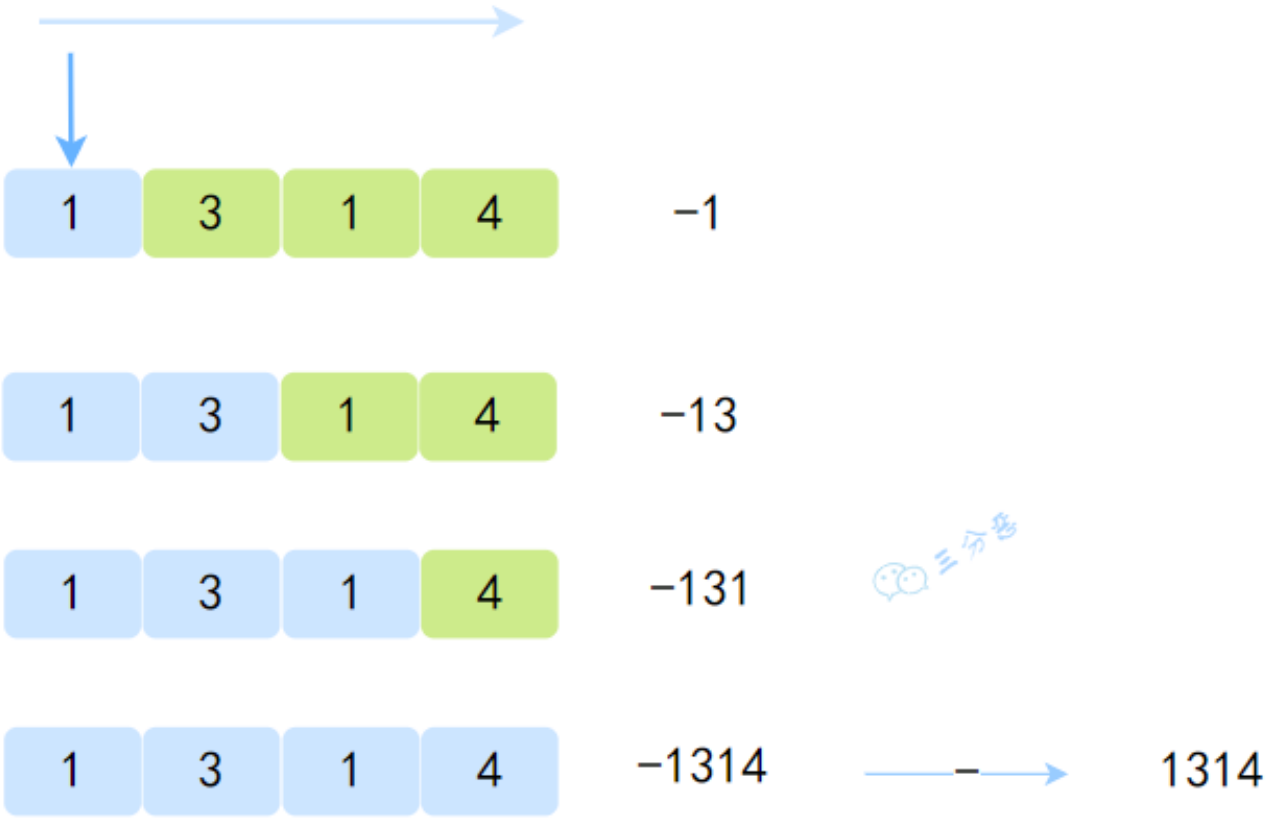
抛去一些边界之类的看看核心代码：

```
public static int parseInt(String s, int radix)
            throws NumberFormatException
   {

       int result = 0;
       //是否是负数
       boolean negative = false;
       //char字符数组下标和长度
       int i = 0, len = s.length();
       ……
       int digit;
       //判断字符长度是否大于0，否则抛出异常
       if (len > 0) {
          ……
          while (i < len) {
             // Accumulating negatively avoids surprises near MAX_VALUE
             //返回指定基数中字符表示的数值。（此处是十进制数值）
             digit = Character.digit(s.charAt(i++),radix);
             //进制位乘以数值
             result *= radix;
             result -= digit;
          }
```

```
        }
        //根据上面得到的是否负数，返回相应的值
        return negative ? result : -result;
    }
```

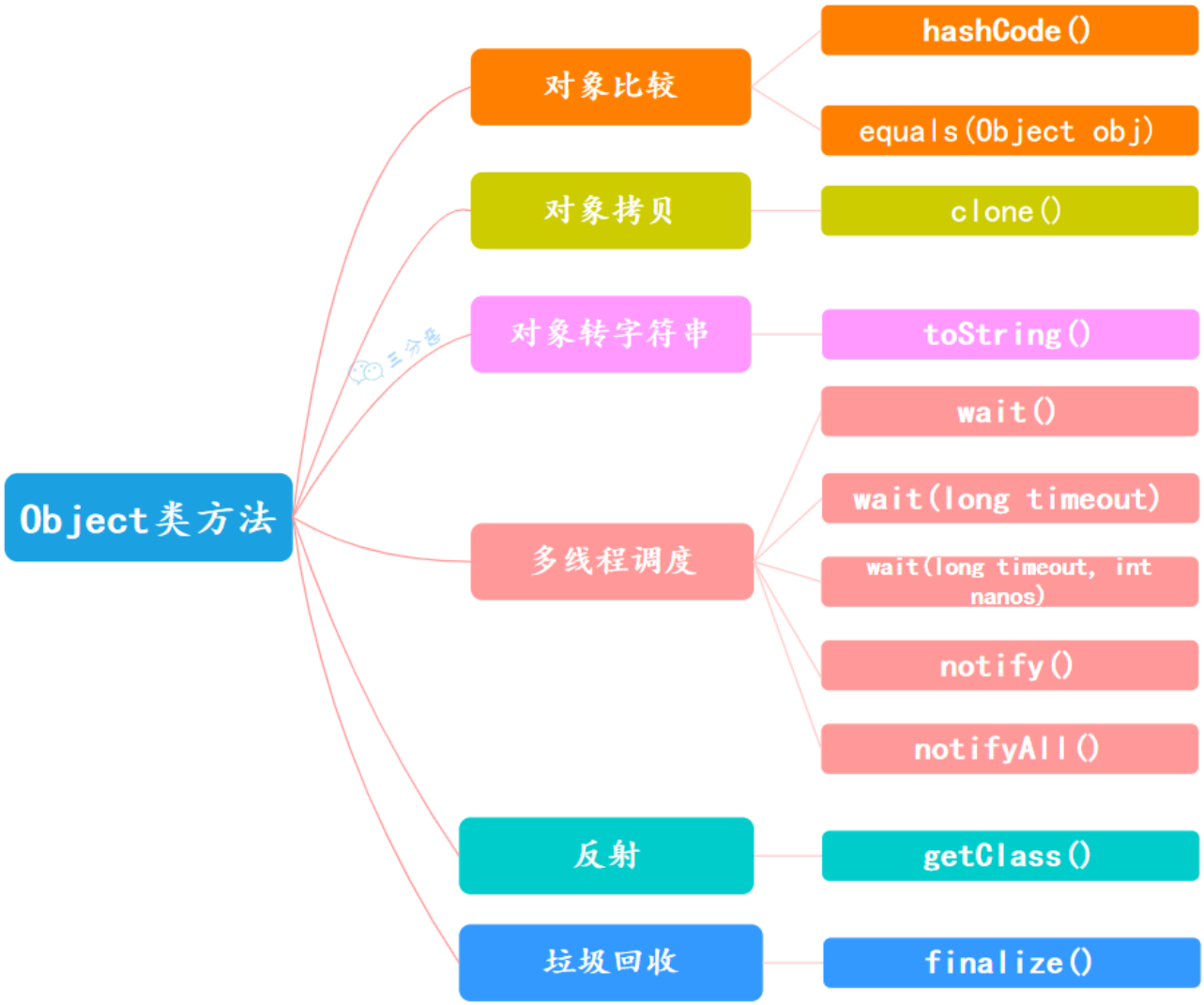
去掉枝枝蔓蔓（当然这些枝枝蔓蔓可以去看看，源码 cover 了很多情况），其实剩下的就是一个简单的字符串遍历计算，不过计算方式有点反常规，是用负的值累减。



Object

38.Object 类的常见方法?

Object 类是一个特殊的类，是所有类的父类，也就是说所有类都可以调用它的方法。它主要提供了以下 11 个方法，大概可以分为六类：



对象比较：

- public native int hashCode() ：native 方法，用于返回对象的哈希码，主要使用在哈希表中，比如 JDK 中的 HashMap。
- public boolean equals(Object obj)：用于比较 2 个对象的内存地址是否相等，String 类对该方法进行了重写用户比较字符串的值是否相等。

对象拷贝：

- protected native Object clone() throws CloneNotSupportedException：naitive 方法，用于创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 x.clone() != x 为 true，x.clone().getClass() == x.getClass() 为 true。Object 本身没有实现 Cloneable 接口，所以不重写 clone 方法并且进行调用的话会发生 CloneNotSupportedException 异常。

对象转字符串：

- public String toString()：返回类的名字@实例的哈希码的 16 进制的字符串。建议 Object 所有

的子类都重写这个方法。

多线程调度：

- public final native void notify()：native 方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。
- public final native void notifyAll()：native 方法，并且不能重写。跟 notify 一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。
- public final native void wait(long timeout) throws InterruptedException：native 方法，并且不能重写。暂停线程的执行。注意：sleep 方法没有释放锁，而 wait 方法释放了锁 。timeout 是等待时间。
- public final void wait(long timeout, int nanos) throws InterruptedException：多了 nanos 参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。 所以超时的时间还需要加上 nanos 毫秒。
- public final void wait() throws InterruptedException：跟之前的 2 个 wait 方法一样，只不过该方法一直等待，没有超时时间这个概念

反射：

- public final native Class<?> getClass()：native 方法，用于返回当前运行时对象的 Class 对象，使用了 final 关键字修饰，故不允许子类重写。

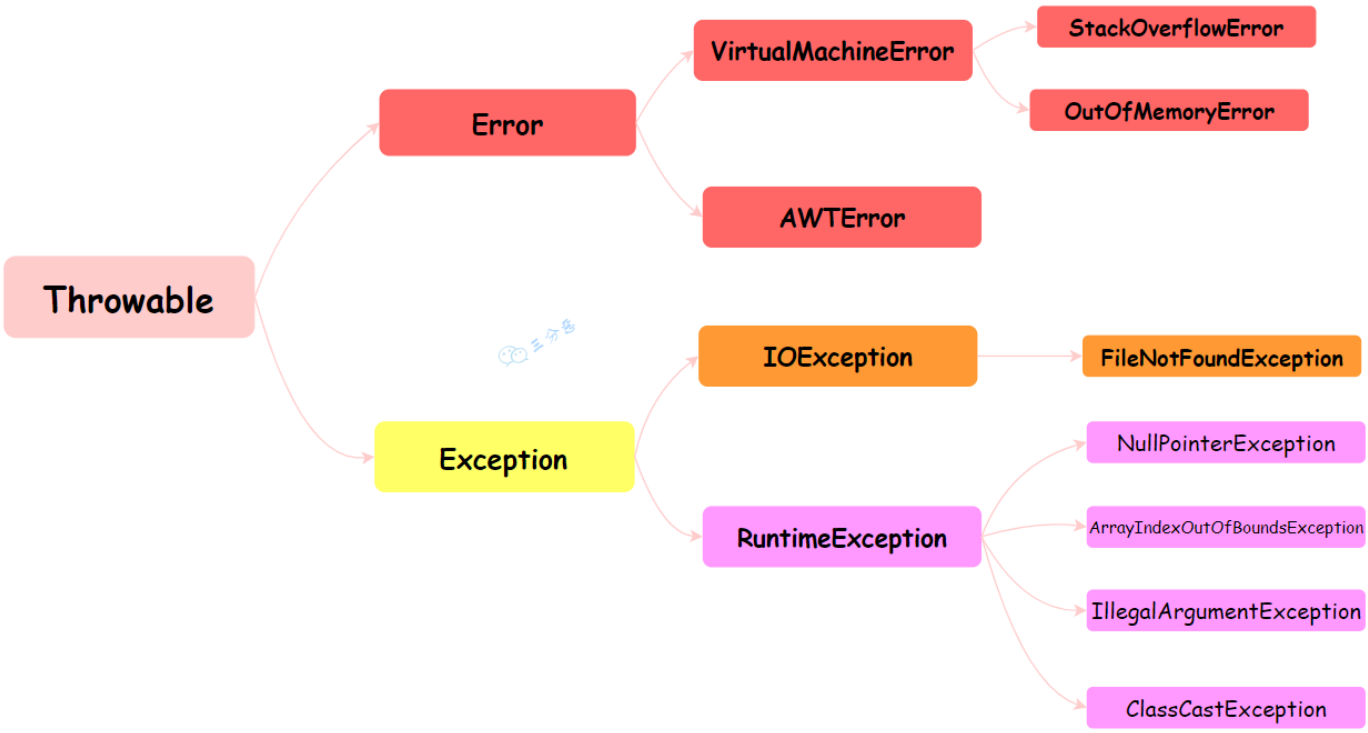
垃圾回收：

- protected void finalize() throws Throwable ： 通知垃圾收集器回收对象。

异常处理

39.Java 中异常处理体系?

Java 的异常体系是分为多层的。

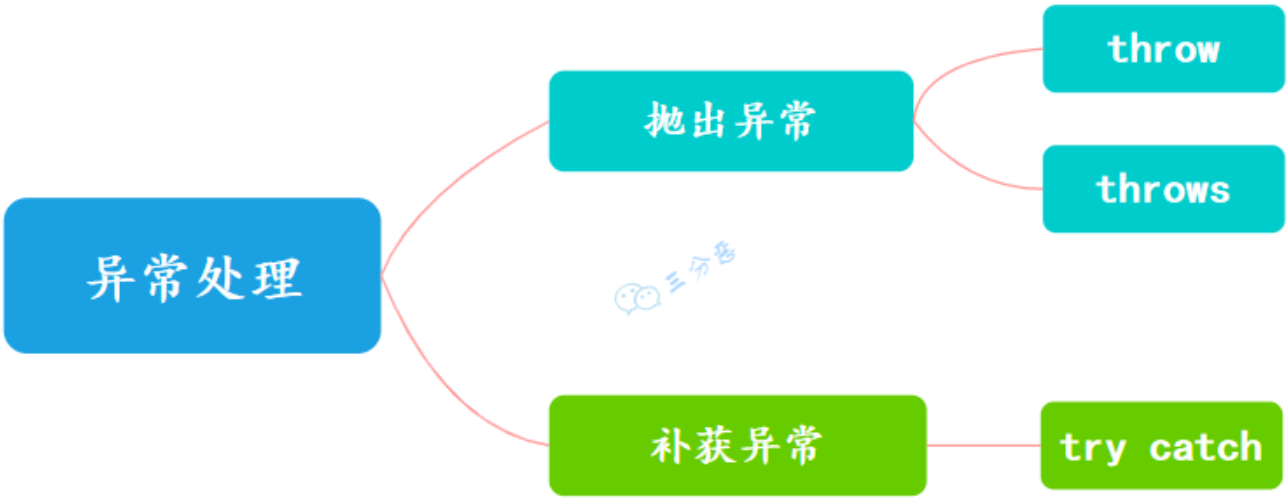


`Throwable`是 Java 语言中所有错误或异常的基类。 Throwable 又分为`Error`和`Exception`，其中 Error 是系统内部错误，比如虚拟机异常，是程序无法处理的。`Exception`是程序问题导致的异常，又分为两种：

- CheckedException 受检异常：编译器会强制检查并要求处理的异常。
- RuntimeException 运行时异常：程序运行中出现异常，比如我们熟悉的空指针、数组下标越界等等

40.异常的处理方式？

针对异常的处理主要有两种方式：



- **遇到异常不进行具体处理，而是继续抛给调用者 （throw，throws）**

抛出异常有三种形式，一是 throw,一个 throws，还有一种系统自动抛异常。

throws 用在方法上，后面跟的是异常类，可以跟多个；而 throw 用在方法内，后面跟的是异常对象。

- **try catch 捕获异常**

在 catch 语句块中补货发生的异常，并进行处理。

```
    try {
        //包含可能会出现异常的代码以及声明异常的方法
     }catch(Exception e) {
        //捕获异常并进行处理
     }finally {                                          }
        //可选，必执行的代码
     }
```

try-catch 捕获异常的时候还可以选择加上 finally 语句块，finally 语句块不管程序是否正常执行，最终它都会必然执行。

41.三道经典异常处理代码题

题目 1

```
public class TryDemo {
    public static void main(String[] args) {
        System.out.println(test());
    }
    public static int test() {
        try {
            return 1;
        } catch (Exception e) {
            return 2;
        } finally {
            System.out.print("3");
        }
    }
}
```

执行结果：31。

try、catch。finally 的基础用法，在 return 前会先执行 finally 语句块，所以是先输出 finally 里的 3，再输出 return 的 1。

题目 2

```
public class TryDemo {
    public static void main(String[] args) {
        System.out.println(test1());
    }
    public static int test1() {
        try {
            return 2;
        } finally {
            return 3;
        }
    }
}
```

执行结果：3。

try 返回前先执行 finally，结果 finally 里不按套路出牌，直接 return 了，自然也就走不到 try 里面的 return 了。

finally 里面使用 return 仅存在于面试题中，实际开发这么写要挨吊的。

题目 3

```
public class TryDemo {
    public static void main(String[] args) {
        System.out.println(test1());
    }
    public static int test1() {
        int i = 0;
        try {
            i = 2;
            return i;
        } finally {
            i = 3;
        }
    }
}
```

执行结果：2。

大家可能会以为结果应该是 3，因为在 return 前会执行 finally，而 i 在 finally 中被修改为 3 了，那最终返回 i 不是应该为 3 吗?

但其实，在执行 finally 之前，JVM 会先将 i 的结果暂存起来，然后 finally 执行完毕后，会返回之前暂存的结果，而不是返回 i，所以即使 i 已经被修改为 3，最终返回的还是之前暂存起来的结果 2。

I/O

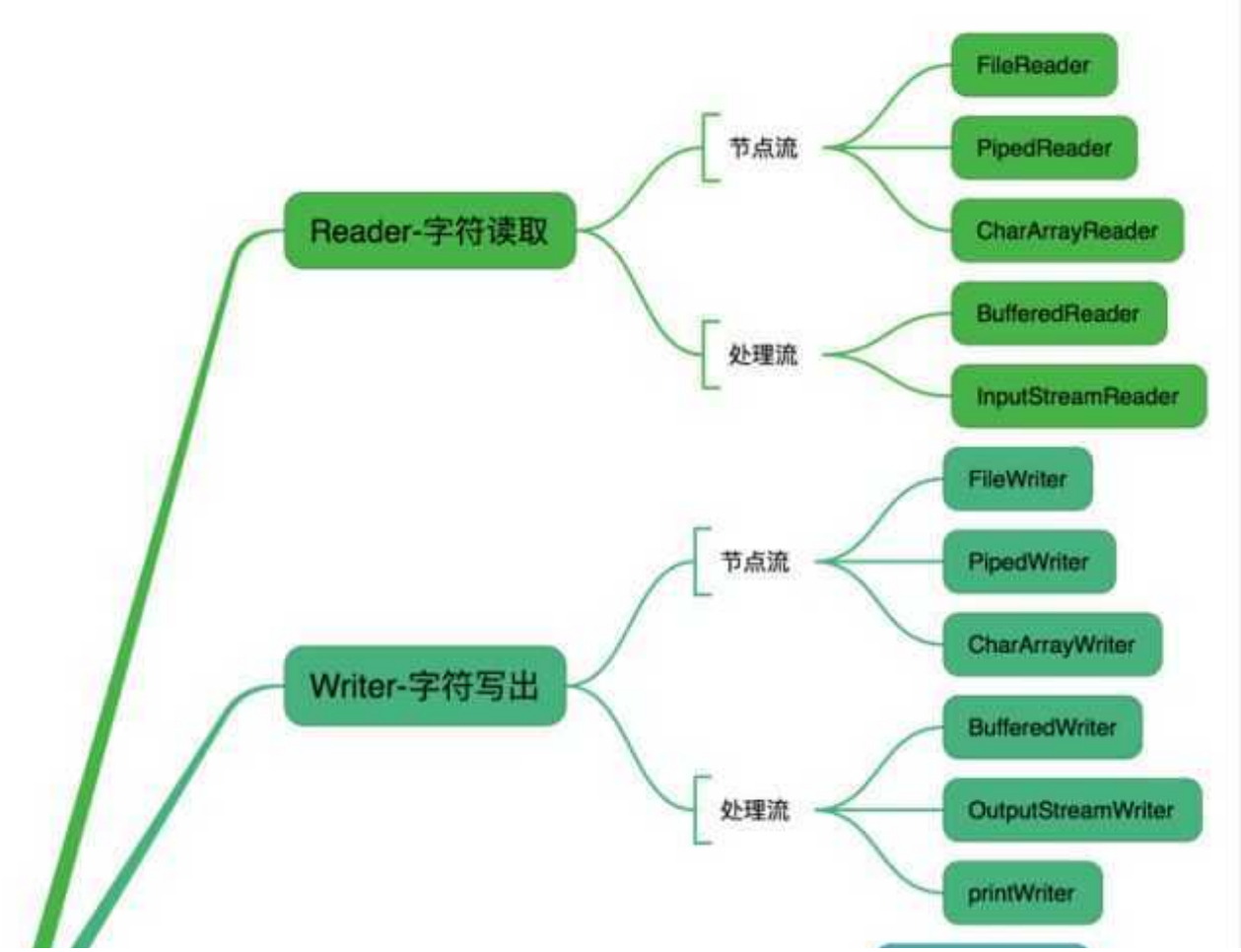
42.Java 中 IO 流分为几种?

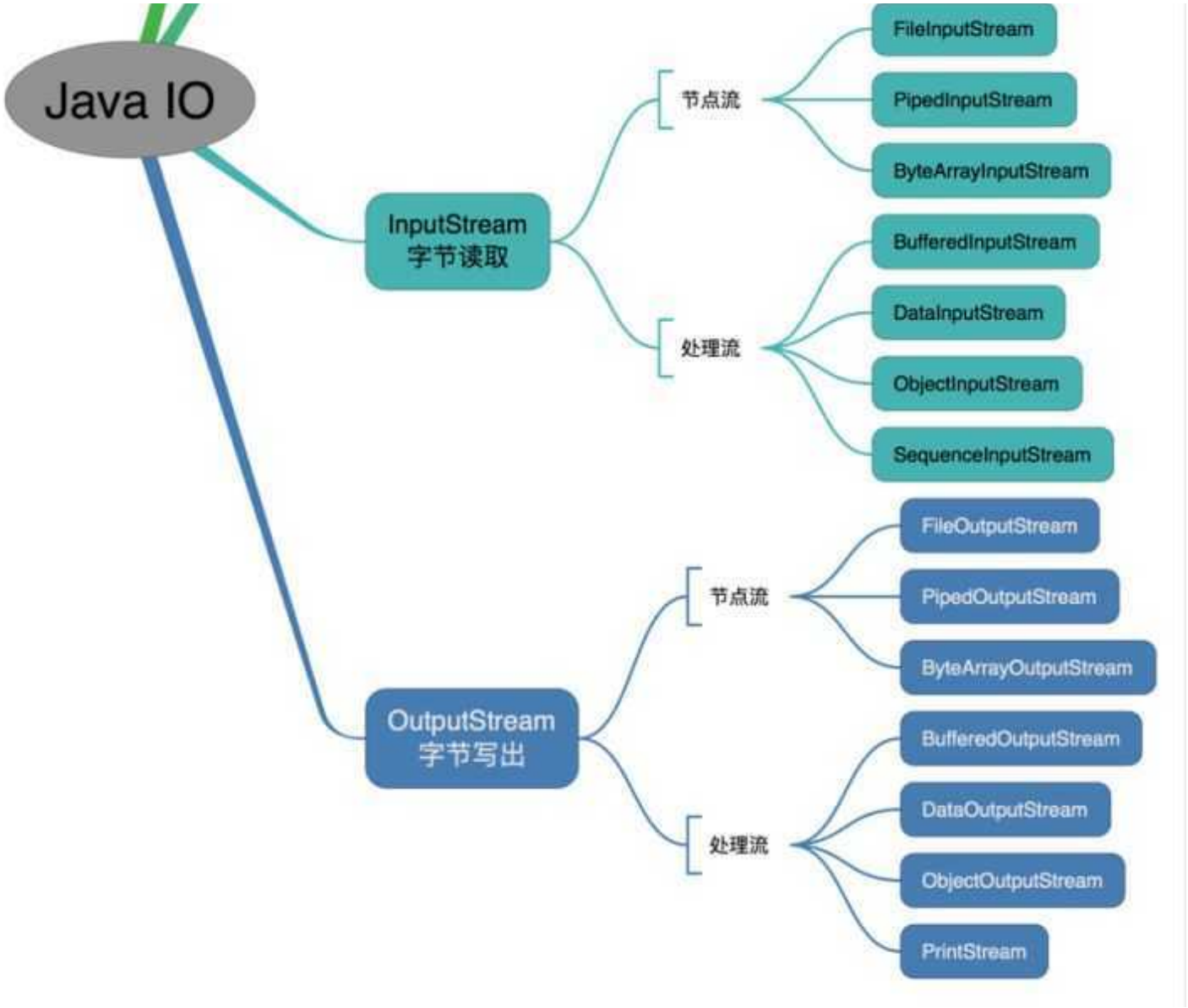
流按照不同的特点，有很多种划分方式。

- 按照流的流向分，可以分为输入流和输出流;
- 按照操作单元划分，可以划分为字节流和字符流;
- 按照流的角色划分为节点流和处理流

Java Io 流共涉及 40 多个类，看上去杂乱，其实都存在一定的关联，Java IO 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

- **InputStream/Reader**: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- **OutputStream/Writer**: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

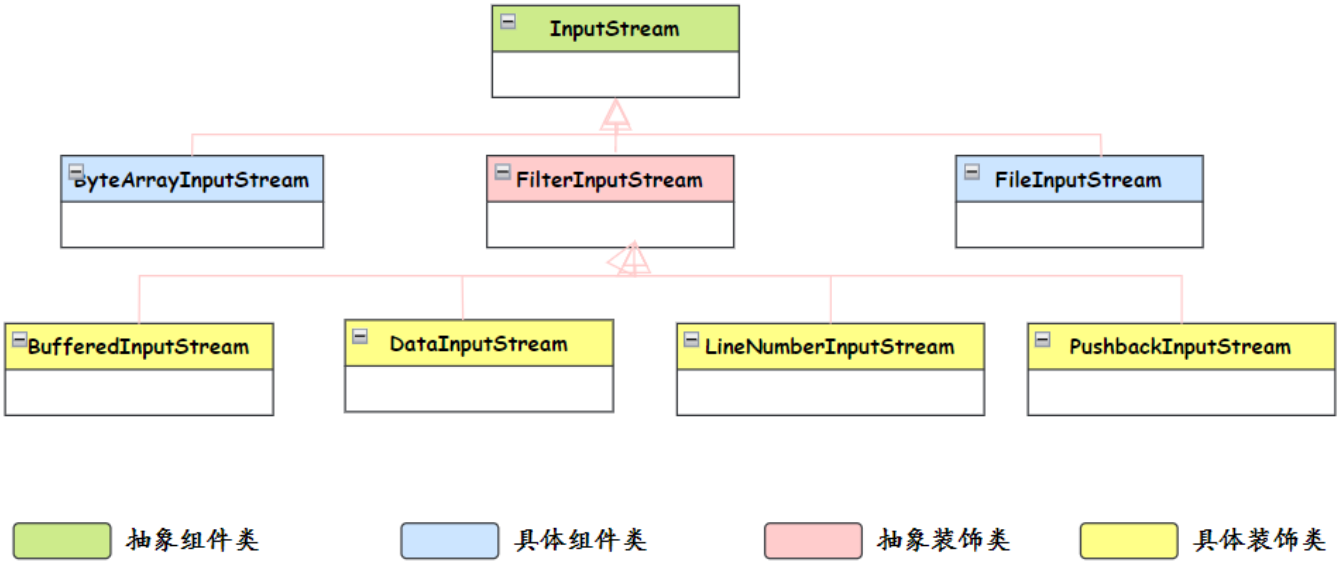




IO 流用到了什么设计模式?

其实，Java 的 IO 流体系还用到了一个设计模式——**装饰器模式**。

InputStream 相关的部分类图如下，篇幅有限，装饰器模式就不展开说了。

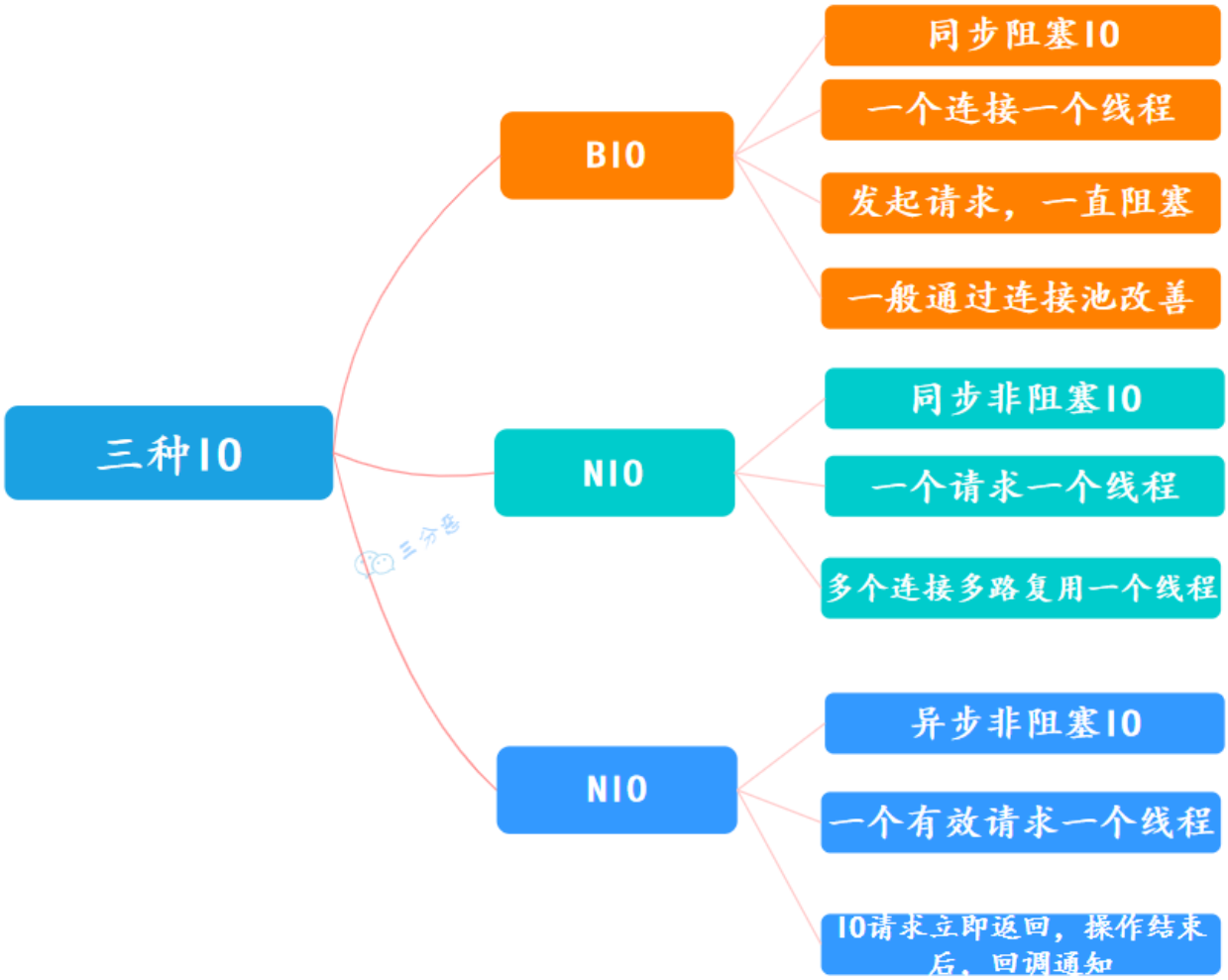


43.既然有了字节流,为什么还要有字符流?

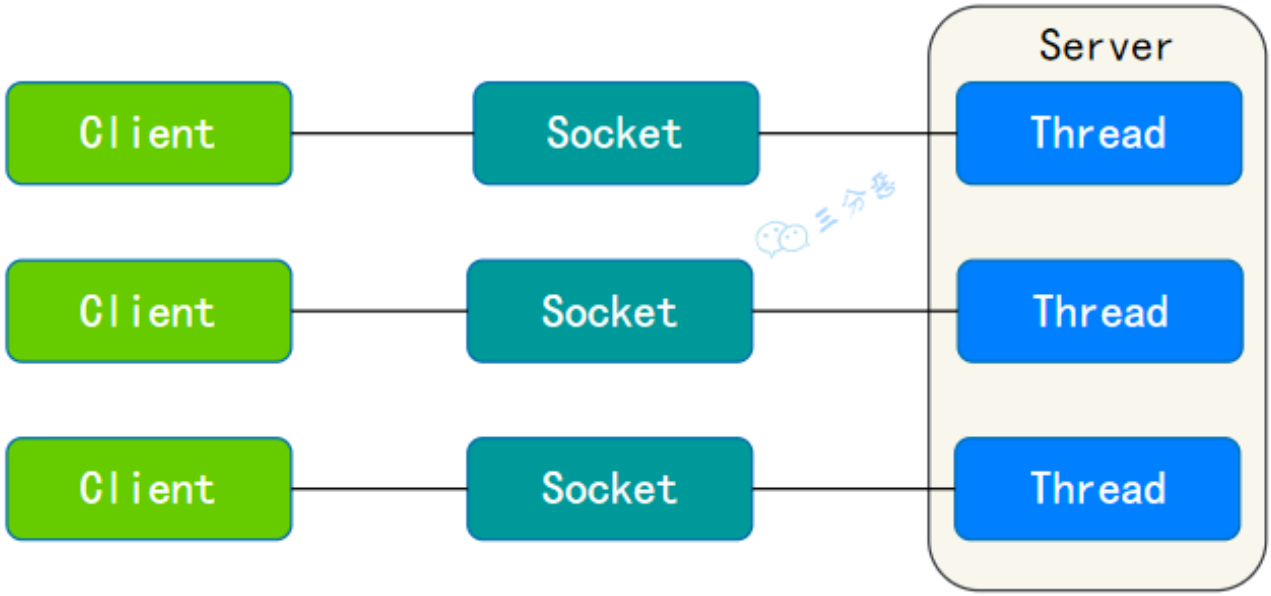
其实字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还比较耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。

所以， I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

44.BIO、NIO、AIO?



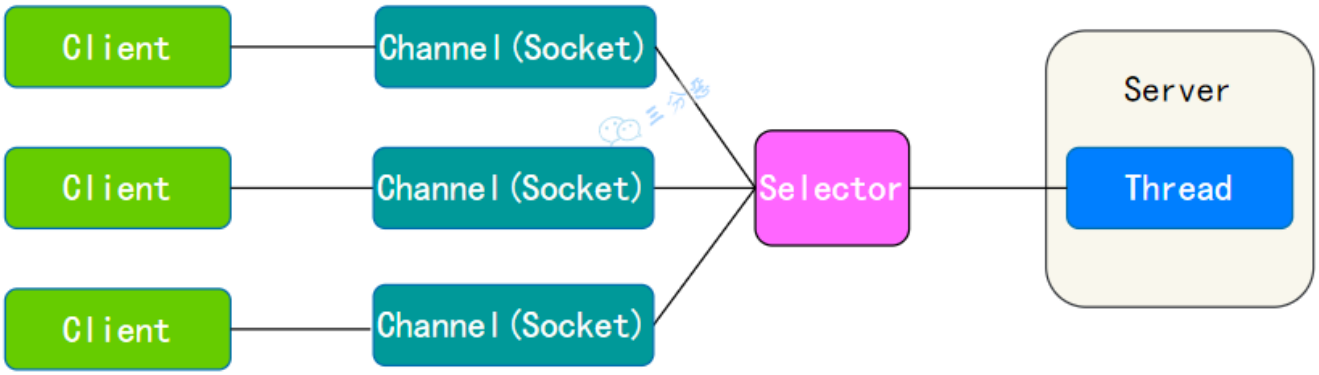
BIO(blocking I/O)： 就是传统的 IO，同步阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，可以通过连接池机制改善(实现多个客户连接服务器)。



BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，程序简单易理解。

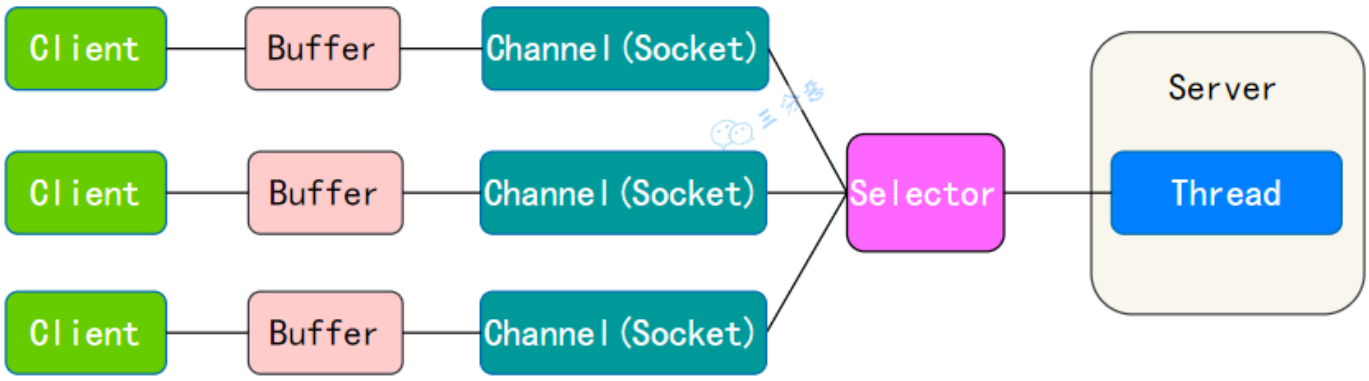
NIO ：全称 java non-blocking IO，是指 JDK 提供的新 API。从 JDK1.4 开始，Java 提供了一系列改进的输入/输出的新特性，被统称为 NIO(即 New IO)。

NIO 是同步非阻塞的，服务器端用一个线程处理多个连接，客户端发送的连接请求会注册到多路复用器上，多路复用器轮询到连接有 IO 请求就进行处理：



NIO 的数据是面向缓冲区 **Buffer**的，必须从 Buffer 中读取或写入。

所以完整的 NIO 示意图：



可以看出，NIO 的运行机制：

- 每个 Channel 对应一个 Buffer。
- Selector 对应一个线程，一个线程对应多个 Channel。
- Selector 会根据不同的事件，在各个通道上切换。
- Buffer 是内存块，底层是数据。

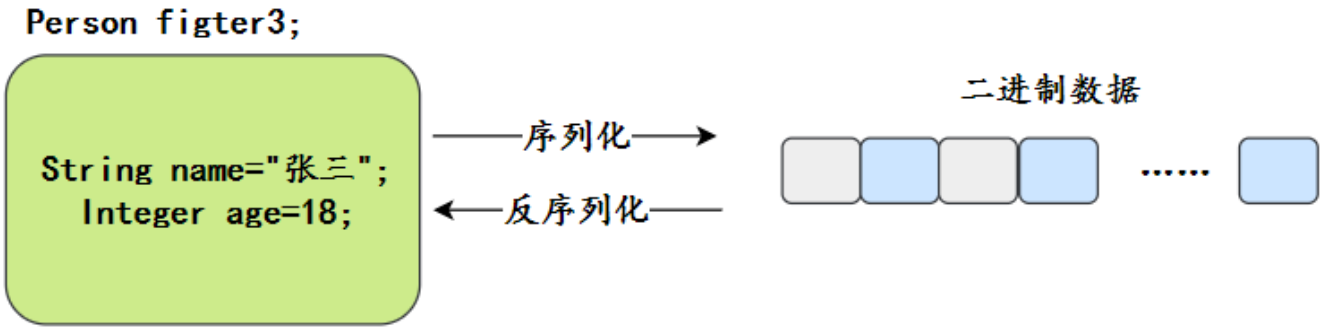
AIO：JDK 7 引入了 Asynchronous I/O，是异步不阻塞的 IO。在进行 I/O 编程中，常用到两种模式：Reactor 和 Proactor。Java 的 NIO 就是 Reactor，当有事件触发时，服务器端得到通知，进行相应的处理，完成后才通知服务端程序启动线程去处理，一般适用于连接数较多且连接时间较长的应用。

序列化

45.什么是序列化？什么是反序列化？

什么是序列化，序列化就是把 **Java** 对象转为二进制流，方便存储和传输。

所以反序列化就是把二进制流恢复成对象。



类比我们生活中一些大件物品的运输，运输的时候把它拆了打包，用的时候再拆包组装。

Serializable 接口有什么用？

这个接口只是一个标记，没有具体的作用，但是如果不实现这个接口，在有些序列化场景会报错，所以一般建议，创建的 JavaBean 类都实现 Serializable。

serialVersionUID 又有什么用？

serialVersionUID 就是起验证作用。

```
private static final long serialVersionUID = 1L;
```

我们经常会看到这样的代码，这个 ID 其实就是用来验证序列化的对象和反序列化对应的对象 ID 是否一致。

这个 ID 的数字其实不重要，无论是 1L 还是 IDE 自动生成的，只要序列化时候对象的 serialVersionUID 和反序列化时候对象的 serialVersionUID 一致的话就行。

如果没有显示指定 serialVersionUID ，则编译器会根据类的相关信息自动生成一个，可以认为是一个指纹。

所以如果你没有定义一个 serialVersionUID， 结果序列化一个对象之后，在反序列化之前把对象的类的结构改了，比如增加了一个成员变量，则此时的反序列化会失败。

因为类的结构变了，所以 serialVersionUID 就不一致。

Java 序列化不包含静态变量？

序列化的时候是不包含静态变量的。

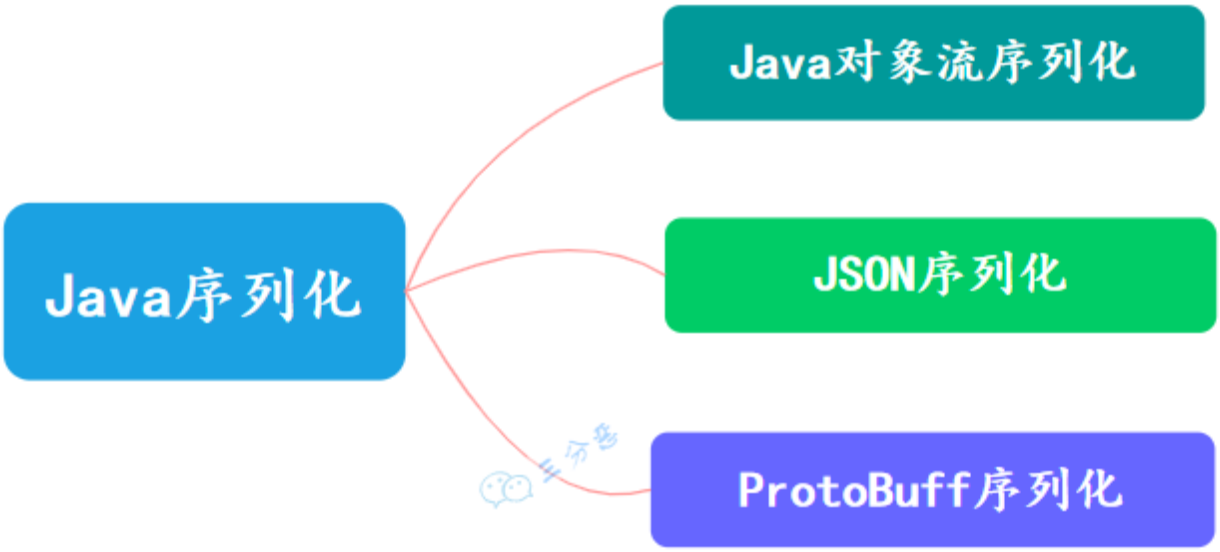
如果有些变量不想序列化，怎么办？

对于不想进行序列化的变量，使用`transient`关键字修饰。

`transient` 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被 `transient` 修饰的变量值不会被持久化和恢复。`transient` 只能修饰变量，不能修饰类和方法。

46.说说有几种序列化方式？

Java 序列化方式有很多，常见的有三种：



- Java 对象序列化 ：Java 原生序列化方法即通过 Java 原生流(InputStream 和 OutputStream 之间的转化)的方式进行转化，一般是对象输出流 `ObjectOutputStream` 和对象输入流 `ObjectInputStream` 。
- Json 序列化：这个可能是我们最常用的序列化方式，Json 序列化的选择很多，一般会使用 jackson 包，通过 ObjectMapper 类来进行一些操作，比如将对象转化为 byte 数组或者将 json 串转化为对象。
- ProtoBuff 序列化：ProtocolBuffer 是一种轻便高效的结构化数据存储格式，ProtoBuff 序列化对象可以很大程度上将其压缩，可以大大减少数据传输大小，提高系统性能。

泛型

47.Java 泛型了解么？什么是类型擦除？介绍一下常用的通配符？

什么是泛型？

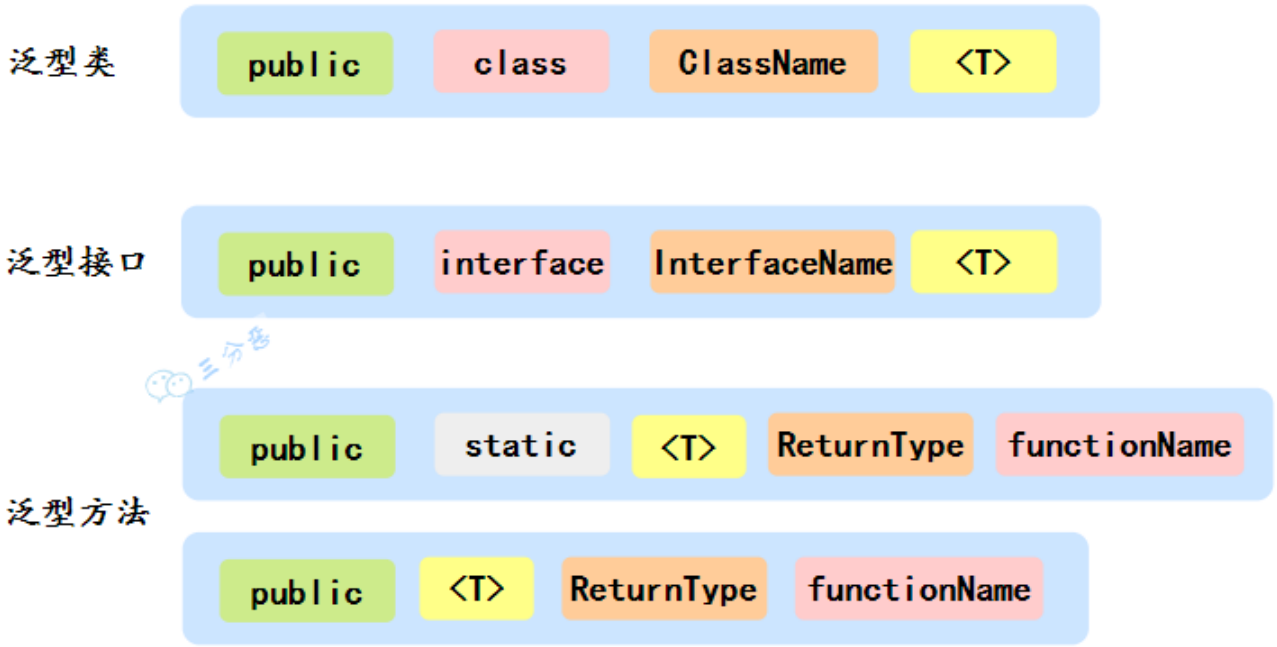
Java 泛型（generics）是 JDK 5 中引入的一个新特性，泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

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

list.add(12);
//这里直接添加会报错
list.add("a");
Class<? extends List> clazz = list.getClass();
Method add = clazz.getDeclaredMethod("add", Object.class);
//但是通过反射添加，是可以的
add.invoke(list, "kl");

System.out.println(list);
```

泛型一般有三种使用方式:泛型类、泛型接口、泛型方法。



1.泛型类:

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
//在实例化泛型类时，必须指定T的具体类型
public class Generic<T>{

    private T key;

    public Generic(T key) {
        this.key = key;
    }
```

```
    public T getKey(){
        return key;
    }
}
```

如何实例化泛型类：

```
Generic<Integer> genericInteger = new Generic<Integer>(123456);
```

2.泛型接口 ：

```
public interface Generator<T> {
    public T method();
}
```

实现泛型接口，指定类型：

```
class GeneratorImpl<T> implements Generator<String>{
    @Override
    public String method() {
        return "hello";
    }
}
```

3.泛型方法 ：

```
    public static < E > void printArray( E[] inputArray )
    {
        for ( E element : inputArray ){
            System.out.printf( "%s ", element );
        }
        System.out.println();
    }
```

使用：

```
// 创建不同类型数组： Integer, Double 和 Character
Integer[] intArray = { 1, 2, 3 };
String[] stringArray = { "Hello", "World" };
printArray( intArray  );
printArray( stringArray  );
```

泛型常用的通配符有哪些？

常用的通配符为： T，E，K，V，？

- ？ 表示不确定的 java 类型
- T (type) 表示具体的一个 java 类型
- K V (key value) 分别代表 java 键值中的 Key Value
- E (element) 代表 Element

什么是泛型擦除？

所谓的泛型擦除，官方名叫“类型擦除”。

Java 的泛型是伪泛型，这是因为 Java 在编译期间，所有的类型信息都会被擦掉。

也就是说，在运行的时候是没有泛型的。

例如这段代码，往一群猫里放条狗：

```
LinkedList<Cat> cats = new LinkedList<Cat>();
LinkedList list = cats; // 注意我在这里把范型去掉了，但是list和cats是同一个链表！
list.add(new Dog()); // 完全没问题！
```

因为 Java 的范型只存在于源码里，编译的时候给你静态地检查一下范型类型是否正确，而到了运行时就不检查了。上面这段代码在 JRE（Java运行环境）看来和下面这段没区别：

```
LinkedList cats = new LinkedList(); // 注意：没有范型！
LinkedList list = cats;
list.add(new Dog());
```

为什么要类型擦除呢？

主要是为了向下兼容，因为 JDK5 之前是没有泛型的，为了让 JVM 保持向下兼容，就出了类型擦除这个策略。

注解

48.说一下你对注解的理解?

Java 注解本质上是一个标记，可以理解成生活中的一个人的一些小装扮，比如戴什么什么帽子，戴什么眼镜。

注解可以标记在类上、方法上、属性上等，标记自身也可以设置一些值，比如帽子颜色是绿色。

有了标记之后，我们就可以在编译或者运行阶段去识别这些标记，然后搞一些事情，这就是注解的用处。

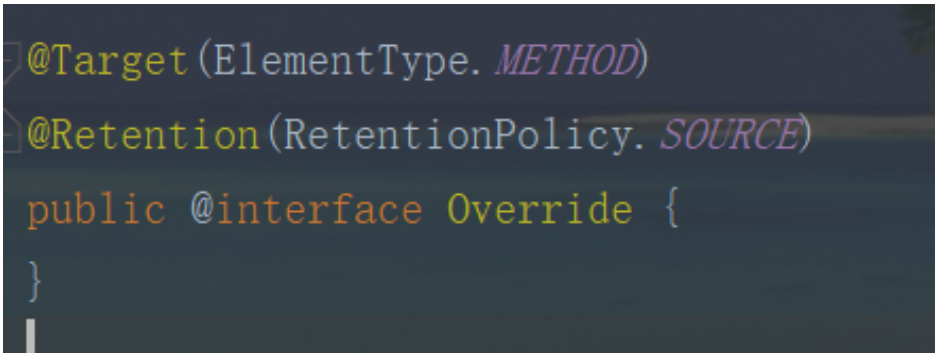
例如我们常见的 AOP，使用注解作为切点就是运行期注解的应用；比如 lombok，就是注解在编译期的运行。

注解生命周期有三大类，分别是：

- RetentionPolicy.SOURCE：给编译器用的，不会写入 class 文件
- RetentionPolicy.CLASS：会写入 class 文件，在类加载阶段丢弃，也就是运行的时候就没这个信息了
- RetentionPolicy.RUNTIME：会写入 class 文件，永久保存，可以通过反射获取注解信息

所以我上文写的是解析的时候，没写具体是解析啥，因为不同的生命周期的解析动作是不同的。

像常见的：



就是给编译器用的，编译器编译的时候检查没问题就 over 了，class 文件里面不会有 Override 这个标记。

再比如 Spring 常见的 Autowired ，就是 RUNTIME 的，所以在运行的时候可以通过反射得到注解的信息，还能拿到标记的值 required 。

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {
    boolean required() default true;
}
```

反射

49.什么是反射？应用？原理？

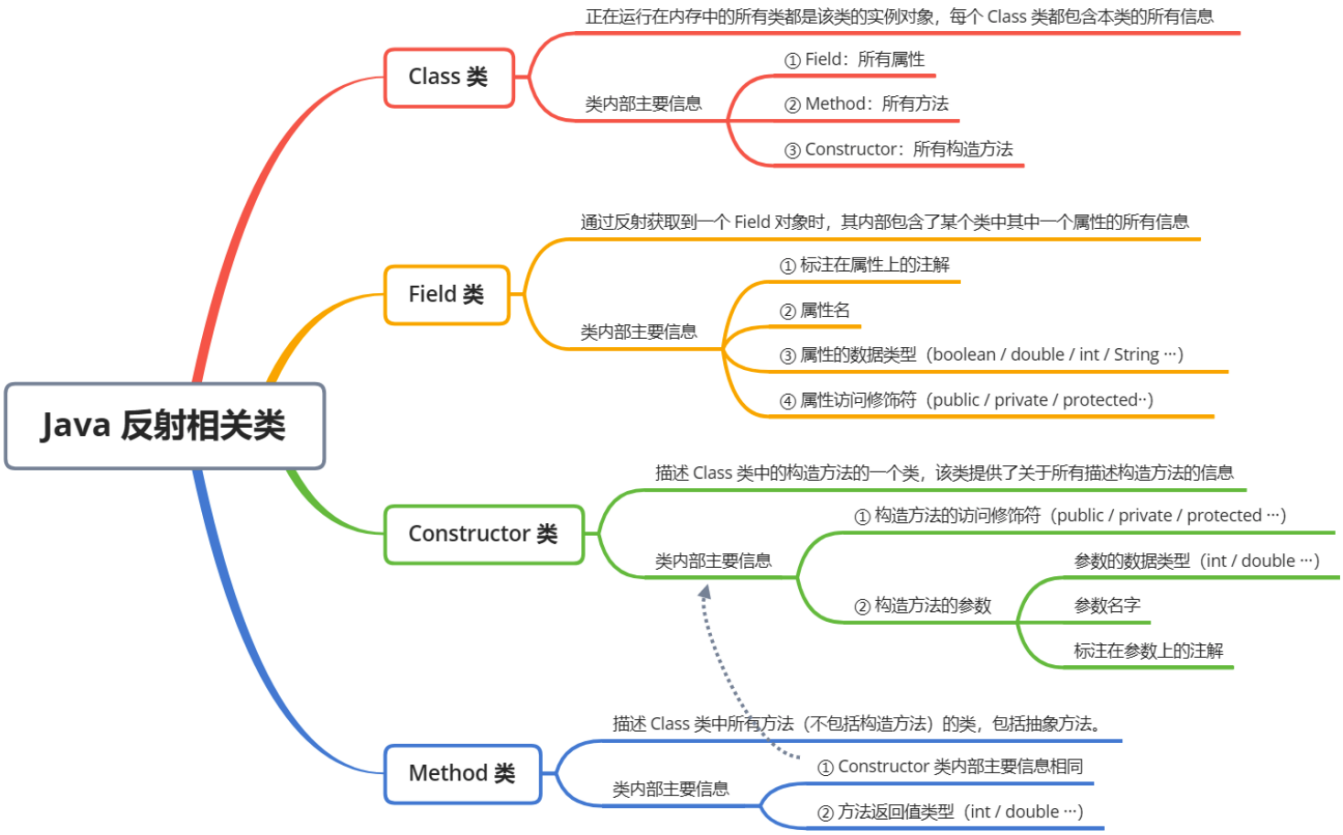
什么是反射？

我们通常都是利用`new`方式来创建对象实例，这可以说就是一种“正射”，这种方式在编译时候就确定了类型信息。

而如果，我们想在时候动态地获取类信息、创建类实例、调用类方法这时候就要用到反射。

通过反射你可以获取任意一个类的所有属性和方法，你还可以调用这些方法和属性。

反射最核心的四个类：



反射的应用场景？

一般我们平时都是在在写业务代码，很少会接触到直接使用反射机制的场景。

但是，这并不代表反射没有用。相反，正是因为反射，你才能这么轻松地使用各种框架。像 Spring/Spring Boot、MyBatis 等等框架中都大量使用了反射机制。

像 Spring 里的很多 注解 ，它真正的功能实现就是利用反射。

就像为什么我们使用 Spring 的时候 ，一个`@Component`注解就声明了一个类为 Spring Bean 呢？为什么通过一个 `@Value`注解就读取到配置文件中的值呢？究竟是怎么起作用的呢？

这些都是因为我们可以基于反射操作类，然后获取到类/属性/方法/方法的参数上的注解，注解这里就有两个作用，一是标记，我们对注解标记的类/属性/方法进行对应的处理；二是注解本身有一些信息，可以参与到处理的逻辑中。

反射的原理？

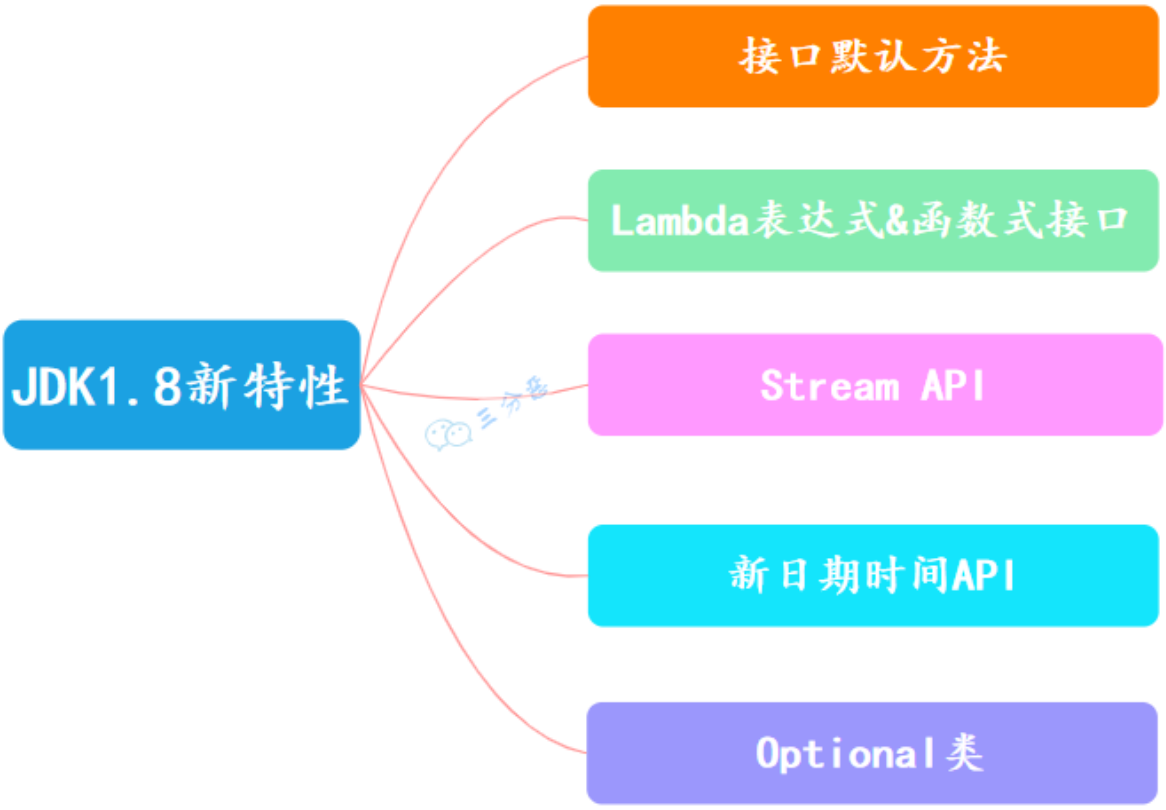
我们都知道 Java 程序的执行分为编译和运行两步，编译之后会生成字节码(.class)文件，JVM 进行类加载的时候，会加载字节码文件，将类型相关的所有信息加载进方法区，反射就是去获取这些信息，然后进行各种操作。

JDK1.8 新特性

JDK 已经出到 17 了，但是你迭代你的版本，我用我的 8。JDK1.8 的一些新特性，当然现在也不新了，其实在工作中已经很常用了。

50.JDK1.8 都有哪些新特性?

JDK1.8 有不少新特性，我们经常接触到的新特性如下：



- 接口默认方法：Java 8 允许我们给接口添加一个非抽象的方法实现，只需要使用 default 关键字修饰即可
- Lambda 表达式和函数式接口：Lambda 表达式本质上是一段匿名内部类，也可以是一段可以传递的代码。Lambda 允许把函数作为一个方法的参数（函数作为参数传递到方法中），使用 Lambda 表达式使代码更加简洁，但是也不要滥用，否则会有可读性等问题，《Effective Java》作者 Josh Bloch 建议使用 Lambda 表达式最好不要超过 3 行。
- Stream API：用函数式编程方式在集合类上进行复杂操作的工具，配合 Lambda 表达式可以方便的对集合进行处理。

Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用 Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 Stream API 来并行执行操作。

简而言之，Stream API 提供了一种高效且易于使用的处理数据的方式。

- 日期时间 API：Java 8 引入了新的日期时间 API 改进了日期时间的管理。
- Optional 类：用来解决空指针异常的问题。很久以前 Google Guava 项目引入了 Optional 作为解决空指针异常的一种方式，不赞成代码被 null 检查的代码污染，期望程序员写整洁的代码。受 Google Guava 的鼓励，Optional 现在是 Java 8 库的一部分。

51.Lambda 表达式了解多少？

Lambda 表达式本质上是一段匿名内部类，也可以是一段可以传递的代码。

比如我们以前使用 Runnable 创建并运行线程：

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread is running before Java8!");
    }
}).start();
```

这是通过内部类的方式来重写 run 方法，使用 Lambda 表达式，还可以更加简洁：

```
new Thread( () -> System.out.println("Thread is running since Java8!") ).start();
```

当然不是每个接口都可以缩写成 Lambda 表达式。只有那些函数式接口（Functional Interface）才能缩写成 Lambda 表示式。

所谓函数式接口（Functional Interface）就是只包含一个抽象方法的声明。针对该接口类型的所有 Lambda 表达式都会与这个抽象方法匹配。

Java8 有哪些内置函数式接口？

JDK 1.8 API 包含了很多内置的函数式接口。其中就包括我们在老版本中经常见到的 **Comparator** 和 **Runnable**，Java 8 为他们都添加了 @FunctionalInterface 注解，以用来支持 Lambda 表达式。

除了这两个之外，还有 Callable、Predicate、Function、Supplier、Consumer 等等。

52.Optional 了解吗？

`Optional`是用于防范`NullPointerException`。

可以将 `Optional` 看做是包装对象（可能是 `null`，也有可能非 `null`）的容器。当我们定义了一个方法，这个方法返回的对象可能是空，也有可能非空的时候，我们就可以考虑用 `Optional` 来包装它，这也是在 Java 8 被推荐使用的做法。

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true
optional.get();                 // "bam"
optional.orElse("fallback");    // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"
```

53.Stream 流用过吗?

`Stream` 流，简单来说，使用 `java.util.Stream` 对一个包含一个或多个元素的集合做各种操作。这些操作可能是 *中间操作* 亦或是 *终端操作*。终端操作会返回一个结果，而中间操作会返回一个 `Stream` 流。

Stream 流一般用于集合，我们对一个集合做几个常见操作：

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

- **Filter 过滤**

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

- **Sorted 排序**

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa1", "aaa2"
```

- **Map 转换**

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

- **Match 匹配**

```
// 验证 list 中 string 是否有以 a 开头的, 匹配到第一个, 即返回 true
boolean anyStartsWithA =
    stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true

// 验证 list 中 string 是否都是以 a 开头的
boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false

// 验证 list 中 string 是否都不是以 z 开头的,
boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);      // true
```

- **Count 计数**

`count` 是一个终端操作，它能够统计 `stream` 流中的元素总数，返回值是 `long` 类型。

```
// 先对 list 中字符串开头为 b 进行过滤，让后统计数量
long startsWithB =
    stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();

System.out.println(startsWithB);    // 3
```

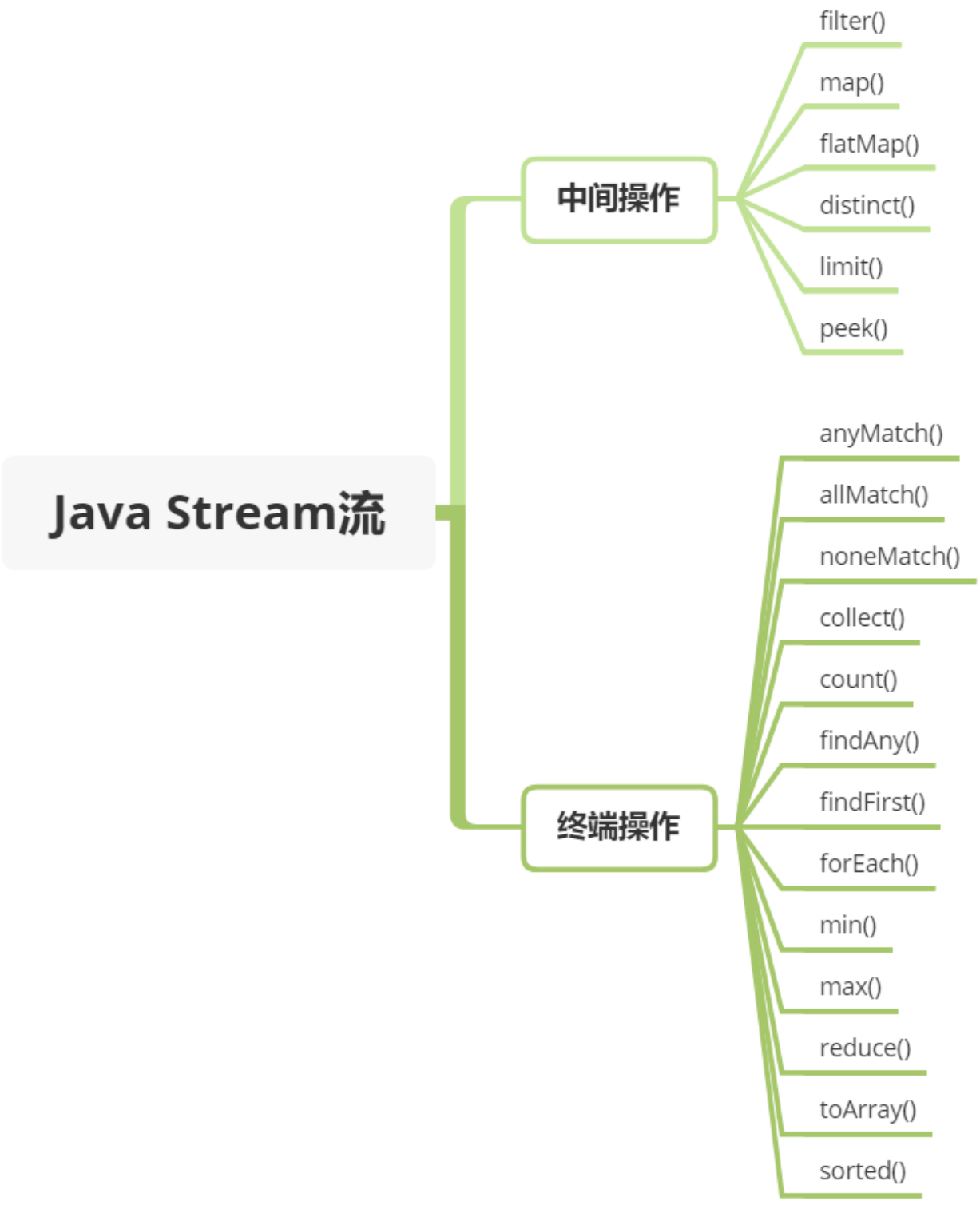
- **Reduce**

`Reduce` 中文翻译为：*减少、缩小*。通过入参的 `Function`，我们能够将 `list` 归约成一个值。它的返回类型是 `Optional` 类型。

```
Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

以上是常见的几种流式操作，还有其它的一些流式操作，可以帮助我们更便捷地处理集合数据。



没有什么使我停留——除了目的，纵然岸旁有玫瑰、有绿荫、有宁静的港湾，我是不系之舟。

系列内容：

- 面渣逆袭 Java SE 篇👍
- 面渣逆袭 Java 集合框架篇👍
- 面渣逆袭 Java 并发编程篇👍
- 面渣逆袭 JVM 篇👍
- 面渣逆袭 Spring 篇👍
- 面渣逆袭 Redis 篇👍
- 面渣逆袭 MyBatis 篇👍
- 面渣逆袭 MySQL 篇👍
- 面渣逆袭操作系统篇👍
- 面渣逆袭计算机网络篇👍

图文详解 53 道Java基础面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。