[一、计算机基础面试题](#_Toc15068)

[1.1算机的组成](#_Toc701)

[1.2 不同进制数组成](#_Toc138)

[1.2.1 二进制](#_Toc8771)

[1.2.2 10进制](#_Toc22890)

[1.2.3 8进制](#_Toc22001)

[1.2.4 16进制](#_Toc21909)

[1.3 机制之间的转换](#_Toc9582)

[1.3.1 2机制转10进制](#_Toc32324)

[1.3.2 10进制转2进制](#_Toc15508)

[1.4 各个进制的前缀](#_Toc353)

[1.5 什么是机器数](#_Toc25905)

[1.6 什么是真值](#_Toc20170)

[1.7 什么是源码](#_Toc4047)

[1.8 什么是反码](#_Toc23679)

[1.9 什么是补码](#_Toc27556)

[1.10 为何要使用原码, 反码和补码](#_Toc12615)

[1.11 什么是ASCII码](#_Toc20250)

[1.12 什么是UNICODE编码,编码的种类是哪些](#_Toc4508)

[1.12.1 概念](#_Toc32150)

[1.12.2 作用](#_Toc18652)

[1.12.3 UNICODE与ASCII的区别](#_Toc31837)

[1.13 什么是三位运算符](#_Toc6414)

[1.14 什么是位运算符](#_Toc30673)

[1.14.1 与运算符 &](#_Toc28090)

[1.14.2 或运算符 |](#_Toc853)

[1.14.3．非运算符 ~](#_Toc4780)

[1.14.4 异或运算符](#_Toc4510)

[1.13.5 记忆要点](#_Toc30100)

[1.15 Java中短路和非短路的概念](#_Toc12346)

[二 Java基本概念](#_Toc29155)

[2.1 什么是程序](#_Toc10380)

[2.2 什么是计算机程序](#_Toc24661)

[2.3 什么是计算机语言](#_Toc31135)

[2.4 java语言的种类](#_Toc12294)

[2.5 开发Java程序的步骤](#_Toc31086)

[2.6 java编码规范](#_Toc15630)

[2.7 JDK和JRE的区别](#_Toc8601)

[2.8 什么是变量和常量](#_Toc30950)

[2.9 什么是自动类型转换和强制类型转换](#_Toc27299)

[2.9 java的数据类型分为几种,分别是什么?](#_Toc23346)

[列举java算数运算符](#_Toc29219)

[列举java 关系运算符](#_Toc1443)

[列举java的赋值运算符](#_Toc19354)

[列举java 逻辑运算符](#_Toc7173)

[&和&&的区别](#_Toc6337)

[三元运算符](#_Toc6593)

[三 Java语法](#_Toc11668)

[3.1 if选择结构的种类和区别是什么](#_Toc30802)

[3.2 switch选择结构的特点是什么](#_Toc25956)

[3.3 循环的种类有哪些](#_Toc22142)

[3.4 什么是重载和重写](#_Toc14764)

[3.6 什么是装箱和拆箱](#_Toc3676)

[为什么要装箱??](#_Toc28814)

[3.7 类成员](#_Toc22627)

[3.8 类和对象的关系](#_Toc10261)

[3.9 什么是全局变量什么是局部变量](#_Toc19480)

[3.10 Integer数值小于127时使用==比较](#_Toc27830)

[3.11 java常用关键字](#_Toc21274)

[1 native](#_Toc15524)

[2 transient](#_Toc2457)

[3 volatile](#_Toc9966)

[4 static](#_Toc8063)

[3.1 2 Java异常面试题](#_Toc27790)

[3.12.1 java异常体系](#_Toc10188)

[3.12.2 你开发常见的异常有哪些](#_Toc31427)

[3.12.3 java异常处理的throw和throws的区别](#_Toc28665)

[3.12.4 Try-catch-finally的执行过程](#_Toc4466)

[3.12.5 try中有return，finally中的code会不会执行，执行是在return前还是后？](#_Toc1893)

[3.15 枚举](#_Toc15717)

[四 Java OOP](#_Toc7947)

[4.1 OOP基础知识](#_Toc19866)

[4.1.1 Java面向对象三大思想是什么,谈谈你的理解](#_Toc27673)

[封装](#_Toc25608)

[继承](#_Toc19494)

[多态](#_Toc1406)

[4.1.2 什么是值传递和引用传递](#_Toc8206)

[4.1.3 final,finally,finalize 三者区别](#_Toc27017)

[4.1.4 static关键字的作用](#_Toc15042)

[4.1.5 什么是多态](#_Toc4745)

[4.1.6 多态的形式](#_Toc27536)

[4.3 集合知识点](#_Toc14165)

[4.3.1 java集合体系图](#_Toc10157)

[4.3.2 Java18集合类框架的基本接口有哪些？](#_Toc4896)

[4.3.3 什么是迭代器(Iterator)](#_Toc13573)

[4.3.3 ArrayList的特点](#_Toc5511)

[4.3.3 LinkedList的特点](#_Toc2210)

[4.3.4 什么是单向链表什么是双向链表？](#_Toc31713)

[4.3.5 Hash的理解](#_Toc23843)

[1) 什么是Hash](#_Toc26192)

[2) 为什么要Hash](#_Toc27165)

[4.3.6 HashMap的特点](#_Toc20008)

[4.3.7 那种方式遍历集合可以实现一边遍历一边删除值](#_Toc11104)

[4.5 字符串知识点](#_Toc12444)

[4.5.1 String StringBuffer StringBuilder区别](#_Toc24430)

[4.5.2 == 和equals的区别](#_Toc6104)

[5.使用new String，一定创建对象](#_Toc24234)

[6.String.intern()](#_Toc15858)

[StringBuffer与StringBuilder的区别，它们的应用场景是什么？](#_Toc14604)

[4.5.3 String在内存中开辟空的形式](#_Toc8280)

[4.5.4 String能不能被继承?](#_Toc13913)

[4.6 jvm-运行时数据区](#_Toc20380)

[1) 堆](#_Toc29997)

[2) 栈](#_Toc26331)

[3)方法区](#_Toc1890)

[4)程序计数器](#_Toc12041)

[5)本地方法栈](#_Toc7827)

[4.8 线程](#_Toc21989)

[4.9 IO流](#_Toc12749)

[4.10 网络编程](#_Toc7535)

[五JavaWeb](#_Toc3497)

[5.1 Jsp和Servlet的区别](#_Toc20491)

[5.2 JSP九大内置对象](#_Toc2665)

[5.3 Jsp的四大作用域](#_Toc26292)

[5.4 request 跟session的区别](#_Toc16334)

[5.4 报错的状态码](#_Toc31226)

[5.5 表单提交方式 get post 有什么区别？](#_Toc3262)

[5.5 HTTP 8种请求方式](#_Toc7055)

[5.6 Forword(请求转发)与Redirect(重定向)](#_Toc14577)

[5.7 Ajax总结](#_Toc26042)

[5.8 HttpClient介绍](#_Toc26947)

[六 脚本语言](#_Toc14059)

[6.1 Bootstrap](#_Toc9054)

[6.2 JavaScript](#_Toc820)

[6.3 Jquery](#_Toc17361)

[6.4 React](#_Toc12940)

[6.5 Vue](#_Toc8663)

[EasyUI](#_Toc31218)

[LayUI](#_Toc13912)

[七 数据库](#_Toc20212)

[7.1数据库语言类型](#_Toc2023)

[7.2 锁机制](#_Toc20519)

[7.2.1 sql锁：](#_Toc27601)

[7.2.2 方法锁](#_Toc4744)

[MySQL](#_Toc27763)

[Oracle](#_Toc751)

[Redis](#_Toc26483)

[MongoDB](#_Toc17391)

[八、框架技术](#_Toc19090)

[8.2 dubbox框架](#_Toc29655)

[8.10 SpringMVC中Restful风格的参数接收方式有哪些??](#_Toc19498)

[数据结构](#_Toc25531)

[队列](#_Toc9375)

[链表](#_Toc20384)

[数组](#_Toc14405)

[二叉树](#_Toc26955)

[算法](#_Toc23948)

[交换算法](#_Toc21056)

[冒泡排序算法](#_Toc2414)

[设计模式](#_Toc18159)

[微信小程序](#_Toc15480)

[大数据](#_Toc3114)

[其他](#_Toc7392)

[你项目中常用的工具类有哪些](#_Toc31054)

# 一、计算机基础面试题

## 1.1算机的组成

输入设备-------CPU-----输入设备

## 1.2 不同进制数组成

### 1.2.1 二进制

计算机中使用二进制表示数据

基数为2，逢2进1

二进制位：使用0和1两种符号

示例：100010110011101

### 1.2.2 10进制

0、1、2、3、4、5、6、7、8、9

基数为10，逢10进1。在十进制中，一共使用10个不同的数字符号，这些符号处于不同位置时，其权值各不相同

### 1.2.3 8进制

用于缩短二进制的数字长度

八进制基数是8，使用的数据

基数为8，逢8进1。八进制使用8种不同的符号 　0、1、2、3、4、5、6、7

(123)8

### 1.2.4 16进制

十六进制数制系统的基数是 16

基数为16，逢16进1。

十六进制使用16种不同的符号

示例：12B、00FFFF

## 1.3 机制之间的转换

### 1.3.1 2机制转10进制

二进制--->十进制

二进制数转换为十进制数

二进制数第0位的权值是2的0次方，第1位的权值是2的1次方

100

用横式计算为：

0 \* 20 + 0 \* 21 + 1 \* 22 + 0 \* 23 + 0 \* 24 + 1 \* 25 + 1\* 26 + 0 \* 27 = 100

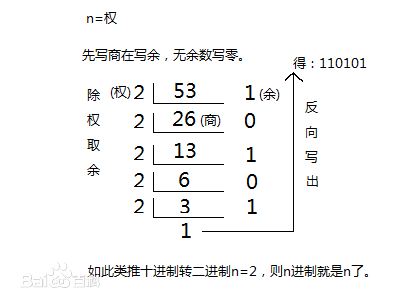
### 1.3.2 10进制转2进制

十进制--->二进制

对于整数部分，用被除数反复除以2，除第一次外，每次除以2均取前一次商的整数部分作被除数并依次记下每次的余数。另外，所得到的商的最后一位余数是所求二进制数的最高位。

十进制转，N进制。

对于小数部分，采用连续乘以基数2，并依次取出的整数部分，直至结果的小数部分为0为止。故该法称“乘基取整法”。



## 1.4 各个进制的前缀

二进制 0b Binary

十进制 0d Decimal

十六进制 0x Hexadecimal

## 1.5 什么是机器数

**一个数在计算机中的二进制表示形式, 叫做这个数的机器数**。

机器数是带符号的，在计算机用一个数的最高位存放符号, 正数为0, 负数为1.

比如，十进制中的数 +3 ，计算机字长为8位，转换成二进制就是00000011;

如果是 -3 ，就是 10000011 。

那么，这里的 00000011 和 10000011 就是机器数。

由于带符号的运算会增加计算机运算的复杂度,因此计算机实际不使用机器数进行运算

## 1.6 什么是真值

因为第一位是符号位，所以机器数的形式值就不等于真正的数值,例如上面的有符号数 10000011，其最高位1代表负，其真正数值是 -3 而不是形式值131（10000011转换成十进制等于131）。所以，为区别起见，将带符号位的机器数对应的真正数值称为机器数的真值。

**例：0000 0001的真值 = +000 0001 = +1，1000 0001的真值 = –000 0001 = –1**

## 1.7 什么是源码

**原码就是符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值. 比如如果是8位二进制:**

[+1]原 = 0000 0001

[-1]原 = 1000 0001

第一位是符号位. 因为第一位是符号位, 所以8位二进制数的取值范围就是:

即 [1111 1111，0111 1111]

原码是人脑最容易理解和计算的表示方式. [-127,127]

## 1.8 什么是反码

反码的表示方法是:

正数的反码是其本身

负数的反码是**在其原码的基础上, 符号位不变，其余各个位取反.**

[+1] = [00000001]原 = [00000001]反

[-1] = [10000001]原 = [11111110]反

可见如果一个反码表示的是负数, 人脑无法直观的看出来它的数值. 通常要将其转换成原码再计算.

## 1.9 什么是补码

补码的表示方法是:

正数的补码就是其本身

负数的补码是**在其反码的基础上, 符号位不变, 位数+1**,

最后+1. (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

对于负数, 补码表示方式也是人脑无法直观看出其数值的. 通常也需要转换成原码在计算其数值.

## 1.10 为何要使用原码, 反码和补码

在开始深入学习前, 我的学习建议是先"死记硬背"上面的原码, 反码和补码的表示方式以及计算方法.

现在我们知道了计算机可以有三种编码方式表示一个数. 对于正数因为三种编码方式的结果都相同:

**[+1] = [00000001]原 = [00000001]反 = [00000001]补**

所以不需要过多解释. 但是对于负数:

**[-1] = [10000001]原 = [11111110]反 = [11111111]补**

可见原码, 反码和补码是完全不同的. 既然原码才是被人脑直接识别并用于计算表示方式, 为何还会有反码和补码呢?

首先, 因为人脑可以知道第一位是符号位, 在计算的时候我们会根据符号位, 选择对真值区域的加减. (真值的概念在本文最开头). 但是对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: 1-1 = 1 + (-1) = 0 , 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了.

于是人们开始探索 将符号位参与运算, 并且只保留加法的方法. 首先来看原码:

计算十进制的表达式: 1-1=0

**1 - 1 = 1 + (-1) = [00000001]原 + [10000001]原 = [10000010]原 = -2**

如果用原码表示, 让符号位也参与计算, 显然对于减法来说, 结果是不正确的.这也就是为何计算机内部不使用原码表示一个数.

为了解决原码做减法的问题, 出现了反码:

计算十进制的表达式: 1-1=0

**1 - 1 = 1 + (-1)**

**= [0000 0001]原 + [1000 0001]原**

**= [0000 0001]反 + [1111 1110]反**

**= [1111 1111]反**

**= [1000 0000]原 = -0**

发现用反码计算减法, 结果的真值部分是正确的. 而唯一的问题其实就出现在"0"这个特殊的数值上. 虽然人们理解上+0和-0是一样的, 但是0带符号是没有任何意义的. 而且会有[0000 0000]原和[1000 0000]原两个编码表示0.

于是补码的出现, 解决了0的符号以及两个编码的问题:

**1-1   
= 1 + (-1)**

**= [0000 0001]原 + [1000 0001]原**

**= [0000 0001]补 + [1111 1111]补**

**= [0000 0000]补=[0000 0000]原**

这样0用[0000 0000]表示, 而以前出现问题的-0则不存在了.而且可以用[1000 0000]表示-128:

**(-1) + (-127)**

**= [1000 0001]原 + [1111 1111]原**

**= [1111 1111]补 + [1000 0001]补**

**= [1000 0000]补**

-1-127的结果应该是-128, 在用补码运算的结果中, [1000 0000]补 就是-128. 但是注意因为实际上是使用以前的-0的补码来表示-128, 所以-128并没有原码和反码表示.(对-128的补码表示[1000 0000]补算出来的原码是[0000 0000]原, 这是不正确的)

使用补码, 不仅仅修复了0的符号以及存在两个编码的问题, 而且还能够多表示一个最低数. 这就是为什么8位二进制, 使用原码或反码表示的范围为[-127, +127], 而使用补码表示的范围为[-128, 127].

因为机器使用补码, 所以对于编程中常用到的32位int类型, 可以表示范围是: [-231, 231-1] 因为第一位表示的是符号位.而使用补码表示时又可以多保存一个最小值.

## 1.11 什么是ASCII码

ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统，主要用于显示现代英语和其他西欧语言。它是现今最通用的单字节编码系统，并等同于国际标准ISO/IEC 646。

ASCII 码使用指定的7 位或8 位二进制数组合来表示128 或256 种可能的字符。标准ASCII 码也叫基础ASCII码，使用7 位二进制数（剩下的1位二进制为0）来表示所有的大写和小写字母，数字0 到9、标点符号， 以及在美式英语中使用的特殊控制字符。



## 1.12 什么是UNICODE编码,编码的种类是哪些

### 1.12.1 概念

Unicode（统一码、万国码、单一码）是计算机科学领域里的一项业界标准，包括字符集、编码方案等。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990年开始研发，1994年正式公布。

　　Unicode 是为了解决传统的字符编码方案的局限而产生的，例如ISO 8859所定义的字符虽然在不同的国家中广泛地使用，可是在不同国家间却经常出现不兼容的情况。很多传统的编码方式都有一个共同的问题，即容许电脑处理双语环境（通常使用拉丁字母以及其本地语言），但却无法同时支持多语言环境（指可同时处理多种语言混合的情况）。

### 1.12.2 作用

　　能够使计算机实现跨语言、跨平台的文本转换及处理。

### 1.12.3 UNICODE与ASCII的区别

最早只有127个字母被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母A的编码是65，小写字母z的编码是122。

　　但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。

　　你可以想得到的是，全世界有上百种语言，日本把日文编到Shift\_JIS里，韩国把韩文编到Euc-kr里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。

　　因此，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

　　Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

　　新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算

## 1.13 什么是三位运算符

**<< : 左移运算符**

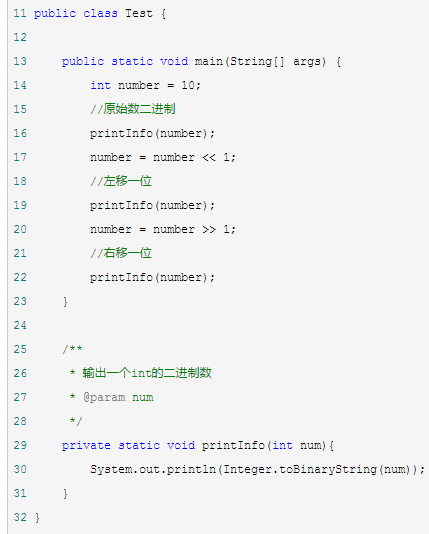
num << 1, 相当于num乘以2

**>> : 右移运算符**

num >> 1, 相当于num除以2

**>>> : 无符号右移**

忽略符号位，空位都以0补齐





## 1.14 什么是位运算符

### 1.14.1 与运算符 &

与运算符用符号“&”表示，其使用规律如下：

两个操作数中位都为1，结果才为1，否则结果为0，例如下面的程序段。

public class data13{

public static void main(String[] args){

int a=129;

int b=128;

System.out.println("a 和b 与的结果是："+(a&b));

}

}

运行结果

a 和b 与的结果是：128

下面分析这个程序：

“a”的值是129，转换成二进制就是10000001，而“b”的值是128，转换成二进制就是10000000。根据与运算符的运算规律，只有两个位都是1，结果才是1，可以知道结果就是10000000，即128。

### 1.14.2 或运算符 |

或运算符用符号“|”表示，其运算规律如下：

两个位只要有一个 为1，那么结果就是1，否则就为0，下面看一个简单的例子。

public class data14{

public static void main(String[] args){

int a=129;

int b=128;

System.out.println("a 和b 或的结果是："+(a|b));

}

}

运行结果

a 和b 或的结果是：129

下面分析这个程序段：

a 的值是129，转换成二进制就是10000001，而b 的值是128，转换成二进制就是10000000，根据或运算符的运算规律，只有两个位有一个是1，结果才是1，可以知道结果就是10000001，即128。

### 1.14.3．非运算符 ~

非运算符用符号“~”表示，其运算规律如下：

如果位为0，结果是1，如果位为1，结果是0，下面看一个简单例子。

public class data15{

public static void main(String[] args){

int a=2; 00000001 11111110

System.out.println("a 非的结果是："+(~a));

}

}

### 1.14.4 异或运算符

异或运算符是用符号“^”表示的，其运算规律是：

两个操作数的位中，相同则结果为0，不同则结果为1。下面看一个简单的例子。

public class data16 {

public static void main(String[] args) {

int a=15;

int b=2;

System.out.println("a 与 b 异或的结果是："+(a^b));

}

}

运行结果

a 与 b 异或的结果是：13

分析上面的程序段：a 的值是15，转换成二进制为1111，而b 的值是2，转换成二进制为0010，根据异或的运算规律，可以得出其结果为1101 即13。

### 1.13.5 记忆要点

**& ：位运算符-与运算 全1则1 否则为0**

**| ：位运算符-或运算 全0则0 否则为1**

**~ ：位运算符-非运算 遇1则0 遇0则1**

**^ ： 位运算符-异或运算 相同则0不同则1**

## 1.15 Java中短路和非短路的概念

短路运算符就是我们常用的“&&”、“||”，一般称为“条件操作”

非短路运算符包括 “& 与”、“| 或”、“^ 异或”，一般称为“逻辑操作”

# 二 Java基本概念

## 2.1 什么是程序

对一系列动作的执行过程的描述

## 2.2 什么是计算机程序

为了让计算机执行某些操作或解决某个问题而编写的一系列有序指令的集合

## 2.3 什么是计算机语言

编写程序的工具就是计算机语言，Java就是多种语言中的一种

## 2.4 java语言的种类

**Java SE：标准版 Java Stand Edition**

Java技术的基础和核心

主要用于开发桌面应用程序

**Java EE：企业版 java Enterprise Edition**

提供了企业级应用开发的完整解决方案

很多的网站都是采用Java EE技术开发

**Java ME**: java移动版 Java Mobile Edition

ME嵌入式开发(手机,小家电，PDA)。[苹果的iOS，黑莓]



## 2.5 开发Java程序的步骤

1. **编写.java源程序**
2. **编译.class字节码文件 javac指令**
3. **运行.class字节码文件 java指令**

## 2.6 java编码规范

1. 类名用pascal帕斯卡命名法
2. 方法名和属性名使用camel骆驼命名法
3. 类、属性、方法的命名必须是数字字母下划线或$符号,且不能以数字开头
4. 不能使用java关键字

## 2.7 JDK和JRE的区别

**JDK:**

1. Java Development Kit, 它提供了Java的开发环境和运行环境;
2. 如果你电脑安装了JDK，那么你不仅可以开发\*\*Java程序，也同时拥有了运行\*\*Java程序的平台

**JRE:**

1. Java Runtime Enviroment,它提供了Java的运行环境;
2. 如果你电脑安装了JRE，那么你的电脑只能运行Java程序，不能从事Java开发
3. JDK包含了JRE，但是，JRE可以独立安装的

## 2.8 什么是变量和常量

**变量:** 计算机存储数据的最小单元

**常量：**一旦声明就不可改变的量

## 2.9 什么是自动类型转换和强制类型转换

小萝卜放大坑是自动类型转换

大萝卜放小坑是强制类型转换

## 2.9 java的数据类型分为几种,分别是什么?

java的数据类型分为基本数据类型和引用类型

**java包含8种基本数据类型**



**java的引用类型包括**

1. String
2. java基本数据类型的包装类
3. 自定义的类、接口和数组

数据在没有指明的情况下 整型的默认类型是int 浮点型的默认数据类型是double



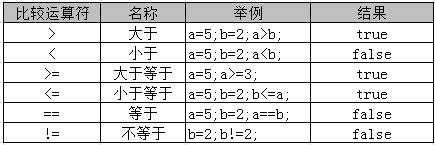
## 列举java算数运算符



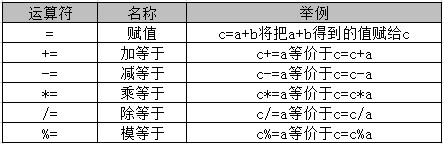
前++ 先自增后赋值

后++ 先赋值后自增

## 列举java 关系运算符



## 列举java的赋值运算符



## 列举java 逻辑运算符



## &和&&的区别

1) &: 位运算符;

2) & 如果左边的条件也会判断右边的条件;

3) &&: 逻辑运算符;

4) &&:如果左边的条件不成立,则不再判断右边的条件;

## 三元运算符

**boolean = (1 < 2) ? true : false;**

# 三 Java语法

## 3.1 if选择结构的种类和区别是什么

## 3.2 switch选择结构的特点是什么

## 3.3 循环的种类有哪些

## 3.4 什么是重载和重写

**重载**

**在一个类中存在多个方法名相同，但参数的个数和类型不同的这些方法，称为方法的重载，与返回类型无关;**

**public class Student(){**

**public void buyBook(String booKName,ouble price){YES**

**}**

**public String buyBook(String bookName){YES**

**}**

**public void buyBook(double price){ NO**

**}**

**public String buyBook(double price){NO**

**}**

**}**

**重写**

在继承中，子类重写父类的方法就叫重写，访问权限修饰符要大于等于父类的访问权限

## 3.6 什么是装箱和拆箱

**装箱**: 基本数据类型转换成包装类，就叫装箱

**拆箱**: 包装类转换成基本数据类型,就叫 拆箱

**public** **static** **void** main(String[] args)

// Integer integerNum =Integer.valueOf(10);

**int** num = 100;

//自动装箱 编译器 class源码 Integer integerNum2 = Integer.valueOf(100);

Integer integerNum2 =100;

//自动拆箱 编译器 class源码 num = integerNum2.intValue();

num =integerNum2;

}

### 为什么要装箱??

装箱类为操作基本类型的数据提供了很多的属性和方法，比如获取int类型的最大取值范围

## 3.7 类成员

**1 全局变量**

1. **构造函数**
2. **方法**

public class Test{

public String name=“Marry”; //全局变量

public void sayHello(){

String name = “tom”; //局部变量

Sysem.out.println(name); //tom

}

}

## 3.8 类和对象的关系

类是对象的类型

对象是属于某个类的实体

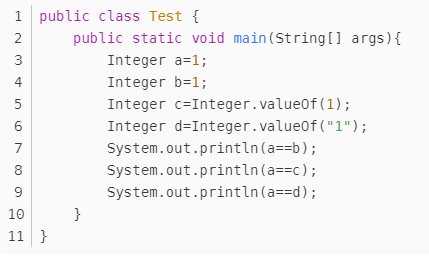
## 3.9 什么是全局变量什么是局部变量

类里声明的变量是全局变量,在类里的所有方法中都可使用

在方法中声明的遍历时局部变量,其使用范围是方法的范围,不能在方法之外使用

## 3.10 Integer数值小于127时使用==比较

Java在处理Integer时使用了一个缓存，其中缓存了-128到127之间的数字对应的Integer对象，所以在绝大多数情况下，当我们把2个小于128的Integer对象用==比较时，Java往往是用缓存中的对象进行的比较，得到的结果也往往是true，比如这样：



输出的结果全是true。

如果用debug模式就会发现，这几个对象的id都是一样的，因为他们都是从缓存中获取的同一个对象。

下面坑来了：

Integer f=new Integer(1);

人工new一个Integer对象，此时这个对象就不是从缓存中获取的了，此时再用==比较的话，返回的就是false，比如这样：

public class Test {

    public static void main(String[] args){

        Integer a=1;

        Integer e=new Integer("1");

        Integer f=new Integer(1);

        System.out.println(a==e);

        System.out.println(a==f);

    }

}

输出的都是false。

结论：

1，尽量不要用new的形式来获得Integer对象。

2，Integer对象比较大小时，不要用==，尽量使用equals方法。

## 3.11 java常用关键字

### 1 native

java源码中需要调用c语言的方法里进行操作系统底层的访问和操作,这些非java语言编写的方法都是用native关键字来修饰的

应用示例

Thread中 currentThread() , yield() , sleep()等

System中 currentTimeMillis()等

**public static native long** currentTimeMillis();

Object的 getClass() , hashCode() , clone()等

**public final native** Class<?> getClass();

**public native int** hashCode();

**protected native** Object clone() **throws** loneNotSupportedException;

### 2 transient

被transient修饰的变量,会在序列化或持久化的过程找那个，被排除掉(不进行序列化和持久化),只作为前后台暂时存储数据的变量而存在；

.

### 3 volatile

作用于变量上 , 是为了解决多线程中的变量可见性的问题 , 所谓的变量可见性 , 是因为多个线程访问同一个共享变量时 , 会为该变量在每个线程内创建变量副本 . 线程内所做的修改也并不会立即同步到原变量中 , 这样 , 当该变量改变时其他线程就无法感知 , 可能就无法访问到变量的最新值 .

Java语言为此提供了一种稍弱的同步机制 , 即volatile变量 , 用来确保将变量的更新操作通知到其他线程 . 用volatile关键字修饰的变量 , 多线程的修改会立即同步到原值 , 线程对变量读取的值也会读取原生值 .

volatile的读取性能与普通变量几乎相同 , 写操作相对稍慢 , 因为它需要在本地代码中插入许多内存屏障来保证处理器不发生乱序执行 , 即禁止指令重排优化 .

在访问volatile变量时不会执行加锁操作 , 也就是不会线程阻塞 , 因此volatile变量是一种比synchronized更轻量的同步机制 .

应用示例

Thread中 name , threadStatus

ConcurrentHashMap中 大量使用

### 4 static

1. static 修饰的属性 在类加载时在静态区开辟空间

2. static 修饰的方法 在加载时读入方法区，可以调用静态的属性,但不能调用非静态类成员

3. static 静态代码块 在类加载时读取

## 3.1 2 Java异常面试题

### 3.12.1 java异常体系

1. 所有的异常都继承自Throwable接口
2. 接口下有两个子接口 error和exception
3. **error**：
   1. 是系统级别的错误,是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息，比如内存溢出。
4. **Exception** ：表示可恢复的例外，这是可捕捉到的

### 3.12.2 你开发常见的异常有哪些

NullPointException 空指针异常

IndexOutOfRangeException 数组越界异常

ClassNotFoundException 找不到class字节码类异常

OutOfMemoryException 内存溢出异常

RuntimeException,

IOException,

SQLException,

### 3.12.3 java异常处理的throw和throws的区别

**throws**：用于声明异常，例如，如果一个方法里面不想有任何的异常处理，则在没有任何代码进行异常处理的时候，必须对这个方法进行声明有可能产生的所有异常（其实就是，不想自己处理，那就交给别人吧，告诉别人我会出现什么异常，报自己的错，让别人处理去吧）。

**throw:** 对于不是语法的异常,但是业务逻辑上不符合要求,我们可以通过throw抛出自定义异常

|  |
| --- |
| public class TestThrow  {  public static void main(String[] args)  {  try  {  //调用带throws声明的方法，必须显式捕获该异常  //否则，必须在main方法中再次声明抛出  **throwChecked**(-3);  }  catch (Exception e)  {  System.out.println(e.getMessage());  }  //调用抛出Runtime异常的方法既可以显式捕获该异常，  //也可不理会该异常  throwRuntime(3);  }  public static void **throwChecked**(int a)throws Exception  {  if (a > 0)  {  //自行抛出Exception异常  //该代码必须处于try块里，或处于带throws声明的方法中  throw new Exception("a的值大于0，不符合要求");  }  }  public static void throwRuntime(int a)  {  if (a > 0)  {  //自行抛出RuntimeException异常，既可以显式捕获该异常  //也可完全不理会该异常，把该异常交给该方法调用者处理  throw new RuntimeException("a的值大于0，不符合要求");  }  }  } |

### 3.12.4 Try-catch-finally的执行过程

|  |
| --- |
| （1）try{}语句块中放的是要检测的java代码，可能有会抛出异常，也可能会正常执行 （2）catch（异常类型）{}块是当java运行时系统接收到try块中所抛出异常对象时，会寻找处理这一异常catch块来进行处理（可以有多个catch块） （3）finally{}不管系统有没有抛出异常都会去执行，一般用来释放资源。除了在之前执行了System.exit（0） |

### 3.12.5 try中有return，finally中的code会不会执行，执行是在return前还是后？

1.当finally中没有return，但是try中有return时，执行顺序是try--finally--return，但是finally中对try中的值进行的操作是不会生效的，在try中的值会保存到函数栈中，继续执行finally，执行完后，再从栈中取出返回；代码如下

|  |
| --- |
| public class FinallyTest {  public static void main(String[] args) {  System.out.println(new FinallyTest().test());;  }  static int test(){  int x = 1;  try{  x++;  return x;}  finally{  ++x;  }}} |

2.当finally中和try中都有return时，执行顺序是先执行try中，将要返回的结果保存到函数栈中，在执行finally中，此时finally中会返回在该快中的值，而不会执行try中的return语句，代码如下：

|  |
| --- |
| **public int** func1(){  System.***out***.println(**"func1"**);  **return** 1; } **public int** func2(){  System.***out***.println(**"func2"**);  **return** 2; }  **public int** test(){  **try**{  **return** func1();  }**finally**{  **return** func2();  } } **public static void** main(String[] args) {  System.***out***.println(**new** PicUploadFileSystemService().test());  } |

## 3.15 枚举

1 什么是枚举

# 四 Java OOP

## 4.1 OOP基础知识

### 4.1.1 Java面向对象三大思想是什么,谈谈你的理解

#### 封装

封装给对象提供了隐藏内部特性和行为的能力。一些被其他访问方法来改 封装给对象提供了隐藏内部特性和行为的能力。一些被其他访问方法来改 变它内部的数据。在 变它内部的数据。在 Java当中，有 当中，有 当中，有 3种修饰符： public private和 protected 。每一

种 。每一种 。每一种 修饰符 给其他的位于同一个包或者不下面对象赋予了访问权限。

下面列出了使用封装的一些好处：

通过隐藏对象的属性来保护内部状态。

提高了代码的可用性和维护，因为对象行以被单独改变或者是扩展。

禁止对象之间的不良交互提高模块化。

#### 继承

继承给对象提供了从基类获取字段和方法的能力。继承提供了代码的重用行，也可以在不修改类的情况下给现存的类添加新特性。

#### 多态

多种形态,一个对象在不同时刻体现的不同状态;

比如: Pet 有时候指的是 Cat

Pet 有时候指的是Dog

继承、接口、向下转型、向上转型都是多态的体现

### 4.1.2 什么是值传递和引用传递

1. 基本数据类型作为参数,方法对参数做的任何操作,都不会影响原参数值;

2. 引用类型作为参数,方法对参数做的任何操作,都会影响原值;

3. String做为参数,方法的任何操作,不会受影响;

### 4.1.3 final,finally,finalize 三者区别

**final**

final修饰类 不能再被继承

final修饰的方法 不能被重写

final 修饰的属性 是常量,一旦被赋值就不能再改变

final 修饰的参数 如果是基本数据类型,则参数值不能被改变,如果是引用类型则参数的引用地址不能被改变但是参数的属性值可以重新赋值

**finally**

java异常的关键字

不论是否有异常,final里的代码都会被执行,比如关闭数据库连接

**finalize**

Object类的一个方法

用于GC垃圾回收时调用的方法,且只能由jmv内部调用

### 4.1.4 interface和abstract的区别

|  |
| --- |
| 抽象类(abstract)  含有abstract修饰符的class即为抽象类，abstract 类不能创建的实例对象。  含有abstract方法的类必须定义为abstract class，abstract class类中的方法不必是抽象的。  abstract class类中定义抽象方法必须在具体(Concrete)子类中实现，所以，不能有抽象构造方法或抽象静态方法。  如果的子类没有实现抽象父类中的所有抽象方法，那么子类也必须定义为abstract类型。  接口（interface）  interface 可以说成是抽象类的一种特例，接口中的所有方法都必须是抽象的。  **接口中的方法定义默认为public abstract类型，接口中的成员变量类型默认为public static final**。  下面比较一下两者的语法区别：  1.抽象类可以有构造方法，接口中不能有构造方法。  2.抽象类中可以有普通成员变量，接口中没有普通成员变量  3.抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。  4. 抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。  5. 抽象类中可以包含静态方法，接口中不能包含静态方法  6. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。  7. 一个类可以实现多个接口，但只能继承一个抽象类。 |

### 4.1.5 static关键字的作用

1. static 修饰的属性 在类加载时在静态区开辟空间

2. static 修饰的方法 在加载时读入方法区，可以调用静态的属性,但不能调用非静态类成员

3. static 静态代码块 在类加载时读取

### 4.1.6 什么是多态

多种形态,一个对象在不同时刻体现的不同状态;

比如: Pet 有时候指的是 Cat

Pet 有时候指的是Dog

### 4.1.7 多态的形式

1. **向上转型**

存在继承关系的子类和父类中，将子类对象赋值给父类称为向上转型

**使用场景1:** Pet pet = new Cat();

**使用场景2:** 当参数是子类时,我们应该讲参数写成父类;

**使用场景3:** 当返回类型是子类，我们应该经返回类型写成父类;

|  |
| --- |
|  |

|  |
| --- |
| //给宠物喂食--参数向上转型后一个方法搞定  **public** **void** feed(Pet pet){    System.***out***.println("主人喂食.....");  pet.eat();    } |

1. **向下转型**

**为什么?**

由于向上转型后,我们可以正常调用父类中通用的方法，但是子类中也有的方法将无法调用

如果想使用子类的特有方法，还需要向下转型

|  |
| --- |
| //常见主人  Owner owner = **new** Owner();    Pet pet = **new** Cat();//买了只猫--向上转型    owner.feed(pet);    //如果想使用子类的特有方法，还需要**向下转型**  **Cat cat1 = (Cat)pet;**  cat1.catchMouse();//让小猫去抓老鼠 |
| //和宠物玩耍  **public** **void** play(Pet pet){  **if**(pet **instanceof** Cat){  Cat cat =(Cat)pet;  cat.playWheel();  }**else** **if**(pet **instanceof** Dog){    Dog dog = (Dog)pet;  dog.paoFeidian();  }  } |

## 4.3 集合知识点

### 4.3.1 java集合体系图

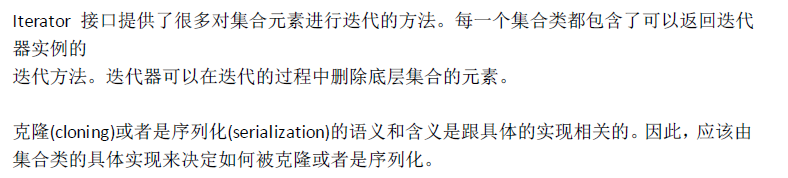


### 4.3.2 Java18集合类框架的基本接口有哪些？

Java集合类提供了一套设计良好的支持对组象进行操作接口和。 Java集合类里面最 基本的接口有：

1. Collection：代表一组对象，每个都是它的子元素。
2. Set ：不包含重复元素的 Collection。
3. List：有顺序的 collection并且可以包含重复元素。
4. Map：可以把键 (key)映射到值 (value)

### 4.3.3 什么是迭代器(Iterator)



### 4.3.3 ArrayList的特点

1 内存结构

* + - * 1. 是数组,因此查询块,增删慢;
        2. 默认数组大小是**10**,
        3. 当调用add方法后，会判断容量是否够，不够的话会进行扩容（x**1.5**）;
        4. 在删除元素时，容量不会减少（如何想减小容量调用trimToSize()）

2 线程安全性

ArrayList 线程不安全,因此效率比较高

### 4.3.3 LinkedList的特点

1. 内存结构

内存结构是双向链表,因此增删块,查询慢

1. 线程安全

### 4.3.4 什么是单向链表什么是双向链表？

1. 单向链表：

如果每个节点仅包含其指向后继节点的引用，这样的链表称为单向链表

2. 双向链表：

每个链表节点，包含两个引用，一个指向前驱节点，一个指向后驱节点，也就是——双向链表。

### 4.3.5 Hash的理解

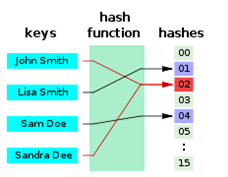
#### 1) 什么是Hash

Hash（哈希），又称“散列”

散列（hash）英文原意是“混杂”、“拼凑”、“重新表述”的意思。

在某种程度上，散列是与排序相反的一种操作，排序是将集合中的元素按照某种方式比如字典顺序排列在一起，而散列通过计算哈希值，打破元素之间原有的关系，使集合中的元素按照散列函数的分类进行排列。

在介绍一些集合时，我们总强调需要重写某个类的 equlas() 方法和 hashCode() 方法，确保唯一性。这里的 hashCode() 表示的是对当前对象的唯一标示。计算 hashCode 的过程就称作 哈希。



#### 2) 为什么要Hash

我们通常使用数组或者链表来存储元素，一旦存储的内容数量特别多，需要占用很大的空间，而且在查找某个元素是否存在的过程中，数组和链表都需要挨个循环比较，而通过 哈希 计算，可以大大减少比较次数。

<https://blog.csdn.net/amandalm/article/details/88080094>

### 4.3.6 HashMap的特点

**什么是**

1. HashMap 是一个采用哈希表实现的键值对集合，继承自 AbstractMap，实现了 Map 接口 ;
2. 内存结构是里面是一个数组,然后数组中每个元素是一个单向链表,JDK8又推荐了二叉树
3. HashMap 的特殊存储结构使得在获取指定元素前需要经过哈希运算，得到目标元素在哈希表中的位置，然后再进行少量比较即可得到元素，这使得 HashMap 的查找效率贼高

**为什么用**

1. 底层实现是 链表数组，JDK 8 后又加了 红黑树
2. 实现了 Map 全部的方法
3. key 用 Set 存放，所以想做到 key 不允许重复，key 对应的类需要重写 hashCode 和 equals 方法
4. 允许空键和空值（但空键只有一个，且放在第一位，下面会介绍）
5. 元素是无序的，而且顺序会不定时改变
6. 插入、获取的时间复杂度基本是 O(1)（前提是有适当的哈希函数，让元素分布在均匀的位置）
7. 遍历整个 Map 需要的时间与 桶(数组) 的长度成正比（因此初始化时 HashMap 的容量不宜太大）
8. 两个关键因子：初始容量、加载因子

### 4.3.7 那种方式遍历集合可以实现一边遍历一边删除值



### 4.3.8 HashTable和ConcurrentHashMap的区别

## 4.5 字符串知识点

### 4.5.1 String StringBuffer StringBuilder区别

答：String的值不能修改，如果修改的话会重新创建一个String对象，StringBuffer的值是可以修改的

1) String在重新复制时,会重新开辟新的内存空间放入新值,放弃之前的内存,容易造成内存耗损;

2) StringBuffer 线程安全,弥补了String的缺陷,并提供了String锁提供的几乎所有功能,因此建议使用

StringBuffer;

3) StringBuilder 线程不安全,也弥补了String的缺陷,因此在效率上比StringBuffer高;

4.5.2

String str1 = “hello”

String str2 = “hello”;

|  |
| --- |
|  |
| public static void main(String[] args) {  /\*\*  \* 情景一：字符串池  \* JAVA虚拟机(JVM)中存在着一个字符串池，其中保存着很多String对象;  \* 并且可以被共享使用，因此它提高了效率。  \* 由于String类是final的，它的值一经创建就不可改变。  \* 字符串池由String类维护，我们可以调用intern()方法来访问字符串池。  \*/  String s1 = "abc"; //↑ 在字符串池创建了一个对象  String s2 = "abc"; //↑ 字符串pool已经存在对象“abc”(共享),所以创建0个对象，累计创建一个对象  System.out.println("s1 == s2 : "+(s1==s2)); //↑ true 指向同一个对象  System.out.println("s1.equals(s2) : " + (s1.equals(s2))); //↑ true 值相等  //↑--------------------------------------------------------------------------------over  /\*\*  \* 情景二：关于new String("")  \*  \*/  String s3 = new String("abc"); //↑ 创建了两个对象，一个存放在字符串池中，一个存在与堆区中；  //↑ 还有一个对象引用s3存放在栈中  String s4 = new String("abc"); //↑ 字符串池中已经存在“abc”对象，所以只在堆中创建了一个对象    System.out.println("s3 == s4 : "+(s3==s4)); //↑false s3和s4栈区的地址不同，指向堆区的不同地址；  System.out.println("s3.equals(s4) : "+(s3.equals(s4))); //↑true s3和s4的值相同  System.out.println("s1 == s3 : "+(s1==s3)); //↑false 存放的地区多不同，一个栈区，一个堆区  System.out.println("s1.equals(s3) : "+(s1.equals(s3))); //↑true 值相同    //↑------------------------------------------------------over  /\*\*  \* 情景三：  \* 由于常量的值在编译的时候就被确定(优化)了。  \* 在这里，"ab"和"cd"都是常量，因此变量str3的值在编译时就可以确定。  \* 这行代码编译后的效果等同于： String str3 = "abcd";  \*/  String str1 = "ab" + "cd"; //1个对象  String str11 = "abcd";  System.out.println("str1 = str11 : "+ (str1 == str11));  //↑------------------------------------------------------over  /\*\*  \* 情景四：  \* 局部变量str2,str3存储的是存储两个拘留字符串对象(intern字符串对象)的地址。  \*  \* 第三行代码原理(str2+str3)：  \* 运行期JVM首先会在堆中创建一个StringBuilder类，  \* 同时用str2指向的拘留字符串对象完成初始化，  \* 然后调用append方法完成对str3所指向的拘留字符串的合并，  \* 接着调用StringBuilder的toString()方法在堆中创建一个String对象，  \* 最后将刚生成的String对象的堆地址存放在局部变量str3中。  \*  \* 而str5存储的是字符串池中"abcd"所对应的拘留字符串对象的地址。  \* str4与str5地址当然不一样了。  \*  \* 内存中实际上有五个字符串对象：  \* 三个拘留字符串对象、一个String对象和一个StringBuilder对象。  \*/  String str2 = "ab"; //1个对象  String str3 = "cd"; //1个对象  String str4 = str2+str3;  String str5 = "abcd";  System.out.println("str4 = str5 : " + (str4==str5)); // false  //↑------------------------------------------------------over  /\*\*  \* 情景五：  \* JAVA编译器对string + 基本类型/常量 是当成常量表达式直接求值来优化的。  \* 运行期的两个string相加，会产生新的对象的，存储在堆(heap)中  \*/  String str6 = "b";  String str7 = "a" + str6;  String str67 = "ab";  System.out.println("str7 = str67 : "+ (str7 == str67));  //↑str6为变量，在运行期才会被解析。  final String str8 = "b";  String str9 = "a" + str8;  String str89 = "ab";  System.out.println("str9 = str89 : "+ (str9 == str89));  //↑str8为常量变量，编译期会被优化  //↑------------------------------------------------------over  } |
| Buffer("111");   **是指向一个先前已经创建的对象。**只有通过new()方法才能保证每次都创建一个新的对象 5.使用new String，一定创建对象 在执行String a = new String("123")的时候，首先走常量池的路线取到一个实例的引用，然后在堆上创建一个新的String实例，走以下构造函数给value属性赋值，然后把实例引用赋值给a：  public String(String original) {  int size = original.count;  char[] originalValue = original.value;  char[] v;  if (originalValue.length > size) {  // The array representing the String is bigger than the new  // String itself. Perhaps this constructor is being called  // in order to trim the baggage, so make a copy of the array.  int off = original.offset;  v = Arrays.copyOfRange(originalValue, off, off+size);  } else {  // The array representing the String is the same  // size as the String, so no point in making a copy.  v = originalValue;  }  this.offset = 0;  this.count = size;  this.value = v;  }  从中我们可以看到，虽然是新创建了一个String的实例，但是value是等于常量池中的实例的value，即是说没有new一个新的字符数组来存放"123"。  如果是String a = new String("123"+b)的情况，首先看回第4点，"123"+b得到一个实例后，再按上面的构造函数执行 6.String.intern() String对象的实例调用intern方法后，可以让JVM检查常量池，如果没有实例的value属性对应的字符串序列比如"123"(注意是检查字符串序列而不是检查实例本身)，就将本实例放入常量池，如果有当前实例的value属性对应的字符串序列"123"在常量池中存在，则返回常量池中"123"对应的实例的引用而不是当前实例的引用，即使当前实例的value也是"123"。  public native String intern();  存在于.class文件中的常量池，在运行期被JVM装载，并且可以扩充。String的 intern()方法就是扩充常量池的 一个方法；当一个String实例str调用intern()方法时，Java 查找常量池中 是否有相同Unicode的字符串常量，如果有，则返回其的引用，如果没有，则在常 量池中增加一个Unicode等于str的字符串并返回它的引用；看示例就清楚了  public static void main(String[] args) {  String s0 = "kvill";  String s1 = new String("kvill");  String s2 = new String("kvill");  System.out.println( s0 == s1 ); //false  System.out.println( "\*\*\*\*\*\*\*\*\*\*" );  s1.intern(); //虽然执行了s1.intern(),但它的返回值没有赋给s1  s2 = s2.intern(); //把常量池中"kvill"的引用赋给s2  System.out.println( s0 == s1); //flase  System.out.println( s0 == s1.intern() ); //true//说明s1.intern()返回的是常量池中"kvill"的引用  System.out.println( s0 == s2 ); //true  }  最后我再破除一个错误的理解：有人说，“使用 String.intern() 方法则可以将一个 String 类的保存到一个全局 String 表中 ，如果具有相同值的 Unicode 字符串已经在这个表中，那么该方法返回表中已有字符串的地址，如果在表中没有相同值的字符串，则将自己的地址注册到表中”如果我把他说的这个全局的 String 表理解为常量池的话，他的最后一句话，”如果在表中没有相同值的字符串，则将自己的地址注册到表中”是错的：  public static void main(String[] args) {  String s1 = new String("kvill");  String s2 = s1.intern();  System.out.println( s1 == s1.intern() ); //false  System.out.println( s1 + " " + s2 ); //kvill kvill  System.out.println( s2 == s1.intern() ); //true  }  在这个类中我们没有声名一个”kvill”常量，所以常量池中一开始是没有”kvill”的，当我们调用s1.intern()后就在常量池中新添加了一 个”kvill”常量，原来的不在常量池中的”kvill”仍然存在，也就不是“将自己的地址注册到常量池中”了。   　　s1==s1.intern() 为false说明原来的”kvill”仍然存在；s2现在为常量池中”kvill”的地址，所以有s2==s1.intern()为true。     StringBuffer与StringBuilder的区别，它们的应用场景是什么？   jdk的实现中StringBuffer与StringBuilder都继承自AbstractStringBuilder，对于多线程的安全与非安全看到StringBuffer中方法前面的一堆synchronized就大概了解了。  这里随便讲讲AbstractStringBuilder的实现原理：我们知道使用StringBuffer等无非就是为了提高java中字符串连接的效率，因为直接使用+进行字符串连接的话，jvm会创建多个String对象，因此造成一定的开销。AbstractStringBuilder中采用一个char数组来保存需要append的字符串，char数组有一个初始大小，当append的字符串长度超过当前char数组容量时，则对char数组进行动态扩展，也即重新申请一段更大的内存空间，然后将当前char数组拷贝到新的位置，因为重新分配内存并拷贝的开销比较大，所以每次重新申请内存空间都是采用申请大于当前需要的内存空间的方式，这里是2倍    【      StringBuffer 始于 JDK 1.0      StringBuilder 始于 JDK 1.5       从 JDK 1.5 开始，带有字符串变量的连接操作（+），JVM 内部采用的是      StringBuilder 来实现的，而之前这个操作是采用 StringBuffer 实现的。  】    我们通过一个简单的程序来看其执行的流程：  public class Buffer {  public static void main(String[] args) {  String s1 = "aaaaa";  String s2 = "bbbbb";  String r = null;  int i = 3694;  r = s1 + i + s2;    for(int j=0;i<10;j++){  r+="23124";  }  }  }  使用命令javap -c Buffer查看其字节码实现：  将清单1和清单2对应起来看，清单2的字节码中ldc指令即从常量池中加载“aaaaa”字符串到栈顶，istore\_1将“aaaaa”存到变量1中，后面的一样，sipush是将一个短整型常量值(-32768~32767)推送至栈顶，这里是常量“3694”，更多的Java指令集请查看另一篇文章“[Java指令集](http://blog.csdn.net/shi1122/article/details/8053605" \t "_blank)”。    让我们直接看到13,13~17是new了一个StringBuffer对象并调用其初始化方法，20~21则是先通过aload\_1将变量1压到栈顶，前面说过变量1放的就是字符串常量“aaaaa”，接着通过指令invokevirtual调用StringBuffer的append方法将“aaaaa”拼接起来，后续的24~30同理。最后在33调用StringBuffer的toString函数获得String结果并通过astore存到变量3中。    看到这里可能有人会说，“既然JVM内部采用了StringBuffer来连接字符串了，那么我们自己就不用用StringBuffer，直接用”+“就行了吧！“。是么？当然不是了。俗话说”存在既有它的理由”，让我们继续看后面的循环对应的字节码。    37~42都是进入for循环前的一些准备工作，37,38是将j置为1。44这里通过if\_icmpge将j与10进行比较，如果j大于10则直接跳转到73，也即return语句退出函数；否则进入循环，也即47~66的字节码。这里我们只需看47到51就知道为什么我们要在代码中自己使用StringBuffer来处理字符串的连接了，因为每次执行“+”操作时jvm都要new一个StringBuffer对象来处理字符串的连接，这在涉及很多的字符串连接操作时开销会很大。 |
|  |

String str1 = **"hello"**;  
String str2 = **"hello"**;  
**str1.equals( str2 );**

//String重写的equals()方法

**public boolean equals**(Object anObject) {  
 **if** (**this** == anObject) {  
 **return true**;  
 }  
 **if** (anObject **instanceof** String) {  
 String anotherString = (String)anObject;  
 **int** n = **value**.**length**;  
 **if** (n == anotherString.**value**.**length**) {  
 **char** v1[] = **value**;  
 **char** v2[] = anotherString.**value**;  
 **int** i = 0;  
 **while** (n-- != 0) {  
 **if** (v1[i] != v2[i])  
 **return false**;  
 i++;  
 }  
 **return true**;  
 }  
 }  
 **return false**;  
}

引用类型的equals() 是Object 类的一个方法,里面实际也是对地址的判断

Object类的equls()

**public boolean** equals(Object obj) {  
 **return** (**this** == obj);  
}

所以自定义的引用类型都必须重写equals()方法

### 4.5.3 String在内存中开辟空的形式

### 4.5.4 String能不能被继承?

不能被继承,因为String是一个final类

## 4.6 jvm-运行时数据区

1. 堆
2. 栈
3. 方法区
4. 程序计数器
5. 本地方法栈

### 1) 堆

1. 堆区 存放引用类型数据
2. 堆是线程共享区域
3. 堆区是GC(垃圾回收)的主要区域,通常被分为两个区块:年轻代和老年代,这两个区域内存空间划分比例: 2:8
4. 如果堆内存不足,会产生OutOfMemoneyError异常 内存溢出异常

### 2) 栈

1. 栈区存放的基本类型数据 运算符 指向堆内存的指针
2. 栈分为java虚拟机栈和本地方法栈，其中虚拟机栈是线程私有的
3. 每个方法执行都会创建一个栈帧，用于存放局部变量表，操作栈，动态链接，方法出口等
4. 方法被调用就会分配一个栈帧,完成入栈、方法执行和方法出栈的过程;
5. 在栈区通常会出现StackOverFlowError错误和栈区空间不足时会产生OutOfMemoneyError内存溢出错误
6. 本地方法栈为虚拟机使用到本地方法服务（native）

### 3)方法区

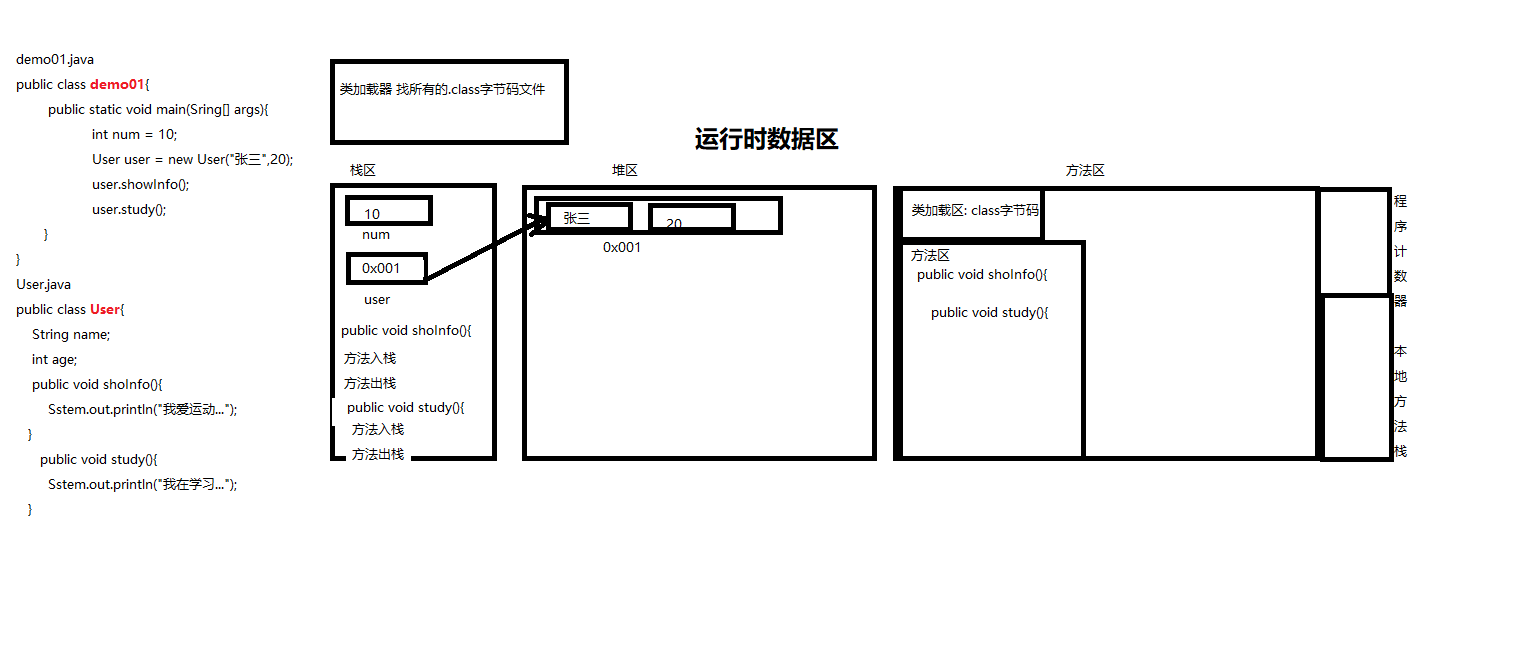
1. 该区通常要被划分为 类加载区、方法区、静态区和常量池等部分;
2. 该区数据所有线程共享;
3. 方法区被划分在永久代也就是jdk8后的元空间中,这个区域一般不会进行GC回收操作

### 4)程序计数器

用于执行指向下一个要执行的指令

### 5)本地方法栈

本地方法栈为虚拟机使用到本地方法服务（native）

****

## 4.8 线程

### 1)单线程和多线程

如果一个程序只有一条执行路径,则称为是单线程应用程序;dos 窗体就是单线程应用程序

如果一个程序只有多条执行路径,则称为是多线程应用程序; 杀毒软件 音频视频播放器

### 2)进程和线程

**进程 :**

1) 正在运行的程序，是系统进行资源分配和调用的独立单位;

2) 每一个进程都有它自己的内存空间和系统资源;

**进程作用:**

1. 提高CPU使用率;
2. 提高应用程序的使用率;

### 3) 线程和进程的区别

Cpu最小调度的单元是线程，进程是划分内存空间的最小单元，进程里面的数据多个线程共享

1.线程(Thread)与进程（Process）

进程定义的是应用程序与应用程序之间的边界，通常来说一个进程就代表一个与之对应的应用程序。不同的进程之间不能共享代码和数据空间，而同一进程的不同线程可以共享代码和数据空间。

一个进程可以包括若干个线程，同时创建多个线程来完成某项任务，便是多线程。

实现线程的三种方式：继承Thread类，实现Runable接口，实现Callable

### 4) 实现多线程的两种方式

多线程有两种实现方法，继承Thread类和实现Runnable接口

|  |
| --- |
| //也可以把类写在外边  public static class MyThread **extends Thread**{  private int tickets=100;  public void **run**() {  while(tickets>0){  System.out.println(this.getName()+"卖出第【"+tickets--+"】张火车票");  }  }  } |
| public class ThreadTest {  public static void main(String[] args){  new MyThread().**start**();  new MyThread().start();  new MyThread().start();  new MyThread().start();  }  } |
| public static class MyThread implements **Runnable**{  private int tickets=100;  public void **run**() {  while(tickets>0){  System.out.println(Thread.currentThread().getName()  +"卖出第【"+tickets--+"】张火车票");  }  }  } |
| /\*\*  \* 使用Runnable接口模拟4个售票窗口共同卖100张火车票的程序  \*/public class RunnableTest {  public static void main(String[] args) {  Runnable runnable=new MyThread();  new Thread(runnable).**start**();  new Thread(runnable).start();  new Thread(runnable).start();  new Thread(runnable).start();  }  } |
| 以上哪种方式更好??在实际开发中通常以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类可以避免继承的局限，一个类可以继承多个接口，适合于资源的共享。 |

### 5）线程能不能多次启动

|  |
| --- |
| 不能  一个线程对象只能调用一次start方法.从new到等待运行是单行道,  所以如果你对一个已经启动的线程对象再调用一次start方法的话,会产生:IllegalThreadStateException异常. |

### 线程调度+

|  |
| --- |
| 假如我们的计算机只有一个 CPU，那么 CPU 在某一个时刻只能执行一条指令，线程只有得到 CPU时间片，也就是使用权，才可以执行指令。那么Java是如何对线程进行调用的呢？  线程有两种调度模型：  分时调度模型 所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间片  抢占式调度模型 优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个，优先级高的线程获取的 CPU 时间片相对多一些。  Java使用的是抢占式调度模型。  演示如何设置和获取线程优先级  public final int getPriority()  public final void setPriority(int newPriority) |

### 8）线程常用方法有哪些

|  |  |
| --- | --- |
| **方法** | **说明** |
| 线程休眠  public static void **sleep**(long millis) | **填好** |
| 等待线程  public static **wait**(); | **填好** |
| 线程加入  public final void **join**() | **填好** |
| 线程礼让  public static void **yield**() | 是暂停当前正在执行的线程对象,把时间让给其它线程 |
| 后台线程  public final void setDaemon(boolean on) | **填好** |
| 中断线程  public final void stop()  public void interrupt() | **填好** |

### 10)在java中wait和sleep方法的不同？

|  |
| --- |
| **sleep方法：**  　　属于Thread类中的方法；会导致程序暂停执行指定的时间，让出cpu给其他线程，但是他的监控状态依然保持着，当指定时间到了之后，又会自动恢复运行状态；在调用sleep方法的过程中，线程不会释放对象锁。（只会让出CPU，不会导致锁行为的改变）  **wait方法：**  　　属于Object类中的方法；在调用wait方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify方法后本线程才进入对象锁定池准备。获取对象锁进入运行状态。（不仅让出CPU，还释放已经占有的同步资源锁） |

### 11)为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？

|  |
| --- |
| 这是另一个非常经典的java多线程面试问题。这也是我刚开始写线程程序时候的困惑。现在这个问题通常在电话面试或者是在初中级Java面试的第一轮被问到。这个问题的回答应该是这样的，当你调用start()方法时你将创建新的线程，并且执行在run()方法里的代码。但是如果你直接调用run()方法，它不会创建新的线程也不会执行调用线程的代码。  简单点来说：  new一个Thread，线程进入了新建状态;调用start()方法，线程进入了就绪状态，当分配到时间片后就可以开始运行了。  start()会执行线程的相应准备工作，然后自动执行run()方法的内容。是真正的多线程工作。  而直接执行run()方法，会把run方法当成一个mian线程下的普通方法去执行，并不会在某个线程中执行它，这并不是多线程工作。 |

|  |
| --- |
| D:\itcast\20140613\day24\code\多线程的生命周期图解.bmp |

### 12)线程的生命周期图

### 13) java线程的状态转换介绍

|  |
| --- |
|  |
| * 新建状态（New）  用new语句创建的线程处于新建状态，此时它和其他Java对象一样，仅仅在堆区中被分配了内存。 * 就绪状态（Runnable）  当一个线程对象创建后，其他线程调用它的start()方法，该线程就进入就绪状态，Java虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得CPU的使用权。 * 运行状态（Running）  处于这个状态的线程占用CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。 * 阻塞状态（Blocked）  阻塞状态是指线程因为某些原因放弃CPU，暂时停止运行。当线程处于阻塞状态时，Java虚拟机不会给线程分配CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。  阻塞状态可分为以下3种：     1. 位于对象等待池中的阻塞状态（Blocked in object’s wait pool）：当线程处于运行状态时，如果执行了某个对象的wait()方法，Java虚拟机就会把线程放到这个对象的等待池中，这涉及到“线程通信”的内容。   2. 位于对象锁池中的阻塞状态（Blocked in object’s lock pool）：当线程处于运行状态时，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，Java虚拟机就会把这个线程放到这个对象的锁池中，这涉及到“线程同步”的内容。   3. 其他阻塞状态（Otherwise Blocked）：当前线程执行了sleep()方法，或者调用了其他线程的join()方法，或者发出了I/O请求时，就会进入这个状态。 * 死亡状态（Dead）  当线程退出run()方法时，就进入死亡状态，该线程结束生命周期。 |

### 14)多线程产生的环境

|  |
| --- |
| 是否是多线程环境  是否有共享数据  是否有多条语句操作共享数据 |

### 15)什么是原子操作，Java中的原子操作是什么？

|  |
| --- |
| 原子性是指操作不可再分割,是最小操作单元,  比如 i++就不是原子性操作,他可以分割为 i = i+1  =》 先执行i+1  =>再执行 赋值操作 |

### 16)现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行？

|  |
| --- |
| 这个线程问题通常会在第一轮或电话面试阶段被问到，目的是检测你对”join”方法是否熟悉。这个多线程问题比较简单，可以用**join方法实现。** |
| 核心：  thread.Join把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程。  比如在线程B中调用了线程A的Join()方法，直到线程A执行完毕后，才会继续执行线程B。  想要更深入了解，建议看一下join的源码，也很简单的，使用wait方法实现的。  t.join(); //调用join方法，等待线程t执行完毕  t.join(1000); //等待 t 线程，等待时间是1000毫秒。 |
| 1. public static void main(String[] args) { 2. method01(); 3. method02(); 4. } 6. */\*\** 7. *\* 第一种实现方式，顺序写死在线程代码的内部了，有时候不方便* 8. *\*/* 9. private static void method01() { 10. Thread t1 = new Thread(new Runnable() { 11. @Override public void run() { 12. System.out.println("t1 is finished"); 13. } 14. }); 15. Thread t2 = new Thread(new Runnable() { 16. @Override public void run() { 17. try { 18. t1.join(); 19. } catch (InterruptedException e) { 20. e.printStackTrace(); 21. } 22. System.out.println("t2 is finished"); 23. } 24. }); 25. Thread t3 = new Thread(new Runnable() { 26. @Override public void run() { 27. try { 28. t2.join(); 29. } catch (InterruptedException e) { 30. e.printStackTrace(); 31. } 32. System.out.println("t3 is finished"); 33. } 34. }); 36. t3.start(); 37. t2.start(); 38. t1.start(); 39. }  42. */\*\** 43. *\* 第二种实现方式，线程执行顺序可以在方法中调换* 44. *\*/* 45. private static void method02(){ 46. Runnable runnable = new Runnable() { 47. @Override public void run() { 48. System.out.println(Thread.currentThread().getName() + "执行完成"); 49. } 50. }; 51. Thread t1 = new Thread(runnable, "t1"); 52. Thread t2 = new Thread(runnable, "t2"); 53. Thread t3 = new Thread(runnable, "t3"); 54. try { 55. t1.start(); 56. t1.join(); 57. t2.start(); 58. t2.join(); 59. t3.start(); 60. t3.join(); 61. } catch (InterruptedException e) { 62. e.printStackTrace(); 63. } 64. } |

### 17)在Java中Lock接口比synchronized块的优势是什么？你需要实现一个高效的缓存，它允许多个用户读，但只允许一个用户写，以此来保持它的完整性，你会怎样去实现它？

|  |
| --- |
| 体上来说Lock是synchronized的扩展版，Lock提供了无条件的、可轮询的(tryLock方法)、定时的(tryLock带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition方法)锁操作。另外Lock的实现类基本都支持非公平锁(默认)和公平锁，synchronized只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。 |

### 18)用Java编程一个会导致死锁的程序，你将怎么解决？

|  |
| --- |
| 产生死锁的四个必要条件：  - 互斥条件：一个资源每次只能被一个进程使用。  - 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。  - 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。  - 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。  如何避免死锁？  - 从死锁的四个必要条件来看，破坏其中的任意一个条件就可以避免死锁。但互斥条件是由资源本身决定的，不剥夺条件一般无法破坏，要实现的话得自己写更多的逻辑。  - 避免无限期的等待：用Lock.tryLock(),wait/notify等方法写出请求一定时间后，放弃已经拥有的锁的程序。  - 注意锁的顺序：以固定的顺序获取锁，可以避免死锁。  - 开放调用：即只对有请求的进行封锁。你应当只想你要运行的资源获取封锁，比如在上述程序中我在封锁的完全的对象资源。但是如果我们只对它所属领域中的一个感兴趣，那我们应当封锁住那个特殊的领域而并非完全的对象。  - 最后，如果能避免使用多个锁，甚至写出无锁的线程安全程序是再好不过了。 |

### 19)Java中你怎样唤醒一个阻塞的线程？

|  |
| --- |
| 这是个关于线程和阻塞的棘手的问题，它有很多解决方法。如果线程遇到了IO阻塞，我并且不认为有一种方法可以中止线程。如果线程因为调用wait()、sleep()、或者join()方法而导致的阻塞，你可以中断线程，并且通过抛出InterruptedException来唤醒它。我之前写的《How to deal with blocking methods in java》有很多关于处理线程阻塞的信息。  这个我们先简单粗暴地对某些阻塞方法进行分类：  - 会抛出InterruptedException的方法：wait、sleep、join、Lock.lockInterruptibly等，针对这类方法，我们在线程内部处理好异常(要不完全内部处理，要不把这个异常抛出去)，然后就可以实现唤醒。  - 不会抛InterruptedException的方法：Socket的I/O,同步I/O，Lock.lock等。对于I/O类型，我们可以关闭他们底层的通道，比如Socket的I/O，关闭底层套接字，然后抛出异常处理就好了;比如同步I/O，关闭底层Channel然后处理异常。对于Lock.lock方法，我们可以改造成Lock.lockInterruptibly方法去实现。 |

### 20)Java中的volatile关键是什么作用？怎样使用它？在Java中它跟synchronized方法有什么不同？

|  |
| --- |
| volatile关键字的作用是：保证变量的可见性。  在java内存结构中，每个线程都是有自己独立的内存空间(此处指的线程栈)。当需要对一个共享变量操作时，线程会将这个数据从主存空间复制到自己的独立空间内进行操作，然后在某个时刻将修改后的值刷新到主存空间。这个中间时间就会发生许多奇奇怪怪的线程安全问题了，volatile就出来了，它保证读取数据时只从主存空间读取，修改数据直接修改到主存空间中去，这样就保证了这个变量对多个操作线程的可见性了。换句话说，被volatile修饰的变量，能保证该变量的 单次读或者单次写 操作是原子的。  但是线程安全是两方面需要的 原子性(指的是多条操作)和可见性。volatile只能保证可见性，synchronized是两个均保证的。  volatile轻量级，只能修饰变量；synchronized重量级，还可修饰方法。  volatile不会造成线程的阻塞，而synchronized可能会造成线程的阻塞。 |

### 21) 什么是竞争条件？你怎样发现和解决竞争？

|  |
| --- |
| 竞争条件，在《java并发编程实战》叫做竞态条件：指设备或系统出现不恰当的执行时序，而得到不正确的结果。  下面是个最简单的例子，是一个单例模式实现的错误示范： |
| 1. @NotThreadSafe 2. public class LazyInitRace { 3. private ExpensiveObject instance = null; 5. public ExpensiveObject getInstance() { 6. if (instance == null) 7. instance = new ExpensiveObject(); 8. return instance; 9. } 10. } |
| 在上述例子中，表现一种很常见的竞态条件类型:“先检查后执行”。根据某个检查结果来执行进一步的操作，但很有可能这个检查结果是失效的！还有很常见的竞态条件“读取-修改-写入”三连，在多线程条件下，三个小操作并不一定会放在一起执行的。  如何对待竞态条件？  首先，警惕复合操作，当多个原子操作合在一起的时候，并不一定仍然是一个原子操作，此时需要用同步的手段来保证原子性。  另外，使用本身是线程安全的类，这样在很大程度上避免了未知的风险。 |

### 同步和异步、阻塞和非阻塞的区别

**同步**是指：发送方发出数据后，等接收方发回响应以后才发下一个数据包的通讯方式。

**异步**是指：发送方发出数据后，不等接收方发回响应，接着发送下个数据包的通讯方式。

**阻塞**： 在调用 IO 时，应用程序被挂起，无法执行其它操作，需要等待 IO 完成才返回

**非阻塞**：在调用 IO 以后，应用程序不带数据立即返回（至于文件内容如何返回给调用方，已经不属于阻塞/非阻塞讨论的范畴了）

## 4.9 IO流

IO流的分类

说出至少5对IO流

写出万能流的代码

## 4.10 网络编程

# 五JavaWeb

## 5.1 Jsp和Servlet的区别

Jsp和servlet都属于javaweb技术jsp是java servelet page的缩写其实jsp就是servelet，

在最初的版本中只有servelet，servelet要写html的脚本语言把业务逻辑和脚本语言混在一起，

对于程序员来说非常不便，后来我们吧html这些脚本语言整合到了另外的文件中就是jsp，

这样就达到了一个页面代码和服务端代码分离的这么一个效果，

其实jsp他最终编译之后是一个.java文件，他继承的还是servelet，jsp要转译成servelet只是为了开发方便。

区别是jsp放的是脚本语言，servelet放的是服务端的业务逻辑。

## 5.2 JSP九大内置对象

1.**PageContext** javax.servlet.jsp.PageContext JSP的页面容器

2.**request** javax.servlet.http.HttpServletrequest 获取用户的请求信息

3.**response**  javax.servlet.http.HttpServletResponse 服务器向客户端的回应信息

4.**session** javax.servlet.http.HttpSession 用来保存每一个用户的信息

5.**application**  javax.servlet.ServletContext 表示所有用户的共享信息

6.**config**  javax.servlet.ServletConfig 服务器配置信息，可以取得初始化参数

7.**out**  javax.servlet.jsp.jspWriter 页面输出

8.**page**  java.lang.object)

9.**exception** java.lang.Throwable

## 5.3 Jsp的四大作用域

pageContext,

request,

session、

application

**pageContext,**  如果把变量放到pageContext里，就说明它的作用域是page，它的有效范围只在当前

jsp页面里。 从把变量放到pageContext开始，到jsp页面结束，你都可以使用这个

变量。

**request**, 如果把变量放到request里，就说明它的作用域是request，它的有效范围是当前请求

周期。所谓请求周期，就是指从http请求发起，到服务器处理结束，返回响应的整个

过程。在这个过程中可能使用forward的方式跳转了多个jsp页面，在这些页面里你都

可以使用这个变量

**session**、 如果把变量放到session里，就说明它的作用域是session，它的有效范围是当前会话。所

谓当前会话，就是指从用户打开浏览器开始，到用户关闭浏览器这中间的过程。这个过程

可能包含多个请求响应。也就是说，只要用户不关浏览器，服务器就有办法知道这些请求是一个人发起的，整个过程被称为一个会话（session），而放到会话中的变量，

**application** 如果把变量放到application里，就说明它的作用域是application，它的有效范围是整个应用。整个应用是指从应用启动，到应用结束。我们没有说“从服务器启动，到服务器关闭”是因为一个服务器可能部署多个应用，当然你关闭了服务器，就会把上面所有的应用都关闭了。application作用域里的变量，它们的存活时间是最长的，如果不进行手工删除，它们就一直可以使用。与上述三个不同的是，application里的变量可以被所有用户共用。如果用户甲的操作修改了application中的变量，用户乙访问时得到的是修改后的值。这在其他scope中都是不会发生的，page, request, session都是完全隔离的，无论如何修改都不会影响其他

## 5.4 request 跟session的区别

1.他们的生命周期不同，

request对应的是一次请求，

session对应的是一次会话

2.request占用资源比较少,相对来说缺乏持续性,

而session资源消耗比较大，所以通常使用request来保存信息

## 5.4 报错的状态码

301 永久重定向

302 临时重定向

304 服务端 未改变

403 访问无权限

200 正常

404 路径

500 内部错误

502 服务器重启

400 语法错误

## 5.5 表单提交方式 get post 有什么区别？

1. get是从服务器上获取数据，post是向服务器传送数据

2. get安全性较低，post安全性较高

3. get传送的数据量小，不能大于2KB。post传送的数据量大，一般被默认为不受限制（但理论上，IIS4

中最大量为80KB，IIS5中为100KB）

4. 对于get方式，服务器端用Request.Querystring获取变量的值，对于post方式，用Request.Form获

取提交的数据

5. get是把参数数据队列加到提交表单的action属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。post是通过HTTP Post机制，将表单内各个字段与其内容放置在html header内一起传送到action属性所指的URL地址，用户看不到这个过程

## 5.5 HTTP 8种请求方式

1、**OPTIONS**

返回服务器针对特定资源所支持的HTTP请求方法，也可以利用向web服务器发送‘\*’的请求来测试服务器的功能性

2、**HEAD**

向服务器索与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以再不必传输整个响应内容的情况下，就可以获取包含在响应小消息头中的元信息。

3、**GET**

向特定的资源发出请求。它本质就是发送一个请求来取得服务器上的某一资源。资源通过一组HTTP头和呈现数据（如HTML文本，或者图片或者视频等）返回给客户端。GET请求中，永远不会包含呈现数据。

4、**POST**

向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 Loadrunner中对应POST请求函数：web\_submit\_data,web\_submit\_form

5、**PUT**

向指定资源位置上传其最新内容

6、**DELETE**

请求服务器删除Request-URL所标识的资源

7、**TRACE**

回显服务器收到的请求，主要用于测试或诊断

8、**CONNECT**

HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

**请求方式幂等性问题**

**幂等: 在操作成功的前提条件下,是否会对数据库造册额外不好的影响**

Get 查询 安全且幂等

Post 新增 不安全不幂等

Put 修改 不安全但幂等

Delete 删除 不安全但幂等

## 5.6 Forword(请求转发)与Redirect(重定向)

1. 从数据共享上 Forward是一个请求的延续，可以共享request的数据 Redirect开启一个新的

请求，不可以共享request的数据(但是可以传递数据)

2. 从地址栏 Forward转发地址栏不发生变化 Redirect转发地址栏发生变化

3. 从跨域上 Redirect可以跨域请求，Forward不可以

## 5.7 Ajax总结

**ajax**

**前端技术**

**为了实现局部刷新,使用户的感受较好**

**我们使用javascipt xmlhttprequest对象来实现的**

**由于原始的 js的ajax语法比较复杂,Jquery作为js的一个框架,简化了**

**Ajax异步请求的语法**

Js的语法有

|  |
| --- |
| function loadXMLDoc() {  var xmlhttp;  //1,创建XMLHttpRequest对象  if(window.XMLHttpRequest){  xmlhttp=new XMLHttpRequest();  }else{ //兼容ie5，ie6  xmlhttp= new ActiveXObject("Microsoft.XMLHTTP");  }  //readyState 属性改变时，就会调用onreadystatechange函数  xmlhttp.onreadystatechange= function() {  if (xmlhttp.readyState == 4 && xmlhttp.status == 200){  var date = JSON.parse(xmlhttp.responseText);  console.log(date);  var code = date.code;  document.getElementById("hehe").innerHTML = code ;  }  //readyState 会从1-4的变化，错误的形式根据状态判断  }  //GET请求写法：  //xmlhttp.open("GET","test.php?id=23&name=45",true);  //xmlhttp.send();    //POST请求写法：  var data = "name=1&id=10";  xmlhttp.open("POST","test.php",true);  //POST请求需要写请求头  xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");  xmlhttp.send(data);  } |

**Jquery主要语法:**

语法1

$.ajax{

url: 发送请求的地址

type: 请求方式 post/get 默认为get

data:  **请求的参数**

dataType: 服务器返回的数据类型

success: 成功后调用的回调函数

function(data){

console.log(data);

}，

error: 失败后调用的回调函数

function(e){

console.log(e);

}

}

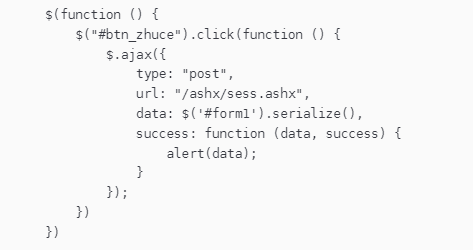
语法2 get方式

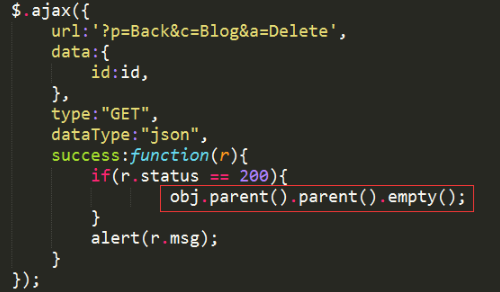
$.get(url,function(){},'json');

语法3 post方式

$.post(url,data,function(r){},'json');

例子





<div id="showMess"></div>

<script>

$.post('./ajax.do', { id:'45',username:'Simon',sex:'男' }, function(data,status){

if(status=='success'){

$('#showMess').html('学号：' + data.id+', 姓名：' + data.username + '，性别：' + data.sex);

}

},'json'); //data参数为json格式

</script>

$.get('./ajax.do'+data, function(data,status){

if(status=='success'){

$('#showMess').html('学号：' + data.id+', 姓名：' + data.username + '，性别：' + data.sex);

}

},'json');

## 5.8 HttpClient介绍

## 5.9 拦截器和过滤器的区别

|  |
| --- |
| 1.过滤器：  依赖于servlet容器。在实现上基于函数回调，可以对几乎所有请求进行过滤，但是缺点是一个过滤器实例只能在容器初始化时调用一次。使用过滤器的目的是用来做一些过滤操作，获取我们想要获取的数据，比如：在过滤器中修改字符编码；在过滤器中修改HttpServletRequest的一些参数，包括：过滤低俗文字、危险字符等  2.拦截器：  依赖于web框架，在SpringMVC中就是依赖于SpringMVC框架。在实现上基于[Java](https://link.jianshu.com?t=http://www.07net01.com/tags-Java-0.html" \t "_blank)的反射机制，属于面向切面编程（AOP）的一种运用。由于拦截器是基于web框架的调用，因此可以使用Spring的依赖注入（DI）进行一些业务操作，同时一个拦截器实例在一个controller生命周期之内可以多次调用。但是缺点是只能对controller请求进行拦截，对其他的一些比如直接访问静态资源的请求则没办法进行拦截处理  3.过滤器和拦截器的区别：  **①拦截器是基于java的反射机制的，而过滤器是基于函数回调。**  ②拦截器不依赖与servlet容器，过滤器依赖与servlet容器。  ③拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用。  ④拦截器可以访问action上下文、值栈里的对象，而过滤器不能访问。  ⑤在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次。  **⑥拦截器可以获取IOC容器中的各个bean，而过滤器就不行，这点很重要，在拦截器里注入一个service，可以调用业务逻辑。**  **过滤器**  **@Override**      public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)throws IOException, ServletException {          System.out.println("before...");          chain.doFilter(request, response);          System.out.println("after...");      }  chain.doFilter(request, response);这个方法的调用作为分水岭。事实上调用Servlet的doService()方法是在chain.doFilter(request, response);这个方法中进行的。  **拦截器**  拦截器是被包裹在过滤器之中的。      @Override      public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)throws Exception {          System.out.println("preHandle");          returntrue;          }      @Override      public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)throws Exception {          System.out.println("postHandle");          }      @Override      public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)throws Exception {          System.out.println("afterCompletion");          }  a.preHandle()这个方法是在过滤器的chain.doFilter(request, response)方法的前一步执行，也就是在 [System.out.println("before...")][chain.doFilter(request, response)]之间执行。  　　b.preHandle()方法之后，在return ModelAndView之前进行，可以操控Controller的ModelAndView内容。  　　c.afterCompletion()方法是在过滤器返回给前端前一步执行，也就是在[chain.doFilter(request, response)][System.out.println("after...")]之间执行。 |

## 5.10 什么是TCP IP 和HTTP协议

## **TCP协议：**

是面向连接的传输层协议。TCP/IP(Transmission Control Protocol/Internet Protocol) 即传输控制协议/网间协议，是一个工业标准的协议集，它是为广域网（WAN）设计的，是以太网协议和 IP 协议的上层协议也是应用层协议的下层协议，以太网协议解决了局域网的点对点通信。但是，以太网协议不能解决多个局域网如何互通，这由 IP 协议解决IP 协议可以连接多个局域网，

**IP 协议**

定义了一套自己的地址规则，称为 IP 地址。它实现了路由功能，允许某个局域网的 A 主机，向另一个局域网的 B 主机发送消息。按照域名去寻找服务器的IP地址---> DNS 把域名解析成IP地址

**HTTPS协议：**

超文本传输协议（HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。

所有的WWW文件都必须遵守这个标准。设计HTTP最初的目的是为了提供一种发布和接收HTML页面的方法。

HTTP是一个客户端和服务器端请求和应答的标准（TCP）。是基于请求响应模式的，客户端是终端用户，服务器端是网站。

通过使用Web浏览器、网络爬虫或者其它的工具，客户端发起一个到服务器上指定端口（默认端口为80）的HTTP请求。

客户端发送给服务器的格式教请求协议

服务器发送给客户端的格式叫响应协议

**HTTP**是无状态协议

**FIP**是有状态的

**URL**统一资源定位符，就是一个网址：协议名//域名，端口/路径。

**请求协议格式：**

**请求首行** ：请求方式/ 请求路径 /协议和版本

**请求头信息** ：请求头名称：请求头内容，即为key:value格式

**空行** ：用来与请求体隔开

请求体 ：GET没有请求体 POST有请求体

如果不是这个格式服务无法解读，有很多请求方法，最常用的就是GET和POST。请求体就是发给服务器的数据，比如登陆账号，FROM表单，账号密码一般不会出现在URL里面，会放在请求体里面，所以GET没有请求体，POST有请求体

## 5.11 GET和POST两种请求方式的区别

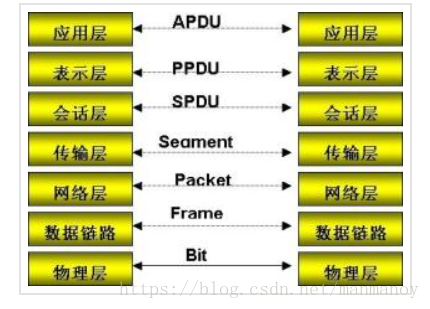
**GET请求常用操作：**

1. 参数url可见，在页面上的超链接
2. 通过拼接url进行传递参数
3. get一般传输数据大小不超过2k-4k

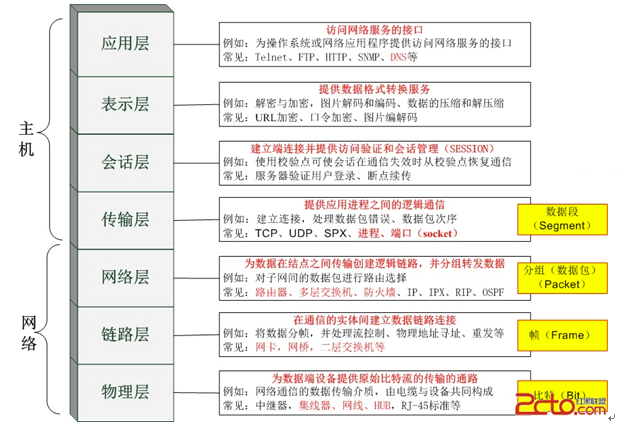
**POST :**

1. url参数不可见,数据不会出现在地址栏里，会在格式的请求头空格下面的请求体里
2. 数据大小没有上限
3. 有请求体
4. 请求体中如果有中文，会使用URL编码/ URL 统一资源定位符

## 5.12 OSI网络传输7层模型







1. **应用层（Application Layer）**

应用层是最靠近用户的OSI层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。。

协议有：HTTP FTP TFTP SMTP SNMP DNS TELNET HTTPS POP3 DHCP等。

应用层也称为应用实体（AE），它由若干个特定应用服务元素（SASE）和一个或多个公用应用服务元素（CASE）组成。每个SASE提供特定的应用服务，例如文件运输访问和管理（FTAM）、电子文电处理（MHS）、虚拟终端协议（VAP）等。CASE提供一组公用的应用服务，例如联系控制服务元素（ACSE）、可靠运输服务元素（RTSE）和远程操作服务元素（ROSE）等。主要负责对软件提供接口以使程序能使用网络服务。术语“应用层”并不是指运行在网络上的某个特别应用程序 ，应用层提供的服务包括文件传输、文件管理以及电子邮件的信息处理。

1. **表示层（Presentation Layer）**

数据的表示、安全、压缩。可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。

格式有：JPEG、ASCll、DECOIC、加密格式等。

应用程序和网络之间的翻译官，在表示层，数据将按照网络能理解的方案进行格式化；这种格式化也因所使用网络的类型不同而不同。

表示层管理数据的解密与加密，如系统口令的处理。例如：在 Internet上查询你银行账户，使用的即是一种安全连接。你的账户数据在发送前被加密，在网络的另一端，表示层将对接收到的数据解密。除此之外，表示层协议还对图片和文件格式信息进行解码和编码。

1. **会话层（Session Layer）**

建立、管理、终止会话，对应主机进程，指本地主机与远程主机正在进行的会话。

通过传输层（端口号：传输端口与接收端口）建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）。

负责在网络中的两节点之间建立、维持和终止通信。 会话层的功能包括：建立通信链接，保持会话过程通信链接的畅通，同步两个节点之间的对话，决定通信是否被中断以及通信中断时决定从何处重新发送。

你可能常常听到有人把会话层称作网络通信的“交通警察”。当通过拨号向你的 ISP （因特网服务提供商）请求连接到因特网时，ISP 服务器上的会话层向你与你的 PC 客户机上的会话层进行协商连接。若你的电话线偶然从墙上插孔脱落时，你终端机上的会话层将检测到连接中断并重新发起连接。会话层通过决定节点通信的优先级和通信时间的长短来设置通信期限。

1. **传输层（Transport Layer）**

定义传输数据的协议端口号，以及流控和差错校验。

协议有：TCP UDP等，数据包一旦离开网卡即进入网络传输层。

定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。

O S I 模型中最重要的一层。传输协议同时进行流量控制或是基于接收方可接收数据的快慢程度规定适当的发送速率。除此之外，传输层按照网络能处理的最大尺寸将较长的数据包进行强制分割。例如，以太网无法接收大于1 5 0 0 字节的数据包。发送方节点的传输层将数据分割成较小的数据片，同时对每一数据片安排一序列号，以便数据到达接收方节点的传输层时，能以正确的顺序重组。该过程即被称为排序。工作在传输层的一种服务是 T C P / I P 协议套中的T C P （传输控制协议），另一项传输层服务是I P X / S P X 协议集的S P X （序列包交换）。

1. **网络层（Network Layer）**

进行逻辑地址寻址，实现不同网络之间的路径选择。

协议有：ICMP IGMP IP（IPV4 IPV6） ARP RARP等。

在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。

O S I 模型的第三层，其主要功能是将网络地址翻译成对应的物理地址，并决定如何将数据从发送方路由到接收方。

网络层通过综合考虑发送优先权、网络拥塞程度、服务质量以及可选路由的花费来决定从一个网络中节点A 到另一个网络中节点B 的最佳路径。由于网络层处理，并智能指导数据传送，路由器连接网络各段，所以路由器属于网络层。在网络中，“路由”是基于编址方案、使用模式以及可达性来指引数据的发送。

网络层负责在源机器和目标机器之间建立它们所使用的路由。这一层本身没有任何错误检测和修正机制，因此，网络层必须依赖于端端之间的由D L L提供的可靠传输服务。

网络层用于本地L A N网段之上的计算机系统建立通信，它之所以可以这样做，是因为它有自己的路由地址结构，这种结构与第二层机器地址是分开的、独立的。这种协议称为路由或可路由协议。路由协议包括I P、N o v e l l公司的I P X以及A p p l e Ta l k协议。

网络层是可选的，它只用于当两个计算机系统处于不同的由路由器分割开的网段这种情况，或者当通信应用要求某种网络层或传输层提供的服务、特性或者能力时。例如，当两台主机处于同一个L A N网段的直接相连这种情况，它们之间的通信只使用L A N的通信机制就可以了(即OSI 参考模型的一二层)。

1. **数据链路层（Datalink Layer）**

建立逻辑连接、进行硬件地址寻址、差错校验等功能。（由底层网络定义协议）

将比特组合成字节进而组合成帧，用MAC地址访问介质，错误发现但不能纠正。

数据链路层协议的代表包括：SDLC、HDLC、PPP、STP、帧中继等。

定义了如何让格式化数据以进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。

OSI模型的第二层，它控制网络层与物理层之间的通信。它的主要功能是如何在不可靠的物理线路上进行数据的可靠传递。为了保证传输，从网络层接收到的数据被分割成特定的可被物理层传输的帧。帧是用来移动数据的结构包，它不仅包括原始数据，还包括发送方和接收方的物理地址以及检错和控制信息。其中的地址确定了帧将发送到何处，而纠错和控制信息则确保帧无差错到达。 如果在传送数据时，接收点检测到所传数据中有差错，就要通知发送方重发这一帧。

数据链路层的功能独立于网络和它的节点和所采用的物理层类型，它也不关心是否正在运行 Wo r d 、E x c e l 或使用I n t e r n e t 。有一些连接设备，如交换机，由于它们要对帧解码并使用帧信息将数据发送到正确的接收方，所以它们是工作在数据链路层的。

数据链路层（DataLinkLayer):在物理层提供比特流服务的基础上，建立相邻结点之间的数据链路，通过差错控制提供数据帧（Frame）在信道上无差错的传输，并进行各电路上的动作系列。

数据链路层在不可靠的物理介质上提供可靠的传输。该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。

1. **物理层（Physical Layer）**

建立、维护、断开物理连接。（由底层网络定义协议）

主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输,到达目的地后在转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。

O S I 模型的最低层或第一层，该层包括物理连网媒介，如电缆连线连接器。物理层的协议产生并检测电压以便发送和接收携带数据的信号。在你的桌面P C 上插入网络接口卡，你就建立了计算机连网的基础。换言之，你提供了一个物理层。尽管物理层不提供纠错服务，但它能够设定数据传输速率并监测数据出错率。网络物理问题，如电线断开，将影响物理层。 　　用户要传递信息就要利用一些物理媒体，如双绞线、同轴电缆等，但具体的物理媒体并不在OSI的7层之内，有人把物理媒体当做第0层，物理层的任务就是为它的上一层提供一个物理连接，以及它们的机械、电气、功能和过程特性。如规定使用电缆和接头的类型、传送信号的电压等。在这一层，数据还没有被组织，仅作为原始的位流或电气电压处理，单位是bit比特。

## 5.13长链接和短链接

### 5.13.1短链接

连接->传输数据->关闭链接

HTTP是无状态的，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连

接。也可以这样说：短连接是指SOCKET连接后发送完数据后马上断开连接。 http 1.0就是短链接,短链接的优点是节省网络带宽

### 5.13.2长链接

连接->传输数据->保持连接 -> 传输数据-> 。。。 ->关闭连接。

长连接指建立SOCKET连接后不管是否使用都保持连接，但安全性较差。

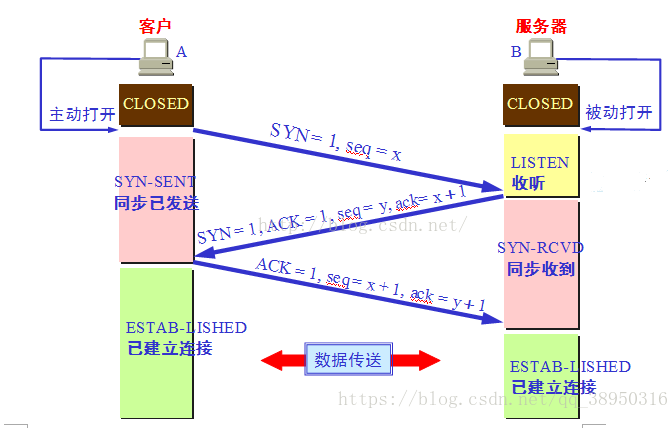
http1.1 支持长链接,但仍然是无状态传输

H5的WebSocket是长链接的,我们可以用它来开发聊天室、股票实时行情等

长链接的优点是 能够进行服务端向前端的消息推送和数据的实时更新

## 5.14 三次握手、四次挥手

### 5.14.1 三次握手过程理解

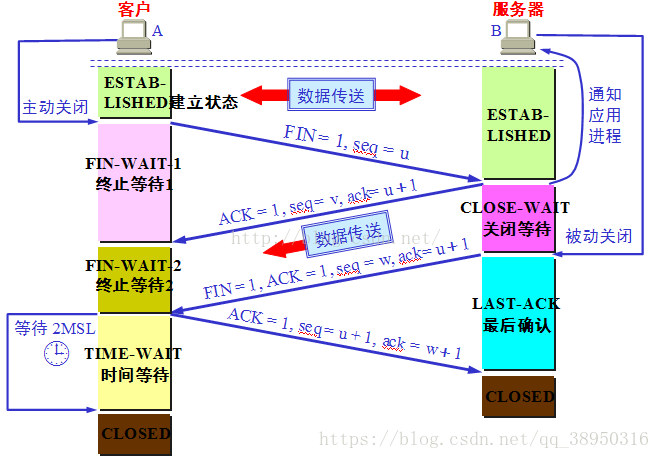


第一次握手：建立连接时，客户端发送syn包（syn=j）到服务器，**并进入SYN\_SENT状态**，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器**进入SYN\_RECV状态**；

第三次握手：客户端收到服务器的SYN+ACK包**，向服务器发送确认包**ACK(ack=k+1），此包发送完毕，客户端和服务器进入**ESTABLISHED**（TCP连接成功）状态，**完成三次握手**

### 5.14.2 四次挥手过程理解



1）**客户端进程发出连接释放报文**，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为seq=u（等于前面已经传送过来的数据的最后一个字节的序号加1），此时，客户端进入FIN-WAIT-1（终止等待1）状态。 TCP规定，FIN报文段即使不携带数据，也要消耗一个序号。

2）**服务器收到连接释放报文，发出确认报文**，ACK=1，ack=u+1，并且带上自己的序列号seq=v，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

3）**客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态**，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

4）**服务器将最后的数据发送完毕后，就向客户端发送连接释放报文**，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为seq=w，此时**，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。**

5）**客户端收到服务器的连接释放报文后，必须发出确认**，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2∗∗MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。

6）**服务器只要收到了客户端发出的确认，立即进入CLOSED状态。**同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

### 5.14.3 为什么连接的时候是三次握手，关闭的时候却是四次握手？

答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

### 5.14.4 为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。在Client发送出最后的ACK回复，但该ACK可能丢失。Server如果没有收到ACK，将不断重复发送FIN片段。所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME\_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

### 5.14.5 为什么不能用两次握手进行连接？

答：3次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

       现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机S和C之间的通信，假定C给S发送一个连接请求分组，S收到了这个分组，并发 送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，C在S的应答分组在传输中被丢失的情况下，将不知道S 是否已准备好，不知道S建立什么样的序列号，C甚至怀疑S是否收到自己的连接请求分组。在这种情况下，C认为连接还未建立成功，将忽略S发来的任何数据分 组，只等待连接确认应答分组。而S在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

### 5.14.6 如果已经建立了连接，但是客户端突然出现故障了怎么办？

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

# 六 脚本语言

## 6.1 Bootstrap

## 6.2 JavaScript

## 6.3 Jquery

## 6.4 React

## 6.5 Vue

## EasyUI

## LayUI

# 七 数据库

## 7.1数据库语言类型

**1. 数据查询语言DQL**

数据查询语言DQL基本结构是由SELECT子句，FROM子句，WHERE

子句组成的查询块：

SELECT <字段名表>

FROM <表或视图名>

WHERE <查询条件>

**2 .数据操纵语言DML**

数据操纵语言DML主要有三种形式：

1) 插入：INSERT

2) 更新：UPDATE

3) 删除：DELETE

**3. 数据定义语言DDL**

数据定义语言DDL用来创建数据库中的各种对象-----表、视图、

索引、同义词、聚簇等如：

CREATE TABLE/VIEW/INDEX/SYN/CLUSTER

表 视图 索引 同义词 簇

DDL操作是隐性提交的！不能rollback

**4. 数据控制语言DCL**

数据控制语言DCL用来授予或回收访问数据库的某种特权，并控制

数据库操纵事务发生的时间及效果，对数据库实行监视等。如：

1) GRANT：授权。

2) ROLLBACK [WORK] TO [SAVEPOINT]：回退到某一点。

回滚---ROLLBACK

回滚命令使数据库状态回到上次最后提交的状态。其格式为：

SQL>ROLLBACK;

3) COMMIT [WORK]：提交。

## 7.2 锁机制

### 7.2.1 sql锁：

**悲观锁**：就是考虑问题很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁；

实现：sql语句后边加上for update

例子：Select id,nam from biao for update

**乐观锁**:就是考虑问题很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁

实现：在表里边加一个vesion

例子：

Select max(nub) ,version from biao

Update biao set nub=nub+1,version=vsersion+1 where id=id and version =version

### 7.2.2 方法锁

方法锁主要包括：synchronized锁和lock锁

区别：

1）Lock是一个接口，而synchronized是Java中的关键字;

2）**synchronized**当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁,不能够响应中断,（释放：1执行完线程自动释放2发生异常jvm让线程释放）（（比如调用sleep方法）），这样的好处是不会导致死锁现象发生。

**Lock**锁，可以不让等待的线程一直无期限地等待下去，比如只等待一定的时间或者响应中断。

但Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时必须在try{}catch{}块中进行，需要在finally块中释放锁；

3）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

　　在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

在并发量比较小的情况下，使用synchronized是个不错的选择，但是在并发量比较高的情况下，其性能下降很严重，此时ReentrantLock（可重入锁，唯一实现了Lock接口的类）是个不错的方案。

## MySQL

## Oracle

## Redis

## MongoDB

# 八、框架技术

## 8.2 dubbox框架

**在service的配置文件中,如何配置dubbo服务**

<dubbo:application name="dubboxdemo-service"/> 服务注册的名称

<dubbo:registry address="zookeeper://192.168.25.132:2181"/> 注册中心的ip和端口

<dubbo:annotation package="cn.itcast.dubboxdemo.service" /> 要注册的服务所在包

**在web消费方的配置文件中,如何配置dubbo服务**

<!-- 引用dubbo 服务 -->

<dubbo:application name="dubboxdemo-web" />

<dubbo:registry address="zookeeper://192.168.25.132:2181"/>

<dubbo:annotation package="cn.itcast.dubboxdemo.controller" />

## 8.10 SpringMVC中Restful风格的参数接收方式有哪些??

|  |
| --- |
| 当使用@RequestMapping URI template 样式映射时， 即 Url/{partyId}, 这时的paramId可通过 @Pathvariable注解绑定它传过来的值到方法的参数上。 |
| **@RequestMapping("/hello/{username}/{age}")**  public String hello(**@PathVariable String username**,@PathVariable int age,Model model) {  System.out.println(username +"~"+age);  model.addAttribute("msg", "hello springMvc4,hello World4~");  return "hello";  } |
|  |
| 用@RequestParam(“partyId”)的方式绑定参数值。 |
| **@RequestMapping(method = RequestMethod.GET)**  public String setupForm**(@RequestParam("partyId") int partyId**) { |
|  |
| 参数是实体类 |
| @RequestMapping("/demo")  public String fun01(**@ResponseBody People people)** { |
|  |
| 接收数组参数 |
| @RequestMapping(params = "fn=deleteCatalogSchemes")  @ResponseBody  public Result deleteCatalogSchemes(**@RequestParam("idList[]") List<String> idList**) { |
|  |
| 接收List集合 |
| @RequestMapping(params = "fn=saveUsers")  @ResponseBody  public Result saveUsers(**@RequestBody List<User> userList**) { |
|  |
| 参数是文件 |
| @RequestMapping(value = "upload")  @ResponseBody  public void uploadExce**l(@RequestParam("file") MultipartFile file)** throws Exception { |
|  |
| 使用自定义的vo容器里接收前端的多表数据 |
|  |

# 数据结构

## 队列

## 链表

## 数组

## 二叉树

# 算法

## 交换算法

## 冒泡排序算法

|  |
| --- |
| 顺口溜  外层循环n-1  内层循环n-i-1 |
| /\*\*  \* @program: JavaSpecialityDeep  \* @author: Mr.Zerah  \* @create: 2018-10-25 22:52  \* @description: 冒泡排序  \* 冒泡排序只会操作相邻的两个数据。每次冒泡操作都会对相邻的两个元素进行比较，看是否满足大小关系要求。  \* 如果不满足就让它俩互换。一次冒泡会让至少一个元素移动到它应该在的位置，重复n 次，  \* 就完成了 n 个数据的排序工作。  \*\*/  public class BubbleSort {  public void bubbleSort(Integer[] arr, int n) {  if (n <= 1) return; //如果只有一个元素就不用排序了    **for (int i = 0; i < n; i++) {**  // 提前退出冒泡循环的标志位,即一次比较中没有交换任何元素，这个数组就已经是有序的了  boolean flag = false;  **for (int j = 0; j < n - i - 1; j++) {**  //此处你可能会疑问的j<n-i-1，因为冒泡是把每轮循环中较大的数飘到后面，  // 数组下标又是从0开始的，i下标后面已经排序的个数就得多减1，总结就是i增多少，j的循环位置减多少  if (arr[j] > arr[j + 1]) { //即这两个相邻的数是逆序的，交换  int **temp** = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  flag = true;  }  }  if (!flag) break;//没有数据交换，数组已经有序，退出排序  }  }    public static void main(String[] args) {  Integer arr[] = {2, 4, 7, 6, 8, 5, 9};  SortUtil.show(arr);  BubbleSort bubbleSort = new BubbleSort();  bubbleSort.bubbleSort(arr, arr.length);  SortUtil.show(arr);  }  } |

## 插入算法

|  |
| --- |
| **package** demo;  **import** java.util.Random;  */\*\*  \* 插入排序: 从一个值开始向前面插入  \* 19 前面没有数,则不用插入  \* 3 向前插入,直到找到前面比它小的数字或没有数字则停下来  \* 3 19 56 8 55 89 1  \*  \* 3 3 19 56 8 55 89 1  \* 19 3 19 56 8 55 89 1  \* 56 3 19 56 8 55 89 1  \* 8 3 8 19 56 55 89 1  \* 55 3 8 19 55 56 89 1  \* 56 3 8 19 55 56 89 1  \* 89 3 8 19 55 56 89 1  \* 1 1 3 8 19 55 56 89  \*/* **public class** InsertSort {   **public static void** main(String[] args) {   **int** [] arrays = **new int**[8];   **for**(**int** i = 0;i<arrays.**length**;i++) {   Random random = **new** Random();  arrays[i] = random.nextInt(100);  }    **for**(**int** in: arrays){  System.***out***.println(in);  }   System.***out***.println(**"-----------------------------"**);   **int**[] ins2 = *sort*(arrays);   **for**(**int** in: ins2){  System.***out***.println(in);  }   }   */\*\*  \* 插入算法  \* 19 3 19 56 8 55 89 1  \* 56 3 19 56 8 55 89 1  \* 8 3 8 19 56 55 89 1  \* 56 3 8 19 56 55 89 1  \* 55 3 8 19 55 56 89 1  \** ***@param ins*** *\** ***@return*** *\*  \*/* **public static int[] sort(int[] ins){   for(int i=1; i< ins.length; i++){   for(int j=i; j>0; j--){   if(ins[j] < ins[j-1]){   int temp = ins[j-1];  ins[j-1] = ins[j];  ins[j] = temp;  }  }  }    return ins;  }** } |

# 设计模式

# 微信小程序

# 大数据

# 其他

## 你项目中常用的工具类有哪些

RandomUtils

StringUtils