1. **基础部分**

1.Java开发环境:

1)java编译运行过程:

1.1)编译期:.java源文件，经过编译，生成.class字节码文件

1.2)运行期:JVM加载.class并运行.class

特点:跨平台、一次编程到处使用

2)名词解释:

2.1)JVM:java虚拟机

加载.class并运行.class

2.2)JRE:java运行环境

除了包含JVM以外还包含了运行java程序所必须的环境

JRE=JVM+java系统类库(小零件)

2.3)JDK:java开发工具包

除了包含JRE以外还包含了开发java程序所必须的命令工具

JDK=JRE+编译、运行等命令工具

说明:

2.1)运行java程序的最小环境为JRE

2.2)开发java程序的最小环境为JDK

3)配置环境变量

3.1)JAVA\_HOME:指向jdk的安装路径

3.2)CLASSPATH:表示类的搜索路径(一般简写为.)

3.3)PATH:指向jdk下的bin目录

2.eclipse:

1)IBM、Java语言开发的、开源的、免费的、不需要安装仅需解压即可

2)开发步骤:

2.1)新建Java项目/工程---------------小区

2.2)新建Java包----------------------楼+单元

2.3)新建Java类----------------------房子

main中:

System.out.println("HelloWorld");

3)注释:解释性文字

3.1)单行注释://

3.2)多行注释:/\* \*/

3.3)文档注释:/\*\* \*/

3.工作空间:所创建的所有java程序的默认保存地址

4.变量:存数的，代词，指代的就是它所存的那个数

1)声明:

int a; //声明一个整型的变量，名为a

int b,c,d; //声明三个整型的变量，名为b,c,d

2)初始化:第一次赋值

2.1)声明的同时初始化:

int a = 250; //声明整型变量a并赋值为250

2.2)先声明后初始化:

int a; //声明整型变量a

a = 250; //给变量a赋值为250

3)使用:

3.1)对变量的使用就是对它所存的那个数的使用

int a = 5;

int b = a+10; //取出a的值5，加10后，再赋值给整型变量b

System.out.println(b); //输出变量b的值15

System.out.println("b"); //输出b，双引号中的原样输出

a = a+10; //取出a的值5，加10后，再赋值给a

//在a本身基础之上增10

System.out.println(a); //15

3.2)变量在使用时必须类型匹配

int a = 3.14; //编译错误，数据类型不匹配

3.3)变量在用之前必须声明并初始化

System.out.println(m); //编译错误，m未声明

int m;

System.out.println(m); //编译错误，m未初始化

4)命名:

4.1)只能包含字母、数字、\_和$符，并且不能以数字开头

4.2)严格区分大小写

4.3)不能使用关键字

4.4)可以中文命名，但不建议

建议"英文的见名知意"、"驼峰命名法"

5.基本数据类型:共8种

1)byte:1个字节即:8位,用于表示最小数据单位,如文件中数据,-128～127

2)short:短整型，2个字节，-32768～32767

3)int:整型，4个字节，-21个多亿到21个多亿

3.1)整数直接量默认为int型，但不能超范围，超范围则编译错误

3.2)两个整数相除，结果还是整数，小数位无条件舍弃(不会四舍五入)

3.3)运算时超范围，则发生溢出，溢出是需要避免的

4)long:长整型，8个字节，很大很大很大

4.1)表示长整型直接量，需在数字后加L或l

4.2)运算时若有可能溢出，建议在第1个数字后加L

4.3)System.currentTimeMillis()用于获取自1970.1.1零时到此时此刻的毫秒数

5)float:单精度浮点型，4个字节，后缀:F或f,1位符号位,8位指数,23位有效尾数

6)double:双精度浮点型，8个字节，后缀:D或d,1位符号位,11位指数,52位有效尾数

6.1)浮点数直接量默认为double型，若想表示float需在数字后加f或F

6.2)double和float在运算时，有可能会出现舍入误差(精确运算场合不能使用)

7)boolean:布尔型，1个字节

7.1)只能取值为true和false

8)char:字符型，2个字节

8.1)采用Unicode字符集编码，一个字符对应一个码(0到65535之间)

表现的形式的字符char，但本质上是码int

ASCII码('a'--97 'A'--65 '0'--48)

8.2)字符直接量必须放在单引号中，只能有一个

8.3)特殊符号需通过\来转义

6.直接量超范围为编译错误，运算超范围为溢出

7.基本数据类型间的转换:

数据类型从小到大依次为:

byte <（short=char）< int < long < float < double

注：boolean类型与所有其他7种类型都不能进行转换

1)两种方式:

1.1)自动类型转换:小类型到大类型

1.2)强制类型转换:大类型到小类型

eg: (要转换成为的数据类型)变量

强转有可能溢出或丢失精度

2)两点规则:

2.1)整数直接量可以直接赋值给byte,short,char，但不能超出范围

2.2)byte,short,char型数据参与运算时，先一律转换为int再运算

8.运算符:

1)算术运算符:+,-,\*,/,%,++,--

2)关系运算符:>,<,>=,<=,==,!= boolean

3)逻辑运算符:&&,||,! boolean

4)赋值运算符:=,+=,-=,\*=,/=,%=

5)字符串连接运算符:+

6)条件/三目运算符:boolean?数1:数2

9.分支结构:

1)if结构:1条路

2)if...else结构:2条路

3)if...else if结构:多条路

4)switch...case结构:多条路

优点:效率高、结构清晰

缺点:整数、相等

break:跳出switch

10.循环:反复执行一段相同或相似的代码

循环三要素:

1)循环变量的初始化

2)循环的条件(以循环变量为基础)

3)循环变量的改变(向着循环的结束变)

循环变量:在整个循环过程中所反复改变的那个数

1. 循环结构

1)while结构:先判断后执行，有可能一次都不执行

2)do...while结构:先执行后判断，至少执行一次

3)for结构:应用率最高，固定次数循环

11.break:跳出循环

continue:跳过循环体中剩余语句而进入下一次循环

12.三种循环结构的更佳适用情况:

1)while:"当..."

要素1与要素3不同时首选while

2)do...while:"直到..."

要素1与要素3相同时首选do...while

3)for:固定次数循环，应用率最高

13.闰年的判断条件:

能被4整除并且不能被100整除或者能被400整除

代码表现为:(year%4==0 && year%100!=0) || year%400==0

14.嵌套循环:

1)循环中套循环，一般多行多列时使用，外层控制行，内层控制列

2)执行规则:外层循环走一次，内层循环走所有次

3)建议:嵌套层数越少越好，能用一层就不用两层，能用两层就不用三层

若需求必须要使用三层以上的循环才能解决，说明设计有问题

4)break只能跳出一层循环

15.程序=算法+数据结构

1)算法:解决问题的流程/步骤(顺序、分支、循环)

2)数据结构:将数据按照某种特定的结构保存,设计良好的/合理的数据结构导致好的算法。

16.数组:

1)是一种数据类型(引用类型)

2)相同数据类型元素的集合

3)数组的定义:

int[] arr = new int[10];

4)数组的初始化:

int[] arr = new int[3]; //0,0,0

int[] arr = {2,5,8}; //2,5,8

int[] arr = new int[]{2,5,8}; //2,5,8

int[] arr;

arr = {2,5,8}; //编译错误，此方式只能声明同时初始化

arr = new int[]{2,5,8}; //正确

5)数组的访问:

5.1)通过数组名.length可以获取数组的长度(元素的个数)

int[] arr = new int[10];

System.out.println(arr.length); //10

5.2)通过数组的下标/索引来访问元素

下标从0开始，最大到(数组的长度-1)

int[] arr = new int[3];

arr[0] = 100; //给arr中第1个元素赋值为100

arr[1] = 200; //给arr中第2个元素赋值为200

arr[2] = 300; //给arr中第3个元素赋值为300

arr[3] = 400; //数组下标越界异常

System.out.println(arr[arr.length-1]); //输出最后一个元素的值

6)数组的遍历:

int[] arr = new int[10];

for(int i=0;i<arr.length;i++){

arr[i] = 100;

}

for(int i=0;i<arr.length;i++){

System.out.println(arr[i]);

}

for(int i=arr.length-1;i>=0;i--){

System.out.println(arr[i]);

}

7)数组的复制:

7.1)System.arraycopy(a,1,a1,0,4); //效率高、灵活性好

a:代表源数组

1:源数组的开始起始位置

a1:目标数组

0:目标数组的开始起始位置

4:要复制的数组的长度

7.2)int[] a1 = Arrays.copyOf(a,6); //效率低、灵活性差

a = Arrays.copyOf(a,a.length+1); //数组的扩容/缩容

8)数组的排序:

8.1)Arrays.sort(arr); //升序，效率高

8.2)冒泡排序:

8.2.1)四个数冒三轮

8.2.2)每一轮都是从第1个元素开始冒

每一次都是和它的下一个元素比

8.2.3)冒出来的就不再参与比较了

17.方法:

1)封装一段特定的业务逻辑功能

2)尽可能独立，一个方法只干一件事

3)可以被反复多次调用

4)减少代码的重复，有利于代码的维护，有利于团队的协作

18.方法的定义:

修饰词 返回值类型 方法名(参数列表){ //形参

方法体

}

19.方法的调用:

1)无返回值: 方法名(有参传参); //实参

2)有返回值: 数据类型 变量 = 方法名(有参传参);

20.return:

1)return 值; //1.1)结束方法的执行 1.2)返回结果给调用方

2)return; //2.1)结束方法的执行(在某种条件之下提前结束方法)

**二、面向对象**

1.什么是类？什么是对象？

1)现实世界是由很多很多对象组成的,基于对象抽出了类

2)对象:真实存在的单个的个体

类:类型/类别，代表一类个体

3)类中可以包含:

3.1)所有对象所共有的属性/特征------------成员变量

3.2)所有对象所共有的行为-----------------方法

4)一个类可以创建多个对象，

同一类型所创建的对象，结构相同，数据不同

5)类是对象的模板，对象是类的具体的实例

2.方法的签名:方法名+参数列表

3.方法的重载(Overload):

1)发生在同一个类中，方法名称相同，参数列表不同，方法体不同

2)编译器在编译时会根据方法的签名自动绑定调用的方法

4.构造方法:构造函数、构造器、构建器

1)给成员变量赋初值

2)与类同名，没有返回值类型

3)在创建对象时被自动调用

4)若自己不写构造方法，则编译器默认一个无参构造方法，

若自己写了构造方法，则不再默认提供

5)构造方法可以重载

5.this:指代当前对象，哪个对象调用方法它指的就是哪个对象

只能用在方法中，方法中访问成员变量之前默认有个this.

this的用法:

1)this.成员变量名--------------访问成员变量

2)this.方法名()----------------调用方法(一般不用)

3)this()-----------------------调用构造方法

6.null:空，没有指向任何对象

若引用的值为null，则该引用不能再进行任何操作了

若操作则发生NullPointerException空指针异常

7.引用类型之间画等号:

1)指向同一个对象

2)通过一个引用对数据的修改会影响另一个引用对数据的访问

eg:房子钥匙

8.基本类型之间画等号:

1)赋值

2)对一个变量的修改不会影响另一个变量

eg:身份证复印件

9.成员变量和局部变量是可以同名的，但是用的时候采用就近原则,当成员变量与局部变量同名时，this不能省略。

10.引用类型数组:

1)Student[] stus = new Student[3];

stus[0] = new Student("zhangsan",25,"LF");

stus[1] = new Student("lisi",26,"JMS");

stus[2] = new Student("wangwu",24,"SD");

2)Student[] stus = new Student[]{

new Student("zhangsan",25,"LF"),

new Student("lisi",26,"JMS"),

new Student("wangwu",24,"SD")

};

3)int[][] arr = new int[3][];------------数组的数组

arr[0] = new int[2];

arr[1] = new int[3];

arr[2] = new int[2];

arr[1][0] = 100;

4)int[][] arr = new int[3][4]; //3行4列

for(int i=0;i<arr.length;i++){

for(int j=0;j<arr[i].length;j++){

arr[i][j] = 100;

}

}

11.继承:

1)作用:代码复用

2)通过extends来实现继承

3)超类/父类:所有派生类所共有的属性和行为

派生类/子类:派生类所特有的属性和行为

4)派生类继承超类后，派生类具有:派生类的+超类的

5)一个超类可以有多个派生类

一个派生类只能有一个超类----------单一继承

6)继承具有传递性的

7)java规定:在构造派生类之前必须先构造超类

在派生类的构造方法中若没有调用超类的构造方法

------则默认super()调用超类的无参构造方法

在派生类的构造方法中若自己调用了超类的构造方法

------则不再默认提供

super()调用超类构造方法必须位于派生类构造方法的第一行

12.super:指代当前对象的超类对象

super的用法:

1)super.成员变量名-------------访问超类的成员变量

2)super.方法名()---------------调用超类的方法

3)super()----------------------调用超类的构造方法

13.问:派生类可以继承超类的构造方法吗?

答:不可以，超类构造是由派生类来调用的

14.向上造型:

1)超类型的引用指向派生类的对象

2)能点出来什么，看引用的类型

15.方法的重写(Override):重新写、覆盖

1)发生在父子类中，方法名称相同，参数列表相同，方法体不同

2)重写方法被调用时，看对象的类型

3)重写遵循"两同两小一大"原则:

3.1)两同:

3.1.1)方法名称相同

3.1.2)参数列表相同

3.2)两小:

3.2.1)派生类方法的返回值类型小于或等于超类方法的

1)void时，必须相等

2)基本数据类型时，必须相等

3)引用数据类型时，小于或等于

3.2.2)派生类方法抛出的异常小于或等于超类方法的

3.3)一大:

3.3.1)派生类方法的访问权限大于或等于超类方法的

16.重写与重载的区别:---------------常见面试题

1)重写(Override):

1.1)发生在父子类中，方法名称相同，参数列表相同，方法体不同

1.2)遵循"运行期"绑定，看对象的类型来调用方法

2)重载(Overload):

2.1)发生在一个类中，方法名称相同，参数列表不同，方法体不同

2.2)遵循"编译期"绑定，看参数/引用的类型来绑定方法

17.package:

1)避免类名的冲突

2)包名可以有层次结构，同包中的类不能同名

3)类的全称为: 包名.类名

4)建议:包名所有字母都小写

import:

1)同包中的类可以直接访问，

不同包中的类不可以直接访问，想访问只有如下两种方式:

1.1)先import声明类再访问类----建议

1.2)类的全称------------------太繁琐

18.访问控制修饰符:

1)public:公开的，任何类

2)private:私有的，本类

3)protected:受保护的，本类、子类、同包类

4)默认的:什么也不写，本类、同包类

说明:

1)类的访问修饰符只能是public或默认的

2)类中成员的访问修饰符如上四种都可以

19.final:最终的、不可改变的----------应用率极低

1)修饰变量:变量不能被改变

2)修饰方法:方法不能被重写

3)修饰类:类不能被继承

20.static:静态的

1)静态变量:

1.1)由static修饰的

1.2)属于类的，存储在方法区中，只有一份

1.3)常常通过类名点来访问

1.4)何时用:所有对象所共享的数据(图片、音频、视频等)

2)静态方法:

2.1)由static修饰的

2.2)属于类的，存储在方法区中，只有一份

2.3)常常通过类名点来访问

2.4)静态方法没有隐式this传递的，

静态方法中不能直接访问实例成员

2.5)何时用:方法的操作仅与参数相关而与对象无关

3)静态块:

3.1)由static修饰的

3.2)在类被加载期间自动执行，因类只被加载一次，

所以静态块也只执行一次

3.3)何时用:初始化/加载静态资源(图片、音频、视频等)

21.static final常量:

1)必须声明同时初始化

2)常常通过类名点来访问、不能被改变

3)建议:常量名所有字母都大写，多个单词之间用\_分隔

4)编译器在编译时会将常量直接替换为具体的值，效率高

5)何时用:数据永远不变，并且经常使用

22.抽象方法:

1)由abstract修饰

2)只有方法的定义，没有具体的实现(连{}都没有)

23.抽象类:

1)由abstract修饰

2)包含抽象方法的类必须是抽象类

不包含抽象方法的类也可以声明为抽象类-------我乐意

3)抽象类不能被实例化

4)抽象类是需要被继承的，派生类:

4.1)重写所有抽象方法------变不完整为完整

4.2)也声明为抽象类--------不常用

5)抽象类的意义:

5.1)封装派生类所共有的属性和行为--------代码复用

5.2)为所有派生类提供一种统一的类型------向上造型

5.3)可以包含抽象方法，为所有派生类提供统一的入口，

派生类的具体实现不同，但入口是一致的

24.成员内部类:应用率低

1)类中套类，外面的称为Outer外部类，里面的称为Inner内部类

2)内部类只服务于外部类，对外不具备可见性

3)内部类对象通常是在外部类中创建的

4)内部类中可以直接访问外部类的成员(包括私有的)

内部类中有个隐式的引用指向了创建它的外部类对象

eg: 外部类名.this.

25.匿名内部类:

1)若想创建一个类(派生类)的对象，并且对象只被创建一个，

此时该类不必命名，称之为匿名内部类

2)jdk1.7(含)以前，匿名内部类中想访问外面的变量，

该变量必须是final的

26.面试题:

问:内部类有独立的.class吗?

答:有

27.接口:

1)是一种数据类型(引用类型)

2)由interface定义

3)只能包含常量和抽象方法

4)接口不能被实例化

5)接口是需要被实现/继承的，实现类/派生类:

必须重写接口中的所有抽象方法

6)一个类可以实现多个接口，用逗号分隔

若又继承又实现时，必须先继承后实现

7)接口可以继承接口

28.java中写某个功能的步骤:

1)先写行为/方法:

a.若行为为某一个对象所特有的行为，则设计在特定的类中

b.若行为为所有对象所共有的行为，则设计在超类中

2)页面去调:

a.定时触发的，在run中调用

b.事件触发的，在侦听器中调用

29.设计规则:

1)将所有派生类所共有的属性和行为，抽到超类中----抽共性

2)派生类的行为都一样，设计为普通方法

派生类的行为不一样，设计为抽象方法

3)将部分派生类所共有的行为，设计到接口中

符合既是也是原则时，使用接口

接口是对继承单根性的扩展-------------实现多继承

30.多态:

1)多态的意义:

1.1)同一类型的引用指向不同的对象时，有不同的实现

----行为的多态:cut(),step(),getImage()...

1.2)同一个对象被造型为不同的类型时，有不同的功能

----对象的多态:我，你，水，被撞的敌人...

2)向上造型/自动类型转换:

2.1)超类型的引用指向派生类的对象

2.2)能造型成为的类型有: 超类+所实现的接口

2.3)能点出来什么，看引用的类型

3)强制类型转换，成功的条件只有如下两种:

3.1)引用所指向的对象，就是该类型

3.2)引用所指向的对象，实现该接口或继承的该类

4)强转时若不符合如上两个条件，则发生ClassCastException类型转换异常，

建议:强转之前先通过instanceof来判断引用的对象是否是该类型

31.内存管理:由JVM来管理

1)堆:

1.1)存储new出来的对象(包括实例变量)

1.2)垃圾:没有任何引用所指向的对象

垃圾回收器(GC)不定时到内存中清扫垃圾，

回收过程是透明的(看不到的)，

不一定一发现垃圾就立刻回收，

调用System.gc()建议虚拟机尽快调度GC来回收垃圾

1.3)实例变量的生命周期:

1.3.1)类中，方法外

1.3.2)有默认值的

1.3.3)创建对象时存储堆中，对象被回收时一并被回收

1.4)内存泄漏:不再使用的对象还没有被及时的回收

建议:不再使用的对象及时将引用设置为null

2)栈:

2.1)存储正在调用的方法中的所有局部变量(包括方法的参数)

2.2)调用方法时在栈中为该方法分配一块对应的栈帧，

栈帧中包含局部变量(包括方法的参数),

方法调用结束时，栈帧被清除，局部变量一并被清除

2.3)局部变量的生命周期:

2.3.1)方法中

2.3.2)没有默认值的

2.3.3)调用方法时存储在栈中，方法结束栈帧被清除，局部变量一并失效

3)方法区:

3.1)存储.class字节码文件(包括方法、静态变量)

3.2)方法只有一份，通过this来区分具体的调用对象

32.面向对象三大特征:

1.封装:

1)类:封装的是对象的属性和行为

2)方法:封装的是一段特定的业务逻辑功能

3)访问控制修饰符:封装的是具体访问的权限

2.继承:

1)作用:代码复用

2)超类:所有派生类所共有的属性和行为

接口:部分派生类所共有的行为

派生类:派生类所特有的属性和行为

3)传递性，单一继承、多接口实现

3.多态:

1)意义:行为的多态(所有抽象方法都是多态的)

对象的多态(所有对象都是多态的)

2)向上造型、强制类型转换、instanceof判断

3)多态的表现形式:

3.1)重写:根据对象的不同来实现多态

3.2)重载:根据参数的不同来实现多态

**三、API**

(一)String

定义：String是不变对象，即:字符串对象一旦创建，内容不可改变，想改变内容一定创建新对象。

注意：

1.java建议创建字符串对象时使用字面量形式直接赋值。因为这样创建时java会重用 已经创建过的字符串对象。使用字面量创建字符串对象时，JVM会首先检查常量池是 否已经缓存过该内容的字符串对象，若有则直接重用该对象，若没有才创建该字符串 对象并缓存它。

2.编译器有一个优化措施:当编译器在编译代码时发现一个计算表达式的参数都是字 面量时，那么会直接计算，并将结果编译到class文件中。所以下面的代码会被编译 器改为: String s5 = "123abc";

eg:String s5 = "123"+"abc";

3.频繁修改字符串带来的性能损耗

1.char charAt(int index):返回当前字符串中给定位置对应的字符(字符串中单个字符的下标从0开始，空格也算为一个字符)

案列:检测回文

String line = "上海自来水来自海上";

for(int i=0;i<line.length()/2;i++){

if(line.charAt(i)!=line.charAt(line.length()-1-i)) {

System.out.println("不是回文");

return;

}

}

System.out.println("是回文");

2.int indexOf(String str):查找给定字符串在当前字符串中的位置。若当前字符串中不含有该内容则返回值为-1(字符串中单个字符的下标从0开始，空格也算为一个字符)

案例:String str = "thinking in java";

//查找第一次出现"in"的位置

int index = str.indexOf("in");

System.out.println(index);//2

//重载方法，从指定位置开始查找第一次出现的位置

index = str.indexOf("in", 3);

System.out.println(index);//5

//查找最后一次出现给定字符的位置

index = str.lastIndexOf("in");

System.out.println(index);//9

3.String方法:int length() 该方法可以返回当前字符串长度(字符个数)

案列: String str = "我爱java";

System.out.println(str.length());//6

4.boolean startsWith(String str)

boolean endsWith(String str)

判断字符串是否是以给定的字符串开始或结束的

案列:String str = "thinking in java";

boolean starts = str.startsWith("thi");

System.out.println("starts:"+starts);//true

boolean ends = str.endsWith("ava");

System.out.println("ends:"+ends);//true

5.String substring(int start,int end):截取当前字符串指定范围内的字符串。需要注意，java API有个特点，通常使用两个数字表示范围时，都是“含头不含尾”的。(字符串中单个字符的下标从0开始，空格也算为一个字符)

案列: String str = "www.baidu.com";

String sub = str.substring(4,9);

System.out.println(sub);//baidu

6.String toUpperCase()

String toLowerCase()

将当前字符串中的英文部分转换为全大写或全小写

案列: String str = "我爱Java";

String upper = str.toUpperCase();

System.out.println("upper:"+upper);//“我爱JAVA”

String lower = str.toLowerCase();

System.out.println("lower:"+lower);//“我爱java”

7.String trim():去除当前字符串两边的空白字符

案列: String str = " hello ";

str = str.trim();

System.out.println(str);//hello

8.String提供了一组静态的valueOf方法，作用是将其他类型数据转换为字符串。比较常用的是将基本类型转换为字符串。

案列: int d = 123;

String str = String.valueOf(d);//等价于String str2 = d+"";

System.out.println(str);//123

1. 由于字符串的设计及优化是针对重用性，所以字符串不适合频繁修改。java设计了一个专门针对字符串修改操作的类:StringBuilder,内部维护一个可变的字符数组，使用它修改字符串方便，并且开销小。需要注意，java API有个特点，通常使用两个数字表示范围时，都是“含头不含尾”的。(字符串中单个字符的下标从0开始，无最大值限制，标点符号也算为一个字符）

相关方法的案例:

String str = "努力学习java";

//表示空字符串 [StringBuilder builder = new StringBuilder();]

StringBuilder builder = new StringBuilder(str);

builder.append(",为了找个好工作!");//努力学习java,为了找个好工作!

//获取StringBuilder内部改变后的字符串

String s = builder.toString();

System.out.println(s);//努力学习java,为了找个好工作!

builder.replace(9, 16, "就是为了改变世界");努力学习java,就是为了改变世界!

builder.delete(0, 8);//,就是为了改变世界!

builder.insert(0, "活着");//活着,就是为了改变世界!

1. 字符串支持正则表达式方法一:boolean matches(String regex)

使用给定的正则表达式验证当前字符串是否满足格式要求

注意:无论正则表达式是否加了边界匹配符:“^”,“$”都是做全匹配验证。

案例:

String mail = "fan\_cq@tedu.cn";

/\*

\* \w+@\w+(\.[a-zA-Z]+)+

\*/

String regex = "\\w+@\\w+(\\.[a-zA-Z]+)+";

boolean match = mail.matches(regex);

if(match) {

System.out.println("是邮箱");

}else {

System.out.println("不是邮箱");

}

1. 字符串支持正则表达式方法二:String[] split(String regex)

按照满足给定的正则表达式的部分进行拆分当前字符串，将拆分后的每部分以数组形式 返回。

案例://图片重命名

String imgName = "123.jpg";

String[] name = imgName.split("\\.");

System.out.println(name.length);

imgName = System.currentTimeMillis()+"."+name[1];

System.out.println("新图片名:"+imgName);

1. 字符串支持正则表达式方法三:String replaceAll(String regex,String str)

将当前字符串中满足正则表达式的部分替换为给定字符串

案例:

String regex = "(wqnmlgb|djb|mdzz|cnm|nc|mmp|dsb)";

String message = "wqnmlgb!你这个dsb!你怎么这么nc!mmp!你个djb!";

message = message.replaceAll(regex, "\*\*\*");

System.out.println(message);//"\*\*\*你这个\*\*\*!你怎么这么\*\*\*!\*\*\*!你个 \*\*\*!";

(二)Object:java中所有的类都继承自Object,当一个类没有使用extends标明继承某个类 时，默认就继承自Object(编译器会在编译该类时补充:extends Object)

1.String toString()

该方法的意义是将当前对象转换为一个字符串， Object实现的该方法返回的字符串内容为当前对象的句柄(地址)，实际意义不大。通常我们使用一个对象的toString方法时就应当重写它，返回的字符串应当包含当前对象有效信息。JAVA的api很多情况下也会调用toString方法。

案例:

Point p = new Point();

p.setX(1);

p.setY(2);

String str = p.toString();

System.out.println(str);

System.out.println(Object o);//该方法是将给定对象的toString方法返回 的字符串输出到控制台。

2.Object提供了equals方法，意义是比较两个对象的内容是否相同。但是本身Object 提供的该方法是直接使用“==”比较的，所以若不重写该方法，没有实际意义。

案例:

Point p = new Point();

p.setX(1);

p.setY(2);

Point p1 = new Point();

p1.setX(1);

p1.setY(2);

System.out.println(p==p1);//false

System.out.println(p.equals(p1));//true

3.重写Object的toString方法。返回的字符串格式上没有要求，按照将来实际开发需 求而定。原则上应当包含当前对象的有效信息(属性值)。

案列:

public String toString() {

return "(" + x + "," + y + ")";

}

4.通常我们需要使用一个对象的equals方法就应当重写它(java提供的类不需要，已 经重写好了)。重写equals的目的是比较两个对象this和参数o的内容是否相同。 而比较也并非要求必须所有属性值都相同。可结合实际开发需求而定。

案例:

public boolean equals(Object o) {

if(o == null) {

return false;

}

if(o == this) {

return true;

}

if(o instanceof Point) {

Point p = (Point)o;

return this.x==p.x && this.y==p.y;

}

return false;

}

(三)包装类:由于java有8个基本类型，它们不具有面向对象的特性，那么在实际开发中不 能直接参与面向对象的开发,这很不方便。为此，java为8个基本类型提供了对应的包 装类。目的是可以让基本类型以对象的形式存在，能够参与到面向对象开发中。6个表 示数字类型的包装类，它们都继承自java.lang.Number。Number是一个抽象类，提供 了在6中类型之间转换数字的相关方法。

1.从基本类型变为包装类

案例:

int d = 128;

double dou = 123.123;

//Integer i1 = new Integer(d);

//Integer i2 = new Integer(d);

//推荐使用valueOf形式转换

Integer i1 = Integer.valueOf(d);

Integer i2 = Integer.valueOf(d);

Double d1 = Double.valueOf(dou);

Double d2 = Double.valueOf(dou);

System.out.println(i1==i2);//false

System.out.println(i1.equals(i2));//true

System.out.println(d1==d2);//false

System.out.println(d1.equals(d2));//true

2.包装类转为基本类型

案例:

d = i1.intValue();

byte b = i1.byteValue();//可能存在溢出问题

long l = i1.longValue();

double doub = i1.doubleValue();

System.out.println("d:"+d);//128

System.out.println("b:"+b);//-128

System.out.println("l:"+l);//128

System.out.println("doub:"+doub);//128.0

3.数字类型的包装类都提供了两个常量:MAX\_VALUE,MIN\_VALUE用来获取其表示的基本类型的取值范围。

案例:

int imax = Integer.MAX\_VALUE;

System.out.println(imax);//int类型数据最大值

int imin = Integer.MIN\_VALUE;

System.out.println(imin);//int类型数据最小值

long max = Long.MAX\_VALUE;

System.out.println(max);//long类型数据最大值

4.包装类提供了一个非常实用的静态方法:parseXXX(String str)可以将字符串转换为 对应的基本类型。包装类将字符串转换为基本类型的前提条件是该字符串要正确描 述基本类型可以保存的值。

案列:

String str = "123";

int d = Integer.parseInt(str);

System.out.println(d+1);//124

double dou = Double.parseDouble(str);

System.out.println(dou+1);//124.0

5.自JDK1.5之后，推出了一个特性:自动拆装箱该特性是编译器认可，而不是虚拟机。 编译器在编译源程序时若发现有基本类型和包装类互相赋值使用时会自动补全它们

之间的转换代码。

案例:

基本类型转换成包装类:

Integer in = 123;

//触发自动装箱特性,编译器会将代码改变为:

//Integer in = Integer.valueOf(123);

包装类转换成基本类型:

int d = new Integer(123);

//触发自动拆箱特性,编译器会将代码改变为:

//int d = new Integer(123).intValue();

(四)File类:java.io.File

File的每一个实例用于表示文件系统(硬盘中)上的一个文件或目录。

使用File可以:

1:访问属性信息(名字，长度，修改时间等)

2:操作文件或目录(创建，删除)

3:获取一个目录的子项,但是不能访问文件数据。

File没有无参构造方法，需要传入一个字符串，该字符串内容为路径。注意，路径通 常不会使用绝对路径，因为不同系统的绝对路径定义不同，会出现跨平台问题。相对路 径的好处在于不存在跨平台问题，但是相对的路径通常有运行环境而定。实际开发中比 较常使用的是类加载路径。在eclipse中，通常相对"当前目录",当前目录指的是当前 程序所在项目的项目目录。

案例1:

File file = new File("./demo.txt");

//获取名字

String name = file.getName();

System.out.println(name);

//获取该文件的长度(字节量)

long length = file.length();

System.out.println(length);

boolean cr = file.canRead();

boolean cw = file.canWrite();

System.out.println("可读:"+cr);

System.out.println("可写:"+cw);

boolean ih = file.isHidden();

System.out.println("是否隐藏:"+ih);

案例2:使用File创建一个文件(./表示当前目录，../表示上级目录)

//在当前目录下创建test.txt文件

File file = new File("./test.txt");

//boolean exists():判断当前File表示的文件或目录是否存在

if(!file.exists()) {

//将当前File表示的文件创建出来

file.createNewFile();

System.out.println("文件已创建!");

}else {

System.out.println("文件已存在!");

}

案例3:删除一个文件

//将当前目录下的test.txt文件删除,"./"不写，默认也是在当前目录中

File file = new File("test.txt");

if(file.exists()) {

file.delete();

System.out.println("文件已删除!");

}else {

System.out.println("文件不存在!");

}

案例4:创建一个目录

//在当前目录中创建一个目录:demo

File dir = new File("demo");

if(!dir.exists()) {

dir.mkdir();

System.out.println("目录已创建！");

}else {

System.out.println("目录已存在!");

}

案例5:创建一个多级目录

//在当前目录下创建:a/b/c/d/e/f

File dir = new File("a/b/c/d/e/f");

//mkdir在创建目录时要求其所在父目录必须存在

//mkdirs则会将所有不存在的父目录一同创建出来

if(!dir.exists()) {

//dir.mkdir();

dir.mkdirs();

System.out.println("创建完毕!");

}else {

System.out.println("目录已存在！");

}

案例6:删除一个目录

File dir = new File("demo");

if(dir.exists()) {

//删除目录的前提条件是，该目录必须是一个空目录！

dir.delete();

System.out.println("删除完毕!");

}else {

System.out.println("目录不存在!");

}

案例7:获取一个目录中的子项

//获取当前目录中的所有子项

File dir = new File(".");

//boolean isFile() 判断当前file是否表示的是文件

//boolean isDirectory() 判断是否表示的是目录

if(dir.isDirectory()) {

//获取当前目录中所有子项

File[] subs = dir.listFiles();

System.out.println(subs.length);

for(int i=0;i<subs.length;i++) {

System.out.println(subs[i].getName());

}

}

案例8:File提供了一个重载的listFiles方法，可以传入一个文件过滤器，然后将

表示的目录中满足过滤器要求的子项返回。

File[] listFiles(FileFilter filter)

//获取当前目录中所有文件的文件名

FileFilter filter = new FileFilter() {

public boolean accept(File file) {

return file.isFile();

}

};

File[] subs = dir.listFiles(filter);

for(int i=0;i<subs.length;i++) {

System.out.println(subs[i].getName());

}

案例9:删除给定File表示的文件或目录。

public static void main(String[] args) {

File dir = new File("a");

delete(dir);

}

//将给定的File表示的文件或目录删除

public static void delete(File file) {

//首先判断该file是文件还是目录,若是目录，

要先将所有子项删除(清空该目录),然后再将

该目录删除;若是文件，直接删除。

if(file.isDirectory()) {

//先清空该目录获取该目录所有子项

File[] subs = file.listFiles();

for(int i=0;i<subs.length;i++) {

File sub = subs[i];

//递归调用:在当前方法内部再次调用

当前方法的现象称为递归调用

delete(sub);

}

}

file.delete();

}

(五)RAF类: java.io.RandomAccessFile

RAF是专门用来读写文件数据的类，它基于指针进行读写。可对文件内容进行编辑操作。

RandomAccessFile常用构造方法:

RandomAccessFile(File file,String mode)

RandomAccessFile(String path,String mode)

第一个参数用于指定要进行读写操作的文件。

第二个参数为操作模式，常用:

r:只读模式，仅对文件进行读取操作

rw:读写模式，对文件数据可读也可写

案列1:

RandomAccessFile raf = new RandomAccessFile("./raf.dat","rw");

/\*

\* void write(int d)

\* 向文件中写入一个字节，将给定的int值所对应的

\* 2进制的低八位写入文件

\* vvvvvvvv

\* 00000000 00000000 00000000 00000001

\*/

raf.write(1);

System.out.println("写出完毕!");

raf.close();

案例2: 复制文件

/\*

\* 创建一个RAF读取原文件

\* 创建一个RAF向复制文件中写

\* 顺序的从原文件读取每个字节写入到复制的文件中

\*/

RandomAccessFile src = new RandomAccessFile("music.mp3","r");

RandomAccessFile desc= new RandomAccessFile("music\_cp.mp3","rw");

//用来保存每次从原文件读取到的字节

int d = -1;

long start = System.currentTimeMillis();

while((d = src.read()) != -1) {

desc.write(d);

}

long end = System.currentTimeMillis();

System.out.println("复制完毕!耗时:"+(end-start)+"ms");

src.close();

desc.close();

案例3: 读取一个字节

RandomAccessFile raf = new RandomAccessFile("raf.dat","r");

/\*

\* int read()

\* 从文件当前指针位置读取1个字节，读取后以int

\* 形式返回。若读取到文件末尾，则返回值为-1。

\* 00000000 00000000 00000000 00000001 \*/

int d = raf.read();

System.out.println(d);

//试图读取第二个字节

d = raf.read();

System.out.println(d);//-1

raf.close();

案列4：RAF提供了读写基本类型数据的相关方法。以及操作指针的相关方法，RAF是 基于指针进行读写数据的，即：RAF总是在指针指向的文件字节位置进行读或 写并且读写后指针会自动向后移动到下一个字节准备读写。

RandomAccessFile raf= new RandomAccessFile("raf.dat","rw");

//获取当前RAF的指针位置

long pos = raf.getFilePointer();

System.out.println("pos:"+pos);//0

/\*

\* d vvvvvvvv

\* 01111111 11111111 11111111 11111111

\*

\* 位操作符

\* >>> 将2进制整体向右移动指定位数

\* d>>>24 vvvvvvvv

\* 00000000 00000000 00000000 01111111 11111111 11111111 11111111

\*

\* d>>>16 vvvvvvvv

\* 00000000 00000000 01111111 11111111 11111111 11111111

\*/

int d = Integer.MAX\_VALUE;

raf.write(d>>>24);

System.out.println("pos:"+raf.getFilePointer());//1

raf.write(d>>>16);//2

raf.write(d>>>8);//3

raf.write(d);//4

System.out.println("pos:"+raf.getFilePointer());//4

/\*

\* RAF提供了方便直接写出基本类型数据的相关方法:

\*

\* void writeInt(int d)

\* 连续写出4个字节，将该int值写出

\*/

raf.writeInt(d);

System.out.println("pos:"+raf.getFilePointer());//8

//一次性写出8字节，将long值写出

raf.writeLong(123L);

System.out.println("pos:"+raf.getFilePointer());//16

raf.writeDouble(123.123);

System.out.println("pos:"+raf.getFilePointer());//24

System.out.println("写出完毕!");

/\*

\* void seek(long pos)

\* 将指针移动到指定位置

\* 移动到第一个字节的位置

\*/

raf.seek(0);

System.out.println("pos:"+raf.getFilePointer());//0

/\*

\* int readInt()

\* 连续读取4个字节，还原该int值

\* 若读取过程中发现到了文件末尾，则抛出:

\* EOFException

\* EOF end of file

\*/

d = raf.readInt();

System.out.println(d);//2147483647(int的最大值)

System.out.println("pos:"+raf.getFilePointer());//4

raf.seek(8);

long l = raf.readLong();

System.out.println(l);//123

System.out.println("pos:"+raf.getFilePointer());//16

double dou = raf.readDouble();

System.out.println(dou);//123.123

System.out.println("pos:"+raf.getFilePointer());//24

d = raf.read();

System.out.println(d);//-1

raf.close();

案列5：通过提高每次读写的数据量，减少实际读写的次数，可以提高读写效率。

读写有两种:

随机读写:单字节读写

块读写:一组一组字节读写

RandomAccessFile src = new RandomAccessFile("mv.wmv","r");

RandomAccessFile desc= new RandomAccessFile("mv\_cp.wmv","rw");

/\*

\* RAF提供的块读写操作的方法

\* 块读：

\* int read(byte[] data)

\* 通过当前RAF一次性读取给定的字节数组总长度的

\* 字节量，并装入到该数组中。返回值为实际读取到

\* 的字节量。若返回值为-1，则表示本次没有读取到

\* 任何数据(已经是文件末尾了)

\* 块写:

\* void write(byte[] data)

\* 通过RAF将给定字节数组所有字节一次性写出

\*

\* void write(byte[] data,int offset,int len)

\* 将给定的字节数组从下标offset处的连续len个字节

\* 一次性写出

\*/

//记录每次实际读取到的字节量

int len = -1;

//10k

byte[] data = new byte[1024\*10];

long start = System.currentTimeMillis();

while((len = src.read(data)) != -1) {

desc.write(data,0,len);

}

long end = System.currentTimeMillis();

System.out.println("复制完毕!耗时:"+(end-start)+"ms");

src.close();

desc.close();

案列6：读取字符串

RandomAccessFile raf= new RandomAccessFile("raf.txt","r");

byte[] data = new byte[(int)raf.length()];

raf.read(data);

//将给定的字节数组中所有字节按照指定字符集还原为字符串

String str = new String(data,"UTF-8");

System.out.println(str);

raf.close();

案例7：向文件中写入字符串

RandomAccessFile raf

= new RandomAccessFile("raf.txt","rw");

String str = "像一颗海草海草海草海草，随波飘摇。";

/\*

\* 字符串提供了转换为字节的方法:

\* byte[] getBytes()

\* 将当前字符串按照系统默认字符集转换为一组字节

\* byte[] getBytes(String csn)

\* 使用指定的字符集将字符串转换为对应的一组字节

\* 常见字符集:

\* GBK:国标编码，英文1字节，中文2字节

\* UTF-8:unicode的一种字符集，也成为万国码，包含

\* 世界上流行的语言对应的字符。

\* 英文1字节，中文3字节。

\* ISO8859-1:欧洲的一种字符集，不支持中文。

\*/

// byte[] data = str.getBytes();

byte[] data = str.getBytes("UTF-8");

raf.write(data);

raf.write("海草海草海草海草，浪花里舞蹈.".getBytes("UTF-8"));

System.out.println("写出完毕!");

raf.close();

(六)IO

案列1：

/\* java标准IO

\* IO指的是输入与输出(Input,Output)

\* java将读写操作按照方向划分为输入流与输出流，其中输入

\* 流用来读，输出流用来写。

\* java.io.InputStream:所有字节输入流的超类，定义了若干

\* 抽象方法，其中规定了读取字节的相关操作。

\*

\* java.io.OutputStream:所有字节输出流的超类。定义了写出

\* 字节的相关方法。

\*

\* 流有两类:

\* 节点流和处理流

\*

\* 节点流:又称为低级流。是实际链接程序与数据源的"管道"，

\* 负责实际数据的搬运工作。读写数据一定是建立在

\* 节点流的基础上进行的。特点:数据源明确。

\*

\* 处理流:又称为高级流。不能独立存在，必须链接在其他流

\* 上，作用是当数据流经该流时，其可以对数据进行

\* 某些加工处理。这样可以简化我们再对数据加工的

\* 操作。

\*

\* 流的链接:将需要的一组处理流链接在一起，最终链接在节点

\* 流上，可对数据进行"流水线"式的加工并进行读写

\* 这也是IO读写的重点。

\*

\*

\* 文件流,文件流是一组节点流，作用是读写文件。

\*

\* 文件流与RAF都是用来读写文件数据的，但是他们也有各自的

\* 优点与缺点。

\* 文件流:由于流的读写模式是顺序读写，所以文件流也是以该

\* 方式读写文件数据的。所以做不到编辑文件数据(只覆盖文件

\* 特定位置的字节)。读写不能回退操作。但是基于流链接操作

\* 可以轻松读写复杂数据。

\*

\* RAF:基于指针进行读写，所以RAF可以操作指针读写特定位置

\* 的字节，能够编辑文件数据。虽然有提供方便的读写基本类型

\* 数据与字符串等方法，但是读写复杂数据需要自行完成。

\*/

/\*

\* 文件输出流有两种创建方式:

\* FileOutputStream(File file)

\* FileOutputStream(String path)

\* 以上构造方法创建的文件流是覆盖写操作，即：若

\* 该文件已经存在，会先将该文件数据清除，然后通

\* 过这个流写出的内容作为文件数据保存。

\*

\* FileOutputStream(File file,boolean append)

\* FileOutputStream(String path,boolean append)

\* 若第二个参数为true，则是追加写模式，即:若该

\* 文件存在，保留该文件所有数据，将通过当前流写出

\* 的数据追加到文件末尾。

\*

\* 做不到像RAF那样操作指针，覆盖部分内容的操作。

\*

\*/

FileOutputStream fos = new FileOutputStream("fos.txt",true);

String str = "我本是王母娘娘身边的一员天庭大将。";

byte[] data = str.getBytes("UTF-8");

fos.write(data);

System.out.println("写出完毕!");

fos.close();

案例2：文件输入流

FileInputStream fis= new FileInputStream("fos.txt");

byte[] data = new byte[100];

int len = fis.read(data);

System.out.println("实际读取了"+len+"个字节");

String str = new String(data,0,len,"UTF-8");

System.out.println(str);

fis.close();

案列3：使用文件流复制文件

/\*

\* 1:创建文件输入流读取原文件

\* 2:创建文件输出流向复制文件写

\* 3:顺序从原文件读取字节写入复制文件中

\* 完成复制操作。

\*/

FileInputStream fis = new FileInputStream("mv.wmv");

FileOutputStream fos= new FileOutputStream("mv\_cp3.wmv");

int len = -1;

byte[] data = new byte[1024\*10];

while((len = fis.read(data))!=-1) {

fos.write(data,0,len);

}

System.out.println("复制完毕!");

fis.close();

fos.close();

案列4：缓冲流

java.io.BufferedOutputStream

java.io.BufferedInputStream

缓冲流是一对高级流，可以加快读写数据的效率。无论我们通过缓冲流进 行的是随机读写还是块读写，最终都会被缓冲流转换为块读写操作。

FileInputStream fis= new FileInputStream("music.mp3");

BufferedInputStream bis= new BufferedInputStream(fis);

FileOutputStream fos= new FileOutputStream("music\_cp3.mp3");

BufferedOutputStream bos= new BufferedOutputStream(fos);

int d = -1;

while((d = bis.read()) != -1) {

bos.write(d);

}

System.out.println("复制完毕!");

bis.close();

bos.close();

案例5：缓冲输出流的缓冲区问题

FileOutputStream fos= new FileOutputStream("bos.txt");

BufferedOutputStream bos = new BufferedOutputStream(fos);

String str = "师傅，我坚持不住了!";

byte[] data = str.getBytes("UTF-8");

bos.write(data);

/\*

\* void flush()

\* 缓冲流提供了强制清空缓冲区的操作，调用flush

\* 会将当前缓冲区已经缓冲的字节一次性写出。

\*/

bos.flush();

System.out.println("写出完毕!");

bos.close();

案列6：使用当前类的实例测试对象流的读写操作

当一个类的实例希望被对象流读写，那么该类必须实现:

java.io.Serializable接口，否则在序列化时会抛出异常

当我们实现Serializable接口后，该类被编译器编译时会多出一个方法， 作用是将当前类实例转换为一组字节。但是源码中不体现.由于不实现该 接口，编译器不会给类添加这个方法，这就是为什么对象流在序列化时要 求必须实现该接口的原因。否则无法将对象转换为一组字节。

public class Person implements Serializable{

/\*\*

\* 当一个类实现了Serializable接口后，应当定义一个

\* 常量:serialVersionUID 序列化版本号

\* 序列化版本号直接影响反序列化对象的结果是否成功

\*

\* 当对象输入流反序列化一个对象时，会检测该对象的

\* 版本号是否与当前类版本号一直，若不一致则直接抛出

\* 反序列化失败异常。若版本号一直，则采用兼容模式，

\* 即:若反序列化的对象与当前类结构不完全一致时，所有

\* 可以还原的属性都还原。若结构一致则直接还原。

\*

\* 若我们没有定义序列化版本号，则编译器在编译当前

\* 类时会根据结构生成一个版本号，弊端是只要当前类的

\* 结构发生改变，版本号一定改变。

\*

\*/

private static final long serialVersionUID = 1L;

private String name;

private int age;

private String gender;

/\*

\* 当一个属性被transient修饰后，那么当前类实例在

\* 序列化时，该属性值会被忽略。

\* 忽略不需要保存的属性可以达到对象瘦身的功能。

\*/

private transient String[] otherInfo;

}

案例7：对象流

java.io.ObjectOutputStream

java.io.ObjectInputStream

对象流是一对高级流，可以方便的读写java中任何对象。

String name = "苍老师";

int age = 18;

String gender = "女";

String[] otherInfo = {"有容乃大","德艺双馨","是一名演员","促进中 日文化交流","广大男性同胞的启蒙老师"};

Person p = new Person(name,age,gender,otherInfo);

System.out.println(p);

FileOutputStream fos= new FileOutputStream("person.obj");

ObjectOutputStream oos = new ObjectOutputStream(fos);

/\*

\* 对象输出流提供了写出对象的方法：

\* void writeObject(Object obj)

\* 将给定的对象转换为一组字节后写出

\*

\* 这里通过oos将给定对象写入文件，实际经历了

\* 两个操作:

\* 1:oos将给定的对象转换为了一组字节，这个过程称为:对象序列化

\* 2:fos将这组字节写入到文件中(硬盘上)，这个过程称为:数据持久化

\*/

oos.writeObject(p);

System.out.println("写出完毕!");

oos.close();

案列8：对象反序列化

FileInputStream fis = new FileInputStream("person.obj");

ObjectInputStream ois = new ObjectInputStream(fis);

/\*

\* 对象输入流提供了对象反序列化方法:

\* Object readObject()

\* 需要注意，该方法在读取字节时，必须保证读取到

\* 的字节是对象输出流将一个对象序列化的字节，否

\* 则会抛出ClassNotFoundException

\*/

Person p = (Person)ois.readObject();

System.out.println(p);

ois.close();

案例9：字符流

java按照读写单位将流划分为:字节流与字符流

字符流的读写单位是以字符为单位的。但底层实际还是要转换为字节进行 写操作，或者将读取到的字节转换为字符。本质还是读写字节。

转换流:

java.io.OutputStreamWriter

java.io.InputStreamReader

它们是字符流的一对常用实现类，将来实际开发中很少直接使用它们，但 它们在流链接中是重要的一环。

FileOutputStream fos= new FileOutputStream("osw.txt");

OutputStreamWriter osw = new OutputStreamWriter(fos,"utf-8");

osw.write("摩羯一生，只有两件事，关你屁事和关我屁事。");

System.out.println("写出完毕!");

osw.close();

案例9：转换流 java.io.InputStreamReader读取字符

FileInputStream fis = new FileInputStream("osw.txt");

InputStreamReader isr = new InputStreamReader(fis,"UTF-8");

int d = -1;

while((d = isr.read())!=-1) {

char c = (char)d;

System.out.print(c);

}

isr.close();

案例10：缓冲字符流

java.io.BufferedWriter

java.io.BufferedReader

缓冲字符流可以块读写，并且特点是可以按行读写字符串

java.io.PrintWriter

具有自动行刷新的缓冲字符输出流。内部连接BufferedWriter作为缓冲 功能。

/\*

\* 向文件中写出字符串

\*/

// PrintWriter pw = new PrintWriter("pw.txt");

PrintWriter pw = new PrintWriter("pw.txt","UTF-8");

pw.println("摩羯一生只有两件事，");

pw.println("管你屁事和管我屁事。");

pw.println("得到摩羯就能得天下。");

System.out.println("写出完毕!");

pw.close();

案例11：流连接中使用PrintWriter

FileOutputStream fos= new FileOutputStream("pw.txt");

OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");

BufferedWriter bw = new BufferedWriter(osw);

PrintWriter pw = new PrintWriter(bw);

pw.println("我祈祷拥有一颗透明的心灵.");

System.out.println("写出完毕!");

pw.close();

案例12：在创建PrintWriter时，若构造方法第一个参数为流(字节流字符流均可)， 那么该构造方法就支持一个重载，可以再传入第二个参数，该参数为 boolean型，若值为true时，则具有了自动行刷新功能。即：每当调用 println方法(注意，不是调用print方法！！！)后就会自动刷新，将该字 符串实际写出。但是该功能会降低写效率(因为提高了写次数)，但是若考 虑到即时性时，该方法很易用。

Scanner scanner = new Scanner(System.in);

System.out.println("请输入文件名:");

String fileName = scanner.nextLine();

FileOutputStream fos = new FileOutputStream(fileName);

OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");

BufferedWriter bw = new BufferedWriter(osw);

PrintWriter pw = new PrintWriter(bw,true);//自动行刷新

System.out.println("请开始输入内容:");

String line = null;

while(true) {

line = scanner.nextLine();

if("exit".equals(line)) {

break;

}

pw.print(line);

// pw.flush();

}

System.out.println("再见!");

pw.close();

案例13：java.io.BufferedReader 缓冲字符输入流，可以按行读取字符串

//将当前源代码读取出来并输出到控制台

FileInputStream fis

= new FileInputStream("src/io/BufferedReaderDemo.java");

InputStreamReader isr = new InputStreamReader(fis);

BufferedReader br = new BufferedReader(isr);

/\*

\* BufferedReader提供方法：

\* String readLine()

\* 顺序读取若干字符，直到读取了换行符为止，然后

\* 将换行符之前的所有字符以一个字符串形式返回。

\* 注意，返回的字符串中是不含有最后的换行符的。

\* 若返回值为null，表示末尾。

\*/

String line = null;

while((line = br.readLine())!=null) {

System.out.println(line);

}

br.close();

(七)Exception

案例1：

/\*\*

\* java异常处理机制中的try-catch

\* 语法:

\* try{

\* 代码片段

\* }catch(XXException e){

\* 处理XXException的代码片段

\* }

\* 或

\* try{

\* ...

\* }finally{

\* ...

\* }

\*/

System.out.println("程序开始了");

try {

String str = "a";

System.out.println(str.length());

System.out.println(str.charAt(0));

System.out.println(Integer.parseInt(str));

/\*

\* 在try语句块中出错的代码以下的内容都不会

\* 被执行

\*/

System.out.println("!!!!!!");

/\*

\* 如果try语句块中的代码没有出现异常，那么

\* catch语句块是不执行的

\*/

}catch(NullPointerException e) {

System.out.println("出现了空指针!");

}catch(StringIndexOutOfBoundsException e) {

System.out.println("字符串下标越界了!");

/\*

\* 应当养成一个好习惯，在最后一个catch上捕获

\* Exception,避免因为try代码块中一个未捕获的

\* 异常导致程序中断。

\*/

}catch(Exception e) {

System.out.println("反正就是出了个错!");

}

System.out.println("程序结束了");

案列2：finally块

finally块只能定义在异常处理机制的最后。即：直接跟在try之后或者 最后一个catch之后。finally块可以保证只要程序执行到try语句块中， 那么finally块中的代码必定执行。所以通常将无关乎程序是否出异常都 要执行的代码放在这里，例如IO操作完毕后的流关闭。

System.out.println("程序开始了");

try {

String str = "";

System.out.println(str.length());

return;

}catch(Exception e) {

System.out.println("出错了!");

}finally {

System.out.println("finally中的代码运行了!");

}

System.out.println("程序结束了");

案例3：finally在IO中的应用

FileOutputStream fos = null;

try {

fos = new FileOutputStream("fos.dat");

fos.write(1);

} catch (Exception e) {

System.out.println("出错了！");

} finally {

try {

if(fos!=null) {

fos.close();

}

} catch (IOException e) {

}

}

案例4：在JDK7之后，推出了一个新的特性:自动关闭。

try(

/\*

\* 实现了AutoCloseable接口的定义在这里，

\* 最终可以被自动关闭

\* 实际上下面的代码会被编译器在编译时修改

\* 为类似:FinallyDemo2代码的内容

\*/

FileOutputStream fos

= new FileOutputStream("fos.dat");

) {

fos.write(1);

}catch(Exception e) {

System.out.println("出错了！");

}

案例5：finally常见面试题

请简述:final,finally,finalize

finalize是Object定义的方法。当GC要释放一个对象时会调用该方法。 该方法执行完毕意味着其即将被释放。该方法不应当有耗时操作。

public static void main(String[] args) {

System.out.println(test("0")+","+test(null)+","+test(""));

// 3,3,3

}

public static int test(String str) {

try {

System.out.println("执行test,参数为:"+str);

return str.charAt(0)-'0';

} catch (NullPointerException e) {

System.out.println("空指针了，return:1");

return 1;

} catch (Exception e) {

System.out.println("出错了,return:2");

return 2;

} finally {

System.out.println("finally了，retrun:3");

return 3;

}

}

案例6：使用当前类测试异常的抛出相关知识。

public class Person {

private int age;

public int getAge() {

return age;

}

/\*\*

\* 使用throw在方法中抛出一个异常通常在一下情况发生:

\* 1:符合语法，但是不符合业务逻辑时，可以当成异常抛出

\* 给调用者。

\* 2:确实发生了一个异常，但是该异常不应当被当前方法

\* 解决，这时可以抛出给调用者。

\* @param age

\*/

public void setAge(int age) {

if(age<0||age>100) {

/\*

\* 通常方法中使用throw什么异常，就要在方法

\* 声明的同时使用throws声明该异常的抛出。

\* 只有RuntimeException及其子类异常不需要。

\*/

throw new RuntimeException("年龄不合法!");

}

this.age = age;

}

}

案例7：throws是用来在方法上声明该方法可能会抛出的异常，要求调用者必须处 理该异常。子类在重写父类含有throws声明异常抛出的方法时对throws 的重写规则如下

public class ThrowsDemo {

public void dosome() throws IOException,AWTException {

}

}

class Son extends ThrowsDemo{

// public void dosome()

// throws IOException,AWTException {

//

// }

//仅抛出部分异常

// public void dosome()

// throws IOException{

//

// }

//不再抛出任何异常

// public void dosome(){

//

// }

//可抛出父类方法抛出异常的子类型异常

// public void dosome()

// throws FileNotFoundException {

//

// }

//不允许抛出额外异常

// public void dosome()

// throws SQLException {

//

// }

//不允许抛出父类方法抛出异常的父类型异常

// public void dosome()

// throws Exception{

//

// }

}

案例8：异常常用方法

System.out.println("程序开始了");

try {

String str = "a";

int a = Integer.parseInt(str);

} catch (Exception e) {

System.out.println("出错了！");

//输出错误堆栈信息，有助于调整错误

// e.printStackTrace();

//获取错误消息，但不是所有异常都有。

String message = e.getMessage();

System.out.println(message);

}

System.out.println("程序结束了");

案例9：自定义异常通常用来说明如业务逻辑错误的情况。

年龄不合法

public class IllegalAgeException extends Exception{

private static final long serialVersionUID = 1L;

public IllegalAgeException() {

super();

// TODO Auto-generated constructor stub

}

public IllegalAgeException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {

super(message, cause, enableSuppression, writableStackTrace);

// TODO Auto-generated constructor stub

}

public IllegalAgeException(String message, Throwable cause) {

super(message, cause);

// TODO Auto-generated constructor stub

}

public IllegalAgeException(String message) {

super(message);

// TODO Auto-generated constructor stub

}

public IllegalAgeException(Throwable cause) {

super(cause);

// TODO Auto-generated constructor stub

}

}

案例10：异常的抛出

Person p = new Person();

try {

/\*

\* 当调用一个含有throws声明异常抛出的方法时

\* 必须处理该异常，否则编译不通过。

\* 处理异常的方式有两种:

\* 1:使用try-catch捕获并处理该异常

\* 2:在当前方法上继续使用throws声明该异常

\* 的抛出。

\*/

p.setAge(999);

} catch (IllegalAgeException e) {

e.printStackTrace();

}

System.out.println("年龄:"+p.getAge());

}

(八)Socket：套接字

Socket封装了TCP协议的通讯细节，为我们提供了简洁的实现过程，通过操 作两个流来完成与远端计算机之间的数据传输。

//聊天室客户端

public class Client {

private Socket socket;

/\*\*

\* 构造方法，用来初始化客户端

\*/

public Client() {

try {

/\*

\* 实例化Socket时需要传入两个参数，并且实例

\* 化的过程就是与远端计算机建立连接的过程。

\* 若远端计算机没有响应，会抛出异常。

\* 参数1:远端计算机地址信息(IP)

\* 参数2:远端计算机开启的端口

\*

\*/

System.out.println("正在连接服务端...");

socket = new Socket("localhost",8088);

System.out.println("与服务端建立连接!");

} catch (UnknownHostException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 客户端开始工作的方法

\*/

public void start() {

try {

/\*

\* Socket提供方法:

\* OutputStream getOutputStream()

\* 该方法获取一个字节输出流，通过该输出流写

\* 出的字节会通过网络发送给远端计算机。

\*/

OutputStream out = socket.getOutputStream();

OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");

BufferedWriter bw = new BufferedWriter(osw);

PrintWriter pw = new PrintWriter(bw,true);

//写出即发送给服务端。

pw.println("你好服务端!");

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

//实例化

Client client = new Client();

//启动客户端

client.start();

}

}

//聊天室服务端

public class Server {

/\*

\* java.net.ServerSocket

\* 运行在服务端的ServerSocket主要有两个作用

\* 1:向服务端所在系统申请一个服务端口

\* 客户端就是通过这个端口与服务端建立连接的

\* 2:监听该服务端口，一旦一个客户端通过该端口与

\* 服务端建立连接，那么ServerSocket就会自动的

\* 创建一个Socket与客户端进行通讯。

\*/

private ServerSocket server;

public Server() {

try {

/\*

\* 实例化ServerSocket的同时要指定向系统

\* 申请的服务端口。注意，该端口不能与当前

\* 系统其它应用程序申请的端口号一致，否则

\* 会抛出端口被占用的异常。

\*/

System.out.println("正在启动服务端...");

server = new ServerSocket(8088);

System.out.println("服务端启动完毕!");

} catch (IOException e) {

e.printStackTrace();

}

public void start() {

try {

/\*

\* ServerSocket提供的方法:

\* Socket accept()

\* 该方法是一个阻塞方法，调用后服务端开始

\* 监听服务端口，一旦一个客户端通过该端口

\* 请求连接，就会创建并返回一个Socket实例

\* 通过这个Socket就可以与刚建立连接的这个

\* 客户端进行通讯了。

\*/

System.out.println("等待客户端连接...");

Socket socket = server.accept();

System.out.println("一个客户端连接了!");

InputStream in = socket.getInputStream();

InputStreamReader isr = new InputStreamReader(in,"UTF-8");

BufferedReader br

= new BufferedReader(isr);

String message = br.readLine();

System.out.println("客户端说:"+message);

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

Server server = new Server();

server.start();

}

}

(九)Thread:多线程

多线程可以实现多段代码"同时运行"的效果。实际上是并发运行的。

创建线程有两种方式：方式1:继承Thread并重写run方法。run方法的作用 是定义该线程要执行的任务。

public class ThreadDemo1 {

public static void main(String[] args) {

Thread t1 = new MyThread1();

Thread t2 = new MyThread2();

/\*

\* 启动线程要调用start方法，而不是直接调用run方法

\* 当一个线程的start方法调用完毕后，该线程会纳入到

\* 线程调度中，一旦被分配CPU时间片，该线程会自动的

\* 执行其run方法开始运行任务。所以一个线程的start

\* 方法执行后run方法会很快的被运行起来。

\*/

t1.start();

t2.start();

System.out.println("两个线程运行了！");

}

}

/\*\*

\* 第一种创建线程的方式比较简单，适合使用匿名内部类快速

\* 创建临时线程跑任务。

\* 但是它也存在两种设计不足

\* 1:由于需要继承Thread，而java又是单继承的，这就导致若

\* 继承了Thread就无法再继承其他类去复用方法。这在实际

\* 开发中是出现比较多的矛盾。

\* 2:继承Thread后需要重写run方法来定义任务，这就导致了

\* 线程与线程要执行的任务之间存在一个必然的耦合关系，

\* 导致线程重用性变差。

\*/

class MyThread1 extends Thread{

public void run() {

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

System.out.println("你是谁啊？");

}

}

}

class MyThread2 extends Thread{

public void run() {

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

System.out.println("我是查水表的!");

}

}

}

第二种创建线程的方式：实现Runnable接口，单独定义线程任务

public class ThreadDemo2 {

public static void main(String[] args) {

//实例化任务

Runnable r1 = new MyRunnable1();

Runnable r2 = new MyRunnable2();

//实例化线程

Thread t1 = new Thread(r1);

Thread t2 = new Thread(r2);

t1.start();

t2.start();

}

}

class MyRunnable1 implements Runnable{

public void run() {

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

System.out.println("你是谁啊？");

}

}

}

class MyRunnable2 implements Runnable{

public void run() {

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

System.out.println("我是查水表的!");

}

}

}

案例1：使用匿名内部类创建形式完成两种线程的创建方式

public class ThreadDemo3 {

public static void main(String[] args) {

//方式1

Thread t1 = new Thread() {

public void run() {

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

System.out.println("你是谁啊？");

}

}

};

//方式2

Runnable r2 = new Runnable() {

public void run() {

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

System.out.println("我是查水表的!");

}

}

};

Thread t2 = new Thread(r2);

t1.start();

t2.start();

}

}

案例2：Thread提供了一个静态方法:

static Thread currentThread()

该方法可以获取到运行该方法的线程并将其返回。

后面会有一个很重要的API：ThreadLocal，就是利用这个方法解决问题 的。包括后期spring中使用aop管理事务时就用到了这个操作。

public static void main(String[] args) {

//获取运行main方法的线程

Thread main = Thread.currentThread();

System.out.println("运行main方法的线程是:"+main);

dosome();

Thread t = new Thread() {

public void run() {

Thread t = Thread.currentThread();

System.out.println("自定义线程:"+t);

dosome();

}

};

t.start();

}

public static void dosome() {

//获取运行dosome方法的线程

Thread t = Thread.currentThread();

System.out.println("获取运行dosome方法的线程是:"+t);

}

案例3：获取线程相关信息的一组方法

public static void main(String[] args) {

//获取main线程

Thread main = Thread.currentThread();

//获取线程名字

String name = main.getName();

System.out.println(name);

//获取唯一标识

long id = main.getId();

System.out.println(id);

//获取线程优先级

int priority = main.getPriority();

System.out.println("优先级:"+priority);

//是否处于活动状态

boolean isAlive = main.isAlive();

//判断线程是否为守护线程

boolean isDaemon = main.isDaemon();

//线程是否被中断

boolean isInterrupted = main.isInterrupted();

System.out.println("isAlive:"+isAlive);

System.out.println("isDaemon:"+isDaemon);

System.out.println("isInterrupted:"+isInterrupted);

}

案列4：线程的优先级

线程在并发运行时对于线程调度的工作是不可控的，即:不能主动获取 CPU时间片，只能被动的被分配。但是线程调度会尽可能均匀的将时间片 分配给所有并发运行的线程，但不能保证每个线程"一人一次"。通过调整 线程的优先级可以最大程度的改善获取CPU时间片的次数。理论上，线程 优先级越高的线程获取CPU时间片的次数越多。

public static void main(String[] args) {

Thread max = new Thread() {

public void run() {

for(int i=0;i<10000;i++) {

System.out.println("max");

}

}

};

Thread norm = new Thread() {

public void run() {

for(int i=0;i<10000;i++) {

System.out.println("nor");

}

}

};

Thread min = new Thread() {

public void run() {

for(int i=0;i<10000;i++) {

System.out.println("min");

}

}

};

/\*

\* 线程优先级有10个等级，分别用整数1-10表示

\* 其中1为最低，10为最高，5为默认

\*/

min.setPriority(Thread.MIN\_PRIORITY);

max.setPriority(Thread.MAX\_PRIORITY);

min.start();

norm.start();

max.start();

}

案例5：static void sleep(long ms)

线程提供的sleep方法可以让运行该方法的线程处于阻塞状态指定毫秒， 当超时后，线程会自动回到RUNNABLE状态等待再次分配时间片并发运行。 当一个线程处于BLOCK(阻塞)状态时，CPU会立刻释放其时间片，去执行 其他线程。

public static void main(String[] args) {

System.out.println("程序开始了!");

try {

for(int i=100;i>0;i--) {

System.out.println(i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("程序结束了!");

}

案例6：sleep方法要求必须处理中断异常:InterruptedException

当我们调用一个线程的interrupted方法时，可以中断一个正在运行的线 程，但是若该线程正处于阻塞时，那么调用该方法时并不是中断开线程， 而仅是中断其阻塞状态，这时会抛出中断异常，通知程序该线程的阻塞状 态被中断。

public static void main(String[] args) {

Thread lin = new Thread() {

public void run() {

System.out.println("林:刚美完容，睡一觉吧!");

try {

/\*

\* 在阻塞过程中当前线程被中断，那么

\* 会中断阻塞状态并抛出异常

\*/

Thread.sleep(10000000);

} catch (InterruptedException e) {

System.out.println("林:干嘛呢!干嘛呢!干嘛呢! 都破了相了!");

}

System.out.println("林:醒了!");

}

};

Thread huang = new Thread() {

public void run() {

System.out.println("黄:开始砸墙！");

for(int i=0;i<5;i++) {

System.out.println("黄:80!");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

}

}

System.out.println("咣当!");

System.out.println("黄:搞定!");

/\* 由于lin线程处于阻塞，这边中断的话,lin的sleep

\* 方法会抛出中断异常 \*/

lin.interrupt();

}

};

lin.start();

huang.start();

}

案例7：守护线程

守护线程又称为后台线程。默认创建出来的线程都是前台线程，后台线程 是需要在线程启动前单独进行设置的。设置的方法为:

void setDaemon(boolean on)

当参数为true时，该线程被设置为守护线程(后台线程)

在使用上前台线程与后台线程没有什么区别，但是在结束时机上有一个不 同，即:当进程结束时，所有还在运行的守护线程都会被强制中断。

进程的结束:当一个进程中的所有前台线程都结束时，进程结束

public static void main(String[] args) {

Thread rose = new Thread() {

public void run() {

for(int i=0;i<5;i++) {

System.out.println("rose:let me go!");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

}

}

System.out.println("rose:啊啊啊啊啊 AAAAAAAaaaaaa.....");

System.out.println("噗通");

}

};

Thread jack = new Thread() {

public void run() {

while(true) {

System.out.println("jack:you jump!i jump!");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

}

}

}

};

rose.start();

jack.setDaemon(true);//设置为守护线程

jack.start();

}

案例8：多个线程之间是异步运行代码的(各自代码运行之间不存在先后关系，各 干各的)

同步运行:执行有先后顺序，通常单线程运行代码是同步的

异步运行:各干各的，多线程执行就是异步执行

线程提供了:

void join()

该方法允许一个线程在join方法所属线程后面等待，直到该线程执行完 毕后再继续运行。所以使用join可以协调线程之间同步运行代码。

public class JoinDemo {

//表示图片是否下载完毕

public static boolean isFinish = false;

public static void main(String[] args) {

final Thread download = new Thread() {

public void run() {

System.out.println("down:开始下载图片...");

for(int i=0;i<=100;i++) {

System.out.println("down:"+i+"%");

try {

Thread.sleep(50);

} catch (InterruptedException e) {

}

}

System.out.println("down:图片下载完毕!");

isFinish = true;

}

};

Thread show = new Thread() {

public void run() {

System.out.println("show:开始显示图片...");

/\*

\* 加载图片前应当等待下载线程将图片下载完毕

\*/

try {

/\*

\* JDK1.8之前有一个要求(源自JVM内存分配

\* 问题):当一个方法的局部内部类当中想引用

\* 该方法的其他局部变量时，该变量必须是 \* final的。

\* 这里main方法的局部内部类show当中想引用

\* download,而download本身是main方法的一个

\* 局部变量，那么该变量就必须是final的

\*/

download.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

if(!isFinish) {

//模拟加载图片失败

/\*

\* 当一个线程的run方法抛出了一个异常，

\* 意味着该线程就会结束

\*/

throw new RuntimeException("图片加载失败!");

}

System.out.println("show:显示图片完毕!");

}

};

download.start();

show.start();

}

}

案例9：多线程并发安全问题

当多个线程并发操作同一资源时，由于线程切换实际不确定导致程序未按照设 计要求的顺序执行而出现了逻辑混乱。严重时可能导致系统瘫痪。

解决多线程并发安全问题的方式就是让多个线程排队访问该资源。将异步操作 变为同步操作。

public class SyncDemo {

public static void main(String[] args) {

final Table table = new Table();

Thread t1 = new Thread() {

public void run() {

while(true) {

int bean = table.getBean();

Thread.yield();//模拟线程发生切换

System.out.println(getName()+":"+bean);

}

}

};

Thread t2 = new Thread() {

public void run() {

while(true) {

int bean = table.getBean();

Thread.yield();//模拟线程发生切换

System.out.println(getName()+":"+bean);

}

}

};

t1.start();

t2.start();

}

}

class Table{

//桌子上有20个豆子

private int beans = 20;

/\*\*

\* 当一个方法被synchronized修饰后，那么该方法称为

\* 同步方法，即：多个线程不能同时进入方法内部执行，

\* 只能一个线程执行完，其他线程再进来执行。

\* 将异步操作改为同步操作就解决了并发安全问题。

\*

\* 成员方法上使用synchronized，那么同步监视器

\* 对象就是当前方法所属对象，即方法中看到的this

\*

\*/

public synchronized int getBean() {

if(beans==0) {

throw new RuntimeException("没有豆子了");

}

Thread.yield();//模拟线程发生切换

return beans--;

}

}

案例10：有效的缩小同步范围可以在保证并发安全的前提下提高并发的效率。

同步块可以更准确的控制需要同步运行的代码片段,语法:

synchronized(同步监视器对象){

需要同步运行的代码片段

}

public class SyncDemo2 {

public static void main(String[] args) {

final Shop shop = new Shop();

Thread t1 = new Thread() {

public void run() {

shop.buy();

}

};

Thread t2 = new Thread() {

public void run() {

shop.buy();

}

};

t1.start();

t2.start();

}

}

class Shop{

// public synchronized void buy() {

public void buy() {

Thread t = Thread.currentThread();

try {

System.out.println(t.getName()+":正在挑衣服...");

Thread.sleep(5000);

/\*

\* 使用同步块精确的控制需要同步的代码片段。

\* 需要注意，若想让多个线程同步运行控制的代码片段

\* 必须保证同步监视器对象多个线程看到的是同一个。

\*/

synchronized (this) {

// synchronized (new Object()) {//这样做就没有同步效果

System.out.println(t.getName()+":正在试衣服...");

Thread.sleep(5000);

}

System.out.println(t.getName()+":结账离开");

} catch (Exception e) {

}

}

}

案例11：静态方法若使用synchronized修饰后，那么该方法一定具有同步效果。

实际上静态方法的同步监视器对象为当前类的类对象。

public class SyncDemo3 {

public static void main(String[] args) {

Thread t1 = new Thread() {

public void run() {

Boo.dosome();

}

};

Thread t2 = new Thread() {

public void run() {

Boo.dosome();

}

};

t1.start();

t2.start();

}

}

class Boo{

public static synchronized void dosome() {

Thread t = Thread.currentThread();

try {

System.out.println(t.getName()+":正在执行dosome方 法...");

Thread.sleep(5000);

System.out.println(t.getName()+":运行dosome方法完 毕!");

} catch (Exception e) {

}

}

}

案例12：互斥锁

当使用synchronized控制多段代码，而这些同步块使用的同步监视器对象时 同一个时，那么控制的这些代码片段之间就是互斥关系，多个线程不能同时执 行它们。

public class SyncDemo4 {

public static void main(String[] args) {

final Foo foo = new Foo();

Thread t1 = new Thread() {

public void run() {

foo.methodA();

}

};

Thread t2 = new Thread() {

public void run() {

foo.methodB();

}

};

t1.start();

t2.start();

}

}

class Foo{

public synchronized void methodA() {

Thread t = Thread.currentThread();

try {

System.out.println(t.getName()+":正在执行A方法...");

Thread.sleep(5000);

System.out.println(t.getName()+":运行A方法完毕!");

} catch (Exception e) {

}

}

public synchronized void methodB() {

Thread t = Thread.currentThread();

try {

System.out.println(t.getName()+":正在执行B方法...");

Thread.sleep(5000);

System.out.println(t.getName()+":运行B方法完毕!");

} catch (Exception e) {

}

}

}

案例13：线程池

线程池主要有两个作用:

1:控制线程数量

2:重用线程

线程数量过多会导致内存消耗大，并且会产生CPU过度切换导致整体并发性能 降低的问题。并且频繁的创建销毁线程也会给系统带来负担。为此在实际开发 中出现上述情况时我们都应当使用线程池来管理线程。

public static void main(String[] args) {

//创建固定大小的线程池(线程数量为2)

ExecutorService threadPool

= Executors.newFixedThreadPool(2);

for(int i=0;i<5;i++) {

Runnable runn = new Runnable() {

public void run() {

Thread t = Thread.currentThread();

try {

System.out.println(t.getName()+":正在运行任务...");

Thread.sleep(5000);

System.out.println(t.getName()+":运行任务完毕");

} catch (Exception e) {

e.printStackTrace();

}

}

};

//将任务交给线程池

threadPool.execute(runn);

System.out.println("指派了一个任务给线程池");

}

/\*

\* 停止线程池

\* shutdown():调用后，线程池不会马上停止，这时的

\* 线程池不再接收新的任务，并且会将所有已经在线程

\* 池中的任务全部执行完毕后停止。

\* shutdownNow():线程池会强制中断所有正在运行的线

\* 程并立即停止。

\*/

threadPool.shutdownNow();

System.out.println("线程池关闭了！");

}

(十)Collection：集合

集合是一组数据结构，与数组一样，用来保存一组元素。集合提供了一系列操作元 素的相关方法，使用更方便。

Collection是所有集合的顶级接口，规定了所有集合都应当具备的方法。

有两个常用的派生接口：

java.util.List:可重复集且有序

java.util.Set:不可重复集

元素是否重复是依靠元素自身的equals比较的结果。

案例1：

Collection c = new ArrayList();

/\*

\* boolean add(E e)

\* 向当前集合中添加给定元素，当该元素成功添加则

\* 返回true。

\* 由于Set集合是不可重复集，所以添加重复元素是会

\* 返回false的。List集合不存在该情况。

\*/

c.add("one");

c.add("two");

c.add("three");

c.add("four");

c.add("five");

System.out.println(c);

/\*

\* int size()

\* 返回当前集合的元素个数

\*/

int size = c.size();

System.out.println("size:"+size);

/\*

\* boolean isEmpty()

\* 判断当前集合是否为空集(不含有任何元素)

\*/

boolean empty = c.isEmpty();

System.out.println("是否为空集:"+empty);

//清空集合

c.clear();

System.out.println("集合已清空");

System.out.println("size:"+c.size());

System.out.println("是否为空集:"+c.isEmpty());

案例2：集合提供了判断是否包含给定元素的方法：

boolean contains(E e)

若集合包含当前元素则返回true，否则返回false

Collection c = new ArrayList();

c.add(new Point(1,2));

c.add(new Point(3,4));

c.add(new Point(5,6));

c.add(new Point(7,8));

System.out.println(c);

Point p = new Point(1,2);

// c.add(p);

/\*

\* contains方法会用参数对象与集合现有元素顺序

\* 进行equals比较(元素自身的equals方法)，若有

\* 返回值为true的则集合认为包含该元素。

\* 所以元素自身equals方法直接决定集合判断包含

\* 该元素的结果。

\*/

boolean contains = c.contains(p);

System.out.println("是否包含:"+contains);

案例3：删除集合元素

boolean remove(E e)

将给定的元素从集合中删除，需要注意，该方法依然是依靠元素equals 比较。

Collection c = new ArrayList();

c.add(new Point(1,2));

c.add(new Point(3,4));

c.add(new Point(5,6));

c.add(new Point(7,8));

c.add(new Point(1,2));

System.out.println(c);

Point p = new Point(1,2);

c.remove(p);

System.out.println(c);

案例4：集合存放的是元素的引用

public static void main(String[] args) {

Collection c = new ArrayList();

int a = 1;

Point p = new Point(2,3);

c.add(p);

String s = "hello";

test(c,a,p,s);

System.out.println("c:"+c);

System.out.println("a:"+a);

System.out.println("p:"+p);

System.out.println("s:"+s);

}

public static void test(Collection c,int a,Point p,String s) {

p.setX(3);

a = a+1;

s = "helloworld";

p = new Point(5,6);

c.clear();

c.add(p);

c = new ArrayList();

c.add(new Point(7,8));

p.setY(a);

}

案例5：集合操作

Collection c1 = new ArrayList();

c1.add("java");

c1.add("c");

c1.add("c++");

System.out.println(c1);

Collection c2 = new HashSet();

c2.add("android");

c2.add("ios");

System.out.println(c2);

/\*

\* boolean addAll(Collection c)

\* 将给定集合中的所有元素添加到当前集合中。添加后

\* 当前集合元素发生改变则返回true。

\*/

c1.addAll(c2);

System.out.println(c1);

/\*

\* boolean containsAll(Collection c)

\* 判断当前集合是否包含给定集合中的所有元素

\*/

Collection c3 = new ArrayList();

c3.add("java");

c3.add("ios");

c3.add("php");

boolean contains = c1.containsAll(c3);

System.out.println("全包含:"+contains);

/\*

\* boolean removeAll(Collection c)

\* 删除当前集合中与给定集合的共有元素(删除交集部分)

\*/

c1.removeAll(c3);

System.out.println(c1);

案例6：遍历集合

对于Collection这个层面而言，下面的集合实现类不都是有序的，所以 无法通过如数组那样操作下标来遍历元素。Collection提供了统一的遍 历集合元素的方式:迭代器模式。

Iterator iterator()

该方法可以获取一个用于遍历当前集合的迭代器实现类

java.util.Iterator

迭代器接口，规定了迭代器遍历集合的相关操作方法，不同的集合都实现 了一个用于遍历自身元素的迭代器实现类。我们无需记住这些实现类的名 字，用多态的角度当他们是Iterator看待即可。

迭代器遍历集合的规则为:问，取，删

其中删除元素不是必须操作。

Collection c = new HashSet();

c.add("one");

c.add("#");

c.add("two");

c.add("#");

c.add("three");

c.add("#");

c.add("four");

c.add("#");

c.add("five");

System.out.println(c);

/\*

\* 迭代器常用方法:

\* boolean hasNext()

\* 判断当前集合是否还有元素可以迭代

\*

\* E next()

\* 取出下一个元素

\*/

Iterator it = c.iterator();

while(it.hasNext()) {

String str = (String)it.next();

System.out.println(str);

if("#".equals(str)) {

/\*

\* 迭代器要求在遍历的过程中不要通过集合

\* 的方法增删元素，否则会抛出异常

\*/

// c.remove(str);

/\*

\* 迭代器的remove方法删除的是通过next

\* 方法取出的元素。

\*/

it.remove();

}

}

System.out.println(c);

案例7：JDK1.5之后推出了一个新的特性:增强for循环，也称为:新循环，for each

不取代传统for循环的工作，新循环只用来遍历集合或数组使用

String[] array = {"one","two","three","four","five"};

for(int i=0;i<array.length;i++) {

String str = array[i];

System.out.println(str);

}

/\*

\* 新循环并非新的语法，编译器认可而不是JVM认可。

\* 编译器在编译源程序时若发现使用新循环遍历数组时

\* 会将代码改为使用普通for循环遍历

\*/

for(String str : array) {

System.out.println(str);

}

案例8：使用新循环遍历集合

Collection c = new ArrayList();

c.add("one");

c.add("two");

c.add("three");

c.add("four");

c.add("five");

System.out.println(c);

/\*

\* 新循环遍历集合，会被编译器改为使用迭代器

\* 遍历集合。

\* 所以在使用新循环遍历集合过程中，不要通过集合

\* 的方法增删元素。否则新循环遍历集合会抛出异常

\*/

for(Object o:c) {

String str = (String)o;

System.out.println(str);

}

System.out.println(c);

案例9：泛型：JDK1.5之后推出的一个新的特性，泛型也称为参数化类型，意图在于将 一个类中属性，方法参数，返回值等类型的定义权交给了使用者，这样大大提 高了代码的灵活性。泛型的原型是Object，即:在使用时若不明确指定泛型的 实际类型时，则默认按照原型Object使用。

public class Pointers<T> {

private T x;

private T y;

public Pointers(T x, T y) {

super();

this.x = x;

this.y = y;

}

public T getX() {

return x;

}

public void setX(T x) {

this.x = x;

}

public T getY() {

return y;

}

public void setY(T y) {

this.y = y;

}

public String toString() {

return "("+x+","+y+")";

}

}

案例10：泛型应用

/\*

\* 泛型原型就是Object

\* 使用时，要指定泛型的实际类型

\*/

Pointers<Integer> p1 = new Pointers<Integer>(1,2);

//编译器检测实参类型是否符合泛型要求

p1.setX(2);

//获取泛型值时，编译器会在编译后加上造型

/\*

\* 下面的代码会被编译器改为:

\* int x1 = (Integer)p1.getX();

\* (实际上还会触发自动拆箱特性，并最终改变为)

\* int x1 = ((Integer)p1.getX()).intValue();

\*/

int x1 = p1.getX();

System.out.println("x1:"+x1);

System.out.println("p1:"+p1);

Pointers<Double> p2 = new Pointers<Double>(1.1,2.2);

p2.setX(3.3);

double x2 = p2.getX();

System.out.println("x2:"+x2);

System.out.println("p2:"+p2);

Pointers<String> p3 = new Pointers<String>("一","二");

p3.setX("三");

String x3 = p3.getX();

System.out.println("x3:"+x3);

System.out.println("p3:"+p3);

/\*

\* 若在使用时不指定泛型的实际类型，那么就使用原型

\* Object。不推荐这样的做法。编译器也会提示警告。

\*/

Pointers p4 = new Pointers("一","二");

案例11：泛型是编译器认可，而非虚拟机

Pointers<Integer> p1 = new Pointers<Integer>(1,2);

p1.setX(2);//编译时检测实参是否符合泛型要求

int x1 = p1.getX();//编译后会添加造型

System.out.println("x1:"+x1);

System.out.println("p1:"+p1);

//p2也引用p1指向的实例

Pointers p2 = p1;

System.out.println("p2:"+p2);

p2.setX("三");

System.out.println("p2:"+p2);

x1 = p1.getX();//抛出类造型异常!

System.out.println("x1:"+x1);//执行不到

案例12：泛型在集合中使用广泛，用于约束集合中的元素类型

Collection<String> c = new ArrayList<String>();

c.add("one");

c.add("two");

c.add("three");

c.add("four");

// c.add(1);//再传入其他类型参数时编译不通过

/\*

\* 新循环遍历就无需再用Object接收元素了

\*/

for(String str : c) {

System.out.println(str);

}

/\*

\* 迭代器也支持泛型，实际类型与其遍历的集合的泛型

\* 一致即可

\*/

Iterator<String> it = c.iterator();

while(it.hasNext()) {

String str = it.next();

System.out.println(str);

}

案例13：java.util.List

List集合:可重复集，并且有序，特点是可以通过下标操作元素

常用实现类:

java.util.ArrayList:内部由数组实现,查询性能更好

java.util.LinkedList:内部由链表实现,增删元素性能更好，尤其首尾增删元素。

/\*

\* void add(int index,E e)

\* 将给定元素添加到指定位置

\*/

List<String> list = new ArrayList<String>();

list.add("one");

list.add("two");

list.add("three");

list.add("four");

System.out.println(list);

//[one,2,two,three,four]

list.add(1, "2");

System.out.println(list);

/\*

\* E remove(int index)

\* 删除指定位置上的元素，并将其返回

\*/

//[one,2,three,four]

String old = list.remove(2);

System.out.println(list);

System.out.println("old:"+old);//two

案例14：List提供了一对get,set方法

List<String> list = new ArrayList<String>();

list.add("one");

list.add("two");

list.add("three");

list.add("four");

System.out.println(list);

/\*

\* E get(int index)

\* 返回给定下标处对应的元素

\* String str = list.get[i];

\* String str = array[i];

\*/

//查看第二个元素

String str = list.get(1);

System.out.println(str);

for(int i=0;i<list.size();i++) {

str = list.get(i);

System.out.println(str);

}

/\*

\* E set(int index,E e)

\* 将给定元素设置到指定位置，返回值为原位置对应的

\* 元素(替换元素操作)

\*/

//[one,2,three,four]

String old = list.set(1, "2");

System.out.println(list);

System.out.println("被替换元素:"+old);

案例15：List提供了获取子集的方法:

List subList(int start,int end)

获取当前集合中指定范围内的子集

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

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

list.add(i);

}

System.out.println(list);

/\*

\* 获取子集3-7

\*/

List<Integer> subList = list.subList(3, 8);

System.out.println(subList);

/\*

\* 将子集各元素扩大10倍

\*/

for(int i=0;i<subList.size();i++) {

int d = subList.get(i);

d = d \* 10;

subList.set(i, d);

}

//[30,40,50,60,70]

System.out.println(subList);

/\*

\* 对子集操作就是对原集合对应元素的操作

\*/

System.out.println(list);

/\*

\* 删除list集合2-8

\*/

list.subList(2, 9).clear();

System.out.println(list);

案例16：集合转换为数组

Collection提供了方法toArray，可以将当前集合转换为一个数组

Collection<String> c = new ArrayList<String>();

c.add("one");

c.add("two");

c.add("three");

c.add("four");

System.out.println(c);

// Object[] array = c.toArray();

/\*

\* 该toArray方法要求传入一个数组，若该数组可用(长度

\* 可以存放集合所有元素)则使用该数组，若不可用则会

\* 自行创建一个与集合size一样的数组。

\*/

String[] array = c.toArray(new String[c.size()]);

System.out.println(array.length);

for(String str : array) {

System.out.println(str);

}

案例17：数组转换为集合

Arrays是数组的工具类，其提供了一个静态方法:asList

该方法可以将给定的数组转换为一个List集合

String[] array = {"one","two","three","four","five"};

List<String> list = Arrays.asList(array);

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

System.out.println(list);

/\*

\* 对集合元素的操作就是对数组对应元素的操作

\*/

list.set(1, "2");

System.out.println(list);

for(String str : array) {

System.out.println(str);

}

/\*

\* 由于数组是定长的，所有该集合不允许增删元素。

\* 否则会抛出异常

\*/

// list.add("six");

/\*

\* 若想对集合操作，需要自行创建一个集合并包含数组

\* 转换的集合中所有元素即可。

\*/

// List<String> list2 = new ArrayList<String>();

// list2.addAll(list);

//该构造方法可以创建当前集合时包含给定集合元素

List<String> list2 = new ArrayList<String>(list);

System.out.println("list2:"+list2);

list2.add("six");

System.out.println("list2:"+list2);

案例17：集合的排序

java.util.Collections是集合的工具类，提供了一系列操作集合的静态方法。 其中sort方法可以对List集合进行自然排序，即:从小到大排序集合元素

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

Random random = new Random();

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

list.add(random.nextInt(100));

}

System.out.println(list);

//排序

Collections.sort(list);

System.out.println(list);

//乱序

Collections.shuffle(list);

System.out.println(list);

案例18：排序自定义类型元素的List集合

//作为集合元素测试Collections的排序

//Collections的sort(List list)方法要求集合元素必须实现Comparable接口

public class Point implements Comparable<Point>{

private int x;

private int y;

public Point() {

}

public Point(int x, int y) {

super();

this.x = x;

this.y = y;

}

public int getX() {

return x;

}

public void setX(int x) {

this.x = x;

}

public int getY() {

return y;

}

public void setY(int y) {

this.y = y;

}

/\*\*

\* 重写Object的toString方法。返回的字符串格式上

\* 没有要求，按照将来实际开发需求而定。原则上应当

\* 包含当前对象的有效信息(属性值)。

\*/

public String toString() {

// (1,2)

return "(" + x + "," + y + ")";

}

/\*\*

\* 通常我们需要使用一个对象的equals方法就应当重写

\* 它(java提供的类不需要，已经重写好了)。

\* 重写equals的目的是比较两个对象this和参数o的内容

\* 是否相同。而比较也并非要求必须所有属性值都相同。

\* 可结合实际开发需求而定。

\*/

public boolean equals(Object o) {

if(o == null) {

return false;

}

if(o == this) {

return true;

}

if(o instanceof Point) {

Point p = (Point)o;

return this.x==p.x && this.y==p.y;

}

return false;

}

/\*\*

\* this compareTo o

\* 当实现了Comparable接口后，就必须重写抽象方法

\* compareTo,该方法的作用是定义当前对象与参数对象

\* 的比较大小规则。

\* 该方法的返回值为int值，该值不关心具体取值，关心

\* 的是取值范围。

\* 当返回值>0:当前对象大于参数对象

\* this>o

\* 当返回值<0:当前对象小于参数对象

\* 当返回值=0:两个对象相等

\*/

public int compareTo(Point o) {

/\*

\* 点到原点距离长的大

\*/

int len = this.x\*this.x+this.y\*this.y;

int olen = o.x\*o.x+o.y\*o.y;

return len-olen;

}

}

//排序自定义类型元素的List集合

public class SortDemo2 {

public static void main(String[] args) {

List<Point> list = new ArrayList<Point>();

list.add(new Point(3,4));

list.add(new Point(1,9));

list.add(new Point(2,2));

list.add(new Point(6,3));

list.add(new Point(7,8));

System.out.println(list);

/\*

\* sort方法要求集合元素必须实现Comparable接口，否则

\* 编译不通过

\*/

Collections.sort(list);

System.out.println(list);

}

}

案例19：

//作为集合元素测试Collections的排序

//Collections的sort(List list)方法要求集合元素必须实现Comparable接口。

//当排序含有自定义类型元素的集合时，不建议使用上述的sort方法，而应当使 用重载的需要传入额外比较器的sort方法:

static void sort(List list,Comparator com)

因为该方法不要求集合元素必须实现Comparable接口，那 么该方法对我们集 合元素Point就没有侵入性。

侵入性:当我们需要调用某个功能方法时，该方法要求我们的程序为其修改其他的 额外代码，修改的越多，侵入性越强。侵入性对后期系统维护不利，在开发中应当 尽量避免

//public class Point implements Comparable<Point>{

public class Point{

private int x;

private int y;

public Point() {

}

public Point(int x, int y) {

super();

this.x = x;

this.y = y;

}

public int getX() {

return x;

}

public void setX(int x) {

this.x = x;

}

public int getY() {

return y;

}

public void setY(int y) {

this.y = y;

}

/\*\*

\* 重写Object的toString方法。返回的字符串格式上

\* 没有要求，按照将来实际开发需求而定。原则上应当

\* 包含当前对象的有效信息(属性值)。

\*/

public String toString() {

// (1,2)

return "(" + x + "," + y + ")";

}

/\*\*

\* 通常我们需要使用一个对象的equals方法就应当重写

\* 它(java提供的类不需要，已经重写好了)。

\* 重写equals的目的是比较两个对象this和参数o的内容

\* 是否相同。而比较也并非要求必须所有属性值都相同。

\* 可结合实际开发需求而定。

\*/

public boolean equals(Object o) {

if(o == null) {

return false;

}

if(o == this) {

return true;

}

if(o instanceof Point) {

Point p = (Point)o;

return this.x==p.x && this.y==p.y;

}

return false;

}

/\*\*

\* this compareTo o

\* 当实现了Comparable接口后，就必须重写抽象方法

\* compareTo,该方法的作用是定义当前对象与参数对象

\* 的比较大小规则。

\* 该方法的返回值为int值，该值不关心具体取值，关心

\* 的是取值范围。

\* 当返回值>0:当前对象大于参数对象

\* this>o

\* 当返回值<0:当前对象小于参数对象

\* 当返回值=0:两个对象相等

\*

\*/

// public int compareTo(Point o) {

// /\*

// \* 点到原点距离长的大

// \*/

// int len = this.x\*this.x+this.y\*this.y;

// int olen = o.x\*o.x+o.y\*o.y;

// return len-olen;

// }

}

//Collections的sort(List list)方法是按照集合元素自身的比较规则比较后由 小到大进行排序的。而有时集合元素已经实现了比较规则，但是该规则不满足我们 排序需求。

Collections提供了一个重载的sort方法:

static void sort(List list,Comparator com)

该排序方法要求再传入一个参数，该参数是一个比较器，该方法会使用该比较器的 比较规则对集合元素进行比较并按照比计较的结果进行排序。

public class SortListDemo3 {

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

list.add("苍老师");

list.add("传奇");

list.add("小泽老师");

System.out.println(list);[苍老师，传奇，小泽老师]

// Collections.sort(list);

// MyComparator com = new MyComparator();

//通常使用匿名内部类创建比较器

Comparator<String> com = new Comparator<String>() {

public int compare(String o1, String o2) {

return o1.length()-o2.length();//从小到大的顺序排列

}

};

Collections.sort(list,com);

System.out.println(list);[传奇，苍老师，小泽老师]

}

}

/\*\*

\* 比较器

\*/

class MyComparator implements Comparator<String>{

/\*\*

\* 返回值>0:o1大于o2

\* 返回值<0:o1小于o2

\* 返回值=0:o1等于o2

\*/

public int compare(String o1, String o2) {

return o1.length()-o2.length();

}

}

案例20：排序自定义类型元素的List集合

List<Point> list = new ArrayList<Point>();

list.add(new Point(3,4));

list.add(new Point(1,9));

list.add(new Point(2,2));

list.add(new Point(6,3));

list.add(new Point(7,8));

System.out.println(list);

/\*

\* sort方法要求集合元素必须实现Comparable接口，否则

\* 编译不通过

\*/

// Collections.sort(list);

Collections.sort(list,new Comparator<Point>() {

public int compare(Point o1, Point o2) {

int len1 = o1.getX()\*o1.getX()+o1.getY()\*o1.getY();

int len2 = o2.getX()\*o2.getX()+o2.getY()\*o2.getY();

return len1-len2;

}

});

System.out.println(list);

案例21：java.util.Queue 队列

队列是经典的数据结构，可以保存一组元素，但是存取元素必须遵循先进先出原则。

/\*

\* LinkedList实现了队列接口，因为链表本身可以保存

\* 一组元素，并且特点是首尾增删元素效率高，这正好

\* 满足队列特点。

\*/

Queue<String> queue = new LinkedList<String>();

/\*

\* boolean offer(E e)

\* 入队操作，将给定元素添加到队列末尾

\*/

queue.offer("one");

queue.offer("two");

queue.offer("three");

queue.offer("four");

queue.offer("five");

System.out.println(queue);

/\*

\* E poll()

\* 出队操作

\* 获取队首元素，获取后该元素即从队列中被删除

\*/

String str = queue.poll();

System.out.println(str);

System.out.println(queue);

/\*

\* E peek()

\* 引用队首元素

\*/

str = queue.peek();

System.out.println(str);

System.out.println(queue);

/\*

\* 由于队列接口继承自Collection，所以可以使用

\* 迭代器遍历队列，并且不会影响队列中的元素

\*/

for(String s : queue) {

System.out.println(s);

}

System.out.println(queue);

System.out.println("开始遍历");

/\*

\* 用poll方法遍历队列元素

\*/

while(queue.size()>0) {

str = queue.poll();

System.out.println(str);

}

System.out.println(queue);

}

案例22：java.util.Deque 双端队列

Deque接口继承自Queue，双端队列是两端都可以做进出队操作的队列。并且 提供了明确方向的进出队方法

常见实现类:java.util.LinkedList

Deque<String> deque = new LinkedList<String>();

deque.offer("one");

deque.offer("two");

System.out.println(deque);

//从队首入队

deque.offerFirst("three");

System.out.println(deque);

//从队尾入队

deque.offerLast("four");

System.out.println(deque);

String str = deque.poll();

System.out.println(str);

System.out.println(deque);

str = deque.pollFirst();

System.out.println(str);

System.out.println(deque);

str = deque.pollLast();

System.out.println(str);

System.out.println(deque);

案例23：stack 栈

栈也是一个经典的数据结构，可以保存一组元素，但是存取必须遵循先进后出原则。

通常使用栈是为了实现"后退"这样的功能。

Deque双端队列可以实现栈操作，并且为栈也提供了对应的入栈

与出栈方法(push,pop).

Deque<String> stack = new LinkedList<String>();

stack.push("one");

stack.push("two");

stack.push("three");

stack.push("four");

System.out.println(stack);

String str = stack.pop();

System.out.println(str);

System.out.println(stack);

for(String s : stack) {

System.out.println(s);

}

System.out.println(stack);

/\*

\* 用pop遍历

\*/

while(stack.size()>0) {

str = stack.pop();

System.out.println(str);

}

System.out.println(stack);

案例24：线程安全的集合

集合常用的实现类有:ArrayList,LinkedList,HashSet

它们都不是线程安全的，在多线程并发操作时会出现并发安全问题

Collections提供了一组静态方法，可以将现有的集合转换为

一个线程安全的。

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

list.add("one");

list.add("two");

list.add("three");

list.add("four");

System.out.println(list);

/\*

\* 将给定的list集合转换为一个线程安全的List

\*/

list = Collections.synchronizedList(list);

System.out.println(list);

//hashset不是线程安全的

Set<String> set = new HashSet<String>(list);

System.out.println(set);

//先给定的set集合转换为线程安全的

set = Collections.synchronizedSet(set);

System.out.println(set);

/\*

\* API手册上也有说明，对于一个线程安全的集合而言

\* 也不与迭代器遍历该集合的操作做互斥。所以若存在

\* 多个线程遍历以及增删元素同时操作的情况，要自行

\* 维护遍历与增删元素的互斥

\*/

/\*

\* 阻塞队列，双缓冲队列

\* 阻塞队列是并发安全的队列，在多线程情况下使用。

\* 并且由于内部由双缓冲实现，并发效率比较好。

\* BlockingQueue

\* 常用实现类:

\* ArrayBlockingQueue

\* LinkedBlockingQueue

\*/

LinkedBlockingQueue<String> queue

= new LinkedBlockingQueue<String>();

queue.offer("one");

queue.offer("two");

System.out.println(queue);

String str = queue.poll();

System.out.println(str);

System.out.println(queue);

/\*

\* 阻塞式入队

\*/

try {

/\*

\* 对于有界队列，有可能会存在队列满了存不进去的情

\* 况，这时可以使用阻塞入队操作，指定超时时间以及

\* 时间单位，在规定时间内若可以将元素入队则方法执

\* 行完毕，否则会抛出超时异常。在等待的过程中会发

\* 生该线程阻塞现象。

\*/

queue.offer("four", 500, TimeUnit.MILLISECONDS);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

(十一)Map 查找表：常用的数据结构，体现的样子为一个多行两列的表格

常用的实现类:HashMap

java.util.HashMap:散列表

使用散列算法实现的Map，当今查询速度最快的数据结构。也称为哈希表

案例1：

/\*

\* Map分为两列，左列称为key，右列称为value

\* key通常保存的是查询条件，value保存的是对应

\* 的值。

\*/

Map<String,Integer> map

= new HashMap<String,Integer>();

/\*

\* V put(K k,V v)

\* 将给定的一组键值对存入到Map中。

\* 注意，Map有一个要求，key在Map中不允许重复，

\* 是否重复是依靠key元素自身equals比较的结果。

\* 若使用重复的key存入新的value，那么该操作为

\* 替换value操作，这时put方法的返回值为被替换的

\* 原value，若put的key在map中不存在，则直接保存

\* 该组键值对并且返回值为null。

\*/

/\*

\* 需要注意，若value的类型是一个包装类，那么在

\* 接收put方法的返回值时不要用其对应的基本类型

\* 接收，避免出现自动拆箱。因为若put方法返回值

\* 为null时自动拆箱会引发空指针异常。

\*

\*/

Integer value = map.put("语文", 99);

System.out.println("value:"+value);

map.put("数学", 98);

map.put("英语", 97);

map.put("物理", 99);

map.put("化学", 96);

System.out.println(map);

value = map.put("语文", 70);

System.out.println(map);

System.out.println(value);

/\*

\* V get(K k)

\* 根据给定的key获取对应的value

\* 若给定的key在Map中不存在，则返回值为null

\*/

value = map.get("数学");

System.out.println("数学:"+value);

value = map.get("体育");

System.out.println("体育:"+value);

/\*

\* V remove(K k)

\* 删除给定的key所对应的这组键值对，返回值为该

\* key所对应的value。

\*/

value = map.remove("语文");

System.out.println(map);

System.out.println(value);

案例2：

HashMap之所以是查询速度最快的数据结构，是因为HashMap本

身也是由数组实现，但是它根据key的hashcode值计算出其应

在数组的位置并将其存放，这样当根据key查找时也直接根据其

hashcode值计算下标快速定位到数组对应位置检索对应的value

这样省去了遍历数组的操作，从而不受数据大小影响查询性能。

对于Key而言，其hashcode方法与equals方法直接影响其在HashMap

内部数组的位置，所以要妥善重写这两个方法，避免在数组中产生

链表，否则会影响散列表查询性能。

产生链表的原因之一:当两个Key的hashcode值相同时，计算对应的

数组下标位置相同，但是这两个Key的equals不同(不是同一个key,

若equals相同Map会认为是相同的key，那么就做替换value操作了)

这时就会在数组该位置处产生链表。

java提供的类都妥善的重写了equals与hashcode，包括Object类

都满足该重写规则，但是若我们自定义的类若需要重写equals方法

时应当一同重写hashcode，这在API手册上有明确说明。

重写规则:

1:成对重写，即:当我们重写一个类的equals方法时就应当连同

重写hashcode方法。

2:一致性,即:当两个对象的equals比较为true时，hashcode返回

的数字必须相等。反之虽然不是必须的，但是最好也要保证当

两个对象hashcode相等时，equals比较也为true。否则会在

HashMap中作为key使用时产生链表，影响查询性能。

3:稳定性,即:当参与equals比较的属性值没有发生改变的前提下，

多次调用hashcode方法返回的数字不能改变。

public class Key {

private int x;

private int y;

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + x;

result = prime \* result + y;

return result;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Key other = (Key) obj;

if (x != other.x)

return false;

if (y != other.y)

return false;

return true;

}

}

案例3：Map的遍历分为三种：

1:遍历所有的key

2:遍历每组键值对

3:遍历所有的value(相对不常用)

/\*

\* LinkedHashMap是有序的Map，遍历时的顺序可以与

\* put元素时的顺序一致。

\*/

Map<String,Integer> map

// = new HashMap<String,Integer>();

= new LinkedHashMap<String,Integer>();

map.put("语文", 99);

map.put("数学", 98);

map.put("英语", 97);

map.put("物理", 99);

map.put("化学", 96);

System.out.println(map);

/\*

\* 遍历所有的key

\*

\* Set<K> keySet()

\* 将当前Map中所有的key以一个Set集合形式返回

\* 遍历该Set集合等于遍历了所有的key

\*

\*/

Set<String> keySet = map.keySet();

for(String key : keySet) {

System.out.println("key:"+key);

}

/\*

\* 遍历每一组键值对

\* java.util.Map.Entry

\* Entry用来定义Map中每一组键值对，即:每个实例

\* 表示一个具体的键值对。

\* Entry提供了获取其表示的键值对中key与value的

\* 方法

\* K getKey()

\* V getValue()

\*

\* Map提供了遍历键值对的方法:

\* Set<Entry> entrySet()

\* 将当前Map中每组键值对(Entry实例)存入Set集合

\* 并返回

\*/

Set<Entry<String,Integer>> entrySet = map.entrySet();

for(Entry<String,Integer> e : entrySet) {

String key = e.getKey();

Integer value = e.getValue();

System.out.println(key+":"+value);

}

/\*

\* 遍历所有的value

\* Collection values()

\* 将所有的value存入一个集合后返回

\*/

Collection<Integer> values = map.values();

for(Integer value : values) {

System.out.println("value:"+value);

}

案例4：判断当前Map是否包含给定的key或value

Map<String,Integer> map = new HashMap<String,Integer>();

map.put("语文", 99);

map.put("数学", 98);

map.put("英语", 97);

map.put("物理", 99);

map.put("化学", 96);

System.out.println(map);

/\*

\* boolean containsKey(K k)

\* boolean containsValue(V v)

\*/

boolean ck = map.containsKey("语文");

System.out.println("是否包含k:"+ck);

boolean cv = map.containsValue(96);

System.out.println("是否包含v:"+cv);

案例5：HashMap并非线程安全的，同样可以使用Collections将一个

Map转换为线程安全的

Map<String,Integer> map= new HashMap<String,Integer>();

map.put("语文", 99);

map.put("数学", 98);

map.put("英语", 97);

System.out.println(map);

map = Collections.synchronizedMap(map);

System.out.println(map);

(十二)XML

案例1：解析XML文档

List<Emp> list = new ArrayList<Emp>();

/\*

\* 使用DOM解析XML的大致步骤

\* 1:创建SAXReader

\* 2:使用SAXReader读取指定的xml文档并生成

\* Document对象

\* 这一步就是DOM解析耗时耗资源的地方，因为

\* 要先将XML文档全部读取完毕并以Document对象

\* 形式保存在内存中

\* 3:通过Document获取根元素

\* 4:根据XML文档的结构逐级获取子元素以达到遍历

\* XML文档的目的

\*/

try {

/\*

\* org.dom4j.io.SAXReader

\* 如果eclipse提示找不到这个类，那么原因在于

\* 没有导入dom4j的jar包

\*

\* 1创建SAXReader

\*/

SAXReader reader = new SAXReader();

//2

// Document doc = reader.read(new File("emplist.xml"));

Document doc = reader.read(

new FileInputStream("emplist.xml")

);

/\*

\* 3

\* Document提供了获取根元素的方法:

\* Element getRootElement()

\*

\* Element的每一个实例用于表示XML文档中的一个元素(一对标 签)

\* Element提供了获取其表示的元素的相关方法常用的有:

\*

\* String getName()

\* 获取当前标签的名字

\*

\* String getText()

\* 获取当前标签中间的文本，例如:<a>获取的文本</a>

\*

\* Element element(String name)

\* 获取当前标签中指定名字的子标签

\*

\* List elements()

\* 获取当前标签中所有子标签

\*

\* List elements(String name)

\* 获取当前标签中所有同名(指定的名字)子标签

\*/

Element root = doc.getRootElement();

/\*

\* 获取根标签<list>中所有的<emp>标签

\*/

List<Element> empList = root.elements("emp");

for(Element empEle : empList) {

//获取员工名字

//1获取<name>标签

Element nameEle = empEle.element("name");

//2获取<name>标签中间的文本

String name = nameEle.getText();

System.out.println(name);

//获取年龄

int age = Integer.parseInt(

// empEle.element("age").getText()

empEle.elementText("age")

);

//获取性别

String gender = empEle.elementText("gender");

//获取工资

int salary = Integer.parseInt(

empEle.elementText("salary")

);

/\*

\* 获取<emp id="xx">标签的属性id对应的值

\* Element提供了获取属性的方法:

\* Attribute attribute(String name)

\* 获取指定名字的属性

\*

\* Attribute的每个实例用于表示某个标签中的一个属性

\* 其有两个常用方法:

\* String getName():获取属性的名字

\* String getValue():获取属性的值

\*

\* Element也提供了快捷获取属性值的方法:

\* String attributeValue(String name)

\* 获取当前标签中指定名字对应属性的值

\*

\*/

// Attribute attr = empEle.attribute("id");

// int id = Integer.parseInt(

// attr.getValue()

// );

int id = Integer.parseInt(

empEle.attributeValue("id")

);

Emp emp = new Emp(id, name, age, gender, salary);

list.add(emp);

}// loop end

System.out.println("解析完毕!");

System.out.println("获取了"+list.size()+"个员工");

for(Emp e : list) {

System.out.println(e);

}

} catch (Exception e) {

e.printStackTrace();

}

案例2：使用DOM生成XML文档

/\*

\* 生成一个XML文档的大致步骤

\* 1:创建一个Document实例，表示一个空白文档

\* 2:向Document实例中添加根元素

\* 3:按照预定生成的XML文档结构向根元素中逐级

\* 添加子元素以及对应数据

\* 4:创建XmlWriter

\* 5:通过XmlWriter将Document写出以生成XML文档

\*

\*/

List<Emp> list = new ArrayList<Emp>();

list.add(new Emp(1,"张三",22,"男",5000));

list.add(new Emp(2,"李四",23,"女",6000));

list.add(new Emp(3,"王五",24,"男",7000));

list.add(new Emp(4,"赵六",25,"女",8000));

list.add(new Emp(5,"钱七",26,"男",9000));

try {

//1 创建一个空白文档

Document doc = DocumentHelper.createDocument();

/\*

\* 2

\* Document提供了添加根元素的方法:

\* Element addElement(String name)

\* 向当前文档中添加给定名字的元素，由于一个

\* 文档中只能有一个根元素，所以该方法只能调

\* 用一次。该方法会返回一个Element实例，该

\* 实例就表示添加的元素,以便对其继续追加子元素

\* 等操作。

\*/

Element root = doc.addElement("list");

/\*

\* 3

\* 将集合中每个Emp实例以一个<emp>元素的形式

\* 添加到根元素<list>中

\*/

for(Emp emp : list) {

//向根标签中添加子标签emp

Element empEle = root.addElement("emp");

//添加名字

Element nameEle = empEle.addElement("name");

nameEle.addText(emp.getName());

//添加年龄

Element ageEle = empEle.addElement("age");

ageEle.addText(emp.getAge()+"");

//添加性别

Element genderEle = empEle.addElement("gender");

genderEle.addText(emp.getGender());

//添加工资

Element salaryEle = empEle.addElement("salary");

salaryEle.addText(emp.getSalary()+"");

//添加id(属性)

empEle.addAttribute("id", emp.getId()+"");

}

//4

XMLWriter writer = new XMLWriter(

new FileOutputStream("myemp.xml"),

OutputFormat.createPrettyPrint()

);

//5

writer.write(doc);

System.out.println("写出完毕!");

writer.close();

} catch (Exception e) {

e.printStackTrace();

}

(十三)Reflect java反射机制

反射机制允许我们的程序在运行时动态加载一个类，并实例化，或者获取其定义 的相关信息(属性，方法等)，甚至可以动态调用这些方法。而非传统的在编码期 间以编码形式实例化或调用方法等。这样做大大的提高了代码的灵活度。但是反 射也是一把双刃剑，过度的使用会降低系统性能。

案例1：

/\*

\* Class类

\* Class类的每一个实例是用来表示一个被JVM加载

\* 的类。通过它可以了解到其表示的类的相关信息，

\* 比如有哪些属性，构造方法，方法等等。

\* 所以我们也称Class的实例叫做类的类对象。

\* 在JVM内部，每个被加载的类都有且只有唯一的一个

\* Class的实例表示它。那么这个Class的实例时在JVM

\* 加载一个类的时候创建的。

\* 想获取一个类的类对象可以通过下面几种方式得到:

\* 1:每个类都有一个静态属性class,可以得到这个类

\* 的类对象

\* 例如:

\* Class cls = Person.class;

\* 通过这个cls可以获取所有有关Person的信息。

\* 比如Person有哪些方法，哪些属性，构造方法等

\* 等信息。

\* 2:通过Class的静态方法forName()加载指定的类并

\* 得到其类对象。

\* 例如:

\* Class cls = Class.forName("object.Person");

\* 加载object包下的Person类，需要注意，这里的

\* 参数必须写类的完全限定名(包名.类名)

\*

\* 3:还可以通过类加载器加载一个类

\*/

/\*

\* 获取reflect包下的Person类的类对象

\*/

// Class cls = Person.class;

//加载指定的类

Class cls = Class.forName("reflect.Person");

//获取类名

String name = cls.getName();

System.out.println(name);

案例2：通过Class实例化其表示的类的一个实例

//实例化Person

Person p = new Person();

System.out.println(p);

//利用反射机制实例化

//1加载Person的类对象

Class cls = Class.forName("reflect.Person");

/\*

\* 2通过类对象实例化其表示的类的实例

\* Class提供了一个newInstance()方法，该方法

\* 调用其表示的类的无参构造方法实例化它。

\*/

Object o = cls.newInstance();

System.out.println(o);

案例3：通过反射调用方法

Person p = new Person();

p.sayHello();

/\*

\* 使用反射调用方法

\* 1:加载对应的类

\* 2:获取该类定义的要调用的方法

\* 3:实例化该类

\* 4:调用该方法

\*/

//1

Class cls = Class.forName("reflect.Person");

/\*

\* 2

\* Method的每一个实例用于表示一个类中定义的一个方法

\* 通过Method可以获取与表示的方法相关的信息，比如:

\* 方法的返回值类型，方法名，参数，访问修饰符等等

\* 甚至可以调用该方法。

\* 下面的例子:

\* method实例表示的是Person类的成员方法sayHello()

\*/

Method method = cls.getDeclaredMethod("sayHello", null);

/\*

\* 3 实例化Person的实例

\*/

Object o = cls.newInstance();

/\*

\* 4 调用成员方法sayHello

\* o.sayHello();

\*/

method.invoke(o, null);

案例4：测试动态调用方法

Scanner scanner = new Scanner(System.in);

System.out.println("请输入要加载的类:");

String className = scanner.nextLine();

System.out.println("请输入要调用的方法:");

String methodName = scanner.nextLine();

Class cls = Class.forName(className);

Method method = cls.getDeclaredMethod(methodName, null);

Object o = cls.newInstance();

method.invoke(o, null);

案例5：调用有参方法

Person p = new Person();

p.sayName("张三");

Class cls = Class.forName("reflect.Person");

/\*

\* 获取有参方法

\* getDeclaredMethod第二个参数是一个Class类型的

\* 数组。数组的每个元素用于指定该方法的参数类型。

\* 数组元素的顺序必须与获取的方法的参数顺序一致

\* 例如:

\* 获取 sayName(String name)

\*/

Method method = cls.getDeclaredMethod(

"sayName", new Class[]{String.class});

Object o = cls.newInstance();

/\*

\* invoke方法在调用方法时，第二个参数为Object类型

\* 数组，数组每个元素为调用该方法时传入的一个实际

\* 参数

\* p.sayName("张三");

\* "张三"就是实际参数

\*/

method.invoke(o, new Object[]{"李四"});

案例6：调用多个参数的方法

/\*

\* 调用Person的sayInfo方法

\*/

Class cls = Class.forName("reflect.Person");

Method method = cls.getDeclaredMethod(

"sayInfo", new Class[]{String.class,int.class}

);

Object o = cls.newInstance();

method.invoke(o, new Object[]{"张三",22});

使用当前类测试反射机制

public class Person {

public void sayHello() {

System.out.println("大家好!");

}

public void sayHi() {

System.out.println("Hi!艾维巴蒂!");

}

public void sayName(String name) {

System.out.println("我的名字叫:"+name);

}

public void sayInfo(String name,int age) {

System.out.println("我的名字叫:"+name+",我今年:"+age+"岁");

}

private void dosome() {

System.out.println("我是私有方法！！！");

}

}

案例7：反射调用私有方法

Person p = new Person();

//类外面不可调用私有方法

// p.dosome();

Class cls = Class.forName("reflect.Person");

Method method = cls.getDeclaredMethod("dosome",null);

Object o = cls.newInstance();

//私有方法在设置强制方法后，是可以执行的

method.setAccessible(true);

method.invoke(o, null);

(十四)java.util.Date

Date的每一个实例用于表示一个具体的时间。

由于Date存在时区以及千年虫问题，所以大部分方法在1.1版

本就被声明为过时的。现在使用Date通常仅用于表示时间。

案例1：

Date date = new Date();

System.out.println(date);

/\*

\* long getTime()

\* 该方法返回一个long值，该值就是当前Date内部

\* 维护的long值，表示自1970年1月1日 00:00:00

\* 到当前Date表示的时间之间所经过的毫秒

\*/

long time = date.getTime();

System.out.println(time);

//表示明天这一刻

time = time+1000\*60\*60\*24;

//也可以设置一个long值，让Date表示该时间

date.setTime(time);

System.out.println(date);

案例2：java.text.SimpleDateFormat

SDF是用来在String与Date之间相互转换的API。

SDF转换是基于一个给定的日期格式。

Date now = new Date();

System.out.println(now);

/\*

\* 2018-07-23 10:32:21

\* yyyy-MM-dd HH:mm:ss

\*/

SimpleDateFormat sdf = new SimpleDateFormat(

"yyyy-MM-dd HH:mm:ss E a"

);

/\*

\* String format(Date date)

\* 该方法会将给定的Date按照当前SDF指定的日期格式

\* 转换为对应的字符串 Date->String

\*/

String line = sdf.format(now);

System.out.println(line);

案例3：将一个字符串解析为Date

String str = "2008-08-08 20:08:08";

SimpleDateFormat sdf = new SimpleDateFormat(

"yyyy-MM-dd HH:mm:ss"

);

/\*

\* Date parse(String str)

\* 将给定的字符串按照当前SDF指定的日期格式解析为

\* Date对象。

\*/

Date date = sdf.parse(str);

System.out.println(date);

案例4：编写一段代码，要求用户输入自己的生日，格式:yyyy-MM-dd

然后经过程序计算，输出到今天为止一共活了多少天。

再输出其出生10000天的纪念日是哪天?格式同上。

Scanner scanner = new Scanner(System.in);

System.out.println("请输入您的生日:");

String birthStr = scanner.nextLine();

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd);

Date birth = sdf.parse(birthStr);

Date now = new Date();

long time = now.getTime() - birth.getTime();

time = time /1000/60/60/24;

System.out.println("恭喜您，已经活了"+time+"天，请继续持!");

//计算10000天纪念日

time = birth.getTime()+10000L\*1000\*60\*60\*24;

Date date = new Date(time);

String line = sdf.format(date);

System.out.println("您出生10000天的纪念日为:"+line);

案例5：java.util.Calendar

日历类，使用日历类可以对时间进行一系列的操作。Date大部分操 作时间的方法都被声明为过时的，取而代之的也是使用 Calendar的 相关操作。Calendar是一个抽象类，定义了日历类操作时间的功能， 常用实现类为:java.util.GregorianCalendar,即:阳历Calendar提 供了一个静态方法:getInstance，该方法可以获取一个当前系统所在 地区适用的实现类，大部分地区返回的也是阳历实现类。

Calendar calendar = Calendar.getInstance();

/\*

\* Calendar默认创建也表示当前系统时间。

\* 但是toString返回的字符串不能直观的体现。

\*/

System.out.println(calendar);

/\*

\* Calendar提供了与Date之间互转的方法

\* Date getTime()

\* 将当前Calendar表示的日期以Date实例形式返回。

\*

\* void setTime(Date date)

\* 使当前Calendar表示给定的Date所表示的日期

\*/

Date date = calendar.getTime();

System.out.println(date);

calendar.setTime(date);

案例6：Calendar提供了一个方法:int get(int field)

该方法可以获取当前Calendar给定的时间分量所对应的值

不同的时间分量对应的数字不同，但是无需记住这些数字，

因为Calendar为这些数字提供了对应的常量。

Calendar calendar = Calendar.getInstance();

//获取年

int year = calendar.get(Calendar.YEAR);

//需要注意，月从0开始，即0表示1月，1表示2月...

int month = calendar.get(Calendar.MONTH)+1;

/\*

\* 与"天"相关的时间分量有:

\* DATE:月中的天

\* DAY\_OF\_MONTH:月中的天，与DATE一致

\* DAY\_OF\_WEEK:周中的天

\* DAY\_OF\_YEAR:年中的天

\*/

int day = calendar.get(Calendar.DATE);

System.out.println(year+"-"+month+"-"+day);

int h = calendar.get(Calendar.HOUR\_OF\_DAY);

int m = calendar.get(Calendar.MINUTE);

int s = calendar.get(Calendar.SECOND);

System.out.println(h+":"+m+":"+s);

//今天是周几？

int dow = calendar.get(Calendar.DAY\_OF\_WEEK);

String[] data = {"日","一","二","三","四","五","六"};

System.out.println("周"+data[dow-1]);

//查看今年一共多少天?

int d = calendar.getActualMaximum(Calendar.DAY\_OF\_YEAR);

System.out.println("今年共:"+d+"天");

d = calendar.getActualMaximum(Calendar.DAY\_OF\_MONTH);

System.out.println("当月共:"+d+"天");

案例7：void set(int field,int value) 设置指定的时间分量

Calendar calendar = Calendar.getInstance();

System.out.println(calendar.getTime());

/\*

\* 表示2008-08-08 20:08:08

\*/

// calendar.set(year, month, date, hourOfDay, minute, second);

//设置年

calendar.set(Calendar.YEAR, 2008);

System.out.println(calendar.getTime());

//设置月

calendar.set(Calendar.MONTH,Calendar.AUGUST);

System.out.println(calendar.getTime());

//设置天

calendar.set(Calendar.DATE, 8);

System.out.println(calendar.getTime());

calendar.set(Calendar.HOUR\_OF\_DAY, 20);

calendar.set(Calendar.MINUTE, 8);

calendar.set(Calendar.SECOND, 8);

System.out.println(calendar.getTime());

//设置为周三

calendar.set(Calendar.DAY\_OF\_WEEK,Calendar.WEDNESDAY);

System.out.println(calendar.getTime());

案例8：Calendar计算时间的操作: void add(int field,int amount)

对指定的时间分量加上给定的值,若给定的值为负数，则是

减去给定的时间。

Calendar calendar = Calendar.getInstance();

System.out.println(calendar.getTime());

/\*

\* 查看3年2个月零25天以后是哪天?

\*/

//加3年

calendar.add(Calendar.YEAR, 3);

System.out.println(calendar.getTime());

//加2个月

calendar.add(Calendar.MONTH, 2);

System.out.println(calendar.getTime());

//加25天

calendar.add(Calendar.DAY\_OF\_YEAR, 25);

System.out.println(calendar.getTime());

//减去7天

calendar.add(Calendar.DAY\_OF\_YEAR, -7);

System.out.println(calendar.getTime());

//那周的周三？

calendar.set(Calendar.DAY\_OF\_WEEK, 4);

System.out.println(calendar.getTime());

案例9：计算商品促销日期

输入一个商品的生产日期,格式:yyyy-MM-dd

再输入该商品的保质期，天数

经过程序的计算，输出该商品的促销日期，格式:yyyy-MM-dd

促销日期规则:该商品过期日前两周的周三

Scanner scanner = new Scanner(System.in);

System.out.println("请输入生产日期:");

String dateStr = scanner.nextLine();

System.out.println("请输入保质期天数:");

int days = Integer.parseInt(scanner.nextLine());

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

Date date = sdf.parse(dateStr);

Calendar calendar = Calendar.getInstance();

calendar.setTime(date);

//计算过期日

calendar.add(Calendar.DAY\_OF\_YEAR, days);

//计算两周前

calendar.add(Calendar.DAY\_OF\_YEAR, -14);

//设置为当周的周三

calendar.set(Calendar.DAY\_OF\_WEEK, 4);

date = calendar.getTime();

String line = sdf.format(date);

System.out.println("促销日为:"+line);

(十五)lambda表达式

JDK8推出的一个特性，可用来快速便捷的创建匿名内部类

语法:

([arg1,arg2...])->{

方法体

}

lambda表达式在创建一个匿名内部类时有一个要求，就是其对应接口必须有且 只能有一个抽象方法时才可以使用。如果存在多个方法时使用编译是不通过的。

案例1：

Runnable r1 = new Runnable() {

public void run() {

System.out.println("hello");

}

};

Runnable r2 = ()->{

System.out.println("hello");

};

/\*

\* 若重写的方法中只有一句代码，那么该方法的"{}"

\* 是可以省略的。

\*/

Runnable r3 = ()->System.out.println("hello");

案例2：方法含有参数

//获取当前目录下所有的文本文件

File file = new File(".");

// FileFilter filter = new FileFilter() {

// public boolean accept(File file) {

// return file.getName().endsWith(".txt");

// }

//

// };

/\*

\* 重写方法若含有参数，参数类型可以忽略。编译器

\* 会结合上下文判定参数类型。

\*/

// FileFilter filter = (f)->{

// return f.getName().endsWith(".txt");

// };

/\*

\* 若可以忽略方法的"{}"，那么当这个方法要求返回值

\* 时，return关键字也必须忽略。

\*/

FileFilter filter = (f)->f.getName().endsWith(".txt");

File[] subs = file.listFiles(filter);

for(File sub : subs) {

System.out.println(sub.getName());

}

案例3：1.8之后，集合和Map推出了为lambda表达式实现的遍历操作。 并且对于一个线程安全的集合或Map而言，使用这种遍历方式是可以 保证并发安全的。而以前使用迭代器则不能保证，需要自行维护并安全。

List<String> list = new ArrayList<String>();

list.add("one");

list.add("two");

list.add("three");

list.add("four");

for(String str : list) {

System.out.println(str);

}

//lambda形式

list.forEach((str)->System.out.println(str));

Map<String,Integer> map = new HashMap<String,Integer>();

map.put("语文", 99);

map.put("数学", 98);

map.put("英语", 97);

map.put("物理", 96);

map.put("化学", 99);

map.forEach((k,v)->System.out.println(k+":"+v));

**四、Web**

网站的架构:

- CS：Client Server 客户端和服务端

移动客户端：Android和iOS

电脑客户端：windows和linux、Mac OS

客户端是由客户端工程师开发

服务端功能：给客户端提供数据

应用场景： 京东、淘宝、唯品会

- BS：Browser Server 浏览器和服务端

浏览器

服务端功能： 给浏览器提供数据 和 页面

应用场景： 页游

前端课程介绍:

1. HTML ： 用来勾勒出页面的结构和内容(骨架)

2. CSS ： 用来美化页面

3. JavaScript： 让页面呈现动态效果和动态数据的

4. JQuery：用来简化JavaScript代码的

(一)HTML

1.html:Hyper Text Markup Language：超文本标记语言

什么是超文本：不仅仅是文本，还包括文本的字体颜色，样式 还包括多媒体（图 片，音频，视频）

2.html结构

<!DOCTYPE html>//文档声明 告诉浏览器使用html的哪个版本来解析 页面内容 ，此写法是告诉浏览器用最新的html5的版本解析

<html>//所有内容都在html标签内部 除了 文档声明

<head>//头 ：里面的内容是给浏览器看的 比如使用什么编码

</head>

<body>//体 ：里面的内容是给用户看的

</body>

</html>

3.head里面的标签

<meta charset="UTF-8">：告诉浏览器使用哪种字符集解析

<title>Insert title here</title> 告诉浏览器 页面标题是什么，title 还可以起到优化SEO的作用（让搜索引擎尽快搜索到此页面）

keywords的作用：让搜索引擎尽快的找到本页面

4.Body内部的标签

(1)文本标签

1. <h1></h1>.......<h6></h6> 内容标题

- align：left/center/right 水平对齐方式

2. <p></p> 段落标签 ，每个段落标签独占一行并且上下会留有 空白区域

3. <hr> 水平分割线

4. <br> 换行

(2)列表标签

1.无序列表：

<ul type="circle"> <!-- u:unordered 无序 l:list 列表 -->

<li>刘备</li><!--l:list列表 i:item 项 -->

<li>貂蝉</li>

<li>孙尚香</li>

<li>孙悟空</li>

</ul>

2.有序列表

<ol>

<li>打开冰箱门</li>

<li>把大象装进去</li>

<li>关上冰箱门</li>

<li>啦啦啦</li>

</ol>

常见属性： type 控制每个列表项前面的表示的（a A i I 1默认） start 列表数值的起始值 reversed 倒序

3.定义列表

<h3>定义列表</h3>

<dl><!-- defined list 定义一个列表 -->

<dt>凉菜</dt><!--defined title 定义标题 -->

<dd>老醋花生</dd><!--defined data 定义数据 -->

<dd>花毛一体</dd>

<dd>大拌菜</dd>

<dt>炒菜</dt>

<dd>宫保鸡丁</dd>

<dd>小炒肉</dd>

<dd>木须肉</dd>

</dl>

4.列表嵌套

有序和无序列表可以任意嵌套多层

5.分区标签（元素） div和span

(1)对页面进行分区有两大好处：

1. 可以多个页面复用同一个区域的内容

2. 可以对整个区域的内容进行统一管理

- 分区元素本身没有显示效果，起到容器的作用

- div： 块级分区 独占一行

- span： 行内分区 和多个行内分区元素 共占一行

- 通常一个页面会分为三大区域

<body>

<div>头部区域</div>

<div>正文区域</div>

<div>尾/脚部区域</div>

</body>

- 在html5中提出了几个分区元素 效果和div一样，但是代码可读性增强了

<body>

<header>头部区域</header>

<article>正文区域</article>

<footer>脚部区域</footer>

</body>

6.元素的分类

-块级元素：独占一行

包括： h1-h6, p,div,hr

- 行内元素：和多个行内元素共占一行

包括：span, <i></i>和 <em></em>（斜体）<b></b>和<strong></strong>(粗 体) <u></u>下划线 <s></s>和<del></del>删除线

7.行内元素空格折叠现象

行内元素中空格如果有多个只能识别一个空格

8.需要转义的常见特殊字符 实体引用

- 空格： &nbsp;

- < : &lt;

- > : &gt;

- 换行：<br>

9.图片元素

- <img>属于单标签

- 常见属性：

1. alt：当图片不能正常显示的时候显示此文本

2. src：写图片的路径，路径分为两种：

- 绝对路径：以http开头的路径，此路径可以指向本网站也可以指向其它网 (盗链，盗链图片如果原网站图片发生改变则不能正常显示)

- 相对路径：相对于当前页面html文件

1. 图片和页面文件在同一个文件夹下，直接写文件名
2. 图片在页面文件的上一层文件夹 ../文件名

3. 在页面文件的下一层文件夹 文件夹名/文件夹名/文件名

- 路径总结： 如果是站内资源使用相对路径，站外资源使用绝对路径

- 图床：专门存放图片的网站，使用图床可以节省本站空间

3. title:图片标题 当鼠标悬停在图片上面的时候显示

4. width：设置图片的宽度，两种方式：像素和百分比，如果只设置宽度 高 度会保证宽高比的情况下自动跟着宽度改变

5. height: 设置图片的高度，两种方式：像素和百分比。

- img支持的图片格式： jpg png gif

10.Map 图像地图

(1)什么是图像地图

可以将图片的某个区域作为可点击的链接

(2)使用方式：

<!-- usemap:使用地图 #代表当前页面 -->

<img alt="" src="../imgs/b.jpg" usemap="#mymap">

<!-- 为了保证兼容性问题 把name和id都写上 -->

<map name="mymap" id="mymap">

<!-- area区域 shape形状 rect矩形 coords坐标-->

<area shape="rect" coords="0,0,200,200"

href="../imgs/2.gif">

<area shape="circle" coords="663,473,90"

href="../imgs/d.jpg">

</map>

- map中的id和name作用是让图片能够找得到此地图

- area常见属性： shape表示形状 常用有circle和rect

- href：值为一个路径，可以写相对路径和绝对路径，路径可以指向页面， 也可以指向文件（浏览器可以打开的文件直接浏览，不能打开的 文件则下载）

11.超链接 a标签

- 如果a标签没有添加href属性 就相当于一个文本不是超链接

- href属性和图像地图中的href作用一样

- target="\_blank" 开启一个新的页面

- 锚点用法： 需要先创建锚点:<a id="a" name="a"></a>, 然后通过a标签 跳转到锚点的位置：<a href="#a">跳转到锚点</a>

- img标签放在a标签内部 则图片就可以被点击了

12.表格 table

- 接触标签有 table tr td

- table的属性：align水平对齐 width宽度 border边框 cellspacing边框 和单元格的距离 cellpadding内容和单元格的距离

- td的属性：colspan跨列 rowspan跨行 align水平对齐

- caption标签

- 分组标签：<thead> <tbody> <tfoot> 没有任何显示效果，提高代码可读性

13.表单

- 表单的作用是收集用户的数据提交到服务端

- 表单中的控件包括： 文本框、密码框、按钮、下拉列表、单选、复选、时 间选择、文件选择、文本域

(1)文本框

<input type="text" placeholder="占位文本" maxlength="最大字符长 度" readonly="只读" >

(2)密码框

<input type="password" maxlength>

(3)单选

<input type="radio" checked默认选中>

(4)多选、复选框

<input type="checkbox" id="aaa"> <label for="aaa">xxx</label>

(5)时间

<input type="date" >

(6)文件

<input type="file" >

(7)下拉选

<select name>

<option value selected></option>

<select>

(8)文本域

<textarea rows="行数" cols="列数"></textarea>

(9)隐藏域

<input type="hidden">

(10)按钮

<input type="submit/reset/button" value="按钮标题">

(二)CSS

- 美化页面的技术

- Cascading 层叠 Style 样式 Sheet 表 ： 层叠样式表，CSS用于美化html页 面，html可以理解成盖房子，CSS相当于装修

1.CSS的引入方式

1. 内联样式：在标签内部通过 style属性添加样式，弊端：只能作用于一个 元素，不能复用，几乎不使用

<div style="color: red;border: 1px solid green;">我是一个div</div>

<span style="color: purple;border: 1px solid pink">我是一个 span</span>

2. 内部样式：在head标签内部通过 style子标签添加样式，好处：可以在 本页面内复用样式，弊端：不能多个页面复用同一样式 ，工作中此种方式用 的不多。

<style type="text/css">

/\* 标签名选择器 \*/

h3{

color: blue;

border: 2px solid yellow;

}

</style>

3. 外部样式：样式代码写在单独的\*.css文件中，通过link标签引入到某个 html页面中，好处：样式可以复用在多个页面中

<link rel="stylesheet" href="demo01.css">

css文件：

@charset "UTF-8";

p{

color: red;

background-color: green;

}

2.三种引入方式的优先级

- 如果三种引入方式作用的样式不同则全部生效

- 如果三种引入方式作用的样式相同则按照下面的优先级规则来决定到底哪 个生效:

1. 内联优先级最高

2. 内部和外部 后执行会覆盖先执行的

3.CSS的选择器

(1)基础选择器

1. 标签名选择器

- 应用场景：当需要对页面中所有的某一种标签设置相同效果的时候

标签名{

样式名称:值;

样式名称:值;

}

2. id选择器

- 通过标签的id属性找到对应的标签

- 应用场景：当需要对页面中的某一个元素设置样式的时候使用

#标签id{

样式名称:值;

样式名称:值;

}

3. 类选择器

- 通过标签的class属性找到对应的标签

- 应用场景：当需要对页面中的多个元素设置样式的时候使用

.class{

样式名称:值;

样式名称:值;

}

(2)其它各种选择器

1. 属性选择器

- 通过标签内部指定属性查找对应的标签

标签名[属性名="属性值"]{

样式名称:值;

样式名称:值;

}

2. 派生选择器(后代选择器)

- 通过多个标签之间的上下级关系查找子元素和孙子元素和。。。。所有后 代元素

基础选择器 基础选择器 基础选择器{

样式名称:值;

样式名称:值;

}

3. 子元素选择器

- 通过多个标签之间的上下级关系查找子元素。

基础选择器>基础选择器>基础选择器{

样式名称:值;

样式名称:值;

}

4. 分组选择器

- 分组选择器可以把多种选择器通过逗号连接到一起，给多个选择器设置 相同的样式

h3,p,.d1,#abc,#bcd{

color:red;

}

5. 伪类选择器

- 用于选择元素的状态

1. visited：访问过的状态

2. link：未点击状态

3. hover：鼠标悬停状态

4. active：鼠标点击时的状态

- 使用方式：

/\* 访问过的状态 \*/

a:visited {

color: red;

}

/\* 未点击状态 \*/

a:link {

color: yellow;

}

/\* 鼠标悬停状态 \*/

a:hover {

color: blue;

}

/\* 鼠标点击时的状态 \*/

a:active {

color: pink;

}

6. 任意元素选择器

- 如果需要给页面中所有的元素设置相同的样式可以使用此选择器

\*{

样式名称:值;

样式名称:值;

}

注：如果多个选择器设置相同样式的效果时，作用范围越小优先级越高

4.CSS中常用的属性

已经接触的属性 color（字体颜色）background-color(背景颜色) border（边框）

5.元素的宽高：

1. 块级元素的默认高度为内容高度，默认宽度为父元素的宽度

2. 行内元素的默认宽高为内容的宽高

- 只能修改块级元素的宽高，不能修改行内元素的宽高

6.设置元素的颜色

/\*1. 通过颜色名称赋值 red yellow green

blue pink purple white black orange gray\*/

/\*2. 6位16进制赋值 每两位表示一个颜色 红 绿 蓝 \*/

/\* background-color: #ff00ff; \*/

/\*3. 3位16进制赋值 每一位表示一个颜色 \*/

/\* background-color: #00f; \*/

/\*4. 通过3位 10进制赋值 每个取值0-255 \*/

/\* background-color:rgb(255,255,0); \*/

/\*5. 4位10进制赋值 第四位代表透明度取值0-1 \*/

background-color: rgba(0,255,0,0.5);

7.设置元素的背景图片

/\* 设置背景图片 \*/

background-image: url("../imgs/a.jpg");

/\* 设置背景图片的大小 参数为宽和高 \*/

background-size: 100px 100px;

/\* 设置背景图片不重复 \*/

background-repeat: no-repeat;

/\* 设置背景图片的位置 left center right top bottom

第一个参数代表水平 第二个参数代表垂直\*/

/\* background-position: right center; \*/

/\* 通过百分比控制位置 \*/

background-position:90% 90%;

8.盒子模型

-网页设计中常听的属性名：内容(content)、内边距(padding)、边框(border)、 外边距(margin)， CSS盒子模式都具备这些属性。

-这些属性我们可以用日常生活中的常见事物——盒子作一个比喻来理解，所以 叫它盒子模式。

-CSS盒子模型就是在网页设计中经常用到的CSS技术所使用的一种思维模型。

- 外边距：指元素到父元素或相邻兄弟元素边框的距离 左上角原点

margin-left/top/right/bottom: 100px;

- 边框:指元素的边框

- 内边距:指元素内容距元素边框的距离

(1)外边距:元素距离父元素或相邻兄弟元素的距离称为外边距

赋值方式:

/\* margin: 40px; 四个外边距都是40px \*/

/\* margin: 20px 50px; 上下20 左右50 \*/

/\* margin: 0 auto; 水平居中 \*/

margin: 10px 30px 60px 90px;/\* 顺时针 上右下左 \*/

- 块级元素：上下左右都生效

- 行内元素：只有左右生效 上下不生效

- 外边距粘连问题： 当元素的上边缘和父元素的上边缘重合的时候，子元素 的上外边距会带着父元素一起移动，通过在父元素中添加 overflow:hidden; 解决。

- 上下外边距取最大值，左右外边距相加

(2)边框

border: 粗细(像素值) 边框的样式(solid) 颜色

/\* border: 10px inset purple; \*/

border-bottom:5px solid yellow;

border-left:5px solid pink;

border-top:5px solid green;

border-right:5px solid black;

border-radius:55px;/\*圆角 值越大越园 \*/

- 块级元素： 上下左右边框全部生效，而且边框都会占显示区域

- 行内元素： 上下左右边框全部生效，但是只有左右边框才会占显示区域， 上下不占显示区域（不会增加元素自身的高度）

(3)内边距

- 内容距离边框的距离

- 给元素添加内边距会影响元素的宽高

- 移动元素内的子元素几种方式？ 两种：1. 给大的添加内边距（大的所占 宽高会跟着改变） 2. 给小的添加外边距

- 移动元素内的文本的时候几种？ 一种：只能给大的添加内边距。

- 块级元素：内边距四个方向都生效，而且都会影响元素的显示范围

- 行内元素：内边距四个方向都生效，左右影响显示范围，上下不影响显示 范围

9.块级元素和行内元素关于盒子模型的总结

1. 块级元素：可以设置宽高，外边距全部生效 并且影响元素的所占宽高, 边 框四个方向都生效，并且影响元素的所占宽高，内边距全部生效，并且影响元 素所占的宽高

2. 行内元素：不可以设置宽高，外边距左右生效，上下不生效，边框都生效， 上下不影响所占高度，内边距都有效果，左右影响所占宽度 上下不影响所占 高度

10.元素所占宽度计算公式

宽高为100\*200 左外边距10 右外30 边框5px 左内8px 右内2px

左外边距+左边框+左内边距+元素宽度+右内边距+右边框+右外边距

11.文本相关的属性

/\* 文本相关属性 \*/

/\* 文本水平对齐方式 center/left/right \*/

text-align: center;

/\* 文本修饰 none 没有

underline 下划线

overline 上划线

line-through 删除线\*/

text-decoration: line-through;

/\* 文本颜色 \*/

color: red;

/\* 行高 通常让文本垂直居中的时候使用行高 \*/

line-height: 100px;

/\* 文本阴影 4个参数

1. 阴影颜色

2和3是阴影的偏移值

4. 阴影的模糊度 值越小越清晰 0是不模糊\*/

text-shadow: #0f0 -15px 15px 1px;

12.字体相关属性

/\* 字体相关属性 \*/

font-size: 20px;

/\* 字体粗细属性 normal 不加粗

bold 加粗

lighter 比普通细一些

\*/

font-weight: lighter;

/\* 字体样式 可以设置多个值通过逗号分隔，以此检测

哪个生效用哪个 \*/

font-family: "楷体";

/\* 设置斜体 \*/

font-style: italic;

13.溢出设置:指子元素显示范围超出父元素范围时的设置

/\* visible(默认 超出显示)

hidden: 超出不显示

scroll:超出滚动显示 \*/

overflow: hidden;

14.元素显示方式

- block： 块级元素默认的显示方式，独占一行，可以修改宽高

- inline：行内元素的默认显示方式，共占一行，不可以修改宽高

- inline-block:行内块，共占一行 并且可以修改宽高

15.元素的定位:position定位

(1)静态定位（文档流定位）（默认）

- 元素如果是块级元素则默认从上到下排列，如果是行内元素则从左向右排列

- 设置方式： position:static;

(2)相对定位

- 相对于元素初始的位置，通过left/right/top/bottom让元素做位置的偏移

- 元素不会脱离文档流

- 场景：当元素需要从当前位置做位置偏移的时候，并且不希望其它元素跟着 移动时使用此定位

(3)绝对定位

- 会脱离文档流 后面有元素的话会顶上来

- left/right/top/bottom 做位置偏移，相对于谁？ 1. 相对于窗口(没有祖 宗或有祖宗但是没有做非static定位) 2. 相对于祖宗元素(有祖宗并且有某 个祖宗做了非static定位，如果都做了，则就近原则)（如果仅仅是为了给子 元素做位置的参考一般都使用relative）

- 场景：当元素需要相对于某个祖先元素或窗口做位置偏移的时候，并且不再 占用原来的位置时使用此定位方式

(4)固定定位

- 固定在窗口的某个位置，不会随着内容位置改变而改变

- 脱离文档流，left/right/top/bottom 坐标相对于窗口

(5)浮动定位

- 元素脱离文档流，元素在当前所在行内 左侧或右侧浮动，当撞到父元素边 框或浮起来的兄弟元素边缘的时候

- 如果所有浮动的元素一行装不下会自动换行，但是如果有凸出的元素会有卡 住的效果

- 如果元素浮动则后面的元素会往上顶(因为浮动会脱离文档流)，如果不希望 后面的元素往上移动，则需要在后面紧邻的元素上添加clear属性设置 both/left/right,禁止元素 两边/左/右 有浮动的元素。

- 如果元素的所有子元素全部浮动，并且元素没有设置高度，此时自动识别的 高度为0，可以通过添加 overflow:hidden; 让元素能够识别出浮动后子元素 的高度

- 应用场景：当多个纵向排列的元素需要改成横向排列的时候 使用浮动定位

16.行内元素的垂直对齐方式

- 默认对齐方式为基线对齐 vertical-align：baseline

- top/middle/bottom

17.CSS的三大特性

1. 继承性：子元素可以继承父元素的部分属性，只能继承color/font-开头 /text-文本相关/line- 此类型的属性，不仅仅子元素继承所有的后代元素都 会继承

- 如果元素自身所带的样式和继承的冲突则以自身所带样式为准，如：h标签的 字体大小，超链接的文本颜色和文本下划线

2. 层叠性：通过不同的选择器可以多次指向同一个元素，如果设置的样式不同 则全部生效，如果多个选择器设置的样式相同，则根据CSS的第三特性优先级 来决定

3. 优先级：

- 直接选中优先级高于间接选中(继承属于间接选中)

- 如果相同选择器，则后执行的覆盖前面的

- 不同选择器，作用范围越小 优先级越高 id>class>标签名>继承

(三)JavaScript 简称JS

html搭建页面结构和内容 CSS用来美化页面 JS给页面添加动态效果或显示动 态内容

1.JS历史

1995年由网景（NetScape）公司发布 LiveScript 同年改名为 JavaScript，和Java 没啥关系主要为了蹭热度

2.JS特点

- 属于脚本语言 不需要编译 由浏览器解析执行

- JS可以嵌入到Html文件中

- JS是基于面向对象语言

- JS属于弱类型语言

3.JS优点

- 交互性： 可以直接和用户进行交互

- 安全性： JS语言只能访问浏览器内部内容，不能访问浏览器以外内容(指磁盘 中的各种数据)

4.如何在html文件中嵌入js代码

<!-- 1. 在元素的事件中添加js代码 -->

<input type="button" value="点我试试"

onclick="alert('第一种引入方式成功！')">

<!-- 2. 通过script标签添加js代码 可以写在任意位置-->

<script type="text/javascript">

alert("第二种引入方式");

</script>

<!-- 3.通过单独的\*.js文件写js代码 以下方式引入 引入文件后

不能在当前script标签体内写js代码 -->

<script type="text/javascript" src="first.js">

//这个位置不能再写js代码

</script>

5.JavaScript语法

(一)数据类型

1. 数值类型（number）

2. 字符串类型(string)

3. 布尔值类型(boolean)

4. undefined (未定义)

5. 对象类型 null

(二)变量的声明和赋值

因为js属于弱类型语言 声明变量的时候不需要指定类型，直接赋值即可

var x = 18;

var s = "abc";

s = 20;

var b = true/false;

var y;

如果变量只声明没有赋值 则此变量的类型为undefined 值也是undefined。

(三)数值类型

- 在js中所有的数值底层都是浮点型， 在使用过程中会自动转换整数或小数

int x = 5/2; x=?(在java中结果是2)

var x= 6/2; 3

var x = 5/2; 2.5

- NaN（Not a Number）：不是一个数 ，用来判断一个变量是否为数值类型 判 断方式通过isNaN()函数判断,当变量不是数的时候为true，是数为false

isNaN(20); false

isNaN("aaa"); true

(四)字符串

- 通过单引号或双引号修饰一个字符串，例如：var s1 = 'abc'; var s2="abc";

(五)布尔值

- 值为true和false

(六)undefined

此类型只有一个值为undefined，变量声明但未定义的话为undefined

(七)null

null为对象类型的值 var p = null;

6.js各种类型间的隐式转换

1. 数值类型

- 转字符串： 18->"18" "abc"+18 "abc18"

- 转布尔值： 0和NaN转false 其它所有数转true

2. 字符串

- 转数字： "18"->18 "abc"->NaN "18abc"->18

- 转布尔值: ""空字符串转成false 其它都是true

3. 布尔值

- 转数字： true->1 false->0

- 转字符串：true->"true" false->"false"

4. undefined

- 转数字 NaN 21+undefined= NaN NaN和任何数值进行运算都得NaN

- 转布尔值 undefined->false

- 转字符串 undefined->"undefined"

5. null

- 转数字 null->0

- 转布尔值 null->false

- 转字符串 null->"null" "abc"+null = "abcnull"

7.运算符

- js中运算符和Java中的基本一致

- ==和===：==会在比较两端变量数值之前先将变量类型统一，统一后再比较 值.===会先比较类型是否相等如果相等再比较数值如果不等就直接false

"666"==666 true "666"===666 false

- typeof 获取变量的类型 typeof 234234+6 = "number6"

- 除法运算会自动根据结果转换整数还是小数

8.语句

包括：if else while do while for switch case

- 条件如果不是布尔类型会自动转换成布尔类型

- for 中 把int改成var， js中不支持增强for循环(foreach)

for(var i=0;i<10;i++){

}

9.函数(方法)

public String run(String name,int age){

return "abc";

}

- 格式一：

function 函数名(参数列表){

return 18;

}

例如：

function fun(name,age){

return "abc";

}

- 格式二：

var 函数名 = function(参数列表){

return "abc";

}

10.和页面元素相关的函数

1. 得到页面元素对象的函数

var x = document.getElementById("id");

2. 获取value值或给value赋值

var z = x.value; x.value="abc";

3. 给元素添加文本内容

x.innerText="abc";

4. 给元素添加html代码

x.innerHTML="<h1>abc</h1>";

11.JS中的API

1.字符串强转成数值

- parseFloat: 把字符串转成整数或小数

- parseInt: 把字符串/小数 转成整数

- Number：等效parseFloat

2.String常用Api

1. 字符串创建的方式

var str1 = "abc";

var str2 = new String("abc");

2. 转大写和转小写

toUpperCase() toLowerCase();

3. 查找字符串出现的位置

str.indexOf('a')获取a在str中第一次出现的位置

str.lastIndexOf("a")获取z在str中最后一次出现的位置

4. 截取字符串

str.substring(start,[end]); //end可以省略

5. 替换字符串

str.replace(old,new);

6. 拆分字符串

str.split(":") a:b:c [a,b,c]

3.数值函数

四舍五入 num.toFixed(2);//参数表示保留几位小数

4.数组相关函数

1. 创建数组

var arr1 = ["刘备",50,true];

var arr2 = new Array();//空数组

2. 数组中添加元素

arr.push("小明");

arr.push(18);

3. 获取数组中的数据

arr[0];

4.数组长度

arr.length = 3;

5.反转

arr.reverse() 1,2,3 3,2,1

6.排序

arr.sort() 通过unicode编码排序

7.自定义排序

function mysort(a,b){return a-b/b-a};

5.日期相关

- var d = new Date();

- var d = new Date("2011/10/22 18:00:00");

- d.getFullYear()

- d.getMonth()

- d.getDate()

- d.getDay()

- d.getHours()

- d.getMinutes()

- d.getSeconds()

- d.toLocaleDateString() 年月日

- d.toLocaleTimeString() 时分秒

6.正则

- 创建方式两种

^(开头) $(结尾)

g(全局查找) m(多行查找) i(忽略大小写)

var reg = /^a/i;//忽略大小写，匹配以a开头的字符串

var reg = new RegExp("^a","i");

- 相关函数：

1. reg.exec(str) 查找内容

2. reg.test(str) 校验文本

3. str.match(reg) 查找内容 得到装着所有内容的数组

4. str.replace(reg,"新内容")

12.事件的取消

- 在元素的事件中如果执行 return false 则取消当前事件

<a href="http://tmooc.cn" onclick="return false">Tmooc</a>

13.失去焦点事件

- 失去焦点以文本框为例，当光标离开的时候，此事件触发

<input type="text" onblur="">

14.控制元素的隐藏和显示

1. display:block/inline/inline-block/none(隐藏 脱离文档流)

2. visibility：hidden(隐藏 不脱离文档流)/visible（显示）

15.自定义对象

/\* 1.通过声明函数的形式声明一个对象 \*/

function Person(name,age){

//声明属性

this.name=name;

this.age=age;

//声明方法

this.run=function(){

alert("name="+this.name+" age="+this.age);

}

}

//创建对象

var p = new Person("貂蝉",85);

p.run();

/\* 2.声明空对象 \*/

function SuperMan(){

}

//创建对象

var sman = new SuperMan();

//添加属性

sman.name="克拉克";

sman.age=18;

//添加方法

sman.fly = function(){

alert("我叫"+this.name+"今年"

+this.age+"岁，我在飞");

}

//调用方法

sman.fly();

/\* 第二种声明对象的方式 \*/

/\* 以声明变量的形式创建对象 \*/

var p = {

"name":"吕布",

"age":25,

"run":function(){

alert("name"+this.name+" age="+this.age);

}

}

//调用对象的方法

p.run();

(四)DHTML

- 简介： Dynamic(动态) HTML，不是一门新技术，把HTML，CSS，JavaScript 整合到一起得到的一门动态页面技术

- DHTML包括： BOM Browser浏览器 Object对象 Model模型 和 DOM Document 文档 Object对象 Model模型

- BOM： 浏览器对象模型，包括一些和浏览器相关的对象和函数

- DOM： 文档对象模型，包括一些和页面相关的对象和函数

BOM的相关内容

1.window

window中所有的属性和函数，调用时可以把window省略掉，window中的 属性称为全局属性，函数称为全局函数

- window的全局函数:

parseFloat/parseInt()

alert();//弹出框

confirm();//确认框

prompt();//弹出文本框

isNaN();

- window的全局属性:

1. location（地址）：

location.href 可以获取和设置浏览器当前的显示地址

location.reload() 页面刷新

2. history（历史）：

history.length 得到当前窗口访问过的页面数量

history.forward() //访问下一个页面

history.back() // 访问上一个页面

history.go(num) 前往某个页面 0代表当前页面，1代表下个页 面，2代表下两个页面 -1 代表上一个 -2 代表上两个页面

3. screen（屏幕）：

width/height: 获取屏幕的分辨率

availWidth/availHeight：获取屏幕可用分辨率

4. navigator（导航、帮助）

navigator.userAgent 获取浏览器的版本信息

2.window相关的事件:

1. 点击事件

2. 加载完成事件

3. 获取到焦点事件

4. 失去焦点事件

/\* 窗口点击事件 \*/

/\* onclick = function(){

//alert("窗口被点击了");

location.href="http://tmooc.cn";

} \*/

/\* 页面加载完成时执行 \*/

onload = function(){

var input = document.getElementById("ip1");

input.value="abc";

}

/\* 获取和失去焦点事件 \*/

onblur = function(){

console.log("失去焦点");

}

onfocus = function(){

console.log("得到焦点");

}

3.eval()函数

可以将字符串以js代码的形式执行

4.通过js给元素添加css样式

div.style.css样式属性名称 = "属性的值";

5.定时器

- 通过定时器可以实现 每隔一段时间执行某些代码

- var id = setInterval(函数,3000);//每隔3秒执行函数

- clearInterval(id);//停止定时器

- setTimeout(函数,2000);//两秒后执行函数

(五)DOM

- DOM 文档对象模型，学习dom主要掌握如何对html中的标签进行增删改查操作

1.查找元素

- 通过id查找 document.getElementById("id");

- 通过标签名查找 document.getElementsByTagName("标签名");

- 通过元素的name属性查找 document.getElementsByName("name属性的值");

- 通过元素的class属性查找 document.getElementsByClassName("class属性的 值");

- 获取body元素 document.body

2.通过层级关系查找元素

1. 获取元素的上级元素 parentElement;

2. 获取元素的下级内容（包括元素和文本内容和换行） childNodes

3. 获取元素的兄弟 x.parentElement.childNodes;

3.创建元素

var h1 = document.createElement("h1");

4.添加元素到某个元素里面

父元素.appendChild(新元素);

5.插入元素到某个元素的上面

父元素.insertBefore(新元素,弟弟元素);

6.删除元素

父元素.removeChild(被删除的元素);

7.事件分类

1. 鼠标事件： onclick(点击) onmouseover(鼠标移动到元素上面) onmouseout(鼠标移出元素) onmousedown(在元素上鼠标按下) nmouseup(在元素上鼠标松手) onmousemove(在元素上移动)

2. 键盘事件： onkeydown（键盘按下） onkeyup(键盘松开)

3. 状态改变事件：onchange(value值发生改变的事件) onload（页面加载完 成事件） onfocus(获得焦点事件) onblur(失去焦点事件) onsubmit(表单提 交事件) onresize(窗口尺寸改变事件)

8.事件绑定

1. 在元素标签中添加

<input type="text" onkeydown="downfn()">

- 此种方式添加事件 ，事件函数中的this代表的是window 如果需要用到事 件对应的元素可以在标签中调用函数的时候传递过去一个this 此时的this 代表元素自身

2. 动态绑定事件

/\* 动态绑定事件 \*/

/\* mybtn.onclick = function(){

alert("动态绑定成功！");

}

\*/

mybtn.onclick = myfn;

- 动态绑定的事件，事件函数中的this代表事件对应的元素

9.Event对象

- 通过Event对象可以得到和事件相关的信息

1. 可以通过event得到鼠标点击屏幕的坐标 event.clientX event.clientY

2. 可以通过event得到键盘按键的编码 event.keyCode

3. 通过event获得事件源 （事件源指添加事件的元素）

/\* 通过event得到事件源 存在兼容性问题

有的浏览器支持target 有的支持srcElement

通过以下写法可以保证任何浏览器都能正常执行\*/

var obj = event.target||event.srcElement;

alert("标签名称："+obj.nodeName);

10.事件传递（事件冒泡）

- 如果同一个区域有多个事件响应，响应的顺序类似气泡，从下往上，也就是 从底层元素往上级元素执行，这个过程称为事件传递

- 如果页面中有多个元素需要添加事件，可以把事件添加在它们共同的上级元 素上面，然后获得事件源，通过判断事件源来处理不同的事件

(六)JQuery

1.什么是JQuery

- 是一个通过JavaScript代码写的 关于JavaScript的框架

- 用来简化js代码

- 可以像css一样通过选择器获取元素

- 可以快速修改元素的样式

2.如何引入jQuery

- 因为jQuery就是一个js文件 所以引入方式和引入一个普通的js文件一样

<script type="text/javascript"

src="../js/jquery-1.4.2.min.js"></script>

3.jQuery对象和js对象互相转换

1. 把js转成jq

/\* 得到js对象 \*/

var input = document.getElementById("myinput");

/\* 把js对象转成jQuery对象 \*/

/\* $是jQuery单词的缩写 $=jQuery \*/

var $jq = $(input);

/\* 弹出文本框的value值 \*/

2. 把jq转成js

/\* 得到jq对象 \*/

var $jq = $("#myinput");

/\* 把jq转成js对象 \*/

//第一种写法

var js1 = $jq[0];

//第二种写法

var js2 = $jq.get(0);

alert(js1.value+"="+js2.value);

4.选择器

(一)基础选择器

1. 标签名选择器

var $div = $("div");

2. id选择器

var $div = $("#id");

3. 类选择器

var $div = $(".class");

4. 分组选择器

var $div = $("div,#id,.class");

5. 任意元素选择器

var $div = $("\*");

(二)其它选择器

1.层级选择器

1. $("div span") 匹配div下所有的span元素

2. $("div>span") 匹配div的所有span子元素

3. $("div+span") 匹配div相邻的后面的兄弟span元素

4. $("div~span") 匹配div后面所有的兄弟span元素

层级函数:

当得到某个元素对象后 需要获得该元素的相关元素的时候使用

1. 获取元素的所有兄弟元素

$("#abc").siblings("div");

2. 获取元素的哥哥元素

$("#abc").prev("div");

3. 获取元素的哥哥们元素

$("#abc").prevAll("div");

4. 获取元素的弟弟元素

$("#abc").next("div");

5. 获取元素的弟弟们元素

$("#abc").nextAll("div");

2.过滤选择器

1. $("div:first")匹配所有div中的第一个

2. $("div:last")匹配所有div中的最后一个

3. $("div:even")匹配所有div中的偶数 从0开始

4. $("div:odd")匹配所有div中的奇数 从0开始

5. $("div:eq(n)") 匹配所有div中下标等于n的 从0开始

6. $("div:lt(n)") 匹配所有div中下标小于n的 从0开始

7. $("div:gt(n)") 匹配所有div中下标大于n的 从0开始

8. $("div:not(.abc)" 匹配所有div中class值不等于abc

3.内容选择器

1. $("div:has(p)") 匹配所有包含p标签的div

2. $("div:empty") 匹配所有空的div

3. $("div:parent")匹配所有非空的div

4. $("div:contains('abc')") 匹配包含abc文本内容的div

4.可见选择器

1. $("div:hidden") 匹配所有隐藏的div

2. $("div:visible")匹配所有可见的div

显示隐藏相关的函数:

1. 让隐藏的元素显示

$("div:hidden").show();

2. 让显示的元素隐藏

$("div:visible").hide();

3. 切换隐藏和显示状态

$("#abc").toggle();

5.属性选择器

1. $("div[id]") 匹配所有有id属性的div元素

2. $("div[id='abc']") 匹配有id属性值为abc的所有div

3. $("div[id!='abc']") 匹配有id属性值不为abc的所有div

6.子元素选择器

1. $("div:nth-child(n)") 获取div中第n个子元素 从1开始

2. $("div:first-child") 获取div中第1个子元素

3. $("div:last-child") 获取div中最后一个子元素

7.表单选择器

1. $(":input") 匹配所有 文本框 密码框 单选 多选 下拉选 文本 域 按钮

2. $(":password") 匹配所有密码框

3. $(":radio")匹配所有单选

4. $(":checkbox")匹配所有多选（复选框）

5. $(":checked")匹配所有被选中的 单选、多选、下拉选

6. $("input:checked") 匹配所有被选中的 单选、多选

7. $(":selected") 匹配被选中的下拉选

5.和页面(DOM)相关

1. 创建元素

var div = document.createElement("div");

var $div = $("<div>abc</div>");

2. 添加到页面中

$("#big").append($div); //添加到最后面

$("#big").prepend($div);//添加到最前面

3. 插入元素

兄弟元素.after(新元素) 加在兄弟元素的后面

兄弟元素.before(新元素) 加在兄弟元素的前面

//添加到二哥前面 before

$("h1:contains('二')").before($h1);

//添加到二哥的后面 after

$("h1:contains('二')").after($h1);

4. 删除元素

- 通过自己删除 $("#id").remove();

- 先匹配到所有的指定类型元素 从里面删除某些元素 $("div").remove("#abc");

5. 修改元素的样式css

$("div").css("color","red");

6. 属性

$("#abc").attr("id"); 获取id的属性值

$("#abc").attr("class","aaa"); 给元素添加class属性值为aaa

7. 文本

$("#abc").text() 获取文本内容

$("#abc").text("啊啊啊") 赋值文本内容

8. html

$("#abc").html() 获取文本内容

$("#abc").html("<h1>啊啊啊</h1>") 赋值文本内容

9.获得元素的子元素和父元素

- children() 得到所有子元素

- parent() 得到父元素

6.常见事件

1. 鼠标相关： click 点击 mouseover mouseout mousedown mouseup mousemove

2. 键盘： keydown keyup

3. 状态： $(function(){}) focus blur submit resize窗口尺寸改变 change当value值发生改变时的事件

7.事件模拟

//开启定时器

setInterval(function(){

//模拟事件

$("input").trigger("click");

},3000);

8.hover事件合并函数

- hover(fn1,fn2); 等效 mouseover + mouseout

$("div").hover(function(){

$("div").css("color","red");

},function(){

$("div").css("color","green");

})

9.动画相关

function fn1(){//隐藏

$("img").hide(3000);

}

function fn2(){//显示

$("img").show(3000,function(){

alert("啦啦啦");

});

}

function fn3(){//滑出

$("img").slideUp(2000);

}

function fn4(){//滑入

$("img").slideDown(2000);

}

function fn5(){//淡入

$("img").fadeIn(2000);

}

function fn6(){//淡出

$("img").fadeOut(2000);

}

function fn7(){//自定义

$("img").animate({"left":"250px"},1000)

.animate({"top":"250px"},1000)

.animate({"left":"0px"},1000)

.animate({"top":"0px"},1000)

.fadeOut(2000,function(){

$(this).remove();

});

}

10.音频视频播放

<!-- autoplay自动播放 loop循环播放 controls显示播放面板-->

<!-- <audio src="music.mp3" autoplay="autoplay"

loop="loop" controls="controls"></audio> -->

<!-- poster视频封面 -->

<video poster="../imgs/k.jpg" controls="controls">

<source src="hero.mp4">

</video>

**五、MySQL**

(一)数据库

1.io流文件存储数据的弊端

1. 存储效率低

2. 不管存还取操作比较麻烦

3. 一般只能保存小量字符串数据

2.DB

- data base 数据库

- 什么是数据库： 数据库实际上就是一个文件集合

- 数据库就是一个存储数据的仓库，本质就是一个文件系统，数据库是按照特 定的格式把数据存储起来，用户可以对存储的数据进行增删改查操作

3.DBMS

- DataBaseManagementSystem：数据库管理系统(软件)

- 什么是DBMS： 用来管理数据库文件的软件，用户需要访问DBMS对数据进 行增删改查操作。

- 常见DBMS： MySQL oracle DB2 sqlite sqlserver

4.数据库的分类

分为两类：

1. 关系型数据库： 经过数学理论验证 可以保存现实生活中的各种关系数据， 数据库中存储数据以表为单位。

2. 非关系型数据库：通常用来解决某些特定的需求如：数据缓存，高并发访 问。存储数据的形式有多种，举例：Redis数据库：通过键值对的形式存 储数据。

5.主流关系型数据库

1. MySQL： 08年被sun公司收购 09年sun被oracle收购，开源免费，到 oracle发布了5.0版本（使用了oracle核心技术 性能提高30%），因为 Oracle数据库地位受到威胁，计划把mysql闭源，原mysql作者们不干了， 原程序员出去单干 发布了MariaDB数据库 名字是因为作者女儿叫 Maria ，市场排名第一

2. Oracle： 闭源， 最贵 性能最高 拉里埃里森32 市场排名第二

3. SQLServer：微软公司的产品 市场排名第三，主要应用在.net(C#)开发的 网站中

4. DB2： IBM公司 有做网站的完整解决方案（操作系统，we服务器（如tomcat）， 数据库等）主要应用在银行等国有大型企业中

5. sqlite：轻量级数据库 只有几十k，一般应用在嵌入式和移动设备中

6.开源和闭源的区别

1. 开源： 开发源代码，盈利方式靠卖服务（赚钱慢，少），用户群大，大牛 程序员会无偿帮着升级维护

2. 闭源：不开放源代码，盈利方式卖产品+服务（赚钱快，多），大牛们各种 攻击找漏洞

7.数据库的安装

- 两点注意： 端口号默认3306，字符集选择utf-8

(二)SQL

Structured Query Language:结构化 查询 语言

- sql是写在客户端 执行在DBMS中，DBMS解析后执行

1.数据库相关的SQL

1. 查看所有数据库

show databases;

2. 创建数据库

-格式：create database 数据库名称;

create database db1;

3. 查看数据库详情

-格式： show create database 数据库名;

show create database db1;

4. 创建数据库指定字符集 utf8或gbk

create database db2 character set utf8;

create database db2 character set gbk;

5. 删除数据库

drop database db2;

6. 使用数据库

use db1;

2.和表相关的SQL

- 什么是表： 关系型数据库中保存数据的单元，类似于Excel中的表，创建 表时需要指定字段信息

1. 创建表

-格式： create table 表名 (字段1名 字段1类型,字段2名 字段2类型);

create table person(name varchar(10),age int);

2.查询所有表

show tables;

3.查看单个表属性

show create table person;

4.表的引擎

1. Myisam： 只支持数据基础的增删改查，不支持高级操作，如：事务、 外键等

2. InnoDB：支持高级操作，默认为InnoDB。

5.创建表并且指定引擎和字符集

create table t1(id int,name varchar(10)) engine=myisam charset=gbk;

-查询：show create table t1;

6.查看表字段信息

desc student;

7.修改表名

rename table 原名 to 新名；

rename table student to t\_stu;

create table hero(name varchar(10));

8.修改表属性 引擎和字符集

alter table hero engine=myisam charset=gbk;

9.添加表字段

- 最后位置添加

alter table hero add age int;

- 最前面添加

alter table hero add money int first;

- 在某个字段的后面添加

alter table hero add gender varchar(5) after name;

10.删除表字段

alter table hero drop money;

11.修改表字段名和类型

alter table hero change 原字段名 新字段名 类型;

alter table hero change name heroname varchar(10);

12.修改表字段类型和位置

alter table hero modify age int first;

alter table hero modify age int after xxx;

13.删除表

drop table hero;

3.数据相关的SQL

1.插入数据

- 创建学生表

create table student(id int,name varchar(10),chinese int,math int,english int);

- 插入数据 全表插入：每个字段都赋值 顺序要和表字段一致

insert into student values(1,'zhangsan',88,38,98);

- 指定字段插入

insert into student (id,name) values (2,'张飞');

- 批量插入

insert into student values(6,'唐僧',56,57,58),(7,'八戒 ',88,89,99);

insert into student (id,name) values

(8,'吕布'),(9,'貂蝉'),(10,'孙尚香');

2.查询

-查询全部数据的全部字段信息

select \* from student;

-查询指定字段

select id,name from student;

-条件查询

select \* from student where id<5;

3.修改数据

update student set math=100;

update student set english=30 where id=7;

4.删除数据

delete from student where id=7;

delete from student;

4.主键约束

1.什么是主键

用来表示数据唯一性的字段称为主键

2.约束

约束就是对表字段的限制条件

3.主键约束

限制主键字段的数据唯一并且非空

create table t1(id int primary key,name varchar(10));

insert into t1 values(1,'Tom');//成功

insert into t1 values(1,'Jerry');//失败 id不能重复

insert into t1 values(null,'Jack');//失败 id不能为null

5.自增： auto\_increment

create table t2(id int primary key auto\_increment,name varchar(10));

insert into t2 values(null,'Lucy');

insert into t2 values(null,'Hanmeimei');

insert into t2 values(10,'LiLei');

insert into t2 values(null,'Lily');

1. 自增会在曾经出现的最大值的基础上+1

2. 自增数值不会因为删除数据减少

3. 使用delete删除整个表的数据时 自增数值不会清零

6.注释 comment

- 创建表声明字段的时候可以通过添加comment给字段添加注释

create table t3(id int primary key auto\_increment comment '这是 主键字段',age int comment '这是年龄字段');

show create table t3;

7.`和' 的区别

- ` : 用来修饰表名和字段名的 可以省略

create table `t4`(`id` int,`name` varchar(10));

- ' : 用来修饰字符串

8.数据冗余

- 如果表设计不够合理随着数据量增多出现了大量的重复数据称为数据冗余， 通过拆分表的形式解决冗余问题

9.事务

数据库中执行sql语句的工作单元，此工作单元不可拆分，能够保证全部成功 或全部失败

create table person(id int primary key auto\_increment,name varchar(10),money int);

insert into person values(null,'超人',50);

insert into person values(null,'钢铁侠',30000);

update person set money=5050 where id=1;

update person set money=25000 where id=2;

10.查看数据库自动提交的状态

show variables like '%autocommit%';

11.设置自动提交的开关 0：关闭 1：开启

set autocommit=0/1;

12.验证过程

- 保存回滚点： savepoint s1;

- 回滚到某个回滚点： rollback to s1;

create table person(id int primary key auto\_increment,name varchar(10),money int);

insert into person values(null,'超人',50);

insert into person values(null,'钢铁侠',30000);

1. update person set money=5050 where id=1;

2. 先在当前窗口执行select \* from person 查看是否改变，结果是修改了 因为查看的是内存中的数据

3. 此时打开另外一个终端，先使用db2 然后select \* from person 查询是 否改变，结果是没有改变因为此时查看的是数据库文件中的数据，没提交就不 会改变

4. 回到原窗口再执行转账的另一部分

update person set money=25000 where id=2;

5. 此时原窗口查询数据因为查询的是内存中的数据所以改变，但是另外一个 窗口数据并没变化

6. 在原窗口中执行commit，此时 两个窗口的数据都发生改变

- 事务回滚： rollback, 此指令会将数据库中的数据回滚到上次提交的点

步骤：1. 在关闭了自动提交的窗口 修改超人的钱 修改后查询 由于查询到 的是内存中的数据所以数据是修改过的，此时执行rollback，则数据会回滚 到修改前的值

13.事务总结

执行sql语句的工作单元，保证全部成功或全部失败

1. 查看自动提交状态 show variables like '%autocommit%';

2. 设置自动提交状态 set autocommit=0/1;

3. 手动提交 commit;

4. 回滚 rollback;

5. 保存回滚点 savepoint s1;

6. 回滚到某个点 rollback to s1;

14.truncate

- 删除表并且创建一个相同的空表，此时表中的自增数值清零

- 格式： truncate table 表名;

- truncate：删除表创建新表 drop：删除表 delete：只删除数据

- 效率：drop>truncate>delete

15.is null 和 is not null

1. 查询没有奖金的员工信息

select \* from emp where comm is null;

2. 查询有奖金的员工信息

select \* from emp where comm is not null;

16.别名

select ename from emp;

select ename as '名字' from emp;

select ename '名字' from emp;

select ename 名字 from emp;

17.去重 distinct

- 去掉重复的数据

select distinct job from emp;

18.比较运算符 > < >= <= = !=和<>

1. 查询工资小于等于1600的员工姓名和工资

select ename,sal from emp where sal<=1600;

2. 查询部门编号是20的员工姓名，职位job和部门编号

select ename,job,deptno from emp where deptno=20;

3. 查询职位是manager的员工姓名和职位

select ename,job from emp where job='manager';

4. 查询部门不是10号部门的所有员工姓名和部门编号，两种方式实现

select ename,deptno from emp where deptno!=10;

select ename,deptno from emp where deptno<>10;

5. 查询商品表t\_item 单价等于23的商品信息

select \* from t\_item where price=23;

6. 查询商品表中价格不等于8443的商品信息

select \* from t\_item where price!=8443;

19.and 和 or

- and相当于 java中的 &&

- or相当于java中的||

1. 查询工资小于2000并且是10号部门的员工信息

select \* from emp where sal<2000 and deptno=10;

2. 查询有奖金comm 或者 工资大于3000的员工姓名，工资，奖金

select ename,sal,comm from emp where comm is not null or sal>3000;

3. 查询t\_item表中单价price大于500并且库存num大于100的商品信息

select \* from t\_item where price>500 and num>100;

20.in

- 当查询某个字段的值等于多个值的时候使用in关键字

查询工资为800，3000，5000，1600的所有员工信息

select \* from emp where sal=800 or sal=3000 or sal=5000 or sal=1600;

select \* from emp where sal in (800,3000,5000,1600);

21.between x and y 包含x和y

- 查询某个字段的值在两个数值之间的时候使用

查询员工工资在2000至4000之间的员工信息

select \* from emp where sal between 2000 and 4000;

22.like

- 用来进行模糊查找

- \_： 代表单个未知字符

- %： 代表0或多个未知字符

- 举例：

1. 以a开头的内容 a%

2. 以a结尾的内容 %a

3. 两个字符第一个是a a\_

4. 第二个字符是a \_a%

5. 倒数第三个字符是a %a\_\_

6. 包含a %a%

案例：

1. 查询所有记事本的价格（title中包含记事本）

select title,price from t\_item where title like '%记事本%';

2. 查询单价低于100的记事本信息

select \* from t\_item where price<100 and title like '%记事本%';

3. 查询单价在50到200之间的得力商品（title包含得力）

select \* from t\_item where price between 50 and 200 and title like '%得力%';

4. 查询有图片的得力商品

select \* from t\_item where image is not null and title like '% 得力%';

5. 查询有赠品的商品信息（sell\_point字段包含 赠）

select \* from t\_item where sell\_point like '%赠%';

6. 查询不包含得力的商品标题

select title from t\_item where title not like '%得力%';

7. 查询价格介于50到200之外的记事本信息

select \* from t\_item where price not between 50 and 200 and title ike '%记事本%';

23.排序 order by

- order by写在where的后面如果没有where 写在表名的后面

- by后面写排序的字段名称

- 默认是升序， 也可指定升序降序： asc 升序 desc降序

- 案例：

1. 查询员工姓名和工资 按照工资降序排序

select ename,sal from emp order by sal desc;

2. 查询所有的dell商品按照价格的降序排序

select \* from t\_item where title like '%dell%' order by price desc;

3. 查询所有员工的姓名，工资，部门编号 按照部门编号升序排序

select ename,sal,deptno from emp order by deptno;

4. 查询所有员工的姓名，工资，部门编号 按照部门编号升序排序如果部门一 致则按照工资降序排序

select ename,sal,deptno from emp order by deptno,sal desc;

24.分页查询 limit

- limit 跳过条数,每页条数

- limit 通常写在sql语句的最后面

1. 查询所有商品 按照单价升序排序 查询第二页 每页7条数据

select \* from t\_item order by price limit 7,7;

2. 查询工资排名前三名的三位员工的信息

select \* from emp order by sal desc limit 0,3;

3. 查询拿最低工资的员工姓名和编号

select ename,empno from emp order by sal limit 0,1;

4. 查询商品表中的记事本，第三页 每页两条数据；

select \* from t\_item where title like '%记事本%' limit 4,2;

25.数值计算 + - \* / % mod(7,2)等效 7%2

1. 查询员工姓名，工资，年终奖 （年终奖=月薪\*5）

select ename,sal,sal\*5 年终奖 from emp;

2. 查询商品单价，库存和总金额（单价\*库存）

select price,num,price\*num 总金额 from t\_item;

3. 修改所有员工的工资 每人涨薪20块钱

update emp set sal=sal+20;

26.日期相关函数

SQL的helloworld select 'helloworld';

- 获得当前时间 年月日 时分秒 now()

select now();

- 获得当前年月日 current

select curdate();

- 获得当前时分秒

select curtime();

- 从年月日时分秒中提取年月日

select date(now());

- 从年月日时分秒中提取时分秒

select time(now());

- 从年月日时分秒中提取时间分量 年，月，日，时，分，秒

select extract(year from now());

select extract(month from now());

select extract(day from now());

select extract(hour from now());

select extract(minute from now());

select extract(second from now());

- 日期格式化 date\_format(日期,格式);

四位年 %Y 两位年 %y

两位月 %m 一位月 %c

日 %d

24小时 %H 12小时 %h

分钟 %i

秒 %s

1. 把now() 格式改成 年月日 时分秒

select date\_format(now(),'%Y年%m月%d日 %H时%i分%s秒');

- 把非标准格式转回标准格式 str\_to\_date(非标准格式的字符串,格式);

1. 把 14.08.2018 09:10:20 转回标准格式

select str\_to\_date('14.08.2018 09:10:20','%d.%m.%Y %H:%i:%s');

27.ifnull函数

- age=ifnull(x,y) 如果x的值为null则age=y,如果不为null则age=x;

1. 修改员工表的奖金，如果奖金值为null修改成0不为null则不变

update emp set comm=ifnull(comm,0);

28.聚合函数

- 对多行数据进行统计： 求和 求平均值 最大值 最小值 统计数量

1.求和 sum(字段名)

1. 查询10号部门工资总和

select sum(sal) from emp where deptno=10;

2. 查询单价在100以内的商品库存总量

select sum(num) from t\_item where price<100;

2.平均值 avg(字段名)

1. 查询20号部门的平均工资

select avg(sal) from emp where deptno=20;

2. 查询记事本的平均价格

select avg(price) from t\_item where title like '%记事本%';

3.最大值 max(字段名) 最小值 min(最小值)

1. 查询所有员工的最大工资和最小工资

select max(sal),min(sal) from emp;

4.统计数量 count(字段名) 一般使用count(\*)

1. 统计 工资在2000以下的员工有多少人

select count(\*) from emp where sal<2000;

29.数学相关函数

- 向下取整 floor()

select floor(3.14);

- 四舍五入 round()

select round(3.8);

- 四舍五入指定小数位数

select round(3.2358,2);

- 非舍五入 truncate()

select truncate(3.2358,2);

- 随机数 rand() 0-1 不包含1

select rand();

0-8的随机数

select floor(rand()\*9);

3-5的随机数

0-2 +3

select floor(rand()\*3)+3;

30.分组查询

1. 查询每个部门的最高工资

select deptno,max(sal) from emp group by deptno;

2. 查询每个职位的平均工资

select job,avg(sal) from emp

group by job;

3. 查询每个领导手下的人数

select mgr,count(\*) from emp

where mgr is not null

group by mgr;

- group by SQL中的位置

select \* from 表名 where .... group by... order by....limit..;

4. 查询每个部门的每个领导下的人数

select deptno,mgr,count(\*) from emp

where mgr is not null

group by deptno,mgr;

31.having

1. 查询每个部门的平均工资，要求查询平均工资大于2000

- 以下是错误写法： where后面不能写聚合函数

select deptno,avg(sal) a from emp

where a>2000

group by deptno;

- 正确写法：

select deptno,avg(sal) a from emp

group by deptno

having a>2000;

- having关键字要和group by结合使用，写在group by的后面，用于在SQL 语句中添加聚合函数的条件

- where后面写普通字段条件， having后面写聚合函数的条件

1. 查询emp表中每个部门的平均工资高于2000的部门编号，部门人数，部门 平均工资，最后根据平均工资排序降序

select deptno,count(\*),avg(sal) a from emp

group by deptno

having a>2000

order by a desc;

2. 查询商品表中每个分类的平均单价，要求显示平均单价低于100的信息

select category\_id,avg(price) a from t\_item

group by category\_id

having a<100;

3. 查询每个分类对应的库存总量，显示高于199999的库存总量

select category\_id,sum(num) s from t\_item

group by category\_id

having s>199999;

4. 查询emp表中 工资在1000-3000之间的员工，每个部门的编号，工资总和， 平均工资，过滤掉平均工资低于2000的部门信息，按照平均工资升序排序

select deptno,sum(sal),avg(sal) a from emp

where sal between 1000 and 3000

group by deptno

having a>=2000

order by a;

5. 查询emp表中名字不是以s开头，每个职位的人数，工资总和，最高工资， 过滤掉平均工资高于3000的职位，根据人数升序排序，如果一致则根据工资 总和降序排序

select job,count(\*) c,sum(sal) s,max(sal) from emp

where ename not like 's%'

group by job

having avg(sal)<=3000

order by c,s desc;

6. （提高题） 查询每年入职的人数

select extract(year from hiredate) year,count(\*) from emp

group by year;

32.子查询(嵌套查询)

查询emp表中工资最高的员工信息

select max(sal) from emp;

select \* from emp where sal=5000;

- 将以上两条合并成一条sql

select \* from emp where sal=(select max(sal) from emp);

33.子查询总结

1. 嵌套在SQL语句中的查询语句称为子查询

2. 子查询能嵌套n层

3. 子查询可写的位置：

- 可以写在 where/having的后面作为查询条件的值

- 可以写在 from后面 当一张新表 \*\*新表必须有别名\*\*

- 可以写在创建表的时候

create table t\_emp\_10 as (select \* from emp where deptno=10);

34.关联查询

- 同时查询多张表的数据称为关联查询

1. 查询每一个员工的姓名和对应的部门名称

select e.ename,d.dname

from emp e,dept d

where e.deptno=d.deptno;

2. 查询在纽约工作的所有员工的信息

select e.\*

from emp e,dept d

where e.deptno=d.deptno and d.loc='new york';

35.笛卡尔积

- 关联查询如果不写关联关系，则查询结果为两张表的乘积，这个乘积称为 笛 卡尔积

- 笛卡尔积是一种错误的查询结果，工作中切记不要出现

36.等值连接和内连接

1. 等值连接：

select \* from A,B

where A.x=B.x and A.age=18

2. 内连接 用的更多

select \* from A join B

on A.x=B.x

where A.age=18;

- 查询每个员工的姓名和对应的部门名称

select e.ename,d.dname

from emp e join dept d

on e.deptno=d.deptno;

37.外连接

- 左外连接： 以join 左边表为主表 左边表显示所有数据右边交集数据

select e.ename,d.dname

from emp e left join dept d

on e.deptno=d.deptno;

- 右外连接： 以join 右边表为主表 右边表显示所有数据左边交集数据

select e.ename,d.dname

from emp e right join dept d

on e.deptno=d.deptno;

38.关联查询总结

等值连接，内连接，外连接都是关联查询的查询方式使用哪一种取决于具体业 务需求

1. 查两个表的交集数据 使用内连接(推荐)或等值连接

2. 查一个表所有数据另外一个表交集数据使用外连接

(三)SQL分类

1.DDL：Data Definition Language（数据定义语言）

- 包括：create,drop,alter,truncate

- 不支持事务

2.DML：Data Manipulation Language（数据操作语言）

- 包括：insert update delete select(DQL)

- 支持事务

3.DQL：Data Query Language（数据查询语言）

- 只包括select

4.TCL：Transaction Control Language（事务控制语言）

- 包括：commit rollback savepoint rollback to

5.DCL：Data Control Language（数据控制语言）

- 分配用户权限相关的sql

(四)数据库的数据类型

1.整型

- 常用类型：int(m) bigint(m)，m代表显示长度

create table t\_int(num int(10) zerofill);

insert into t\_int values(123);

select \* from t\_int;

注：zerofill为自动补0，当某列添加了此属性，mysql会为该列自动添加 unsigned属性

比如插入的类型为int(4)，值是1，显示的是0001，当插入负数时

显示0000

2.浮点型

- 常用类型： double(m,d) m代表总长度 d代表小数长度 76.232 m=5 d=3, decimal(m,d):超高精度小数，当需要涉及超高精度运算的时候使用此类型

3.字符串

- char：固定长度 char(10) 'abc' 占10 ，执行效率高，最大255

- varchar:长度可变 varchar(10) 'abc' 占3 ，节省空间，最大65535 超 高255建议使用text

- text:长度可变 最大值65535 用于保存大文本

4.日期

- date:只保存年月日

- time:只保存时分秒

- datetime:保存年月日时分秒，默认值为null，最大值 9999-12-31

- timestamp:保存年月日时分秒 保存距离1970年1月1日八点的毫秒数，默 认值为当前时间，最大值2038年1月19号

create table t\_date(d1 date,d2 time,d3 datetime,d4 timestamp);

insert into t\_date values('2018-04-23',null,null,null);

insert into t\_date values

(null,'12:38:45','2018-05-12 12:38:33',null);

5.字符串相关函数

1. 字符串拼接 concat(s1,s2) s1s2

select concat('abc','mm');

- 查询员工姓名 和工资 在工资后面添加 元 字

select ename,concat(sal,'元') from emp;

2. 获取字符串的长度 char\_length('abc') 3

select char\_length('abc');

- 查询员工姓名和姓名的长度

select ename,char\_length(ename) from emp;

3. 获取一个字符串在另一个字符串中出现的位置

- 格式： instr('abcdefg','d') 从1开始

select instr('abcdefg','d');

- 格式： locate('d','abcdefg')

select locate('d','abcdefg');

4. 插入字符串

- 格式: insert(str,start,length,newstr);

select insert('abcdefg',3,2,'m');

5. 转大小写

- 格式： upper(str) lower(str)

select upper('Nba'),lower('CBA');

6. 从左边截取和从右边截取

- 格式： left(str,count) right(str,count)

select left('abcdefg',2);

select right('abcdefg',2);

7. 截取字符串

- 格式： substring(str,start) substring(str,start,count)

select substring('abcdefg',2);

select substring('abcdefg',2,3);

8. 去字符串两端的空白

- 格式： trim(str);

select trim(' a b ');

9. 重复

- 格式： repeat(str,count)

select repeat('ab',2);

10. 替换

- 格式： replace(str,old,new)

select replace('abcdefg','c','m');

11. 反转

- 格式： reverse(str)

select reverse('abc');

(五)表设计之关联关系

1.一对一

AB两张表，A表中的一条数据对应B表中的一条数据，同时B表的一条数据对 应A表中的一条数据，此时两张表的关系称为一对一关系。

2.应用场景

商品表和商品详情表， 用户表和用户信息扩展表

主键是表示数据唯一性的 外键是用来建立关系的

3.如何让两张表建立关系

在从表中添加一个外键指向主表的主键。

4.案例：

创建user表和userinfo表

user: id int, username varchar(10) password varchar(10)

userinfo:nick varchar(10),qq varchar(10),phone varchar(15),uid int

保存以下数据：

1 libai admin 李白 66668888 13838383388

2 liubei 123456 刘备 3334444 1333333333

- 创建表：

create table user(id int primary key auto\_increment,username varchar(10),password varchar(10));

create table userinfo(nick varchar(10),qq varchar(10),phone varchar(15),uid int);

insert into user values (null,'libai','admin'),(null,'liubei','123456');

insert into userinfo values('李白','66668888','13838383388',1),('刘 备','3334444','1333333333',2);

5.一对多

AB两张表 A表中的一条数据对应B表中的多条，同时B表中的一条数据对应 A表中的一条，此时两张表的关系为一对多。

6.应用场景

部门表和员工表， 分类表和商品表， 用户表和收货地址表

7.如何建立关系

在两张表中多的表中添加外键指向另外一张表的主键。

8.案例:创建员工表和部门表 emp dept 创建db5 utf8 并使用

- 保存以下数据：

保存盘丝洞的妖怪部门的白骨精 年龄28 工资3000 蜘蛛精 年龄32 工 资2000

美国的英雄部门 钢铁侠 年龄45 工资8888 美国队长 年龄252 工资 6000

日本的海贼部门 路飞 年龄18 工资100 娜美 年龄20 工资500

create table emp(id int primary key auto\_increment,name varchar(10),age int,sal int,deptid int);

create table dept(id int primary key auto\_increment,name varchar(10), loc varchar(10));

insert into dept values(null,'妖怪部','盘丝洞'),(null,'英雄部',' 美国'),(null,'海贼部','日本');

insert into emp values(null,'白骨精',28,3000,1),(null,'蜘蛛精 ',32,2000,1),(null,'钢铁侠',45,8888,2),(null,'美国队长 ',252,6000,2),(null,'路飞',18,100,3),(null,'娜美',20,500,3);

9.多对多

AB两张表：A表中一条数据对应B表中多条数据，同时B表中一条数据对应A 表中的多条数据，称为多对多

10.应用场景：老师表和学生表

11.如何建立关系

创建中间关系表 在关系表中添加两个外键指向两个表的主键

12.案例

1. 创建两张表主表 teacher（id,name） 和 student(id,name)

create table teacher(id int primary key auto\_increment,name varchar(10));

create table student(id int primary key auto\_increment,name varchar(10));

2. 创建一个关系表 t\_s(tid,sid)

create table t\_s(tid int,sid int);

3. 插入以下数据

传奇老师：刘德华 张学友

苍老师：刘德华 张学友 小明 小红

insert into teacher values(null,'传奇老师'),(null,'苍老师');

insert into student values(null,'刘德华'),(null,'张学友 '),(null,'小明'),(null,'小红');

insert into t\_s values(1,1),(1,2),(2,1),(2,2),(2,3),(2,4);

4. 查询每个学生对应的老师姓名

select s.name,t.name

from student s join t\_s ts

on s.id=ts.sid

join teacher t

on t.id=ts.tid;

5. 查询刘德华的老师姓名

select t.name

from student s join t\_s ts

on s.id=ts.sid

join teacher t

on t.id=ts.tid

where s.name='刘德华';

6. 查询苍老师的学生信息

select s.name

from student s join t\_s ts

on s.id=ts.sid

join teacher t

on t.id=ts.tid

where t.name='苍老师';

13.自关联：当前表的外键指向自己表的主键这种称为自关联

14.应用场景

用于保存有层级关系，并且不确定有多少层的数据，如：员工和上级领导， 部 门和上级部门，分类和上级分类。

可以保存一对一或一对多的数据

15.连接方式和关联关系的区别

- 连接方式： 包括等值连接、内连接、外连接，是指关联查询的查询方式

- 关联关系： 指表设计时两张表之间存在的逻辑关系包括 一对一，一对多和 多对多

16.表设计案例：权限管理

1. 创建三张主表，用户表user:id,name 角色表role:id,name 权限表 module:id,name

create table user(id int primary key auto\_increment,name varchar(10));

create table role(id int primary key auto\_increment,name varchar(10));

create table module(id int primary key auto\_increment,name varchar(10));

2. 创建两张关系表： 用户-角色关系表 u\_r:uid,rid 角色-权限关系表 r\_m:rid,mid

create table u\_r(uid int, rid int);

create table r\_m(rid int, mid int);

3. 用户表插入：刘德华、张学友、凤姐

insert into user (name) values('刘德华'),('张学友'),('凤姐');

4. 权限表插入： 男浏览、男评论、男发帖、男删帖，女浏览、女评论、女发 帖、女删帖

insert into module (name) values('男浏览'),('男评论'),('男发帖 '),('男删帖'),('女浏览'),('女评论'),('女发帖'),('女删帖');

5. 角色表插入：男会员、男管理员、女游客、女会员

insert into role (name) values ('男会员'),('男管理员'),('女游客'),(' 女会员');

6. 角色和权限 保存以下关系 男会员对应权限：男浏览，男评论，男发帖； 男 管理员对应权限：男浏览、男评论、男发帖、男删帖；女游客对应权限：女浏 览；女会员对应权限：女浏览，女评论，女发帖

insert into r\_m values(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(2,4),(3,5),(4,5),(4,6),( 4,7);

7. 用户和角色 保存以下关系 刘德华 男会员，张学友男管理员，凤姐女会员 和男会员.

insert into u\_r values(1,1),(2,2),(3,4),(3,1);

8. 查询刘德华所拥有的权限名称

- 1. 子查询

- 得到刘德华的id

select id from user where name='刘德华'

- 通过用户id 得到对应的角色id

select rid from u\_r where uid=(select id from user where name=' 刘德华');

- 通过角色id找到对应的权限id

select mid from r\_m where rid in (select rid from u\_r where uid=(select id from user where name='刘德华'));

- 通过权限id 得到权限的名字

select name from module where id in (select mid from r\_m where rid in (select rid from u\_r where uid=(select id from user where name='刘德华')));

- 2. 内连接

select m.name

from user u join u\_r ur

on u.id=ur.uid

join r\_m rm

on rm.rid=ur.rid

join module m

on rm.mid=m.id

where u.name='刘德华';

9. 查询拥有男浏览权限的用户有哪些

select u.name

from user u join u\_r ur

on u.id=ur.uid

join r\_m rm

on rm.rid=ur.rid

join module m

on rm.mid=m.id

where m.name='男浏览';

10. 查询每个用户拥有的权限名称

select u.name,m.name

from user u join u\_r ur

on u.id=ur.uid

join r\_m rm

on rm.rid=ur.rid

join module m

on rm.mid=m.id;

(六)视图

1.什么是视图

数据库中存在的表和视图都是其内部的对象，视图可以理解成是一个虚拟的 表，数据来自原表，视图本质上就是取代了一段sql语句

2.为什么使用视图

因为有些数据查询的SQL语句比较长，每次书写比较麻烦，使用视图可以起到 sql语句重用的作用，提高开发效率，可以隐藏敏感信息

3.格式

create view 视图名 as 子查询;

-创建10号部门员工的视图

create view v\_emp\_10 as (select \* from emp where deptno=10);

-创建没有工资的员工的信息

create view v\_emp\_nosal as (select ename,job,deptno,mgr from emp);

4.视图分类

- 简单视图： 创建视图的子查询中 不包含去重、函数、分组、关联查询的视 图称为简单视图，简单视图可以对数据进行增删改查操作

- 复杂视图： 包含去重、函数、分组、关联查询的视图称为复杂视图，一般只进行查询操作。

5.对视图中的数据进行增删改查

- 视图的操作方式和表的方式一样

1. 插入数据

insert into v\_emp\_10 (empno,ename,deptno) values (10001,'钢铁侠 ',10);//原表中有 视图中有

insert into v\_emp\_10 (empno,ename,deptno) values (10002,'葫芦娃 ',20);//原表中有 视图中没有

- 往视图中插入一条 视图中不可见但是在原表中存在的数据 称为数据污染

- 解决数据污染： 在创建视图的时候添加 with check option 关键字

create view v\_emp\_30 as (select \* from emp where deptno=30) with check option;

insert into v\_emp\_30 (empno,ename,deptno) values (10003,'蜘蛛侠 ',20);//报错

insert into v\_emp\_30 (empno,ename,deptno) values (10004,'嘻哈侠 ',30);//成功

2. 修改数据 只能修改视图中存在的数据

update v\_emp\_30 set ename='煎饼侠' where empno=10004;

update v\_emp\_30 set ename='葫芦娃' where empno=10001;

3. 删除数据 只能删除视图中存在的数据

delete from v\_emp\_30 where empno=10004;//视图中存在，成功

delete from v\_emp\_30 where empno=10001;//视图中不存在，不成功

6.修改视图

- 格式： create or replace view 视图名 as 子查询;

create or replace view v\_emp\_20 as (select \* from emp);

7.删除视图

drop view v\_emp\_20;

8.视图别名

- 如果创建视图的时候使用了别名 则后面的各种操作只能使用别名

create view v\_emp\_20 as (select ename name,sal from emp);

update v\_emp\_20 set name='aaaa' where name='钢铁侠';//成功！

update v\_emp\_20 set ename='aaaa' where ename='钢铁侠';//失败！

(七)约束

1.什么是约束

约束是给表字段添加的限制条件

2.种类

(1)非空约束 not null

- 限制字段的值不能为null

create table t1(id int,age int not null);

insert into t1 values (1,20);//成功！

insert into t1 values (2,null);//失败

(2)唯一约束 unique

- 限制字段的值不能重复

create table t2(id int,age int unique);

insert into t2 values(1,20);//成功！

insert into t2 values(2,20);//失败

(3)主键约束 primary key

- 限制字段值唯一并且非空

- 创建表时添加：

create table t\_pri(id int primary key auto\_increment);

- 创建表后添加主键约束：

create table t\_pri2(id int);

alter table t\_pri2 add primary key (id);

- 删除主键约束

alter table t\_pri2 drop primary key;

desc t\_pri2;

(4)默认约束 default

- 给字段添加默认值，当字段不赋值的时候 此约束的值生效

create table t3(id int,age int default 20);

insert into t3 values (1,88);

insert into t3 (id) values (2);

insert into t3 values (3,null);

(5)检查约束 check mysql中没有效果 但是语法不报错

create table t4(id int,age int check(age>10));

(6)外键约束

1. 字段的值可以为null，可以重复，但是不能是不存在的值

2. 被依赖的数据不能先删除

3. 被依赖的表不能先删除

- 测试外键约束

1.创建部门表

create table dept(id int primary key auto\_increment,name varchar(10));

create table emp(id int primary key auto\_increment,name varchar(10),deptid int,constraint fk\_dept foreign key(deptid) references dept(id));

- 格式： 在创建表最后一个字段后面添加 constraint 约束名 foreign key (外键字段名称) references 被依赖的表名(被依赖的 字段名)

2.插入数据

insert into dept values(null,'神仙'),(null,'妖怪');

-测试：

insert into emp values(null,'悟空',1); //成功！

insert into emp values(null,'赛亚人',3);//失败！不存在的数据

delete from dept where id=1;//失败 被关联的数据

(八)索引

1.什么是索引

索引是数据库中用来提高查询效率的技术，类似于目录。

2.为什么使用索引

如果不使用索引，查询数据时会依次遍历每一个保存数据的磁盘块，直到找到 目标数据为止，使用索引后，磁盘块会以树状结构保存，查询数据时会大大降 低磁盘块的访问量，从而提高查询效率。

3.索引是越多越好吗？

不是。因为索引会占用储存空间，只对常用的查询字段创建索引。

4.有索引就一定好吗？

如果数据量小的话 添加索引反而会降低查询效率。

5.索引的分类

1. 聚集索引（聚簇索引）： 通过主键创建的索引称为聚集索引，一个表只能 有一个聚集索引，添加了主键约束的表会自动创建聚集索引，聚集索引的树状 结构中保存了数据

2. 非聚集索引： 通过非主键字段创建的索引称为非聚集索引，一个表可以有 多个，树状结构中不保存数据只保存指针（磁盘块地址）

6.如何创建索引

- 格式： create index 索引名 on 表名(字段名(长度));

-给title字段添加索引

create index index\_item2\_title on item2(title);

-再次查询看时间多少？ 0.04秒 提高近30倍

select \* from item2 where title='100';

7.查看索引

show index from item2;

8.删除索引

-格式：drop index 索引名称 on 表名;

drop index index\_item2\_title on item2;

9.复合索引

- 通过多个字段创建的索引称为复合索引

- 应用场景：频繁使用某几个字段作为查询条件的时候，可以为这几个字段创 建复合索引

- 创建 标题和价格的索引：

create index index\_item2\_title\_price on item2(title,price);

10.索引总结

1. 索引是提高查询效率的技术

2. 索引占存储空间不是越多越好

3. 数据量小时使用索引会降低效率

4. 聚集索引：只有一个 添加主键约束后自动创建，保存数据

5. 非聚集索引：多个 非主键字段创建 没有数据只保存指针

6. 尽量不要在频繁修改的表上面添加索引

11.事务

数据库中执行SQL语句的工作单元，此工作单元不可拆分，可以保证同一业务 中的所有SQL语句全部成功或全部失败

12.事务的ACID特性

- Atomicity：原子性，最小不可拆分，保证全部成功，全部失败

- Consistency：一致性，从一个一致状态到另一个一致状态

- Isolation：隔离性， 多个事务之间互不影响

- Durability：持久性，事务完成后数据提交到数据库文件中 持久保存

13.事务相关指令

show variables like '%autocommit%';

set autocommit=0/1;

commit;

rollback;

savepoint s1;

rollback to s1;

(九)JDBC

1.什么是JDBC

Java DataBase Connectivity,java数据库连接，实际上jdbc是java中的一 套和数据库交互的api（application program interface 应用程序编程接口）

2.为什么使用JDBC

- 因为java程序员需要连接多种数据库(orcale,mysql,db2等)为了避免每一 个数据库都学习一套新的api，sun公司提出了一个jdbc接口，各个数据库厂 商根据此接口写实现类（驱动）,这样java程序员只需要掌握JDBC接口中的 一套方法，就可以访问任何数据库。

3.如何使用JDBC

1. 创建maven工程

2. 下载mysql相关jar包

登陆达内的私服：maven.tedu.cn

登陆阿里私服：maven.aliyun.com

- 首页搜索栏输入mysql 搜索

- 找到5.1.6版本 复制坐标 到自己工程的pom.xml文件里

3. 创建JDBCDemo1.java类 添加main方法

4. 使用步骤：

- 注册驱动

- 获取连接对象

- 创建sql执行对象

- 执行sql语句

- 关闭资源

//1.注册驱动

Class.forName("com.mysql.jdbc.Driver");

//2.获取连接对象

Connection conn =

DriverManager.getConnection(

"jdbc:mysql://localhost:3306/db3",

"root", "root");

//3.创建sql执行对象

Statement stat = conn.createStatement();

//4.执行sql

stat.execute("drop table if exists jdbct2");

System.out.println("删除成功！");

//5. 关闭资源

stat.close();

conn.close();

4.执行sql的方法

- execute(sql) 可以执行任意sql，但是推荐执行DDL

返回值为boolean值 true代表有结果集 false代表没有结果集，成功或 失败通过是否有异常来判断

- executeUpdate(sql) 推荐执行DML

返回值为int 代表生效行数

- executeQuery(sql)推荐执行DQL

insert into jdbct1 values(2,'Jerry'),(3,'李白');

返回值ResultSet 里面装了查询结果

next()方法的作用：判断有没有下一条数据有返回值为true（同时游标 往下移动） 没有则false

//4.执行sql

ResultSet rs =

stat.executeQuery("select \* from jdbct1");

//遍历结果

while(rs.next()){

int id = rs.getInt("id");

String name = rs.getString("name");

System.out.println(id+":"+name);

}

5.数据库类型和Java类型对比

mysql java

int getInt

varchar getString

float/double getFloat/getDouble

datetime/timestamp getDate

6.ResultSet获取数据的方式

1. 通过字段名获取 如：getString("name")

2. 通过查询到结果数据的位置获取 如：getString(2); 位置从1开始

7.读取配置文件

//创建读取配置文件的对象

Properties prop = new Properties();

//获取文件输入流

InputStream ips =

DBUtils.class.getClassLoader()

.getResourceAsStream("jdbc.properties");

//把文件加载到对象中

try {

prop.load(ips);

//读取连接数据库的信息

driver = prop.getProperty("driver");

url = prop.getProperty("url");

username = prop.getProperty("username");

password = prop.getProperty("password");

} catch (IOException e) {

e.printStackTrace();

}

8.数据库连接池

DBCP DataBase Connection Pool:数据库连接池

1.为什么使用

如果没有连接池，每一次业务都需要和数据库服务器建立一次连接，业务 处理完断开连接，如果有上万次业务就会有上万次的开关连接，频繁开关 连接非常浪费资源，使用数据库连接池，可以设置几个初始连接，如果有 业务需要使用连接，则从连接池中直接获取，如果连接池中连接用光，则 会等待连接归还后再获取连接

2.如何使用

1. 下载jar包： 从maven私服中搜索 dbcp

2. 代码：

//1.创建数据源对象

BasicDataSource dataSource = new BasicDataSource();

//2.设置数据库连接信息

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/db3");

dataSource.setUsername("root");

dataSource.setPassword("root");

//3. 设置连接池策略信息

dataSource.setInitialSize(3);//初始连接数量

dataSource.setMaxActive(5);//最大连接数量

//4.从连接池中获取连接对象

Connection conn = dataSource.getConnection();

System.out.println(conn);

3.解决jdbc乱码问题

在url后面添加如下内容

jdbc:mysql://localhost:3306/db3?useUnicode=true&characterEncodi ng=UTF-8

9.Properties属性配置文件

//创建属性对象

Properties prop = new Properties();

//得到文件输入流 通过反射方式获得

InputStream ips = Demo03.class.getClassLoader()

.getResourceAsStream("jdbc.properties");

//把文件加载到属性对象中

prop.load(ips);

//获取属性配置文件中的数据

String age = prop.getProperty("age");

String name = prop.getProperty("name");

System.out.println(age+"-"+name);

10.PreparedStatement

- 好处：

1. 相比较Statement 代码结构整齐，可读性高。

2. 可以避免SQL注入，因为在编译的时候已经把SQL的逻辑固定，替换？时 如果有修改逻辑的语句如or 则不会生效

3. 如果执行大量的SQL语句 preparedstatement 执行效率略高于statement

- 如果SQL中有变量则使用preparedStatement，没有变量则可以使用 statement

11.批量操作

案例1：

public class Demo06 {

public static void main(String[] args) {

Connection conn = null;

Statement stat = null;

ResultSet rs = null;

try {

conn = DBUtils.getConn();

stat = conn.createStatement();

String sql1 = "insert into user values(null,'aaa','111')";

String sql2 = "insert into user values(null,'bbb','222')";

String sql3 = "insert into user values(null,'ccc','333')";

//添加到批量操作

stat.addBatch(sql1);

stat.addBatch(sql2);

stat.addBatch(sql3);

//执行批量操作 返回值为每条sql语句的生效行数

int[] arr = stat.executeBatch();

System.out.println("执行完成！");

} catch (Exception e) {

e.printStackTrace();

} finally {

DBUtils.close(conn, stat, rs);

}

}

}

案例2：

public class Demo07 {

public static void main(String[] args) {

Connection conn = null;

PreparedStatement stat = null;

ResultSet rs = null;

try {

conn = DBUtils.getConn();

String sql = "insert into user values(null,?,?)";

stat = conn.prepareStatement(sql);

for (int i = 0; i < 100; i++) {

stat.setString(1, "name"+i);

stat.setString(2, "password"+i);

//添加到批量操作

stat.addBatch();

//每隔20条执行一次

if(i%20==0) {

//执行批量操作

stat.executeBatch();

//清空

stat.clearBatch();

}

}

//执行批量操作

stat.executeBatch();

System.out.println("执行完成！");

} catch (Exception e) {

e.printStackTrace();

} finally {

DBUtils.close(conn, stat, rs);

}

}

}

12.分页查询

案例：

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("请输入查询的页数");

int page = sc.nextInt();

System.out.println("请输入每页的条数");

int count = sc.nextInt();

Connection conn = null;

PreparedStatement stat = null;

ResultSet rs = null;

try {

conn = DBUtils.getConn();

String sql = "select \* from user limit ?,?";

stat = conn.prepareStatement(sql);

stat.setInt(1, (page-1)\*count);

stat.setInt(2, count);

rs = stat.executeQuery();

while(rs.next()) {

String name = rs.getString("username");

System.out.println(name);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

DBUtils.close(conn, stat, rs);

}

}

13.事务

public static void main(String[] args) {

//事务相关

// 超人 500 钢铁侠 5000

//创建person表 id name money

// create table person

// (id int,name varchar(10),money int);

// insert into person values(1,'超人',500),

// (2,'钢铁侠',5000);

Connection conn = null;

Statement stat = null;

ResultSet rs = null;

try {

conn = DBUtils.getConn();

//关闭自动提交

conn.setAutoCommit(false);

stat = conn.createStatement();

String sql1 = "update person set money=money+2000 "

+ "where id=1";

String sql2 = "update person set money=money-2000 "

+ "where id=2";

stat.executeUpdate(sql1);

stat.executeUpdate(sql2);

//查询钢铁侠剩余的钱是否>=0

rs = stat.executeQuery

("select money from person where id=2");

while(rs.next()) {

//得到查询的钱数

int money = rs.getInt(1);

if (money>=0) {

System.out.println("转账成功！");

conn.commit();//提交

}else {

System.out.println("转账失败！");

conn.rollback();//回滚

}

}

} catch (Exception e) {

e.printStackTrace();

} finally {

DBUtils.close(conn, stat, rs);

}

}

14.获取自增主键值

案例：

public static void main(String[] args) {

Connection conn = null;

Statement stat = null;

ResultSet rs = null;

try {

conn = DBUtils.getConn();

stat = conn.createStatement();

String sql =

"insert into user values(null,'Tom','abcd')";

//执行sql 并且设置需要返回生成的主键值

stat.executeUpdate(sql,

Statement.RETURN\_GENERATED\_KEYS);

//获取自增主键值

rs = stat.getGeneratedKeys();

while(rs.next()) {

//因为得到的数据只有一个 所以写1

int id = rs.getInt(1);

System.out.println(id);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

DBUtils.close(conn, stat, rs);

}

}

15.元数据

- 数据库元数据: 一些和数据库相关的信息如：数据库版本信息，数据库厂商等

- 表元数据: 一些和表相关的信息：表字段的数量，字段名称，字段类型等

案例：

public static void main(String[] args) {

Connection conn = null;

Statement stat = null;

ResultSet rs = null;

try {

conn = DBUtils.getConn();

//得到数据库元数据

DatabaseMetaData data = conn.getMetaData();

System.out.println(data.getDatabaseProductName());

System.out.println(data.getDatabaseProductVersion());

stat = conn.createStatement();

String sql = "select \* from emp";

rs = stat.executeQuery(sql);

//得到表的元数据

ResultSetMetaData rsData = rs.getMetaData();

//得到表的字段数量

int count = rsData.getColumnCount();

//得到表的字段名称和类型

for (int i = 0; i < count; i++) {

System.out.println("字段名："

+rsData.getColumnName(i+1));

System.out.println("字段类型:"

+rsData.getColumnTypeName(i+1));

}

} catch (Exception e) {

e.printStackTrace();

} finally {

DBUtils.close(conn, stat, rs);

}

}

**六、Servlet**

1.什么是Servlet?

sun(oracle)公司制订的一种用来扩展web服务器功能的组件规范。

(1)用来扩展web服务器功能

web服务器(比如apache web server,iis,nginx等)只能够处理静态资源的请 求，(即需要事先将html文档准备好，并存放到web服务器上)，不能够处理 动态资源的请求(即需要通过一些计算，生成相应的html文档)。所以，需要 扩展。可以使用Servlet来扩展。

(2)组件和容器

### 1)什么是组件?

符合规范，具有部分功能，并且需要部署到相应的容器里面才能运行的软件模块。

Servlet就是一个组件。

### 2)什么是容器?

符合规范，为组件提供运行环境的程序。

Servlet需要部署到Servlet容器里面才能运行，Servlet容器为

Servlet提供相应的运行环境（即网络相关的服务）。

注：

Tomcat是一个符合Servlet规范的容器。

2.如何写一个Servlet?

step1.写一个java类，实现Servlet接口或者继承HttpServlet。

step2.编译。

step3.打包。

注：按照Servlet规范，创建一个具有如下结构的文件夹

appname (文件夹的名称可以自定义，一般称之为应用名)

WEB-INF

classes (放.class文件)

lib (放.jar文件　该文件夹可选)

web.xml (部署描述文件)

step4.部署。

将step3创建好的文件夹拷贝到容器相应的位置。

注:

也可以将step3创建好的文件夹使用jar命令压缩成".war"为后缀的文 件，然后再拷贝。

step5.运行。

启动容器，访问Servlet。

打开浏览器，在地址栏输入 http://ip:port/appname/url-pattern。

注：

url-pattern是一个字符串，在web.xml中设置。

3.使用eclipse开发Servlet

## step1.集成tomcat

注:

即使得eclipse可以管理tomcat(包括启动、关闭tomcat,自动部署等)

可以参考

http://doc.tedu.cn/tomcat/index.html

## step2.创建maven工程

## step3.添加一个java类

注:

继承HttpServlet

override service方法。

参考代码:

public class HelloServlet extends HttpServlet{

public HelloServlet() {

System.out.println("HelloServlet的构造器");

}

@Override

/\*\*

\* Servlet容器(比如Tomcat)在收到请求之后，

\* 会解析请求数据包，将解析到的数据存放到request

\* 对象里面，同时，还会创建一个response对象。

\* 接下来，容器会创建Servlet对象，然后调用该

\* 对象的service方法来处理请求。

\* 注:

\* 作为开发人员，不用再处理网络相关的问题。

\* 比如，要读取请求参数，只需要调用request对象

\* 提供的方法即可；如果要向浏览器发送响应，只需要

\* 调用response对象提供的方法。

\*/

protected void service(

HttpServletRequest request,

HttpServletResponse response)

throws ServletException,

IOException {

System.out.println(

"HelloServlet的service方法");

/\*

\* 设置content-type消息头的值，告诉浏览器，

\* 服务器返回的数据类型。

\* 注：

\* 浏览器会按照指定的数据类型来解析。

\*/

response.setContentType("text/html");

/\*

\* 通过response对象获得一个输出流。

\*/

PrintWriter out = response.getWriter();

/\*

\* 将数据写到response对象上。

\* 注：

\* 容器会从response对象上提取数据，生成

\* 响应数据包并发送给浏览器。

\*/

out.println("Hello Kitty");

out.close();

}

}

## step4.在web.xml文件中，添加Servlet配置信息。

参考配置:

<servlet>

<servlet-name>helloServlet</servlet-name>

<!--

类名一定要写完整(注意大小写)

注:

容器会使用java反射完成类的加载，

如果类名写错，则容器会找不到该类的字节码文件。

-->

<servlet-class>web.HelloServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>helloServlet</servlet-name>

<!--

url-pattern用来定义访问路径，即

如何调用该servlet

(http://ip:port/appname/url-pattern)。

比如:

http://localhost:8080/day01-lab/hello

-->

<url-pattern>/hello</url-pattern>

</servlet-mapping>

4.Servlet是如何运行的?

比如，在浏览器地址栏输入http://ip:port/appname/hello

step1.浏览器依据ip、port建立与Servlet容器之间的连接。

step2.浏览器创建请求数据包并发送给容器。

step3.容器解析请求数据包中的数据，然后将解析到的数据添加到

request对象里面，同时，容器还会创建一个response对象。

step4.容器依据请求路径，创建对应的Servlet对象，然后调用该对象的

service方法。

注：

容器会将request对象和response对象作为参数传递过来。

开发人员可以通过调用request对象提供的方法来获得请求数据包中

的数据（比如请求参数），然后将处理结果写到response对象即可。

step5.容器通过response对象获得处理结果，然后创建响应数据包并发送给 浏览器。

step6.浏览器解析响应数据包中的数据，生成相应的页面。

5.常见问题

## (1)404

含义:404是一个状态码，表示服务器依据请求路径找不到对应的资源。

原因:

a.请求路径写错。没有严格按照http://ip:port/appname/url-pattern

来写。

b.没有部署或者部署失败。

## (2)500

含义:500也是一个状态码，表示服务器端程序运行出错。

原因:

a.没有严格按照Servlet规范来写代码和配置。

比如，没有继承HttpServlet。

又比如，<server-class>类名写错。

b.代码不严谨。

比如，对请求参数值没有做检查就做类型转换。

(3)405

含义：405也是一个状态码，表示服务器找不到处理方法。

原因:

service方法没有正确override。

6.http协议

(1)什么是http协议?

是一种网络应用层协议，规定了浏览器如何与web服务器之间进行通信以及相 应的数据包的结构。

浏览器与web服务器之间如何通信?

step1.建立连接

step2.发送请求

step3.发送响应

step4.关闭连接

如果浏览器要再发送请求，需要重新建立新的连接，即“一次请求，一次连接”。

优点：web服务器可以利用有限的连接为尽可能多的请求服务。

(2)数据包的结构

1)请求数据包

a.请求行 (请求方式　请求资源路径　协议和版本)

b.若干消息头

消息头是一些键值对（使用": "隔开），浏览器与web服务器之间

可以通过发送一些消息头来传递一些特定的信息。比如，浏览器

可以发送"user-agent"消息头来告诉web服务器浏览器的类型和版本。

c.实体内容

只有当请求方式为post时，实体内容才会有数据。

2)响应数据包

a.状态行　(协议和版本　状态码　状态描述)

注:

状态码是一个三位数字，描述服务器处理请求的一种状态

　　 200: 正解

500: 系统出错

404: 依据请求路径找不到对应的资源

405: 找不到处理方法

b.若干消息头

web服务器也可以发送一些消息头给浏览器，比如，可以发送 "content-type"消息头，告诉浏览器，服务器返回的数据类型和编码。

c.实体内容

程序的处理结果，浏览器会解析出来，生成相应的页面。

(3)两种请求方式

1)get请求

a.哪一些情况下浏览器会发送get请求?

在浏览器地址栏输入某个地址

点击链接

表单默认的提交方式

b.特点

b1.会将请求参数显示在浏览器地址栏，不安全。

注：

有一些网络设备，比如路由器，会记录请求地址。

b2.会将请求参数添加到请求资源路径的后面（即请求行里面），只

能提交少量的数据。

注：

整个请求行，大约能存放2k左右的数据。

2)post请求

a.哪一些情况下，浏览器会发送post请求?

将表单的method属性设置为post。

b.特点

b1.不会将请求参数显示在浏览器地址栏，相对安全一些。

注：

http协议对数据包并不会加密处理，所以，对于敏感数据

需要加密(使用https协议)。

b2.会将请求参数添加到实体内容里面，可以提交大量数据。

7.Servlet输出中文

## (1)为什么会有乱码?

out.println方法在默认情况下，会使用"iso-8859-1"来编码。

## (2)如何解决?

response.setContentType("text/html;charset=utf-8");

8.如何读取请求参数值?

## (1)方式一

String request.getParameter(String paramName);

注:

paramName是请求参数名，如果请求参数名写错，返回null。

## (2)方式二

String[] request.getParameterValues(String paramName);

注：

当有多个请求参数名相同的时候，使用此方法。

对于多选框，如果一个都不选择，会返回null值。

9.表单包含有中文参数值，如何处理?

## (1)为什么会有乱码?

提交表单时，浏览器会对中文参数值进行编码(比如，使用utf-8),

而服务器端默认会使用iso-8859-1来解码，所以，会产生乱码。

注:

浏览器会使用打开该表单所在的页面时的字符集来编码。

## (2)如何解决?

如果是get请求:

在tomcat的配置文件的conntector节点中添加URLEncoding属性

并设置值为”utf-8”

如果是post请求:

request.setCharacterEncoding(“utf-8”);

此代码要添加到所有request.getParameter方法的前面

10.访问数据库

## (1)步骤

step1.导包(jdbc驱动、连接池)。

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.6</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

</dependencies>

step2.拷贝DBUtils到util包下。

step3.拷贝jdbc.properties文件。

step4.测试DBUtils。

step5.建表

create table t\_user(

id int primary key auto\_increment,

username varchar(50) unique,

password varchar(30),

email varchar(30)

);

step6.在service方法里面，使用jdbc api访问数据库。

参考代码:

//将用户信息插入到数据库

Connection conn = null;

PreparedStatement ps = null;

try {

conn = DBUtils.getConn();

String sql = "INSERT INTO t\_user "

+ "VALUES(null,?,?,?)";

ps = conn.prepareStatement(sql);

ps.setString(1, username);

ps.setString(2, pwd);

ps.setString(3, email);

ps.executeUpdate();

out.println("添加成功");

} catch (SQLException e) {

/\*

\* step1.记日志(保留现场)

\*/

e.printStackTrace();

/\*

\* step2.看异常能否恢复，如果不能够

\* 恢复(比如，数据库服务停止了、网络

\* 中断了等等，这样的异常我们通常称之为

\* 系统异常)，则提示用户稍后重试。

\* 如果能够恢复，则立即恢复。

\*

\*/

out.println("系统繁忙，稍后重试");

}finally {

DBUtils.close(conn, ps, null);

}

11.重定向

## (1)什么是重定向?

服务器通知浏览器向某个地址发送请求。

注:

服务器可以通过发送302状态码和Location消息头(该消息头的值

是一个地址，一般称之为重定向地址)，浏览器收到之后，会立即

向重定向地址发送请求。

## (2)如何重定向?

response.sendRedirect(String url);

注：

a.url就是重定向地址。

b.重定向之前，容器会先清空response对象上存放的所有数据。

## (3)特点

a.重定向地址是任意的。

b.重定向之后，浏览器地址栏的地址会发生变化。

12.DAO

## (1)DAO是什么? (Data Access Object)

封装了数据访问逻辑的对象。

## (2)如何写一个DAO?

step1.写一个java类，用于存放从数据库中查询出来的数据。

注：

关系数据库里面存放的是一条条记录，而java语言擅长处理

对象，所以，我们经常有这样的需要，即将记录转换成一个对应的

java对象，方便处理。

step2.写一个java类，提供一些访问数据库的方法。

## (3)DAO的优点?

a.方便测试

注:

如果将jdbc代码直接写在servlet类里面，需要部署整个web应用才能

测试，而将jdbc代码写在DAO类里面，可以直接测试（不需要部署整个 应用）。

b.方便分工协作

c.方便代码的维护

注：

dao实现发生改变，比如原来使用jdbc，现在使用mybatis,

并不影响调用者(即不影响Servlet)。

13.JSP

## (1)jsp是什么?

sun公司制订的一种服务器端的动态页面技术规范。

注：

因为使用servlet虽然也可以生成动态页面，但是过于繁琐(

需要使用大量的out.println语句),并且不利于页面的维护(如果

要修改页面，就需要修改java代码),所以，sun才制订了jsp技术规范。

jsp是一个以.jsp为后缀的文件，该文件主要由html和少量的

java代码组成。容器会将这个文件自动转换成一个servlet然后执行。

jsp本质就是一个servlet!

## (2)如何写一个jsp?

### step1. 写一个以.jsp为后缀的文件。

### step2. 在文件里面，可以使用如下元素：

1)html(css,javascript)

直接写即可。

2)java代码

a.java代码片断:

<% java代码 %>

b.jsp表达式

<%= java表达式 %>

C.jsp声明

<%! 声明变量或者方法 %>

3)隐含对象

a.什么是隐含对象?

可以直接使用的对象，比如out,request,response。

b.为什么可以直接使用这些隐含对象?

因为容器在生成对应的servlet源代码时，会自动添加获得这些

对象的代码。

c.有哪一些隐含对象?

c1. out

c2. request

c3. response

c4. session

c5. application (servlet上下文)

c6. exception (只有当page指令的isErrorPage属性值为true时 才能用)

c7. pageContext(页面上下文)

c1. pageContext是什么?

容器会为每一个jsp实例创建唯一的一个符合PageContext

接口要求的对象，该对象我们可以称之为页面上下文。

注：

只要jsp实例还在，pageContext就会一直存在。

c2. 作用1 绑订数据 (a5.jsp,a6.jsp)

注：

绑订到pageContext上的数据，只有对应的jsp实例能 够访问。类似“钱包”。

作用2 提供了一些方法用来获得其它所有的隐含对象。

c8. config (a7.jsp ServletConfig)

c9. page: jsp实例本身。

注：

jsp实例是jsp对应的servlet对象。

4)指令

a.什么是指令?

告诉容器，在将jsp转换成一个Servlet时，额外做一些处理，比如，

导一些包。

b.指令的语法

<%@ 指令名　　属性=值 %>

注：

多个属性要使用空格隔开。

c.page指令

import属性：用于指定要导入的包名。比如

<%@ page import="java.util.\*"%>

<%@ page import="java.util.\*,java.text.\*"%>

contentType属性：用于设置response.setContentType方法的内容。

pageEncoding属性：指定jsp文件的编码。

session属性:

缺省值是true,如果值是false,则容器不再添加获得

session对象的代码了，也就是说，不能够使用session

隐含对象了。

errorPage属性:

用来指定一个异常处理页面，当jsp运行发生异常，则容器

会调用异常处理页面。

isErrorPage属性:

缺省值是false,如果值是true,就可以使用exception

隐含对象了。

注：

exception对象用于获得异常的一个简短的描述。

d.include指令

e.taglib指令

## (3)jsp是如何执行的?

step1.容器将jsp转换成一个Servlet。

html(css,javascript) -----> service方法里面，使用out.write输出。

注(了解)：

out.println方法会将null直接输出，而out.write方法会将

null转换成""输出（更友好）。

<% %> -------> service方法里，照搬。

<%= %> -------> service方法里，使用out.print方法输出。

step2.容器再调用该Servlet。

注:

容器会将该Servlet编译，然后实例化，调用其service方法。

14.include指令

## (1)语法

<%@ include file=""%>

注:

file属性用来指定被包含的文件名。当容器在将jsp转换成

对应的servlet源代码时，会将file属性指定的文件的内容插入到

该指令所在的位置。

## (2)细节

被包含的jsp文件并没有对应的Servlet。

15.转发

## (1)什么是转发?

一个web组件将未完成的处理交给另外一个web组件继续做。

注：

web组件是servlet和jsp的统称。

通常是一个Servlet获得数据之后转交给jsp来展现。

## (2)如何转发?

step1.将数据绑订到request对象上。

request.setAttribute(String name,Object obj);

注:

name:绑订名

obj:绑订值

依据绑订名获得绑订值，如果绑订值不存在，返回null。

Object request.getAttribute(String name);

step2.获得转发器。

RequestDispatcher rd =

request.getRequestDispatcher(String uri);

注:

RequestDispatcher是一个接口，该方法会返回一个符合该接口

要求的对象，该对象我们称之为转发器。

转发的本质是一个web组件通知容器去调用另外一个web组件。

可以将转发器看作是一个媒介，通过该媒介，就可以通知容器去

调用另外一个web组件了。

uri就是转发的目的地，通常是一个jsp。

step3.转发

rd.forward(request,response);

## (3)特点

a.转发之后，浏览器地址栏的地址是不变的。

b.转发的目的地有限制，要求属于同一个web应用。

16.比较转发与重定向

## (1)能否共享request?

转发可以、重定向不行。

注:

a.容器收到请求之后，会立即创建request和response对象，一旦

响应发送完毕，会立即销毁这两个对象。也就是说，request和response

对象的生存时间是一次请求和响应期间存在。

b.转发是一次请求，重定向是二次请求。

## (2)浏览器地址栏的地址有无变化?

转发没有变化，重定向有变化。

## (3)目的地有无限制?

转发有（要求属于同一个应用），重定向无任何限制。

17.状态管理

## (1)什么是状态管理?

将浏览器与web服务器之间多次交互当做一个整体来处理，并且将多次

交互所涉及的数据(即状态)保存下来。

## (2)如何进行状态管理?

a.将状态保存在浏览器端(Cookie)

b.将状态保存在服务器端(Session 会话)

## (3)Cookie

### 1)Cookie是什么?

服务器临时存放在浏览器端的少量数据，用于跟踪用户状态。

### 2)Cookie的工作原理

当浏览器访问服务器时，服务器时可以将少量数据以set-cookie消息头的形 式发送给浏览器，浏览器会保存下来；

当浏览器再次访问服务器时，会将之前保存的这些数据以cookie消息头的形 式发送给服务器。

### 3)如何添加Cookie?

Cookie c = new Cookie(String name,String value);

response.addCookie(c);

注:

name一般称之为cookie的名称，value称之为cookie的值。

### 4)如何读取Cookie?

Cookie[] request.getCookies();

注：

该方法有可能返回null。

String cookie.getName();

String cookie.getValue();

### 5)生存时间问题

a.默认情况下，浏览器会将cookie保存在内存里面。浏览器如果关闭，则

cookie会被删除。

b.可以调用setMaxAge方法来设置cookie的生存时间。

cookie.setMaxAge(int seconds);

注:

单位是秒。 365 \* 24 \* 60 \* 60

seconds > 0: 浏览器会将cookie保存在硬盘上(一般是文件的形式), 超过指定的时间，cookie会被删除。

seconds < 0: 默认情况（即保存在内存里面）。

seconds = 0: 删除cookie。

比如，要删除一个名称为username的cookie:

Cookie c = new Cookie("username","");

c.setMaxAge(0);

response.addCookie(c);

### 6)编码问题

a.cookie只能保存合法的ascii字符，如果是中文，需要转换成对应的

ascii字符的形式。

b.可以使用以下方法进行处理

String URLEncoder.encode(String str,String charset);

String URLDecoder.decode(String str,String charset);

c.建议，在添加cookie时，不管是否为中文，统一进行编码处理。

### 7)路径问题

a.什么是cookie的路径问题?

浏览器访问服务器上的某个地址时，会比较该地址是否符合cookie的路径

要求，只有符合条件cookie才会被发送。

b.Cookie的默认路径

默认路径等于添加该cookie的web组件的路径。

比如，/day06/biz01/addCookie.jsp添加了一个cookie,则该

cookie默认的路径是"/day06/biz01"。

c.哪一些cookie会被发送?

请求地址等于cookie的路径或者是其子路径,则该cookie会被发送。

比如，cookie的路径是"/day06/biz01",则

"/day06/findCookie1.jsp",　不会被发送;

"/day06/biz01/findCookie2.jsp",会被发送;

"/day06/biz01/sub/findCookie3.jsp",会被发送。

d.可以调用setPath方法来修改cookie的路径

cookie.setPath(String path);

### 8)Cookie的限制

a.Cookie可以被用户禁止。

b.不安全。

注：

Cookie是将状态保存在浏览器端，服务器端并不保存任何状态，

所以，对于敏感数据，尽量不要保存在cookie里面，如果一定要保存，

则加密。

c.保存的数据大小有限制。

注：

一般是4k左右。

d.数量也有限制。

注：

一般是几百个左右。

e.只能存放字符串。

18.Session (会话)

## (1)什么是Session?

服务器为了保存用户状态而创建一个特殊的对象。

## (2)工作原理

浏览器访问服务器时，服务器创建一个特殊的对象(即session对象，该对象

有一个唯一的id,一般称之为sessionId),服务器将sessionId以cookie的方 式发送给浏览器。

当浏览器再次访问服务器，会将sessionId发送过来，服务器依据sessionId

就可以找到对应的session对象。

## (3)如何获得session对象?

### 1)方式一

HttpSession s = request.getSession(boolean flag);

注:

HttpSession是一个接口，request.getSession方法会返回一个

实现了该接口的对象，该对象我们称之为session对象。

当flag为true时：

先查看请求当中是否有sessionId,如果没有，则创建一个

session对象；如果有sessionId,则依据该sessionId查找

对应的session对象,如果找到了，则返回该对象，找不到，

则创建一个新的session对象。

当flag为false时:

先查看请求当中是否有sessionId,如果没有，返回null;

如果有sessionId,则依据该sessionId查找

对应的session对象,如果找到了，则返回该对象，找不到，

返回null。

### 2)方式二

HttpSession s = request.getSession();

注：

等价于request.getSession(true);

## (4)session对象的常用方法

a.获得sessionId

String session.getId();

b.绑订数据

session.setAttribute(String name,Object obj);

c.依据绑订名获得绑订值

Object session.getAttribute(String name);

注：如果绑订名对应的值不存在，返回null。

d.解除绑订

session.removeAttribute(String name);

## (5)session超时

### 1)什么是session超时?

服务器会将空闲时间过长的session对象删除掉。

注：

为了节省内存空间。

服务器默认的超时时长一般是30分钟。

### 2)如何修改session超时时间?

方式一　修改服务器的配置文件

<session-config>

<session-timeout>30</session-timeout>

</session-config>

方式二　编程的方式

session.setMaxInactiveInterval(int seconds);

注:

设置两次请求之间的最大间隔时间。如果超过这个间隔时间，

session对象会被删除。

## (6)删除session

session.invalidate();

## (7)session验证 (案例)

step1.登录成功之后，在session对象上绑订一些数据，比如:

session.setAttribute("user",username);

step2.当用户发送请求过来，检查session对象上是否有对应的数据，比如:

注：

只有访问受保护的资源时，才需要进行session验证。

Object obj = session.getAttribute("user");

if(obj == null){

//没有登录,重定向到登录页面

response.sendRedirect("login.jsp");

}

## (8) 比较session与cookie

session作为一种服务器端的状态管理技术、相对于cookie更安全、

可以存放更多的数据、支持更丰富的数据类型，缺点是会占用服务器端的

大量的内存资源。

19.Servlet的生命周期

## (1)什么是Servlet的生命周期?

Servlet容器如何创建Servlet实例、如何对其进行初始化操作、

如何调用其方法来处理请求、以及如何销毁其实例的整个过程。

也可以用一句话来描述：即容器是如何管理Servlet的。

## (2)生命周期分成哪几个阶段?

### 1)实例化

a.什么是实例化?

容器调用Servlet的构造器，创建Servlet实例。

b.什么时候实例化?

默认情况下，容器只有在收到请求之后，才会创建Servlet实例。

注:

容器只会创建一个Servlet实例！

在添加适当配置之后，可以让容器启动之后，立即创建Servlet实例

在web.xml关于Servlet的配置中添加配置启动加载，即：load-on-startup 节点，参数值是个大于等于0的整数，值越小越优先被创建

### 2)初始化

a.什么是初始化?

容器在创建好Servlet实例之后，会调用该实例的init(ServletConfig config)方法。

注:

该方法只会执行1次！

b.GenericServlet已经提供了init(ServletConfig config)方法的实现。

该方法是这样实现的:

将容器传递过来的ServletConfig对象保存下来了，并且提供了一个

getServletConfig方法来获得该对象。另外，该方法还调用了一个

init()方法。作为开发人员，如果要扩展，只需要override init() 方法

c.ServletConfig提供了一些实用方法，比如读取初始化参数的方法。

配置初始化参数:

在web.xml关于Servlet的配置中添加以下节点:

<init-param>

<param-name>company</param-name>

<param-value>码多多</param-value>

</init-param>

调用ServletConfig对象提供的方法来读取初始化参数：

ServletConfig config = getServletConfig();

String company = config.getInitParameter(“company”);

### 3)就绪(调用)

a.什么是就绪?

容器在收到请求之后，会调用Servlet实例的service方法来处理请求。

b.HttpServlet已经提供了service方法的实现。

该方法是这样实现的：

依据请求类型调用对应的doXXX方法，比如get请求调用doGet方法，

post请求调用doPost方法。doXXX方法只是简单地抛出了一个异常。

作为开发人员，在开发一个Servlet时，有两个选择，一是直接override

HttpServlet的service方法，或者override HttpServlet的doGet和

doPost方法。

### 4)销毁

a.什么是销毁?

容器在删除Servlet实例之前，会调用该实例的destroy方法。

注：

该方法只会执行一次。

b.GenericServlet已经提供了destroy方法的实现。

该方法的实现为空（什么都没有做），开发人员可以override

destroy方法来扩展。

## (3)相关的接口与类(了解)

a. Servlet接口

init(ServletConfig config);

service(ServletRequest req,ServletResponse response);

destroy();

b. GenricServlet抽象类

实现了Servlet接口中的部分方法(init,destroy)。

c. HttpServlet抽象类

继承了GenericServlet，实现了service方法。

20.容器是如何处理请求资源路径的?

比如，在浏览器地址栏输入http://ip:port/appname/abc.html,容器如何处理?

step1. 容器默认会认为访问的是一个Servlet，所以容器会查看web.xml中

有没有匹配的Servlet。

有三种匹配方式:

精确匹配:

<url-pattern>的值必须等于请求路径。比如说，对于上述请求，

url-pattern的值必须是"/abc.html"。

通配符匹配:

使用 "\*" 匹配任意的零个或者多个字符。比如:

<url-pattern>/\*</url-pattern>

<url-pattern>/demo/\*</url-pattern>

后缀匹配:

使用"\*."开头，后接一个后缀，用来匹配所有以该后缀结尾的请求。

比如:

<url-pattern>\*.do</url-pattern>

<url-pattern>\*.action</url-pattern>

step2.如果没有匹配的Servlet,则容器会查找对应的文件。

21.如何让一个Servlet处理多种请求?

step1.使用后缀匹配，比如

<url-pattern>\*.do</url-pattern>

step2.分析请求资源路径，进行不同的处理

//step1.获得请求资源路径

String uri = request.getRequestURI();

System.out.println("uri:" + uri);

//step2.分析请求资源路径，进行不同的处理

String path =

uri.substring(

uri.lastIndexOf("/"),

uri.lastIndexOf("."));

System.out.println("path:" + path);

if("/list".equals(path)) {

System.out.println("处理用户列表请求");

}else if("/add".equals(path)) {

System.out.println("处理添加用户请求");

}

22.过滤器

## (1)什么是过滤器?

Servlet规范当中定义的一种特殊的组件，用于拦截Servlet容器的调用过程。

注：

容器收到请求之后，会先调用过滤器，再调用Servlet。

## (2)如何写一个过滤器?

step1.写一个java类，实现Filter接口。

step2.在接口方法当中，实现拦截处理逻辑。

step3.配置过滤器。(web.xml)

## (3)过滤器的优先级

当有多个过滤器都满足拦截的要求，则容器依据<filter-mapping>

配置的先后顺序来执行。

## (4)初始化参数

step1.配置初始化参数

<!-- 配置初始化参数 -->

<init-param>

<param-name>illegal</param-name>

<param-value>狗</param-value>

</init-param>

step2.使用FilterConfig对象提供的方法来读取

//读取初始化参数

String illegal = config.getInitParameter("illegal");

23.过滤器的优点

a.在不修改原有代码的基础上，添加一些简单的功能。

b.将多个组件相同的功能集中写在一个类里面(即过滤器里面)，方便

代码的维护。

24.监听器

## (1)什么是监听器?

Servlet规范当中定义的一种特殊的组件，用于监听Servlet容器产生的事件 并进行相应的处理。

注：

容器产生的事件分为两类:

a.生命周期相关的事件:

容器创建了或者销毁了request、session、servlet上下文

时产生的事件。

b.绑订数据相关的事件:

调用了request、session、servlet上下文的setAttribute、

removeAttribute时产生的事件。

## (2)servlet上下文

a.什么是servlet上下文?

容器启动之后，会为每一个web应用创建唯一的一个符合ServletContext

接口要求的对象，该对象我们称之为servlet上下文。

b.如何获得servlet上下文?

GenericServlet,ServletConfig,FilterConfig,HttpSession

都提供了一个方法(getServletContext)来获得上下文。

c.作用

c1.绑订数据

setAttribute,getAttribute,removeAttribute

注:

request、session、servlet上下文都可以绑订数据，有什么区别?

区别1:

绑订的数据生存时间不一样，按照生存时间的长短来排序，依次是

request < session < servlet上下文。

在满足使用条件的情况下，优先使用生命周期短的。

区别2:

绑订到session对象上的数据，只有与之对应的用户能访问到，

而绑订到上下文上的数据，所有用户都能访问到。

c2.读取全局的初始化参数

<!-- 配置全局的初始化参数 -->

<context-param>

<param-name>company</param-name>

<param-value>北京达内科技有限公司</param-value>

</context-param>

String company = sctx.getInitParameter("company");

## (3)如何写一个监听器?

step1.写一个java类，实现相应的监听器接口。

注：

要依据监听的事件类型去选择实现相应的接口，比如，

要监听session对象的创建和销毁，需要实现

HttpSessionListener接口。

step2.在监听器接口方法当中，实现相应的监听处理逻辑。

step3.配置监听器。(web.xml)

25.路径问题

## (1)什么是路径问题？

在链接、表单提交、重定向以及转发时，如何填写相应的路径。

比如:

<a href="del.do">

<form action="login.do">

response.sendRedirect("login.jsp")

request.getRequestDispatcher("error.jsp")

## (2)相对路径

不以"/"开头的路径。

## (3)绝对路径

以"/"开头的路径。

## (4)如何写绝对路径

链接、表单提交、重定向从应用名开始写，转发从应用名之后开始写。

注，不要将应用名直接写在路径里面，而应该使用以下

方法来获得实际部署时的应用名。

String request.getContextPath();

26.Servlet的线程安全问题

## (1)为什么说Servlet会有线程安全问题?

a.容器只会创建一个Servlet实例。

b.容器收到一个请求，就会启动一个线程来处理。

这样，就有可能有多个线程同时去调用某个Servlet实例，就有可能

产生线程安全问题(比如，这些线程要修改该实例的某个属性)。

## (2)如何解决?

使用synchronized对有可能产生线程安全问题的代码加锁。

注：加锁会影响性能。

27.注释

a. <!-- 注释内容 -->

注：

如果注释的内容是java代码，java代码会执行。

b. <%-- 注释内容 --%>

注：

如果注释的内容是java代码，java代码不会执行。

28.jsp标签和el表达式

## (1)什么是jsp标签?

jsp标签类似于html标签，用于替换jsp文件中的java代码，使用jsp标签，

可以简化jsp文件，便于代码的复用，并且也利于jsp文件的维护。

注：

如果将包含有java代码的jsp文件给美工去修改就很不方便。

## (2)什么是el表达式

一套简单的运算规则，用于给jsp标签的属性赋值，也可以脱离jsp标签直接 使用。

## (3)el表达式的用法

### 1)读取bean的属性值

注(了解)

如果一个java类满足如下几个条件，可以称之为一个javabean。

public class

public 构造器

最好实现Serializable接口

有一些属性以及对应的get/set方法

a.方式一 ${user.username}

注:

a1.执行过程:

容器会依次从pageContext-->request-->session

-->application中查找绑订名为"user"的对象(即调用

getAttribute方法),找到该对象之后，会调用该对象的

"getUsername"方法，然后输出该方法的返回结果。

a2.优点:

将null转换成""输出。

如果找不到该对象，会输出"",不会报空指针异常。

a3.可以指定查找范围:

可以使用pageScope、requestScope、sessionScope、

applicationScope来指定查询范围，比如

${sessionScope.user.username}。

b.方式二 ${user['username']}

注:

b1.[]里面可以出现绑订名。

b2.[]里面可以出现从0开始的下标，用于读取数组中指定下标的

某个元素的值。

1. 使用el表达式进行一些简单的运算

## (1)算术运算

+,-,\*,/,%

注:

"+" 只能求和,不能够连接字符串。

## (2)关系运算

>,>=,<,<=,==,!=

## (3)逻辑运算

&&,||,!

## (4)empty

empty 运算：用于判断一个集合内容是否为空，或者是否为一个空字符串。

以下四种情况，运算结果都为true。

a.集合内容为空。

b.空字符串。

c.null。

d.找不到对应的值。

30.使用el表达式读取请求参数值

a.${param.username}等价于

request.getParameter("username");

b.${paramValues.city}等价于

request.getParameterValues("city");

31.获得应用名

${pageContext.request.contextPath}

等价于

request.getContextPath();

32.jstl (java standard tag lib)

## (1)jstl是什么?

apache开发的一套jsp标签，后来捐献给了sun,sun将其命名为jstl。

## (2)如何使用jstl?

step1.导包(jstl)。

<dependency>

<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

step2.使用taglib指令导入要使用的jsp标签。

<%@ taglib uri="" prefix=""%>

uri: 指定jsp标签的命名空间。

注:

命名空间：为了区分同名的元素而在元素前添加的一个限定，

通常是一个域名。

prefix:指定命名空间的别名。

##　(3)if标签 (c1.jsp)

１)语法

<c:if test="" var="" scope="">

标签体

</c:if>

2)作用：

相当于一个if语句，当test属性值为true时，容器会执行标签体的内容。

test属性可以使用el表达式来计算。

var属性用于指定绑订名，scope属性指定绑订的范围 ("page","request","session","application")。

## (4)jsp标签的执行过程

容器依据标签的命名空间找到标签的描述文件(.tld文件),然后依据

标签的名称找到标签类，接下来，容器会将标签类实例化并调用该实例的

相应方法。

## (5)choose标签 (c2.jsp)

1)语法

<c:choose>

<c:when test="">

</c:when>

...

<c:otherwise>

</c:otherwise>

</c:choose>

2)作用

相当于if..else if..else语句。其中,when标签可以出现１次或者多次，

相当于一个if语句，otherwise标签可以出现０次或者１次，相当于最 后那个else语句。

当test属性值为true,执行标签体的内容，可以使用el表达式来赋值。

## (6)forEach标签 (c3.jsp)

1)语法

<c:forEach items="" var="" varStatus="">

</c:forEach>

2)作用

用于遍历集合或者数组，其中，items属性用于指定要遍历的集合或者数 组，可以使用el表达式来赋值。

var属性用于指定绑订名，绑订的范围固定是pageContext,绑订值是集合

或者数组中的某个元素。

注：

该标签每次从集合或者数组中取一个元素，然后将该元素绑订到

pageContext上。

varStatus属性用于指定绑订名，绑订的范围固定是pageContext,绑订

值是一个特殊的对象，该对象由该标签提供，可以通过该对象获得当前遍 历的状态。

注：

这个特殊的对象，提供下以下几个方法

getIndex() : 获得当前正在被遍历的元素的下标（从０开始）。

getCount() : 获得当前是第几次迭代(从１开始)。

33.自定义标签 (了解)

step1. 写一个java类，继承SimpleTagSupport类。

注:

jsp标签有两个版本，早期版本称之为复杂标签技术，新的

版本称之为简单标签技术。

step2. override doTag方法，在该方法里写上处理逻辑。

/\*\*

\* 标签类

\* 注:

\* 标签有哪些属性，对应的标签类也必须有相应的

\* 属性(属性名要相同，类型要匹配,并且要有对应的

\* set方法)。

\*/

public class HelloTag extends SimpleTagSupport{

private String info;

private int qty;

public HelloTag() {

System.out.println("HelloTag的构造器");

}

public void setInfo(String info) {

System.out.println("HelloTag的setInfo方法...");

this.info = info;

}

public void setQty(int qty) {

System.out.println("HelloTag的setQty方法...");

this.qty = qty;

}

@Override

public void doTag() throws JspException, IOException {

System.out.println("HelloTag的doTag方法...");

/\*

\* 通过继承自SimpleTagSupport类提供的

\* 方法来获得pageContext。

\* pageContext提供了用来获得其它所有隐含对象

\* 的方法。

\*/

PageContext pctx = (PageContext)getJspContext();

JspWriter out = pctx.getOut();

for(int i = 0; i < qty; i ++){

out.println(info + "<br/>");

}

}

}

step3. 描述标签(.tld文件)

<tag>

<name>hello</name>

<tag-class>tag.HelloTag</tag-class>

<!--

设置标签体的内容。

有三个值:

empty:

该标签没有标签体。

scriptless:

该标签有标签体，但是，在标签体里面，不能够

出现java代码(<% %>,<%= %>,<%! %>)

JSP:

该标签有标签体，并且，标签体里面可以出现

java代码。但是，只有复杂标签才支持该值。

-->

<body-content>empty</body-content>

<attribute>

<name>info</name>

<!--

如果值为true,表示该属性必选。

-->

<required>true</required>

<!--

如果值为true,表示该属性可以动态赋值

(可以使用el表达式来赋值)。

-->

<rtexprvalue>false</rtexprvalue>

</attribute>

<attribute>

<name>qty</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

34.MVC (Model,View,Controller)

## (1)MVC是什么?

是一种软件架构思想、其核心思想是:在设计一个软件时，应该将数据处理与

数据展现分开，按照这种思想，可以将一个软件划分成三种不同类型的模块，

分别是模型、视图和控制器。

其中，模型负责数据处理（封装业务逻辑），视图负责数据展现(表示逻辑)、

控制器负责协调模型和视图(请求先发送给控制器，由控制器选择对应的模型

来处理；模型将处理结果要交给控制器，由控制器选择对应的视图来展现)。

## (2)如何使用MVC?

在开发一个web应用时，可以使用java类来充当模型、使用Servlet来充当

控制器，使用jsp来充当视图。

## (3)MVC的优点

a.将数据处理与数据展现分开，处理结果可以使用不同的方式来展现。

b.方便测试。

注:

比如，将业务处理逻辑写在Servlet里面，需要部署整个应用才能

测试，如果将业务逻辑写在java里面，就可以直接测试。

c.方便分工协作。

d.方便代码的维护。

## (4)MVC的缺点

使用mvc,会增加代码量，会增加软件设计的难度，相应会增加软件的成本。

所以，只有具有一定规模，并且强调软件的可维护性的情况下，才需要使用

MVC。

35.什么是框架

一个软件的半成品。我们可以将一些常用的、功能相同的功能提前写好，这样 在开发时，就不用重复开发了。使用框架，可以提高开发的效率，提升软件的 质量。

**七、SSM(Spring + SpringMVC + MyBatis)**

见代码...