Java SE

# API

## **JDK API**

JDK中包含大量的API类库, API(Application Programming Interface, 应用程序编程接口), 就是一些已写好/可供直接调用的功能

### **经常使用的包**

|  |  |
| --- | --- |
| 包 | 功能 |
| java.lang | java程序的基础类, 如字符串, 多线程等, 不需要import |
| java.util | 常用工具类, 如集合/随机数产生器/日历/时钟 |
| java.io | 文件操作, 输入/输出操作 |
| java.net | 网络操作 |
| java.math | 数学运算相关操作 |
| java.security | 安全相关操作 |
| java.sql | 数据库访问 |
| java.text | 处理文字, 日期, 数字, 信息的格式 |

## **文档注释**

### 2.1) **文档注释的规范**

**以/\*\* 开始, 以\*/结束, 加在类和方法的开头.**

是一个功能级注释, 用来说明一个类的功能, 方法的功能, 常量的功能, 而不是说明一句代码的作用

在类中使用, 用来描述当前类的设计意图, 功能等.

1. 加在文档的开头

@author Administrator: 作者

@version 1.0: 版本

@since JDK1.0

@see java.lang.String:参考某类etc

1. 加在类的开头

为指定用户添加一个问候语

@param name: 给定用户的名字

@return 含有问候语的字符串

# 字符串基本操作

## **String及其常用API**

### 1.1) **String是不可变对象**

1) java.lang.String使用了final修饰, 不能被继承

2) 字符串一旦创建, 对象永远无法改变, 但字符串引用可以重新赋值(字符串是不变对象, 内容一旦创建不可改变, 若改变一定会创建新对象)

### 1.2) **String常量池**

为了提高性能, 静态字符串(字面量/常量/常量连接的结果)在常量池中创建, 并尽量使用同一个对象, 重用静态字符串.对于重复出现的字符串直接量, JVM会首先在常量池中查找, 如果存在即返回该对象

### 1.3) **常用API**

1) **获得字符串长度: .length()**

任何字符都算1个char字符长度, 占用两个字节

2) **获得某字符(串)在数组中的位置:** .**indexOf(String str)/.lastIndexOf(String str)**

String str = "thinking in java";

int index = str.indexOf("in", 3); // 输出5

3) **截取当前字符串中指定范围内的字符串: .substring(int beginIndex(, int endIndex))**

java api有一个特点:通常用两个数字表示范围时, 都是:"含头不含尾"

1. **去掉字符串的前导和后继空字符: .trim()**
2. **返回指定位置对应的字符: char charAt(int index)**

String str = "thinking in java";

char c = str.charAt(1);

1. **判断当前字符串是否是以给定字符串结尾或开始的:** boolean starts[ends]With(String str)

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

1. **将一个字符串中的英文全部转化为大写或小写:** String toUpperCase(), String toLowerCase()

7) **将基本类型转化为字符串: .valueOf()**

int a = 1; String str = String.valueOf(a);

## **StringBuilder及其API**

字符串不变对象特性只针对字符串重用, 并没有考虑修改操作的性能, 因此string不适合频繁修改内容

若有频繁修改操作, 使用StringBuilder来完成

它是专门设计出来为了修改字符串内容的, 其提供了对字符串内容编辑操作所对应的:增, 删, 插, 改

|  |  |
| --- | --- |
| 常用方法 | 功能 |
| append(String str) | 追加字符串 |
| insert(int dstOffset, String s) | 插入字符串 |
| delete(int start, int end) | 删除字符串 |
| replace(int start, int end, String str | 替换字符串 |
| reverse() | 字符串反转 |

String line = "好好学习";

**StringBuilder builder = new StringBuilder(line);** // 基于给定字符串修改

**builder.append(**",天天向上"**)**; // 增加

line = builder**.toString()**; // 输出为String

System.out.println(line);

2019.02.28 - Day 20

# 正则表达式

## **基本正则表达式**

### 1.1) **基本正则表达式**

|  |  |
| --- | --- |
| 正则表达式 | 功能 |
| [abc] | a,b,c中任意一个字符 |
| [^abc] | 除了a,b,c的任意字符 |
| [a-z] | a,b,c,...,z中的任意一个字符 |
| [a-z]+ | 一个或多个a-z字符组成的字符串 |
| [a-zA-Z0-9] | a-z, A-Z, 0-9中任意一个字符 |
| [a-z&&[^bc]] | a-z中除了b和c意外的任意一个字符 |

### 1.2) **预定义字符集**

|  |  |
| --- | --- |
| 正则表达式 | 功能 |
| . | 任意一个字符 |
| \d | 任意一个数字字符 |
| \w | 单词字符, 相当于[a-zA-Z0-9\_] |
| \S | 空白字符, 相当于[\t\n\x0B\f\r] |
| \D | 非数字字符 |
| \W | 非单词字符 |
| \S | 非空白字符 |

### 1.3) **数量词**

|  |  |
| --- | --- |
| 正则表达式 | 功能 |
| []? | 出现0-1次 |
| []\* | 出现0到任意次 |
| []+ | 出现一次以上 |
| []{3,6} | 出现3-6次 |
| \D | 非数字字符 |
| \W | 非单词字符 |
| \S | 非空白字符 |

### 1.4) **()**

()圆括号表示分组, 可以将一系列正则表达式看做一个整体, 分组可以使用|表示或关系

email邮件地址

[a-zA-Z0-9\_]+@[a-zA-Z0-9]+(\.[a-zA-Z]+)+

↑至少一个字符 ↑转义

## **正则表达式**

### 2.1) **使用给定的正则表达式判断当前字符串是否符合格式要求, 满足返回true**

**boolean matches(String regex)**

// 注意, 该方法是做完全匹配验证的, 无论是否添加正则表达式的边界匹配符”^...$”都是做全匹配验证

// email邮件地址

// [a-zA-Z0-9\_]+@[a-zA-Z0-9]+(\\.[a-zA-Z]+)+

↑至少一个字符 ↑转义

String email = [“ai7@live.com”;](mailto:\“ai7@live.com\”;)

email.matches(regex);

boolean check = email.matches(regex);

if (check){

System.out.println(“是邮箱”);

}else{

System.out.println(“不是邮箱”);

}

### 2.2) **将当前字符串按照满足正则表达式要求的部分进行拆分, 并将拆分出的内容返回**

**String[] split(String regex)**

将当前字符串按照满足正则表达式要求的部分进行拆分, 并将拆分出的内容返回

String str = "abc123def456ghi";

// 按照数字部分将字符串拆分

String[] data = str.split("[0-9]+");

System.out.println(Arrays.toString(data)); // [abc, def, ghi]

// 按照字母, 拆分时, 若两侧没有内容时, 会拆分出空字符串

String[] data2 = str.split("[a-z]+");

System.out.println(Arrays.toString(data2)); // [, 123, 456]

// 末尾拆除的连续空串被忽视

String[] data3 = str.split("[a-z]");

System.out.println(Arrays.toString(data3)); // [, , , 123, , , 456]

String str2 = "abc123def456ghi7";

String[] data4 = str2.split("[a-z]");

System.out.println(Arrays.toString(data4)); // [, , , 123, , , 456, , , 7]

// 将图片重命名

String imageName = "123.jpg";

String[] arr = imageName.split("\\.");

System.out.println(arr.length);

System.out.println(Arrays.toString(arr));

imageName = System.currentTimeMillis()+"."+arr[arr.length-1];

System.out.println(imageName);

### 2.3) **将当前字符串中满足正则表达式的部分替换为给定内容**

**String replaceAll(String regex, String str)**

String str = "abc123def456ghi";

String str2 = str.replaceAll("[0-9]", "\*"); // abc\*\*\*def\*\*\*ghi

# Object

### 1) **重写toString方法**

Point p = new Point(1,2);

// Object类提供给了toString方法,意义是将该对象转换为一个字符串

// Object已经实现了该方法,返回的字符串内容为该对象的句柄(类名@地址),或称为对象的引用

String str = p.toString();

System.out.println(str); // day21\_Object.Point@15db9742

// System.out.println(Object o)

// 该方法就是将给定对象toString方法返回的字符串输出到控制台的

System.out.println(p); // day21\_Object.Point@15db9742

// 字符串与引用类型变量连接时,也是先调用其toString方法再与字符串连接的

String line = “p: ”+p;

System.out.println(line); // p: day21\_Object.Point@15db9742

// 如果重写toString方法

// 通常我们自己定义的类的toString方法被调用,就应当重写该方法.

// 重写没有固定要求.原则上返回的字符串中应当包含当前对像的属性信息

public String toString(){

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

}

### 2) **重写equals方法**

// Object提供了equals方法,定义为 boolean equals(Object obj)

// 设计意图是比较当前对像与参数对象是否相同

// 但是Ojbect实现这个方法时,内部用: this == obj进行判定,所以若不重写这个方法,没有实际意义

Point p1 = new Point(1,2);

Point p2 = new Point(1,2);

System.out.println(p1 == p2); // false

System.out.println(p1.equals(p2)); // false → true

// 如果**重写equals方法**

public boolean equals(Object obj){

// 重写Object提供的equals方法,判断当前对象对参数对象的内容是否相同

// 判断内容实际上就是判断属性值,但没有要求必须所有属性的值都相同刚才认定为相同,可根据实际需求而定

if (obj == null){

return false;

}

if (obj == this){

return true;

}

if (obj instanceof Point){

Point p = (Point)obj;

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

}

return false;

}

// 上方的p1.equals(p2)变为true

// 对于引用类型变量而言,由于保存的是地址,所有"=="做值比较时,比较的是两个变量指向的是否为"同一个对象"

// 而equals比较的是两个对象的内容是否相同

// java提供的类中很多都已经重写了equals方法,比如String

# 包装类

**包装类是为了解决基本类型不能直接参与面向对象开发的问题,使得基本类型可以以对象的形式保存**

### 1) **包装类**

int d = 1;

Integer i1 = new Integer(d);

Integer i2 = new Integer(d);

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

System.out.println(i1.equals(i2)); // true,integer类中已重写equals

// 建议使用valueOf将基本类型转化为包装类

Integer i3 = Integer.valueOf(d);

Integer i4 = Integer.valueOf(d);

System.out.println(i3==i4); // true,valueOf可使其指向同一对象,但要在byte范围内

System.out.println(i3.equals(i4)); // true

Double d1 = Double.valueOf(1.1);

Double d2 = Double.valueOf(1.1);

System.out.println(d1==d2); // false, double类型不成立

System.out.println(d1.equals(d2)); // true,integer类中已重写equals

// 从包装类转换为基本类型

double dou = d1.doubleValue(); // 以double形式返回指定的数值

System.out.println(dou);

### 2) **(String str)**

包装类提供了一个静态方法:parseXXX(String str),可以将字符串解析为对应的基本类型数据,前提是这个字符串内容正确描述了基本类型可以保存的值,否则会抛出异常

String line = "123.123";

int i = Integer.parseInt(line);

System.out.println(i); // 报错

double d = Double.parseDouble(line);

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

### 3) **自动拆装箱**

JDK5以后,出现了一个特性:自动拆装箱.该特性是编译器任何,而非虚拟机认可.

编译器在编译代码时如果发现需要在基本类型和对应包装类之间赋值,则编译器会自动补全代码在他们之间记性转换

// 触发自动拆箱特性,编译器会补充代码,将包装类转换为对应的基本类型:

// int d1 = new Integer(123)[.intValue()];

int d = new Integer(123); // 1.5以下报错

// 触发自动装箱特性,编译器会补全代码将基本类型高转换为包装类

// Integer in = Integer.valueOf(d)

Integer in = d;

2019.03.01 - Day 21

# File

**java.io.File**

**File的每一个实例用于表示文件系统中的一个文件或目录的**

使用File可以

1. 访问其表示的文件或目录的属性信息
2. 创建或删除文件,目录
3. 访问一个目录中的所有子项

但是不可以: 访问文件数据

### 1) **基本方法**

// 在这里指定路径时,应当指定相对路径,因为绝对路径的根不同的操作系统是不相同的,不利于跨平台

// 相对路径的好处是屏蔽了系统差异,但是具体相对哪里要根据运行程序时的环境而定

// "./"即当前目录

// 在eclipse中,"./"指的是当前程序所在项目的下项目目录

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

// 获取名字

String name = file.getName();

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

// 获取文件长度(字节量)

long length = file.length();

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

### 2) **使用File新建一个文件**

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

// 判断File表示的文件或目录是否已经存在了

if (!file.exists()){

// 创建该文件

file.createNewFile();

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

}else{

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

}

### 3) **使用File删除一个文件**

**file.delete();**

### 4) 创建一个新目录

File dir = new File("./demo");

**dir.mkdir();**

### 5) 创建多级目录

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

if (!dir.exists()){

**dir.mkdirs();**

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

}else{

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

}

### 6) **删除目录**

**dir.delete();**

删除目录的前提是这个目录是一个空目录

### 7) **获取当前目录下的所有子项**

File dir = new File(".");

// boolean isFile()

// 判断当前File表示的是否为文件

// boolean isDirectory()

// 判断当前File表示的是否为目录

if (dir.isDirectory()){

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

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

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

}

}

### 8) **删除当前目录下的所有内容**

public static void delete(File file){

if (file.isDirectory()){

// 先将该目录清空

// 1 获取目录中的所有子项

File[] subs = file.listFiles();

// 2 逐个删除子项

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

File sub = subs[i];

**// 递归调用**

delete(sub);

}

}

file.delete();

}

### 9) **文件过滤器**

listFiles的重载方法允许我们传入一个文件过滤器

File[] listFiles(FileFilter filter)

该方法会将当前File对象表示的目录中满足过滤器要求的子项获取回来

File dir = new File(".");

if (dir.isDirectory()){

// 该方法实际上会将dir中所有子项都获取到,并且每个子项都会经过给定的过滤器的accept方法

// 但是只有返回值为true的子项会被保留,最终返回

**FileFilter filter = new FileFilter(){**

**public boolean accept(File file){**

**String name = file.getName();**

**return name.startsWith(".");**

**}**

**};**

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

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

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

}

}

(See ListFilesDemo2.java)

# RandomAccessFile

**java.io.RandomAccessFile**

专门用来读写文件数据的API,其基于指针队列,可以对文件任意位置进行读写操作,编辑文件数据内容非常灵活

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

最后要写上**raf.close()**

### 1) **对文件进行写操作**

创建一个对当前目录下的test.dat文件操作的RAF

创建RAF时第一个参数为要操作的文件,第二个参数为模式,模式有两个比较常用的选项

r:只读模式

rw:读写模式

**raf.write(255);**

### 2) **对文件进行读操作**

**raf.read();**

### 3) **对文件进行复制操作**

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

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

int d = 0;

**while((d = ori.read())!= -1){**

**cp.write(d);**

**}**

2019.03.04 - Day 22

### 3.2) **对文件进行快速复制操作**

由于机械硬盘读写效率差,频繁进行读写赋值效率不高,我们可以通过提高每次读写的数据量,减少实际读写的次数,从而提高读写效率.单字节读写一般也可称为随机读写,一组字节读写也可称为块读写

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

RandomAccessFile des = new RandomAccessFile("music\_cp2.mp3", "rw");

// int read(byte[] data)

// 连续读取给定字节数组总长度的字节量,并将读取到的字节存入该数组中

// 返回值为实际读取到的字节量,我们应当根据实际读取的字节量判定该数组中本次读取数据的实际长度

// void write(byte[] data)

// 一次性将给定个字节数组中所有字节写出

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

// 将给定个字节数组从下标offset出的连续len个字节一些性写出

byte[] data = new byte[1024\*10]; // 期望读取的字节量

int len = -1; // 表示每次实际读取的字节量

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

des.write(data,0,len);

}

### 4) **向文件写出字符串**

String提供了将字符串转换为字节的方法: **byte[] getBytes()**,按照系统默认字符集转换(不推荐)

**byte[] getBytes(String csn)**,按照给定的字符集进行转换.

常用字符集名称: utf-8:英文一字节,中文3字节; gbk:英文1字节,中文2字节

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

String str = "一二三四五~";

byte[] data = str.getBytes();

raf.write(data);

### 5) **读取字符串数据**

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

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

raf.read(data);

String line = new String(data,"utf-8");

System.out.println(line);

### 6) **读写基本类型数据,以及基于指针的操作**

**raf.writeInt(num);**

**raf.seek(0); // 将指针移到指定位置**

**int d = raf.readInt()**

2019.03.05 - Day 23

# 流

java将流按照读写功能分为:

\* 输入流:用于从外界读取数据到程序中 (负责读操作)

\* 输出流:用于将数据从程序中写出到外界 (负责写操作)

java.IO.InputStream是所有字节输入流的超类,java.IO.OutputStream是所有字节输出流的超类

它们规定了所有流都必须具备的读写操作,因此无论读写什么设备的流,读写方法都是一样的

## **流的分类**

java将流分为节点流和处理流两大类

1)**节点流**:又称为低级流,是实际连接程序与另一端的"管道"

负责实际读写数据的流,读写操作一定是建立在节点流的基础上进行的

2)**处理流**:又称为高级流,不能独立存在,必须连接在其他流上

目的是当数据流经当前处理流时可以对数据进行某种加工处理操作,简化我们对数据的相关操作

3)**文件流**:是一对低级流,用于读写文件

## **流与RAF的区别**

文件流与RandomAccessFile都是用来读写文件数据的,虽然读写方法定义都相同(read,write方法),但是**底层实际的读写形式**完全不同

文件流是基于java标准IO对文件进行数据读写的,而流的读写形式为[顺序读写],即:读或写操作只能顺序向后,不可回退

RAF是基于指针的随机读写形式,可以操作指针对文件任意位置进行读写

所以文件流做不到对文件任意位置进行编辑操作,但是基于流连接可以轻松完成 RAF不容易办到的事情

## **流连接**

**流连接**:使用流读写时,串联若干的高级流,病最终串联到某个低级流对某设备进行读写操作的过程

流连接也是IO的精髓.

## **流的使用**

### 4.1) **文件输出流**

File 提供的构造方法: **FileOutputStrem(File file) / FileOutputStrem(String path)**

创建文件输出流时,若指定的文件已经存在,那么流闯将时会先将文件数据全部清除,所以这种方式为覆盖写模式.

如果在创建时再传入一个boolean值类型的参数,那么当前文件输出流就变为了追加模式.

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

/\*方法1\*/

String str0 = "好嗨哦~感觉人生已经到达了高潮~";

byte[] data = str0.getBytes("UTF-8");

fos.write(data);

/\*方法2\*/

String str = "test2";

fos.write(str.getBytes("utf-8"));

/\*方法3\*/

fos.write("We can add more into it.".getBytes("utf-8"));

### 4.2) **文件输入流**

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

byte[] data = new byte[1000];

int len = fis.read(data);

String line = new String (data, 0, len, "utf-8");

System.out.println(line);

### 4.3) **文件的复制**

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

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

byte[] data = new byte[100];

int len = -1;

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

fos.write(data, 0, len);

}

## **缓冲流**

**java.io.BufferedInputStream / java.io.BufferedOutputStream**

它们是一组高级流,将他们连接上以后可以提高读写效率,这样一来无论我们使用单字节读写还是块读写,都可以保证读写效率

### 5.1) **使用缓冲流完成文件复制**

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

BufferedInputStream bis = new BufferedInputStream(fis);

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

BufferedOutputStream bos = new BufferedOutputStream(fos);

int d = -1;

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

bos.write(d);

}

### 5.2) **缓冲输出流的缓冲区问题**

使用**.flush()**将缓冲区内的内容直接输出, .close()中也包含了.flush()

flush()方法是将缓冲区中已经缓存的数据一次性写出,频繁的调用会降低写出效率,但可改变写出数据的及时性

## **对象流**

当前类实例若想被对象流进行读写,那么必须实现接口: **java.io.Serializable**

### 6.1) **对象输出流写对象**

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

ObjectOutputStream oos = new ObjectOutputStream(fos);

Person person = new Person(name, age, gender, otherInfo); // 将属性写入

oos.writeObject(person);

写对象时每个流负责的工作分别为:

对象流首先将对象按照其结构转换为一组字节,这个过程称为:**对象序列化**

序列化后的字节在经过文件输出流最终写入文件(等于写入了磁盘做长久保存),这个过程为:**数据持久化**

writeObject方法时ObjectOutputStream提供的方法,作用是将给定的对象按照其结果转换为一组字节,然后再将字节通过其连接的流写出

实际转换的自己比该对象保存的内容要大,因为还要包含该对象的结构信息等内容

### 6.2) **对象输入流**

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

ObjectInputStream ois = new ObjectInputStream(fis);

Person person = (Person)ois**.readObject();**  // 进行对象的反序列化

System.out.println(person);

(See Person.java & OOSDemo & OISDemo.java)

### 6.3) **transient关键字**

被其修饰的属性在改对象进行序列化时值会被忽略

忽略不必要的属性值可以达到对象序列化时的瘦身操作,减少资源的开销.

2019.03.07 - Day 24

## **字符流**

java将流按照读写数据的单位划分了两类:

字节流: 以字节为单位读写数据,超类:**InputStream, Outputstream**

字符流: 以字符为单位读写数据,超类:**Reader, Writer**

字符流底层实际还是按照字节形式读写,但是字符与字节的转换操作字符流自行完成,所以字符流只适合读写文本数据

### 7.1) **转换流**

**java.io.OutputStreamWriter / java.io.InputStreamWriter**

转换流是一对高级流,是可以连接在字节流上的字符流,实际应用中我们读写文本数据是通常不会直接操作这对流,但是它们在流连接中是非常重要的一环

1. 写文本数据 **void write(String str);**

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

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

osw.write("one");

osw.write("two");

1. 读取文本数据 **int read();**

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

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

// 每次读取100个字符

char[] data = new char[100];

int len = isr.read(data);

String str = new String (data, 0, len);

System.out.println(str);

// 每次读取1个字符

int d = -1;

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

System.out.print( (char)d );

}

### 7.2) **缓冲字符流**

**java.io.BufferedWrite / java.io.BufferedReader**

**PrintWrite(String path) / PrintWrite(File file)**

缓冲字符流内部有缓冲区,读写字符效率高,并且可以按行读写字符串

但是比较常用的缓冲字符输出流为PrintWrite,它内部链接BufferedWriter作为缓冲那个加速,而它自身提供了自动行刷新操作

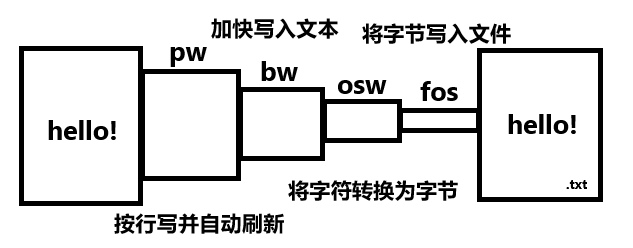
PrintWriter pw = new PrintWriter("pw.txt","utf-8");

pw.println("three.");

pw.println("four.");

### 7.3) **在流连接中使用PW**

在流连接中创建PW时,构造方法允许我们再传入一个boolean型参数,当这个值为true时,那么当前PW就具有了自动行刷新功能,即:当我们调用println方法后,会自动flush.但需要注意的是print方法不会自动flush



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

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

BufferedWriter bw = new BufferedWriter(osw);

PrintWriter pw = new PrintWriter(bw, true); // 当第一个参数为流时,true可以自动行刷新

pw.println("");`

### 7.4) **缓冲字符输入流(按行读取字符串)**

**String readLine();**

读取一行字符串,该方法会连续读取若干字符,当读取到换行符时停止,并将改换行符之前的内容以一个字符串形式返回,注意返回的字符串中不包含最后的换行符

如果返回值为null,则表示流读取到了末尾(若是读取文件则表示文件读取到了末尾)

FileInputStream fis = new FileInputStream("./src/day25\_IO/BRDemo.java");

InputStreamReader isr = new InputStreamReader(fis);

BufferedReader br = new BufferedReader(isr);

String line = null;

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

System.out.println(line);

}

# 异常处理机制

当程序中抛出一个异常后,程序从程序中导致异常的代码处跳出,java虚拟机检测寻找和try关键字匹配的处理该异常的catch块,如果找到,将控制权交到catch块中的代码,然后继续往下执行程序,try块中发生异常的代码捕获被重新执行.如果没有找到处理该异常的catch块,在所有的finally块代码被执行和当前线程的所属的ThreadGroup的uncaughtException方法被调用后,遇到异常的当前线程被终止.

## **异常的捕获和处理**

语法:

try{

代码片段

}catch(XXXException e){

解决try代码片段中出现XXXException的处理代码

}

### 1.1) **Throwable, Error和Exception**

Exception: 由网络故障/文件损坏/设备错误/用户非法输入等情况导致的异常

Error: java运行时环境出现的异常(如jvm内存资源耗尽等)

### 1.2) **try - catch机制**

catch**可以定义多个**,针对不同异常有不同处理方式时,我们可以分别捕获这些异常进行处理

try{

...

// if Exception

...

// try语句块中如果某句代码抛出了异常,那么try语句块**该句代码后面的内容都不会运行**

// 子类型异常在前,父类型异常在后,这样的顺序依次捕获,否则编译不通过

}catch(NullPointerException e){

...

}catch(StringIndexOutOfBoundsException e){

...

}catch(Exception e){

// 可以在**最后一个catch捕获Exception**,避免因为一个未捕获的异常导致程序中断

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

}

### 1.3) **finally块**

finally块是异常处理机制的最后一块,可以直接跟在try或者最后一个catch之后

finally可以保证只要代码执行到try当中,无论try语句块中的代码是否抛出异常,finally块中的内容都必定执行,所以我们通常将释放资源的这类操作放在这里确保运行

### 1.4) **在IO操作中使用异常处理**

对FileOutputStream进行异常处理

FileOutputStream fos = null; // 若定义在try内,finally中无法接受到fos,必须定义在外

try {

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

fos.write(1);

} catch (IOException e) {

e.printStackTrace();

} finally{

try{

if (fos != null){

fos.close();

}

}catch(IOException e){

e.printStackTrace();

}

}

### 1.5) **自动关闭**

JDK7之后的这个特性允许我们将需要关闭的内容定义在try后面的"()"当中,但是需要注意,只有实现了Autocloseable接口的类才可以在这里定义,剩下的还是应当在try语句块中定义,否则会出现编译不通过的情况.该特性是编译器认可,当编译成class文件后,还是在finally中将流关闭的

### 1.6) **throw/throws关键字**

1) throw关键词

当程序发生错误而无法处理时,会抛出对应的异常对象,若想自行抛出异常,例如在异常处理结束后,再将异常抛出,让下一层异常处理块来捕捉,可以使用throw关键词,并生成指定的异常对象后抛出

2) throws关键词

使调用这个方法的上层方法来统一处理异常

通常一个方法中使用throw抛出什么异常,就要在方法声明是使用throws声明这个异常的抛出,只有RuntimeException是个例外

public void setAge(int age) throws Exception{

if (age < 0 || age > 120){

throw new Exception("Ilegal Age.");

}

this.age = age;

}

当我们调用一个含有throws声明异常抛出的方法时,编译器要求我们必须处理该异常.

处理异常的方式有两种:

1. 使用try-catch捕获并处理抛出的异常类型
2. 在当前方法上继续使用throws声明异常的抛出(具体使用哪种,结合实际业务需求而定)

public static void main(String[] args) {

Person p = new Person();

try {

// //

p.setAge(1220);

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println(p.getAge());

}

### **子类重写父类含有throws声明异常抛出的方法时对throws的重写规则**

1. 完全相同
2. 仅抛出部分异常
3. 不抛出异常
4. 可以抛出父类方法中抛出异常的子类型异常

2019.03.08 - Day 25

## **异常的常见方法**

Java的异常可分为可检测异常,非检测异常

可检测异常: 可检测异常经编译器验证,对于声明抛出异常的任何方法,编译器将强制执行处理货声明规则,不捕捉这个异常,编译器就通不过,不允许编译

非检测异常: 非检测异常不遵循处理或者声明规则.在产生此类一次样时,不一定要采取任何适当操作,编译器不会检查是否已经解决了这样一个异常

### 2.1) **Java异常API**

**RuntimeException** 非检测异常

IllegalArgumentException: 抛出的异常表明向方法传递了一个不合法货不正确的参数

NullPointerException: 当程序试图在需要对象的地方使用null时,抛出该异常

ArrayIndexOutOfBoundsException: 当使用的数组下班超出数组允许范围时,抛出该异常

ClassCastException: 当试图将对象强制转换为不是实例的子类时,抛出该异常

NumberFormatException: 当应用程序试图将字符串转换为一种数值类型,但该字符串补鞥呢转换为适当格式时,抛出该异常

(See ExceptionApiDemo.java)

### 2.2) **Exception常用API**

1) **printStackTrace** - 输出错误信息,用来跟踪异常事件发生时执行堆栈的内容.

System.out向控制台输出能让和System.err向控制台输出内容

实际上在两个线程上进行的并发输出,所以使用这两个流输出时,控制台会出现串行的情况

e.printStackTrace()这个方法就是使用System.err向控制台输出的错误信息

try{

...

}catch(Exception e){

e.printStackTrace(); // 输出执行堆栈信息

}

2) **getMessage** - 得到有关异常事件的信息

System.out.println(e.getMessage());

3) **getCause** - 检索导致异常的原因

## **自定义异常**

**自定义异常通常是用来定义我们程序中一些业务逻辑错误,**比如:年龄不合法异常

自定义异常需要

1. 定义异常的类名,这个最重要.应当做到通过该异常的类名能大致判断出因为什么原因会引起这个异常
2. 需要继承Exception,至少是它的子类
3. 提供序列化版本号(eclipse生成即可)
4. 提供所有构造方法(与Exception定义的构造方法一样)这个也可以通过eclipse生成

(See Day26\_Exception other twoo)

# TCP通信

## **Socket原理**

### 1.1) **Socket简介**

Socket通常称为"套接字",用于描述IP地址和端口,是一个通信链的句柄.在Internet上的主机一般运行了多个服务软件,同时提供几种服务.每种服务都打开一个Socket,并绑定到一个端口上,不同的端口对应于不同的服务

**java.net.Socket** 套接字

Socket封装了底层TCP协议的通讯细节,使用它可以通过TCP协议与服务端建立连接,并以两条流进行IO操作完成与服务端的数据交换

应用程序通过"套接字"向网络发出请求或者应答网络请求.Socket和ServerSocket类位于java.net包中.SeverSocket用于服务端,Socket时建立网络连接时使用的.在连接成功时,应用程序两端都会产生一个Socket实例,操作这个实例,完成所需的会话

**Socket封装了底层TCP协议的通讯细节,使用它可以通过TCP协议与服务端建立连接,并以两条流进行IO操作完成与服务端的数据交换**

### 1.2) **实例化Socket**

**Socket(String host, int port)**

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

第一个参数为服务端的IP地址信息

第二个参数为服务端开启的服务端口

需要注意,这里实例化Socket的过程就是连接服务端的过程,若指定的地址和端口对应的服务端没有响应则会抛出异常

通过指定的IP地址可以找到网络上服务端所在的计算机,通过端口则可以连接到该机器对应的服务端应用程序

### 1.3) **发送字节数据**

Socket提供的方法: **OutputStream getOutputStream()**

通过Socket获取的字节输出流写出的数据会通过网络发送给远端计算机,对于客户端这边而言,远端那就是服务端

OutputStream out = socket.getOutputStream();

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

BufferedWriter bw = new BufferedWriter(osw);

PrintWriter pw = new PrintWriter(bw,true);

### 1.4) **ServerSocket**

1) 运行在服务端的ServerSocket的作用

① 向系统申请服务端口,客户端就是通过这个端口与服务端建立连接的

② 监听该端口,这样一来客户端通过这个端口连接时,ServerSocket会自动实例化一个SOkcet与客户端建立连接,这样服务端与客户端各有一个Socket,就可以进行双向通讯

2) 实例化ServerSocket

**server = new ServerSocket(8088);**

需要制定服务端口,该端口不能与当前系统运行的其他应用程序已经申请的端口冲突

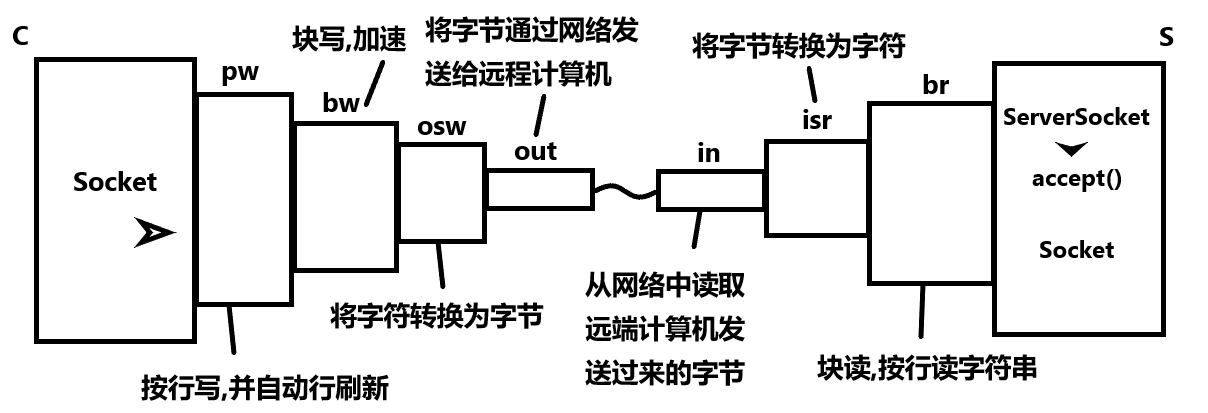
### 1.5) **建立与客户端的连接**

ServerSocket提供方法: **Socket accept()**

这是一个阻塞方法,调用后程序会"卡住",这时就开始监听服务端口,

直到一个客户端通过这个端口建立连接方法才会执行完毕并返回Socket,通过这个Socket实例就可以与该客户端进行交互了,多次调用accept方法可以接受多个客户端的连接

## **Socket通信模型**



2019.03.11 - Day 26

# 多.基础

## **进程与线程**

**进程**(process)**:** 操作系统中运行的一个任务,是一块包含了某些资源的内存区域

**线程**(thread): 操作系统里有进程把它的工作划分为一些功能单元,进程所包含的一个或多个执行单元称为线程

线程只能归属于一个进程并且它只能访问该进程所拥有的资源,当操作系统创建一个进程后,该进程会自动申请一个名为主线程或首要线程的线程

一个线程是进程的一个顺序执行流,同类的多个线程共享一块内存空间和一组系统资源,线程本身有一个工层序执行时的堆栈,线程在切换时符合小,因此线程也被称为请复核进程,一个进程中可以保护多个线程

## **创建线程**

Thread类是线程类,每一个实例表示一个可以并发运行的线程,我们可以通过集成该类并重写run方法来定义一个具体的线程.

### 2.1) **使用Thread创建并启动线程**

1. 重写run方法: 目的是定义该线程要执行的逻辑
2. 启动线程时调用线程的start()方法: 将当前线程纳入线程调度,使当前线程可以开始并发运行,当线程获取时间片段后悔自动开始执行run方法中的逻辑

线程纳入整个调度器后,只能被动的等待分配CPU时间片,得到时间片后,CPU变回运行该线程的任务代码,时间片用完后,CPU离开,此时线程调度会再分配给某个线程使其运行

线程调度器分配时间片的概率是一样的,但是所有并发运行的线程不保证一人一次这样均匀的分配时间片

public class ThreadDemo1 {

public static void main(String[] args) {

Thread t1 = new MyThread1();

Thread t2 = new MyThread2();

t1.start();

t2.start();

}

}

class MyThread1 extends Thread{

public void run(){

...

}

}

class MyThread2 extends Thread{

...

}

优点:创建简单,当临时需要执行某个任务时使用这种方式创建更直接

缺点:由于java是单继承的,这会导致我们继承了线程后就不能再去继承其他类去复用方法,实际开发非常不便

定义线程的同时重写run方法,将任务也一同定义处理,这会导致线程与任务有一个必然耦合关系,不利于线程的重用

### 2.2) **使用Runnable创建并启动线程**

实现Runnable接口并重写run方法来定义线程体,然后在创建线程的时候将Runnable的实例传入并启动线程

这样做的好处在于可以将线程与线程要执行的任务分离开减少耦合,同时java是单继承的,定义一个雷实现Runnable接口这样的做法可以更好的实现其他父类或接口,因为接口是多继承关系

public class ThreadDemo2 {

public static void main(String[] args) {

// 实例化任务

Runnable r1 = new MyRunnable1();

Runnable r2 = new MyRunnable2();

// 创建两个线程

Thread t1 = new Thread(r1);

Thread t2 = new Thread(r2);

t1.start();

t2.start();

}

}

class MyRunnable1 implements Runnable{

public void run(){

...

}

}

class MyRunnable2 implements Runnable{

...

}

### 2.3) **使用匿名内部类创建并启动线程**

通过匿名内部类的方式创建线程,使用该方式可以简化编写代码的复杂度,当一个线程仅需要一个实例时常使用此方法

Runnable r1 = new Runnable() {

@Override

public void run() {

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

System.out.println("who?");

}

}

};

Thread t1 = new Thread(r1);

t1.start();

## **线程操作API**

### 3.1) **常见API**

1. **Thread.currentThread**方法 - 获取运行当前代码片段的线程

Thread current = Thread.currentThread();

1. **getId()**方法 - 获取线程的唯一标识

long id = current.getId();

1. **getName()**方法 - 获取线程名字

String name = main.getName();

1. **getPriority()**方法 - 获取线程的优先级

int priority = main.getPriority();

1. **isAlive()**方法 - 线程是否处于活动状态

boolean isAlive = main.isAlive();

1. **isDaemon()**方法 - 线程是否为守护线程

boolean isDaemon = main.isDaemon();

1. **isInterrupted()**方法 - 线程是否已经中断

boolean isInterpreted = main.isInterrupted();

### 3.2) **线程优先级**

线程启动后纳入到线程调度,线程时刻处于被动获取CPU时间片而无法主动获取.我们可以通过调整线程的优先级来最大程度的干涉线程调度分配时间片的几率

理论上优先级越高的线程获取CPU时间片的次数越多

线程优先级有10个等级,分别用正数1-10来表示

其中1为最低优先级,10为最高优先级,5为默认

调用线程的方法: **setPriority()**方法来设置优先级

### 3.3) **sleep()方法**

**static void sleep(long ms)**

这是一个静态方法,当一个线程执行了这个方法后就会进入阻塞状态,并阻塞指定的毫秒,当超时后,线程会自动回到Runnable装订等待再次获取时间片并发运行

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

\* sleep方法要求处理中断异常:InterruptedException

当一个线程调用sleep方法处于阻塞状态的过程中,这个线程的中断方法interrupt被调用时,

则sleep方法会抛出中短一长,此时该线程的睡眠阻塞被打断

(See SleepDemo.java)

### 3.4) **setDaemon()方法**

**setDaemon(boolean on)**

**守护线程**又称为后台线程,默认创建出来的线程都是普通线程,守护线程使用上与普通多线程没有区别,其特点是,当进程中只剩下守护线程时,所有守护线程强制终止.

GC就是跑在守护线程上的,我们也可以将某些会一致运行的任务,并且当主要业务执行完毕后可以跟着一同结束的就放在守护线程上运行即可

(See DaemonThreadDemo.java)

2019.03.12 - Day 27

### 3.5) **void join()**

**download.join();** 目前方法在download方法所属线程后等待

线程提供了一个方法: void join(),该方法允许执行这个方法的线程在该方法所属线程后等待,直到该方法所属线程结束后方可继续运行, 否则会一致处于阻塞状态, 所以join可以协调线程之间的同步运行

同步运行: 执行有先后顺序

异步运行: 执行没有先后顺序, 多线程就是异步运行的

JDK8之前有一个强制要求(JDK8不强制要求,但是若存在使用不当情况,编译器仍然会编译失败):

当一个方法的局部内部类中引用了这个方法的其他局部变量时,要求这个变量必须声明为final,这源自JVM的内存分配问题

## **线程同步**

### 4.1) **sychronized关键字**

当一个方法使用关键字synchronized修饰后,这个方法称为同步方法,即多个线程不能同时进到方法内部执行

将多个线程并发执行操作临界资源改为同步执行就可以有效的解决多线程的并发安全问题

### 4.2) **同步块**

语法:

**synchronized(同步监视器对象){**

**需要同步运行的代码片段**

**}**

同步块可以更准确的控制需要同步运行的代码片段,有效的缩小同步范围可以保证并发安全的前提下尽可能的提高并发的效率

同步块要求指定一个同步监视器对象,即:上锁的对象

这个对象可以是java中任何对象,但是必须保证多个需要同步运行此代码片段的线程看到的对象是同一个

### 4.3) **静态方法锁**

**public synchronized static void dosome() {}**

这个方法一定具有同步效果.

静态方法上使用的同步监视器对象为这个类的"类对象",每个java定义的类都只有唯一的一个类对象(class类型的实例),而类对象后面的反射之后会介绍

(See SyncDemo3.java)

### 4.4) **互斥锁**

当使用synchronized锁定多个代码片段,并且它们指定的同步监视器对象是同一个时,那么这些代码片段之间就是互斥的,多个线程不能同时在这些代码片段中运行

class Boo{

// 当两方都有synchronized时,为互斥

public synchronized void methodA(){

try {

Thread t = new Thread();

System.out.println(t.getName()+":doingA");

Thread.sleep(1000);

System.out.println(t.getName()+":finishA");

} catch (Exception e) {

// TODO: handle exception

}

}

public synchronized void methodB(){

try {

Thread t = new Thread();

System.out.println(t.getName()+":doingB");

Thread.sleep(1000);

System.out.println(t.getName()+":finishB");

} catch (Exception e) {

// TODO: handle exception

}

}

}

2019.03.13 - Day 28

Java SE02

# 集合框架

**java.util.Collection**集合

Collection是所有集合的顶级接口,规定了所有集合都必须具备的相关方法

集合与数组功能相似,用来保存一组元素,并提供了对元素操作的相关方法,使用更便捷

集合中存储的都是引用类型元素,并且集合只保存每个元素对象的引用,而非将元素对象本身存入集合

## **Collection**

### 1.1) **List和Set**

Collection下有两个常见的集合类型

java.util.List: 可重复集,并且有序

Collection c = new ArrayList();

java.util.Set: 不可重复集

Collection c = new HashSet();

Set集合不可以存放重复元素,而元素是否重复是依靠元素自身equals比较的结果而定

### 1.2 **集合的基本方法**

1) **add方法**

boolean add(E e): 向当前集合中添加给定元素,成功添加后则返回true

**addAll方法:**

addAll(E e): 将给定集合中的所有元素添加到当前集合中,根据被加集合的属性决定是否可以重复

2) **size方法**

int size(): 返回当前集合的元素个数

1. **isEmpty方法**

boolean isEmpty(): 判断当前集合是否为一个空集,集合中不含有任何元素

1. **clear方法**

.clear(): 清空集合元素

1. **remove方法**

boolean remove(E e): 从集合中删除给定元素,删除的是集合中与给定equals比较为true的元素.一次只删一个,删除后结束

**removeAll方法**

.removeAll(E e): 删除当前集合中与给定集合的共有元素(删交集)

1. **contains方法**

boolean contains(E e): 判断当前集合是否包含给定元素,判断也是根据元素equals比较的

通常有必要重写equals()保证contains()方法的和理解过

**containsAll方法**

boolean containsAll(Collection c): 判断当前集合是否包含给定集合的所有元素

### 1.3) **HashSet**

HashSet是最常用的Set集合实现类

Collection c = new HashSet();

## **Iterator**

Collection提供了统一的遍历集合元素的操作:迭代器模式

**Iterator iterator()**: 该方法会返回一个用于遍历该集合的迭代器,使用这个迭代器便可以遍历当前集合元素

java.util.Iterator接口: 它是所有迭代器的顶级接口,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.迭代器遍历集合遵循的规则是: 问,取,删. 其中删除元素不是遍历过程的必要操作

Iterator it = c.iterator();

### 2.1) **基本方法**

1. **hasNext()方法**

boolean hasNext():判断集合是否还有下一个元素可以迭代

1. **next()方法**

E next(): 获取集合下一个元素

3) **remove方法**

对迭代器使用remove方法,删除迭代器刚取出的对象

Collection c = new ArrayList();

c.add("one");

c.add("#");

Iterator it = c.iterator();

while (it.hasNext()) {

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

System.out.println(str);

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

it.remove();

}

}

### 2.2) **增强for循环**

JDK5之后推出了一个特性: 增强for循环,也称之为新循环,for each

新循环不是用来取代传统for循环的操作,而仅用来遍历集合或数组使用,编译器中与传统for循环一致

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

**for (String str: array)**{ // for (int i = 0; i < array.length(); i++){

// 自包含String str = array[i]与自加

System.out.println(str);

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

String str = array[i];

System.out.println(str);

}

}

所以在遍历的过程中不能使用集合的方法增删元素

## **泛型机制**

泛型又称为参数化类型,允许我们在使用一个类时指定其属性,方法参数以及返回值的类型,使得代码的灵活度提高. 泛型应用最广的地方就是集合,用来约束集合的元素类型

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

# 集合操作 - 线性表

**java.util.List**

List是Collection常用的子接口,是可以重复的集合. 特点是有序,提供了一套通过下标操作元素的方法

## **List**

### 1.1) **ArrayList和LinkedSet**

1. **java.util.ArrayList**: 内部使用数组实现

ArrayList查询性能更好,增删元素慢

1. **java.util.LinkedList**: 内部使用链表实现

LinkedList增删元素性能好,尤其首尾增删元素性能最好,但是查询元素效率慢

对性能没有极端苛刻情况下通常使用ArrayList

### 1.2) **get和set**

1) **E get(int index)**

str = list.get(i): 获取指定位置对应的元素

2) **E set(int index, E e)**

String old = list.set(1, "2") // 将1号位的元素替换为"2"

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)

2019.03.14 - Day 29

### 1.3) **add和remove**

1) **void add(int index, E e)**

list.add(1,"2"): 将给定元素插入到指定位置

2) **E remove(int index)**

String old = list.remove(2): 删除并返回指定位置对应的元素

### 1.3) **对子集的操作**

1) **list.subList(fromIndex, toIndex)**

List<Integer> sub = list.subList(2, 8): 获取指定范围内的子集(含头不含尾)

2) **通过subList删除元素**

list.subList(2,9).clear(): 取出子集并clear (**对子集的操作就是对原集合的操作**)

### 1.3) **集合与数组的转换**

1) 集合转换为数组

Collection定义的方法: **toArray**,可以将当前集合转换为一个数组

String[] arr = list.toArray(new String[list.size()]);

2)数组转换为集合

数组的工具类: Arrays提供了一个静态方法**asList**,可以将给定的数组转换为一个List集合

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

1. 注意事项

由于数组是定长的,所以集合增删元素的操作是不支持的,会抛出异常,若想对数组转换的集合元素增删,那只能自行创建一个新的集合并包含该集合元素

所有集合都提供了参数为Collection的构造方法,可以在创建当前集合的同时包含给定集合中的所有元素

List<String> list1 = new ArrayList<String>(list);

## **集合的排序**

### 2.1) **Collections.sort()**

java.util.Collection提供了一个静态方法sort,可以对List集合进行自然排序(从小到大)

Random random = new Random();

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

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

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

}

System.out.println(list);

**Collections.sort(**list**)**;

System.out.println(list);

### 2.2) **重载sort()**

sort方法要求集合元素必须实现Comparable接口,java提供的常用类如包装类String等它们都实现了这个接口,定义了元素之间的大小规则.但是对于我们自定义类型的元素,不建议实现这个接口,因为这个的操作对我们的代码有侵入性,不利于维护

重载的sort方法要求再传入一个参数,是Compartor接口的实现类.该接口是用来单独定义一个比较器,为集合元素指定比较大小的规则.这样一来,该sort方法会将集合元素利用该比较器的比较规则两两比较,并按照从小到大的顺序将集合排序

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

@Override

public int compare(String o1, String o2) {

int len1 = o1.length();

int len2 = o2.length();

return len1 - len2;

}

**});**

## **队列和栈**

### 3.1) **队列 Queue**

队列Queue继承自Collection,是经典的数据结构之一,可以保存一组元素,但是存取元素必须顺序执行

常用实现类: java.util.LinkedList

1. 创建队列

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

1. **offer()**

queue.offer("5"): 入队操作,将元素添加到队列末尾

1. **poll()**

String str = queue.poll(): 出队操作,获取并删除当前队列的队首元素

1. **peek()**

str = queue.peek(): 获取队首元素,但是不做删除操作

1. 遍历队列

for(String s : queue){}

### 3.2) **双端队列 Deque**

双端队列Deque继承自Q ueue,是两端都可以进出队的队列,常用实现类:LinkedList

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

有入队(offer)和出队(poll)两个操作

### 3.3) **栈Stack**

栈可以保存一组元素,但是存取必须遵循先进后出原则,通常使用栈完成"后退"/"前进"这样的操作

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

有压栈(push)和弹栈(pop)两个操作

2019.03.15 - Day 30

# 查找表

**java.util.Map**

Map是java中非常经典的数据结构之一

Map的结构看起来像是一个多行两列的表格,其中左列称为:key,右列称为:Value

Map总是以Key-Value对的形式保存数据的,并且Map提供了根据Key获取其对应的Value的查找方法

Map有一个要求:Key是不允许重复的(Key的equals方法决定)

常用实现类:java.util.HashMap

HashMap称为散列表或哈希表,使用散列算法实现的Map,是当今世界上查询速度最快的额数据结构,其查询速度不收数据量影响,现如今所有网站的缓存都是用HashMap实现的

## **Map接口**

**Map<Key, Value> map = new HashMap<>();**

### 0.1) **基本方法**

1. **put()**方法

map.put(Key key, Value value): 将给定的key,value对保存到Map中,若给定的key在当前Map中已经存在,则是替换value操作,那么返回值就是原Key对应的value,否则返回值为null

1. **get()**方法

map.get(Object key): 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nul3) **size()**方法

map.size(): 获得Map的大小

4) **remove()**方法

map.remove(Object key): 删除给定的key所对应的键值对,返回值为key对应的value

5) **contains**方法

map.containsKey(Object key): 是否包含key,返回值为boolean

map.containsValue(Object value): 是否包含value,返回值为boolean

### 0.2) **Map的遍历**

遍历Map有三种方式:

1. **遍历所有key**

将当前Map中的所有key以一个Set集合形式返回

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

for(String s: keySet){}

1. **遍历所有的key-value对**

将当前Map中每一组键值对以一个Entry实例形式存放与Set集合后返回

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

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

String key = entry.getKey();

Integer value = entry.getValue();

}

1. **遍历所有的value(相对不常用)**

将当前Map中的所有Value以一个Collection集合的形式返回.

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

for (Integer value: values){}

2019.03.18 - Day 31

# XML

XML即可扩展标记语言(EXtensible Markup Language),是独立于软件和硬件的信息传输工具

应用于web开发的许多方法,常用于简化数据的存储和共享

## **XML语法**

1. 所有的处理指令都以<?开始,?>结束

文档开始: <?xml version="1.0" encoding="utf-8"?>

1. 大小写敏感
2. 必须有关闭标签
3. 元素必须正确嵌套
4. 实体应用:&lt = < ; &gt = > ; &amp; = & ; &apos = ' & &quot = "
5. CDATA段

<![CDATA [文本内容]]>: 文本内容出无论写什么都会被当做一个文本

## **XML解析**

1. SAX解析方式: 一种XML解析的替代方法,它逐行扫描文档,一遍扫描一遍解析

优点: 解析可以立即开始,速度快,没有内存压力

缺点: 不能对节点做修改

1. DOM解析方式: 在解析XML文档时,会把文档的所有元素,按照其出现的层次关系,解析成一个个Node节点

优点: 把XML文件在内存中构造属性结果,可以遍历和修改节点

缺点: 如果文件大,内存有压力,解析的时间会比较长

### 2.1) **使用DOM解析XML文档**

使用DOM4J解析XML的大致步骤

①创建SAXReader

②使用SAXReader读取指定的xml文件并生成Document对象

③通过Document对象获取根元素

④从根元素开始逐级获取子元素以达到XML文档数据的目的

### 2.2) **读取XML**

### 1) SAXReader读取XML文档

使用SAXReader需要导入dom4j-full.jar包

try{

// 创建SAXReader

**SAXReader reader = new SAXReader();**

// 读取指定文件

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

} catch(...){}

2) Document的getRootElement方法

**Element getRootElement()**: 用于获取根元素,Element每个实例用于表示XML文档中的一个元素(Element interface defines an XML element. An element can have declared namespaces, attributes, child nodes and textual content.)

Element root = doc.getRootElement();

3) 获取子标签

//将根元素下所有名为<mime-mapping>的子标签获取出来

List<Element> list = root.elements("mime-mapping");

for (Element ele: list){

Element keyEle = ele.element("extension"); // 子标签下标签设置为key

String key = keyEle.getTextTrim();

Element valueEle = ele.element("mime-type");

String value = valueEle.getTextTrim();

MIME\_MAPPING.put(key, value); // 放入MAP

### 2.3) **Element常用方法**

1) **getName()方法**

String getName(): 获取当前标签的名字

2) **getValue()方法**

String getValue(): 获取属性的值

3) **getText()方法**

String getText(): 获取当前标签中间的文本信息

4) **element方法**

Element element(String name): 获取当前标签中指定名字的子标签

5) **elements方法**

List elements(): 获取当前标签中所有子标签,返回的List集合中会包含若干的Element实例,每个实例是其中一个子标签

List elements(String name): 获取当前标签中所有同名子标签(指定的名字)  
6) **attribute方法**

Attribute attribute(int index): 获取当前元素的指定属性,index为索引,从0开始

Attribute attribute(String name): 获取当前元素的指定名字的属性

## **写XML**

### 3.1) **使用DOM4J生成XML文档**

写出一个XML文档的大致步骤

①创建一个Document对象,表示一个空白文档

②向Document对象添加根元素

③向根元素逐级添加子元素,以形成XML文档应有的树结构

④创建XmlWriter

⑤通过XmlWriter将该Document对象写出以形成XML文档

⑥将XmlWriter关闭

### 3.2) **具体步骤**

1) 构建Document对象

使用DocumentHelper创建一个Document实例(相等于新建一个XML?)

Document doc = **DocumentHelper.createDocument();**

2) 添加根标签

**Element addElement(String name):** 向当前文档中添加指定名字的根元素,并将其以一个Element实例形式返回,以便后续操作.只能调用一次(一个文档只能有一个标签)

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

1. 向根元素逐级添加子元素

for(Emp emp : empList){

Element empEle = root.addElement("emp"); // 向根元素中追加一个<emp>标签

Element nameEle = empEle.addElement("name"); // 向子元素继续添加标签

nameEle.addText(emp.getName()); // 可以和上一行合并

emp.addAttribute("id", "410106"); // 为emp元素添加属性id,值为420106

}

① **addAttribute方法**

Element addAtrribute(String name, String value): 向当前元素添加指定的属性及对应的值,返回值为当前元素

② **addText方法**

Element addText(String text): 向当前元素添加指定内容的文本

1. XMLWriter输出XML文档

XMLWriter writer = new XMLWriter();

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

writer.setOutputStream(fos);

writer.write(doc);

writer.close();

## 完成注册业务操作

(See WebServer\_v12)

1. 如果检测到用户访问reg.html,调用RegServlet方法,进行用户输入信息的解析存储
2. 在RegServlet中调用HttpRequest,将url中包含的用户名等信息解析,传送到RegServlet
3. 将用户信息写入user.dat文件,每个用户占用相同的字节
4. 设置Response响应注册成功页面(编写html文件)

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

// 将指针移动到文件末尾

raf.seek(raf.length());

// 写用户名

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

data = Arrays.copyOf(data, 32);

raf.write(data); // 写了32字节

// 写年龄

raf.writeInt(65);

// 设置Response响应注册成功页面

response.setEntity(new File("./webapps/myweb/reg\_success.html"));

}catch{...}

2019.03.19 - Day 32

# Java反射机制

java反射机制是一种动态机制,它允许我们家族一个类实例化一个类,调用方法或操作属性从编码期间确定转为在运行期间确定,这样做可以大大提高代码的灵活度,但是反射有更大的资源开销,所以不能过度依赖反射

## Class类

class类的每一个实例用于表示JVM已经加载的一个类.并且在JVM内部每个被加载的类都有且只有一个Class的实例与之对应.通过类对象我们可以:获取其表示的类的名字,构造方法,方法,属性,并且可以快速实例化

获取一个类的类对象有以下方法:

1. 直接通过类的静态属性class获得.比如像获取String的类对象,我们可以

Class cls = String.class

1. 通过class的静态方法forName加载

Class cls = Class.forName("java.lang.String");

1. 通过类加载器ClassLoader

try {

// 加载String

Class cls = Class.forName("java.lang.String");

// 获取类名

String name = cls.getName();

System.out.println(name);

// 获取所有方法

Method[] methods = cls.getMethods();

// 获取本类定义的方法(不含超类继承的方法)

// Method[] methods = cls.getDeclaredMethods();

System.out.println(methods.length);

for (Method method: methods){

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

}

} catch () {}

### 1.2) **使用类对象快速实例化对象**

Class提供了实例化对象的方法: **Object newInstance()**

这个方法要求Class表示的类必须有无参构造方法,否则会抛出异常;若当前类没有无参构造方法,我们需要先通过类对象获取其定义的某种构造方法,然后通过构造方法对象来进行实例化

1) 加载需要实例化对象的类的类对象

2) 通过类对象实例化

System.out.println("Input the class name you want:");

Scanner scan = new Scanner(System.in);

String className = scan.nextLine();

Class cls = Class.forName(className);

Object obj = cls.newInstance();

System.out.println(obj);

### 1.2) **使用反射调用某个类的成员方法**

// 利用反射

Class cls = Class.forName("day33\_Reflect.Person"); // 包名.类名

Object o = cls.newInstance();

1. 通过类对象获取要调用的方法

**Method method = cls.getMethod(**"sayHello"**)**;

// 调用指定对象的方法

**method.invoke(**o**)**;

1. 调用有参数的方法

Method m = cls.getMethod("sayHi", String.class);

m.invoke(o, "Doris");

3) 调用私有方法

Method m2 = cls.getDeclaredMethod("say");

// 在访问私有方法前设置访问操作(不设置直接调用会报错)

m.setAccessible(true);

m.invoke(o);

# 线程池

线程池主要用来:

1. 控制线程数量(防止CPU过度切换和过多的资源消耗)
2. 重用线程

// 线程池里最多允许两个线程同时执行

ExecutorService threadPool = Executors.newFixedThreadPool(2);

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

Runnable runn = new Runnable() {

public void run() {

try {

Thread t = Thread.currentThread();

System.out.println(t.getName()+"doing");

Thread.sleep(5000);

System.out.println(t.getName()+"done");

} catch (...){...}

}

};

threadPool.execute(runn);

System.out.println("将一个任务交给了线程池");

}

threadPool.shutdown(); // 将缓存的所有线程跑完后关闭

threadPool.shutdownNow(); // 立即关闭所有线程

2019.03.26 - Day 33

# Date & Calender

**java.util.Date**

Date的每一个实例用于表示一个具体的时间点.内部使用一个long值,该值表示的时自: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

Date由于存在时区等问题,所以大部分操作时间的方法都被声明为过时的(开发中不建议使用的方法),因为我们现在仅使用Date表示一个时间

## Date

### **Date及其常用API**

1. Date类

导入Date包,直接获得当前时间

Date now = new Date();

System.out.println(now);

1. setTime和getTime方法

① long getTime(): 获取当前Date内部维护的long值

long time = now.getTime();

② setTime(): 设置当前时间

now.setTime(0);

System.out.println(now);

### **Simple Date Format**

**java.text.SimpleDateFormat:** 可以将String与Date按照指定的格式进行相互转换

### 日期转换为字符串

**String format(Date date)**: 将给定的Date对象表示的日期按照当前SDF指定的日期格式转换为字符串

// 2019-03-27 10:25:33

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

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH-mm:ss E a");

String line = sdf.format(now);

1. 字符串转换为Date对象

**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

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

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date date = sdf.parse(line);

System.out.println(date);

## Calendar

**java.util.Calendar 日历类**

Calender是一个抽象类,常用实现类为:GregorianCalenda r

日历类是可以操作时间的API,Date操作时间的大部分方法被声明为过时的,取而代之的是Calendar

2.1) **实例化Calendar**

Calendar提供了静态方法getInstance(),该方法可以根据当前系统所在时区获取一个使用的实现类,大部分地区获取的都是GregorianCalendar.默认创建的日历也是表示当前系统时间的

Calendar cal = Calendar.getInstance();

### 2.2) **Calendar与Date互转**

1. getTime(): 获得Cal表示的日期

Date date = cal.getTime();

1. void setTime(Date date): 使当前Calendar调整为Date所表示的日期

cal.setTime(date);

### 2.3) **获取指定时间分量所对应的值**

int get(int field): 获取当前Calendar指定时间分量所对应的值,此方法替代了Date中的getXXX方法

Calendar cal = Calendar.getInstance()();

int year = cal.get(Calendar.YEAR); // 获取年

int month = cal.get(Calendar.MONTH)+1; // 月(月从0开始)

int day = cal.get(Calendar.DAY\_OF\_MONTH); // 日

// DATE: 同day; HOUR\_OF\_DAY:时; MINUTE:分; SECOND: 秒

// 获取指定时间分量所允许的最大值

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

### 2.4) **调整时间分量的值**

1) void **set(**int field , int value**)**: 调整当前Calendar指定时间分量的值

cal.set(Calendar.YEAR, 2008); // 调整年

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

cal.set(Calendar.MONTH, Calendar.AUGUST); // 月

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

// 如果两个时间分量有冲突,会产生覆盖,调整以后须调用getTime方法

2) void **add(**int field, int amount**)** :对给定的时间分量加上给定的值,若给定的值为负数,则是减去

调用后calendar表示的时间会更新,不会出现set方法存在的问题

// 查看3年2个月零30天以后的那周五

cal.add(Calendar.YEAR, -3);

cal.add(Calendar.MONTH, -2);

cal.add(Calendar.DAY\_OF\_MONTH, -30);

cal.getTime();

cal.set(Calendar.DAY\_OF\_WEEK, 6);

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

# Lambda表达式

Lambda表达式是JDK8推出的特性,可以以更简单的方式创建匿名内部类

语法:

([实际参数])->{

方法体

}

需要注意,使用lambda创建的匿名内部类所实现的接口只能有一个抽象方法

### 无参数的方法

原方法:

Runnable r1 = new Runnable(){

public void run(){

System.out.println("hello");

}

};

使用lambda表达式:

Runnable r2 = () -> { // ()相当于run的括号,所以只能有一个抽象方法

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

};

### 有参数的方法

### 原方法:

### File file = new File(".");

// 获取当前目录下所有文本文件

FileFilter filter = new FileFilter(){

public boolean accept(File file){

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

}

};

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

for (File sub : subs){

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

}

使用lambda表达式:

// 获取所有.dat文件

FileFilter filter1 = (f) -> {

return f.getName().endsWith(".dat");

};

还可以继续优化:

FileFilter filter1 = (f) -> f.getName().endsWith(".dat");

输出:

subs = file.listFiles(filter1);

for (File sub : subs){

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

}

2019.03.27 - Day 34