前沿技术:Netty

RPC框架,远程过程调用入手,逐步加强

Alt+shift+r 修改与当前相等的单词

目录

[Java串讲 1](#_Toc10657064)

[1 基础语法 1](#_Toc10657065)

[2 面向对象 1](#_Toc10657066)

[3 基础API 1](#_Toc10657067)

[4 集合 1](#_Toc10657068)

[5 异常 1](#_Toc10657069)

[6 io 1](#_Toc10657070)

[7 线程 1](#_Toc10657071)

[8 网络 1](#_Toc10657072)

[9 反射 1](#_Toc10657073)

Java串讲

# 基础语法

* 数据类型
  + 基本类型
  + 引用类型
* 八种基本类型
  + 字节量
  + 字面值规则(5条)
    - 235, int
    - byte a = 127, byte

byte a = 128, 错,int

* + - 3.14, double
    - L F D
    - 0b 0x 0 \u
  + 运算规则(5条)
    - 3/2, 1
    - byte a = 3;

a = a+a; 错,int+int

* + - Integer.MAX\_VALUE+1 得负数最小值
    - 2-1.9, 0.1000000000000009

4.35\*100, 434.99999999999994

* + - 浮点数特殊值

Infinity, 3.14/0

NaN, 0.0/0, Math.sqrt(-5)

* 运算符
  + +-\*/
  + %
  + == != > >= < <=
  + && || !
  + & | ^ ~
  + >> >>> <<
  + ++, --
  + ? :
  + =
  + += -= ^= >>=
  + ()
  + instanceof
* 流程控制
  + if
  + switch
    - 只能判断:
    - byte,short,char,int
    - enum
    - jdk1.7, String
  + for
  + while
  + do-while
  + break
  + continue
  + 循环命名

outer:

for() {

}

* 数组
  + 创建
    - int[] a = new int[6];
    - int[] a = {4,2,56,2};
    - a = new int[]{6,3,1,7,2};
  + a.length
    - java允许创建0长度的数组
  + 二维数组
    - 数组中存放数组
* 变量
  + 局部变量
    - 必须手动初始化
    - 作用域
  + 成员变量
    - 自动初始化
    - 访问范围受访问控制符控制
* 方法递归
  + 一步一步地简化问题,把问题简化成最简,再倒推求出结果

# 面向对象

* 什么是面向对象
  + 封装,继承,多态
* 封装
  + 类, 模板
  + 对象, 实例
  + 引用, 遥控器
  + 构造方法
    - 不定义,有默认
    - 构造方法重载
  + this
    - this.xxx
      * 特殊引用,引用当前实例的地址
    - this(...)
      * 构造方法之间调用
  + 方法重载 Overload
    - 同名不同参
  + private
    - 隐藏,便于修改维护
* 继承
  + 作用:代码重用,复用
  + 单继承
  + 子类实例
    - 父类实例和子类实例绑定,整体作为一个实例
    - 调用成员时,先找子类,再找父类
      * 编译期绑定
        + 变量,静态,私有
      * 运行期绑定
        + 实例方法
  + 方法重写 Override
    - 继承的方法重新定义,重新编写
    - 覆盖,改写
  + super
    - super.xxx()
      * 重写时,调用父类的同一个方法
    - super()
      * 调用父类构造方法
      * 默认 super()
      * 手动 super(参数)
* 多态
  + 作用:一致的类型
  + 向上转型,向下转型
  + instanceof
    - 对真实类型和父类型判断,都返回 true

Shape s = new Line();

s instanceof Line true

s instanceof Shape true

s instanceof Object true

s.getClass() == Line.class true

s.getClass() == Shape.class false

* 抽象类
  + 抽象方法
    - 作为通用方法,在父类中定义
    - 要求子类必须实现
* final
  + 常量,方法,类
* static
  + 静态属于类
  + 由类来执行静态成员
  + static {

类加载时,只执行一次

}

* 访问控制
  + public,protected,[default],private
* 对象创建过程
  + 10步
* 接口
  + 作用:结构设计工具,解耦合,隔离实现
  + 接口中只能定义
    - 公开的常量
    - 公开的抽象方法
    - 公开的内部类内部接口
  + 多实现
  + 接口之间多继承
* 内部类
  + 匿名内部类
  + 非静态内部类
    - 辅助外部实例,封装局部数据或局部运算逻辑
  + 静态内部类
    - 和普通的类没有区别
    - 设计选项,可选可不选
  + 局部内部类

# 基础API

* String
  + 常量池

String s1 = "abc";

//编译器优化成 "abcdef"

String s2 = "abc" + "def";

String s3 = "abcdef";

s2 == s3 true

* + 不可变

String s1 = "aaa"

String s2 = "bbb"

s1 + s2

StringBuilder sb = new StringBuilder(s1);

sb.append(s2);

sb.toString();

* StringBuilder,StringBuffer
  + append()
  + StringBuilder, 线程不安全,效率高
  + StringBuffer, 线程安全
* 正则表达式
  + 百度
* Integer
  + Integer.valueOf(6)

256个缓存对象,-128到127,

范围内,访问缓存对象

范围外,新建

* Date
* SimpleDateFormat

# 集合

* ArrayList
  + 数组默认初始长度10
  + 存满, 1.5倍
  + 访问任意位置效率高
  + 增删数据效率可能降低
* LinkedList
  + 双向链表
  + 首尾效率高
* HashMap
  + 使用 Entry[] 数组存放数据
  + 默认初始长度16
  + 哈希运算过程
    - key.hashCode()来计算下标 i
    - 新建 Entry 实例封装键值对,放入i位置
      * 空位置,直接放入
      * 有数据,依次equals()比较是否相等
        + 有相等的,覆盖值
        + 没有相等的,链表连接在一起
      * 负载率,加载因子 0.75
        + 新建翻倍长度新数组
        + 所有数据重新哈希运算,放入新数组
      * jdk1.8
        + 链表长度到8,转成红黑树
        + 减少到6,转回链表
* LinkedHashMap
* ConcurrentHashMap
* ConcurrentHashSet

# 异常

* 异常继承结构
* try-catch-finally
* throws
* throw
* 异常包装
* 自定义异常

# io

File

FileInputStream,FileOutputStream

ObjectInputStream,ObjectOutputStream

Serialiable

serialVersionUID

InputStreamReader,OutputStreamWriter

BufferedReader

PrintWriter

# 线程

* 线程创建
  + 继承 Thread
  + 实现 Runnable
* 线程方法
  + Thread.currentThread()
  + Thread.sleep()
  + Thread.yield()
  + start()
  + getName(),setName()
  + interrupt()
  + join()
  + setDaemon(true)
  + setPriority(优先级)
* 线程同步 synchronized
  + 步调一致地执行
  + synchronized(对象) {

}

抢指定对象的锁

* + synchronized void f() {

}

抢当前对象 this 的锁

* + static synchronized void f() {

}

抢类的锁

* 生产者消费者
  + 线程间通信
* 等待和通知
  + Object的方法
  + wait()
  + notify()
  + notifyAll()
  + 必须在 synchronized 代码内调用
  + 必须在加锁的对象上等待或发出通知
  + wait() 外面总应该是循环条件判断

# 网络

* ServerSocket
  + accept()
  + close()
* Socket
  + getInputStream()
  + getOutputStream()
  + setSoTimeout()
  + close()

# 反射

* 获得类对象
  + A.class
  + Class.forName()
  + a1.getClass()
* 新建实例
  + c.newInstance()

熟练掌握用法,

使用熟练的基础上,再研究底层原理,底层源码

# l 基础语法5天

# java

java最底层的东西是面向字节码编程和jvm

## 验证环境变量

WIN+R，输入 cmd，(打开命令行终端)

输入命令：

echo %JAVA\_HOME%

D:\Program Files\Java\jdk1.8.0\_191

path

PATH=D:\Program Files\Java\jdk1.8.0\_191\bin;C:\Program Files (x86)\Int......

java -version

java version "1.8.0\_191"

Java(TM) SE Runtime Environment (build 1.8.0\_191-b12)

Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)

## 编译

src目录下的HelloWorld.java 编译成 HelloWorld.class 字节码文件，存放在 bin 目录下

## 执行

先启动Java虚拟机（JVM）

JVM加载HelloWorld.class字节码文件

进入这个类的 main() 方法执行

## 八种基本类型

### 基本类型（8种）

基本类型变量，直接保存值

### 引用类型

引用类型变量，保存对象的地址

|  |  |  |
| --- | --- | --- |
| byte | 1 | -128到127，-27到27-1 |
| short | 2 | -215到215-1 |
| int | 4 | -231到231-1 |
| long | 8 | -263到263-1 |
| float | 4 | 单精度 |
| double | 8 | 双精度 |
| char | 2 | 字符，或字符的整数编码Unicode 编码 |
| boolean | 1 | 真    true 假    false |

### 基本类型最大最小值

Integer.MAX\_VALUE+1,负数最小值

MAX\_VALUE 正数最大值

提示:window->preference->java->editor->

\* content assist->auto activation

## 基本类型的字面值规则（5条）

1. 整数字面值，是int类型

2. byte,short,char三种比int小的整数，可以用范围内的值直接赋值

3. 浮点数字面值，是 double 类型

4. 字面值的后缀:L l ,F f,D d

5. 字面值前缀: 0b - 二进制

0x - 16进制

0 - 8进制

\u - char类型16进制:'\u0061'

## 基本类型的运算规则（5条）

1. 计算结果的数据类型，与算项的最大类型一致

3d/2 double/int 结果是 double

2. byte,short,char三种比int小的整数，运算时，会自动转成 int

a += a; //复合的赋值运算，自动转型

byte a = 3;byte b = 4;

byte c = a + b; //错，int + int，结果是int

3. 整数运算溢出

整数运算，可以理解为像时钟转圈，超出范围不出错，而是转到一个错误的结果

Integer.MAX\_VALUE + 1

得负数最小值

300000000\*60\*60\*24\*365 溢出的错误结果

300000000L\*60\*60\*24\*365 long类型正确结果

4. 浮点数运算不精确

IEEE-754规范规定了计算机中浮点数的表示格式，根据这个格式规则，在做某些浮点数运算时，就会产生不精确结果

2-1.9 得 0.100000000009

5. 浮点数的特殊值

Infinity

无穷大

3.14 / 0

NaN

Not a Number

Math.sqrt(-6) 开方根

存在使i>j||i<=j不成立的数吗?

答:成立 eg:Double.NaN||Float.NaN

## 基本类型的类型转换

数字类型之间，可以互相转换，从小到大转换，可以自动完成转换；从大到小转换，需要强制转换

## 运算符的优先级

运算符优先级不需要看，多加小括号

l 想让谁先算谁先算

l 便于代码阅读理解，可读性更强

(1+(2-3))\*4/5

### 闰年的条件：

\*) 能被4整除，并且不能被100整除,能被400整除

## 方法

命名的代码块，方便重复调用

## 二进制

计算机中用 0 和 1 表示数据，一个电位是 bit，8个bit是一个字节

计算机可以用10进制来和人交互

0 0 1 1 10 2 11 3 100 4

一个16进制字符，可以表示二进制4位

l 两个16进制字符，可以表示二进制的一个字节，8位

10111011 01100111 01101011 0110101

bb 67 6b 6b

## 流程控制

If,switch,for,do while, while

### switch

switch只能判断：

byte,short,char,int

enum-------jdk1.7，String

从成立的case，向后无条件穿透执行所有 case ，包括default，直到结束，或者遇到 break 中断跳出switch

### while

先判断，再执行

### do-while

先执行，再判断

代码至少执行一次

### for-each循环

数组遍历、集合迭代遍历的语法简化

### 方法的返回值

方法的计算结果数据，可以传回到调用位置

如果方法不返回值，用 void

如果方法有返回值，要定义返回值的数据类型

### break和continue

1. break

中断、跳出循环和switch

2 continue

继续进入循环的下一轮执行\

ourth:循环命名,跳出外循环

# l 面向对象4天

## 对象

对象可以把数据，打包在一起,定义一个类，是对象模板, 从模板创建的具体实例, 实例是数据的打包

## 面向对象 :

封装,继承,多态,组合

## 封装

#### 1  类

模板、图纸

局部变量，栈

成员变量，堆

类在第一次用到时，会被加载到内存的“方法区”

#### 2 实例

从模板创建的具体实例

新建实例时，在“堆内存”中新分配内存空给给这一个实例

#### 3 引用变量

理解成“遥控器”

引用类型的变量，保存一个实例的内存地址

引用变量的特殊值：null，空，不保存任何实例的地址

#### 4 构造方法

构造一个实例之后，立即执行的一个特殊方法

一个类，必须有构造方法

如果自己不定义构造方法，编译器会添加默认构造方法

自己定义构造方法，一般是要给属性变量赋值

#### 5 this

This:当前对象,正在调用的对象

this(...):重载的构造方法之间调用,必须是首行代码

#### 6 构造方法重载 Overload

一个类中，可以定义多个不同参数的构造方法

同名不同参

## 变量

### 局部

定义在方法中，或局部代码块中

局部变量必须手动初始化（第一次赋值时，分配内存空间）

局部变量的作用域、作用范围，在它定义的大括号内有效

在作用范围内，不能重复定义

### 成员变量

定义在类中

访问范围，受访问控制符的控制

局部变量，可以和成员变量同名

### 递归

在方法中，调用自身

递归算法，一步一步地简化问题，简化成最简，再反向倒推求出结果

一般不同时做多次递归调用，否则运算量会急剧增加，这时考虑是否能用循环来代替递归

## 数组

用来存放一组数据的数据结构

### 数组的创建（3种语法）

int[] a = new int[6];

6个位置都存默认值 0,数组的起始地址值，保存到变量a

int[] a = {6,2,6,8}; 直接初始化数组数据

a = new int[]{7,3,8,1,7,9,3,1};

为存在的数组变量赋值，直接初始化数据，要添加类型

### 数组方法

a.length 长度

Arrays.copyOf(a,a.length+1)将数组扩容

Arrays.toString(数组)把数组数据，连接成字符串 ，返回此对象本身（它已经是一个字符串！）

Arrays.sort(数组)数组排序，基本类型数组，优化的快速排序，引用类型数组，优化的合并排序

Arrays.binarySearch(数组, 目标值)二分法查找、折半查找，在有序数组中，查找目标值的位置下标，找不到，返回 -(插入点+1)

Arrays.copyOf(数组, 长度)把数组复制成一个指定长度的新数组，会创建新数组

数组的复制：

System.arraycopy(原数组,原数组起始位置,目标数组,目标数组起始位置,复制的数据数量)不会新建数组，目标数组要事先存在

数组一旦创建，长度不可变

最大下标 a.length - 1

允许0长度的数组

### 数组遍历

从头到尾，依次访问每个位置

## 继承

作用：代码重用、复用

Java的继承是单继承，只能继承一个父类

不能继承：构造方法,私有成员

在类名后加extends 父类

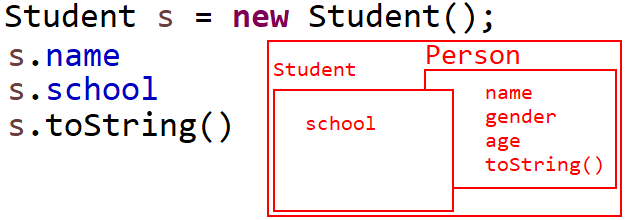
### 子类对象

l 先新建父类对象

l 再新建子类对象

l 两个对象绑定在一起，整体作为一个对象

l 调用成员时，先找子类，再找父类



### 方法重写 Override

继承的方法，在子类中重新定义重新编写

可以理解成“覆盖”、“改写”

### 父类的构造方法

新建子类对象时，会先新建父类对象，也会先执行父类的构造方法

1.默认执行父类的无参构造，默认隐含调用 super()

new Student()， 默认先执行父类的无参

2. 手动调用父类有参构造，super(参数)

当父类没有无参构造，必须手动调用有参构造

### super

super.xxxxx()

l 方法重写时，调用父类中同一个方法的代码

### super(参数)

l 调用父类构造方法

2 默认调用父类无参构造 super()

3 手动调用父类有参构造 super(参数)

4 必须是首行代码

冒泡排序

public static void main(String[] args) {

int[] a=new int[6];

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

a[i] = new Random().nextInt(100);

}

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

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

for (int j = a.length-1; j >i; j--) {

if (a[j]<a[j-1]) {

int t=a[j];

a[j]=a[j-1];

a[j-1]=t;

}

}

}

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

}

Arrays.sort(数组)

基本类型数组，优化的快速排序

引用类型数组，优化的合并排序

## 多态

一个子类型实例，可以被当做父类型来处理

多态作用：一致的类型

### 类型转换

#### 向上转型

子类型实例，转型成父类型

用父类型的引用变量，来引用子类实例

向上转型后，只能调用父类定义的通用成员，子类特有成员被隐藏

#### 向下转型

已经转成父类型的子类实例，转回成子类型

### Instanceof

运行期类型识别

当多种子类型，都当做父类型来处理，要对某一种子类型进行特殊处理，可以先判断真实类型，再向下转型

对真实类型，及其父类型判断，都返回true

## 2 Object

所有类的顶层父类，一个类如果不继承其他类，默认从 Object 继承

### Object 的方法

toString,equals,hashCode(),wait(),notify(),notifyAll()

#### toString()

获得一个对象的字符串表示

Object中的默认实现是：

"day0801.Point@a19c832c"

"类名@地址"

如果需要，可以在子类中重写 toString()

#### equals(Object obj)

当前对象，与参数对象 obj 比较是否相等

a.equals(b)

Object中的默认实现是比较内存地址

如果需要，在子类中，可以重写这个方法

## 3 String

String是封装 char[] 数组的对象

s = "abcd"

s = {

value: ['a', 'b', 'c', 'd']

}

### 字符串的常量池

第一次使用一个字符串字面值时，会在“字符串常量池”中新分配内存，再次使用相同字面值时，直接访问常量池中存在的对象，而不重复创建

String s1 = "abcd";

String s2 = "abcd";

### 字符串不可变

String s1 = "aaa" String s2 = "bbb" String s3 = "ccc"

String s4 = s1 + s2 + s3

字符串不可变,加号连接效率低

## 字符串的常用方法

charAt(i)

获取指定位置的字符

length()

字符串长度，字符的数量

indexOf(子串)

找第一个子串出现的起始位置，找不到返回 -1

indexOf(子串, start)

从执行位置向后查找

lastIndexOf(子串)

从后向前找

substring(start)

截取 start 到末尾

substring(start, end)

截取 [start, end) 范围

trim()

去除两端的空白字符

字符串连接，效率低，每次连接都新建字符串对象

## 二进制十六进制格式判断

s.toLowerCase(); 转成小写

stoUpcase(0,1); 下标为0的值大写,左闭右开

s.substring(2); 截取下标2后的所有数

s.startsWith("0b") 判断前缀

Integer.parseInt(n, 2); 二进制转十进制

s.startsWith("0x") 判断前缀

Integer.parseInt(n, 16) 十六进制转十进制

## StringBuilder

可变的字符序列

封装 char[] 数组，提供了一组方法，可以对内部封装的字符进行修改

常用来代替字符串，做高效率的字符串连接

### StringBuilder 的方法(6种)

append() 方法

追加字符内容,内部数组默认初始容量16,放满后，容量翻倍+2

delete(start, end)

删除区间 [start, end)

deleteCharAt(i)

删除指定位置的单个字符

insert(i, 内容)

在指定位置插入

setCharAt(i, 字符)

替换指定位置的单个字符

replace(start, end, 内容)

替换指定范围的内容.

## StringBuilder 和 StringBuffer的区别

StringBuffer 是一个旧版本的类，JDK1.0中就存在StringBuffer

StringBuilder 是一个新版本的类，JDK1.5中新添加的类

StringBuffer 线程安全

StringBuilder 线程不安全，效率高

## 5   正则表达式 Regex

Regex - Regular Expression

### 规则表达式

正确的字符串格式规则

一般用来判断用户输入的内容是否符合格式要求

例如，email，用户名，密码，生日，网址，手机号

# 正则表达式大全

|  |  |
| --- | --- |
| 正则表达式 | 匹配的字符串 |
| k | k |
| abc | abc |
| [abc] | a, b, c |
| [abc][123] | a1, a2, a3, b1, b2, b3, c1, c2, |
| [a-z] | a, z, b, k |
| [a-zA-Z\_0-9] | a,A,9,\_ |
| [\u4e00-\u9fa5] | 中文范围 |
| [^a-zA-Z] | 排除英文9,\*,中 |
| \d | 数字[0-9] |
| \D | 排除数字[^0-9] |
| \w | 单词字符[a-zA-Z\_0-9] |
| \W | 排除单词字符 |
| \s | 空白字符 |
| \S | 排除空白字符 |
| . | 任意字符 |
| [abc]? | ? 0或1个a,b,,c |
| [abc]?[123] | a1,b3,a2,1,2,3 |
| [abc]\* | \* 0到多个a,ab,,abccbbccaacbc |
| [abc]+ | + 1到多个a,ab,abcbcbcaa |
| [abc]{3} | 3个aaa,bbb,abc,ccb,caa |
| [abc]{3,5} | 3到5个abc,abca,abcab |
| [abc]{3,} | 3到多个，至少3个  abc,abca,abcccbbacb |
| | |  |
| ^ | 匹配起始位置 |
| $ | 匹配结束位置 |

## 正则方法

字符串的 matches() 方法，用来做匹配判断

if(s.matches(regex)) {

}

split(正则)

用匹配的子串，来拆分字符串

String s = "aaa,bbb,ccc";

String[] a = s.split(",");

replaceAll(正则, 子串)

替换所有匹配的子串

String s = "sdf754sdfsfd44767sdf7575";

s = s.replaceAll("\\d+", "\*\*\*");

## 基本类型的包装类

有时需要把基本类型，当做一个对象来使用

f(new Integer(6))

|  |  |
| --- | --- |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

## 数字父类 Number

### 子类：

Byte,Short,Integer,Long

Float,Double

BigDecimal,BigInteger

## 取出基本类型值的方法：

byteValue()

shortValue()

intValue()

longValue()

floatValue()

doubleValue()

## Integer 类

Integer类中，存在 256 个Integer 的缓存对象，封装 -128 到 127

如果指定范围内的值，访问缓存对象

如果指定范围外的值，直接新建对象

### Integer 的方法

字符串解析成 int

Integer.parseInt("进制数", 几进制) 转成十进制

Byte.parseByte()

Short.parseShort()

Integer.parseInt()

Long.parseLong()

Float.parseFloat()

Double.parseDouble()

Boolean.parseBoolean()

### 进制转换

Integer.toBinaryString(255) "11111111" 转二进制

Integer.toOctalString(255) "377" 8进制

Integer.toHexString(255) "ff" 十六进制

## 自动装箱、自动拆箱

### 自动装箱

基本类型值，自动包装成包装对象

Integer a = 6;

编译器编译成：

Integer a = Integer.valueOf(6);

### 自动拆箱

int i = a;

编译器编译成：

int i = a.intValue();

a = a + 1;

编译器编译成：

a = Integer.valueOf(a.intValue() + 1);

自动拆箱，要当心 null 值

## 抽象类

半成品类，没有写完的类

抽象方法，没有代码，只有方法定义,包含抽象方法的类，必须是抽象类

抽象类不能创建实例

### 抽象方法的作用：

作为通用方法，在父类中定义

要求子类，必须实现这个方法

## 二分法查找、折半查找

在有序数组中查找一个值所在的位置下标，

如果找不到，返回 -(插入点+1)

29

mid

\* [2,3,3,5,6,8,12,13,13,13,17,28,30,32] 14

\* lo

\* hi

lo和hi定位低位和高位下标

中间位置是 mid = (lo+hi)/2;

mid位置大，hi定位到mid-1

mid位置小，lo定位到mid+1

mid位置找到相等的，mid下标就是结果

找不到，-(lo+1)，为什么+1，防止lo是0

## BigDecimal，BigInteger

BigDecimal精确的浮点数运算

BigInteger超大整数运算

把运算数，先都封装成 BigDecimal 对象，再使用 BigDecimal 提供的计算方法来计算

创建 BigDecimal 对象

BigDecimal bd = BigDecimal.valueOf(2);

### 方法

add(BigDecimal bd)

subtract(BigDecimal bd)

multiply(BigDecimal bd)

divide(BigDecimal bd)

divide(BigDecimal bd,保留位数,舍入方式)

setScale(保留位数,舍入方式)

舍入运算，保留1位小数

d = BigDecimal

.valueOf(d)

.setScale(1, BigDecimal.ROUND\_HALF\_UP)

.doubleValue();

## 位运算符

+-\*/% == != > >= < <= && || ! ++ -- ? : = += () instanceof

|  |  |
| --- | --- |
| & | 位与，两位都是1，结果才是1 |
| | | 位或，任意一位是1，结果就是1 |
| ^ | 异或，不同是1,对同一个值异或两次得到原值 |
| ~ | 求反，1变0,0变1 |
| >> | 带符号右移位,符号位是1，左边补1 |
| >>> | 不带符号右移位,左边都补0 |
| << | 左移位 |

## day10\_\_\_final

可以修饰常量、方法、类,

final常量值不可变

### final方法

不能在子类中重写

### final类

不能被继承，没有子类

java.lang.System

java.lang.String

java.lang.Integer

## static

静态

静态成员属于类,而不属于实例

### 静态成员

用“类”来调用静态成员

Soldier.count

Math.random()

Integer.parseInt()

### 实例成员

用“实例”来调用实例成员

s1.id

s2.blood

### 静态的使用场景

static 的使用原则：能不用就不用

静态是“非面向对象”的语法

1 共享的数据

Soldier.count

2 工具方法

Math.random()

Arrays.toString()

String.valueOf()

### Static运行规则

静态方法中，不能直接调用实例的成员

静态调静态

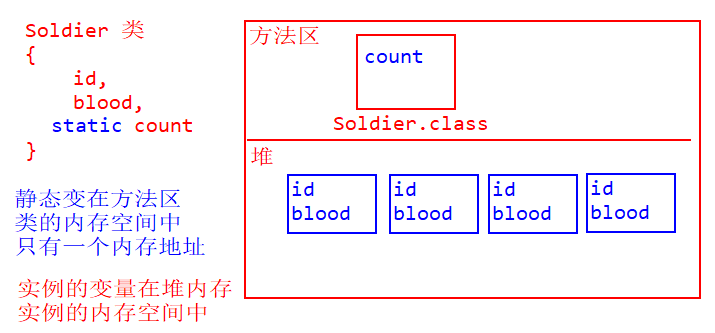
静态不能直接调用非静态, 只能用实例调用

静态初始化块,类被加载时，只执行一次

在方法区，类的内存空间中，只保存一份

静态属于类，一般用类名调用

一般执行初始化工作，例如，加载一些资源文件



### 静态代码块的创建过程

**package** day1202;

**public** **class** Text1 {

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

分配内存是底层代码，观察不到，

**new** B();

System.***out***.println("--------------------");

**new** B();

}}

**class** A{

**int** v1=1;

**static** **int** *v2*=2;

**static**{

System.***out***.println("A静态块");

}

A(){

System.***out***.println("A构造");

}

}

**class** B **extends** A{

**int** v3=3;

**static** **int** *v4*=4;

**static**{

System.***out***.println("B静态块");

}

B(){

System.***out***.println("B构造");

}

## 访问控制符

用来控制一个类、成员的访问范围

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 类 | 包 | 子类 | 任意 |
| public |  |  |  |  |
| protected |  |  |  |  |
| [default] |  |  |  |  |
| private |  |  |  |  |

### 使用原则：

尽量使用小范围

public 是开发人员的一个契约，约定公开的东西会尽量保持稳定不变

private 私有隐藏，便于修改

### 常量

static final int MAX\_VALUE=100;

static final 两个关键字定义常量

final - 不可变

static - 节省内存

**命名习惯，全大写，单词之间用下划线连接**

## 类加载

1. 加载父类，为父类静态变量分配内存

2. 加载子类，为子类静态变量分配内存

3. 执行父类静态变量的赋值运算，和静态初始化块

4. 执行子类静态变量的赋值运算，和静态初始化块

## 新建实例

5. 新建父类实例，为父类的实例变量分配内存

6. 新建子类实例，为子类的实例变量分配内存

7. 执行父类的实例变量赋值运算

8. 执行父类的构造方法

9. 执行子类的实例变量赋值运算

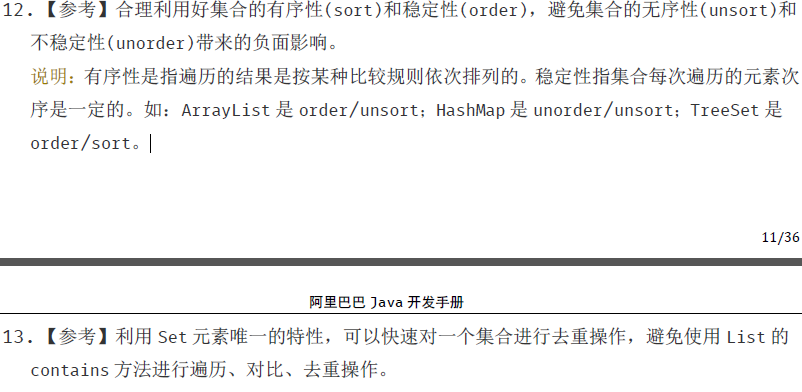
10. 执行子类的构造方法

# 集合 2天

用来存放一组数据的数据结构

## 集合的继承结构(重点)





Collection 接口

|- List 接口

|- ArrayList

|- LinkedList

|- Set 接口

|- HashSet

|- TreeSet

|- ConcurrentHashSet

Map 接口

|- LinkedHashMap

|- HashTable

|- ConcurrentHashMap 康 扣 run t hashmap

|- TreeMap

|- HashMap

Iterator 接口

Collections 工具类

**实例**

package com.tedu;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map.Entry;

import java.util.Set;

import com.sun.xml.internal.stream.Entity;

public class test {

public static void main(String[] args) {

// 推荐

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

// list.add("1");

// list.add("2");

// Iterator<String> iterator = list.iterator();

// while (iterator.hasNext()) {

// String item = iterator.next();

// if ("1".equals(item)) {

// iterator.remove();

// System.out.println(item);

// }

// System.out.println(list);

// }

// ---------------------------------

// 不推荐

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

// list.add("1");

// list.add("2");

// for (String item : list) {

// if ("1".equals(item)) {

// list.remove(item);

// }

// System.out.println(item);

// }

// ------------------------------------

HashMap<String, String> map = new HashMap<>(16);

map.put("1", "1.0");

map.put("2", "2.0");

map.put("3", "3.0");

map.put("4", "4.0");

// 推荐

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

System.out.println(entrySet);

// 不推荐

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

System.out.println(keySet);

}

}

### 数组的缺点：

1.长度固定

2.访问方式单一，只能下标访问

3.前面增删数据，操作繁琐

### ArrayList

数组列表，封装了一个数组，封装了数组操作代码，提供了几个更简便的方法, 访问任意位置，效率高. 增删数据，效率可能降低

内部数组默认的初始容量 10

放满后，1.5倍增长

#### 迭代器

iterator()

//迭代器遍历集合

//Iterator - 迭代器

//新建迭代器，必须用集合的方法辅助创建

Iterator<Integer> it = list.iterator();

//当还有数据

while(it.hasNext()) {

//取下一个

Integer n = it.next();

System.out.println(n);

}

#### ArrayList方法

add(数据)

添加数据

get(int i)

访问指定下标数据

remove(int i)

移除指定位置数据，返回被移除的数据

remove(数据)

找到第一个相等数据，移除，

找到并移除，返回true , 找不到，返回 false

size()

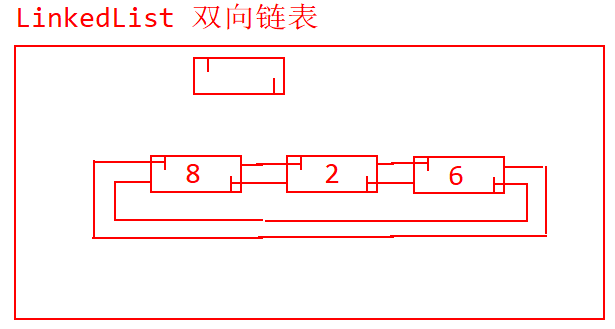
元素的数量

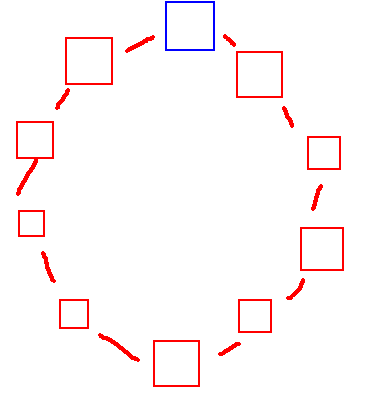
### LinkedList

双向链表

首尾效率高

两端操作数据的方法





#### 方法

和ArrayList具有相同的方法

LinkedList 两端操作数据的方法,(头尾添加,头尾删除,头尾获取)

addFirst(数据)

addLast(数据)

getFirst()

getLast()

removeFirst()

removeLast()

### 队列操作 Queue，FIFO

offer() addLast()

peek() getLast()

poll() removeFirst()

### 栈操作 Stack，LIFO

push() addFirst()

pop() removeFirst()

HashMap

### 哈希表、散列表

存放键值对数据

用键，可以快速定位数据，来提取键对应的值

键：不重复,无序

#### 方法：

put(key, value)

放入键值对数据，重复的键，新值覆盖旧值

get(key)

获得键对应的值，键不存在，得到null

remove(key)

移除键值对数据，返回被移除的值

size()

键值对的数量

#### hashCode()

hashCode() 是 Object 的方法，默认实现是用内存地址作为哈希值

如果希望相同属性，具有相同哈希值，可以重写这个方法

#### **算法**，

@Override

public int hashCode() {

\* \*) 相同属性，产生相同的哈希值

\* \*) 不同属性，要尽量产生分散的哈希值

\* 数学家发明了一种算法，

\* 是一种惯用的，有效算法

\*/

int p = 31; //神奇值31

int r = 1; //r起始值任意

r = r\*p + x;

r = r\*p + y;

return r;

}

@Override

public boolean equals(Object obj) {

//参数obj是Point类型

if(obj instanceof Point) {

Point p = (Point) obj;

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

}

return false;

}

## **哈希运算过程(重点背会)**

HashMap内部，使用 Entry[] 数组存放数据

数组默认初始容量 16

放满后容量翻倍+2

key.hashCode() 获得键的哈希值

用哈希值和数组长度，运算产生一个下标值 i

新建Entry 对象来封装键值对数据

Entry对象，放在 i 位置

如果是空位置，直接放入

有数据，依次equals()比较是否相等

找到相等的，覆盖旧值

没有相等的，链表连接在一起

负载率、加载因子 0.75

新建翻倍容量的新数组

所有数据，重新执行哈希运算，放入新数组

jdk1.8

链表长度到8，转成红黑树

树上的数据减少到6，转回成链表

# 异常1天

## 异常

封装错误信息的对象

错误信息：

类型

提示消息

行号

### 异常的继承结构

Throwable

|- Error 系统级错误

|- Exception 可修复的错误

|- 其他Exception

|- RuntimeException

|- NullPointerException

|- ArrayIndexOutOfBoundsException

|- NumberFormatException

|- ArithmeticException

|- ClassNotFoundException

|- ...

### throw

throw 手动抛出异常，执行异常的抛出动作

类比成 return

当程序出现逻辑错误，不自动创建并抛出异常，可以手动判断逻辑错误，手动创建异常对象并抛出

if(...) {

AException e = new AException();

throw e;}

### 异常包装

捕获的异常对象，包装成另一种类型，再抛出

try {

} catch(AException e) {

BException e2 = new BException(e);

throw e2;

}

#### 使用场景：

不能抛出的异常，包装成能抛出的异常再抛

多种类型异常，简化成一种类型

### 自定义异常

[附录代码](#OLE_LINK2)

现有的异常类型，不能表示所有业务中的错误情况

A到B转账失败

ZhuanZhangShiBaiException

用户名错误

UsernameNotFoundException

密码错误

WrongPasswordException

自定义异常

1.起一个合适的类名

2.选择一个合适的父类

3.添加合适的构造方法

### RuntimeException 和 其他Exception

#### 1.RuntimeException

非检查异常

编译器不检查RuntimeException 是否有处理代码

有默认的抛出管道

#### 2.其他异常

检查异常

编译器检查是否有处理代码，不处理，不能编译

强制开发人员事先考虑错误处理

#### 3. throws 和 try-catch

不能随便 catch

底层的异常，一般要向上层抛，在上层处理

当经验不足时，不知道该catch还是throws，就选throws

## 接口

极端的抽象类

作用：结构设计工具，用来解耦合、隔离实现

用 interface 代替了 class 关键字

用 implements 代替了 extends 关键字

接口中只能定义：

1.公开的抽象方法

2.公开的常量

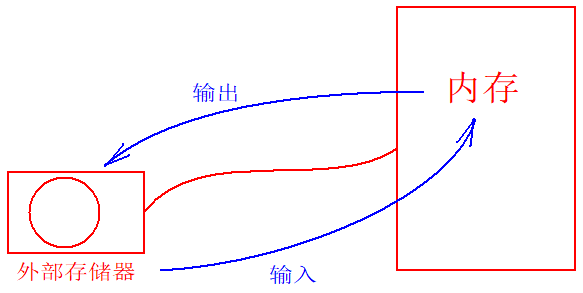
3.公开的内部类、内部接口

类可以同时继承多个接口

# io

Input / Ouput

输入 / 输出



## java.io 包

File

InputStream/OutputStream

FileInputStream/FileOutputStream

ObjectinputStream/ObjectOutputStream

Reader/Writer

InputStreamReader/OutputStreamWriter

BufferedReader

PrintWriter、PrintStream

## File

封装一个磁盘路径字符串，提供了一组对文件、文件夹操作的方法

可以封装文件路径、文件夹路径、不存在的路径

{path="d:/abc"}

{path="d:/a.txt"}

{path="f:/xx/xxx/xx"}

### 文件、文件夹属性方法

getName()

文件名

getParent()

父目录

getAbsolutePath()

完整路径，绝对路径

length()

文件的字节量，对文件夹无效

isFile()

是否是文件

isDirectory()

是否是文件夹

### 创建、删除

**如果文件夹内有文件无法删除**

createNewFile()

新建文件

文件已经存在，不会再新建，返回 false

所在的文件夹不存在，会出现异常

mkdirs()

逐层创建多层文件夹

delete()

删除文件，或空目录

### 目录列表

list()

得到 String[]，包含所有的文件名

["a.txt", "b.mp3", "c.jpg"]

listFiles()

得到 File[]，包含所有文件的封装的File对象

[{...}, {...}, {...}]

### 文本文件按行读写

BufferedReader，readLine()

读取一行字符串，读完再读，返回null

PrintWriter, println()

打印输出一行字符串，末尾补换行

BufferedReader in =

new BufferedReader( //处理换行，按行读取

new InputStreamReader( //编码转换流

new FileInputStream("d:/abc/f8"),"GBK"));

PW--OSW--FOS--f9

FOS - 插在文件上

OSW - 编码转换

PW - println()自动加换行

PrintWriter out =

new PrintWriter(

new OutputStreamWriter(

new FileOutputStream("d:/abc/f9"),"GBK"));

## 流 Stream

数据的读写操作（io操作），抽象成数据在管道中流动

单方向流动

输入流，只能用来读取数据（读入内存）

输出流，只能用来输出数据（内存数据向外输出）

只能从头到尾，顺序流动一次，不能反复流动，如果要重复流动，可以重新创建新的流

InputStream,OutputStream

字节流的抽象父类

### 方法：

write(int b)

只输出int四个字节中，末尾的一个字节值

[1][2][3][4] ---> [4]

write(byte[], start, length)

输出byte[] 数组中，从start开始的length个字节值

read()

读取一个字节值，补三个0字节，变成int

[4] ---> [1][2][3][4]

读取结束后，再读取，返回 -1

read(byte[] buff)

按数组的长度，读取一批字节值，存放到指定的数组中，并返回这一批的字节数量

读取结束后，再读取，返回 -1

### ObjectIOStream

对象序列化、反序列化

序列化 把一个对象的信息，按固定的字节格式，变成一串字节序列输出

#### 方法：

writeObject(Object obj)

把对象变成一串字节序列输出

readObject()

读取序列化数据，反序列化恢复对象

#### Serializable 接口

被序列化的对象，必须实现 Serializable 接口

#### 不序列化的变量

static - 属于类，不随对象被序列化输出

transient - 临时，只在程序运行期间，在内存中存在，不会被序列化持久保存

#### 序列化版本

控制旧版本的数据，不允许恢复成新版本的类型

如果自己不定义版本，编译器编译时，会根据类的定义信息，计算生成一个版本值

static final long serialVersionUID

#### 空接口，

\* 标识接口，用来标识学生类型实例，允许被序列化

### Reader，Writer

字符流的抽象父类，以字符为单位读写数据

### 字符编码转换流

OutputStreamWriter

把Java的Unicode编码字符，转成其他编码输出

InputStreamReader

读取其他编码字符，转成Unicode字符

-------------------------------------------------------

String path;

//path = "D:\\home\\java\\eclipse\\eclipse.exe";//存在的文件

//path = "D:/home/java/eclipse";//存在的文件夹

path = "f:/xxx/xx/xx";//不存在的路径

File f = new File(path);//封装路径

System.out.println(f.getName());//取文件名

System.out.println(f.getParent());//取父目录

System.out.println(f.getAbsolutePath());//取完整路径

System.out.println(f.length());//文件字节量，目录无效

System.out.println(f.isFile());//是否是文件

System.out.println(f.isDirectory());//是否是文件夹

# 内部类

定义在类内部、方法内部或局部代码块内部的类

非静态内部类、属于实例的内部类

class A {

class Inner {}}

非静态内部类实例，必须依赖于一个外部类的实例才能存在

A a1 = new A();

Inner i = a1.new Inner();

用来辅助外部实例运算，封装局部数据，或局部的运算逻辑

### 静态内部类

class A {

static class Inner {

}}

Inner i = new Inner();

静态内部类，与普通的类没有区别

静态内部类，只是一个设计的选项，可以选择嵌套定义，也可以选择独立定义

### 局部内部类

class A {

Weapon f() {

class Inner implements Weapon{

}

Inner i = new Inner();

return i;

}}

A a = new A();

Weapon w = a.f();

局部定义的类型，类似于局部变量，有作用范围，只能在局部代码块内使用这种类型

### 匿名内部类

Weapon w = new Weapon() {...};

{} - 匿名类

new - 新建匿名类的实例

Weapon - 父类型

() - super()，可传参数super(1,2,3)

# 字符编码

### ASC-II

0到 127，英文、指令字符

### iso-8859-1，Latin-1

西欧编码

ASC-II扩展到255

### CJK编码

亚洲编码，中日韩

### GBK

国标码

英文单字节

中文双字节

### Unicode

统一码，万国码

100万+

常用表，双字节

生僻字符表，三字节、四字节...

### UTF-8

Unicode 的传输格式

Unicode Transformation format

英文，单字节

某些字符，双字节

中文，三字节

特殊符号，四字节

Java的char类型是 Unicode

### Java的转码运算

Unicode ---> 其他编码

String s = "abc中文";

//转成系统默认编码

byte[] a = s.getBytes();

//转成指定编码

byte[] a = s.getBytes("UTF-8");

其他编码 --> Unicode

//从系统默认编码转成 Unicode

String s = new String(byte[]);

//从指定编码转成 Unicode

String s = new String(byte[], "UTF-8");

## 比较器

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

sort()方法，可以使用一个接口，接入一个比较器对象

sort(list, 比较器)

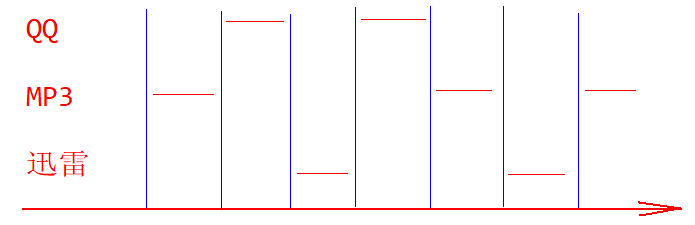
sort()方法，排序运算过程中，

对数据比大小时，调用比较器来比较

# 线程 1天

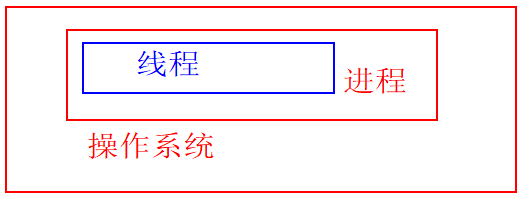
### 进程

在操作系统中，并行执行的任务



### 线程

在进程内部，并行执行的任务



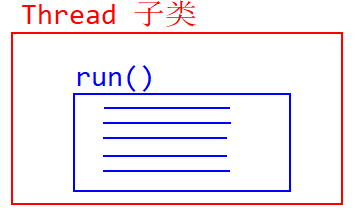
## 创建线程（两种方式）

继承 Thread

实现 Runnable

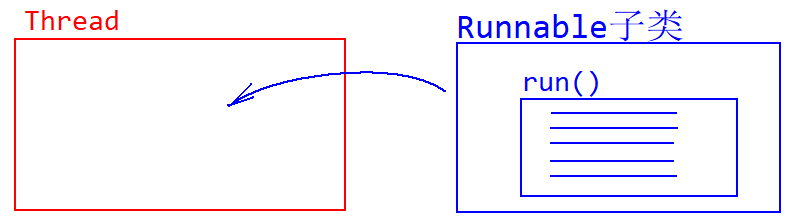
### 继承 Thread

编写 Thread 的子类，并重写 run() 方法。启动之后，自动运行 run() 方法中的代码

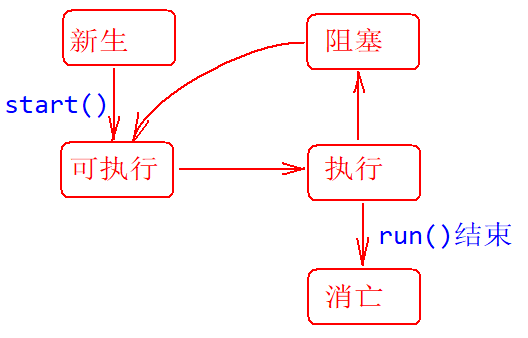


### 实现 Runnable

实现 Runnable 接口，实现它的 run() 方法，Runnable 封装在线程中执行的代码，新建线程对象时，把Runnable对象放在线程内，启动



## 线程的状态



锁

重入锁(公平锁),可防止死锁

private static ReentrantLock lock = new ReentrantLoce();

public static Long getId(){

lock.lock();

try{

count++;

}finally{

lock.unlock();

}

}

## 线程的方法

Thread.currentThread()

获得正在执行的线程实例

Thread.sleep(毫秒值)

让正在执行的线程，暂停指定的毫秒值时长

setName(),getName()

设置,获取线程名

start()

启动线程

interrupt()

打断线程的暂停状态

join()

当前线程暂停，等待被调用的线程结束, 得到线程自己的锁再暂停

Thread.yield()

让步，主动放弃 cpu 时间片，让给其他线程执行

setDaemon(true)

后台线程、守护线程

JVM虚拟机退出条件，是所有前台线程结束，当所有前台线程结束，虚拟机会自动退出,不会等待后台线程结束

例如：垃圾回收器是一个后台线程

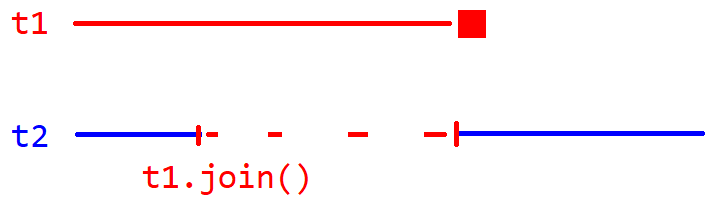
wait() 等待 等待是会释放锁

notify() 唤醒

notifyAll() 唤醒所有

Thread.sleep() 睡眠 睡眠时不会释放锁

**Join图**



## 多线程共享数据访问冲突

一个线程修改数据，另一个线程同时访问数据，可能引起数据访问混乱

## 线程同步 synchronized

让多个线程共享访问数据时，步调一致的执行。

一个线程修改时，其他线程等待修改完成后才能执行；

一个线程访问时，其他线程等待访问结束

任何实例，都有一个“同步锁”，synchronized 关键字，要求一个线程必须抢到同步锁才能执行

synchronized(对象) {

共享的数据访问代码

}

必须抢到指定对象的锁，才能执行

synchronized void f() {

}

抢当前实例（this）的锁

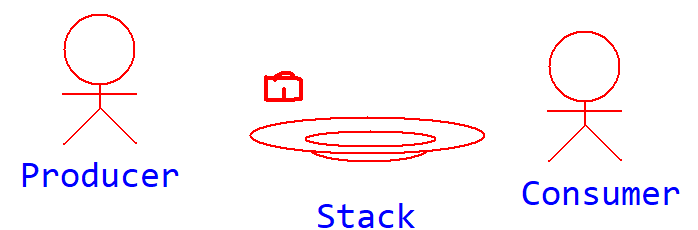
static synchronized void f() {

}

抢类的锁

### 生产者、消费者模型

线程之间传递数据



### 等待和通知

#### Object 的方法

wait()

notify()

notifyAll()

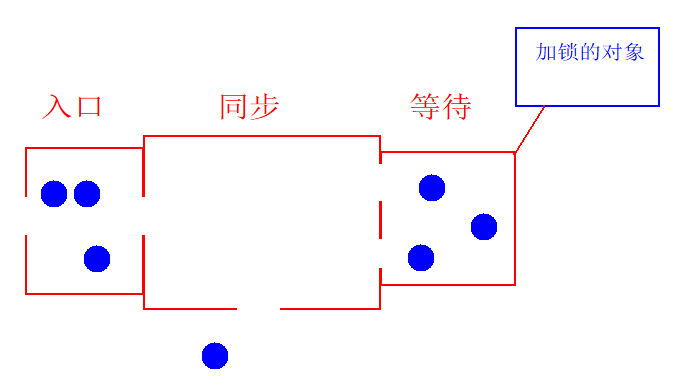
 必须在 synchronized 代码内调用

 等待和通知的对象，必须是加锁的对象

wait() 外面，总应该是一个循环条件判断

### 同步监视器

遇到 synchronized 关键字，会在加锁的对象上，关联一个同步监视器



# 网络 1天

## 查ip地址

输入命令：ipconfig查看ip地址

ping www.baidu.com

如果ping不通，要关闭防火墙

## Socket 通信

Socket - 网络套接字

通过 ip 地址，两台主机可以互相找到对象

在两台主机上，各选择一个端口号

服务器端

被动等待客户端发起连接

### 网络连接(4个步骤)

**1.** ServerSocket 在服务器端,选择一个端口号,在指定端口上等待客户端发起连接, 1024前的端口是常见服务占用,比如80—---用中间的---50000以后系统保留端口,给程序自动分配端口号就用50000后的

ServerSocket ss = new ServerSocket(8000);

**2.**等待客户端发起连接，建立连接通道，并返回连接通道的服务器端插头对象

Socket s = ss.accept();

客户端

**3.**主动去连接服务器

Socket s = new Socket(ip, port);

4.取出双向的流

InputStream in = s.getInputStream();

OutputStream out = s.getOutputStream();

超时

socket.setSoTimeout(毫秒值)

设置接收数据，等待超时时长，超时会出现 SocketTimeoutException

项目：day1701\_网络

类：day1701.Server1

**package** day1701;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.OutputStream;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** Server1 {

**public** **static** **void** main(String[] args) **throws** Exception {

    ServerSocket ss = **new** ServerSocket(8000);

    System.***out***.println(

     "服务已经在 8000 端口上启动");

    //暂停等待客户端发起连接

    System.***out***.println("等待客户端连接");

    Socket s = ss.accept();

    System.***out***.println("客户端已连接");

    //从插头对象，获取输入流和输出流

    InputStream in = s.getInputStream();

    OutputStream out = s.getOutputStream();

    /\*

     \* 通信协议，通信流程，数据格式

     \*

     \*     服务器端：

     \*       1. 接收 "hello"

     \*       2. 发送 "world"

     \*/

**for** (**int** i = 0; i < 5; i++) {

**char** c = (**char**) in.read();

        System.***out***.print(c);

    }

    out.write("world".getBytes());

    out.flush(); //刷出内存缓存

    s.close();//断开连接

    ss.close();//停止服务，释放端口

}

}

Client1

**package** day1701;

**import** java.io.InputStream;

**import** java.io.OutputStream;

**import** java.net.Socket;

**public** **class** Client1 {

**public** **static** **void** main(String[] args) **throws** Exception {

    Socket s = **new** Socket("旁边同学ip", 8000);

    InputStream in = s.getInputStream();

    OutputStream out = s.getOutputStream();

    /\* 客户端：

     \*    1. 发送 "hello"

     \*    2. 接收 "world"

     \*/

    out.write("hello".getBytes());

    out.flush(); //刷出内存缓存

**for** (**int** i = 0; i < 5; i++) {

**char** c = (**char**) in.read();

        System.***out***.print(c);

    }

    s.close();

}

}

### 阻塞操作

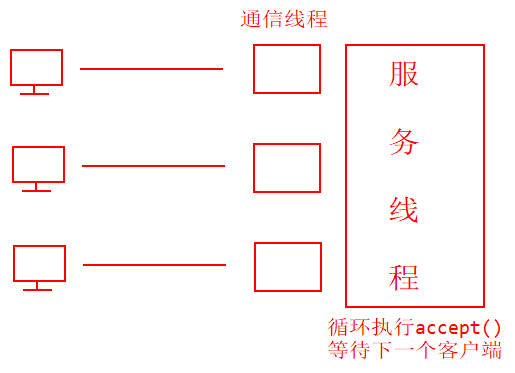
ss.accept()

暂停，等待客户端发起连接

in.read()

对方不发送数据，暂停，死等数据

### 服务器端的线程模型



**练习2  回声**

客户端发送到服务器的数据，原封不动的发回客户端

项目：day1702\_回声

类：day1702.EchoServer

             EchoClient

EchoServer

**package** day1702;

**import** java.io.BufferedReader;

**import** java.io.InputStreamReader;

**import** java.io.OutputStreamWriter;

**import** java.io.PrintWriter;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** EchoServer {

**public** **void** launch() {

**new** Thread() {

        @Override

**public** **void** run() {

**try** {

              ServerSocket ss = **new** ServerSocket(8000);

              System.***out***.println("服务已在 8000 端口上启动");

**while**(**true**) {

                 System.***out***.println("等待下一个客户端连接");

                 Socket s = ss.accept();

                 System.***out***.println("客户端已连接");

                 TongXinThread t = **new** TongXinThread(s);

                 t.start();

              }

           } **catch** (Exception e) {

              System.***out***.println(

               "无法在 8000 端口上启动服务，或服务已停止");

           }

        }

    }.start();

}

**class** TongXinThread **extends** Thread {

    Socket s;

**public** TongXinThread(Socket s) {

**this**.s = s;

    }

    @Override

**public** **void** run() {

        /\*

         \* 通信协议，流程和格式

         \* UTF-8编码的字符串，

         \* 每段字符串末尾添加换行

         \*

         \* BR--ISR--网络输入流

         \* PW--OSW--网络输出流

         \*/

**try** {

           BufferedReader in =

**new** BufferedReader(

**new** InputStreamReader(

            s.getInputStream(), "UTF-8"));

           PrintWriter out =

**new** PrintWriter(

**new** OutputStreamWriter(

            s.getOutputStream(),"UTF-8"));

           String line;

**while**((line = in.readLine()) != **null**) {

              out.println(line);

              out.flush();

           }

           //断开

        } **catch** (Exception e) {

           //断开

        }

        System.***out***.println("一个客户端端已断开");

    }

}

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

    EchoServer s = **new** EchoServer();

    s.launch();

}

}

EchoClient

**package** day1702;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.OutputStreamWriter;

**import** java.io.PrintWriter;

**import** java.io.UnsupportedEncodingException;

**import** java.net.Socket;

**import** java.util.Scanner;

**public** **class** EchoClient {

**public** **static** **void** main(String[] args) **throws** Exception {

    Socket s = **new** Socket("找一个人启动服务器", 8000);

    BufferedReader in =

**new** BufferedReader(

**new** InputStreamReader(

     s.getInputStream(), "UTF-8"));

    PrintWriter out =

**new** PrintWriter(

**new** OutputStreamWriter(

     s.getOutputStream(),"UTF-8"));

**while**(**true**) {

        System.***out***.print("输入：");

        String str = **new** Scanner(System.***in***).nextLine();

        out.println(str);

        out.flush();

        String echo = in.readLine();

        System.***out***.println("回声："+echo);

        System.***out***.println("-----------------");

    }

}

}

**练习3  聊天室**

项目：day1703\_聊天室

类：day1703.ChatServer

             ChatClient

ChatServer

**package** day1703;

**import** java.io.BufferedReader;

**import** java.io.InputStreamReader;

**import** java.io.OutputStreamWriter;

**import** java.io.PrintWriter;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.util.ArrayList;

**public** **class** ChatServer {

**private** ArrayList<TongXinThread> list =

**new** ArrayList<>();

**public** **void** launch() {

    //启动服务线程

**new** Thread() {

        @Override

**public** **void** run() {

**try** {

              ServerSocket ss = **new** ServerSocket(8000);

              System.***out***.println("聊天室服务器已启动");

**while**(**true**) {

                 Socket s = ss.accept();

                 TongXinThread t =

**new** TongXinThread(s);

                 t.start();

              }

           } **catch** (Exception e) {

              System.***out***.println(

               "服务无法在 8000 端口上启动，或服务已经停止");

           }

        }

    }.start();

}

**class** TongXinThread **extends** Thread {

    Socket s;

    BufferedReader in;

    PrintWriter out;

**private** String name;

**public** TongXinThread(Socket s) {

**this**.s = s;

    }

**public** **void** send(String msg) {

        out.println(msg);

        out.flush();

    }

**public** **void** sendAll(String msg) {

**synchronized** (list) {

**for** (TongXinThread t : list) {

              t.send(msg);

           }

        }

    }

    @Override

**public** **void** run() {

**try** {

           //UTF-8, 换行

           in = **new** BufferedReader(**new** InputStreamReader(s.getInputStream(), "UTF-8"));

           out = **new** PrintWriter(**new** OutputStreamWriter(s.getOutputStream(),"UTF-8"));

           //接收客户端的昵称

**this**.name = in.readLine();

           //把当前通信线程实例，加入集合

**synchronized** (list) {

              list.add(**this**);

           }

           //发送欢迎信息

           send("欢迎进入激情聊天室");

           //群发上线消息

           sendAll(name+"进入了聊天室，在线人数："+list.size());

           String line;

**while**((line = in.readLine()) != **null**) {

              sendAll(name+"说："+line);

           }

           //断开

        } **catch** (Exception e) {

           //断开

        }

        //删除当前通信线程实例

**synchronized** (list) {

           list.remove(**this**);

        }

        //群发离线消息

        sendAll(name+"离开了聊天室，在线人数："+list.size());

    }

}

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

    ChatServer s = **new** ChatServer();

    s.launch();

}

}

ChatClient

**package** day1703;

**import** java.io.BufferedReader;

**import** java.io.InputStreamReader;

**import** java.io.OutputStreamWriter;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**import** java.util.LinkedList;

**import** java.util.Scanner;

**public** **class** ChatClient {

**private** Socket s;

**private** BufferedReader in;

**private** PrintWriter out;

**private** String name;

**private** LinkedList<String> list = **new** LinkedList<>();

**private** **boolean** flag; //开关

**public** **void** launch() {

**try** {

        s = **new** Socket("192.168.21.61", 8000);

        in = **new** BufferedReader(**new** InputStreamReader(s.getInputStream(), "UTF-8"));

        out = **new** PrintWriter(**new** OutputStreamWriter(s.getOutputStream(),"UTF-8"));

        //昵称

        System.***out***.print("起一个昵称： ");

        name = **new** Scanner(System.***in***).nextLine();

        out.println(name);

        out.flush();

        //接收线程

**new** Thread() {

           @Override

**public** **void** run() {

              receive();

           }

        }.start();

        //输入线程

**new** Thread() {

           @Override

**public** **void** run() {

              input();

           }

        }.start();

        //打印线程

**new** Thread() {

           @Override

**public** **void** run() {

              print();

           }

        }.start();

    } **catch** (Exception e) {

        System.***out***.println("无法连接聊天室服务器");

        e.printStackTrace();

    }

}

**protected** **void** print() {

**while**(**true**) {

**synchronized** (list) {

**while** (list.isEmpty() || flag) {

**try** {

                 list.wait();

              } **catch** (InterruptedException e) {

              }

           }

           String msg = list.removeFirst();

           System.***out***.println(msg);

        }

    }

}

**protected** **void** input() {

    System.***out***.println("按回车输入聊天内容");

**while**(**true**) {

**new** Scanner(System.***in***).nextLine();

        flag = **true**;//打开开关

        System.***out***.print("输入聊天内容：");

        String s = **new** Scanner(System.***in***).nextLine();

        out.println(s);

        out.flush();

        flag = **false**;//关闭开关

        //通知打印线程可以继续打印

**synchronized** (list) {

           list.notify();

        }

    }

}

**protected** **void** receive() {

**try** {

        String line;

**while**((line = in.readLine()) != **null**) {

**synchronized** (list) {

              list.add(line);

              //通知打印线程已经有数据可以打印了

              list.notify();

           }

        }

    } **catch** (Exception e) {

    }

    System.***out***.println("已经与服务器断开连接");

}

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

    ChatClient c = **new** ChatClient();

    c.launch();

}

}

### 通信协议，流程和格式

\* UTF-8编码的字符串，

\* 每段字符串末尾添加换行

BR--ISR--网络输入流

BufferedReader in =

new BufferedReader(

new InputStreamReader(

s.getInputStream(), "UTF-8"));

PW--OSW--网络输出流

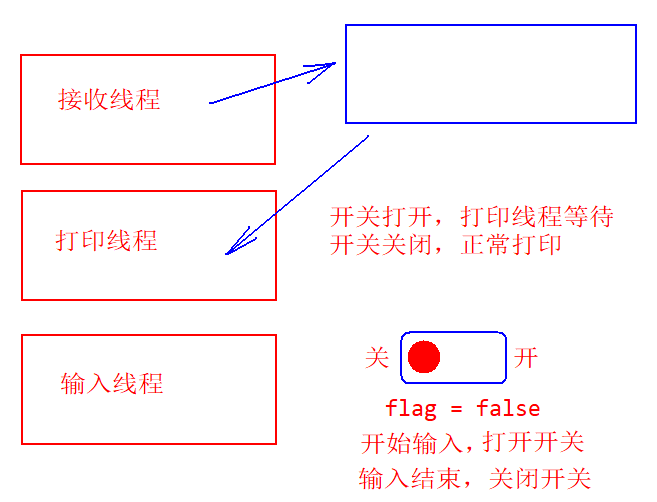
PrintWriter out =

new PrintWriter(

new OutputStreamWriter(

s.getOutputStream(),"UTF-8"));

#### 网络开关



# 反射1天

反射 Reflect

config.txt

--------------------------

day1801.A;a

day1801.B;b

day1801.C;c

day1801.D;d

### **加载到方法区的“类对象”的反射操作**

可以(3步)

1.获得一个类的定义信息

2.反射新建实例

3.反射调用成员

## 获取“类对象”

Class.forName("day1801.A")

A.class

a1.getClass()

## 获得包名、类名

包名

c1.getPackage().getName()

完整类名

c1.getName()

类名，不含包名

c1.getSimpleName()

## 成员变量定义信息

getFields()

获得所有可见的成员变量，包括继承的变量

getDeclaredFields()

获得本类定义的成员变量，包括私有

不包括继承的变量

getField(变量名)

getDeclaredField(变量名)

Field实例，封装一个变量的定义信息

## 构造方法定义信息

getConstructors()

获得所有可见的构造方法

getDeclaredConstructors()

获得本类定义的构造方法，包括私有

getConstructor(参数类型列表)

getDeclaredConstructor(int.class, String.class)

Constructor实例，封装构造方法的定义信息

## 方法定义信息

getMethods()

获得所有可见的方法，包括继承的方法

getDeclaredMethods()

获得本类定义的方法，包括私有

不包括继承的方法

getMethod(方法名,参数类型列表)

getDeclaredMethod(方法名, int.class, String.class)

Method实例，封装方法的定义信息

## 反射新建实例

新建实例时，执行无参构造

Object obj = c.newInstance();

新建实例时，执行有参构造

获取构造方法

Constructor t = c.getConstructor(int.class, String.class);

新建实例，并执行该构造方法

Object obj = t.newInstance(6, "abc");

## 反射调用成员变量

获取变量

Field f = c.getDeclaredField(变量名);

使私有成员允许访问

f.setAccessible(true);

反射给变量赋值

为指定实例的变量赋值，静态变量，第一参数给 null

f.set(实例, 值);

反射访问变量的值

访问指定实例的变量的值，静态变量，第一参数给 null

Object v = f.get(实例);

## 反射调用成员方法

获取方法

Method m = c.getDeclaredMethod(方法名, 参数类型列表);

使私有方法允许被调用

m.setAccessible(true)

反射调用方法

让指定的实例来执行该方法

Object returnValue = m.invoke(实例, 参数数据)

## 反射的作用

反射用来做动态编程

根据配置文件的配置来执行

练习2 .动态编程

for (String s : list) {

//"day1801.A;a" --> ["day1801.A", "a"]

// 0 1

String[] a = s.split(";");//拆分字符串

Class<?> c = Class.forName(a[0]);//获取“类对象”

Object obj = c.newInstance();//新建实例执行无参构造

Method m = c.getMethod(a[1]);//获取方法

m.invoke(obj);//反射调用该方法

}

# 注解 annotation

注解是为其他开发工具，或为其他 java 程序提供代码的额外信息

@Override

public String toString() {

}

@Override 编译器编译代码时，会根据该注解，检查方法定义是否是正确的方法重写语法

自定义注解，需要自己编写处理程序来处理注解

### Runner 运行器

Runner 运行器，从类中，自动发现、运行有@Test注解的方法

### 元注解：对注解的注解

项目：day1802\_注解

package day1802;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

元注解：对注解的注解

@Target 设置注解目标：类、方法、成员变量、参数变量...

@Retention 保留范围：源码、字节码、运行期内存

源码：编译成字节码时被丢弃

字节码：类被加载到内存时丢弃

运行期内存：在内存中保留

@Target({ElementType.TYPE,ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Test {

// 如果没有默认值，使用注解时，就必须赋值

// 有默认值，可以不赋值

int id() default 0;

String title() default "";

特殊的属性名 value，

有特殊待遇，单独赋值时，可以不写value=

可以作为一个有意义属性名的别名

@Test("sdfsdfs")

@Test(id=5, value="dfgsdf")

@Test(id=5, title="dfgsdf")

//title的别名

String value() default "";

}

io流，网络，反射，注解

是底层代码，有框架或上层API封装

## JUnit单元测试框架

JUnit 单元测试框架是第三方的开源工具，需要下载开发包，引入到项目中，才能使用

JUnit 是Java单元测试的事实标准，eclipse也集成了 JUnit

右键点击项目 -- Build path -- add library/Other -- 选择 JUnit -- JUnit4

# 网络爬虫 1天

抓取网页内容，要和服务器的80端口建立Soket连接，再向服务器发送 http 请求协议数据，再接收服务器返回的页面数据，再从页面数据中，解析提取需要的内容

HttpClient

JSoup

JSoup

第三方开源 API，要下载开发包，引入到项目中

练习5 网络爬虫

项目：day1804\_爬虫

类：day1804.Test1

右键点击项目 -- Build path -- add library/Other -- 选择 JUnit -- JUnit4

右键点击项目 -- Build path -- add external archive.. -- 选择解压的10个jar文件

https://item.jd.com/44076405947.html

https://p.3.cn/prices/mgets?skuIds=44076405947

Java version 查看jdk版本

# API

二进制转十进制

s.toLowerCase() 大写转小写

s.substring(2) 取下标2后面数

s.startsWith("0b") 判断前缀

parseInt() 解析

outer: 循环命名,跳出外循环

double d = Math.random(); 随机浮点数

continue 继续进入循环的下一轮执行

break 中断,跳出循环

Arrays.toString(a) 把数组中的数据，连接成字符串

数组的长度属性 数组一旦创建，长度不可变,最大下标 a.length – 1,允许0长度的数组

s.indexOf(子串), 在s中寻找子串的起始位置

\* s.indexOf("bc") 1 下标值

\* s.indexOf("xx") -1 特殊值表示不存在

看代码:Day13

**Day12自定义异常**

**package** day1201;

**public** **class** UsernameNotFoundException **extends** Exception{

**public** UsernameNotFoundException() {

**super**();

}

**public** UsernameNotFoundException(String message, Throwable cause, **boolean** enableSuppression,

**boolean** writableStackTrace) {

**super**(message, cause, enableSuppression, writableStackTrace);

}

**public** UsernameNotFoundException(String message, Throwable cause) {

**super**(message, cause);

}

**public** UsernameNotFoundException(String message) {

**super**(message);

}

**public** UsernameNotFoundException(Throwable cause) {

**super**(cause);

}

}

WrongPasswordException

**package** day1201;

**public** **class** WrongPasswordException **extends** Exception{

**public** WrongPasswordException() {

**super**();

}

**public** WrongPasswordException(String message, Throwable cause, **boolean** enableSuppression,

**boolean** writableStackTrace) {

**super**(message, cause, enableSuppression, writableStackTrace);

}

**public** WrongPasswordException(String message, Throwable cause) {

**super**(message, cause);

}

**public** WrongPasswordException(String message) {

**super**(message);

}

**public** WrongPasswordException(Throwable cause) {

**super**(cause);

}

}