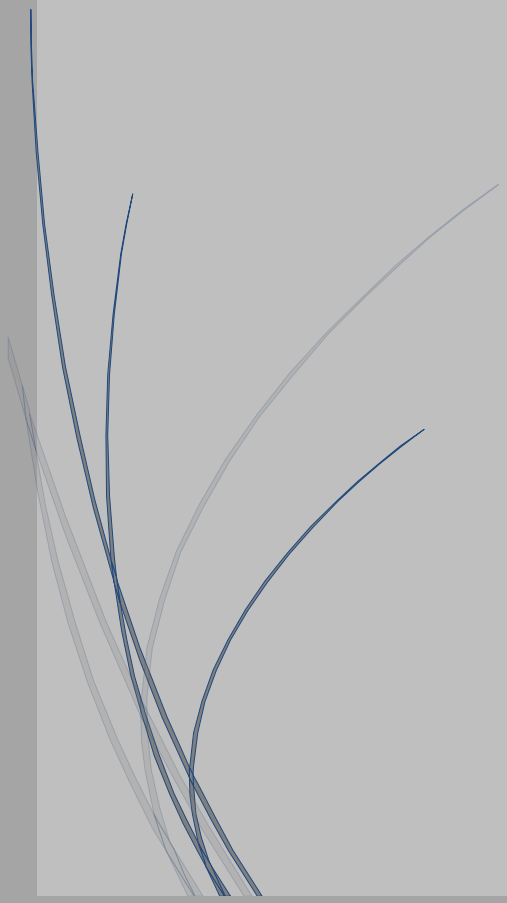
学习笔记

2018-7-14

javaEE + 大数据

蒋文明(JWnMing)



一生很短，但又很长

恰韶华盛时奋，如不竭心，何来血拼？

在这技术流互相邂逅的时代里

若想淘出一角天地

IT正是一件战士降的重型铠甲

加油！

--- 致自己

蒋 文明

百知教育-郑州校区

目 录（Ctrl 键定位）

Ip:192.168.77.13
ftp:192.168.77.5 one 123456
百知教评系统 ： jwnming jwnming

1 –啧啧(JDK8)

1).小知识点

skr：

1、 if (1) 这句是错误的; switch（byte、short、int、char(特殊的 int 类型)、 String(1.7jdk 版本)）

2、 Vector(是 1.0 版本，95 年，已经被淘汰)、ArrayList(98 年至今)

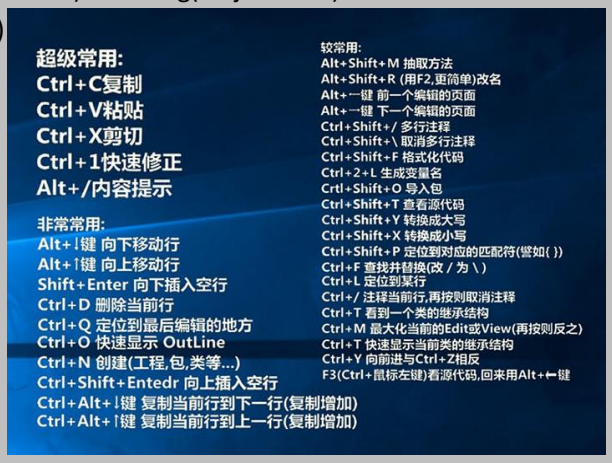
3、 ++a 比 a++ 运行快，因为++a 没有使用寄存器，
不用进栈和出栈

4、 交换整型变量 a 与 b 的值
① a = a^b; b = a^b; a = a^b;
② a = a+b; b = a-b; a = a-b;
③ int c = a; a = b; b = c;

5、 在类中 private 修饰的变量可通过 get、set 方法

6、 静态（static）方法可以通过类名直接调用该方法
非静态方法要通过实例化对象使用

7、 抽象类不能创建实例化对象，只能定义变量和数组



8、 当创建子类对象时，先加载父类静态代码块，再加载子类静态代码块。此间会在子类构造函数内执行 super(), 此时若父类有有参构造却无无参构造，程序出错

9、 static、final、private 不能和 abstract 同时出现（同时修饰）

10、可变长参数（T … t），例如：（String … s） 如同 （String[] s）
一个方法中最多有一个可变长参数，并且在参数表最后 ：（String a, String b, String… c）
传值时：（"DF", “DFG”, “SF”….）

11、输出 System.out.printf(“%d”, 20);

12、随机时：（int）（Math.random() * 1000） //随机 1000 以内的整数

13、wait() / sleep()， wait 会丢失所有锁标记，sleep 不会丢失锁标记

14、Oracle:查看所有用户 -> select username from dba_users;
切换用户 -> alter user 用户名 identifi ed by 密码;
授权 -> grant connect,resource,dba to 非系统用户

15、List 等集合只存放数据，本身不能遍历，而是通过迭代器实现的
List<String> list = new ArrayList<>();
Iterator<String> it = list.iterator(); //Stream

16、List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7); //此时 list 的长度不能变

17、泛型类型的约定 <T extends U> 约定 T 为 U 的类型

18、

----> fork-join 框架，jdk8 新语法（Lambda 编程、Stream 编程）

2).JDK8_Lambda 表达式

jdk8 新语法：

```
## JDK8
JDK1.0    95      Vector  Hashtable  synchronized
JDK1.2    98      List Set Map
JDK1.5    2004    泛型 枚举 标注 多线程   自动封箱   静态导入   可变长参数(本文档有讲解)
```

JDK6　　Arrays.copyOf()
JDK7　　String 作为 switch 表达式
　　switch（String） --> 原理是获得字符串 hashCode(int)、再做等值判断
　　例如：switch("a") { case “a” : {①} }
　　==> switch(“a”.hashCode()) { case “a”.hashCode():if(“a”.equals(“a”) {①}}

　　int 字面值 0b0001111(二进制表达)　分隔符（int a = 1000_000_000）
　　try-with-resources　Fork-Join　泛型自动推断
JDK8　2014　lambda 表达式

Lambda 表达式

避免冗余代码，提高程序的可重用性
提高可重用性：将代码的不变部分，和可变部分 分离
1. 继承关系
2. 将数据作为方法的参数
3. 将代码作为方法的参数　定义接口,通过接口回调实现
　Lambda ： 函数式编程

Lambda 表达式　匿名内部类的简便写法　实现的接口必须只有一个抽象方法　(函数式接口)
语法:
1.　(参数表) -> {代码块}
2.　(参数表) -> 表达式
参数表中形参的类型可以省略，由编译器自动推断
如果参数表只有一个参数,()可以省略

```
Runnable r= new Runnable(){
    public void run(){
        System.out.println("hehe");
    }
};
Runnable r = ()->{System.out.println("hehe");};//与上等价

Callable<Integer> c = new Callable<Integer>(){
   public Integer call(){
        return 10;
    }
}
Callable<Integer> c = ()->10;//与上等价
```

方法引用

```
      main:
        A a2 = (MyClass mc)->mc.method();
        A a3 = MyClass::method;

        B b2 = (MyClass mc , String s)->mc.method(s);
        B b3 = MyClass::method;

        C c2 = (MyClass mc,String s1,String s2)->mc.method(s1, s2);
        C c3 = MyClass::method;
```

```
        D d2 = ()->MyClass.staticMethod();
        D d3 = MyClass::staticMethod;

        E e2 = (s)->System.out.println(s);
        E e3 = System.out::println;

    interface A {
        void act(MyClass mc);
    }
    interface B {
        void act(MyClass mc, String s);
    }
    interface C {
        void act(MyClass mc,String s1, String s2);
    }
    interface D {
        void act();
    }
    interface E {
        void act(String s);
    }
    class MyClass {
        public void method() {
            System.out.println("method()");
        }
        public void method(String s) {
            System.out.println("method(String)");
        }
        public void method(String s, String s) {
            System.out.println("method(String, String)");
        }
        public static void staticMethod() {
            System.out.println("static method()");
        }
    }
```

接口的新语法:

1. 接口中可以定义默认方法 `default void print(){}`
2. 接口中可以定义静态方法 `static void print(){}`
3. 接口中可以定义私有方法 since JDK9

例题在最后;

3).JDK8_Stream 编程

函数式接口: 函数描述符

* Runnable   ()-> Void
* Callable   ()-> T

接口名	方法名	函数描述符	含义
Predicate<T>	test()	T->boolean	判断
Consumer<T>	accept()	T->void	消费
Function<T,R>	apply()	T->R	转换
Supplier<T>	get()	()->T	供应商
BiConsumer<T,U>	accept()	(T,U)->void	复杂特化
BiFunction<T,U,R>	apply()	(T,U)->R	
UnaryOperator<T>	apply()	T->T	
IntFunction<R>	apply()	int->R	基本类型特化
ToIntFunction<T>	applyAsInt	T->int	

Stream 函数式数据处理

Stream: 处理集合中数据的运算
集合: 负责数据的存储

(1)获得 Stream

1. Collection stream(): 获得单线程的 Stream
2. Collection parallelStream():获得并发的 Stream
3. Arrays.stream(T[]) Stream<T>
4. Files.lines(Path) Stream<String> 读取文本文件,并把每行文本放入 Stream<String>

例:

```
Stream<String> s = Files.lines(Paths.get("a.txt"));   、、处理英文
Stream<String> s = Files.lines(Paths.get("a.txt") , Charset.forName("GBK"));  处理中文
```

(2)中间操作

方法名	操作描述
filter(Predicate)	对流中的数据做过滤
distinct()	去掉流中的重复元素
limit(int n)	取流中的前 n 个元素
skip(int n)	跳过流中的前 n 个元素
sorted(Comparator)	排序
map(Function<T,R>) T->R	Stream<R> 对每个元素应用函数,将 T 对象转换为 R 对象存入 Stream

例子:

```
List<Employee> list = new ArrayList<Employee>();
//打印所有雇员中年龄有小到大排序后的地第三到第六位的名字:
list.stream().sorted(Comparator.comparingInt(Employee::getAge)).skip(2).limit(4).map(Employee::getName).forEach(System.out::println);
```

(3)收集操作

方法名	返回值	操作描述
allMatch/anyMatch/noneMatch (Predicate<T>)	boolean	判断流中的元素是否能够匹配条件
count()	long	返回流中的元素数量
forEach(Consumer<T>) T->void	void	对流中的所有元素做遍历
max/min (Comparator<T>)	Optional<T>	找出最大的元素/最小的元素

findAny()/findFirst()	Optional<T>	从流中找出任一个/第一个元素
collect()		收集元素

收集操作 collect() ----> 收集器 Collector<T,A,R> Collectors 直接获得 Collector

T:流中的元素类型、 A:中间类型、 R:收集之后的结果类型

方法	参数	第三泛型(R)	方法描述
counting()	无	Long	统计个数
summingInt()	T->int	Integer	对整数求和
averagingInt()	T->int	Double	对整数求平均数
minBy()/maxBy()	(T,T)->int	Optional<T>	求最小/最大值
summarizingInt()	T->int	IntSummaryStatistics	统计个数,总和,平均数,最大值,最小值
toList()/toSet()	无	List<T>/Set<T>	将 Stream 中的元素放入 List/Set
toMap()	T->K,T->U	Map<K,U>	将 Stream 中的元素放入 Map
toConcurrentMap()	T->K,T->U	ConcurrentMap<K,U>	将 Stream 中的元素放入 ConcurrentMap
joining()	无	String	将 Stream 中的字符串拼接成 String
collectingAndThen()	Collector<T,?,R>, R->RR	RR	先收集数据,然后对数据做转换
mapping()	T->U,Collector<U,?,R>	R	先对元素做转换,再收集数据
groupingBy()	T->K	Map<K,List<T>>	将每个元素转换为 K 对象,并以 K 对象作为分组依据,将所有元素分组,形成 Map
groupingBy()	T->K,Collector<T,?,D>	Map<K,D>	分组后,再对分组结果做收集
partitioningBy()	T->boolean	Map<Boolean,List<T>>	根据谓词条件,分成 2 组
partitioningBy()	T->boolean,Collector<T,?,D>	Map<Boolean,D>	分区后,在对分区结果做收集

例题在最后（题呗）；

原始流特化

IntStream LongStream DoubleStream

获得原始特化流的方法:

1. Stream<T> 调用 mapToInt(T->int)
2. IntStream.range(a,b) a ---> b-1
   IntStream.rangeClosed(a,b) a ---> b
3. IntStream.generate(()->int) 由函数生成每个元素
4. IntStream.iterate(a,int->int) 第一个元素是 a,利用函数,通过前一个元迭代计算后一个元素

方法:

* average() OptionalDouble 求平均数
* sum() int 求和
* boxed() Stream<Integer>

2 - Java 的工作方式、环境变量、编译、标识符等

Java 工作方式：先编译（.class），再解释执行

编译命令：在放 java 文件的目录下 javac 文件名（.java） ----------> 将文件里的类生成.class 文件

运行命令：在放.class 文件目录下 java 类名 ---------> 运行这个类

重量级名词：

JVM:java 虚拟机，屏蔽操作系统差异，统一编程标准

JDK: JVM + 编译器 + 解释器 + 工具 + 类库

JRE: JVM + 解释器

环境变量：

JAVA_HOME：是安装 jdk 的安装文件完整路径，例：C:\Program Files\Java\jdk1.8.0_181

Path：java 工具的加载路径（C:\Program Files\Java\jdk1.8.0_181\bin）

CLASSPATH：加载要运行时的类文件的目录，例：如果是当前目录就填入”.”;

一个 java 文件内最多有一个 public 修饰的公开类，且这个类的类名应与源文件名一致

带包编译：javac 包名(可以是多个包).源文件名 例如：javac p1.p2.p3.hello.java （包都存在）

Javac -d 包名.源文件名 例如：javac –d p1.p2.p3.hello.java（包不全，**-d** 自动补全包）

Java 程序基本结构：

package 0 – 1 个

import 0 – n 个

class 1 – n 个 公开类只有一个

注释：被注释的字符串将不被编译运行

//单行注释 快捷键：ctrl + / 再按一次取消注释

/* */ 多行注释 快捷键：Ctrl + shift + / 再按 Ctrl + shift + \ 取消多行注释

```
/**
*       多行注释配合 javadoc 工具生成文档    用法 --------> Javadoc   -d   doc   源文件.java
*/
```

标识符：

保留字 （goto、const）、true、false、null

标识符没有长度限制 例如：int weruhsaiodufhpashdf…asdf = 123;

标识符习惯：

1、类名标识符：单词首字母大写 （例如：class **Book**）

2、变量标识符：单词首字母小写，后面的每个单词首字母大写 （例如：int **priceOfBook**）

3、包名标识符：全部小写 （例如：package **com.zzu.one**）

4、常量名标识符：全部大写 （例如： final int **A** = 12; ）

3 -数据的基本类型、进制

1、 byte 1B -128 -- 127

2、 short 2B -32768 -- 32767

3、 int 4B -2147483648 – 2147483647 整数默认为 int 类型

```
int i = 1;
int j=0xA2234;//十六进制赋值 //可以赋值八进制、二进制(jdk7.0)
```

4、 long 8B -9223372036854775808 – 9223372036854775807 例如：long x =30_000L;分隔符”_”

5、 float 4B 符号位 1，指数位 8，尾数位 23 例如：1.3e7 = 1.3* 10^7 F;

6、 double 8B 符号位 1，指数位 11，尾数为 52 例如：1.3e7 = 1.3* 10^7

7、 char 2B 阿斯克码值：0 --- 65535

没有符号位

‘A’ ----> 65;
char c =’A’; 是’A’
char c =65；是’A’
char c=‘\u0041’;是 A 是十六进制的 Unicode 的编码方式

c++; 是 c = ’B’
c+=1; 是 c = ’B’
c = c + 1; 错误，1 为 int 类型

8、 Boolean 布尔类型 字面值 true、false

进制： 八进制数 以 0 开头 例如：047
二进制数 以 0b 开头 例如：0b1001
十六进制 以 0x 开头 例如：ox4A34

位运算：
&：按位求与
|：按位求或
^：按位求异或（10—>1 , 01—>1 , 11—>0 , 00-->0）
~：按位求反
<<：向左移一位 高位补 0
<<<：无符号向左移动一位

4 -数据类型转换

首先，对一个表达式判断顺序（范围最高的那个类），以 a + b 为例：
如果 a 或 b 有 double 类型，则表达式的类型就为 double
否则，a 或 b 若有 float 类型，则表达式的类型为 float
否则，a 或 b 若有 long 类型，则表达式的类型为 long
否则，表达式值为 int

低级到高级赋值，自动转化：例如：short s =12； int l = s；
高级到低级赋值，需要考虑丢失精度，必须强制转
byte ---> char & int ---> long ---> float ---> double (char 可以看成是特殊的 int 类型)
例如：int l =2; 则 byte = (int) l;

例子： byte a = 10;
a++; （可以）
a+=1; （可以）
a = a +1; （不可以，1 是一个 int 类型，必须强转）

5 -流程控制、for/switch

for(初始条件;条件范围; 控制条件的改变) { } 是正确的，在条件确定的循环中“;”不能省略，若省去条件则是死循环，条件始终为 true；

除了 continue 和 break，还有指定名称跳出指定循环

例如：

```
name：  for(…；…；…){
            for(…；…；…){
                break name; //直接跳出第一个循环
            }
        }
```

switch 接受参数的几种类型：

switch(byte/short/int/char/(String 是在 jkd1.7 版本后有的))

例如：switch（1）；　　（switch（12.2）不行）

```
swhich（）{
  case 1：…
  case 2：…          如果执行 case 1，则后面的全部执行；如若执行了 2，其后都要执行
  case 3：…          直到 break 结束，case 只是相当于执行代码块的一个入口
  default: …
}
```

6 -函数（方法）

函数作用：

减少代码冗余

方法用 return 返回后，方法内其其后的所有代码将不会再执行

利用函数库，提高程序的可重用性

使程序更结构化、简单

定义方式：修饰符（public、static、private 等，不分顺序）+ 返回值类型（无返回 void、其他返回类型（如 int））+ 方法名（参数表）{ 代码实现部分 }

7 -数组

int a[]　可以

int[] a　可以

初始化（赋值）：　a = new int[3];

int [] a = {2,3,4,5}; //定义长度为 4 的数组

String[] c= new String[2];//定义了长度为 2 的数组，默认值都为 null

数组名 a 存的是数组的 地址，而 a[0] 存的是第一个元素的值；

①**数组拷贝**：System.arraycopy(src, 0, dest, 0, length);

System.arraycopy(原数组，原数组起始位置，目标数组，目标数组接受拷贝的起始位置，原数组要复制的长度)

用法：

```
int[] a = {2,4,5};
int[] b = new int[a.length * 2];
```

System.arraycopy(a, 0, b, 0, a.length);

②**数组拷贝**：Arrays.copyOf(src, newlength);
Arrays.copyOf(源数组，新数组的长度);

用法：
int a = {3,5,6};
int b = Arrays.copyOf(a, a.length * 2);

8 -方法重载

重载条件：参数表不同（参数个数不同、参数类型不同、参数类型排列不同），对返回值不作要求，与形参的名字无关。

例如：public void f（）
Public void f（double d）
Public void f（double d , int i）
Public void f（int t , double d）

方法重载的好处：让对象方法有参数表的不同所造成的差异对用户屏蔽

由编译器根据实参来匹配相应的方法

9 -对象构造、构造方法

对象构造：类名 引用名 = new 类名（构造参数表）
引用名.属性 或 引用名.方法（） ---> 访问对象的属性或方法

1、分配空间（根据成员属性）---> 属性被赋默认值
(不考虑继承，若有继承关系应先递归创建父类对象，注意静态代码块加载顺序)
2、初始化属性 ---> 属性被赋初始值 （声明过程）
3、调用构造方法 ---> 属性被再次赋值
注意：如果一个类中没有定义任何构造方法，则系统提供默认公开无参构造；如果类中写了一个构造方法，则系统不会调用默认构造方法

构造方法：是特殊的方法，不能被继承
1、 声明没有返回值类型，是区分其与普通方法的唯一标志
2、 方法名和类名相同
3、 不允许直接调用，在对象构造的过程中自动调用一次
注意：当程序员自己定义构造方法后，默认无参将不会被调用（最好把无参也写了，以便类继承）

10 - Java 中的变量按照数据类型划分

主要有两类：

1、**基本类型**：byte（B）、short（2B）、int（4B）、long（8B）、float（4B）、double（8B）、char（2B）(char 是特殊的 int，字符以阿斯克码值存储)、boolean
储存值：存储的是数值，当传参时传递的实参

2、**引用类型**：例如，String（String 存的变量是调用了其重载的方法）、以及其他自定义的类类型等

储存值：存对象地址，当传参时传递的是对象**地址**

11 -面向对象三大特性：封装

方法：将属性值修饰为私有，提供 get 和 set 方法。造成所有对对象的访问都是通过方法的调用来完成（配合 this 的使用）

结果：用户不能直接随意改变一个对象内的属性，必须通过调用方法（验证）来访问和修改

12 -面向对象三大特性：继承

继承；关键字 extents

修饰符	使用范围	继承与否
Public	公开使用	可以继承
Protected	在本类、同包其它类和子类、其它包的子类	可以继承
(default)	在本类、同包其它类	同包的子类中可以继承
Private	在本类内部使用	不能被继承(实际被继承，无权访问)

注意：构造方法不能被继承

继承的对象创建过程：

C extends B

B extends A

创建 C 对象: //不考虑静态代码块，若有静态代码块，先通过递归的由基类到派生类加载代码块

分配空间

初始化 A 的属性

调用 A 的构造方法

初始化 B 的属性

调用 B 的构造方法（默认调用 A 的构造，super（））

初始化 C 的属性

调用 C 的构造方法（默认调用 B 的构造，super（））

13 -方法覆盖，super 关键字

方法覆盖：

定义：子类用自己的方法实现替换掉能继承自父类的方法实现

要求：方法名形态、参数表相同、返回值类型相同、（访问修饰符相同或范围更广）

this 关键字的使用

1、 本身是引用，代表当前对象。在类中访问自己的属性和方法时，如果不加 this，则自动默认 this 调用，当在局部方法内出现与类属性同名的局部变量时，如加 this 表示类的成员变量，不加则代表是局部变量。

2、 this（），调用本类的无参构造方法

this（…），调用本类中其他相应参数表的构造方法

注意：在使用 this 时，必须将 this 关键字放在构造方法内的第一行

Super 关键字：

1、 是个引用，指向父类的对象，用例调用父类**的方法**，super.父类方法( )，super 须在方法内使用

2、 可以用在构造方法内，指明调用父类的构造方法，用法和 this 相同。但必须放在构造函数的第一行，

例子：

```
class A {
    public A(int i){}
}
class B extends A{
}
```

程序出错，系统默认为

```
class A {
    public A(int i){}
}
class B extends A{
    public B(){
        super();
    }
}
```

14 -面向对象三大特性：多态

没有继承就没有多态

Java 实行的单继承

多态的开闭原则:（继承和多态）对修改关闭，对扩展开放

利用多态，可以使代码针对父亲展开编程，使代码更通透

多态的两种常见的应用场景：

1、 把多态应用在方法的参数上

m (A a) ： 方法可以接受 A 类，也可以接受 A 的子类对象

2、 把多态应用在方法的返回值类型上

A m() ： 方法返回的可以是 A 类或 A 类的某个子类对象

例子:父类对象引用子类对象，以方便扩展新的子类而不需改变方法代码:

例子 1：

```
public class test {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();
        eats(dog);
        eats(cat);
    }
    static void eats(Animal animal){
        animal.eat();
    }
}
class Animal {
    public void eat(){
```

```
            System.out.println("吃的方法");
        }
    }
    class Dog extends Animal {
        public void eat(){
            System.out.println("狗吃骨头");
        }
    }
    class Cat extends Animal{
        public void eat(){
            System.out.println("猫吃小鱼");
        }
    }
```

例子 2

```
    public class ExcEmployee{
        public static void main(String[] args){
            Employee[] es = new Employee[4];
            es[0] = new ProductEmployee("Yangdd",31);
            es[1] = new SalesEmployee("Wucj",36);
            es[2] = new ManageEmployee("Xusy",29);
            es[3] = new CeoEmployee("Huxz" , 16);
            for(int i = 0 ; i <es.length ; i++){
                es[i].work();
            }
            double result = 0 ;
            int count = 0;
            for(int i = 0 ; i < es.length ; i++){
                if (es[i] instanceof ManageEmployee){
                    count++;
                    result+=es[i].getAge();
                }
            }
            System.out.println(result/count);    //平均年龄
        }
    }
    class Employee{
        private String name;
        private int age;
        public Employee(){}
        public Employee(String name){this.name = name;}
        public Employee(String name , int age){this(name);this.age = age;}
        public int getAge(){ return age;}
        public void setAge(int age){this.age=age;}
        public void work(){ }
    }
    class ProductEmployee extends Employee{
        public ProductEmployee(String name , int age){super(name,age);}
        public void work(){ System.out.println("在车间劳动");}
    }
```

```
class SalesEmployee extends Employee{
    public SalesEmployee(String name , int age){super(name,age);}
    public void work(){ System.out.println("出差跑客户");}
}
class ManageEmployee extends Employee{
    public ManageEmployee(String name , int age){super(name,age);}
    public void work(){ System.out.println("在办公室斗地主");}
}
class CeoEmployee extends ManageEmployee{
    public CeoEmployee(String name , int age){  super(name,age);}
    public void work(){ System.out.println("在会议室开会");}
}
```

15 - instanceof 关键字

对象引用名　instanceof　类名 ，来判读引用的对象和类名是否兼容（是否继承该类，或爷爷辈的类）

例子：

```
Team team = new Team();
team.addMember(new Magicer("jwnming",6));
team.addMember(new Soldier("dogee",4));
team.addMember(new Magicer("wanggl",7));
System.out.print("\n" + "团队成员:");
for(int i = 0;i < team.allRole.length;i++){
    if (team.allRole[i] instanceof Role ) {
        System.out.print(team.allRole[i].getName() + "    ");
    }
}
System.out.println("\n" + "\n" +"这个团队的总伤害值:" + team.attackSum());
```

16 - abstract 修饰符、及其他修饰符的修饰对象

public	可以修饰属性、方法、构造方法、类
protected	可以修饰属性、方法、构造方法
default	可以修饰属性、方法、构造方法、类
private	可以修饰属性、方法、构造方法*（如果修饰构造方法全为私有，则该类不支持被继承）*
abstract	可以修饰方法、类
final	可以修饰属性、方法、类

1. abstract 修饰类时，这个类只能声明引用、不能创建对象，可供子类继承（多态）
2. abstract 修饰方法，抽象方法没有实现({})，只有声明(abstract void method()；)，供子类去覆盖实现
3. 总结：
   a) 如果一个类具有抽象方法，那么这个类就必须是抽象类，但抽象类不一定有抽象方法
   b) 如果一个类继承抽象类，但若这个类不希望成为抽象类，这个类就必须实现父亲类中所有的抽象方法
   c) 抽象类也拥有构造方法，供子类用 super()调用
   d) 抽象方法可以使方法的声明和实现部分分离，声明提取到父类，实现留在子类，则更好的体现子类的共性放在父类的设计思想

17 - static 修饰符、类加载

1. 修饰属性：静态的属性不属于任何对象，全类及其对象共用，可以用类名直接调用
2. 修饰方法：静态的方法可以直接用类名调用，和对象无关。静态方法只能访问静态成员和静态方法，非静态方法可以调用静态方法
3. 静态方法只能被子类的的静态方法覆盖，而且没有多态（因为对引用调用静态方法，等价于对引用的类型调用静态方法），静态修饰符和抽象修饰符不能在一起使用
4. 静态初始代码块在类加载的时候执行一次
   a） 类加载：当 JVM 第一次使用一个类的时候，需要提取这个类的字节码文件，获取类的信息并保存起来
   b） 类加载过程(顺序): ①如果需要先加载父类的静态代码块，再加载子类的静态代码块；
   ②再加载父类的初始化代码块进行初始化和调用构造方法；
   ③最后加载子类的初始化代码块进行初始化和调用构造方法
   （总结：创建类时、先递归的从基类到派生类加载静态代码块，再依次初始化属性和调用构造方法）
   c） 加载类的时机：①当第一次创建对象时；②第一次访问类的静态成员时

18 - final 修饰符

1. final 修饰方法：用 final 修饰的方法不能被子类覆盖，但可以被继承
2. final 修饰类：用 final 修饰的类不能被继承
3. final 修饰变量：
   a) 修饰局部变量：一旦赋值就不能更改（常量）
   b) 修饰成员变量：用 final 修饰的类的成员变量，没有默认值，可以通过初始化赋值一次。如果只声明而没有初始化赋值，就必须在所有构造方法里赋值或用 static 修饰的代码块

19 -接口 interface

接口：接口是一个特殊的抽象类

注意：

）接口中所有的方法都是公开抽象方法（public abstract 方法名，可以省略不写）
(在 jdk8 后接口中可以有 default、static 方法. jdk9 后几口中可以有 private 方法) 变态不？
）接口中的属性都是公开静态常量（public static final 常量名 ，可以省略不写）
）接口中没有构造方法，而抽象类中有构造方法
）一个类实现一个接口，如果这个类不希望成为一个抽象类，则就必须实现接口中的所有方法

接口的特性：

）接口之间可以定义多继承关系（接口之间的继承仍然是用 extends，用“,”分隔）
）一个类在继承另外一个类的同时，还能实现多个接口此时多个接口之间用“,”隔开

接口的作用：

）实现多继承（注意不全是多继承，例如一个类不可能有两个方法名相同、参数表相同、放回值类型相同的两个方法，所有就要求继承多个父类时，父类如果有相同方法时，就会出错）
）接口是一个标准（提高编程的弱耦合性）
定义接口就是定义一个标准，就把标准的实现者和标准的使用者分离，从而形成弱耦合关系

20 -内部类

例如：

```
class A {                    //外部类
    class B {  }             //内部类
}
```

上面的类编译之后会生成两个独立的类：A.class　　A$B.class

内部类可以访问外部类的私有成员

集中内部类：

1、**成员内部类：**在一个类里面有一个类，例如上面的例子
创建内部类对象：先创建外部类对象，再创建内部类对象(A a = new A(); A.B b = a.new B();)

2、**静态内部类：**内部类的修饰符为 static
特点：①只能访问外部类的静态成员②可以直接创建内部类对象
创建内部类对象：A.B b = new A.B();

3、**局部内部类：**在外部类的成员方法里面定义一个类
作用范围：从定义开始，到所在代码块结束
特点：①把创建对象语句写在方法里，当外部类调用方法时创建局部内部类②局部内部类不仅可以访问外部类的成员，还能访问外部类的局部变量，但必须加上 final（常量），（1.8 以上可以不写，虚拟机会根据代码默认加上）

```
main:A a = new A();
     a.method();  //注意，应将局部类的创建在方法内

class A {
     public void method() {
          class B {     }
     }
}
```

4、**匿名内部类：**是一种特殊的内部类
条件：①实现一个接口或是继承一个类时②只会创建这个类的一个对象时
例如：在 main 函数内实现一个接口 IA：

```
IA a = new IA(){      //直接实现该几口，没有要实现该接口的类名
     @Override
     public void method() { }//实现该接口的方法
}
```

21 - Object 类

Object 是 java 中所有类的父类，Object　o = 任何对象。Object 类中的方法是所有类对象都具有的方法

Object 类的方法：

1. getClass():获得对象的实际类型，例如 getClass(animal)和 getClass(dog) 的类型不同（dog extends animal）
2. finalize():在对象被垃圾回收的时候，由垃圾收集器自动调用。当内存资源即将耗尽，不得不回应时，

才会启动垃圾回收（垃圾回收不应该在 finalize()里执行）

3. toString():返回对象的字符串形式，打印一个对象就是打印对象的 toString()方法的返回值
4. equals():判断两个对象（引用，例如 String 类型）内容是否相同

   “==” 是判断两个引用是不是指向同一对象。

   基本属性（如 int，long）可以用“==”比较值是否相等；对象类型用 equals 比较引用的地址内的内容是否相同

   equals 方法覆盖：

```
public boolean equals(Object o) {
    if(this == o) return true; //自反
    if(o == null) return false; //空对象
    if(this.getClass() != o.getClass()) return false; //判断实际类型不同
    T t = (T) o; //T 代表当前对象类型
    逐一比较属性，若全部相等和相同，返回 true;
    return false ;     //根据以上筛选返回 true 或 false;
}
```

在逐一属性比较时注意字符串要判断是否为空,如下：

```
if (name == null) {
    if (other.name != null)
        return false;
} else if (!name.equals(other.name))
    return false;
```

5. hashCode()：哈希码（public int hashCode()）

```
public int hashCode() { //简单覆盖
    将字符串属性的 hashCode 与其他数值类型属性的值相加，返回
}
```

6. wait(),notify(),线程的等待(进入阻塞，释放锁标记和 CPU)与通信(将其他线程从阻塞状态进入可执行状态，当执行完当前线程后，将锁标记归还给其他使用 wait 的线程)，在线程加锁的代码块内使用

   还有 wait(),notify() 在多线程一节的线程通信讲解

22 -包装类、及字符序列 CharSequence 接口

作用：*(使 Object 可以指向 java 内所有的数据类型)*

①使 Object 对 java 所有类型更统一

②可以区分 0 和 null（例如一个学生对象的成绩，如果 0 分，可能是考了 0，也可能是没考）

###基本类型与包装类型的转化：

int –> Integer：

```
int value = 33;
Integer int1 = new Integer(value);   //构造方法(分配空间)
Integer int1 = new Integer(“33”);//构造方法
Integer int1 = new Ingeger.valueOf(value);//静态方法(全局性)
```

Integer --> int：

```
int value = int1.intValue();
```

基本类型	包装类型
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Charactor
boolean	Boolean

总结：

基本类型转化为包装类：①通过构造方法②调用包装类静态 valueOf()方法

包装类转化为基本类型：包装类类调用 xxxValue();给对应基本类型赋值

###基本类型与 String 类型的转化

***基本类型* --> String：** *①利用””空字符串同化 ②利用静态 valueOf();（String s = String.valueOf(3);）*

String –> 基本类型： *利用包装类的静态 parseXXX(String S);(”XXX”为包装类型) 例如：int I = “123”.parseIngeger();*

###String 类型的常见方法：

charAt(int index)	**返回指定指定下标字符**
length()	**返回字符串长度**
compareTo(String s)	**字符串比较，例如 a>A**
compareToIgnoreCase(String str)	比较，忽略大小写
concat(String s)	类似于“+”
contains(String s)	是否包含子字符串
endsWith(String s)	是否以子字符串结束
startsWith(String s)	是否以子字符串开头
equals(String s)	**判断是否相同**
equalsIgnoreCase(String s)	判断是否相同，忽略大小写
indexOf(int ch)	首次出现该字符的下标
indexOf(int ch, int fromIndex)	指定位置向后，返回首次出现的下标
indexOf(String s)	首次出现该子字符串的下标
lastIndexOf(String s)	最后一次出现的下标
replace(String oldS, String newS)	**用 newS 替换所有 oldS**
split(String regex)	**分割，返回字符串数组**
substring(int start, int end)	**截取，开区间[start, end)**
toLowerCase()	转为小写
toUpperCase()	转为大写
trim()	**忽略前后空格**

CharSequence：字符序列接口

String:不可变的字符序列。（字符串长度不可变，字符串所有的方法都不会修改原字符串对象内容）

串池技术:所有字面值形式的字符串默认进入串池、或调用 intern()方法将字符串放入串池

StringBuilder:可变字符序列，有很多方法（不似 String 在改变字面值时会产生很多中间对象）

23 -日期处理和精度处理

日期处理:long time = System.currentTimeMillis(); //取 1970-1-1-零点 到 现在的毫秒数

```
Calendar = Calendar.getInstance();  (java.util.Calendar)
c.setTimeInMillis(time);

int year = c.get(Calendar.YEAR);
int month = c.get(Calendar.MONTH);
int day = c.get(Calendar.DAY_OF_MONTH);
int weekday = c.get(Calendar.DAY_OF_WEEK);
int hour = c.get(Calendar.HOUR_OF_DAY);
int minute = c.get(Calendar.MINUTE);
int second = c.get(Calendar.SECOND);

System.out.println(year + " 年 "+(month+1) +" 月 "+day+" 日 星期"+((weekday>1)?(weekday-1):"
日")+" "+hour+":"+minute+":"+second);
}
```

精度处理: System.out.println(2.0 – 1.1); //运行结果为 0.89999999

利用 BigDecimal 来处理精度丢失问题：

```
BigDecimal b1 = new BigDecimal(“2.0”); //不要直接传 2.0，不然又默认为 double 类型
BigDecimal b2 = new BigDecimal(“1.1”);
```

BigDecimal b3 = B1.substract(b2); //调用相减的方法

24 -异常处理

异常：程序运行过程中不正常的情况

异常的传递方式：

① 传递状态码方式（例如，抛出 1 , -1）：
a)包含的信息有限；
b)可能和正常结果冲突；
c)不强制要求处理。

② 传递异常对象的方式：通过 throw 手动抛出异常，相当于 return 方法抛出异常后，调用者如果不能处理该异常，则继续上抛，知道异常被处理，或直接抛给 JVM，当 JNM 获得异常后会终止程序运行，并将异常方法调用栈信息输出

Throwable (所有异常的超类)
有两种常见方法：
String getMessage():返回详细消息字符串
printStackTrace():将此 throwable 的详细方法调用栈信息输出到控制台

两种子类：

---Error 错误，严重的底层错误，无法完全避免，可以不用处理

---Exception 异常，可以处理

1. RuntimeException：运行时异常，可以完全避免，可以处理也可以不处理，优先选择避免；编译器对于程序运行时可能产生的运行时异常，不强制要求提供处理方案
2. 非 RuntimeException：非运行时异常（编译时异常，已检查异常），不可完全避免，必须处理；编译器对于程序运行时可能产生的编译时异常强制要求必须提供处理方案

自定义异常：①自定义编译时异常，需要继承 Exception ②自定义运行时异常，需要继承 RuntimeException；③需要提供至少两个构造方法，有一个无参构造

异常处理：两种方法
1.异常的声明：当一个方法无法处理可能产生的异常时，可以在方法声明处声明可能抛出的异常类型，当前的方法不在负责处理该异常，交由调用者处理
2.try {可能出现异常的代码块} catch（异常类型 e）{异常出现后处理代码块}：捕获异常，并处理（当异常处理后，程序会继续向下执行，可能跳过一些代码块）

注：try-with-source 语法格式（***jdk7.0***）：

```
try(此处放入实现了 AtoCloseable 接口的对象，以可以自动释放资源，例如 InputStream is = new FileInputStream("路径");){
    //其他的代码块
}catch() {}
```

说明：try（xxx; xxx;）内放的是需要释放资源的语句，即实现了 AutoCloseable 接口的对象构造语句，释放顺序是从后到前。此种语法可自动释放或关闭资源，不写 finally

案例： try {①；②；③；}

若②处可能出现**异常类型 1**、**异常类型 2**、**异常类型 3**
若是**异常类型 1**、执行的代码块有①②④；若是**异常类型 2**、执行的代码块有①②⑤；若是**异常类型 3**、执行①②后 catch 无法捕获，向上抛出

catch(异常类型 1　e) {④}
catch(异常类型 2　e) {⑤}

结论：当 try 块中的代码产生异常时，会通过 catch 自上而下的尝试匹配，一旦匹配成功，泽后续的 catch 不知尝试执行；当 catch 的异常类型有继承关系，优先捕获子类异常

finally ：该代码块中的代码表示无论 如何都一定要执行（通常会将关闭流、连接等代码现在此处）

25 -集合框架、泛型<T>

1).Collection

集合:一类特殊的对象(用于保存多个对象的对象)
Collection：元素必须是引用类型，在遍历时需要 for each 方式（因为数据没有下标）
Collection 重要的两个子类：①List　②Set

Collection 集合的重要的方法：
add（E e）　顺序的添加元素到集合中
remove（Object o）　从集合移除
size（）返回 collection 集合中的元素数量
Object[]　toArray（）返回包含 collection 中所有元素的数组
void clear（）　移除 collocation 中所有的元素
boolean contains（）　判断集合是否包含特定元素

注：
在遍历时要使用 for(集合类型 引用名：数组或集合)；

2).List

元素内容可以重复，有顺序也有下标（可用 for 遍历，也可用 for...each 遍历）
常见实现类：
（1）ArrayList　jdk1.2　底层数组实现　查询快，增删慢　线程不安全　轻量级实现　效率高
（2）LinkedList　jdk1.2　底层链表实现　查询慢，增删快
（3）Vector　jdk1.1　底层数组实现　查询快，增删慢　线程安全　重量级实现　效率低

常见方法：
void add（）顺序添加
void add（int index，E element）　指定下标插入
E get（int index）指定下标获取值
E remove（int index）指定下标删除
E set（int index，E element）　指定下标更新元素
size（）返回长度
indexOf（Object o）　获取从前往后第一次出现的下标
List<E>　subList（int fromIndex，int toIndex）　返回从 fromindex 到 toindex-1 处所有元素组成的子集合

3).Set

元素内容不可以重复，储存没有顺序，而且无下标（只能用 for…each 遍历）

常见实现类：

（1）HashSet

添加元素过程：①元素调用 hashCode（），获取哈希码，然后对长度求模（%），获取下标（随机性）②如果在 index 处为空，则直接添加；否则使用 equals 比较两个元素是否相同（可自己覆盖其 equals 方法），如果相同添加失败，如果元素不同，则使用链表和前面的元素连起来

注意：开发中往往可能需要覆盖父类的 hashCode（）和 equals（）方法；

如果是自定义类型，为保证元素的内容相同就是相同元素，我们必须重新 hashCode 和 equals 方法，其要求：（1）内容相同的对象（利用 equals 方法），必须放回相同的哈希码（2）内容不同的对象，极可能返回不同的哈希码

a) LinkedHashSet(继承自 HashSet)

自定义类型，为保证元素内容不重复，必须必须重写 hashCode 和 equals 方法；添加是有顺序的，遍历时和添加的时的顺序一致

（2）SortedSet

b) TreeSet(继承自 SortedSet)

可以根据元素的内容自动升序排序，并且不重复。在自定义该集合类型时，为保证元素内容不重复，必须实现 conparable 接口并实现 compareTo 方法

常见方法：

void add（E e） 添加元素，存储顺序随机

E remove（Object o） 指定元素删除

int size（） 返回长度

void clear（）清除所有元素

boolean contains（E e）判断是否包含

HashSet	*jdk1.2*	*底层实现是散列表*
LinkedHashSet	*jdk1.2*	*底层实现是散列表和链表*
TreeSet	*jdk1.2*	*底层实现是红黑树*

4).泛型

参数化类型、模板编程

定义类型时，使用类型变量代替；使用泛型类型时，必须传入确切的类型，泛型默认类型时 object 类型，利用泛型技术可以构建元素类型安全的集合

```
List<Object> list = new List<>();  //系统会根据传入的参数进行推断其类型
List<T>.....此处的 T，可以是自定义的类型，比如自定义类
```

在定义类时也可以用泛型：

```
public class Student<T> {
    public T method(){return T;}
}
```

泛型可约定范围

<T extends Number> 约定 T 的类型为数值类型

Number 是 short int double float long byte 的父类

5).Map

集合 Map 是单独的集合，并不是 Collection 的子类集合，其常见的实现类有：

（1）HashMap
（2）LinkedHashMap
（3）TreeMap
（4）Hashtable
（5）Properties　　*继承自 Hashtable，键值对为 String,大多用于配置文件的存取*

Map：元素有 key（键）和 value（值），其中的键（key）不能重复，而值可以重复
map 内的键-值对存储是无序的，没有下标的

HashMap：

常见方法：①put（Object K,Object V）②remove（key）删除 key 键的键值对③size（）④get（key）⑤containsKey（Object Key）是否包含 key⑥containsValue（Object Value）是否包含值⑦values（）获取所有的值⑧KeySet（）获取所有的键

遍历方法：

```
Map<T, T>    map = new HashMap<>();
```

（1）对键（key）遍历

a) for（T　t ：　map.KeySet() ）{}　　//获取的 key 集合给 o 引用
b) Set<T> set = map.KeySet();　　//用 set 集合实现，可以通过键找到值
　　for（T　t：set ）{}

（2）对值（value）遍历

a) for（T　t ：　map.ValueSet() ）{}　　//对值进行遍历
b) Collection<T> c = map.values();　　//通过 collection 集合实现（值因为可能有重复），通过值无法找到键
　　for（T　t：　c）{}

（3）对键-值（key, value）对遍历

```
Set<Map.Entry<K, V>>    set = map.entrySet();
for（Map.Entry<K, V>    me : set）{}
```

注：　//Map.Entry<T, T>是一个类类型 ，它有两个重要的方法：getKey() / getValue();　//获取建和值

6.了解内容

（1）HashMap	jdk1.2	底层是散列表	线程不安全，轻量级实现，允许 null 键-值对
（2）LinkedHashMap	jdk1.2	散列表+链表	保证键-值对的遍历时顺序和添加时顺序一致
（3）TreeMap	jdk1.2	红黑树	可以根据 key 的内容对键值对进行自动升序排序
（4）Hashtable	jdk1.0	底层是散列表	线程安全，重量级实现，不允许有 null 键-值对
（5）Properties	jdk1.0	Hashtable 子类	键-值对都是 String，专门用于读取配置文件的信息

26 -多线程

1).进程

在操作系统中可以并发执行的一个任务，采用分时间片（微观串行，宏观并行），由操作系统调度

2).线程

是进程中并发执行的一个顺序流程

线程组成：

a.CPU 时间片，由操作系统调度

b.内存（JVM 内存）：堆空间（保存对象，即实例变量，线程共享）、栈空间（保存局部变量，线程独立）

c.代码，是由程序员决定

3).Thread 类

此类型的对象就代表了一个线程。线程调用 start 启动后自动执行 run 方法，run 方法是自定义代码

用法：

使用继承 Thread 的类（推荐使用匿名内部类）

```
class MyThread extends Thread{public void run(){   该线程的自定义流程；}}
main: Thread t1 = new MyThread();
        t1.start();   //线程启动
        t1.run();     //线程并未启动，知识最普通的函数调用
```

使用实现 Runnable 接口的类（推荐使用匿名内部类），缺点是不能抛异常,无返回值；可参考下文 Callable 接口

```
class MyThread implements Runnable{public void run() {   该线程的自定义流程}}
main:Runnable rthread = new MyThread();
        Thread t2 = new Thread(rthread);
        t2.start();

//匿名内部类：
Thread t = new Thread(new Runnable() {
      public void run() {}
});
```

4).线程同步（最重要，有线程池，fork-join 框架）

在多线程环境下，并发访问同一个对象（临界资源），由于彼此间切割此对象的原子性操作，对象的状态就会处于一个不一致的状态。在 java 中，任意对象（Object o）都有一把互斥锁标记（syjchronized（o））用于分配线程

①. synchronized：（同步）互斥锁

第一种：同步代码块

```
Object o = new Object();
synchronized(o) { 需要同步的代码块 1}
synchronized(o) { 需要同步的代码块 2}
```

总结：当一个线程要执行同步代码块时，必须获得锁标记对象的互斥锁标记，没有获取到时就进入线程阻塞，知道得到互斥锁标记。线程执行同步代码块，执行完毕后自动释放互斥锁标记，归还给锁对象。

线程是否同步，取决于是否争用同一锁对象

第二种：同步方法

```
public synchronized void method() {}
```

总结：当线程要执行同步实例方法时，必须获得当前对象的互斥锁标记，不能获得就进入阻塞。当获取互斥锁标记执行完同步方法，自动释放互斥锁标记，并归还给对象。

一个线程可以在持有互斥锁标记的前提下获取其他锁对象的互斥锁标记

②. Lock:锁接口，提供了比 synchronized 更广泛的所定义语句（有个实现类为 ReentrantLock）

ReadWriteLock:读写锁，持有一对读锁和写锁，（有一个实现类 ReenTrantReadWriteLock）

读锁：允许分配给多个线程（并发） readLocke();

写锁：最多只允许分配给一个线程（互斥）， writeLock(); *读锁和写锁两者互斥*

总结：当线程持有读锁时，不能分配写锁给其他线程，但可以分配读锁；同样在线程持有写锁时，也不能分配读锁和写锁给其他线程。（*读读并发、写写互斥、读写互斥*）

用法：

```
ReadLock rwl = new ReentrantReadWriteLock();
Lock readLock = rwl.readLock();  //读锁
Lock writeLock = rwl.writeLock(); //写锁
readLock.lock();
  中间为获得读锁的代码块，如果有 try/catch 时，可将 readLock.unlock();写在 finally 代码块里
readLock.unlock();

writeLock 用法与此相同
```

③.join()方法

如有线程 t1，如果有线程 t2 要在 t1 执行完再执行时，则可在 t2 的代码块中添加 t1.join()；语句

例如：在 main 函数中有线程 t1、t2、t3，当线程执行完在执行输出语句时：

```
public static void main (String[] args) {
    t1.join();  //将三个线程添加到主线程，并且先执行这三个线程
    t2.join();
    t3.join();
    System.out.println("输出语句");
}
```

④.线程池 ExecutorService es = Executors.newFixedThreadPool(int n);//创建线程池，并设置固定并发线程的个数 n，

将实现 Callable 接口*(Callable<T>,实现方法 public T call(){return null})*的线程提交（submit）到线程池中，

再用 Future<T>异步接收线程结束后的结果，用 get()方法获取

用法：

```
Callable<Integer> t1 = new Callable<Integer>(){
    public Integer call() { return null;} //需要实现 call 方法
};
ExecutorService es = Executors.newFixedThreadPool(2);//可并发两个线程
Future<Integer> f1 = es.submit(t1); //将 t1 提交到线程池，并将执行后的结果给 Future 对象
Future<Integer> f2 = es.submit(t2); //如果有两个，那么 f1，f2 是异步执行，互不干扰
System.out.println(f1.get());//将执行的结果通过 future 对象的 get()方法获取
```

⑤.fork-join 框架（重要，自从 jdk1.7），也是线程池

思想：分治-归并算法，大任务--->小任务--->将小任务的结果合并成完整结果

用法：

```
class Task extends RecursiveTask<T> {
    protected   T   compute() {
        return T 类型数据或 null;
    }
}
main: ForkJoinPool fjp = new ForkJoinPool();
      Task t1 = new Task();
      fjp.invoke(new Task());     //将线程放入线程池
      与此同时可以在执行再一个当前的线程 t2
      Task t2 = new Task();
      T tmp2 = t2.compute();   //将线程 t2 执行后的结果返回给 tmp2
      T tmp1 = t1.join();          //在线程 t2 中将 t1 执行后的结果返回给 tmp1
```

5).实现线程安全的方法

1).加锁：synchronized，互斥锁，对对象加锁，缺点是并发效率低

2).锁分级：ReadWriteLock，分配写锁时，不分配读锁；写锁未分配时，可分配多个读锁

3).锁分段：ConcurrentHashMap，对整个 Map 加锁，分 16 个片段，分别独自加锁（**jdk5 – jdk7**）

4).无锁算法：CAS 比较交换算法（利用状态值）

ConcurrentHashMap，利用 CAS 算法，自从 **jdk8**

5).尽量回避临界资源，尽可能的使用局部变量

6).线程间的通信（等待 -- 通知 机制）

例如：有两个 t1，t2 线程，有对象 Object o = new Object();

t1: 如果使用 o.wait();必须出现在对 o 加锁的同步代码块中，此时，t1 将会释放它拥有的所有锁标记，并进入 o 的等待队列(阻塞)和释放 CPU 资源

synchronize(o) { o.wait(); }

t2: 此时使用 t1 所释放的锁标记，利用 o.notify() / o.notifyAll()，此方法也有放在对 o 加锁的同步代码块中，t2 会从 o 的等待队列中释放一个或全部线程

wait 和 sleep 方法的区别是：wait 会失去锁标记，而 sleep 不会失去锁标记

代码例子：

```
/**
 * 两个线程：  一个输出1 - 52
 *             一个输出A - Z
 *   效果：1 2 A 3 4 B … 52 Z
 */
final Object o = new Object();
Thread t1 = new Thread() {
    public void run() {
        synchronized (o) {
```

```
                    for (int i = 1; i <= 52; i += 2) {
                        int tmp = i + 1;
                        System.out.print(i + " " + tmp + " ");
                        o.notifyAll();
                        try {
                            o.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        };
        t1.start();
        Thread t2 = new Thread() {
            public void run() {
                synchronized (o) {
                    for (char i = 'A'; i <= 'Z'; i++) {
                        System.out.print(i + " ");
                        o.notifyAll();
                        try {
                            o.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        };
        t2.start();
        t1.join();
        t2.join();
```

27 -File

java.io.File 包：代表了一个文件或目录，并不创建

用法：　File f = new File(“D:/test.txt”);

　　File f = new File(“D:/test”, “.txt”);

　　File f = new File(“D:\\test.txt”);　　// “\”转义字符

常用方法：

createNewFile（）：创建新文件

mkdir（）：创建新目录

delete（）：删除

exists（）：判断文件是否存在

isDirectory（）：判断文件对象是否为目录

isFile（）：判断文件对象是否为文件

getAbsolutePath（）：获得对象绝对路径

getName（）；获取文件（目录）名
listFiles（）；列出目录中所有的内容
listFiles（FileFilter f）；增加过滤器 （例如：用过滤获取一个目录里所有的 java 文件）

FileFilter 接口：实现是需要实现其 accept（File f）方法，返回类型为 boolean
使用—匿名内部类实现 FileFilter：

```
File[] f = new File("D:\\test.txt").listFile(new FileFilter() {
    public boolean accept(File f) {

        return false;
    }
});
```

28 -I / O 流

1).流

是一个对象，是 JVM 与外部用来传递数据的

I / O 编程步骤：

a 创建节点流
b 封装过滤流
c 读写数据
d 关闭流

有三个分类：

a.按数据方向：输入流、输出流
b.按数据单位：字节流、字符流
　字节流：以字节为单位，可以处理一切数据
　字符流：以字符为单位，只能处理文本数据
c.按流的功能：节点流、过滤流
　节点流：实际传输数据的流
　过滤流：给节点类怎去功能

2).字节流

字节流父类：InputStream(相对 JVM，从外部读到 JVM) /OutputStream

文件字节流（节点流）：FileInputStream/FileOutputStream
①文件输出流：FileOutputStream
常见的方法:
　write(int a);写出一个字符到文件里，例如写 A：write(65);
　write(byte[] b); 写出字节数组到文件
　write(byte[] b, int startIndex, int length); 写出固定长度字节数组到文件
　close(); 关闭流
使用方法：

```
OutputStream os = new FileOutputStream("D:/test.txt", true);//第二个参数是否追加，不追加则覆盖原有文件
```

os.write(65);

②文件输入流：FileInputStream

常见的方法: 返回-1 结束

int read(int a); 从文件读入一个字符到 JVM,

int read(byte[] b); 从文件读入字节数组

int read(byte[] b, int startIndex, int length); 从文件读入固定长度字节数组到

使用方法：

InputStream os = new FileInputStream("D:/test.txt");//第二个参数是是否追加，若不追加则覆盖原有文件

System.out.prinltn((char)os.read());

数据流：DataOutputStream/DataInputStream 向文件写、读八种基本类型数据和 String

常见方法：

writeUTF(); 写出字符串类型

writeLong(); 写长整型

writeDouble(); 写 double 类型

readUTF(); 读入字符串类型

readLong(); 读长整型

readDouble(); 读 double 类型

用法：InputStream is = new FileInputStream("D:\\test.txt"); //创建节点流

DataInputStream dis = new DataInputStream(is); //将节点流封装为数据流

dis.readLong(); //读文件中的长整型数据

dis.close();

I / O 增加缓冲区：BufferedInputStream、BufferedOutputStream

最后一步是清空缓冲区（flush）

用法：FileOutputStream fos = new FileOutputStream("D:\\test.txt");

BufferedOutputStream bos = enw BufferedOutputStream(fos);//增加缓冲区

bos.write('A');

bos.flush(); //清空缓冲区，将数据写到文件里

多层包装：

FileOutputStream fos = new FileOutputStream("D:\\test.txt");

BufferedOutputStream bos = enw BufferedOutputStream(fos);//增加缓冲区

DataOutputStream dos = new DataOutputStream(bos); //增加过滤

PrintStream 既有缓冲，又有过滤，例如常见的 System.out

可以写八种基本类型

管道流：PipedInputStream / PipedOutputStream 用来在线程间进行数据交换

RandomAccessFile:随机访问文件

3).字符流

编解码问题：由于字符的编码与解码方式不统一，可能造成乱码

String str = "abc";

byte[] b = str.getBytes(“GBK”); //以 GBK 编码方式转化
String s = new String(b, “GBK”); //再以 GBK 编码方式接受
乱码问题解决：如果编码和解码方式不统一，可在一次用错误的顺序从小编解码

字符流：Reader / Writer (处理文本)
文件字符流：FileReader / FileWriter
FileReader fr = new FileReader(“D:\\test.txt”, true);

字符缓冲流：BufferedReader / BufferedWriter (一般在写字符串是用 PrintWriter)
需要清空缓存 flush;

桥转换：InputStreamReader / OutputStreamWriter
实现字节流与字符流的转换
FileOutputStream fos = new FileOutputStream(“D:\\test.txt”);
Writer s = new OutputStreamWriter(fos, “GBK”); //将字节流转换为字符流
BufferedWriter bw = new BufferedWriter(w);//添加缓冲区

或
PrintWriter pw = new PrintWriter(“文件路径”); **//与上面的三行代码的功能类似，比较简便**
pw.println(“asdg”); //写出一行

4).对象的序列化

将对象通过流进行传递
ObjectOutputStream / ObjectInputStream
只用实现了 Serializable 接口的对象才能进行序列化，并且也需要其对象属性也实现该接口

用 transient 修饰的属性为临时属性，不会参与对象的序列化

用法：
class Student implements Serializable {假设里面有四个属性和其他方法}
main 函数中：
Student s = new Student(“jiang”, 23, “男”, 14823412344);
OutputStream os = new FileOutputStream(“D:\\test.txt”);
FileWriter fw = new OutputStreamWriter(os, “GBK”);
ObjectOutputStream oos = new ObjectOutputStream(fw);
oos.writeObject(s); /将对象写出到文件内；

5).对象序列化扩展、对象的克隆(clone)

对象序列化扩展：
对象序列化-版本号作用： public static final long serialVersionUID = 666L;
1.为了类的变化后，在对象的的序列化（存对象）和反序列化（读对象，*先加载类*）时，尽可能的不影响其结果，可以为类添加 SerialVersionUID 私有静态长整型的常量。控制类的版本号，当类的版本号修改时，意味着之前的序列化数据全部作废
2.当反序列化一个类对象时，如果这个类实现了 Serializable 接口，那么可以直接还原对象，如果这个类没有实现 Serializable（这个类必是父类），那么就需要其调用构造方法

对象的克隆：

即对象的复制：在内存中复制出一个内容完全相同的对象，调用 Object 类中 clone()方法，返回 Object 对象

1.覆盖 clone()方法：clone 是受保护的，复写时应将其修饰符提高到 public

2.需要被克隆的类必须实现 Cloneable 接口(标记接口)，

浅拷贝：Object.clone();

深拷贝：对象复制过程中，对象的属性如果又是对象，则递归的进行复制；实现深拷贝，利用对象序列化自定义克隆方法时，可以不实现 Cloneable 接口

使用 ByteArrayOutputStream 将对象写入字节数组，再利用 ByteArrayInputStream 将对象读回，此方法在内存中操作，很快

6).Commso – io.jar 工具包

jar 引入项目：右键对 jar 包 BuildPath 再点击 Add to BuildPath

类：

FileUtils

方法：

(static) copyFile(scrFile, destFile); //参数为路径 复制文件

29 –枚举

枚举：Enum 子类，final 修饰 自 jdk5.0

1).创建枚举类

①：是个类，并且这个类的对象是构造好的，不允许用户构造该类的新对象

常用方法：

values(); 获取枚举类内的所有枚举值，返回当前枚举类类型的数组

name(); 获取枚举值(枚举类的对象)的名字

ordinal();获取枚举值的序号

用法：

```
//enum 继承自Enum
enum Season { //创建枚举类Season
    SPRING , SUMMER , AUTUMN ,WINTER
}
main:   Season s = Season.SUMMER;
        System.out.println(s.name());//打印s对象的名称
        System.out.println(s.ordinal());//打印s对象所在的序号
        //遍历枚举类内的枚举值
        Season[] arrSeason = Season.values();
        for(Season ss : arrSeason) {
            System.out.println(ss.name());
        }
```

2).枚举类的构造方法

②：是个特殊的类，是 Enum 的子类，它所有的构造方法是私有的(private，不能被子类继承的特性)，可以有属性，一旦有有参构造，则需要其内部的枚举值(枚举对象)也需要构造

用法：

```
//enum 继承自Enum
enum Season {
    SPRING("春季", 1, 3),
    SUMMER("夏季", 4, 6),
    AUTUMN("秋季", 7, 9),
    WINTER("冬季", 10, 12); //构造后，最后一个对象需要有“；”

    String name;
    private int startMonth;
    private int endMonth;
    //省略getters/setters
    Season(String name, int startMonth, int endMonth) {//构造方法，默认private
        this.name= name;
        this.startMonth = startMonth;
        this.endMonth = endMonth;
    }
}
main(): Season s = Season.SUMMER;
        System.out.println(s.getName() + “开始月份是” + s.getStartMonth());
```

3).枚举类的抽象方法

③：枚举类的内部可以有抽象方法。枚举类的本身是被 final 修饰的，不能被继承，故不存在子类实现其抽象方法。所以其抽象方法是供内部的枚举值(枚举对象)实现的

用法：

```
//enum 继承自Enum
enum Season {
    SPRING("春季", 1, 3) {
        @Override  //覆盖抽象方法
        public void describe() {
            System.out.println("春燕衔春泥");
        }
    },
    SUMMER("夏季", 4, 6) {
        @Override //覆盖抽象方法
        public void describe() {
            System.out.println("夏天很炎热");
        }
    },
    AUTUMN("秋季", 7, 9) {
```

```
            @Override //覆盖抽象方法
            public void describe() {
                System.out.println("秋天落叶黄");
            }
        },
        WINTER("冬季", 10, 12) {
            @Override //覆盖抽象方法
            public void describe() {
                System.out.println("大雪深数尺");
            }
        };

        String name;
        private int startMonth;
        private int endMonth;
        //省略getters/setters
        Season(String name, int startMonth, int endMonth) {//构造方法，默认private
            this.name= name;
            this.startMonth = startMonth;
            this.endMonth = endMonth;
        }

        public abstract void describe();//一个方法实现描述
    }
    main(): Season s = Season.SUMMER;
            s.describe ();
```

30 –网络编程

1).认识网络名词

IP 地址：表示网络上的一台主机，逻辑地址
MAC 地址：表示网络上的一台主机，物理地址
端口：标识主机中的一个进程 0－65535， 1024 一下为预留端口
协议：通信双方之间的约定和标准
物理层 -> 数据链路层 -> 网络层 -> 传输层 -> 会话层 -> 表示层 -> 应用层

传输层：
TCP:传输控制协议，面向连接的，是可靠的
UDP:用户数据报协议，非面向连接到，不可靠的

2).TCP 编程

TCP: Socket 编程（套接字） ServerSocket Socket (服务器端用两个)
java 包：java.net.*;

用法：如下

TCP: 模拟客户端与服务器端的通信代码：先启动服务器端

客户端（Client）：

```
public static void main(String[] args) throws Exception {
    Socket s = new Socket("192.168.77.13", 9000);  //向服务器发起连接，参数分别是 ip 和端口
    InputStream is = new s.getInputStream();
    BufferedReader br = new BufferedReader(new InputStreamReader(is));
    for(int i = 1; i <= 20; i++) {
        String str = br.readLine(); //从服务器端读
        System.out.println(str);
    }
    s.close();
}
```

服务器端（Service）

```
public static void main(String[] args) throws Exception {
    ServerSocket ss = new ServerSocket(9000); //服务器进程绑定端口为 9000
    Socket s = ss.accept();//接受访问
    OutputStream os = s.getOutputStream();
    PrintWriter out = new PrintWriter(os);
    for(int I = 1; I <= 20; i++) {
        out.println("Hello" + i); //向客户端写
        out.flush();
        Thread.sleep(200);
    }
    s.close();
}
```

3).UDP 编程

UDP:Socket 编程　　DatagramSocket　　DatagramPacket

java 包：java.net.*;

用法：如下

UDP: 模拟客户端与服务器端的通信代码：先启动服务器端

客户端（Client）:向服务器端送一封信

```
public static void main(String[] args) throws Exception {
    //先发一封信
    DatagramSocket ds = new DatagramSocket();
    String str = "这里是客户端的信";
    byte[] b = str.getBytes(); //可以设置编码方式

    DatagramPacket sendLetter = new DatagramPacket(b, 0, b.length, InetAddress.getLocalHost(), 9090);//发送信，
前三个参数为信的内容，后两个参数分别为本地 IP 和送信端口端口
    ds.send(sendLetter);

    //接受 30 封信
    for(int I = 0; I < 30; I++) {
```

```
            DatagramPacket receiveLetter = new DatagramPacket(new byte[100], 0, 100);
            ds.receive(receiveLetter);
            byte[] bs = receiveLetter.getData();
            int offset = receiveLetter.getOffset();
            int length = receiveLetter.getLength();
            String newStr = new String(bs, offset, length);//将数组转化为字符串
        }
        ds.close();
    }
```

服务器端（Server）:从客户端端收信，返回 30 封信

```
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket(9090);
        DatagramPacket receiveLetter = new DatagramPacket(new byte[100], 0 , 100);//将接收的信内容存入 byte 数组
        ds.receive(receiveLetter);
        InetAddress address = receiveLetter.getAddress(); //获得客户端地址
        int port = receiveLetter.getPort();//获取来信客户端的端口

        //给客户端发送 30 封信
        forI(int I = 0; I <= 30; i++) {
            String str = “这是服务器端的信”;
            byte[] b = str.getBytes();
            DatagramPacket sendLetter = new DatagramPacket(b, 0, b.length, address, port);
            ds.send(sendLetter);
            Thread.sleep(200);
        }
        ds.close();
    }
```

4).URL 编程

URL 编程是在 应用层， 统一资源定位器

格式(字符串)：协议名://主机名:端口号/相对路径

java 包 java.net.*;

用法：

```
        public static void main(String[] args) throws Exception {
            URL url = new URL(“http://192.168.77.13/corejava.txt”);//创建 url
            URLConnection uc = url.openConnection(); //打开连接
            InputStream is = uc.getInputStream(); //读取
            BufferedReader br = new BufferedReader(new InputStreamReader(is));//桥转换
            while(true) {
                String str = br.readLine();
                if(Str == null)
                    break;
                System.out.println(str);
            }
            in.close();
```

```
}
```

31 –反射

反射：是底层技术 （开发工具和框架，使代码更通用）
java.lang.reflect//反射包

1).类对象 Class

是类加载的产物，封装了一个类的所有信息（类名，父类，接口，属性，方法，构造方法…）

类加载: 当 JVM 第一次使用一个类的时候,需要读取这个类对应的字节码文件,获取类的信息并保存起来
类对象: 记录类的信息的对象. 类加载后,将类的信息封装成类对象,保存在方法区中

获得类对象的三种方法：

① Class c1 = HashMap.class; //获取类对象

② Object o = new HashMap();
Class c2 = o.getClass();

③ String className = “java.util.HashMap”; //写类的全名，带包
Class c3 = Class.forName(className); //Class.forName，加载

常用方法：

getName()	获取全类名
getSimpleName()	获取简单类名，不含包名
getSuperClass()	获取父类的类对象
getInterfaces()	获取所有实现了的接口类对象，返回 Class[]
getField()	获取一个公开属性对象，
getDeclaredField()	获取一个属性，包括非公开的
getFields()	获取所有的公开属性对象，包括继承自父类的属性
getDeclaredFields()	获取本类中所有的属性对象，
getMethod(“方法名”，参数表.class)	获取一个公开方法，注意参数表的位子
getDeclaredMethod(“方法名”，参数表.class)	获取一个本类的方法，包括非公开的
getMethods()	获取所有的公开方法，包括父类的
getDeclaredMethods()	获取本类所有的方法
getConstructors()	返回所有的构造方法的对象信息

用法：例

```
Class c = HashMap.class;
System.out.println(c.getName());                    //带包名，全名
System.out.println(c.getSimpleName());                  //不带包名，简单名
System.out.println(c.getSuperclass().getName());    //父类的带包名

Class[] cc = c.getInterfaces();                         //获得实现的接口
for(int i = 0; i < cc.length; i++) {                //遍历以实现的接口并打印名字
    System.out.println(cc[i].getName());}
```

2).Field

Field：封装一个类对象的所有属性信息

常用方法：

getFields();返回类中所有的公开属性，包括父类中的属性，返回 Field 数组

getDeclaredFields();返回本类中所有属性，包括非公开属性，返回 Field 数组

用法：

class Student { public String name; public int age;}

main();

```
Class c = Student.class;
Field[] f = c.getFields();
for(int i = 0; i < f.length; i++) {
    System.out.println(f[i]);
}
```

3).Method

Method：封装一个类对象的所有方法信息

常用方法：

getMethod("方法名"， 此方法的参数表（例如 Object.class）***);返回类中的一个方法;***

getDeclaredMethod("方法名"， 此方法的参数表（例如 Object.class）***); 获取的一个包括非公开的方法***

getMethods();返回类中所有的公开方法，包括父类中的方法，返回 Method 数组

getDeclaredMethods();返回本类中所有方法，包括非公开方法，返回 Method 数组

用法：

①
```
Class c = HashMap.class;
Method m1 = c.getMethod("put");//返回无参 put 方法
        m1 = c.getMethod("put", Object.class, Object.class);//返回有两个 Object 类型参数的 put 方法
System.out.println(m1.getReturnType().getName());//打印该方法的返回值类型名称
```

②
```
Method[] m2 = c.getMethods(); //返回类对象c内的所有公开方法，包括父类的方法
for(int i = 0; i < m2.length; i++) {
    System.out.println(m2[i].getName());
}
```

③
```
Method[] m3 = c.getMethods(); //返回类对象c内的所有非公开方法
for(int i = 0; i < m3.length; i++) {
    System.out.println(m3[i].getName());
}
```

3).Constructor

Constructor：封装一个类对象所有构造方法信息

常用方法：

getConstructos();返回类中所有的构造方法，包括父类中的方法，返回 Constructor 数组

用法：

```
Class c = Student.class;
Constructor[] ct = c.getConstructors(); //获取所有构造方法
System.out.println(ct[0].getName());
```

4).利用反射处理类的对象(实例化对象、调用方法)

动态对类做操作：

①对Class对象调用newInstance(),可以创建该类的对象调用无参构造方法

```
Class c = Class.forName("com.baizhi.Student"); //注意抛异常
c.newInstance();
```

②对Method对象调用invoke(Object o),对o调用方法

```
Method m = 对象.getMethod("方法名", 参数类型.class);
m.invoke(o);
```

③可以调用私有方法，但必须先将方法的可访问性属性改为true（setAccessible(true)）

```
Method m = 对象.getDeclaredMethod("方法名", 参数类型.class);//私有方法
m.setAccessible(true); //修改可访问性
m.invoke(o);
```

④访问并设置对象的属性

用法：

```
class Student {
    public void study() { System.out.pritln("在学习");}
    public String study(String course) { return "学习课程：" + course; }
}
main():
    //Student o = new Student();
    Class c = Class.forName("包.Student");//加载类 Student
    Object o = c.newInstance();//创建类的对象

    //o.study();调用无参方法
    Method m1 = c.getDeclaredMethod("study"); //获取无参的 study 方法
    m1.invoke(o); //调用方法

    //o.study();调用有参方法
    Method m2 = c.getDeclaredMethod("study", String.class); //获取方法
    Object result = m2.invoke(o, "CoreJava"); //调用有参构造，并将返回值赋给 o
    System.out.println(result);
```

32 –注解（标注）

标注：Annotation 描述代码的**代码**，给计算机识别的
传统注释：描述代码的**文字**，给用户看的

标记标注　　@标注名
单值标注　　@标注名（属性名 = 属性值）

普通标注　　@标注名（属性 1 = 值 1， 属性 2 = 值 2， …）

注意：

特例：对于单值标注（@标注名(属性名 = 属性值)），如果属性名为 value，可简化为@标注名（属性值）

用法：

创建标注：File -> new -> Annotation

----------第一步：创建自定义标注 myAnnotation----------

```
package jwnming;
@Target(value = { }) //Target 是指明能标注什么，是枚举，ElementType.TYPE(标注类), ElementType.Field(标注属性), //ElementType.CONSTRUCTOR(标标注构造方法), ElementType.METHOD(标注方法)
@Retentioin(value = RetentioinPolicy.RUNTIME ); //也是枚举类型

public @interface myAnnotation{
    public String name() default “这是标注”;  //定义属性，类似方法，名后加(); 默认值为“这是标注”
}
```

```
import java.lang.annotation.*;

@Target(value = {ElementType.TYPE,ElementType.FIELD}) //说明能标注什么
@Retention(value = RetentionPolicy.RUNTIME)

public @interface testAnnotation {
    public String name() default "test";
    public int intValue() default 100;
}
```

----------第二步：使用自定义标注----------

```
@myAnnotation //可以标注类（@Target({ElementType.TYPE})），取默认值“这是标注”
public class test {
    @myAnnotation(“标注值”) //可以标注属性（@Target({ElementType.TYPE, ElementType.Field})）
    String name;
    @myAnnotatio//标注构造方法(@Target({ElementType.TYPE, ElementType.Field, ElementType.CONSTRUCTOR}))
    public test() { }
}
```

33 –设计模式

特定的编程套路

1).单例模式

单例模式 ：一个只有单一对象的类

①饿汉式　　空间性能低

```
class A{
    private static final A instance = new A();
    public static A newInstance() {
        return instance;
    }
    private A(){}
```

```
}
```

②懒汉式　　时间性能低

```
class B{
    private static B instance = null;
    public static synchronized B newInstance() {
        if (instance == null) instance = new B();
        return instance;
    }
    private B() {}
}
```

③融合饿汉式和懒汉式的优点

```
class C{
    private static class Holder{
        static final C instance = new C();
    }
    public static C newInstance() {
        return Holder.instance;
    }
    private C() {}
}
```

2).工厂模式

对象的创建和对象的使用分离,履行开闭原则

版本一: 创建对象和使用对象分离　职能单一(各司其职)

```
static Dog create() {
    return new Dog();
}
```

问题：当 Dog 变为 Cat 时，代码需要修改　违反开闭原则

版本二: 当 Dog 变为 Cat 时,函数声明不用修改

```
static Animal create() {
    return new Cat();
}
```

问题: 函数实现还是需要修改

版本三: 利用参数,选择创建哪类对象

```
static Animal create(int i) {
    if ( i == 0) return new Dog();
    if ( i == 1) return new Cat();
    return null;
}
```

问题: 当有新的子类出现时,依然需要修改工厂方法的实现代码

版本四: 利用反射，通过类名创建对象

```
    static Animal create(String className) {
        try {
            Class c = Class.forName(className);
            return (Animal)c.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

问题: 需求固化在代码中

版本五: 需求集中管理, 集中配置

```
    static Animal create() {
        try (FileInputStream fis = new FileInputStream("config.txt")){
            Properties ps = new Properties();
            ps.load(fis);
            String className = ps.getProperty("Animal");
            Class c = Class.forName(className);
            return (Animal)c.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

问题: 只能创建 Animal 对象, 不够通用

版本六(最终版本):利用泛型技术, 实现通用工厂　　参数: 接口的类对象

```
    static <T> T create(Class<T> c) {
        try (FileInputStream fis = new FileInputStream("config.txt")){
            Properties ps = new Properties();
            ps.load(fis);
            String interfaceName = c.getSimpleName();
            String className = ps.getProperty(interfaceName);
            Class<?> cc = Class.forName(className);
            Object o = cc.newInstance();
            return c.cast(o);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

34 –其他

1).格式化时间字符串格式

```
Date date=new Date();
```

```
String date1=String.format("%tF", date);
System.out.println(date1);

System.out.println(String.format("%tY",date)+"年");
System.out.println(String.format("%tB",date));
System.out.println(String.format("%td日",date));
System.out.println(String.format("%td",date)+"日");    //此行和上一行输出的 结果一样

System.out.println(String.format("%tH 点",date));
System.out.println(String.format("%tM 分",date));
System.out.println(String.format("%tS 秒",date));

System.out.println(String.format("%tD",date));
System.out.println(String.format("%tc",date));
System.out.println(String.format("%tr",date));
```

更新时间

```
while(true) {
    String h = String.format("%tH 点 : ", new Date());
    String m = String.format("%tM 分 : ", new Date());
    String s = String.format("%tS 秒", new Date());
    System.out.println();
    try {
        Thread.sleep(1000); //秒级
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

2).xx

35 –题呗

**概念

1. 介绍一下 Java 的 8 中基本数据类型。

byte 1B 表示范围-128 - 127
short 2B 表示范围-32768 - 32767
int 4B 表示范围-2147483648 – 2147483647 (默认整数类型)
long 8B 表示范围-9223372036854775808 - 9223372036854775807
float 单精度浮点数 4B 第一个是符号位，中间 8 个是指数位，其余为尾数位字面值后加 F 或 f 表示
double 双精度浮点数 8B 第一个是符号位，中间 11 个是指数位，其余为尾数位 是默认浮点类型
char 字符 2B 数据范围:0-65535
8:boolean 字面值 :true 、false

2. 说说 JDK、JRE、JVM 的区别？

JDK:JVM + 解释器 + 编译器 + 工具 + 类库

JRE:JVM + 解释器

JVM:Java 虚拟机，屏蔽不同操作系统的的不同，为编程提供标准

3. 简述对象的创建过程？

先分配空间-构造父类对象-初始化本类属性-调用本类构造方法

4. 描述以下 4 个访问修饰符的区别

private、protected、default、public

private 私有的，只能在本类内部访问，不能被继承;protected 保护的，能在本类和同包的其他类及子类访问，可以继承;default 默认的，本类及同包其他类可以访问，同包子类可以继承;public 公开的，可以继承

5. 方法覆盖和方法重载的区别？

方法覆盖:用子类方法替换父类继承给它的方法，要求方法名、参数表和返回值类型要相同，访问修饰符要相同或更宽。

方法重载:本类中方法，使方法功能更强，要求方法名相同，参数表不同，对返回值不做要求。

6. 方法覆盖的语法要求？

要求:方法名、参数表和返回值类型要相同，访问修饰符要相同或修饰范围更广。

7. 简述以下关键词的含义：

break、continue、instanceof、synchronized、static、final、abstract

break: 结束当前循环

continue: 跳过本次循环

instanceof: 判断对象是否是指定的对象类型

synchronized: 为线程加锁，使当前代码需要同步执行

static: 静态属性，在类加载时执行，而且只执行一次

final : 声明属性为常量，一旦声明赋值，则不能修改

abstract: 生猛类或方法为抽象

8. 接口和抽象类的区别？

接口:一个类可以实现多个接口，接口中所有方法都是抽象的，接口中只提供方法声明不实现，需要实现类实现所有方法，接口只能定义公开静态常量，接口可以继承接口。

抽象类:一个类只能继承一个抽象类，抽象类中有抽象方法也可以与有普通方法，可以实现部分方法，抽象类中基本数据类型没有要求。

9. Integer 和 int 有什么区别？

Integer 是 int 的包装类，需要实例初始化后才能使用，默认值是 null。

int 是基本数据类型，默认值是 0。

10. String 和 StringBuilder 的区别？

String:本身不可改变字符序列，只能赋值一次，每次改变都会创建一个新的对象，原有对象引用新对象。

StringBuilder:可变字符序列，每次操作都是对原有对象操作，不会生成新的对象，其所占得空间随内容增加而扩充。

11. 简单描述串池的工作机制？好处是什么？

以字面值形式的字符串默认进串池查找，避免了频繁创建、销毁对象而影响系统性能。

12. 说说 final finally finalize 的区别

final :修饰符，修饰类时不能被继承，修饰方法时不能被重载，修饰变量时不可改变。

finally:关键字，处理异常时放在 try catch 代码块后面，一定被执行，一般存放释放资源代码。

finalize:方法名，定义在 Object 类中的方法，在对象被垃圾回收的时候，由垃圾收集器自动调用。

13. 简单说说内存泄露和内存溢出的区别？

内存泄漏是程序在申请内存后，无法释放已申请的内存空间，导致了系统无法将空间再次分配给需要的程序。

内存溢出是程序申请内存时，系统不能满足其内存规格，产生溢出。

14. ArrayList 和 LinkedList 的区别？

ArrayList: 数组实现，查询快，增删慢。

LinkedList: 链表实现，查询慢，增删快。

15. HashSet 和 TreeSet 的区别？HashSet 要想实现，对象的属性值一样则认为是重复对象，只保留一份该如何实现

HashSet:不能保证元素的排列顺序，顺序可能出现变化，最多存放一个 null

TreeSet:可以根据元素内容进行自动的升序排列自定义类型为保证元素内容不重复,必须实现 Comparable 接口,提供 compareTo 方法实现；

如果保证元素内容不重复,要重写 hashCode 和 equals，hashCode 重写要求: 内容相同的对象必须返回相同哈希码，内容不同的对象极可能返回不同的哈希码

16. 说说什么叫类加载？

JVM 运行时需要引用某个类时会先去加载这个类，读取这个类 class 文件到内存中。

17. 说说"NullPionterException"是已检查异常还是未检查异常，导致该异常的原因是什么，如何避免该异常？

是未检查异常，导致该异常的原因是空指针异常，在使用的变量前赋值，或者判断是否为空

**编程

1.StringBuffer

```
StringBuffer c=new StringBuffer("zbcdefghijklmn");
System.out.println(c);
c.setCharAt(3,'D');          //修改字符串特定位置的字符;
System.out.println(c.reverse());       //翻转字符串的字符顺序;
System.out.println(c.delete(3,7));     // 删除索引为 3 到 7 之间的字符
```

2.自定义单链表：

```
class Node{
    Object data; //数据区
    Node next; //引用区 指向下一个节点
    public Node(Object data, Node next) {
        super();
        this.data = data;
        this.next = next;
    }
    public String toString() {return "Node [data=" + data + ", next=" + next + "]"; }
}
class MyLinkedList{
    private Node head;  //Node类型的属性保存链表的头结点
    private int count; //记录节点数量
    public void add(Object data){
        Node newNode = new Node(data,null); //1 新建节点
        if(head == null){ //如果是第1次添加,特殊处理
            head = newNode;
        }else{
            //2 找到尾部节点
            Node p = head;  //如果p指向的当前节点后续还有节点
            while(p.next != null){
                p = p.next; //则将p指向下一个节点
            }
                p.next = newNode; //3 尾部节点指向新节点
        }
        count++;
    }
    private Node node(int index){ //查询指定下标节点
```

```
        Node p = head;
        int count = 0;
        while(count < index){
            p = p.next;
            count++;
        }
        return p;
    }
    public void add(int index,Object data){ //添加节点
        Node newNode = new Node(data,null); //1 新建节点newNode
        if(index == 0){      //对于插入头部,进行特殊处理
            newNode.next = head;
            head = newNode;
        }else{
            //2 查找到index-1处节点 p
            Node p = node(index - 1);
            // 查找到index处节点 q
            Node q = p.next;
            //3 p指向newNode,newNode指向q
            p.next = newNode;
            newNode.next = q;
        }
        count++;
    }
    public void remove(int index){  //删除指定下标节点
        if(count == 0){
            return ;
        }
        if(index == 0){
            Node p = head;
            head = p.next;
            p.next = null;
        }else{
            //获取index-1处节点p
            //获取index处节点q
            //获取index+1处节点k
            Node p = node(index-1);
            Node q = p.next;
            Node k = q.next;
            //p指向k,q断开和k的联系
            p.next = k;
            q.next = null;
        }
        count--;
    }
    public void set(int index, Object data){  //修改指定节点数据
        node(index).data = data;
    }
    public Object get(int index){ //获取指定下标节点的元素
```

```
        return node(index).data;
    }
    public int size(){ //返回长度
        return count;
    }
    public int indexOf(Object data){  //返回指定数据的节点下标
        if(count == 0){
            return -1;
        }
        Node p = head;
        int count = 0;
        while(count < this.count){
            if(p.data.equals(data)){
                return count;
            }
            p = p.next;
            count ++;
        }
        return -1;
    }
    public boolean contains(Object data){ //判断是否包含该数据的节点
        return indexOf(data) >= 0;
    }
```

3.利用 lambda 表达式遍历数组

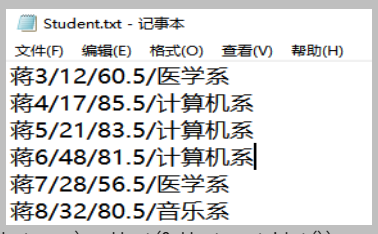
```
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Huxz");
        list.add("Xusy");
        list.add("Yangdd");
        list.forEach(new Consumer<String>() {  //实现接口
            public void accept(String s) {
                System.out.println(s);
            }
        });
        list.forEach(s->System.out.println(s));
        list.forEach(System.out::println);
    }
```

4.lambda 表达式

```
    public class Test {
        public static void main(String[] args) {
            myclass m = new myclass();
            TA ta = new TA() {
                public void mm(myclass mc) {
                    m.method();
                }
            };
            ta.mm(m);//调用mm  1
            TA ta2 = (myclass mc) -> m.method();
            ta2.mm(m); //调用mm  2
```

```
        }
    }
    interface TA {
        public abstract void mm(myclass m);
    }
    class myclass {
        public void method() {
            System.out.println("m");
        }
        public void method(String s) {
            System.out.println("mm");
        }
    }
```

5.(Lambda、Stream)有一个文件 Student;先转化为 List<Student>,再操作




```
    public class Stream_Student {
    public static void main(String[] args) throws IOException {
//  1.读取Student.txt 将Student对象放入List中
    //文件中不能有空行
    List<Student> list = Files.lines(Paths.get("D:\\Student.txt"), Charset.forName("GBK")).map(Student::new).collect(Collectors.toList());
//  2.统计所有学生中大于18岁的学生人数
    System.out.println("-------2.统计所有学生中大于18岁的学生人数---------");
    long countOfAgeUp18 = list.stream().filter((s) -> s.getAge() >= 18).count();
    System.out.println(countOfAgeUp18);
//  3.打印所有学生中成绩最高的三个学生的名字
    System.out.println("-------3.打印所有学生中成绩最高的三个学生的名字---------");
    list.stream().sorted(Comparator.comparingDouble(Student::getScore).reversed()).limit(3).forEach((s) ->
System.out.println(s.getName()));
//  4.把所有学生的名字拼成字符串,用逗号隔开 打印出来
    System.out.println("-------4.把所有学生的名字拼成字符串,用逗号隔开 打印出来---------");
    String nameString = list.stream().map(Student::getName).collect(Collectors.joining(","));
    System.out.println(nameString);
//  5.打印所有的院系名称,要求无重复
    System.out.println("-------5.打印所有的院系名称,要求无重复---------");
    //list.stream().collect(Collectors.groupingBy(Student::getDept)).forEach((k, v) -> System.out.println(k));
    list.stream().map(Student::getDept).distinct().forEach(System.out::println);
//  6.把每个院系的学生名字,拼成字符串,用逗号隔开, 打印出来
    System.out.println("-------6.把每个院系的学生名字,拼成字符串,用逗号隔开, 打印出来---------");
    list.stream().collect(Collectors.groupingBy(Student::getDept)).forEach((k, v) ->
    System.out.println(v.stream().map(Student::getName).collect(Collectors.joining(","))));
//  7.打印每个院系年龄最小的学生的名字
    System.out.println("-------7.打印每个院系年龄最小的学生的名字---------");
    list.stream().collect(Collectors.groupingBy(Student::getDept)).forEach((k, v) ->
    System.out.println(k + ":" + v.stream().sorted(Comparator.comparingInt(Student::getAge)).limit(1).
            collect(Collectors.toList()).get(0).getName()) );
//  8.打印每个院系的平均成绩
    System.out.println("-------8.打印每个院系的平均成绩---------");
    list.stream().collect(Collectors.groupingBy(Student::getDept)).forEach((k, v) ->
            System.out.println(k + ":" +
v.stream().collect(Collectors.averagingDouble(Student::getScore)).doubleValue()));
```

```
//   9.打印平均成绩最高的院系的名称
     System.out.println("-------9.打印平均成绩最高的院系的名称--------");
     Collection<List<Student>>collection = list.stream().collect(Collectors.groupingBy(Student::getDept)).values();
     Optional<Dept> optional = collection.stream().map(Dept::new).max(Comparator.comparingDouble(Dept::getAvgScore));
     System.out.println(optional.get().getName() + " " + optional.get().getAvgScore());
//   10.统计每个院系考试及格和考试不及格的学生人数
     System.out.println("-------10.统计每个院系考试及格和考试不及格的学生人数--------");
          list.stream().collect(Collectors.groupingBy(Student::getDept)).forEach((k, v) -> {
               System.out.print(k + ": ");
               Map<Boolean, Long> collect = v.stream().collect(Collectors.partitioningBy((s) ->
                    s.getScore() >= 60,Collectors.counting()));
               System.out.print("及格：" + collect.get(true) + "    ");
               System.out.print("不及格：" + collect.get(false));
               System.out.println();
          });
     }
}
class Dept {
     private String name;
     private double avgScore;
     public String getName() {return name;}
     public void setName(String name) { this.name = name;}
     public double getAvgScore() { return avgScore;}
     public void setAvgScore(double avgScore) {this.avgScore = avgScore;}
     public Dept(List<Student> list) {
          this.name = list.get(0).getDept();
          this.avgScore = list.stream().collect(Collectors.averagingDouble(Student::getScore));
     }
}
class Student {
     private String name;
     private int age;
     private double score;
     private String dept;  //院系
     public String getName() {return name;}
     public void setName(String name) { this.name = name;}
     public int getAge() {return age;}
     public void setAge(int age) { this.age = age;}
     public double getScore() {return score;}
     public void setScore(double score) {this.score = score;    }
     public String getDept() {return dept;}
     public void setDept(String dept) { this.dept = dept;}
     public Student(String s) {  //构造方法，参数为读取的文本中的一行，不能有空行，不然会有空指针异常
          String[] tmp = s.split("/");
          this.name = tmp[0];
          this.age = Integer.parseInt(tmp[1]);
          this.score = Double.parseDouble(tmp[2]);
          this.dept = tmp[3];
     }
```

```
        public String toString() {return "Student [name=" + name + ", age=" + age + ", score=" + score + ", dept=" + dept +
    "]";}
    }
```

6.

7.

Part2 - DB