常用API

Object类

1.重写obj类的equals方法

obj的equals方法默认比较的是两个对象的地址值，没有意义，所以我们要重写equals方法，来比较两个对象的属性。

问题：

隐含着一个多态

多态的弊端：无法使用子类特有的内容（属性和方法）

解决方法：向下转型（强转）把obj转换为person类型

```
@Override
    public boolean equals(Object obj){
        //增加一个判断，如果为null，直接返回false
        if(obj==null){
            return false;
        }
        //增加判断，this
        if(obj==this){
            return true;
        }
        //使用向下转型，将obj转化为person
        //增加一个判断，防止类型转换异常
        if(obj instanceof person){
            person p=(person)obj;
            boolean b=this.name==p.name&&this.age==p.age;
            return b;
        }
        else{
            return false;
        }
```

```
/*Object类得equals方法：对两个对象进行比较，防止空指针异常*/
        public static boolean equals(Object a,Object b){
            return (a==b)||(a!=null&&a.equals(b));
        }
```

Date类

概述：Java.util.Date包中，表示日期和时间的类。

把日期转换为毫秒：当前的日期 （2021年1月31日09点32分） 时间原点（1970年1月1日00：00：00），就是计算当前日期到时间原点共经历了多少毫秒。

```
import java.util.Date;

public class date {
    public static void main(String[] args) {
```

```
        //long类型的值
        //零点时间为英国格林威治时间，中国属于东八区
        System.out.println(System.currentTimeMillis());//获取从零点开始经历了多少毫秒
        //无参构造
        demo01();
        /*Date的带参构造
        传递毫秒值（long类型），转换为时间*/
        demo02();

        demo03();
    }
    //带参构造
    private static void demo02(){
        Date date=new Date(342323L);
        System.out.println(date);
    }
    //无参构造
    private static void demo01(){
        Date date=new Date();
        System.out.println(date);//应该打印地址值，打印了时间，说明重写了
    }

    private static void demo03(){
        Date date =new Date();
        long time =date.getTime();//将日期转换为毫秒值
        System.out.println(time);
    }


}
```

java.text.DateFormate : 是日期/时间格式化子类的抽象类

作用：格式化（日期->文本），解析（文本->时间）

成员方法：

String format(Date date) 按照指定的模式，把Date日期，格式化为符合模式的字符串

Date parse(String source) 把符合模式的字符串，解析为Date日期

DateFormate类是一个抽象类，无法直接创建对象使用，可以使用DaFormate类的子类（SimpleDateFormat）

```
package Demo01.Demo002;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/*
构造方法：
    SimpleDateFormate(String pattern)
    用给定的模式和默认语言环境的日期格式符号构造
    参数：String pattern:传递指定的模式
    模式：区分大小写的
    注意：模式中的字母不能更改，连接的符号可以改变
    */
```

```
public class dateformate {
    public static void main(String[] args) throws ParseException {
        /*使用DateFormat类中的format，将日期格式转化为文本
        使用步骤：
        1.创建SimpleDateFormat对象，构造方法中传递指定的模式
        2.调用SimpleDateFormat对象中的format方法,把Date日期格式化为符合模式的字符串（文本）*/
        SimpleDateFormat sdf=new SimpleDateFormat("yyy年MM月dd日 HH时mm分ss秒");
        Date date=new Date();
        String d=sdf.format(date);
        System.out.println(d);
        /*
        使用DateFormat类中的parse方法，把文本解析为日期
        使用步骤：
            1.创建SimpleDateFormat对象，构造方法中传递指定的模式
            2.调用SimpleDateFormat对象中的parse，把符合构造方法中模式的字符串，解析为Date日期
            如果字符串和构造方法中的模式不一样，那么程序就会抛出异常
        */
        Date date1=sdf.parse("2021年01月31日 21时49分48秒");
        //Alt+回车
        System.out.println(date1);
    }
}
```

Calendar类

```
package Demo01.Demo003;

import java.util.Calendar;
import java.util.Date;

/*
    java.util.Calendar类：日历类
    Calendar类是一个抽象类，里面提供了很多操控日历字段的方法
    Calendar类无法直接创建对象使用，里面有一个静态方法叫getInstance()，该方法返回Calendar类的子类对象

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

        Calendar c=Calendar.getInstance();
        //打印结果非地址，表明重写了toString方法
        System.out.println(c);

        /*
        Calendar类的常用成员方法：
          public int get(int field):返回给定日历字段的值
          public void set(int field,int value):将给定的日历字段设置为给定值
          public abstract void add(int field,int amount):根据日历的规则，为给定的日历字段添加或减去指定的时间量
          public Date getTime():返回一个表示此Calendar时间值（从历元到现在的毫秒偏移量）的Date对象
        成员方法的参数：
          int field：日历类中的字段，可以通过Calendar类的静态成员变量获取
```

```
        */
        demo01();
        demo02();
        demo03();
        demo04();
    }

    public static void demo01(){
        //使用getInstance()方法获取Calendar对象
        Calendar c=Calendar.getInstance();
        int year=c.get(Calendar.YEAR);
        System.out.println(year);
        int month=c.get(Calendar.MONTH);
        System.out.println(month+1);//西方月0-11
        int date=c.get(Calendar.DAY_OF_MONTH);
        System.out.println(date);
    }

    public static void demo02(){
        Calendar c=Calendar.getInstance();
        //单个设置
        c.set(Calendar.YEAR,2017);
        c.set(Calendar.MONTH,9);
        c.set(Calendar.DATE,4);
        //同时设置
        c.set(2017,9,4);

    }
    public static void demo03(){
        Calendar c=Calendar.getInstance();
        c.add(Calendar.YEAR,-4);
    }
    public static void demo04(){
        Calendar c=Calendar.getInstance();
        Date d=c.getTime();
        System.out.println(d);
    }
}
```

System类

java.long.System包中。不用导包。

```
package Demo01.Demo004;

import java.util.Arrays;

public class sys {
    public static void main(String[] args) {
        //程序执行前，获取一次毫秒值
        long c1 = System.currentTimeMillis();
        for (int i = 0; i <9999 ; i++) {
            System.out.println(i);
        }
        long c2=System.currentTimeMillis();
        System.out.println("耗时"+(c2-c1));
```

```
        demo02();
    }
    public static void demo02(){
        int[]a={1,2,3,4,5};
        int[]b={6,7,8,9,0};
        System.out.println(Arrays.toString(b));
        //使用arraycopy
        System.arraycopy(a,0,b,0,3);
        System.out.println(Arrays.toString(b));
    }
}
```

StringBuilder类

也被称为字符串缓冲区。可以提高效率。（看成长度可以变化的字符串）

底层也是一个数组，但是没有被final修饰，可以改变长度。

byte[] value=new byte[16];

在内存中始终是一个数组，如果超出容量，自动扩大一倍。

String类：

字符串是常量，他们的值在创建之后不能被改变。

String字符串的底层是一个被final修饰的数组，不能改变，是一个常量。

private final byte[] value;

进行字符串的相加，内存中就会有多个字符串，占用空间多，效率低下。

```
package Demo01.strbuilder;
/*
StringBuilder和String可以相互转换
String-->StringBuilder可以使用StringBuilder的构造器
StringBuilder-->String可以食用toString方法
*/
public class str {
    public static void main(String[] args) {
        //空参数构造方法
        StringBuilder stringBuilder = new StringBuilder();
        //带字符串的构造方法
        StringBuilder bu2=new StringBuilder("abc");
        /*
        常用方法：
        */
        //使用append方法往StringBuilder中添加数据
        //append方法的返回值是this，使用append方法无需接收返回值
        bu2=stringBuilder.append("abc");
        /*
        链式编程
        */
        System.out.println("abc".toUpperCase().toLowerCase());
        String str="hello";
        StringBuilder stringBuilder1 = new StringBuilder();
        String a=stringBuilder.toString();
    }
```

```
}
```

包装类

基本数据类型很好用，但没有对应的方法来操作这些基本数据类型，可以使用一个类，把基本数据类型的数据装起来。

```
package Demo01.Demo005;
/*
装箱：把基本类型的数据包装到包装类中
      构造方法：
           Integer(int value)
           Integer(String s)
               传递的字符串必须是基本类型的字符串，否则会抛出异常
      静态方法：
           static Integer valueOf(int i)
           static Integer valueOf(String s)
拆箱：在包装类中取出数据

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

        //方法上有横线，说明过时了
        Integer i=new Integer(1);
        System.out.println(i);//重写了toString方法
    }
}
```

自动装箱与拆箱

```
package Demo01.Demo005;

import java.util.ArrayList;
/*
基本类型与字符串之间的转换
基本类型--》字符串
      1.解百纳类型的值+"" 最简单的方法
      2.包装类的toString方法
      3.String中有个valueOf方法
字符串--》基本类型
      使用包装类的静态方法parse
*/
public class demo1 {
    public static void main(String[] args) {
        /*
        自动装箱：直接把int类型的整数赋值为包装类
        */
        Integer i=1;
        /*
        自动拆箱，原来的包装类，无法直接参与运算，可以自动转换为基本数据类型，再进行运算
        自动装箱
        */
        i=i+2;
```

```
        ArrayList<Integer>list=new ArrayList<>();
        list.add(1);
        String a=1+"";//最简单的方法
        String s2=Integer.toString(6);
        String s3=String.valueOf(9);
        int i1=Integer.parseInt("67231");

    }
}
```

容器类

Collection

常用功能:

```
package Demo01.Demo006;

import java.util.ArrayList;
import java.util.Collection;

/*
java.util.Collection接口
    所有单列集合的最顶层的接口，里面定义了所有单列集合共性的方法
    任意的单列集合都可以使用Collection接口中的方法
    共性的方法:
          int size();
          boolean isEmpty();
          boolean contains(Object o);
          boolean add(E e);
          boolean remove(Object o);
          void clear();
*/
public class coll {
    public static void main(String[] args) {

        //创建集合对象可以使用多态
        Collection<String>coll=new ArrayList<>();
        System.out.println(coll);
        coll.add("张三");
        coll.add("李四");
        coll.add("王五");
        coll.remove("赵六");//没有的会返回false
        coll.contains("天七");
        coll.isEmpty();
        int i=coll.size();
        Object[] a=coll.toArray();
        for (int j = 0; j <a.length ; j++) {
            System.out.println(a[j]);
        }
    }
}
```

迭代器

Iterator

```
package Demo01.Demo007;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

/*
    java.util.Iterator
    有两个常用的方法
        boolean hasNext(); 如果仍有元素可以迭代，则返回true
        E next(); 返回迭代的下一个元素
     Iterator迭代器，是一个接口，我们无法直接使用，需要使用接口的实现类
     Collection接口中有一个方法，叫iterator(),这个放法返回的就是迭代器的实现类对象
     迭代器的使用步骤：
           1.使用集合中的方法iterator(),获取迭代器的实现类对象，使用Iterator
           2.使用Iterator接口中的方法hasNext判断还有没有下一个元素
           3.使用Iterator接口中的next取出集合中的下一个元素
*/
public class demoiterator {
    public static void main(String[] args) {

        Collection<String>coll=new ArrayList<>();
        coll.add("koi");
        coll.add("misaka");
        coll.add("tamako");
        /*
        使用集合中的方法iterator()获取迭代器的实现类对象，使用Iterator接口接受
        注意：
            Iterator<E>接口也是有泛型的，迭代器的泛型跟着集合走，集合是什么泛型，迭代器就是
什么泛型
        */
        Iterator<String>it=coll.iterator();
        while(it.hasNext()){
            String e=it.next();
            System.out.println(e);
        }
        boolean b=it.hasNext();
        System.out.println(b);//输出有无下一个值
        String s=it.next();//取出元素
        /*
        增强for循环，底层是迭代器，使用for循环结构
        */
        int[]arr={1,2,3,4,5};
        for(int i:arr){
            System.out.println(arr[i]);
        }
    }
}
```

泛型

概念：

一种未知的数据类型，当我们不知道用什么数据类型时，用泛型。

泛型也可以看成一个变量，用来接受数据类型。

E e：Element元素

T t：Type类型

创建集合对象的时候，就会确定泛型的数据类型，将数据类型作为参数传递，赋值给泛型E

使用泛型的好处：

```
package Demo01.Demo008;

import java.util.ArrayList;
import java.util.Iterator;

/*
创建集合对象，不使用泛型
好处：
    集合不使用泛型，默认的类型就是Object类型，可以储存任意类型的数据
弊端：
     不安全，会引发异常

创建集合对象，使用泛型
好处：
     1.避免类型转换引起的麻烦，存储的是什么类型，取出就是什么类型
     2.把运行期异常（代码运行之后抛出的异常），提升到了编译期（写代码的时候抛出异常）
弊端：
      泛型是什么类型，只能存储什么类型的数据

*/
public class fan {
    public static void main(String[] args) {
        ArrayList list=new ArrayList();
        list.add("koi");
        list.add(9);
        Iterator it=list.iterator();
        while(it.hasNext()){
            Object next = it.next();
            System.out.println(next);
            //String s=(String)next;
        }
    }
}
```

定义含泛型的类和方法：

```
package Demo01.Demo008;
/*
定义一个含有泛型的类，模拟ArrayList集合
泛型是一个未知的数据类型，当我们不知道用什么数据类型时，可以使用泛型
泛型可以接受任意的数据类型，可以使用Integer，String
创建队象的时候确定泛型的数据类型
```

```
定义含有泛型的方法：泛型定义在方法的修饰符和返回值类型之间
*/
public class crefan<E> {
    private E name;

    public E getName() {
        return name;
    }

    public crefan setName(E name) {
        this.name = name;
        return this;
    }

    //定义含泛型的方法
    public <M> void method(M m){
        System.out.println(m);
    }

    public static <S> void method01(S s){
        System.out.println(s);
    }
}
```

定义一个含有泛型的接口：

```
package Demo01.Demo008;

public interface genint<I> {
    public abstract void method(I i);
}
```

```
package Demo01.Demo008;
/*
含泛型的接口第一种使用方式
Scanner类实现了Iterator接口，并指定接口的泛型为String，所以重写的next方法泛型默认就是String
*/
public class genintimp implements genint<String>{
    @Override
    public void method(String s) {
        System.out.println(s);
    }
}
```

```
package Demo01.Demo008;
/*
含泛型的接口的第二种使用方法：
接口用什么泛型，实现类就用什么泛型
相当于定义了一个含有泛型的类，创建对象的时候确定泛型的类型
*/
public class genintimp2<I> implements genint<I>{
```

```
    @Override
    public void method(I i) {

        System.out.println(i);
    }
}
```

泛型的通配符：

```
package Demo01.Demo008.d001;

import java.util.ArrayList;
import java.util.Iterator;

/*
泛型的通配符：
      ？代表任意的数据类型
使用方法：
      不能创建对象使用
      只能作为方法的参数使用
*/
public class normal {
    public static void main(String[] args) {
        ArrayList<Integer>list01=new ArrayList<>();
        list01.add(12);
        list01.add(123);
        ArrayList<String>list02=new ArrayList<>();
        list02.add("koi");
        list02.add("misaka");

    }
    /*
    定义一个方法，能遍历所有类型的ArrayList集合
    这时候我们不知道ArrayList集合能使用什么数据类型，可以使用泛型的通配符？来接收数据类型
    注意：泛型没有继承概念
    */
    public static void printarray(ArrayList<?> list){

        //使用迭代器遍历集合
        Iterator<?>iterator=list.iterator();
        while(iterator.hasNext()){
            //iterator.next方法，取出元素是Object，可以接受任意的数据类型
            Object o=iterator.next();
            System.out.println(o);
        }
    }
}
```

泛型的限定：

泛型的上限限定：？ extends E 代表使用的泛型只能是E类型的子类/本身
泛型的下线限定：？ super E 代表使用的泛型只能是E类型的父类/本身

斗地主案例

```
package Demo01.Demopractice;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;

public class DemoMain {
    public static void main(String[] args) {
        //存储54张牌
        ArrayList<String>poker=new ArrayList<>();
        //存花色和存序号
        String[]colors={"♠","♥","♣","♦"};
        String[]numbers={"2","A","K","Q","J","10","9","8","7","6","5","4","3"};
        //存大王和小王
        poker.add("大王");
        poker.add("小王");
        for (String num:numbers){
            for(String clo:colors){
                poker.add(clo+num);
            }
        }
        //2.洗牌
        Collections.shuffle(poker);
        //3.发牌
        ArrayList<String>player1=new ArrayList<>();
        ArrayList<String>player2=new ArrayList<>();
        ArrayList<String>player3=new ArrayList<>();
        ArrayList<String>owner=new ArrayList<>();
        for(int i=0;i<poker.size();i++){
            String p=poker.get(i);
            if(i>=51){
                owner.add(p);
            }else if(i%3==0){
                player1.add(p);
            }else if(i%3==1){
                player2.add(p);
            }else if(i%3==2){
                player3.add(p);
            }
        }
        System.out.println(player1);
        System.out.println(player2);
        System.out.println(player3);
    }
}
```

数据结构（部分）

数组：

查询快：数组的地址是连续的，可以通过数组的首地址找到数组，可以通过索引快速找到某一个元素。

增删慢：数组的长度是固定的，如果要增删，就必须创建一个新数组，把原数组的元素复制过来。

在堆内存中，频繁的创建数组，复制数组中的元素，销毁数组，效率低下。

链表：

查询慢：链表中的地址不连续。

增删快：链表结构，增删对整体影响不大。

红黑树：

查询速度快。查询的最大次数不能超过最小次数的二倍。

约束：

1.结点只能是红色或黑色

2.根节点为黑色

3.叶子结点（空结点）为黑色

4.任何红色的结点的子节点都是黑色的

5.任何一个结点到其每一个叶子结点的所有路径上黑色节点数目相同

List集合

List集合简介&常用方法

```
package Demo01.Demo009;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
/*
java.util.List接口 extends Collection接口
List接口的特点：
    1.有序的集合，存储元素和取出元素的顺序是一致的
    2.存在索引，所以包含了一些带索引的方法
    3.允许存储重复的元素
*/
public class DemoMain {
    public static void main(String[] args) {
        List<String> list=new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("d");
        System.out.println(list);
        list.add(3,"koi");
        list.remove(3);//返回值是被移除的元素
        list.set(3,"D");//返回值是被替换的元素
        list.get(0);//获取指定索引处的元素
        //list有三种遍历方法
        //普通的for循环遍历
```

```
        for(int i=0;i<list.size();i++){
            String s=list.get(i);
            System.out.println(s);
        }
        //使用迭代器来遍历
        Iterator<String> it= list.iterator();
        while (it.hasNext()){
            String s=it.next();
            System.out.println(s);
        }
        //增强for循环
        for(String s:list){
            System.out.println(s);
        }
    }
}
```

List集合的实现类

```
package Demo01.Demo009;

import java.util.LinkedList;

/*
java.util.Linklist 集合 implements List接口
LinkedList集合的特点：
1.底层是一个链表，查询慢，增删快
2.里面包含了大量操作首尾元素的方法
注意：使用LinkedList集合特有的方法，不能使用多态
 */
public class Demolink {
    public static void main(String[] args) {
        show01();
        show02();
        show03();
    }

    private static void show03() {
        LinkedList<String> linked = new LinkedList<>();
        linked.add("a");
        linked.add("b");
        linked.add("c");
        linked.removeFirst();//返回被移除的元素
        linked.removeLast();
        linked.pop();//和removeFirst();方法一致
    }

    //获取元素
    private static void show02() {
        LinkedList<String> linked = new LinkedList<>();
        linked.add("a");
        linked.add("b");
        linked.add("c");
        String first=linked.getFirst();
        System.out.println(first);
    }
```

```
    //添加
    public static void show01(){
        LinkedList<String> linked = new LinkedList<>();
        linked.add("a");
        linked.add("b");
        linked.add("c");
        System.out.println(linked);
        linked.addFirst("koi");
        linked.push("suki");//等效于assFirst
        linked.addLast("misaka");
    }
}
```

Vector集合

已经被淘汰了，比较慢。

Set接口

哈希值

```
package Demo01.Demo10;
/*
哈希值：是一个十进制的整数，由系统随机给出（就是对象的地址值，是一个逻辑地址，是模拟出来的地址，
不是数据储存的实际地址）
在Object类中有一个方法，可以获取对象的哈希值
int hashCode()
    返回该对象的哈希值。
hashCode方法源码
    public native int hashCode();
    native代表该方法调用的是本地操作系统的方法
*/
public class Hashstu {
    public static void main(String[] args) {

        //person继承了Object类
        person person = new person();
        int i=person.hashCode();
        System.out.println(i);

        /*
        toString方法的源码：
              public String toString() {
                   return getClass().getName() + "@" +
Integer.toHexString(hashCode());
              }
         重写的toString方法返回的也是哈希值（十六进制）
        */
        System.out.println(person);
        /*
        String类的哈希值
              String类重写了Object类的hashCode方法
        */
        String s1=new String("abc");
        String s2=new String("abc");
        System.out.println(s1.hashCode());
        System.out.println(s2.hashCode());
```

```
        System.out.println("重地".hashCode());//巧合，虽然字符串不一样，但哈希值一样
        System.out.println("通话".hashCode());//1179395
    }
}
```

存储结构

HashSet集合存储数据的结构（哈希表）

jdk1.8版本之前：哈希表=数组+链表

jdk1.8版本之后：哈希表=数组+红黑树

哈希表的特点：查询快

数据结构：把元素进行了分组（相同哈希值的元素为一组），链表/红黑树把相同哈希值的元素链接到一起

哈希冲突：两个元素不同，哈希值冲突。

如果链表的长度超过八位，就会把链表转换为红黑树。

原理

Set集合在调用add方法的时候会调用hashCode方法和equals方法，判断元素是否重复。

当发现哈希值相同的时候，会调用equals方法。

重写equals和hashCode方法

```
package Demo01.Demo10;

import java.util.Objects;

public class person extends Object{

    private String name;
    private int age;

    public person() {
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        person person = (person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

    @Override
    public String toString() {
        return "person{"+"name="+name+",age="+age+"}";
    }
```

```
    public person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    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;
    }
}
```

LinkedHashSet

```
package Demo01.Demo10;

import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.LinkedList;

/*
LinkedHashSet集合的特点：
      底层是一个哈希表（数组+链表/红黑树）+链表：多了一条链表（记录元素的存储顺序），保证元素有序

*/
public class linkedhash {
    public static void main(String[] args) {
        HashSet<String>set=new HashSet<>();
        set.add("abc");
        set.add("www");
        set.add("itcast");
        System.out.println(set);
        LinkedHashSet<String>linked=new LinkedHashSet<>();
        linked.add("www");
        linked.add("abc");
        linked.add("itcast");
        System.out.println(linked);
    }
}
```

可变参数

```
package Demo01.Demo10;
/*
可变参数：JDK1.5之后出现的新特性
使用前提：
        当方法的参数列表数据类型已经确定，但是参数的个数不确定，就可以使用可变参数
使用格式：定义方法时使用
        修饰符 返回值类型 方法名（数据类型...变量名）{}
可变参数的原理：
        可变参数底层就是一个数组，根据参数传递个数的不同，会创建不同长度的数组，来存储这些数
        传递的参数个数，可以是多个
注意事项：
    1.一个方法的参数列表只能有一个可变参数
    2.如果参数有多个，可变参数必须在末尾
*/
public class couldmodify {
    public static void main(String[] args) {

        int i=add(110,10,50);
        System.out.println(i);
    }
    /*
    定义计算（0-n）整数和的方法
    已知数据类型，参数个数不知道
    */
    //就是一个数组，传进来几个参数就会创建长度为几的数组
    public static int  add(int ...arr){
        int sum=0;
        for(int i:arr){
            sum+=i;
        }
        return sum;
    }
}
```

Collections集合工具类

```
package Demo01.Demo10;

public class student implements Comparable<student>{
    private String name;
    private int age;

    public student() {
    }


    public student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    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;
    }

    @Override
    public String toString() {
        return "student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    @Override
    public int compareTo(student o) {
        //自定义比较的规则
        return this.getAge()-o.getAge();//升序排列
        //return 0;//认为元素都是相同的
    }
}
```

```
package Demo01.Demo10;
/*
注意：被排序的集合里面存储的元素，必须实现Comparable，重写接口中的compareTo定义排序的规则
Comparable升序
this-o;
*/
import java.util.ArrayList;
import java.util.Collections;

public class collections {
    public static void main(String[] args) {
        ArrayList<String>list=new ArrayList<>();
        list.add("a");
        list.add("b");
        Collections.addAll(list,"koi","misaka");
        Collections.shuffle(list);
        ArrayList<Integer>list01=new ArrayList<>();
        list01.add(12);
        list01.add(43);
        Collections.addAll(list01,45,7,8,95,2);
        Collections.sort(list01);
        System.out.println(list01);
        ArrayList<student>p=new ArrayList<>();
        p.add(new student("koi",67));
        p.add(new student("misaka",15));
        p.add(new student("tamako",17));
        Collections.sort(p);
```

```
        System.out.println(p);
    }
}
```

```
package Demo01.Demo10;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/*
Comparator和Comparable的区别
   Comparable：自己（this）和别人（参数）进行比较，自己需要实现Comparable接口，重写比较规则
comparaTo方法
   Comparator：相当于找一个第三方裁判比较两个
*/
public class finalcom {
    public static void main(String[] args) {

        ArrayList<Integer>list=new ArrayList<>();
        list.add(1);
        list.add(2);
        Collections.sort(list, new Comparator<Integer>() {
            //重写比较的规则
            @Override
            public int compare(Integer o1, Integer o2) {
                return o1-02;//前减后，升序
            }
        });

        ArrayList<stu>s=new ArrayList<>();
        s.add(new stu(12,"koi"));
        s.add(new stu(13,"misaka"));
        Collections.sort(s, new Comparator<stu>() {
            @Override
            public int compare(stu o1, stu o2) {
                return o1.getAge()-o2.getAge();
            }
        });
        System.out.println(s);
    }
}
```

Map集合

Map简介&常用子类&常用方法

```
package Demo01.Demo011;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/*
```

```
java.util.Map<k,v>;
Map集合的特点：
   1.Map集合是一个双列集合，一个元素包括两个值（key，value）
   2.Map集合中的元素，key和value的数据类型可以相同，也可以不同
   3.Map集合中的元素，key不允许重复，value是可以重复的
   4.Map集合中的元素，key和value是一一对应的
java.util.HashMap<k,v>集合
HashMap集合特点：
   1.HashMap底层是哈希表：查询的速度特别快
     jdk1.8之前：数组+单向链表
     jdk1.8之后：数组+单向链表/红黑树（链表长度超过8）：提高查询速度
   2.HashMap是一个无序的集合
LinkedHashMap的特点：
   1.LinkedHashMap集合底层是哈希表+链表
   2.LinkedHashMap集合是一个有序的集合，存储取出顺序一致
Map集合的第一种遍历方法：通过找值的方式
Map集合中的方法：
   Set<K>keySet() 返回此映射中包含的键的Set视图
实现步骤：
   1.使用Map集合中的方法keySet()：把Map集合中的key取出来，存储到一个Set集合中
   2.遍历Set集合，获取Map集合中的每一个key
   3.通过Map集合中的方法get（Key），通过key找到value
*/
public class packDemo01 {
    public static void main(String[] args) {
        Map<String,String> map=new HashMap<>();
        String v1=map.put("koi","misaka");
        map.put("misaka","安柏");
        map.put("安柏","可莉");
        map.put("moji","tamako");
        map.remove("安柏");
        String s=map.get("misaka");//根据键去获得值
        boolean koi = map.containsKey("koi");//可以判断键或者值是否存在
        Set<String> set = map.keySet();
        for(String i:set){
            System.out.println(i);
            String s1 = map.get(i);
            System.out.println(s1);
        }
    }
}
```

Entry

Map.entry<K,V>：在Map接口中有一个内部接口Entry

作用：当Map集合一创建，那么就会在Map集合中创建一个Entry对象，用来记录键与值（键值对对对象，键与值得一一映射关系）

Set<Map.Entry<K,V>>entrySet()　把Map集合内部的多个Entry对象取出来，存储到一个Set集合中。

```
package Demo01.Demo011;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/*
```

```
Map集合遍历的第二种方式：使用Entry对象遍历
Map集合中的方法：
    Set<Map.Entry<K,V>>entrySet(),返回包含该映射关系的Set视图
使用步骤：
    1.使用Map集合中的方法entrySet(),把Map集合中的Entry对象取出来，存到一个Set中
    2.遍历Set集合，获取每一个Set对象
    3.使用Entry对象中的方法getKey()和getValue获取键与值
*/
public class PACKdEMO02 {
    public static void main(String[] args) {

        Map<String,String> map=new HashMap<>();
        map.put("tangm","misaka");
        map.put("moji","tamako");
        Set<Map.Entry<String,String>> set=map.entrySet();
        for(Map.Entry<String,String> entry:set){
            String key = entry.getKey();
            String value = entry.getValue();
        }
    }
}
```

HashMap存储自定义类型

```
package Demo01.Demo011;


import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/*
HashMap存储自定义类型
Map集合要保证key的唯一性
    作为key的元素，必须重写hashCode方法和equals方法，以保证key唯一
*/
public class PackDemo003 {
    public static void main(String[] args) {
        show01();
        show02();
    }
    public static void show01(){
        HashMap<String,person>map=new HashMap<>();
        map.put("北京",new person("koi",17));
        map.put("上海",new person("misaka",15));
        //key重复，新的value会替换旧的value
        Set<String> set = map.keySet();
        for(String s:set){
            System.out.println(s);
        }
    }
    public static void show02(){
        HashMap<person,String>map=new HashMap<>();
        map.put(new person("koi",17),"杭州");
        map.put(new person("misaka",15),"学园都市");
        map.put(new person("tamako",17),"tokoy");
```

```
        map.put(new person("kuruko",14),"学园都市");
        Set<Map.Entry<person, String>> entries = map.entrySet();
        for(Map.Entry<person,String> entry:entries){
            person key = entry.getKey();
            String value = entry.getValue();
            System.out.println(key.getName());
            System.out.println(key.getAge());
            System.out.println(value);
        }

    }
}
```

LinkedHashMap&Hashtable

```
package Demo01.Demo011;
/*
Java.util.Hashtable implements Map<K,V>接口
Hashtable：底层也是一个哈希表，是一个线程安全的集合，是单线程的集合，速度慢
HashMap：底层是一个哈希表，是一个线程不安全的集合，是多线程的集合，速度快
HashMap（之前所学的所有的集合）：可以储存null值，null键；
Hashtable集合：不能储存null值，null键；
Hashtable和vector一样被淘汰了
*/
import java.util.HashMap;
import java.util.Hashtable;
import java.util.LinkedHashMap;

/*
LinkedHashMap
    Map接口的哈希表和链表实现，具有可预知的迭代顺序
    底层结构：
       哈希表+链表（记录元素的顺序）
Hashtable
    和Vector集合一样，已经被淘汰了
    但子类Properties
*/
public class PackDemo004 {
    public static void main(String[] args) {
        //有序的集合，但同样不允许重复
        LinkedHashMap<String, String> link = new LinkedHashMap<>();
        Hashtable<String,String>table=new Hashtable<>();
        //table.put(null,null);
    }
}
```

JDK91对于集合添加的优化

```
package Demo01.Demo011;

import java.util.List;
import java.util.Map;
import java.util.Set;

/*
JDK9新特性
```

```
    List接口，Set接口，Map接口里新添加了一个静态的方法of，可以一次性添加多个元素
    使用前提：
        当集合中存储的元素的个数已经确定了，不再改变的时候
注意：
   1.of方法只适用于List接口，Set接口，Map接口，不适用于接接口的实现类
   2.of方法的返回值是一个不能改变的集合，集合不能再使用add，put方法添加元素，会抛出异常
   3.Set接口和Map接口在调用of方法的时候，不能有重复的元素
*/
public class PackDemo005 {
    public static void main(String[] args) {
        List<String>list= List.of("a","b","c");
        Set<Integer>set=Set.of(1,2,3,3);//异常，重复
        Map<Integer,Integer>map=Map.of(1,1,2,3);
    }
}
```

Debug

```
package Demo01.Demo011;
/*
Debug调试程序
     让代码逐行执行，查看代码执行的过程，调试程序中出现的bug
使用方式：
      在行号的右边，查看代码执行的情况
*/
public class DemoDebug {
    public static void main(String[] args) {
        int a=10;
        int b=10;
        int sum=a+b;
        System.out.println(sum);
    }
}
```

斗地主案例plus

```
package Demo01.Demo011;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;

public class Demoprac {
    public static void main(String[] args) {
        //定义一个Map集合，存储牌的索引和组装好的牌
        HashMap<Integer,String>poker=new HashMap<>();
        //创建一个List集合，存储牌的索引
        ArrayList<Integer>pokerIndex=new ArrayList<>();
        //定义两个集合，存储花色和序号
        List<String>colors= List.of("♠","♥","♣","♦");

 List<String>numbers=List.of("2","A","K","Q","J","10","9","8","7","6","5","4","3
");
        int index=0;
        poker.put(index,"大王");
```

```
        pokerIndex.add(index);
        index++;
        poker.put(index,"小王");
        pokerIndex.add(index);
        index++;
        for(String number:numbers){
            for(String color:colors){
                poker.put(index,color+number);
                pokerIndex.add(index);
                index++;
            }
        }
        /*洗牌*/
        Collections.shuffle(pokerIndex);
        //发牌
        ArrayList<Integer> player01 = new ArrayList<>();
        ArrayList<Integer> player02 = new ArrayList<>();
        ArrayList<Integer> player03 = new ArrayList<>();
        ArrayList<Integer> dipai = new ArrayList<>();
        for(int i=0;i<pokerIndex.size();i++){
            Integer in=pokerIndex.get(i);
            if(i>=51){
                dipai.add(in);
            }else if(i%3==0){
                player01.add(in);
            }else if(i%3==1){
                player02.add(in);
            }else if(i%3==2){
                player03.add(in);
            }
        }
        Collections.sort(player01);
        Collections.sort(player02);
        Collections.sort(player03);
        Collections.sort(dipai);
        /*定义一个看牌的代码*/
        look("player01",poker,player01);
        look("player02",poker,player02);
        look("player03",poker,player03);
    }
    public static void look(String
name,HashMap<Integer,String>poker,ArrayList<Integer>list){
        System.out.print(name+": ");
        for(int i:list){
            String value=poker.get(i);
            System.out.print(value+" ");
        }
        System.out.println();
    }
}
```

异常&多线程

异常

异常概念&体系

```
package Demo02.Demo001;


import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/*
java.lang.Throwable:类是Java语言所有错误或者异常的超类
     Exception：编译期异常，进行编译（写代码）java程序出现的异常
          RuntimeException：运行期异常，java程序运行期间出现的问题
          异常是小问题，处理完异常后，程序可以继续执行
     Error：错误
           说明程序得了一个无法修复的问题，必须修改源代码
*/
public class PackDemo01 {
    public static void main(String[] args) /*throws ParseException*/ {
        SimpleDateFormat sdf= new SimpleDateFormat("yy-mm-dd");
        //第一种可以在异常出Alt+Enter，抛出异常
        //第二种，使用try catch

        Date parse = null;
        try {
            parse = sdf.parse("1990-11-25");
        } catch (ParseException e) {
            e.printStackTrace();
        }
        System.out.println(parse);

        //运行期异常的处理

        int[]arr={1,2,3};

        try{
            //可能会出现异常的代码
            System.out.println(arr[3]);
        }catch(Exception e){
            //异常的处理逻辑
            System.out.println(e);
        }


        //占用内存过大，必须修改代码
        //int[]arr=new int[1024*1024*1024];
    }
}
```

异常的产生过程

```
package Demo02.Demo001;
/*
异常的产生过程分析
*/
public class PackDemo02 {
    public static void main(String[] args) {
         int[]arr={1,2,3};
        int element = getElement(arr, 0);
        System.out.println(element);
        /*
        JVM会根据检测出的异常做两件事：
          1.JVM会根据异常产生的原因创建一个异常对象，这个对象包含了异常产生的（内容，原因，位置）
          2.若在相应方法中没有异常的处理逻辑（try。。。catch），那么JVM就会把异常对象抛出给方法的调用者
        JVM接收到了这个异常，会做两件事情
          1.把异常对象（内容，原因，位置）以红色的字体打印到控制台
          2.JVM会中断当前程序
        被调用的方法--》main方法--》JVM
        */
    }
    public static int getElement(int[]arr,int index){
        int ele=arr[index];
        return ele;
    }
}
```

异常的处理方法

```
package Demo02.Demo001;
/*
throw关键字：
     作用：
        可以使用throw关键字在指定的方法中抛出指定的异常
     使用格式：
        throw new xxxException（"异常产生的原因"）；
     注意：
        1.throw关键字必须写在方法的内部
        2.throw关键字后new的对象必须是Exception或者Exception的子类
        3.throw关键字抛出指定的异常对象，我们就必须处理这个异常对象
                throw关键字后创建的是RuntimeException或者是RuntimeException的子类对象，可以不处理，交给JVM
                throw关键字后创建的是编译异常，就必须处理
        在工作中，要对方法传进来的参数进行合法性检验
        注意：NullPointerException是一个运行期异常，不用处理，默认交给JVM
*/
public class PackDemo03 {
    public static void main(String[] args) {
        int[]arr={1,2,3};
        int element = getElement(arr, 0);

    }
    public static int getElement(int[]arr,int index){
        if(arr==null){
```

```
            throw new NullPointerException("传递数组为空");
        }
        //如果传入的索引超出范围，抛出异常
        if(index<0||index>= arr.length){
            throw new ArrayIndexOutOfBoundsException("数组越界");
        }
        int i=arr[index];
        return i;
    }
}
```

Object非空判断

```
package Demo02.Demo001;

import java.util.Objects;

/*
Object类中的静态方法
public static <T> T requireNonNull(T obj):查看指定的对象是不是null
源码:
   public static <T> T requireNonNull(T obj){
          if(obj==null){
            throw new NullPointerException();
          }
          return obj;
   }
*/
public class PackDemo04 {
    public static void main(String[] args) {
        metnod(null);
    }

    public static void metnod(Object obj) {
        //自动抛出异常，不用再写了
        //Objects.requireNonNull(obj);
        Objects.requireNonNull(obj,"参数为空");
    }
}
```

throws处理异常

```
package Demo02.Demo001;

import java.io.FileNotFoundException;
import java.io.IOException;

/*
throws关键字：异常处理的第一种方式，交给别人处理
作用:
   当方法内部抛出异常对象的时候，那我们就必须处理这个异常对象
   可以使用throws关键字处理异常对象，就会把异常对象声明抛出给方法的调用者处理，最终交给JVM--》
中断
注意:
   1.throws关键字必须写在方法声明处
   2.throws关键字后面声明的异常必须是Exception或者是Exception的子类
```

```
   3.方法内部如果抛出了多个异常对象，那么throws也必须声明多个异常
   如果抛出的的异常有字符类关系，那么直接声明父类异常即可
   4.调用了一个声明抛出异常的方法，我们就必须处理声明的异常
       要么继续使用throws声明抛出，交给方法的调用者处理，最终交给JVM
       要么try。。。catch自己处理异常
*/
public class PackDemo05 {
    public static void main(String[] args) throws
FileNotFoundException,IOException {
        //调用了抛出异常的方法，自己也需要进行抛出
        readfile("a");
    }
    /*
    FileNotFoundException()是编译异常，编译异常需要处理
    */
    public static void readfile(String filename) throws
FileNotFoundException,IOException {
        if(!filename.equals("as")){
            throw new FileNotFoundException("文件路径错误");
        }
        if(!filename.endsWith("s")){
            throw new IOException("结尾错误");
        }
        System.out.println("读取成功");

    }
}
```

try catch

```
package Demo02.Demo001;

import java.io.FileNotFoundException;
import java.io.IOException;

/*
try...catch:异常处理的第二种方法
try{
   //可能产生异常的代码
}catch (定义一个异常的变量){
   异常的处理逻辑，怎么处理异常对象
}
。。。。
catch(){

}
注意：
   1.try中可能会抛出多个异常，那么就可以使用多个catch来处理异常
   2.如果try中产生了异常，那么就会执行catch中的异常处理逻辑，执行完毕catch中的处理逻辑，继续
执行之后的代码
     如果try中没有产生异常，那么就不会执行catch中的异常处理逻辑，执行完try中的代码，继续执行
try。。。catch之后的代码
*/
public class PackDemo006 {
    public static void main(String[] args) {
        try{
            readfile("a");
```

```
        }catch(IOException e){//对异常进行捕捉，检测到异常后执行异常执行catch之后的代码
            /*
            Throwable类中定义了3个异常处理的方法
               String getMessage() 返回此throwable的简短描述
               String toString() 返回此throwable的详细信息字符串
               void printStackTrace() JVM打印异常，默认此方法，打印异常信息是最全面的
            */
            System.out.println(e.getMessage());//打印异常中的内容
            System.out.println(e.toString());//会多打印异常名称
            e.printStackTrace();//打印的信息很全面
        }

    }
    public static void readfile(String filename) throws
FileNotFoundException,IOException {
        if(!filename.equals("as")){
            throw new FileNotFoundException("文件路径错误");
        }
        if(!filename.endsWith("s")){
            throw new IOException("结尾错误");
        }
        System.out.println("读取成功");
    }
}
```

finally代码块

```
package Demo02.Demo001;

import java.io.FileNotFoundException;
import java.io.IOException;

/*
finally代码块
和try代码块是一起使用的
注意：
   1.finall代码快不能单独使用，必须和try一起使用
   2.finally一般用于资源释放（资源回收），无论程序是否出现异常，最后都要释放（IO）
*/
public class PackDemo07 {
    public static void main(String[] args) {
        try {
            readfile("as");
        } catch (IOException e) {
            e.printStackTrace();
        }finally {//无论是否出现异常都会执行
            System.out.println("资源释放");
        }
    }
    public static void readfile(String filename) throws FileNotFoundException,
IOException {
        if(!filename.equals("as")){
            throw new FileNotFoundException("文件路径错误");
        }
        if(!filename.endsWith("s")){
            throw new IOException("结尾错误");
        }
```

```
        System.out.println("读取成功");
    }
}
```

异常的注意事项

多异常处理

```
package Demo02.Demo001;
/*
多个异常分别处理
多个异常一次捕获，多次处理
多个异常一次捕获，一次处理
*/
import java.util.List;

public class PackDemo08 {
    public static void main(String[] args) {
        int[]arr={1,2,3};

        try{
            System.out.println(arr[3]);
        }catch(ArrayIndexOutOfBoundsException e){
            e.printStackTrace();
        }
        List<Integer> list = List.of(1, 2, 3);
        try {
            System.out.println(list.get(3));
        }catch (IndexOutOfBoundsException e){
            e.printStackTrace();
        }

        //一次捕获，多次处理
        /*
        catch里定义的异常变量，如果有子父类关系，那么子类异常变量必须写在上面，否则会报错

        */
        try{
            System.out.println(arr[3]);
            System.out.println(list.get(3));

        }catch (ArrayIndexOutOfBoundsException e){
            e.printStackTrace();
        }catch (IndexOutOfBoundsException e){
            e.printStackTrace();
        }
        //一次捕获，一次处理
        try{
            System.out.println(arr[3]);
            System.out.println(list.get(3));

        }catch (Exception e){
            e.printStackTrace();
        }

        //运行期异常可以不捕获，不处理
        //默认交给虚拟机
```

```
    }
}
```

finally含return

```
package Demo02.Demo001
/*
如果finally中有return语句，永远返回finally中的结果，避免该情况
*/
public class PackDemo09 {
    public static void main(String[] args) {
        int a = getA();
        System.out.println(a);
    }

    public static int getA() {
        int a=10;
        try{
            return a;
        }catch (Exception e){
            System.out.println(e);
        }finally {
            a=100;
            return a;
        }
    }
}
```

子父类异常

```
package Demo02.Demo001;
/*
子父类的异常：
      如果父类抛出了多个异常，子类重写父类方法时，抛出和父类相同的异常或者是父类异常的子类或者不抛出
      父类方法没有抛出异常，子类重写父类方法时也不能抛出异常。此时子类产生该异常，只能捕获，不能抛出
注意：
   父类出现什么异常，子类就出现什么异常
*/
public class PackDemo10 {
    public void show01()throws NullPointerException,ClassCastException{

    }
    public void show02()throws IndexOutOfBoundsException{

    }
    public void show03()throws IndexOutOfBoundsException{

    }
}
class zi extends PackDemo10{
    public void show01()throws NullPointerException,ClassCastException{

    }
    public void show02() throws ArrayIndexOutOfBoundsException{
```

```
    }
    public void show03(){

    }
}
```

自定义异常类

```
package Demo02.Demo001;
/*
自定义异常类
java提供的不够，需自定义
注意：
   1.自定义异常类一般都是以Exception结尾
   2.自定义异常类必须继承Exception或者RuntimeException
        继承Exception：那么就是一个编译期异常，如果方法中抛出这个异常，就必须处理
        继承RuntimeException：那么就是一个运行期异常，无需处理，交给虚拟机
*/
public class RegisterException extends Exception{
    //添加一个空参数的构造方法
    public RegisterException() {
        super();
    }
    /*
    添加一个带异常信息的构造方法
    查看源码发现，所有的异常类都会有一个带异常信息的构造方法，方法内部会调用父类带异常信息的构造方法
    */

    public RegisterException(String message) {
        super(message);
    }
}
```

```
package Demo02.Demo001;

import java.util.Scanner;

/*
分析：
   1.使用数组保存已注册的用户名（数据库）
   2.使用Scanner获取用户输入的注册的用户名（前端，页面） 、
   3.定义一个方法，遍历数组。获取每一个用户名，使用获取到的和输入的进行比较
*/
public class PackDemoPractice {
    static String[]usernames={"3","4","5"};
    public static void main(String[] args) throws RegisterException {
        Scanner sc=new Scanner(System.in);
        System.out.println("请输入您要注册的用户名");
        String username=sc.next();
        check(username);
    }
    public static void check(String username) throws RegisterException {
        for(String s:usernames){
            if(s.equals(username)){
                throw new RegisterException("该用户已被注册");
```

```
            }
        }
        System.out.println("注册成功");
    }
}
```

多线程

并发与并行

并发：指两个或者多个事件在同一个时间段内发生（交替执行）

并行：指两个或者多个事件在同一时刻发生（同时发生）

进程与线程：

进程：是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。

线程：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程

```
package Demo02.Demo002;
/*
主线程：执行主方法（main）的线程
单线程程序：java程序中只有一个线程
执行从main方法开始，依次向下执行

创建多线程程序的第一种方法：创建Thread子类
java.lang.Thread类：是描述线程的类，我们想要实现多线程程序，就必须继承Thread类
实现步骤：
   1.创建一个Thread子类
   2.在Thread类的子类中重写Thread类中的run方法，设置线程任务（开启线程要做什么）
   3.创建Thread的子类对象
   4.调用start方法
结果是两个线程并发的进行；当前线程（main）和另一线程（创建的新线程，执行run方法）
多次启动一个线程是犯法的。特别是线程结束后，不再重新启动
java程序属于抢占式调度，哪个优先级高就执行哪个，同级别随机执行
*/
public class PackDemo001 {
    public static void main(String[] args) {
        person p1=new person("koi");
        p1.run();
        //中间出现异常后续代码无法执行
        /*person p2=new person("kois");
        p2.run();*/
        MyThread mp=new MyThread();
        mp.start();
        for (int i = 0; i <20 ; i++) {
            System.out.println("main"+i);
        }
    }
}
```

多线程原理

Thread

```
package Demo02.Demo002;

public class MyThread extends Thread{
    @Override
    public void run() {
        String name = getName();
        System.out.println(name);
        Thread thread = Thread.currentThread();
        System.out.println(thread);
        //链式编程
        System.out.println(Thread.currentThread().getName());
    }
}
```

```
package Demo02.Demo002;
/*
获取线程的名称：
  1.使用Thread类中的方法getName()
     String getName()返回该线程的名称
  2.可以先获取到当前正在执行的线程，使用线程中的方法getName()获取线程的的名称
     static Thread currentThread()返回对当前正在执行的线程对象的引用
*/
public class PackDemo002 {
    public static void main(String[] args) {
        MyThread mp=new MyThread();
        mp.start();
        MyThread mp1=new MyThread();
        mp1.start();
        System.out.println(Thread.currentThread().getName());
    }
}
```

设置线程名称

```
package Demo02.Demo002;

public class MyThread extends Thread{
    public MyThread() {
    }

    public MyThread(String name) {
        super(name);//把线程名称传递给父类，让父类（Thread）给子线程起一个名称
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}

package Demo02.Demo002;
/*
```

```
设置线程名称：
   1.使用Thread类中的方法setName（名字）
   2.创建一个带参数的构造方法，参数传递线程的名称，调用父类的带参构造方法，把线程名称传递给父
类，让父类（Thread）给子线程起一个名称

*/
public class PackDemo003 extends Thread{
    public static void main(String[] args) {
        MyThread mp=new MyThread();
        mp.setName("koi");
        mp.start();
        MyThread kkk = new MyThread("kkk");
        kkk.start();
    }
}
```

Thread类常用方法

```
package Demo02.Demo002;
/*
sleep(long millis):使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行）
毫秒数停止之后，线程继续执行
*/
public class PackDemo003 extends Thread{
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 20; i++) {
            System.out.println(i);
            Thread.sleep(1000);
        }
    }
}
```

Runnable

```
package Demo02.Demo002;
/*
创建多线程程序的第二种方式：
    java.lang.Runnable
        Runnable接口应该由那些打算通过某一线程执行其实例的类来实现。类必须定义一个run的无参数
方法
实现步骤：
    1.创建一个Runnable接口的实现类
    2.在实现类中重写Runnable接口中的run方法，设置线程任务
    3.创建一个Runnable接口的实现类对象
    4.创建Thread类对象，构造方法中传递Runnable接口的实现类对象
    5.调用Thread类中的start方法，开启新线程执行run方法

实现Runnable接口创建多线程的好处：
   1.避免了单继承的局限性
        一个类只能继承一个接口，继承了Thread就不能继承其他的类
        实现了Runnable接口，还可以继承其他的类
   2.增强了程序的扩展性，就是降低了耦合性
     实现了Runnable接口的方式，把设置线程任务和开启新线程进行了分离
     在实现类中，重写了run方法，用来设置线程任务
*/
```

```
public class PackDeno005 {
    public static void main(String[] args) {
        Runnableimp run=new Runnableimp();
        Thread t=new Thread(run);
        t.start();
        for (int i = 0; i < 20; i++) {
            System.out.println(Thread.currentThread().getName()+i);
        }
    }
}
```

匿名内部类实现多线程

```
package Demo02.Demo002;
/*
匿名内部类方式实现线程的创建
匿名：没有名字
内部类：写在其他内部的类
匿名内部类作用：简化代码
    把子类继承父类，重写父类的方法，创建子类对象合成一部完成
    把实现类接口，重写接口中的方法，创建实现类对象合成一步完成
*/
public class PackDemo004 {
    public static void main(String[] args) {
        new Thread(){
            @Override
            public void run() {
                for (int i = 0; i < 20; i++) {
                    System.out.println(Thread.currentThread().getName()+i);
                }
            }
        }.start();
        //线程的接口
        Runnable r=new Runnable(){
            @Override
            public void run() {
                for (int i = 0; i < 20; i++) {
                    System.out.println(Thread.currentThread().getName()+i);
                }
            }
        };
        new Thread(r).start();
    }
}
```

线程安全问题

线程安全问题原理

线程安全问题是不能产生的，我们可以让一个线程在访问共享数据时，无论是否失去了cpu执行权，都只能等待。

线程同步

同步代码块

```
package Demo02.Demo003;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/*
出现了线程安全问题
第一种方法：同步代码块
格式：
   synchronized(锁对象){
         可能会出现线程安全问题的代码（访问了共享数据的代码）
   }
注意：
    1.通过代码块中的锁对象，可以使用任意的对象
    2.但是必须保证多个线程中使用的锁对象时同一个
    3.锁对象作用
        把同步代码块锁住，只让一个线程在同步代码块中执行
第二种方法：使用同步方法
    使用步骤：
       1.把访问了共享数据的代码抽取出来，放到一个方法中
       2.在方法上加上synchronized修饰符
第三种方法：lock锁
     Lock提供了比使用synchronized方法和语句可获得的更广泛的锁定操作
     lock()获取锁
     unlock()释放锁
     使用步骤：
        1.在成员位置创建一个ReentrantLock对象
        2.在可能会出现安全问题的代码前调用Lock接口中的方法Lock获取锁
        3.在可能会出现安全问题的代码后调用Lock接口中的方法unlock释放锁
*/
public class Runnableimp implements Runnable{
    //设置线程任务
    //创建锁对象
    Object obj=new Object();
    private static int tack=100;
    /*@Override
    public void run() {
        //创建同步代码块
        synchronized (obj){
            while(tack>0){
                //提高几率，让程序睡眠一下
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+"正在
卖"+tack);
                tack--;
            }
        }
        //也可调用同步方法
    }*/
```

```
    //创建同步方法
    /*public synchronized void payTick(){
        while(tack>0){
            //提高几率，让程序睡眠一下
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+"正在卖"+tack);
            tack--;
        }
    }*/
    //静态同步方法
    //只有静态方法才能访问静态方法
    /*
    锁对象不再是this
    this是创建之后产生的，静态方法优先于对象
    静态方法的锁对象是本类的class属性
    */
    public static void payTickstatic(){
        synchronized (Runnableimp.class){
            while(tack>0){
                //提高几率，让程序睡眠一下
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+"正在
卖"+tack);
                tack--;
            }
        }
    }
    Lock l=new ReentrantLock();
    @Override
    public void run() {
        l.lock();
        while(tack>0){
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally{
                l.unlock();
            }
            System.out.println(Thread.currentThread().getName()+"正在卖"+tack);
            tack--;
        }
    }
}
```

同步技术原理

使用了一个锁对象，这个锁对象叫同步锁，也叫对象锁，也叫对象监视器。第一个线程会先检测同步代码块是否有锁对象，发现有就获取锁对象，第二个发现没有锁对象，就会堵塞，直到第一个线程返还锁对象。 （效率会有些降低）

线程状态

等待唤醒案例

```
package Demo02.Demo004;
/*
等待唤醒案例：线程之间的通信
    创建一个顾客线程：得到信息后，调用wait
    创建一个老板线程：完成后调用notify
注意：
   两个线程必须用同步代码块包起来，保证只有一个在执行
   同步使用的锁对象必须保证唯一
   只有锁对象才能调用wait和notify方法
*/
public class PackDemo01 {
    public static void main(String[] args) {
        Object obj=new Object();
        new Thread(){
            @Override
            public void run() {
                //保证等待和唤醒只有一个再执行
                synchronized (obj){
                    System.out.println("告知包子的种类和数量");
                    try {
                        obj.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("吃");
                }
            }
        }.start();
        new Thread(){
            @Override
            public void run() {
                //花五秒做包子
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (obj){
                    System.out.println("可以吃了");
                    obj.notify();
                }
            }
        }.start();
    }
}
```

```
package Demo02.Demo004;
```

```java
/*
进入到TimeWaiting(记时等待)有两种方式
1.使用sleep方法，在毫秒值结束之后，线程睡醒后进入到Runnable/Block状态
2.使用wait方法，如果在毫秒值结束还没有notify，就会进入上述状态
唤醒的方法：
   notify
   notifyAll
*/
public class PackDemo02 {
    public static void main(String[] args) {
        Object obj=new Object();
        new Thread(){
            @Override
            public void run() {
                synchronized (obj){
                    System.out.println("顾客一告知包子的种类和数量");
                    try {
                        obj.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("顾客一吃");
                }
            }
        }.start();
        new Thread(){
            @Override
            public void run() {
                synchronized (obj){
                    System.out.println("顾客二告知包子的种类和数量");
                    try {
                        obj.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("顾客二吃");
                }
            }
        }.start();
        new Thread(){
            @Override
            public void run() {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (obj){
                    System.out.println("可以吃了");
                    //obj.notify();随机唤醒一个
                    obj.notifyAll();
                }
            }
        }.start();
    }
}
```

线程间通信

```
package Demo02.Demo004;

public class eat extends Thread{
    private baozi bz;
    public eat(baozi bz){
        this.bz=bz;
    }
    @Override
    public void run() {
        while(true){
            synchronized (bz){
                if(bz.flag==false){
                    try {
                        bz.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                //被唤醒后
                System.out.println("正在吃包子"+bz.pi+bz.xian+"包真好吃");
                bz.flag=false;
                bz.notify();
                System.out.println("吃完了，接着做");
            }
        }
    }
}
```

```
package Demo02.Demo004;
/*
包子铺线程和包子线程为互斥关系，使用同步技术
锁对象必须保证唯一，可以使用包子对象作为锁对象
*/
public class store extends Thread{

    private baozi bz;

    public store(baozi bz) {
        this.bz = bz;
    }

    @Override
    public void run() {
        int cnt=0;
        while(true){
            synchronized (bz){
                if (bz.flag==true){
                    try {
                        bz.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                if(cnt%2==0){
```

```
                    bz.xian="三鲜";
                    bz.pi="薄皮";
                }else {
                    bz.pi="冰皮";
                    bz.xian="牛肉";
                }
                cnt++;
                System.out.println("正在生产"+bz.pi+bz.xian+"包");
                //三秒生产
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                bz.flag=true;
                bz.notify();
                System.out.println("包子做好了，是"+bz.pi+bz.xian+"包");
            }
        }
    }
}
```

线程池

```
package Demo02.Demo005;

import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/*
JDK1.5之后提供的
java.util.concurrent.Executors:线程池的工厂类,用来生产线程池
静态方法:
   static ExecutorService newFixedThreadPool(int nThreads)创建一个指定数量的线程池
   参数:
      int nThreads线程池中的线程数量
   返回值：ExecutorService接口，返回值是其实现类，但我们可以用其接受
java.util.concurrent.ExecutorService:线程池接口
   用来从线程池中获取线程，调用start方法，执行线程任务
       submit(Runnable task)提交任务用于执行
   关闭/销毁任务
       void shutdown()
线程池使用步骤:
   1.使用工厂类生产一个线程池
   2.创建一个类，实现Runnable接口，重写run方法，设置线程任务
   3.调用submit方法，传递线程任务，开启线程
   4.调用shutdown销毁线程池
 */
public class PackDemo01 {
    public static void main(String[] args) {
        ExecutorService es = Executors.newFixedThreadPool(2);
        es.submit(new runnableimp());
        //线程池会一直开启，使用完了线程，会把线程归还给线程池，线程可以继续使用
        es.submit(new runnableimp());
        es.submit(new runnableimp());
        es.shutdown();
```

```
    }
}
```

Lambda表达式

体验

```
package Demo02.Demo005;

public class PackDemo03 {
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName()+"创建了一个新线
程");
            }
        }).start();
        //使用Lambda表达式
        new Thread(()-> {
            System.out.println(Thread.currentThread().getName()+"创建了一个新线
程");
        }).start();
    }
}
```

```
package Demo02.Demo005;
/*
Lambda表达式的标准格式
  1.一些参数
  2.一个箭头
  3.一段代码
格式:
   (参数列表)->{一些重写方法的代码};
   解释说明:
      ():接口中抽象方法的参数列表，没有参数，就空着
      ->:传递的意思，把参数传递给方法体{}
      {}:重写接口的抽象方法的方法体
*/
public class PackDemo04 {
    public static void main(String[] args) {
        invokecook(new cook(){
            @Override
            public void makefood() {
                System.out.println("干饭了");
            }
        });
        //使用Lambda表达式
        invokecook(()->{
            System.out.println("干饭了");
        });
    }
    //定义一个方法，参数传递cook接口，方法内部调用makefood()方法
    public static void invokecook(cook cook){
        cook.makefood();
    }
```

```
}
```

有参有返回

```
package Demo02.Demo005;

import java.util.Arrays;
import java.util.Comparator;

public class PackDemo05 {
    public static void main(String[] args) {
     person[]arr={new person("koi",18),
             new person("koimisaka",15),
             new person("koitakaji",14)};
     /*Arrays.sort(arr, new Comparator<person>() {
            @Override
            public int compare(person o1, person o2) {
                return o1.getAge()-o2.getAge();
            }
        });*/
     //Lambda表达式
     Arrays.sort(arr,(person p1,person p2)->{
         return p1.getAge()-p2.getAge();
     });
     for(person p:arr){
         System.out.println(p);
     }
    }
}
```

```
package Demo02.Demo005;
/*
有参数有返回值
需求：
   定义一个计算器Calculator接口，内含抽象方法可计算和
   使用Lambda表达式调用invokecalc方法，完成计算
*/
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PackDemo06 {
    public static void main(String[] args) {
        //调用invokecalc方法，方法参数是一个接口，可以使用匿名内部类
        invokecalc(10,20,(int a,int b)->{
           return a+b;
        });
    }

    public static int invokecalc(int a,int b,calcualtor ca) {
        int sum=ca.calc(a,b);
        System.out.println(sum);
        return sum;
    }
}
```

省略

```
package Demo02.Demo005;
/*
Lambda表达式：可推导的内容可省略
凡是根据上下可以推导出来的内容，可以省略
可以省略的内容：
   1.（参数列表）：括号中参数列表的数据类型可以省略
   2.（参数列表）：括号中的参数如果只有一个，那么类型和（）都可以省略
   3.{一些代码}：如果{}中的代码只有一行，无论是否有返回值，都可以省略（{}，return，;）
          注意：要省略必须一起省略
*/

import java.util.Arrays;

public class PackDemo07 {
    public static void main(String[] args) {
        //JDK1.7之前，创建集合对象要把泛型都写上
        new Thread(()->
            System.out.println(Thread.currentThread().getName()+"创建了一个新线程")
        ).start();
        person[]arr={new person("koi",18),
                new person("koimisaka",15),
                new person("koitakaji",14)};

        //Lambda表达式
        Arrays.sort(arr,(p1, p2)->
             p1.getAge()-p2.getAge()
        );
    }
}
```

注意事项

1. 使用Lambda必须具有接口，且要求**接口中有且仅有一个抽象方法。**
   无论是JDK内置的`Runnable`、`Comparator`接口还是自定义的接口，只有当接口中的抽象方法存在且唯一时，才可以使用Lambda。
2. 使用Lambda必须具有**上下文推断。**
   也就是方法的参数或局部变量类型必须为Lambda对应的接口类型，才能使用Lambda作为该接口的实例。

File类与IO流

File类

概述

```
package Demo03.Demo001;

import java.io.File;

/*
java.io.File 文件和目录路径名的抽象表示
java把电脑中的文件和文件夹（目录）封装为了一个File类，我们可以使用File类对文件和文件夹进行操作
File是一个与文件无关的类，任何操作系统都可以使用其中方法
```

```
重要：
   file：文件
   directory：文件夹/目录
   path：路径
*/
public class PackDemo01 {
    public static void main(String[] args) {
        String pathSeparator = File.pathSeparator;
        System.out.println(pathSeparator);//路径分隔符 Windows ; Linux :
        String separator = File.separator;
        System.out.println(separator);//文件名称分割符 Windows \ Linux /
        //路径不能写死了，在不同系统不一样
    }
}
```

构造

```
package Demo03.Demo001;

import java.io.File;

public class PackDemo03 {
    public static void main(String[] args) {
        show01();
        show02("c:\\","a.txt");
        show03();
    }

    /*
    File(File parent, String child) 从父抽象路径名和子路径名字符串创建新的 File实例。
    好处：
       父路径和子路径，可以单独书写，使用起来非常灵活，父路径和子路径都可以变化
       父路径是File类型，可以使用File的方法对路径进行一些操作，再使用路径创建对象
    */
    private static void show03() {
        File parent = new File("c:\\");
        File file = new File(parent,"hello.java");
        System.out.println(file);
    }

    /*
    File(String parent, String child) 从父路径名字符串和子路径名字符串创建新的 File实
例。
    参数：把路径分为两部分
       String parent:父路径
       String child:子路径
    好处：
       父路径和子路径，可以单独书写，使用起来非常灵活，父路径和子路径都可以变化
    */
    private static void show02(String parent,String child) {
        File file = new File(parent,child);
        System.out.println(file);
    }

    /*
    File(String pathname)通过将给定的路径名字符转换为抽象路径名来创建一个新的File实例
    参数：
```

```
        String pathname：字符串的路径名称
        路径可以以文件结尾也可以以文件夹结尾
        路径可以是相对路径，也可以是绝对路径
        路径可以存在也可以不存在
        创建File对象，只是把字符串路径封装为File对象，不考虑路径的真假情况
    */
    private static void show01() {
        File fi=new File("E:\\兴趣开放\\oj\\后端\\Java基础II\\07-网络编程\\第3节 综合案例_文件上传");
        System.out.println(fi);//重写了toString
        File f1=new File("E:\\兴趣开放\\oj");
        System.out.println(f1);
    }
}
```

常用方法

获取功能的方法

- `public String getAbsolutePath()`：返回此File的绝对路径名字符串。
- `public String getPath()`：将此File转换为路径名字符串。
- `public String getName()`：返回由此File表示的文件或目录的名称。
- `public long length()`：返回由此File表示的文件的长度。

  方法演示，代码如下：

```
public class FileGet {
    public static void main(String[] args) {
        File f = new File("d:/aaa/bbb.java");
        System.out.println("文件绝对路径:"+f.getAbsolutePath());
        System.out.println("文件构造路径:"+f.getPath());
        System.out.println("文件名称:"+f.getName());
        System.out.println("文件长度:"+f.length()+"字节");

        File f2 = new File("d:/aaa");
        System.out.println("目录绝对路径:"+f2.getAbsolutePath());
        System.out.println("目录构造路径:"+f2.getPath());
        System.out.println("目录名称:"+f2.getName());
        System.out.println("目录长度:"+f2.length());
    }
}
输出结果：
文件绝对路径:d:\aaa\bbb.java
文件构造路径:d:\aaa\bbb.java
文件名称:bbb.java
文件长度:636字节

目录绝对路径:d:\aaa
目录构造路径:d:\aaa
目录名称:aaa
目录长度:4096
```

> API中说明：length()，表示文件的长度。但是File对象表示目录，则返回值未指定。

绝对路径和相对路径

- **绝对路径**：从盘符开始的路径，这是一个完整的路径。
- **相对路径**：相对于项目目录的路径，这是一个便捷的路径，开发中经常使用。

```
public class FilePath {
    public static void main(String[] args) {
        // D盘下的bbb.java文件
        File f = new File("D:\\bbb.java");
        System.out.println(f.getAbsolutePath());

        // 项目下的bbb.java文件
        File f2 = new File("bbb.java");
        System.out.println(f2.getAbsolutePath());
    }
}
输出结果：
D:\bbb.java
D:\idea_project_test4\bbb.java
```

判断功能的方法

- `public boolean exists()` ： 此File表示的文件或目录是否实际存在。
- `public boolean isDirectory()` ： 此File表示的是否为目录。
- `public boolean isFile()` ： 此File表示的是否为文件。

方法演示，代码如下：

```
public class FileIs {
    public static void main(String[] args) {
        File f = new File("d:\\aaa\\bbb.java");
        File f2 = new File("d:\\aaa");
        // 判断是否存在
        System.out.println("d:\\aaa\\bbb.java 是否存在:"+f.exists());
        System.out.println("d:\\aaa 是否存在:"+f2.exists());
        // 判断是文件还是目录
        System.out.println("d:\\aaa 文件?:"+f2.isFile());
        System.out.println("d:\\aaa 目录?:"+f2.isDirectory());
    }
}
输出结果：
d:\aaa\bbb.java 是否存在:true
d:\aaa 是否存在:true
d:\aaa 文件?:false
d:\aaa 目录?:true
```

创建删除功能的方法

- `public boolean createNewFile()` ： 当且仅当具有该名称的文件尚不存在时，创建一个新的空文件。
- `public boolean delete()` ： 删除由此File表示的文件或目录。
- `public boolean mkdir()` ： 创建由此File表示的目录。
- `public boolean mkdirs()` ： 创建由此File表示的目录，包括任何必需但不存在的父目录。

方法演示，代码如下：

```java
public class FileCreateDelete {
    public static void main(String[] args) throws IOException {
        // 文件的创建
        File f = new File("aaa.txt");
        System.out.println("是否存在:"+f.exists()); // false
        System.out.println("是否创建:"+f.createNewFile()); // true
        System.out.println("是否存在:"+f.exists()); // true

        // 目录的创建
        File f2= new File("newDir");
        System.out.println("是否存在:"+f2.exists());// false
        System.out.println("是否创建:"+f2.mkdir()); // true
        System.out.println("是否存在:"+f2.exists());// true

        // 创建多级目录
        File f3= new File("newDira\\newDirb");
        System.out.println(f3.mkdir());// false
        File f4= new File("newDira\\newDirb");
        System.out.println(f4.mkdirs());// true

        // 文件的删除
        System.out.println(f.delete());// true

        // 目录的删除
        System.out.println(f2.delete());// true
        System.out.println(f4.delete());// false
    }
}
```

> API中说明：delete方法，如果此File表示目录，则目录必须为空才能删除。

目录的遍历

- `public String[] list()`：返回一个String数组，表示该File目录中的所有子文件或目录。
- `public File[] listFiles()`：返回一个File数组，表示该File目录中的所有的子文件或目录。

```java
public class FileFor {
    public static void main(String[] args) {
        File dir = new File("d:\\java_code");

        //获取当前目录下的文件以及文件夹的名称。
        String[] names = dir.list();
        for(String name : names){
            System.out.println(name);
        }
        //获取当前目录下的文件以及文件夹对象，只要拿到了文件对象，那么就可以获取更多信息
        File[] files = dir.listFiles();
        for (File file : files) {
            System.out.println(file);
        }
    }
}
```

> 小贴士：
>
> 调用listFiles方法的File对象，表示的必须是实际存在的目录，否则返回null，无法进行遍历。

递归

打印多级目录

```
package Demo03.Demo001;

import java.io.File;

public class PackDemo06 {

    public static void main(String[] args) {
        File file = new File("E:\\java\\study\\src\\Demo03\\Demo001");
        getAllFile(file);
    }
    /*
    定义一个方法，参数传递File类型的目录
    方法中对目录进行遍历
    */
    public static void getAllFile(File dir){
        File[] files = dir.listFiles();
        for(File f:files){
            if(f.isDirectory()==true){
                getAllFile(f);
            }else {
                System.out.println(f);
            }
        }
    }
}
```

筛选文件

```
package Demo03.Demo001;

import java.io.File;

public class PackDemo07 {

    public static void main(String[] args) {
        File file = new File("E:\\java\\study\\src\\Demo03\\Demo001");
        getAllFile(file);
    }
    /*
    定义一个方法，参数传递File类型的目录
    方法中对目录进行遍历,只要.java结尾的文件
    */
    public static void getAllFile(File dir){
        File[] files = dir.listFiles();
        for(File f:files){
            if(f.isDirectory()==true){
                getAllFile(f);
            }else {
                /*
                只要.java结尾
                先转换为字符串，再调用方法
                若不区分大小写，可以先全都转换为小写
```

```
                */
                /*String name=f.getName();
                String s = name.toLowerCase();
                if(s.endsWith(".java")){
                    System.out.println(f);
                }*/
                //链式编程
                if(f.getName().toLowerCase().endsWith(".java")){
                    System.out.println(f);
                }
            }
        }
    }
}
```

过滤器

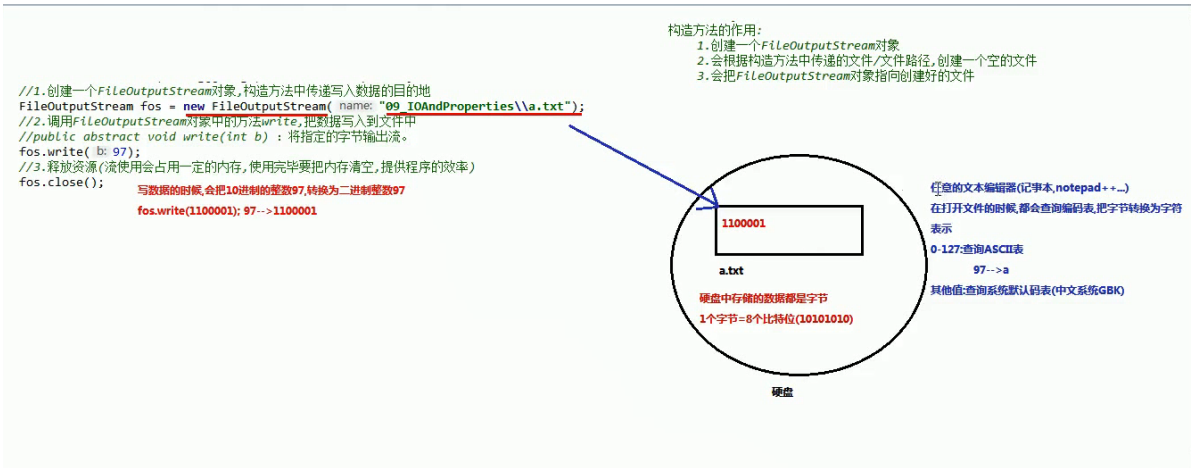
```
package Demo03.Demo002;

import java.io.File;

/*
java.io.FileFilter接口：用于抽象路径名（File对象）的过滤器
作用：用来过滤文件（File对象）
抽象方法：用来过滤文件的方法
boolean accept(File pathname)测试指定抽象路径名是否应该包含在某个路径名列表中
参数：
   File pathname:使用ListFiles方法遍历目录，得到的每一个文件对象
File[] listFiles(FilenameFilter filter)
java.io.FilenameFilter接口：实现此接口的类实例可用于过滤器文件名
作用：用于过滤文件名称
抽象方法：用来过滤文件的方法
boolean accept(File dir,String name)测试指定文件是否应该包含在某一文件列表中
参数：
   File dir:构造方法中传递的被遍历的目录
   String name:使用ListFiles方法遍历目录，获取的每一个文件/文件夹的名称
实现此接口的类实例可用于过滤器文件名
注意：
   两个过滤器接口是没有实现类的，需要我们自己写实现类，重写过滤的方法accept，在方法中定义自己的过滤规则
*/
public class PackDemo01 {
    public static void main(String[] args) {
        File file = new File("E:\\java\\study\\src\\Demo03\\Demo001");
        getAllFile(file);
    }
    public static void getAllFile(File dir){
        File[] files = dir.listFiles(new FileFilterImp());
        for(File f:files){
            if(f.isDirectory()==true){
                getAllFile(f);
            }else {
                System.out.println(f);
            }
        }
    }
}
```

```java
package Demo03.Demo002;

import java.io.File;
import java.io.FileFilter;

public class FileFilterImp implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        String name = pathname.getName();
        if(name.toLowerCase().endsWith(".java")){
            return true;
        }else if(pathname.isDirectory()){
            return true;
        }else{
            return false;
        }
    }
}
```

```java
package Demo03.Demo002;

import java.io.File;
import java.io.FileFilter;
import java.io.FilenameFilter;


public class PackDemo02 {
    public static void main(String[] args) {
        File file = new File("E:\\java\\study\\src\\Demo03\\Demo001");
        getAllFile(file);
    }
    public static void getAllFile(File dir){
        /*File[] files = dir.listFiles(new FileFilter(){
            @Override
            public boolean accept(File pathname) {
                String name = pathname.getName();
                if(name.toLowerCase().endsWith(".java")){
                    return true;
                }else if(pathname.isDirectory()){
                    return true;
                }else{
                    return false;
                }
            }
        });*/
        File[] files = dir.listFiles((File pathname)->{
            String name = pathname.getName();
            if(name.toLowerCase().endsWith(".java")){
                return true;
            }else if(pathname.isDirectory()){
                return true;
            }else{
                return false;
            }
```

```
        });
        /*File[] files = dir.listFiles(new FilenameFilter() {
            @Override
            public boolean accept(File dir, String name) {
                return new
File(dir,name).isDirectory()||name.toLowerCase().endsWith(".java");
            }
        });*/
        //使用Lambda（接口中只有一个抽象方法）
        /*File[] files=dir.listFiles((File d,String name)->{
            return new
File(dir,name).isDirectory()||name.toLowerCase().endsWith(".java");
        });*/
        for(File f:files){
            if(f.isDirectory()==true){
                getAllFile(f);
            }else {
                System.out.println(f);
            }
        }
    }
}
```

IO字节流



OutputStream

```
package Demo03.Demo002;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/*
java.io.OutputStream:此抽象类是表示输出字节流的所有类的超类。
java.io.FileOutputStream extends OutputStream
文件字节输出流：把内存中的数据写入到硬盘的文件中
构造方法：
   FileOutputStream(String name)创建一个向具有指定名称的文件中写入数据的输出文件流
   FileOutputStream(File file)创建一个向指定File对象表示的文件中写入数据的文件输出流
   参数：写入数据的目的地
      String name:目的地是一个文件的路径
```

```
        File file:目的地是一个文件
    构造方法的作用：
        1.创建一个FileOutputStream对象
        2.会根据构造方法中传递的文件/文件路径，创建一个空的文件夹
        3.会把FileOutputStream对象指向创建好的文件
    写入数据的原理：（内存--》硬盘）
        java程序--》JVM（java虚拟机）--》os（操作系统）--》os调用写数据的方法--》把数据写入到文件中
    使用步骤：
        1.创建一个FileOutputStream对象，构造方法中传递写入数据的目的地
        2.调用FileOutputStream对象中的方法，把数据写入到文件中
        3.释放资源
*/
public class PackDemo03 {
    public static void main(String[] args) throws IOException {
        //FileOutputStream fos=new FileOutputStream("E:\\java\\study\\src\\Demo03\\Demo002\\a.txt");
        //fos.write(89);
        //写数据的时候实际写的二进制
        //fos.close();
        FileOutputStream fos=new FileOutputStream(new File("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\a.txt"));
        /*fos.write(49);
        fos.write(48);
        fos.write(48);*/
        /*
        一次写多个字节：
            如果写的第一个字节为正数（0-127），那么显示的时候就会查询Ascii表
            如果写的第一个字节为负数，那么第一个字节会和第二个字节，两个字节组成一个中文显示，查询系统默认码表（GBK）

        */
        byte[]by={-65,66,-67,68};
        fos.write(by);
        fos.write(by,0,2);
        /*写入字符串的方法*/
        String s="你好";
        byte[] bytes = s.getBytes();
        fos.write(bytes);
        fos.close();
    }
}
```

换行&续写

```
package Demo03.Demo002;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/*
追加&续写
FileOutputStream(File file, boolean append) 创建文件输出流以写入由指定的 File对象表示的文件。
FileOutputStream(String name, boolean append) 创建文件输出流以指定的名称写入文件。
```

```
*/
public class PackDemo04 {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos=new
FileOutputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\a.txt",true
);
        fos.write("你好".getBytes(StandardCharsets.UTF_8));
        fos.write("\r\n".getBytes(StandardCharsets.UTF_8));
        fos.close();
        /*
        换行符
        Windows: \r\n
        Linux: /n
        mac: /r
        */
    }
}
```

InputStream

```
package Demo03.Demo002;

import java.io.FileInputStream;
import java.io.IOException;

/*
java.io.InputStream:字节输入流
FileInputStream：文件字节输入流
作用：把硬盘文件中的数据，读取到内存中使用
构造方法：
   FileInputStream(String name)
   FileInputStream(File file)
构造方法的作用：
   1.会创建一个FileInputStream对象
   2.会把FileInputStream对象指定构造方法中要读取的文件
使用步骤：
   1.创建FileInputStream对象，构造方法中要绑定数据源
   2.使用read方法，读取文件
   3.释放资源
*/
public class PackDemo05 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis=new
FileInputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\a.txt");
        /*int len = fis.read();
        System.out.println(len);
        int len1 = fis.read();
        System.out.println(len1);
        int len2 = fis.read();
        System.out.println(len2);*/
        //以上内容重复，可以使用循环
        int len=0;
        while(len!=-1){
            System.out.println(len);
            len=fis.read();
        }
```

```
        fis.close();
    }
}
```

读取多个字节

```
package Demo03.Demo002;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Arrays;

/*
字节输入流一次读取多个字节的方法：
   int read(byte[] b)从输入流中读取一定数量的字节，将器存在缓冲区数组中
   明确两件事：
      1.方法的阐述byte[]的作用
        起到缓冲作用，存储到每次读取到的多个字节
        数组的长度一般定义为1024或其整数倍
      2.方法的返回值
        每次读取的长度
*/
public class PackDemo06 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis=new
FileInputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\a.txt");
        byte[]bytes=new byte[1024];
        int len = fis.read(bytes);
        /*System.out.println(len);
        //转换为String类型，
        System.out.println(new String(bytes));
        System.out.println(Arrays.toString(bytes));*/
        while(len!=-1){
            System.out.println(new String(bytes,0,len));
            //只读有效的
            len=fis.read(bytes);
        }
        fis.close();
    }
}
```

文件复制

```
package Demo03.Demo002;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/*
文件复制
明确：
   数据源
   目的地
步骤：
   1.创建字节输入流对象，构造方法中绑定数据源
```

```
    2.创建字节输出流对象，构造方法中写入目的地
    3.使用字节输入流对象中的read读取文件
    4.使用字节输出流中的write
    5.释放
*/
public class PackDemo07 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis=new
FileInputStream("E:\\java\\study\\src\\Demo03\\Demo002\\02.jpg");
        FileOutputStream fos=new
FileOutputStream("E:\\java\\study\\src\\Demo03\\02.jpg");
        //一次读一个字节，写一个字节
        /*int len=0;
        while((len=fis.read())!=-1){
            fos.write(len);
        }*/
        byte[]bytes=new byte[1024];
        int len=0;
        while((len=fis.read(bytes))!=-1){
            fos.write(bytes,0,len);
        }
        //先关闭写的流，在关闭读的流
        fos.close();
        fis.close();
    }
}
```

IO字符流

Reader

```
package Demo03.Demo003;

import java.io.FileReader;
import java.io.IOException;

/*
java.io.Reader:字符输入流，是字符输入流最顶层的父类，定义了一些成员方法
FileReader(File file)
FileReader(String fileName)
参数：数据源
使用步骤：
        1.创建FileReader对象，构造方法中绑定数据源
        2.使用FileReader对象中的read读取数据源
        3.释放资源
*/
public class PackDemo01 {
    public static void main(String[] args) throws IOException {
        FileReader fr=new
FileReader("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\a.txt");
        /*int len=0;
        while((len=fr.read())!=-1){
            System.out.println((char)len);
        }*/
        char[]cs=new char[1024];//存储读取到的多个字符
        int len=0;
        while((len=fr.read(cs))!=-1){
```

```
            System.out.println(new String(cs,0,len));
        }
        fr.close();
    }
}
```

Writer

```
package Demo03.Demo003;

import java.io.FileWriter;
import java.io.IOException;

/*
Writer
FileWriter:文件字符输出流
作用：将内存中的字符数据写入到文件中
使用步骤：
   1.创建一个FileWriter对象，构造方法中绑定要写入数据的目的地
   2.使用FileWriter中的方法Write，把数据写入到内存中
   3.使用FileWriter中的flush方法，把内存缓冲区中的数据，刷新到文件中
   4.释放资源（同时数据会被刷新到文件）
*/
public class PackDemo02 {
    public static void main(String[] args) throws IOException {
        FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\b.txt",true);
        fw.write(97);
        fw.flush();
        fw.close();
    }
}
```

flush&close

```
package Demo03.Demo003;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/*
flush:刷新缓冲区，流对象可以继续使用
close:先刷新缓冲区，然后通知系统释放资源
*/
public class PackDemo03 {
    public static void main(String[] args) throws IOException {
        FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\b.txt",true);
        fw.write(98);
        fw.flush();
        fw.write(97);
        fw.close();
    }
}
```

```
package Demo03.Demo003;

import java.io.FileWriter;
import java.io.IOException;

public class PackDemo05 {
    public static void main(String[] args) throws IOException {
        FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo003\\\\b.txt",true);
        //写字符数组
        char[]cs={'a','t','r','s','v','k','l'};
        fw.write(cs);
        fw.write(cs,1,3);
        fw.write("你真的可爱");
        fw.write("你真的可爱",1,4);
        fw.close();
    }
}
```

利用trycatch出理

```
package Demo03.Demo003;

import java.io.FileWriter;
import java.io.IOException;

public class PackDemo04 {
    public static void main(String[] args) {
        //提高变量fw的作用域
        FileWriter fw=null;//必须先复制
        try{
            fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo002\\\\b.txt",true);
            for (int i = 0; i < 10; i++) {
                fw.write("Hello,World"+i+"\r\n");
            }
        }catch(IOException e){
            System.out.println(e);
        }finally{
            //如果创建对象失败，有可能会抛出空指针异常
            if(fw!=null){
                try {
                    //该方法声明抛出了IOException
                    fw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

JDK新特性

```
package Demo03.Demo003;

import java.io.FileWriter;
import java.io.IOException;

/*
JDK7新特性
在try的后面可以增加一个()，在括号中可以定义流对象
那么这个流对象的作用域进try中有效
try中代码执行完毕，会自动把流对象释放，不用写finally
格式:
   try(){}
*/
public class PackDemo06 {
    public static void main(String[] args) {
        try(FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo003\\\\b.txt",true)){
            fw.write("koi");
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

```
package Demo03.Demo003;

import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;

/*
JDK9的新特性
try前面可以定义流对象
在try后边的()中可以直接引入流对象的名称（变量名）
在try代码执行完毕之后，流对象也可以释放掉，不用写finally
*/
public class PackDemo07 {
    public static void main(String[] args) throws IOException {
        FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo003\\\\b.txt",true);
        try(fw){
            fw.write("\r\n"+"ll");
        }catch(IOException e){
            System.out.println(e);
        }
        //流对象已被释放
    }
}
```

Properties

```
package Demo03.Demo004;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
import java.util.Set;

/*
java.util.Properties
Properties类表示一组持久的属性。 Properties可以保存到流中或从流中加载。
该集合是唯一一个和IO流相结合的集合
  可以使用store方法，将集合中的临时数据写入到文件
  可以使用load，将硬盘中的数据读取到文件
*/
public class PackDemo01 {
    public static void main (String[] args) throws IOException {
        show01();
        show02();
        show03();
    }

    private static void show03() throws IOException {
        /*
        可以使用Properties集合中的方法load，将硬盘中的文件，读取到集合中使用
        使用步骤：
           1.使用Properties集合对象
           2.使用Properties集合对象中的方法load读取保存键值对的文件
           3.遍历对象
        注意：
           1.存储键值对的文件中，键与值默认的连接符号可以使用=，空格（其他符号）
           2.存储键值对的文件中，可以使用#注释，被注释的不会被读取
           3.存储键值对的文件，默认都是字符串，不用再加引号
        */
        Properties pros=new Properties();
        pros.load(new
FileReader("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\b.txt"));
        Set<String> strings = pros.stringPropertyNames();
        for(String s:strings){
            String property = pros.getProperty(s);
            System.out.println(s+"="+property);
        }
    }

    private static void show02() throws IOException {
        /*
        可以使用store方法，将集合中的临时数据，持久化到硬盘中
        void store(OutputStream out,String comments)
        void store(Writer writer,String comments)
        参数：
            OutputStream out:字节输出流，不能写入中文
            Writer writer：字符输出流，可以写入中文
            String comments：注释，用来解释说明保存的文件是做什么用的
                             不能使用中文，默认Unicode编码
```

```
        */
        /*
        使用步骤：
            1.创建Properties对象，添加数据
            2.创建字节输出流/字符输出流对象，构造方法中绑定目的地
            3.使用store方法，将临时数据存储到硬盘
            4.释放资源
        */
        Properties pros=new Properties();
        pros.setProperty("koi","18");
        pros.setProperty("misaka","15");
        pros.setProperty("kuruko","15");
        FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\b.txt",true);
        pros.store(fw,"");
        fw.close();
    }

    private static void show01() {
        /*
        使用Properties集合存储数据，遍历取出数据
        其为一个双列集合，key和value默认都是字符串
        */
        Properties pros=new Properties();
        pros.setProperty("koi","18");
        pros.setProperty("misaka","15");
        pros.setProperty("kuruko","15");
        Set<String> strings = pros.stringPropertyNames();
        for(String s:strings){
            String property = pros.getProperty(s);
            System.out.println(property);
        }
    }
}
```

缓冲流

字节缓冲输出流

```
package Demo03.Demo004;

import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/*
java.io.BufferedOutputStream
构造方法：
     参数：
        OutputStream out:字节输出流，缓冲流会给该输出流增加一个缓冲区，提高写入效率
        int size:指定缓冲流大小
使用步骤：
   1.创建FileOutputStream对象，构造方法中绑定要输出的目的地
   2.创建BufferedOutputStream对象，构造方法中传递FileOutputStream对象
   3.使用write方法，将数据写入缓冲区
   4.使用flush方法，将缓冲区数据刷新到文件
```

```
   5.释放资源
*/
public class PackDemo02 {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos=new
FileOutputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\b.txt");
        BufferedOutputStream bos=new BufferedOutputStream(fos);
        bos.write("给爷写".getBytes(StandardCharsets.UTF_8));
        bos.flush();
        bos.close();
    }
}
```

字节缓冲输入流

```
package Demo03.Demo004;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

/*
使用步骤:
   1.创建FileInputStream对象，构造方法中绑定数据源
   2.创建BufferedInputStream对象，构造方法中传递FileInputStream对象，提高效率
   3.使用read方法，读取文件
   4.释放资源
*/
public class PackDemo03 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis=new
FileInputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\b.txt");
        BufferedInputStream bis=new BufferedInputStream(fis);
        byte[]bytes=new byte[1024];
        int len=0;
        while((len=bis.read(bytes))!=-1){
            System.out.println(new String(bytes,0,len));
        }
        bis.close();
    }
}
```

文件复制

```
package Demo03.Demo004;

import java.io.*;

public class PackDemo04 {
    public static void main(String[] args) throws IOException {
        long l = System.currentTimeMillis();
        FileInputStream fis=new
FileInputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\02.jpg");
        FileOutputStream fos=new
FileOutputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\01.jpg");
        BufferedInputStream bis=new BufferedInputStream(fis);
```

```
        BufferedOutputStream bos=new BufferedOutputStream(fos);
        byte[]bytes=new byte[1024];
        int len=0;
        while((len=bis.read(bytes))!=-1){
            bos.write(bytes,0,len);
        }
        bos.close();
        bis.close();
        long l1 = System.currentTimeMillis();
        System.out.println(l1-l);
    }
}
```

字符缓冲输出流&字符缓冲输出流

```
package Demo03.Demo004;

import java.io.*;

/*
BufferedWriter中有一个特有方法
  void newline() 写入一个换行符，可根据不同系统获得不同的换行符
BufferedReader中有一个特有方法
  String readLine() 读取一个文本行。读取一行数据
  读一行文字。 一行被视为由换行符（'\ n'），回车符（'\ r'）中的任何一个或随后的换行符终止
  包含行的内容的字符串，不包括任何行终止字符，如果已达到流的末尾，则为null
*/
public class PackDemo05 {
    public static void main(String[] args) throws IOException {
        FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\b.txt",true);
        FileReader fr=new
FileReader("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\b.txt");
        BufferedReader br=new BufferedReader(fr);
        BufferedWriter bw=new BufferedWriter(fw);
        bw.newLine();
        bw.write("koi suki misaka");
        bw.flush();
        bw.close();
        char[]chars=new char[1024];
        int len=0;
        String s="";
        while((s=br.readLine())!=null){
            System.out.println(s);
        }

        //System.out.println(s);
        /*while((len=br.read(chars))!=-1){
            System.out.println(new String(chars,0,len));
        }*/
        br.close();
    }
}
```

文件排序

```
package Demo03.Demo004;

import java.io.*;
import java.util.HashMap;

/*
练习：对文本内容进行排序
分析：
    1.创建一个HashMap集合对象，key存储序号，value存储文本
    2.创建字符缓冲输入流，构造方法中绑定字符输入流
    3.创建字符缓冲输出流，构造方法中绑定字符输出流
    4.使用readline读取
    5.对文本进行切割，获取行中序号和文本内容
    6.把切割好的序号和文本的内容保存到HaahMap
    7.遍历HashMap，获取每一个键值对
    8.把每一个键值对，拼接为文本行
    9.把拼接好的文本，使用write，写入到文件
    10.释放资源
*/
public class PackDemo07 {
    public static void main(String[] args) throws IOException {
        HashMap<String,String>hashMap=new HashMap<>();
        FileReader fr=new
FileReader("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\b.txt");
        FileWriter fw=new
FileWriter("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo004\\\\a.txt");
        BufferedReader br=new BufferedReader(fr);
        BufferedWriter bw=new BufferedWriter(fw);
        String line;
        while((line=br.readLine())!=null){
            String[] arr = line.split("\\.");
            hashMap.put(arr[0],arr[1]);
        }
        for(String key:hashMap.keySet()){
            String value=hashMap.get(key);
            line=key+"."+value;
            bw.write(line);
            bw.newLine();
        }
        bw.close();
        br.close();
    }
}
```

转换流

输出流

```
package Demo03.Demo005;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
```

```
import java.nio.charset.StandardCharsets;

/*
FileReader可以读取IDE默认编码格式（UTF-8）的文件
FileReader读取系统默认编码（GBK）会产生乱码
java.io.OutputStreamWriter
OutputStreamWriter:是字符流通向字节流的桥梁，可使用指定的2charset将要写入流中的字符编码成字节
使用步骤：
   1.创建OutputStreamWriter对象，构造方法中传递字节输出流和指定的编码表名称
   2.使用OutputStreamWriter对象中的方法write，把字符转换为字节存储缓冲区中（编码）
   3.使用OutputStreamWrite中的方法flush
   4.释放资源
*/
public class PackDemo02 {
    public static void main(String[] args) throws IOException {
        write_utf_8();
    }

    private static void write_utf_8() throws IOException {
        OutputStreamWriter osw=new OutputStreamWriter(new
FileOutputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo005\\\\a.txt"),
StandardCharsets.UTF_8);
        osw.write("你好");
        osw.flush();
        osw.close();
    }
}
```

输入流

```
package Demo03.Demo005;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
/*
使用步骤：
   1.创建InputStreamReader对象，构造方法中传递字节输入流和指定的编码表名称
   2.使用InputStreamReader对象中的方法read读取文件
   3.释放资源
注意事项：
   构造方法中指定的编码表名称要和文件的编码相同，否则会发生乱码
*/
public class PackDemo03 {
    public static void main(String[] args) throws IOException{
        reader_GBK();
    }

    private static void reader_GBK() throws IOException {
        InputStreamReader isr=new InputStreamReader(new
FileInputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo005\\\\GBK.txt"),"G
BK");
        int len=0;
        while((len=isr.read())!=-1){
            System.out.println((char)len);
```

```
        }
        isr.close();
    }
}
```

案例：转码

```
package Demo03.Demo005;

import java.io.*;

/*
练习：转换文件编码
分析：
   1.创建InputStreamReader对象，构造方法中传递指定的字节输入流和指定的编码表名称
   2.创建OutputStreamWriter对象，构造方法中传递字节输出流和指定的编码表名称UTF-8
   3.使用read读取文件
   4.使用write，把读取的数据写入到文件中
   5.释放资源
*/
public class PackDemo04 {
    public static void main(String[] args) throws IOException {
        InputStreamReader isr=new InputStreamReader(new
FileInputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo005\\\\GBK.txt"),"G
BK");
        OutputStreamWriter osw=new OutputStreamWriter(new
FileOutputStream("E:\\\\java\\\\study\\\\src\\\\Demo03\\\\Demo005\\\\UTF-
8.txt"),"utf-8");
        char[]chars=new char[1024];
        int len=0;
        while((len=isr.read(chars))!=-1){
            osw.write(chars,0,len);
            osw.flush();
        }
        osw.close();
        isr.close();
    }
}
```

序列化流

ObjectOutputStream

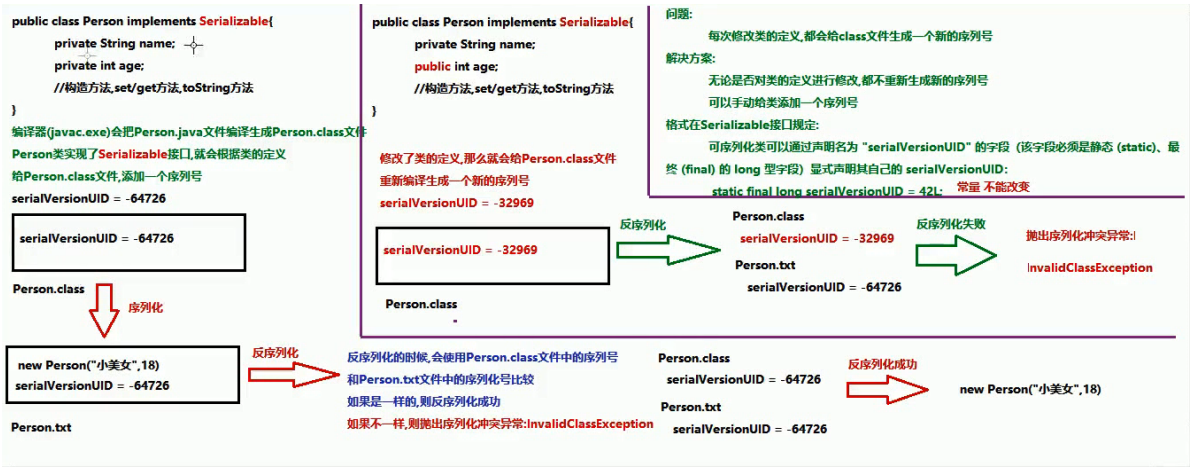
```
package Demo03.Demo005;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

/*
java.io.ObjectOutputStream
ObjectOutputStream:对象的序列化流
作用：把对象以流的方式写入到文件中保存
使用步骤：
   1.创建ObjectOutputStream对象，构造方法中传递字节输出流
   2.使用ObjectOutputStream对象中的方法writeObject，把对象写入到文件中
```

```
序列化的条件：
   1.该类必须实现`java.io.Serializable 接口，Serializable 是一个标记接口，不实现此接口的类将不会使任何状态序列化或反序列化，
   会抛出NotSerializableException。
   2.该类的所有属性必须是可序列化的。如果有一个属性不需要可序列化的，则该属性必须注明是瞬态的，使用transient关键字修饰。
*/
public class PackDemo05 {
    public static void main(String[] args) throws IOException {
        ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("c.txt"));
        oos.writeObject(new person("koi",18));
        oos.close();
    }
}
```

ObjectInputStream

```
package Demo03.Demo005;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

/*
ObjectInputStream
使用步骤：
   1.创建ObjectInputStream对象，构造方法中传递字节输入流
   2.使用ObjectInputStream对象中的方法readObject读取保存对象的文件
   3.释放资源
   4.使用读取出来的对象
 static关键字：静态关键字
        静态优先于非静态加载到内存中（静态优先于对象进入到内存中）
        被static关键字修饰的成员变量不能被序列化的，序列化的都是对象
 transient关键字：瞬态关键字
        被其修饰的成员变量，不能被序列化
*/
public class PackDemo06 {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        ObjectInputStream ois=new ObjectInputStream(new FileInputStream("c.txt"));
        Object o = ois.readObject();
        System.out.println(o);
        ois.close();
    }
}
```

InvalidClassException



23333 词

```
private static final long serialVersionUID=42L;
```

序列化集合

```
package Demo03.Demo005;

import java.io.*;
import java.util.ArrayList;

/*
练习：序列化集合
    当我们想在文件中保存多个对象时
    可以把多个对象存储在一个集合中
    对集合进行序列化和反序列化
分析：
    1.定义一个存储person对象的ArrayList结合
    2.存储person对象
    3.创建序列化流
    4.使用writeObject方法，对集合进行序列化
    5.创建一个反序列化ObjectInputStream对象
    6.使用ObjectInputStream对象中的readObject方法
    7.把Object类型的集合转换为ArrayList类型
    8.遍历集合
    9.释放资源
*/
public class PackDemo07 {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        ArrayList<person>list=new ArrayList<>();
        list.add(new person("koi",18));
        list.add(new person("misaka",15));
        list.add(new person("kuruko",14));
        ObjectOutputStream oos=new ObjectOutputStream(new
FileOutputStream("d.txt"));
        oos.writeObject(list);
        ObjectInputStream ois=new ObjectInputStream(new
FileInputStream("d.txt"));
        Object o = ois.readObject();
        ArrayList<person>list2=(ArrayList<person>)o;
        for(person p:list2){
```

```
            System.out.println(p);
        }
        oos.close();
        ois.close();
    }
}
```

打印流

```
package Demo03.Demo005;

import java.io.FileNotFoundException;
import java.io.PrintStream;

/*
java.io.PrintStream:打印流
      为其他输出流添加了功能，使他们能够方便的打印各种数据值表示形式
PrintStream特点：
      1.只负责数据的输出，不负责数据的读取
      2.与其他输出流不同，永远不会抛出IOException
      3.有特有的方法，print，println
构造方法：
      PrintStream(File file) 使用指定的文件创建一个新的打印流，而不需要自动换行
      PrintStream(String fileName) 使用指定的文件名创建新的打印流，无需自动换行。
注意：
     如果使用继承自父类的write方法，那么查看数据的时候会查询编码表
     如果使用自己特有的方法，写的数据原样输出
*/
public class PackDemo08 {
    public static void main(String[] args) throws FileNotFoundException {
        PrintStream ps=new PrintStream("print.txt");
        ps.write(97);
        ps.println(97);
        ps.println(9.08);
        ps.println(true);
        ps.println("sa");
        ps.close();
        /*
        可以改变输出语句的目的地（打印流的流向）
        输出语句，默认在控制台输出
        使用System.setOut方法改变输出语句的目的地改为参数传递的打印流的目的地
        */
        print01();
    }

    private static void print01() throws FileNotFoundException {
        System.out.println("我是在控制台输出");
        PrintStream ps=new PrintStream("prints.txt");
        System.setOut(ps);
        System.out.println("我在设置的位置打印");
        ps.close();
    }
}
```

JDK8新特性

常用函数接口

使用

```
package Demo05.Demo001;
/*
函数式接口的使用：一般可以作为方法的参数和返回值类型
*/
public class PackDemo01 {
    //定义一个方法，参数使用函数式接口
    public static void show(Function1 myfunction){
        myfunction.method();
    }

    public static void main(String[] args) {
        //调用show方法，方法的参数是一个接口，可以传递接口的实现类对象
        show(new MyFunctionimp());
        //可以传递接口的匿名内部类
        show(new Function1() {
            @Override
            public void method() {
                System.out.println("使用匿名内部类重写接口的抽象方法");
            }
        });
        //调用show方法，可以传递Lambda表达式
        show(()->{
            System.out.println("使用Lambda表达式重写接口中的抽象方法");
        });
    }
}
```

```
package Demo05.Demo001;
/*
函数式接口：有且只有一个抽象方法的接口，称之为函数式接口
可以有其他方法（默认，静态，私有）
*/
@FunctionalInterface
//可以检测接口是否为函数式接口
public interface Function1 {
    public abstract void method();
}
```

Lambda优化日志

```
package Demo05.Demo001;
/*
使用Lambda优化日志案例
Lambda特点：延迟加载
Lambda使用前提：必须存在函数式接口
*/
public class PackDemo02 {
    //定义一个显示日志的方法，方法的参数传递日志的等级
    public static void showlog(int level,Function2 function2){
        //对日志的等级进行判断
        if(level==1){
```

```
            System.out.println(function2.buildMessage());
        }
    }

    public static void main(String[] args) {
        String msg1="Hello";
        String msg2="World";
        String msg3="Java";
        showlog(1,()->{
            return msg1+msg2+msg3;
        });
        /*
        使用Lambda表达式，仅仅是把参数传递到showlog方法中
        只有满足条件，才会调用接口中的方法
        所以不会存在性能的浪费
         */
    }
}
```

Stream流式思想概述

获取Stream流的方式

方法引用

反射

概述

框架设计的灵魂

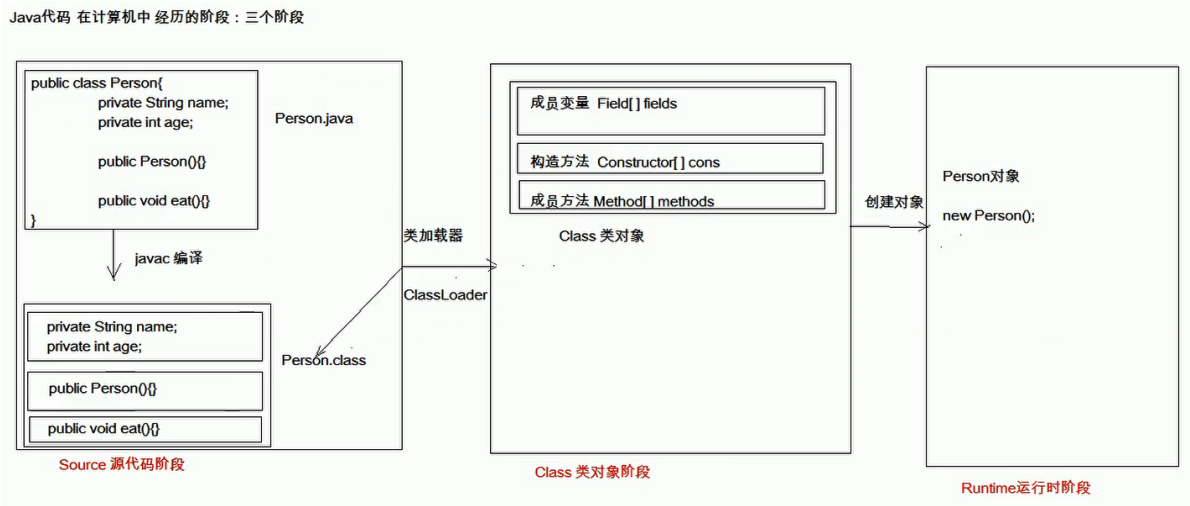
框架：半成品软件。可以在框架上进行软件开发，简化代码

反射：将类的各个组成部分封装为其他对象，这就是反射机制

好处：

1.在程序运行过程中，操作这些对象

2.可以解耦，提高程序的可扩展性



获取

```
package Demo04.Demo001;

import Demo04.person;

/*
获取Class对象的三种方法
1.Class.forName("全类名"):将字节码文件加载进内存，返回Class对象
  多用于配置文件
2.类名.Class：通过类名的属性class获取
  多用于参数的传递
3.对象.getClass():getClass()方法在Object类中定义着
  多用于对象获取字节码文件
同一个字节码文件，在一次程序运行过程中，只会被加载一次
*/
public class PackDemo01 {
    public static void main(String[] args) throws ClassNotFoundException {
        Class<?> cls = Class.forName("Demo04.person");
        System.out.println(cls);
        Class<person> cls1 = person.class;
        System.out.println(cls1);
        person p = new person();
        Class<? extends person> cls2 = p.getClass();
        System.out.println(cls2);
    }
}
```

功能

获取成员变量

```
package Demo04.Demo001;

import Demo04.person;

import java.lang.reflect.Field;

/*
功能：
   1.获取成员变量
     getField(String name) 返回一个 Field对象，它反映此表示的类或接口的指定公共成员字段 类对象
     getFields() 返回包含一个数组 Field对象反射由此表示的类或接口的所有可访问的公共字段 类对象。
     getDeclaredField(String name) 返回一个 Field对象，它反映此表示的类或接口的指定已声明字段 类对象。
     getDeclaredFields() 返回的数组 Field对象反映此表示的类或接口声明的所有字段 类对象。
   2.获取构造方法
     getConstructor(类<?>... parameterTypes) 返回一个 Constructor对象，该对象反映Constructor对象表示的类的指定的公共 类函数
     getConstructors() 返回包含一个数组 Constructor对象反射由此表示的类的所有公共构造 类对象。
   3.获取成员方法
   4.获取类名
*/
```

```
public class PackDemo02 {
    public static void main(String[] args) throws Exception,
IllegalAccessException {
        Class<person> cls0 = person.class;
        Field[] fields = cls0.getFields();
        for(Field f:fields){
            System.out.println(f);
        }
        Field app = cls0.getField("app");

        //获取成员变量的值
        person person = new person();
        Object o = app.get(person);
        System.out.println(o);
        app.set(person,"beautiful");
        System.out.println(person);

        //可以获取私有的
        Field[] declaredFields = cls0.getDeclaredFields();
        for(Field f:declaredFields){
            System.out.println(f);
        }
        //忽略访问权限修饰符的安全检查
        Field declaredField = cls0.getDeclaredField("name");
        declaredField.setAccessible(true);
        Object o1 = declaredField.get(person);
        System.out.println(o1);
    }
}
```

获取构造方法

```
package Demo04.Demo001;

import Demo04.person;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class PackDemo03 {
    public static void main(String[] args) throws NoSuchMethodException,
IllegalAccessException, InvocationTargetException, InstantiationException {
        Class<person> personClass = person.class;
        Constructor<person> constructor =
personClass.getConstructor(int.class,String.class);
        System.out.println(constructor);
        person koi = constructor.newInstance(16, "koi");
        System.out.println(koi);
        //如果使用空参数构造方法，操作可以简化：Class对象的newInstance方法
    }
}
```

获取成员方法

```
package Demo04.Demo001;

import Demo04.person;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class PackDemo04 {
    public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
        Class<person> personClass = person.class;
        Method method = personClass.getMethod("eat",String.class);
        person p=new person();
        method.invoke(p,"misaka");
        Method[] methods = personClass.getMethods();
        for(Method m:methods){
            System.out.println(m);
            String name = m.getName();
            System.out.println(name);
        }
    }
}
```

案例

```
package Demo04.Demo001;

import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.net.URL;
import java.util.Properties;

/*
        不能改变任何的代码
        可以创建任意类得对象可以执行任意方法
          1.配置文件
          2.反射
        步骤:
           1.将需要创建的对象的全类名和需要执行的方法定义在配置文件中
           2.在程序中加载读取配置文件
           3.使用反射技术来加载文件进内存
           4.创建对象
           5.执行方法
        */
public class PackDemo05 {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, IllegalAccessException, InstantiationException,
InvocationTargetException, NoSuchMethodException {
        //1.1创建一个Properties对象
        Properties pro=new Properties();
        //1.2加载配置文件，转换为一个集合
        //1.2.1获取class目录下的配置文件
```

```
        ClassLoader classLoader = PackDemo05.class.getClassLoader();
        InputStream is = classLoader.getResourceAsStream("pro.properties");
        pro.load(is);
        //获取配置文件中定义的数据
        String className = pro.getProperty("className");
        String methodName = pro.getProperty("methodName");
        //加载该类进内存
        Class cls = Class.forName(className);
        //创建对象
        Object obj = cls.newInstance();
        //获取方法对象
        Method method = cls.getMethod(methodName);
        //执行方法
        method.invoke(obj);
    }
}
```

注解

概念

说明程序给计算机看的

ava 注解（Annotation）又称 Java 标注，是 JDK5.0 引入的一种注释机制。

Java 语言中的类、方法、变量、参数和包等都可以被标注。和 Javadoc 不同，Java 标注可以通过反射获取标注内容。在编译器生成类文件时，标注可以被嵌入到字节码中。Java 虚拟机可以保留标注内容，在运行时可以获取到标注内容 。 当然它也支持自定义 Java 标注。

@+注解名称

作用分类：

1.编写文档：通过文档里标识的注解生成文档【生成doc文档】

2.代码检查：通过代码里标识的注解对代码进行分析【使用反射】

3.编译检查：通过代码里标识的注解让编译器能够实现基本的编译检查【override】

内置注解

@Override：检测被该注解标注的方法是否是继承自父类（接口）的

@Deprecated：该注解标注的内容，表示已过时

@SuppressWarnings：压制警告

```
package Demo04.Demo003;
@SuppressWarnings("all")
public class PackDemo01 {
    public static void main(String[] args) {
        show01();
    }
    @Deprecated
    //表示已过时
    private static void show01() {
    }
}
```

自定义注解

格式：元注解 public @interface 注解名称{}

`public interface myan extends java.lang.annotation.Annotation {}`

本质：注解本质上就是一个接口

属性

接口中定义的成员方法

1.属性的返回值类型

- 基本数据类型
- String
- 枚举
- 注解
- 以上类型的数组

```
package Demo04.Demo003;

public @interface myan {
    String show1();
    int show2()default(15);
    person per();
    Myanno2 myann0();
    String[] strs();
}
```

注意

必须赋值

若只有一个属性，且名字叫value，value可省略

在定义属性时，可使用default关键字给属性默认初始化

数组赋值时，值可以使用{}包裹，若只有一个，{}可以省略

`@myan(name="koi",age=12)`

元注解

描述注解的注解

@Target：描述注解能够作用的位置`ElementType.TYPE`

- TYPE：表示可以作用于类上
- METHOD：作用于方法上
- FIELD：作用于成员变量上

@Retention：描述注解被保留的阶段`RetentionPolicy.RUNTIME`

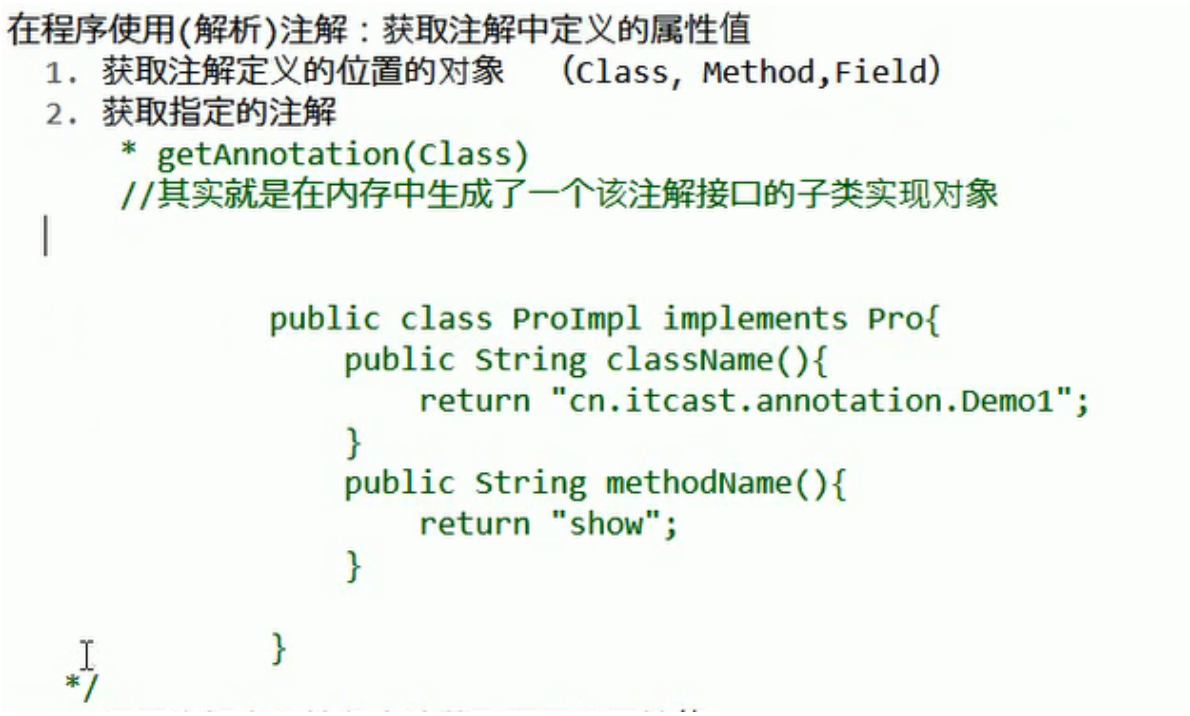
@Documented：描述注解是否被抽象到api文档中

@Inherited：描述注解是否被子类继承

```java
package Demo04.Demo003;

import java.lang.annotation.*;

/*
* @Target：描述注解能够作用的位置

* @Retention：描述注解被保留的阶段

* @Documented：描述注解是否被抽象到api文档中

* @Inherited：描述注解是否被子类继承
*
* */
@Target(value={ElementType.TYPE})//表示该注解只能作用于类上
@Retention(RetentionPolicy.RUNTIME)
@Documented//表示会被抽取到API文档中
@Inherited
public @interface Myanno3 {
}
```

解析注释

```java
package Demo04.Demo003;
@pro(className = "Demo04.Demo003",methodName = "show01")
public class PackDemo02 {
    public static void main(String[] args) {
        //解析注解
        //1.1获取该类的字节码文件
        Class<PackDemo02> packDemo02Class = PackDemo02.class;
        //获取注解对象
        pro annotation = packDemo02Class.getAnnotation(pro.class);
        //其实就是在内存中生成了一个该注解接口的实现类对象
        //调用注解对象中定义的抽象方法，获取返回值
        String className = annotation.className();
        String methodName = annotation.methodName();
        //后续同反射代码
    }
}
```



小结

1.以后大多时候，我们会使用注解而不是，而不是自定义注解

2.编译器使用及解析程序用

3.注解不是程序的一部分

```
package Demo04.Demo003;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/*
* 当主方法执行后，会自动执行被检测的所有方法，判断方法是否有异常
* */
public class TestCheck {
    public static void main(String[] args) throws IOException {
        int num=0;
        BufferedWriter bw = new BufferedWriter(new FileWriter("bug.txt"));

        //1.创建对象
        Calculator c=new Calculator();
        //2.获取字节码文件
        Class<? extends Calculator> cls = c.getClass();
        //获取所有方法
        Method[] methods = cls.getMethods();
        for(Method m:methods){
            //判断是否有注解
            if(m.isAnnotationPresent(check.class)){
                //有就执行
                try {
                    m.invoke(c);
                } catch (Exception e) {
```

```
                    //捕获异常
                    //记录到文件中
                    num++;
                    bw.write(m.getName()+"出现异常");
                    bw.newLine();
                    bw.write("异常名称"+e.getCause().getClass().getSimpleName());
                    bw.newLine();
                    bw.write("异常原因"+e.getCause().getMessage());
                    bw.newLine();
                    bw.write("---------------");
                }

            }
        }
        bw.write("本次共出现"+num+"次异常");
        bw.flush();
        bw.close();

    }
}
```

编程

概述

C/S结构：全称为Client/Server结构，是指客户端和服务器结构。常见程序有ＱＱ、迅雷等软件。

B/S结构：全称为Browser/Server结构，是指浏览器和服务器结构。常见浏览器有谷歌、火狐等。

网络通信协议

- **网络通信协议：**通过计算机网络可以使多台计算机实现连接，位于同一个网络中的计算机在进行连接和通信时需要遵守一定的规则，这就好比在道路中行驶的汽车一定要遵守交通规则一样。在计算机网络中，这些连接和通信的规则被称为网络通信协议，它对数据的传输格式、传输速率、传输步骤等做了统一规定，通信双方必须同时遵守才能完成数据交换。
- **TCP/IP协议：** 传输控制协议/因特网互联协议(Transmission Control Protocol/Internet Protocol)，是Internet最基本、最广泛的协议。它定义了计算机如何连入因特网，以及数据如何在它们之间传输的标准。它的内部包含一系列的用于处理数据通信的协议，并采用了4层的分层模型，每一层都呼叫它的下一层所提供的协议来完成自己的需求。

部分内容在相应文件

TCP

服务器端

```
package Demo02.Demo006;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
```

```
import java.nio.charset.StandardCharsets;

/*
TCP通信的服务器端：接受客户端发来的数据，给客户端回写数据
表示服务器的类：
    java.net.ServerSocket 这个类实现了服务器套接字
构造方法：
    ServerSocket(int port)创建绑定到指定端口的服务器套接字
服务器段必须明确一件事，必须知道是哪个客户端请求的服务器
所以可以使用accept方法获取到请求的客户端对象Socket
成员方法：
    Socket accept()侦听要连接到此套接字并接受它。
服务器的实现步骤：
    1.创建服务器ServerSocket对象和系统要指定的端口号
    2.使用ServerSocket中的accept方法，获取到请求的客户端对象Socket
    3.使用Socket对象中的方法getInputStream()获取网络字节输入流InputStream对象
    4.使用网络字节输入流InputStream对象中的方法read，读取服务器回写的数据
    5.使用Socket对象中的方法getOutputStream()获取网络字节输出流OutputStream对象
    6.使用网络字节输出流OutputStream对象中的方法write，给客户端回写数据
    7.释放资源
*/
public class Sever {
    public static void main(String[] args) throws IOException {
        ServerSocket server=new ServerSocket(8888);
        Socket socket = server.accept();
        InputStream is = socket.getInputStream();
        byte[]bytes=new byte[1024];
        int len=is.read(bytes);
        System.out.println(new String(bytes,0,len));
        OutputStream os = socket.getOutputStream();
        os.write("收到谢谢".getBytes(StandardCharsets.UTF_8));
        socket.close();
        server.close();
    }
}
```

客户端

```
package Demo02.Demo006;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

/*
TCP通信的客户端：向服务器发送请求，给服务器发送数据，读取服务器回写的数据
表示客户端的类：
    java.net.Socket:此类实现客户端套接字（也称为“套接字”）。 套接字是两台机器之间通讯的端点。
    套接字：包含了IP地址和端口号的网络单位
    构造方法：
    Socket(String host, int port)
        创建流套接字并将其连接到指定主机上的指定端口号。
    String host：表示服务器主机的名称/服务器的IP地址
    int port：服务器的端口号
    成员方法：
```

```
        OutputStream getOutputStream() 返回此套接字的输出流。
        InputStream getInputStream() 返回此套接字的输入流。
        void close() 关闭此套接字。
    步骤：
        1.创建一个客户端对象Socket，构造方法绑定服务器的IP地址和端口号
        2.使用Socket对象中的方法getOutputStream()获取网络字节输出流OutputStream对象
        3.使用网络字节输出流OutputStream对象中的方法write，给服务器发送数据
        4.使用Socket对象中的方法getInputStream()获取网络字节输入流InputStream对象
        5.使用网络字节输入流InputStream对象中的方法read，读取服务器回写的数据
        6.释放资源Socket
    注意：
        1.客户端和服务器进行交互，必须使用Socket中提供的网络流，不能使用自己创建的流对象
        2.当我们创建客户端对象Socket时，就会去请求服务器和服务器经过3次握手建立连接通路
          如果服务器未启动，就会抛出异常
          如果服务器已经启动，就可以正常交互
*/
public class PackDemo01 {
    public static void main(String[] args) throws IOException {
        Socket socket=new Socket("127.0.0.1",8888);
        OutputStream os=socket.getOutputStream();
        os.write("你好服务器".getBytes(StandardCharsets.UTF_8));
        InputStream is = socket.getInputStream();
        byte[]bytes=new byte[1024];
        int len=is.read(bytes);
        System.out.println(new String(bytes,0,len));
        socket.close();
    }
}
```