关系运算

这里其实和一些地方看到的东西略有不同，值得一提。

判断是否相等的==与!=的优先级是要比其他优先级来的低，而连续的关系运算符是从左到右进行运算的。举几个例子以助理解。

```
5>3==6>4;
6>5>4;//无法运行。因为true无法与4运行
a==b==true;//无法运行。因为true无法与b比较。
a==b>false;//注意，与某些语言不一样，false不等价于NULL或0。
(a==b)>false;
```

哈哈 sbC#

一般来说如果你非得要去比较两个浮点数的大小，一般需要采用以下策略：

```
Math.abs(f1-f2) < 0.000001//尽量小，不行直接上1e-10
```

因为浮点数运算本身存在精度问题。

循环控制

标号的使用可以使得循环控制简洁很多。

可以给语句块加标号赋予它们名称，标号位于语句之前。标号只能被continue和break引用。格式如下：

`label:statement`

语句前只允许加一个标号，标号后面不能跟大括号。通过用break后加标号对处于标号中的语句进行控制。往往标号后是for.while.do-while等循环。

通过用标号，我们可以对外层循环进行控制

下面是用continue控制标号：

```
public class Label {

    public static void main(String[] args) {
        System.out.println("i j");
        search:
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 50; j++) {
                    if (j == 3)
                        continue search;
                    System.out.println(i+" "+j);
                }
            }
    }
}
```

标号语句必须紧接在循环的头部。标号语句不能用在非循环语句的前面。

for用法的小贴士

在c#中存在一种用法为

```
foreach(int i in array_1):
{
    //whatever
}
```

在java中有类似用法：

```
for(int k : array_1):
{
    //whatever
}
```

三种print的区别

java中有三种方式向控制台输入信息，分别是print，printf，println，现在简单介绍一下他们之间的异同。

- `printf`主要是继承了C语言的printf的一些特性，可以进行格式化输出
- `print`就是一般的标准输出，但是不换行
- `println`和`print`基本没什么差别，就是最后会换行

细说printf

目前支持的格式有：

format	function
%c	单个字符
%d	十进制整数
%f	十进制浮点数
%o	八进制数
%s	字符串
%u	无符号十进制数
%x	十六进制数

format	function
%%	输出百分号%

printf的格式控制的完整格式：
% - 0 m.n l或h 格式字符
下面对组成格式说明的各项加以说明：
①%：表示格式说明的起始符号，不可缺少。
②-：有-表示左对齐输出，如省略表示右对齐输出。
③0：有0表示指定空位填0,如省略表示指定空位不填。
④m.n：m指域宽，即对应的输出项在输出设备上所占的字符数。N指精度。用于说明输出的实型数的小数位数。为指定n时，隐含的精度为n=6位。
⑤l或h:l对整型指long型，对实型指double型。h用于将整型的格式字符修正为short型。

格式字符
格式字符用以指定输出项的数据类型和输出格式。
①d格式：用来输出十进制整数。有以下几种用法：
%d：按整型数据的实际长度输出。
%md：m为指定的输出字段的宽度。如果数据的位数小于m，则左端补以空格，若大于m，则按实际位数输出。
%ld：输出长整型数据。
②o格式：以无符号八进制形式输出整数。对长整型可以用"%lo"格式输出。同样也可以指定字段宽度用“%mo”格式输出。
例：

```
main() {
  int a = -1;
  printf("%d, %o", a, a);
}
```

运行结果：-1,177777
程序解析：-1在内存单元中（以补码形式存放）为(1111111111111111)2，转换为八进制数为(177777)8。
③x格式：以无符号十六进制形式输出整数。对长整型可以用"%lx"格式输出。同样也可以指定字段宽度用"%mx"格式输出。
④u格式：以无符号十进制形式输出整数。对长整型可以用"%lu"格式输出。同样也可以指定字段宽度用“%mu”格式输出。
⑤c格式：输出一个字符。
⑥s格式：用来输出一个串。有几中用法
%s：例如:printf("%s", "CHINA")输出"CHINA"字符串（不包括双引号）。
%ms：输出的字符串占m列，如字符串本身长度大于m，则突破获m的限制,将字符串全部输出。若串长小于m，则左补空格。
%-ms：如果串长小于m，则在m列范围内，字符串向左靠，右补空格。
%m.ns：输出占m列，但只取字符串中左端n个字符。这n个字符输出在m列的右侧，左补空格。
%-m.ns：其中m、n含义同上，n个字符输出在m列范围的左侧，右补空格。如果n>m，则自动取n值，即保证n个字符正常输出。
⑦f格式：用来输出实数（包括单、双精度），以小数形式输出。有以下几种用法：
%f：不指定宽度，整数部分全部输出并输出6位小数。
%m.nf：输出共占m列，其中有n位小数，如数值宽度小于m左端补空格。
%-m.nf：输出共占n列，其中有n位小数，如数值宽度小于m右端补空格。
⑧e格式：以指数形式输出实数。可用以下形式：
%e：数字部分（又称尾数）输出6位小数，指数部分占5位或4位。
%m.ne和%-m.ne：m、n和”-”字符含义与前相同。此处n指数据的数字部分的小数位数，m表示整个输出数据所占的宽度。
⑨g格式：自动选f格式或e格式中较短的一种输出，且不输出无意义的零。
- -

关于printf函数的进一步说明:
如果想输出字符"%",则应该在“格式控制”字符串中用连续两个%表示，如:
printf("%f%%", 1.0/3);
输出0.333333%。
\- -
对于单精度数，使用%f格式符输出时，仅前7位是有效数字，小数6位.
对于双精度数，使用%lf格式符输出时，前16位是有效数字，小数6位.
######################################拾遗##
由高手指点
对于m.n的格式还可以用如下方法表示（例）

```
char ch[20];
printf("%*.*s\n",m,n,ch);
```

前边的*定义的是总的宽度，后边的定义的是输出的个数。分别对应外面的参数m和n 。我想这种方法的好处是可以在语句之外对参数m和n赋值，从而控制输出格式。
\---
今天又看到一种输出格式 %n 可以将所输出字符串的长度值赋给一个变量, 见下例:

```
int slen;
printf("hello world%n", &slen);
```

执行后变量被赋值为11。
不记得的时候，看看还是很有帮助的，特别是scanf的格式化输入，让我大汗，以前对其理解太少了。原来正则也可以在里面使用。
自己在阅读源码的时候，也发现了一些上面所谓提及的。慢慢积累下来，供自己和再看的读者享用。
《UNIX系统编程》P9页（英文版P15）：

```
fprintf(stderr, "a at %p and\nx at %p\n", (void *)a, (viod *)&x);
```

其中提及了%p,自己理解为打印指针地址的格式。
\--

```
public class TestPrintf {
public static void main(String[] args) {
//定义一些变量，用来格式化输出。
double d = 345.678;
String s = "你好！";
int i = 1234;
//"%"表示进行格式化输出，"%"之后的内容为格式的定义。
System.out.printf("%f",d); //"f"表示格式化输出浮点数。
System.out.println();
System.out.printf("%9.2f",d); //"9.2"中的9表示输出的长度，2表示小数点后的位数。
System.out.println();
System.out.printf("%+9.2f",d); //"+"表示输出的数带正负号。
System.out.println();
System.out.printf("%-9.4f",d); //"-"表示输出的数左对齐（默认为右对齐）。
System.out.println();
System.out.printf("%+-9.3f",d); //"+-"表示输出的数带正负号且左对齐。
System.out.println();
System.out.printf("%d",i); //"d"表示输出十进制整数。
System.out.println();
System.out.printf("%o",i); //"o"表示输出八进制整数。
System.out.println();
System.out.printf("%x",i); //"d"表示输出十六进制整数。
System.out.println();
```

```
System.out.printf("%#x",i); //"d"表示输出带有十六进制标志的整数。
System.out.println();
System.out.printf("%s",s); //"d"表示输出字符串。
System.out.println();
System.out.printf("输出一个浮点数：%f，一个整数：%d，一个字符串：%s",d,i,s);
//可以输出多个变量，注意顺序。
System.out.println();
System.out.printf("字符串：%2$s，%1$d的十六进制数：%1$#x",i,s);
//"X$"表示第几个变量。
    }
}
```

字符串

字符之间进行加减不会合成字符串，而是会返回Unicode编码的差值。

逃逸字符就是各种反斜杠字符，包括但不限于\n、\b啥的。注意一点，由于编辑器编译问题，\b在编译器里看上去没啥用，得到其他的编译环境才能进行看到效果。

字符	意义
\b	回退一格
\t	到下一个表格位
\n	换行
\r	回车
"	双引号
\’	单引号
\ \	反斜杠本身

包裹类型

对于**基本数据类型**，Java提供了对应的包裹(wrap)类型。这些包裹类型将一个基本数据类型的数据转换成对象的形式，从而使得它们可以像对象一样参与运算和传递。下表列出了基本数据类型所对应的包裹类型：

基本类型	包裹类型
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`

基本类型	包裹类型
short	Short
int	Integer
long	Long
float	Float
double	Double

字符串

```
String s = new String("hello world");
```

这里需注意String的第一个字母大写。因为这个类似于一个特殊的类库，所以相较于其他的特殊变量而言，这个变量你可以认为是类似于数组一样的存在。其实在之后的学习里，数据结构中也确实存在一个结构叫串。在这里就不做展开。

由于和数组类似，故可以联想到其命名其实和之前数组类似，为标签而非数组本身。

字符串本身是可以通过+号进行连接的。当字符串与一个非字符的量相加的时候，此时会把非字符串的内容自动转化为字符串，从而实现相加。

这里不得不提一下这个神奇的Scanner类了。Scanner类用法

在使用+号键进行长字符串输出的时候一定要注意运算的优先度。

字符串的比较原则

```
if(input == "bye"){
    //whatever
}

if(input.equals("bye")){
    //whatever
}
```

这两者比较有所不同。第一个比较的是是否为同一对象，一般情况下如果出现了诸如字符一致但不属于一个字符串对象的时候会出问题。而第二个方法则为单纯的比较字符串内容是否相同。

字符串常用方法

```
s1.compareTo(s2);//输出0，-1，1
```

比较两字符串的大小。（比较标准灵活，如Unicode编码或是长度什么的。）

```
s1.length();
```

返回长度。

```
s.charAt(index);//返回在index上的单个字符
```

index这个参数不存在复数。

而且竟然不能用for-each循环来遍历字符串。

```
s.substring(n);
s.substring(b,e);
```

得到从n号位置到末尾（或是从n到e-1（没错，和python一样。））的全部内容。

```
s.indexOf(c);//得到c字符所在的位置，-1表示不存在。
s.indexOf(c,n);//从n号位置开始寻找c字符
s.indexOf(t);//找到字符串t所在的位置
s.lastIndexOf(c);//从右开始寻找
s.lastIndexOf(c,n);
s.lastIndexOf(t);
```

字符串常用方法-较为完整版

一、String基本操作方法

首先说一下基本操作方法，字符串的基本操作方法中包含以下几种：(1)获取字符串长度length() (2)获取字符串中的第i个字符charAt(i) (3)获取指定位置的字符方法getChars(4个参数)

1、 获取字符串长度方法length()

```
int length = str.length();
```

2、获取字符串中的第i个字符方法charAt(i)

```
char ch = str.charAt(i); //i为字符串的索引号，可得到字符串任意位置处的字符，保存到字符变量中
```

3、获取指定位置的字符方法getChars(4个参数)

```
char array[] = new char[80]; //先要创建以一个容量足够大的char型数组，数组名为array

str.getChars(indexBegin,indexEnd,array,arrayBegin);
```

解释一下括号中四个参数的指向意义：

1、indexBegin：需要复制的字符串的开始索引

2、 indexEnd：需要复制的字符串的结束索引，indexEnd-1

3、array:前面定义的char型数组的数组名

4、arrayBegin:数组array开始存储的位置索引号

这样我们就可以将字符串中想要的范围内的字符都复制到字符数组中，将字符数组打印输出即可。

与getChars()类似的方法有一个getBytes(),两者使用上基本相同，只是getBytes()方法创建的是byte类型的数组，而byte编码是默认字符集编码，它是用编码表示的字符。

下面就上代码简单演示一下三种方法的用法：

```
//String类基本操作方法
public class StringBasicOpeMethod {
    public static void main(String args[]){
        String str = "如何才能变得像棋哥一样优秀？算了吧，憋吹牛逼！"; //定义一个字符串
        System.out.println(str);  //输出字符串
        /***1、length()方法***/
        int length = str.length();//得到字符串长度
        System.out.println("字符串的长度为："+length);
        /***2、charAt()方法***/
        char ch = str.charAt(7);  //得到索引为7的字符
        System.out.println("字符串中的第8个字符为："+ch);
        /***3、getChars()方法***/
        char chardst[] = new char[80]; //定义容量为80的字符数组，用于存储从字符串中提取出的一串字符
        str.getChars(0,14,chardst,0);
        //System.out.println("字符数组中存放的内容为："+chardst);//错误，输出的是编码
        System.out.println(chardst); //**括号中不可带其他字符串
    }
}
```

二、字符串比较

我们知道，明确的数值之间可以很方便地进行比较，那么字符串该如何进行比较呢？字符串的比较是将两个字符串从左到右逐个字符逐个字符进行比较，比较的依据是当前字符的Unicode编码值，直到比较出两个不同字符的大小。

字符串比较也分为两大类：一类是字符串大小的比较，这样的比较有三种结果，大于、等于以及小于；还有一类比较方法就是比较两个字符串是否相等，这样产生的比较结果无非就两种，true和false。

1、首先看一下第一种比较大小这类需求中的方法：

(1)不忽略字符串大小写情况下字符串的大小比较方法compareTo(another str)

```
int result = str1.compareTo(str2);
```

输出三种比较结果：若该字符串的Unicode值<参数字符串的Unicode值，结果返回一负整数；若若该字符串的Unicode值=参数字符串的Unicode值，结果返回0；若该字符串的Unicode值>参数字符串的Unicode值，结果返回一正整数。

(2) 忽略字符串大小写情况下字符串的大小比较方法compareToIgnoreCase(another str)

```
int result = str1.compareToIgnoreCase(str2);
```

在忽略字符串大小写情况下，返回三种比较结果：输出三种比较结果：若该字符串的Unicode值<参数字符串的Unicode值，结果返回一负整数；若若该字符串的Unicode值=参数字符串的Unicode值，结果返回0；若该字符串的Unicode值>参数字符串的Unicode值，结果返回一正整数。

2、然后看一下第二种判别两种字符串是否相等(相等情况下必须保证二者长度相等)需求中的方法：

(1)不忽略字符串大小写情况下判别字符串相等的方法equals(another str)

```
boolean result = str1.equals(str2);
```

当且仅当str1和str2的长度相等，且对应位置字符的Unicode编码完全相等，返回true,否则返回false

(2) 忽略字符串大小写情况下判别字符串相等的方法equalsIgnoreCase(another str)

```
boolean result = str1.equals(str2);
```

demo如下：

```
public class StringCompareMethod {
    public static void main(String args[]){
        String str1 = "elapant";
        String str2 = "ELEPANT";  //定义两个字符串
        String str3 = "Apple";
        String str4 = "apple";
        /***1、compareTo方法***/
        //不忽略字符串字符大小写
        if(str1.compareTo(str2)>0){
            System.out.println(str1+">"+str2);
        }else if(str1.compareTo(str2) == 0){
            System.out.println(str1+"="+str2);
        }else{
            System.out.println(str1+"="+str2);
        }
        /***2、compareToIgnoreCase()方法***/
        //忽略字符串字符大小写
        if(str1.compareToIgnoreCase(str2)>0){
            System.out.println(str1+">"+str2);
        }else if(str1.compareToIgnoreCase(str2) == 0){
            System.out.println(str1+"="+str2);
        }else{
            System.out.println(str1+"<"+str2);
        }
        /***3、equals()方法***/
        //不忽略字符串字符大小写
        if(str3.equals(str4)){
            System.out.println(str3+"="+str4);
        }else{
            System.out.println(str3+"!="+str4);
        }
        /***4、equalsIgnoreCase()方法***/
        //忽略字符串字符大小写
        if(str3.equalsIgnoreCase(str4)){
            System.out.println(str3+"="+str4);
        }else{
            System.out.println(str3+"!="+str4);
        }
    }
}
```

三、字符串与其他数据类型的转换

有时候我们需要在字符串与其他数据类型之间做一个转换，例如将字符串数据变为整形数据，或者反过来将整形数据变为字符串类型数据，"20"是字符串，20就是整形数。我们都知道整形和浮点型之间可以利用强制类型转换和自动类型转换两种机制实现两者之间的转换，那么"20"和20这两种属于不同类型的数据就需要用到String类提供的数据类型转换方法了。

由于数据类型较多，因而转换使用的方法也比较多，在此我就用一个表格罗列一下：

数据类型	字符串转换为其他数据类型的方法	其它数据类型转换为字符串的方法1	其他数据类型转换为字符串的方法2
boolean	Boolean.getBoolean(str)	String.valueOf([boolean] b)	Boolean.toString([boolean] b)
int	Integer.parseInt(str)	String.valueOf([int] i)	Int.toString([int] i)
long	Long.parseLong(str)	String.valueOf([long] l)	Long.toString([long] l)
float	Float.parseFloat(str)	String.valueOf([float] f)	Float.toString([float] f)
double	double.parseFloat(str)	String.valueOf([double] d)	Double.toString([double] d)
byte	Byte.parseByte(str)	String.valueOf([byte] bt)	Byte.toString([byte] bt)
char	str.charAt(i)	String.valueOf([char] c)	Character.toString([char] c)

```
public class StringConvert {
    public static void main(String args[]){
        /***将字符串类型转换为其他数据类型***/
        boolean bool = Boolean.getBoolean("false"); //字符串类型转换为布尔类型
        int integer = Integer.parseInt("20");     //字符串类型转换为整形
        long LongInt = Long.parseLong("1024");    //字符串类型转换为长整形
        float f = Float.parseFloat("1.521");      //字符串类型转换为单精度浮点型
        double d = Double.parseDouble("1.52123");//字符串类型转换为双精度浮点型
        byte bt = Byte.parseByte("200");          //字符串类型转换为byte型
        char ch = "棋哥".charAt(0);
        /***将其他数据类型转换为字符串类型方法1***/
        String strb1 = String.valueOf(bool);     //将布尔类型转换为字符串类型
        String stri1 = String.valueOf(integer); //将整形转换为字符串类型
        String strl1 = String.valueOf(LongInt); //将长整型转换为字符串类型
        String strf1 = String.valueOf(f);        //将单精度浮点型转换为字符串类型
        String strd1 = String.valueOf(d);        //将double类型转换为字符串类型
        String strbt1 = String.valueOf(bt);      //将byte转换为字符串类型
        String strch1 = String.valueOf(ch);      //将字符型转换为字符串类型
    }
}
```

四、字符串查找

我们有时候需要在一段很长的字符串中查找我们需要的其中一部分字符串或者某个字符，String类恰恰提供了相应的查找方法，这些方法返回的都是目标查找对象在字符串中的索引值，所以都是整形值。具体分类情况如下：

字符串查找无非分为两类：查找字符串和查找单个字符，而查找又可分为查找对象在字符串中第一次出现的位置和最后一次出现的位置，再扩展一步，我们可以缩小查找范围，在指定范围之内查找其第一次或最后一次出现的位置。

（1）查找字符出现的位置

1、indexOf()方法

```
str.indexOf(ch);//1
str.indexOf(ch,fromIndex); //2.包含fromIndex位置
```

格式1返回指定字符在字符串中第一次出现位置的索引

格式2返回指定索引位置之后第一次出现该字符的索引号

2、lastIndexOf()方法

```
str.lastIndexOf(ch);//1
str.lastIndexOf(ch,fromIndex);//2
```

格式1返回指定字符在字符串中最后一次出现位置的索引

格式2返回指定索引位置之前最后一次出现该字符的索引号

(2) 查找字符串出现的位置

1、indexOf()方法

```
str.indexOf(str);//1

str.indexOf(str,fromIndex);//2
```

格式1返回指定子字符串在字符串中第一次出现位置的索引

格式2返回指定索引位置之前第一次出现该子字符串的索引号

2、lastIndexOf()方法

```
str.lastIndexOf(str);//1

str.lastIndexOf(str,fromIndex); //2
```

格式1返回指定子字符串在字符串中最后一次出现位置的索引

格式2返回指定索引位置之前最后一次出现该子字符串的索引号

看代码:

```
//字符与字符串查找
public class StringSearchChar {
    public static void main(String args[]){
        String str = "How qi become handsome like qi ge"; //定义一个长字符串
        System.out.println("该字符串为: "+str);
        /***1、indexOf()方法查找字符首个出现位置格式1,2***/
        int index1 = str.indexOf(" ");  //找到第一个空格所在的索引
        int index2 = str.indexOf(" ",4); //找到索引4以后的第一个空格所在索引
        System.out.println("第一个空格所在索引为: "+index1);
        System.out.println("索引4以后的第一个空格所在索引为: "+index2);
        System.out.println("*****************");
        /***2、lastIndexOf()方法查找字符最后出现位置格式1,2***/
        int index3 = str.lastIndexOf(" ");  //找到最后一个空格所在的索引
        int index4 = str.lastIndexOf(" ",10);//找到索引10以后的第一个空格所在索引
        System.out.println("最后一个空格所在索引为: "+index3);
        System.out.println("索引10以前最后一个空格所在索引为: "+index4);
```

```
        System.out.println("*****************");
        /***3、indexOf()方法查找子字符串第一次出现位置格式1,2***/
        int index5 = str.indexOf("qi"); //找到"qi"子字符串第一次出现位置的索引
        int index6 = str.indexOf("qi",5);//找到索引5以后子字符串"qi"第一个出现位置所在索引
        System.out.println("子字符串qi第一次出现位置的索引号为："+index5);
        System.out.println("索引5以后子字符串qi第一次出现位置的索引号为："+index6);
        System.out.println("*****************");
        /***4、lastIndexOf()方法查找子字符串最后一次出现位置格式1,2***/
        int index7 = str.lastIndexOf("qi");
        int index8 = str.lastIndexOf("qi",5);
        System.out.println("子字符串qi最后一次出现位置的索引号为："+index7);
        System.out.println("索引号5以后子字符串qi最后一次出现位置的索引号为："+index8);
    }
}
```

五、截取与拆分

这类方法是截取出一个长字符串中的一个子字符串或将字符串按照正则表达式的要求全部拆分保存到一个字符串数组中。具体方法如下所示：

(1) 截取方法

1、substring()方法

```
String result = str.substring(index); //1
String result = str.substring(beginIndex,EndIndex); //2.实际索引号[beginIndex,EndIndex-1]
```

结果：截取出范围内的字符串

(2) 拆分方法

1、split()方法

```
String strArray[] = str.split(正则表达式);// 1.拆分的结果保存到字符串数组中
String strArray[] = str.split(正则表达式，limit);//2
```

代码示例如下：

```
//字符串截取与拆分
public class StringCutAndSplit {
    public static void main(String args[]){
        String str = "How to cut and split strings"; //定义字符串
        System.out.println("字符串为："+str);
        int length = str.length(); //获取字符串长度，保存到变量中
        System.out.println("字符串长度为："+length);
        /***1、substring()方法截取出第一个单词和最后一个单词***/
        //首先配合indexOf()和lastIndexOf()方法找到第一个单词和最后一个单词前后空格的索引号
        //第一个单词的左范围索引为0，最后一个单词的右范围索引为length-1
        int firstIndex = str.indexOf(' '); //找到第一个空格所在位置
        int lastIndex = str.lastIndexOf(' '); //找到最后一个空格所在位置
        System.out.println("第一个空格的索引号为："+firstIndex);
        System.out.println("最后一个空格的索引号为："+lastIndex);
        //利用substring()方法根据第一个和最后一个单词的索引范围截取出第一个和最后一个单词
        String firstWord = str.substring(0,firstIndex); //截取出第一个单词
        String lastWord = str.substring(lastIndex+1,length);//截取出最后一个单词
        System.out.println("第一个单词为："+firstWord);
```

```
        System.out.println("最后一个单词为："+lastWord);
        /***1、split()方法拆分出所有单词***/
        String stringArray[] = str.split(" ");  //根据空格要求拆分出所有单词保存到字符串数组中
        System.out.println("拆分之后的各个词汇为："); //输出提示信息
        for(int i = 0;i<stringArray.length;i++){
            System.out.print(stringArray[i]+"\t");
        }
    }
}
```

六、替换或修改

有时候我们需要对原字符串中的某些子字符串进行替换或修改，此时也需要String类提供的一些简单快捷好用的方法.

(1)concat()方法合并字符串

```
String result = str1.concat(str2);  //将str1和str2合并
```

(2) toLowerCase()方法 将字符全部转化为小写

```
String result = str.toLowerCase();
```

(3) toUpperCase()方法 将字符全部转化为大写

```
String result = str.toUpperCase();
```

(4)replaceAll()、replaceFirst()方法：需要匹配正则表达式

代码如下：

```
//字符串替换与修改
public class StringFindandReplace {
    public static void main(String args[]){
        String str1 = "vbasic";
        String str2 = "Vbasic";
        System.out.println("str1 = "+str1);
        System.out.println("str2 = "+str2);
        /***1、concat()方法将两字符串合并***/
        String str3 = str1.concat(str2);
        System.out.println("str1和str2合并后的字符串为："+str3);
        /***2、toLowerCase()方法将str1字符全部转换为小写***/
        String str4 = str1.toLowerCase();
        System.out.println("str1的字符全部转换为小写："+str4);
        /***3、toUpperCase()方法将str2字符全部转换为大写***/
        String str5 = str2.toUpperCase();
        System.out.println("str2的字符全部转换为大写："+str5);
        /***4、实现字符串的替换，原字符串内容不变***/
        String str6 = str1.replaceFirst("(?i)VBASIC","C++");
        String str7 = str2.replaceFirst("(?-i)VBASIC","C++");
        System.out.println("替换后的str1："+str6);
        System.out.println("替换后的str2:"+str7);
    }
}
```

TIPs

String相关操作操作并不能改变本身字符串的样貌。也就是说，本身String类型是不可变的，类似于数组，我有可能会采用某些特殊函数、特殊方法来获取数组的特定序列，但这仅仅只是返回一个新的数组，原数组并不发生改变。

有兴趣一定要去了解数据结构，来体会字符串与数组的差异与联系。

注意在之前提到过的"switch-case"语句中，case不仅可以接受实数型，也可以接受字符串型。（高级Java才可以使用）

函数**

本身的命名方式非常类似c#。所以基本上掌握了c#的函数定义方式就掌握了java的函数命名方式。

到这里不得不讲一个东西，叫重构代码。

为了我们接下来的开发工作，有必要明白一些这方面的东西，即别写玩具代码了。

重构及工程代码

这里会花上一点时间来去聊这个事，书本的话其实网上找一找也就那几本，在这里我就不多谈了。

先摘抄一处知乎老哥说的话：

> 个人在创业小团队中的体会是：
> 重构绝对不是一项任务。不应该说我们从今天开始做重构，三个月后完成。
> 重构应该是随时发生的。每一次功能改进都需要进行一点重构。

为什么我到现在才开始讲重构的重要性呢？

因为直到我们接触到函数，代码的重构才现实起来。

代码乱七八糟，甚至不知道这个那个从哪来的，一切靠“bug”来支撑主要功能的实现，可读性近乎为零，是决不可容忍的。

所以，我想提出一个倡议，我们以后每一个代码块都写注释，每一天都花一点时间考虑代码可否做到更加清晰，更加明朗。

接下来就是一些摘抄等一些原生材料了。具体的代码重构意识我觉得需要各位去有意培养，有意训练。

摘抄_1

作者：Lowry
链接：https://www.zhihu.com/question/19574943/answer/1582562633
来源：知乎

经典著作《重构》这本书中有这么一段话：

> 一开始，我所做的重构都停留在细枝末节上。随着代码趋向简洁，我发现自己可以看到一些设计层面的东西了，这些是我以前理解不到的，如果没有重构，我达不到这种高度。

重构，着实是一件让程序员兴奋的事情。

今年年初，我们团队完成了一个复杂项目的重构工作，它属于广告系统最核心的引擎部分，大概有 300 多个文件，3 万多行代码。

从技术方案设计到最终全量上线仅仅花了 1 个月左右的时间，而且没有产生事故。

这应该是我 8 年程序生涯中，经历过的最大型的同时最成功的一次重构项目：速度足够快、计划比较周全、质量过关。

下面我详细复盘下整个重构过程，我相信你耐心看完后，一定会有所收获！

---

01 先聊聊这个系统的历史包袱

我们的广告引擎在这次重构前大概经历了1年半时间的迭代，初期针对的是搜索场景，业务单一，流程清晰。

2019年开始，公司的广告业务开始快速扩张，收入几乎是指数级的增长。在这个过程中，我们的广告引擎面临了两个挑战：

> 1、业务场景开始变得复杂，除了搜索广告，还需要支持信息流推荐以及相似推荐场景。
> 2、广告流量开始快速增加，除了满足功能性需求，还需要兼顾好性能。

经过梳理，整个引擎有大部分逻辑是可以公用的，因此我们定义了一个主体框架，同时将可扩展部分进行了抽象。这样，各个场景能够根据自身业务的特殊性实现某些公共接口即可。另外，从性能角度考虑，我们牺牲了一些代码可读性，把某些逻辑并行化了。

随着业务的发展，搜索场景开始进入快速迭代期，新增策略越来越多，我们的主体框架也是在这个时候逐渐变得不灵活。

如果动主体框架，搜索以外的场景都需要跟着重构。 在业务的快速发展期，工期根本不允许，因此我们只能在现有框架上进行补丁式的开发。 这样，带来了两个很明显的问题：

> 1、为了兼容搜索的特殊逻辑，我们需要在其他场景中增加各种 if 判断来绕过这些逻辑。
> 2、广告策略越来越多，累计了几十个，当框架失去清晰的结构后，有些策略的实现开始变得定制化，缺少层次化的划分和可插拔式的抽象设计。

在这样的背景下，随着改动的积累，代码开始偏离了设计的初衷，技术债务越来越重。但是，我们又始终找不到合适的时机进行重构。

转机出现在 2019 年年底，由于广告业务的特殊性，流量开始自然走低，另外产品运营团队将重心放在了第 2 年的工作规划上，因此给了我们非常好的窗口期开始此次重构。

我们将工期定成了 1 个月，最终仅比预期晚上线了一天，虽然出现了两个线上问题，但是在灰度期都及时发现和修复了，并没有造成线上事故。

总体来说，这是一次难度颇大并且比较成功的重构项目，下面详细说一下我从这个项目中吸取到的宝贵经验。

---

02 重构前，我们做了哪些准备工作?

这次重构的代码量很大，3 万多行，而且是广告系统最核心的引擎部分。启动前，我们能预期到下面这些困难：

1、业务侧的阻力：广告是极其以业务为导向的，本次重构虽然能带来长期研发效率的提升，但是没法直接提升业务收益，而且开发周期不会太短，如何才能得到业务同学的支持?

2、技术侧的顾虑：重构一旦引起线上事故，公司是有处罚制度的，如何让大家轻装上阵？同时，重构过程中如果还有非常重的业务迭代穿插，交付时间没人敢保证，质量也很难得到控制。

针对这两方的顾虑，我认为下面这几项工作起到了很关键的作用。

1、让所有人看到痛点

前面提到：随着业务迭代，我们广告引擎的主体框架已经变得模糊不清，另外几十个广告策略散落在不同的业务场景中，配置凌乱。

针对这两个痛点，我们提前1个月启动了现有业务的梳理，走读旧代码、同时翻阅以前的需求文档，最终我们将不同场景的核心流程以及广告策略归类成了一张清晰的表格。

正是这一张表格，让技术和产品第一次很清晰地看到了我们引擎部分的全貌，体会到了业务的复杂度以及当前技术上的瓶颈。

2、明确重构的目标和价值

让所有人感受到痛点后，我们规划了本次重构的两个核心目标：

> 1、主体框架的重构：将主流程模块化，重新定义上下层协议，确保接口清晰；各层级内部也需要做好抽象，具备良好的扩展性。
> 2、策略灵活可配置：将广告策略按照业务意图进行归类抽象，策略的执行条件动态可配置，同时策略可任意插拔。

此外，我们将这两个核心目标完成后可带来的预期收益进行了细化：

> 1、技术收益：代码结构更清晰，更容易理解和维护；可扩展性增强，引擎的开发效率将进一步提升。
> 2、业务收益：策略能做到更细粒度的配置和扩展，对业务支持更友好；研发提效后能进一步加快业务的迭代速度。

将重构的价值同步给大家后，进一步提升了所有人的兴奋度，让大家有了更强的动力参与进来。

3、整体节奏的把控

整体节奏的把控也是非常重要的一环，能让所有人对这件事情有一个时间上的预期。

首先，我们将工期定成了 1 个月，一方面考虑了业务侧可以接受的最大周期，技术上也希望速战速决；另一方面，春节即将来临，我们必须赶在公司封网前上线，同时预留出1-2周的 buffer 以防意外情况发生。

此外，我们和业务侧达成了一致：重构期间，引擎部分的非紧急需求一律不接，这样可最大限度地减少并行开发和代码冲突，让团队精力更集中。

---

03 执行过程中有哪些可分享的经验?

这次重构能够实施得如此顺利，有 4 点我认为很有价值的经验跟大家分享下。

1、高质量的技术设计方案

这一点得益于日常的要求，针对开发周期超过3天的项目我们都会进行技术方案设计，本次重构当然也不例外。

框架部分的整体架构、模块之间的协议设计、以及策略的可扩展性设计是本次技术方案的重点，团队前后讨论了不下3次。

在大方案定稿后，团队进一步对数据库、接口字段、缓存结构、日志埋点等公共部分进行了细化，因为涉及到多人协作开发，团队约定以文档作为沟通界面，文档始终保持和代码同步。

在这样的高要求下，团队产出了 5000 多字的技术方案文档，合计 36 页，这些为整体质量的保障打下了很好的基础。

2、 预重构出框架性代码

这一个 PR 非常关键，是我们从技术方案落地到代码最重要的一步。我们把重构后的包结构、模块划分、各层之间的API定义、不同广告策略的抽象进行了梳理，先忽略实现的细节。

这样主体代码基本成型，能很清楚地描绘出我们理想中的框架。然后，我们组织了多次集中代码审查，最终形成了统一意见。

这一步能很好地避免过早陷入实现细节，导致主体框架关注不够、代码不稳固，后期再返工反而会拖累效率。

3、 频繁沟通和成对代码 Review 机制

进入到细节实现阶段后，很重要的一点是：对现有逻辑的理解。引擎代码经过一年半的迭代，历史上被很多人开发过，但是本次只有 3 个同学参与重构。

整个过程中，我们遇到任何代码逻辑不明确的地方，都是反复沟通和求证，不主观猜想，这一份谨慎其实很关键。

另外在代码审查上，我们按模块分配了对这块业务比较熟悉的同学来负责，成对搭配，机制灵活。

4、 有效的测试方案

重构未动，测试先行。这个原则是《重构》一书中重点强调的，也是我们本次技术方案讨论的重点，我这里单独拎出来详细展开下。

首先，我们前期便约定好：不动任何老代码，完全建新的 package 进行重构。这样方便比对重构前后的结果，同时进行线上灰度实验。

测试方案上，以下 4 点非常值得借鉴：

> 1、端到端测试：本次重构不涉及功能性的调整，因此外层API的行为是不会有任何变化的，这样端到端的测试方法最为有效，这个是研发和QA测试最主要的手段。
> 2、冒烟测试：QA同学提供冒烟 Case，由研发同学进行冒烟，研发提测前必须保证所有冒烟 Case 执行通过。这一点在大部分互联网公司都不常见，但是对于大型项目绝对有效。
> 3、沙箱环境双流程验证：前面提到我们重构前后的代码都保留了，因此可以通过脚本抓取线上环境的入参作为case，然后用自动化的方式对 API 的返回字段进行逐一比对。
> 4、线上环境灰度实验：灰度对于重构非常重要，我们利用已有的ABTest平台，逐步放开灰度流量，从5%、到10%、到30%、最后到100%，制定了很谨慎的放量节奏，然后通过日志以及业务指标监控进行验证。

写在最后

回顾整个重构的过程，总结成下面 7 条高价值的建议：

> 1、把握好重构时机
> 2、前期梳理很重要，先找到痛点
> 3、明确出目标和价值，让大家兴奋起来
> 4、不宜长线作战，不宜和业务并行
> 5、需要高质量的技术方案
> 6、重构未动，测试先行
> 7、小心求证，为每行代码负责

除了上述7条建议，还有一个很关键的因素「人」，大型项目重构极其考验团队的协作能力，如果每个人都很靠谱，重构就已经成功了一半。

摘抄_2

作者：陈宇明
链接：https://www.zhihu.com/question/19574943/answer/119537441
来源：知乎

重构的定义

重构是在不改变软件可观察行为的前提下改善其内部结构。

重构的节奏

以微小的步伐修改程序。如果你犯下错误，很容易便可发现它。

一个方法里面，不应该有很多的代码，我们可以通过分解后重组。

好的代码应该清楚的表达出自己的功能，变量名称是代码清晰的关键。

尽量减少临时变量，大量参数被传来传去，很容易跟丢，可读性差。

提炼出逻辑代码，以便功能复用。

> 重构（名词）：对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。
> 重构（动词）：使用一系列重构首发，在不改变软件可观察行为的前提下，调整其结构。

为何重构?

重构改进软件设计

重构是软件更容易理解

重构帮助找到bug

重构提高编程速度

何时重构?

几乎任何情况下我都反对专门拨出时间进行重构。在我看来重构本来就不是一件应该特别拨出时间做的事情，重构应该随时随地的进行。你不应该为重构而重构，你之所以重构，是因为你想做别的什么事，而重构可以帮助你把那些事做好。

三次法则

第一次做某件事只管去做，第二次做类似的事情会有反感，第三次再做类似的事，你就应该重构。
事不过三，三则重构。

添加功能时重构

修补错误时重构

复审代码时重构

为什么重构有用?

难以修改的程序

难以阅读的

逻辑重复的

添加新行为需要修改已有代码的

带复杂条件逻辑的

好的程序

容易阅读

所有逻辑都只是在唯一地点指定

新的改动不会危及现有行为

尽可能简单表达条件逻辑

重构是这样一个过程：它在一个目前可运行的程序上进行，在不改变程序行为的前提下使其具备上述美好性质，使我们能够继续保持高速开发，从而新增程序的价值。

何时不该重构?

无法稳定运行直接重写不用重构

项目以及接近最后期限，不应该重构，虽然重构能够提高生产力，但是你没有足够的时间，通常标示你其实早该进行重构了。

代码的坏味道

Duplicated Code 重复代码

Long Method 过长函数

Large Class 过大的类

Long Parameter List 过长参数列

Divergent Change 发散式变化

Shotgun Surgery 散弹式修改

Feature Envy 依恋情结 （Strategy\Visitor）

Data Clumps 数据泥团

Primitive Obsession 基本类型偏执

Switch Statements switch惊悚现身 （使用多态性替换）

Parallel Inheritance Hierarchies 平行继承体系

Lazy Class 冗赘类

Speculative Generality 夸夸其谈未来性

Temporary Field 令人迷惑的暂时字段

Message Chains 过渡耦合的消息链

Middle Man 中间人

Inappropriate Intimacy 狎昵关系

Alternative Classes with Different Interfaces 异曲同工的类

Incomplete Library Class 不完美的库类

Data Class 纯稚的数据类

Refused Bequest 被拒绝的遗赠

Comments 过多的注释

> 当你感觉需要撰写注释时，请先尝试重构，试着让所有注释变得多余。

构建测试体系

确保所有测试都完全自动化，让他们检查自己的测试结果。

一套测试就是一个强大的bug侦察器，能够大大缩减查找bug所需要的时间。

频繁地运行测试。每次编译请把测试也考虑进去—— 每天至少执行每个测试一次。

每当你收到bug报告，请先写一个单元测试来暴露这个bug。

编写未完善的测试并执行，好过对完美测试的无尽等待。

考虑可能出错的边界条件，把测试火力集中在那儿。

当事情被认为应该会出错时，别忘了检查是否抛出了预期的异常。

不要因为测试无法捕捉所有bug就不写测试，因为测试可以捕捉到大多数的bug。

重构列表

重构记录格式

名称

概要

描述解决的问题

描述要做的事情

速写图展示重构前和重构后的示例

动机

做法

范例

> 重构的基本技巧—小步前进、频繁测试
> 模式和重构之间有着一种与生俱来的关系。模式是你希望到达的目标，重构则是到达之路。

重新组织函数

Extract Method 提炼函数

你有一段代码可以被组织在一起并独立出来。将这段代码放进一个独立函数中，并让函数名称解释该函数的用途。

Inline Method 内联函数

一个函数，其本体应该与其名称同样清楚易懂。在函数调用点插入函数本体，然后移除该函数。

Inline Temp 内联临时变量

你有一个临时变量，只被一个简单表达式赋值一次，而它妨碍了其他重构方法。将所有对该变量的引用动作，替换为对它赋值的那个表达式本身。

Replace Temp with Query 已查询取代临时变量

你的程序以一个临时变量(temp)保存某一个表达式的运算结果。将这个表达式提炼到一个独立函数(query查询式)中。将这个临时变量的所有被引用点替换为对新函数的调用。新函数可被其他函数使用。

Introduce Explaining Variable 引入解释性变量

你有一个复杂的表达式。将该复杂表达式(或其中一部分)的结果放进一个临时变量，以此变量名称来解释表达式用途。

Split Temporary Variable 分解临时变量

你的程序有某个临时变量被赋值超过一次，它既不是循环变量，也不是一个集用临时变量(collecting temporary variable)。针对每次赋值，创造一个独立。对应的临时变量。

Remove Assignments to Parameters 移除对参数的赋值

你的代码对一个参数进行赋值动作。以一个临时变量取代该参数的位置。

Replace Method with Method Object 以函数对象取代函数

你有一个大型函数，其中对局部变量的使用，使你无法采用Extract Method。将这个函数放在一个独立的对象中，如此一来局部变量就变成了对象内的值域，然后你可以在同一个对象中将这个大型函数分解为数个小型函数。

Substitute Algorithm 替换算法

你想要把某个算法替换为另一个更清晰的算法。将函数本体替换为另一个算法。

在对象之间搬移特性

Move Method 搬移函数

你的程序中，有个函数与其所驻class之外的另一个class进行更多交流：调用后者，或被后者调用。在函数最常引用的class中建立一个有着类似行为的新函数。将旧函数变成一个单纯的委托函数，或是将旧函数完全移除。

Extract Class 提炼类

你的程序中，某个field(值域)被所驻class之外的另一个class更多地用到。在target class 建立一个new field ,修改source field 的所有用户，令它们改用new field.

Inline Class 将类内敛化

某个class做了应该由两个classes做的事情。建立一个新的class，将相关的值域和函数从就class搬移到新class。

Hide Delegate 隐藏委托类

客户直接调用其server object(服务对象)的delegate class。在sever端（某个class）建立客户所需要的所有函数，用以隐藏委托关系。

Remove Middle Man 移除中间人

某个class做了过多的简单委托(simple delegation).让客户直接调用delegate(受托类)。

Introduce Foreign Method 引入外加函数

你所使用的server class 需要一个额外函数，但你无法修改这个class。在client class 中建立一个函数，并以一个server class实体作为第一引数(argument)。

Introduce Local Extension 引入本地扩展

你所使用的server class 需要一些额外函数，但你无法修改这个class。建立一个新class，使它包含这些额外函数。让这个扩展品成为source class的subclass(子类)或(wrapper)外覆类。

重新组织数据

Self Encapsulate Field 自封装字段

你直接访问一个值域(field)，但与值域直接的耦合关系变得逐渐变得笨拙。为这个值域建立取值/设值函数，并且只有这些函数来访问值域。

Replace Data Value with Object 以对象取代数据值

你有一笔数据项(data item),需要额外的数据和行为。将这笔数据项变成一个对象。

Change Value to Reference 将值对象改成引用对象

你有一个class，衍生出许多相等视图(equal instance),你希望将它们替换为单元对象。将这个 value object（实值对象）变成一个reference object（引用对象）。

Change Reference to Value 将引用对象改成值对象 （equals hashCode）

你有一个reference object(引用对象)，很小且不可变，而且不易管理。将他变成一个value object(实值对象)。

Replace Array with Object 以对象取代数组

你有一个数组(array),其中的元素个各自代表不同的东西，以对象替换数组。对于数组中的每个元素，以一个值域表示之。

Duplicate Observed Data 复制“被监听数据”

你有一些domain data 置身GUI控件中，而domain method 需要访问之。即那个该笔数据拷贝到以到domain object。建立一个observer模式，用以对domain object和GUI object 内的重复数据进行同步控制(sync).

Change Unidirectional Association to Bidirectional 将单向关联改为双向关联

两个classes都需要使用对方特性，但其间只有一条单向连接。添加一个反向指针，并使修改函数能够同时更新两条连接。

Change Bidirectional Association to Unidirectional 将双向关联改成单向关联

两个classes之间有双向关联，但其中一个class如今不再需要另一个class的特性。去除不必要的关联(association)。

Replace Magic Number with Symbolic Constant 以字面常量取代魔法数

你有一个字面值，带有特别的含义。创造一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量。

Encapsulate Field 封装字段

你的class存在一个public值域。将它声明为private,并提供相应的访问函数。

Encapsulate Collection 封装集合

有个函数返回一个群集(collection)。放这个函数返回该群集的一个只读映件，并在这个class中提供添加移除群集元素的函数。

Replace Record with Data Class 以数据类取代记录

你需要面对传统编程环境中的record structure（记录结构）。为该record（记录）创建一个哑数据对象（dumb data object）。

Replace Type Code with Class 以类取代类型码

class之中有一个数值别码，但他并不影响class的行为。以一个新的class替换数值型别码。

Replace Type Code with Subclasses 以子类取代类型码 （多态机制）

你有一个不可变的type code，它会影响class的行为。以一个subclass取代这个type code。

Replace Type Code with State/Strategy 以State/Strategy取代类型码

你有一个type code,它会影响class 的行为，但你无法使用subclassing。以state object取代type code.

Replace Subclass with Fields 以字段取代子类

你的各个subclasses的唯一差别只在返回常量数据的函数身上。修改这些函数，使他们返回superclass中的某个(新增)值域，然后销毁sublcasses。

简化条件表达式

Decompose Conditional 分解条件表达式

你有一个复杂的条件语句。从if、then、else三个段落中分别提出独立函数。

Consolidate Conditional Expression 合并条件表达式

你有一系列条件测试，都得到相同结果。将这些测试合并为一个条件式，并将这个条件式提炼成为一个独立函数。

Consolidate Duplicate Conditional Fragments 合并重复的条件片段

在条件式的每一个分支上着相同的代码。将这个段重复代码搬移到条件式之外。

Remove Control Flag 移除控制标记 （break/continue/return）

在一系列布尔表达式中，某个变量带有控制标记的作用，以break语句或return语句取代控制标记。

Replace Nested Conditional with Guard Clauses 以卫语句取代嵌套条件表达式 （单独判断被称为“卫语句”）

函数中的条件逻辑使人难以看清正常的执行路径。使用卫语句表现所有特殊的情况。卫语句就是把复杂的条件表达式拆分成多个条件表达式，比如一个很复杂的表达式，嵌套了好几层的if - then-else语句，转换为多个if语句，实现它的逻辑，这多条的if语句就是卫语句

Replace Conditional with Polymorphism 以多态取代条件表达式

你手上有个条件式，它根据对象型别的不同而选择不同的行为。将这个条件式的每个分支放进一个subclass内的覆写函数中，然后将原始函数声明为抽象函数。

Introduce Null Object 引入Null对象

你需要再三检查某个是否为null value。将null value(无效值)替换为null object(无效物)

Introduce Assertion 引入断言

某一段代码需要对程序状态做出某种假设。以assertion(断言)明确表现这种假设。

简化函数调用

Rename Method 函数改名

函数的名称未能揭示函数的用途，修改函数名称。

Add Parameter 添加参数

某个函数需要从调用端得到更多信息。为此函数添加一个对象参数，让该对象带进函数所需信息。

Remove Parameter 移除参数

函数本体不再需要某个参数，将该参数去除。

Separate Query from Modifier 讲查询函数和修改函数分离

某个函数即返回函数对象状态值，又修改对象状态。将来两个不同的函数，其中一个负责查询，另一个负责修改。

Parameterize Method 令函数携带参数

若干函数做了类似的工作，但在函数本体中却包含了不同的值。建立单一函数，以参数表达那些不同的值。

Replace Parameter with Explicit Methods 以明确函数取代参数

你有一个函数，其内完全取决于参数值而采取不同反应。针对该参数的每一个可能值，建立一个独立函数。

Preserve Whole Object 保持对象完整

你从某个对象中取出某个值，将它们作为某一次函数调用的参数。该引用（传递）整个对象。

Replace Parameter with Methods 以函数取代参数

对象调用某个函数，并将所得结果作为参数，传递给另一个函数。而接受该参数的函数也可以调用前一个函数。让参数接受者去除该项参数，并直接调用前一个函数。

Introduce Parameter Object 引入参数对象

某个参数总是很自然地同时出现。以一个对象取代这些参数。

Remove Setting Method 移除设置函数

你的class中的某个值域，应该在对象初创时被设值，然后就不再改变。去掉该值域的所有设置函数。

Hide Method 隐藏函数

有一个函数，从来没有被其他任何class用到。将这个函数改为private。

Replace Constructor with Factory Method 以工厂函数取代构造函数

你希望在创建对象时不仅仅是对它做简单的构建工作，将construcotr(构造函数)替换为factory method(工厂函数)

Encapsulate Downcast 封装向下转型

某个函数返回的对象，需要由函数调用者执行向下转型动作。将向下转型动作移到函数中。

Replace Error Code with Exception 以异常取代错误码

某个函数返回一个特定的代码，用以表示某种错误情况。改用异常。

Replace Exception with Test 以测试取代异常

面对一个调用者可预先加以检查的条件，你抛出一个异常。修改调用者，使它在调用函数之前先做检查。

处理概括关系

Pull Up Field 字段上移

两个subclass拥有相同的值域。将此一值域移至superclass。

Pull Up Method 函数上移

有些函数，在各个subclass中产生完全相同的效果。将该函数移至superclass。

Pull Up Constructor Body 构造函数本体上移

你的各个subclass中拥有一些构造函数，它们的本体几乎完全一致。在superclass中新建一个构造函数，并在subclass构造函数中调用它。

Push Down Method 函数下移

superclass中的某个函数只与部分（而非全部）subclass有关。将这个函数移到相关的那些subclasses去。

Push Down Field 字段下移

superclass中的某个值域只被部分subclass用到。将这个值域到需要它的那些subclasses去。

Extract Subclass 提炼子类

class中某些特性只被某些而非全部实体用到。新建一个subclass，将上面所说的那一部分特性移到subclass中。

Extract Superclass 提炼超类

两个classes有相似特性。为这连个classes建立一个superclass.将相同特性移至superclass.

Extract Interface 提炼接口

若干客户使用class接口中的同一子集。或者，两个Classes的接口有部分相同。将相同的子集提炼到一个独立接口中。

Collapse Hierarchy 折叠继承体系

superclass和subclass之间无太大区别。将它们和为一体。

Form Template Method 塑造模板函数

有一些subclasses，其中相应的某些函数以相同顺序执行类似的措施，但各措施实际上有所不同。将各个措施分别放进独立函数中，并保持它们都有相同的签名式，于是原函数也就变得相同了。然后将原函数上移至superclass。

Replace Inheritance with Delegation 以委托取代继承

某个subclass只使用superclass接口中的一部分，或是更本不需要继承而来的数据。在subclass中新建一个值域用以保存superclass；调整subclass函数，令它改而委托superclass；然后去掉两者之间的继承关系。

Replace Delegation with Inheritance 以继承取代委托

你的两个classes之间使用了委托关系，并经常为整个接口编写许多极其简单的请托函数。让请托Class继承受托class。

大型重构

Tease Apart Inheritance 梳理并分解继承体系

某个继承体系同时承担两项责任。建立两个继承体系，并通过委托关系让其中一个可以调用另一个。

Convert Procedural Design to Objects 将过程化设计转化为对象设计

你手上有一些代码，以传统的过程化风格写就。将数据记录变成对象，将行为分开，并将行为移入相关对象中。

Separate Domain from Presentation 将领域和表述/显示分离

某些GUI classes之中包含了domain login(领域逻辑)。将domain loginc(领域逻辑)分离出来，为它们建立独立的domain classes.

Extract Hierarchy 提炼继承体系

你有某个class做了太多工作，其中一个部分工作以大量条件式完成的。建立继承体系，以一个subclass表示一种特殊情况。

TIPS

一般来说如果传入参数可以隐式转换为形参的类型的话，那么解释器会自动转换。

没错，`char`可以隐式转换为`Int`。

Java语言在调用函数的时候，永远只能传值给函数。

OOP***

终于学到点东西了。

部分代码部分其实和c#依旧高度类似。不过这里还是有一些与c#不太一样的地方。

定义类

老规矩，属性与方法。

属性的定义无他，只需要指名属性的类型及一些别的属性的属性（如private等）。

值得注意的是，属性定义中你甚至可以只写一个类型定义。如下所示：

```
int price;//这是上边说的那种简单地。
private int balance;//这是长一点的写法。
```

方法的定义貌似视屏里讲的不够干货，在这里我摘抄博客园的完整类定义的摘抄来具体讲解其中的内容。

摘抄_3

类是 Java 中的一种重要的引用数据类型，也是组成 Java 程序的基本要素，因为所有的 Java 程序都是基于类的。本节介绍如何定义类。

在 Java 中定义一个类，需要使用 class 关键字、一个自定义的类名和一对表示程序体的大括号。完整语法如下：

```
[public][abstract|final]class<class_name>[extends<class_name>][implements<interface_name>] {
    // 定义属性部分
    <property_type><property1>;
    <property_type><property2>;
    <property_type><property3>;
    …
    // 定义方法部分
    function1();
    function2();
    function3();
    …
}
```

提示：上述语法中，中括号“[]”中的部分表示可以省略，竖线“|”表示“或关系”，例如 abstract|final，说明可以使用 abstract 或 final 关键字，但是两个关键字不能同时出现。

上述语法中各关键字的描述如下。

- `public`：表示“共有”的意思。如果使用 public 修饰，则可以被其他类和程序访问。每个 Java 程序的主类都必须是 public 类，作为公共工具供其他类和程序使用的类应定义为 public 类。
- `abstract`：如果类被 abstract 修饰，则该类为抽象类，抽象类不能被实例化，但抽象类中可以有抽象方法（使用 abstract 修饰的方法）和具体方法（没有使用 abstract 修饰的方法）。继承该抽象类的所有子类都必须实现该抽象类中的所有抽象方法（除非子类也是抽象类）。
- `final`：如果类被 final 修饰，则不允许被继承。
- `class`：声明类的关键字。
- `class_name`：类的名称。
- `extends`：表示继承其他类。
- `implements`：表示实现某些接口。
- `property_type`：表示成员变量的类型。
- `property`：表示成员变量名称。
- `function()`：表示成员方法名称。

Java 类名的命名规则：

1. 类名应该以下划线（_）或字母开头，最好以字母开头。
2. 第一个字母最好大写，如果类名由多个单词组成，则每个单词的首字母最好都大写。
3. 类名不能为 Java 中的关键字，例如 `boolean`、`this`、`int` 等。
4. 类名不能包含任何嵌入的空格或点号以及除了下划线（_）和美元符号（$）字符之外的特殊字符。

例 1

创建一个新的类，就是创建一个新的数据类型。实例化一个类，就是得到类的一个对象。因此，对象就是一组变量和相关方法的集合，其中变量表明对象的状态和属性，方法表明对象所具有的行为。定义一个类的步骤如下所述。

(1) 声明类。编写类的最外层框架，声明一个名称为 `Person` 的类。

```
public class Person {    // 类的主体}
```

(2) 编写类的属性。类中的数据和方法统称为类成员。其中，类的属性就是类的数据成员。通过在类的主体中定义变量来描述类所具有的特征（属性），这里声明的变量称为类的成员变量。

(3) 编写类的方法。类的方法描述了类所具有的行为，是类的方法成员。可以简单地把方法理解为独立完成某个功能的单元模块。

下面来定义一个简单的 Person 类。

```
public class Person {
    private String name;    // 姓名
    private int age;    // 年龄
    public void tell()
    {           // 定义说话的方法
        System.out.println(name+"今年"+age+"岁！");
    }
}
```

如上述代码，在 Person 类中首先定义了两个属性，分别为 name 和 age，然后定义了一个名称为 tell() 的方法。

目前我还不太敢一口咬死方法的写法。等到之后的笔记中我会进一步的说明。

目前的初步方法写法是这样的：(方括号里为可选项，圆括号里为必写项)

```
[public] [static] (return_value) function_name([value_type value, ......]) {
    //方法体代码;
    [return [返回值];]
}
```

成员变量

这里有一个有意思的词，是this。

和python像的地方在于python中的`self`也是干这个事情的，不一样的地方在于python里的self不一定写作self。只要你愿意，充当self角色的单词可以由你而定，但java不太一样，他就是`this`，不能更改。

注意本地变量与成员变量的区别。比如说，本地变量在未初始化前是不可以拿来被调用的，但成员变量会自动给一个初始值。

与c不同的是，你不需要关心垃圾变量（即用不到的变量）的去向，因为java有自己的一套垃圾回收机制。

初始化函数

比较一下两段代码：

```python
class A:
    def __init__(self,a):
        self.a_value = a
```

```java
public class A{
    int a_value;
    A(int a){
        this.a_value = a;
    }
}
```

这两者起到的效果是一样的。当我在初始化这个类的时候需要传入一个参数a来实现对a_value的初始化。

此时与类名相同、无任何返回值的函数便是**构造函数**。一般构造函数可以写很多个，我就接着上面那个类继续写下去。

```java
public class A{
    int a_value;
    A(){
        this.a_value = 0;//我啥也没写的情况
    }
    A(int a){
        this.a_value = a;//正常传入一个int的情况
    }
    A(double a){
        this.a_value = (int)a;//有个人传入double型的情况
    }
}
```

有时候你甚至可以在一个构造函数里调用其他的构造函数，方法是使用this()这个函数。

```java
public class A{
    int a_value;
    int b_value
    A(){
        this.a_value = 0;//我啥也没写的情况
        this.b_value = 0;
    }
    A(int a){
        this.a_value = a;//正常传入一个int的情况
        this();//直接调用第一个构造函数。(貌似会把a清0，不过演示吗，无所谓的)
    }
    A(double b){
        this.b_value = (int)b;//有个人传入double型的情况
        this(int a);//调用第二个函数。
    }
}
```

这里我就把重载的定义摘抄过来。

重载(Overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

重载规则:

- 被重载的方法必须改变参数列表(参数个数或类型不一样);
- 被重载的方法可以改变返回类型;
- 被重载的方法可以改变访问修饰符;
- 被重载的方法可以声明新的或更广的检查异常;
- 方法能够在同一个类中或者在一个子类中被重载。
- 无法以返回值类型作为重载函数的区分标准。

  实例:

```
public class Overloading {
    public int test(){
        System.out.println("test1");
        return 1;
    }

    public void test(int a){
        System.out.println("test2");
    }

    //以下两个参数类型顺序不同
    public String test(int a,String s){
        System.out.println("test3");
        return "returntest3";
    }

    public String test(String s,int a){
        System.out.println("test4");
        return "returntest4";
    }

    public static void main(String[] args){
        Overloading o = new Overloading();
        System.out.println(o.test());
        o.test(1);
        System.out.println(o.test(1,"test3"));
        System.out.println(o.test("test4",1));
    }
}
```

Main函数

到了这个地方我觉得不得不去面对一个问题，便是Main函数的相关知识。

网上前辈们的说法是Main函数是一个很难理解的函数，需要掌握一定的虚拟机知识，所以，在这里我尽量找一些能讲明白得一些案例给我们做讲解。

先给一段较为基础的对于main函数本身的讲解，具体之后的讲解等到以后我们进阶到一定程度后在继续学习。

更新：

main函数为程序的接口，大致是这样的流程，当使用编译器编译程序时，JVM虚拟机自动识别main函数，从而以此为入口函数去调用。

好家伙比c#好理解多了

以后为了便于调试与管理，每个类里都应有main函数。但最后写完之后应删除不需调试代码的main函数以避免安全问题。

节选片段

从写java至今，写的最多的可能就是主函数

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

但是以前一直都没有问自己，为什么要这么写，因为在c语言中就没有这样子的要求。其实这是一个不需要解释的问题，因为java标准就是这么规定的，那么既然是java标准规定的，我们按照规定来执行就好了。不过，这并不是一个很好的学习态度，如果总是知其然而不知其所以然，总会对java有种隔膜的感觉。就是发现问题了，不去解决，不去了解为什么，心里总是会有牵绊。今天既然自己都这么问自己了，为什么java的主函数要按照这个格式来写，那么我就得弄明白为什么。

在java中，main()方法是java应用程序的入口方法。java虚拟机通过main方法找到需要启动的运行程序，并且检查main函数所在类是否被java虚拟机装载。如果没有装载，那么就装载该类，并且装载所有相关的其他类。因此程序在运行的时候，第一个执行的方法就是main()方法。通常情况下， 如果要运行一个类的方法，必须首先实例化出来这个类的一个对象，然后通过"对象名.方法名()"的方式来运行方法，但是因为main是程序的入口，这时候还没有实例化对象，因此将main方法声明为static的，这样这个方法就可以直接通过“类名.方法名()”的方式来调用。

对象交互

差不多类似于我可以再写一个类，对子类进行一定的管控。

他与继承完全不一样。你可以把它理解为一种特别的使用方法。比如，我在一个类当中调用Integer，即整形的包裹类型时，虽然在使用中完全没感觉，但这玩意确实是一个类，你也确实在类里边调用类。

这便拓展了一个思路。在没有明显继承关系的时候，如果想要进行代码重构或是功能实现，可以考虑使用该方法进行重写。

访问修饰符

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

- **default** (即默认，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。
- **private** : 在同一类内可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**
- **public** : 对所有类可见。使用对象：类、接口、变量、方法
- **protected** : 对同一包内的类和所有子类可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）。**

我们可以通过以下表来说明访问权限：

修饰符	当前类	同一包内	子孙类(同一包)	子孙类(不同包)	其他包
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	Y/N（说明）	N
default/friendly(不加修饰符)	Y	Y	Y	N	N
private	Y	N	N	N	N

值得提上一句的是，你甚至可以在构造函数使用private修饰符，只不过你想创建对象的话只能在该类里头创建对象。

这里顺带提一嘴非访问修饰符。

非访问修饰符

为了实现一些其他的功能，Java 也提供了许多非访问修饰符。

- static 修饰符，用来修饰类方法和类变量。
- final 修饰符，用来修饰类、方法和变量，final 修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。
- abstract 修饰符，用来创建抽象类和抽象方法。
- synchronized 和 volatile 修饰符，主要用于线程的编程

一个文件里只有一个访问修饰符为public的类，而且其名称必须与文件名相同。

当写到这个地方的时候就不得不提一下编译单元的概念了。

编译单元及相关概念

- java编译单元

.java文件是一个编译单元，每个编译单元顶多只有一个public标记的类，被public标记的类名字必须和文件的名字相同（包含大小写形式，但排除文件扩展名.java）当然一个.java文件可以包含许多其它类，但是其它类都是为这个public类起支撑作用的，他们都不能是public的。当.java文件中没有public标记的类时，这个类不能被外部访问，也就没意义了。

- java编译

可以利用`javac xx.java`对java编译单元进行编译，可以发现一个.java文件中有几个类，就会变成产生几个.class文件，这个和C++中一个编译单元产生一个.object文件是不一样的。需要注意的是如果在一个包中包含两个java文件，如A.java,B.java，如果两个java文件中都包含一个相同名字的类如Test。那么这两个java文件编译后，只能产生一个`Test.class`,当然是谁后编译，这个`Test.class`就属于谁的。一个有效的程序就是一系列的class文件，他们可以封装和 压缩到一个jar文件。

- package

`package` 其实就是想当于一个`namespace`，为了避免命名冲突，可以用`package`将许多`.java`文件以及编译产生后的许多.class文件给包起来。当你建立一个package时候。会自动产生一个同名的文件夹。例如，我们创建一个`ddxxpackage`时候，会自动的创建一个`ddxxpackage`的文件夹，当我们在一个java文件中的第一个非注释语句写

package ddxxpackage;时，我们的java文件就处于这个ddxxpackage文件夹里面。说道本质就是利用操作系统的文件路径的相互独立性来避免命名冲突。

- 自动编译

当在程序中利用导入包中的类来产生一个类对象的时候，编译器会寻找同名的.class文件，如果就这一个.class文件，那么就利用这一个class来创建对象，如果还有一个同名的.java文件，那么会比较诸如xx.class 和xx.java两个文件的时间先后， 如果xx.java比较新，那么就会自动编译xx.java产生新的xx.class来产生对象。

- 在同一个package中注意的问题

同一个包中不同的java编译单元中不能包含相同的类名，即便是friend的类也不行，因为这样在编译的时候会产生类已经定义的错误告示。如A.java包含test class,B.java包含test.class那么编译时候就会出现the type test isalready defined。

- 访问权限

声明为public的类可以在包外被访问，但是默认的类只能宅包内被访问，类中的成员也是这样，声明为public的成员可以被随便的访问如果类也是public的话，那么该成员也可以被访问，但是类是public，成员是default的话，就不能被包外访问。

- 包和目录

其实一个包就相当于一个目录，一个个类包括public，default类都会编译成这个包下的class文件。

项目文件中，里边有bin文件夹与src文件夹，src文件夹中存在一堆叫做package的文件夹(不是名字是package，而是名字是用到的package的名字)，在之后才会有相应的代码文件。

闲的没事千万别import package.*。等哪次出现了严重的bug哭都来不及哭。原因我也简单讲一下，因为不可能保证每一个包里的函数以及成员的名字都不一样，一旦出现名字冲突就出大问题了。用哪个方法就调用哪个方法。

在命名包的时候，形如package.subpackage的命名，最后在文件系统中会变成在package的文件底下有个名叫subpackage的文件。

所以，在java里利用这种方式来进行包的管理。sbpython哈哈

类变量

类变量是该类的所有对象共享的变量，任何一个该类的对象去访问它时，取到的都是相同的值，同样任何一个该类的对象去修改它时，修改的也是同一个变量。

下列两个代码的效果是等同的：

```
class A:
    a_value = 0
    def __init__(self,b):
        self.b_value = b

    @classmethod
    def totalNumber(cls):
        cls.a_value += 1

    def showNum(self):
        print(self.a_value)
```

```
public class A{
    private static int a_value = 0;
    private int b_value;
    public A(int b){
        this.b_value = b;
    }
    public void addNum(){
        this.a_value += 1;
    }
    public int getValue(){
        return a_value;
    }
}
```

一段使用python的类方法写的，一段是用java实现的。

类方法

也就是静态方法。直白点来讲就是有点类似于工件包的感觉。

比如一个方法他压根用不到类里的成员，他只是简单的处理一下数据并返回一个东西，那么这个时候用静态方法再合适不过了。这里其实就和python不一样的太多了。python可以写成纯面向过程编程，但Java想办到这一点有点困难。

泛型容器类

此处原理部分参考笔记中的Java泛型与容器初探。（点击即看）

那里相对来说比较枯燥乏味，先看点简单点的东西。

```
private ArrayList<String> notes = new ArrayList<String>();
```

这便是对容器的定义。关于这个ArrayList的使用方法在这里贴出来：

ArrayList模块使用

对象数组

对象数组对数组的概念进行了扩充。理论上只要是一个类就可以定义一个对应的类的数组。此时会分配到一个全为null的空间共该数组使用。

for-each依旧可以使用于对象数组或是容器。值得注意的是要进行初始化的时候依旧需要new一个对象。

继承

```
public class CD extends Item{
    //whatever
}
```

子类父类关系

Java只支持单继承，不支持多继承。

1. 一个类只能有一个父类，不可以有多个父类。
2. `class SubDemo extends Demo{}`//ok
3. `class SubDemo extends Demo1,Demo2...`//error

Java支持多层继承(继承体系)

```
class A{}

class B extends A{}

class C extends B{}
```

继承的注意事项：

1. 子类只能继承父类所有非私有的成员(成员方法和成员变量)
2. 子类不能继承父类的构造方法，但是可以通过super(马上讲)关键字去访问父类构造方法。
3. 不要为了部分功能而去继承

在子类方法中访问一个变量

1. 首先在子类局部范围找
2. 不能访问到父类局部范围)
3. 如然后在子类成员范围找
4. 最后在父类成员范围找(肯定果还是没有就报错。(不考虑父亲的父亲…)

关于`super()`的一些用法视频中讲解较为详细，在这里我就复制一段网上关于`this`与`super`的区别以供参考。

- `super`（参数）：调用基类中的某一个构造函数（应该为构造函数中的第一条语句）
- `this`（参数）：调用本类中另一种形成的构造函数（应该为构造函数中的第一条语句）
- `super`: 它引用当前对象的直接父类中的成员（用来访问直接父类中被隐藏的父类中成员数据或函数，基类与派生类中有相同成员定义时如：`super.`变量名 `super.`成员函数据名（实参）
- `this`：它代表当前对象名（在程序中易产生二义性之处，应使用`this`来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用`this`来指明成员变量名）
- 调用`super()`必须写在子类构造方法的第一行，否则编译不通过。每个子类构造方法的第一条语句，都是隐含地调用`super()`，如果父类没有这种形式的构造函数，那么在编译的时候就会报错。
- `super()`和`this()`类似,区别是，`super()`从子类中调用父类的构造方法，`this()`在同一类内调用其它方法。
- `super()`和`this()`均需放在构造方法内第一行。
- 尽管可以用`this`调用一个构造器，但却不能调用两个。
- `this`和`super`不能同时出现在一个构造函数里面，因为`this`必然会调用其它的构造函数，其它的构造函数必然也会有`super`语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。
- `this()`和`super()`都指的是对象，所以，均不可以在`static`环境中使用。包括：`static`变量,`static`方法，`static`语句块。
- 从本质上讲，`this`是一个指向本对象的指针, 然而`super`是一个Java关键字。

造型

所有的赋值语句的含义均为管理，即地址导向，而不是真正意义上改变这个变量。

子类的对象可以向上造型为父类的类型。即父类引用子类对象，这种方式被称为向上造型。可以知道子类对象可以安全的赋值给父类对象，但反向不行。

一般用括号围起类型放在值前面，对象本身并没有发生任何变化，所以这并非类型转化。类型转换直接干掉了元对象里所有的东西，而造型不是。

向上造型是拿一个子类的对象，当作父类的对象来用，而这是安全的，不需要特殊的运算符。

覆盖

方法覆盖（Overriding）： 如果在子类中定义一个方法，其名称、返回类型及参数签名正好与父类中某个方法的名称、返回类型及参数签名相匹配，那么可以说，子类的方法覆盖了父类的方法。

```
public class Findareas{
    public static void main (String []agrs){
        Figure f= new Figure(10 , 10);
        Rectangle r= new Rectangle(9 , 5);
        Figure figref;
        figref=f;
        System.out.println("Area is :"+figref.area());
        figref=r;
        System.out.println("Area is :"+figref.area());
    }
}
class Figure{
    double dim1;
    double dim2;
    Figure(double a , double b) {
        dim1=a;
        dim2=b;
    }
    Double area() {
        System.out.println("Inside area for figure.");
        return(dim1*dim2);
    }
}
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a ,b);
    }
    Double area() {
        System.out.println("Inside area for rectangle.");
        return(dim1*dim2);
    }
}
```

这个时候我不用去关心父类对象在调用被覆盖方法的时候需要调用哪个类的方法，我只需要关心每个子类的重名函数的的内部方法。

值得一提的是`@Override`类似于一种标注。具体看下解析。

摘抄

@Override是伪代码,表示重写。(当然不写@Override也可以)，不过写上有如下好处:
1、可以当注释用,方便阅读;
2、编译器可以给你验证@Override下面的方法名是否是你父类中所有的，如果没有则报错。例如，你如果没写@Override，而你下面的方法名又写错了，这时你的编译器是可以编译通过的，因为编译器以为这个方法是你的子类中自己增加的方法。

举例：在重写父类的onCreate时，在方法前面加上@Override系统可以帮你检查方法的正确性。

```
@Override
public void onCreate(Bundle savedInstanceState){
    //whatever
    //注意看函数名
}
```

这种写法是正确的，如果你写成：

```
@Override
public void oncreate(Bundle savedInstanceState){
    //whatever
    //注意看函数名
}
```

编译器会报如下错误：`The method oncreate(Bundle) of type HelloWorld must override or implement a supertype method`，以确保你正确重写onCreate方法（因为oncreate应该为onCreate）。

而如果你不加@Override，则编译器将不会检测出错误，而是会认为你为子类定义了一个新方法：oncreate

抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。也是因为这个原因，通常在设计阶段决定要不要设计抽象类。

父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。

在Java中抽象类表示的是一种继承关系，一个类只能继承一个抽象类，而一个类却可以实现多个接口。

使用abstract修饰符来修饰抽象函数与抽象类。有抽象函数的类一定是抽象类，抽象类不能制造对象，但是可以定义变量。

- 1. 抽象类不能被实例化(初学者很容易犯的错)，如果被实例化，就会报错，编译无法通过。只有抽象类的非抽象子类可以创建对象。
- 2. 抽象类中不一定包含抽象方法，但是有抽象方法的类必定是抽象类。
- 3. 抽象类中的抽象方法只是声明，不包含方法体，就是不给出方法的具体实现也就是方法的具体功能。
- 4. 构造方法，类方法（用 static 修饰的方法）不能声明为抽象方法。
- 5. 抽象类的子类必须给出抽象类中的抽象方法的具体实现，除非该子类也是抽象类。

接口

Java 接口

接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在 .java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在 .class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别：

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多继承。

接口特性

- 接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 `public abstract`（只能是 `public abstract`，其他修饰符都会报错）。
- 接口中可以含有变量，但是接口中的变量会被隐式的指定为 `public static final`变量（并且只能是 `public`，用 `private` 修饰会报编译错误）。
- 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

抽象类和接口的区别

- 1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
- 2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的。
- 3. 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。
- 4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

接口的声明

接口的声明语法格式如下：

```
[可见度] interface 接口名称 [extends 其他的接口名] {
    // 声明变量        // 抽象方法
}
```

Interface关键字用来声明一个接口。下面是接口声明的一个简单例子。

NameOfInterface.java 文件代码：

```
/* 文件名 : NameOfInterface.java */
import java.lang.*;
//引入包
public interface NameOfInterface {
    //任何类型 final, static 字段   //抽象方法
}
```

接口有以下特性：

- 接口是隐式抽象的，当声明一个接口的时候，不必使用**abstract**关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要**abstract**关键字。
- 接口中的方法都是公有的。

实例

Animal.java 文件代码：

```
/* 文件名 : Animal.java */
interface Animal {
    public void eat();
    public void travel();
}
```

接口的实现

当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。

类使用implements关键字实现接口。在类声明中，Implements关键字放在class声明后面。

实现一个接口的语法，可以使用这个公式：

Animal.java 文件代码：

```
...implements 接口名称[, 其他接口名称, 其他接口名称..., ...] ...
```

实例

MammalInt.java 文件代码：

```
/* 文件名 : MammalInt.java */ public class MammalInt implements Animal{
    public void eat(){
        System.out.println("Mammal eats");
    }
    public void travel(){
        System.out.println("Mammal travels");
    }
    public int noOfLegs(){
        return 0;
    }
    public static void main(String args[]){
        MammalInt m = new MammalInt();
        m.eat();
```

```
        m.travel();
    }
}
```

以上实例编译运行结果如下:

```
Mammal eats
Mammal travels
```

重写接口中声明的方法时，需要注意以下规则：

- 类在实现接口的方法时，不能抛出强制性异常，只能在接口中，或者继承接口的抽象类中抛出该强制性异常。
- 类在重写方法时要保持一致的方法名，并且应该保持相同或者相兼容的返回值类型。
- 如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口的时候，也要注意一些规则：

- 一个类可以同时实现多个接口。
- 一个类只能继承一个类，但是能实现多个接口。
- 一个接口能继承另一个接口，这和类之间的继承比较相似。

接口的继承

一个接口能继承另一个接口，和类之间的继承方式比较相似。接口的继承使用extends关键字，子接口继承父接口的方法。

下面的Sports接口被Hockey和Football接口继承：

```
// 文件名: Sports.java
public interface Sports {
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name); }
// 文件名: Football.java
public interface Football extends Sports {
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter); }
// 文件名: Hockey.java
public interface Hockey extends Sports {
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
    public void overtimePeriod(int ot);
}
```

Hockey接口自己声明了四个方法，从Sports接口继承了两个方法，这样，实现Hockey接口的类需要实现六个方法。

相似的，实现Football接口的类需要实现五个方法，其中两个来自于Sports接口。

接口的多继承

在Java中，类的多继承是不合法，但接口允许多继承。

在接口的多继承中extends关键字只需要使用一次，在其后跟着继承接口。 如下所示：

```
public interface Hockey extends Sports, Event
```

以上的程序片段是合法定义的子接口，与类不同的是，接口允许多继承，而 Sports及 Event 可能定义或是继承相同的方法

标记接口

最常用的继承接口是没有包含任何方法的接口。

标记接口是没有任何方法和属性的接口.它仅仅表明它的类属于一个特定的类型,供其他代码来测试允许做一些事情。

标记接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：java.awt.event包中的 MouseListener接口继承的 java.util.EventListener 接口定义如下：

```
package java.util;
public interface EventListener {}
```

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

- 建立一个公共的父接口：

  正如EventListener接口，这是由几十个其他接口扩展的Java API，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了EventListener接口，Java虚拟机(JVM)就知道该接口将要被用于一个事件的代理方案。
- 向一个类添加数据类型：

  这种情况是标记接口最初的目的，实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

内部类

简单来说就是内部定义一个类。

我会在接下来的《Java编程思想》阅读笔记中简述这个方法。

有一个有意思的特性是我可以自由使用外部类的变量。

匿名类

Java 中可以实现一个类中包含另外一个类，且不需要提供任何的类名直接实例化。

主要是用于在我们需要的时候创建一个对象来执行特定的任务，可以使代码更加简洁。

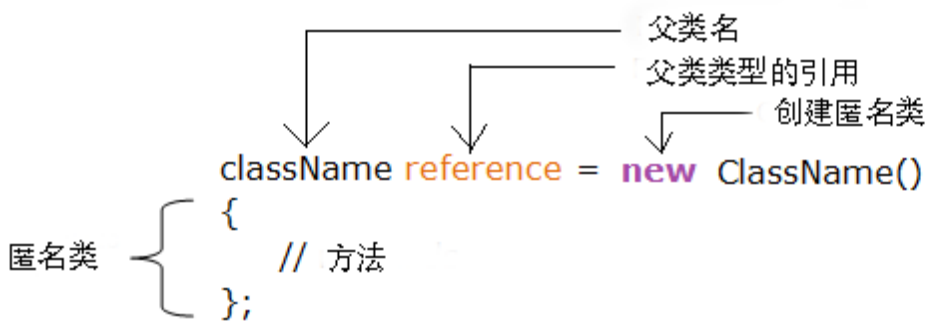
匿名类是不能有名字的类，它们不能被引用，只能在创建时用 **new** 语句来声明它们。

匿名类语法格式：

```
class outerClass {
    // 定义一个匿名类
    object1 = new Type(parameterList) {
        // 匿名类代码
    };
}
```

以上的代码创建了一个匿名类对象 object1，匿名类是表达式形式定义的，所以末尾以分号 ; 来结束。

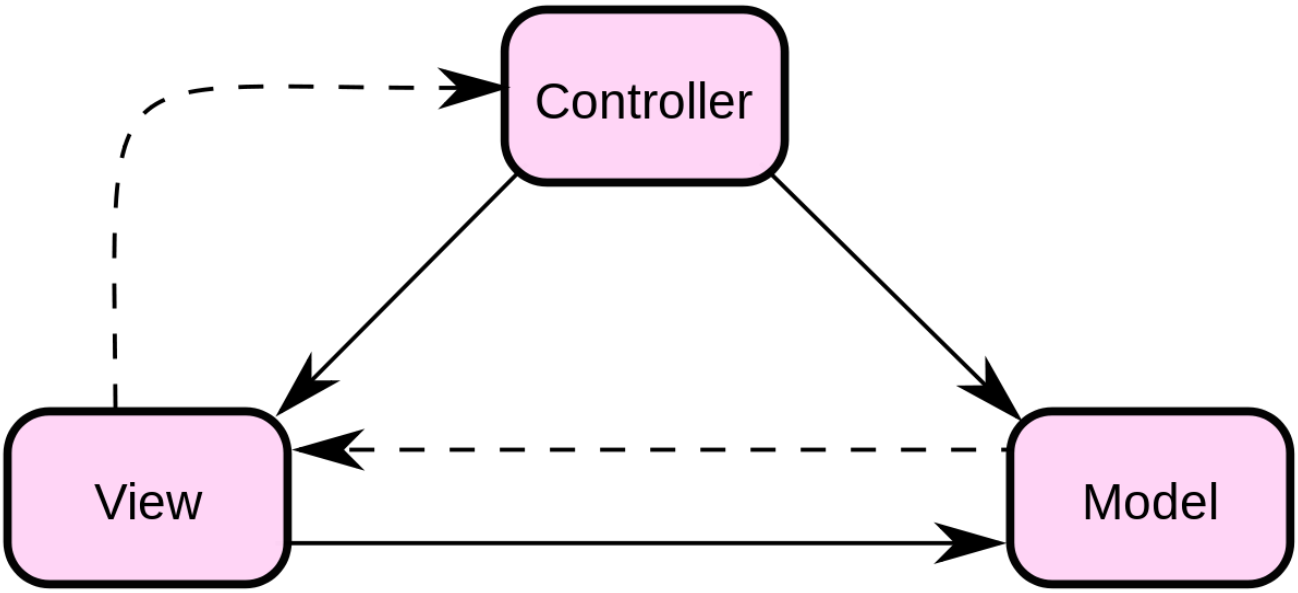
匿名类通常继承一个父类或实现一个接口。



MVC设计模式

MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

- **Model（模型）** - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
- **View（视图）** - 视图代表模型包含的数据的可视化。
- **Controller（控制器）** - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。



但应注意一点，控制器并不直接修改view的内容，与view并无直接关系。

IoC

作者：咔啡
链接：https://www.zhihu.com/question/381216328/answer/1093857745
来源：知乎

IoC不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。

传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活

IoC很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找

DI—Dependency Injection，即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

- 谁依赖于谁：当然是应用程序依赖于IoC容器；
- 为什么需要依赖：应用程序需要IoC容器来提供对象需要的外部资源；
- 谁注入谁：很明显是IoC容器注入应用程序某个对象，应用程序依赖的对象；
- 注入了什么：就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）

本质上IoC和DI是同一思想下不同维度的表现 ,用通俗的话说就是，IoC是bean的注册，DI是bean的初始化。

异常

捕获异常

使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。

try/catch代码块中的代码称为保护代码，使用 try/catch 的语法如下：

```
try
{
   // 程序代码
}catch(ExceptionName e1)
{
   //Catch 块
}
```

Catch 语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，try 后面的 catch 块就会被检查。

如果发生的异常包含在 catch 块中，异常会被传递到该 catch 块，这和传递一个参数到方法是一样。

Java 内置异常类

Java 语言定义了一些异常类在 java.lang 标准包中。

标准运行时异常类的子类是最常见的异常类。由于 java.lang 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。

Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

异常	描述
`ArithmeticException`	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
`ArrayIndexOutOfBoundsException`	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
`ArrayStoreException`	试图将错误类型的对象存储到一个对象数组时抛出的异常。
`ClassCastException`	当试图将对象强制转换为不是实例的子类时，抛出该异常。
`IllegalArgumentException`	抛出的异常表明向方法传递了一个不合法或不正确的参数。
`IllegalMonitorStateException`	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
`IllegalStateException`	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。
`IllegalThreadStateException`	线程没有处于请求操作所要求的适当状态时抛出的异常。
`IndexOutOfBoundsException`	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
`NegativeArraySizeException`	如果应用程序试图创建大小为负的数组，则抛出该异常。
`NullPointerException`	当应用程序试图在需要对象的地方使用 `null` 时，抛出该异常
`NumberFormatException`	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
`SecurityException`	由安全管理器抛出的异常，指示存在安全侵犯。
`StringIndexOutOfBoundsException`	此异常由 `String` 方法抛出，指示索引或者为负，或者超出字符串的大小。
`UnsupportedOperationException`	当不支持请求的操作时，抛出该异常。

下面的表中列出了 Java 定义在 java.lang 包中的检查性异常类。

异常	描述
`ClassNotFoundException`	应用程序试图加载类时，找不到相应的类，抛出该异常。

异常	描述
CloneNotSupportedException	当调用 Object 类中的 clone 方法克隆对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 Class 类中的 newInstance 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

异常方法

下面的列表是 Throwable 类的主要方法:

序号	方法及说明
1	public String getMessage() 返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。
2	public Throwable getCause()返回一个Throwable对象代表异常原因。
3	public String toString()使用getMessage()的结果返回类的串级名字。
4	public void printStackTrace() 打印toString()结果和栈层次到System.err，即错误输出流。
5	public StackTraceElement [] getStackTrace() 返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。
6	public Throwable fillInStackTrace()用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。

throws/throw 关键字:

如果一个方法没有捕获到一个检查性异常，那么该方法必须使用throws 关键字来声明。throws 关键字放在方法签名的尾部。

也可以使用 throw 关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

下面方法的声明抛出一个 RemoteException 异常:

```
import java.io.*;
public class className {
    public void deposit(double amount) throws RemoteException  {
        // Method implementation
        throw new RemoteException();
    }
    //Remainder of class definition
}
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

例如，下面的方法声明抛出 RemoteException 和 InsufficientFundsException：

```
import java.io.*;
public class className {
    public void withdraw(double amount) throws RemoteException, InsufficientFundsException   {
        // Method implementation
    }
    //Remainder of class definition
}
```

---

finally关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句。

finally 代码块出现在 catch 代码块最后，语法如下：

```
try{  // 程序代码
}catch(异常类型1 异常的变量名1){  // 程序代码
}catch(异常类型2 异常的变量名2){  // 程序代码
}
finally
{
    // 程序代码
}
```

实例

ExcepTest.java 文件代码：

```
public class ExcepTest{
    public static void main(String args[]){
        int a[] = new int[2];
        try{
            System.out.println("Access element three :" + a[3]);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Exception thrown  :" + e);
```

```
        }
        finally{
            a[0] = 6;
            System.out.println("First element value: " +a[0]);
            System.out.println("The finally statement is executed");
        }
    }
}
```

以上实例编译运行结果如下：

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

注意下面事项：

- catch 不能独立于 try 存在。
- 在 try/catch 后面添加 finally 块并非强制性要求的。
- try 代码后不能既没 catch 块也没 finally 块。
- try, catch, finally 块之间不能添加任何代码。

异常声明遇到继承关系

- 当覆盖一个函数的时候，子类不能声明抛出比父类版本更多的异常
- 在子类的构造函数中，必须声明父类可能抛出的全部异常。

Java泛型与容器初探

接下来将是非常无聊、枯燥但绝对对你未来的程序的理解有帮助的东西。

摘抄自[https://www.cnblogs.com/mutojack/p/9160342.html]

一、泛型

“泛型”的意思是“适用于许多许多的类型”，实现了参数化类型的概念。其最初的目的是希望类或方法具备最广泛的表达能力，通过解耦类或方法与所使用的类型之间的约束。不用像参数是类或接口那样对程序有过多约束（方法的参数不必仅限于一种类或接口与它们的子类）

使用泛型，具体来说，在定义一个类的时候，类名后面加上<T>这个*类型参数*，那么在类中，可以用T来表示不特定的数据类型。在创建该泛型实例时，将T换成你所需要的具体的数据类型。这个数据类型不能是基本类型。

```
class A<T> {
    private T v;
    public T get()
    {
        return T;
    }
}
//......
A<Integer> temp = new A<Integer>;
Integer result = temp.get();
```

> 告诉编译器想使用什么类型，然后编译器帮你处理一切细节。

```
import java.util.*;

public class C{
    public static void main(String[] args){
        List list1 = new ArrayList();
      list1.add("heheh")
      list1.add(35.5);
      for(int i = 0; i < list1.size();i++){
            Object s = list1.get(i);
            System.out.println(s);
        }
     }
}
```

对这段代码，编译器会报使用了未经检查或不安全的操作。注: 有关详细信息, 请使用 -Xlint:unchecked 重新编译。

但是任然可以运行。

```
import java.util.*;
public class C {
    public static void main(String[] args){
    List<String> list2 = new ArrayList<String>();
    list2.add("heheh");
    list2.add("wuwuw");
    //list2.add(35.5);
    for(int i = 0; i < list2.size();i++){
        Object s = list2.get(i);
        System.out.println(s);
        }
    }
}
```

而对于这段代码，添加了<String>类型参数以后，如果依然add(35.5)这个double类型的数，就会报错无法运行。因为加入的double类型与你已经确定的<String>类型冲突了，相当于你给add(String str)方法传入了一个double类型的实参。

对于接口来说，使用泛型的方式也是一样的。(List实际上就是一个接口的栗子)

接口使用泛型的一个例子是生成器，generator是*工厂设计模式*的一种应用，专门负责创建对象。一般而言，一个生成器只定义一个方法，该方法用以产生新对象。

```
public interface Generator<T>{
    T next();//返回一个T类型的对象。
}
```

当你需要一个生成器时，就可以：

```
public class AGenerator implements Generator<A>{
    public A next(){
    //......
    }
}
```

当然，不仅仅类和接口可以实现泛型，**方法**也可以。以下是一个基本的指导原则：无论何时，只要你能做到，你就应该尽量使用泛型方法。要定义泛型方法，需将泛型参数置于返回值之前。与泛型类中的方法相比，区别在于只在该方法中出现了类型参数（或者在泛型类中，方法里出现了新的类型参数，需要把方法定义为泛型方法？ 没有考证，想当然的个人理解）。

```
public class Lalala{
    public <T> void f(T x){
    System.out.println(x.getClass().getName());
    }
}
```

注意，对于一个static的方法而言，无法访问泛型类的类型参数，所以，如果static方法需要使用泛型能力，就必须使其成为泛型方法。**也就是说，如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法** 。

```
class Lalala<T>{
    static <T> void xixixi(T t){}
}
```

如果你希望有多个不同种类的参数的话，可以将扩展成<A,B,C,D>这样的形式。

通配符：`<? extends A>`，这个类型参数可以代表A与所有继承了A的类型，而`<? super A>` （*超类型通配符*）则代表了所有A与A的超类作为类型参数。更特殊的还有无界通配符 `<?>`，表示： *我是想用Java的泛型来编写这段代码，我在这里并不要用原生类型，但是在当前这种情况下，泛型参数可以持有任何类型。* 比如`List`(相当于`List<object>`)和`List<?>`，前者表示持有任何Object类型的原生List，后者表示 具有某种特定类型的非原生List，只是我们不知道那种类型是什么。

（在定义对象时）这并不是说通配符限制了这个参数的范围，恰恰相反，通配符反而让不同类之间可以交流：

```
Class A{}
Class B extends A{}
Class G<T>{}
//下面这行就会报错，尽管B的确是A的子类:不兼容的类型: C<B>无法转换为C<A>
//G<A> g = new G<B>();
```

但是如果改成这样

```
G<? extends A> g = new G<B>();
```

就不会报错。

而当在定义类或方法时使用了通配符，那就限制了传入参数的类型。

```
public class G<T extends A>{
    private T key;
}
```

- tip:*如果你在**定义**类的时候使用了class A<T extends E>，那你就真的限制了上界或者下界，注意与通配符区分。*

尽管如此，以List作为G为例，你还是不能对g使用add()这样的方法（除了add(null)），因为编译器不知道g当时指向的会是哪一种具体的类型。你的g对象中可能有A的任何子类，由于向上转型，到底是什么子类没有人关心。

> 对于参数值是未知类型的容器类，只能读取其中元素，不能向其中添加元素， 因为，其类型是未知，所以编译器无法识别添加元素的类型和容器的类型是否兼容，唯一的例外是NULL

那么如何添加元素进去？只要那个函数接受的参数是Object就行了。对于上述的add例子，它的参数随着你创建g时变成了? extends A，从这个描述中，编译器不能了解具体需要的是那个子类，所以不会接受任何类型的A。而对于接受参数是Object类型的函数，不涉及任何通配符，编译器就允许这个调用，add(Object o)这样就可以了。

对于<? super T>：与上述例子对于，它的get方法你不清楚返回的是哪个类型。

有一种情况特别需要使用<?>而不是原生类型。如果向一个使用<?>的方法传递原生类型，那么对编译器来说，可能会推断出实际的类型参数，使得这种方法可以回转并调用另一个使用这个确切类型的方法。这种技术成为捕获转换。捕获转换只有在这样的情况下可以工作：即在方法内部，你需要使用确切的类型。比如说f2()使用的是<?>，但是在f2()中调用了需要确切类型的函数f1()，这时候就会使用到捕获转换。

擦除特性：

> 在泛型代码内部，无法获得任何有关泛型参数类型的信息。

不管你的类型参数是什么，SomeClass<A>和SomeClass<B>使用getClass得到的是一样的。

如果你希望在SomeClass中调用A类的方法，会报错，因为编译器根本不知道你的类型参数A自己带有的方法。

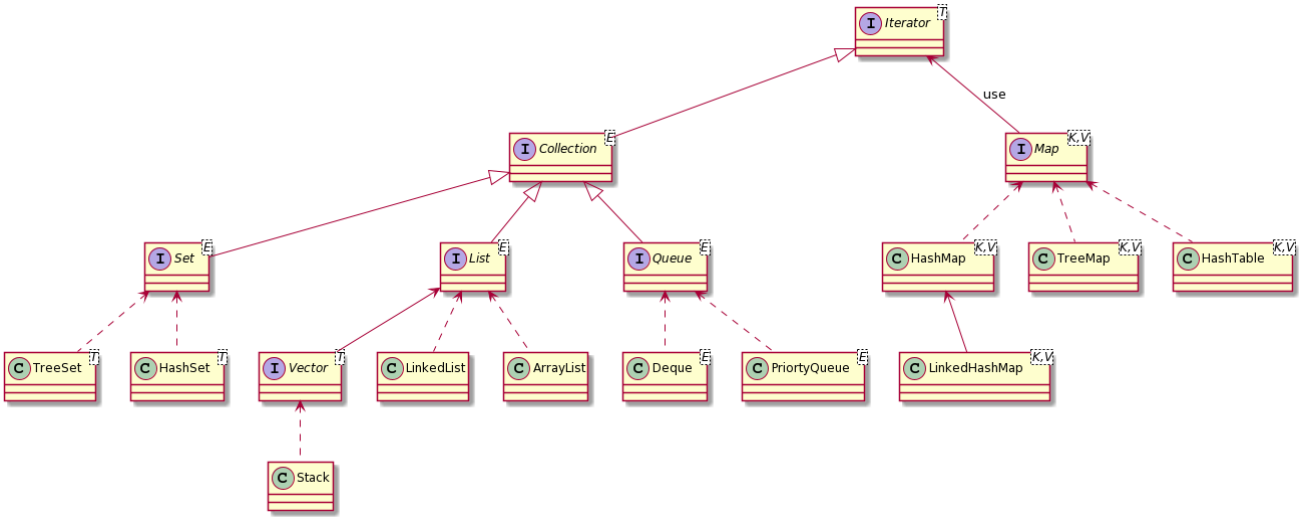
二、容器

> 事先注明，有些接口没有介绍，所以有些方法要自己看看相应的类实现了哪些接口才知道

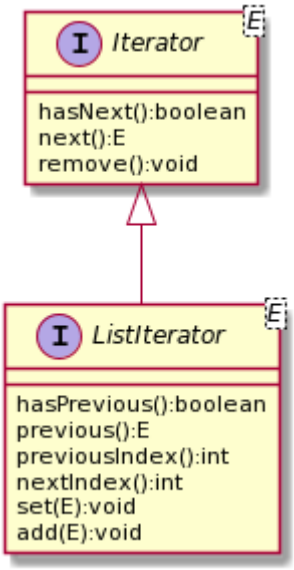
容器的用途是“保存对象”。包括Collection,Map两大类等。

类图关系大致 如图，但是实际上并不严谨，有些其他关系和抽象类什么的没有画出，在此仅为了表示一个直观的大概的印象，方便初学者理清层次关系：

（另，有*画错*的地方，Deque应该是Queue的子接口，并且LinkedList同时也继承并实现了它）



在介绍容器之前，先了解一下容器中存在的一个比较重要的成员：Iterator。



Iterator接口继承了顶级接口Iterable，通常用来遍历序列中的对象。

以Collection为例，它的用法是，当Collection的子类对象调用了iterator()方法，返回一个Iterator对象。该Iterator对象第一次调用next方法时，会获得（返回）Collection对象中的第一个元素。（可以把迭代器理解成一个指针，每次调用next方法，就指向下一个元素）

遍历时，通常用it.next()来表示遍历过程中的对象。

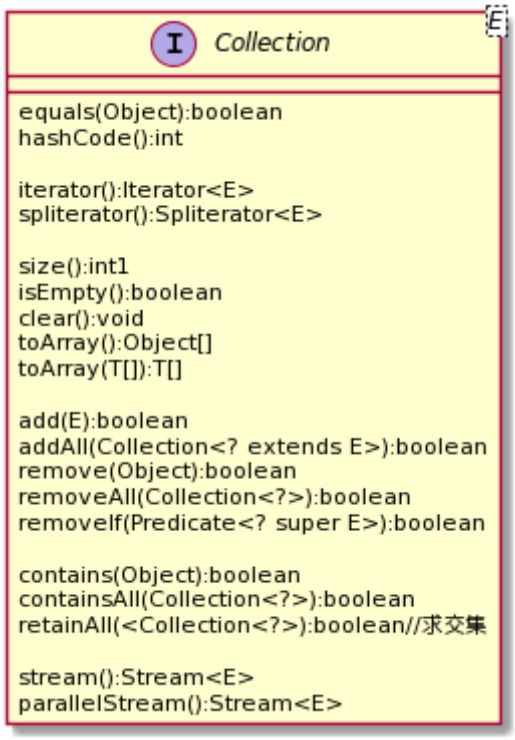
而在List中有一个listIterator方法，返回一个ListIterator对象，从类图中看，它可以双向遍历，也多了一些功能。要注意的是，add方法将指定元素插入当前位置之前。

（如果在遍历过程中，用了remove方法，是不影响遍历的，可以正常调用next等方法）。

除了ListIterator以外，还有可分割迭代器spliterator和倒序迭代器descendingIterator的存在。

一、Collection<T>

Collection<T>是一个泛型接口，List<T>,Set<T>是继承它的接口，再往下就分别是实现了List的LinkedList，ArrayList（以及Vector接口和实现了Vector的Stack），还有实现了Set<T>的TreeSet(实际上TreeSet的上层是SortedTree<T>，SortedTree<T>是Set<T>的子接口)和HashSet。



`Object`类中的`equals(Object)`方法用于检测一个对象是否等于另一个对象，在`Object`类中，这个方法判断的根据是两个对象是否具有相同的*引用*，如果是，那么就返回true。而`hashCode`方法（散列码）是由对象导出得到的一个整型值，不同的对象得到的散列码一般是不同的。由于`hashCode`方法定义在Object类中，因此每个对象都有一个默认的散列码，也就是对象的存储地址。如果需要覆写equals方法时，必须也覆写`hashCode`（也就是定义一个返回整型数的方法，这个整型数的产生方式要根据你的需要区分开不同的对象，比如调用该类中各个属性的`hashCode`方法然后做个运算）。`equals`和`hashCode`的定义必须一致：如果两个对象equals了，那么它们的散列码就必须具有相同的值。

> tips：在字符串类型中，散列码是由内容导出的，所以内容相同散列码也相同。而`StringBuffer`类中没有定义`hashCode`方法，它的散列码就还是对象的存储地址。Double啊什么的类都覆写了自己的根据内容比较的方法。

两个`toArray`方法注意返回值，`toArray()`返回的是该`Collection`里所有元素对象的`Object`类型的数组（因此有时要注意转化类型），而`toArray(<T>[])`里这个参数就是用来存储这些元素的数组，返回的也正是这个数组。

范例：

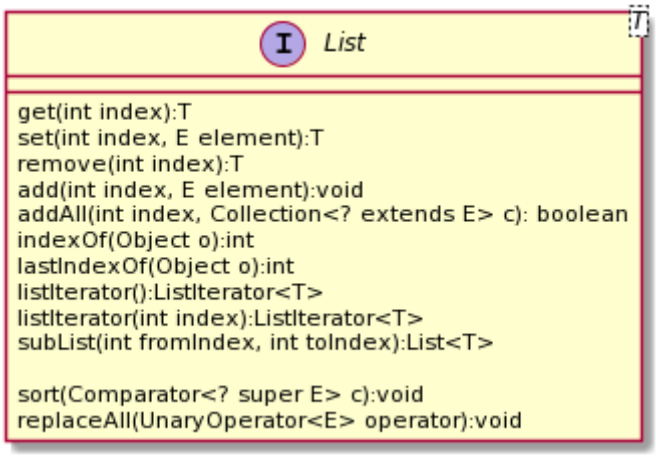
```
String[] y = x.toArray(new String[0])
/*@param  a the array into which the elements of this collection are to be stored, if it is big
enough; otherwise, a new array of the same runtime type is allocated for this purpose.*/
```

`removeIf()`方法移除的是符合参数格式的字符串。(有点python的列表解析时的感觉)

```
c.removeIf(obj -> obj%2 == 1)
```

最后强调一下，`Collection`接口中的方法，↓下面这些 都是有的，但是可能有些没有画在图里，也没有重复介绍了。

`List<T>`：特点是有序，元素有存入的前后之分，以线性方式存储。而且可以重复。



`listIterator(int index)`方法是从index位置开始产生一个`ListIterator`对象，也就是第一次调用`next()`时返回的对象。

`subList(int fromIndex,int toIndex)`，左闭右开区间，返回一个新的`List`，并不修改原来的`List`。

tip:

> 在遍历的过程中是不对List操作进行修改的，无论是删除和添加，因为如果在遍历中一直向集合中新增加元素，会造成死循环的，还有就是如果在遍历过程中删除元素，会造成数组下表越界等问题.

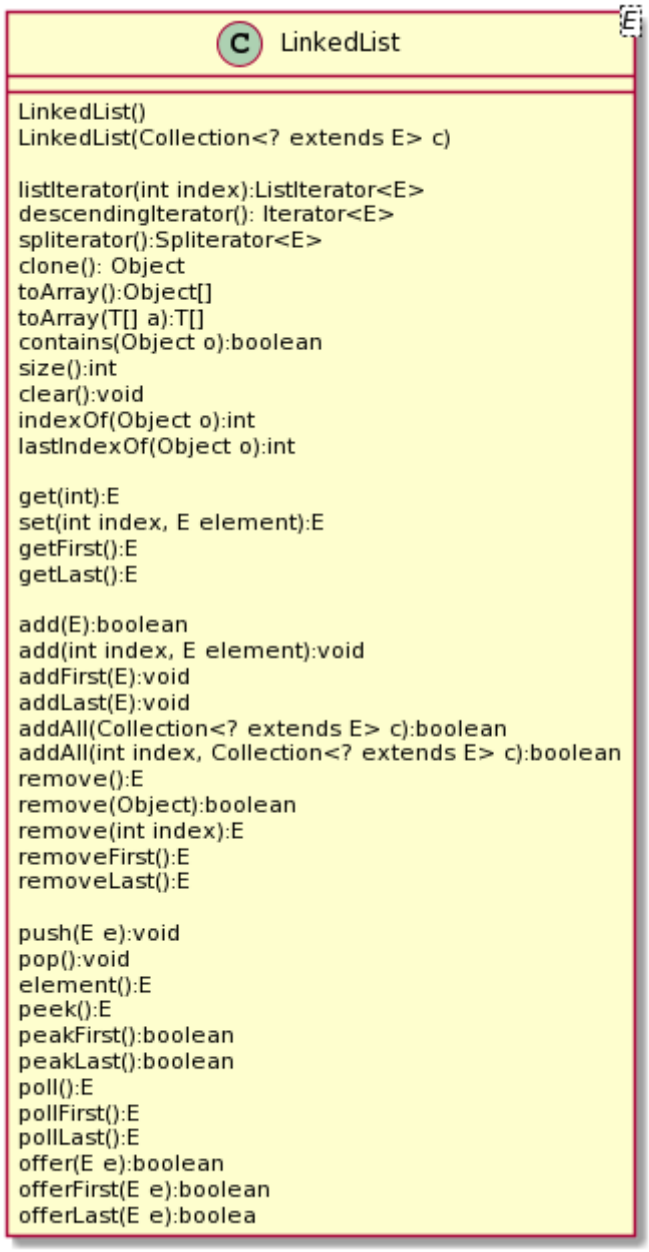
所以一种操作就是先找到要删的元素，放入一个`List`中，然后用`removeAll`方法。或者就直接用`removeIf`，亦或是用`Iterator`的`remove`方法在遍历过程中删除。

接下来重点介绍一下实现了`List`接口的两个类，

1. *`LinkedList(*√)`：底层数据结构是链表。线程不安全。把这个当成数据结构课自己实现过的链表用就行了，当你需要用链表时，建立一个`LinkedList`对象，链表的一般操作都已经帮你实现了。
   - All Implemented Interfaces:

     `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `Deque<E>`, `List<E>`, `Queue<E>`

   看实现了的接口，大概就知道`LinkedList`可以用来实现链表、队列和双端队列了



1. 方法还是挺多的，但是学过数据结构，基本看名字就知道有什么用了，也没什么需要注意的。

   `clone`返回的是浅拷贝`@return a shallow copy of this {@code LinkedList} instance`。

   `peek`和`element`的效果是基本一样的，返回第一个元素，只是如果没有元素时，peek返回null，element抛出一个异常。

   `poll`和`remove`一样，都是移除并返回第一个元素，但是为空时，前者返回null，后者抛异常。

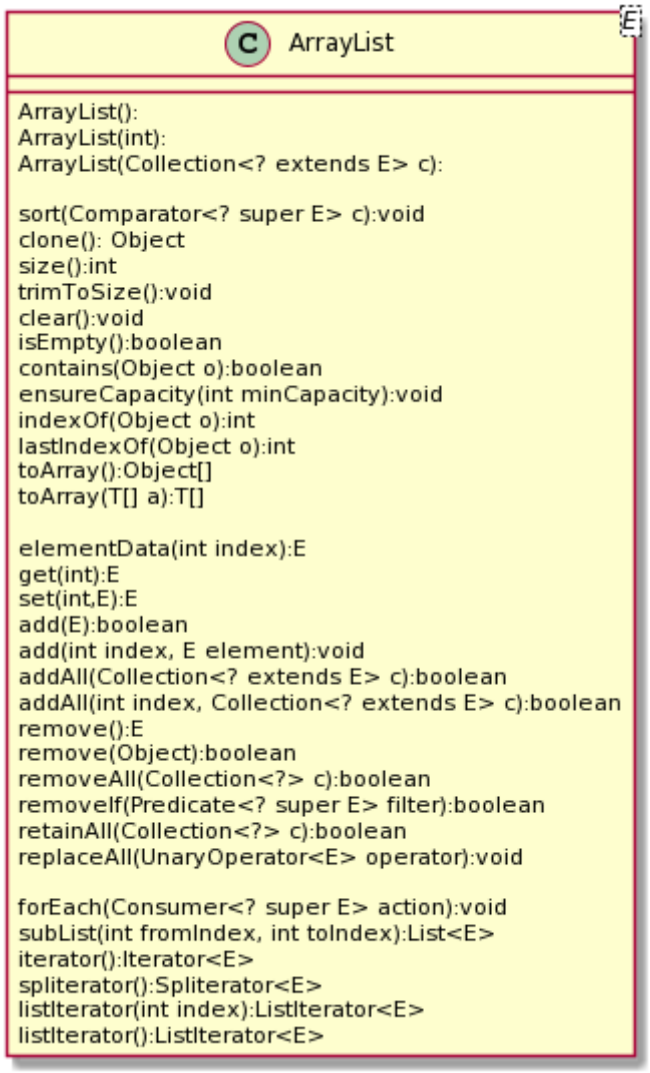
   `add`和`offer`一样，都是把一个元素加到最后，为空的时候，使用add方法会报错，而offer方法会返回false。这两个方法分别来自List和Queue（大概也可以根据自己要用`LinkedList`干什么来选方法）

   `pop`是加一个元素到`front of the list, push`就是返回并移除开头的元素。

2. *ArrayList*(√)：底层数据结构是数组。线程不安全。粗略的把它当作一个能够长度不定，操作灵活的数组就行了。

   - All Implemented Interfaces:

     `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`



1. 1. 说白了就是把数组当成一个类，然后加入了一些对“数组”的方法操作。不过，很自然就能想到，便利的代价就是效率更低。

   `sort`方法就是传入一个比较器参数，把`ArrayList`里的参数按比较器来排序。

   `clone:@return a clone of this ArrayList instance`,是浅拷贝。

   `trimToSize`方法可以修剪`ArrayList`的容量，因为`ArrayList`会预留一些空间，用这个方法可以删除多余的。

   手动扩容： `ensureCapacity(int minCapacity)`。

   `forEach(Consumer<? super E> action)` 对每个元素执行action操作。

   2. Vector：底层数据结构是数组。线程安全。Vector是List的子接口，和Queue是一个层次上的，而与实现Queue的Deque相对应的是，实现了Vector的Stack类。但是据说过时了，所以不介绍了，了解有这样一个东西就好。

2. **Set**：**元素不可重复** 。同时没有链表按序的特性。它最常被使用的是测试归属性（归属 性），可以很容易地询问某个对象是否在某个Set中，因此，查找就成了Set中最重要的操作。Set具有和Collection完全一样的接口，没有额外的功能。所以把Collection的类图，名字换成Set就行了。根据编程思想里的原话：

   > 实际上Set就是Collection，只是行为不同。

   （像`LinkedList`这些子接口会重复声明父接口的部分方法，大概也是一样的感觉？）

   Set是基于对象的值来确定归属性的。

   重点来看下面两种实现类。

   1. `*TreeSet*`：基于`TreeMap`实现，其底层数据结构是红黑树(是一个自平衡的二叉树)，保证元素的*排序*方式，默认是自然顺序(`Comparable`)，也可以自定义比较器顺序(`Comparator`)。

      - All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, NavigableSet<E>, Set<E>, SortedSet<E>

实现了NavigableSet接口意味着它支持一系列的查找、定位的操作。

1. TreeSet(NavigableMap\<E,Object> m) 构造方法通过传递的参数可以扩展对于边界查询的方法。

   TreeSet(Comparator<? super E> comparator) 传递的是一个Comparator参数，Comparator是一个接口，通过实现这个接口，可以传递一个自定义的排序的方式。元素将根据这个比较器进行排序。（一般是你传入的自定义类，实现了Comparator接口，从而达到给该类元素排序的目的，后面的PriorityQueue一样的道理）

   TreeSet(Collection<? extends E> c)，创建了包含了Collection参数中所含有的元素的对象。

   TreeSet(SortedSet\<E> set)，Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.和上面一个差不多但是连顺序也记录。

   public SortedSet\<E> subSet(E fromElement, E toElement) 调用了另一个subSet函数，return subSet(fromElement, true, toElement, false);

   headSet(E toElement) 得到一个以 toElement为上界（不包括）的Set(Returns a view of the portion of this set whose elements are strictly less than toElement.)，而 headSet(E toElement, boolean inclusive) Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement.，可以看出后面这个inclusive参数就是决定是否要包括toElement。

   tailSet类似，只不过是决定下界的。

   subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)，则是返回处于上界下界之中的Set，两个Inclusive就不用解释了把。subSet(E fromElement, E toElement) 对于这个方法来说，默认是fromInclusive是true的，而toInclusive是false的，也就是，含头不含尾。

   由于TreeSet是有序的，所以就有first()方法返回the first (lowest) element，和它对应的是last()方法 。

   而floor返回的是比floor的参数更小或相等的元素中最大的那个（因为它是地板嘛）。注意和lower的对比，lower是不包含相等的。 也就是 floor(E e)返回的是 element <= e 中最大的,lower则是 element < e。 与floor对应的是ceiling方法。

对于pollFirst和pollLast，看到poll就应该猜到了，(Retrieves and removes the first (lowest) /last(highest) element, or returnsnullif this set is empty.)也就是说检索到以后还要把它取出来。

TreeSet的clone方法返回的是：a shallow copy of this set浅拷贝。

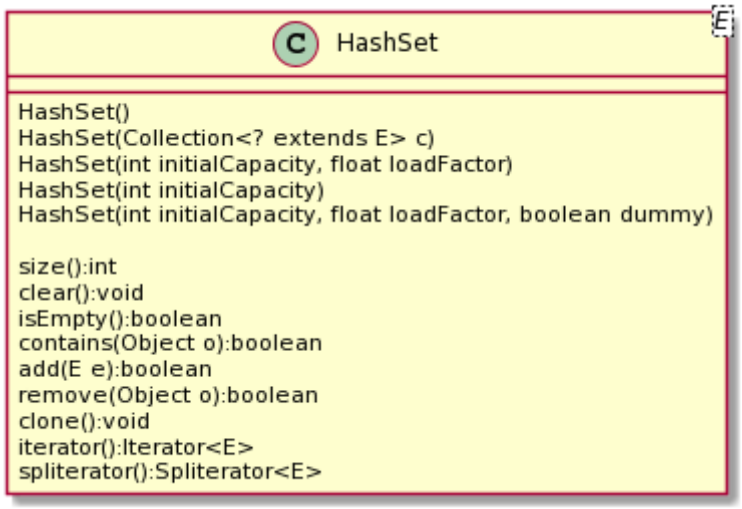
2. *HashSet*：底层数据结构是哈希表(是一个 元素 为链表 的 数组) 。和TreeSet不同，它的输出顺序是随机的。

   - All Implemented Interfaces:

     Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>

   HashSet存储的对象存取都是哈希值，自身先后通过hashcode()和equals()方法来判断元素是否重复。哈希值就是元素的身份证（map中的Key)。当hashCode值不相同，就直接存储了，当hashCode值相同时，会在判断一次euqals方法的返回值是否为true，如果为true则视为用一个元素，不用存储，如果为false，这些相同哈希值不同内容的元素都存放一个桶里（当哈希表中有一个桶结构，每一个桶都有一个哈希值）

   在添加自定义对象的时候，两个类的属性值相等，但是依然会被判定为不同的元素，因为没有重写hashCode()，所以默认调用的是Object类的hashCode()，而不同类的hashCode一般是不同的。所以当你以自定义类对象为类型参数的时候，记得要重写。



1. 1. 构造方法：

   HashSet构造的默认容量(initialCapacity)是16。

   HashSet(int initialCapacity, float loadFactor)，前一个是初始容量可以理解，后一个load factor，加载因子（表示元素填满的程度）。加载因子越高，空间开销越小，但是查找的成本就会提高；反之，空间开销大，查找的成本低。对于不用手动设置的三个初始化方式，它们的默认加载因子是0.75，是一个折中的值。（那个带dummy的也是没看懂）

   具体的解释找到这一段：

   > The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets. As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

   还有这个(讲HashMap的一篇很全面的博客：https://www.cnblogs.com/skywang12345/p/3310835.html)

> 容量 是哈希表中桶的数量，初始容量 只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。通常，**默认加载因子是 0.75**, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数 `HashMap` 类的操作中，包括 get 和 put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作。

3.Queue：队列的特征是先进先出，这部分功能在`LinkedList`的方法里实现了。（别忘了`LinkedList`也实现了`DeQue`接口，需要用到普通的队列功能的时候别搞错了）。

就这么几种方法。和`LinkedList`对比一下，就可以知道哪部分是实现Queue的，当用作一个队列的时候该用哪些方法。

而和`Collection`相比：`provide additional insertion, extraction, and inspection operations.`

1. Deque子接口

- All Superinterfaces:

  `Collection<E>`, `Iterable<E>`, `Queue<E>`

主要方法如下：

> This interface extends the `Queue` interface. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the `Queue` interface are precisely equivalent to `Deque`methods as indicated in the following table:
>
> | Queue Method | Equivalent Deque Method |
> | --- | --- |
> | `add(e)` | `addLast(e)` |
> | `offer(e)` | `offerLast(e)` |
> | `remove()` | `removeFirst()` |
> | `poll()` | `pollFirst()` |
> | `element()` | `getFirst()` |
> | `peek()` | `peekFirst()` |
>
> Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy `Stack` class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Stack methods are precisely equivalent to `Deque` methods as indicated in the table below:
>
> | Stack Method | Equivalent Deque Method |
> | --- | --- |
> | `push(e)` | `addFirst(e)` |
> | `pop()` | `removeFirst()` |
> | `peek()` | `peekFirst()` |

从摘自API文档的这段介绍来看，当你需要FIFO结构时，意味着元素从队尾进，从队首出，就用继承自Queue接口的方法，而这些方法有了含义更清晰的表达，也就是表格中的对应关系，`addLast`把新来的元素放到队尾，效果和Queue的add一样的。而当你需要LIFO的栈结构， 下面那个也是一样的。

另外还两个方法：`removeFirstOccurrence(Object o)`和`removeLastOccurrence(Object o)`，看方法名就知道是什么了。

`PriortyQueue` :优先队列

- All Implemented Interfaces:

  `Serializable`, `Iterable`, `Collection`, `Queue`

Constructor	Description
`PriorityQueue()`	Creates a `PriorityQueue` with the default initial capacity (11) that orders its elements according to their natural ordering.
`PriorityQueue(int initialCapacity)`	Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to their natural ordering.
`PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`	Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to the specified comparator.
`PriorityQueue(Collection<? extends E> c)`	Creates a `PriorityQueue` containing the elements in the specified collection.
`PriorityQueue(Comparator<? super E> comparator)`	Creates a `PriorityQueue` with the default initial capacity and whose elements are ordered according to the specified comparator.
`PriorityQueue(PriorityQueue<? extends E> c)`	Creates a `PriorityQueue` containing the elements in the specified priority queue.
`PriorityQueue(SortedSet<? extends E> c)`	Creates a `PriorityQueue` containing the elements in the specified sorted set.

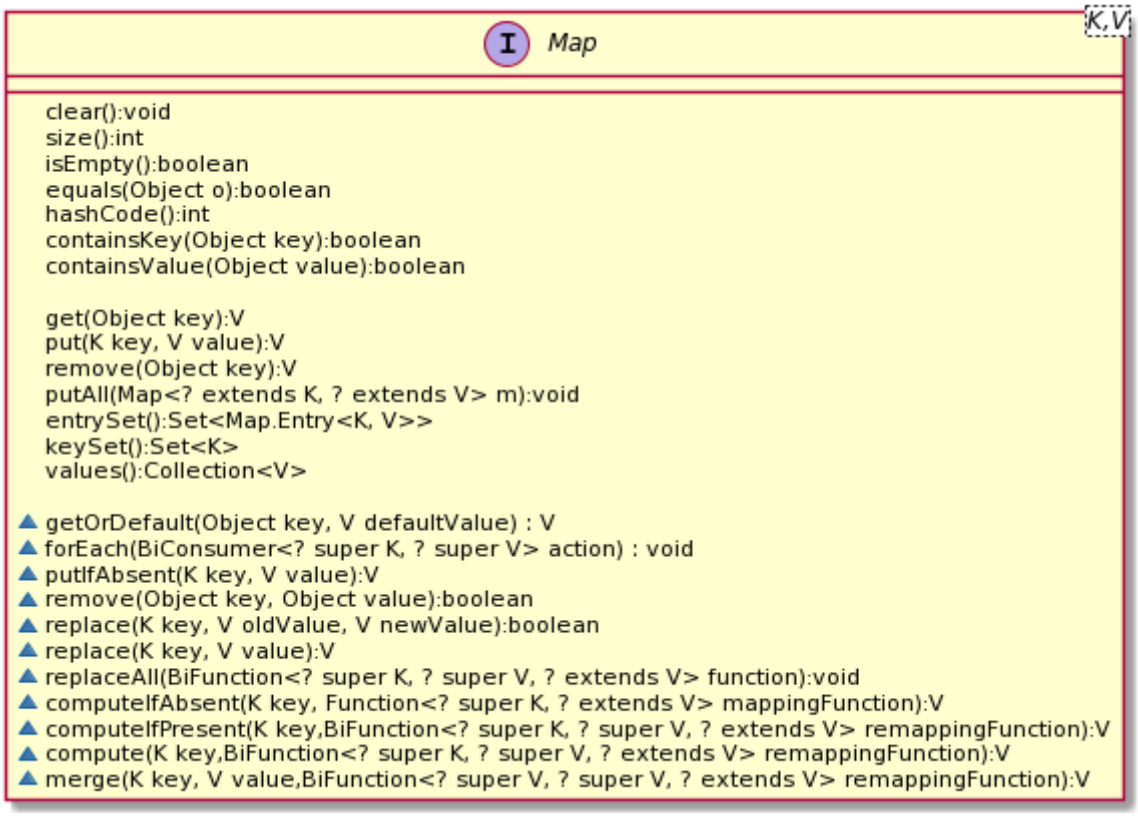
可以看出和`TreeSet`有相似的地方，那就是可以传入`Comparator`参数。而这就是`PriorityQueue`的特点。`Comparator`是一个接口，根据`Comparator`中实现的规则，可以改变你入列以后的排序顺序。比如说你希望有一个按官职大小排队的队伍，在`Comparator`中实现了，传参进来，然后来了个县令，队里只有一个人，又来了个捕头，排第二没毛病（这个排序不是因为FIFO，而是因为他官职更小），这时候来了个尚书，因为他官职最大，所以他就排队到第一。出列的时候，没有变，队首出。这就是`PriorityQueue`。

其余的方法，看实现的接口就知道了，基本就是`Queue`的方法。

要说的话，`public Comparator<? super E> comparator() { return comparator;}` 这个方法可以返回该队列的`Comparator`属性（`private final Comparator<? super E> comparator`）。

二、*Map*

`Map`是和`Collection`一个层次上的接口，它的结构是`key-value`。



K,V两个参数分别对应了key和value，这样基本懂了大部分方法了。

`entrySet`、`keySet`、`values`三个方法分别得到所有键值的Set（元素类型是Map的内部接口Entry，Entry提供了一些操作)、键的Set、值的Collection。

三个带compute的方法都是用来计算参数中的key对应的新value的。

`getOrDefault`返回key对应的value，如果key不存在的，就返回`defaultValue`。

`putIfAbsent`，如果不存在参数中的那个键值对或者值为空，就put。普通的put方法会直接覆盖。

`remove(Object key, Object value)` 移除的前提是二者都要与map中的键值对匹配，才能执行删除操作。

`merge`，如果参数中的键不存在或者值为空，就用value和它组成一对；如果存在的话，就用第三个参数计算出来的结果代替它。具体怎么用，和compute差不多，写的时候还没有试过……

HashMap 散列表

存储键值映射的数据,不保证映射的顺序。 可以和`reeMap`一起对比`HashSet`和`TreeSet`。

- All Implemented Interfaces:

  `Serializable`, `Cloneable`, `Map`

Constructor	Description
`HashMap()`	Constructs an empty `HashMap` with the default initial capacity (16) and the default load factor (0.75).
`HashMap(int initialCapacity)`	Constructs an empty `HashMap` with the specified initial capacity and the default load factor (0.75).
`HashMap(int initialCapacity, float loadFactor)`	Constructs an empty `HashMap` with the specified initial capacity and load factor.

Constructor	Description
`HashMap(Map<? extends K,? extends V> m)`	Constructs a new `HashMap` with the same mappings as the specified `Map`.

又是和Hash挂钩的，所以又能看见load factor。

clone方法照例 Returns a shallow copy

没有其他特别的方法。

`TreeMap`

底层实现基于红黑树。有序。

- All Implemented Interfaces:

  `Serializable`, `Cloneable`, `Map<K,V>`, `NavigableMap<K,V>`, `SortedMap<K,V>`

Constructor	Description
`TreeMap()`	Constructs a new, empty tree map, using the natural ordering of its keys.
`TreeMap(Comparator<? super K> comparator)`	Constructs a new, empty tree map, ordered according to the given comparator.
`TreeMap(Map<? extends K,? extends V> m)`	Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
`TreeMap(SortedMap<K,? extends V> m)`	Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

方法还挺多的，但是由于实现都基于红黑树，原理相同，所以基本可以和TreeSet对标。

`LinkedHashMap`

双向链表+`HashMap`

```
public class LinkedHashMap<K,V>
extends HashMap<K,V>
implements Map<K,V>
```

也就是说实现了`HashMap`的按存放顺序排列的功能。

附博客一篇：https://blog.csdn.net/justloveyou_/article/details/71713781

部分模块的使用及说明

Random类用法

Java中存在着两种Random函数：

java.lang.Math.Random

调用这个Math.Random()函数能够返回带正号的double值，该值大于等于0.0且小于1.0，即取值范围是[0.0,1.0)的左闭右开区间，返回值是一个伪随机选择的数，在该范围内（近似）均匀分布。例子如下：

```
package IO;
import java.util.Random;

public class TestRandom {

    public static void main(String[] args) {
        // 案例1
        System.out.println("Math.random()=" + Math.random());// 结果是个double类型的值，区间为[0.0,1.0）
        int num = (int) (Math.random() * 3); // 注意不要写成(int)Math.random()*3，这个结果为0，因为先执行了强制转换
        System.out.println("num=" + num);
        /**
         * 输出结果为：
         *
         * Math.random()=0.02909671613289655
         * num=0
         *
         */    }}
```

java.util.Random

下面Random()的两种构造方法：

Random()：创建一个新的随机数生成器。

Random(long seed)：使用单个 long 种子创建一个新的随机数生成器。

我们可以在构造Random对象的时候指定种子（这里指定种子有何作用，请接着往下看），如：`Random r1 = new Random(20);`

或者默认当前系统时间的毫秒数作为种子数:`Random r1 = new Random();`

需要说明的是：你在创建一个Random对象的时候可以给定任意一个合法的种子数，种子数只是随机算法的起源数字，和生成的随机数的区间没有任何关系。如下面的Java代码：

```
Random rand =new Random(25);
int i;
i=rand.nextInt(100);
```

初始化时25并没有起直接作用（注意：不是没有起作用）,rand.nextInt(100);中的100是随机数的上限,产生的随机数为0-100的整数,不包括100。

具体用法如下例：

```
package IO;

import java.util.ArrayList;
import java.util.Random;

public class TestRandom {

    public static void main(String[] args) {

        // 案例2
        // 对于种子相同的Random对象，生成的随机数序列是一样的。
        Random ran1 = new Random(10);
        System.out.println("使用种子为10的Random对象生成[0,10)内随机整数序列: ");
        for (int i = 0; i < 10; i++) {
            System.out.print(ran1.nextInt(10) + " ");
        }
        System.out.println();
        Random ran2 = new Random(10);
        System.out.println("使用另一个种子为10的Random对象生成[0,10)内随机整数序列: ");
        for (int i = 0; i < 10; i++) {
            System.out.print(ran2.nextInt(10) + " ");
        }
        /**
         * 输出结果为:
         *
         * 使用种子为10的Random对象生成[0,10)内随机整数序列:
         * 3 0 3 0 6 6 7 8 1 4
         * 使用另一个种子为10的Random对象生成[0,10)内随机整数序列:
         * 3 0 3 0 6 6 7 8 1 4
         *
         */

        // 案例3
        // 在没带参数构造函数生成的Random对象的种子缺省是当前系统时间的毫秒数。
        Random r3 = new Random();
        System.out.println();
        System.out.println("使用种子缺省是当前系统时间的毫秒数的Random对象生成[0,10)内随机整数序列");
        for (int i = 0; i < 10; i++) {
            System.out.print(r3.nextInt(10)+" ");
        }
        /**
         * 输出结果为:
         *
         * 使用种子缺省是当前系统时间的毫秒数的Random对象生成[0,10)内随机整数序列
         * 1 1 0 4 4 2 3 8 8 4
         *
         */

        // 另外，直接使用Random无法避免生成重复的数字，如果需要生成不重复的随机数序列，需要借助数组和集合类
        ArrayList list=new TestRandom().getDiffNO(10);
```

```
        System.out.println();
        System.out.println("产生的n个不同的随机数："+list);
    }

    /**
     * 生成n个不同的随机数，且随机数区间为[0,10)
     * @param n
     * @return
     */
    public ArrayList getDiffNO(int n){
        // 生成 [0-n) 个不重复的随机数
        // list 用来保存这些随机数
        ArrayList list = new ArrayList();
        Random rand = new Random();
        boolean[] bool = new boolean[n];
        int num = 0;
        for (int i = 0; i < n; i++) {
            do {
                // 如果产生的数相同继续循环
                num = rand.nextInt(n);
            } while (bool[num]);
            bool[num] = true;
            list.add(num);
        }
        return list;
    }


}
```

备注：下面是Java.util.Random()方法摘要：

1. protected int next(int bits)：生成下一个伪随机数。
2. boolean nextBoolean()：返回下一个伪随机数，它是取自此随机数生成器序列的均匀分布的boolean值。
3. void nextBytes(byte[] bytes)：生成随机字节并将其置于用户提供的 byte 数组中。
4. double nextDouble()：返回下一个伪随机数，它是取自此随机数生成器序列的、在0.0和1.0之间均匀分布的double值。
5. float nextFloat()：返回下一个伪随机数，它是取自此随机数生成器序列的、在0.0和1.0之间均匀分布float值。
6. double nextGaussian()：返回下一个伪随机数，它是取自此随机数生成器序列的、呈高斯（“正态”）分布的double值，其平均值是0.0标准差是1.0。
7. int nextInt()：返回下一个伪随机数，它是此随机数生成器的序列中均匀分布的 int 值。
8. int nextInt(int n)：返回一个伪随机数，它是取自此随机数生成器序列的、在（包括和指定值（不包括）之间均匀分布的int值。
9. long nextLong()：返回下一个伪随机数，它是取自此随机数生成器序列的均匀分布的 long 值。
10. void setSeed(long seed)：使用单个 long 种子设置此随机数生成器的种子。

下面给几个例子：

1. 生成[0,1.0)区间的小数：double d1 = r.nextDouble();
2. 生成[0,5.0)区间的小数：double d2 = r.nextDouble() * 5;
3. 生成[1,2.5)区间的小数：double d3 = r.nextDouble() * 1.5 + 1;
4. 生成-231到231-1之间的整数：int n = r.nextInt();
5. 生成[0,10)区间的整数：

```
int n2 = r.nextInt(10);//方法一
n2 = Math.abs(r.nextInt() % 10);//方法二
```

Math类用法

```
public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println(Math.E);//比任何其他值都更接近 e（即自然对数的底数）的 double 值。
        System.out.println(Math.PI);//比任何其他值都更接近 pi（即圆的周长与直径之比）的 double 值。
        /*
         * 1.abs绝对值函数
         * 对各种数据类型求绝对值
         */
        System.out.println(Math.abs(-10));//输出10

        /*
         * 2.三角函数与反三角函数
         * cos求余弦
         * sin求正弦
         * tan求正切
         * acos求反余弦
         * asin求反正弦
         * atan求反正切
         * atan2(y,x)求向量(x,y)与x轴夹角
         */
        System.out.println(Math.acos(-1.0));//输出圆周率3.14...
        System.out.println(Math.atan2(1.0, 1.0));//输出 π/4 的小数值

        /*
         * 3.开根号
         * cbrt(x)开立方
         * sqrt(x)开平方
         * hypot(x,y)求sqrt(x*x+y*y)在求两点间距离时有用sqrt((x1-x2)^2+(y1-y2)^2)
         */
        System.out.println(Math.sqrt(4.0));//输出2.0
        System.out.println(Math.cbrt(8.0));//输出2.0
        System.out.println(Math.hypot(3.0, 4.0));//输出5.0

        /*
         * 4.最值
         * max(a,b)求最大值
         * min(a,b)求最小值
         */
        System.out.println(Math.max(1, 2));//输出2
        System.out.println(Math.min(1.9, -0.2));//输出-0.2
        /*
         * 5.对数
         * log(a) a的自然对数(底数是e)
         * log10(a) a 的底数为10的对数
         * log1p(a) a+1的自然对数
         * 值得注意的是，前面其他函数都有重载，对数运算的函数只能传double型数据并返回double型数据
         */
        System.out.println(Math.log(Math.E));//输出1.0
        System.out.println(Math.log10(10));//输出1.0
        System.out.println(Math.log1p(Math.E-1.0));//输出1.0
        /*
         * 6.幂
         * exp(x) 返回e^x的值
         * expm1(x) 返回e^x - 1的值
         * pow(x,y) 返回x^y的值
         * 这里可用的数据类型也只有double型
```

```
         */
        System.out.println(Math.exp(2));//输出E^2的值
        System.out.println(Math.pow(2.0, 3.0));//输出8.0

        /*
         * 7.随机数
         * random()返回[0.0,1.0)之间的double值
         * 这个产生的随机数其实可以通过*x控制
         * 比如(int)(random*100)后可以得到[0,100)之间的整数
         */
        System.out.println((int)(Math.random()*100));//输出[0,100)间的随机数

        /*
         * 8.转换
         * toDegrees(a) 弧度换角度
         * toRadians(a) 角度换弧度
         */
        System.out.println(Math.toDegrees(Math.PI));//输出180.0
        System.out.println(Math.toRadians(180));//输出 π 的值
        /*
         * 9.其他
         */

        //copySign(x,y) 返回 用y的符号取代x的符号后新的x值
        System.out.println(Math.copySign(-1.0, 2.0));//输出1.0
        System.out.println(Math.copySign(2.0, -1.0));//输出-2.0

        //ceil(a) 返回大于a的第一个整数所对应的浮点数(值是整的，类型是浮点型)
        //可以通过强制转换将类型换成整型
        System.out.println(Math.ceil(1.3443));//输出2.0
        System.out.println((int)Math.ceil(1.3443));//输出2

        //floor(a) 返回小于a的第一个整数所对应的浮点数(值是整的，类型是浮点型)
        System.out.println(Math.floor(1.3443));//输出1.0

        //rint(a) 返回最接近a的整数的double值
        System.out.println(Math.rint(1.2));//输出1.0
        System.out.println(Math.rint(1.8));//输出2.0


        //nextAfter(a,b) 返回(a,b)或(b,a)间与a相邻的浮点数 b可以比a小
        System.out.println(Math.nextAfter(1.2, 2.7));//输出1.2000000000000002
        System.out.println(Math.nextAfter(1.2, -1));//输出1.1999999999999997
        //所以这里的b是控制条件

        //nextUp(a) 返回比a大一点点的浮点数
        System.out.println(Math.nextUp(1.2));//输出1.2000000000000002

        //nextDown(a) 返回比a小一点点的浮点数
        System.out.println(Math.nextDown(1.2));//输出1.1999999999999997
    }
}
```

Java Scanner类的方法及用法

Scanner类简介

Java 5添加了java.util.Scanner类，这是一个用于扫描输入文本的新的实用程序。它是以前的StringTokenizer和Matcher类之间的某种结合。由于任何数据都必须通过同一模式的捕获组检索或通过使用一个索引来检索文本的各个部分。于是可以结合使用正则表达式和从输入流中检索特定类型数据项的方法。这样，除了能使用正则表达式之外，Scanner类还可以任意地对字符串和基本类型(如int和double)的数据进行分析。借助于Scanner，可以针对任何要处理的文本内容编写自定义的语法分析器。

String	**next**() 查找并返回来自此扫描器的下一个完整标记。
String	**next**(Pattern pattern) 如果下一个标记与指定模式匹配，则返回下一个标记。
String	**next**(String pattern) 如果下一个标记与从指定字符串构造的模式匹配，则返回下一个标记。
BigDecimal	**nextBigDecimal**() 将输入信息的下一个标记扫描为一个 BigDecimal。
BigInteger	**nextBigInteger**() 将输入信息的下一个标记扫描为一个 BigInteger。
BigInteger	**nextBigInteger**(int radix) 将输入信息的下一个标记扫描为一个 BigInteger。
boolean	**nextBoolean**() 扫描解释为一个布尔值的输入标记并返回该值。
byte	**nextByte**() 将输入信息的下一个标记扫描为一个 byte。
byte	**nextByte**(int radix) 将输入信息的下一个标记扫描为一个 byte。
double	**nextDouble**() 将输入信息的下一个标记扫描为一个 double。
float	**nextFloat**() 将输入信息的下一个标记扫描为一个 float。
int	**nextInt**() 将输入信息的下一个标记扫描为一个 int。
int	**nextInt**(int radix) 将输入信息的下一个标记扫描为一个 int。
String	**nextLine**() 此扫描器执行当前行，并返回跳过的输入信息。
long	**nextLong**() 将输入信息的下一个标记扫描为一个 long。
long	**nextLong**(int radix) 将输入信息的下一个标记扫描为一个 long。
short	**nextShort**() 将输入信息的下一个标记扫描为一个 short。
short	**nextShort**(int radix) 将输入信息的下一个标记扫描为一个 short。

其实上图的意思就是，比如：nextInt():只读取int值，就是只能读取整数类型的数据，如果输入了非整型的数据（浮点型字符串等）就会报错。

nextFloat()、nextDouble()这些也是以此类推，只能读取符合该类型的数据。

next()和nextLine()的区别

next():只读取输入直到空格。它不能读两个由空格或符号隔开的单词。此外，next()在读取输入后将光标放在同一行中。(next()只读空格之前的数据,并且光标指向本行)

nextLine():读取输入，包括单词之间的空格和除回车以外的所有符号(即。它读到行尾)。读取输入后，nextLine()将光标定位在下一行。

代码演示：

```
public class Text {
    public static void main(String []args) {
        Scanner input = new Scanner(System.in);
        System.out.println("请输入一个字符串(中间能加空格或符号)");
        String a = input.nextLine();
```

```
        System.out.println("请输入一个字符串(中间不能加空格或符号)");
        String b = input.next();
        System.out.println("请输入一个整数");
        int c;
        c = input.nextInt();
        System.out.println("请输入一个double类型的小数");
        double d = input.nextDouble();
        System.out.println("请输入一个float类型的小数");
        float f = input.nextFloat();
        System.out.println("按顺序输出abcdf的值：");
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
        System.out.println(f);
    }
}
```

运行结果：

```
请输入一个字符串(中间能加空格或符号)
我爱祖国!
请输入一个字符串(中间不能加空格或符号)
ILoveChina
请输入一个整数
520
请输入一个double类型的小数
12.26e3
请输入一个float类型的小数
3.1415926
按顺序输出abcdf的值：
我爱祖国!
ILoveChina
520
12260.0
3.1415925
```

Scanner类的常用方法2

boolean	**hasNextBigDecimal**() 如果通过使用 nextBigDecimal() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 BigDecimal，则返回 true。
boolean	**hasNextBigInteger**() 如果通过使用 nextBigInteger() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 BigInteger 值，则返回 true。
boolean	**hasNextBigInteger**(int radix) 如果通过使用 nextBigInteger() 方法，此扫描器输入信息中的下一个标记可以解释为指定基数中的一个 BigInteger 值，则返回 true。
boolean	**hasNextBoolean**() 如果通过使用一个从字符串 "true|false" 创建的大小写敏感的模式，此扫描器输入信息中的下一个标记可以解释为一个布尔值，则返回 true。
boolean	**hasNextByte**() 如果通过使用 nextByte() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个字节值，则返回 true。
boolean	**hasNextByte**(int radix) 如果通过使用 nextByte() 方法，此扫描器输入信息中的下一个标记可以解释为指定基数中的一个字节值，则返回 true。
boolean	**hasNextDouble**() 如果通过使用 nextDouble() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 double 值，则返回 true。
boolean	**hasNextFloat**() 如果通过使用 nextFloat() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 float 值，则返回 true。
boolean	**hasNextInt**() 如果通过使用 nextInt() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 int 值，则返回 true。
boolean	**hasNextInt**(int radix) 如果通过使用 nextInt() 方法，此扫描器输入信息中的下一个标记可以解释为指定基数中的一个 int 值，则返回 true。
boolean	**hasNextLine**() 如果在此扫描器的输入中存在另一行，则返回 true。
boolean	**hasNextLong**() 如果通过使用 nextLong() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 long 值，则返回 true。
boolean	**hasNextLong**(int radix) 如果通过使用 nextLong() 方法，此扫描器输入信息中的下一个标记可以解释为指定基数中的一个 long 值，则返回 true。
boolean	**hasNextShort**() 如果通过使用 nextShort() 方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 short 值，则返回 true。
boolean	**hasNextShort**(int radix) 如果通过使用 nextShort() 方法，此扫描器输入信息中的下一个标记可以解释为指定基数中的一个 short 值，则返回 true。


```java
public class inputInformation {
    public static void main(String args[]) {
        Scanner scan = new Scanner(System.in);  //构造Scanner类的对象scan，接收从控制台输入的信息
        System.out.println("请输入你的姓名");
        String name = scan.nextLine();//接收一个字符串，可以加除Enter以外的所有符号，包括空格和Tab
        System.out.println("请输入你的ID");
        String ID ;
        while(scan.hasNextLine()) {// hasNextLine()方法判断当前是否有输入，当键盘有输入后执行循环
            if(scan.hasNextInt()) {// 判断输入的值是否为整数类型，当为整数类型时执行循环
                ID = scan.nextLine();
                System.out.println("你输入的姓名为："+name);
                System.out.println("你输入的ID为："+ID);
                break;
            }else {
                System.out.println("请输入数字哦！");
                ID = scan.nextLine();
                continue;
            }
        }
    }
}
```

运行结果如下：

```
请输入你的姓名
西地那非
请输入你的ID
q764323221
请输入数字哦！
764323221
```

Java ArrayList动态数组

1. 什么是ArrayList

ArrayList就是传说中的动态数组，用MSDN中的说法，就是Array的复杂版本，它提供了如下一些好处：

动态的增加和减少元素

实现了ICollection和IList接口

灵活的设置数组的大小

2、如何使用ArrayList

最简单的例子：

```
ArrayList List = new ArrayList();
for( int i=0;i <10;i++ ) //给数组增加10个Int元素
List.Add(i);
//..程序做一些处理
List.RemoveAt(5);//将第6个元素移除
for( int i=0;i <3;i++ ) //再增加3个元素
List.Add(i+20);
Int32[] values = (Int32[])List.ToArray(typeof(Int32));//返回ArrayList包含的数组
```

这是一个简单的例子，虽然没有包含ArrayList所有的方法，但是可以反映出ArrayList最常用的用法

3、ArrayList重要的方法和属性

1）构造器

ArrayList提供了三个构造器：

public ArrayList();

默认的构造器，将会以默认（16）的大小来初始化内部的数组

public ArrayList(ICollection);

用一个ICollection对象来构造，并将该集合的元素添加到ArrayList

public ArrayList(int);

用指定的大小来初始化内部的数组

2）IsSynchronized属性和ArrayList.Synchronized方法

IsSynchronized属性指示当前的ArrayList实例是否支持线程同步，而ArrayList.Synchronized静态方法则会返回一个ArrayList的线程同步的封装。

如果使用非线程同步的实例，那么在多线程访问的时候，需要自己手动调用lock来保持线程同步，例如：

```
ArrayList list = new ArrayList();
//...
lock( list.SyncRoot ) //当ArrayList为非线程包装的时候，SyncRoot属性其实就是它自己，但是为了满足ICollection的SyncRoot定义，这里还是使用SyncRoot来保持源代码的规范性
{
list.Add( “Add a Item” );
}
```

如果使用ArrayList.Synchronized方法返回的实例，那么就不用考虑线程同步的问题，这个实例本身就是线程安全的，实际上ArrayList内部实现了一个保证线程同步的内部类，ArrayList.Synchronized返回的就是这个类的实例，它里面的每个属性都是用了lock关键字来保证线程同步。

3）Count属性和Capacity属性

Count属性是目前ArrayList包含的元素的数量，这个属性是只读的。

Capacity属性是目前ArrayList能够包含的最大数量，可以手动的设置这个属性，但是当设置为小于Count值的时候会引发一个异常。

4）Add、AddRange、Remove、RemoveAt、RemoveRange、Insert、InsertRange

这几个方法比较类似

Add方法用于添加一个元素到当前列表的末尾

AddRange方法用于添加一批元素到当前列表的末尾

Remove方法用于删除一个元素，通过元素本身的引用来删除

RemoveAt方法用于删除一个元素，通过索引值来删除

RemoveRange用于删除一批元素，通过指定开始的索引和删除的数量来删除

Insert用于添加一个元素到指定位置，列表后面的元素依次往后移动

InsertRange用于从指定位置开始添加一批元素，列表后面的元素依次往后移动

另外，还有几个类似的方法：

Clear方法用于清除现有所有的元素

Contains方法用来查找某个对象在不在列表之中

其他的我就不一一累赘了，大家可以查看MSDN，上面讲的更仔细

5）TrimSize方法

这个方法用于将ArrayList固定到实际元素的大小，当动态数组元素确定不在添加的时候，可以调用这个方法来释放空余的内存。

6）ToArray方法

这个方法把ArrayList的元素Copy到一个新的数组中。

4、ArrayList与数组转换

例1：

```
ArrayList List = new ArrayList();
List.Add(1);
List.Add(2);
List.Add(3);

Int32[] values = (Int32[])List.ToArray(typeof(Int32));
```

例2：

```
ArrayList List = new ArrayList();
List.Add(1);
List.Add(2);
List.Add(3);

Int32[] values = new Int32[List.Count];
List.CopyTo(values);
```

上面介绍了两种从ArrayList转换到数组的方法

例3：

```
ArrayList List = new ArrayList();
List.Add( “string” );
List.Add( 1 );
//往数组中添加不同类型的元素

object[] values = List.ToArray(typeof(object)); //正确
string[] values = (string[])List.ToArray(typeof(string)); //错误
```

和数组不一样，因为可以转换为Object数组，所以往ArrayList里面添加不同类型的元素是不会出错的，但是当调用ArrayList方法的时候，要么传递所有元素都可以正确转型的类型或者Object类型，否则将会抛出无法转型的异常。

5、ArrayList最佳使用建议

这一节我们来讨论ArrayList与数组的差别，以及ArrayList的效率问题

1）ArrayList是Array的复杂版本

ArrayList内部封装了一个Object类型的数组，从一般的意义来说，它和数组没有本质的差别，甚至于ArrayList的许多方法，如Index、IndexOf、Contains、Sort等都是在内部数组的基础上直接调用Array的对应方法。

2）内部的Object类型的影响

对于一般的引用类型来说，这部分的影响不是很大，但是对于值类型来说，往ArrayList里面添加和修改元素，都会引起装箱和拆箱的操作，频繁的操作可能会影响一部分效率。

但是恰恰对于大多数人，多数的应用都是使用值类型的数组。

消除这个影响是没有办法的，除非你不用它，否则就要承担一部分的效率损失，不过这部分的损失不会很大。

3）数组扩容

这是对ArrayList效率影响比较大的一个因素。

每当执行Add、AddRange、Insert、InsertRange等添加元素的方法，都会检查内部数组的容量是否不够了，如果是，它就会以当前容量的两倍来重新构建一个数组，将旧元素Copy到新数组中，然后丢弃旧数组，在这个临界点的扩容操作，应该来说是比较影响效率的。

例1：比如，一个可能有200个元素的数据动态添加到一个以默认16个元素大小创建的ArrayList中，将会经过：

```
16*2*2*2*2 = 256
```

四次的扩容才会满足最终的要求，那么如果一开始就以：

```
ArrayList List = new ArrayList( 210 );
```

的方式创建ArrayList，不仅会减少4次数组创建和Copy的操作，还会减少内存使用。

例2：预计有30个元素而创建了一个ArrayList：

```
ArrayList List = new ArrayList(30);
```

在执行过程中，加入了31个元素，那么数组会扩充到60个元素的大小，而这时候不会有新的元素再增加进来，而且有没有调用TrimSize方法，那么就有1次扩容的操作，并且浪费了29个元素大小的空间。如果这时候，用：

```
ArrayList List = new ArrayList(40);
```

那么一切都解决了。

所以说，正确的预估可能的元素，并且在适当的时候调用TrimSize方法是提高ArrayList使用效率的重要途径。

4）频繁的调用IndexOf、Contains等方法（Sort、BinarySearch等方法经过优化，不在此列）引起的效率损失

首先，我们要明确一点，ArrayList是动态数组，它不包括通过Key或者Value快速访问的算法，所以实际上调用IndexOf、Contains等方法是执行的简单的循环来查找元素，所以频繁的调用此类方法并不比你自己写循环并且稍作优化来的快，如果有这方面的要求，建议使用Hashtable或SortedList等键值对的集合。

```
ArrayList al=new ArrayList();

al.Add("How");
al.Add("are");
al.Add("you!");

al.Add(100);
al.Add(200);
al.Add(300);

al.Add(1.2);
al.Add(22.8);
```

5）ToArray方法

这个方法把ArrayList的元素Copy到一个新的数组中。

使用ArrayList类

ArrayList类实现了List接口，由ArrayList类实现的List集合采用数组结构保存对象。数组结构的优点是便于对集合进行快速的随机访问，如果经常需要根据索引位置访问集合中的对象，使用由ArrayList类实现的List集合的效率较好。数组结构的缺点是向指定索引位置插入对象和删除指定索引位置对象的速度较慢，如果经常需要向List集合

的指定索引位置插入对象，或者是删除List集合的指定索引位置的对象，使用由ArrayList类实现的List集合的效率则较低，并且插入或删除对象的索引位置越小效率越低，原因是当向指定的索引位置插入对象时，会同时将指定索引位置及之后的所有对象相应的向后移动一位，如图1所示。当删除指定索引位置的对象时，会同时将指定索引位置之后的所有对象相应的向前移动一位，如图2所示。如果在指定的索引位置之后有大量的对象，将严重影响对集合的操作效率。

就是因为用ArrayList类实现的List集合在插入和删除对象时存在这样的缺点，在编写例程06时才没有利用ArrayList类实例化List集合，下面看一个模仿经常需要随机访问集合中对象的例子。
在编写该例子时，用到了Java.lang.Math类的random()方法，通过该方法可以得到一个小于10的double型随机数，将该随机数乘以5后再强制转换成整数，将得到一个0到4的整数，并随机访问由ArrayList类实现的List集合中该索引位置的对象，具体代码如下：
src\com\mwq\TestCollection.Java关键代码：

```
public static void main(String[] args) {
String a = "A", b = "B", c = "C", d = "D", e = "E";
List<String> list = new ArrayList<String>();
list.add(a);   // 索引位置为 0
list.add(b);   // 索引位置为 1
list.add(c);   // 索引位置为 2
list.add(d);   // 索引位置为 3
list.add(e);   // 索引位置为 4
System.out.println(list.get((int) (Math.random() * 5)));   // 模拟随机访问集合中的对象
}
```

我实际中的练习例子：

```
package code;
import java.util.ArrayList;
import java.util.Iterator;
public class SimpleTest {


public static void main(String []args){

  ArrayList list1 = new ArrayList();
  list1.add("one");
  list1.add("two");
  list1.add("three");
  list1.add("four");
  list1.add("five");
  list1.add(0,"zero");
  System.out.println("<--list1中共有>" + list1.size()+ "个元素");
  System.out.println("<--list1中的内容:" + list1 + "-->");

  ArrayList list2 = new ArrayList();
  list2.add("Begin");
  list2.addAll(list1);
  list2.add("End");
  System.out.println("<--list2中共有>" + list2.size()+ "个元素");
  System.out.println("<--list2中的内容:" + list2 + "-->");

  ArrayList list3 =  new ArrayList();
  list3.removeAll(list1);
  System.out.println("<--list3中是否存在one: "+ (list3.contains("one")? "是":"否")+ "-->");

```

```
30   list3.add(0,"same element");
31   list3.add(1,"same element");
32   System.out.println("<--list3中共有>" + list3.size()+ "个元素");
33   System.out.println("<--list3中的内容:" + list3 + "-->");
34   System.out.println("<--list3中第一次出现same element的索引是" + list3.indexOf("same element")
+ "-->");
35   System.out.println("<--list3中最后一次出现same element的索引是" + list3.lastIndexOf("same
element") + "-->");
36
37
38   System.out.println("<--使用Iterator接口访问list3->");
39   Iterator it = list3.iterator();
40   while(it.hasNext()){
41    String str = (String)it.next();
42    System.out.println("<--list3中的元素:" + list3 + "-->");
43   }
44
45   System.out.println("<--将list3中的same element修改为another element-->");
46   list3.set(0,"another element");
47   list3.set(1,"another element");
48      System.out.println("<--将list3转为数组-->");
49     // Object []  array =(Object[]) list3.toArray(new   Object[list3.size()] );
50      Object [] array = list3.toArray();
51      for(int i = 0; i < array.length ; i ++){
52       String str = (String)array[i];
53       System.out.println("array[" + i + "] = "+ str);
54      }
55
56      System.out.println("<---清空list3->");
57      list3.clear();
58      System.out.println("<--list3中是否为空: " + (list3.isEmpty()?"是":"否") + "-->");
59      System.out.println("<--list3中共有>" + list3.size()+ "个元素");
60
61   //System.out.println("hello world!");
62  }
63 }
```

java中的String.format()

常规类型的格式化

String类的format()方法用于创建格式化的字符串以及连接多个字符串对象。熟悉C语言的读者应该记得C语言的sprintf()方法，两者有类似之处。format()方法有两种重载形式。

```
format(String format, Object... args)
```

该方法使用指定的字符串格式和参数生成格式化的新字符串。 新字符串始终使用本地语言环境。例如当前日期信息在中国语言环境中的表现形式为“2007-10-27”，但是在其他国家有不同的表现形式。

语法：

```
String.format(format,args...)
```

`format`：字符串格式。

`args...`：字符串格式中由格式说明符引用的参数。如果还有格式说明符以外的参数，则忽略这些额外的参数。参数的数目是可变的，可以为0。

format(Locale locale, String format, Object... args)

该方法使用指定的语言环境、字符串格式和参数生成一个格式化的新字符串。新字符串始终使用指定的语言环境。

语法：

String.format(locale,format,args...)

locale：指定的语言环境。

format：字符串格式。

args...：字符串格式中由格式说明符引用的参数。如果还有格式说明符以外的参数，则忽略这些额外的参数。参数的数目是可变的，可以为0。

format()方法中的字符串格式参数有很多种转换符选项，例如：日期、整数、浮点数等。这些转换符的说明如表7.1所示。

表7.1　　转换符

转换符	说明	示例
%s	字符串类型	"mingrisoft"
%c	字符类型	'm'
%b	布尔类型	true
%d	整数类型（十进制）	99
%x	整数类型（十六进制）	FF
%o	整数类型（八进制）	77
%f	浮点类型	99.99
%a	十六进制浮点类型	FF.35AE
%e	指数类型	9.38e+5
%g	通用浮点类型（f和e类型中较短的）	
%h	散列码	
%%	百分比类型	%
%n	换行符	
%tx	日期与时间类型（x代表不同的日期与时间转换符	

下面的实例使用了表7.1中的各种转换符实现不同数据类型到字符串的转换，并将转换后的字符串通过System.out.printf()方法输出到控制台中。实现步骤如下。

（1）创建StrConversion类，将下面这段代码复制到类定义中。

```
public static void main(String[] args) {
  String str=null;
  str=String.format("Hi,%s", "飞龙");      // 格式化字符串
  System.out.println(str);                 // 输出字符串变量str的内容
  System.out.printf("字母a的大写是：%c %n", 'A');
  System.out.printf("3>7的结果是：%b %n", 3>7);
```

```
    System.out.printf("100的一半是：%d %n", 100/2);
    System.out.printf("100的16进制数是：%x %n", 100);
    System.out.printf("100的8进制数是：%o %n", 100);
    System.out.printf("50元的书打8.5折扣是：%f 元%n", 50*0.85);
    System.out.printf("上面价格的16进制数是：%a %n", 50*0.85);
    System.out.printf("上面价格的指数表示：%e %n", 50*0.85);
    System.out.printf("上面价格的指数和浮点数结果的长度较短的是：%g %n", 50*0.85);
    System.out.printf("上面的折扣是%d%% %n", 85);
    System.out.printf("字母A的散列码是：%h %n", 'A');
}
```

（2）运行StrConversion类，在控制台中输出的结果如下：

```
Hi,飞龙
字母a的大写是：A
3>7的结果是：false
100的一半是：50
100的16进制数是：64
100的8进制数是：144
50元的书打8.5折扣是：42.500000 元
上面价格的16进制数是：0x1.54p5
上面价格的指数表示：4.250000e+01
上面价格的指数和浮点数结果的长度较短的是：42.5000
上面的折扣是85%
字母A的散列码是：41
```

这些字符串格式参数不但可以灵活将其他数据类型转换成字符串，而且可以与各种标志组合在一起，生成各种格式的字符串，这些标志如表7.2所示。

表7.2　　搭配转换符的标志

标志	说明	示例	结果
+	为正数或者负数添加符号	("%+d",15)	+15
−	左对齐	("%-5d",15)	|15 |
0	数字前面补0	("%04d", 99)	0099
空格	在整数之前添加指定数量的空格	("% 4d", 99)	| 99|
,	以“,”对数字分组	("%,f", 9999.99)	9,999.990000
(	使用括号包含负数	("%(f", -99.99)	(99.990000)
#	如果是浮点数则包含小数点，如果是16进制或8进制则添加0x或0	("%#x", 99)("%#o", 99)	0x630143
<	格式化前一个转换符所描述的参数	("%f和%<3.2f", 99.45)	99.450000和99.45
$	被格式化的参数索引	("%1$d,%2$s", 99,"abc")	99,abc

下面的实例使用几种常用的转换符组合标志实现字符串的格式化，并通过System.out.printf()方法输出到控制台中。实现步骤如下。

（1）创建StrDateTime类，将下面这段代码复制到类定义中。

```
public static void main(String[] args) {
  String str=null;
  str=String.format("格式参数$的使用：%1$d,%2$s", 99,"abc");      // 格式化字符串
  System.out.println(str);                                          // 输出字符串变量
  System.out.printf("显示正负数的符号：%+d与%d%n", 99,-99);
  System.out.printf("最牛的编号是：%03d%n", 7);
  System.out.printf("Tab键的效果是：% 8d%n", 7);
  System.out.printf("整数分组的效果是：%,d%n", 9989997);
  System.out.printf("一本书的价格是：%2.2f元%n", 49.8);
}
```

(2) 运行StrFormat类，将在控制台输出字符串的格式化结果。

```
格式参数$的使用：99,abc
显示正负数的符号：+99与-99
最牛的编号是：007
Tab键的效果是：      7
整数分组的效果是：9,989,997
一本书的价格是：49.80元
```

7.4.2 日期和时间字符串格式化

在程序界面中经常需要显示时间和日期，但是其显示的 格式经常不尽人意，需要编写大量的代码经过各种算法才得到理想的日期与时间格式。字符串格式中还有%tx转换符没有详细介绍，它是专门用来格式化日期和时 间的。%tx转换符中的x代表另外的处理日期和时间格式的转换符，它们的组合能够将日期和时间格式化成多种格式。

1．常见日期时间格式化

格式化日期与时间的转换符定义了各种格式化日期字符串的方式，其中最常用的日期和时间的组合格式如表7.3所示。

表7.3　　常见日期和时间组合的格式

转换符	说明	示例
c	包括全部日期和时间信息	星期六 十月 27 14:21:20 CST 2007
F	“年-月-日”格式	2007-10-27
D	“月/日/年”格式	10/27/07
r	“HH:MM:SS PM”格式（12时制）	02:25:51 下午
T	“HH:MM:SS”格式（24时制）	14:28:16
R	“HH:MM”格式（24时制）	14:28

下面的实例使用表7.3中的转换符格式化当前日期和时间，并通过System.out.printf()方法输出到控制台中。实现步骤如下。

(1) 创建StrDateTime类，将下面这段代码复制到类定义中。

```
public static void main(String[] args) {
  Date date=new Date();                                    // 创建日期对象
  System.out.printf("全部日期和时间信息：%tc%n",date);          // 格式化输出日期或时间
  System.out.printf("年-月-日格式：%tF%n",date);
  System.out.printf("月/日/年格式：%tD%n",date);
  System.out.printf("HH:MM:SS PM格式（12时制）：%tr%n",date);
  System.out.printf("HH:MM:SS格式（24时制）：%tT%n",date);
  System.out.printf("HH:MM格式（24时制）：%tR",date);
}
```

（2）运行该实例，将在控制台输出本地时间格式的当前日期和时间。运行结果如下：

```
全部日期和时间信息：星期日十月28 13:53:24 CST 2007
年-月-日格式：2007-10-28
月/日/年格式：10/28/07
HH:MM:SS PM格式（12时制）：01:53:24 下午
HH:MM:SS格式（24时制）：13:53:24
HH:MM格式（24时制）：13:53
```

2．格式化日期字符串

定义日期格式的转换符可以使日期通过指定的转换符生成新字符串。这些日期转换符如表7.4所示。

表7.4　　日期格式化转换符

转换符	说明	示例
b或者h	月份简称	中：十月英：Oct
B	月份全称	中：十月英：October
a	星期的简称	中：星期六英：Sat
A	星期的全称	中：星期六英：Saturday
C	年的前两位数字（不足两位前面补0）	20
y	年的后两位数字（不足两位前面补0）	07
Y	4位数字的年份（不足4位前面补0）	2007
j	一年中的天数（即年的第几天）	300
m	两位数字的月份（不足两位前面补0）	10
d	两位数字的日（不足两位前面补0）	27
e	月份的日（前面不补0）	5

下面的实例将使用各种转换符格式化当前系统的日期，并通过System.out.printf()方法输出到控制台中。实现步骤如下。

（1）创建StrDate类，将下面这段代码复制到类定义中。

```
public static void main(String[] args) {
  Date date=new Date();                                    // 创建日期对象
  String str=String.format(Locale.US,"英文月份简称：%tb",date);   // 格式化日期字符串
  System.out.println(str);                                 // 输出字符串内容
```

```
        System.out.printf("本地月份简称：%tb%n",date);
        str=String.format(Locale.US,"英文月份全称：%tB",date);
        System.out.println(str);
        System.out.printf("本地月份全称：%tB%n",date);
        str=String.format(Locale.US,"英文星期的简称：%ta",date);
        System.out.println(str);
        System.out.printf("本地星期的简称：%tA%n",date);
        System.out.printf("年的前两位数字（不足两位前面补0）：%tC%n",date);
        System.out.printf("年的后两位数字（不足两位前面补0）：%ty%n",date);
        System.out.printf("一年中的天数（即年的第几天）：%tj%n",date);
        System.out.printf("两位数字的月份（不足两位前面补0）：%tm%n",date);
        System.out.printf("两位数字的日（不足两位前面补0）：%td%n",date);
        System.out.printf("月份的日（前面不补0）：%te",date);
    }
```

(2) 运行本实例，将在控制台输出各种日期格式化的字符串。运行结果如下：

```
英文月份简称：Oct
本地月份简称：十月
英文月份全称：October
本地月份全称：十月
英文星期的简称：Sun
本地星期的简称：星期日
年的前两位数字（不足两位前面补0）：20
年的后两位数字（不足两位前面补0）：07
一年中的天数（即年的第几天）：301
两位数字的月份（不足两位前面补0）：10
两位数字的日（不足两位前面补0）：28
月份的日（前面不补0）：28
```

3．格式化时间字符串

和日期格式转换符相比，时间格式的转换符要更多、更精确。它可以将时间格式化成时、分、秒甚至时毫秒等单位。格式化时间字符串的转换符如表7.5所示。

表7.5　　时间格式化转换符

转换符	说明	示例
H	2位数字24时制的小时（不足2位前面补0）	15
I	2位数字12时制的小时（不足2位前面补0）	03
k	2位数字24时制的小时（前面不补0）	15
l	2位数字12时制的小时（前面不补0）	3
M	2位数字的分钟（不足2位前面补0）	03
S	2位数字的秒（不足2位前面补0）	09
L	3位数字的毫秒（不足3位前面补0）	015
N	9位数字的毫秒数（不足9位前面补0）	562000000
p	小写字母的上午或下午标记	中：下午英：pm
z	相对于GMT的RFC822时区的偏移量	+0800

转换符	说明	示例
Z	时区缩写字符串	CST

续表

转换符	说明	示例
s	1970-1-1 00:00:00 到现在所经过的秒数	1193468128
Q	1970-1-1 00:00:00 到现在所经过的毫秒数	1193468128984

下面通过实例使用各种转换符格式化当前系统的时间，并通过System.out.printf()方法输出到控制台中。实现步骤如下。

(1) 创建StrTime类，将下面这段代码复制到类定义中。

```
public static void main(String[] args) {
  Date date=new Date();                  // 创建日期对象
  System.out.printf("2位数字24时制的小时（不足2位前面补0）:%tH%n",date);
  System.out.printf("2位数字12时制的小时（不足2位前面补0）:%tI%n",date);
  System.out.printf("2位数字24时制的小时（前面不补0）:%tk%n",date);
  System.out.printf("2位数字12时制的小时（前面不补0）:%tl%n",date);
  System.out.printf("2位数字的分钟（不足2位前面补0）:%tM%n",date);
  System.out.printf("2位数字的秒（不足2位前面补0）:%tS%n",date);
  System.out.printf("3位数字的毫秒（不足3位前面补0）:%tL%n",date);
  System.out.printf("9位数字的毫秒数（不足9位前面补0）:%tN%n",date);
  String str=String.format(Locale.US,"小写字母的上午或下午标记(英)：%tp",date);
  System.out.println(str);               // 输出字符串变量str的内容
  System.out.printf ("小写字母的上午或下午标记（中）：%tp%n",date);
  System.out.printf("相对于GMT的RFC822时区的偏移量:%tz%n",date);
  System.out.printf("时区缩写字符串:%tZ%n",date);
  System.out.printf("1970-1-1 00:00:00 到现在所经过的秒数：%ts%n",date);
  System.out.printf("1970-1-1 00:00:00 到现在所经过的毫秒数：%tQ%n",date);
}
```

(2) 运行实例，在控制台将输出以下信息：

```
2位数字24时制的小时（不足2位前面补0）:15
2位数字12时制的小时（不足2位前面补0）:03
2位数字24时制的小时（前面不补0）:15
2位数字12时制的小时（前面不补0）:3
2位数字的分钟（不足2位前面补0）:24
2位数字的秒（不足2位前面补0）:56
3位数字的毫秒（不足3位前面补0）:828
9位数字的毫秒数（不足9位前面补0）:828000000
小写字母的上午或下午标记(英)：pm
小写字母的上午或下午标记（中）：下午
相对于GMT的RFC822时区的偏移量:+0800
时区缩写字符串:CST
1970-1-1 00:00:00到现在所经过的秒数：1193556296
1970-1-1 00:00:00到现在所经过的毫秒数：1193556296828
```

Java HashSet

对于 HashSet而言，它是基于 HashMap 实现的，HashSet 底层采用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，查看 HashSet的源代码，可以看到如下代码：

Java代码

```
1. public class HashSet<E>
2.  extends AbstractSet<E>
3.  implements Set<E>, Cloneable, java.io.Serializable
4. {
5.  // 使用 HashMap 的 key 保存 HashSet 中所有元素
6.  private transient HashMap<E,Object> map;
7.  // 定义一个虚拟的 Object 对象作为 HashMap 的 value
8.  **private static final Object PRESENT = new Object();
9.  ...
10.  // 初始化 HashSet，底层会初始化一个 HashMap
11.  public HashSet()
12.  {
13.    map = new HashMap<E,Object>();
14.  }
15.  // 以指定的 initialCapacity、loadFactor 创建 HashSet
16.  // 其实就是以相应的参数创建 HashMap
17.  public HashSet(int initialCapacity, float loadFactor)
18.  {
19.    map = new HashMap<E,Object>(initialCapacity, loadFactor);
20.  }
21.  public HashSet(int initialCapacity)
22.  {
23.    map = new HashMap<E,Object>(initialCapacity);
24.  }
25.  HashSet(int initialCapacity, float loadFactor, boolean dummy)
26.  {
27.    map = new LinkedHashMap<E,Object>(initialCapacity**, loadFactor);
28.
29.  }
30.  // 调用 map 的 keySet 来返回所有的 key
31.  public Iterator<E> iterator()
32.  {
33.    return map.keySet().iterator();
34.  }
35.  // 调用 HashMap 的 size() 方法返回 Entry 的数量，就得到该 Set 里元素的个数
36.  public int size()
37.  {
38.    return map.size();
39.  }
40.  // 调用 HashMap 的 isEmpty() 判断该 HashSet 是否为空，
41.  // 当 HashMap 为空时，对应的 HashSet 也为空
42.  public boolean isEmpty()
43.  {
44.    return map.isEmpty();
45.  }
46.  // 调用 HashMap 的 containsKey 判断是否包含指定 key
47.  //HashSet 的所有元素就是通过 HashMap 的 key 来保存的
48.  public boolean contains(Object o)
49.  {
50.    return map.containsKey(o);
51.  }
52.  // 将指定元素放入 HashSet 中，也就是将该元素作为 key 放入 HashMap
53.  public boolean add(E e)
54.  {
55.    return map.put(e, PRESENT) == null;
56.  }
```

```
57.  // 调用 HashMap 的 remove 方法删除指定 Entry，也就删除了 HashSet 中对应的元素
58.  public boolean remove(Object o)
59.  {
60.    return map.remove(o)==PRESENT;
61.  }
62.  // 调用 Map 的 clear 方法清空所有 Entry，也就清空了 HashSet 中所有元素
63.  public void clear()
64.  {
65.    map.clear();
66.  }
67.  ...
```

由上面源程序可以看出，HashSet 的实现其实非常简单，它只是封装了一个 HashMap 对象来存储所有的集合元素，所有放入 HashSet 中的集合元素实际上由 HashMap的 key 来保存，而 HashMap 的 value 则存储了一个 PRESENT，它是一个静态的 Object 对象。

HashSet 的绝大部分方法都是通过调用 HashMap 的方法来实现的，因此HashSet和HashMap 两个集合在实现本质上是相同的。
掌握上面理论知识之后，接下来看一个示例程序，测试一下自己是否真正掌握了 HashMap 和 HashSet 集合的功能。

Java代码

```
1.  class Name
2. {
3.   private String first;
4.   private String last;
5.
6.   public Name(String first, String last)
7.   {
8.     this.first = first;
9.     this.last = last;
10.   }
11.
12.   public boolean equals(Object o)
13.   {
14.     if (this == o)
15.     {
16.       return true;
17.     }
18.
19.   if (o.getClass() == Name.class)
20.     {
21.       Name n = (Name)o;
22.       return n.first.equals(first)
23.         && n.last.equals(last);
24.     }
25.     return false;
26.   }
27. }
28.
29. public class HashSetTest
30. {
31.   public static void main(String[] args)
32.   {
33.     Set<Name> s = new HashSet<Name>();
34.     s.add(new Name("abc", "123"));
35.     System.out.println(
36.     s.contains(new Name("abc", "123")));
```

```
  }
}
```

上面程序中向 HashSet 里添加了一个 new Name("abc", "123")对象之后，立即通过程序判断该HashSet 是否包含一个 new Name("abc", "123")对象。粗看上去，很容易以为该程序会输出 true。

实际运行上面程序将看到程序输出 false，这是因为 HashSet判断两个对象相等的标准除了要求通过 equals()方法比较返回 true之外，还要求两个对象的hashCode() 返回值相等。而上面程序没有重写 Name类的hashCode() 方法，两个Name对象的 hashCode()返回值并不相同，因此HashSet会把它们当成 2 个对象处理，因此程序返回false。

由此可见，当我们试图把某个类的对象当成 HashMap的 key，或试图将这个类的对象放入 HashSet中保存时，重写该类的 equals(Object obj) 方法和 hashCode() 方法很重要，而且这两个方法的返回值必须保持一致：当该类的两个的hashCode()返回值相同时，它们通过 equals() 方法比较也应该返回true。通常来说，所有参与计算 hashCode() 返回值的关键属性，都应该用于作为 equals() 比较的标准。
如下程序就正确重写了 Name 类的 hashCode()和 equals() 方法，程序如下：

Java代码

```
class Name
{
  private String first;
  private String last;
  public Name(String first, String last)
  {
    this.first = first;
    this.last = last;
  }
  // 根据 first 判断两个 Name 是否相等
  public boolean equals(Object o)
  {
    if (this == o)
    {
      return true;
    }
    if (o.getClass() == Name.class)
    {
      Name n = (Name)o;
      return n.first.equals(first);
    }
    return false;
  }

  // 根据 first 计算 Name 对象的 hashCode() 返回值
  public int hashCode()
  {
    return first.hashCode();
  }

  public String toString()
  {
    return "Name[first=" + first + ", last=" + last + "]";
  }
}

public class HashSetTest2
{
  public static void main(String[] args)
  {
```

```
41.     HashSet<Name> set = **new HashSet<Name>();
42.     set.add(new Name("abc" , "123"));
43.     set.add(new Name("abc" , "456"));
44.     System.out.println(set);
45.   }
46. }
```

上面程序中提供了一个 Name类，该 Name类重写了 equals() 和 toString() 两个方法，这两个方法都是根据 Name类的 first 实例变量来判断的，当两个 Name 对象的 first 实例变量相等时，这两个 Name 对象的hashCode()返回值也相同，通过equals()比较也会返回 true。

程序主方法先将第一个 Name对象添加到 HashSet中，该 Name 对象的 first 实例变量值为"abc"，接着程序再次试图将一个 first为"abc"的 Name对象添加到 HashSet 中，很明显，此时没法将新的 Name 对象添加到该 HashSet中，因为此处试图添加的 Name对象的 first也是" abc"，HashSet 会判断此处新增的Name 对象与原有的Name对象相同，因此无法添加进入，程序在①号代码处输出 set 集合时将看到该集合里只包含一个 Name对象，就是第一个、last为"123"的 Name 对象。

HashMap

类似于字典，采用哈希算法与两两组合的方式存储数据。采用类似于图论的东西去理解。

由于这一块与数据结构相对来说关系更大，故在这里只简单地列出方法。

方法概览：

1. put(Object key,Object value)和putAll(Collection c)添加映射
2. get(Object key)根据键来获取对应的值
3. containsKey(Object key)和containsValue(Object value)
4. remove(Object key)
5. values()
6. isEmpty()
7. entrySet()
8. keySet()

Map与其他两个集合的区别就是，在Map中数据是以键-值对的形式存储的。一个键值对交一个映射；访问元素的时候只能通过键来访问指定的元素

全局Map

```
HashMap<String,String> hm1 = new HashMap<String,String>();
HashMap<String,String> hm2 = new HashMap<String,String>();
```

- put(Object key,Object value)在此映射中关联指定的Key-value
- putAll(Collection c)在此映射中将指定的映射关系添加到被操作的映射中

```
String [] key = {"name","age","tender"};
        String [] value = {"zhangsan","16","men"};
        hm2.put("id","012");
        hm2.put("describe", "zhangdelaohaokanle");
        for(int i = 0;i<3;i++){
            hm1.put(key[i], value[i]);
        }
        hm1.putAll(hm2);
```

- get(Object key)根据key获取指定的value

```
System.out.println(hm1.get("name"));
```

- containsKey(Object key)检测该映射中是否存在指定key的映射，有则返回true；没有则返回false
- containsValue(Object value)检测该映射中是否存在指定value的映射，有则返回true；没有则返回false

```
System.out.println(hm1.containsKey("id")+"    "+hm1.containsValue("013"));
```

- remove(Object key)根据key的值删除指定的映射关系

```
System.out.println(hm1.remove("describe"));
```

- values()返回值的集合，

```
        Collection<String> li = hm1.values();
        for (String string : li) {
            System.out.print(string+"     ");
        }
```

- isEmpty()测试映射是否为空

```
System.out.println("\n"+hm1.isEmpty());
```

遍历map得到key和value的两种方法

1. entrySet()将此映射所包含的映射关系返回到Set中，通过Iterator迭代器迭代输出，或者用foreach输出

- 这里我们推荐forearch，为什么呢？因为foreach代码量少，还有一点就是foreach效率高，不仅是开发效率，还有运行效率

```
for (Map.Entry<String, String> me:hm1.entrySet()) {
            System.out.println(me.getKey()+":"+me.getValue());
        }
        /*Set<Entry<String, String>> se = hm1.entrySet();
        Iterator<Entry<String, String>> it = se.iterator();
        while(it.hasNext()){
            Map.Entry me =  it.next();
            System.out.println(me.getKey()+":"+me.getValue());
        }*/
```

2、keySet()将映射中所包含的键返回到Set中，通过Set的Iterator迭代器迭代输出，或者用foreach输出

```
for (String key1:hm1.keySet()) {
            System.out.println(key1+":"+hm1.get(key1));
        }
        /*Set<String> set = hm1.keySet();
        Iterator<String> it1 = set.iterator();
        while(it1.hasNext()){
            System.out.println(hm1.get(it1.next()));
        }*/
```

object类

由于文章过长并且非常不便于归档，在这里直接贴出超链接。

[Java：Object类详解]

StringBuffer类

在这里就不做过多的原理性描述。

StringBuffer 方法

以下是 StringBuffer 类支持的主要方法：

序号	方法描述
1	`public StringBuffer append(String s)` 将指定的字符串追加到此字符序列。
2	`public StringBuffer reverse()` 将此字符序列用其反转形式取代。
3	`public delete(int start, int end)` 移除此序列的子字符串中的字符。
4	`public insert(int offset, int i)` 将 `int` 参数的字符串表示形式插入此序列中。
5	`replace(int start, int end, String str)` 使用给定 `String` 中的字符替换此序列的子字符串中的字符。

下面的列表里的方法和 String 类的方法类似：

序号	方法描述
1	`int capacity()` 返回当前容量。
2	`char charAt(int index)` 返回此序列中指定索引处的 `char` 值。
3	`void ensureCapacity(int minimumCapacity)`确保容量至少等于指定的最小值。

序号	方法描述
4	`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`将字符从此序列复制到目标字符数组 `dst`。
5	`int indexOf(String str)`返回第一次出现的指定子字符串在该字符串中的索引。
6	`int indexOf(String str, int fromIndex)`从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。
7	`int lastIndexOf(String str)` 返回最右边出现的指定子字符串在此字符串中的索引。
8	`int lastIndexOf(String str, int fromIndex)` 返回 String 对象中子字符串最后出现的位置。
9	`int length()` 返回长度（字符数）。
10	`void setCharAt(int index, char ch)`将给定索引处的字符设置为 `ch`。
11	`void setLength(int newLength)`设置字符序列的长度。
12	`CharSequence subSequence(int start, int end)` 返回一个新的字符序列，该字符序列是此序列的子序列。
13	`String substring(int start)`返回一个新的 `String`，它包含此字符序列当前所包含的字符子序列。
14	`String substring(int start, int end)`返回一个新的 `String`，它包含此序列当前所包含的字符子序列。
15	`String toString()` 返回此序列中数据的字符串表示形式。

Swing类

这玩意我先空着，因为我还没有找到成体系的、免费的API文档。

JTable类

没错这玩意我也空着，原因同上。

小贴士

赋值语句中一般是以这样的形式进行赋值：

```
int a = 1,b = 2;
```

但一般不这么写。

括号内存在优先级运算，故在输出字符串时不可简单的认为是元组转化为字符串的形式输出。其实这一点在python中也有所耳闻。

大括号之后一般没有分号。

多个变量引用时要注意引用是否得当。

final和const是一个意思.

具体来说可看下行代码：

```
int a = 1, b = 2;
System.out.println(a+b/3);//此时b/3输出数据类型依旧为int，故为0。
int a = 1, b = 2;
System.out.println(a+b/3.0);//此时由于涉及到整形与浮点型混合运算，所以整形自动转换成浮点型进行运算。
```

注意一点：

> (1)**float** 型 内存分配4个字节,占32位,范围从10^{-38}到10^{38} 和 -10^{38}到-10^{-38}
> 例float x=123.456f,y=2e20f; 注意float型定义的数据末尾必须 有"**f**"或"F",为了和double区别
>
> (2)**double** 型 内存分配8个字节,范围从10^{-308}到10^{308} 和 -10^{-308}到-10^{-308}
> 例double x=1234567.98,y=8980.09d; 末尾可以有"d"也可以不写

注意浮点数计算是存在误差的。

//为注释。/**/为多行注释。

强制类型转换（这玩意在java里要好用多了）

```
int a = 1, b = 2;
System.out.println((double)(a+b/2));//(type)(underchanged num)
```

值得一提的是强制类型转换不会转变之前值得类型，故需要容器来接应这个数值。

布尔值的命名为boolean而不是bool。

不同于python的负数索引，java仅支持对应的index索引。比如下列代码会报错：

```
int[] array_1 = new int[10];
System.out.println(array_1[-1]);
//error:ArrayIndexOutOfBoundsException
```

数组成员length。

```
int[] array_1 = new int[10];
System.out.println(array_1.length);//10
```

java数组的命名有点像python中对list的定义。此时列表名是一个类似于标签一样的存在。

int数组的元素初始值均为0.