字符串：`String`、`StringBuffer`和`char []`

千言万语归为一句话：Java并没有C语言里面的字符串数组的控制方法！

`String`

`String`是只读的

有一个很重要的点，`String`是只读的！

如果我们声明了一个`String str;`，我们可以之后再赋值，也可以初始化它，但是我们的任何一种函数操作都不能改变其指向的内存里的内容！

比如

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner cin = new Scanner(System.in);
        String str=new String("abcd");
        String org=str; // 别忘了这里本质是指针层面的等于
        str=str.toUpperCase();
        System.out.println(org); // ABCD
        System.out.println(str); // abcd
    }
}
```

实际上是会在内存上产生一个新的串`"ABCD"`，然后`str`从原来的指向这个新的，而原来的部分并没有变

声明方法和字面值问题

`String`的声明方法有很多，下面介绍一些

- 使用`""`的构造：`String str=new String("abcd");`
- 复制构造：`String S=new String(str);`

需要注意，`String`也可以指向一个“字面值”，看下例：

```
String str1="hello";
String str2="hello";
System.out.println(str1==str2);
System.out.println(str1=="hello");
String[] strs=new String[1];
strs[0]="hello";
System.out.println(str1==strs[0]);
String str3=new String("hello");
System.out.println(str3==str1);
```

输出的是三个`true`最后一个`false`，也就是`String str="string"`并不是“新建了一个对象”，而是指向了同一个空间内容，里面存储的是`"hello"`，不过`String str=new String("str")`就确实是产生了一个新的对象

实际上还可以使用`StringBuffer`或者字符数组等初始化，但是这俩本身就是一个大坑，见后

`String.valueOf()`

Java提供了一套非常好用的将类直接转换成字符串的接口函数，这些接口函数都被重载为`String.valueOf()`方法

输入对象和字符数组外的数组

对于对象，`String.valueOf(Object)`会调用对象的`.toString()`函数。如下：

```
// in String.java
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

先说下`obj==null`。我们已经了解过Java中对象即指针，输出一个没有被构造过的指针指向的内容是危险的，所以需要先判断这个指针是不是`null`，所以一个没有被构造的对象是可以转换成`String`的，其得到的内容是`null`。而`obj`不是`null`就会调用`obj`的`.toString()`函数

特别的，如果输入的是`int []`，`String.valueOf()`输出的会是其地址（因为数组的`.toString()`返回的就是地址），如果想要输出数组内容需要使用`System.out.println(Arrays.toString(xs));`

输入字符数组

对于字符数组这个对象，`String.valueOf()`重载了很多套函数，本质都是新建了一个`String`

```
// in String.java
public static String valueOf(char data[]) {
    return new String(data);
}
```

注意，字符数组的`.toString()`的表现和其他的不一样，实际上只要是数组类的对象使用`toString()`输出的都是地址

输入基本类型

但是我们知道Java并不是彻底的“一切都是对象”，实际上还是有基本类型的存在的，那么Java怎么对待基本类型的转换呢？见下

```
// in String.java
public static String valueOf(double d) { // 实际上对于char会有所不同，这里不展开
    return Double.toString(d);
}
```

这是`String.valueOf(double)`的源码，可以看出对于`double`这个基本类型，Java调用的是`Double.toString(double)`，也就是`double`的包装类`Double`的`toString()`函数。对于别的基本类型也一样，比如`int`就会调用`valueOf(int i)`，然后调用`Integer.toString(i);`

总结

总结下就是，`String.valueOf()`函数：
对于字符数组输入，直接使用字符构造一个`String`并返回；对于其他引用类型的输入，会调用对象的`toString()`函数；对于基本类型，会调用类型的包装类的`toString()`函数。特别的，数组`toString()`返回的是地址信息，所以想要输出内容需要使用`Arrays.toString()`

一些方法

- `.charAt(int idx)`：

  返回`idx`位置的字符（下标从`0`开始）

- `.split(String regex)`：

  将输入的正则表达式作为分隔符分割字符串，返回分割后的字符数组

- `String.format(String source,Obejcts ...)`：

  `str=String.format(source,1,2);`相当于C语言里`sprintf(str,source,1,2);`

`.toString()`

实际上`.toString()`并不应该是`String`里的内容，我们会在其他部分着重解释，但是这里先介绍下

故名思义，`.toString()`可以把某一个东西变成`String`，可以有一个基本概念：对于基本类型，有一个包装类，包装类有一个对应的`toString`函数，比如如果有`double b=3.14;`，那就可以使用`Double.toString(b);`将其转化成一个字符串；对于其他类型，可以直接使用`.toString()`

`.toString()`是`Object`类的一个函数，由于Java中每个类都会隐式继承自`Object`类，所以每个类都有这个函数

自定义类没有写`.toString()`时，因为所有类全部都会隐式继承自`Object`类，所以这里调用的实际上是基类的`toString()`函数。`Object`的`toString()`函数，输出的是类名+地址

+

我们都知道Java并没有运算符重载，但是实际上可以写出下面的语句

```
String str=new String("abcd");
int x=10;
System.out.println(str+x);
String str2=str+"e"+str;
```

这里的`+`实际上是一个“语法糖”，可以直接“连接字符串”。有人认为这是Java中唯一的运算符重载

实际上出于效率考虑，Java的编译器会使用`StringBuilder`自动优化。比如上面的`String str2=str+"e"+str;`，Java编译器会自动使用`StringBuilder`（和后面会介绍的`StringBuffer`除了线程上没有区别）优化，使拼接过程中只产生一个`StringBuilder`中间变量，而不是递归调用`+`（比如C++的`<<`的过程就是递归<比如`cout<<a<<b;`，本质是调用`cout<<a`返回一个`cout`再调用`cout<<b`>，虽然在C++中这是无所谓的）

需要注意的是，这个语法糖在面对非`String`的`obj`和`String`相加时的处理方式是：调用`String.valueOf(obj)`再将得到的字符串拼接

实际上，对于任何需要输入字符串但是输入的不是字符串的情况，Java的编译器都会自动调用`String.valueOf()`

总结

总之，虽然String的效率极其感人，但是如果只是出于应试考虑的话还是可以考虑使用的

以及`String.valueOf`和`toString`这两个还是要记住的

`StringBuffer`

和`StringBuilder`相比，`StringBuffer`是线程安全的，`StirngBuilder`是非线程安全的

`StringBuffer`也不是C语言风格字符串

`StringBuffer`听起来像是一个C语言风格字符串，但是实际上完全不是

```
StringBuffer ss=new StringBuffer(10);
ss.setCharAt(1,'a');
```

这个语句会发生错误!

我们看起来是生成了一个字符串，它的长度是10，但是实际上它存储到串的长度是0

StringBuffer是一个规定了存储长度的缓存区

即`StringBuffer`使用`int n`的初始化，产生的只是一个**容量**为`n`的Buffer，和内容完全没关！

可以观察下面语句的输出结果来理解

```
import java.math.BigInteger;
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner cin = new Scanner(System.in);
        StringBuffer ss=new StringBuffer(10);
        for(int i=0;i<=5;i++){
            ss.append((char)('a'+i));
            // 这里如果写成ss.append('a'+i);会发生其他奇怪的事情，可以试试
            System.out.println("ss.length() : "+ss.length());
        }
        System.out.println("ss : "+ss.toString());
        ss.setCharAt(2,'\0');
        System.out.println("ss.length() : "+ss.length());
        System.out.println("ss : "+ss);
    }
}
```

输出

```
ss.length() : 1
ss.length() : 2
ss.length() : 3
ss.length() : 4
ss.length() : 5
ss.length() : 6
ss : abcdef
ss.length() : 6
ss : abdef
```

最后一行有`'\0'`

但是`StringBuffer`可以用来处理字符串

虽然`StringBuffer`不像C语言的字符数组，但是还是可以用来处理字符串的：

```
Scanner cin=new Scanner("This is A String to input");
StringBuffer sb=new StringBuffer(cin.nextLine());
for(int i=0;i<sb.length();++i){
    if(Character.isUpperCase(sb.charAt(i))){
        sb.setCharAt(i,Character.toLowerCase(sb.charAt(i)));
    }else if(Character.isLowerCase(sb.charAt(i))){
        sb.setCharAt(i,Character.toUpperCase(sb.charAt(i)));
    }
}
System.out.println(sb); // tHIS IS a sTRING TO INPUT
```

使用`charAt(int index)`和`setCharAt(int index,char newChar)`就行

`.append()`

`.append()`可以往StringBuffer里原来存储的字符串的后面加内容，但是要注意的是，这里加的内容是根据括号里的参数决定的，比如：

```
StringBuffer org=new StringBuffer("oeg : ");
System.out.println(org.append(100));
System.out.println(org.append('a'));
System.out.println(org.append(99.00));
```

这个语句跑出来的结果是

```
org : 100
org : 100a
org : 100a99.0
```

也就是实际上如果`.append()`里是一个`int`，就会把`int`转化的字符串加到后面。其他类型类似。上面的语句和下面语句输出一致

```
StringBuffer org=new StringBuffer("org : ");
System.out.println(org.append(Integer.toString(100)));
System.out.println(org.append(Character.toString('a')));
System.out.println(org.append(Double.toString(99.00)));
```

看过上面的`toString`的介绍后，也可以猜想下面和上面的等效

```
StringBuffer org=new StringBuffer("org : ");
System.out.println(org.append(String.valueOf(100)));
System.out.println(org.append(String.valueOf('a')));
System.out.println(org.append(String.valueOf(99.00)));
```

（实际确实等效）

字符数组

Java的字符数组总能产生奇怪的表现

如果我们只看下面的例子的话

```
char [] str=new char[10];
for(int i=0;i<=4;i++){
    str[i]=(char)('a'+i);
}
System.out.println(str);
```

我们或许以为找到了一个很好的C风格字符串的替代品，但是再加上下面的语句

```
str[6]='g';
System.out.println(str);
```

看了输出结果我想大多数人就会打消用它当作C风格字符串的念头了，原因是：

Java的字符数组并不支持到'\0'自动停止。关于字符数组的输出在后面会有进一步的介绍

另外注意区分

```
char [] str=new char[10];...;
System.out.println(str);
System.out.println(str.toString());
System.out.println(Arrays.toString(str));
```

这三个的区别

可以通过下面的例子理解

```
char [] str=new char[10];
for(int i=0;i<=4;i++){
    str[i]=(char)('a'+i);
}
str[6]='g';
System.out.println(str);
System.out.println(str.toString());
System.out.println(Arrays.toString(str));
```

一种输出：

```
abcdeg
[C@9f70c54
[a, b, c, d, e,  , g,  ,  , ]
```

Java的输出

`System.out.*`

我们一般使用的都是`System.out.`，这里面我们常用的是：

- `.println()`：会输出参数并且自带换行
- `.printf()`：C++风格的输出，后面单独介绍
- `.print()`：和`.println()`就差一个自动换行

这些函数都提供了接受`Object obj`的重载，具体怎么调用的见后面介绍

马上会具体介绍这几个

引子：未初始化类使用`println`输出

在Java中如果一个指针类型的对象没有被初始化，是没有办法被使用的，会在编译阶段被报错，比如下例

```
String str;
System.out.println(str);
```

这个语句会过不了编译，因为IDE检查到你的str并没有被初始化

但是对于类包装的元素，编译器可能没有办法在编译时检查元素是否正常初始化，这时候会产生奇怪的效果，如下例：

```
// in tes.java
class tes {
    int num;
    String str;
    public tes(){
//        num=1;str=Integer.toString(num);
    }
    void print(){
        System.out.printf("%d %s\n",num,str);
    }
    @Override // 这个是一种标记，不需要管，后面会解释
    public String toString() {
        return "tes{" +
                "num=" + num +
                ", str='" + str + '\'' +
                '}';
    }
}
```

这个类的默认构造函数时不做任何事的，根据Java的机制，`num`会被初始化为0而`str`会被初始化为`null`，那这时使用`.println()`会产生什么结果呢？我们通过下面`main`中的内容可以看一下输出

```
// in Main.java
tes _t=new tes();
System.out.println(_t);
_t.print();
```

输出：

```
tes{num=0, str='null'}
tes{num=0, str='null'}
0 null
```

也就是实际上会输出一个null，所以有一个经典笑话就是一个程序员把字符串值改成`null`，另一个程序员就始终以为字符串没有被`new`（好冷啊）

同时我们也看到，`System.out.println(_t)`正常工作了，即使似乎我们并没有写过`println`关于`tes`类的重载

那么这都是为什么呢？下面我们将从这里出发，逐步讲解Java的输入输出

`print`和`println`区分

我们先区分这一对，看源码：

```
// in PrintStream.java
private BufferedWriter textOut; // 这是一个对象
private OutputStreamWriter charOut;

public void print(Object obj) {
    write(String.valueOf(obj));
}
public void print(String s) {
    write(String.valueOf(s));
}
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}
```

对于上述源码，我们可以看到`print`和`println`的差别（除了线程上）就是后者显式生成了一个`String`并调用了一次`print`和`newLine`

实际上，`println`里的`String s`只是`print`中`String.valueOf(obj)`生成的没有写出来的中间变量，所以本质上这里最终需要研究的就是

- `print()`函数
- `write()`函数
- `String.valueOf()`函数，在前面介绍过
- `newLine()`函数，可以理解成换行+刷新缓存区的函数，也就是C++中的`cout<<endl;`和`cout.flush();`

  另外，补充一个小点，Java中输出字符数组或者`String`时如果输出了`'\n'`的话会自动执行`newLine()`

下面就会研究前两个函数

`print`函数及其各个重载

直接看代码

```
// in PrintStream.java
public void print(Object obj) { // 数组会被传到这里
    write(String.valueOf(obj));
}
public void print(String s) {
    write(String.valueOf(s));
}
public void print(double d) {
    write(String.valueOf(d)); // 根据前文，这里调用的是Double.toString(d)
}
public void print(char s[]) {
    write(s);
}
```

这里只是一部分，但是已经可以代表所有的类：`String`、基本类型、其他类（包括非字符数组）、字符数组

可以看出，除了字符数组所有的`print`都调用了`String.valueOf()`函数（的对应重载），并将这个函数返回的`String`输入`write`

实际上字符数组也存在`String.valueOf()`的重载，但是其值是一个和内容不相关的地址字符串

`String.valueOf()`已经在字符那块讲解过了，我们看`write`函数

`write`函数及其各个重载

直接上源码

```
private void write(String s) {
    try {
        synchronized (this) {
```

```
                ensureOpen();
                textOut.write(s);
                textOut.flushBuffer();
                charOut.flushBuffer();
                if (autoFlush && (s.indexOf('\n') >= 0))
                    out.flush();
            }
        }
        catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        }
        catch (IOException x) {
            trouble = true;
        }
    }
    private void write(char buf[]) {
        try {
            synchronized (this) {
                ensureOpen();
                textOut.write(buf);
                textOut.flushBuffer();
                charOut.flushBuffer();
                if (autoFlush) {
                    for (int i = 0; i < buf.length; i++)
                        if (buf[i] == '\n')
                            out.flush();
                }
            }
        }
        catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        }
        catch (IOException x) {
            trouble = true;
        }
    }
```

可以看出`write`只可以接受字符数组或者`String`，并且会将输入的内容输出

调用`print`的输出结果总结

`String.valueOf()`、`print()`、`write()`的介绍，实际上已经可以预测任何一种类型的`print`输出的形式了，这里做个总结

类型	语句	结果预测	输出
基本类型	`System.out.print(3.14);`	正常输出基本类型的值 `boolean`的是`true`和`false`	`3.14`
字符数组	`char[]chs={'a','b'};` `System.out.print(chs);`	将字符数组当作字符串输出	`ab`
字符串	`System.out.print("ipLee");`	直接输出字符串	`ipLee`
非字符数组的数组 （直接输入）	`System.out.print(new int[2]);`	输出数组地址	`[I@2752f6e2`
非字符数组的数组 （使用`Arrays.toString()`）	`int[] ia=new int[2];` `System.out.print(Arrays.toString(ia));`	正常输出	`[0, 0]`

其他引用类型	`Main main=new Main();`
	`System.out.print(main);`
		相当于
		`Main main=new Main();`
		`System.out.print(main.toString())`
			*

*处是这样的：如果调用的是`Object`的`.toString()`（也就是没有覆写`.toString()`），那么输出`TestBasic.Main@2752f6e2`这样的信息，否则输出的是覆写的`.toString()`

`printf`方法

就是`System.out.printf("%.6f\n",4*pi);`会调用`printf`式输出，注意的是，`%f`用于`double`

Java的输入

`Scanner`类及常用函数

这里只介绍使用`Scanner`的输入输出。实际上Java的IO是一个大话题，在后面会专门提到

`Scanner`是一个类，而且这个类有个特点：如果想要正常使用必须生成一个专门的对象，比如如果想要从标准输入读取数据，就需要写：`Scanner cin = new Scanner(System.in);`

当然实际上`Scanner`的构造函数里接受的参数还可以有很多，比如`String`，`File`类，`InputStream`等等，但是无论是怎么构造出来的，其接口函数都是一样的

使用`String`初始化的和C++的`StringStream`很像

`.useDelimiter()`

一般情况下`Scanner`的分隔符是任意个空格，但是也可以专门制定分隔符。这里的制定需要用到正则表达式，这里简单探讨下简单应用：

如果想要用`'\n'`作为分隔符，可以使用`cin.useDelimiter("\n");`

如果想要用逗号及其两侧空格作为分隔符，可以使用`cin.useDelimiter("\\s*,\\s*");`，这里`\\s`表示空格，`*`表示匹配前一个字符出现0次或者无限次，即可有可无，也就是这个方法**会把与输入的正则表达式匹配的字符作为分隔符**

对于默认的，可以理解成`cin.useDelimiter("\\s+");`，其中`+`表示匹配前一个字符出现1次或者无限次，即至少有1次

如果同时使用逗号和空格作为分隔符，可以使用`cin.useDelimiter("[\\s,]+");`，其中`[]`表示字符集合。分隔`123,456,,,789   001,123  ,23`得到`123 456 789 001 123 23`，~~非常Amazing啊~~

另外，似乎任何情况下`'\n'`都会被视为一个分隔符？《？》

一些简单的正则表达式规则见CSDN:正则表达式匹配多个字符(4)

`.next()`

`.next()`就是分隔符意义下读入下一个字符串，分隔符会被略去

`.nextInt()`

实际上有`.nextInt()`、`.nextDouble()`等多种方法（但是没有`.nextChar()`），每一个都会将当前位置到下一个分隔符之间的内容转化成相应类型然后读进来。

关于能不能直接读取一个类对象，这个目前没有查到相关资料证明有这种重载