字符串和正则表达式

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握包装类与字符串和基本数据类型的转换
2.掌握常用类的基本使用
3.掌握枚举的基本使用
4.掌握异常的分类
5.掌握异常的常用结构
6.熟练使用自定义异常
```

1.2 教学难点:

```
1.枚举的进阶理解和使用
对于我们来说,枚举只要先会基本使用即可,对于进阶的部分,有精力的同学可以学习一下.
2.使用自定义异常
因为自定义异常是在理解异常的前提下实现的,所以稍微有一些难度.
```

二 教学目标

```
1.理解字符串的原理
2.掌握String的常用方法使用
3.掌握StringBuffer/StringBuilder的常用方法使用
4.熟练编写简单的正则表达式
```

三 教学导读

3.1. 字符串

我们进行软件开发的目的就是为人们的生活工作提供便捷,而人们表达感情最常用的方式就是语言,所以我们必须想办法将人们的语言通过计算机存储,传输.语言在计算机中的存在形式就是字符串.

字符串，是由若干个字符组成的一个有序序列。用String来表示一个字符串。
字符串中的内容，用双引号括起来。在双引号中，字符的数量不限制，可以是0个，可以是1个，也可以是多个。

String类:java将与字符串相关的功能面向对象了,形成了对应的类--字符串类.

例如大家最常见的:

```
String str1 = "hello world";
```

3.2. 正则表达式

正则表达式， 不是Java特有的。 是一套独立的， 自成体系的知识点。 在很多语言中， 都有对正则的使用。

正则表达式， 使用来做字符串的校验、匹配的， 其实正则只有一个作用：验证一个字符串是否与指定的规则匹配。

但是， 在很多的语言中， 都在匹配的基础上， 添加了其他的功能。 例如Java: 在匹配的基础上， 还添加了 删除、替换... 功能。

四 教学内容

4.1 字符串(会)

4.1.1. 字符串的分类

不可变字符串:

- 对应的类:String.
- 特点:字符串本身不能发生改变,与指向字符串的引用无关.
- 直接使用"",创建的是不可变字符串

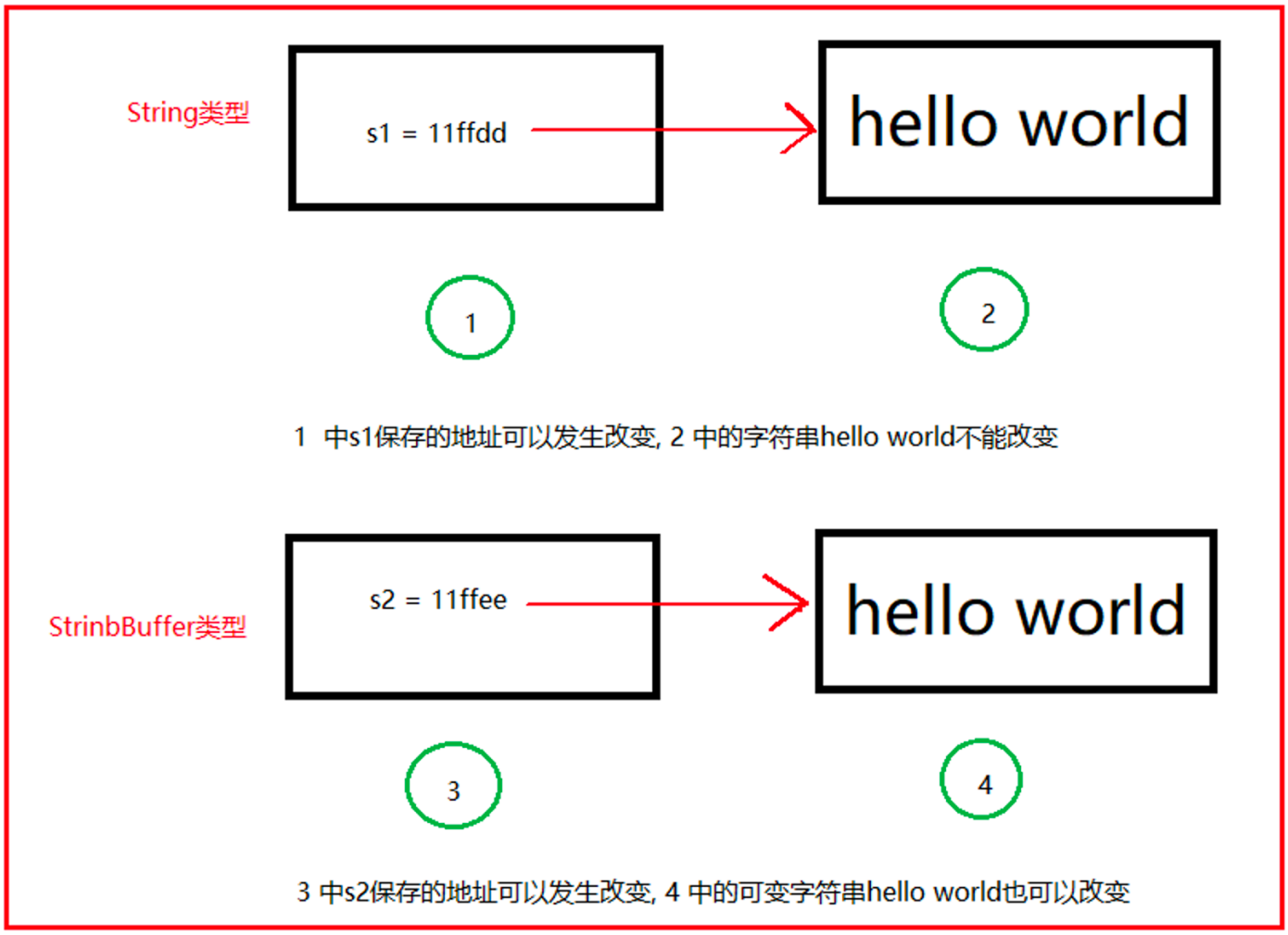
String s1 = "hello word";

可变字符串:

- 对应的类:StringBuilder/StringBuffer.
- 特点:字符串本身可以发生变化,与指向可变字符串的引用无关
- 创建可变字符串

  StringBuffer stringBuffer = new StringBuffer("hello world");

图示



4.1.2. 字符串的内存分析

字符串， 是一个引用数据类型。 但是字符串的引用， 和之前在面向对象部分的引用有一点差别。

差别： 类的对象， 是直接在堆上开辟的空间。 字符串， 是在 **常量池** 中开辟的空间。 （常量池， 是在方法区中的一个子空间）

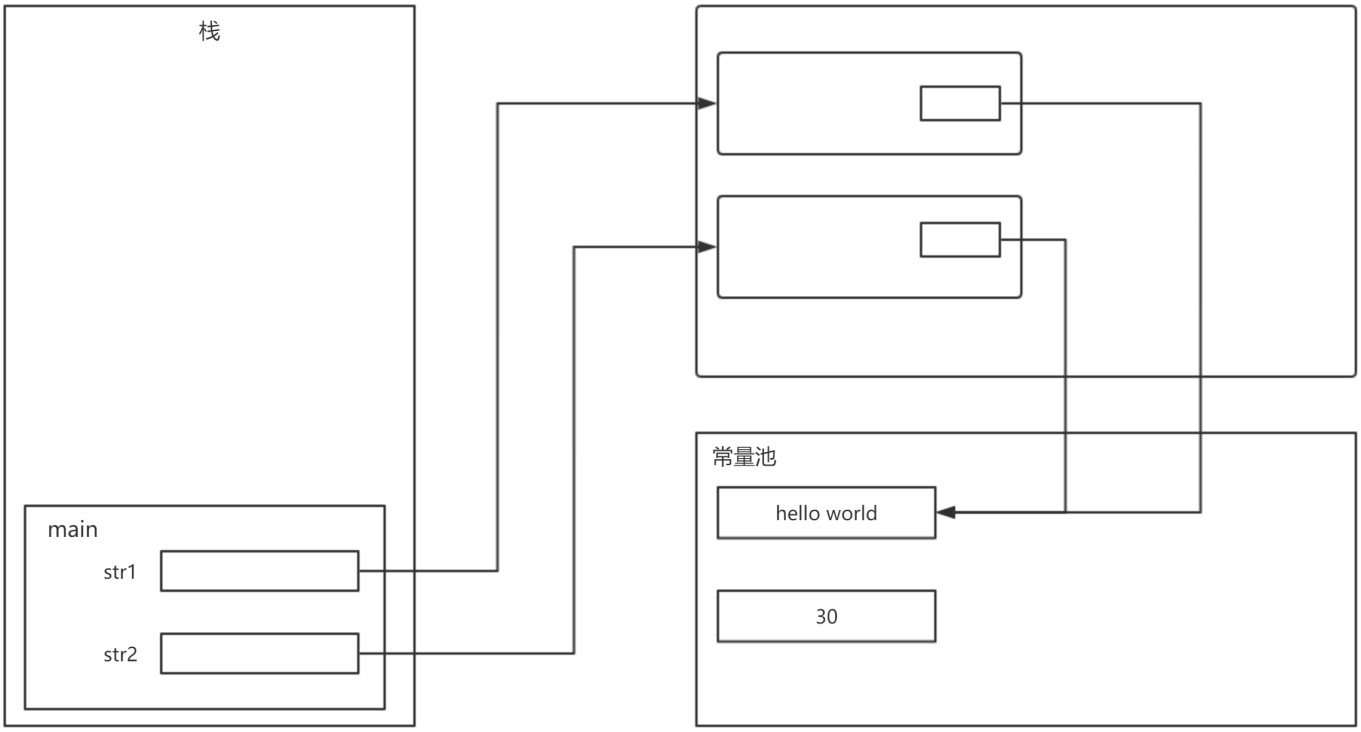
- **String str = "hello world";**
  - 此时， "hello world"， 是在 **常量池** 中开辟的空间。 str里面存储的， 其实是常量池中的某一个内存的地址。

- 当 str = "30"； 的时候， 其实， 并不是修改了 str指向的空间中的内容。 因为常量池空间特性， 一个空间一旦开辟完成了， 里面的值是不允许修改的。 此时， 是在常量池中开辟了一块新的空间， 存储了"30"， 并把这个新的空间的地址给str赋值了。
- 字符串类型， 之所以选择在常量池中进行空间的开辟， 而不是在堆上。 原因是需要使用 **享元原则** 。

```
// 当第一次使用到"hello world"这个字符串的时候， 常量池中并没有这块内存。
// 此时， 就需要去开辟一块新的空间， 存储为 "hello world"， 并且把空间的地址赋值给了str1。
String str1 = "hello world";

// 当再次使用到 "hello world" 这个字符串的时候， 常量池中现在是有这块空间的。
// 此时， 就不需要在开辟新的空间了， 直接将现有的这个空间地址赋值给 str2。
String str2 = "hello world";

// 即： str1 和 str2 现在都指向 "hello world"
System.out.println(str1 == str2);
```

- **String str = new String("hello world");**
  - String是一个Java中用来描述字符串的类， 里面是有构造方法的。
  - 通过String类提供的构造方法， 实例化的字符串对象， 在堆上开辟的空间。 在堆空间中， 有一个内部维护的属性， 指向了常量池中的某一块空间。

```java
// 在堆上开辟了一个String对象的空间， 把这个堆上空间的地址给了str1
// 在堆空间中， 有一个内部的属性， 指向了常量池中的 "hello world"
String str1 = new String("hello world");

// 在堆上开辟了一个String对象的空间， 把这个堆上空间的地址给了str2
// 在堆空间中， 有一个内部的属性， 指向了常量池中的 "hello world"
String str2 = new String("hello world");

System.out.println(str1 == str2);   // false: 因为此时 str1和str2 里面存储的是两块堆空间的地址。
System.out.println(str1.equals(str2));     // true: 因为在String类中，已经重写过equals方法了， 重写的实现为比较实际指向的常量池中的字符串。
```



课上练习

题目:有四个字符串,判断比较结果

```
String s1 = "1000phone";
String s2 = "1000phone";
String s3 = new String("1000phone");
String s4 = new String("1000phone");

System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s3 == s4);

//使用equals
System.out.println(s1.equals(s3));
```

题目分析:

```
当执行s1的时候,会到常量池找叫1000phone的字符串,如果有直接让s1保存他的地址,如果没有,会在常量区开辟一块儿空间存1000phone.
执行s2时同理s1
执行s3时,由于进行了new,一定会先在堆中开辟一块儿空间,而1000phone是作为参数传给了对象.保存在了对象的一个String类型的成员变量内,所以直接判断s1与s3不相同.
执行s4同理s3

所以最终结果:第一个true ,第二个false,第三个false

对于equals方法

结果是true,因为String默认重写了Object的equals方法,重新制定了比较规则,变成了让s1与s3属性的地址比较
总结:以后尽量使用equals进行String的比较
```

4.1.3 字符串拼接的内存分析

- **直接使用两个字符串字面量进行拼接**
  - 其实，就是直接将两个由双引号直接括起来的字符串进行拼接 。类似于 `String str = "hello" + "world";` 。
  - 这里,直接在常量池中进行空间操作。将常量池中拼接之后的结果，地址给 str 进行赋值。
- **使用一个字符串变量和其他的进行拼接**
  - 这里的拼接，不是在常量池中直接完成的。
  - 在这个拼接的过程中， 隐式的实例化了一个String类的对象， 在堆上开辟了空间。 堆上空间内部维护了一个指向了常量池中拼接结果的一个属性。 这个堆上的空间地址给左侧的引用进行了赋值。

4.1.2. 字符串的常用方法

4.1.2.1. 字符串的构造方法

- 字符串构造方法列举

构造方法	方法描述
String()	无参构造， 实例化一个空的字符串对象。 所谓的空字符串，其实是 "", 并不是null。
String(String str)	通过一个字符串， 实例化另外一个字符串。
String(char[] arr)	通过一个字符数组， 实例化一个字符串。 将字符数组中的所有的字符拼接到一起。
String(char[] arr, int offset, int count)	通过一个字符数组， 实例化一个字符串。 将字符数组中的指定范围的字符拼接到一起。
String(byte[] arr)	通过一个字节数组， 实例化一个字符串。 将字节数组中的所有的字节拼接成字符串。
String(byte[] arr, int offset, int count)	通过一个字节数组， 实例化一个字符串。 将字节数组中的指定范围的字节拼接成字符串。

- 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/14
 * @Description 字符串的构造方法
 */
public class StringMethod1 {
    public static void main(String[] args) {
        // 1. 无参构造， 实例化一个空的字符串对象。 所谓的空字符串，其实是 "", 并不是null。
        String s1 = new String();   // String s1 = "";
```

```
11          System.out.println(s1);
12
13          // 2. 通过一个字符串，  实例化另外一个字符串。
14          String s2 = new String("hello");
15          System.out.println(s2);
16
17          char[] arr1 = { 'h', 'e', 'l', 'l', 'o' };
18          // 3. 通过一个字符数组，  实例化一个字符串。将字符数组中
    的所有的字符拼接到一起。
19          String s3 = new String(arr1);
20          System.out.println(s3);
21          // 4. 通过一个字符数组，  实例化一个字符串。 将字符数组
    中的指定范围的字符拼接到一起。
22          String s4 = new String(arr1, 2, 3);
23          System.out.println(s4);
24
25          byte[] arr2 = { 97, 98, 99, 100, 101, 102, 103,
    104 };
26          // 5. 将一个字节数组中的每一个字节，转成对应的字符，再
    拼接到一起，组成一个字符串。
27          String s5 = new String(arr2);
28          System.out.println(s5);
29          // 6. 将一个字节数组中的offset位开始，取length个字
    节，将每一个字节，转成对应的字符，再拼接到一起，组成一个字符串。
30          String s6 = new String(arr2, 2, 4);
31          System.out.println(s6);
32      }
33  }
```

4.1.2.2. 字符串的非静态方法

因为字符串， 是常量。 任何的修改字符串的操作， 都不会对所修改的字符串造成任何的影响。 所有的对字符串的修改操作， 其实都是实例化了新的字符串对象。 在这个新的字符串中， 存储了修改之后的结果。 并将这个新的字符串以返回值的形式返回。 所以， 如果需要得到对一个字符串修改之后的结果， 需要接收方法的返回值。

- 9.2.2.2. 常用的非静态方法

返回值	方法	方法描述
String	concat(String str)	字符串拼接.将一个字符串与另一个字符串进行拼接并返回拼接之后的结果
String	substring(int beginIndex)	字符串截取。 从beginIndex开始， 一直截取到字符串的结尾。
String	substring(int beginIndex, int endIndex)	字符串截取。 截取字符串中[beginIndex, endIndex) 范围内的子字符串。
String	replace(char oldChar, char newChar)	字符串替换。 用新的字符替换原字符串中所有的旧的字符。
String	replace(CharSequence old, CharSequence newC)	字符串替换。 用新的字符序列替换原字符串中所有的旧的字符序列。
char	charAt(int index)	字符获取。 获取指定位的字符。
char[]	toCharArray()	将字符串转成字符数组。
byte[]	getBytes()	将字符串转成字节数组。
int	indexOf（char c)	获取某一个字符在一个字符串中第一次出现的下标。 如果没有出现， 返回 -1
		获取某一个字符在一个字符串中从

int	indexOf(char c, int fromIndex)	fromIndex位开始往后第一次出的下标。
int	lastIndexOf(char c)	获取某一个字符在一个字符串中最后一次出现的下标。
int	lastIndexOf(char c, int fromIndex)	获取某一个字符在一个字符串中,从fromIndex位开始往前最后一次出现的下标。
String	toUppeerCase()	将字符串中的所有的小写字母转成大写字母。
String	toLowerCase()	将字符串中的所有的大写字母转成小写字母。
boolean	isEmpty()	判断一个字符串是否是空字符串。
int	length()	获取一个字符串的长度。
boolean	contains(String str)	判断一个字符串中， 是否包含另外一个字符串。
boolean	startsWith(String prefix)	判断一个字符串， 是否是以指定的字符串作为开头。
boolean	endsWith(String shuffix)	判断一个字符串， 是否是以指定的字符串作为结尾。
String	trim()	去除一个字符串首尾的空格。
boolean	equals(Object obj)	判断两个字符串的内容是否相同。
boolean	equalsIgnoreCase(String str)	判断两个字符串的内容是否相同， 忽略大小写。
int	compareTo(String other)	对两个字符串进行大小比较。
int	compareToIgnoreCase(String other)	对两个字符串进行大小比较， 忽略大小写。

- 示例代码

```java
import java.util.Arrays;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/14
 * @Description 字符串常用的非静态方法
 */
public class StringMethod2 {
    public static void main(String[] args) {
        // 1. 字符串的拼接，效率比加号拼接高
        String ret1 = "hello".concat("world");
        System.out.println(ret1);       // helloworld

        // 2. 字符串截取
        String ret2 = "hello world".substring(3);
        System.out.println(ret2);       // lo world
        String ret3 = "hello world".substring(3, 8);
        System.out.println(ret3);       // lo wo
        // *. 字符序列截取,与字符串截取没有太大区别
        CharSequence charSequence = "hello world".subSequence(3, 8);
        System.out.println(charSequence);

        // 3. 用新的字符替换原字符串中所有的旧的字符
        String ret4 = "hello world".replace('l', 'L');
        System.out.println(ret4);       // heLLo worLd
        // 4. 用新的字符序列替换原字符串中所有的旧的字符序列
        String ret5 = "hello world".replace("ll", "~");
        System.out.println(ret5);       // he~o world

```

```
30
31          // 5. 转成字符数组
32          char[] ret6 = "hello world".toCharArray();
33          System.out.println(Arrays.toString(ret6));
// [h, e, l, l, o,  , w, o, r, l, d]
34          // 6. 转成字节数组
35          byte[] ret7 = "hello world".getBytes();
36          System.out.println(Arrays.toString(ret7));
// [104, 101, 108, 108, 111, 32, 119, 111, 114, 108,
100]
37
38
39          // 7. 获取某一个字符在一个字符串中第一次出现的下标。
40          int ret8 = "hello world".indexOf('L');
41          System.out.println(ret8);                    //
2
42          // 8. 获取某一个字符在一个字符串中从fromIndex位开始
往后，第一次出现的下标。
43          int ret9 = "hello world".indexOf('l', 4);
44          System.out.println(ret9);                    //
9
45          // 9. 获取某一个字符在一个字符串中最后一次出现的下标。
46          int ret10 = "hello world".lastIndexOf('o');
47          System.out.println(ret10);                   //
7
48          // 10. 获取某一个字符在一个字符串中，从fromIndex位开
始往前，最后一次出现的下标
49          int ret11 = "hello world".lastIndexOf('o', 5);
50          System.out.println(ret11);                   //
4
51
52          // 11. 字符串大小写字母转变
```

```
53          System.out.println("hello
    WORLD".toUpperCase());
54          System.out.println("hello
    WORLD".toLowerCase());
55
56          // 12. 判断一个字符串中，  是否包含另外一个字符串。
57          System.out.println("hello
    world".contains("loo"));
58
59          // 需求：判断一个字符串中是否包含某一个字符
60          // 答案：或者这个字符在字符串中出现的下标，如果不
    是-1，说明包含。
61
62          // 13. 判断一个字符串，  是否是以指定的字符串作为开头。
63          System.out.println("哈利波特与魔法
    石.mp4".startsWith("哈利波特"));
64          System.out.println("哈利波特与魔法
    石.mp4".endsWith(".mp4"));
65
66          // 14. 去除一个字符串首尾的空格
67          System.out.println("          hello world
    ".trim());
68
69          // 15. 判断两个字符串的内容是否相同
70          System.out.println("hello world".equals("HELLO
    WORLD"));
71          System.out.println("hello
    world".equalsIgnoreCase("HELLO WORLD"));  // true
72
73          // 16. 比较两个字符串的大小
74          // 大体的逻辑：
75          // > 0：前面的字符串 > 参数字符串
76          // == 0：两个字符串大小相等
```

```java
        // < 0: 前面的字符串 < 参数字符串

        /*
         * 字典顺序:按照ASCII表比较当前的两个字符,ASCII码大的认为是大的字符
         * 规则:从左边第一个字符开始比较
         * 如果当前的字符不相同,直接认为ASCII大的字符串是大字符串,后面的字符停止比较
         * 当前字符比较的具体规则:使用前面的字符-后面的字符,返回差值.如果是负数,说明前面的字符串小于后面的.反之前面的大.
         * 如果当前的字符相同,再去比较第二个字符,依次往后推,如果比到最后都相同,则认为两个字符串相等,差值返回0.
         * 如果两个字符串中一个是另一个从开始算的子串.比较的结果就是字符个数的差值,如果后面的是子串,差值为正数,
         * 前面的是子串,差值为负数
         */

        int result = "hello world".compareTo("hh");
        System.out.println(result);

        //17.切割: String[] split(String)
        String s6 = "h.e.l.l.o";
        //当我们将字符串中的某个字符作为刀,它就不再是内容
        String[]  strings1 = s6.split("\\.");//.是一个特殊字符,表示任意
        for (String ss:strings1) {
            System.out.println(ss);
        }
    }
}
```

4.1.2.3. 字符串的静态方法

- 常用的静态方法

返回值	方法	描述
String	join(CharSequence delimiter, CharSequence elements)	将若干个字符串拼接到一起，在拼接的时候，元素与元素之间以指定的分隔符进行分隔。
String	format(String format, Object... args)	以指定的格式， 进行字符串的格式化。

- 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/14
 * @Description
 */
public class StringMethod3 {
    public static void main(String[] args) {

        // 将若干个字符串拼接到一起，在拼接的时候，元素与元素之间以指定的分隔符进行分隔
        String str1 = String.join(", ", "lily", "lucy", "uncle wang", "polly");
        System.out.println(str1);

```

```
14          float score = 100;
15          String name ="xiaoming";
16          int age = 19;
17          // 大家好，我叫xiaoming，今年19岁了，本次考试考了100分。
18          /**
19           * 常见占位符：
20           * %s : 替代字符串          -> %ns: 凑够n位字符串，如果不够，补空格
21           * %d : 整型数字占位符      -> %nd: 凑够n位，如果不够补空格。
22           * %f : 浮点型数字占位符   -> %.nf: 保留小数点后面指定位的数字
23           * %c : 字符型占位符
24           */
25          String str2 = String.format("大家好，我叫%11s，今年%03d岁了，本次考试考了%.6f分。", name, age, score);
26          System.out.println(str2);
27      }
28  }
```

4.1.3. StringBuffer和StringBuilder类

4.1.3.1. 概念

都是用来操作字符串的类,我们成为可变字符串

字符串都是常量， 所有的操作字符串的方法， 都不能直接修改字符串本身。 如果我们需要得到修改之后的结果， 需要接收返回值。

StringBuffer和StringBuilder不是字符串类， 是用来操作字符串的类。 在类中维护了一个字符串的属性， 这些字符串操作类中的方法， 可以直接修改这个属性的值。 对于使用方来说， 可以不去通过返回值获取操作的结果。

在StringBuffer或者StringBuilder类中， 维护了一个字符数组。 这些类中所有的操作方法， 都是对这个字符数组进行的操作。

4.1.3.2. 常用方法

返回值	常用方法	方法描述
	构造方法()	实例化一个字符串操作类对象， 操作的是一个空字符串。
	构造方法(String str)	实例化一个字符串操作类对象， 操作的是一个指定的字符串。
StringBuffer/StringBuilder	append(...)	将一个数据拼接到现有的字符串的结尾。
StringBuffer/StringBuilder	insert(int offset, ...)	将一个数据插入到字符串的指定位。
StringBuffer/StringBuilder	delete(int start, int end)	删除一个字符串中 [start, end) 范围内的数据。 如果start越界了， 会出现下标越界异常。 如果end越界了， 没影响， 会将字符串后面的所有的内容都删除。
StringBuffer/StringBuilder	deleteCharAt(int index)	删除指定下标位的字符。
StringBuffer/StringBuilder	replace(int start, int end, String str)	替换，将字符串中 [start, end) 范围内的数据替换成指定的字符串。
void	setChatAt(int index, char c)	将指定下标位的字符， 替换成新的字符。
StringBuffer/StringBuilder	reverse()	将一个字符串前后倒置、翻转。
String	toString()	返回一个正在操作的字符串。

4.1.3.3. 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/15
 * @Description
 */
public class Test {
    public static void main(String[] args) {
        // 1. 构造方法
        StringBuilder sb = new StringBuilder("hello world");

        // 2. 增：在一个字符串后面拼接其他的字符串
        sb.append('!');

        // 3. 增：在指定的下标位插入一条数据
        sb.insert(3, "AAAAA");

        // 4. 删：删除字符串中的 [start, end) 范围内的数据
        sb.delete(3, 5);

        // 5. 删：删除指定位的字符
        sb.deleteCharAt(6);

        // 6. 截取一部分的字符串，这个操作不会修改到自己，如果希望得到截取的部分，需要接收返回值。
        String sub = sb.substring(4, 6);

        // 7. 替换，将字符串中 [start, end) 范围内的数据替换成指定的字符串
```

```
28          sb.replace(3, 6, "l");
29
30          // 8. 修改指定下标位的字符
31          sb.setCharAt(0, 'H');
32
33          // 9. 将字符串前后翻转
34          sb.reverse();
35
36          System.out.println(sb);
37      }
38  }
```

4.1.3.4. 区别

StringBuffer和StringBuilder从功能上来讲， 是一模一样的。 但是他们两者还是有区别的:

- StringBuffer是线程安全的。
- StringBuilder是线程不安全的。

使用场景:

- 当处于多线程的环境中， 多个线程同时操作这个对象， 此时使用StringBuffer。
- 当没有处于多线程环境中， 只有一个线程来操作这个对象， 此时使用StringBuilder。

4.1.3.5. 备注

但凡是涉及到字符串操作的使用场景， 特别是在循环中对字符串进行的操作。 一定不要使用字符串的方法， 用StringBuffer或者StringBuilder的方法来做。

由于字符串本身是不可变的， 所以String类所有的修改操作， 其实都是在方法内实例化了一个新的字符串对象， 存储拼接之后的新的字符串的地址， 返回这个新的字符串。 如果操作比较频繁， 就意味着有大量的临时字符串被实例化、被销毁， 效率极低。 StringBuffer、StringBuilder不同，在内部维护了一个字符数组， 所有的操作都是围绕这个字符数组进行的操作。 当需要转成字符串的时候， 才会调用 toString() 方法进行转换。 当频繁用到字符串操作的时候， 没有中间的临时的字符串出现， 效率较高。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/15
 * @Description String、StringBuffer、StringBuilder 拼接效率比较
 */
public class Test2 {
    public static void main(String[] args) {
        // 需求：进行字符串的拼接 100000 次。

        String str = "";
        StringBuffer buffer = new StringBuffer();
        StringBuilder builder = new StringBuilder();

        int times = 100000;

        // String类的拼接
        long time0 = System.currentTimeMillis();
        for (int i = 0; i < times; i++) {
            str += i;
        }
        long time1 = System.currentTimeMillis();

        // StringBuffer的拼接
```

```
        for (int i = 0; i < times; i++) {
            buffer.append(i);
        }
        long time2 = System.currentTimeMillis();

        for (int i = 0; i < times; i++) {
            builder.append(i);
        }
        long time3 = System.currentTimeMillis();

        System.out.println("String: " + (time1 - time0));
        System.out.println("StringBuffer: " + (time2 - time1));
        System.out.println("StringBuilder: " + (time3 - time2));
    }
}
```

课上练习一

模拟一个trim方法，去除字符串两端的空格。 思路： 1.判断字符串第一个位置是否是空格，如果是继续向下判断，直到不是空格为止。 结尾处判断空格也是如此。 2.当开始和结尾都判断到不是空格时，就是要获取的字符串。

```
class lx1
{
  public static void main(String[] args)
  {
    String str="   abcd  efg   ";
    String t=trim(str);
    System.out.println(t);
```

```
  }
  public static String trim(String str)
  {
    int start=0;
    int end=str.length()-1;
    while (str.charAt(start)==' ')
    {
      start++;
    }
    while(str.charAt(end)==' ')
    {
      end--;
    }
    return str.substring(start,end+1);
  }
}
```

课上练习二

将一个字符串进行反转。将字符串中指定部分进行反转，"abcdefg"; 思路： 1，曾经学习过对数组的元素进行反转。 2，将字符串变成数组，对数组反转。 3，将反转后的数组变成字符串。 4，只要将或反转的部分的开始和结束位置作为参数传递即可。

```

class lx1
{
  public static void main(String[] args)
  {
    String str="abcdefg";
    System.out.println(fanZhuan(str,2,5));
  }
```

```
  public static String fanZhuan(String str,int start,int end)
  {
    char[] t=str.toCharArray();
    daoxu (t,start,end);
    return  new String(t);

  }
  public static void daoxu(char arr[],int start,int end)
  {
    for (int i=start,j=end;i<j;i++,j--)
    {
      char ch;
      ch=arr[i];
      arr[i]=arr[j];
      arr[j]=ch;
    }

  }
}
```

课上练习三

设计一个方法，将一个字符串中的大小写字母翻转。

> 例如:
>
> reverse("heLLo woRld") -> HEllO WOrLD

```
public static String reverse(String str) {
    // 1. 将字符串转成字符数组
    char[] array = str.toCharArray();
    // 2. 遍历数组，修改大小写
```

```
    for (int i = 0; i < array.length; i++) {
        char c = array[i];
        if (c >= 'a' && c <= 'z') {
            array[i] -= 32;
        }
        else if (c >= 'A' && c <= 'Z') {
            array[i] += 32;
        }
    }
    // 3. 转成字符串返回
    return new String(array);
}
```

课上练习四

输入一个字符串,要求将字符从小到大排序并查找出字母a的个数

```
   public static void main(){
        String s1 = "hello aaa world";
        String[] tmp1  = test(s1);
        System.out.println(tmp1[0]+"    "+tmp1[1]);

    }
    public static String[] test(String str){
        char[] arr = str.toCharArray();
        //冒泡
        for (int i = 0; i <arr.length-1; i++) {
            for (int j = 0; j < arr.length-1-i; j++) {
                if(arr[j] > arr[j+1]){
                    arr[j] ^= arr[j+1];
                    arr[j+1] ^= arr[j];
                    arr[j] ^= arr[j+1];
```

```
                }
            }
        }

        int num = 0;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == 'a'){
                num++;
            }
        }

        String tmp = new String(arr);
        return  new String[]{tmp,""+num};
    }
}
```

课上练习五

定义接口，封装+-*/的方法，实现并测试。

课上练习六

对字符串中字符进行自然顺序排序。

```
1:将字符串转成字符数组
2：排序
2：将数组转成字符串
```

课上练习七

课上练习八

4.2. 正则表达式(了解)

4.2.1. 正则表达式的使用(会)

实现相同的功能， 用String、StringBuffer、StringBuilder可以实现， 用正则表达式也可以实现。

但是在实现过程中， 复杂程度是完全不一样的。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/15
 * @Description
 */
public class Regex1 {

    public static void main(String[] args) {
        System.out.println(checkWithRegex("123456"));
// true
        System.out.println(checkWithRegex("1234"));
// false

System.out.println(checkWithRegex("1234567890987"));
// false
        System.out.println(checkWithRegex("123.456"));
// false
        System.out.println(checkWithRegex("abcdefg"));
// false
        System.out.println(checkWithRegex("0123456"));
// false
    }

```

```java
    /**
     * 使用正则表达式，验证一个字符串是否是一个合法的QQ号码
     * @param str 字符串
     * @return 验证的结果
     */
    private static boolean checkWithRegex(String str) {
        return str.matches("[1-9]\\d{5,10}");
    }

    /**
     * 验证一个字符串是否是一个合法的QQ号码
     * @param str 字符串
     * @return 验证的结果
     */
    private static boolean check(String str) {
        // 1. 校验长度
        if (str.length() < 6 || str.length() > 11) {
            return false;
        }
        // 2. 校验纯数字组成
        try {
            Long.parseLong(str);
        }
        catch (NumberFormatException e) {
            // 说明不是纯数字，无法转成long型的变量
            return false;
        }
        // 3. 首字符校验
        return str.charAt(0) != '0';
    }
}
```

4.2.2. 基本的元字符

4.2.2.1. 正则表达式的匹配规则

逐个字符进行匹配， 判断是否和正则表达式中定义的规则一致。

以下借助 String 类中的 matches 方法进行正则基础的语法讲解。

> boolean matches(String regex);
>
> 是String类中的非静态方法， 使用字符串对象调用这个方法， 参数是一个正则表达式。
>
> 使用指定的正则表达式， 和当前的字符串进行匹配。 验证当前的字符串是否和指定的规则是相匹配的。

4.2.2.2. 元字符

元字符	意义
^	匹配一个字符串的开头。 在Java的正则匹配中，可以省略不写。
$	匹配一个字符串的结尾。 在Java的正则匹配中，可以省略不写。
[]	匹配一位字符。 [abc]: 表示这一位的字符， 可以是a、也可以是b、也可以是c 。 [a-z]: 表示这一位的字符， 可以是 [a, z] 范围内的任意的字符。 [a-zABC]: 表示这一位的字符， 可以是 [a,z] 范围内的任意字符， 或者A、或者B、或者C 。 [a-zA-Z]: 表示这一位的字符， 可以是任意的字母， 包括大写字母和小写字母。 [^abc]: 表示这一位的字符， 可以是除了 a、b、c 之外的任意字

	符。 [^a-z[hk]]: 表示这一位的字符， 不能是任意的小写字母， 但是 h 、 k 除外。
\	转义字符。 由于正则表达式在Java中是需要写在一个字符串中。 而字符串中的\也是一个转义字符。 因此Java中写正则表达式的时候， 转义字符都是 `\\` 使得某些特殊字符成为普通字符， 可以进行规则的指定。 使得某些普通字符变得具有特殊含义。
\d	匹配所有的数字， 等同于 [0-9] 。
\D	匹配所有的非数字， 等同于 [^0-9] 。
\w	匹配所有的单词字符， 等同于 [a-zA-Z0-9_] 。
\W	匹配所有的非单词字符， 等同于 [^a-zA-Z0-9_] 。
.	通配符， 可以匹配一个任意的字符。
+	前面的一位或者一组字符， 连续出现了一次或多次。
?	前面的一位或者一组字符， 连续出现了一次或零次。
*	前面的一位或者一组字符， 连续出现了零次、一次或多次。
{}	对前面的一位或者一组字符出现次数的精准匹配。 {m} : 表示前面的一位或者一组字符连续出现了m次。 {m,} ： 表示前面的一位或者一组字符连续出现了至少m次。 {m,n} : 表示前面的一位或者一组字符连续出现了至少m次，最多n次。
|	作用于整体或者是一个分组， 表示匹配的内容， 可以是任意的一个部分。 abc|123|opq : 表示整体的部分， 可以是 abc， 也可以是 123, 也可以是 opq

()	分组。 把某些连续的字符视为一个整体对待。

4.2.2.3. 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/15
 * @Description 正则表达式的基础语法
 */
public class Regex2 {
    public static void main(String[] args) {
        // 校验规则:
        System.out.println(",ello".matches("[ahj1,8]ello"));
        System.out.println("fello".matches("[a-z]ello"));
        // 需求: 首位字符，可以是任意的小写字母，或者是 HAQ
        System.out.println("Hello".matches("[Ha-zAQ]ello"));
        // 需求: 首位字符，可以是任意的字母，包括大写字母和小写字母
        System.out.println("hello".matches("[a-zA-Z]ello"));
        // 需求: 首位字符，可以是除了h之外的任意字符
        System.out.println("Hello".matches("[^hel]ello"));
        // 需求: 首位字符，不可以是任意的小写字母，但是 h、e、q 除外
```

```
19          System.out.println("lello".matches("[^a-z[heq]]ello"));
20
21          // 希望首位字符，可以是 h e [
22          System.out.println("]ello".matches("[he\\[]ello"));
23          // 希望首位字符，可以是 a - z
24          System.out.println("hello".matches("[az-]ello"));
25
26          System.out.println("hello".matches("hel+o"));
27          System.out.println("hello".matches("hel?o"));
28          System.out.println("heo".matches("hel*o"));
29
30          System.out.println("hello".matches("hel{3}o"));
31
System.out.println("hello".matches("hel{3,}o"));
32
System.out.println("hello".matches("hel{3,5}o"));
33
34          // 需求：匹配一个字符串可以是 126 或者是 163 或者是 qq 或者是 QQ
35
System.out.println("123".matches("126|163|qq|QQ"));
36          // 需求：匹配一个qq邮箱、126邮箱、163邮箱
37
System.out.println("admin@sina.com".matches("admin@(qq|126|163)\\.com"));
38      }
39  }
```

4.2.3. String类中的常用的方法

4.2.3.1. 字符串的匹配

```
// QQ号的规则：
//      1. 纯数字组成的
//      2. 不能以0作为开头
//      3. 长度在 [6,11]
private static boolean chechQQ(String qqNumber) {
    return qqNumber.matches("[1-9]\\d{5,10}");
}

// 验证邮箱的合法性
//      1. 126
//      2. 前半部分可以是任意的单词字符，长度限制在 [4,12]
//      3. 以.com作为结尾
private static boolean checkEmail(String email) {
    return email.matches("\\w{4,12}@126\\.com");
}
```

4.2.3.2. 字符串的切割

```
private static void split() {
    // 需求：将一个存储有所有的姓名的字符串，切割出每一个名字。
    String names = "lily        lucy            polly Jim       LiLei       HanMeimei";
    // 实现：借助字符串的一个方法 split(String regex)
    //      将字符串中，满足正则规则的子部分，切割掉
    String[] nameArr = names.split(" +");
    System.out.println(Arrays.toString(nameArr));
}
```

4.2.3.3. 字符串的替换

```
private static void replace() {
    // 需求：将这个存储有所有的名字的字符串，名字之间的分隔用，来替代
    String names = "lily        lucy            polly        Jim        LiLei       HanMeimei";
    // 实现：借助字符串中的一个方法 replaceAll(String regex, String replacement)
    //       将字符串中，满足指定正则的部分，替换成 replacement
    // String result = names.replaceAll(" +", ", ");
    String result = names.replaceFirst(" +", ", ");
    System.out.println(result);
}
```

```
private static void example() {
    // 需求：将一个手机号的中间4位替换成 ****
    //       17788889999 => 177****9999
    String phoneNumber = "17788889999";
    // 在replace方法中，使用$1获取第一个分组的值
    String result = phoneNumber.replaceAll("(1\\d{2})(\\d{4})(\\d{4})", "$1****$3");
    System.out.println(result);
}
```

4.2.4. Pattern和Matcher类

4.2.4.1. 简介

在 java.util.regex 包中

Pattern类： 在Java中，正则表达式的载体。使用正则表达式进行字符串的校验、切割、替换，都需要使用到这个类。

> 在Java中，使用字符串的形式来写正则表达式。此时，如果需要这个字符串被当做是一个正则表达式使用， 必须先由这样的字符串，编译为 Pattern 类的对象。然后才可以使用这个对象的某些方法， 进行常见的操作（校验、切割、替换）。

Matcher类： 在Java中， 对一个正则校验的结果描述。

4.2.4.2. 常用方法

Pattern

修饰符&返回值	方法	描述
static boolean	matches(String regex, CharSequence sequence)	静态的规则校验， 直接校验某一个字符串是否符合某一个规则。
static Pattern	compile(String regex)	将一个字符串， 编译为Pattern对象， 从而可以当做是一个正则表达式使用。
String[]	split(CharSequence sequence)	将一个字符序列， 按照指定的规则进行切割， 得到每一个切割部分。
String[]	split(CharSequence sequence, int limit)	将一个字符序列， 按照指定的规则进行切割， 切割成指定的段数， 得到每一个切割部分。
Matcher	matcher(CharSequence sequence)	将一个正则表达式和一个字符串进行校验。

Matcher

修饰符&返回值	方法	描述
boolean	matches()	得到本次整体匹配的结果。 一个字符串是否和一个正则表达式是整体匹配的。
boolean	**find()**	查找字符串中， 是否有满足指定规则的子部分。
boolean	find(int start)	查找字符串中， 从指定的下标位开始往后，有没有满足指定规则的子部分。
String	group()	获取当前匹配到的字符串。
int	start()	获取当前匹配到的部分首字符的下标。
int	end()	获取当前匹配到的部分尾字符的下标 + 1。
boolean	lookingAt()	查找字符串中， 是否有满足指定规则的子部分。 每次查询都从第0位开始。

find()

从字符串的第0位开始查询， 查询是否有满足指定规则的子部分。

当再次调用find()的时候， 从上次查询到的位置开始， 继续向后查询。

关于查询

无论是 matches， 还是find， 还是lookingAt， 在字符串中进行校验、匹配的时候， 是有一个浮标存在的。

4.2.4.3. 示例代码

Pattern 类的使用：

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/16
 * @Description Pattern类 和 Matcher类
 */
public class Regex1 {
    public static void main(String[] args) {
        // 1. 静态的校验
        boolean ret = Pattern.matches("[1-9]\\d{5,10}",
"123456");
        System.out.println(ret);

        // 2. 将一个字符串编译为正则表达式对象(Pattern对象)
        Pattern pattern = Pattern.compile(" +");
        // 3. 字符串切割
        String[] array1 = pattern.split("Lily     Lucy
Polly     Jim     Tom");
        System.out.println(Arrays.toString(array1));

        String[] array2 = pattern.split("Lily     Lucy
Polly     Jim     Tom", 0);
        System.out.println(Arrays.toString(array2));
    }
}
```

Matcher:

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/16
 * @Description Pattern和Matcher类的综合使用
 */
public class Regex2 {
    public static void main(String[] args) {
        // 1. 准备一个数据源
        String data = "abc123hello world456lily789lucy012uncle111wang";

        // 2. 将一个字符串变异成正则对象
        Pattern pattern = Pattern.compile("\\d+");

        // 3. 使用一个正则和一个字符串进行校验
        Matcher matcher = pattern.matcher(data);

        // 3.1. 获取整体校验的结果
        // 匹配逻辑:
        //      从第0个字符开始，判断每一个字符是否否则当前的正则。
        //      当找到了有不符合当前正则部分的时候，就会停止继续向后查找，直接返回false
        System.out.println(matcher.matches());

        // 3.2. find()
        while (matcher.find()) {
            System.out.print("匹配到了字符串: " + matcher.group() + ", ");
```

```
            System.out.println("下标范围: [" + matcher.start() + ", " + matcher.end() + ")");
        }

        // 3.3. lookingAt()
        System.out.println(matcher.lookingAt() + ", " + matcher.group() + ", [" + matcher.start() + ", " + matcher.end() + ")");
    }
}
```

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Date 2020/4/16
 * @Description 每一个分组内容的获取
 */
public class Regex3 {
    public static void main(String[] args) {
        //
        Pattern pattern = Pattern.compile("(1\\d{2})(\\d{4})(\\d{4})");
        Matcher matcher = pattern.matcher("17788889999");

        // 获取分组的数量
        System.out.println(matcher.groupCount());
        // 获取每一个分组的值之前，先进行整体的匹配
        boolean ret = matcher.matches();
        // 获取到某一个分组的内容
```

```
21            System.out.println(matcher.group(1));
22        }
23    }
```