第十七章 使用基础API

1. API（概述）

1. JavaAPI（Java Application Programming Interface）：Java应用程序接口
2. 之前Oracle公司程序员写好的类，接口，方法等，供其他程序员使用
3. 我们要学习API就是要学习别人写好的类，接口和方法等
    a. 类：干什么用的，包含哪些方法
    b. 方法：干什么用的，怎么用
        i. 是否为静态
        ii. 参数列表
        iii. 返回类型

2. 包装类

2.1 包装类的概述

代码块

```
Java为什么要设计包装类：
    1.Java语言是面向对象的语言，基本类型具有面向对象的特征（属性和方法），为了让基本类型具有面向对象的特征，
    所以为每个基本类型都设计了一个与之对应的包装类
    2.包装类：提供了一些列的方法，供我们使用，比如：基本类型  转换  为包装类等
    3.包装类还提供了对集合的支持
    4.包装类还提供了对泛型的支持
    基本类型   ------------------------------ 包装类
    byte      ------------------------------ Byte
    short      ------------------------------Short
    int      ------------------------------ Integer
    long      ------------------------------Long
    float      ------------------------------Float
    double      ------------------------------Double
    char      ------------------------------ Character
    boolean       ------------------------------Boolean

```

2.2 包装类与基本类型的转换

- 装箱：基本类型 --> 包装类
  - 自动装箱
  - 手动装箱
- 拆箱：包装类 --> 基本类型
  - 自动装箱
  - 手动装箱

代码块

```
package com.powernode.api13;

import javax.xml.crypto.dom.DOMCryptoContext;

public class Test01 {
    public static void main(String[] args) {
        /**
         *   - 装箱：基本类型  --> 包装类
         *      - 自动装箱
         *      - 手动装箱
         *   - 拆箱：包装类 --> 基本类型
         *      - 自动拆箱
         *      - 手动拆箱
         */
        //1.装箱：基本类型  --> 包装类
        Integer i = 123;//自动装箱
        String str = "123";
        Integer i1 =  Integer.valueOf(str);//手动装箱
        System.out.println("-----------");
        //2.拆箱：包装类 --> 基本类型
        Integer i2 = 123;
        int x = i2;//自动拆箱
        Double d = 1.1;
        int i3 = d.intValue();//手动拆箱
        //3.包装类提供了其他的方法
        System.out.println(Integer.max(2,3));
        //把字符串转换为整数
        int i4 = Integer.parseInt("123");
    }
}
```

2.3 整数型常量池源码分析

- Integer类中有一个 静态内部类 IntegerCache
- IntegerCache 内部类中有个static{}
- 加载Integer类静态块会执行

代码块

```
1   private static class IntegerCache {
2       static final int low = -128;//最小值
3       static final int high;//最大值
4       static final Integer[] cache;//缓存
5       static Integer[] archivedCache;
6
7       static {
8           // high value may be configured by property
9           int h = 127;
10          String integerCacheHighPropValue =
11              VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
12          if (integerCacheHighPropValue != null) {
13              try {
14                  h = Math.max(parseInt(integerCacheHighPropValue), 127);
15                  // Maximum array size is Integer.MAX_VALUE
16                  h = Math.min(h, Integer.MAX_VALUE - (-low) -1);
17              } catch( NumberFormatException nfe) {
18                  // If the property cannot be parsed into an int, ignore it.
19              }
20          }
21          high = h;//high = 127
22
23          // Load IntegerCache.archivedCache from archive, if possible
24          CDS.initializeFromArchive(IntegerCache.class);
25          // size = (127 - (-128)) + 1 = 256
26          int size = (high - low) + 1;
27
28          // Use the archived cache if it exists and is large enough
29          if (archivedCache == null || size > archivedCache.length) {
30              Integer[] c = new Integer[size];//创建了长度为256的Integer数组
31              int j = low;//j = -128
32              for(int i = 0; i < c.length; i++) {
33                  c[i] = new Integer(j++);
34                  //c[0] = new Integer(-128)
35                  //c[1] = new Integer(-127)
36                  //把-128到127存储到Integer数组中
37              }
```

```
            archivedCache = c;
        }
        cache = archivedCache;
        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

2.4 整数型常量池

代码块

```
package com.powernode.api13;

public class Test02 {
    public static void main(String[] args) {
        int i1 = 123;
        System.out.println("i1 = " + i1);
        int i2 = 123;
        System.out.println("i2 = " + i2);
        System.out.println(i1 == i2);//true
        System.out.println("==============");
        /**
         * 1.Integer类加载的时候会执行IntegerCache中的静态块
         * 2.IntegerCache静态快中，会值自动创建一个长度为256的Integer数组
         * 3.Integer[] c = new Integer[size];//创建了长度为256的Integer数组
         *     for(int i = 0; i < c.length; i++) {
         *         c[i] = new Integer(j++);
         *         //c[0] = new Integer(-128)
         *         //c[1] = new Integer(-127)
         *         //把-128到127存储到Integer数组中
         *    }
         * 4.创建完毕会把数组放入cache中（static final Integer[] cache;//缓存）
         * 5.cahe是static常量，所以引用在元空间，对象在堆中，我们也称为静态常量池
         * 6.Java创建Integer对象时，首先从静态常量池中取，如果没有才创建，如果有直接用
         *
         *
         */
        Integer i3 = 123;
        System.out.println("i3 = " + i3);
        Integer i4 = 123;
        System.out.println("i4 = " + i4);
        System.out.println(i3 == i4);//true
        System.out.println("---------------");
```

```
33
34          Integer i5 = 128;
35          System.out.println("i5 = " + i5);
36          Integer i6 = 128;
37          System.out.println("i6 = " + i6);
38          System.out.println(i5 == i6);//false
39
40      }
41  }
```

作业

1. 练习题(枚举)
   a. 声明Week枚举类，其中包含星期一至星期日的定义；
   b. 在TestWeek类中声明方法printWeek(Week week)，根据参数值打印相应的中文星期字符串。
   c. 在main方法中从命令行接收一个1-7的整数，分别代表星期一至星期日，打印该值对应的枚举值，然后以此枚举值调用printWeek方法，输出中文星期。

代码块

```
1   package com.powernode.exercise01;
2
3   import java.util.Scanner;
4
5   enum Week {
6       MONDAY,
7       TUESDAY,
8       WEDNESDAY,
9       THURSDAY,
10      FRIDAY,
11      SATURDAY,
12      SUNDAY;
13  }
14
15  public class TestWeek {
16      public static void main(String[] args) {
17          /**
18           * 练习题
19           * 1. 声明Week枚举类，其中包含星期一至星期日的定义；
20           * 2. 在TestWeek类中声明方法中printWeek(Week week)，根据参数值打印相应的中文
    星期字符串。
```

```
21              * 3. 在main方法中从命令行接收一个1-7的整数，分别代表星期一至星期日，
22              * 打印该值对应的枚举值，然后以此枚举值调用printWeek方法，输出中文星期。
23              */
24             int index = new Scanner(System.in).nextInt();
25             Week[] values = Week.values();
26             try {
27                 Week  week = values[index - 1];
28                 printWeek(week);
29             } catch (ArrayIndexOutOfBoundsException e) {
30                 System.out.println("输入的数据不合法，请输入(1-7)");
31             }
32
33         }
34
35         public static void printWeek(Week week) {
36             switch (week){
37                 case MONDAY -> System.out.println("星期一");
38                 case TUESDAY -> System.out.println("星期二");
39                 case WEDNESDAY -> System.out.println("星期三");
40                 case THURSDAY -> System.out.println("星期四");
41                 case FRIDAY -> System.out.println("星期五");
42                 case SATURDAY -> System.out.println("星期六");
43                 case SUNDAY -> System.out.println("星期日");
44             }
45         }
46     }
```

2. 练习题（try-catch）
   a. 编写TestException类，在控制台中接收两个整型参数，并用第一个数除以第二个数（不为零），打印结果。
   b. 在两数相除的下一行输出“测试除数为零是否输出”；
   c. 测试如下： 其中第一个数不为0，第二个参数为0
   d. 使用try-catch对异常处理
3. 练习题(自定义异常)
   a. 声明受检异常类IlleagalNumberException，用来表示无效数字异常
   b. 在TestException类中声明int divide(int m, int n)方法，该方法可抛出IlleagalNumberException异常。
      i. 方法的两个参数分别为被除数和除数，返回值为商
      ii. 如果除数为0，则方法抛出IlleagalNumberException异常
   c. 在TestException类的main方法中调用divide方法计算商值打印输出，并捕获可能出现的异常。

备注：获得控制台输入语句

代码块

```
Scanner scanner = new Scanner(System.in);
int x = scanner.nextInt();
int y = scanner.nextInt();
```

3. String字符串

3.1 字符串的==

代码块

```
package com.powernode.string03;

public class Test01 {
    public static void main(String[] args) {
        //1.==
        String s1 = "123";
        String s2 = "123";
        /**
         * 字符串常量池
         *   1.s1 = "123"
         *   2.首先去常量池中找有没有字符串123
         *   3.如果有拿来使用
         *   4.如果没有创建字符串“123”，创建后放入常量池
         *   5.s2 = "123",在常量池中找到了字符串“123”
         *   6.所以 s1 == s2 ,同一块内存
         */
        System.out.println(s1 == s2);//true
        //强制开辟了新的空间存储123，不去常量池中取
        String s3 = new String("123");
        System.out.println(s1 == s3);//false
        System.out.println("=================");
        //2.hashCode是否相等
        System.out.println(s1.hashCode() == s2.hashCode());//true
        System.out.println(s2.hashCode() == s3.hashCode());//true
        //3.物理hash：因为s1 和 s2 是同一块内存，s3新开辟的内存，字符串重写了hashCode，无法覆盖物理hash值
        System.out.println(System.identityHashCode(s1));
        System.out.println(System.identityHashCode(s2));
        System.out.println(System.identityHashCode(s3));
```

```
    }
}
```

3.2 String类是final的，为什么可以改变值

代码块

```
package com.powernode.string03;

public class Test02 {
    public static void main(String[] args) {
        String s1 = "123";
        System.out.println(System.identityHashCode(s1));
        s1 = "456";
        System.out.println(System.identityHashCode(s1));
        /**
         * String是final的
         *    1.对字符串的修改，不是改变了值
         *    2.而是开辟了新的空间，给与新的内存地址
         */
    }
}
```

3.3 字符串字面量拼接

代码块

```
package com.powernode.string03;

public class Test03 {
    public static void main(String[] args) {
        /**
         * 1. + 运算符的两边都是【字面量】的时候，运算是在编译时完成的
         * 2. 在生成.class文件的代码里面，实际上就是拼接后的“abcdef”
         * 3. 程序执行到【s1 = "abc" + "def"】,先去常量池中找“abcdef”
         * 4. 如果不存在，就创建“abcdef”，放入常量池
         * 5. 执行到【s2 = "abcdef"】，从常量池中取到“abcdef”
         */
        String s1 = "abc" + "def";
        String s2 = "abcdef";

        System.out.println(s1 == s2);//true
    }
}
```

3.4 字符串变量的拼接

代码块

```
package com.powernode.string03;

public class Test04 {
    public static void main(String[] args) {
        //如下程序创建了几个对象
        //1.创建x对象放入常量池
        String s1 = "x";
        //2.+ 底层创建一个StringBuilder对象进行拼接
        //3.创建y对像放入常量池
        //4.拼接后的StringBuilder对象，转换为String对象（s2),s2不放入常量池，因为s2拼接的是变量
        String s2 = s1 + "y";
        //5.创建xy对象放入常量池
        String s3 = "xy";
        System.out.println(s2 == s3);//false
    }
}
```

3.5 字符串常量拼接

代码块

```
package com.powernode.string03;

public class Test05 {
    public static void main(String[] args) {
        /**
         * 常量池中有几个对象
         *    1."abc"
         *    2."abcdef"
         */
        //s1 = "abc" ,final修饰了s1，不可以改变值
        final String s1 = "abc";//创建对象abc
        //因为s1常量，因此【s1 + "def"】也是在编译期完成拼接，底层不会创建StringBuilder对象
        //创建对象放入常量池
        String s2 = s1 + "def";
        System.out.println(System.identityHashCode(s2));
        //s3从常量池取"abcdef"
        String s3 = "abcdef";
        System.out.println(System.identityHashCode(s3));
        System.out.println(s2 == s3);//true
```

```
    }
}
```

3.6 String构造器

代码块

```
package com.powernode.string04;

import java.io.UnsupportedEncodingException;

public class Test01 {
    public static void main(String[] args) throws Exception {
        /**
         * String() 创建空String对象
         * String(String original)  指定字符串创建对象
         * String(byte[] bytes, String charsetName) 按照指定字符集编码创建String对象。
         * String(char[] value) 通过字符数组创建一个字符串对象
         */
        String s1 = new String();
        /*String s2 = new String("你好");
        byte[] bytes = s2.getBytes();//获得字节数组
        //UTF-8,utf-8,utf8,UTF8,国际编码，代表的是一个意思*/
        String s3 = new String("你好".getBytes(), "UTF-8");
        System.out.println(s3);

        //GBK,gbk,GB2312,GB18030,代表都是中文编码
        String s4 = new String("你好".getBytes(), "GBK");
        System.out.println(s4);
        //默认编码格式
        System.out.println(System.getProperty("file.encoding"));
        //统一使用GBK编码
        String s5 = new String("你好".getBytes("GBK"), "GBK");
        System.out.println(s5);

        char[] chars = {'a', 'b', 'c'};
        //通过字符数组创建对象
        String s6 = new String(chars);
    }
}
```

3.7 String常用方法（一）

```java
package com.powernode.string04;

public class Test02 {
    public static void main(String[] args) {
        /**
         * length() 可以获取到字符串中有多少个字符
         * charAt(int index)    返回指定索引位置的char值
         * int indexOf(int ch)  返回指定字符第一次出现在字符串内的索引。如果没有找到返回 -1
         * int indexOf(int ch, int fromIndex)   返回指定字符第一次出现在字符串内的索引，以指定的索引开始搜索。如果没有找到返回 -1
         * int indexOf(String str)  返回指定子字符串第一次出现在字符串内的索引。如果没有找到返回 -1
         * int indexOf(String str, int fromIndex)   返回指定子串的第一次出现在字符串中的索引，从指定的索引开始搜索。如果没有找到返回 -1
         * int lastIndexOf(int ch)  返回指定字符最后一次出现在字符串内的索引,如果没有找到返回 -1
         * int lastIndexOf(int ch, int fromIndex)   返回指定字符最后一次出现在字符串内的索引，以指定的索引开始反向搜索，没找到为 -1
         * int lastIndexOf(String str)  返回指定子字符串最后一次出现在字符串内的索引，如果没有找到返回 -1
         * int lastIndexOf(String str, int fromIndex)   返回指定子串的最后一次出现在字符串中的索引，从指定的索引开始反向搜索，没找到为 -1
         */
        String s = "abcABCabc";
        System.out.println(s.length());//9
        System.out.println(s.charAt(0));//a
        System.out.println(s.indexOf(97));//0
        System.out.println(s.indexOf(97, 2));//6
        System.out.println(s.indexOf("a"));//0
        System.out.println(s.indexOf("a", 2));//6
        System.out.println(s.lastIndexOf(97));//6
        System.out.println(s.lastIndexOf(97, 5));//0
        System.out.println(s.lastIndexOf("a"));//6
        System.out.println(s.lastIndexOf("a", 5));//0

    }
}
```

3.8 String常用方法（二）

代码块

```java
package com.powernode.string04;

public class Test03 {
```

```
4       public static void main(String[] args) {
5           /**
6            * startsWith(String prefix)    判断字符串是否以指定字符串开头，返回值为
    boolean类型。
7            * endsWith(String prefix)  判断字符串是否以指定字符串结尾，返回值为boolean
    类型。
8            * contains(CharSequence s) 判断字符串中是否包含指定字符串，返回值为boolean
    类型
9            * char[] toCharArray() 将此字符串转换为新的字符数组
10           * byte[] getBytes()    得到一个操作系统默认的编码格式的字节数组
11           * String toUpperCase() 返回一个新的字符串，该字符串中所有英文字符转换为大写字
    母
12           * String toLowerCase() 返回一个新的字符串，该字符串中所有英文字符转换为小写字
    母
13           */
14          String s = "abcABCabc";
15          System.out.println("s.startsWith(\"abc\") = " +
    s.startsWith("abc"));//true
16          System.out.println("s.endsWith(\"abc\") = " + s.endsWith("abc"));//true
17          System.out.println("s.contains(\"abc\") = " + s.contains("abc"));//true
18          char[] charArray = s.toCharArray();
19          byte[] bytes = s.getBytes();
20          System.out.println("s.toUpperCase() = " + s.toUpperCase());
21          System.out.println("s.toLowerCase() = " + s.toLowerCase());
22          /**
23           * String substring(int beginIndex) 从beginIndex开始截取字符串，到字符串末
    尾结束
24           * String substring(int beginIndex, int endIndex)   从beginIndex开始截取
    字符串，到字符索引endIndex-1结束
25           * String replace(char oldChar, char newChar)   通过用newChar字符替换字符
    串中出现的所有oldChar字符，并返回替换后的新字符串
26           * String replace(CharSequence target, CharSequence replacement)    将需
    要替换的字符串（target）替换为指定字符串（replacement）
27           * String concat(String str)    ，将指定的字符串连接到该字符串的末尾，该方法
    实现的功能和“+”连接符比较相似。
28           * boolean isEmpty()    判断字符串内容是否为空。当字符串长度为0，则返回true，
    否则返回false。
29           * boolean equals(Object anObject)  判断字符串内容是否相同
30           * boolean equalsIgnoreCase(String str) 判断字符串内容是否相同，忽略字母大小
    写
31           * valueOf(Object obj)  方法是一个静态方法，可以把基本数据类型转化为字符串类型
32           */
33          s = "abcABCabc";
34          System.out.println("s.substring(1) = " + s.substring(1));//"bcABCabc"
35          System.out.println("s.substring(1,3) = " + s.substring(1, 3));//bc
36          System.out.println("s.replace('b','x') = " + s.replace('b',
    'x'));//axcABCaxc
```

```
37          System.out.println("s.replace(\"ab\",\"xx\") = " + s.replace("ab",
    "xx"));//xxcABCxxc
38          /**
39           * concat 和 + 的区别
40           *    1.concat如果拼接的字符串为“”不会创建新的对象
41           *    2.+ 拼接的字符串如果为“”也会创建新的对象
42           *    3.如果拼接都不为“”都会创建新的对象
43           */
44          String s1 = "abc";
45          String s2 = s1.concat("");
46          System.out.println(s1 == s2);//true
47
48          String s3 = "abc";
49          String s4 = s3 + "";
50          System.out.println(s3 == s4);//false
51          /**
52           * isEmpty 和 null
53           *   1.isEmpty有对象，没有内容，内存有开辟空间
54           *   2.null,没有对象，内存没有开辟空间
55           */
56          String s5 = "";
57          System.out.println("s5.isEmpty() = " + s5.isEmpty());//true
58          System.out.println("\"abc\".equals(\"ABC\") = " +
    "abc".equals("ABC"));//false
59          System.out.println("\"abc\".equalsIgnoreCase(\"ABC\") = " +
    "abc".equalsIgnoreCase("ABC"));//true
60          //把基本类型转换为字符串
61          String s6 = String.valueOf(123);
62
63      }
64  }
```

3.9 String去除空格专项

代码块

```
1   package com.powernode.string04;
2
3   public class Test04 {
4       public static void main(String[] args) {
5           /**
6            * String trim()    去除首,尾空格
7            * String strip()   去除首,尾空格，Java 11中引入了strip()方法作为trim()的替
    代品
8            * String stripLeading()    去除首空格
9            * String stripTrailing()   去除尾空格
```

```
10            * replace(" ","")  去除所有空格
11            * replace(" ","FF")    把空格替换为FF
12            */
13           String s = "  abc  ";
14           System.out.println("--" + s + "--");
15           System.out.println("--" + s.trim() + "--");
16           System.out.println("--" + s.strip() + "--");
17           System.out.println("--" + s.stripLeading() + "--");
18           System.out.println("--" + s.stripTrailing() + "--");
19
20           s = "  a bc  ";
21           System.out.println("--" + s + "--");
22           System.out.println(s.replace(" ", ""));
23           System.out.println(s.replace(" ", "FF"));
24           /**
25            * --  abc  --
26            * --abc--
27            * --abc--
28            * --abc  --
29            * --  abc--
30            * --  a bc  --
31            * abc
32            * FFFFaFFbcFFFF
33            */
34       }
35   }
```

3.10 String使用正则表达式

代码块

```
1   正则表达式常见符号:
2   1.元字符
3   .          匹配除换行符以外的任意字符
4   \w         匹配字母或数字或下划线或汉字
5   \s         匹配任意的空白符
6   \d         匹配数字
7   \b         匹配单词的开始或结束
8   ^          匹配字符串的开始
9   $          匹配字符串的结束
10  2.字符转义
11  \. 表示一个普通的.字符。      \* 表示一个普通*字符。
12  3.重复次数
13  *          重复零次或更多次 (0 - n)
14  +          重复一次或更多次(1 - n)
15  ?          重复零次或一次 (0 或 1)
```

```
{n}         重复n次 (n)
{n,}         重复n次或更多次 ( >= n)
{n,m}         重复n到m次 (n - m)

4.字符类
[abcdef]          匹配abcdef这几个字符中的任意一个字符
[0-9]                匹配0-9中的任意一个数字
[a-zA-Z0-9]          匹配a-z，A-Z，0-9的任意一个字符
[.?!]                匹配标点符号（.或?或!）
[abc-]          匹配abc-四个字符中的任意一个字符（注意-只能出现在末尾。如果-在中间则表示区间）
5.分支条件
0\d{2}-\d{8}|0\d{3}-\d{7}这个表达式能匹配两种以连字号分隔的电话号码：
一种是三位区号，8位本地号(如010-12345678)，一种是4位区号，7位本地号(0376-2233445)
6.分组
(\d{1,3}.){3}\d{1,3}是一个简单的IP地址匹配表达式。要理解这个表达式，请按下列顺序分析它：\d{1,3}匹配1到3位的数字，
(\d{1,3}.){3}匹配三位数字加上一个英文句号(这个整体也就是这个分组)重复3次，最后再加上一个一到三位的数字(\d{1,3})
7.反义
\W                匹配任意不是字母，数字，下划线，汉字的字符
\S                匹配任意不是空白符的字符
\D                匹配任意非数字的字符
\B                匹配不是单词开头或结束的位置
[^x]                匹配除了x以外的任意字符
[^aeiou]          匹配除了aeiou这几个字母以外的任意字符
```

代码块

```java
package com.powernode.string04;

import java.rmi.RMISecurityManager;
import java.util.Arrays;
import java.util.regex.Pattern;

public class Test05 {
    public static void main(String[] args) {
        String s = "aaaaaa";
        //1.遇到了参数名：regex，让传递是正则表达式
        System.out.println("s.matches(\"[a]\") = " + s.matches("[a]*"));
        /**
         * 1.[a]:匹配a
         * 2.[a]* 匹配0个或者多个a
         */
```

```
        //2.正则表达式一般工作中用于数据合法性校验，比如电话号码是否合法
        String phone = "18610241888";
        String regex = "^1[3-9]\\d{9}";
        /**
         * 1.^1:第一位，字符串开头必须是1
         * 2.[3-9]:第二位，必须是3-9中的任意一个数字
         * 3.\d:匹配数字
         * 4.{9}：重复9次
         */
        boolean matches = Pattern.matches(regex, phone);
        System.out.println(matches);

        String s1 = "a,b:c|d";
        //按照标点符号进行分割，返回一个字符串数组
        String[] split = s1.split("[,:|]");
        System.out.println(Arrays.toString(split));

        String s2 = "Hello,123 word 456";
        String s3 = s2.replaceAll("\\d", "");
        System.out.println(s3);

    }
}
```

4. StringBuffer和StringBuilder

4.1 String不可变 与StringBuffer和StringBuilder 的可变

代码块

```
package com.powernode.bufferandbuilder05;

public class Test01 {
    public static void main(String[] args) {
        String s1 = new String("abc");
        String s2 = s1 + "def";
        System.out.println(s1 == s2);//false

        StringBuffer sb1 = new StringBuffer("abc");
        StringBuffer sb2 = sb1.append("def");
        System.out.println(sb1 == sb2);

        StringBuilder sb3 = new StringBuilder("abc");
        StringBuilder sb4 = sb3.append("def");
        System.out.println(sb3 == sb4);//true
```

```
16
17
18
19        }
20    }
```

4.2 String,StringBuffer和StringBuilder 的区别

1. String的拼接会开辟新的空间，String不可变字符串
2. StringBuffer和StringBuilder 都是可变字符串，每次对字符串的拼接都不会开辟新的空间
3. StringBuffer是线程安全的，效率较低
4. StringBuilder 不是线程安全的，效率较高

```
@Override
@IntrinsicCandidate
public StringBuilder append(String str) {
    super.append(str);
    return this;
}
```

```
@Override
@IntrinsicCandidate
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}
```

4.3 StringBuffer的常用方法

代码块

```
1   public final class StringBuilder
2       extends AbstractStringBuilder
3       implements Appendable, java.io.Serializable, Comparable<StringBuilder>,
    CharSequence
4
5
6   public final class StringBuffer
7      extends AbstractStringBuilder
8      implements Appendable, Serializable, Comparable<StringBuffer>, CharSequence
```

代码块

```java
package com.powernode.bufferandbuilder05;

public class Test02 {
    public static void main(String[] args) {
        /**
         * new StringBuffer()   定义一个空的字符串缓冲区，含有16个字符的容量
         * new StringBuffer(5); 定义一个含有5个字符容量的字符串缓冲区
         * new StringBuffer("天方地圆");     定义一个含有(16+4)的字符串缓冲区，"天方地圆"为4个字符
         * int capacity()   capacity()方法返回字符串的容量大小
         * void trimToSize()    该方法的作用是将StringBuffer对象的中存储空间缩小到和字符串长度一样的长度，减少空间的浪费。
         * StringBuffer append(String str)  向 StringBuffer 对象追加 str 字符串到末尾处
         * StringBuffer insert(int offset, String str)  在指定位置把字符串数据插入到字符串缓冲区里面，并返回字符串缓冲区本身
         */
        StringBuffer sb = new StringBuffer();//定义一个空的字符串缓冲区，含有16个字符的容量
        System.out.println("sb.capacity() = " + sb.capacity());//16
        StringBuffer sb1 = new StringBuffer(5);
        System.out.println("sb1.capacity() = " + sb1.capacity());//5
        StringBuffer sb2 = new StringBuffer("中国你好");
        System.out.println("sb2.capacity() = " + sb2.capacity());//20
        sb2.trimToSize();
        System.out.println("sb2.capacity() = " + sb2.capacity());//4
        StringBuffer sb3 = new StringBuffer("伟大祖国");
        sb3.append(",未来更好");
        System.out.println(sb3);//伟大祖国,未来更好
        System.out.println("sb3.insert(2,6) = " + sb3.insert(2, 6));//伟大6祖国,未来更好
        /**
         * void setCharAt(int index, char ch)   方法用于在字符串的指定索引位置替换一个字符
         * StringBuffer replace(int start, int end, String str) 把[start,end)区间的字符串替换为str
         * StringBuffer reverse()   对字符串进行反转
         * StringBuffer deleteCharAt(int index) 移除序列中指定位置的字符
         * StringBuffer delete(int start, int end)  start 表示要删除字符的起始索引值（包括索引值所对应的字符），end 表示要删除字符串的结束索引值（不包括索引值所对应的字符）
         * String substring(int start)  截取字符串从第[start 位开始到最后
         * String substring(int start, int end) 截取字符串从第[start,end)结束
         */
        System.out.println(sb3);//伟大6祖国,未来更好
        sb3.setCharAt(2,'的');
        System.out.println(sb3);//伟大的祖国,未来更好
        sb3.replace(0, 3, "*");//*祖国,未来更好
```

```
        System.out.println(sb3);
        System.out.println("sb3.reverse() = " + sb3.reverse());//好更来未,国祖*
        System.out.println(sb3.deleteCharAt(0));//更来未,国祖*
        System.out.println("sb3.delete(0,3) = " + sb3.delete(0, 3));//,国祖*

        StringBuffer sb4 = new StringBuffer("伟大的祖国,未来更好");
        System.out.println("sb4.substring(1) = " + sb4.substring(1));//"大的祖国,未来更好"
        System.out.println(sb4);//伟大的祖国,未来更好
        System.out.println("sb4.substring(3,5) = " + sb4.substring(3, 5));//祖国


    }
}
```

4.4 测试String，StringBuffer和StringBuilder 的效率

代码块

```
package com.powernode.bufferandbuilder05;

public class Test03 {
    public static void main(String[] args) {
        int count = 150000;
        testString(count);
        testStringBuffer(count);
        testStringBuilder(count);
        /**
         * String耗时：6209毫秒
         * StringBuffer耗时：5毫秒
         * StringBuilder耗时：4毫秒
         */
    }

    private static void testString(int count) {
        String s = new String();
        //1.开始计时
        long start = System.currentTimeMillis();//获得时间的毫秒数
        //2.字符串处理
        for (int i = 0; i < count; i++) {
            s += i;
        }
        //3.结束计时
        long end = System.currentTimeMillis();//获得时间的毫秒数
        //4.耗时：结束时间 - 开始时间
        System.out.println("String耗时：" + (end - start) + "毫秒");
```

```
    }
    //线程安全，效率较低
    private static void testStringBuffer(int count) {
        StringBuffer sb = new StringBuffer();
        //1.开始计时
        long start = System.currentTimeMillis();//获得时间的毫秒数
        //2.字符串处理
        for (int i = 0; i < count; i++) {
            sb.append(i);
        }
        //3.结束计时
        long end = System.currentTimeMillis();//获得时间的毫秒数
        //4.耗时：结束时间 - 开始时间
        System.out.println("StringBuffer耗时: " + (end - start) + "毫秒");
    }
    //非线程安全，效率较高
    private static void testStringBuilder(int count) {
        StringBuilder sb = new StringBuilder();
        //1.开始计时
        long start = System.currentTimeMillis();//获得时间的毫秒数
        //2.字符串处理
        for (int i = 0; i < count; i++) {
            sb.append(i);
        }
        //3.结束计时
        long end = System.currentTimeMillis();//获得时间的毫秒数
        //4.耗时：结束时间 - 开始时间
        System.out.println("StringBuilder耗时: " + (end - start) + "毫秒");
    }
}
```

4.5 String，StringBuffer和StringBuilder 的效率 代码优化

代码块

```
package com.powernode.bufferandbuilder05;
//定义一个任务接口
interface Task{
    //测量耗时的方法
    static long measureTask(Task task){
        //1.开始计时
        long start = System.currentTimeMillis();//获得时间的毫秒数
        //2.字符串处理
        task.performTask();
        //3.结束计时
        long end = System.currentTimeMillis();//获得时间的毫秒数
```

```
12          //4.耗时：结束时间 - 开始时间
13          return  end - start;
14      }
15
16      //执行具体任务的方法
17      void performTask();
18  }
19
20  public class Test04 {
21      public static void main(String[] args) {
22          int count = 150000;
23          testString(count);
24          testStringBuffer(count);
25          testStringBuilder(count);
26          /**
27           * String耗时：6209毫秒
28           * StringBuffer耗时：5毫秒
29           * StringBuilder耗时：4毫秒
30           */
31      }
32
33      private static void testString(int count) {
34          long time = Task.measureTask(new Task() {
35              String s = new String();
36              @Override
37              public void performTask() {
38                  for (int i = 0; i < count; i++) {
39                      s += i;
40                  }
41              }
42          });
43          System.out.println("String耗时：" + time + "毫秒");
44      }
45      //线程安全，效率较低
46      private static void testStringBuffer(int count) {
47          long time = Task.measureTask(new Task() {
48              StringBuffer sb = new StringBuffer();
49              @Override
50              public void performTask() {
51                  for (int i = 0; i < count; i++) {
52                      sb.append(i);
53                  }
54              }
55          });
56          System.out.println("StringBuffer耗时：" + time + "毫秒");
57      }
58      //非线程安全，效率较高
```

```
59      private static void testStringBuilder(int count) {
60          long time = Task.measureTask(new Task() {
61              StringBuilder sb = new StringBuilder();
62              @Override
63              public void performTask() {
64                  for (int i = 0; i < count; i++) {
65                      sb.append(i);
66                  }
67              }
68          });
69          System.out.println("StringBuilder耗时: " + time + "毫秒");
70      }
71  }
```

在这段代码中，策略模式（Strategy Pattern） 的体现非常典型。策略模式的核心思想是：定义一系列算法（或行为），将它们封装起来，并且使它们可以相互替换。这样可以让算法独立于使用它的客户端而变化。

我们结合代码来详细分析策略模式的各个角色和实现：

1. 策略模式的核心角色

策略模式包含三个核心角色，在这段代码中都有明确体现：

角色	代码中的实现	作用描述
抽象策略（Strategy）	Task 接口	定义所有具体策略的公共接口（performTask()
具体策略（ConcreteStrategy）	三个匿名内部类（new Task() { ... } ）	分别实现 Task 接口，封装了三种不同的字符串拼 StringBuilder）。
环境类（Context）	Task.measureTask() 方法	持有策略对象的引用，负责调用具体策略的算法

2. 代码中策略模式的具体体现

（1）抽象策略（`Task` 接口）

代码块

```
1   interface Task {
2       // 执行具体任务的方法（策略的核心接口）
3       void performTask();
4
5       // 环境类的方法（负责调用策略）
6       static long measureTask(Task task) {
7           long start = System.currentTimeMillis();
```

```
        task.performTask(); // 调用具体策略的算法
        long end = System.currentTimeMillis();
        return end - start;
    }
}
```

- `Task` 接口定义了所有策略必须实现的方法 `performTask()`，这是策略模式的 "标准接口"。
- 静态方法 `measureTask()` 则是环境类的核心，它接收一个策略对象，统一计算任务耗时，屏蔽了不同策略的具体实现细节。

（2）具体策略（三种字符串拼接实现）

代码中通过匿名内部类创建了三个 `Task` 接口的实现类，分别对应三种字符串拼接策略

代码块

```
// 具体策略1：使用String拼接
new Task() {
    String s = new String();
    @Override
    public void performTask() {
        for (int i = 0; i < count; i++) {
            s += i; // String拼接算法
        }
    }
}

// 具体策略2：使用StringBuffer拼接
new Task() {
    StringBuffer sb = new StringBuffer();
    @Override
    public void performTask() {
        for (int i = 0; i < count; i++) {
            sb.append(i); // StringBuffer拼接算法
        }
    }
}

// 具体策略3：使用StringBuilder拼接
new Task() {
    StringBuilder sb = new StringBuilder();
    @Override
    public void performTask() {
```

```
28            for (int i = 0; i < count; i++) {
29                sb.append(i); // StringBuilder拼接算法
30            }
31        }
32    }
```

- 每个具体策略都实现了 `performTask()` 方法，但内部使用了不同的字符串拼接算法（`+` 运算符、`StringBuffer.append()`、`StringBuilder.append()`）。
- 这些策略之间可以相互替换，因为它们都遵循 `Task` 接口的规范。

（3）环境类（`measureTask()` 方法）

环境类（`measureTask()`）是策略模式的 "调用者"，它的作用是：

1. 接收具体策略对象（`Task task` 参数）；
2. 统一调用策略的算法（`task.performTask()`）；
3. 附加公共逻辑（计时功能）。

代码块

```
1   static long measureTask(Task task) {
2       long start = System.currentTimeMillis(); // 公共逻辑：开始计时
3       task.performTask(); // 调用具体策略的算法
4       long end = System.currentTimeMillis();   // 公共逻辑：结束计时
5       return end - start;
6   }
```

- 环境类将 "计时逻辑" 与 "具体拼接算法" 解耦：无论具体策略如何变化（比如新增一种拼接方式），计时逻辑都无需修改。
- 客户端（`main` 方法）只需传入不同的策略对象，即可得到不同算法的耗时，无需关心计时的实现细节。

4. 策略模式的优势（结合代码分析）

这段代码通过策略模式体现了以下优势：

1. 算法的自由切换：
   客户端可以在 `testString()`、`testStringBuffer()`、`testStringBuilder()` 中自由选择不同的拼接策略，且切换时无需修改其他代码。
2. 避免大量条件判断：
   如果不使用策略模式，可能需要用 `if-else` 判断选择哪种拼接方式（如 `if (type == "String") { ... } else if (...)`），而策略模式通过多态直接消除了这种判断。
3. 符合开闭原则：
   如果需要新增一种字符串拼接策略（比如 `StringJoiner`），只需新增一个 `Task` 接口的实现类，无需修改 `measureTask()` 或其他现有代码。
4. 算法的复用与隔离：
   每种拼接算法被封装在独立的策略类中，便于复用和单独维护（比如优化某一种算法的性能）。

总结

这段代码通过 `Task` 接口定义策略规范，通过三个匿名内部类实现具体策略（三种拼接方式），通过 `measureTask()` 方法作为环境类统一调用策略，完美诠释了策略模式的设计思想。其核心目的是将 "做什么"（计时）与 "怎么做"（具体拼接算法）分离，使得算法可以灵活替换和扩展。

5. 日期类

5.1 JDK8之前

- java.util.Date :表示一个特定的时间瞬间，精确到毫秒
- DateFormat： 日期格式的抽象类
- SimpleDateFormat： 日期格式的抽象类的子类，它可以指定格式化格式
- Calendar：日历类，不保存特定时间的瞬间，但是提供日期相关计算的功能

字母	含义	示例
y	年，一个y代表一位	yy代表24，"yyyy"代表2024
M	月份	例如八月，M代表8，MM代表08
w	一年中的第几周	常用ww表示
W	一个月中的第几周	常用WW表示
d	一个月中的第几天	常用dd表示
D	一年中的第几天	常用DDD表示
E	星期几，用E表示星期	CHINA表示星期几，US表示英文缩写
a	上午或下午	am代表上午，pm代表下午
H	一天中的小时数，二十四小时制	常用HH表示
h	一天中的小时数，十二小时制	常用hh表示
m	分钟数	常用mm表示
s	秒数	常用ss表示
S	毫秒数	常用SSS表示

代码块

```
package com.powernode.date06;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class Test01 {
    public static void main(String[] args) {
        /**
         * 日期类：年月日,时分秒，星期，上午下午
         */
        //1.创建日期对象
        Date date = new Date();
        System.out.println(date);//Sat Aug 09 14:07:26 CST 2025
        //2.创建日期格式化对象，并指定格式
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd a HH:mm:ss SSS E");
        //3.通过指定的格式对当前日期对象格式化
        System.out.println(simpleDateFormat.format(date));//2025-08-09 下午 14:10:53 428 周六
        //4.今天是一年中的第几天
        simpleDateFormat = new SimpleDateFormat("DDD");
        System.out.println(simpleDateFormat.format(date));
        System.out.println("-----------");
        //1.获得日历对象
```

```
        Calendar instance = Calendar.getInstance();
        //2.通过日历对象获得当前时间
        Date time = instance.getTime();
        System.out.println(time);
        //3.创建日期格式化对象，并指定格式
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd a HH:mm:ss SSS E");
        //4.通过指定的格式对当前日期对象格式化
        System.out.println(dateFormat.format(date));
        //5.今天是一年当中的第几天
        System.out.println(instance.get(Calendar.DAY_OF_YEAR));
    }
}
```

5.2 JDK8及之后

代码块

```
package com.powernode.date06;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Calendar;
import java.util.Date;

public class Test02 {
    public static void main(String[] args) {
        //1.获得当前的时间
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);
        //2.创建日期格式化对象，并指定格式
        DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd a HH:mm:ss SSS E");
        //3.对当前日期进行格式
        System.out.println(dateTimeFormatter.format(now));
        //4.今天是一年当中的第几天
        System.out.println(now.getDayOfYear());
    }
}
```

6. 获得系统属性

代码块

```
package com.powernode.system07;

public class Test {
    public static void main(String[] args) {
        System.out.println("获得当前工程的路径: " +
System.getProperty("user.dir"));
        System.out.println("Java 运行时环境版本: " +
System.getProperty("java.version"));
        System.out.println("Java运行时环境供应商: " +
System.getProperty("java.vendor"));
        System.out.println("Java 安装目录: " + System.getProperty("java.home"));
        System.out.println("Java 虚拟机规范版本: " +
System.getProperty("java.vm.specification.version"));
        System.out.println("Java 虚拟机规范供应商: " +
System.getProperty("java.vm.specification.vendor"));
        System.out.println("Java 虚拟机规范名称: " +
System.getProperty("java.vm.specification.name"));
        System.out.println("Java 虚拟机实现版本: " +
System.getProperty("java.vm.version"));
        System.out.println("Java 虚拟机实现供应商: " +
System.getProperty("java.vm.vendor"));
        System.out.println("Java 运行时环境规范版本: " +
System.getProperty("java.specification.version"));
        System.out.println("Java 运行时环境规范供应商: " +
System.getProperty("java.specification.vendor"));
        System.out.println("Java 运行时环境规范名称: " +
System.getProperty("java.specification.name"));
        System.out.println("Java 类路径: " +
System.getProperty("java.class.path"));
        System.out.println("操作系统的名称: " + System.getProperty("os.name"));
        System.out.println("操作系统的架构: " + System.getProperty("os.arch"));
        System.out.println("操作系统的版本: " +
System.getProperty("os.version"));

    }
}
```

7. 数学相关工具类

7.1 Math工具类

代码块

```
package com.powernode.math08;

```

```java
public class Test01 {
    public static void main(String[] args) {
        /**
         * public static double ceil(double a)  返回大于参数的最小整数（向上取整）
         * public static double floor(double a) 返回小于参数的最大整数（向下取整）
         * public static long round(double a)   返回四舍五入的整数
         * public static int max(int a, int b)  返回两个数的最大值
         * public static int min(int a, int b)  返回两个数的最小值
         * public static int abs(int a) 获取a的绝对值
         * public static double pow(double a, double b) 获得a的b次方
         * public static double sqrt(double a)  对a执行开平方的操作
         * public static double random()    获得[0.0, 1.0)之间的随机小数
         */
        System.out.println(Math.ceil(2.1));//3.0,舍小数，进1（向上取整）
        System.out.println(Math.floor(2.9));//2.0,舍小数（向下取整）
        System.out.println(Math.round(2.5));//3
        System.out.println(Math.max(2, 3));
        System.out.println(Math.min(2, 3));
        System.out.println(Math.abs(-3));//3
        System.out.println(Math.pow(2, 3));//8
        System.out.println(Math.sqrt(9));//3
        System.out.println(Math.random());
    }
}
```

7.2 随机数

代码块

```java
package com.powernode.math08;

import java.util.Random;

public class Test02 {
    public static void main(String[] args) {
        //1.随机数获得方式一
        double random = Math.random();//[0.0-1.0)
        System.out.println(random);
        //2.获得[0,10]之间的随机数
        System.out.println(Math.round(Math.random() * 10));
        System.out.println("-----------");
        //3.获得随机数方式二
        Random random1 = new Random();
        //4.获得[0,10)之间的随机数
        System.out.println(random1.nextInt(10));

```

```
    }
}
```

7.3 BigInteger

代码块

```
package com.powernode.Integeranddecaimal09;

import java.math.BigInteger;

public class Test01 {
    public static void main(String[] args) {
        //long l = 12345678901234567890L;//Long number too large
        BigInteger bigInteger = new BigInteger("12345678901234567890");
        //加
        System.out.println(bigInteger.add(new BigInteger("1")));
        System.out.println("bigInteger = " + bigInteger);
        //减
        System.out.println(bigInteger.subtract(new BigInteger("1")));
        //乘
        System.out.println(bigInteger.multiply(new BigInteger("10")));
        //除
        System.out.println(bigInteger.divide(new BigInteger("10")));
        //取余
        BigInteger bigInteger1 = new BigInteger("10");
        BigInteger[] bigIntegers = bigInteger1.divideAndRemainder(new
BigInteger("3"));
        System.out.println(bigIntegers[0]);//商
        System.out.println(bigIntegers[1]);//余数
    }
}
```

7.4 BigDecaimal

代码块

```
package com.powernode.Integeranddecaimal09;

import java.math.BigDecimal;

public class Test02 {
    public static void main(String[] args) {
        //小数不够用了使用BigDecimal
        BigDecimal bigDecimal = new BigDecimal("123.123");
```

```
9           System.out.println(bigDecimal.add(new BigDecimal("1.1")));
10
11
12      }
13  }
```

8. UUID 主要用途

1. 数据库主键
   在分布式数据库或分库分表场景中，传统自增 ID 可能因不同节点独立生成而重复，UUID 可作为全局唯一主键，确保跨节点数据的唯一性。
2. 分布式系统标识
   在微服务、分布式缓存、消息队列等场景中，用于标识跨服务的资源（如订单 ID、会话 ID、任务 ID），避免不同服务生成的标识冲突。
3. 临时文件 / 资源命名
   生成临时文件名、缓存键、日志文件标识等，确保同一系统或不同系统中创建的资源不会因重名被覆盖。
4. 安全验证与令牌
   用于生成会话令牌（Session ID）、验证码、重置密码链接等，利用其唯一性和不可预测性提升安全性。
5. 版本控制与实体追踪
   在协同编辑、版本管理中标识不同的修改版本或实体实例，确保每个版本 / 实例可被唯一识别。

代码块

```
1   package com.powernode.uuid;
2
3   import java.util.UUID;
4
5   public class Test {
6       public static void main(String[] args) {
7           UUID uuid = UUID.randomUUID();
8           System.out.println("uuid = " + uuid);
9       }
10  }
```

作业

1. StringBuilder 的常用方法调用一遍 （P63）

```
new StringBuilder()          定义一个空的字符串缓冲区，含有16个字符的容量
new StringBuilder(5);          定义一个含有5个字符容量的字符串缓冲区
new StringBuilder("天方地圆");          定义一个含有(16+4)的字符串缓冲区，"天方地圆"为4个字符
int capacity()          capacity()方法返回字符串的容量大小
void trimToSize()          该方法的作用是将StringBuilder对象的中存储空间缩小到和字符串长度一样的长度，减少空间的浪费。
StringBuilder  append(String str)          向 StringBuffer 对象追加 str 字符串到末尾处
StringBuilder  insert(int offset, int i)          在指定位置把任意类型的数据插入到字符串缓冲区里面，并返回字符串缓冲区本身
void setCharAt(int index, char ch)          方法用于在字符串的指定索引位置替换一个字符
StringBuilder  replace(int start, int end, String str)          把[start,end)区间的字符串替换为str
StringBuilder  reverse()          对字符串进行反转
StringBuilder  deleteCharAt(int index)          移除序列中指定位置的字符
StringBuilder  delete(int start, int end)          start 表示要删除字符的起始索引值（包括索引值所对应的字符），end 表示要删除字符串的结束索引值（不包括索引值所对应的字符）
String substring(int start)          截取字符串从第[start 位开始到最后
String substring(int start, int end)          截取字符串从第[start,end)结束
```

2. 练习题
   a. 在main方法中创建一个字符串对象” abcdjklnmuxwxyz ”，打印字符串长度；
   b. 删除字符串开始和结尾处的空白，以获得新字符串，并打印输出新串的长度；
   c. 判断新字符串是否以“abc”开头，是否以“xyz”结尾，打印判断结果；
   d. 去空格后截取第3位至第6位间的子串，将其转换为大写并打印；
   e. 查找该新串是否包含“lnm”子串，并打印子串在字符串中的位置。
3. 练习题
   a. 编写程序TestStringBuilder类；以”dljd”字符串为基础创建StringBuilder对象；
   b. 将”2009”字符串拼接到StringBuilder结尾；
   c. 将“Java”字符串插入到StringBuilder第0个位置；
   d. 打印该StringBuilder。
4. 上课讲的代码敲一遍