1.为啥学习Java？

1.1java的优点

- 跨平台（一次编译，处处执行）
- 开源项目（找到解决方式）

2.JDK8的安装配置

2.1 JDK1.5以上版本安装配置

- 新建JAVA_HOME变量：D:\jdk1.8
- 在Path变量中新建:.;%JAVA_HOME%\bin;

  (在地址栏，按住home,在地址头输入上述内容)

  以上是jdk1.5以上

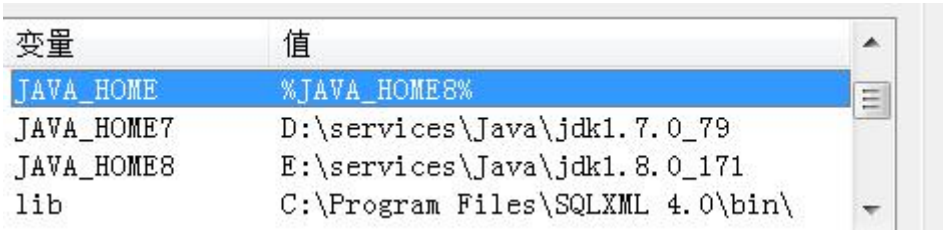
2.2 JDK1.5以下版本安装配置

- 新建JAVA_HOME变量：D:\jdk1.8,也就是jdk的安装目录；
- x新建classpath变量(类加载路径): .;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar
- Path变量中添加%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

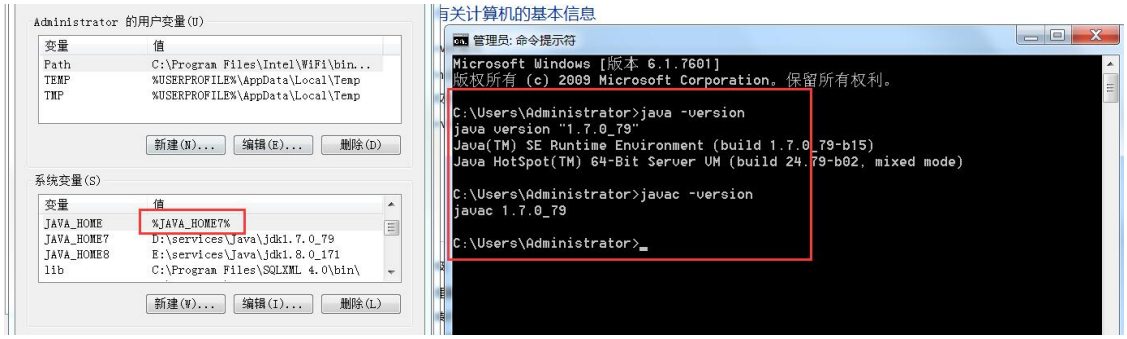
3.win7安装多版本

安装教程：

1. 使用三个JAVA_HOME变量



2. 只需要更改JAVA_HOME中的变量7或者8就行了



3. path变量.;%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;要放到最前面

4.java各版本特点

JDK Version 1.0

1996-01-23 Oak(橡树)

> 初代版本，伟大的一个里程碑，但是是纯解释运行，使用外挂JIT，性能比较差，运行速度慢。

JDK Version 1.1

1997-02-19

> - JDBC(Java DataBase Connectivity);
> - 支持内部类;
> - RMI(Remote Method Invocation) ;
> - 反射;
> - Java Bean;

JDK Version 1.2

1998-12-08 Playground(操场)

> 集合框架; JIT(Just In Time)编译器; 对打包的Java文件进行数字签名; JFC(Java Foundation Classes), 包括Swing 1.0, 拖放和Java2D类库; Java插件; JDBC中引入可滚动结果集,BLOB,CLOB,批量更新和用户自定义类型; Applet中添加声音支持.

JDK Version 1.3

2000-05-08 Kestrel(红隼)

> - Java Sound API;
> - jar文件索引;
> - 对Java的各个方面都做了大量优化和增强;

JDK Version 1.4

2002-02-13 Merlin(隼)

> - XML处理;
> - Java打印服务;
> - Logging API;
> - Java Web Start;
> - JDBC 3.0 API;
> - 断言;
> - Preferences API;
> - 链式异常处理;
> - 支持IPV6;
> - 支持正则表达式;
> - 引入Imgae I/O API.

JAVA 5

2004-09-30 Tiger(老虎)

> - 泛型;
> - 增强循环,可以使用迭代方式;
> - 自动装箱与自动拆箱;
> - 类型安全的枚举;

- 可变参数;
- 静态引入;
- 元数据(注解);
- Instrumentation;

JAVA 6

2006-12-11 Mustang(野马)

> 支持脚本语言; JDBC 4.0API; Java Compiler API; 可插拔注解; 增加对Native PKI(Public Key Infrastructure), Java GSS(Generic Security Service),Kerberos和LDAP(Lightweight Directory Access Protocol)支持; 继承Web Services;

JAVA 7

2011-07-28 Dolphin(海豚)

> switch语句块中允许以字符串作为分支条件; 在创建泛型对象时应用类型推断; 在一个语句块中捕获多种异常; 支持动态语言; 支持try-with-resources(在一个语句块中捕获多种异常); 引入Java NIO.2开发包; 数值类型可以用二进制字符串表示,并且可以在字符串表示中添加下划线; 钻石型语法(在创建泛型对象时应用类型推断); null值得自动处理;

JAVA 8

2014-03-18

> Lambda 表达式 − Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中。
>
> 方法引用 − 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。
>
> 默认方法 − 默认方法就是一个在接口里面有了一个实现的方法。
>
> 新工具 − 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。
>
> Stream API −新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。
>
> Date Time API − 加强对日期与时间的处理。
>
> Optional 类 − Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。
>
> Nashorn, JavaScript 引擎 − Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。
>
> 详细参考:http://www.runoob.com/java/java8-new-features.html

JAVA 9

2017-09-22

> 模块系统：模块是一个包的容器，Java 9 最大的变化之一是引入了模块系统（Jigsaw 项目）。 REPL (JShell)：交互式编程环境。 HTTP 2 客户端：HTTP/2标准是HTTP协议的最新版本，新的 HTTPClient API 支持 WebSocket 和 HTTP2 流以及服务器推送特性。 改进的 Javadoc：Javadoc 现在支持在 API 文档中的进行搜索。另外，Javadoc 的输出现在符合兼容 HTML5 标准。 多版本兼容 JAR 包：多版本兼容 JAR 功能能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本。 集合工厂方法：List，Set 和 Map 接口中，新的静态工厂方法可以创建这些集合的不可变实例。 私有接口方法：在接口中使用private私有方法。我们可以使用 private 访问修饰符在接口中编写私有方法。 进程 API: 改进的 API 来控制和管理操作系统进程。引进 java.lang.ProcessHandle 及其嵌套接口 Info 来让开发者逃离时常因为要获取一个本地进程的

> PID 而不得不使用本地代码的窘境。 改进的 Stream API：改进的 Stream API 添加了一些便利的方法，使流处理更容易，并使用收集器编写复杂的查询。 改进 try-with-resources：如果你已经有一个资源是 final 或等效于 final 变量,您可以在 try-with-resources 语句中使用该变量，而无需在 try-with-resources 语句中声明一个新变量。 改进的弃用注解 @Deprecated：注解 @Deprecated 可以标记 Java API 状态，可以表示被标记的 API 将会被移除，或者已经破坏。 改进钻石操作符(Diamond Operator) ：匿名类可以使用钻石操作符(Diamond Operator)。 改进 Optional 类：java.util.Optional 添加了很多新的有用方法，Optional 可以直接转为 stream。 多分辨率图像 API：定义多分辨率图像API，开发者可以很容易的操作和展示不同分辨率的图像了。 改进的 CompletableFuture API ： CompletableFuture 类的异步机制可以在 ProcessHandle.onExit 方法退出时执行操作。 轻量级的 JSON API：内置了一个轻量级的JSON API 响应式流（ Reactive Streams) API: Java 9中引入了新的响应式流 API 来支持 Java 9 中的响应式编程。 详细参考:http://www.runoob.com/java/java9-new-features.html

JAVA 10

2018-03-21

根据官网的公开资料，共有12个重要特性，如下：

> JEP286，var 局部变量类型推断。 JEP296，将原来用 Mercurial 管理的众多 JDK 仓库代码，合并到一个仓库中，简化开发和管理过程。 JEP304，统一的垃圾回收接口。 JEP307，G1 垃圾回收器的并行完整垃圾回收，实现并行性来改善最坏情况下的延迟。 JEP310，应用程序类数据 (AppCDS) 共享，通过跨进程共享通用类元数据来减少内存占用空间，和减少启动时间。 JEP312，ThreadLocal 握手交互。在不进入到全局 JVM 安全点 (Safepoint) 的情况下，对线程执行回调。优化可以只停止单个线程，而不是停全部线程或一个都不停。 JEP313，移除 JDK 中附带的 javah 工具。可以使用 javac -h 代替。 JEP314，使用附加的 Unicode 语言标记扩展。 JEP317，能将堆内存占用分配给用户指定的备用内存设备。 JEP317，使用 Graal 基于 Java 的编译器，可以预先把 Java 代码编译成本地代码来提升效能。 JEP318，在 OpenJDK 中提供一组默认的根证书颁发机构证书。开源目前 Oracle 提供的的 Java SE 的根证书，这样 OpenJDK 对开发人员使用起来更方便。 JEP322，基于时间定义的发布版本，即上述提到的发布周期。版本号为 $FEATURE.$INTERIM.$UPDATE.$PATCH，分别是大版本，中间版本，升级包和补丁版本。

JAVA 11

2018-09-25

翻译后的新特性有：

> 181:Nest-Based访问控制 309:动态类文件常量 315:改善Aarch64 intrinsic 318:无操作垃圾收集器 320:消除Java EE和CORBA模块 321:HTTP客户端(标准) 323:局部变量的语法λ参数 324:Curve25519和Curve448关键协议 327:Unicode 10 328:飞行记录器 329:ChaCha20和Poly1305加密算法 330:发射一列纵队源代码程序 331:低开销堆分析 332:传输层安全性(Transport Layer Security,TLS)1.3 333:动作:一个可伸缩的低延迟垃圾收集器 (实验) 335:反对Nashorn JavaScript引擎 336:反对Pack200工具和API

JAVA 12

2019-03-19

作为“功能性版本”，JDK 12 总共包含 8 个新的 JEP ，分别为：

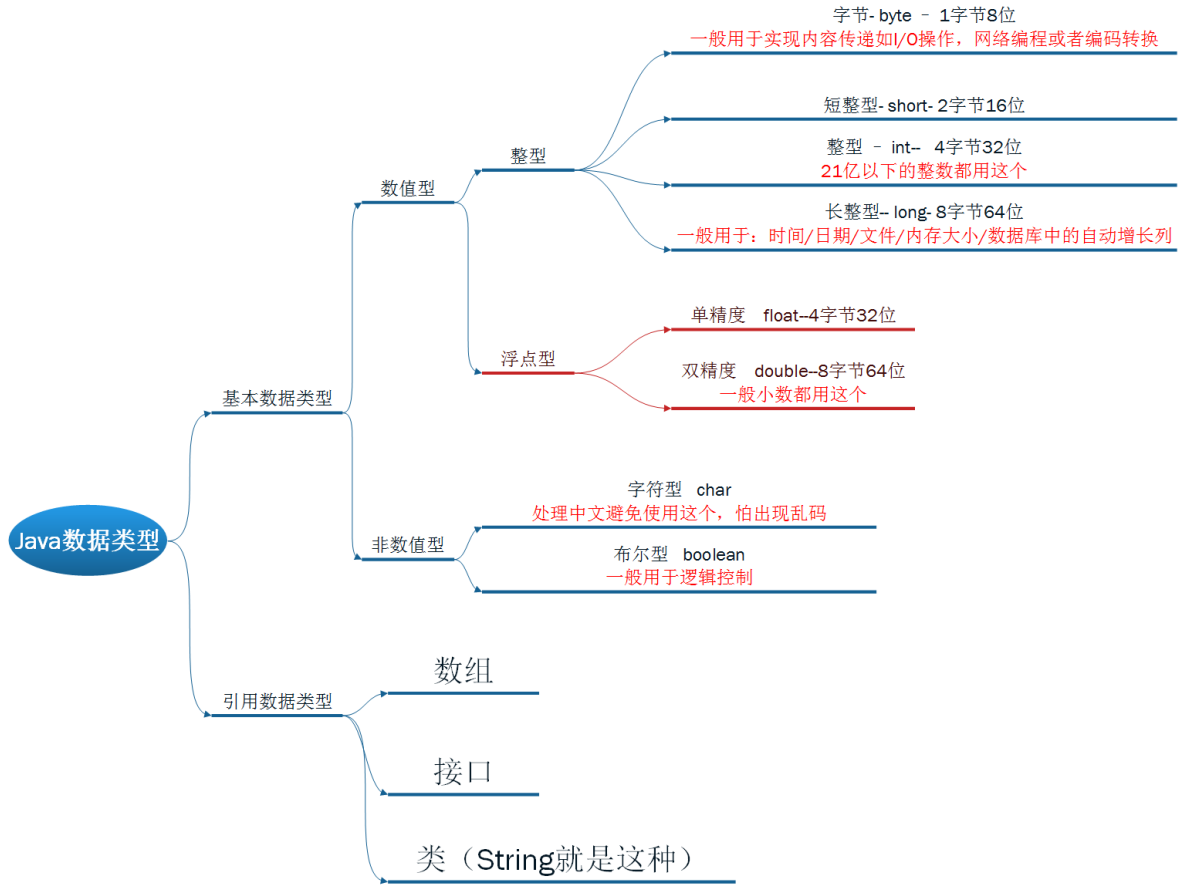
> 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental) ：新增一个名为 Shenandoah 的垃圾回收器，它通过在 Java 线程运行的同时进行疏散 (evacuation) 工作来减少停顿时间。

> 230: Microbenchmark Suite：新增一套微基准测试，使开发者能够基于现有的 Java Microbenchmark Harness（JMH）轻松测试 JDK 的性能，并创建新的基准测试。
>
> 325: Switch Expressions (Preview) ：对 switch 语句进行扩展，使其可以用作语句或表达式，简化日常代码。
>
> 334: JVM Constants API ：引入一个 API 来对关键类文件 (key class-file) 和运行时工件的名义描述（nominal descriptions）进行建模，特别是那些可从常量池加载的常量。
>
> 340: One AArch64 Port, Not Two ：删除与 arm64 端口相关的所有源码，保留 32 位 ARM 移植和 64 位 aarch64 移植。
>
> 341: Default CDS Archives ：默认生成类数据共享（CDS）存档。
>
> 344: Abortable Mixed Collections for G1 ：当 G1 垃圾回收器的回收超过暂停目标，则能中止垃圾回收过程。
>
> 346: Promptly Return Unused Committed Memory from G1 ：改进 G1 垃圾回收器，以便在空闲时自动将 Java 堆内存返回给操作系统。

JAVA 13

> 原生字符串文字（raw string literals），它可以跨多行源码而不对转义字符进行转义。目标是简化 Java 的开发，比如开发者可以可读形式表示字符序列，而不会掺杂一些 Java 指示符，或者提供针对 Java 以外的语法的字符串。在 JDK 13 中进行预期的测试运行后，该功能可以在随后的 JDK 14 版本中跟进。
>
> 可在生产环境中使用的 switch 表达式，JDK 13 中将带来一个 beta 版本实现。switch 表达式扩展了 switch 语句，使其不仅可以作为语句（statement），还可以作为表达式（expression），并且两种写法都可以使用传统的 switch 语法，或者使用简化的“case L ->”模式匹配语法作用于不同范围并控制执行流。这些更改将简化日常编码工作，并为 switch 中的模式匹配（JEP 305）做好准备

5.java的基本数据类型

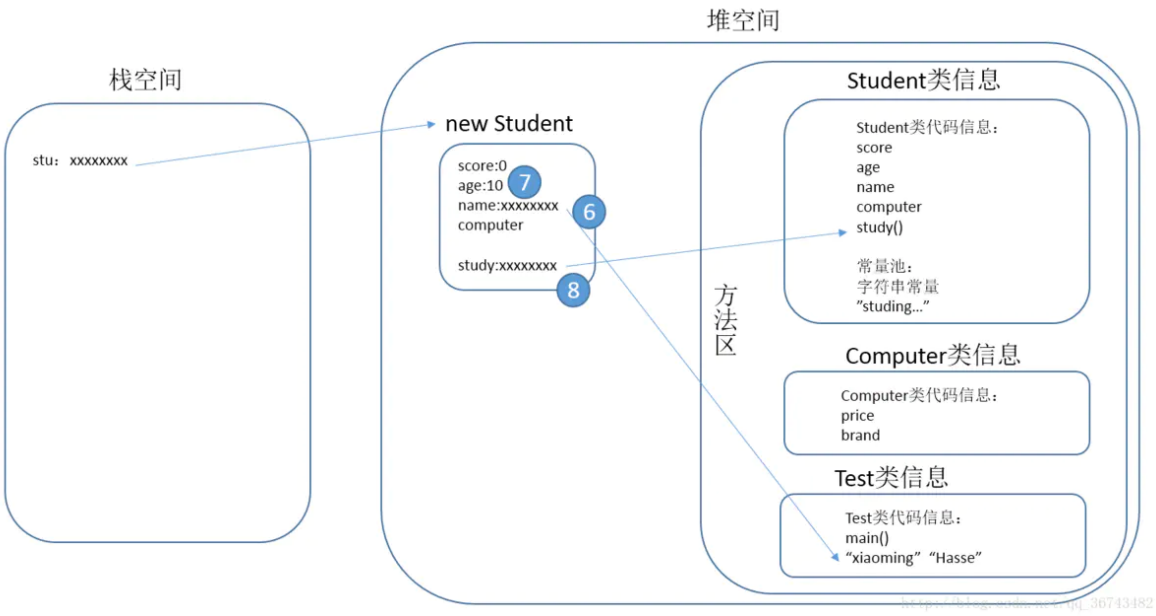


Java数据类型在内存中的存储：

1）基本数据类型的存储原理：所有的简单数据类型不存在“引用”的概念，基本数据类型都是直接存储在内存中的内存栈上的，数据本身的值就是存储在栈空间里面，而Java语言里面八种数据类型是这种存储模型；

2）引用类型的存储原理:引用类型继承于Object类（也是引用类型）都是按照Java里面存储对象的内存模型来进行数据存储的，使用Java内存堆和内存栈来进行这种类型的数据存储，简单地讲，“引用”是存储在有序的内存栈上的，而对象本身的值存储在内存堆上的；

区别:基本数据类型和引用类型的区别主要在于基本数据类型是分配在栈上的，而引用类型是分配在堆上的



5.1 字符类型

字符类型主要用来存储单个字符；

字符型常量有3种表示形式；

1，直接通过单个字符来指定字符型常量，如'A'，'B'，'5'；

2，通过转义字符表示特殊字符型常量，如'\n'，'\'；

3，直接使用Unicode值来表示字符型常量，如'\u66f9'，'\u950b'；

转义字符	说明
\b	退格
\n	换行
\t	制表符
\"	双引号
\'	单引号
\\	反斜杠
\r	回车符

6.编码

6.1 ASCII码

> ASCII 码一共规定了128个字符的编码，比如空格`SPACE`是32（二进制`00100000`），大写的字母`A`是65（二进制`01000001`）。这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的一位统一规定为`0`。

（注解：一个字节为八个二进制，能表示256个状态）

6.2 unicode编码方式

> Unicode 当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，`U+0639`表示阿拉伯字母`Ain`，`U+0041`表示英语的大写字母`A`，`U+4E25`表示汉字`严`。具体的符号对应表，可以查询unicode.org，或者专门的汉字对应表。

6.3 UTF-8码

- UTF-8 是 Unicode 的实现方式之一。
- UTF-8 最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

> UTF-8 的编码规则很简单，只有二条：

> 1）对于单字节的符号，字节的第一位设为`0`，后面7位为这个符号的 Unicode 码。因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。

> 2）对于`n`字节的符号（`n > 1`），第一个字节的前`n`位都设为`1`，第`n + 1`位设为`0`，后面字节的前两位一律设为`10`。剩下的没有提及的二进制位，全部为这个符号的 Unicode 码。

下表总结了编码规则，字母`x`表示可用编码的位。

```
Unicode符号范围      |        UTF-8编码方式
(十六进制)           |                （二进制）
---------------------+---------------------------------------------
0000 0000-0000 007F | 0xxxxxxx
0000 0080-0000 07FF | 110xxxxx 10xxxxxx
0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

`严`的 Unicode 是`4E25`（`100111000100101`），根据上表，可以发现`4E25`处在第三行的范围内（`0000 0800 - 0000 FFFF`），因此`严`的 UTF-8 编码需要三个字节，即格式是`1110xxxx 10xxxxxx 10xxxxxx`。然后，从`严`的最后一个二进制位开始，依次从后向前填入格式中的`x`，多出的位补`0`。这样就得到了，`严`的 UTF-8 编码是`11100100 10111000 10100101`，转换成十六进制就是`E4B8A5`。

7.结构

7.1 选择结构

7.1.1 if-else

```
public class Demo12 {

    public static void main(String[] args) {
        int a=-1;
        // if语句
        // 多行注释快捷方式  ctrl+shift+/
        if(a>0){
            System.out.println(a+"是正数");
        }

        // if...else语句
        if(a>0){
            System.out.println(a+"是正数");
        }else{
            System.out.println(a+"不是正数");
        }

        // if...else if...else
        if(a>0){
            System.out.println(a+"是正数");
        }else if(a<0){
            System.out.println(a+"是负数");
        }else{
            System.out.println(a+"是0");
        }
    }
}
```

```
-1不是正数
-1是负数
```

7.1.2 switch

> 在switch在jdk1.6或者jdk1.6以下版本，仅支持整型类型；jdk1.7开始支持字符串；

我们先看一个int类型的实例：

```
import java.util.Scanner;

public class Demo13 {

    public static void main(String[] args) {
        System.out.println("请输入一个数字：");
        // 定义一个系统输入对象
        // 自动导包  ctrl+shift+o
        Scanner scanner=new Scanner(System.in);
        int n=scanner.nextInt();
        scanner.close();
        switch(n){
            case 1:{
                System.out.println("用户输入的是1");
                break;
            }
            case 2:{
                System.out.println("用户输入的是2");
                break;
            }
            default:{
                System.out.println("用户输入的是其他数字");
            }
        }
    }
}
```

```
请输入一个数字：
2
用户输入的是2
```

我们再来看一个字符串类型的实例：

```
import java.util.Scanner;

public class Demo14 {

    public static void main(String[] args) {
        System.out.println("请输入一个字符串：");
        Scanner scanner=new Scanner(System.in);
        String str=scanner.next();
        scanner.close();
        switch(str){
            case "张三":{
                System.out.println("输入的是张三");
                break;
            }
            case "李四":{
                System.out.println("输入的是李四");
                break;
            }
            default:{
```

```
                System.out.println("用户输入的是其他字符串");
            }
        }
    }
}
```

用switch实现：

```
public class Test {

    public static void main(String[] args) {
        System.out.println("请输入一个数字：");
        // 定义一个系统输入对象
        Scanner scanner=new Scanner(System.in);
        int n=scanner.nextInt();
        scanner.close();
        switch(n){
            case 1:{
                System.out.println("现在是一月份");
                break;
            }
            case 2:{
                System.out.println("现在是二月份");
                break;
            }
            case 3:{
                System.out.println("现在是三月份");
                break;
            }
            case 4:{
                System.out.println("现在是四月份");
                break;
            }
            case 5:{
                System.out.println("现在是五月份");
                break;
            }
            case 6:{
                System.out.println("现在是六月份");
                break;
            }
            case 7:{
                System.out.println("现在是七月份");
                break;
            }
            case 8:{
                System.out.println("现在是八月份");
                break;
            }
            case 9:{
                System.out.println("现在是九月份");
                break;
            }
            case 10:{
                System.out.println("现在是十月份");
                break;
            }
```

```
            case 11:{
                System.out.println("现在是十一月份");
                break;
            }
            case 12:{
                System.out.println("现在是十二月份");
                break;
            }
            default:{
                System.out.println("用户输入有误");
            }
        }
    }
}
```

7.2 循环结构

7.2.1 while循环

> 我们来用while来实现下在控制台输出1到10；

```
int i=1;
while(i<=10){
    System.out.print(i+" ");
    i++;
}
```

7.2.2 do-while循环

```
int j=1;
do{
    System.out.print(j+" ");
    j++;
}while(j<=10);
```

7.2.3 for循环

```
for(int k=1;k<=10;k++){
    System.out.print(k+" ");
}
```

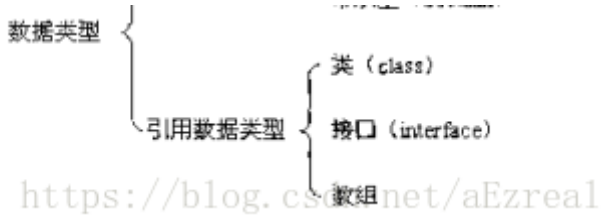
> **for循环嵌套**

```
for(int m=0;m<10;m++){
    for(int n=0;n<10;n++){
        System.out.print("m="+m+"n="+n+" ");
    }
    System.out.println();
}
```

8.数组

> - 数组是一种特殊的数据类型，

- 固定的有序集合.(分别是大小固定和有序.)
- 数组的长度在创建时指定,每个元素都有下标,可以通过下标来访问数组的元素.



1. 栈内存:用于存储局部变量,当数据使用完,所占空间会自动释放,
2. 堆内存:
    1. 数组和对象,通用new建立的实例都存在在堆内存中
        2. 每个实体都有内存地址值.
            3. 实体中的变量都有默认初始化值.
            4. 实体不被使用会在不确定的时间内被垃圾回收器回收.

8.1 数组的初始化

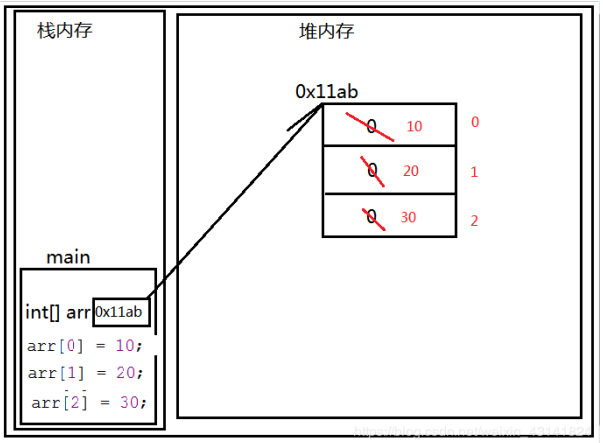
静态初始化

```
public class Demo18 {

    public static void main(String[] args) {
        // 定义一个数组，并且静态初始化
        int arr[]=new int[]{1,2,3};//int arr[]={1,2,3}

        // 普通的遍历数组方式
        for(int i=0;i<arr.length;i++){
            System.out.println(arr[i]);
        }

        System.out.println("----------------");
        // foreach方式
        for(int j:arr){
            System.out.println(j);
        }
    }
}
```

动态初始化

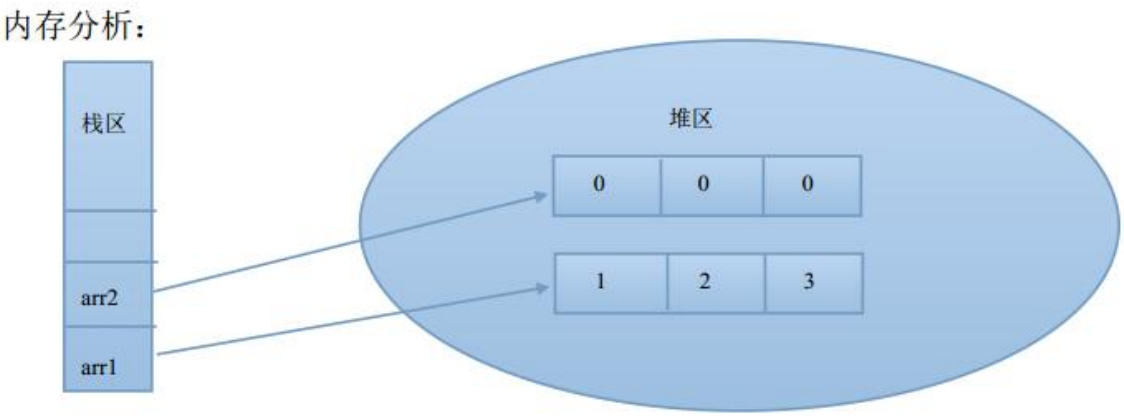
```
public class Demo18 {

    public static void main(String[] args) {
        // 定义一个数组，然后动态初始化，长度是3
        int arr[]=new int[3];
        // int类型的数组，默认是0
        for(int i=0;i<arr.length;i++){
            System.out.println(arr[i]);
        }
    }
}
```

```java
class Demo4_一个数组的内存图解 {
    public static void main(String[] args) {
        int[] arr = new int[3];
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        System.out.println(arr[0]+"--"+arr[1]+"--"+arr[2]);
    }
}
```

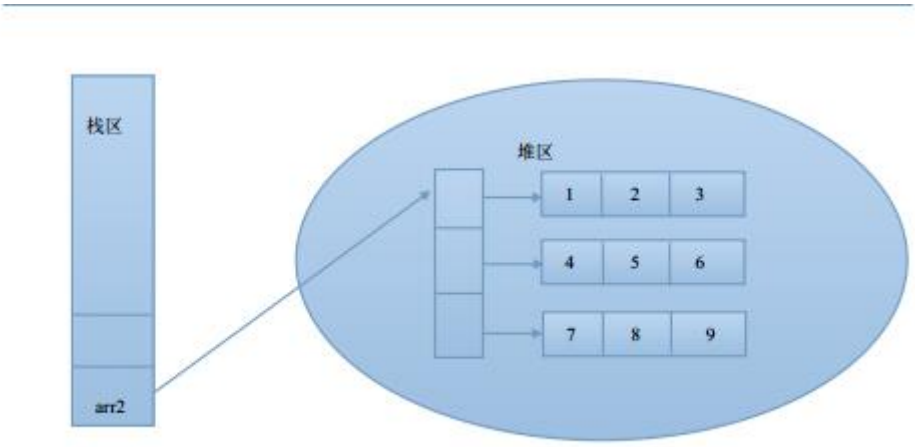


8.2 数组的内存分析

> 一维数组



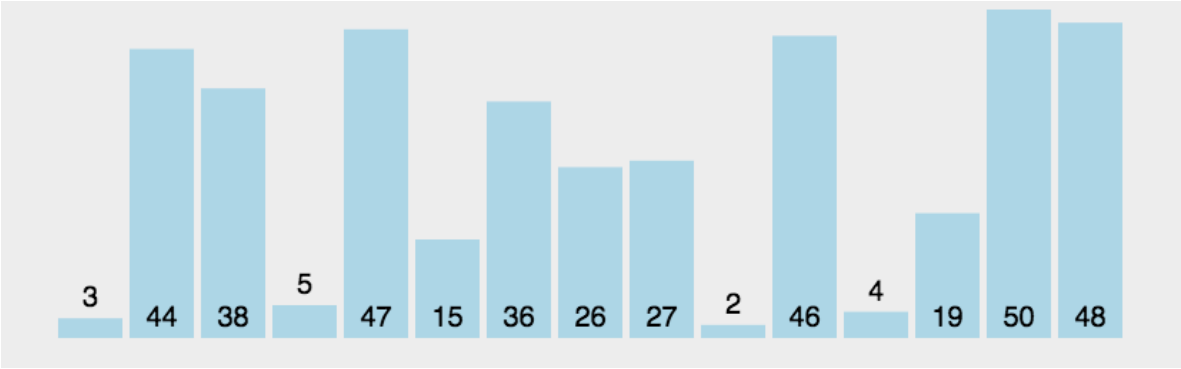
> 二维数组



8.3 数组的排序算法

8.3.1 冒泡法

> 对数组进行排序，冒泡排序法的原理就是将一组无序数组进行排序，同过把值较小的数逐渐向数组的顶部（即朝第一个元素）冒出来，就像水中的气泡上升一样。同时，值较大的数据逐渐向数组的底部（即朝最后一个元素）沉下去。这种算法用嵌套的循环对整个数组进行数次遍历，每次遍历都要比较数组中相邻的一对元素，如果这对元素以升序（或者值相等）的顺序排列，那么保持它们的位置不变；如果这对元素以降序的顺序排列，那么交换他们的值。



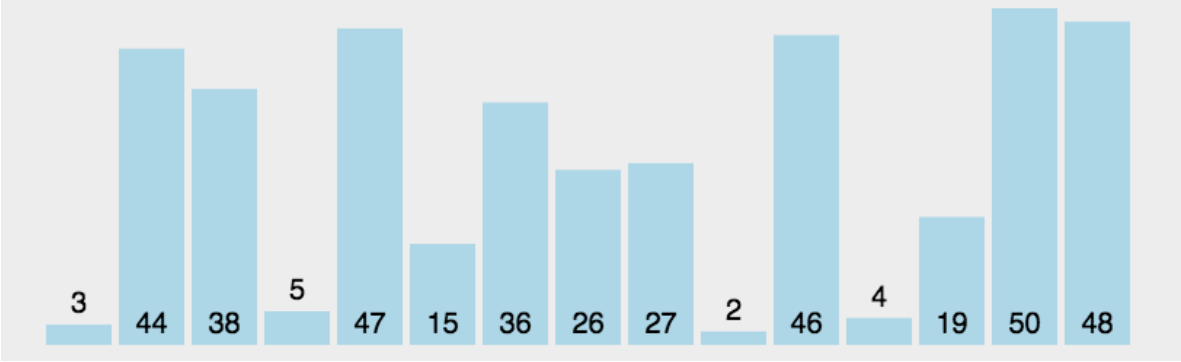
总结：对于一个具有n个数值的无序数组，需要进行（n-1）次重新排序，每次排序需进行（n-i）次比较。（i为重新排序的次数）

> **冒泡法特点：效率低，实现简单**

```
public void bubbleSort(int array[]) {
        int t = 0;
        for (int i = 0; i < array.length - 1; i++)
            for (int j = 0; j < array.length - 1 - i; j++)
                if (array[j] > array[j + 1]) {
                    t = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = t;
                }
        }
}
```

8.3.2 选择排序

> **特点：效率低，容易实现。 思想：每一趟从待排序序列选择一个最小的元素放到已排好序序列的末尾，剩下的为待排序序列，重复上述步骤直到完成排序。**

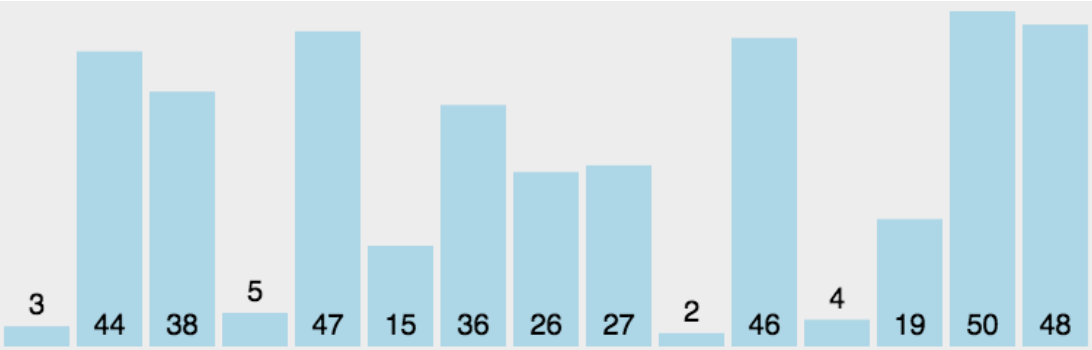



```
public void selectSort(int array[]) {
        int t = 0;
        for (int i = 0; i < array.length - 1; i++){
            int index=i;
            for (int j = i + 1; j < array.length; j++)
                if (array[index] > array[j])
                    index=j;
                    if(index!=i){ //找到了比array[i]小的则与array[i]交换位置
                        t = array[i];
                        array[i] = array[index];
                        array[index] = t;
                    }
            }
```

```
        }
    }
```

8.3.3 插入算法

> 特点：效率低，容易实现。 思想：将数组分为两部分，将后部分元素逐一与前部分元素比较，如果前部分元素比array[i]小，就将前部分元素往后移动。当没有比array[i]小的元素，即是合理位置，在此位置插入array[i]




```
public static void insertionSort(int[] arr){
        for( int i = 1; i < arr.length; i++ ) {
            int temp = arr[i];    // 取出下一个元素，在已经排序的元素序列中从后向前扫描
            for( int j = i; j >= 0; j-- ) {
                if( j > 0 && arr[j-1] > temp ) {
                    arr[j] = arr[j-1];    // 如果该元素（已排序）大于取出的元素temp，将该元素移到下一位置
                    System.out.println("Temping:  " + Arrays.toString(arr));
                } else {
                            // 将新元素插入到该位置后
                    arr[j] = temp;
                    System.out.println("Sorting:  " + Arrays.toString(arr));
                    break;
                }
            }
        }
}
```

8.4 快速排序

> 特点：高效，时间复杂度为nlogn。 采用分治法的思想：首先设置一个轴值pivot，然后以这个轴值为划分基准将待排序序列分成比pivot大和比pivot小的两部分，接下来对划分完的子序列进行快排直到子序列为一个元素为止。

```
public void quickSort(int array[], int low, int high) {// 传入low=0, high=array.length-1;
```

```
        int pivot, p_pos, i, t;// pivot->位索引;p_pos->轴值。
        if (low < high) {
            p_pos = low;
            pivot = array[p_pos];
            for (i = low + 1; i <= high; i++)
                if (array[i] > pivot) {
                    p_pos++;
                    t = array[p_pos];
                    array[p_pos] = array[i];
                    array[i] = t;
                }
            t = array[low];
            array[low] = array[p_pos];
            array[p_pos] = t;
            // 分而治之
            quickSort(array, low, p_pos - 1);// 排序左半部分
            quickSort(array, p_pos + 1, high);// 排序右半部分
```

```
import java.util.Arrays;
public class sortTest {
    // 冒泡排序
    public void bubbleSort(int array[]) {
        int t = 0;
        for (int i = 0; i < array.length - 1; i++)
            for (int j = 0; j < array.length - 1 - i; j++)
                if (array[j] > array[j + 1]) {
                    t = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = t;
                }
    }

    // 选择排序
    public void selectSort(int array[]) {
        int t = 0;
        for (int i = 0; i < array.length - 1; i++){
            int index=i;
            for (int j = i + 1; j < array.length; j++)
                if (array[index] > array[j])
                    index=j;
            if(index!=i){ //找到了比array[i]小的则与array[i]交换位置
                t = array[i];
                array[i] = array[index];
                array[index] = t;
            }
        }
    }

public void insertionSort(int array[]) {
        int i, j, t = 0;
        for (i = 1; i < array.length; i++) {
            if(a[i]<a[i-1]){
                t = array[i];
                for (j = i - 1; j >= 0 && t < array[j]; j--)
                    array[j + 1] = array[j];
                //插入array[i]
                array[j + 1] = t;
```

```
            }
        }
}

    // 分治法快速排序
    public void quickSort(int array[], int low, int high) {// 传入low=0,
high=array.length-1;
        int pivot, p_pos, i, t;// pivot->位索引;p_pos->轴值。
        if (low < high) {
            p_pos = low;
            pivot = array[p_pos];
            for (i = low + 1; i <= high; i++)
                if (array[i] > pivot) {
                    p_pos++;
                    t = array[p_pos];
                    array[p_pos] = array[i];
                    array[i] = t;
                }
            t = array[low];
            array[low] = array[p_pos];
            array[p_pos] = t;
            // 分而治之
            quickSort(array, low, p_pos - 1);// 排序左半部分
            quickSort(array, p_pos + 1, high);// 排序右半部分
        }
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int[] array = { 37, 47, 23, 100, 19, 56, 56, 99, 9 };
        sortTest st = new sortTest();
        // st.bubbleSort(array);
        // st.selectSort(array);
        // st.insertionSort(array);
        st.quickSort(array, 0, array.length - 1);
        System.out.println("排序后: " + Arrays.toString(array));
    }
}
```

初始状态 10 个数

索引	0	1	2	3	4	5	6	7	8	9
数值	72	6	57	88	60	42	83	73	48	85

指定起始索引 i = 0 和结束索引 j = 9 ；取第一个数 72 为基准数，将 72 取出来空出位置，此时我们把他当作是空的，但是还是有东西的。

索引	0	1	2	3	4	5	6	7	8	9		
数值		6	57	88	60	42	83	73	48	85		72
	i=0									j=9		x

从右向左移动 j 找到第一个小于 72 的数 48

索引	0	1	2	3	4	5	6	7	8	9		
数值		6	57	88	60	42	83	73	48	85		72
	i=0								j = 8			x

将 48 放在 72 空出的位置，并将 i 从左向右移动一位（i++），此时 j = 8 的位置变成了空

索引	0	1	2	3	4	5	6	7	8	9		
数值	48	6	57	88	60	42	83	73		85		72
		i=1							j = 8			x

继续向右移动 i，直到找到第一个大于 72 的数 88 停止，此时 i=3

索引	0	1	2	3	4	5	6	7	8	9		
数值	48	6	57	88	60	42	83	73		85		72
				i = 3					j = 8			x

将找到的大于 72 的 88 放在 j = 8 的空的位置，此时 i = 3 的位置变成空

索引	0	1	2	3	4	5	6	7	8	9		
数值	48	6	57		60	42	83	73	88	85		72
				i = 3					j = 8			x

再从右向左移动 j ，找到第一个小于 72 的数 42，将 42 放进 i = 3 的空位置里，此时 j= 5 的位置变成空的了，

索引	0	1	2	3	4	5	6	7	8	9		
数值	48	6	57	42	60		83	73	88	85		72
				i = 3		j=5						x

接下来我不说也应该知道了 继续从左向右移动 i 找第一个比 72 大的，但是直到 i = j，都没出现比 72 大的，那么这个 i = j 便是一个结束的信号，可不是排序结束哈，是这一次排序结束的信号。

索引	0	1	2	3	4	5	6	7	8	9		
数值	48	6	57	42	60		83	73	88	85		72
						j=5						x
						i =5						

遇见了结束信号，我们就要将取出的基准数放在空的位置里了，此时比 72 小的都在左边，比 72 大的都在右边了。那么第一次排序结束。文中出现的 i 右移 j 左移。分别对应的是（i ++ 和 j --）。
想要完全结束还早着呢，再来就是将基准数两边的区间的数进行我以上的操作了（递归），小生就不再继续演示了。（要吐了。。。。。。。）

索引	0	1	2	3	4	5	6	7	8	9		
数值	48	6	57	42	60	72	83	73	88	85		
						j=5						x
						i =5						

9.面向对象

9.1 面向对象三大特性

1. 封装：类里有方法，属性，可以封装很多功能 仅仅对外暴露一些接口，来实现模块化，组建化，和安全性；
2. 继承：Java中的类可以继承，类似子女可以继承父母的东西；来实现可扩展；
3. 多态：Java中的父类接口可以指向子类实现的引用；这个我们后面通过实例详细讲解；

9.1 认识类

```
package com.java1234.chap03.sec01;

/**
 * Person类 文件名和类名必须一致
 * @author user
 *
 */
public class Person {
```

```
    String name; // 在类中，定义一个姓名name字符串属性 可以存在字符串 类似"张三"
    int age; // 在类中，定义一个年龄age属性

    /**
     * 定义一个方法  public表示共有 权限最大  void表示返回值是空  speak是方法名 括号里可以加参数
     */
    public void speak(){
        System.out.println("我叫"+name+" 我今年"+age);
    }

    public static void main(String[] args) {
        // 定义一个Person类的对象zhangsan
        Person zhangsan;
        // 实例化对象
        zhangsan=new Person();
        // 给对象的name属性赋值
        zhangsan.name="张三";
        // 给对象的age属性赋值
        zhangsan.age=23;
        zhangsan.speak(); // 调用对象的方法
    }
}
```

9.2方法

> 方法是对象的一部分，也称为行为

9.2.1 无参的方法

```
package com.java1234.chap03.sec03;

public class Person {

    void speak(){
        System.out.println("我是张三");
    }

    public static void main(String[] args) {
        Person person=new Person();
        person.speak();
    }
}
```

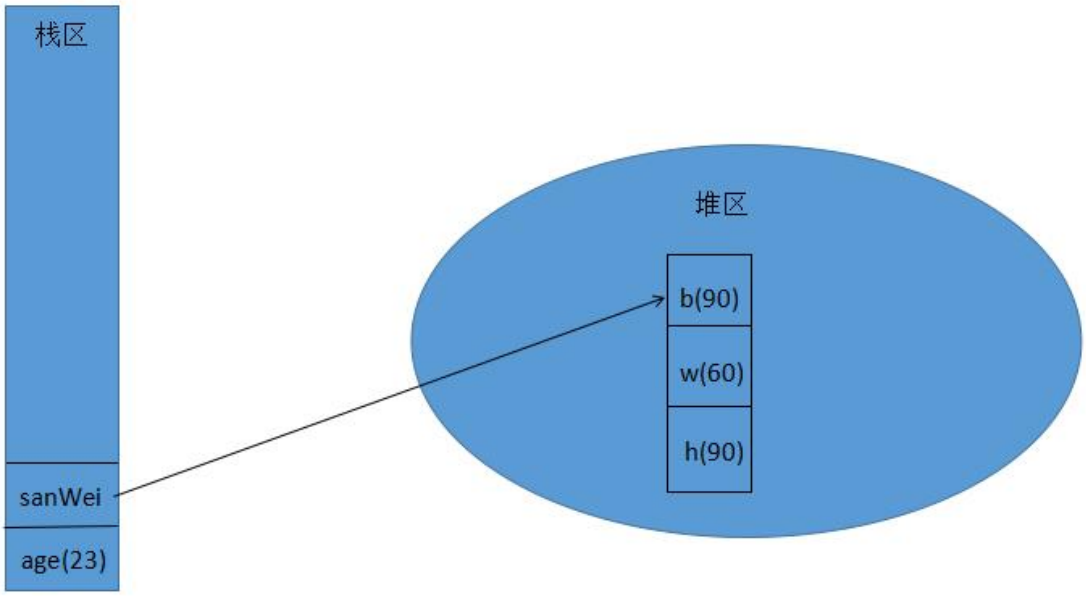
9.2.2 有参的方法

```java
package com.java1234.chap03.sec03;

public class Person {

    void speak(String name,int age){
        System.out.println("我叫"+name+",我今年"+age+"岁了");
    }

    public static void main(String[] args) {
        Person person=new Person();
        person.speak("张三",23);
    }
}
```

9.2.3 不固定参数的方法

```java
package com.java1234.chap03.sec03;

public class Person {

    int speak(String name,int age,String ...hobbies){
        System.out.println("我叫"+name+",我今年"+age+"岁了");
        for(String hobby:hobbies){
            System.out.println(hobby+" ");
        }
        // 获取爱好的长度
        int totalHobbies=hobbies.length;
        return totalHobbies;
    }

    public static void main(String[] args) {
        Person person=new Person();
        int n=person.speak("张三",23,"游泳","唱歌");
        System.out.println("有"+n+"个爱好");
    }
}
```

9.3 方法的值传递和引用传递



9.3.1 值传递

> 方法的值传递：
>
> 值传递 在方法里改变变量的值 作用范围仅仅是方法里 对外面不影响：

上代码：

```
package com.java1234.chap03.sec03;

public class Person {

    void speak(int age){
        System.out.println("我今年"+age+"岁了");
        age=24; // 作用范围是方法里
    }

    public static void main(String[] args) {
        Person xiaoli=new Person();
        int age=23;
        xiaoli.speak(age);
        System.out.println(age);
    }
}
```

> 引用传递，传递的是地址，对象里的属性在方法里值修改，对外面有影响，我们通过对象.属性可以获取到最新的数据；

上代码：

```
package com.java1234.chap03.sec03;

class SanWei{
    int b; // 胸围
    int w; // 腰围
    int h; // 腿围
```

```
}

public class Person {

    void speak(int age,SanWei sanWei){
        System.out.println("我今年"+age+"岁了，我的三围
是："+sanWei.b+","+sanWei.w+","+sanWei.h);
        age=24; // 作用范围是方法里
        sanWei.b=80; // 胸围改成80
    }

    public static void main(String[] args) {
        Person xiaoli=new Person();
        int age=23;
        SanWei sanWei=new SanWei();
        sanWei.b=90;
        sanWei.w=60;
        sanWei.h=90;
        // age传递的是值 ， sanWei是对象，传递的是引用(地址,c里叫指针)
        xiaoli.speak(age,sanWei);
        System.out.println(age);
        System.out.println(sanWei.b);
    }
}
```

9.4方法的重载

> 类里面有两个或者多个重名的方法，但是方法的参数个数、类型、顺序至少有一个不一样，这时候局构成方法重载； 上代码：

```
ackage com.java1234.chap03.sec03;

public class Demo01 {

    int add(int a,int b){
        System.out.println("方法一");
        return a+b;
    }

    /**
     * 方法的重载，参数个数不一样
     * @param a
     * @param b
     * @param c
     * @return
     */
    int add(int a,int b,int c){
        System.out.println("方法二");
        return a+b+c;
    }

    /**
     * 方法的重载，参数的类型不一样
     * @param a
     * @param b
     * @return
     */
```

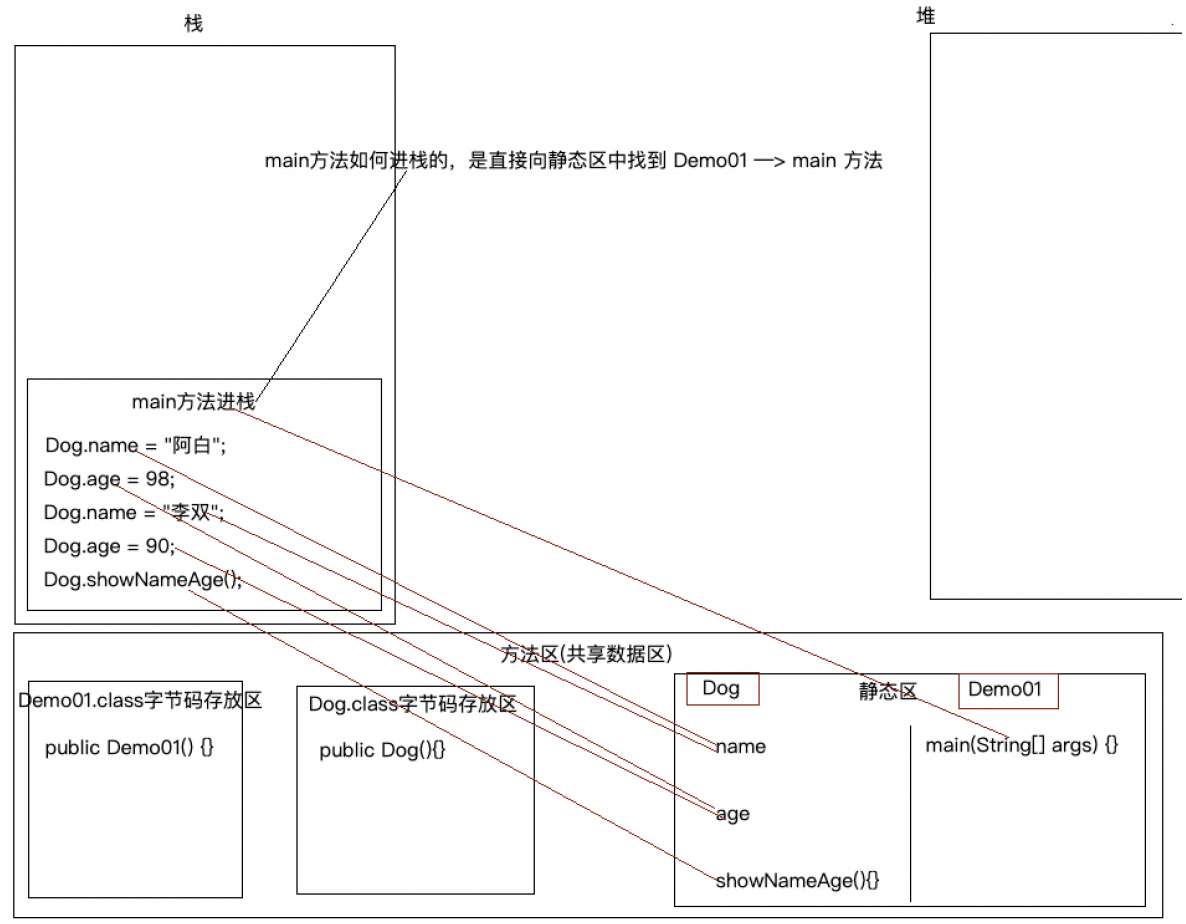
```
    int add(int a,String b){
        System.out.println("方法三");
        return a+Integer.parseInt(b);
    }

    /**
     * 参数类型个数一样，返回值类型不一样 不算重载，直接会报错，说方法重名
     * @param args
     */
    /*long add(int a,int b){
        return a+b;
    }*/

    public static void main(String[] args) {
        Demo01 demo=new Demo01();
        System.out.println(demo.add(1, 2));
        System.out.println(demo.add(1, 2,3));
        System.out.println(demo.add(1, "3"));
    }
}
```

（注释：这里有个注意点 假如参数个数和类型一样，返回值不一样，不能算重载，直接是编译出错，编译器认为是方法重复了。）

```
package com.java1234.chap03.sec03;

public class Demo01 {

    int add(int a,int b){
        System.out.println("方法一");
        return a+b;
    }

    /**
     * 参数类型个数一样，返回值类型不一样 不算重载，直接会报错，说方法重名
     * @param args
     */
    long add(int a,int b){
        return a+b;
    }

    public static void main(String[] args) {
        Demo01 demo=new Demo01();
        int m=demo.add(1, 2);  // 调用返回值是int类型的方法
        long n=demo.add(1, 2); // 调用返回值是long类型的方法

        demo.add(1, 2); // 假如这样些  编译器晕了，你TM是调用返回值是int类型还是long类型的方法

    }
}
```

9.5 静态方法和普通方法

> 静态方法 是加了static修饰词的方法



上代码：

```
package android.java.oop10;

public class Demo01 {

    public static void main(String[] args) {
        Dog.name = "阿白";
        Dog.age = 98;

        Dog.name = "李双";
        Dog.age = 90;

        Dog.showNameAge();
    }

}
```

9.5.1 执行特点

> 静态方法和实例方法的区别主要体现在两个方面：
>
> 在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
>
> 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制。

1.调用静态方法实例

```
//-----------hasStaticMethod.java-----------------
public class hasStaticMethod{
//定义一个静态方法
public static void callMe(){
    System.out.println("This is a static method.");
}
}
```

下面这个程序使用两种形式来调用静态方法。

```
//-----------invokeStaticMethod.java-----------------
public class invokeStaticMethod{
    public static void main(String args[]){
    hasStaticMethod.callMe(); //不创建对象，直接调用静态方法
    hasStaticMethod oa = new hasStaticMethod();  //创建一个对象
    oa.callMe();   //利用对象来调用静态方法
}
}
```

程序两次调用静态方法，都是允许的，程序的输出如下： This is a static method.This is a static method.

2、静态方法访问成员变量示例

```
//-----------accessMember.java-----------------
class accessMember{
private static int sa; //定义一个静态成员变量
private int ia; //定义一个实例成员变量
//下面定义一个静态方法
static void statMethod(){
    int i = 0;  //正确，可以有自己的局部变量sa = 10;
    //正确，静态方法可以使用静态变量
    otherStat();
    //正确，可以调用静态方法
    ia = 20;  //错误，不能使用实例变量
    insMethod(); //错误，不能调用实例方法
}
static void otherStat(){}
//下面定义一个实例方法
void insMethod(){
    int i = 0;  //正确，可以有自己的局部变量
    sa = 15;  //正确，可以使用静态变量
    ia = 30;  //正确，可以使用实例变量
    statMethod();  //正确，可以调用静态方法
}
}
```

9.6 递归

> 用递归实现斐波那契数列 1、1、2、3、5、8、13、21、...
>
> F(1)=1, f(2)=1
>
> F(N)=F(N-1)+F(N-2)

上代码：

```
package com.java1234.chap03.sec03;

public class Test {

    long fun(int n){
        if(n==1 || n==2){
            return 1;
        }
        return fun(n-1)+fun(n-2);
    }

    public static void main(String[] args) {
        System.out.println(new Test().fun(7));
    }
}
```

9.7 构造方法和this 关键字

> 构造方法:
>
> 构造方法是一个特殊的方法，这个特殊方法用于创建实例时执行初始化操作；

上代码：

```
package com.java1234.chap03.sec04;

/**
 * 定义人类
 * @author user
 *
 */
public class People {

    // 定义属性
    private String name; // 实例化对象时，默认值是null
    private int age; // 实例化对象时，默认值是0

    /**
     * 默认构造方法
     */
    People(){
        System.out.println("默认构造方法！");
    }



    public void say(){
        System.out.println("我叫："+name+"，我今年："+age);
    }

    public static void main(String[] args) {
        People people=new People();
        people.say();
    }
}
```

运行输出:

```
默认构造方法！
我叫：null，我今年：0
```

这里我们发现： 实例化对象的时候，String类型的默认值是null，int基本类型的默认值是0；

People(){} 构造方法特点 ：

- 没有返回值类型，区别于其他一般方法；
- 方法名和类名一样；

9.7.1 有参构造方法

> 用来初始化对象属性；

```java
package com.java1234.chap03.sec04;

/**
 * 定义人类
 * @author user
 *
 */
public class People {

    // 定义属性
    private String name; // 实例化对象时，默认值是null
    private int age; // 实例化对象时，默认值是0

    /**
     * 默认构造方法
     */
    People(){
        System.out.println("默认构造方法！");
    }

    /**
     * 有参数的构造方法
     * @param name2
     * @param age2
     */
    People(String name2,int age2){
        System.out.println("调用的是有参数的构造方法");
        name=name2;
        age=age2;
    }

    public void say(){
        System.out.println("我叫："+name+"，我今年："+age);
    }

    public static void main(String[] args) {
        // People people=new People();
        People people=new People("张三",20);
        people.say();
    }
}
```

运行输出：

```
调用的是有参数的构造方法
我叫：张三，我今年：20
```

(注意点：假如没有构造方法，系统会自动生成一个默认的无参构造方法；)

9.7.2 this关键字

> this关键字
>
> this表示当前对象
>
> 1，使用this调用本类中的属性；
>
> 2，使用this调用构造方法；

这里警告的意思 是自己赋值给自己 没有意义，这里的name和age变量 作用范围在方法里 所以和类里的属性名搭不上边；

上代码解决上述问题，采用this.属性:

```
package com.java1234.chap03.sec04;

/**
 * 定义人类
 * @author user
 *
 */
public class People {

    // 定义属性
    private String name; // 实例化对象时，默认值是null
    private int age; // 实例化对象时，默认值是0

    /**
     * 默认构造方法
     */
    /*People(){
        System.out.println("默认构造方法！");
```

```
    }*/

    /**
     * 有参数的构造方法
     * @param name
     * @param age
     */
    People(String name,int age){
        System.out.println("调用的是有参数的构造方法");
        this.name=name;
        this.age=age;
    }

    public void say(){
        System.out.println("我叫："+name+"，我今年："+age);
    }

    public static void main(String[] args) {
        // People people=new People();
        People people=new People("张三",20);
        people.say();
    }
}
```

9.7.3 this()

> this()就是调用无参数构造方法，括号里也可以加参数，来调用有参数的构造方法

上代码:

```
package com.java1234.chap03.sec04;

/**
 * 定义人类
 * @author user
 *
 */
public class People {

    // 定义属性
    private String name; // 实例化对象时，默认值是null
    private int age; // 实例化对象时，默认值是0

    /**
     * 默认构造方法
     */
    People(){
        System.out.println("默认构造方法！");
    }

    /**
     * 有参数的构造方法
     * @param name
     * @param age
     */
    People(String name,int age){
        this(); // 调用无参数的构造方法
```

```
        System.out.println("调用的是有参数的构造方法");
        this.name=name;
        this.age=age;
    }

    public void say(){
        System.out.println("我叫："+name+"，我今年："+age);
    }

    public static void main(String[] args) {
        // People people=new People();
        People people=new People("张三",20);
        people.say();
    }
}
```

输出:

```
默认构造方法!
调用的是有参数的构造方法
我叫：张三，我今年：20
```

9.8 访问控制权限

> Java中，可以通过一些Java关键字，来设置访问控制权限；
>
> 主要有 private(私有)， package(包访问权限)，protected(子类访问权限)，public(公共访问权限)

	private	package	protected	public
同一个类中	√	√	√	√
同一个包中		√	√	√
子类中			√	√
全局范围内				√

上代码:

```
package com.java1234.chap03.sec05;

public class Demo1 {

    /**
     * 定义一个私有的属性a
     */
    private int a;

    public int getA() {
        return a;
    }

    public void setA(int a) {
        this.a = a;
```

```
    }


}
```

测试类：

```
package com.java1234.chap03.sec05;

public class TestDemo1 {

    public static void main(String[] args) {
        Demo1 demo1=new Demo1();
        demo1.setA(2);
        int a=demo1.getA();
        System.out.println(a);
    }
}
```

9.9 内部类

> 定义：在类的内部定义的类；
>
> 优点：可以方便的额使用外部类的属性；
>
> 缺点：破坏类的基本结构；
>
> 建议，慎用内部类

上代码：（间接调用内部类）

```
package com.java1234.chap03.sec06;

/**
 * 外部类
 * @author user
 *
 */
public class Outer {

    private int a=1;

    /**
     * 内部类
     * @author user
     *
     */
    class Inner{
        public void show(){
            System.out.println(a); // 可以方便的额使用外部类的属性；
        }
    }

    public void show(){
        Inner inner=new Inner();
        inner.show();
    }
```

```
    public static void main(String[] args) {
        Outer outer=new Outer();
        outer.show();
    }
}
```

我们新建一个TestInner类，直接调用内部类的方法：

```
package com.java1234.chap03.sec06;

public class TestInner {

    public static void main(String[] args) {
        Outer outer=new Outer();
        Outer.Inner inner=outer.new Inner();
        inner.show();
    }
}
```

9.10 代码块

> 代码块主要就是通过{}花括号 括起来的代码；
>
> 主要分为 普通代码块 构造块 静态代码块三类。后面学到线程还有一个同步代码块

普通代码块：仅仅是花括号括起来的代码块，个人感觉作用不大，我们来看一个实例：

```
package com.java1234.chap03.sec07;

public class Demo1 {

    public static void main(String[] args) {
        int a=1;
        /**
         * 普通代码块
         */
        {
            a=2;
            System.out.println("普通代码块");
        }
        System.out.println("a="+a);
    }
}
```

运行输出:

```
普通代码块
a=2
```

构造块 构造块作用就是扩展构造器功能 每次实例化对象都会执行构造块里的内容：

上代码：

```
package com.java1234.chap03.sec07;
```

```
public class Demo2 {

    /**
     * 构造块
     */
    {
        System.out.println("通用构造块！");
    }

    /**
     * 构造方法一
     */
    public Demo2(){
        System.out.println("构造方法一");
    }

    /**
     * 构造方法二
     */
    public Demo2(int i){
        System.out.println("构造方法二");
    }

    /**
     * 构造方法三
     */
    public Demo2(int i,int j){
        System.out.println("构造方法三");
    }

    public static void main(String[] args) {
        new Demo2();
        new Demo2(1);
        new Demo2(1,2);
    }
}
```

运行输出：

```
通用构造块！

构造方法一

通用构造块！

构造方法二

通用构造块！

构造方法三
```

我们发现 每次调用构造方法 都会先执行 构造块

静态代码块：{}花括号前加static修饰词 在类加载的时候执行 而且只执行一次

上代码：

```java
package com.java1234.chap03.sec07;

public class Demo3 {

    /**
     * 静态代码块 类加载的时候执行
     */
    static{
        System.out.println("静态代码块！");
    }

    /**
     * 构造块
     */
    {
        System.out.println("通用构造块！");
    }

    /**
     * 构造方法一
     */
    public Demo3(){
        System.out.println("构造方法一");
    }

    /**
     * 构造方法二
     */
    public Demo3(int i){
        System.out.println("构造方法二");
    }

    /**
     * 构造方法三
     */
    public Demo3(int i,int j){
        System.out.println("构造方法三");
    }

    public static void main(String[] args) {
        new Demo3();
        new Demo3(1);
        new Demo3(1,2);
    }
}
```

运行输出:

```
静态代码块!

通用构造块!

构造方法一

通用构造块!

构造方法二

通用构造块!

构造方法三
```

10.String类

10.1 实例化Sring对象的两种方法

> 方法一

```
String name1="张三";
System.out.println("name1="+name1);
```

> 方法二

```
String name2=new String("李四");
System.out.println("name2="+name2);
```

10.2 ==和equals的区别

> “==” 比较的是引用(是否指向同一个内存块) “
>
> equals”比较的是具体内容

上代码

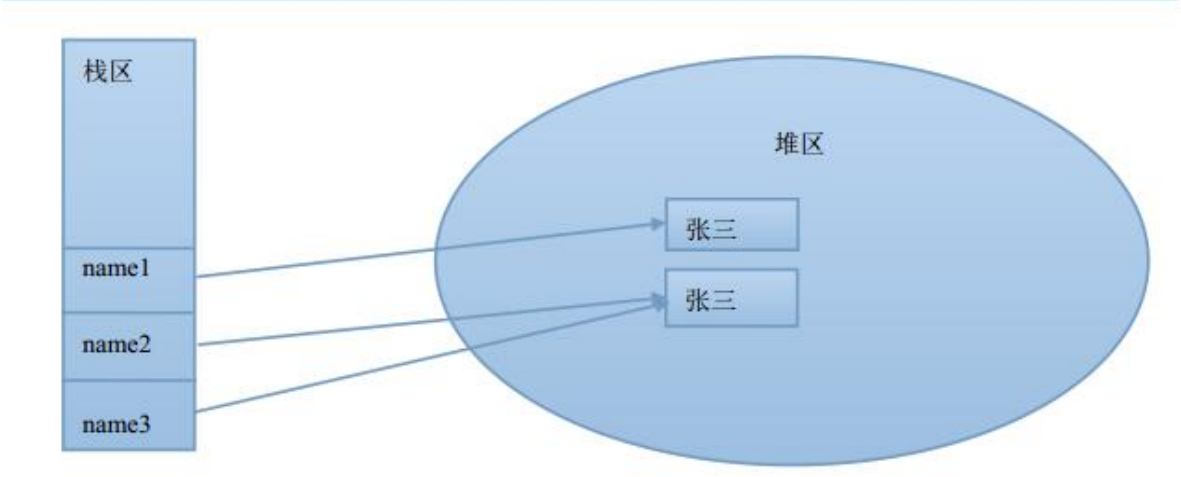
```
package com.java1234.chap03.sec08;

public class Demo2 {

    public static void main(String[] args) {
        String name1="张三";  // 直接赋值方式
        String name2=new String("张三"); // new方式
        String name3=name2; // 传递引用

        // ==比较的是引用
        System.out.println("name1==name2:"+(name1==name2));
        System.out.println("name1==name3:"+(name1==name3));
        System.out.println("name2==name3:"+(name2==name3));

        // equals比较的是具体内容
        System.out.println("name1.equals(name2):"+(name1.equals(name2)));
        System.out.println("name1.equals(name3:"+(name1.equals(name3)));
        System.out.println("name2.equals(name3):"+(name2.equals(name3)));
    }
```

```
}
```

运行输出：

```
name1==name2:false

name1==name3:false

name2==name3:true

name1.equals(name2):true

name1.equals(name3:true

name2.equals(name3):true
```

内存示意图如下:



10.3 两种String实例化方式的不同

> 直接赋值方式，创建的对象存放到字符串对象池里，假如存在的，就不会再创建；
>
> new对象方式，每次都创建一个新的对象

上代码:

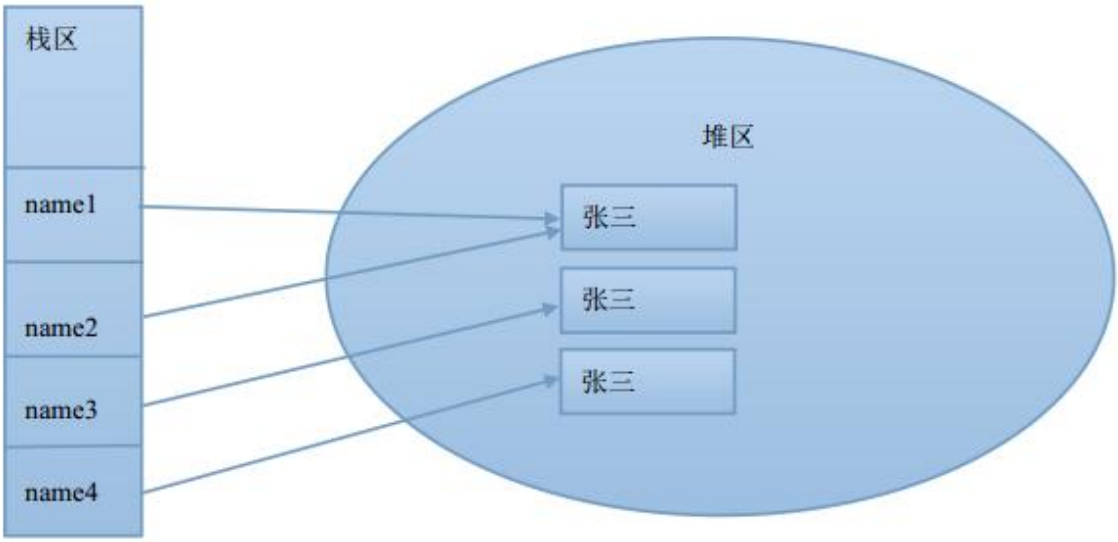
```
package com.java1234.chap03.sec08;

public class Demo3 {

    public static void main(String[] args) {
        String name1="张三";
        String name2="张三";
        String name3=new String("张三");
        String name4=new String("张三");

        System.out.println("name1==name2:"+(name1==name2));
        System.out.println("name1==name3:"+(name1==name3));
        System.out.println("name3==name4:"+(name3==name4));
    }

}
```

运行输出:

```
name1==name2:true

name1==name3:false

name3==name4:false
```

内存示意图

（示意图：jdk7之后，引入常量池）



10.4 字符串常量池（jdk8）

> 常量池大体可以分为：静态常量池，运行时常量池。

> 静态常量池 存在于class文件中，比如经常使用的javap -verbose中，常量池总是在最前面把？
>
> 运行时常量池呢，就是在class文件被加载进了内存之后，常量池保存在了方法区中，通常说的常量池 值的是运行时常量池。所以呢，讨论的都是运行时常量池

```
String a="hello"//存放在堆里面的字符串常量池
```

10.5 String 类的常用方法

1.char chartAt(int index) 返回指定索引处的char值,index从0开始

```
package com.java1234.chap03.sec08;

public class Demo5 {

    public static void main(String[] args) {
        String name="张三";
        char ming=name.charAt(1);
        System.out.println(ming);

        String str="我是中国人";
        for(int i=0;i<str.length();i++){
            System.out.println(str.charAt(i));
        }
    }
}
```

运行输出：

```
三

我

是

中

国

人
```

2.int length()

> 返回字符串的长度；

3.int indexOf(int ch)

> 返回指定字符在此字符串中第一次出现处的索引。

```
package com.java1234.chap03.sec08;

public class Demo06 {

    public static void main(String[] args) {
        // indexOf方法使用实例
        String str="abcdefghijdklmoqprstuds";
        System.out.println("d在字符串str中第一次出现的索引位置: "+str.indexOf('d'));
        System.out.println("d在字符串str中第一次出现的索引位置,从索引4位置开
始: "+str.indexOf('d',4));
    }
}
```

3.String substring(int beginIndex)

> 返回一个新的字符串，它是此字符串的一个子字符串

```
package com.java1234.chap03.sec08;

public class Demo07 {

    public static void main(String[] args) {
        // substring方式读取
        String str="不开心每一天";
        String str2="不开心每一天，不可能";
        String newStr=str.substring(1);
        System.out.println(newStr);
        String newStr2=str2.substring(1, 6);
        System.out.println(newStr2);
    }
}
```

4.public String toUpperCase()

> String 中的所有字符都转换为大写

```
package com.java1234.chap03.sec08;

public class Demo08 {

    public static void main(String[] args) {
        String str="I'm a boy!";
        String upStr=str.toUpperCase(); // 转成大写
        System.out.println(upStr);
        String lowerStr=upStr.toLowerCase(); // 转成小写
        System.out.println(lowerStr);
    }
}
```

10.6 String类综合案例

“ aB232 23 &*(s2 ”指定字符串，要求去掉前后空格，然后分别统计其中英文字符，空格，数字和其他字符的个数；

思路：首先去掉前后空格，我们查找api文本，可以找到trim()方法；要统计的话，我们遍历字符串，然后通过if判断来统计各种字符的个数；

```
package com.java1234.chap03.sec08;

public class Demo09 {

    public static void main(String[] args) {
        String str=" aB232 23 &*( s2 ";
        String newStr=str.trim(); // 去掉前后空格
        System.out.println("str="+str);
        System.out.println("newStr="+newStr);

        int yingWen=0; // 英文个数
        int kongGe=0; // 空格个数
        int shuZi=0; // 数字个数
        int qiTa=0; // 其他

        for(int i=0;i<newStr.length();i++){
            char c=newStr.charAt(i);
            // 判断英文字符
            if((c>='a'&&c<='z')||(c>='A'&&c<='Z')){
                yingWen++;
                System.out.println("英文字符："+c);
            }else if(c>='0'&&c<='9'){
                shuZi++;
                System.out.println("数字字符："+c);
            }else if(c==' '){
                kongGe++;
                System.out.println("空格字符："+c);
            }else{
                qiTa++;
                System.out.println("其他字符："+c);
            }
        }

        System.out.println("英文个数："+yingWen);
        System.out.println("空格个数："+kongGe);
        System.out.println("数字个数："+shuZi);
        System.out.println("其他个数："+qiTa);
    }
}
```

实例2：

> 字符串转数组
>
> int的包装类的parseInt方法 把字符串转成int类型；

```
package com.java1234.chap03.sec08;

public class Test {

    public static void main(String[] args) {
        String str="1,3,5,7,9";
        int shuZi=0;
        for(int i=0;i<str.length();i++){
```

```
            if(str.charAt(i)!=','){
                shuZi++;
            }
        }
        int []arr=new int[shuZi];
        int j=0;
        for(int i=0;i<str.length();i++){
            if(str.charAt(i)!=','){
                arr[j]=Integer.parseInt(str.charAt(i)+"");
                j++;
            }
        }
        for(int a:arr){
            System.out.print(a+" ");
        }
    }
}
```

11 继承

> 定义：子类能够继承父类的属性和方法；
>
> 注意点：Java中只支持单继承；私有方法不能继承；

11.1 对象实例过程以及super关键字

对象实例化 先实例化调用父类构造方法，再调用子类实例化构造方法；

super关键主要是调用父类方法或者属性；

```
public class Animal {

    private String name; // 姓名
    private int age;  // 年龄


    /**
     * 无参父类构造方法
     */
    public Animal() {
        System.out.println("无参父类构造方法");
    }

    /**
     * 有参父类构造方法
     * @param name 姓名
     * @param age 年龄
     */
    public Animal(String name,int age) {
        System.out.println("有参父类构造方法");
        this.name=name;
        this.age=age;
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    public void say(){
        System.out.println("我是一个动物，我叫："+this.name+",我的年龄
是："+this.age);
    }
}
```

```
/**
 * 定义一个Cat类，继承自Animal
 * @author user
 *
 */
public class Cat extends Animal{

    private String address;


    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public Cat() {
        super();
        System.out.println("子类无参构造方法");
    }

    public Cat(String name, int age,String address) {
        super(name, age);
        this.address=address;
        System.out.println("子类有参构造方法");
    }

    /**
     * 重写父类的say方法
     */
    public void say(){
        super.say(); // 调用父类的say()方法
        System.out.println("我是一个猫，我叫："+this.getName()+",我的年龄
是："+this.getAge()+"，我来自："+this.getAddress());
    }

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

```
        Cat cat=new Cat("Mini",2,"火星");
        /*cat.setName("Mini");
        cat.setAge(2);*/
        cat.say();
    }
}
```

运行输出：

```
有参父类构造方法

子类有参构造方法

我是一个动物，我叫：Mini,我的年龄是：2

我是一个猫，我叫：Mini,我的年龄是：2，我来自：火星
```

12 final

> 使用final声明的类不能被继承；
>
> 使用final声明的方法不能被子类覆盖；
>
> 使用final声明的变量不能被修改，即为常量；

final修饰字段的时候 一般和static一起使用，来定义一些不可变的静态常量 方便程序使用；

```
package com.java1234.chap03.sec10;

public class Common {

    /**
     * 静态常量
     */
    public static final String CHINA_CAPITAL="北京";
}
```

我们调用的话 直接类名.属性 即可；

```
package com.java1234.chap03.sec10;

public class Test{

    public static void main(String[] args) {
        System.out.println(Common.CHINA_CAPITAL);
    }
}
```

13 抽象类

> 定义：在java中，含有抽象方法的类称为抽象类，同样不能生成对象；
>
> 注意点：
>
> 1，包含一个抽象方法的类是抽象类；

> 2，抽象类和抽象方法都要用abstract关键字修饰；
>
> 3，抽象方法只需要声明而不需要实现；
>
> 4，抽象类必须被子类（假如不是抽象类）必须重写抽象中的全部抽象方法；
>
> 5，抽象类不能被实例化；

定义一个抽象类：

```
package com.java1234.chap03.sec11;

/**
 * 定义一个抽象类People
 * @author user
 *
 */
public abstract class People {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void say(){
        System.out.println("我的姓名是："+this.getName());
    }

    /**
     * 定义一个抽象方法 职业 让子类去具体实现
     */
    public abstract void profession();

}
```

14 接口

> 接口定义：一种特殊的“抽象类”，没有普通方法，由全局常量和公共的抽象方法所组成；

上代码：

```
package com.java1234.chap03.sec12;

/**
 * 定义一个接口A
 * @author user
 *
 */
public interface A {

    /**
     * 全局常量
```

```
     */
    public static final String TITLE="www.java1234.com";

    /**
     * 定义一个抽象方法 abstract可以省略
     */
    public abstract void a();
}
```

(注意点：由于接口里的方法都是抽象的，所以abstract可以省略，实际开发一般都是省略的，开发者的习惯)

实现接口：

```
package com.java1234.chap03.sec12;

public class Test implements A{

    @Override
    public void a() {
        System.out.println("a方法");
    }

    public static void main(String[] args) {
        Test t=new Test();
        t.a();
        System.out.println(Test.TITLE);
    }
}
```

运行输出：

```
a方法
```

14.1 继承类和多接口

> 接口A

```
/**
 * 定义一个接口A
 * @author user
 *
 */
public interface A {

    /**
     * 全局常量
     */
    public static final String TITLE="www.java1234.com";

    /**
     * 定义一个抽象方法 abstract可以省略
     */
    public abstract void a();
}
```

> 接口B

```
 /**
 * 定义一个接口B
 * @author user
 *
 */
public interface B {

    /**
     * 全局常量
     */
    public static final String TITLE2="java知识分享网";

    /**
     * 定义一个抽象方法 abstract可以省略
     */
    public abstract void b();
}
```

> 类C

```
package com.java1234.chap03.sec12;

public class C {

    public void c(){
        System.out.println("c方法");
    }
}
```

> 测试类（先继承，后接口）

```
public class Test extends C implements A,B{

    @Override
    public void a() {
        System.out.println("a方法");
    }

    @Override
    public void b() {
        System.out.println("b方法");
    }

    public static void main(String[] args) {
        Test t=new Test();
        t.a();
        t.b();
        t.c();
        System.out.println(Test.TITLE);
        System.out.println(Test.TITLE2);
    }


}
```

运行输出：

```
a方法

b方法

c方法

www.java1234.com

java知识分享网
```

14.2 接口的多继承

> 定义一个D接口 继承A,B接口：

```
public interface D extends A,B{

    public void d();
}
```

> 测试类

```
public class Test2 extends C implements D{

    @Override
    public void a() {
        System.out.println("a方法");
    }

    @Override
    public void b() {
        System.out.println("b方法");
    }

    @Override
    public void d() {
        System.out.println("d方法");
    }

    public static void main(String[] args) {
        Test2 t=new Test2();
        t.a();
        t.b();
        t.c();
        t.d();
        System.out.println(Test2.TITLE);
        System.out.println(Test2.TITLE2);
    }
}
```

运行输出：

```
a方法

b方法

c方法

d方法

www.java1234.com

java知识分享网
```

15.多态

> Java中的多态性表现：
>
> 1，方法的重载和重写；
>
> 2，可以用父类的引用指向子类的具体实现，而且可以随时更换为其他子类的具体实现；

> 1.先定义一个父类Animal

```
public class Animal {

    public void say(){
        System.out.println("我是一个动物");
    }
}
```

> 2.定义两个子类，分别是Dog和Cat类，重写父类方法say：

```
public class Cat extends Animal{

    public void say(){
        System.out.println("我是一个猫");
    }
}
```

```
public class Dog extends Animal{

    public void say(){
        System.out.println("我是一个狗");
    }
}
```

> 3.写测试类

```
public class Test {

    public static void main(String[] args) {
        Dog dog=new Dog();
        dog.say();

        Cat cat=new Cat();
        cat.say();
    }
}
```

运行输出：

```
我是一个狗

我是一个猫
```

> 4.多态测试，父类引用指向子类具体实现

```
public class Test {

    public static void main(String[] args) {
        // 父类引用指向Dog子类的具体实现
        Animal animal=new Dog();
        animal.say();

        // 更换实现
        animal=new Cat();
        animal.say();
    }
}
```

运行输出：

```
我是一个狗

我是一个猫
```

15.1 对象的转型

> 对象的转型：
>
> 向上转型：子类对象->父类对象 安全
>
> 向下转型：父类对象->子类对象 不安全

比如上面的实例：Animal animal=new Dog(); 就是后面的new Dog() 子类对象 向上 Animail animal 转型 是安全的；

15.2 多态在接口上的应用

> 1.定义父类接口Person

```
public interface People {

    public void say();
}
```

> 2.实现类Student和Teachert

```
public class Student implements People{

    @Override
    public void say() {
        System.out.println("我是学生");
    }

}
```

```
public class Teacher implements People{

    @Override
    public void say() {
        System.out.println("我是老师");
    }

}
```

> 3.编写测试类

```
public class Test2 {

    public static void main(String[] args) {
        People p1=new Student();
        p1.say();

        p1=new Teacher();
        p1.say();
    }
}
```

运行输出:

```
我是学生

我是老师
```

16.Object类

> Object类是所有类的父类；

Object类的常用方法

1，public String toString() 返回该对象的字符串表示。

2，public boolean equals(Object obj) 指示其他某个对象是否与此对象“相等”

> alt+shift+s:调出方法菜单
>
> ctrl+o:调出当前类的方法outline

> 1.测试类（默认）

```
public class People {

    private String name;

    public People(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        People p1=new People("张三");
        System.out.println(p1);
        System.out.println(p1.toString());
    }
}
```

运行输出：

```
com.java1234.chap03.sec14.People@15db9742

com.java1234.chap03.sec14.People@15db9742
```

> 2.重写toString()

```
public class People {

    private String name;

    public People(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }


    @Override
    public String toString() {
        return this.getName();
    }

    public static void main(String[] args) {
        People p1=new People("张三");
        System.out.println(p1);
        System.out.println(p1.toString());
    }
}
```

运行输出：

```
张三

张三
```

17.equals( )方法

> 不同于String类的equals方法，其他事比较对象的引用

> equals 是比较对象的引用，是否指向同一个堆内存；

```
public class People {

    private String name;

    public People(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }


    public static void main(String[] args) {
        People p1=new People("张三");
        People p2=new People("张三");
        System.out.println(p1.equals(p2));
    }
}
```

运行输出：

```
false
```

18 instanceof关键字

> 作用：判断一个对象是否属于一个类
>
> 格式：对象 instanceof 类 返回布尔类型

> 1.新建Animal类

```
public class Animal {

    public void say(){
        System.out.println("我是一个动物");
    }
}
```

> 2.子类Dog，Cat类：

```
public class Dog extends Animal{

    public void say(){
        System.out.println("我是一只狗");
    }
}
```

```
public class Cat extends Animal{

    public void say(){
        System.out.println("我是一只猫");
    }
}
```

> 3.测试类：

```
package com.java1234.chap03.sec14;

public class Test {

    public static void main(String[] args) {
        Animal dog=new Dog();
        System.out.println("dog对象是否属于Animal类："+(dog instanceof Animal));
        System.out.println("dog对象是否属于Dog类："+(dog instanceof Dog));
        System.out.println("dog对象是否属于Cat类："+(dog instanceof Cat));
    }
}
```

输出：

```
dog对象是否属于Animal类：true

dog对象是否属于Dog类：true

dog对象是否属于Cat类：false
```

> instanceof我们可以确保向下转型的不出问题

```
public class Test {

    public static void doSomeThing(Animal animal){
        animal.say();
        if(animal instanceof Dog){
            ((Dog) animal).f1();
        }else if(animal instanceof Cat){
            ((Cat) animal).f2();
        }
    }

    public static void main(String[] args) {
        Animal dog=new Dog();
        System.out.println("dog对象是否属于Animal类："+(dog instanceof Animal));
        System.out.println("dog对象是否属于Dog类："+(dog instanceof Dog));
        System.out.println("dog对象是否属于Cat类："+(dog instanceof Cat));

        doSomeThing(new Dog());
        doSomeThing(new Cat());
    }
}
```

输出：

```
dog对象是否属于Animal类：true

dog对象是否属于Dog类：true

dog对象是否属于Cat类：false

我是一只狗

汪汪...

我是一只猫

我喜欢吃鱼
```

19.匿名内部类

> 匿名内部类 这里指的是实例化内部对象 就是没有名字的内部类；
>
> 作用：假如某个类只使用一次，则可以使用匿名内部类；

上代码：

```
public class Test {

    public void test(A a){
        a.a();
    }

    public static void main(String[] args) {
        Test t=new Test();
        t.test(new B());

        // 匿名内部类
        t.test(new A(){

            @Override
            public void a() {
                System.out.println("匿名内部类，一次性使用");
            }

        });
    }
```

这里我们直接new 接口名字即可 然后写上实现方法 之所以说匿名内部类，我们这里压根没有定义类似A a=new B()的a对象；

> 1.接口

```
public interface A {

    public void a();
}
```

> 2.具体实现类

```
public class B implements A{

    @Override
    public void a() {
        System.out.println("a方法");
    }

}
```

> 测试类：

```
public class Test {

    public void test(A a){
        a.a();
    }

    public static void main(String[] args) {
        Test t=new Test();
        t.test(new B());


    }
}
```

运行输出：

```
a方法
```

20. 包装类

> 每个基本类型都有一个对应的类；就是所谓的包装类；

序号	基本类型	包装类
1	int	Integer
2	char	Character
3	short	Short
4	long	Long
5	float	Float
6	double	Double
7	boolean	Boolean
8	byte	Byte

> 1，装箱和拆箱
>
> 基本类型和类类型可以相互转换；
>
> 基本类型到类类型的转换叫做装箱；
>
> 类类型到基本类型的转换叫做拆箱；

20.1 拆箱和装箱

上代码：

```java
package com.java1234.chap03.sec17;

public class Demo1 {

    public static void main(String[] args) {
        int a=1;
        Integer i=new Integer(a); // 装箱
        int b=i.intValue(); // 拆箱
        System.out.println("a="+a);
        System.out.println("i="+i);
        System.out.println("b="+b);
    }
}
```

运行输出：

```
a=1

i=1

b=1
```

20.2 自动拆箱和装箱

> 在类类型和基本类型的转换中，是自动转换的 无需强制类型转换；

上代码：

```java
package com.java1234.chap03.sec17;

public class Demo2 {

    public static void main(String[] args) {
        Integer i=1; // 自动装箱的过程 自动把基本类型转换成类类型
        int i2=i; // 自动拆箱的过程 自动把类类型转成基本类型
        System.out.println("i="+i);
        System.out.println("i2="+i2);
    }
}
```

输出：

```
i=1

i2=1
```

20.3 包装类的作用

> 包装类的作用:因为包装类是类类型 所有jdk里提供了很多有用的方法给我们用；例如比如从用户界面来两个字符串数据a,b 然后我们程序里要进行相加运算。这时候包装类就派上用场了，我们可以用包装类的方法类进行类型转换。

上代码：

```
public class Demo3 {

    public static void main(String[] args) {
        String a="3";
        String b="5";
        // 调用Integer类的valuesOf方法 把字符串类型转换成int类型W
        int m=Integer.valueOf(a);
        int n=Integer.valueOf(b);
        System.out.println("a+b="+(m+n));
    }
}
```

运行输出：

```
a+b=8
```

21 单例模式

> 在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在；
>
> 有两种实现，一种是饿汉式，一种是懒汉式；

21.1 饿汉式

```
public class Singleton {

    /**
     * 构造方法私有
     */
    private Singleton(){

    }

    /**
     * 饿汉式单例实现
     */
    private static final Singleton single=new Singleton();

    /**
     * 获取实例
     */
    public static Singleton getInstance(){
        return single;
    }
}
```

构造方法私有 这样就保证了在外部是无法来实例化对象；先在内部定义一个静态常量对象，然后再写一个static方法 来返回这个对象，这样就保证是一个对象了；

测试类：

```
public class Test {

    public static void main(String[] args) {
        Singleton singleton1=Singleton.getInstance();
        Singleton singleton2=Singleton.getInstance();
        System.out.println(singleton1==singleton2);
    }
}
```

运行输出：

```
true
```

“==”来判断对象是否指向同一个内存区域

21.2懒汉式

```
public class Singleton2 {

    /**
     * 构造方法私有
     */
    private Singleton2(){

    }

    /**
     * 懒汉式单例实现  在第一次调用的时候实例化
     */
    private static Singleton2 single;

    /**
     * 获取实例
     */
    public synchronized static Singleton2 getInstance(){
        if(single==null){
            System.out.println("第一次调用的实例化");
            single=new Singleton2();
        }
        return single;
    }
}
```

> 饿汉式是先定义实例的 而懒汉式是先定义引用，当第一次调用getInstance的时候 进行对象实例化操作；
>
> 当然这里我们考虑到多并发的情况 多个线程同时调用这个方法的时候，会出现问题，所以我们加了同步锁 synchronized 来保证
>
> 同一时刻只有一个线程进入方法；

> 测试类

```
public class Test {

    public static void main(String[] args) {
        Singleton singleton1=Singleton.getInstance();
        Singleton singleton2=Singleton.getInstance();
        System.out.println("饿汉式："+(singleton1==singleton2));

        Singleton2 singleton3=Singleton2.getInstance();
        Singleton2 singleton4=Singleton2.getInstance();
        System.out.println("懒汉式："+(singleton3==singleton4));

    }
}
```

22 异常处理

> 程序在执行过程中，出现意外，我们专业属于叫做出现了异常；

上代码:

```
public class ExceptionDemo {

    public static void main(String[] args) {
        String str="123a";
        try{
            int a=Integer.parseInt(str);
        }catch(NumberFormatException e){
            e.printStackTrace();
        }
        System.out.println("继续执行");
    }
}
```

> throws表示当前方法不处理异常，而是交给方法的调用出去处理；
>
> throw表示直接抛出一个异常；

22.1 throws和throws关键字

> 1.throws

```
public class Demo1 {

    /**
     * 把异常向外面抛
     * @throws NumberFormatException
     */
    public static void testThrows()throws NumberFormatException{
        String str="123a";
        int a=Integer.parseInt(str);
        System.out.println(a);
    }

    public static void main(String[] args) {
        try{
            testThrows();
```

```
            System.out.println("here");
        }catch(Exception e){
            System.out.println("我们在这里处理异常");
            e.printStackTrace();
        }
        System.out.println("I'm here");
    }
}
```

运行输出：

```
我们在这里处理异常
java.lang.NumberFormatException: For input string: "123a"
    at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at com.java1234.chap04.sec03.Demo1.testThrows(Demo1.java:11)
    at com.java1234.chap04.sec03.Demo1.main(Demo1.java:17)
I'm here
```

> 2.throw

```
public class Demo2 {

    public static void testThrow(int a) throws Exception{
        if(a==1){
            // 直接抛出一个异常类
            throw new Exception("有异常");
        }
        System.out.println(a);
    }

    public static void main(String[] args) {
        try {
            testThrow(1);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

运行输出：

```
java.lang.Exception: 有异常
    at com.java1234.chap04.sec03.Demo2.testThrow(Demo2.java:8)
    at com.java1234.chap04.sec03.Demo2.main(Demo2.java:15)
```

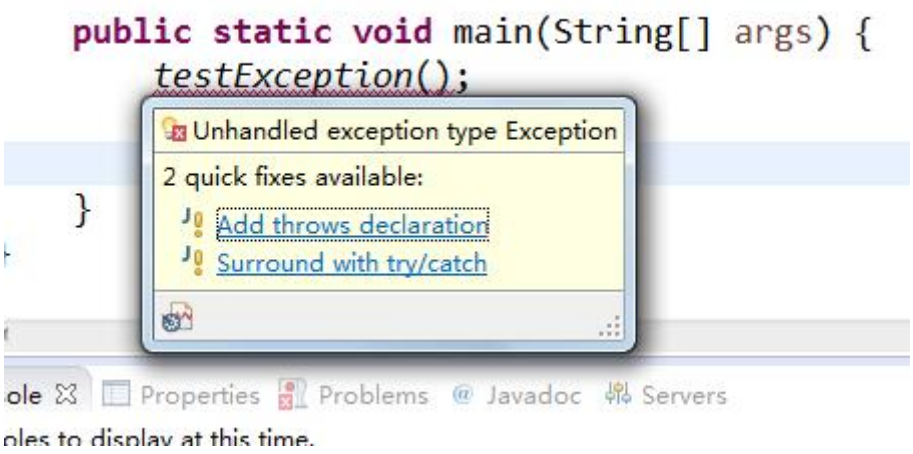
22.2 RuntimeException和Exception

> Exception是检查型异常，在程序中必须使用try...catch进行处理；
>
> RuntimeException是非检查型异常，例如NumberFormatException，可以不使用try...catch进行处理，

> 但是如果产生异常，则异常将由JVM进行处理；
>
> RuntimeException最好也用try...catch捕获；

```
public class Demo1 {

    /**
     * 运行时异常，编译时不检查，可以不使用try...catch捕获
     * @throws RuntimeException
     */
    public static void testRuntimeException()throws RuntimeException{
        throw new RuntimeException("运行时异常");
    }

    /**
     * Exception异常，编译时会检查，必须使用try...catch捕获
     * @throws Exception
     */
    public static void testException()throws Exception{
        throw new Exception("Exception异常");
    }

    public static void main(String[] args) {
        testException();

        testRuntimeException();
    }
}
```

我们会发现 testException()报错了 编译不通过 但是testRuntimeException()就编译通过了。



我们对testException()加下try...catch即可；

```
public class Demo1 {

    /**
     * 运行时异常，编译时不检查，可以不使用try...catch捕获
     * @throws RuntimeException
     */
    public static void testRuntimeException()throws RuntimeException{
        throw new RuntimeException("运行时异常");
    }
```

```
    /**
     * Exception异常，编译时会检查，必须使用try...catch捕获
     * @throws Exception
     */
    public static void testException()throws Exception{
        throw new Exception("Exception异常");
    }

    public static void main(String[] args) {
        try {
            testException();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        testRuntimeException();
    }
}
```

22.3 自定义异常

> 1.自定义异常

```
/**
 * 自定义异常，继承自Exception
 * @author user
 *
 */
public class CustomException extends Exception{

    public CustomException(String message) {
        super(message);
    }

}
```

> 2.测试类

```
public class TestCustomException {

    public static void test()throws CustomException{
        throw new CustomException("自定义异常");
    }

    public static void main(String[] args) {
        try {
            test();
        } catch (CustomException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

运行输出：

```
com.java1234.chap04.sec05.CustomException: 自定义异常
at.com.java1234.chap04.sec05.TestCustomException.test(TestCustomException.java:6
)
at.com.java1234.chap04.sec05.TestCustomException.main(TestCustomException.java:1
1)
```

22.4 java常用类

> 1.Date类
>
> Date类是jdk给我们提高的标准日期类，在java.util包下；

上代码：

```
import java.util.Date;

public class TestDate {

    public static void main(String[] args) {
        Date date=new Date();
        System.out.println("当前日期："+date);
    }
}
```

运行输出：

```
当前日期：Wed Nov 02 16:54:01 CST 2016
```

> 2.Calendar类
>
> Calendar是日历类，也是java.util包下的，功能比较强大，能获取到年月日时分秒的具体值；

上代码：

```
import java.util.Calendar;

public class TestCalendar {

    public static void main(String[] args) {
        Calendar calendar=Calendar.getInstance();
        System.out.println(calendar.get(Calendar.YEAR));
        System.out.println(calendar.get(Calendar.MONTH)+1); // 月份从0开始 要+1

        System.out.println("现在是："+calendar.get(Calendar.YEAR)+"年"
                +(calendar.get(Calendar.MONTH)+1)+"月"
                +calendar.get(Calendar.DAY_OF_MONTH)+"日"
                +calendar.get(Calendar.HOUR_OF_DAY)+"时"
                +calendar.get(Calendar.MINUTE)+"分"
                +calendar.get(Calendar.SECOND)+"秒");
    }
}
```

运行输出:

```
2016
11
现在是：2016年11月2日17时15分17秒
```

> 3.SimpleDateFormat类
>
> SimpleDateFormat类主要是用作日期类型转换用的，在java.text包下：

我们写个示例，把日期对象和日期字符串相互转换：

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

public class TestSimpleDateFormat {

    /**
     * 将日期对象格式化为指定格式的日期字符串
     * @param date 传入的日期对象
     * @param format 格式
     * @return
     */
    public static String formatDate(Date date,String format){
        String result="";
        SimpleDateFormat sdf=new SimpleDateFormat(format);
        if(date!=null){
            result=sdf.format(date);
        }
        return result;
    }

    /**
     * 将日期字符串转换成一个日期对象
     * @param dateStr 日期字符串
     * @param format 格式
     * @return
     * @throws ParseException
     */
    public static Date formatDate(String dateStr,String format) throws
ParseException{
        SimpleDateFormat sdf=new SimpleDateFormat(format);
        return sdf.parse(dateStr);
    }

    public static void main(String[] args) throws ParseException {
        Date date=new Date();
        //将日期对象格式化为指定格式的日期字符串
        System.out.println(formatDate(date,"yyyy-MM-dd"));
        System.out.println(formatDate(date,"yyyy-MM-dd HH:mm:ss"));
        System.out.println(formatDate(date,"yyyy年MM月dd日HH时mm分ss秒"));
        //将日期字符串转换成一个日期对象
        String dataStr="1989-11-02 18:01:41";
        Date date2=formatDate(dataStr,"yyyy-MM-dd HH:mm:ss");
        System.out.println(formatDate(date2,"yyyy-MM-dd HH:mm:ss"));
    }
}
```

运行输出：

```
2016-11-02
2016-11-02 18:06:50
2016年11月02日18时06分50秒
1989-11-02 18:01:41
```

> 思路 首先查看api 我们会发现Calendar有个DAY_OF_WEEK 可以返回一个星期中的第几天；这里说下注意点 老外的第一天是从星期日开始的，所以要-1；

```
import java.util.Calendar;

public class Test {

    public static void main(String[] args) {
        String[] weekDays = {"星期日", "星期一", "星期二", "星期三", "星期四", "星期五", "星期六"};
        Calendar calendar=Calendar.getInstance();
        System.out.println("今天是"+weekDays[calendar.get(Calendar.DAY_OF_WEEK)-1]);
    }
```

23 String和StringBuffer类

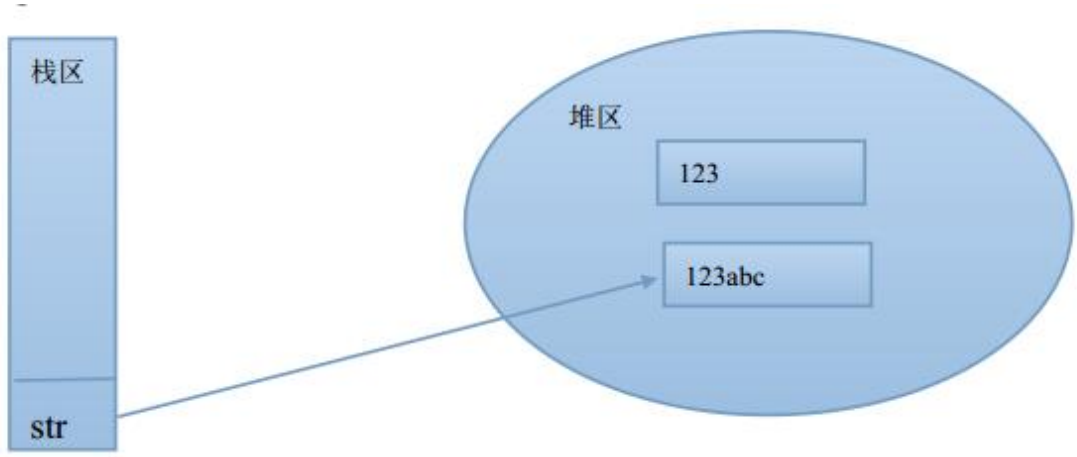
> String：对String类型的对象操作，等同于重新生成一个新对象，然后讲引用指向它；
>
> StringBuffer：对StringBuffer类型的对象操作，操作的始终是同一个对象；

上代码：

```
public class TestString {

    public static void main(String[] args) {
        String str="123";
        str+="abc";
        System.out.println(str);
    }
}
```

运行输出：

```
123abc
```



str原先指向的是123 通过+= 重新指向了123abc；

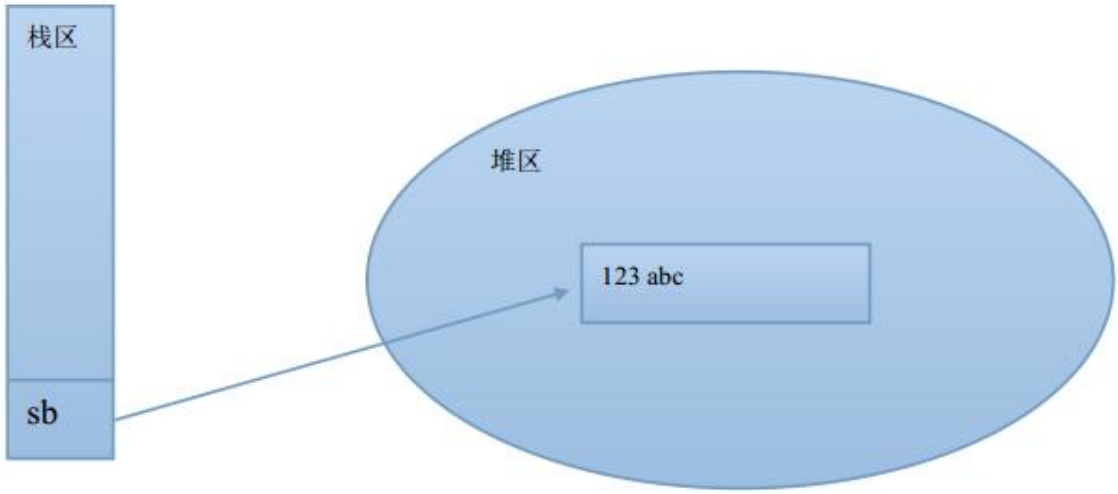
上代码：

```
public class TestStringBuffer {

    public static void main(String[] args) {
        StringBuffer sb=new StringBuffer("123");
        sb.append("abc");
        System.out.println(sb.toString());
    }
}
```

运行输出：

```
123abc
```

这两个实例的内部运行有本质区别的：



总结下：假如定义的字符串内容基本不变或者很少变化，用String效率高；假如定义的字符串内容经常变动，要用StringBuffer；

24.Math类

> 1，max方法 求最大值；
>
> 2，min方法 求最小值；
>
> 3，round方法 四舍五入；
>
> 4，pow方法 求次幂；

```
public class TestMath {

    public static void main(String[] args) {
        System.out.println("最大值："+Math.max(1, 2));
        System.out.println("最小值："+Math.min(1, 2));
        System.out.println("四舍五入："+Math.round(4.5));
        System.out.println("3的4次方："+Math.pow(3, 4));
    }
}
```

运行输出：

```
最大值：2
最小值：1
四舍五入：5
3的4次方：81.0
```

25.Arrays类

> Arrays类主要是封装了很多操作数组的工具方法，方便开发者直接调用；

1，toString()方法 返回指定数组内容的字符串表示形式；

2，sort()方法 对指定的类型数组按数字升序进行排序；

3，binarySearch()方法 使用二分搜索法来搜索指定类型数组，以获取指定值；

4，fill()方法，将指定类型值分配给指定类型数组的每个元素；

```
import java.util.Arrays;

public class TestArrays {

    public static void main(String[] args) {
        int arr[]={1,7,3,8,2};
        System.out.println(arr);
        System.out.println("以字符串形式输出数组："+Arrays.toString(arr));
        Arrays.sort(arr); // 给数组排序
        System.out.println("排序后的数组："+Arrays.toString(arr));
        System.out.println(Arrays.binarySearch(arr, 1));
        Arrays.fill(arr, 0); // 将指定内容填充到数组中
        System.out.println("填充数组后的字符串："+Arrays.toString(arr));
    }
}
```

运行输出：

```
[I@15db9742
以字符串形式输出数组：[1, 7, 3, 8, 2]
排序后的数组：[1, 2, 3, 7, 8]
0
填充数组后的字符串：[0, 0, 0, 0, 0]
```

26.java泛型

> 定义：使用泛型可以指代任意对象类型；

> 1.定义一个C1类：里面有个Integer属性 给出构造方法 以及打印类型 和get set方法

```
public class C1 {

    private Integer a;



    public C1(Integer a) {
        super();
        this.a = a;
    }

    public Integer getA() {
        return a;
    }

    public void setA(Integer a) {
        this.a = a;
    }

    /**
     * 打印a的类型
     */
    public void print(){
        System.out.println("a的类型是："+a.getClass().getName());
    }

}
```

> 定义一个C2类，里面定义一个String类型属性：

```
public class C2 {

    private String a;

    public String getA() {
        return a;
    }

    public void setA(String a) {
        this.a = a;
    }

    public C2(String a) {
```

```
        super();
        this.a = a;
    }

    /**
     * 打印a的类型
     */
    public void print(){
        System.out.println("a的类型是："+a.getClass().getName());
    }
}
```

测试类：

```
public class Test1 {

    public static void main(String[] args) {
        // begin test c1
        C1 c1=new C1(1);
        c1.print();
        int i=c1.getA();
        System.out.println("i="+i);
        // end test c1

        // begin test c2
        C2 c2=new C2("Hi");
        c2.print();
        String s1=c2.getA();
        System.out.println("s1="+s1);
        // end test c2


    }
}
```

运行输出：

```
a的类型是：java.lang.Integer
i=1
a的类型是：java.lang.String
s1=Hi
```

> 所有类都继承自Object类，所以直接定义成Object类型的属性；

```
public class C12 {

    private Object object;

    public Object getObject() {
        return object;
    }

    public void setObject(Object object) {
        this.object = object;
    }
```

```
    public C12(Object object) {
        super();
        this.object = object;
    }

    /**
     * 打印object的类型
     */
    public void print(){
        System.out.println("object的类型是："+object.getClass().getName());
    }
}
```

测试类：

```
public class Test1 {

    public static void main(String[] args) {
        // begin test c1
        C1 c1=new C1(1);
        c1.print();
        int i=c1.getA();
        System.out.println("i="+i);
        // end test c1

        // begin test c2
        C2 c2=new C2("Hi");
        c2.print();
        String s1=c2.getA();
        System.out.println("s1="+s1);
        // end test c2

        // begin test c12
        C12 c12=new C12(1); // 向上转型
        c12.print();
        int i12=(Integer) c12.getObject(); // 向下转型
        System.out.println("i12="+i12);

        C12 c122=new C12("你好");// 向上转型
        c122.print();
        String s122=(String) c122.getObject();   // 向下转型
        System.out.println("s122="+s122);
        // end test c12

    }
}
```

运行输出：

```
a的类型是：java.lang.Integer
i=1
a的类型是：java.lang.String
s1=Hi
object的类型是：java.lang.Integer
i12=1
object的类型是：java.lang.String
s122=你好
```

测试类里需要转型，类简便了，但是测试类复杂了，有没有一种类简单，测试也简单的方式呢，这时候，泛型诞生了；

```
/**
 * 定义泛型类
 * @author caofeng
 *
 * @param <T>
 */
public class CC<T>{

    private T ob;


    public CC(T ob) {
        super();
        this.ob = ob;
    }

    public T getOb() {
        return ob;
    }

    public void setOb(T ob) {
        this.ob = ob;
    }

    /**
     * 打印T的类型
     */
    public void print(){
        System.out.println("T的实际类型是："+ob.getClass().getName());
    }

}
```

用指代任意类型，当然也可以用其他字母，但是一般用T，Type的意思；

> 测试类

```
public class Test1 {

    public static void main(String[] args) {
        // begin test c1
        C1 c1=new C1(1);
        c1.print();
```

```
        int i=c1.getA();
        System.out.println("i="+i);
        // end test c1

        // begin test c2
        C2 c2=new C2("Hi");
        c2.print();
        String s1=c2.getA();
        System.out.println("s1="+s1);
        // end test c2

        // begin test c12
        C12 c12=new C12(1); // 向上转型
        c12.print();
        int i12=(Integer) c12.getObject(); // 向下转型
        System.out.println("i12="+i12);

        C12 c122=new C12("你好");// 向上转型
        c122.print();
        String s122=(String) c122.getObject();    // 向下转型
        System.out.println("s122="+s122);
        // end test c12

        // begin test CC
        CC<Integer> cc=new CC<Integer>(1);
        cc.print();
        int icc=cc.getOb();
        System.out.println("icc="+icc);

        CC<String> cc2=new CC<String>("我是泛型，好简单啊");
        cc2.print();
        String icc2=cc2.getOb();
        System.out.println("icc2="+icc2);
        // end test CC
    }
}
```

运行输出：

```
a的类型是：java.lang.Integer
i=1
a的类型是：java.lang.String
s1=Hi
object的类型是：java.lang.Integer
i12=1
object的类型是：java.lang.String
s122=你好
T的实际类型是：java.lang.Integer
icc=1
T的实际类型是：java.lang.String
icc2=我是泛型，好简单啊
```

25.1 限制泛型类

```
public class Test {
```

```
    public static void main(String[] args) {
        Demo<Dog> demo=new Demo<Dog>(new Dog());
        Dog dog=demo.getOb();
        dog.print();


        Demo<Cat> demo2=new Demo<Cat>(new Cat());
        Cat cat=demo2.getOb();
        cat.print();

        Demo<Animal> demo3=new Demo<Animal>(new Animal());
    }
}
```

运行输出：

```
Dog

Cat
```

> 测试类

```
public class Test {

    public static void main(String[] args) {
        Demo<Dog> demo=new Demo<Dog>(new Dog());
        Dog dog=demo.getOb();
        dog.print();


        Demo<Cat> demo2=new Demo<Cat>(new Cat());
        Cat cat=demo2.getOb();
        cat.print();

        Demo<Animal> demo3=new Demo<Animal>(new Animal());
    }
}
```

运行输出：

```
Dog
Cat
```

```
    public static void main(String[] args) {
        Demo<Dog> demo=new Demo<Dog>(new Dog());
        Dog dog=demo.getOb();
        dog.print();


        Demo<Cat> demo2=new Demo<Cat>(new Cat());
        Cat cat=demo2.getOb();
        cat.print();

        Demo<Animal> demo3=new Demo<Animal>(new Animal());

        Demo<Integer> demo4=new Demo<Integer>(1);
    }
```

Bound mismatch: The type Integer is not a valid substitute for the bounded parameter <T extends Animal> of the type Demo<T>

nsole Properti

nated> Test (23) [Java Application] C:\jdk1.8\bin\javaw.exe (2016年11月15日 上午10:33:20)

25.2 java泛型-通配符泛型

> 通配符泛型在使用泛型 特殊的场景下用到，比如把泛型对象作为方法参数传入方法的时候，就用到通配符泛型；

```
import com.java1234.chap06.sec02.Animal;
import com.java1234.chap06.sec02.Cat;
import com.java1234.chap06.sec02.Demo;
import com.java1234.chap06.sec02.Dog;

public class Test {

    /**
     * 通配符泛型
     * @param a
     */
    private static void take(Demo<?> a){
        a.print();
    }

    public static void main(String[] args) {
        Demo<Dog> dog=new Demo<Dog>(new Dog());
        take(dog);
        Demo<Cat> cat=new Demo<Cat>(new Cat());
        take(cat);
        Demo<Animal> animal=new Demo<Animal>(new Animal());
        take(animal);
    }
}
```

25.3 泛型方法

> 泛型方法指返回值和参数都用泛型表示的方法；

```
public class Test {
```

```
    /**
     * 泛型方法
     * @param t
     */
    //注意<T> 不是返回值，此处的返回值是void ，此处的<T> 表示传入参数有泛型,<T>存在的作
//用，是为了保证参数中能够出现T这种数据类型。
    public static <T> void f(T t){
        System.out.println("T的类型是："+t.getClass().getName());
    }

    public static void main(String[] args) {
        f("");
        f(1);
        f(1.0f);
        f(new Object());
    }
}
```

运行输出：

```
T的类型是：java.lang.String
T的类型是：java.lang.Integer
T的类型是：java.lang.Float
T的类型是：java.lang.Object
```

27.反射

> 如果知道一个类，那可定可以通过这个类创建对象；
>
> 但是如果要求通过一个对象找到一个类，这时候反射就派上用场了。
>
> java反射实现的核心就是Class类 java.lang包下的。

27.1 Class类

> Class类获取对象完整包类名：

> 1.新建Student类

```
package com.java1234.chap07.sec01;

public class Student {

}
```

> 2.新建测试类

```
package com.java1234.chap07.sec01;

public class Test01 {

    public static void main(String[] args) {
        Student student=new Student();
        System.out.println(student.getClass().getName());
    }
}
```

运行输出：

```
com.java1234.chap07.sec01.Student
```

这里对象.getClass() 调用的是Object类的getClass() 得到Class对象 然后再调用Class里的getName()方法，获取到完整包路径类；

> 3.通过完整包路径类型来实例化Class对象：

```
package com.java1234.chap07.sec01;

public class Test02 {

    public static void main(String[] args) {
        try {
            Class<?> c=Class.forName("com.java1234.chap07.sec01.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

运行输出：

```
com.java1234.chap07.sec01.Student
```

通过得到Class对象，我们可以通过Class对象得到com.java1234.chap07.sec01.Student类的完整结构；

> 4.Class基本使用

> 4.1新建Student类

```
package com.java1234.chap07.sec01;

public class Student {

    private String name;
    private Integer age;
```

```
    public Student(String name, Integer age) {
        super();
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + "]";
    }


}
```

> 4.2 测试代码

```
package com.java1234.chap07.sec01;

import java.lang.reflect.Constructor;

public class Test04 {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.sec01.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Student s=null;
        Constructor<?>[] cons=c.getConstructors();
        try {
            //通过getConstructors()方法获取所有构造方法，测试类：
            s=(Student) cons[0].newInstance("小锋",28);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println(s);
    }
}
```

运行输出：

```
com.java1234.chap07.sec01.Student
Student [name=小锋, age=28]
```

28.通过反射获取类的基本结构

> 1.新建Student类

```
package com.java1234.chap07.sec02;

public class Student {

    private String name;
    private Integer age;

    public Student(String name) {
        super();
        this.name = name;
    }

    public Student(String name, Integer age) {
        super();
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }

    public void say(){
        System.out.println("我的姓名："+name);
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + "]";
    }


}
```

> 2.通过getConstructors()方法获取所有构造方法，测试类：

```
package com.java1234.chap07.sec02;

import java.lang.reflect.Constructor;
```

```
public class Test1 {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.sec02.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Constructor<?> cons[]=c.getConstructors();
        for(Constructor<?> con:cons){
            System.out.println("构造方法："+con);
        }
    }
}
```

运行输出：

```
com.java1234.chap07.sec02.Student
构造方法：public com.java1234.chap07.sec02.Student(java.lang.String)
构造方法：public
com.java1234.chap07.sec02.Student(java.lang.String,java.lang.Integer)
```

> 3.通过getMethods()方法获取所有方法，测试类：

```
package com.java1234.chap07.sec02;

import java.lang.reflect.Method;

public class Test2 {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.sec02.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Method mds[]=c.getMethods();
        for(Method m:mds){
            System.out.println(m);
        }
    }
}
```

运行输出：

```
com.java1234.chap07.sec02.Student

public java.lang.String com.java1234.chap07.sec02.Student.toString()
```

```
public java.lang.String com.java1234.chap07.sec02.Student.getName()

public void com.java1234.chap07.sec02.Student.setName(java.lang.String)

public java.lang.Integer com.java1234.chap07.sec02.Student.getAge()

public void com.java1234.chap07.sec02.Student.setAge(java.lang.Integer)

public void com.java1234.chap07.sec02.Student.say()

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException

public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()
```

> 4.通过getDeclaredFields()方法获取所有属性，测试类：

```
package com.java1234.chap07.sec02;

import java.lang.reflect.Field;

public class Test3 {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.sec02.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Field fs[]=c.getDeclaredFields();
        for(Field f:fs){
            System.out.println(f);
        }
    }
}
```

运行输出：

```
com.java1234.chap07.sec02.Student

private java.lang.String com.java1234.chap07.sec02.Student.name

private java.lang.Integer com.java1234.chap07.sec02.Student.age
```

28.1 通过反射调用方法和操作属性

> 1.新建Student类

```
package com.java1234.chap07.sec04;

public class Student {

    private String name;
    private Integer age;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }

    public void say(){
        System.out.println("我的姓名："+name);
    }

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

> 2.通过反射调用方法，主要通过invoke方法，我们给下演示代码：

```
package com.java1234.chap07.sec04;

import java.lang.reflect.Method;

public class Test1 {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.sec04.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }

        try {
            Object obj=c.newInstance();
            Method m2=obj.getClass().getMethod("setName", String.class);
            m2.invoke(obj, "小锋");
            Method m=obj.getClass().getMethod("getName");
            String name=(String) m.invoke(obj);
            System.out.println("name="+name);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

> 3.通过反射操作属性，java里反射可以操作私有属性，只需要设置下，我们给下演示代码：

```
package com.java1234.chap07.sec04;

import java.lang.reflect.Field;

public class Test2 {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.sec04.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        try {
            Object obj=c.newInstance();
            Field nameField=c.getDeclaredField("name");
            nameField.setAccessible(true);
            nameField.set(obj, "小锋");
            System.out.println("name="+nameField.get(obj));
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

运行输出：

```
com.java1234.chap07.sec04.Student
name=小锋
```

28.2 通过反射获取类所实现的所有接口和获取父类

> 1.新建Student类

```
package com.java1234.chap07;

public class Student extends A implements B,C{


}
```

> A类

```
package com.java1234.chap07;

public class A {

}
```

> B，C类

```
package com.java1234.chap07;

public interface B {

}


public interface C {

}
```

> 2.获得接口测试类

```
package com.java1234.chap07;

public class Test {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Class<?> []ifs= c.getInterfaces();
        for(Class<?> i:ifs){
            System.out.println(i);
        }
    }
}
```

运行输出：

```
com.java1234.chap07.Student
interface com.java1234.chap07.B
interface com.java1234.chap07.C
```

> 3.获得父类测试类

```
package com.java1234.chap07;

public class Test2 {

    public static void main(String[] args) {
        Class<?> c=null;
        try {
            c=Class.forName("com.java1234.chap07.Student");
            System.out.println(c.getName());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Class<?> s=c.getSuperclass();
        System.out.println(s);
    }
}
```

运行输出：

```
com.java1234.chap07.Student
class com.java1234.chap07.A
```

29 集合

> 1.Collection接口是集合的老祖宗，

> 2.List是Collection接口的子接口，也是最常用的接口，此接口对Collection接口进行了大量的扩展，List集合里的元素是可以重复的。

List接口的主要实现类有ArrayList，和LinkedList。在数据量不大的情况下，这两个类性能差别不大，一般情况下，集合里的元素很少变化的，一般用ArrayList，假如集合里元素经常变动，要用LinkedList；底层实现有差别的。

上代码：

> 1.ArrayList

```
package com.java1234.chap08.sec02;

import java.util.ArrayList;

public class TestArrayLit {

    private static void pringArrayList(ArrayList<String> arrayList){
        System.out.println("当前的集合元素：");
        for(int i=0;i<arrayList.size();i++){
            System.out.println(arrayList.get(i));
        }
```

```
    }

    public static void main(String[] args) {
        ArrayList<String> arrayList=new ArrayList<String>();
        // 添加元素
        arrayList.add("张三");
        arrayList.add("李四");
        pringArrayList(arrayList);
        // 在指定位置插入元素
        arrayList.add(1, "小张三");
        pringArrayList(arrayList);
        // 元素的替换
        arrayList.set(2, "小李四");
        pringArrayList(arrayList);
        // 移除元素
        arrayList.remove(0);
        pringArrayList(arrayList);
    }
}
```

运行输出：

```
当前的集合元素:
张三
李四
当前的集合元素:
张三
小张三
李四
当前的集合元素:
张三
小张三
小李四
当前的集合元素:
小张三
小李四
```

> 2.LinkedList

```
package com.java1234.chap08.sec02;

import java.util.LinkedList;

public class TestLinkedList {

    private static void pringLinkedList(LinkedList<String> linkedList){
        System.out.println("当前元素的集合：");
        for(int i=0;i<linkedList.size();i++){
            System.out.print(linkedList.get(i)+" ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        LinkedList<String> linkedList=new LinkedList<String>();
        linkedList.add("张三");
```

```
        linkedList.add("李四");
        linkedList.add("王五");
        linkedList.add("李四");
        linkedList.add("赵六");
        pringLinkedList(linkedList);

        // indexOf 寻找位置
        System.out.println(linkedList.indexOf("李四"));
        pringLinkedList(linkedList);

        // peekFirst 获取第一个元素
        System.out.println(linkedList.peekFirst());
        pringLinkedList(linkedList);

        // peekLast 获取最后一个元素
        System.out.println(linkedList.peekLast());
        pringLinkedList(linkedList);

        // pollFirst 摘取第一个元素
        System.out.println(linkedList.pollFirst());
        pringLinkedList(linkedList);

        // pollLast 榨取最后一个元素
        System.out.println(linkedList.pollLast());
        pringLinkedList(linkedList);
    }
}
```

运行输出：

```
当前元素的集合:
张三 李四 王五 李四 赵六
1
当前元素的集合:
张三 李四 王五 李四 赵六
张三
当前元素的集合:
张三 李四 王五 李四 赵六
赵六
当前元素的集合:
张三 李四 王五 李四 赵六
张三
当前元素的集合:
李四 王五 李四 赵六
赵六
当前元素的集合:
李四 王五 李四
```

29.1 集合遍历

> 1.Iterator遍历

新建Student类：

```
ackage com.java1234.chap08.sec03;
```

```
public class Student {

    private String name;
    private Integer age;



    public Student() {
        super();
        // TODO Auto-generated constructor stub
    }
    public Student(String name, Integer age) {
        super();
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
}
```

测试类：

```
package com.java1234.chap08.sec03;

import java.util.Iterator;
import java.util.LinkedList;


public class TestIterator {

    public static void main(String[] args) {
        LinkedList<Student> list=new LinkedList<Student>();
        list.add(new Student("张三",10));
        list.add(new Student("李四",20));
        list.add(new Student("王五",30));

        /**
         * 用Iterator遍历集合
         */
        Iterator<Student> it=list.iterator();  // 返回一个迭代器
        while(it.hasNext()){
            Student s=it.next();   // 返回迭代的下一个元素。
            System.out.println("姓名："+s.getName()+"年龄："+s.getAge());
        }
    }
}
```

运行输出：

```
姓名：张三年龄：10
姓名：李四年龄：20
姓名：王五年龄：30
```

29.2 foreach循环

```
package com.java1234.chap08.sec03;
import java.util.LinkedList;
public class TestForeach {
    public static void main(String[] args) {
        LinkedList<Student> list=new LinkedList<Student>();
        list.add(new Student("张三",10));
        list.add(new Student("李四",20));
        list.add(new Student("王五",30));
        /**
         * 用foreach遍历
         */
        for(Student s:list){
            System.out.println("姓名："+s.getName()+"年龄："+s.getAge());
        }
    }
```

运行输出：

```
姓名：张三年龄：10
姓名：李四年龄：20
姓名：王五年龄：30
```

30 set集合

> Set集合是Collection接口的子接口，没有对Collection接口进行扩展，里面不允许存在重复的内容；-实现类HashSet

```
package com.java1234.chap08.sec04;

import java.util.HashSet;
import java.util.Iterator;

public class TestHashSet {

    public static void main(String[] args) {
        /**
         * 1，HashSet是无序
         * 2，不允许有重复的值
         */
        HashSet<String> hs=new HashSet<String>();
        hs.add("21221");
        hs.add("112");
        hs.add("312");
        hs.add("421");
        hs.add("312");
```

```
        /**
         * 用Iterator遍历集合
         */
        Iterator<String> it=hs.iterator();
        while(it.hasNext()){
            String s=it.next();
            System.out.println(s+" ");
        }
    }
}
```

运行输出：

```
112
421
312
21221
```

31 Map集合

> - 是存放一对值的最大接口（即接口中的每一个元素都是一对，以key->value键值对的形式保存）
> - 我们这里讲解下Map的常用实现类HashMap；

```
package com.java1234.chap08.sec05;

import java.util.HashMap;
import java.util.Iterator;

public class TestHashMap {

    public static void main(String[] args) {
        HashMap<String,Student> hashMap=new HashMap<String,Student>();
        hashMap.put("1号", new Student("张三",10));
        hashMap.put("2号", new Student("李四",20));
        hashMap.put("3号", new Student("王五",30));

        // 通过key，获取value
        Student s=hashMap.get("1号");
        System.out.println(s.getName()+":"+s.getAge());

        Iterator<String> it=hashMap.keySet().iterator(); // 获取key的集合，再获取迭代器
        while(it.hasNext()){
            String key=it.next();  // 获取key
            Student student=hashMap.get(key);  // 通过key获取value
            System.out.println("key="+key+" value=["+student.getName()+","+student.getAge()+"]");
        }
    }
}
```

运行输出：

```
key=3号 value=[王五,30]
key=2号 value=[李四,20]
key=1号 value=[张三,10]
```

32.多线程

> 定义：同时对多项任务加以控制；程序里同时执行多个任务并且加以控制 这个是java多线程的含义。同时干多个事，能充分利用cpu 内存等硬件设备，提高程序运行效率。

> 1.新建Eat类和Music类

```
package com.java1234.chap09.sec01;

public class Eat extends Thread{

    @Override
    public void run() {
        for(int i=0;i<100;i++){
            try {
                Thread.sleep(100);
                System.out.println("吃饭");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.java1234.chap09.sec01;

public class Music extends Thread{

    @Override
    public void run() {
        for(int i=0;i<100;i++){
            try {
                Thread.sleep(100);
                System.out.println("听音乐");
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

}
```

> 2.测试类

```
package com.java1234.chap09.sec01;

public class Test02 {

    public static void main(String[] args) {
        Music musicThread=new Music();
        Eat eatThread=new Eat();
        musicThread.start();
        eatThread.start();
    }
}
```

运行输出：

```
吃饭
听音乐
.
.
.
吃饭
听音乐
.
吃饭
听音乐
```

32.1 多线程的实现

1.继承Thread类

```
package com.java1234.chap09.sec02;

public class Thread1 extends Thread{

    private int baoZi=1;

    private String threadName;

    public Thread1(String threadName) {
        super();
        this.threadName = threadName;
    }

    @Override
    public void run() {
        while(baoZi<=10){
            System.out.println(threadName+" 吃"+baoZi+"第个包子");
            baoZi++;
        }
    }

    public static void main(String[] args) {
        // 张三 李四一起吃包子 每人吃10个
        Thread1 t1=new Thread1("张三线程");
        Thread1 t2=new Thread1("李四线程");
        t1.start();
```

```
        t2.start();
    }
}
```

运行输出：

```
运行输出：
张三线程 吃1第个包子
张三线程 吃2第个包子
张三线程 吃3第个包子
张三线程 吃4第个包子
张三线程 吃5第个包子
张三线程 吃6第个包子
张三线程 吃7第个包子
张三线程 吃8第个包子
张三线程 吃9第个包子
张三线程 吃10第个包子
李四线程 吃1第个包子
李四线程 吃2第个包子
李四线程 吃3第个包子
李四线程 吃4第个包子
李四线程 吃5第个包子
李四线程 吃6第个包子
李四线程 吃7第个包子
李四线程 吃8第个包子
李四线程 吃9第个包子
李四线程 吃10第个包子
```

2.实现Runnable接口

```
package com.java1234.chap09.sec02;

public class Thread2 implements Runnable{

    private int baoZi=1;

    private String threadName;

    public Thread2(String threadName) {
        super();
        this.threadName = threadName;
    }

    @Override
    public void run() {
        while(baoZi<=10){
            System.out.println(threadName+" 吃"+baoZi+"第个包子");
            baoZi++;
        }
    }

    public static void main(String[] args) {
        // 张三 李四一起吃包子 每人吃10个
        Thread2 t1=new Thread2("张三线程");
        Thread2 t2=new Thread2("李四线程");
        Thread t11=new Thread(t1);
```

```
        Thread t12=new Thread(t2);
        t11.start();
        t12.start();
    }

}
```

运行输出：

```
张三线程 吃1第个包子

张三线程 吃2第个包子

李四线程 吃1第个包子

张三线程 吃3第个包子

李四线程 吃2第个包子

李四线程 吃3第个包子

李四线程 吃4第个包子

李四线程 吃5第个包子

李四线程 吃6第个包子

李四线程 吃7第个包子

李四线程 吃8第个包子

张三线程 吃4第个包子

李四线程 吃9第个包子

李四线程 吃10第个包子

张三线程 吃5第个包子

张三线程 吃6第个包子

张三线程 吃7第个包子

张三线程 吃8第个包子

张三线程 吃9第个包子

张三线程 吃10第个包子
```

3.多线程实现数据共享

```
package com.java1234.chap09.sec02;

public class Thread3 implements Runnable{

    private int baoZi=1;
```

```
    private String threadName;

    public Thread3(String threadName) {
        super();
        this.threadName = threadName;
    }

    @Override
    public synchronized void run() {
        while(baoZi<=10){
            System.out.println(threadName+" 吃"+baoZi+"第个包子");
            baoZi++;
        }
    }

    public static void main(String[] args) {
        Thread3 t1=new Thread3("超级张三线程");
        Thread t11=new Thread(t1);
        Thread t12=new Thread(t1);
        Thread t13=new Thread(t1);
        t11.start();
        t12.start();
        t13.start();
    }

}
```

output:

```
超级张三线程 吃1第个包子

超级张三线程 吃2第个包子

超级张三线程 吃3第个包子

超级张三线程 吃4第个包子

超级张三线程 吃5第个包子

超级张三线程 吃6第个包子

超级张三线程 吃7第个包子

超级张三线程 吃8第个包子

超级张三线程 吃9第个包子

超级张三线程 吃10第个包子
```

这里我们定义一个实例 然后用这个实例来是实例化三个Thread对象，run方法我们要加synchronized锁，否则会出现多个线程同时进入方法的情况，导致多个线程吃同一个包子；

32.2 多线程的状态

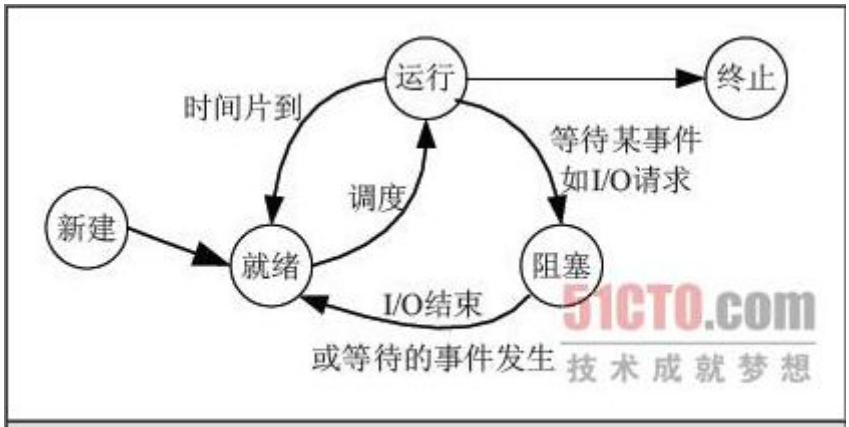


图2-1 进程状态转换图

1，创建状态

在程序中用构造方法创建了一个线程对象后，新的线程对象便处于新建状态，此时，它已经有了相应的内存空间和其他资源，但还处于不可运行状态。新建一个线程对象可采用Thread 类的构造方法来实现，例如，“Thread thread=new Thread();”。

2，就绪状态

新建线程对象后，调用该线程的start()方法就可以启动线程。当线程启动时，线程进入就绪状态。此时，线程将进入线程队列排队，等待CPU 服务，这表明它已经具备了运行条件。

3，运行状态

当就绪状态的线程被调用并获得处理器资源时，线程就进入了运行状态。此时，自动调用该线程对象的run()方法。run()方法定义了该线程的操作和功能。

4，堵塞状态

一个正在执行的线程在某些特殊情况下，如被人为挂起或需要执行耗时的输入/输出操作时，将让出CPU 并暂时中止自己的执行，进入堵塞状态。堵塞时，线程不能进入排队队列，只有当引起堵塞的原因被消除后，线程才可以转入就绪状态。

5，死亡状态

线程调用stop()方法时或run()方法执行结束后，即处于死亡状态。处于死亡状态的线程不具有继续运行的能力。

32.3 多线程的同步

> 我们发现 会有多个线程同时进入方法吃包子的情况发生，这时候，就引入了线程同步。可以给方法加同步，同一时刻，只允许一个线程进入方法；
>
> 关键字 synchronized

```
package com.java1234.chap09.sec05;

public class Thread3 implements Runnable{

    private int baoZi=1;

    private String threadName;

    public Thread3(String threadName) {
        super();
        this.threadName = threadName;
    }
```

```
    @Override
    public synchronized void run() {
        while(baoZi<=10){
            System.out.println(threadName+" 吃第"+baoZi+"个包子");
            baoZi++;
        }
    }

    public static void main(String[] args) {
        Thread3 t1=new Thread3("超级张三线程");

        Thread t11=new Thread(t1);
        Thread t12=new Thread(t1);
        Thread t13=new Thread(t1);

        t11.start();
        t12.start();
        t13.start();
    }

}
```

运行输出：

```
超级张三线程 吃第1个包子

超级张三线程 吃第2个包子

超级张三线程 吃第3个包子

超级张三线程 吃第4个包子

超级张三线程 吃第5个包子

超级张三线程 吃第6个包子

超级张三线程 吃第7个包子

超级张三线程 吃第8个包子

超级张三线程 吃第9个包子

超级张三线程 吃第10个包子
```

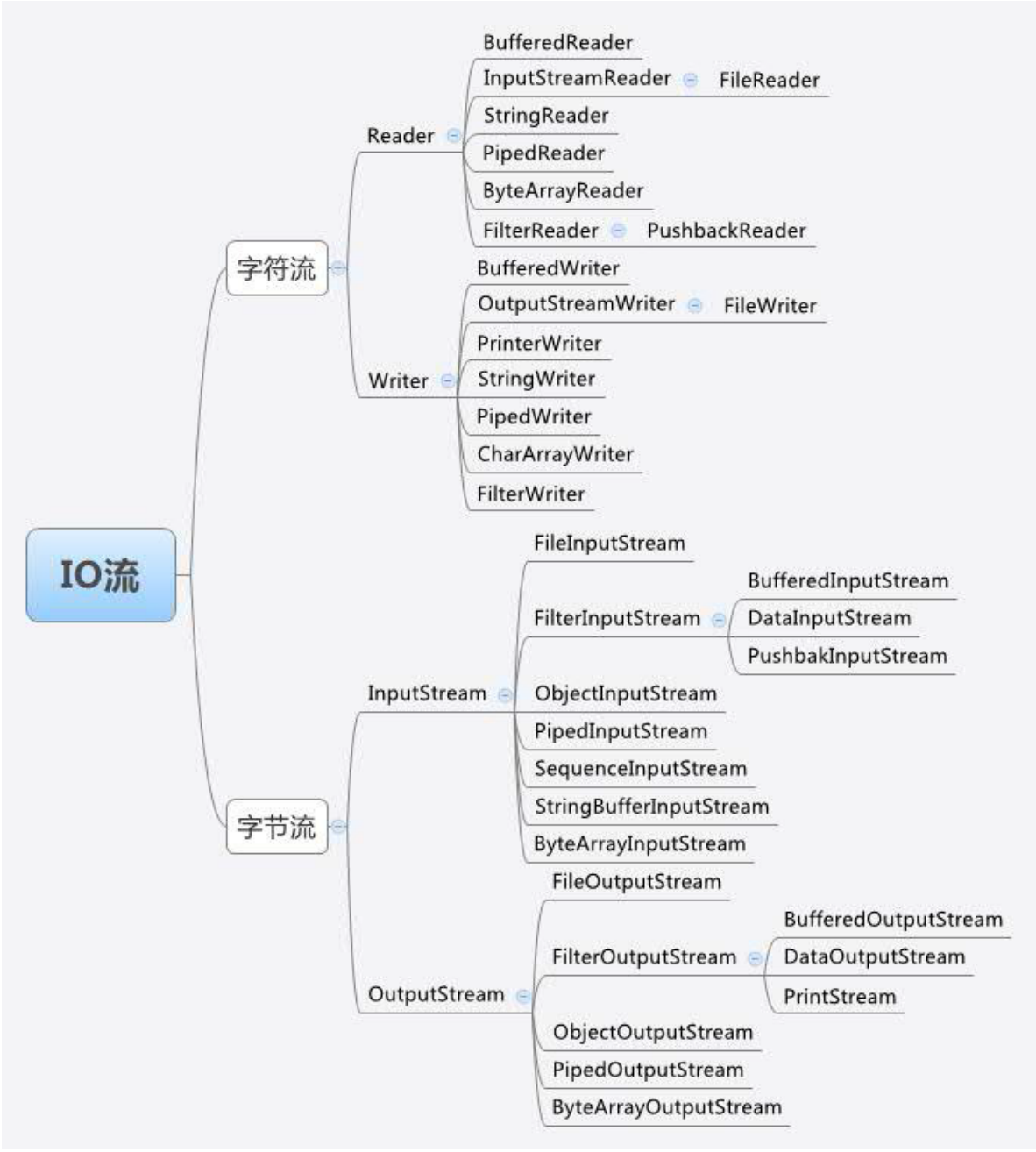
> 同步块

```
package com.java1234.chap09.sec05;

public class Thread4 implements Runnable{

    private int baoZi=1;

    private String threadName;

    public Thread4(String threadName) {
```

```
        super();
        this.threadName = threadName;
    }

    @Override
    public void run() {
        /**
         * 同步块
         */
        synchronized (this) {
            while(baoZi<=10){
                System.out.println(threadName+" 吃第"+baoZi+"个包子");
                baoZi++;
            }
        }
    }

    public static void main(String[] args) {
        Thread4 t1=new Thread4("超级张三线程");

        Thread t11=new Thread(t1);
        Thread t12=new Thread(t1);
        Thread t13=new Thread(t1);

        t11.start();
        t12.start();
        t13.start();
    }

}
```

33.IO流

> 定义：流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流，流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。



1.IO 流的分类

根据处理数据类型的不同分为：字符流和字节流

根据数据流向不同分为：输入流和输出流

2.文件操作File类

1，public boolean mkdir()

创建此抽象路径名指定的目录。

2，public boolean createNewFile() 创建一个文件

3，public boolean delete()

删除此抽象路径名表示的文件或目录。如果此路径名表示一个目录，则该目录必须为空才能删除。

4，public boolean exists()

测试此抽象路径名表示的文件或目录是否存在。

5，public File[] listFiles()

返回一个抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的文件。

6，public boolean isDirectory()

测试此抽象路径名表示的文件是否是一个目录。

> 实例一：创建文件目录和文件

```
package com.java1234.chap10;

import java.io.File;
import java.io.IOException;

public class Demo1 {

    public static void main(String[] args) throws IOException {
        File file=new File("c://java创建的目录");
        boolean b=file.mkdir();  // 创建虚拟目录
        if(b){
            System.out.println("目录创建成功！");
            file=new File("c://java创建的目录//java创建的文件.txt");
            boolean b2=file.createNewFile();  // 创建文件
            if(b2){
                System.out.println("文件创建成功！");
            }else{
                System.out.println("文件创建失败！");
            }
        }else{
            System.out.println("目录创建失败！");
        }
    }
}
```

运行输出：

```
目录创建成功！
文件创建成功！
```

> 实例二：删除文件和文件目录

```
package com.java1234.chap10;

import java.io.File;
import java.io.IOException;

public class Demo2 {

    public static void main(String[] args) throws IOException {
        File file=new File("c://java创建的目录//java创建的文件.txt");
        if(file.exists()){  // 假如文件存在
            boolean b=file.delete();  // 删除文件
            if(b){
                System.out.println("删除文件成功！");
            }else{
                System.out.println("删除文件失败！");
            }
        }
```

```
        file=new File("c://java创建的目录");
        if(file.exists()){
            boolean b=file.delete();  // 删除目录
            if(b){
                System.out.println("删除目录成功！");
            }else{
                System.out.println("删除目录失败！");
            }
        }
    }
}
```

运行输出：

```
删除文件成功!
删除目录成功!
```

> 实例三：遍历目录

```
package com.java1234.chap10;

import java.io.File;

public class Demo3 {

    public static void main(String[] args) {
        File file=new File("C://apache-cxf-3.1.5");
        File files[]=file.listFiles();  // 遍历目录
        for(int i=0;i<files.length;i++){
            System.out.println(files[i]);
        }
    }
}
```

> 实例四：递归遍历所有文件

```
package com.java1234.chap10;

import java.io.File;

public class Demo4 {

    /**
     * 打印文件
     * @param file
     */
    public static void listFile(File file){
        if(file!=null){
            if(file.isDirectory()){  // 是目录
                System.out.println(file);  // 打印下目录
                File f[]=file.listFiles();  // 遍历目录
                if(f!=null){
                    for(int i=0;i<f.length;i++){
                        listFile(f[i]);  // 递归调用
                    }
                }
```

```
            }else{    // 是文件
                System.out.println(file);  // 是文件，直接打印文件的路径
            }
        }
    }

    public static void main(String[] args) {
        File file=new File("C://apache-tomcat-7.0.63");
        listFile(file);
    }
}
```

3.InputStream和OutputStream

> nputStream是输入流 OutputStream是输出流；
>
> InputStream输入流可以把文件从硬盘读取到内存；
>
> OutputStream输出流可以把文件从内存写入到硬盘；

我们实际使用的都是InputStream和OutputStream的子类；

比如文件操作方面用的是FileInputStream和FileOutputStream；

> 1.定义了固定字节数组 一批读取

```
package com.java1234.chap10.sec03;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

public class Demo1 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        InputStream inputStream=new FileInputStream(file);  // 实例化
FileInputStream
        byte b[]=new byte[1024];
        int len=inputStream.read(b);
        inputStream.close(); // 关闭输入流
        System.out.println("读取的内容是："+new String(b,0,len));
    }
}
```

运行输出：

```
读取的内容是：我是人 www.java1234.com
```

> 2.获取文件长度，然后定义指定字节数组的长度；

```
package com.java1234.chap10.sec03;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
```

```
public class Demo2 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        InputStream inputStream=new FileInputStream(file);  // 实例化
FileInputStream
        int fileLength=(int)file.length();
        byte b[]=new byte[fileLength];
        inputStream.read(b);
        inputStream.close(); // 关闭输入流
        System.out.println("读取的内容是："+new String(b));
    }
}
```

> 3.一个字节一个字节读取

```
package com.java1234.chap10.sec03;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

public class Demo3 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        InputStream inputStream=new FileInputStream(file);  // 实例化
FileInputStream
        int fileLength=(int)file.length();
        byte b[]=new byte[fileLength];
        int temp=0;
        int len=0;
        while((temp=inputStream.read())!=-1){
            // 一个字节一个字节读取，放到b字节数组里
            b[len++]=(byte)temp;
        }
        inputStream.close(); // 关闭输入流
        System.out.println("读取的内容是："+new String(b));
    }
}
```

> 4.输出流

```
package com.java1234.chap10.sec03;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class Demo4 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        OutputStream out=new FileOutputStream(file);
        String str="你好，我好，大家好，Java好";
        byte b[]=str.getBytes();
```

```
        out.write(b); //  将b字节数组写入到输出流
        out.close();  // 关闭输出流
    }
}
```

> 5.输出流，追加

```
package com.java1234.chap10.sec03;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class Demo5 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        OutputStream out=new FileOutputStream(file,true);
        String str="你好，我好，大家好，Java好";
        byte b[]=str.getBytes();
        out.write(b); //  将b字节数组写入到输出流
        out.close();  // 关闭输出流
    }
}
```

4.BufferedInputStream和BufferedOutputStream

> 带缓冲的输入和输出流；

```
package com.java1234.chap10.sec03;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;

public class Demo6 {

    /**
     * 缓冲
     * @throws Exception
     */
    public static void bufferStream()throws Exception{
        // 定义了一个带缓冲的字节输入流
        BufferedInputStream bufferedInputStream=new BufferedInputStream(new
FileInputStream("C://《一头扎进J2SE》V2.0视频笔录2.doc"));
        // 定义了一个带缓冲的字节输出流
        BufferedOutputStream bufferedOutputStream=new BufferedOutputStream(new
FileOutputStream("C://复制的《一头扎进J2SE》V2.0视频笔录2.doc"));
        int b=0;
        long startTime=System.currentTimeMillis(); // 开始时间
        while((b=bufferedInputStream.read())!=-1){
            bufferedOutputStream.write(b);
        }
```

```
        bufferedInputStream.close();
        bufferedOutputStream.close();
        long endTime=System.currentTimeMillis();  // 结束时间
        System.out.println("缓冲花费的时间是："+(endTime-startTime));
    }

    /**
     * 非缓冲
     * @throws Exception
     */
    public static void stream() throws Exception{
        InputStream inputStream=new FileInputStream("C://《一头扎进J2SE》V2.0视频笔
录.doc");  // 定义一个输入流
        OutputStream outputStream=new FileOutputStream("C://复制的《一头扎进J2SE》
V2.0视频笔录.doc");
        int b=0;
        long startTime=System.currentTimeMillis(); // 开始时间
        while((b=inputStream.read())!=-1){
            outputStream.write(b);
        }
        inputStream.close();
        outputStream.close();
        long endTime=System.currentTimeMillis();  // 结束时间
        System.out.println("非缓冲花费的时间是："+(endTime-startTime));
    }

    public static void main(String[] args)throws Exception {
        stream();
        bufferStream();
    }
}
```

输出：

```
把文件从A地址复制到B地址，运行输出：

非缓冲花费的时间是：2368

缓冲花费的时间是：
```

5.Writer和Reader

> 主要用于文本的读取和写入，一般使用的实现类是FileReader和FileWriter；

> 1.直接读取

```
package com.java1234.chap10.sec04;

import java.io.File;
import java.io.FileReader;
import java.io.Reader;

public class Demo1 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
```

```
        Reader reader=new FileReader(file);
        char c[]=new char[1024]; // 字符数组
        int len=reader.read(c);
        reader.close();  // 关闭输入流
        System.out.println("读取的内容是："+new String(c,0,len));
    }
}
```

> 2.一个一个字符读取；

```
package com.java1234.chap10.sec04;

import java.io.File;
import java.io.FileReader;
import java.io.Reader;

public class Demo2 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        Reader reader=new FileReader(file);
        char c[]=new char[1024]; // 字符数组
        int temp=0;
        int len=0;
        while((temp=reader.read())!=-1){
            c[len++]=(char)temp;
        }
        reader.close();  // 关闭输入流
        System.out.println("读取的内容是："+new String(c,0,len));
    }
}
```

> 3.写入文件

```
package com.java1234.chap10.sec04;

import java.io.File;
import java.io.FileWriter;
import java.io.Writer;

public class Demo3 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        Writer out=new FileWriter(file);
        String str="我爱中华";
        out.write(str);  // 将字符串写入输出流
        out.close();  // 关闭输出流
    }
}
```

> 4.追加写入

```
package com.java1234.chap10.sec04;

import java.io.File;
```

```
import java.io.FileWriter;
import java.io.Writer;

public class Demo4 {

    public static void main(String[] args) throws Exception {
        File file=new File("C://测试文件.txt");
        Writer out=new FileWriter(file,true);
        String str="我爱中华2";
        out.write(str);  // 将字符串写入输出流
        out.close();  // 关闭输出流
    }
}
```

34.Eclispe配置

34.1 快捷键

- F6单步
- F8完成
- F5进入方法内部
- ctrl+shift+i:运行表达式的值

35.java301-重定向

```
HttpServletResponse response=(HttpServletResponse)servletResponse;
response.setStatus(301);
response.setHeader("Location","http://www.xxxxx.com/"); // 要重定向到网站
response.setHeader("Connection", "close");
```