

第1章 Java概述

1.1 Java语言发展历史 (记关键点)

Java诞生于SUN (Stanford University Network) , 09年SUN被Oracle (甲骨文) 收购。

Java之父是詹姆斯·高斯林(James Gosling)。

1996年发布JDK1.0版。大约26年。

目前最新的版本是Java17。我们学习的Java8。

1.2 Java语言特点 (后面需要关注和体会)

(1) 优点

- **面向对象:** Java语言支持封装、继承、多态，面向对象编程，让程序更好达到高内聚，低耦合的标准。
- **支持分布式:** Java语言支持Internet应用的开发，在基本的Java应用编程接口中有一个网络应用编程接口 (java net) ，它提供了用于网络应用编程的类库，包括URL、URLConnection、Socket、ServerSocket等。Java的RMI (远程方法激活) 机制也是开发分布式应用的重要手段。
- **健壮型:** Java的强类型机制、异常处理、垃圾的自动收集等是Java程序健壮性的重要保证。对指针的丢弃是Java的明智选择。
- **安全性高:** Java通常被用在网络环境中，为此，Java提供了一个安全机制以防恶意代码的攻击。如：安全防范机制 (类ClassLoader) ，如分配不同的名字空间以防替代本地的同名类、字节代码检查。
- **跨平台性:** Java程序 (后缀为java的文件) 在Java平台上被编译为体系结构中立的字节码格式 (后缀为class的文件) ，然后可以在实现这个Java平台的任何系统中运行。

(2) 缺点

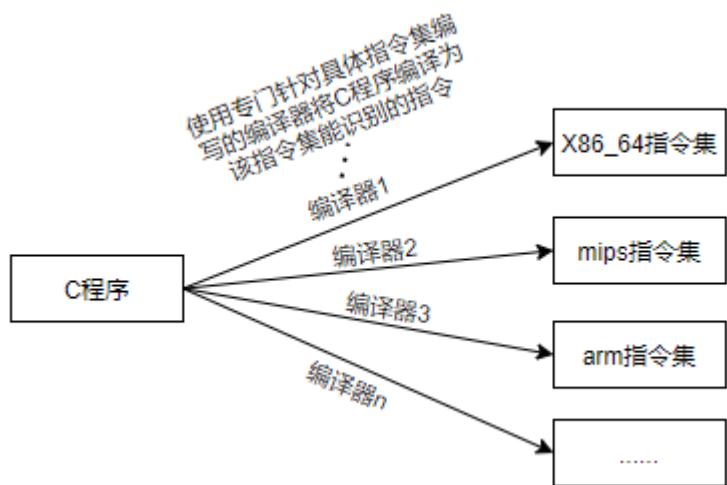
- 语法过于复杂，严谨，对程序员的约束比较多，与python和php等相比入门相对较难。但是一旦学会了，就业岗位需求量大，而且薪资待遇节节攀升。
- 一般适用于大型网站开发，整个架构会比较重，对于初创公司开发和维护人员的成本比较高 (即薪资高) ，选择用Java语言开发网站或应用系统的需要一定的经济实力。

1.3 Java语言跨平台原理 (理解)

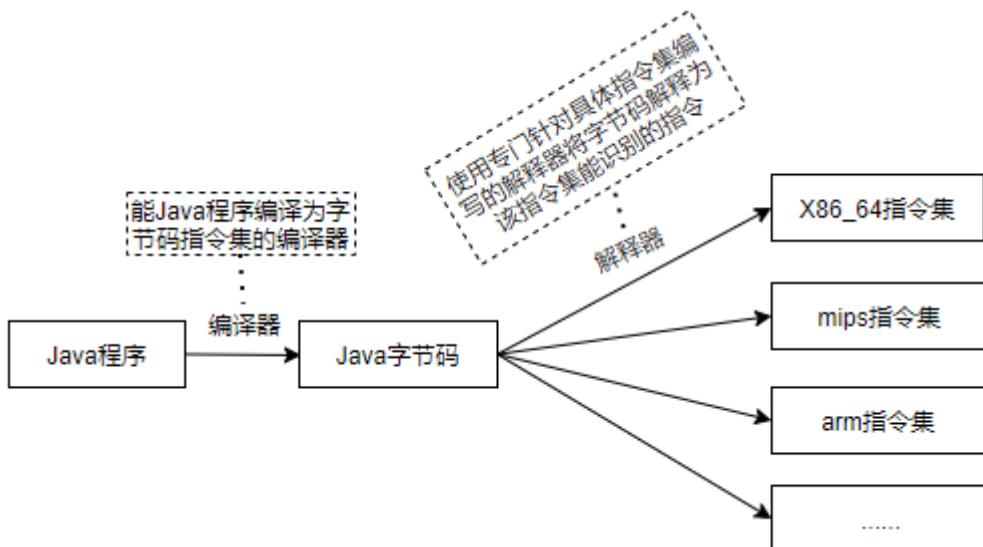
1、Java语言跨平台原理

很多时候，我们写的程序可能要在多个操作系统运行，这个时候就要求我们的程序需要在尽可能不改动的情况下完实现这个目标。不同的语言实现跨平台的方式不同。Java语言实现跨平台是建立在“虚拟机”基础之上的。

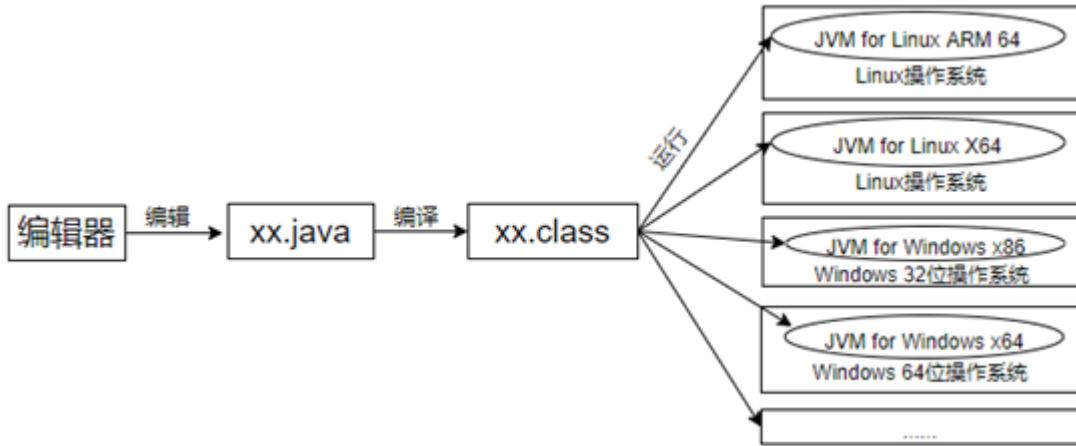
在Java出现之前，最为流行的编程语言是C和C++。如果我们想要在一台使用x86_64指令集的CPU的机器 (如个人PC) 上运行一个C语言程序，就需要编写一个将C语言翻译成x86_64汇编语言的编译器。如果想要在一台使用arm指令集的CPU的机器 (如苹果手机) 上，运行一个C语言程序，同样需要编写一个将C语言翻译成arm汇编语言的编译器。这样严重影响了C程序的跨平台性，因为针对特定的指令集开发编译器是一个难度非常大的工作。



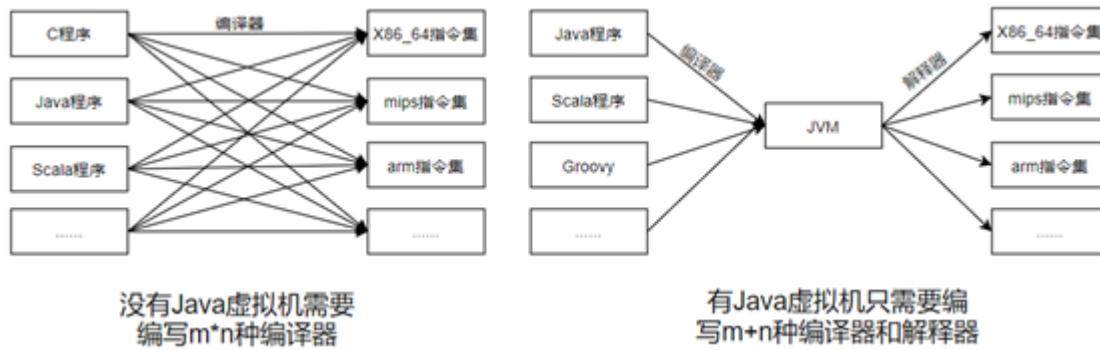
那么Java是如何解决这个问题的呢？Java设计了一套简洁的虚拟指令集，也就是字节码。如果我们想要在一台机器上运行Java程序，只需要将Java程序编译成字节码就可以了。编写一个将Java程序翻译成Java字节码的编译器，比起编写一个将Java程序翻译成x86_64指令集的编译器来说，要简单得多。可是这里产生了一个问题，难道我们的机器可以直接执行字节码这样的虚拟指令集吗？当然是不能的。我们需要针对不同的指令集，开发对应的字节码解释器。这个工作同样比较简单。



Java虚拟机 (JVM, Java Virtual Machine) 是由软件技术模拟出计算机运行的一个虚拟的计算机，它负责解释执行字节码指令集。也就是说，只要一台机器可以运行Java的虚拟机，那么就能运行Java语言编写的程序。而不同的平台，需要安装不同的Java虚拟机程序。那么我们编写完Java程序之后，需要先将.java的源文件编译为.class的字节码文件，然后在Java虚拟机中来执行这些字节码文件。



Java虚拟机的设计不仅仅解决了Java程序跨平台的问题，同时解决了很多语言的跨平台问题。



- .NET (C#, VB等语言) 也有虚拟机，也能实现跨平台，但是只能在Windows操作系统下安装.NET环境。
- C++不受虚拟机的限制，但是需要用不同平台的编译器重新编译一次。需要编写n个版本的编译器。
- Java虚拟机可以配置在MacOS, Windows (PC操作系统), Linux等上，但是不能配置在WP (Windows Phone), IOS (移动操作系统) 上，只能配置在android (移动操作系统)。Java程序可以实现一次编译处处运行。

| Linux | macOS | Windows | 不同平台 不同指令集 | File size | Download |
|---------------------------|-------|---------|---------------|-----------|--|
| Arm 64 Compressed Archive | | | | 170.95 MB | https://download.oracle.com/java/17/latest/jdk-17_linux-aarch64_bin.tar.gz (sha256) |
| Arm 64 RPM Package | | | | 153.12 MB | https://download.oracle.com/java/17/latest/jdk-17_linux-aarch64_bin.rpm (sha256) |
| x64 Compressed Archive | | | | 172.19 MB | https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.tar.gz (sha256) |

2、JVM、JRE、JDK的关系

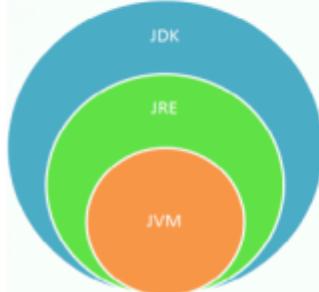
- **JVM** (Java Virtual Machine)：Java虚拟机，是运行所有Java程序的假想计算机，是Java程序的运行环境之一，也是Java最具吸引力的特性之一。我们编写的Java代码，都运行在**JVM**之上。
- **JRE** (Java Runtime Environment)：是Java程序的运行时环境，包含**JVM**和运行时所需要的核心类库。
- **JDK** (Java Development's Kit)：是Java程序开发工具包，包含**JRE**和开发人员使用的工具。

我们想要运行一个已有的Java程序，那么只需安装**JRE**即可。

我们想要开发一个全新的Java程序，那么必须安装**JDK**，其内部包含**JRE**。

| Java Language | | Java Language | | | | | | | | | |
|------------------------------|-----------------------------------|---------------|-----------------|-----------------------|-----------------------|----------------------|---------------------|--|--|--|--|
| Tools & Tool APIs | java | javac | javadoc | jar | javap | jdeps | Scripting | | | | |
| | Security | Monitoring | JConsole | VisualVM | JMC | JFR | | | | | |
| | JPDA | JVM TI | IDL | RMI | Java DB | Deployment | | | | | |
| | Internationalization | | | Web Services | | Troubleshooting | | | | | |
| Deployment | Java Web Start | | | | Applet / Java Plug-in | | | | | | |
| | JavaFX | | | | | | | | | | |
| User Interface Toolkits | Swing | | Java 2D | | AWT | Accessibility | | | | | |
| | Drag and Drop | | Input Methods | | Image I/O | Print Service | Sound | | | | |
| Integration Libraries | IDL | JDBC | JNDI | RMI | RMI-IIOP | | Scripting | | | | |
| | Beans | Security | | Serialization | | Extension Mechanism | | | | | |
| Other Base Libraries | JMX | XML JAXP | | Networking | | Override Mechanism | | | | | |
| | JNI | Date and Time | | Input/Output | | Internationalization | | | | | |
| Lang and util Base Libraries | lang and util | | | | | | | | | | |
| | Math | | Collections | | Ref Objects | | Regular Expressions | | | | |
| | Logging | Management | Instrumentation | Concurrency Utilities | | | | | | | |
| Java Virtual Machine | Reflection | Versioning | Preferences API | JAR | | Zip | | | | | |
| | Java HotSpot Client and Server VM | | | | | | | | | | |

◆ JVM、JRE、JDK

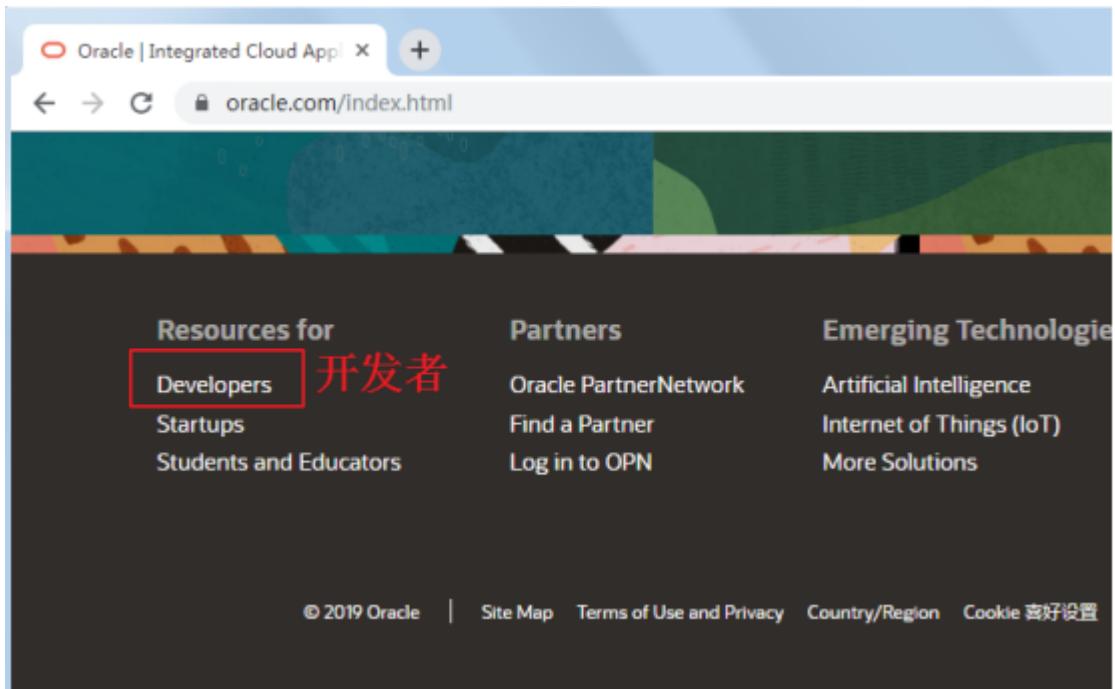


- JRE = JVM + Java SE 标准类库
- JDK = JRE + 开发工具集（例如Javac编译工具等）

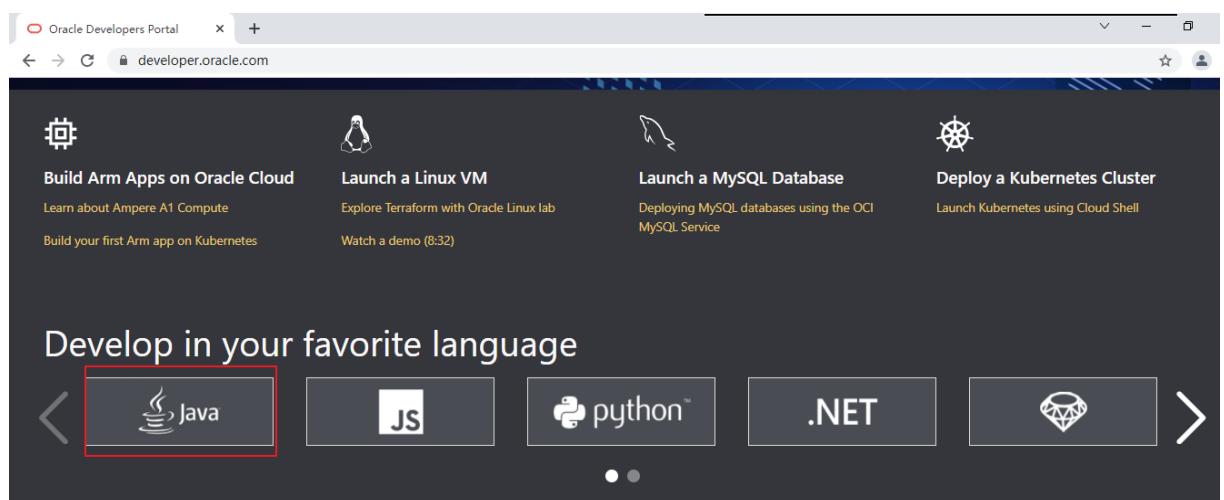
1.4 Java语言开发环境（会操作）

1、JDK的下载

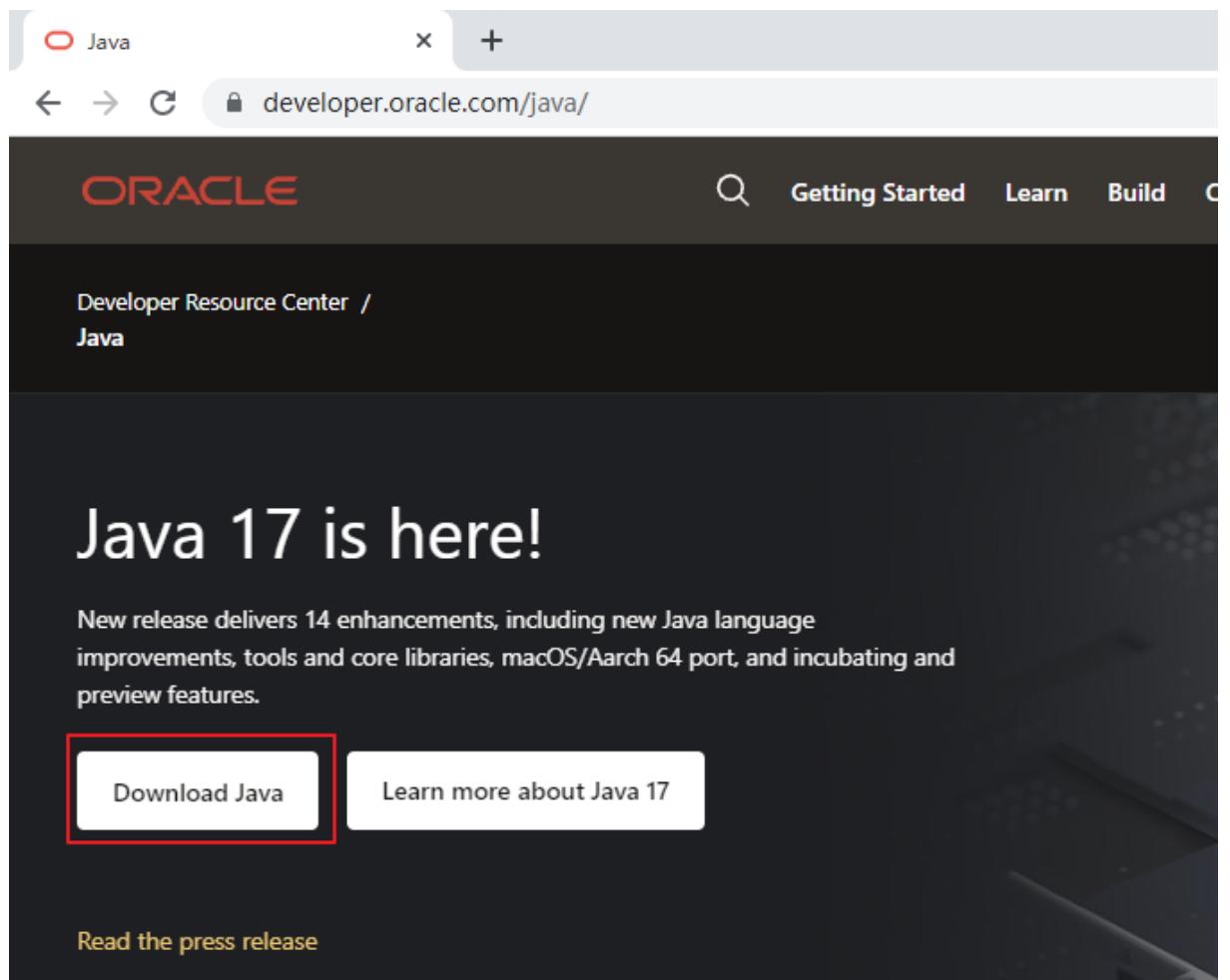
- 下载网址: www.oracle.com
- 下载步骤:
 - 登录Oracle公司官网, www.oracle.com, 如图所示: 在底部选择Developers开发者



- 在Developers页面中间的技术分类部分，选择Java，单击进入，如图所示：



展示的是最新Java版本，例如Java17。单击Download Java，然后选择具体的版本下载。



- 选择Download Java按钮后

The screenshot shows the Java Downloads page. At the top, it says 'Java 17 available now'. It provides information about Java 17 LTS being the latest long-term support release and being free to use in production. It also mentions that JDK 17 will receive updates until September 2024. A 'Learn about Java SE Subscription' button is visible. Below this, there's a section for 'Java SE Development Kit 17 downloads' with tabs for 'Documentation Download' (which is selected), 'Linux', 'macOS', and 'Windows'. A note says '最上面是Java17的下载，如果要下载别的版本，请往下拉'. A table lists download links for different Java editions:

| Product/file description | File size | Download |
|---------------------------|-----------|--|
| Arm 64 Compressed Archive | 170.95 MB | https://download.oracle.com/java/17/latest/jdk-17_linux-aarch64_bin.tar.gz (sha256) |
| Arm 64 RPM Package | 153.12 MB | https://download.oracle.com/java/17/latest/jdk-17_linux-aarch64_bin.rpm (sha256) |
| x64 Compressed Archive | 172.19 MB | https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.tar.gz (sha256) |

The screenshot shows the Oracle Java Downloads page. At the top, there are tabs for Java, Java Downloads | Oracle, and oracle.com/java/technologies/downloads/#java8-windows. The main navigation bar includes links for ORACLE, Products, Industries, Resources, Support, Events, Developer, View Accounts, and Contact Sales. A red box highlights the "Java SE subscribers have more choices" section. Below it, a note says "Also available for development, personal use, and to run other licensed Oracle products." A horizontal menu bar has three items: Java 16, Java 11, and Java 8, with Java 8 highlighted by a red box. The main content area is titled "Java SE Development Kit 8u301". It states that Java SE subscribers will receive JDK 8 updates until at least December of 2030. A note about the license change in April 2019 follows. It mentions that the Oracle Technology Network License Agreement for Oracle Java SE is substantially different from prior Oracle JDK 8 licenses. The page also notes that commercial license and support are available for a low cost with Java SE Subscription. It links to the Oracle Technology Network License Agreement for Oracle Java SE. A "JDK 8u301 checksum" link is also present. The download section for Windows is shown, with a "Documentation Download" button and a table for Windows. The table has columns for Product/file description, File size, and Download. It lists two options: x86 Installer (156.45 MB) and x64 Installer (169.46 MB). The x64 Installer download link is highlighted with a red box.

| Product/file description | File size | Download |
|--------------------------|-----------|--|
| x86 Installer | 156.45 MB | jdk-8u301-windows-i586.exe |
| x64 Installer | 169.46 MB | jdk-8u301-windows-x64.exe |

选择Accept License Agreement,

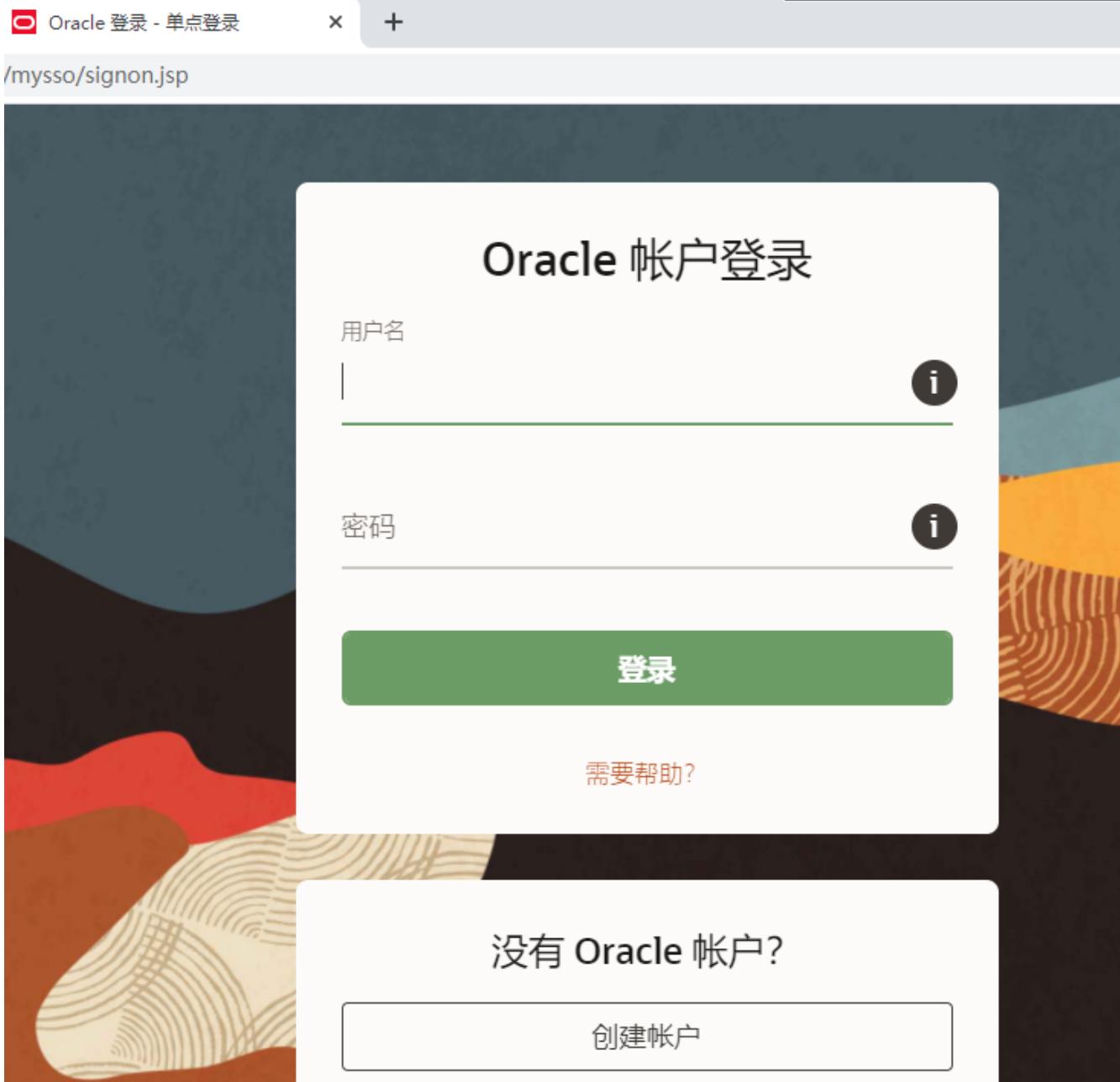
You must accept the [Oracle Technology Network License Agreement for Oracle Java SE](#) to download this software.

I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE
Required

You will be redirected to the login screen in order to download the file.

[Download jdk-8u301-windows-x64.exe](#)

注册或登录后下载



2、JDK的安装

- 安装步骤：
 - 双击 `jdk-8u202-windows-x64.exe` 文件，并单击 `下一步`，如图所示：



- 取消独立JRE的安装，单击公共JRE前的下拉列表，选择此功能将不可用如图所示：



- 修改安装路径，单击更改，如图所示：



- 将安装路径修改为 D:\develop\Java\jdk1.8.0_202\，并单击确定，如图所示：



- 单击下一步，如图所示：



- 稍后几秒，安装完成，如图所示：



- 目录结构，如图所示：

| 名称 | | 修改日期 | 类型 | 大小 |
|----------------------------------|-----------|----------------|-------------------|-----------|
| bin | 开发工具集 | 2017/2/7 10:12 | 文件夹 | |
| db | | 2017/2/7 10:12 | 文件夹 | |
| include | | 2017/2/7 10:12 | 文件夹 | |
| jre | JDK中包含JRE | 2017/2/7 10:12 | 文件夹 | |
| lib | | 2017/2/7 10:12 | 文件夹 | |
| COPYRIGHT | | 2015/6/8 18:22 | 文件 | 4 KB |
| javafx-src.zip | | 2017/2/7 10:12 | 好压 ZIP 压缩文件 | 5,053 KB |
| LICENSE | | 2017/2/7 10:12 | 文件 | 1 KB |
| README.html | | 2017/2/7 10:12 | Firefox HTML D... | 1 KB |
| release | | 2017/2/7 10:12 | 文件 | 1 KB |
| src.zip | 源代码压缩包 | 2015/6/8 18:22 | 好压 ZIP 压缩文件 | 20,745 KB |
| THIRDPARTYLICENSEREADME.txt | | 2017/2/7 10:12 | TXT 文件 | 175 KB |
| THIRDPARTYLICENSEREADME-JAVAF... | | 2017/2/7 10:12 | TXT 文件 | 108 KB |

3、配置环境变量

为什么配置path?

希望在命令行使用javac.exe等工具时，任意目录下都可以找到这个工具所在的目录。

例如：我们在C:\Users\Irene目录下使用java命令，结果如下：

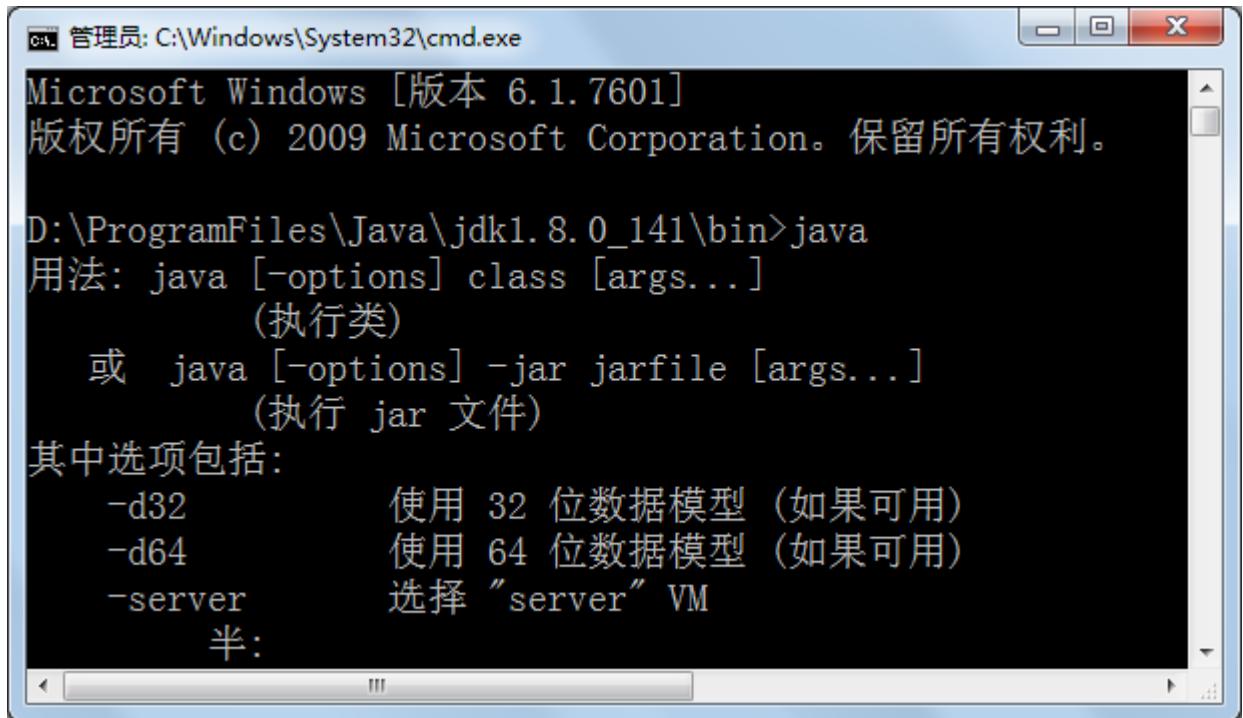


```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Irene>java
'java' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

C:\Users\Irene>
```

我们在JDK的安装目录的bin目录下使用java命令，结果如下：



管理员: C:\Windows\System32\cmd.exe

Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

D:\ProgramFiles\Java\jdk1.8.0_141\bin>java
用法: java [-options] class [args...]
 (执行类)
或 java [-options] -jar jarfile [args...]
 (执行 jar 文件)

其中选项包括:

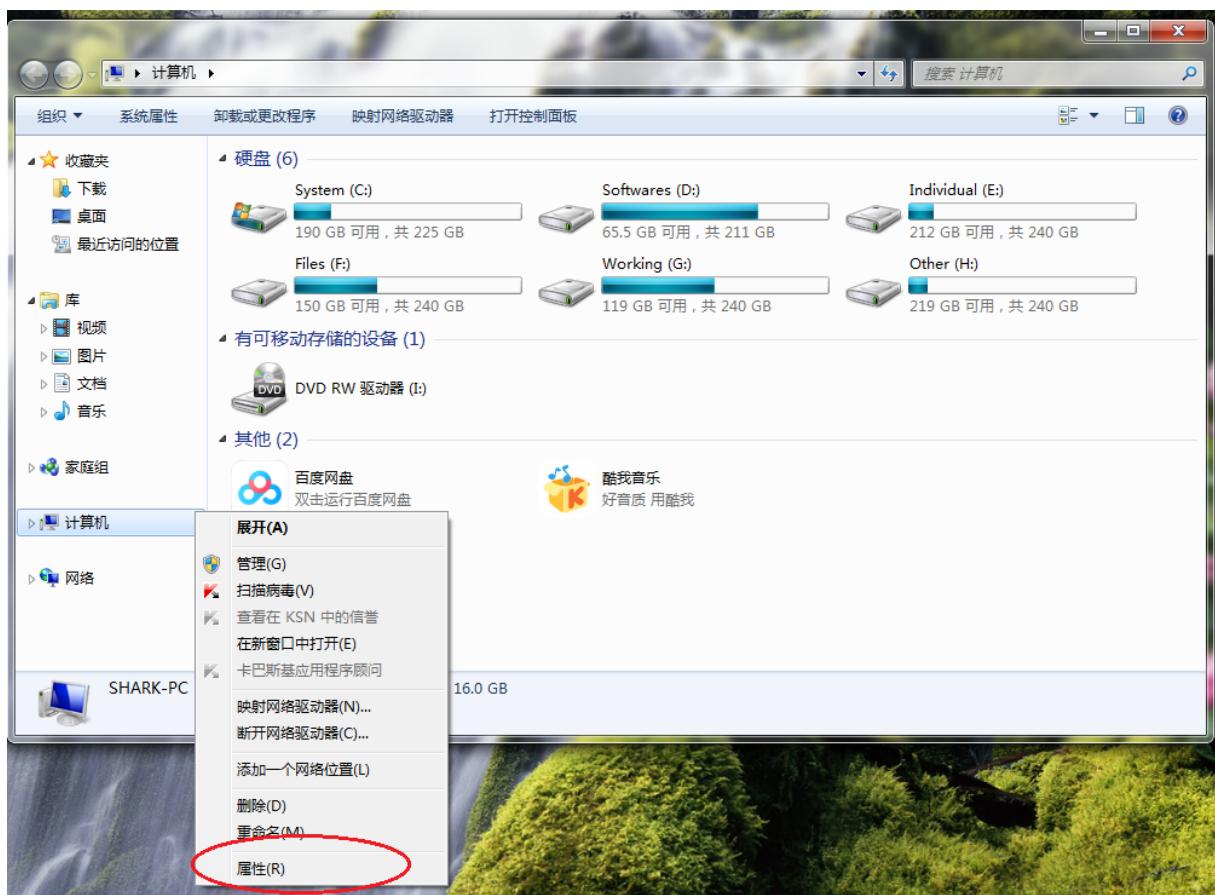
| | |
|---------|--------------------|
| -d32 | 使用 32 位数据模型 (如果可用) |
| -d64 | 使用 64 位数据模型 (如果可用) |
| -server | 选择 "server" VM |

半:

我们不可能每次使用java.exe, javac.exe等工具的时候都进入到DK的安装目录下，太麻烦了。我们希望在任意目录下都可以使用JDK的bin目录的开发工具，因此我们需要告诉操作系统去哪里找这些开发工具，这就需要配置path环境变量。

方案一：只配置path

- 步骤：
 - 打开桌面上的计算机，进入后在左侧找到 计算机，单击鼠标右键，选择 属性，如图所示：



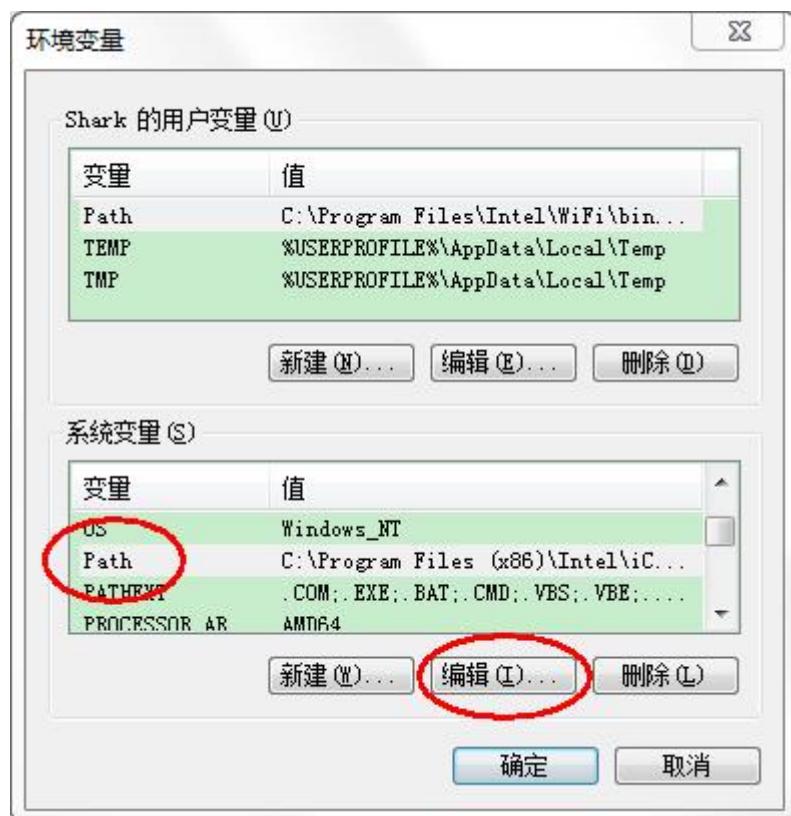
- 选择 **高级系统设置**，如图所示：



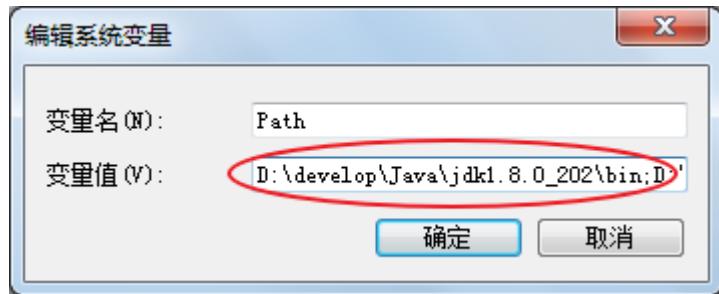
- 在高级选项卡，单击 **环境变量**，如图所示：



- 在系统变量中，选中 Path 环境变量，双击或者点击编辑，如图所示：



- 在变量值的最前面，键入 d:\develop\Java\jdk1.8.0_202\bin; 分号必须要写，而且还要是英文符号。如图所示：



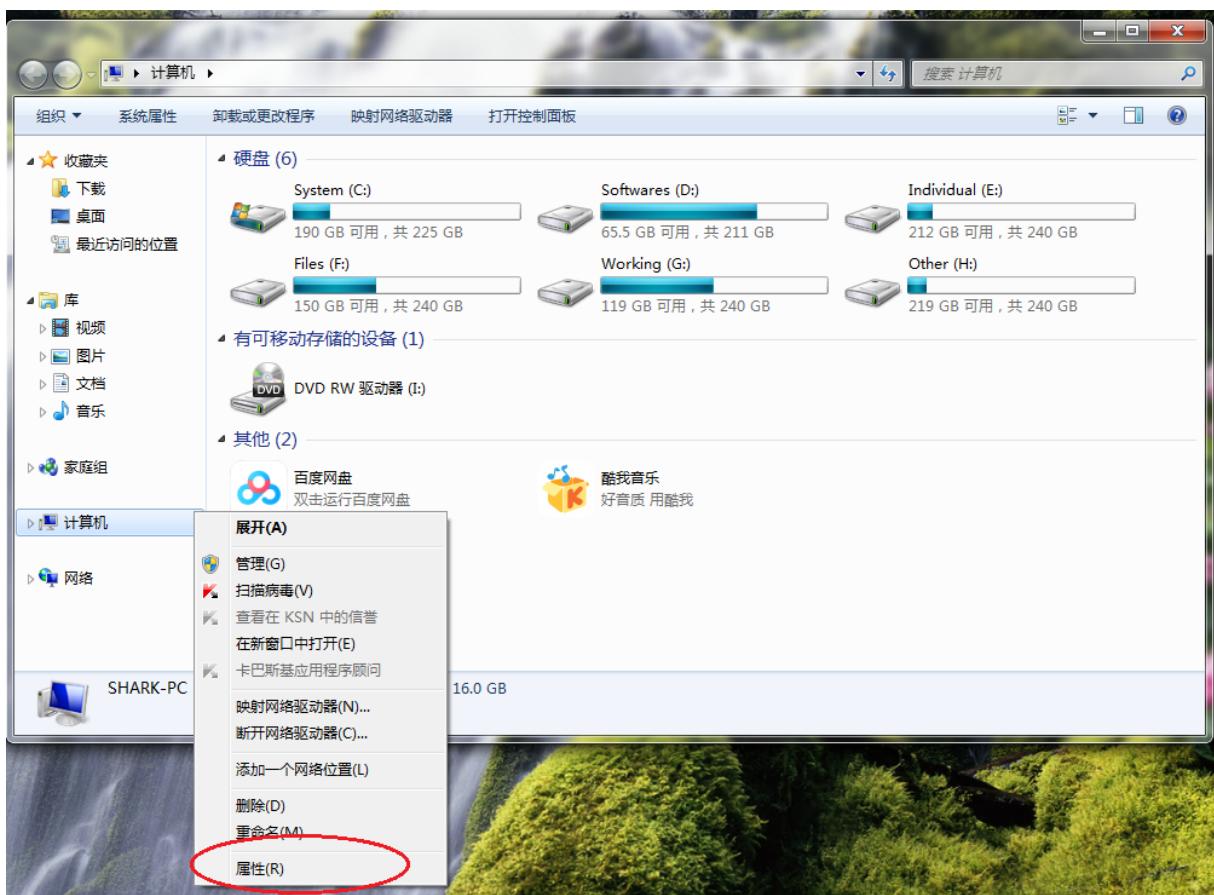
- 环境变量配置完成，**重新开启**DOS命令行，在任意目录下输入javac命令，运行成功。

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 © 2009 Microsoft Corporation。保留所有权利。

C:\Users\Shark>javac
用法: javac <options> <source files>
其中, 可能的选项包括:
-g                                     生成所有调试信息
-g:none                                不生成任何调试信息
-g:{lines,vars,source}                  只生成某些调试信息
-nowarn                                不生成任何警告
-verbose                               输出有关编译器正在执行的操作的消息
-deprecation                          输出使用已过时的 API 的源位置
-classpath <路径>                   指定查找用户类文件和注释处理程序的位置
-cp <路径>                            指定查找用户类文件和注释处理程序的位置
-sourcepath <路径>                  指定查找输入源文件的位置
-bootclasspath <路径>                覆盖引导类文件的位置
-extdirs <目录>                      覆盖所安装扩展的位置
-endorseddirs <目录>                覆盖签名的标准路径的位置
-proc:{none,only}                     控制是否执行注释处理和/或编译。
-processor <class1>[,<class2>,<class3>...] 要运行的注释处理程序的名称; 绕过默认的搜索进程
-processorpath <路径>                指定查找注释处理程序的位置
-parameters                           生成元数据以用于方法参数的反射
-d <目录>                            指定放置生成的类文件的位置
-s <目录>                            指定放置生成的源文件的位置
```

方案二：配置JAVA_HOME+path

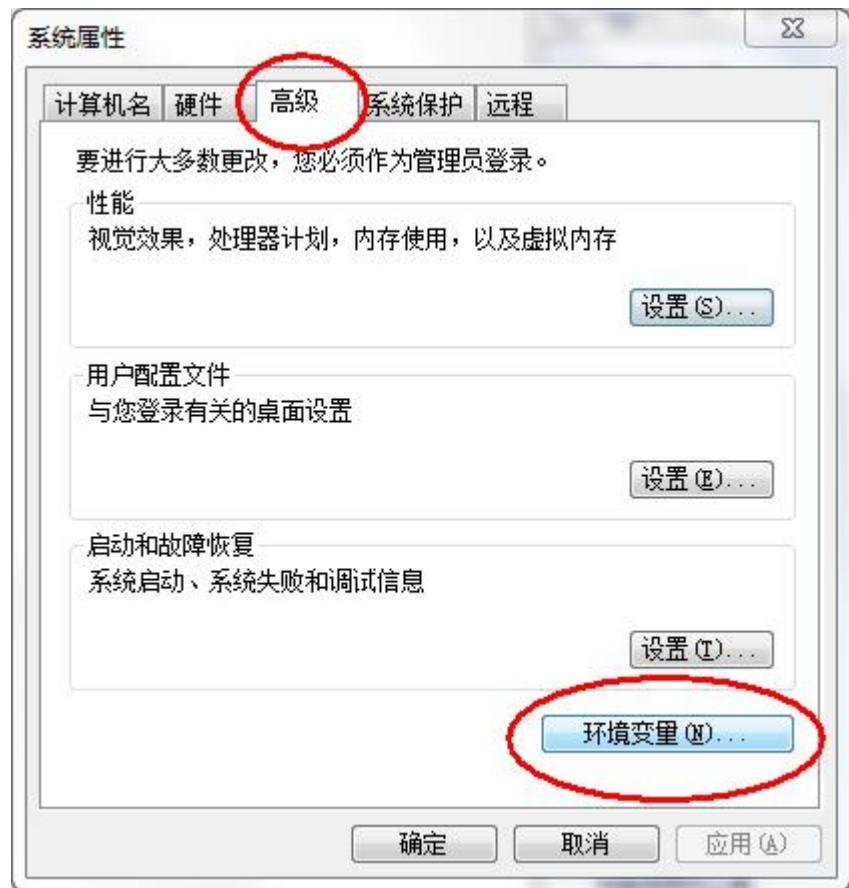
- 步骤：
 - 打开桌面上的计算机，进入后在左侧找到计算机，单击鼠标右键，选择属性，如图所示：



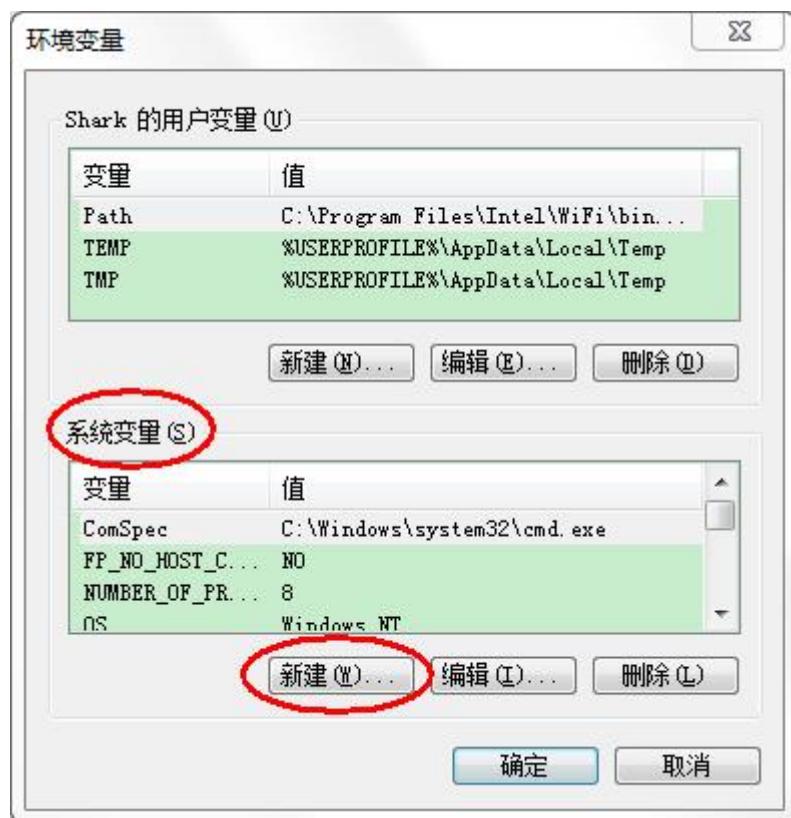
- 选择 **高级系统设置**，如图所示：



- 在高级选项卡，单击 **环境变量**，如图所示：



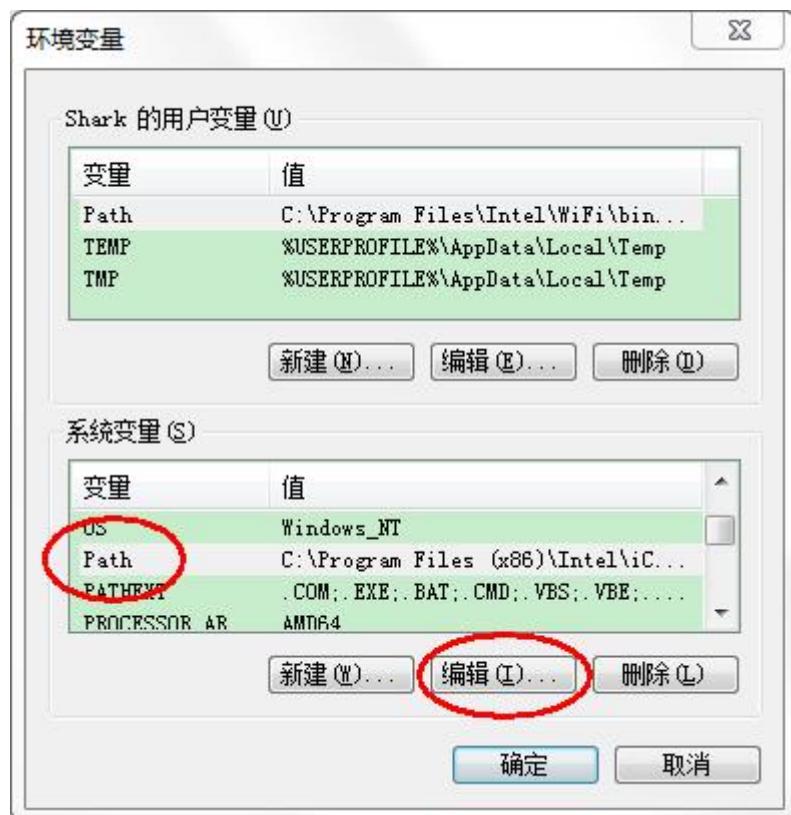
- 在系统变量中，单击新建，创建新的环境变量，如图所示：



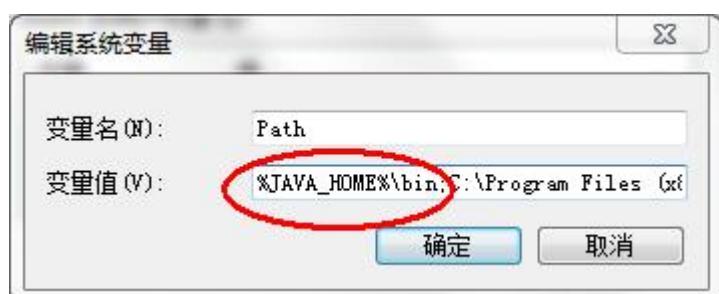
- 变量名输入 JAVA_HOME，变量值输入 D:\develop\Java\jdk1.8.0_202，并单击确定，如图所示：



- 选中 Path 环境变量，双击或者点击编辑，如图所示：



- 在变量值的最前面，键入 %JAVA_HOME%\bin; 分号必须要写，而且还要是英文符号。如图所示：



- 环境变量配置完成，重新开启DOS命令行，在任意目录下输入 javac 命令，运行成功。

```

C:\Users\Shark>javac
用法: javac <options> <source files>
其中, 可能的选项包括:
-g                                     生成所有调试信息
-g:none                               不生成任何调试信息
-g:<lines,vars,source>                只生成某些调试信息
-nowarn                                不生成任何警告
-verbose                                输出有关编译器正在执行的操作的消息
-deprecation                           输出使用已过时的 API 的源位置
-classpath <路径>                      指定查找用户类文件和注释处理程序的位置
-cp <路径>                            指定查找用户类文件和注释处理程序的位置
-sourcepath <路径>                     指定查找输入源文件的位置
-bootclasspath <路径>                  覆盖引导类文件的位置
-extdirs <目录>                        覆盖所安装扩展的位置
-endorseddirs <目录>                  覆盖签名的标准路径的位置
-proc:<none,only>                     控制是否执行注释处理和/或编译。
-processor <class1>[,<class2>,<class3>...]> 要运行的注释处理程序的名称; 绕过默
认的搜索进程
-processorpath <路径>                  指定查找注释处理程序的位置
-parameters                            生成元数据以用于方法参数的反射
-d <目录>                             指定放置生成的类文件的位置
-s <目录>                             指定放置生成的源文件的位置

```

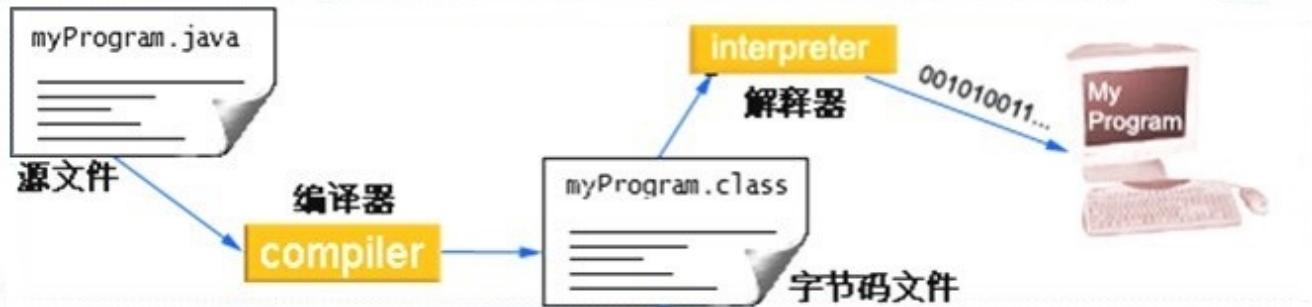
1.5 HelloWorld (动手实现)

1.5.1 开发说明

1、程序开发步骤说明

JDK安装完毕，可以开发我们第一个Java程序了。

Java程序开发三步骤：编写、编译、运行。



2、编写Java源程序保存.java源文件

- 在 D:\atguigu\javaee\JavaSE20190624\code\day01_code 目录下新建文本文件，完整的文件名修改为 `Helloworld.java`，其中文件名为 `Helloworld`，后缀名必须为`.java`。
- 用notepad++等文本编辑器打开（虽然是记事本也可以，但是不够没有关键字颜色标识，不利于初学者学习）
- 在文件中输入如下代码，并且保存：

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("HelloWorld");  
    }  
}
```

友情提示：

每个字母和符号必须与示例代码一模一样。

第一个 HelloWorld 源程序就编写完成了，但是这个文件是程序员编写的，JVM 是看不懂的，也就不能运行，因此我们必须将编写好的 Java 源文件 编译成 JVM 可以看懂的 字节码文件，也就是 ==.class== 文件。

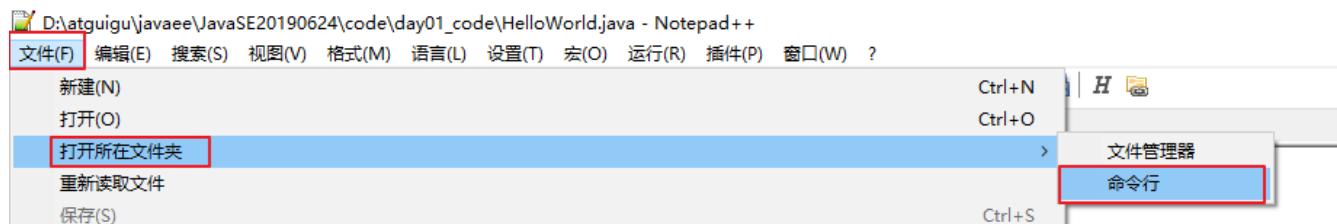
3、编译Java源文件生成.class字节码文件

在 DOS 命令行中，进入 D:\atguigu\javaee\JavaSE20190624\code\day01_code 目录，使用 javac 命令进行编译。

方式一：使用文件资源管理器打开 D:\atguigu\javaee\JavaSE20190624\code\day01_code 目录，然后在地址栏输入 cmd。



方式二：在 notepad++ 软件的文件菜单-->打开所在文件夹-->命令行（要求 notepad++ 软件必须是用管理员权限启动的，否则会出现已经正确配置了环境变量，却仍然找不到 javac 命令的问题）

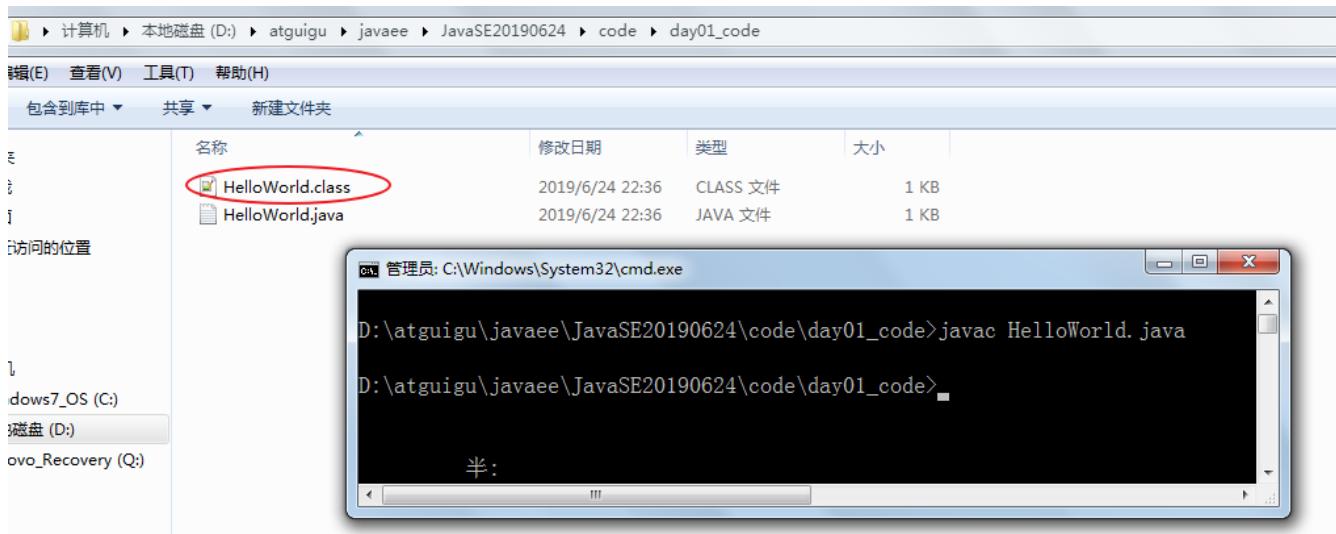


命令：

```
javac Java源文件名.后缀名
```

举例：

```
javac HelloWorld.java
```



编译成功后，命令行没有任何提示。打开 `D:\atguigu\javaee\JavaSE20190624\code\day01_code` 目录，发现产生了一个新的文件 `HelloWorld.class`，该文件就是编译后的文件，称为**字节码文件**，有了字节码文件，就可以运行程序了。

Java源文件的编译工具 `javac.exe`

4、运行Java程序

在DOS命令行中，在字节码文件目录下，使用 `java` 命令进行运行。

命令：

```
java 主类名
```

主类是指包含main方法的类，main方法是Java程序的入口：

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

举例：

```
java HelloWorld
```

友情提示：

`java HelloWorld` 不要写 不要写 不要写 `.class`

The screenshot shows a Windows Command Prompt window titled "管理员: C:\Windows\System32\cmd.exe". The command `javac HelloWorld.java` is run, followed by `java HelloWorld`, which outputs "HelloWorld". The window has a standard title bar, minimize, maximize, and close buttons.

```
D:\atguigu\javaee\JavaSE20190624\code\day01_code>javac HelloWorld.java  
D:\atguigu\javaee\JavaSE20190624\code\day01_code>java HelloWorld  
HelloWorld  
D:\atguigu\javaee\JavaSE20190624\code\day01_code>
```

Java字节码文件的运行工具：java.exe

1.5.2 常见错误（会解决错误）

1、书写错误

- 单词拼写问题
 - 正确：class 错误：Class
 - 正确：String 错误：string
 - 正确：System 错误：system
 - 正确：main 错误：mian
- Java语言是一门严格区分大小写的语言
- 标点符号使用问题
 - 不能用中文符号，英文半角的标点符号（正确）
 - 括号问题，成对出现

2、Java程序的结构与格式

结构：

```
类{  
    方法{  
        语句;  
    }  
}
```

格式：

- (1) 每一级缩进一个Tab键
- (2) {} 的左半部分在行尾，右半部分单独一行，与和它成对的 "{}" 的行首对齐

3、字符编码问题

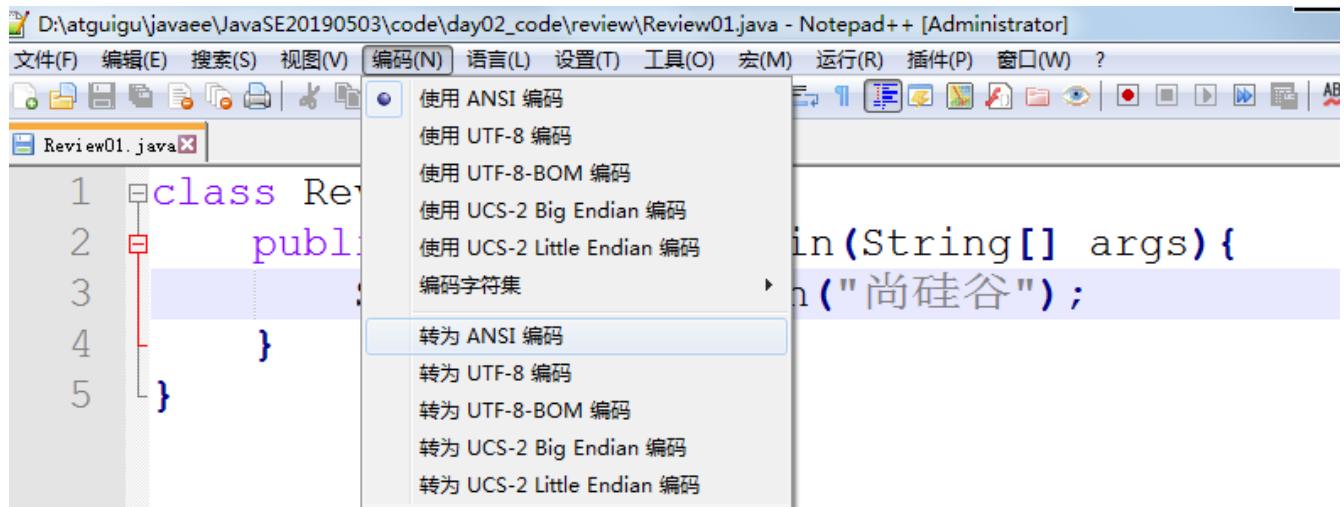
当cmd命令行窗口的字符编码与.java源文件的字符编码不一致，如何解决？

```
D:\atguigu\javaee\JavaSE20190503\code\day02_code\review>javac Review01.java  
Review01.java:3: 错误：编码GBK的不可映射字符  
        System.out.println("灝氯 璇?");
```

1 个错误

解决方案一：

在Notepad++等编辑器中，修改源文件的字符编码



解决方案二：

在使用javac命令式，可以指定源文件的字符编码

```
javac -encoding utf-8 Review01.java
```

4、大小写问题

(1) 源文件名：

在windows操作系统中.java的源文件名不区分大小写，我们建议大家养成区分大小写的习惯。

(2) 字节码文件名与类名

严格区分大小写

(3) 代码中

严格区分大小写

5、源文件名与类名一致问题？

(1) 源文件名是否必须与类名一致? public呢?

如果这个类不是public, 那么源文件名可以和类名不一致。但是不利于代码维护。

如果这个类是public, 那么要求源文件名必须与类名一致。否则编译报错。

我们建议大家, 不管是否是public, 都与源文件名保持一致, 而且一个源文件尽量只写一个类, 目的是为了好维护。

(2) 一个源文件中是否可以有多个类? public呢?

一个源文件中可以有多个类, 编译后会生成多个.class字节码文件。

但是一个源文件只能有一个public的类。

(3) main方法必须在public的类中吗?

不是。

但是后面写代码时, 基本上main方法(主方法)习惯上都在public类中。

第2章 Java基础语法

2.1 注释 (*annotation*) (掌握)

- **注释:** 就是对代码的解释和说明。其目的是让人们能够更加轻松地了解代码。为代码添加注释, 是十分必须要的, 它不影响程序的编译和运行。
- Java中有**单行注释**、**多行注释**和**文档注释**
 - 单行注释以//开头, 以换行结束, 格式如下:

```
// 注释内容
```

- 多行注释以/*开头, 以*/结束, 格式如下:

```
/*
    注释内容
*/
```

- 文档注释以/**开头, 以 */结束, Java特有的注释, 结合

```
/**
    注释内容
*/
```

```
//单行注释
/*
多行注释
*/
/**
文档注释演示
@author chai
*/
public class Comments{

    /**
     * Java程序的入口
     * @param String[] args main方法的命令参数
     */
    public static void main(String[] args){
        System.out.println("hello");
    }
}
```

常见的几个注释：

- @author 标明开发该类模块的作者，多个作者之间使用,分割
- @version 标明该类模块的版本
- @see 参考转向，也就是相关主题
- @since 从哪个版本开始增加的
- @param 对方法中某参数的说明，如果没有参数就不能写（后面再学）
- @return 对方法返回值的说明，如果方法的返回值类型是void就不能写（后面再学）
- @throws/@exception 对方法可能抛出的异常进行说明，如果方法没有用throws显式抛出的异常就不能写（后面再学）

其中 @param @return 和 @exception 这三个标记都是只用于方法的。

- @param的格式要求：@param 形参名 形参类型 形参说明
- @return 的格式要求：@return 返回值类型 返回值说明
- @exception 的格式要求：@exception 异常类型 异常说明
- @param和@exception可以并列多个

使用javadoc工具可以基于文档注释生成API文档。

用法： javadoc [options] [packagenames] [sourcefiles] [@files]

例如：

```
javadoc -author -d doc Comments.java
```

```
C:\Windows\System32\cmd.exe
D:\atguigu\javaee\JavaSE20220106\day20220106_01code>javadoc -author -d doc Comments.java
正在加载源文件Comments.java...
正在构造 Javadoc 信息...
标准 Doclet 版本 1.8.0_271
正在构建所有程序包和类的树...
正在生成doc\Comments.html...
正在生成doc\package-frame.html...
正在生成doc\package-summary.html...
正在生成doc\package-tree.html...
正在生成doc\constant-values.html...
正在构建所有程序包和类的索引...
正在生成doc\overview-tree.html...
正在生成doc\index-all.html...
正在生成doc\deprecated-list.html...
正在构建所有类的索引...
正在生成doc\allclasses-frame.html...
正在生成doc\allclasses-noframe.html...
正在生成doc\index.html...
正在生成doc\help-doc.html...
```

> Data (D:) > atguigu > javaee > JavaSE20220106 > day20220106_01code > doc

| 名称 | 修改日期 | 类型 | 大小 |
|-------------------------|------------------|------------------|-------|
| allclasses-frame.html | 2021/12/26 16:37 | Chrome HTML D... | 1 KB |
| allclasses-noframe.html | 2021/12/26 16:37 | Chrome HTML D... | 1 KB |
| Comments.html | 2021/12/26 16:37 | Chrome HTML D... | 8 KB |
| constant-values.html | 2021/12/26 16:37 | Chrome HTML D... | 4 KB |
| deprecated-list.html | 2021/12/26 16:37 | Chrome HTML D... | 4 KB |
| help-doc.html | 2021/12/26 16:37 | Chrome HTML D... | 7 KB |
| index.html | 2021/12/26 16:37 | Chrome HTML D... | 3 KB |
| index-all.html | 2021/12/26 16:37 | Chrome HTML D... | 5 KB |
| overview-tree.html | 2021/12/26 16:37 | Chrome HTML D... | 4 KB |
| package-frame.html | 2021/12/26 16:37 | Chrome HTML D... | 1 KB |
| package-list | 2021/12/26 16:37 | 文件 | 1 KB |
| package-summary.html | 2021/12/26 16:37 | Chrome HTML D... | 4 KB |
| package-tree.html | 2021/12/26 16:37 | Chrome HTML D... | 4 KB |
| script.js | 2021/12/26 16:37 | JavaScript 文件 | 1 KB |
| stylesheet.css | 2021/12/26 16:36 | 层叠样式表文档 | 14 KB |

The screenshot shows a Java documentation interface. At the top, there's a header bar with tabs for '生成的文档 (无标题)' (Generated Document (Untitled)), '文件 | D:/atguigu/javaee/JavaSE20220106/day20220106_01code/doc/index.html' (File | D:/atguigu/javaee/JavaSE20220106/day20220106_01code/doc/index.html), and a '+' button. Below the header is a toolbar with navigation icons: back, forward, search, and others. The main content area has a sidebar on the left labeled '所有类' (All Classes) with 'Comments' selected. The main panel shows the 'Comments' class definition under 'java.lang.Object'. It includes the class code, a '文档注释演示' (Documentation Annotation Demonstration) section, author information ('作者: chai'), and two tabs for '构造器概要' (Constructor Summary) and '方法概要' (Method Summary). The '构造器概要' tab is active, showing a single constructor 'Comments()'. The '方法概要' tab is also shown with tabs for '所有方法' (All Methods), '静态方法' (Static Methods), and '具体方法' (Concrete Methods). The '所有方法' tab is active, showing the 'main' method with its signature 'main(java.lang.String[] args)' and description 'Java程序的入口' (Entry point of the Java program).

2.2 关键字 (*keyword*) (掌握)

关键字: 是指在程序中, Java已经定义好的单词, 具有特殊含义。

- HelloWorld案例中, 出现的关键字有 `public`、`class`、`static`、`void` 等, 这些单词已经被Java定义好
- 关键字的特点: 全部都是 小写字母。
- 关键字比较多, 不需要死记硬背, 学到哪里记到哪里即可。

50 character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers (§3.8).

Keyword:

(one of)

| | | | | |
|----------|----------|------------|-----------|--------------|
| abstract | continue | for | new | switch |
| assert | default | if | package | synchronized |
| boolean | do | goto | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.

While `true` and `false` might appear to be keywords, they are technically boolean literals (§3.10.3). Similarly, while `null` might appear to be a keyword, it is technically the null literal (§3.10.7).

关键字一共50个，其中`const`和`goto`是保留字。

`true`,`false`,`null`看起来像关键字，但从技术角度，它们是特殊的布尔值和空值。

2.3 标识符(identifier) (掌握)

简单的说，凡是程序员自己命名的部分都可以称为标识符。

即给类、变量、方法、包等命名的字符序列，称为标识符。

1、标识符的命名规则（必须遵守的硬性规则）

- (1) Java的标识符只能使用26个英文字母大小写，0-9的数字，下划线_，美元符号\$
- (2) 不能使用Java的关键字（包含保留字）和特殊值
- (3) 数字不能开头
- (4) 不能包含空格
- (5) 严格区分大小写

2、标识符的命名规范（建议遵守的软性规则，否则容易被鄙视和淘汰）

(1) 见名知意

(2) 类名、接口名等：每个单词的首字母都大写，形式：XxxYyyZzz，

例如：HelloWorld, String, System等

(3) 变量、方法名等：从第二个单词开始首字母大写，其余字母小写，形式：xxxYyyZzz，

例如：age, name, bookName, main

(4) 包名等：每一个单词都小写，单词之间使用点.分割，形式：xxx.yyy.zzz，

例如：java.lang

(5) 常量名等：每一个单词都大写，单词之间使用下划线_分割，形式：XXX_YYY_ZZZ，

例如：MAX_VALUE, PI

更多细节详见《代码整洁之道.pdf》《Java开发手册（泰山版）》

2.4 初识数据类型(data type) (掌握)

Java的数据类型分为两大类：

- **基本数据类型**：包括 整数、浮点数、字符、布尔。
- **引用数据类型**：包括数组、类、接口、枚举、注解。



2.5 常量值 (constant) (掌握)

- **常量值**：在程序执行的过程中，其值不可以发生改变
- 常量值的分类：

| 类型 | 举例 |
|--------|-------------------------|
| 整数常量值 | 12, -23, 1567844444557L |
| 浮点常量值 | 12.34F, 12.34 |
| 字符常量值 | 'a', '0', '尚' |
| 布尔常量值 | true, false |
| 字符串常量值 | "HelloWorld" |

- 整数常量值，超过int范围的必须加L或l（小写L）
- 小数常量值，无论多少，不加F，就是double类型。要表示float类型，必须加F或f
- char常量值，必须使用单引号
- String字符串常量值，必须使用双引号

```
/*
常量值：
    代码里面写死的，固定不变的。
    一目了然的值。
整数常量值: 1      或 1L
小数常量值: 1.5    或 1.5F
单字符常量值: 'a'
布尔型常量值: true, false
字符串常量值: "hello"
*/
public class ConstantValue{
    public static void main(String[] args){
        System.out.println(1); //识别为int
        System.out.println(1L); //识别为long, 数字后面加L或l
        System.out.println(1.5); //识别为double
        System.out.println(1.5F); //识别为float类型, 数字后面加F或f
        System.out.println('a'); //识别为char类型, 单引号
        System.out.println(true); //识别为boolean类型
        System.out.println(false); //识别为boolean类型
        System.out.println("helloworld"); //识别为String类型, 双引号
        System.out.println("1"); //识别为String类型, 双引号
    }
}
```

2.6 变量（掌握）

2.6.1 变量的概念

变量：在程序执行的过程中，其值可以发生改变的量

变量的作用：用来存储数据，代表内存的一块存储区域，这块内存中的值是可以改变的。

2.6.2 变量的声明

```
数据类型 变量名；  
例如：  
//存储一个整数类型的年龄  
int age;  
  
//存储一个小数类型的体重  
double weight;  
  
//存储一个单字符类型的性别  
char gender;  
  
//存储一个布尔类型的婚姻状态  
boolean marry;  
  
//存储一个字符串类型的姓名  
String name;  
  
//声明多个同类型的变量  
int a,b,c; //表示a,b,c三个变量都是int类型。
```

注意：变量的数据类型可以是基本数据类型，也可以是引用数据类型。

2.6.3 变量的赋值

给变量赋值，就是把“值”存到该变量代表的内存空间中。

1、变量赋值的语法格式

```
变量名 = 值；
```

- 给变量赋值，变量名必须在=左边，值必须在=右边
- 给变量赋的值类型必须与变量声明的类型一致或兼容 (<=)

2、可以使用合适类型的常量值给变量赋值

```
int age = 18;  
double weight = 44.4;  
char gender = '女';  
boolean marry = true;  
String name = "柴林燕";
```

long类型：如果赋值的常量整数超过int范围，那么需要在数字后面加L。

float类型：如果赋值为常量小数，那么需要在小数后面加F。

char类型：使用单引号"

String类型：使用双引号""

3、可以使用其他变量或者表达式给变量赋值

```
int m = 1;
int n = m;

int x = 1;
int y = 2;
int z = 2 * x + y;
```

2.6.4 变量值的输出

```
//输出变量的值
System.out.println(age);

//输出变量的值
System.out.println("年龄: " + age);
System.out.println("age: " + age);
System.out.println("name" + name + ",age = " + age + ", gender = " + gender + ",weight = " +
weight + ",marry = " + marry);
```

如果()中有多项内容，那么必须使用 + 连接起来

如果某些内容想要原样输出，就用""引起来，而要输出变量中的内容，则不要把变量名用""引起来

2.6.5 变量可以反复赋值

- 变量的第一次赋值称为初始化；
- 变量的再赋值称为修改变量的值；

```
//先声明，后初始化
char gender;
gender = '女';

//声明的同时初始化
int age = 18;
System.out.println("age = " + age);///age = 18

//给变量重新赋值，修改age变量的值
age = 19;
System.out.println("age = " + age);///age = 19
```

2.6.6 变量的三要素

1、数据类型

- 变量的数据类型决定了在内存中开辟多大空间
- 变量的数据类型也决定了该变量可以存什么值

2、变量名

- 见名知意非常重要
- 给这块内存区域赋值，例如：我们现在在“宏福科技园”，我们住的校区叫做“流星花园”

3、值

- 基本数据类型的变量：存储数据值
- 引用数据类型的变量：存储地址值，即对象的首地址。例如：String类型的变量存储的是字符串对象的首地址
(关于对象后面章节再详细讲解)

2.6.7 变量的使用应该注意什么？

1、先声明后使用

如果没有声明，会报“找不到符号”错误

2、在使用之前必须初始化

如果没有初始化，会报“未初始化”错误

3、变量有作用域

如果超过作用域，也会报“找不到符号”错误

4、在同一个作用域中不能重名

5、变量值的类型必须与变量声明的类型一致或兼容 (<=)

一致：一样

```
int age = 18; 18是int类型的常量值, age也是int类型
int = int
```

兼容：可以装的下，=右边的值要 小于等于 =左边的变量类型

```
long bigNum =18; 18是int类型的常量值, bigNum是long类型
int < long
```

```
int age = 18L; 错误 18L是long类型的常量值, age是int类型
long > int
```

2.6.8 练习

```
/*
4、用合适类型的变量存储个人信息并输出
存储自己的姓名、年龄、性别、体重、婚姻状况
(已婚用true表示，单身用false表示) 等等
*/
public class MyInfo{
    public static void main(String[] args){
        //存储姓名用String类型
        //=左边是变量名，右边是常量值，给变量赋值
        String name = "柴林燕";

        System.out.println("name");//原样显示字符串常量值"name"
        System.out.println(name);//把变量name中的值输出

        int age = 18;
        char gender = '女';
        double weight = 42.5;
```

```

        boolean marry = true;
        System.out.println(age);
        System.out.println(gender);
        System.out.println(weight);
        System.out.println(marry);

        System.out.println("-----");
        // + 表示拼接，把"姓名："字符串常量 和name字符串变量的值，拼接起来，构成一个字符串值
        System.out.println("姓名：" + name);
        System.out.println("name=" + name);
        //System.out.println("姓名：" ,name); //错误

        System.out.println("-----");
        System.out.println("name" + name + ",age = " + age + ", gender = " + gender +
",weight = " + weight + ",marry = " + marry);

    }
}

```

2.7 最终变量/常量 (final)

最终变量习惯上也称为常量，因为它是通过在声明变量的数据类型前面加final的方式实现的，所以叫最终变量。加final修饰后，这个变量的值就不能修改了，一开始赋值多少，就是多少，所以此时的变量名通常称为常量名。常量名通常所有字母都大写，每一个单词之间使用下划线分割，从命名上和变量名区分开来。

这样做的好处，就是可以见名知意，便于维护。

```

public class FinalvariableDemo {
    public static void main(String[] args){
        //定义常量
        final int FULL_MARK = 100;//满分
        // FULL_MARK = 150;//错误，final修饰的变量，是常量，不能重新赋值

        //输出常量值
        System.out.println("满分：" + FULL_MARK);

        //小王的成绩比满分少1分
        int wang = FULL_MARK - 1;
        //小尚得了满分
        int shang = FULL_MARK;
        //小刘得了一半分
        int liu = FULL_MARK/2;

        //输出变量值
        System.out.println("小王成绩：" + wang);
        System.out.println("小尚成绩：" + shang);
        System.out.println("小刘成绩：" + liu);
    }
}

```

2.8 计算机如何存储数据

计算机世界中只有二进制。那么在计算机中存储和运算的所有数据都要转为二进制。包括数字、字符、图片、声音、视频等。

2.8.1 进制（了解）

1、进制的分类

(1) 十进制：数字组成：0-9 进位规则：逢十进一

(2) 二进制：数字组成：0-1 进位规则：逢二进一

十进制的256，二进制：100000000，为了缩短二进制的表示，又要贴近二进制，在程序中引入八进制和十六进制

(3) 八进制：很少使用 数字组成：0-7 进位规则：逢八进一

与二进制换算规则：每三位二进制是一位八进制值

(4) 十六进制 数字组成：0-9, a-f 进位规则：逢十六进一

与二进制换算规则：每四位二进制是一位十六进制值

2、进制的换算

| 十进制 | 二进制 | 八进制 | 十六进制 |
|-----|-------|-----|------|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | a或A |
| 11 | 1011 | 13 | b或B |
| 12 | 1100 | 14 | c或C |
| 13 | 1101 | 15 | d或D |
| 14 | 1110 | 16 | e或E |
| 15 | 1111 | 17 | f或F |
| 16 | 10000 | 20 | 10 |

- **十进制数据转成二进制数据:** 使用除以2倒取余数的方式

十进制数字6转成二进制,除以2获取余数

$$\begin{array}{r}
 & \text{余} \\
 2 & \overline{)6} & 0 & \uparrow & \text{结果是110} \\
 2 & \overline{)3} & 1 & & \\
 2 & \overline{)1} & 1 & & \\
 & \overline{)0} & & &
 \end{array}$$

- **二进制数据转成十进制数据:**

从右边开始依次是2的0次, 2的1次, 2的2次。 . . .

二进制数据 1001011转成十进制

| | | | | | | |
|----|----|----|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 64 | 32 | 16 | 8 | 4 | 2 | 1 |

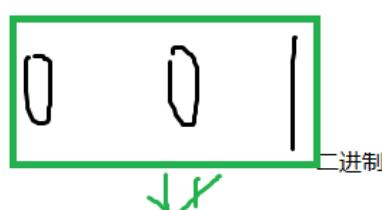
把有1位上的十进制求和
 $64+8+2+1=75$

- 二进制数据转八进制数据

从右边开始，三位一组



$$\begin{aligned} &0^*2的2次 + 1^*2的1次 + 1^*2的0次 \\ &= 0 + 2 + 1 \\ &= 3 \end{aligned}$$



$$\begin{aligned} &0^*2的2次 + 0^*2的1次 + 1^*2的0次 \\ &= 0 + 0 + 1 \\ &= 1 \end{aligned}$$

发现八进制0-7，对于二进制在三位以内
把二进制三位一组

八进制-->十进制

$$\begin{aligned} &3\ 1 \\ &3^*8的1次 + 1^*8的0次 \\ &= 24 + 1 \\ &= 25 \end{aligned}$$

- 二进制数据转十六进制数据

从右边开始，四位一组



$$\text{十六进制} = 19$$

发现十六进制的0-9，a-f，在二进制的四位范围内
我们用四位二进制表示一位十六进制

$$\begin{aligned} &\text{十六进制 } 19 \rightarrow \text{十进制} \\ &1^*16的1次 + 0^*16的0次 \\ &= 16 + 9 \\ &= 25 \end{aligned}$$

3、在代码中如何表示四种进制的常量值

请分别用四种类型的进制来表示10，并输出它的结果：（了解）

- (1) 十进制：正常表示

```
System.out.println(10);
```

- (2) 二进制：0b或0B开头

```
System.out.println(0B10);
```

- (3) 八进制：0开头

```
System.out.println(010);
```

(4) 十六进制：0x或0X开头

```
System.out.println(0X10);
```

2.8.2 计算机存储单位（掌握）

- **字节 (Byte)**：是计算机信息技术用于计量存储容量的一种计量单位，一字节等于八位。
- **位 (bit)**：是数据存储的最小单位。也就是二进制。二进制数系统中，每个0或1就是一个位，叫做bit (比特)，其中8 bit 就称为一个字节(Byte)。
- **转换关系：**
 - 8 bit = 1 Byte
 - 1024 Byte = 1 KB
 - 1024 KB = 1 MB
 - 1024 MB = 1 GB
 - 1024 GB = 1 TB

2.8.3 Java的基本数据类型的存储范围（掌握）

| 数据类型 | 关键字 | 内存占用(字节) | 取值范围 |
|------|-------------|----------|---|
| 整数 | byte | 1 | -2的7次方到2的7次方-1 (-128~127) |
| | short | 2 | -2的15次方到2的15次方-1 (-32768~32767) |
| | int (默认) | 4 | -2的31次方到2的31次方-1 (-2147483648~2147483647) |
| | long | 8 | -2的63次方到2的63次方-1 (-9223372036854775808~9223372036854775807) |
| 浮点数 | float | 4 | 负数：-3.402823E38到-1.401298E-45 整数：0.0 正数：1.401298E-45到3.402823E38 |
| | double (默认) | 8 | 负数：-1.797693E308到-4.940656E-324 整数：0.0 正数：4.940656E-324 到1.797693E308 |
| 字符 | char | 2 | 0-65535 |
| 布尔 | boolean | 1 | true , false |

float：单精度浮点型，占内存：4个字节，精度：科学记数法的小数点后6~7位

double：双精度浮点型，占内存：8个字节，精度：科学记数法的小数点后15~16位

2.8.4 计算机如何表示数据（理解）

1、如何表示boolean类型的值

true底层使用1表示。

false底层使用0表示。

2、如何表示整数？

原码、反码、补码与符号位概念

计算机数据的存储使用二进制补码形式存储，并且最高位是符号位，最高位1是负数，最高位0是正数。

规定：正数的补码与反码、原码一样，称为三码合一；

负数的补码与反码、原码不一样：

负数的原码：把十进制转为二进制，然后最高位设置为1

负数的反码：在原码的基础上，最高位不变，其余位取反（0变1, 1变0）

负数的补码：反码+1

例如：byte类型（1个字节，8位）

25 ==> 原码 0001 1001 ==> 反码 0001 1001 --> 补码 0001 1001

-25 ==> 原码 1001 1001 ==> 反码 1110 0110 ==> 补码 1110 0111

整数：

正数：25 00000000 00000000 00000000 00011001 (原码)

正数：25 00000000 00000000 00000000 00011001 (反码)

正数：25 00000000 00000000 00000000 00011001 (补码)

负数：-25 10000000 00000000 00000000 00011001 (原码)

负数：-25 11111111 11111111 11111111 11100110 (反码)

负数：-25 11111111 11111111 11111111 11100111 (补码)

一个字节可以存储的整数范围是多少？

1个字节：8位

0000 0001 ~ 0111 1111 ==> 1~127

1000 0001 ~ 1111 1111 ==> -127 ~ -1

0000 0000 ==> 0

1000 0000 ==> -128 (特殊规定) = -127 - 1

3、如何表示小数？

了解小数如何存储是为了理解如下问题：

- 为什么float（4个字节）比long（8个字节）的存储范围大？
- 为什么float和double不精确？
- 为什么double（8个字节）比float（4个字节）精度范围大？

因为float、double底层也是二进制，先把小数转为二进制，然后把二进制表示为科学记数法，然后只保存：

①符号位②指数位（需要移位）③尾数位

详见《float型和double型数据的存储方式.docx》

float: 符号位 (1位) , 指数位 (8位, 偏移127) , 尾数位 (23位)
double: 符号位 (1位) , 指数位 (11位, 偏移1023) , 尾数为 (52位)
float指数-126~+127
double指数-1022~+1023

float类型

小数: 8.25 1000.01

1.00001 (科学计数法)
 符号位0, 指数位3+127 (偏移量) =130->10000010, 尾数00001
 0 10000010 00001000000000000000000000000000 原码
 0 10000010 00001000000000000000000000000000 反码
 0 10000010 00001000000000000000000000000000 补码

小数: -8.25 -1000.01 (原码)

1 10000010 00001000000000000000000000000000 原码
1 01111101 111101111111111111111111 反码
1 01111101 111110000000000000000000 补码

double类型:

小数: 8.25 1000.01

1.00001 (科学计数法)
 符号位0, 指数位3+1023 (偏移量) =1026->10000000010, 尾数00001
 0 10000000010 0000 10000000 00000000 00000000 00000000 00000000
原码
 0 10000000010 0000 10000000 00000000 00000000 00000000 00000000
反码
 0 10000000010 0000 10000000 00000000 00000000 00000000 00000000
补码

double类型:

小数: -8.25 -1000.01 (原码)

1.00001 (科学计数法)
 符号位0, 指数位3+1023 (偏移量) =1026->10000000010, 尾数00001
 1 10000000010 0000 10000000 00000000 00000000 00000000 00000000
原码
 1 0111111101 1111 01111111 11111111 11111111 11111111 11111111 11111111
反码
 1 0111111101 1111 10000000 00000000 00000000 00000000 00000000 00000000
补码

为什么float类型指数位偏移127, double类型指数位偏移1023。

因为指数+3, 偏移127就是130

因为指数-3, 偏移127就是124

130>124, 比较大小比较方便。

s , m , and e , then if it happened that m were even and e were less than 2^{K-1} , one could halve m and increase e by 1 to produce a second representation for the same value v . A representation in this form is called *normalized* if $m \geq 2^{N-1}$; otherwise the representation is said to be *denormalized*. If a value in a value set cannot be represented in such a way that $m \geq 2^{N-1}$, then the value is said to be a *denormalized value*, because it has no normalized representation.

The constraints on the parameters N and K (and on the derived parameters E_{min} and E_{max}) for the two required and two optional floating-point value sets are summarized in Table 4.2.3-A.

Table 4.2.3-A. Floating-point value set parameters

| Parameter | float | float-extended-exponent | double | double-extended-exponent |
|-----------|-------|-------------------------|--------|--------------------------|
| N | 24 | 24 | 53 | 53 |
| K | 8 | ≥ 11 | 11 | ≥ 15 |
| E_{max} | +127 | $\geq +1023$ | +1023 | $\geq +16383$ |
| E_{min} | -126 | ≤ -1022 | -1022 | ≤ -16382 |

4、Java程序中如何表示和处理单个字符？

(1) 使用单引号将单个字符引起来：例如：'A', '0', '尚'

```
char c = '尚';//使用单引号
String s = '尚';//错误的，哪怕是一个字符，也要使用双引号

char kongChar = '';//错误，单引号中有且只能有一个字符
String kongStr = "";//可以，双引号中可以没有其他字符，表示是空字符串
```

(2) 特殊的转义字符

```
\n: 换行
\r: 回车
\t: Tab键
\\: \
\": "
\'': '
\b: 删删除键Backspace
```

```
public class TestEscapeCharacter {
    public static void main(String[] args){
        System.out.println("hello\tjava");
        System.out.println("hello\rjava");
        System.out.println("hello\njava");
        System.out.println("hello\\world");
        System.out.println("\\"hello\"");
        char shuang = '''';
        System.out.println(shuang + "hello" + shuang);
        System.out.println('''hello''');
        char dan ='\'';
    }
}
```

```

        System.out.println(dan + "hello" + dan);
    }
}

public class TestTab {
    public static void main(String[] args){
        System.out.println("hello\tworld\tjava.");
        System.out.println("chailinyan\tis\tbeautiful.");
        System.out.println("姓名\t基本工资\t年龄");
        System.out.println("张三\t10000.0\t23");
    }
}

```

(3) 用十进制的0~65535之间的Unicode编码值，表示一个字符

在JVM内存中，一个字符占2个字节，Java使用Unicode字符集来表示每一个字符，即每一个字符对应一个唯一的Unicode编码值。char类型的数值参与算术运算或比较大小时，都是用编码值进行计算的。

| 字符 | Unicode编码值 |
|-----|------------|
| '0' | 48 |
| '1' | 49 |
| 'A' | 65 |
| 'B' | 66 |
| 'a' | 97 |
| 'b' | 98 |
| '尚' | 23578 |

```

char c1 = 23578;
System.out.println(c1);//尚

char c2 = 97;
System.out.println(c2);//a

//如何查看某个字符的unicode编码?
//将一个字符赋值给int类型的变量即可
int codeOfA = 'A';
System.out.println(codeOfA);

int codeOfShang = '尚';
System.out.println(codeOfShang);

int codeOfTab = '\t';
System.out.println(codeOfTab);

```

(4) \u字符的Unicode编码值的十六进制型

例如: '\u5c1a'代表'尚'

```
char c = '\u0041'; //十进制unicode值65, 对应十六进制是41, 但是\u后面必须写4位  
char c = '\u5c1a'; //十进制unicode值23578, 对应十六进制是5c1a
```

5、一个字符到底占几个字节?

在JVM内存中, 一个字符占2个字节, Java使用Unicode字符集来表示每一个字符, 即每一个字符对应一个唯一的Unicode编码值。char类型的数值参与算术运算或比较大小时, 都是用编码值进行计算的。

在文件中保存或网络中传输时文本数据时, 和环境编码有关。如果环境编码选择ISO8859-1 (又名Latin), 那么一个字符占一个字节; 如果环境编码选择GBK, 那么一个字符占1个或2个字节; 如果环境编码选择UTF-8, 那么一个字符占1-4个字节。 (后面讲String类时再详细讲解)

2.9 基本数据类型转换 (Conversion) (掌握)

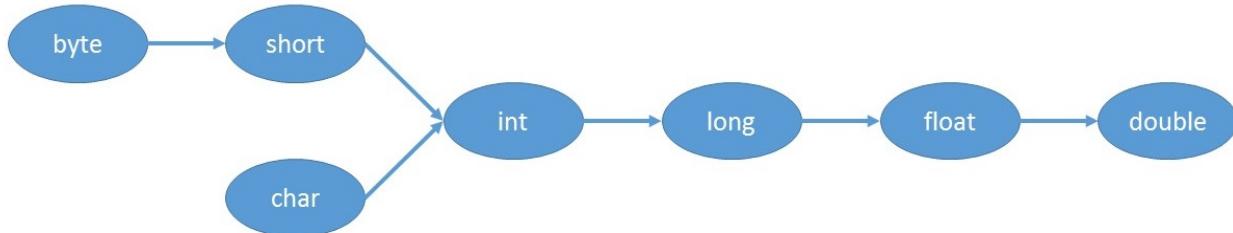
在Java程序中, 不同的基本数据类型的值经常需要进行相互转换。Java语言所提供的**七种数值类型**之间可以相互转换, 基本数据类型转换有两种转换方式: 自动类型转换和强制类型转换。==boolean类型不参与。==

2.9.1 自动类型转换 (隐式类型转换)

自动转换:

- 将取值范围小的类型自动提升为取值范围大的类型。

基本数据类型的转换规则如图所示:



自动类型转换图

(1) 当把存储范围小的值(常量值、变量的值、表达式计算的结果值)赋值给了存储范围大的变量时。

```
int i = 'A'; //char自动升级为int, 其实就是把字符的编码值赋值给i变量了  
double d = 10; //int自动升级为double  
  
byte b = 127; //右边的整数常量值必须在-128~127范围内  
//byte bigB = 130; //错误, 右边的整数常量值超过byte范围  
long num = 1234567; //右边的整数常量值如果在int范围呢, 编译和运行都可以通过, 这里涉及到数据类型转换  
long bigNum = 12345678912L; //右边的整数常量值如果超过int范围, 必须加L, 否则编译不通过
```

(2) 当存储范围小的数据类型与存储范围大的数据类型一起混合运算时, 会按照其中最大的类型运算。

```
int i = 1;
byte b = 1;
double d = 1.0;

double sum = i + b + d;//混合运算，升级为double
```

(3) 当byte,short,char数据类型进行算术运算时，按照int类型处理。

```
byte b1 = 1;
byte b2 = 2;
byte b3 = b1 + b2;//编译报错，b1 + b2自动升级为int

char c1 = '0';
char c2 = 'A';
System.out.println(c1 + c2);//113
```

2.9.2 强制类型转换（显示类型转换）

将1.5赋值到int类型变量会发生什么？产生编译失败，肯定无法赋值。

```
int i = 3.14; // 错误
```

想要赋值成功，只有通过强制类型转换，将double类型强制转换成int类型才能赋值。

- **强制类型转换**：将取值范围大的类型强制转换成取值范围小的类型。

比较而言，自动转换是Java自动执行的，而强制转换需要我们自己手动执行。

转换格式：

```
数据类型 变量名 = (数据类型) 被强转数据值； //()中的数据类型必须<=变量的数据类型，一般都是=
```

(1) 当把存储范围大的值（常量值、变量的值、表达式计算的结果值）赋值给了存储范围小的变量时，需要强制类型转换，提示：有风险，可能会损失精度或溢出

```
int i = (int)3.14;//强制类型转换，损失精度

double d = 1.2;
int num = (int)d;//损失精度

int i = 200;
byte b = (byte)i;//溢出
```

(2) 当某个值想要提升数据类型时，也可以使用强制类型转换

```
int i = 1;
int j = 2;
double shang = (double)i/j;
```

提示：这个情况的强制类型转换是没有风险的。

2.9.3 基本数据类型与字符串类型的转换

1、任意数据类型的数据与String类型进行“+”运算时，结果一定是String类型

```
System.out.println("") + 1 + 2); //12
```

2、但是String类型不能通过强制类型()转换，转为其他的类型

```
String str = "123";
int num = (int)str; //错误的
int num = Integer.parseInt(str); //后面才能讲到，借助包装类的方法才能转
```

2.10 运算符 (Operator) 和标点符号 (Separators) (掌握)

在Java中，一共有38个运算符。

3.12 Operators

38 tokens, formed from ASCII characters, are the *operators*.

Operator:
(one of)

| | | | | | | | |
|----|----|----|----|----|---|------|-----|
| = | > | < | ! | ~ | ? | : | -> |
| == | >= | <= | != | && | | ++ | -- |
| + | - | * | / | & | | ^ | % |
| += | -= | *= | /= | &= | = | ^= | %= |
| | | | | | | << | >> |
| | | | | | | <<= | >>= |
| | | | | | | >>>= | |

运算符的分类：

- 按照功能分：算术运算符、赋值运算符、比较运算符、逻辑运算、条件运算符、Lambda运算符

| 分类 | 运算符 |
|----------------|---|
| 算术运算符 (7个) | +、-、*、/、%、++、-- |
| 赋值运算符 (12个) | =、+=、-=、*=、/=、%=、>>=、<<=、>>>=、&=、 =、^=等 |
| 关系运算符 (6个) | >、>=、<、<=、==、!= |
| 逻辑运算符 (6个) | &、 、^、!、&&、 |
| 条件运算符 (2个) | (条件表达式)?结果1:结果2 |
| 位运算符 (7个) | &、 、^、~、<<、>>、>>> |
| Lambda运算符 (1个) | -> (后面学) |

- 按照操作数个数分：一元运算符（单目运算符）、二元运算符（双目运算符）、三元运算符（三目运算符）

| 分类 | 运算符 |
|---------------|-------------------------------|
| 一元运算符 (单目运算符) | 正号 (+) 、负号 (-) 、 ++、 --、 !、 ~ |
| 二元运算符 (双目运算符) | 除了一元和三元运算符剩下的都是二元运算符 |
| 三元运算符 (三目运算符) | (条件表达式)?结果1:结果2 |

2.10.1 算术运算符

| 算术运算符 | 符号解释 |
|-------|-------------------|
| + | 加法运算, 字符串连接运算, 正号 |
| - | 减法运算, 负号 |
| * | 乘法运算 |
| / | 除法运算, 整数/整数结果还是整数 |
| % | 求余运算, 余数的符号只看被除数 |
| ++、-- | 自增自减运算 |

1、加减乘除模

```
public class OperatorDemo01 {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;

        System.out.println(a + b); // 7
        System.out.println(a - b); // -1
        System.out.println(a * b); // 12
        System.out.println(a / b); // 计算机结果是0, 为什么不是0.75呢? 整数/整数结果还是整数
        System.out.println(a % b); // 3

        //余数的符号只看被除数
        System.out.println(5%2); //1
        System.out.println(5%-2); //1
        System.out.println(-5%2); // -1
        System.out.println(-5%-2); // -1
        //商*除数 + 余数 = 被除数
        //5%-2 ==> 商是-2, 余数是1 (-2)*(-2)+1 = 5
        //-5%2 ==> 商是-2, 余数是-1 (-2)*2+(-1) = -4-1=-5
    }
}
```

2、“+”号的两种用法

- 第一种：对于+两边都是数值的话，+就是加法的意思

- 第二种：对于`+`两边至少有一边是字符串得话，`+`就是拼接的意思

```
public class OperatorDemo02 {
    public static void main(String[] args) {
        // 字符串类型的变量基本使用
        // 数据类型 变量名称 = 数据值;
        String str1 = "Hello";
        System.out.println(str1); // Hello

        System.out.println("Hello" + "world"); // HelloWorld

        String str2 = "Java";
        // String + int --> String
        System.out.println(str2 + 520); // Java520
        // String + int + int
        // String           + int
        // String
        System.out.println(str2 + 5 + 20); // Java520
    }
}
```

3、自加自减运算

理解：`++` 运算，变量自己的值加1。反之，`--` 运算，变量自己的值减少1，用法与`++`一致。

1、单独使用

- 变量在单独运算的时候，变量`前++`和变量`后++`，变量的是一样的；
- 变量`前++`：例如`++a`。
- 变量`后++`：例如`a++`。

```
public class OperatorDemo3 {
    public static void main(String[] args) {
        // 定义一个int类型的变量a
        int a = 3;
        //++a;
        a++;
        // 无论是变量前++还是变量后++, 结果都是4
        System.out.println(a);
    }
}
```

2、复合使用

- 和其他变量放在一起使用或者和输出语句放在一起使用，`前++`和`后++`就产生了不同。
- 变量`前++`：变量先自身加1，然后再取值。
- 变量`后++`：变量先取值，然后再自身加1。

```
public class OperatorDemo03 {
    public static void main(String[] args) {
        // 其他变量放在一起使用
```

```

int x = 3;
//int y = ++x; // y的值是4, x的值是4,
int y = x++; // y的值是3, x的值是4

System.out.println(x);
System.out.println(y);
System.out.println("=====");

// 和输出语句一起
int z = 5;
//System.out.println(++z); // 输出结果是6, z的值也是6
System.out.println(z++); // 输出结果是5, z的值是6
System.out.println(z);

int a = 1;
a = a++; // (1)先取a的值“1”放操作数栈 (2)a再自增,a=2 (3)再把操作数栈中的“1”赋值给a,a=1

int i = 1;
int j = i++ + ++i * i++;
/*
从左往右加载
(1)先算i++
①取i的值“1”放操作数栈
②i再自增 i=2
(2)再算++i
③i先自增 i=3
④再取i的值“3”放操作数栈
(3)再算i++
⑤取i的值“3”放操作数栈
⑥i再自增 i=4
(4)先算乘法
用操作数栈中3 * 3 = 9，并把9压会操作数栈
(5)再算求和
用操作数栈中的 1 + 9 = 10
(6)最后算赋值
j = 10
*/
}
}

```

- 小结：
 - **++在前，先自加，后使用；**
 - **++在后，先使用，后自加。**
- 分析

```
public class TestIncrementOperator1{
    public static void main(String[] args){
        int i = 1;
        i++;
        ++i;
    }
}
```

C:\Windows\System32\cmd.exe

```
C:\Users\final\Desktop>javac TestIncrementOperator1.java

C:\Users\final\Desktop>javap -c TestIncrementOperator1.class
Compiled from "TestIncrementOperator1.java"
public class TestIncrementOperator1 {
    public TestIncrementOperator1();
        Code:
            0: aload_0
            1: invokespecial #1                  // Method java/lang/Object."<init>":()V
            4: return

    public static void main(java.lang.String[]);
        Code:
            0: iconst_1
            1: istore_1
            2: iinc           1, 1
            5: iinc           1, 1
            8: return
}
```

```
public class TestIncrementOperator2{
    public static void main(String[] args){
        int i = 1;
        i = i++;
    }
}
```

```
C:\Windows\System32\cmd.exe
C:\Users\final\Desktop>javac TestIncrementOperator2.java

C:\Users\final\Desktop>javap -c TestIncrementOperator2.class
Compiled from "TestIncrementOperator2.java"
public class TestIncrementOperator2 {
    public TestIncrementOperator2();
        Code:
          0: aload_0
          1: invokespecial #1                  // Method java/lang/Object."<init>":()V
          4: return

    public static void main(java.lang.String[]);
        Code:
          0: iconst_1
          1: istore_1
          2: iload_1
          3: iinc           1, 1
          6: istore_1
          7: return
}
```

```
public class TestIncrementOperator3{
    public static void main(String[] args){
        int i = 1;
        i = ++i;
    }
}
```

```
C:\Windows\System32\cmd.exe
C:\Users\final\Desktop>javac TestIncrementOperator3.java

C:\Users\final\Desktop>javap -c TestIncrementOperator3.class
Compiled from "TestIncrementOperator3.java"
public class TestIncrementOperator3 {
    public TestIncrementOperator3();
        Code:
          0: aload_0
          1: invokespecial #1                  // Method java/lang/Object."<init>":()V
          4: return

    public static void main(java.lang.String[]);
        Code:
          0: iconst_1
          1: istore_1
          2: iinc           1, 1
          5: iload_1
          6: istore_1
          7: return
}
```

2.10.2 关系运算符/比较运算符

| 关系运算符 | 符号解释 |
|-------|--------------------------------------|
| < | 比较符号左边的数据是否小于右边的数据，如果小于结果是true。 |
| > | 比较符号左边的数据是否大于右边的数据，如果大于结果是true。 |
| <= | 比较符号左边的数据是否小于或者等于右边的数据，如果大于结果是false。 |
| >= | 比较符号左边的数据是否大于或者等于右边的数据，如果小于结果是false。 |
| == | 比较符号两边数据是否相等，相等结果是true。 |
| != | 不等于符号，如果符号两边的数据不相等，结果是true。 |

- 比较运算符，是两个数据之间进行比较的运算，运算结果一定是boolean值 true 或者 false。
- 其中>,<,>=,<=不支持boolean, String类型， ==和!=支持boolean和String。

```
public class OperatorDemo05 {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;

        System.out.println(a < b); // true
        System.out.println(a > b); // false
        System.out.println(a <= b); // true
        System.out.println(a >= b); // false
        System.out.println(a == b); // false
        System.out.println(a != b); // true
    }
}
```

2.10.3 逻辑运算符

- 逻辑运算符，是用来连接两个布尔类型结果的运算符（! 除外），运算结果一定是boolean值 true 或者 false

| 逻辑运算符 | 符号解释 | 符号特点 |
|-------|--------|-------------------------------|
| & | 与，且 | 有 false 则 false |
| | 或 | 有 true 则 true |
| ^ | 异或 | 相同为 false，不同为 true |
| ! | 非 | 非 false 则 true，非 true 则 false |
| && | 双与，短路与 | 左边为false，则右边就不看 |
| | 双或，短路或 | 左边为true，则右边就不看 |

&&和&区别， ||和|区别：

- && 和 & 区别：

- `&&` 和 `&` 结果一样, `&&` 有短路效果, 左边为`false`, 右边不执行; `&` 左边无论是什么, 右边都会执行。
- `||` 和 `|` 区别:
 - `||` 和 `|` 结果一样, `||` 有短路效果, 左边为`true`, 右边不执行; `|` 左边无论是什么, 右边都会执行。

```

public class OperatorDemo06 {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int c = 5;

        // & 与, 且; 有false则false
        System.out.println((a > b) & (a > c));
        System.out.println((a > b) & (a < c));
        System.out.println((a < b) & (a > c));
        System.out.println((a < b) & (a < c));
        System.out.println("=====");

        // | 或; 有true则true
        System.out.println((a > b) | (a > c));
        System.out.println((a > b) | (a < c));
        System.out.println((a < b) | (a > c));
        System.out.println((a < b) | (a < c));
        System.out.println("=====");

        // ^ 异或; 相同为false, 不同为true
        System.out.println((a > b) ^ (a > c));
        System.out.println((a > b) ^ (a < c));
        System.out.println((a < b) ^ (a > c));
        System.out.println((a < b) ^ (a < c));
        System.out.println("=====");

        // ! 非; 非false则true, 非true则false
        System.out.println(!false);
        System.out.println(!true);

        // &和&&的区别
        System.out.println((a > b) & (a++ > c));
        System.out.println("a = " + a);
        System.out.println((a > b) && (a++ > c));
        System.out.println("a = " + a);
        System.out.println((a == b) && (a++ > c));
        System.out.println("a = " + a);

        // |和||的区别
        System.out.println((a > b) | (a++ > c));
        System.out.println("a = " + a);
        System.out.println((a > b) || (a++ > c));
        System.out.println("a = " + a);
        System.out.println((a == b) || (a++ > c));
        System.out.println("a = " + a);
    }
}

```

```

/*
3、逻辑运算符

```

逻辑与: &

```
true & true 结果是true  
true & false 结果是false  
false & true 结果是false  
false & false 结果是false
```

只有两个边都是true, 结果才为true。

逻辑或: |

```
true | true 结果是true  
true | false 结果是true  
false | true 结果是true  
false | false 结果是false
```

只要有一边是true, 结果就为true。

逻辑非: !

```
!true 变为false  
!false 变为true
```

逻辑异或: ^

```
true | true 结果是false  
true | false 结果是true  
false | true 结果是true  
false | false 结果是false
```

只有两边不一样, 一个是true, 一个是false, 结果才为true。

短路与: &&

```
true && true 结果是true  
true && false 结果是false  
false && ? 结果是false  
false && ? 结果是false
```

只有两个边都是true, 结果才为true。

但是它如果左边已经是false, 右边不看。这样的好处就是可以提高效率。

短路或: ||

```
true || ? 结果是true  
true || ? 结果是true  
false || true 结果是true  
false || false 结果是false
```

只要有一边是true, 结果就为true。

但是它如果左边已经是true, 右边就不看了。这样的好处就是可以提高效率。

特殊:

- (1) 逻辑运算符的操作数必须是boolean值
- (2) 逻辑运算符的结果也是boolean值

```
*/  
public class LogicOperator{
```

```

public static void main(String[] args){
    /*
    表示条件，成绩必须在[0,100]之间
    成绩是int类型变量score
    */
    int score = 56;

    //System.out.println(0<=score<=100);
    /*
    LogicOperator.java:23: 错误：二元运算符 '<=' 的操作数类型错误
    System.out.println(0<=score<=100);
                           ^
    第一个类型： boolean      0<=score的结果 true
    第二个类型： int

    true <= 100? 不对的
    1 个错误*/
    System.out.println(0<=score & score<=100);

}
}

```

2.10.4 条件运算符

- 条件运算符格式：

条件表达式？结果1：结果2

- 条件运算符计算方式：

- 条件判断的结果是true，条件运算符整体结果为结果1，赋值给变量。
- 判断条件的结果是false，条件运算符整体结果为结果2，赋值给变量。

```

public class ConditionOperator{
    public static void main(String[] args){
        //判断两个变量a,b谁大，把大的变量赋值给max
        int a = 2;
        int b = 2;
        int max = a >= b ? a : b;
        //如果a>=b成立，就取a的值赋给max，否则取b的值赋给max
        System.out.println(max);

        boolean marry = false;
        System.out.println(marry ? "已婚" : "未婚");
    }
}

```

2.10.5 位运算符

| 位运算符 | 符号解释 |
|------|-------------------------|
| & | 按位与，当两位相同时为1时才返回1 |
| | 按位或，只要有一位为1即可返回1 |
| ~ | 按位非，将操作数的每个位（包括符号位）全部取反 |
| ^ | 按位异或。当两位相同时返回0，不同时返回1 |
| << | 左移运算符 |
| >> | 右移运算符 |
| >>> | 无符号右移运算符 |

- 位运算符的运算过程都是基于补码运算，但是看结果，我们得换成原码，再换成十进制看结果
- 从二进制到十进制都是基于原码
- 正数的原码反码补码都一样，负数原码反码补码不一样
- byte,short,char在计算时按照int类型处理

如何区分&,|,^是逻辑运算符还是位运算符？

如果操作数是boolean类型，就是逻辑运算符，如果操作数是整数，那么就位运算符。

(1) 左移: <<

运算规则：左移几位就相当于乘以2的几次方

注意：当左移的位数n超过该数据类型的总位数时，相当于左移（n-总位数）位

byte,short,char在计算时按照int类型处理

3<<4 类似于 $3 \times 2^4 = 48$

```
/*
3的二进制: 0000 0000 0000 0000 0000 0000 0000 0011
3<<4: 0000 0000 0000 0000 0000 0000 0011 0000
*/      左边移出去4位,                               右边补4个0
```

-3<<4 类似于 $-3 \times 2^4 = -48$

```

/*
-3的二进制:
    原码: 1000 0000 0000 0000 0000 0000 0000 0011
    反码: 1111 1111 1111 1111 1111 1111 1111 1100
    补码: 1111 1111 1111 1111 1111 1111 1111 1101
-3<<4: 1111 1111 1111 1111 1111 1111 1111 1101 0000
    左边移出去4位  

    补码: 1111 1111 1111 1111 1111 1111 1111 1101 0000 右边补4个0
    反码: 1111 1111 1111 1111 1111 1111 1111 1100 1111
    原码: 1000 0000 0000 0000 0000 0000 0000 0011 0000
*/
        运算用补码，看结果用原码

```

(2) 右移: >>

快速运算：类似于除以2的n次，如果不能整除，向下取整

69>>4 类似于 $69/2^4 = 69/16 = 4$

```

/*
69的二进制: 0000 0000 0000 0000 0000 0000 0100 0101
69>>4: 0000 0000 0000 0000 0000 0000 0100 0101
*/
        正数左边补4个0 右边移出去4位

```

-69>>4 类似于 $-69/2^4 = -69/16 = -5$

```

/*
-69的二进制:
    补码: 1000 0000 0000 0000 0000 0000 0100 0101
    反码: 1111 1111 1111 1111 1111 1111 1011 1010
    补码: 1111 1111 1111 1111 1111 1111 1011 1011
-69>>4: 1111 1111 1111 1111 1111 1111 1111 1011 1011
负数左边补4个1 补码: 1111 1111 1111 1111 1111 1111 1111 1111 1011 右边移出去4位
    反码: 1111 1111 1111 1111 1111 1111 1111 1111 1010
    原码: 1000 0000 0000 0000 0000 0000 0000 0101
*/

```

(3) 无符号右移: >>>

运算规则：往右移动后，左边空出来的位直接补0，不看符号位

正数：和右移一样

负数：右边移出去几位，左边补几个0，结果变为正数

69>>>4 类似于 $69/2$ 的4次 = $69/16 = 4$

```
/*
69的二进制: 0000 0000 0000 0000 0000 0000 0100 0101
69>>>4:   0000 0000 0000 0000 0000 0000 0100 0101
*/           左边补4个0           右边移出去4位
```

-69>>>4 结果: 268435451

```
/*
-69的二进制:
    补码: 1000 0000 0000 0000 0000 0000 0100 0101
    反码: 1111 1111 1111 1111 1111 1111 1011 1010
    补码: 1111 1111 1111 1111 1111 1111 1011 1011
-69>>>4:   0000 1111 1111 1111 1111 1111 1111 1011 1011
无符号右移 补码: 0000 1111 1111 1111 1111 1111 1111 1011 右边移出
左边补4个0 反码: 0000 1111 1111 1111 1111 1111 1111 1011 去4位
原码: 0000 1111 1111 1111 1111 1111 1111 1111 1011
最高位是0, 是正数, 那么原码, 反码, 补码一样
*/           计算用补码, 看结果用原码
```

(4) 按位与: &

运算规则: 对应位都是1才为1

1 & 1 结果为1

1 & 0 结果为0

0 & 1 结果为0

0 & 0 结果为0

9&7 = 1

```
/*
9的二进制: 0000 0000 0000 0000 0000 0000 0000 1001
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0111
9&7:        0000 0000 0000 0000 0000 0000 0000 0001
*/
```

-9&7 = 7

```

/*
-9的二进制:
    原码: 1000 0000 0000 0000 0000 0000 0000 0000 1001
    反码: 1111 1111 1111 1111 1111 1111 1111 1111 0110
    补码: 1111 1111 1111 1111 1111 1111 1111 1111 0111
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 0111 &
-9&7:      0000 0000 0000 0000 0000 0000 0000 0000 0111
    补码: 0000 0000 0000 0000 0000 0000 0000 0000 0111
    反码: 0000 0000 0000 0000 0000 0000 0000 0000 0111
    原码: 0000 0000 0000 0000 0000 0000 0000 0000 0111
*/

```

(5) 按位或: |

运算规则: 对应位只要有1即为1

1 | 1 结果为1

1 | 0 结果为1

0 | 1 结果为1

0 & 0 结果为0

9|7 结果: 15

```

/*
9的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 1001
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 0111 |
9 | 7:      0000 0000 0000 0000 0000 0000 0000 0000 1111
*/

```

-9|7 结果: -9

```

/*
-9的二进制:
    原码: 1000 0000 0000 0000 0000 0000 0000 0000 1001
    反码: 1111 1111 1111 1111 1111 1111 1111 1111 0110
    补码: 1111 1111 1111 1111 1111 1111 1111 1111 0111
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 0111
-9|7:      1111 1111 1111 1111 1111 1111 1111 1111 0111
    补码: 1111 1111 1111 1111 1111 1111 1111 1111 0111
    反码: 1111 1111 1111 1111 1111 1111 1111 1111 0110
    原码: 1000 0000 0000 0000 0000 0000 0000 0000 1001
*/

```

(6) 按位异或: ^

运算规则: 对应位一个为1一个为0, 才为1

$1 \wedge 1$ 结果为0

$1 \wedge 0$ 结果为1

$0 \wedge 1$ 结果为1

$0 \wedge 0$ 结果为0

9 \wedge 7 结果为14

```

/*
9的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 1001
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 0111
9 ^ 7:      0000 0000 0000 0000 0000 0000 0000 0000 1110
*/

```

-9 \wedge 7 结果为-16

```

/*
-9的二进制:
    原码: 1000 0000 0000 0000 0000 0000 0000 0000 1001
    反码: 1111 1111 1111 1111 1111 1111 1111 1111 0110
    补码: 1111 1111 1111 1111 1111 1111 1111 1111 0111 ^

7的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 0111

-9^7:      1111 1111 1111 1111 1111 1111 1111 1111 0000
    补码: 1111 1111 1111 1111 1111 1111 1111 1111 0000
    反码: 1111 1111 1111 1111 1111 1111 1111 1110 1111
    原码: 1000 0000 0000 0000 0000 0000 0001 0000

*/

```

(7) 按位取反: ~

运算规则: ~0就是1

~1就是0

~9 结果: -10

```

/*
9的二进制: 0000 0000 0000 0000 0000 0000 0000 0000 1001 取反
~9:      1111 1111 1111 1111 1111 1111 1111 1111 0110
    补码: 1111 1111 1111 1111 1111 1111 1111 1111 0110
    反码: 1111 1111 1111 1111 1111 1111 1111 1111 0101
    原码: 1000 0000 0000 0000 0000 0000 0000 1010

*/

```

--9 结果: 8

```

/*
-9的二进制:
    原码: 1000 0000 0000 0000 0000 0000 0000 0000 1001
    反码: 1111 1111 1111 1111 1111 1111 1111 1111 0110
    补码: 1111 1111 1111 1111 1111 1111 1111 1111 0111 取反

~~9:      0000 0000 0000 0000 0000 0000 0000 0000 1000
    补码: 0000 0000 0000 0000 0000 0000 0000 0000 1000
    反码: 0000 0000 0000 0000 0000 0000 0000 0000 1000
    原码: 0000 0000 0000 0000 0000 0000 0000 1000

*/

```

2.10.6 赋值运算符

| 运算符 | 符号解释 |
|------|---|
| = | 将右边的常量值/变量值/表达式的值, 赋值给左边的变量 |
| += | 将左边变量的值和右边的常量值/变量值/表达式的值进行相加, 最后将结果赋值给左边的变量 |
| -= | 将左边变量的值和右边的常量值/变量值/表达式的值进行相减, 最后将结果赋值给左边的变量 |
| *= | 将左边变量的值和右边的常量值/变量值/表达式的值进行相乘, 最后将结果赋值给左边的变量 |
| /= | 将左边变量的值和右边的常量值/变量值/表达式的值进行相除, 最后将结果赋值给左边的变量 |
| %= | 将左边变量的值和右边的常量值/变量值/表达式的值进行相模, 最后将结果赋值给左边的变量 |
| <<= | 将左边变量的值左移右边常量/变量值/表达式的值的相应位, 最后将结果赋值给左边的变量 |
| >>= | 将左边变量的值右移右边常量/变量值/表达式的值的相应位, 最后将结果赋值给左边的变量 |
| >>>= | 将左边变量的值无符号右移右边常量/变量值/表达式的值的相应位, 最后将结果赋值给左边的变量 |
| &= | 将左边变量的值和右边的常量值/变量值/表达式的值进行按位与, 最后将结果赋值给左边的变量 |
| = | 将左边变量的值和右边的常量值/变量值/表达式的值进行按位或, 最后将结果赋值给左边的变量 |
| ^= | 将左边变量的值和右边的常量值/变量值/表达式的值进行按位异或, 最后将结果赋值给左边的变量 |

```
public class OperatorDemo04 {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int c = a + b;

        b += a;// 相当于 b = b + a ;
        System.out.println(a); // 3
        System.out.println(b); // 7
        System.out.println(c); //7

        short s = 3;
        // s = s + 4; 代码编译报错, 因为将int类型的结果赋值给short类型的变量s时, 可能损失精度
        s += 4; // 代码没有报错
        //因为在得到int类型的结果后, JVM自动完成一步强制类型转换, 将int类型强转成short
        System.out.println(s);

        int j = 1;
        j += ++j * j++;//相当于 j = j + (++j * j++);
        System.out.println(j);//5

        int m = 1;
        m <<= 2;
        System.out.println(m);
    }
}
```

- 扩展赋值运算符在将最后的结果赋值给左边的变量前，多做了一步强制类型转换。
- 注意：所有的赋值运算符的=左边一定是一个变量

2.10.7 运算符优先级



| | |
|----------------------|---|
| ++ -- ~ ! | 高 |
| * / % | |
| + | |
| << >> >>> | |
| < > <= >= instanceof | |
| == != | |
| & | |
| ^ | |
| | |
| && | |
| | |
| ? : | |
| = *= /= %= | |
| += -= <<= >>= | |
| >>>= &= ^= = | 低 |

提示说明：

- (1) 表达式不要太复杂
- (2) 先算的使用()

口诀：

单目运算排第一；
 乘除余二加减三；
 移位四，关系五；
 等和不等排第六；
 位与、异或和位或；
 短路与和短路或；
 依次从七到十一；
 条件排在第十二；
 赋值一定是最后；

2.10.8 标点符号

在Java中一共有12个标点符号。 (后面再一一学习)

3.11 Separators

Twelve tokens, formed from ASCII characters, are the *separators* (punctuators).

Separator:

(one of)

() { } [] ; , + ... @ ::

- 小括号()用于强制类型转换、表示优先运算表达式、方法参数列表
- 大括号{}用于数组元素列表、类体、方法体、复合语句代码块边界符
- 中括号[]用于数组
- 分号;用于结束语句
- 逗号,用于多个赋值表达式的分隔符和方法参数列表分隔符
- 英文句号.用于成员访问和包目录结构分隔符
- 英文省略号...用于可变参数
- @用于注解
- 双冒号::用于方法引用

第3章 流程控制语句结构

3.1 表达式和语句

常量、变量代表数据。由变量或常量 + 运算符构成的计算表达式。

但其实表达式一共分为三种：

- (1) 变量或常量 + 运算符构成的计算表达式
- (2) new 表达式，结果是一个数组或类的对象。 (后面讲)
- (3) 方法调用表达式，结果是方法返回值或void (无返回值)。 (后面讲)

程序的功能是由语句来完成的，语句分为单语句和复合语句。

单语句又分为：

- (1) 空语句，什么功能都没有。它就是单独的一个分号； (==需要避免==)
- (2) 表达式语句，就是表达式后面加分号;

不是所有表达式加分号都能称为一个独立的语句的，只有以下三种表达式加上分号才能构成一个独立的语句。

- new表达式，
- 方法调用表达式，
- 计算表达式中的赋值表达式、自增自减表达式

```
//空语句  
;  
  
//表达式语句  
i++; //自增表达式 + ;  
System.out.println("hello"); //方法调用表达式 + ;
```

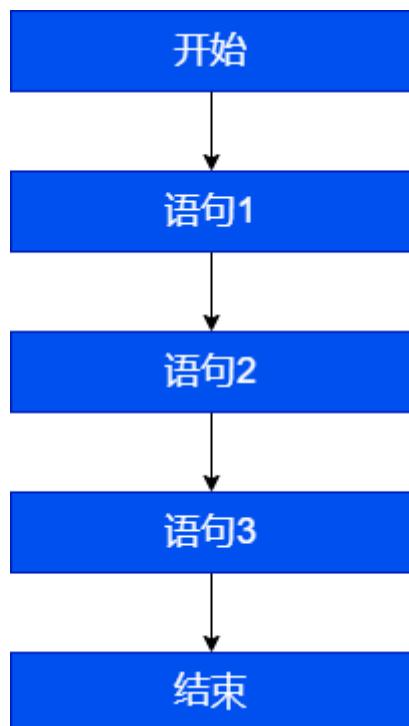
复合语句分为：

- (1) 分支语句：if...else, switch...case
- (2) 循环语句：for,while,do...while
- (3) 跳转语句：break,continue,return,throw
- (4) try语句：try...catch...finally (后面学习)
- (5) 同步语句：synchronized (后面学习)

不同的语句执行顺序和效果是不同的，下面我们一一学习它们。

3.2 顺序结构

顺序结构就是程序从上到下逐行地执行。表达式语句都是顺序执行的。并且上一行对某个变量的修改对下一行会产生影响。



```
public class TestStatement{  
    public static void main(String[] args){  
        int x = 1;  
        int y = 2;  
        System.out.println("x = " + x);  
        System.out.println("y = " + y);  
        //对x、y的值进行修改
```

```

    x++;
    y = 2 * x + y;
    x = x * 10;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
}
}

```

3.3 输入输出语句

3.3.1 输出语句

1、两种常见的输出语句（基础阶段常用）

- **换行输出语句**：输出内容后进行换行，格式如下：

```
System.out.println(输出内容); //输出内容之后，紧接着换行
```

| | |
|---|---------------------|
| <code>void <u>println</u>()</code> | 通过写入行分隔符字符串终止当前行。 |
| <code>void <u>println</u>(boolean x)</code> | 打印 boolean 值，然后终止行。 |
| <code>void <u>println</u>(char x)</code> | 打印字符，然后终止该行。 |
| <code>void <u>println</u>(char[] x)</code> | 打印字符数组，然后终止该行。 |
| <code>void <u>println</u>(double x)</code> | 打印 double，然后终止该行。 |
| <code>void <u>println</u>(float x)</code> | 打印 float，然后终止该行。 |
| <code>void <u>println</u>(int x)</code> | 打印整数，然后终止该行。 |
| <code>void <u>println</u>(long x)</code> | 打印 long，然后终止该行。 |
| <code>void <u>println</u>(Object x)</code> | 打印 Object，然后终止该行。 |
| <code>void <u>println</u>(String x)</code> | 打印 String，然后终止该行。 |

- **不换行输出语句**：输出内容后不换行，格式如下

```
System.out.print(输出内容); //输出内容之后不换行
```

| | |
|-------------------------------------|---------------|
| <code>void print(boolean b)</code> | 打印 boolean 值。 |
| <code>void print(char c)</code> | 打印字符。 |
| <code>void print(char[] s)</code> | 打印字符数组。 |
| <code>void print(double d)</code> | 打印双精度浮点数。 |
| <code>void print(float f)</code> | 打印浮点数。 |
| <code>void print(int i)</code> | 打印整数。 |
| <code>void print(long l)</code> | 打印 long 整数。 |
| <code>void print(Object obj)</code> | 打印对象。 |
| <code>void print(String s)</code> | 打印字符串。 |

示例代码：

```
public class TestPrintlnAndPrint {
    public static void main(String[] args) {
        String name = "柴林燕";
        int age = 18;

        //对比如下两组代码:
        System.out.println(name);
        System.out.println(age);

        System.out.print(name);
        System.out.print(age);
        System.out.println(); //()里面为空, 效果等同于换行, 输出一个换行符
        //等价于 System.out.print("\n"); 或 System.out.print('\n');
        //System.out.print();//错误, ()里面不能为空 核心类库PrintStream类中没有提供print()这样的方法

        //对比如下两组代码:
        System.out.print("姓名: " + name + ",");//""中的内容会原样显示
        System.out.println("年龄: " + age);//""中的内容会原样显示

        System.out.print("name = " + name + ",");
        System.out.println("age = " + age);
    }
}
```

注意事项：

换行输出语句，括号内可以什么都不写，只做换行处理

不换行输出语句，括号内什么都不写的话，编译报错

如果()中有多项内容，那么必须使用 + 连接起来

如果某些内容想要原样输出，就用""引起来，而要输出变量中的内容，则不要把变量名用""引起来

2、格式化输出（了解）

- %d: 十进制整数
- %f: 浮点数
- %c: 单个字符
- %b: boolean值
- %s: 字符串
-

```
public class TestPrintf {  
    public static void main(String[] args) {  
        byte b = 127;  
        int age = 18;  
        long bigNum = 123456789L;  
        float weight = 123.4567F;  
        double money = 589756122.22552;  
        char gender = '男';  
        boolean marry = true;  
        String name = "张三";  
        System.out.printf("byte整数: %d, 年龄: %d, 大整数: %d, 身高: %f, 身高: %.1f, 钱: %f, 钱:  
%.2f, 性别: %c, 婚否: %b, 姓名: %s", b, age, bigNum, weight, weight, money, money, gender, marry, name);  
    }  
}
```

3.3.2 输入语句

键盘输入代码的四个步骤：

- 1、申请资源，创建Scanner类型的对象
- 2、提示输入xx
- 3、接收输入内容
- 4、全部输入完成之后，释放资源，归还资源

1、输入各种类型的数据

```
import java.util.Scanner;  
  
/*  
控制台键盘输入：  
1、先声明一个Scanner类型的变量，并赋值一个对象  
Scanner: 是一个文本扫描仪类型，它的全名称是java.util.Scanner，它是核心类库中定义好的，可以直接用  
Scanner是一个类，不是基本数据类型，是引用数据类型，所以必须给它赋值一个“对象”。  
*/
```

2、建议大家写，提示用户输入xx

不写不会错，不够“友好”

3、从控制台接收用户数的一个xx数据，并且赋值给合适的变量

```
int 变量 = input.nextInt(); //input是Scanner类型的变量，上面叫什么名，下面就用什么名
double 变量 = input.nextDouble();
long 变量 = input.nextLong();
boolean 变量 = input.nextBoolean();
String 变量 = input.next();
char 变量 = input.next().charAt(0);
```

如果要接收数据的变量的数据类型和用户输入的数据的数据类型不符合，会报 InputMismatchException输入不匹配错误

4、关闭IO流

```
/*
public class TestInput {
    public static void main(String[] args) {
        //全名称使用法
//        java.util.Scanner input = new java.util.Scanner(System.in);

        Scanner input = new Scanner(System.in);
        //这句代码唯一可以改的是input，它是一个变量名

        System.out.print("请输入一个整数: ");//先执行，先显示这句话
        int num = input.nextInt();//接收键盘输入，它们有顺序要求
        System.out.println("num = " + num);

        System.out.print("请输入一个小数: ");
        double d = input.nextDouble();
        System.out.println("d = " + d);

        System.out.print("请输入一个布尔值: ");
        boolean b = input.nextBoolean();
        System.out.println("b = " + b);

        System.out.print("请输入一个大整数");
        long big = input.nextLong();
        System.out.println("big = " + big);

        System.out.print("请输入一个字符串");
        String str = input.next();
        System.out.println("str = " + str);

        System.out.print("请输入单个字符: ");
        char c = input.next().charAt(0);
        /*
        input.next(): 接收一个字符串，很多个字符
        input.next().charAt(0): 表示从多个字符中取1个字符，取第1个

        charAt(0) : 表示取一个字符串的第一个
        charAt(1) : 表示取一个字符串的第二个
        charAt(2) : 表示取一个字符串的第三个
        ...
    }
}
```

```

如果输入的字符串的个数少于你要获取的位置，就报错StringIndexOutOfBoundsException字符串下标越界
*/
System.out.println("c = " + c);

input.close(); //建议大家记得它，代码没有错误，但是会造成JVM以外的操作系统相关内存没有得到是否
}
}

```

2、next()与nextLine()的区别

```

import java.util.Scanner;

/*
键盘输入：
next()与nextLine()的区别？

next(), 读取输入的数据时，遇到空格等空白字符，就认为本次数据输入结束
nextLine(), 读取输入的数据时，遇到回车换行才认为结束

```

上一个接收输入的语句是nextInt(), next(), nextDouble()....
 下一个紧接着的接收输入的语句是nextLine()
 这个时间发现第二个输入语句，还没有输入呢，就结束了。

因为：nextInt(), next(), nextDouble().... 它没有读取 回车换行符，数据通道（IO流）中还有回车换行符，那么下一个nextLine()，一看通道中有 回车换行符，就以为输入结束了。

结论：

如果字符串中不会包含空格，那么建议大家使用next()更好。

如果字符串中想要包含空格，那么nextLine()的前面还有其他输入的话，加一句input.nextLine()把前面的回车换行读取掉。

```

*/
public class TestNextAndNextLine {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("请输入年龄：");
        int age = input.nextInt();
        System.out.println("age = " + age);
        input.nextLine(); //左边不用变量不用接收，目的只是把年龄后面的回车换行符读取掉
        //当下面name用的是nextLine()，就需要加这句代码，如果下面name用的是next()就不
        //用加这句代码。

        System.out.print("请输入一个姓名：");
        String name = input.nextLine();
        /*
        next()方法：
        张三          name = "张三";
        张 三，认为张后面空格，就是结束了，而不是回车换行结束  name = "张";

        nextLine()方法：
        张三

```

```
张 三  
都可以接收  
*/  
  
System.out.println("name = " + name);  
  
input.close();  
}  
}
```

3.4 分支语句

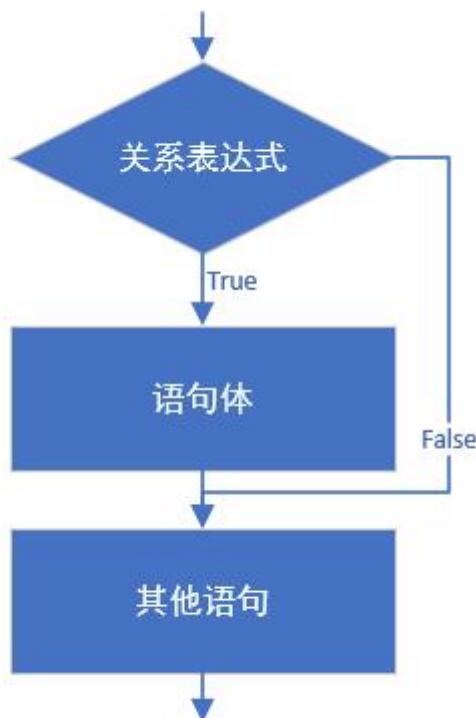
3.4.1 单分支条件判断：if

- if语句第一种格式：if

```
if(条件表达式) {  
    语句体;  
}
```

- 执行流程

- 首先判断条件表达式看其结果是true还是false
- 如果是true就执行语句体
- 如果是false就不执行语句体



案例：从键盘第一个小的整数赋值给small，第二个大的整数赋值给big，如果输入的第一个整数大于第二个整数，就交换。输出显示small和big变量的值。

```
import java.util.Scanner;

public class Test09If {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("请输入第一个整数: ");
        int small = input.nextInt();

        System.out.print("请输入第二个整数: ");
        int big = input.nextInt();

        if (small > big) {
            int temp = small;
            small = big;
            big = temp;
        }
        System.out.println("small=" + small + ",big=" + big);

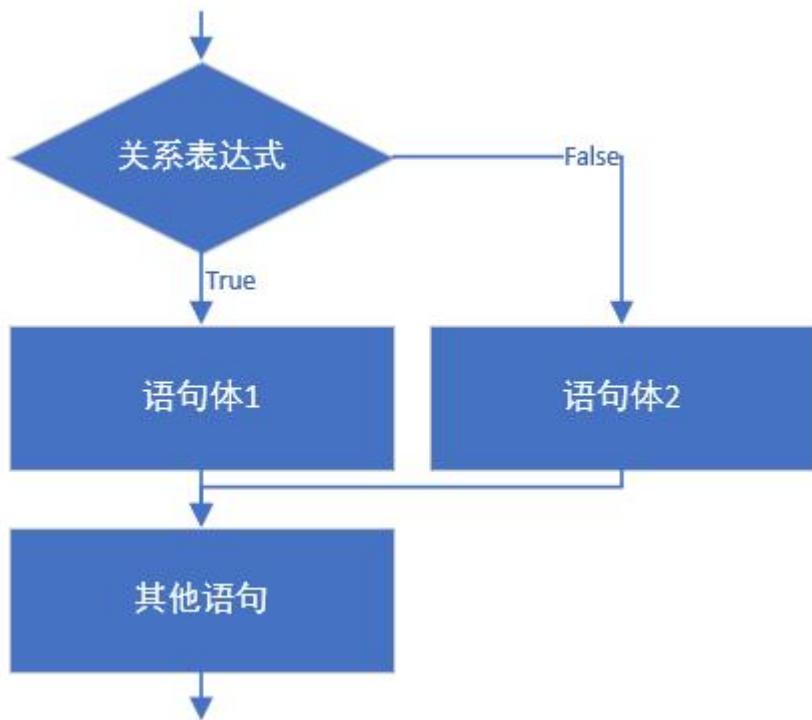
        input.close();
    }
}
```

3.4.2 双分支条件判断：if...else

- **if语句第二种格式：** if...else

```
if(关系表达式) {
    语句体1;
} else {
    语句体2;
}
```

- 执行流程
 - 首先判断关系表达式看其结果是true还是false
 - 如果是true就执行语句体1
 - 如果是false就执行语句体2



案例：从键盘输入一个整数，判定是偶数还是奇数

```

import java.util.Scanner;

public class Test10IfElse {
    public static void main(String[] args){
        // 判断给定的数据是奇数还是偶数
        Scanner input = new Scanner(System.in);

        System.out.print("请输入整数: ");
        int a = input.nextInt();

        if(a % 2 == 0) {
            System.out.println(a + "是偶数");
        } else{
            System.out.println(a + "是奇数");
        }

        input.close();
    }
}
  
```

3.4.3 多分支条件判断：if...else if

- **if语句第三种格式：** if...else if ...else

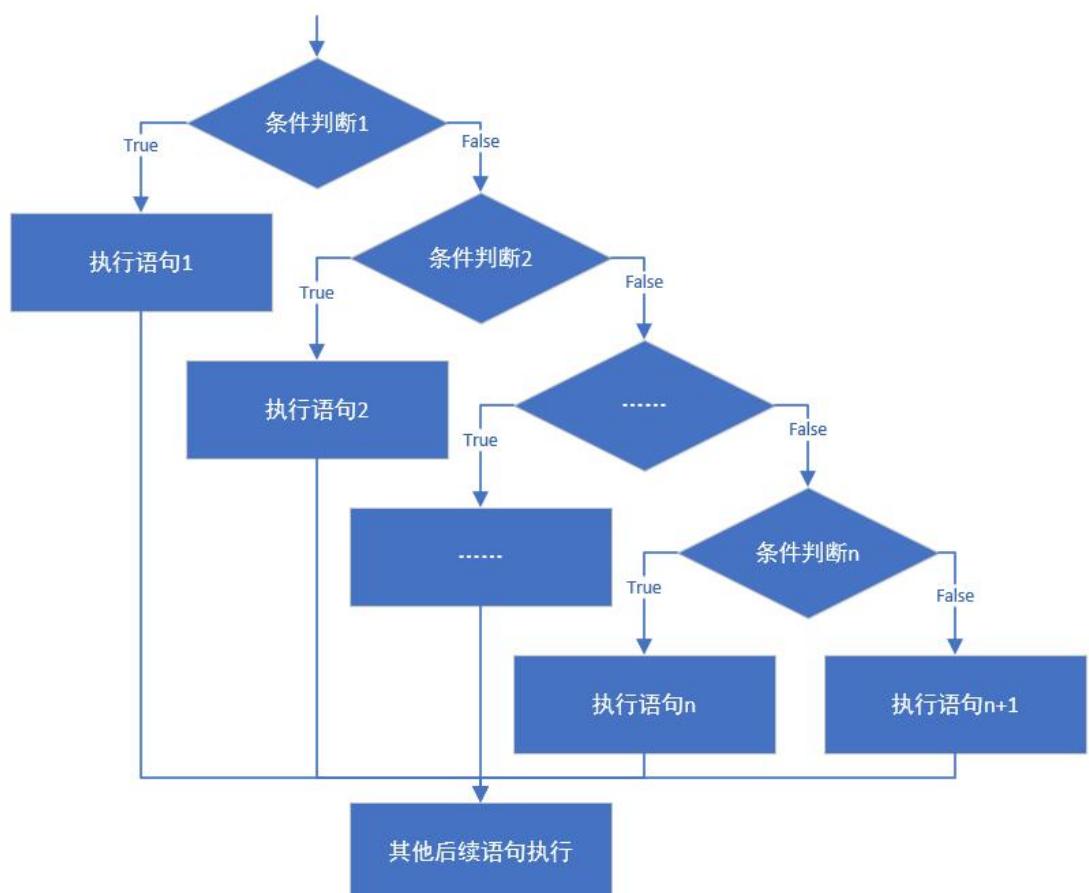
```

if (判断条件1) {
    执行语句1;
} else if (判断条件2) {
    执行语句2;
}
...
else if (判断条件n) {
    执行语句n;
} else {
    执行语句n+1;
}

```

• 执行流程

- 首先判断关系表达式1看其结果是true还是false
- 如果是true就执行语句体1，然后结束当前多分支
- 如果是false就继续判断关系表达式2看其结果是true还是false
- 如果是true就执行语句体2，然后结束当前多分支
- 如果是false就继续判断关系表达式...看其结果是true还是false
- ...
- 如果没有任何关系表达式为true，就执行语句体n+1，然后结束当前多分支。



案例：通过指定考试成绩，判断学生等级，成绩范围[0,100]

- 90-100 优秀
- 80-89 好
- 70-79 良
- 60-69 及格
- 60以下 不及格

```

import java.util.Scanner;

public class Test11IfElseIf {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入成绩[0,100]: ");
        int score = input.nextInt();

        if(score<0 || score>100){
            System.out.println("你的成绩是错误的");
        }else if(score>=90 && score<=100){
            System.out.println("你的成绩属于优秀");
        }else if(score>=80 && score<90){
            System.out.println("你的成绩属于好");
        }else if(score>=70 && score<80){
            System.out.println("你的成绩属于良");
        }else if(score>=60 && score<70){
            System.out.println("你的成绩属于及格");
        }else {
            System.out.println("你的成绩属于不及格");
        }

        input.close();
    }
}

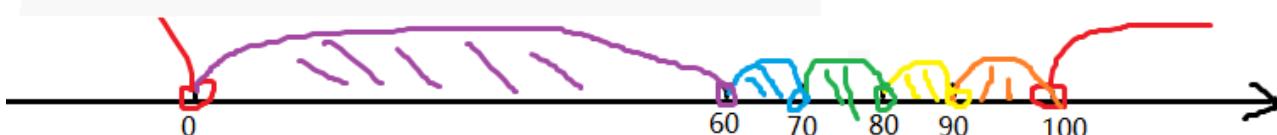
```

```

if(score<0 || score>100){
    System.out.println("你的成绩是错误的");
}else if(score>=90 && score<=100){
    System.out.println("你的成绩属于优秀");
}else if(score>=80 && score<90){
    System.out.println("你的成绩属于好");
}else if(score>=70 && score<80){
    System.out.println("你的成绩属于良");
}else if(score>=60 && score<70){
    System.out.println("你的成绩属于及格");
}else {
    System.out.println("你的成绩属于不及格");
}

```

条件之间没有交集
各个条件顺序可以换，
除了最后的else



```

import java.util.Scanner;

public class Test11IfElseIf {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入成绩[0,100]: ");
        int score = input.nextInt();

        if(score<0 || score>100){
            System.out.println("你的成绩是错误的");
        }else if(score>=90){
            System.out.println("你的成绩属于优秀");
        }else if(score>=80){
            System.out.println("你的成绩属于好");
        }else if(score>=70){
            System.out.println("你的成绩属于良");
        }else if(score>=60){
            System.out.println("你的成绩属于及格");
        }else {
            System.out.println("你的成绩属于不及格");
        }

        input.close();
    }
}

```

```

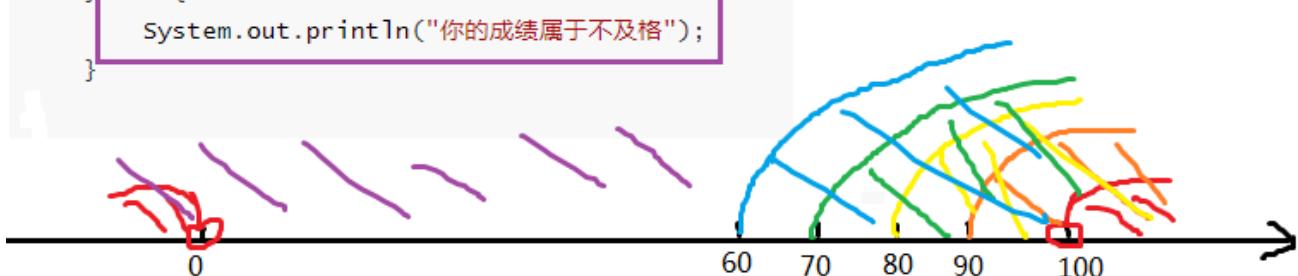
if(score<0 || score>100){
    System.out.println("你的成绩是错误的");
}else if(score>=90){
    System.out.println("你的成绩属于优秀");
}else if(score>=80){
    System.out.println("你的成绩属于好");
}else if(score>=70){
    System.out.println("你的成绩属于良");
}else if(score>=60){
    System.out.println("你的成绩属于及格");
}else {
    System.out.println("你的成绩属于不及格");
}

```

如果单从if条件范围来看，各个条件范围有互相覆盖部分

那么各个条件顺序不能调换

总的原则是：
范围小的在上，大的在下



3.4.4 if..else嵌套

在if的语句块中，或者是在else语句块中，又包含了另外一个条件判断（可以是单分支、双分支、多分支）

执行的特点：（1）如果是嵌套在if语句块中的，只有当外部的if条件满足，才会去判断内部的条件。（2）如果是嵌套在else语句块中的，只有当外部的if条件不满足，进入else后，才会去判断内部的条件。

案例：从键盘输入一个年份值和月份值，输出该月的总天数

要求：年份为正数，月份1-12。

例如：输入2022年5月，总天数是31天。

输入2022年2月，总天数是28天。

输入2020年2月，总天数是29天。

```
import java.util.Scanner;

public class Test12NestIfElse {
    public static void main(String[] args){
        //从键盘输入一个年份和月份
        Scanner input = new Scanner(System.in);

        System.out.print("年份: ");
        int year = input.nextInt();

        System.out.print("月份: ");
        int month = input.nextInt();

        if(year>0){
            if(month>=1 && month<=12){
                //合法的情况
                int days;
                if(month==2){
                    if(year%4==0 && year%100!=0 || year%400==0){
                        days = 29;
                    }else{
                        days = 28;
                    }
                }else if(month==4 || month==6 || month==9 || month==11){
                    days = 30;
                }else{
                    days = 31;
                }
                System.out.println(year+"年" + month + "月有" + days +"天");
            }else{
                System.out.println("月份输入不合法");
            }
        }else{
            System.out.println("年份输入不合法");
        }

        input.close();
    }
}
```

3.4.5 switch...case多分支选择结构

语法格式：

```
switch(表达式){  
    case 常量值1:  
        语句块1;  
        【break;】  
    case 常量值2:  
        语句块2;  
        【break;】  
    . . .  
    【default:  
        语句块n+1;  
        【break;】  
    】  
}
```

执行过程：

(1) 入口

- ①当switch(表达式)的值与case后面的某个常量值匹配，就从这个case进入；
- ②当switch(表达式)的值与case后面的所有常量值都不匹配，寻找default分支进入；不管default在哪里

(2) 一旦从“入口”进入switch，就会顺序往下执行，直到遇到“出口”，即可能发生贯穿

(3) 出口

- ①自然出口：遇到了switch的结束}

- ②中断出口：遇到了break等

注意：

- (1) switch(表达式)的值的类型，只能是：4种基本数据类型 (byte,short,int,char)，两种引用数据类型 (JDK1.5之后枚举、JDK1.7之后String)
- (2) case后面必须是常量值，而且不能重复

1、如何避免case穿透

案例：从键盘输入星期的整数值，输出星期的英文单词

```
import java.util.Scanner;  
  
public class Test13SwitchDemo1 {  
    public static void main(String[] args) {  
        //定义指定的星期  
        Scanner input = new Scanner(System.in);  
        System.out.print("请输入星期值: ");  
        int weekday = input.nextInt();  
  
        //switch语句实现选择
```

```

switch(weekday) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("你输入的星期值有误! ");
        break;
}

input.close();
}
}

```

2、利用case的穿透性

在switch语句中，如果case的后面不写break，将出现穿透现象，也就是一旦匹配成功，不会在判断下一个case的值，直接向后运行，直到遇到break或者整个switch语句结束，switch语句执行终止。

练习：根据指定的月份输出对应季节

```

import java.util.Scanner;

/*
 * 需求：指定一个月份，输出该月份对应的季节。
 *      一年有四季
 *      3,4,5 春季
 *      6,7,8 夏季
 *      9,10,11 秋季
 *      12,1,2 冬季
 */
public class Test14SwitchDemo2 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入月份: ");
        int month = input.nextInt();
    }
}

```

```
/*
    switch(month) {
case 1:
    System.out.println("冬季");
    break;
case 2:
    System.out.println("冬季");
    break;
case 3:
    System.out.println("春季");
    break;
case 4:
    System.out.println("春季");
    break;
case 5:
    System.out.println("春季");
    break;
case 6:
    System.out.println("夏季");
    break;
case 7:
    System.out.println("夏季");
    break;
case 8:
    System.out.println("夏季");
    break;
case 9:
    System.out.println("秋季");
    break;
case 10:
    System.out.println("秋季");
    break;
case 11:
    System.out.println("秋季");
    break;
case 12:
    System.out.println("冬季");
    break;
default:
    System.out.println("你输入的月份有误");
    break;
}

// 改进版
switch(month) {
case 1:
case 2:
case 12:
    System.out.println("冬季");
    break;
case 3:
```

```

        case 4:
        case 5:
            System.out.println("春季");
            break;
        case 6:
        case 7:
        case 8:
            System.out.println("夏季");
            break;
        case 9:
        case 10:
        case 11:
            System.out.println("秋季");
            break;
        default:
            System.out.println("你输入的月份有误");
            break;
    }

    input.close();
}
}

```

常见错误实现1：

```

switch(month){
    case 3|4|5://3|4|5 用了位运算符, 11 | 100 | 101结果是 111是7
        System.out.println("春季");
        break;
    case 6|7|8://6|7|8用了位运算符, 110 | 111 | 1000结果是1111是15
        System.out.println("夏季");
        break;
    case 9|10|11://9|10|11用了位运算符, 1001 | 1010 | 1011结果是1011是11
        System.out.println("秋季");
        break;
    case 12|1|2://12|1|2 用了位运算符, 1100 | 1 | 10 结果是1111, 是15
        System.out.println("冬季");
        break;
    default:
        System.out.println("输入有误");
}

```

常见错误实现2：

```

//编译不通过
switch(month){
    case 3,4,5:
        System.out.println("春季");
        break;
    case 6,7,8:
        System.out.println("夏季");

```

```

        break;
    case 9,10,11:
        System.out.println("秋季");
        break;
    case 12,1,2:
        System.out.println("冬季");
        break;
    default:
        System.out.println("输入有误");
}

```

3、Java12之后switch新特性（选讲）

Switch 表达式也是作为预览语言功能的第一个语言改动被引入Java12 中，开始支持如下写法：

```

switch(month) {
    case 3,4,5 -> System.out.println("春季");
    case 6,7,8 -> System.out.println("夏季");
    case 9,10,11 -> System.out.println("秋季");
    case 12,1,2 -> System.out.println("冬季");
    default->System.out.println("月份输入有误! ");
}

```

4、if语句与switch语句比较

- if语句的条件是一个布尔类型值，if条件表达式为true则进入分支，可以用于范围的判断，也可以用于等值的判断，使用范围更广。
- switch语句的条件是一个常量值 (byte,short,int,char,枚举,String)，只能判断某个变量或表达式的结果是否等于某个常量值，使用场景较狭窄。
- 当条件是判断某个变量或表达式是否等于某个固定的常量值时，使用if和switch都可以，习惯上使用switch更多。当条件是区间范围的判断时，只能使用if语句。
- 另外，使用switch可以利用穿透性，同时执行多个分支，而if...else没有穿透性。

案例1：使用if、switch都可以

使用if实现根据指定的月份输出对应季节

```

import java.util.Scanner;

/*
 * 需求：定义一个月份，输出该月份对应的季节。
 *      一年有四季
 *          3,4,5    春季
 *          6,7,8    夏季
 *          9,10,11 秋季
 *          12,1,2  冬季
 *
 * 分析：
 *      A:指定一个月份
 *      B:判断该月份是几月，根据月份输出对应的季节
 *          if
 *          switch

```

```
/*
public class Test15IforswitchDemo1 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入月份: ");
        int month = input.nextInt();

        /*
         if (month == 1) {
             System.out.println("冬季");
         } else if (month == 2) {
             System.out.println("冬季");
         } else if (month == 3) {
             System.out.println("春季");
         } else if (month == 4) {
             System.out.println("春季");
         } else if (month == 5) {
             System.out.println("春季");
         } else if (month == 6) {
             System.out.println("夏季");
         } else if (month == 7) {
             System.out.println("夏季");
         } else if (month == 8) {
             System.out.println("夏季");
         } else if (month == 9) {
             System.out.println("秋季");
         } else if (month == 10) {
             System.out.println("秋季");
         } else if (month == 11) {
             System.out.println("秋季");
         } else if (month == 12) {
             System.out.println("冬季");
         } else {
             System.out.println("你输入的月份有误");
         }
     }

// 改进版
if ((month == 1) || (month == 2) || (month == 12)) {
    System.out.println("冬季");
} else if ((month == 3) || (month == 4) || (month == 5)) {
    System.out.println("春季");
} else if ((month == 6) || (month == 7) || (month == 8)) {
    System.out.println("夏季");
} else if ((month == 9) || (month == 10) || (month == 11)) {
    System.out.println("秋季");
} else {
    System.out.println("你输入的月份有误");
}

input.close();
}
```

```
}
```

案例2：使用switch更好

用year、month、day分别存储今天的年、月、日值，然后输出今天是这一年的第几天。

注：判断年份是否是闰年的两个标准，满足其一即可

- 1) 可以被4整除，但不可被100整除
- 2) 可以被400整除

例如：1900, 2200等能被4整除，但同时能被100整除，但不能被400整除，不是闰年

```
public class Test16IforSwitchDemo2 {  
    public static void main(String[] args) {  
        int year = 2021;  
        int month = 12;  
        int day = 18;  
        //判断这一天是当年的第几天==>从1月1日开始，累加到xx月xx日这一天  
        //1) [1,month-1]个月满月天数  
        //2)单独考虑2月份是否是29天（依据是看year是否是闰年）  
        //3)第month个月的day天  
  
        //声明一个变量days，用来存储总天数  
        int days = 0;  
  
        //累加[1,month-1]个月满月天数  
        switch (month) {  
            case 12:  
                //累加的1-11月  
                days += 30; //这个30是代表11月份的满月天数  
                //这里没有break，继续往下走  
            case 11:  
                //累加的1-10月  
                days += 31; //这个31是代表10月份的满月天数  
                //这里没有break，继续往下走  
            case 10:  
                days += 30; //9月  
            case 9:  
                days += 31; //8月  
            case 8:  
                days += 31; //7月  
            case 7:  
                days += 30; //6月  
            case 6:  
                days += 31; //5月  
            case 5:  
                days += 30; //4月  
            case 4:  
                days += 31; //3月  
            case 3:  
                days += 28; //2月
```

```

//在这里考虑是否可能是29天
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0) {
    days++; //多加1天
}
case 2:
    days += 31; //1月
case 1:
    days += day; //第month月的day天
}

//输出结果
System.out.println(year + "年" + month + "月" + day + "日是这一年的第" + days + "天");
}
}

```

案例3：只能使用if

从键盘输入一个整数，判断是正数、负数、还是零。

```

import java.util.Scanner;

public class Test17IforswitchDemo3 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("请输入整数: ");
        int num = input.nextInt();

        if (num > 0) {
            System.out.println(num + "是正整数");
        } else if (num < 0) {
            System.out.println(num + "是负整数");
        } else {
            System.out.println(num + "是零");
        }

        input.close();
    }
}

```

3.5 循环语句

循环语句可以在满足循环条件的情况下，反复执行某一段代码，这段被重复执行的代码被称为循环体语句，当反复执行这个循环体时，需要通过修改循环变量使得循环判断条件为false，从而结束循环，否则循环将一直执行下去，形成死循环。

3.5.1 for循环

for循环语句格式：

```
for(初始化语句①; 循环条件语句②; 迭代语句④){  
    循环体语句③  
}
```

注意：

- (1) for();中的两个；是不能多也不能少
- (2) 循环条件必须是boolean类型

执行流程：

- 第一步：执行初始化语句①，完成循环变量的初始化；
- 第二步：执行循环条件语句②，看循环条件语句的值是true，还是false；
 - 如果是true，执行第三步；
 - 如果是false，循环语句中止，循环不再执行。
- 第三步：执行循环体语句③
- 第四步：执行迭代语句④，针对循环变量重新赋值
- 第五步：根据循环变量的新值，重新从第二步开始再执行一遍

1、使用for循环重复执行某些语句

案例：输出1-5的数字

```
public class Test01For {  
    public static void main(String[] args) {  
        for (int i = 1; i <=5; i++) {  
            System.out.println(i);  
        }  
        /*  
         * 执行步骤：  
         */  
    }  
}
```

思考：

- (1) 使用循环和不使用循环的区别
- (2) 如果要实现输出从5到1呢
- (3) 如果要实现输出从1-100呢，或者1-100之间3的倍数或以3结尾的数字呢

2、变量作用域

案例输出1-5。查看退出循环时i的值

```
public class Test02ForvariablesScope {  
    public static void main(String[] args) {  
        //考虑变量的作用域  
        int i;  
        for (i = 1; i <= 5 ; i++) {  
            System.out.println("i = " + i);  
        }  
        System.out.println("结束循环时i = " + i);  
    }  
}
```

3、死循环

```
for(;;){  
    循环体语句块; //如果循环体中没有跳出循环体的语句，那么就是死循环  
}
```

注意：

- (1) 如果两个;之间写true的话，就表示循环条件成立
- (2) 如果两个;之间的循环条件省略的话，就默认为循环条件成立
- (3) 如果循环变量的值不修改，那么循环条件就会永远成立

案例：实现爱你到永远

```
public class Test03EndlessFor {  
    public static void main(String[] args) {  
        for (; ;){  
            System.out.println("我爱你! ");  
        }  
        // System.out.println("end");//永远无法到达的语句，编译报错  
    }  
}
```

```
public class Test03EndlessFor {  
    public static void main(String[] args) {  
        for (; true;){ //条件永远成立  
            System.out.println("我爱你! ");  
        }  
    }  
}
```

```
public class Test03EndlessFor {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; ){ //循环变量没有修改，条件永远成立，死循环  
            System.out.println("我爱你! ");  
        }  
    }  
}
```

思考一下如下代码执行效果：

```
public class Test03EndlessFor {  
    public static void main(String[] args) {  
        for (int i=1; i>=10; ){ //?? 一次都不执行  
            System.out.println("我爱你! ");  
        }  
    }  
}
```

3.5.2 关键字break

使用场景：终止switch或者当前循环

- 在选择结构switch语句中
- 在循环语句中
- 离开使用场景的存在是没有意义的

案例：从键盘输入一个大于1的自然数，判断它是否是素数 提示：素数是指大于1的自然数中，除了1和它本身以外不能再有其他因数的自然数，即某个素数n，在[2,n-1]范围内没有其他自然数可以把n整除

```
import java.util.Scanner;  
  
public class Test04Break {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
  
        System.out.print("请输入一个整数: ");  
        int num = input.nextInt();  
  
        boolean flag = true;//假设num是素数  
        //找num不是素数的证据  
        for(int i=2; i<num; i++){//i<=Math.sqrt(num);  
            if(num % i ==0){//num被某个i整除了， num就不是素数  
                flag = false;  
                break;//找到其中一个可以把num整除的数，就可以结束了，因为num已经可以判定不是素数了  
            }  
        }  
  
        //只有把[2,num-1]之间的所有数都检查过了，才能下定结论， num是素数  
        if(num >1 && flag){  
            System.out.println(num + "是素数");  
        }else{  
            System.out.println(num + "不是素数");  
        }  
    }  
}
```

3.5.3 while循环

1、while循环语句基本格式：

```
while (循环条件语句①) {  
    循环体语句②;  
}
```

注意：

while(循环条件)中循环条件必须是boolean类型

执行流程：

- 第一步：执行循环条件语句①，看循环条件语句的值是true，还是false；
 - 如果是true，执行第二步；
 - 如果是false，循环语句中止，循环不再执行。
- 第二步：执行循环体语句②；
- 第三步：循环体语句执行完后，重新从第一步开始再执行一遍

1、使用while循环重复执行某些语句

```
import java.util.Scanner;  
  
public class Test05While {  
    public static void main(String[] args) {  
        //输出5次我爱尚硅谷  
        int i = 1;  
        while(i<=5){  
            System.out.println("我爱尚硅谷！");  
            i++;  
        }  
  
        System.out.println("-----");  
        int count = 1;  
        while(true){  
            System.out.println("循环第" + count +"次");  
  
            //当循环次数达到5次之后，结束while循环  
            if(count==5){  
                break;  
            }  
            count++;  
        }  
    }  
}
```

2、死循环

```
while(true){  
    循环体语句;//如果此时循环体中没有跳出循环的语句，就是死循环  
}
```

注意：

- (1) while(true): 常量true表示循环条件永远成立
- (2) while(循环条件), 如果循环条件中的循环变量值不修改, 那么循环条件就会永远成立
- (3) while()中的循环条件不能空着

```
import java.util.Scanner;

public class Test05while {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        int positive = 0;
        int negative = 0;

        while(true){
            System.out.print("请输入整数 (0表示结束) : ");
            int num = input.nextInt();

            if(num > 0){
                positive++;
            }else if(num < 0){
                negative++;
            }else{
                break;
            }
        }
        System.out.println("正数个数: " + positive);
        System.out.println("负数个数: " + negative);

        input.close();
    }
}
```

思考下面代码的执行效果, 为什么?

- 输入0
- 输入1

```
import java.util.Scanner;

public class Test05while {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        int positive = 0;
        int negative = 0;

        System.out.print("请输入整数 (0表示结束) : ");
        int num = input.nextInt();

        while(num != 0){
            if(num > 0){
```

```

        positive++;
    }else if(num < 0){
        negative++;
    }
}
System.out.println("正数个数: " + positive);
System.out.println("负数个数: " + negative);

input.close();
}
}

```

3.5.4 do...while循环

do...while循环语句标准格式：

```

do {
    循环体语句①;
} while (循环条件语句②);

```

注意：

- (1) while(循环条件)中循环条件必须是boolean类型
- (2) do{}while();最后有一个分号
- (3) do...while结构的循环体语句是至少会执行一次，这个和for和while是不一样的

执行流程：

- 第一步：执行循环体语句①；
- 第二步：执行循环条件语句②，看循环条件语句的值是true，还是false；
 - 如果是true，执行第三步；
 - 如果是false，循环语句终止，循环不再执行。
- 第三步：循环条件语句执行完后，重新从第一步开始再执行一遍

1、do...while循环至少执行一次循环体

案例：随机生成一个100以内的数，猜这个随机数是多少？

从键盘输入数，如果大了提示，大了，如果小了，提示小了，如果对了，就不再猜了，并统计一共猜了多少次

提示：随机数 Math.random()

double num = Math.random()// [0,1)的小数

```

import java.util.Scanner;

public class Test07Dowhile {
    public static void main(String[] args) {
        //随机生成一个100以内的整数
    }
}

```

```

/*
Math.random() ==> [0,1)的小数
Math.random()* 100 ==> [0,100)的小数
(int)(Math.random()* 100) ==> [0,100)的整数
*/
int num = (int)(Math.random()* 100);
//System.out.println(num);

//声明一个变量，用来存储猜的次数
int count = 0;

Scanner input = new Scanner(System.in);
int guess;//提升作用域
do{
    System.out.print("请输入100以内的整数: ");
    guess = input.nextInt();

    //输入一次，就表示猜了一次
    count++;

    if(guess > num){
        System.out.println("大了");
    }else if(guess < num){
        System.out.println("小了");
    }
}while(num != guess);

System.out.println("一共猜了: " + count+"次");

input.close();
}
}

```

2、死循环

```

do{
    循环体语句;//如果此时循环体中没有跳出循环的语句，就是死循环
}while(true);

```

注意：

- (1) while(true): 常量true表示循环条件永远成立
- (2) while(循环条件)，如果循环条件中的循环变量值不修改，那么循环条件就会永远成立
- (3) while()中的循环条件不能空着

```

import java.util.Scanner;

public class Test08EndlessDowhile {
    public static void main(String[] args) {
        //随机生成一个100以内的整数
        /*

```

```

        Math.random() ==> [0,1)的小数
        Math.random()* 100 ==> [0,100)的小数
        (int)(Math.random()* 100) ==> [0,100)的整数
    */
    int num = (int)(Math.random()* 100);
    //System.out.println(num);

    //声明一个变量，用来存储猜的次数
    int count = 0;
    Scanner input = new Scanner(System.in);
    do{
        System.out.print("请输入100以内的整数: ");
        int guess = input.nextInt();

        //输入一次，就表示猜了一次
        count++;

        if(guess > num){
            System.out.println("猜大了");
        }else if(guess < num){
            System.out.println("猜小了");
        }else{
            System.out.println("猜对了，一共猜了" + count+"次");
            break;
        }
    }while(true);

    input.close();
}
}

```

3.5.5 循环语句的区别

- 从循环次数角度分析
 - do...while循环至少执行一次循环体语句
 - for和while循环先循环条件语句是否成立，然后决定是否执行循环体，至少执行零次循环体语句
- 如何选择
 - 遍历有明显的循环次数（范围）的需求，选择for循环
 - 遍历没有明显的循环次数（范围）的需求，循环while循环
 - 如果循环体语句块至少执行一次，可以考虑使用do...while循环
 - 本质上：三种循环之间完全可以互相转换，都能实现循环的功能
- 三种循环结构都具有四要素：
 - (1) 循环变量的初始化表达式
 - (2) 循环条件
 - (3) 循环变量的修改的迭代表达式
 - (4) 循环体语句块

3.5.6 循环嵌套

所谓嵌套循环，是指一个循环的循环体是另一个循环。比如for循环里面还有一个for循环，就是嵌套循环。当然可以是三种循环任意互相嵌套。

例如：两个for嵌套循环格式

```
for(初始化语句①; 循环条件语句②; 迭代语句③) {  
    for(初始化语句④; 循环条件语句⑤; 迭代语句⑥) {  
        循环体语句⑦;  
    }  
}
```

执行特点：外循环执行一次，内循环执行一轮。

案例1：打印5行直角三角形

```
*  
**  
***  
****  
*****
```

```
public static void main(String[] args){  
    for (int i = 0; i < 5; i++) {  
        for (int j = 0; j <= i; j++) {  
            System.out.print("*");  
        }  
        System.out.println();  
    }  
}
```

案例2：break结束当层循环

```
for (int i = 1; i <=5 ; i++) {//两层循环  
    for (int j=1; j<=5; j++){  
        System.out.print(i);  
        if(j==i){  
            break;//在内循环中，只能结束每一轮的内循环  
        }  
    }  
    System.out.println();  
}  
/*  
1  
22  
333  
4444  
55555  
*/
```

3.5.7 关键字：continue

使用场景：提前结束本次循环，继续下一次的循环

1、跳过本次循环

分析如下代码运行结果：

```
public class Test10Continue {  
    public static void main(String[] args) {  
        for(int i=1; i<=5; i++){  
            for(int j=1; j<=5; j++){  
                if(i==j){  
                    continue;  
//                    break;  
                }  
                System.out.print(j);  
            }  
            System.out.println();  
        }  
    }  
}
```

2、使用continue提高效率

```
public class Test10Continue {  
    public static void main(String[] args) {  
        //找出1-100之间所有的素数（质数）  
        for(int i=2; i<=100; i++){  
            if(i!=2 && i%2==0 || i!=5 && i%5==0){//偶数一定不是素数,  
                continue;  
            }  
  
            //里面的代码会运行100遍  
            //每一遍i的值是不同的，i=2,3,4,5...100  
            //每一遍都要判断i是否是素数，如果是，就打印i  
            /*  
             * 如何判断i是否是素数  
             * (1) 假设i是素数  
             * boolean flag = true;//true代表素数  
             * (2) 找i不是素数的证据  
             * 如果在[3,i-1]之间只要有一个数能够把i整除了，说明i就不是素数  
             * 修改flag = false;  
             * 这里从3开始找，是因为我们前面排除了偶数  
             * (3) 判断这个flag  
             */  
            // (1) 假设i是素数  
            boolean flag = true;//true代表素数  
            // (2) 找i不是素数的证据  
            for(int j=3; j<i; j++){ // j<=Math.sqrt(i);  
                if(i%j==0){  
                    flag = false;//找到一个就可以了  
                    break;  
                }  
            }  
        }  
    }  
}
```

```
        }
        // (3) 判断这个flag
        if(flag){
            System.out.println(i);
        }
    }
}
```

第4章 数组

4.1 数组的概念

4.1.1 容器概述

需求分析:

现在需要统计某公司员工的工资情况，例如计算平均工资、找到最高工资等。假设该公司有50名员工，用前面所学的知识，程序首先需要声明50个变量来分别记住每位员工的工资，然后在进行操作，这样做会显得很麻烦，而且错误率也会很高。因此我们可以使用容器进行操作。将所有的数据全部存储到一个容器中，统一操作。

容器概念：

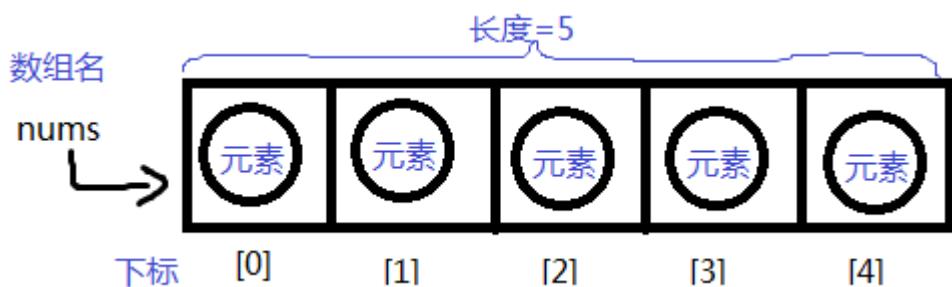
- **生活中的容器**: 水杯(装水等液体), 衣柜(装衣服等物品), 教室(装学生等人员)。
 - **程序中的容器**: 是将多个数据存储到一起, 每个数据称为该容器的元素。

4.1.2 数组的概念

- **数组概念：**数组就是用于存储数据的长度固定的容器，保证多个数据的数据类型要一致。

百度百科中对数组的定义：

所谓**数组**(array)，就是相同数据类型的元素按一定顺序排列的集合，就是把有限个类型相同的变量用一个名字命名，以便统一管理他们，然后用编号区分他们，这个名字称为**数组名**，编号称为**下标或索引**(index)。组成数组的各个变量称为数组的**元素**(element)。数组中元素的个数称为**数组的长度**(length)。



数组的特点：

- 1、数组的长度一旦确定就不能修改
 - 2、创建数组时会在内存中开辟一整块连续的空间。
 - 3、存取元素的速度快，因为可以通过[下标]，直接定位到任意一个元素。

4.1.3 数组的分类

1、按照维度分：

- 一维数组：存储一组数据
- 二维数组：存储多组数据，相当于二维表，一行代表一组数据，这是这里的二维表每一行长度不要求一样。

一维数组：



二维数组：



2、按照元素类型分：

- 基本数据类型的元素：存储数据值
- 引用数据类型的元素：存储对象（本质上存储对象的首地址）（这个在面向对象部分讲解）

注意：无论数组的元素是基本数据类型还是引用数据类型，数组本身都是引用数据类型。

4.2 一维数组的声明与使用

4.2.1 一维数组的声明

- 一维数组的声明/定义格式

```
//推荐  
元素的数据类型[] 二维数组的名称；  
  
//不推荐  
元素的数据类型 二维数组名[]；
```

- 数组的声明，就是要确定：

- (1) 数组的维度：在Java中数组的标点符号是[]，[]表示一维，[][]表示二维
- (2) 数组的元素类型：即创建的数组容器可以存储什么数据类型的数据。元素的类型可以是任意的Java的数据类型。例如：int, String, Student等
- (3) 数组名：就是代表某个数组的标识符，数组名其实也是变量名，按照变量的命名规范来命名。数组名是个引用数据类型的变量，因为它代表一组数据。

- 示例

```
public class Test01ArrayDeclare {
    public static void main(String[] args) {
        //比如，要存储一个小组的成绩
        int[] scores;
        int grades[];
        // System.out.println(scores); //未初始化不能使用

        //比如，要存储一组字母
        char[] letters;

        //比如，要存储一组姓名
        String[] names;

        //比如，要存储一组价格
        double[] prices;

    }
}
```

4.2.2 一维数组的静态初始化

- 什么是初始化?
 - 初始化就是确定数组元素的总个数（即数组的长度）和元素的值
- 什么是静态初始化?
 - 静态初始化就是用静态数据（编译时已知）为数组初始化。此时数组的长度由静态数据的个数决定。
- **一维数组静态初始化格式1:**

数据类型[] 数组名 = {元素1,元素2,元素3...};//必须在一个语句中完成，不能分开两个语句写

例如，定义存储1, 2, 3, 4, 5整数的数组容器

```
int[] arr = {1,2,3,4,5}; //正确

int[] arr;
arr = {1,2,3,4,5}; //错误
```

- **一维数组静态初始化格式2:**

数据类型[] 数组名 = new 数据类型[] {元素1,元素2,元素3...};
或
数据类型[] 数组名;
数组名 = new 数据类型[] {元素1,元素2,元素3...};

例如，定义存储1, 2, 3, 4, 5整数的数组容器。

```
int[] arr = new int[]{1,2,3,4,5};//正确  
  
int[] arr;  
arr = new int[]{1,2,3,4,5};//正确
```

- 一维数组静态初始化演示

```
public class Test02ArrayInitialize {  
    public static void main(String[] args) {  
        int[] arr = {1,2,3,4,5};//右边不需要写new int[]  
  
        int[] nums;  
        nums = new int[]{10,20,30,40}; //声明和初始化在两个语句完成，就不能使用new int[]  
  
        char[] word = {'h','e','l','l','o'};  
  
        String[] names = {"张三", "李四", "王五"};  
  
        System.out.println("arr数组: " + arr); //arr数组: [I@1b6d3586  
        System.out.println("nums数组: " + nums); //nums数组: [I@4554617c  
        System.out.println("word数组: " + word); //word数组: [C@74a14482  
        System.out.println("names数组: " + names); //names数组: [Ljava.lang.String;@1540e19d  
    }  
}
```

4.2.3 一维数组的使用

- 如何获取数组的元素总个数，即数组的长度

数组的长度属性：每个数组都具有长度，而且是固定的，Java中赋予了数组的一个属性，可以获取到数组的长度，语句为：`数组名.length`，属性length的执行结果是数组的长度，int类型结果。

数组名.length

- 如何表示数组中的一个元素？

每一个存储到数组的元素，都会自动的拥有一个编号，从0开始，这个自动编号称为**数组索引(index)**或**下标**，可以通过数组的索引/下标访问到数组中的元素。

数组名[索引/下标]

- 数组的下标范围？

Java中数组的下标从[0]开始，下标范围是[0, 数组的长度-1]，即[0, 数组名.length-1]

- 一维数组的使用演示

```
public class Test03ArrayUse {  
    public static void main(String[] args) {  
        int[] arr = {1,2,3,4,5};
```

```

        System.out.println("arr数组的长度: " + arr.length);
        System.out.println("arr数组的第一个元素: " + arr[0]); //下标从0开始
        System.out.println("arr数组的第二个元素: " + arr[1]);
        System.out.println("arr数组的第三个元素: " + arr[2]);
        System.out.println("arr数组的第4个元素: " + arr[3]);
        System.out.println("arr数组的第5个元素: " + arr[4]);

        //修改第一个元素的值
        //此处arr[0]相当于一个int类型的变量
        arr[0] = 100;
        System.out.println("arr数组的第一个元素: " + arr[0]);
    }
}

```

4.2.4 数组下标越界异常

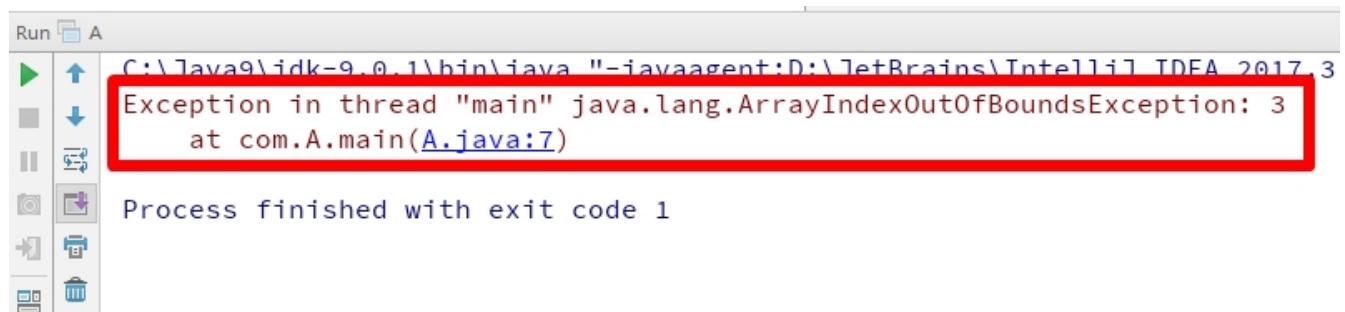
当访问数组元素时，下标指定超出[0, 数组名.length-1]的范围时，就会报数组下标越界异常：
ArrayIndexOutOfBoundsException。

```

public class Test04ArrayIndexOutOfBoundsException {
    public static void main(String[] args) {
        int[] arr = {1,2,3};
        // System.out.println("最后一个元素: " + arr[3]); //错误, 下标越界
        ArrayIndexOutOfBoundsException
        // System.out.println("最后一个元素: " + arr[arr.length]); //错误, 下标越界
        ArrayIndexOutOfBoundsException
        System.out.println("最后一个元素: " + arr[arr.length-1]); //对
    }
}

```

创建数组，赋值3个元素，数组的索引就是0, 1, 2，没有3索引，因此我们不能访问数组中不存在的索引，程序运行后，将会抛出 `ArrayIndexOutOfBoundsException` 数组越界异常。在开发中，数组的越界异常是**不能出现的**，一旦出现了，就必须要修改我们编写的代码。



4.2.4 一维数组的遍历

数组遍历：就是将数组中的每个元素分别获取出来，就是遍历。遍历也是数组操作中的基石。`for`循环与数组的遍历是绝配。

```

public class Test05ArrayIterate {
    public static void main(String[] args) {
        int[] arr = new int[]{1,2,3,4,5};
        //打印数组的属性，输出结果是5
        System.out.println("数组的长度: " + arr.length);

        //遍历输出数组中的元素
        System.out.println("数组的元素有: ");
        for(int i=0; i<arr.length; i++){
            System.out.println(arr[i]);
        }
    }
}

```

4.2.5 一维数组动态初始化

- 什么是动态初始化？

动态初始化就是先确定元素的个数（即数组的长度），而元素此时只是默认值，并不是真正的数据。元素真正的数据需要后续单独一个一个赋值。

- 格式：**

数组存储的数据类型[] 数组名字 = new 数组存储的数据类型[长度]；

或

数组存储的数据类型[] 数组名字;
数组名字 = new 数组存储的数据类型[长度]；

- new：关键字，创建数组使用的关键字。因为数组本身是引用数据类型，所以要用new创建数组对象。
- [长度]：数组的长度，表示数组容器中可以存储多少个元素。
- 注意：数组有定长特性，长度一旦指定，不可更改。**和水杯道理相同，买了一个2升的水杯，总容量就是2升是固定的。

例如，定义可以存储5个整数的数组容器，代码如下：

```

int[] arr = new int[5];

int[] arr;
arr = new int[5];

int[] arr = new int[5]{1,2,3,4,5}; //错误的，后面有{}指定元素列表，就不需要在[]中指定元素个数了。

```

- 一维数组的动态初始化演示

```

public class Test06ArrayInitialize {
    public static void main(String[] args) {
        int[] arr = new int[5];
    }
}

```

```
System.out.println("arr数组的长度: " + arr.length);
System.out.print("存储数据到arr数组之前: [");
for (int i = 0; i < arr.length; i++) {
    if(i==0){
        System.out.print(arr[i]);
    }else{
        System.out.print(", " + arr[i]);
    }
}
System.out.println("]");

//初始化
/*
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;
    arr[3] = 8;
    arr[4] = 10;*/

for (int i = 0; i < arr.length; i++) {
    arr[i] = (i+1) * 2;
}

System.out.print("存储数据到arr数组之后: [");
for (int i = 0; i < arr.length; i++) {
    if(i==0){
        System.out.print(arr[i]);
    }else{
        System.out.print(", " + arr[i]);
    }
}
System.out.println("]");
}
}
```

4.2.6 数组元素的默认值

当我们使用动态初始化方式创建数组时，元素只是默认值。

| 数组元素类型 | 元素默认初始值 |
|---------|-----------------------|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0F |
| double | 0.0 |
| char | 0 或写为: '\u0000'(表现为空) |
| boolean | false |
| 引用类型 | null |

```

public class Test07ArrayElementDefaultValue {
    public static void main(String[] args) {
        //存储26个字母
        char[] letters = new char[26];
        System.out.println("letters数组的长度: " + letters.length);
        System.out.print("存储字母到letters数组之前: [");
        for (int i = 0; i < letters.length; i++) {
            if(i==0){
                System.out.print(letters[i]);
            }else{
                System.out.print(", " + letters[i]);
            }
        }
        System.out.println("]");
    }

    //存储5个姓名
    String[] names = new String[5];
    System.out.println("names数组的长度: " + names.length);
    System.out.print("存储姓名到names数组之前: [");
    for (int i = 0; i < names.length; i++) {
        if(i==0){
            System.out.print(names[i]);
        }else{
            System.out.print(", " + names[i]);
        }
    }
    System.out.println("]");
}
}

```

4.3 一维数组内存分析

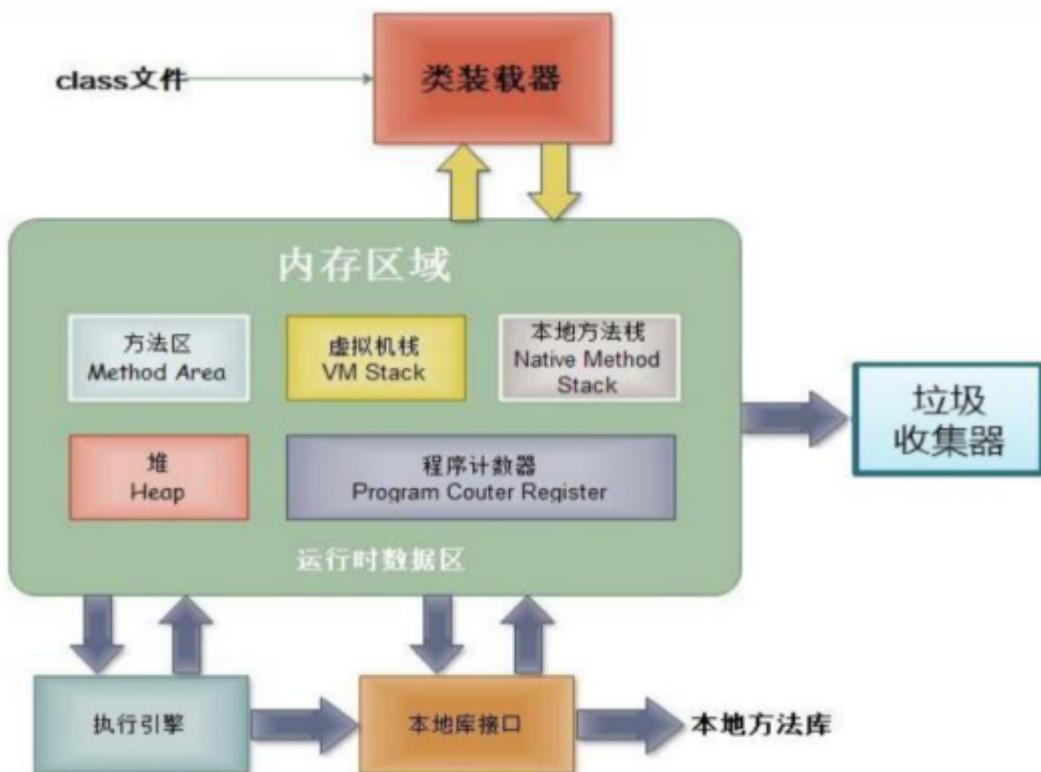
4.3.1 内存概述

内存是计算机中重要的部件之一，它是与CPU进行沟通的桥梁。其作用是用于暂时存放CPU中的运算数据，以及与硬盘等外部存储器交换的数据。只要计算机在运行中，CPU就会把需要运算的数据调到内存中进行运算，当运算完成后CPU再将结果传送出来。我们编写的程序是存放在硬盘中的，在硬盘中的程序是不会运行的，必须放进内存中才能运行，运行完毕后会清空内存。

Java虚拟机要运行程序，必须要对内存进行空间的分配和管理。

4.3.2 Java虚拟机的内存划分

为了提高运算效率，就对空间进行了不同区域的划分，因为每一片区域都有特定的处理数据方式和内存管理方式。



| 区域名称 | 作用 |
|-------|--|
| 程序计数器 | 程序计数器是CPU中的寄存器，它包含每一个线程下一条要执行的指令的地址 |
| 本地方法栈 | 当程序中调用了native的本地方法时，本地方法执行期间的内存区域 |
| 方法区 | 存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。 |
| 堆内存 | 存储对象（包括数组对象），new来创建的，都存储在堆内存。 |
| 虚拟机栈 | 用于存储正在执行的每个Java方法的局部变量表等。局部变量表存放了编译期可知长度的各种基本数据类型、对象引用，方法执行完，自动释放。 |

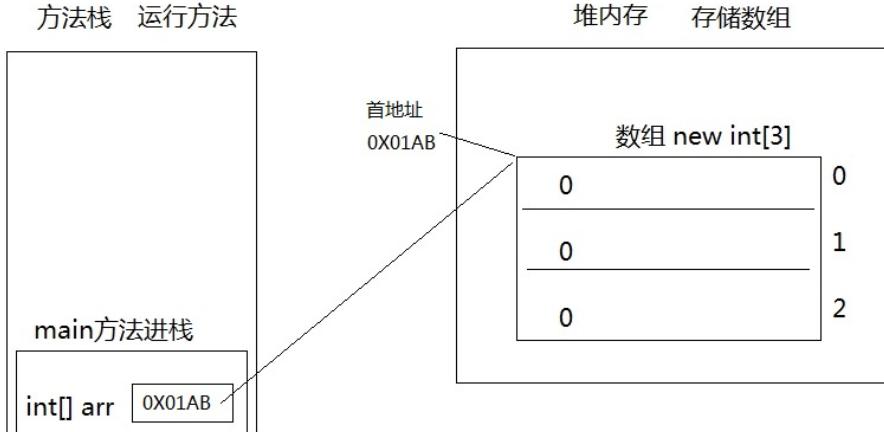
4.4.3 一维数组在内存中的存储

1、一个一维数组内存图

```
public static void main(String[] args) {
    int[] arr = new int[3];
    System.out.println(arr); // [I@5f150435
}
```

程序执行流程：

1. main方法进入方法栈执行
2. 创建数组，JVM会在堆内存中开辟空间，存储数组
3. 数组在内存中会有自己的内存地址，以十六进制数表示
4. 数组中有3个元素，默认值0
5. JVM将数组的内存首地址赋值给引用类型变量arr
6. 变量arr保存的是数组内存中的地址，而不是一个具体是数值，因此称为引用数据类型。



思考：打印arr为什么是[Ljava.util.ArrayList@5f150435，它是数组的地址吗？

答：它不是数组的地址。

问？不是说arr中存储的是数组对象的首地址吗？

答：arr中存储的是数组的首地址，但是因为数组是引用数据类型，打印arr时，会自动调用arr数组对象的toString()方法，该方法默认实现的是对象类型名@该对象的hashCode()值的十六进制值。

问？对象的hashCode值是否就是对象内存地址？

答：不一定，因为这个和不同品牌的JVM产品的具体实现有关。例如：Oracle的OpenJDK中给出了5种实现，其中有一种是直接返回对象的内存地址，但是OpenJDK默认没有选择这种方式。

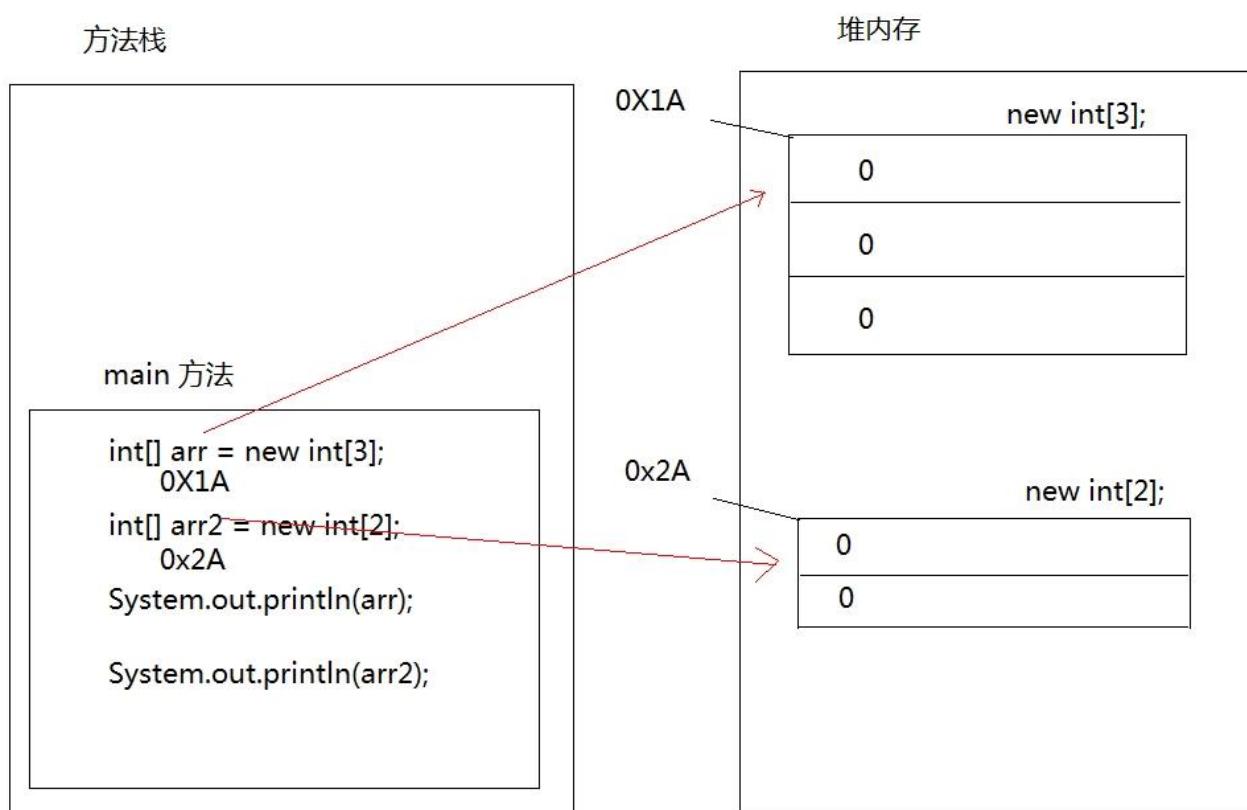
2、数组下标为什么是0开始

因为第一个元素距离数组首地址间隔0个单元格。

3、两个一维数组内存图

两个数组独立

```
public static void main(String[] args) {  
    int[] arr = new int[3];  
    int[] arr2 = new int[2];  
    System.out.println(arr);  
    System.out.println(arr2);  
}
```



4、两个变量指向一个一维数组

两个数组变量本质上代表同一个数组。

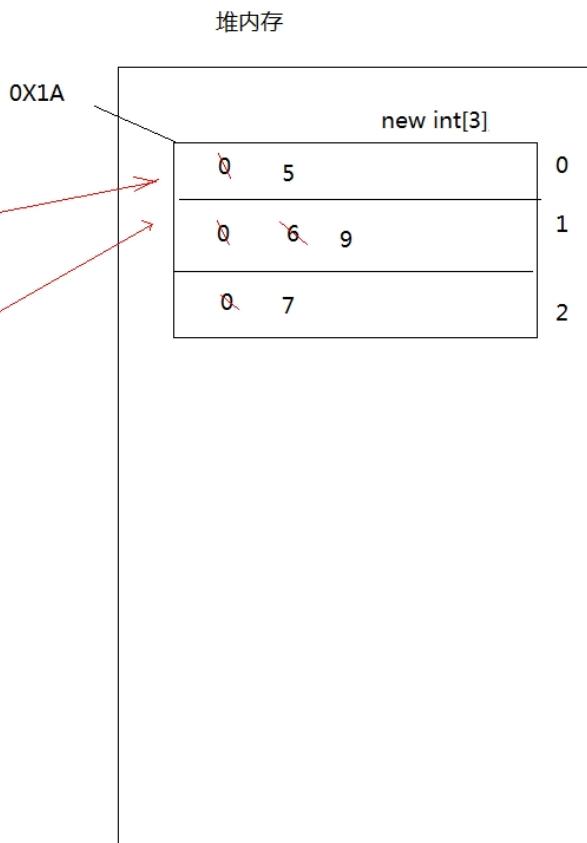
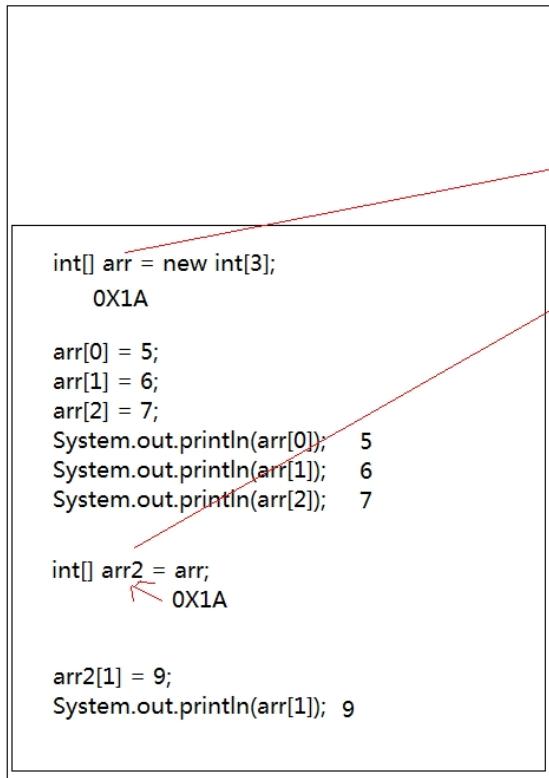
```
public static void main(String[] args) {  
    // 定义数组，存储3个元素  
    int[] arr = new int[3];  
    //数组索引进行赋值
```

```

arr[0] = 5;
arr[1] = 6;
arr[2] = 7;
//输出3个索引上的元素值
System.out.println(arr[0]);
System.out.println(arr[1]);
System.out.println(arr[2]);
//定义数组变量arr2, 将arr的地址赋值给arr2
int[] arr2 = arr;
arr2[1] = 9;
System.out.println(arr[1]);
}

```

方法栈



4.4 一维数组的常见算法

4.4.1 数组统计：求总和、均值、统计偶数个数等

示例代码1：

```

public class Test08ArrayElementSum {
    public static void main(String[] args) {
        int[] arr = {4,5,6,1,9};
        //求总和、均值
        int sum = 0;//因为0加上任何数都不影响结果
        for(int i=0; i<arr.length; i++){
            sum += arr[i];
        }
    }
}

```

```

        double avg = (double)sum/arr.length;

        System.out.println("sum = " + sum);
        System.out.println("avg = " + avg);
    }
}

```

示例代码2：

```

public class Test09ArrayElementMul {
    public static void main(String[] args) {
        int[] arr = {4,5,6,1,9};

        //求总乘积
        long result = 1;//因为1乘以任何数都不影响结果
        for(int i=0; i<arr.length; i++){
            result *= arr[i];
        }

        System.out.println("result = " + result);
    }
}

```

示例代码3：

```

public class Test10ArrayElementEvenCount {
    public static void main(String[] args) {
        int[] arr = {4,5,6,1,9};
        //统计偶数个数
        int evenCount = 0;
        for(int i=0; i<arr.length; i++){
            if(arr[i]%2==0){
                evenCount++;
            }
        }

        System.out.println("evenCount = " + evenCount);
    }
}

```

4.4.2 数组找最值

1、找最大值/最小值

五、思维逻辑题（每题 10 分，共 10 分）
 20、一楼到十楼的每层电梯门口都放着一颗钻石，钻石大小不一。你乘坐电梯从一楼到十楼，每层楼电梯门都会打开一次，手里只能拿一颗钻石，问怎样才能拿到最大的一颗？

思路：

- (1) 先假设第一个元素最大/最小
- (2) 然后用max/min与后面的元素一一比较

示例代码：

```
public class Test11ArrayMax {
    public static void main(String[] args) {
        int[] arr = {4, 5, 6, 1, 9};
        //找最大值
        int max = arr[0];
        for(int i=1; i<arr.length; i++){//此处i从1开始，是max不需要与arr[0]再比较一次了
            if(arr[i] > max){
                max = arr[i];
            }
        }

        System.out.println("max = " + max);
    }
}
```

2、找最值及其第一次出现的下标

思路：

- (1) 先假设第一个元素最大/最小
- (2) 用max/min变量表示最大/小值，用max/min与后面的元素一一比较
- (3) 用index时刻记录目前比对的最大/小的下标

示例代码：

```
public class Test12MaxIndex {
    public static void main(String[] args) {
        int[] arr = {4, 5, 6, 1, 9};
        //找最大值以及第一个最大值下标
        int max = arr[0];
        int index = 0;
        for(int i=1; i<arr.length; i++){
            if(arr[i] > max){
                max = arr[i];
                index = i;
            }
        }

        System.out.println("max = " + max);
        System.out.println("index = " + index);
    }
}
```

或

思路：

(1) 先假设第一个元素最大/最小

(2) 用maxIndex时刻记录目前比对的最大/小的下标，那么arr[maxIndex]就是目前的最大值

```
public class Test12MaxIndex2 {  
    public static void main(String[] args) {  
        int[] arr = {4,5,6,1,9};  
        //找最大值  
        int maxIndex = 0;  
        for(int i=1; i<arr.length; i++){  
            if(arr[i] > arr[maxIndex]){  
                maxIndex = i;  
            }  
        }  
        System.out.println("最大值: " + arr[maxIndex]);  
    }  
}
```

3、找最值及其所有最值的下标（选讲）

有一种情况是元素是重复的，那么最大值就有很多个。

思路：

(1) 先找最大值

①假设第一个元素最大

②用max与后面的元素一一比较

(2) 遍历数组，看哪些元素和最大值是一样的

示例代码：

```
public class Test13AllMaxIndex {  
    public static void main(String[] args) {  
        int[] arr = {4,5,6,1,9,9,3};  
        //找最大值  
        int max = arr[0];  
        for(int i=1; i<arr.length; i++){  
            if(arr[i] > max){  
                max = arr[i];  
            }  
        }  
        System.out.println("最大值是: " + max);  
        System.out.print("最大值的下标有: ");  
  
        //遍历数组，看哪些元素和最大值是一样的  
        for(int i=0; i<arr.length; i++){  
            if(max == arr[i]){  
                System.out.print(i+"\t");  
            }  
        }  
        System.out.println();  
    }  
}
```

```
    }
}
```

优化

```
public class Test13AllMaxIndex2 {
    public static void main(String[] args) {
        int[] arr = {4,5,6,1,9,9,3};
        //找最大值
        int max = arr[0];
        String index = "0";
        for(int i=1; i<arr.length; i++){
            if(arr[i] > max){
                max = arr[i];
                index = i + "";
            }else if(arr[i] == max){
                index += "," + i;
            }
        }

        System.out.println("最大值是" + max);
        System.out.println("最大值的下标是[" + index+"]");
    }
}
```

4.4.3 数组的元素查找

1、顺序查找

顺序查找：挨个查看

要求：对数组元素的顺序没要求

顺序查找示例代码：

```
public class Test14ArrayOrderSearch {
    //查找value第一次在数组中出现的index
    public static void main(String[] args){
        int[] arr = {4,5,6,1,9};
        int value = 1;
        int index = -1;

        for(int i=0; i<arr.length; i++){
            if(arr[i] == value){
                index = i;
                break;
            }
        }

        if(index==-1){
            System.out.println(value + "不存在");
        }
    }
}
```

```

        }else{
            System.out.println(value + "的下标是" + index);
        }
    }
}

```

2、二分查找

```

import java.util.Scanner;

public class Test15ArrayBinarySearch {
    public static void main(String[] args){
        //数组一定是有序的
        int[] arr = {8,15,23,35,45,56,75,85};

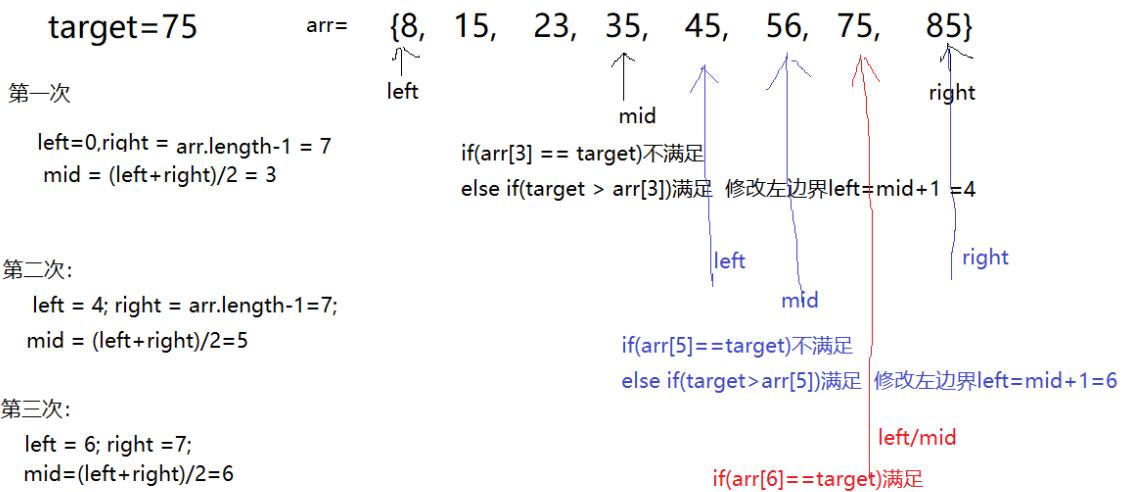
        Scanner input = new Scanner(System.in);
        System.out.print("请输入你要查找的值: ");
        int target = input.nextInt();

        int index = -1;
        for(int left = 0,right = arr.length-1; left<=right; ){
            //int mid = (left+right)/2;
            int mid = left + (right-left)/2;

            if(arr[mid] == target){
                index = mid;
                break;
            }else if(target > arr[mid]){
                //说明target在[mid]右边
                left = mid+1;
            }else{
                //说明target<arr[mid], target在[mid]左边
                right= mid-1;
            }
        }
        if(index!=-1){
            System.out.println("找到了, 下标是"+index);
        }else{
            System.out.println("不存在");
        }

    }
}

```

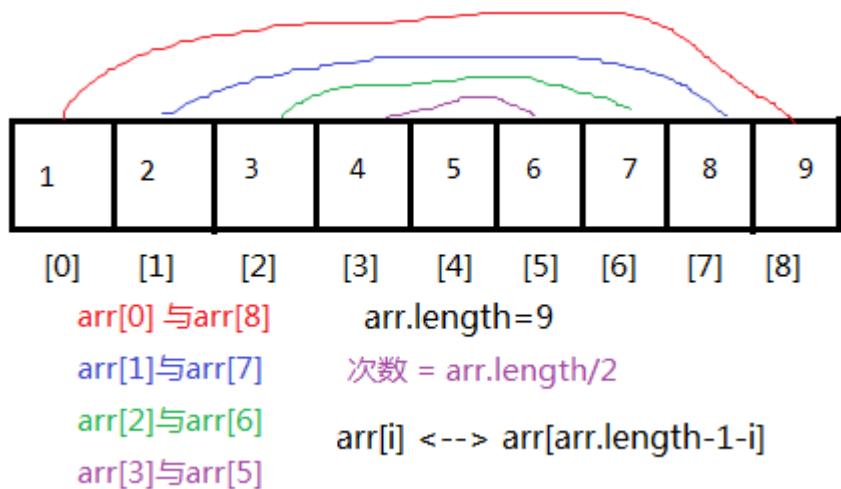


第五次: $\text{left}=7, \text{right}=6$

发现 $\text{left}>\text{right}$ 说明所有该比较的数字都比较过了，结束查找

4.4.5 数组元素的反转

实现思想：数组对称位置的元素互换。



```

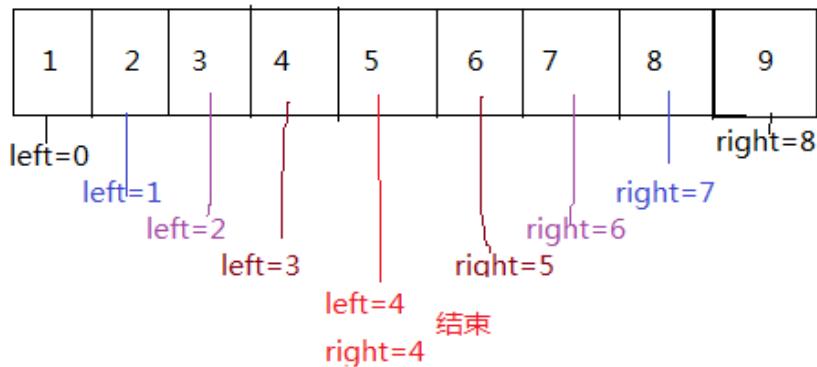
public class Test17ArrayReverse {
    public static void main(String[] args) {
        int[] arr = {1,2,3,4,5};
        System.out.println("反转之前: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }

        //反转
        /*
        思路: 首尾对应位置的元素交换
        (1) 确定交换几次
        次数 = 数组.length / 2
        (2) 谁和谁交换
        for(int i=0; i<次数; i++){
            int temp = arr[i];
            arr[i] = arr[arr.length-1-i];
            arr[arr.length-1-i] = temp;
        }
        */
        for(int i=0; i<arr.length/2; i++){
            int temp = arr[i];
            arr[i] = arr[arr.length-1-i];
            arr[arr.length-1-i] = temp;
        }

        System.out.println("反转之后: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}

```

或



arr[0] 与 arr[8]交换

arr[1] 与 arr[7] 交换

arr[2] 与 arr[6]交换

arr[3] 与 arr[5]交换

即arr[left]与 arr[right]交换

一开始 left =0 , right=arr.length-1

然后 left++ , right-- 只要left<right就交换

```
public class Test17ArrayReverse2 {
    public static void main(String[] args) {
        int[] arr = {1,2,3,4,5};
        System.out.println("反转之前: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }

        //反转
        //左右对称位置交换
        for(int left=0,right=arr.length-1; left<right; left++,right--){
            //首 与 尾交换
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
        }

        System.out.println("反转之后: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}
```

4.4.6 数组元素排序

1、排序算法概述

数组的排序算法很多，实现方式各不相同，时间复杂度、空间复杂度、稳定性也各不相同：

| 排序方法 | 时间复杂度（平均） | 时间复杂度（最坏） | 时间复杂度（最好） | 空间复杂度 | 稳定性 |
|------|-----------------|-----------------|-----------------|-----------------|-----|
| 插入排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 希尔排序 | $O(n^{1.3})$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 不稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 堆排序 | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(1)$ | 不稳定 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 快速排序 | $O(n \log_2 n)$ | $O(n^2)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ | 不稳定 |
| 归并排序 | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n)$ | 稳定 |
| 计数排序 | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | 稳定 |
| 桶排序 | $O(n+k)$ | $O(n^2)$ | $O(n)$ | $O(n+k)$ | 稳定 |
| 基数排序 | $O(n*k)$ | $O(n*k)$ | $O(n*k)$ | $O(n+k)$ | 稳定 |

<https://help.osdn.net/qq/38098813>

- 时间复杂度：

常见的算法时间复杂度由小到大依次为： $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < \dots < O(2^n) < O(n!)$

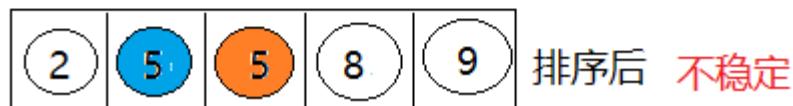
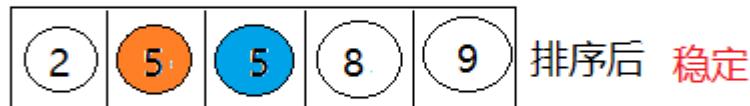
一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为 $T(n)$ 。 n 称为问题的规模，当 n 不断变化时，时间频度 $T(n)$ 也会不断变化。但有时我们知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 $T(n)$ 表示，若有某个辅助函数 $f(n)$ ，使得当 n 趋近于无穷大时， $T(n)/f(n)$ 的极限值为不等于零的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n)=O(f(n))$ ，称 $O(f(n))$ 为算法的渐进时间复杂度，简称时间复杂度。

- 空间复杂度：

类似于时间复杂度的讨论，一个算法的空间复杂度 (Space Complexity) $S(n)$ 定义为该算法所耗费的存储空间，它也是问题规模 n 的函数。

- 稳定性：

排序一定会设计到数组元素位置的交换。如果两个元素相等，无论它们原来是否相邻，在排序过程中，最后它们变的相邻，但是它们前后顺序并没有改变，就称为稳定的，否则就是不稳定的。



2、直接选择排序

```
/*
1、直接选择排序
```

思想：每一轮找出本轮的最大值/最小值，然后看它是否在它应该在的位置。
如果不在正确的位置，就与这个位置的元素交换。

过程：arr{6,9,2,9,1} 目标：从小到大

第1轮：最大值是9，它现在在arr[1]，它应该在arr[4]，不对，交换arr[1]和arr[4]，{6,1,2,9,9}

第2轮：最大值是9，它现在在arr[3]，它应该在arr[3]，对，不动

第3轮：最大值是6，它现在在arr[0]，它应该在arr[2]，不对，交换arr[0]和arr[2]，{2,1,6,9,9}

第4轮：最大值是2，它现在在arr[0]，它应该在arr[1]，不对，交换arr[0]和arr[1]，{1,2,6,9,9}

过程：arr{6,9,2,9,1} 目标：从小到大

第1轮：最小值是1，它现在在arr[4]，它应该在arr[0]，不对，交换arr[4]和arr[0]，{1,9,2,9,6}

第2轮：最小值是2，它现在在arr[2]，它应该在arr[1]，不对，交换arr[2]和arr[1]，{1,2,9,9,6}

第3轮：最小值是6，它现在在arr[4]，它应该在arr[2]，不对，交换arr[4]和arr[2]，{1,2,6,9,9}

第4轮：最小值是7，它现在在arr[3]，它应该在arr[3]，对，不动

```
*/
public class Test18SelectSort{
    public static void main(String[] args){
        int[] arr = {6,9,2,9,1};

        //直接选择排序, 轮数 = 数组的元素总个数-1
        /*
        arr.length=5
        i=0
        i=1
        i=2
        i=3
        */
        for(int i=0; i<arr.length-1; i++){
            //找出本轮的最小值, 及其下标
            /*
            i=0, 第1轮, 查找的范围是[0,4], 一开始假设arr[0]最小
            i=1, 第2轮, 查找的范围是[1,4], 一开始假设arr[1]最小
            */
        }
    }
}
```

```

        i=2, 第3轮, 查找的范围是[2,4], 一开始假设arr[2]最小
        i=3, 第4轮, 查找的范围是[3,4], 一开始假设arr[3]最小
        int min = arr[i];
        */
    int min = arr[i];
    int index = i;
    //用[i+1, arr.length-1]范围的元素与min比较
    for(int j=i+1; j<arr.length; j++){
        if(arr[j] < min){
            min = arr[j];
            index = j;
        }
    }

    //判断min是否在它应该在的位置
    /*
        i=0, 第1轮, 最小值应该在arr[0]位置, 它现在在arr[index]位置
        i=1, 第2轮, 最小值应该在arr[1]位置, 它现在在arr[index]位置
        i=2, 第3轮, 最小值应该在arr[2]位置, 它现在在arr[index]位置
        i=3, 第4轮, 最小值应该在arr[3]位置, 它现在在arr[index]位置

        最小值应该在arr[i]位置, 如果index!=i, 说明它不在应该在的位置,
        就交换arr[i]和arr[index]位置
    */
    if(index!=i){
        int temp = arr[i];
        arr[i] = arr[index];
        arr[index] = temp;
    }

}

//完成排序, 遍历结果
for(int i=0; i<arr.length; i++){
    System.out.print(arr[i]+" ");
}

}
}

```

3、冒泡排序

Java中的经典算法之冒泡排序（Bubble Sort）

原理：比较两个相邻的元素，将值大的元素交换至右端。

思路：依次比较相邻的两个数，将小数放到前面，大数放到后面。

即第一趟，首先比较第1个和第2个元素，将小数放到前面，大数放到后面。

然后比较第2个和第3个元素，将小数放到前面，大数放到后面。

如此继续，直到比较最后两个数，将小数放到前面，大数放到后面。

重复第一趟步骤，直至全部排序完成。

```
/*
```

1、冒泡排序（最经典）

思想：每一次比较“相邻（位置相邻）”元素，如果它们不符合目标顺序（例如：从小到大），就交换它们，经过多轮比较，最终实现排序。

（例如：从小到大） 每一轮可以把最大的沉底，或最小的冒顶。

过程：arr{6,9,2,9,1} 目标：从小到大

第一轮：

第1次，arr[0]与arr[1]，6>9不成立，满足目标要求，不交换

第2次，arr[1]与arr[2]，9>2成立，不满足目标要求，交换arr[1]与arr[2] {6,2,9,9,1}

第3次，arr[2]与arr[3]，9>9不成立，满足目标要求，不交换

第4次，arr[3]与arr[4]，9>1成立，不满足目标要求，交换arr[3]与arr[4] {6,2,9,1,9}

第一轮所有元素{6,9,2,9,1}已经都参与了比较，结束。

第一轮的结果：第“一”最大值9沉底（本次是后面的9沉底），即到{6,2,9,1,9}元素的最右边

第二轮：

第1次，arr[0]与arr[1]，6>2成立，不满足目标要求，交换arr[0]与arr[1] {2,6,9,1,9}

第2次，arr[1]与arr[2]，6>9不成立，满足目标要求，不交换

第3次：arr[2]与arr[3]，9>1成立，不满足目标要求，交换arr[2]与arr[3] {2,6,1,9,9}

第二轮未排序的所有元素 {6,2,9,1}已经都参与了比较，结束。

第二轮的结果：第“二”最大值9沉底（本次是前面的9沉底），即到{2,6,1,9}元素的最右边

第三轮：

第1次，arr[0]与arr[1]，2>6不成立，满足目标要求，不交换

第2次，arr[1]与arr[2]，6>1成立，不满足目标要求，交换arr[1]与arr[2] {2,1,6,9,9}

第三轮未排序的所有元素{2,6,1}已经都参与了比较，结束。

第三轮的结果：第三最大值6沉底，即到 {2,1,6}元素的最右边

第四轮：

第1次，arr[0]与arr[1]，2>1成立，不满足目标要求，交换arr[0]与arr[1] {1,2,6,9,9}

第四轮未排序的所有元素{2,1}已经都参与了比较，结束。

第四轮的结果：第四最大值2沉底，即到{1,2}元素的最右边

```
*/
```

```
public class Test19BubbleSort{
    public static void main(String[] args){
        int[] arr = {6,9,2,9,1};

        //目标：从小到大
        //冒泡排序的轮数 = 元素的总个数 - 1
        //轮数是多轮，每一轮比较的次数是多次，需要用到双重循环，即循环嵌套
        //外循环控制 轮数，内循环控制每一轮的比较次数和过程
        for(int i=1; i<arr.length; i++){ //循环次数是arr.length-1次/轮
            /*
            假设arr.length=5
            i=1, 第1轮，比较4次
                arr[0]与arr[1]
                arr[1]与arr[2]
                arr[2]与arr[3]
                arr[3]与arr[4]

            arr[j]与arr[j+1], int j=0;j<4; j++

            i=2, 第2轮，比较3次
        }
    }
}
```

```

        arr[0]与arr[1]
        arr[1]与arr[2]
        arr[2]与arr[3]

        arr[j]与arr[j+1], int j=0;j<3; j++

        i=3,第3轮, 比较2次
            arr[0]与arr[1]
            arr[1]与arr[2]

            arr[j]与arr[j+1], int j=0;j<2; j++
        i=4,第4轮, 比较1次
            arr[0]与arr[1]

            arr[j]与arr[j+1], int j=0;j<1; j++

            int j=0; j<arr.length-i; j++
        */

    for(int j=0; j<arr.length-i; j++){
        //希望的是arr[j] < arr[j+1]
        if(arr[j] > arr[j+1]){
            //交换arr[j]与arr[j+1]
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
}

//完成排序, 遍历结果
for(int i=0; i<arr.length; i++){
    System.out.print(arr[i]+" ");
}
}
}

```

4、冒泡排序优化（选讲）

```

/*
思考：冒泡排序是否可以优化
*/
class Test19BubbleSort2{
    public static void main(String[] args){
        int[] arr = {1,3,5,7,9};

        //从小到大排序
        //int lun = 0;//声明lun变量, 统计比较几轮
        //int count = 0;//声明count变量, 统计比较的次数
        for(int i=1; i<arr.length; i++){
            //lun++;
            boolean flag = true;//假设数组已经是有序的
            for(int j=0; j<arr.length-i; j++){
                //count++;

```

```
//希望的是arr[j] < arr[j+1]
if(arr[j] > arr[j+1]){
    //交换arr[j]与arr[j+1]
    int temp = arr[j];
    arr[j] = arr[j+1];
    arr[j+1] = temp;

    flag = false;//如果元素发生了交换，那么说明数组还没有排好序
}
if(flag){
    break;
}
}

//System.out.println("一共比较了" + lun +"轮");
//System.out.println("一共比较了" + count +"次");

//完成排序，遍历结果
for(int i=0; i<arr.length; i++){
    System.out.print(arr[i]+ " ");
}
}
```

第5章 面向对象（上）

5.1 面向对象编程

5.1.2 类和对象

1、什么是类

类是一类具有相同特性的事物的抽象描述，是一组相关**属性**和**行为**的集合。

- **属性**：就是该事物的状态信息。
- **行为**：就是在你这个程序中，该状态信息要做什么操作，或者基于事物的状态能做什么。

2、什么是对象

对象是一类事物的一个具体个体（对象并不是找个女朋友）。即对象是类的一个**实例**，必然具备该类事物的属性和行为。

例如：做一个养宠物的小游戏

类：人、猫、狗等

```
public class Dog{
    String type; //种类
    String nickname; //昵称
    int energy; //能量
    final int MAX_ENERGY = 10000;

    //买东西
    void eat(){
        if(energy < MAX_ENERGY){
            energy += 10;
        }
    }
}
```

```
public class Person{
    String name;
    char gender;
    Dog dog;

    //喂宠物
    void feed(){
        dog.eat();
    }
}
```

```
public class Game{
    public static void main(String[] args){
        Person p = new Person();
        p.name = "张三";
        p.gender = '男';
        p.dog = new Dog();

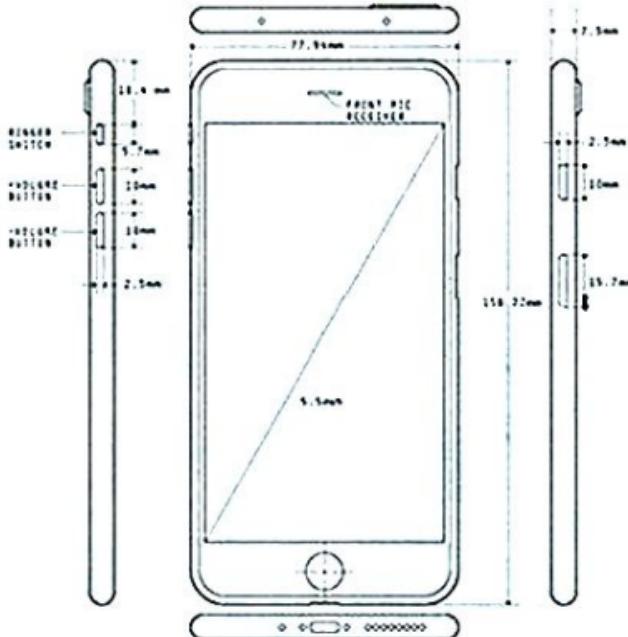
        p.dog.type = "哈巴狗";
        p.dog.nickname = "小白";

        for(int i=1; i<=5; i++){
            p.feed();
        }

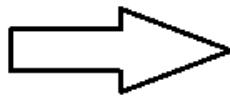
        System.out.println(p.dog.energy);
    }
}
```

3、类与对象的关系

- 类是对一类事物的描述，是**抽象的**。
- 对象是一类事物的实例，是**具体的**。
- **类是对象的模板，对象是类的实体。**



手机的设计图（抽象的）



真正的手机（具体的）

5.1.3 如何定义类

1、类的定义格式

关键字：class（小写）

```
【修饰符】 class 类名{  
}
```

类的定义格式举例：

```
public class Student{  
}
```

2、对象的创建

关键字：new

```
new 类名() //也称为匿名对象  
  
//给创建的对象命名  
//或者说，把创建的对象用一个引用数据类型的变量保存起来，这样就可以反复使用这个对象了  
类名 对象名 = new 类名();
```

那么，对象名中存储的是什么呢？答：对象地址

```

public class TestStudent{
    public static void main(String[] args){
        System.out.println(new Student());//student@7852e922

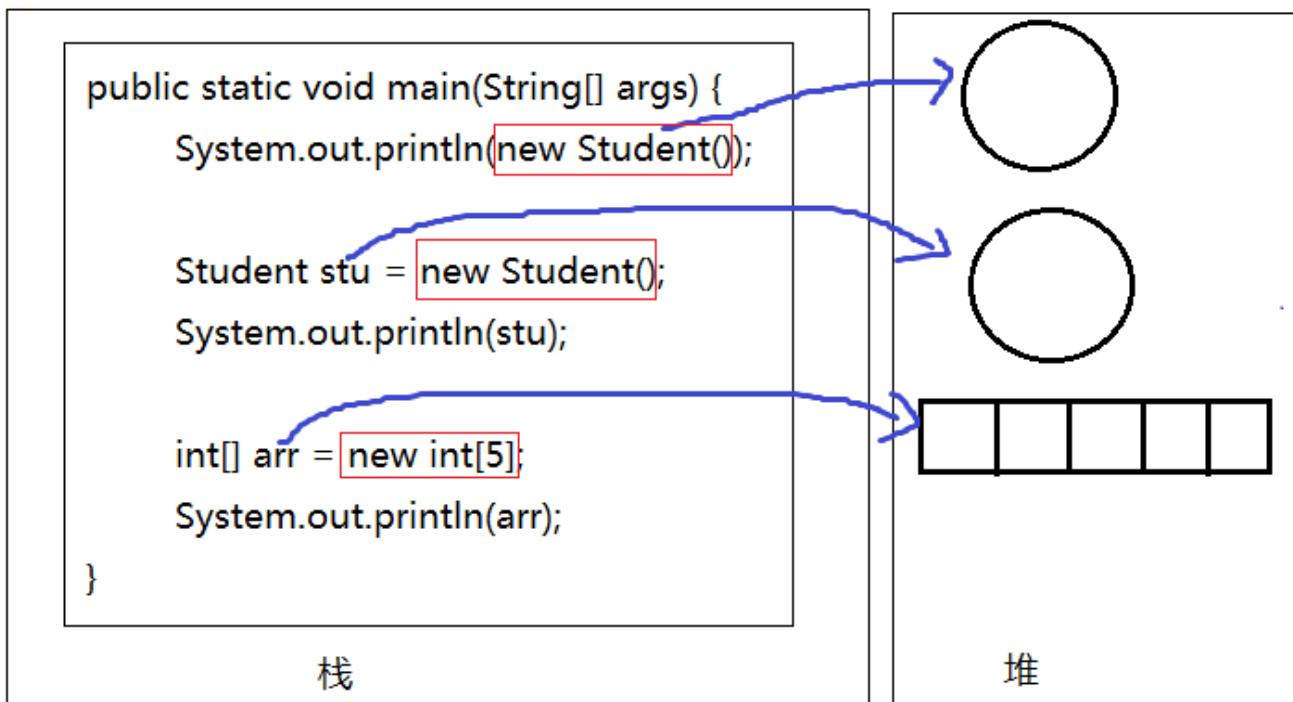
        Student stu = new Student();
        System.out.println(stu);//Student@4e25154f

        int[] arr = new int[5];
        System.out.println(arr);//[I@70dea4e
    }
}

```

发现学生对象和数组对象类似，直接打印对象名和数组名都是显示“类型@对象的hashCode值”，所以说类、数组都是引用数据类型，引用数据类型的变量中存储的是对象的地址，或者说指向堆中对象的首地址。

那么像“Student@4e25154f”是对象的地址吗？不是，因为Java是对程序员隐藏内存地址的，不暴露内存地址信息，所以打印对象时不直接显示内存地址，而是JVM帮你调用了对象的toString方法，将对象的基本信息转换为字符串并返回，默认toString方法返回的是“对象的运行时类型@对象的hashCode值的十六进制值”，程序员可以自己改写toString方法的代码（后面会讲如何改写）。



5.2 包 (Package)

5.2.1 包的作用

- (1) 可以避免类重名：有了包之后，类的全名称就变为：包.类名
- (2) 可以控制某些类型或成员的可见范围

如果某个类型或者成员的权限修饰缺省的话，那么就仅限于本包使用。

- (3) 分类组织管理众多的类

例如：

- java.lang----包含一些Java语言的核心类，如String、Math、Integer、System和Thread等，提供常用功能
- java.net----包含执行与网络相关的操作的类和接口。
- java.io ----包含能提供多种输入/输出功能的类。
- java.util----包含一些实用工具类，如集合框架类、日期时间、数组工具类Arrays，文本扫描仪Scanner，随机值产生工具Random。
- java.text----包含了一些java格式化相关的类
- java.sql和javax.sql---包含了java进行JDBC数据库编程的相关类/接口
- java.awt和java.swing----包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。

5.2.2 如何声明包

关键字： package

```
package 包名;
```

注意：

- (1) 必须在源文件的代码首行
- (2) 一个源文件只能有一个声明包的package语句

包的命名规范和习惯： (1) 所有单词都小写，每一个单词之间使用.分割 (2) 习惯用公司的域名倒置开头

例如： com.atguigu.xxx;

建议大家取包名时不要使用“java.xx”包

5.2.3 如何跨包使用类

==注意： ==只有public的类才能被跨包使用

- (1) 使用类型的全名称

例如： java.util.Scanner input = new java.util.Scanner(System.in);

- (2) 使用import语句之后，代码中使用简名称

import语句告诉编译器到哪里去寻找类。

import语句的语法格式：

```
import 包.类名;  
import 包.*;
```

注意：

使用java.lang包下的类，不需要import语句，就直接可以使用简名称

import语句必须在package下面，class的上面

当使用两个不同包的同名类时，例如：java.util.Date和java.sql.Date。一个使用全名称，一个使用简名称

示例代码：

```
package com.atguigu.test02.pkg;

import com.atguigu.test01.oop.Student;

import java.util.Date;
import java.util.Scanner;

public class TestPackage {
    public static void main(String[] args) {
/*        java.util.Scanner input = new java.util.Scanner(System.in);
        com.atguigu.test01.oop.Student stu = new com.atguigu.test01.oop.Student();*/
        Scanner input = new Scanner(System.in);
        Student student = new Student();

        Date d1 = new Date();
        java.sql.Date d2 = new java.sql.Date(0);
    }
}
```

5.3 成员变量

5.3.1 如何声明成员变量

```
【修饰符】 class 类名{
    【修饰符】 数据类型 成员变量名;
}
```

示例：

```
public class Person{
    String name;
    char gender;
    int age;
}
```

位置要求：必须在类中，方法外

类型要求：可以是Java的任意类型，包括基本数据类型、引用数据类型（类、接口、数组等）

修饰符：成员变量的修饰符有很多，例如：public、protected、private、static、volatile、transient、final等，后面会一一学习。

其中static可以将成员变量分为两大类，静态变量和非静态变量。其中静态变量又称为类变量，非静态变量又称为实例变量或者属性。==接下来先学习实例变量。==

5.3.2 对象的实例变量

1、实例变量的特点

(1) 实例变量的值是属于某个对象的

- 必须通过对象才能访问实例变量
- 每个对象的实例变量的值是独立的

(2) 实例变量有默认值

| 分类 | 数据类型 | 默认值 |
|------|-----------------------------|----------|
| 基本类型 | 整数 (byte, short, int, long) | 0 |
| | 浮点数 (float, double) | 0.0 |
| | 字符 (char) | '\u0000' |
| | 布尔 (boolean) | false |
| | 数据类型 | 默认值 |
| 引用类型 | 数组, 类, 接口 | null |

2、实例变量的访问

对象.实例变量

例如：

```
package com.atguigu.test03.field;

public class TestPerson {
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.name = "张三";
        p1.age = 23;
        p1.gender = '男';

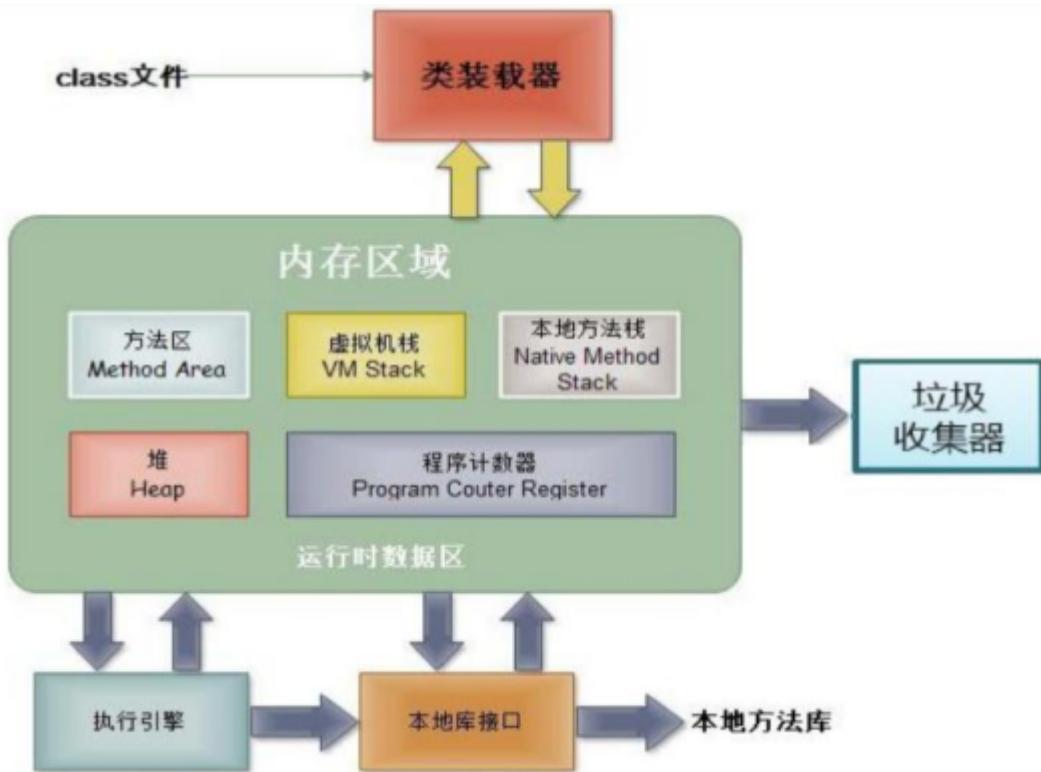
        Person p2 = new Person();
        /*
         * (1) 实例变量的值是属于某个对象的
         * - 必须通过对象才能访问实例变量
         * - 每个对象的实例变量的值是独立的
         * (2) 实例变量有默认值
         */
        System.out.println("p1对象的实例变量：");
        System.out.println("p1.name = " + p1.name);
        System.out.println("p1.age = " + p1.age);
        System.out.println("p1.gender = " + p1.gender);

        System.out.println("p2对象的实例变量：");
        System.out.println("p2.name = " + p2.name);
        System.out.println("p2.age = " + p2.age);
        System.out.println("p2.gender = " + p2.gender);
    }
}
```

3、实例变量的内存分析

内存是计算机中重要的部件之一，它是与CPU进行沟通的桥梁。其作用是用于暂时存放CPU中的运算数据，以及与硬盘等外部存储器交换的数据。只要计算机在运行中，CPU就会把需要运算的数据调到内存中进行运算，当运算完成后CPU再将结果传送出来。我们编写的程序是存放在硬盘中的，在硬盘中的程序是不会运行的，必须放进内存中才能运行，运行完毕后会清空内存。Java虚拟机要运行程序，必须要对内存进行空间的分配和管理，每一片区域都有特定的处理数据方式和内存管理方式。

JVM的运行时内存区域分为：方法区、堆、虚拟机栈、本地方法栈、程序计数器几大块。



| 区域名称 | 作用 |
|-------|--|
| 程序计数器 | 程序计数器是CPU中的寄存器，它包含每一个线程下一条要执行的指令的地址 |
| 本地方法栈 | 当程序中调用了native的本地方法时，本地方法执行期间的内存区域 |
| 方法区 | 存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。 |
| 堆内存 | 存储对象（包括数组对象），new来创建的，都存储在堆内存。 |
| 虚拟机栈 | 用于存储正在执行的每个Java方法的局部变量表等。局部变量表存放了编译期可知长度的各种基本数据类型、对象引用，方法执行完，自动释放。 |

Java对象保存在内存中时，由以下三部分组成：

- 对象头

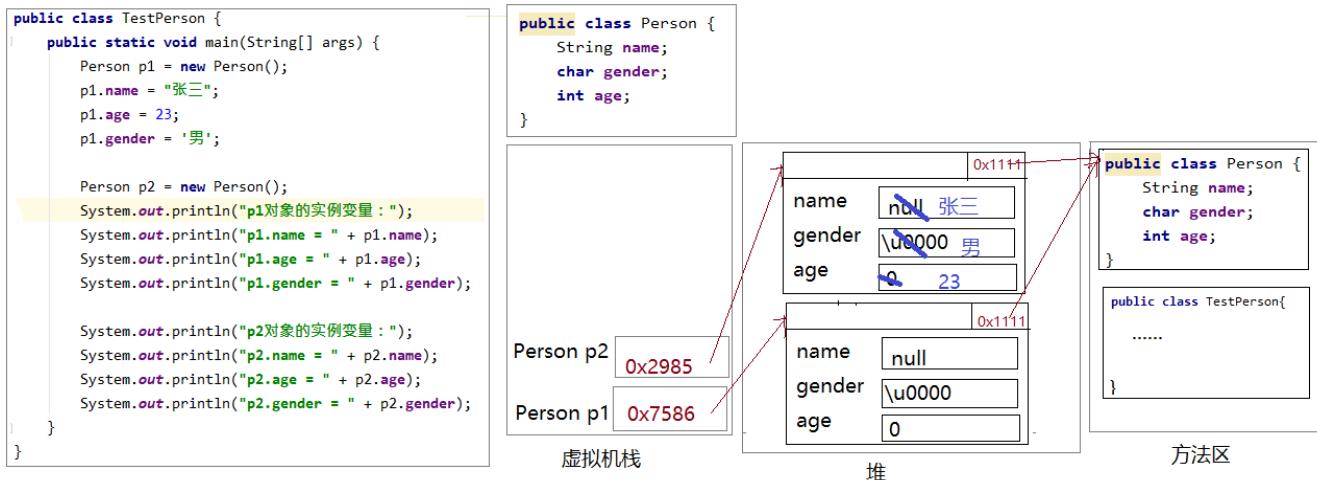
- Mark Word：记录了和当前对象有关的GC、锁等信息。（后面再讲）
- 指向类的指针：每一个对象需要记录它是由哪个类创建出来的，而Java对象的类数据保存在方法区，指向类的指针就是记录创建该对象的类数据在方法区的首地址。该指针在32位JVM中的长度是32bit，在64位JVM中长度是64bit。
- 数组长度（只有数组对象才有）

- 实例数据

- 即实例变量的值

- 对齐填充

- 因为JVM要求Java对象占的内存大小应该是8bit的倍数，如果不满足该大小，则需要补齐至8bit的倍数，没有特别的功能。



5.4 方法 (Method)

5.4.1 方法的概念

方法也叫函数，是一组代码语句的封装，从而实现代码重用，从而减少冗余代码，通常它是一个独立功能的定义，方法是一个类中最基本的功能单元。

Math.random()的random()方法
Math.sqrt(x)的sqrt(x)方法
System.out.println(x)的println(x)方法

Scanner input = new Scanner(System.in);
input.nextInt()的nextInt()方法

5.4.2 方法的特点

(1) 必须先声明后使用

类，变量，方法等都要先声明后使用

(2) 不调用不执行，调用一次执行一次。

5.4.3 如何声明方法

1、声明方法的位置

声明方法的位置==必须在类中方法外==，即不能在一个方法中直接定义另一个方法。

声明位置示例：

```
类{
    方法1() {
        }
    方法2() {
        }
}
```

错误示例：

```
类{
    方法1() {
        方法2() { //位置错误
            }
    }
}
```

2、声明方法的语法格式

```
【修饰符】 返回值类型 方法名(【形参列表】)【throws 异常列表】{
    方法体的功能代码
}
```

(1) 一个完整的方法 = 方法头 + 方法体。

- 方法头就是 【修饰符】 返回值类型 方法名(【形参列表】)【throws 异常列表】，也称为方法签名，通常调用方法时只需要关注方法头就可以，从方法头可以看出这个方法的功能和调用格式。
- 方法体就是方法被调用后要指定的代码，也是完成方法功能的具体实现代码，对于调用者来说，不了解方法体如何实现的，并影响方法的使用。

(2) 方法头可能包含5个部分，但是有些部分是可能缺省的

- 修饰符：可选的。方法的修饰符也有很多，例如：public、protected、private、static、abstract、native、final、synchronized等，后面会一一学习。其中根据是否有static，可以将方法分为静态方法和非静态方法。其中静态方法又称为类方法，非静态方法又称为实例方法。==接下来咱们先学习实例方法==。

返回值类型：表示方法运行的结果的数据类型，方法执行后将结果返回到调用者

- 基本数据类型
- 引用数据类型
- 无返回值类型：void

方法名：给方法起一个名字，见名知意，能准确代表该方法功能的名字

参数列表：表示完成方法体功能时需要外部提供的数据列表

- 无论是否有参数，()不能省略
- 如果有参数，每一个参数都要指定数据类型和参数名，多个参数之间使用逗号分隔，例如：
 - 一个参数：(数据类型 参数名)
 - 二个参数：(数据类型1 参数1, 数据类型2 参数2)
- 参数的类型可以是基本数据类型、引用数据类型
- throws 异常列表：可选，在异常章节再讲

(3) 方法体：方法体必须有{}括起来，在{}中编写完成方法功能的代码

关于方法体中return语句的说明：

- return语句的作用是结束方法的执行，并将方法的结果返回去
- 如果返回值类型不是void，方法体中必须保证一定有 return 返回值; 语句，并且要求该返回值结果的类型与声明的返回值类型一致或兼容。
- 如果返回值类型为void时，方法体中可以没有return语句，如果要用return语句提前结束方法的执行，那么return后面不能跟返回值，直接写return ; 就可以。
- return语句后面就不能再写其他代码了，否则会报错：Unreachable code

示例：

```
package com.atguigu.test04.method;

/**
 * 方法定义案例演示
 */
public class MethodDefineDemo {
    /**
     * 无参无返回值方法的演示
     */
    void sayHello(){
        System.out.println("hello");
    }

    /**
     * 有参无返回值方法的演示
     * @param length int 第一个参数，表示矩形的长
     * @param width int 第二个参数，表示矩形的宽
     * @param sign char 第三个参数，表示填充矩形图形的符号
     */
    void printRectangle(int length, int width, char sign){
        for (int i = 1; i <= length ; i++) {
            for(int j=1; j <= width; j++){
                System.out.print(sign);
            }
            System.out.println();
        }
    }
}
```

```
/*
 * 无参有返回值方法的演示
 * @return
 */
int getIntBetweenOneToHundred(){
    return (int)(Math.random()*100+1);
}

/*
 * 有参有返回值方法的演示
 * @param a int 第一个参数，要比较大小的整数之一
 * @param b int 第二个参数，要比较大小的整数之二
 * @return int 比较大小的两个整数中较大者的值
 */
int max(int a, int b){
    return a > b ? a : b;
}
}
```

5.4.4 如何调用实例方法

1、方法调用语法格式

对象.非静态方法(【实参列表】)

例如：

```
package com.atguigu.test04.method;

/**
 * 方法调用案例演示
 */
public class MethodInvokeDemo {
    public static void main(String[] args) {
        //创建对象
        MethodDefineDemo md = new MethodDefineDemo();

        System.out.println("-----方法调用演示-----");

        //调用MethodDefineDemo类中无参无返回值的方法sayHello
        md.sayHello();
        md.sayHello();
        md.sayHello();
        //调用一次，执行一次，不调用不执行

        System.out.println("-----");
        //调用MethodDefineDemo类中有参无返回值的方法printRectangle
        md.printRectangle(5,10, '@');

        System.out.println("-----");
```

```

//调用MethodDefineDemo类中无参有返回值的方法getIntBetweenOneToHundred
md.getIntBetweenOneToHundred(); //语法没问题，就是结果丢失

int num = md.getIntBetweenOneToHundred();
System.out.println("num = " + num);

System.out.println(md.getIntBetweenOneToHundred());
//上面的代码调用了getIntBetweenOneToHundred三次，这个方法执行了三次

System.out.println("-----");
//调用MethodDefineDemo类中有参有返回值的方法max
md.max(3, 6); //语法没问题，就是结果丢失

int bigger = md.max(5, 6);
System.out.println("bigger = " + bigger);

System.out.println("8,3中较大者是: " + md.max(8, 9));
}

}

```

回忆之前的代码：

```

//1、创建Scanner的对象
Scanner input = new Scanner(System.in); //System.in默认代表键盘输入

//2、提示输入xx
System.out.print("请输入一个整数: "); //对象.非静态方法(实参列表)

//3、接收输入内容
int num = input.nextInt(); //对象.非静态方法()

```

2、形参和实参

- 形参 (formal parameter)：在定义方法时方法名后面括号中声明的变量称为形式参数（简称形参）即形参出现在方法定义时。
- 实参 (actual parameter)：调用方法时方法名后面括号中的使用的值/变量/表达式称为实际参数（简称实参）即实参出现在方法调用时。
- 调用时，实参的个数、类型、顺序顺序要与形参列表一一对应。如果方法没有形参，就不需要也不能传实参。
- 无论是否有参数，声明方法和调用方法是==()都不能丢失==

3、返回值问题

方法调用表达式是一个特殊的表达式：

- 如果被调用方法的返回值类型是void，调用时不需要也不能接收和处理（打印或参与计算）返回值结果，即方法调用表达式==只能==直接加;成为一个独立语句。
- 如果被调用方法有返回值，即返回值类型不是void，
 - 方法调用表达式的结果可以作为赋值表达式的值，
 - 方法调用表达式的结果可以作为计算表达式的一个操作数，
 - 方法调用表达式的结果可以作为另一次方法调用的实参，

- 方法调用表达式的结果可以不接收和处理，方法调用表达式直接加;成为一个独立的语句，这种情况，返回值丢失。

```

package com.atguigu.test04.method;

public class MethodReturnValue {
    public static void main(String[] args) {
        //创建对象
        MethodDefineDemo md = new MethodDefineDemo();

        //无返回值的都只能单独加;成一个独立语句
        //调用MethodDefineDemo类中无参无返回值的方法sayHello
        md.sayHello();
        //调用MethodDefineDemo类中有参无返回值的方法printRectangle
        md.printRectangle(5,10,'@');

        //有返回值的
        //(1)方法调用表达式可以作为赋值表达式的值
        int bigger = md.max(7,3);
        System.out.println("bigger = " + bigger);

        //(2)方法调用表达式可以作为计算表达式的一个操作数
        //随机产生两个[1,100]之间的整数，并求和
        int sum = md.getIntBetweenOneToHundred() + md.getIntBetweenOneToHundred();
        System.out.println("sum = " + sum);

        //(3)方法调用表达式可以作为另一次方法调用的实参
        int x = 4;
        int y = 5;
        int z = 2;
        int biggest = md.max(md.max(x,y),z);
        System.out.println("biggest = " + biggest);

        //(4)方法调用表达式直接加;成为一个独立的语句，这种情况，返回值丢失
        md.getIntBetweenOneToHundred();
    }
}

```

5.4.5 实例方法使用当前对象的成员

在实例方法中还可以使用当前对象的其他成员。在Java中当前对象用this表示。

- this: 在实例方法中，表示调用该方法的对象
- 如果没有歧义，完全可以省略this。

1、使用this.

案例：矩形类

```

package com.atguigu.test04.method;

```

```
public class Rectangle {  
    int length;  
    int width;  
  
    int area() {  
        return this.length * this.width;  
    }  
  
    int perimeter(){  
        return 2 * (this.length + this.width);  
    }  
  
    void print(char sign) {  
        for (int i = 1; i <= this.width; i++) {  
            for (int j = 1; j <= this.length; j++) {  
                System.out.print(sign);  
            }  
            System.out.println();  
        }  
    }  
  
    String getInfo(){  
        return "长: " + this.length + ", 宽: " + this.width + ", 面积: " + this.area() + ", 周长: " +  
+ this.perimeter();  
    }  
}
```

测试类

```
package com.atguigu.test04.method;  
  
public class TestRectangle {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        Rectangle r2 = new Rectangle();  
  
        System.out.println("r1对象: " + r1.getInfo());  
        System.out.println("r2对象: " + r2.getInfo());  
  
        r1.length = 10;  
        r1.width = 2;  
        System.out.println("r1对象: " + r1.getInfo());  
        System.out.println("r2对象: " + r2.getInfo());  
  
        r1.print('#');  
        System.out.println("-----");  
        r1.print('&');  
  
        System.out.println("-----");  
        r2.print('#');  
        System.out.println("-----");  
        r2.print('%');
```

```
    }
}
```

2、省略this.

```
package com.atguigu.test04.method;

public class Rectangle {
    int length;
    int width;

    int area() {
        return length * width;
    }

    int perimeter(){
        return 2 * (length + width);
    }

    void print(char sign) {
        for (int i = 1; i <= width; i++) {
            for (int j = 1; j <= length; j++) {
                System.out.print(sign);
            }
            System.out.println();
        }
    }

    String getInfo(){
        return "长: " + length + ", 宽: " + width + ", 面积: " + area() + ", 周长: " + perimeter();
    }
}
```

5.4.6 方法调用内存分析

方法不调用不执行，调用一次执行一次，每次调用会在栈中有一个入栈动作，即给当前方法开辟一块独立的内存区域，用于存储当前方法的局部变量的值，当方法执行结束后，会释放该内存，称为出栈，如果方法有返回值，就会把结果返回调用处，如果没有返回值，就直接结束，回到调用处继续执行下一条指令。

栈结构：先进后出，后进先出。

```

package com.atguigu.test04.method;

public class MethodMemory {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle();
        r1.length = 10;
        r1.width = 2;
        r1.print('#');
        System.out.println("r1对象: " + r1.getInfo());
        System.out.println("r2对象: " + r2.getInfo());
    }
}

```

```

public class MethodMemory {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle();
        r1.length = 10;
        r1.width = 2;
        r1.print('#');
        System.out.println("r1对象: " + r1.getInfo());
        System.out.println("r2对象: " + r2.getInfo());
    }
}

```

```

public class Rectangle {
    int length;
    int width;

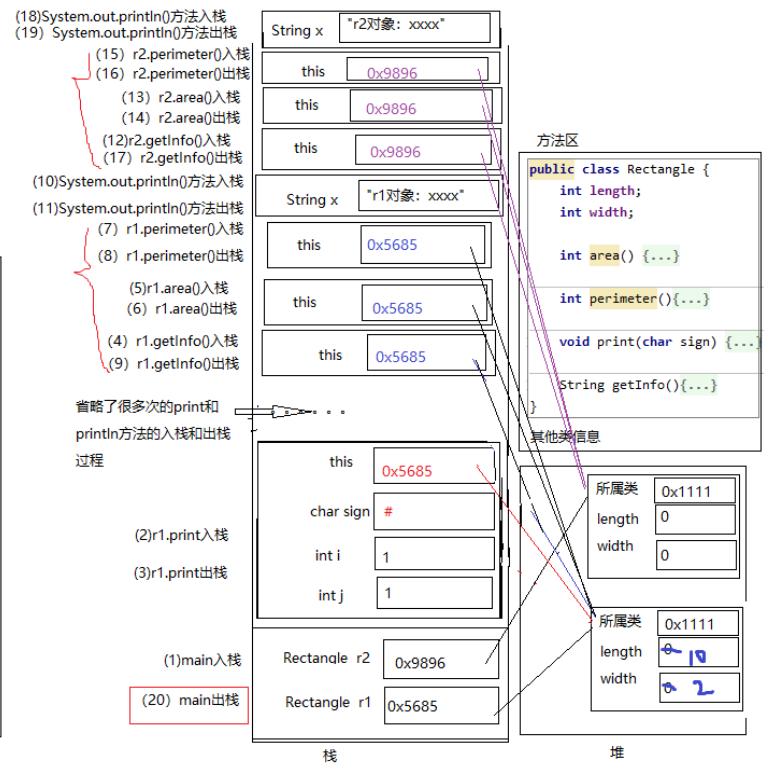
    int area() { return this.length * this.width; }

    int perimeter() { return 2 * (this.length + this.width); }

    void print(char sign) {
        for (int i = 1; i <= this.width; i++) {
            for (int j = 1; j <= this.length; j++) {
                System.out.print(sign);
            }
            System.out.println();
        }
    }

    String getInfo(){
        return "长: " + this.length + ", 宽: " + this.width
               +", 面积: " + this.area() +", 周长: " + this.perimeter();
    }
}

```



5.4.7 实例变量与局部变量的区别

- 1、声明位置和方式
 - (1) 实例变量：在类中方法外
 - (2) 局部变量：在方法体{}中或方法的形参列表、代码块中
- 2、在内存中存储的位置不同
 - (1) 实例变量：堆
 - (2) 局部变量：栈
- 3、生命周期
 - (1) 实例变量：和对象的生命周期一样，随着对象的创建而存在，随着对象被GC回收而消亡，而且每一个对象的实例变量是独立的。
 - (2) 局部变量：和方法调用的生命周期一样，每一次方法被调用而在存在，随着方法执行的结束而消亡，而且每一次方法调用都是独立的。
- 4、作用域
 - (1) 实例变量：通过对象就可以使用，本类中“this.”，没有歧义还可以省略`this.`，其他类中“对象.”
 - (2) 局部变量：出了作用域就不能使用
- 5、修饰符（后面来讲）
 - (1) 实例变量：`public, protected, private, final, volatile, transient` 等
 - (2) 局部变量：`final`
- 6、默认值
 - (1) 实例变量：有默认值
 - (2) 局部变量：没有，必须手动初始化。其中的形参比较特殊，靠实参给它初始化。

5.5 参数问题

5.5.1 特殊参数之一：可变参数

在JDK1.5之后，当定义一个方法时，形参的类型可以确定，但是形参的个数不确定，那么可以考虑使用可变参数。
可变参数的格式：

```
【修饰符】 返回值类型 方法名(【非可变参数部分的形参列表,】参数类型... 形参名){ }
```

可变参数的特点和要求：

- (1) 一个方法最多只能有一个可变参数
- (2) 如果一个方法包含可变参数，那么可变参数必须是形参列表的最后一个
- (3) 在声明它的方法中，可变参数当成数组使用
- (4) 其实这个书写“~”

```
【修饰符】 返回值类型 方法名(【非可变参数部分的形参列表,】参数类型[] 形参名){ }
```

只是后面这种定义，在调用时必须传递数组，而前者更灵活，既可以传递数组，又可以直接传递数组的元素，这样更灵活了。

1、方法只有可变参数

案例：求n个整数的和

```
package com.atguigu.test05.param;

public class NumberTools {
    int total(int[] nums){
        int he = 0;
        for (int i = 0; i < nums.length; i++) {
            he += nums[i];
        }
        return he;
    }

    int sum(int... nums){
        int he = 0;
        for (int i = 0; i < nums.length; i++) {
            he += nums[i];
        }
        return he;
    }
}
```

```
package com.atguigu.test05.param;

public class TestVarParam {
```

```
public static void main(String[] args) {
    NumberTools tools = new NumberTools();

    System.out.println(tools.sum());//0个实参
    System.out.println(tools.sum(5));//1个实参
    System.out.println(tools.sum(5,6,2,4));//4个实参
    System.out.println(tools.sum(new int[]{5,6,2,4}));//传入数组实参

    System.out.println("-----");
    System.out.println(tools.total(new int[]{})�//0个元素的数组
    System.out.println(tools.total(new int[]{5}))//1个元素的数组
    System.out.println(tools.total(new int[]{5,6,2,4}));//传入数组实参
}
}
```

2、方法包含非可变参数和可变参数

- 非可变参数部分必须传入对应类型和个数的实参；
- 可变参数部分按照可变参数的规则传入0~n个对应类型的实参或传入1个对应类型的数组实参；

案例：

n个字符串进行拼接，每一个字符串之间使用某字符进行分割，如果没有传入字符串，那么返回空字符串””

```
package com.atguigu.test05.param;

public class StringTools {
    String concat(char seperator, String... args){
        String str = "";
        for (int i = 0; i < args.length; i++) {
            if(i==0){
                str += args[i];
            }else{
                str += seperator + args[i];
            }
        }
        return str;
    }
}
```

```
package com.atguigu.test05.param;

public class StringToolsTest {
    public static void main(String[] args) {
        StringTools tools = new StringTools();

        System.out.println(tools.concat('-'));
        System.out.println(tools.concat('-', "hello"));
        System.out.println(tools.concat('-', "hello", "world"));
        System.out.println(tools.concat('-', "hello", "world", "java"));
    }
}
```

5.5.2 特殊参数之二：命令行参数（了解）

通过命令行给main方法的形参传递的实参称为命令行参数

```
public class TestCommandParam{
    //形参: String[] args
    public static void main(String[] args){
        System.out.println(args);
        System.out.println(args.length);

        for(int i=0; i<args.length; i++){
            System.out.println("第" + (i+1) + "个参数的值是: " + args[i]);
        }
    }
}
```

命令行:

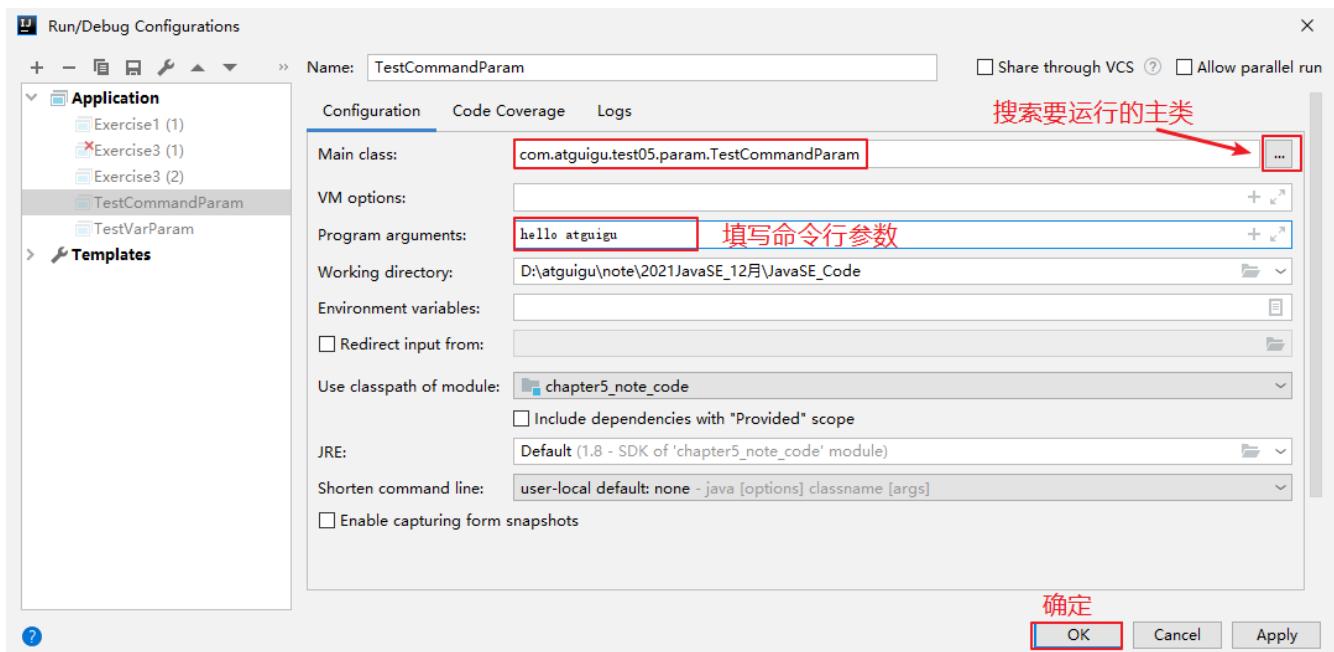
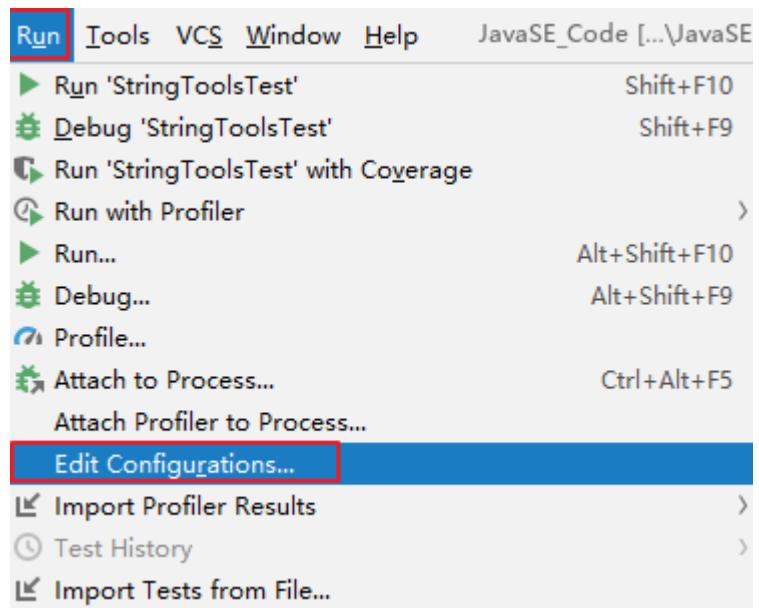
```
java TestCommandParam
```

```
java TestCommandParam 1 2 3
```

```
java TestCommandParam hello atguigu
```

IDEA工具:

(1) 配置运行参数



(2) 运行程序

```
1 package com.atguigu.test05.param;
2
3 public class TestCommandParam {
4     Run 'TestCommandParam.main()'
5     Debug TestCommandParam.main()
6     Run 'TestCommandParam.main()' with Coverage
7     Run with Profiler 'TestCommandParam.main()'
8
9     String[] args){
10    System.out.println(args.length);
11
12    for(int i=0; i<args.length; i++){
13        System.out.println("第" + (i+1) + "个参数的值是: " + args[i]);
14    }
15}
```

5.5.3 方法的参数传递机制

方法的参数传递机制：实参给形参赋值，那么反过来形参会影响实参吗？

- 方法的形参是基本数据类型时，形参值的改变不会影响实参；
- 方法的形参是引用数据类型时，形参地址值的改变不会影响实参，但是形参地址值里面的数据的改变会影响实参，例如，修改数组元素的值，或修改对象的属性值。
 - 注意：String、Integer等特殊类型容易错

1、形参是基本数据类型

案例：编写方法，交换两个整型变量的值

```
package com.atguigu.test05.param;

public class PrimitiveTypeParam {
    void swap(int a, int b){//交换两个形参的值
        int temp = a;
        a = b;
        b = temp;
    }

    public static void main(String[] args) {
        PrimitiveTypeParam tools = new PrimitiveTypeParam();
        int x = 1;
        int y = 2;
        System.out.println("交换之前: x = " + x + ", y = " + y);//1,2
        tools.swap(x,y); //实参x,y是基本数据类型，给形参的是数据的“副本”，调用完之后，x与y的值不变
        System.out.println("交换之后: x = " + x + ", y = " + y);//1,2
    }
}
```

2、形参是引用数据类型

```

package com.atguigu.test05.param;

public class ReferenceTypeParam {
    void swap(MyData my) { //形参my是引用数据类型，接收的是对象的地址值，形参my和实参data指向同一个对象
        //里面交换了对象的两个实例变量的值
        int temp = my.x;
        my.x = my.y;
        my.y = temp;
    }

    public static void main(String[] args) {
        ReferenceTypeParam tools = new ReferenceTypeParam();
        MyData data = new MyData();
        data.x = 1;
        data.y = 2;
        System.out.println("交换之前: x = " + data.x + ", y = " + data.y); //1,2
        tools.swap(data); //实参是data，给形参my的是对象的地址值，调用完之后，x与y的值交换
        System.out.println("交换之后: x = " + data.x + ", y = " + data.y); //2,1
    }
}

```

```

public class MyData{
    int x;
    int y;
}

```

3、形参是数组

```

package com.atguigu.test05.param;

public class ArrayTypeParam {
    void sort(int[] arr) { //给数组排序，修改了数组元素的顺序，这里对arr数组进行排序，就相当于对nums数组进行排序
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length - i; j++) {
                if(arr[j] > arr[j+1]){
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }

    void iterate(int[] arr) { //输出数组的元素，元素之间使用空格分隔，元素打印完之后换行
        //这个方法没有修改元素的值
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}

```

```

        System.out.println();
    }

public static void main(String[] args) {
    ArrayTypeParam tools = new ArrayTypeParam();

    int[] nums = {4,3,1,6,7};
    System.out.println("排序之前: ");
    tools.iterate(nums); //实参nums把数组的首地址给形参arr, 这个调用相当于输出nums数组的元素
                         //对数组的元素值没有影响

    tools.sort(nums); //对nums数组进行排序

    System.out.println("排序之后: ");
    tools.iterate(nums); //输出nums数组的元素
    //上面的代码, 从头到尾, 堆中只有一个数组, 没有产生新数组, 无论是排序还是遍历输出都是同一个数组
}
}

```

4、形参指向新对象

```

package com.atguigu.test05.param;

public class AssignNewObjectToFormalParam {
    void swap(MyData my){
        my = new MyData(); //这里让my形参指向了新对象, 此时堆中有两个MyData对象, 和main中的data对象无关
        int temp = my.x;
        my.x = my.y;
        my.y = temp;
    }

    public static void main(String[] args) {
        //创建这个对象的目的是为了调用swap方法
        AssignNewObjectToFormalParam tools = new AssignNewObjectToFormalParam();

        MyData data = new MyData();
        data.x = 1;
        data.y = 2;
        System.out.println("交换之前: x = " + data.x + ", y = " + data.y); //1,2
        tools.swap(data); //调用完之后, x与y的值交换?
        System.out.println("交换之后: x = " + data.x + ", y = " + data.y); //1,2
    }
}

```

5.6 方法的重载

- **方法重载**: 指在同一个类中, 允许存在一个以上的同名方法, 只要它们的参数列表不同即可, 与修饰符和返回值类型无关。

- 参数列表：数据类型个数不同，数据类型不同（按理来说数据类型顺序不同也可以，但是很少见，也不推荐，逻辑上容易有歧义）。
- 重载方法调用：JVM通过方法的参数列表，调用匹配的方法。
 - 先找个数、类型最匹配的
 - 再找个数和类型可以兼容的，如果同时多个方法可以兼容将会报错

案例，用重载实现：

- (1) 定义方法求两个整数的最大值
- (2) 定义方法求三个整数的最大值
- (3) 定义方法求两个小数的最大值
- (4) 定义方法求n个整数最大值

```
package com.atguigu.test06.overload;

public class MathTools {
    //求两个整数的最大值
    public int max(int a,int b){
        return a>b?a:b;
    }

    //求两个小数的最大值
    public double max(double a, double b){
        return a>b?a:b;
    }

    //求三个整数的最大值
    public int max(int a, int b, int c){
        return max(max(a,b),c);
    }

    //求n整数的最大值
    public int max(int... nums){
        int max = nums[0];//如果没有传入整数，或者传入null，这句代码会报异常
        for (int i = 1; i < nums.length; i++) {
            if(nums[i] > max){
                max = nums[i];
            }
        }
        return max;
    }
}
```

1、找最匹配的

```
package com.atguigu.test06.overload;

public class MethodOverloadMostMatch {
    public static void main(String[] args) {
        MathTools tools = new MathTools();

        System.out.println(tools.max(5,3));
        System.out.println(tools.max(5,3,8));
        System.out.println(tools.max(5.7,2.5));
    }
}
```

2、找唯一可以兼容的

```
package com.atguigu.test06.overload;

public class MethodOverloadMostCompatible {
    public static void main(String[] args) {
        MathTools tools = new MathTools();

        System.out.println(tools.max(5.7,9));
        System.out.println(tools.max(5,6,8,3));
        //      System.out.println(tools.max(5.7,9.2,6.9)); //没有兼容的
    }
}
```

3、多个方法可以匹配或兼容

```
package com.atguigu.test06.overload;

public class MathTools {
    //求两个整数的最大值
    public int max(int a,int b){
        return a>b?a:b;
    }

    //求两个小数的最大值
    public double max(double a, double b){
        return a>b?a:b;
    }

    //求三个整数的最大值
    public int max(int a, int b, int c){
        return max(max(a,b),c);
    }

    //求n整数的最大值
    public int max(int... nums){
        int max = nums[0];//如果没有传入整数，或者传入null，这句代码会报异常
        for (int i = 1; i < nums.length; i++) {
            if(nums[i] > max){
```

```

        max = nums[i];
    }
}
return max;
}

/*
//求n整数的最大值
public int max(int[] nums){ //编译就报错，与(int... nums)无法区分
    int max = nums[0];//如果没有传入整数，或者传入null，这句代码会报异常
    for (int i = 1; i < nums.length; i++) {
        if(nums[i] > max){
            max = nums[i];
        }
    }
    return max;
} */

/*
//求n整数的最大值
public int max(int first, int... nums){ //当前类不报错，但是调用时会引起多个方法同时匹配
    int max = first;
    for (int i = 0; i < nums.length; i++) {
        if(nums[i] > max){
            max = nums[i];
        }
    }
    return max;
} */
}

```

5.7 方法的递归调用

递归调用：方法自己调用自己的现象就称为递归。

递归的分类：

- 递归分为两种，直接递归和间接递归。
- 直接递归称为方法自身调用自己。
- 间接递归可以A方法调用B方法，B方法调用C方法，C方法调用A方法。

注意事项：

- 递归一定要有条件限定，保证递归能够停止下来，否则会发生栈内存溢出。
- 在递归中虽然有限定条件，但是递归深度不能太深，否则效率低下，或者也会发生栈内存溢出。
 - 能够使用循环代替的，尽量使用循环代替递归

案例：计算斐波那契数列（Fibonacci）的第n个值，斐波那契数列满足如下规律，

1, 1, 2, 3, 5, 8, 13, 21, ...

即从第三个数开始，一个数等于前两个数之和。假设 $f(n)$ 代表斐波那契数列的第n个值，那么 $f(n)$ 满足：

$f(n) = f(n-2) + f(n-1);$

```
package com.atguigu.test07.recursion;

public class FibonacciTest {
    public static void main(String[] args) {
        FibonacciTest t = new FibonacciTest();
        //创建FibonacciTest的对象，目的是为了调用f方法

        for(int i=1; i<=10; i++){
            System.out.println("斐波那契数列第" +i +"个数:" + t.f(i));
        }

        System.out.println(t.f(20));//6765
    }

    //使用递归的写法
    int f(int n){//计算斐波那契数列第n个值是多少
        if(n<1){//负数是返回特殊值1，表示不计算负数情况
            return 1;
        }
        if(n==1 || n==2){
            return 1;
        }
        return f(n-2) + f(n-1);
    }
}
```

```
package com.atguigu.exer.recursion;

public class FibonacciTest {
    public static void main(String[] args) {
        FibonacciTest t = new FibonacciTest();
        //创建FibonacciTest的对象，目的是为了调用f方法

        for(int i=1; i<=10; i++){
            System.out.println("斐波那契数列第" +i +"个数:" + t.fValue(i));
        }

        System.out.println(t.fValue(20));//6765
    }

    //不用递归
    int fValue(int n){//计算斐波那契数列第n个值是多少
        if(n<1){//负数是返回特殊值1，表示不计算负数情况
            return 1;
        }
        if(n==1 || n==2){
            return 1;
        }
    }
}
```

```

//从第三个数开始， 等于 前两个整数相加
int beforeBefore = 1; //相当于n=1时的值
int before = 1;//相当于n=2时的值
int current = beforeBefore + before; //相当于n=3的值
//再完后
for(int i=4; i<=n; i++){
    beforeBefore = before;
    before = current;
    current = beforeBefore + before;
    /*
    假设i=4
        beforeBefore = before; //相当于n=2时的值
        before = current; //相当于n=3的值
        current = beforeBefore + before; //相当于n = 4的值
    假设i=5
        beforeBefore = before; //相当于n=3的值
        before = current; //相当于n = 4的值
        current = beforeBefore + before; //相当于n = 5的值
        ...
    */
}
return current;
}

}

```

5.8 对象数组

数组是用来存储一组数据的容器，一组基本数据类型的数据可以用数组装，那么一组对象也可以使用数组来装。

即数组的元素可以是基本数据类型，也可以是引用数据类型。当元素是引用数据类型时，我们称为对象数组。

注意：对对象数组，首先要创建数组对象本身，即确定数组的长度，然后再创建每一个元素对象，如果不创建，数组的元素的默认值就是null，所以很容易出现空指针异常NullPointerException。

5.8.1 对象数组的声明和使用

案例：

(1) 定义矩形类，包含长、宽属性，area()求面积方法，perimeter()求周长方法，String getInfo()返回圆对象的详细信息的方法

(2) 在测试类中创建长度为5的Rectangle[]数组，用来装3个矩形对象，并给3个矩形对象的长分别赋值为10,20,30，宽分别赋值为5,15,25，遍历输出

```
package com.atguigu.test08.array;
```

```

public class Rectangle {
    double length;
    double width;

    double area(){//面积
        return length * width;
    }

    double perimeter(){//周长
        return 2 * (length + width);
    }

    String getInfo(){
        return "长: " + length +
            ", 宽: " + width +
            ", 面积: " + area() +
            ", 周长: " + perimeter();
    }
}

```

```

package com.atguigu.test08.array;

public class ObjectArrayTest {
    public static void main(String[] args) {
        //声明并创建一个长度为3的矩形对象数组
        Rectangle[] array = new Rectangle[3];

        //创建3个矩形对象，并为对象的实例变量赋值,
        //3个矩形对象的长分别是10,20,30
        //3个矩形对象的宽分别是5,15,25
        //调用矩形对象的getInfo()返回对象信息后输出
        for (int i = 0; i < array.length; i++) {
            //创建矩形对象
            array[i] = new Rectangle();

            //为矩形对象的成员变量赋值
            array[i].length = (i+1) * 10;
            array[i].width = (2*i+1) * 5;

            //获取并输出对象对象的信息
            System.out.println(array[i].getInfo());
        }
    }
}

```

5.8.2 对象数组的内存图分析

对象数组中数组元素存储的是元素对象的首地址。

```

public class ObjectArrayTest {
    public static void main(String[] args) {
        // 声明并创建一个长度为3的矩形对象数组
        Rectangle[] array = new Rectangle[3];

        // 创建3个矩形对象，并为对象的实例变量赋值
        // 3个矩形对象的长分别是10, 20, 30
        // 3个矩形对象的宽分别是5, 15, 25。
        // 调用矩形对象的getInfo()返回对象信息后输出
        for (int i = 0; i < array.length; i++) {
            // 创建矩形对象
            array[i] = new Rectangle();

            // 为矩形对象的成员变量赋值
            array[i].length = (i+1) * 10;
            array[i].width = (2*i+1) * 5;

            // 获取并输出对象对象的信息
            System.out.println(array[i].getInfo());
        }
    }
}

```

```

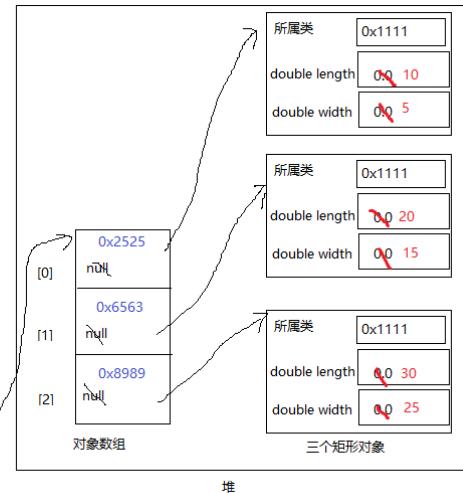
public class Rectangle {
    double length;
    double width;

    double area() { return length * width; }

    double perimeter() { return 2 * (length + width); }

    String getInfo(){
        return "长: " + length +
               ", 宽: " + width +
               ", 面积: " + area() +
               ", 周长: " + perimeter();
    }
}

```



5.8.3 二维数组

1. 什么是二维数组？

一个一维数组只能存储一组同类型的数据，如果需要同时存储多组同类型的数据，就需要使用二维数组。例如，使用一维数组存储一个小组的学员成绩，使用二维数组可以存储多个小组的学员成绩。

- 二维数组：本质上就是元素为一维数组的一个数组。
- 二维数组的标记：`[][]`

```
int[][] arr; // arr是一个二维数组，可以看成元素是int[]一维数组类型的一维数组
```

==二维数组也可以看成一个二维表，行*列组成的二维表==，只不过这个二维表，每一行的列数还可能不同。但是每一个单元格中的元素的数据类型是一致的，例如：都是int，都是String等。

==二维数组也可以看成一个一维数组，只是此时元素是一维数组对象==。

| | | | | | | | | | | | | | | | | | | | |
|------|---|----|----|----|---|---|---|---|---|---|---|---|---|----|----|----|----|---|---|
| 二维表1 | <table border="1"><tr><td>23</td><td>6</td><td>78</td><td>1</td><td>0</td><td>5</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr><tr><td>90</td><td>34</td><td>78</td><td>67</td><td>3</td><td>6</td></tr></table> | 23 | 6 | 78 | 1 | 0 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 90 | 34 | 78 | 67 | 3 | 6 |
| 23 | 6 | 78 | 1 | 0 | 5 | | | | | | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | | | | | | | | | | | | | | |
| 90 | 34 | 78 | 67 | 3 | 6 | | | | | | | | | | | | | | |

| | | | | | | | | | | | | | | |
|------|---|---|---|----|----|---|---|---|----|----|---|---|---|---|
| 二维表2 | <table border="1"><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>2</td><td>4</td><td>6</td><td>8</td><td>10</td><td>12</td></tr><tr><td>1</td><td>3</td><td>5</td><td>7</td></tr></table> | 4 | 5 | 6 | 2 | 4 | 6 | 8 | 10 | 12 | 1 | 3 | 5 | 7 |
| 4 | 5 | 6 | | | | | | | | | | | | |
| 2 | 4 | 6 | 8 | 10 | 12 | | | | | | | | | |
| 1 | 3 | 5 | 7 | | | | | | | | | | | |

二维数组可以看成二维表，也可以看成元素是一维数组对象的一维数组。

2. 二维数组的声明

二维数组声明的语法格式：

```
//推荐  
元素的数据类型[][] 二维数组的名称；  
  
//不推荐  
元素的数据类型 二维数组名[][]；  
//不推荐  
元素的数据类型[] 二维数组名[]；
```

例如：

```
public class Test20TwoDimensionalArrayDefine {  
    public static void main(String[] args) {  
        //存储多组成绩  
        int[][] grades;  
  
        //存储多组姓名  
        String[][] names;  
    }  
}
```

面试：

```
int[] x, y[];  
//x是一维数组，y是二维数组
```

3.二维数组的静态初始化

静态初始化就是用静态数据（编译时已知）为数组初始化。

```
//以下格式要求声明与静态初始化必须一起完成  
元素的数据类型[][] 二维数组的名称 = {  
    {元素1, 元素2, 元素3 . . . },  
    {第二行的值列表},  
    ...  
    {第n行的值列表}  
};  
  
元素的数据类型[][] 二维数组名 = new 元素的数据类型[][]{  
    {元素1, 元素2, 元素3 . . . },  
    {第二行的值列表},  
    ...  
    {第n行的值列表}  
};  
  
元素的数据类型[][] 二维数组名；  
二维数组名 = new 元素的数据类型[][]{  
    {元素1, 元素2, 元素3 . . . },  
    {第二行的值列表},  
    ...  
    {第n行的值列表}  
};
```

如果是静态初始化，右边new 数据类型[][]中不能写数字，因为行数和列数，由{}的元素个数决定

举例：

```
int[][] arr = {{1,2,3},{4,5,6},{7,8,9,10}};//声明与初始化必须在一句完成

int[][] arr = new int[][]{{1,2,3},{4,5,6},{7,8,9,10}};

int[][] arr;
arr = new int[][]{{1,2,3},{4,5,6},{7,8,9,10}};

arr = new int[3][3]{{1,2,3},{4,5,6},{7,8,9,10}};//错误，静态初始化右边new 数据类型[] [] 中不能写数字
```

二维数组静态初始化演示：

```
public class Test21TwoDimensionalArrayInitialize {
    public static void main(String[] args) {
        //存储多组成绩
        int[][] grades = {
            {89,75,99,100},
            {88,96,78,63,100,86},
            {56,63,58},
            {99,66,77,88}
        };

        //存储多组姓名
        String[][] names = {
            {"张三", "李四", "王五", "赵六"},
            {"刘备", "关羽", "张飞", "诸葛亮", "赵云", "马超"},
            {"曹丕", "曹植", "曹冲"},
            {"孙权", "周瑜", "鲁肃", "黄盖"}
        };
    }
}
```

4.二维数组的使用

因为二维数组是用来存储多组数据的，因此要比一维数组麻烦一些，需要我们搞清楚如下几个概念：

- 二维数组的长度/行数：二维数组名.length
- 二维数组的某一行：二维数组名[行下标]，此时相当于获取其中一组数据。它本质上是一个一维数组。行下标的范围：[0, 二维数组名.length-1]。此时把二维数组看成一维数组的话，元素是行对象。
- 某一行的列数：二维数组名[行下标].length，因为二维数组的每一行是一个一维数组。
- 某一个元素：二维数组名[行下标][列下标]，即先确定行/组，再确定列。

```
public class Test22TwoDimensionalArrayUse {
    public static void main(String[] args){
        //存储3个小组的学员的成绩，分开存储，使用二维数组。
        /*
        int[][] scores1;
        int scores2[][];
```

```

int[] scores3[];/*
int[][] scores = {
    {85,96,85,75},
    {99,96,74,72,75},
    {52,42,56,75}
};

System.out.println(scores);//[I@15db9742
System.out.println("一共有" + scores.length +"组成绩.");

//[: 代表二维数组, I代表元素类型是int
System.out.println(scores[0]);//[I@6d06d69c
//[:] 代表一维数组, I代表元素类型是int
System.out.println(scores[1]);//[I@7852e922
System.out.println(scores[2]);//[I@4e25154f
//System.out.println(scores[3]);//ArrayIndexOutOfBoundsException: 3

System.out.println("第1组有" + scores[0].length +"个学员.");
System.out.println("第2组有" + scores[1].length +"个学员.");
System.out.println("第3组有" + scores[2].length +"个学员.");

System.out.println("第1组的每一个学员成绩如下: ");
//第一行的元素
System.out.println(scores[0][0]);//85
System.out.println(scores[0][1]);//96
System.out.println(scores[0][2]);//85
System.out.println(scores[0][3]);//75
//System.out.println(scores[0][4]);//java.lang.ArrayIndexOutOfBoundsException: 4
}

}
}

```

5.二维数组的遍历

```

for(int i=0; i<二维数组名.length; i++){ //二维数组对象.length
    for(int j=0; j<二维数组名[i].length; j++){//二维数组行对象.length
        System.out.print(二维数组名[i][j]);
    }
    System.out.println();
}

```

```

public class Test23TwoDimensionalArrayIterate {
    public static void main(String[] args) {
        //存储3个小组的学员的成绩，分开存储，使用二维数组。
        int[][] scores = {
            {85,96,85,75},
            {99,96,74,72,75},
            {52,42,56,75}
        };

        System.out.println("一共有" + scores.length +"组成绩.");
        for (int i = 0; i < scores.length; i++) {

```

```

        System.out.print("第" + (i+1) +"组有" + scores[i].length + "个学员, 成绩如下: ");
        for (int j = 0; j < scores[i].length; j++) {
            System.out.print(scores[i][j]+\t");
        }
        System.out.println();
    }
}
}
}

```

6.二维数组动态初始化

如果二维数组的每一个数据，甚至是每一行的列数，需要后期单独确定，那么就只能使用动态初始化方式了。动态初始化方式分为两种格式：

(1) 规则二维表：每一行的列数是相同的

```
// (1) 确定行数和列数
元素的数据类型[][] 二维数组名 = new 元素的数据类型[m][n];
    m:表示这个二维数组有多少个一维数组。或者说一共二维表有几行
    n:表示每一个一维数组的元素有多少个。或者说每一行共有一个单元格
```

//此时创建完数组，行数、列数确定，而且元素也都有默认值

```
// (2) 再为元素赋新值
二维数组名[行下标][列下标] = 值;
```

```
/*
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
*/
public class Test24SameElementCount {
    public static void main(String[] args) {
        //1、声明二维数组，并确定行数和列数
        int[][] arr = new int[4][5];

        //2、确定元素的值
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr.length; j++) {
                arr[i][j] = i + 1;
            }
        }

        //3、遍历显示
        for(int i=0; i<arr.length; i++){
            for(int j=0; j<arr[i].length; j++){
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
    }
}
}
```

(2) 不规则：每一行的列数不一样

```
// (1) 先确定总行数
元素的数据类型[][] 二维数组名 = new 元素的数据类型[总行数][];

//此时只是确定了总行数，每一行里面现在是null

// (2) 再确定每一行的列数，创建每一行的一维数组
二维数组名[行下标] = new 元素的数据类型[该行的总列数];

//此时已经new完的行的元素就有默认值了，没有new的行还是null

//(3)再为元素赋值
二维数组名[行下标] [列下标] = 值;
```

```
/*
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
*/
public class Test25DifferentElementCount {
    public static void main(String[] args){
        //1、声明一个二维数组，并且确定行数
        //因为每一行的列数不同，这里无法直接确定列数
        int[][] arr = new int[5][];
        //2、确定每一行的列数
        for(int i=0; i<arr.length; i++){
            /*
            arr[0] 的列数是1
            arr[1] 的列数是2
            arr[2] 的列数是3
            arr[3] 的列数是4
            arr[4] 的列数是5
            */
            arr[i] = new int[i+1];
        }
        //3、确定元素的值
        for(int i=0; i<arr.length; i++){
            for(int j=0; j<arr[i].length; j++){
                arr[i][j] = i+1;
            }
        }
        //4、遍历显示
    }
}
```

```

        for(int i=0; i<arr.length; i++){
            for(int j=0; j<arr[i].length; j++){
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }

    }
}

```

7.空指针异常

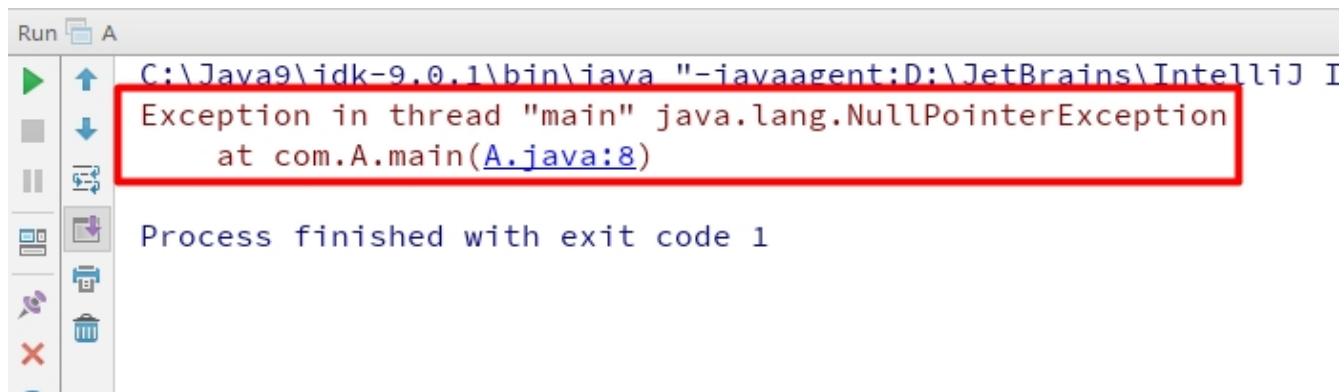
观察一下代码，运行后会出现什么结果。

```

public class Test26NullPointerException {
    public static void main(String[] args) {
        //定义数组
        int[][] arr = new int[3][];
        System.out.println(arr[0][0]); //NullPointerException
    }
}

```

因为此时数组的每一行还未分配具体存储元素的空间，此时arr[0]是null，此时访问arr[0][0]会抛出 NullPointerException 空指针异常。

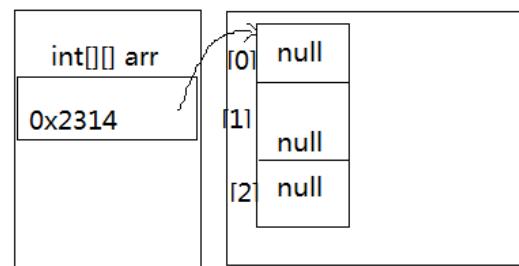


空指针异常在内存图中的表现

```

public static void main(String[] args) {
    //定义数组
    int[][] arr = new int[3][];
    System.out.println(arr[0][0]); //NullPointerException
}

```



8.二维数组的内存图分析

二维数组本质上是元素类型是一维数组的一维数组。

```

int[][] arr = {
    {1},
    {2,2},
    {3,3,3},
    {4,4,4,4},
    {5,5,5,5,5}
};

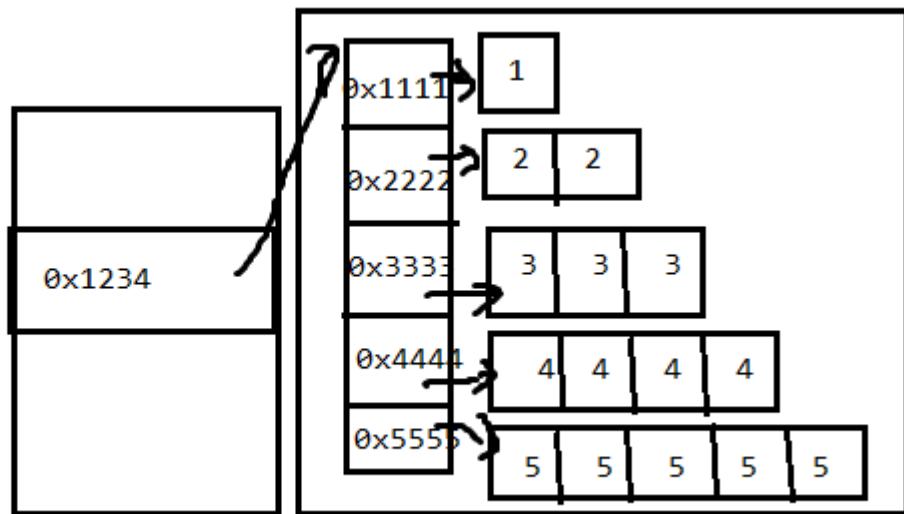
```

```

int[][] arr = {
    {1},
    {2,2},
    {3,3,3},
    {4,4,4,4},
    {5,5,5,5,5}
};

```

相当于元素是
5个一维数组
的一维数组



```

//1、声明二维数组，并确定行数和列数
int[][] arr = new int[4][5];

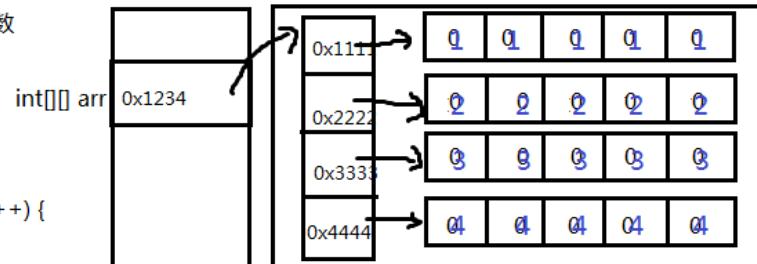
//2、确定元素的值
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr.length; j++) {
        arr[i][j] = i + 1;
    }
}

```

```

1 1 1 1 1 //1、声明二维数组，并确定行数和列数
2 2 2 2 2 int[] arr = new int[4][5];
3 3 3 3 3 //2、确定元素的值
4 4 4 4 4 for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr.length; j++) {
                arr[i][j] = i + 1;
            }
        }

```



```

//1、声明一个二维数组，并且确定行数
//因为每一行的列数不同，这里无法直接确定列数
int[][] arr = new int[5][];

//2、确定每一行的列数
for(int i=0; i<arr.length; i++){
    /*

```

```

        arr[0] 的列数是1
        arr[1] 的列数是2
        arr[2] 的列数是3
        arr[3] 的列数是4
        arr[4] 的列数是5
    */
    arr[i] = new int[i+1];
}

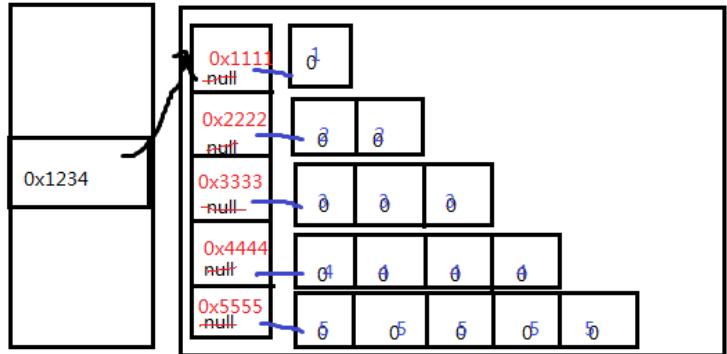
//3、确定元素的值
for(int i=0; i<arr.length; i++){
    for(int j=0; j<arr[i].length; j++){
        arr[i][j] = i+1;
    }
}

```

```

1 //1、声明一个二维数组，并且确定行数
2 //因为每一行的列数不同，这里无法直接确定列数
3 int[][] arr = new int[5][];
4
5 //2、确定每一行的列数
6 for(int i=0; i<arr.length; i++){
7     arr[i] = new int[i+1];
8 }
9
10 //3、确定元素的值
11 for(int i=0; i<arr.length; i++){
12     for(int j=0; j<arr[i].length; j++){
13         arr[i][j] = i+1;
14     }
15 }

```



第六章 面向对象基础--中

6.1 封装

6.1.1 封装概述

1、为什么需要封装？

- 我要用洗衣机，只需要按一下开关和洗涤模式就可以了。有必要了解洗衣机内部的结构吗？有必要碰电动机吗？
- 我们使用的电脑，内部有CPU、硬盘、键盘、鼠标等等，每一个部件通过某种连接方式一起工作，但是各个部件之间又是独立的
- 现实生活中，每一个个体与个体之间是有边界的，每一个团体与团体之间是有边界的，而同一个个体、团体内部的信息是互通的，只是对外有所隐瞒。

面向对象编程语言是对客观世界的模拟，客观世界里每一个事物的内部信息都是隐藏在对象内部的，外界无法直接操作和修改，只能通过指定的方式进行访问和修改。封装可以被认为是一个保护屏障，防止该类的代码和数据被其他类随意访问。适当的封装可以让代码更容易理解与维护，也加强了代码的安全性。

随着我们系统越来越复杂，类会越来越多，那么类之间的访问边界必须把握好，面向对象的开发原则要遵循“高内聚、低耦合”，而“高内聚，低耦合”的体现之一：

- 高内聚：类的内部数据操作细节自己完成，不允许外部干涉；
- 低耦合：仅对外暴露少量的方法用于使用

隐藏对象内部的复杂性，只对外公开简单和可控的访问方式，从而提高系统的可扩展性、可维护性。通俗的讲，把该隐藏的隐藏起来，该暴露的暴露出来。这就是封装性的设计思想。

2、如何实现封装呢？

实现封装就是指控制类或成员的可见性范围？这就需要依赖访问控制修饰符，也称为权限修饰符来控制。

权限修饰符：public,protected,缺省,private

| 修饰符 | 本类 | 本包 | 其他包子类 | 其他包非子类 |
|-----------|----|----|-------|--------|
| private | √ | ✗ | ✗ | ✗ |
| 缺省 | √ | √ | ✗ | ✗ |
| protected | √ | √ | √ | ✗ |
| public | √ | √ | √ | √ |

外部类：public和缺省

成员变量、成员方法、构造器、成员内部类：public,protected,缺省,private

6.1.2 成员变量/属性私有化问题

成员变量（field）私有化之后，提供标准的get/set方法，我们把这种成员变量也称为属性（property）。

或者可以说只要能通过get/set操作的就是事物的属性，哪怕它没有对应的成员变量。

1、成员变量封装的目的

- 隐藏类的实现细节
- 让使用者只能通过事先预定的方法来访问数据，从而可以在该方法里面加入控制逻辑，限制对成员变量的不合理访问。还可以进行数据检查，从而有利于保证对象信息的完整性。
- 便于修改，提高代码的可维护性。主要说的是隐藏的部分，在内部修改了，如果其对外可以的访问方式不变的话，外部根本感觉不到它的修改。例如：Java8->Java9，String从char[]转为byte[]内部实现，而对外的方法不变，我们使用者根本感觉不到它内部的修改。

2、实现步骤

1. 使用 private 修饰成员变量

```
private 数据类型 变量名 ;
```

代码如下：

```
public class Person {  
    private String name;  
    private int age;  
    private boolean marry;  
}
```

2. 提供 `getXXX` 方法 / `setXXX` 方法，可以访问成员变量，代码如下：

```
public class Person {  
    private String name;  
    private int age;  
    private boolean marry;  
  
    public void setName(String n) {  
        name = n;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setAge(int a) {  
        age = a;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setMarry(boolean m){  
        marry = m;  
    }  
  
    public boolean isMarry(){  
        return marry;  
    }  
}
```

3、测试

```
package com.atguigu.encapsulation;  
  
public class TestPerson {  
    public static void main(String[] args) {  
        Person p = new Person();  
  
        //实例变量私有化，跨类是无法直接使用的  
/*          p.name = "张三";  
          p.age = 23;  
          p.marry = true;*/
```

```
p.setName("张三");
System.out.println("p.name = " + p.getName());

p.setAge(23);
System.out.println("p.age = " + p.getAge());

p.setMarry(true);
System.out.println("p.marry = " + p.isMarry());
}

}
```

6.1.3 IDEA自动生成get/set方法模板

1、如何解决局部变量与实例变量同名问题

当局部变量与实例变量（非静态成员变量）同名时，在实例变量必须前面加“this.”

```
package com.atguigu.encapsulation;

public class Employee {
    private String name;
    private int age;
    private boolean marry;

    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 boolean isMarry() {
        return marry;
    }

    public void setMarry(boolean marry) {
        this.marry = marry;
    }
}
```

```

package com.atguigu.encapsulation;

public class TestEmployee {
    public static void main(String[] args) {
        Employee e = new Employee();

        e.setName("张三");
        System.out.println("e.name = " + e.getName());

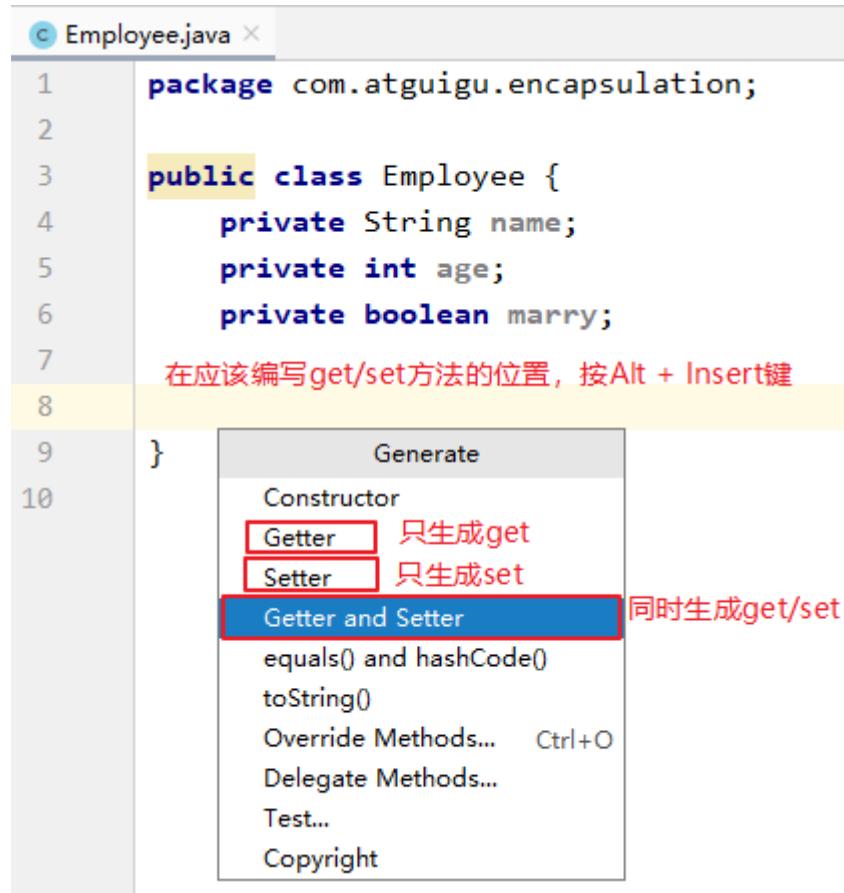
        e.setAge(23);
        System.out.println("e.age = " + e.getAge());

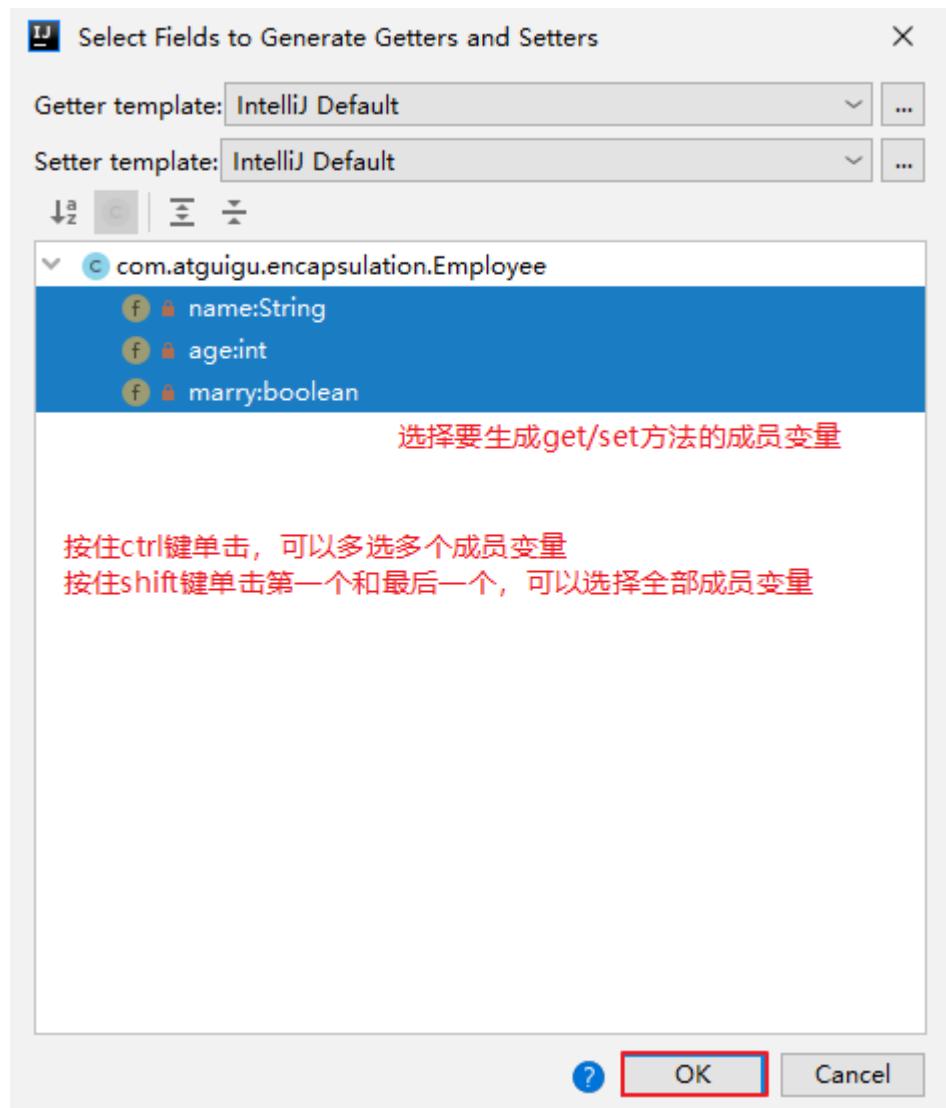
        e.setMarry(true);
        System.out.println("e.marry = " + e.isMarry());
    }
}

```

2、IDEA自动生成get/set方法模板

- 大部分键盘模式按Alt + Insert键。
- 部分键盘模式需要按Alt + Insert + Fn键。
- Mac电脑快捷键需要单独设置





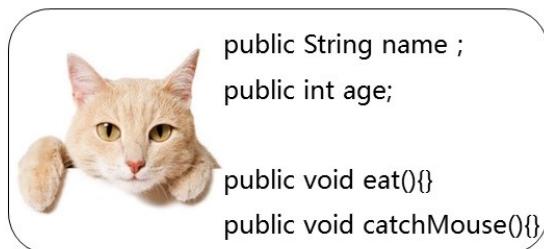
6.2 继承

6.2.1 继承的概述

继承有延续（下一代延续上一代的基因、财富）、扩展（下一代和上一代又有所不同）的意思。

Java中的继承

如图所示：

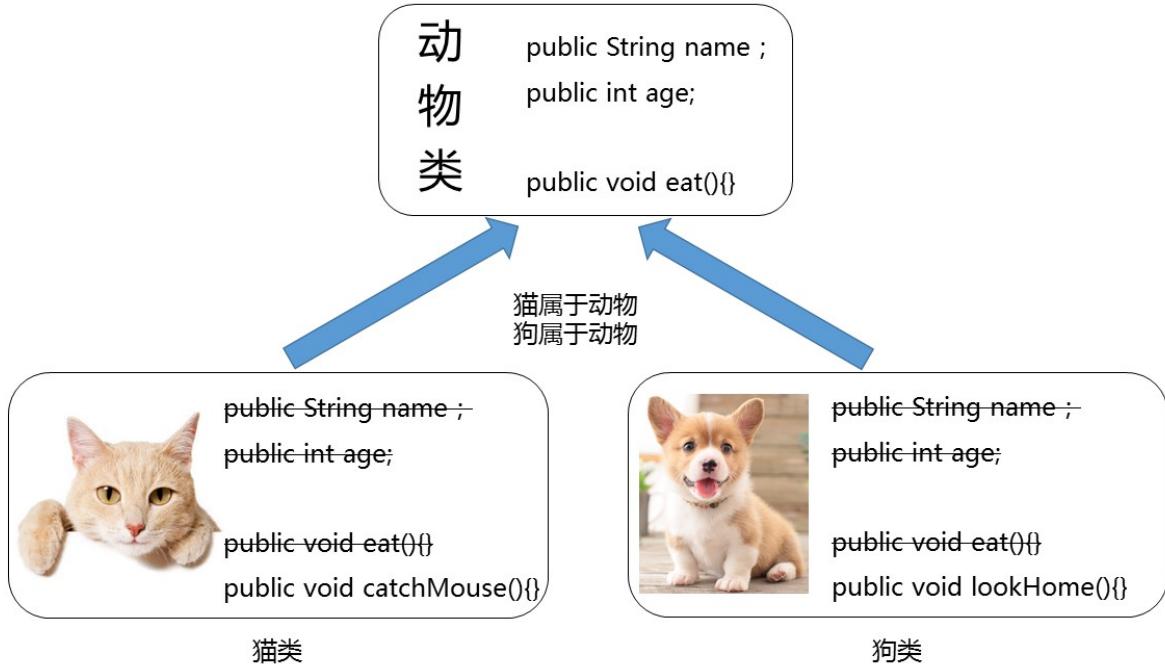


猫类



狗类

多个类中存在相同属性和行为时，将这些内容抽取到单独一个类中，那么多个类中无需再定义这些属性和行为，只需要和抽取出来的类构成某种关系。如图所示：



其中，多个类可以称为**子类**，也叫**派生类**；多个类抽取出来的这个类称为**父类、超类 (superclass)** 或者**基类**。

继承描述的是事物之间的所属关系，这种关系是：`is-a` 的关系。例如，图中猫属于动物，狗也属于动物。可见，父类更通用或更一般，子类更具体。我们通过继承，可以使多种事物之间形成一种关系体系。

继承的好处

- 提高代码的复用性。
- 提高代码的扩展性。
- 表示类与类之间的`is-a`关系

6.2.2 继承的语法格式

通过 `extends` 关键字，可以声明一个子类继承另外一个父类，定义格式如下：

```

【修饰符】 class 父类 {
    ...
}

【修饰符】 class 子类 extends 父类 {
    ...
}

```

1、父类

```

package com.atguigu.inherited.grammar;

/*
 * 定义动物类Animal，做为父类
 */

```

```
public class Animal {  
    // 定义name属性  
    String name;  
    // 定义age属性  
    int age;  
  
    // 定义动物的吃东西方法  
    public void eat() {  
        System.out.println(age + "岁的"  
                           + name + "在吃东西");  
    }  
}
```

2、子类

```
package com.atguigu.inherited.grammar;  
  
/*  
 * 定义猫类Cat 继承 动物类Animal  
 */  
public class Cat extends Animal {  
    int count;//记录每只猫抓的老鼠数量  
  
    // 定义一个猫抓老鼠的方法catchMouse  
    public void catchMouse() {  
        count++;  
        System.out.println("抓老鼠, 已经抓了"  
                           + count + "只老鼠");  
    }  
}
```

3、测试类

```
package com.atguigu.inherited.grammar;  
  
public class TestCat {  
    public static void main(String[] args) {  
        // 创建一个猫类对象  
        Cat cat = new Cat();  
        // 为该猫类对象的name属性进行赋值  
        cat.name = "Tom";  
        // 为该猫类对象的age属性进行赋值  
        cat.age = 2;  
        // 调用该猫继承来的eat()方法  
        cat.eat();  
        // 调用该猫的catchMouse()方法  
        cat.catchMouse();  
        cat.catchMouse();  
        cat.catchMouse();  
    }  
}
```

6.2.3 继承的特点

1. 子类会继承父类所有的实例变量和实例方法

从类的定义来看，类是一类具有相同特性的事物的抽象描述。父类是所有子类共同特征的抽象描述。而实例变量和实例方法就是事物的特征，那么父类中声明的实例变量和实例方法代表子类事物也有这个特征。

- 当子类对象被创建时，在堆中给对象申请内存时，就要看子类和父类都声明了什么实例变量，这些实例变量都要分配内存。
- 当子类对象调用方法时，编译器会先在子类模板中看该类是否有这个方法，如果没找到，会看它的父类甚至父类的父类是否声明了这个方法，遵循从下往上找的顺序，找到了就停止，一直到根父类都没有找到，就会报编译错误。

所以继承意味着子类的对象除了看子类的类模板还要看父类的类模板。

```
Animal.java
package com.atguigu.inherited.grammar;
/*
 * 定义动物类Animal, 做为父类
 */
public class Animal {
    // 定义name属性
    String name;
    // 定义age属性
    int age;

    // 定义动物的吃东西方法
    public void eat() {
        System.out.println(age + "岁的"
                           + name + "在吃东西");
    }
}

Cat.java
package com.atguigu.inherited.grammar;
/*
 * 定义猫类Cat 继承 动物类Animal
 */
public class Cat extends Animal {
    int count; //记录每只猫抓的老鼠数量

    // 定义一个猫抓老鼠的方法catchMouse
    public void catchMouse() {
        count++;
        System.out.println("抓老鼠, 已经抓了"
                           + count + "只老鼠");
    }
}

TestCat.java
package com.atguigu.inherited.grammar;
public class TestCat {
    public static void main(String[] args) {
        // 创建一个猫类对象
        Cat cat = new Cat();
        // 为该猫类对象的name属性进行赋值
        cat.name = "Tom";
        // 为该猫类对象的age属性进行赋值
        cat.age = 2;
        // 调用该猫继承的eat()方法
        cat.eat();
        // 调用该猫的catchMouse()方法
        cat.catchMouse();
        cat.catchMouse();
        cat.catchMouse();
    }
}
```

2. Java只支持单继承，不支持多重继承

```
public class A{}
class B extends A{}

//一个类只能有一个父类，不可以有多个直接父类。
class C extends B{}      //ok
class C extends A, B... //error
```

3. Java支持多层次继承(继承体系)

```
class A{}
class B extends A{}
class C extends B{}
```

顶层父类是Object类。所有的类默认继承Object，作为父类。

4. 一个父类可以同时拥有多个子类

```
class A{}  
class B extends A{}  
class D extends A{}  
class E extends A{}
```

6.2.4 IDEA中如何查看继承关系

1.子类和父类是一种相对的概念

例如：B类对于A来说是子类，但是对于C类来说是父类

2.查看继承关系快捷键

例如：选择A类名，按Ctrl + H就会显示A类的继承树。



: A类的父类和子类



: A类的父类



: A类的所有子类

例如：在类继承目录树中选中某个类，比如C类，按Ctrl+ Alt+U就会用图形化方式显示C类的继承祖宗

The screenshot shows the Java code for class A.java and its inheritance hierarchy. The code defines classes A, B, C, D, and E. Class A is selected, highlighted with a red border. A tooltip above the code says "选择A类, 按Ctrl+H". In the right-hand tool window, the "Hierarchy: Subtypes of A" tab is active, showing a tree view of subclasses: A, B, C, D, and E. Class C is selected, highlighted with a red border. A tooltip above the tree says "选择C, 按Ctrl+Alt+U". To the right of the tree is a "Diagram for..." panel displaying a vertical inheritance hierarchy diagram. At the top is class A, followed by B, and then C at the bottom. Arrows point upwards from B to A and from C to B, indicating the inheritance relationship.

6.2.5 权限修饰符限制问题

权限修饰符：public,protected,缺省,private

| 修饰符 | 本类 | 本包 | 其他包子类 | 其他包非子类 |
|-----------|----|----------------|-----------------|--------|
| private | √ | ✗ | ✗ | ✗ |
| 缺省 | √ | ✓ (本包子类非子类都可见) | ✗ | ✗ |
| protected | √ | ✓ (本包子类非子类都可见) | ✓ (其他包仅限于子类中可见) | ✗ |
| public | √ | ✓ | ✓ | ✓ |

外部类：public和缺省

成员变量、成员方法等：public,protected,缺省,private

1、外部类要跨包使用必须是public，否则仅限于本包使用

(1) 外部类的权限修饰符如果缺省，本包使用没问题

```

package com.atguigu.inherited.modifier;
public class Father {
    private int a; Father类有public修饰
    int b;
    protected int c;
    public int d;
}
class Son extends Father {
    public void method(){}
}
class Mother {
}
class Daughter extends Mother {
}

public class TestFamily {
    public static void main(String[] args) {
        Father f = new Father();
        Mother m = new Mother();
    }
}

```

package语句只要相同，就是同一个包。
同一个包中public修饰的类和没有public修饰的类，在本包中都可以直接使用。

(2) 外部类的权限修饰符如果缺省，跨包使用有问题

```

package com.atguigu.inherited.modifier;
public class Father {
    private int a;
    int b;
    protected int c;
    public int d;
}

package com.atguigu.inherited.other;
import com.atguigu.inherited.modifier.Father;
//IllegitimateChild: 私生子，非婚生子女
public class IllegitimateChild extends Father{
    public void method(){}
}

package com.atguigu.inherited.other;
import com.atguigu.inherited.modifier.Mother;
public class IllegitimateChild2 extends Mother{
}

public class TestFatherMother {
    public static void main(String[] args) {
        Father f = new Father();
        //Mother类没有public修饰
        //跨包不可以使用，import也没用
        Mother m = new Mother();
    }
}

```

只要package语句不完全一致就是不同包。
外部类没有public修饰就不能跨包使用

2、成员的权限修饰符问题

(1) 本包下使用：成员的权限修饰符可以是public、protected、缺省

The screenshot shows four Java code editors side-by-side:

- Father.java**: A class with members `a`, `b`, `c`, and `d`. A note says: "只要package语句完全相同就是本包" (As long as the package statement is completely the same, it's the same package).
- Son.java**: Extends `Father`. It prints `a`, `b`, `c`, and `d`. Notes: "关于成员 (例如, 成员变量或成员方法等)" (About members (e.g., member variables or member methods)), "private: 仅限于本类中使用, 只要跨类就不能使用, 不管包不管是否是子类" (private: only within the class, cannot be used across classes, regardless of package or inheritance), "缺省、protected、public: 本包下都可以使用, 子类可以直接用, 非子类只要有对象就可以使用" (Default, protected, public: can be used within the package, subclasses can use directly, non-subclasses can use objects).
- Stepchild.java**: Extends `Father`. It prints `a`, `b`, `c`, and `d`. Notes: "关于成员 (例如, 成员变量或成员方法等)" (About members (e.g., member variables or member methods)), "private: 仅限于本类中使用, 只要跨类就不能使用, 不管包不管是否是子类" (private: only within the class, cannot be used across classes, regardless of package or inheritance), "缺省、protected、public: 本包下都可以使用, 子类可以直接用, 非子类只要有对象就可以使用" (Default, protected, public: can be used within the package, subclasses can use directly, non-subclasses can use objects).
- TestFamily.java**: A test driver. It creates `Mother`, `Father`, and `Stepchild` objects and prints their `a`, `b`, `c`, and `d` values.

(2) 跨包下使用: 要求严格

The screenshot shows three Java code editors:

- Father.java**: A class with members `a`, `b`, `c`, and `d`. Notes: "package语句不完全相同就是不同包" (package statements are different, so they are in different packages), "private: 仅限于本类" (private: only within the class), "缺省: 仅限于本包" (default: only within the package), "protected: 跨包仅限于子类中" (protected: only within subclasses across packages), "public: 同模块任意位置" (public: anywhere within the module).
- IllegitimateChild.java**: Extends `Father`. It prints `a`, `b`, `c`, and `d`. Notes: "关于成员 (例如, 成员变量或成员方法等)" (About members (e.g., member variables or member methods)), "private: 仅限于本类中使用, 只要跨类就不能使用" (private: only within the class, cannot be used across classes), "缺省的跨包不可以直接使用" (default cross-package cannot be used directly), "System.out.println("c = " + c); //protected的跨包子类中可以直接使用" (System.out.println("c = " + c); protected cross-package subclasses can be used directly), "System.out.println("d = " + d); //public的同模块任意位置可以使用" (System.out.println("d = " + d); public within the module can be used anywhere).
- Nephew.java**: Extends `Father`. It prints `a`, `b`, `c`, and `d`. Notes: "关于成员 (例如, 成员变量或成员方法等)" (About members (e.g., member variables or member methods)), "private: 仅限于本类中使用, 只要跨类就不能使用" (private: only within the class, cannot be used across classes), "缺省的跨包不可以直接使用" (default cross-package cannot be used directly), "System.out.println("c = " + c); //protected的跨包非子类中不可以直接使用" (System.out.println("c = " + c); protected cross-package non-subclasses cannot be used directly), "System.out.println("d = " + d); //public的同模块任意位置可以使用" (System.out.println("d = " + d); public within the module can be used anywhere).

(3) 跨包使用时, 如果类的权限修饰符缺省, 成员权限修饰符>类的权限修饰符也没有意义

The screenshot shows two Java code editors:

- Mother.java**: A class with member `d`. Notes: "d虽然有public修饰, 但是Mother类不是public的, d跨包也无法使用" (d has public modifier, but Mother class is not public, d cannot be used across packages).
- IllegitimateChild2.java**: Extends `Mother`. It prints `d`. Notes: "关于成员 (例如, 成员变量或成员方法等)" (About members (e.g., member variables or member methods)), "private: 仅限于本类中使用, 只要跨类就不能使用" (private: only within the class, cannot be used across classes), "缺省的跨包不可以直接使用" (default cross-package cannot be used directly), "System.out.println("d = " + d); //public的同模块任意位置可以使用" (System.out.println("d = " + d); public within the module can be used anywhere).

3、父类成员变量私有化 (private)

子类虽会继承父类私有(private)的成员变量, 但子类不能对继承的私有成员变量直接进行访问, 可通过继承的get/set方法进行访问。如图所示:



父类代码：

```
package com.atguigu.inherited.modifier;

public class Person {
    private String name;
    private int 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 String getInfo(){
        return "姓名: " + name + ", 年龄: " + age;
    }
}
```

子类代码：

```
package com.atguigu.inherited.modifier;

public class Student extends Person {
    private int score;

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
```

```

        this.score = score;
    }

    public String getInfo(){
//        return "姓名: " + name + ", 年龄: " + age;
        //在子类中不能直接使用父类私有的name和age
        return "姓名: " + getName() + ", 年龄: " + getAge();
    }
}

```

测试类代码：

```

package com.atguigu.inherited.modifier;

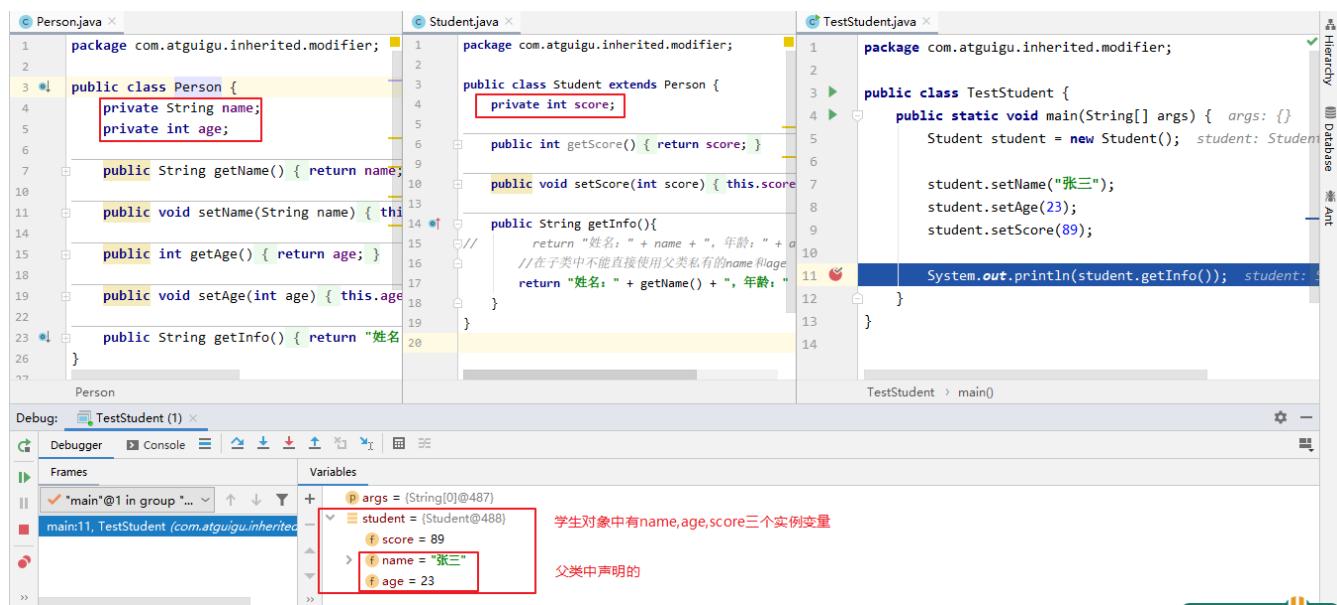
public class TestStudent {
    public static void main(String[] args) {
        Student student = new Student();

        student.setName("张三");
        student.setAge(23);
        student.setScore(89);

        System.out.println(student.getInfo());
    }
}

```

IDEA在Debug模式下查看学生对象信息：



6.2.6 方法重写 (Override)

我们说父类的所有方法子类都会继承，但是当某个方法被继承到子类之后，子类觉得父类原来的实现不适合于子类，该怎么办呢？我们可以进行方法重写 (Override)

1、方法重写

比如新的手机增加来电显示头像的功能，代码如下：

```
package com.atguigu.inherited.method;

public class Phone {
    public void sendMessage(){
        System.out.println("发短信");
    }
    public void call(){
        System.out.println("打电话");
    }
    public void showNum(){
        System.out.println("来电显示号码");
    }
}
```

```
package com.atguigu.inherited.method;

//smartphone: 智能手机
public class Smartphone extends Phone{
    //重写父类的来电显示功能的方法
    public void showNum(){
        //来电显示姓名和图片功能
        System.out.println("显示来电姓名");
        System.out.println("显示头像");
    }
}
```

```
package com.atguigu.inherited.method;

public class TestOverride {
    public static void main(String[] args) {
        // 创建子类对象
        Smartphone sp = new Smartphone();

        // 调用父类继承而来的方法
        sp.call();

        // 调用子类重写的方法
        sp.showNum();
    }
}
```

2、在子类中如何调用父类被重写的方法

```
package com.atguigu.inherited.method;

//smartphone: 智能手机
public class Smartphone extends Phone{
    //重写父类的来电显示功能的方法
```

```

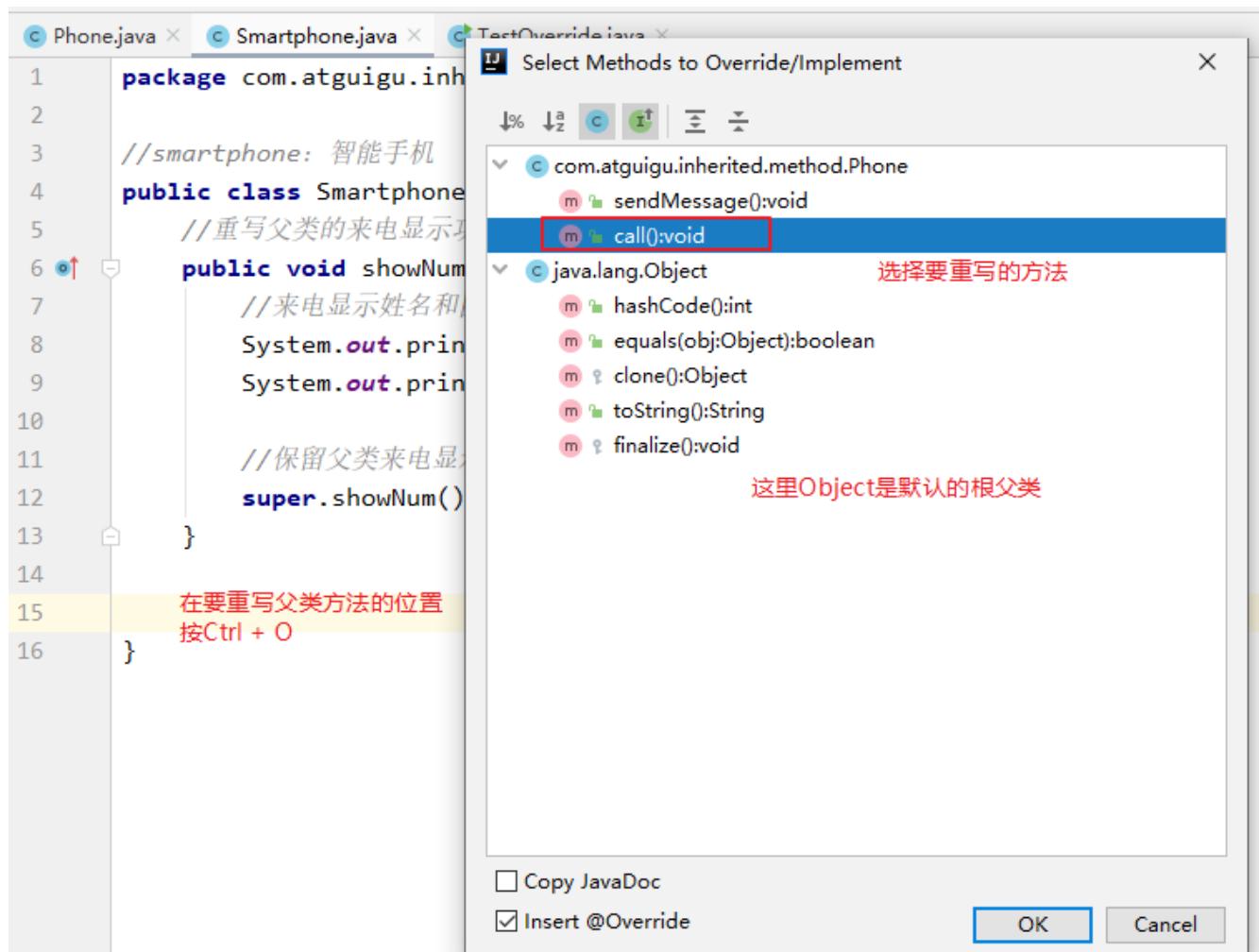
public void showNum(){
    //来电显示姓名和图片功能
    System.out.println("显示来电姓名");
    System.out.println("显示头像");

    //保留父类来电显示号码的功能
    super.showNum(); //此处必须加super., 否则就是无限递归, 那么就会栈内存溢出
}

}

```

3、IDEA重写方法快捷键：Ctrl + O



```

package com.atguigu.inherited.method;

//smartphone: 智能手机
public class Smartphone extends Phone{
    //重写父类的来电显示功能的方法
    public void showNum(){
        //来电显示姓名和图片功能
        System.out.println("显示来电姓名");
        System.out.println("显示头像");

        //保留父类来电显示号码的功能
        super.showNum(); //此处必须加super., 否则就是无限递归, 那么就会栈内存溢出
    }
}

```

```
}

@Override
public void call() {
    super.call();
    System.out.println("视频通话");
}
}
```

@Override：写在方法上面，用来检测是不是满足重写方法的要求。这个注解就算不写，只要满足要求，也是正确的方法覆盖重写。建议保留，这样编译器可以帮助我们检查格式，另外也可以让阅读源代码的程序员清晰的知道这是一个重写的方法。

4、重写方法的要求

1. 必须保证父子类之间重写方法的名称相同。
2. 必须保证父子类之间重写方法的参数列表也完全相同。

2. 子类方法的返回值类型必须【小于等于】父类方法的返回值类型（小于其实就是是它的子类，例如：Student < Person）。

注意：如果返回值类型是基本数据类型和void，那么必须是相同

3. 子类方法的权限必须【大于等于】父类方法的权限修饰符。

注意：public > protected > 缺省 > private

父类私有方法不能重写

跨包的父类缺省的方法也不能重写

5、方法的重载和方法的重写

方法的重载：方法名相同，形参列表不同。不看返回值类型。

方法的重写：见上面。

(1) 同一个类中

```
package com.atguigu.inherited.method;

public class TestOverload {
    public int max(int a, int b){
        return a > b ? a : b;
    }
    public double max(double a, double b){
        return a > b ? a : b;
    }
    public int max(int a, int b,int c){
        return max(max(a,b),c);
    }
}
```

(2) 父子类中

```
package com.atguigu.inherited.method;

public class TestOverloadOverride {
    public static void main(String[] args) {
        Son s = new Son();
        s.method(1); //只有一个形式的method方法

        Daughter d = new Daughter();
        d.method(1);
        d.method(1, 2); //有两个形式的method方法
    }
}

class Father{
    public void method(int i){
        System.out.println("Father.method");
    }
}
class Son extends Father{
    public void method(int i){ //重写
        System.out.println("Son.method");
    }
}
class Daughter extends Father{
    public void method(int i, int j){ //重载
        System.out.println("Daughter.method");
    }
}
```

6.3 多态

多态是继封装、继承之后，面向对象的第三大特性。

生活中，比如求面积的功能，圆、矩形、三角形实现起来是不一样的。跑的动作，小猫、小狗和大象，跑起来是不一样的。再比如飞的动作，昆虫、鸟类和飞机，飞起来也是不一样的。可见，同一行为，通过不同的事物，可以体现出不同的形态。那么此时就会出现各种子类的类型。

6.3.1 多态解决什么样的问题

有的时候，我们在设计一个数组、或一个成员变量、或一个方法的形参、返回值类型时，无法确定它具体的类型，只能确定它是某个系列的类型。

案例：

- (1) 声明一个Dog类，包含public void eat()方法，输出“狗狗啃骨头”
- (2) 声明一个Cat类，包含public void eat()方法，输出“猫咪吃鱼仔”
- (3) 声明一个Person类，
 - 包含宠物属性

- 包含领养宠物方法 public void adopt(宠物类型 pet)
- 包含喂宠物吃东西的方法 public void feed(), 实现为调用宠物对象.eat()方法

```
package com.atguigu.polymorphism.problem;

public class Dog {
    public void eat(){
        System.out.println("狗狗啃骨头");
    }
}
```

```
package com.atguigu.polymorphism.problem;

public class Cat {
    public void eat(){
        System.out.println("猫咪吃鱼仔");
    }
}
```

```
package com.atguigu.polymorphism.problem;

public class Person {
    private Dog dog;

    //adopt: 领养
    public void adopt(Dog dog){
        this.dog = dog;
    }

    //feed: 喂食
    public void feed(){
        if(dog != null){
            dog.eat();
        }
    }
    /*
    问题:
    1、从养狗切换到养猫怎么办?
        修改代码把Dog修改为养猫?
    2、或者有的人养狗，有的人养猫怎么办?
    3、要是同时养多个狗，或猫怎么办?
    4、要是还有更多其他宠物类型怎么办?
    如果Java不支持多态，那么上面的问题将会非常麻烦，代码维护起来很难，扩展性很差。
    */
}
```

6.3.2 多态的形式和体现

1、多态引用

Java规定父类类型的变量可以接收子类类型的对象，这一点从逻辑上也是说得通的。

父类类型 变量名 = 子类对象；

父类类型：指子类继承的父类类型，或者实现的父接口类型。

所以说继承是多态的前提

2、多态引用的表现

表现：编译时类型与运行时类型不一致，编译时看“父类”，运行时看“子类”。

3、多态引用的好处和弊端

弊端：编译时，只能调用父类声明的方法，不能调用子类扩展的方法；

好处：运行时，看“子类”，如果子类重写了方法，一定是执行子类重写的方法体；变量引用的子类对象不同，执行的方法就不同，实现动态绑定。代码编写更灵活、功能更强大，可维护性和扩展性更好了。

4、多态演示

让Dog和Cat都继承Pet宠物类。

```
package com.atguigu.polymorphism.grammar;

public class Pet {
    private String nickname;

    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    public void eat(){
        System.out.println(nickname + "吃东西");
    }
}
```

```
package com.atguigu.polymorphism.grammar;

public class Cat extends Pet {
    //子类重写父类的方法
    @Override
    public void eat() {
        System.out.println("猫咪" + getNickname() + "吃鱼仔");
    }

    //子类扩展的方法
    public void catchMouse() {
        System.out.println("抓老鼠");
    }
}
```

```
package com.atguigu.polymorphism.grammar;

public class Dog extends Pet {
    //子类重写父类的方法
    @Override
    public void eat() {
        System.out.println("狗狗" + getNickname() + "啃骨头");
    }

    //子类扩展的方法
    public void watchHouse() {
        System.out.println("看家");
    }
}
```

```
package com.atguigu.polymorphism.grammar;

public class TestPet {
    public static void main(String[] args) {
        //多态引用
        Pet pet = new Dog();
        pet.setNickname("小白");

        //多态的表现形式
        /*
        编译时看父类：只能调用父类声明的方法，不能调用子类扩展的方法；
        运行时，看“子类”，如果子类重写了方法，一定是执行子类重写的方法体；
        */
        pet.eat(); //运行时执行子类Dog重写的方法
        //pet.watchHouse(); //不能调用Dog子类扩展的方法

        pet = new Cat();
        pet.setNickname("雪球");
        pet.eat(); //运行时执行子类Cat重写的方法
    }
}
```

6.3.3 应用多态解决问题

1、声明变量是父类类型，变量赋值子类对象

- 方法的形参是父类类型，调用方法的实参是子类对象
- 实例变量声明父类类型，实际存储的是子类对象

```
package com.atguigu.polymorphism.grammar;

public class OnePersonOnePet {
    private Pet pet;
    public void adopt(Pet pet) { //形参是父类类型，实参是子类对象
        this.pet = pet;
    }
    public void feed(){
        pet.eat(); //pet实际引用的对象类型不同，执行的eat方法也不同
    }
}
```

```
package com.atguigu.polymorphism.grammar;

public class TestOnePersonOnePet {
    public static void main(String[] args) {
        OnePersonOnePet person = new OnePersonOnePet();

        Dog dog = new Dog();
        dog.setNickname("小白");
        person.adopt(dog); //实参是dog子类对象，形参是父类Pet类型
        person.feed();

        Cat cat = new Cat();
        cat.setNickname("雪球");
        person.adopt(cat); //实参是cat子类对象，形参是父类Pet类型
        person.feed();
    }
}
```

2、数组元素是父类类型，元素对象是子类对象

```
package com.atguigu.polymorphism.grammar;

public class OnePersonManyPets {
    private Pet[] pets; //数组元素类型是父类类型，元素存储的是子类对象

    public void adopt(Pet[] pets) {
        this.pets = pets;
    }

    public void feed() {
        for (int i = 0; i < pets.length; i++) {
            pets[i].eat(); //pets[i]实际引用的对象类型不同，执行的eat方法也不同
        }
    }
}
```

```
package com.atguigu.polymorphism.grammar;

public class TestPets {
```

```

public static void main(String[] args) {
    Pet[] pets = new Pet[2];
    pets[0] = new Dog(); //多态引用
    pets[0].setNickname("小白");
    pets[1] = new Cat(); //多态引用
    pets[1].setNickname("雪球");

    OnePersonManyPets person = new OnePersonManyPets();
    person.adopt(pets);
    person.feed();
}
}

```

3、方法返回值类型声明为父类类型，实际返回的是子类对象

```

package com.atguigu.polymorphism.grammar;

public class PetShop {
    //返回值类型是父类类型，实际返回的是子类对象
    public Pet sale(String type){
        switch (type){
            case "Dog":
                return new Dog();
            case "Cat":
                return new Cat();
        }
        return null;
    }
}

```

```

package com.atguigu.polymorphism.grammar;

public class TestPetShop {
    public static void main(String[] args) {
        PetShop shop = new PetShop();

        Pet dog = shop.sale("Dog");
        dog.setNickname("小白");
        dog.eat();

        Pet cat = shop.sale("Cat");
        cat.setNickname("雪球");
        cat.eat();
    }
}

```

6.3.4 向上转型与向下转型

首先，一个对象在new的时候创建是哪个类型的对象，它从头至尾都不会变。即这个对象的运行时类型，本质的类型用于不会变。但是，把这个对象赋值给不同类型的变量时，这些变量的编译时类型却不同。

这个和基本数据类型的转换是不同的。基本数据类型是把数据值copy了一份，相当于有两种数据类型的值。而对象的赋值不会产生两个对象。

1、为什么要类型转换呢？

因为多态，就一定会有把子类对象赋值给父类变量的时候，这个时候，在编译期间，就会出现类型转换的现象。

但是，使用父类变量接收了子类对象之后，我们就不能调用子类拥有，而父类没有的方法了。这也是多态给我们带来的一点“小麻烦”。所以，想要调用子类特有的方法，必须做类型转换，使得编译通过。

- **向上转型：**当左边的变量的类型（父类）>右边对象/变量的类型（子类），我们就称为向上转型
 - 此时，编译时按照左边变量的类型处理，就只能调用父类中有的变量和方法，不能调用子类特有的变量和方法了
 - 但是，运行时，仍然是对象本身的类型，所以执行的方法是子类重写的方法体。
 - 此时，一定是安全的，而且也是自动完成的
- **向下转型：**当左边的变量的类型（子类）<右边对象/变量的编译时类型（父类），我们就称为向下转型
 - 此时，编译时按照左边变量的类型处理，就可以调用子类特有的变量和方法了
 - 但是，运行时，仍然是对象本身的类型
 - 不是所有通过编译的向下转型都是正确的，可能会发生ClassCastException，为了安全，可以通过isinstance关键字进行判断

2、如何向上转型与向下转型

向上转型：自动完成

向下转型：（子类类型）父类变量

```
package com.atguigu.polymorphism.grammar;

public class ClassCastTest {
    public static void main(String[] args) {
        //没有类型转换
        Dog dog = new Dog(); //dog的编译时类型和运行时类型都是Dog

        //向上转型
        Pet pet = new Dog(); //pet的编译时类型是Pet，运行时类型是Dog
        pet.setNickname("小白");
        pet.eat(); //可以调用父类Pet有声明的方法eat，但执行的是子类重写的eat方法体
        //        pet.watchHouse(); //不能调用父类没有的方法watchHouse

        Dog d = (Dog) pet;
        System.out.println("d.nickname = " + d.getNickname());
        d.eat(); //可以调用eat方法
        d.watchHouse(); //可以调用子类扩展的方法watchHouse

        Cat c = (Cat) pet; //编译通过，因为从语法检查来说，pet的编译时类型是Pet，Cat是Pet的子类，所以向下转型语法正确
        //这句代码运行报错ClassCastException，因为pet变量的运行时类型是Dog，Dog和Cat之间是没有继承关系的
    }
}
```

3、 instanceof关键字

为了避免ClassCastException的发生，Java提供了 `instanceof` 关键字，给引用变量做类型的校验，只要用 `instanceof` 判断返回true的，那么强转为该类型就一定是安全的，不会报ClassCastException异常。

变量/匿名对象 `instanceof` 数据类型

那么，哪些`instanceof`判断会返回true呢？

- 变量/匿名对象的编译时类型与 `instanceof` 后面数据类型是直系亲属关系才可以比较
- 变量/匿名对象的运行时类型<= `instanceof` 后面数据类型，才为true

示例代码：

```
package com.atguigu.polymorphism.grammar;

public class TestInstanceof {
    public static void main(String[] args) {
        Pet[] pets = new Pet[2];
        pets[0] = new Dog(); //多态引用
        pets[0].setNickname("小白");
        pets[1] = new Cat(); //多态引用
        pets[1].setNickname("雪球");

        for (int i = 0; i < pets.length; i++) {
            pets[i].eat();

            if(pets[i] instanceof Dog){
                Dog dog = (Dog) pets[i];
                dog.watchHouse();
            }else if(pets[i] instanceof Cat){
                Cat cat = (Cat) pets[i];
                cat.catchMouse();
            }
        }
    }
}
```

6.3.5 虚方法

在Java中虚方法是指在编译阶段和类加载阶段都不能确定方法的调用入口地址，在运行阶段才能确定的方法，即可能被重写的方法。

当我们通过“对象xx.方法”的形式调用一个虚方法时，要如何确定它具体执行哪个方法呢？

(1) 静态分派：先看这个对象xx的编译时类型，在这个对象的编译时类型中找到能匹配的方法

匹配的原则：看实参的编译时类型与方法形参的类型的匹配程度

A: 找最匹配 实参的编译时类型 = 方法形参的类型

B: 找兼容 实参的编译时类型 < 方法形参的类型

(2) 动态绑定：再看这个对象xx的运行时类型，如果这个对象xx的运行时类重写了刚刚找到的那个匹配的方法，那么执行重写的，否则仍然执行刚才编译时类型中的那个匹配的方法

```
class MyClass{
    public void method(Father f) {
        System.out.println("father");
    }
    public void method(Son s) {
        System.out.println("son");
    }
}
class MySub extends MyClass{
    public void method(Father d) {
        System.out.println("sub--father");
    }
    public void method(Daughter d) {
        System.out.println("daughter");
    }
}
class Father{
}
class Son extends Father{
}
class Daughter extends Father{
}
```

```
public class TestVirtualMethod {
    public static void main(String[] args) {
        MyClass my = new MySub();
        Father f = new Father();
        Son s = new Son();
        Daughter d = new Daughter();
        my.method(f); //sub--
    }
}
```

(1) 静态分派：看my的编译时类型MyClass，在MyClass中找最匹配的
匹配的原则：看实参的编译时类型与方法形参的类型的匹配程度

- A: 找最匹配 实参的编译时类型 = 方法形参的类型
 - B: 找兼容 实参的编译时类型 < 方法形参的类型
- 实参f的编译时类型是Father，形参(Father f)、(Son s)
最匹配的是public void method(Father f)

(2) 动态绑定：看my的运行时类型MySub，看在MySub中是否有对 public void method(Father f)进行重写

发现有重写，如果有重写，就执行重写的

```
public void method(Father d) {
    System.out.println("sub--");
}
*/
my.method(s); //son
/*
```

(1) 静态分派：看my的编译时类型MyClass，在MyClass中找最匹配的
 匹配的原则：看实参的编译时类型与方法形参的类型的匹配程度
 A：找最匹配 实参的编译时类型 = 方法形参的类型
 B：找兼容 实参的编译时类型 < 方法形参的类型
 实参s的编译时类型是Son，形参(Father f)、(Son s)
 最匹配的是public void method(Son s)

(2) 动态绑定：看my的运行时类型MySub，看在MySub中是否有对 public void method(Son s)进行重写

发现没有重写，如果没有重写，就执行刚刚父类中找到的方法

```
/*
my.method(d); //sub--
*/
(1) 静态分派：看my的编译时类型MyClass，在MyClass中找最匹配的
匹配的原则：看实参的编译时类型与方法形参的类型的匹配程度
A：找最匹配 实参的编译时类型 = 方法形参的类型
B：找兼容 实参的编译时类型 < 方法形参的类型
实参d的编译时类型是Daughter，形参(Father f)、(Son s)
最匹配的是public void method(Father f)
(2) 动态绑定：看my的运行时类型MySub，看在MySub中是否有对 public void method(Father f)进行重写
发现有重写，如果有重写，就执行重写的
public void method(Father d) {
    System.out.println("sub--");
}
*/
}
```

6.3.6 成员变量没有多态一说

```
package com.atguigu.polymorphism.grammar;

public class TestVariable {
    public static void main(String[] args) {
        Base b = new Sub();
        System.out.println(b.a);
        System.out.println(((Sub)b).a);

        Sub s = new Sub();
        System.out.println(s.a);
        System.out.println(((Base)s).a);
    }
}

class Base{
    int a = 1;
}

class Sub extends Base{
    int a = 2;
}
```

6.4 实例初始化

6.4.1 构造器

我们发现我们new完对象时，所有成员变量都是默认值，如果我们需要赋别的值，需要挨个为它们再赋值，太麻烦了。我们能不能在new对象时，直接为当前对象的某个或所有成员变量直接赋值呢。

可以，Java给我们提供了构造器（Constructor）。

1、构造器的作用

new对象，并在new对象的时候为实例变量赋值。

2、构造器的语法格式

构造器又称为构造方法，那是因为它长的很像方法。但是和方法还是有所区别的。

```
【修饰符】 class 类名{
    【修饰符】 构造器名() {
        // 实例初始化代码
    }
    【修饰符】 构造器名(参数列表) {
        // 实例初始化代码
    }
}
```

代码如下：

```
package com.atguigu.constructor;

public class Student {
    private String name;
    private int age;

    // 无参构造
    public Student() {}

    // 有参构造
    public Student(String name, int age) {
        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 String getInfo(){
    return "姓名: " + name + ", 年龄: " + age;
}
}
```

注意事项：

1. 构造器名必须与它所在的类名必须相同。
2. 它没有返回值，所以不需要返回值类型，甚至不需要void
3. 如果你不提供构造器，系统会给出无参数构造器，并且该构造器的修饰符默认与类的修饰符相同
4. 如果你提供了构造器，系统将不再提供无参数构造器，除非你自己定义。
5. 构造器是可以重载的，既可以定义参数，也可以不定义参数。
6. 构造器的修饰符只能是权限修饰符，不能被其他任何修饰

```
package com.atguigu.constructor;

public class TestStudent {
    public static void main(String[] args) {
        //调用无参构造创建学生对象
        Student s1 = new Student();

        //调用有参构造创建学生对象
        Student s2 = new Student("张三",23);

        System.out.println(s1.getInfo());
        System.out.println(s2.getInfo());
    }
}
```

3、同一个类中的构造器互相调用

- this(): 调用本类的无参构造
- this(实参列表): 调用本类的有参构造
- this()和this(实参列表)只能出现在构造器首行
- 不能出现递归调用

```
package com.atguigu.constructor;

public class Student {
    private String name;
    private int age;

    // 无参构造
    public Student() {
//        this("",18); //调用本类有参构造
    }

    // 有参构造
    public Student(String name,int age) {
        this(); //调用本类无参构造
    }
}
```

```

        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 String getInfo(){
        return "姓名: " + name +", 年龄: " + age;
    }
}

```

4、继承时构造器如何处理

- 子类继承父类时，不会继承父类的构造器。只能通过super()或super(实参列表)的方式调用父类的构造器。
- super();: 子类构造器中一定会调用父类的构造器，默认调用父类的无参构造，super();可以省略。
- super(实参列表);: 如果父类没有无参构造或者有无参构造但是子类就是想要调用父类的有参构造，则必须使用super(实参列表);的语句。
- super()和super(实参列表)都只能出现在子类构造器的首行

```

package com.atguigu.constructor;

public class Employee {
    private String name;
    private int age;
    private double salary;

    public Employee() {
        System.out.println("父类Employee无参构造");
    }

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
        System.out.println("父类Employee有参构造");
    }

    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 double getSalary() {
    return salary;
}

public void setSalary(double salary) {
    this.salary = salary;
}

public String getInfo(){
    return "姓名: " + name + ", 年龄: " + age +", 薪资: " + salary;
}
}
```

```
package com.atguigu.constructor;

public class Manager extends Employee{
    private double bonusRate;

    public Manager() {
        super(); //可以省略
    }

    public Manager(String name, int age, double salary, double bonusRate) {
        super(name, age, salary); //调用父类的有参构造
        this.bonusRate = bonusRate;
    }

    public double getBonusRate() {
        return bonusRate;
    }

    public void setBonusRate(double bonusRate) {
        this.bonusRate = bonusRate;
    }

    @Override
    public String getInfo() {
        return super.getInfo() +", 奖金比例: " + bonusRate;
    }
}
```

```
}
```

```
package com.atguigu.constructor;

public class TestEmployee {
    public static void main(String[] args) {
        Manager m1 = new Manager();
        System.out.println(m1.getInfo());

        Manager m2 = new Manager("张三", 23, 20000, 0.1);
        System.out.println(m2.getInfo());
    }
}
```

形式一：

```
class A{
}

class B extends A{

}

class Test{
    public static void main(String[] args){
        B b = new B();
        //A类和B类都是默认有一个无参构造，B类的默认无参构造中还会默认调用A类的默认无参构造
        //但是因为都是默认的，没有打印语句，看不出来
    }
}
```

形式二：

```
class A{
    A(){
        System.out.println("A类无参构造器");
    }
}
class B extends A{

}
class Test{
    public static void main(String[] args){
        B b = new B();
        //A类显示声明一个无参构造,
        //B类默认有一个无参构造,
        //B类的默认无参构造中会默认调用A类的无参构造
        //可以看到会输出“A类无参构造器”
    }
}
```

```
    }  
}
```

形式三：

```
class A{  
    A(){  
        System.out.println("A类无参构造器");  
    }  
}  
class B extends A{  
    B(){  
        System.out.println("B类无参构造器");  
    }  
}  
class Test{  
    public static void main(String[] args){  
        B b = new B();  
        //A类显示声明一个无参构造，  
        //B类显示声明一个无参构造，  
        //B类的无参构造中虽然没有写super()，但是仍然会默认调用A类的无参构造  
        //可以看到会输出“A类无参构造器”和“B类无参构造器”  
    }  
}
```

形式四：

```
class A{  
    A(){  
        System.out.println("A类无参构造器");  
    }  
}  
class B extends A{  
    B(){  
        super();  
        System.out.println("B类无参构造器");  
    }  
}  
class Test{  
    public static void main(String[] args){  
        B b = new B();  
        //A类显示声明一个无参构造，  
        //B类显示声明一个无参构造，  
        //B类的无参构造中明确写了super()，表示调用A类的无参构造  
        //可以看到会输出“A类无参构造器”和“B类无参构造器”  
    }  
}
```

形式五：

```

class A{
    A(int a){
        System.out.println("A类有参构造器");
    }
}
class B extends A{
    B(){
        System.out.println("B类无参构造器");
    }
}
class Test05{
    public static void main(String[] args){
        B b = new B();
        //A类显示声明一个有参构造，没有写无参构造，那么A类就没有无参构造了
        //B类显示声明一个无参构造，
        //B类的无参构造没有写super(...), 表示默认调用A类的无参构造
        //编译报错，因为A类没有无参构造
    }
}

```

```

class A{
    A(int a){
        System.out.println("A类有参构造器");
    }
}
class B extends A{
    /* B( ) { Implicit super constructor A() is undefined for default constructor. Must define an explicit
    // constructor
    // 1 quick fix available:
    }* Add constructor 'B(int)'
}
class Test05{
    public static void main(String[] args){
        B b = new B();
        //A类显示声明一个有参构造，没有写无参构造，那么A类就没有无参构造了
        //B类显示声明一个无参构造，
        //B类的无参构造没有写super(...), 表示默认调用A类的无参构造
        //编译报错，因为A类没有无参构造
    }
}

```

如果此时B类不写构造器，那么B类会自动产生无参构造，
默认也需要调用父类的无参构造，父类没有，报错

```

class A{
    A(int a){
        System.out.println("A类有参构造器");
    }
}
class B extends A{
    B(){
        // Implicit super constructor A() is undefined. Must explicitly invoke another constructor
        // A无参构造没有定义
    }
}
class Test05{
    public static void main(String[] args){
        B b = new B();
        //A类显示声明一个有参构造，没有写无参构造，那么A类就没有无参构造了
        //B类显示声明一个无参构造，
        //B类的无参构造没有写super(...), 表示默认调用A类的无参构造
        //编译报错，因为A类没有无参构造
    }
}

```

形式六：

```

class A{
    A(int a){
        System.out.println("A类有参构造器");
    }
}
class B extends A{
    B(){
        super();
        System.out.println("B类无参构造器");
    }
}
class Test06{
    public static void main(String[] args){
        B b = new B();
        //A类显示声明一个有参构造，没有写无参构造，那么A类就没有无参构造了
        //B类显示声明一个无参构造，
        //B类的无参构造明确写super(), 表示调用A类的无参构造
        //编译报错，因为A类没有无参构造
    }
}

```

```

7 class A{
8     A(int a){
9         System.out.println("A类有参构造器");
10    }
11 }
12 class B extends A{
13     B(){
14         super();
15         System.out.println("B类无参构造器");
16     }
17 }
18 class Test06{
19     public static void main(String[] args){
20         B b = new B();
21         //A类显示声明一个有参构造，没有写无参构造，那么A类就没有无参构造了
22         //B类显示声明一个无参构造，
23         //B类的无参构造明确写super()，表示调用A类的无参构造
24         //编译报错，因为A类没有无参构造
25     }
26 }

```

形式七：

```

class A{
    A(int a){
        System.out.println("A类有参构造器");
    }
}
class B extends A{
    B(int a){
        super(a);
        System.out.println("B类有参构造器");
    }
}
class Test07{
    public static void main(String[] args){
        B b = new B(10);
        //A类显示声明一个有参构造，没有写无参构造，那么A类就没有无参构造了
        //B类显示声明一个有参构造，
        //B类的有参构造明确写super(a)，表示调用A类的有参构造
        //会打印“A类有参构造器”和“B类有参构造器”
    }
}

```

形式八：

```

class A{
    A(){
        System.out.println("A类无参构造器");
    }
}

```

```

    }

    A(int a){
        System.out.println("A类有参构造器");
    }
}

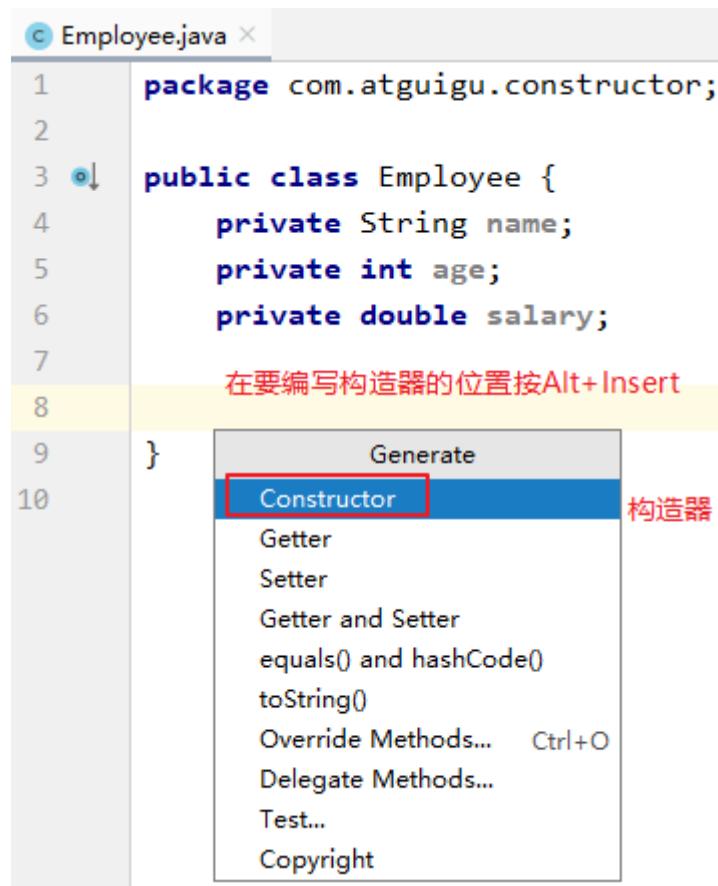
class B extends A{
    B () {
        super(); //可以省略，调用父类的无参构造
        System.out.println("B类无参构造器");
    }

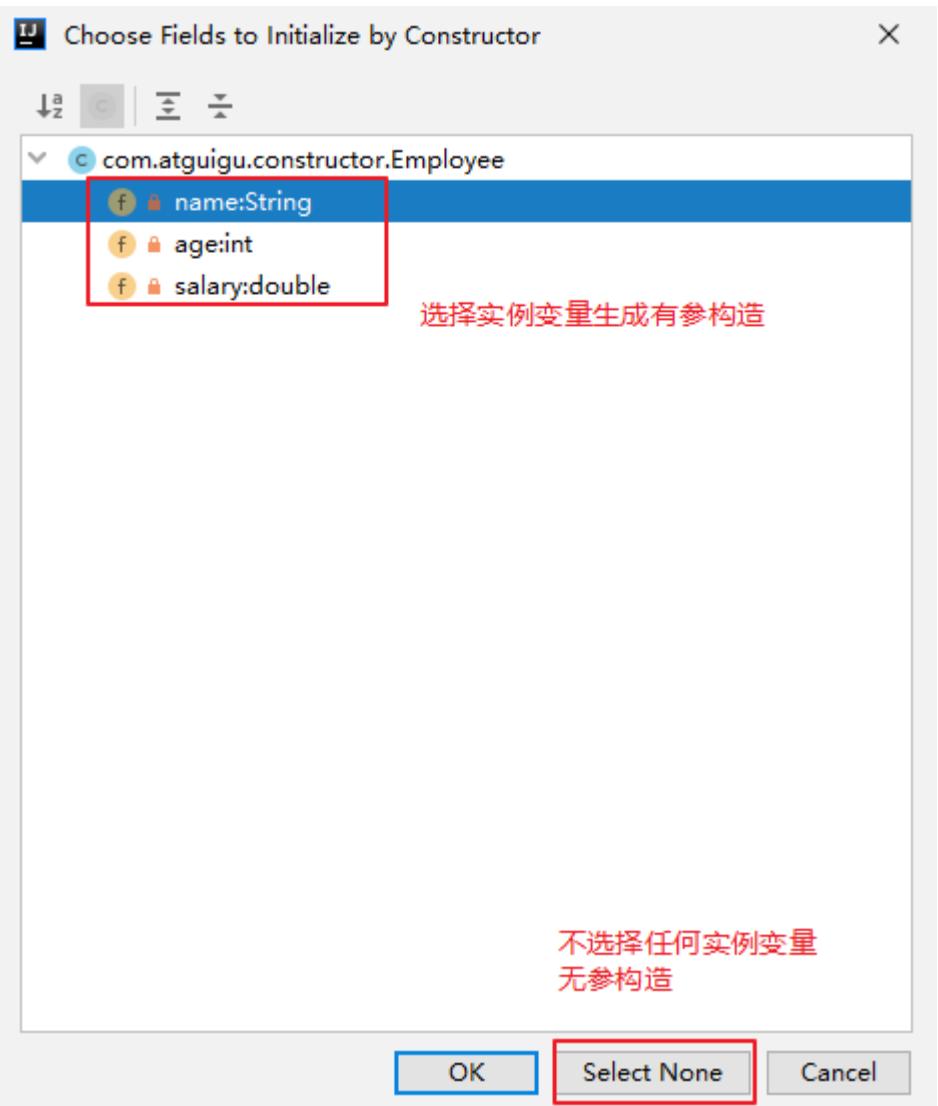
    B(int a){
        super(a); //调用父类有参构造
        System.out.println("B类有参构造器");
    }
}

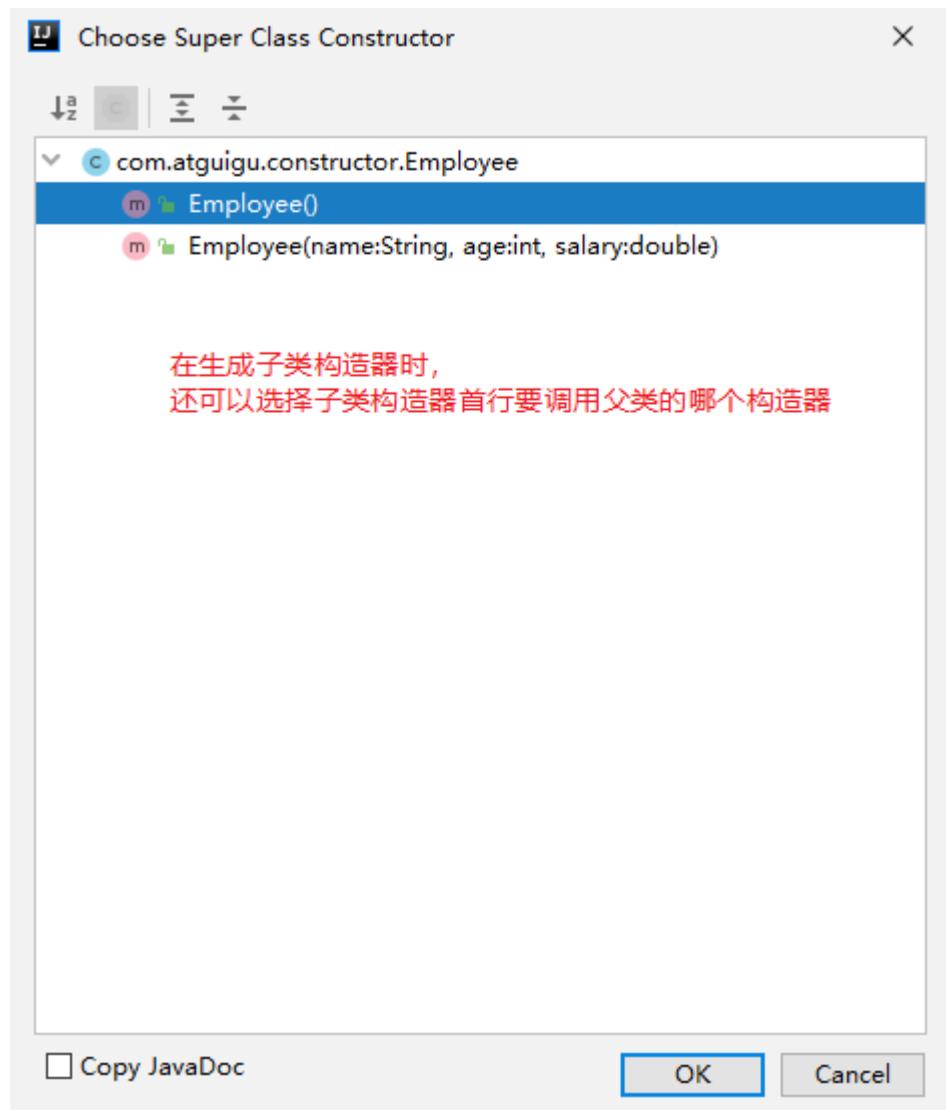
class Test8{
    public static void main(String[] args){
        B b1 = new B();
        B b2 = new B(10);
    }
}

```

5、IDEA生成构造器：Alt + Insert







6、IDEA查看构造器和方法形参列表快捷键：Ctrl + P

```
public class TestPeople {  
    public static void main(String[] args) {  
        Person p = new Person( name: "张三", age: 23, gender: '男');  
        System.out.println(p.getInfo());  
  
        Student s = new Student( name: "李四", age: 20, gender: '女', score: 90);  
        System.out.println(s.getInfo());  
  
        Teacher t = new Teacher( name: "王五", age: 30, gender: '男', salary: 5000);  
        System.out.println(t.getInfo());  
    }  
}
```

在构造器或方法的()中按Ctrl + P, IDEA就会提示
构造器或方法的形参列表

6.4.2 非静态代码块（了解）

1、非静态代码块的作用

和构造器一样，也是用于实例变量的初始化等操作。

2、非静态代码块的意义

如果多个重载的构造器有公共代码，并且这些代码都是先于构造器其他代码执行的，那么可以将这部分代码抽取到非静态代码块中，减少冗余代码。

3、非静态代码块的执行特点

所有非静态代码块中代码都是在new对象时自动执行，并且一定是先于构造器的代码执行。

4、非静态代码块的语法格式

```
【修饰符】 class 类{
    {
        非静态代码块
    }
    【修饰符】 构造器名() {
        // 实例初始化代码
    }
    【修饰符】 构造器名(参数列表) {
        // 实例初始化代码
    }
}
```

5、非静态代码块的应用

案例：

(1) 声明User类，

- 包含属性：username (String类型)， password (String类型)， registrationTime (long类型)， 私有化
- 包含get/set方法，其中registrationTime没有set方法
- 包含无参构造，
 - 输出“新用户注册”，
 - registrationTime赋值为当前系统时间，
 - username就默认为当前系统时间值，
 - password默认为“123456”
- 包含有参构造(String username, String password)，
 - 输出“新用户注册”，
 - registrationTime赋值为当前系统时间，
 - username和password由参数赋值
- 包含public String getInfo()方法，返回：“用户名：xx，密码：xx，注册时间：xx”

(2) 编写测试类，测试类main方法的代码如下：

```
public static void main(String[] args) {  
    User u1 = new User();  
    System.out.println(u1.getInfo());  
  
    User u2 = new User("chai","8888");  
    System.out.println(u2.getInfo());  
}
```

如果不使用静态代码块，User类是这样的：

```
package com.atguigu.block.no;  
  
public class User {  
    private String username;  
    private String password;  
    private long registrationTime;  
  
    public User() {  
        System.out.println("新用户注册");  
        registrationTime = System.currentTimeMillis();  
        username = registrationTime+"";  
        password = "123456";  
    }  
  
    public User(String username,String password) {  
        System.out.println("新用户注册");  
        registrationTime = System.currentTimeMillis();  
        this.username = username;  
        this.password = password;  
    }  
  
    public String getUsername() {  
        return username;  
    }  
  
    public void setUsername(String username) {  
        this.username = username;  
    }  
  
    public String getPassword() {  
        return password;  
    }  
  
    public void setPassword(String password) {  
        this.password = password;  
    }  
  
    public long getRegistrationTime() {  
        return registrationTime;  
    }  
    public String getInfo(){  
        return "用户名：" + username + ", 密码：" + password + ", 注册时间：" + registrationTime;
```

```
    }
}
```

如果提取构造器公共代码到非静态代码块，User类是这样的：

```
package com.atguigu.block.use;

public class User {
    private String username;
    private String password;
    private long registrationTime;

    {
        System.out.println("新用户注册");
        registrationTime = System.currentTimeMillis();
    }

    public User() {
        username = registrationTime + "";
        password = "123456";
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public long getRegistrationTime() {
        return registrationTime;
    }

    public String getInfo(){
        return "用户名：" + username + ", 密码：" + password + ", 注册时间：" + registrationTime;
    }
}
```

6.4.3 实例初始化过程（了解）

1、实例初始化的目的

实例初始化的过程其实就是在new对象的过程中为实例变量赋有效初始值的过程

2、实例初始化相关代码

在new对象的过程中给实例变量赋初始值可以通过以下3个部分的代码完成：

- (1) 实例变量直接初始化
- (2) 非静态代码块
- (3) 构造器

当然，如果没有编写上面3个部分的任何代码，那么实例变量也有默认值。

3、实例初始化方法

实际上我们编写的代码在编译时，会自动处理代码，整理出一个或多个的(...)实例初始化方法。一个类有几个实例初始化方法，由这个类就有几个构造器决定。

实例初始化方法的方法体，由4部分构成：

- (1) super()或super(实参列表)
 - 这里选择哪个，看原来构造器首行是super()还是super(实参列表)
 - 如果原来构造器首行是this()或this(实参列表)，那么就取对应构造器首行的super()或super(实参列表)
 - 如果原来构造器首行既没写this()或this(实参列表)，也没写super()或super(实参列表)，默认就是super()
- (2) 非静态实例变量的显示赋值语句
- (3) 非静态代码块
- (4) 对应构造器中剩下的的代码

特别说明：其中 (2) 和 (3) 是按顺序合并的，(1) 一定在最前面 (4) 一定在最后面

4、实例初始化执行特点

- 创建对象时，才会执行
- 每new一个对象，都会完成该对象的实例初始化
- 调用哪个构造器，就是执行它对应的实例初始化方法
- 子类super()还是super(实参列表)实例初始化方法中的super()或super(实参列表)不仅仅代表父类的构造器代码了，而是代表父类构造器对应的实例初始化方法。

5、演示父类实例初始化

```
package com.atguigu.init;

public class Father {
    private int a = 1;

    public Father(){
        System.out.println("Father类的无参构造");
```

```

}
public Father(int a, int b){
    System.out.println("Father类的有参构造");
    this.a = a;
    this.b = b;
}
{
    System.out.println("Father类的非静态代码块1, a = " + a);
    System.out.println("Father类的非静态代码块1, b = " + this.b);
}
private int b = 1;
{
    System.out.println("Father类的非静态代码块2, a = " + a);
    System.out.println("Father类的非静态代码块2, b = " + b);
}

public String getInfo(){
    return "a = " + a + ", b = " + b;
}
}

```

```

package com.atguigu.init;

public class TestFather {
    public static void main(String[] args) {
        Father f1 = new Father();
        System.out.println(f1.getInfo());
        System.out.println("-----");
        Father f2 = new Father(10,10);
        System.out.println(f2.getInfo());
    }
}

```

```

public class Father {
    private int a = 1;

    public Father(){
        System.out.println("Father类的无参构造");
    }

    public Father(int a, int b){
        System.out.println("Father类的有参构造");
        this.a = a;
        this.b = b;
    }

    {
        System.out.println("Father类的非静态代码块1, a = " + a);
        System.out.println("Father类的非静态代码块1, b = " + this.b);
    }

    private int b = 1;

    {
        System.out.println("Father类的非静态代码块2, a = " + a);
        System.out.println("Father类的非静态代码块2, b = " + b);
    }

    public String getInfo(){
        return "a = " + a + ", b = " + b;
    }
}

```

编译后
代码重组



```

public class Father {
    private int a;//默认值0
    private int b;//默认值0

    <init>(){
        super(); //Father的默认父类Object, 存在, 但看不到打印结果而已
        a = 1;
        System.out.println("Father类的非静态代码块1, a = " + a);
        System.out.println("Father类的非静态代码块1, b = " + this.b);
        b = 1;
        System.out.println("Father类的非静态代码块2, a = " + a);
        System.out.println("Father类的非静态代码块2, b = " + b);

        System.out.println("Father类的无参构造");
    }

    <init>(int a, int b){
        super(); //Father的默认父类Object, 存在, 但看不到打印结果而已
        a = 1;
        System.out.println("Father类的非静态代码块1, a = " + a);
        System.out.println("Father类的非静态代码块1, b = " + this.b);
        b = 1;
        System.out.println("Father类的非静态代码块2, a = " + a);
        System.out.println("Father类的非静态代码块2, b = " + b);

        System.out.println("Father类的有参构造");
        this.a = a;
        this.b = b;
    }

    public String getInfo(){
        return "a = " + a + ", b = " + b;
    }
}

```

```

public static void main(String[] args) {
    Father f1 = new Father();
    System.out.println(f1.getInfo());
    System.out.println("-----");
    Father f2 = new Father( a: 10, b: 10 );
    System.out.println(f2.getInfo());
}

Father类的非静态代码块1, a = 1
Father类的非静态代码块1, b = 0
Father类的非静态代码块2, a = 1
Father类的非静态代码块2, b = 1
Father类的无参构造
a = 1, b = 1
-----
Father类的非静态代码块1, a = 1
Father类的非静态代码块1, b = 0
Father类的非静态代码块2, a = 1
Father类的非静态代码块2, b = 1
Father类的有参构造
a = 10, b = 10

```

```

public class Father {
    private int a;//默认值0
    private int b;//默认值0

    <init>(){
        super(); //Father的默认父类Object, 存在, 但看不到打印结果而已
        a = 1;
        System.out.println("Father类的非静态代码块1, a = " + a);
        System.out.println("Father类的非静态代码块1, b = " + this.b);
        b = 1;
        System.out.println("Father类的非静态代码块2, a = " + a);
        System.out.println("Father类的非静态代码块2, b = " + b);

        System.out.println("Father类的无参构造");
    }

    <init>(int a, int b){
        super(); //Father的默认父类Object, 存在, 但看不到打印结果而已
        a = 1;
        System.out.println("Father类的非静态代码块1, a = " + a);
        System.out.println("Father类的非静态代码块1, b = " + this.b);
        b = 1;
        System.out.println("Father类的非静态代码块2, a = " + a);
        System.out.println("Father类的非静态代码块2, b = " + b);

        System.out.println("Father类的有参构造");
        this.a = a;
        this.b = b;
    }

    public String getInfo(){
        return "a = " + a + ", b = " + b;
    }
}

```

6、演示子类实例初始化

```

package com.atguigu.init;

public class Son extends Father {
    private int c = 1;
    {
        System.out.println("Son类的非静态代码块,c = " + c);
    }
    public Son() {
        System.out.println("Son类的无参构造");
    }

    public Son(int a, int b, int c) {
        super(a, b);
        this.c = c;
        System.out.println("Son类的有参构造");
    }

    @Override
    public String getInfo() {
        return super.getInfo() + ",c = " + c;
    }
}

```

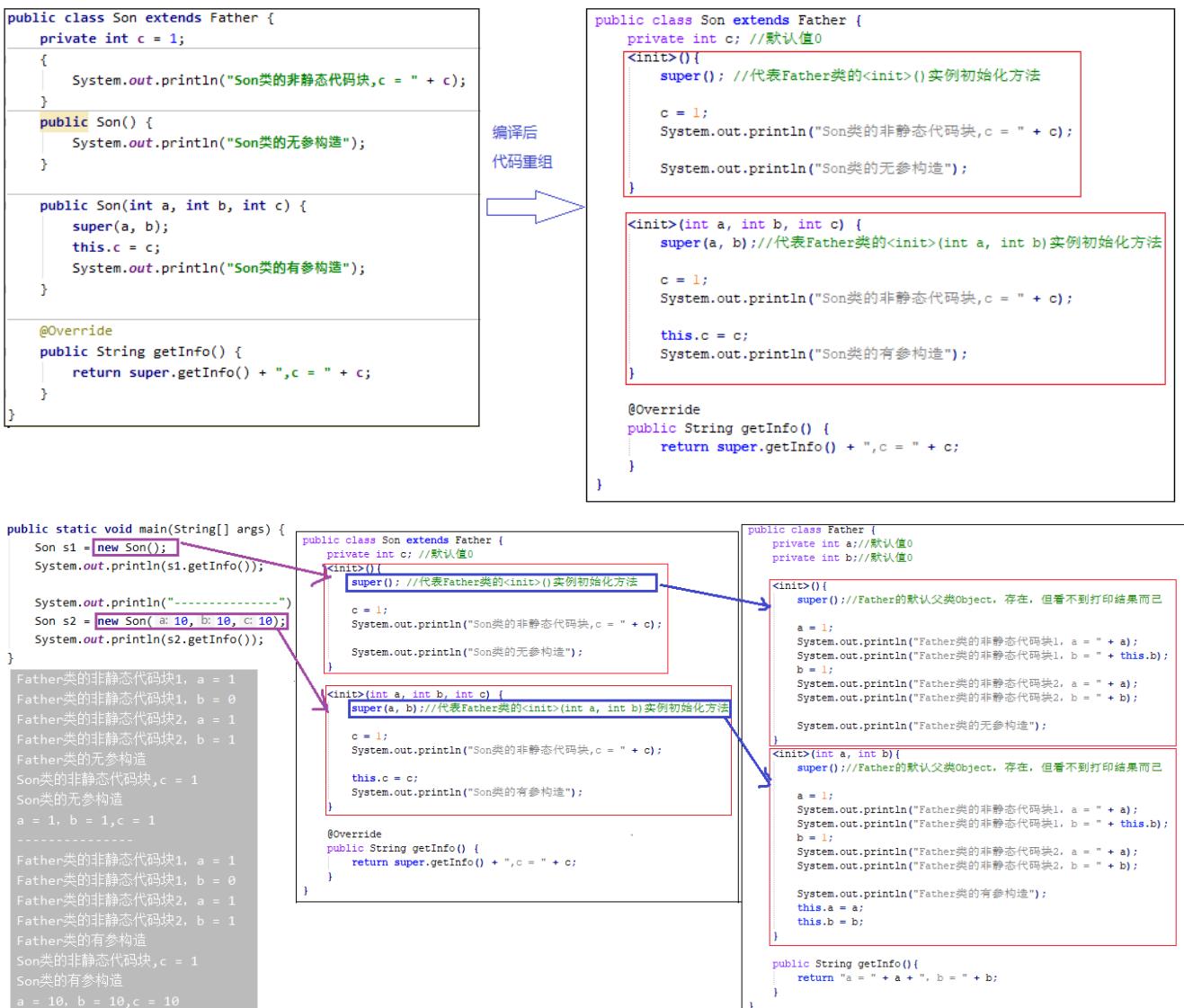
```

package com.atguigu.init;

public class TestSon {
    public static void main(String[] args) {
        Son s1 = new Son();
        System.out.println(s1.getInfo());

        System.out.println("-----");
        Son s2 = new Son(10,10,10);
        System.out.println(s2.getInfo());
    }
}

```



6.5 关键字和API

6.5.1 this和super关键字

1.this和super的意义

this: 当前对象

- 在构造器和非静态代码块中，表示正在new的对象
- 在实例方法中，表示调用当前方法的对象

super: 引用父类声明的成员

无论是this和super都是和对象有关的。

2.this和super的使用格式

- this
 - this.成员变量：表示当前对象的某个成员变量，而不是局部变量
 - this.成员方法：表示当前对象的某个成员方法，完全可以省略this.
 - this()或this(实参列表)：调用另一个构造器协助当前对象的实例化，只能在构造器首行，只会找本类的构造器，找不到就报错
- super
 - super.成员变量：表示当前对象的某个成员变量，该成员变量在父类中声明的
 - super.成员方法：表示当前对象的某个成员方法，该成员方法在父类中声明的
 - super()或super(实参列表)：调用父类的构造器协助当前对象的实例化，只能在构造器首行，只会找直接父类的对应构造器，找不到就报错

3.避免子类和父类声明重名的成员变量

特别说明：应该避免子类声明和父类重名的成员变量

因为，子类会继承父类所有的成员变量，所以：

- 如果重名的成员变量表示相同的意义，就无需重复声明
- 如果重名的成员变量表示不同的意义，会引起歧义

在阿里的开发规范等文档中都做出明确说明：

The screenshot shows a PDF viewer window with the following details:

- Title Bar:** 《Java开发手册 (泰山版)》.pdf
- Toolbar:** Back, Forward, Refresh, File (文件), E:/work/book/《Java开发手册 (泰山版)》.pdf
- Header:** 目录 5 / 61 搜索
- Content Area:**
 - Section 10:** 10.【强制】避免在子父类的成员变量之间、或者不同代码块的局部变量之间采用完全相同的命名，使可读性降低。
 - Description:** 子类、父类成员变量名相同，即使是 public 类型的变量也是能够通过编译，而局部变量在同一方法内的不同代码块中同名也是合法的，但是要避免使用。对于非 setter/getter 的参数名称也要避免与成员变量名称相同。
 - Example:**

```
public class ConfusingName {
    public int stock;

    // 非 setter/getter 的参数名称，不允许与本类成员变量同名
    public void get(String alibaba) {
        if (condition) {
            final int money = 666;
            // ...
        }

        for (int i = 0; i < 10; i++) {
            // 在同一方法体中，不允许与其它代码块中的 money 命名相同
            final int money = 15978;
            // ...
        }
    }

    class Son extends ConfusingName {
        // 不允许与父类的成员变量名称相同
        public int stock;
    }
}
```

4.解决成员变量重名问题

- 如果实例变量与局部变量重名，可以在实例变量前面加this.进行区别
- 如果子类实例变量和父类实例变量重名，并且父类的该实例变量在子类仍然可见，在子类中要访问父类声明的实例变量需要在父类实例变量前加super.，否则默认访问的是子类自己声明的实例变量
- 如果父子类实例变量没有重名，只要权限修饰符允许，在子类中完全可以直接访问父类中声明的实例变量，也可以用this.实例访问，也可以用super.实例访问

```
class Father{
    int a = 10;
    int b = 11;
}

class Son extends Father{
    int a = 20;

    public void test(){
        //子类与父类的属性同名，子类对象中就有两个a
        System.out.println("子类的a: " + a); //20 先找局部变量找，没有再从本类成员变量找
        System.out.println("子类的a: " + this.a); //20 先从本类成员变量找
        System.out.println("父类的a: " + super.a); //10 直接从父类成员变量找

        //子类与父类的属性不同名，是同一个b
        System.out.println("b = " + b); //11 先找局部变量找，没有再从本类成员变量找，没有再从父类找
        System.out.println("b = " + this.b); //11 先从本类成员变量找，没有再从父类找
        System.out.println("b = " + super.b); //11 直接从父类局部变量找
    }
}
```

```

public void method(int a, int b){
    //子类与父类的属性同名，子类对象中就有两个成员变量a，此时方法中还有一个局部变量a
    System.out.println("局部变量的a: " + a); //30 先找局部变量
    System.out.println("子类的a: " + this.a); //20 先从本类成员变量找
    System.out.println("父类的a: " + super.a); //10 直接从父类成员变量找

    System.out.println("b = " + b); //13 先找局部变量
    System.out.println("b = " + this.b); //11 先从本类成员变量找
    System.out.println("b = " + super.b); //11 直接从父类局部变量找
}
}

class Test{
    public static void main(String[] args){
        Son son = new Son();
        son.test();
        son.method(30,13);
    }
}

```

总结：起点不同（就近原则）

- 变量前面没有super.和this.

- 在构造器、代码块、方法中如果出现使用某个变量，先查看是否是当前块声明的==局部变量==，
- 如果不是局部变量，先从当前执行代码的==本类去找成员变量==
- 如果从当前执行代码的本类中没有找到，会往上找==父类声明的成员变量==（权限修饰符允许在子类中访问的）

- 变量前面有this.

- 通过this找成员变量时，先从当前执行代码的==本类去找成员变量==
- 如果从当前执行代码的本类中没有找到，会往上找==父类声明的成员变量（==权限修饰符允许在子类中访问的）

- 变量前面super.

- 通过super找成员变量，直接从当前执行代码的直接父类去找成员变量（权限修饰符允许在子类中访问的）
- 如果直接父类没有，就去父类的父类中找（权限修饰符允许在子类中访问的）

5.解决成员方法重写后调用问题

- 如果子类没有重写父类的方法，只有权限修饰符运行，在子类中完全可以直接调用父类的方法；
- 如果子类重写了父类的方法，在子类中需要通过super.才能调用父类被重写的方法，否则默认调用的子类重写的方法

```

public class Test{
    public static void main(String[] args){
        Son s = new Son();
        s.test();

        Daughter d = new Daughter();
        d.test();
    }
}

```

```

class Father{
    protected int num = 10;
    public int getNum(){
        return num;
    }
}
class Son extends Father{
    private int num = 20;

    public void test(){
        System.out.println(getNum());//10 本类没有找父类，执行父类中的getNum()
        System.out.println(this.getNum());//10 本类没有找父类，执行父类中的getNum()
        System.out.println(super.getNum());//10 本类没有找父类，执行父类中的getNum()
    }
}
class Daughter extends Father{
    private int num = 20;

    @Override
    public int getNum(){
        return num;
    }

    public void test(){
        System.out.println(getNum());//20 先找本类，执行本类的getNum()
        System.out.println(this.getNum());//20 先找本类，执行本类的getNum()
        System.out.println(super.getNum());//10 直接找父类，执行父类中的getNum()
    }
}

```

总结：

- **方法前面没有super.和this.**
 - 先从子类找匹配方法，如果没有，再从直接父类找，再没有，继续往上追溯
- **方法前面有this.**
 - 先从子类找匹配方法，如果没有，再从直接父类找，再没有，继续往上追溯
- **方法前面有super.**
 - 从当前子类的直接父类找，如果没有，继续往上追溯

6.5.2 native关键字

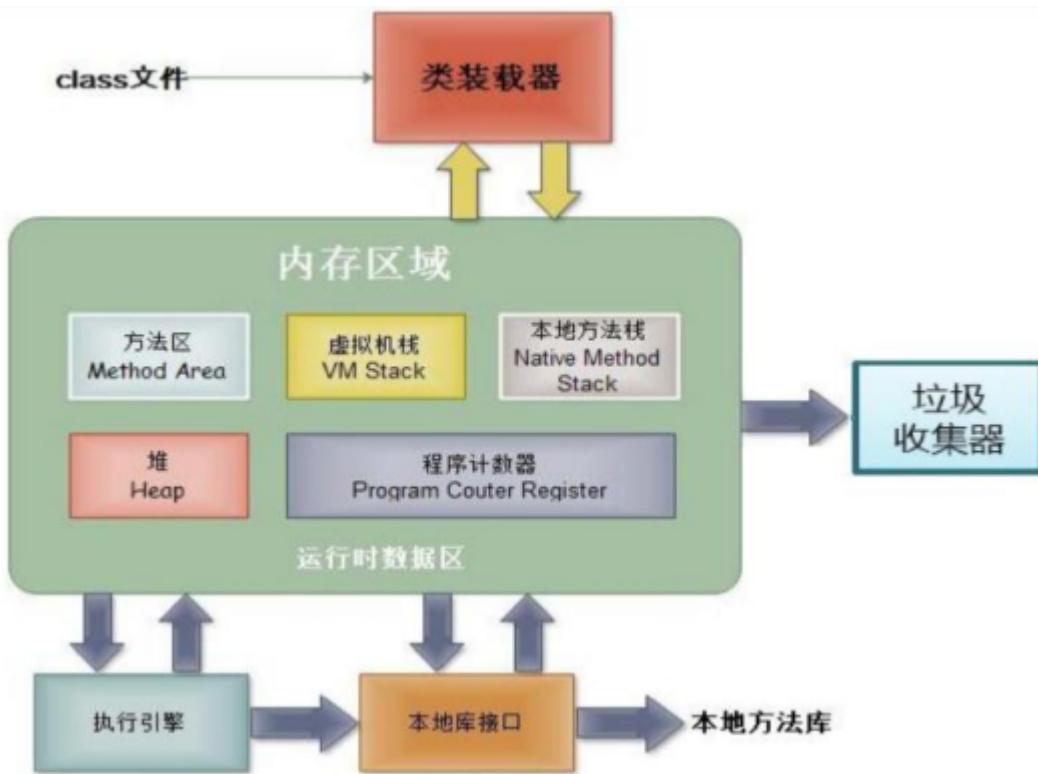
1.native的意义

native：本地的，原生的

2.native的语法

native只能修饰方法，表示这个方法的方法体代码不是用Java语言实现的，而是由C/C++语言编写的。但是对于Java程序员来说，可以当做Java的方法一样去正常调用它，或者子类重写它。

JVM内存的管理：



| 区域名称 | 作用 |
|-------|--|
| 程序计数器 | 程序计数器是CPU中的寄存器，它包含每一个线程下一条要执行的指令的地址 |
| 本地方法栈 | 当程序中调用了native的本地方法时，本地方法执行期间的内存区域 |
| 方法区 | 存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。 |
| 堆内存 | 存储对象（包括数组对象），new来创建的，都存储在堆内存。 |
| 虚拟机栈 | 用于存储正在执行的每个Java方法的局部变量表等。局部变量表存放了编译期可知长度的各种基本数据类型、对象引用，方法执行完，自动释放。 |

6.5.3 final关键字

1.final的意义

final：最终的，不可更改的

2.final修饰类

表示这个类不能被继承，没有子类

```
final class Eunuch{//太监类  
}  
class Son extends Eunuch{//错误  
}
```

3.final修饰方法

表示这个方法不能被子类重写

```
class Father{  
    public final void method(){  
        System.out.println("father");  
    }  
}  
class Son extends Father{  
    public void method(){//错误  
        System.out.println("son");  
    }  
}
```

4.final修饰变量

final修饰某个变量（成员变量或局部变量），表示它的值就不能被修改，即常量，常量名建议使用大写字母。

如果某个成员变量用final修饰后，没有set方法，并且必须初始化（可以显式赋值、或在初始化块赋值、实例变量还可以在构造器中赋值）

```
package com.atguigu.keyword.finals;  
  
public class TestFinal {  
    public static void main(String[] args){  
        final int MIN_SCORE = 0;  
        final int MAX_SCORE = 100;  
  
        MyDate m1 = new MyDate();  
        System.out.println(m1.getInfo());  
  
        MyDate m2 = new MyDate(2022,2,14);  
        System.out.println(m2.getInfo());  
    }  
}  
class MyDate{  
    //没有set方法，必须有显示赋值的代码  
    private final int year;  
    private final int month;  
    private final int day;  
  
    public MyDate(){  
        year = 1970;  
        month = 1;
```

```

        day = 1;
    }

    public MyDate(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public int getYear() {
        return year;
    }

    public int getMonth() {
        return month;
    }

    public int getDay() {
        return day;
    }

    public String getInfo(){
        return year + "年" + month + "月" + day + "日";
    }
}

```

6.5.4 Object根父类

1.如何理解根父类

类 `java.lang.Object` 是类层次结构的根类，即所有类的父类。每个类都使用 `Object` 作为超类。

- Object类型的变量与除Object以外的任意引用数据类型的对象都多态引用
- 所有对象（包括数组）都实现这个类的方法。
- 如果一个类没有特别指定父类，那么默认则继承自Object类。例如：

```

public class MyClass /*extends Object*/ {
    // ...
}

```

2.Object类的其中5个方法

API(Application Programming Interface)，应用程序编程接口。Java API是一本程序员的字典，是JDK中提供给我们使用的类的说明文档。所以我们可以通过查询API的方式，来学习Java提供的类，并得知如何使用它们。在API文档中是无法得知这些类具体是如何实现的，如果要查看具体实现代码，那么我们需要查看src源码。

根据JDK源代码及Object类的API文档，Object类当中包含的方法有11个。今天我们主要学习其中的5个：

(1) `toString()`

方法签名： `public String toString()`

①默认情况下，`toString()`返回的是“对象的运行时类型 @ 对象的hashCode值的十六进制形式”

②通常是建议重写

③如果我们直接System.out.println(对象)，默认会自动调用这个对象的toString()

因为Java的引用数据类型的变量中存储的实际上时对象的内存地址，但是Java对程序员隐藏内存地址信息，所以不能直接将内存地址显示出来，所以当你打印对象时，JVM帮你调用了对象的toString()。

例如自定义的Person类：

```
public class Person {  
    private String name;  
    private int age;  
  
    @Override  
    public String toString() {  
        return "Person{" + "name='" + name + '\'' + ", age=" + age + '}';  
    }  
}
```

(2) getClass()

public final Class<?> getClass(): 获取对象的运行时类型

因为Java有多态现象，所以一个引用数据类型的变量的编译时类型与运行时类型可能不一致，因此如果需要查看这个变量实际指向的对象的类型，需要用getClass()方法

```
public static void main(String[] args) {  
    Object obj = new Person();  
    System.out.println(obj.getClass());//运行时类型  
}
```

(3) equals()

public boolean equals(Object obj): 用于判断当前对象this与指定对象obj是否“相等”

①默认情况下，equals方法的实现等价于与“==”，比较的是对象的地址值

②我们可以选择重写，重写有些要求：

A:

B：如果重写equals，那么一定要遵循如下几个原则：

a：自反性：x.equals(x)返回true

b：传递性：x.equals(y)为true, y.equals(z)为true，然后x.equals(z)也应该为true

c：一致性：只要参与equals比较的属性值没有修改，那么无论何时调用结果应该一致

d：对称性：x.equals(y)与y.equals(x)结果应该一样

e：非空对象与null的equals一定是false

```
class User{  
    private String host;  
    private String username;
```

```
private String password;
public User(String host, String username, String password) {
    super();
    this.host = host;
    this.username = username;
    this.password = password;
}
public User() {
    super();
}
public String getHost() {
    return host;
}
public void setHost(String host) {
    this.host = host;
}
public String getUsername() {
    return username;
}
public void setUsername(String username) {
    this.username = username;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
@Override
public String toString() {
    return "User [host=" + host + ", username=" + username + ", password=" +
password + "]";
}
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((host == null) ? 0 : host.hashCode());
    result = prime * result + ((password == null) ? 0 : password.hashCode());
    result = prime * result + ((username == null) ? 0 : username.hashCode());
    return result;
}
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    User other = (User) obj;
    if (host == null) {
        if (other.host != null)
            return false;
    }
    if (!host.equals(other.host))
        return false;
    if (password == null) {
        if (other.password != null)
            return false;
    }
    if (!password.equals(other.password))
        return false;
    if (username == null) {
        if (other.username != null)
            return false;
    }
    if (!username.equals(other.username))
        return false;
    return true;
}
```

```
        return false;
    } else if (!host.equals(other.host))
        return false;
    if (password == null) {
        if (other.password != null)
            return false;
    } else if (!password.equals(other.password))
        return false;
    if (username == null) {
        if (other.username != null)
            return false;
    } else if (!username.equals(other.username))
        return false;
    return true;
}

}
```

(4) hashCode()

public int hashCode(): 返回每个对象的hash值。

如果重写equals，那么通常会一起重写hashCode()方法， hashCode()方法主要是为了当对象存储到哈希表（后面集合章节学习）等容器中时提高存储和查询性能用的，这是因为关于hashCode有两个常规协定：

- ①如果两个对象的hash值是不同的，那么这两个对象一定不相等；
- ②如果两个对象的hash值是相同的，那么这两个对象不一定相等。

重写equals和hashCode方法时，要保证满足如下要求：

- ①如果两个对象调用equals返回true，那么要求这两个对象的hashCode值一定是相等的；
- ②如果两个对象的hashCode值不同的，那么要求这个两个对象调用equals方法一定是false；
- ③如果两个对象的hashCode值相同的，那么这个两个对象调用equals可能是true，也可能是false

```
public static void main(String[] args) {
    System.out.println("Aa".hashCode());//2112
    System.out.println("BB".hashCode());//2112
}
```

(5) finalize()

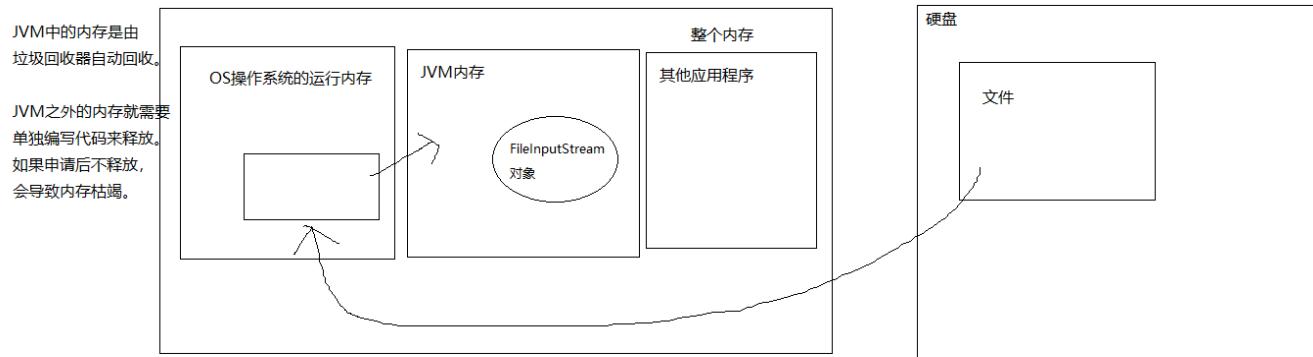
protected void finalize(): 用于最终清理内存的方法

Java定义了finalize()方法，用于在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。在《Java编程思想》一书中，有这样一段话：

Java有垃圾回收器负责回收无用对象占据的内存资源。但也有特殊情况：假定你的对象（并非使用new）获得了一块“特殊”的内存区域，由于垃圾回收器只知道释放那些经由new分配的内存，所有它不知道如何释放该对象的这块“特殊”内存。为了应对这种情况，Java允许在类中定义一个名为finalize()的方法。它的工作原理“假定”是这样的：一旦垃圾回收器准备好释放对象占用的存储空间，将首先调用其finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。所以要是你打算用finalize()，就能在垃圾回收时刻做一些重要的清理工作。

例如：

很多资源类中会用到一些native方法，本地方法是一种在Java中调用非Java代码的方式。在非Java代码中，也许会调用C的malloc()函数系列来分配存储空间，而且除非调用了free()函数，否则存储空间将得不到释放，从而造成内存泄漏。当然，free()是C和C++中的函数，所以需要在finalize()中用本地方法调用它。



FileInputStream类的对象在JVM内存中
文件读取结束，FileInputStream对象称为垃圾，会由GC（垃圾回收器）自动回收。
FileInputStream在读取文件内容时，需要操作系统协助，就会在JVM以外的操作系统部分分配内存，用完之后，也需要释放。
这个内存的释放需要在finalize方法中编写。
FileInputStream对象使用完之后，通常会调用close()方法，表示这个对象不用了。GC就会调用finalize方法。

演示finalize()方法被调用：

```
package com.atguigu.api;

public class TestFinalize {
    public static void main(String[] args) throws Throwable{
        for (int i=1; i <=10; i++){
            MyDemo my = new MyDemo(i);
            //每一次循环my就会指向新的对象，那么上次的对象就没有变量引用它了，就成垃圾对象
        }

        //为了看到垃圾回收器工作，我要加下面的代码，让main方法不那么快结束，因为main结束就会导致JVM退出，GC也会跟着结束。
        System.gc(); //如果不调用这句代码，GC可能不工作，因为当前内存很充足，GC就觉得不着急回收垃圾对象。
        //调用这句代码，会让GC尽快来工作。
        Thread.sleep(5000); //单位是毫秒，让当前程序休眠5秒再结束
    }
}

class MyDemo{
    private int value;
```

```

public MyDemo(int value) {
    this.value = value;
}

@Override
public String toString() {
    return "MyDemo{" + "value=" + value + '}';
}

//重写finalize方法，让大家看一下它的调用效果
@Override
protected void finalize() throws Throwable {
//    正常重写，这里是编写清理系统内存的代码
//    这里写输出语句是为了看到finalize()方法被调用的效果
    System.out.println(this+ "轻轻的走了，不带走一段代码....");
}
}

```

每一个对象的finalize()只会被调用一次，哪怕它多次被标记为垃圾对象。当一个对象没有有效的引用/变量指向它，那么这个对象就是垃圾对象。GC（垃圾回收器）通常会在第一次回收某个垃圾对象之前，先调用一下它的finalize()方法，然后再彻底回收它。但是如果在finalize()方法，这个垃圾对象“复活”了（即在finalize()方法中意外的又有某个引用指向了当前对象，这是要避免的），被“复活”的对象如果再次称为垃圾对象，GC就不再调用它的finalize方法了，避免这个对象称为“僵尸”。

```

package com.atguigu.api;

public class TestFinalize {
    private static MyDemo[] arr = new MyDemo[10];
    private static int total;

    public static void add(MyDemo demo){
        arr[total++] = demo;
    }

    public static void main(String[] args) throws Throwable{
        for (int i=1; i <=10; i++){
            MyDemo my = new MyDemo(i);
            //每一次循环my就会指向新的对象，那么上次的对象就没有变量引用它了，就成垃圾对象
        }
    }

    //为了看到垃圾回收器工作，我要加下面的代码，让main方法不那么快结束，因为main结束就会导致JVM退出，GC
    //也会跟着结束。
    System.gc(); //如果不调用这句代码，GC可能不工作，因为当前内存很充足，GC就觉得不着急回收垃圾对象。
    //调用这句代码，会让GC尽快来工作。
    Thread.sleep(5000); //单位是毫秒，让当前程序休眠5秒再结束

    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]); //MyDemo的对象还在，没有被回收掉，因为在回收过程中被复活了
    }

    for (int i = 0; i < arr.length; i++) {

```

```

        arr[i] = null;//让这些元素不引用MyDemo的对象，这些对象再次称为垃圾对象
        System.out.println(arr[i]);
    }
    arr = null;

    System.gc();//再次让GC工作，使得MyDemo的对象再次被回收
    Thread.sleep(5000);//单位是毫秒，让当前程序休眠5秒再结束
}
}

class MyDemo{
    private int value;

    public MyDemo(int value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "MyDemo{" + "value=" + value + '}';
    }

    //重写finalize方法，让大家看一下它的调用效果
    @Override
    protected void finalize() throws Throwable {
//        正常重写，这里是编写清理系统内存的代码
//        这里写输出语句是为了看到finalize()方法被调用的效果
        System.out.println("我轻轻的走了，不带走一段代码....");

        TestFinalize.add(this);
        //把当前对象this放到一个数组中，这样就有变量引用它，当前对象就不能被回收了
        //当下次this对象再次称为垃圾对象之后，GC就不会调用它的finalize()方法了
    }
}

```

面试题：对finalize()的理解？

- 当对象被GC确定为要被回收的垃圾，在回收之前由GC帮你调用这个方法，不是由程序员手动调用。
- 这个方法与C语言的析构函数不同，C语言的析构函数被调用，那么对象一定被销毁，内存被回收，而finalize方法的调用不一定会销毁当前对象，因为可能在finalize()中出现了让当前对象“复活”的代码
- 每一个对象的finalize方法只会被调用一次。
- 子类可以选择重写，一般用于彻底释放一些资源对象，而且这些资源对象往往时通过C/C++等代码申请的资源内存

(6) 重写toString、equals和hashCode方法 (Alt+Insert)

建议使用IDEA中的Alt + Insert快捷键，而不是Ctrl + O快捷键。

3. 标准JavaBean

JavaBean 是 Java语言编写类的一种标准规范。符合 JavaBean 的类，要求：

- (1) 类必须是具体的和公共的,
- (2) 并且具有无参数的构造方法,
- (3) 成员变量私有化，并提供用来操作成员变量的 set 和 get 方法。
- (4) 重写toString方法

```
public class className{  
    //成员变量  
  
    //构造方法  
    //无参构造方法【必须】  
    //有参构造方法【建议】  
  
    //getXXX()  
    //setXXX()  
    //其他成员方法  
}
```

编写符合 JavaBean 规范的类，以学生类为例，标准代码如下：

```
public class Student {  
    // 成员变量  
    private String name;  
    private int age;  
  
    // 构造方法  
    public Student() {  
    }  
  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // get/set成员方法  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

```
//其他成员方法列表
public String toString(){
    return "姓名: " + name + ", 年龄: " + age;
}
}
```

测试类，代码如下：

```
public class TestStudent {
    public static void main(String[] args) {
        // 无参构造使用
        Student s = new Student();
        s.setName("柳岩");
        s.setAge(18);
        System.out.println(s.getName() + "----" + s.getAge());
        System.out.println(s);

        // 带参构造使用
        Student s2 = new Student("赵丽颖", 18);
        System.out.println(s2.getName() + "----" + s2.getAge());
        System.out.println(s2);
    }
}
```

第7章 面向对象基础（下）

7.1 静态

7.1.1 静态关键字（static）

在类中声明的实例变量，其值是每一个对象独立的。但是有些成员变量的值不需要或不能每一个对象单独存储一份，即有些成员变量和当前类的对象无关。

在类中声明的实例方法，在类的外面必须要先创建对象，才能调用。但是有些方法的调用和当前类的对象无关，那么创建对象就有点麻烦了。

此时，就需要将和当前类的对象无关的成员变量、成员方法声明为静态的（static）。

7.1.2 静态变量

1、语法格式

有static修饰的成员变量就是静态变量。

```
【修饰符】 class 类{
    【其他修饰符】 static 数据类型 静态变量名;
}
```

2、静态变量的特点

- 静态变量的默认值规则和实例变量一样。
- 静态变量值是所有对象共享。
- 静态变量的值存储在方法区。
- 静态变量在本类中，可以在任意方法、代码块、构造器中直接使用。
- 如果权限修饰符允许，在其他类中可以通过“类名.静态变量”直接访问，也可以通过“对象.静态变量”的方式访问（但是更推荐使用类名.静态变量的方式）。
- 静态变量的get/set方法也静态的，当局部变量与静态变量重名时，使用“类名.静态变量”进行区分。

| 分类 | 数据类型 | 默认值 |
|------|-----------------------------|----------|
| 基本类型 | 整数 (byte, short, int, long) | 0 |
| | 浮点数 (float, double) | 0.0 |
| | 字符 (char) | '\u0000' |
| 引用类型 | 布尔 (boolean) | false |
| | 数据类型 | 默认值 |
| 引用类型 | 数组, 类, 接口 | null |

演示：

```
package com.atguigu.keyword;

public class Employee {
    private static int total;//这里私有化，在类的外面必须使用get/set方法的方式来访问静态变量
    static String company; //这里缺省权限修饰符，是为了演示在类外面演示“类名.静态变量”的方式访问
    private int id;
    private String name;

    {
        //两个构造器的公共代码可以提前到非静态代码块
        total++;
        id = total; //这里使用total静态变量的值为id属性赋值
    }

    public Employee() {
    }

    public Employee(String name) {
        this.name = name;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

```

public String getName() {
    return name;
}

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

public static int getTotal() {
    return total;
}

public static void setTotal(int total) {
    Employee.total = total;
}

@Override
public String toString() {
    return "Employee{company = " + company + ",id = " + id + " ,name=" + name + "}";
}
}

```

```

package com.atguigu.keyword;

public class TestStaticVariable {
    public static void main(String[] args) {
        //静态变量total的默认值是0
        System.out.println("Employee.total = " + Employee.getTotal());

        Employee c1 = new Employee("张三");
        Employee c2 = new Employee();
        System.out.println(c1); //静态变量company的默认值是null
        System.out.println(c2); //静态变量company的默认值是null
        System.out.println("Employee.total = " + Employee.getTotal()); //静态变量total值是2

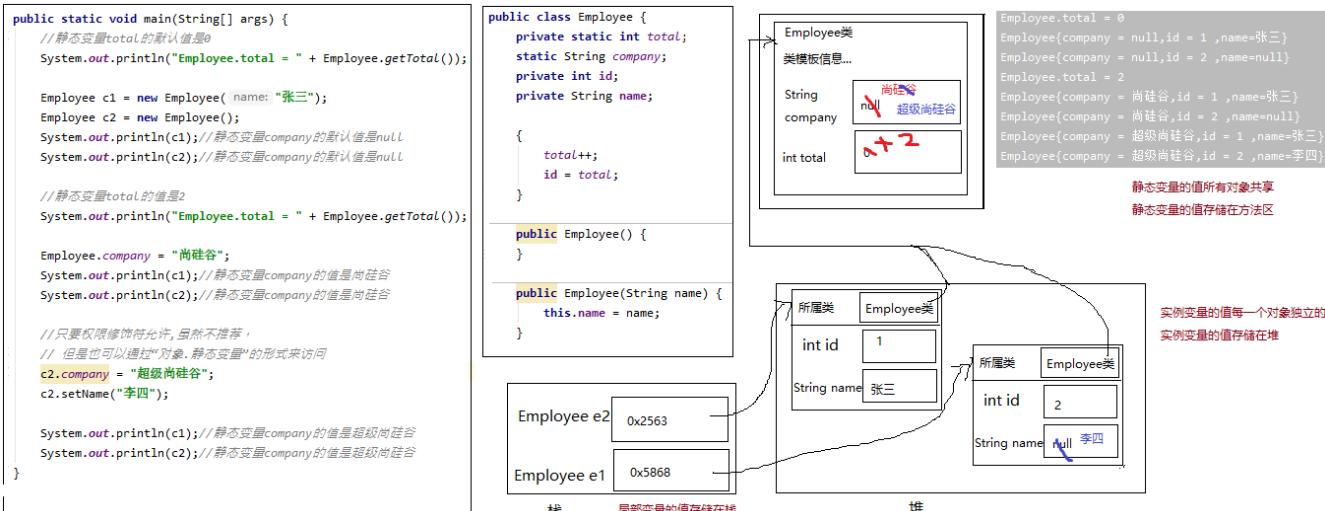
        Employee.company = "尚硅谷";
        System.out.println(c1); //静态变量company的值是尚硅谷
        System.out.println(c2); //静态变量company的值是尚硅谷

        //只要权限修饰符允许，虽然不推荐，但是也可以通过“对象.静态变量”的形式来访问
        c1.company = "超级尚硅谷";

        System.out.println(c1); //静态变量company的值是超级尚硅谷
        System.out.println(c2); //静态变量company的值是超级尚硅谷
    }
}

```

3、静态变量内存分析



4、静态类变量和非静态实例变量、局部变量

- 静态类变量（简称静态变量）：存储在方法区，有默认值，所有对象共享，生命周期和类相同，还可以有权限修饰符、final等其他修饰符
- 非静态实例变量（简称实例变量）：存储在堆中，有默认值，每一个对象独立，生命周期每一个对象也独立，还可以有权限修饰符、final等其他修饰符
- 局部变量：存储在栈中，没有默认值，每一次方法调用都是独立的，有作用域，只能有final修饰，没有其他修饰符

7.1.3 静态方法

1、语法格式

有static修饰的成员方法就是静态方法。

```
【修饰符】 class 类{
    【其他修饰符】 static 返回值类型 方法名(形参列表){
        方法体
    }
}
```

2、静态方法的特点

- 静态方法在本类的任意方法、代码块、构造器中都可以直接被调用。
- 只要权限修饰符允许，静态方法在其他类中可以通过“类名.静态方法”的方式调用。也可以通过“对象.静态方法”的方式调用（但是更推荐使用类名.静态方法的方式）。
- 静态方法可以被子类继承，但不能被子类重写。
- 静态方法的调用都只看编译时类型。

```
package com.atguigu.keyword;

public class Father {
    public static void method(){
        System.out.println("Father.method");
    }

    public static void fun(){
        System.out.println("Father.fun");
    }
}
```

```
package com.atguigu.keyword;

public class Son extends Father{
//    @Override //尝试重写静态方法，加上@Override编译报错，去掉@Override不报错，但是也不是重写
    public static void fun(){
        System.out.println("Son.fun");
    }
}
```

```
package com.atguigu.keyword;

public class TestStaticMethod {
    public static void main(String[] args) {
        Father.method();
        Son.method(); //继承静态方法

        Father f = new Son();
        f.method(); //执行Father类中的method
    }
}
```

7.1.4 静态代码块

如果想要为静态变量初始化，可以直接在静态变量的声明后面直接赋值，也可以使用静态代码块。

1、语法格式

在代码块的前面加static，就是静态代码块。

```
【修饰符】 class 类{
    static{
        静态代码块
    }
}
```

2、静态代码块的特点

每一个类的静态代码块只会执行一次。

静态代码块的执行优先于非静态代码块和构造器。

```
package com.atguigu.keyword;

public class Chinese {
//    private static String country = "中国";

    private static String country;
    private String name;

    {
        System.out.println("非静态代码块, country = " + country);
    }

    static {
        country = "中国";
        System.out.println("静态代码块");
    }

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

```
package com.atguigu.keyword;

public class TestStaticBlock {
    public static void main(String[] args) {
        Chinese c1 = new Chinese("张三");
        Chinese c2 = new Chinese("李四");
    }
}
```

3、静态代码块和非静态代码块

静态代码块在类初始化时执行，只执行一次

非静态代码块在实例初始化时执行，每次new对象都会执行

7.1.5 类初始化

(1) 类的初始化就是为静态变量初始化。实际上，类初始化的过程是在调用一个()方法，而这个方法是编译器自动生成的。编译器会将如下两部分的**所有**代码，**按顺序**合并到类初始化()方法体中。

- 静态类成员变量的显式赋值语句
- 静态代码块中的语句

(2) 每个类初始化只会进行一次，如果子类初始化时，发现父类没有初始化，那么会先初始化父类。

(3) 类的初始化一定优先于实例初始化。

1、类初始化代码只执行一次

```
package com.atguigu.keyword;

public class Fu{
    static{
        System.out.println("Fu静态代码块1, a = " + Fu.a);
    }
    private static int a = 1;
    static{
        System.out.println("Fu静态代码块2, a = " + a);
    }

    public static void method(){
        System.out.println("Fu.method");
    }

}
```

```
package com.atguigu.keyword;

public class TestClassInit {
    public static void main(String[] args) {
        Fu.method();
    }
}
```

2、父类优先于子类初始化

```
package com.atguigu.keyword;

public class Zi extends Fu{
    static{
        System.out.println("Zi静态代码块");
    }
}
```

```
package com.atguigu.keyword;

public class TestZiInit {
    public static void main(String[] args) {
        Zi z = new Zi();
    }
}
```

3、类初始化优先于实例初始化

```
package com.atguigu.keyword;

public class Fu{
    static{
        System.out.println("Fu静态代码块1, a = " + Fu.a);
```

```
}

private static int a = 1;
static{
    System.out.println("Fu静态代码块2, a = " + a);
}

{
    System.out.println("Fu非静态代码块");
}
public Fu(){
    System.out.println("Fu构造器");
}

public static void method(){
    System.out.println("Fu.method");
}
}
```

```
package com.atguigu.keyword;

public class Zi extends Fu{
    static{
        System.out.println("Zi静态代码块");
    }
    {
        System.out.println("Zi非静态代码块");
    }
    public Zi(){
        System.out.println("Zi构造器");
    }
}
```

```
package com.atguigu.keyword;

public class TestZiInit {
    public static void main(String[] args) {
        Zi z1 = new Zi();
        Zi z2 = new Zi();
    }
}
```

7.1.6 静态和非静态的区别

1、本类中的访问限制区别

静态的类变量和静态的方法可以在本类的任意方法、代码块、构造器中直接访问。

非静态的实例变量和非静态的方法只能在本类的非静态的方法、非静态代码块、构造器中直接访问。

即：

- 静态直接访问静态，可以
- 非静态直接访问非静态，可以
- 非静态直接访问静态，不可以
- 静态直接访问非静态，不可以

2、在其他类的访问方式区别

静态的类变量和静态的方法可以通过“类名.”的方式直接访问；也可以通过“对象.”的方式访问。（但是更推荐使用`==>类名.==>`的方式）

非静态的实例变量和非静态的方法只能通过“对象.”方式访问。

3、this和super的使用

静态的方法和静态的代码块中，不允许出现this和super关键字，如果有重名问题，使用“类名.”进行区别。

非静态的方法和非静态的代码块中，可以使用this和super关键字。

7.1.7 静态导入

如果大量使用另一个类的静态成员，可以使用静态导入，简化代码。

```
import static 包.类名.静态成员名;
import static 包.类名.*;
```

演示：

```
package com.atguigu.keyword;

import static java.lang.Math.*;

public class TestStaticImport {
    public static void main(String[] args) {
        //使用Math类的静态成员
        System.out.println(Math.PI);
        System.out.println(Math.sqrt(9));
        System.out.println(Math.random());

        System.out.println("-----");
        System.out.println PI;
        System.out.println(sqrt(9));
        System.out.println(random());
    }
}
```

7.2 枚举

7.2.1 概述

某些类型的对象是有限的几个，这样的例子举不胜举：

- 星期：Monday(星期一).....Sunday(星期天)
- 性别：Man(男)、Woman(女)
- 月份：January(1月).....December(12月)
- 季节：Spring(春节).....Winter(冬天)
- 支付方式：Cash (现金)、WeChatPay (微信)、Alipay(支付宝)、BankCard(银行卡)、CreditCard(信用卡)
- 员工工作状态：Busy (忙)、Free (闲)、Vocation (休假)
- 订单状态：Nonpayment (未付款)、Paid (已付款)、Fulfilled (已配货)、Delivered (已发货)、Checked (已确认收货)、Return (退货)、Exchange (换货)、Cancel (取消)

枚举类型本质上也是一种类，只不过是这个类的对象是固定的几个，而不能随意让用户创建。

在JDK1.5之前，需要程序员自己通过特殊的方式来定义枚举类型。

在JDK1.5之后，Java支持enum关键字来快速的定义枚举类型。

7.2.2 JDK1.5之前

在JDK1.5之前如何声明枚举类呢？

- 构造器加private私有化
- 本类内部创建一组常量对象，并添加public static修饰符，对外暴露这些常量对象

示例代码：

```
public class Season{  
    public static final Season SPRING = new Season();  
    public static final Season SUMMER = new Season();  
    public static final Season AUTUMN = new Season();  
    public static final Season WINTER = new Season();  
  
    private Season(){  
    }  
  
    public String toString(){  
        if(this == SPRING){  
            return "春";  
        }else if(this == SUMMER){  
            return "夏";  
        }else if(this == AUTUMN){  
            return "秋";  
        }else{  
            return "冬";  
        }  
    }  
}
```

```
public class TestSeason {  
    public static void main(String[] args) {  
        Season spring = Season.SPRING;  
        System.out.println(spring);  
    }  
}
```

7.2.3 JDK1.5之后

1、enum关键字声明枚举

```
【修饰符】 enum 枚举类名{  
    常量对象列表  
}
```

```
【修饰符】 enum 枚举类名{  
    常量对象列表;  
  
    其他成员列表;  
}
```

示例代码：

```
package com.atguigu.enumeration;  
  
public enum Week {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}
```

```
public class TestEnum {  
    public static void main(String[] args) {  
        Season spring = Season.SPRING;  
        System.out.println(spring);  
    }  
}
```

2、枚举类的要求和特点

枚举类的要求和特点：

- 枚举类的常量对象列表必须在枚举类的首行，因为是常量，所以建议大写。
- 如果常量对象列表后面没有其他代码，那么“；”可以省略，否则不可以省略“；”。
- 编译器给枚举类默认提供的是private的无参构造，如果枚举类需要的是无参构造，就不需要声明，写常量对象列表时也不用加参数。
- 如果枚举类需要的是有参构造，需要手动定义，有参构造的private可以省略，调用有参构造的方法就是在常量对象名后面加(实参列表)就可以。
- 枚举类默认继承的是java.lang.Enum类，因此不能再继承其他的类型。
- JDK1.5之后switch，提供支持枚举类型，case后面可以写枚举常量名。

- 枚举类型如有其它属性，建议（**不是必须**）这些属性也声明为final的，因为常量对象在逻辑意义上应该不可变。

示例代码：

```
package com.atguigu.enumeration;

public enum Week {
    MONDAY("星期一"),
    TUESDAY("星期二"),
    WEDNESDAY("星期三"),
    THURSDAY("星期四"),
    FRIDAY("星期五"),
    SATURDAY("星期六"),
    SUNDAY("星期日");

    private final String description;

    private Week(String description){
        this.description = description;
    }

    @Override
    public String toString() {
        return super.toString() + ":" + description;
    }
}
```

```
package com.atguigu.enumeration;

public class TestWeek {
    public static void main(String[] args) {
        Week week = Week.MONDAY;
        System.out.println(week);

        switch (week){
            case MONDAY:
                System.out.println("怀念周末，困意很浓");break;
            case TUESDAY:
                System.out.println("进入学习状态");break;
            case WEDNESDAY:
                System.out.println("死撑");break;
            case THURSDAY:
                System.out.println("小放松");break;
            case FRIDAY:
                System.out.println("又信心满满");break;
            case SATURDAY:
                System.out.println("开始盼周末，无心学习");break;
            case SUNDAY:
                System.out.println("一觉到下午");break;
        }
    }
}
```

```
    }  
}
```

3、枚举类型常用方法

- 1. `String toString()`: 默认返回的是常量名（对象名），可以继续手动重写该方法！
- 2. `String name()`: 返回的是常量名（对象名）
- 3. `int ordinal()`: 返回常量的次序号，默认从0开始
- 4. 枚举类型`[] values()`: 返回该枚举类的所有常量对象，返回类型是当前枚举的数组类型，是一个静态方法
- 5. 枚举类型 `valueOf(String name)`: 根据枚举常量对象名称获取枚举对象

示例代码：

```
package com.atguigu.enumeration;  
  
import java.util.Scanner;  
  
public class TestEnumMethod {  
    public static void main(String[] args) {  
        Week[] values = Week.values();  
        for (int i = 0; i < values.length; i++) {  
            System.out.println((values[i].ordinal() + 1) + "->" + values[i].name());  
        }  
        System.out.println("-----");  
  
        Scanner input = new Scanner(System.in);  
        System.out.print("请输入星期值: ");  
        int weekValue = input.nextInt();  
        Week week = values[weekValue - 1];  
        System.out.println(week);  
  
        System.out.print("请输入星期名: ");  
        String weekName = input.next();  
        week = Week.valueOf(weekName);  
        System.out.println(week);  
  
        input.close();  
    }  
}
```

7.3 包装类

7.3.1 包装类

Java提供了两个类型系统，基本类型与引用类型，使用基本类型在于效率，然而当要使用只针对对象设计的API或新特性（例如泛型），那么基本数据类型的数据就需要用包装类来包装。

| 序号 | 基本数据类型 | 包装类 (java.lang包) |
|----|---------|------------------|
| 1 | byte | Byte |
| 2 | short | Short |
| 3 | int | Integer |
| 4 | long | Long |
| 5 | float | Float |
| 6 | double | Double |
| 7 | char | Character |
| 8 | boolean | Boolean |
| 9 | void | Void |

7.3.2 装箱与拆箱

装箱：把基本数据类型转为包装类对象。

 转为包装类的对象，是为了使用专门为对象设计的API和特性

拆箱：把包装类对象拆为基本数据类型。

 转为基本数据类型，一般是因为需要运算，Java中的大多数运算符是为基本数据类型设计的。比较、算术等
基本数值---->包装对象

```
Integer obj1 = new Integer(4); //使用构造函数函数
Integer obj2 = Integer.valueOf(4); //使用包装类中的valueOf方法
```

包装对象---->基本数值

```
Integer obj = new Integer(4);
int num1 = obj.intValue();
```

JDK1.5之后，可以自动装箱与拆箱。

 注意：只能与自己对应的类型之间才能实现自动装箱与拆箱。

```
Integer i = 4; //自动装箱。相当于Integer i = Integer.valueOf(4);
i = i + 5; //等号右边：将i对象转成基本数值(自动拆箱) i.intValue() + 5;
//加法运算完成后，再次装箱，把基本数值转成对象。
```

```
Integer i = 1;
Double d = 1; //错误的，1是int类型
```

7.3.3 包装类的一些API

1、基本数据类型和字符串之间的转换

(1) 把基本数据类型转为字符串

```
int a = 10;
//String str = a;//错误的
//方式一:
String str = a + "";
//方式二:
String str = String.valueOf(a);
```

(2) 把字符串转为基本数据类型

String转换成对应的基本类型，除了Character类之外，其他所有包装类都具有parseXxx静态方法可以将字符串参数转换为对应的基本类型，例如：

- `public static int parseInt(String s)`：将字符串参数转换为对应的int基本类型。
- `public static long parseLong(String s)`：将字符串参数转换为对应的long基本类型。
- `public static double parseDouble(String s)`：将字符串参数转换为对应的double基本类型。

或把字符串转为包装类，然后可以自动拆箱为基本数据类型

- `public static Integer valueof(String s)`：将字符串参数转换为对应的Integer包装类，然后可以自动拆箱为int基本类型
- `public static Long valueof(String s)`：将字符串参数转换为对应的Long包装类，然后可以自动拆箱为long基本类型
- `public static Double valueof(String s)`：将字符串参数转换为对应的Double包装类，然后可以自动拆箱为double基本类型

注意：如果字符串参数的内容无法正确转换为对应的基本类型，则会抛出 `java.lang.NumberFormatException` 异常。

```
int a = Integer.parseInt("整数的字符串");
double d = Double.parseDouble("小数的字符串");
boolean b = Boolean.parseBoolean("true或false");

int a = Integer.valueOf("整数的字符串");
double d = Double.valueOf("小数的字符串");
boolean b = Boolean.valueOf("true或false");
```

2、数据类型的最大最小值

`Integer.MAX_VALUE` 和 `Integer.MIN_VALUE`
`Long.MAX_VALUE` 和 `Long.MIN_VALUE`
`Double.MAX_VALUE` 和 `Double.MIN_VALUE`

3、字符转大小写

```
Character.toUpperCase('x');
Character.toLowerCase('X');
```

4、整数转进制

```
Integer.toBinaryString(int i)
Integer.toHexString(int i)
Integer.toOctalString(int i)
```

5、比较的方法

```
Double.compare(double d1, double d2)
Integer.compare(int x, int y)
```

7.3.4 包装类对象的特点

1、包装类缓存对象

| 包装类 | 缓存对象 |
|-----------|------------|
| Byte | -128~127 |
| Short | -128~127 |
| Integer | -128~127 |
| Long | -128~127 |
| Float | 没有 |
| Double | 没有 |
| Character | 0~127 |
| Boolean | true和false |

```
Integer a = 1;
Integer b = 1;
System.out.println(a == b); //true

Integer i = 128;
Integer j = 128;
System.out.println(i == j); //false

Integer m = new Integer(1); //new的在堆中
Integer n = 1; //这个用的是缓冲的常量对象，在方法区
System.out.println(m == n); //false

Integer x = new Integer(1); //new的在堆中
```

```
Integer y = new Integer(1); //另一个new的在堆中
System.out.println(x == y); //false
```

```
Double d1 = 1.0;
Double d2 = 1.0;
System.out.println(d1==d2); //false 比较地址，没有缓存对象，每一个都是新new的
```

2、类型转换问题

```
Integer i = 1000;
double j = 1000;
System.out.println(i==j); //true 会先将i自动拆箱为int，然后根据基本数据类型“自动类型转换”规则，转为
double比较
```

```
Integer i = 1000;
int j = 1000;
System.out.println(i==j); //true 会自动拆箱，按照基本数据类型进行比较
```

```
Integer i = 1;
Double d = 1.0
System.out.println(i==d); //编译报错
```

3、包装类对象不可变

```
public class TestExam {
    public static void main(String[] args) {
        int i = 1;
        Integer j = new Integer(2);
        Circle c = new Circle();
        change(i,j,c);
        System.out.println("i = " + i); //1
        System.out.println("j = " + j); //2
        System.out.println("c.radius = " + c.radius); //10.0
    }

    /*
     * 方法的参数传递机制：
     * (1) 基本数据类型：形参的修改完全不影响实参
     * (2) 引用数据类型：通过形参修改对象的属性值，会影响实参的属性值
     * 这类Integer等包装类对象是“不可变”对象，即一旦修改，就是新对象，和实参就无关了
     */
    public static void change(int a ,Integer b,Circle c){
        a += 10;
        // b += 10; //等价于 b = new Integer(b+10);
        c.radius += 10;
        /*c = new Circle();
        c.radius+=10;*/
    }
}

class Circle{
```

```
    double radius;  
}
```

7.4 抽象类

7.4.1 由来

抽象：即不具体、或无法具体

例如：当我们声明一个几何图形类：圆、矩形、三角形类等，发现这些类都有共同特征：求面积、求周长、获取图形详细信息。那么这些共同特征应该抽取到一个公共父类中。但是这些方法在父类中又**无法给出具体的实现**，而是应该交给子类各自具体实现。那么父类在声明这些方法时，**就只有方法签名，没有方法体**，我们把没有方法体的方法称为**抽象方法**。Java语法规规定，包含抽象方法的类必须是**抽象类**。

7.4.2 语法格式

- **抽象方法**：被abstract修饰没有方法体的方法。
- **抽象类**：被abstract修饰的类。

抽象类的语法格式

```
【权限修饰符】 abstract class 类名{  
}  
【权限修饰符】 abstract class 类名 extends 父类{  
}
```

抽象方法的语法格式

```
【其他修饰符】 abstract 返回值类型 方法名(【形参列表】);
```

注意：抽象方法没有方法体

代码举例：

```
public abstract class Animal {  
    public abstract void eat();  
}
```

```
public class Cat extends Animal {  
    public void run (){  
        System.out.println("小猫吃鱼和猫粮");  
    }  
}
```

```

public class CatTest {
    public static void main(String[] args) {
        // 创建子类对象
        Cat c = new Cat();

        // 调用eat方法
        c.eat();
    }
}

```

此时的方法重写，是子类对父类抽象方法的完成实现，我们将这种方法重写的操作，也叫做**实现方法**。

7.3.3 注意事项

关于抽象类的使用，以下为语法上要注意的细节，虽然条目较多，但若理解了抽象的本质，无需死记硬背。

1. 抽象类**不能创建对象**，如果创建，编译无法通过而报错。只能创建其非抽象子类的对象。

理解：假设创建了抽象类的对象，调用抽象的方法，而抽象方法没有具体的方法体，没有意义。

2. 抽象类中，也有构造方法，是供子类创建对象时，初始化父类成员变量使用的。

理解：子类的构造方法中，有默认的super()或手动的super(实参列表)，需要访问父类构造方法。

3. 抽象类中，不一定包含抽象方法，但是有抽象方法的类必定是抽象类。

理解：未包含抽象方法的抽象类，目的就是不想让调用者创建该类对象，通常用于某些特殊的类结构设计。

4. 抽象类的子类，必须重写抽象父类中**所有的**抽象方法，否则，编译无法通过而报错。除非该子类也是抽象类。

理解：假设不重写所有抽象方法，则类中可能包含抽象方法。那么创建对象后，调用抽象的方法，没有意义。

7.3.4 修饰符一起使用问题？

| | 外部类 | 成员变量 | 代码块 | 构造器 | 方法 | 局部变量 | 内部类（后面讲） |
|-----------|-----|------|-----|-----|----|------|----------|
| public | √ | √ | × | √ | √ | × | √ |
| protected | × | √ | × | √ | √ | × | √ |
| 缺省 | √ | √ | × | √ | √ | × | √ |
| private | × | √ | × | √ | √ | × | √ |
| static | × | √ | √ | × | √ | × | √ |
| final | √ | √ | × | × | √ | √ | √ |
| abstract | √ | × | × | × | √ | × | √ |
| native | × | × | × | × | √ | × | × |

不能和abstract一起使用的修饰符？

(1) abstract和final不能一起修饰方法和类

(2) abstract和static不能一起修饰方法

(3) abstract和native不能一起修饰方法

(4) abstract和private不能一起修饰方法

static和final一起使用：

(1) 修饰方法：可以，因为都不能被重写

(2) 修饰成员变量：可以，表示静态常量

(3) 修饰局部变量：不可以， static不能修饰局部变量

(4) 修饰代码块：不可以， final不能修改代码块

(5) 修饰内部类：可以一起修饰成员内部类，不能一起修饰局部内部类

7.5 接口

7.5.1 概述

生活中大家每天都在用USB接口，那么USB接口与我们今天要学习的接口有什么相同点呢？

USB是通用串行总线的英文缩写，是Intel公司开发的总线架构，使得在计算机上添加串行设备（鼠标、键盘、打印机、扫描仪、摄像头、充电器、MP3机、手机、数码相机、移动硬盘等）非常容易。只须将设备插入计算机的USB端口中，系统会自动识别和配置。有了USB，我们电脑需要提供的各种插槽的口越来越少，而能支持的其他设备的连接却越来越多。

那么我们平时看到的电脑上的USB插口、以及其他设备上的USB插口是什么呢？

其实，不管是电脑上的USB插口，还是其他设备上的USB插口都只是遵循了USB规范的一种具体设备而已。

根据时代发展，USB接口标准经历了一代USB、第二代USB 2.0和第三代USB 3.0。

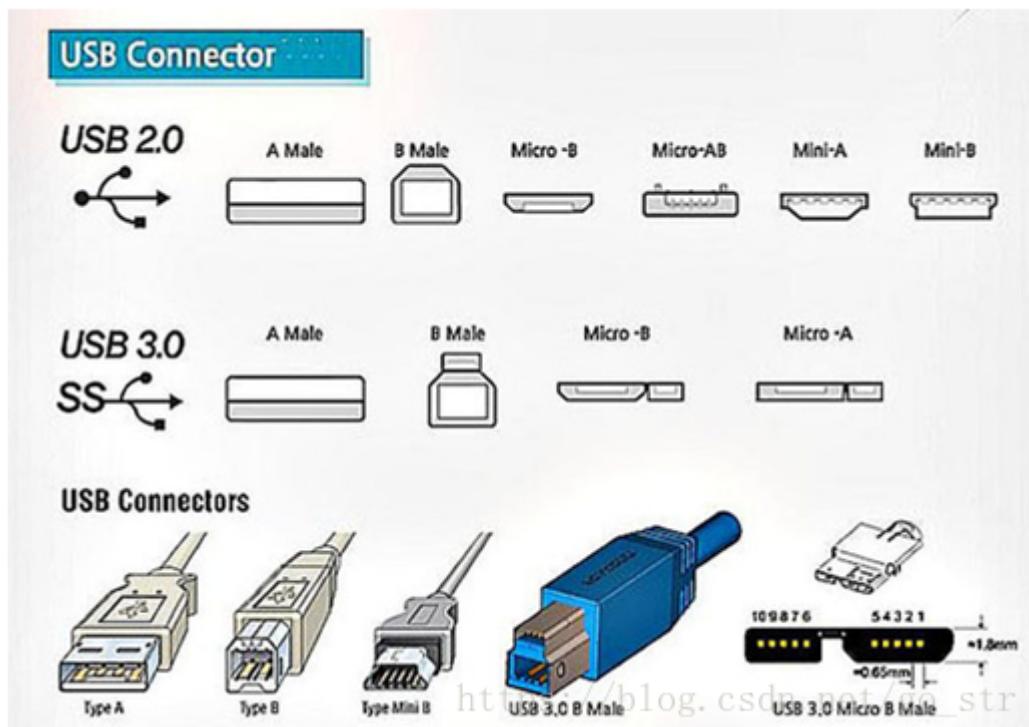
USB规格第一次是于1995年，由Intel、IBM、Compaq、Microsoft、NEC、Digital、North Telecom等七家公司组成的USBIF(USB Implement Forum)共同提出，USBIF于1996年1月正式提出USB1.0规格，频宽为1.5Mbps。

USB2.0技术规范是有由Compaq、Hewlett Packard、Intel、Lucent、Microsoft、NEC、Philips共同制定、发布的，规范把外设数据传输速度提高到了480Mbps，被称为USB 2.0的高速(High-speed)版本。

USB 3.0是最新的USB规范，该规范由英特尔等公司发起，USB3.0的最大传输带宽高达5.0Gbps(640MB/s)，USB3.0引入全双工数据传输。5根线路中2根用来发送数据，另2根用来接收数据，还有1根是地线。也就是说，USB 3.0可以同步全速地进行读写操作。

| USB版本 | 最大传输速率 | 速率称号 | 最大输出电流 | 推出时间 |
|---------|------------------|--------------------|----------|----------|
| USB1.0 | 1.5Mbps(192KB/s) | 低速(Low-Speed) | 5V/500mA | 1996年1月 |
| USB1.1 | 12Mbps(1.5MB/s) | 全速(Full-Speed) | 5V/500mA | 1998年9月 |
| USB2.0 | 480Mbps(60MB/s) | 高速(High-Speed) | 5V/500mA | 2000年4月 |
| USB3.0 | 5Gbps(500MB/s) | 超高速(Super-Speed) | 5V/900mA | 2008年11月 |
| USB 3.1 | 10Gbps(1280MB/s) | 超高速+(Super-speed+) | 20V/5A | 2013年12月 |

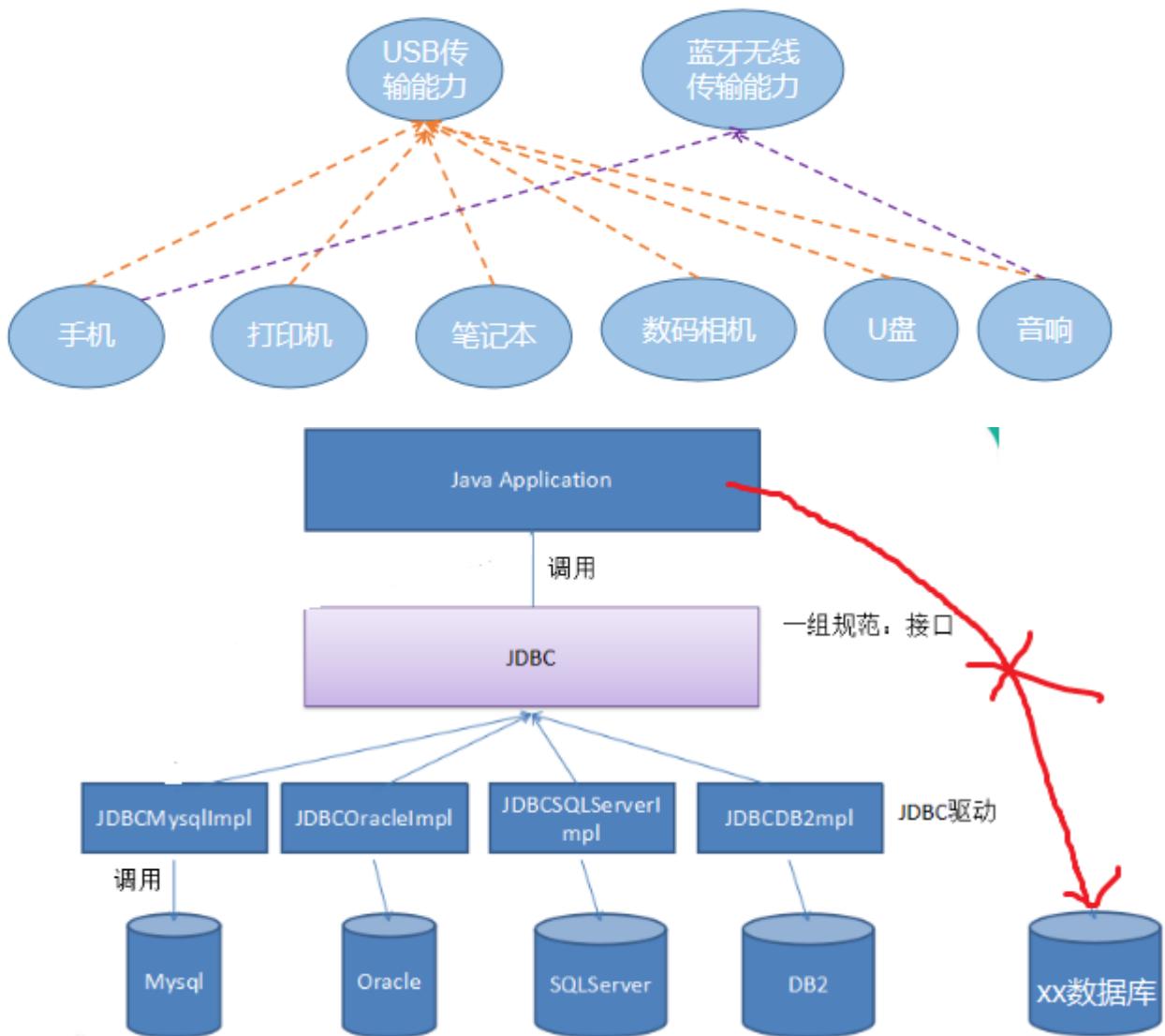
下面是USB2.0和USB3.0标准下的各类接口示意图：



电脑边上提供了USB插槽，这个插槽遵循了USB的规范，只要其他设备也是遵循USB规范的，那么就可以互联，并正常通信。至于这个电脑、以及其他设备是哪个厂家制造的，内部是如何实现的，我们都无需关心。

这种设计是将规范和实现分离，这也正是Java接口的好处。Java的软件系统会有很多模块组成，那么各个模块之间也应该采用这种面向接口的低耦合，为系统提供更好的可扩展性和可维护性。

- 接口就是规范，定义的是一组规则，体现了现实世界中“如果你是/要...则必须能...”的思想。继承是一个“是不是”的is-a关系，而接口实现则是“能不能”的has-a关系。
 - 例如：你能不能用USB进行连接，或是否具备USB通信功能，就看你是否遵循USB接口规范
 - 例如：Java程序是否能够连接使用某种数据库产品，那么要看该数据库产品有没有实现Java设计的JDBC规范



```

package com.atguigu.interfacetype;

public class Computer {
//    private Mouse mouse;//只能连接鼠标
//    private Keyboard keyboard;//只能连接键盘
    private Usb3 usb;
    //Usb3如果是类的话，有单继承限制
    //Usb3如果是接口的话，就不会有单继承限制

    public Usb3 getUsb() {
        return usb;
    }

    public void setUsb(Usb3 usb) {
        this.usb = usb;
    }
}

```

7.5.2 定义格式

接口的定义，它与定义类方式相似，但是使用 `interface` 关键字。它也会被编译成.class文件，但一定要明确它并不是类，而是另外一种引用数据类型。

引用数据类型：数组，类，枚举，接口，注解。

1、接口的声明格式

```
【修饰符】 interface 接口名{  
    //接口的成员列表:  
    // 公共的静态常量  
    // 公共的抽象方法  
    // 公共的默认方法 (JDK1.8以上)  
    // 公共的静态方法 (JDK1.8以上)  
    // 私有方法 (JDK1.9以上)  
}
```

示例代码：

```
package com.atguigu.interfacetype;  
  
public interface Usb3{  
    //静态常量  
    long MAX_SPEED = 500*1024*1024;//500MB/s  
  
    //抽象方法  
    void in();  
    void out();  
  
    //默认方法  
    default void start(){  
        System.out.println("开始");  
    }  
    default void stop(){  
        System.out.println("结束");  
    }  
  
    //静态方法  
    static void show(){  
        System.out.println("USB 3.0可以同步全速地进行读写操作");  
    }  
}
```

2、接口的成员说明

接口定义的是多个类共同的公共行为规范，这些行为规范是与外部交流的通道，这就意味着接口里通常是定义一组公共方法。

在JDK8之前，接口中只允许出现：

- (1) 公共的静态的常量：其中`public static final`可以省略
- (2) 公共的抽象的方法：其中`public abstract`可以省略

理解：接口是从多个相似类中抽象出来的规范，不需要提供具体实现

在JDK1.8时，接口中允许声明默认方法和静态方法：

(3) 公共的默认的方法：其中public 可以省略，建议保留，但是default不能省略

(4) 公共的静态的方法：其中public 可以省略，建议保留，但是static不能省略

在JDK1.9时，接口又增加了：

(5) 私有方法

除此之外，接口中不能有其他成员，没有构造器，没有初始化块，因为接口中没有成员变量需要动态初始化。

3、面试题拷问？

1、为什么接口中只能声明公共的静态的常量？

因为接口是标准规范，那么在规范中需要声明一些底线边界值，当实现者在实现这些规范时，不能去随意修改和触碰这些底线，否则就有“危险”。

例如：USB1.0规范中规定最大传输速率是1.5Mbps，最大输出电流是5V/500mA

USB3.0规范中规定最大传输速率是5Gbps(500MB/s)，最大输出电流是5V/900mA

例如：尚硅谷学生行为规范中规定学员，早上8:25之前进班，晚上21:30之后离开等等。

2、为什么JDK1.8之后要允许接口定义静态方法和默认方法呢？因为它违反了接口作为一个抽象标准定义的概念。

静态方法：因为之前的標準类库设计中，有很多Collection/Colletions或者Path/Paths这样成对的接口和类，后面的类中都是静态方法，而这些静态方法都是为前面的接口服务的，那么这样设计一对API，不如把静态方法直接定义到接口中使用和维护更方便。

默认方法：(1) 我们要在已有的老版接口中提供新方法时，如果添加抽象方法，就会涉及到原来使用这些接口的类就会有问题，那么为了保持与旧版本代码的兼容性，只能允许在接口中定义默认方法实现。比如：Java8中对Collection、List、Comparator等接口提供了丰富的默认方法。(2) 当我们接口的某个抽象方法，在很多实现类中的实现代码是一样的，此时将这个抽象方法设计为默认方法更为合适，那么实现类就可以选择重写，也可以选择不重写。

3、为什么JDK1.9要允许接口定义私有方法呢？因为我们说接口是规范，规范时需要公开让大家遵守的

私有方法：因为有了默认方法和静态方法这样具有具体实现的方法，那么就可能出现多个方法由共同的代码可以抽取，而这些共同的代码抽取出来的方法又只希望在接口内部使用，所以就增加了私有方法。

7.5.3 接口的使用

1、使用接口的静态成员

接口不能直接创建对象，但是可以通过接口名直接调用接口的静态方法和静态常量。

```
package com.atguigu.interfacetype;

public class Testusb3 {
    public static void main(String[] args) {
        //通过“接口名.”调用接口的静态方法
        Usb3.show();
        //通过“接口名.”直接使用接口的静态常量
        System.out.println(Usb3.MAX_SPEED);
    }
}
```

2、类实现接口 (implements)

接口**不能创建对象**，但是可以被类实现 (`implements`，类似于被继承)。

类与接口的关系为实现关系，即**类实现接口**，该类可以称为接口的实现类，也可以称为接口的子类。实现的动作类似继承，格式相仿，只是关键字不同，实现使用 `implements` 关键字。

```
【修饰符】 class 实现类 implements 接口{
    // 重写接口中抽象方法【必须】，当然如果实现类是抽象类，那么可以不重写
    // 重写接口中默认方法【可选】
}
```

```
【修饰符】 class 实现类 extends 父类 implements 接口{
    // 重写接口中抽象方法【必须】，当然如果实现类是抽象类，那么可以不重写
    // 重写接口中默认方法【可选】
}
```

注意：

1. 如果接口的实现类是非抽象类，那么必须==重写接口中所有抽象方法==。
2. 默认方法可以选择保留，也可以重写。

重写时，`default`单词就不要再写了，它只用于在接口中表示默认方法，到类中就没有默认方法的概念了

3. 接口中的静态方法不能被继承也不能被重写

示例代码：

```
package com.atguigu.interfacetype;

public class MobileHDD implements Usb3 {
    //重写/实现接口的抽象方法，【必选】
    public void out() {
        System.out.println("读取数据并发送");
    }
    public void in(){
        System.out.println("接收数据并写入");
    }

    //重写接口的默认方法，【可选】
    //重写默认方法时，default单词去掉
    public void end(){
```

```
        System.out.println("清理硬盘中的隐藏回收站中的东西，再结束");
    }
}
```

3、使用接口的非静态方法

- 对于接口的静态方法，直接使用“接口名.”进行调用即可
 - 也只能使用“接口名.”进行调用，不能通过实现类的对象进行调用
- 对于接口的抽象方法、默认方法，只能通过实现类对象才可以调用
 - 接口不能直接创建对象，只能创建实现类的对象

```
package com.atguigu.interfacetype;

public class TestMobileHDD {
    public static void main(String[] args) {
        //创建实现类对象
        MobileHDD b = new MobileHDD();

        //通过实现类对象调用重写的抽象方法，以及接口的默认方法，如果实现类重写了就执行重写的默认方法，如果没有重写，就执行接口中的默认方法
        b.start();
        b.in();
        b.stop();

        //通过接口名调用接口的静态方法
//        MobileHDD.show();
//        b.show();
        Usb3.show();
    }
}
```

4、接口的多实现 (implements)

之前学过，在继承体系中，一个类只能继承一个父类。而对于接口而言，一个类是可以实现多个接口的，这叫做接口的**多实现**。并且，一个类能继承一个父类，同时实现多个接口。

实现格式：

```
【修饰符】 class 实现类 implements 接口1, 接口2, 接口3。。。{
    // 重写接口中所有抽象方法【必须】，当然如果实现类是抽象类，那么可以不重写
    // 重写接口中默认方法【可选】
}

【修饰符】 class 实现类 extends 父类 implements 接口1, 接口2, 接口3。。。{
    // 重写接口中所有抽象方法【必须】，当然如果实现类是抽象类，那么可以不重写
    // 重写接口中默认方法【可选】
}
```

接口中，有多个抽象方法时，实现类必须重写所有抽象方法。**如果抽象方法有重名的，只需要重写一次。**

定义多个接口:

```
package com.atguigu.interfacetype;

public interface A {
    void showA();
    void show();
}
```

```
package com.atguigu.interfacetype;

public interface B extends A {
    void showB();
    void show();
}
```

定义实现类:

```
package com.atguigu.interfacetype;

public class C implements A,B {
    @Override
    public void showA() {
        System.out.println("showA");
    }

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

    @Override
    public void show() {
        System.out.println("show");
    }
}
```

测试类

```
package com.atguigu.interfacetype;

public class TestC {
    public static void main(String[] args) {
        C c = new C();
        c.showA();
        c.showB();
        c.show();
    }
}
```

5、接口的多继承 (extends)

一个接口能继承另一个或者多个接口，接口的继承也使用 `extends` 关键字，子接口继承父接口的方法。

定义父接口：

```
package com.atguigu.interfacetype;

public interface Chargeable {
    void charge();
    void in();
    void out();
}
```

定义子接口：

```
package com.atguigu.interfacetype;

public interface Usbc extends Chargeable, Usb3 {
    void reverse();
}
```

定义子接口的实现类：

```
package com.atguigu.interfacetype;

public class TypeCConverter implements Usbc {
    @Override
    public void reverse() {
        System.out.println("正反面都支持");
    }

    @Override
    public void charge() {
        System.out.println("可充电");
    }

    @Override
    public void in() {
        System.out.println("接收数据");
    }

    @Override
    public void out() {
        System.out.println("输出数据");
    }
}
```

所有父接口的抽象方法都有重写。

方法签名相同的抽象方法只需要实现一次。

6、接口与实现类对象构成多态引用

实现类实现接口，类似于子类继承父类，因此，接口类型的变量与实现类的对象之间，也可以构成多态引用。通过接口类型的变量调用方法，最终执行的是你new的实现类对象实现的方法体。

接口的不同实现类：

```
package com.atguigu.interfacetype;

public class Mouse implements Usb3 {
    @Override
    public void out() {
        System.out.println("发送脉冲信号");
    }

    @Override
    public void in() {
        System.out.println("不接收信号");
    }
}
```

```
package com.atguigu.interfacetype;

public class KeyBoard implements Usb3{
    @Override
    public void in() {
        System.out.println("不接收信号");
    }

    @Override
    public void out() {
        System.out.println("发送按键信号");
    }
}
```

测试类

```
package com.atguigu.interfacetype;

public class TestComputer {
    public static void main(String[] args) {
        Computer computer = new Computer();
        Usb3 usb = new Mouse();
        computer.setUsb(usb);
        usb.start();
        usb.out();
        usb.in();
        usb.stop();
        System.out.println("-----");

        usb = new KeyBoard();
```

```

        computer.setUsb(usb);
        usb.start();
        usb.out();
        usb.in();
        usb.stop();
        System.out.println("-----");

        usb = new MobileHDD();
        computer.setUsb(usb);
        usb.start();
        usb.out();
        usb.in();
        usb.stop();
    }
}

```

7.5.4 冲突问题

1、默认方法冲突问题

(1) 亲爹优先原则

当一个类，既继承一个父类，又实现若干个接口时，父类中的成员方法与接口中的抽象方法重名，子类就近选择执行父类的成员方法。代码如下：

定义接口：

```

package com.atguigu.interfacetype;

public interface Friend {
    default void date(){//约会
        System.out.println("吃喝玩乐");
    }
}

```

定义父类：

```

package com.atguigu.interfacetype;

public class Father {
    public void date(){//约会
        System.out.println("爸爸约吃饭");
    }
}

```

定义子类：

```

package com.atguigu.interfacetype;

public class Son extends Father implements Friend {
    @Override
    public void date() {

```

```
//(1)不重写默认保留父类的  
//(2)调用父类被重写的  
//    super.date();  
//(3)保留父接口的  
//    Friend.super.date();  
//(4)完全重写  
System.out.println("学Java");  
}  
}
```

定义测试类：

```
package com.atguigu.interfacetype;  
  
public class TestSon {  
    public static void main(String[] args) {  
        Son s = new Son();  
        s.date();  
    }  
}
```

(2) 左右为难

- 当一个类同时实现了多个父接口，而多个父接口中包含方法签名相同的默认方法时，怎么办呢？



无论你多难抉择，最终都是要做出选择的。

声明接口：

```
package com.atguigu.interfacetype;  
  
public interface BoyFriend {  
    default void date(){//约会  
        System.out.println("神秘约会");  
    }  
}
```

选择保留其中一个，通过“接口名.super.方法名”的方法选择保留哪个接口的默认方法。

```
package com.atguigu.interfacetype;

public class Girl implements Friend,BoyFriend{

    @Override
    public void date() {
        //1)保留其中一个父接口的
//        Friend.super.date();
//        BoyFriend.super.date();
        //2)完全重写
        System.out.println("学Java");
    }

}
```

测试类

```
package com.atguigu.interfacetype;

public class TestGirl {
    public static void main(String[] args) {
        Girl girl = new Girl();
        girl.date();
    }
}
```

- 当一个子接口同时继承了多个接口，而多个父接口中包含方法签名相同的默认方法时，怎么办呢？

另一个父接口：

```
package com.atguigu.interfacetype;

public interface Usb2 {
    //静态常量
    long MAX_SPEED = 60*1024*1024;//60MB/s

    //抽象方法
    void in();
    void out();

    //默认方法
    public default void start(){
        System.out.println("开始");
    }
    public default void stop(){
        System.out.println("结束");
    }

    //静态方法
    public static void show(){
```

```
        System.out.println("USB 2.0可以高速地进行读写操作");
    }
}
```

子接口：

```
package com.atguigu.interfacetype;

public interface Usb extends Usb2,Usb3 {
    @Override
    default void start() {
        System.out.println("Usb.start");
    }

    @Override
    default void stop() {
        System.out.println("Usb.stop");
    }
}
```

小贴士：

子接口重写默认方法时，`default`关键字可以保留。

子类重写默认方法时，`default`关键字不可以保留。

2、常量冲突问题

- 当子类继承父类又实现父接口，而父类中存在与父接口常量同名的成员变量，并且该成员变量在子类中仍然可见。
- 当子类同时继承多个父接口，而多个父接口存在相同同名常量。

此时在子类中想要引用父类或父接口的同名的常量或成员变量时，就会有冲突问题。

父类和父接口：

```
package com.atguigu.interfacetype;

public class SuperClass {
    int x = 1;
}
```

```
package com.atguigu.interfacetype;

public interface SuperInterface {
    int x = 2;
    int y = 2;
}
```

```
package com.atguigu.interfacetype;

public interface MotherInterface {
    int x = 3;
}
```

子类：

```
package com.atguigu.interfacetype;

public class SubClass extends superClass implements SuperInterface, MotherInterface {
    public void method(){
        // System.out.println("x = " + x); //模糊不清
        System.out.println("super.x = " + super.x);
        System.out.println("SuperInterface.x = " + SuperInterface.x);
        System.out.println("MotherInterface.x = " + MotherInterface.x);
        System.out.println("y = " + y); //没有重名问题，可以直接访问
    }
}
```

7.5.4 接口的特点总结

- 接口本身不能创建对象，只能创建接口的实现类对象，接口类型的变量可以与实现类对象构成多态引用。
- 声明接口用interface，接口的成员声明有限制：（1）公共的静态常量（2）公共的抽象方法（3）公共的默认方法（4）公共的静态方法（5）私有方法（JDK1.9以上）
- 类可以实现接口，关键字是implements，而且支持多实现。如果实现类不是抽象类，就必须实现接口中所有的抽象方法。如果实现类既要继承父类又要实现父接口，那么继承（extends）在前，实现（implements）在后。
- 接口可以继承接口，关键字是extends，而且支持多继承。
- 接口的默认方法可以选择重写或不重写。如果有冲突问题，另行处理。子类重写父接口的默认方法，要去掉default，子接口重写父接口的默认方法，不要去掉default。
- 接口的静态方法不能被继承，也不能被重写。接口的静态方法只能通过“接口名.静态方法名”进行调用。

7.5.5 经典接口介绍

1、java.lang.Comparable

我们知道基本数据类型的数据（除boolean类型外）需要比较大小的话，之间使用比较运算符即可，但是引用数据类型是不能直接使用比较运算符来比较大小的。那么，如何解决这个问题呢？

Java给所有引用数据类型的大小比较，指定了一个标准接口，就是java.lang.Comparable接口：

```
package java.lang;

public interface Comparable{
    int compareTo(Object obj);
}
```

那么我们想要使得我们某个类的对象可以比较大小，怎么做呢？步骤：

第一步：哪个类的对象要比较大小，哪个类就实现java.lang.Comparable接口，并重写方法

- 方法体就是你要如何比较当前对象和指定的另一个对象的大小

第二步：对象比较大小时，通过对象调用compareTo方法，根据方法的返回值决定谁大谁小。

- this对象（调用compareTo方法的对象）大于指定对象（传入compareTo()的参数对象）返回正整数
- this对象（调用compareTo方法的对象）小于指定对象（传入compareTo()的参数对象）返回负整数
- this对象（调用compareTo方法的对象）等于指定对象（传入compareTo()的参数对象）返回零

代码示例：

```
package com.atguigu.api;

public class Student implements Comparable {
    private int id;
    private String name;
    private int score;
    private int age;

    public Student(int id, String name, int score, int age) {
        this.id = id;
        this.name = name;
        this.score = score;
        this.age = age;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
        this.score = score;
    }

    public int getAge() {
```

```

        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", score=" + score +
            ", age=" + age +
            '}';
    }

    @Override
    public int compareTo(Object o) {
        //这些需要强制，将o对象向下转型为Student类型的变量，才能调用Student类中的属性
        //默认按照学号比较大小
        Student stu = (Student) o;
        return this.id - stu.id;
    }
}

```

测试类

```

package com.atguigu.api;

public class TestStudent {
    public static void main(String[] args) {
        Student[] arr = new Student[5];
        arr[0] = new Student(3,"张三",90,23);
        arr[1] = new Student(1,"熊大",100,22);
        arr[2] = new Student(5,"王五",75,25);
        arr[3] = new Student(4,"李四",85,24);
        arr[4] = new Student(2,"熊二",85,18);

        //单独比较两个对象
        System.out.println(arr[0].compareTo(arr[1]));
        System.out.println(arr[1].compareTo(arr[2]));
        System.out.println(arr[2].compareTo(arr[2]));

        System.out.println("所有学生: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
        System.out.println("按照学号排序: ");
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length-i; j++) {
                if(arr[j].compareTo(arr[j+1])>0){
                    Student temp = arr[j];

```

```
        arr[j] = arr[j+1];
        arr[j+1] = temp;
    }
}
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
}
```

2. `java.util.Comparator`

思考：

- (1) 如果一个类，没有实现Comparable接口，而这个类你又不方便修改（例如：一些第三方的类，你只有.class文件，没有源文件），那么这样类的对象也要比较大小怎么办？

- (2) 如果一个类，实现了Comparable接口，也指定了两个对象的比较大小的规则，但是此时此刻我不想按照它预定义的方法比较大小，但是我又不能随意修改，因为会影响其他地方的使用，怎么办？

JDK在设计类库之初，也考虑到这种情况了，所以又增加了一个java.util.Comparator接口。

```
package java.util;

public interface Comparator{
    int compare(Object o1, Object o2);
}
```

那么我们想要比较某个类的两个对象的大小，怎么做呢？步骤：

第一步：编写一个类，我们称之为比较器类型，实现java.util.Comparator接口，并重写方法

- 方法体就是你要如何指定的两个对象的大小

第二步：比较大小时，通过比较器类型的对象调用compare()方法，将要比较大小的两个对象作为compare方法的实参传入，根据方法的返回值决定谁大谁小。

- o1对象大于o2返回正整数
 - o1对象小于o2返回负整数
 - o1对象等于o2返回零

代码示例：定义定制比较器类

```
package com.atguigu.api;

import java.util.Comparator;

public class StudentScoreComparator implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        Student s1 = (Student) o1;
        Student s2 = (Student) o2;
        int result = s1.getScore() - s2.getScore();
        return result != 0 ? result : s1.getId() - s2.getId();
    }
}
```

代码示例：测试类

```
package com.atguigu.api;

public class TestStudent {
    public static void main(String[] args) {
        Student[] arr = new Student[5];
        arr[0] = new Student(3,"张三",90,23);
        arr[1] = new Student(1,"熊大",100,22);
        arr[2] = new Student(5,"王五",75,25);
        arr[3] = new Student(4,"李四",85,24);
        arr[4] = new Student(2,"熊二",85,18);

        //单独比较两个对象
        System.out.println(arr[0].compareTo(arr[1]));
        System.out.println(arr[1].compareTo(arr[2]));
        System.out.println(arr[2].compareTo(arr[2]));

        System.out.println("所有学生: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
        System.out.println("按照学号排序: ");
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length-i; j++) {
                if(arr[j].compareTo(arr[j+1])>0){
                    Student temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }

        System.out.println("按照成绩排序");
        StudentScoreComparator sc = new StudentScoreComparator();
```

```
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length-i; j++) {
                if(sc.compare(arr[j],arr[j+1])>0){
                    Student temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}
```

3、java.lang.Cloneable

在java.lang.Object类中有一个方法：

```
protected Object clone() throws CloneNotSupportedException
```

所有类型都可以重写这个方法，它是获取一个对象的克隆体对象用的，就是造一个和当前对象各种属性值一模一样的对象。当然地址肯定不同。

我们在重写这个方法后时，调用super.clone()，发现报异常CloneNotSupportedException，因为我们没有实现java.lang.Cloneable接口。

```
class Teacher implements Cloneable{
    private int id;
    private String name;
    public Teacher(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
    public Teacher() {
        super();
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
```

```

        return "Teacher [id=" + id + ", name=" + name + "]";
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Teacher other = (Teacher) obj;
        if (id != other.id)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}

```

```

public class TestClonable {
    public static void main(String[] args) throws CloneNotSupportedException {
        Teacher src = new Teacher(1,"柴老师");
        Object clone = src.clone();
        System.out.println(clone);
        System.out.println(src == clone);
        System.out.println(src.equals(clone));
    }
}

```

4、java.lang.Iterable接口

从JDK1.5之后引入java.lang.Iterable接口。实现这个接口允许对象成为 "foreach" 语句的目标。java.lang.Iterable接口包含一个抽象方法：Iterator iterator()，实现Iterable接口就要实现这个抽象方法，而Java中的数组默认都是实现了这个接口的，不用程序员手动实现这个抽象方法。

foreach循环的语法格式：

```
for(元素类型 元素名 : 数组名){  
}  
//这里元素名就是一个临时变量,自己命名就可以
```

代码示例：

```
package com.atguigu.api;  
  
public class TestForeach {  
    public static void main(String[] args) {  
        int[] nums = {1,2,3,4,5};  
        for (int num : nums) {  
            System.out.println(num);  
        }  
        System.out.println("-----");  
        String[] names = {"张三", "李四", "王五"};  
        for (String name : names) {  
            System.out.println(name);  
        }  
    }  
}
```

7.6 内部类

7.6.1 概述

1、什么是内部类？

将一个类A定义在另一个类B里面，里面的那个类A就称为**内部类**，B则称为**外部类**。

2、为什么要声明内部类呢？

总的来说，遵循高内聚低耦合的面向对象开发总原则。便于代码维护和扩展。

具体来说，当一个事物的内部，还有一个部分需要一个完整的结构进行描述，而这个内部的完整的结构又只为外部事物提供服务，不在其他地方单独使用，那么整个内部的完整结构最好使用内部类。而且内部类因为在外部类的里面，因此可以直接访问外部类的私有成员。

3、内部类都有哪些形式？

根据内部类声明的位置（如同变量的分类），我们可以分为：

(1) 成员内部类：

- 静态成员内部类
- 非静态成员内部类

(2) 局部内部类

- 有名字的局部内部类
- 匿名的内部类

7.6.2 成员内部类

如果成员内部类中不使用外部类的非静态成员，那么通常将内部类声明为静态内部类，否则声明为非静态内部类。

语法格式：

```
【修饰符】 class 外部类{  
    【其他修饰符】 【static】 class 内部类{  
    }  
}
```

1、静态内部类

有static修饰的成员内部类叫做静态内部类。它的特点：

- 和其他类一样，它只是定义在外部类中的另一个完整的类结构
 - 可以继承自己的想要继承的父类，实现自己想要实现的父接口们，和外部类的父类和父接口无关
 - 可以在静态内部类中声明属性、方法、构造器等结构，包括静态成员
 - 可以使用abstract修饰，因此它也可以被其他类继承
 - 可以使用final修饰，表示不能被继承
 - 编译后有自己的独立的字节码文件，只不过在内部类名前面冠以外部类名和\$符号。
- 和外部类不同的是，它可以允许四种权限修饰符：public, protected, 缺省, private
 - 外部类只允许public或缺省的
- 只可以在静态内部类中使用外部类的**静态成员**
 - 在静态内部类中不能使用外部类的非静态成员哦
 - 如果在内部类中有变量与外部类的静态成员变量同名，可以使用“外部类名.”进行区别
- 在外部类的外面不需要通过外部类的对象就可以创建静态内部类的对象（通常应该避免这样使用）

其实严格的讲（在James Gosling等人编著的《The Java Language Specification》）静态内部类不是内部类，而是类似于C++的嵌套类的概念，外部类仅仅是静态内部类的一种命名空间的限定名形式而已。所以接口中的内部类通常都不叫内部类，因为接口中的内部成员都是隐式是静态的（即public static）。例如：Map.Entry。

2、非静态成员内部类

没有static修饰的成员内部类叫做非静态内部类。非静态内部类的特点：

- 和其他类一样，它只是定义在外部类中的另一个完整的类结构
 - 可以继承自己的想要继承的父类，实现自己想要实现的父接口们，和外部类的父类和父接口无关
 - 可以在非静态内部类中声明属性、方法、构造器等结构，但是**不允许声明静态成员**，但是可以**继承父类的静态成员，而且可以声明静态常量**。
 - 可以使用abstract修饰，因此它也可以被其他类继承
 - 可以使用final修饰，表示不能被继承
 - 编译后有自己的独立的字节码文件，只不过在内部类名前面冠以外部类名和\$符号。
- 和外部类不同的是，它可以允许四种权限修饰符：public, protected, 缺省, private
 - 外部类只允许public或缺省的
 - 还可以在非静态内部类中使用外部类的**所有成员**，哪怕是私有的

- 在外部类的静态成员中不可以使用非静态内部类哦
 - 就如同静态方法中不能访问本类的非静态成员变量和非静态方法一样
- 在外部类的外面必须通过外部类的对象才能创建非静态内部类的对象（通常应该避免这样使用）
 - 如果要在外部类的外面使用非静态内部类的对象，通常在外部类中提供一个方法来返回这个非静态内部类的对象比较合适
 - 因此在非静态内部类的方法中有两个this对象，一个是外部类的this对象，一个是内部类的this对象

```

package com.atguigu.inner.member;

public class TestMemberInnerClass {
    public static void main(String[] args) {
        Outer.outMethod();
        System.out.println("-----");
        Outer out = new Outer();
        out.outFun();

        System.out.println("#####");
        Outer.Inner.inMethod();
        System.out.println("-----");
        Outer.Inner inner = new Outer.Inner();
        inner.inFun();

        System.out.println("#####");
        Outer outer = new Outer();
        //    Outer.Nei nei = outer.new Nei();
        Outer.Nei nei = out.getNei();
        nei.inFun();
    }
}

class Outer{
    private static String a = "外部类的静态a";
    private static String b = "外部类的静态b";
    private String c = "外部类对象的非静态c";
    private String d = "外部类对象的非静态d";

    static class Inner{
        private static String a ="静态内部类的静态a";
        private String c = "静态内部类对象的非静态c";
        public static void inMethod(){
            System.out.println("Inner.inMethod");
            System.out.println("Outer.a = " + Outer.a);
            System.out.println("Inner.a = " + a);
            System.out.println("b = " + b);
        }
        //    System.out.println("c = " + c); //不能访问外部类和自己的非静态成员
        //    System.out.println("d = " + d); //不能访问外部类的非静态成员
    }

    public void inFun(){
        System.out.println("Inner.inFun");
        System.out.println("Outer.a = " + Outer.a);
        System.out.println("Inner.a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}

```

```
//           System.out.println("d = " + d);//不能访问外部类的非静态成员
        }
    }

class Nei{
    private String a = "非静态内部类对象的非静态a";
    private String c = "非静态内部类对象的非静态c";

    public void inFun(){
        System.out.println("Nei.inFun");
        System.out.println("Outer.a = " + Outer.a);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("Outer.c = " + outer.this.c);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}

public static void outMethod(){
    System.out.println("Outer.outMethod");
    System.out.println("a = " + a);
    System.out.println("Inner.a = " + Inner.a);
    System.out.println("b = " + b);
//    System.out.println("c = " + c);
//    System.out.println("d = " + d);
    Inner in = new Inner();
    System.out.println("in.c = " + in.c);
}

public void outFun(){
    System.out.println("Outer.outFun");
    System.out.println("a = " + a);
    System.out.println("Inner.a = " + Inner.a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    Inner in = new Inner();
    System.out.println("in.c = " + in.c);
}

public Nei getNei(){
    return new Nei();
}
}
```

| | | 静态内部类 | 非静态内部类 |
|------|--------------------|--|-----------------------|
| 类角色 | 字节码文件 | 外部类名\$内部类名 | 相同 |
| | 修饰符 | public, 缺省, abstract, final | 相同 |
| | 父类或父接口 | 可以 | 相同 |
| | 可以包含的成员 | 所有成员 | ==不允许有静态成员== |
| 成员角色 | 修饰符 | public、protected、缺省、private, final, static | 没有static |
| | 依赖于外部类 | 依赖 | 相同 |
| | 依赖于外部类的对象 | 不依赖 | ==依赖== |
| 使用 | 在外部类中使用内部类 | 没有限制 | 在外部类的静态方法等中不能使用非静态内部类 |
| | 在内部类中使用外部类 | 静态内部类中不能使用外部类的非静态成员 | 没有限制 |
| | 在外部类的外面使用内部类的静态成员 | 外部类名.静态内部类名.静态成员 | ==没有== |
| | 在外部类的外面使用内部类的非静态成员 | 见下面的框1 | 见下面的框2 |
| 重名 | | 外部类名.重名的成员名 | 外部类名.this.重名的成员 |

外部类名.静态内部类名 变量 = 外部类名.静态内部类名();
 变量.非静态成员();

外部类名 变量1 = new 外部类();
 外部类名.非静态内部类名 变量 = 变量1.new 非静态内部类名();
 变量.非静态成员();

7.6.4 局部内部类

1、局部内部类

语法格式：

```
【修饰符】 class 外部类{
    【修饰符】 返回值类型 方法名(【形参列表】){
        【final/abstract】 class 内部类{
        }
    }
}
```

局部内部类的特点：

- 和外部类一样，它只是定义在外部类的某个方法中的另一个完整的类结构
 - 可以继承自己的想要继承的父类，实现自己想要实现的父接口们，和外部类的父类和父接口无关
 - 可以在局部内部类中声明属性、方法、构造器等结构，**但不包括静态成员，除非是从父类继承的或静态常量**
 - 可以使用abstract修饰，因此它也可以被同一个方法的在它后面的其他内部类继承
 - 可以使用final修饰，表示不能被继承
 - 编译后有自己的独立的字节码文件，只不过在内部类名前面冠以外部类名、\$符号、编号。
 - 这里有编号是因为同一个外部类中，不同的方法中存在相同名称的局部内部类
- 和成员内部类不同的是，它前面不能有权限修饰符等
- 局部内部类如同局部变量一样，有作用域
- 局部内部类中是否能访问外部类的非静态的成员，取决于所在的方法
- 局部内部类中还可以使用所在方法的局部常量，即用final声明的局部变量
 - JDK1.8之后，如果某个局部变量在局部内部类中被使用了，自动加final
 - 为什么在局部内部类中使用外部类方法的局部变量要加final呢？考虑生命周期问题。

示例代码：

```
package com.atguigu.inner.local;

public class TestLocalInner {
    public static void main(String[] args) {
        Runner runner = Outer.getRunner();
        runner.run();

        System.out.println("-----");
        Outer.outMethod();

        System.out.println("-----");
        Outer out = new Outer();
        out.outTest();
    }
}

class Outer{
    private static String a = "外部类的静态变量a";
    private String b = "外部类对象的非静态变量b";

    public static void outMethod(){
        System.out.println("Outer.outMethod");
        final String c = "局部变量c";
```

```

class Inner{
    public void inMethod(){
        System.out.println("Inner.inMethod");
        System.out.println("out.a = " + a);
        System.out.println("out.b = " + b); //错误的，因为outMethod是静态的
        System.out.println("out.local.c = " + c);
    }
}

Inner in = new Inner();
in.inMethod();

public void outTest(){
    class Inner{
        public void inMethod(){
            System.out.println("out.a = " + a);
            System.out.println("out.b = " + b); //可以，因为outTest是非静态的
        }
    }

    Inner in = new Inner();
    in.inMethod();
}

public static Runner getRunner(){
    class LocalRunner implements Runner{
        @Override
        public void run() {
            System.out.println("LocalRunner.run");
        }
    }
    return new LocalRunner();
}

}

interface Runner{
    void run();
}

```

2、匿名内部类

当我们在开发过程中，需要用到一个抽象类的子类的对象或一个接口的实现类的对象，而且只创建一个对象，而且逻辑代码也不复杂。那么我们原先怎么做的呢？

- (1) 编写类，继承这个父类或实现这个接口
- (2) 重写父类或父接口的方法
- (3) 创建这个子类或实现类的对象

这里，因为考虑到这个子类或实现类是一次性的，那么我们“费尽心机”的给它取名字，就显得多余。那么我们完全可以使用匿名内部类的方式来实现，避免给类命名的问题。

```
new 父类(【实参列表】){  
    重写方法...  
}  
//()中是否需要【实参列表】，看你想要让这个匿名内部类调用父类的那个构造器，如果调用父类的无参构造，那么()中就不用写参数，如果调用父类的有参构造，那么()中需要传入实参
```

```
new 父接口(){  
    重写方法...  
}  
//()中没有参数，因为此时匿名内部类的父类是Object类，它只有一个无参构造
```

匿名内部类是没有名字的类，因此在声明类的同时就创建好了唯一的对象。

注意：

匿名内部类是一种特殊的局部内部类，只不过没有名称而已。所有局部内部类的限制都适用于匿名内部类。例如：

- 在匿名内部类中是否可以使用外部类的非静态成员变量，看所在方法是否静态
- 在匿名内部类中如果需要访问当前方法的局部变量，该局部变量需要加final

思考：这个对象能做什么呢？

(1) 使用匿名内部类的对象直接调用方法

```
interface A{  
    void a();  
}  
public class Test{  
    public static void main(String[] args){  
        new A(){  
            @Override  
            public void a() {  
                System.out.println("aaaa");  
            }  
        }.a();  
    }  
}
```

(2) 通过父类或父接口的变量多态引用匿名内部类的对象

```
interface A{  
    void a();  
}  
public class Test{  
    public static void main(String[] args){  
        A obj = new A(){  
            @Override  
            public void a() {  
                System.out.println("aaaa");  
            }  
        };  
        obj.a();  
    }  
}
```

```
    }  
}
```

(3) 匿名内部类的对象作为实参

```
interface A{  
    void method();  
}  
public class Test{  
    public static void test(A a){  
        a.method();  
    }  
  
    public static void main(String[] args){  
        test(new A(){  
  
            @Override  
            public void method(){  
                System.out.println("aaaa");  
            }  
        });  
    }  
}
```

7.7 注解

7.7.1 什么是注解

注解是以“@注释名”在代码中存在的，还可以添加一些参数值，例如：

```
@SuppressWarnings(value="unchecked")  
@Override  
@Deprecated
```

注解Annotation是从JDK5.0开始引入。

虽然说注解也是一种注释，因为它们都不会改变程序原有的逻辑，只是对程序增加了某些注释性信息。不过它又不同于单行注释和多行注释，对于单行注释和多行注释是给程序员看的，而注解是可以被编译器或其他程序读取的一种注释，程序还可以根据注解的不同，做出相应的处理。所以注解是插入到代码中以便有工具可以对它们进行处理的标签。

7.7.2 三个最基本的注解

1、@Override

用于检测被标记的方法为有效的重写方法，如果不是，则报编译错误！

只能标记在方法上。

它会被编译器程序读取。

2、@Deprecated

用于表示被标记的数据已经过时，不建议使用。

可以用于修饰属性、方法、构造、类、包、局部变量、参数。

它会被编译器程序读取。

3、@SuppressWarnings

抑制编译警告。

可以用于修饰类、属性、方法、构造、局部变量、参数

它会被编译器程序读取。

示例代码：

```
package com.atguigu.annotation;

import java.util.ArrayList;

public class TestAnnotation {
    @SuppressWarnings("all")
    public static void main(String[] args) {
        int i;

        ArrayList list = new ArrayList();
        list.add("hello");
        list.add(123);
        list.add("world");

        Father f = new Son();
        f.show();
        f.method01();
    }
}

class Father{
    @Deprecated
    void show() {
        System.out.println("Father.show");
    }
    void method01() {
        System.out.println("Father Method");
    }
}

class Son extends Father{
/*    @Override
    void method01() {
        System.out.println("Son Method");
    }*/
}
```

7.7.3 JUnit

JUnit是由 Erich Gamma 和 Kent Beck 编写的一个回归测试框架 (regression testing framework), 供Java开发人员编写单元测试之用。多数Java的开发环境都已经集成了JUnit作为单元测试的工具。JUnit测试是程序员测试, 即所谓白盒测试, 因为程序员知道被测试的软件如何 (How) 完成功能和完成什么样 (What) 的功能。

要使用JUnit, 必须在项目的编译路径中必须引入JUnit的库, 即相关的.class文件组成的jar包。如何把JUnit的jar添加到编译路径如图所示:

后面会学习maven, 在maven仓库中统一管理所有第三方框架和工具组件的jar, 但是现在没有学习maven之前, 可以使用本地jar包。

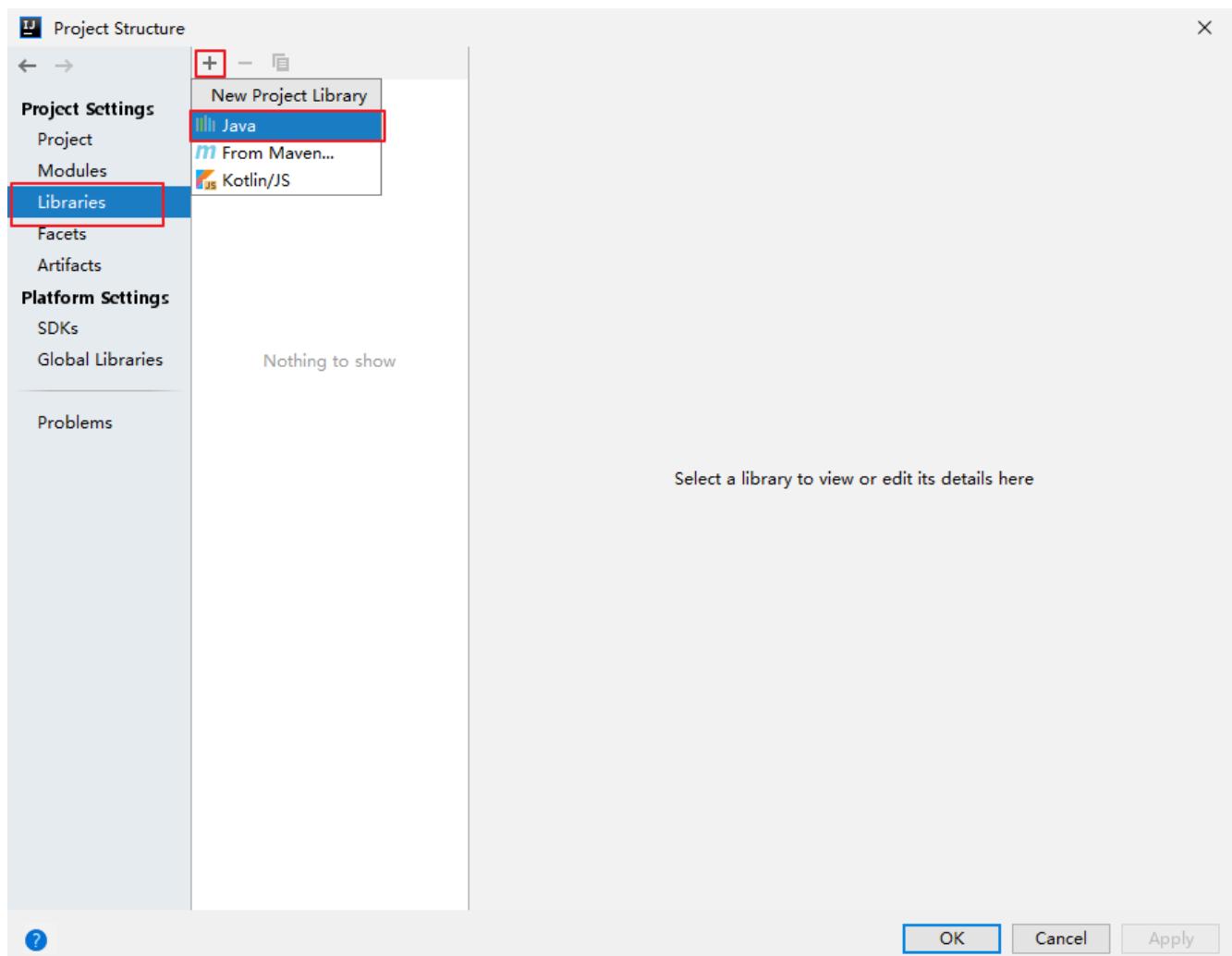
1、引入本地JUnitjar

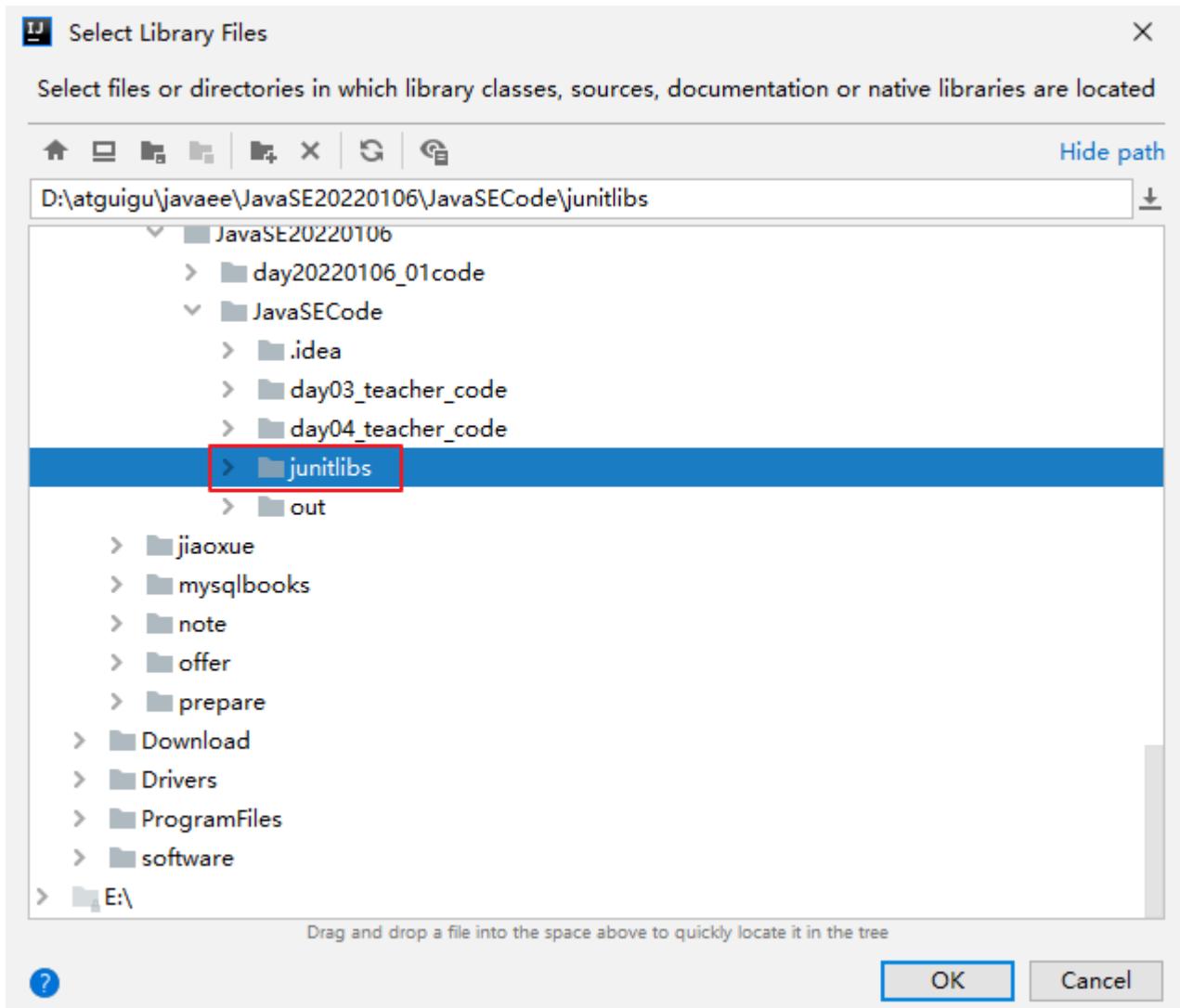
第一步: 在当前IDEA项目目录下建立junitlibs, 把下载的JUnit的相关jar包放进去:

Data (D:) > atguigu > javaee > JavaSE20220106 > JavaSECode > junitlibs

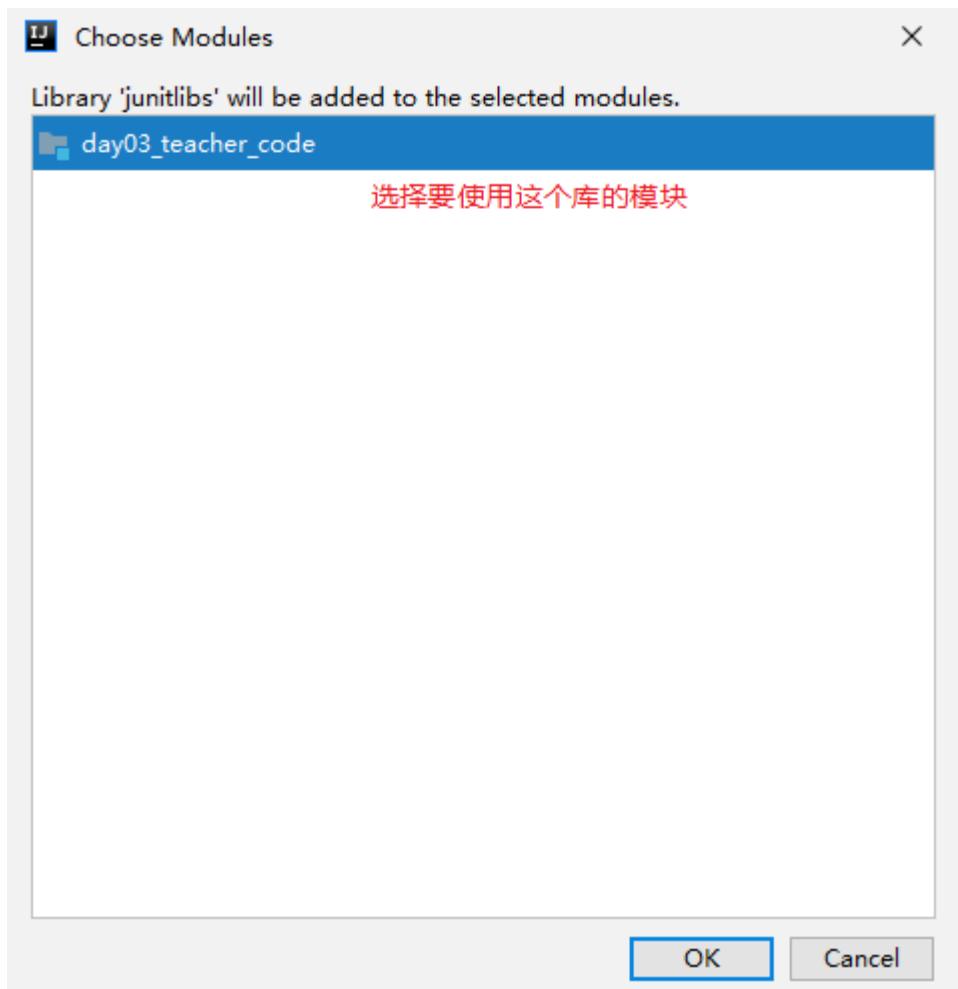
| 名称 | 修改日期 | 类型 | 大小 |
|-----------------------|------------------|---------------------|--------|
| hamcrest-core-1.3.jar | 2019/12/17 19:38 | Executable Jar File | 44 KB |
| junit-4.12.jar | 2019/12/17 19:38 | Executable Jar File | 308 KB |

第二步: 在项目中添加Libraries库

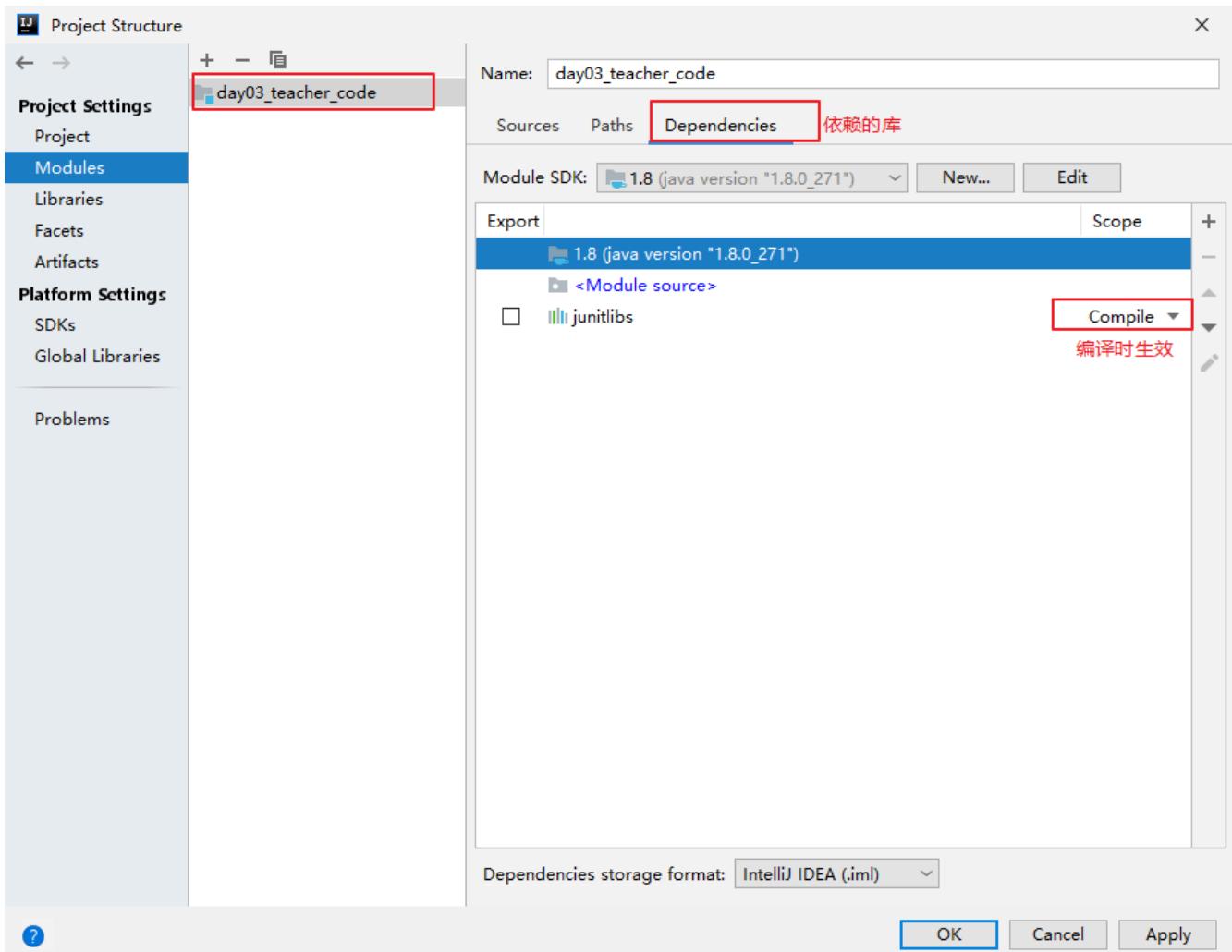




第三步：选择要在哪些module中应用JUnit库

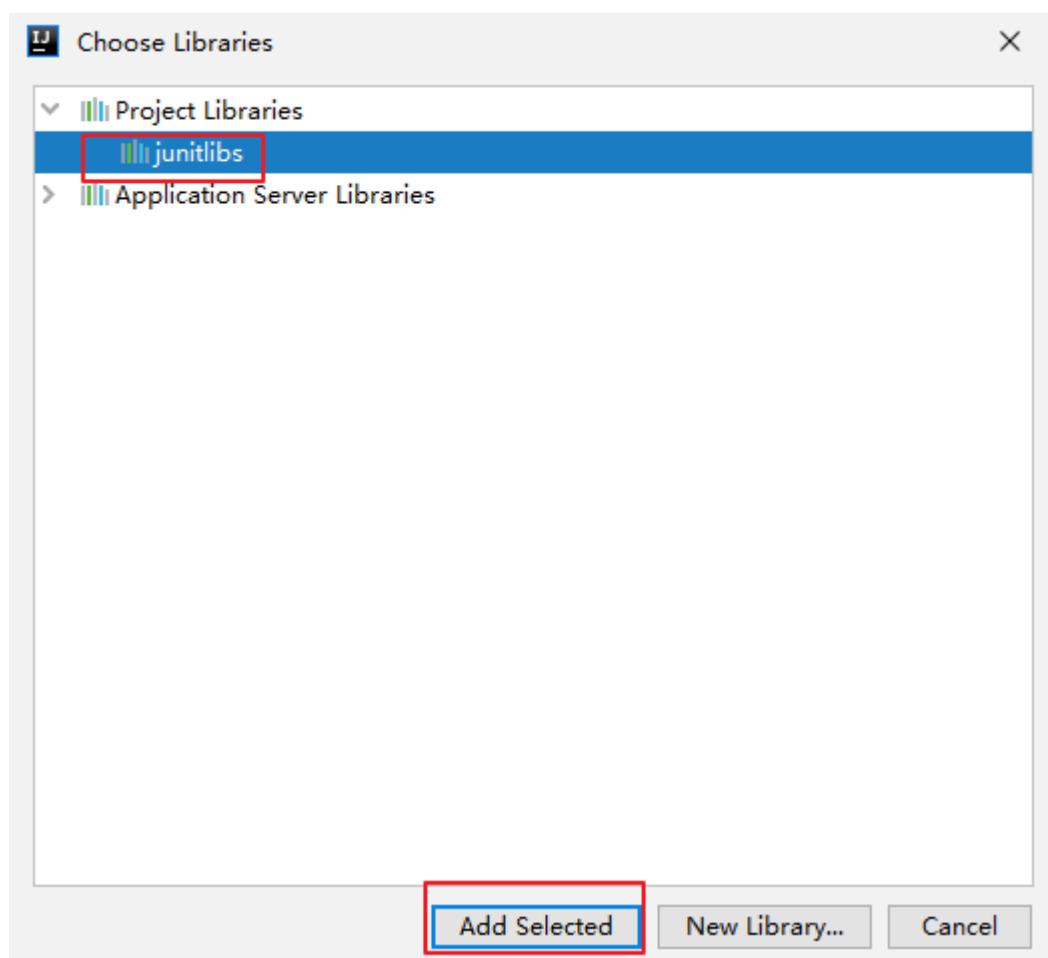
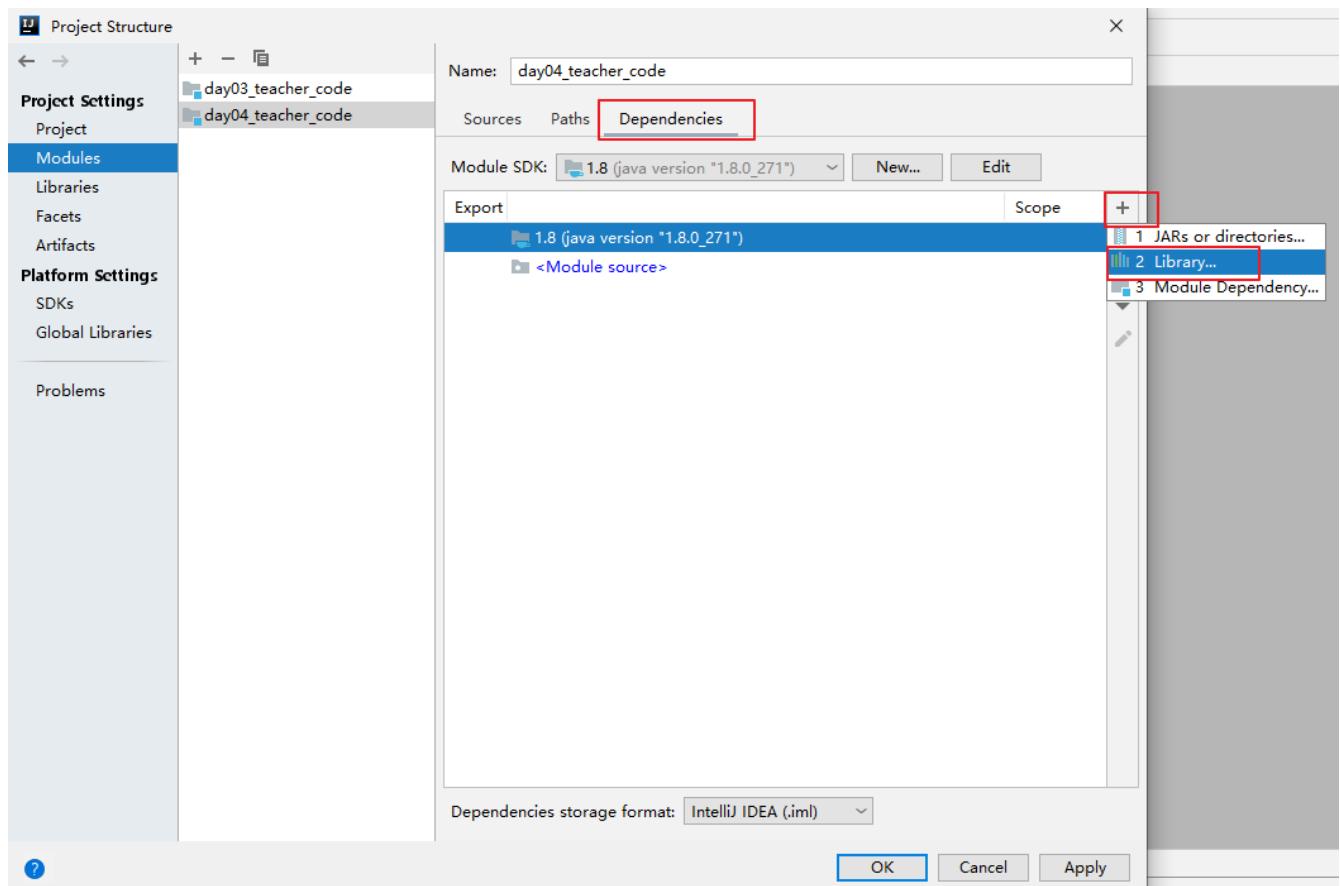


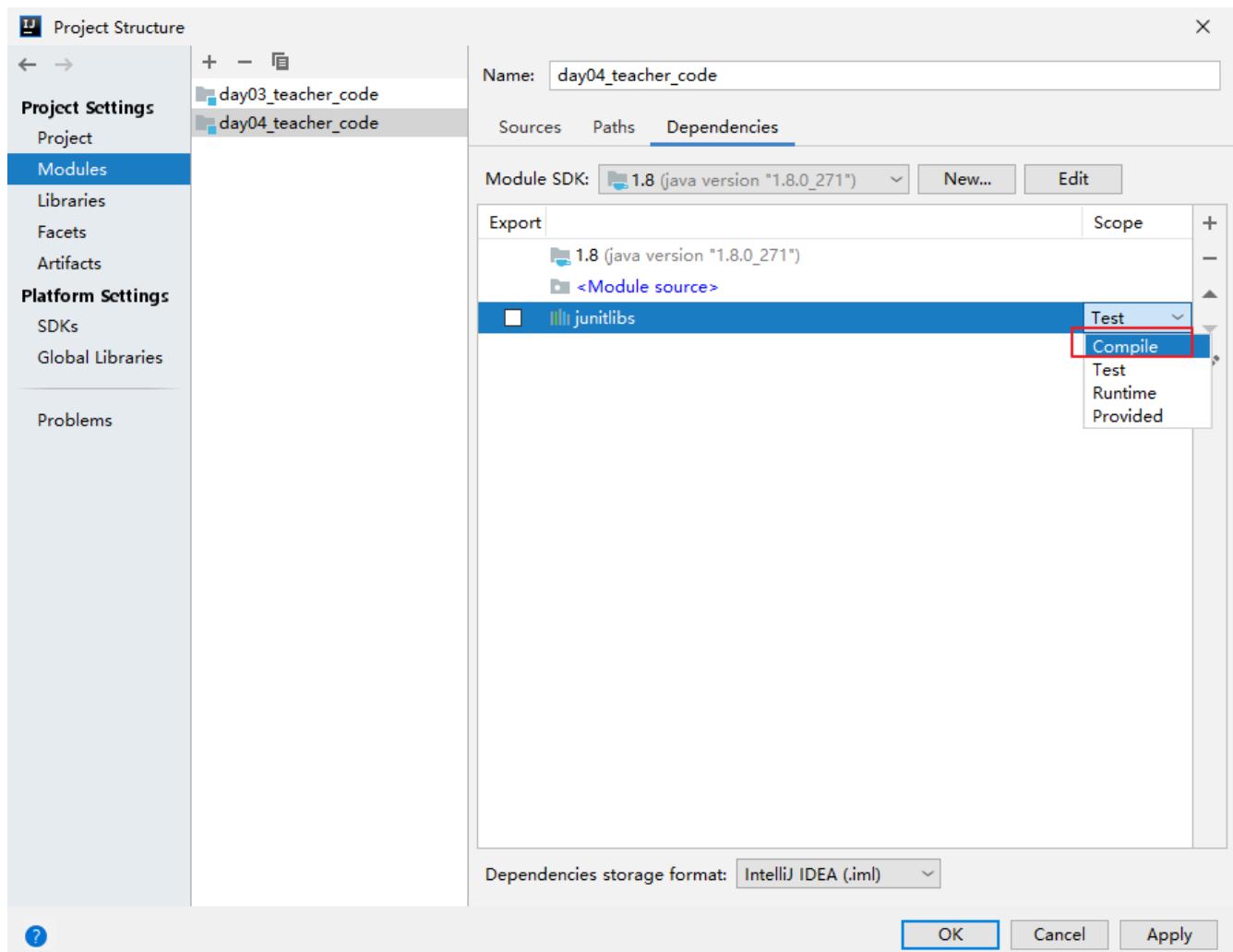
第四步：检查是否应用成功

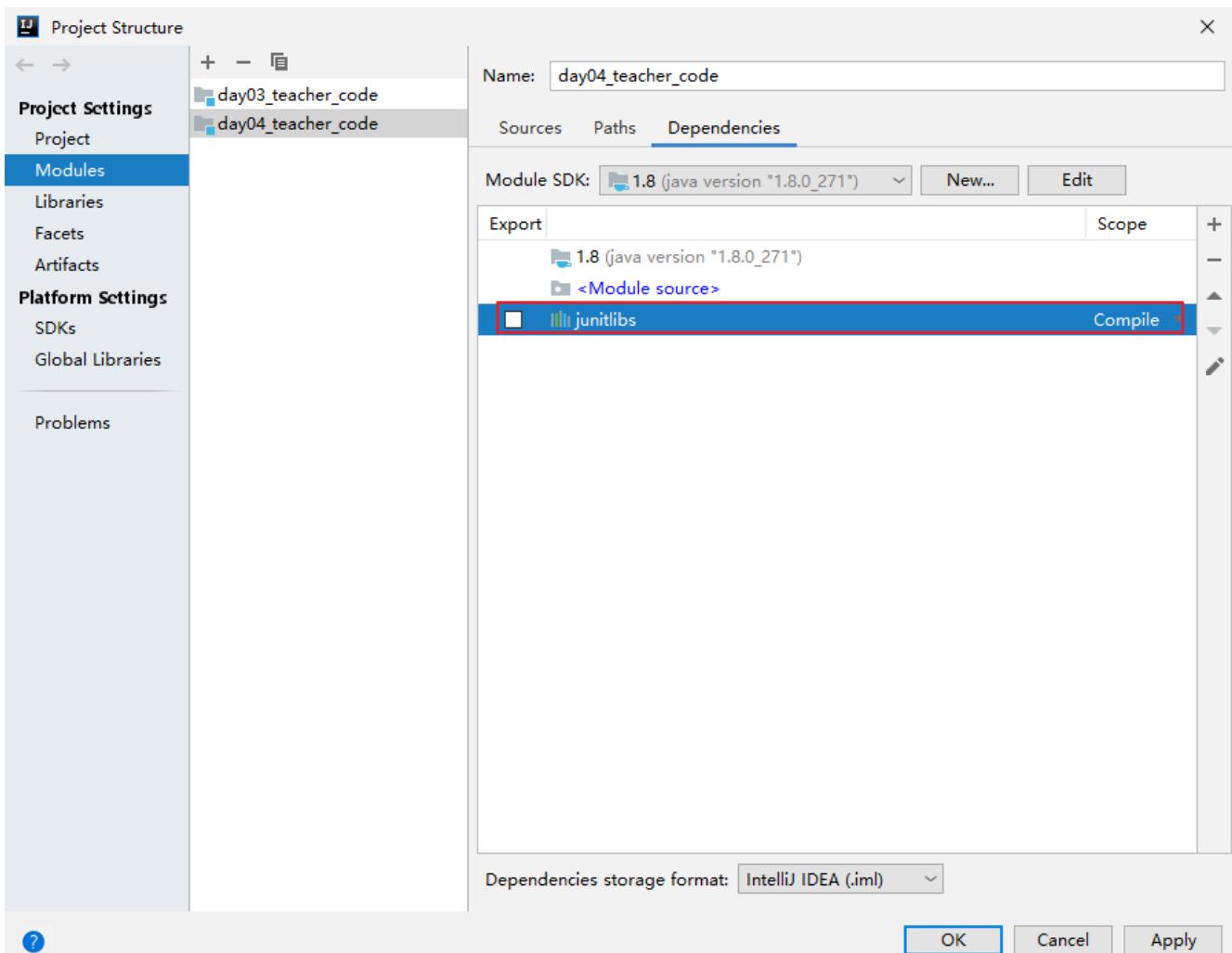


注意Scope：选择Complie，否则编译时，无法使用JUnit。

第5步：下次如果有新的模块要使用该libs库，这样操作即可







2、编写和运行@Test单元测试方法

JUnit4版本，要求@Test标记的方法必须满足如下要求：

- 所在的类必须是public的，非抽象的，包含唯一的无参构造的。
- @Test标记的方法本身必须是public，非抽象，非静态的，void无返回值，()无参数的。

```
package com.atguigu.junit;

import org.junit.Test;

public class TestJUnit {
    @Test
    public void test01(){
        System.out.println("TestJUnit.test01");
    }

    @Test
    public void test02(){
        System.out.println("TestJUnit.test02");
    }

    @Test
    public void test03(){
    }
}
```

```
        System.out.println("TestJUnit.test03");
    }
}
```

The screenshot shows the IntelliJ IDEA interface. The top part displays the Java code for `TestJUnit`. The bottom part shows the `Run` tool window with three configurations:

- Configuration 1 (highlighted with a red circle): `Run` button, `Test` scope, `test01` method. A tooltip says "运行本类所有单元测试方法".
- Configuration 2: `Run` button, `All` scope, `test02` method.
- Configuration 3: `Run` button, `Test` scope, `test03` method. A tooltip says "单独运行某个单元测试方法".

3、设置执行JUnit用例时支持控制台输入

在idea64.exe.vmoptions配置文件中加入下面一行设置，重启idea后生效。

需要注意的是，要看你当前IDEA读取的是哪个idea64.exe.vmoptions配置文件文件。如果在C盘的用户目录的config下（例如：C:\Users\Irene\IntelliJIdea2019.2\config）也有一个idea64.exe.vmoptions文件，那么将优先使用C盘用户目录下的。否则用的是IDEA安装目录的bin目录（例如：

D:\ProgramFiles\JetBrains\IntelliJ_IDEA_2019.2.3\bin）下的idea64.exe.vmoptions文件。

```
-Deditable.java.test.console=true
```

The screenshot shows the `idea64.exe.vmoptions` configuration file in IntelliJ IDEA. The file contains the following settings:

```
-Djava.net.preferIPv4Stack=true
-Djdk.http.auth.tunneling.disabledSchemes=""
-XX:+HeapDumpOnOutOfMemoryError
-XX:-OmitStackTraceInFastThrow
-Djdk.attach.allowAttachSelf
-Dkotlinx.coroutines.debug=off
-Djdk.module.illegalAccess.silent=true
-Deditable.java.test.console=true
```

8.1 异常概述

8.1.1 认识Java的异常

1、什么是异常

在使用计算机语言进行项目开发的过程中，即使程序员把代码写得尽善尽美，在系统的运行过程中仍然会遇到一些问题，因为很多问题不是靠代码能够避免的，比如：客户输入数据的格式问题，读取文件是否存在，网络是否始终保持通畅等等。

- **异常**：指的是程序在执行过程中，出现的非正常的情况，如果不处理最终会导致JVM的非正常停止。

异常指的并不是语法错误，语法错了，编译不通过，不会产生字节码文件，根本不能运行。

异常也不是指逻辑代码错误而没有得到想要的结果，例如：求a与b的和，你写成了a-b

2、如何对待异常

程序员在编写程序时，就应该充分考虑到各种可能发生的异常和错误，极力预防和避免，实在无法避免的，要编写相应的代码进行异常的检测、异常消息的提示，以及异常的处理。

3、异常的抛出机制

Java中是如何表示不同的异常情况，又是如何让程序员得知，并处理异常呢？

Java中把不同的异常用不同的类表示，一旦发生某种异常，就通过创建该异常类型的对象，并且抛出，然后程序员可以catch到这个异常对象，并处理，如果无法catch到这个异常对象，那么这个异常对象将会导致程序终止。

运行下面的程序，程序会产生一个数组索引越界异常`ArrayIndexOutOfBoundsException`。我们通过图解来解析下异常产生和抛出的过程。

工具类

```
public class ArrayTools {  
    // 对给定的数组通过给定的角标获取元素。  
    public static int getElement(int[] arr, int index) {  
        int element = arr[index];  
        return element;  
    }  
}
```

测试类

```
public class ExceptionDemo {  
    public static void main(String[] args) {  
        int[] arr = { 34, 12, 67 };  
        intnum = ArrayTools.getElement(arr, 4)  
        System.out.println("num=" + num);  
        System.out.println("over");  
    }  
}
```

上述程序执行过程图解：

```
由于没有找到4索引，导致运行时发生了异常。这个异常JVM认识。ArrayIndexOutOfBoundsException  
这个异常Java本身有描述：描述内容包括：异常的名称、异常的内容、异常的产生位置。  
Java将这些信息直接封装到异常对象中。new ArrayIndexOutOfBoundsException(4);
```

```
class ArrayTools {  
    // 对给定的数组通过给定的角标获取元素。  
    public static int getElement(int[] arr, int index) {  
        int element = arr[index];  
        JVM throw new ArrayIndexOutOfBoundsException(4); 产生异常对象  
        return element;  
    }  
}  
  
class ExceptionDemo2 {  
    public static void main(String[] args) {  
        int[] arr = {34,12,67};  
        int num = ArrayTools.getElement(arr,4)  
        System.out.println("num="+num);  
        System.out.println("over");  
    }  
}
```

运行结果：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4  
        at day21_01.ArrayTools.getElement(ArrayTools.java:6)  
        at day21_01.ExceptionDemo2.main(ExceptionDemo2.java:6)
```

异常的抛出机制



如果代码执行过程中发生异常，没有相应的处理代码，就会导致程序崩溃，终止运行。

为保证程序的正常运行，必须充分考虑到可能发生的各种异常和错误，极力预防，实在无法避免的，就要提前编写对可能发生的异常进行处理的代码，保证代码程序的健壮性。

8.1.2 Java异常体系

1、Throwable

`java.lang.Throwable` 类是 Java 语言中所有错误或异常的超类。

- 只有当对象是此类（或其子类之一）的实例时，才能通过 Java 虚拟机或者 Java 的 `throw` 语句抛出。类似地，只有此类或其子类之一才可以是 `catch` 子句中的参数类型。

Throwable中的常用方法：

- `public void printStackTrace()`: 打印异常的详细信息。

包含了异常的类型，异常的原因，还包括异常出现的位置，在开发和调试阶段都得使用`printStackTrace`。

- `public String getMessage()`: 获取发生异常的原因。

提示给用户的时候就提示错误原因。

2、Error和Exception

`Throwable`有两个直接子类：`java.lang.Error`与`java.lang.Exception`，平常所说的异常指`java.lang.Exception`。

- **Error**: 表示严重错误，一旦发生必须停下来查看问题并解决问题才能继续，无法仅仅通过try...catch解决的错误。（如果拿生病做比喻，就像是突发疾病，而且是危重症，必须立刻停下来治疗而不是靠短暂休息、吃药、打针、或小手术简单解决处理）
 - 例如：`StackOverflowError`（栈内存溢出）和`OutOfMemoryError`（堆内存溢出，简称OOM）。
- **Exception**: 表示普通异常，其它因编程错误或偶然的外在因素导致的一般性问题，程序员可以通过代码的方式检测、提示和纠正，使程序继续运行，但是只要发生也是必须处理，否则程序也会挂掉。（这就好比普通感冒、阑尾炎、牙疼等，可以通过短暂休息、吃药、打针、或小手术简单解决，但是也不能搁置不处理，不然也会要人命）。
 - 例如：空指针访问、试图读取不存在的文件、网络连接中断、数组下标越界等

无论是Error还是Exception，还有很多子类，异常的类型非常丰富。**当代码运行出现异常时，特别是我们不熟悉的异常时，不要紧张，把异常的简单类名，拷贝到API中去查去认识它即可。**

```

1 public class Demo {
2
3     public static void main(String[] args) {
4         // 定义一个数组
5         int[] arr = {3,4,56};          勇敢的面对异常！
6
7         System.out.println(arr[3]);
8     }
9 }
10 异常出现的位置

```

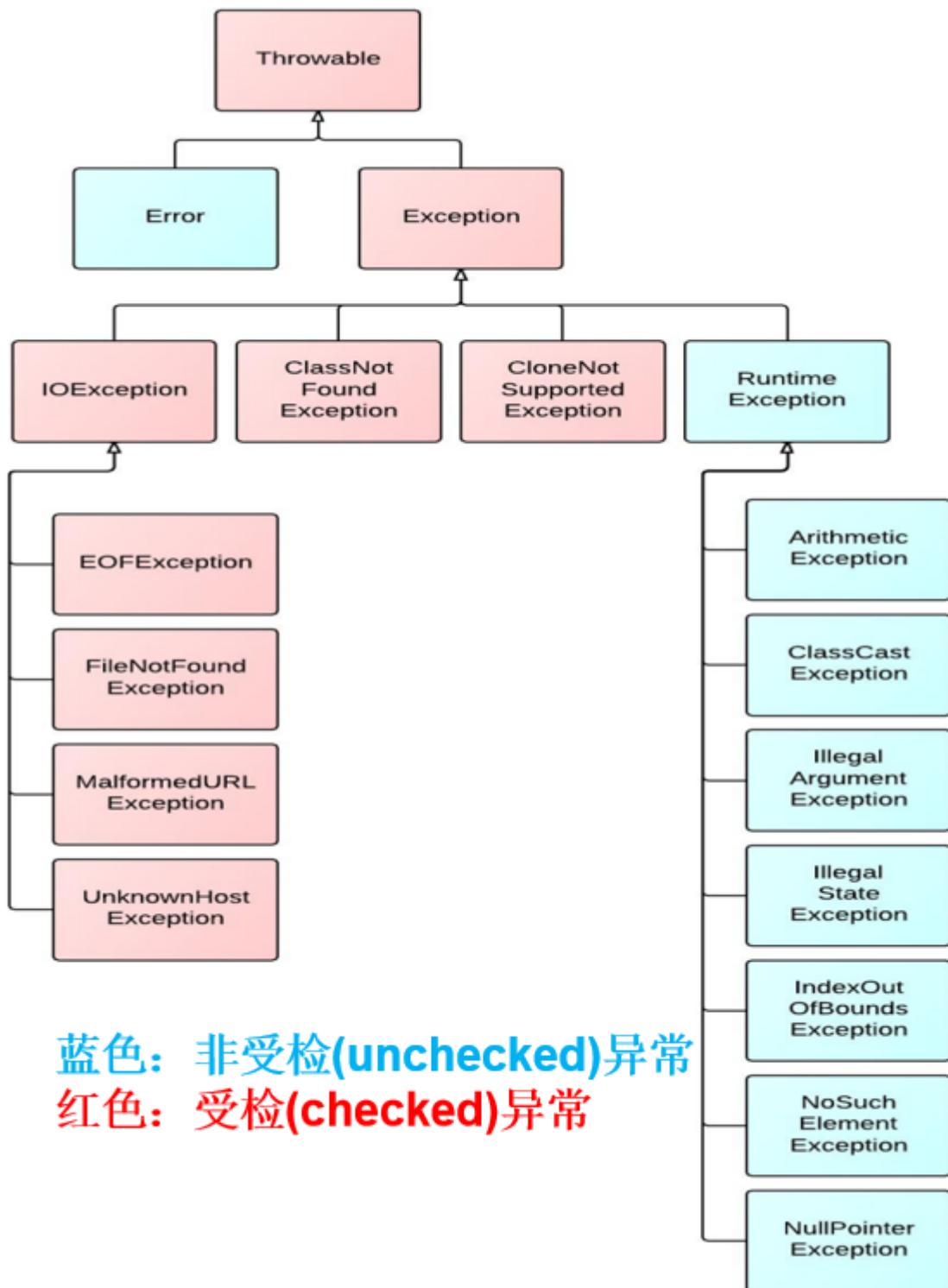
Demo

D:\develop\Java\jdk-9.0.1\bin\java "-javaagent:D:\develop\JetBrains\IntelliJ IDEA 2017.3.2\lib\idea_rt.jar" Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
 at Demo.main(Demo.java:7)
 原因
 异常的类型
 Process finished with exit code 1

8.1.3 受检异常和非受检异常

我们平常说的异常就是指Exception，根据代码的编写编译阶段，编译器是否会**警示**当前代码可能发生xx异常，并**督促**程序员提前编写处理它的代码为依据，可以将异常分为：

- **编译时期异常**（即checked异常、受检异常）：在代码编译阶段，编译器就能明确**警示**当前代码**可能发生（不是一定发生）**xx异常，并**督促**程序员提前编写处理它的代码。如果程序员**不听话**，没有编写对应的异常处理代码，则编译器就会**发威**，直接判定编译失败，从而程序无法执行。通常，这类异常的发生不是由程序员的代码引起的，或者不是靠加简单判断就可以避免的，例如：`FileNotFoundException`（文件找不到异常）。
- **运行时期异常**（即runtime异常、unchecked非受检异常）：即在代码编译阶段，编译器完全不做任何**检查**，无论该异常是否会发生，编译器都不给出任何提示。只有等代码运行起来并确实发生了xx异常，它才能被发现。通常，这类异常是由程序员的代码编写不当引起的，只要稍加判断，或者细心检查就可以避免的。例如：`ArrayIndexOutOfBoundsException`（数组下标越界异常），`ClassCastException`（类型转换异常）。



8.1.4 演示常见的错误和异常

1、Error

最常见的就是`VirtualMachineError`, 它有两个经典的子类: `StackOverflowError`、`OutOfMemoryError`。

```

package com.atguigu.exception;

import org.junit.Test;
  
```

```
public class TestStackoverflowError {
    @Test
    public void test01(){
        //StackOverflowError
        digui();
    }

    public void digui(){
        digui();
    }
}
```

```
package com.atguigu.exception;

import org.junit.Test;

public class TestOutOfMemoryError {
    @Test
    public void test02(){
        //OutOfMemoryError
        //方式一:
        int[] arr = new int[Integer.MAX_VALUE];
    }

    @Test
    public void test03(){
        //OutOfMemoryError
        //方式二:
        StringBuilder s = new StringBuilder();
        while(true){
            s.append("atguigu");
        }
    }
}
```

2、运行时异常

```
package com.atguigu.exception;

import org.junit.Test;

import java.util.Scanner;

public class TestRuntimeException {
    @Test
    public void test01(){
        //NullPointerException
        int[][] arr = new int[3][];
        System.out.println(arr[0].length);
    }
}
```

```

@Test
public void test02(){
    //ClassCastException
    Object obj = 15;
    String str = (String) obj;
}

@Test
public void test03(){
    //ArrayIndexOutOfBoundsException
    int[] arr = new int[5];
    for (int i = 1; i <= 5; i++) {
        System.out.println(arr[i]);
    }
}

@Test
public void test04(){
    //InputMismatchException
    Scanner input = new Scanner(System.in);
    System.out.print("请输入一个整数: ");//输入非整数
    int num = input.nextInt();
    input.close();
}

@Test
public void test05(){
    int a = 1;
    int b = 0;
    //ArithmaticException
    System.out.println(a/b);
}
}

```

3、编译时异常

```

package com.atguigu.exception;

import org.junit.Test;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestCheckedException {
    @Test
    public void test06() throws InterruptedException{
        Thread.sleep(1000);//休眠1秒
    }
}

```

```
@Test  
public void test07() throws FileNotFoundException {  
    FileInputStream fis = new FileInputStream("Java学习秘籍.txt");  
}  
  
@Test  
public void test08() throws SQLException {  
    Connection conn = DriverManager.getConnection("....");  
}  
}
```

8.2 异常的处理

Java异常处理的五个关键字：**try、catch、finally、throw、throws**

8.2.1 捕获异常：try...catch

1、try...catch基本格式

捕获异常语法如下：

```
try{  
    可能发生xx异常的代码  
}catch(异常类型1 e){  
    处理异常的代码1  
}catch(异常类型2 e){  
    处理异常的代码2  
}  
....
```

try{}中编写可能发生xx异常的业务逻辑代码。

catch分支，分为两个部分，catch()中编写异常类型和异常参数名，{}中编写如果发生了这个异常，要做什么处理的代码。如果有多个catch分支，并且多个异常类型有父子类关系，必须保证小的子异常类型在上，大的父异常类型在下。

当某段代码可能发生异常，不管这个异常是编译时异常（受检异常）还是运行时异常（非受检异常），我们都可以使用try块将它括起来，并在try块下面编写catch分支尝试捕获对应的异常对象。

- 如果在程序运行时，try块中的代码没有发生异常，那么catch所有的分支都不执行。
- 如果在程序运行时，try块中的代码发生了异常，根据异常对象的类型，将从上到下选择第一个匹配的catch分支执行。此时try中发生异常的语句下面的代码将不执行，而整个try...catch之后的代码可以继续运行。
- 如果在程序运行时，try块中的代码发生了异常，但是所有catch分支都无法匹配（捕获）这个异常，那么JVM将会终止当前方法的执行，并把异常对象“抛”给调用者。如果调用者不处理，程序就挂了。

示例代码：

```
package com.atguigu.keyword;  
  
public class TestTryCatch {  
    public static void main(String[] args) {  
        try {
```

```

        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        int result = a/b;
        System.out.println("result = " + result);
    } catch (NumberFormatException e) {
        System.out.println("数字格式不正确, 请输入两个整数");
    } catch (ArrayIndexOutOfBoundsException e){
        System.out.println("数字个数不正确, 请输入两个整数");
    } catch (ArithmetricException e){
        System.out.println("第二个整数不能为0");
    }

    System.out.println("你输入了" + args.length +"个参数。");
    System.out.println("你输入的被除数和除数分别是: ");
    for (int i = 0; i < args.length; i++) {
        System.out.print(args[i]+ " ");
    }
    System.out.println();
}
}

```

2、JDK1.7try...catch新特性

如果多个catch分支的异常处理代码一致，那么在JDK1.7之后还支持如下写法：

```

try{
    可能发生xx异常的代码
}catch(异常类型1 | 异常类型2 e){
    处理异常的代码1
}catch(异常类型3 e){
    处理异常的代码2
}
....

```

示例代码：

```

package com.atguigu.keyword;

public class TestJDK7 {
    public static void main(String[] args) {
        try {
            int a = Integer.parseInt(args[0]);
            int b = Integer.parseInt(args[1]);
            int result = a/b;
            System.out.println("result = " + result);
        } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
            System.out.println("数字格式不正确, 请输入两个整数");
        } catch (ArithmetricException e){
            System.out.println("第二个整数不能为0");
        }
    }
}

```

```
    System.out.println("你输入了" + args.length +"个参数。");
    System.out.println("你输入的被除数和除数分别是: ");
    for (int i = 0; i < args.length; i++) {
        System.out.print(args[i]+ " ");
    }
    System.out.println();
}
}
```

3、在catch分支中获取异常信息

如何获取异常信息，`Throwable`类中定义了一些查看方法：

- `public string getMessage()` :获取异常的描述信息,原因(提示给用户的时候,就提示错误原因)。
- `public void printStackTrace()` :打印异常的跟踪栈信息并输出到控制台。

包含了异常的类型,异常的原因,还包括异常出现的位置,在开发和调试阶段,都得使用`printStackTrace()`。

8.2.2 finally块

1、finally块

因为异常会引发程序跳转，从而会导致有些语句执行不到。而程序中有一些特定的代码无论异常是否发生，都需要执行。例如，IO流的关闭，数据库连接的断开等。这样的代码通常就会放到finally块中。

```
try{
}catch(...){
}finally{
    无论try中是否发生异常，也无论catch是否捕获异常，也不管try和catch中是否有return语句，都一定会执行
}

或

try{
}finally{
    无论try中是否发生异常，也不管try中是否有return语句，都一定会执行。
}
```

注意：finally不能单独使用。

当只有在try或者catch中调用退出VM的相关方法，例如`System.exit(0)`，此时finally才不会执行，否则finally永远会执行。

示例代码：

```
package com.atguigu.keyword;

import java.util.InputMismatchException;
import java.util.Scanner;
```

```

public class TestFinally {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        try {
            System.out.print("请输入第一个整数: ");
            int a = input.nextInt();
            System.out.print("请输入第二个整数: ");
            int b = input.nextInt();
            int result = a/b;
            System.out.println(a + "/" + b +"=" + result);
        } catch (InputMismatchException e) {
            System.out.println("数字格式不正确, 请输入两个整数");
        }catch (ArithmaticException e){
            System.out.println("第二个整数不能为0");
        } finally {
            System.out.println("程序结束, 释放资源");
            input.close();
        }
    }
}

```



2、finally与return

finally中写了return语句，那么try和catch中的return语句就失效了，最终返回的是finally块中的

形式一：从try回来

```
public class TestReturn {  
    public static void main(String[] args) {  
        int result = test("12");  
        System.out.println(result);  
    }  
  
    public static int test(String str){  
        try{  
            Integer.parseInt(str);  
            return 1;  
        }catch(NumberFormatException e){  
            return -1;  
        }finally{  
            System.out.println("test结束");  
        }  
    }  
}
```

形式二：从catch回来

```
public class TestReturn {  
    public static void main(String[] args) {  
        int result = test("a");  
        System.out.println(result);  
    }  
  
    public static int test(String str){  
        try{  
            Integer.parseInt(str);  
            return 1;  
        }catch(NumberFormatException e){  
            return -1;  
        }finally{  
            System.out.println("test结束");  
        }  
    }  
}
```

形式三：从finally回来

```
public class TestReturn {  
    public static void main(String[] args) {  
        int result = test("a");  
        System.out.println(result);  
    }  
  
    public static int test(String str){  
        try{  
            Integer.parseInt(str);  
            return 1;  
        }
```

```
        }catch(NumberFormatException e){
            return -1;
        }finally{
            System.out.println("test结束");
            return 0;
        }
    }
}
```

8.2.3 转换异常处理位置： throws

1、 throws编译时异常

如果在编写方法体的代码时，某句代码可能发生某个==编译时异常==，不处理编译不通过，但是在当前方法体中可能不适合处理或无法给出合理的处理方式，就可以通过throws在方法签名中声明该方法可能会发生xx异常，需要调用者处理。

声明异常格式：

```
修饰符 返回值类型 方法名(参数) throws 异常类名1,异常类名2...{ }
```

在throws后面可以写多个异常类型，用逗号隔开。

代码演示：

```
package com.atguigu.keyword;

public class TestThrowsCheckedException {
    public static void main(String[] args) {
        System.out.println("上课.....");
        try {
            afterClass();//换到这里处理异常
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.out.println("准备提前上课");
        }
        System.out.println("上课.....");
    }

    public static void afterClass() throws InterruptedException {
        for(int i=10; i>=1; i--){
            Thread.sleep(1000);//本来应该在这里处理异常
            System.out.println("距离上课还有：" + i + "分钟");
        }
    }
}
```

2、 throws运行时异常

当然，throws后面也可以写运行时异常类型，只是运行时异常类型，写或不写对于编译器和程序执行来说都没有任何区别。如果写了，唯一的区别就是调用者调用该方法后，使用try...catch结构时，IDEA可以获得更多的信息，需要添加什么catch分支。

```
package com.atguigu.keyword;

import java.util.InputMismatchException;
import java.util.Scanner;

public class TestThrowsRuntimeException {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        try {
            System.out.print("请输入第一个整数: ");
            int a = input.nextInt();
            System.out.print("请输入第二个整数: ");
            int b = input.nextInt();
            int result = divide(a,b);
            System.out.println(a + "/" + b + "=" + result);
        } catch (ArithmaticException | InputMismatchException e) {
            e.printStackTrace();
        } finally {
            input.close();
        }
    }

    public static int divide(int a, int b) throws ArithmaticException{
        return a/b;
    }
}
```

3、方法重写对于throws要求

方法重写时，对于方法签名是有严格要求的：

- (1) 方法名必须相同
- (2) 形参列表必须相同
- (3) 返回值类型
 - 基本数据类型和void：必须相同
 - 引用数据类型：<=
- (4) 权限修饰符：>=，而且要求父类被重写方法在子类中是可见的
- (5) 不能是static, final修饰的方法
- (6) throws异常列表要求
 - 如果父类被重写方法的方法签名后面没有“throws 编译时异常类型”，那么重写方法时，方法签名后面也不能出现“throws 编译时异常类型”。
 - 如果父类被重写方法的方法签名后面有“throws 编译时异常类型”，那么重写方法时，throws的编译时异常类型必须<=被重写方法throws的编译时异常类型，或者不throws编译时异常。

- 方法重写，对于“throws 运行时异常类型”没有要求。

```
package com.atguigu.keyword;

import java.io.IOException;

public class TestOverride {

}

class Father{
    public void method() throws Exception{
        System.out.println("Father.method");
    }
}
class Son extends Father{
    @Override
    public void method() throws IOException,ClassCastException {
        System.out.println("Son.method");
    }
}
```

8.2.4 手工抛出异常对象：throw

Java程序的执行过程中如出现异常，会生成一个异常类对象，该异常对象将被提交给Java运行时系统，这个过程称为抛出(throw)异常。异常对象的生成有两种方式：

- 由虚拟机自动生成：程序运行过程中，虚拟机检测到程序发生了问题，就会在后台自动创建一个对应异常类的实例对象并抛出——自动抛出。
- 由开发人员手动创建：new 异常类型(【实参列表】)；如果创建好的异常对象不抛出对程序没有任何影响，和创建一个普通对象一样，但是一旦throw抛出，就会对程序运行产生影响了。

使用格式：

```
throw new 异常类名(参数);
```

throw语句抛出的异常对象，和JVM自动创建和抛出的异常对象一样。

- 如果是编译时异常类型的对象，同样需要使用throws或者try...catch处理，否则编译不通过。
- 如果是运行时异常类型的对象，编译器不提示。
- 但是无论是编译时异常类型的对象，还是运行时异常类型的对象，如果没有被try..catch合理的处理，都会导致程序崩溃。

throw语句会导致程序执行流程被改变，throw语句是明确抛出一个异常对象，因此它下面的代码将不会执行，如果当前方法没有try...catch处理这个异常对象，throw语句就会代替return语句提前终止当前方法的执行，并返回一个异常对象给调用者。

```
package com.atguigu.keyword;

public class TestThrow {
```

```

public static void main(String[] args) {
    try {
        System.out.println(max(4, 2, 31, 1));
    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        System.out.println(max(4));
    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        System.out.println(max());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static int max(int... nums){
    if(nums == null || nums.length==0){
        throw new IllegalArgumentException("没有传入任何整数，无法获取最大值");
    }
    int max = nums[0];
    for (int i = 1; i < nums.length; i++) {
        if(nums[i] > max){
            max = nums[i];
        }
    }
    return max;
}
}

```

8.3 自定义异常

为什么需要自定义异常类:

我们说了Java中不同的异常类,分别表示着某一种具体的异常情况,那么在开发中总是有些异常情况是核心类库中没有定义好的, 此时我们需要根据自己业务的异常情况来定义异常类。例如年龄负数问题, 考试成绩负数问题等等。

异常类如何定义:

1. 自定义一个编译时异常类型: 自定义类 并继承 `java.lang.Exception`。
2. 自定义一个运行时异常类型: 自定义类 并继承 `java.lang.RuntimeException`。

==注意==自定义的异常只能通过throw抛出。

自定义异常:

- (1) 要继承一个异常类型
- (2) 建议大家提供至少两个构造器, 一个是无参构造, 一个是(String message)构造器
- (3) 自定义异常对象只能手动抛出。抛出后由try..catch处理, 也可以甩锅throws给调用者处理。

演示自定义异常：

```
package com.atguigu.define;

public class NotTriangleException extends Exception{
    public NotTriangleException() {
    }

    public NotTriangleException(String message) {
        super(message);
    }
}
```

```
package com.atguigu.define;

public class Triangle {
    private double a;
    private double b;
    private double c;

    public Triangle(double a, double b, double c) throws NotTriangleException {
        if(a<=0 || b<=0 || c<=0){
            throw new NotTriangleException("三角形的边长必须是正数");
        }
        if(a+b<=c || b+c<=a || a+c<=b){
            throw new NotTriangleException(a+" , " + b + ", " + c +"不能构造三角形, 三角形任意两边之后必须大于第三边");
        }
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public double getA() {
        return a;
    }

    public void setA(double a) throws NotTriangleException{
        if(a<=0){
            throw new NotTriangleException("三角形的边长必须是正数");
        }
        if(a+b<=c || b+c<=a || a+c<=b){
            throw new NotTriangleException(a+" , " + b + ", " + c +"不能构造三角形, 三角形任意两边之后必须大于第三边");
        }
        this.a = a;
    }

    public double getB() {
        return b;
    }
```

```

public void setB(double b) throws NotTriangleException {
    if(b<=0){
        throw new NotTriangleException("三角形的边长必须是正数");
    }
    if(a+b<=c || b+c<=a || a+c<=b){
        throw new NotTriangleException(a+", " + b + ", " + c +"不能构造三角形, 三角形任意两边之后必须大于第三边");
    }
    this.b = b;
}

public double getC() {
    return c;
}

public void setC(double c) throws NotTriangleException {
    if(c<=0){
        throw new NotTriangleException("三角形的边长必须是正数");
    }
    if(a+b<=c || b+c<=a || a+c<=b){
        throw new NotTriangleException(a+", " + b + ", " + c +"不能构造三角形, 三角形任意两边之后必须大于第三边");
    }
    this.c = c;
}

@Override
public String toString() {
    return "Triangle{" +
        "a=" + a +
        ", b=" + b +
        ", c=" + c +
        '}';
}
}

```

```

package com.atguigu.define;

public class TestTriangle {
    public static void main(String[] args) {
        Triangle t = null;
        try {
            t = new Triangle(2,2,3);
            System.out.println("三角形创建成功: ");
            System.out.println(t);
        } catch (NotTriangleException e) {
            System.err.println("三角形创建失败");
            e.printStackTrace();
        }
        try {
            if(t != null) {

```

```
        t.setA(1);
    }
    System.out.println("三角形边长修改成功");
} catch (NotTriangleException e) {
    System.out.println("三角形边长修改失败");
    e.printStackTrace();
}
}
```

第九章 多线程

我们在之前，学习的程序在没有跳转语句的前提下，都是由上至下依次执行，那现在想要设计一个程序，边打游戏边听歌，怎么设计？

要解决上述问题，咱们得使用多进程或者多线程来解决。

9.1 相关概念（了解）

9.1.1 线程与进程

- **程序**：为了完成某个任务和功能，选择一种编程语言编写的一组指令的集合。
- **软件**：1个或多个应用程序+相关的素材和资源文件等构成一个软件系统。
- **进程**：是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。
- **线程**：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

简而言之：一个软件中至少有一个应用程序，应用程序的一次运行就是一个进程，一个进程中至少有一个线程。

- 面试题：进程是操作系统调度和分配资源的最小单位，线程是CPU调度的最小单位。不同的进程之间是不共享内存的。进程之间的数据交换和通信的成本是很高。不同的线程是共享同一个进程的内存的。当然不同的线程也有自己独立的内存空间。对于方法区，堆中的同一个对象的内存，线程之间是可以共享的，但是栈的局部变量永远是独立的。另外进程之间切换的复杂度要远远高于线程之间的切换调度。

9.1.2 查看进程和线程

我们可以再电脑底部任务栏，右键----->打开任务管理器，可以查看当前任务的进程：

- 1、每个应用程序的运行都是一个进程

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称

应用 (6)

| | 9% CPU | 46% 内存 | 2% 磁盘 | 0% 网络 |
|----------------------|--------|---------|----------|----------|
| Google Chrome (32 位) | 0.6% | 47.8 MB | 0.1 MB/秒 | 0 Mbps |
| Task Manager | 1.2% | 18.8 MB | 0 MB/秒 | 0 Mbps |
| WeChat (32 位) | 0% | 58.8 MB | 0 MB/秒 | 0 Mbps |
| Windows 资源管理器 | 1.0% | 56.7 MB | 0 MB/秒 | 0 Mbps |
| 百度浏览器 (32 位) | 0.7% | 24.7 MB | 0.1 MB/秒 | 0.1 Mbps |
| 有道云笔记 (32 位) | 0% | 44.4 MB | 0 MB/秒 | 0 Mbps |

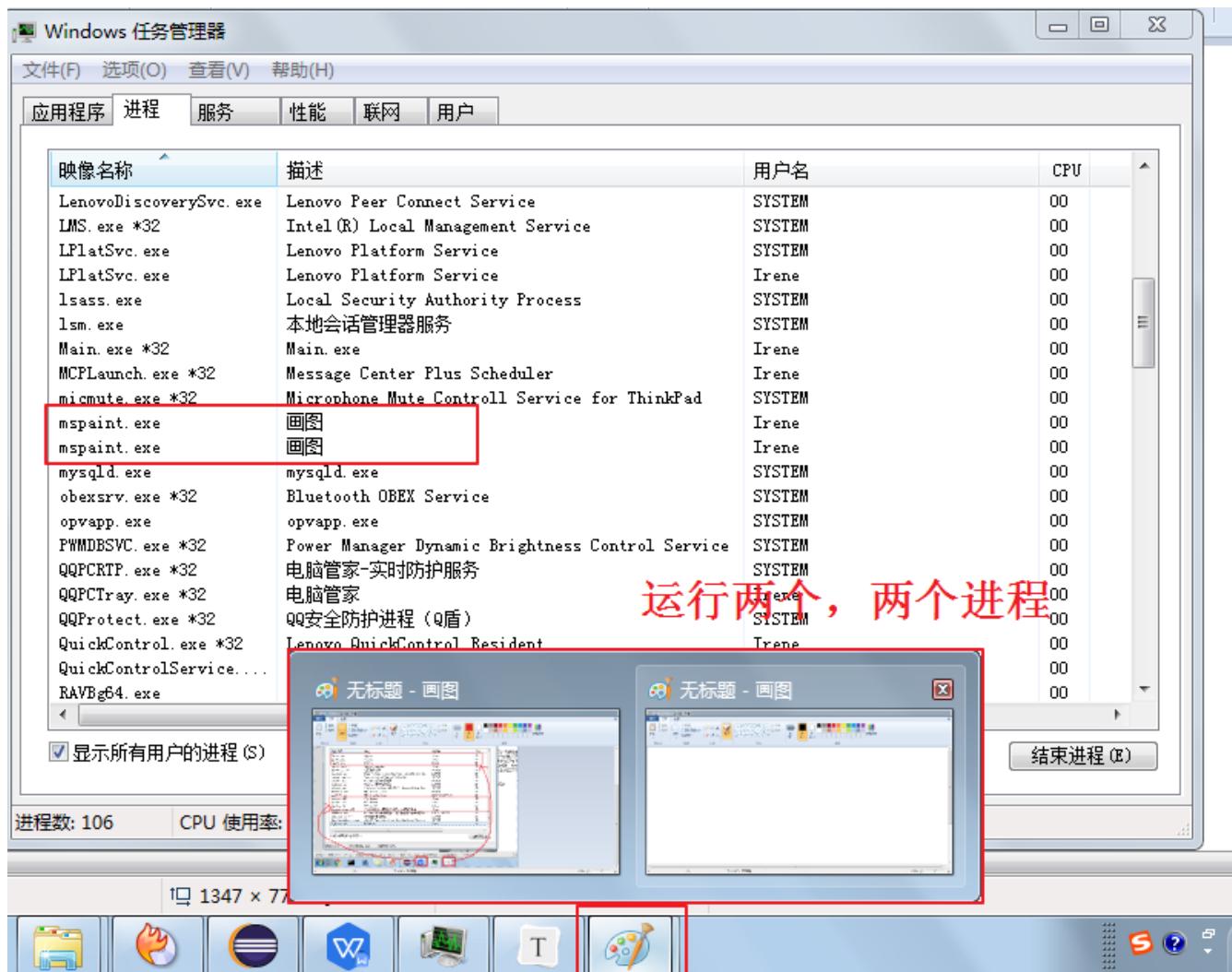
后台进程 (57)

| | | | | |
|------------------------|----|--------|--------|--------|
| Application Frame Host | 0% | 3.7 MB | 0 MB/秒 | 0 Mbps |
| bbnetservice | 0% | 1.5 MB | 0 MB/秒 | 0 Mbps |
| COM Surrogate | 0% | 1.5 MB | 0 MB/秒 | 0 Mbps |
| Cortana (小娜) | 0% | 0.6 MB | 0 MB/秒 | 0 Mbps |
| Elan Service | 0% | 0.6 MB | 0 MB/秒 | 0 Mbps |
| ETD Control Center | 0% | 2.1 MB | 0 MB/秒 | 0 Mbps |

简略信息(D) 结束任务(E)

应用下的和后台进程下的每一行都是一个进程

2、一个应用程序的多次运行，就是多个进程



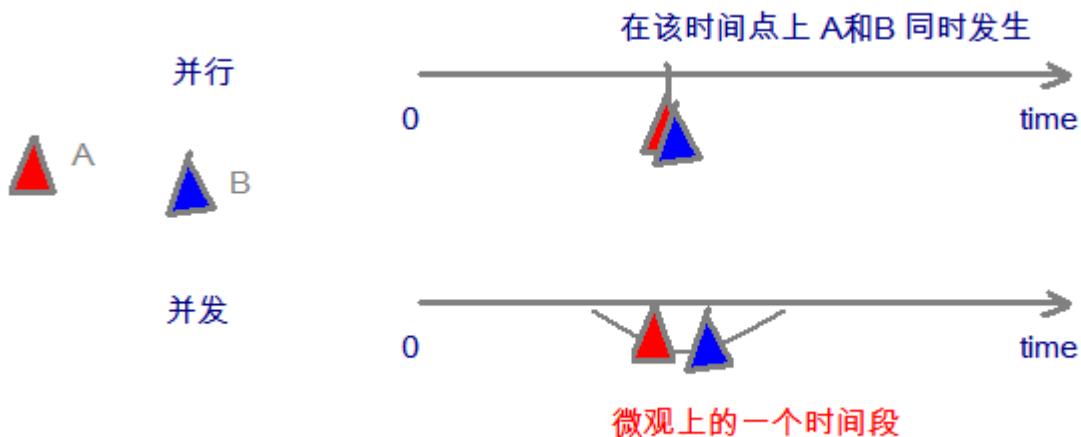
3、一个进程中包含多个线程



9.1.3 并发与并行

- 并行 (parallel)**：指两个或多个事件在同一时刻发生（同时发生）。指在同一时刻，有多条指令在多个处理器上同时执行。

- **并发** (concurrency) : 指两个或多个事件在同一个时间段内发生。指在同一个时刻只能有一条指令执行，但多个进程的指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。



在操作系统中，启动了多个程序，并发指的是在一段时间内宏观上有多个程序同时运行，这在单 CPU 系统中，每一时刻只能有一个程序执行，即微观上这些程序是分时的交替运行，只不过是给人的感觉是同时运行，那是因为分时交替运行的时间是非常短的。

而在多个 CPU 系统中，则这些可以并发执行的程序便可以分配到多个处理器上（CPU），实现多任务并行执行，即利用每个处理器来处理一个可以并发执行的程序，这样多个程序便可以同时执行。目前电脑市场上说的多核 CPU，便是多核处理器，核越多，**并行**处理的程序越多，能大大的提高电脑运行的效率。

例子：

- 并行：多项工作一起执行，之后再汇总，例如：泡方便面，电水壶烧水，一边撕调料倒入桶中
- 并发：同一时刻多个线程在访问同一个资源，多个线程对一个点，例如：春运抢票、电商秒杀...

注意：单核处理器的计算机肯定是不能并行的处理多个任务的，只能是多个任务在单个CPU上并发运行。同理，线程也是一样的，从宏观角度上理解线程是并行运行的，但是从微观角度上分析却是串行运行的，即一个线程一个线程的去运行，当系统只有一个CPU时，线程会以某种顺序执行多个线程，我们把这种情况称之为线程调度。

单核CPU：只能并发

多核CPU：并行+并发

9.1.4 线程调度

- 分时调度

所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。

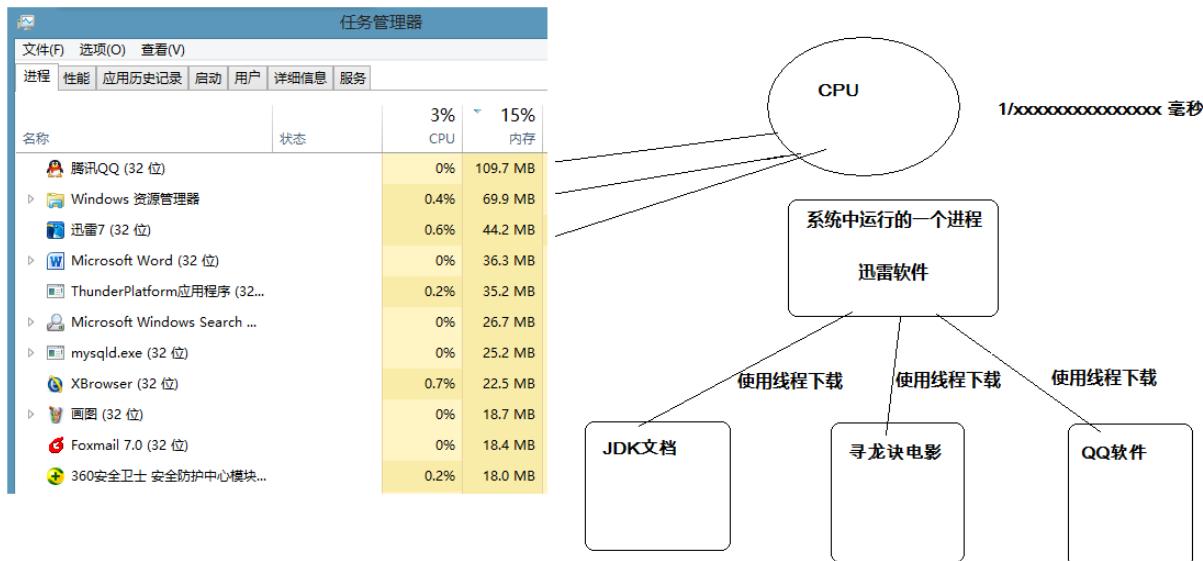
- 抢占式调度

优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个(线程随机性)，Java使用的为抢占式调度。

- 抢占式调度详解

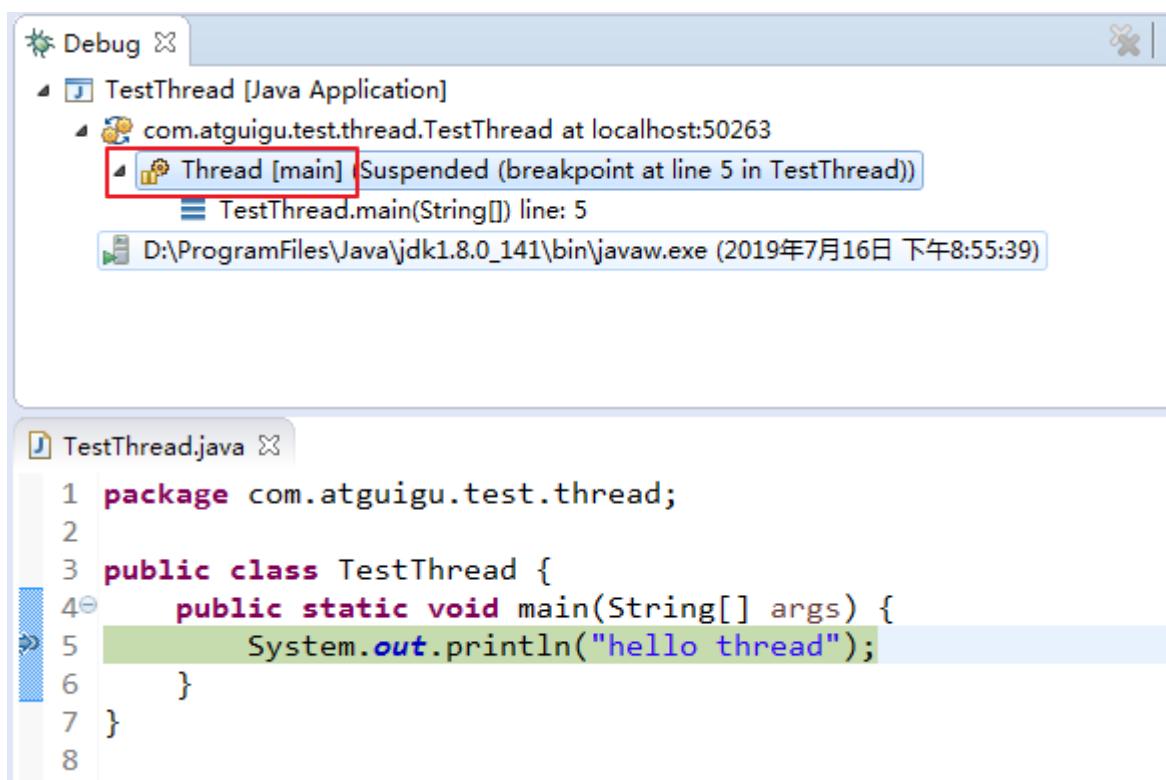
大部分操作系统都支持多进程并发运行，现在的操作系统几乎都支持同时运行多个程序。比如：现在我们上课一边使用编辑器，一边使用录屏软件，同时还开着画图板，dos窗口等软件。此时，这些程序是在同时运行，“感觉这些软件好像在同一时刻运行着”。

实际上，CPU(中央处理器)使用抢占式调度模式在多个线程间进行着高速的切换。对于CPU的一个核而言，某个时刻，只能执行一个线程，而CPU的在多个线程间切换速度相对我们的感觉要快，看上去就是在同一时刻运行。其实，多线程程序并不能提高程序的运行速度，但能够提高程序运行效率，==让CPU的使用率更高==。



9.2 另行创建和启动线程

当运行Java程序时，其实已经有一个线程了，那就是main线程。



那么如何创建和启动main线程以外的线程呢？

9.2.1 继承Thread类

Java使用 `java.lang.Thread` 类代表线程，所有的线程对象都必须是 `Thread` 类或其子类的实例。每个线程的作用是完成一定的任务，实际上就是执行一段程序流即一段顺序执行的代码。Java 使用线程执行体来代表这段程序流。Java 中通过继承 `Thread` 类来 **创建并启动多线程** 的步骤如下：

1. 定义 `Thread` 类的子类，并重写该类的 `run()` 方法，该 `run()` 方法的方法体就代表了线程需要完成的任务，因此把 `run()` 方法称为线程执行体。
2. 创建 `Thread` 子类的实例，即创建了线程对象
3. 调用线程对象的 `start()` 方法来启动该线程

代码如下：

自定义线程类：

```
package com.atguigu.thread;

public class MyThread extends Thread {
    //定义指定线程名称的构造方法
    public MyThread(String name) {
        //调用父类的String参数的构造方法，指定线程的名称
        super(name);
    }
    /**
     * 重写run方法，完成该线程执行的逻辑
     */
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName()+"：正在执行！ "+i);
        }
    }
}
```

测试类：

```
package com.atguigu.thread;

public class TestMyThread {
    public static void main(String[] args) {
        //创建自定义线程对象
        MyThread mt = new MyThread("新的线程！");
        //开启新线程
        mt.start();
        //在主方法中执行for循环
        for (int i = 0; i < 10; i++) {
            System.out.println("main线程！ "+i);
        }
    }
}
```

9.2.2 实现Runnable接口

Java有单继承的限制，当我们无法继承Thread类时，那么该如何做呢？在核心类库中提供了Runnable接口，我们可以实现Runnable接口，重写run()方法，然后再通过Thread类的对象代理启动和执行我们的线程体run()方法。

步骤如下：

1. 定义Runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。
2. 创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正 的线程对象。
3. 调用线程对象的start()方法来启动线程。 代码如下：

自定义线程类：

```
package com.atguigu.thread;

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
    }
}
```

测试类：

```
package com.atguigu.thread;

public class TestMyRunnable {
    public static void main(String[] args) {
        //创建自定义类对象 线程任务对象
        MyRunnable mr = new MyRunnable();
        //创建线程对象
        Thread t = new Thread(mr, "长江");
        t.start();
        for (int i = 0; i < 20; i++) {
            System.out.println("黄河 " + i);
        }
    }
}
```

通过实现Runnable接口，使得该类有了多线程类的特征。run()方法是多线程程序的一个执行目标。所有的多线程 代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread 对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是继承Thread类还是实现 Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的，熟悉Thread类的API是进行多线程编程的基础。

tips:Runnable对象仅仅作为Thread对象的target, Runnable实现类里包含的run()方法仅作为线程执行体。而实际的线程对象依然是Thread实例，只是该Thread线程负责执行其target的run()方法。

9.2.3 使用匿名内部类对象来实现线程的创建和启动

```
new Thread("新的线程! "){  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(getName()+"正在执行! "+i);  
        }  
    }  
}.start();
```

```
new Thread(new Runnable(){  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(Thread.currentThread().getName()+" "+ i);  
        }  
    }  
}).start();
```

9.3 Thread类

9.3.1 构造方法

- public Thread(): 分配一个新的线程对象。
- public Thread(String name): 分配一个指定名字的新的线程对象。
- public Thread(Runnable target): 分配一个带有指定目标新的线程对象。
- public Thread(Runnable target, String name): 分配一个带有指定目标新的线程对象并指定名字。

9.3.2 常用方法系列1

- public void run(): 此线程要执行的任务在此处定义代码。
- public String getName(): 获取当前线程名称。
- public static Thread currentThread(): 返回对当前正在执行的线程对象的引用。
- public final boolean isAlive(): 测试线程是否处于活动状态。如果线程已经启动且尚未终止，则为活动状态。
- public final int getPriority(): 返回线程优先级
- public final void setPriority(int newPriority): 改变线程的优先级
 - 每个线程都有一定的优先级，优先级高的线程将获得较多的执行机会。每个线程默认的优先级都与创建它的父线程具有相同的优先级。Thread类提供了setPriority(int newPriority)和getPriority()方法类设置和获取线程的优先级，其中setPriority方法需要一个整数，并且范围在[1,10]之间，通常推荐设置Thread类的三个优先级常量：
 - MAX_PRIORITY (10) : 最高优先级
 - MIN_PRIORITY (1) : 最低优先级
 - NORM_PRIORITY (5) : 普通优先级，默认情况下main线程具有普通优先级。

示例：

- 获取main线程对象的名称和优先级。
- 声明一个匿名内部类继承Thread类，重写run方法，在run方法中获取线程名称和优先级。设置该线程优先级为最高优先级并启动该线程。

```
public static void main(String[] args) {  
    Thread t = new Thread(){  
        public void run(){  
            System.out.println(getName() + "的优先级：" + getPriority());  
        }  
    };  
    t.setPriority(Thread.MAX_PRIORITY);  
    t.start();  
  
    System.out.println(Thread.currentThread().getName() +"的优先级：" +  
Thread.currentThread().getPriority());  
}
```

9.3.3 常用方法系列2

- `public void start()` :导致此线程开始执行; Java虚拟机调用此线程的run方法。
- `public static void sleep(long millis)` :使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行）。
- `public static void yield()`: yield只是让当前线程暂停一下，让系统的线程调度器重新调度一次，希望优先级与当前线程相同或更高的其他线程能够获得执行机会，但是这个不能保证，完全有可能的情况是，当某个线程调用了yield方法暂停之后，线程调度器又将其调度出来重新执行。
- `void join()` : 等待该线程终止。

`void join(long millis)` : 等待该线程终止的时间最长为 millis 毫秒。如果millis时间到，将不再等待。

`void join(long millis, int nanos)` : 等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。

案例：

- 声明一个匿名内部类继承Thread类，重写run方法，实现打印[1,100]之间的偶数，要求每隔1秒打印1个偶数。
- 声明一个匿名内部类继承Thread类，重写run方法，实现打印[1,100]之间的奇数，
 - 当打印到5时，让奇数线程暂停一下，再继续。
 - 当打印到5时，让奇数线程停下来，让偶数线程执行完再打印。
 - 当打印到5时，让奇数线程停下来，让偶数线程先执行10秒完再打印。

```
package com.atguigu.api;  
  
public class TestThreadStateChange {  
    public static void main(String[] args) {  
        Thread te = new Thread() {  
            @Override  
            public void run() {  
                for (int i = 2; i <= 100; i += 2) {  
                    System.out.println("偶数线程：" + i);  
                    try {  
                        Thread.sleep(100);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        };  
        te.start();  
    }  
}
```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

te.start();

Thread to = new Thread() {
    @Override
    public void run() {
        for (int i = 1; i <= 100; i += 2) {
            System.out.println("奇数线程: " + i);
            if (i == 5) {
                // Thread.yield();
                try {
                    te.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
};

to.start();
}
}

```

9.3.4 如何让线程提前结束

一个线程如何让另一个线程提前结束呢？

线程的死亡有两种：

自然死亡：当一个线程的run方法执行完，线程自然会停止。

意外死亡：当一个线程遇到未捕获处理的异常，也会挂掉。

我们肯定希望是让线程自然死亡更好。

- `public final void stop()`: 强迫线程停止执行。该方法具有固有的不安全性，已经标记为`@Deprecated`（已过时、已废弃）`=不建议再使用`，那么我们就需要通过其他方式来停止线程了，其中一种方式是使用变量的值的变化来控制线程是否结束。
- 标记法

案例：

声明一个PrintEvenThread线程类，继承Thread类，重写run方法，实现打印[1,100]之间的偶数，要求每隔1毫秒打印1个偶数。

声明一个PrintOddThread线程类，继承Thread类，重写run方法，实现打印[1,100]之间的奇数。

在main线程中：

- (1) 创建两个线程对象，并启动两个线程

(2) 当打印奇数的线程结束了，让偶数的线程也停下来，就算偶数线程没有全部打印完[1,100]之间的偶数。

```
package com.atguigu.api;

public class PrintEvenThread extends Thread{
    private boolean flag = true;
    @Override
    public void run() {
        for (int i = 2; i <= 100 && flag; i += 2) {
            System.out.println("偶数线程: " + i);
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void setFlag(boolean flag) {
        this.flag = flag;
    }
}
```

```
package com.atguigu.api;

public class PrintOddThread extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 100; i += 2) {
            System.out.println("奇数线程: " + i);
        }
    }
}
```

```
package com.atguigu.api;

public class TestThreadStop {
    public static void main(String[] args) {
        PrintEvenThread pe = new PrintEvenThread();
        PrintOddThread po = new PrintOddThread();
        pe.start();
        po.start();

        try {
            po.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        pe.setFlag(false);
    }
}
```

9.3.5 守护线程（了解）

有一种线程，它是在后台运行的，它的任务是为其他线程提供服务的，这种线程被称为“守护线程”。JVM的垃圾回收线程就是典型的守护线程。

守护线程有个特点，就是如果所有非守护线程都死亡，那么守护线程自动死亡。

调用setDaemon(true)方法可将指定线程设置为守护线程。必须在线程启动之前设置，否则会报IllegalThreadStateException异常。

调用isDaemon()可以判断线程是否是守护线程。

```
public class TestThread {
    public static void main(String[] args) {
        MyDaemon m = new MyDaemon();
        m.setDaemon(true);
        m.start();

        for (int i = 1; i <= 100; i++) {
            System.out.println("main:" + i);
        }
    }
}

class MyDaemon extends Thread {
    public void run() {
        while (true) {
            System.out.println("我一直守护者你...");
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

9.4 线程安全

当我们使用多个线程访问同一资源（可以是同一个变量、同一个文件、同一条记录等）的时候，若多个线程只有读操作，那么不会发生线程安全问题，但是如果多个线程中对资源有读和写的操作，就容易出现线程安全问题。

我们通过一个案例，演示线程的安全问题：电影院要卖票，我们模拟电影院的卖票过程。假设要播放的电影是“葫芦娃大战奥特曼”，本次电影的座位共100个(本场电影只能卖100张票)。我们来模拟电影院的售票窗口，实现多个窗口同时卖“葫芦娃大战奥特曼”这场电影票(多个窗口一起卖这100张票)

9.4.1 同一个资源问题和线程安全问题

1、局部变量不能共享

示例代码：

```

package com.atguigu.unsafe;

public class SaleTicketDemo1 {
    public static void main(String[] args) {
        Window w1 = new Window();
        Window w2 = new Window();
        Window w3 = new Window();

        w1.start();
        w2.start();
        w3.start();
    }
}

class Window extends Thread {
    public void run() {
        int total = 100;
        while (total > 0) {
            System.out.println(getName() + "卖出一张票, 剩余:" + --total);
        }
    }
}

```

结果：发现卖出300张票。

问题：局部变量是每次调用方法都是独立的，那么每个线程的run()的total是独立的，不是共享数据。

2、不同对象的实例变量不共享

```

package com.atguigu.unsafe;

public class SaleTicketDemo2 {
    public static void main(String[] args) {
        TicketSale t1 = new TicketSale();
        TicketSale t2 = new TicketSale();
        TicketSale t3 = new TicketSale();

        t1.start();
        t2.start();
        t3.start();
    }
}

class TicketSale extends Thread {
    private int total = 100;

    public void run() {
        while (total > 0) {
            System.out.println(getName() + "卖出一张票, 剩余:" + --total);
        }
    }
}

```

结果：发现卖出300张票。

问题：不同的实例对象的实例变量是独立的。

3、静态变量是共享的

示例代码：

```
package com.atguigu.unsafe;

public class SaleTicketDemo3 {
    public static void main(String[] args) {
        TicketSaleThread t1 = new TicketSaleThread();
        TicketSaleThread t2 = new TicketSaleThread();
        TicketSaleThread t3 = new TicketSaleThread();

        t1.start();
        t2.start();
        t3.start();
    }
}

class TicketSaleThread extends Thread{
    private static int total = 100;
    public void run(){
        while(total>0) {
            try {
                Thread.sleep(10); //加入这个，使得问题暴露的更明显
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(getName() + "卖出一张票，剩余：" + --total);
        }
    }
}
```

运行结果：

```
Thread-1卖出一张票，剩余:99
Thread-2卖出一张票，剩余:98
Thread-0卖出一张票，剩余:99
Thread-2卖出一张票，剩余:96
Thread-0卖出一张票，剩余:95
Thread-1卖出一张票，剩余:97
Thread-1卖出一张票，剩余:94
Thread-2卖出一张票，剩余:92
Thread-0卖出一张票，剩余:93
Thread-1卖出一张票，剩余:91
Thread-2卖出一张票，剩余:89
Thread-0卖出一张票，剩余:90
Thread-0卖出一张票，剩余:87
Thread-1卖出一张票，剩余:86
Thread-2卖出一张票，剩余:88
Thread-2卖出一张票，剩余:85
```

```
Thread-0卖出一张票，剩余:85
Thread-1卖出一张票，剩余:84
Thread-1卖出一张票，剩余:83
Thread-0卖出一张票，剩余:82
Thread-2卖出一张票，剩余:81
Thread-1卖出一张票，剩余:80
Thread-0卖出一张票，剩余:78
Thread-2卖出一张票，剩余:79
Thread-1卖出一张票，剩余:77
Thread-2卖出一张票，剩余:77
Thread-0卖出一张票，剩余:77
Thread-1卖出一张票，剩余:76
Thread-0卖出一张票，剩余:74
Thread-2卖出一张票，剩余:75
Thread-1卖出一张票，剩余:73
Thread-0卖出一张票，剩余:71
Thread-2卖出一张票，剩余:72
Thread-2卖出一张票，剩余:70
Thread-0卖出一张票，剩余:69
Thread-1卖出一张票，剩余:68
Thread-2卖出一张票，剩余:66
Thread-1卖出一张票，剩余:67
Thread-0卖出一张票，剩余:65
Thread-1卖出一张票，剩余:64
Thread-2卖出一张票，剩余:62
Thread-0卖出一张票，剩余:63
Thread-0卖出一张票，剩余:60
Thread-1卖出一张票，剩余:59
Thread-2卖出一张票，剩余:61
Thread-1卖出一张票，剩余:58
Thread-0卖出一张票，剩余:56
Thread-2卖出一张票，剩余:57
Thread-2卖出一张票，剩余:55
Thread-0卖出一张票，剩余:54
Thread-1卖出一张票，剩余:53
Thread-2卖出一张票，剩余:52
Thread-0卖出一张票，剩余:50
Thread-1卖出一张票，剩余:51
Thread-2卖出一张票，剩余:49
Thread-1卖出一张票，剩余:48
Thread-0卖出一张票，剩余:48
Thread-2卖出一张票，剩余:47
Thread-1卖出一张票，剩余:47
Thread-0卖出一张票，剩余:46
Thread-2卖出一张票，剩余:45
Thread-1卖出一张票，剩余:43
Thread-0卖出一张票，剩余:44
Thread-1卖出一张票，剩余:42
Thread-0卖出一张票，剩余:40
Thread-2卖出一张票，剩余:41
Thread-1卖出一张票，剩余:39
Thread-2卖出一张票，剩余:38
Thread-0卖出一张票，剩余:37
```

```
Thread-2卖出一张票，剩余:36
Thread-0卖出一张票，剩余:34
Thread-1卖出一张票，剩余:35
Thread-2卖出一张票，剩余:33
Thread-1卖出一张票，剩余:31
Thread-0卖出一张票，剩余:32
Thread-1卖出一张票，剩余:29
Thread-2卖出一张票，剩余:30
Thread-0卖出一张票，剩余:30
Thread-2卖出一张票，剩余:28
Thread-1卖出一张票，剩余:27
Thread-0卖出一张票，剩余:26
Thread-2卖出一张票，剩余:25
Thread-0卖出一张票，剩余:24
Thread-1卖出一张票，剩余:23
Thread-2卖出一张票，剩余:21
Thread-0卖出一张票，剩余:20
Thread-1卖出一张票，剩余:22
Thread-2卖出一张票，剩余:18
Thread-1卖出一张票，剩余:19
Thread-0卖出一张票，剩余:18
Thread-2卖出一张票，剩余:17
Thread-1卖出一张票，剩余:17
Thread-0卖出一张票，剩余:16
Thread-1卖出一张票，剩余:14
Thread-2卖出一张票，剩余:15
Thread-0卖出一张票，剩余:13
Thread-2卖出一张票，剩余:12
Thread-0卖出一张票，剩余:10
Thread-1卖出一张票，剩余:11
Thread-1卖出一张票，剩余:9
Thread-2卖出一张票，剩余:8
Thread-0卖出一张票，剩余:7
Thread-0卖出一张票，剩余:5
Thread-1卖出一张票，剩余:4
Thread-2卖出一张票，剩余:6
Thread-2卖出一张票，剩余:3
Thread-0卖出一张票，剩余:2
Thread-1卖出一张票，剩余:1
Thread-1卖出一张票，剩余:0
Thread-2卖出一张票，剩余:-1
Thread-0卖出一张票，剩余:-2
```

结果：发现卖出近100张票。

问题（1）：但是有重复票或负数票问题。

原因：线程安全问题

问题（2）：如果要考虑有两场电影，各卖100张票等

原因：TicketThread类的静态变量，是所有TicketThread类的对象共享

4、同一个对象的实例变量共享

示例代码：多个Thread线程使用同一个Runnable对象

```
package com.atguigu.safe;

package com.atguigu.unsafe;

public class SaleTicketDemo4 {
    public static void main(String[] args) {
        TicketsaleRunnable tr = new TicketsaleRunnable();
        Thread t1 = new Thread(tr, "窗口一");
        Thread t2 = new Thread(tr, "窗口二");
        Thread t3 = new Thread(tr, "窗口三");

        t1.start();
        t2.start();
        t3.start();
    }
}

class TicketsaleRunnable implements Runnable {
    private int total = 100;

    public void run() {
        while (total > 0) {
            try {
                Thread.sleep(10); //加入这个，使得问题暴露的更明显
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "卖出一张票，剩余：" + --total);
        }
    }
}
```

结果：发现卖出近100张票。

问题：但是有重复票或负数票问题。

原因：线程安全问题

5、抽取资源类，共享同一个资源对象

示例代码：

```
package com.atguigu.unsafe;

public class SaleTicketDemo5 {
    public static void main(String[] args) {
        //2、创建资源对象
        Ticket ticket = new Ticket();

        //3、启动多个线程操作资源类的对象
    }
}
```

```

    Thread t1 = new Thread("窗口一") {
        public void run() {
            while (true) {
                ticket.sale();
            }
        }
    };
    Thread t2 = new Thread("窗口二") {
        public void run() {
            while (true) {
                ticket.sale();
            }
        }
    };
    Thread t3 = new Thread(new Runnable() {
        public void run() {
            ticket.sale();
        }
    }, "窗口三");

    t1.start();
    t2.start();
    t3.start();
}
}

//1、编写资源类
class Ticket {
    private int total = 100;

    public void sale() {
        if (total > 0) {
            try {
                Thread.sleep(10); //加入这个，使得问题暴露的更明显
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "卖出一张票，剩余：" + --total);
        } else {
            throw new RuntimeException("没有票了");
        }
    }

    public int getTotal() {
        return total;
    }
}

```

结果：发现卖出近100张票。

问题：但是有重复票或负数票问题。

原因：线程安全问题

9.4.2 尝试解决线程安全问题

要解决上述多线程并发访问一个资源的安全性问题：也就是解决重复票与不存在票问题，Java中提供了同步机制(synchronized)来解决。



根据案例简述：

窗口1线程进入操作的时候，窗口2和窗口3线程只能在外等着，窗口1操作结束，窗口1和窗口2和窗口3才有机会进入代码去执行。也就是说在某个线程修改共享资源的时候，其他线程不能修改该资源，等待修改完毕同步之后，才能去抢夺CPU资源，完成对应的操作，保证了数据的同步性，解决了线程不安全的现象。

为了保证每个线程都能正常执行原子操作，Java引入了线程同步机制。注意：在任何时候，最多允许一个线程拥有同步锁，谁拿到锁就进入代码块，其他的线程只能在外等着(BLOCKED)。

1、同步机制的原理

同步解决线程安全的原理：

同步机制的原理，其实就相当于给某段代码加“锁”，任何线程想要执行这段代码，都要先获得“锁”，我们称它同步锁。因为Java对象在堆中的数据分为对象头、实例变量、空白的填充。而对象头中包含：

- Mark Word：记录了和当前对象有关的GC、锁标记等信息。
- 指向类的指针：每一个对象需要记录它是由哪个类创建出来的。
- 数组长度（只有数组对象才有）

哪个线程获得了“同步锁”对象之后，“同步锁”对象就会记录这个线程的ID，这样其他线程就只能等待了，除非这个线程“释放”了锁对象，其他线程才能重新获得/占用“同步锁”对象。

2、同步代码块和同步方法

同步方法：synchronized关键字直接修饰方法，表示同一时刻只有一个线程能进入这个方法，其他线程在外面等着。

```
public synchronized void method(){  
    可能会产生线程安全问题的代码  
}
```

同步代码块：synchronized关键字可以用于某个区块前面，表示只对这个区块的资源实行互斥访问。格式：

```
synchronized(同步锁){  
    需要同步操作的代码  
}
```

3、同步锁对象的选择

同步锁对象可以是任意类型，但是必须保证竞争“同一个共享资源”的多个线程必须使用同一个“同步锁对象”。

对于同步代码块来说，同步锁对象是由程序员手动指定的，但是对于同步方法来说，同步锁对象只能是默认的，

- 静态方法：当前类的Class对象
- 非静态方法：this

4、同步代码的范围选择

锁的范围太小：不能解决安全问题

锁的范围太大：因为一旦某个线程抢到锁，其他线程就只能等待，所以范围太大，效率会降低，不能合理利用CPU资源。

5、代码演示

示例一：静态方法加锁

```
package com.atguigu.safe;

public class SaleTicketDemo3 {
    public static void main(String[] args) {
        TicketSaleThread t1 = new TicketSaleThread();
        TicketSaleThread t2 = new TicketSaleThread();
        TicketSaleThread t3 = new TicketSaleThread();

        t1.start();
        t2.start();
        t3.start();
    }
}

class TicketSaleThread extends Thread{
    private static int total = 100;
    public void run(){//直接锁这里，肯定不行，会导致，只有一个窗口卖票
        while(total>0) {
            saleOneTicket();
        }
    }
}

public synchronized static void saleOneTicket(){//锁对象是TicketSaleThread类的Class对象，而
一个类的Class对象在内存中肯定只有一个
    if(total > 0) {//不加条件，相当于条件判断没有进入锁管控，线程安全问题就没有解决
        System.out.println(Thread.currentThread().getName() + "卖出一张票，剩余：" + --
total);
    }
}
```

示例二：非静态方法加锁

```

package com.atguigu.safe;

public class SaleTicketDemo4 {
    public static void main(String[] args) {
        TicketSaleRunnable tr = new TicketSaleRunnable();
        Thread t1 = new Thread(tr, "窗口一");
        Thread t2 = new Thread(tr, "窗口二");
        Thread t3 = new Thread(tr, "窗口三");

        t1.start();
        t2.start();
        t3.start();
    }
}

class TicketSaleRunnable implements Runnable {
    private int total = 1000;

    public void run() {//直接锁这里，肯定不行，会导致，只有一个窗口卖票
        while (total > 0) {
            saleOneTicket();
        }
    }

    public synchronized void saleOneTicket()//锁对象是this， 这里就是TicketSaleRunnable对象，因为
上面3个线程使用同一个TicketSaleRunnable对象，所以可以
        if(total > 0) {//不加条件，相当于条件判断没有进入锁管控，线程安全问题就没有解决
            System.out.println(Thread.currentThread().getName() + "卖出一张票，剩余：" + --
total);
        }
    }
}

```

示例三：同步代码块

```

package com.atguigu.safe;

public class SaleTicketDemo5 {
    public static void main(String[] args) {
        //2、创建资源对象
        Ticket ticket = new Ticket();

        //3、启动多个线程操作资源类的对象
        Thread t1 = new Thread("窗口一") {
            public void run() {//不能给run()直接假设，因为t1,t2,t3的三个run方法分别属于三个Thread类
对象,
                // run方法是非静态方法，那么锁对象默认选this，那么锁对象根本不是同一个
                while (true) {
                    synchronized (ticket) {
                        ticket.sale();
                    }
                }
            }
        }
    }
}

```

```

    };
    Thread t2 = new Thread("窗口二") {
        public void run() {
            while (true) {
                synchronized (ticket) {
                    ticket.sale();
                }
            }
        }
    };
    Thread t3 = new Thread(new Runnable() {
        public void run() {
            synchronized (ticket) {
                ticket.sale();
            }
        }
    }, "窗口三");
}

t1.start();
t2.start();
t3.start();
}
}

//1、编写资源类
class Ticket {
    private int total = 1000;

    public void sale() {//也可以直接给这个方法加锁，锁对象是this，这里就是Ticket对象
        if (total > 0) {
            System.out.println(Thread.currentThread().getName() + "卖出一张票，剩余：" + --total);
        } else {
            throw new RuntimeException("没有票了");
        }
    }

    public int getTotal() {
        return total;
    }
}

```

9.4.6 单例设计模式的线程安全问题

1、饿汉式没有线程安全问题

饿汉式：在类初始化时就直接创建单例对象，而类初始化过程是没有线程安全问题的

形式一：

```
/*
public class HungryOne{
    public static final HungryOne INSTANCE = new Hungryone();
    private HungryOne(){}
}*/
public enum HungryOne{
    INSTANCE
}
```

形式二：

```
package com.atguigu.single.hungry;

public class HungrySingle {
    private static final Hungrysingle INSTANCE = new HungrySingle();
    private Hungrysingle(){}
    public static Hungrysingle getInstance(){
        return INSTANCE;
    }
}
```

测试类：

```
package com.atguigu.single.hungry;

public class TestHungry {
    public static void main(String[] args) {
        HungryOne h1 = HungryOne.INSTANCE;
        HungryOne h2 = HungryOne.INSTANCE;
        System.out.println(h1 == h2);

        System.out.println("-----");
        Hungrysingle s1 = Hungrysingle.getInstance();
        Hungrysingle s2 = Hungrysingle.getInstance();
        System.out.println(s1 == s2);
    }
}
```

2、懒汉式线程安全问题

懒汉式：延迟创建对象，第一次调用getInstance方法再创建对象

形式一：

```
package com.atguigu.single.lazy;

public class LazyOne {
    private static LazyOne instance;
```

```

private LazyOne() {}

public static synchronized LazyOne getInstance(){
    if(instance == null){
        instance = new LazyOne();
    }
    return instance;
}

//有指令重排问题
/*  public static LazyOne getInstance(){
    if(instance == null){
        synchronized (LazyOne.class) {
            try {
                Thread.sleep(10); //加这个代码，暴露问题
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if(instance == null){
                instance = new LazyOne();
            }
        }
    }
    return instance;
} */
}

```

形式二：

```

package com.atguigu.single.lazy;

public class LazySingle {
    private LazySingle instance;
    private LazySingle(){}
    private static class Inner{
        static final LazySingle INSTANCE = new LazySingle();
    }
    public static LazySingle getInstance(){
        return Inner.INSTANCE;
    }
}

```

测试类：

```

package com.atguigu.single.lazy;

import org.junit.Test;

public class TestLazy {
    @Test

```

```
public void test01(){
    LazyOne s1 = LazyOne.getInstance();
    LazyOne s2 = LazyOne.getInstance();

    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s1 == s2);
}

//把s1和s2声明在外面，是想要在线程的匿名内部类中为s1和s2赋值
LazyOne s1;
LazyOne s2;
@Test
public void test02(){
    Thread t1 = new Thread(){
        public void run(){
            s1 = LazyOne.getInstance();
        }
    };
    Thread t2 = new Thread{
        public void run(){
            s2 = LazyOne.getInstance();
        }
    };

    t1.start();
    t2.start();

    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s1 == s2);
}

LazySingle obj1;
LazySingle obj2;
@Test
public void test03(){
    Thread t1 = new Thread(){
        public void run(){
            obj1 = LazySingle.getInstance();
        }
    };
    Thread t2 = new Thread{
        public void run(){
            obj2 = LazySingle.getInstance();
        }
    };
}
```

```
        }
    };

    t1.start();
    t2.start();

    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println(obj1);
    System.out.println(obj2);
    System.out.println(obj1 == obj2);
}
}
```

9.5 等待唤醒机制

9.5.1 线程间通信

为什么要处理线程间通信：

多个线程在处理同一个资源，但是处理的动作（线程的任务）却不同。而多个线程并发执行时，在默认情况下CPU是随机切换线程的，当我们需要多个线程来共同完成一件任务，并且我们希望他们有规律的执行，那么多线程之间需要一些通信机制，可以协调它们的工作，以此来帮我们达到多线程共同操作一份数据。

比如：线程A用来生成包子的，线程B用来吃包子的，包子可以理解为同一资源，线程A与线程B处理的动作，一个生产，一个是消费，此时B线程必须等到A线程完成后才能执行，那么线程A与线程B之间就需要线程通信，即——等待唤醒机制。

9.5.2 等待唤醒机制

什么是等待唤醒机制

这是多个线程间的一种协作机制。谈到线程我们经常想到的是线程间的**竞争（race）**，比如去争夺锁，但这并不是故事的全部，线程间也会有协作机制。

就在一个线程满足某个条件时，就进入等待状态（**wait() / wait(time)**），等待其他线程执行完他们的指定代码过后再将其唤醒（**notify()**）；或可以指定wait的时间，等时间到了自动唤醒；在有多个线程进行等待时，如果需要，可以使用**notifyAll()**来唤醒所有的等待线程。**wait/notify**就是线程间的一种协作机制。

1. **wait**: 线程不再活动，不再参与调度，进入 wait set 中，因此不会浪费 CPU 资源，也不会去竞争锁了，这时的线程状态即是 WAITING 或 TIMED_WAITING。它还要等着别的线程执行一个**特别的动作**，也即是“**通知（notify）**”或者等待时间到，在这个对象上等待的线程从 wait set 中释放出来，重新进入到调度队列（ready queue）中
2. **notify**: 则选取所通知对象的 wait set 中的一个线程释放；
3. **notifyAll**: 则释放所通知对象的 wait set 上的全部线程。

注意：

被通知线程被唤醒后也不一定能立即恢复执行，因为它当初中断的地方是在同步块内，而此刻它已经不持有锁，所以她需要再次尝试去获取锁（很可能面临其它线程的竞争），成功后才能在当初调用 wait 方法之后的地方恢复执行。

总结如下：

- 如果能获取锁，线程就从 WAITING 状态变成 RUNNABLE（可运行）状态；
- 否则，线程就从 WAITING 状态又变成 BLOCKED（等待锁）状态

调用wait和notify方法需要注意的细节

1. wait方法与notify方法必须要由同一个锁对象调用。因为：对应的锁对象可以通过notify唤醒使用同一个锁对象调用的wait方法后的线程。
2. wait方法与notify方法是属于Object类的方法的。因为：锁对象可以是任意对象，而任意对象的所属类都是继承了Object类的。
3. wait方法与notify方法必须要在同步代码块或者是同步函数中使用。因为：必须要通过锁对象调用这2个方法。

9.5.3 生产者与消费者问题

等待唤醒机制可以解决经典的“生产者与消费者”的问题。

生产者与消费者问题（英语：Producer-consumer problem），也称有限缓冲问题（英语：Bounded-buffer problem），是一个多线程同步问题的经典案例。该问题描述了两个（多个）共享固定大小缓冲区的线程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

生产者与消费者问题中其实隐含了两个问题：

- 线程安全问题：因为生产者与消费者共享数据缓冲区，不过这个问题可以使用同步解决。
- 线程的协调工作问题：
 - 要解决该问题，就必须让生产者线程在缓冲区满时等待(wait)，暂停进入阻塞状态，等到下次消费者消耗了缓冲区中的数据的时候，通知(notify)正在等待的线程恢复到就绪状态，重新开始往缓冲区添加数据。同样，也可以让消费者线程在缓冲区空时进入等待(wait)，暂停进入阻塞状态，等到生产者往缓冲区添加数据之后，再通知(notify)正在等待的线程恢复到就绪状态。通过这样的通信机制来解决此类问题。

1、一个厨师一个服务员问题

案例：有家餐馆的取餐口比较小，只能放10份快餐，厨师做完快餐放在取餐口的工作台上，服务员从这个工作台取出快餐给顾客。现在有1个厨师和1个服务员。

```
package com.atguigu.thread5;

public class TestCommunicate {
    public static void main(String[] args) {
        // 1、创建资源类对象
        workbench workbench = new Workbench();

        // 2、创建和启动厨师线程
        new Thread("厨师") {
            public void run() {
```

```
        while (true) {
            workbench.put();
        }
    }.start();

// 3、创建和启动服务员线程
new Thread("服务员") {
    public void run() {

        while (true) {
            workbench.take();
        }
    }
}.start();
}

// 1、定义资源类
class workbench {
    private static final int MAX_VALUE = 10;
    private int num;

    public synchronized void put() {
        if (num >= MAX_VALUE) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        num++;
        System.out.println(Thread.currentThread().getName() + "制作了一份快餐，现在工作台上有：" +
+ num + "份快餐");
        this.notify();
    }

    public synchronized void take() {
        if (num <= 0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        num--;
        System.out.println(Thread.currentThread().getName() + "取走了一份快餐，现在工作台上有：" +
+ num + "份快餐");
        this.notify();
    }
}
```

2、多个厨师多个服务员问题

案例：有家餐馆的取餐口比较小，只能放10份快餐，厨师做完快餐放在取餐口的工作台上，服务员从这个工作台取出快餐给顾客。现在有多个厨师和多个服务员。

```
package com.atguigu.thread5;

public class TestCommunicate2 {
    public static void main(String[] args) {
        // 1、创建资源类对象
        WindowBoard windowBoard = new WindowBoard();

        // 2、创建和启动厨师线程
        // 3、创建和启动服务员线程
        Cook c1 = new Cook("张三",windowBoard);
        Cook c2 = new Cook("李四",windowBoard);
        Waiter w1 = new Waiter("小红",windowBoard);
        Waiter w2 = new Waiter("小绿",windowBoard);

        c1.start();
        c2.start();
        w1.start();
        w2.start();
    }

}

//1、定义资源类
class WindowBoard {
    private static final int MAX_VALUE = 10;
    private int num;

    public synchronized void put() {
        while (num >= MAX_VALUE) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        num++;
        System.out.println(Thread.currentThread().getName() + "制作了一份快餐，现在工作台上有：" + num + "份快餐");
        this.notifyAll();
    }

    public synchronized void take() {
        while (num <= 0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        }
    }
    num--;
    System.out.println(Thread.currentThread().getName() + "取走了一份快餐，现在工作台
上有：" + num + "份快餐");
    this.notifyAll();
}
}

//2、定义厨师类
class Cook extends Thread{
    private WindowBoard windowBoard;

    public Cook(String name,WindowBoard windowBoard) {
        super(name);
        this.windowBoard = windowBoard;
    }

    public void run(){
        while(true) {
            windowBoard.put();
        }
    }
}

//3、定义服务员类
class Waiter extends Thread{
    private WindowBoard windowBoard;

    public Waiter(String name,WindowBoard windowBoard) {
        super(name);
        this.windowBoard = windowBoard;
    }

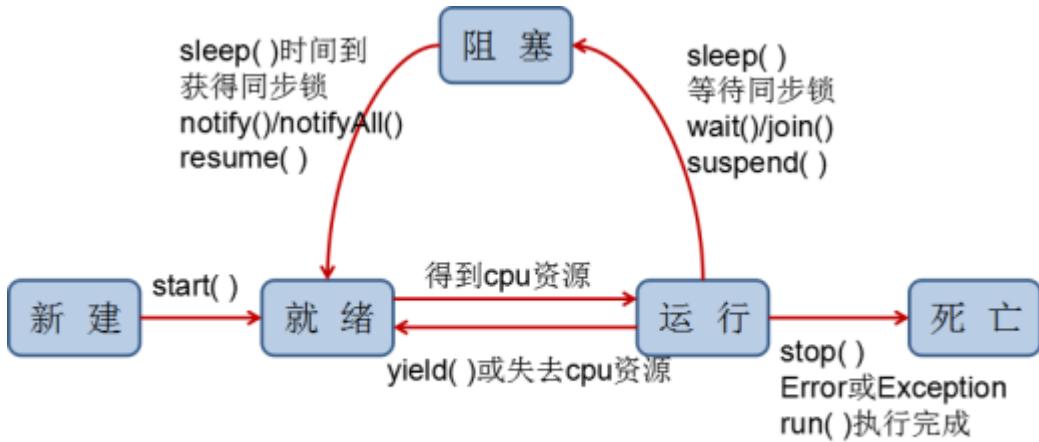
    public void run(){
        while(true) {
            windowBoard.take();
        }
    }
}

```

9.6 线程生命周期

9.6.1 观点1：5种状态（JDK1.5之前）

简单来说，线程的生命周期有五种状态：新建（New）、就绪（Runnable）、运行（Running）、阻塞（Blocked）、死亡（Dead）。CPU需要在多条线程之间切换，于是线程状态会多次在运行、阻塞、就绪之间切换。



1. 新建

当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态。此时它和其他Java对象一样，仅仅由JVM为其分配了内存，并初始化了实例变量的值。此时的线程对象并没有任何线程的动态特征，程序也不会执行它的线程体run()。

2. 就绪

但是当线程对象调用了start()方法之后，就不一样了，线程就从新建状态转为就绪状态。JVM会为其创建方法调用栈和程序计数器，当然，处于这个状态中的线程并没有开始运行，只是表示已具备了运行的条件，随时可以被调度。至于什么时候被调度，取决于JVM里线程调度器的调度。

注意：

程序只能对新建状态的线程调用start()，并且只能调用一次，如果对非新建状态的线程，如已启动的线程或已死亡的线程调用start()都会报错IllegalThreadStateException异常。

3. 运行

如果处于就绪状态的线程获得了CPU，开始执行run()方法的线程体代码，则该线程处于运行状态。如果计算机只有一个CPU，在任何时刻只有一个线程处于运行状态，如果计算机有多个处理器，将会有多个线程并行(Parallel)执行。

当然，美好的时光总是短暂的，而且CPU讲究雨露均沾。对于抢占式策略的系统而言，系统会给每个可执行的线程一个小时时间段来处理任务，当该时间用完，系统会剥夺该线程所占用的资源，让其回到就绪状态等待下一次被调度。此时其他线程将获得执行机会，而在选择下一个线程时，系统会适当考虑线程的优先级。

4. 阻塞

当在运行过程中的线程遇到如下情况时，线程会进入阻塞状态：

- 线程调用了sleep()方法，主动放弃所占用的CPU资源；
- 线程试图获取一个同步监视器，但该同步监视器正被其他线程持有；
- 线程执行过程中，同步监视器调用了wait()，让它等待某个通知（notify）；
- 线程执行过程中，同步监视器调用了wait(time)
- 线程执行过程中，遇到了其他线程对象的加塞（join）；
- 线程被调用suspend方法被挂起（已过时，因为容易发生死锁）；

当前正在执行的线程被阻塞后，其他线程就有机会执行了。针对如上情况，当发生如下情况时会解除阻塞，让该线程重新进入就绪状态，等待线程调度器再次调度它：

- 线程的sleep()时间到；
- 线程成功获得了同步监视器；
- 线程等到了通知(notify)；
- 线程wait的时间到了
- 加塞的线程结束了；
- 被挂起的线程又被调用了resume恢复方法（已过时，因为容易发生死锁）；

5. 死亡

线程会以以下三种方式之一结束，结束后的线程就处于死亡状态：

- run()方法执行完成，线程正常结束
- 线程执行过程中抛出了一个未捕获的异常（Exception）或错误（Error）
- 直接调用该线程的stop()来结束该线程（已过时，因为容易发生死锁）

9.6.2 观点2：6种状态（JDK1.5之后）

在java.lang.Thread.State的枚举类中这样定义：

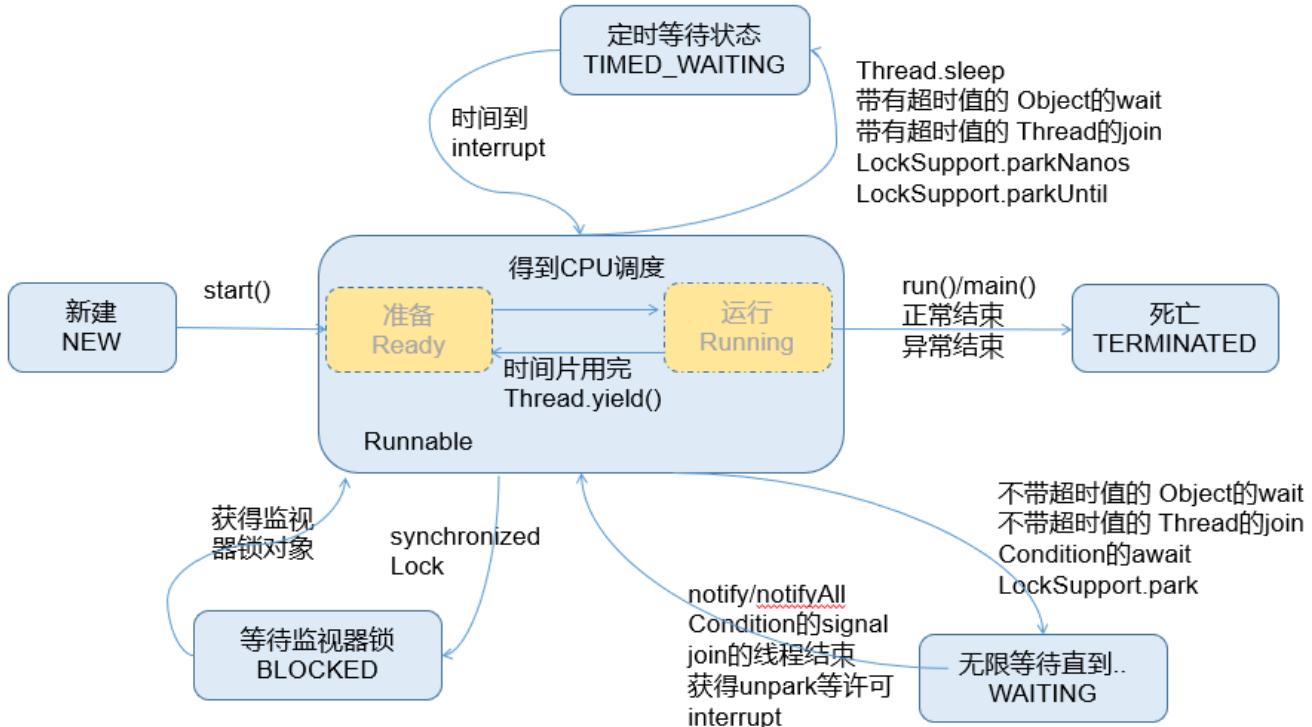
```
public enum State {
    NEW,
    RUNNABLE,
    BLOCKED,
    WAITING,
    TIMED_WAITING,
    TERMINATED;
}
```

首先它没有区分：就绪和运行状态，因为对于Java对象来说，只能标记为可运行，至于什么时候运行，不是JVM来控制的了，是OS来进行调度的，而且时间非常短暂，因此对于Java对象的状态来说，无法区分。只能我们人为的进行想象和理解。

其次根据Thread.State的定义，阻塞状态是分为三种的：BLOCKED、WAITING、TIMED_WAITING。

- BLOCKED：是指互有竞争关系的几个线程，其中一个线程占有锁对象时，其他线程只能等待锁。只有获得锁对象的线程才能有执行机会。
- TIMED_WAITING：当前线程执行过程中遇到Thread类的sleep或join，Object类的wait，LockSupport类的park方法，并且在调用这些方法时，设置了时间，那么当前线程会进入TIMED_WAITING，直到时间到，或被中断。
- WAITING：当前线程执行过程中遇到Object类的wait，Thread类的join，LockSupport类的park方法，并且在调用这些方法时，没有指定时间，那么当前线程会进入WAITING状态，直到被唤醒。
 - 通过Object类的wait进入WAITING状态的要有Object的notify/notifyAll唤醒；
 - 通过Condition的await进入WAITING状态的要有Condition的signal方法唤醒；
 - 通过LockSupport类的park方法进入WAITING状态的要有LockSupport类的unpark方法唤醒；
 - 通过Thread类的join进入WAITING状态，只有调用join方法的线程对象结束才能让当前线程恢复；

说明：当从WAITING或TIMED_WAITING恢复到Runnable状态时，如果发现当前线程没有得到监视器锁，那么会立刻转入BLOCKED状态。



9.7 释放锁操作与死锁

任何线程进入同步代码块、同步方法之前，必须先获得对同步监视器的锁定，那么何时会释放对同步监视器的锁定呢？

1、释放锁的操作

当前线程的同步方法、同步代码块执行结束。

当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致当前线程异常结束。

当前线程在同步代码块、同步方法中执行了锁对象的wait()方法，当前线程被挂起，并释放锁。

2、不会释放锁的操作

线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行。

线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）。应尽量避免使用suspend()和resume()这样的过时来控制线程。

3、死锁

不同的线程分别锁住对方需要的同步监视器对象不释放，都在等待对方先放弃时就形成了线程的死锁。一旦出现死锁，整个程序既不会发生异常，也不会给出任何提示，只是所有线程处于阻塞状态，无法继续。

```

public class TestDeadLock {
    public static void main(String[] args) {
        Object g = new Object();
        Object m = new Object();
        Owner s = new Owner(g,m);
    }
}
  
```

```

        Customer c = new Customer(g,m);
        new Thread(s).start();
        new Thread(c).start();
    }
}

class Owner implements Runnable{
    private Object goods;
    private Object money;

    public Owner(Object goods, Object money) {
        super();
        this.goods = goods;
        this.money = money;
    }

    @Override
    public void run() {
        synchronized (goods) {
            System.out.println("先给钱");
            synchronized (money) {
                System.out.println("发货");
            }
        }
    }
}

class Customer implements Runnable{
    private Object goods;
    private Object money;

    public Customer(Object goods, Object money) {
        super();
        this.goods = goods;
        this.money = money;
    }

    @Override
    public void run() {
        synchronized (money) {
            System.out.println("先发货");
            synchronized (goods) {
                System.out.println("再给钱");
            }
        }
    }
}

```

4、面试题：sleep()和wait()方法的区别

- (1) sleep()不释放锁，wait()释放锁
- (2) sleep()指定休眠的时间，wait()可以指定时间也可以无限等待直到notify或notifyAll
- (3) sleep()在Thread类中声明的静态方法，wait方法在Object类中声明

因为我们调用wait()方法是由锁对象调用，而锁对象的类型是任意类型的对象。那么希望任意类型的对象都要有的方法，只能声明在Object类中。

第十章 基础API与常见算法

10.1 和数学相关的类

10.1.1 java.lang.Math

java.lang.Math类包含用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数。类似这样的工具类，其所有方法均为静态方法，并且不会创建对象，调用起来非常简单。

- `public static double abs(double a)`：返回double值的绝对值。

```
double d1 = Math.abs(-5); //d1的值为5  
double d2 = Math.abs(5); //d2的值为5
```

- `public static double ceil(double a)`：返回大于等于参数的最小的整数。

```
double d1 = Math.ceil(3.3); //d1的值为 4.0  
double d2 = Math.ceil(-3.3); //d2的值为 -3.0  
double d3 = Math.ceil(5.1); //d3的值为 6.0
```

- `public static double floor(double a)`：返回小于等于参数最大的整数。

```
double d1 = Math.floor(3.3); //d1的值为3.0  
double d2 = Math.floor(-3.3); //d2的值为-4.0  
double d3 = Math.floor(5.1); //d3的值为 5.0
```

- `public static long round(double a)`：返回最接近参数的long。(相当于四舍五入方法)

```
long d1 = Math.round(5.5); //d1的值为6.0  
long d2 = Math.round(5.4); //d2的值为5.0
```

- `public static double pow(double a,double b)`：返回a的b幂次方法
- `public static double sqrt(double a)`：返回a的平方根
- `public static double random()`：返回[0,1)的随机值
- `public static final double PI`：返回圆周率
- `public static double max(double x, double y)`：返回x,y中的最大值
- `public static double min(double x, double y)`：返回x,y中的最小值

```
double result = Math.pow(2,31);  
double sqrt = Math.sqrt(256);  
double rand = Math.random();  
double pi = Math.PI;
```

10.1.2 java.math包

(1) BigInteger

不可变的任意精度的整数。

- BigInteger(String val)
- BigInteger add(BigInteger val)
- BigInteger subtract(BigInteger val)
- BigInteger multiply(BigInteger val)
- BigInteger divide(BigInteger val)
- BigInteger remainder(BigInteger val)
-

```
@Test
public void test01(){
//    long bigNum = 123456789123456789123456789L;

    BigInteger b1 = new BigInteger("123456789123456789123456789");
    BigInteger b2 = new BigInteger("78923456789123456789123456789");

//    System.out.println("和: " + (b1+b2)); //错误的，无法直接使用+进行求和

    System.out.println("和: " + b1.add(b2));
    System.out.println("减: " + b1.subtract(b2));
    System.out.println("乘: " + b1.multiply(b2));
    System.out.println("除: " + b2.divide(b1));
    System.out.println("余: " + b2.remainder(b1));
}
```

(2) RoundingMode枚举类

CEILING：向正无限大方向舍入的舍入模式。

DOWN：向零方向舍入的舍入模式。

FLOOR：向负无限大方向舍入的舍入模式。

HALF_DOWN：向最接近数字方向舍入的舍入模式，如果与两个相邻数字的距离相等，则向下舍入。

HALF_EVEN：向最接近数字方向舍入的舍入模式，如果与两个相邻数字的距离相等，则向相邻的偶数舍入。

HALF_UP：向最接近数字方向舍入的舍入模式，如果与两个相邻数字的距离相等，则向上舍入。

UNNECESSARY：用于断言请求的操作具有精确结果的舍入模式，因此不需要舍入。 UP：远离零方向舍入的舍入模式。

(3) BigDecimal

不可变的、任意精度的有符号十进制数。

- BigDecimal(String val)
- BigDecimal add(BigDecimal val)
- BigDecimal subtract(BigDecimal val)

- BigDecimal multiply(BigDecimal val)
- BigDecimal divide(BigDecimal val)
- BigDecimal divide(BigDecimal divisor, int roundingMode)
- BigDecimal divide(BigDecimal divisor, int scale, RoundingMode roundingMode)
- BigDecimal remainder(BigDecimal val)
-

```

    @Test
    public void test02(){
        /*double big = 12.123456789123456789123456789;
        System.out.println("big = " + big);*/

        BigDecimal b1 = new BigDecimal("123.45678912345678912345678912345678");
        BigDecimal b2 = new BigDecimal("7.8923456789123456789123456789998898888");

        //System.out.println("和: " + (b1+b2));//错误的，无法直接使用+进行求和

        System.out.println("和: " + b1.add(b2));
        System.out.println("减: " + b1.subtract(b2));
        System.out.println("乘: " + b1.multiply(b2));
        System.out.println("除: " +
b1.divide(b2,20,RoundingMode.UP));//divide(BigDecimal divisor, int scale, int roundingMode)
        System.out.println("除: " +
b1.divide(b2,20,RoundingMode.DOWN));//divide(BigDecimal divisor, int scale, int
roundingMode)
        System.out.println("余: " + b1.remainder(b2));
    }

```

10.1.3 java.util.Random

用于产生随机数

- boolean nextBoolean():返回下一个伪随机数，它是取自此随机数生成器序列的均匀分布的 boolean 值。
- void nextBytes(byte[] bytes):生成随机字节并将其置于用户提供的 byte 数组中。
- double nextDouble():返回下一个伪随机数，它是取自此随机数生成器序列的、在 0.0 和 1.0 之间均匀分布的 double 值。
- float nextFloat():返回下一个伪随机数，它是取自此随机数生成器序列的、在 0.0 和 1.0 之间均匀分布的 float 值。
- double nextGaussian():返回下一个伪随机数，它是取自此随机数生成器序列的、呈高斯（“正态”）分布的 double 值，其平均值是 0.0，标准差是 1.0。
- int nextInt():返回下一个伪随机数，它是此随机数生成器的序列中均匀分布的 int 值。
- int nextInt(int n):返回一个伪随机数，它是取自此随机数生成器序列的、在 0 (包括) 和指定值 (不包括) 之间均匀分布的 int 值。
- long nextLong():返回下一个伪随机数，它是取自此随机数生成器序列的均匀分布的 long 值。

```
@Test
public void test03(){
    Random r = new Random();
    System.out.println("随机整数: " + r.nextInt());
    System.out.println("随机小数: " + r.nextDouble());
    System.out.println("随机布尔值: " + r.nextBoolean());
}
```

10.2 日期时间API

10.2.1 JDK1.8之前

1、java.util.Date

new Date(): 当前系统时间

long getTime(): 返回该日期时间对象距离1970-1-1 0:0:0 0毫秒之间的毫秒值

new Date(long 毫秒): 把该毫秒值换算成日期时间对象

```
@Test
public void test5(){
    long time = Long.MAX_VALUE;
    Date d = new Date(time);
    System.out.println(d);
}

@Test
public void test4(){
    long time = 1559807047979L;
    Date d = new Date(time);
    System.out.println(d);
}

@Test
public void test3(){
    Date d = new Date();
    long time = d.getTime();
    System.out.println(time); //1559807047979
}

@Test
public void test2(){
    long time = System.currentTimeMillis();
    System.out.println(time); //1559806982971
    //当前系统时间距离1970-1-1 0:0:0 0毫秒的时间差，毫秒为单位
}

@Test
public void test1(){
    Date d = new Date();
    System.out.println(d);
}
```

2、java.text.SimpleDateFormat

SimpleDateFormat用于日期时间的格式化。

| 字母 | 日期或时间元素 | 表示 | 示例 |
|----|--------------------|-------------------|---------------------------------------|
| G | Era 标志符 | Text | AD |
| y | 年 | Year | 1996; 96 |
| M | 年中的月份 | Month | July; Jul; 07 |
| w | 年中的周数 | Number | 27 |
| W | 月份中的周数 | Number | 2 |
| D | 年中的天数 | Number | 189 |
| d | 月份中的天数 | Number | 10 |
| F | 月份中的星期 | Number | 2 |
| E | 星期中的天数 | Text | Tuesday; Tue |
| a | Am/pm 标记 | Text | PM |
| H | 一天中的小时数 (0-23) | Number | 0 |
| k | 一天中的小时数 (1-24) | Number | 24 |
| K | am/pm 中的小时数 (0-11) | Number | 0 |
| h | am/pm 中的小时数 (1-12) | Number | 12 |
| m | 小时中的分钟数 | Number | 30 |
| s | 分钟中的秒数 | Number | 55 |
| S | 毫秒数 | Number | 978 |
| z | 时区 | General time zone | Pacific Standard Time; PST; GMT-08:00 |
| Z | 时区 | RFC 822 time zone | -0800 |

```
@Test
public void test10() throws ParseException{
    String str = "2019年06月06日 16时03分14秒 545毫秒 星期四 +0800";
    SimpleDateFormat sf = new SimpleDateFormat("yyyy年MM月dd日 HH时mm分ss秒 SSS毫秒 E
Z");
    Date d = sf.parse(str);
    System.out.println(d);
}

@Test
public void test9(){
    Date d = new Date();

    SimpleDateFormat sf = new SimpleDateFormat("yyyy年MM月dd日 HH时mm分ss秒 SSS毫秒 E
Z");
    //把Date日期转成字符串，按照指定的格式转
    String str = sf.format(d);
    System.out.println(str);
}
```

3、java.util.TimeZone

通常，使用 `getDefault` 获取 `TimeZone`，`getDefault` 基于程序运行所在的时区创建 `TimeZone`。

也可以用 `getTimezone` 及时区 ID 获取 `Timezone`。例如美国太平洋时区的时区 ID 是 "America/Los_Angeles"。

```

import org.junit.Test;

import java.util.TimeZone;

public class TestTimezone {
    @Test
    public void test1(){
        String[] all = TimeZone.getAvailableIDs();
        for (int i = 0; i < all.length; i++) {
            System.out.println(all[i]);
        }
    }

    @Test
    public void test2(){
        TimeZone t = TimeZone.getTimeZone("America/Los_Angeles");
        System.out.println(t);
    }
}

```

常见时区ID：

```

Asia/Shanghai
UTC
America/New_York

```

4、java.util.Locale

`Locale` 对象表示了特定的地理、政治和文化地区。需要 `Locale` 来执行其任务的操作称为 **语言环境敏感的操作**，它使用 `Locale` 为用户量身定制信息。

语言参数是一个有效的 **ISO 语言代码**。这些代码是由 ISO-639 定义的小写两字母代码。

国家/地区参数是一个有效的 **ISO 国家/地区代码**。这些代码是由 ISO-3166 定义的大写两字母代码。

`Locale` 类提供了一些方便的常量，可用这些常量为常用的语言环境创建 `Locale` 对象。

```

import org.junit.Test;

import java.util.Locale;

public class TestLocale {
    @Test
    public void test01(){
        Locale[] all = Locale.getAvailableLocales();
        for (int i = 0; i < all.length; i++) {
            System.out.println(all[i]);
        }
    }

    @Test
    public void test02(){

```

```

        Locale china = Locale.CHINA;
        System.out.println("china = " + china);
    }
}

```

5、java.util.Calendar

`Calendar` 类是一个抽象类，它为特定瞬间与一组诸如 `YEAR`、`MONTH`、`DAY_OF_MONTH`、`HOUR` 等 `日历字段` 之间的转换提供了一些方法，并为操作日历字段（例如获得下星期的日期）提供了一些方法。瞬间可用毫秒值来表示，它是距 `1970年1月1日 00:00:00.000` 的偏移量。与其他语言环境敏感类一样，`Calendar` 提供了一个类方法 `getInstance`，以获得此类型的一个通用的对象。

| | |
|---|----------------------|
| <code>static Calendar getInstance()</code> | 使用默认时区和语言环境获得一个日历。 |
| <code>static Calendar getInstance(Locale aLocale)</code> | 使用默认时区和指定语言环境获得一个日历。 |
| <code>static Calendar getInstance(TimeZone zone)</code> | 使用指定时区和默认语言环境获得一个日历。 |
| <code>static Calendar getInstance(TimeZone zone, Locale aLocale)</code> | 使用指定时区和语言环境获得一个日历。 |

修改和获取 `YEAR`、`MONTH`、`DAY_OF_MONTH`、`HOUR` 等 `日历字段` 对应的时间值。

```

void add(int field,int amount)
int get(int field)
void set(int field, int value)

```

示例代码：

```

import org.junit.Test;

import java.util.Calendar;
import java.util.TimeZone;

public class TestCalendar {
    @Test
    public void test1(){
        Calendar c = Calendar.getInstance();
        System.out.println(c);

        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH)+1;
        int day = c.get(Calendar.DATE);
        int hour = c.get(Calendar.HOUR_OF_DAY);
        int minute = c.get(Calendar.MINUTE);

        System.out.println(year + "-" + month + "-" + day + " " + hour + ":" + minute);
    }

    @Test
    public void test2(){

```

```

        TimeZone t = TimeZone.getTimeZone("America/Los_Angeles");
        Calendar c = Calendar.getInstance(t);
        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH)+1;
        int day = c.get(Calendar.DATE);
        int hour = c.get(Calendar.HOUR_OF_DAY);
        int minute = c.get(Calendar.MINUTE);

        System.out.println(year + "-" + month + "-" + day + " " + hour + ":" + minute);
    }
}

```

10.2.2 JDK1.8之后

Java1.0中包含了一个Date类，但是它的大多数方法已经在Java 1.1引入Calendar类之后被弃用了。而Calendar并不比Date好多少。它们面临的问题是：

- 可变性：象日期和时间这样的类对象应该是不可变的。Calendar类中可以使用三种方法更改日历字段：set()、add() 和 roll()。
- 偏移性：Date中的年份是从1900开始的，而月份都是从0开始的。
- 格式化：格式化只对Date有用，Calendar则不行。
- 此外，它们也不是线程安全的，不能处理闰秒等。

可以说，对日期和时间的操作一直是Java程序员最痛苦的地方之一。第三次引入的API是成功的，并且java 8中引入的java.time API 已经纠正了过去的缺陷，将来很长一段时间内它都会为我们服务。

Java 8 吸收了 Joda-Time 的精华，以一个新的开始为 Java 创建优秀的 API。

- java.time – 包含值对象的基础包
- java.time.chrono – 提供对不同的日历系统的访问。
- java.time.format – 格式化和解析时间和日期
- java.time.temporal – 包括底层框架和扩展特性
- java.time.zone – 包含时区支持的类

Java 8 吸收了 Joda-Time 的精华，以一个新的开始为 Java 创建优秀的 API。新的 java.time 中包含了所有关于时钟 (Clock) ，本地日期 (LocalDate) 、本地时间 (LocalTime) 、本地日期时间 (LocalDateTime) 、时区 (ZonedDateTime) 和持续时间 (Duration) 的类。

1、本地日期时间： LocalDate、 LocalTime、 LocalDateTime

| 方法 | 描述 |
|---|--------------------------------|
| now() / now(ZoneId zone) | 静态方法，根据当前时间创建对象/指定时区的对象 |
| of() | 静态方法，根据指定日期/时间创建对象 |
| getDayOfMonth()/getDayOfYear() | 获得月份天数(1-31) /获得年份天数(1-366) |
| getDayOfWeek() | 获得星期几(返回一个 DayOfWeek 枚举值) |
| getMonth() | 获得月份, 返回一个 Month 枚举值 |
| getMonthValue() / getYear() | 获得月份(1-12) /获得年份 |
| getHours()/getMinute()/getSecond() | 获得当前对象对应的小时、分钟、秒 |
| withDayOfMonth()/withDayOfYear()/withMonth()/withYear() | 将月份天数、年份天数、月份、年份修改为指定的值并返回新的对象 |
| with(TemporalAdjuster t) | 将当前日期时间设置为校对器指定的日期时间 |
| plusDays(), plusWeeks(), plusMonths(), plusYears(),plusHours() | 向当前对象添加几天、几周、几个月、几年、几小时 |
| minusMonths() / minusWeeks()/minusDays()/minusYears()/minusHours() | 从当前对象减去几月、几周、几天、几年、几小时 |
| plus(TemporalAmount t)/minus(TemporalAmount t) | 添加或减少一个 Duration 或 Period |
| isBefore()/isAfter() | 比较两个 LocalDate |
| isLeapYear() | 判断是否是闰年 (在LocalDate类中声明) |
| format(DateTimeFormatter t) | 格式化本地日期、时间，返回一个字符串 |
| parse(CharSequence text) | 将指定格式的字符串解析为日期、时间 |

```

import org.junit.Test;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class TestLocalDateTime {
    @Test
    public void test7(){
        LocalDate now = LocalDate.now();
        LocalDate before = now.minusDays(100);
        System.out.println(before); //2019-02-26
    }

    @Test

```

```

public void test06(){
    LocalDate lai = LocalDate.of(2019, 5, 13);
    LocalDate go = lai.plusDays(160);
    System.out.println(go);//2019-10-20
}

@Test
public void test05(){
    LocalDate lai = LocalDate.of(2019, 5, 13);
    System.out.println(lai.getDayOfYear());
}

@Test
public void test04(){
    LocalDate lai = LocalDate.of(2019, 5, 13);
    System.out.println(lai);
}

@Test
public void test03(){
    LocalDateTime now = LocalDateTime.now();
    System.out.println(now);
}

@Test
public void test02(){
    LocalTime now = LocalTime.now();
    System.out.println(now);
}

@Test
public void test01(){
    LocalDate now = LocalDate.now();
    System.out.println(now);
}
}

```

2、指定时区日期时间：ZoneId和ZonedDateTime

常见时区ID：

```

Asia/Shanghai
UTC
America/New_York

```

可以通过ZoneId获取所有可用的时区ID：

```

import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Set;

```

```

public class TestZone {
    @Test
    public void test01() {
        //需要知道一些时区的id
        //Set<String>是一个集合，容器
        Set<String> availableZoneIds = ZoneId.getAvailableZoneIds();
        //快捷模板iter
        for (String availableZoneId : availableZoneIds) {
            System.out.println(availableZoneId);
        }
    }

    @Test
    public void test02(){
        ZonedDateTime t1 = ZonedDateTime.now();
        System.out.println(t1);

        ZonedDateTime t2 = ZonedDateTime.now(ZoneId.of("America/New_York"));
        System.out.println(t2);
    }
}

```

3、持续日期/时间：Period和Duration

Period:用于计算两个“日期”间隔

Duration:用于计算两个“时间”间隔

```

import org.junit.Test;

import java.time.Duration;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Period;

public class TestPeriodDuration {
    @Test
    public void test01(){
        LocalDate t1 = LocalDate.now();
        LocalDate t2 = LocalDate.of(2018, 12, 31);
        Period between = Period.between(t1, t2);
        System.out.println(between);

        System.out.println("相差的年数: "+between.getYears());
        System.out.println("相差的月数: "+between.getMonths());
        System.out.println("相差的天数: "+between.getDays());
        System.out.println("相差的总数: "+between.toTotalMonths());
    }

    @Test
    public void test02(){
        LocalDateTime t1 = LocalDateTime.now();

```

```

        LocalDateTime t2 = LocalDateTime.of(2017, 8, 29, 0, 0, 0);
        Duration between = Duration.between(t1, t2);
        System.out.println(between);

        System.out.println("相差的总天数: "+between.toDays());
        System.out.println("相差的总小时数: "+between.toHours());
        System.out.println("相差的总分钟数: "+between.toMinutes());
        System.out.println("相差的总秒数: "+between.getSeconds());
        System.out.println("相差的总毫秒数: "+between.toMillis());
        System.out.println("相差的总纳秒数: "+between.toNanos());
        System.out.println("不够一秒的纳秒数: "+between.getNano());
    }
}

```

4、DateTimeFormatter：日期时间格式化

该类提供了三种格式化方法：

预定义的标准格式。如：ISO_DATE_TIME;ISO_DATE

本地化相关的格式。如：ofLocalizedDate(FormatStyle.MEDIUM)

自定义的格式。如：ofPattern("yyyy-MM-dd hh:mm:ss")

```

import org.junit.Test;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;

public class TestDatetimeFormatter {
    @Test
    public void test1(){
        LocalDateTime now = LocalDateTime.now();
        //      DateTimeFormatter df =
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG); //2019年6月6日 下午04时40分03秒
        //      DateTimeFormatter df =
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM); //2019-6-6 16:40:37
        //      DateTimeFormatter df =
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT); //19-6-6 下午4:40
        DateTimeFormatter df =
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL).withZone(ZoneId.systemDefault());
        String str = df.format(now);
        System.out.println(str);
    }

    @Test
    public void test2(){
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter df = DateTimeFormatter.ISO_DATE_TIME; //2019-06-06T16:38:23.756
        String str = df.format(now);
        System.out.println(str);
    }
}

```

```

    }

    @Test
    public void test3(){
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy年MM月dd日 HH时mm分ss秒  sss毫
秒 E 是这一年的D天");
        String str = df.format(now);
        System.out.println(str);
    }
}

```

10.3 系统相关类

10.3.1 java.lang.System类

系统类中很多好用的方法，其中几个如下：

- static long currentTimeMillis()：返回当前系统时间距离1970-1-1 0:0:0的毫秒值
- static void exit(int status)：退出当前系统
- static void gc()：运行垃圾回收器。
- static String getProperty(String key): 获取某个系统属性，例如：java.version、user.language、user.country、file.encoding、user.name、os.version、os.name等等

```

import org.junit.Test;

public class TestSystem {
    @Test
    public void test01(){
        long time = System.currentTimeMillis();
        System.out.println("现在的系统时间距离1970年1月1日凌晨：" + time + "毫秒");

        System.exit(0);

        System.out.println("over");//不会执行
    }

    @Test
    public void test02(){
        System.out.println(System.getProperty("java.version"));
        System.out.println(System.getProperty("user.language"));
        System.out.println(System.getProperty("user.country"));
        System.out.println(System.getProperty("file.encoding"));
        System.out.println(System.getProperty("user.name"));
        System.out.println(System.getProperty("os.version"));
        System.out.println(System.getProperty("os.name"));
    }

    @Test
    public void test03() throws InterruptedException {

```

```

        for (int i=1; i <=10; i++){
            MyDemo my = new MyDemo(i);
            //每一次循环my就会指向新的对象，那么上次的对象就没有变量引用它了，就成垃圾对象
        }

        //为了看到垃圾回收器工作，我要加下面的代码，让main方法不那么快结束，因为main结束就会导致JVM退出，GC
        //也会跟着结束。
        System.gc(); //如果不调用这句代码，GC可能不工作，因为当前内存很充足，GC就觉得不着急回收垃圾对象。
        //调用这句代码，会让GC尽快来工作。
        Thread.sleep(5000);
    }
}

class MyDemo{
    private int value;

    public MyDemo(int value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "MyDemo{" + "value=" + value + '}';
    }

    //重写finalize方法，让大家看一下它的调用效果
    @Override
    protected void finalize() throws Throwable {
        // 正常重写，这里是编写清理系统内存的代码
        // 这里写输出语句是为了看到finalize()方法被调用的效果
        System.out.println(this+ "轻轻的走了，不带走一段代码....");
    }
}

```

10.3.3 java.lang.Runtime类

每个Java应用程序都有一个 `Runtime` 类实例，使应用程序能够与其运行的环境相连接。可以通过 `getRuntime` 方法获取当前运行时。应用程序不能创建自己的 `Runtime` 类实例。

`public static Runtime getRuntime()`: 返回与当前 Java 应用程序相关的运行时对象。

`public long totalMemory()`: 返回 Java 虚拟机中的内存总量。此方法返回的值可能随时间的推移而变化，这取决于主机环境。

`public long freeMemory()`: 回 Java 虚拟机中的空闲内存量。调用 `gc` 方法可能导致 `freeMemory` 返回值的增加。

```

package com.atguigu.system;

public class TestRuntime {
    public static void main(String[] args) {
        Runtime runtime = Runtime.getRuntime();
        long totalMemory = runtime.totalMemory();
        String str = "";
    }
}

```

```

        for (int i = 0; i < args.length; i++) {
            str += i;
        }
        long freeMemory = runtime.freeMemory();
        System.out.println("总内存: " + totalMemory);
        System.out.println("空闲内存: " + freeMemory);
        System.out.println("已用内存: " + (totalMemory-freeMemory));
    }
}

```

10.4 数组工具类

10.4.1 java.util.Arrays类

java.util.Arrays数组工具类，提供了很多静态方法来对数组进行操作，而且如下每一个方法都有各种重载形式，以下只列出int[]和Object[]类型的，其他类型的数组依次类推：

- 数组元素拼接
 - static String toString(int[] a)：字符串表示形式由数组的元素列表组成，括在方括号 ("[]") 中。相邻元素用字符 "," (逗号加空格) 分隔。形式为：[元素1, 元素2, 元素3。。。]
 - static String toString(Object[] a)：字符串表示形式由数组的元素列表组成，括在方括号 ("[]") 中。相邻元素用字符 "," (逗号加空格) 分隔。元素将自动调用自己从Object继承的toString方法将对象转为字符串进行拼接，如果没有重写，则返回类型@hash值，如果重写则按重写返回的字符串进行拼接。
- 数组排序
 - static void sort(int[] a)：将a数组按照从小到大进行排序
 - static void sort(int[] a, int fromIndex, int toIndex)：将a数组的[fromIndex, toIndex)部分按照升序排列
 - static void sort(Object[] a)：根据元素的自然顺序对指定对象数组按升序进行排序。
 - static void sort(T[] a, Comparator<? super T> c)：根据指定比较器产生的顺序对指定对象数组进行排序。
- 数组元素的二分查找
 - static int binarySearch(int[] a, int key)
 - static int binarySearch(Object[] a, Object key)：要求数组有序，在数组中查找key是否存在，如果存在返回第一次找到的下标，不存在返回负数
- 数组的复制
 - static int[] copyOf(int[] original, int newLength)：根据original原数组复制一个长度为newLength的新数组，并返回新数组
 - static T[] copyOf(T[] original,int newLength): 根据original原数组复制一个长度为newLength的新数组，并返回新数组
 - static int[] copyOfRange(int[] original, int from, int to)：复制original原数组的[from,to)构成新数组，并返回新数组
 - static T[] copyOfRange(T[] original,int from,int to): 复制original原数组的[from,to)构成新数组，并返回新数组
- 比较两耳数组是否相等
 - static boolean equals(int[] a, int[] a2)：比较两个数组的长度、元素是否完全相同
 - static boolean equals(Object[] a, Object[] a2)：比较两个数组的长度、元素是否完全相同
- 填充数组
 - static void fill(int[] a, int val)：用val值填充整个a数组

- static void fill(Object[] a, Object val): 用val对象填充整个a数组
- static void fill(int[] a, int fromIndex, int toIndex, int val): 将a数组[fromIndex,toIndex)部分填充为val值
- static void fill(Object[] a, int fromIndex, int toIndex, Object val): 将a数组[fromIndex,toIndex)部分填充为val对象

```

import org.junit.Test;

import java.text.Collator;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Locale;

public class TestArrays {
    @Test
    public void test01() {
        int[] arr = {1, 2, 3, 4, 5};
        System.out.println(Arrays.toString(arr));

        Student[] students = new Student[3];
        students[0] = new Student("张三", 96);
        students[1] = new Student("李四", 85);
        students[2] = new Student("王五", 98);
        System.out.println(Arrays.toString(students));
    }

    @Test
    public void test02() {
        int[] arr = {3, 2, 5, 1, 6};
        System.out.println("排序前" + Arrays.toString(arr));
        Arrays.sort(arr);
        System.out.println("排序后" + Arrays.toString(arr));

        Student[] students = new Student[3];
        students[0] = new Student("张三", 96);
        students[1] = new Student("李四", 85);
        students[2] = new Student("王五", 98);

        System.out.println(Arrays.toString(students));
        Arrays.sort(students);
        System.out.println(Arrays.toString(students));
        Arrays.sort(students, new Comparator() {
            @Override
            public int compare(Object o1, Object o2) {
                return Collator.getInstance(Locale.CHINA).compare(((Student) o1).getName(),
                        ((Student) o2).getName());
            }
        });
        System.out.println(Arrays.toString(students));
    }

    @Test
    public void test03() {
        int[] arr1 = {1, 2, 3, 4, 5};
    }
}

```

```

        int[] arr2 = {1, 2, 3, 4, 5};
        System.out.println(Arrays.equals(arr1, arr2));
    }

    @Test
    public void test04() {
        int[] arr1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int[] arr2 = Arrays.copyOf(arr1, 5);
        int[] arr3 = Arrays.copyOfRange(arr1, 3, 8);
        System.out.println(Arrays.toString(arr2));
        System.out.println(Arrays.toString(arr3));

        Arrays.fill(arr1, 5, 9, 3);
        System.out.println(Arrays.toString(arr1));
    }

}

```

```

package com.atguigu.arrays;

public class Student implements Comparable {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    public String getName() {
        return name;
    }

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

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
            "name='" + name + '\'' +
            ", score=" + score +
            '}';
    }
}

```

```
@Override  
public int compareTo(Object o) {  
    return this.score - ((Student)o).score;  
}  
}
```

10.4.2 java.lang.System类

系统类中很多好用的方法，其中几个如下：

- static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length):

从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。常用于数组的插入和删除

```
import org.junit.Test;  
  
import java.util.Arrays;  
  
public class TestSystemArrayCopy {  
    @Test  
    public void test01(){  
        int[] arr1 = {1,2,3,4,5};  
        int[] arr2 = new int[10];  
        System.arraycopy(arr1,0,arr2,3,arr1.length);  
        System.out.println(Arrays.toString(arr1));  
        System.out.println(Arrays.toString(arr2));  
    }  
  
    @Test  
    public void test02(){  
        int[] arr = {1,2,3,4,5};  
        System.arraycopy(arr,0,arr,1,arr.length-1);  
        System.out.println(Arrays.toString(arr));  
    }  
  
    @Test  
    public void test03(){  
        int[] arr = {1,2,3,4,5};  
        System.arraycopy(arr,1,arr,0,arr.length-1);  
        System.out.println(Arrays.toString(arr));  
    }  
}
```

10.4.3 数组的算法升华

1、数组的扩容

(1) 至于新数组的长度定义多少合适，看实际情况，如果新增的元素个数确定，那么可以增加指定长度，如果新增元素个数不确定，那么可以扩容为原来的1.5倍、2倍等

(2) 数组扩容太多会造成浪费，太少会导致频繁扩容，效率低下

```
import org.junit.Test;

import java.util.Arrays;

public class TestArrayExpand {
    @Test
    public void test01(){
        int[] arr = {1,2,3,4,5};
        arr = Arrays.copyOf(arr, arr.length+1);
        System.out.println(Arrays.toString(arr));
    }

    @Test
    public void test02(){
        int[] arr = {1,2,3,4,5};
        // arr = Arrays.copyOf(arr, (int)(arr.length*1.5));
        arr = Arrays.copyOf(arr, arr.length + (arr.length>>1));
        System.out.println(Arrays.toString(arr));
    }

    @Test
    public void test03(){
        int[] arr = {1,2,3,4,5};
        arr = Arrays.copyOf(arr, arr.length << 1);
        System.out.println(Arrays.toString(arr));
    }
}
```

2、删除数组[index]位置的元素

```
package com.atguigu.arrays;

import org.junit.Test;

import java.util.Arrays;

public class TestArrayRemove {
    @Test
    public void test01(){
        String[] strings = {"hello","java","world","atguigu","chai"};
        int total = strings.length;

        //删除[0]位置元素
        int index = 0;
        System.arraycopy(strings, index+1,strings, index, total-index-1);
        strings[--total] = null;
        System.out.println(Arrays.toString(strings));

        //删除[2]位置元素
        index = 3;
```

```
        System.arraycopy(strings, index+1, strings, index, total-index-1);;
        strings[--total] = null;
        System.out.println(Arrays.toString(strings));
    }
}
```

3、数组[index]位置插入新元素

```
package com.atguigu.arrays;

import org.junit.Test;

import java.util.Arrays;

public class TestArrayInsert {
    @Test
    public void test01(){
        String[] strings = {"hello", "java", "world", null, null};
        int total = 3;
        //在[0]位置插入"haha"
        int index = 0;
        System.arraycopy(strings, index, strings, index+1, total-index);
        total++;
        strings[index] = "haha";
        System.out.println(Arrays.toString(strings));
    }
}
```

10.5 字符串

`java.lang.String` 类代表字符串。Java程序中所有的字符串文字（例如 "abc"）都可以被看作是实现此类的实例。字符串是常量；它们的值在创建之后不能更改。字符串缓冲区支持可变的字符串。因为 String 对象是不可变的，所以可以共享。

`String` 类包括的方法可用于检查序列的单个字符、比较字符串、搜索字符串、提取子字符串、创建字符串副本并将所有字符全部转换为大写或小写。

Java 语言提供对字符串串联符号 ("+") 以及将其他对象转换为字符串的特殊支持 (`toString()`方法)。

10.5.1 字符串的特点

- 1、字符串String类型本身是final声明的，意味着我们不能继承String。
- 2、字符串的对象也是不可变对象，意味着一旦进行修改，就会产生新对象

我们修改了字符串后，如果想要获得新的内容，必须重新接收。

如果程序中涉及到大量的字符串的修改操作，那么此时的时空消耗比较高。可能需要考虑使用`StringBuilder`或`StringBuffer`的可变字符序列。

3、String对象内部是用字符数组进行保存的

JDK1.9之前有一个char[] value数组，JDK1.9之后byte[]数组

"abc" 等效于 char[] data={ 'a' , 'b' , 'c' }。

例如：

```
String str = "abc";
```

相当于：

```
char data[] = {'a', 'b', 'c'};  
String str = new String(data);  
// String底层是靠字符数组实现的。
```

4、String类中这个char[] values数组也是final修饰的，意味着这个数组不可变，然后它是private修饰，外部不能直接操作它，String类型提供的所有的方法都是用新对象来表示修改后内容的，所以保证了String对象的不可变。

5、就因为字符串对象设计为不可变，那么所以字符串有常量池来保存很多常量对象

常量池在方法区。

如果细致的划分：

- (1) JDK1.6及其之前：方法区
- (2) JDK1.7：堆
- (3) JDK1.8：元空间

```
String s1 = "abc";  
String s2 = "abc";  
System.out.println(s1 == s2);  
// 内存中只有一个"abc"对象被创建，同时被s1和s2共享。
```

10.5.2 构造字符串对象

1、使用构造方法

- `public String()`：初始化新创建的 String对象，以使其表示空字符序列。
- `String(String original)`：初始化一个新创建的 `String` 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。
- `public String(char[] value)`：通过当前参数中的字符数组来构造新的String。
- `public String(char[] value, int offset, int count)`：通过字符数组的一部分来构造新的String。
- `public String(byte[] bytes)`：通过使用平台的默认字符集解码当前参数中的字节数组来构造新的String。
- `public String(byte[] bytes, String charsetName)`：通过使用指定的字符集解码当前参数中的字节数组来构造新的String。

构造举例，代码如下：

```
//字符串常量对象  
String str = "hello";
```

```

// 无参构造
String str1 = new String();

//创建"hello"字符串常量的副本
String str2 = new String("hello");

//通过字符数组构造
char chars[] = {'a', 'b', 'c', 'd', 'e'};
String str3 = new String(chars);
String str4 = new String(chars,0,3);

// 通过字节数组构造
byte bytes[] = {97, 98, 99 };
String str5 = new String(bytes);
String str6 = new String(bytes,"GBK");

```

2、使用静态方法

- static String copyValueOf(char[] data): 返回指定数组中表示该字符序列的 String
- static String copyValueOf(char[] data, int offset, int count): 返回指定数组中表示该字符序列的 String
- static String valueOf(char[] data) : 返回指定数组中表示该字符序列的 String
- static String valueOf(char[] data, int offset, int count) : 返回指定数组中表示该字符序列的 String
- static String valueOf(xx value): xx支持各种数据类型, 返回各种数据类型的value参数的字符串表示形式。

```

public static void main(String[] args) {
    char[] data = {'h','e','l','l','o','j','a','v','a'};
    String s1 = String.copyValueOf(data);
    String s2 = String.copyValueOf(data,0,5);
    int num = 123456;
    String s3 = String.valueOf(num);
    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s3);
}

```

3、使用""+

任意数据类型与"字符串"进行拼接, 结果都是字符串

```

public static void main(String[] args) {
    int num = 123456;
    String s = num + "";
    System.out.println(s);

    Student stu = new Student();
    String s2 = stu + "";//自动调用对象的toString(), 然后与""进行拼接
    System.out.println(s2);
}

```

10.5.3 字符串的对象的个数

1、字符串常量对象

```
String str1 = "hello";//1个, 在常量池中
```

2、字符串的普通对象和常量对象一起

```
String str3 = new String("hello");
//str3首先指向堆中的一个字符串对象, 然后堆中字符串的value数组指向常量池中常量对象的value数组
```

3、面试题

```
String str1 = "hello";
String str2 = new String("hello");

//上面的代码一共有几个字符串对象。
//2个
```

10.5.4 字符串对象的内存分析

```
String s;

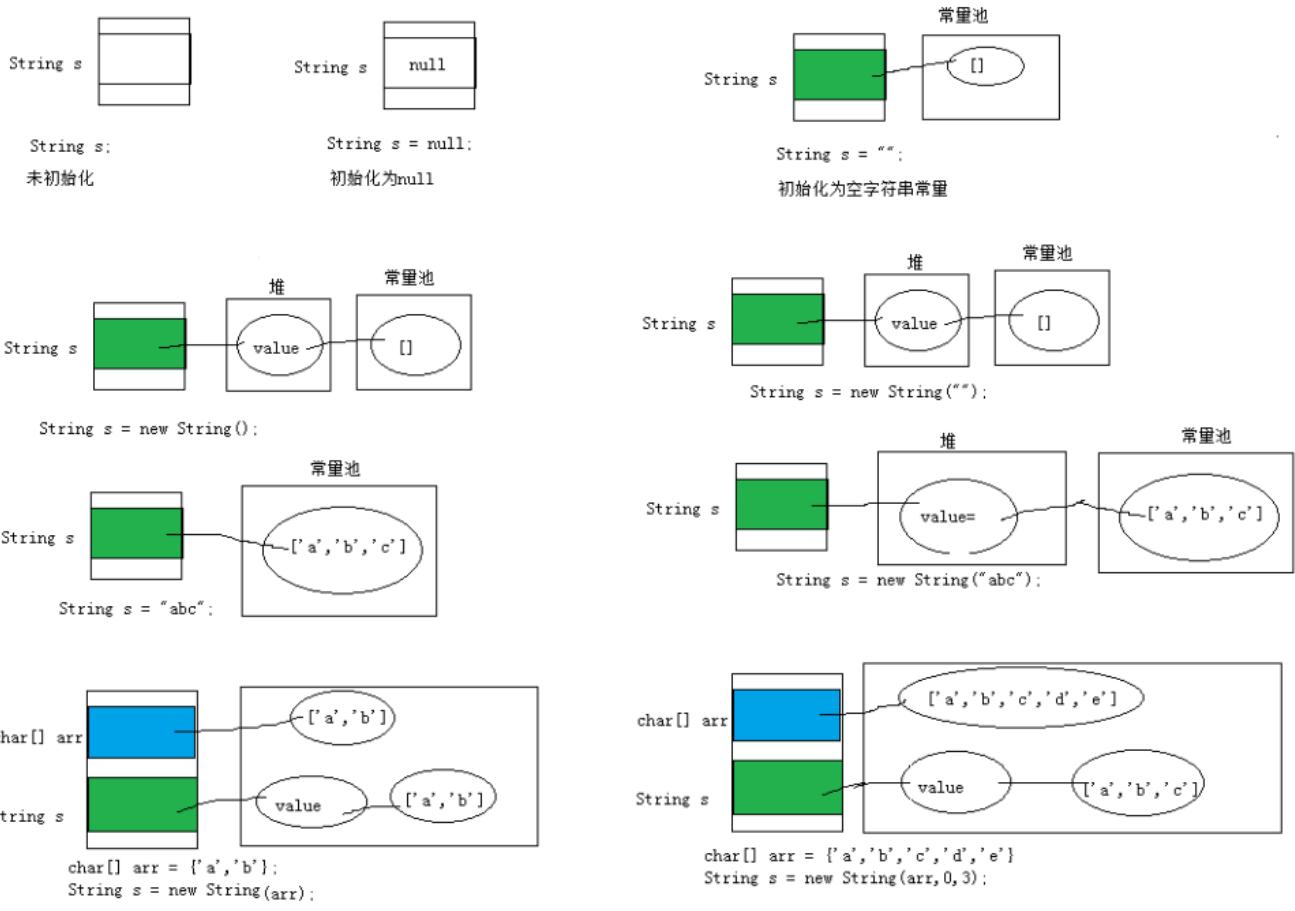
String s = null;

String s = "";
String s = new String();
String s = new String("");

String s = "abc";
String s = new String("abc");

char[] arr = {'a','b'};
String s = new String(arr);

char[] arr = {'a','b','c','d','e'};
String s = new String(arr,0,3);
```



10.5.5 字符串拼接问题

1、拼接结果的存储和比较问题

原则：

- (1) 常量+常量：结果是常量池
- (2) 常量与变量 或 变量与变量：结果是堆
- (3) 拼接后调用intern方法：结果在常量池

```

    @Test
    public void test06(){
        String s1 = "hello";
        String s2 = "world";
        String s3 = "helloworld";

        String s4 = (s1 + "world").intern();//把拼接的结果放到常量池中
        String s5 = (s1 + s2).intern();

        System.out.println(s3 == s4);//true
        System.out.println(s3 == s5);//true
    }

```

```

public void test05(){
    final String s1 = "hello";
    final String s2 = "world";
    String s3 = "helloworld";

    String s4 = s1 + "world";//s4字符串内容也helloworld, s1是常量, "world"常量, 常量+
常量 结果在常量池中
    String s5 = s1 + s2;//s5字符串内容也helloworld, s1和s2都是常量, 常量+ 常量 结果在常
量池中
    String s6 = "hello" + "world";//常量+ 常量 结果在常量池中, 因为编译期间就可以确定结果

    System.out.println(s3 == s4);//true
    System.out.println(s3 == s5);//true
    System.out.println(s3 == s6);//true
}

@Test
public void test04(){
    String s1 = "hello";
    String s2 = "world";
    String s3 = "helloworld";

    String s4 = s1 + "world";//s4字符串内容也helloworld, s1是变量, "world"常量, 变量 +
常量的结果在堆中
    String s5 = s1 + s2;//s5字符串内容也helloworld, s1和s2都是变量, 变量 + 变量的结果在
堆中
    String s6 = "hello" + "world";//常量+ 常量 结果在常量池中, 因为编译期间就可以确定结果

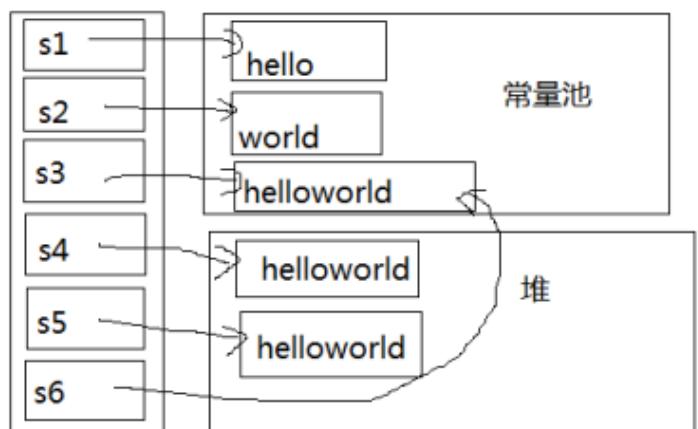
    System.out.println(s3 == s4);//false
    System.out.println(s3 == s5);//false
    System.out.println(s3 == s6);//true
}

```

```

String s1 = "hello";
String s2 = "world";
String s3 = "hello" + "world";
String s4 = s1 + "world";
String s5 = s1 + s2;
String s6 = (s1 + s2).intern();
System.out.println(s3==s4);//false
System.out.println(s3==s5);//false
System.out.println(s4==s5);//false
System.out.println(s3==s6);//true

```

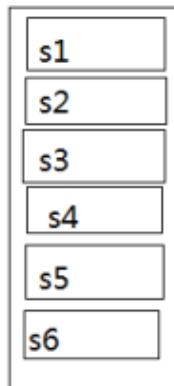


```

final String s1 = "hello";
final String s2 = "world";
String s3 = "hello" + "world";
String s4 = s1 + "world";
String s5 = s1 + s2;
String s6 = (s1 + s2).intern();

System.out.println(s3==s4); //true
System.out.println(s3==s5); //true
System.out.println(s4==s5); //true
System.out.println(s3==s6); //true

```



2、拼接效率问题

```

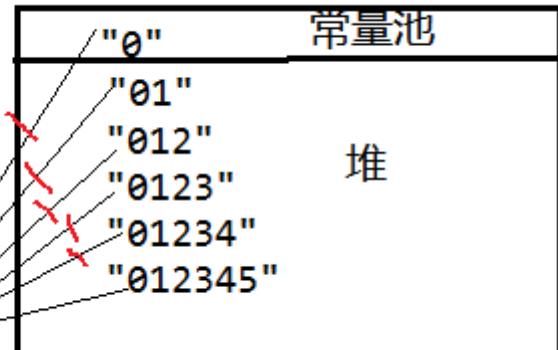
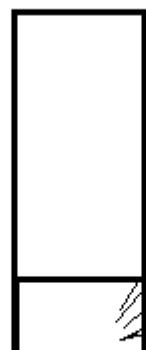
public class TestString {
    public static void main(String[] args) {
        String str = "0";
        for (int i = 0; i <= 5; i++) {
            str += i;
        }
        System.out.println(str);
    }
}

```

```

String s = "0";
for(int i=1;i<=5;i++){
    s += i;
}
System.out.println(s);

```



不过现在的JDK版本，都会使用可变字符序列对如上代码进行优化，我们反编译查看字节码：

```
javap -c TestString.class
```

```

C:\Users\Irene\Desktop>javap -c TestString.class
Compiled from "TestString.java"
public class TestString {
    public TestString();
    Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object.<init>:()V
        4: return

    public static void main(java.lang.String[]);
    Code:
        0: ldc           #2                  // String 0
        2: astore_1
        3: iconst_0
        4: istore_2
        5: iload_2
        6: iconst_5
        7: if_icmpgt   35
        10: new          #3                  // class java/lang/StringBuilder
        13: dup
        14: invokespecial #4                  // Method java/lang/StringBuilder.<init>:()V
        17: aload_1
        18: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        21: iload_2
        22: invokevirtual #6                  // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
        25: invokevirtual #7                  // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
        28: astore_1
        29: iinc         2, 1
        32: goto         5
        35: getstatic    #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
        38: aload_1
        39: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        42: return
}

```

3、两种拼接

```

public class TestString {
    public static void main(String[] args) {
        String str = "hello";
        String str2 = "world";
        String str3 = "helloworld";

        String str4 = "hello".concat("world");
        String str5 = "hello"+ "world";

        System.out.println(str3 == str4); //false
        System.out.println(str3 == str5); //true
    }
}

```

concat方法拼接，哪怕是两个常量对象拼接，结果也是在堆。

10.5.6 字符串对象的比较

1、`==`: 比较是对象的地址

只有两个字符串变量都是指向字符串的常量对象时，才会返回true

```
String str1 = "hello";
String str2 = "hello";
System.out.println(str1 == str2); //true

String str3 = new String("hello");
String str4 = new String("hello");
System.out.println(str1 == str4); //false
System.out.println(str3 == str4); //false
```

2、equals: 比较是对象的内容，因为String类型重写equals，区分大小写

只要两个字符串的字符内容相同，就会返回true

```
String str1 = "hello";
String str2 = "hello";
System.out.println(str1.equals(str2)); //true

String str3 = new String("hello");
String str4 = new String("hello");
System.out.println(str1.equals(str3)); //true
System.out.println(str3.equals(str4)); //true
```

3、equalsIgnoreCase: 比较的是对象的内容，不区分大小写

```
String str1 = new String("hello");
String str2 = new String("HELLO");
System.out.println(str1.equalsIgnoreCase(str2)); //true
```

4、compareTo: String类型重写了Comparable接口的抽象方法，自然排序，按照字符的Unicode编码值进行比较大小的，严格区分大小写

```
String str1 = "hello";
String str2 = "world";
str1.compareTo(str2) //小于0的值
```

5、compareToIgnoreCase: 不区分大小写，其他按照字符的Unicode编码值进行比较大小

```
String str1 = new String("hello");
String str2 = new String("HELLO");
str1.compareToIgnoreCase(str2) //等于0
```

10.5.7 空字符串的比较

1、哪些是空字符串

```
String str1 = "";
String str2 = new String();
String str3 = new String("");
```

空字符串：长度为0

2、如何判断某个字符串是否是空字符串

```
if("".equals(str))  
  
if(str!=null && str.isEmpty())  
  
if(str!=null && str.equals(""))  
  
if(str!=null && str.length()==0)
```

10.5.8 字符串的常用方法

1、系列1：常用方法

- (1) boolean isEmpty(): 字符串是否为空
- (2) int length(): 返回字符串的长度
- (3) String concat(xx): 拼接，等价于+
- (4) boolean equals(Object obj): 比较字符串是否相等，区分大小写
- (5) boolean equalsIgnoreCase(Object obj): 比较字符串是否相等，不区分大小写
- (6) int compareTo(String other): 比较字符串大小，区分大小写，按照Unicode编码值比较大小
- (7) int compareIgnoreCase(String other): 比较字符串大小，不区分大小写
- (8) String toLowerCase(): 将字符串中大写字母转为小写
- (9) String toUpperCase(): 将字符串中小写字母转为大写
- (10) String trim(): 去掉字符串前后空白符
- (11) public String intern(): 结果在常量池中共享

```
@Test  
public void test01(){  
    //将用户输入的单词全部转为小写，如果用户没有输入单词，重新输入  
    Scanner input = new Scanner(System.in);  
    String word;  
    while(true){  
        System.out.print("请输入单词: ");  
        word = input.nextLine();  
        if(word.trim().length()!=0){  
            word = word.toLowerCase();  
            break;  
        }  
    }  
    System.out.println(word);  
}  
  
@Test  
public void test02(){
```

```

//随机生成验证码，验证码由0-9, A-Z,a-z的字符组成
char[] array = new char[26*2+10];
for (int i = 0; i < 10; i++) {
    array[i] = (char)('0' + i);
}
for (int i = 10,j=0; i < 10+26; i++,j++) {
    array[i] = (char)('A' + j);
}
for (int i = 10+26,j=0; i < array.length; i++,j++) {
    array[i] = (char)('a' + j);
}
String code = "";
Random rand = new Random();
for (int i = 0; i < 4; i++) {
    code += array[rand.nextInt(array.length)];
}
System.out.println("验证码: " + code);
//将用户输入的单词全部转为小写，如果用户没有输入单词，重新输入
Scanner input = new Scanner(System.in);
System.out.print("请输入验证码: ");
String inputCode = input.nextLine();

if(!code.equalsIgnoreCase(inputCode)){
    System.out.println("验证码输入不正确");
}
}

```

2、系列2：查找

- (11) boolean contains(xx): 是否包含xx
- (12) int indexOf(xx): 从前往后找当前字符串中xx，即如果有返回第一次出现的下标，要是没有返回-1
- (13) int lastIndexOf(xx): 从后往前找当前字符串中xx，即如果有返回最后一次出现的下标，要是没有返回-1

```

@Test
public void test01(){
    String str = "尚硅谷是一家靠谱的培训机构，尚硅谷可以说是IT培训的小清华，JavaEE是尚硅谷的当家学科，尚硅谷的大数据培训是行业独角兽。尚硅谷的前端和运维专业一样独领风骚。";
    System.out.println("是否包含清华: " + str.contains("清华"));
    System.out.println("培训出现的第一次下标: " + str.indexOf("培训"));
    System.out.println("培训出现的最后一次下标: " + str.lastIndexOf("培训"));
}

```

3、系列3：字符串截取

- (14) String substring(int beginIndex) : 返回一个新的字符串，它是此字符串的从beginIndex开始截取到最后的一个子字符串。
- (15) String substring(int beginIndex, int endIndex) : 返回一个新字符串，它是此字符串从beginIndex开始截取到endIndex(不包含)的一个子字符串。

```

@Test
public void test01(){
    String str = "helloworldjavaatguigu";
    String sub1 = str.substring(5);
    String sub2 = str.substring(5,10);
    System.out.println(sub1);
    System.out.println(sub2);
}

@Test
public void test02(){
    String fileName = "快速学习Java的秘诀.dat";
    //截取文件名
    System.out.println("文件名: " +
fileName.substring(0,fileName.lastIndexOf(".")));
    //截取后缀名
    System.out.println("后缀名: " +
fileName.substring(fileName.lastIndexOf(".")));
}

```

4、系列4：和字符相关

- (16) char charAt(index): 返回[index]位置的字符
- (17) char[] toCharArray(): 将此字符串转换为一个新的字符数组返回
- (18) String(char[] value): 返回指定数组中表示该字符序列的 String。
- (19) String(char[] value, int offset, int count): 返回指定数组中表示该字符序列的 String。
- (20) static String copyValueOf(char[] data): 返回指定数组中表示该字符序列的 String
- (21) static String copyValueOf(char[] data, int offset, int count): 返回指定数组中表示该字符序列的 String
- (22) static String valueOf(char[] data, int offset, int count) : 返回指定数组中表示该字符序列的 String
- (23) static String valueOf(char[] data) : 返回指定数组中表示该字符序列的 String

```

@Test
public void test01(){
    //将字符串中的字符按照大小顺序排列
    String str = "helloworldjavaatguigu";
    char[] array = str.toCharArray();
    Arrays.sort(array);
    str = new String(array);
    System.out.println(str);
}

@Test
public void test02(){
    //将首字母转为大写
    String str = "jack";
    str = Character.toUpperCase(str.charAt(0))+str.substring(1);
    System.out.println(str);
}

```

5、系列5：编码与解码

(24) byte[] getBytes(): 编码，把字符串变为字节数组，按照平台默认的字符编码方式进行编码

byte[] getBytes(字符编码方式): 按照指定的编码方式进行编码

(25) new String(byte[]) 或 new String(byte[], int, int): 解码，按照平台默认的字符编码进行解码

new String(byte[], 字符编码方式) 或 new String(byte[], int, int, 字符编码方式): 解码，按照指定的编码方式进行解码

== (编码方式见附录10.7.1) ==

3. 请指出下列程序片段的输出结果(8分)

```
public static void main(String[] args) throws Exception {
    String str = "中国";
    System.out.println(str.getBytes("UTF-8").length);
    System.out.println(str.getBytes("GBK").length);
    System.out.println(str.getBytes("ISO-8859-1").length);
    System.out.println(new String(str.getBytes("ISO-8859-1"),
"ISO-8859-1"));
    System.out.println(new String(str.getBytes("UTF-8"),
"UTF-8"));
    System.out.println(new String(str.getBytes("GBK"), "GBK"));
}
```

```
package com.atguigu.string;

import org.junit.Test;

public class StringMethod5 {
    @Test
    public void test01() throws Exception{
        byte[] data = {(byte)0B11100101, (byte)0B10110000, (byte)0B10011010,
        (byte)0B11000111, (byte)0B10101011,(byte)0B01110110;
        System.out.println(new String(data, "ISO8859-1"));
        System.out.println(new String(data, "GBK"));
        System.out.println(new String(data, "UTF-8"));
    }

    @Test
    public void test02() throws Exception {
        String str = "中国";
        System.out.println(str.getBytes("ISO8859-1").length); // 2
        // ISO8859-1把所有的字符都当做一个byte处理，处理不了多个字节
        System.out.println(str.getBytes("GBK").length); // 4 每一个中文都是对应2个字节
        System.out.println(str.getBytes("UTF-8").length); // 6 常规的中文都是3个字节
    }
}
```

```

/*
 * 不乱码：（1）保证编码与解码的字符集名称一样（2）不缺字节
 */
System.out.println(new String(str.getBytes("ISO8859-1"), "ISO8859-1"));// 乱码
System.out.println(new String(str.getBytes("GBK"), "GBK"));// 中国
System.out.println(new String(str.getBytes("UTF-8"), "UTF-8"));// 中国
}
}

```

6、系列6：开头与结尾

(26) boolean startsWith(xx): 是否以xx开头

(27) boolean endsWith(xx): 是否以xx结尾

```

@Test
public void test2(){
    String name = "张三";
    System.out.println(name.startsWith("张"));
}

@Test
public void test(){
    String file = "Hello.txt";
    if(file.endsWith(".java")){
        System.out.println("Java源文件");
    }else if(file.endsWith(".class")){
        System.out.println("Java字节码文件");
    }else{
        System.out.println("其他文件");
    }
}

```

7、系列7：正则匹配

(28) boolean matches(正则表达式): 判断当前字符串是否匹配某个正则表达式。== (正则表达式见附录10.7.2)
==

```

@Test
public void test1(){
    //简单判断是否全部是数字，这个数字可以是1~n位
    String str = "12a345";

    //正则不是Java的语法，它是独立与Java的规则
    //在正则中\是表示转义,
    //同时在Java中\也是转义
    boolean flag = str.matches("\d+");
    System.out.println(flag);
}

@Test
public void test2(){
}

```

```

        string str = "123456789";

        //判断它是否全部由数字组成，并且第1位不能是0，长度为9位
        //第一位不能是0，那么数字[1-9]
        //接下来8位的数字，那么[0-9]{8}+
        boolean flag = str.matches("[1-9][0-9]{8}");
        System.out.println(flag);
    }

    @Test
    public void test03(){
        //密码要求：必须有大写字母，小写字母，数字组成，6位
        System.out.println("cly892".matches("^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9])[A-Za-z0-9]{6}$")); //true
        System.out.println("1A2c45".matches("^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9])[A-Za-z0-9]{6}$")); //true
        System.out.println("clyyyy".matches("^(?=.*[A-Z])(?=.*[0-9])[A-Za-z0-9]{6}$")); //false
    }

```

8、系列8：替换

- (29) String replace(xx,xx): 不支持正则
- (30) String replaceFirst(正则, value): 替换第一个匹配部分
- (31) String repalceAll(正则, value): 替换所有匹配部分

```

    @Test
    public void test4(){
        String str = "hello244world.java;887";
        //把其中的非字母去掉
        str = str.replaceAll("[^a-zA-Z]", " ");
        System.out.println(str);
    }

```

9、系列9：拆分

- (32) String[] split(正则): 按照某种规则进行拆分

```

    @Test
    public void test4(){
        String str = "张三.23|李四.24|王五.25";
        //|在正则中是有特殊意义，我这里要把它当做普通的|
        String[] all = str.split("\\|");

        //转成一个一个学生对象
        Student[] students = new Student[all.length];
        for (int i = 0; i < students.length; i++) {
            //.在正则中是特殊意义，我这里想要表示普通的。
            String[] strings = all[i].split("\\."); //张三， 23
        }
    }

```

```

        String name = strings[0];
        int age = Integer.parseInt(strings[1]);
        students[i] = new Student(name,age);
    }

    for (int i = 0; i < students.length; i++) {
        System.out.println(students[i]);
    }

}

@Test
public void test3(){
    String str = "1Hello2world3java4atguigu5";
    str = str.replaceAll("\\d|\\d$", "");
    String[] all = str.split("\\d");
    for (int i = 0; i < all.length; i++) {
        System.out.println(all[i]);
    }
}

@Test
public void test2(){
    String str = "1Hello2world3java4atguigu";
    str = str.replaceFirst("\\d", "");
    System.out.println(str);
    String[] all = str.split("\\d");
    for (int i = 0; i < all.length; i++) {
        System.out.println(all[i]);
    }
}

@Test
public void test1(){
    String str = "Hello world java atguigu";
    String[] all = str.split(" ");
    for (int i = 0; i < all.length; i++) {
        System.out.println(all[i]);
    }
}

```

10.6 可变字符序列

10.6.1 String与可变字符序列的区别

因为String对象是不可变对象，虽然可以共享常量对象，但是对于频繁字符串的修改和拼接操作，效率极低。因此，JDK又在java.lang包提供了可变字符序列StringBuilder和StringBuffer类型。

StringBuffer: 老的，线程安全的（因为它的方法有synchronized修饰）

StringBuilder: 线程不安全的

10.6.2 StringBuilder、StringBuffer的API

常用的API，StringBuilder、StringBuffer的API是完全一致的

- (1) StringBuffer append(xx): 拼接，追加
- (2) StringBuffer insert(int index, xx): 在[index]位置插入xx
- (3) StringBuffer delete(int start, int end): 删除[start,end)之间字符

StringBuffer deleteCharAt(int index): 删除[index]位置字符

- (4) void setCharAt(int index, xx): 替换[index]位置字符
- (5) StringBuffer reverse(): 反转
- (6) void setLength(int newLength): 设置当前字符序列长度为newLength
- (7) StringBuffer replace(int start, int end, String str): 替换[start,end)范围的字符序列为str
- (8) int indexOf(String str): 在当前字符序列中查询str的第一次出现下标

int indexOf(String str, int fromIndex): 在当前字符序列[fromIndex,最后]中查询str的第一次出现下标

int lastIndexOf(String str): 在当前字符序列中查询str的最后一次出现下标

int lastIndexOf(String str, int fromIndex): 在当前字符序列[fromIndex,最后]中查询str的最后一次出现下标

- (9) String substring(int start): 截取当前字符序列[start,最后]
- (10) String substring(int start, int end): 截取当前字符序列[start,end)
- (11) String toString(): 返回此序列中数据的字符串表示形式

```
@Test
public void test6(){
    StringBuilder s = new StringBuilder("helloworld");
    s.setLength(30);
    System.out.println(s);
}

@Test
public void test5(){
    StringBuilder s = new StringBuilder("helloworld");
    s.setCharAt(2, 'a');
    System.out.println(s);
}

@Test
public void test4(){
    StringBuilder s = new StringBuilder("helloworld");
    s.reverse();
    System.out.println(s);
}

@Test
public void test3(){}
```

```

        StringBuilder s = new StringBuilder("helloworld");
        s.delete(1, 3);
        s.deleteCharAt(4);
        System.out.println(s);
    }

    @Test
    public void test2(){
        StringBuilder s = new StringBuilder("helloworld");
        s.insert(5, "java");
        s.insert(5, "chailinyan");
        System.out.println(s);
    }

    @Test
    public void test1(){
        StringBuilder s = new StringBuilder();
        s.append("hello").append(true).append('a').append(12).append("atguigu");
        System.out.println(s);
        System.out.println(s.length());
    }
}

```

10.6.3 效率测试

```

package com.atguigu.stringbuffer;

import org.junit.Test;

public class TestTime {

    @Test
    public void testString(){
        long start = System.currentTimeMillis();
        String s = new String("0");
        for(int i=1;i<=10000;i++){
            s += i;
        }
        long end = System.currentTimeMillis();
        System.out.println("String拼接+用时: "+(end-start));//367

        long memory = Runtime.getRuntime().totalMemory() -
        Runtime.getRuntime().freeMemory();
        System.out.println("String拼接+memory占用内存: " + memory);//473081920字节
    }

    @Test
    public void testStringBuilder(){
        long start = System.currentTimeMillis();
        StringBuilder s = new StringBuilder("0");
        for(int i=1;i<=10000;i++){
            s.append(i);
        }
        long end = System.currentTimeMillis();
        System.out.println("StringBuilder拼接+用时: "+(end-start));//1毫秒
    }
}

```

```

    }
    long end = System.currentTimeMillis();
    System.out.println("StringBuilder拼接+用时: "+(end-start));//5
    long memory = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
    System.out.println("StringBuilder拼接+memory占用内存: " + memory);//13435032
}

@Test
public void testStringBuffer(){
    long start = System.currentTimeMillis();
    StringBuffer s = new StringBuffer("0");
    for(int i=1;i<=10000;i++){
        s.append(i);
    }
    long end = System.currentTimeMillis();
    System.out.println("StringBuffer拼接+用时: "+(end-start));//5
    long memory = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
    System.out.println("StringBuffer拼接+memory占用内存: " + memory);//13435032
}
}

```

10.7 附录

10.7.1 字符编码的发展

1、ASCII码

计算机一开始发明的时候是用来解决数字计算的问题，后来人们发现，计算机还可以做更多的事，例如文本处理。但由于计算机只识“数”，因此人们必须告诉计算机哪个数字来代表哪个特定字符，例如65代表字母‘A’，66代表字母‘B’，以此类推。但是计算机之间字符-数字的对应关系必须得一致，否则就会造成同一段数字在不同计算机上显示出来的字符不一样。因此美国国家标准协会ANSI制定了一个标准，规定了常用字符的集合以及每个字符对应的编号，这就是ASCII字符集（Character Set），也称ASCII码。

那时候的字符编解码系统非常简单，就是简单的查表过程。其中：

- 0 ~ 31及127(共33个)是控制字符或通信专用字符（其余为可显示字符），如控制符：LF（换行）、CR（回车）、FF（换页）、DEL（删除）、BS（退格）
- 32 ~ 126(共95个)是字符(32是空格)，其中48 ~ 57为0到9十个阿拉伯数字。65 ~ 90为26个大写英文字母，97 ~ 122号为26个小写英文字母，其余为一些标点符号、运算符号等。

2、OEM字符集的衍生

当计算机开始发展起来的时候，人们逐渐发现，ASCII字符集里那可怜的128个字符已经不能再满足他们的需求了。人们就在想，一个字节能够表示的数字（编号）有256个，而ASCII字符只用到了0x00~0x7F，也就是占用了前128个，后面128个数字不用白不用，因此很多人打起了后面这128个数字的主意。可是问题在于，很多人同时有这样的想法，但是大家对于0x80-0xFF这后面的128个数字分别对应什么样的字符，却有各自的想法。这就导致了当时销往世界各地的机器上出现了大量各式各样的OEM字符集。不同的OEM字符集导致人们无法跨机器交流各种文档。例如职员甲发了一封简历résumés给职员乙，结果职员乙看到的却是r?sum?s，因为é字符在职员甲机器上的OEM字符集中对应的字节是0x82，而在职员乙的机器上，由于使用的OEM字符集不同，对0x82字节解码后得到的字符却是？。

3、多字节字符集（MBCS）和中文字符集

上面我们提到的字符集都是基于单字节编码，也就是说，一个字节翻译成一个字符。这对于拉丁语系国家来说可能没有什么问题，因为他们通过扩展第8个比特，就可以得到256个字符了，足够用了。但是对于亚洲国家来说，256个字符是远远不够用的。因此这些国家的人为了用上电脑，又要保持和ASCII字符集的兼容，就发明了多字节编码方式，相应的字符集就称为多字节字符集（Multi-Bytes Character Set）。例如中国使用的就是双字节字符集编码。

例如目前最常用的中文字符集GB2312，涵盖了所有简体字符以及一部分其他字符；GBK（K代表扩展的意思）则在GB2312的基础上加入了对繁体字符等其他非简体字符。这两个字符集的字符都是使用1-2个字节来表示。Windows系统采用936代码页来实现对GBK字符集的编解码。在解析字节流的时候，如果遇到字节的最高位是0的话，那么就使用936代码页中的第1张码表进行解码，这就和单字节字符集的编解码方式一致了。如果遇到字节的最高位是1的话，那么就表示需要两个字节值才能对应一个字符。

假如你使用 GB2312 写了这么一句话：

我叫 ABC

它的二进制编码是这样的：

11001110 11010010 10111101 11010000 01000001 01000002 01000003

4、ANSI标准、国家标准、ISO标准

不同ASCII衍生字符集的出现，让文档交流变得非常困难，因此各种组织都陆续进行了标准化流程。例如美国ANSI组织制定了ANSI标准字符编码（注意，我们现在通常说到ANSI编码，通常指的是平台的默认编码，例如英文操作系统中是ISO-8859-1，中文系统是GBK），ISO组织制定的各种ISO标准字符编码，还有各国也会制定一些国家标准字符集，例如中国的GBK，GB2312和GB18030。

操作系统在发布的时候，通常会往机器里预装这些标准的字符集还有平台专用的字符集，这样只要你的文档是使用标准字符集编写的，通用性就比较高了。例如你用GB2312字符集编写的文档，在中国大陆内的任何机器上都能正确显示。同时，我们也可以在一台机器上阅读多个国家不同语言的文档了，前提是本机必须安装该文档使用的字符集。

5、Unicode的出现

虽然通过使用不同字符集，我们可以在一台机器上查阅不同语言的文档，但是我们仍然无法解决一个问题：如果一份文档中含有不同国家的不同语言的字符，那么无法在一份文档中显示所有字符。为了解决这个问题，我们需要一个全人类达成共识的巨大的字符集，这就是Unicode字符集。

Unicode字符集涵盖了目前人类使用的所有字符，并为每个字符进行统一编号，分配唯一的字符码（Code Point）。Unicode字符集将所有字符按照使用上的频繁度划分为17个层面（Plane），每个层面上有 $2^{16}=65536$ 个字符码空间。其中第0个层面BMP，基本涵盖了当今世界用到的所有字符。其他的层面要么是用来表示一些远古时期的文字，要么是留作扩展。我们平常用到的Unicode字符，一般都是位于BMP层面上的。目前Unicode字符集中尚有大量字符空间未使用。

==在内存中每一个字符使用它在Unicode字符集中的唯一编码值表示==，这是没有问题的。因为Unicode字符集中字符编码值的范围是[0, 65535]，在Java的JVM内存中无论这个字符的编码值是多少，都分配2个字节。

但是==在其他环境中==，例如文件中、IO流中等，Unicode就不完美了，这里有三个的问题，一个是，在文件或IO流中英文字母等ASCII码表中的字符只用一个字节表示，第二个问题是如何才能区别这是Unicode和ASCII，即计算机怎么知道两个字节表示一个符号，而不是分别表示两个符号呢？第三个，如果和GBK等双字节编码方式一样，用最高位是1或0表示两个字节和一个字节，就少了很多值无法用于表示字符，不够表示所有字符。Unicode在很长一段时间内无法推广，直到互联网的出现，为解决Unicode如何在网络上传输的问题，于是面向传输的众多UTF（UCS Transfer Format）标准出现了，顾名思义，UTF-8就是每次8个位传输数据，而UTF-16就是每次16个位。UTF-8就是在互联网上使用最广的一种Unicode的实现方式，这是为传输而设计的编码，并使编码无国界，这样就可以显示全世界上所有文化的字符了。

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。从unicode到uft-8并不是直接的对应，而是要过一些算法和规则来转换（即Uncidoe字符集≠UTF-8编码方式）。

Unicode符号范围 | UTF-8编码方式

(十六进制) | (二进制)

0000 0000-0000 007F | 0xxxxxxx (兼容原来的ASCII)

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

尚

Unicode编码值：23578 十六进制 5C1A 二进制 0101 1100 0001 1010

UTF-8编码方式处理

1110xxxx 10xx xxxx 10xx xxxx
1110 0101 1011 0000 1001 1010

UTF-8编码：

1110 0101 1011 0000 1001 1010

e5 b0 9a

[-27, -80, -102]

因此，Unicode只是定义了一个庞大的、全球通用的字符集，并为每个字符规定了唯一确定的编号，具体存储成什么样的字节流，取决于字符编码方案。推荐的Unicode编码是UTF-16和UTF-8。

早期字符编码、字符集和代码页等概念都是表达同一个意思。例如GB2312字符集、GB2312编码，936代码页，实际上说的是同个东西。

但是对于Unicode则不同，Unicode字符集只是定义了字符的集合和唯一编号，Unicode编码，则是对UTF-8、UCS-2/UTF-16等具体编码方案的统称而已，并不是具体的编码方案。所以当需要用到字符编码的时候，你可以写gb2312，codepage936，utf-8，utf-16，但请不要写Unicode。

10.7.2 正则表达式

正则表达式，又称规则表达式（英语：Regular Expression，在代码中常简写为regex、regexp或RE）。正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。通常被用来检索、替换那些符合某个模式(规则)的文本。

1、正则表达式构造摘要

(1) 字符类

[abc]：a、b 或 c (简单类)

[^abc]：任何字符，除了 a、b 或 c (否定)

[a-zA-Z]：a 到 z 或 A 到 Z，两头的字母包括在内 (范围)

(2) 预定义字符类

.：任何字符（与行结束符可能匹配也可能不匹配）

`\d`: 数字: [0-9]

`\D`: 非数字: [^0-9]

`\s`: 空白字符: [\t\n\x0B\f\r]

`\S`: 非空白字符: [^\s]

`\w`: 单词字符: [a-zA-Z_0-9]

`\W`: 非单词字符: [^\w]

(3) POSIX 字符类 (仅 US-ASCII)

`\p{Lower}` 小写字母字符: [a-z]

`\p{Upper}` 大写字母字符: [A-Z]

`\p{ASCII}` 所有 ASCII: [\x00-\x7F]

`\p{Alpha}` 字母字符: [\p{Lower}\p{Upper}]

`\p{Digit}` 十进制数字: [0-9]

`\p{AlphaNum}` 字母数字字符: [\p{Alpha}\p{Digit}]

`\p{Punct}` 标点符号: !"#\$%&()'*,.-./;<=>?@[]^_`{|}~

`\p{Blank}` 空格或制表符: [\t]

(4) 边界匹配器

`^`: 行的开头

`$`: 行的结尾

(5) Greedy 数量词

`X?`: X , 一次或一次也没有

`X*`: X , 零次或多次

`X+`: X , 一次或多次

`X{n}`: X , 恰好 n 次

`X{n,}`: X , 至少 n 次

`X{n,m}`: X , 至少 n 次, 但是不超过 m 次

(6) Logical 运算符

`XY`: X 后跟 Y

`X|Y`: X 或 Y

`(X)`: X , 作为捕获组

(7) 特殊构造 (非捕获)

(?:X) X, 作为非捕获组
(?=X) X, 通过零宽度的正 lookahead
(?!X) X, 通过零宽度的负 lookahead
(?<=X) X, 通过零宽度的正 lookbehind
(?<!X) X, 通过零宽度的负 lookbehind
(?>X) X, 作为独立的非捕获组

2、常见的正则表达式示例

- 验证用户名和密码，要求第一个字必须为字母，一共6~16位字母数字下划线组成： (`^[a-zA-Z]\w{5,15}$`)
- 验证电话号码：xxx/xxxx-xxxxxx/xxxxxxx： (`(^\d{3,4}-)\d{7,8}$`)
- 验证手机号码：(`^(13[0-9]|14[5|7]|15[0|1|2|3|5|6|7|8|9]|18[0|1|2|3|5|6|7|8|9])\d{8}$`)
- 验证身份证号：(`^\d{15}$|^\d{18}$|^\d{17}(\d|X|x)$`)
- 验证Email地址：(`^\w+([-.\w+])@\w+([-.\w+]).\w+([-.\w+]*$)`)
- 只能输入由数字和26个英文字母组成的字符串：(`^[A-Za-z0-9]+$`)
- 整数或者小数：(`^[0-9]+([0-9]+\{0,1\})$`)
- 中文字符的正则表达式：(`[\u4e00-\u9fa5]`)
- 金额校验(非零开头的最多带两位小数的数字)：(`^([1-9][0-9]*)(.[0-9]{1,2})?${`)
- IPV4地址：(`((\d{1,2})|(1\d{1,2})|(2[0-4]\d)|(25[0-5]))\.\{3}((\d{1,2})|(1\d{1,2})|(2[0-4]\d)|(25[0-5]))`)

第11章 集合与迭代器

11.1 Collection集合

11.1.1 集合的概念

集合是java中提供的一种容器，可以用来存储多个数据。

集合和数组既然都是容器，它们有啥区别呢？

- 数组的长度是固定的。集合的长度是可变的。
- 数组中可以存储基本数据类型值，也可以存储对象，而集合中只能存储对象

集合主要分为两大系列：Collection和Map，Collection 表示一组对象，Map表示一组映射关系或键值对。

11.1.2 Collection接口

Collection 层次结构中的根接口。Collection 表示一组对象，这些对象也称为 collection 的元素。一些 collection 允许有重复的元素，而另一些则不允许。一些 collection 是有序的，而另一些则是无序的。JDK 不提供此接口的任何直接实现：它提供更具体的子接口（如 Set 和 List、Queue）实现。此接口通常用来传递 collection，并在需要最大普遍性的地方操作这些 collection。

Collection是所有单列集合的父接口，因此在Collection中定义了单列集合(List和Set)通用的一些方法，这些方法可用于操作所有的单列集合。方法如下：

1、添加元素

(1) add(E obj): 添加元素对象到当前集合中

(2) addAll(Collection<? extends E> other): 添加other集合中的所有元素对象到当前集合中，即this = this ∪ other

2、删除元素

(1) boolean remove(Object obj) : 从当前集合中删除第一个找到的与obj对象equals返回true的元素。

(2) boolean removeAll(Collection<?> coll): 从当前集合中删除所有与coll集合中相同的元素。即this = this - this ∩ coll

(3) boolean removeIf(Predicate<? super E> filter) : 删除满足给定条件的此集合的所有元素。

(4) boolean retainAll(Collection<?> coll): 从当前集合中删除两个集合中不同的元素，使得当前集合仅保留与c集合中的元素相同的元素，即当前集合中仅保留两个集合的交集，即this = this ∩ coll;

3、查询与获取元素

(1) boolean isEmpty(): 判断当前集合是否为空集合。

(2) boolean contains(Object obj): 判断当前集合中是否存在一个与obj对象equals返回true的元素。

(3) boolean containsAll(Collection<?> c): 判断c集合中的元素是否在当前集合中都存在。即c集合是否是当前集合的“子集”。

(4) int size(): 获取当前集合中实际存储的元素个数

(5) Object[] toArray(): 返回包含当前集合中所有元素的数组

11.1.3 API演示

1、演示添加

注意：add和addAll的区别

```
package com.atguigu.collection;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;

public class TestCollectionAdd {
    @Test
    public void testAdd(){
        //ArrayList是Collection的子接口List的实现类之一。
        Collection coll = new ArrayList();
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");
        System.out.println(coll);
    }

    @Test
    public void testAddAll(){
        Collection c1 = new ArrayList();
    }
}
```

```

c1.add(1);
c1.add(2);
System.out.println("c1集合元素的个数: " + c1.size());//2
System.out.println("c1 = " + c1);

collection c2 = new ArrayList();
c2.add(1);
c2.add(2);
System.out.println("c2集合元素的个数: " + c2.size());//2
System.out.println("c2 = " + c2);

collection other = new ArrayList();
other.add(1);
other.add(2);
other.add(3);
System.out.println("other集合元素的个数: " + other.size());//3
System.out.println("other = " + other);
System.out.println();

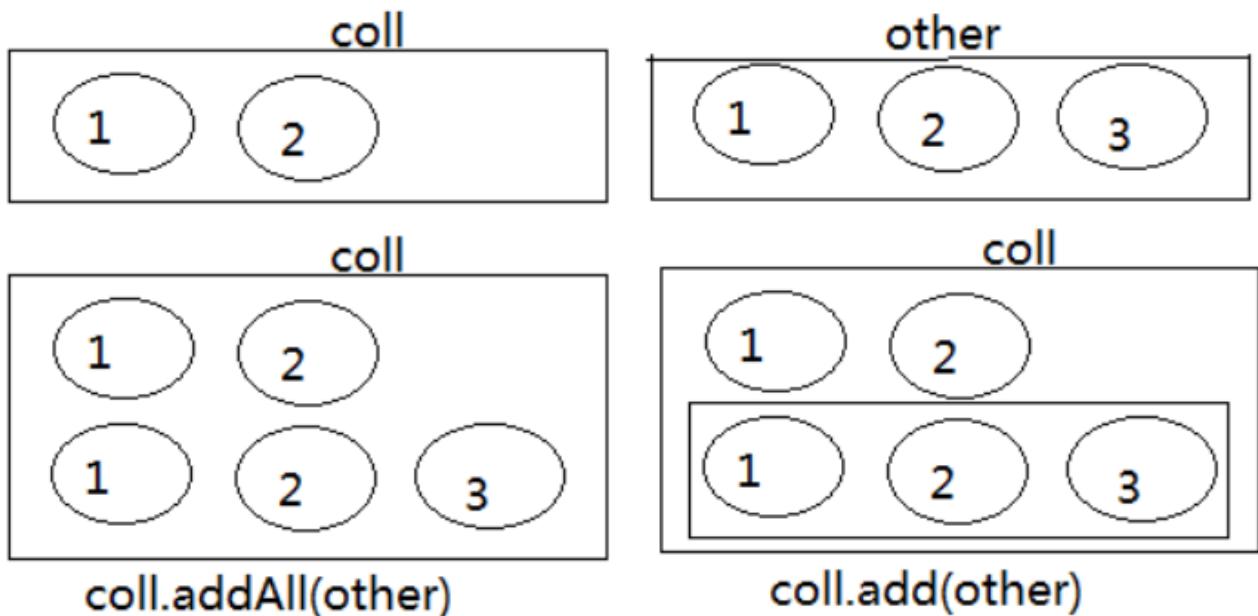
c1.addAll(other);
System.out.println("c1集合元素的个数: " + c1.size());//5
System.out.println("c1.addAll(other) = " + c1);

c2.add(other);
System.out.println("c2集合元素的个数: " + c2.size());
System.out.println("c2.add(other) = " + c2);
}

}

```

注意: coll.addAll(other);与coll.add(other);



2、演示删除

注意几种删除方法的区别

```
package com.atguigu.collection;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;
import java.util.function.Predicate;

public class TestCollectionRemove {
    @Test
    public void test01(){
        Collection coll = new ArrayList();
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");
        coll.add("佛地魔");
        System.out.println("coll = " + coll);

        coll.remove("小李广");
        System.out.println("删除元素\"小李广\"之后coll = " + coll);

        coll.removeIf(new Predicate() {
            @Override
            public boolean test(Object o) {
                String str = (String) o;
                return str.contains("地");
            }
        });
        System.out.println("删除包含\"地\"字的元素之后coll = " + coll);

        coll.clear();
        System.out.println("coll清空之后, coll = " + coll);
    }

    @Test
    public void test02() {
        Collection coll = new ArrayList();
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");
        coll.add("佛地魔");
        System.out.println("coll = " + coll);

        Collection other = new ArrayList();
        other.add("小李广");
        other.add("扫地僧");
        other.add("尚硅谷");
        System.out.println("other = " + other);

        coll.removeAll(other);
        System.out.println("coll.removeAll(other)之后, coll = " + coll);
        System.out.println("coll.removeAll(other)之后, other = " + other);
    }
}
```

```

@Test
public void test03() {
    Collection coll = new ArrayList();
    coll.add("小李广");
    coll.add("扫地僧");
    coll.add("石破天");
    coll.add("佛地魔");
    System.out.println("coll = " + coll);

    Collection other = new ArrayList();
    other.add("小李广");
    other.add("扫地僧");
    other.add("尚硅谷");
    System.out.println("other = " + other);

    coll.retainAll(other);
    System.out.println("coll.retainAll(other)之后, coll = " + coll);
    System.out.println("coll.retainAll(other)之后, other = " + other);
}

}

```

3、演示查询与获取元素

```

package com.atguigu.collection;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;

public class TestCollectionContains {
    @Test
    public void test01() {
        Collection coll = new ArrayList();
        System.out.println("coll在添加元素之前, isEmpty = " + coll.isEmpty());
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");
        coll.add("佛地魔");
        System.out.println("coll的元素个数" + coll.size());
        Object[] objects = coll.toArray();
        System.out.println("用数组返回coll中所有元素: " + Arrays.toString(objects));
        System.out.println("coll在添加元素之后, isEmpty = " + coll.isEmpty());
        coll.clear();
        System.out.println("coll在clear之后, isEmpty = " + coll.isEmpty());
    }

    @Test
    public void test02() {
        Collection coll = new ArrayList();

```

```
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");
        coll.add("佛地魔");
        System.out.println("coll = " + coll);
        System.out.println("coll是否包含“小李广” = " + coll.contains("小李广"));
        System.out.println("coll是否包含“宋红康” = " + coll.contains("宋红康"));

        Collection other = new ArrayList();
        other.add("小李广");
        other.add("扫地僧");
        other.add("尚硅谷");
        System.out.println("other = " + other);

        System.out.println("coll.containsAll(other) = " + coll.containsAll(other));
    }

    @Test
    public void test03(){
        Collection c1 = new ArrayList();
        c1.add(1);
        c1.add(2);
        System.out.println("c1集合元素的个数: " + c1.size());//2
        System.out.println("c1 = " + c1);

        Collection c2 = new ArrayList();
        c2.add(1);
        c2.add(2);
        System.out.println("c2集合元素的个数: " + c2.size());//2
        System.out.println("c2 = " + c2);

        Collection other = new ArrayList();
        other.add(1);
        other.add(2);
        other.add(3);
        System.out.println("other集合元素的个数: " + other.size());//3
        System.out.println("other = " + other);
        System.out.println();

        c1.addAll(other);
        System.out.println("c1集合元素的个数: " + c1.size());//5
        System.out.println("c1.addAll(other) = " + c1);
        System.out.println("c1.contains(other) = " + c1.contains(other));
        System.out.println("c1.containsAll(other) = " + c1.containsAll(other));
        System.out.println();

        c2.add(other);
        System.out.println("c2集合元素的个数: " + c2.size());
        System.out.println("c2.add(other) = " + c2);
        System.out.println("c2.contains(other) = " + c2.contains(other));
        System.out.println("c2.containsAll(other) = " + c2.containsAll(other));
    }
}
```

```
}
```

11.2 Iterator迭代器

11.2.1 Iterator接口

在程序开发中，经常需要遍历集合中的所有元素。针对这种需求，JDK专门提供了一个接口 `java.util.Iterator`。`Iterator` 接口也是Java集合中的一员，但它与 `Collection`、`Map` 接口有所不同，`Collection` 接口与 `Map` 接口主要用于存储元素，而 `Iterator` 主要用于迭代访问（即遍历）`Collection` 中的元素，因此 `Iterator` 对象也被称为迭代器。

想要遍历Collection集合，那么就要获取该集合迭代器完成迭代操作，下面介绍一下获取迭代器的方法：

- `public Iterator iterator()`：获取集合对应的迭代器，用来遍历集合中的元素的。

下面介绍一下迭代的概念：

- **迭代**：即Collection集合元素的通用获取方式。在取元素之前先要判断集合中有没有元素，如果有，就把这个元素取出来，继续在判断，如果还有就再取出出来。一直把集合中的所有元素全部取出。这种取出方式专业术语称为迭代。

Iterator接口的常用方法如下：

- `public E next()`：返回迭代的下一个元素。
- `public boolean hasNext()`：如果仍有元素可以迭代，则返回 true。

接下来我们通过案例学习如何使用Iterator迭代集合中元素：

```
package com.atguigu.iterator;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class TestIterator {
    @Test
    public void test01(){
        Collection coll = new ArrayList();
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");

        Iterator iterator = coll.iterator();
        System.out.println(iterator.next());
        System.out.println(iterator.next());
        System.out.println(iterator.next());
        System.out.println(iterator.next());
    }
}
```

```

@Test
public void test02(){
    Collection coll = new ArrayList();
    coll.add("小李广");
    coll.add("扫地僧");
    coll.add("石破天");

    Iterator iterator = coll.iterator(); //获取迭代器对象
    while(iterator.hasNext()) { //判断是否还有元素可迭代
        System.out.println(iterator.next()); //取出下一个元素
    }
}
}

```

提示：在进行集合元素取出时，如果集合中已经没有元素了，还继续使用迭代器的next方法，将会发生java.util.NoSuchElementException没有集合元素的错误。

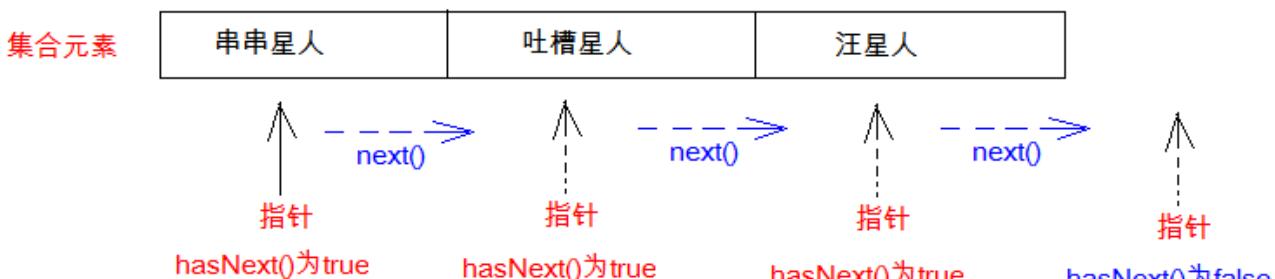
11.2.2 迭代器的实现原理

我们在之前案例已经完成了Iterator遍历集合的整个过程。当遍历集合时，首先通过调用集合的iterator()方法获得迭代器对象，然后使用hasNext()方法判断集合中是否存在下一个元素，如果存在，则调用next()方法将元素取出，否则说明已到达了集合末尾，停止遍历元素。

Iterator迭代器对象在遍历集合时，内部采用指针的方式来跟踪集合中的元素，为了让初学者能更好地理解迭代器的工作原理，接下来通过一个图例来演示Iterator对象迭代元素的过程：

迭代集合元素：

1. 指针当前位置 判断hasNext()为true
2. 执行next()获取元素，移动指针来到下一个元素位置



hasNext结果为false 迭代中止

在调用Iterator的next方法之前，迭代器指向第一个元素，当第一次调用迭代器的next方法时，返回第一个元素，然后迭代器的索引会向后移动一位，指向第二个元素，当再次调用next方法时，返回第二个元素，然后迭代器的索引会再向后移动一位，指向第三个元素，依此类推，直到hasNext方法返回false，表示到达了集合的末尾，终止对元素的遍历。

11.2.3 Iterable接口与Iterator接口

Java5 (JDK1.5) 中增加了java.lang.Iterable接口，实现这个接口允许对象成为 "foreach" 语句的目标。Java 5时Collection接口继承了java.lang.Iterable接口，因此Collection系列的集合就可以直接使用foreach循环遍历。

java.lang.Iterable接口的抽象方法：

- public Iterator iterator(): 获取对应的迭代器，用来遍历数组或集合中的元素的。

从上面的方法定义可以看出，其实foreach循环其实就是使用Iterator迭代器来完成元素的遍历的。

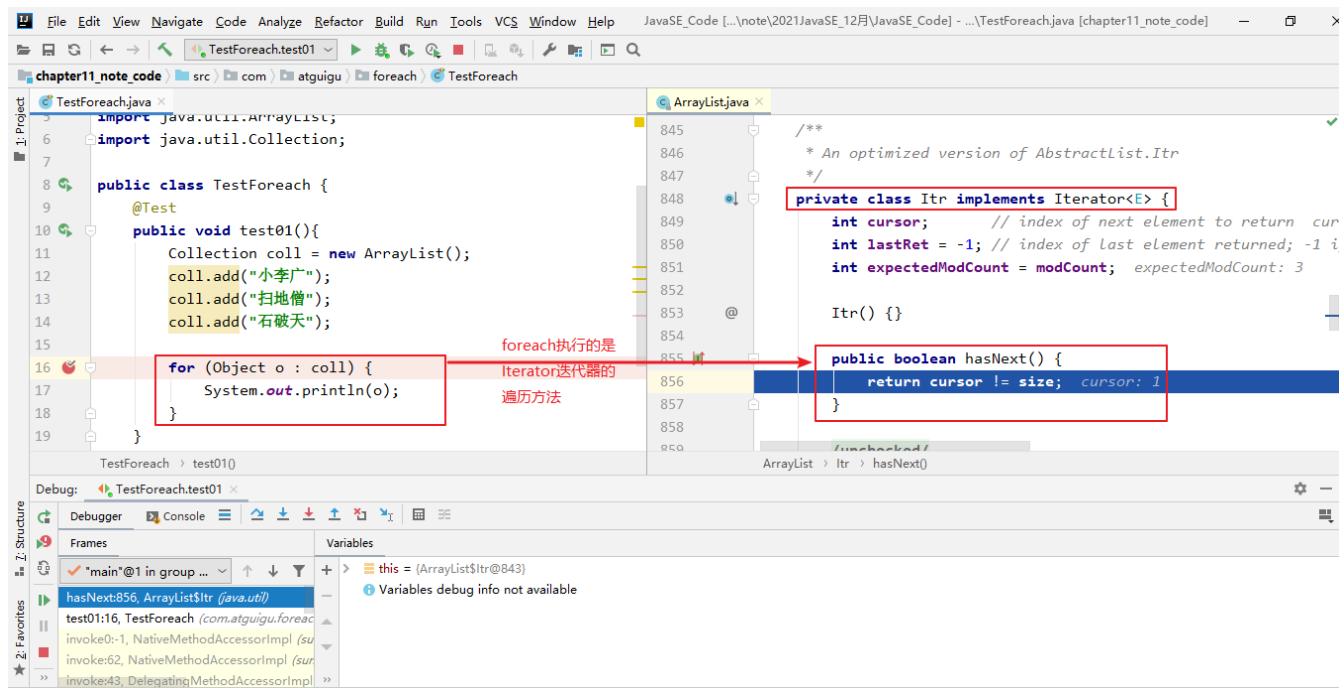
```
package com.atguigu.iterator;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;

public class TestForeach {
    @Test
    public void test01(){
        Collection coll = new ArrayList();
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");

        for (Object o : coll) {
            System.out.println(o);
        }
    }
}
```



11.2.4 使用Iterator迭代器删除元素

`java.util.Iterator`迭代器中有一个方法：

```
void remove();
```

那么，既然Collection已经有remove(xx)方法了，为什么Iterator迭代器还要提供删除方法呢？

因为在JDK1.8之前Collection接口没有removeIf方法，即无法根据条件删除。

例如：要删除以下集合元素中的偶数

```
package com.atguigu.iterator;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class TestIteratorRemove {
    @Test
    public void test01(){
        Collection coll = new ArrayList();
        coll.add(1);
        coll.add(2);
        coll.add(3);
        coll.add(4);

        // coll.remove(?)//没有removeIf方法无法实现删除“偶数”

        Iterator iterator = coll.iterator();
        while(iterator.hasNext()){
            Integer element = (Integer) iterator.next();
            if(element%2 == 0){
                iterator.remove();
            }
        }
        System.out.println(coll);
    }
}
```

11.2.5 Iterator迭代器的快速失败 (fail-fast) 机制

如果在Iterator、ListIterator迭代器创建后的任意时间从结构上修改了集合（通过迭代器自身的remove或add方法之外的任何其他方式），则迭代器将抛出ConcurrentModificationException。因此，面对并发的修改，迭代器很快就完全失败，而不是冒着在将来不确定的时间任意发生不确定行为的风险。

这样设计是因为，迭代器代表集合中某个元素的位置，内部会存储某些能够代表该位置的信息。当集合发生改变时，该信息的含义可能会发生变化，这时操作迭代器就可能会造成不可预料的事情。因此，果断抛异常阻止，是最好的方法。这就是Iterator迭代器的快速失败(fail-fast)机制。

1、ConcurrentModificationException异常

```
package com.atguigu.iterator;

import java.util.ArrayList;
import java.util.Collection;
```

```

import java.util.Iterator;

public class TestConcurrentModificationException {
    public static void main(String[] args) {
        Collection coll = new ArrayList();
        coll.add("hello");
        coll.add("world");
        coll.add("java");
        coll.add("haha");
        coll.add("mysql");

        Iterator iterator = coll.iterator();
        while(iterator.hasNext()){
            String str = (String)iterator.next();
            if(str.contains("a")){
                coll.remove(str); //foreach遍历集合过程中，调用集合的remove方法
            }
        }

        /*for (Object o : coll) {
            String str = (String) o;
            if(str.contains("a")){
                coll.remove(o); //foreach遍历集合过程中，调用集合的remove方法
            }
        }*/
    }
}

```

2、modCount变量

那么迭代器如何实现快速失败 (fail-fast) 机制的呢？

- 在ArrayList等集合类中都有一个modCount变量。它用来记录集合的结构被修改的次数。
- 当我们给集合添加和删除操作时，会导致modCount++。
- 然后当我们用Iterator迭代器遍历集合时，创建集合迭代器的对象时，用一个变量记录当前集合的modCount。例如：`int expectedModCount = modCount;`，并且在迭代器每次next()迭代元素时，都要检查`expectedModCount != modCount`，如果不相等了，那么说明你调用了Iterator迭代器以外的Collection的add,remove等方法，修改了集合的结构，使得modCount++，值变了，就会抛出ConcurrentModificationException。

下面以AbstractList和ArrayList.Itr迭代器为例进行源码分析：

AbstractList类中声明了modCount变量：

```

/**
 * The number of times this List has been <i>structurally modified</i>.
 * Structural modifications are those that change the size of the
 * list, or otherwise perturb it in such a fashion that iterations in
 * progress may yield incorrect results.
 *
 * <p>This field is used by the iterator and list iterator implementation
 * returned by the {@code iterator} and {@code listIterator} methods.
 * If the value of this field changes unexpectedly, the iterator (or list

```

```

* iterator) will throw a {@code ConcurrentModificationException} in
* response to the {@code next}, {@code remove}, {@code previous},
* {@code set} or {@code add} operations. This provides
* <i>fail-fast</i> behavior, rather than non-deterministic behavior in
* the face of concurrent modification during iteration.
*
* <p><b>Use of this field by subclasses is optional.</b> If a subclass
* wishes to provide fail-fast iterators (and list iterators), then it
* merely has to increment this field in its {@code add(int, E)} and
* {@code remove(int)} methods (and any other methods that it overrides
* that result in structural modifications to the list). A single call to
* {@code add(int, E)} or {@code remove(int)} must add no more than
* one to this field, or the iterators (and list iterators) will throw
* bogus {@code ConcurrentModificationExceptions}. If an implementation
* does not wish to provide fail-fast iterators, this field may be
* ignored.
*/
protected transient int modCount = 0;

```

翻译解释：modCount是这个list被结构性修改的次数。子类使用这个字段是可选的，如果子类希望提供fail-fast迭代器，它仅仅需要在add(int, E),remove(int)方法（或者它重写的其他任何会结构性修改这个列表的方法）中添加这个字段。调用一次add(int,E)或者remove(int)方法时必须且仅仅给这个字段加1，否则迭代器会抛出伪装的ConcurrentModificationExceptions错误。如果一个实现类不希望提供fail-fast迭代器，则可以忽略这个字段。

ArrayList的Itr迭代器：

```

private class Itr implements Iterator<E> {
    int cursor;
    int lastRet = -1;
    int expectedModCount = modCount; //在创建迭代器时，expectedModCount初始化为当前集合的
                                     modCount的值

    public boolean hasNext() {
        return cursor != size;
    }

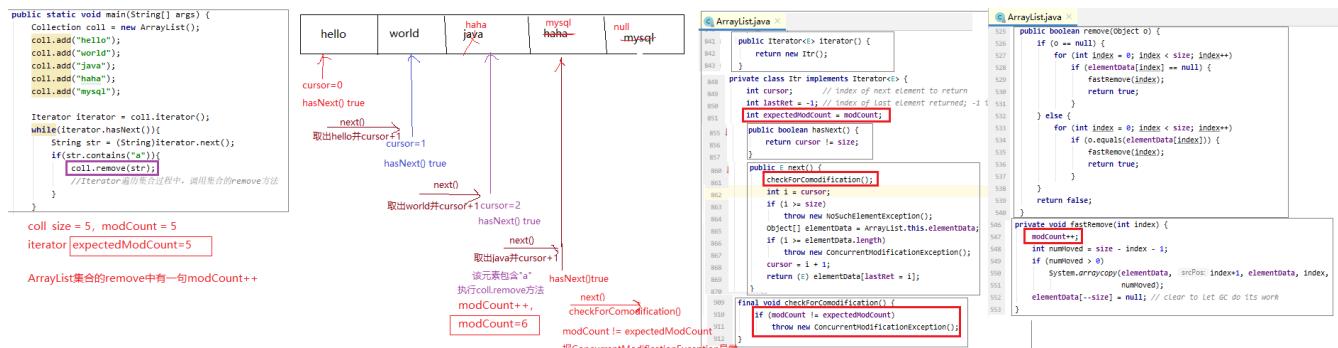
    @SuppressWarnings("unchecked")
    public E next() {
        checkForComodification(); //校验expectedModCount与modCount是否相等
        int i = cursor;
        if (i >= size)
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;
        return (E) elementData[lastRet = i];
    }

    final void checkForComodification() {
        if (modCount != expectedModCount) //校验expectedModCount与modCount是否相等
            throw new ConcurrentModificationException(); //不相等，抛异常
    }
}

```

ArrayList的remove方法：

```
public boolean remove(Object o) {  
    if (o == null) {  
        for (int index = 0; index < size; index++)  
            if (elementData[index] == null) {  
                fastRemove(index);  
                return true;  
            }  
    } else {  
        for (int index = 0; index < size; index++)  
            if (o.equals(elementData[index])) {  
                fastRemove(index);  
                return true;  
            }  
    }  
    return false;  
}  
  
private void fastRemove(int index) {  
    modCount++;  
    int numMoved = size - index - 1;  
    if (numMoved > 0)  
        System.arraycopy(elementData, index+1, elementData, index,  
                         numMoved);  
    elementData[--size] = null; // clear to let GC do its work  
}
```



注意，迭代器的快速失败行为不能得到保证，一般来说，存在不同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 `ConcurrentModificationException`。因此，编写依赖于此异常的程序的方式是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测bug。例如：

```
package com.atguigu.iterator;  
  
import java.util.ArrayList;  
import java.util.Collection;  
import java.util.Iterator;  
  
public class TestNoConcurrentModificationException {  
    public static void main(String[] args) {
```

```

collection coll = new ArrayList();
coll.add("hello");
coll.add("world");
coll.add("java");
coll.add("haha");

Iterator iterator = coll.iterator();
while (iterator.hasNext()) {
    String str = (String) iterator.next();
    if (str.contains("a")) {
        coll.remove(str);
        //Iterator遍历集合过程中，调用集合的remove方法
    }
}
}

```

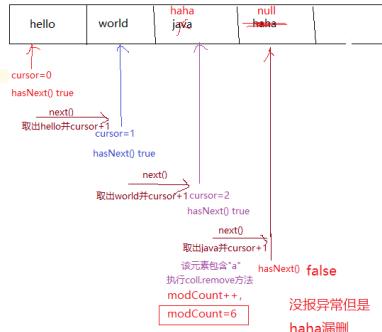
```

public static void main(String[] args) {
    Collection coll = new ArrayList();
    coll.add("hello");
    coll.add("world");
    coll.add("java");
    coll.add("haha");
}

Iterator iterator = coll.iterator();
while (iterator.hasNext()) {
    String str = (String) iterator.next();
    if (str.contains("a")) {
        coll.remove(str);
        //Iterator遍历集合过程中，调用集合的remove方法
    }
}

```

coll.size=4, modCount=4
iterator.expectedModCount=4
ArrayList集合的remove中有一句modCount++



```

ArrayList.java
public Iterator<E> iterator() {
    return new Itr();
}

private class Itr implements Iterator<E> {
    int cursor; // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if none
    int expectedModCount = modCount;

    public boolean hasNext() {
        return cursor < size;
    }

    public E next() {
        checkForComodification();
        if (cursor >= size)
            throw new NoSuchElementException();
        Object elementData = elementData[cursor];
        if (elementData == null)
            throw new ConcurrentModificationException();
        cursor++;
        return (E) elementData[lastRet = cursor];
    }

    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}

ArrayList.java
public boolean remove(Object o) {
    for (int index = 0; index < size; index++) {
        if (elementData[index] == null)
            fastRemove(index);
        return true;
    }
    for (int index = 0; index < size; index++) {
        if (o.equals(elementData[index])) {
            fastRemove(index);
            return true;
        }
    }
    return false;
}

private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index + 1, elementData, index,
                        numMoved);
    elementData[--size] = null; // clear to let GC do its work
}

```

第12章 泛型

12.1 泛型的概念

12.1.1 泛型的引入

例如：生产瓶子的厂家，一开始并不知道我们将来会用瓶子装什么，我们什么都可以装，但是有的时候，我们在使用时，想要限定某个瓶子只能用来装什么，这样我们不会装错，而用的时候也可以放心的使用，无需再三思量。我们生活中是在使用这个瓶子时在瓶子上“贴标签”，这样就轻松解决了问题。



还有，在Java中我们在声明方法时，当在完成方法功能时如果有未知的数据需要参与，这些未知的数据需要在调用方法时才能确定，那么我们把这样的数据通过形参表示。那么在方法体中，用这个形参名来代表那个未知的数据，而调用者在调用时，对应的传入值就可以了。

```

public static void main(String[] args) {
    int max = max(3,6); //实参 3,6
    System.out.println(max);
}

public static int max(int a, int b) { //形参 a,b
    return a > b ? a : b;
}

```

受以上两点启发，JDK1.5设计了泛型的概念。泛型即为“类型参数”，这个类型参数在声明它的类、接口或方法中，代表未知的通用的类型。例如：

java.lang.Comparable接口和java.util.Comparator接口，是用于对象比较大小的规范接口，这两个接口只是限定了当一个对象大于另一个对象时返回正整数，小于返回负整数，等于返回0。但是并不确定是什么类型的对象比较大，之前的时候只能用Object类型表示，使用时既麻烦又不安全，因此JDK1.5就给它们增加了泛型。

```

public interface Comparable<T>{
    int compareTo(T o) ;
}

```

```

public interface Comparator<T>{
    int compare(T o1, T o2) ;
}

```

其中就是类型参数，即泛型。

12.1.2 泛型的好处

示例代码：

JavaBean：圆类型

```
package com.atguigu.generic;

public class Circle{
    private double radius;

    public Circle(double radius) {
        super();
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public String toString() {
        return "Circle [radius=" + radius + "]";
    }
}
```

比较器

```
package com.atguigu.generic;

import java.util.Comparator;

public class CircleRadiusComparator implements Comparator{

    @Override
    public int compare(Object o1, Object o2) {
        //强制类型转换
        Circle c1 = (Circle) o1;
        Circle c2 = (Circle) o2;
        return Double.compare(c1.getRadius(), c2.getRadius());
    }
}
```

测试类

```
package com.atguigu.generic;

public class TestNoGeneric {
    public static void main(String[] args) {
        CircleRadiusComparator com = new CircleRadiusComparator();
        System.out.println(com.compare(new Circle(1), new Circle(2)));

        System.out.println(com.compare("圆1", "圆2")); //运行时异常: ClassCastException
    }
}
```

那么我们在使用如上面这样的接口时，如果没有泛型或不指定泛型，很麻烦，而且有安全隐患。

因为在设计（编译）Comparator接口时，不知道它会用于哪种类型的对象比较，因此只能将compare方法的形参设计为Object类型，而实际在compare方法中需要向下转型为Circle，才能调用Circle类的getRadius()获取半径值进行比较。

使用泛型：

比较器：

```
package com.atguigu.generic;

import java.util.Comparator;

public class CircleComparator implements Comparator<Circle> {

    @Override
    public int compare(Circle o1, Circle o2) {
        //不再需要强制类型转换，代码更简洁
        return Double.compare(o1.getRadius(), o2.getRadius());
    }
}
```

测试类

```
package com.atguigu.generic;

public class TestHasGeneric {
    public static void main(String[] args) {
        CircleComparator com = new CircleComparator();
        System.out.println(com.compare(new Circle(1), new Circle(2)));

        // System.out.println(com.compare("圆1", "圆2")); //编译错误，因为"圆1", "圆2"不是
        // Circle类型，是String类型，编译器提前报错，而不是冒着风险在运行时再报错
    }
}
```

如果有了泛型并使用泛型，那么既能保证安全，又能简化代码。

因为把不安全的因素在编译期间就排除了；既然通过了编译，那么类型一定是符合要求的，就避免了类型转换。

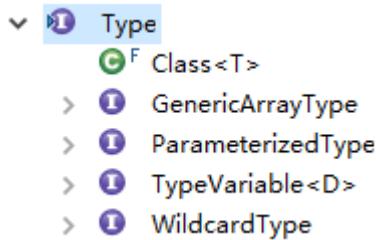
12.1.3 泛型的相关名词

<类型>这种语法形式就叫泛型。

其中：

- 是类型变量 (Type Variables) , 而是代表未知的数据类型, 我们可以指定为, , 等, 那么<类型>的形式我们成为类型参数;
 - 对比方法的参数的概念, 我们可以把, 称为类型形参, 将称为类型实参, 有助于我们理解泛型;
- Comparator这种就称为参数化类型 (Parameterized Types) 。

自从有了泛型之后, Java的数据类型就更丰富了:



Class: `class` 类的实例表示正在运行的 Java 应用程序中的类和接口。枚举是一种类, 注释是一种接口。每个数组属于被映射为 Class 对象的一个类, 所有具有相同元素类型和维数的数组都共享该 `class` 对象。基本的 Java 类型 (`boolean`、`byte`、`char`、`short`、`int`、`long`、`float` 和 `double`) 和关键字 `void` 也表示为 `class` 对象。

- GenericArrayType: 泛化的数组类型, 即`T[]`
- ParameterizedType: 参数化类型, 例如: Comparator, Comparator
- TypeVariable: 类型变量, 例如: Comparator中的T, Map<K,V>中的K,V
- WildcardType: 通配符类型, 例如: Comparator<?>等

12.1.4 在哪里可以声明类型变量<T>

- 声明类或接口时, 在类名或接口名后面声明类型变量, 我们把这样的类或接口称为泛型类或泛型接口

```
【修饰符】 class 类名<类型变量列表> 【extends 父类】 【implements 父接口们】 {
```

```
}
```

```
【修饰符】 interface 接口名<类型变量列表> 【implements 父接口们】 {
```

```
}
```

例如:

```
public class ArrayList<E>
public interface Map<K,V>{
    ...
}
```

- 声明方法时, 在【修饰符】与返回值类型之间声明类型变量, 我们把声明 (是**声明**不是单纯的使用) 了类型变量的方法称为泛型方法

```
【修饰符】 <类型变量列表> 返回值类型 方法名(【形参列表】)【throws 异常列表】 {  
    //...  
}
```

例如：java.util.Arrays类中的

```
public static <T> List<T> asList(T... a){  
    ...  
}
```

12.2 泛型类与泛型接口

12.2.1 使用核心类库中的泛型类/接口

自从JDK1.5引入泛型的概念之后，对之前核心类库中的API做了很大的修改，例如：集合框架集中的相关接口和类、java.lang.Comparable接口、java.util.Comparator接口、Class类等等。

下面以Collection、ArrayList集合以及Iterator迭代器为例演示，泛型类与泛型接口的使用。

案例一：Collection集合相关类型

- (1) 创建一个Collection集合（暂时创建ArrayList集合对象），并指定泛型为
- (2) 添加5个[0,100)以内的整数到集合中，
- (3) 使用foreach遍历输出5个整数，
- (4) 使用集合的removeIf方法删除偶数，为Predicate接口指定泛型
- (5) 再使用Iterator迭代器输出剩下的元素，为Iterator接口指定泛型。

```
package com.atguigu.genericclass.use;  
  
import java.util.ArrayList;  
import java.util.Collection;  
import java.util.Iterator;  
import java.util.Random;  
import java.util.function.Predicate;  
  
public class TestNumber {  
    public static void main(String[] args) {  
        Collection<Integer> coll = new ArrayList<Integer>();  
        Random random = new Random();  
        for (int i = 1; i <= 5 ; i++) {  
            coll.add(random.nextInt(100));  
        }  
  
        System.out.println("coll中5个随机数是: ");  
        for (Integer integer : coll) {  
            System.out.println(integer);  
        }  
    }  
}
```

```

        coll.removeIf(new Predicate<Integer>() {
            @Override
            public boolean test(Integer integer) {
                return integer % 2 == 0;
            }
        });

        System.out.println("coll中删除偶数后: ");
        Iterator<Integer> iterator = coll.iterator();
        while(iterator.hasNext()){
            Integer number = iterator.next();
            System.out.println(number);
        }

    }
}

```

案例二： Comparable接口

- (1) 声明矩形类Rectangle，包含属性长和宽，属性私有化，提供有参构造、get/set方法、重写toString方法，提供求面积和周长的方法。
- (2) 矩形类Rectangle实现java.lang.Comparable接口，并指定泛型为，重写int compareTo(T t)方法，按照矩形面积比较大小，面积相等的，按照周长比较大小。
- (3) 在测试类中，创建Rectangle数组，并创建5个矩形对象
- (4) 调用Arrays的sort方法，给矩形数组排序，并显示排序前后的结果。

```

package com.atguigu.genericclass.use;

public class Rectangle implements Comparable<Rectangle>{
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double getLength() {
        return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    @Override
    public int compareTo(Rectangle o) {
        if (this.length * this.width < o.length * o.width) {
            return -1;
        } else if (this.length * this.width > o.length * o.width) {
            return 1;
        } else {
            return Double.compare(this.length + this.width, o.length + o.width);
        }
    }

    @Override
    public String toString() {
        return "Rectangle{" +
                "length=" + length +
                ", width=" + width +
                '}';
    }
}

```

```
}

public double area(){
    return length * width;
}

public double perimeter(){
    return 2 * (length + width);
}

@Override
public String toString() {
    return "Rectangle{" +
        "length=" + length +
        ", width=" + width +
        ",area =" + area() +
        ",perimeter = " + perimeter() +
        '}';
}

@Override
public int compareTo(Rectangle o) {
    int compare = Double.compare(area(), o.area());
    return compare != 0 ? compare : Double.compare(perimeter(),o.perimeter());
}
}
```

```
package com.atguigu.genericclass.use;

import java.util.Arrays;

public class TestRectangle {
    public static void main(String[] args) {
        Rectangle[] arr = new Rectangle[4];
        arr[0] = new Rectangle(6,2);
        arr[1] = new Rectangle(4,3);
        arr[2] = new Rectangle(12,1);
        arr[3] = new Rectangle(5,4);

        System.out.println("排序之前: ");
        for (Rectangle rectangle : arr) {
            System.out.println(rectangle);
        }

        Arrays.sort(arr);

        System.out.println("排序之后: ");
        for (Rectangle rectangle : arr) {
            System.out.println(rectangle);
        }
    }
}
```

```
}
```

12.2.2 自定义泛型类与泛型接口

当我们在类或接口中定义某个成员时，该成员的相关类型是不确定的，而这个类型需要在使用这个类或接口时才可以确定，那么我们可以使用泛型。

- 当某个类/接口的非静态实例变量的类型不确定，需要在创建对象或子类继承时才能确定
- 当某个（些）类/接口的非静态方法的形参类型不确定，需要在创建对象或子类继承时才能确定

语法格式：

```
【修饰符】 class 类名<类型变量列表> 【extends 父类】 【implements 父接口们】 {  
}  
【修饰符】 interface 接口名<类型变量列表> 【extends 父接口们】 {  
}
```

注意：

- <类型变量列表>：可以是一个或多个类型变量，一般都是使用单个的大写字母表示。例如：、<K,V>等。
- <类型变量列表>中的类型变量不能用于静态成员上。

示例代码：

例如：我们要声明一个学生类，该学生包含姓名、成绩，而此时学生的成绩类型不确定，为什么呢，因为，语文老师希望成绩是“优秀”、“良好”、“及格”、“不及格”，数学老师希望成绩是89.5, 65.0，英语老师希望成绩是'A'-'B'-'C'-'D'-'E'。那么我们在设计这个学生类时，就可以使用泛型。

```
package com.atguigu.genericclass.define;  
  
public class Student<T>{  
    private String name;  
    private T score;  
  
    public Student() {  
        super();  
    }  
    public Student(String name, T score) {  
        super();  
        this.name = name;  
        this.score = score;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {
```

```

        this.name = name;
    }
    public T getScore() {
        return score;
    }
    public void setScore(T score) {
        this.score = score;
    }
    @Override
    public String toString() {
        return "姓名: " + name + ", 成绩: " + score;
    }
}

```

12.2.3 使用泛型类与泛型接口小结

在使用这种参数化的类与接口时，我们需要指定泛型变量的实际类型参数：

- (1) 实际类型参数必须是引用数据类型，不能是基本数据类型
- (2) 子类继承泛型父类时，子接口继承泛型父接口、或实现类实现泛型父接口时，
 - 指定类型变量对应的实际类型参数，此时子类或实现类不再是泛型类

```

package com.atguigu.genericclass.define;

//ChineseStudent不再是泛型类
public class ChineseStudent extends Student<String>{

    public ChineseStudent() {
        super();
    }

    public ChineseStudent(String name, String score) {
        super(name, score);
    }

}

```

```
public class Rectangle implements Comparable<Rectangle>
```

- 指定类型变量（该类型变量可以和原来字母一样，也可以换一个字母），此时子类、子接口、实现类仍然是泛型类或泛型接口

```
public interface Iterable<T>
```

```
public interface Collection<E> extends Iterable<E> //E:Element元素
```

```
public interface List<E> extends Collection<E>
```

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable
```

(3) 在创建泛型类的对象时指定类型变量对应的实际类型参数

```
package com.atguigu.genericclass.define;

public class TestStudent {
    public static void main(String[] args) {
        //语文老师使用时:
        Student<String> stu1 = new Student<String>("张三", "良好");
        ChineseStudent chineseStudent = new ChineseStudent("张三", "良好");

        //数学老师使用时:
        //Student<double> stu2 = new Student<double>("张三", 90.5); //错误, 必须是引用数据类型
        Student<Double> stu2 = new Student<Double>("张三", 90.5);

        //英语老师使用时:
        Student<Character> stu3 = new Student<Character>("张三", 'C');

        //错误的指定
        //Student<Object> stu = new Student<String>(); //错误的
    }
}
```

JDK1.7支持简写形式: Student stu1 = new Student<>("张三", "良好");

指定泛型实参时, 必须左右两边一致, 不存在多态现象

12.3 泛型方法

12.3.1 泛型方法的调用

在java.util.Arrays数组工具类中, 有很多泛型方法, 例如:

- public static List<T...> asList(T... a): 将实参对象依次添加到一个固定大小的List列表集合中。
- public static T[] copyOf(T[] original, int newLength): 复制任意对象数组, 新数组长度为newLength。
- ...

```
package com.atguigu.genericmethod;

import java.util.Arrays;
import java.util.List;

public class TestArrays {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("java", "world", "hello", "atguigu");
    }
}
```

```

        System.out.println(list);

        String[] arr = {"java", "world", "hello"};
        String[] strings = Arrays.copyOf(arr, arr.length * 2);
        System.out.println(Arrays.toString(strings));
    }
}

```

泛型方法在调用时，由实参的类型确定泛型方法类型变量的具体类型。

12.3.2 自定义泛型方法

前面介绍了在定义类、接口时可以声明<类型变量>，在该类的方法和属性定义、接口的方法定义中，这些<类型变量>可被当成普通类型来用。但是，在另外一些情况下，

- (1) 如果我们定义类、接口时没有使用<类型变量>，但是某个方法形参类型不确定时，这个方法可以单独定义<类型变量>；
- (2) 另外我们之前说类和接口上的类型形参是不能用于静态方法中，那么当某个静态方法的形参类型不确定时，静态方法可以单独定义<类型变量>。

语法格式：

```

【修饰符】 <类型变量列表> 返回值类型 方法名(【形参列表】)【throws 异常列表】 {
    //...
}

```

- <类型变量列表>：可以是一个或多个类型变量，一般都是使用单个的大写字母表示。例如：、<K,V>等。

示例代码：

我们编写一个数组工具类，包含可以给任意对象数组进行从小到大排序，调用元素对象的compareTo方法比较元素的大小关系。

```

package com.atguigu.genericmethod;

public class MyArrays {
    public static <T> void sort(T[] arr){
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length-i; j++) {
                if(((Comparable<T>)arr[j]).compareTo(arr[j+1])>0){
                    T temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}

```

```

package com.atguigu.genericmethod;

```

```
import com.atguigu.generic.Circle;

import java.util.Arrays;

public class MyArraysTest {
    public static void main(String[] args) {
        int[] arr = {3,2,5,1,4};
//        MyArrays.sort(arr); //错误的，因为int[]不是对象数组

        String[] strings = {"hello","java","chai"};
        MyArrays.sort(strings);
        System.out.println(Arrays.toString(strings));

        Circle[] circles = {new Circle(2.0),new Circle(1.2),new Circle(3.0)};
        MyArrays.sort(circles); //编译通过，运行报错，Circle没有实现Comparable接口
    }
}
```

12.4 类型变量的上限与泛型的擦除

12.4.1 类型变量的上限

当在声明类型变量时，如果不希望这个类型变量代表任意引用数据类型，而是某个系列的引用数据类型，那么可以设定类型变量的上限。

语法格式：

```
<类型变量 extends 上限>
```

如果有多个上限

```
<类型变量 extends 上限1 & 上限2>
```

如果多个上限中有类有接口，那么只能有一个类，而且必须写在最左边。接口的话，可以多个。

如果在声明<类型变量>时没有指定任何上限，默认上限是java.lang.Object。

1、定义泛型类的类型变量时指定上限

例如：我们要声明一个两个数算术运算的工具类，要求两个数必须是Number数字类型，并且实现Comparable接口。

```
package com.atguigu.limmit;

import java.math.BigDecimal;
import java.math.BigInteger;

public class NumberTools<T extends Number & Comparable<T>>{
    private T a;
    private T b;
```

```

public NumberTools(T a, T b) {
    super();
    this.a = a;
    this.b = b;
}

public T getSum(){
    if(a instanceof BigInteger){
        return (T) ((BigInteger) a).add((BigInteger)b);
    }else if(a instanceof BigDecimal){
        return (T) ((BigDecimal) a).add((BigDecimal)b);
    }else if(a instanceof Byte){
        return (T)(Byte.valueOf((byte)((Byte)a+(Byte)b)));
    }else if(a instanceof Short){
        return (T)(short.valueOf((short)((Short)a+(Short)b)));
    }else if(a instanceof Integer){
        return (T)(Integer.valueOf((Integer)a+(Integer)b));
    }else if(a instanceof Long){
        return (T)(Long.valueOf((Long)a+(Long)b));
    }else if(a instanceof Float){
        return (T)(Float.valueOf((Float)a+(Float)b));
    }else if(a instanceof Double){
        return (T)(Double.valueOf((Double)a+(Double)b));
    }
    throw new UnsupportedOperationException("不支持该操作");
}

public T getSubtract(){
    if(a instanceof BigInteger){
        return (T) ((BigInteger) a).subtract((BigInteger)b);
    }else if(a instanceof BigDecimal){
        return (T) ((BigDecimal) a).subtract((BigDecimal)b);
    }else if(a instanceof Byte){
        return (T)(Byte.valueOf((byte)((Byte)a-(Byte)b)));
    }else if(a instanceof Short){
        return (T)(short.valueOf((short)((Short)a-(Short)b)));
    }else if(a instanceof Integer){
        return (T)(Integer.valueOf((Integer)a-(Integer)b));
    }else if(a instanceof Long){
        return (T)(Long.valueOf((Long)a-(Long)b));
    }else if(a instanceof Float){
        return (T)(Float.valueOf((Float)a-(Float)b));
    }else if(a instanceof Double){
        return (T)(Double.valueOf((Double)a-(Double)b));
    }
    throw new UnsupportedOperationException("不支持该操作");
}
}

```

测试类

```

package com.atguigu.limmit;

public class NumberToolsTest {
    public static void main(String[] args) {
        NumberTools<Integer> tools = new NumberTools<Integer>(8,5);
        Integer sum = tools.getSum();
        System.out.println("sum = " + sum);
        Integer subtract = tools.getSubtract();
        System.out.println("subtract = " + subtract);
    }
}

```

2、定义泛型方法的类型变量时指定上限

我们编写一个数组工具类，包含可以给任意对象数组进行从小到大排序，调用元素对象的compareTo方法比较元素的大小关系。要求数组的元素类型必须是java.lang.Comparable接口类型。

```

package com.atguigu.limmit;

public class MyArrays {
    public static <T extends Comparable<T>> void sort(T[] arr){
        for (int i = 1; i < arr.length; i++) {
            for (int j = 0; j < arr.length-i; j++) {
                if(arr[j].compareTo(arr[j+1])>0){
                    T temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}

```

测试类

```

package com.atguigu.limmit;

import com.atguigu.generic.Circle;
import java.util.Arrays;

public class MyArraysTest {
    public static void main(String[] args) {
        int[] arr = {3,2,5,1,4};
//        MyArrays.sort(arr); //错误的，因为int[]不是对象数组

        String[] strings = {"hello","java","chai"};
        MyArrays.sort(strings);
        System.out.println(Arrays.toString(strings));

        Circle[] circles = {new Circle(2.0),new Circle(1.2),new Circle(3.0)};
//        MyArrays.sort(circles); //编译报错
    }
}

```

```
    }  
}
```

12.4.2 泛型擦除

当使用参数化类型的类或接口时，如果没有指定泛型，那么会怎么样呢？

会发生泛型擦除，自动按照最左边的第一个上限处理。如果没有指定上限，上限即为Object。

```
package com.atguigu.limmit;  
  
import java.util.ArrayList;  
import java.util.Collection;  
  
public class TestErase {  
    public static void main(String[] args) {  
        NumberTools tools = new NumberTools(8,5);  
        Number sum = tools.getSum(); //自动按照Number处理  
        System.out.println("sum = " + sum);  
        Number subtract = tools.getSubtract();  
        System.out.println("subtract = " + subtract);  
  
        Collection coll = new ArrayList();  
        coll.add("hello");  
        coll.add(1);  
        for (Object o : coll) { //自动按照Object处理  
            System.out.println(o);  
        }  
    }  
}
```

12.5 类型通配符

12.5.1 Java泛型指定限制问题

声明一个方法，形参是Collection，但是元素类型不确定，怎么办？

```
package com.atguigu.wildcard;  
  
import java.util.ArrayList;  
import java.util.Collection;  
  
public class TestProblem {  
    public static void m1(Collection<Object> coll){  
        for (Object o : coll) {  
            System.out.println(o);  
        }  
    }  
}
```

```

public static void m2(Collection coll){
    for (Object o : coll) {
        System.out.println(o);
    }
}

public static <T> void m3(Collection<T> coll){
    for (T o : coll) {
        System.out.println(o);
    }
}

public static void main(String[] args) {
    m1(new ArrayList<Object>());//Collection<Object> coll = new ArrayList<Object>();
    m1(new ArrayList<>());//Collection<Object> coll = new ArrayList<>();//与上面完全等价
    m1(new ArrayList());//Collection<Object> coll = new ArrayList();//有警告
//    m1(new ArrayList<String>());//Collection<Object> coll = new ArrayList<String>();//错误
}

//编译看左边，左边泛型擦除，此处泛型按照Object处理，右边泛型指定啥都没用
m2(new ArrayList<Object>());//Collection coll = new ArrayList<Object>();
m2(new ArrayList<>());//Collection coll = new ArrayList<>();//与上面完全等价
m2(new ArrayList());//Collection coll = new ArrayList();//泛型擦除
m2(new ArrayList<String>());//Collection coll = new ArrayList<String>();

m3(new ArrayList<Object>());//Collection<Object> coll = new ArrayList<Object>();
m3(new ArrayList<>());//Collection<> coll = new ArrayList<>();//与上面完全等价
m3(new ArrayList());//Collection<Object> coll = new ArrayList();//有警告
m3(new ArrayList<String>());//Collection<String> coll = new ArrayList<String>();
}
}

```

12.5.2 类型通配符

当我们声明一个变量/形参时，这个变量/形参的类型是一个泛型类或泛型接口，例如：Comparator类型，但是我们仍然无法确定这个泛型类或泛型接口的类型变量的具体类型，此时我们考虑使用类型通配符？。

```

package com.atguigu.wildcard;

import java.util.ArrayList;
import java.util.Collection;

public class TestWildcard {
    public static void m4(Collection<?> coll){
        for (Object o : coll) {
            System.out.println(o);
        }
    }

    public static void main(String[] args) {
        //右边泛型指定为任意类型或不指定都可以
        m4(new ArrayList<Object>());//Collection<?> coll = new ArrayList<Object>();
        m4(new ArrayList<>());//Collection<?> coll = new ArrayList<>();
    }
}

```

```
m4(new ArrayList());//Collection<?> coll = new ArrayList();
m4(new ArrayList<String>());//Collection<?> coll = new ArrayList<String>();
}
}
```

12.5.3 类型通配符的三种使用形式

类型通配符 ? 有三种使用形式：

- <?>：完整形式为：类名<?> 或接口名<?>，此时?代表任意类型。
- <? extends 上限>：完整形式为：类名<? extends 上限类型> 或接口名<? extends 上限类型>，此时?代表上限类型本身或者上限的子类，即?代表<= 上限的类型。
- <? super 下限>：完整形式为：类名<? super 下限类型> 或接口名<? super 下限类型>，此时?代表下限类型本身或者下限的父类，即?代表>= 下限的类型。

案例：

声明一个集合工具类MyCollections，要求包含：

- public static boolean different(Collection<?> c1, Collection<?> c2)：比较两个Collection集合，此时两个Collection集合的泛型可以是任意类型，如果两个集合中没有相同的元素，则返回true，否则返回false。
- public static void addAll(Collection<? super T> c1, T... args)：可以将任意类型的多个对象添加到一个Collection集合中，此时要求Collection集合的泛型指定必须>=元素类型。
- public static void copy(Collection<? super T> dest, Collection<? extends T> src)：可以将一个Collection集合的元素复制到另一个Collection集合中，此时要求原Collection泛型的类型<=目标Collection的泛型类型。

```
package com.atguigu.wildcard;

import java.util.Collection;

public class MyCollections {
    public static boolean different(Collection<?> c1, Collection<?> c2){
        return c1.containsAll(c2) && c2.containsAll(c1);
    }

    public static <T> void addAll(Collection<? super T> c1, T... args){
        for (int i = 0; i < args.length; i++) {
            c1.add(args[i]);
        }
    }

    public static <T> void copy(Collection<? super T> dest, Collection<? extends T> src){
        for (T t : src) {
            dest.add(t);
        }
    }
}
```

测试类

```

package com.atguigu.wildcard;

import java.util.ArrayList;
import java.util.Collection;

public class MyCollectionsTest {
    public static void main(String[] args) {
        Collection<Integer> c1 = new ArrayList<Integer>();
        MyCollections.addAll(c1, 1, 2, 3, 4, 5);
        System.out.println("c1 = " + c1);

        Collection<String> c2 = new ArrayList<String>();
        MyCollections.addAll(c2, "hello", "java", "world");
        System.out.println("c2 = " + c2);

        System.out.println("c1 != c2 " + MyCollections.different(c1, c2));

        Collection<Object> c3 = new ArrayList<>();
        MyCollections.copy(c3, c1);
        MyCollections.copy(c3, c2);
        System.out.println("c3 = " + c3);
    }
}

```

12.5.4 使用类型通配符来指定类型参数的问题

(1) 如果把“泛型类”指定为“泛型类<?>”：那么该泛型类中所有参数是T类型的方法或成员都无法正常使用。参数类型不是T类型的方法照常使用。

```

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;

public class TestProblem {
    @Test
    public void test01(){
        Collection<?> coll = new ArrayList<>();
//        coll.add("hello");
//        coll.add(1);
//        coll.add(1.0);
        /*
         * 上面所有添加操作都报错。
         * 为什么？
         * 因为<?>表示未知的类型，集合的元素是不确定的，那么添加任意类型对象都有风险。
         */

        void add(E t)方法无法正常使用
        因为此时E由?表示，即表示直到add方法被调用时，E的类型仍然不确定，所以该方法无法正常使用
    }
}

```

```

        collection<?> coll2 = Arrays.asList("hello", "java", "world");
        for (Object o : coll2) {
            System.out.println(o);
        }
    }
}

```

(2) 如果把“泛型类”指定为“泛型类<? extends 上限>”：那么该泛型类中所有参数是T类型的方法或成员都无法正常使用。参数类型不是T类型的方法照常使用。

```

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;

public class TestProblem {
    @Test
    public void test02() {
        Collection<? extends Number> coll = new ArrayList<Double>();
        // coll.add(1);
        // coll.add(1.0);
        // coll.add("hello");
        /*
         * 上面所有添加操作都报错。
         * 为什么？
         * 因为<?>表示未知的类型，代表<=Number的任意一种
         *
         * void add(E t)方法无法正常使用
         * 因为此时<E>由<? extends Number>表示，即表示直到add方法被调用时，E的类型仍然不确定，所以该方法无法
         * 正常使用。它可以是<=Number的任意一种类型。
         */
    }
}

```

(3) 如果把“泛型类”指定为“泛型类<? super 下限>”：那么该泛型类中所有参数是T类型的方法或成员都可以使用，但是有要求。参数类型不是T类型的方法照常使用。

```

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;

public class TestProblem {
    @Test
    public void test03() {
        Collection<? super Number> coll = new ArrayList<>();
        coll.add(1);
        coll.add(1.0);
        // coll.add("hello");
        /*
         *
         */
    }
}

```

```

前两个可以，最后一个不行
<? super Number>代表<=Number类型。最小可能是Number。
//可以添加Number对象或Number子类对象
*/
}
}

```

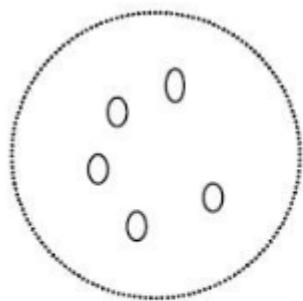
第13章 集合与数据结构

13.1 数据结构

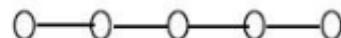
数据结构就是研究数据的逻辑结构和物理结构以及它们之间相互关系，并对这种结构定义相应的运算，而且确保经过这些运算后所得到的新结构仍然是原来的结构类型。

(1) 数据的逻辑结构指反映数据元素之间的逻辑关系，而与他们在计算机中的存储位置无关：

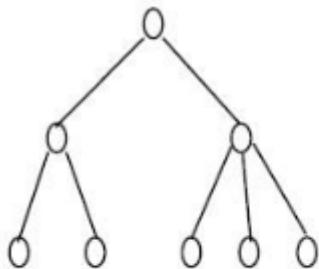
- 散列结构：数据结构中的元素之间除了“同属一个集合”的相互关系外，别无其他关系；
- 线性结构：数据结构中的元素存在一对一的相互关系；
- 树形结构：数据结构中的元素存在一对多的相互关系；
- 图形结构：数据结构中的元素存在多对多的相互关系。



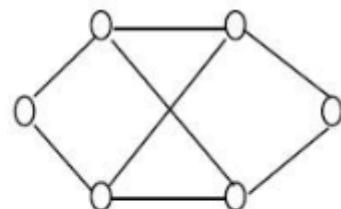
(a) 散列结构



(b) 线性结构



(c) 树型结构



(d) 图形结构

图 1.4 四类基本结构的示意图

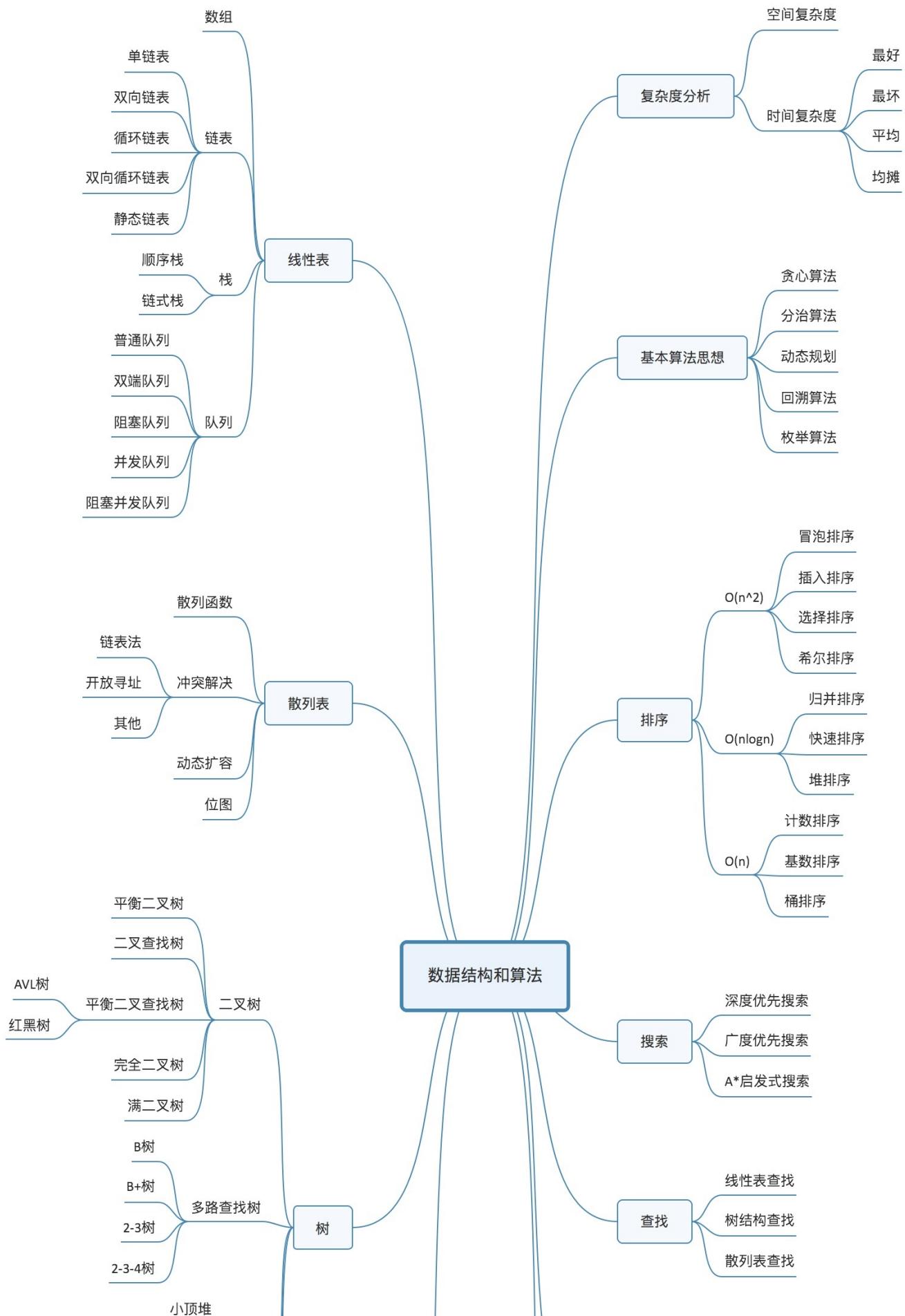
(2) 数据的物理结构/存储结构：是描述数据具体在内存中的存储（如：数组结构、链式结构、索引结构、哈希结构）等，一种数据逻辑结构可表示成一种或多种物理存储结构。

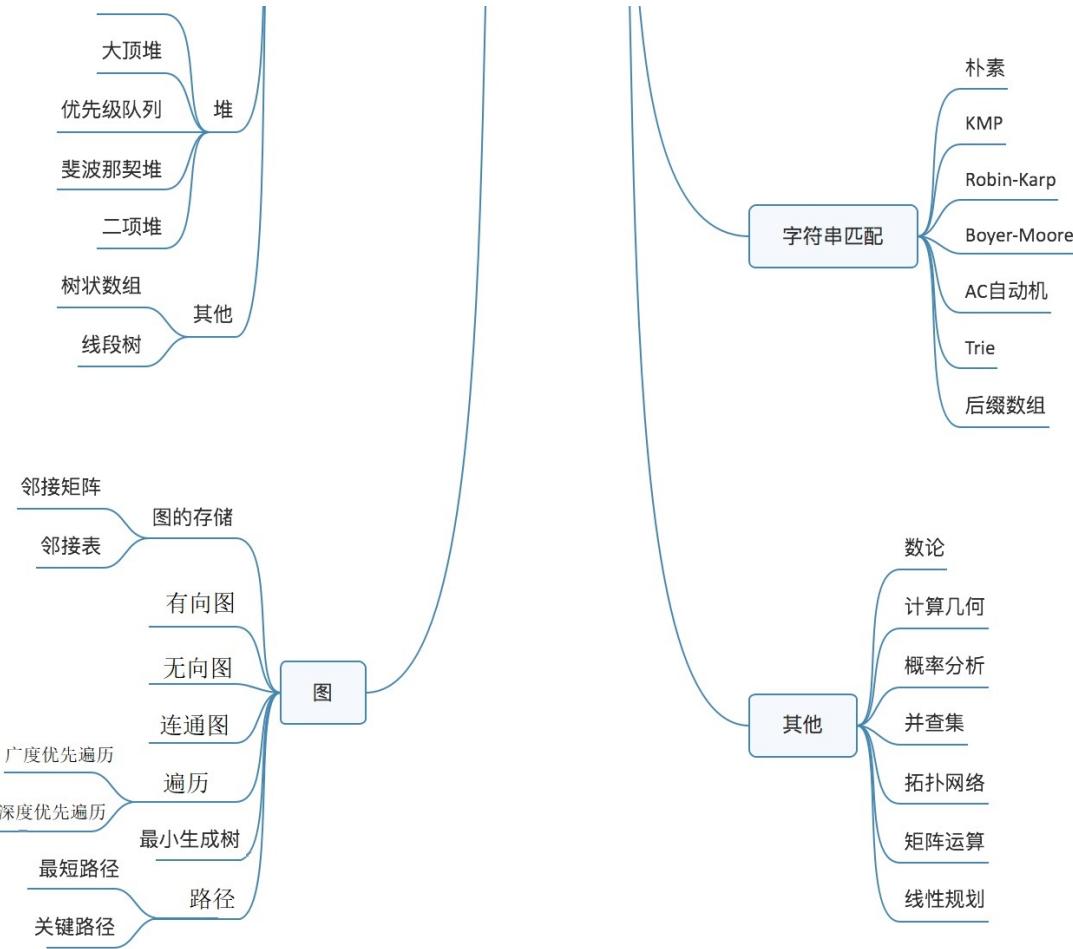
- 数组结构：元素在内存中是==连续存储==的，即元素存储在一整块连续的存储空间中，此时根据索引的查询效率是非常高的，因为可以根据下标索引直接一步到位找到元素位置，如果在数组末尾添加和删除元素效率也非常低。

常高。缺点是，如果事先申请足够大的内存空间，可能造成空间浪费，如果事先申请较小的内存空间，可能造成频繁扩容导致元素频繁搬家。另外，在数组中间添加、删除元素操作，就需要移动元素，此时效率也要打折。

- 链式结构：元素在内存中是不要求连续存储的，但是==元素是封装在结点==当中的，结点中需要存储元素数据，以及相关结点对象的引用地址。结点与结点之间可以是一对一的关系，也可以一对多的关系，比如：链表、树等。遍历链式结构只能从头遍历，对于较长的链表来说查询效率不高，对于树结构来说，查询效率比链表要高一点，因为每次可以确定一个分支，从而排除其他分支，但是相对于数组来说，还是数组[下标]的方式更快。树的实现方式有很多种，无非就是在添加/删除效率与查询效率之间权衡。
- 索引结构：元素在内存中是不要求连续存储的，但是需要有==单独的一个索引表==来记录每一个元素的地址，这种结构根据索引的查询效率很高，但是需要额外存储和维护索引表。
- 哈希结构：元素的存储位置需要通过其==hashCode值==来计算，查询效率也很多，但是要考虑和解决好哈希冲突问题。

数据结构和算法是一门完整并且复杂的课程。





Java的核心类库中提供很多数据结构对应的集合类型，例如动态数组、双向链表、顺序栈、链式栈、队列、双端队列、红黑树、哈希表等等。

13.2 List集合

Collection 层次结构中的根接口。一些 collection 允许有重复的元素，而另一些则不允许。一些 collection 是有序的，而另一些则是无序的。JDK 不提供此接口的任何直接实现：它提供更具体的子接口（如 Set 和 List、Queue）实现。我们掌握了Collection接口的使用后，再来看看Collection接口中的子接口，他们都具备那些特性呢？

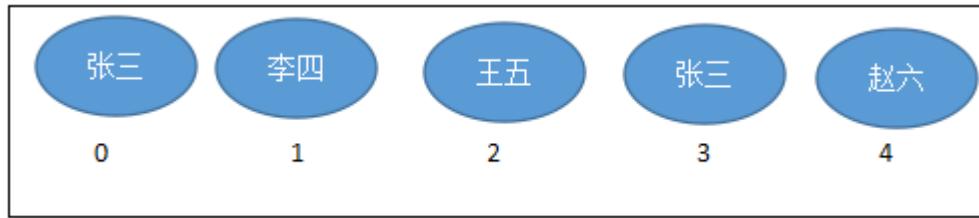
13.2.1 List接口介绍

`java.util.List` 接口继承自 `collection` 接口，是单列集合的一个重要分支，习惯性地会将实现了 `List` 接口的对象称为List集合。

List接口特点：

- List集合所有的元素是以一种==线性方式==进行存储的，例如，存元素的顺序是11、22、33。那么集合中，元素的存储就是按照11、22、33的顺序完成的)
- 它是一个元素==存取有序==的集合。即元素的存入顺序和取出顺序有保证。
- 它是一个==带有索引==的集合，通过索引就可以精确的操作集合中的元素（与数组的索引是一个道理）。
- 集合中可以有==重复==的元素，通过元素的`equals`方法，来比较是否为重复的元素。

List集合类中元素有序、且可重复。这就像银行门口客服，给每一个来办理业务的客户分配序号：第一个来的是“张三”，客服给他分配的是0；第二个来的是“李四”，客服给他分配的1；以此类推，最后一个序号应该是“总人数-1”。



注意：

List集合关心元素是否有序，而不关心是否重复，请大家记住这个原则。例如“张三”可以领取两个号。

13.2.2 List接口中常用方法

List作为Collection集合的子接口，不但继承了Collection接口中的全部方法，而且还增加了一些根据元素索引来操作集合的特有方法，如下：

List除了从Collection集合继承的方法外，List集合里添加了一些根据索引来操作集合元素的方法。

1、添加元素

- void add(int index, E ele)
- boolean addAll(int index, Collection<? extends E> eles)

2、获取元素

- E get(int index)
- List subList(int fromIndex, int toIndex)

3、获取元素索引

- int indexOf(Object obj)
- int lastIndexOf(Object obj)

4、删除和替换元素

- E remove(int index)
- E set(int index, E ele)

List集合特有的方法都是跟索引相关：

```
package com.atguigu.list;

import java.util.ArrayList;
import java.util.List;

public class TestListMethod {
    public static void main(String[] args) {
        // 创建List集合对象
        List<String> list = new ArrayList<String>();

        // 往尾部添加 指定元素
        list.add("图图");
        list.add("小美");
        list.add("不高兴");

        System.out.println(list);
    }
}
```

```

// add(int index, String s) 往指定位置添加
list.add(1, "没头脑");

System.out.println(list);
// String remove(int index) 删除指定位置元素 返回被删除元素
// 删除索引位置为2的元素
System.out.println("删除索引位置为2的元素");
System.out.println(list.remove(2));

System.out.println(list);

// String set(int index, String s)
// 在指定位置 进行 元素替代 (改)
// 修改指定位置元素
list.set(0, "三毛");
System.out.println(list);

// String get(int index) 获取指定位置元素
// 跟size() 方法一起用 来 遍历的
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
// 还可以使用增强for
for (String string : list) {
    System.out.println(string);
}
}

}

```

在JavaSE中List名称的类型有两个，一个是java.util.List集合接口，一个是java.awt.List图形界面的组件，别导错包了。

13.2.3 ListIterator迭代器

List 集合额外提供了一个 listIterator() 方法，该方法返回一个 ListIterator 列表迭代器对象， ListIterator 接口继承了 Iterator 接口，提供了专门操作 List 的方法：

- void add(): 通过迭代器添加元素到对应集合
- void set(Object obj): 通过迭代器替换正迭代的元素
- void remove(): 通过迭代器删除刚迭代的元素
- boolean hasPrevious(): 如果以逆向遍历列表，往前是否还有元素。
- Object previous(): 返回列表中的前一个元素。
- int previousIndex(): 返回列表中的前一个元素的索引
- boolean hasNext()
- Object next()
- int nextIndex()

```

package com.atguigu.list;

import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

```

```

public class TestListIterator {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张三");
        list.add("李四");
        list.add("王五");
        list.add("赵六");
        list.add("钱七");

        //从指定位置往前遍历
        System.out.println("从后往前遍历: ");
        ListIterator<String> listIterator = list.listIterator(list.size());
        while(listIterator.hasPrevious()){
            int previousIndex = listIterator.previousIndex();
            String previous = listIterator.previous();
            System.out.println(previousIndex + ":" + previous);
        }

        //从前往后遍历
        System.out.println("从前往后遍历: ");
        while(listIterator.hasNext()){
            int nextIndex = listIterator.nextInt();
            String next = listIterator.next();
            System.out.println(nextIndex + ":" + next);
        }

        //在“王五”前面添加“光头强”
        System.out.println("在“王五”前面添加\"光头强\"");
        while(listIterator.hasPrevious()){
            String previous = listIterator.previous();
            if("王五".equals(previous)){
                listIterator.add("光头强");
            }
        }

        //把“李四”替换为“李强”
        System.out.println("把\"李四\"替换为\"李强\"");
        while(listIterator.hasNext()){
            String next = listIterator.next();
            if("李四".equals(next)){
                listIterator.set("李强");
            }
        }

        System.out.println("list = " + list);
    }
}

```

13.2.4 List接口的实现类们

List接口的实现类有很多，常见的有：

ArrayList：动态数组

Vector：动态数组

LinkedList：双向链表

Stack：栈

当然，还有很多List接口的实现类这里没有列出来，基础阶段先了解这几个。

13.2.5 动态数组

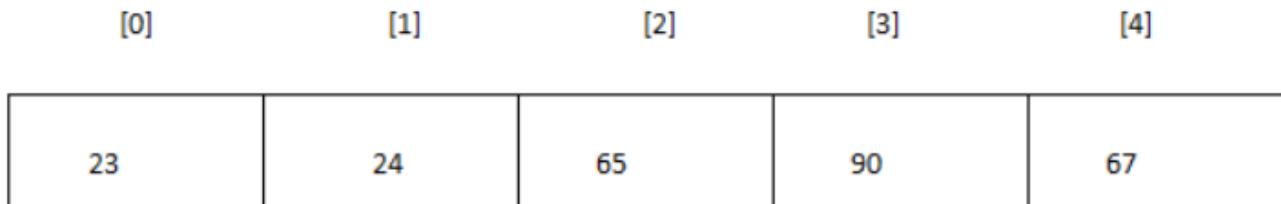
1、动态数组的特点

逻辑结构特点：线性结构

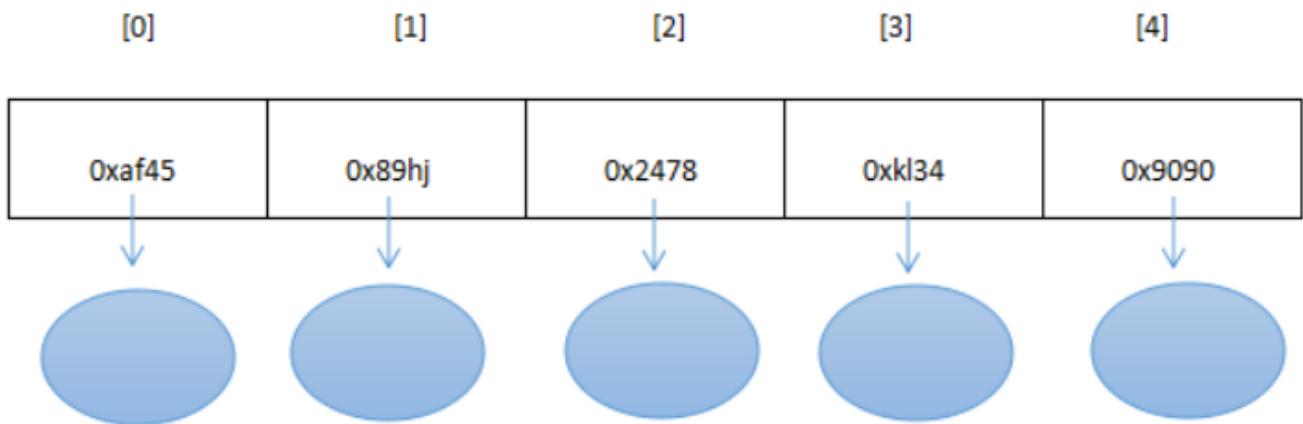
物理结构特点：

- 申请内存：一次申请一大段连续的空间，一旦申请到了，内存就固定了。
- 存储特点：所有数据存储在这个连续的空间中，数组中的每一个元素都是一个具体的数据（或对象），所有数据都紧密排布，不能有间隔。

例如：整型数组



例如：对象数组



2、自定义动态数组

```
package com.atguigu.list;  
  
import java.util.Arrays;
```

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class MyArrayList<E> implements Iterable<E>{
    private Object[] all;
    private int total;

    public MyArrayList(){
        all = new Object[10];
    }

    public void add(E e) {
        ensureCapacityEnough();
        all[total++] = e;
    }

    private void ensureCapacityEnough() {
        if(total >= all.length){
            all = Arrays.copyOf(all, all.length + (all.length>>1));
        }
    }

    public void insert(int index, E value) {
        //是否需要扩容
        ensureCapacityEnough();
        //添加元素的下标检查
        addCheckIndex(index);
        if(total-index > 0) {
            System.arraycopy(all, index, all, index+1, total-index);
        }
        all[index]=value;
        total++;
    }

    private void addCheckIndex(int index) {
        if(index<0 || index>total){
            throw new IndexOutOfBoundsException(index+"越界");
        }
    }

    public void delete(E e) {
        int index = indexOf(e);
        if(index== -1){
            throw new NoSuchElementException(e+"不存在");
        }
        delete(index);
    }

    public void delete(int index) {
        //删除元素的下标检查
        checkIndex(index);
        if(total-index-1 > 0) {
            System.arraycopy(all, index+1, all, index, total-index-1);
        }
    }
}
```

```
        }
        all[--total] = null;
    }

private void checkIndex(int index) {
    if(index<0 || index>total){
        throw new IndexOutOfBoundsException(index+"越界");
    }
}

public void update(int index, E value) {
    //更新修改元素的下标检查
    checkIndex(index);
    all[index] = value;
}

public void update(E old, E value) {
    int index = indexOf(old);
    if(index!=-1){
        update(index, value);
    }
}

public boolean contains(E e) {
    return indexOf(e) != -1;
}

public int indexOf(E e) {
    int index = -1;
    if(e==null){
        for (int i = 0; i < total; i++) {
            if(e == all[i]){
                index = i;
                break;
            }
        }
    }else{
        for (int i = 0; i < total; i++) {
            if(e.equals(all[i])){
                index = i;
                break;
            }
        }
    }
    return index;
}

public E get(int index) {
    //获取元素的下标检查
    checkIndex(index);
    return (E) all[index];
}
```

```

public int size() {
    return total;
}

public Iterator<E> iterator() {
    return new Itr();
}

private class Itr implements Iterator<E>{
    private int cursor;

    @Override
    public boolean hasNext() {
        return cursor!=total;
    }

    @Override
    public E next() {
        return (E) all[cursor++];
    }

    @Override
    public void remove() {
        MyArrayList.this.delete(--cursor);
    }
}
}

```

测试类：

```

package com.atguigu.list;

import java.util.Iterator;

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

        MyArrayList<String> my = new MyArrayList<>();
        my.add("hello");
        my.add("java");
        my.add("java");
        my.add("world");
        my.add(null);
        my.add(null);
        my.add("atguigu");
        my.add("list");
        my.add("data");

        System.out.println("元素个数: " + my.size());
        for (String s : my) {
            System.out.println(s);
        }
    }
}

```

```
}

System.out.println("-----");
System.out.println("在[1]插入尚硅谷后: ");
my.insert(1, "尚硅谷");
System.out.println("元素个数: " + my.size());
for (String s : my) {
    System.out.println(s);
}

System.out.println("-----");
System.out.println("删除[1]位置的元素后: ");
my.delete(1);
System.out.println("元素个数: " + my.size());
for (String s : my) {
    System.out.println(s);
}

System.out.println("删除atguigu元素后: ");
my.delete("atguigu");
System.out.println("元素个数: " + my.size());
for (String s : my) {
    System.out.println(s);
}

System.out.println("删除null元素后: ");
my.delete(null);
System.out.println("元素个数: " + my.size());
for (String s : my) {
    System.out.println(s);
}

System.out.println("-----");
System.out.println("替换[3]位置的元素为尚硅谷后: ");
my.update(3, "尚硅谷");
System.out.println("元素个数: " + my.size());
for (String s : my) {
    System.out.println(s);
}

System.out.println("替换java为JAVA后: ");
my.update("java", "JAVA");
System.out.println("元素个数: " + my.size());
for (String s : my) {
    System.out.println(s);
}

System.out.println("-----");
System.out.println("是否包含java: " + my.contains("java"));
System.out.println("java的位置: " + my.indexOf("java"));
System.out.println("haha的位置: " + my.indexOf("haha"));
System.out.println("[0]位置元素是: " + my.get(0));

System.out.println("-----");
System.out.println("删除字符串长度>4的元素后: ");
Iterator<String> iterator = my.iterator();
while(iterator.hasNext()) {
```

```

        String next = iterator.next();
        if(next != null && next.length()>4) {
            iterator.remove();
        }
    }
    System.out.println("元素个数: " + my.size());
    for (String string : my) {
        System.out.println(string);
    }
}
}

```

3、Java核心类库中的动态数组

Java的List接口的实现类中有两个动态数组的实现：Vector和ArrayList。

(1) ArrayList与Vector的区别？

它们的底层物理结构都是数组，我们称为动态数组。

- ArrayList是新版的动态数组，线程不安全，效率高，Vector是旧版的动态数组，线程安全，效率低。
- 动态数组的扩容机制不同，ArrayList扩容为原来的1.5倍，Vector扩容增加为原来的2倍。
- 数组的初始化容量，如果在构建ArrayList与Vector的集合对象时，没有显式指定初始化容量，那么Vector的内部数组的初始容量默认为10，而ArrayList在JDK1.6及之前的版本也是10，JDK1.7之后的版本ArrayList初始化为长度为0的空数组，之后在添加第一个元素时，再创建长度为10的数组。
 - 用的时候，再创建数组，避免浪费。因为很多方法的返回值是ArrayList类型，需要返回一个ArrayList的对象，例如：后期从数据库查询对象的方法，返回值很多就是ArrayList。有可能你要查询的数据不存在，要么返回null，要么返回一个没有元素的ArrayList对象。
- Vector因为版本古老，支持Enumeration迭代器。但是该迭代器不支持快速失败。而Iterator和ListIterator迭代器支持快速失败。如果在迭代器创建后的任意时间从结构上修改了向量（通过迭代器自身的remove或add方法之外的任何其他方式），则迭代器将抛出ConcurrentModificationException。因此，面对并发的修改，迭代器很快就完全失败，而不是冒着在将来不确定的时间任意发生不确定行为的风险。

(2) Vector部分源码分析

```

public Vector() {
    this(10); //指定初始容量initialCapacity为10
}
public Vector(int initialCapacity) {
    this(initialCapacity, 0); //指定capacityIncrement增量为0
}
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    //判断了形参初始容量initialCapacity的合法性
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal capacity: "+
                                           initialCapacity);
    //创建了一个Object[]类型的数组
    this.elementData = new Object[initialCapacity]; //默认是10
    //增量，默认是0，如果是0，后面就按照2倍增加，如果不是0，后面就按照你指定的增量进行增量
    this.capacityIncrement = capacityIncrement;
}

```

```

//synchronized意味着线程安全的
    public synchronized boolean add(E e) {
        modCount++;
        //看是否需要扩容
        ensureCapacityHelper(elementCount + 1);
        //把新的元素存入[elementCount], 存入后, elementCount元素的个数增1
        elementData[elementCount++] = e;
        return true;
    }

    private void ensureCapacityHelper(int minCapacity) {
        // overflow-conscious code
        //看是否超过了当前数组的容量
        if (minCapacity - elementData.length > 0)
            grow(minCapacity); //扩容
    }

    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length; //获取目前数组的长度
        //如果capacityIncrement增量是0, 新容量 = oldCapacity的2倍
        //如果capacityIncrement增量是不是0, 新容量 = oldCapacity + capacityIncrement增量;
        int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                         capacityIncrement : oldCapacity);

        //如果按照上面计算的新容量还不够, 就按照你指定的需要的最小容量来扩容minCapacity
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;

        //如果新容量超过了最大数组限制, 那么单独处理
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);

        //把旧数组中的数据复制到新数组中, 新数组的长度为newCapacity
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}

```

```

public boolean remove(Object o) {
    return removeElement(o);
}

public synchronized boolean removeElement(Object obj) {
    modCount++;
    //查找obj在当前Vector中的下标
    int i = indexOf(obj);
    //如果i>=0, 说明存在, 删除[i]位置的元素
    if (i >= 0) {
        removeElementAt(i);
        return true;
    }
    return false;
}

public int indexOf(Object o) {
}

```

```

        return indexOf(o, 0);
    }

    public synchronized int indexOf(Object o, int index) {
        if (o == null) {//要查找的元素是null值
            for (int i = index ; i < elementCount ; i++)
                if (elementData[i]==null)//如果是null值, 用==null判断
                    return i;
        } else {//要查找的元素是非null值
            for (int i = index ; i < elementCount ; i++)
                if (o.equals(elementData[i]))//如果是非null值, 用equals判断
                    return i;
        }
        return -1;
    }

    public synchronized void removeElementAt(int index) {
        modCount++;
        //判断下标的合法性
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >= " +
                elementCount);
        }
        else if (index < 0) {
            throw new ArrayIndexOutOfBoundsException(index);
        }

        //j是要移动的元素的个数
        int j = elementCount - index - 1;
        //如果需要移动元素, 就调用System.arraycopy进行移动
        if (j > 0) {
            //把index+1位置以及后面的元素往前移动
            //index+1的位置的元素移动到index位置, 依次类推
            //一共移动j个
            System.arraycopy(elementData, index + 1, elementData, index, j);
        }
        //元素的总个数减少
        elementCount--;
        //将elementData[elementCount]这个位置置空, 用来添加新元素, 位置的元素等着被GC回收
        elementData[elementCount] = null; /* to let gc do its work */
    }
}

```

(3) ArrayList部分源码分析

JDK1.6:

```
public ArrayList() {
    this(10); //指定初始容量为10
}
public ArrayList(int initialCapacity) {
    super();
    //检查初始容量的合法性
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    //数组初始化为长度为initialCapacity的数组
    this.elementData = new Object[initialCapacity];
}
```

JDK1.7

```
private static final int DEFAULT_CAPACITY = 10; //默认初始容量10
private static final Object[] EMPTY_ELEMENTDATA = {};
public ArrayList() {
    super();
    this.elementData = EMPTY_ELEMENTDATA; //数组初始化为一个空数组
}
public boolean add(E e) {
    //查看当前数组是否够多存一个元素
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == EMPTY_ELEMENTDATA) { //如果当前数组还是空数组
        //minCapacity按照 默认初始容量和minCapacity中的的最大值处理
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    //看是否需要扩容处理
    ensureExplicitCapacity(minCapacity);
}
//...
```

JDK1.8

```
private static final int DEFAULT_CAPACITY = 10;
private static final Object[] EMPTY_ELEMENTDATA = {};
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA; //初始化为空数组
}
public boolean add(E e) {
    //查看当前数组是否够多存一个元素
    ensureCapacityInternal(size + 1); // Increments modCount!!

    //存入新元素到[size]位置，然后size自增1
    elementData[size++] = e;
```

```

        return true;
    }

    private void ensureCapacityInternal(int minCapacity) {
        //如果当前数组还是空数组
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            //那么minCapacity取DEFAULT_CAPACITY与minCapacity的最大值
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }
        //查看是否需要扩容
        ensureExplicitCapacity(minCapacity);
    }

    private void ensureExplicitCapacity(int minCapacity) {
        modCount++; //修改次数加1

        // 如果需要的最小容量 比 当前数组的长度 大, 即当前数组不够存, 就扩容
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }

    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length; //当前数组容量
        int newCapacity = oldCapacity + (oldCapacity >> 1); //新数组容量是旧数组容量的1.5倍
        //看旧数组的1.5倍是否够
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        //看旧数组的1.5倍是否超过最大数组限制
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);

        //复制一个新数组
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}

```

```

public boolean remove(Object o) {
    //先找到o在当前ArrayList的数组中的下标
    //分o是否为空两种情况讨论
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) { //null值用==比较
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) { //非null值用equals比较
                fastRemove(index);
                return true;
            }
    }
    return false;
}

private void fastRemove(int index) {
    modCount++; //修改次数加1
}

```

```
//需要移动的元素个数
int numMoved = size - index - 1;

//如果需要移动元素，就用System.arraycopy移动元素
if (numMoved > 0)
    System.arraycopy(elementData, index+1, elementData, index,
                     numMoved);

//将elementData[size-1]位置置空，让GC回收空间，元素个数减少
elementData[--size] = null; // clear to let GC do its work
}
```

```
public E remove(int index) {
    rangeCheck(index); //检验index是否合法

    modCount++; //修改次数加1

    //取出[index]位置的元素，[index]位置的元素就是要被删除的元素，用于最后返回被删除的元素
    E oldValue = elementData(index);

    //需要移动的元素个数
    int numMoved = size - index - 1;

    //如果需要移动元素，就用System.arraycopy移动元素
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    //将elementData[size-1]位置置空，让GC回收空间，元素个数减少
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
```

```
public E set(int index, E element) {
    rangeCheck(index); //检验index是否合法

    //取出[index]位置的元素，[index]位置的元素就是要被替换的元素，用于最后返回被替换的元素
    E oldValue = elementData(index);
    //用element替换[index]位置的元素
    elementData[index] = element;
    return oldValue;
}

public E get(int index) {
    rangeCheck(index); //检验index是否合法

    return elementData(index); //返回[index]位置的元素
}
```

```
public int indexOf(Object o) {
    //分为o是否为空两种情况
    if (o == null) {
```

```

//从前往后找
for (int i = 0; i < size; i++)
    if (elementData[i]==null)
        return i;
} else {
    for (int i = 0; i < size; i++)
        if (o.equals(elementData[i]))
            return i;
}
return -1;
}

public int lastIndexOf(Object o) {
    //分为o是否为空两种情况
    if (o == null) {
        //从后往前找
        for (int i = size-1; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = size-1; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

```

13.2.6 链表

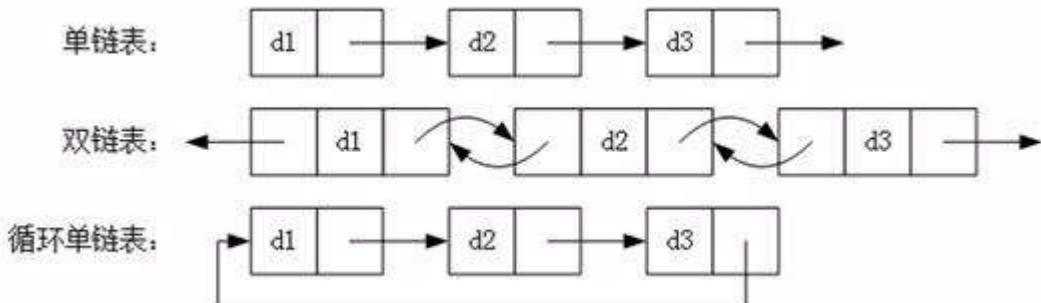
1、链表的特点

逻辑结构：线性结构

物理结构：不要求连续的存储空间

存储特点：数据必须封装到“结点”中，结点包含多个数据项，数据值只是其中的一个数据项，其他的数据项用来记录与之有关的结点的地址。

例如：以下列出几种常见的链式存储结构（当然远不止这些）



2、自定义双链表（选讲）

```
package com.atguigu.list;
```

```
import java.util.Iterator;

public class MyLinkedList<E> implements Iterable<E>{
    private Node first;
    private Node last;
    private int total;

    public void add(E e){
        Node newNode = new Node(last, e, null);

        if(first == null){
            first = newNode;
        }else{
            last.next = newNode;
        }
        last = newNode;
        total++;
    }

    public int size(){
        return total;
    }

    public void delete(Object obj){
        Node find = findNode(obj);
        if(find != null){
            if(find.prev != null){
                find.prev.next = find.next;
            }else{
                first = find.next;
            }
            if(find.next != null){
                find.next.prev = find.prev;
            }else{
                last = find.prev;
            }

            find.prev = null;
            find.next = null;
            find.data = null;

            total--;
        }
    }

    private Node findNode(Object obj){
        Node node = first;
        Node find = null;

        if(obj == null){
            while(node != null){
                if(node.data == null){
                    find = node;
                }
            }
        }
    }
}
```

```

        break;
    }
    node = node.next;
}
}else{
    while(node != null){
        if(obj.equals(node.data)){
            find = node;
            break;
        }
        node = node.next;
    }
}
return find;
}

public boolean contains(Object obj){
    return findNode(obj) != null;
}

public void update(E old, E value){
    Node find = findNode(old);
    if(find != null){
        find.data = value;
    }
}

@Override
public Iterator<E> iterator() {
    return new Itr();
}

private class Itr implements Iterator<E>{
    private Node node = first;

    @Override
    public boolean hasNext() {
        return node!=null;
    }

    @Override
    public E next() {
        E value = node.data;
        node = node.next;
        return value;
    }
}

private class Node{
    Node prev;
    E data;
    Node next;
}

```

```

        Node(Node prev, E data, Node next) {
            this.prev = prev;
            this.data = data;
            this.next = next;
        }
    }
}

```

自定义双链表测试：

```

package com.atguigu.list;

public class TestMyLinkedList {
    public static void main(String[] args) {
        MyLinkedList<String> my = new MyLinkedList<>();
        my.add("hello");
        my.add("world");
        my.add(null);
        my.add(null);
        my.add("java");
        my.add("java");
        my.add("atguigu");

        System.out.println("一共有: " + my.size());
        System.out.println("所有元素: ");
        for (String s : my) {
            System.out.println(s);
        }
        System.out.println("-----");
        System.out.println("查找java,null,haha的结果: ");
        System.out.println(my.contains("java"));
        System.out.println(my.contains(null));
        System.out.println(my.contains("haha"));

        System.out.println("-----");
        System.out.println("替换java,null后: ");
        my.update("java","JAVA");
        my.update(null,"chai");
        System.out.println("所有元素: ");
        for (String s : my) {
            System.out.println(s);
        }
        System.out.println("-----");
        System.out.println("删除hello, JAVA, null, atguigu后: ");
        my.delete("hello");
        my.delete("JAVA");
        my.delete(null);
        my.delete("atguigu");
        System.out.println("所有元素: ");
        for (String s : my) {
            System.out.println(s);
        }
    }
}

```

```
    }
}
```

3、核心类库中LinkedList源码分析

Java中有双链表的实现：LinkedList，它是List接口的实现类。

```
int size = 0;
Node<E> first;//记录第一个结点的位置
Node<E> last;//记录最后一个结点的位置

private static class Node<E> {
    E item;//元素数据
    Node<E> next;//下一个结点
    Node<E> prev;//前一个结点

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

```
public boolean add(E e) {
    linkLast(e); //默认把新元素链接到链表尾部
    return true;
}
void linkLast(E e) {
    final Node<E> l = last; //用l 记录原来的最后一个结点

    //创建新结点
    final Node<E> newNode = new Node<E>(l, e, null);
    //现在的新结点是最后一个结点了
    last = newNode;

    //如果l==null, 说明原来的链表是空的
    if (l == null)
        //那么新结点同时也是第一个结点
        first = newNode;
    else
        //否则把新结点链接到原来的最后一个结点的next中
        l.next = newNode;
    //元素个数增加
    size++;
    //修改次数增加
    modCount++;
}

public void add(int index, E element) {
    checkPositionIndex(index); //检查index范围

    if (index == size)//如果index==size, 连接到当前链表的尾部
```

```

        linkLast(element);
    else
        linkBefore(element, node(index));
}

Node<E> node(int index) {
    // assert isElementIndex(index);

    //如果index<size/2, 就从前往后找目标结点
    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else { //否则从后往前找目标结点
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}

//把新结点插入到[index]位置的结点succ前面
void linkBefore(E e, Node<E> succ) { //succ是[index]位置对应的结点
    // assert succ != null;
    final Node<E> pred = succ.prev; // [index]位置的前一个结点

    //新结点的prev是原来[index]位置的前一个结点
    //新结点的next是原来[index]位置的结点
    final Node<E> newNode = new Node<E>(pred, e, succ);

    // [index]位置对应的结点的prev指向新结点
    succ.prev = newNode;

    //如果原来[index]位置对应的结点是第一个结点, 那么现在新结点是第一个结点
    if (pred == null)
        first = newNode;
    else
        pred.next = newNode; //原来[index]位置的前一个结点的next指向新结点
    size++;
    modCount++;
}

```

```

public boolean remove(Object o) {
    //分o是否为空两种情况
    if (o == null) {
        //找到o对应的结点x
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x); //删除x结点
                return true;
            }
        }
    }
}

```

```

} else {
    //找到o对应的结点x
    for (Node<E> x = first; x != null; x = x.next) {
        if (o.equals(x.item)) {
            unlink(x); //删除x结点
            return true;
        }
    }
}
return false;
}

E unlink(Node<E> x) { //x是要被删除的结点
    // assert x != null;
    final E element = x.item; //被删除结点的数据
    final Node<E> next = x.next; //被删除结点的下一个结点
    final Node<E> prev = x.prev; //被删除结点的上一个结点

    //如果被删除结点的前面没有结点，说明被删除结点是第一个结点
    if (prev == null) {
        //那么被删除结点的下一个结点变为第一个结点
        first = next;
    } else { //被删除结点不是第一个结点
        //被删除结点的上一个结点的next指向被删除结点的下一个结点
        prev.next = next;
        //断开被删除结点与上一个结点的链接
        x.prev = null; //使得GC回收
    }

    //如果被删除结点的后面没有结点，说明被删除结点是最后一个结点
    if (next == null) {
        //那么被删除结点的上一个结点变为最后一个结点
        last = prev;
    } else { //被删除结点不是最后一个结点
        //被删除结点的下一个结点的prev执行被删除结点的上一个结点
        next.prev = prev;
        //断开被删除结点与下一个结点的连接
        x.next = null; //使得GC回收
    }

    //把被删除结点的数据也置空，使得GC回收
    x.item = null;
    //元素个数减少
    size--;
    //修改次数增加
    modCount++;
    //返回被删除结点的数据
    return element;
}

```

4、自定义单链表（选讲）

```

package com.atguigu.list;

import java.util.Iterator;

```

```

public class MyOneWayLinkedList<E> implements Iterable<E>{
    private Node head;
    private int total;

    public void add(E e){
        Node newNode = new Node(e, null);
        if(head == null){
            head = newNode;
        }else{
            Node node = head;
            while(node.next!=null){
                node = node.next;
            }
            node.next = newNode;
        }
        total++;
    }

    private Node[] findNodes(Object obj){
        Node[] result = new MyOneWayLinkedList.Node[2];
        Node node = head;
        Node find = null;
        Node beforeFind = null;

        if(obj == null){
            while(node != null){
                if(node.data == null){
                    find = node;
                    break;
                }
                beforeFind = node;
                node = node.next;
            }
        }else{
            while(node != null){
                if(obj.equals(node.data)){
                    find = node;
                    break;
                }
                beforeFind = node;
                node = node.next;
            }
        }
        result[0] = beforeFind;
        result[1] = find;
        return result;
    }

    public void delete(Object obj){
        Node[] nodes = findNodes(obj);
        Node beforeFind = nodes[0];

```

```

        Node find = nodes[1];
        if(find != null){
            if(beforeFind == null){
                head = find.next;
            }else {
                beforeFind.next = find.next;
            }
            total--;
        }
    }

    private Node findNode(Object obj){
        return findNodes(obj)[1];
    }

    public boolean contains(Object obj){
        return findNode(obj) != null;
    }

    public void update(E old, E value) {
        Node find = findNode(old);
        if(find != null){
            find.data = value;
        }
    }

    public int size() {
        return total;
    }

    @Override
    public Iterator<E> iterator() {
        return new Itr();
    }

    private class Itr implements Iterator<E>{
        private Node node = head;

        @Override
        public boolean hasNext() {
            return node != null;
        }

        @Override
        public E next() {
            E value = node.data;
            node = node.next;
            return value;
        }
    }

    private class Node{

```

```
E data;
Node next;

Node(E data, Node next) {
    this.data = data;
    this.next = next;
}
}
```

```
package com.atguigu.list;

public class TestMyOneWayLinkedList{
    public static void main(String[] args) {
        MyOneWayLinkedList<String> my = new MyOneWayLinkedList<>();
        my.add("hello");
        my.add("world");
        my.add(null);
        my.add(null);
        my.add("java");
        my.add("java");
        my.add("atguigu");

        System.out.println("一共有: " + my.size());
        System.out.println("所有元素: ");
        for (String s : my) {
            System.out.println(s);
        }
        System.out.println("-----");
        System.out.println("查找java,null,haha的结果: ");
        System.out.println(my.contains("java"));
        System.out.println(my.contains(null));
        System.out.println(my.contains("haha"));
        System.out.println("-----");
        System.out.println("替换java,null后: ");
        my.update("java","JAVA");
        my.update(null,"chai");
        System.out.println("所有元素: ");
        for (String s : my) {
            System.out.println(s);
        }
        System.out.println("-----");
        System.out.println("删除hello,JAVA,null,atguigu后: ");
        my.delete("hello");
        my.delete("JAVA");
        my.delete(null);
        my.delete("atguigu");
        System.out.println("所有元素: ");
        for (String s : my) {
            System.out.println(s);
        }
    }
}
```

5、链表与动态数组的区别

动态数组底层的物理结构是数组，因此根据索引访问的效率非常高。但是非末尾位置的插入和删除效率不高，因为涉及到移动元素。另外添加操作时涉及到扩容问题，就会增加时空消耗。

链表底层的物理结构是链表，因此根据索引访问的效率不高，但是插入和删除不需要移动元素，只需要修改前后元素的指向关系即可，而且链表的添加不会涉及到扩容问题。

13.2.7 栈

堆栈是一种先进后出 (FILO: first in last out) 或后进先出 (LIFO: last in first out) 的结构。

栈只是逻辑结构，其物理结构可以是数组，也可以是链表，即栈结构分为顺序栈和链式栈。

核心类库中的栈结构有Stack和LinkdeList。Stack就是顺序栈，它是Vector的子类。LinkedList是链式栈。

体现栈结构的操作方法：

- peek()方法：查看栈顶元素，不弹出
- pop()方法：弹出栈
- push(E e)方法：压入栈

```
package com.atguigu.list;

import org.junit.Test;

import java.util.LinkedList;
import java.util.Stack;

public class TestStack {
    @Test
    public void test1(){
        Stack<Integer> list = new Stack<>();
        list.push(1);
        list.push(2);
        list.push(3);

        System.out.println("list = " + list);

        System.out.println("list.peek()=" + list.peek());
        System.out.println("list.peek()=" + list.peek());
        System.out.println("list.peek()=" + list.peek());

        /*
        System.out.println("list.pop() =" + list.pop());
        System.out.println("list.pop() =" + list.pop());
        System.out.println("list.pop() =" + list.pop());
        System.out.println("list.pop() =" + list.pop());//java.util.NoSuchElementException
        */

        while(!list.empty()){
            System.out.println("list.pop() =" + list.pop());
        }
    }
}
```

```

}

@Test
public void test2(){
    LinkedList<Integer> list = new LinkedList<>();
    list.push(1);
    list.push(2);
    list.push(3);

    System.out.println("list = " + list);

    System.out.println("list.peek()=" + list.peek());
    System.out.println("list.peek()=" + list.peek());
    System.out.println("list.peek()=" + list.peek());

    /*
    System.out.println("list.pop() =" + list.pop());
    System.out.println("list.pop() =" + list.pop());
    System.out.println("list.pop() =" + list.pop());
    System.out.println("list.pop() =" + list.pop());//java.util.NoSuchElementException
    */

    while(!list.isEmpty()){
        System.out.println("list.pop() =" + list.pop());
    }
}
}

```

13.3 队列

队列 (Queue) 是一种 (但并非一定) 先进先出 (FIFO) 的结构。

队列是逻辑结构，其物理结构可以是数组，也可以是链表。队列有普通队列、双端队列、并发队列等等，核心类库中的队列实现类有很多（后面会学到很多），`LinkedList`是双端队列的实现类。

`==Queue==`除了基本的 `Collection` 操作外，`==队列==`还提供其他的插入、提取和检查操作。每个方法都存在两种形式：一种抛出异常（操作失败时），另一种返回一个特殊值（`null` 或 `false`，具体取决于操作）。`Queue` 实现通常不允许插入元素，尽管某些实现（如）并不禁止插入。即使在允许 `null` 的实现中，也不应该将插入到 中，因为也用作方法的一个特殊返回值，表明队列不包含元素。

| | 抛出异常 | 返回特殊值 |
|----|------------------------|-----------------------|
| 插入 | <code>add(e)</code> | <code>offer(e)</code> |
| 移除 | <code>remove()</code> | <code>poll()</code> |
| 检查 | <code>element()</code> | <code>peek()</code> |

`==Deque==`，名称 `deque` 是“double ended queue”（双端队列）的缩写，通常读为“deck”。此接口定义在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。每种方法都存在两种形式：一种形式在操作失败时抛出异常，另一种形式返回一个特殊值（`null` 或 `false`，具体取决于操作）。`Deque` 接口的实现类有 `ArrayDeque` 和 `LinkedList`，它们一个底层是使用数组实现，一个使用双向链表实现。

| | 第一个元素（头部） | | 最后一个元素（尾部） | |
|-----------|------------------|---------------|-------------------|--------------|
| | 抛出异常 | 特殊值 | 抛出异常 | 特殊值 |
| 插入 | addFirst(e) | offerFirst(e) | addLast(e) | offerLast(e) |
| 移除 | removeFirst() | pollFirst() | removeLast() | pollLast() |
| 检查 | getFirst() | peekFirst() | getLast() | peekLast() |

此接口扩展了 `Queue` 接口。在将双端队列用作队列时，将得到 FIFO（先进先出）行为。将元素添加到双端队列的末尾，从双端队列的开头移除元素。从 `Queue` 接口继承的方法完全等效于 `Deque` 方法，如下表所示：

| <code>Queue</code> 方法 | 等效 <code>Deque</code> 方法 |
|-----------------------|---------------------------------|
| add(e) | addLast(e) |
| offer(e) | offerLast(e) |
| remove() | removeFirst() |
| poll() | pollFirst() |
| element() | getFirst() |
| peek() | peekFirst() |

双端队列也可用作 LIFO（后进先出）堆栈。应优先使用此接口而不是遗留 `Stack` 类。在将双端队列用作堆栈时，元素被推入双端队列的开头并从双端队列开头弹出。堆栈方法完全等效于 `Deque` 方法，如下表所示：

| 堆栈方法 | 等效 <code>Deque</code> 方法 |
|-------------|---------------------------------|
| push(e) | addFirst(e) |
| pop() | removeFirst() |
| peek() | peekFirst() |

结论：`Deque` 接口的实现类既可以用作 FILO 堆栈使用，又可以用作 FIFO 队列使用。

```
package com.atguigu.queue;

import java.util.LinkedList;

public class TestQueue {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.addLast("张三");
        list.addLast("李四");
        list.addLast("王五");
        list.addLast("赵六");
    }
}
```

```
        while (!list.isEmpty()){
            System.out.println("list.removeFirst()=" + list.removeFirst());
        }
    }
}
```

13.4 Set集合

Set接口是Collection的子接口，set接口没有提供额外的方法。但是比collection接口更加严格了。

Set集合不允许包含相同的元素，如果试把两个相同的元素加入同一个Set集合中，则添加操作失败。

Set集合支持的遍历方式和Collection集合一样：foreach和Iterator。

Set的常用实现类有：HashSet、TreeSet、LinkedHashSet。

13.4.1 HashSet

HashSet是Set接口的典型实现，大多数时候使用Set集合时都使用这个实现类。

`java.util.HashSet`底层的实现其实是一个`java.util.HashMap`支持，然后HashMap的底层物理实现是一个Hash表。（什么是哈希表，下一节在HashMap小节在细讲，这里先不展开）

HashSet按Hash算法来存储集合中的元素，因此具有很好的存取和查找性能。HashSet集合判断两个元素相等的标准：两个对象通过`hashCode()`方法比较相等，并且两个对象的`equals()`方法返回值也相等。因此，存储到HashSet的元素要重写`hashCode`和`equals`方法。

```
package com.atguigu.set;

import java.util.Objects;

public class MyDate {
    private int year;
    private int month;
    private int day;

    public MyDate(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MyDate myDate = (MyDate) o;
        return year == myDate.year &&
               month == myDate.month &&
               day == myDate.day;
    }
}
```

```
    @Override
    public int hashCode() {
        return Objects.hash(year, month, day);
    }

    @Override
    public String toString() {
        return "MyDate{" +
            "year=" + year +
            ", month=" + month +
            ", day=" + day +
            '}';
    }
}
```

```
package com.atguigu.set;

import org.junit.Test;

import java.util.HashSet;

public class TestHashSet {
    @Test
    public void test01(){
        HashSet<String> set = new HashSet<>();
        set.add("张三");
        set.add("张三");
        set.add("李四");
        set.add("王五");
        set.add("王五");
        set.add("赵六");

        System.out.println("set = " + set);//不允许重复, 无序
    }

    @Test
    public void test02(){
        HashSet<MyDate> set = new HashSet<>();
        set.add(new MyDate(2021,1,1));
        set.add(new MyDate(2021,1,1));
        set.add(new MyDate(2022,2,4));
        set.add(new MyDate(2022,2,4));

        System.out.println("set = " + set);//不允许重复, 无序
    }
}
```

13.4.2 LinkedHashSet

LinkedHashSet是HashSet的子类，它在HashSet的基础上，在结点中增加两个属性before和after维护了结点的前后添加顺序。`java.util.LinkedHashSet`，它是链表和哈希表组合的一个数据存储结构。LinkedHashSet插入性能略低于HashSet，但在迭代访问Set里的全部元素时有很好的性能。

```
package com.atguigu.set;

import org.junit.Test;

import java.util.LinkedHashSet;

public class TestLinkedHashSet {
    @Test
    public void test01(){
        LinkedHashSet<String> set = new LinkedHashSet<>();
        set.add("张三");
        set.add("张三");
        set.add("李四");
        set.add("王五");
        set.add("王五");
        set.add("赵六");

        System.out.println("set = " + set); //不允许重复，体现添加顺序
    }
}
```

13.4.3 TreeSet

TreeSet里面维护了一个TreeMap，底层是基于红黑树实现的！

TreeSet特点：

- 不允许重复
- 实现排序：自然排序或定制排序

如何实现去重的？

如果使用的是自然排序，则通过调用实现的`compareTo`方法
如果使用的是定制排序，则通过调用比较器的`compare`方法

如何排序？

方式一：自然排序
让待添加的元素类型实现`Comparable`接口，并重写`compareTo`方法

方式二：定制排序
创建Set对象时，指定`Comparator`比较器接口，并实现`compare`方法

示例代码：

```
package com.atguigu.set;
```

```
import org.junit.Test;

import java.text.Collator;
import java.util.Arrays;
import java.util.Locale;
import java.util.TreeSet;

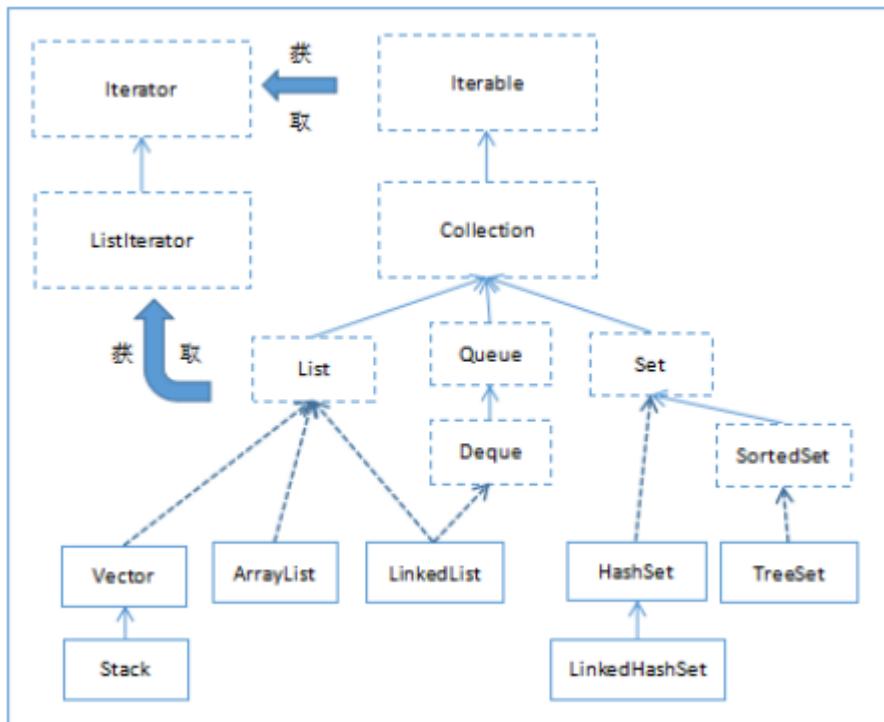
public class TestTreeSet {
    @Test
    public void test01(){
        TreeSet<String> set = new TreeSet<>();
        set.add("张三");
        set.add("张三");
        set.add("张飞");
        set.add("李四");
        set.add("王五");
        set.add("王五");
        set.add("赵六");

        System.out.println("set = " + set);
        //不允许重复，体现大小顺序，String对象自然排序是依据unicode编码值大小
        for (String s : set) {
            char[] chars = s.toCharArray();
            int[] codes = new int[chars.length];
            for (int i = 0; i < codes.length; i++) {
                codes[i] = chars[i];
            }
            System.out.println(s + ":" + Arrays.toString(codes));
        }
    }

    @Test
    public void test02(){
        //Collator.getInstance(Locale.CHINA)是Comparator接口的实现类对象
        //String对象将按照字典顺序排列
        TreeSet<String> set = new TreeSet<>(Collator.getInstance(Locale.CHINA));
        set.add("张三");
        set.add("张三");
        set.add("张飞");
        set.add("李四");
        set.add("王五");
        set.add("王五");
        set.add("赵六");

        System.out.println("set = " + set);
        //不允许重复，体现大小顺序
    }
}
```

13.5 Collection系列的集合框架图



13.6 Map

13.6.1 概述

现实生活中，我们常会看到这样的一种集合：IP地址与主机名，身份证号与个人，系统用户名与系统用户对象等，这种一一对应的关系，就叫做映射。Java提供了专门的集合类用来存放这种对象关系的对象，即 `java.util.Map<K,V>` 接口。

我们通过查看 `Map` 接口描述，发现 `Map<K,V>` 接口下的集合与 `collection<E>` 接口下的集合，它们存储数据的形式不同。

- `Collection` 中的集合，元素是孤立存在的（理解为单身），向集合中存储元素采用一个个元素的方式存储。
- `Map` 中的集合，元素是成对存在的（理解为夫妻）。每个元素由键与值两部分组成，通过键可以找对所对应的值。
- `Collection` 中的集合称为单列集合，`Map` 中的集合称为双列集合。
- 需要注意的是，`Map` 中的集合不能包含重复的键，值可以重复；每个键只能对应一个值（这个值可以是单个值，也可以是个数组或集合值）。

Collection 接口 定义了 单列集合规范
每次 存储 一个元素 单个元素

单身集合

Collection<E>

范冰冰
王菲
张柏芝
杨颖
杨幂
孙俪
马伊琍
姚笛

通过 键 可以找 对应的值
1 : 键唯一 (值可以重复)
2 : 键和值——映射
一个键对应一个值
(值可以是单个值 , 也可以
是个数组或集合)
3 : 靠键维护他们关系

Map 接口
定义了 双列集合的规范
Map<K,V> K 代表键的类型

每次 存储 一对 元素 情侣对集合 V 代表值的类型

Key 键 Value 值

| Key | Value |
|-----|---------|
| 范冰冰 | 李晨 |
| 王菲 | 谢霆锋、李亚鹏 |
| 张柏芝 | 谢霆锋 |
| 杨颖 | 黄晓明 |
| 杨幂 | 刘恺威 |
| 孙俪 | 邓超 |
| 马伊琍 | 文章 |
| 姚笛 | 文章 |

13.6.2 Map常用方法

1、添加操作

- V put(K key,V value)
- void putAll(Map<? extends K,? extends V> m)

2、删除

- void clear()
- V remove(Object key)

3、元素查询的操作

- V get(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- boolean isEmpty()

4、元视图操作的方法:

- Set keySet()
- Collection values()
- Set<Map.Entry<K,V>> entrySet()

5、其他方法

- int size()

```
package com.atguigu.map;

import java.util.HashMap;

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

```

//创建 map对象
HashMap<String, String> map = new HashMap<String, String>();

//添加元素到集合
map.put("黄晓明", "杨颖");
map.put("文章", "马伊琍");
map.put("文章", "姚笛");
map.put("邓超", "孙俪");
System.out.println(map);

//String remove(String key)
System.out.println(map.remove("黄晓明"));
System.out.println(map);

// 想要查看 黄晓明的媳妇 是谁
System.out.println(map.get("黄晓明"));
System.out.println(map.get("邓超"));
}

}

```

tips:

使用put方法时，若指定的键(key)在集合中没有，则没有这个键对应的值，返回null，并把指定的键值添加到集合中；

若指定的键(key)在集合中存在，则返回值为集合中键对应的值（该值为替换前的值），并把指定键所对应的值，替换成指定的新值。

13.6.3 Map集合的遍历

Collection集合的遍历： (1) foreach (2) 通过Iterator对象遍历

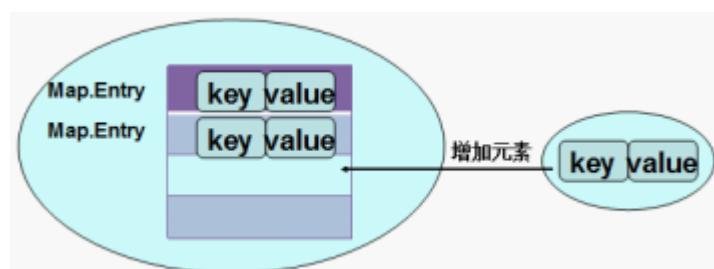
Map的遍历，不能支持foreach，因为Map接口没有继承java.lang.Iterable接口，也没有实现Iterator iterator()方法。只能用如下方式遍历：

(1) 分开遍历：

- 单独遍历所有key
- 单独遍历所有value

(2) 成对遍历：

- 遍历的是映射关系Map.Entry类型的对象，Map.Entry是Map接口的内部接口。每一种Map内部有自己的Map.Entry的实现类。在Map中存储数据，实际上是将Key--->value的数据存储在Map.Entry接口的实例中，再在Map集合中插入Map.Entry的实例化对象，如图示：



```
package com.atguigu.map;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class TestMapIterate {
    public static void main(String[] args) {
        HashMap<String, String> map = new HashMap<>();
        map.put("许仙", "白娘子");
        map.put("董永", "七仙女");
        map.put("牛郎", "织女");
        map.put("许仙", "小青");

        System.out.println("所有的key:");
        Set<String> keySet = map.keySet();
        for (String key : keySet) {
            System.out.println(key);
        }

        System.out.println("所有的value: ");
        Collection<String> values = map.values();
        for (String value : values) {
            System.out.println(value);
        }

        System.out.println("所有的映射关系");
        Set<Map.Entry<String, String>> entrySet = map.entrySet();
        for (Map.Entry<String, String> entry : entrySet) {
            System.out.println(entry);
            System.out.println(entry.getKey() + "->" + entry.getValue());
        }
    }
}
```

13.6.4 Map的实现类们

Map接口的常用实现类：HashMap、TreeMap、LinkedHashMap和Properties。其中HashMap是Map接口使用频率最高的实现类。

1、HashMap和Hashtable

HashMap和Hashtable都是哈希表。HashMap和Hashtable判断两个key相等的标准是：两个key的hashCode值相等，并且equals()方法也返回true。因此，为了成功地在哈希表中存储和获取对象，用作键的对象必须实现hashCode方法和equals方法。

- Hashtable是线程安全的，任何非null对象都可以用作键或值。
- HashMap是线程不安全的，并允许使用null值和null键。

示例代码：添加员工姓名为key，薪资为value

```
package com.atguigu.map;

import org.junit.Test;

import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;
import java.util.Set;

public class TestHashMap {
    @Test
    public void test01(){
        HashMap<String,Double> map = new HashMap<>();
        map.put("张三", 10000.0);
        //key相同，新的value会覆盖原来的value
        //因为String重写了hashCode和equals方法
        map.put("张三", 12000.0);
        map.put("李四", 14000.0);
        //HashMap支持key和value为null值
        String name = null;
        Double salary = null;
        map.put(name, salary);

        Set<Map.Entry<String, Double>> entrySet = map.entrySet();
        for (Map.Entry<String, Double> entry : entrySet) {
            System.out.println(entry);
        }
    }

    @Test
    public void test02(){
        Hashtable<String,Double> map = new Hashtable<>();
        map.put("张三", 10000.0);
        //key相同，新的value会覆盖原来的value
        //因为String重写了hashCode和equals方法
        map.put("张三", 12000.0);
        map.put("李四", 14000.0);
        //Hashtable不支持key和value为null值
        /*String name = null;
        Double salary = null;
        map.put(name, salary);*/

        Set<Map.Entry<String, Double>> entrySet = map.entrySet();
        for (Map.Entry<String, Double> entry : entrySet) {
            System.out.println(entry);
        }
    }
}
```

2、LinkedHashMap

LinkedHashMap 是 HashMap 的子类。此实现与 HashMap 的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序通常就是将键插入到映射中的顺序（插入顺序）。

示例代码：添加员工姓名为key，薪资为value

```
package com.atguigu.map;

import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;

public class TestLinkedHashMap {
    public static void main(String[] args) {
        LinkedHashMap<String,Double> map = new LinkedHashMap<>();
        map.put("张三", 10000.0);
        //key相同，新的value会覆盖原来的value
        //因为String重写了hashCode和equals方法
        map.put("张三", 12000.0);
        map.put("李四", 14000.0);
        //HashMap支持key和value为null值
        String name = null;
        Double salary = null;
        map.put(name, salary);

        Set<Map.Entry<String, Double>> entrySet = map.entrySet();
        for (Map.Entry<String, Double> entry : entrySet) {
            System.out.println(entry);
        }
    }
}
```

3、TreeMap

基于红黑树（Red-Black tree）的 NavigableMap 实现。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

代码示例：添加员工姓名为key，薪资为value

```
package com.atguigu.map;

import java.util.Comparator;
import java.util.Map.Entry;
import java.util.Set;
import java.util.TreeMap;

import org.junit.Test;

public class TestTreeMap {
    @Test
    public void test1() {
        TreeMap<String,Integer> map = new TreeMap<>();
        map.put("Jack", 11000);
        map.put("Alice", 12000);
        map.put("zhangsan", 13000);
        map.put("baitao", 14000);
        map.put("Lucy", 15000);
    }
}
```

```

        //String实现了Comparable接口，默认按照Unicode编码值排序
        Set<Entry<String, Integer>> entrySet = map.entrySet();
        for (Entry<String, Integer> entry : entrySet) {
            System.out.println(entry);
        }
    }

    @Test
    public void test2() {
        //指定定制比较器Comparator，按照Unicode编码值排序，但是忽略大小写
        TreeMap<String, Integer> map = new TreeMap<>(new Comparator<String>() {

            @Override
            public int compare(String o1, String o2) {
                return o1.compareToIgnoreCase(o2);
            }
        });

        map.put("Jack", 11000);
        map.put("Alice", 12000);
        map.put("zhangsan", 13000);
        map.put("baitao", 14000);
        map.put("Lucy", 15000);

        Set<Entry<String, Integer>> entrySet = map.entrySet();
        for (Entry<String, Integer> entry : entrySet) {
            System.out.println(entry);
        }
    }
}

```

4、Properties

Properties 类是 Hashtable 的子类，Properties 可保存在流中或从流中加载。属性列表中每个键及其对应值都是一个字符串。

存取数据时，建议使用setProperty(String key, String value)方法和getProperty(String key)方法。

代码示例：

```

package com.atguigu.map;

import org.junit.Test;

import java.util.Properties;

public class TestProperties {
    @Test
    public void test01() {
        Properties properties = System.getProperties();
        String fileEncoding = properties.getProperty("file.encoding");//当前源文件字符编码
        System.out.println("fileEncoding = " + fileEncoding);
    }
}

```

```

public void test02() {
    Properties properties = new Properties();
    properties.setProperty("user", "chai");
    properties.setProperty("password", "123456");
    System.out.println(properties);
}

}

```

13.6.5 Set集合与Map集合的关系

Set的内部实现其实是一个Map。即HashSet的内部实现是一个HashMap， TreeSet的内部实现是一个TreeMap， LinkedHashSet的内部实现是一个LinkedHashMap。

部分源代码摘要：

HashSet源码：

```

public HashSet() {
    map = new HashMap<>();
}

public HashSet(Collection<? extends E> c) {
    map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
    addAll(c);
}

public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<>(initialCapacity, loadFactor);
}

public HashSet(int initialCapacity) {
    map = new HashMap<>(initialCapacity);
}

//这个构造器是给子类LinkedHashSet调用的
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}

```

LinkedHashSet源码：

```

public LinkedHashSet(int initialCapacity, float loadFactor) {
    super(initialCapacity, loadFactor, true); //调用HashSet的某个构造器
}

public LinkedHashSet(int initialCapacity) {
    super(initialCapacity, .75f, true); //调用HashSet的某个构造器
}

public LinkedHashSet() {
    super(16, .75f, true);
}

```

```
}

public LinkedHashSet(Collection<? extends E> c) {
    super(Math.max(2*c.size(), 11), .75f, true); //调用HashSet的某个构造器
    addAll(c);
}
```

TreeSet源码：

```
public TreeSet() {
    this(new TreeMap<E, Object>());
}

public TreeSet(Comparator<? super E> comparator) {
    this(new TreeMap<>(comparator));
}

public TreeSet(Collection<? extends E> c) {
    this();
    addAll(c);
}

public TreeSet(SortedSet<E> s) {
    this(s.comparator());
    addAll(s);
}
```

但是，咱们存到Set中只有一个元素，又是怎么变成(key,value)的呢？

以HashSet中的源码为例：

```
private static final Object PRESENT = new Object();
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
public Iterator<E> iterator() {
    return map.keySet().iterator();
}
```

原来是，把添加到Set中的元素作为内部实现map的key，然后用一个常量对象PRESENT对象，作为value。

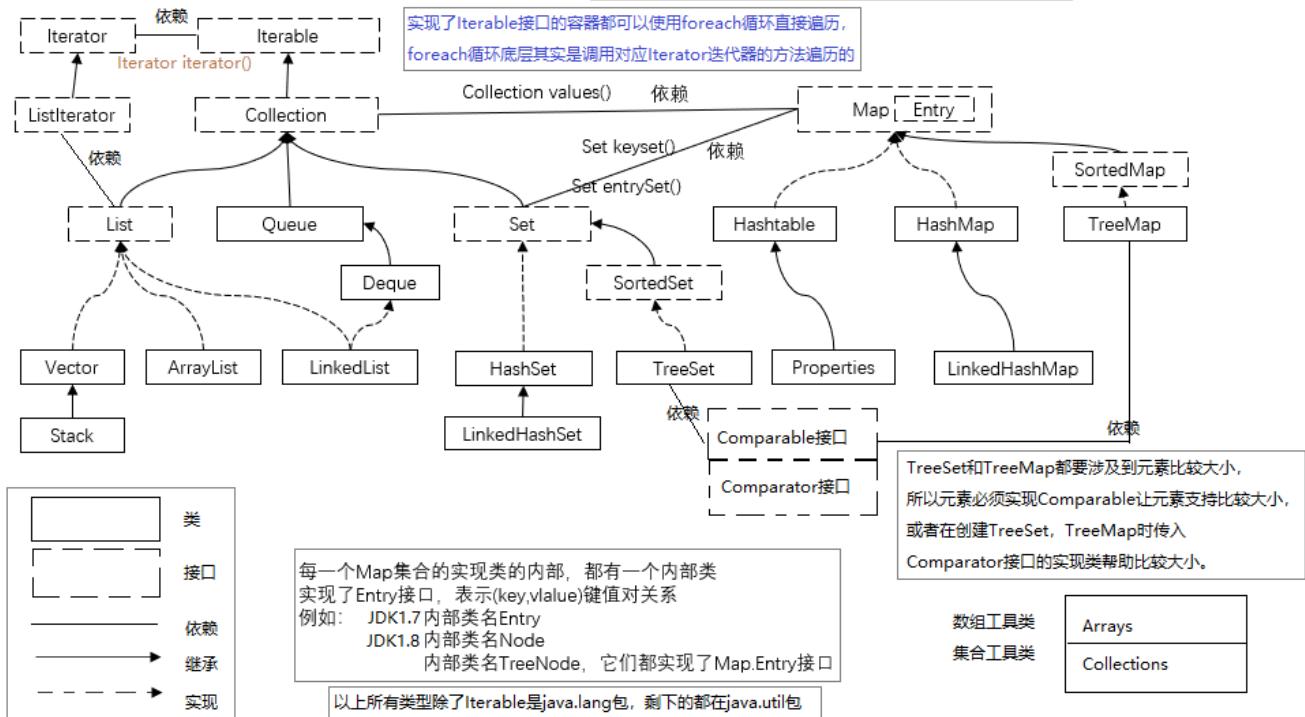
这是因为Set的元素不可重复和Map的key不可重复有相同特点。Map有一个方法keySet()可以返回所有key。

13.7 集合框架

集合框架图

在Collection系列集合的实现类中都有一个内部类实现了Iterator接口，然后通过iterator0方法，返回这个内部对象。
例如：ArrayList中有一个Itr内部类，它实现了Iterator接口，
ArrayList的集合调用iterator()返回的迭代器对象是Itr类的对象。

Map的values()方法、keySet()方法、entrySet()方法
返回的Collection、Set接口的实现类对象都是在Map系列
的集合的实现类中有对应的内部类实现类实现了Collection、
Set接口，而不是我们这张图中看到的ArrayList、HashSet这些。



当然实际开发中使用的集合类型远不止这些，比如还有ConcurrentMap, BlockingQueue等，只是以上这些是最基础的，先掌握这些，才能拓展更多

13.8 Collections工具类

参考操作数组的工具类：Arrays。

Collections 是一个操作 Set、List 和 Map 等集合的工具类。Collections 中提供了一系列静态的方法对集合元素进行排序、查询和修改等操作，还提供了对集合对象设置不可变、对集合对象实现同步控制等方法：

- `public static boolean addAll(Collection<? super T> c, T... elements)` 将所有指定元素添加到指定 collection 中。
- `public static int binarySearch(List<? extends Comparable<? super T>> list, T key)` 在 List 集合中查找某个元素的下标，但是 List 的元素必须是 T 或 T 的子类对象，而且必须是可比较大小的，即支持自然排序的。而且集合也事先必须是有序的，否则结果不确定。
- `public static int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)` 在 List 集合中查找某个元素的下标，但是 List 的元素必须是 T 或 T 的子类对象，而且集合也事先必须是按照 c 比较器规则进行排序过的，否则结果不确定。
- `public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)` 在 coll 集合中找出最大的元素，集合中的对象必须是 T 或 T 的子类对象，而且支持自然排序
- `public static T max(Collection<? extends T> coll, Comparator<? super T> comp)` 在 coll 集合中找出最大的元素，集合中的对象必须是 T 或 T 的子类对象，按照比较器 comp 找出最大者
- `public static void reverse(List<?> list)` 反转指定列表 list 中元素的顺序。
- `public static void shuffle(List<?> list)` List 集合元素进行随机排序，类似洗牌

- public static <T extends Comparable<? super T>> void sort(List list)根据元素的自然顺序对指定 List 集合元素按升序排序
- public static void sort(List list, Comparator<? super T> c)根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
- public static void swap(List<?> list, int i, int j)将指定 list 集合中的 i 处元素和 j 处元素进行交换
- public static int frequency(Collection<?> c, Object o)返回指定集合中指定元素的出现次数
- public static void copy(List<? super T> dest, List<? extends T> src)将src中的内容复制到dest中
- public static boolean replaceAll(List list, T oldVal, T newVal): 使用newValue替换 List 对象的所有旧值
- Collections 类中提供了多个 synchronizedXxx() 方法，该方法可使将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题
- Collections类中提供了多个unmodifiableXxx()方法，该方法返回指定 Xxx 的不可修改的视图。

```

package com.atguigu.collections;

import org.junit.Test;

import java.text.Collator;
import java.util.*;

public class TestCollections {
    @Test
    public void test11(){
        /*
         * public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal): 使用newValue替换
         * List 对象的所有旧值
         */
        List<String> list = new ArrayList<>();
        Collections.addAll(list, "hello", "java", "world", "hello", "hello");

        Collections.replaceAll(list, "hello", "chai");
        System.out.println(list);
    }

    @Test
    public void test10(){
        List<Integer> list = new ArrayList<>();
        for(int i=1; i<=5; i++){//1-5
            list.add(i);
        }

        List<Integer> list2 = new ArrayList<>();
        for(int i=11; i<=13; i++){//11-13
            list2.add(i);
        }

        list.addAll(list2);
        System.out.println(list); // [1, 2, 3, 4, 5, 11, 12, 13]
    }

    @Test
    public void test09(){
        /*
    
```

```

    * public static <T> void copy(List<? super T> dest,List<? extends T> src)将src中的
内容复制到dest中
    */
    List<Integer> list = new ArrayList<>();
    for(int i=1; i<=5; i++){//1-5
        list.add(i);
    }

    List<Integer> list2 = new ArrayList<>();
    for(int i=11; i<=13; i++){//11-13
        list2.add(i);
    }

    Collections.copy(list, list2);
    System.out.println(list);

    List<Integer> list3 = new ArrayList<>();
    for(int i=11; i<=20; i++){//11-20
        list3.add(i);
    }

    Collections.copy(list, list3); //java.lang.IndexOutOfBoundsException: Source does
not fit in dest
    System.out.println(list);

}

@Test
public void test08(){
    /*
    public static int frequency(Collection<?> c, Object o)返回指定集合中指定元素的出现次数
    */
    List<String> list = new ArrayList<>();
    Collections.addAll(list,"hello","java","world","hello","hello");
    int count = Collections.frequency(list, "hello");
    System.out.println("count = " + count);
}

@Test
public void test07(){
    /*
    public static void swap(List<?> list,int i,int j)将指定 list 集合中的 i 处元素和 j 处元
素进行交换
    */
    List<String> list = new ArrayList<>();
    Collections.addAll(list,"hello","java","world");

    Collections.swap(list,0,2);
    System.out.println(list);
}

@Test
public void test06() {
    /*

```

```
* public static <T extends Comparable<? super T>> void sort(List<T> list)根据元素的
自然顺序对指定 List 集合元素按升序排序
* public static <T> void sort(List<T> list,Comparator<? super T> c)根据指定的
Comparator 产生的顺序对 List 集合元素进行排序
*/
List<Man> list = new ArrayList<>();
list.add(new Man("张三",23));
list.add(new Man("李四",24));
list.add(new Man("王五",25));

Collections.sort(list);
System.out.println(list);

Collections.sort(list, new Comparator<Man>() {
    @Override
    public int compare(Man o1, Man o2) {
        return
Collator.getInstance(Locale.CHINA).compare(o1.getName(),o2.getName());
    }
});
System.out.println(list);
}

@Test
public void test05(){
/*
public static void shuffle(List<?> list) List 集合元素进行随机排序，类似洗牌，打乱顺序
*/
List<String> list = new ArrayList<>();
Collections.addAll(list,"hello","java","world");

Collections.shuffle(list);
System.out.println(list);
}

@Test
public void test04(){
/*
public static void reverse(List<?> list)反转指定列表List中元素的顺序。
*/
List<String> list = new ArrayList<>();
Collections.addAll(list,"hello","java","world");
System.out.println(list);

Collections.reverse(list);
System.out.println(list);
}

@Test
public void test03(){
/*
 * public static <T extends Object & Comparable<? super T>> T max(Collection<?
extends T> coll)

```

```

        *      <T extends Object & Comparable<? super T>>; 要求T必须继承Object，又实现
Comparable接口，或者T的父类实现Comparable接口
        *      在coll集合中找出最大的元素，集合中的对象必须是T或T的子类对象，而且支持自然排序
        * public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
        *      在coll集合中找出最大的元素，集合中的对象必须是T或T的子类对象，按照比较器comp找出最大者
        *
        */
List<Man> list = new ArrayList<>();
list.add(new Man("张三", 23));
list.add(new Man("李四", 24));
list.add(new Man("王五", 25));

/*Man max = Collections.max(list); //要求Man实现Comparable接口，或者父类实现
System.out.println(max); */

Man max = Collections.max(list, new Comparator<Man>() {
    @Override
    public int compare(Man o1, Man o2) {
        return o2.getAge() - o2.getAge();
    }
});
System.out.println(max);
}

@Test
public void test02() {
    /*
     * public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T
key)
     *      要求List集合的元素类型 实现了 Comparable接口，这个实现可以是T类型本身也可以T的父类实现
这个接口。
     *      说明List中的元素支持自然排序功能。
     *      在List集合中查找某个元素的下标，但是List的元素必须是T或T的子类对象，而且必须是可比较大小
的，即支持自然排序的。而且集合也事先必须是有序的，否则结果不确定。
     * public static <T> int binarySearch(List<? extends T> list, T key, Comparator<?
super T> c)
     *      说明List集合中元素的类型<=T， Comparator<? super T>说明要传入一个Comparator接口的实
现类对象，实现类泛型的指定要求>=T
     *      例如：List中存储的是Man（男）对象，T可以是Person类型，实现Comparator的时候可以是
Comparator<Person>
     *      例如：List中存储的是Man（男）对象，T可以是Man类型，实现Comparator的时候可以是
Comparator<Person>
     *      在List集合中查找某个元素的下标，但是List的元素必须是T或T的子类对象，而且集合也事先必须是
按照c比较器规则进行排序过的，否则结果不确定。
     *
     * 二分查找要求数组或List必须是“有大小顺序”。
     * 二分查找的思路： 和[mid]元素比较，如果相同，就找到了，不相同要看大小关系，决定去左边还是右边继
续查找。
    */
List<Man> list = new ArrayList<>();
list.add(new Man("张三", 23));
list.add(new Man("李四", 24));
list.add(new Man("王五", 25));
}

```

```

//      int index = Collections.binarySearch(list, new Man("王五", 25)); //要求实现
Comparable接口
//      System.out.println(index);

int index = Collections.binarySearch(list, new Man("王五", 25), new
Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getAge() - o2.getAge();
    }
});
System.out.println(index);
}

@Test
public void test01(){
    /*
    public static <T> boolean addAll(Collection<? super T> c, T... elements)将所有指定元素
添加到指定 collection 中。
    Collection的集合的元素类型必须>=T类型
    */
    Collection<Object> coll = new ArrayList<>();
    Collections.addAll(coll, "hello", "java");
    Collections.addAll(coll, 1, 2, 3, 4);

    Collection<String> coll2 = new ArrayList<>();
    Collections.addAll(coll2, "hello", "java");
//    Collections.addAll(coll2, 1, 2, 3, 4); //String和Integer之间没有父子类关系
}
}

```

13.9 二叉树了解

13.9.1 二叉树的基本概念

二叉树 (Binary tree) 是树形结构的一个重要类型。二叉树特点是每个结点最多只能有两棵子树，且有左右之分。许多实际问题抽象出来的数据结构往往是二叉树形式，二叉树的存储结构及其算法都较为简单，因此二叉树显得特别重要。



13.9.2 二叉树的遍历

- 前序遍历：中左右（根左右）
- 中序遍历：左中右（左根右）
- 后序遍历：左右中（左右根）



前序遍历：ABDHIECFG

中序遍历：HDIBEAFCG

后序遍历：HIDEFGCA

13.9.3 经典二叉树

1、满二叉树：除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树。第n层的结点数是2的n-1次方，总的结点个数是2的n次方-1



2、完全二叉树：叶结点只能出现在最底层的两层，且最底层叶结点均处于次底层叶结点的左侧。



3、平衡二叉树：平衡二叉树（Self-balancing binary search tree）又被称为AVL树（有别于AVL算法），且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树，但不要求非叶节点都有两个子结点。平衡二叉树的常用实现方法有红黑树、AVL、替罪羊树、Treap、伸展树等。例如红黑树的要求：

- 节点是红色或者黑色
- 根节点是黑色
- 每个叶子的节点都是黑色的空节点（NULL）
- 每个红色节点的两个子节点都是黑色的。
- 从任意节点到其每个叶子的所有路径都包含相同的黑色节点数量。

当我们插入或删除节点时，可能会破坏已有的红黑树，使得它不满足以上5个要求，那么此时就需要进行处理：

1、recolor：将某个节点变红或变黑

2、rotation：将红黑树某些结点分支进行旋转（左旋或右旋）

使得它继续满足以上以上的5个要求。

例如：插入了结点21之后，红黑树处理成：

13.9.4 二叉树及其结点的表示

普通二叉树：

```
public class BinaryTree<E>{
    private Node root; //二叉树的根结点
    private int total;//结点总个数

    private class Node{
        Node parent;
        Node left;
        E data;
        Node right;

        public Node(Node parent, Node left, E data, Node right) {
            this.parent = parent;
        }
    }
}
```

```

        this.left = left;
        this.data = data;
        this.right = right;
    }
}
}
}

```

TreeMap红黑树：

```

public class TreeMap<K,V> {
    private transient Entry<K,V> root;
    private transient int size = 0;

    static final class Entry<K,V> implements Map.Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left;
        Entry<K,V> right;
        Entry<K,V> parent;
        boolean color = BLACK;

        /**
         * Make a new cell with given key, value, and parent, and with
         * {@code null} child links, and BLACK color.
         */
        Entry(K key, V value, Entry<K,V> parent) {
            this.key = key;
            this.value = value;
            this.parent = parent;
        }
    }
}

```

13.10 哈希表

HashMap和Hashtable都是哈希表。

13.10.1 hashCode值

hash算法是一种可以从任何数据中提取出其“指纹”的数据摘要算法，它将任意大小的数据映射到一个固定大小的序列上，这个序列被称为hash code、数据摘要或者指纹。比较出名的hash算法有MD5、SHA。hash是具有唯一性且不可逆的，唯一性是指相同的“对象”产生的hash code永远是一样的。

 1563797150134

13.10.2 哈希表的物理结构

HashMap和Hashtable是散列表，其中维护了一个长度为 $2^{\text{幂次方}}$ 的Entry类型的数组table，数组的每一个元素被称为一个桶(bucket)，你添加的映射关系(key,value)最终都被封装为一个Map.Entry类型的对象，放到了某个table[index]桶中。使用数组的目的是查询和添加的效率高，可以根据索引直接定位到某个table[index]。

1、数组元素类型：Map.Entry

JDK1.7:

映射关系被封装为HashMap.Entry类型，而这个类型实现了Map.Entry接口。

观察HashMap.Entry类型是个结点类型，即table[index]下的映射关系可能串起来一个链表。因此我们把table[index]称为“桶bucket”。

```
public class HashMap<K,V>{
    transient Entry<K,V>[] table = (Entry<K,V>[][]) EMPTY_TABLE;
    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        Entry<K,V> next;
        int hash;
        //...省略
    }
    //...
}
```



JDK1.8:

映射关系被封装为HashMap.Node类型或HashMap.TreeNode类型，它俩都直接或间接的实现了Map.Entry接口。

存储到table数组的可能是Node结点对象，也可能是TreeNode结点对象，它们也是Map.Entry接口的实现类。即table[index]下的映射关系可能串起来一个链表或一棵红黑树。

```
public class HashMap<K,V>{
    transient Node<K,V>[] table;
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
        //...省略
    }
    static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
        TreeNode<K,V> parent; // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;
        boolean red;//是红结点还是黑结点
        //...省略
    }
    //...
}
```

```
public class LinkedHashMap<K,V>{
    static class Entry<K,V> extends HashMap.Node<K,V> {
        Entry<K,V> before, after;
        Entry(int hash, K key, V value, Node<K,V> next) {
            super(hash, key, value, next);
        }
    }
    //...
}
```



2、那么HashMap是如何决定某个映射关系存在哪个桶的呢？

因为hash值是一个整数，而数组的长度也是一个整数，有两种思路：

①hash 值 % table.length会得到一个[0,table.length-1]范围的值，正好是下标范围，但是用%运算效率没有&运算符高。

②hash 值 & (table.length-1)，任何数 & (table.length-1)的结果也一定在[0, table.length-1]范围。

```
hashCode值是    ?
table.length是10
table.length-1是9

?      ????????
9      00001001
&
00000000      [0]
00000001      [1]
00001000      [8]
00001001      [9]
一定[0]~[9]
```

```
hashCode值是    ?
table.length是16
table.length-1是15

?      ????????
15     00001111
&
00000000      [0]
00000001      [1]
00000010      [2]
00000011      [3]
...
00001111      [15]
范围是[0,15]，一定在[0,table.length-1]范围内
```

JDK1.7：

```
static int indexFor(int h, int length) {  
    // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";  
    return h & (length-1); //此处h就是hash  
}
```

JDK1.8:

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {  
    Node<K,V>[] tab; Node<K,V> p; int n, i;  
    if ((tab = table) == null || (n = tab.length) == 0)  
        n = (tab = resize()).length;  
    if ((p = tab[i = (n - 1) & hash]) == null) // i = (n - 1) & hash  
        tab[i] = newNode(hash, key, value, null);  
    //....省略大量代码  
}
```

3、数组的长度始终是2的n次幂

table数组的默认初始化长度：

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
```

如果你手动指定的table长度不是2的n次幂，会通过如下方法给你纠正为2的n次幂

JDK1.7:

HashMap处理容量方法：

```
private static int roundUpToPowerOf2(int number) {  
    // assert number >= 0 : "number must be non-negative";  
    return number >= MAXIMUM_CAPACITY  
        ? MAXIMUM_CAPACITY  
        : (number > 1) ? Integer.highestOneBit((number - 1) << 1) : 1;  
}
```

Integer包装类：

```
public static int highestOneBit(int i) {  
    // HD, Figure 3-1  
    i |= (i >> 1);  
    i |= (i >> 2);  
    i |= (i >> 4);  
    i |= (i >> 8);  
    i |= (i >> 16);  
    return i - (i >>> 1);  
}
```

JDK1.8:

```

static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}

```

如果数组不够了，扩容了怎么办？扩容了还是2的n次幂，因为每次数组扩容为原来的2倍

JDK1.7：

```

void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length); //扩容为原来的2倍
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }
    createEntry(hash, key, value, bucketIndex);
}

```

JDK1.8：

```

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length; //oldCap原来的容量
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        } //newCap = oldCap << 1 新容量=旧容量扩容为原来的2倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
        }
        //.....此处省略其他代码
    }
}

```

那么为什么要保持table数组一直是2的n次幂呢？

因为如果数组的长度为2的n次幂，那么table.length-1的二进制就是一个高位全是0，低位全是1的数字，这样才能保证每一个下标位置都有机会被用到。

hashCode值是 ?
table.length是10
table.length-1是9

? ????????
9 00001001
& _____
00000000 [0]
00000001 [1]
00001000 [8]
00001001 [9]
一定[0]~[9]

hashCode值是 ?
table.length是16
table.length-1是15

? ????????
15 00001111
& _____
00000000 [0]
00000001 [1]
00000010 [2]
00000011 [3]
...
00001111 [15]
范围是[0,15], 一定在[0,table.length-1]范围内

4、hash是hashCode的再运算

不管是JDK1.7还是JDK1.8中，都不是直接用key的hashCode值直接与table.length-1计算求下标的，而是先对key的hashCode值进行了一个运算，JDK1.7和JDK1.8关于hash()的实现代码不一样，但是不管怎么样都是为了提高hash code值与(table.length-1)的按位与完的结果，尽量的均匀分布。

JDK1.7:

```
final int hash(Object k) {  
    int h = hashSeed;  
    if (0 != h && k instanceof String) {  
        return sun.misc.Hashing.stringHash32((String) k);  
    }  
  
    h ^= k.hashCode();  
    h ^= (h >>> 20) ^ (h >>> 12);  
    return h ^ (h >>> 7) ^ (h >>> 4);  
}
```

JDK1.8:

```
static final int hash(Object key) {  
    int h;  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

虽然算法不同，但是思路都是将hashCode值的高位二进制与低位二进制值进行了异或，然高位二进制参与到index的计算中。

为什么要hashCode值的二进制的高位参与到index计算呢？

因为一个HashMap的table数组一般不会特别大，至少在不断扩容之前，那么table.length-1的大部分高位都是0，直接用hashCode和table.length-1进行&运算的话，就会导致总是只有最低的几位是有效的，那么就算你的hashCode()实现的再好也难以避免发生碰撞，这时让高位参与进来的意义就体现出来了。它对hashcode的低位添加了随机性并且混合了高位的部分特征，显著减少了碰撞冲突的发生。

5、解决[index]冲突问题

虽然从设计hashCode()到上面HashMap的hash()函数，都尽量减少冲突，但是仍然存在两个不同的对象返回的hashCode值相同，或者hashCode值就算不同，通过hash()函数计算后，得到的index也会存在大量的相同，因此key分布完全均匀的情况是不存在的。那么发生碰撞冲突时怎么办？

JDK1.8之间使用：数组+链表的结构。

JDK1.8之后使用：数组+链表/红黑树的结构。

即hash相同或hash&(table.length-1)的值相同，那么就存入同一个“桶”table[index]中，使用链表或红黑树连接起来。

1563802656661

1563802665708

6、为什么JDK1.8会出现红黑树和链表共存呢？

因为当冲突比较严重时，table[index]下面的链表就会很长，那么会导致查找效率大大降低，而如果此时选用二叉树可以大大提高查询效率。

但是二叉树的结构又过于复杂，如果结点个数比较少的时候，那么选择链表反而更简单。

所以会出现红黑树和链表共存。

7、什么时候树化？什么时候反树化？

```
static final int TREEIFY_THRESHOLD = 8;//树化阈值  
static final int UNTREEIFY_THRESHOLD = 6;//反树化阈值  
static final int MIN_TREEIFY_CAPACITY = 64;//最小树化容量
```

- 当某table[index]下的链表的结点个数达到8，并且table.length>=64，那么如果新Entry对象还添加到该table[index]中，那么就会将table[index]的链表进行树化。
- 当某table[index]下的红黑树结点个数少于6个，此时，
 - 当继续删除table[index]下的树结点，最后这个根结点的左右结点有null，或根结点的左结点的左结点为null，会反树化
 - 当重新添加新的映射关系到map中，导致了map重新扩容了，这个时候如果table[index]下面还是小于等于6的个数，那么会反树化

```
package com.atguigu.map;

public class MyKey{
    int num;

    public MyKey(int num) {
        super();
        this.num = num;
    }

    @Override
    public int hashCode() {
        if(num<=20){
            return 1;
        }else{
            final int prime = 31;
            int result = 1;
            result = prime * result + num;
            return result;
        }
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        MyKey other = (MyKey) obj;
        if (num != other.num)
            return false;
        return true;
    }
}
```

```
package com.atguigu.map;

import org.junit.Test;

import java.util.HashMap;

public class TestHashMapMyKey {
    @Test
    public void test1(){
        //这里为了演示的效果，我们造一个特殊的类，这个类的hashCode () 方法返回固定值1
        //因为这样就可以造成冲突问题，使得它们都存到table[1]中
        HashMap<MyKey, String> map = new HashMap<>();
        for (int i = 1; i <= 11; i++) {
```

```

        map.put(new MyKey(i), "value"+i);//树化演示
    }
}

@Test
public void test2(){
    HashMap<MyKey, String> map = new HashMap<>();
    for (int i = 1; i <= 11; i++) {
        map.put(new MyKey(i), "value"+i);
    }
    for (int i = 1; i <=11; i++) {
        map.remove(new MyKey(i));//反树化演示
    }
}
@Test
public void test3(){
    HashMap<MyKey, String> map = new HashMap<>();
    for (int i = 1; i <= 11; i++) {
        map.put(new MyKey(i), "value"+i);
    }

    for (int i = 1; i <=5; i++) {
        map.remove(new MyKey(i));
    }//table[1]下剩余6个结点

    for (int i = 21; i <= 100; i++) {
        map.put(new MyKey(i), "value"+i); //添加到扩容时，反树化
    }
}
}
}

```

13.10.3 JDK1.7的put方法源码分析

(1) 几个关键的常量和变量值的作用:

初始化容量:

```
int DEFAULT_INITIAL_CAPACITY = 1 << 4;//16 目的是体现2的n次方
```

①默认负载因子

```
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

②阈值: 扩容的临界值

```
int threshold;
```

```
threshold = table.length * loadFactor;
```

③负载因子

```
final float loadFactor;
```

负载因子的值大小有什么关系？

如果太大，threshold就会很大，那么如果冲突比较严重的话，就会导致table[index]下面的结点个数很多，影响效率。

如果太小，threshold就会很小，那么数组扩容的频率就会提高，数组的使用率也会降低，那么会造成空间的浪费。

```
public HashMap() {
    //DEFAULT_INITIAL_CAPACITY: 默认初始容量16
    //DEFAULT_LOAD_FACTOR: 默认加载因子0.75
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
}

public HashMap(int initialCapacity, float loadFactor) {
    //校验initialCapacity合法性
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
            initialCapacity);
    //校验initialCapacity合法性
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    //校验loadFactor合法性
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
            loadFactor);
    //加载因子，初始化为0.75
    this.loadFactor = loadFactor;
    // threshold 初始为初始容量
    threshold = initialCapacity;
    init();
}
```

```
public V put(K key, V value) {
    //如果table数组是空的，那么先创建数组
    if (table == EMPTY_TABLE) {
        //threshold一开始是初始容量的值
        inflateTable(threshold);
    }
    //如果key是null，单独处理，存储到table[0]中，如果有另一个key为null，value覆盖
    if (key == null)
        return putForNullKey(value);

    //对key的hashCode进行干扰，算出一个hash值
    /*
    hashCode值     xxxxxxxxxx
    table.length-1  000001111

    hashCode值 xxxxxxxxxx 无符号右移几位和原来的hashCode值做^运算，使得hashCode高位二进制值参与计算，也
    发挥作用，降低index冲突的概率。
    */
}
```

```

int hash = hash(key);

//计算新的映射关系应该存到table[i]位置,
//i = hash & table.length-1, 可以保证i在[0,table.length-1]范围内
int i = indexFor(hash, table.length);

//检查table[i]下面有没有key与我新的映射关系的key重复, 如果重复替换value
for (Entry<K,V> e = table[i]; e != null; e = e.next) {
    Object k;
    if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
        v.oldvalue = e.value;
        e.value = value;
        e.recordAccess(this);
        return oldvalue;
    }
}

modCount++;
//添加新的映射关系
addEntry(hash, key, value, i);
return null;
}

private void inflateTable(int toSize) {
    // Find a power of 2 >= toSize
    int capacity = roundUpToPowerOf2(tosize); //容量是等于tosize值的最接近的2的n次方
    //计算阈值 = 容量 * 加载因子
    threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
    //创建Entry[]数组, 长度为capacity
    table = new Entry[capacity];
    initHashSeedAsNeeded(capacity);
}

//如果key是null, 直接存入[0]的位置
private V putForKey(V value) {
    //判断是否有重复的key, 如果有重复的, 就替换value
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
        if (e.key == null) {
            v.oldvalue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldvalue;
        }
    }
    modCount++;
    //把新的映射关系存入[0]的位置, 而且key的hash值用0表示
    addEntry(0, null, value, 0);
    return null;
}

void addEntry(int hash, K key, V value, int bucketIndex) {
    //判断是否需要扩容
    //扩容: (1) size达到阈值 (2) table[i]正好非空
    if ((size >= threshold) && (null != table[bucketIndex])) {
        //table扩容为原来的2倍, 并且扩容后, 会重新调整所有映射关系的存储位置
        resize(2 * table.length);
    }
}

```

```

    //新的映射关系的hash和index也会重新计算
    hash = (null != key) ? hash(key) : 0;
    bucketIndex = indexFor(hash, table.length);
}

//存入table中
createEntry(hash, key, value, bucketIndex);
}

void createEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
    //原来table[i]下面的映射关系作为新的映射关系next
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++; //个数增加
}

```

1、put(key,value)

- (1) 当第一次添加映射关系时，数组初始化为一个长度为16的**HashMap\$Entry**的数组，这个HashMap\$Entry类型是实现了java.util.Map.Entry接口
- (2) 特殊考虑：如果key为null，index直接是[0],hash也是0
- (3) 如果key不为null，在计算index之前，会对key的hashCode()值，做一个hash(key)再次哈希的运算，这样可以使得Entry对象更加散列的存储到table中
- (4) 计算index = table.length-1 & hash;
- (5) 如果table[index]下面，已经有映射关系的key与我要添加的新的映射关系的key相同了，会用新的value替换旧的value。
- (6) 如果没有相同的，会把新的映射关系添加到链表的头，原来table[index]下面的Entry对象连接到新的映射关系的next中。
- (7) 添加之前先判断if(size >= threshold && table[index] != null)如果该条件为true，会扩容

```

if(size >= threshold && table[index] != null){

    ①会扩容

    ②会重新计算key的hash

    ③会重新计算index

}

```

(8) size++

1563804039202

2、get(key)

- (1) 计算key的hash值，用这个方法hash(key)
- (2) 找index = table.length-1 & hash;
- (3) 如果table[index]不为空，那么就挨个比较哪个Entry的key与它相同，就返回它的value

3、remove(key)

- (1) 计算key的hash值，用这个方法hash(key)
- (2) 找index = table.length-1 & hash;
- (3) 如果table[index]不为空，那么就挨个比较哪个Entry的key与它相同，就删除它，把它前面的Entry的next的值修改为被删除Entry的next

13.10.4 JDK1.8的put方法源码分析

几个常量和变量：

- (1) DEFAULT_INITIAL_CAPACITY: 默认的初始容量 16
- (2) MAXIMUM_CAPACITY: 最大容量 1 << 30
- (3) DEFAULT_LOAD_FACTOR: 默认加载因子 0.75
- (4) TREEIFY_THRESHOLD: 默认树化阈值8，当链表的长度达到这个值后，要考虑树化
- (5) UNTREEIFY_THRESHOLD: 默认反树化阈值6，当树中的结点的个数达到这个阈值后，要考虑变为链表
- (6) MIN_TREEIFY_CAPACITY: 最小树化容量64
当单个的链表的结点个数达到8，并且table的长度达到64，才会树化。
当单个的链表的结点个数达到8，但是table的长度未达到64，会先扩容
- (7) Node<K,V>[] table: 数组
- (8) size: 记录有效映射关系的对数，也是Entry对象的个数
- (9) int threshold: 阈值，当size达到阈值时，考虑扩容
- (10) double loadFactor: 加载因子，影响扩容的频率

```
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    // all other fields defaulted, 其他字段都是默认值
}
```

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
//目的：干扰hashCode值
static final int hash(Object key) {
    int h;
    //如果key是null，hash是0
    //如果key非null，用key的hashCode值 与 key的hashCode值高16位进行异或
    //即就是用key的hashCode值高16位与低16位进行了异或的干扰运算

    /*
     * index = hash & table.length-1
     * 如果用key的原始的hashCode值 与 table.length-1 进行按位与，那么基本上高16位没机会用上。
     * 这样就会增加冲突的概率，为了降低冲突的概率，把高16位加入到hash信息中。
     */
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {
    Node<K,V>[] tab; //数组
    Node<K,V> p; //一个结点
```

```

int n, i;//n是数组的长度    i是下标

//tab和table等价
//如果table是空的
if ((tab = table) == null || (n = tab.length) == 0){
    n = (tab = resize()).length;
    /*
        tab = resize();
        n = tab.length; */
        /*
        如果table是空的, resize()完成了①创建了一个长度为16的数组②threshold = 12
        n = 16
        */
}
//i = (n - 1) & hash , 下标 = 数组长度-1 & hash
//p = tab[i] 第1个结点
//if(p==null) 条件满足的话说明 table[i]还没有元素
if ((p = tab[i = (n - 1) & hash]) == null){
    //把新的映射关系直接放入table[i]
    tab[i] = newNode(hash, key, value, null);
    //newNode () 方法就创建了一个Node类型的新结点, 新结点的next是null
} else {
    Node<K,V> e;
    K k;
    //p是table[i]中第一个结点
    //if(table[i]的第一个结点与新的映射关系的key重复)
    if (p.hash == hash &&
        ((k = p.key) == key || (key != null && key.equals(k)))) {
        e = p;//用e记录这个table[i]的第一个结点
        } else if (p instanceof TreeNode){//如果table[i]第一个结点是一个树结点
            //单独处理树结点
            //如果树结点中, 有key重复的, 就返回那个重复的结点用e接收, 即e!=null
            //如果树结点中, 没有key重复的, 就把新结点放到树中, 并且返回null, 即e=null
            e = ((TreeNode<K,V>)p).putTreeval(this, tab, hash, key,
value);
        } else {
            //table[i]的第一个结点不是树结点, 也与新的映射关系的key不重复
            //binCount记录了table[i]下面的结点的个数
            for (int binCount = 0; ; ++binCount) {
                //如果p的下一个结点是空的, 说明当前的p是最后一个结点
                if ((e = p.next) == null) {
                    //把新的结点连接到table[i]的最后
                    p.next = newNode(hash, key, value, null);

                    //如果binCount>=8-1, 达到7个时
                    if (binCount >= TREEIFY_THRESHOLD - 1){ // -1 for 1st
                        //要么扩容, 要么树化
                        treeifyBin(tab, hash);
                    }
                }
                break;
            }
        }
    }
}
//如果key重复了, 就跳出for循环, 此时e结点记录的就是那个key重
复的结点

```

```

        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
            break;
        }
        p = e;//下一次循环, e=p.next, 就类似于e=e.next, 往链表下移动
    }
}

//如果这个e不是null, 说明有key重复, 就考虑替换原来的value
if (e != null) { // existing mapping for key
    v oldvalue = e.value;
    if (!onlyIfAbsent || oldvalue == null){
        e.value = value;
    }
    afterNodeAccess(e);//什么也没干
    return oldvalue;
}
++modCount;

//元素个数增加
//size达到阈值
if (++size > threshold){
    resize();//一旦扩容, 重新调整所有映射关系的位置
}
afterNodeInsertion(evict);//什么也没干
return null;
}

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;//oldTab原来的table
    //oldCap: 原来数组的长度
    int oldCap = (oldTab == null) ? 0 : oldTab.length;

    //oldThr: 原来的阈值
    int oldThr = threshold;//最开始threshold是0

    //newCap, 新容量
    //newThr: 新阈值
    int newCap, newThr = 0;
    if (oldCap > 0) {//说明原来不是空数组
        if (oldCap >= MAXIMUM_CAPACITY) {//是否达到数组最大限制
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                  oldCap >= DEFAULT_INITIAL_CAPACITY){
            //newCap = 旧的容量*2 , 新容量<最大数组容量限制
            //新容量: 32,64, ...
            //oldCap >= 初始容量16
            //新阈值重新算 = 24, 48 ....
            newThr = oldThr << 1; // double threshold
        }
    }else if (oldThr > 0){ // initial capacity was placed in threshold
        newCap = oldThr;
    }else { // zero initial threshold signifies using defaults

```

```

newCap = DEFAULT_INITIAL_CAPACITY; //新容量是默认初始化容量16
        //新阈值= 默认的加载因子 * 默认的初始化容量 = 0.75*16 = 12
newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
}
if (newThr == 0) {
    float ft = (float)newCap * loadFactor;
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
               (int)ft : Integer.MAX_VALUE);
}
threshold = newThr; //阈值赋值为新阈值12, 24...。

//创建了一个新数组，长度为newCap, 16, 32, 64...。
@SuppressWarnings({"rawtypes", "unchecked"})
Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
table = newTab;

if (oldTab != null) { //原来不是空数组
    //把原来的table中映射关系，倒腾到新的table中
    for (int j = 0; j < oldCap; ++j) {
        Node<K,V> e;
        if ((e = oldTab[j]) != null) { //e是table下面的结点
            oldTab[j] = null; //把旧的table[j]位置清空
            if (e.next == null) //如果是最后一个结点
                newTab[e.hash & (newCap - 1)] = e; //重新计算e在新table中的存储位置，然后
放入
        else if (e instanceof TreeNode) //如果e是树结点
            //把原来的树拆解，放到新的table
            ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
        else { // preserve order
            Node<K,V> loHead = null, loTail = null;
            Node<K,V> hiHead = null, hiTail = null;
            Node<K,V> next;
            /*
             把原来table[i]下面的整个链表，重新挪到了新的table中
             */
            do {
                next = e.next;
                if ((e.hash & oldCap) == 0) {
                    if (loTail == null)
                        loHead = e;
                    else
                        loTail.next = e;
                    loTail = e;
                }
                else {
                    if (hiTail == null)
                        hiHead = e;
                    else
                        hiTail.next = e;
                    hiTail = e;
                }
            } while ((e = next) != null);
        }
    }
}

```

```

        if (loTail != null) {
            loTail.next = null;
            newTab[j] = loHead;
        }
        if (hiTail != null) {
            hiTail.next = null;
            newTab[j + oldCap] = hiHead;
        }
    }
}
}

return newTab;
}

Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {
    //创建一个新结点
    return new Node<>(hash, key, value, next);
}

final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index;
    Node<K,V> e;
    //MIN_TREEIFY_CAPACITY: 最小树化容量64
    //如果table是空的，或者 table的长度没有达到64
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize(); //先扩容
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        //用e记录table[index]的结点的地址
        TreeNode<K,V> hd = null, t1 = null;
        /*
         * do...while, 把table[index]链表的Node结点变为TreeNode类型的结点
         */
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (t1 == null)
                hd = p; //hd记录根结点
            else {
                p.prev = t1;
                t1.next = p;
            }
            t1 = p;
        } while ((e = e.next) != null);

        //如果table[index]下面不是空
        if ((tab[index] = hd) != null)
            hd.treeify(tab); //将table[index]下面的链表进行树化
    }
}

```

1、添加过程

- A. 先计算key的hash值，如果key是null，hash值就是0，如果为null，使用($h = key.hashCode() \wedge (h >>> 16)$)得到hash值；
- B. 如果table是空的，先初始化table数组；
- C. 通过hash值计算存储的索引位置 $index = hash \& (table.length - 1)$
- D. 如果 $table[index] == null$ ，那么直接创建一个Node结点存储到 $table[index]$ 中即可
- E. 如果 $table[index] != null$
 - a) 判断 $table[index]$ 的根结点的key是否与新的key“相同” (hash值相同并且(满足key的地址相同或key的equals返回true))，如果是那么用e记录这个根结点
 - b) 如果 $table[index]$ 的根结点的key与新的key“不相同”，而且 $table[index]$ 是一个TreeNode结点，说明 $table[index]$ 下是一棵红黑树，如果该树的某个结点的key与新的key“相同” (hash值相同并且(满足key的地址相同或key的equals返回true))，那么用e记录这个相同的结点，否则将(key,value)封装为一个TreeNode结点，连接到红黑树中
 - c) 如果 $table[index]$ 的根结点的key与新的key“不相同”，并且 $table[index]$ 不是一个TreeNode结点，说明 $table[index]$ 下是一个链表，如果该链表中的某个结点的key与新的key“相同”，那么用e记录这个相同的结点，否则将新的映射关系封装为一个Node结点直接链接到链表尾部，并且判断 $table[index]$ 下结点个数达到**TREEIFY_THRESHOLD(8)**个，如果 $table[index]$ 下结点个数已经达到，那么再判断 $table.length$ 是否达到**MIN_TREEIFY_CAPACITY(64)**，如果没达到，那么先扩容，扩容会导致所有元素重新计算index，并调整位置，如果 $table[index]$ 下结点个数已经达到**TREEIFY_THRESHOLD(8)**个并 $table.length$ 也已经达到**MIN_TREEIFY_CAPACITY(64)**，那么会将该链表转成一棵自平衡的红黑树。
- F. 如果在 $table[index]$ 下找到了新的key“相同”的结点，即e不为空，那么用新的value替换原来的value，并返回旧的value，结束put方法
- G. 如果新增结点而不是替换，那么size++，并且还要重新判断size是否达到threshold阈值，如果达到，还要扩容。

```

if(size > threshold ){
    ①会扩容

    ②会重新计算key的hash

    ③会重新计算index

}

```

2、remove(key)

- (1) 计算key的hash值，用这个方法hash(key)
- (2) 找 $index = table.length - 1 \& hash$;
- (3) 如果 $table[index]$ 不为空，那么挨个比较哪个Entry的key与它相同，就删除它，把它前面的Entry的next的值修改为被删除Entry的next
- (4) 如果 $table[index]$ 下面原来是红黑树，结点删除后，个数小于等于6，会把红黑树变为链表

13.10.5 关于映射关系的key是否可以修改？

映射关系存储到HashMap中会存储key的hash值，这样就不用在每次查找时重新计算每一个Entry或Node (TreeNode) 的hash值了，因此如果已经put到Map中的映射关系，再修改key的属性，而这个属性又参与hashcode值的计算，那么会导致匹配不上。

这个规则也同样适用于LinkedHashMap、HashSet、LinkedHashSet、Hashtable等所有散列存储结构的集合。

JDK1.7：

```
public class HashMap<K,V>{
    transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE;
    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        Entry<K,V> next;
        int hash; //记录Entry映射关系的key的hash(key.hashCode())值
        //...省略
    }
    //...
}
```

JDK1.8：

```
public class HashMap<K,V>{
    transient Node<K,V>[] table;
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;//记录Node映射关系的key的hash(key.hashCode())值
        final K key;
        V value;
        Node<K,V> next;
        //...省略
    }
    //...
}
```

示例代码：

```
package com.atguigu.map;

public class ID{
    private int id;

    public ID(int id) {
        super();
        this.id = id;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        return result;
    }
}
```

```
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    ID other = (ID) obj;
    if (id != other.id)
        return false;
    return true;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

}
```

```
package com.atguigu.map;

import java.util.HashMap;

public class TestHashMapID{
    public static void main(String[] args) {
        HashMap<ID, String> map = new HashMap<>();
        ID i1 = new ID(1);
        ID i2 = new ID(2);
        ID i3 = new ID(3);

        map.put(i1, "haha");
        map.put(i2, "hehe");
        map.put(i3, "xixi");

        System.out.println(map.get(i1)); //haha
        i1.setId(10);
        System.out.println(map.get(i1)); //null
    }
}
```

所以实际开发中，经常选用String, Integer等作为key，因为它们都是不可变对象。

第14章 File类与IO流

14.1 java.io.File类

14.1.1 概述

File类是java.io包下代表与平台无关的文件和目录，也就是说如果希望在程序中操作文件和目录都可以通过File类来完成，File类能新建、删除、重命名文件和目录。

在API中File的解释是文件和目录路径名的抽象表示形式，即File类是文件或目录的路径，而不是文件本身，因此File类不能直接访问文件内容本身，如果需要访问文件内容本身，则需要使用输入/输出流。

File类代表磁盘或网络中某个文件或目录的路径名称，如：/atguigu/javase/io/佟刚.jpg

但不能直接通过File对象读取和写入数据，如果要操作数据，需要IO流。File对象好比是到水库的“路线地址”，要“存取”里面的水到你“家里”，需要“管道”。



14.1.2 构造方法

- `public File(String pathname)` : 通过将给定的路径名字符串转换为抽象路径名来创建新的 File实例。
- `public File(String parent, String child)` : 从父路径名字符串和子路径名字符串创建新的 File实例。
- `public File(File parent, String child)` : 从父抽象路径名和子路径名字符串创建新的 File实例。
- 构造举例，代码如下：

```
package com.atguigu.file;

import java.io.File;

public class FileObjectTest {
    public static void main(String[] args) {
        // 文件路径名
        String pathname = "D:\\aaa.txt";
        File file1 = new File(pathname);

        // 文件路径名
        String pathname2 = "D:\\aaa\\bbb.txt";
        File file2 = new File(pathname2);

        // 通过父路径和子路径字符串
        String parent = "d:\\aaa";
        String child = "bbb.txt";
        File file3 = new File(parent, child);

        // 通过父级File对象和子路径字符串
        File parentDir = new File("d:\\aaa");
        String childFile = "bbb.txt";
        File file4 = new File(parentDir, childFile);
    }
}
```

小贴士：

1. 一个File对象代表硬盘或网络中可能存在的一一个文件或者目录。
2. 无论该路径下是否存在文件或者目录，都不影响File对象的创建。
3. 如果File对象代表的文件或目录存在，则File对象实例初始化时，就会用硬盘中对应文件或目录的属性信息（例如，时间、类型等）为File对象的属性赋值，否则除了路径和名称，File对象的其他属性将会保留默认值。

14.1.3 常用方法

1、获取文件和目录基本信息的方法

- `public string getName()`：返回由此File表示的文件或目录的名称。
- `public long length()`：返回由此File表示的文件的长度。
- `public string getPath()`：将此File转换为路径名字符串。
- `public long lastModified()`：返回File对象对应的文件或目录的最后修改时间（毫秒值）

方法演示，代码如下：

```
package com.atguigu.file;

import java.io.File;
import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneId;
```

```
public class FileInfoMethod {
    public static void main(String[] args) {
        File f = new File("d:/aaa/bbb.txt");
        System.out.println("文件构造路径:"+f.getPath());
        System.out.println("文件名称:"+f.getName());
        System.out.println("文件长度:"+f.length()+"字节");
        System.out.println("文件最后修改时间: " +
LocalDateTime.ofInstant(Instant.ofEpochMilli(f.lastModified()),ZoneId.of("Asia/Shanghai"))
));

        File f2 = new File("d:/aaa");
        System.out.println("目录构造路径:"+f2.getPath());
        System.out.println("目录名称:"+f2.getName());
        System.out.println("目录长度:"+f2.length()+"字节");
        System.out.println("文件最后修改时间: " +
LocalDateTime.ofInstant(Instant.ofEpochMilli(f.lastModified()),ZoneId.of("Asia/Shanghai"))
));
    }
}
```

输出结果：

```
文件构造路径:d:\aaa\bbb.java
文件名称:bbb.java
文件长度:636字节
文件最后修改时间: 2019-07-23T22:01:32.065
```

```
目录构造路径:d:\aaa
目录名称:aaa
目录长度:4096字节
文件最后修改时间: 2019-07-23T22:01:32.065
```

API中说明：length()，表示文件的长度。如果File对象表示目录，则返回值未指定。

2、各种路径问题

- `public String getPath()`：将此File转换为路径名字符串。
- `public String getAbsolutePath()`：返回此File的绝对路径名字符串。
- `String getCanonicalPath()`：返回此File对象所对应的规范路径名。

File类可以使用文件路径字符串来创建File实例，该文件路径字符串既可以是绝对路径，也可以是相对路径。

默认情况下，系统总是依据用户的工作路径来解释相对路径，这个路径由系统属性“user.dir”指定，通常也就是运行Java虚拟机时所作的路径。

- **绝对路径**：从盘符开始的路径，这是一个完整的路径。
- **相对路径**：相对于**项目目录**的路径，这是一个便捷的路径，开发中经常使用。
- **规范路径**：所谓规范路径名，即对路径中的“..”等进行解析后的路径名

```
package com.atguigu.file;

import org.junit.Test;
```

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

public class FilePath {
    @Test
    public void test1() throws IOException{
        File f1 = new File("d:\\atguigu\\javase\\HelloIO.java"); //绝对路径
        System.out.println("文件/目录的名称: " + f1.getName());
        System.out.println("文件/目录的构造路径名: " + f1.getPath());
        System.out.println("文件/目录的绝对路径名: " + f1.getAbsolutePath());
        System.out.println("文件/目录的规范路径名: " + f1.getCanonicalPath());
        System.out.println("文件/目录的父目录名: " + f1.getParent());
    }
    @Test
    public void test02()throws IOException{
        File f2 = new File("/HelloIO.java");//绝对路径, 从根路径开始
        System.out.println("文件/目录的名称: " + f2.getName());
        System.out.println("文件/目录的构造路径名: " + f2.getPath());
        System.out.println("文件/目录的绝对路径名: " + f2.getAbsolutePath());
        System.out.println("文件/目录的规范路径名: " + f2.getCanonicalPath());
        System.out.println("文件/目录的父目录名: " + f2.getParent());
    }

    @Test
    public void test03() throws IOException {
        File f3 = new File("HelloIO.java");//相对路径
        System.out.println("user.dir =" + System.getProperty("user.dir"));
        System.out.println("文件/目录的名称: " + f3.getName());
        System.out.println("文件/目录的构造路径名: " + f3.getPath());
        System.out.println("文件/目录的绝对路径名: " + f3.getAbsolutePath());
        System.out.println("文件/目录的规范路径名: " + f3.getCanonicalPath());
        System.out.println("文件/目录的父目录名: " + f3.getParent());
    }
    @Test
    public void test04() throws IOException{
        File f4 = new File("../HelloIO.java");//相对路径
        System.out.println("user.dir =" + System.getProperty("user.dir"));
        System.out.println("文件/目录的名称: " + f4.getName());
        System.out.println("文件/目录的构造路径名: " + f4.getPath());
        System.out.println("文件/目录的绝对路径名: " + f4.getAbsolutePath());
        System.out.println("文件/目录的规范路径名: " + f4.getCanonicalPath());
        System.out.println("文件/目录的父目录名: " + f4.getParent());
    }

    public static void main(String[] args) throws IOException {
        File f5 = new File("HelloIO.java");//相对路径
        System.out.println("user.dir =" + System.getProperty("user.dir"));
        System.out.println("文件/目录的名称: " + f5.getName());
        System.out.println("文件/目录的构造路径名: " + f5.getPath());
        System.out.println("文件/目录的绝对路径名: " + f5.getAbsolutePath());
        System.out.println("文件/目录的规范路径名: " + f5.getCanonicalPath());
        System.out.println("文件/目录的父目录名: " + f5.getParent());
    }
}
```

```
    }  
}
```

- window的路径分隔符使用"\\", 而Java程序中的 "\\" 表示转义字符，所以在Windows中表示路径，需要用\"。或者直接使用"/"也可以，Java程序支持将"/"当成平台无关的路径分隔符。或者直接使用File.separator常量值表示。
- 把构造File对象指定的文件或目录的路径名，称为构造路径，它可以是绝对路径，也可以是相对路径
- 当构造路径是绝对路径时，那么getPath和getAbsolutePath结果一样
- 当构造路径是相对路径时，那么getAbsolutePath的路径 = user.dir的路径 + 构造路径
- 当路径中不包含".."和"/开头"等形式的路径，那么规范路径和绝对路径一样，否则会将..等进行解析。路径中如果出现".."表示上一级目录，路径名如果以"/"开头，表示从"根目录"下开始导航。

3、判断功能的方法

- `public boolean exists()`：此File表示的文件或目录是否实际存在。
- `public boolean isDirectory()`：此File表示的是否为目录。
- `public boolean isFile()`：此File表示的是否为文件。

方法演示，代码如下：

```
package com.atguigu.file;  
  
import java.io.File;  
  
public class FileIs {  
    public static void main(String[] args) {  
        File f = new File("d:\\aaa\\bbb.java");  
        File f2 = new File("d:\\aaa");  
        // 判断是否存在  
        System.out.println("d:\\aaa\\bbb.java 是否存在:"+f.exists());  
        System.out.println("d:\\aaa 是否存在:"+f2.exists());  
        // 判断是文件还是目录  
        System.out.println("d:\\aaa 文件?:"+f2.isFile());  
        System.out.println("d:\\aaa 目录?:"+f2.isDirectory());  
    }  
}
```

输出结果：

```
d:\\aaa\\bbb.java 是否存在:true  
d:\\aaa 是否存在:true  
d:\\aaa 文件?:false  
d:\\aaa 目录?:true
```

如果文件或目录不存在，那么exists()、isFile()和isDirectory()都是返回true

4、创建删除功能的方法

- `public boolean createNewFile()`：当且仅当具有该名称的文件尚不存在时，创建一个新的空文件。
- `public boolean delete()`：删除由此File表示的文件或目录。只能删除空目录。
- `public boolean mkdir()`：创建由此File表示的目录。

- `public boolean mkdirs()` : 创建由此File表示的目录，包括任何必需但不存在的父目录。

方法演示，代码如下：

```
package com.atguigu.file;

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

public class FileCreateDelete {
    public static void main(String[] args) throws IOException {
        // 文件的创建
        File f = new File("aaa.txt");
        System.out.println("aaa.txt是否存在:" + f.exists());
        System.out.println("aaa.txt是否创建:" + f.createNewFile());
        System.out.println("aaa.txt是否存在:" + f.exists());

        // 目录的创建
        File f2= new File("newDir");
        System.out.println("newDir是否存在:" + f2.exists());
        System.out.println("newDir是否创建:" + f2.mkdir());
        System.out.println("newDir是否存在:" + f2.exists());

        // 创建一级目录
        File f3= new File("newDira\\newDirb");
        System.out.println("newDira\\newDirb创建: " + f3.mkdir());
        File f4= new File("newDir\\newDirb");
        System.out.println("newDir\\newDirb创建: " + f4.mkdir());
        // 创建多级目录
        File f5= new File("newDira\\newDirb");
        System.out.println("newDira\\newDirb创建: " + f5.mkdirs());

        // 文件的删除
        System.out.println("aaa.txt删除: " + f.delete());

        // 目录的删除
        System.out.println("newDir删除: " + f2.delete());
        System.out.println("newDir\\newDirb删除: " + f4.delete());
    }
}
```

运行结果：

```
aaa.txt是否存在:false
aaa.txt是否创建:true
aaa.txt是否存在:true
newDir是否存在:false
newDir是否创建:true
newDir是否存在:true
newDira\newDirb创建: false
newDir\newDirb创建: true
newDira\newDirb创建: true
aaa.txt删除: true
```

```
newDir删除: false  
newDir\newDirb删除: true
```

API中说明：delete方法，如果此File表示目录，则目录必须为空才能删除。

5、列出目录的下一级

- `public String[] list()`：返回一个String数组，表示该File目录中的所有子文件或目录。
- `public File[] listFiles()`：返回一个File数组，表示该File目录中的所有的子文件或目录。
- `public File[] listFiles(FileFilter filter)`：返回所有满足指定过滤器的文件和目录。如果给定filter为null，则接受所有路径名。否则，当且仅当在路径名上调用过滤器的`FileFilter.accept(File pathname)`方法返回true时，该路径名才满足过滤器。如果当前File对象不表示一个目录，或者发生I/O错误，则返回null。
- `public String[] list(FilenameFilter filter)`：返回返回所有满足指定过滤器的文件和目录。如果给定filter为null，则接受所有路径名。否则，当且仅当在路径名上调用过滤器的`FilenameFilter.accept(File dir, String name)`方法返回true时，该路径名才满足过滤器。如果当前File对象不表示一个目录，或者发生I/O错误，则返回null。
- `public File[] listFiles(FilenameFilter filter)`：返回返回所有满足指定过滤器的文件和目录。如果给定filter为null，则接受所有路径名。否则，当且仅当在路径名上调用过滤器的`FilenameFilter.accept(File dir, String name)`方法返回true时，该路径名才满足过滤器。如果当前File对象不表示一个目录，或者发生I/O错误，则返回null。

```
package com.atguigu.file;  
  
import org.junit.Test;  
  
import java.io.File;  
import java.io.FileFilter;  
import java.io.FilenameFilter;  
  
public class DirListFiles {  
    @Test  
    public void test01() {  
        File dir = new File("d:/atguigu");  
        String[] subs = dir.list();  
        for (String sub : subs) {  
            System.out.println(sub);  
        }  
    }  
  
    @Test  
    public void test02() {  
        File dir = new File("d:/atguigu");  
        listSubFiles(dir);  
    }  
  
    public void listSubFiles(File dir) {  
        if (dir != null && dir.isDirectory()) {  
            File[] listFiles = dir.listFiles();  
            if (listFiles != null) {  
                for (File sub : listFiles) {  
                    listSubFiles(sub); //递归调用  
                }  
            }  
        }  
    }  
}
```

```
        }
    }
}
System.out.println(dir);
}

@Test
public void test03() {
    File dir = new File("D:/atguigu");
    listByFilenameFilter(dir);
}

public void listByFilenameFilter(File file) {
    if (file != null && file.isDirectory()) {
        File[] listFiles = file.listFiles(new FilenameFilter() {
            @Override
            public boolean accept(File dir, String name) {
                return name.endsWith(".java") || new File(dir, name).isDirectory();
            }
        });
        if (listFiles != null) {
            for (File sub : listFiles) {
                if (sub != null && sub.isFile()) {
                    System.out.println(sub);
                }
                listByFilenameFilter(sub); // 递归调用
            }
        }
    }
}

@Test
public void test04() {
    File dir = new File("D:/atguigu");
    listByFileFilter(dir);
}

public void listByFileFilter(File file) {
    if (file != null && file.isDirectory()) {
        File[] listFiles = file.listFiles(new FileFilter() {
            @Override
            public boolean accept(File pathname) {
                return pathname.getName().endsWith(".java") || pathname.isDirectory();
            }
        });
        if (listFiles != null) {
            for (File sub : listFiles) {
                if (sub != null && sub.isFile()) {
                    System.out.println(sub);
                }
                listByFileFilter(sub); // 递归调用
            }
        }
    }
}
```

```
    }
}
}
```

14.2 IO概述

1、什么是IO

生活中，你肯定经历过这样的场景。当你编辑一个文本文件，忘记了 `ctrl+s`，可能文件就白白编辑了。当你电脑上插入一个U盘，可以把一个视频，拷贝到你的电脑硬盘里。那么数据都是在哪些设备上的呢？键盘、内存、硬盘、外接设备等等。

我们把这种数据的传输，可以看做是一种数据的流动，按照流动的方向，以内存为基准，分为输入 `input` 和输出 `output`，即流向内存是输入流，流出内存的输出流。

Java中I/O操作主要是指使用 `java.io` 包下的内容，进行输入、输出操作。**输入**也叫做**读取**数据，**输出**也叫做**写出**数据。

2、IO的分类

根据数据的流向分为：**输入流和输出流**。

- **输入流**：把数据从 其他设备 上读取到 内存 中的流。
 - 以 `InputStream`, `Reader` 结尾
- **输出流**：把数据从 内存 中写出到 其他设备 上的流。
 - 以 `OutputStream`, `Writer` 结尾

根据数据的类型分为：**字节流和字符流**。

- **字节流**：以字节为单位，读写数据的流。
 - 以 `InputStream` 和 `OutputStream` 结尾
- **字符流**：以字符为单位，读写数据的流。
 - 以 `Reader` 和 `Writer` 结尾

根据IO流的角色不同分为：**节点流和处理流**。

- **节点流**：可以从或向一个特定的地方（节点）读写数据。如 `FileReader`.
- **处理流**：是对一个已存在的流进行连接和封装，通过所封装的流的功能调用实现数据读写。如 `BufferedReader`.
处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。

这种设计是装饰模式（Decorator Pattern）也称为包装模式（Wrapper Pattern），其使用一种对客户端透明的方式来动态地扩展对象的功能，它是通过继承扩展功能的替代方案之一。在现实生活中你也有很多装饰者的例子，例如：人需要各种各样的衣着，不管你穿着怎样，但是，对于你个人本质来说是不变的，充其量只是在外面加上了一些装饰，有，“遮羞的”、“保暖的”、“好看的”、“防雨的”....

常用的节点流：

- 文件 `FileInputStream` `FileOutputStream` `FileReader` `FileWriter` 文件进行处理的节点流。
- 字符串 `StringReader` `StringWriter` 对字符串进行处理的节点流。

- 数组 ByteArrayInputStream、ByteArrayOutputStream、CharArrayReader、CharArrayWriter 对数组进行处理的节点流(对应的不再是文件，而是内存中的一个数组)。
- 管道 PipedInputStream、PipedOutputStream、PipedReader、PipedWriter 对管道进行处理的节点流。

常用处理流：

- 缓冲流：BufferedInputStream、BufferedOutputStream、BufferedReader、BufferedWriter---增加缓冲功能，避免频繁读写硬盘。
- 转换流：InputStreamReader、OutputStreamReader---实现字节流和字符流之间的转换。
- 数据流：DataInputStream、DataOutputStream - 提供读写Java基础数据类型功能
- 对象流：ObjectInputStream、ObjectOutputStream--提供直接读写Java对象功能

3、4大顶级抽象父类们

| | 输入流 | 输出流 |
|------------|--------------------------|---------------------------|
| 字节流 | 字节输入流 InputStream | 字节输出流 OutputStream |
| 字符流 | 字符输入流 Reader | 字符输出流 Writer |

14.3 字节流

14.3.1 一切皆为字节

一切文件数据(文本、图片、视频等)在存储时，都是以二进制数字的形式保存，都一个一个的字节，那么传输时一样如此。所以，字节流可以传输任意文件数据。在操作流的时候，我们要时刻明确，无论使用什么样的流对象，底层传输的始终为二进制数据。

14.3.2 字节输出流【OutputStream】

`java.io.OutputStream` 抽象类是表示字节输出流的所有类的超类，将指定的字节信息写出到目的地。它定义了字节输出流的基本共性功能方法。

- `public void write(int b)`：将指定的字节输出流。虽然参数为int类型四个字节，但是只会保留一个字节的信息写出。
- `public void write(byte[] b)`：将 b.length字节从指定的字节数组写入此输出流。
- `public void write(byte[] b, int off, int len)`：从指定的字节数组写入 len字节，从偏移量 off开始输出到此输出流。
- `public void flush()`：刷新此输出流并强制任何缓冲的输出字节被写出。
- `public void close()`：关闭此输出流并释放与此流相关联的任何系统资源。

小贴士：close方法，当完成流的操作时，必须调用此方法，释放系统资源。

14.3.3 FileOutputStream类

`OutputStream`有很多子类，我们从最简单的一个子类开始。`java.io.FileOutputStream`类是文件输出流，用于将数据写出到文件。

- `public FileOutputStream(File file)`：创建文件输出流以写入由指定的 File对象表示的文件。
- `public FileOutputStream(String name)`： 创建文件输出流以指定的名称写入文件。

当你创建一个流对象时，必须传入一个文件路径。如果该文件不存在，会创建该文件。如果有这个文件，会清空这个文件的数据。如果传入的是一个目录，则会报IOException异常。

1、写出字节数据

```
package com.atguigu.fileio;

import org.junit.Test;

import java.io.FileOutputStream;
import java.io.IOException;

public class Foswrite {
    @Test
    public void test01() throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 写出数据
        fos.write(97); // 写出第1个字节
        fos.write(98); // 写出第2个字节
        fos.write(99); // 写出第3个字节
        // 关闭资源
        fos.close();
        /* 输出结果: abc*/
    }

    @Test
    public void test02() throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 字符串转换为字节数组
        byte[] b = "尚硅谷".getBytes();
        // 写出字节数组数据
        fos.write(b);
        // 关闭资源
        fos.close();
    }

    @Test
    public void test03() throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 字符串转换为字节数组
        byte[] b = "abcde".getBytes();
        // 写出从索引2开始，2个字节。索引2是c，两个字节，也就是cd。
        fos.write(b, 2, 2);
        // 关闭资源
        fos.close();
    }
}
```

2、数据追加续写

经过以上的演示，每次程序运行，创建输出流对象，都会清空目标文件中的数据。如何保留目标文件中数据，还能继续添加新数据呢？

- `public FileOutputStream(File file, boolean append)`： 创建文件输出流以写入由指定的 File 对象表示的文件。
- `public FileOutputStream(String name, boolean append)`： 创建文件输出流以指定的名称写入文件。

这两个构造方法，参数中都需要传入一个 boolean 类型的值，`true` 表示追加数据，`false` 表示清空原有数据。这样创建的输出流对象，就可以指定是否追加续写了，代码使用演示：

```
package com.atguigu.fileio;

import java.io.FileOutputStream;
import java.io.IOException;

public class FOSwriteAppend {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt", true);
        // 字符串转换为字节数组
        byte[] b = "abcde".getBytes();
        fos.write(b);
        // 关闭资源
        fos.close();
    }
}

//这段程序如果多运行几次，每次都会在原来文件末尾追加abcde
```

3、写出换行

- 回车符 `\r` 和换行符 `\n`：
 - 回车符：回到一行的开头（return）。
 - 换行符：下一行（newline）。
- 系统中的换行：
 - Windows 系统里，每行结尾是 回车+换行，即 `\r\n`；
 - Unix 系统里，每行结尾只有 换行，即 `\n`；
 - Mac 系统里，每行结尾是 回车，即 `\r`。从 Mac OS X 开始与 Linux 统一。

代码使用演示：

```
package com.atguigu.fileio;

import java.io.FileOutputStream;
import java.io.IOException;

public class FOSwriteNewLine {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 定义字节数组
        byte[] words = {97, 98, 99, 100, 101};
```

```

    // 遍历数组
    for (int i = 0; i < words.length; i++) {
        // 写出一个字节
        fos.write(words[i]);
        // 写出一个换行，换行符号转成数组写出
        fos.write("\r\n".getBytes());
    }
    // 关闭资源
    fos.close();
}

/*
输出结果:
    a
    b
    c
    d
    e*/

```

14.3.4 字节输入流【InputStream】

`java.io.InputStream` 抽象类是表示字节输入流的所有类的超类，可以读取字节信息到内存中。它定义了字节输入流的基本共性功能方法。

- `public int read()`：从输入流读取一个字节。返回读取的字节值。虽然读取了一个字节，但是会自动提升为int类型。如果已经到达流末尾，没有数据可读，则返回-1。
- `public int read(byte[] b)`：从输入流中读取一些字节数，并将它们存储到字节数组 b 中。每次最多读取 b.length个字节。返回实际读取的字节个数。如果已经到达流末尾，没有数据可读，则返回-1。
- `public int read(byte[] b, int off, int len)`：从输入流中读取一些字节数，并将它们存储到字节数组 b 中，从b[off]开始存储，每次最多读取len个字节。返回实际读取的字节个数。如果已经到达流末尾，没有数据可读，则返回-1。
- `public void close()`：关闭此输入流并释放与此流相关联的任何系统资源。

小贴士：close方法，当完成流的操作时，必须调用此方法，释放系统资源。

14.3.5 FileInputStream类

`java.io.FileInputStream` 类是文件输入流，从文件中读取字节。

- `FileInputStream(File file)`：通过打开与实际文件的连接来创建一个 `FileInputStream`，该文件由文件系统中的 File对象 file命名。
- `FileInputStream(String name)`：通过打开与实际文件的连接来创建一个 `FileInputStream`，该文件由文件系统中的路径名 name命名。

当你创建一个流对象时，必须传入一个文件路径。如果文件不存在，会抛出 `FileNotFoundException`。如果传入的是一个目录，则会报`IOException`异常。

```

package com.atguigu.fileio;

import org.junit.Test;

```

```
import java.io.FileInputStream;
import java.io.IOException;

public class FISRead {
    @Test
    public void test() throws IOException {
        // 使用文件名称创建流对象
        FileInputStream fis = new FileInputStream("read.txt");
        // 读取数据，返回一个字节
        int read = fis.read();
        System.out.println((char) read);
        // 读取到末尾,返回-1
        read = fis.read();
        System.out.println( read);
        // 关闭资源
        fis.close();
        /*
        文件内容: abcde
        输出结果:
        a
        b
        c
        d
        e
        -1
        */
    }

    @Test
    public void test02() throws IOException{
        // 使用文件名称创建流对象
        FileInputStream fis = new FileInputStream("read.txt");
        // 定义变量, 保存数据
        int b;
        // 循环读取
        while ((b = fis.read())!=-1) {
            System.out.println((char)b);
        }
        // 关闭资源
        fis.close();
    }

    @Test
    public void test03() throws IOException{
```

```

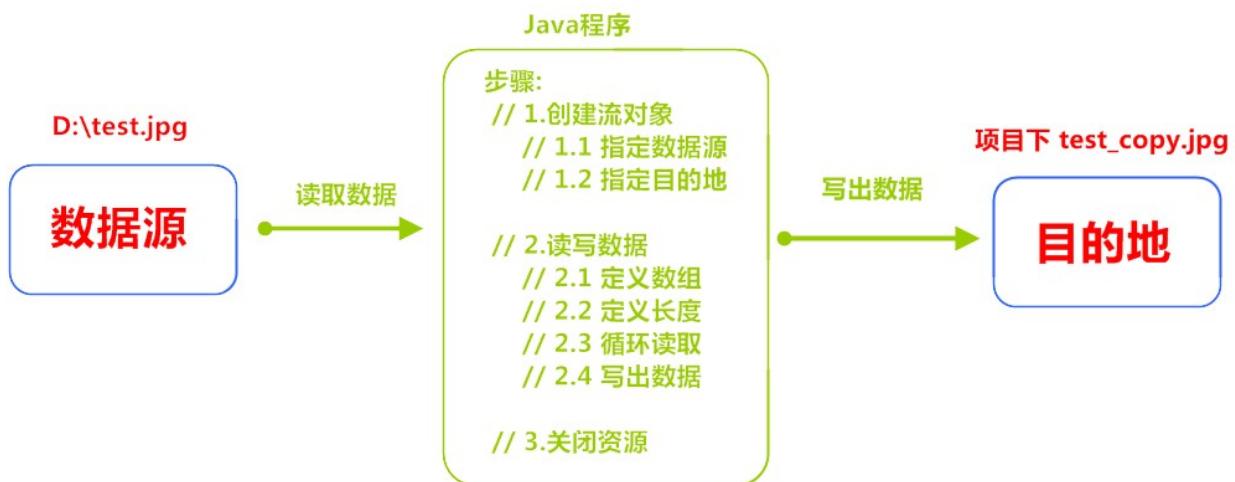
// 使用文件名称创建流对象.
FileInputStream fis = new FileInputStream("read.txt"); // 文件中为abcde
// 定义变量, 作为有效个数
int len;
// 定义字节数组, 作为装字节数据的容器
byte[] b = new byte[2];
// 循环读取
while ((len= fis.read(b))!=-1) {
    // 每次读取后, 把数组变成字符串打印
    System.out.println(new String(b));
}
// 关闭资源
fis.close();
/*
输出结果:
ab
cd
ed
最后错误数据`d`，是由于最后一次读取时，只读取一个字节`e`，数组中，上次读取的数据没有被完全替换，所以要通过`len`，获取有效的字节
*/
}

@Test
public void test04() throws IOException{
    // 使用文件名称创建流对象.
    FileInputStream fis = new FileInputStream("read.txt"); // 文件中为abcde
    // 定义变量, 作为有效个数
    int len;
    // 定义字节数组, 作为装字节数据的容器
    byte[] b = new byte[2];
    // 循环读取
    while ((len= fis.read(b))!=-1) {
        // 每次读取后, 把数组的有效字节部分, 变成字符串打印
        System.out.println(new String(b,0,len)); // len 每次读取的有效字节个数
    }
    // 关闭资源
    fis.close();
    /*
    输出结果:
    ab
    cd
    e
    */
}
}

```

14.3.6 复制文件

原理:从已有文件中读取字节,将该字节写出到另一个文件中



复制图片文件，代码使用演示：

```
package com.atguigu.fileio;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopy {
    public static void main(String[] args) throws IOException {
        // 1. 创建流对象
        // 1.1 指定数据源
        FileInputStream fis = new FileInputStream("D:\\\\test.jpg");
        // 1.2 指定目的地
        FileOutputStream fos = new FileOutputStream("test_copy.jpg");

        // 2. 读写数据
        // 2.1 定义数组
        byte[] b = new byte[1024];
        // 2.2 定义长度
        int len;
        // 2.3 循环读取
        while ((len = fis.read(b)) != -1) {
            // 2.4 写出数据
            fos.write(b, 0, len);
        }

        // 3. 关闭资源
        fos.close();
        fis.close();
    }
}
```

14.4 字符流

当使用字节流读取文本文件时，可能会有一个小问题。就是遇到中文字符时，可能不会显示完整的字符，那是因为一个中文字符可能占用多个字节存储。所以Java提供一些字符流类，以字符为单位读写数据，专门用于处理文本文件。

小贴士：字符流，只能操作文本文件，不能操作图片，视频等非文本文件。当我们单纯读或者写文本文件时 使用字符流 其他情况使用字节流。

14.4.1 字符输入流【Reader】

`java.io.Reader` 抽象类是表示用于读取字符流的所有类的超类，可以读取字符信息到内存中。它定义了字符输入流的基本共性功能方法。

- `public int read()`：从输入流读取一个字符。虽然读取了一个字符，但是会自动提升为int类型。返回该字符的Unicode编码值。如果已经到达流末尾了，则返回-1。
- `public int read(char[] cbuf)`：从输入流中读取一些字符，并将它们存储到字符数组 `cbuf` 中。每次最多读取`cbuf.length`个字符。返回实际读取的字符个数。如果已经到达流末尾，没有数据可读，则返回-1。
- `public int read(char[] cbuf, int off, int len)`：从输入流中读取一些字符，并将它们存储到字符数组 `cbuf` 中，从`cbuf[off]`开始的位置存储。每次最多读取`len`个字符。返回实际读取的字符个数。如果已经到达流末尾，没有数据可读，则返回-1。
- `public void close()`：关闭此流并释放与此流相关联的任何系统资源。

小贴士：close方法，当完成流的操作时，必须调用此方法，释放系统资源。

14.4.2 FileReader类

`java.io.FileReader` 类是读取字符文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

- `FileReader(File file)`：创建一个新的 `FileReader`，给定要读取的 `File` 对象。
- `FileReader(String fileName)`：创建一个新的 `FileReader`，给定要读取的文件的名称。

当你创建一个流对象时，必须传入一个文件路径。类似于 `FileInputStream`。如果该文件不存在，则报 `FileNotFoundException`。如果传入的是一个目录，则会报 `IOException` 异常。

```
package com.atguigu.fileio;

import org.junit.Test;

import java.io.FileReader;
import java.io.IOException;

public class FRRead {
    @Test
    public void test01() throws IOException {
        // 使用文件名称创建流对象
        FileReader fr = new FileReader("read.txt");
        // 定义变量，保存数据
        int b;
        // 循环读取
        while ((b = fr.read()) != -1) {
            System.out.println((char)b);
        }
        // 关闭资源
        fr.close();
    }
    /*输出结果:
    */
}
```

```
    尚
    硅
    谷*/
}

@Test
public void test02() throws IOException {
    // 使用文件名称创建流对象
    FileReader fr = new FileReader("read.txt");
    // 定义变量，保存有效字符个数
    int len;
    // 定义字符数组，作为装字符数据的容器
    char[] cbuf = new char[2];
    // 循环读取
    while ((len = fr.read(cbuf)) != -1) {
        System.out.println(new String(cbuf));
    }
    // 关闭资源
    fr.close();
    /*
    输出结果:
    尚硅
    谷硅
    最后错误数据硅，是因为最后一次流中只有一个字符“谷”，读取一个字符没有覆盖char[]数组cbuf的所有元素
    */
}

@Test
public void test03() throws IOException {
    // 使用文件名称创建流对象
    FileReader fr = new FileReader("read.txt");
    // 定义变量，保存有效字符个数
    int len;
    // 定义字符数组，作为装字符数据的容器
    char[] cbuf = new char[2];
    // 循环读取
    while ((len = fr.read(cbuf)) != -1) {
        System.out.println(new String(cbuf, 0, len));
    }
    // 关闭资源
    fr.close();
    /*
    输出结果:
    尚硅
    谷
    */
}

}
```

14.4.3 字符输出流【Writer】

`java.io.Writer` 抽象类是表示用于写出字符流的所有类的超类，将指定的字符信息写出到目的地。它定义了字节输出流的基本共性功能方法。

- `public void write(int c)` 写入单个字符。
- `public void write(char[] cbuf)` 写入字符数组。
- `public void write(char[] cbuf, int off, int len)` 写入字符数组的某一部分,off数组的开始索引,len写入的字符个数。
- `public void write(String str)` 写入字符串。
- `public void write(String str, int off, int len)` 写入字符串的某一部分,off字符串的开始索引,len写入的字符个数。
- `public void flush()` 刷新该流的缓冲。
- `public void close()` 关闭此流，但要先刷新它。

14.4.4 FileWriter类

`java.io.FileWriter` 类是写出字符到文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

- `FileWriter(File file)`： 创建一个新的 `FileWriter`，给定要读取的 `File` 对象。
- `FileWriter(String fileName)`： 创建一个新的 `FileWriter`，给定要读取的文件的名称。

当你创建一个流对象时，必须传入一个文件路径，类似于 `FileOutputStream`。如果文件不存在，则会自动创建。如果文件已经存在，则会清空文件内容，写入新的内容。

1、写出字符数据

```
package com.atguigu.fileio;

import org.junit.Test;

import java.io.FileWriter;
import java.io.IOException;

public class Fwwrite {
    @Test
    public void test01() throws IOException {
        // 使用文件名称创建流对象
        FileWriter fw = new FileWriter("fw.txt");
        // 写出数据
        fw.write(97); // 写出第1个字符
        fw.write('b'); // 写出第2个字符
        fw.write('c'); // 写出第3个字符
        fw.write(30000); // 写出第4个字符，中文编码表中30000对应一个汉字。

        /*
         * 【注意】FileWriter与FileOutputStream不同。
         * 如果不关闭，数据只是保存到缓冲区，并未保存到文件。
         */
        // fw.close();
    }

    @Test
    public void test02() throws IOException {
```

```

// 使用文件名称创建流对象
FileWriter fw = new FileWriter("fw.txt");
// 字符串转换为字节数组
char[] chars = "尚硅谷".toCharArray();

// 写出字符数组
fw.write(chars); // 尚硅谷

// 写出从索引1开始, 2个字符。
fw.write(chars,1,2); // 硅谷

// 关闭资源
fw.close();
}

@Test
public void test03() throws IOException {
    // 使用文件名称创建流对象
    FileWriter fw = new FileWriter("fw.txt");
    // 字符串
    String msg = "尚硅谷";

    // 写出字符数组
    fw.write(msg); // 尚硅谷

    // 写出从索引1开始, 2个字符。
    fw.write(msg,1,2); // 硅谷

    // 关闭资源
    fw.close();
}
}

```

2、续写

- `public FileWriter(File file,boolean append)`： 创建文件输出流以写入由指定的 File对象表示的文件。
- `public FileWriter(String fileName,boolean append)`： 创建文件输出流以指定的名称写入文件。

这两个构造方法，参数中都需要传入一个boolean类型的值，`true` 表示追加数据，`false` 表示清空原有数据。这样创建的输出流对象，就可以指定是否追加续写了，代码使用演示：

操作类似于`FileOutputStream`。

```

package com.atguigu.fileio;

import org.junit.Test;

import java.io.FileWriter;
import java.io.IOException;

```

```
public class FwwriteAppend {
    @Test
    public void test01() throws IOException {
        // 使用文件名称创建流对象，可以续写数据
        FileWriter fw = new FileWriter("fw.txt", true);
        // 写出字符串
        fw.write("尚硅谷真棒");
        // 关闭资源
        fw.close();
    }
}
```

3、换行

```
package com.atguigu.fileio;

import java.io.FileWriter;
import java.io.IOException;

public class Fwwrite.NewLine {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象，可以续写数据
        FileWriter fw = new FileWriter("fw.txt");
        // 写出字符串
        fw.write("尚");
        // 写出换行
        fw.write("\r\n");
        // 写出字符串
        fw.write("硅谷");
        // 关闭资源
        fw.close();
    }
}
```

4、关闭和刷新

【注意】`FileWriter`与`FileOutputStream`不同。因为内置缓冲区的原因，如果不关闭输出流，无法写出字符到文件中。但是关闭的流对象，是无法继续写出数据的。如果我们既想写出数据，又想继续使用流，就需要`flush`方法了。

- `flush`：刷新缓冲区，流对象可以继续使用。
- `close`：先刷新缓冲区，然后通知系统释放资源。流对象不可以再被使用了。

代码使用演示：

```
package com.atguigu.fileio;

import java.io.FileWriter;
import java.io.IOException;

public class FwwriteFlush {
    public static void main(String[] args) throws IOException {
```

```
// 使用文件名称创建流对象
FileWriter fw = new FileWriter("fw.txt");
// 写出数据，通过flush
fw.write('刷'); // 写出第1个字符
fw.flush();
fw.write('新'); // 继续写出第2个字符，写出成功
fw.flush();

// 写出数据，通过close
fw.write('关'); // 写出第1个字符
fw.close();
fw.write('闭'); // 继续写出第2个字符，【报错】java.io.IOException: Stream closed
fw.close();
}

}
```

小贴士：即便是flush方法写出了数据，操作的最后还是要调用close方法，释放系统资源。

14.5 缓冲流

缓冲流也叫高效流，按照数据类型分类：

- **字节缓冲流**: `BufferedInputStream`, `BufferedOutputStream`
- **字符缓冲流**: `BufferedReader`, `BufferedWriter`

缓冲流的基本原理，是在创建流对象时，会创建一个内置的默认大小的缓冲区数组，通过缓冲区读写，减少系统IO次数，从而提高读写效率。

14.5.1 构造方法

- `public BufferedInputStream(InputStream in)` : 创建一个新的缓冲输入流。
- `public BufferedOutputStream(OutputStream out)` : 创建一个新的缓冲输出流。

构造举例，代码如下：

```
// 创建字节缓冲输入流
BufferedInputStream bis = new BufferedInputStream(new FileInputStream("bis.txt"));
// 创建字节缓冲输出流
BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("bos.txt"));
```

- `public BufferedReader(Reader in)` : 创建一个新的缓冲输入流。
- `public BufferedWriter(Writer out)` : 创建一个新的缓冲输出流。

构造举例，代码如下：

```
// 创建字符缓冲输入流
BufferedReader br = new BufferedReader(new FileReader("br.txt"));
// 创建字符缓冲输出流
BufferedWriter bw = new BufferedWriter(new FileWriter("bw.txt"));
```

14.5.2 效率测试

查询API，缓冲流读写方法与基本的流是一致的，我们通过复制大文件（375MB），测试它的效率。

```
package com.atguigu.buffer;

import org.junit.Test;

import java.io.*;

public class BufferedIO {
    @Test
    public void testNoBuffer() throws IOException {
        // 记录开始时间
        long start = System.currentTimeMillis();
        // 创建流对象
        FileInputStream fis = new FileInputStream("jdk8.exe");
        FileOutputStream fos = new FileOutputStream("copy.exe");
        // 读写数据
        byte[] data = new byte[1024];
        int len;
        while ((len = fis.read(data)) != -1) {
            fos.write(data, 0, len);
        }

        fos.close();
        fis.close();

        // 记录结束时间
        long end = System.currentTimeMillis();
        System.out.println("普通流复制时间:" + (end - start) + " 毫秒");
    }

    @Test
    public void testUseBuffer() throws IOException {
        // 记录开始时间
        long start = System.currentTimeMillis();
        // 创建流对象
        BufferedInputStream bis = new BufferedInputStream(new FileInputStream("jdk8.exe"));
        BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream("copy.exe"));
        // 读写数据
        int len;
        byte[] data = new byte[1024];
        while ((len = bis.read(data)) != -1) {
            bos.write(data, 0, len);
        }

        bos.close();
        bis.close();
        // 记录结束时间
        long end = System.currentTimeMillis();
        System.out.println("缓冲流使用数组复制时间:" + (end - start) + " 毫秒");
    }
}
```

```
    }  
}
```

14.5.3 字符缓冲流特有方法

字符缓冲流的基本方法与普通字符流调用方式一致，不再阐述，我们来看它们具备的特有方法。

- BufferedReader: `public String readLine()`: 读一行文字。
- BufferedWriter: `public void newLine()`: 写一行行分隔符,由系统属性定义符号。

```
package com.atguigu.buffer;  
  
import org.junit.Test;  
  
import java.io.*;  
  
public class BufferedIOLine {  
    @Test  
    public void testReadLine() throws IOException {  
        // 创建流对象  
        BufferedReader br = new BufferedReader(new FileReader("in.txt"));  
        // 定义字符串,保存读取的一行文字  
        String line;  
        // 循环读取,读取到最后返回null  
        while ((line = br.readLine()) != null) {  
            System.out.println(line);  
        }  
        // 释放资源  
        br.close();  
    }  
  
    @Test  
    public void testNewLine() throws IOException{  
        // 创建流对象  
        BufferedWriter bw = new BufferedWriter(new FileWriter("out.txt"));  
        // 写出数据  
        bw.write("尚");  
        // 写出换行  
        bw.newLine();  
        bw.write("硅");  
        bw.newLine();  
        bw.write("谷");  
        bw.newLine();  
        // 释放资源  
        bw.close();  
    }  
}
```

14.5.4 流的关闭顺序

```
package com.atguigu.buffer;

import org.junit.Test;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class IOclose {
    @Test
    public void test01() throws IOException {
        FileWriter fw = new FileWriter("d:/1.txt");
        BufferedWriter bw = new BufferedWriter(fw);

        bw.write("hello");

        fw.close();
        bw.close(); //java.io.IOException: Stream closed
        /*
        缓冲流BufferedWriter，把数据先写到缓冲区，  

        默认情况下是当缓冲区满，或调用close，或调用flush这些情况才会把缓冲区的数据输出。
        bw.close()时，需要把数据从缓冲区的数据输出。
        数据的流向： 写到bw（缓冲区） --> fw ->"d:/1.txt"  

        此时，我先把fw关闭了，bw的数据无法输出了
        */
    }

    @Test
    public void test02() throws IOException {
        FileWriter fw = new FileWriter("d:/1.txt");
        BufferedWriter bw = new BufferedWriter(fw);

        bw.write("hello");

        bw.close();
        fw.close();
        /*
        原则：  

        先关外面的，再关里面的。  

        例如：  

        FileWriter fw = new FileWriter("d:/1.txt"); //里面  

        BufferedWriter bw = new BufferedWriter(fw); //外面  

        关闭  

        bw.close(); //先关外面的  

        fw.close(); //再关里面的  

        */
    }
}
```

14.6 转换流

14.6.1 字符编码和字符集

1、编码与解码

计算机中储存的信息都是用二进制数表示的，而我们在屏幕上看到的数字、英文、标点符号、汉字等字符是二进制数转换之后的结果。按照某种规则，将字符存储到计算机中，称为**编码**。反之，将存储在计算机中的二进制数按照某种规则解析显示出来，称为**解码**。比如说，按照A规则存储，同样按照A规则解析，那么就能显示正确的文本符号。反之，按照A规则存储，再按照B规则解析，就会导致乱码现象。

编码:字符(人能看懂的)-->字节(人看不懂的)

解码:字节(人看不懂的)-->字符(人能看懂的)

- **字符编码 Character Encoding** : 就是一套自然语言的字符与二进制数之间的对应规则。

编码表:生活中文字和计算机中二进制的对应规则

2、字符集

- **字符集 Charset** : 也叫编码表。是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等。

计算机要准确的存储和识别各种字符集符号，需要进行字符编码，一套字符集必然至少有一套字符编码。常见字符集有ASCII字符集、GBK字符集、Unicode字符集等。



可见，当指定了**编码**，它所对应的**字符集**自然就指定了，所以**编码**才是我们最终要关心的。

- **ASCII字符集** :

- ASCII (American Standard Code for Information Interchange, 美国信息交换标准代码) 是基于拉丁字母的一套电脑编码系统，用于显示现代英语，主要包括控制字符 (回车键、退格、换行键等) 和可显示字符 (英文大小写字母、阿拉伯数字和西文符号)。
- 基本的ASCII字符集，使用7位 (bits) 表示一个字符，共128字符。
- ASCII的扩展字符集使用8位 (bits) 表示一个字符，共256字符，方便支持欧洲常用字符。

- **ISO-8859-1字符集**:

- 拉丁码表，别名Latin-1，用于显示欧洲使用的语言，包括荷兰、丹麦、德语、意大利语、西班牙语等。
- ISO-8859-1使用单字节编码，兼容ASCII编码。

- **GBxxx字符集**:

- GB就是国标的意思，是为了显示中文而设计的一套字符集。
- **GB2312**: 简体中文码表。一个小于127的字符的意义与原来相同。但两个大于127的字符连在一起时，就表示一个汉字，这样大约可以组合了包含7000多个简体汉字，此外数学符号、罗马希腊的字母、日文的假名们都编进去了，连在ASCII里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的“全角”字符，而原来在127号以下的那些就叫“半角”字符了。
- **GBK**: 最常用的中文码表。是在GB2312标准基础上的扩展规范，使用了双字节编码方案，共收录了21003个汉字，完全兼容GB2312标准，同时支持繁体汉字以及日韩汉字等。
- **GB18030**: 最新的中文码表。收录汉字70244个，采用多字节编码，每个字可以由1个、2个或4个字节组成。支持中国国内少数民族的文字，同时支持繁体汉字以及日韩汉字等。

- **Unicode字符集** :

- Unicode编码系统为表达任意语言的任意字符而设计，是业界的一种标准，也称为统一码、标准万国码。
- 它最多使用4个字节的数字来表达每个字母、符号，或者文字。有三种编码方案，UTF-8、UTF-16和UTF-32。最为常用的UTF-8编码。
- UTF-8编码，可以用来表示Unicode标准中任何字符，它是电子邮件、网页及其他存储或传送文字的应用中，优先采用的编码。互联网工程工作小组 (IETF) 要求所有互联网协议都必须支持UTF-8编码。所以，我们开发Web应用，也要使用UTF-8编码。它使用一至四个字节为每个字符编码，编码规则：
 1. 128个US-ASCII字符，只需一个字节编码。
 2. 拉丁文等字符，需要二个字节编码。
 3. 大部分常用字（含中文），使用三个字节编码。
 4. 其他极少使用的Unicode辅助字符，使用四字节编码。

14.6.2 编码引出的问题

使用 `FileReader` 读取项目中的文本文件。由于项目设置了UTF-8编码，当读取Windows系统中创建的文本文件时，由于Windows系统的默认是GBK编码，就会出现乱码。

```
package com.atguigu.transfer;

import java.io.FileReader;
import java.io.IOException;

public class Problem {
    public static void main(String[] args) throws IOException {
        FileReader fileReader = new FileReader("E:\\File_GBK.txt");
        int read;
        while ((read = fileReader.read()) != -1) {
            System.out.print((char)read);
        }
        fileReader.close();
    }
}
```

输出结果：

◆◆◆

那么如何读取GBK编码的文件呢？

14.6.3 InputStreamReader类

转换流 `java.io.InputStreamReader`，是Reader的子类，是从字节流到字符流的桥梁。它读取字节，并使用指定的字符集将其解码为字符。它的字符集可以由名称指定，也可以接受平台的默认字符集。

- `InputStreamReader(InputStream in)`：创建一个使用默认字符集的字符流。
- `InputStreamReader(InputStream in, String charsetName)`：创建一个指定字符集的字符流。

构造举例，代码如下：

```
InputStreamReader isr = new InputStreamReader(new FileInputStream("in.txt"));
InputStreamReader isr2 = new InputStreamReader(new FileInputStream("in.txt") , "GBK");
```

示例代码：

```
package com.atguigu.transfer;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class InputStreamReaderDemo {
    public static void main(String[] args) throws IOException {
        // 定义文件路径,文件为gbk编码
        String fileName = "E:\\file_gbk.txt";
        // 创建流对象,默认UTF8编码
        InputStreamReader isr1 = new InputStreamReader(new FileInputStream(fileName));
        // 定义变量,保存字符
        int charData;
        // 使用默认编码字符流读取,乱码
        while ((charData = isr1.read()) != -1) {
            System.out.print((char)charData); // ♫♪h♪
        }
        isr1.close();

        // 创建流对象,指定GBK编码
        InputStreamReader isr2 = new InputStreamReader(new FileInputStream(fileName) ,
"GBK");
        // 使用指定编码字符流读取,正常解析
        while ((charData = isr2.read()) != -1) {
            System.out.print((char)charData); // 大家好
        }
        isr2.close();
    }
}
```

14.6.4 OutputStreamWriter类

转换流 `java.io.OutputStreamWriter`，是Writer的子类，是从字符流到字节流的桥梁。使用指定的字符集将字符编码为字节。它的字符集可以由名称指定，也可以接受平台的默认字符集。

- `OutputStreamWriter(OutputStream in)`：创建一个使用默认字符集的字符流。
- `OutputStreamWriter(OutputStream in, String charsetName)`：创建一个指定字符集的字符流。

构造举例，代码如下：

```
OutputStreamWriter isr = new OutputStreamWriter(new FileOutputStream("out.txt"));
OutputStreamWriter isr2 = new OutputStreamWriter(new FileOutputStream("out.txt") , "GBK");
```

示例代码：

```
package com.atguigu.transfer;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class OutputStreamWriterDemo {
    public static void main(String[] args) throws IOException {
        // 定义文件路径
        String fileName = "E:\\out_utf8.txt";
        // 创建流对象,默认UTF8编码
        OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream(fileName));
        // 写出数据
        osw.write("你好"); // 保存为6个字节
        osw.close();

        // 定义文件路径
        String fileName2 = "E:\\out_gbk.txt";
        // 创建流对象,指定GBK编码
        OutputStreamWriter osw2 = new OutputStreamWriter(new
FileOutputStream(fileName2),"GBK");
        // 写出数据
        osw2.write("你好");// 保存为4个字节
        osw2.close();
    }
}
```

14.6.5 转换流理解图解

转换流是字节与字符间的桥梁！



14.7 数据流与对象流

前面学习的IO流，在程序代码中，要么将数据直接按照字节处理，要么按照字符处理。那么，如果读写Java其他数据类型的数据，怎么办呢？

```
String name = "巫师";
int age = 300;
char gender = '男';
int energy = 5000;
double price = 75.5;
boolean relive = true;

Student stu = new Student("张三", 23, 89);
```

Java提供了数据流和对象流来处理这些类型的数据：

- DataOutputStream：数据输出流允许应用程序以适当方式将基本 Java 数据类型写入输出流中。然后，应用程序可以使用数据输入流（DataInputStream）将数据读入。
- DataInputStream：数据输入流允许应用程序以与机器无关方式从底层输入流中读取基本 Java 数据类型。
- ObjectOutputStream：将 Java 基本数据类型和对象写入字节输出流中。稍后可以使用 ObjectInputStream 将数据读入。通过在流中使用文件可以实现Java各种基本数据类型的数据以及对象的持久存储。如果流是网络套接字流，则可以在另一台主机上或另一个进程中接收这些数据或重构对象。
- ObjectInputStream：ObjectInputStream 对以前使用 ObjectOutputStream 写入的基本数据和对象进行反序列化。

因为DataOutputStream和DataInputStream只支持Java基本数据类型和字符串的读写，而不支持Java对象的对象。而ObjectOutputStream和ObjectInputStream既支持Java基本数据类型的数据读写，又支持Java对象的读写，所以下面直接介绍对象流ObjectOutputStream和ObjectInputStream即可。

- `public ObjectOutputStream(OutputStream out)`：创建一个指定OutputStream的ObjectOutputStream。
- `public ObjectInputStream(InputStream in)`：创建一个指定InputStream的ObjectInputStream。

构造举例，代码如下：

```
FileOutputStream fos = new FileOutputStream("game.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
FileInputStream fis = new FileInputStream("game.dat");
ObjectInputStream ois = new ObjectInputStream(fis);
```

14.7.1 使用对象流读写各种类型的数据

ObjectOutpuStream也从OutputStream父类中继承基本方法：

- `public void write(int b)`：将指定的字节输出流。虽然参数为int类型四个字节，但是只会保留一个字节的信息写出。
- `public void write(byte[] b)`：将 b.length字节从指定的字节数组写入此输出流。
- `public void write(byte[] b, int off, int len)`：从指定的字节数组写入 len字节，从偏移量 off开始输出到此输出流。

- `public void flush()`：刷新此输出流并强制任何缓冲的输出字节被写出。
- `public void close()`：关闭此输出流并释放与此流相关联的任何系统资源。

还支持将各种Java数据类型的数据写入输出流中：

- `public void writeBoolean(boolean val)`: 写入一个 boolean 值。
- `public void writeByte(int val)`: 写入一个8位字节。
- `public void writeShort(int val)`: 写入一个16位的 short 值。
- `public void writeChar(int val)`: 写入一个16位的 char 值。
- `public void writeInt(int val)`: 写入一个32位的 int 值。
- `public void writeLong(long val)`: 写入一个64位的 long 值。
- `public void writeFloat(float val)`: 写入一个32位的 float 值。
- `public void writeDouble(double val)`: 写入一个64位的 double 值。
- `public void writeUTF(String str)`: 将表示长度信息的两个字节写入输出流，后跟字符串 s 中每个字符的 UTF-8 修改版表示形式。如果 s 为 null，则抛出 NullPointerException。根据字符的值，将字符串 s 中每个字符转换成一个字节、两个字节或三个字节的字节组。注意，将 String 作为基本数据写入流中与将它作为 Object 写入流中明显不同。

`ObjectInputStream`除了从`InputStream`父类中继承基本方法之外，

- `public int read()`：从输入流读取一个字节。返回读取的字节值。虽然读取了一个字节，但是会自动提升为int类型。如果已经到达流末尾，没有数据可读，则返回-1。
- `public int read(byte[] b)`：从输入流中读取一些字节数，并将它们存储到字节数组 b 中。每次最多读取 b.length 个字节。返回实际读取的字节个数。如果已经到达流末尾，没有数据可读，则返回-1。
- `public int read(byte[] b, int off, int len)`：从输入流中读取一些字节数，并将它们存储到字节数组 b 中，从 b[off] 开始存储，每次最多读取 len 个字节。返回实际读取的字节个数。如果已经到达流末尾，没有数据可读，则返回-1。
- `public void close()`：关闭此输入流并释放与此流相关联的任何系统资源。

还支持从输入流中读取各种Java数据类型的数据：

- `public boolean readBoolean()`：读取一个 boolean 值。
- `public byte readByte()`：读取一个 8 位的字节。
- `public short readShort()`：读取一个 16 位的 short 值。
- `public char readChar()`：读取一个 16 位的 char 值。
- `public int readInt()`：读取一个 32 位的 int 值。
- `public long readLong()`：读取一个 64 位的 long 值。
- `public float readFloat()`：读取一个 32 位的 float 值。
- `public double readDouble()`：读取一个 64 位的 double 值。
- `public String readUTF()`：读取 UTF-8 修改版格式的 String。

注意：读的顺序和方法与写的顺序和方法要一一对应。

示例代码：

```
package com.atguigu.object;

import org.junit.Test;

import java.io.*;

public class ReadwriteDataOfAnyType {
```

```
@Test
public void save() throws IOException {
    String name = "巫师";
    int age = 300;
    char gender = '男';
    int energy = 5000;
    double price = 75.5;
    boolean relive;

    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("game.dat"));
    oos.writeUTF(name);
    oos.writeInt(age);
    oos.writeChar(gender);
    oos.writeInt(energy);
    oos.writeDouble(price);
    oos.writeBoolean(relive);
    oos.close();
}

@Test
public void reload() throws IOException{
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("game.dat"));
    String name = ois.readUTF();
    int age = ois.readInt();
    char gender = ois.readChar();
    int energy = ois.readInt();
    double price = ois.readDouble();
    boolean relive = ois.readBoolean();

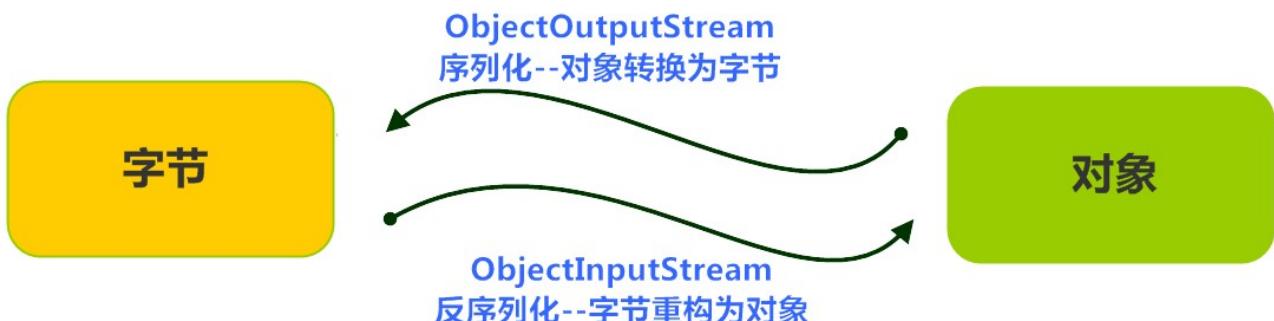
    System.out.println(name + "," + age + "," + gender + "," + energy + "," + price + ","
+ relive);

    ois.close();
}
}
```

14.7.2 序列化与反序列化概念

Java 提供了一种对象**序列化**的机制。用一个字节序列可以表示一个对象，该字节序列包含该对象的类型和对象中存储的属性等信息。字节序列写出到文件之后，相当于文件中**持久保存**了一个对象的信息。

反之，该字节序列还可以从文件中读取回来，重构对象，对它进行**反序列化**。**对象的数据**、**对象的类型**和**对象中存储的数据信息**，都可以用来在内存中创建对象。看图理解序列化：



ObjectOutputStream流中支持序列化的方法是：

- `public final void writeObject (Object obj)` : 将指定的对象写出。

ObjectInputStream流中支持反序列化的方法是：

- `public final Object readObject ()` : 读取一个对象。

14.7.3 Serializable序列化接口与transient关键字

某个类的对象需要序列化输出时，该类必须实现 `java.io.Serializable` 接口，`Serializable` 是一个标记接口，不实现此接口的类将不会使任何状态序列化或反序列化，会抛出 `NotSerializableException`。

- 如果对象的某个属性也是引用数据类型，那么如果该属性也要序列化的话，也要实现 `Serializable` 接口
- 该类的所有属性必须是可序列化的。如果有一个属性不需要可序列化的，则该属性必须注明是瞬态的，使用 `transient` 关键字修饰。
- 静态变量的值不会序列化。因为静态变量的值不属于某个对象。

```
package com.atguigu.object;

import java.io.Serializable;

public class Employee implements Serializable {
    public static String company; //static修饰的类变量，不会被序列化
    public String name;
    public String address;
    public transient int age; // transient瞬态修饰成员，不会被序列化

    public Employee(String name, String address, int age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }

    public static String getCompany() {
        return company;
    }

    public static void setCompany(String company) {
        Employee.company = company;
    }
}
```

```
}

public String getName() {
    return name;
}

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

public String getAddress() {
    return address;
}

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

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

@Override
public String toString() {
    return "Employee{" +
        "name='" + name + '\'' +
        ", address='" + address + '\'' +
        ", age=" + age +
        ", company='" + company +
        '}';
}
}
```

```
package com.atguigu.object;

import org.junit.Test;

import java.io.*;

public class ReadWriteObject {
    @Test
    public void save() throws IOException {
        Employee.setCompany("尚硅谷");
        Employee e = new Employee("张三", "宏福苑", 23);
        // 创建序列化流对象
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("employee.dat"));
        // 写出对象
    }
}
```

```

        oos.writeObject(e);
        // 释放资源
        oos.close();
        System.out.println("Serialized data is saved"); // 姓名, 地址被序列化, 年龄没有被序列化。
    }

    @Test
    public void reload() throws IOException, ClassNotFoundException {
        // 创建反序列化流
        FileInputStream fis = new FileInputStream("employee.dat");
        ObjectInputStream ois = new ObjectInputStream(fis);
        // 读取一个对象
        Employee e = (Employee) ois.readObject();
        // 释放资源
        ois.close();
        fis.close();

        System.out.println(e);
    }

    @Test
    public void writeString() throws IOException{
        ObjectOutputStream oos1 = new ObjectOutputStream(new FileOutputStream("str1.dat"));
        oos1.writeUTF("atguigu");
        oos1.writeInt(1);
        oos1.close();

        ObjectOutputStream oos2 = new ObjectOutputStream(new FileOutputStream("str2.dat"));
        oos2.writeObject("atguigu");
        oos2.writeObject(1);
        oos2.close();
    }
}

```

14.7.4 反序列化失败问题

首先，对于JVM可以反序列化对象，它必须是能够找到class文件的类。如果找不到该类的class文件，则抛出一个 `ClassNotFoundException` 异常。

其次，当JVM反序列化对象时，能找到class文件，但是class文件在序列化对象之后发生了修改，那么反序列化操作也会失败，抛出一个 `InvalidClassException` 异常。发生这个异常的原因如下：

- 该类的序列版本号与从流中读取的类描述符的版本号不匹配
- 该类包含未知数据类型

`Serializable` 接口给需要序列化的类，提供了一个序列版本号。`serialVersionUID` 该版本号的目的在于验证序列化的对象和对应类是否版本匹配。如果没有声明`serialVersionUID`，则每次编译都会产生新的`serialVersionUID`序列化版本ID值，这样如果在序列化完成之后修改了类导致类重新编译，则原来的数据将无法反序列化。所以通常我们都会在实现`Serializable`接口时，声明一个`serialVersionUID`，并为其指定一个值。`serialVersionUID`必须是`static`和`final`修饰的`long`类型的数据，它的值由程序员随意指定即可。

如果声明了serialVersionUID，即使在序列化完成之后修改了类导致类重新编译，则原来的数据也能正常反序列化，只是新增的字段值是默认值而已。

```
package com.atguigu.object;

import java.io.Serializable;

public class Employee implements Serializable {
    private static final long serialVersionUID = 1L; //增加serialVersionUID
    public static String company; //static修饰的类变量，不会被序列化
    public String name;
    public String address;
    public transient int age; // transient瞬态修饰成员，不会被序列化

    public Employee(String name, String address, int age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }

    public static String getCompany() {
        return company;
    }

    public static void setCompany(String company) {
        Employee.company = company;
    }

    public String getName() {
        return name;
    }

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

    public String getAddress() {
        return address;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
```

```
public String toString() {
    return "Employee{" +
        "name='" + name + '\'' +
        ", address='" + address + '\'' +
        ", age=" + age +
        ", company='" + company +
        '}';
}
```

14.7.5 序列化多个对象

如果有多个对象需要序列化，则可以将对象放到集合中，再序列化集合对象即可。

```
package com.atguigu.object;

import org.junit.Test;

import java.io.*;
import java.util.ArrayList;

public class Readwritecollection {
    @Test
    public void save() throws IOException {
        ArrayList<Employee> list = new ArrayList<>();
        list.add(new Employee("张三", "宏福苑", 23));
        list.add(new Employee("李四", "白庙", 24));
        list.add(new Employee("王五", "平西府", 25));
        // 创建序列化流对象
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("employees.dat"));
        // 写出对象
        oos.writeObject(list);
        // 释放资源
        oos.close();
    }

    @Test
    public void reload() throws IOException, ClassNotFoundException {
        // 创建反序列化流
        FileInputStream fis = new FileInputStream("employees.dat");
        ObjectInputStream ois = new ObjectInputStream(fis);
        // 读取一个对象
        ArrayList<Employee> list = (ArrayList<Employee>) ois.readObject();
        // 释放资源
        ois.close();
        fis.close();

        System.out.println(list);
    }
}
```

14.8 重新认识System.out和Scanner

14.8.1 PrintStream类

我们每天都在用的System.out对象是PrintStream类型的。它也是IO流对象。

PrintStream 为其他输出流添加了功能，使它们能够方便地打印各种数据值表示形式。它还提供其他两项功能。与其他输出流不同，PrintStream 永远不会抛出 IOException；另外，PrintStream 可以设置自动刷新。

- PrintStream(File file)：创建具有指定文件且不带自动行刷新的新打印流。
- PrintStream(File file, String csn)：创建具有指定文件名称和字符集且不带自动行刷新的新打印流。
- PrintStream(OutputStream out)：创建新的打印流。
- PrintStream(OutputStream out, boolean autoFlush)：创建新的打印流。autoFlush如果为 true，则每当写入 byte 数组、调用其中一个 println 方法或写入换行符或字节 ('\n') 时都会刷新输出缓冲区。
- PrintStream(OutputStream out, boolean autoFlush, String encoding)：创建新的打印流。
- PrintStream(String fileName)：创建具有指定文件名称且不带自动行刷新的新打印流。
- PrintStream(String fileName, String csn)：创建具有指定文件名称和字符集且不带自动行刷新的新打印流。

| | |
|------|--|
| void | <code>print(boolean b)</code> 打印 boolean 值。 |
| void | <code>print(char c)</code> 打印字符。 |
| void | <code>print(char[] s)</code> 打印字符数组。 |
| void | <code>print(double d)</code> 打印双精度浮点数。 |
| void | <code>print(float f)</code> 打印浮点数。 |
| void | <code>print(int i)</code> 打印整数。 |
| void | <code>print(long l)</code> 打印 long 整数。 |
| void | <code>print(Object obj)</code> 打印对象。 |
| void | <code>print(String s)</code> 打印字符串。 |

| | |
|---|---------------------|
| <code>void <u>println</u>()</code> | 通过写入行分隔符字符串终止当前行。 |
| <code>void <u>println</u>(boolean x)</code> | 打印 boolean 值，然后终止行。 |
| <code>void <u>println</u>(char x)</code> | 打印字符，然后终止该行。 |
| <code>void <u>println</u>(char[] x)</code> | 打印字符数组，然后终止该行。 |
| <code>void <u>println</u>(double x)</code> | 打印 double，然后终止该行。 |
| <code>void <u>println</u>(float x)</code> | 打印 float，然后终止该行。 |
| <code>void <u>println</u>(int x)</code> | 打印整数，然后终止该行。 |
| <code>void <u>println</u>(long x)</code> | 打印 long，然后终止该行。 |
| <code>void <u>println</u>(Object x)</code> | 打印 Object，然后终止该行。 |
| <code>void <u>println</u>(String x)</code> | 打印 String，然后终止该行。 |

```
package com.atguigu.systemio;

import java.io.FileNotFoundException;
import java.io.PrintStream;

public class TestPrintStream {
    public static void main(String[] args) throws FileNotFoundException {
        PrintStream ps = new PrintStream("io.txt");
        ps.println("hello");
        ps.println(1);
        ps.println(1.5);
        ps.close();
    }
}
```

14.8.2 Scanner类

构造方法

- `Scanner(File source)`：构造一个新的 Scanner，它生成的值是从指定文件扫描的。
- `Scanner(File source, String charsetName)`：构造一个新的 Scanner，它生成的值是从指定文件扫描的。
- `Scanner(InputStream source)`：构造一个新的 Scanner，它生成的值是从指定的输入流扫描的。
- `Scanner(InputStream source, String charsetName)`：构造一个新的 Scanner，它生成的值是从指定的输入流扫描的。

常用方法：

- `boolean hasNextXxx()`：如果通过使用`nextXxx()`方法，此扫描器输入信息中的下一个标记可以解释为默认基数中的一个 Xxx 值，则返回 true。

- Xxx nextXxx(): 将输入信息的下一个标记扫描为一个Xxx

```
package com.atguigu.systemio;

import org.junit.Test;

import java.io.*;
import java.util.Scanner;

public class TestScanner {

    @Test
    public void test01() throws IOException {
        Scanner input = new Scanner(System.in);
        PrintStream ps = new PrintStream("1.txt");
        while(true){
            System.out.print("请输入一个单词: ");
            String str = input.nextLine();
            if("stop".equals(str)){
                break;
            }
            ps.println(str);
        }
        input.close();
        ps.close();
    }

    @Test
    public void test2() throws IOException {
        Scanner input = new Scanner(new FileInputStream("1.txt"));
        while(input.hasNextLine()){
            String str = input.nextLine();
            System.out.println(str);
        }
        input.close();
    }
}
```

14.8.3 System类的三个IO流对象

System类中有三个常量对象：

- System.out
- System.in
- System.err

查看System类中这三个常量对象的声明：

```
public final static InputStream in = null;
public final static PrintStream out = null;
public final static PrintStream err = null;
```

奇怪的是，

- 这三个常量对象有final声明，但是却初始化为null。final声明的常量一旦赋值就不能修改，那么null不会空指针异常吗？
- 这三个常量对象为什么要小写？final声明的常量按照命名规范不是应该大写吗？
- 这三个常量的对象有set方法？final声明的常量不是不能修改值吗？set方法是如何修改它们的值的？

final声明的常量，表示在Java的语法体系中它们的值是不能修改的，而这三个常量对象的值是由C/C++等系统函数进行初始化和修改值的，所以它们故意没有用大写，也有set方法。

```
public static void setOut(PrintStream out) {
    checkIO();
    setOut0(out);
}

public static void setErr(PrintStream err) {
    checkIO();
    setErr0(err);
}

public static void setIn(InputStream in) {
    checkIO();
    setIn0(in);
}

private static void checkIO() {
    SecurityManager sm = getSecurityManager();
    if (sm != null) {
        sm.checkPermission(new RuntimePermission("setIO"));
    }
}

private static native void setIn0(InputStream in);
private static native void setOut0(PrintStream out);
private static native void setErr0(PrintStream err);
```

14.9 JDK1.7之后引入新try..catch

14.9.1 IO流关闭和异常处理

```
package com.atguigu.io;

import org.junit.Test;

import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;

/*
 * JDK1.7新增的语法，称为try...catch...with...resource，专门为关闭资源和处理相应异常的新try...catch
 * 形式
 * 语法格式：
 * try(
 *     资源对象的创建和声明
 * ){
```

```
    可能发生异常的业务逻辑代码
}catch(异常类型1 参数名){
    处理异常的代码
}catch(异常类型2 参数名){
    处理异常的代码
}
```

这个形式的try...catch，可以保证在try()中声明的资源，无论是否发生异常，无论是否处理异常，都会自动关闭。这里有一个要求，在try()中的资源对象的类型必须实现java.lang.AutoClosable接口

```
/*
public class TestIoclose {
    @Test
    public void test05() {
        try{
            FileWriter fw = new FileWriter("d:/1.txt");
            BufferedWriter bw = new BufferedWriter(fw);
        }{
            bw.write("hello");
        }catch(IOException e){
            e.printStackTrace();
        }
    }
    @Test
    public void test03() {
        FileWriter fw = null;//提取出来的目的是，为了在finally中仍然可以使用fw,bw
        BufferedWriter bw = null;
        try {
            fw = new FileWriter("d:/1.txt");
            bw = new BufferedWriter(fw);

            bw.write("hello");
        } catch (IOException e) {
            e.printStackTrace();
        } finally{
            try {
                if(bw!=null) {
                    bw.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }finally{
                try {
                    if(fw != null){
                        fw.close();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    @Test
}
```

```

public void test04() {
    FileWriter fw = null;
    BufferedWriter bw = null;
    try {
        fw = new FileWriter("d:/1.txt");
        bw = new BufferedWriter(fw);

        bw.write("hello");
    } catch (IOException e) {
        e.printStackTrace();
    } finally{
        try {
            bw.close(); //如果这句代码关闭时发生异常了，下面fw.close()不执行，关闭可能不彻底
            fw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

```

14.9.2 JDK1.7之后引入新try..catch

语法格式：

```

try(需要关闭的资源对象的声明){
    业务逻辑代码
}catch(异常类型 e){
    处理异常代码
}catch(异常类型 e){
    处理异常代码
}
...

```

它没有finally，也不需要程序员去关闭资源对象，无论是否发生异常，都会关闭资源对象。

需要指出的是，为了保证try语句可以正常关闭资源，这些资源实现类必须实现AutoCloseable或Closeable接口，实现这两个接口就必须实现close方法。Closeable是AutoCloseable的子接口。Java7几乎把所有的“资源类”（包括文件IO的各种类、JDBC编程的Connection、Statement等接口...）进行了改写，改写后资源类都是实现了AutoCloseable或Closeable接口，并实现了close方法。

写到try()中的资源类的变量默认是final声明的，不能修改。

示例代码：

```

@Test
public void test03() {
    //从d:/1.txt(GBK)文件中，读取内容，写到项目根目录下1.txt(UTF-8)文件中
    try(
        FileInputStream fis = new FileInputStream("d:/1.txt");
        InputStreamReader isr = new InputStreamReader(fis,"GBK");

```

```

        BufferedReader br = new BufferedReader(isr);

        FileOutputStream fos = new FileOutputStream("1.txt");
        OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");
        BufferedWriter bw = new BufferedWriter(osw);
    }

    String str;
    while((str = br.readLine()) != null){
        bw.write(str);
        bw.newLine();
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}

```

第15章 网络编程

15.1 软件结构

- **C/S结构**：全称为Client/Server结构，是指客户端和服务器结构。常见程序有QQ、红蜘蛛、飞秋等软件。

B/S结构：全称为Browser/Server结构，是指浏览器和服务器结构。常见浏览器有IE、谷歌、火狐等。

两种架构各有优势，但是无论哪种架构，都离不开网络的支持。**网络编程**，就是在一定的协议下，实现两台计算机的通信的程序。

15.2 网络编程三要素

15.2.1 IP地址和域名

1、IP地址

IP地址：指互联网协议地址（Internet Protocol Address），俗称IP。IP地址用来给一个网络中的计算机设备做唯一的编号。假如我们把“个人电脑”比作“一台电话”的话，那么“IP地址”就相当于“电话号码”。

IP地址分类方式一：

- IPv4：是一个32位的二进制数，通常被分为4个字节，表示成 a.b.c.d 的形式，例如 192.168.65.100。其中 a、b、c、d都是0~255之间的十进制整数，那么最多可以表示42亿个。
- IPv6：由于互联网的蓬勃发展，IP地址的需求量愈来愈大，但是网络地址资源有限，使得IP的分配越发紧张。

为了扩大地址空间，拟通过IPv6重新定义地址空间，采用128位地址长度，每16个字节一组，分成8组十六进制数，表示成 ABCD:EF01:2345:6789:ABCD:EF01:2345:6789，号称可以为全世界的每一粒沙子编上一个网址，这样就解决了网络地址资源数量不够的问题。IPv4和IPv6地址格式不相同，因此在很长一段时间里，互联网中出现IPv4和IPv6长期共存的局面。2012年6月6日，国际互联网协会举行了世界IPv6启动纪念日，这一天，全球IPv6网络正式启动。多家知名网站，如Google、Facebook和Yahoo等，于当天全球标准时间0点（北京时间8点

整)开始永久性支持IPv6访问。2018年6月，三大运营商联合阿里云宣布，将全面对外提供IPv6服务，并计划在2025年前助推中国互联网真正实现“IPv6 Only”。7月，百度云制定了中国的IPv6改造方案。8月3日，工信部通信司在北京召开IPv6规模部署及专项督查工作全国电视电话会议，中国将分阶段有序推进规模建设IPv6网络，实现下一代互联网在经济社会各领域深度融合。

IP地址分类方式二：

公网地址(万维网使用)和私有地址(局域网使用)。192.168.开头的就是私有地址，范围即为192.168.0.0-192.168.255.255，专门为组织机构内部使用

常用命令：

- 查看本机IP地址，在控制台输入：

```
ipconfig
```

- 检查网络是否连通，在控制台输入：

```
ping 空格 IP地址  
ping 220.181.57.216
```

特殊的IP地址：

- 本地回环地址(hostAddress): 127.0.0.1
- 主机名(hostName): localhost

2、域名

因为IP地址数字不便于记忆，因此出现了域名，域名容易记忆，当在连接网络时输入一个主机的域名后，域名服务器(DNS)负责将域名转化成IP地址，这样才能和主机建立连接。----- 域名解析



- 在浏览器中输入www.qq.com域名，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。
- 如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。
- 如果hosts与本地DNS解析器缓存都没有相应的网址映射关系，首先会找TCP/ip参数中设置的首选DNS服务器，在此我们叫它本地DNS服务器，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。
- 如果要查询的域名，不由本地DNS服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。
- 如果本地DNS服务器本地区域文件与缓存解析都失效，则根据本地DNS服务器的设置(是否设置转发器)进行查询，如果未用转发模式，本地DNS就把请求发至13台根DNS，根DNS服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址(<http://qq.com>)给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找(<http://qq.com>)域服务器，重复上面的动作，进行查询，直至找到www.qq.com主机。
- 如果用的是转发模式，此DNS服务器就会把请求转发至上一级DNS服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地DNS服务器用是是转发，还是根提示，最后都是把结果返回给本地DNS服务器，由此DNS服务器再返回给客户机。

15.2.2 端口号

网络的通信，本质上是两个进程（应用程序）的通信。每台计算机都有很多的进程，那么在网络通信时，如何区分这些进程呢？

如果说**IP地址**可以唯一标识网络中的设备，那么**端口号**就可以唯一标识设备中的进程（应用程序）了。

- **端口号：用两个字节表示的整数，它的取值范围是0~65535。**
 - 公认端口：0~1023。被预先定义的服务通信占用，如：HTTP（80），FTP（21），Telnet（23）
 - 注册端口：1024~49151。分配给用户进程或应用程序。如：Tomcat（8080），MySQL（3306），Oracle（1521）。
 - 动态/私有端口：49152~65535。

如果端口号被另外一个服务或应用所占用，会导致当前程序启动失败。

15.2.3 网络通信协议

- **网络通信协议：**通过计算机网络可以使多台计算机实现连接，位于同一个网络中的计算机在进行连接和通信时需要遵守一定的规则，这就好比在道路中行驶的汽车一定要遵守交通规则一样。在计算机网络中，这些连接和通信的规则被称为网络通信协议，它对数据的传输格式、传输速率、传输步骤等做了统一规定，通信双方必须同时遵守才能完成数据交换。
- **TCP/IP协议：**传输控制协议/因特网互联协议(Transmission Control Protocol/Internet Protocol)，是Internet最基本、最广泛的协议。它定义了计算机如何连入因特网，以及数据如何在它们之间传输的标准。它的内部包含一系列的用于处理数据通信的协议，并采用了4层的分层模型，每一层都呼叫它的下一层所提供的协议来完成自己的需求。



上图中，OSI参考模型：模型过于理想化，未能在因特网上进行广泛推广。TCP/IP参考模型(或TCP/IP协议)：事实上的国际标准。

- TCP/IP协议中的四层分别是应用层、传输层、网络层和链路层，每层分别负责不同的通信功能。链路层：链路层是用于定义物理传输通道，通常是对某些网络连接设备的驱动协议，例如针对光纤、网线提供的驱动。
- 网络层：网络层是整个TCP/IP协议的核心，它主要用于将传输的数据进行分组，将分组数据发送到目标计算机或者网络。而IP协议是一种非常重要的协议。IP (internet protocol) 又称为互联网协议。IP的责任就是把数据从源传送到目的地。它在源地址和目的地址之间传送一种称之为数据包的东西，它还提供对数据大小的重新组装功能，以适应不同网络对包大小的要求。
- 传输层：主要使网络程序进行通信，在进行网络通信时，可以采用TCP协议，也可以采用UDP协议。
TCP (Transmission Control Protocol) 协议，即传输控制协议，是一种面向连接的、可靠的、基于字节流的传输层通信协议。UDP(User Datagram Protocol, 用户数据报协议)：是一个无连接的传输层协议、提供面向事务的简单不可靠的信息传送服务。
- 应用层：主要负责应用程序的协议，例如HTTP协议、FTP协议、SNMP (简单网络管理协议)、SMTP (简单邮件传输协议) 和POP3 (Post Office Protocol 3的简称,即邮局协议的第3个版) 等。

而通常我们说的TCP/IP协议，其实是指TCP/IP协议族，因为该协议家族的两个最核心协议：TCP（传输控制协议）和IP（网际协议），为该家族中最早通过的标准，所以简称为TCP/IP协议。

15.3 TCP与UDP协议

通信的协议还是比较复杂的，`java.net` 包中包含的类和接口，它们提供低层次的通信细节。我们可以直接使用这些类和接口，来专注于网络程序开发，而不用考虑通信的细节。

`java.net` 包中提供了两种常见的网络协议的支持：

- **UDP**: 用户数据报协议(User Datagram Protocol)。
- **TCP**: 传输控制协议 (Transmission Control Protocol)。

15.3.1 UDP协议

UDP: 用户数据报协议(User Datagram Protocol), 它是**非面向连的，不可靠的**无连接通信协议，即在数据传输时，数据的发送端和接收端不建立逻辑连接。简单来说，当一台计算机向另外一台计算机发送数据时，发送端不会确认接收端是否存在，就会发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。

由于使用UDP协议消耗资源小，通信效率高，所以通常都会用于音频、视频和普通数据的传输例如视频会议都使用UDP协议，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。

但是在使用UDP协议传送数据时，由于UDP的面向无连接性，不能保证数据的完整性，因此在传输重要数据时不建议使用UDP协议。

- **大小限制的**: 数据被限制在64kb以内，超出这个范围就不能发送了。
- **数据报(Datagram)**: 网络传输的基本单位

15.3.2 TCP协议

TCP: 传输控制协议 (Transmission Control Protocol)。它是**面向连接的，可靠的**通信协议，即传输数据之前，在发送端和接收端建立逻辑连接，然后再传输数据，它提供了两台计算机之间可靠无差错的数据传输。是一种面向连接的、可靠的、基于字节流的传输层的通信协议，可以连续传输大量的数据。类似于打电话的效果。

这是因为它为当一台计算机需要与另一台远程计算机连接时，TCP协议会采用“三次握手”方式让它们建立一个连接，用于发送和接收数据的虚拟链路。数据传输完毕TCP协议会采用“四次挥手”方式断开连接。

TCP协议负责收集这些数据信息包，并将其按适当的次序放好传送，在接收端收到后再将其正确的还原。TCP协议保证了数据包在传送中准确无误。TCP协议使用重发机制，当一个通信实体发送一个消息给另一个通信实体后，需要收到另一个通信实体确认信息，如果没有收到另一个通信实体确认信息，则会再次重复刚才发送的消息。

1、三次握手

TCP协议中，在发送数据的准备阶段，客户端与服务器之间的三次交互，以保证连接的可靠。

- 第一次握手，客户端向服务器端发出连接请求，等待服务器确认。
- 第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求。
- 第三次握手，客户端再次向服务器端发送确认信息，确认连接。



完成三次握手，连接建立后，客户端和服务器就可以开始进行数据传输了。由于这种面向连接的特性，TCP协议可以保证传输数据的安全，所以应用十分广泛，例如下载文件、浏览网页等。

2、四次挥手

TCP协议中，在发送数据结束后，释放连接时需要经过四次挥手。

- 第一次挥手：客户端向服务器端提出结束连接，让服务器做最后的准备工作。此时，客户端处于半关闭状态，即表示不再向服务器发送数据了，但是还可以接受数据。
- 第二次挥手：服务器接收到客户端释放连接的请求后，会将最后的数据发给客户端。并告知上层的应用进程不再接收数据。

- 第三次挥手：服务器发送完数据后，会给客户端发送一个释放连接的报文。那么客户端接收后就知道可以正式释放连接了。
- 第四次挥手：客户端接收到服务器最后的释放连接报文后，要回复一个彻底断开的报文。这样服务器收到后才会彻底释放连接。这里客户端，发送完最后的报文后，会等待2MSL，因为有可能服务器没有收到最后的报文，那么服务器迟迟没收到，就会再次给客户端发送释放连接的报文，此时客户端在等待时间范围内接收到，会重新发送最后的报文，并重新计时。如果等待2MSL后，没有收到，那么彻底断开。



15.4 网络编程API

15.4.1 InetAddress类

InetAddress类主要表示IP地址，两个子类：Inet4Address、Inet6Address。

Internet上的主机有两种方式表示地址：

- 域名(hostName): www.atguigu.com
- IP地址(hostAddress): 202.108.35.210

IInetAddress类没有提供公共的构造器，而是提供了如下几个静态方法来获取InetAddress实例

- public static InetAddress getLocalHost()
- public static InetAddress getByAddress(String host)
- public static InetAddress getByName(byte[] addr)

InetAddress提供了如下几个常用的方法

- public String getHostAddress()：返回IP地址字符串（以文本表现形式）。
- public String getHostName()：获取此IP地址的主机名

```
package com.atguigu.ip;

import java.net.InetAddress;
import java.net.UnknownHostException;

import org.junit.Test;

public class TestInetAddress {
    @Test
    public void test01() throws UnknownHostException{
        InetAddress localHost = InetAddress.getLocalHost();
        System.out.println(localHost);
    }

    @Test
    public void test02() throws UnknownHostException{
        InetAddress atguigu = InetAddress.getByName("www.atguigu.com");
        System.out.println(atguigu);
    }

    @Test
    public void test03() throws UnknownHostException{
```

```
//           byte[] addr = {112,54,108,98};  
byte[] addr = {(byte)192,(byte)168,24,56};  
InetAddress atguigu = InetAddress.getByAddress(addr);  
System.out.println(atguigu);  
}  
}
```



15.4.2 Socket分类

通信的两端都要有Socket（也可以叫“套接字”），是两台机器间通信的端点。网络通信其实就是Socket间的通信。Socket可以分为：

- 流套接字（stream socket）：使用TCP提供可依赖的字节流服务
 - ServerSocket：此类实现TCP服务器套接字。服务器套接字等待请求通过网络传入。
 - Socket：此类实现客户端套接字（也可以就叫“套接字”）。套接字是两台机器间通信的端点。
- 数据报套接字（datagram socket）：使用UDP提供“尽力而为”的数据报服务
 - DatagramSocket：此类表示用来发送和接收UDP数据报包的套接字。



15.4.3 Socket相关类API

1、ServerSocket类

ServerSocket类的构造方法：

- ServerSocket(int port)：创建绑定到特定端口的服务器套接字。

ServerSocket类的常用方法：

- Socket accept()：侦听并接受到此套接字的连接。

2、Sokcet类

Socket类的常用构造方法：

- public Socket(InetAddress address,int port)：创建一个流套接字并将其连接到指定IP地址的指定端口号。
- public Socket(String host,int port)：创建一个流套接字并将其连接到指定主机上的指定端口号。

Socket类的常用方法：

- public InputStream getInputStream()：返回此套接字的输入流，可以用于接收消息
- public OutputStream getOutputStream()：返回此套接字的输出流，可以用于发送消息
- public InetAddress getInetAddress()：此套接字连接到的远程IP地址；如果套接字是未连接的，则返回null。
- public InetAddress getLocalAddress()：获取套接字绑定的本地地址。
- public int getPort()：此套接字连接到的远程端口号；如果尚未连接套接字，则返回0。
- public int getLocalPort()：返回此套接字绑定到的本地端口。如果尚未绑定套接字，则返回-1。
- public void close()：关闭此套接字。套接字被关闭后，便不可在以后的网络连接中使用（即无法重新连接或重新绑定）。需要创建新的套接字对象。关闭此套接字也将关闭该套接字的InputStream和OutputStream。

- public void shutdownInput(): 如果在套接字上调用 shutdownInput() 后从套接字输入流读取内容，则流将返回 EOF (文件结束符)。即不能在此套接字的输入流中接收任何数据。
- public void shutdownOutput(): 禁用此套接字的输出流。对于 TCP 套接字，任何以前写入的数据都将被发送，并且后跟 TCP 的正常连接终止序列。如果在套接字上调用 shutdownOutput() 后写入套接字输出流，则该流将抛出 IOException。即不能通过此套接字的输出流发送任何数据。

注意：先后调用Socket的shutdownInput()和shutdownOutput()方法，仅仅关闭了输入流和输出流，并不等于调用Socket的close()方法。在通信结束后，仍然要调用Socket的close()方法，因为只有该方法才会释放Socket占用的资源，比如占用的本地端口号等。

3、DatagramSocket

DatagramSocket 类的常用方法：

- public DatagramSocket(int port) 创建数据报套接字并将其绑定到本地主机上的指定端口。套接字将被绑定到通配符地址，IP 地址由内核来选择。
- public DatagramSocket(int port,InetAddress laddr) 创建数据报套接字，将其绑定到指定的本地地址。本地端口必须在 0 到 65535 之间（包括两者）。如果 IP 地址为 0.0.0.0，套接字将被绑定到通配符地址，IP 地址由内核选择。
- public void close() 关闭此数据报套接字。
- public void send(DatagramPacket p) 从此套接字发送数据报包。DatagramPacket 包含的信息指示：将要发送的数据、其长度、远程主机的 IP 地址和远程主机的端口号。
- public void receive(DatagramPacket p) 从此套接字接收数据报包。当此方法返回时，DatagramPacket 的缓冲区填充了接收的数据。数据报包也包含发送方的 IP 地址和发送方机器上的端口号。此方法在接收到数据报前一直阻塞。数据报包对象的 length 字段包含所接收信息的长度。如果信息比包的长度长，该信息将被截短。

4、DatagramPacket类

DatagramPacket类的常用方法：

- public DatagramPacket(byte[] buf,int length) 构造 DatagramPacket，用来接收长度为 length 的数据包。length 参数必须小于等于 buf.length。
- public DatagramPacket(byte[] buf,int length,InetAddress address,int port) 构造数据报包，用来将长度为 length 的包发送到指定主机上的指定端口号。length 参数必须小于等于 buf.length。
- public int getLength() 返回将要发送或接收到的数据的长度。

15.5 TCP网络编程

15.5.1 通信模型

Java语言的基于套接字TCP编程分为服务端编程和客户端编程，其通信模型如图所示：



15.5.2 开发步骤

服务器端程序包含以下四个基本的 步骤：

- 调用 ServerSocket(int port)：创建一个服务器端套接字，并绑定到指定端口上。用于监听客户端的请求。
- 调用 accept()：监听连接请求，如果客户端请求连接，则接受连接，返回通信套接字对象。
- 调用 该Socket类对象的getOutputStream() 和getInputStream()：获取输出流和输入流，开始网络数据的发送和接收。

- 关闭Socket 对象：客户端访问结束，关闭通信套接字。

客户端程序包含以下四个基本的步骤：

- 创建 Socket：根据指定服务端的 IP 地址或端口号构造 Socket 类对象。若服务器端响应，则建立客户端到服务器的通信线路。若连接失败，会出现异常。
- 打开连接到 Socket 的输入/ 出流：使用 getInputStream()方法获得输入流，使用getOutputStream()方法获得输出流，进行数据传输
- 按照一定的协议对 Socket 进行读/ 写操作：通过输入流读取服务器放入线路的信息（但不能读取自己放入线路的信息），通过输出流将信息写入线路。
- 关闭 Socket：断开客户端到服务器的连接，释放线路

15.5.3 演示单个客户端与服务器单次通信

需求：客户端连接服务器，连接成功后给服务发送“lalala”，服务器收到消息后，给客户端返回“欢迎登录”，客户端接收消息后，断开连接

1、服务器端示例代码

```
package com.atguigu.tcp.one;

import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {
    public static void main(String[] args) throws Exception {
        //1、准备一个ServerSocket对象，并绑定8888端口
        ServerSocket server = new ServerSocket(8888);
        System.out.println("等待连接....");

        //2、在8888端口监听客户端的连接，该方法是个阻塞的方法，如果没有客户端连接，将一直等待
        Socket socket = server.accept();
        InetAddress inetAddress = socket.getInetAddress();
        System.out.println(inetAddress.getHostAddress() + "客户端连接成功！！");

        //3、获取输入流，用来接收该客户端发送给服务器的数据
        InputStream input = socket.getInputStream();
        //接收数据
        byte[] data = new byte[1024];
        StringBuilder s = new StringBuilder();
        int len;
        while ((len = input.read(data)) != -1) {
            s.append(new String(data, 0, len));
        }
        System.out.println(inetAddress.getHostAddress() + "客户端发送的消息是：" + s);

        //4、获取输出流，用来发送数据给该客户端
        OutputStream out = socket.getOutputStream();
        //发送数据
    }
}
```

```

        out.write("欢迎登录".getBytes());
        out.flush();

        //5、关闭socket，不再与该客户端通信
        //socket关闭，意味着InputStream和OutputStream也关闭了
        socket.close();

        //6、如果不再接收任何客户端通信，可以关闭serverSocket
        server.close();
    }
}

```

2、客户端示例代码

```

package com.atguigu.tcp.one;

import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class Client {

    public static void main(String[] args) throws Exception {
        // 1、准备Socket，连接服务器，需要指定服务器的IP地址和端口号
        Socket socket = new Socket("127.0.0.1", 8888);

        // 2、获取输出流，用来发送数据给服务器
        OutputStream out = socket.getOutputStream();
        // 发送数据
        out.write("lalala".getBytes());
        //会在流末尾写入一个“流的末尾”标记，对方才能读到-1，否则对方的读取方法会一致阻塞
        socket.shutdownOutput();

        //3、获取输入流，用来接收服务器发送给该客户端的数据
        InputStream input = socket.getInputStream();
        // 接收数据
        byte[] data = new byte[1024];
        StringBuilder s = new StringBuilder();
        int len;
        while ((len = input.read(data)) != -1) {
            s.append(new String(data, 0, len));
        }
        System.out.println("服务器返回的消息是：" + s);

        //4、关闭socket，不再与服务器通信，即断开与服务器的连接
        //socket关闭，意味着InputStream和OutputStream也关闭了
        socket.close();
    }
}

```

15.5.4 演示多个客户端与服务器之间的多次通信

通常情况下，服务器不应该只接受一个客户端请求，而应该不断地接受来自客户端的所有请求，所以Java程序通常会通过循环，不断地调用ServerSocket的accept()方法。

如果服务器端要“同时”处理多个客户端的请求，因此服务器端需要为每一个客户端单独分配一个线程来处理，否则无法实现“同时”。

咱们之前学习IO流的时候，提到过装饰者设计模式，该设计使得不管底层IO流是怎样的节点流：文件流也好，网络Socket产生的流也好，程序都可以将其包装成处理流，甚至可以多层包装，从而提供更多方便的处理。

案例需求：多个客户端连接服务器，并进行多次通信

- 每一个客户端连接成功后，从键盘输入英文单词或中国成语，并发送给服务器
- 服务器收到客户端的消息后，把词语“反转”后返回给客户端
- 客户端接收服务器返回的“词语”，打印显示
- 当客户端输入“stop”时断开与服务器的连接
- 多个客户端可以同时给服务器发送“词语”，服务器可以“同时”处理多个客户端的请求

1564027041074

1、服务器端示例代码

```
package com.atguigu.tcp.many;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {
    public static void main(String[] args) throws IOException {
        // 1、准备一个ServerSocket
        ServerSocket server = new ServerSocket(8888);
        System.out.println("等待连接...");

        int count = 0;
        while(true){
            // 2、监听一个客户端的连接
            Socket socket = server.accept();
            System.out.println("第" + ++count + "个客户
端"+socket.getInetAddress().getHostAddress()+"连接成功！！");
            ClientHandlerThread ct = new ClientHandlerThread(socket);
            ct.start();
        }
        //这里没有关闭server，永远监听
    }
    static class ClientHandlerThread extends Thread{
```

```

private Socket socket;
private String ip;

public ClientHandlerThread(Socket socket) {
    super();
    this.socket = socket;
    ip = socket.getInetAddress().getHostAddress();
}

public void run(){
    try{
        // (1) 获取输入流，用来接收该客户端发送给服务器的数据
        BufferedReader br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

        // (2) 获取输出流，用来发送数据给该客户端
        PrintStream ps = new PrintStream(socket.getOutputStream());
        String str;
        // (3) 接收数据
        while ((str = br.readLine()) != null) {
            // (4) 反转
            StringBuilder word = new StringBuilder(str);
            word.reverse();

            // (5) 返回给客户端
            ps.println(word);
        }
        System.out.println("客户端" + ip+"正常退出");
    }catch(Exception e){
        System.out.println("客户端" + ip+"意外退出");
    }finally{
        try {
            // (6) 断开连接
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

2、客户端示例代码

```

package com.atguigu.tcp.many;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.util.Scanner;

```

```
public class Client {
    public static void main(String[] args) throws Exception {
        // 1、准备Socket，连接服务器，需要指定服务器的IP地址和端口号
        Socket socket = new Socket("127.0.0.1", 8888);

        // 2、获取输出流，用来发送数据给服务器
        OutputStream out = socket.getOutputStream();
        PrintStream ps = new PrintStream(out);

        // 3、获取输入流，用来接收服务器发送给该客户端的数据
        InputStream input = socket.getInputStream();
        BufferedReader br;
        if(args!= null && args.length>0) {
            String encoding = args[0];
            br = new BufferedReader(new InputStreamReader(input,encoding));
        }else{
            br = new BufferedReader(new InputStreamReader(input));
        }

        Scanner scanner = new Scanner(System.in);
        while(true){
            System.out.println("输入发送给服务器的单词或成语：");
            String message = scanner.nextLine();
            if(message.equals("stop")){
                socket.shutdownOutput();
                break;
            }

            // 4、发送数据
            ps.println(message);
            // 接收数据
            String feedback = br.readLine();
            System.out.println("从服务器收到的反馈是：" + feedback);
        }

        //5、关闭socket，断开与服务器的连接
        scanner.close();
        socket.close();
    }
}
```

15.6 UDP网络编程

UDP(User Datagram Protocol，用户数据报协议)：是一个无连接的传输层协议、提供面向事务的简单不可靠的信息传送服务，类似于短信。

15.6.1 通信模型

UDP协议是一种**面向非连接**的协议，面向非连接指的是在正式通信前不必与对方先建立连接，不管对方状态就直接发送，至于对方是否可以接收到这些数据内容，UDP协议无法控制，因此说，UDP协议是一种**不可靠的协议**。无连接的好处就是快，省内存空间和流量，因为维护连接需要创建大量的数据结构。UDP会尽最大努力交付数据，但不保证可靠交付，没有TCP的确认机制、重传机制，如果因为网络原因没有传送到对端，UDP也不会给应用层返回错误信息。

UDP协议是面向数据报文的信息传送服务。UDP在发送端没有缓冲区，对于应用层交付下来的报文在添加了首部之后就直接交付于ip层，不会进行合并，也不会进行拆分，而是一次交付一个完整的报文。比如我们要发送100个字节的报文，我们调用一次send()方法就会发送100字节，接收方也需要用receive()方法一次性接收100字节，不能使用循环每次获取10个字节，获取十次这样的做法。

UDP协议没有拥塞控制，所以当网络出现的拥塞不会导致主机发送数据的速率降低。虽然UDP的接收端有缓冲区，但是这个缓冲区只负责接收，并不会保证UDP报文的到达顺序是否和发送的顺序一致。因为网络传输的时候，由于网络拥塞的存在是很大的可能导致先发的报文比后发的报文晚到达。如果此时缓冲区满了，后面到达的报文将直接被丢弃。这个对实时应用来说很重要，比如：视频通话、直播等应用。

因此UDP适用于一次只传送少量数据、对可靠性要求不高的应用环境，数据报大小限制在64K以下。



15.6.2 开发步骤

发送端程序包含以下四个基本的 步骤：

- 创建DatagramSocket：默认使用系统随机分配端口号。
- 创建DatagramPacket：将要发送的数据用字节数组表示，并指定要发送的数据长度，接收方的IP地址和端口号。
- 调用 该DatagramSocket 类对象的 send方法：发送数据报DatagramPacket对象。
- 关闭DatagramSocket 对象：发送端程序结束，关闭通信套接字。

接收端程序包含以下四个基本的步骤：

- 创建DatagramSocket：指定监听的端口号。
- 创建DatagramPacket：指定接收数据用的字节数组，起到临时数据缓冲区的效果，并指定最大可以接收的数据长度。
- 调用 该DatagramSocket 类对象的receive方法：接收数据报DatagramPacket对象。。
- 关闭DatagramSocket：接收端程序结束，关闭通信套接字。

15.6.3 演示发送和接收消息

基于UDP协议的网络编程仍然需要在通信实例的两端各建立一个Socket，但这两个Socket之间并没有虚拟链路，这两个Socket只是发送、接收数据报的对象，Java提供了DatagramSocket对象作为基于UDP协议的Socket，使用DatagramPacket代表DatagramSocket发送、接收的数据报。

1、发送端示例代码

```
package com.atguigu.udp;

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.ArrayList;

public class Send {
```

```

public static void main(String[] args) throws Exception {
//    1、建立发送端的DatagramSocket
    DatagramSocket ds = new DatagramSocket();

    //要发送的数据
    ArrayList<String> all = new ArrayList<String>();
    all.add("尚硅谷让天下没有难学的技术! ");
    all.add("学高端前沿的IT技术来尚硅谷! ");
    all.add("尚硅谷让你的梦想变得更具体! ");
    all.add("尚硅谷让你的努力更有价值! ");

    //接收方的IP地址
    InetAddress ip = InetAddress.getByName("127.0.0.1");
    //接收方的监听端口号
    int port = 9999;
    //发送多个数据报
    for (int i = 0; i < all.size(); i++) {
//        2、建立数据包DatagramPacket
        byte[] data = all.get(i).getBytes();
        DatagramPacket dp = new DatagramPacket(data, 0, data.length, ip, port);
//        3、调用Socket的发送方法
        ds.send(dp);
    }

//        4、关闭socket
    ds.close();
}
}

```

2、接收端示例代码

```

package com.atguigu.udp;

import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class Receive {

    public static void main(String[] args) throws Exception {
//        1、建立接收端的DatagramSocket，需要指定本端的监听端口号
        DatagramSocket ds = new DatagramSocket(9999);

        //一直监听数据
        while(true){
            //2、建立数据包DatagramPacket
            byte[] buffer = new byte[1024*64];
            DatagramPacket dp = new DatagramPacket(buffer , buffer.length);

            //3、调用Socket的接收方法
            ds.receive(dp);

            //4、拆封数据
        }
    }
}

```

```
        String str = new String(dp.getData(), 0, dp.getLength());
        System.out.println(str);
    }

//      ds.close();
}
}
```

第16章 反射 (Reflect)

16.1 反射的概念

Java程序中，所有的对象都有两种类型：编译时类型和运行时类型，而很多时候对象的编译时类型和运行时类型不一致。

例如：某些变量或形参的类型是Object类型，但是程序却需要调用该对象运行时类型的方法，该方法不是Object中的方法，那么如何解决呢？

为了解决这些问题，程序需要在运行时发现对象和类的真实信息，现在有两种方案：

方案1：在编译和运行时都完全知道类型的具体信息，在这种情况下，我们可以直接先使用`instanceof`运算符进行判断，再利用强制类型转换符将其转换成运行时类型的变量即可。

方案2：编译时根本无法预知该对象和类的真实信息，程序只能依靠运行时信息来发现该对象和类的真实信息，这就必须使用反射。

因为加载完类之后，就产生了一个Class类型的对象，并将引用存储到方法区，那么每一个类在方法区内存都可以找到唯一Class对象与之对应，这个对象包含了完整的类的结构信息，我们可以通过这个对象获取类的结构。这种机制就像一面镜子，Class对象像是类在镜子中的镜像，通过观察这个镜像就可以知道类的结构，所以，把这种机制形象地称为反射机制。

非反射：类（原物）-->类信息

反射：Class对象（镜像）-->类（原物）



16.2 java.lang.Class类

要想解剖一个类，必须先要获取到该类的Class对象。而剖析一个类或用反射解决具体的问题就是使用相关API (1) `java.lang.Class` (2) `java.lang.reflect.*`。所以，Class对象是反射的根源。

16.2.1 哪些类型可以获取Class对象

哪些类型可以获取Class对象？所有Java类型

```
// (1) 基本数据类型和void  
例如: int.class  
      void.class  
// (2) 类和接口  
例如: String.class  
      Comparable.class  
// (3) 枚举  
例如: ElementType.class  
// (4) 注解  
例如: override.class  
// (5) 数组  
例如: int[].class
```

示例代码：

```
package com.atguigu.classtype;  
  
import java.lang.annotation.ElementType;  
  
public class JavaType {  
    public static void main(String[] args) {  
        // (1) 基本数据类型和void  
        Class c1 = int.class;  
        Class c2 = void.class;  
  
        System.out.println("c1 = " + c1);  
        System.out.println("c2 = " + c2);  
        // (2) 类和接口  
        Class c3 = String.class;  
        Class c4 = Comparable.class;  
        System.out.println("c3 = " + c3);  
        System.out.println("c4 = " + c4);  
  
        // (3) 枚举  
        Class c5 = ElementType.class;  
        System.out.println("c5 = " + c5);  
  
        // (4) 注解  
        Class c6 = Override.class;  
        System.out.println("c6 = " + c6);  
  
        // (5) 数组  
        Class c7 = int[].class;  
        Class c9 = String[].class;  
        Class c8 = int[][].class;  
        System.out.println("c7 = " + c7);  
        System.out.println("c8 = " + c8);  
        System.out.println("c9 = " + c9);  
    }  
}
```

16.2.2 获取Class对象的四种方式

(1) 类型名.class

要求编译期间已知类型

(2) 对象.getClass()

获取对象的运行时类型

(3) Class.forName(类型全名称)

可以获取编译期间未知的类型

(4) ClassLoader的类加载器对象.loadClass(类型全名称)

可以用系统类加载对象或自定义加载器对象加载指定路径下的类型

```
package com.atguigu.classtype;

import org.junit.Test;

public class GetClassObject {
    @Test
    public void test01() throws ClassNotFoundException{
        Class c1 = GetClassObject.class;
        GetClassObject obj = new GetClassObject();
        Class c2 = obj.getClass();
        Class c3 = Class.forName("com.atguigu.classtype.GetClassObject");
        Class c4 =
            ClassLoader.getSystemClassLoader().loadClass("com.atguigu.classtype.GetClassObject");

        System.out.println("c1 = " + c1);
        System.out.println("c2 = " + c2);
        System.out.println("c3 = " + c3);
        System.out.println("c4 = " + c4);

        System.out.println(c1 == c2);
        System.out.println(c1 == c3);
        System.out.println(c1 == c4);
    }

    @Test
    public void test02(){
        int[] arr1 = {1,2,3,4,5};
        int[] arr2 = {10,34,5,66,34,22};
        int[][] arr3 = {{1,2,3,4},{4,5,6,7}};
        String[] arr4 = {"Hello","world"};

        Class c1 = arr1.getClass();
        Class c2 = arr2.getClass();
        Class c3 = arr3.getClass();
        Class c4 = arr4.getClass();

        System.out.println("c1 = " + c1);
    }
}
```

```

        System.out.println("c2 = " + c2);
        System.out.println("c3 = " + c3);
        System.out.println("c4 = " + c4);
        System.out.println(c1 == c2);
        System.out.println(c1 == c3);
        System.out.println(c1 == c4);
        System.out.println(c3 == c4);
    }
}

```

16.3 类加载

类在内存中完整的生命周期：加载-->使用-->卸载

16.3.1 类的加载过程

当程序主动使用某个类时，如果该类还未被加载到内存中，系统会通过加载、连接、初始化三个步骤来对该类进行初始化，如果没有意外，JVM将会连续完成这三个步骤，所以有时也把这三个步骤统称为类加载。

类的加载又分为三个阶段：

(1) 加载: load

就是指将类型的class字节码数据读入内存

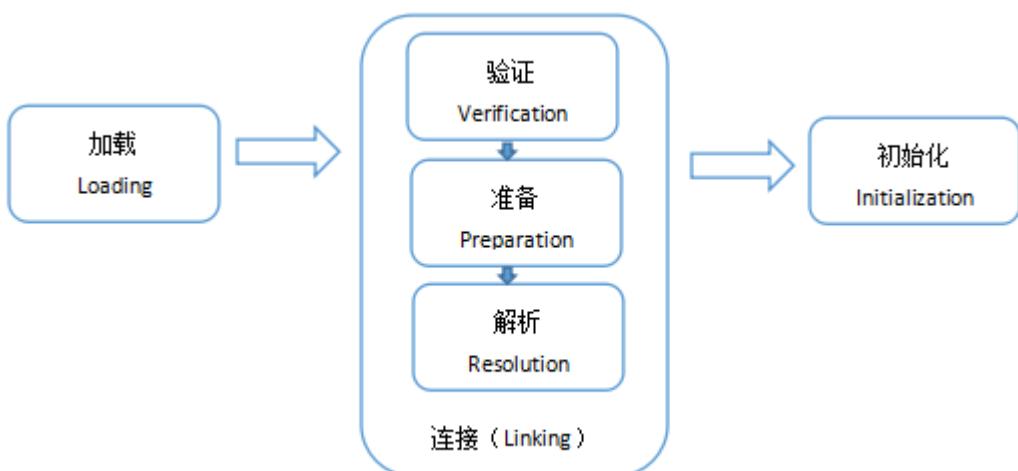
(2) 连接: link

①验证：校验合法性等

②准备：准备对应的内存（方法区），创建Class对象，为类变量赋默认值，为静态常量赋初始值。

③解析：把字节码中的符号引用替换为对应的直接地址引用

(3) 初始化: initialize (类初始化) 即执行类初始化方法，大多数情况下，类的加载就完成了类的初始化，有些情况下，会延迟类的初始化。



16.3.2 类初始化

1、哪些操作会导致类的初始化？

- (1) 运行主方法所在的类，要先完成类初始化，再执行main方法
- (2) 第一次使用某个类型就是在new它的对象，此时这个类没有初始化的话，先完成类初始化再做实例初始化
- (3) 调用某个类的静态成员（类变量和类方法），此时这个类没有初始化的话，先完成类初始化
- (4) 子类初始化时，发现它的父类还没有初始化的话，那么先初始化父类
- (5) 通过反射操作某个类时，如果这个类没有初始化，也会导致该类先初始化

类初始化执行的是()，该方法由 (1) 类变量的显式赋值代码 (2) 静态代码块中的代码构成

```
package com.atguigu.init;

class Father{
    static{
        System.out.println("main方法所在的类的父类(1)");//初始化子类时，会初始化父类
    }
}

public class ClassInitialize extends Father{
    static{
        System.out.println("main方法所在的类(2)");//主方法所在的类会初始化
    }

    public static void main(String[] args) throws ClassNotFoundException {
        new A();//第一次使用A就是创建它的对象，会初始化A类

        B.test();//直接使用B类的静态成员会初始化B类

        Class clazz = Class.forName("com.atguigu.test02.C");//通过反射操作C类，会初始化C类
    }
}

class A{
    static{
        System.out.println("A类初始化");
    }
}

class B{
    static{
        System.out.println("B类初始化");
    }

    public static void test(){
        System.out.println("B类的静态方法");
    }
}

class C{
    static{
        System.out.println("C类初始化");
    }
}
```

2、哪些使用类的操作，但是不会导致类的初始化？

- (1) 使用某个类的静态的常量 (static final)

- (2) 通过子类调用父类的静态变量，静态方法，只会导致父类初始化，不会导致子类初始化，即只有声明静态成员的类才会初始化
- (3) 用某个类型声明数组并创建数组对象时，不会导致这个类初始化

```
public class TestClinit2 {  
    public static void main(String[] args) {  
        System.out.println(D.NUM); //D类不会初始化，因为NUM是final的  
  
        System.out.println(F.num);  
        F.test(); //F类不会初始化，E类会初始化，因为num和test()是在E类中声明的  
  
        //G类不会初始化，此时还没有正式用的G类  
        G[] arr = new G[5]; //没有创建G的对象，创建的是准备用来装G对象的数组对象  
        //G[]是一种新的类型，是数组类型，动态编译生成的一种新的类型  
        //G[].class  
    }  
}  
class D{  
    public static final int NUM = 10;  
    static{  
        System.out.println("D类的初始化");  
    }  
}  
class E{  
    static int num = 10;  
    static{  
        System.out.println("E父类的初始化");  
    }  
    public static void test(){  
        System.out.println("父类的静态方法");  
    }  
}  
class F extends E{  
    static{  
        System.out.println("F子类的初始化");  
    }  
}  
  
class G{  
    static{  
        System.out.println("G类的初始化");  
    }  
}
```

16.3.3 类加载器

很多开发人员都遇到过java.lang.ClassNotFoundException或java.lang.NoClassDefError，想要更好的解决这类问题，或者在一些特殊的应用场景，比如需要支持类的动态加载或需要对编译后的字节码文件进行加密解密操作，那么需要你自定义类加载器，因此了解类加载器及其类加载机制也就成了每一个Java开发人员的必备技能之一。

1、类加载器分为：

(1) 引导类加载器 (Bootstrap Classloader) 又称为根类加载器

它负责加载jre/rt.jar核心库
它本身不是Java代码实现的，也不是ClassLoader的子类，获取它的对象时往往返回null

(2) 扩展类加载器 (Extension ClassLoader)

它负责加载jre/lib/ext扩展库
它是ClassLoader的子类

(3) 应用程序类加载器 (Application Classloader)

它负责加载项目的classpath路径下的类
它是ClassLoader的子类

(4) 自定义类加载器

当你的程序需要加载“特定”目录下的类，可以自定义类加载器；
当你的程序的字节码文件需要加密时，那么往往会提供一个自定义类加载器对其进行解码
后面会见到的自定义类加载器：tomcat中

2、Java系统类加载器的双亲委托模式

简单描述：

下一级的类加载器，如果接到任务时，会先搜索是否加载过，如果没有，会先把任务往上传，如果都没有加载过，一直到根加载器，如果根加载器在它负责的路径下没有找到，会往下传，如果一路回传到最后一级都没有找到，那么会报ClassNotFoundException或NoClassDefError，如果在某一级找到了，就直接返回Class对象。

应用程序类加载器 把 扩展类加载器视为父加载器，

扩展类加载器 把 引导类加载器视为父加载器。

不是继承关系，是组合的方式实现的。

16.3.4 查看某个类的类加载器对象

(1) 获取默认的系统类加载器

```
ClassLoader classLoader = ClassLoader.getSystemClassLoader()
```

(2) 查看某个类是哪个类加载器加载的

```
ClassLoader classObject.getClass().getClassLoader()  
//如果是根加载器加载的类，则会得到null
```

(3) 获取某个类加载器的父加载器

```
ClassLoader classObject.getClass().getParent()
```

示例代码：

```
package com.atguigu.loader;

import org.junit.Test;

public class TestClassLoader {
    @Test
    public void test01(){
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
        System.out.println("systemClassLoader = " + systemClassLoader);
    }

    @Test
    public void test02()throws Exception{
        ClassLoader c1 = String.class.getClassLoader();
        System.out.println("加载String类的类加载器: " + c1);

        ClassLoader c2 =
class.forName("sun.util.resources.cldr.zh.TimeZoneNames_zh").getClassLoader();
        System.out.println("加载sun.util.resources.cldr.zh.TimeZoneNames_zh类的类加载器: " +
c2);

        ClassLoader c3 = TestClassLoader.class.getClassLoader();
        System.out.println("加载当前类的类加载器: " + c3);
    }

    @Test
    public void test03(){
        ClassLoader c1 = TestClassLoader.class.getClassLoader();
        System.out.println("加载当前类的类加载器c1=" + c1);

        ClassLoader c2 = c1.getParent();
        System.out.println("c1.parent = " + c2);

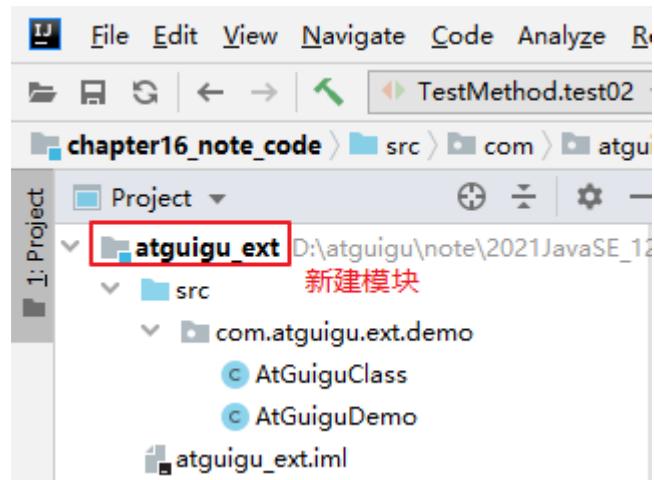
        ClassLoader c3 = c2.getParent();
        System.out.println("c2.parent = " + c3);

    }
}
```

16.3.5 演示导出jar包至jre/lib/ext

演示导出jar包放到“D:\ProgramFiles\Java\jdk1.8.0_271\jre\lib\ext”目录。具体操作步骤请看《尚硅谷-第18章-IDE开发工具Idea使用》第18.13。

新建模块，包含如下类型：



```
package com.atguigu.ext.demo;

public class AtGuiguClass {
    public static void printInfo(String info){
        System.out.println("AtGuiguClass.method");
        System.out.println("info = " + info);
    }

    public int doubleNum(int num){
        return 2 * num;
    }

}
```

```
package com.atguigu.ext.demo;

public class AtGuiguDemo {
    private static String info;
    private String title;
    private int num;

    public AtGuiguDemo(String title, int num) {
        this.title = title;
        this.num = num;
    }

    public static String getInfo() {
        return info;
    }

    public static void setInfo(String info) {
        AtGuiguDemo.info = info;
    }

    public String getTitle() {
        return title;
    }
}
```

```
public void setTitle(String title) {
    this.title = title;
}

public int getNum() {
    return num;
}

public void setNum(int num) {
    this.num = num;
}

@Override
public String toString() {
    return "AtGuiguDemo{" +
        "title='" + title + '\'' +
        ", num=" + num +
        '}';
}
}
```



16.4 反射的基本应用

16.4.1 获取类型的详细信息

可以获取：包、修饰符、类型名、父类（包括泛型父类）、父接口（包括泛型父接口）、成员（属性、构造器、方法）、注解（类上的、方法上的、属性上的）

示例代码获取常规信息：

```
package com.atguigu.reflect;

package com.atguigu.reflect;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class TestClassInfo {
    public static void main(String[] args) throws Exception {
        //1. 先得到某个类型的Class对象
        Class clazz = String.class;
        //比喻clazz好比是镜子中的影子

        //2. 获取类信息
        // (1) 获取包对象，即所有java的包，都是Package的对象
        Package pkg = clazz.getPackage();
```

```

System.out.println("包名: " + pkg.getName());

// (2) 获取修饰符
//其实修饰符是Modifier, 里面有很多常量值
/*
 * 0x是十六进制
 * PUBLIC          = 0x00000001;  1    1
 * PRIVATE         = 0x00000002;  2    10
 * PROTECTED      = 0x00000004;  4    100
 * STATIC          = 0x00000008;  8    1000
 * FINAL           = 0x00000010;  16   10000
 * ...
 *
 * 设计的理念, 就是用二进制的某一位是1, 来代表一种修饰符, 整个二进制中只有一位是1, 其余都是0
 *
 * mod = 17          0x00000011
 * if ((mod & PUBLIC) != 0) 说明修饰符中有public
 * if ((mod & FINAL) != 0) 说明修饰符中有final
 */
int mod = clazz.getModifiers();
System.out.println("类的修饰符有: " + Modifier.toString(mod));

// (3) 类型名
String name = clazz.getName();
System.out.println("类名: " + name);

// (4) 父类, 父类也有父类对应的Class对象
Class superclass = clazz.getSuperclass();
System.out.println("父类: " + superclass);

// (5) 父接口们
System.out.println("父接口们: ");
Class[] interfaces = clazz.getInterfaces();
for (Class iter : interfaces) {
    System.out.println(iter);
}

// (6) 类的属性, 你声明的一个属性, 它是Field的对象
/*
 Field clazz.getField(name) 根据属性名获取一个属性对象, 但是只能得到公共的
 Field[] clazz.getFields(); 获得所有公共的属性
 Field clazz.getDeclaredField(name) 根据属性名获取一个属性对象, 可以获取已声明的
 Field[] clazz.getDeclaredFields()       获得所有已声明的属性
 */
Field valueField = clazz.getDeclaredField("value");
System.out.println("valueField = " + valueField);

System.out.println("-----");
System.out.println("成员如下: ");
System.out.println("属性有: ");
Field[] declaredFields = clazz.getDeclaredFields();
for (Field field : declaredFields) {
    //修饰符、数据类型、属性名
    int modifiers = field.getModifiers();
}

```

```
System.out.println("属性的修饰符: " + Modifier.toString(modifiers));

String name2 = field.getName();
System.out.println("属性名: " + name2);

Class<?> type = field.getType();
System.out.println("属性的数据类型: " + type);
}

System.out.println("-----");
// (7) 构造器们
System.out.println("构造器列表: ");
Constructor[] constructors = clazz.getDeclaredConstructors();
for (int i=0; i<constructors.length; i++) {
    Constructor constructor = constructors[i];
    System.out.println("第" + (i+1) +"个构造器: ");
    //修饰符、构造器名称、构造器形参列表 、抛出异常列表
    int modifiers = constructor.getModifiers();
    System.out.println("构造器的修饰符: " + Modifier.toString(modifiers));

    String name2 = constructor.getName();
    System.out.println("构造器名: " + name2);

    //形参列表
    System.out.println("形参列表: ");
    Class[] parameterTypes = constructor.getParameterTypes();
    for (Class parameterType : parameterTypes) {
        System.out.println(parameterType);
    }
}

//异常列表
System.out.println("异常列表: ");
Class<?>[] exceptionTypes = constructor.getExceptionTypes();
for (Class<?> exceptionType : exceptionTypes) {
    System.out.println(exceptionType);
}
System.out.println();
}

System.out.println("-----");
//(8)方法们
System.out.println("方法列表: ");
Method[] declaredMethods = clazz.getDeclaredMethods();
for (int i=0; i<declaredMethods.length; i++) {
    Method method = declaredMethods[i];
    System.out.println("第" + (i+1) +"个方法: ");
    //修饰符、返回值类型、方法名、形参列表 、异常列表
    int modifiers = method.getModifiers();
    System.out.println("方法的修饰符: " + Modifier.toString(modifiers));

    Class<?> returnType = method.getReturnType();
    System.out.println("返回值类型: " + returnType);

    String name2 = method.getName();
    System.out.println("方法名: " + name2);
}
```

```

        //形参列表
        System.out.println("形参列表: ");
        Class[] parameterTypes = method.getParameterTypes();
        for (Class parameterType : parameterTypes) {
            System.out.println(parameterType);
        }

        //异常列表
        System.out.println("异常列表: ");
        Class<?>[] exceptionTypes = method.getExceptionTypes();
        for (Class<?> exceptionType : exceptionTypes) {
            System.out.println(exceptionType);
        }
        System.out.println();
    }

}

```

16.4.2 创建任意引用类型的对象

两种方式：

- 1、直接通过Class对象来实例化（要求必须有公共的无参构造）
- 2、通过获取构造器对象来进行实例化

方式一的步骤：

- (1) 获取该类型的Class对象 (2) 创建对象

方式二的步骤：

- (1) 获取该类型的Class对象 (2) 获取构造器对象 (3) 创建对象

如果构造器的权限修饰符修饰的范围不可见，也可以调用setAccessible(true)

示例代码：

```

package com.atguigu.reflect;

import org.junit.Test;

import java.lang.reflect.Constructor;

public class TestCreateObject {
    @Test
    public void test1() throws Exception{
//        AtGuiguclass obj = new AtGuiguclass(); //编译期间无法创建

        Class<?> clazz = Class.forName("com.atguigu.ext.demo.AtGuiguClass");
    }
}

```

```

//clazz代表com.atguigu.ext.demo.AtGuiguClass类型
//clazz.newInstance()创建的就是AtGuiguClass的对象
Object obj = clazz.newInstance();
System.out.println(obj);
}

@Test
public void test2()throws Exception{
    Class<?> clazz = Class.forName("com.atguigu.ext.demo.AtGuiguDemo");
    //java.lang.InstantiationException: com.atguigu.ext.demo.AtGuiguDemo
    //Caused by: java.lang.NoSuchMethodException: com.atguigu.ext.demo.AtGuiguDemo.
<init>()
    //即说明AtGuiguDemo没有无参构造，就没有无参实例初始化方法<init>
    Object stu = clazz.newInstance();
    System.out.println(stu);
}

@Test
public void test3()throws Exception{
    //①获取Class对象
    Class<?> clazz = Class.forName("com.atguigu.ext.demo.AtGuiguDemo");
    /*
     * 获取AtGuiguDemo类型中的有参构造
     * 如果构造器有多个，我们通常是根据形参【类型】列表来获取指定的一个构造器的
     * 例如：public AtGuiguDemo(String title, int num)
     */
    //②获取构造器对象
    Constructor<?> constructor = clazz.getDeclaredConstructor(String.class,int.class);

    //③创建实例对象
    // T newInstance(Object... initargs) 这个Object...是在创建对象时，给有参构造的实参列表
    Object obj = constructor.newInstance("尚硅谷",2022);
    System.out.println(obj);
}
}

```

16.4.3 操作任意类型的属性

(1) 获取该类型的Class对象

```
Class clazz = Class.forName("包.类名");
```

(2) 获取属性对象

```
Field field = clazz.getDeclaredField("属性名");
```

(3) 如果属性的权限修饰符不是public，那么需要设置属性可访问

```
field.setAccessible(true);
```

(4) 创建实例对象：如果操作的是非静态属性，需要创建实例对象

```
Object obj = clazz.newInstance(); //有公共的无参构造
```

```
Object obj = 构造器对象.newInstance(实参...); //通过特定构造器对象创建实例对象
```

(4) 设置属性值

```
field.set(obj,"属性值");
```

如果操作静态变量，那么实例对象可以省略，用null表示

(5) 获取属性值

```
Object value = field.get(obj);
```

如果操作静态变量，那么实例对象可以省略，用null表示

示例代码：

```
package com.atguigu.reflect;

public class Student {
    private int id;
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "Student{" +
            "id=" + id +
            ", name='" + name + '\'' +
            '}';
    }
}
```

```
package com.atguigu.reflect;

import java.lang.reflect.Field;

public class TestField {
    public static void main(String[] args) throws Exception {
        //1、获取Student的Class对象
        Class clazz = Class.forName("com.atguigu.reflect.Student");
```

```

//2、获取属性对象，例如：id属性
Field idField = clazz.getDeclaredField("id");

//3、如果id是私有的等在当前类中不可访问access的，我们需要做如下操作
idField.setAccessible(true);

//4、创建实例对象，即，创建Student对象
Object stu = clazz.newInstance();

//5、获取属性值
/*
 * 以前：int 变量= 学生对象.getId()
 * 现在：Object id属性对象.get(学生对象)
 */
Object value = idField.get(stu);
System.out.println("id = " + value);

//6、设置属性值
/*
 * 以前：学生对象.setId(值)
 * 现在：id属性对象.set(学生对象,值)
 */
idField.set(stu, 2);

value = idField.get(stu);
System.out.println("id = " + value);
}
}

```

16.4.4 调用任意类型的方法

(1) 获取该类型的Class对象

```
Class clazz = Class.forName("包.类名");
```

(2) 获取方法对象

```
Method method = clazz.getDeclaredMethod("方法名",方法的形参类型列表);
```

(3) 创建实例对象

```
Object obj = clazz.newInstance();
```

(4) 调用方法

```
Object result = method.invoke(obj, 方法的实参值列表);
```

如果方法的权限修饰符修饰的范围不可见，也可以调用setAccessible(true)

如果方法是静态方法，实例对象也可以省略，用null代替

示例代码：

```
package com.atguigu.reflect;

import org.junit.Test;

import java.lang.reflect.Method;

public class TestMethod {
    @Test
    public void test() throws Exception {
        // 1、获取Student的Class对象
        Class<?> clazz = Class.forName("com.atguigu.reflect.Student");

        //2、获取方法对象
        /*
         * 在一个类中，唯一定位到一个方法，需要：（1）方法名（2）形参列表，因为方法可能重载
         *
         * 例如：void setName(String name)
         */
        Method setNameMethod = clazz.getDeclaredMethod("setName", String.class);

        //3、创建实例对象
        Object stu = clazz.newInstance();

        //4、调用方法
        /*
         * 以前：学生对象.setName(值)
         * 现在：方法对象.invoke(学生对象，值)
         */
        Object setNameMethodReturnValue = setNameMethod.invoke(stu, "张三");

        System.out.println("stu = " + stu);
        //setName方法返回值类型void，没有返回值，所以setNameMethodReturnValue为null
        System.out.println("setNameMethodReturnValue = " + setNameMethodReturnValue);

        Method getNameMethod = clazz.getDeclaredMethod("getName");
        Object getNameMethodReturnValue = getNameMethod.invoke(stu);
        //getName方法返回值类型String，有返回值，getNameMethod.invoke的返回值就是getName方法的返回值
        System.out.println("getNameMethodReturnValue = " + getNameMethodReturnValue); //张三
    }

    @Test
    public void test02() throws Exception{
        Class<?> clazz = Class.forName("com.atguigu.ext.demo.AtGuiguClass");
        Method printInfoMethod = clazz.getMethod("printInfo", String.class);
        //printInfo方法是静态方法
        printInfoMethod.invoke(null, "尚硅谷");
    }
}
```

16.5 自定义注解

注解是以“@注释名”在代码中存在的，还可以添加一些参数值，例如：

```
@SuppressWarnings(value="unchecked")
@Override
@Deprecated
```

注解Annotation是从JDK5.0开始引入。

虽然说注解也是一种注释，因为它们都不会改变程序原有的逻辑，只是对程序增加了某些注释性信息。不过它又不同于单行注释和多行注释，对于单行注释和多行注释是给程序员看的，而注解是可以被编译器或其他程序读取的一种注释，程序还可以根据注解的不同，做出相应的处理。所以注解是插入到代码中以便有工具可以对它们进行处理的标签。

一个完整的注解应该包含三个部分：（1）声明（2）使用（3）读取

16.5.1 元注解

JDK1.5在java.lang.annotation包定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。

(1) @Target: 用于描述注解的使用范围

- 可以通过枚举类型ElementType的10个常量对象来指定
- TYPE, METHOD, CONSTRUCTOR, PACKAGE.....

(2) @Retention: 用于描述注解的生命周期

- 可以通过枚举类型RetentionPolicy的3个常量对象来指定
- SOURCE (源代码)、CLASS (字节码)、RUNTIME (运行时)
- ==唯有RUNTIME阶段才能被反射读取到==。

(3) @Documented: 表明这个注解应该被 javadoc工具记录。

(4) @Inherited: 允许子类继承父类中的注解

示例代码：

```
package java.lang;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override { }
```

```
package java.lang;

import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

```
package java.lang;

import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
public @interface Deprecated {
}
```

16.5.2 自定义注解

1、声明自定义注解

```
【元注解】
【修饰符】 @interface 注解名{
    【成员列表】
}
```

- 自定义注解可以通过四个元注解@Retention,@Target, @Inherited,@Documented, 分别说明它的声明周期, 使用位置, 是否被继承, 是否被生成到API文档中。
- Annotation 的成员在 Annotation 定义中以无参数有返回值的抽象方法的形式来声明, 我们又称为配置参数。返回值类型只能是八种基本数据类型、String类型、Class类型、enum类型、Annotation类型、以上所有类型的数组
- 可以使用 default 关键字为抽象方法指定默认返回值
- 如果定义的注解含有抽象方法, 那么使用时必须指定返回值, 除非它有默认值。格式是“方法名 = 返回值”, 如果只有一个抽象方法需要赋值, 且方法名为value, 可以省略“value=”, 所以如果注解只有一个抽象方法成员, 建议使用方法名value。

```
package com.atguigu.annotation;

import java.lang.annotation.*;

@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Table {
    String value();
}
```

```
package com.atguigu.annotation;

import java.lang.annotation.*;

@Inherited
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Column {
    String columnName();
    String columnType();
}
```

2、使用自定义注解

```
package com.atguigu.annotation;

@Table("t_stu")
public class Student {
    @Column(columnName = "sid",columnType = "int")
    private int id;
    @Column(columnName = "sname",columnType = "varchar(20)")
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

```
    @Override
    public String toString() {
        return "Student{" +
            "id=" + id +
            ", name='" + name + '\'' +
            '}';
    }
}
```

3、读取和处理自定义注解

自定义注解必须配上注解的信息处理流程才有意义。

我们自己定义的注解，只能使用反射的代码读取。所以自定义注解的声明周期必须是RetentionPolicy.RUNTIME。

```
package com.atguigu.annotation;

import java.lang.reflect.Field;

public class TestAnnotation {
    public static void main(String[] args) {
        Class studentClass = Student.class;
        Table tableAnnotation = (Table) studentClass.getAnnotation(Table.class);
        String tableName = "";
        if(tableAnnotation != null){
            tableName = tableAnnotation.value();
        }

        Field[] declaredFields = studentClass.getDeclaredFields();
        String[] columns = new String[declaredFields.length];
        int index = 0;
        for (Field declaredField : declaredFields) {
            Column column = declaredField.getAnnotation(Column.class);
            if(column!= null) {
                columns[index++] = column.columnName();
            }
        }

        String sql = "select ";
        for (int i=0; i<index; i++) {
            sql += columns[i];
            if(i<index-1){
                sql += ",";
            }
        }
        sql += " from " + tableName;
        System.out.println("sql = " + sql);
    }
}
```

第17章 Java8新特性

17.1 函数式编程思想

在数学中，**函数**就是有输入量、输出量的一套计算方案，也就是“拿什么东西做什么事情”。编程中的函数，也有类似的概念，你调用我的时候，给我实参为形参赋值，然后通过运行方法体，给你返回一个结果。对于调用者来说，关注这个方法具备什么样的功能。相对而言，面向对象过分强调“必须通过对象的形式来做事情”，而函数式思想则尽量忽略面向对象的复杂语法——**强调做什么，而不是以什么形式做**。

- 面向对象的思想：
 - 做一件事情，找一个能解决这个事情的对象，调用对象的方法，完成事情。
- 函数式编程思想：
 - 只要能获取到结果，谁去做的，怎么做的都不重要，重视的是结果，不重视过程

Java8引入了Lambda表达式之后，Java也开始支持函数式编程。

Lambda表达式不是Java最早使用的，很多语言就支持Lambda表达式，例如：C++，C#，Python，Scala等。如果有Python或者Javascript的语言基础，对理解Lambda表达式有很大帮助，可以说Lambda表达式其实就是实现SAM接口的语法糖，使得Java也算是支持函数式编程的语言。Lambda写的好可以极大的减少代码冗余，同时可读性也好过冗长的（啰嗦的）匿名内部类。

备注：“语法糖”是指使用更加方便，但是原理不变的代码语法。例如在遍历集合时使用的for-each语法，其实底层的实现原理仍然是迭代器，这便是“语法糖”。从应用层面来讲，Java中的Lambda可以被当做是匿名内部类的“语法糖”，但是二者在原理上是不同的。

17.1.1 冗余的匿名内部类

当需要启动一个线程去完成任务时，通常会通过 `java.lang.Runnable` 接口来定义任务内容，并使用 `java.lang.Thread` 类来启动该线程。代码如下：

```
package com.atguigu.fp;

public class UseFunctionalProgramming {
    public static void main(String[] args) {
        new Thread(() -> System.out.println("多线程任务执行! ")).start(); // 启动线程
    }
}
```

本着“一切皆对象”的思想，这种做法是无可厚非的：首先创建一个 `Runnable` 接口的匿名内部类对象来指定任务内容，再将其交给一个线程来启动。

代码分析：

对于 `Runnable` 的匿名内部类用法，可以分析出几点内容：

- `Thread` 类需要 `Runnable` 接口作为参数，其中的抽象 `run` 方法是用来指定线程任务内容的核心；
- 为了指定 `run` 的方法体，**不得不**需要 `Runnable` 接口的实现类；
- 为了省去定义一个 `RunnableImpl` 实现类的麻烦，**不得不**使用匿名内部类；

- 必须覆盖重写抽象 `run` 方法，所以方法名称、方法参数、方法返回值**不得不再写一遍**，且不能写错；
- 而实际上，**似乎只有方法体才是关键所在**。

17.1.2 编程思想转换

做什么，而不是谁来做，怎么做

我们真的希望创建一个匿名内部类对象吗？不。我们只是为了做这件事情而**不得不**创建一个对象。我们真正希望做的事情是：将 `run` 方法体内的代码传递给 `Thread` 类知晓。

传递一段代码——这才是我们真正的目的。而创建对象只是受限于面向对象语法而不得不采取的一种手段方式。那，有没有更加简单的办法？如果我们将关注点从“怎么做”回归到“做什么”的本质上，就会发现只要能够更好地达到目的，过程与形式其实并不重要。

生活举例：

当我们需要从北京到上海时，可以选择高铁、汽车、骑行或是徒步。我们的真正目的是到达上海，而如何才能到达上海的形式并不重要，所以我们一直在探索有没有比高铁更好的方式——搭乘飞机。

而现在这种飞机（甚至是飞船）已经诞生：2014年3月Oracle所发布的Java 8（JDK 1.8）中，加入了**Lambda表达式**的重量级新特性，为我们打开了新世界的大门。

17.1.3 体验Lambda的更优写法

借助Java 8的全新语法，上述 `Runnable` 接口的匿名内部类写法可以通过更简单的Lambda表达式达到等效：

```
public class Demo02LambdaRunnable {
    public static void main(String[] args) {
        new Thread(() -> System.out.println("多线程任务执行! ")).start(); // 启动线程
    }
}
```

这段代码和刚才的执行效果是完全一样的，可以在1.8或更高的编译级别下通过。从代码的语义中可以看出：我们启动了一个线程，而线程任务的内容以一种更加简洁的形式被指定。

不再有“不得不创建接口对象”的束缚，不再有“抽象方法覆盖重写”的负担，就是这么简单！

17.2 函数式接口

17.2.1 函数接口的概念

lambda表达式其实就是实现SAM接口的语法糖，所谓SAM接口就是Single Abstract Method，即该接口中只有一个抽象方法需要实现，当然该接口可以包含其他非抽象方法。

其实只要满足“SAM”特征的接口都可以称为函数式接口，都可以使用Lambda表达式，但是如果要更明确一点，最好在声明接口时，加上`@FunctionalInterface`。一旦使用该注解来定义接口，编译器将会强制检查该接口是否确实有且仅有一个抽象方法，否则将会报错。

17.2.2 盘点之前学过的接口

之前学过的接口已经很多了：`Cloneable`、`Comparable`、`Comparator`、`Runnable`、`Iterable`、`Iterator`、`Collection`、`List`、`Queue`、`Deque`、`Set`、`Map`、`Serializable`、`FileFilter`、`FilenameFilter`等。

上述接口中，满足SAM接口特点的有：

- java.lang.Runnable
 - public void run()
- java.lang.Iterable
 - public Iterator iterate()
- java.lang.Comparable
 - public int compareTo(T t)
- java.util.Comparator
 - public int compare(T t1, T t2)
- java.io.FileFilter
 - public boolean accept(File pathname)
- java.io.FilenameFilter
 - public boolean accept(File dir, String name)

上述SAM接口中，标记了@FunctionalInterface注解有：Runnable, Comparator, FileFilter, FilenameFilter。

17.2.3 java.util.function包四大类函数式接口

Java8在java.util.function新增了很多函数式接口：主要分为四大类，消费型、供给型、判断型、功能型。基本可以满足我们的开发需求。当然你也可以定义自己的函数式接口。

1、消费型接口

消费型接口的抽象方法特点：有形参，但是返回值类型是void

| 接口名 | 抽象方法 | 描述 |
|-------------------|--------------------------------|------------------|
| Consumer | void accept(T t) | 接收一个对象用于完成功能 |
| BiConsumer<T,U> | void accept(T t, U u) | 接收两个对象用于完成功能 |
| DoubleConsumer | void accept(double value) | 接收一个double值 |
| IntConsumer | void accept(int value) | 接收一个int值 |
| LongConsumer | void accept(long value) | 接收一个long值 |
| ObjDoubleConsumer | void accept(T t, double value) | 接收一个对象和一个double值 |
| ObjIntConsumer | void accept(T t, int value) | 接收一个对象和一个int值 |
| ObjLongConsumer | void accept(T t, long value) | 接收一个对象和一个long值 |

2、供给型接口

这类接口的抽象方法特点：无参，但是有返回值

| 接口名 | 抽象方法 | 描述 |
|-----------------|------------------------|--------------|
| Supplier | T get() | 返回一个对象 |
| BooleanSupplier | boolean getAsBoolean() | 返回一个boolean值 |
| DoubleSupplier | double getAsDouble() | 返回一个double值 |
| IntSupplier | int getAsInt() | 返回一个int值 |
| LongSupplier | long getAsLong() | 返回一个long值 |

3、判断型接口

这类接口的抽象方法特点：有参，但是返回值类型是boolean结果。

| 接口名 | 抽象方法 | 描述 |
|------------------|----------------------------|-------------|
| Predicate | boolean test(T t) | 接收一个对象 |
| BiPredicate<T,U> | boolean test(T t, U u) | 接收两个对象 |
| DoublePredicate | boolean test(double value) | 接收一个double值 |
| IntPredicate | boolean test(int value) | 接收一个int值 |
| LongPredicate | boolean test(long value) | 接收一个long值 |

4、功能型接口

这类接口的抽象方法特点：既有参数又有返回值

| 接口名 | 抽象方法 | 描述 |
|-------------------------|--------------------------------------|-----------------------------|
| Function<T,R> | R apply(T t) | 接收一个T类型对象，返回一个R类型对象结果 |
| UnaryOperator | T apply(T t) | 接收一个T类型对象，返回一个T类型对象结果 |
| DoubleFunction | R apply(double value) | 接收一个double值，返回一个R类型对象 |
| IntFunction | R apply(int value) | 接收一个int值，返回一个R类型对象 |
| LongFunction | R apply(long value) | 接收一个long值，返回一个R类型对象 |
| ToDoubleFunction | double applyAsDouble(T value) | 接收一个T类型对象，返回一个double |
| ToIntFunction | int applyAsInt(T value) | 接收一个T类型对象，返回一个int |
| ToLongFunction | long applyAsLong(T value) | 接收一个T类型对象，返回一个long |
| DoubleToIntFunction | int applyAsInt(double value) | 接收一个double值，返回一个int结果 |
| DoubleToLongFunction | long applyAsLong(double value) | 接收一个double值，返回一个long结果 |
| IntToDoubleFunction | double applyAsDouble(int value) | 接收一个int值，返回一个double结果 |
| IntToLongFunction | long applyAsLong(int value) | 接收一个int值，返回一个long结果 |
| LongToDoubleFunction | double applyAsDouble(long value) | 接收一个long值，返回一个double结果 |
| LongToIntFunction | int applyAsInt(long value) | 接收一个long值，返回一个int结果 |
| DoubleUnaryOperator | double applyAsDouble(double operand) | 接收一个double值，返回一个double |
| IntUnaryOperator | int applyAsInt(int operand) | 接收一个int值，返回一个int结果 |
| LongUnaryOperator | long applyAsLong(long operand) | 接收一个long值，返回一个long结果 |
| BiFunction<T,U,R> | R apply(T t, U u) | 接收一个T类型和一个U类型对象，返回一个R类型对象结果 |
| BinaryOperator | T apply(T t, T u) | 接收两个T类型对象，返回一个T类型对象结果 |
| ToDoubleBiFunction<T,U> | double applyAsDouble(T t, U u) | 接收一个T类型和一个U类型对象，返回一个double |
| ToIntBiFunction<T,U> | int applyAsInt(T t, U u) | 接收一个T类型和一个U类型对象，返回一个int |

| 接口名 | 抽象方法 | 描述 |
|-----------------------|---|--------------------------|
| ToLongBiFunction<T,U> | long applyAsLong(T t, U u) | 接收一个T类型和一个U类型对象，返回一个long |
| DoubleBinaryOperator | double applyAsDouble(double left, double right) | 接收两个double值，返回一个double结果 |
| IntBinaryOperator | int applyAsInt(int left, int right) | 接收两个int值，返回一个int结果 |
| LongBinaryOperator | long applyAsLong(long left, long right) | 接收两个long值，返回一个long结果 |

17.2.4 自定义函数式接口

只要确保接口中有且仅有一个抽象方法即可：

```
修饰符 interface 接口名称 {
    public abstract 返回值类型 方法名称(可选参数信息);
    // 其他非抽象方法内容
}
```

接口当中抽象方法的 public abstract 是可以省略的

例如：声明一个计算器 `calculator <T,R>` 接口，内含抽象方法 `calculate` 可以对两个参数进行计算，并返回结果。其中 T 是参数类型，R 是返回值类型。

```
package com.atguigu.fi;

@FunctionalInterface
public interface Calculator<T,R> {
    R calculate(T a, T b);
}
```

例如：声明一个转换器 `Convertor<T,R>`，包含抽象方法 `change`，可以将参数转换为另一个值，并返回结果。其中 T 是参数类型，R 是返回值类型。

```
package com.atguigu.fi;

@FunctionalInterface
public interface Convertor<T,R> {
    R change(T t);
}
```

17.3 Lambda表达式

17.3.1 Lambda表达式语法

Lambda 表达式是用来给【函数式接口】的变量或形参赋值用的。其实本质上，Lambda 表达式是用于实现【函数式接口】的“抽象方法”，或者是给函数式接口的变量传递一段实现抽象方法的方法体代码。

Lambda表达式语法格式

```
(形参列表) -> {Lambda体}
```

语法格式说明：

- (形参列表) 它就是你要赋值的函数式接口的抽象方法的(形参列表), 照抄
- {Lambda体} 就是实现这个抽象方法的方法体
- -> 称为Lambda操作符 (减号和大于号中间不能有空格, 而且必须是英文状态下半角输入方式)

17.3.2 使用Lambda表达式标准格式

```
package com.atguigu.lambda;

import com.atguigu.fi.calculator;

public class LambdaGrammar {
    public static void main(String[] args) {
        calculator<Integer, Integer> c1 = (Integer a, Integer b) -> {return a+b;};
        calculator<Integer, Integer> c2 = (Integer a, Integer b) -> {return a-b;};
        calculator<Integer, Integer> c3 = (Integer a, Integer b) -> {return a*b;};
        calculator<Integer, Integer> c4 = (Integer a, Integer b) -> {return a/b;};

        System.out.println(c1.calculate(5, 2));
        System.out.println(c2.calculate(5, 2));
        System.out.println(c3.calculate(5, 2));
        System.out.println(c4.calculate(5, 2));
    }
}
```

17.3.3 Lambda表达式的简化

某些情况下Lambda表达式可以精简：

- 当{Lambda体}中只有一句语句时, 可以省略{}和();
- 当{Lambda体}中只有一句语句时, 并且这个语句还是一个return语句, 那么{}、return、;三者可以省略。它们要么一起省略, 要么都不省略。
- 当Lambda表达式(形参列表)的类型已知, 获取根据泛型规则可以自动推断, 那么(形参列表)的数据类型可以省略。
- 当Lambda表达式(形参列表)的形参个数只有一个, 并且类型已知或可以自动推断, 则形参的数据类型和()可以一起省略, 但是形参名不能省略。
- 当Lambda表达式(形参列表)是空参时, ()不能省略

示例代码：

```
package com.atguigu.lambda;

import com.atguigu.fi.calculator;
import com.atguigu.fi.Convertor;
import org.junit.Test;
```

```

public class LambdaGrammarsimple {
    @Test
    public void test01() {
        //使用Lambda表达式实现Calculator接口，求两个整数的和的功能
        Calculator<Integer, Integer> c1 = (Integer a, Integer b) -> {return a+b;};
        System.out.println(c1.calculate(5, 2));

        Calculator<Integer, Integer> c2 = (Integer a, Integer b) -> a+b;
        System.out.println(c2.calculate(5, 2));

        Calculator<Integer, Integer> c3 = (a, b) -> a+b;
        System.out.println(c3.calculate(5, 2));
        //上面三种写法完全等价
    }

    @Test
    public void test02() {
        //使用Lambda表达式实现Convertor接口，实现取字符串的首字母的功能
        Convertor<String, Character> c1 = (String str)-> {return str.charAt(0);};
        System.out.println(c1.change("hello"));

        Convertor<String, Character> c2 = (String str)-> str.charAt(0);
        System.out.println(c2.change("world"));

        Convertor<String, Character> c3 = (str)-> str.charAt(0);
        System.out.println(c3.change("atguigu"));

        Convertor<String, Character> c4 = str-> str.charAt(0);
        System.out.println(c4.change("chai"));
        //上面四种写法完全一致
    }
}

```

17.3.4 四大类函数式接口使用演示

1、消费型接口Consumer<T>

已在JDK1.8中java.lang.Iterable接口中增加了一个默认方法：

- `public default void forEach(Consumer<? super T> action)` 该方法功能是遍历Collection集合，并将传递给action参数的操作代码应用在每一个元素上。

因为Collection接口继承了Iterable接口，这就意味着所有Collection系列的接口都包含该方法。

```
package com.atguigu.four;

import java.util.Arrays;
import java.util.List;

public class TestConsumer {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("java", "c", "python", "c++", "VB", "C#");
        list.forEach(s -> System.out.println(s));
    }
}
```

2、供给型接口Supplier<T>

```
package com.atguigu.four;

import java.util.function.Supplier;

public class TestSupplier {
    public static void main(String[] args) {
        Supplier<String> supplier = () -> "尚硅谷";
        System.out.println(supplier.get());
    }
}
```

3、判断型接口Predicate<T>

已知：JDK1.8时，Collecton接口增加了一下方法，其中一个如下：

- `public default boolean removeIf(Predicate<? super E> filter)` 用于删除集合中满足filter指定的条件判断的。

```
package com.atguigu.four;

import java.util.ArrayList;

public class TestPredicate {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("hello");
        list.add("java");
        list.add("atguigu");
        list.add("ok");
        list.add("yes");

        System.out.println("删除之前: ");
        list.forEach(t-> System.out.println(t));

        //删除包含o字母的元素
        list.removeIf(s -> s.contains("o"));
    }
}
```

```
        System.out.println("删除包含o字母的元素之后: ");
        list.forEach(t-> System.out.println(t));
    }
}
```

4、功能型接口Function<T,R>

```
package com.atguigu.four;

import java.util.function.Function;

public class TestFunction {
    public static void main(String[] args) {
        //使用Lambda表达式实现Function<T,R>接口，可以实现将一个字符串首字母转为大写的功能。
        Function<String, String> fun = s -> s.substring(0,1).toUpperCase() + s.substring(1);
        System.out.println(fun.apply("hello"));
    }
}
```

17.4 方法引用与构造器引用

Lambda表达式是可以简化函数式接口的变量与形参赋值的语法。而方法引用和构造器引用是为了简化Lambda表达式的。

17.4.1 方法引用

当Lambda表达式满足一些特殊的情况时，还可以再简化：

(1) Lambda体只有一句语句，并且是通过调用一个对象的/类现有的方法来完成的

例如：System.out对象，调用println()方法来完成Lambda体

Math类，调用random()静态方法来完成Lambda体

(2) 并且Lambda表达式的形参正好是给该方法的实参

例如：t->System.out.println(t)

() -> Math.random() 都是无参

方法引用的语法格式：

(1) 实例对象名::实例方法

(2) 类名::静态方法

(3) 类名::实例方法

说明：

- ::称为方法引用操作符（两个::中间不能有空格，而且必须英文状态下半角输入）

- Lambda表达式的形参列表，全部在Lambda体中使用上了，
 - 类名.静态方法：Lambda表达式的形参列表正好全部作为所调用的静态方法的实参
 - 对象.实例方法：Lambda表达式的形参列表正好全部作为所调用的实例方法的实参
 - 对象.实例方法：Lambda表达式的形参列表的第一个形参就是调用方法的对象，Lambda表达式的形参列表剩下的形参正好作为所调用实例方法的实参。
- 在整个Lambda体中没有额外的数据。

```

package com.atguigu.reference;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.function.Supplier;

public class MethodReference {

    @Test
    public void test44(){
        String[] arr = {"Hello","java","chai"};
        Arrays.sort(arr, (s1,s2) -> s1.compareToIgnoreCase(s2));

        //用方法引用简化
        /*
         * Lambda表达式的形参，第一个（例如：s1），正好是调用方法的对象，剩下的形参（例如：s2）正好是给这个方法的实参
         */
        Arrays.sort(arr, String::compareToIgnoreCase);
    }

    @Test
    public void test3(){
        Random random = new Random();
        Supplier<Integer> s1 = () -> random.nextInt();
        Supplier<Integer> s2 = random :: nextInt;//用方法引用简化
        System.out.println(s1.get());
        System.out.println(s2.get());
        //上面两个写法是等价的

        Supplier<Integer> s3 = () -> random.nextInt(100);
        Supplier<Integer> s4 = random :: nextInt;//用方法引用简化 缺100
        System.out.println(s3.get());
        System.out.println(s4.get());
        //上面两个写法是不等价的
    }

    @Test
    public void test2(){
        Supplier<Double> s1 = () -> Math.random();
        Supplier<Double> s2 = Math :: random;//用方法引用简化
        System.out.println(s1.get());
    }
}

```

```

        System.out.println(s2.get());
        //上面两个写法是等价的
    }

    @Test
    public void test1(){
        List<Integer> list = Arrays.asList(1,3,4,8,9);
        //list.forEach(t -> System.out.println(t));

        //用方法引用再简化
        list.forEach(System.out::println);
    }
}

```

17.4.2 构造器引用

当Lambda表达式是创建一个对象，并且满足Lambda表达式形参，正好是给创建这个对象的构造器的实参列表，就可以使用构造器引用：

- 类名::new

```

package com.atguigu.reference;

import java.util.function.Function;

public class ConstructorReference {
    public static void main(String[] args) {
//        Function<String,Person> function = s -> new Person(s);
        Function<String,Person> function = Person::new;

        String[] names = {"张三", "李四", "王五"};
        Person[] people = new Person[names.length];
        for (int i = 0; i < people.length; i++) {
            people[i] = function.apply(names[i]);
        }

        for (Person person : people) {
            System.out.println(person);
        }
    }
}

```

```

package com.atguigu.reference;

public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }
}

```

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

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

@Override
public String toString() {
    return "Person{" +
        "name='" + name + '\'' +
        '}';
}
}
```

17.4.3 数组构造引用

当Lambda表达式是创建一个数组对象，并且满足Lambda表达式形参，正好是给创建这个数组对象的长度，就可以数组构造引用：

- 数组类型名::new

示例代码：

```
package com.atguigu.reference;

import java.util.Arrays;
import java.util.function.Function;

public class ArrayCreateReference {
    public static void main(String[] args) {
        Function<Integer, String[]> function = String[]::new;

        String[] arr = function.apply(5);
        System.out.println(Arrays.toString(arr));
    }
}
```

17.5 StreamAPI

Java8中有两大最为重要的改变。第一个是 Lambda 表达式；另外一个则是 Stream API。

17.5.1 Stream特点

Stream API (`java.util.stream`) 把真正的函数式编程风格引入到Java中。这是目前为止对Java类库最好的补充，因为 Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

Stream 是 Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 Stream API 来并行执行操作。简言之，Stream API 提提供了一种高效且易于使用的处理数据的方式。

Stream是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。“集合讲的是数据，负责存储数据，Stream流讲的是计算，负责处理数据！”

注意：

- ① Stream自己不会存储元素。
- ② Stream不会改变源对象。每次处理都会返回一个持有结果的新Stream。
- ③ Stream操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

17.5.2 Stream 的操作三个步骤

1- 创建 Stream：通过一个数据源（如：集合、数组），获取一个流

2- 中间操作：每次处理都会返回一个持有结果的新Stream，即中间操作的方法返回值仍然是Stream类型的对象，因此中间操作可以是个操作链，可对数据源的数据进行n次处理，但是在终结操作前，并不会真正执行。

3- 终止操作：终止操作的方法返回值类型就不再是Stream了，因此一旦执行终止操作，就结束整个Stream操作了。一旦执行终止操作，就执行中间操作链，最终产生结果并结束Stream。





```
package com.atguigu.stream;

import org.junit.Test;

import java.util.stream.Stream;

public class StreamDemo {

    @Test
    public void test01(){
        Stream<Integer> stream = Stream.of(1,2,3,4,5,2,4,6); //创建Stream
        stream = stream.filter(num -> num%2==0); //中间处理，筛选偶数
        stream = stream.distinct(); //中间处理，去重复
        stream.forEach(System.out::println); //终结操作
    }

    @Test
    public void test02(){
        //连写
        Stream.of(1,2,3,4,5,2,4,6) //创建Stream
            .filter(num -> num%2==0) //中间处理，筛选偶数
            .distinct() //中间处理，去重复
            .forEach(System.out::println); //终结操作
    }
}
```

17.5.3 创建StreamAPI

1、创建 Stream 方式一：通过集合

Java8 中的 Collection 接口被扩展，提供了两个获取流的方法：

- public default Stream stream() : 返回一个顺序流
- public default Stream parallelStream() : 返回一个并行流

2、创建 Stream 方式二：通过数组

Java8 中的 Arrays 的静态方法 stream() 可以获取数组流：

- public static Stream stream(T[] array): 返回一个流

重载形式，能够处理对应基本类型的数组：

- public static IntStream stream(int[] array): 返回一个整型数据流
- public static LongStream stream(long[] array): 返回一个长整型数据流
- public static DoubleStream stream(double[] array): 返回一个浮点型数据流

3、创建 Stream 方式三：通过 Stream 的 of()

可以调用 Stream 类静态方法 of(), 通过显示值创建一个流。它可以接收任意数量的参数。

- public static Stream of(T... values) : 返回一个顺序流

4、创建 Stream 方式四：创建无限流

可以使用静态方法 Stream.iterate() 和 Stream.generate(), 创建无限流。

- public static Stream iterate(final T seed, final UnaryOperator f): 返回一个无限流
- public static Stream generate(Supplier s) : 返回一个无限流

```
package com.atguigu.stream;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class StreamCreate {
    @Test
    public void test06(){
        /*
         * Stream<T> iterate(T seed, UnaryOperator<T> f)
         * UnaryOperator 接口, SAM 接口, 抽象方法:
         *
         * UnaryOperator<T> extends Function<T,T>
         *     T apply(T t)
         */
        Stream<Integer> stream = Stream.iterate(1, num -> num+=2);
        //        stream = stream.limit(10);
        stream.forEach(System.out::println);
    }

    @Test
    public void test05(){}
```

```
Stream<Double> stream = Stream.generate(Math::random);
stream.forEach(System.out::println);
}

@Test
public void test04(){
    Stream<Integer> stream = Stream.of(1,2,3,4,5);
    stream.forEach(System.out::println);
}

@Test
public void test03(){
    String[] arr = {"hello","world"};
    Stream<String> stream = Arrays.stream(arr);
}

@Test
public void test02(){
    int[] arr = {1,2,3,4,5};
    IntStream stream = Arrays.stream(arr);
}

@Test
public void test01(){
    List<Integer> list = Arrays.asList(1,2,3,4,5);

    //JDK1.8中，Collection系列集合增加了方法
    Stream<Integer> stream = list.stream();
}
}
```

17.5.4 中间操作API

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何的处理！而在终止操作时一次性全部处理，称为“惰性求值”。

| 方法 | 描述 |
|--|---|
| filter(Predicate p) | 接收 Lambda , 从流中排除某些元素 |
| distinct() | 筛选, 通过流所生成元素的equals() 去除重复元素 |
| limit(long maxSize) | 截断流, 使其元素不超过给定数量 |
| skip(long n) | 跳过元素, 返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个, 则返回一个空流。与 limit(n) 互补 |
| peek(Consumer action) | 接收Lambda, 对流中的每个数据执行Lambda体操作 |
| sorted() | 产生一个新流, 其中按自然顺序排序 |
| sorted(Comparator com) | 产生一个新流, 其中按比较器顺序排序 |
| map(Function f) | 接收一个函数作为参数, 该函数会被应用到每个元素上, 并将其映射成一个新的元素。 |
| mapToDouble(ToDoubleFunction f) | 接收一个函数作为参数, 该函数会被应用到每个元素上, 产生一个新的 DoubleStream。 |
| mapToInt(ToIntFunction f) | 接收一个函数作为参数, 该函数会被应用到每个元素上, 产生一个新的 IntStream。 |
| mapToLong(ToLongFunction f) | 接收一个函数作为参数, 该函数会被应用到每个元素上, 产生一个新的 LongStream。 |
| flatMap(Function f) | 接收一个函数作为参数, 将流中的每个值都换成另一个流, 然后把所有流连接成一个流 |

```

package com.atguigu.stream;

import org.junit.Test;

import java.util.Arrays;
import java.util.stream.Stream;

public class StreamMiddleOperate {

    @Test
    public void test12(){
        String[] arr = {"hello","world","java"};
        Arrays.stream(arr)
            .flatMap(t -> Stream.of(t.split("|")))//Function<T,R>接口抽象方法 R apply(T t)
    现在的R是一个Stream
            .forEach(System.out::println);
    }

    @Test
    public void test11(){
        String[] arr = {"hello","world","java"};
    }
}

```

```
        Arrays.stream(arr)
            .map(t->t.toUpperCase())
            .forEach(System.out::println);
    }

    @Test
    public void test10(){
        Stream.of(1,2,3,4,5)
            .map(t -> t+1)//Function<T,R>接口抽象方法 R apply(T t)
            .forEach(System.out::println);
    }

    @Test
    public void test09(){
        //希望能够找出前三个最大值，前三名最大的，不重复
        Stream.of(11,2,39,4,54,6,2,22,3,3,4,54,54)
            .distinct()
            .sorted((t1,t2) -> -Integer.compare(t1, t2))//Comparator接口 int compare(T t1, T t2)
            .limit(3)
            .forEach(System.out::println);
    }

    @Test
    public void test08(){
        long count = Stream.of(1,2,3,4,5,6,2,2,3,3,4,4,5)
            .distinct()
            .peek(System.out::println) //Consumer接口的抽象方法 void accept(T t)
            .count();
        System.out.println("count="+count);
    }

    @Test
    public void test07(){
        Stream.of(1,2,3,4,5,6,2,2,3,3,4,4,5)
            .skip(5)
            .distinct()
            .filter(t -> t%3==0)
            .forEach(System.out::println);
    }

    @Test
    public void test06(){
        Stream.of(1,2,3,4,5,6,2,2,3,3,4,4,5)
            .skip(5)
            .forEach(System.out::println);
    }

    @Test
    public void test05(){
        Stream.of(1,2,2,3,3,4,4,5,2,3,4,5,6,7)
```

```

        .distinct() // (1,2,3,4,5,6,7)
        .filter(t -> t%2!=0) // (1,3,5,7)
        .limit(3)
        .forEach(System.out::println);
    }

@Test
public void test04(){
    Stream.of(1,2,3,4,5,6,2,2,3,3,4,4,5)
        .limit(3)
        .forEach(System.out::println);
}

@Test
public void test03(){
    Stream.of(1,2,3,4,5,6,2,2,3,3,4,4,5)
        .distinct()
        .forEach(System.out::println);
}

@Test
public void test02(){
    Stream.of(1,2,3,4,5,6)
        .filter(t -> t%2==0)
        .forEach(System.out::println);
}

@Test
public void test01(){
    // 1、创建 Stream
    Stream<Integer> stream = Stream.of(1,2,3,4,5,6);

    // 2、加工处理
    // 过滤: filter(Predicate p)
    // 把里面的偶数拿出来
    /*
     * filter(Predicate p)
     * Predicate 是函数式接口, 抽象方法: boolean test(T t)
     */
    stream = stream.filter(t -> t%2==0);

    // 3、终结操作: 例如: 遍历
    stream.forEach(System.out::println);
}
}

```

17.5.5 终结操作API

终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List、Integer，甚至是void。流进行了终止操作后，不能再次使用。

| 方法 | 描述 |
|---|--|
| boolean allMatch(Predicate p) | 检查是否匹配所有元素 |
| boolean anyMatch(Predicate p) | 检查是否至少匹配一个元素 |
| boolean noneMatch(Predicate p) | 检查是否没有匹配所有元素 |
| Optional findFirst() | 返回第一个元素 |
| Optional findAny() | 返回当前流中的任意元素 |
| long count() | 返回流中元素总数 |
| Optional max(Comparator c) | 返回流中最大值 |
| Optional min(Comparator c) | 返回流中最小值 |
| void forEach(Consumer c) | 迭代 |
| T reduce(T iden, BinaryOperator b) | 可以将流中元素反复结合起来，得到一个值。返回 T |
| U reduce(BinaryOperator b) | 可以将流中元素反复结合起来，得到一个值。返回 Optional |
| R collect(Collector c) | 将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream中元素做汇总的方法 |

Collector 接口中方法的实现决定了如何对流执行收集的操作(如收集到 List、Set、Map)。另外，Collectors 实用类提供了很多静态方法，可以方便地创建常见收集器实例。

```
package com.atguigu.stream;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import org.junit.Test;

public class StreamEndding {

    @Test
    public void test14(){
        List<Integer> list = Stream.of(1,2,4,5,7,8)
            .filter(t -> t%2==0)
```

```
        .collect(Collectors.toList());

        System.out.println(list);
    }

@Test
public void test13(){
    Optional<Integer> max = Stream.of(1,2,4,5,7,8)
        .reduce((t1,t2) -> t1>t2?t1:t2);//BinaryOperator接口 T apply(T t1, T t2)
    System.out.println(max);
}

@Test
public void test12(){
    Integer reduce = Stream.of(1,2,4,5,7,8)
        .reduce(0, (t1,t2) -> t1+t2);//BinaryOperator接口 T apply(T t1, T t2)
    System.out.println(reduce);
}

@Test
public void test11(){
    Optional<Integer> max = Stream.of(1,2,4,5,7,8)
        .max((t1,t2) -> Integer.compare(t1, t2));
    System.out.println(max);
}

@Test
public void test10(){
    Optional<Integer> opt = Stream.of(1,2,4,5,7,8)
        .filter(t -> t%3==0)
        .findFirst();
    System.out.println(opt);
}

@Test
public void test09(){
    Optional<Integer> opt = Stream.of(1,2,3,4,5,7,9)
        .filter(t -> t%3==0)
        .findFirst();
    System.out.println(opt);
}

@Test
public void test08(){
    Optional<Integer> opt = Stream.of(1,3,5,7,9).findFirst();
    System.out.println(opt);
}

@Test
public void test04(){
    boolean result = Stream.of(1,3,5,7,9)
        .anyMatch(t -> t%2==0);
```

```

        System.out.println(result);
    }

    @Test
    public void test03(){
        boolean result = Stream.of(1,3,5,7,9)
            .allMatch(t -> t%2!=0);
        System.out.println(result);
    }

    @Test
    public void test02(){
        long count = Stream.of(1,2,3,4,5)
            .count();
        System.out.println("count = " + count);
    }

    @Test
    public void test01(){
        Stream.of(1,2,3,4,5)
            .forEach(System.out::println);
    }
}

```

17.6 Optional类

到目前为止，臭名昭著的空指针异常是导致Java应用程序失败的最常见原因。以前，为了解决空指针异常，Google公司著名的Guava项目引入了Optional类，Guava通过使用检查空值的方式来防止代码污染，它鼓励程序员写更干净的代码。受到Google Guava的启发，Optional类已经成为Java 8类库的一部分。

Optional实际上是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不需要显式地进行空值检测。

1、如何创建Optional对象？

如何用Optional来装值对象或null值呢？

- (1) static Optional empty()：用来创建一个空的Optional
- (2) static Optional of(T value)：用来创建一个非空的Optional
- (3) static Optional ofNullable(T value)：用来创建一个可能是空，也可能非空的Optional

2、如何从Optional容器中取出所包装的对象呢？

- (1) T get()：要求Optional容器必须非空

T get()与of(T value)使用是安全的

- (2) T orElse(T other)：

orElse(T other) 与 ofNullable(T value) 配合使用，

如果 Optional 容器中非空，就返回所包装值，如果为空，就用 orElse(T other) other 指定的默认值（备胎）代替

(3) T orElseGet(Supplier<? extends T> other) :

如果 Optional 容器中非空，就返回所包装值，如果为空，就用 Supplier 接口的 Lambda 表达式提供的值代替

(4) T orElseThrow(Supplier<? extends X> exceptionSupplier)

如果 Optional 容器中非空，就返回所包装值，如果为空，就抛出你指定的异常类型代替原来的 NoSuchElementException

3、其他方法

(1) boolean isPresent() : 判断 Optional 容器中的值是否存在

(2) void ifPresent(Consumer<? super T> consumer) :

判断 Optional 容器中的值是否存在，如果存在，就对它进行 Consumer 指定的操作，如果不存在就不做

(3) Optional map(Function<? super T,? extends U> mapper)

判断 Optional 容器中的值是否存在，如果存在，就对它进行 Function 接口指定的操作，如果不存在就不做

```
package com.atguigu.optional;

import java.util.Optional;

import org.junit.Test;

public class TestOptional {
    @Test
    public void test9(){
        String str = "Hello1";
        Optional<String> opt = Optional.ofNullable(str);
        // 判断是否是纯字母单词，如果是，转为大写，否则保持不变
        String result = opt.filter(s->s.matches("[a-zA-Z]+"))
            .map(s -> s.toUpperCase()).orElse(str);
        System.out.println(result);
    }
}
```

```
@Test
public void test8(){
    String str = null;
    Optional<String> opt = Optional.ofNullable(str);
    String string = opt.orElseThrow(()->new RuntimeException("值不存在"));
    System.out.println(string);
}
```

```
@Test
public void test7(){
    String str = null;
    Optional<String> opt = Optional.ofNullable(str);
```

```
        String string = opt.orElseGet(String::new);
        System.out.println(string);
    }

    @Test
    public void test6(){
        String str = "hello";
        Optional<String> opt = Optional.ofNullable(str);
        String string = opt.orElse("atguigu");
        System.out.println(string);
    }

    @Test
    public void test5(){
        String str = null;
        Optional<String> opt = Optional.ofNullable(str);
//        System.out.println(opt.get());//java.util.NoSuchElementException: No value present
    }

    @Test
    public void test4(){
        String str = "hello";
        Optional<String> opt = Optional.of(str);

        String string = opt.get();
        System.out.println(string);
    }

    @Test
    public void test3(){
        String str = null;
        Optional<String> opt = Optional.ofNullable(str);
        System.out.println(opt);
    }

    @Test
    public void test2(){
        Optional<String> opt = Optional.empty();
        System.out.println(opt);
    }

    @Test
    public void test1(){
        String str = "hello";
        Optional<String> opt = Optional.of(str);
        System.out.println(opt);
    }
}
```

