《Java 基础从入门到精通（JDK17 版）》

电子书作者：尚硅谷教育-宋红康

官网：http://www.atguigu.com

第 01 章_Java 语言概述

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

1. Java 知识脉络图

1.1 Java 基础全程脉络图

Java 基础全程脉络图

1.2 本章专题与脉络

第01章-Java语言概述
说明
重点
难点
案例、项目
专题1-1：课程目录与Java基础知识脉络图
专题1-2：抽丝剥茧话Java
就业形势分析
IT行业岗位分析
Java的应用领域
专题1-3：Java中高级程序员全程学习路线图
Java职业晋升路线图
专题1-4：软件开发及人机交互方式
计算机组成：硬件+软件
系统软件vs应用软件
图形化界面vs命令行方式
常用DOS命令
专题1-5：计算机编程语言概述
计算机语言简史
语言排行榜
编程语言，该学哪个
专题1-6：Java语言简史、平台体系
Java发展简史
Java之父
SUN、Oracle、Google八卦
Java技术体系平台
专题1-7：JDK下载、安装与环境变量配置
JDK、JRE、JVM关系
JDK版本选择
JDK下载与安装
配置path环境变量
专题1-8：HelloWorld的编写、编译与运行
HelloWorld的编写
编译与运行
EditPlus软件的使用
专题1-9：常见问题汇总与HelloWorld小结
专题1-10：Java中的注释与API文档
单行注释、多行注释
文档注释
API文档
专题1-11：Java核心机制：JVM
Java语言的优缺点
JVM的两大功能
专题1-12：案例
案例：个人信息输出
案例：心形表白
专题1-13：本章随堂复习与企业真题
专题1-14：谈学习方法和学习经验
学习方法
学习习惯
鸡汤
困惑

2. 抽丝剥茧话 Java

2.1 中国大学生就业形势

- 麦可思研究院发布了《**2022 年中国大学生就业报告**》，针对 2021 届毕业生收入较高的本科专业排行榜：

img

- 从国家统计局发布的 2021 年全国平均工资来看，不管在城镇非私营单位还是私营单位，IT 业均为最高。就业蓝皮书数据显示，应届毕业生在信息传输/软件和信息技术服务业就业月收入持续较高（2021 届本科：6781 元，高职：5023 元）。

2.2 IT 互联网是否依旧靠谱

过去不能代表未来！互联网是否依旧靠谱？！

数据来自：《清华大学：中国产业互联网发展指数 china industrial internet index（cii）》

image-20220523002428542

2014 年至 2018 年间，我国网民规模从 6.49 亿增长为 8.29 亿，增幅为 27.5%。同一时间段，全国移动互联网接入的流量却从 20.6EB 增长到了 711.1EB，增幅达 3352%（获取和处理的信息量大幅增加）。

随着 5G 技术进一步拓宽移动互联网的速度和容量，产业互联网将在消费型流量的基础上创造生产型流量，根据报告的预测，至 2024 年，全国移动互联网的接入流量将达到 127663.8EB， 流量规模达到 2018 年的 179.5 倍。

当下，5G、物联网、人工智能、产业互联网都是国家政策大方向，需要大量能与机器对话的中高端人才。

2.3 IT 行业岗位分析

image-20220522234544537

软件开发，是进入互联网 IT 圈最好的选择之一！

- 起始薪资高
- 工作环境好
- 涨薪幅度高
- 瓶颈出现晚
- 行业更公平

2.4 软件开发之 Java 开发

Java，

是学习JavaEE后台、大数据、Android开发的基石！

- **移动应用领域（集成 Android 平台）**：Java 在 Android 端是主要开发的语言，占有重要的地位。

image-20220511092649122

```java
GradationScrollView scrollview;
@Bind(R.id.tv_usercenter)
TextView tvUsercenter;
private Context mContext;
private int height;

@Nullable
@Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup
    mContext = getActivity();

    View view = View.inflate(mContext, R.layout.fragment_person, null
    ButterKnife.bind(this, view);

    return view;
}
  vto.addOnGlobalLayoutListener(() -> {
          rlHeader.getViewTreeObserver().removeGlobalOnLayoutListener(this);

          height = rlHeader.getHeight();

          scrollview.setScrollViewListener((scrollView, x, y, oldx, oldy) -> {

                  if (y <= 0) {
                      tvUsercenter.setBackgroundColor(Color.argb(0, 255, 0, 0));
                  } else if (y > 0 && y <= height) {
                      float scale = (float) y / height;
                      float alpha = 255 * scale;

                      tvUsercenter.setTextColor(Color.argb((int) alpha, 255, 255, 255));
                      tvUsercenter.setBackgroundColor(Color.argb((int) alpha, 255, 0, 0));
                  } else {
                      tvUsercenter.setBackgroundColor(Color.argb(255, 255, 0, 0));
                  }
          });
  });
}
```

20220608_160310

- **企业级应用领域（JavaEE 后台）：**用来开发企业级的应用程序，大型网站如淘宝、京东、12306，以及各大物流、银行、金融、社交、医疗、交通、各种 OA 系统等都是用 JavaEE 技术开发的。

image-20220511090528043

- **大数据分析、人工智能领域：**流行的大数据框架，如 Hadoop、Flink 都是用 Java 编写的。Spark 使用 Scala 编写，但可以用 Java 开发应用。

image-20220511091647358

```java
@InterfaceAudience.Public
@InterfaceStability.Stable
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

  /**
   * The <code>Context</code> passed on to the {@link Mappe
   */
  public abstract class Context
    implements MapContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
  }

  /**
   * Called once at the beginning of the task.
   */
  protected void setup(Context context
                       ) throws IOException, InterruptedExc
    // NOTHING
  }
```

Hadoop源码

```java
public class EventTimeSessionWindows extends MergingWindowAssigner<Object, TimeWindow> {
    private static final long serialVersionUID = 1L;

    protected long sessionTimeout;

    protected EventTimeSessionWindows(long sessionTimeout) {
        if (sessionTimeout <= 0) {
            throw new IllegalArgumentException(
                    "EventTimeSessionWindows parameters must satisfy 0 < size");
        }

        this.sessionTimeout = sessionTimeout;
    }

    @Override
    public Collection<TimeWindow> assignWindows(
            Object element, long timestamp, WindowAssignerContext context) {
        return Collections.singletonList(new TimeWindow(timestamp, timestamp + sessionTimeout));
```

Flink源码

Eversoft 公司在提到 2022 年 Java 发展趋势时写道：

Java 是用于开发大数据项目的最主流的语言。我们可以轻松地预测它也将在之后继续主导大数据！

- **游戏领域、桌面应用、嵌入式领域：**很多大型游戏的后台、桌面应用等也是 Java 开发的。

2.5 到底多少人在用 Java

2020 年，根据 IDC 的报告“Java Turns 25”显示，超过 900 万名开发人员（全球专职开发人员中的 69%）在使用 Java——比其他任何语言都多。该报告指出，大多数企业业务服务都依靠 Java 来实现。

凭借自身不断提高平台性能、稳定性和安全性的能力，Java 一直是开发人员中最流行的编程语言，被誉为“宇宙第一语言”。

我想告诉大家：

“市场的需求比较大，市场的供给比较大”

“如果你在 Java 领域里持续积累 5-7 年以上，那么你至少会成为这个行业的一个专家！”

2.6 八卦一下 Java 程序员

image-20220602105134523

还可以是：

2.7 Java 系列课程体系

- 见 02_学习路线图之《Java 中高级程序员全程学习路线图.xmind》

2.8 Java 职业晋升路线图

薪资数据统计来源：拉勾网

3. 计算机的硬件与软件

3.1 计算机组成：硬件+软件

3.2 CPU、内存与硬盘

- **CPU（Central Processing Unit，中央处理器）**
  - 人靠大脑思考，电脑靠 CPU 来运算、控制。
- **硬盘（Hard Disk Drive）**
  - 计算机最主要的存储设备，容量大，断电数据不丢失。

- 正常分类：机械硬盘（HDD）、固态硬盘（SSD）以及混合硬盘（SSHD）
- 固态硬盘在开机速度和程序加载速度远远高于机械硬盘，缺点就是贵，所有无法完全取代机械硬盘。

- **内存（Memory）**
  - 负责硬盘上的数据与 CPU 之间数据交换处理
  - 具体的：保存从硬盘读取的数据，提供给 CPU 使用；保存 CPU 的一些临时执行结果，以便 CPU 下次使用或保存到硬盘。
  - 断电后数据丢失。

3.3 输入设备：键盘输入

- 熟悉指法

image-20220517100223784

不熟悉键盘的小伙伴，可以“金山打字通”走起了。坚决杜绝二指禅！！

image-20220517100543176

4. 软件相关介绍

4.1 什么是软件

软件，即一系列按照特定顺序组织的计算机数据和指令的集合。有**系统软件**和**应用软件**之分。

Pascal 之父 Nicklaus Wirth：“Programs = Data Structures + Algorithms”

系统软件：

应用软件：

4.2 人机交互方式

- **图形化界面**（Graphical User Interface，GUI），这种方式简单直观，使用者易于接受，容易上手操作。
- **命令行方式**（Command Line Interface，CLI），需要在控制台输入特定的指令，让计算机完成一些操作。需要记忆一些指令，较为麻烦。

img

4.3 常用的 DOS 命令

DOS（Disk Operating System，磁盘操作系统）是 Microsoft 公司在 Windows 之前推出的一个操作系统，是单用户、单任务（即只能执行一个任务）的操作系统。现在被 Windows 系统取代。

对于 Java 初学者，学习一些 DOS 命令，会非常有帮助。

进入 DOS 操作窗口：

- 按下 Windows+R 键盘，打开运行窗口，输入 cmd 回车，进入到 DOS 的操作窗口。

image-20220520100110104

常用指令：

- 操作 1：进入和回退

操作	说明
盘符名称:	盘符切换。E:回车，表示切换到 E 盘。
dir	列出当前目录下的文件以及文件夹
cd 目录	进入指定单级目录。
cd 目录 1\目录 2...	进入指定多级目录。cd atguigu\JavaSE
cd ..	回退到上一级目录。
cd \ 或 cd /	回退到盘符目录。

- 操作 2：增、删

操作	说明
md 文件目录名	创建指定的文件目录。
rd 文件目录名	删除指定的文件目录（如文件目录内有数据，删除失败）

- 操作 3：其它

操作	说明
cls	清屏。
exit	退出命令提示符窗口。
← →	移动光标
↑↓	调阅历史操作命令
Delete 和 Backspace	删除字符

5. 计算机编程语言

5.1 计算机语言是什么

- **语言**：是人与人之间用于沟通的一种方式。例如：中国人与中国人用普通话沟通。而中国人要和英国人交流，可以使用英语或普通话。
- **计算机编程语言**，就是人与计算机交流的方式。人们可以使用编程语言对计算机下达命令，让计算机完成人们需要的功能。
- 计算机语言有很多种。如：C 、C++、Java、Go、JavaScript、Python，Scala 等。

体会：语言 = 语法 + 逻辑

5.2 计算机语言简史

- **第一代：机器语言（相当于人类的石器时代）**
  - 1946 年 2 月 14 日，世界上第一台计算机 ENAC 诞生，使用的是最原始的穿孔卡片。这种卡片上使用的是用二进制代码表示的语言，与人类语言差别极大，这种语言就称为机器语言。比如一段典型的机器码：

1. 0000,0000,000000010000 代表 LOAD A, 16
2. 0000,0001,000000000001 代表 LOAD B, 1
3. 0001,0001,000000010000 代表 STORE B, 16

- 这种语言本质上是计算机能识别的唯一语言，人类很难理解。可以大胆想象"此时的程序员 99.9%都是异类！"

- **第二代：汇编语言（相当于人类的青铜&铁器时代）**
  - 使用英文缩写的助记符来表示基本的操作，这些助记符构成了汇编语言的基础。比如：LOAD、MOVE 等，使人更容易使用。因此，汇编语言也称为符号语言。
  - 优点：能编写高效率的程序
  - 缺点：汇编语言是面向机器的，不同计算机机型特点不同，因此会有不同的汇编语言，彼此之间不能通用。程序不易移植，较难调试。

image-20220309223725671

- 比起机器语言，汇编大大进步了。尽管还是复杂，用起来容易出错，但在计算机语言发展史上是机器语言向更高级的语言进化的桥梁。目前仍然应用于工业电子编程领域、软件的加密解密、计算机病毒分析等。

- **第三代：高级语言（相当于人类的信息时代）**
  - 高级语言发展于 20 世纪 50 年代中叶到 70 年代，是一种接近于人们使用习惯的程序设计语言。它允许程序员使用接近日常英语的指令来编写程序，程序中的符号和算式也与日常用的数学式子差不多，接近于自然语言和数学语言，容易为人们掌握。比如：

  - 高级语言独立于机器，有一定的通用性；计算机不能直接识别和执行用高级语言编写的程序，需要使用编译器或者解释器，转换为机器语言才能被识别和执行。

image-20211218092630678

此外，高级语言按照程序设计方法的不同，又分为：面向过程的语言、面向对象的语言。

- C、Pascal、Fortran 面向过程的语言
- C++面向过程/面向对象
- Java 跨平台的纯面向对象的语言
- C#、Python、JavaScript、Scala...

目前以及可预见的将来，计算机语言仍然处于“第三代高级语言”阶段。但是不管是什么语言，最后都要向机器语言靠近，因为 CPU 只认识 0 和 1。

5.3 计算机语言排行榜

TIOBE （https://www.tiobe.com/tiobe-index/）是一个流行编程语言排行，每月更新。排名权重基于世界范围内 工程师数量，Google、Bing、Yahoo! 、Wikipedia、Amazon、Youtube 和百度这些主流的搜索引擎，也将作为排名权重的参考指标。

Oct 2022	Oct 2021	Change	Programming Language	Ratings	Change
1	1		Python	17.08%	+5.81%
2	2		C	15.21%	+4.05%
3	3		Java	12.84%	+2.38%
4	4		C++	9.92%	+2.42%
5	5		C#	4.42%	-0.84%
6	6		Visual Basic	3.95%	-1.29%
7	7		JavaScript	2.74%	+0.55%
8	10	^	Assembly language	2.39%	+0.33%
9	9		PHP	2.04%	-0.06%
10	8	v	SQL	1.78%	-0.39%
11	12	^	Go	1.27%	-0.01%
12	14	^	R	1.22%	+0.03%
13	29	^^	Objective-C	1.21%	+0.76%
14	13	v	MATLAB	1.18%	-0.02%
15	17	^	Swift	1.05%	-0.06%

计算机语言走势

5.4 编程语言，该学哪个？

image-20220310151657860

网传的编程语言鄙视链：

C > C++ > Java > C# > JavaScript > VB > Python > PHP > ALL

- C 语言：万物之源
- 诞生于 1972 年，由 AT&T 公司旗下贝尔实验室创建完成，用于构建 Unix 操作系统。
- 偏向计算机底层操作。可以说，现今这个世界上几乎所有重要的软件都跟 C 有直接和间接的关系，操作系统、网络、硬件驱动等。
- 优势：几乎所有现代编程语言都脱胎于 C
- 劣势：相当陡的学习曲线；不支持面向对象编程
- C++语言：难学的语言
- 诞生于 1983 年，作为 C 语言的增强方案、升级版本。C++是 C 语言的超集，C 语言的大部分知识也适用于 C++。
- 用途：windows 或 MacOS UI、office 全家桶、主流的浏览器、Oracle、MySQL、著名的游戏引擎（如星际争霸、魔兽世界）等

- 优势：很多公司都用 C++ 开发核心架构，如 Google、腾讯、百度、阿里云等；面向对象性
- 劣势：体系极为庞大，这是世界目前来说最复杂也是最难的编程语言。
- C#语言：背靠大树的语言
- 诞生于 2000 年，一款强大而灵活的编程语言。靠着微软这棵大树，所以可以充分发挥微软.NET 平台的优势，也是多年来 windows 平台的一门主流编程语言。
- 用途：windows 桌面应用开发、Windows Azure、游戏开发
- PHP 语言：最好的语言？
- 诞生于 1994 年，一款服务器端脚本语言。最初表示个人主页（Personal Home Page）
- PHP 语法和 C 类似，有很多的模板和框架，简单易懂，也许你可以在短短几天做出 web app。它主要用于 web 端，快速建站网络开发
- 劣势：学习门槛太低；其代码运行速度低于大部分编程语言党争对手
- Python：易学的语言
- 诞生于 1991 年，一种面向对象的语言，虽然运行效率不高，但是开发效率非常高。
- Python 被称为胶水语言，哪里都可以用。
- JavaScript 语言：前端为王
- 诞生于 1995 年，网景公司开发完成。
- JavaScript 是 JavaScript 是目前所有主流浏览器上唯一支持的脚本语言。在前端开发中，占有不可替代的地位。
- Java 语言：需求旺盛
- 创建于 1995 年，Java 是一种面向对象、基于类的编程语言。
- Java 可能是目前运用最广的项目语言。代码稳定性超过 C 和 C++，生产力远超 C 和 C++。有 JVM 在，可以轻松地跨平台。
- 具有强大的开源开发工具，大量的开源共享库。
- Java 拥有世界上数量最多的程序员，最不缺人。
- Go 语言：夹缝中生存
- Go 语言现在很受关注，它是取代 C 和 C++ 的另一门有潜力的语言。
  - C 语言太原始了，C++ 太复杂了，Java 太高级了，所以 Go 语言就在这个夹缝中出现了。

- Go 语言已成为云计算领域事实上的标准语言，尤其是在 Docker/Kubernetes 等项目中。
- Go 语言语法特别简单，你有了 C 和 C++ 的基础，学习 Go 的学习成本基本为零。
- Go 社区从 Java 社区移植了各种优秀的框架或库。

总结：

- 程序设计语言有很多种，每种语言都是为了实现某个特定的目的而发明的。
- **没有“最好”的语言**，只有在特定场景下相对来说，最适合的语言而已。
- 如果你掌握了一种编程语言，也会更容易上手其它的编程语言。关键是学习如何使用程序设计方法来解决问题。这也是本套课程的主旨。
- Talk is cheap，Show me the code。

6. Java 语言概述

6.1 Java 概述

- 是 SUN(Stanford University Network，斯坦福大学网络公司) 1995 年推出的一门高级编程语言。
- 是一种面向 Internet 的编程语言。Java 一开始富有吸引力是因为 Java 程序可以在 Web 浏览器中运行。这些 Java 程序被称为 Java 小程序（applet），内嵌在 HTML 代码中。
- 伴随着互联网的迅猛发展，以及 Java 技术在 web 方面的不断成熟，已经成为 Web 应用程序的首选开发语言。

6.2 Java 语言简史

起步阶段：

1991 年，Sun 公司的工程师小组想要设计一种语言，应用在电视机、电话、闹钟、烤面包机等家用电器的控制和通信。由于这些设备的处理能力和内存都很有限，并且不同的厂商会选择不同的中央处理器(CPU)，因此这种语言的关键是代码短小、紧凑且与平台无关（即不能与任何特定的体系结构捆绑在一起）。

Gosling 团队率先创造了这个语言，并命名为“Oak"（起名的原因是因为他非常喜欢自己办公室外的橡树)。后因智能化家电的市场需求没有预期的高，Sun 公司放弃了该项计划。

随着 20 世纪 90 年代互联网的发展，Sun 公司发现该语言在互联网上应用的前景，于是改造了 Oak，于 1995 年 5 月以 Java 的名称正式发布。（Java 是印度尼西亚爪哇岛的英文名称，因盛产咖啡而闻名。）

发展阶段：

发行版本	发行时间	备注
Java 1.0	1996.01.23	Sun 公司发布了 Java 的第一个开发工具包
Java 1.1	1997.02.19	JavaOne 会议召开，创当时全球同类会议规模之最。
Java 1.2	1998.12.08	Java 拆分成：J2SE（标准版）、J2EE（企业版）、J2ME（小型版）
Java 1.3	2000.05.08	
Java1.4	2004.02.06	
Java 5.0	**2004.09.30**	**①版本号从 1.4 直接更新至 5.0；②平台更名为 JavaSE、JavaEE、JavaME**
Java 6.0	2006.12.11	2009.04.20 Oracle 宣布收购 SUN 公司
	2009.04.20	Oracle 公司收购 SUN，交易价格 74 亿美元。
Java 7.0	2011.07.02	
Java 8.0	**2014.03.18**	**此版本是继 Java 5.0 以来变化最大的版本。是长期支持版本（LTS）**
Java 9.0	2017.09.22	①此版本开始，**每半年更新一次**；②Java 9.0 开始不再支持 windows 32 位系统
Java 10.0	2018.03.21	
Java 11.0	**2018.09.25**	**JDK 安装包取消独立 JRE 安装包，是长期支持版本（LTS）**
Java 12.0	2019.03.19	

发行版本	发行时间	备注
…	…	
Java17.0	**2021.09**	**发布 Java 17.0，版本号也称为 21.9，是长期支持版本。**
…	…	
Java19.0	2022.09	发布 Java19.0，版本号也称为 22.9。

6.3 Java 之父

Java之父是詹姆斯·高斯林(James Gosling)

image-20220310152923961

- 詹姆斯·高斯林(James Gosling)先生以“Java 技术之父”而闻名于世。他是 Java 技术的创始人，他亲手设计了 Java 语言，并开发了 Java 编译器和 Java 虚拟机，使 Java 成为了世界上最流行的开发语言。
- James Gosling 于 1984 年加入 Sun 公司，并一直服务于 Sun 公司，直至 2010 年前后，Sun 被 Oracle 并购而加入 Oracle，担任客户端软件集团的首席技术官; 2010 年 4 月从 Oracle 离职。

6.3 公司八卦

- **SUN 与 Oracle**

SUN 是一家极具创新能力的公司，2001 年 “9.11”以前，SUN 公司市值超过 1000 亿美元。 但是没能利用 Java 构建一个强有力、可变现的生态系统，没打好 Java 这张牌。此后，互联网泡沫破裂，硬件需求大幅减少，它的市值在一个月之内跌幅超过 90%。**SUN 公司的成长用了 20 年，而衰落只用了 1 年！**

image-20220309230923533

- **Oracle 与 Google**

  Google 和 Oracle 的侵权事件：

  2010 年 8 月，Oracle 起诉 Google 的 Android 系统侵权，要求赔偿 26 亿美元。

- Oracle 认为 Google 的代码中使用了 Java 的 37 个 API，并且认为 Google 是故意为之，因为这样做的好处是可以让更多的 Java 程序员更容易接受 Android 的代码。
- Oracle 认为 Android 中有 9 行代码直接抄袭了 Java 的实现。这 9 行牛气哄哄的代码都出自一人之手，他就是 Java 大牛------Joshua Bloch。

  2018 年 3 月，美国联邦巡回上诉法院裁决，谷歌侵犯了甲骨文的版权，支付高达 88 亿美元的赔偿金。

  2021 年 4 月，美国最高法院给出了最终裁决：谷歌胜诉，其代码属于“合理使用”的范畴。为期十多年的软件行业“第一版权案”落幕。

```
//Code In OpenJDK / Android :
private static void rangeCheck(int arrayLen, int fromIndex, int toIn
dex) {
    if (fromIndex > toIndex)
        throw new IllegalArgumentException("fromIndex(" + fromIndex +
                 ") > toIndex(" + toIndex+")");
    if (fromIndex < 0)
        throw new ArrayIndexOutOfBoundsException(fromIndex);
    if (toIndex > arrayLen)
        throw new ArrayIndexOutOfBoundsException(toIndex);
}
```

6.5 Java 技术体系平台

- **Java SE(Java Standard Edition)标准版**
  - 支持面向桌面级应用（如 Windows 下的应用程序）的 Java 平台，即定位个人计算机的应用开发。
  - 包括用户界面接口 AWT 及 Swing，网络功能与国际化、图像处理能力以及输入输出支持等。
  - 此版本以前称为 J2SE
- **Java EE(Java Enterprise Edition)企业版**
  - 为开发企业环境下的应用程序提供的一套解决方案，即定位在服务器端的 Web 应用开发。
  - JavaEE 是 JavaSE 的扩展，增加了用于服务器开发的类库。如：Servlet 能够延伸服务器的功能，通过请求-响应的模式来处理客户端的请求；JSP 是一种可以将 Java 程序代码内嵌在网页内的技术。
  - 版本以前称为 J2EE
- **Java ME(Java Micro Edition)小型版**
  - 支持 Java 程序运行在移动终端（手机、机顶盒）上的平台，即定位在消费性电子产品的应用开发
  - JavaME 是 JavaSE 的内伸，精简了 JavaSE 的核心类库，同时也提供自己的扩展类。增加了适合微小装置的类库：javax.microedition.io.*等。
  - 此版本以前称为 J2ME

注意：

Android 开发不等同于 Java ME 的开发

7. Java 开发环境搭建（掌握）

7.1 什么是 JDK、JRE

- **JDK** (Java Development Kit)：是 Java 程序开发工具包，包含 JRE 和开发人员使用的工具。
- **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。

如下是 Java 8.0 Platform：

image-20220310200731185

小结：

JDK = JRE + 开发工具集（例如 Javac 编译工具等）

JRE = JVM + Java SE 标准类库

7.2 JDK 版本选择

image-20221108221057540

- 自 Java 8 版本发布以来，其后的每次更新，都会有小伙伴高呼：**Java8 YYDS！**
- 论坛的声音：“你发任你发，我用 Java 8！”

数据说话 1：

JRebel 于 2022 年 4 月前后发布了《2022 年 Java 生态系统状况报告》，报告中提到使用 Java11 的占比最多，Java 8 紧随其后，如下图。而此前 2020 年的报告显示，Java8 占比达到了 84.48%。

我的分析：

G1 是最受欢迎的 GC 算法。Java 11 及更高版本的 G1 收集器是默认的 GC，而 Java 8 中并不是。出于对 G1 的喜爱，很多开发者才会选择抛弃 Java 8。

数据说话 2：

此外，某美国软件开发商在对近千名专业的 Java 开发者调研后，发布的《2022 年 Java 开发者生产力报告》称：八年前发布的 Java 8 依然是 Java 中应用最广泛的版本，占比 37%，其次是 Java 11，占比 29% 。

高斯林说话：

image-20220428151952049

Spring 框架说话：

在 Java 17 正式发布之前，Java 开发框架 Spring 率先在官博宣布，Spring Framework 6 和 Spring Boot 3 计划在 2022 年第四季度实现总体可用性的高端基线：

- Java 17+(来自 Spring Framework 5.3.x 线中的 Java 8-17)
- Jakarta EE 9+（来自 Spring 框架 5.3.x 线中的 Java EE 7-8）

Spring 官方说明：https://spring.io/blog/2022/01/20/spring-boot-3-0-0-m1-is-now-available

spring

Why Spring　Learn　Projects　Training

Spring Blog　All Posts　Engineering　Releases　News and Events

Spring Boot 3.0.0-M1 is now available

RELEASES | PHIL WEBB | JANUARY 20, 2022 0 COMMENT

On behalf of the team and everyone who has contributed, I'm happy to announce that Spring Boot 3.0.0-M1 has been released and is now available from https://repo.spring.io/milestone.

This milestone starts our exciting journey to the next generation of the Spring Framework and raises our baseline from Java 8 to Java 17. We are planning to release a new milestone of Spring Boot 3.0 every two months. M2 should arrive on March 24 and we are planning on a GA release in late November.

意味着：springboot3.0 是需要用 java17 和 spring6.0 为基础建设。如果从企业选型最新 springboot3.0 作为架构来说，它搭配 jdk17 肯定是标配了。

针对于 Spring 6，官网的说明会弃用 java8 以 9 为最低版本，而且兼容 tomcat10+。

7.3 JDK 的下载

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

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

image-20211019110551338

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

image-20211019110843394

（3）选择 Download Java 按钮后

（4）如果想安装 Java8 可以选择如下位置：

image-20211019111150970

（5）选择 Accept License Agreement，

image-20211019111252989

（6）注册或登录后下载：

image-20211019111922387

（7）如果需要安装其它版本，可以选择 Java archive：

image-20220801004606762

接着进行选择下载即可：

image-20220801004648997

7.4 JDK 的安装

- 安装说明
  - 傻瓜式安装，下一步即可。
  - **建议：安装路径不要有中文或者空格等特殊符号。**

- **本套课程会同时安装 JDK8 和 JDK17，并以 JDK17 为默认版本进行讲解。**

- 安装步骤：

（1）双击 `jdk-17_windows-x64_bin.exe` 文件，并单击下一步，如图所示：

（2）修改安装路径，单击更改，如图所示：

（3）将安装路径修改为 D:\develop_tools\jdk\jdk-17.0.2\，并单击下一步，如图所示：

（4）稍后几秒，安装完成，如图所示：

Java(TM) SE Development Kit 17.0.2 (64-bit) - 进度
Java
ORACLE
状态:
正在复制新文件

7.5 配置 path 环境变量

7.5.1 理解 path 环境变量

什么是 path 环境变量？

答：window 操作系统执行命令时，所要搜寻的路径。

为什么配置 path？

答：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。

以 JDK 为例演示

我们在 `C:\Users\songhk` 目录下使用 javac 命令，结果如下：

我们在 JDK 的安装目录的 bin 目录下使用 javac 命令，结果如下：

image-20220607113708022

我们不可能每次使用 java.exe，javac.exe 等工具的时候都进入到 JDK 的安装目录下，太麻烦了。这时就需要配置 path 环境变量。

7.5.2 JDK8 配置方案 1：只配置 path

- 步骤：

（1）打开桌面上的计算机，进入后在左侧找到此电脑，单击鼠标右键，选择属性，如图所示：

（2）选择高级系统设置，如图所示：

（3）在高级选项卡，单击环境变量，如图所示：

（4）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：

（5）点击新建，填入 `D:\develop_tools\jdk\jdk1.8.0_271\bin`，并将此值上移到变量列表的首位。如图所示：

- 编辑模式 1：

image-20220607114102135

- 编辑模式 2：（注意，结尾需要有英文模式下的;）

（6）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` 或 `java` 命令或 `java -version`，运行成功。

image-20220607114432286

image-20220607114517652

7.5.3 JDK8 配置方案 2：配置 JAVA_HOME+path（推荐）

- 步骤：

（1）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：

（2）选择高级系统设置，如图所示：

（3）在高级选项卡，单击环境变量，如图所示：

（4）在系统变量中，单击新建，创建新的环境变量，如图所示：

image-20220310225245208

（5）变量名输入 `JAVA_HOME`，变量值输入 `D:\develop_tools\jdk\jdk1.8.0_271` ，单击确定，如图所示：

image-20220607114800372

（6）选中 `Path` 环境变量，双击或者点击编辑，如图所示：

（7）在变量值的最前面，键入`%JAVA_HOME%\bin`。如图所示：

image-20220607114953758

注意：强烈建议将%JAVA_HOME%\bin 声明在 path 环境变量中所有变量的最前面！

（8）环境变量配置完成，**重启** DOS 命令行，在任意目录下输入 `javac` 或 `java` 命令或 `java -version`，运行成功。

C:\WINDOWS\system32\cmd.exe

```
Microsoft Windows [版本 10.0.18363.1500]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\songhk>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
```

image-20220607114432286

```
C:\Users\songhk>java -version
java version "1.8.0_271"
Java(TM) SE Runtime Environment (build 1.8.0_271-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.271-b09, mixed mode)

C:\Users\songhk>
```

image-20220607114517652

我想说：

有的书籍、论坛、视频上还提到配置 classpath，用于指名 class 文件识别的路径。其实是没必要的，反而建议大家如果配置了 classpath 环境变量，要删除。对于初学者，反而不友好。

小结如下：

7.5.4 JDK17 配置方案：自动配置

不管大家有没有提前安装 JDK8 或其它版本 JDK，在我们安装完 JDK17 之后，理应按 JDK8 的方式配置 path 环境变量。但是，我们发现在安装完 JDK17 以后，配置环境变量之前，执行 CMD 指令：

image-20220607115700870

竟然成功了！而且是 17.0.2 版本。因为 JDK17 在安装之后，自动进行了环境变量的配置。如下：

image-20220607115840388

思考：如果你仍然希望在 JDK8 下开发 Java 程序？如何做呢？

8. 开发体验：HelloWorld（掌握）

JDK 安装完毕，我们就可以开发第一个 Java 程序了，习惯性的称为：HelloWorld。

8.1 开发步骤

Java 程序开发三步骤：**编写**、**编译**、**运行**。

- 将 Java 代码**编写**到扩展名为 .java 的源文件中
- 通过 javac.exe 命令对该 java 文件进行**编译**，生成一个或多个字节码文件
- 通过 java.exe 命令对生成的 class 文件进行**运行**

8.2 编写

（1）在 D:\JavaSE\chapter01 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。

image-20220801010222222

（2）用记事本或 editplus 等文本编辑器打开（虽然记事本也可以，但是没有关键字颜色标识，不利于初学者学习）

（3）在文件中输入如下代码，并且保存：

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

- **友情提示 1：每个字母和符号必须与示例代码一模一样，包括大小写在内。**
- 友情提示 2：

image-20220310230618659

image-20220521173215013

第一个 HelloWord 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。

8.3 编译

在 DOS 命令行中，进入 D:\JavaSE\chapter01 目录，使用 javac 命令进行编译。

使用文件资源管理器打开 D:\JavaSE\chapter01 目录，然后在地址栏输入 cmd。

image-20220801010435144

命令：

`javac Java 源文件名.后缀名 java`

举例：

`javac HelloWorld.java`

image-20220801010840640

编译成功后，命令行没有任何提示。打开 `D:\JavaSE\chapter01` 目录，发现产生了一个新的文件 `HelloChina.class`，该文件就是编译后的文件，是 Java 的可运行文件，称为**字节码文件**，有了字节码文件，就可以运行程序了。

8.4 运行

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 主类名字`

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 HelloChina
```

错误演示：

java HelloChina.class

image-20220801011104381

9. Java 开发工具

9.1 都有哪些开发 Java 的工具

- 级别一：文本开发工具

image-20220513185907022

- 级别二：集成开发环境（Integrated Development Environment，IDE）

  把代码编写，编译，执行，调试等多种功能综合到一起的开发工具。

image-20220513190655773

9.2 如何选择

前期我们先使用文本开发工具，培养代码感，利于公司笔、面试。

后期我们使用 IDE，提供更强大的功能支持。

10. HelloWorld 案例常见错误

10.1 拼写问题

- 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

10.2 编译、运行路径问题

举例 1：

```
D:\>javac Test1.java
javac: 找不到文件: Test1.java
用法: javac <options> <source files>
-help 用于列出可能的选项
```

- 源文件名不存在或者写错
- 当前路径错误
- 后缀名隐藏问题

举例 2：

```
D:\>java Test1
错误: 找不到或无法加载主类 Test1
```

- 类文件名写错，尤其文件名与类名不一致时，要小心
- 类文件不在当前路径下，或者不在 classpath 指定路径下

10.3 语法问题

举例 1：

```
D:\>javac Test.java
Test.java:1: 错误: 类Test1是公共的, 应在名为 Test1.java 的文件中声明
public class Test1{
       ^
1 个错误
```

声明为 public 的类应与文件名一致，否知编译失败。

举例 2：

```
D:\>javac Test.java
Test.java:3: 错误: 需要';'
                System.out.println("hello")
                                          ^
1 个错误
```

编译失败，注意错误出现的行数，再到源代码中指定位置改错

10.4 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("灏氣 璋?");
                                   ^
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：

- 在 EditPlus 中可以将 Java 源文件另存为 ANSI 编码方式（中文操作系统下即为 GBK 字符集）

1658810752424

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
```

10.5 建议

- 注意缩进!
  - 一定要有缩进。缩进就像人得体的衣着一样！
  - 只要遇到{}就缩进，缩进的快捷键 tab 键。
- 必要的空格
  - 变量类型、变量、赋值符号、变量值之间填充相应空格，更美观。比如： int num = 10;

11. HelloWorld 小结

11.1 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

11.2 Java 程序的入口

Java 程序的入口是 main 方法

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

}
```

11.3 两种常见的输出语句

- **换行输出语句**：输出内容，完毕后进行换行，格式如下：

  ```
  System.out.println(输出内容);
  ```

- **直接输出语句**：输出内容，完毕后不做任何处理，格式如下

  ```
  System.out.print(输出内容);
  ```

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

直接输出语句，括号内什么都不写的话，编译报错

11.4 源文件名与类名

（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。但是不便于代码维护。

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。

12. 注释(comment)

image-20220610113151774

- 什么是注释？
  - 源文件中用于解释、说明程序的文字就是注释。
- 注释是一个程序员必须要具有的良好编程习惯。实际开发中，程序员可以先将自己的思想通过注释整理出来，再用代码去体现。

  程序员最讨厌两件事：

  一件是自己写代码被要求加注释

  另一件是接手别人代码，发现没有注释
- 不加注释的危害

- Java 中的注释类型：
  - 单行注释

    ```
    //注释文字
    ```

  - 多行注释

    ```
    /*
    注释文字 1
    注释文字 2
    注释文字 3
    */
    ```

  - 文档注释 (Java 特有)

    ```
    /**
      @author   指定 java 程序的作者
      @version   指定源文件的版本
    */
    ```

- 注释的作用
  - 它提升了程序的可阅读性。（不加注释的危害性，见图。）
  - 调试程序的重要方法。
- **具体使用 1：单行注释、多行注释**
  - 对于单行和多行注释，被注释的文字，不会不会出现在字节码文件中，进而不会被 JVM（java 虚拟机）解释执行。

- 多行注释里面不允许有多行注释嵌套。
- 一个段子

A：嘿 //是什么意思啊？
B：嘿.
A：呃 我问你//是什么意思？
B：问吧.
A：我刚才不是问了么？
B：啊？
A：你再看看记录...
B：看完了.
A：.....所以//是啥？
B：所以什么？
A：你存心耍我呢吧？
B：没有啊 你想问什么？
……
不断循环之后，A 一气之下和 B 绝交，自己苦学程序。
N 年之后，A 终于修成正果，回想起 B，又把聊天记录翻出来看，这时，他突然发现 B 没有耍他……
而他自己也不知道当年他问 B 的究竟是什么问题……

- **具体使用 2：文档注释（Java 特有）**
  - 文档注释内容可以被 JDK 提供的工具 javadoc 所解析，生成一套以网页文件形式体现的该程序的说明文档。
  - 操作方式。比如：

```
javadoc -d mydoc -author -version HelloWorld.java
```

尚硅谷

```
D:\code\day01>javadoc -d mydoc -author -version HelloWorld.java
正在加载源文件HelloWorld.java...
正在构造 Javadoc 信息...
正在创建目标目录: "mydoc\"
标准 Doclet 版本 1.8.0_131
正在构建所有程序包和类的树...
正在生成mydoc\HelloWorld.html...
正在生成mydoc\package-frame.html...
正在生成mydoc\package-summary.html...
正在生成mydoc\package-tree.html...
正在生成mydoc\constant-values.html...
正在构建所有程序包和类的索引...
正在生成mydoc\overview-tree.html...
正在生成mydoc\index-all.html...
正在生成mydoc\deprecated-list.html...
正在构建所有类的索引...
正在生成mydoc\allclasses-frame.html...
正在生成mydoc\allclasses-noframe.html...
正在生成mydoc\index.html...
正在生成mydoc\help-doc.html...
```

image-20220310232527578

案例：

```java
//单行注释
/*
多行注释
*/
/**
文档注释演示。这是我的第一个Java 程序！^_^
@author songhk
@version 1.0
*/
public class HelloWorld{

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

13. Java API 文档

- API （Application Programming Interface，应用程序编程接口）是 Java 提供的基本编程接口。

- Java 语言提供了大量的基础类，因此 Oracle 也为这些基础类提供了相应的说明文档，用于告诉开发者如何使用这些类，以及这些类里包含的方法。大多数 Java 书籍中的类的介绍都要参照它来完成，它是编程者经常查阅的资料。

- Java API 文档，即为 JDK 使用说明书、帮助文档。类似于：

新华字典

- 下载 API 文档：
  - 在线看：https://docs.oracle.com/en/java/javase/17/docs/api/index.html
  - 离线下载：https://www.oracle.com/java/technologies/javase-jdk17-doc-downloads.html

14. Java 核心机制：JVM

14.1 Java 语言的优缺点

Java 确实是从 C 语言和 C++语言继承了许多成份，甚至可以将 Java 看成是类 C 语言发展和衍生的产物。“青出于蓝，而胜于蓝”。

14.1.1 优点

- **跨平台性**：这是 Java 的核心优势。Java 在最初设计时就很注重移植和跨平台性。比如：Java 的 int 永远都是 32 位。不像 C++可能是 16，32，可能是根据编译器厂商规定的变化。
  - 通过 Java 语言编写的应用程序在不同的系统平台上都可以运行。“Write once , Run Anywhere”。
- 原理：只要在需要运行 java 应用程序的操作系统上，先安装一个 Java 虚拟机 (JVM ， Java Virtual Machine) 即可。由 JVM 来负责 Java 程序在该系统中的运行。

JVM 的跨平台性

Linux macOS Windows

不同平台
不同指令集

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)

- **面向对象性：**
  - 面向对象是一种程序设计技术，非常适合大型软件的设计和开发。面向对象编程支持封装、继承、多态等特性，让程序更好达到高内聚，低耦合的标准。
  - 由于 C++为了照顾大量 C 语言使用者而兼容了 C，使得自身仅仅成为了带类的 C 语言，多少影响了其面向对象的彻底性！Java 则是完全面向对象的语言。
- **健壮性：**吸收了 C/C++语言的优点，但去掉了其影响程序健壮性的部分（如指针、内存的申请与释放等），提供了一个相对安全的内存管理和访问机制。
- **安全性高：**

  Java 适合于网络/分布式环境，需要提供一个安全机制以防恶意代码的攻击。为了达到这个目标，在安全性方面 Java 投入了很大的精力，使 Java 可以很容易构建防病毒，防篡改的系统。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。
- **简单性：**

  Java 就是 C++语法的简化版，我们也可以将 Java 称之为“C++--”。比如：头文件，指针运算，结构，联合，操作符重载，虚基类等。
- **高性能：**
  - Java 最初发展阶段，总是被人诟病“性能低”；客观上，高级语言运行效率总是低于低级语言的，这个无法避免。Java 语言本身发展中通过虚拟机的优化提升了几十倍运行效率。比如，通过 JIT(JUST IN TIME)即时编译技术提高运行效率。 将一些“热点字节码”编译成本地机器码，并将结果缓存起来，在需要的时候重新调用。这样的话，使 Java 程序的执行效率大大提高，某些代码直接达到 C++的效率。
  - 因此，Java 低性能的短腿，已经被完全解决了。业界发展上，我们也看到很多 C++应用转到 Java 开发，很多 C++程序员转型为 Java 程序员。

14.1.2 缺点

- 语法过于复杂、严谨，对程序员的约束比较多，与 python、php 等相比入门较难。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。
- 并非适用于所有领域。在某些领域其他语言有更出色的表现，比如，Objective C、Swift 在 iOS 设备上就有着无可取代的地位。浏览器中的处理几乎完全由 JavaScript 掌控。Windows 程序通常都用 C++或 C#编写。Java 在服务器端编程和跨平台客户端应用领域则很有优势。

14.2 JVM 功能说明

JVM（Java Virtual Machine，Java 虚拟机）：是一个虚拟的计算机，是 Java 程序的运行环境。JVM 具有指令集并使用不同的存储区域，负责执行指令，管理数据、内存、寄存器。

image-20220310165805739

14.2.1 功能 1：实现 Java 程序的跨平台性

我们编写的 Java 代码，都运行在 **JVM** 之上。正是因为有了 JVM，才使得 Java 程序具备了跨平台性。

image-20220513191856727

使用 JVM 前后对比：

14.2.2 功能 2：自动内存管理(内存分配、内存回收)

- Java 程序在运行过程中，涉及到运算的数据的分配、存储等都由 JVM 来完成
- Java 消除了程序员回收无用内存空间的职责。提供了一种系统级线程跟踪存储空间的分配情况，在内存空间达到相应阈值时，检查并释放可被释放的存储器空间。
- GC 的自动回收，提高了内存空间的利用效率，也提高了编程人员的效率，很大程度上减少了因为没有释放空间而导致的内存泄漏。

面试题：

Java 程序还会出现内存溢出和内存泄漏问题吗？ Yes!

15. 章节案例

案例 1：个人信息输出

```
姓名：康师傅

性别：男
家庭住址：北京程序员聚集地：回龙观
```

```java
class Exercise1{
	public static void main(String[] args){
		System.out.println("姓名：康师傅");
		System.out.println();//换行操作
		System.out.println("性别：男");
		System.out.println("家庭住址：北京程序员聚集地：回龙观");
	}
}
```

案例 2：输出：心形

结合\n(换行)，\t(制表符)，空格等在控制台打印出如下图所示的效果。

```
	*		*
*	*	I love Java	*	*
	*		*
	*		*
		*		*
		*		*
		
```

```
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");

System.out.println("*");


System.out.print("*");
System.out.print("\t");
//System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("I love java");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.println("*");


System.out.print("\t");
System.out.print("*");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");

System.out.println("*");


System.out.print("\t");
System.out.print("\t");
System.out.print("*");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");
System.out.print("\t");

System.out.println("*");


System.out.print("\t");
```

```
            System.out.print("\t");
            System.out.print("\t");
            System.out.print("*");
            System.out.print("\t");
            System.out.print("\t");
            System.out.print("\t");

            System.out.println("*");


            System.out.print("\t");
            System.out.print("\t");
            System.out.print("\t");
            System.out.print("\t");
            System.out.print("*");
            System.out.print("\t");

            System.out.println("*");


            System.out.print("\t");
            System.out.print("\t");
            System.out.print("\t");
            System.out.print("\t");
            System.out.print("    ");
            System.out.print("*");

      }

}
```

方式二：

```
class Exercise3{
      public static void main(String[] args){

            System.out.print("\t"+"*"+"\t\t\t\t\t\t\t\t\t\t\t\t\t"+"*"+"\
t"+"\n");
            System.out.print("*"+"\t\t"+"*"+"\t\t\t\t"+"I love Java"+"\
t\t\t\t"+"*"+"\t\t\t"+"*"+"\n");
            System.out.print("\t"+"*"+"\t\t\t\t\t\t\t\t\t\t\t\t\t"+"*"+"\
t"+"\n");
            System.out.print("\t\t"+"*"+"\t\t\t\t\t\t\t\t\t\t\t"+"*"+"\t\
t"+"\n");
            System.out.print("\t\t\t"+"*"+"\t\t\t\t\t\t\t\t\t"+"*"+"\t"+"
\n");
            System.out.print("\t\t\t\t"+"*"+"\t\t\t\t\t\t\t"+"*"+""+"\t"+
"\n");
```

```
            System.out.print("\t\t\t\t\t"+"*"+"\t\t\t\t"+"*"+""+"\t\t"+
"\n");
            System.out.print("\t\t\t\t\t\t"+"*"+"\t\t"+"*"+""+"\t\t"+"\
n");
            System.out.print("\t\t\t\t\t\t\t"+"*"+"\n");


        }

}
```

第 02 章_变量与运算符

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 关键字（keyword）（掌握）

- 定义：被 Java 语言赋予了特殊含义，用做专门用途的字符串（单词）
  - 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
- 特点：全部关键字都是小写字母。
- 关键字比较多，不需要死记硬背，学到哪里记到哪里即可。
- 官方地址：https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Java Language Keywords

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	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

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

说明：

1. 关键字一共 50 个，其中 const 和 goto 是保留字(reserved word)。
2. 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。

用于定义数据类型的关键字				
class	interface	enum	byte	short
int	long	float	double	char
boolean	void			
用于定义流程控制的关键字				
if	else	switch	case	default
while	do	for	break	continue
return				
用于定义访问权限修饰符的关键字				
private	protected	public		

image-20220310234557454

用于定义类，函数，变量修饰符的关键字				
abstract	final	static	synchronized	
用于定义类与类之间关系的关键字				
extends	implements			
用于定义建立实例及引用实例，判断实例的关键字				
new	this	super	instanceof	
用于异常处理的关键字				
try	catch	finally	throw	throws
用于包的关键字				
package	import			
其他修饰符关键字				
native	strictfp	transient	volatile	assert
*** 用于定义数据类型值的字面值**				
true	false	null		

image-20220310234607198

2. 标识符(identifier)（掌握）

Java 对各种变量、方法和类等要素命名时使用的字符序列称为标识符。

技巧：凡是自己可以起名字的地方都叫标识符。

标识符的命名规则（必须遵守的硬性规定）：

> 由 26 个英文字母大小写，0-9 ，_或 $ 组成
> 数字不可以开头。

> 不可以使用关键字和保留字，但能包含关键字和保留字。
> Java 中严格区分大小写，长度无限制。
> 标识符不能包含空格。

练习：miles, Test, a++, --a, 4#R, $4, #44, apps, class, public, int, x, y, radius

标识符的命名规范（建议遵守的软性要求，否则容易被鄙视）：

> 包名：多单词组成时所有字母都小写：xxxyyyzzz。
  例如：java.lang、com.atguigu.bean

> 类名、接口名：多单词组成时，所有单词的首字母大写：XxxYyyZzz
  例如：HelloWorld，String，System 等

> 变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写：xxxYyyZzz
  例如：age,name,bookName,main,binarySearch,getName

> 常量名：所有字母都大写。多单词时每个单词用下划线连接：XXX_YYY_ZZZ
  例如：MAX_VALUE,PI,DEFAULT_CAPACITY

注意 1：在起名字时，为了提高阅读性，要尽量有意义，“见名知意”。

注意 2：java 采用 unicode 字符集，因此标识符也可以使用汉字声明，但是不建议使用。

更多细节详见《代码整洁之道.pdf》《Java 开发手册 1.7.0（嵩山版）》

3. 变量（掌握）

3.1 为什么需要变量

一花一世界，如果把一个程序看做一个世界或一个社会的话，那么变量就是程序世界的花花草草。即，变量是程序的基本组成单位。

3.2 初识变量

- 变量的概念：
  - 内存中的一个存储区域
  - 该区域的数据可以在同一类型范围内不断变化
  - 变量是程序中最基本的存储单元。包含三个要素：变量的数据类型、变量名和存储的值

image-20220311000744867

- 变量的作用：用于在内存中保存数据
- 使用变量注意：
  - Java 中每个变量必须先声明，后使用
  - 使用变量名来访问这块区域的数据
  - 变量的作用域：其定义所在的一对{}内
  - 变量只有在其作用域内才有效
  - 同一个作用域内，不能定义重名的变量

3.3 Java 中的数据类型

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。

3.4 变量的使用

3.4.1 变量的声明

格式：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。

3.4.2 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已经声明的变量赋值

```
age = 18;
weight = 109;
gender = '女';
```

也可以将变量的声明和赋值一并执行

```
boolean isBeauty = true;
String name = "迪丽热巴";
```

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;
```

说明：

- 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；

4、变量可以反复赋值

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

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

3.4 小结：变量的三要素

1、数据类型

尚硅谷

- 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

3、值

- 基本数据类型的变量：存储数据值

image-20220514152216148

- 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）

4. 基本数据类型介绍

4.1 整数类型：byte、short、int、long

Java 各整数类型有固定的表数范围和字段长度，不受具体 OS 的影响，以保证 java 程序的可移植性。

Java 的整型常量默认为 int 型，声明 long 型常量须后加‘l’或‘L’

Java 程序中变量通常声明为 int 型，除非不足以表示较大的数，才使用 long

类　型	占用存储空间	表数范围
byte	1字节=8bit位	-128 ~ 127
short	2字节	-2^{15} ~$2^{15}-1$
int	4字节	-2^{31} ~ $2^{31}-1$ (约21亿)
long	8字节	-2^{63} ~ $2^{63}-1$

image-20220311001553945

image-20220520111756274

4.1.1 计算机存储单位（掌握）

- **字节（Byte）：**是计算机用于计量存储容量的基本单位，一个字节等于 8 bit。
- **位（bit）：**是数据存储的最小单位。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

4.2 浮点类型：float、double

- 与整数类型类似，Java 浮点类型也有固定的表数范围和字段长度，不受具体操作系统的影响。

- 浮点型常量有两种表示形式：
  - 十进制数形式：如：5.12 512.0f .512 (必须有小数点）
  - 科学计数法形式:如：5.12e2 512E2 100E-2
- float:单精度，尾数可以精确到 7 位有效数字。很多情况下，精度很难满足需求。
- double:双精度，精度是 float 的两倍。通常采用此类型。
- Java 的浮点型常量默认为 double 型，声明 float 型常量，须后加‘f’或‘F’。

类 型	占用存储空间	表数范围
单精度float	4字节	-3.403E38 ~ 3.403E38
双精度double	8字节	-1.798E308 ~ 1.798E308

image-20220311001749699

康师傅有话说：

1. 并不是所有的小数都能可以精确的用二进制浮点数表示。二进制浮点数不能精确的表示 0.1、0.01、0.001 这样 10 的负次幂。
2. 浮点类型 float、double 的数据不适合在不容许舍入误差的金融计算领域。如果需要精确数字计算或保留指定位数的精度，需要使用 BigDecimal 类。
3. 不要使用浮点数进行比较！很多新人甚至很多理论不扎实的有工作经验的程序员也会犯这个错误！需要比较请使用 BigDecimal 类

测试：

```
//测试1:
float f = 0.1f;
double d = 1.0/10;
System.out.println(f==d);//结果为false

//测试2  解释见《为什么0.1 + 0.2 不等于0.3.doc》
System.out.println(0.1 + 0.2);//0.30000000000000004

//测试3:
float f1 = 123123123f;
float f2 = f1+1;
System.out.println(f1);
System.out.println(f2);
System.out.println(f1 == f2);
```

4.3 字符类型：char

- char 型数据用来表示通常意义上“字符”(2 字节)
- Java 中的所有字符都使用 Unicode 编码，故一个字符可以存储一个字母，一个汉字，或其他书面语的一个字符。
- 字符型变量的三种表现形式：
  - 字符常量是用单引号(‘ ’)括起来的单个字符。例如：char c1 = 'a'; char c2 = '中'; char c3 = '9';
  - 直接使用 Unicode 值来表示字符型常量：‘\uXXXX’。其中，XXXX 代表一个十六进制整数。如：\u000a 表示 \n。
  - Java 中还允许使用转义字符‘\’来将其后的字符转变为特殊字符型常量。例如：char c3 = ‘\n’; // '\n'表示换行符

转义字符	说明	Unicode 表示方式
\n	换行符	\u000a
\t	制表符	\u0009
\"	双引号	\u0022
\'	单引号	\u0027
\\	反斜线	\u005c
\b	退格符	\u0008
\r	回车符	\u000d

- char 类型是可以进行运算的。因为它都对应有 Unicode 码。

4.3.1 了解：ASCII 码

- 在计算机内部，所有数据都使用二进制表示。每一个二进制位（bit）有 0 和 1 两种状态，因此 8 个二进制位就可以组合出 256 种状态，这被称为一个字节（byte）。一个字节一共可以用来表示 256 种不同的状态，每一个状态对应一个符号，就是 256 个符号，从 0000000 到 11111111。
- ASCII 码：上个世纪 60 年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为 ASCII 码。ASCII 码一共规定了 128 个字符的编码，比如空格“SPACE”是 32（二进制 00100000），大写的字母 A 是 65（二进制 01000001）。这 128 个符号（包括 32 个不能打印出来的控制符号），只占用了一个字节的后面 7 位，最前面的 1 位统一规定为 0。

- 缺点：
  - 不能表示所有字符。
  - 相同的编码表示的字符不一样：比如，130 在法语编码中代表了 é，在希伯来语编码中却代表了字母 Gimel (ג)

ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符
0	NUT	32	(space)	64	@	96	、
1	SOH	33	!	65	A	97	a
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DCI	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	X	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	/	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	—	127	DEL

4.3.2 了解：ISO-8859-1

西欧字符集，占用 1 个字节。

4.3.3 了解：GB2312、GBK、GB18030

GB2312：大陆使用最早、最广的简体中文字符集。占用 2 个字节。

GBK：GB2312 的扩展，可以表示简体、繁体中文。占用 2 个字节。

GB18030：最新 GBK 的扩展，可以表示汉字、维吾尔文、藏文等中华民族字符。占用 2 个字节

ANSI：（美国国家标准学会、AMERICAN NATIONAL STANDARDS INSTITUTE: ANSI）获取你当前操作系统的编码格式 ---> 一般我们用的 GBK

4.3.4 了解：Unicode 码

- 乱码：世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。
- Unicode：一种编码，也称为统一码、万国码等。将世界上所有的符号都纳入其中，为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。使用 Unicode 没有乱码的问题。
- Unicode 的缺点：这里就有三个问题。
- 一个是，我们已经知道，英文字母只用一个字节表示就够了，每个英文字母前都必然有二到三个字节是 0，这对于存储空间来说是极大的浪费。
- 第二个问题是如何才能区别 Unicode 和 ASCII？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

4.3.5 了解：UTF-8

- Unicode 是字符集，UTF-32/ UTF-16/ UTF-8 是三种字符编码方案。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。其中，UTF-8 是在互联网上使用最广的一种 Unicode 的实现方式。
- UTF-8 是一种变长的编码方式。它可以使用 1-4 个字节表示一个符号，根据不同的符号而变化字节长度。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 Unicode 字符集≠UTF-8 编码方式）。

- UTF-8 的编码规则：
  - 对于单字节的 UTF-8 编码，该字节的最高位为 0，其余 7 位用来对字符进行编码（等同于 ASCII 码）。
  - 对于多字节的 UTF-8 编码，如果编码包含 n 个字节，那么第一个字节的前 n 位为 1，第一个字节的第 n+1 位为 0，该字节的剩余各位用来对字符进行编码。在第一个字节之后的所有的字节，都是最高两位为"10"，其余 6 位用来对字符进行编码。
- 举例

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

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]

- 小结

image-20220525165610847

4.4 布尔类型：boolean

- boolean 类型用来判断逻辑条件，一般用于程序流程控制：
  - if 条件控制语句；
  - while 循环控制语句；
  - do-while 循环控制语句；
  - for 循环控制语句；
- **boolean 类型数据只允许取值 true 和 false，无 null。**
  - 不可以使用 0 或非 0 的整数替代 false 和 true，这点和 C 语言不同。
  - Java 虚拟机中没有任何供 boolean 值专用的字节码指令，Java 语言表达所操作的 boolean 值，在编译之后都使用 java 虚拟机中的 int 数据类型来代替：true 用 1 表示，false 用 0 表示。———《java 虚拟机规范 8 版》
- 举例：

```
boolean isFlag = true;

if(isFlag){
    //true 分支
}else{
    //false 分支
}
```

康师傅有话说：

Less is More！建议不要这样写：if (isFlag = = true)，只有新手才如此。关键也很容易写错成 if(isFlag = true)，这样就变成赋值 isFlag 为 true 而不是判断！老鸟的写法是 if (isFlag)或者 if (!isFlag)

5. 基本数据类型间运算规则（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：自动类型提升和强制类型转换。`boolean` 类型不参与。

5.1 自动类型提升

规则：将取值范围小的类型自动提升为取值范围大的类型 。

image-20220523162200891

基本数据类型的转换规则如图所示：

（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
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
```

```
//byte bigB = 130;//错误，右边的整数常量值超过byte 范围
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';
int i = c1 + c2;//至少需要使用int 类型来接收
System.out.println(c1 + c2);//113
```

练习：

设 x 为 float 型变量，y 为 double 型变量，a 为 int 型变量，b 为 long 型变量，c 为 char 型变量，则表达式
x + y * a / x + b / y + c 的值类型为：
A. int　B. long　C. double　D. char

5.2 强制类型转换

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。

image-20220523162407722

规则：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，自动类型提升是 Java 自动执行的，而强制类型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。这种转换有风险，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。这种情况的强制类型转换是没有风险的。

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

练习：判断是否能通过编译

```
1）short  s = 5;
   s = s-2;                        //判断:
2）  byte b = 3;
     b = b + 4;                    //判断:
     b = (byte)(b+4);              //判断:
3）char c = ‘a’;
   int  i = 5;
   float d = .314F;
   double result = c+i+d;          //判断:
4）  byte b = 5;
     short s = 3;
     short t = s + b;              //判断:
```

5.3 基本数据类型与字符串类型的运算

5.3.1 字符串类型：String

- String 不是基本数据类型，属于引用数据类型
- 使用方式与基本数据类型一致。例如：String str = “abcd”;

5.3.2 运算规则

1、任意八种基本数据类型的数据与 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);//正确的，后面才能讲到，借助包装类的方法才能转
```

练习：

```
String str1 = 4;                        //判断对错:
String str2 = 3.5f + “”;                //判断 str2 对错:
System.out.println(str2);               //输出:
System.out .println(3+4+“Hello!”);      //输出:
System.out.println(“Hello!”+3+4);       //输出:
System.out.println(‘a’+1+“Hello!”);     //输出:
System.out.println(“Hello”+‘a’+1);      //输出:
```

6. 计算机底层如何存储数据

计算机世界中只有二进制，所以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。

010101

世界上有 10 种人，认识和不认识二进制的。

6.1 进制的分类

- **十进制（decimal）**
  - 数字组成：0-9
  - 进位规则：满十进一
- **二进制（binary）**
  - 数字组成：0-1
  - 进位规则：满二进一，以 0b 或 0B 开头
- **八进制（octal）：很少使用**
  - 数字组成：0-7
  - 进位规则：满八进一，以数字 0 开头表示
- **十六进制**
  - 数字组成：0-9，a-f
  - 进位规则：满十六进一，以 0x 或 0X 开头表示。此处的 a-f 不区分大小写

6.2 进制的换算举例

十进制	二进制	八进制	十六进制

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

6.3 二进制的由来

二进制，是计算技术中广泛采用的一种数制，由德国数理哲学大师莱布尼茨于 1679 年发明。

二进制数据是用 0 和 1 两个数码来表示的数。它的基数为 2，进位规则是“逢二进一”。

二进制广泛应用于我们生活的方方面面。比如，广泛使用的摩尔斯电码（Morse Code），它由两种基本信号组成：短促的点信号“·”，读“滴”；保持一定时间的长信号“–”，读“嗒”。然后，组成了 26 个字母，从而拼写出相应的单词。

image-20220520105721126

我们偶尔会看到的：SOS，即为：

morsecode

6.4 二进制转十进制

二进制如何表示整数？

- 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。
  - 正数：最高位是 0
  - 负数：最高位是 1
- 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

二进制转十进制：权相加法

针对于 byte 数据举例来说：

- 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 000000000 00011001（原码）
正数：25   00000000 00000000 000000000 00011001（反码）
正数：25   00000000 00000000 000000000 00011001（补码）

负数：-25  10000000 00000000 000000000 00011001（原码）
负数：-25  11111111 11111111 111111111 11100110（反码）
负数：-25  11111111 11111111 111111111 11100111（补码）
```

为什么要使用原码、反码、补码表示形式呢？

计算机辨别“符号位”显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: 1-1 = 1 + (-1) = 0 , 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了。

$$1-1 = 1 + (-1) = [0000\ 0001]_{原} + [1000\ 0001]_{原} = [0000\ 0001]_{补} + [1111\ 1111]_{补} = [0000\ 0000]_{补} = [0000\ 0000]_{原}$$

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 111 ==> 1~127

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

```
0000 0000 ==>0

1000 0000 ==> -128（特殊规定）=-127-1
```

6.5 十进制转二进制

十进制转二进制：除 2 取余的逆

image-20220312000042595

6.6 二进制与八进制、十六进制间的转换

二进制转八进制

image-20220312000233863

二进制转十六进制

image-20220312000251113

八进制、十六进制转二进制

image-20220312000341297

6.7 各进制间的转换

image-20220312000143438

练习：

1.将以下十进制数转换为十六进制和二进制
123 256 87 62

2.将以下十六进制数转换为十进制和二进制
0x123 0x25F 0x38 0x62

7. 运算符（Operator）（掌握）

运算符是一种特殊的符号，用以表示数据的运算、赋值和比较等。

运算符的分类：

- 按照功能分为：算术运算符、赋值运算符、比较(或关系)运算符、逻辑运算符、位运算符、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
比较(或关系)运算符（6 个）	>、>=、<、<=、==、!=
逻辑运算符（6 个）	&、|、^、!、&&、||
位运算符（7 个）	&、|、^、~、<<、>>、>>>
条件运算符（1 个）	(条件表达式)?结果 1:结果 2
Lambda 运算符（1 个）	->（后面学）

- 按照操作数个数分为：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

尚硅谷

7.1 算术运算符

运算符	运算	范例	结果
+	正号	+3	3
-	负号	b=4; -b	-4
+	加	5+5	10
-	减	6-4	2
*	乘	3*4	12
/	除	5/5	1
%	取模(取余)	7%5	2
++ ++	自增（前）：先运算后取值 自增（后）：先取值后运算	a=2;b=++a; a=2;b=a++;	a=3;b=3 a=3;b=2
- - - -	自减（前）：先运算后取值 自减（后）：先取值后运算	a=2;b=- -a a=2;b=a- -	a=1;b=1 a=1;b=2
+	字符串连接	“He”+”llo”	“Hello”

举例 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？
            System.out.println(a % b);// 3

        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，余数时 1     (-2)*(-2)+1 = 5
            //-5%2  ==>商是-2，余数是-1    (-2)*2+(-1) = -4-1=-5
      }
}
```

举例 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的话，+就是拼接的意思

```java
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
	}
}
```

举例 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++ 。

```java
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
        ①i 先自增 i=3
        ②再取i 的值“3”放操作数栈
         （3）再算i++
        ①取i 的值“3”放操作数栈
        ②i 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。**

练习 1：算术运算符：自加、自减

```
public class SignTest{
        public static void main(String[] args){
                int i1 = 10;
                int i2 = 20;
                int i = i1++;
                System.out.print("i="+i);
                System.out.println("i1="+i1);
                i = ++i1;
                System.out.print("i="+i);
                System.out.println("i1="+i1);
                i = i2--;
                System.out.print("i="+i);
                System.out.println("i2="+i2);
                i = --i2;
                System.out.print("i="+i);
                System.out.println("i2="+i2);
        }
}
```

练习 2：

```
System.out.println("5+5="+5+5); //打印结果是？ 5+5=55 ?
```

练习 3：

随意给出一个整数，打印显示它的个位数，十位数，百位数的值。
格式如下：
数字 xxx 的情况如下：
个位数：
十位数：
百位数：

例如：
数字 153 的情况如下：
个位数：3
十位数：5
百位数：1

练习 4：下列程序的输出结果是

```
int i = 2;
i = i++;
int j = i++;
System.out.println(i + "," + j);
```

7.2 赋值运算符

- 符号：=
  - 当“=”两侧数据类型不一致时，可以使用自动类型转换或使用强制类型转换原则进行处理。
  - 支持连续赋值。
- 扩展赋值运算符： +=, -=, *=, /=, %=

赋值运算符	符号解释
+=	将符号左边的值和右边的值进行相加操作，最后将结果赋值给左边的变量
-=	将符号左边的值和右边的值进行相减操作，最后将结果赋值给左边的变量
*=	将符号左边的值和右边的值进行相乘操作，最后将结果赋值给左边的变量
/=	将符号左边的值和右边的值进行相除操作，最后将结果赋值给左边的变量
%=	将符号左边的值和右边的值进行取余操作，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);
    }
}
```

思考 1：

```
short s = 3;
s = s+2;  //①
s += 2;   //②

//①和②有什么区别？
```

思考 2：

```
int i = 1;
i *= 0.1;
System.out.println(i);//
i++;
System.out.println(i);//
```

思考 3：

```
int m = 2;
int n = 3;
n *= m++;   //n = n * (m++)
System.out.println("m=" + m);
System.out.println("n=" + n);
```

思考 4：

```
int n = 10;
n += (n++) + (++n);  //n = n + (n++) + (++n)
System.out.println(n);
```

尚硅谷

7.3 关系运算符/比较运算符

运算符	运算	范例	结果
==	相等于	4==3	false
!=	不等于	4!=3	true
<	小于	4<3	false
>	大于	4>3	true
<=	小于等于	4<=3	false
>=	大于等于	4>=3	true
instanceof	检查是否是类的对象	“Hello” instanceof String	true

- 比较运算符的结果都是 boolean 型，也就是要么是 true，要么是 false。
- 比较运算符“==”不能误写成“=”。

思考：

```
boolean b1 = false;
//区分好==和=的区别。
if(b1=true)
      System.out.println("结果为真");
else
      System.out.println("结果为假");
```

7.4 逻辑运算符

- 逻辑运算符，是用来连接两个布尔类型值的运算符（!除外），运算结果也是 boolean 值 true 或者 false

a	b	a&b	a&&b	a|b	a||b	!a	a^b
true	true	true	true	true	true	false	false
true	false	false	false	true	true	false	true
false	true	false	false	true	true	true	true
false	false	false	false	false	false	true	false

image-20220312001943403

- 结果说明：

- & 和 &&：表示"且"关系，当符号左右两边布尔值都是 true 时，结果才能为 true。否则，为 false。
- | 和 || ：表示"或"关系，当符号两边布尔值有一边为 true 时，结果为 true。当两边都为 false 是，结果为 false
- ! ：表示"非"关系，当变量布尔值为 true 时，结果为 false。当变量布尔值为 false 时，结果为 true。
- ^ ：当符号左右两边布尔值不同时，结果为 true。当两边布尔值相同时，结果为 false。
    - 理解：异或，追求的是“异”！

- 逻辑运算符用于连接布尔型表达式，在 Java 中不可以写成 3<x<6，应该写成 x>3 & x<6 。
- “&”和“&&”的区别：
    - 单&时，左边无论真假，右边都进行运算；
    - 双&时，如果左边为真，右边参与运算，如果左边为假，那么右边不参与运算。
- “|”和“||”的区别同理，||表示：当左边为真，右边不参与运算。

```java
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));
```

```java
            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);
    }
}
```

练习 1：区分 & 和 &&

```java
int x = 1;
int y=1;

if(x++==2 & ++y==2){
      x =7;
}
System.out.println("x="+x+",y="+y);

int x = 1,y = 1;

if(x++==2 && ++y==2){
      x =7;
}
System.out.println("x="+x+",y="+y);
```

练习 2：区分 | 和 ||

```java
int x = 1,y = 1;

if(x++==1 | ++y==1){
      x =7;
}
System.out.println("x="+x+",y="+y);
```

```java
int x = 1,y = 1;

if(x++==1 || ++y==1){
      x =7;
}
System.out.println("x="+x+",y="+y);
```

练习 3：程序输出

```java
class  Test  {
      public static void main (String []  args)  {
            boolean x=true;
            boolean y=false;
            short z=42;
            //if(y == true)
            if( (z++==42) && (y=true) )z++;
            if((x=false) || (++z==45))  z++;

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

//结果为:
//z= 46
```

7.5 位运算符

位运算符		
运算符	运算	范例
<<	左移	3 << 2 = 12 --> 3*2*2=12
>>	右移	3 >> 1 = 1 --> 3/2=1
>>>	无符号右移	3 >>> 1 = 1 --> 3/2=1
&	与运算	6 & 3 = 2
|	或运算	6 | 3 = 7
^	异或运算	6 ^ 3 = 5
~	取反运算	~6 = -7

注意：无<<<

image-20220313174721111

位运算符的细节	
<<	空位补0，被移除的高位丢弃，空缺位补0。
>>	被移位的二进制最高位是0，右移后，空缺位补0； 最高位是1，空缺位补1。
>>>	被移位二进制最高位无论是0或者是1，空缺位都用0补。
&	二进制位进行&运算，只有1&1时结果是1，否则是0;
|	二进制位进行 | 运算，只有0 | 0时结果是0，否则是1;
^	相同二进制位进行 ^ 运算，结果是0；1^1=0 , 0^0=0 不相同二进制位 ^ 运算结果是1。1^0=1 , 0^1=1
~	正数取反，各二进制码按补码各位取反 负数取反，各二进制码按补码各位取反

image-20220312002506339

- 位运算符的运算过程都是基于二进制的补码运算

（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*2 的 **4** 次= 3*16 = 48

```
/*
3的二进制: 0000 0000 0000 0000 0000 0000 0000 0011
3<<4:      0000 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*2 的 **4** 次= -3*16 = -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 1101 0000   右边补4个0
        反码: 1111 1111 1111 1111 1111 1111 1100 1111
        原码: 1000 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 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 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 的 4 次 = -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 1011   右边移出去4位
        反码:  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 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
        补码:  0000 1111 1111 1111 1111 1111 1111 1011
        反码:  0000 1111 1111 1111 1111 1111 1111 1011
        原码:  0000 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
 */
```

无符号右移
左边补4个0

右边移出去4位

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 1001
      反码: 1111 1111 1111 1111 1111 1111 1111 0110
      补码: 1111 1111 1111 1111 1111 1111 1111 0111
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0111
-9&7:      0000 0000 0000 0000 0000 0000 0000 0111
      补码: 0000 0000 0000 0000 0000 0000 0000 0111
      反码: 0000 0000 0000 0000 0000 0000 0000 0111
      原码: 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 1001
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0111
9 | 7:      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 1001
      反码: 1111 1111 1111 1111 1111 1111 1111 0110
      补码: 1111 1111 1111 1111 1111 1111 1111 0111
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0111
-9|7:      1111 1111 1111 1111 1111 1111 1111 0111
      补码: 1111 1111 1111 1111 1111 1111 1111 0111
      反码: 1111 1111 1111 1111 1111 1111 1111 0110
      原码: 1000 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 ^ 1 结果为 0
- 1 ^ 0 结果为 1
- 0 ^ 1 结果为 1
- 0 ^ 0 结果为 0

```
9 ^ 7  //结果为14
```

```
/*
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 1110
*/
```

```
-9 ^ 7 //结果为-16
```

```
/*
-9的二进制:
      原码: 1000 0000 0000 0000 0000 0000 0000 1001
      反码: 1111 1111 1111 1111 1111 1111 1111 0110
      补码: 1111 1111 1111 1111 1111 1111 1111 0111
7的二进制: 0000 0000 0000 0000 0000 0000 0000 0111
-9^7:      1111 1111 1111 1111 1111 1111 1111 0000
      补码: 1111 1111 1111 1111 1111 1111 1111 0000
      反码: 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 1001 取反
~9:        1111 1111 1111 1111 1111 1111 1111 0110
    补码: 1111 1111 1111 1111 1111 1111 1111 0110
    反码: 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 1001
        反码: 1111 1111 1111 1111 1111 1111 1111 0110
        补码: 1111 1111 1111 1111 1111 1111 1111 0111 取反
  ~-9:        0000 0000 0000 0000 0000 0000 0000 1000
        补码: 0000 0000 0000 0000 0000 0000 0000 1000
        反码: 0000 0000 0000 0000 0000 0000 0000 1000
        原码: 0000 0000 0000 0000 0000 0000 0000 1000
     */
```

练习 1：

snipaste_20220312_002549

练习 2：体会 m = k ^ n = (m ^ n) ^ n

image-20220312002736157

练习 3：

```
int a = 8,b = 33;
int result1 = a >>> b;
```

```
int result2 = (a >>> b) | 1;
System.out.println("result = " + result1 + ",result2 = " + result2);
```

练习 4：下列程序的输出结果是

```
int a = 0x12;
int b = a << 1;
System.out.println(b);
```

7.6 条件运算符

- 条件运算符格式：

(条件表达式)？表达式 1:表达式 2

image-20220312002841945

- 表达式 1 和表达式 2 为同种或兼容的类型

```
public static void main(String[] args) {
    int i = (1==2 ? 100 : 200);
    System.out.println(i);//200
    int j = (3<=4 ? 500 : 600);
    System.out.println(j);//500
}

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"  );
```

```
    }
}
```

练习 1：

获取两个数中的较大数

练习 2：

获取三个数中的较大数

练习 3：

历史中，刘关张桃园三结义大家非常熟悉了。现有三个童鞋，年龄如下：

刘小备：23 岁

关小羽：22 岁

张小飞：21 岁

获取三人中最大的年龄

7.7 运算符优先级

运算符有不同的优先级，所谓优先级就是在表达式运算中的运算符顺序。下面列出了包括分隔符在内的所有运算符的优先级顺序，上一行中的运算符总是优先于下一行的。

优先级	运算符说明	Java 运算符
1	括号	`()`、`[]`、`{}`
2	正负号	`+`、`-`
3	单元运算符	`++`、`--`、`~`、`!`
4	乘法、除法、求余	`*`、`/`、`%`
5	加法、减法	`+`、`-`
6	移位运算符	`<<`、`>>`、`>>>`
7	关系运算符	`<`、`<=`、`>=`、`>`、`instanceof`
8	等价运算符	`==`、`!=`
9	按位与	`&`
10	按位异或	`^`
11	按位或	`|`
12	条件与	`&&`
13	条件或	`||`

优先级	运算符说明	Java 运算符
14	三元运算符	? :
15	赋值运算符	=、+=、-=、*=、/=、%=
16	位赋值运算符	&=、|=、<<=、>>=、>>>=

康师傅有话说：

1. 不要把一个表达式写得过于复杂，如果一个表达式过于复杂，则把它分成几步来完成。例如：

(num1 + num2) * 2 > num3 && num2 > num3 ? num3 : num1 + num2;

1. 不要过多的依赖运算的优先级来控制表达式的执行顺序，这样可读性太差，尽量使用()来控制、改变表达式的执行顺序
2. 逻辑与、逻辑或、逻辑非的优先级一定要熟悉！（逻辑非 > 逻辑与 > 逻辑或）。如：

   a || b && c 的运算结果是：a || (b && c)，而不是(a|| b) && c

第 03 章_流程控制语句

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

- 流程控制语句是用来控制程序中各语句执行顺序的语句，可以把语句组合成能完成一定功能的小逻辑模块。
- 其流程控制方式采用结构化程序设计中规定的三种基本流程结构，即：
  - **顺序结构**
    - 程序从上到下逐行地执行，中间没有任何判断和跳转。
  - **分支结构**
    - 根据条件，选择性地执行某段代码。
    - 有 if...else 和 switch-case 两种分支语句。
  - **循环结构**
    - 根据循环条件，重复性的执行某段代码。

- 有 while、do...while、for 三种循环语句。
- 注：JDK5.0 提供了 foreach 循环，方便的遍历集合、数组元素。

1. 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);
    }
}
```

Java 中定义成员变量时采用合法的前向引用。如：

```
public static void main(String[] args) {
      int num1 = 12;
```

```
        int num2 = num1 + 2;
}
```

错误形式：

```
public static void main(String[] args) {
        int num2 = num1 + 2;
        int num1 = 12;
}
```

2. 分支语句

2.1 if-else 条件判断结构

2.1.1 单分支条件判断：if

格式：

```
if(条件表达式) {
        语句块;
}
```

执行流程：

1. 首先判断条件表达式看其结果是 true 还是 false
2. 如果是 true 就执行语句块
3. 如果是 false 就不执行语句块

尚硅谷

案例：定义两个整数，分别为 small 和 big，如果第一个整数 small 大于第二个整数 big，就交换。输出显示 small 和 big 变量的值。

```
public class TestIf01 {
    public static void main(String[] args) {
        int small = 10;
        int big = 9;

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

2.1.2 双分支条件判断：if...else

格式：

```
if(条件表达式) {
        语句块 1;
}else {
        语句块 2;
}
```

执行流程：

1. 首先判断条件表达式看其结果是 true 还是 false
2. 如果是 true 就执行语句块 1
3. 如果是 false 就执行语句块 2

案例：定义一个整数，判定是偶数还是奇数

```
public class TestIfElse02 {
    public static void main(String[] args){
        int a = 10;

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

2.1.3 多分支条件判断：if...else if

格式：

```
if (条件表达式1) {
      语句块1;
} else if (条件表达式2) {
      语句块2;
}
...
}else if (条件表达式n) {
      语句块n;
} else {
      语句块n+1;
}
```

执行流程：

1. 首先判断关系表达式 1 看其结果是 true 还是 false
2. 如果是 true 就执行语句块 1，然后结束当前多分支
3. 如果是 false 就继续判断关系表达式 2 看其结果是 true 还是 false
4. 如果是 true 就执行语句块 2，然后结束当前多分支
5. 如果是 false 就继续判断关系表达式...看其结果是 true 还是 false

...

n. 如果没有任何关系表达式为 true，就执行语句块 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 TestIfElseIf03 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入成绩[0,100]: ");
```

```java
        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();
    }
}
```

```java
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

```java
import java.util.Scanner;

public class TestIfElseIf04 {
    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

2.1.4 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 TestNestIfElse05 {
    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();
```

```
    }
}
```

2.1.5 小结

- 条件表达式必须是布尔表达式（关系表达式或逻辑表达式）、布尔变量
- 语句块只有一条执行语句时，一对{}可以省略，但建议保留
- if-else 语句结构，根据需要可以嵌套使用
- 当 if-else 结构是“多选一”时，最后的 else 是可选的，根据需要可以省略
- 当多个条件是“互斥”关系时，条件判断语句及执行语句间顺序无所谓 当多个条件是“包含”关系时，“小上大下 / 子上父下”

2.1.6 案例

案例 1：

```java
public class AgeTest {
    public static void main(String args[]) {
        int age = 75;
        if (age < 0) {
            System.out.println("不可能！");
        } else if (age > 250) {
            System.out.println("是个妖怪！");
        } else {
            System.out.println("人家芳龄 " + age + " ,马马乎乎啦！");
        }
    }
}
```

案例 2：

- 需求

岳小鹏参加 Java 考试，他和父亲岳不群达成承诺：
如果：
成绩为 100 分时，奖励一辆 BMW；
成绩为(80，99]时，奖励一台 iphone 13 pro max；
当成绩为[60,80]时，奖励一个 iPad；
其它时，什么奖励也没有。
请从键盘输入岳小鹏的期末成绩，并加以判断

案例 3：

- 需求

编写程序：由键盘输入三个整数分别存入变量 num1、num2、num3，对它们进行排序(使用 if-else if-else),并且从小到大输出。

2.1.7 练习

练习 1：

```
//1)对下列代码，若有输出，指出输出结果。
int x = 4;
int y = 1;
if (x > 2) {
       if (y > 2)
            System.out.println(x + y);
                System.out.println("atguigu");
} else
       System.out.println("x is " + x);
```

练习 2：

```
boolean b = true;
//如果写成if(b=false)能编译通过吗？如果能，结果是？
if(b == false)
      System.out.println("a");
else if(b)
      System.out.println("b");
else if(!b)
      System.out.println("c");
else
      System.out.println("d");
```

练习 3：

编写程序，声明 2 个 int 型变量并赋值。判断两数之和，如果大于等于 50，打印“hello world!”

练习 4：

编写程序，声明 2 个 double 型变量并赋值。判断第一个数大于 10.0，且第 2 个数小于 20.0，打印两数之和。否则，打印两数的乘积。

练习 5：

我家的狗 5 岁了，5 岁的狗相当于人类多大呢？其实，狗的前两年每一年相当于人类的 10.5 岁，之后每增加一年就增加四岁。那么 5 岁的狗相当于人类多少年龄呢？应该是：10.5 + 10.5 + 4 + 4 + 4 = 33 岁。

编写一个程序，获取用户输入的狗的年龄，通过程序显示其相当于人类的年龄。如果用户输入负数，请显示一个提示信息。

练习 6：

假设你想开发一个玩彩票的游戏，程序随机地产生一个两位数的彩票，提示用户输入一个两位数，然后按照下面的规则判定用户是否能赢。
1)如果用户输入的数匹配彩票的实际顺序，奖金 10 000 美元。
2)如果用户输入的所有数字匹配彩票的所有数字，但顺序不一致，奖金 3 000 美元。
3)如果用户输入的一个数字仅满足顺序情况下匹配彩票的一个数字，奖金 1 000 美元。
4)如果用户输入的一个数字仅满足非顺序情况下匹配彩票的一个数字，奖金 500 美元。
5)如果用户输入的数字没有匹配任何一个数字，则彩票作废。

提示：使用(int)(Math.random() * 90 + 10)产生随机数。
Math.random() : [0,1) * 90 ---> [0,90) + 10 ---> [10,100) ---> [10,99]

练习 7：

大家都知道，男大当婚，女大当嫁。那么女方家长要嫁女儿，当然要提出一定的条件：高：180cm 以上；富：财富 1 千万以上；帅：是。

如果这三个条件同时满足，则：“我一定要嫁给他!!!”
如果三个条件有为真的情况，则：“嫁吧，比上不足，比下有余。”
如果三个条件都不满足，则：“不嫁！”

提示：
Sysout(“身高： (cm))
scanner.nextInt();
Sysout(“财富： (千万))
scanner.nextDouble();
Sysout(“帅否： (true/false)) (是/否)
scanner.nextBoolean(); scanner.next(); “是”.equals(str)

练习 8：判断水的温度

如果大于 95℃，则打印“开水”；

如果大于 70℃且小于等于 95℃，则打印“热水”；

如果大于 40℃且小于等于 70℃，则打印“温水”；

如果小于等于 40℃，则打印“凉水”。

2.2 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;
   ]
}
```

执行流程图：

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：

- switch(表达式)中表达式的值必须是下述几种类型之一：byte，short，char，int，枚举 (jdk 5.0)，String (jdk 7.0)；
- case 子句中的值必须是常量，不能是变量名或不确定的表达式值；
- 同一个 switch 语句，所有 case 子句中的常量值互不相同；
- break 语句用来在执行完一个 case 分支后使程序跳出 switch 语句块；如果没有 break，程序会顺序执行到 switch 结尾
- default 子句是可任选的。同时，位置也是灵活的。当没有匹配的 case 时，执行 default

举例 1：

```
public class SwitchTest {
    public static void main(String args[]) {
        int i = 1;
        switch (i) {
            case 0:
                System.out.println("zero");
                break;
            case 1:
                System.out.println("one");
                break;
            default:
                System.out.println("default");
                break;
        }
    }
}
```

举例 2：

```
public class SwitchTest1 {
    public static void main(String args[]) {
        String season = "summer";
        switch (season) {
            case "spring":
                System.out.println("春暖花开");
                break;
            case "summer":
                System.out.println("夏日炎炎");
                break;
            case "autumn":
                System.out.println("秋高气爽");
                break;
            case "winter":
                System.out.println("冬雪皑皑");
                break;
            default:
```

```
                System.out.println("季节输入有误");
                break;
        }
    }
}
```

2.2.1 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();
```

```
    }
}
```

2.2.2 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。

案例：根据指定的月份输出对应季节

```java
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");
            }
```

2.2.3 Java12 之后 switch 新特性（后面讲）

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！");
};
```

2.2.4 练习

练习 1：

使用 **switch** 把小写类型的 char 型转为大写。只转换 a，b，c，d，e. 其它的输出“other”。

提示：`String word = scan.next();  char c = word.charAt(0); switch(c){}`

练习 2：

对学生成绩大于 **60** 分的，输出“合格”。低于 **60** 分的，输出“不合格”。

练习 3：

编写程序：从键盘上输入 **2022** 年的“month”和“day”，要求通过程序输出输入的日期为 **2022** 年的第几天。

练习 4：

从键盘分别输入年、月、日，判断这一天是当年的第几天

注：判断一年是否是闰年的标准：
1）可以被 4 整除，但不可被 100 整除
或
2）可以被 400 整除

练习 5：

使用 switch 语句改写下列 if 语句：

```
int a = 3;
int x = 100;
if(a==1)
      x+=5;
else if(a==2)
      x+=10;
```

```
else if(a==3)
      x+=16;
else
      x+=34;
```

练习 6：

编写程序：从键盘上读入一个学生成绩，存放在变量 score 中，根据 score 的值输出其对应的成绩等级：

```
score>=90            等级:  A
70<=score<90         等级:  B
60<=score<70         等级:  C
score<60             等级:  D
```

方式一：使用 if-else
方式二：使用 switch-case: score / 10:   0 - 10

练习 7：

编写一个程序，为一个给定的年份找出其对应的中国生肖。中国的生肖基于 12 年一个周期，每年用一个动物代表：rat、ox、tiger、rabbit、dragon、snake、horse、sheep、monkey、rooster、dog、pig。

提示：2021 年：牛   2021 % 12 == 5

尚硅谷

2.2.5 if 语句与 switch 语句比较

- if 语句优势
  - 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。
- switch 语句优势
  - 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。因为效率稍高。当条件是区间范围的判断时，只能使用 if 语句。
  - 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。
- 补充：使用 switch-case 的，都可以改写为 if-else。反之不成立。

案例 1：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 IfOrSwitchDemo3 {
    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();
    }
}
```

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");
    }
}
```

3. 循环语句

- 理解：循环语句具有在某些条件满足的情况下，反复执行特定代码的功能。
- 循环结构分类：
  - for 循环
  - while 循环
  - do-while 循环
- 循环结构四要素：
  - 初始化部分(init_statement)
  - 循环条件部分(test_exp)
  - 循环体部分(body_statement)
  - 迭代部分(alter_statement)

3.1 for 循环

3.1.1 基本格式

for 循环语句格式：

```
for（①初始化部分；②循环条件部分；④迭代部分）{
            ③循环体部分；
}
```

执行过程：①-②-③-④-②-③-④-②-③-④-.....-②

图示：

尚硅谷

image-20220315013023236

注意：

- for(;;)中的两个；是不能多也不能少
- ②循环条件部分为 boolean 类型表达式，当值为 false 时，退出循环
- ①初始化部分可以声明多个变量，但必须是同一个类型，用逗号分隔
- ④可以有多个变量更新，用逗号分隔

3.1.2 应用举例

1、使用 for 循环重复执行某些语句

举例：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);
        }
```

```
    }
}
```

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

举例：求 1-100 的累加和

```
public class Test02ForVariableScope {
    public static void main(String[] args) {
        //考虑变量的作用域
        int sum = 0;
        for (int i = 1; i <= 100 ; i++) {
//            int sum = 0;
            sum += i;
        }
//        System.out.println("i = " + i);
        System.out.println("sum = " + sum);
    }
}
```

3、结合分支结构使用

举例：编写程序从 1 循环到 150，并在每行打印一个值，另外在每个 3 的倍数行上打印出“foo”,在每个 5 的倍数行上打印“biz”,在每个 7 的倍数行上打印输出“baz”。

尚硅谷

```
1
2
3 foo
4
5 biz
6 foo
7 baz
8
9 foo
10 biz
11
12 foo
13
14 baz
15 foo biz
```

```
100 biz
101
102 foo
103
104
105 foo biz baz
106
107
108 foo
109
110 biz
111 foo
112 baz
113
114 foo
```

image-20220315013722962

4、结合 break 的使用

举例：输入两个正整数 m 和 n，求其最大公约数和最小公倍数。

比如：12 和 20 的最大公约数是 4，最小公倍数是 60。

3.1.3 练习

练习 1：打印 1~100 之间所有奇数的和

练习 2：打印 1~100 之间所有是 7 的倍数的整数的个数及总和（体会设置计数器的思想）

练习 3：输出所有的水仙花数，所谓水仙花数是指一个 3 位数，其各个位上数字立方和等于其本身。 例如： 153 = 1*1*1 + 3*3*3 + 5*5*5

```
public class ForTest3 {
      public static void main(String[] args) {
            //定义统计变量，初始化值是0
            int count = 0;

            //获取三位数，用for 循环实现
            for(int x=100; x<1000; x++) {
                  //获取三位数的个位，十位，百位
```

```
                int ge = x%10;
                int shi = x/10%10;
                int bai = x/10/10%10;

                //判断这个三位数是否是水仙花数，如果是，统计变量++
                if((ge*ge*ge+shi*shi*shi+bai*bai*bai) == x) {
            System.out.println("水仙花数：" + x);
                    count++;
                }
            }

            //输出统计结果就可以了
            System.out.println("水仙花数共有"+count+"个");
        }
}
```

拓展：打印出四位数字中个位+百位=十位+千位并且个位数为偶数，千位数为奇数的数字，并打印符合条件的数字的个数。

练习 4：

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n
um 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");
        }
    }
}
```

3.2 while 循环

3.2.1 基本格式

while 循环语句基本格式：

```
①初始化部分
while(②循环条件部分) {
    ③循环体部分;
    ④迭代部分;
}
```

执行过程：①-②-③-④-②-③-④-②-③-④-...-②

图示：

尚硅谷

image-20220315013023236

注意：

- while(循环条件)中循环条件必须是 boolean 类型
- 注意不要忘记声明④迭代部分。否则，循环将不能结束，变成死循环。
- for 循环和 while 循环可以相互转换

3.2.2 应用举例

举例 1：计算 100 以内自然数的和

```
public class WhileLoop {
    public static void main(String args[]) {
        int result = 0;
        int i = 1;
        while (i <= 100) {
            result += i;
            i++;
        }
        System.out.println("result=" + result);
```

```
    }
}
```

举例 2：从键盘输入整数，输入 0 结束，统计输入的正数、负数的个数。

```java
import java.util.Scanner;

public class Test05While {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        int positive = 0; //记录正数的个数
        int negative = 0;  //记录负数的个数
        int num = 1; //初始化为特殊值，使得第一次循环条件成立
        while(num != 0){
            System.out.print("请输入整数（0 表示结束）: ");
            num = input.nextInt();

            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();
    }
}
```

举例 3：猜数字游戏

随机生成一个 1-10 的整数，从键盘输入一个数，如果比随机数大，则打印"大了"；否则打印"小了"，循环输入，直到正确打印"恭喜您猜对了!"。

```java
public class GuessNumber {
    public static void main(String[] args) {
        int targetNumber = (int)(Math.random() * 10 + 1);
        Scanner input = new Scanner(System.in);
        System.out.println("请输入你猜的数字: ");
        int enterNumber = input.nextInt();

        while(enterNumber != targetNumber){
            if(enterNumber > targetNumber){
                System.out.println("大了");
            }else{
                System.out.println("小了");
            }
```

```
            System.out.println("继续输入数字:");
            enterNumber = input.nextInt();
        }
        System.out.println("恭喜你猜对了！ ");
    }
}
```

3.3 do-while 循环

3.3.1 基本格式

do-while 循环语句标准格式：

```
①初始化部分;
do{
        ③循环体部分
        ④迭代部分
}while(②循环条件部分);
```

执行过程：①-③-④-②-③-④-②-③-④-…-②

图示：

尚硅谷

image-20220512165558698

注意：

- while(循环条件)中循环条件必须是 boolean 类型
- do{}while();最后有一个分号
- 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

3.3.2 应用举例

举例 1：

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

```
        int result = 0, i = 1;
        do {
            result += i;
            i++;
        } while (i <= 100);
            System.out.println("result=" + result);
        }
}
```

举例 2：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();
    }
}
```

3.4 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 "无限"循环

最简单“无限” 循环格式：`while(true)`，`for(;;)`，无限循环存在的原因是并不知道循环多少次，需要根据循环体内部某些条件，来控制循环的结束。

举例：实现爱你到永远

```
public class EndlessFor1 {
    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 EndlessFor2 {
    public static void main(String[] args) {
        for (; true;){ //条件永远成立
            System.out.println("我爱你！");
        }
    }
}

public class EndlessFor3 {
    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！");
        }
    }
}
```

举例：从键盘读入个数不确定的整数，并判断读入的正数和负数的个数，输入为 0 时结束程序。

```
class PositiveNegative {
      public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            int positiveNumber = 0;//统计正数的个数
            int negativeNumber = 0;//统计负数的个数
            for(;;){  //while(true){
                  System.out.println("请输入一个整数：");
                  int z = scanner.nextInt();
                  if(z>0)
                        positiveNumber++;
                  else if(z<0)
                        negativeNumber++;
                  else
                        break;
         }
```

```
            System.out.println("正数的个数为: "+ positiveNumber);
            System.out.println("负数的个数为: "+ negativeNumber);
        }
}
```

3.6 嵌套循环（或多重循环）

- **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。其中，for ,while ,do...while 均可以作为外层循环或内层循环。

- 实质上，嵌套循环就是把内层循环当成外层循环的循环体。当只有内层循环的循环条件为 false 时，才会完全跳出内层循环，才可结束外层的当次循环，开始下一次的循环。

- 设外层循环次数为 m 次，内层为 n 次，则内层循环体实际上需要执行 m*n 次。

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

执行特点：外层循环执行一次，内层循环执行一轮。

案例 1：将一天中的时间打印到控制台

```
public class ForForDemo {
      public static void main (String[] args) {
            for (int hour = 0;hour < 24 ;hour++ ) {
                  for (int min = 0; min < 60 ; min++) {
                        System.out.println(hour + "时" + min +"分");
                  }
            }
      }
}
```

案例 2：打印 5 行直角三角形

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

```
public class TriAngle {
    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 3：九九乘法表

```
1*1=1
1*2=2   2*2=4
1*3=3   2*3=6   3*3=9
1*4=4   2*4=8   3*4=12  4*4=16
1*5=5   2*5=10  3*5=15  4*5=20  5*5=25
1*6=6   2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7   2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8   2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9   2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```

```
public class Exercise17 {
    public static void main(String[] args) {
        for(int i =1;i<=9;i++){
            for(int j=1;j<=i;j++){
                System.out.print(j+"*"+i+"="+(i*j) + "\t");
            }
            System.out.println();
        }
    }
}
```

案例 4：break 结束当层循环

案例：找出 100 以内所有的素数（质数）

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

```
public class Test09LoopNesting {
    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++){
            //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[2,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;
                     （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=2; j<i; j++){
                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);
            }
        }
    }
}
```

康师傅经验之谈：

开发中关于嵌套循环的使用，一般也就两层，最多不要超过 3 层, 否则，代码的可读性会很差。

4. break 和 continue 的使用

4.1 break 关键字

break 语句用于终止某个语句块的执行

```
{     ……
      break;
       ……
}
```

break 语句出现在多层嵌套的语句块中时，可以通过标签指明要终止的是哪一层语句块

```
      label1: {    ……
      label2:          {    ……
      label3:                   {    ……
                                     break label2;
                                     ……
                                }
                          }
                    }
```

举例：

```
public class BreakTest {
    public static void main(String args[]) {
        for (int i = 0; i < 10; i++) {
            if (i == 3)
                break;
            System.out.println(" i =" + i);
        }
        System.out.println("Game Over!");
    }
}
```

练习：

随机生成 1-100 的一个数，直到生成了 97 这个数，看看你一共用了几次？

提示：使用 (int)(Math.random() * 100) + 1

4.2 continue 关键字

- continue 只能使用在循环结构中
- continue 语句用于跳过其所在循环语句块的**一次**执行，继续下一次循环
- continue 语句出现在多层嵌套的循环语句体中时，可以通过标签指明要跳过的是哪一层循环

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);
            }
        }
    }
}
```

小结：

- break 只能用于 switch 语句和循环语句中。
- continue 只能用于循环语句中。
- 二者功能类似，但 continue 是终止本次循环，break 是终止本层循环。
- break、continue 之后不能有其他的语句，因为程序永远不会执行其后的语句。

- 标号语句必须紧接在循环的头部。标号语句不能用在非循环语句的前面。
- 很多语言都有 goto 语句，goto 语句可以随意将控制转移到程序中的任意一条语句上，然后执行它。但使程序容易出错。Java 中的 break 和 continue 是不同于 goto 的。

5. Scanner：键盘输入功能的实现

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

5.1 各种类型的数据输入

示例代码：

```
import java.util.Scanner;
//如果在.java 源文件上面没有这句import 语句，
//那么在代码中每次使用Scanner 就要用java.util.Scanner 的全名称，比较麻烦

/*
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

如果你在键盘输入过程中，遇到java.util.InputMismatchException 异常，
说明你输入的数据，其类型与接收数据的变量的类型不匹配。
 */
public class Test05Input {
    public static void main(String[] args) {
        //1、准备Scanner 类型的对象
        //Scanner 是一个引用数据类型，它的全名称是java.util.Scanner
        //input 就是一个引用数据类型的变量了，赋给它的值是一个对象（对象的概念我们后面学习，暂时先这么叫）
        //new Scanner(System.in)是一个new 表达式，该表达式的结果是一个对象
        //引用数据类型  变量 = 对象;
        //这个等式的意思可以理解为用一个引用数据类型的变量代表一个对象，所以这个变量的名称又称为对象名
        //我们也把input 变量叫做input 对象
        Scanner input = new Scanner(System.in);//System.in 默认代表键盘输
```

```
入
        //这里变量名是input，下面就用input

        //2、提示输入xx
        System.out.print("请输入一个整数：");

        //3、接收输入内容
        int num = input.nextInt();
        System.out.println("num = " + num);

        //列出其他常用数据类型的输入
        /*
        long bigNum = input.nextLong();
        double d = input.nextDouble();
        boolean b = input.nextBoolean();
        String s = input.next();
        char c = input.next().charAt(0);//先按照字符串接收，然后再取字符串
的第一个字符（下标为0）
        */

        //释放资源
        input.close();
    }
}
```

5.2 next()与 nextLine()

```
import java.util.Scanner;

/*
next()方法:
      遇到空格等空白符或回车，就认为输入结束
nextLine()方法:
      遇到回车换行，才认为输入结束
 */
public class Test08NextAndNextLine {
    public static void main(String[] args) {
        //申请资源
        Scanner input = new Scanner(System.in);

        System.out.print("请输入姓名：");
        //String name = input.next();//张 三  只能接收张，后面的空格和三无
法接收，被下面的输入接收
        String name = input.nextLine();
        System.out.println("name = " + name);

        System.out.print("请输入年龄：");
        int age = input.nextInt();  //23 回车换行  这里只接收23，回车换行
```

```
被下面的输入接收
        input.nextLine();//读取23后面的回车换行，但是这个不需要接收，只有下面一个输入是nextLine()情况下才需要这样，如果下面的输入是next()或者是nextInt(),nextDouble()等就不需要这么干
        System.out.println("age = " + age);

        System.out.print("请输入电话号码：");
        String tel = input.nextLine();
        System.out.println("tel = " + tel);

        //释放资源
        input.close();
    }
}
```

第 04 章_IDEA 的安装与使用

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

> 2020 年，为了庆祝 Java 25 周年，JetBrains 从多个来源收集了数据，以查看该语言当下的发展状态。其中，关于 Java 最受欢迎的 IDE 是 JetBrains 的 IntelliJ IDEA，有 72％ 的用户使用它；其次是 Eclipse，占比为 13％。
>
> 2022 年，某美国软件开发商在对近千名专业的 Java 开发者调研后，发布了《2022 年 Java 开发者生产力报告》。报告提到：JetBrains 的 IntelliJ 是最受欢迎的 Java IDE，占 48%，其次是 Eclipse，占 24%，Visual Studio Code 占 18%。

IDEA 全称 IntelliJ IDEA，是 Java 语言开发的集成环境，目前已经基本代替了 Eclipse 的使用。IntelliJ 在业界被公认为最好的 Java 开发工具之一，因其功能强悍、设置人性化，而深受 Java、大数据、移动端程序员的喜爱。

本着"工欲善其事必先利其器"的精神，本章节从 IDEA 的介绍、安装、设置入手，讲解 IDEA 中项目的创建、模板的使用、断点调试等。

1. IntelliJ IDEA 介绍

1.1 JetBrains 公司介绍

IDEA(https://www.jetbrains.com/idea/)是 JetBrains 公司的产品，公司旗下还有其它产品，比如：

- WebStorm：用于开发 JavaScript、HTML5、CSS3 等前端技术；
- PyCharm：用于开发 python
- PhpStorm：用于开发 PHP
- RubyMine：用于开发 Ruby/Rails
- AppCode：用于开发 Objective - C/Swift
- CLion：用于开发 C/C++
- DataGrip：用于开发数据库和 SQL
- Rider：用于开发.NET
- GoLand：用于开发 Go
- Datalore： 用于构建机器学习模型并在 Python 中创建丰富的可视化。

而且用于开发 Android 的 Android Studio，也是 Google 基于 IDEA 社区版进行迭代的。

1576217376761

1.2 IntelliJ IDEA 介绍

IDEA，全称 IntelliJ IDEA，是 Java 语言的集成开发环境，IDEA 在业界被公认为是最好的 Java 开发工具之一，尤其在智能代码助手、代码自动提示、重构、J2EE 支持、Ant、JUnit、CVS 整合、代码审查、创新的 GUI 设计等方面的功能可以说是超常的。

IntelliJ IDEA 在 2015 年的官网上这样介绍自己：

> Excel at enterprise, mobile and web development with Java, Scala and Groovy,with all the latest modern technologies and frameworks available out of thebox.

它内置的工具和支持的框架有：

1.3 IDEA 的主要优势：(相较于 Eclipse 而言)

① 强大的整合能力。比如：Git、Maven、Spring 等

② 提示功能的快速、便捷

③ 提示功能的范围广

④ 好用的快捷键和代码模板

⑤ 精准搜索

1.4 IDEA 的下载

下载网址： https://www.jetbrains.com/idea/download/#section=windows

IDEA 分为两个版本： 旗舰版(Ultimate)和 社区版(Community)。

两个不同版本的详细对比，可以参照官网：
https://www.jetbrains.com/idea/features/editions*comparison*matrix.html

官网提供的详细使用文档： https://www.jetbrains.com/help/idea/meet-intellij-idea.html

1.5 了解：常见软件的不同版本叫法

常见版本：

- alpha 内部测试版
- beta 外部测试版
- demo 演示版
- Enhance 增强版或者加强版，属于正式版
- Free 自由版

- Full version 完全版，属于正式版
- shareware 共享版
- Release 发行版，有时间限制
- Upgrade 升级版
- Retail 零售版
- Enterprise 企业版
- Professional 专业版
- Ultimate 旗舰版
- Community 社区版

其他版本：

- Plus 属增强版，不过这种大部分是在程序界面及多媒体功能上增强。
- Preview 属预览版
- Corporation & Enterprise 企业版
- Standard 标准版
- Mini 迷你版也叫精简版只有最基本的功能
- Express 特别版
- Deluxe 豪华版
- Regged 已注册版
- CN 简体中文版
- CHT 繁体中文版
- EN 英文版
- Multilanguage 多语言版
- trail 试用版（含有某些限制，如时间、功能，注册后也有可能变为正式版）
- RTM 版。这基本就是最终的版本，英文是 Release To Manufactur，意思是发布到生产商。

2. 安装与卸载

2.1 安装前的准备

- 64 位版本的 windows10,8,7；
- 最少 2G 内存，建议 8G 内存以上；
- 安装需要 2.5G 硬盘，建议固态硬盘；
- 分辨率 1024*768 以上；

System requirements

- 64-bit versions of Microsoft Windows 10, 8, 7 (SP1)
- 2 GB RAM minimum, 8 GB RAM recommended
- 2.5 GB hard disk space, SSD recommended
- 1024x768 minimum screen resolution

从安装上来看，IntelliJ IDEA 对硬件的要求似乎不是很高。可是实际在开发中其实并不是这样的，因为 IntelliJ IDEA 执行时会有大量的缓存、索引文件，所以如果你正在使用 Eclipse / MyEclipse，想通过 IntelliJ IDEA 来解决计算机的卡、慢等问题，这基本上是不可能的，本质上你应该对自己的硬件设备进行升级。

2.2 安装过程

1、双击 idealU-2020.2.2.exe

2、欢迎安装

image-20220304222052536

3、是否删除电脑上低版本的 IDEA

如果电脑上有低版本的 IDEA，可以选择删除或保留。

4、选择安装目录

选择安装目录，目录中要避免中文和空格。

IntelliJ IDEA Setup
Choose Install Location
Choose the folder in which to install IntelliJ IDEA.
Setup will install IntelliJ IDEA in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
D:\develop_tools\IDEA\IntelliJ IDEA 2020.2.2\
Browse...
Space required: 1.5 GB
Space available: 319.5 GB
< Back
Next >
Cancel

5、确认操作系统类型

确认是否与.java、.groovy、.kt 格式文件进行关联。这里建议选择不关联。

6、在【开始】菜单新建一个文件夹（这里需要确认文件夹的名称），来管理 IDEA 的相关内容。

IntelliJ IDEA Setup
Choose Start Menu Folder
Choose a Start Menu folder for the IntelliJ IDEA shortcuts.
Select the Start Menu folder in which you would like to create the program's shortcuts. You can also enter a name to create a new folder.
JetBrains
360安全中心
Accessibility
Accessories
Administrative Tools
Apowersoft
AutoHotkey
Binary Viewer
Camtasia Studio 7
DBeaver Community
eDiary
EV加密
EV录屏
< Back
Install
Cancel

IntelliJ IDEA Setup
Installing
Please wait while IntelliJ IDEA is being installed.
Extract: index.js... 100%
Show details
< Back
Next >
Cancel

7、完成安装

2.3 卸载过程

在【控制面板】找到【卸载程序】

双击 IntelliJ IDEA 2019.2.3 进行卸载：

组织

名称	发布者	安装时间	大小	版本
Intel(R) Update Manager	Intel Corporation	2016/10/17	23.1 MB	3.4.1942
Intel(R) WiDi	Intel Corporation	2015/6/5	80.1 MB	5.1.23.0
Intel® Hardware Accelerated Execution Manager	Intel Corporation	2016/2/23	618 KB	6.0.1
IntelliJ IDEA 2017.3.1	JetBrains s.r.o.	2019/10/24		173.3942.27
IntelliJ IDEA 2018.1.5	JetBrains s.r.o.	2019/10/24		181.5281.24
IntelliJ IDEA 2019.2.3	JetBrains s.r.o.	2019/10/24		192.6817.14
IntelliJ IDEA Community Edition 2019.2.3	JetBrains s.r.o.	2019/10/21		192.6817.14
Java SE Development Kit 7 Update 80 (64-bit)	Oracle	2017/7/11	217 MB	1.7.0.800
Java SE Development Kit 8 Update 141 (64-bit)	Oracle Corporation	2019/11/2	271 MB	8.0.1410.15
Java(TM) SE Development Kit 6 Update 43 (64-bit)	Oracle	2018/9/17	113 MB	1.6.0.430
Java(TM) SE Development Kit 9.0.1 (64-bit)	Oracle Corporation	2019/10/24	494 MB	9.0.1.0
Java(TM) SE Development Kit 10.0.2 (64-bit)	Oracle Corporation	2019/10/24	514 MB	10.0.2.0
Java(TM) SE Development Kit 11.0.2 (64-bit)	Oracle Corporation	2019/10/24	277 MB	11.0.2.0
Java(TM) SE Development Kit 12.0.2 (64-bit)	Oracle Corporation	2019/10/24	299 MB	12.0.2.0
Java(TM) SE Development Kit 13.0.1 (64-bit)	Oracle Corporation	2019/10/23	295 MB	13.0.1.0

IntelliJ IDEA Uninstall
Uninstall IntelliJ IDEA
IntelliJ IDEA will be uninstalled from the following folder:
d:\ProgramFiles\JetBrains\IntelliJ_Ultimate_IDEA_2019.2.3\bin
Please confirm if you would like to delete the caches and settings:
Delete IntelliJ IDEA 2019.2.3 caches and local history
删除缓存和历史记录
Delete IntelliJ IDEA 2019.2.3 settings and installed plugins
删除设置和安装的插件
Share the uninstall feedback
如果要保留就不打钩
Uninstall
Cancel

IntelliJ IDEA Uninstall
Uninstalling
Please wait while IntelliJ IDEA is being uninstalled.
Delete file: d:\ProgramFiles\JetBrains\IntelliJ_Ultimate_IDEA_2019.2.3\bin\..\plugins\android\
Show details
< Back
Close
Cancel

3. 初始化配置与注册

1、双击

2、是否导入之前版本的配置

3、启动界面

4、选择 UI 界面的主题

这里根据个人喜好，进行选择，也可以选择跳过(skip all and set defaults)。后面在 settings 里也可以再设置主题等。

5、注册：略

6、创建项目

1576221581332

7、创建 Java 项目

1576221856659

1576221875313

8、指定项目名称和代码存储目录

9、关闭提示

10、完成项目创建

补充：登录 IDEA 闪退问题

问题描述：2021.2 启动不了，双击桌面图标，没有响应；CMD 里输入 idea 有报错。

进入到安装目录...\IntelliJ IDEA 2021.2\bin，打开 CMD。输入 idea，发现有报错。

```
OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was depreca
ted in version 9.0 and will likely be removed in a future release.
```

```
 ====================================================
 =======        Jetbrains License Crack       =======
 =======           https://zhile.io           =======
 ====================================================

 @See: https://zhile.io/2018/08/17/jetbrains-license-server-crack.html

 @Version: 3.2.0, @Build Date: 2020-04-10


Exception in thread "main" java.lang.reflect.InvocationTargetException
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invo
ke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invo
ke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.
invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:566)
        at java.instrument/sun.instrument.InstrumentationImpl.loadClass
AndStartAgent(InstrumentationImpl.java:513)
        at java.instrument/sun.instrument.InstrumentationImpl.loadClass
AndCallPremain(InstrumentationImpl.java:525)
Caused by: java.lang.RuntimeException: java.lang.NullPointerException
        at fuck_the_regulations_v320.en.b(en.java)
        at fuck_the_regulations_v320.dH.premain(dH.java)
        ... 6 more
Caused by: java.lang.NullPointerException
        ... 8 more
*** java.lang.instrument ASSERTION FAILED ***: "result" with message ag
ent load/premain call failed at ./src/java.instrument/share/native/libi
nstrument/JPLISAgent.c line: 422
FATAL ERROR in native method: processing of -javaagent failed, processJ
avaStart failed
```

解决办法：

打开 C:\Users\songhk\AppData\Roaming\JetBrains\IntelliJIdea2021.2\idea64.exe.vmoptions 这个文件。内容如下所示：

```
-Xms128m
-Xmx2048m
-XX:ReservedCodeCacheSize=240m
-XX:+UseConcMarkSweepGC
-XX:SoftRefLRUPolicyMSPerMB=50
-ea
-XX:CICompilerCount=2
```

```
-Dsun.io.useCanonPrefixCache=false
-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
-javaagent:D:\develop_tools\IDEA\IntelliJ IDEA 2019.2.2\bin\jetbrains-a
gent.jar
-Deditable.java.test.console=true
```

需要将倒数第 2 行-javaagent:D:\develop_tools\IDEA\IntelliJ IDEA 2019.2.2\bin\jetbrains-agent.jar 删除。再次启动 idea，成功。

原因：之前使用过 2019.2.2 版本，pojie 了。新版 IEDA 太智能了，把现有的启运参数也都复制过去了。又因为最新的 IEDA，不兼容 pojie 程序-javaagent:D:\develop_tools\IDEA\IntelliJ IDEA 2019.2.2\bin\jetbrains-agent.jar 了，所以报错了，所以 JVM 结束了，所以没有启动画面，凉凉了。

4. HelloWorld

4.1 IDEA 项目结构

层级关系：

`project - module - package - class`

具体的：

`project` 中可以创建多个 `module`

`module` 中可以创建多个 `package`

`package` 中可以创建多个 `class`

这些结构的划分，是为了方便管理功能代码。

4.2 Project 和 Module 的概念

在 IntelliJ IDEA 中，提出了 Project 和 Module 这两个概念。

image-20220523014358169

在 IntelliJ IDEA 中 Project 是最顶级的结构单元，然后就是 Module，一个 Project 可以有多个 Module。目前，主流的大型项目结构基本都是多 Module 的结构，这类项目一般是按功能划分的，比如：user-core-module、user-facade-module 和 user-hessian-module 等等，模块之间彼此可以相互依赖。通过这些 Module 的命名可以看出，它们都是处于同一个项目中的模块，彼此之间是有着不可分割的业务关系。因此，我们可以大致总结出：**一个 Project 是由一个或多个 Module 组成：**

- 当为单 Module 项目的时候，这个单独的 Module 实际上就是一个 Project；
- 当为多 Module 项目的时候，多个模块处于同一个 Project 之中，此时彼此之间具有互相依赖的关联关系。
- 当然多个模块没有建立依赖关系的话，也可以作为单独一个“小项目”运行

我们可以简单地理解 Project 就是一个单纯的目录，只是这个目录在命名上必须有其代表性的意义。在缺省情况下，IntelliJ IDEA 是默认单 Project 单 Module 的，这时 Project 和 Module 合二为一，在没有修改存储路径的时候，显然 Project 对 Module 具有强约束作用！

4.3 新建 Java 类

4.4 编写代码

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

4.5 运行

5. 详细设置

5.1 如何打开详细配置界面

1、显示工具栏

2、选择详细配置菜单或按钮

5.2 系统设置

1、默认启动项目配置

启动 IDEA 时，默认自动打开上次开发的项目？还是自己选择？

如果去掉 Reopen last project on startup 前面的对勾，每次启动 IDEA 就会出现如下界面：

1576460329524

2、取消自动更新

Settings-->Appearance & Behavior->System Settings -> Updates

5.3 设置整体主题

1、选择主题

2、设置菜单和窗口字体和大小

5.4 设置编辑器主题样式

1、编辑器主题

2、字体大小与颜色

更详细的字体与颜色如下：

温馨提示：如果选择某个 font 字体，中文乱码，可以在 fallback font（备选字体）中选择一个支持中文的字体。

3、显示行号与方法分隔符

5.5 代码智能提示功能

IntelliJ IDEA 的代码提示和补充功能有一个特性：区分大小写。区分大小写的情况是这样的：比如我们在 Java 代码文件中输入 system，IntelliJ IDEA 默认是不会帮我们提示或是代码补充的，但是如果我们输入 System 就可以进行代码提示和补充。如果想不区分大小写的话，就把这个对勾去掉。建议去掉勾选。

5.6 自动导包配置

- 默认需要自己手动导包，Alt+Enter 快捷键

- 自动导包设置

- 动态导入明确的包：Add unambiguous imports on the fly，该设置具有全局性；
- 优化动态导入的包：Optimize imports on the fly，该设置只对当前项目有效；

5.7 设置项目文件编码（一定要改）

Settings-->Editor-->File Encodings

说明： Transparent native-to-ascii conversion 主要用于转换 ascii，一般都要勾选。

6. 编译问题（了解）

6.1 设置自动编译

Settings-->Build,Execution,Deployment-->Compiler

6.2 项目的 JDK 设置

方式一：File-->Project Structure...-->Platform Settings -->SDKs

方式二：工具栏-->Project Structure -->Platform Settings -->SDKs

注：SDKs 全称是 Software Development Kit ，这里一定是选择 JDK 的安装根目录，而不是 JRE 的目录。

6.3 out 目录和编译版本

方式一：File-->Project Structure...-->Project Settings -->Project

方式二：工具栏-->Project Structure -->Project Settings -->Project

Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Project name:
JavaSECode
Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
1.8 (java version "1.8.0_271")
New...
Edit
选择JDK8
Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as required.
8 - Lambdas, type annotations etc.
选择8版本语法规范编译
Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production code and test sources, respectively.
A module specific compiler output path can be configured for each of the modules as required.
D:\atguigu\javaee\JavaSE20220106\JavaSECode\out
设置out目录
out目录是存放项目中各个模块编译后的.class文件的
如果out目录没有正确设置，将无法编译
OK
Cancel
Apply

7. 快捷键和代码模板

7.1 取消双击 shift 搜索

因为我们按 shift 切换中英文输入方式，经常被按到，总是弹出搜索框，太麻烦了。可以取消它。

双击 shift 或 ctrl + shift + a，打开如下搜索窗口：

选择 registry...，找到"ide.suppress.double.click.handler"，把复选框打上勾就可以取消双击 shift 出现搜索框了。

7.2 常用快捷代码模板（第一部分）

见《IntelliJ IDEA 常用快捷键一览表.md》

7.3 查看和自定义快捷键

1、已知快捷键操作名，未知快捷键

2、已知快捷键，不知道对应的操作名

尚硅

3、自定义自己的快捷键

4、其他主题快捷键

苹果电脑或者是用惯 eclipse 快捷的，可以选择其他快捷键插件。

Settings
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Languages & Frameworks
Tools
Keymap
选择其他快捷键插件
Default for Windows
Default for Windows
Default for GNOME
Default for KDE
Default for macOS
Default for XWin
Eclipse
Eclipse (macOS)
Emacs
Version Control Systems
Debugger Actions
Ant Targets
Remote External Tools
Macros
Quick Lists
Plug-ins
Other
OK
Cancel
Apply

7.4 查看和定义代码模板

1、预定义 Postfix Completion 模板，不能修改

2、预定义 Live Templates 模板，可以修改

3、自定义代码模板

例如：定义 sop 代表 System.out.print();语句

①在 Live Templates 中增加模板

②先定义一个模板的组，这样方便管理所有自定义的代码模板

③在模板组里新建模板

④定义模板

- Abbreviation：模板的缩略名称
- Description：模板的描述
- Template text：模板的代码片段
- 模板应用范围。比如点击 Define。选择如下：应用在 java 代码中。

- [ ] AspectJ
- [ ] CoffeeScript
- [ ] ColdFusion
- [ ] CSS
- [ ] Cucumber feature
- [ ] ECMAScript 6 or higher
- [ ] Groovy
- [ ] GSP
- [ ] Haml
- [ ] HTML
- [ ] HTTP Request
- [x] Java
- [ ] JavaScript and TypeScript
- [ ] JSON
- [ ] JSP
- [ ] Kotlin
- [ ] Maven
- [ ] MXML
- [ ] OGNL
- [ ] Shell Script
- [ ] SQL
- [ ] TypeScript
- [ ] XML
- [ ] Other

8. 多模块管理

8.1 创建多模块

JavaSE 阶段建议创建“Empty 空工程”，然后创建多模块，每一个模块可以独立运行，相当于一个小项目。JavaSE 阶段不涉及到模块之间的依赖。后期再学习模块之间的依赖。

步骤：

（1）选择创建模块

（2）选择模块类型：这里选择创建 Java 模块

（3）给模块命名

New Module
Module name:
day04_teacher_code
Content root:
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day04_teacher_code
设置新模块存储路径
Module file location:
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day04_teacher_code
完成
Previous
Finish
Cancel
Help

（4）多个模块并列存在

8.2 删除模块

（1）移除模块

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Windo
Add Configuration...
day04_teacher_code
Project
1: Project
day03_teacher_code D:\at
src
day03_teacher_code.iml
day04_teacher_code D:\at
src
选择要删除的模块
day04_teacher_code.iml
External Libraries
Scratches and Consoles
New
Add Framework Support...
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy Relative Path Ctrl+Alt+Shift+C
Paste Ctrl+V
Find Usages Alt+F7
Find in Path... Ctrl+Shift+F
Replace in Path... Ctrl+Shift+R
Analyze
Refactor
Add to Favorites
Show Image Thumbnails Ctrl+Shift+T
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
移除模块
Remove Module Delete
Build Module 'day04_teacher_code'

Remove Module
Remove Module 'day04_teacher_code' from the project?
No files will be deleted.
Remove
Cancel

（2）彻底删除模块

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window H
Add Configuration...
day03_teacher_code
Project
day03_teacher_code D:\atguig
src
选择模块名
day03_teacher_code.iml
External Libraries
Scratches and Consoles
New
Add Framework Support...
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy Relative Path Ctrl+Alt+Shift+C
Paste Ctrl+V
Find Usages Alt+F7
Find in Path... Ctrl+Shift+F
Replace in Path... Ctrl+Shift+R
Analyze
Refactor
Add to Favorites
Show Image Thumbnails Ctrl+Shift+T
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Remove Module Delete
Build Module 'day03_teacher_code'
Rebuild ...'day03_teacher_code' Ctrl+Shift+F9
打开资源管理器
Show in Explorer

8.3 导入老师的模块

（1）将老师的模块整个的复制到自己 idea 项目的路径下

（2）选择新建 Java 模块

File
Edit
View
Navigate
Code
Analyze
Refactor
Build
Run
Tools
VCS
New
Open...
Open URL...
Open Recent
Close Project
Project...
Project from Existing Sources...
Project from Version Control...
Module...
Module from Existing Sources...

New Module
Java
Java Enterprise
JBoss
Clouds
Spring
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Kotlin
Static Web
Node.js and NPM
Flash
Module SDK:
Project SDK (1.8)
New...
Additional Libraries and Frameworks:
JBoss
Arquillian JUnit
Arquillian TestNG
JBoss Drools
Java EE
Web Application
Struts 2
WebServices
CDI: Contexts and Dependency Injection
Thymeleaf
IE Conditional Comments support
Previous
Next
Cancel
Help

（3）选择老师的模块

New Module
Module name:
day03_teacher_code_第2节课
Content root:
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day03_teacher_code_第2节课
Module file location:
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day03_teacher_code_第2节课
选择复制到项目中老师的模块
Previous
Finish
Cancel
Help

Select Module Content Root
Selected directory will be marked as module content root
Hide path
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day03_teacher_code_第2节课
JavaSE20211018
JavaSE20220106
JavaSECode
.idea
day03_self_code
day03_teacher_code_第1节课
day03_teacher_code_第2节课
out
JavaSEHomework
jiaoxue
mysqlbooks
note
Drag and drop a file into the space above to quickly locate it in the tree
OK
Cancel

New Module
Module name:
day03_teacher_code_第2节课
Content root:
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day03_teacher_code_第2节课
Module file location:
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day03_teacher_code_第2节课
File Already Exists
The module file
'D:\atguigu\javaee\JavaSE20220106\JavaSECode\day03_teacher_code_第2节课\day03_teacher_code_第2节课.iml'
already exists.
Would you like to overwrite it?
Yes
No
Previous
Finish
Cancel
Help

File
Edit
View
Navigate
Code
Analyze
Refacto
Add Configuration...
day03_teacher_code_第2节课
1: Project
Project
day03_self_code D:\atguigu\javaee\JavaSE202201
day03_teacher_code_第1节课 D:\atguigu\javaee\J
day03_teacher_code_第2节课 D:\atguigu\javaee\J
External Libraries
Scratches and Consoles

8.4 同时打开两个 IDEA 项目工程

1、两个 IDEA 项目工程效果

有些同学想要把上课练习代码和作业代码分开两个 IDEA 项目工程。

2、新建两个 IDEA 项目

注意：第一次需要新建，之后直接打开项目工程即可

New Project
Java
Java Enterprise
JBoss
Clouds
Spring
Java FX
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Kotlin
Static Web
Node.js and NPM
Flash
Empty Project
Empty Project
Empty project without modules. Use it to create free-style module structure.
空工程
Previous
Next
Cancel
Help

3、打开两个 IDEA 项目

8.5 导入前几天非 idea 工程代码

1.先将前 2 天的代码文件放到 idea 工程代码目录下，并用第 8.3 节的方式创建两个模块。

2.把.java 文件移动到 src 中

3.设置编码

当前项目是 UTF-8。如果原来的.java 文件都是 GBK 的（如果原来.java 文件有的 GBK 有的 UTF-8 就比较麻烦了）。

可以单独把这两个模块设置为 GBK 编码的。

Settings
Keymap
Editor
General
Font
Color Scheme
Code Style
Inspections
File and Code Templates
File Encodings
Live Templates
File Types
Android Layout Editor
Copyright
Inlay Hints
Android Data Binding
Duplicates
Emmet
GUI Designer
Images
Intentions
Language Injections
Spelling
TextMate Bundles
TODO
Editor › File Encodings
For current project
Reset
Global Encoding: UTF-8
Project Encoding: UTF-8
Path
Encoding
Encodings are not configured
To change encoding IntelliJ IDEA uses for a file, a directory, or the entire project, add its path if necessary and then select encoding from the encoding list. Built-in file encoding (e.g. JSP, HTML or XML) overrides encoding you specify here. If not specified, files and directories inherit encoding settings from the parent directory or from the Project Encoding.
Properties Files (*.properties)
Default encoding for properties files: UTF-8
Transparent native-to-ascii conversion
BOM for new UTF-8 files
Create UTF-8 files: with NO BOM
IDEA will NOT add UTF-8 BOM to every created file in UTF-8 encoding
OK
Cancel
Apply

Select Path
Hide path
D:\atguigu\javaee\JavaSE20220106\JavaSECode\day02_code
JavaSE20210906
JavaSE20211018
JavaSE20220106
JavaSECode
.idea
day01_code
day02_code
day03_self_code
day03_teacher_code_第1节课
day03_teacher_code_第2节课
out
JavaSEHomework
jiaoxue
Drag and drop a file into the space above to quickly locate it in the tree
OK
Cancel

9. Debug(调试)程序

9.1 为什么需要 Debug

编好的程序在执行过程中如果出现错误，该如何查找或定位错误呢？简单的代码直接就可以看出来，但如果代码比较复杂，就需要借助程序调试来查找错误了。

在断点调试的过程中，是运行状态，是以对象的运行类型来执行的。比如：

```
class SuperClass{
      public void method1(){
            System.out.println("执行父类的方法");
      }
}

class SubClass{
      public void method1(){
            System.out.println("执行子类重写的方法");
      }
}

class Test{
    public static void main(String[] args){
        SuperClass s = new SubClass();
        s.method1(); //断定此方法执行的是谁的method1()?
    }
}
```

9.2 Debug 的步骤

Debug(调试)程序步骤如下：

1、添加断点

2、启动调试

3、单步执行

4、观察变量和执行流程，找到并解决问题

9.3 启动 Debug

1）添加断点：在源代码文件中，在想要设置断点的代码行的前面的标记行处，单击鼠标左键就可以设置断点，在相同位置再次单击即可取消断点。

TestMap.java

```
1   import java.util.HashMap;
2
3   public class TestMap {
4       public static void main(String[] args) {
5           HashMap<Integer, String> map = new HashMap<>();
6           map.put(1, "hello");
7           map.put(2, "java");
8           map.put(3, "world");
9   断点    System.out.println(map);
10      }
11  }
```

经验：初学者对于在哪里加断点，缺乏经验，这也是调试程序最麻烦的地方，需要一定的经验。

简单来说，在可能发生错误的代码的前面加断点。如果不会判断，就在程序执行的起点处加断点。

2）启动调试：IDEA 提供多种方式来启动程序(Launch)的调试，分别是通过菜单(Run –> Debug)、图标(“绿色臭虫” 等等

ideatest1 [D:\atguigu\javaee\JavaSE20191213\code\ideatest1] - ...\src\TestMap.java - IntelliJ IDEA (Administrator)
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
TestMap
ideatest1 src TestMap
Project
TestMap.java
HashMap.java
ideatest1 D:\atguigu\javaee\JavaSE20191
.idea
out
src
HelloWorld
MyImpl
MyInter
Student
TestMap
ideatest1.iml
External Libraries
5 HashMap<Integer, String> map = new HashMap<>(); map: "{1=hello}"
6 map.put(1, "hello");
7 map.put(2, "java"); map: "{1=hello}"
8 map.put(3, "world");
9 System.out.println(map);
10 }
11 }
TestMap › main()
Debug: TestMap
Debugger Console
Frames
Variables
"main"@1 in group "mai...
putVal:630, HashMap (java.util)
put:611, HashMap (java.util)
main:7, TestMap
args: String[] = {String[0]@527}
map: HashMap = {HashMap@528} "{1=hello}"
: void = undefined
table: HashMap$Node[] = {HashMap$Node[16]@532}
entrySet: Set = {HashMap$EntrySet@530} size = 1
size: int = 1
modCount: int = 1
threshold: int = 12
loadFactor: float = 0.75
keySet: Set = null
values: Collection = null
7: Structure
2: Favorites

9.4 单步调试工具介绍

尚硅谷

Run Tools VCS Window Help

Run 'TestMap'	Shift+F10
Debug 'TestMap'	Shift+F9
Run 'TestMap' with Coverage	
Run with Profiler	›
Run...	Alt+Shift+F10
Debug...	Alt+Shift+F9
Profile...	
Attach to Process...	Ctrl+Alt+F5
Attach Profiler to Process...	
Edit Configurations...	
Import Profiler Results	›
Test History	›
Import Tests from File...	
Stop 'TestMap'	Ctrl+F2
Stop Background Processes...	Ctrl+Shift+F2
Show Running List	
Update Running Application...	Ctrl+F10
Reload Changed Classes	
Restart Activity	
Step Over	F8
Force Step Over	Alt+Shift+F8
Step Into	F7
Force Step Into	Alt+Shift+F7
Smart Step Into	Shift+F7
Step Out	Shift+F8
Step Out Of Code Block	
Run to Cursor	Alt+F9
Force Run to Cursor	Ctrl+Alt+F9
Force Return	
Throw Exception	
Pause Program	
Resume Program	F9

或

：Step Over（F8）：进入下一步，如果当前行断点是调用一个方法，则不进入当前方法体内

：Step Into（F7）：进入下一步，如果当前行断点是调用一个自定义方法，则进入该方法体内

：Force Step Into（Alt +Shift + F7）：进入下一步，如果当前行断点是调用一个核心类库方法，则进入该方法体内

：Step Out（Shift + F8）：跳出当前方法体

：Run to Cursor（Alt + F9）：直接跳到下一个断点处继续调试

：Resume Program（F9）：恢复程序运行，但如果该断点下面代码还有断点则停在下一个断点上

：Stop（Ctrl + F2）：结束调试

：View Breakpoints（Ctrl + Shift + F8）：查看所有断点

：Mute Breakpoints：使得当前代码后面所有的断点失效， 一下执行到底

说明：可以在 debug 过程中，动态的下断点。

注意：使用 Step Into 时，会出现无法进入源码的情况。如何解决？

方案 1：使用 force step into 即可

方案 2：点击 Setting -> Build,Execution,Deployment -> Debugger -> Stepping

把 Do not step into the classess 中的 `java.*`、`javax.*` 取消勾选即可。

image-20220516200427750

9.5 自定义调试数据视图

9.6 案例剖析

举例 1：

```
public class Debug01 {
    public static void main(String[] args) {
        //测试代码
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum += i;
            System.out.println("i = " + i);
            System.out.println("sum = " + i);
        }
        System.out.println("退出循环....");
    }
}
```

举例 2：

```java
public class Debug02 {
    public static void main(String[] args) {
            int i = 10;
            int j = 20;
            System.out.println("i = " + i + ", j = " + j);

            Debug02 test = new Debug02();
            int max = test.getMax(i, j);

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

      private int getMax(int k, int m) {
            int max = 0;
            if (k < m) {
                  max = k;
            } else {
                  max = m;
            }
            return max;
      }
}
```

举例 3：演示异常

```java
public class Debug03 {
    public static void main(String[] args) {
        int[] arr = new int[]{1, 2, 3};
        for (int i = 0; i <= arr.length; i++) {
            System.out.println(arr[i]);
        }
        System.out.println("退出循环结构...");
    }
}
```

举例 4：

```java
public class Debug04 {

      public static void main(String[] args) {
            int[] arr = new int[] {1,2,3,4,5};
            System.out.println(arr);//地址值

            char[] arr1 = new char[] {'a','b','c'};
            System.out.println(arr1);//abc
      }

}
```

举例 5：演示源码

```
public class Debug05 {
    public static void main(String[] args) {
        HashMap<Integer,String> map = new HashMap<>();
        map.put(1,"高铁");
        map.put(2,"网购");
        map.put(3,"支付宝");
        map.put(4,"共享单车");

        System.out.println(map);
    }
}
```

举例 6：熟悉对象实例化过程

```
public class Debug06 {
    public static void main(String[] args) {
        String str = new String("atguigu");
        System.out.println(str);
    }
}

public class Debug07 {
    public static void main(String[] args) {
        //使用force step into 测试下行代码与 new String()的不同。（会执行
ClassLoader 的 loadClass()）
        Person per = new Person("Tom",12);
        System.out.println(per);
    }
}

class Person{
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

10. 其他常用代码模板和快捷键

10.1 常用代码模板

1.非空判断

- ifn：if(xx == null)
- inn：if(xx != null)
- 变量.null：if(变量 == null)
- 变量.nn：if(变量 != null)

2.遍历数组和集合

- fori：for 循环
- itar：遍历数组
- iter：foreach 循环
- List 集合遍历
  - list 集合.for：foreach 循环
  - list 集合.fori：用 for 循环遍历 list 集合，正向遍历
  - list 集合.forr：用 for 循环遍历 list 集合，逆向遍历

3.静态常量声明

- psf：public static final
- prsf：private static final
- psfi：public static final int
- psfs：public static final String

10.2 常用快捷键

1.提高代码编写速度

- 智能提示：Alt + 回车
- 自动代码
  - 自动补全函数括号、分号、当前行缩进：Ctrl + Shift + 回车
  - 提示代码模板：Ctrl + J

尚硅谷

  - 使用 xx 块环绕：Ctrl + Alt + T，例如 try...catch，for，while 等
  - 自动生成构造器、get/set 等：Alt + Insert （有的键盘需要同时按 Fn 键）
  - 由方法自动生成返回值变量：Ctrl + Alt + V
- 抽取代码重构方法：Ctrl + Alt + M
- 查看所调用方法的形参列表提醒：Ctrl + P
- 重写或实现方法
  - 实现接口的方法：Ctrl + i
  - 重写父类的方法：Ctrl + O

2.移动光标定位

- 移动定位
  - 跳转到上一个/下一个位置编辑位置：Ctrl + Alt + 左/右键
  - 跳转到上一个/下一个方法位置： Alt + 上/下键
  - 打开的类文件之间切换： Alt + 左/右键
  - 定位某行：Ctrl + G
- 选择文本：Ctrl + W
  - 按一下选择单词，再按选择语句，再按选择行，再按选择方法

3.查找与查看源码

- 按住 Ctrl 键，单击某个类或方法就可以打开某个类或方法的源码
  - 但是当某个变量是通过多态引用调用父类或父接口的方法时，想要打开方法的实现类：Ctrl +Alt +B
- 查看方法文档：选中方法名后按 Ctrl + Q
- 查看某个类的结构信息
  - 类的继承层次：Ctrl + H
  - 类的 UML 关系图：Ctrl + Alt + U
  - 当前文件结构：Ctrl + F12 键
- 搜索

  - 搜索类：Ctrl + N
  - 全项目搜索文本：Ctrl + Shift + F
  - 全项目替换文本：Ctrl +Shift + R
  - 当前源文件查找文本：Ctrl + F
  - 当前源文件替换问题：Ctrl + R
  - 任何地方搜索：双击 Shift（建议取消）
- 折叠与展开
  - 折叠方法实现：Ctrl + Shift + -
  - 展开方法实现：Ctrl +Shift + +

尚硅谷

11. 文档注释

11.1 IDEA 使用 javadoc 生成 API

尚硅谷

11.2 自定义文档注释模板

类文档注释：

```
/**
$className$
 * date $date$
 * @since 1.8
 * @author $author$
*/
```

无参方法：

```
/**
    $methodname$
 * date $date$
 * @since 1.8
 * @author $user$
 * @return $returnType$
 */
```

![image-20201016130630633](images/image-20201016130630633.png

有参方法：

```
/**
    $methodname$
 * @date $date$
 * @since 1.8
 * @author $user$
$params$
 * @return $returnType$
 */
```

```
groovyScript("def result=''; def params=\"${_1}\".replaceAll('[\\\\[|\\
\\]|\\\\s]', '').split(',').toList(); for(i = 0; i < params.size(); i++)
 {result+=' * @param ' + params[i] + ((i < params.size() - 1) ? '\\n' :
 '')}; return result", methodParameters())
```

注意：

设置模板之后要选择支持的语言。

使用这个两个方法的文档注释模板时，如果方法的返回值类型是 void，生成之后把@return void 删掉，否则生成文档注释报错

使用这两个文档注释模板时必须把光标定位在方法体内部才能有效，否则读取不到 methodName 等信息。

```
public class TestDocTemplate {
    public int method(int a, int b){
        mhp
        mhp                                  方法文档注释有参数
    }   MethodHandleProxies (java.lang.invoke)
        Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
}
```

```
/**
TestDocTemplate 类演示文档注释
 * date 2020/10/16
 * @since 1.8
 * @author Irene
*/
public class TestDocTemplate {
    /**
     method 是演示方法
     * date 2020/10/16
     * @since 1.8
     * @author Irene
     */
    public void method(){

    }

    /**
     method 的作用是求两个整数的和
     * date 2020/10/16
     * @since 1.8
     * @author Irene
     * @param a int
     * @param b int
     * @return int
     */
    public int method(int a ,int b){
        return a+b;
    }
}
```

12. 使用第三方框架和组件（以 JUnit 为例）

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。

12.1 引入本地 JUnit.jar

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：

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

Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
day03_teacher_code
day04_teacher_code
Name: day04_teacher_code
Sources
Paths
Dependencies
Module SDK: 1.8 (java version "1.8.0_271")
New...
Edit
Export
Scope
1.8 (java version "1.8.0_271")
<Module source>
1 JARs or directories...
2 Library...
3 Module Dependency...
Dependencies storage format: IntelliJ IDEA (.iml)
OK
Cancel
Apply

Choose Libraries
Project Libraries
junitlibs
Application Server Libraries
Add Selected
New Library...
Cancel

Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
day03_teacher_code
day04_teacher_code
Name: day04_teacher_code
Sources
Paths
Dependencies
Module SDK: 1.8 (java version "1.8.0_271")
New...
Edit
Export
Scope
1.8 (java version "1.8.0_271")
<Module source>
junitlibs
Test
Compile
Test
Runtime
Provided
Dependencies storage format: IntelliJ IDEA (.iml)
OK
Cancel
Apply

12.2 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
```

```
idea64.exe.vmoptions
-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
```

12.3 定义 test 测试方法模板

选中自定义的模板组，点击”+”（1.Live Template）来定义模板。

13. 导出 jar

（1）单击工具栏的 打开项目设置。

（2）选择 Artifacts（打包部署），选择添加 jar。

（3）给导出的 jar 命名，确认导出的路径

（4）确认创建或选择已有的 Manifest.MF 文件。单词“manifest”的意思是“显示”。这个 manifest 文件定义了与扩展和包相关的数据。

说明： 打开 Java 的 JAR 文件我们经常可以看到文件中包含着一个 META-INF 目录，这个目录下会有一些文件，其中必有一个 MANIFEST.MF，这个文件描述了该 Jar 文件的很多信息

（5） 选择 Manifest.MF 文件存储目录。

（6）选择 jar 要包含的.class 文件的目录，即哪些模块的 out 目录

Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
helloworld
Name: helloworld
Type: JAR
Output directory: atguigu\javaee\JavaSE20191213\code\1213Javacode\out\artifacts\helloworld
Include in project build
Output Layout
Pre-processing
Post-processing
Available Elements
选择要打包的模块的out目录
Library Files
Module Output
Module Sources
File
Directory Content
Extracted Directory
1213Javacode
day25_teacher_code
'helloworld.jar' manifest properties:
Manifest File:
Main Class:
Class Path:
Show content of elements
OK
Cancel
Apply

Choose Module
day03_teacher_code
day04_teacher_code
helloworld
选择对应模块的out目录
OK
Cancel

Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
helloworld
Name: helloworld
Type: JAR
Output directory: D:\atguigu\javaee\JavaSE20220106\JavaSECode\out\artifacts\helloworld
Include in project build
Output Layout
Pre-processing
Post-processing
Available Elements
helloworld.jar
META-INF
'helloworld' compile output
day03_teacher_code
day04_teacher_code
helloworld
'helloworld.jar' manifest properties:
Manifest File: duction\helloworld\META-INF\MANIFEST.MF
Main Class:
Class Path:
Show content of elements
OK

（7）编译生成 jar

13\code\1213Javacode] - IntelliJ IDEA (Administrato
ctor
Build
Run
Tools
VCS
Window
Help
Build Project Ctrl+F9
Build Module
Recompile Ctrl+Shift+F9
Rebuild Project
Generate Ant Build...
Build Artifacts...
Analyze APK...
Run Ant Target Ctrl+Shift+F10

14. 缓存和索引的清理

IntelliJ IDEA 首次加载项目的时候，都会创建索引，而创建索引的时间跟项目的文件多少成正比。

在 IntelliJ IDEA 创建索引过程中即使你编辑了代码也是编译不了、运行不起来的，所以还是安安静静等 IntelliJ IDEA 创建索引完成。

IntelliJ IDEA 的缓存和索引主要是用来加快文件查询，从而加快各种查找、代码提示等操作的速度，所以 IntelliJ IDEA 的索引的重要性再强调一次也不为过。

但是，IntelliJ IDEA 的索引和缓存并不是一直会良好地支持 IntelliJ IDEA 的，某些特殊条件下，IntelliJ IDEA 的缓存和索引文件也是会损坏的，比如：断电、蓝屏引起的强制关机，当你重新打开 IntelliJ IDEA，很可能 IntelliJ IDEA 会报各种莫名其妙错误，甚至项目打不开，IntelliJ IDEA 主题还原成默认状态。即使没有断电、蓝屏，也会有莫名奇怪的问题的时候，也很有可能是 IntelliJ IDEA 缓存和索引出现了问题，这种情况还不少。遇到此类问题也不用过多担心。我们可以清理缓存和索引。如下：

ideatest1 [D:\atguigu\javaee\JavaSE201912
File
Edit
View
Navigate
Code
Analyze
New
Open...
Open URL...
Open Recent
Close Project
Settings...
Ctrl+Alt+S
Project Structure...
Ctrl+Alt+Shift+S
Other Settings
Import Settings...
Export Settings...
Settings Repository...
Sync Settings to JetBrains Account...
Export Project to Eclipse...
Export to Zip File...
Save All
Ctrl+S
Synchronize
Ctrl+Alt+Y
Invalidate Caches / Restart...
Print...
Associate with File Type...
Line Separators
Power Save Mode
Exit

- 一般建议点击 Invalidate and Restart，这样会比较干净。
- 上图警告：清除索引和缓存会使得 IntelliJ IDEA 的 Local History 丢失。**所以如果你项目没有加入到版本控制，那你最好备份下你的 LocalHistory 目录。** LocalHistory 目录在用户目录下，例如：C:\Users\Irene.IntelliJIdea2019.2\system。
- 通过上面方式清除缓存、索引本质也就是去删除 C 盘下的 system 目录下的对应的文件而已，所以如果你不用上述方法也可以手动删除整个 system。当 IntelliJ IDEA 再次启动项目的时候会重新创建新的 system 目录以及对应项目缓存和索引。

15. 自定义代码快捷模板

15.1 冒泡排序的快捷模板 bsort

①在 Settings->Editor->Live Templates 中增加模板，如果没有 CustomDefine 的请看本比较《18.7.6 的 3.自定义代码模板》

②定义 bsort 对应冒泡排序代码

```
for(int $INDEX$ = 1; $INDEX$ < $ARRAY$.length; $INDEX$++) {
    for(int $INDEX2$ = 0; $INDEX2$ < $ARRAY$.length-$INDEX$; $INDEX2$++)
 {
        if($ARRAY$[$INDEX2$] > $ARRAY$[$INDEX2$+1]){
            $ELEMENT_TYPE$ temp = $ARRAY$[$INDEX2$];
            $ARRAY$[$INDEX2$] = $ARRAY$[$INDEX2$+1];
            $ARRAY$[$INDEX2$+1] = temp;
        }
    }
}
```

Settings
Editor › Live Templates
Reset
By default expand with Tab
AndroidParcelable
AndroidXML
Angular
AngularJS
ColdFusion
CustomDefine
arrprint (输出数组元素)
Abbreviation: bsort
快捷模板缩略词
Description: 冒泡排序
功能描述
Template text:
for(int $INDEX$ = 1; $INDEX$ < $ARRAY$.length; $INDEX$++) {
for(int $INDEX2$ = 0; $INDEX2$ < $ARRAY$.length-$INDEX$; $INDEX2$++) {
if($ARRAY$[$INDEX2$] > $ARRAY$[$INDEX2$+1]){
$ELEMENT_TYPE$ temp = $ARRAY$[$INDEX2$];
$ARRAY$[$INDEX2$] = $ARRAY$[$INDEX2$+1];
$ARRAY$[$INDEX2$+1] = temp;
}
}
}
对应代码
Edit variables
Options
Expand with Default (Tab)
Reformat according to style
Use static import if possible
Shorten FQ names
Applicable in Java; Java: statement, expression, declaration, comment, string, smart type completion... Change
OK
Cancel
Apply

③定义代码中$变量$含义

Settings
Editor › Live Templates
Reset
By default expand with Tab
AndroidParcelable
AndroidXML
Angular
AngularJS
ColdFusion
CustomDefine
arrprint (输出数组元素)
Abbreviation: bsort
Description: 冒泡排序
设置$变量$的含义
Template text:
for(int $INDEX$ = 1; $INDEX$ < $ARRAY$.length; $INDEX$++) {
for(int $INDEX2$ = 0; $INDEX2$ < $ARRAY$.length-$INDEX$; $INDEX2$++) {
if($ARRAY$[$INDEX2$] > $ARRAY$[$INDEX2$+1]){
$ELEMENT_TYPE$ temp = $ARRAY$[$INDEX2$];
$ARRAY$[$INDEX2$] = $ARRAY$[$INDEX2$+1];
$ARRAY$[$INDEX2$+1] = temp;
}
}
}
Edit variables
Options
Expand with Default (Tab)
Reformat according to style
Use static import if possible
Shorten FQ names
Applicable in Java; Java: statement, expression, declaration, comment, string, smart type completion... Change
OK
Cancel
Apply

Settings
Editor › Live Templates
Reset
By default expand with Tab
Edit Template Variables
Name
Expression
Default value
Skip if defined
INDEX
suggestIndexName()
ARRAY
arrayVariable()
INDEX2
suggestIndexName()
ELEMENT_TYPE
componentTypeOf(ARRAY)
变量名
变量值 表达式
OK
Cancel
Appearance & Behavior
Keymap
Editor
General
Font
Color Scheme
Code Style
Inspections
File and Code Templates
File Encodings
Live Templates
File Types
Android Layout Editor
Copyright
Inlay Hints
Android Data Binding
Duplicates
Emmet
GUI Designer
Images
Abbreviation: bsort
Description: 冒泡排序
Template text:
for(int $INDEX$ = 1; $INDEX$ < $ARRAY$.length; $INDEX$++) {
for(int $INDEX2$ = 0; $INDEX2$ < $ARRAY$.length-$INDEX$; $INDEX2$++) {
if($ARRAY$[$INDEX2$] > $ARRAY$[$INDEX2$+1]){
$ELEMENT_TYPE$ temp = $ARRAY$[$INDEX2$];
$ARRAY$[$INDEX2$] = $ARRAY$[$INDEX2$+1];
Edit variables
Options
Expand with Default (Tab)
Reformat according to style
Use static import if possible
Shorten FQ names

④代码模板应用范围

Settings
Editor › Live Templates
By default expand with Tab
AndroidParcelable
AndroidXML
Angular
AngularJS
ColdFusion
CustomDefine
arrprint (输出数组元素)
bsort (冒泡排序)
itarr
sop (输出不换行)
Groovy
GSP
HTML/XML
HTTP Request
iterations
JavaScript
Abbreviation: bsort
Description: 冒泡排序
Template text:
for(int $INDEX$ = 1; $INDEX$ < $ARRAY$.length; $INDEX$++) {
for(int $INDEX2$ = 0; $INDEX2$ < $ARRAY$.length-$INDEX$; $INDEX2$++) {
if($ARRAY$[$INDEX2$] > $ARRAY$[$INDEX2$+1]){
$ELEMENT_TYPE$ temp = $ARRAY$[$INDEX2$];
Edit variables
Options
No applicable contexts.
Define
选择语言
OK
Cancel
Apply
Appearance & Behavior
Keymap
Editor
General
Font
Color Scheme
Code Style
Inspections
File and Code Templates
File Encodings
Live Templates
File Types
Android Layout Editor
Copyright
Inlay Hints
Android Data Binding
Duplicates
Emmet
GUI Designer
Images
Intentions
Language Injections
Spelling
TextMate Bundles

15.2 输出数组的元素快捷模板 arrprint

①在 Settings->Editor->Live Templates 中增加模板，如果没有 CustomDefine 的请看本比较《18.7.6 的 3.自定义代码模板》

②增加 arrprint 快捷模板

```
for(int $INDEX$ = 0; $INDEX$ < $ARRAY$.length; $INDEX$++) {
    System.out.println($ARRAY$[$INDEX$]$END$);
}
```

③定义代码中$变量$含义

Settings
Appearance & Behavior
Keymap
Editor
General
Font
Color Scheme
Code Style
Inspections
File and Code Templates
File Encodings
Live Templates
File Types
Android Layout Editor
Copyright
Inlay Hints
Android Data Binding
Duplicates
Emmet
GUI Designer
Images
Intentions
Language Injections
Spelling
TextMate Bundles
Editor › Live Templates
By default expand with Tab
AndroidParcelable
AndroidXML
Angular
AngularJS
ColdFusion
CustomDefine
arrprint (输出数组元素)
Abbreviation: arrprint
Description: 输出数组元素
Template text:
for(int $INDEX$ = 0; $INDEX$ < $ARRAY$.length; $INDEX$++) {
System.out.println($ARRAY$[$INDEX$]END);
}
Edit variables
Options
Expand with Default (Tab)
Reformat according to style
Use static import if possible
Shorten FQ names
Applicable in Java; Java: statement, expression, declaration, comment, string, smart type completion... Change
OK
Cancel
Apply

Edit Template Variables
Name
Expression
Default value
Skip if defined
INDEX
suggestIndexName()
ARRAY
arrayVariable()
OK
Cancel

④设置语言范围

第 05 章_数组

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

1. 数组的概述

1.1 容器概述

需求分析 1：

现在需要统计某公司 50 个员工的工资情况，例如计算平均工资、找到最高工资等。用前面所学的知识，首先需要声明 50 个变量来分别记住每位员工的工资，这样做会显得很麻烦。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。

需求分析 2：

snipaste_20220317_000101

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。

1.2 数组的概念

- 数组(Array)，是多个相同类型数据按一定顺序排列的集合，并使用一个名字命名，并通过编号的方式对这些数据进行统一管理。
- 数组中的概念
  - 数组名
  - 下标（或索引）

- 元素
- 数组的长度

定价	菜名
20.32	拌海蜇
43.21	龙须菜
43.22	炝冬笋
54.21	玉兰片
64.21	烧鱼头
33.87	炸豆腐
65.33	拔丝山药
78.32	杏仁酪
56.98	烧萝卜
22.65	麻婆豆腐
62.32	红扒鱼翅
71.99	鱼香肉丝
44.55	叫花鸡
39.21	酸菜鱼
29.43	茄汁鱼卷
37.24	粉蒸鸡
48.21	酱茄子
50.62	佛跳墙
71.42	西湖醋鱼
89.49	太极虾
50.26	烤方

数组的特点：

- 数组本身是引用数据类型，而数组中的元素可以是任何数据类型，包括基本数据类型和引用数据类型。
- 创建数组对象会在内存中开辟一整块连续的空间，而数组名中引用的是这块连续空间的首地址。
- 数组的长度一旦确定，就不能修改。
- 我们可以直接通过下标(或索引)的方式调用指定位置的元素，速度很快。

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，只是这里的二维表每一行长度不要求一样。

2、按照元素类型分：

- 基本数据类型的元素：存储基本数据类型的值
- 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）

2. 一维数组的使用

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[];
```

举例：

```
int a[];
int[] a1;
double  b[];
String[] c;  //引用类型变量数组
```

- 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;

    }
}
```

注意：Java 语言中声明数组时不能指定其长度(数组中元素的数)。 例如： int a[5]; //非法

2.2 一维数组的初始化

2.2.1 静态初始化

- 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 = 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...};
```

- 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。

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

- **一维数组静态初始化格式 2：**

```
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
```

- 举例

```
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[] heros = {"袁隆平","邓稼先","钱学森"};

        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("heros 数组: " + heros);//heros 数组: [Ljava.lang.String;@1540e19d
    }
}
```

2.2.2 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），而元素值此时只是默认值，还并未真正附自己期望的值。真正期望的数据需要后续单独一个一个赋值。

- **格式：**

```
数组存储的元素的数据类型[] 数组名字 = new 数组存储的元素的数据类型[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];
```

- [长度]：数组的长度，表示数组容器中可以最多存储多少个元素。
- **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。
- 举例 1

```
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。
```

2.3 一维数组的使用

2.3.1 数组的长度

- 数组的元素总个数，即数组的长度
- 每个数组都有一个属性 length 指明它的长度，例如：a.length 指明数组 a 的长度(元素个数)
- 每个数组都具有长度，而且一旦初始化，其长度是不可变的

2.3.2 数组元素的引用

- 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]

- 数组元素下标可以是整型常量或整型表达式。如 a[3] , b[i] , c[6*i];

- 举例

```
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 1 个元素：" + arr[0]);//下标从 0 开始
        System.out.println("arr 数组的第 2 个元素：" + arr[1]);
        System.out.println("arr 数组的第 3 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 1 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 1 个元素：" + arr[0]);
    }
}
```

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。

- 举例 1

```
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]);
        }
```

```
    }
}
```

- 举例 2

```
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("]");
    }
}
```

2.5 数组元素的默认值

数组是引用类型，当我们使用动态初始化方式创建数组时，元素值只是默认值。例如：

```java
public class Test {
      public static void main(String argv[]){
            int a[]= new int[5];
            System.out.println(a[3]); //a[3]的默认值为0
      }
}
```

对于基本数据类型而言，默认初始化值各有不同。

对于引用数据类型而言，默认初始化值为 null（注意与 0 不同！）

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

```java
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("]");
    }
}
```

3. 一维数组内存分析

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。

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。

第 02 章_JVM 架构-简图

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。

3.3 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
}
```

思考：打印 arr 为什么是[I@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]);
}
```

4. 一维数组的应用

案例 1：升景坊单间短期出租 4 个月，550 元/月（水电煤公摊，网费 35 元/月），空调、卫生间、厨房齐全。屋内均是 IT 行业人士，喜欢安静。所以要求来租者最好是同行或者刚毕业的年轻人，爱干净、安静。

```
public class ArrayTest {
      public static void main(String[] args) {
      int[] arr = new int[]{8,2,1,0,3};
      int[] index = new int[]{2,0,3,2,4,0,1,3,2,3,3};
      String tel = "";
      for(int i = 0;i < index.length;i++){
            tel += arr[index[i]];
      }
      System.out.println("联系方式: " + tel);
      }
}
```

案例 2：从键盘读入学生成绩，找出最高分，并输出学生成绩等级。

- 成绩>=最高分-10 等级为'A'
- 成绩>=最高分-20 等级为'B'
- 成绩>=最高分-30 等级为'C'
- 其余 等级为'D'

提示：先读入学生人数，根据人数创建 int 数组，存放学生成绩。

```
请输入学生人数:5
请输入5个成绩
56
74
89
41
89
最高分是：89
student 0 score is 56 grade is D
student 1 score is 74 grade is B
student 2 score is 89 grade is A
student 3 score is 41 grade is D
student 4 score is 89 grade is A
```

image-20220317004637748

案例 3：抽奖程序的实现

制作一个抽奖程序，可以从控制台输入数名员工的姓名，然后随机抽取一名，作为年终大奖：马尔代夫 7 日游！

5. 多维数组的使用

5.1 概述

- Java 语言里提供了支持多维数组的语法。
- 如果说可以把一维数组当成几何中的线性图形，那么二维数组就相当于是一个表格，像 Excel 中的表格、围棋棋盘一样。

姓名	联系电话
陈伟霆	13387654384
刘诗诗	13845678765
周笔畅	13012393458
鹿晗	13623490545
张艺兴	13334505689
杨幂	13623495439
宋茜	13723490897
李敏镐	13789459034
苍井空	13323409435
赵丽颖	13945893324
迪丽热巴	18623495489
周杰伦	18523489094
胡歌	13723409895
郑爽	13423409548
唐嫣	13721348984
戚薇	13909234895
王俊凯	13012390435
华晨宇	18723490438
佟丽娅	18823467128
刘亦菲	13834589533
邓紫棋	17623485739

二维表1

23	6	78	1	0	5
1	2	3	4	5	6
90	34	78	67	3	6

二维表2

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。

- 应用举例：
- 举例 1：

某公司 2021 年全年各个月份的销售额进行登记。按月份存储，可以使用一维数组。如下：

```
int[] monthData = new int[]{23,43,22,34,55,65,44,67,45,78,67,66};
```

如果改写为按季度为单位存储怎么办呢？

```
int[][] quarterData = new int[][]{{23,43,22},{34,55,65},{44,67,45},{78,67,66}};
```

- 举例 2：

蓝框的几个元素，可以使用一维数组来存储。但现在发现每个元素下还有下拉框，其内部还有元素，那就需要使用二维数组来存储：

- 对于二维数组的理解，我们可以看成是一维数组 array1 又作为另一个一维数组 array2 的元素而存在。
- 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*
- 其实，从数组底层的运行机制来看，其实没有多维数组。

5.2 声明与初始化

5.2.1 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
```

5.2.2 静态初始化

格式：

```
int[][] arr = new int[][]{{3,8,2},{2,7},{9,0,1,6}};
```

定义一个名称为 arr 的二维数组，二维数组中有三个一维数组

- 每一个一维数组中具体元素也都已初始化
  - 第一个一维数组 arr[0] = {3,8,2};
  - 第二个一维数组 arr[1] = {2,7};
  - 第三个一维数组 arr[2] = {9,0,1,6};
- 第三个一维数组的长度表示方式：arr[2].length;
- 注意特殊写法情况：int[] x,y[]; x 是一维数组，y 是二维数组。

- 示例：

举例 1：

```
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
```

举例 2：

```
public class TwoDimensionalArrayInitialize {
    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"}
        };
    }
}
```

5.2.3 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：

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];
        //其中，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
        //其中，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;
```

举例：

```
int[][] arr = new int[3][2];
```

- 定义了名称为 arr 的二维数组
- 二维数组中有 3 个一维数组
- 每一个一维数组中有 2 个元素
- 一维数组的名称分别为 arr[0], arr[1], arr[2]
- 给第一个一维数组 1 脚标位赋值为 78 写法是：`arr[0][1] = 78;`

格式 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;
```

举例：

```
int[][] arr = new int[3][];
```

- 二维数组中有 3 个一维数组。
- 每个一维数组都是默认初始化值 null (注意：区别于格式 1）
- 可以对这个三个一维数组分别进行初始化：arr[0] = new int[3]; arr[1] = new int[1]; arr[2] = new int[2];
- 注：`int[][]arr = new int[][3]; //非法`

练习：

```
/*
 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();
        }

    }
}
```

小结：

Java 中多维数组不必都是规则矩阵形式

int[][] arr = new int[3][2];

或者

```
int[][] arr = new int[3][];
arr[0] = new int[2];
arr[1] = new int[2];
arr[2] = new int[2];
```

```
int[][] arr = new int[3][];
arr[0] = new int[2];
arr[1] = new int[3];
arr[2] = new int[4];
```

5.3 使用说明

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]);//ArrayIndexOutOfBoundsExceptio
n: 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.ArrayIndexOutOfB
oundsException: 4
    }
}
```

二维数组的遍历：

- 格式：

```
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();
}
```

- 举例：

```
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();
        }
    }
}
```

5.4 内存解析

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}
        };
```

```
            //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
             int[][] arr = new int[4][5];

2 2 2 2 2

             //2、确定元素的值
3 3 3 3 3    for (int i = 0; i < arr.length; i++) {
                 for (int j = 0; j < arr.length; j++) {
4 4 4 4 4            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
             //因为每一行的列数不同，这里无法直接确定列数
2 2          int[][]  arr = new int[5][];

3 3 3        //2、确定每一行的列数
             for(int i=0; i<arr.length; i++){
4 4 4 4          arr[i] = new int[i+1];
             }

5 5 5 5 5

             //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;
                 }
             }
```

5.5 应用举例

案例 1：获取 arr 数组中所有元素的和。

提示：使用 for 的嵌套循环即可。

j / i	j = 0	j = 1	j = 2	j = 3
i = 0	3	5	8	-
i = 1	12	9	-	-
i = 2	7	0	6	4

案例 2：声明：int[] x,y[]; 在给 x,y 变量赋值以后，以下选项允许通过编译的是：

声明：int[] x,y[]; 在给 x,y 变量赋值以后，以下选项允许通过编译的是：

```
a)    x[0] = y;
b)    y[0] = x;
c)    y[0][0] = x;
d)    x[0][0] = y;
e)    y[0][0] = x[0];
f)    x = y;
```

提示：
一维数组：int[] x 或者 int x[]
二维数组：int[][] y 或者 int[] y[] 或者 int y[][]

案例 3：使用二维数组打印一个 10 行杨辉三角。

提示：

1. 第一行有 1 个元素, 第 n 行有 n 个元素
2. 每一行的第一个元素和最后一个元素都是 1
3. 从第三行开始, 对于非第一个元素和最后一个元素的元素。即：

   ```
   yanghui[i][j] = yanghui[i-1][j-1] + yanghui[i-1][j];
   ```

[0]	1									
[1]	1	1								
[2]	1	2	1							
[3]	1	3	3	1						
[4]	1	4	6	4	1					
[5]	1	5	10	10	5	1				
[6]	1	6	15	20	15	6	1			
[7]	1	7	21	35	35	21	7	1		
[8]	1	8	28	56	70	56	28	8	1	
[9]	1	9	36	84	126	126	84	36	9	1
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

6. 数组的常见算法

6.1 数值型数组特征值统计

- 这里的特征值涉及到：平均值、最大值、最小值、总和等

举例 1：数组统计：求总和、均值

```
public class TestArrayElementSum {
    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);
    }
}
```

举例 2：求数组元素的总乘积

```
public class TestArrayElementMul {
    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);
    }
}
```

举例 3：求数组元素中偶数的个数

```
public class TestArrayElementEvenCount {
    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);
    }
}
```

举例 4：求数组元素的最大值

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？

```
public class TestArrayMax {
    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 a
rr[0]再比较一次了
            if(arr[i] > max){
                max = arr[i];
            }
        }
```

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

举例 5：找最值及其第一次出现的下标:

```
public class TestMaxIndex {
    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);
    }
}
```

举例 6：找最值及其所有最值的下标（选讲）

```
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+"]");
    }
}
```

举例 7：输入一个整形数组，数组里有正数也有负数。数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。求所有子数组的和的最大值。要求时间复杂度为 O(n)。 例如：输入的数组为 1, -2, 3, -10, -4, 7, 2, -5，和最大的子数组为 3, 10, -4, 7, 2，因此输出为该子数组的和 18。

```
public class Test5 {
      public static void main(String[] args) {
            int[] arr = new int[]{1, -2, 3, 10, -4, 7, 2, -5};
            int i = getGreatestSum(arr);
            System.out.println(i);
      }

      public static int getGreatestSum(int[] arr){
            int greatestSum = 0;
            if(arr == null || arr.length == 0){
                  return 0;
            }
            int temp = greatestSum;
            for(int i = 0;i < arr.length;i++){
                  temp += arr[i];

                  if(temp < 0){
                        temp = 0;
                  }

                  if(temp > greatestSum){
                        greatestSum = temp;
```

```
                }
            }
            if(greatestSum == 0){
                greatestSum = arr[0];
                for(int i = 1;i < arr.length;i++){
                    if(greatestSum < arr[i]){
                        greatestSum = arr[i];
                    }
                }
            }
            return greatestSum;
        }
}
```

举例 8：评委打分

分析以下需求，并用代码实现：

（1）在编程竞赛中，有 10 位评委为参赛的选手打分，分数分别为：5，4，6，8，9，0，1，2，7，3

（2）求选手的最后得分（去掉一个最高分和一个最低分后其余 8 位评委打分的平均值）

6.2 数组元素的赋值

举例 1：杨辉三角（前面已讲过）

举例 2：使用简单数组

(1)创建一个名为 ArrayTest 的类，在 main()方法中声明 array1 和 array2 两个变量，他们是 int[]类型的数组。

(2)使用大括号{}，把 array1 初始化为 8 个素数：2,3,5,7,11,13,17,19。

(3)显示 array1 的内容。

(4)赋值 array2 变量等于 array1，修改 array2 中的偶索引元素，使其等于索引值(如 array[0]=0,array[2]=2)。打印出 array1。 array2 = array1;

思考：array1 和 array2 是什么关系？

拓展：修改题目，实现 array2 对 array1 数组的复制

举例 3：一个数组，让数组的每个元素去除第一个元素，得到的商作为被除数所在位置的新值。

```
public class Test3 {
      public static void main(String[] args) {
            int[] arr = new int[]{12,43,65,3,-8,64,2};

//          for(int i = 0;i < arr.length;i++){
```

```
//                  arr[i] = arr[i] / arr[0];
//              }
            for(int i = arr.length -1;i >= 0;i--){
                  arr[i] = arr[i] / arr[0];
            }
            //遍历arr
            for(int i = 0;i < arr.length;i++){
                  System.out.print(arr[i] + " ");
            }
      }
}
```

举例 4：创建一个长度为 6 的 int 型数组，要求数组元素的值都在 1-30 之间，且是随机赋值。同时，要求元素的值各不相同。

```
public class Test4 {
      // 5-67 Math.random() * 63 + 5;
      @Test
      public void test1() {
            int[] arr = new int[6];
           for (int i = 0; i < arr.length; i++) {// [0,1) [0,30) [1,31)
                  arr[i] = (int) (Math.random() * 30) + 1;

                  boolean flag = false;
                  while (true) {
                        for (int j = 0; j < i; j++) {
                              if (arr[i] == arr[j]) {
                                    flag = true;
                                    break;
                              }
                        }
                        if (flag) {
                              arr[i] = (int) (Math.random() * 30) + 1;
                              flag = false;
                              continue;
                        }
                        break;
                  }
            }

            for (int i = 0; i < arr.length; i++) {
                  System.out.println(arr[i]);
            }
      }
      //更优的方法
      @Test
      public void test2(){
            int[] arr = new int[6];
           for (int i = 0; i < arr.length; i++) {// [0,1) [0,30) [1,31)
                  arr[i] = (int) (Math.random() * 30) + 1;
```

```
                        for (int j = 0; j < i; j++) {
                              if (arr[i] == arr[j]) {
                                    i--;
                                    break;
                              }
                        }
                  }

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

举例 5：回形数

从键盘输入一个整数（1~20），则以该数字为矩阵的大小，把 1,2,3...n*n 的数字按照顺时针螺旋的形式填入其中。

例如： 输入数字 2，则程序输出： 1 2 4 3

输入数字 3，则程序输出： 1 2 3 8 9 4 7 6 5 输入数字 4， 则程序输出： 1 2 3 4 12 13 14 5 11 16 15 6 10 9 8 7

```
//方式1
public class RectangleTest {
      public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            System.out.println("输入一个数字");
            int len = scanner.nextInt();
            int[][] arr = new int[len][len];

            int s = len * len;
            /*
             * k = 1:向右
             * k = 2:向下
             * k = 3:向左
             * k = 4:向上
             */
            int k = 1;
            int i = 0,j = 0;
            for(int m = 1;m <= s;m++){
                  if(k == 1){
                        if(j < len && arr[i][j] == 0){
                              arr[i][j++] = m;
                        }else{
                              k = 2;
                              i++;
```

```
                        j--;
                        m--;
                    }
                }else if(k == 2){
                    if(i < len && arr[i][j] == 0){
                        arr[i++][j] = m;
                    }else{
                        k = 3;
                        i--;
                        j--;
                        m--;
                    }
                }else if(k == 3){
                    if(j >= 0 && arr[i][j] == 0){
                        arr[i][j--] = m;
                    }else{
                        k = 4;
                        i--;
                        j++;
                        m--;
                    }
                }else if(k == 4){
                    if(i >= 0 && arr[i][j] == 0){
                        arr[i--][j] = m;
                    }else{
                        k = 1;
                        i++;
                        j++;
                        m--;
                    }
                }
            }

            //遍历
            for(int m = 0;m < arr.length;m++){
                for(int n = 0;n < arr[m].length;n++){
                    System.out.print(arr[m][n] + "\t");
                }
                System.out.println();
            }
        }
}

//方式2
/*
        01 02 03 04 05 06 07
        24 25 26 27 28 29 08
        23 40 41 42 43 30 09
        22 39 48 49 44 31 10
        21 38 47 46 45 32 11
```

```
      20 37 36 35 34 33 12
      19 18 17 16 15 14 13
 */
public class RectangleTest1 {

      public static void main(String[] args) {
            int n = 7;
            int[][] arr = new int[n][n];

            int count = 0; //要显示的数据
            int maxX = n-1; //x 轴的最大下标
            int maxY = n-1; //Y 轴的最大下标
            int minX = 0; //x 轴的最小下标
            int minY = 0; //Y 轴的最小下标
            while(minX<=maxX) {
                  for(int x=minX;x<=maxX;x++) {
                        arr[minY][x] = ++count;
                  }
                  minY++;
                  for(int y=minY;y<=maxY;y++) {
                        arr[y][maxX] = ++count;
                  }
                  maxX--;
                  for(int x=maxX;x>=minX;x--) {
                        arr[maxY][x] = ++count;
                  }
                  maxY--;
                  for(int y=maxY;y>=minY;y--) {
                        arr[y][minX] = ++count;
                  }
                  minX++;
            }


            for(int i=0;i<arr.length;i++) {
                  for(int j=0;j<arr.length;j++) {
                        String space = (arr[i][j]+"").length()==1 ? "0":
"";
                        System.out.print(space+arr[i][j]+" ");
                  }
                  System.out.println();
            }
      }
}
```

6.3 数组元素的反转

实现思想：数组对称位置的元素互换。

```java
public class TestArrayReverse1 {
    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

```java
public class TestArrayReverse2 {
    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]);
        }
    }
}
```

练习：

某个政府机构使用传真传递数据信息，数据是 8 位的整数。为了确保数据安全，在传递过程中需要加密。加密规则如下： 首先，将各个位上的数据倒序，然后将每位数字都加上 6，再用和除以 8 的余数代替该数字，最后交换首末位上的数字。

请任意给定一个 8 位的整数，并将加密后的结果在控制台打印出来。

6.4 数组的扩容与缩容

数组的扩容

题目：现有数组 int[] arr = new int[]{1,2,3,4,5}; ，现将数组长度扩容 1 倍，并将 10,20,30 三个数据添加到 arr 数组中，如何操作？

```
public class ArrTest1 {
    public static void main(String[] args) {

        int[] arr = new int[]{1,2,3,4,5};
        int[] newArr = new int[arr.length << 1];

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

        newArr[arr.length] = 10;
        newArr[arr.length + 1] = 20;
        newArr[arr.length + 2] = 30;

        arr = newArr;

        //遍历arr
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}
```

数组的缩容

题目：现有数组 int[] arr={1,2,3,4,5,6,7}。现需删除数组中索引为 4 的元素。

```
public class ArrTest2 {
    public static void main(String[] args) {

        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        //删除数组中索引为4 的元素
        int delIndex = 4;
        //方案1:
        /*//创建新数组
        int[] newArr = new int[arr.length - 1];

        for (int i = 0; i < delIndex; i++) {
            newArr[i] = arr[i];
        }
```

```
        for (int i = delIndex + 1; i < arr.length; i++) {
            newArr[i - 1] = arr[i];
        }

        arr = newArr;
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }*/

        //方案2:
        for (int i = delIndex; i < arr.length - 1; i++) {
            arr[i] = arr[i + 1];
        }
        arr[arr.length - 1] = 0;

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

6.5 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

```
public class TestArrayOrderSearch {
    //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

step 1:
mid
-
+
2 5 7 8 10 15 18 20 22 25 28
step 2:
mid
-
+
18 20 22 25 28
step 3:
mid
-
+
18 20

image-20220514162231846

```
//二分法查找：要求此数组必须是有序的。
int[] arr3 = new int[]{-99,-54,-2,0,2,33,43,256,999};
boolean isFlag = true;
int value = 256;
//int value = 25;
int head = 0;//首索引位置
int end = arr3.length - 1;//尾索引位置
while(head <= end){
    int middle = (head + end) / 2;
    if(arr3[middle] == value){
        System.out.println("找到指定的元素，索引为：" + middle);
        isFlag = false;
        break;
    }else if(arr3[middle] > value){
        end = middle - 1;
    }else{//arr3[middle] < value
        head = middle + 1;
    }
}
```

```
if(isFlag){
    System.out.println("未找打指定的元素");
}
```

6.6 数组元素排序

6.6.1 算法概述

- **定义**
  - 排序：假设含有 n 个记录的序列为{R1，R2，...,Rn},其相应的关键字序列为{K1，K2，...,Kn}。将这些记录重新排序为{Ri1,Ri2,...,Rin},使得相应的关键字值满足条 Ki1<=Ki2<=...<=Kin,这样的一种操作称为排序。
  - 通常来说，排序的目的是快速查找。
- 算法的 5 大特征：

输入（Input）	有0个或多个输入数据，这些输入必须有清楚的描述和定义
输出（Output）	至少有1个或多个输出结果，不可以没有输出结果
有穷性（有限性，Finiteness）	算法在有限的步骤之后会自动结束而不会无限循环，并且每一个步骤可以在可接受的时间内完成
确定性（明确性，Definiteness）	算法中的每一步都有确定的含义，不会出现二义性
可行性（有效性，Effectiveness）	算法的每一步都是清楚且可行的，能让用户用纸笔计算而求出答案

说明：满足确定性的算法也称为：确定性算法。现在人们也关注更广泛的概念，例如考虑各种非确定性的算法，如并行算法、概率算法等。另外，人们也关注并不要求终止的计算描述，这种描述有时被称为过程（procedure）。

- **衡量排序算法的优劣：**
  - 时间复杂度：分析关键字的比较次数和记录的移动次数
    - 常见的算法时间复杂度由小到大依次为：O(1)＜O(log2n)＜O(n)＜O(nlog2n)＜O(n2)＜O(n3)＜...＜O(2n)＜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。但有时我们想

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：分析排序算法中需要多少辅助内存

  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：若两个记录 A 和 B 的关键字值相等，但排序后 A、B 的先后次序保持不变，则称这种排序算法是稳定的。

6.6.2 排序算法概述

- **排序算法分类：内部排序和外部排序**

  - 内部排序：整个排序过程不需要借助于外部存储器（如磁盘等），所有排序操作都在内存中完成。

  - 外部排序：参与排序的数据非常多，数据量非常大，计算机无法把整个排序过程放在内存中完成，必须借助于外部存储器（如磁盘）。外

部排序最常见的是多路归并排序。可以认为外部排序是由多次内部排序组成。

- **十大内部排序算法**

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

6.6.3 冒泡排序

介绍：

Java 中的经典算法之冒泡排序（Bubble Sort）。冒泡排序的原理非常简单，它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。

image-20220516094637228

排序思想：

1. 比较相邻的元素。如果第一个比第二个大（升序），就交换他们两个。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

3. 针对所有的元素重复以上的步骤，除了最后一个。

4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较为止。

动态演示：https://visualgo.net/zh/sorting

```
/*
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]+"  ");
        }
    }
}
```

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]+"  ");
            }
      }
}
```

6.6.4 快速排序

介绍：快速排序通常明显比同为 O(nlogn)的其他算法更快，因此常被采用，而且快排采用了分治法的思想，所以在很多笔试面试中能经常看到快排的影子。可见掌握快排的重要性。

快速排序（Quick Sort）由图灵奖获得者 Tony Hoare 发明，被列为 20 世纪十大算法之一，是迄今为止所有内排序算法中速度最快的一种。冒泡排序的升级版，交换排序的一种。快速排序的时间复杂度为 O(nlog(n))。

排序思想：

1. 从数列中挑出一个元素，称为"基准"（pivot），
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

4. 递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

动态演示：https://visualgo.net/zh/sorting

图示 1：

图示 2：

6.6.5 内部排序性能比较与选择

- 性能比较
  - **从平均时间而言：**快速排序最佳。但在最坏情况下时间性能不如堆排序和归并排序。

- **从算法简单性看**：由于直接选择排序、直接插入排序和冒泡排序的算法比较简单，将其认为是简单算法。对于 Shell 排序、堆排序、快速排序和归并排序算法，其算法比较复杂，认为是复杂排序。
- **从稳定性看**：直接插入排序、冒泡排序和归并排序时稳定的；而直接选择排序、快速排序、 Shell 排序和堆排序是不稳定排序
- **从待排序的记录数 n 的大小看**，n 较小时，宜采用简单排序；而 n 较大时宜采用改进排序。

- 选择
  - 若 n 较小(如 n≤50)，可采用直接插入或直接选择排序。 当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插入，应选直接选择排序为宜。
  - 若文件初始状态基本有序(指正序)，则应选用直接插入、冒泡或随机的快速排序为宜；
  - 若 n 较大，则应采用时间复杂度为 O(nlgn)的排序方法：快速排序、堆排序或归并排序。

7. Arrays 工具类的使用

java.util.Arrays 类即为操作数组的工具类，包含了用来操作数组（比如排序和搜索）的各种方法。 比如：

- 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
- 比较两个数组是否相等
  - 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

举例：java.util.Arrays 类的 sort()方法提供了数组元素排序功能：

```
import java.util.Arrays;
public class SortTest {
      public static void main(String[] args) {
            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));
      }
}
```

8. 数组中的常见异常

8.1 数组角标越界异常

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 TestArrayIndexOutOfBoundsException {
    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
      //  System.out.println("最后一个元素: " + arr[arr.length]);//错误，
下标越界
        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。

8.2 空指针异常

观察一下代码，运行后会出现什么结果。

```java
public class TestNullPointerException {
    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

```java
public static void main(String[] args) {
    //定义数组
    int[][] arr = new int[3][];

    System.out.println(arr[0][0]);//NullPointerException
}
```

小结：空指针异常情况

```java
            //举例一:
//          int[] arr1 = new int[10];
//          arr1 = null;
//          System.out.println(arr1[9]);

            //举例二:
//          int[][] arr2 = new int[5][];
//          //arr2[3] = new int[10];
//          System.out.println(arr2[3][3]);

            //举例三:
```

```
String[] arr3 = new String[10];
System.out.println(arr3[2].toString());
```

第 06 章_面向对象编程（基础）

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

学习面向对象内容的三条主线

- Java 类及类的成员
- 面向对象的特征
- 其他关键字的使用

1. 面向对象编程概述

1.1 程序设计语言的任务

Java 是一种计算机程序设计语言。所有的计算机程序一直都围绕着如下两件事展开：

1. 如何表示和存储数据
   - 基本数据类型的常量和变量：表示和存储一个个独立的数据
   - 对象：表示和存储与某个具体事物相关的多个数据（例如：某个学生的姓名、年龄、联系方式等）
   - 数据结构：表示和存储一组对象，数据结构有数组（或顺序表）、链表、栈、队列、散列表、二叉树等
2. 基于这些数据都有什么操作行为，其实就是实现什么功能
   - 数据的输入和输出
   - 基于一个或两个数据的操作：赋值运算、算术运算、比较运算、逻辑运算等
   - 基于一组数据的操作：统计分析、查找最大值、查找元素、排序、遍历等

尚硅谷

1.2 程序设计方法

高级语言中，存在着两种程序设计的编程思想：面向过程的思想、面向对象的思想。基于不同的思想会产生不同的程序设计方法。

1. 面向过程的程序设计思想（Process-Oriented Programming），简称 POP
   - 关注的焦点是过程：过程就是操作数据的步骤，如果某个过程的实现代码在很多地方重复出现，那么就可以把这个过程抽象为一个函数，这样就可以大大简化冗余代码，也便于维护。

- 代码结构：以函数为组织单位。独立于函数之外的数据称为全局数据，在函数内部的称为局部数据。
  - 是一种“执行者思维”，适合解决简单问题

1. 面向对象的程序设计思想（ Object Oriented Programming），简称 OOP
   - 关注的焦点是类：面向对象思想就是在计算机程序设计过程中，参照现实中事物，将事物的属性特征、行为特征抽象出来，用类来表示。某个事物的一个具体个体称为实例或对象。
   - 代码结构：以类为组织单位。每种事物都具备自己的属性（即表示和存储数据，在类中用成员变量表示）和行为/功能（即操作数据，在类中用成员方法表示）。
   - 是一种“设计者思维”，适合解决复杂问题。

早期先有面向过程思想，随着软件规模的扩大，问题复杂性的提高，面向过程的弊端越来越明显的显示出来，出现了面向对象思想并成为目前主流的方式。两者都贯穿于软件分析、设计和开发各个阶段，对应面向对象就分别称为面向对象分析（OOA）、面向对象设计（OOD）和面向对象编程（OOP）。C 语言是一种典型的面向过程语言，Java 是一种典型的面向对象语言。

1.3 由问题的引入看两种设计

思考 1：如何开车？

面向过程思想思考问题时，我们首先思考“怎么按步骤实现？”并将步骤对应成方法，一步一步，最终完成。 这个适合简单任务，不需要过多协作的情况。针对如何开车，可以列出步骤：

image-20220520154055429

面向过程适合简单、不需要协作的事务，重点关注如何执行。

思考 2：如何造车？

造车太复杂，需要很多协作才能完成。此时我们思考的是“车怎么设计？”，而不是“怎么按特定步骤造车的问题”。这就是思维方式的转变。这里后者是面向过程思想，前者就是面向对象思想。所以，面向对象(Oriented-Object)思想更契合人的思维模式。

比如，我们用面向对象思想思考“如何设计车”：

a49bd1fc8152291f0c5b43ee3322672d

自然地，我们就会从“车由什么组成”开始思考。发现，车由如下结构组成：

image-20220520155357634

为了便于协作，我们找轮胎厂完成制造轮胎的步骤，发动机厂完成制造发动机的步骤；这样，发现大家可以同时进行车的制造，最终进行组装，大大提高了效率。但是，具体到轮胎厂的一个流水线操作，仍然是有步骤的，还是离不开执行者，离不开面向过程思维！

因此，**面向对象可以帮助我们从宏观上把握、从整体上分析整个系统。** 但是，具体到实现部分的微观操作（就是一个个方法），仍然需要面向过程的思路去处理。

> 康师傅有话说：
>
> 我们千万不要把面向过程和面向对象对立起来。他们是相辅相成的。面向对象离不开面向过程！

类比举例 1：

当需求单一，或者简单时，我们一步步去操作没问题，并且效率也挺高。可随着需求的更改，功能的增多，发现需要面对每一个步骤很麻烦了，这时就开始思索，能不能把这些步骤和功能进行封装，封装时根据不同的功能，进行不同的封装，功能类似的封装在一起。这样结构就清晰了很多。用的时候，找到对应的类就可以了。这就是面向对象的思想。

类比举例 2：人把大象装进冰箱

- 面向过程

  1.打开冰箱

  2.把大象装进冰箱

  3.把冰箱门关住

  面向过程开发，其实就是面向着具体的每一个步骤和过程，把每一个步骤和过程完成，然后由这些功能方法相互调用，完成需求。

- 面向对象

```
人{
    打开（冰箱）{
            冰箱.开门();
    }
```

```
        操作(大象){
                大象.进入(冰箱);
        }
        关闭(冰箱){
            冰箱.关门();
        }
    }

    冰箱{
         开门(){ }
         关门(){ }
    }

    大象{
         进入(冰箱){   }
    }
```

1.4 面向对象思想分析问题的思路

1、根据问题需要，选择问题所针对的现实世界中的实体。

2、从实体中寻找解决问题相关的属性和功能，这些属性和功能就形成了概念世界中的类。

3、把抽象的实体用计算机语言进行描述，形成计算机世界中类的定义。即借助某种程序语言，把类构造成计算机能够识别和处理的数据结构。

4、将类实例化成计算机世界中的对象。对象是计算机世界中解决问题的最终工具。

练习：抽象出下面系统中的“类”及其关系

image-20220319180839009

1.5 如何掌握这种思想？

image-20220319180301856

2. Java 语言的基本元素：类和对象

2.1 引入

人认识世界，其实就是面向对象的。比如，我们认识认识一下美人鱼，都没见过。现在摆 4 个照片。

image-20220520162643547

经过大家“仔细学习”，一定能总结出什么才是美人鱼。通常具备如下一些特征：

- 女孩
- 有鱼尾
- 美丽

这个总结的过程，其实是抽象化的过程。抽象出来的美人鱼的特征，我们就可以归纳为一个美人鱼类。而上面图片中的都是这个类呈现出来的具体的对象。

2.2 类和对象概述

类(Class)和对象(Object)是面向对象的核心概念。

1、什么是类

类：具有相同特征的事物的抽象描述，是抽象的、概念上的定义。

2、什么是对象

对象：是实际存在的该类事物的每个个体，是具体的，因而也称为实例(instance)。

手机的设计图（抽象的）

真正的手机（具体的）

image-20220503101940126

- 可以理解为：类 => 抽象概念的人；对象 => 实实在在的某个人
- 面向对象程序设计的重点是类的设计

2.3 类的成员概述

类的设计，其实就是类的成员的设计。

- 现实世界的生物体，大到鲸鱼，小到蚂蚁，都是由最基本的细胞构成的。同理，Java 代码世界是由诸多个不同功能的类构成的。

image-20220319211527249

- 现实生物世界中的细胞又是由什么构成的呢？细胞核、细胞质、... 那么，Java 中用类 class 来描述事物也是如此。类，是一组相关属性和行为的集合。
  - **属性**：就是该类事物的状态信息。对应类中的成员变量
  - **行为**：就是该类事物要做什么操作，或者基于事物的状态能做什么。对应类中的成员方法

20220319_211611

- 类的最基本的两个成员就是：成员变量、成员方法
  - 成员变量 = Field = 属性
  - (成员)方法 = Method = 函数

```
class Person {
    String name;
    int age;
    boolean isMarried;

    public void walk(){
        System.out.println("人走路...");
    }
    public String display(){
        return "名字是: "+name+",年龄是: "+age+",Married:"+isMarried;
    }
}
```

属性，或成员变量

方法，或函数

- 对象是类的一个**实例**，必然具备该类事物的属性和行为。

2.4 类的定义

类的定义使用关键字：class。格式如下：

```
[修饰符] class 类名{
      属性声明;
    方法声明;
}
```

举例 1：

```
public class Student{

}
```

举例 2：

```
public class Person{
    int age ;                          //声明变量 age

    public void showAge() {          //声明方法showAge( )
          System.out.println("年龄为:" + age);
    }
}
```

举例 3：

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

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

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

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

```
    }
}
```

2.5 对象的创建

image-20220319213201568

- 创建对象，使用关键字：**new**
- 创建对象语法：

```
//方式1：
new 类名()//也称为匿名对象

//方式2：给创建的对象命名
//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();
```

- 使用“对象名.对象成员”的方式访问对象成员（包括属性和方法）

举例 1：

```
//声明Animal 类
public class Animal { //动物类
    public int legs;

    public void eat() {
        System.out.println("Eating.");
    }

    public void move() {
        System.out.println("Move.");
    }
}
```

```java
//声明测试类
public class AnimalTest {
    public static void main(String args[]) {
        //创建对象
        Animal xb = new Animal();
        xb.legs = 4;//访问属性
        System.out.println(xb.legs);
        xb.eat();//访问方法
        xb.move();//访问方法
    }
}
```

图示理解：

举例 2：针对上面的举例 3：类的实例化（创建类的对象）

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

        p.dog.type = "柯基犬";
        p.dog.nickname = "小白";

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

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

2.6 对象的再理解

1、类与对象的关系错误理解

曰：“白马非马，可乎？”
曰：“可。”
曰：“何哉？”
曰：“马者，所以命形也。白者，所以命色也。命色者，非命形也，故曰白马非马。”

2、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
```

```
    }
}
```

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。**

3、创建类的多个对象，相关之间没有关联关系

创建了一个类的多个对象，对于类中定义的属性，每个对象都拥有各自的一套副本，且互不干扰。

```
public class Zoo {
    public static void main(String args[]) {
        Animal xb = new Animal();
        Animal xh = new Animal();
        xb.legs = 4;
        xh.legs = 0;
        System.out.println(xb.legs); // 4
        System.out.println(xh.legs); // 0

        xb.legs = 2;
        System.out.println(xb.legs); // 2
        System.out.println(xh.legs); // 0
    }
}
```

4、匿名对象

- 我们也可以不定义对象的句柄，而直接调用这个对象的方法。这样的对象叫做匿名对象。
  - 如：new Person().shout();
- 使用情况
  - 如果对一个对象只需要进行一次方法调用，那么就可以使用匿名对象。
  - 我们经常将匿名对象作为实参传递给一个方法调用。

5、类的访问机制

- 在一个类中的访问机制：类中的方法可以直接访问类中的成员变量。
  - （例外：static 方法访问非 static，编译不通过。）
- 在不同类中的访问机制：先创建要访问类的对象，再用对象访问类中定义的成员。

2.7 对象的内存解析

1、内存结构说明：

堆（Heap）：此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在 Java 虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配。

栈（Stack）：是指虚拟机栈。虚拟机栈用于存储局部变量等。局部变量表存放了编译期可知长度的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不等同于对象本身，是对象在堆内存的首地址）。 方法执行完，自动释放。

方法区（Method Area）：用于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。

2、代码举例

举例 1：

```
class Person { //类：人
    String name;
    int age;
    boolean isMale;
}

public class PersonTest { //测试类
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.name = "赵同学";
        p1.age = 20;
        p1.isMale = true;

        Person p2 = new Person();
        p2.age = 10;

        Person p3 = p1;
        p3.name = "郭同学";
    }
}
```

内存解析图：

尚硅谷

image-20220319215723183

说明：

- 堆：凡是 new 出来的结构(对象、数组)都放在堆空间中。
- 对象的属性存放在堆空间中。

2.8 练习

练习 1：

编写教师类和学生类，并通过测试类创建对象进行测试。

image-20220319214928206

练习 2：根据代码，画出内存图

```
class Car {
    String color = "red";
    int num = 4;

    void show() {
        System.out.println("color=" + color + "..num=" + num);
    }
}

class CarTest {
    public static void main(String[] args) {
        Car c1 = new Car();   //建立对象c1
        Car c2 = new Car();   //建立对象c2
        c1.color = "blue";   //对对象的属性进行修改
        c1.show();   //使用对象的方法
        c2.show();
    }
}
```

3. 类的成员之一：成员变量(field)

3.1 如何声明成员变量

- 语法格式：

```
[修饰符 1] class 类名{
    [修饰符 2] 数据类型 成员变量名 [= 初始化值];
}
```

- 说明：
  - 位置要求：必须在类中，方法外
  - 修饰符 2(暂不考虑)
    - 常用的权限修饰符有：private、缺省、protected、public
    - 其他修饰符：static、final
  - 数据类型
    - 任何基本数据类型(如 int、Boolean) 或 任何引用数据类型。
  - 成员变量名
    - 属于标识符，符合命名规则和规范即可。
  - 初始化值
    - 根据情况，可以显式赋值；也可以不赋值，使用默认值

示例：

```
public class Person{
      private int age;              //声明private 变量 age
      public String name = “Lila”;     //声明public 变量 name
}
```

3.2 成员变量 vs 局部变量

1、变量的分类：成员变量与局部变量

- 在方法体外，类体内声明的变量称为成员变量。
- 在方法体内部等位置声明的变量称为局部变量。

```java
public class Student {
    String name;    //成员变量
    int age;        //成员变量

    public void sayHello(){
        String info = "hello!";        //局部变量
        System.out.println(info);
    }
}
```

image-20220511101608038

image-20220319230744617

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。

2、成员变量 与 局部变量 的对比

- 相同点
  - 变量声明的格式相同： 数据类型 变量名 = 初始化值
  - 变量必须先声明、后初始化、再使用。
  - 变量都有其对应的作用域。只在其作用域内是有效的

- 不同点

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。

4、作用域 （1）实例变量：通过对象就可以使用，本类中直接调用，其他类中“对象.实例变量” （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。

3、对象属性的默认初始化赋值

当一个对象被创建时，会对其中各种类型的成员变量自动进行初始化赋值。

成员变量类型	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

image-20220319231821747

4、举例

```java
class Person {//人类
    //1.属性
    String name;//姓名
    int age = 1;//年龄
    boolean isMale;//是否是男性

    public void show(String nation) {
        //nation:局部变量
        String color;//color:局部变量
        color = "yellow";
    }
}

//测试类
class PersonTest {
    public static void main(String[] args) {
        Person p = new Person();
        p.show("CHN");
    }
}
```

尚硅谷

image-20220319231703240

4. 类的成员之二：方法(method)

4.1 方法的引入

比如，《街霸》游戏中，每次人物出拳、出脚或跳跃等动作都需要编写 50-80 行的代码，在每次出拳、出脚或跳跃的地方都需要重复地编写这 50-80 行代码，这样程序会变得很臃肿，可读性也非常差。为了解决代码重复编写的问题，可以将出拳、出脚或跳跃的代码提取出来放在一个{}中，并为这段代码起个名字，这样在每次的出拳、出脚或跳跃的地方通过这个名字来调用这个{}的代码就可以了。

上述过程中，所提取出来的代码可以被看作是程序中定义的一个方法，程序在需要出拳、出脚或跳跃时调用该方法即可。

4.2 方法(method、函数)的理解

- 方法是类或对象行为特征的抽象，用来完成某个功能操作。在某些语言中也称为函数或过程。
- 将功能封装为方法的目的是，可以实现代码重用，减少冗余，简化代码
- Java 里的方法不能独立存在，所有的方法必须定义在类里。
- 举例 1：
  - Math.random()的 random()方法
  - Math.sqrt(x)的 sqrt(x)方法
  - System.out.println(x)的 println(x)方法
  - new Scanner(System.in).nextInt()的 nextInt()方法
- 举例 2：

```
public class Person{
    private int age;
    public int getAge()  {  //声明方法 getAge()
            return age;
    }
    public void setAge(int i) {  //声明方法 setAge
            age = i;        //将参数 i 的值赋给类的成员变量 age
    }
}
```

4.3 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。

（3）方法中只能调用方法或属性，不可以在方法内部定义方法。

正确示例：

```
类{
    方法1(){

    }
    方法2(){

    }
}
```

错误示例：

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

            }
    }
}
```

（4）方法不调用不执行，调用一次执行一次。

4.4 如何声明方法

1、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执行的代码。对于调用者来说，不了解方法体如何实现的，并不影响方法的使用。

（2）方法头可能包含 5 个部分

- **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，权限修饰符有 public、protected、private。在讲封装性之前，我们先默认使用 pulbic 修饰方法。

  - 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。咱们在讲 static 前先学习实例方法。

- **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。
  - 无返回值，则声明：void
  - 有返回值，则声明出返回值类型（可以是任意类型）。与方法体中“`return` 返回值”搭配使用
- **方法名**：属于标识符，命名时遵循标识符命名规则和规范，“见名知意”
- **形参列表**：表示完成方法体功能时需要外部提供的数据列表。可以包含零个，一个或多个参数。
  - 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【第 09 章-异常处理】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

（4）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

补充：方法的分类：按照是否有形参及返回值

	无返回值	有返回值
无形参	void 方法名（）{}	返回值的类型 方法名（）{}
有形参	void 方法名（形参列表）{}	返回值的类型 方法名（形参列表）{}

image-20220320000047155

2、类比举例

方法参数

方法体

返回值

image-20220503102323689

3、代码示例：

```
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
     */
    public 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
```

```
     */
    public 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
     */
    public 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
     */
    public int max(int a, int b){
        return a > b ? a : b;
    }
}
```

4.5 如何调用实例方法

方法通过方法名被调用，且只有被调用才会执行。

1、方法调用语法格式

对象.方法名([实参列表])

2、示例

举例 1：

```
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 getIntBetweenOneToHu
ndred
        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));
    }
}
```

举例 2:

```
//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()
```

4.6 关键字 return 的使用

- return 在方法中的作用：
  - 作用 1：结束一个方法
  - 作用 2：结束一个方法的同时，可以返回数据给方法的调用者
- 注意点：在 return 关键字的直接后面不能声明执行语句

4.7 方法调用内存分析

- 方法没有被调用的时候，都在方法区中的字节码文件(.class)中存储。
- 方法被调用的时候，需要进入到栈内存中运行。方法每调用一次就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。

举例分析：

```
/**
 * @author 尚硅谷-宋红康
 * @create 9:21
 */
public class Person {
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.eat();

    }
    public static void eat() {
        sleep();
        System.out.println("人：吃饭");
        doSport();
    }
    public static void sleep(){
        System.out.println("人：睡觉");
    }
    public static void doSport(){
        System.out.println("人：运动");
```

```
    }
}
```

内存分析：

```java
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());
    }
}
```

```java
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());
    }
}
```

```java
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();
    }
}
```

4.8 练习

练习 1：创建一个 Person 类，其定义如下：

image-20220320001337145

要求：

(1)创建 Person 类的对象，设置该对象的 name、age 和 sex 属性，调用 study 方法，输出字符串“studying”，调用 showAge()方法显示 age 值，调用 addAge()方法给对象的 age 属性值增加 2 岁。 (2)创建第二个对象，执行上述操作，体会同一个类的不同对象之间的关系。

练习 2：利用面向对象的编程方法，设计圆类 Circle，包含属性（半径）和计算圆面积的方法。定义测试类，创建该 Circle 类的对象，并进行测试。

练习 3：

3.1 编写程序，声明一个 method 方法，在方法中打印一个 `10*8` 的*型矩形，在 main 方法中调用该方法。

3.2 修改上一个程序，在 method 方法中，除打印一个 `10*8` 的*型矩形外，再计算该矩形的面积，并将其作为方法返回值。在 main 方法中调用该方法，接收返回的面积值并打印。

3.3 修改上一个程序，在 method 方法提供 m 和 n 两个参数，方法中打印一个 m*n 的*型矩形，并计算该矩形的面积， 将其作为方法返回值。在 main 方法中调用该方法，接收返回的面积值并打印。

练习 4：声明一个日期类型 MyDate：有属性：年 year,月 month，日 day。创建 2 个日期对象，分别赋值为：你的出生日期，你对象的出生日期，并显示信息。

练习 5：用面向对象的方式编写用户登录程序。

用户类：

- 属性：用户名，密码
- 方法：登录

界面类：

- 在界面类中添加 main 方法，接受用户输入，并调用用户类的登录方法进行验证。
  - 输出：
    - 登录失败：用户名或密码错误！
    - 登录成功：欢迎你，用户名！

提示：

```
//判断字符串是否相等，使用 equals 方法
if(str1.equals(str2)){
      //字符串 str1 和 str2 相等
}
```

5. 对象数组

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

1、案例

定义类 Student，包含三个属性：学号 number(int)，年级 state(int)，成绩 score(int)。 创建 20 个学生对象，学号为 1 到 20，年级和成绩都由随机数确定。

问题一：打印出 3 年级(state 值为 3）的学生信息。

问题二：使用冒泡排序按学生成绩排序，并遍历所有学生信息

提示：

1) 生成随机数：Math.random()，返回值类型 double;

2) 四舍五入取整：Math.round(double d)，返回值类型 long。

```
/*
 * 定义类Student，包含三个属性：学号number(int)，年级state(int)，成绩scor
e(int)。
 */
public class Student {

      int number;//学号
      int state;//年级
      int score;//成绩


      public void info(){
            System.out.println("number : " + number
                        + ",state : " + state + ",score : " + score);
      }

}

public class StudentTest {

      public static void main(String[] args) {

            // Student s1 = new Student();
            // s1.number = 1;
            // s1.state = (int)(Math.random() * 6 + 1);//[1,6]
            // s1.score = (int)(Math.random() * 101);//[0,100]
            //
            // Student s2 = new Student();
            // s2.number = 2;
            // s2.state = (int)(Math.random() * 6 + 1);//[1,6]
            // s2.score = (int)(Math.random() * 101);//[0,100]
            //
            // //....
            // 对象数组
            // String[] arr = new String[10];
            // 数组的创建
            Student[] students = new Student[20];
            // 通过循环结构给数组的属性赋值
            for (int i = 0; i < students.length; i++) {
                  // 数组元素的赋值
                  students[i] = new Student();
                  // 数组元素是一个对象，给对象的各个属性赋值
                  students[i].number = (i + 1);
                  students[i].state = (int) (Math.random() * 6 + 1);//
[1,6]
```

```
                students[i].score = (int) (Math.random() * 101);// [0,
100]
            }

            // 问题一：打印出3年级(state 值为3）的学生信息。
            for (int i = 0; i < students.length; i++) {

                if (students[i].state == 3) {
//                  System.out.println(
//                              "number:" + students[i].number + ",
state:" + students[i].state + ",score:" + students[i].score);
                    students[i].info();

                }

            }
            System.out.println("****************************");
            // 问题二：使用冒泡排序按学生成绩排序，并遍历所有学生信息
            // 排序前
            for (int i = 0; i < students.length; i++) {
//              System.out.println(
//                          "number:" + students[i].number + ",state:
" +
//                                      students[i].state + ",score:"
 + students[i].score);

                students[i].info();
            }

            System.out.println();
            // 排序:
            for (int i = 0; i < students.length - 1; i++) {
                for (int j = 0; j < students.length - 1 - i; j++) {
                    if (students[j].score > students[j + 1].score)
{
                        Student temp = students[j];
                        students[j] = students[j + 1];
                        students[j + 1] = temp;
                    }
                }
            }

            // 排序后:
            for (int i = 0; i < students.length; i++) {
//              System.out.println(
//                          "number:" + students[i].number + ",state:
" +
//                                      students[i].state + ",score:"
 + students[i].score);
```

```
                students[i].info();
            }

        }

}
```

内存解析：

2、注意点

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。

3、练习

（1）定义矩形类 Rectangle，包含长、宽属性，area()返回矩形面积的方法，perimeter()返回矩形周长的方法，String getInfo()返回圆对象的详细信息（如：长、宽、面积、周长等数据）的方法

（2）在测试类中创建长度为 3 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 {
```

```java
    double length;
    double width;

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

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

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

```java
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());
        }
    }
}
```

内存解析：

6. 再谈方法

6.1 方法的重载（overload）

6.1.1 概念及特点

- **方法重载**：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。
  - 参数列表不同，意味着参数个数或参数类型的不同
- **重载的特点**：与修饰符、返回值类型无关，只看参数列表，且参数列表必须不同。(参数个数或参数类型)。调用时，根据方法参数列表的不同来区别。
- **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

6.1.2 示例

举例 1：

```
//System.out.println()方法就是典型的重载方法，其内部的声明形式如下：
public class PrintStream {
    public void println(byte x)
      public void println(short x)
      public void println(int x)
      public void println(long x)
      public void println(float x)
      public void println(double x)
      public void println(char x)
```

```
      public void println(double x)
      public void println()

}

public class HelloWorld{
    public static void main(String[] args) {
        System.out.println(3);
        System.out.println(1.2f);
        System.out.println("hello!");
    }
}
```

举例 2：

```
//返回两个整数的和
public int add(int x,int y){
    return x+y;
}

//返回三个整数的和
public int add(int x,int y,int z){
    return x+y+z;
}
//返回两个小数的和
public double add(double x,double y){
    return x+y;
}
```

举例 3：方法的重载和返回值类型无关

```
public class MathTools {
    //以下方法不是重载，会报错
    public int getOneToHundred(){
      return (int)(Math.random()*100);
    }

    public double getOneToHundred(){
      return Math.random()*100;
    }
}
```

6.1.3 练习

练习 1：判 断与 void show(int a,char b,double c){}构成重载的有：

```
a)void show(int x,char y,double z){}     //

b)int show(int a,double c,char b){}      //
```

```
c) void show(int a*,*double c,char b){}  //

d) boolean show(int c,char b){}          //

e) void show(double c){}                 //

f) double show(int x,char y,double z){}  //

g) void shows(){double c}                //
```

练习 2：编写程序，定义三个重载方法并调用。

- 方法名为 mOL。
- 三个方法分别接收一个 int 参数、两个 int 参数、一个字符串参数。分别执行平方运算并输出结果，相乘并输出结果，输出字符串信息。
- 在主类的 main ()方法中分别用参数区别调用三个方法。

练习 3：定义三个重载方法 max()，第一个方法求两个 int 值中的最大值，第二个方法求两个 double 值中的最大值，第三个方法求三个 double 值中的最大值，并分别调用三个方法。

6.2 可变个数的形参

在 **JDK 5.0** 中提供了 **Varargs(**variable number of arguments**)**机制。即当定义一个方法时，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个数的形参。

格式：

方法名(参数的类型名 ...参数名)

举例：

```
//JDK 5.0 以前：采用数组形参来定义方法，传入多个同一类型变量
public static void test(int a ,String[] books);

//JDK5.0：采用可变个数形参来定义方法，传入多个同一类型变量
public static void test(int a ,String...books);
```

特点：

1. 可变参数：方法参数部分指定类型的参数个数是可变多个：0 个，1 个或多个
2. 可变个数形参的方法与同名的方法之间，彼此构成重载
3. 可变参数方法的使用与方法参数部分使用数组是一致的，二者不能同时声明

4. 方法的参数部分有可变形参，需要放在形参声明的最后

5. 在一个方法的形参位置，最多只能声明一个可变个数形参

案例分析：

案例 1：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""

```
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"));
    }
}
```

案例 2：求 n 个整数的和

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

    public int sum(int... nums){
        int sum = 0;
        for (int i = 0; i < nums.length; i++) {
```

```
            sum += nums[i];
        }
        return sum;
    }
}

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
    }
}
```

案例 3：如下的方法彼此构成重载

```
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 个整数的最大值
    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;
    }*/
}
```

6.3 方法的参数传递机制

6.3.1 形参和实参

若方法含有参数：

- 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。

6.3.2 参数传递机制：值传递

Java 里方法的参数传递方式只有一种：值传递。 即将实际参数值的副本（复制品）传入方法内，而参数本身不受影响。

- 形参是基本数据类型：将实参基本数据类型变量的“数据值”传递给形参
- 形参是引用数据类型：将实参引用数据类型变量的“地址值”传递给形参

6.3.3 举例

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

```
public class ValueTransferTest1 {

	public static void main(String[] args) {


		int m = 10;
		int n = 20;

		System.out.println("m = " + m + ", n = " + n);
		//交换m和n的值
//		int temp = m;
//		m = n;
//		n = temp;

		ValueTransferTest1 test = new ValueTransferTest1();
		test.swap(m, n);

		System.out.println("m = " + m + ", n = " + n);
	}

	public void swap(int m,int n){
		int temp = m;
		m = n;
		n = temp;
	}

}
```

内存解析：

尚硅谷

image-20220320231755654

2、形参是引用数据类型

```
public class ValueTransferTest2 {
      public static void main(String[] args) {

            Data d1 = new Data();
            d1.m = 10;
            d1.n = 20;

            System.out.println("m = " + d1.m + ", n = " + d1.n);

            //实现 换序

            ValueTransferTest2 test = new ValueTransferTest2();
            test.swap(d1);

            System.out.println("m = " + d1.m + ", n = " + d1.n);

      }

      public void swap(Data data){
            int temp = data.m;
            data.m = data.n;
            data.n = temp;
```

```
        }
}

class Data{
        int m;
        int n;
}
```

内存解析：

6.3.4 练习

练习 1：判断如下程序输出的结果

```
public class AssignNewObject {
    public void swap(MyData my){
        my = new MyData(); //考虑堆空间此新创建的对象，和main 中的data 对象是否有关
        int temp = my.x;
        my.x = my.y;
        my.y = temp;

    }

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

        MyData data = new MyData();
        data.x = 1;
        data.y = 2;
        System.out.println("交换之前: x = " + data.x +",y = " + data.y);
```

```
//
        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);
//
    }
}

class MyData{
    int x ;
    int y;
}
```

练习 2：如下操作是否可以实现数组排序

```
public class ArrayTypeParam {

    //冒泡排序，实现数组从小到大排序
    public void sort(int[] arr){
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if(arr[j] > arr[j+1]){
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
    //打印数组的元素
    public void print(int[] arr){
        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.print(nums);

        tools.sort(nums);//对 nums 数组进行排序

        System.out.println("排序之后：");
        tools.print(nums);//输出 nums 数组的元素
```

```
    }
}
```

练习 3：通过内存结构图，写出如下程序的输出结果

```
//栈：每个方法在调用时，都会有以栈帧的方法压入栈中。栈帧中保存了当前方法中声明的变量：方法内声明的，形参
//堆：存放new出来的"东西"：对象（成员变量在对象中）、数组实体（数组元素）。
//注意：变量前如果声明有类型，那么这就是一个新的刚要定义的变量。如果变量前没有声明类型，那就说明此变量在之前已经声明过。
public class TransferTest3 {
    public static void main(String args[]) {
        TransferTest3 test = new TransferTest3();
        test.first();
    }
    public void first() {
        int i = 5;
        Value v = new Value();
        v.i = 25;
        second(v, i);
        System.out.println(v.i);
    }
    public void second(Value v, int i) {
        i = 0;
        v.i = 20;
        Value val = new Value();
        v = val;
        System.out.println(v.i + " " + i);
    }
}

class Value {
    int i = 15;
}
```

内存解析：

20220320_233145

练习 4：貌似是考查方法的参数传递

附加题：（每题 10 分）

```
1. public class Test {
    public static void main(String[] args) {
        int a=10;
        int b=10;
        method(a,b);//需要在method方法被调用之后，仅打印出a=100,b=200，请写出method方法的代码
        System.out.println("a="+a);
        System.out.println("b="+b);
    }
    //代码编写处
```

```
    //法一：
    public static void method(int a, int b) {
        // 在不改变原本题目的前提下，如何写这个函数才能在main 函数中输出a=100，b=200？
```

```
        a = a * 10;
        b = b * 20;
        System.out.println(a);
        System.out.println(b);
        System.exit(0);
    }

    //法二:
    public static void method(int a, int b) {

        PrintStream ps = new PrintStream(System.out) {
            @Override
            public void println(String x) {

                if ("a=10".equals(x)) {
                    x = "a=100";
                } else if ("b=10".equals(x)) {
                    x = "b=200";
                }
                super.println(x);
            }
        };

        System.setOut(ps);

    }
```

练习 5：将对象作为参数传递给方法

（1）定义一个 Circle 类，包含一个 double 型的 radius 属性代表圆的半径，一个 findArea()方法返回圆的面积。（2）定义一个类 PassObject，在类中定义一个方法 printAreas()，该方法的定义如下：public void printAreas(Circle c, int time)，在 printAreas 方法中打印输出 1 到 time 之间的每个整数半径值，以及对应的面积。例如，times 为 5，则输出半径 1，2，3，4，5，以及对应的圆面积。（3）在 main 方法中调用 printAreas()方法，调用完毕后输出当前半径值。程序运行结果如图所示。

6.4 递归(recursion)方法

image-20220521160246138

递归方法调用：方法自己调用自己的现象就称为递归。

递归的分类:直接递归、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。

说明：

- 递归方法包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。
- 递归一定要向已知方向递归，否则这种递归就变成了无穷递归，停不下来，类似于死循环。最终发生栈内存溢出。

举例：

举例 1：计算 1～n 的和

```
public class RecursionDemo {
      public static void main(String[] args) {
        RecursionDemo demo = new RecursionDemo();
            //计算1~num 的和，使用递归完成
            int num = 5;
            // 调用求和的方法
            int sum = demo.getSum(num);
            // 输出结果
            System.out.println(sum);

      }
      /*
        通过递归算法实现.
        参数列表:int
        返回值类型: int
      */
      public int getSum(int num) {
            /*
               num 为1 时,方法返回1,
               相当于是方法的出口,num 总有是1 的情况
            */
            if(num == 1){
                  return 1;
            }
            /*
          num 不为1 时,方法返回 num +(num-1)的累和
          递归调用getSum 方法
        */
            return num + getSum(num-1);
      }
}
```

代码执行图解：

```java
public class RecursionDemo {
    public static void main(String[] args) {
        RecursionDemo demo = new RecursionDemo();
        //计算1~n的和，使用递归完成
        int n = 5;
        int sum = demo.getSum(n);
        System.out.println(sum);

    }

    public int getSum(int n) {
        if(n == 1){
            return 1;
        }
        return n + getSum( n: n-1);
    }

}
```

举例 2：递归方法计算 n!

```java
public int multiply(int num){
      if(num == 1){
            return 1;
      }else{
            return num * multiply(num - 1);
      }
}
```

举例 3：已知有一个数列：f(0) = 1，f(1) = 4，f(n+2)=2*f(n+1) + f(n)，其中 n 是大于 0 的整数，求 f(10)的值。

```
public int f(int num){
      if(num == 0){
            return 1;
      }else if(num == 1){
            return 4;
      }else{
            return 2 * f(num - 1) + f(num - 2);
      }
}
```

举例 4：已知一个数列：f(20) = 1,f(21) = 4,f(n+2) = 2*f(n+1)+f(n)，其中 n 是大于 0 的整数，求 f(10)的值。

```
public int func(int num){
      if(num == 20){
            return 1;
      }else if(num == 21){
            return 4;
      }else{
            return func(num + 2) - 2 * func(num + 1);
      }
}
```

举例 5：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，

1,1,2,3,5,8,13,21,34,55,....

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);

```
  //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);
}

//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;
    }
```

举例 6：面试题

宋老师，我今天去百度面试，遇到一个一个双重递归调用的问题，我琢磨了一下，完全不知道为什么。打断点了，也还是没看懂为什么程序会那样走。您有空可以看一下，求指教。

```
    /**
     * 测试递归调用的次数
     */
    @Test
    public void binomial() { recursion(10); }

    private static int count = 0 ;
    public static int recursion(int k){
        count++;
        System.out.println("count1:"+count + "   k:" + k);
        if(k<=0){
            return 0 ;
        }
        return recursion(k-1) + recursion(k-2);//287
        //return recursion(k-1);//11
        //return recursion(k-1) + recursion(k-1);//2047
    }
```

1 test passed – 8ms

```
count1:274   k:3
count1:275   k:2
count1:276   k:1
count1:277   k:0
count1:278   k:-1
count1:279   k:0
count1:280   k:1
count1:281   k:0
count1:282   k:-1
count1:283   k:2
count1:284   k:1
count1:285   k:0
count1:286   k:-1
count1:287   k:0

Process finished with exit code 0
```

```
    private int count = 0;

    public int recursion(int k) {
        count++;
```

```
        System.out.println("count1:" + count + "  k:" + k);
        if (k <= 0) {
            return 0;
        }
        return recursion(k - 1) + recursion(k - 2);//287
        //return recursion(k - 1);//11
        //return recursion(k - 1) + recursion(k - 1);//2047
    }
```

剖析：

image-20220320235229941

康师傅有话说：

1. 递归调用会占用大量的系统堆栈，内存耗用多，在递归调用层次多时速度要比循环慢的多，所以在使用递归时要慎重。
2. 在要求高性能的情况下尽量避免使用递归，递归调用既花时间又耗内存。考虑使用循环迭代

7. 关键字：package、import

7.1 package(包)

package 语句作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包。(若缺省该语句，则指定为无名包)。

7.1.1 语法格式

package 顶层包名.子包名 ;

举例：pack1\pack2\PackageTest.java

```java
package pack1.pack2;     //指定类PackageTest 属于包pack1.pack2

public class PackageTest{
      public void display(){
            System.out.println("in  method display()");
      }
}
```

说明：

- 包对应于文件系统的目录，package 语句中，用 “.” 来指明包(目录)的层次；
- 包通常用小写单词标识。通常使用所在公司域名的倒置：com.atguigu.xxx。
  - 大家取包名时不要使用"java.xx"包
- 一个源文件只能有一个声明包的 package 语句

7.1.2 包的作用

- 包帮助管理大型软件系统：将功能相近的类划分到同一个包中。比如：MVC 的设计模式
- 包可以包含类和子包，划分项目层次，便于管理
- 解决类命名冲突的问题
- 控制访问权限

7.1.3 应用举例

举例 1：某航运软件系统包括：一组域对象、GUI 和 reports 子系统

举例 2：MVC 设计模式

MVC 是常用的设计模式之一，将整个程序分为三个层次：视图模型层，控制器层，与数据模型层。这种将程序输入输出、数据处理，以及数据的展示分离开来的设计模式使程序结构变的灵活而且清晰，同时也描述了程序各个对象间的通信方式，降低了程序的耦合性。

模型层 model：主要处理数据
>数据对象封装 model.bean/domain
>数据库操作类 model.dao
>数据库 model.db

控制层 controller：处理业务逻辑
>应用界面相关 controller.activity
>存放 fragment controller.fragment
>显示列表的适配器 controller.adapter
>服务相关的 controller.service
>抽取的基类 controller.base

视图层 view：显示数据
>相关工具类 view.utils
>自定义 view view.ui

7.1.4 JDK 中主要的包介绍

java.lang----包含一些 Java 语言的核心类，如 String、Math、Integer、 System 和 Thread，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，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。 java.text----包含了一些 java 格式化相关的类 java.sql----包含了 java 进行 JDBC 数据库编程的相关类/接口 java.awt----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)。 B/S C/S

7.2 import(导入)

为使用定义在不同包中的 Java 类，需用 import 语句来引入指定包层次下所需要的类或全部类。import 语句告诉编译器到哪里去寻找类。

7.2.1 语法格式

```
import 包名.类名;
```

7.2.2 应用举例

```
import pack1.pack2.Test;    //import pack1.pack2.*;表示引入pack1.pack2 包中的所有结构

public class PackTest{
      public static void main(String args[]){
            Test t = new Test();          //Test 类在pack1.pack2 包中定义
            t.display();
      }
}
```

7.2.3 注意事项

1. 在源文件中使用 import 显式的导入指定包下的类或接口
2. 声明在包的声明和类的声明之间。
3. 如果需要导入多个类或接口，那么就并列显式多个 import 语句即可
4. 举例：可以使用 java.util.*的方式，一次性导入 util 包下所有的类或接口。
5. 如果导入的类或接口是 java.lang 包下的，或者是当前包下的，则可以省略此 import 语句。
6. 如果在代码中使用不同包下的同名的类。那么就需要使用类的全类名的方式指明调用的是哪个类。
7. 如果已经导入 java.a 包下的类。那么如果需要使用 a 包的子包下的类的话，仍然需要导入。

8. import static 组合的使用：调用指定类或接口下的静态的属性或方法

8. 面向对象特征一：封装性(encapsulation)

8.1 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？
- 我要开车，...，我不需要懂离合、油门、制动等原理和维修也可以驾驶。
- 客观世界里每一个事物的内部信息都是隐藏在其内部的，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。

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：仅暴露少量的方法给外部使用，尽量方便外部调用

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，加强了代码的安全性。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。

8.2 Java 如何实现数据封装

- 实现封装就是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`。具体访问范围如下：

修饰符	本类内部	本包内	其他包的子类	其他包非子类
private	√	×	×	×
缺省	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

- 具体修饰的结构：
  - 外部类：public、缺省
  - 成员变量、成员方法、构造器、成员内部类：public、protected、缺省、private

image-20220321222327616

8.3 封装性的体现

8.3.1 成员变量/属性私有化

概述：私有化类的成员变量，提供公共的 get 和 set 方法，对外暴露获取和修改属性的功能。

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：

```java
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：

```java
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());
```

```
    }
}
```

成员变量封装的好处：

- 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。

开心一笑：

```
A man and woman are in a computer programming lecture. The man touches
the woman's breasts.

"Hey!" she says. "Those are private!"

The man says, "But we're in the same class!"
```

8.3.2 私有化方法

```
/**
 *
 * @Description 自定义的操作数组的工具类
 * @author 尚硅谷-宋红康 Email:shkstart@126.com
 * @version
 *
 */
public class ArrayUtil {

      /**
       *
       * @Description 求int 型数组的最大值
       * @author 尚硅谷-宋红康
       * @param arr
       * @return
       */
      public int max(int[] arr) {
            int maxValue = arr[0];
            for(int i = 1;i < arr.length;i++){
                  if(maxValue < arr[i]){
                        maxValue = arr[i];
                  }
            }
            return maxValue;
      }
```

```java
/**
 *
 * @Description 求int型数组的最小值
 * @author 尚硅谷-宋红康
 * @param arr
 * @return
 */
public int min(int[] arr){
      int minValue = arr[0];
      for(int i = 1;i < arr.length;i++){
            if(minValue > arr[i]){
                  minValue = arr[i];
            }
      }
      return minValue;
}

/**
 *
 * @Description 求int型数组的总和
 * @author 尚硅谷-宋红康
 * @param arr
 * @return
 */
public int sum(int[] arr) {
      int sum = 0;
      for(int i = 0;i < arr.length;i++){
            sum += arr[i];
      }
      return sum;
}

/**
 *
 * @Description 求int型数组的元素的平均值
 * @author 尚硅谷-宋红康
 * @param arr
 * @return
 */
public int avg(int[] arr) {
      int sumValue = sum(arr);
      return sumValue / arr.length;
}

// 创建一系列重载的上述方法
// public double max(double[] arr){}
// public float max(float[] arr){}
// public byte max(byte[] arr){}
```

```java
/**
 *
 * @Description 遍历数组
 * @author 尚硅谷-宋红康
 * @param arr
 */
public void print(int[] arr) {
      for(int i = 0;i < arr.length;i++){
            System.out.print(arr[i] + "  ");
      }
      System.out.println();
}

/**
 *
 * @Description 复制数组arr
 * @author 尚硅谷-宋红康
 * @param arr
 * @return
 */
public int[] copy(int[] arr) {
      int[] arr1 = new int[arr.length];
      for(int i = 0;i < arr.length;i++){
            arr1[i] = arr[i];
      }
      return arr1;
}

/**
 *
 * @Description 反转数组
 * @author 尚硅谷-宋红康
 * @param arr
 */
public void reverse(int[] arr) {
      for(int i = 0,j = arr.length - 1;i < j;i++,j--){
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
      }
}

/**
 *
 * @Description 数组的排序
 * @author 尚硅谷-宋红康
 * @param arr
```

```java
     * @param desc 指明排序的方式。 ascend:升序    descend:降序
     */
    public void sort(int[] arr,String desc) {

        if("ascend".equals(desc)){//if(desc.equals("ascend")){
            for (int i = 0; i < arr.length - 1; i++) {
                for (int j = 0; j < arr.length - 1 - i; j++) {
                    if (arr[j] > arr[j + 1]) {
//                      int temp = arr[j];
//                      arr[j] = arr[j + 1];
//                      arr[j + 1] = temp;
                        swap(arr,j,j+1);
                    }
                }
            }
        }else if ("descend".equals(desc)){
            for (int i = 0; i < arr.length - 1; i++) {
                for (int j = 0; j < arr.length - 1 - i; j++) {
                    if (arr[j] < arr[j + 1]) {
//                      int temp = arr[j];
//                      arr[j] = arr[j + 1];
//                      arr[j + 1] = temp;
                        swap(arr,j,j+1);
                    }
                }
            }
        }else{
            System.out.println("您输入的排序方式有误！");
        }
    }

    private void swap(int[] arr,int i,int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /**
     *
     * @Description 查找指定的value 值在arr 数组中出现的位置
     * @author 尚硅谷-宋红康
     * @param arr
     * @param value
     * @return 返回value 值出现的位置 或 -1: 未找到
     */
    public int getValue(int[] arr, int value) {
        //方法: 线性查找
        for(int i = 0;i < arr.length;i++){
            if(value == arr[i]){
```

```
                        return i;
                    }
                }

                return - 1;
            }
}
```

康师傅有话说：

开发中，一般成员实例变量都习惯使用 private 修饰，再提供相应的 public 权限的 get/set 方法访问。

对于 final 的实例变量，不提供 set()方法

对于 static final 的成员变量，习惯上使用 public 修饰。

8.4 练习

练习 1：

创建程序：在其中定义两个类：Person 和 PersonTest 类。定义如下：

用 setAge()设置人的合法年龄(0~130)，用 getAge()返回人的年龄。在 PersonTest 类中实例化 Person 类的对象 b，调用 setAge()和 getAge()方法，体会 Java 的封装性。

image-20220321223633916

练习 2：

自定义图书类。设定属性包括：书名 bookName，作者 author，出版社名 publisher，价格 price；方法包括：相应属性的 get/set 方法，图书信息介绍等。

9. 类的成员之三：构造器

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)，也称为构造方法。

9.1 构造器的作用

new 对象，并在 new 对象的时候为实例变量赋值。

举例：Person p = new Person(“Peter”,15);

解释：如同我们规定每个“人”一出生就必须先洗澡，我们就可以在“人”的构造器中加入完成“洗澡”的程序代码，于是每个“人”一出生就会自动完成“洗澡”，程序就不必再在每个人刚出生时一个一个地告诉他们要“洗澡”了。

8.2 构造器的语法格式

```
[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
    }
}
```

说明：

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，也不需要 void。
3. 构造器的修饰符只能是权限修饰符，不能被其他任何修饰。比如，不能被 static、final、synchronized、abstract、native 修饰，不能有 return 语句返回值。

代码如下：

```
public class Student {
    private String name;
    private int age;

    // 无参构造
```

```java
    public Student() {}

    // 有参构造
    public Student(String n,int a) {
        name = n;
        age = a;
    }

    public String getName() {
        return name;
    }
    public void setName(String n) {
        name = n;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int a) {
        age = a;
    }

    public String getInfo(){
        return "姓名: " + name +", 年龄: " + age;
    }
}

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());
    }
}
```

9.3 使用说明

1. 当我们没有显式的声明类中的构造器时，系统会默认提供一个无参的构造器并且该构造器的修饰符默认与类的修饰符相同

```java
/**
 * @author 尚硅谷-宋红康
 * @create 9:21
 */
public class Person {
    String name;
    int age;

    public void eat(){
        System.out.println("人吃饭！");
    }
}
```

查看反编译情况

```java
Decompiled .class file, bytecode version: 52.0 (Java 8)
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package com.atguigu.java;

public class Person {
    String name;
    int age;

    public Person() {
    }

    public void eat() {
        System.out.println("人吃饭！"); }
}
```

2. 当我们显式的定义类的构造器以后，系统就不再提供默认的无参的构造器了。
3. 在类中，至少会存在一个构造器。
4. 构造器是可以重载的。

9.4 练习

练习 1：编写两个类，TriAngle 和 TriAngleTest，其中 TriAngle 类中声明私有的底边长 base 和高 height，同时声明公共方法访问私有变量。此外，提供类必要的构造器。另一个类中使用这些公共方法，计算三角形的面积。

练习 2：

(1)定义 Student 类,有 4 个属性： String name; int age; String school; String major;

(2)定义 Student 类的 3 个构造器:

- 第一个构造器 Student(String n, int a)设置类的 name 和 age 属性；
- 第二个构造器 Student(String n, int a, String s)设置类的 name, age 和 school 属性；
- 第三个构造器 Student(String n, int a, String s, String m)设置类的 name, age ,school 和 major 属性；

(3)在 main 方法中分别调用不同的构造器创建的对象，并输出其属性值。

练习 3：

1、写一个名为 Account 的类模拟账户。该类的属性和方法如下图所示。该类包括的属性：账号 id，余额 balance，年利率 annualInterestRate；包含的方法：访问器方法（getter 和 setter 方法），取款方法 withdraw()，存款方法 deposit()。

Account
private int id private double balance private double annualInterestRate
public Account (int id, double balance, double annualInterestRate)

public int getId()
public double getBalance()
public double getAnnualInterestRate()
public void setId(int id)
public void setBalance(double balance)
public void setAnnualInterestRate(double annualInterestRate)
public void withdraw (double amount)//取钱
public void deposit (double amount)//存钱

提示：在提款方法 withdraw 中，需要判断用户余额是否能够满足提款数额的要求，如果不能，应给出提示。

1. 创建 Customer 类。

Customer
private String firstName private String lastName private Account account
public Customer(String f,String l)
public String getFirstName() public String getLastName() public Account getAccount() public void setAccount(Account account)

a. 声明三个私有对象属性：firstName、lastName 和 account。 b. 声明一个公有构造器，这个构造器带有两个代表对象属性的参数（f 和 l） c. 声明两个公有存取器来访问该对象属性，方法 getFirstName 和 getLastName 返回相应的属性。 d. 声明 setAccount 方法来对 account 属性赋值。 e. 声明 getAccount 方法以获取 account 属性。

3.写一个测试程序。

（1）创建一个 Customer ，名字叫 Jane Smith, 他有一个账号为 1000，余额为 2000 元，年利率为 1.23％ 的账户。（2）对 Jane Smith 操作。 存入 100 元，再取出 960 元。再取出 2000 元。 打印出 Jane Smith 的基本信息

```
成功存入 ：100.0
成功取出：960.0
余额不足，取款失败
Customer [Smith, Jane] has a account: id is 1000, annualInterestRate is
 1.23％, balance is 1140.0
```

10. 阶段性知识补充

10.1 类中属性赋值过程

1、在类的属性中，可以有哪些位置给属性赋值？

① 默认初始化

② 显式初始化

③ 构造器中初始化

④ 通过"对象.属性"或"对象.方法"的方式，给属性赋值

2、这些位置执行的先后顺序是怎样？

顺序：① - ② - ③ - ④

3、说明：

- 上述中的①、②、③在对象创建过程中，只执行一次。
- ④ 是在对象创建后执行的，可以根据需求多次执行。

10.2 JavaBean

- JavaBean 是一种 Java 语言写成的可重用组件。
  - 就好像你做了一个扳手，而这个扳手会在很多地方被拿去用，这个扳子也提供多种功能(你可以拿这个扳手扳、锤、撬等等)，而这个扳手就是一个组件。
- 所谓 JavaBean，是指符合如下标准的 Java 类：
  - 类是公共的
  - 有一个无参的公共的构造器

- 有属性，且有对应的 get、set 方法

- 用户可以使用 JavaBean 将功能、处理、值、数据库访问和其他任何可以用 Java 代码创造的对象进行打包，并且其他的开发者可以通过内部的 JSP 页面、Servlet、其他 JavaBean、applet 程序或者应用来使用这些对象。用户可以认为 JavaBean 提供了一种随时随地的复制和粘贴的功能，而不用关心任何改变。

- 《Think in Java》中提到，JavaBean 最初是为 Java GUI 的可视化编程实现的。你拖动 IDE 构建工具创建一个 GUI 组件（如多选框），其实是工具给你创建 java 类，并提供将类的属性暴露出来给你修改调整，将事件监听器暴露出来。

- 示例

```
public class JavaBean {
    private String name; // 属性一般定义为private
    private int age;
    public JavaBean() {
    }
    public int getAge() {
        return age;
    }
    public void setAge(int a) {
        age = a;
    }
    public String getName() {
        return name;
    }
    public void setName(String n) {
        name = n;
    }
}
```

10.3 UML 类图

- UML，统一建模语言。用来描述软件模型和架构的图形化语言。

- 常用的 UML 工具软件有 PowerDesinger、Rose 和 Enterprise Architect。

- UML 工具软件不仅可以绘制软件开发中所需的各种图表，还可以生成对应的源代码。

- 在软件开发中，使用 UML(Unified Modeling Language)类图可以更加直观地描述类以及类之间的关系。

  - 表示 public 类型， - 表示 private 类型，#表示 protected 类型

- 方法的写法: 方法的类型(+、-) 方法名(参数名： 参数类型)：返回值类型
- 斜体表示抽象方法或类。

image-20220321232230831

第 07 章_面向对象编程(进阶)

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 关键字：this

1.1 this 是什么？

- 在 Java 中，this 关键字比较难理解，它的作用和其词义很接近。

- 它在方法（准确的说是实例方法或非 static 的方法）内部使用，表示调用该方法的对象
- 它在构造器内部使用，表示该构造器正在初始化的对象。

- this 可以调用的结构：成员变量、方法和构造器

1.2 什么时候使用 this

1.2.1 实例方法或构造器中使用当前对象的成员

在实例方法或构造器中，如果使用当前类的成员变量或成员方法可以在其前面添加 this，增强程序的阅读性。不过，通常我们都习惯省略 this。

但是，当形参与成员变量同名时，如果在方法内或构造器内需要使用成员变量，必须添加 this 来表明该变量是类的成员变量。即：我们可以用 this 来区分成员变量和局部变量。比如：

image-20220503102947013

另外，使用 this 访问属性和方法时，如果在本类中未找到，会从父类中查找。这个在继承中会讲到。

举例 1：

```
class Person{            // 定义 Person 类
      private String name ;
      private int age ;
      public Person(String name,int age){
            this.name = name ;
            this.age = age ;
    }
    public void setName(String name){
        this.name = name;
    }
    public void setAge(int age){
        this.age = age;
    }
```

```
    public void getInfo(){
        System.out.println("姓名: " + name) ;
        this.speak();
    }
    public void speak(){
        System.out.println("年龄: " + this.age);
    }
}
```

举例 2:

```
public class Rectangle {
    int length;
    int width;

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

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

    public 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();
        }
    }

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

测试类:

```
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('%');
    }
}
```

1.2.2 同一个类中构造器互相调用

this 可以作为一个类中构造器相互调用的特殊格式。

- this()：调用本类的无参构造器
- this(实参列表)：调用本类的有参构造器

```
public class Student {
    private String name;
    private int age;

    // 无参构造
    public Student() {
//        this("",18);//调用本类有参构造器
    }

    // 有参构造
    public Student(String name) {
        this();//调用本类无参构造器
        this.name = name;
    }
    // 有参构造
    public Student(String name,int age){
        this(name);//调用本类中有一个String 参数的构造器
        this.age = age;
    }

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

```java
    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：

- 不能出现递归调用。比如，调用自身构造器。
  - 推论：如果一个类中声明了 n 个构造器，则最多有 n - 1 个构造器中使用了"this(形参列表)"
- this()和 this(实参列表)只能声明在构造器首行。
  - 推论：在类的一个构造器中，最多只能声明一个"this(形参列表)"

1.3 练习

练习 1：添加必要的构造器，综合应用构造器的重载，this 关键字。

image-20220323211320412

练习 2：

（1）按照如下的 UML 类图，创建相应的类，提供必要的结构：

image-20220323211412409

在提款方法 withdraw()中，需要判断用户余额是否能够满足提款数额的要求，如果不能，应给出提示。deposit()方法表示存款。

（2）按照如下的 UML 类图，创建相应的类，提供必要的结构

尚硅谷

image-20220323211454372

（3）按照如下的 UML 类图，创建相应的类，提供必要的结构

image-20220323211521808

- addCustomer 方法必须依照参数（姓，名）构造一个新的 Customer 对象，然后把它放到 customer 数组中。还必须把 numberOfCustomer 属性的值加 1。
- getNumOfCustomers 方法返回 numberofCustomers 属性值。
- getCustomer 方法返回与给出的 index 参数相关的客户。

（4）创建 BankTest 类，进行测试。

内存解析图：

image-20220323211653907

2. 面向对象特征二：继承(Inheritance)

2.1 继承的概述

2.1.1 生活中的继承

- 财产继承：

image-20220323215921441

- 绿化：前人栽树，后人乘凉

image-20220520171727778

“绿水青山，就是金山银山”

- 样貌：

image-20220323220246746

- 继承之外，是不是还可以"进化"：

image-20220323220537644

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。

2.1.2 Java 中的继承

角度一：从上而下

为描述和处理**个人**信息，定义类 Person：

```java
class Person {
    public String name;
    public int age;
    public Date birthDate;

    public String getInfo() {
        //...
    }
}
```

为描述和处理**学生**信息，定义类 Student：

```java
class Student {
    public String name;
    public int age;
    public Date birthDate;
    public String school;

    public String getInfo() {
        // ...
    }
}
```

通过继承，简化 Student 类的定义：

```java
class Person {
    public String name;
    public int age;
    public Date birthDate;

    public String getInfo() {
        // ...
    }
}

class Student extends Person {
    public String school;
}
```

说明：Student 类继承了父类 Person 的所有属性和方法，并增加了一个属性 school。Person 中的属性和方法，Student 都可以使用。

角度二：从下而上

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继承关系。如图所示：

再举例：

image-20220323221436571

2.1.3 继承的好处

- 继承的出现减少了代码冗余，提高了代码的复用性。
- 继承的出现，更有利于功能的扩展。

- 继承的出现让类与类之间产生了 is-a 的关系，提供了多态的前提。
  - 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。可见，父类更通用或更一般，子类更具体。

注意：不要仅为了获取其他类中某个功能而去继承

2.2 继承的语法

2.2.1 继承中的语法格式

通过 extends 关键字，可以声明一个类 B 继承另外一个类 A，定义格式如下：

```
[修饰符] class 类 A {
      ...
}

[修饰符] class 类 B extends 类 A {
      ...
}
```

2.2.2 继承中的基本概念

类 B，称为子类、派生类(derived class)、SubClass

类 A，称为父类、超类、基类(base class)、SuperClass

2.3 代码举例

1、父类

```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");
    }
}
```

2、子类

```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");
    }
}
```

3、测试类

```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;
        // 调用该猫继承来的eat()方法
        cat.eat();
        // 调用该猫的catchMouse()方法
        cat.catchMouse();
        cat.catchMouse();
        cat.catchMouse();
    }
}
```

2.4 继承性的细节说明

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。

2、子类不能直接访问父类中私有的(private)的成员变量和方法

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：

3、在 Java 中，继承的关键字用的是“extends”，即子类不是父类的子集，而是对父类的“扩展”

子类在继承父类以后，还可以定义自己特有的方法，这就可以看做是对父类功能上的扩展。

4、Java 只支持单继承，不支持多重继承

image-20220514162507692

大头儿子 与 小头爸爸

image-20220523201405061

```
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
```

5、Java 支持多层继承(继承体系)

image-20220323225441417

```
class A{}
class B extends A{}
class C extends B{}
```

说明：

- 子类和父类是一种相对的概念
- 顶层父类是 Object 类。所有的类默认继承 Object，作为父类。

6、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{}
```

2.5 练习

练习 1：定义一个学生类 Student，它继承自 Person 类

image-20220323231804928

练习 2：

(1)定义一个 ManKind 类，包括

- 成员变量 int sex 和 int salary；
- 方法 void manOrWoman()：根据 sex 的值显示“man”(sex==1)或者“woman”(sex==0)；
- 方法 void employeed()：根据 salary 的值显示“no job”(salary==0)或者“ job”(salary!=0)。

(2)定义类 Kids 继承 ManKind，并包括

- 成员变量 int yearsOld；
- 方法 printAge()打印 yearsOld 的值。

(3)定义类 KidsTest，在类的 main 方法中实例化 Kids 的对象 someKid，用该对象访问其父类的成员变量及方法。

练习 3：根据下图实现类。在 CylinderTest 类中创建 Cylinder 类的对象，设置圆柱的底面半径和高，并输出圆柱的体积。

image-20220323231942361

3. 方法的重写（override/overwrite）

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自己当前的类，该怎么办呢？子类可以对从父类中继承来的方法进行改造，我们称为方法的重写（override、overwrite）。也称为方法的重置、覆盖。

在程序执行时，子类的方法将覆盖父类的方法。

3.1 方法重写举例

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");
    }
```

```java
    public void call(){
        System.out.println("打电话");
    }
    public void showNum(){
        System.out.println("来电显示号码");
    }
}
```

```java
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
      @Override
    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");
    }
    //重写父类的通话功能的方法
    @Override
    public void call() {
        System.out.println("语音通话 或 视频通话");
    }
}
```

```java
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();
    }
}
```

@Override 使用说明：

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。

3.2 方法重写的要求

1. 子类重写的方法必须和父类被重写的方法具有相同的方法名称、参数列表。
2. 子类重写的方法的返回值类型不能大于父类被重写的方法的返回值类型。（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

1. 子类重写的方法使用的访问权限不能小于父类被重写的方法的访问权限。（public > protected > 缺省 > private）

注意：① 父类私有方法不能重写 ② 跨包的父类缺省的方法也不能重写

1. 子类方法抛出的异常不能大于父类被重写方法的异常

此外，子类与父类中同名同参数的方法必须同时声明为非 static 的(即为重写)，或者同时声明为 static 的（不是重写）。因为 static 方法是属于类的，子类无法覆盖父类的方法。

3.3 小结：方法的重载与重写

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");
    }
}
```

3.4 练习

练习 1：如果现在父类的一个方法定义成 private 访问权限，在子类中将此方法声明为 default 访问权限，那么这样还叫重写吗？ (NO)

练习 2：修改继承内容的练习 2 中定义的类 Kids，在 Kids 中重新定义 employeed()方法，覆盖父类 ManKind 中定义的 employeed()方法，输出“Kids should study and no job.”

4. 再谈封装性中的 4 种权限修饰

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

（2）外部类的权限修饰符如果缺省，跨包使用有问题

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

Father.java

```java
package com.atguigu.modifier;

public class Father {
    private int a;
    int b;
    protected int c;
    public int d;
}
```

只要package语句完全相同就是本包

关于成员（例如，成员变量或成员方法等），
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
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。

Father › c

Son.java

```java
package com.atguigu.modifier;

public class Son extends Father {
    public void method(){
        System.out.println("a = " + a);
        //私有的跨类不可以直接使用
        System.out.println("b = " + b);
        //缺省的本包子类可以直接使用
        System.out.println("c = " + c);
        //protected本包子类可以直接使用
        System.out.println("d = " + d);
        //public同一个模块任意位置可以直接使用
    }
}
```

Stepchild.java

```java
package com.atguigu.modifier;

//Stepchild：继子
public class Stepchild {
    Father stepfather = new Father();//继父

    public void method(){
        System.out.println("a = " + f.a);
        //私有的跨类不可以直接使用
        System.out.println("b = " + stepfather.b);
        //缺省的本包非子类可以直接使用
        System.out.println("c = " + stepfather.c);
        //protected本包非子类可以直接使用
        System.out.println("d = " + stepfather.d);
        //public同一个模块任意位置可以直接使用
    }
}
```

TestFamily.java

```java
package com.atguigu.modifier;
public class TestFamily {
    public static void main(String[] args) {
        Mother m = new Mother();
        Father f = new Father();
        System.out.println("a = " + f.a);
        //私有的跨类不可以直接使用
        System.out.println("b = " + f.b);
        //缺省的本包可以直接使用
        System.out.println("c = " + f.c);
        //protected本包可以直接使用
        System.out.println("d = " + f.d);
        //public同一个模块任意位置可以直接使用
    }
}
```

（2）跨包下使用：要求严格

Father.java

```java
package com.atguigu.inherited.modifier;

public class Father {
    private int a;
    int b;
    protected int c;
    public int d;
}
```

package语句不完全相同就是不同包。

private：仅限于本类
缺省：仅限于本包
protected：跨包仅限于子类中
public：同模块任意位置

IllegitimateChild.java

```java
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(){
        System.out.println("a = " + a);
        //私有的跨类不可以直接使用

        System.out.println("b = " + b);
        //缺省的跨包不可以直接使用

        System.out.println("c = " + c);
        //protected的跨包子类中可以直接使用

        System.out.println("d = " + d);
        //public的同模块任意位置可以使用
    }
}
```

是Father的子类

Nephew.java

```java
package com.atguigu.inherited.other;

import com.atguigu.inherited.modifier.Father;

//Nephew：侄子，外甥
public class Nephew {
    Father uncle = new Father();

    public void method(){
        System.out.println("a = " + uncle.a);
        //私有的跨类不可以直接使用

        System.out.println("b = " + uncle.b);
        //缺省的跨包不可以直接使用

        System.out.println("c = " + uncle.c);
        //protected的跨包非子类中不可以直接使用

        System.out.println("d = " + uncle.d);
        //public的同模块任意位置可以使用
    }
}
```

不是Father的子类

（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

Mother.java

```java
package com.atguigu.inherited.modifier;

class Mother {
    public int d;
}
```

d虽然有public修饰，
但是Mother类不是public的，
d跨包也无法使用

IllegitimateChild2.java

```java
package com.atguigu.inherited.other;

import com.atguigu.inherited.modifier.Mother;

public class IllegitimateChild2 extends Mother{
    public void method(){
        System.out.println("d = " + d);
    }
}
```

5. 关键字：super

5.1 super 的理解

在 Java 类中使用 super 来调用父类中的指定操作：

- super 可用于访问父类中定义的属性
- super 可用于调用父类中定义的成员方法
- super 可用于在子类构造器中调用父类的构造器

注意：

- 尤其当子父类出现同名成员时，可以用 super 表明调用的是父类中的成员
- super 的追溯不仅限于直接父类
- super 和 this 的用法相像，this 代表本类对象的引用，super 代表父类的内存空间的标识

5.2 super 的使用场景

5.2.1 子类中调用父类被重写的方法

- 如果子类没有重写父类的方法，只要权限修饰符允许，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

举例：

```
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
    }
}
```

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

5.2.2 子类中调用父类中同名的成员变量

- 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.实例变量访问

举例：

```
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）

再举例：

```
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()
    }
}
```

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：

5.2.3 子类构造器中调用父类构造器

- 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。
- 当父类中没有空参数的构造器时，子类的构造器必须通过 this(参数列表)或者 super(参数列表)语句指定调用本类或者父类中相应的构造器。同时，只能"二选一"
- super()和 super(实参列表)都只能出现在子类构造器的首行
- 如果子类构造器中既未显式调用父类或本类的构造器，且父类中又没有无参的构造器，则**编译出错**

```java
public class Person {
    private String name;
    private int age;
    private Date birthDate;

    public Person(String name, int age, Date d) {
        this.name = name;
        this.age = age;
        this.birthDate = d;
    }
    public Person(String name, int age) {
        this(name, age, null);
    }
    public Person(String name, Date d) {
        this(name, 30, d);
    }
    public Person(String name) {
        this(name, 30);
    }
}

public class Student extends Person {
    private String school;
    public Student(String name, int age, String s) {
        super(name, age);
        school = s;
    }
    public Student(String name, String s) {
        super(name);
        school = s;
    }
    // 编译出错: no super(),系统将调用父类无参数的构造器。
    public Student(String s) {
```

```
        school = s;
    }
}
```

```
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(
//
    }*
}
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

Implicit super constructor A() is undefined for default constructor. Must define an explicit constructor

1 quick fix available:

Add constructor 'B(int)'

```
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
    }
}
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
    }
}
```

A无参构造没有定义

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
16     }
17 }
18 class Test06{
19     public sta
20         B 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 }
```

The constructor A() is undefined

3 quick fixes available:

- Add argument to match 'A(int)'
- Change constructor 'A(int)': Remove parameter 'int'
- Create constructor 'A()'

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：

```java
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);
    }
}
```

5.3 小结：this 与 super

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

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

5.4 练习

练习 1：修改方法重写的练习 2 中定义的类 Kids 中 employeed()方法，在该方法中调用父类 ManKind 的 employeed()方法，然后再输出"but Kids should study and no job."

练习 2：修改继承中的练习 3 中定义的 Cylinder 类，在 Cylinder 类中覆盖 findArea()方法，计算圆柱的表面积。考虑：findVolume 方法怎样做相应的修改？

在 CylinderTest 类中创建 Cylinder 类的对象，设置圆柱的底面半径和高，并输出圆柱的表面积和体积。

附加题：在 CylinderTest 类中创建一个 Circle 类的对象，设置圆的半径，计算输出圆的面积。体会父类和子类成员的分别调用。

练习 3：

1、写一个名为 Account 的类模拟账户。该类的属性和方法如下图所示。该类包括的属性：账号 id，余额 balance，年利率 annualInterestRate；包含的方法：访问器方法（getter 和 setter 方法），返回月利率的方法 getMonthlyInterest()，取款方法 withdraw()，存款方法 deposit()。

Account
private int id private double balance private double annualInterestRate
public Account (int id, double balance, double annualInterestRate)
public int getId() public double getBalance() public double getAnnualInterestRate() public void setId(int id) public void setBalance(double balance) public void setAnnualInterestRate(double annualInterestRate) public double getMonthlyInterest() public void withdraw (double amount) public void deposit (double amount)

写一个用户程序测试 Account 类。在用户程序中，创建一个账号为 1122、余额为 20000、年利率 4.5%的 Account 对象。使用 withdraw 方法提款 30000 元，并打印余额。 再使用 withdraw 方法提款 2500 元，使用 deposit 方法存款 3000 元，然后打印余额和月利率。

提示：在提款方法 withdraw 中，需要判断用户余额是否能够满足提款数额的要求，如果不能，应给出提示。 运行结果如图所示：

2、创建 Account 类的一个子类 CheckAccount 代表可透支的账户，该账户中定义一个属性 overdraft 代表可透支限额。在 CheckAccount 类中重写 withdraw 方法，其算法如下：

```
如果（取款金额<账户余额），
        可直接取款
如果（取款金额>账户余额），
        计算需要透支的额度
        判断可透支额 overdraft 是否足够支付本次透支需要，如果可以
                将账户余额修改为 0，冲减可透支金额
        如果不可以
                提示用户超过可透支额的限额
```

要求：写一个用户程序测试 CheckAccount 类。在用户程序中，创建一个账号为 1122、余额为 20000、年利率 4.5%，可透支限额为 5000 元的 CheckAccount 对象。

使用 withdraw 方法提款 5000 元，并打印账户余额和可透支额。

再使用 withdraw 方法提款 18000 元，并打印账户余额和可透支额。

再使用 withdraw 方法提款 3000 元，并打印账户余额和可透支额。

提示：

（1）子类 CheckAccount 的构造方法需要将从父类继承的 3 个属性和子类自己的属性全部初始化。

（2）父类 Account 的属性 balance 被设置为 private，但在子类 CheckAccount 的 withdraw 方法中需要修改它的值，因此应修改父类的 balance 属性，定义其为 protected。

运行结果如下图所示：

```
C:\WINNT\System32\cmd.exe
C:\>javac CheckAccount.java

C:\>java CheckAccount
您的账户余额: 15000.0
您的可透支额:5000.0

您的账户余额: 0.0
您的可透支额:2000.0

超过可透支限额!

您的账户余额: 0.0
您的可透支额:2000.0

C:\>
```

6. 子类对象实例化全过程

```
Dog dog = new Dog("小花","小红");
```

image-20220324003735416

image-20220324003813163

举例：

```
class Creature {
    public Creature() {
```

```
        System.out.println("Creature 无参数的构造器");
    }
}
class Animal extends Creature {
    public Animal(String name) {
        System.out.println("Animal 带一个参数的构造器，该动物的 name 为" +
name);
    }
    public Animal(String name, int age) {
        this(name);
        System.out.println("Animal 带两个参数的构造器，其 age 为" + age);
    }
}
public class Dog extends Animal {
    public Dog() {
        super("汪汪队阿奇", 3);
        System.out.println("Dog 无参数的构造器");
    }
    public static void main(String[] args) {
        new Dog();
    }
}
```

7. 面向对象特征三：多态性

7.1 为什么需要多态性？

开发中，有时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“狗啃骨头”

（2）声明一个 Cat 类，包含 public void eat()方法，输出“猫吃鱼仔”

（3）声明一个 Person 类，功能如下：

- 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 {
```

```java
    public void eat(){
        System.out.println("狗啃骨头");
    }
}
```

```java
package com.atguigu.polymorphism.problem;

public class Cat {
    public void eat(){
        System.out.println("猫吃鱼仔");
    }
}
```

```java
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。
    */
}
```

7.2 多态的形式和体现

7.2.1 对象的多态性

多态性，是面向对象中最重要的概念，在 Java 中的体现：**对象的多态性：父类的引用指向子类的对象**

格式：（父类类型：指子类继承的父类类型，或者实现的接口类型）

父类类型 变量名 = 子类对象；

举例：

```
Person p = new Student();

Object o = new Person();//Object 类型的变量o，指向Person 类型的对象

o = new Student(); //Object 类型的变量o，指向Student 类型的对象
```

对象的多态：在 Java 中，子类的对象可以替代父类的对象使用。所以，一个引用类型变量可能指向(引用)多种不同类型的对象

7.2.2 多态的理解

Java 引用变量有两个类型：编译时类型和运行时类型。编译时类型由声明该变量时使用的类型决定，运行时类型由实际赋给该变量的对象决定。简称：**编译时，看左边；运行时，看右边。**

- 若编译时类型和运行时类型不一致，就出现了对象的多态性(Polymorphism)
- 多态情况下，“看左边”：看的是父类的引用（父类中不具备子类特有的方法）
  “看右边”：看的是子类的对象（实际运行的是子类重写父类的方法）

多态的使用前提：① 类的继承关系 ② 方法的重写

7.2.3 举例

```
package com.atguigu.polymorphism.grammar;

public class Pet {
    private String nickname; //昵称

    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 {
```

```java
    //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");
    }
}
```

```java
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("狗子" + getNickname() + "啃骨头");
    }

    //子类扩展的方法
    public void watchHouse() {
        System.out.println("看家");
    }
}
```

1、方法内局部变量的赋值体现多态

```java
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
```

```
    }
}
```

2、方法的形参声明体现多态

```
package com.atguigu.polymorphism.grammar;

public class Person{
    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 TestPerson {
    public static void main(String[] args) {
        Person person = new Person();

        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();
    }
}
```

3、方法返回值类型体现多态

```
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();
    }
}
```

7.2.4 多态的好处和弊端

好处：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。

弊端：一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，那么该变量就不能再访问子类中添加的属性和方法。

```
Student m = new Student();
m.school = "pku";      //合法,Student 类有school 成员变量
Person e = new Student();
e.school = "pku"; //非法,Person 类没有school 成员变量

// 属性是在编译时确定的，编译时e 为Person 类型，没有school 成员变量，因而编译错误。
```

康师傅有话说：

1. 使用父类做方法的形参，是多态使用最多的场合。即使增加了新的子类，方法也无需改变，符合开闭原则。
2. 父类引用做方法的形参，实参可以是任意的子类对象，可以通过不同的子类对象实现不同的行为方式。另外即使增加了新的子类，方法也无需改变，提高了扩展性，符合开闭原则。

补充：开闭原则 OCP

- 对扩展开放，对修改关闭

- 通俗解释：软件系统中的各种组件，如模块（Modules）、类（Classes）以及功能（Functions）等，应该在不修改现有代码的基础上，引入新功能

7.3 虚方法调用(Virtual Method Invocation)

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。

```
Person e = new Student();
e.getInfo();        //调用Student 类的getInfo()方法
```

子类中定义了与父类同名同参数的方法，在多态情况下，将此时父类的方法称为虚方法，父类根据赋给它的不同子类对象，动态调用属于子类的该方法。这样的方法调用在编译期是无法确定的。

举例：

前提：Person 类中定义了 welcome()方法，各个子类重写了 welcome()。

执行：多态的情况下，调用对象的 welcome()方法，实际执行的是子类重写的方法。

7.4 成员变量没有多态性

- 若子类重写了父类方法，就意味着子类里定义的方法彻底覆盖了父类里的同名方法，系统将不可能把父类里的方法转移到子类中。
- 对于实例变量则不存在这样的现象，即使子类里定义了与父类完全相同的实例变量，这个实例变量依然不可能覆盖父类中定义的实例变量

```
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;
}
```

7.5 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。

7.4.1 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 isInstanceof 关键字进行判断

7.4.2 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e
at 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Do
g，Dog 和Cat 之间是没有继承关系的
    }
}
```

7.4.3 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。如下代码格式：

```
//检验对象a 是否是数据类型A 的对象，返回值为boolean 型
对象 a instanceof 数据类型 A
```

- 说明：
  - 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。
  - 如果对象 a 属于类 A 的子类 B，a instanceof A 值也为 true。
  - 要求对象 a 所属的类与类 A 必须是子类和父类的关系，否则编译错误。

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
```

```java
        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();
            }
        }
    }
}
```

7.4.4 复习：类型转换

- 基本数据类型的 Casting：
  - 自动类型转换：小的数据类型可以自动转换成大的数据类型 如 long g=20; double d=12.0f
  - 强制类型转换：可以把大的数据类型强制转换(casting)成小的数据类型 如 float f=(float)12.0; int a=(int)1200L
- 对 Java 对象的强制类型转换称为造型
  - 从子类到父类的类型转换可以自动进行
  - 从父类到子类的类型转换必须通过造型(强制类型转换)实现
  - 无继承关系的引用类型间的转换是非法的
  - 在造型前可以使用 instanceof 操作符测试一个对象的类型

image-20220324235337563

- 举例

```
public class ConversionTest {
    public static void main(String[] args) {
        double d = 13.4;
        long l = (long) d;
        System.out.println(l);
        int in = 5;
        // boolean b = (boolean)in;
        Object obj = "Hello";
        String objStr = (String) obj;
        System.out.println(objStr);
        Object objPri = new Integer(5);
        // 所以下面代码运行时引发ClassCastException 异常
        String str = (String) objPri;
    }
}

public class Test {
    public void method(Person e) { // 设Person 类中没有getschool() 方法
       // System.out.pritnln(e.getschool()); //非法,编译时错误
        if (e instanceof Student) {
            Student me = (Student) e; // 将e 强制转换为Student 类型
            System.out.pritnln(me.getschool());
        }
    }
    public static void main(String[] args){
        Test t = new Test();
        Student m = new Student();
        t.method(m);
```

```
    }
}
```

7.6 练习

练习 1：笔试&面试

题目 1：继承成员变量和继承方法的区别

```
class Base {
    int count = 10;
    public void display() {
        System.out.println(this.count);
    }
}

class Sub extends Base {
    int count = 20;
    public void display() {
        System.out.println(this.count);
    }
}

public class FieldMethodTest {
    public static void main(String[] args){
        Sub s = new Sub();
        System.out.println(s.count);
        s.display();
        Base b = s;
        System.out.println(b == s);
        System.out.println(b.count);
        b.display();
    }
}
```

题目 2：

```
//考查多态的笔试题目:
public class InterviewTest1 {

      public static void main(String[] args) {
            Base base = new Sub();
            base.add(1, 2, 3);

//          Sub s = (Sub)base;
//          s.add(1,2,3);
      }
}

class Base {
```

```java
	public void add(int a, int... arr) {
		System.out.println("base");
	}
}

class Sub extends Base {

	public void add(int a, int[] arr) {
		System.out.println("sub_1");
	}

//	public void add(int a, int b, int c) {
//		System.out.println("sub_2");
//	}

}
```

题目 3：

```java
//getXxx()和setXxx()声明在哪个类中，内部操作的属性就是哪个类里的。
public class InterviewTest2 {
	public static void main(String[] args) {
		Father f = new Father();
		Son s = new Son();
		System.out.println(f.getInfo());//atguigu
		System.out.println(s.getInfo());//尚硅谷
		s.test();//尚硅谷  atguigu
		System.out.println("----------------");
		s.setInfo("大硅谷");
		System.out.println(f.getInfo());//atguigu
		System.out.println(s.getInfo());//大硅谷
		s.test();//大硅谷  atguigu
	}
}

class Father {
	private String info = "atguigu";

	public void setInfo(String info) {
		this.info = info;
	}

	public String getInfo() {
		return info;
	}
}

class Son extends Father {
	private String info = "尚硅谷";
```

```java
        public void setInfo(String info) {
                this.info = info;
        }

        public String getInfo() {
                return info;
        }

        public void test() {
                System.out.println(this.getInfo());
                System.out.println(super.getInfo());
        }
}
```

题目 4：多态是编译时行为还是运行时行为？

```java
//证明如下：
class Animal  {
        protected void eat() {
                System.out.println("animal eat food");
        }
}

class Cat  extends Animal  {
        protected void eat() {
                System.out.println("cat eat fish");
        }
}

class Dog  extends Animal  {
        public void eat() {
                System.out.println("Dog eat bone");
        }
}

class Sheep  extends Animal  {
        public void eat() {
                System.out.println("Sheep eat grass");
        }
}

public class InterviewTest {
        public static Animal  getInstance(int key) {
                switch (key) {
                case 0:
                        return new Cat ();
                case 1:
                        return new Dog ();
```

```
            default:
                  return new Sheep ();
            }

      }

      public static void main(String[] args) {
            int key = new Random().nextInt(3);
            System.out.println(key);

            Animal  animal = getInstance(key);
            animal.eat();
      }
}
```

练习 2：

```
class Person {
      protected String name="person";
      protected int age=50;
      public String getInfo() {
                return "Name: "+ name + "\n" +"age: "+ age;
      }
}
class Student extends Person {
      protected String school="pku";
      public String getInfo() {
                return  "Name: "+ name + "\nage: "+ age
                + "\nschool: "+ school;
      }
}
class Graduate extends Student{
      public String major="IT";
      public String getInfo()
      {
            return  "Name: "+ name + "\nage: "+ age
                + "\nschool: "+ school+"\nmajor:"+major;
      }
}
```

建立 InstanceTest 类，在类中定义方法 method(Person e); 在 method 中: (1)根据 e 的类型调用相应类的 getInfo()方法。 (2)根据 e 的类型执行： 如果 e 为 Person 类的对象，输出： “a person”; 如果 e 为 Student 类的对象，输出： “a student” “a person ” 如果 e 为 Graduate 类的对象，输出： “a graduated student” “a student” “a person”

练习 3：定义三个类，父类 GeometricObject 代表几何形状，子类 Circle 代表圆形，MyRectangle 代表矩形。定义一个测试类 GeometricTest，编写 equalsArea 方法测试两个对象的面积是否相等（注意方法的参数类型，利用动态绑定技术），编写 displayGeometricObject 方法显示对象的面积（注意方法的参数类型，利用动态绑定技术）。

image-20220325000034619

8. Object 类的使用

8.1 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。

image-20220503104750655

- 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

```
method(Object obj){…} //可以接收任何类作为其参数

Person o=new Person();
method(o);
```

- 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 Person {
      ...
}
//等价于:
public class Person extends Object {
      ...
}
```

8.2 Object 类的方法

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、equals()

= =：

- 基本类型比较值:只要两个变量的值相等，即为 true。

```
int a=5;
if(a==6){…}
```

- 引用类型比较引用(是否指向同一个对象)：只有指向同一个对象时，==才返回 true。

```
Person p1=new Person();
Person p2=new Person();
if (p1==p2){…}
```

  - 用“==”进行比较时，符号两边的数据类型必须兼容(可自动转换的基本数据类型除外)，否则编译出错

equals()：所有类都继承了 Object，也就获得了 equals()方法。还可以重写。

- 只能比较引用类型，其作用与“==”相同,比较是否指向同一个对象。
- 格式:obj1.equals(obj2)

- 特例：当用 equals()方法进行比较时，对类 File、String、Date 及包装类（Wrapper Class）来说，是比较类型及内容而不考虑引用的是否是同一个对象；
  - 原因：在这些类中重写了 Object 类的 equals()方法。
- 当自定义使用 equals()时，可以重写。用于比较两个对象的“内容”是否都相等
- 重写 equals()方法的原则
  - 对称性：如果 x.equals(y)返回是“true”，那么 y.equals(x)也应该返回是“true”。
  - 自反性：x.equals(x)必须返回是“true”。
  - 传递性：如果 x.equals(y)返回是“true”，而且 y.equals(z)返回是“true”，那么 z.equals(x)也应该返回是“true”。
  - 一致性：如果 x.equals(y)返回是“true”，只要 x 和 y 内容一直不变，不管你重复 x.equals(y)多少次，返回都是“true”。
  - 任何情况下，x.equals(null)，永远返回是“false”；

    x.equals(和 x 不同类型的对象)永远返回是“false”。
- 重写举例：

```
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 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;
		} 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;
	}

}
```

面试题：==和 equals 的区别

从我面试的反馈，85%的求职者“理直气壮”的回答错误...

- == 既可以比较基本类型也可以比较引用类型。对于基本类型就是比较值，对于引用类型就是比较内存地址

- equals 的话，它是属于 java.lang.Object 类里面的方法，如果该方法没有被重写过默认也是==;我们可以看到 String 等类的 equals 方法是被重写过的，而且 String 类在日常开发中用的比较多，久而久之，形成了 equals 是比较值的错误观点。
- 具体要看自定义类里有没有重写 Object 的 equals 方法来判断。
- 通常情况下，重写 equals 方法，会比较类中的相应属性是否都相等。

练习 1：

```
int it = 65;
float fl = 65.0f;
System.out.println("65 和 65.0f 是否相等？" + (it == fl)); //

char ch1 = 'A'; char ch2 = 12;
System.out.println("65 和'A'是否相等？ " + (it == ch1));//
System.out.println("12 和 ch2 是否相等？ " + (12 == ch2));//

String str1 = new String("hello");
String str2 = new String("hello");
System.out.println("str1 和 str2 是否相等？"+ (str1 == str2));//

System.out.println("str1 是否 equals str2？ "+(str1.equals(str2)));//

System.out.println("hello" == new java.util.Date()); //
```

练习 2：

编写 Order 类，有 int 型的 orderId，String 型的 orderName，相应的 getter()和 setter()方法，两个参数的构造器，重写父类的 equals()方法：public boolean equals(Object obj)，并判断测试类中创建的两个对象是否相等。

练习 3：

请根据以下代码自行定义能满足需要的 MyDate 类,在 MyDate 类中覆盖 equals 方法，使其判断当两个 MyDate 类型对象的年月日都相同时，结果为 true，否则为 false。 public boolean equals(Object o)

```
public class EqualsTest {
    public static void main(String[] args) {
        MyDate m1 = new MyDate(14, 3, 1976);
        MyDate m2 = new MyDate(14, 3, 1976);
        if (m1 == m2) {
            System.out.println("m1==m2");
        } else {
            System.out.println("m1!=m2"); // m1 != m2
        }
```

```
        if (m1.equals(m2)) {
            System.out.println("m1 is equal to m2");// m1 is equal to m
2
        } else {
            System.out.println("m1 is not equal to m2");
        }
    }
}
```

2、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 在进行 String 与其它类型数据的连接操作时，自动调用 toString()方法

```
Date now=new Date();
System.out.println(“now=”+now);  //相当于
System.out.println(“now=”+now.toString());
```

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()。

④ 可以根据需要在用户自定义类型中重写 toString()方法 如 String 类重写了 toString()方法，返回字符串的值。

```
s1="hello";
System.out.println(s1);//相当于System.out.println(s1.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 + '}
';
    }
}
```

练习：定义两个类，父类 GeometricObject 代表几何形状，子类 Circle 代表圆形。

image-20220325002959156

image-20220325002932102

3、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
}
```

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、clone()

```
//Object 类的clone()的使用
public class CloneTest {
      public static void main(String[] args) {
            Animal a1 = new Animal("花花");
            try {
                  Animal a2 = (Animal) a1.clone();
                  System.out.println("原始对象: " + a1);
                  a2.setName("毛毛");
                  System.out.println("clone 之后的对象: " + a2);
            } catch (CloneNotSupportedException e) {
                  e.printStackTrace();
            }
      }
}

class Animal implements Cloneable{
      private String name;

      public Animal() {
            super();
      }

      public Animal(String name) {
```

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

    public String getName() {
        return name;
    }

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

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

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // TODO Auto-generated method stub
        return super.clone();
    }

}
```

6、finalize()

- 当对象被回收时，系统自动调用该对象的 finalize() 方法。子类可以重写该方法，做一些释放资源的操作。（如果重写该方法，让一个新的引用变量重新引用该对象，则会重新激活对象）。
- 永远不要主动调用某个对象的 finalize 方法，应该交给垃圾回收机制调用。
- 什么时候被回收：当某个对象没有任何引用时，JVM 就认为这个对象是垃圾对象，就会在之后不确定的时间使用垃圾回收机制来销毁该对象，在销毁该对象前，会先调用 finalize()方法。
  - 垃圾回收发生具有不可预知性，程序无法精确控制垃圾回收机制执行。
- 垃圾回收机制的调用，是由系统来决定，也可以通过 System.gc() 或者 Runtime.getRuntime().gc()来通知系统进行垃圾回收，会有一些效果，但是系统是否进行垃圾回收依然不确定。

```
public class FinalizeTest {
    public static void main(String[] args) {
        Person p = new Person("Peter", 12);
        System.out.println(p);
        p = null;//此时对象实体就是垃圾对象，等待被回收。但时间不确定。
        System.gc();//强制性释放空间
```

```
        }
}

class Person{
        private String name;
        private int age;

        public Person(String name, int age) {
                super();
                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;
        }
        //子类重写此方法，可在释放对象前进行某些操作
        @Override
        protected void finalize() throws Throwable {
                System.out.println("对象被释放--->" + this);
        }
        @Override
        public String toString() {
                return "Person [name=" + name + ", age=" + age + "]";
        }

}
```

8.3 native 关键字的理解

使用 native 关键字说明这个方法是原生函数，也就是这个方法是用 C/C++等非 Java 语言实现的，并且被编译成了 DLL，由 Java 去调用。在定义一个 native 方法时，并不提供实现体。

1. 为什么要用 native 方法

Java 使用起来非常方便，然而有些层次的任务用 java 实现起来不容易，或者我们对程序的效率很在意时，问题就来了。例如：有时 Java 应用需要与 Java 外面的环境交互。这是本地方法存在的主要原因，你可以想想 java 需要与一些底层系统如操

作系统或某些硬件交换信息时的情况。本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解 Java 应用之外的繁琐的细节。

2. native 声明的方法，对于调用者，可以当做和其他 Java 方法一样使用

一个 native method 方法可以返回任何 java 类型，包括非基本类型，而且同样可以进行异常控制。

native method 的存在并不会对其他类调用这些本地方法产生任何影响，实际上调用这些方法的其他类甚至不知道它所调用的是一个本地方法。JVM 将控制调用本地方法的所有细节。

如果一个含有本地方法的类被继承，子类会继承这个本地方法并且可以用 Java 语言重写这个方法（如果需要的话）。

第 08 章_面向对象编程(高级)

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 关键字：static

1.1 类属性、类方法的设计思想

当我们编写一个类时，其实就是在描述其对象的属性和行为，而并没有产生实质上的对象，只有通过 new 关键字才会产生出对象，这时系统才会分配内存空间给对象，其方法才可以供外部调用。我们有时候希望无论是否产生了对象或无论产生了多少对象的情况下，某些特定的数据在内存空间里只有一份，例如所有的中国人都有个国家名称，每一个中国人都共享这个国家名称，不必在每一个中国人的实例对象中都单独分配一个用于代表国家名称的变量。

1、回顾类中的实例变量

```
class Circle{
      private double radius;
      public Circle(double radius){
        this.radius=radius;
      }
      public double findArea(){
        return Math.PI*radius*radius;
    }
}
```

创建两个 Circle 对象：

```
Circle c1=new Circle(2.0);    //c1.radius=2.0
Circle c2=new Circle(3.0);    //c2.radius=3.0
```

Circle 类中的变量 radius 是一个实例变量(instance variable)，它属于类的每一个对象，不能被同一个类的不同对象所共享。 上例中 c1 的 radius 独立于 c2 的 radius，存储在不同的空间。c1 中的 radius 变化不会影响 c2 的 radius，反之亦然。

如果想让一个类的所有实例共享数据，就用类变量（或类属性）！

2、类变量、类方法

类属性作为该类各个对象之间共享的变量。在设计类时，分析哪些属性不因对象的不同而改变，将这些属性设置为类属性。相应的方法设置为类方法。

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者和当前类的对象无关，这样的方法通常被声明为类方法，由于不需要创建对象就可以调用类方法，从而简化了方法的调用。

这里的类变量、类方法，只需要使用 `static` 修饰即可。所以也称为静态变量、静态方法。

1.2 static 关键字

- 使用范围：
  - 在 Java 类中，可用 static 修饰属性、方法、代码块、内部类
- 被修饰后的成员具备以下特点：
  - 随着类的加载而加载
  - 优先于对象存在
  - 修饰的成员，被所有对象所共享
  - 访问权限允许时，可不创建对象，直接被类调用

1.3 静态变量

1.3.1 语法格式

使用 static 修饰的成员变量就是静态变量（或类变量、类属性）

```
[修饰符] class 类{
        [其他修饰符] static 数据类型 变量名;
}
```

1.3.2 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）	0 或'\u0000'
	布尔（boolean）	false
	数据类型	默认值
引用类型	数组，类，接口	null

1.3.3 举例

举例 1：

```
class Chinese{
    //实例变量
    String name;
    int age;
    //类变量
    static String nation;//国籍

    public Chinese() {
    }
```

```java
    public Chinese(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Chinese{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", nation='" + nation + '\'' +
                '}';
    }
}
public class StaticTest {
    public static void main(String[] args) {
        Chinese c1 = new Chinese("康师傅",36);
        c1.nation = "中华人民共和国";

        Chinese c2 = new Chinese("老干妈",66);

        System.out.println(c1);
        System.out.println(c2);

        System.out.println(Chinese.nation);
    }
}
```

对应的内存结构：

尚硅谷

image-20220514183814514

举例 2：

```
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;

    public Employee() {
        total++;
        id = total;//这里使用total 静态变量的值为id 属性赋值
    }

    public Employee(String name) {
        this();
```

```
        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 + " ,nam
e=" + 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 e1 = new Employee("张三");
        Employee e2 = new Employee("李四");
        System.out.println(e1);//静态变量company 的默认值是null
        System.out.println(e2);//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(e1);//静态变量company 的值是尚硅谷
        System.out.println(e2);//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
        e1.company = "超级尚硅谷";

        System.out.println(e1);//静态变量company 的值是超级尚硅谷
        System.out.println(e2);//静态变量company 的值是超级尚硅谷
    }
}
```

1.3.4 内存解析

小结：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

1.4 静态方法

1.4.1 语法格式

用 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
    }
}
```

1.4.2 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）。
- 在 static 方法内部只能访问类的 static 修饰的属性或方法，不能访问类的非 static 的结构。
- 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。
- 因为不需要实例就可以访问 static 方法，因此 static 方法内部不能有 this，也不能有 super。如果有重名问题，使用“类名.”进行区别。

1.4.3 举例

```java
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");
    }
}
```

```java
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");
    }
}
```

```java
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
    }
}
```

1.5 练习

笔试题：如下程序执行会不会报错

```
/**
 * @author 尚硅谷-宋红康
 * @create 14:30
 */
public class StaticTest {
    public static void main(String[] args) {
        Demo test = null;
        test.hello();
    }
}

class Demo{
    public static void hello(){
        System.out.println("hello!");
    }
}
```

练习：

编写一个类实现银行账户的概念，包含的属性有“帐号”、“密码”、“存款余额”、“利率”、“最小余额”，定义封装这些属性的方法。账号要自动生成。

编写主类，使用银行账户类，输入、输出 3 个储户的上述信息。

考虑：哪些属性可以设计成 static 属性。

2. 单例(Singleton)设计模式

2.1 设计模式与面向对象设计原则

设计模式是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式。设计模式免去我们自己再思考和摸索。就像是经典的棋谱，不同的棋局，我们用不同的棋谱。”套路”

面向对象设计原则是面向对象设计的基石，面向对象设计质量的依据和保障，设计模式是面向对象设计原则的经典应用。

经典的设计模式共有 23 种。每个设计模式均是特定环境下特定问题的处理方法。

面向对象设计原则	创建型模式 5+1	结构型模式 7	行为型模式 11
· 单一职责原则	· 简单工厂模式	· 外观模式	· 模板方法模式
· 开闭原则	· 工厂方法模式	· 适配器模式	· 观察者模式
· 里氏替代原则	· 抽象工厂模式	· 代理模式	· 状态模式
· 依赖注入原则	· 创建者模式	· 装饰模式	· 策略模式
· 接口分离原则	· 原型模式	· 桥接模式	· 职责链模式
· 迪米特原则	· 单例模式	· 组合模式	· 命令模式
· 组合/聚合复用原则		· 享元模式	· 访问者模式
			· 调停者模式
			· 备忘录模式
			· 迭代器模式
			· 解释器模式

image-20220520174508815

简单工厂模式并不是 23 中经典模式的一种，是其中工厂方法模式的简化版

2.2 何为单例模式

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类**只能存在一个对象实例**，并且该类只提供一个取得其对象实例的方法。

2.3 实现思路

如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造器的访问权限设置为 private，这样，就不能用 new 操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。

2.4 单例模式的两种实现方式

2.4.1 饿汉式

```
class Singleton {
    // 1.私有化构造器
    private Singleton() {
    }

    // 2.内部提供一个当前类的实例
    // 4.此实例也必须静态化
    private static Singleton single = new Singleton();

    // 3.提供公共的静态的方法，返回当前类的对象
    public static Singleton getInstance() {
        return single;
```

```
    }
}
```

2.4.2 懒汉式

```
class Singleton {
    // 1.私有化构造器
    private Singleton() {
    }
    // 2.内部提供一个当前类的实例
    // 4.此实例也必须静态化
    private static Singleton single;
    // 3.提供公共的静态的方法，返回当前类的对象
    public static Singleton getInstance() {
        if(single == null) {
            single = new Singleton();
        }
        return single;
    }
}
```

2.4.3 饿汉式 vs 懒汉式

饿汉式：

- 特点：立即加载，即在使用类的时候已经将对象创建完毕。
- 优点：实现起来简单；没有多线程同步问题。
- 缺点：当类被加载的时候，会初始化 static 的实例，静态变量被创建并分配内存空间，从这以后，这个 static 的实例便一直占着这段内存，直到类被卸载时，静态变量被摧毁，并释放所占有的内存。因此在某些特定条件下会耗费内存。

懒汉式：

- 特点：延迟加载，即在调用静态方法时实例才被创建。
- 优点：实现起来比较简单；当类被加载的时候，static 的 instance 未被创建并分配内存空间，当静态方法第一次被调用时，初始化实例变量，并分配内存，因此在某些特定条件下会节约内存。
- 缺点：在多线程环境中，这种实现方法是完全错误的，线程不安全，根本不能保证单例的唯一性。
    - 说明：在多线程章节，会将懒汉式改造成线程安全的模式。

4、单例模式的优点：

由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。

举例：

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    /**
     * Returns the runtime object associated with the current Java application.
     * Most of the methods of class <code>Runtime</code> are instance
     * methods and must be invoked with respect to the current runtime object.
     *
     * @return  the <code>Runtime</code> object associated with the current
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}

    /**
```

image-20220325222541203

5、应用场景

- 网站的计数器，一般也是单例模式实现，否则难以同步。
- 应用程序的日志应用，一般都使用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只
- 能有一个实例去操作，否则内容不好追加。
- 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。
- 项目中，读取配置文件的类，一般也只有一个对象。没有必要每次使用配置文件数据，都生成一个对象去读取。
- Application 也是单例的典型应用
- Windows 的 Task Manager (任务管理器)就是很典型的单例模式
- Windows 的 Recycle Bin (回收站)也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

3. 理解 main 方法的语法

由于 Java 虚拟机需要调用类的 main()方法，所以该方法的访问权限必须是 public，又因为 Java 虚拟机在执行 main()方法时不必创建对象，所以该方法必须是 static 的，该方法接收一个 String 类型的数组参数，该数组中保存执行 Java 命令时传递给所运行的类的参数。

又因为 main() 方法是静态的，我们不能直接访问该类中的非静态成员，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静态成员，这种情况，我们在之前的例子中多次碰到。

命令行参数用法举例

```
public class CommandPara {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "] = " + args[i]);
        }
    }
}

//运行程序CommandPara.java
java CommandPara "Tom" "Jerry" "Shkstart"

//输出结果
args[0] = Tom
args[1] = Jerry
args[2] = Shkstart
```

IDEA 工具：

（1）配置运行参数

image-20211228101828718

image-20211228102022216

（2）运行程序

image-20211228102059327

笔试题：

```
//此处，Something 类的文件名叫OtherThing.java
class Something {
    public static void main(String[] something_to_do) {
        System.out.println("Do something ...");
    }
}


//上述程序是否可以正常编译、运行？
```

4. 类的成员之四：代码块

如果成员变量想要初始化的值不是一个硬编码的常量值，而是需要通过复杂的计算或读取文件、或读取运行环境信息等方式才能获取的一些值，该怎么办呢？此时，可以考虑代码块（或初始化块）。比如：

```
private static DataSource dataSource = null;

static{
      InputStream is = null;
      try {
            is = DBCPTest.class.getClassLoader().getResourceAsStream("d
bcp.properties");
            Properties pros = new Properties();
            pros.load(is);
            //调用BasicDataSourceFactory 的静态方法，获取数据源。
            dataSource = BasicDataSourceFactory.createDataSource(pros);
      } catch (Exception e) {
            e.printStackTrace();
      }finally{
```

```
            if(is != null){
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
}
```

- 代码块(或初始化块)的作用：
  - 对 Java 类或对象进行初始化
- 代码块(或初始化块)的分类：
  - 一个类中代码块若有修饰符，则只能被 static 修饰，称为静态代码块(static block)
  - 没有使用 static 修饰的，为非静态代码块。

4.1 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。

4.1.1 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
    }
}
```

4.1.2 静态代码块的特点

1.可以有输出语句。

2．可以对类的属性、类的声明进行初始化操作。

3．不可以对非静态的属性初始化。即：不可以调用非静态的属性和方法。

4．若有多个静态的代码块，那么按照从上到下的顺序依次执行。

5．静态代码块的执行要先于非静态代码块。

6．静态代码块随着类的加载而加载，且只执行一次。

```java
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;
    }
}
```

```java
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");
    }
}
```

4.2 非静态代码块

4.2.1 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
    }
}
```

4.2.2 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。

4.2.3 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。

4.2.4 非静态代码块的执行特点

1. 可以有输出语句。
2. 可以对类的属性、类的声明进行初始化操作。
3. 除了调用非静态的结构外，还可以调用静态的变量或方法。
4. 若有多个非静态的代码块，那么按照从上到下的顺序依次执行。
5. 每次创建对象的时候，都会执行一次。且先于构造器执行。

4.3 举例

（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("song","8888");
        System.out.println(u2.getInfo());
    }
```

如果不用非静态代码块，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;
    }
}
```

4.4 小结：实例变量赋值顺序

image-20220325230208941

4.5 练习

练习 1：分析加载顺序

```
class Root{
      static{
            System.out.println("Root 的静态初始化块");
      }
      {
            System.out.println("Root 的普通初始化块");
      }
      public Root(){
            System.out.println("Root 的无参数的构造器");
      }
}
class Mid extends Root{
      static{
            System.out.println("Mid 的静态初始化块");
      }
      {
```

```
            System.out.println("Mid 的普通初始化块");
        }
        public Mid(){
            System.out.println("Mid 的无参数的构造器");
        }
        public Mid(String msg){
            //通过 this 调用同一类中重载的构造器
            this();
            System.out.println("Mid 的带参数构造器，其参数值："
                + msg);
        }
}
class Leaf extends Mid{
        static{
            System.out.println("Leaf 的静态初始化块");
        }
        {
            System.out.println("Leaf 的普通初始化块");
        }
        public Leaf(){
            //通过 super 调用父类中有一个字符串参数的构造器
            super("尚硅谷");
            System.out.println("Leaf 的构造器");
        }
}
public class LeafTest{
        public static void main(String[] args){
            new Leaf();
            //new Leaf();
        }
}
```

练习 2：分析加载顺序

```
class Father {
        static {
            System.out.println("11111111111");
        }
        {
            System.out.println("22222222222");
        }

        public Father() {
            System.out.println("33333333333");

        }

}
```

```java
public class Son extends Father {
	static {
		System.out.println("44444444444");
	}
	{
		System.out.println("55555555555");
	}
	public Son() {
		System.out.println("66666666666");
	}


	public static void main(String[] args) {
		System.out.println("77777777777");
		System.out.println("************************");
		new Son();
		System.out.println("************************");

		new Son();
		System.out.println("************************");
		new Father();
	}

}
```

5. final 关键字

5.1 final 的意义

final：最终的，不可更改的

5.2 final 的使用

5.2.1 final 修饰类

表示这个类不能被继承，没有子类。提高安全性，提高程序的可读性。

例如：String 类、System 类、StringBuffer 类

```java
final class Eunuch{//太监类

}
class Son extends Eunuch{//错误

}
```

5.2.2 final 修饰方法

表示这个方法不能被子类重写。

例如：Object 类中的 getClass()

```
class Father{
      public final void method(){
            System.out.println("father");
      }
}
class Son extends Father{
      public void method(){//错误
            System.out.println("son");
      }
}
```

5.2.3 final 修饰变量

final 修饰某个变量（成员变量或局部变量），一旦赋值，它的值就不能被修改，即常量，常量名建议使用大写字母。

例如：final double MY_PI = 3.14;

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）

- 修饰成员变量

```
public final class Test {
    public static int totalNumber = 5;
    public final int ID;

    public Test() {
        ID = ++totalNumber; // 可在构造器中给final 修饰的“变量”赋值
    }
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(t.ID);
    }
}
```

- 修饰局部变量：

```
public class TestFinal {
    public static void main(String[] args){
        final int MIN_SCORE ;
        MIN_SCORE = 0;
        final int MAX_SCORE = 100;
        MAX_SCORE = 200; //非法
    }
}
```

- 错误演示：

```java
class A {
    private final String INFO = "atguigu";  //声明常量

    public void print() {
        //The final field A.INFO cannot be  assigned
        //INFO = "尚硅谷";
    }
}
```

5.3 笔试题

题 1：排错

```java
public class Something {
    public int addOne(final int x) {
        return ++x;
        // return x + 1;
    }
}
```

题 2：排错

```java
public class Something {
    public static void main(String[] args) {
        Other o = new Other();
        new Something().addOne(o);
    }
    public void addOne(final Other o) {
        // o = new Other();
        o.i++;
    }
}
class Other {
    public int i;
}
```

6. 抽象类与抽象方法(或 abstract 关键字)

6.1 由来

举例 1：

随着继承层次中一个个新子类的定义，类变得越来越具体，而父类则更一般，更通用。类的设计应该保证父类和子类能够共享特征。有时将一个父类设计得非常抽象，以至于它没有具体的实例，这样的类叫做抽象类。

尚硅谷

举例 2：

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 语法规定，包含抽象方法的类必须是**抽象类**。

6.2 语法格式

- **抽象类**：被 abstract 修饰的类。
- **抽象方法**：被 abstract 修饰没有方法体的方法。

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

```
public abstract void findArea(){}
```

代码举例：

```
public abstract class Animal {
    public abstract void eat();
}

public class Cat extends Animal {
    public void eat (){
            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**。

6.3 使用说明

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。

   抽象类是用来被继承的，抽象类的子类必须重写父类的抽象方法，并提供方法体。若没有重写全部的抽象方法，仍为抽象类。

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。

6.4 注意事项

- 不能用 abstract 修饰变量、代码块、构造器；
- 不能用 abstract 修饰私有方法、静态方法、final 的方法、final 的类。

6.5 应用举例 1

在航运公司系统中，Vehicle 类需要定义两个方法分别计算运输工具的燃料效率和行驶距离。

问题：卡车(Truck)和驳船(RiverBarge)的燃料效率和行驶距离的计算方法完全不同。Vehicle 类不能提供计算方法，但子类可以。

解决方案：Java 允许类设计者指定：超类声明一个方法但不提供实现，该方法的实现由子类提供。这样的方法称为抽象方法。有一个或更多抽象方法的类称为抽象类。

```
//Vehicle 是一个抽象类，有两个抽象方法。
public abstract class Vehicle{
      public abstract double calcFuelEfficiency();    //计算燃料效率的抽象方法
```

```
      public abstract double calcTripDistance();      //计算行驶距离的抽象方法
}
public class Truck extends Vehicle{
      public double calcFuelEfficiency( )   { //写出计算卡车的燃料效率的具体方法   }
      public double calcTripDistance( )    {  //写出计算卡车行驶距离的具体方法   }
}
public class RiverBarge extends Vehicle{
       public double calcFuelEfficiency( ) { //写出计算驳船的燃料效率的具体方法  }
       public double calcTripDistance( )  {  //写出计算驳船行驶距离的具体方法}
}
```

6.6 应用举例 2：模板方法设计模式(TemplateMethod)

抽象类体现的就是一种模板模式的设计，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展、改造，但子类总体上会保留抽象类的行为方式。

解决的问题：

- 当功能内部一部分实现是确定的，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。
- 换句话说，在软件开发中实现一个算法时，整体步骤很固定、通用，这些步骤已经在父类中写好了。但是某些部分易变，易变部分可以抽象出来，供不同子类实现。这就是一种模板模式。

类比举例：英语六级模板

Topic sentence ①——has become a common part ofpeople’s life. ②And——has always aroused thegreatest concern. ③what impresses us most is ——. ④The reasons——are varied.	信息提示 1. 空格内用概括性的词语填出最近出现的总体现象。 2. 空格内填题目要去讨论的具体现象 3. 现象的具体表现 4. 过度句，填现象或现象带来的后果，为下文分析产生的原因做铺垫。
5. Amongthe various reasons,——plays an important role. 6. Thatis to say,—— 7. Whatis more,—— 8. Forexample,——	5. 原因之一。 6. 具体说明原因一。 7. 原因二。 8. 举例说明原因二
9. When talking about——，——， 10. On the one hand—— 11. On the other hand,——	9. 空格一填作者要讨论的现象，空格二填作者的看法。 10. 支持看法的理由一或说明看法的第

image-20220503145003315

image-20220517205013997

举例 1：

```
abstract class Template {
    public final void getTime() {
        long start = System.currentTimeMillis();
        code();
        long end = System.currentTimeMillis();
        System.out.println("执行时间是: " + (end - start));
    }

    public abstract void code();
}

class SubTemplate extends Template {
    public void code() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(i);
        }
    }
}
```

举例 2：

```
package com.atguigu.java;
//抽象类的应用：模板方法的设计模式
public class TemplateMethodTest {

      public static void main(String[] args) {
            BankTemplateMethod btm = new DrawMoney();
```

```
            btm.process();

            BankTemplateMethod btm2 = new ManageMoney();
            btm2.process();
      }
}
abstract class BankTemplateMethod {
      // 具体方法
      public void takeNumber() {
            System.out.println("取号排队");
      }

      public abstract void transact(); // 办理具体的业务 //钩子方法

      public void evaluate() {
            System.out.println("反馈评分");
      }

      // 模板方法，把基本操作组合到一起，子类一般不能重写
      public final void process() {
            this.takeNumber();

            this.transact();// 像个钩子，具体执行时，挂哪个子类，就执行哪个
子类的实现代码

            this.evaluate();
      }
}

class DrawMoney extends BankTemplateMethod {
      public void transact() {
            System.out.println("我要取款！！！");
      }
}

class ManageMoney extends BankTemplateMethod {
      public void transact() {
            System.out.println("我要理财！我这里有 2000 万美元!!");
      }
}
```

模板方法设计模式是编程中经常用得到的模式。各个框架、类库中都有他的影子，比如常见的有：

- 数据库访问的封装
- Junit 单元测试

- JavaWeb 的 Servlet 中关于 doGet/doPost 方法调用
- Hibernate 中模板程序
- Spring 中 JDBCTemlate、HibernateTemplate 等

6.7 思考与练习

思考：

问题 1：为什么抽象类不可以使用 final 关键字声明？

问题 2：一个抽象类中可以定义构造器吗？

问题 3：是否可以这样理解：抽象类就是比普通类多定义了抽象方法，除了不能直接进行类的实例化操作之外，并没有任何的不同？

练习 1：

编写一个 Employee 类，声明为抽象类，包含如下三个属性：name，id，salary。提供必要的构造器和抽象方法：work()。

对于 Manager 类来说，他既是员工，还具有奖金(bonus)的属性。

请使用继承的思想，设计 CommonEmployee 类和 Manager 类，要求类中提供必要的方法进行属性访问。

练习 2：软件外包公司外派管理

有一家软件外包公司，可以外派开发人员，该公司有两个角色：普通开发人员 Developer 和项目经理 Manager。他们的关系如下图：

普通开发人员的工作内容是“开发项目”，项目经理的工作内容是“项目管理”。对外的报价是普通开发人员每天 500,元，超过 60 天每天 400 元。项目经理每天 800 元，超过 60 天每天 700 元。

有一家银行需要 1 名项目经理、2 名开发人员，现场开发 90 天，计算银行需要付给软件公司的总金额。

提示：创建数组 Employee[] emps = new Employee[3]。其中存储驻场的 3 名员工。

练习 3：

创建父类 Shape，包含绘制形状的抽象方法 draw()。

创建 Shape 的子类 Circle 和 Rectangle，重写 draw()方法，绘制圆形和矩形。

绘制多个圆形和矩形。

练习 4：

1、声明抽象父类 Person，包含抽象方法 public abstract void eat(); 2、声明子类中国人 Chinese，重写抽象方法，打印用筷子吃饭 3、声明子类美国人 American，重写抽象方法，打印用刀叉吃饭 4、声明子类印度人 Indian，重写抽象方法，打印用手抓饭 5、声明测试类 PersonTest，创建 Person 数组，存储各国人对象，并遍历数组，调用 eat()方法

练习 5：工资系统设计

编写工资系统，实现不同类型员工(多态)的按月发放工资。如果当月出现某个 Employee 对象的生日，则将该雇员的工资增加 100 元。

实验说明：

（1）定义一个 Employee 类，该类包含：

private 成员变量 name,number,birthday，其中 birthday 为 MyDate 类的对象；

abstract 方法 earnings()；

toString()方法输出对象的 name,number 和 birthday。

（2）MyDate 类包含:

private 成员变量 year,month,day ；

toDateString()方法返回日期对应的字符串：xxxx 年 xx 月 xx 日

（3）定义 SalariedEmployee 类继承 Employee 类，实现按月计算工资的员工处理。该类包括：private 成员变量 monthlySalary；

实现父类的抽象方法 earnings(),该方法返回 monthlySalary 值；toString()方法输出员工类型信息及员工的 name，number,birthday。

（4）参照 SalariedEmployee 类定义 HourlyEmployee 类，实现按小时计算工资的员工处理。该类包括：

private 成员变量 wage 和 hour；

实现父类的抽象方法 earnings(),该方法返回 wage*hour 值；

toString()方法输出员工类型信息及员工的 name，number,birthday。

（5）定义 PayrollSystem 类，创建 Employee 变量数组并初始化，该数组存放各类雇员对象的引用。利用循环结构遍历数组元素，输出各个对象的类型,name,number,birthday,以及该对象生日。当键盘输入本月月份值时，如果本月是某个 Employee 对象的生日，还要输出增加工资信息。

```
//提示:
//定义People 类型的数组People c1[]=new People[10];
//数组元素赋值
c1[0]=new People("John","0001",20);
c1[1]=new People("Bob","0002",19);
//若People 有两个子类Student 和Officer，则数组元素赋值时，可以使父类类型的
数组元素指向子类。
c1[0]=new Student("John","0001",20,85.0);
c1[1]=new Officer("Bob","0002",19,90.5);
```

练习 6：ATM 系统

使用面向对象的方式编程模拟 ATM 操作过程。

ATM 支持本行卡和他行卡等多种银行卡。

ATM 支持任意银行卡进行查询和取款操作，只有本行卡可以进行存款操作。

他行卡取款每笔收取 2 元手续费。

程序运行后，用户需要输入账号和密码，登录成功后才能操作。

用户完成某种操作后，可以继续进行其它操作，直到用户退出程序。

7. 接口(interface)

7.1 类比

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。

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 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。

7.2 概述

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：电脑都预留了可以插入 USB 设备的 USB 接口，USB 接口具备基本的数据传输的开启功能和关闭功能。你能不能用 USB 进行连接，或是否具备 USB 通信功能，就看你能否遵循 USB 接口规范

image-20220517211517846

- 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能否实现 Java 设计的 JDBC 规范

image-20220325235434103

接口的本质是契约，标准，规范，就像我们的法律一样。制定好后大家都要遵守。

7.3 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。

7.3.1 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：

```java
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");
    }
}
```

7.3.2 接口的成员说明

在 JDK8.0 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 JDK8.0 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 JDK9.0 时，接口又增加了：

（5）私有方法

除此之外，接口中没有构造器，没有初始化块，因为接口中没有成员变量需要动态初始化。

7.3.3 面试题

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 JDK8.0 之后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：因为之前的标准类库设计中，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，规范是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。

7.4 接口的使用规则

7.4.1 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（JDK8.0 才能开始使用）
        USB3.show();
        //通过“接口名.”直接使用接口的静态常量
        System.out.println(USB3.MAX_SPEED);
    }
}
```

7.4.2 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。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】
}
```

image-20220514163212312

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**

举例：

```
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");
```

```
    }
}
```

举例：

```
interface USB{          //
      public void start() ;
      public void stop() ;
}
class Computer{
      public static void show(USB usb){
            usb.start() ;
            System.out.println("=========== USB 设备工作 ========") ;
            usb.stop() ;
      }
};
class Flash implements USB{
      public void start(){    // 重写方法
            System.out.println("U 盘开始工作。") ;
      }
      public void stop(){          // 重写方法
            System.out.println("U 盘停止工作。") ;
      }
};
class Print implements USB{
      public void start(){    // 重写方法
            System.out.println("打印机开始工作。") ;
      }
      public void stop(){          // 重写方法
            System.out.println("打印机停止工作。") ;
      }
};
public class InterfaceDemo{
      public static void main(String args[]){
            Computer.show(new Flash()) ;
            Computer.show(new Print()) ;

            c.show(new USB(){
                  public void start(){
                        System.out.println("移动硬盘开始运行");
                  }
                  public void stop(){
                        System.out.println("移动硬盘停止运行");
                  }
            });
      }
};
```

7.4.3 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();
    }
}
```

7.4.4 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。**

举例：

image-20220514163311418

image-20220325235321778

定义多个接口：

```
package com.atguigu.interfacetype;

public interface A {
    void showA();
}

package com.atguigu.interfacetype;

public interface B {
    void showB();
}
```

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");
    }
}
```

测试类

```
package com.atguigu.interfacetype;

public class TestC {
    public static void main(String[] args) {
        C c = new C();
```

```
        c.showA();
        c.showB();
    }
}
```

7.4.5 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。

7.4.6 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();
    }
}
```

康师傅有话说：

接口语法本身不复杂，但是如何真正使用的好？是大学问。

7.5 冲突问题

7.5.1 默认方法冲突问题

（1）类优先原则

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：

```java
package com.atguigu.interfacetype;

public class Father {
    public void date(){//约会
        System.out.println("爸爸约吃饭");
    }
}
```

定义子类：

```java
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("跟康师傅学 Java");
    }
}
```

定义测试类：

```java
package com.atguigu.interfacetype;

public class TestSon {
    public static void main(String[] args) {
        Son s = new Son();
        s.date();
    }
}
```

（2）接口冲突（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("跟康师傅学 Java");
    }

}
```

测试类

```
package com.atguigu.interfacetype;
```

```java
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：

```java
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：

```java
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。

7.5.2 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，并且该成员变量名在子类中仍然可见。
- 当子类同时实现多个接口，而多个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,Moth
erInterface {
    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
```

```
    }
}
```

7.6 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（JDK8.0 及以上）
  - （4）公共的静态方法（JDK8.0 及以上）
  - （5）私有方法（JDK9.0 及以上）
- 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。

7.7 接口与抽象类之间的对比

No.	区别点	抽象类	接口
1	定义	可以包含抽象方法的类	主要是抽象方法和全局常量的集合
2	组成	构造方法、抽象方法、普通方法、常量、变量	常量、抽象方法、(jdk8.0:默认方法、静态方法)
3	使用	子类继承抽象类(extends)	子类实现接口(implements)
4	关系	抽象类可以实现多个接口	接口不能继承抽象类，但允许继承多个接口
5	常见设计模式	模板方法	简单工厂、工厂方法、代理模式
6	对象	都通过对象的多态性产生实例化对象	
7	局限	抽象类有单继承的局限	接口没有此局限
8	实际	作为一个模板	是作为一个标准或是表示一种能力
9	选择	如果抽象类和接口都可以使用的话，优先使用接口，因为避免单继承的局限	

在开发中，常看到一个类不是去继承一个已经实现好的类，而是要么继承抽象类，要么实现接口。

7.8 练习

笔试题：排错

```
interface A {
    int x = 0;
}
class B {
    int x = 1;
}
class C extends B implements A {
    public void pX() {
        System.out.println(x);
    }
    public static void main(String[] args) {
        new C().pX();
    }
}
```

笔试题：排错

```
interface Playable {
    void play();
}

interface Bounceable {
    void play();
}
```

```java
interface Rollable extends Playable, Bounceable {
    Ball ball = new Ball("PingPang");

}

class Ball implements Rollable {
    private String name;

    public String getName() {
        return name;
    }

    public Ball(String name) {
        this.name = name;
    }

    public void play() {
        ball = new Ball("Football");
        System.out.println(ball.getName());
    }
}
```

练习 1：

定义一个接口用来实现两个对象的比较。

```java
interface CompareObject{
      //若返回值是 0 ，代表相等；若为正数，代表当前对象大；负数代表当前对象小
      public int compareTo(Object o);
}
```

定义一个 Circle 类，声明 redius 属性，提供 getter 和 setter 方法

定义一个 ComparableCircle 类，继承 Circle 类并且实现 CompareObject 接口。在 ComparableCircle 类中给出接口中方法 compareTo 的实现体，用来比较两个圆的半径大小。

定义一个测试类 InterfaceTest，创建两个 ComparableCircle 对象，调用 compareTo 方法比较两个类的半径大小。

思考：参照上述做法定义矩形类 Rectangle 和 ComparableRectangle 类，在 ComparableRectangle 类中给出 compareTo 方法的实现，比较两个矩形的面积大小。

练习 2：交通工具案例

阿里的一个工程师，声明的属性和方法如下：

其中，有一个乘坐交通工具的方法 takingVehicle()，在此方法中调用交通工具的 run()。为了出行方便，他买了一辆捷安特自行车、一辆雅迪电动车和一辆奔驰轿车。这里涉及到的相关类及接口关系如下：

其中，电动车增加动力的方式是充电，轿车增加动力的方式是加油。在具体交通工具的 run()中调用其所在类的相关属性信息。

请编写相关代码，并测试。

提示：创建 Vehicle[]数组，保存阿里工程师的三辆交通工具，并分别在工程师的 takingVehicle()中调用。

8. 内部类（InnerClass)

8.1 概述

8.1.1 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（InnerClass）**，类 B 则称为**外部类（OuterClass）**。

8.1.2 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。

8.1.3 内部类的分类

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

8.2 成员内部类

8.2.1 概述

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{
    }
}
```

成员内部类的使用特征，概括来讲有如下两种角色：

- 成员内部类作为类的成员的角色：
  - 和外部类不同，Inner class 还可以声明为 private 或 protected；
  - 可以调用外部类的结构
  - Inner class 可以声明为 static 的，但此时就不能再使用外层类的非 static 的成员变量；
- 成员内部类作为类的角色：
  - 可以在内部定义属性、方法、构造器等结构
  - 可以声明为 abstract 类，因此可以被其它的内部类继承
  - 可以声明为 final 的
  - 编译以后生成 OuterClass$InnerClass.class 字节码文件（也适用于局部内部类）

注意点：

1. 非 static 的成员内部类中的成员不能声明为 static 的，只有在外部类或 static 的成员内部类中才可声明 static 成员。
2. 外部类访问成员内部类的成员，需要“内部类.成员”或“内部类对象.成员”的方式
3. 成员内部类可以直接使用外部类的所有成员，包括私有的数据
4. 当想要在外部类的静态成员部分使用内部类时，可以考虑内部类声明为静态的

具体来讲，内容如下。

8.2.2 静态成员内部类

静态成员内部类用 static 修饰，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
  - 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。

8.2.3 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
  - 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

8.2.4 创建成员内部类对象

- 实例化静态内部类

```
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();
```

- 实例化非静态内部类

```
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();
```

8.2.5 举例

```
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("-----------------------");
        //实例化静态内部类
        Outer.Inner inner = new Outer.Inner();
        inner.inFun();

        System.out.println("####################################");
        //实例化非静态内部类
        Outer outer = new Outer();
//        Outer.Nei nei = outer.new Nei();
```

```java
        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();
    }
}
```

8.3 局部内部类

8.3.1 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。

举例：

```java
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();
}
```

8.3.2 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哪个构造器，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");
                  }
      });
    }
}
```

举例：

```
public class ProductTest {

      public static void main(String[] args) {
            TestProduct tp = new TestProduct();
            Product p = tp.getProduct();
```

```
        System.out.println(p.getName());
        System.out.println(p.getPrice());
        // ----------------------
        tp.showProduct(p);
        // ------------------------
        tp.showProduct(new Product() {// 创建了一个匿名类的对象。
                public int getPrice() {
                        return 7899;
                }

                public String getName() {
                        return "iphone5s-土豪金";
                }

        });
}

public Product getProduct() {
        // 创建了一个局部内部类的对象
        // class MyProduct implements Product{
        //
        // @Override
        // public int getPrice() {
        // // TODO Auto-generated method stub
        // return 3899;
        // }
        //
        // @Override
        // public String getName() {
        // // TODO Auto-generated method stub
        // return "ipad air";
        // }
        //
        // }
        // return new MyProduct();
        return new Product() {// 创建了一个匿名内部类的对象。
                public int getPrice() {
                        return 7899;
                }

                public String getName() {
                        return "iphone5s-土豪金";
                }

        };
}

public void showProduct(Product p) {
        System.out.println(p.getName());
```

```
            System.out.println(p.getPrice());
        }
}

interface Product {
        public abstract int getPrice();

        public abstract String getName();
}
```

8.4 练习

练习：判断输出结果为何？

```
public class Test {
    public Test() {
        Inner s1 = new Inner();
        s1.a = 10;
        Inner s2 = new Inner();
        s2.a = 20;
        Test.Inner s3 = new Test.Inner();
        System.out.println(s3.a);
    }
    class Inner {
        public int a = 5;
    }
    public static void main(String[] args) {
        Test t = new Test();
        Inner r = t.new Inner();
        System.out.println(r.a);
    }
}
```

练习 2：

编写一个匿名内部类，它继承 Object，并在匿名内部类中，声明一个方法 public void test()打印尚硅谷。

请编写代码调用这个方法。

```
package com.atguigu.test01;

public class Test01 {
        public static void main(String[] args) {
                new Object(){
                        public void test(){
                                System.out.println("尚硅谷");
                        }
```

```
            }.test();
        }
}
```

9. 枚举

9.1 概述

- 某些类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)
  - 三原色：red(红色)、green(绿色)、blue(蓝色)
  - 支付方式：Cash（现金）、WeChatPay（微信）、Alipay(支付宝)、BankCard(银行卡)、CreditCard(信用卡)
  - 就职状态：Busy、Free、Vocation、Dimission
  - 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）
- 线程状态：创建、就绪、运行、阻塞、死亡
- 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。
- **若枚举只有一个对象, 则可以作为一种单例模式的实现方式。**
- 枚举类的实现：
  - 在 JDK5.0 之前，需要程序员自定义枚举类型。
  - 在 JDK5.0 之后，Java 支持 `enum` 关键字来快速定义枚举类型。

9.2 定义枚举类（JDK5.0 之前）

在 JDK5.0 之前如何声明枚举类呢？

- 私有化类的构造器，保证不能在类的外部创建其对象
- 在类的内部创建枚举类的实例。声明为：`public static final`，对外暴露这些常量对象

- 对象如果有实例变量，应该声明为 private final（建议，不是必须），并在构造器中初始化

示例代码：

```
class Season{
    private final String SEASONNAME;//季节的名称
    private final String SEASONDESC;//季节的描述
    private Season(String seasonName,String seasonDesc){
        this.SEASONNAME = seasonName;
        this.SEASONDESC = seasonDesc;
    }
    public static final Season SPRING = new Season("春天", "春暖花开");
    public static final Season SUMMER = new Season("夏天", "夏日炎炎");
    public static final Season AUTUMN = new Season("秋天", "秋高气爽");
    public static final Season WINTER = new Season("冬天", "白雪皑皑");

    @Override
    public String toString() {
        return "Season{" +
                "SEASONNAME='" + SEASONNAME + '\'' +
                ", SEASONDESC='" + SEASONDESC + '\'' +
                '}';
    }
}
class SeasonTest{
    public static void main(String[] args) {
        System.out.println(Season.AUTUMN);
    }
}
```

9.3 定义枚举类（JDK5.0 之后）

9.3.1 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 1：

```
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);
      }
}
```

9.3.2 enum 方式定义的要求和特点

- 使用 enum 定义的枚举类默认继承了 java.lang.Enum 类，因此不能再继承其他类。同时，说明枚举本质上还是类。
- 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。
- 列出的实例系统会自动添加 public static final 修饰。
- 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。
- JDK5.0 之后 switch，提供支持枚举类型，case 后面可以写枚举常量名，无需添加枚举类作为限定。

示例代码 2：

```
public enum SeasonEnum {
    SPRING("春天","春风又绿江南岸"),
    SUMMER("夏天","映日荷花别样红"),
    AUTUMN("秋天","秋水共长天一色"),
    WINTER("冬天","窗含西岭千秋雪");

    private final String seasonName;
    private final String seasonDesc;
    private SeasonEnum(String seasonName, String seasonDesc) {
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }
    public String getSeasonName() {
        return seasonName;
    }
```

```
    public String getSeasonDesc() {
        return seasonDesc;
    }
}
```

示例代码 3：

```
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;
        }
    }
}
```

康师傅有话说：

开发中，当需要定义一组常量时，强烈建议使用枚举类。

9.4 enum 中常用方法

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！

static 枚举类型[] values():返回枚举类型的对象数组。该方法可以很方便地遍历所有的枚举值，是一个静态方法

static 枚举类型 valueOf(String name)：可以把一个字符串转为对应的枚举类对象。要求字符串必须是枚举类对象的“名字”。如不是，会有运行时异常：IllegalArgumentException。

String name():得到当前枚举常量的名称。建议优先使用 toString()。

int ordinal():返回当前枚举常量的次序号，默认从 0 开始

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();
    }
}
```

9.5 实现接口的枚举类

- 和普通 Java 类一样，枚举类可以实现一个或多个接口
- 若每个枚举值在调用实现的接口方法呈现相同的行为方式，则只要统一实现该方法即可。
- 若需要每个枚举值在调用实现的接口方法呈现出不同的行为方式，则可以让每个枚举值分别来实现该方法

语法：

```
//1、枚举类可以像普通的类一样，实现接口，并且可以多个，但要求必须实现里面所有的抽象方法！
enum A implements 接口1，接口2{
      //抽象方法的实现
}

//2、如果枚举类的常量可以继续重写抽象方法!
enum A implements 接口1，接口2{
    常量名1(参数){
        //抽象方法的实现或重写
    },
    常量名2(参数){
        //抽象方法的实现或重写
    },
    //...
}
```

示例代码：

```
interface Info{
      void show();
}

//使用enum 关键字定义枚举类
enum Season1 implements Info{
      //1. 创建枚举类中的对象,声明在enum 枚举类的首位
      SPRING("春天","春暖花开"){
            public void show(){
                  System.out.println("春天在哪里？");
```

```
            }
        },
        SUMMER("夏天","夏日炎炎"){
            public void show(){
                System.out.println("宁静的夏天");
            }
        },
        AUTUMN("秋天","秋高气爽"){
            public void show(){
                System.out.println("秋天是用来分手的季节");
            }
        },
        WINTER("冬天","白雪皑皑"){
            public void show(){
                System.out.println("2002 年的第一场雪");
            }
        };

        //2. 声明每个对象拥有的属性:private final 修饰
        private final String SEASON_NAME;
        private final String SEASON_DESC;

        //3. 私有化类的构造器
        private Season1(String seasonName,String seasonDesc){
            this.SEASON_NAME = seasonName;
            this.SEASON_DESC = seasonDesc;
        }


        public String getSEASON_NAME() {
            return SEASON_NAME;
        }

        public String getSEASON_DESC() {
            return SEASON_DESC;
        }


    }
```

10. 注解(Annotation)

10.1 注解概述

10.1.1 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：

```
@Override

@Deprecated

@SuppressWarnings(value=”unchecked”)
```

注解 Annotation 是从 JDK5.0 开始引入。Annotation 可以像修饰符一样被使用，可用于修饰包、类、构造器、方法、成员变量、参数、局部变量的声明，这些信息被保存在 Annotation 的 “name=value” 对中。

10.1.2 注解与注释

注解也可以看做是一种注释，通过使用 Annotation, 程序员可以在不改变原有逻辑的情况下, 在源文件中嵌入一些补充信息。

注解，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于工具可以对它们进行处理的标签。

10.1.3 注解的重要性

在 JavaSE 中，注解的使用目的比较简单，例如标记过时的功能，忽略警告等。在 JavaEE/Android 中注解占据了更重要的角色，例如用来配置应用程序的任何切面，代替 JavaEE 旧版中所遗留的繁冗代码和 XML 配置等。

未来的开发模式都是基于注解的，JPA 是基于注解的，Spring2.5 以上都是基于注解的，Hibernate3.x 以后也是基于注解的，Struts2 有一部分也是基于注解的了。注解是一种趋势，一定程度上可以说：框架 = 注解 + 反射 + 设计模式。

10.2 常见的 Annotation 作用

使用 Annotation 时要在其前面增加 @ 符号, 并**把该 Annotation 当成一个修饰符使用。**用于修饰它支持的程序元素。

示例一：生成文档相关的注解

@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
@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
@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

```
package com.annotation.javadoc;
/**
 * @author 尚硅谷-宋红康
 * @version 1.0
 * @see Math.java
 */
public class JavadocTest {
      /**
       * 程序的主方法，程序的入口
       * @param args String[] 命令行参数
       */
      public static void main(String[] args) {
      }

      /**
       * 求圆面积的方法
       * @param radius double 半径值
       * @return double 圆的面积
       */
      public static double getArea(double radius){
            return Math.PI * radius * radius;
      }
}
```

示例二：在编译时进行格式检查(JDK 内置的三个基本注解)

@Override: 限定重写父类方法, 该注解只能用于方法

@Deprecated: 用于表示所修饰的元素(类, 方法等)已过时。通常是因为所修饰的结构危险或存在更好的选择

@SuppressWarnings: 抑制编译器警告

```
package com.annotation.javadoc;

public class AnnotationTest{

      public static void main(String[] args) {
            @SuppressWarnings("unused")
            int a = 10;
```

```
        }
        @Deprecated
        public void print(){
              System.out.println("过时的方法");
        }

        @Override
        public String toString() {
              return "重写的 toString 方法()";
        }
}
```

示例三：跟踪代码依赖性，实现替代配置文件功能

- Servlet3.0 提供了注解(annotation)，使得不再需要在 web.xml 文件中进行 Servlet 的部署。

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletRespons
e response) throws ServletException, IOException { }

    protected void doPost(HttpServletRequest request, HttpServletRespon
se response) throws ServletException, IOException {
        doGet(request, response);
      }
}
```

```
 <servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>com.servlet.LoginServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
  </servlet-mapping>
```

- Spring 框架中关于“事务”的管理

```
@Transactional(propagation=Propagation.REQUIRES_NEW,isolation=Isolation.
READ_COMMITTED,readOnly=false,timeout=3)
public void buyBook(String username, String isbn) {
      //1.查询书的单价
    int price = bookShopDao.findBookPriceByIsbn(isbn);
    //2. 更新库存
    bookShopDao.updateBookStock(isbn);
    //3. 更新用户的余额
```

```
    bookShopDao.updateUserAccount(username, price);
}
```

```
<!-- 配置事务属性 -->
<tx:advice transaction-manager="dataSourceTransactionManager" id="txAdv
ice">
       <tx:attributes>
       <!-- 配置每个方法使用的事务属性 -->
       <tx:method name="buyBook" propagation="REQUIRES_NEW"
       isolation="READ_COMMITTED"  read-only="false"  timeout="3" />
       </tx:attributes>
</tx:advice>
```

10.3 三个最基本的注解

10.3.1 @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。

10.3.2 @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。

10.3.3 @SuppressWarnings

- 抑制编译警告。当我们不希望看到警告信息的时候，可以使用 SuppressWarnings 注解来抑制警告信息
- 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。
- 可以指定的警告类型有
  - all，抑制所有警告
  - boxing，抑制与封装/拆装作业相关的警告
  - cast，抑制与强制转型作业相关的警告
  - dep-ann，抑制与淘汰注释相关的警告

- deprecation，抑制与淘汰的相关警告
- fallthrough，抑制与 switch 陈述式中遗漏 break 相关的警告
- finally，抑制与未传回 finally 区块相关的警告
- hiding，抑制与隐藏变数的区域变数相关的警告
- incomplete-switch，抑制与 switch 陈述式(enum case)中遗漏项目相关的警告
- javadoc，抑制与 javadoc
- nls，抑制与非 nls 字串文字相关的警告
- null，抑制与空值分析相关的警告
- rawtypes，抑制与使用 raw 类型相关的警告
- resource，抑制与使用 Closeable 类型的资源相关的警告
- restriction，抑制与使用不建议或禁止参照相关的警告
- serial，抑制与可序列化的类别遗漏 serialVersionUID 栏位相关的警告
- static-access，抑制与静态存取不正确相关的警告
- static-method，抑制与可能宣告为 static 的方法相关的警告
- super，抑制与置换方法相关但不含 super 呼叫的警告
- synthetic-access，抑制与内部类别的存取未最佳化相关的警告
- sync-override，抑制因为置换同步方法而遗漏同步化的警告
- unchecked，抑制与未检查的作业相关的警告
- unqualified-field-access，抑制与栏位存取不合格相关的警告
- unused，抑制与未用的程式码及停用的程式码相关的警告

示例代码：

```
package com.atguigu.annotation;

import java.util.ArrayList;

public class TestAnnotation {
    @SuppressWarnings("all")
```

```java
    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.methodO1();
    }
}

class Father{
    @Deprecated
    void show() {
        System.out.println("Father.show");
    }
    void methodO1() {
        System.out.println("Father Method");
    }
}

class Son extends Father{
/*    @Override
      void method01() {
            System.out.println("Son Method");
      }*/
}
```

10.4 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, PAR
AMETER, TYPE})
public @interface Deprecated {
}
```

10.5 自定义注解的使用

一个完整的注解应该包含三个部分：（1）声明（2）使用（3）读取

10.5.1 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。

```java
package com.atguigu.annotation;

import java.lang.annotation.*;

@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Table {
    String value();
}
```

```java
package com.atguigu.annotation;

import java.lang.annotation.*;

@Inherited
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Column {
    String columnName();
    String columnType();
}
```

10.5.2 使用自定义注解

```java
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;
```

```java
    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 + '\'' +
                '}';
    }
}
```

10.5.3 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。

具体的使用见《尚硅谷_宋红康_第 17 章_反射机制.md》。

10.6 JUnit 单元测试

10.6.1 测试分类

黑盒测试：不需要写代码，给输入值，看程序是否能够输出期望的值。

白盒测试：需要写代码的。关注程序具体的执行流程。

image-20220511181800694

10.6.2 JUnit 介绍

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引入 JUnit 的库，即相关的.class 文件组成的 jar 包。

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。

10.6.1 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：

第二步：在项目中添加 Libraries 库

image-20211228180938922

image-20211228181053362

第三步：选择要在哪些 module 中应用 JUnit 库

image-20211228181121005

第四步：检查是否应用成功

image-20211228181151890

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

image-20211228181316021

尚硅

image-20211228181335615

image-20211228181354471

image-20211228181405547

10.6.2 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器。
- @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");
    }
}
```

10.6.3 设置执行 JUnit 用例时支持控制台输入

1. 设置数据：

默认情况下，在单元测试方法中使用 Scanner 时，并不能实现控制台数据的输入。需要做如下设置：

在 `idea64.exe.vmoptions` 配置文件中加入下面一行设置，重启 idea 后生效。

`-Deditable.java.test.console=true`

2. 配置文件位置：

① 如果在 C 盘的用户目录的 config 下（`C:\Users\用户名\.IntelliJIdea2019.2\config`）或 `C:\Users\用户名\AppData\Roaming\JetBrains\IntelliJIdea2020.2` 有 `idea64.exe.vmoptions` 文件，那么将优先使用 C 盘用户目录下的。

② 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。

```
idea64.exe.vmoptions
-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
```

1576488062778

11. 包装类

11.1 为什么需要包装类

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），怎么办呢？例如：

```
//情况 1
equals(Object obj)

//情况 2
ArrayList list = …
list.add(Object obj)

//情况 3
add(int number)
add(double d)
add(boolean b)
```

11.2 有哪些包装类

Java 针对八种基本数据类型定义了相应的引用类型：包装类（封装类）。有了类的特点，就可以调用类中的方法，Java 才是真正的面向对象。

基本数据类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

父类:Number

image-20220329001912486

封装以后的，内存结构对比：

```
public static void main(String[] args){
      int num = 520;
      Integer obj = new Integer(520);
}
```

image-20220514163725830

11.3 自定义包装类

```
public class MyInteger {
    int value;

    public MyInteger() {
    }

    public MyInteger(int value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return String.valueOf(value);
    }
}
```

11.4 包装类与基本数据类型间的转换

11.4.1 装箱

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

基本数值---->包装对象

```
Integer obj1 = new Integer(4);//使用构造函数函数
Float f = new Float("4.56");
Long l = new Long("asdf");  //NumberFormatException

Integer obj2 = Integer.valueOf(4);//使用包装类中的valueOf 方法
```

11.4.2 拆箱

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

包装对象---->基本数值

```
Integer obj = new Integer(4);
int num1 = obj.intValue();
```

自动装箱与拆箱：

由于我们经常要做基本类型与包装类之间的转换，从 JDK5.0 （JDK 1.5）开始，基本类型与包装类的装箱、拆箱动作可以自动完成。例如：

```
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。
```

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 = 1;
Double d = 1;//错误的，1是int类型
```

11.5 基本数据类型、包装类与字符串间的转换

（1）基本数据类型转为字符串

方式 1：调用字符串重载的 valueOf()方法

```
int a = 10;
//String str = a;//错误的

String str = String.valueOf(a);
```

方式 2：更直接的方式

```
int a = 10;

String str = a + "";
```

（2）字符串转为基本数据类型

方式 1：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。

方式 2：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。

方式 3：通过包装类的构造器实现

```
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");

int i = new Integer("12");
```

其他方式小结：

11.6 包装类的其它 API

11.6.1 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

11.6.2 字符转大小写

```
Character.toUpperCase('x');

Character.toLowerCase('X');
```

11.6.3 整数转进制

```
Integer.toBinaryString(int i)

Integer.toHexString(int i)

Integer.toOctalString(int i)
```

11.6.4 比较的方法

```
Double.compare(double d1, double d2)

Integer.compare(int x, int y)
```

11.7 包装类对象的特点

11.7.1 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);//新 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);//新 new 的在堆中
```

尚硅谷

```
Integer y = new Integer(1);//另一个新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
```

11.7.2 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
```

11.7.3 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;
}
```

11.8 练习

笔试题：如下两个题目输出结果相同吗？各是什么。

```
Object o1 = true ? new Integer(1) : new Double(2.0);
System.out.println(o1);//1.0

Object o2;
if (true)
    o2 = new Integer(1);
else
    o2 = new Double(2.0);
System.out.println(o2);//1
```

面试题：

```
public void method1() {
    Integer i = new Integer(1);
    Integer j = new Integer(1);
    System.out.println(i == j);

    Integer m = 1;
    Integer n = 1;
    System.out.println(m == n);//

    Integer x = 128;
    Integer y = 128;
    System.out.println(x == y);//
}
```

练习：

利用 Vector 代替数组处理：从键盘读入学生成绩（以负数代表输入结束），找出最高分，并输出学生成绩等级。

- 提示：数组一旦创建，长度就固定不变，所以在创建数组前就需要知道它的长度。而向量类 java.util.Vector 可以根据需要动态伸缩。
- 创建 Vector 对象：Vector v=new Vector();
- 给向量添加元素：v.addElement(Object obj); //obj 必须是对象
- 取出向量中的元素：Object obj=v.elementAt(0);
  - 注意第一个元素的下标是 0，返回值是 Object 类型的。

- 计算向量的长度：v.size();
- 若与最高分相差 10 分内：A 等；20 分内：B 等；30 分内：C 等；其它：D 等

第 09 章_异常处理

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

本章专题与脉络

1. 异常概述

1.1 什么是生活的异常

男主角小明每天开车上班，正常车程 1 小时。但是，不出意外的话，可能会出现意外。

image-20220520210232149

出现意外，即为异常情况。我们会做相应的处理。如果不处理，到不了公司。处理完了，就可以正常开车去公司。

1.2 什么是程序的异常

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和逻辑错误。语法错了，编译不通过，不会产生字节码文件，根本不能运行。

  代码逻辑错误，只是没有得到想要的结果，例如：求 a 与 b 的和，你写成了 a-b

1.3 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，并处理异常的呢？

Java 中把不同的异常用不同的类表示，一旦发生某种异常，就创建该异常类型的对象，并且抛出（throw）。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 `ArrayIndexOfBoundsException`。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

```java
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：

异常的抛出机制

image-20220331112000671

1.4 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、以及异常的处理，保证代码的健壮性。

2. Java 异常体系

2.1 Error 和 Exception

Java 程序在执行过程中所发生的异常事件可分为两类：Error 和 Exception。分别对应着 `java.lang.Error` 与 `java.lang.Exception` 两个类。

Error：Java 虚拟机无法解决的严重问题。如：JVM 系统内部错误、资源耗尽等严重情况。一般不编写针对性的代码进行处理。

- 例如：StackOverflowError（栈内存溢出）和 OutOfMemoryError（堆内存溢出，简称 OOM）。

Exception: 其它因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理，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。例如：

- 空指针访问
- 试图读取不存在的文件
- 网络连接中断
- 数组角标越界

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。**

黑马程序

```java
/**
 * @author shkstart
 * @create 16:14
 */
public class ExceptionDemo {
    public static void main(String[] args) {
        //定义一个数组
        int[] arr = new int[10];

        System.out.println(arr[10]);

    }
```

```
ExceptionDemo › main()
Run: ExceptionDemo
D:\develop_tools\jdk\jdk1.8.0_131\bin\java.exe ...
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at com.atguigu.java4.ExceptionDemo.main(ExceptionDemo.java:12)
```

异常类型 原因 出现位置

image-20220511161910951

我们平常说的异常就是指 Exception。

2.2 Throwable

`java.lang.Throwable` 类是 Error 和 Exception 的父类。

- 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、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。

2.3 编译时异常和运行时异常

对于程序出现的异常，一般有两种解决方法：一是遇到错误就终止程序的运行。另一种方法是由程序员在编写程序时，就考虑到错误的检测、错误消息的提示，以及错误的处理。

捕获异常最理想的是在编译期间，但有的错误只有在运行时才会发生。比如：除数为 0，数组下标越界等。

image-20220330002449526

因此，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，并明确督促程序员提前编写处理它的代码。如果程序员没有编写对应的异常处理代码，则编译器就会直接判定编译失败，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。

  - **java.lang.RuntimeException** 类及它的子类都是运行时异常。比如：ArrayIndexOutOfBoundsException 数组下标越界异常，ClassCastException 类型转换异常。

  - 对于这类异常，可以不作处理，因为这类异常很普遍，若全处理可能会对程序的可读性和运行效率产生影响。

Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception
Illegal State Exception
IndexOut OfBounds Exception
NoSuch Element Exception
NullPointer Exception
蓝色：非受检(unchecked)异常
红色：受检(checked)异常

3. 常见的错误和异常

3.1 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
        recursion();
    }

    public void recursion(){ //递归方法
        recursion();
    }
}
```

```
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");
        }
    }
}
```

3.2 运行时异常

```
package com.atguigu.exception;

import org.junit.Test;
```

```java
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;
        //ArithmeticException
        System.out.println(a/b);
    }
}
```

3.3 编译时异常

```java
package com.atguigu.exception;

import org.junit.Test;
```

```java
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() {
        Thread.sleep(1000);//休眠1 秒  InterruptedException
    }

    @Test
    public void test07(){
        Class c = Class.forName("java.lang.String");//ClassNotFoundExce
ption
    }

    @Test
    public void test08() {
        Connection conn = DriverManager.getConnection("....");  //SQLEx
ception
    }
    @Test
    public void test09()  {
        FileInputStream fis = new FileInputStream("尚硅谷 Java 秘籍.txt");
 //FileNotFoundException
    }
    @Test
    public void test10() {
        File file = new File("尚硅谷 Java 秘籍.txt");
            FileInputStream fis = new FileInputStream(file);//FileNotFo
undException
            int b = fis.read();//IOException
            while(b != -1){
                  System.out.print((char)b);
                  b = fis.read();//IOException
            }

            fis.close();//IOException
    }
}
```

4. 异常的处理

4.1 异常处理概述

在编写程序时，经常要在可能出现错误的地方加上检测的代码，如进行 x/y 运算时，要检测分母为 0，数据为空，输入的不是数据而是字符等。过多的 if-else 分支会导致程序的代码加长、臃肿，可读性差，程序员需要花很大的精力“堵漏洞”。因此采用异常处理机制。

Java 异常处理

Java 采用的异常处理机制，是将异常处理的程序代码集中在一起，与正常的程序代码分开，使得程序简洁、优雅，并易于维护。

Java 异常处理的方式：

方式一：try-catch-finally

方式二：throws + 异常类型

尚硅谷

4.2 方式 1：捕获异常（try-catch-finally）

Java 提供了异常处理的**抓抛模型**。

- 前面提到，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。

- 如果一个方法内抛出异常，该异常对象会被抛给调用者方法中处理。如果异常没有在调用者方法中处理，它继续被抛给这个调用方法的上层方法。这个过程将一直继续下去，直到异常被处理。这一过程称为捕获(catch)异常。
- 如果一个异常回到 main()方法，并且 main()也不处理，则程序运行终止。

4.2.1 try-catch-finally 基本格式

捕获异常语法如下：

```
try{
      ......       //可能产生异常的代码
}
catch( 异常类型 1 e ){
      ......       //当产生 ExceptionName1 型异常时的处置措施
}
catch( 异常类型 2 e ){
......       //当产生 ExceptionName2 型异常时的处置措施
}
[ finally{
......       //无论是否发生异常，都无条件执行的语句
            }  ]
```

1、整体执行过程：

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。

image-20220503122722605

2、try

- 捕获异常的第一步是用 `try{…}`语句块选定捕获异常的范围，将可能出现异常的业务逻辑代码放在 try 语句块中。

3、catch (Exceptiontype e)

- 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。

- 如果明确知道产生的是何种异常，可以用该异常类作为 catch 的参数；也可以用其父类作为 catch 的参数。

  比如：可以用 ArithmeticException 类作为参数的地方，就可以用 RuntimeException 类作为参数，或者用所有异常的父类 Exception 类作为参数。但不能是与 ArithmeticException 类无关的异常，如 NullPointerException（catch 中的语句将不会执行）。

- 每个 try 语句块可以伴随一个或多个 catch 语句，用于处理可能产生的不同类型的异常对象。

- 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。

4、finally

image-20220331215517077

- 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。

- 不论在 try 代码块中是否发生了异常事件，catch 语句是否执行，catch 语句是否有异常，catch 语句中是否有 return，finally 块中的语句都会被执行。

- finally 语句和 catch 语句是任选的，但 finally 不能单独使用。

```
try{

}finally{

}
```

4.2.2 使用举例

举例 1：

```
public class IndexOutExp {
    public static void main(String[] args) {
        String friends[] = { "lisa", "bily", "kessy" };
        try {
            for (int i = 0; i < 5; i++) {
            System.out.println(friends[i]);
```

```
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("index err");
        }
        System.out.println("\nthis is the end");
    }
}
```

举例 2：

```
public class DivideZero1 {
    int x;
    public static void main(String[] args) {
        int y;
        DivideZero1 c = new DivideZero1();
        try {
            y = 3 / c.x;
        } catch (ArithmeticException e) {
            System.out.println("divide by zero error!");
        }
        System.out.println("program ends ok!");
    }
}
```

举例 3：

```
@Test
public void test1(){
      try{
            String str1 = "atguigu.com";
            str1 = null;
            System.out.println(str1.charAt(0));
      }catch(NullPointerException e){
            //异常的处理方式1
            System.out.println("不好意思，亲~出现了小问题，正在加紧解决...
");
      }catch(ClassCastException e){
            //异常的处理方式2
            System.out.println("出现了类型转换的异常");
      }catch(RuntimeException e){
            //异常的处理方式3
            System.out.println("出现了运行时异常");
      }
      //此处的代码，在异常被处理了以后，是可以正常执行的
      System.out.println("hello");
}
```

举例 4：

```java
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 (ArithmeticException 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();
        }
    }
}
```

4.2.3 catch 中获取异常信息

捕获异常的有关信息：与其它对象一样，可以访问一个异常对象的成员变量或调用它的方法。为此，Throwable 类中定义了一些查看方法：

- `public String getMessage()`：获取异常的描述信息，返回字符串
- `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。

image-20220331180736381

```java
@Test
public void test1(){
```

```
        FileInputStream fis = null;
        try{
                File file = new File("hello1.txt");
                fis = new FileInputStream(file);//FileNotFoundException
                int b = fis.read();//IOException
                while(b != -1){
                        System.out.print((char)b);
                        b = fis.read();//IOException
                }

        }catch(IOException e){
                e.printStackTrace();
        }finally{
                try {
                        if(fis != null)
                                fis.close();//IOException
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

4.2.4 finally 的再说明

体会 finally 中是一定要执行的代码的功能。

唯一的例外，使用 System.exit(0) 来终止当前正在运行的 Java 虚拟机。

开发中，对于一些系统资源，为了保证即使在出现异常的情况下仍能够关闭，需要将这些关闭操作声明在 finally 中。这些资源涉及到：数据库连接、输入流输出流、Socket 连接、Lock 锁的关闭

举例 1：从 try 回来

```
public class FinallyTest1 {
    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");
        }
```

```
    }
}
```

举例 2：从 catch 回来

```
public class FinallyTest2 {
    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");
        }
    }
}
```

举例 3：从 finally 回来

```
public class FinallyTest3 {
    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;
        }
    }
}
```

笔试题：

```
public class ExceptionTest {
    public static void main(String[] args) {
        int result = test();
        System.out.println(result); //100
```

```
    }

    public static int test(){
        int i = 100;
        try {
            return i;
        } finally {
            i++;
        }
    }
}
```

笔试题：final、finally、finalize 有什么区别？

4.2.5 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.java5;

/**
 * @author 尚硅谷-宋红康
 * @create 21:47
 */
public class JDK7Test {
    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 (ArithmeticException 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();
    }
}
```

4.2.6 练习

编写一个类 ExceptionTest，在 main 方法中使用 try、catch、finally，要求：

- 在 try 块中，编写被零除的代码。
- 在 catch 块中，捕获被零除所产生的异常，并且打印异常信息
- 在 finally 块中，打印一条语句。

4.2.7 小结

- 前面使用的异常都是 `RuntimeException` 类或是它的子类，这些类的异常的特点是：即使没有使用 try 和 catch 捕获，Java 自己也能捕获，并且编译通过 (但运行时会发生异常使得程序运行终止)。
- 如果抛出的异常是 IOException 等类型的非运行时异常，则必须捕获，否则编译错误。也就是说，我们必须处理编译时异常，将异常进行捕捉，转化为运行时异常。

4.3 方式 2：声明抛出异常类型（throws）

- 如果一个方法(中的语句执行时)可能生成某种异常，但是并不能确定如何处理这种异常，则此方法应显示地声明抛出异常，表明该方法将不对这些异常进行处理，而由该方法的调用者负责处理。
- 在方法声明中用 `throws` 语句可以声明抛出异常的列表，throws 后面的异常类型可以是方法中产生的异常类型，也可以是它的父类。

4.3.1 throws 基本格式

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。

举例：

```
public void readFile(String file)  throws FileNotFoundException {
      ...
```

```
        // 读文件的操作可能产生 FileNotFoundException 类型的异常
        FileInputStream fis = new FileInputStream(file);
    ...
}
```

4.3.2 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。

举例：

```
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");
        }
    }
}
```

4.3.3 throws 运行时异常

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，需要添加哪种 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 (ArithmeticException | InputMismatchException e) {
            e.printStackTrace();
        } finally {
            input.close();
        }
    }

    public static int divide(int a, int b)throws ArithmeticException{
        return a/b;
    }
}
```

4.3.4 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。复习：

（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

此外，对于 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;

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");
    }
}
```

5. 手动抛出异常对象：throw

Java 中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，那么针对当前代码，就会在后台自动创建一个对应异常类的实例对象并抛出。
- 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。

5.1 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。
- 可以抛出的异常必须是 Throwable 或其子类的实例。下面的语句在编译时将会产生语法错误：

  ```
  throw new String("want to throw");
  ```

5.2 使用注意点：

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 {
```

```java
    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;
    }
}
```

6. 自定义异常

6.1 为什么需要自定义异常类

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，某员工已在团队中等。

6.2 如何自定义异常类

（1）要继承一个异常类型

自定义一个编译时异常类型：自定义类继承 `java.lang.Exception`。

自定义一个运行时异常类型：自定义类继承 `java.lang.RuntimeException`。

（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）自定义异常需要提供 serialVersionUID

6.3 注意点

1. 自定义的异常只能通过 throw 抛出。
2. 自定义异常最重要的是异常类的名字和 message 属性。当异常出现时，可以根据名字判断异常类型。比如：TeamException("成员已满，无法添加");、TeamException("该员工已是某团队成员");
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。

6.4 举例

举例 1：

```
class MyException extends Exception {
    static final long serialVersionUID = 23423423435L;
    private int idnumber;

    public MyException(String message, int id) {
        super(message);
        this.idnumber = id;
    }

    public int getId() {
        return idnumber;
    }
}

public class MyExpTest {
    public void regist(int num) throws MyException {
        if (num < 0)
            throw new MyException("人数为负值，不合理", 3);
        else
            System.out.println("登记人数" + num);
    }
    public void manager() {
        try {
            regist(100);
        } catch (MyException e) {
            System.out.print("登记失败，出错种类" + e.getId());
        }
        System.out.print("本次登记操作结束");
    }
    public static void main(String args[]) {
```

```
        MyExpTest t = new MyExpTest();
        t.manager();
    }
}
```

举例 2：

```
package com.atguigu.define;
//自定义异常:
public class NotTriangleException extends Exception{
    static final long serialVersionUID = 13465653435L;

    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 NotTriangleExc
eption {
        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");
        }
```

```java
        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;
```

```java
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();
        }
    }
}
```

7. 练习

练习 1：

```java
public class ReturnExceptionDemo {
    static void methodA() {
        try {
            System.out.println("进入方法 A");
            throw new RuntimeException("制造异常");
        }finally {
            System.out.println("用 A 方法的 finally");
        }
    }

    static void methodB() {
        try {
            System.out.println("进入方法 B");
            return;
        } finally {
            System.out.println("调用 B 方法的 finally");
        }
    }
      public static void main(String[] args) {
      try {
          methodA();
```

```
            } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        methodB();
        }
}
```

练习 2：

从键盘接收学生成绩，成绩必须在 0~100 之间。

自定义成绩无效异常。

编写方法接收成绩并返回该成绩，如果输入无效，则抛出自定义异常。

练习 3：

编写应用程序 EcmDef.java，接收命令行的两个参数，要求不能输入负数，计算两数相除。 对数据类型不一致(NumberFormatException)、缺少命令行参数(ArrayIndexOutOfBoundsException、 除 0(ArithmeticException)及输入负数(EcDef 自定义的异常)进行异常处理。

提示： (1)在主类(EcmDef)中定义异常方法(ecm)完成两数相除功能。

(2)在 main()方法中使用异常处理语句进行异常处理。

(3)在程序中，自定义对应输入负数的异常类(EcDef)。

(4)运行时接受参数 java EcmDef 20 10 //args[0]="20" args[1]="10"

(5)Interger 类的 static 方法 parseInt(String s)将 s 转换成对应的 int 值。 如：int a=Interger.parseInt("314"); //a=314;

8. 小结与小悟

8.1 小结：异常处理 5 个关键字

image-20220330003738109

例如：上游排污，下游治污

8.2 感悟

小哲理：

世界上最遥远的距离，是我在 `if` 里你在 `else` 里，似乎一直相伴又永远分离；

世界上最痴心的等待，是我当 `case` 你是 `switch`，或许永远都选不上自己；

世界上最真情的相依，是你在 `try` 我在 `catch`。无论你发神马脾气，我都默默承受，静静处理。到那时，再来期待我们的 `finally`。

歌词：

image-20220511163035280

死了都要爱　　死了都要try
不淋漓尽致不痛快　　不彻底完全还得改
感情多深只有这样　　异常太多只有这样
才足够表白　　才能够交待
死了都要爱　　死了都要try
不哭到微笑不痛快　　不改到没有警告不自在
宇宙毁灭心还在　　系统Crash蓝屏还在

把每天当成是末日来相爱　　把每段代码当成薄冰来try
一分一秒都美到泪水掉下来　　一改一天都累到汗水流出来
不理会别人是看好或看坏　　不管代码风格是好或是坏
只要你勇敢跟我来　　只要愿意分析下来
爱 不用刻意安排　　Try 是自然的存在
凭感觉去亲吻相拥就会很愉快　　凭拷贝加粘贴可以开发的很快
享受现在别一开怀就怕受伤害　　忍受现在别一重构就怕受伤害
许多奇迹我们相信才会存在　　很多设计不太合理还得要改
死了都要爱　　死了都要try
不淋漓尽致不痛快　　不彻底完全还得改
感情多深只有这样才足够表白　　异常太多只有这样才能够交待
死了都要爱　　死了都要try
不哭到微笑不痛快　　不改到没有警告不自在
宇宙毁灭心还在　　系统Crash蓝屏还在

穷途末路都要爱　　Dump存在就要try
不极度浪漫不痛快　　不极度疲惫还得改
发会雪白土会掩埋　　代码被Review分支被覆盖
思念不腐坏　　Bug还是存在
到绝路都要爱　　到现场也要try
不天荒地老不痛快　　不通过验收不言败
不怕热爱变火海　　不怕乱try变祸害
爱到沸腾才精采　　Try到最终是无赖

第 10 章_多线程

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

我们之前学习的程序在没有跳转语句的前提下，都是由上至下沿着一条路径依次执行。那现在想要设计一个程序，一边游戏，一边 qq 聊天，一边听歌，怎么设计？

c3dcb61e3b

要解决上述问题，咱们得使用多进程或者多线程来解决。

1. 相关概念

1.1 程序、进程与线程

- **程序（program）**：为完成特定任务，用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。

- **进程（process）**：程序的一次执行过程，或是正在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，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。——生命周期
  - 如：运行中的 QQ，运行中的 MP3 播放器
  - 程序是静态的，进程是动态的
  - 进程作为操作系统调度和分配资源的最小单位（亦是系统运行程序的基本单位），系统在运行时会为每个进程分配不同的内存区域
- **线程（thread）**：进程可进一步细化为线程，是一个程序内部的一条执行路径。一个进程中至少有一个线程。
  - 若一个进程同一时间若并行执行多个线程，就是支持多线程的

image-20220331233204504

  - 线程作为 CPU 调度和执行的最小单位，每个线程拥有独立的运行栈和程序计数器，线程切换的开销小
  - 一个进程中的多个线程共享相同的内存单元/内存地址空间-->它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。
  - 下图中，红框的灰色区域为线程独享，红色区域为线程共享。

小结：一个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。

拓展：

1. 不同的进程之间是不共享内存的。
2. 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。
3. 进程之前切换的复杂度要远远高于线程之间的切换调度。

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

2、一个应用程序的多次运行，就是多个进程

1563267431480

3、一个进程中包含多个线程

1563270525077

1.3 单核 CPU 和多核 CPU

单核 CPU，其实是因为在一个时间单元内，也只能执行一个线程的任务。例如：虽然有多车道，但是一种假的多线程，收费站只有一个工作人员在收费，只有收了费才能通过，那么 CPU 就好比收费人员。如果有某个人不想交钱，那么收费人员可以把他“挂起”（晾着他，等他想通了，准备好了钱，再去收费）。但是因为 CPU 时间单元特别短，因此感觉不出来。

如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）

1.4 并行与并发

- **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 CPU 上同时执行。比如：多个人同时做不同的事。

image-20220513181758585

- **并发（concurrency）**：指两个或多个事件在同一个时间段内发生。指在一段时间内，有多条指令在单个 CPU 上交替执行（或在同一个时刻，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。）。比如：多个人做同一件事。

image-20220513181815978

并行与并发的区别：

- 并发是在一台处理器上“同时”处理多个任务，并行是在多台处理器上同时处理多个任务。如 Hadoop 分布式集群。
- 并发不是真正意义上的“同时进行”，只是 CPU 把一个时间段划分成几个时间片段（时间区间），然后在这几个时间区间之间来回切换，由于 CPU 处理的速度非常快，只要时间间隔处理得当，即可让用户感觉是多个应用程序同时在进行。

- 当一个 CPU 核心执行一个进程时，另一个 CPU 核心可以执行另一个进程，两个进程互不抢占 CPU 资源，可以同时进行，我们称之为并行(Parallel)。

- 并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。而并行是真正意义上的“同时执行”。从理论上讲，N 个并行处理的执行速度会是在单一处理机上执行速度的 N 倍。

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，这在单核 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。

而在多核 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。

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

1.5 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，现代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 在多个线程间切换速度远比我们的感觉要快，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。

2. 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？

2.1 概述

- Java 语言的 JVM 允许程序运行多个线程，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。
- Thread 类的特性
  - 每个线程都是通过某个特定 Thread 对象的 run()方法来完成操作的，因此把 run()方法体称为线程执行体。
  - 通过该 Thread 对象的 start()方法来启动这个线程，而非直接调用 run()

2.2 方式 1：继承 Thread 类

Java 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
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：

```java
package com.atguigu.thread;
//自定义线程类
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：

```java
package com.atguigu.thread;

public class TestMyThread {
    public static void main(String[] args) {
        //创建自定义线程对象1
        MyThread mt1 = new MyThread("子线程1");
        //开启子线程1
        mt1.start();

        //创建自定义线程对象2
        MyThread mt2 = new MyThread("子线程2");
        //开启子线程2
        mt2.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);
        }
    }
}
```

image-20220401221215860

注意：

1. 如果自己手动调用 run()方法，那么就只是普通方法，没有启动多线程模式。
2. run()方法由 JVM 调用，什么时候调用，执行的过程控制都有操作系统的 CPU 调度决定。
3. 想要启动多线程，必须调用 start 方法。
4. 一个线程对象只能调用一次 start()方法启动，如果重复调用了，则将抛出以上的异常“`IllegalThreadStateException`”。

2.3 方式 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 参数来创建 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，启动线程。调用 Runnable 接口实现类的 run 方法。

代码如下：

```
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。所有的分线程要执行的代码都在 run 方法里面。

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。

说明：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。

image-20220401222212377

2.4 变形写法

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();
```

2.5 对比两种方式

联系

Thread 类实际上也是实现了 Runnable 接口的类。即：

```
public class Thread extends Object implements Runnable
```

区别

- 继承 Thread：线程代码存放 Thread 子类 run 方法中。
- 实现 Runnable：线程代码存在接口的子类的 run 方法。

实现 Runnable 接口比继承 Thread 类所具有的优势

- 避免了单继承的局限性
- 多个线程可以共享同一个接口实现类的对象，非常适合多个相同线程来处理同一份资源。
- 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。
- 线程池只能放入实现 Runable 或 Callable 类线程，不能直接放入继承 Thread 的类。

2.6 练习

创建两个分线程，让其中一个线程输出 1-100 之间的偶数，另一个线程输出 1-100 之间的奇数。

3. Thread 类的常用结构

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) :指定创建线程的目标对象，它实现了 Runnable 接口中的 run 方法
- 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。

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 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 String getName() :获取当前线程名称。
- public void setName(String name)：设置该线程名称。
- public static Thread currentThread() :返回对当前正在执行的线程对象的引用。在 Thread 子类中就是 this，通常用于主线程和 Runnable 实现类
- 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，同优先级线程组成先进先出队列（先到先服务），使用分时调度策略。优先级高的线程采用抢占式策略，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。
  - 在单核处理器上，线程优先级可能会影响到线程执行的优先度。 在多核处理器中，线程优先级没有实际意义。

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: " + ge
tPriority());
                }
        };
        t.setPriority(Thread.MAX_PRIORITY);
```

```
            t.start();

            System.out.println(Thread.currentThread().getName() +"的优先级：" +
                                                                Thread.currentThread().getPriority());
        }
```

3.3 常用方法系列 2

- 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。
- public final void stop()：已过时，强制线程生命期结束。
- 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。

```
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(1000);
                    } 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();
    }
}
```

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。形象理解：兔死狗烹，鸟尽弓藏

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();
                  }
            }
      }
}
```

4. 多线程的优点、场景、生命周期

4.1 使用多线程的优点

背景：以单核 CPU 为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

1. 提高应用程序的响应。对图形化界面更有意义，可增强用户体验。
2. 提高计算机系统 CPU 的利用率
3. 改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

4.2 何时需要多线程

- 程序需要同时执行两个或多个任务。
- 程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。
- 需要一些后台运行的程序时。

4.3 线程的生命周期

要想实现多线程，必须在主线程中创建新的线程对象。Java 语言使用 Thread 类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下一些状态：

4.3.1 JDK1.5 之前：5 种状态

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。

image-20220401002307038

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 资源时，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 核心，在任何时刻只有一个线程处于运行状态，如果计算机有多个核心，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，会让出 CPU 并临时中止自己的执行，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）

4.3.2 JDK1.5 及之后：6 种状态

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`（锁阻塞）：在 API 中的介绍为：一个正在阻塞、等待一个监视器锁（锁对象）的线程处于这一状态。只有获得锁对象的线程才能有执行机会。
  - 比如，线程 A 与线程 B 代码中使用同一锁，如果线程 A 获取到锁，线程 A 进入到 Runnable 状态，那么线程 B 就进入到 Blocked 锁阻塞状态。
- `TIMED_WAITING`（计时等待）：在 API 中的介绍为：一个正在限时等待另一个线程执行一个（唤醒）动作的线程处于这一状态。
  - 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`（无限等待）：在 API 中介绍为：一个正在无限期等待另一个线程执行一个特别的（唤醒）动作的线程处于这一状态。
  - 当前线程执行过程中遇到遇到 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。

我们在翻阅 API 的时候会发现 Timed Waiting（计时等待） 与 Waiting（无限等待） 状态联系还是很紧密的， 比如 Waiting（无限等待） 状态中 wait 方法是空参的，而 timed waiting（计时等待） 中 wait 方法是带参的。 这种带参的方法，其实是一种倒计时操作，相当于我们生活中的小闹钟，我们设定好时间，到时通知，可是 如果提前得到（唤醒）通知，那么设定好时间在通知也就显得多此一举了，那么这种设计方案其实是一举两 得。如果没有得到（唤醒）通知，那么线程就处于 Timed Waiting 状态，直到倒计时完毕自动醒来；如果在倒 计时期间得到（唤醒）通知，那么线程从 Timed Waiting 状态立刻唤醒。

代码演示：

```
/**
 * @author 尚硅谷-宋红康
 * @create 22:15
 */
public class ThreadStateTest {
    public static void main(String[] args) throws InterruptedException
{
        SubThread t = new SubThread();
```

```java
        System.out.println(t.getName() + " 状态 " + t.getState());
        t.start();

        while (Thread.State.TERMINATED != t.getState()) {
            System.out.println(t.getName() + " 状态 " + t.getState());
            Thread.sleep(500);
        }
        System.out.println(t.getName() + " 状态 " + t.getState());
    }
}

class SubThread extends Thread {
    @Override
    public void run() {
        while (true) {
            for (int i = 0; i < 10; i++) {
                System.out.println("打印: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            break;
        }
    }
}
```

命令行演示：

```
C:\Users\songhk>jps
10100 Jps
10196 ThreadSyncTest
15484
3068 Launcher

C:\Users\songhk>jstack 10196
2021-01-16 15:34:11
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.131-b11 mixed mode):

"DestroyJavaVM" #14 prio=5 os_prio=0 tid=0x0000000002b42800 nid=0x268c waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"线程2" #13 prio=5 os_prio=0 tid=0x000000002763e000 nid=0x608 waiting for monitor entry [0x000000002934f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at com.atguigu.jstack.Number.run(ThreadSyncTest.java:29)
        - waiting to lock <0x00000007166981a8> (a com.atguigu.jstack.Number)
        at java.lang.Thread.run(Thread.java:748)

"线程1" #12 prio=5 os_prio=0 tid=0x000000002763d000 nid=0x1fec waiting on condition [0x000000002924e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at com.atguigu.jstack.Number.run(ThreadSyncTest.java:32)
        - locked <0x00000007166981a8> (a com.atguigu.jstack.Number)
        at java.lang.Thread.run(Thread.java:748)

"Service Thread" #11 daemon prio=9 os_prio=0 tid=0x00000000275d7000 nid=0xa88 runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread3" #10 daemon prio=9 os_prio=2 tid=0x000000002753e800 nid=0x38d4 waiting on condition [0x0000000000000000]
```

5. 线程安全问题及解决

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。

举例 1：

image-20220401233218326

举例 2：

火车站要卖票，我们模拟火车站的卖票过程。因为疫情期间，本次列车的座位共 100 个（即，只能出售 100 张火车票）。我们来模拟车站的售票窗口，实现多个窗口同时售票的过程。

5.1 同一个资源问题和线程安全问题

5.1.1 局部变量不能共享

示例代码：

```
package com.atguigu.unsafe;

class Window extends Thread {
    public void run() {
        int ticket = 100;
        while (ticket > 0) {
            System.out.println(getName() + "卖出一张票，票号:" + ticket);
            ticket--;
        }
    }
}
```

```java
public class SaleTicketDemo1 {
    public static void main(String[] args) {
        Window w1 = new Window();
        Window w2 = new Window();
        Window w3 = new Window();

        w1.setName("窗口 1");
        w2.setName("窗口 2");
        w3.setName("窗口 3");

        w1.start();
        w2.start();
        w3.start();
    }
}
```

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 ticket 是独立的，不是共享数据。

5.1.2 不同对象的实例变量不共享

```java
package com.atguigu.unsafe;

class TicketWindow extends Thread {
    private int ticket = 100;

    public void run() {
        while (ticket > 0) {
            System.out.println(getName() + "卖出一张票，票号:" + ticket);
            ticket--;
        }
    }
}

public class SaleTicketDemo2 {
    public static void main(String[] args) {
        TicketWindow w1 = new TicketWindow();
        TicketWindow w2 = new TicketWindow();
        TicketWindow w3 = new TicketWindow();

        w1.setName("窗口 1");
        w2.setName("窗口 2");
        w3.setName("窗口 3");

        w1.start();
        w2.start();
```

```
        w3.start();
    }
}
```

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。

5.1.3 静态变量是共享的

示例代码：

```
package com.atguigu.unsafe;

class TicketSaleThread extends Thread {
    private static int ticket = 100;

    public void run() {
        while (ticket > 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，票号:" + ticket);
            ticket--;
        }
    }
}

public class SaleTicketDemo3 {
    public static void main(String[] args) {
        TicketSaleThread t1 = new TicketSaleThread();
        TicketSaleThread t2 = new TicketSaleThread();
        TicketSaleThread t3 = new TicketSaleThread();

        t1.setName("窗口 1");
        t2.setName("窗口 2");
        t3.setName("窗口 3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

运行结果：

```
窗口 1 卖出一张票，票号:100
窗口 2 卖出一张票，票号:100
窗口 3 卖出一张票，票号:100
窗口 3 卖出一张票，票号:97
窗口 1 卖出一张票，票号:97
窗口 2 卖出一张票，票号:97
窗口 1 卖出一张票，票号:94
窗口 3 卖出一张票，票号:94
窗口 2 卖出一张票，票号:94
窗口 2 卖出一张票，票号:91
窗口 1 卖出一张票，票号:91
窗口 3 卖出一张票，票号:91
窗口 3 卖出一张票，票号:88
窗口 1 卖出一张票，票号:88
窗口 2 卖出一张票，票号:88
窗口 3 卖出一张票，票号:85
窗口 1 卖出一张票，票号:85
窗口 2 卖出一张票，票号:85
窗口 3 卖出一张票，票号:82
窗口 1 卖出一张票，票号:82
窗口 2 卖出一张票，票号:82
窗口 2 卖出一张票，票号:79
窗口 3 卖出一张票，票号:79
窗口 1 卖出一张票，票号:79
窗口 3 卖出一张票，票号:76
窗口 1 卖出一张票，票号:76
窗口 2 卖出一张票，票号:76
窗口 1 卖出一张票，票号:73
窗口 2 卖出一张票，票号:73
窗口 3 卖出一张票，票号:73
窗口 2 卖出一张票，票号:70
窗口 1 卖出一张票，票号:70
窗口 3 卖出一张票，票号:70
窗口 2 卖出一张票，票号:67
窗口 3 卖出一张票，票号:67
窗口 1 卖出一张票，票号:67
窗口 1 卖出一张票，票号:64
窗口 3 卖出一张票，票号:64
窗口 2 卖出一张票，票号:64
窗口 2 卖出一张票，票号:61
窗口 3 卖出一张票，票号:61
窗口 1 卖出一张票，票号:61
窗口 1 卖出一张票，票号:58
窗口 2 卖出一张票，票号:58
窗口 3 卖出一张票，票号:58
```

窗口 2 卖出一张票，票号:55
窗口 1 卖出一张票，票号:55
窗口 3 卖出一张票，票号:55
窗口 3 卖出一张票，票号:52
窗口 1 卖出一张票，票号:52
窗口 2 卖出一张票，票号:52
窗口 2 卖出一张票，票号:49
窗口 1 卖出一张票，票号:49
窗口 3 卖出一张票，票号:49
窗口 2 卖出一张票，票号:46
窗口 3 卖出一张票，票号:46
窗口 1 卖出一张票，票号:46
窗口 2 卖出一张票，票号:43
窗口 3 卖出一张票，票号:43
窗口 1 卖出一张票，票号:43
窗口 3 卖出一张票，票号:40
窗口 1 卖出一张票，票号:40
窗口 2 卖出一张票，票号:40
窗口 2 卖出一张票，票号:37
窗口 3 卖出一张票，票号:37
窗口 1 卖出一张票，票号:37
窗口 2 卖出一张票，票号:34
窗口 1 卖出一张票，票号:34
窗口 3 卖出一张票，票号:34
窗口 3 卖出一张票，票号:31
窗口 2 卖出一张票，票号:31
窗口 1 卖出一张票，票号:31
窗口 1 卖出一张票，票号:28
窗口 2 卖出一张票，票号:28
窗口 3 卖出一张票，票号:28
窗口 2 卖出一张票，票号:25
窗口 1 卖出一张票，票号:25
窗口 3 卖出一张票，票号:25
窗口 2 卖出一张票，票号:22
窗口 3 卖出一张票，票号:22
窗口 1 卖出一张票，票号:22
窗口 3 卖出一张票，票号:19
窗口 1 卖出一张票，票号:19
窗口 2 卖出一张票，票号:19
窗口 2 卖出一张票，票号:16
窗口 3 卖出一张票，票号:16
窗口 1 卖出一张票，票号:16
窗口 2 卖出一张票，票号:13
窗口 1 卖出一张票，票号:13
窗口 3 卖出一张票，票号:13

```
窗口 2 卖出一张票，票号:10
窗口 1 卖出一张票，票号:10
窗口 3 卖出一张票，票号:10
窗口 3 卖出一张票，票号:7
窗口 1 卖出一张票，票号:7
窗口 2 卖出一张票，票号:7
窗口 3 卖出一张票，票号:4
窗口 1 卖出一张票，票号:4
窗口 2 卖出一张票，票号:4
窗口 3 卖出一张票，票号:1
窗口 2 卖出一张票，票号:1
窗口 1 卖出一张票，票号:1
```

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

5.1.4 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

```java
package com.atguigu.safe;

class TicketSaleRunnable implements Runnable {
    private int ticket = 100;

    public void run() {
        while (ticket > 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，票号:" + ticket);
            ticket--;
        }
    }
}

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();
    }
}
```

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

5.1.5 抽取资源类，共享同一个资源对象

示例代码：

```
package com.atguigu.unsafe;

//1、编写资源类
class Ticket {
    private int ticket = 100;

    public void sale() {
        if (ticket > 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，票号:" + ticket);
            ticket--;
        } else {
            throw new RuntimeException("没有票了");
        }
    }

    public int getTicket() {
        return ticket;
    }
}

public class SaleTicketDemo5 {
    public static void main(String[] args) {
        //2、创建资源对象
```

```java
        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();
    }
}
```

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

5.2 同步机制解决线程安全问题

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)。

5.2.1 同步机制解决线程安全问题的原理

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”，我们称它为同步锁。因为 Java 对象在堆中的数据分为分为对象头、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。

5.2.2 同步代码块和同步方法

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
}
```

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
}
```

5.2.3 同步锁机制

在《Thinking in Java》中，是这么说的：对于并发工作，你需要某种方式来防止两个任务访问相同的资源（其实就是共享资源竞争）。防止这种冲突的方法就是当资源被一个任务使用时，在其上加锁。第一个访问某项资源的任务必须锁定这项资源，使其他任务在其被解锁之前，就无法访问它了，而在其被解锁之时，另一个任务就可以锁定并使用它了。

5.2.4 synchronized 的锁是什么

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（很多时候也是指定为 this 或类名.class），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（类名.class）
- 非静态方法：this

5.2.5 同步操作的思考顺序

1、如何找问题，即代码是否存在线程安全？（非常重要）（1）明确哪些代码是多线程运行的代码 （2）明确多个线程是否有共享数据 （3）明确多线程运行代码中是否有多条语句操作共享数据

2、如何解决呢？（非常重要）对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行。即所有操作共享数据的这些语句都要放在同步范围中

3、切记：

范围太小：不能解决安全问题

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。

5.2.6 代码演示

示例一：静态方法加锁

```
package com.atguigu.safe;

class TicketSaleThread extends Thread{
    private static int ticket = 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 (ticket > 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(ticket > 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，票号:" + ticket);
            ticket--;
        }
    }
}
public class SaleTicketDemo3 {
    public static void main(String[] args) {
        TicketSaleThread t1 = new TicketSaleThread();
        TicketSaleThread t2 = new TicketSaleThread();
        TicketSaleThread t3 = new TicketSaleThread();

        t1.setName("窗口 1");
        t2.setName("窗口 2");
        t3.setName("窗口 3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

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 ticket = 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 (ticket > 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 (ticket > 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，票号:" + ticket);
            ticket--;
        }
    }
}
```

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()直接加锁，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() {
                while (true) {
                    synchronized (ticket) {
                        ticket.sale();
                    }
                }
            }
        }, "窗口三");


        t1.start();
        t2.start();
        t3.start();
    }
}

//1、编写资源类
class Ticket {
    private int ticket = 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 (ticket > 0) {
            System.out.println(Thread.currentThread().getName() + "卖出一张票，票号:" + ticket);
            ticket--;
        } else {
            throw new RuntimeException("没有票了");
        }
    }

    public int getTicket() {
        return ticket;
    }
}
```

5.3 练习

银行有一个账户。 有两个储户分别向同一个账户存 3000 元，每次存 1000，存 3 次。每次存完打印账户余额。

问题：该程序是否有安全问题，如果有，如何解决？

【提示】 1，明确哪些代码是多线程运行代码，须写入 run()方法 2，明确什么是共享数据。 3，明确多线程运行代码中哪些语句是操作共享数据的。

【拓展问题】可否实现两个储户交替存钱的操作

6. 再谈同步

6.1 单例设计模式的线程安全问题

6.1.1 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 HungrySingle INSTANCE = new HungrySingle(); //对象是否声明为final 都可以

    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);
```

```
    }
}
```

6.1.2 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：

```java
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
mem = allocate(); 为单例对象分配内存空间
instance = mem;   instance 引用现在非空，但还未初始化
ctorSingleton(instance); 为单例对象通过instance 调用构造函数
从 JDK2 开始，分配空间、初始化、调用构造函数会在线程的工作存储区一次性完成，然后复制到主存储区。但是需要volatile
*/

/*    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(){}

    public static LazySingle getInstance(){
        return Inner.INSTANCE;
    }

    private static class Inner{
        static final LazySingle INSTANCE = new LazySingle();
    }

}
```

内部类只有在外部类被调用才加载，产生 INSTANCE 实例；又不用加锁。

此模式具有之前两个模式的优点，同时屏蔽了它们的缺点，是最好的单例模式。

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(){
```

```java
        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);
```

```
    }
}
```

6.2 死锁

不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，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。

举例 1：

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

            StringBuilder s1 = new StringBuilder();
            StringBuilder s2 = new StringBuilder();

            new Thread() {
                  public void run() {
                        synchronized (s1) {
                              s1.append("a");
                              s2.append("1");

                              try {
                                    Thread.sleep(10);
                              } catch (InterruptedException e) {
                                    e.printStackTrace();
                              }

                              synchronized (s2) {
                                    s1.append("b");
                                    s2.append("2");

                                    System.out.println(s1);
                                    System.out.println(s2);

                              }
                        }
                  }
            }.start();

            new Thread() {
                  public void run() {
                        synchronized (s2) {
                              s1.append("c");
                              s2.append("3");
```

```
                                try {
                                        Thread.sleep(10);
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }

                                synchronized (s1) {
                                        s1.append("d");
                                        s2.append("4");

                                        System.out.println(s1);
                                        System.out.println(s2);

                                }

                        }
                }
            }.start();

        }
}
```

举例 2：

```
class A {
        public synchronized void foo(B b) {
                System.out.println("当前线程名: " + Thread.currentThread().g
etName()
                                + " 进入了 A 实例的 foo 方法"); // ①
                try {
                        Thread.sleep(200);
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
                System.out.println("当前线程名: " + Thread.currentThread().g
etName()
                                + " 企图调用 B 实例的 last 方法"); // ③
                b.last();
        }

        public synchronized void last() {
                System.out.println("进入了 A 类的 last 方法内部");
        }
}

class B {
        public synchronized void bar(A a) {
                System.out.println("当前线程名: " + Thread.currentThread().g
etName()
```

```
                        + " 进入了 B 实例的 bar 方法"); // ②
            try {
                  Thread.sleep(200);
            } catch (InterruptedException ex) {
                  ex.printStackTrace();
            }
            System.out.println("当前线程名: " + Thread.currentThread().g
etName()
                        + " 企图调用 A 实例的 last 方法"); // ④
            a.last();
      }

      public synchronized void last() {
            System.out.println("进入了 B 类的 last 方法内部");
      }
}

public class DeadLock implements Runnable {
      A a = new A();
      B b = new B();

      public void init() {
            Thread.currentThread().setName("主线程");
            // 调用 a 对象的 foo 方法
            a.foo(b);
            System.out.println("进入了主线程之后");
      }

      public void run() {
            Thread.currentThread().setName("副线程");
            // 调用 b 对象的 bar 方法
            b.bar(a);
            System.out.println("进入了副线程之后");
      }

      public static void main(String[] args) {
            DeadLock dl = new DeadLock();
            new Thread(dl).start();
            dl.init();
      }
}
```

举例 3：

```
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");
                }
            }
        }
}
```

解决方法：

- 尽量减少同步资源的定义

- 尽量避免嵌套同步

【小故事】

面试官：你能解释清楚什么是死锁，我就录取你！ 面试者：你录取我，我就告诉你什么是死锁！ .... 恭喜你，面试通过了

6.3 JDK5.0 新特性：Lock(锁)

- JDK1.5 后新增功能，保证线程的安全。与采用 synchronized 相比，lock 可提供多种锁方案，更灵活、更强大。Lock 通过显式定义同步锁对象来实现同步。同步锁使用 Lock 对象充当。
- java.util.concurrent.locks.Lock 接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对 Lock 对象加锁，线程开始访问共享资源之前应先获得 Lock 对象。
- ReentrantLock 类实现了 Lock 接口，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。
- 在实现线程安全的控制中，比较常用的是 ReentrantLock，可以显式加锁、释放锁。
- Lock 锁也称同步锁，加锁与释放锁方法，如下：
  - public void lock() :加同步锁。
  - public void unlock() :释放同步锁。
- 代码结构

```
class A{
	private final ReentrantLock lock = new ReenTrantLock();
	public void m(){
		lock.lock();
		try{
			//保证线程安全的代码;
		}
		finally{
			lock.unlock();
		}
	}
}
```

注意：如果同步代码有异常，要将 unlock()写入 finally 语句块。

举例：

```java
import java.util.concurrent.locks.ReentrantLock;

class Window implements Runnable{
	int ticket = 100;
	private final ReentrantLock lock = new ReentrantLock();
	public void run(){

		while(true){
			try{
				lock.lock();
				if(ticket > 0){
					try {
						Thread.sleep(10);
					} catch (InterruptedException e) {
						e.printStackTrace();
					}
					System.out.println(ticket--);
				}else{
					break;
				}
			}finally{
				lock.unlock();
			}
		}
	}
}

public class ThreadLock {
	public static void main(String[] args) {
		Window t = new Window();
		Thread t1 = new Thread(t);
		Thread t2 = new Thread(t);

		t1.start();
		t2.start();
	}
}
```

synchronized 与 Lock 的对比

1. Lock 是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized 是隐式锁，出了作用域、遇到异常等自动解锁
2. Lock 只有代码块锁，synchronized 有代码块锁和方法锁
3. 使用 Lock 锁，JVM 将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类），更体现面向对象。
4. （了解）Lock 锁可以对读不加锁，对写加锁，synchronized 不可以

5. （了解）Lock 锁可以有多种获取锁的方式，可以从 sleep 的线程中抢到锁，synchronized 不可以

康师傅有话说：

开发建议中处理线程安全问题优先使用顺序为：

- Lock ----> 同步代码块 ----> 同步方法

7. 线程的通信

7.1 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）却不相同。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，以此实现多线程共同操作一份数据。

比如：线程 A 用来生产包子的，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，一个是生产，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。**

7.2 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。

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“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，所以它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

7.3 举例

例题：使用两个线程打印 1-100。线程 1, 线程 2 交替打印

```
class Communication implements Runnable {
    int i = 1;
    public void run() {
        while (true) {
            synchronized (this) {
                notify();
                if (i <= 100) {
                    System.out.println(Thread.currentThread().getName()
 + ":" + i++);
                } else
                    break;
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

7.4 调用 wait 和 notify 需注意的细节

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。否则会报 java.lang.IllegalMonitorStateException 异常。

7.5 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。**

举例：

生产者(Productor)将产品交给店员(Clerk)，而消费者(Customer)从店员处取走产品，店员一次只能持有固定数量的产品(比如:20），如果生产者试图生产更多的产品，店员会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。

类似的场景，比如厨师和服务员等。

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，产生安全问题。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。

代码实现：

```
public class ConsumerProducerTest {
      public static void main(String[] args) {
            Clerk clerk = new Clerk();
            Producer p1 = new Producer(clerk);

            Consumer c1 = new Consumer(clerk);
            Consumer c2 = new Consumer(clerk);
```

```java
            p1.setName("生产者 1");
            c1.setName("消费者 1");
            c2.setName("消费者 2");

            p1.start();
            c1.start();
            c2.start();
        }
}

//生产者
class Producer extends Thread{
        private Clerk clerk;

        public Producer(Clerk clerk){
                this.clerk = clerk;
        }

        @Override
        public void run() {

                System.out.println("=========生产者开始生产产品========");
                while(true){

                        try {
                                Thread.sleep(40);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }

                        //要求 clerk 去增加产品
                        clerk.addProduct();
                }
        }
}

//消费者
class Consumer extends Thread{
        private Clerk clerk;

        public Consumer(Clerk clerk){
                this.clerk = clerk;
        }
        @Override
        public void run() {
                System.out.println("=========消费者开始消费产品========");
                while(true){
```

```java
                try {
                        Thread.sleep(90);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                //要求clerk去减少产品
                clerk.minusProduct();
        }
    }
}

//资源类
class Clerk {
    private int productNum = 0;//产品数量
    private static final int MAX_PRODUCT = 20;
    private static final int MIN_PRODUCT = 1;

    //增加产品
    public synchronized void addProduct() {
        if(productNum < MAX_PRODUCT){
            productNum++;
            System.out.println(Thread.currentThread().getName() +

                    "生产了第" + productNum + "个产品");
            //唤醒消费者
            this.notifyAll();
        }else{

            try {
                    this.wait();
            } catch (InterruptedException e) {
                    e.printStackTrace();
            }
        }
    }

    //减少产品
    public synchronized void minusProduct() {
        if(productNum >= MIN_PRODUCT){
            System.out.println(Thread.currentThread().getName() +

                    "消费了第" + productNum + "个产品");
            productNum--;

            //唤醒生产者
            this.notifyAll();
        }else{
```

```
                try {
                        this.wait();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
            }
        }

}
```

7.6 面试题：区分 sleep()和 wait()

相同点：一旦执行，都会使得当前线程结束执行状态，进入阻塞状态。

不同点：

① 定义方法所属的类：sleep():Thread 中定义。 wait():Object 中定义

② 使用范围的不同：sleep()可以在任何需要使用的位置被调用； wait():必须使用在同步代码块或同步方法中

③ 都在同步结构中使用的时候，是否释放同步监视器的操作不同：sleep():不会释放同步监视器 ;wait():会释放同步监视器

④ 结束等待的方式不同：sleep()：指定时间一到就结束阻塞。 wait():可以指定时间也可以无限等待直到 notify 或 notifyAll。

7.7 是否释放锁的操作

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？

7.7.1 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、同步方法中遇到 break、return 终止了该代码块、该方法的继续执行。

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。

7.7.2 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。

8. JDK5.0 新增线程创建方式

8.1 新增方式一：实现 Callable 接口

- 与使用 Runnable 相比， Callable 功能更强大些
  - 相比 run()方法，可以有返回值
  - 方法可以抛出异常
  - 支持泛型的返回值
  - 需要借助 FutureTask 类，比如获取返回结果
- Future 接口
  - 可以对具体 Runnable、Callable 任务的执行结果进行取消、查询是否完成、获取结果等。
  - FutureTask 是 Futrue 接口的唯一的实现类
  - FutureTask 同时实现了 Runnable, Future 接口。它既可以作为 Runnable 被线程执行，又可以作为 Future 得到 Callable 的返回值
- 代码举例

```
/*
 * 创建多线程的方式三：实现Callable （jdk5.0 新增的）
 */
//1.创建一个实现Callable 的实现类
class NumThread implements Callable {
    //2.实现call 方法，将此线程需要执行的操作声明在call()中
    @Override
    public Object call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= 100; i++) {
            if (i % 2 == 0) {
                System.out.println(i);
                sum += i;
            }
```

```
        }
        return sum;
    }
}


public class CallableTest {
    public static void main(String[] args) {
        //3.创建Callable接口实现类的对象
        NumThread numThread = new NumThread();

        //4.将此Callable接口实现类的对象作为传递到FutureTask构造器中，创建FutureTask的对象
        FutureTask futureTask = new FutureTask(numThread);
        //5.将FutureTask的对象作为参数传递到Thread类的构造器中，创建Thread对象，并调用start()
        new Thread(futureTask).start();


//      接收返回值
        try {
            //6.获取Callable中call方法的返回值
            //get()返回值即为FutureTask构造器参数Callable实现类重写的call()的返回值。
            Object sum = futureTask.get();
            System.out.println("总和为: " + sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

}
```

8.2 新增方式二：使用线程池

现有问题：

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务？

思路：提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。

好处：

- 提高响应速度（减少了创建新线程的时间）
- 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
- 便于线程管理
  - corePoolSize：核心池的大小
  - maximumPoolSize：最大线程数
  - keepAliveTime：线程没有任务时最多保持多长时间后会终止
  - ...

线程池相关 API

- JDK5.0 之前，我们必须手动自定义线程池。从 JDK5.0 开始，Java 内置线程池相关的 API。在 java.util.concurrent 包下提供了线程池相关 API：`ExecutorService` 和 `Executors`。
- `ExecutorService`：真正的线程池接口。常见子类 ThreadPoolExecutor
  - `void execute(Runnable command)`：执行任务/命令，没有返回值，一般用来执行 Runnable

- <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行 Callable
- void shutdown()：关闭连接池

- Executors：一个线程池的工厂类，通过此类的静态工厂方法可以创建多种类型的线程池对象。
  - Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
  - Executors.newFixedThreadPool(int nThreads);创建一个可重用固定线程数的线程池
  - Executors.newSingleThreadExecutor()：创建一个只有一个线程的线程池
  - Executors.newScheduledThreadPool(int corePoolSize)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

代码举例：

```
class NumberThread implements Runnable{

    @Override
    public void run() {
        for(int i = 0;i <= 100;i++){
            if(i % 2 == 0){
                System.out.println(Thread.currentThread().getName() + ":
 " + i);
            }
        }
    }
}

class NumberThread1 implements Runnable{

    @Override
    public void run() {
        for(int i = 0;i <= 100;i++){
            if(i % 2 != 0){
                System.out.println(Thread.currentThread().getName() + ":
 " + i);
            }
        }
    }
}
```

```
public class ThreadPool {

    public static void main(String[] args) {
        //1. 提供指定线程数量的线程池
        ExecutorService service = Executors.newFixedThreadPool(10);
        ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;
//        //设置线程池的属性
//        System.out.println(service.getClass());//ThreadPoolExecutor
        service1.setMaximumPoolSize(20);


        //2.执行指定的线程的操作。需要提供实现Runnable 接口或Callable 接口
实现类的对象
        service.execute(new NumberThread());//适合适用于Runnable
        service.execute(new NumberThread1());//适合适用于Runnable

//        service.submit(Callable callable);//适合使用于Callable
        //3.关闭连接池
        service.shutdown();
    }

}
```

第 11 章_常用类和基础 API

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 字符串相关类之不可变字符序列：String

1.1 String 的特性

- `java.lang.String` 类代表字符串。Java 程序中所有的字符串文字（例如 `"hello"` ）都可以看作是实现此类的实例。
- 字符串是常量，用双引号引起来表示。它们的值在创建之后不能更改。
- 字符串 String 类型本身是 final 声明的，意味着我们不能继承 String。
- String 对象的字符内容是存储在一个字符数组 value[]中的。`"abc"` 等效于 `char[] data={'h','e','l','l','o'}`。

image-20220514184404024

```
//jdk8 中的String 源码:
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequ
ence {
    /** The value is used for character storage. */
    private final char value[]; //String 对象的字符内容是存储在此数
组中

    /** Cache the hash code for the string */
    private int hash; // Default to 0
```

- private 意味着外面无法直接获取字符数组，而且 String 没有提供 value 的 get 和 set 方法。
- final 意味着字符数组的引用不可改变，而且 String 也没有提供方法来修改 value 数组某个元素值
- 因此字符串的字符数组内容也不可变的，即 String 代表着不可变的字符序列。即，一旦对字符串进行修改，就会产生新对象。
- JDK9 只有，底层使用 byte[]数组。

```
public final class String implements java.io.Serializable,
Comparable<String>, CharSequence {
    @Stable
    private final byte[] value;
}
//官方说明: ... that most String objects contain only Latin-
1 characters. Such characters require only one byte of stor
age, hence half of the space in the internal char arrays of
 such String objects is going unused.
//细节: ... The new String class will store characters encod
ed either as ISO-8859-1/Latin-1 (one byte per character), o
r as UTF-16 (two bytes per character), based upon the conte
nts of the string. The encoding flag will indicate which en
coding is used.
```

- 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）。

1.2 String 的内存结构

1.2.1 概述

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。

JDK6 中，字符串常量池在方法区。JDK7 开始，就移到堆空间，直到目前 JDK17 版本。举例内存结构分配：

1.2.2 练习类型1：拼接

```
String s1 = "hello";
String s2 = "hello";
System.out.println(s1 == s2);
// 内存中只有一个"hello"对象被创建，同时被s1和s2共享。
```

对应内存结构为：（以下内存结构以 JDK6 为例绘制）：

image-20220405152839525

进一步：

image-20220405152941599

image-20220405152955544

```
Person p1 = new Person();
p1.name = “Tom";

Person p2 = new Person();
p2.name = “Tom";

System.out.println(p1.name .equals( p2.name)); //
System.out.println(p1.name == p2.name); //
System.out.println(p1.name == “Tom"); //
```

image-20220405153027693

1.2.3 练习类型 2：new

String str1 = “abc”; 与 String str2 = new String(“abc”);的区别？

image-20220405160149200

str2 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。

- 字符串常量存储在字符串常量池，目的是共享。

- 字符串非常量对象存储在堆中。

练习：

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

System.out.println(s1 == s2);//true
System.out.println(s1 == s3);//false
System.out.println(s1 == s4);//false
System.out.println(s3 == s4);//false
```

image-20220405160321172

练习：String str2 = new String("hello"); 在内存中创建了几个对象？

两个

1.2.4 练习类型 3：intern()

- **String s1 = "a";**

说明：在字符串常量池中创建了一个字面量为"a"的字符串。

- **s1 = s1 + "b";**

说明：实际上原来的“a”字符串对象已经丢弃了，现在在堆空间中产生了一个字符串 s1+"b"（也就是"ab"）。如果多次执行这些改变串内容的操作，会导致大量副本字符串对象存留在内存中，降低效率。如果这样的操作放到循环中，会极大影响程序的性能。

- **String s2 = "ab";**

说明：直接在字符串常量池中创建一个字面量为"ab"的字符串。

- **String s3 = "a" + "b";**

说明：s3 指向字符串常量池中已经创建的"ab"的字符串。

- **String s4 = s1.intern();**

说明：堆空间的 s1 对象在调用 intern()之后，会将常量池中已经存在的"ab"字符串赋值给 s4。

练习：

```
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);
System.out.println(s3 == s5);
System.out.println(s4 == s5);
System.out.println(s3 == s6);
```

结论：

（1）常量+常量：结果是常量池。且常量池中不会存在相同内容的常量。

（2）常量与变量 或 变量与变量：结果在堆中

（3）拼接后调用 intern 方法：返回值在常量池中

练习：

```
@Test
public void test01(){
      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，"wo
rld"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
}

@Test
public void test02(){
        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，"wo
rld"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 test01(){
        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
}
```

练习：下列程序运行的结果：

```
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。

练习：下列程序运行的结果：

```
public class StringTest {

    String str = new String("good");
    char[] ch = { 't', 'e', 's', 't' };

    public void change(String str, char ch[]) {
        str = "test ok";
        ch[0] = 'b';
    }
    public static void main(String[] args) {
        StringTest ex = new StringTest();
        ex.change(ex.str, ex.ch);
        System.out.print(ex.str + " and ");//
        System.out.println(ex.ch);
    }
}
```

1.3 String 的常用 API-1

1.3.1 构造器

- `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。

举例：

```
//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");

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);
}
```

1.3.2 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：比较的是对象的内容，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(strs)); //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
```

1.3.3 空字符的比较

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)
```

1.3.4 String 与其他结构间的转换

字符串 --> 基本数据类型、包装类：

- Integer 包装类的 public static int parseInt(String s)：可以将由“数字”字符组成的字符串转换为整型。
- 类似地，使用 java.lang 包中的 Byte、Short、Long、Float、Double 类调相应的类方法可以将由“数字”字符组成的字符串，转化为相应的基本数据类型。

基本数据类型、包装类 --> 字符串：

- 调用 String 类的 public String valueOf(int n)可将 int 型转换为字符串
- 相应的 valueOf(byte b)、valueOf(long l)、valueOf(float f)、valueOf(double d)、valueOf(boolean b)可由参数的相应类型到字符串的转换。

字符数组 --> 字符串：

- String 类的构造器：String(char[]) 和 String(char[]，int offset，int length) 分别用字符数组中的全部字符和部分字符创建字符串对象。

字符串 --> 字符数组：

- public char[] toCharArray()：将字符串中的全部字符存放在一个字符数组中的方法。
- public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)：提供了将指定索引范围内的字符串存放到数组中的方法。

字节数组 --> 字符串：（解码）

- String(byte[])：通过使用平台的默认字符集解码指定的 byte 数组，构造一个新的 String。
- String(byte[]，int offset，int length) ：用指定的字节数组的一部分，即从数组起始位置 offset 开始取 length 个字节构造一个字符串对象。
- 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。

字符串 --> 字节数组：（编码）

- public byte[] getBytes() ：使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。

- public byte[] getBytes(String charsetName)：使用指定的字符集将此 String 编码到 byte 序列，并将结果存储到新的 byte 数组。

代码示例：

```
    @Test
  public void test01() 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
  }

    @Test
  public void test02()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"));
  }
```

1.4 String 的常用 API-2

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。

1.4.1 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，理解为 +

（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 compareToIgnoreCase(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");
        }
    }
```

1.4.2 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 indexOf(String str, int fromIndex)：返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始

（14）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

（15）int lastIndexOf(String str, int fromIndex)：返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。

```
    @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 UI 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"));
    }
```

1.4.3 系列3：字符串截取

（16）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。

（17）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,fileNam
e.lastIndexOf(".")));
        //截取后缀名
        System.out.println("后缀名: " + fileName.substring(fileName.
lastIndexOf(".")));
    }
```

1.4.4 系列 4：和字符/字符数组相关

（18）char charAt(index)：返回[index]位置的字符

（19）char[] toCharArray()： 将此字符串转换为一个新的字符数组返回

（20）String(char[] value)：返回指定数组中表示该字符序列的 String。

（21）String(char[] value, int offset, int count)：返回指定数组中表示该字符序列的 String。

（22）static String copyValueOf(char[] data)： 返回指定数组中表示该字符序列的 String

（23）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

（24）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

（25）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);
    }
```

```java
@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);
}
@Test
public void test03(){
      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);
}
```

1.4.5 系列5：开头与结尾

（26）boolean startsWith(xx)：测试此字符串是否以指定的前缀开始

（27）boolean startsWith(String prefix, int toffset)：测试此字符串从指定索引开始的子字符串是否以指定前缀开始

（28）boolean endsWith(xx)：测试此字符串是否以指定的后缀结束

```java
@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");
      }
}
```

1.4.6 系列 6：正则匹配

（29）boolean matchs(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 8.2）

```
@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
    /*
     （1）密码的长度为6，且只能有[A-Za-z0-9]组成。
     （2）另外，三个非捕获组都能匹配到自己的值。
    (?=.*[A-Z])：匹配值  C
    (?=.*[a-z])：匹配值  Clyya
    (?=.*[0-9])：匹配值  Clyya1
    三个非捕获组都有值。
     */
}
```

1.4.7 系列7：替换

（30）String replace(char oldChar, char newChar)：返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。 不支持正则。

（31）String replace(CharSequence target, CharSequence replacement)：使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。

（32）String replaceAll(String regex, String replacement)：使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。

（33）String replaceFirst(String regex, String replacement)：使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。

```
@Test
public void test4(){
      String str1 = "hello244world.java;887";
      //把其中的非字母去掉
      str1 = str1.replaceAll("[^a-zA-Z]", "");
      System.out.println(str1);

  String str2 = "12hello34world5java7891mysql456";
      //把字符串中的数字替换成,，如果结果中开头和结尾有，的话去掉
      String string = str2.replaceAll("\\d+", ",").replaceAll("^,
|,$", "");
      System.out.println(string);

}
```

1.4.8 系列8：拆分

（34）String[] split(String regex)：根据给定正则表达式的匹配拆分此字符串。

（35）String[] split(String regex, int limit)：根据匹配给定的正则表达式来拆分此字符串，最多不超过 limit 个，如果超过了，剩下的全部都放到最后一个元素中。

```
@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]);
      }
}

@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 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 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]);
        }
    }
```

1.5 常见算法题目

题目 1：模拟一个 trim 方法，去除字符串两端的空格。

```
    public String myTrim(String str) {
        if (str != null) {
            int start = 0;// 用于记录从前往后首次索引位置不是空格的位置的索引
            int end = str.length() - 1;// 用于记录从后往前首次索引位置不是空格的位置的索引
```

```
            while (start < end && str.charAt(start) == ' ') {
                start++;
            }

            while (start < end && str.charAt(end) == ' ') {
                end--;
            }
            if (str.charAt(start) == ' ') {
                return "";
            }

            return str.substring(start, end + 1);
        }
        return null;
    }

    @Test
    public void testMyTrim() {
        String str = "   a   ";
        // str = " ";
        String newStr = myTrim(str);
        System.out.println("---" + newStr + "---");
    }
```

题目 2：将一个字符串进行反转。将字符串中指定部分进行反转。比如“abcdefg”反转为”abfedcg”

```
    // 方式一:
    public String reverse1(String str, int start, int end) {// start:
2,end:5
        if (str != null) {
            // 1.
            char[] charArray = str.toCharArray();
            // 2.
            for (int i = start, j = end; i < j; i++, j--) {
                char temp = charArray[i];
                charArray[i] = charArray[j];
                charArray[j] = temp;
            }
            // 3.
            return new String(charArray);

        }
        return null;

    }
```

```java
// 方式二:
public String reverse2(String str, int start, int end) {
	// 1.
	String newStr = str.substring(0, start);// ab
	// 2.
	for (int i = end; i >= start; i--) {
		newStr += str.charAt(i);
	} // abfedc
		// 3.
	newStr += str.substring(end + 1);
	return newStr;
}

// 方式三: 推荐 （相较于方式二做的改进）
public String reverse3(String str, int start, int end) {// ArrayL
ist list = new ArrayList(80);
	// 1.
	StringBuffer s = new StringBuffer(str.length());
	// 2.
	s.append(str.substring(0, start));// ab
	// 3.
	for (int i = end; i >= start; i--) {
		s.append(str.charAt(i));
	}

	// 4.
	s.append(str.substring(end + 1));

	// 5.
	return s.toString();

}

@Test
public void testReverse() {
	String str = "abcdefg";
	String str1 = reverse3(str, 2, 5);
	System.out.println(str1);// abfedcg

}
```

题目 3：获取一个字符串在另一个字符串中出现的次数。 比如：获取“ab”在“abkkcadkabkebfkabkskab” 中出现的次数

```java
// 第 3 题
// 判断 str2 在 str1 中出现的次数
public int getCount(String mainStr, String subStr) {
```

```
            if (mainStr.length() >= subStr.length()) {
                  int count = 0;
                  int index = 0;
                  // while((index = mainStr.indexOf(subStr)) != -1){
                  // count++;
                  // mainStr = mainStr.substring(index + subStr.length
());
                  // }
                  // 改进:
                 while ((index = mainStr.indexOf(subStr, index)) != -1)
 {
                        index += subStr.length();
                        count++;
                  }

                  return count;
            } else {
                  return 0;
            }

      }

      @Test
      public void testGetCount() {
            String str1 = "cdabkkcadkabkebfkabkskab";
            String str2 = "ab";
            int count = getCount(str1, str2);
            System.out.println(count);
      }
```

题目 4：获取两个字符串中最大相同子串。比如： str1 = "abcwerthelloyuiodef“;str2 = "cvhellobnm" 提示：将短的那个串进行长度依次递减的子串与较长的串比较。

```
      // 第4题
      // 如果只存在一个最大长度的相同子串
      public String getMaxSameSubString(String str1, String str2) {
            if (str1 != null && str2 != null) {
                  String maxStr = (str1.length() > str2.length()) ? str
1 : str2;
                  String minStr = (str1.length() > str2.length()) ? str
2 : str1;

                  int len = minStr.length();

                  for (int i = 0; i < len; i++) {// 0 1 2 3 4 此层循环决
定要去几个字符
```

```
                    for (int x = 0, y = len - i; y <= len; x++, y++)
 {
                        if (maxStr.contains(minStr.substring(x,
y))) {
                            return minStr.substring(x, y);
                        }
                    }
                }
            }
            return null;
        }

        // 如果存在多个长度相同的最大相同子串
        // 此时先返回String[]，后面可以用集合中的ArrayList 替换，较方便
        public String[] getMaxSameSubString1(String str1, String str2) {
            if (str1 != null && str2 != null) {
                StringBuffer sBuffer = new StringBuffer();
                String maxString = (str1.length() > str2.length()) ?
str1 : str2;
                String minString = (str1.length() > str2.length()) ?
str2 : str1;

                int len = minString.length();
                for (int i = 0; i < len; i++) {
                    for (int x = 0, y = len - i; y <= len; x++, y++)
 {
                        String subString = minString.substring(x,
 y);
                        if (maxString.contains(subString)) {
                            sBuffer.append(subString + ",");
                        }
                    }
                    System.out.println(sBuffer);
                    if (sBuffer.length() != 0) {
                        break;
                    }
                }
                String[] split = sBuffer.toString().replaceAll(",$",
"").split("\\,");
                return split;
            }

            return null;
        }
```

```
        // 如果存在多个长度相同的最大相同子串：使用ArrayList
//    public List<String> getMaxSameSubString1(String str1, String str2)
 {
//          if (str1 != null && str2 != null) {
//                List<String> list = new ArrayList<String>();
//                String maxString = (str1.length() > str2.length()) ?
str1 : str2;
//                String minString = (str1.length() > str2.length()) ?
str2 : str1;
//
//                int len = minString.length();
//                for (int i = 0; i < len; i++) {
//                     for (int x = 0, y = len - i; y <= len; x++, y++)
 {
//                          String subString = minString.substring(x,
 y);
//                          if (maxString.contains(subString)) {
//                               list.add(subString);
//                          }
//                     }
//                     if (list.size() != 0) {
//                          break;
//                     }
//                }
//                return list;
//          }
//
//          return null;
//    }

        @Test
        public void testGetMaxSameSubString() {
              String str1 = "abcwerthelloyuiodef";
              String str2 = "cvhellobnmiodef";
              String[] strs = getMaxSameSubString1(str1, str2);
              System.out.println(Arrays.toString(strs));
        }
```

题目 5：对字符串中字符进行自然顺序排序。 提示： 1）字符串变成字符数组。 2）对数组排序，选择，冒泡，Arrays.sort(); 3）将排序后的数组变成字符串。

```
        // 第5题
        @Test
        public void testSort() {
              String str = "abcwerthelloyuiodef";
              char[] arr = str.toCharArray();
              Arrays.sort(arr);
```

```
        String newStr = new String(arr);
        System.out.println(newStr);
    }
```

2. 字符串相关类之可变字符序列：StringBuffer、StringBuilder

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，空间消耗也比较高。因此，JDK 又在 java.lang 包提供了可变字符序列 StringBuilder 和 StringBuffer 类型。

2.1 StringBuffer 与 StringBuilder 的理解

- java.lang.StringBuffer 代表可变的字符序列，JDK1.0 中声明，可以对字符串内容进行增删，此时不会产生新的对象。比如：

```
//情况 1:
String s = new String("我喜欢学习");
//情况 2:
StringBuffer buffer = new StringBuffer("我喜欢学习");
buffer.append("数学");
```

- 作为参数传递的话，方法内部 String 不会改变其值，StringBuffer 和 StringBuilder 会改变其值。
- 继承结构：

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    /**
     * The value is used for character storage.
     */
    char[] value;

    /**
     * The count is the number of characters used.
     */
    int count;
```

value没有final声明,value可以指向新的数组。

count记录有效字符的个数。

- StringBuilder 和 StringBuffer 非常类似，均代表可变的字符序列，而且提供相关功能的方法也一样。
- 区分 String、StringBuffer、StringBuilder
  - String:不可变的字符序列； 底层使用 char[]数组存储(JDK8.0 中)
  - StringBuffer:可变的字符序列；线程安全（方法有 synchronized 修饰），效率低；底层使用 char[]数组存储 (JDK8.0 中)
  - StringBuilder:可变的字符序列； jdk1.5 引入，线程不安全的，效率高；底层使用 char[]数组存储(JDK8.0 中)

2.2 StringBuilder、StringBuffer 的 API

StringBuilder、StringBuffer 的 API 是完全一致的，并且很多方法与 String 相同。

1、常用 API

（1）StringBuffer append(xx)：提供了很多的 append()方法，用于进行字符串追加的方式拼接

（2）StringBuffer delete(int start, int end)：删除[start,end)之间字符

（3）StringBuffer deleteCharAt(int index)：删除[index]位置字符

（4）StringBuffer replace(int start, int end, String str)：替换[start,end)范围的字符序列为 str

（5）void setCharAt(int index, xx)：替换[index]位置字符

（6）char charAt(int index)：查找指定 index 位置上的字符

（7）StringBuffer insert(int index, xx)：在[index]位置插入 xx

（8）int length()：返回存储的字符数据的长度

（9）StringBuffer reverse()：反转

- 当 append 和 insert 时，如果原来 value 数组长度不够，可扩容。
- 如上(1)(2)(3)(4)(9)这些方法支持方法链操作。原理：

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

2、其它 API

（1）int indexOf(String str)：在当前字符序列中查询 str 的第一次出现下标

（2）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

（3）int lastIndexOf(String str)：在当前字符序列中查询 str 的最后一次出现下标

（4）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

（5）String substring(int start)：截取当前字符序列[start,最后]

（6）String substring(int start, int end)：截取当前字符序列[start,end)

（7）String toString()：返回此序列中数据的字符串表示形式

（8）void setLength(int newLength)：设置当前字符序列长度为 newLength

```
@Test
public void test1(){
        StringBuilder s = new StringBuilder();
        s.append("hello").append(true).append('a').append(12).appen
d("atguigu");
        System.out.println(s);
        System.out.println(s.length());
}

@Test
public void test2(){
        StringBuilder s = new StringBuilder("helloworld");
        s.insert(5, "java");
        s.insert(5, "chailinyan");
        System.out.println(s);
}
```

```java
@Test
public void test3(){
    StringBuilder s = new StringBuilder("helloworld");
    s.delete(1, 3);
    s.deleteCharAt(4);
    System.out.println(s);
}
@Test
public void test4(){
    StringBuilder s = new StringBuilder("helloworld");
    s.reverse();
    System.out.println(s);
}

@Test
public void test5(){
    StringBuilder s = new StringBuilder("helloworld");
    s.setCharAt(2, 'a');
    System.out.println(s);
}

@Test
public void test6(){
    StringBuilder s = new StringBuilder("helloworld");
    s.setLength(30);
    System.out.println(s);
}
```

2.3 效率测试

```java
//初始设置
long startTime = 0L;
long endTime = 0L;
String text = "";
StringBuffer buffer = new StringBuffer("");
StringBuilder builder = new StringBuilder("");

//开始对比
startTime = System.currentTimeMillis();
for (int i = 0; i < 20000; i++) {
    buffer.append(String.valueOf(i));
}
endTime = System.currentTimeMillis();
System.out.println("StringBuffer 的执行时间: " + (endTime - startTime));

startTime = System.currentTimeMillis();
for (int i = 0; i < 20000; i++) {
    builder.append(String.valueOf(i));
}
endTime = System.currentTimeMillis();
```

```
System.out.println("StringBuilder 的执行时间: " + (endTime - startTime));

startTime = System.currentTimeMillis();
for (int i = 0; i < 20000; i++) {
    text = text + i;
}
endTime = System.currentTimeMillis();
System.out.println("String 的执行时间: " + (endTime - startTime));
```

2.4 练习

笔试题：程序输出：

```
String str = null;
StringBuffer sb = new StringBuffer();
sb.append(str);

System.out.println(sb.length());//

System.out.println(sb);//

StringBuffer sb1 = new StringBuffer(str);
System.out.println(sb1);//
```

3. JDK8 之前：日期时间 API

3.1 java.lang.System 类的方法

- System 类提供的 public static long currentTimeMillis()：用来返回当前时间与 1970 年 1 月 1 日 0 时 0 分 0 秒之间以毫秒为单位的时间差。
- 此方法适于计算时间差。
- 计算世界时间的主要标准有：
  - UTC(Coordinated Universal Time)
  - GMT(Greenwich Mean Time)
  - CST(Central Standard Time)

  在国际无线电通信场合，为了统一起见，使用一个统一的时间，称为通用协调时(UTC, Universal Time Coordinated)。UTC 与格林尼治平均时(GMT, Greenwich Mean Time)一样，都与英国伦敦的本地时相同。这里，UTC 与 GMT 含义完全相同。

3.2 java.util.Date

表示特定的瞬间，精确到毫秒。

- 构造器：
  - Date()：使用无参构造器创建的对象可以获取本地当前时间。
  - Date(long 毫秒数)：把该毫秒值换算成日期时间对象
- 常用方法
  - getTime(): 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
  - toString(): 把此 Date 对象转换为以下形式的 String： dow mon dd hh:mm:ss zzz yyyy 其中： dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)，zzz 是时间标准。
  - 其它很多方法都过时了。
- 举例：

```java
@Test
public void test1(){
      Date d = new Date();
      System.out.println(d);
}

@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 test3(){
      Date d = new Date();
      long time = d.getTime();
      System.out.println(time);//1559807047979
}

@Test
public void test4(){
      long time = 1559807047979L;
      Date d = new Date(time);
      System.out.println(d);
}

@Test
public void test5(){
      long time = Long.MAX_VALUE;
```

```
        Date d = new Date(time);
        System.out.println(d);
    }
```

3.3 java.text.SimpleDateFormat

- Date 类的 API 不易于国际化，大部分被废弃了，java.text.SimpleDateFormat 类是一个不与语言环境有关的方式来格式化和解析日期的具体类。
- 可以进行格式化：日期 --> 文本
- 可以进行解析：文本 --> 日期
- **格式化：**
  - SimpleDateFormat() ：默认的模式和语言环境创建对象
  - public SimpleDateFormat(String pattern)：该构造方法可以用参数 pattern 指定的格式创建一个对象，该对象调用：
  - public String format(Date date)：方法格式化时间对象 date
- **解析：**
  - public Date parse(String source)：从给定字符串的开始解析文本，以生成一个日期。

尚硅谷

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

```
    //格式化
    @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);
    }
    //解析
    @Test
    public void test10() throws ParseException{
        String str = "2022 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);
    }
```

3.4 java.util.Calendar(日历)

image-20220511105252261

- `Calendar` 类是一个抽象类，主用用于完成日期字段之间相互操作的功能。
- 获取 Calendar 实例的方法
  - 使用 `Calendar.getInstance()`方法

static Calendar	**getInstance**() 使用默认时区和语言环境获得一个日历。
static Calendar	**getInstance**(Locale aLocale) 使用默认时区和指定语言环境获得一个日历。
static Calendar	**getInstance**(TimeZone zone) 使用指定时区和默认语言环境获得一个日历。
static Calendar	**getInstance**(TimeZone zone, Locale aLocale) 使用指定时区和语言环境获得一个日历。

  - 调用它的子类 GregorianCalendar（公历）的构造器。

- 一个 Calendar 的实例是系统时间的抽象表示，可以修改或获取 YEAR、MONTH、DAY*OF*WEEK、HOUR*OF*DAY 、MINUTE、SECOND 等日历字段对应的时间值。
  - public void set(int field,int value)
  - public int get(int field)
  - public void add(int field,int amount)
  - public final Date getTime()
  - public final void setTime(Date date)
- 注意:
  - 获取月份时：一月是 0，二月是 1，以此类推，12 月是 11
  - 获取星期时：周日是 1，周二是 2， 。。。。周六是 7
- 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);
    }

    @Test
    public void test3(){
        Calendar calendar = Calendar.getInstance();
            // 从一个 Calendar 对象中获取 Date 对象
            Date date = calendar.getTime();

            // 使用给定的 Date 设置此 Calendar 的时间
            date = new Date(234234235235L);
            calendar.setTime(date);
            calendar.set(Calendar.DAY_OF_MONTH, 8);
            System.out.println("当前时间日设置为 8 后,时间是:" + calendar.
getTime());

            calendar.add(Calendar.HOUR, 2);
            System.out.println("当前时间加 2 小时后,时间是:" + calendar.ge
tTime());

            calendar.add(Calendar.MONTH, -2);
            System.out.println("当前日期减 2 个月后,时间是:" + calendar.ge
tTime());
    }
}
```

3.5 练习

输入年份和月份，输出该月日历。

闰年计算公式：年份可以被 4 整除但不能被 100 整除，或者可以被 400 整除。

```
请输入年份：2016
请输入月份：6
日      一      二      三      四      五      六
                        1       2       3       4
5       6       7       8       9       10      11
12      13      14      15      16      17      18
19      20      21      22      23      24      25
26      27      28      29      30
```

image-20220503120722810

4. JDK8：新的日期时间 API

如果我们可以跟别人说："我们在 1502643933071 见面，别晚了！"那么就再简单不过了。但是我们希望时间与昼夜和四季有关，于是事情就变复杂了。JDK 1.0 中包含了一个 java.util.Date 类，但是它的大多数方法已经在 JDK 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：像日期和时间这样的类应该是不可变的。
- 偏移性：Date 中的年份是从 1900 开始的，而月份都从 0 开始。
- 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。

  闰秒，是指为保持协调世界时接近于世界时时刻，由国际计量局统一规定在年底或年中（也可能在季末）对协调世界时增加或减少 1 秒的调整。由于地球自转的不均匀性和长期变慢性（主要由潮汐摩擦引起的），会使世界时（民用时）和原子时之间相差超过到±0.9 秒时，就把协调世界时向前拨 1 秒（负闰秒，最后一分钟为 59 秒）或向后拨 1 秒（正闰秒，最后一分钟为 61 秒）；闰秒一般加在公历年末或公历六月末。

  目前，全球已经进行了 27 次闰秒，均为正闰秒。

总结：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 以一个新的开始为 Java 创建优秀的 API。新的日期时间 API 包含：

- `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

说明：大多数开发者只会用到基础包和 format 包，也可能会用到 temporal 包。因此，尽管有 68 个新的公开类型，大多数开发者，大概将只会用到其中的三分之一。

比如，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。

4.1 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(xx,xx,xx,xx,xx,xxx)`	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

方法	描述
	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

```java
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
    }
```

```java
    @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);
    }
}
```

4.2 瞬时：Instant

- Instant：时间线上的一个瞬时点。 这可能被用来记录应用程序中的事件时间戳。

- 在处理时间和日期的时候，我们通常会想到年、月、日、时、分、秒。然而，这只是时间的一个模型，是面向人类的。第二种通用模型是面向机器的，或者说是连续的。在此模型中，时间线中的一个点表示为一个很大的数，这有利于计算机处理。在 UNIX 中，这个数从 1970 年开始，以秒为的单位；同样的，在 Java 中，也是从 1970 年开始，但以毫秒为单位。

- `java.time` 包通过值类型 `Instant` 提供机器视图，不提供处理人类意义上的时间单位。Instant 表示时间线上的一点，而不需要任何上下文信息，例如，时区。概念上讲，它只是简单的表示自 `1970` 年 `1` 月 `1` 日 `0` 时 `0` 分 `0` 秒（`UTC`）开始的秒数。因为 java.time 包是基于纳秒计算的，所以 Instant 的精度可以达到纳秒级。

- (1 ns = 10-9 s) 1 秒 = 1000 毫秒 =10^6 微秒=10^9 纳秒

方法	描述
`now()`	静态方法，返回默认 UTC 时区的 Instant 类的对象
`ofEpochMilli(long epochMilli)`	静态方法，返回在 1970-01-01 00:00:00 基础上加上指定毫秒数之后的 Instant 类的对象
atOffset(ZoneOffset offset)	结合即时的偏移来创建一个 OffsetDateTime
`toEpochMilli()`	返回 1970-01-01 00:00:00 到当前时间的毫秒数，即为时间戳

时间戳是指格林威治时间 1970 年 01 月 01 日 00 时 00 分 00 秒(北京时间 1970 年 01 月 01 日 08 时 00 分 00 秒)起至现在的总秒数。

中国大陆、中国香港、中国澳门、中国台湾、蒙古国、新加坡、马来西亚、菲律宾、西澳大利亚州的时间与 UTC 的时差均为+8，也就是 UTC+8。

instant.atOffset(ZoneOffset.ofHours(8));

尚硅谷

整个地球分为二十四时区，每个时区都有自己的本地时间。北京时区是东八区，领先 UTC 八个小时，在电子邮件信头的 Date 域记为+0800。如果在电子邮件的信头中有这么一行：

Date: Fri, 08 Nov 2002 09:42:22 +0800

说明信件的发送地的地方时间是二○○二年十一月八号，星期五，早上九点四十二分（二十二秒），这个地方的本地时领先 UTC 八个小时(+0800，就是东八区时间)。电子邮件信头的 Date 域使用二十四小时的时钟，而不使用 AM 和 PM 来标记上下午。

以这个电子邮件的发送时间为例，如果要把这个时间转化为 UTC，可以使用一下公式：

UTC + 时区差 ＝ 本地时间

时区差东为正，西为负。在此，把东八区时区差记为 +0800，

UTC + (＋0800) = 本地（北京）时间 (1)

那么，UTC = 本地时间（北京时间）- 0800 (2)

0942 - 0800 = 0142

即 UTC 是当天凌晨一点四十二分二十二秒。

4.3 日期时间格式化：DateTimeFormatter

该类提供了三种格式化方法：

- 预定义的标准格式。如：ISO*LOCAL*DATE*TIME*、*ISO*LOCAL*DATE*、*ISO*LOCAL_TIME
- 本地化相关的格式。如：ofLocalizedDate(FormatStyle.LONG)

```
// 本地化相关的格式。如：ofLocalizedDateTime()
// FormatStyle.LONG / FormatStyle.MEDIUM / FormatStyle.SHORT :适用于LocalDateTime

// 本地化相关的格式。如：ofLocalizedDate()
// FormatStyle.FULL / FormatStyle.LONG / FormatStyle.MEDIUM / FormatStyle.SHORT : 适用于LocalDate
```

- 自定义的格式。如：ofPattern("yyyy-MM-dd hh:mm:ss")

方 法	描 述
ofPattern(String pattern)	静态方法，返回一个指定字符串格式的DateTimeFormatter
format(TemporalAccessor t)	格式化一个日期、时间，返回字符串
parse(CharSequence text)	将指定格式的字符序列解析为一个日期、时间

举例：

```
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(){
        // 方式一：预定义的标准格式。如：ISO_LOCAL_DATE_TIME;ISO_LOCAL_DATE;ISO_LOCAL_TIME
            DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
            // 格式化:日期-->字符串
            LocalDateTime localDateTime = LocalDateTime.now();
            String str1 = formatter.format(localDateTime);
            System.out.println(localDateTime);
            System.out.println(str1);// 2019-02-18T15:42:18.797
```

```
            // 解析：字符串 -->日期
            TemporalAccessor parse = formatter.parse("2019-02-18T15:42:
18.797");
            System.out.println(parse);
    }

    @Test
    public void test2(){
        // 方式二：
            // 本地化相关的格式。如：ofLocalizedDateTime()
            // FormatStyle.LONG / FormatStyle.MEDIUM / FormatStyle.SHOR
T :适用于LocalDateTime
            DateTimeFormatter formatter1 = DateTimeFormatter.ofLocalize
dDateTime(FormatStyle.LONG);
            // 格式化
            String str2 = formatter1.format(localDateTime);
            System.out.println(str2);// 2019年2月18日 下午03时47分16
秒

            // 本地化相关的格式。如：ofLocalizedDate()
            // FormatStyle.FULL / FormatStyle.LONG / FormatStyle.MEDIUM
 / FormatStyle.SHORT : 适用于LocalDate
            DateTimeFormatter formatter2 = DateTimeFormatter.ofLocalize
dDate(FormatStyle.FULL);
            // 格式化
            String str3 = formatter2.format(LocalDate.now());
            System.out.println(str3);// 2020年3月14日 星期六
    }

    @Test
    public void test3(){
        //方式三：自定义的方式（关注、重点）
            DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofP
attern("yyyy/MM/dd hh:mm:ss");
            //格式化
           String strDateTime = dateTimeFormatter.format(LocalDateTime.
now());
            System.out.println(strDateTime);
            //解析
            TemporalAccessor accessor = dateTimeFormatter.parse("2020/0
3/14 10:45:24");
            System.out.println(accessor);
    }
}
```

4.4 其它 API

1、指定时区日期时间：ZondId 和 ZonedDateTime

- ZoneId：该类中包含了所有的时区信息，一个时区的 ID，如 Europe/Paris
- ZonedDateTime：一个在 ISO-8601 日历系统时区的日期时间，如 2007-12-03T10:15:30+01:00 Europe/Paris。
  - 其中每个时区都对应着 ID，地区 ID 都为“{区域}/{城市}”的格式，例如：Asia/Shanghai 等
- 常见时区 ID：

```
Asia/Shanghai
UTC
America/New_York
```

- 可以通过 ZondId 获取所有可用的时区 ID：

```java
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_Yor
k"));
        System.out.println(t2);
    }
}
```

2、持续日期/时间：Period 和 Duration

- 持续时间：Duration，用于计算两个“时间”间隔
- 日期间隔：Period，用于计算两个“日期”间隔

```java
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, 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());
    }
    @Test
    public void test03(){
        //Duration:用于计算两个“时间”间隔，以秒和纳秒为基准
            LocalTime localTime = LocalTime.now();
            LocalTime localTime1 = LocalTime.of(15, 23, 32);
            //between():静态方法，返回Duration 对象，表示两个时间的间隔
           Duration duration = Duration.between(localTime1, localTime);
            System.out.println(duration);
```

```
            System.out.println(duration.getSeconds());
            System.out.println(duration.getNano());

            LocalDateTime localDateTime = LocalDateTime.of(2016, 6, 12,
 15, 23, 32);
           LocalDateTime localDateTime1 = LocalDateTime.of(2017, 6, 12,
 15, 23, 32);

            Duration duration1 = Duration.between(localDateTime1, local
DateTime);
            System.out.println(duration1.toDays());
    }

    @Test
    public void test4(){
        //Period:用于计算两个“日期”间隔，以年、月、日衡量
            LocalDate localDate = LocalDate.now();
            LocalDate localDate1 = LocalDate.of(2028, 3, 18);

            Period period = Period.between(localDate, localDate1);
            System.out.println(period);

            System.out.println(period.getYears());
            System.out.println(period.getMonths());
            System.out.println(period.getDays());

            Period period1 = period.withYears(2);
            System.out.println(period1);

    }
}
```

3、Clock：使用时区提供对当前即时、日期和时间的访问的时钟。

4、

TemporalAdjuster : 时间校正器。有时我们可能需要获取例如：将日期调整到“下一个工作日”等操作。 TemporalAdjusters : 该类通过静态方法(firstDayOfXxx()/lastDayOfXxx()/nextXxx())提供了大量的常用 TemporalAdjuster 的实现。

```
@Test
public void test1(){
    // TemporalAdjuster:时间校正器
      // 获取当前日期的下一个周日是哪天？
      TemporalAdjuster temporalAdjuster = TemporalAdjusters.next(DayOfW
eek.SUNDAY);
      LocalDateTime localDateTime = LocalDateTime.now().with(temporalAd
```

```
juster);
        System.out.println(localDateTime);
        // 获取下一个工作日是哪天？
        LocalDate localDate = LocalDate.now().with(new TemporalAdjuster()
 {
                @Override
                public Temporal adjustInto(Temporal temporal) {
                LocalDate date = (LocalDate) temporal;
                if (date.getDayOfWeek().equals(DayOfWeek.FRIDAY)) {
                        return date.plusDays(3);
                } else if (date.getDayOfWeek().equals(DayOfWeek.SATURDAY))
{
                        return date.plusDays(2);
                } else {
                        return date.plusDays(1);
                }
        }
        });
        System.out.println("下一个工作日是: " + localDate);

}
```

4.5 与传统日期处理的转换

类	To 遗留类	From 遗留类
java.time.Instant 与 java.util.Date	Date.from(instant)	date.toInstant()
java.time.Instant 与 java.sql.Timestamp	Timestamp.from(instant)	timestamp.toInstant()
java.time.ZonedDateTime 与 java.util.GregorianCalendar	GregorianCalendar.from(zonedDateTime)	cal.toZonedDateTime()
java.time.LocalDate 与 java.sql.Time	Date.valueOf(localDate)	date.toLocalDate()
java.time.LocalTime 与 java.sql.Time	Date.valueOf(localDate)	date.toLocalTime()
java.time.LocalDateTime 与 java.sql.Timestamp	Timestamp.valueOf(localDateTime)	timestamp.toLocalDateTime()
java.time.ZoneId 与 java.util.TimeZone	Timezone.getTimeZone(id)	timeZone.toZoneId()
java.time.format.DateTimeFormatter 与	formatter.toFormat()	无

类	To 遗留类	From 遗留类
java.text.DateFormat		

5. Java 比较器

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？

- 在 Java 中经常会涉及到对象数组的排序问题，那么就涉及到对象之间的比较问题。
- Java 实现对象排序的方式有两种：
  - 自然排序：java.lang.Comparable
  - 定制排序：java.util.Comparator

5.1 自然排序：java.lang.Comparable

- Comparable 接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序。
- 实现 Comparable 的类必须实现 `compareTo(Object obj)`方法，两个对象即通过 compareTo(Object obj) 方法的返回值来比较大小。如果当前对象 this 大于形参对象 obj，则返回正整数，如果当前对象 this 小于形参对象 obj，则返回负整数，如果当前对象 this 等于形参对象 obj，则返回零。

```java
package java.lang;

public interface Comparable{
    int compareTo(Object obj);
}
```

- 实现 Comparable 接口的对象列表（和数组）可以通过 Collections.sort 或 Arrays.sort 进行自动排序。实现此接口的对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器。
- 对于类 C 的每一个 e1 和 e2 来说，当且仅当 e1.compareTo(e2) == 0 与 e1.equals(e2) 具有相同的 boolean 值时，类 C 的自然排序才叫做与 equals 一致。建议（虽然不是必需的）最好使自然排序与 `equals` 一致。
- Comparable 的典型实现：(默认都是从小到大排列的)
  - String：按照字符串中字符的 Unicode 值进行比较
  - Character：按照字符的 Unicode 值来进行比较
  - 数值类型对应的包装类以及 BigInteger、BigDecimal：按照它们对应的数值大小进行比较
  - Boolean：true 对应的包装类实例大于 false 对应的包装类实例
  - Date、Time 等：后面的日期时间比前面的日期时间大
- 代码示例：

```java
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) {
```

```java
        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Stude
nt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

```java
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]);
        }
    }
}
```

再举例：

```java
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;
    }
}
```

测试：

```
@Test
public void test02() {
      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));
}
```

再举例：

```
class Goods implements Comparable {
    private String name;
    private double price;

    //按照价格，比较商品的大小
```

```java
    @Override
    public int compareTo(Object o) {
        if(o instanceof Goods) {
            Goods other = (Goods) o;
            if (this.price > other.price) {
                return 1;
            } else if (this.price < other.price) {
                return -1;
            }
            return 0;
        }
        throw new RuntimeException("输入的数据类型不一致");
    }
    //构造器、getter、setter、toString()方法略
}
```

测试：

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

        Goods[] all = new Goods[4];
        all[0] = new Goods("《红楼梦》", 100);
        all[1] = new Goods("《西游记》", 80);
        all[2] = new Goods("《三国演义》", 140);
        all[3] = new Goods("《水浒传》", 120);

        Arrays.sort(all);

        System.out.println(Arrays.toString(all));

    }

}
```

5.2 定制排序：java.util.Comparator

- 思考？
  - 当元素的类型没有实现 java.lang.Comparable 接口而又不方便修改代码（例如：一些第三方的类，你只有.class 文件，没有源文件）
  - 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。强行对多个对象进行整体排序的比较。

- 重写 compare(Object o1,Object o2)方法，比较 o1 和 o2 的大小：如果方法返回正整数，则表示 o1 大于 o2；如果返回 0，表示相等；返回负整数，表示 o1 小于 o2。 可以将 Comparator 传递给 sort 方法（如 Collections.sort 或 Arrays.sort），从而允许在排序顺序上实现精确控制。

```
package java.util;

public interface Comparator{
    int compare(Object o1,Object o2);
}
```

- 还可以使用 Comparator 来控制某些数据结构（如有序 set 或有序映射）的顺序，或者为那些没有自然顺序的对象 collection 提供排序。

举例：

```
package com.atguigu.api;

import java.util.Comparator;
//定义定制比较器类
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();
    }
}
```

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]));
```

```java
        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]);
        }
    }
}
```

再举例:

```java
@Test
public void test02() {
      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));
      //定制排序
```

```
        Arrays.sort(students, new Comparator() {
                @Override
                public int compare(Object o1, Object o2) {
                        return Collator.getInstance(Locale.CHINA).compare(((S
tudent) o1).getName(), ((Student) o2).getName());
                }
        });
        System.out.println(Arrays.toString(students));
}
```

再举例：

```
Goods[] all = new Goods[4];
all[0] = new Goods("War and Peace", 100);
all[1] = new Goods("Childhood", 80);
all[2] = new Goods("Scarlet and Black", 140);
all[3] = new Goods("Notre Dame de Paris", 120);

Arrays.sort(all, new Comparator() {

    @Override
    public int compare(Object o1, Object o2) {
        Goods g1 = (Goods) o1;
        Goods g2 = (Goods) o2;

        return g1.getName().compareTo(g2.getName());
    }
});

System.out.println(Arrays.toString(all));
```

6. 系统相关类

6.1 java.lang.System 类

- System 类代表系统，系统级的很多属性和控制方法都放置在该类的内部。该类位于 `java.lang` 包。
- 由于该类的构造器是 private 的，所以无法创建该类的对象，也就是无法实例化该类。其内部的成员变量和成员方法都是 `static` 的，所以也可以很方便的进行调用。
- 成员变量 Scanner scan = new Scanner(System.in);
  - System 类内部包含 `in`、`out` 和 `err` 三个成员变量，分别代表标准输入流(键盘输入)，标准输出流(显示器)和标准错误输出流(显示器)。
- 成员方法

- native long currentTimeMillis()：该方法的作用是返回当前的计算机时间，时间的表达格式为当前计算机时间和 GMT 时间(格林威治时间)1970 年 1 月 1 号 0 时 0 分 0 秒所差的毫秒数。

- void exit(int status)：该方法的作用是退出程序。其中 status 的值为 0 代表正常退出，非零代表异常退出。使用该方法可以在图形界面编程中实现程序的退出功能等。

- void gc()：该方法的作用是请求系统进行垃圾回收。至于系统是否立刻回收，则取决于系统中垃圾回收算法的实现以及系统执行时的情况。

- String getProperty(String key)：该方法的作用是获得系统中属性名为 key 的属性对应的值。系统中常见的属性名以及属性的作用如下表所示：

属性名	属性说明
java.version	Java 运行时环境版本
java.home	Java 安装目录
os.name	操作系统的名称
os.version	操作系统的版本
user.name	用户的账户名称
user.home	用户的主目录
user.dir	用户的当前工作目录

- 举例

```
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：" + tim
e + "毫秒");

        System.exit(0);

        System.out.println("over");//不会执行
    }
```

```java
    @Test
    public void test02(){
        String javaVersion = System.getProperty("java.version");
            System.out.println("java 的 version:" + javaVersion);

            String javaHome = System.getProperty("java.home");
            System.out.println("java 的 home:" + javaHome);

            String osName = System.getProperty("os.name");
            System.out.println("os 的 name:" + osName);

            String osVersion = System.getProperty("os.version");
            System.out.println("os 的 version:" + osVersion);

            String userName = System.getProperty("user.name");
            System.out.println("user 的 name:" + userName);

            String userHome = System.getProperty("user.home");
            System.out.println("user 的 home:" + userHome);

            String userDir = System.getProperty("user.dir");
            System.out.println("user 的 dir:" + userDir);
    }

    @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;
    }
```

```java
    @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+ "轻轻的我走了，不带走一段代码....");
    }
}
```

- 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

```java
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));
    }
}
```

6.2 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 = "";
        //模拟占用内存
        for (int i = 0; i < 10000; 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));
    }
}
```

7. 和数学相关的类

7.1 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
- 其它：acos,asin,atan,cos,sin,tan 三角函数

```
double result = Math.pow(2,31);
double sqrt = Math.sqrt(256);
double rand = Math.random();
double pi = Math.PI;
```

7.2 java.math 包

7.2.1 BigInteger

- Integer 类作为 int 的包装类，能存储的最大整型值为 2^31-1，Long 类也是有限的，最大为 2^63-1。如果要表示再大的整数，不管是基本数据类型还是他们的包装类都无能为力，更不用说进行运算了。
- java.math 包的 BigInteger 可以表示不可变的任意精度的整数。BigInteger 提供所有 Java 的基本整数操作符的对应物，并提供 java.lang.Math 的所有相关方法。另外，BigInteger 还提供以下运算：模算术、GCD 计算、质数测试、素数生成、位操作以及一些其他操作。
- 构造器

- BigInteger(String val)：根据字符串构建 BigInteger 对象

- 方法
  - public BigInteger **abs**()：返回此 BigInteger 的绝对值的 BigInteger。
  - BigInteger **add**(BigInteger val)：返回其值为 (this + val) 的 BigInteger
  - BigInteger **subtract**(BigInteger val)：返回其值为 (this - val) 的 BigInteger
  - BigInteger **multiply**(BigInteger val)：返回其值为 (this * val) 的 BigInteger
  - BigInteger **divide**(BigInteger val)：返回其值为 (this / val) 的 BigInteger。整数相除只保留整数部分。
  - BigInteger **remainder**(BigInteger val)：返回其值为 (this % val) 的 BigInteger。
  - BigInteger[] **divideAndRemainder**(BigInteger val)：返回包含 (this / val) 后跟 (this % val) 的两个 BigInteger 的数组。
  - BigInteger **pow**(int exponent)：返回其值为 (this^exponent) 的 BigInteger。

```
    @Test
    public void test01(){
//          long bigNum = 1234567891234567891234567 89L;

            BigInteger b1 = new BigInteger("1234567891234567891234567 89");
            BigInteger b2 = new BigInteger("7892345678912345678912345 67 89");

//          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));
    }
```

7.2.2 RoundingMode 枚举类

CEILING：向正无限大方向舍入的舍入模式。

FLOOR：向负无限大方向舍入的舍入模式。

DOWN：向零方向舍入的舍入模式。

UP：远离零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_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。

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。

7.2.3 BigDecimal

- 一般的 Float 类和 Double 类可以用来做科学计算或工程计算，但在**商业计算中，要求数字精度比较高，故用到 java.math.BigDecimal 类。**
- BigDecimal 类支持不可变的、任意精度的有符号十进制定点数。
- 构造器
  - public BigDecimal(double val)
  - public BigDecimal(String val)
- 常用方法
  - public BigDecimal **add**(BigDecimal augend)
  - public BigDecimal **subtract**(BigDecimal subtrahend)
  - public BigDecimal **multiply**(BigDecimal multiplicand)
  - public BigDecimal **divide**(BigDecimal divisor, int scale, int roundingMode)
- 举例：

```
@Test
public void test02(){
      /*double big = 12.123456789123456789123456789;
      System.out.println("big = " + big);*/

      BigDecimal b1 = new BigDecimal("123.45678912345678912345678
```

```
912345678");
            BigDecimal b2 = new BigDecimal("7.892345678912345678912345
789998898888");

//          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.U
P));//divide(BigDecimal divisor, int scale, int roundingMode)
            System.out.println("除: " + b1.divide(b2,20,RoundingMode.DO
WN));//divide(BigDecimal divisor, int scale, int roundingMode)
            System.out.println("余: " + b1.remainder(b2));
      }
```

- 举例

```
public void testBigInteger() {
    BigInteger bi = new BigInteger("12433241123");
    BigDecimal bd = new BigDecimal("12435.351");
    BigDecimal bd2 = new BigDecimal("11");
    System.out.println(bi);
    // System.out.println(bd.divide(bd2));
    System.out.println(bd.divide(bd2, BigDecimal.ROUND_HALF_UP));
    System.out.println(bd.divide(bd2, 15, BigDecimal.ROUND_HALF_UP));
}
```

7.3 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());
}
```

第 12 章_集合框架

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 集合框架概述

1.1 容器：数组、集合

- **生活中的容器**

image-20220523190743146

尚硅谷

- 一方面，面向对象语言对事物的体现都是以对象的形式，为了方便对多个对象的操作，就要对对象进行存储。另一方面，使用 Array 存储对象方面具有一些弊端，而 Java 集合就像一种容器，可以动态地把多个对象的引用放入容器中。
- 数组在内存存储方面的特点：
  - 数组初始化以后，长度就确定了。
  - 数组声明的类型，就决定了进行元素初始化时的类型
  - 可以存储基本数据类型值，也可以存储对象
- 数组在存储数据方面的弊端：
  - 数组初始化以后，长度就不可变了，不便于扩展
  - 数组中提供的属性和方法少，不便于进行添加、删除、插入等操作，且效率不高。同时无法直接获取存储元素的个数
  - 数组存储的数据是有序的、可以重复的。---->存储数据的特点单一
- Java 集合类可以用于存储数量不等的多个对象，还可用于保存具有映射关系的关联数组。

1.2 集合的使用场景

image-20220407203003545

1.3 Java 集合框架体系

- Java 集合可分为 Collection 和 Map 两种体系：
  - Collection 接口：单列数据，定义了存取一组对象的方法的集合
    - List：元素有序、可重复的集合
    - Set：元素无序、不可重复的集合
  - Map 接口：双列数据，保存具有映射关系“key-value 对”的集合
- JDK 提供的集合 API 位于 java.util 包内
- 图示：集合框架全图

- 简图 1：**Collection 接口继承树**

image-20220407203244029

- 简图 2：**Map 接口继承树**

image-20220407203412665

2. Collection 接口及方法

- Collection 接口是 List、Set 和 Queue 接口的父接口，该接口里定义的方法既可用于操作 Set 集合，也可用于操作 List 和 Queue 集合。

- JDK 不提供此接口的任何直接实现，而是提供更具体的子接口(如：Set 和 List)实现。
- 在 Java5 之前，Java 集合会丢失容器中所有对象的数据类型，把所有对象都当成 Object 类型处理；从 JDK 5.0 增加了泛型以后，Java 集合可以记住容器中对象的数据类型。

Collection<E>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：

2.1 添加

（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

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);

2.2 判断

（3）**boolean isEmpty()**：判断当前集合是否为空集合。

（4）**int size()**：获取当前集合中实际存储的元素个数

（5）**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。

（6）**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”。

（7）**boolean equals(Object obj)**：判断当前集合与 obj 是否相等

演示：

```
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.isEmp
ty());
        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());

        System.out.println("coll 在添加元素之后，isEmpty = " + coll.isEmp
ty());
        coll.clear();
        System.out.println("coll 在 clear 之后，isEmpty = " + coll.isEmpt
y());
    }

    @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.contains
All(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(othe
r));
        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(othe
r));
        System.out.println("c2.containsAll(other) = " + c2.containsAll
(other));
    }
```

}

2.3 删除

（8）**void clear()**：清空集合元素

（9） **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。

（10）**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

（11）**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；

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.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 = " + othe
r);
    }

    @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 = " + othe
r);
    }

}
```

2.4 其它

（12）**Object[] toArray()**：返回包含当前集合中所有元素的数组

（13）**hashCode()**：获取集合对象的哈希值

（14）**iterator()**：返回迭代器对象，用于集合遍历

```java
public class TestCollectionContains {
    @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");

        Object[] objects = coll.toArray();
        System.out.println("用数组返回 coll 中所有元素： " + Arrays.toStrin
g(objects));

    }
}
```

3. Iterator(迭代器)接口

3.1 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。

Iterator 仅用于遍历集合，Iterator 本身并不提供承装对象的能力。如果需要创建 Iterator 对象，则必须有一个被迭代的集合。

Collection 接口继承了 java.lang.Iterable 接口，该接口有一个 iterator()方法，那么所有实现了 Collection 接口的集合类都有一个 iterator()方法，用以返回一个实现了 Iterator 接口的对象。

- `public Iterator iterator()`: 获取集合对应的迭代器，用来遍历集合中的元素的。
- 集合对象每次调用 iterator()方法都得到一个全新的迭代器对象，默认游标都在集合的第一个元素之前。

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。

注意：在调用 it.next()方法之前必须要调用 it.hasNext()进行检测。若不调用，且下一条记录无效，直接调用 it.next()会抛出 NoSuchElementException 异常。

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
        }
    }
}
```

3.2 迭代器的执行原理

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 hashNext()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：

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。

3.3 使用 Iterator 迭代器删除元素

java.util.Iterator 迭代器中有一个方法：void remove() ;

```
Iterator iter = coll.iterator();//回到起点
while(iter.hasNext()){
    Object obj = iter.next();
    if(obj.equals("Tom")){
        iter.remove();
    }
}
```

- 注意：
  - Iterator 可以删除集合的元素，但是是遍历过程中通过迭代器对象的 remove 方法，不是集合对象的 remove 方法。
  - 如果还未调用 next()或在上一次调用 next 方法之后已经调用了 remove 方法，再调用 remove 都会报 IllegalStateException。

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？因为迭代器的 remove()可以按指定的条件进行删除。

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);
    }
}
```

在 JDK8.0 时，Collection 接口有了 removeIf 方法，即可以根据条件删除。

```
package com.atguigu.collection;

import org.junit.Test;
```

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.function.Predicate;

public class TestCollectionRemoveIf {
    @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.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);
    }
}
```

3.4 Iterable 接口与增强 for 循环

增强 for 循环(也称 for each 循环)是 JDK5.0 中定义的一个高级 for 循环，专门用来遍历数组和集合的。它的内部原理其实是个 Iterator 迭代器。

foreach 循环的语法格式：

```
for(元素的数据类型  元素名 : Collection 集合 or 数组){
      //写操作代码
}
//这里元素名就是一个临时变量，自己命名就可以
```

代码示例：

```java
package com.atguigu.iterator;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Collection;

public class TestForeach {
    @Test
    public void test01(){
```

```java
        Collection coll = new ArrayList();
        coll.add("小李广");
        coll.add("扫地僧");
        coll.add("石破天");
            //foreach 循环其实就是使用Iterator 迭代器来完成元素的遍历的。
        for (Object o : coll) {
            System.out.println(o);
        }
    }
    @Test
    public void test02(){
        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);
        }
    }
}
```

说明：

它用于遍历 Collection 和数组。通常只进行遍历元素，不要在遍历的过程中对集合元素进行增删操作。

练习：判断输出结果为何？

```java
public class ForTest {
    public static void main(String[] args) {
        String[] str = new String[5];
        for (String myStr : str) {
            myStr = "atguigu";
            System.out.println(myStr);
        }
        for (int i = 0; i < str.length; i++) {
            System.out.println(str[i]);
        }
    }
}
```

3.5 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

```java
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remo
ve 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 implemen
tation
     * returned by the {@code iterator} and {@code listIterator} method
s.
     * If the value of this field changes unexpectedly, the iterator (o
r list
     * iterator) will throw a {@code ConcurrentModificationException} i
n
```

```
     * 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 behavio
r in
     * the face of concurrent modification during iteration.
     *
     * <p><b>Use of this field by subclasses is optional.</b> If a subc
lass
     * wishes to provide fail-fast iterators (and list iterators), then
 it
     * merely has to increment this field in its {@code add(int, E)} an
d
     * {@code remove(int)} methods (and any other methods that it overr
ides
     * 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 th
row
     * bogus {@code ConcurrentModificationExceptions}.  If an implement
ation
     * 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，expectedModCou
nt 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
            }
        }
    }
}
```

4. Collection 子接口 1：List

4.1 List 接口特点

- 鉴于 Java 中数组用来存储数据的局限性，我们通常使用 `java.util.List` 替代数组
- List 集合类中元素有序、且可重复，集合中的每个元素都有其对应的顺序索引。
- List 容器中的元素都对应一个整数型的序号记载其在容器中的位置，可以根据序号存取容器中的元素。

举例：List 集合存储数据，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”。

- JDK API 中 List 接口的实现类常用的有：`ArrayList`、`LinkedList` 和 `Vector`。

4.2 List 接口方法

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。

- 添加元素
  - `void add(int index, Object ele)`:在 index 位置插入 ele 元素

- boolean addAll(int index, Collection eles):从 index 位置开始将 eles 中的所有元素添加进来

- 获取元素
  - `Object get(int index)`:获取指定 index 位置的元素
  - List subList(int fromIndex, int toIndex):返回从 fromIndex 到 toIndex 位置的子集合
- 获取元素索引
  - int indexOf(Object obj):返回 obj 在集合中首次出现的位置
  - int lastIndexOf(Object obj):返回 obj 在当前集合中末次出现的位置
- 删除和替换元素
  - `Object remove(int index)`:移除指定 index 位置的元素，并返回此元素
  - `Object set(int index, Object ele)`:设置指定 index 位置的元素为 ele

举例：

```
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
```

```java
        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。

4.2 List 接口主要实现类：ArrayList

- ArrayList 是 List 接口的典型实现类、主要实现类
- 本质上，ArrayList 是对象引用的一个"变长"数组
- Arrays.asList(...) 方法返回的 List 集合，既不是 ArrayList 实例，也不是 Vector 实例。 Arrays.asList(...) 返回值是一个固定长度的 List 集合

4.3 List 的实现类之二：LinkedList

- 对于频繁的插入或删除元素的操作，建议使用 LinkedList 类，效率较高。

- 新增方法：
  - void addFirst(Object obj)
  - void addLast(Object obj)
  - Object getFirst()
  - Object getLast()
  - Object removeFirst()
  - Object removeLast()

4.4 List 的实现类之三：Vector

- Vector 是一个古老的集合，JDK1.0 就有了。大多数操作与 ArrayList 相同，区别之处在于 Vector 是线程安全的。
- 在各种 list 中，最好把 `ArrayList` 作为缺省选择。当插入、删除频繁时，使用 LinkedList；Vector 总是比 ArrayList 慢，所以尽量避免使用。
- 新增方法：
  - void addElement(Object obj)
  - void insertElementAt(Object obj,int index)
  - void setElementAt(Object obj,int index)

- void removeElement(Object obj)
- void removeAllElements()

4.5 练习

面试题：

```
@Test
public void testListRemove() {
    List list = new ArrayList();
    list.add(1);
    list.add(2);
    list.add(3);
    updateList(list);
    System.out.println(list);//[1,2]
}

private static void updateList(List list) {
    list.remove(2);
}
```

练习 1：

- 键盘录入学生信息，保存到集合中。
  - 循环录入的方式，1：表示继续录入，0：表示结束录入。
  - 定义学生类，属性为姓名，年龄，使用学生对象保存录入数据。
  - 使用 ArrayList 集合，保存学生对象，录入结束后，用 foreach 遍历集合。
- 代码实现，效果如图所示：

```
选择（1、录入；0、退出）：1
姓名：张三
年龄：23
选择（1、录入；0、退出）：1
姓名：李四
年龄：24
选择（1、录入；0、退出）：0
Student [name=张三, age=23]
Student [name=李四, age=24]
```

```java
package com.atguigu.test02;

import java.util.ArrayList;
import java.util.Scanner;

public class StudentTest {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        ArrayList stuList = new ArrayList();

        for (;;) {

            System.out.println("选择（录入 1 ；结束 0）");
            int x = scanner.nextInt();//根据 x 的值，判断是否需要继续循环

            if (x == 1) {
                System.out.println("姓名");
                String name = scanner.next();
                System.out.println("年龄");
                int age = scanner.nextInt();
                Student stu = new Student(age, name);
                stuList.add(stu);

            } else if (x == 0) {
                break;

            } else {
```

```
                System.out.println("输入有误，请重新输入");
            }
        }

        for (Object stu : stuList) {
            System.out.println(stu);
        }
    }
}

public class Student {

    private int age;
    private String name;

    public Student() {
    }


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


      public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

    }

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

}

练习 2：

1、请定义方法 public static int listTest(Collection list,String s)统计集合中指定元素出现的次数

2、创建集合，集合存放随机生成的 30 个小写字母

3、用 listTest 统计，a、b、c、x 元素的出现次数

4、效果如下

```
[y, v, i, r, i, k, g, s, g, v, c, k, g, j, g, x, w, h, s, q, r, f, k, c, b, n, e, t, b, k]
a:0
b:2
c:2
x:1
```

```java
package com.atguigu.test07;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Random;

public class Test02 {
	public static void main(String[] args) {
		Collection list = new ArrayList();
		Random rand = new Random();
		for (int i = 0; i < 30; i++) {
			list.add((char)(rand.nextInt(26)+97)+"");
		}
		System.out.println(list);
		System.out.println("a:"+listTest(list, "a"));
		System.out.println("b:"+listTest(list, "b"));
		System.out.println("c:"+listTest(list, "c"));
		System.out.println("x:"+listTest(list, "x"));
	}

	public static int listTest(Collection list, String string) {
		int count = 0;
		for (Object object : list) {
			if(string.equals(object)){
				count++;
			}
		}
		return count;
	}
}
```

练习 3：KTV 点歌系统

任务描述

分别使用 LinkedList 和 ArrayList 集合，实现编写一个**模拟 KTV 点歌系统**的程序。在程序中，指令 1 代表添加歌曲，指令 2 代表将所选歌曲置顶，指令 3 代表将所选歌曲提前一位，指令 4 代表退出该系统。要求根据用户输入的指令和歌曲名展现歌曲列表。例如输入指令 1，输入歌曲名"菊花台"，则输出“当前歌曲列表：[菊花台]”。

任务目标

- 学会分析**模拟 KTV 点歌系统**程序设计的实现思路
- 根据思路独立完成**模拟 KTV 点歌系统**程序设计的源代码编写、编译及运行
- 掌握 ArrayList 集合和 LinkedList 集合中方法的使用

实现思路

- 为了指引用户操作，首先要经各个指令所表示的含义打印到控制台
- 程序中需要创建一个集合作为歌曲列表，并向其添加一部分歌曲
- 由于控制台需要实时等待用户输入命令，所以可以使用 while(true)来使程序一直处于等待用户输入指令的状态。
- 可以通过 Scanner 类的 nextInt()方法来接受控制台的信息
- 可以使用 switch 语句，根据控制台传入的指令，判断应该执行的操作
- 通过 LinkedList 集合或 ArrayList 集合定义的方法操作歌曲列表

实现代码

- 使用 LinkedList 集合模拟点歌系统的实现代码，如下所示：

```
import java.util.LinkedList;
import java.util.Scanner;

public class KTVByLinkedList {
      private static LinkedList<String> musicList = new LinkedLis
t<>();// 创建歌曲列表
      private static Scanner sc = new Scanner(System.in);

      public static void main(String[] args) {
            System.out.println("-------------欢迎来到点歌系统------
------");
            System.out.println("1.添加歌曲至列表");
```

```
            System.out.println("2.将歌曲置顶");
            System.out.println("3.将歌曲前移一位");
            System.out.println("4.退出");

            addMusicList();// 添加一部分歌曲至歌曲列表
            System.out.println("初始歌曲列表: " + musicList);
            boolean flag = true;
            while (flag) {
                  System.out.print("请输入要执行的操作序号: ");
                  int key = sc.nextInt();// //接收键盘输入的功能选
项序号
                  // 执行序号对应的功能
                  switch (key) {
                  case 1:// 添加歌曲至列表
                        addMusic();
                        break;
                  case 2:// 将歌曲置顶
                        setTop();
                        break;
                  case 3:// 将歌曲前移一位
                        setBefore();
                        break;
                  case 4:// 退出
                        flag = exit();
                        break;
                  default:
                        System.out.println("----------------------
-------------");
                        System.out.println("功能选择有误，请输入正
确的功能序号!");
                        break;
                  }
                  System.out.println("当前歌曲列表: " + musicList);
            }
      }

      // 初始时添加歌曲名称
      private static void addMusicList() {
            musicList.add("菊花台");
            musicList.add("江南 style");
            musicList.add("学猫叫");
            musicList.add("双节棍");
            musicList.add("突然的自我");
      }

      // 执行添加歌曲
      private static void addMusic() {
```

```
            System.out.print("请输入要添加的歌曲名称：");
            String musicName = sc.next();// 获取键盘输入内容
            musicList.addLast(musicName);// 添加歌曲到列表的最后
            System.out.println("已添加歌曲：" + musicName);
        }

        // 执行将歌曲置顶
        private static void setTop() {
            System.out.print("请输入要置顶的歌曲名称：");
            String musicName = sc.next();// 获取键盘输入内容
            int musicIndex = musicList.indexOf(musicName);// 查找
指定歌曲位置
            if (musicIndex < 0) {// 判断输入歌曲是否存在
                System.out.println("当前列表中没有输入的歌曲！");
            } else {
                musicList.remove(musicName);// 移除指定的歌曲
                musicList.addFirst(musicName);// 将指定的歌曲放到
第一位
            }
            System.out.println("已将歌曲" + musicName + "置顶");
        }

        // 执行将歌曲置前一位
        private static void setBefore() {
            System.out.print("请输入要置前的歌曲名称：");
            String musicName = sc.next();// 获取键盘输入内容
            int musicIndex = musicList.indexOf(musicName);// 查找
指定歌曲位置
            if (musicIndex < 0) {// 判断输入歌曲是否存在
                System.out.println("当前列表中没有输入的歌曲！");
            } else if (musicIndex == 0) {// 判断歌曲是否已在第一位
                System.out.println("当前歌曲已在最顶部！");
            } else {
                musicList.remove(musicName);// 移除指定的歌曲
                musicList.add(musicIndex - 1, musicName);// 将
指定的歌曲放到前一位
            }
           System.out.println("已将歌曲" + musicName + "置前一位");
        }

        // 退出
        private static boolean exit() {
            System.out.println("----------------退出-------------
--");
            System.out.println("您已退出系统");
            return false;
```

```
        }
    }
```

- 使用 ArrayList 集合模拟点歌系统的实现代码，如下所示：

```
import java.util.ArrayList;
import java.util.Scanner;

public class KTVByArrayList {
    private static ArrayList<String> musicList = new ArrayList<>
();// 创建歌曲列表
    private static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.println("-------------欢迎来到点歌系统------
------");
        System.out.println("1.添加歌曲至列表");
        System.out.println("2.将歌曲置顶");
        System.out.println("3.将歌曲前移一位");
        System.out.println("4.退出");

        addMusicList();// 添加一部分歌曲至歌曲列表
        System.out.println("初始歌曲列表: " + musicList);
        boolean flag = true;
        while (flag) {
            System.out.print("请输入要执行的操作序号: ");
            int key = sc.nextInt();// //接收键盘输入的功能选
项序号
            // 执行序号对应的功能
            switch (key) {
            case 1:// 添加歌曲至列表
                addMusic();
                break;
            case 2:// 将歌曲置顶
                setTop();
                break;
            case 3:// 将歌曲前移一位
                setBefore();
                break;
            case 4:// 退出
                flag = exit();
                break;
            default:
                System.out.println("-------------------
-------------");
                System.out.println("功能选择有误，请输入正
确的功能序号!");
                break;
```

```
                }
                System.out.println("当前歌曲列表: " + musicList);
            }
        }

        // 初始时添加歌曲名称
        private static void addMusicList() {
            musicList.add("菊花台");
            musicList.add("江南 style");
            musicList.add("学猫叫");
            musicList.add("双节棍");
            musicList.add("突然的自我");
        }

        // 执行添加歌曲
        private static void addMusic() {
            System.out.print("请输入要添加的歌曲名称: ");
            String musicName = sc.next();// 获取键盘输入内容
            musicList.add(musicName);// 添加歌曲到列表的最后
            System.out.println("已添加歌曲: " + musicName);
        }

        // 执行将歌曲置顶
        private static void setTop() {
            System.out.print("请输入要置顶的歌曲名称: ");
            String musicName = sc.next();// 获取键盘输入内容
            int musicIndex = musicList.indexOf(musicName);// 查找
指定歌曲位置
            if (musicIndex < 0) {// 判断输入歌曲是否存在
                System.out.println("当前列表中没有输入的歌曲! ");
            } else {
                musicList.remove(musicName);// 移除指定的歌曲
                musicList.add(0,musicName);// 将指定的歌曲放到第
一位
            }
            System.out.println("已将歌曲" + musicName + "置顶");
        }

        // 执行将歌曲置前一位
        private static void setBefore() {
            System.out.print("请输入要置前的歌曲名称: ");
            String musicName = sc.next();// 获取键盘输入内容
            int musicIndex = musicList.indexOf(musicName);// 查找
指定歌曲位置
            if (musicIndex < 0) {// 判断输入歌曲是否存在
                System.out.println("当前列表中没有输入的歌曲! ");
```

```
            } else if (musicIndex == 0) {// 判断歌曲是否已在第一位
                System.out.println("当前歌曲已在最顶部！");
            } else {
                musicList.remove(musicName);// 移除指定的歌曲
                musicList.add(musicIndex - 1, musicName);// 将指定的歌曲放到前一位
            }
        System.out.println("已将歌曲" + musicName + "置前一位");
    }

    // 退出
    private static boolean exit() {
        System.out.println("----------------退出----------------");
        System.out.println("您已退出系统");
        return false;
    }
}
```

5. Collection 子接口 2：Set

5.1 Set 接口概述

- Set 接口是 Collection 的子接口，set 接口没有提供额外的方法
- 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 判断两个对象是否相同不是使用 == 运算符，而是根据 equals() 方法
- 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。

5.2 Set 主要实现类：HashSet

5.2.1 HashSet 概述

- 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。
- HashSet 按 Hash 算法来存储集合中的元素，因此具有很好的存取、查找、删除性能。
- HashSet 具有以下特点：
  - 不能保证元素的排列顺序
  - HashSet 不是线程安全的

- 集合元素可以是 null

- 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。

- 对于存放在 Set 容器中的对象，**对应的类一定要重写 equals()和 hashCode(Object obj)方法**，以实现对象相等规则。即："相等的对象必须具有相等的散列码"。

5.2.2 HashSet 中添加元素的过程：

- 第 1 步：当向 HashSet 集合中存入一个元素时，HashSet 会调用该对象的 hashCode() 方法来得到该对象的 hashCode 值，然后根据 hashCode 值，通过某种散列函数决定该对象在 HashSet 底层数组中的存储位置。
  - （这个散列函数会与底层数组的长度相计算得到在数组中的下标，并且这种散列函数计算还尽可能保证能均匀存储元素，越是散列分布，该散列函数设计的越好）

- 第 2 步：
  - 如果两个元素的 hashCode()值不相等，则添加成功；
  - 如果两个元素的 hashCode()值相等：
    - 会继续调用 equals()方法，如果 equals()方法结果为 true，添加失败；
    - 如果 equals()方法结果为 false，那么会保存该元素，但是该数组的位置已经有元素了，那么会通过链表的方式继续链接。

举例：

```
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
```

```java
    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 +
                '}';
    }
}
```

```java
package com.atguigu.set;

import org.junit.Test;

import java.util.HashSet;

public class TestHashSet {
    @Test
    public void test01(){
        HashSet 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 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
    }
}
```

5.2.3 重写 hashCode() 方法的基本原则

- 在程序运行时，同一个对象多次调用 hashCode() 方法应该返回相同的值。
- 当两个对象的 equals() 方法比较返回 true 时，这两个对象的 hashCode() 方法的返回值也应相等。
- 对象中用作 equals() 方法比较的 Field，都应该用来计算 hashCode 值。
- 如果两个元素的 equals() 方法返回 true，但它们的 hashCode() 返回值不相等，hashSet 将会把它们存储在不同的位置，但依然可以添加成功。

5.2.4 重写 equals()方法的基本原则

以自定义的 Customer 类为例，何时需要重写 equals()？

重写 equals 方法的时候一般都需要同时复写 hashCode 方法。通常参与计算 hashCode 的对象的属性也应该参与到 equals()中进行计算。

5.2.5 Eclipse/IDEA 工具里 hashCode()的重写

以 Eclipse/IDEA 为例，在自定义类中可以调用工具自动重写 equals 和 hashCode。

问题：为什么用 Eclipse/IDEA 复写 hashCode 方法，有 31 这个数字？

选择系数的时候要选择尽量大的系数。因为如果计算出来的 hash 地址越大，所谓的“冲突”就越少，查找起来效率也会提高。（减少冲突）

并且 31 只占用 5bits,相乘造成数据溢出的概率较小。

31 可以 由 i*31== (i<<5)-1 来表示,现在很多虚拟机里面都有做相关优化。（提高算法效率）

31 是一个素数，素数作用就是如果我用一个数字来乘以这个素数，那么最终出来的结果只能被素数本身和被乘数还有 1 来整除！(减少冲突)

5.2.6 练习

练习 1：在 List 内去除重复数字值，要求尽量简单

```
public static List duplicateList(List list) {
      HashSet set = new HashSet();
```

```
        set.addAll(list);
        return new ArrayList(set);
}
public static void main(String[] args) {
        List list = new ArrayList();
        list.add(new Integer(1));
        list.add(new Integer(2));
        list.add(new Integer(2));
        list.add(new Integer(4));
        list.add(new Integer(4));
        List list2 = duplicateList(list);
        for (Object integer : list2) {
            System.out.println(integer);
        }
}
```

考虑到泛型以后，代码应当如下：

```
public static List<Integer> duplicateList(List<Integer> list) {
    HashSet<Integer> set = new HashSet<Integer>();
    set.addAll(list);
    return new ArrayList<Integer>(set);
}
public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(new Integer(1));
    list.add(new Integer(2));
    list.add(new Integer(2));
    list.add(new Integer(4));
    list.add(new Integer(4));
    List<Integer> list2 = duplicateList(list);
    for (Integer integer : list2) {
        System.out.println(integer);
    }
}
```

练习 2：获取随机数

编写一个程序，获取 10 个 1 至 20 的随机数，要求随机数不能重复。并把最终的随机数输出到控制台。

提示：考查 HashSet

```
/**
 *
 * @Description
 * @author 尚硅谷-宋红康  Email:shkstart@126.com
 * @version
 * @date 2021年5月7日上午12:43:01
 *
```

```
 */

public class RandomValueTest {
      public static void main(String[] args) {
            HashSet hs = new HashSet(); // 创建集合对象
            Random r = new Random();
            while (hs.size() < 10) {
                  int num = r.nextInt(20) + 1; // 生成1到20的随机数
                  hs.add(num);
            }

            for (Integer integer : hs) { // 遍历集合
                  System.out.println(integer); // 打印每一个元素
            }
      }
}
```

练习 3：去重

使用 Scanner 从键盘读取一行输入，去掉其中重复字符，打印出不同的那些字符。比如：aaaabbbcccddd

提示：考查 HashSet

```
/**
 *
 * @Description
 * @author 尚硅谷-宋红康  Email:shkstart@126.com
 * @version
 * @date 2019年5月7日上午12:43:01
 *
 */
public class DistinctTest {
      public static void main(String[] args) {
            Scanner sc = new Scanner(System.in); // 创建键盘录入对象
            System.out.println("请输入一行字符串:");
            String line = sc.nextLine(); // 将键盘录入的字符串存储在line
中
            char[] arr = line.toCharArray(); // 将字符串转换成字符数组
            HashSet hs = new HashSet(); // 创建HashSet集合对象

            for (Object c : arr) { // 遍历字符数组
                  hs.add(c); // 将字符数组中的字符添加到集合中
            }

            for (Object ch : hs) { // 遍历集合
                  System.out.print(ch);
            }
```

```
    }
}
```

练习 4：面试题

```
HashSet set = new HashSet();
Person p1 = new Person(1001,"AA");
Person p2 = new Person(1002,"BB");

set.add(p1);
set.add(p2);
p1.name = "CC";
set.remove(p1);
System.out.println(set);

set.add(new Person(1001,"CC"));
System.out.println(set);

set.add(new Person(1001,"AA"));
System.out.println(set);

//其中Person 类中重写了hashCode()和equal()方法
```

5.3 Set 实现类之二：LinkedHashSet

- LinkedHashSet 是 HashSet 的子类
- LinkedHashSet 根据元素的 hashCode 值来决定元素的存储位置，但它同时使用双向链表维护元素的次序，这使得元素看起来是以插入顺序保存的。
- 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。
- LinkedHashSet 不允许集合元素重复。

image-20220408235936404

举例：

```
package com.atguigu.set;

import org.junit.Test;

import java.util.LinkedHashSet;

public class TestLinkedHashSet {
    @Test
    public void test01(){
        LinkedHashSet 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
    }
}
```

5.4 Set 实现类之三：TreeSet

5.4.1 TreeSet 概述

- TreeSet 是 SortedSet 接口的实现类，TreeSet 可以确保集合元素处于排序状态。
- TreeSet 底层使用红黑树结构存储数据
- 新增的方法如下：（了解）
  - Comparator comparator()
  - Object first()
  - Object last()
  - Object lower(Object e)
  - Object higher(Object e)
  - SortedSet subSet(fromElement, toElement)
  - SortedSet headSet(toElement)
  - SortedSet tailSet(fromElement)
- TreeSet 特点：不允许重复、实现排序（自然排序或定制排序）
- 如何判断 TreeSet 中数据不重复？
  - 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
- TreeSet 两种排序方法：自然排序和定制排序。默认情况下，TreeSet 采用自然排序。
  - 自然排序：TreeSet 会调用集合元素的 compareTo(Object obj) 方法来比较元素之间的大小关系，然后将集合元素按升序(默认情况)排列。
    - 如果试图把一个对象添加到 TreeSet 时，则该对象的类必须实现 Comparable 接口。
    - 实现 Comparable 的类必须实现 compareTo(Object obj) 方法，两个对象即通过 compareTo(Object obj) 方法的返回值来比较大小。
  - 定制排序：如果元素所属的类没有实现 Comparable 接口，或不希望按照升序(默认情况)的方式排列元素或希望按照其它属性大小进行排

序，则考虑使用定制排序。定制排序，通过 Comparator 接口来实现。需要重写 compare(T o1,T o2)方法。

- 利用 int compare(T o1,T o2)方法，比较 o1 和 o2 的大小：如果方法返回正整数，则表示 o1 大于 o2；如果返回 0，表示相等；返回负整数，表示 o1 小于 o2。
- 要实现定制排序，需要将实现 Comparator 接口的实例作为形参传递给 TreeSet 的构造器。
- 使用定制排序判断两个元素相等的标准是：通过 Comparator 比较两个元素返回了 0。

- 因为只有相同类的两个实例才会比较大小，所以向 TreeSet 中添加的应该是同一个类的对象。
- 对于 TreeSet 集合而言，它判断两个对象是否相等的唯一标准是：两个对象通过 `compareTo(Object obj)`方法比较返回值。
- 当需要把一个对象放入 TreeSet 中，重写该对象对应的 equals() 方法时，应保证该方法与 compareTo(Object obj) 方法有一致的结果：如果两个对象通过 equals() 方法比较返回 true，则通过 compareTo(Object obj) 方法比较应返回 0。否则，让人难以理解。

5.4.2 举例

```
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 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 (Object obj : set) {
            String s = (String)obj;
            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));
        }
    }
}
```

5.4.3 练习

练习 1：在一个 List 集合中存储了多个无大小顺序并且有重复的字符串，定义一个方法，让其有序(从小到大排序)，而且还不能去除重复。

提示：考查 ArrayList、TreeSet

```
/**
 *
 * @Description
 * @author 尚硅谷-宋红康 Email:shkstart@126.com
 * @version
 * @date 2022 年4 月7 日上午12:50:46
 *
 */
public class SortTest {
      public static void main(String[] args) {
            ArrayList list = new ArrayList();
            list.add("ccc");
            list.add("ccc");
            list.add("aaa");
            list.add("aaa");
            list.add("bbb");
            list.add("ddd");
            list.add("ddd");
            sort(list);
            System.out.println(list);
      }

      /*
       * 对集合中的元素排序,并保留重复
       */
      public static void sort(List list) {
            TreeSet ts = new TreeSet(new Comparator() { // 定义比较器(new Comparator(){}是Comparator 的子类对象)
```

```
                    @Override
                    public int compare(Object o1, Object o2) { // 重写compare 方法
                    String s1 = (String)o1;
                    String s2 = (String)o2;
                            int num = s1.compareTo(s2); // 比较内容
                            return num == 0 ? 1 : num; // 如果内容一样返回一个不为0 的数字即可
                    }
            });

            ts.addAll(list); // 将list 集合中的所有元素添加到ts 中
            list.clear(); // 清空list
            list.addAll(ts); // 将ts 中排序并保留重复的结果在添加到list 中
        }
}
```

练习 2：TreeSet 的自然排序和定制排序

1. 定义一个 Employee 类。 该类包含：private 成员变量 name,age,birthday，其中 birthday 为 MyDate 类的对象； 并为每一个属性定义 getter, setter 方法；并重写 toString 方法输出 name, age, birthday

2. MyDate 类包含: private 成员变量 year,month,day；并为每一个属性定义 getter, setter 方法；

3. 创建该类的 5 个对象，并把这些对象放入 TreeSet 集合中（下一章：TreeSet 需使用泛型来定义）

4. 分别按以下两种方式对集合中的元素进行排序，并遍历输出：

   1). 使 Employee 实现 Comparable 接口，并按 name 排序 2). 创建 TreeSet 时传入 Comparator 对象，按生日日期的先后排序。

6. Map 接口

现实生活与开发中，我们常会看到这样的一类集合：用户 ID 与账户信息、学生姓名与考试成绩、IP 地址与主机名等，这种一一对应的关系，就称作映射。Java 提供了专门的集合框架用来存储这种映射关系的对象，即 `java.util.Map` 接口。

6.1 Map 接口概述

- Map 与 Collection 并列存在。用于保存具有映射关系的数据：key-value
  - `Collection` 集合称为单列集合，元素是孤立存在的（理解为单身）。
  - `Map` 集合称为双列集合，元素是成对存在的(理解为夫妻)。

- Map 中的 key 和 value 都可以是任何引用类型的数据。但常用 String 类作为 Map 的“键”
- Map 中的 `key` 用 `Set` 来存放，不允许重复，即同一个 Map 对象所对应的类，须重写 hashCode()和 equals()方法

image-20220514190412763

- key 和 value 之间存在单向一对一关系，即通过指定的 key 总能找到唯一的、确定的 value，不同 key 对应的 `value` 可以重复。
- 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

image-20220409001015034

- Map 中存储的 key、value 的特点如下：

image-20220409001213720

6.2 Map 接口的常用方法

- **添加、修改操作：**
  - Object put(Object key,Object value)：将指定 key-value 添加到(或修改)当前 map 对象中
  - void putAll(Map m):将 m 中的所有 key-value 对存放到当前 map 中
- **删除操作：**
  - Object remove(Object key)：移除指定 key 的 key-value 对，并返回 value
  - void clear()：清空当前 map 中的所有数据
- **元素查询的操作：**
  - Object get(Object key)：获取指定 key 对应的 value
  - boolean containsKey(Object key)：是否包含指定的 key
  - boolean containsValue(Object value)：是否包含指定的 value
  - int size()：返回 map 中 key-value 对的个数
  - boolean isEmpty()：判断当前 map 是否为空

- boolean equals(Object obj)：判断当前 map 和参数对象 obj 是否相等

- **元视图操作的方法：**
  - Set keySet()：返回所有 key 构成的 Set 集合
  - Collection values()：返回所有 value 构成的 Collection 集合
  - Set entrySet()：返回所有 key-value 对构成的 Set 集合

举例：

```
package com.atguigu.map;

import java.util.HashMap;

public class TestMapMethod {
    public static void main(String[] args) {
        //创建 map 对象
        HashMap map = new HashMap();

        //添加元素到集合
        map.put("黄晓明", "杨颖");
        map.put("李晨", "李小璐");
        map.put("李晨", "范冰冰");
        map.put("邓超", "孙俪");
        System.out.println(map);

        //String remove(Object key)
        System.out.println(map.remove("黄晓明"));
        System.out.println(map);

        // Object get(Object key)
        System.out.println(map.get("邓超"));
        System.out.println(map.get("黄晓明"));

    }
}
```

举例：

```
      public static void main(String[] args) {
        HashMap 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 keySet = map.keySet();
        for (Object key : keySet) {
            System.out.println(key);
        }

        System.out.println("所有的 value: ");
        Collection values = map.values();
        for (Object value : values) {
            System.out.println(value);
        }

        System.out.println("所有的映射关系");
        Set entrySet = map.entrySet();
        for (Object mapping : entrySet) {
//                System.out.println(entry);
            Map.Entry entry = (Map.Entry) mapping;
            System.out.println(entry.getKey() + "->" + entry.getValue
());
        }
    }
```

6.3 Map 的主要实现类：HashMap

6.3.1 HashMap 概述

- HashMap 是 Map 接口使用频率最高的实现类。
- 存储数据采用的哈希表结构，允许使用 null 键和 null 值，与 HashSet 一样，元素的存取顺序不能保证一致。
- 所有的 key 构成的集合是 Set：无序的、不可重复的。所以，key 所在的类要重写：equals()和 hashCode()
- 所有的 value 构成的集合是 Collection：无序的、可以重复的。所以，value 所在的类要重写：equals()
- 一个 key-value 构成一个 entry
- 所有的 entry 构成的集合是 Set:无序的、不可重复的
- HashMap 判断两个 key 相等的标准是：两个 key 通过 equals() 方法返回 true，hashCode 值也相等。
- HashMap 判断两个 value 相等的标准是：两个 value 通过 equals() 方法返回 true。

6.3.2 练习

练习 1：WordCount 统计

需求：统计字符串中每个字符出现的次数

String str = "aaaabbbcccccccccc";

提示：

char[] arr = str.toCharArray(); //将字符串转换成字符数组

HashMap hm = new HashMap(); //创建双列集合存储键和值，键放字符，值放次数

```
/**
 *
 * @Description
 * @author 尚硅谷-宋红康  Email:shkstart@126.com
 * @version
 * @date 2021 年 5 月 7 日上午 12:26:59
 *
 */
public class WordCountTest {
      public static void main(String[] args) {
            String str = "aaaabbbcccccccccc";
            char[] arr = str.toCharArray(); // 将字符串转换成字符数组
            HashMap<Character, Integer> map = new HashMap<>(); // 创建双
列集合存储键和值

            for (char c : arr) { // 遍历字符数组
                  if (!map.containsKey(c)) { // 如果不包含这个键
                        map.put(c, 1); // 就将键和值为 1 添加
                  } else { // 如果包含这个键
                        map.put(c, map.get(c) + 1); // 就将键和值再加 1 添
加进来
                  }

            }

            for (Object key : map.keySet()) { // 遍历双列集合
                  System.out.println(key + "=" + map.get(key));
            }

      }
}
```

练习 2：二级联动

将省份和城市的名称保存在集合中，当用户选择省份以后，二级联动，显示对应省份的地级市供用户选择。

效果演示：

```
黑龙江    上海    吉林    天津    河北    重庆    北京
请选择你所在的省份：
吉林
长春    延边    吉林    白山    白城    四平    松原
请选择你所在的城市：
长春
信息登记完毕
```

```
/**
 *
 * @Description
 * @author 尚硅谷-宋红康  Email:shkstart@126.com
 * @version
 * @date 2021 年 5 月 7 日上午 12:26:59
 *
 */
class CityMap{

      public static Map model = new HashMap();

      static {
            model.put("北京", new String[] {"北京"});
            model.put("上海", new String[] {"上海"});
            model.put("天津", new String[] {"天津"});
            model.put("重庆", new String[] {"重庆"});
            model.put("黑龙江", new String[] {"哈尔滨","齐齐哈尔","牡丹江
","大庆","伊春","双鸭山","绥化"});
            model.put("吉林", new String[] {"长春","延边","吉林","白山","
白城","四平","松原"});
            model.put("河北", new String[] {"石家庄","张家口","邯郸","邢
台","唐山","保定","秦皇岛"});
      }

}

public class ProvinceTest {
      public static void main(String[] args) {

            Set keySet = CityMap.model.keySet();
```

```
            for(Object s : keySet) {
                  System.out.print(s + "\t");
            }
            System.out.println();
            System.out.println("请选择你所在的省份：");
            Scanner scan = new Scanner(System.in);
            String province = scan.next();

            String[] citys = (String[])CityMap.model.get(province);
            for(String city : citys) {
                  System.out.print(city + "\t");
            }
            System.out.println();
            System.out.println("请选择你所在的城市：");
            String city = scan.next();

            System.out.println("信息登记完毕");
      }

}
```

练习 3：添加你喜欢的歌手以及你喜欢他唱过的歌曲

例如：

```
张学友=[《一路上有你》，《吻别》，《一千个伤心的理由》]
王菲=[《红豆》，《传奇》，《容易受伤的女人》]
```

```
//方式1
package com.atguigu.test12;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Set;

public class Test12 {
      @SuppressWarnings("all")
      public static void main(String[] args) {
            HashMap map = new HashMap();

            ArrayList wangfei = new ArrayList();
            wangfei.add("《红豆》");
            wangfei.add("《传奇》");
            wangfei.add("《容易受伤的女人》");
            map.put("王菲", wangfei);
```

```
            ArrayList zxy = new ArrayList();
            zxy.add("《一路上有你》");
            zxy.add("《吻别》");
            zxy.add("《一千个伤心的理由》");
            map.put("张学友", zxy);

            Set entrySet = map.entrySet();
            for (Object object : entrySet) {
                  System.out.println(object);
            }
      }
}

//方式2：使用上泛型（下章讲）
public class SingerTest {
      public static void main(String[] args) {
            HashMap<String, ArrayList<String>> singers = new HashMap<>
();
            singers.put("张学友", new ArrayList<String>() {
                  {
                        add("一路上有你");
                        add("吻别");
                        add("一千个伤心的理由");
                  }
            });
            singers.put("王菲", new ArrayList<String>() {
                  {
                        add("红豆");
                        add("传奇");
                        add("容易受伤的女人");
                  }
            });
            // 打印
            for (Map.Entry<String, ArrayList<String>> singer : singers.
entrySet()) {
                  System.out.println(singer.getKey() + "的歌曲有：" + si
nger.getValue().toString());
            }
      }
}

//方式3：
public class Test1 {
      @Test
      public void test1() {
```

```java
            Singer singer1 = new Singer("张学友");
            Singer singer2 = new Singer("王菲");

            Song song1 = new Song("一路上有你");
            Song song2 = new Song("吻别");
            Song song3 = new Song("一千个伤心的理由");
            Song song4 = new Song("红豆");
            Song song5 = new Song("传奇");
            Song song6 = new Song("容易伤心的女人");

            HashSet h1 = new HashSet();// 放歌手一的歌曲
            h1.add(song1);
            h1.add(song2);
            h1.add(song3);

            HashSet h2 = new HashSet();// 放歌手二的歌曲
            h2.add(song4);
            h2.add(song5);
            h2.add(song6);

            HashMap hashMap = new HashMap();// 放歌手和他对应的歌曲
            hashMap.put(singer1, h1);
            hashMap.put(singer2, h2);

            for (Object obj : hashMap.keySet()) {
                  System.out.println(obj + "=" + hashMap.get(obj));
            }

      }
}

//歌曲
public class Song implements Comparable{
      private String songName;//歌名

      public Song() {
            super();
      }

      public Song(String songName) {
            super();
            this.songName = songName;
      }

      public String getSongName() {
            return songName;
      }
```

```java
        public void setSongName(String songName) {
                this.songName = songName;
        }

        @Override
        public String toString() {
                return "《" + songName + "》";
        }

        @Override
        public int compareTo(Object o) {
                if(o == this){
                        return 0;
                }
                if(o instanceof Song){
                        Song song = (Song)o;
                        return songName.compareTo(song.getSongName());
                }
                return 0;
        }


}
//歌手
public class Singer implements Comparable{
        private String name;
        private Song song;

        public Singer() {
                super();
        }

        public Singer(String name) {
                super();
                this.name = name;

        }

        public String getName() {
                return name;
        }

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

        public Song getSong() {
                return song;
```

```
    }

    public void setSong(Song song) {
        this.song = song;
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public int compareTo(Object o) {
        if(o == this){
            return 0;
        }
        if(o instanceof Singer){
            Singer singer = (Singer)o;
            return name.compareTo(singer.getName());
        }
        return 0;
    }


}
```

6.4 Map 实现类之二：LinkedHashMap

- LinkedHashMap 是 HashMap 的子类
- 存储数据采用的哈希表结构+链表结构，在 HashMap 存储结构的基础上，使用了一对双向链表来记录添加元素的顺序，可以保证元素的存取顺序一致。
- 通过哈希表结构可以保证的键的唯一、不重复，需要重写键的 hashCode()方法、equals()方法。

```
public class TestLinkedHashMap {
    public static void main(String[] args) {
        LinkedHashMap 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 entrySet = map.entrySet();
        for (Object obj : entrySet) {
            Map.Entry entry = (Map.Entry)obj;
            System.out.println(entry);
        }
    }
}
```

6.5 Map 实现类之三：TreeMap

- TreeMap 存储 Key-Value 对时，需要根据 key-value 对进行排序。TreeMap 可以保证所有的 Key-Value 对处于有序状态。
- TreeSet 底层使用红黑树结构存储数据
- TreeMap 的 Key 的排序：
  - 自然排序：TreeMap 的所有的 Key 必须实现 Comparable 接口，而且所有的 Key 应该是同一个类的对象，否则将会抛出 ClasssCastException
  - 定制排序：创建 TreeMap 时，传入一个 Comparator 对象，该对象负责对 TreeMap 中的所有 key 进行排序。此时不需要 Map 的 Key 实现 Comparable 接口
- TreeMap 判断两个 key 相等的标准：两个 key 通过 compareTo()方法或者 compare()方法返回 0。

```
/**
 * @author 尚硅谷-宋红康
 * @create 1:23
 */
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<Map.Entry<String, Integer>> entrySet = map.entrySet();
        for (Map.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<Str
ing>() {

            @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<Map.Entry<String, Integer>> entrySet = map.entrySet();
        for (Map.Entry<String, Integer> entry : entrySet) {
            System.out.println(entry);
        }
    }
}
```

6.6 Map 实现类之四：Hashtable

- Hashtable 是个古老的 Map 实现类，JDK1.0 就提供了。不同于 HashMap，Hashtable 是线程安全的。
- Hashtable 实现原理和 HashMap 相同，功能相同。底层都使用哈希表结构，查询速度快，很多情况下可以互用。
- 与 HashMap 不同，Hashtable 不允许使用 null 作为 key 和 value
- 与 HashMap 一样，Hashtable 也不能保证其中 Key-Value 对的顺序
- Hashtable 判断两个 key 相等、两个 value 相等的标准，与 HashMap 一致。

面试题：Hashtable 和 HashMap 的区别

```
HashMap:底层是一个哈希表,是一个线程不安全的集合,速度快
Hashtable:底层也是一个哈希表,是一个线程安全的集合,速度慢

HashMap 集合:可以存储 null 值,null 键
Hashtable 集合,不能存储 null 值,null 键
```

Hashtable 和 Vector 集合一样,在 jdk1.2 版本之后被更先进的集合(HashMap,ArrayList)取代了

Hashtable 的子类 Properties（配置文件）依然活跃在历史舞台　　xml
Properties 集合是一个唯一和 IO 流相结合的集合

6.7 Map 实现类之五：Properties

- Properties 类是 Hashtable 的子类，该对象用于处理属性文件
- 由于属性文件里的 key、value 都是字符串类型，所以 Properties 里的 key 和 value 都是字符串类型
- 存取数据时，建议使用 setProperty(String key,String value)方法和 getProperty(String key)方法

```
    @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);
  }
  @Test
  public void test02() {
      Properties properties = new Properties();
      properties.setProperty("user","songhk");
      properties.setProperty("password","123456");
      System.out.println(properties);
  }

  @Test
  public void test03() throws IOException {
      Properties pros = new Properties();
      pros.load(new FileInputStream("jdbc.properties"));
      String user = pros.getProperty("user");
      System.out.println(user);
  }
```

7. 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。

7.1 常用方法

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：

排序操作：（均为 static 方法）

- reverse(List)：反转 List 中元素的顺序
- shuffle(List)：对 List 集合元素进行随机排序
- sort(List)：根据元素的自然顺序对指定 List 集合元素按升序排序
- sort(List，Comparator)：根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
- swap(List，int， int)：将指定 list 集合中的 i 处元素和 j 处元素进行交换

查找

- Object max(Collection)：根据元素的自然顺序，返回给定集合中的最大元素
- Object max(Collection，Comparator)：根据 Comparator 指定的顺序，返回给定集合中的最大元素
- Object min(Collection)：根据元素的自然顺序，返回给定集合中的最小元素
- Object min(Collection，Comparator)：根据 Comparator 指定的顺序，返回给定集合中的最小元素
- int binarySearch(List 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。
- int binarySearch(List list,T key,Comparator 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。
- int frequency(Collection c，Object o)：返回指定集合中指定元素的出现次数

复制、替换

- void copy(List dest,List src)：将 src 中的内容复制到 dest 中
- boolean replaceAll(List list，Object oldVal，Object newVal)：使用新值替换 List 对象的所有旧值

- 提供了多个 unmodifiableXxx()方法，该方法返回指定 Xxx 的不可修改的视图。

添加

- boolean addAll(Collection c,T... elements)将所有指定元素添加到指定 collection 中。

同步

- 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：

static <T> Collection<T>	synchronizedCollection(Collection<T> c)	Returns a synchronized (thread-safe) collection backed by the specified collection.
static <T> List<T>	synchronizedList(List<T> list)	Returns a synchronized (thread-safe) list backed by the specified list.
static <K,V> Map<K,V>	synchronizedMap(Map<K,V> m)	Returns a synchronized (thread-safe) map backed by the specified map.
static <K,V> NavigableMap<K,V>	synchronizedNavigableMap(NavigableMap<K,V> m)	Returns a synchronized (thread-safe) navigable map backed by the specified navigable map.
static <T> NavigableSet<T>	synchronizedNavigableSet(NavigableSet<T> s)	Returns a synchronized (thread-safe) navigable set backed by the specified navigable set.
static <T> Set<T>	synchronizedSet(Set<T> s)	Returns a synchronized (thread-safe) set backed by the specified set.
static <K,V> SortedMap<K,V>	synchronizedSortedMap(SortedMap<K,V> m)	Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
static <T> SortedSet<T>	synchronizedSortedSet(SortedSet<T> s)	Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.

7.2 举例

```
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): 使用新值替换 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<? exte
nds 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.IndexOutOfBoundsExcep
tion: Source does not fit in dest
        System.out.println(list);

    }
    @Test
```

```java
    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C
omparator 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", 2
5));//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... e
lements)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
    }
}
```

7.3 练习

练习 1：

请从键盘随机输入 10 个整数保存到 List 中，并按倒序、从大到小的顺序显示出来

练习 2：

姓氏统计：一个文本文件中存储着北京所有高校在校生的姓名，格式如下：

每行一个名字，姓与名以空格分隔：
张 三
李 四
王 小五

现在想统计所有的姓氏在文件中出现的次数，请描述一下你的解决方案。

```java
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader(new File("e:/name.tx
t")));
            String value = null; // 临时接收文件中的字符串变量
            StringBuffer buffer = new StringBuffer();
            flag:
            while ((value = br.readLine()) != null) { // 开始读取文件中
的字符
                char[] c = value.toCharArray();
                for (int i = 0; i < c.length; i++) {
                    if (c[i] != ' ') {
                        buffer.append(String.valueOf(c[i]));
                    } else {
                        if (map.containsKey(buffer.toString())) {
                            int count = map.get(buffer.toString());
                            map.put(buffer.toString(), count + 1);
                        } else {
                            map.put(buffer.toString(), 1);
                        }
                        buffer.delete(0, buffer.length());
                        continue flag;
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        Set<Map.Entry<String, Integer>> set = map.entrySet();
        Iterator<Map.Entry<String, Integer>> it = set.iterator();
        while (it.hasNext()) {
            Map.Entry<String, Integer> end = (Map.Entry<String, Intege
r>) it.next();
            System.out.println(end);
        }

    }
```

练习 3：模拟斗地主洗牌和发牌，牌没有排序

效果演示：

```
Tom:
[梅花K, 梅花8, 梅花J, 梅花10, 黑桃7, 方片J, 红桃6, 方片4, 红桃9, 小王, 黑桃J, 方片2, 红桃3, 方片5, 方片K, 方片Q, 红桃A]
Jerry:
[梅花3, 方片6, 黑桃8, 梅花5, 黑桃6, 红桃2, 红桃K, 红桃5, 梅花7, 梅花Q, 黑桃K, 大王, 梅花4, 梅花9, 方片9, 红桃J, 黑桃9]
me:
[红桃4, 黑桃4, 黑桃2, 红桃10, 黑桃3, 方片3, 方片7, 方片10, 红桃8, 方片A, 黑桃Q, 红桃Q, 梅花6, 方片8, 黑桃10, 黑桃A, 梅花2]
底牌:
[黑桃5, 梅花A, 红桃7]
```

提示：

```
String[] num = {"A","2","3","4","5","6","7","8","9","10","J","Q","K"};

String[] color = {"方片","梅花","红桃","黑桃"};

ArrayList<String> poker = new ArrayList<>();
```

代码示例：

```
/**
 *
 * @Description
 * @author 尚硅谷-宋红康  Email:shkstart@126.com
 * @version
 * @date 2019 年 5 月 7 日上午 12:26:59
 *
 */
public class PokerTest {

      public static void main(String[] args) {
            String[] num = {"A","2","3","4","5","6","7","8","9","10","J
","Q","K"};
            String[] color = {"方片","梅花","红桃","黑桃"};
            ArrayList<String> poker = new ArrayList<>();

            for (String s1 : color) {
                  for (String s2 : num) {
                        poker.add(s1.concat(" " + s2));
                  }
            }

            poker.add("小王");
            poker.add("大王");
            // 洗牌
            Collections.shuffle(poker);
            // 发牌
            ArrayList<String> Tom = new ArrayList<>();
```

```
            ArrayList<String> Jerry = new ArrayList<>();
            ArrayList<String> me = new ArrayList<>();
            ArrayList<String> lastCards = new ArrayList<>();

            for (int i = 0; i < poker.size(); i++) {
                  if (i >= poker.size() - 3) {
                        lastCards.add(poker.get(i));
                  } else if (i % 3 == 0) {
                        Tom.add(poker.get(i));
                  } else if (i % 3 == 1) {
                        Jerry.add(poker.get(i));
                  } else {
                        me.add(poker.get(i));
                  }
            }

            // 看牌
            System.out.println("Tom: \n" + Tom);
            System.out.println("Jerry: \n" + Jerry);
            System.out.println("me: \n" + me);
            System.out.println("底牌: \n" + lastCards);
      }
}
```

练习 4：模拟斗地主洗牌和发牌并对牌进行排序的代码实现

提示：考查 HashMap、TreeSet、ArrayList、Collections

代码示例：

```
public class PokerTest1 {

      public static void main(String[] args) {
            String[] num = { "3", "4", "5", "6", "7", "8", "9", "10", "
J", "Q", "K", "A", "2" };
            String[] color = { "方片", "梅花", "红桃", "黑桃" };
            HashMap<Integer, String> map = new HashMap<>(); // 存储索引
和扑克牌
            ArrayList<Integer> list = new ArrayList<>(); // 存储索引
            int index = 0; // 索引的开始值
            for (String s1 : num) {
                  for (String s2 : color) {
                        map.put(index, s2.concat(" " + s1)); // 将索引和
扑克牌添加到HashMap 中
                        list.add(index); // 将索引添加到ArrayList 集合中
                        index++;
                  }
            }
            map.put(index, "小王");
```

```
            list.add(index);
            index++;
            map.put(index, "大王");
            list.add(index);
            // 洗牌
            Collections.shuffle(list);
            // 发牌
            TreeSet<Integer> Tom = new TreeSet<>();
            TreeSet<Integer> Jerry = new TreeSet<>();
            TreeSet<Integer> me = new TreeSet<>();
            TreeSet<Integer> lastCards = new TreeSet<>();
            for (int i = 0; i < list.size(); i++) {
                  if (i >= list.size() - 3) {
                        lastCards.add(list.get(i)); // 将list 集合中的索
引添加到TreeSet 集合中会自动排序
                  } else if (i % 3 == 0) {
                        Tom.add(list.get(i));
                  } else if (i % 3 == 1) {
                        Jerry.add(list.get(i));
                  } else {
                        me.add(list.get(i));
                  }
            }

            // 看牌
            lookPoker("Tom", Tom, map);
            lookPoker("Jerry", Jerry, map);
            lookPoker("康师傅", me, map);
            lookPoker("底牌", lastCards, map);

      }

      public static void lookPoker(String name, TreeSet<Integer> ts, Ha
shMap<Integer, String> map) {
            System.out.println(name + "的牌是:");
            for (Integer index : ts) {
                  System.out.print(map.get(index) + " ");
            }

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

第 13 章_泛型(Generic)

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

1. 泛型概述

1.1 泛型的引入

泛型：标签

举例：

- 中药店，每个抽屉外面贴着标签

- 超市购物架上很多瓶子，每个瓶子装的是什么，有标签

- 家庭厨房中：

image-20220514191533296

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。

举例 1：

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小，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<T>就是类型参数，即泛型。

举例 2：

集合容器类在设计阶段/声明阶段不能确定这个容器到底实际存的是什么类型的对象，所以**在 JDK1.5 之前只能把元素类型设计为 Object，JDK1.5 之后使用泛型来解决。**因为这个时候除了元素的类型不确定，其他的部分是确定的，例如关于这个元素如何保存，如何管理等是确定的，因此此时**把元素的类型设计成一个参数，这个类型参数叫做泛型。**Collection<E>，List<E>，ArrayList<E> 这个<E>就是类型参数，即泛型。

1.2 泛型的概念

所谓泛型，就是允许在定义类、接口时通过一个标识表示类中某个属性的类型或者是某个方法的返回值及参数类型。这个类型参数将在使用时（例如，继承或实现这

个接口，用这个类型声明变量、创建对象时）确定（即传入实际的类型参数，也称为类型实参）。

从 JDK1.5 以后，Java 引入了“参数化类型（Parameterized type）”的概念，允许我们在创建集合时再指定集合元素的类型，正如：List，这表明该 List 只能保存字符串类型的对象。

JDK1.5 改写了集合框架中的全部接口和类，为这些接口、类增加了泛型支持，从而可以在声明集合变量、创建集合对象时传入类型实参。

2. 使用泛型的好处

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。

2.1 泛型在集合中的使用

那么为什么要有泛型呢，直接 Object 不是也可以存储数据吗？

解决元素存储的安全性问题，好比商品、药品标签，不会弄错。

解决获取数据元素时，需要类型强制转换的问题，好比不用每回拿商品、药品都要辨别。

2.1.1 举例

在集合中没有泛型时：

image-20220411001522636

在集合中有泛型时：

只有指定类型才可以添加到集合中：**类型安全**　读取出来的对象不需要强转：**便捷**

image-20220411001549747

Java 泛型可以保证如果程序在编译时没有发出警告，运行时就不会产生 ClassCastException 异常。同时，代码更加简洁、健壮。

此外，

```
ArrayList<Integer> list = new ArrayList<>();//类型推断

list.add(78);
list.add(88);
list.add(77);
list.add(66);

//遍历方式一:
//for(Integer i : list){
//不需要强转
    //System.out.println(i);
//}

//遍历方式二:
Iterator<Integer> iterator = list.iterator();
while(iterator.hasNext()){
    System.out.println(iterator.next());
}

Map<String,Integer> map = new HashMap<String,Integer>();

map.put("Tom1",34);
map.put("Tom2",44);
map.put("Tom3",33);
map.put("Tom4",32);
//添加失败
//map.put(33, "Tom");

Set<Entry<String,Integer>> entrySet = map.entrySet();

Iterator<Entry<String,Integer>> iterator =  entrySet.iterator();

while(iterator.hasNext()){
     Entry<String,Integer> entry = iterator.next();
```

```
    System.out.println(entry.getKey() + "--->" + entry.getValue());
}
```

2.1.2 练习

练习 1：

（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);
```

```
        }
    }
}
```

练习 2：编写一个简单的同学通迅录

需求说明：

- 查询所有通讯录的同学信息。
- 输入姓名，根据姓名查询指定同学信息。如果该姓名不存在，输出提示信息。
- 添加同学，姓名重复的不能添加。
- 根据学员姓名删除学员。
- 按姓名排序查询学员。

分析：

- 使用 HashMap<K,V>存储同学信息，使用同学姓名做 key，同学对象做 value。
- 同学对象包含的属性有：姓名、年龄、住址、爱好等。

2.2 泛型在比较器中的使用

2.2.1 举例

```java
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 CircleComparator 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) {
        CircleComparator com = new CircleComparator();
        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：Cla
ssCastException
    }
}
```

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：

```java
package com.atguigu.generic;

import java.util.Comparator;

public class CircleComparator1 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

```java
package com.atguigu.generic;

public class TestHasGeneric {
    public static void main(String[] args) {
        CircleComparator1 com = new CircleComparator1();
        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，而不是运行期；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。

2.2.2 练习

（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。

```java
package com.atguigu.genericclass.use;

public class Rectangle implements Comparable<Rectangle>{
```

```java
    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;
    }

    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.pe
rimeter());
    }
}
```

```java
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);
        }
    }
}
```

3. 自定义泛型结构

3.1 泛型相关说明

1、<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
  - 类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；
- 这里的 T，可以替换成 K，V 等任意字母。习惯用 T 表示，是 Type 的缩写。
- Comparator 这种就称为参数化类型（Parameterized Types）。

2、泛型的实例化

一定要在类名后面指定类型参数的值（类型）。如：

```
List<String> strList = new ArrayList<String>();

Iterator<Customer> iterator = customers.iterator();
```

- T 只能是类，不能用基本数据类型填充。但可以使用包装类填充
- 把一个集合中的内容限制为一个特定的数据类型，这就是 generics 背后的核心思想。

3、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){
    ....
}
```

3.2 自定义泛型类或泛型接口

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。

3.2.1 说明

1. 泛型类可能有多个参数，此时应将多个参数一起放在尖括号内。比如：<E1,E2,E3>
2. 泛型类的构造器如下：public GenericClass(){}。 而下面是错误的：public GenericClass(){}
3. 当创建泛型类对象、子类继承泛型类时、实现类实现泛型接口时可以确定泛型结构中的泛型参数。
4. 泛型不同的引用不能相互赋值。

尽管在编译时 ArrayList 和 ArrayList 是两种类型，但是，在运行时只有一个 ArrayList 被加载到 JVM 中。

1. 泛型如果不指定，将被擦除，泛型对应的类型均按照 Object 处理，但不等价于 Object。

   经验：泛型要使用一路都用。要不用，一路都不要用。
2. 如果泛型结构是一个接口或抽象类，则不可创建泛型类的对象。
3. jdk1.7 开始泛型的简化操作：`ArrayList<Fruit> flist = new ArrayList<>();`
4. 泛型的指定中不能使用基本数据类型，可以使用包装类替换。
5. 在类/接口上声明的泛型，在本类或本接口中即代表某种类型，可以作为非静态属性的类型、非静态方法的参数类型、非静态方法的返回值类型。但在静态方法中不能使用类的泛型。
6. 异常类不能是泛型的
7. 不能使用 new E[]。但是可以：E[] elements = (E[])new Object[capacity];

   参考：ArrayList 源码中声明：Object[] elementData，而非泛型参数类型数组。

12.父类有泛型，子类可以选择保留泛型也可以选择指定泛型类型：

- 子类不保留父类的泛型：按需实现
  - 没有类型 擦除
  - 具体类型
- 子类保留父类的泛型：泛型子类
  - 全部保留

- 部分保留

- 结论：子类必须是“富二代”，子类除了指定或保留父类的泛型，还可以增加自己的泛型

3.2.2 举例

举例 1：

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。

```java
package com.atguigu.genericclass.define;

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;
    }
}

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");

        //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
    }
}
```

举例 2:

```
class GenericTest {
    public static void main(String[] args) {
        // 1、使用时：类似于Object，不等同于Object
        ArrayList list = new ArrayList();
        // list.add(new Date());//有风险
        list.add("hello");

        test(list);// 泛型擦除，编译不会类型检查

        // ArrayList<Object> list2 = new ArrayList<Object>();
        // test(list2);//一旦指定Object，编译会类型检查，必须按照Object 处理
    }

    public static void test(ArrayList<String> list) {
        String str = "";
        for (String s : list) {
            str += s + ",";
        }
        System.out.println("元素:" + str);
    }
}
```

举例 3:

```
class Father<T1, T2> {
}
// 子类不保留父类的泛型
// 1)没有类型 擦除
```

```
class Son1 extends Father {// 等价于class Son extends Father<Object,Obj
ect>{
}
// 2)具体类型
class Son2 extends Father<Integer, String> {
}
// 子类保留父类的泛型
// 1)全部保留
class Son3<T1, T2> extends Father<T1, T2> {
}
// 2)部分保留
class Son4<T2> extends Father<Integer, T2> {
}
```

举例 4：

```
class Father<T1, T2> {
}
// 子类不保留父类的泛型
// 1)没有类型 擦除
class Son<A, B> extends Father{//等价于class Son extends Father<Object,
Object>{
}
// 2)具体类型
class Son2<A, B> extends Father<Integer, String> {
}
// 子类保留父类的泛型
// 1)全部保留
class Son3<T1, T2, A, B> extends Father<T1, T2> {
}
// 2)部分保留
class Son4<T2, A, B> extends Father<Integer, T2> {
}
```

举例 5：

```
class Person<T> {
    // 使用T类型定义变量
    private T info;
    // 使用T类型定义一般方法
    public T getInfo() {
        return info;
    }
    public void setInfo(T info) {
        this.info = info;
    }
    // 使用T类型定义构造器
    public Person() {
    }
```

```
    public Person(T info) {
        this.info = info;
    }
    // static 的方法中不能声明泛型
    //public static void show(T t) {
    //
    //}
    // 不能在try-catch 中使用泛型定义
    //public void test() {
        //try {
        //
        //} catch (MyException<T> ex) {
        //
        //}
    //}
}
```

3.3 自定义泛型方法

3.3.1 说明

- 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>。
- 方法，也可以被泛型化，不管此时定义在其中的类是不是泛型类。在泛型方法中可以定义泛型参数，此时，参数的类型就是传入数据的类型。
- 泛型方法的格式：

```
[访问权限]  <泛型>  返回值类型  方法名([泛型标识 参数名称])  [抛出的异常]{

}
```

3.3.2 举例

举例 1：

```
public class DAO {

    public <E> E get(int id, E e) {

        E result = null;

        return result;
    }
}
```

举例 2：

```
public static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
```

```
        c.add(o);
    }
}

public static void main(String[] args) {
    Object[] ao = new Object[100];
    Collection<Object> co = new ArrayList<Object>();
    fromArrayToCollection(ao, co);

    String[] sa = new String[20];
    Collection<String> cs = new ArrayList<>();
    fromArrayToCollection(sa, cs);

    Collection<Double> cd = new ArrayList<>();
    // 下面代码中T是Double类，但sa是String类型，编译错误。
    // fromArrayToCollection(sa, cd);
    // 下面代码中T是Object类型，sa是String类型，可以赋值成功。
    fromArrayToCollection(sa, co);
}
```

举例 3：

```
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;
                }
            }
        }
    }
}

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","song"};
        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Compa
rable接口
```

```
    }
}
```

4. 泛型在继承上的体现

请输出如下来两段代码有何不同：

片段 1：

```
public void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (int k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

片段 2：

```
public void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

如果 B 是 A 的一个子类型（子类或者子接口），而 G 是具有泛型声明的类或接口，G 并不是 G 的子类型！

比如：String 是 Object 的子类，但是 List 并不是 List 的子类。

image-20220411003422259

```
public void testGenericAndSubClass() {
    Person[] persons = null;
    Man[] mans = null;
    // 而 Person[] 是 Man[] 的父类.
    persons = mans;

    Person p = mans[0];
```

```
    // 在泛型的集合上
    List<Person> personList = null;
    List<Man> manList = null;
    // personList = manList;(报错)
}
```

5. 通配符的使用

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？。

5.1 通配符的理解

1. 使用类型通配符：？ 比如：`List<?>`，`Map<?,?>`

`List<?>`是 `List<String>`、`List<Object>`等各种泛型 List 的父类。

1. 读取 List<?>的对象 list 中的元素时，永远是安全的，因为不管 list 的真实类型是什么，它包含的都是 Object。
2. 写入 list 中的元素时，不行。因为我们不知道 c 的元素类型，我们不能向其中添加对象。 唯一的例外是 null，它是所有类型的成员。

5.2 通配符的读与写

写操作：

将任意元素加入到其中不是类型安全的：

```
Collection<?> c = new ArrayList<String>();

c.add(new Object()); // 编译时错误
```

因为我们不知道 c 的元素类型，我们不能向其中添加对象。add 方法有类型参数 E 作为集合的元素类型。我们传给 add 的任何参数都必须是一个未知类型的子类。因为我们不知道那是什么类型，所以我们无法传任何东西进去。

唯一的例外的是 null，它是所有类型的成员。

读操作：

另一方面，我们可以调用 get()方法并使用其返回值。返回值是一个未知的类型，但是我们知道，它总是一个 Object。

举例 1：

```
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 ArrayLis
t<Object>();
        m4(new ArrayList<>());//Collection<?> coll = new ArrayList<>();
        m4(new ArrayList());//Collection<?> coll = new ArrayList();
        m4(new ArrayList<String>());//Collection<?> coll = new ArrayLis
t<String>();
    }
}
```

举例 2：

```
public static void main(String[] args) {
    List<?> list = null;
    list = new ArrayList<String>();
    list = new ArrayList<Double>();
    // list.add(3);//编译不通过
    list.add(null);

    List<String> l1 = new ArrayList<String>();
    List<Integer> l2 = new ArrayList<Integer>();
    l1.add("尚硅谷");
    l2.add(15);
    read(l1);
    read(l2);
}

public static void read(List<?> list) {
    for (Object o : list) {
        System.out.println(o);
    }
}
```

5.3 使用注意点

注意点 1：编译错误：不能用在泛型方法声明上，返回值类型前面<>不能使用?

```
public static <?> void test(ArrayList<?> list){
}
```

注意点 2：编译错误：不能用在泛型类的声明上

```
class GenericTypeClass<?>{
}
```

注意点 3：编译错误：不能用在创建对象上，右边属于创建集合对象

```
ArrayList<?> list2 = new ArrayList<?>();
```

5.4 有限制的通配符

- <?>
  - 允许所有泛型的引用调用
- 通配符指定上限：<? extends 类/接口 >
  - 使用时指定的类型必须是继承某个类，或者实现某个接口，即<=
- 通配符指定下限：<? super 类/接口 >
  - 使用时指定的类型必须是操作的类或接口，或者是操作的类的父类或接口的父接口，即>=
- 说明：

```
<? extends Number>      //(无穷小 , Number]
//只允许泛型为Number 及Number 子类的引用调用

<? super Number>       //[Number , 无穷大)
//只允许泛型为Number 及Number 父类的引用调用

<? extends Comparable>
//只允许泛型为实现Comparable 接口的实现类的引用调用
```

- 举例 1

```
class Creature{}
class Person extends Creature{}
class Man extends Person{}
class PersonTest {
    public static <T extends Person> void test(T t){
        System.out.println(t);
    }

    public static void main(String[] args) {
        test(new Person());
        test(new Man());
        //The method test(T) in the type PersonTest is not
        //applicable for the arguments (Creature)
        test(new Creature());
```

```
    }
}
```

- 举例 2：

```
public static void main(String[] args) {
    Collection<Integer> list1 = new ArrayList<Integer>();
    Collection<String> list2 = new ArrayList<String>();
    Collection<Number> list3 = new ArrayList<Number>();
    Collection<Object> list4 = new ArrayList<Object>();

    getElement1(list1);
    getElement1(list2);//报错
    getElement1(list3);
    getElement1(list4);//报错

    getElement2(list1);//报错
    getElement2(list2);//报错
    getElement2(list3);
    getElement2(list4);

}
// 泛型的上限：此时的泛型?，必须是Number 类型或者Number 类型的子类
public static void getElement1(Collection<? extends Number> coll)
{}
// 泛型的下限：此时的泛型?，必须是Number 类型或者Number 类型的父类
public static void getElement2(Collection<? super Number> coll){}
```

- 举例 3：

```
public static void printCollection3(Collection<? extends Person>
coll) {
    //Iterator 只能用Iterator<?>或Iterator<? extends Person>.why?
    Iterator<?> iterator = coll.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}

public static void printCollection4(Collection<? super Person> co
ll) {
    //Iterator 只能用Iterator<?>或Iterator<? super Person>.why?
    Iterator<?> iterator = coll.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

练习：为什么编译如下的操作会报错？

```
59
60     public static void addStrings(List<? extends Object> list) {
61
62 //        list.add("aaa");
63 //        list.add("aaa");
64
65         list.add(new A());
66         list.add(new Circle());
67         list.add(new GeometricObject());
68         list.add(new Object());
69     }
70
71     public void addString(List<? super A> list) {
72
73         list.add(new A());
74         list.add(new Circle());
75         list.add(new GeometricObject());
76         list.add(new Object());
77     }
```

5.5 泛型应用举例

举例 1：泛型嵌套

```
public static void main(String[] args) {
    HashMap<String, ArrayList<Citizen>> map = new HashMap<String, Array
List<Citizen>>();
    ArrayList<Citizen> list = new ArrayList<Citizen>();
    list.add(new Citizen("刘恺威"));
    list.add(new Citizen("杨幂"));
    list.add(new Citizen("小糯米"));
    map.put("刘恺威", list);

    Set<Entry<String, ArrayList<Citizen>>> entrySet = map.entrySet();
    Iterator<Entry<String, ArrayList<Citizen>>> iterator = entrySet.ite
rator();
    while (iterator.hasNext()) {
        Entry<String, ArrayList<Citizen>> entry = iterator.next();
        String key = entry.getKey();
        ArrayList<Citizen> value = entry.getValue();
        System.out.println("户主: " + key);
        System.out.println("家庭成员: " + value);
    }
}
```

举例 2：个人信息设计

用户在设计类的时候往往会使用类的关联关系，例如，一个人中可以定义一个信息的属性，但是一个人可能有各种各样的信息（如联系方式、基本信息等），所以此信息属性的类型就可以通过泛型进行声明，然后只要设计相应的信息类即可。

image-20220411004301224

```
interface Info{          // 只有此接口的子类才是表示人的信息
}
class Contact implements Info{        // 表示联系方式
      private String address ;        // 联系地址
      private String telephone ;      // 联系方式
      private String zipcode ;        // 邮政编码
      public Contact(String address,String telephone,String zipcode){
            this.address = address;
            this.telephone = telephone;
            this.zipcode = zipcode;
      }
      public void setAddress(String address){
            this.address = address ;
      }
      public void setTelephone(String telephone){
            this.telephone = telephone ;
      }
      public void setZipcode(String zipcode){
            this.zipcode = zipcode;
      }
      public String getAddress(){
            return this.address ;
      }
      public String getTelephone(){
            return this.telephone ;
      }
      public String getZipcode(){
            return this.zipcode;
      }
```

```
	@Override
	public String toString() {
		return "Contact [address=" + address + ", telephone=" + tel
ephone
		+ ", zipcode=" + zipcode + "]";
	}
}
class Introduction implements Info{
	private String name ;		// 姓名
	private String sex ;		// 性别
	private int age ;		// 年龄
	public Introduction(String name,String sex,int age){
		this.name = name;
		this.sex = sex;
		this.age = age;
	}
	public void setName(String name){
		this.name = name ;
	}
	public void setSex(String sex){
		this.sex = sex ;
	}
	public void setAge(int age){
		this.age = age ;
	}
	public String getName(){
		return this.name ;
	}
	public String getSex(){
		return this.sex ;
	}
	public int getAge(){
		return this.age ;
	}
	@Override
	public String toString() {
		return "Introduction [name=" + name + ", sex=" + sex + ", a
ge=" + age
		+ "]";
	}
}
class Person<T extends Info>{
	private T info ;
	public Person(T info){		// 通过构造器设置信息属性内容
		this.info = info;
	}
	public void setInfo(T info){
		this.info = info ;
	}
```

```
	public T getInfo(){
		return info ;
	}
	@Override
	public String toString() {
		return "Person [info=" + info + "]";
	}

}
public class GenericPerson{
	public static void main(String args[]){
		Person<Contact> per = null ;		// 声明Person 对象
		per = new Person<Contact>(new Contact("北京市","01088888888",
"102206")) ;
		System.out.println(per);

		Person<Introduction> per2 = null ;		// 声明Person 对象
		per2 = new Person<Introduction>(new Introduction("李雷","男",
24));
		System.out.println(per2) ;
	}
}
```

第 14 章_集合源码与数据结构

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 数据结构剖析

我们举一个形象的例子来理解数据结构的作用：

战场：程序运行所需的软件、硬件环境

敌人：项目或模块的功能需求

指挥官：编写程序的程序员

士兵和装备：一行一行的代码

战术和策略：数据结构

上图：没有战术，打仗事倍功半

上图：有战术，打仗事半功倍

总结：简单来说，数据结构，就是一种程序设计优化的方法论，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，**目的是加快程序的执行速度、减少内存占用的空间。**

1.1 数据结构的研究对象一：数据间逻辑结构

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。比如：排队
- **树形结构**：数据结构中的元素存在一对多的相互关系。比如：家谱、文件系统、组织架构
- **图形结构**：数据结构中的元素存在多对多的相互关系。比如：全国铁路网、地铁图

图 1.4　四类基本结构的示意图

1.2 数据结构的研究对象二：数据的存储结构（或物理结构）

数据的物理结构/存储结构：是描述数据元素本身在内存中的存储及数据元素之间关系的表示，是数据的逻辑结构在计算机中的表示。一种数据逻辑结构可表示成一种或多种物理存储结构。

- **顺序结构**：在 Java 中体现为数组结构。

- **链式结构**：元素在内存中是不要求连续存储的，逻辑上相邻节点物理上不必相邻。

- 链表(Linked List)
  - 申请内存：一次申请一小块内存，按需申请
  - 存储思路
    - 每一个小数据单独存在一小块内存中，这个单元被称作结点(Node)
      - 数据域
      - 指针域
    - 每一小块内存知道下一小块的地址　比如：地下党，只知道自己的下线，单线联系
    - 多个不连续的小内存空间组成，通过内存地址引用形成一个链的结构
    -

    - 图示
      - 表头(头指针)：程序员获得表头就可以找到链表中所有的元素
      - 表尾：有的链表有，有的没有
  - 读：查:节点是没有下标的，只能从链表的表头开始查找某个节点，进行遍历
  - 写
    - 增
      - 申请一小块内存--结点
      - 插入到链中
        - 从头部插入
        - 从尾部插入
        - 从中间插入
    - 删
      - 从链中剥离结点
      - 释放这个结点的内存
    - 改：改变这个结点中所存储的值
  - 特点
    - 逻辑上相邻节点物理上不必相邻
    - 查找速度慢
    - 链表插入/删除较数组快
  - 适用范围
    - 插入/删除操作远多于查询操作的场景
    - 抽象数据结构的基石　栈、队列、树

- **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。在使用索引查询时，效率很高，但是需要额外存储和维护索引表。

image-20220521115200921

- **散列结构：**也称为哈希结构。根据结点的关键字通过其 `hashCode` 值来计算出该结点的存储地址，比如 Java 中的 HashSet、HashMap 底层就是散列存储结构。这种结构，添加、查询速度快。不适合进行范围查找。同时，要考虑和解决好哈希冲突问题。

image-20220521115734571

1.3 数据结构与算法脉络

尚硅谷

数据结构和算法
线性表
数组
链表
单链表
双向链表
循环链表
双向循环链表
静态链表
栈
顺序栈
链式栈
队列
普通队列
双端队列
阻塞队列
并发队列
阻塞并发队列
散列表
散列函数
冲突解决
链表法
开放寻址
其他
动态扩容
位图
树
二叉树
平衡二叉树
二叉查找树
平衡二叉查找树
AVL树
红黑树
完全二叉树
满二叉树
多路查找树
B树
B+树
2-3树
2-3-4树
堆
小顶堆
大顶堆
优先级队列
斐波那契堆
二项堆
其他
树状数组
线段树
复杂度分析
空间复杂度
时间复杂度
最好
最坏
平均
均摊
基本算法思想
贪心算法
分治算法
动态规划
回溯算法
枚举算法
排序
O(n^2)
冒泡排序
插入排序
选择排序
希尔排序
O(nlogn)
归并排序
快速排序
堆排序
O(n)
计数排序
基数排序
桶排序
搜索
深度优先搜索
广度优先搜索
A*启发式搜索
查找
线性表查找
树结构查找
散列表查找
字符串匹配
朴素
KMP
Robin-Karp
Boyer-Moore
AC自动机

算法设计中一些常见的通用想法可以称为算法设计模式。常见模式包括：

枚举法：根据具体问题枚举出各种可能，从中选出有用信息或者问题的解。这种利用计算机的速度优势，在解决简单问题时十分有效。

贪心法：根据问题的信息尽可能做出部分的解，并基于部分解逐步扩充得到完整的解。在解决复杂问题时，这种做法未必能得到最好的解。

分治法：把复杂问题分解为相对简单的子问题，分别求解，最后通过组合起子问题的解的方式得到原问题的解。

回溯法（搜索法）：专指通过探索的方式求解。如果问题很复杂，没有清晰的求解路径，可能就需要分步骤进行，而每一步骤可能有多种选择。在这种情况下，只能采用试探的方式，根据实际情况选择一个可能方向。当后面无法继续时，就需要退回到前面的步骤，另行选择求解路径，这种动作称为回溯。

动态规划法：在一些复杂情况下，问题求解很难直截了当地进行，因为需要在前面的步骤中积累信息，在后续步骤中根据已知信息，动态选择已知的最好求解路径（同时可能进一步积累信息）。这种算法模式称为动态规划。

分支限界法：可看做搜索方式的改良。如果在搜索过程中可以得到一些信息，确定某些可能的选择实际上并不真正有用，就可以及早将其删除，以缩小可能的求解空间，加速问题求解过程。

2. List 接口剖析

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。

集合框架图

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

2.1 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。

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。

- List 接口的主要实现类
  - ArrayList：动态数组
  - Vector：动态数组
  - LinkedList：双向链表
  - Stack：栈

2.2 动态数组

2.2.1 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

[0]	[1]	[2]	[3]	[4]
23	24	65	90	67

1563432676234

例如：对象数组

1563432696340

2.2.2 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);
```

```java
        }
        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;
```

```java
    }

    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");
```

```java
        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);
        }
    }
}
```

2.2.3 Java API 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 增量为0) {
    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 + capac
ityIncrement 增量;
        int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                         capacityIncrement : oldCapacit
y);

        //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
```

```java
        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;
```

```java
        //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;
    }
```

2.3 链表

2.3.1 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：链表由一系列结点 node（链表中每一个元素称为结点）组成，结点可以在代码执行过程中动态创建。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

常见的链表结构有如下的形式：

各种结点

单链表结点（1）当前结点的数据（2）下一个结点的地址

如果要某个结点，只能从头结点开始往后找

整个链表需要记录头结点

```
class Node<E>{
    E data; //记录数据
    Node next; //记录下一个结点 地址
}
```

```
class 单链表{
    private Node head; //单链表的头结点
    private int size;
}
```

双向链表结点（1）previous：上一个结点的地址

（2）当前结点的数据

（3）next：下一个结点的地址

如果要找某个结点，可以从前往后，也可以从后往前

整个链表需要记录头结点和尾结点

```
class Node<E>{
    Node previous;//记录上一个结点的地址
    E data; //记录数据
    Node next; //记录下一个结点 地址
}
```

```
class 双链表{
    private Node first; //双链表的头结点
    private Node last; //双链表的尾结点
    private int size;
}
```

二叉树结点（1）parent：父结点的地址

（2）left：左子结点的地址（3）right：右子节点的地址

（4）data：当前结点的数据

整个二叉树需要记录一个根结点，从根结点开始查找，最后一级的称为叶子结点。

```
class TreeNode<E>{
    TreeNode parent;父结点地址
    TreeNode left;左子结点地址
    E data;数据
    TreeNode right;右子节点地址
}
```

```
class 二叉树{
    private TreeNode root; //根结点
    private int size;//可选
}
```

2.3.2 自定义链表（选讲）

- 自定义单向链表

```
/*
单链表中的节点。
节点是单向链表中基本的单元。
每一个节点Node 都有两个属性：
    一个属性：是存储的数据。
    另一个属性：是下一个节点的内存地址。
 */
public class Node {

    // 存储的数据
    Object data;

    // 下一个节点的内存地址
    Node next;

    public Node(){

    }

    public Node(Object data, Node next){
        this.data = data;
        this.next = next;
    }
}

/*
链表类。(单向链表)
 */
public class Link<E> {

    // 头节点
    Node header;
```

尚硅谷

```java
    int size = 0;

    public int size(){
        return size;
    }

    // 向链表中添加元素的方法（向末尾添加）
    public void add(E data){
    //public void add(Object data){
        // 创建一个新的节点对象
        // 让之前单链表的末尾节点next 指向新节点对象。
        // 有可能这个元素是第一个，也可能是第二个，也可能是第三个。
        if(header == null){
            // 说明还没有节点。
            // new 一个新的节点对象，作为头节点对象。
            // 这个时候的头节点既是一个头节点，又是一个末尾节点。
            header = new Node(data, null);
        }else {
            // 说明头不是空！
            // 头节点已经存在了！
            // 找出当前末尾节点，让当前末尾节点的next 是新节点。
            Node currentLastNode = findLast(header);
            currentLastNode.next = new Node(data, null);
        }
        size++;
    }

    /**
     * 专门查找末尾节点的方法。
     */
    private Node findLast(Node node) {
        if(node.next == null) {
            // 如果一个节点的next 是null
            // 说明这个节点就是末尾节点。
            return node;
        }
        // 程序能够到这里说明：node 不是末尾节点。
        return findLast(node.next); // 递归算法！
    }

    /*// 删除链表中某个数据的方法
    public void remove(Object obj){
        //...
    }

    // 修改链表中某个数据的方法
    public void modify(Object newObj){
        //...
```

```
    }

    // 查找链表中某个元素的方法。
    public int find(Object obj){
        return 1;
    }*/
}
```

- 自定义双向链表

```
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;
            }
```

```java
        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();
}
```

```java
    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：

```java
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);
        }
    }
}
```

2.3.3 Java API 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。

LinkedList 是一个双向链表，如图所示：

```
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<>(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<>(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;
}
```

2.3.4 自定义单链表（选讲）

```java
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;
            }
        }
```

```java
        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;
```

```java
        }

        @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;
        }
    }
}
```

```java
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);
        }
    }
}
```

2.3.5 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，即查找元素慢。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，所以插入、删除元素快。而且链表的添加不会涉及到扩容问题。

2.4 栈（选讲）

栈（Stack）是限制仅在表的一端进行插入和删除运算的线性表。

栈的修改原则：

栈的修改是按后进先出（LIFO：last in first out）的原则进行。每次删除（退栈）的总是当前栈中"最新"的元素，即最后插入（进栈）的元素，而最先插入的是被放在栈的底部，要到最后才能删除。

- 栈(Stack)
  - 图示
  - 名词
    - 栈顶　插入、删除的这一端为栈顶
    - 栈底　相对于栈顶的另一端
    - 空栈　当表中没有元素时称为空栈
  - 特点
    - 继承于java.util.Vector
    - 后进先出
  - 操作
    - 进栈(压栈、入栈): push(ele)
    - 出栈: pop()
    - 查看栈顶元素: peek()
    - 清栈: clear()
    - 大小: size()
    - 判断是否是空:empty()
    - 查找元素出现位置: search(Object obj)
  - 使用场景
    - 二叉树和森林的遍历
    - CPU的中断处理
    - 图形的深度优先查找法(DFS)
    - 递归调用
      - 斐波那契数列
      - 汉诺塔问题
    - 子程序的调用

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 LinkedList。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());
        }
    }
}
```

练习：

编写程序，使用一维数组，模拟栈数据结构。编写测试程序，new 栈对象，调用 push、pop 方法来模拟压栈、弹栈的操作。

要求： 1、这个栈可以存储 java 中的任何引用类型的数据。 2、在栈中提供 push 方法模拟压栈。（栈满了，要有提示信息。） 3、在栈中提供 pop 方法模拟弹栈。（栈空了，也有有提示信息。） 4、假设栈的默认初始化容量是 10.（请注意无参数构造方法的编写方式。）

```
public class MyStack {
    // 向栈当中存储元素，我们这里使用一维数组模拟。存到栈中，就表示存储到数组
中。
    // 为什么选择Object 类型数组？因为这个栈可以存储java 中的任何引用类型的
数据
    private Object[] elements;

    // 栈帧，永远指向栈顶部元素
```

```java
    // 那么这个默认初始值应该是多少。注意：最初的栈是空的，一个元素都没有。
    //private int index = 0; // 如果index采用0，表示栈帧指向了顶部元素的上方。
    //private int index = -1; // 如果 index 采用-1，表示栈帧指向了顶部元素。
    private int index;

    /**
     * 无参数构造方法。默认初始化栈容量10.
     */
    public MyStack() {
        // 一维数组动态初始化
        // 默认初始化容量是10.
        this.elements = new Object[10];
        // 给index 初始化
        this.index = -1;
    }

    /**
     * 压栈的方法
     * @param obj 被压入的元素
     */
    public void push(Object obj) throws Exception {
        if(index >= elements.length - 1){
            //方式1:
            //System.out.println("压栈失败，栈已满！");
            //return;
            //方式2:
            throw new Exception("压栈失败，栈已满！");
        }
        // 程序能够走到这里，说明栈没满
        // 向栈中加1个元素，栈帧向上移动一个位置。
        index++;
        elements[index] = obj;
        System.out.println("压栈" + obj + "元素成功，栈帧指向" + index);
    }

    /**
     * 弹栈的方法，从数组中往外取元素。每取出一个元素，栈帧向下移动一位。
     * @return
     */
    public void pop() throws Exception {
        if(index < 0){
            //方式1:
            //System.out.println("弹栈失败，栈已空！");
            //return;
            //方式2:
            throw new Exception("弹栈失败，栈已空！");
```

```java
        }
        // 程序能够执行到此处说明栈没有空。
        System.out.print("弹栈" + elements[index] + "元素成功，");
        // 栈帧向下移动一位。
        index--;
        System.out.println("栈帧指向" + index);
    }

    // set 和 get 也许用不上，但是你必须写上，这是规矩。你使用 IDEA 生成就行了。
    // 封装：第一步：属性私有化，第二步：对外提供set 和get 方法。
    public Object[] getElements() {
        return elements;
    }

    public void setElements(Object[] elements) {
        this.elements = elements;
    }

    public int getIndex() {
        return index;
    }

    public void setIndex(int index) {
        this.index = index;
    }
}
```

3. 队列

队列（Queue）是只允许在一端进行插入，而在另一端进行删除的运算受限的线性表。

队列是逻辑结构，其物理结构可以是数组，也可以是链表。

队列的修改原则：队列的修改是依先进先出（FIFO）的原则进行的。新来的成员总是加入队尾（即不允许"加塞"），每次离开的成员总是队列头上的（不允许中途离队），即当前"最老的"成员离队。

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，尽管某些实现并不禁止插入 。

	抛出异常	返回特殊值
插入	add(e)	offer(e)
移除	remove()	poll()
检查	element()	peek()

Deque，是“double ended queue（双端队列）”的缩写，通常读为“deck”。双端队列是指两端都可以进行进队和出队操作的队列，如图所示，将队列的两端分别称为前端和后端，两端都可以入队和出队。其元素的逻辑结构仍是线性结构。

在双端队列进队时：前端进的元素排列在队列中后端进的元素的前面，后端进的元素排列在队列中前端进的元素的后面。在双端队列出队时，无论前端出还是后端出，先出的元素排列在后出的元素的前面。

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：

Queue 方法	**等效 Deque 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 Deque 方法**
push(e)	addFirst(e)
pop()	removeFirst()

堆栈方法	等效 Deque 方法
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
());
        }
    }
}
```

3. Set 接口剖析

3.1 Set 接口实现类

1、HashSet

`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。

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。

笔试题：

```
HashSet set = new HashSet();
Person p1 = new Person(1001,"AA");
Person p2 = new Person(1002,"BB");

set.add(p1);
```

```
set.add(p2);
p1.name = "CC";
set.remove(p1);
System.out.println(set);

set.add(new Person(1001,"CC"));
System.out.println(set);

set.add(new Person(1001,"AA"));
System.out.println(set);

//其中Person 类中重写了hashCode()和equal()方法
```

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。

3、TreeSet

TreeSet 里面维护了一个 TreeMap，底层是基于红黑树实现的！

3.2 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。

4. Map 接口剖析

4.1 Map 接口的实现类

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。

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）。

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。

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。

4.2 哈希表

HashMap 和 Hashtable 都是哈希表。

4.2.1 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。

4.2.2 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 的幂次方**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"。

```java
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之前：

image-20220514191416903

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);
        }
    }
    //...
}
```

JDK1.8之后：

同一个桶下的元素，都是因为他们的key①hashCode值相同，或②hash()结果相同或③hash&(table.length-1)的结果相同满足其一的冲突情况了

image-20220514191915271

2、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
```

3、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%运算效率没有位运算符&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。

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
}
```

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：

```java
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：

```java
  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.lengt-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。

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

```java
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;
    }

}
```

```java
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
        }
    }
}
```

4.2.3 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 capacit
y: " +
        //校验initialCapacity 合法性
   initialCapacity);
        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n
ull，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h
ashCode 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，
```

```java
        //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_CAPAC
ITY + 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 putForNullKey(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) {
        //判断是否需要库容
        //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++

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

4.2.4 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1
6 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 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, bool
ean 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, ha
sh, 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 f
or 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va
lue
            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 thresh
old
```

```
            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_CAPACI
TY);
        }
        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 的
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, oldCa
p);
                    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, tl = 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 (tl == null)
                    hd = p;//hd 记录根结点
                else {
                    p.prev = tl;
                    tl.next = p;
                }
                tl = 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()) ^ (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

}
```

HashMap--put执行过程
计算索引、逻辑判断
扩容
链表
红黑树
Start
table是否为空或length为0
是
否
resize(扩容)
根据key计算hash值得到插入的数组索引i
table[i]==null
否
table[i].key是否与要存的key相同
否
是
是
直接插入
覆盖value
++size>threshold
是
否
resize(扩容)
end
table[i]是否为treeNode
是
否
红黑树直接插入键值对
开始遍历链表准备插入
链表长度是否大于8
是
否
转换为红黑树，插入键值对
链表插入，若key存在则覆盖，不存在则插入到最后

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

（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

4.2.5 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
```

```java
    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;
    }

}
```

```java
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。

4.3 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。

4.3 树与二叉树

4.3.1 树的理解

自然界的树

计算机世界的树

image-20220521111904272

专有名词解释：

结点：树中的数据元素都称之为结点

根节点：最上面的结点称之为根，一颗树只有一个根且由根发展而来，从另外一个角度来说，每个结点都可以认为是其子树的根

父节点：结点的上层结点，如图中，结点 K 的父节点是 E、结点 L 的父节点是 G

子节点：节点的下层结点，如图中，节点 E 的子节点是 K 节点、节点 G 的子节点是 L 节点

兄弟节点：具有相同父节点的结点称为兄弟节点，图中 F、G、H 互为兄弟节点

结点的度数：每个结点所拥有的子树的个数称之为结点的度，如结点 B 的度为 3

树叶：度数为 0 的结点，也叫作终端结点，图中 D、K、F、L、H、I、J 都是树叶

非终端节点（或分支节点）：树叶以外的节点，或度数不为 0 的节点。图中根、A、B、C、E、G 都是

树的深度（或高度）：树中结点的最大层次数，图中树的深度为 4

结点的层数：从根节点到树中某结点所经路径上的分支树称为该结点的层数，根节点的层数规定为 1，其余结点的层数等于其父亲结点的层数+1

同代：在同一棵树中具有相同层数的节点

4.3.2 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。

4.3.2 二叉树的遍历

- **前序遍历：中左右（根左右）**

  即先访问根结点，再前序遍历左子树，最后再前序遍历右子 树。前序遍历运算访问二叉树各结点是以根、左、右的顺序进行访问的。

- **中序遍历：左中右（左根右）**

  即先中前序遍历左子树，然后再访问根结点，最后再中序遍 历右子树。中序遍历运算访问二叉树各结点是以左、根、右的顺序进行访问的。

- **后序遍历：左右中（左右根）**

  即先后序遍历左子树，然后再后序遍历右子树，最后访问根 结点。后序遍历运算访问二叉树各结点是以左、右、根的顺序进行访问的。

前序遍历：ABDHIECFG

中序遍历：HDIBEAFCG

后序遍历：HIDEBFGCA

4.3.3 经典二叉树

image-20220521153016348

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、二叉排序/查找/搜索树：即为 BST (binary search/sort tree)。满足如下性质：（1）若它的左子树不为空，则左子树上所有结点的值均小于它的根节点的值；（2）若它的右子树上所有结点的值均大于它的根节点的值； （3）它的左、右子树也分别为二叉排序树。

image-20220521145208018

对二叉查找树进行中序遍历，得到有序集合。便于检索。

4、平衡二叉树：（Self-balancing binary search tree）首先是二叉排序树，此外具有以下性质：（1）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（2）并且左右两个子树也都是一棵平衡二叉树 （3）不要求非叶节点都有两个子结点 。 平衡二叉树的目的是为了减少二叉查找树层次，提高查找速度。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。

尚硅谷

image-20220521150151219

6、红黑树：R-B Tree，全称是 Red-Black Tree。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它是在 1972 年由 Rudolf Bayer 发明的，他称之为"对称二叉 B 树"，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的一篇论文中获得的。它是复杂的，但它的操作有着良好的最坏情况运行时 间，并且在实践中是高效的: 它可以在 O(log n)时间内做查找，插入和删除， 这里的 n 是树中元素的数目。

红黑树的特性：

- 每个节点是红色或者黑色
- 根节点是黑色
- 每个叶子节点（NIL）是黑色。（注意：这里叶子节点，是指为空(NIL 或 NULL)的叶子节点！）
- 每个红色节点的两个子节点都是黑色的。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点（确保没有一条路径会比其他路径长出俩倍）

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。

红黑树可以通过红色节点和黑色节点尽可能的保证二叉树的平衡。主要是用它来存储有序的数据，它的时间复杂度是 O(logN)，效率非常之高。

4.3.4 二叉树及其结点的表示

普通二叉树：

```
public class BinaryTree<E>{
    private TreeNode root; //二叉树的根结点
    private int total;//结点总个数

    private class TreeNode{
        //至少有以下几个部分
        TreeNode parent;
        TreeNode left;
        E data;
        TreeNode right;

        public TreeNode(TreeNode parent, TreeNode left, E data, TreeNod
e 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;
        }
    }
}
```

第 15 章_File 类与 IO 流

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

1. java.io.File 类的使用

1.1 概述

java.io.File 类：文件和文件目录路径的抽象表示形式，与平台无关。一个 File 对象代表硬盘或网络中可能存在的一个文件或者目录。

File 能新建、删除、重命名文件和目录，但 File 不能访问文件内容本身。如果需要访问文件内容本身，则需要使用输入/输出流。

想要在 Java 程序中表示一个真实存在的文件或目录，那么必须有一个 File 对象，但是 Java 程序中的一个 File 对象，可能没有一个真实存在的文件或目录。

File 对象可以作为参数传递给流的构造器。

1.2 构造器

- `public File(String pathname)`：以 pathname 为路径创建 File 对象，可以是绝对路径或者相对路径，如果 pathname 是相对路径，则默认的当前路径在系统属性 user.dir 中存储。
  - **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。
- `public File(String parent, String child)`：以 parent 为父路径，child 为子路径创建 File 对象。
- `public File(File parent, String child)`：根据一个父 File 对象和子文件路径创建 File 对象
- 构造举例，代码如下：

```
package com.atguigu.file;

import java.io.File;
```

```java
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);
    }

    @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/目录的父目录名: " + f3.getParent());
    }
    @Test
    public void test04() 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/目录的父目录名: " + f5.getParent());
    }
}
```

注意：

1. 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。
2. 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。比如：

   File file2 = new File("d:" + File.separator + "atguigu" + File.separator + "info.txt");
3. 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

1.3 常用方法

1、获取文件和目录基本信息

- public String getName() ：获取名称
- public String getPath() ：获取路径
- public String getAbsolutePath()：获取绝对路径
- public File getAbsoluteFile()：获取绝对路径表示的文件
- public String getParent()：获取上层文件目录路径。若无，返回 null
- public long length() ：获取文件长度（即：字节数）。不能获取目录的长度。

- public long lastModified()：获取最后一次的修改时间，毫秒值

  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。

- 演示代码：

```
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.ofI
nstant(Instant.ofEpochMilli(f.lastModified()),ZoneId.of("Asia/Sha
nghai")));

        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.ofI
nstant(Instant.ofEpochMilli(f.lastModified()),ZoneId.of("Asia/Sha
nghai")));
```

```
        }
    }
```

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

2、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。

```java
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);
        }
    }

}
```

3、File 类的重命名功能

- public boolean renameTo(File dest):把文件重命名为指定的文件路径

4、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。
- public boolean canRead()：判断是否可读
- public boolean canWrite()：判断是否可写
- public boolean isHidden()：判断是否隐藏

演示代码：

```
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

5、创建、删除功能

- `public boolean createNewFile()`：创建文件。若文件存在，则不创建，返回 false。
- `public boolean mkdir()`：创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。
- `public boolean mkdirs()`：创建文件目录。如果上层文件目录不存在，一并创建。
- `public boolean delete()`：删除文件或者文件夹

删除注意事项：① Java 中的删除不走回收站。② 要删除一个文件目录，请注意该文件目录内不能包含文件或者文件目录。

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。

1.4 练习

练习 1：利用 File 构造器，new 一个文件目录 file

1) 在其中创建多个文件和目录

2) 编写方法，实现删除 file 中指定文件的操作

练习 2：判断指定目录下是否有后缀名为.jpg 的文件，如果有，就输出该文件名称

```
public class FindJPGFileTest {

	@Test
	public void test1(){
		File srcFile = new File("d:\\code");

		String[] fileNames = srcFile.list();
		for(String fileName : fileNames){
			if(fileName.endsWith(".jpg")){
				System.out.println(fileName);
			}
		}
	}
	@Test
	public void test2(){
		File srcFile = new File("d:\\code");

		File[] listFiles = srcFile.listFiles();
		for(File file : listFiles){
			if(file.getName().endsWith(".jpg")){
				System.out.println(file.getAbsolutePath());
			}
		}
	}
	/*
	 * File 类提供了两个文件过滤器方法
	 * public String[] list(FilenameFilter filter)
```

```
 * public File[] listFiles(FileFilter filter)

 */
@Test
public void test3(){
      File srcFile = new File("d:\\code");

      File[] subFiles = srcFile.listFiles(new FilenameFilter() {

            @Override
            public boolean accept(File dir, String name) {
                  return name.endsWith(".jpg");
            }
      });

      for(File file : subFiles){
            System.out.println(file.getAbsolutePath());
      }
}

}
```

练习 3：遍历指定目录所有文件名称，包括子文件目录中的文件。

拓展 1：并计算指定目录占用空间的大小

拓展 2：删除指定文件目录及其下的所有文件

```
public class ListFilesTest {

      public static void main(String[] args) {
            // 递归:文件目录
            /** 打印出指定目录所有文件名称，包括子文件目录中的文件 */

            // 1.创建目录对象
            File dir = new File("E:\\teach\\01_javaSE\\_尚硅谷 Java 编程语言\\3_软件");

            // 2.打印目录的子文件
            printSubFile(dir);
      }

      public static void printSubFile(File dir) {
            // 打印目录的子文件
            File[] subfiles = dir.listFiles();

            for (File f : subfiles) {
```

```java
            if (f.isDirectory()) {// 文件目录
                printSubFile(f);
            } else {// 文件
                System.out.println(f.getAbsolutePath());
            }

        }
    }

    // 方式二：循环实现
    // 列出file目录的下级内容，仅列出一级的话
    // 使用File类的String[] list()比较简单
    public void listSubFiles(File file) {
        if (file.isDirectory()) {
            String[] all = file.list();
            for (String s : all) {
                System.out.println(s);
            }
        } else {
            System.out.println(file + "是文件！");
        }
    }

    // 列出file目录的下级，如果它的下级还是目录，接着列出下级的下级，依次类推
    // 建议使用File类的File[] listFiles()
    public void listAllSubFiles(File file) {
        if (file.isFile()) {
            System.out.println(file);
        } else {
            File[] all = file.listFiles();
            // 如果all[i]是文件，直接打印
            // 如果all[i]是目录，接着再获取它的下一级
            for (File f : all) {
                listAllSubFiles(f);// 递归调用：自己调用自己就叫递归
            }
        }
    }

    // 拓展1：求指定目录所在空间的大小
    // 求任意一个目录的总大小
    public long getDirectorySize(File file) {
        // file是文件，那么直接返回file.length()
        // file是目录，把它的下一级的所有大小加起来就是它的总大小
        long size = 0;
        if (file.isFile()) {
            size += file.length();
```

```
        } else {
            File[] all = file.listFiles();// 获取file 的下一级
            // 累加all[i]的大小
            for (File f : all) {
                size += getDirectorySize(f);// f 的大小;
            }
        }
        return size;
    }

    // 拓展2：删除指定的目录
    public void deleteDirectory(File file) {
        // 如果file 是文件，直接delete
        // 如果file 是目录，先把它的下一级干掉，然后删除自己
        if (file.isDirectory()) {
            File[] all = file.listFiles();
            // 循环删除的是file 的下一级
            for (File f : all) {// f 代表file 的每一个下级
                deleteDirectory(f);
            }
        }
        // 删除自己
        file.delete();
    }

}
```

2. IO 流原理及流的分类

Google I/O 寓为“开放中创新”
(Innovation in the Open)

Input/Output
二进制1,0

让天下没有难学的技术

2.1 Java IO 原理

- 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。

- Java 程序中，对于数据的输入/输出操作以“流(stream)” 的方式进行，可以看做是一种数据的流动。

image-20220503123117300

- `java.io` 包下提供了各种“流”类和接口，用以获取不同种类的数据，并通过标准的方法输入或输出数据。
- `I/O` 是 `Input/Output` 的缩写，`I/O` 技术是非常实用的技术，用于处理设备之间的数据传输。如读/写文件，网络通讯等。
  - 输入 `input`：读取外部数据（磁盘、光盘等存储设备的数据）到程序（内存）中。
  - 输出 `output`：将程序（内存）数据输出到磁盘、光盘等存储设备中。

2.2 流的分类

- 按数据的流向不同分为：**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
- 按操作数据单位的不同分为：**字节流（8bit）**和**字符流（16bit）**。
  - **字节流**：以字节为单位，读写数据的流。
    - 以 InputStream、OutputStream 结尾
  - **字符流**：以字符为单位，读写数据的流。

- 以 Reader、Writer 结尾

- 根据 IO 流的角色不同分为：**节点流**和**处理流**。
  - **节点流**：直接从数据源或目的地读写数据

  - **处理流**：不直接连接到数据源或目的地，而是“连接”在已存在的流（节点流或处理流）之上，通过对数据的处理为程序提供更为强大的读写功能。

小结：图解

image-20220412225253349

2.3 流的 API

- Java 的 IO 流共涉及 40 多个类，实际上非常规则，都是从如下 4 个抽象基类派生的。

（抽象基类）	输入流	输出流
字节流	InputStream	OutputStream
字符流	Reader	Writer

- 由这四个类派生出来的子类名称都是以其父类名作为子类名后缀。

分类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	InputStream	OutputStream	Reader	Writer
访问文件	FileInputStream	FileOutputStream	FileReader	FileWriter
访问数组	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
访问管道	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
访问字符串			StringReader	StringWriter
缓冲流	BufferedInputStream	BufferedOuputStream	BufferedReader	BufferedWriter
转换流			InputStreamReader	OutputStreamWriter
对象流	ObjectInputStream	ObjectOutputStream		
	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
打印流		PrintStream		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

常用的节点流：

- 文件流： FileInputStream、FileOutputStrean、FileReader、FileWriter
- 字节/字符数组流： 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）。

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. 节点流之二：FileReader\FileWriter

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。

3.1 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。

3.2 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
```

```java
        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
         */
    }

}
```

3.3 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。

3.4 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。

3.4.1 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();
    }
```

```java
    @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();
    }
}
```

3.4.2 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。

```java
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();
    }
}
```

3.4.3 换行

```
package com.atguigu.fileio;

import java.io.FileWriter;
import java.io.IOException;

public class FWWriteNewLine {
    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();
    }
}
```

3.4.4 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。

3.5 小结

- 定义文件路径时，注意：可以用“/”或者“\”。
- 在写入一个文件时，如果使用构造器 FileOutputStream(file)，则目录下有同名文件将被覆盖。 如果使用构造器 FileOutputStream(file,true)，则目录下的同名文件不会被覆盖，在文件内容末尾追加内容。
- 在读取文件时，必须保证该文件已存在，否则报异常。
- 字节流操作字节，比如：.mp3，.avi，.rmvb，mp4，.jpg，.doc，.ppt
- 字符流操作字符，只能操作普通文本文件。最常见的文本文件：.txt，.java，.c，.cpp 等语言的源代码。尤其注意.doc,excel,ppt 这些不是文本文件。

4. 节点流之一：FileInputStream\FileOutputStream

4.1 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。

4.2 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。

4.3 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。

4.3.1 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();
    }
}
```

4.3.2 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;
```

```java
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
```

4.3.3 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：

```java
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*/
```

4.4 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。

4.5 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);
        read = fis.read();
        System.out.println((char) read);
        read = fis.read();
        System.out.println((char) read);
        read = fis.read();
        System.out.println((char) read);
        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
```

```java
        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
         */
    }
}
```

4.6 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");
```

```java
        // 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();
    }
}
```

5. 处理流之一：缓冲流

为了提高数据读写的速度，Java API 提供了带缓冲功能的流类：缓冲流。

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（缺省使用 8192 个字节(8Kb)的缓冲区），通过缓冲区读写，减少系统 IO 次数，从而提高读写的效率。

```java
public
class BufferedInputStream extends FilterInputStream {

    private static int DEFAULT_BUFFER_SIZE = 8192;
```

缓冲流要“套接”在相应的节点流之上，根据数据操作单位可以把缓冲流分为：

- **字节缓冲流**：BufferedInputStream，BufferedOutputStream
- **字符缓冲流**：BufferedReader，BufferedWriter

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 FileOutputStrea
m("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"));
```

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 FileInput
Stream("jdk8.exe"));
        BufferedOutputStream bos = new BufferedOutputStream(new FileOut
```

```
putStream("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
");
    }
}
```

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();
    }
```

```java
    @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();
    }
}
```

5.4 流的关闭顺序

```java
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
         */
    }
```

```java
    @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        穿内衣
        BufferedWriter bw = new BufferedWriter(fw); //外面        穿外套

        关闭
        bw.close();  //先关外面的                                先脱外套
        fw.close(); //再关里面的                                 再脱内衣
         */
    }
}
```

5.5 练习

练习 1：分别使用节点流：FileInputStream、FileOutputStream 和缓冲流：BufferedInputStream、BufferedOutputStream 实现文本文件/图片/视频文件的复制。并比较二者在数据复制方面的效率。

练习 2：实现图片加密操作。

提示：

```java
int b = 0;
while((b = fis.read()) != -1){
    fos.write(b ^ 5);
}
```

练习 3：获取文本上每个字符出现的次数

提示：遍历文本的每一个字符；字符及出现的次数保存在 Map 中；将 Map 中数据写入文件

6. 处理流之二：转换流

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（英文大小写字符、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。

- 举例

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

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]

- 小结

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？

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.t
xt"));
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 FileInputStr
eam(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 FileInputStr
eam(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();
    }
}
```

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("o
ut.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 FileOutputS
tream(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 FileOutput
Stream(FileName2),"GBK");
        // 写出数据
        osw2.write("你好");// 保存为4 个字节
```

```
        osw2.close();
    }
}
```

6.5 转换流理解图解

转换流是字节与字符间的桥梁！

6.6 练习

把课件文件夹下的《我想对你说.txt》字符编码为 GBK，复制到当前项目的 testIO 文件夹下的《宋老师的话.txt》字符编码为 UTF-8

```
package com.atguigu.test04;

import java.io.BufferedInputStream;
```

```java
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

public class Test04 {
	public static void main(String[] args) {
		try(
			FileInputStream fis = new FileInputStream("d:/atguigu/day22_homework/我想对你说.txt");
			BufferedInputStream bis = new BufferedInputStream(fis);
			InputStreamReader isr = new InputStreamReader(bis,"GBK");

			FileOutputStream fos = new FileOutputStream("testIO/宋老师的话.txt");
			BufferedOutputStream bos = new BufferedOutputStream(fos);
			OutputStreamWriter osw = new OutputStreamWriter(bos, "UTF-8");
		){
			char[] data = new char[1024];
			int len;
			while((len = isr.read(data))!=-1){
				osw.write(data, 0, len);
			}
		}catch(IOException e){
			e.printStackTrace();
		}
	}
}
```

7. 处理流之三/四：数据流、对象流

7.1 数据流与对象流说明

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？

```java
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：

- **数据流：DataOutputStream、DataInputStream**
  - 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。
  - 对象流 DataInputStream 中的方法：

    ```
    boolean readBoolean()          byte readByte()
    char readChar()                     float readFloat()
    double readDouble()                 short readShort()
    long readLong()                     int readInt()
    String readUTF()             void readFully(byte[] b)
    ```

  - 对象流 DataInputStream 中的方法：将上述的方法的 read 改为相应的 write 即可。
- **对象流：ObjectOutputStream、ObjectInputStream**
  - 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);
```

7.2 使用对象流读写各种类型的数据

对象流的强大之处就是可以把 Java 中的对象写入到数据源中，也能把对象从数据源中还原回来。

ObjectOutputStream 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 从 InputStream 父类中继承的方法：

- `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。

此外，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.*;
```

```java
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 = true;

        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputS
tream("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 FileInputStre
am("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();
    }
}
```

7.3 认识对象序列化机制

对象序列化机制允许把内存中的 Java 对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。//当其它程序获取了这种二进制流，就可以恢复成原来的 Java 对象。

- 序列化过程：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。

- 反序列化过程：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。

image-20220503123328452

序列化是 RMI（Remote Method Invoke – 远程方法调用）过程的参数和返回值都必须实现的机制，而 RMI 是 JavaEE 的基础。因此序列化机制是 JavaEE 平台的基础。

序列化的好处，在于可将任何实现了 Serializable 接口的对象转化为**字节数据**，使其在保存和传输时可被还原。

图理解序列化机制：

- 序列化：用 ObjectOutputStream 类保存基本类型数据或对象的机制。方法为：
  - `public final void writeObject (Object obj)`：将指定的对象写出。
- 反序列化：用 ObjectInputStream 类读取基本类型数据或对象的机制。方法为：
  - `public final Object readObject ()`：读取一个对象。

7.4 如何实现序列化机制

如果需要让某个对象支持序列化机制，则必须让对象所属的类及其属性是可序列化的，为了让某个类是可序列化的，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（static）变量的值不会序列化。因为静态变量的值不属于某个对象。

举例：

```
package com.atguigu.object;

import java.io.Serializable;

public class Employee implements Serializable {
    //static final long serialVersionUID = 23234234234L;
    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;
```

```java
    }

    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("小谷姐姐", "宏福苑", 23);
        // 创建序列化流对象
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputS
tream("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 FileOu
tputStream("str1.dat"));
            oos1.writeUTF("atguigu");
            oos1.writeInt(1);
            oos1.close();

            ObjectOutputStream oos2 = new ObjectOutputStream(new FileOu
tputStream("str2.dat"));
            oos2.writeObject("atguigu");
            oos2.writeObject(1);
            oos2.close();
    }
}
```

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` 。凡是实现 Serializable 接口的类都有一个表示序列化版本标识符的静态变量：

```
static final long serialVersionUID = 234242343243L; //它的值由程序员随意指定即可。
```

- serialVersionUID 用来表明类的不同版本间的兼容性。简单来说，Java 的序列化机制是通过在运行时判断类的 serialVersionUID 来验证版本一致性的。在进行反序列化时，JVM 会把传来的字节流中的 serialVersionUID 与本地相应实体类的 serialVersionUID 进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常(InvalidCastException)。
- 如果类没有显示定义这个静态常量，它的值是 Java 运行时环境根据类的内部细节自动生成的。若类的实例变量做了修改，serialVersionUID 可能发生变化。故建议，显式声明。
- 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 = 1324234L; //增加serial
VersionUID
    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 +
                '}';
    }
}
```

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 FileOutputS
tream("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);
    }
}
```

7.6 练习

面试题：谈谈你对 java.io.Serializable 接口的理解，我们知道它用于序列化，是空方法接口，还有其它认识吗？

实现了 Serializable 接口的对象，可将它们转换成一系列字节，并可在以后完全恢复回原来的样子。这一过程亦可通过网络进行。这意味着序列化机制能自动补偿操作系统间的差异。换句话说，可以先在 Windows 机器上创建一个对象，对其序列化，然后通过网络发给一台 Unix 机器，然后在那里准确无误地重新“装配”。不必关心数据在不同机器上如何表示，也不必关心字节的顺序或者其他任何细节。

由于大部分作为参数的类如 String、Integer 等都实现了 java.io.Serializable 的接口，也可以利用多态的性质，作为参数使接口更灵活。

练习：

- 需求说明：
  - 网上购物时某用户填写订单，订单内容为产品列表，保存在“save.bin”中。
  - 运行时，如果不存在“save.bin”，则进行新订单录入，如果存在，则显示并计算客户所需付款。

- 分析：
  - 编写 Save()方法保存对象到“save.bin”
  - 编写 Load()方法获得对象，计算客户所需付款

```
请输入产品名：可乐
请输入单价：2
请输入数量：10
是否继续（y/n）：y
请输入产品名：薯片
请输入单价：3.5
请输入数量：3
是否继续（y/n）：n
订单已保存
```

```
产品名      单价      数量
可乐         2.00      10
薯片         3.50      3
订单总价：30.50
```

image-20220503123603551

8. 其他流的使用

8.1 标准输入、输出流

- System.in 和 System.out 分别代表了系统标准的输入和输出设备
- 默认输入设备是：键盘，输出设备是：显示器
- System.in 的类型是 InputStream
- System.out 的类型是 PrintStream，其是 OutputStream 的子类 FilterOutputStream 的子类
- 重定向：通过 System 类的 setIn，setOut 方法对默认设备进行改变。
  - public static void setIn(InputStream in)
  - public static void setOut(PrintStream out)

举例：

从键盘输入字符串，要求将读取到的整行字符串转成大写输出。然后继续进行输入操作，直至当输入“e”或者“exit”时，退出程序。

```
System.out.println("请输入信息(退出输入 e 或 exit):");
// 把"标准"输入流(键盘输入)这个字节流包装成字符流,再包装成缓冲流
BufferedReader br = new BufferedReader(new InputStreamReader(System.i
n));
```

```java
String s = null;
try {
    while ((s = br.readLine()) != null) { // 读取用户输入的一行数据 -->
阻塞程序
        if ("e".equalsIgnoreCase(s) || "exit".equalsIgnoreCase(s)) {
            System.out.println("安全退出!!");
            break;
        }
        // 将读取到的整行字符串转成大写输出
        System.out.println("-->:" + s.toUpperCase());
        System.out.println("继续输入信息");
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (br != null) {
            br.close(); // 关闭过滤流时,会自动关闭它包装的底层节点流
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

拓展：

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：

```java
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。

```java
    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);
```

练习：

Create a program named MyInput.java: Contain the methods for reading int, double, float, boolean, short, byte and String values from the keyboard.

```
Scanner s = new Scanner(System.in);
s.nextInt();
s.next();
```

```
class MyInput{
  Scanner s = new Scanner(System.in);

 public String readString(){
    return s.next();
 }
 ....

}
```

image-20220412232254819

```java
package com.atguigu.java;
// MyInput.java: Contain the methods for reading int, double, float, bo
olean, short, byte and
// string values from the keyboard

import java.io.*;

public class MyInput {
    // Read a string from the keyboard
    public static String readString() {
        BufferedReader br = new BufferedReader(new InputStreamReader(Sy
```

```
stem.in));

        // Declare and initialize the string
        String string = "";

        // Get the string from the keyboard
        try {
            string = br.readLine();

        } catch (IOException ex) {
            System.out.println(ex);
        }

        // Return the string obtained from the keyboard
        return string;
    }

    // Read an int value from the keyboard
    public static int readInt() {
        return Integer.parseInt(readString());
    }

    // Read a double value from the keyboard
    public static double readDouble() {
        return Double.parseDouble(readString());
    }

    // Read a byte value from the keyboard
    public static double readByte() {
        return Byte.parseByte(readString());
    }

    // Read a short value from the keyboard
    public static double readShort() {
        return Short.parseShort(readString());
    }

    // Read a long value from the keyboard
    public static double readLong() {
        return Long.parseLong(readString());
    }

    // Read a float value from the keyboard
    public static double readFloat() {
        return Float.parseFloat(readString());
    }
}
```

8.2 打印流

- 实现将基本数据类型的数据格式转化为字符串输出。
- 打印流：`PrintStream` 和 `PrintWriter`
  - 提供了一系列重载的 print()和 println()方法，用于多种数据类型的输出

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

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

- PrintStream 和 PrintWriter 的输出不会抛出 IOException 异常
- PrintStream 和 PrintWriter 有自动 flush 功能
- PrintStream 打印的所有字符都使用平台的默认字符编码转换为字节。在需要写入字符而不是写入字节的情况下，应该使用 PrintWriter 类。
- System.out 返回的是 PrintStream 的实例

- 构造器
  - 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。

- 代码举例 1

```
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();
```

```
    }
}
```

- 代码举例 2

```
PrintStream ps = null;
try {
    FileOutputStream fos = new FileOutputStream(new File("D:\\IO\\text.
txt"));
    // 创建打印输出流,设置为自动刷新模式(写入换行符或字节 '\n' 时都会刷新输出
缓冲区)
    ps = new PrintStream(fos, true);
    if (ps != null) {// 把标准输出流(控制台输出)改成文件
        System.setOut(ps);
    }
    for (int i = 0; i <= 255; i++) { // 输出ASCII 字符
        System.out.print((char) i);
        if (i % 50 == 0) { // 每50 个数据一行
            System.out.println(); // 换行
        }
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (ps != null) {
        ps.close();
    }
}
```

- 代码举例 3：自定义一个日志工具

```
/*
日志工具
 */
public class Logger {
    /*
    记录日志的方法。
     */
    public static void log(String msg) {
        try {
            // 指向一个日志文件
            PrintStream out = new PrintStream(new FileOutputStream("log.
txt", true));
            // 改变输出方向
            System.setOut(out);
            // 日期当前时间
            Date nowTime = new Date();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:
mm:ss SSS");
```

```
            String strTime = sdf.format(nowTime);

            System.out.println(strTime + ": " + msg);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}

public class LogTest {
    public static void main(String[] args) {
        //测试工具类是否好用
        Logger.log("调用了 System 类的 gc()方法，建议启动垃圾回收");
        Logger.log("调用了 TeamView 的 addMember()方法");
        Logger.log("用户尝试进行登录，验证失败");
    }
}
```

8.3 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;
```

```java
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();
    }
}
```

9. apache-common 包的使用

在导入 commons-io-2.5.jar 包之后，内部的 API 都可以使用。

FileUtil 类的使用

- public static void writeStringToFile(File file,String content)：将内容 content 写入到 file 中
- public static String readFileToString(File file)：读取文件内容，并返回一个 String
- public static void copyFile(File srcFile,File destFile)：文件复制

- public static void copyDirectoryToDirectory(File srcDir,File destDir)：文件夹复制

举例：

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

        try {
            FileUtils.copyFile(new File("day23\\yangm.png"),new File("day23\\yangm2.png"));
            System.out.println("复制成功");
        } catch (IOException e) {
            e.printStackTrace();

        }
    }
```

10. JDK1.7 之后引入新 try..catch

10.1 IO 流关闭和异常处理

```
package com.atguigu.io;

import org.junit.Test;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class TestIOClose {

    @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();
            }
        }
    }

}
```

10.2 JDK1.7 之后引入新 try..catch

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){
    业务逻辑代码
}catch(异常类型 1 e){
    处理异常代码
}catch(异常类型 2 e){
    处理异常代码
```

```
}
....
```

这个形式的 try...catch，没有 finally。这种格式可以保证在 try()中声明的资源，无论是否发生异常，无论是否处理异常，都不需要程序员去关闭资源对象，都会自动关闭资源对象。

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,"GB
K");
                  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();
            }
      }

    @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();
        }
    }
```

第 16 章_网络编程

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 网络编程概述

Java 是 Internet 上的语言，它从语言级上提供了对网络应用程序的支持，程序员能够很容易开发常见的网络应用程序。

Java 提供的网络类库，可以实现无痛的网络连接，联网的底层细节被隐藏在 Java 的本机安装系统里，由 JVM 进行控制。并且 Java 实现了一个跨平台的网络库，程序员面对的是一个统一的网络编程环境。

1.1 软件架构

- **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 Q Q、美团 app、360 安全卫士等软件。

尚硅谷

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。

尚硅

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。

1.2 网络基础

- **计算机网络：** 把分布在不同地理区域的计算机与专门的外部设备用通信线路互连成一个规模大、功能强的网络系统，从而使众多的计算机可以方便地互相传递信息、共享硬件、软件、数据信息等资源。
- **网络编程的目的：**直接或间接地通过网络协议与其它计算机实现数据交换，进行通讯。
- **网络编程中有三个主要的问题：**
  - 问题 1：如何准确地定位网络上一台或多台主机
  - 问题 2：定位主机上的特定的应用
  - 问题 3：找到主机后如何可靠高效地进行数据传输

地球村

image-20220415001614195

2. 网络通信要素

2.1 如何实现网络中的主机互相通信

- 通信双方地址
  - IP
  - 端口号
- 一定的规则：不同的硬件、操作系统之间的通信，所有的这一切都需要一种规则。而我们就把这种规则称为协议，即网络通信协议。

生活类比：

image-20220503144420721

image-20220503144450241

2.2 通信要素一：IP 地址和域名

2.2.1 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，即唯一的通信实体。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，以点分十进制表示，例如 `192.168.65.100` 。其中 a、b、c、d 都是 0~255 之间的十进制整数。

  –

image-20220503145929568

- 这种方式最多可以表示 42 亿个。其中，30 亿都在北美，亚洲 4 亿，中国 2.9 亿。2011 年初已经用尽。
- IP 地址 = 网络地址 +主机地址
  - 网络地址：标识计算机或网络设备所在的网段
  - 主机地址：标识特定主机或网络设备

类型	范 围
A	0.0.0.0 到 127.255.255.255
B	128.0.0.0 到 191.255.255.255
C	192.0.0.0 到 223.255.255.255
D	224.0.0.0 到 239.255.255.255
E	240.0.0.0 到 247.255.255.255

其中，E 类用于科研。

- `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，共 16 个字节，写成 8 个无符号整数，每个整数用四个十六进制位表示，数之间用冒号（：）分开。比如：
  `ABCD:EF01:2345:6789:ABCD:EF01:2345:6789`，按保守方法估算 IPv6 实际可分配的地址，整个地球的每平方米面积上仍可分配 1000 多个地址，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。

在 IPv6 的设计过程中除了一劳永逸地解决了地址短缺问题以外，还考虑了在 IPv4 中解决不好的其它问题，主要有端到端 IP 连接、服务质量（QoS）、安全性、多播、移动性、即插即用等。

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.开头的就是私有址址，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`

2.2.2 域名

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

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，Domain Name System，域名系统)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

简单理解：

image-20220415003729013

详细理解：

1. 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。

2. 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。

3. 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。

4. 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。

5. 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。

6. 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。

2.3 通信要素二：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）。

不同的进程，设置不同的端口号。

- **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。

image-20220415004400166

2.4 通信要素三：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。

- 网络通信协议：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、出错控制等做了统一规定，通信双方必须同时遵守才能完成数据交换。

新的问题：网络协议涉及内容太多、太复杂。如何解决？

计算机网络通信涉及内容很多，比如指定源地址和目标地址，加密解密，压缩解压缩，差错控制，流量控制，路由控制，如何实现如此复杂的网络协议呢？通信协议分层思想。

在制定协议时，把复杂成份分解成一些简单的成份，再将它们复合起来。最常用的复合方式是层次方式，即同层间可以通信、上一层可以调用下一层，而与再下一层不发生关系。各层互不影响，利于系统的开发和扩展。

这里有两套参考模型

- 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。

OSI参考模型	各层作用	TCP/IP参考模型	TCP/IP参考模型各层对应协议	对应结构
应用层	为应用程序提供服务	应用层	HTTP、FTP、Telnet、DNS…	应用程序
表示层	数据格式转化、数据加密			
会话层	建立、管理和维护会话			
传输层	建立、管理和维护端对端的连接	传输层	TCP、UDP、…	操作系统
网络层	IP选址及路由选择	网络层	IP、ICMP、ARP…	
数据链路层	提供介质访问和链路管理	物理+数据链路层	Link	设备驱动程序、网络接口
物理层	物理传输			

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：** 传输控制协议/因特网互联协议(Transmission Control Protocol/Internet Protocol)，TCP/IP 以其两个主要协议：传输控制协议(TCP)和网络互联协议(IP)而得名，实际上是一组协议，包括多个具有不同功能且互为关联的协议。是 Internet 最基本、最广泛的协议。

图：TCP/IP 是互联网相关的各类协议族的总称

image-20220512234904047

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分别是应用层、传输层、网络层和物理+数据链路层，每层分别负责不同的通信功能。

TCP/IP 协议中的四层介绍：

- 应用层：应用层决定了向用户提供应用服务时通信的活动。主要协议有：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。

- 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。

- 网络层：网络层是整个 TCP/IP 协议的核心，支持网间互连的数据通信。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 protocal）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。

- 物理+数据链路层：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。

2. 谈传输层协议：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)。

2.1 TCP 协议与 UDP 协议

TCP 协议：

- 使用 TCP 协议前，须先建立 TCP 连接，形成基于字节流的传输数据通道
- 传输前，采用“三次握手”方式，点对点通信，是可靠的
- 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。
- TCP 协议进行通信的两个应用进程：客户端、服务端。
- 在连接中可进行大数据量的传输
- 传输完毕，需释放已建立的连接，效率低

UDP 协议：

- 将数据、源、目的封装成数据包（传输的基本单位），不需要建立连接
- 每个数据报的大小限制在 64K 内
- 发送不管对方是否准备好，接收方收到也不确认，不能保证数据的完整性，故是不可靠的
- 可以广播发送
- 发送数据结束时无需释放资源，开销小，通信效率高
- 适用场景：音频、视频和普通数据的传输例如视频会议

TCP 生活案例：打电话

UDP 生活案例：发送短信、发电报

2.2 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。

image-20220415010105484

1.客户端通过向服务器端发送一个 SYN 来创建一个主动打开，作为三路握手的一部分。客户端把这段连接的序号设定为随机数 A。

2.服务器端应当为一个合法的 SYN 回送一个 SYN/ACK。ACK 的确认码应为 A+1，SYN/ACK 包本身又有一个随机序号 B。

3.最后，客户端再发送一个 ACK。当服务端受到这个 ACK 的时候，就完成了三路握手，并进入了连接创建状态。此时包序号被设定为收到的确认号 A+1，而响应则为 B+1。

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。

2.3 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。

image-20220415010205891

客户端或服务器均可主动发起挥手动作，在 socket 编程中，任何一方执行 close()操作即可产生挥手操作。（1）客户端 A 发送一个 FIN，用来关闭客户 A 到服务器 B 的数据传送。（2）服务器 B 收到这个 FIN，它发回一个 ACK，确认序号为收到序号加 1。和 SYN 一样，一个 FIN 将占用一个序号。（3）服务器 B 关闭与客户端 A 的连接，发送一个 FIN 给客户端 A。（4）客户端 A 发回 ACK 报文确认，并将确认序号设置为收到序号加 1。

3. 网络编程 API

3.1 InetAddress 类

InetAddress 类主要表示 IP 地址，两个子类：Inet4Address、Inet6Address。

InetAddress 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 getByName(String host)
- public static InetAddress getByAddress(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
- public boolean isReachable(int timeout)：测试是否可以达到该地址

```
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);
```

```
    }
}
```

192.168.24.56

1100 0000	192
1010 1000	168
0001 1000	24
0011 1000	56

内部用一个int存储

1100 0000 1010 1000 0001 1000 0011 1000

3.2 Socket 类

- 利用套接字(Socket)开发网络应用程序早已被广泛的采用，以至于成为事实上的标准。网络通信其实就是 Socket 间的通信。
- 网络上具有唯一标识的 IP 地址和端口号组合在一起才能构成唯一能识别的标识符套接字（Socket）。
- 通信的两端都要有 Socket，是两台机器间通信的端点。
- Socket 允许程序把网络连接当成一个流，数据在两个 Socket 间通过 IO 传输。
- 一般主动发起通信的应用程序属客户端，等待通信请求的为服务端。
- Socket 分类：
  - 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。

3.3 Socket 相关类 API

3.3.1 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。

3.3.2 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。关闭此套接字也将会关闭该套接字的 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）。 即不能在从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 Scoket 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。

3.3.3 DatagramSocket 类

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。

- public InetAddress getLocalAddress()获取套接字绑定的本地地址。

- public int getLocalPort()返回此套接字绑定的本地主机上的端口号。

- public InetAddress getInetAddress()返回此套接字连接的地址。如果套接字未连接，则返回 null。

- public int getPort()返回此套接字的端口。如果套接字未连接，则返回 -1。

3.3.4 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 InetAddress getAddress()返回某台机器的 IP 地址，此数据报将要发往该机器或者是从该机器接收到的。
- public int getPort()返回某台远程主机的端口号，此数据报将要发往该主机或者是从该主机接收到的。
- `public byte[] getData()`返回数据缓冲区。接收到的或将要发送的数据从缓冲区中的偏移量 offset 处开始，持续 length 长度。
- `public int getLength()`返回将要发送或接收到的数据的长度。

4. TCP 网络编程

4.1 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：

尚硅谷

基于TCP的Socket通信

4.2 开发步骤

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

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。

4.3 例题与练习

例题 1：客户端发送内容给服务端，服务端将内容打印到控制台上。

例题 2：客户端发送文件给服务端，服务端将文件保存在本地。

例题 3：从客户端发送文件给服务端，服务端保存到本地。并返回“发送成功”给客户端。并关闭相应的连接。

练习 1：服务端读取图片并发送给客户端，客户端保存图片到本地

练习 2：客户端给服务端发送文本，服务端会将文本转成大写在返回给客户端。

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
```

```java
        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();
    }
}
```

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

image-20220514173031651

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;
```

```java
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.getIne
tAddress().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 InputStreamR
eader(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,encodin
g));
        }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();
    }
}
```

4.3 案例：聊天室

服务端：

```
package com.atguigu.tcp;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;

public class TestChatServer {
      //这个集合用来存储所有在线的客户端
      static ArrayList<Socket> online = new  ArrayList<Socket>();

      public static void main(String[] args)throws Exception {
            //1、启动服务器，绑定端口号
            ServerSocket server = new ServerSocket(8989);

            //2、接收n多的客户端同时连接
            while(true){
                  Socket accept = server.accept();

                  online.add(accept);//把新连接的客户端添加到online 列表中
```

```
                MessageHandler mh = new MessageHandler(accept);
                mh.start();//
            }
        }

        static class MessageHandler extends Thread{
            private Socket socket;
            private String ip;

            public MessageHandler(Socket socket) {
                super();
                this.socket = socket;
            }

            public void run(){
                try {
                    ip = socket.getInetAddress().getHostAddress();

                    //插入：给其他客户端转发“我上线了”
                    sendToOther(ip+"上线了");

                    //(1)接收该客户端的发送的消息
                    InputStream input = socket.getInputStream();
                    InputStreamReader reader = new InputStreamReade
r(input);
                    BufferedReader br = new BufferedReader(reader);

                    String str;
                    while((str = br.readLine())!=null){
                        //(2)给其他在线客户端转发
                        sendToOther(ip+":"+str);
                    }

                    sendToOther(ip+"下线了");
                } catch (IOException e) {
                    try {
                        sendToOther(ip+"掉线了");
                    } catch (IOException e1) {
                        e1.printStackTrace();
                    }
                }finally{
                    //从在线人员中移除我
                    online.remove(socket);
                }
            }

            //封装一个方法：给其他客户端转发xxx 消息
            public void sendToOther(String message) throws IOException{
```

```
                //遍历所有的在线客户端，一一转发
                for (Socket on : online) {
                        OutputStream every = on.getOutputStream();
                        //为什么用PrintStream？目的用它的println 方法，按
行打印
                        PrintStream ps = new PrintStream(every);

                        ps.println(message);
                }
        }
    }
}
```

客户端：

```
package com.atguigu.tcp;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.util.Scanner;

public class TestChatClient {
    public static void main(String[] args)throws Exception {
        //1、连接服务器
        Socket socket = new Socket("127.0.0.1",8989);

        //2、开启两个线程
        //(1)一个线程负责看别人聊，即接收服务器转发的消息
        Receive receive = new Receive(socket);
        receive.start();

        //(2)一个线程负责发送自己的话
        Send send = new Send(socket);
        send.start();

        send.join();//等我发送线程结束了，才结束整个程序

        socket.close();
    }
}
class Send extends Thread{
    private Socket socket;

    public Send(Socket socket) {
```

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

        public void run(){
            try {
                OutputStream outputStream = socket.getOutputStream();
                //按行打印
                PrintStream ps = new PrintStream(outputStream);

                Scanner input = new Scanner(System.in);

                //从键盘不断的输入自己的话，给服务器发送，由服务器给其他人
转发
                while(true){
                    System.out.print("自己的话: ");
                    String str = input.nextLine();
                    if("bye".equals(str)){
                        break;
                    }
                    ps.println(str);
                }

                input.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

}
class Receive extends Thread{
        private Socket socket;

        public Receive(Socket socket) {
            super();
            this.socket = socket;
        }

        public void run(){
            try {
                InputStream inputStream = socket.getInputStream();
                Scanner input = new Scanner(inputStream);

                while(input.hasNextLine()){
                    String line = input.nextLine();
                    System.out.println(line);
                }
            } catch (IOException e) {
                e.printStackTrace();
```

```
            }
        }
}
```

4.4 理解客户端、服务端

- 客户端：
  - 自定义
  - 浏览器(browser --- server)
- 服务端：
  - 自定义
  - Tomcat 服务器

5. 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。

5.1 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。

类 DatagramSocket 和 DatagramPacket 实现了基于 UDP 协议网络程序。

UDP 数据报通过数据报套接字 DatagramSocket 发送和接收，系统不保证 UDP 数据报一定能够安全送到目的地，也不能确定什么时候可以抵达。

DatagramPacket 对象封装了 UDP 数据报，在数据报中包含了发送端的 IP 地址和端口号以及接收端的 IP 地址和端口号。

UDP 协议中每个数据报都给出了完整的地址信息，因此无须建立发送方和接收方的连接。如同发快递包裹一样。

5.2 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。

5.3 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。

举例 1：

发送端：

```
DatagramSocket ds = null;
try {
    ds = new DatagramSocket();
    byte[] by = "hello,atguigu.com".getBytes();
    DatagramPacket dp = new DatagramPacket(by, 0, by.length,     InetAd
dress.getByName("127.0.0.1"), 10000);
    ds.send(dp);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (ds != null)
        ds.close();
}
```

接收端：

```
DatagramSocket ds = null;
try {
    ds = new DatagramSocket(10000);
    byte[] by = new byte[1024*64];
    DatagramPacket dp = new DatagramPacket(by, by.length);
    ds.receive(dp);
    String str = new String(dp.getData(), 0, dp.getLength());
    System.out.println(str + "--" + dp.getAddress());
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (ds != null)
        ds.close();
}
```

举例 2：

发送端：

```java
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

```java
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.lengt
h);

            //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();
    }
}
```

6. URL 编程

6.1 URL 类

- URL(Uniform Resource Locator)：统一资源定位符，它表示 Internet 上某一资源的地址。
- 它是一种具体的 URI，即 URL 可以用来标识一个资源，而且还指明了如何 locate 这个资源。
- 通过 URL 我们可以访问 Internet 上的各种网络资源，比如最常见的 www，ftp 站点。浏览器通过解析给定的 URL 可以在网络上查找相应的文件或其他资源。
- URL 的基本结构由 5 部分组成：

<传输协议>://<主机名>:<端口号>/<文件名>#片段名?参数列表

- 例如: http://192.168.1.100:8080/helloworld/index.jsp#a?username=shkstart&password=123
  - 片段名：即锚点，例如看小说，直接定位到章节
  - 参数列表格式：参数名=参数值&参数名=参数值....

- 为了表示 URL，java.net 中实现了类 URL。我们可以通过下面的构造器来初始化一个 URL 对象：
  - public URL (String spec)：通过一个表示 URL 地址的字符串可以构造一个 URL 对象。例如：

    ```
    URL url = new URL ("http://www. atguigu.com/");
    ```

  - public URL(URL context, String spec)：通过基 URL 和相对 URL 构造一个 URL 对象。例如：

    ```
    URL downloadUrl = new URL(url, “download.html")
    ```

    public URL(String protocol, String host, String file); 例如：

    ```
    URL url = new URL("http", "www.atguigu.com", “download. html");
    ```

  - public URL(String protocol, String host, int port, String file); 例如:

    ```
    URL gamelan = new URL("http", "www.atguigu.com", 80, “download.html");
    ```

- URL 类的构造器都声明抛出非运行时异常，必须要对这一异常进行处理，通常是用 try-catch 语句进行捕获。

填写URL地址，进行数据的下载

6.2 URL 类常用方法

一个 URL 对象生成后，其属性是不能被改变的，但可以通过它给定的方法来获取这些属性：

- public String getProtocol( ) 获取该 URL 的协议名
- public String getHost( ) 获取该 URL 的主机名
- public String getPort( ) 获取该 URL 的端口号
- public String getPath( ) 获取该 URL 的文件路径
- public String getFile( ) 获取该 URL 的文件名
- public String getQuery( ) 获取该 URL 的查询名

```
URL url = new URL("http://localhost:8080/examples/myTest.txt");
System.out.println("getProtocol() :"+url.getProtocol());
System.out.println("getHost() :"+url.getHost());
System.out.println("getPort() :"+url.getPort());
System.out.println("getPath() :"+url.getPath());
System.out.println("getFile() :"+url.getFile());
System.out.println("getQuery() :"+url.getQuery());
```

6.3 针对 HTTP 协议的 URLConnection 类

- URL 的方法 openStream()：能从网络上读取数据
- 若希望输出数据，例如向服务器端的 CGI （公共网关接口-Common Gateway Interface-的简称，是用户浏览器和服务器端的应用程序进行连接的接口）程序发送一些数据，则必须先与 URL 建立连接，然后才能对其进行读写，此时需要使用 URLConnection 。
- URLConnection：表示到 URL 所引用的远程对象的连接。当与一个 URL 建立连接时，首先要在一个 URL 对象上通过方法 openConnection() 生成对应的 URLConnection 对象。如果连接过程失败，将产生 IOException.
  - URL netchinaren = new URL ("http://www.atguigu.com/index.shtml");
  - URLConnectonn u = netchinaren.openConnection( );
- 通过 URLConnection 对象获取的输入流和输出流，即可以与现有的 CGI 程序进行交互。
  - public Object getContent( ) throws IOException
  - public int getContentLength( )
  - public String getContentType( )
  - public long getDate( )
  - public long getLastModified( )

- **public InputStream getInputStream ( ) throws IOException**
- public OutputSteram getOutputStream( )throws IOException

6.4 小结

- 位于网络中的计算机具有唯一的 IP 地址，这样不同的主机可以互相区分。
- 客户端－服务器是一种最常见的网络应用程序模型。服务器是一个为其客户端提供某种特定服务的硬件或软件。客户机是一个用户应用程序，用于访问某台服务器提供的服务。端口号是对一个服务的访问场所，它用于区分同一物理计算机上的多个服务。套接字用于连接客户端和服务器，客户端和服务器之间的每个通信会话使用一个不同的套接字。TCP 协议用于实现面向连接的会话。
- Java 中有关网络方面的功能都定义在 java.net 程序包中。Java 用 InetAddress 对象表示 IP 地址，该对象里有两个字段：主机名(String) 和 IP 地址(int)。
- 类 Socket 和 ServerSocket 实现了基于 TCP 协议的客户端－服务器程序。Socket 是客户端和服务器之间的一个连接，连接创建的细节被隐藏了。这个连接提供了一个安全的数据传输通道，这是因为 TCP 协议可以解决数据在传送过程中的丢失、损坏、重复、乱序以及网络拥挤等问题，它保证数据可靠的传送。
- 类 URL 和 URLConnection 提供了最高级网络应用。URL 的网络资源的位置来同一表示 Internet 上各种网络资源。通过 URL 对象可以创建当前应用程序和 URL 表示的网络资源之间的连接，这样当前程序就可以读取网络资源数据，或者把自己的数据传送到网络上去。

第 17 章_反射机制

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

---

1. 反射(Reflection)的概念

1.1 反射的出现背景

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。

1.2 反射概述

Reflection（反射）是被视为动态语言的关键，反射机制允许程序在执行期借助于 Reflection API 取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。

加载完类之后，在堆内存的方法区中就产生了一个 Class 类型的对象（一个类只有一个 Class 对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：反射。

image-20220417161529285

从内存加载上看反射：

1.3 补充：动态语言 vs 静态语言

1、动态语言

是一类在运行时可以改变其结构的语言：例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。通俗点说就是在运行时代码可以根据某些条件改变自身结构。

- 主要动态语言：Object-C、C#、JavaScript、PHP、Python、Erlang。

2、静态语言

与动态语言相对应的，运行时结构不可变的语言就是静态语言。如 Java、C、C++。

> Java 不是动态语言，但 Java 可以称之为“准动态语言”。即 Java 有一定的动态性，我们可以利用反射机制、字节码操作获得类似动态语言的特性。
>
> Java 的动态性让编程的时候更加灵活！

1.4 Java 反射机制研究及应用

Java 反射机制提供的功能：

- 在运行时判断任意一个对象所属的类
- 在运行时构造任意一个类的对象
- 在运行时判断任意一个类所具有的成员变量和方法

- 在运行时获取泛型信息
- 在运行时调用任意一个对象的成员变量和方法
- 在运行时处理注解
- 生成动态代理

1.5 反射相关的主要 API

`java.lang.Class`:代表一个类 java.lang.reflect.Method:代表类的方法 java.lang.reflect.Field:代表类的成员变量 java.lang.reflect.Constructor:代表类的构造器 … …

1.6 反射的优缺点

优点：

- 提高了 Java 程序的灵活性和扩展性，降低了耦合性，提高自适应能力
- 允许程序创建和控制任何类的对象，无需提前硬编码目标类

缺点：

- 反射的性能较低。
  - 反射机制主要应用在对灵活性和扩展性要求很高的系统框架上
- 反射会模糊程序内部逻辑，可读性较差。

2. 理解 Class 类并获取 Class 实例

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:

- java.lang.Class
- java.lang.reflect.*

所以，Class 对象是反射的根源。

2.1 理解 Class

2.1.1 理论上

在 Object 类中定义了以下的方法，此方法将被所有子类继承：

```
public final Class getClass()
```

以上的方法返回值的类型是一个 Class 类，此类是 Java 反射的源头，实际上所谓反射从程序的运行结果来看也很好理解，即：可以通过对象反射求出类的名称。

image-20220417162559217

对象照镜子后可以得到的信息：某个类的属性、方法和构造器、某个类到底实现了哪些接口。对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个结构(class/interface/enum/annotation/primitive type/void/[])的有关信息。

- Class 本身也是一个类
- Class 对象只能由系统建立对象
- 一个加载的类在 JVM 中只会有一个 Class 实例
- 一个 Class 对象对应的是一个加载到 JVM 中的一个.class 文件
- 每个类的实例都会记得自己是由哪个 Class 实例所生成
- 通过 Class 可以完整地得到一个类中的所有被加载的结构
- Class 类是 Reflection 的根源，针对任何你想动态加载、运行的类，唯有先获得相应的 Class 对象

2.1.2 内存结构上

image-20220514180100176

2.2 获取 Class 类的实例(四种方法)

方式 1：要求编译期间已知类型

前提：若已知具体的类，通过类的 class 属性获取，该方法最为安全可靠，程序性能最高

实例：

```
Class clazz = String.class;
```

方式 2：获取对象的运行时类型

前提：已知某个类的实例，调用该实例的 getClass()方法获取 Class 对象

实例：

```
Class clazz = "www.atguigu.com".getClass();
```

方式 3：可以获取编译期间未知的类型

前提：已知一个类的全类名，且该类在类路径下，可通过 Class 类的静态方法 forName()获取，可能抛出 ClassNotFoundException

实例：

```
Class clazz = Class.forName("java.lang.String");
```

方式 4：其他方式(不做要求)

前提：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

实例：

```
ClassLoader cl = this.getClass().getClassLoader();
Class clazz4 = cl.loadClass("类的全类名");
```

再举例：

```
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.at
guigu.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);
    }
}
```

2.3 哪些类型可以有 Class 对象

简言之，所有 Java 类型！

（1）class：外部类，成员(成员内部类，静态内部类)，局部内部类，匿名内部类 （2）interface：接口 （3）[]：数组 （4）enum：枚举 （5）annotation：注解 @interface （6）primitive type：基本数据类型 （7）void

举例：

```
Class c1 = Object.class;
Class c2 = Comparable.class;
Class c3 = String[].class;
Class c4 = int[][].class;
Class c5 = ElementType.class;
Class c6 = Override.class;
Class c7 = int.class;
Class c8 = void.class;
Class c9 = Class.class;
```

```
int[] a = new int[10];
int[] b = new int[100];
Class c10 = a.getClass();
Class c11 = b.getClass();
// 只要元素类型与维度一样，就是同一个Class
System.out.println(c10 == c11);
```

2.4 Class 类的常用方法

方法名	功能说明
static Class forName(String name)	返回指定类名 name 的 Class 对象
Object newInstance()	调用缺省构造函数，返回该 Class 对象的一个实例
getName()	返回此 Class 对象所表示的实体（类、接口、数组类、基本类型或 void）名称
Class getSuperClass()	返回当前 Class 对象的父类的 Class 对象
Class [] getInterfaces()	获取当前 Class 对象的接口
ClassLoader getClassLoader()	返回该类的类加载器
Class getSuperclass()	返回表示此 Class 所表示的实体的超类的 Class
Constructor[] getConstructors()	返回一个包含某些 Constructor 对象的数组
Field[] getDeclaredFields()	返回 Field 对象的一个数组
Method getMethod(String name,Class ... paramTypes)	返回一个 Method 对象，此对象的形参类型为 paramType

举例：

```
String str = "test4.Person";
Class clazz = Class.forName(str);

Object obj = clazz.newInstance();

Field field = clazz.getField("name");
field.set(obj, "Peter");
Object name = field.get(obj);
System.out.println(name);

//注：test4.Person 是test4 包下的Person 类
```

3. 类的加载与 ClassLoader 的理解

3.1 类的生命周期

类在内存中完整的生命周期：加载-->使用-->卸载。其中加载过程又分为：装载、链接、初始化三个阶段。

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、链接、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。

image-20220417171411631

类的加载又分为三个阶段：

（1）装载（Loading）

将类的 class 文件读入内存，并为之创建一个 java.lang.Class 对象。此过程由类加载器完成

（2）链接（Linking）

①验证 Verify：确保加载的类信息符合 JVM 规范，例如：以 cafebabe 开头，没有安全方面的问题。

②准备 Prepare：正式为类变量（static）分配内存并设置类变量默认初始值的阶段，这些内存都将在方法区中进行分配。

③解析 Resolve：虚拟机常量池内的符号引用（常量名）替换为直接引用（地址）的过程。

（3）初始化（Initialization）

- 执行类构造器`<clinit>()`方法的过程。类构造器`<clinit>()`方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造该类对象的构造器）。
- 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。

- 虚拟机会保证一个类的`<clinit>()`方法在多线程环境中被正确加锁和同步。

3.2 类加载器（classloader)

image-20220417173647473

3.2.1 类加载器的作用

将 class 文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后在堆中生成一个代表这个类的 java.lang.Class 对象，作为方法区中类数据的访问入口。

类缓存：标准的 JavaSE 类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过 JVM 垃圾回收机制可以回收这些 Class 对象。

image-20220417173616344

3.2.2 类加载器的分类

JVM 支持两种类型的类加载器，分别为引导类加载器（`Bootstrap ClassLoader`）和自定义类加载器（`User-Defined ClassLoader`）。

从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是 Java 虚拟机规范却没有这么定义，而是将所有派生于抽象类 ClassLoader 的类加载器都划分为自定义类加载器。无论类加载器的类型如何划分，在程序中我们最常见的类加载器结构主要是如下情况：

image-20220417173819953

（1）启动类加载器（引导类加载器，Bootstrap ClassLoader）

- 这个类加载使用 C/C++语言实现的，嵌套在 JVM 内部。获取它的对象时往往返回 null
- 它用来加载 Java 的核心库（JAVA_HOME/jre/lib/rt.jar 或 sun.boot.class.path 路径下的内容）。用于提供 JVM 自身需要的类。
- 并不继承自 java.lang.ClassLoader，没有父加载器。
- 出于安全考虑，Bootstrap 启动类加载器只加载包名为 java、javax、sun 等开头的类
- 加载扩展类和应用程序类加载器，并指定为他们的父类加载器。

（2）扩展类加载器（Extension ClassLoader）

- Java 语言编写，由 sun.misc.Launcher$ExtClassLoader 实现。
- 继承于 ClassLoader 类

- 父类加载器为启动类加载器
- 从 java.ext.dirs 系统属性所指定的目录中加载类库，或从 JDK 的安装目录的 jre/lib/ext 子目录下加载类库。如果用户创建的 JAR 放在此目录下，也会自动由扩展类加载器加载。

（3）应用程序类加载器（系统类加载器，AppClassLoader）

- java 语言编写，由 sun.misc.Launcher$AppClassLoader 实现
- 继承于 ClassLoader 类
- 父类加载器为扩展类加载器
- 它负责加载环境变量 classpath 或系统属性 java.class.path 指定路径下的类库
- 应用程序中的类加载器默认是系统类加载器。
- 它是用户自定义类加载器的默认父加载器
- 通过 ClassLoader 的 getSystemClassLoader()方法可以获取到该类加载器

（4）用户自定义类加载器（了解）

- 在 Java 的日常应用程序开发中，类的加载几乎是由上述 3 种类加载器相互配合执行的。在必要时，我们还可以自定义类加载器，来定制类的加载方式。
- 体现 Java 语言强大生命力和巨大魅力的关键因素之一便是，Java 开发者可以自定义类加载器来实现类库的动态加载，加载源可以是本地的 JAR 包，也可以是网络上的远程资源。

- 同时，自定义加载器能够实现应用隔离，例如 Tomcat，Spring 等中间件和组件框架都在内部实现了自定义的加载器，并通过自定义加载器隔离不同的组件模块。这种机制比 C/C++程序要好太多，想不修改 C/C++程序就能为其新增功能，几乎是不可能的，仅仅一个兼容性便能阻挡住所有美好的设想。
- 自定义类加载器通常需要继承于 ClassLoader。

3.2.3 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 classloader = Class.forName("exer2.ClassloaderDemo").getCla
ssLoader();

//如果是根加载器加载的类，则会得到null
ClassLoader classloader1 = Class.forName("java.lang.Object").getClassLo
ader();
```

（3）获取某个类加载器的父加载器

```
ClassLoader parentClassloader = classloader.getParent();
```

示例代码：

```
package com.atguigu.loader;

import org.junit.Test;

public class TestClassLoader {
    @Test
    public void test01(){
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoade
r();
        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.Time
ZoneNames_zh").getClassLoader();
        System.out.println("加载 sun.util.resources.cldr.zh.TimeZoneName
s_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);

    }
}
```

3.2.4 使用 ClassLoader 获取流

关于类加载器的一个主要方法：getResourceAsStream(String str):获取类路径下的指定文件的输入流

```
InputStream in = null;
in = this.getClass().getClassLoader().getResourceAsStream("exer2\\test.properties");
System.out.println(in);
```

举例：

```
//需要掌握如下的代码
    @Test
    public void test5() throws IOException {
        Properties pros = new Properties();
        //方式1：此时默认的相对路径是当前的module
//        FileInputStream is = new FileInputStream("info.properties");
//        FileInputStream is = new FileInputStream("src//info1.properties");

        //方式2：使用类的加载器
        //此时默认的相对路径是当前module 的src 目录
        InputStream is = ClassLoader.getSystemClassLoader().getResourceAsStream("info1.properties");


        pros.load(is);

        //获取配置文件中的信息
```

```
        String name = pros.getProperty("name");
        String password = pros.getProperty("password");
        System.out.println("name = " + name + ", password = " + passwor
d);
    }
```

4. 反射的基本应用

有了 Class 对象，能做什么？

4.1 应用 1：创建运行时类的对象

这是反射机制应用最多的地方。创建运行时类的对象有两种方式：

方式 1：直接调用 Class 对象的 newInstance()方法

要 求： 1）类必须有一个无参数的构造器。2）类的构造器的访问权限需要足够。

方式 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）调用 Class 对象的 newInstance()方法创建对象

方式二的步骤：

1）通过 Class 类的 getDeclaredConstructor(Class ... parameterTypes)取得本类的指定形参类型的构造器 2）向构造器的形参中传递一个对象数组进去，里面包含了构造器中所需的各个参数。 3）通过 Constructor 实例化对象。

> 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.AtGuiguCla
ss");
        //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.AtGuiguDem
o");
        //java.lang.InstantiationException: com.atguigu.ext.demo.AtGuig
uDemo
        //Caused by: java.lang.NoSuchMethodException: com.atguigu.ext.d
emo.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.AtGuiguDem
o");
        /*
         * 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(Strin
g.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);
    }
}
```

4.2 应用 2：获取运行时类的完整结构

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）。

4.2.1 相关 API

```
//1.实现的全部接口
public Class<?>[] getInterfaces()
```

```
//确定此对象所表示的类或接口实现的接口。

//2.所继承的父类
public Class<? Super T> getSuperclass()
//返回表示此 Class 所表示的实体（类、接口、基本类型）的父类的 Class。

//3.全部的构造器
public Constructor<T>[] getConstructors()
//返回此 Class 对象所表示的类的所有public 构造方法。
public Constructor<T>[] getDeclaredConstructors()
//返回此 Class 对象表示的类声明的所有构造方法。

//Constructor 类中:
//取得修饰符:
public int getModifiers();
//取得方法名称:
public String getName();
//取得参数的类型:
public Class<?>[] getParameterTypes();

//4.全部的方法
public Method[] getDeclaredMethods()
//返回此Class 对象所表示的类或接口的全部方法
public Method[] getMethods()
//返回此Class 对象所表示的类或接口的public 的方法

//Method 类中:
public Class<?> getReturnType()
//取得全部的返回值
public Class<?>[] getParameterTypes()
//取得全部的参数
public int getModifiers()
//取得修饰符
public Class<?>[] getExceptionTypes()
//取得异常信息

//5.全部的Field
public Field[] getFields()
//返回此Class 对象所表示的类或接口的public 的Field。
public Field[] getDeclaredFields()
//返回此Class 对象所表示的类或接口的全部Field。

//Field 方法中:
public int getModifiers()
//以整数形式返回此Field 的修饰符
public Class<?> getType()
//得到Field 的属性类型
```

```
public String getName()
//返回Field 的名称。

//6. Annotation 相关
get Annotation(Class<T> annotationClass)
getDeclaredAnnotations()

//7.泛型相关
//获取父类泛型类型:
Type getGenericSuperclass()
//泛型类型: ParameterizedType
//获取实际的泛型类型参数数组:
getActualTypeArguments()

//8.类所在的包
Package getPackage()
```

4.2.2 获取所有的属性及相关细节

```
package com.atguigu.java2;

import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

import org.junit.Test;

import com.atguigu.java1.Person;

public class FieldTest {

	@Test
	public void test1(){

		Class clazz = Person.class;
		//getFields():获取到运行时类本身及其所有的父类中声明为public 权限的属性
//		Field[] fields = clazz.getFields();
//
//		for(Field f : fields){
//			System.out.println(f);
//		}

		//getDeclaredFields():获取当前运行时类中声明的所有属性
		Field[] declaredFields = clazz.getDeclaredFields();
		for(Field f : declaredFields){
			System.out.println(f);
		}
	}
```

```java
    //权限修饰符  变量类型  变量名
    @Test
    public void test2(){
        Class clazz = Person.class;
      Field[] declaredFields = clazz.getDeclaredFields();
      for(Field f : declaredFields){
          //1.权限修饰符
          /*
          * 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 modifier = f.getModifiers();
          System.out.print(Modifier.toString(modifier) + "\t");

//            //2.数据类型
          Class type = f.getType();
          System.out.print(type.getName() + "\t");
//
//            //3.变量名
          String fName = f.getName();
          System.out.print(fName);
//
          System.out.println();
      }
    }
}
```

4.2.3 获取所有的方法及相关细节

```java
package com.atguigu.java2;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

import org.junit.Test;
```

```java
import com.atguigu.java1.Person;

public class MethodTest {

	@Test
	public void test1() {

		Class clazz = Person.class;
		// getMethods():获取到运行时类本身及其所有的父类中声明为public
权限的方法
		// Method[] methods = clazz.getMethods();
		//
		// for(Method m : methods){
		// System.out.println(m);
		// }

		// getDeclaredMethods():获取当前运行时类中声明的所有方法
		Method[] declaredMethods = clazz.getDeclaredMethods();
		for (Method m : declaredMethods) {
			System.out.println(m);
		}
		//
	}

	// 注解信息
	// 权限修饰符 返回值类型 方法名(形参类型1 参数1,形参类型2 参数2,...)
 throws 异常类型1,...{}
	@Test
	public void test2() {
		Class clazz = Person.class;
		Method[] declaredMethods = clazz.getDeclaredMethods();
		for (Method m : declaredMethods) {
			// 1.获取方法声明的注解
			Annotation[] annos = m.getAnnotations();
			for (Annotation a : annos) {
				System.out.println(a);
			}

			// 2.权限修饰符
			System.out.print(Modifier.toString(m.getModifiers())
+ "\t");

			// 3.返回值类型
			System.out.print(m.getReturnType().getName() + "\t");

			// 4.方法名
			System.out.print(m.getName());
			System.out.print("(");
```

```
                    // 5.形参列表
                    Class[] parameterTypes = m.getParameterTypes();
                    if (!(parameterTypes == null && parameterTypes.length
 == 0)) {
                        for (int i = 0; i < parameterTypes.length; i++)
 {

                            if (i == parameterTypes.length - 1) {
                                System.out.print(parameterTypes[i].
getName() + " args_" + i);
                                break;
                            }

                            System.out.print(parameterTypes[i].getNam
e() + " args_" + i + ",");
                        }
                    }

                    System.out.print(")");

                    // 6.抛出的异常
                    Class[] exceptionTypes = m.getExceptionTypes();
                    if (exceptionTypes.length > 0) {
                        System.out.print("throws ");
                        for (int i = 0; i < exceptionTypes.length; i++)
 {
                            if (i == exceptionTypes.length - 1) {
                                System.out.print(exceptionTypes[i].
getName());
                                break;
                            }
                            System.out.print(exceptionTypes[i].getNam
e() + ",");
                        }
                    }
                    System.out.println();
                }
            }
}
```

4.2.4 获取其他结构(构造器、父类、接口、包、注解等)

```
package com.atguigu.java2;

import com.atguigu.java1.Person;
import org.junit.Test;

import java.lang.annotation.Annotation;
import java.lang.reflect.Constructor;
import java.lang.reflect.ParameterizedType;
```

```java
import java.lang.reflect.Type;

/**
 * @author 尚硅谷-宋红康
 * @create 2020 下午 2:47
 */
public class OtherTest {

    /*
      获取当前类中的所有的构造器
     */
    @Test
    public void test1(){
        Class clazz = Person.class;
        Constructor[] cons = clazz.getDeclaredConstructors();
        for(Constructor c :cons){
            System.out.println(c);
        }
    }
    /*
      获取运行时类的父类
     */
    @Test
    public void test2(){
        Class clazz = Person.class;
        Class superclass = clazz.getSuperclass();
        System.out.println(superclass);//class com.atguigu.java1.Creatu
re
    }
    /*
      获取运行时类的所在的包
     */
    @Test
    public void test3(){
        Class clazz = Person.class;
        Package pack = clazz.getPackage();
        System.out.println(pack);

    }
    /*
      获取运行时类的注解
     */
    @Test
    public void test4(){
        Class clazz = Person.class;
        Annotation[] annos = clazz.getAnnotations();
        for (Annotation anno : annos) {

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

```
        }

    }

    /*
      获取运行时类所实现的接口
     */
    @Test
    public void test5(){
        Class clazz = Person.class;
        Class[] interfaces = clazz.getInterfaces();
        for (Class anInterface : interfaces) {

            System.out.println(anInterface);
        }

    }
    /*
      获取运行时类的带泛型的父类
     */
    @Test
    public void test6(){
        Class clazz = Person.class;
        Type genericSuperclass = clazz.getGenericSuperclass();
        System.out.println(genericSuperclass);//com.atguigu.java1.Creat
ure<java.lang.String>
    }
}
```

4.2.5 获取泛型父类信息（选讲）

示例代码获取泛型父类信息：

```
/* Type:
 *  （1）Class
 *  （2）ParameterizedType
 *          例如: Father<String,Integer>
 *                ArrayList<String>
 *  （3）TypeVariable
 *          例如: T, U,E,K,V
 *  （4）WildcardType
 *          例如:
 *          ArrayList<?>
 *          ArrayList<? super 下限>
 *          ArrayList<? extends 上限>
 *  （5）GenericArrayType
 *          例如: T[]
 *
```

```
 */
public class TestGeneric {
      public static void main(String[] args) {
            //需求：在运行时，获取Son 类型的泛型父类的泛型实参<String,Integ
er>

            //（1）还是先获取Class 对象
            Class clazz = Son.class;//四种形式任意一种都可以

            //（2）获取泛型父类
//          Class sc = clazz.getSuperclass();
//          System.out.println(sc);
            /*
             * getSuperclass()只能得到父类名，无法得到父类的泛型实参列表
             */
            Type type = clazz.getGenericSuperclass();

            // Father<String,Integer>属于ParameterizedType
            ParameterizedType pt = (ParameterizedType) type;

            //（3）获取泛型父类的泛型实参列表
            Type[] typeArray = pt.getActualTypeArguments();
            for (Type type2 : typeArray) {
                  System.out.println(type2);
            }
      }
}
//泛型形参：<T,U>
class Father<T,U>{

}
//泛型实参：<String,Integer>
class Son extends Father<String,Integer>{

}
```

4.2.6 获取内部类或外部类信息（选讲）

public Class<?>[] getClasses()：返回所有公共内部类和内部接口。包括从超类继承的公共类和接口成员以及该类声明的公共类和接口成员。

public Class<?>[] getDeclaredClasses()：返回 Class 对象的一个数组，这些对象反映声明为此 Class 对象所表示的类的成员的所有类和接口。包括该类所声明的公共、保护、默认（包）访问及私有类和接口，但不包括继承的类和接口。

public Class<?> getDeclaringClass()：如果此 Class 对象所表示的类或接口是一个内部类或内部接口，则返回它的外部类或外部接口，否则返回 null。

Class<?> getEnclosingClass() ：返回某个内部类的外部类

```
@Test
public void test5(){
	Class<?> clazz = Map.class;
	Class<?>[] inners = clazz.getDeclaredClasses();
	for (Class<?> inner : inners) {
		System.out.println(inner);
	}

	Class<?> ec = Map.Entry.class;
	Class<?> outer = ec.getDeclaringClass();
	System.out.println(outer);
}
```

4.2.7 小结

1. 在实际的操作中，取得类的信息的操作代码，并不会经常开发。
2. 一定要熟悉 java.lang.reflect 包的作用，反射机制。

4.3 应用 3：调用运行时类的指定结构

4.3.1 调用指定的属性

在反射机制中，可以直接通过 Field 类操作类中的属性，通过 Field 类提供的 set()和 get()方法就可以完成设置和取得属性内容的操作。

（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）设置指定对象 obj 上此 Field 的属性内容

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）取得指定对象 obj 上此 Field 的属性内容

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");
```

```java
        //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);
    }
}
```

关于 setAccessible 方法的使用：

- Method 和 Field、Constructor 对象都有 setAccessible()方法。
- setAccessible 启动和禁用访问安全检查的开关。
- 参数值为 true 则指示反射的对象在使用时应该取消 Java 语言访问检查。
  - 提高反射的效率。如果代码中必须用反射，而该句代码需要频繁的被调用，那么请设置为 true。
  - 使得原本无法访问的私有成员也可以访问
- 参数值为 false 则指示反射的对象应该实施 Java 语言访问检查。

4.3.2 调用指定的方法

image-20220417181700813

（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", Strin
g.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setNameMethodReturn
Value 为null
        System.out.println("setNameMethodReturnValue = " + setNameMetho
dReturnValue);

        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 = " + getNameMetho
dReturnValue);//张三
    }

    @Test
    public void test02()throws Exception{
        Class<?> clazz = Class.forName("com.atguigu.ext.demo.AtGuiguCla
ss");
        Method printInfoMethod = clazz.getMethod("printInfo", String.cl
ass);
        //printInfo 方法是静态方法
        printInfoMethod.invoke(null,"尚硅谷");
    }
}
```

4.3.3 练习

读取 user.properties 文件中的数据，通过反射完成 User 类对象的创建及对应方法的调用。

配置文件：user.properties

```
className:com.atguigu.bean.User
methodName:show
```

User.java 文件：

```java
package com.atguigu.bean;

/**
 * @author 尚硅谷-宋红康
 * @create 18:41
 */
public class User {
    private String name;

    public User() {
    }

    public User(String name) {
        this.name = name;
    }

    public void show(){
        System.out.println("我是一个脉脉平台的用户");
    }
}
```

ReflectTest.java 文件：

```java
package com.atguigu.java4;

import org.junit.Test;

import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.Method;
import java.util.Properties;

/**
 * @author 尚硅谷-宋红康
 * @create 18:43
 */
public class ReflectTest {
    @Test
    public void test() throws Exception {
        //1.创建Properties 对象
        Properties pro = new Properties();
```

```
        //2.加载配置文件，转换为一个集合
        ClassLoader classLoader = ClassLoader.getSystemClassLoader();
        InputStream is = classLoader.getResourceAsStream("user.properti
es");
        pro.load(is);

        //3.获取配置文件中定义的数据
        String className = pro.getProperty("className");
        String methodName = pro.getProperty("methodName");

        //4.加载该类进内存
        Class clazz = Class.forName(className);

        //5.创建对象
        Object instance = clazz.newInstance();

        //6.获取方法对象
        Method showMethod = clazz.getMethod(methodName);

        //7.执行方法
        showMethod.invoke(instance);
    }
}
```

5. 应用 4：读取注解信息

一个完整的注解应该包含三个部分：（1）声明（2）使用（3）读取

5.1 声明自定义注解

```
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();
}
```

- 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。

5.2 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 + '\'' +
                '}';
    }
}
```

5.3 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(Tabl
e.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);
    }
}
```

6. 体会反射的动态性

体会 1：

```
public class ReflectionTest {

    //体会反射的动态性：动态的创建给定字符串对应的类的对象
    public <T> T getInstance(String className) throws Exception {

        Class clazz = Class.forName(className);

        Constructor constructor = clazz.getDeclaredConstructor();
        constructor.setAccessible(true);
        return (T) constructor.newInstance();

    }

    @Test
    public void test1() throws Exception {
        String className = "com.atguigu.java1.Person";
        Person p1 = getInstance(className);
        System.out.println(p1);
    }
}
```

体会 2：

```
public class ReflectionTest {
    //体会反射的动态性：动态的创建指定字符串对应类的对象，并调用指定的方法
    public Object  invoke(String className,String methodName) throws Ex
ception {
        Class clazz = Class.forName(className);
        Constructor constructor = clazz.getDeclaredConstructor();
        constructor.setAccessible(true);
        //动态的创建指定字符串对应类的对象
        Object obj = constructor.newInstance();

        Method method = clazz.getDeclaredMethod(methodName);
        method.setAccessible(true);
        return method.invoke(obj);
    }

    @Test
    public void test2() throws Exception {
        String info = (String) invoke("com.atguigu.java1.Person", "show
");

        System.out.println("返回值为：" + info);
```

```
    }
}
```

体会 3：

```
public class ReflectionTest {
      /**
       * 榨汁机(Juicer)榨汁的案例
      * 分别有水果(Fruit)苹果(Apple)香蕉(Banana)桔子(Orange)榨汁(squeeze)
       *
       * @throws Exception
       */
      public static void main(String[] args) throws Exception {

            BufferedReader br = new BufferedReader(new FileReader("conf
ig.properties")); // 创建输入流对象,关联配置文件
            Class<?> clazz = Class.forName(br.readLine()); // 读取配置文
件一行内容,获取该类的字节码对象
            Fruit f = (Fruit) clazz.newInstance(); // 通过字节码对象创建
实例对象
            Juicer j = new Juicer();
            j.run(f);
      }

}

interface Fruit {
      public void squeeze();
}

class Apple implements Fruit {
      public void squeeze() {
            System.out.println("榨出一杯苹果汁儿");
      }
}

class Orange implements Fruit {
      public void squeeze() {
            System.out.println("榨出一杯桔子汁儿");
      }
}

class Juicer {
      public void run(Fruit f) {
            f.squeeze();
      }
}
```

其中，配置文件【config.properties】

com.atguigu.java1.Orange

第 18 章_JDK8-17 新特性

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

1. Java 版本迭代概述

1.1 发布特点（小步快跑，快速迭代）

从 Java 9 这个版本开始，Java 的计划发布周期是 6 个月，下一个 Java 的主版本将于 2018 年 3 月发布，命名为 Java 18.3，紧接着再过六个月将发布 Java 18.9。

这意味着 Java 的更新从传统的以特性驱动的发布周期，转变为以时间驱动的（6 个月为周期发布一个大版本，每个季度发布一个中间特性版本）发布模式，并且承诺不会跳票。通过这样的方式，开发团队可以把一些关键特性尽早合并到 JDK 之中，以快速得到开发者反馈，在一定程度上避免出现像 Java 9 这样两次被迫延迟发布的窘况。

针对企业客户的需求，Oracle 将以三年为周期发布长期支持版本（long term support）。

Oracle 的官方观点认为：**与 Java 7->8->9 相比，Java 9->10->11 的升级和 8->8u20->8u40 更相似。**

新模式下的 Java 版本发布都会包含许多变更，包括**语言变更和 JVM 变更**，这两者都会对 IDE、字节码库和框架产生重大影响。此外，不仅会新增其他 API，还**会有 API 被删除（这在 Java 8 之前没有发生过）**。

目前看这种发布策略是非常成功的，**解开了 Java/JVM 演进的许多枷锁，至关重要的是，OpenJDK 的权力中心，正在转移到开发社区和开发者手中。**在新的模式中，既可以利用 LTS 满足企业长期可靠支持的需求，也可以满足各种开发者对于新特性迭代的诉求。你可能注意到了 Switch Expressions 被打上了预览（Preview）的标签，Shenandoah GC 则是实验（Experimental）特性，这些都是以往的发布周期下不大现实的，因为用 2-3 年的最小间隔粒度来实验一个特性，基本是不现实的。

2021 年 9 月，Java 17 发布了，Oracle 宣布 JDK 17 及之后的版本都免费使用，包括商用。新许可是“Oracle 免费条款和条件”(NFTC) 许可。此 Oracle JDK 许可证允许所有用户免费使用，甚至可以用于商业和生产用途。只要不收费，再分发是允许

的。此外，Oracle 将继续按照自 Java 9 以来的相同版本和时间表提供 GPL 下的 Oracle OpenJDK 版本。

1.2 名词解释

名词解释：Oracle JDK 和 Open JDK

这两个 JDK 最大不同就是许可证不一样。**但是对于个人用户来讲，没区别。**

```
C:\> java -version
java version "17" 2021-09-14 LTS
Java(TM) SE Runtime Environment (build 17+35-LTS-2724)
Java HotSpot(TM) 64-Bit Server VM (build 17+35-LTS-2724, mixed mode, sh
aring)
```

```
[root@ ~]# java  --version
openjdk 11.0.5 2019-10-15
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.5+10)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.5+10, mixed mode)
```

	Oracle JDK	Open JDK
来源	Oracle 团队维护	Oracle 和 Open Java 社区
授权协议	Java 17 及更高版本 Oracle Java SE 许可证 Java16 及更低版本甲骨文免费条款和条件（NFTC）许可协议	GPL v2 许可证
关系	由 Open JDK 构建，增加了少许内容	
发行周期	6 个月，每两年一个 LTS	6 个月
是否收费	2021 年 9 月起 Java17 及更高版本所有用户免费。16 及更低版本，个人用户、开发用户免费。	2017 年 9 月起，所有版本免费
LTS	目前 7,8,11,17	无，但是有社区编译的其他支持
对语法的支持	一致	一致
性能	最佳	接近，几乎无差

其中，有哪些 Open JDK 发行版可选？

Open JDK 虽然没有官方的 LTS 版本，但是开源社区有支持。会有一些公司或组织基于 Open JDK 做发行版，提供 LTS。

名称	支持团队	官网

名称	支持团队	官网
AdoptOpenJDK	Amazon，Microsoft，IBM，Red Hat，Pivotal(EMC 和 VMware)	AdoptOpenJDK - 开源，预建 OpenJDK 二进制文件
Alibaba Dragonwell （龙井）	阿里巴巴	Dragonwell (dragonwell-jdk.io)
Tencent Kona	腾讯	Home · Tencent/TencentKona-8 Wiki · GitHub
Microsoft JDK	Microsoft 微软	Microsoft Build of OpenJDK
毕昇 JDK	华为	毕昇 JDK-鲲鹏社区 (hikunpeng.com)
Amazon Corretto	Amazon 亚马逊	Amazon Corretto

康师傅有话说：

以上是比较出名的版本（有些 JDK 加入了自己的特色）。建议 AdoptOpenJDK。

名词解释：JEP

JEP(JDK Enhancement Proposals)：jdk 改进提案，每当需要有新的设想时候，JEP 可以在 JCP(Java community Process)之前或者同时提出非正式的规范(specification)，被正式认可的 JEP 正式写进 JDK 的发展路线图并分配版本号。

名词解释：LTS

LTS（Long-term Support）即长期支持。Oracle 官网提供了对 Oracle JDK 个别版本的长期支持，即使发发行了新版本，比如目前最新的 JDK18，在结束日期前，LTS 版本都会被长期支持。（出了 bug，会被修复，非 LTS 则不会再有补丁发布）所以，一定要选一个 LTS 版本，不然出了漏洞没人修复了。

版本	开始日期	结束日期	延期结束日期
7（LTS）	2011 年 7 月	2019 年 7 月	2022 年 7 月
8（LTS）	2014 年 3 月	2022 年 3 月	2030 年 12 月
11（LTS）	2018 年 9 月	2023 年 9 月	2026 年 9 月
17（LTS）	2021 年 9 月	2026 年 9 月	2029 年 9 月
21（LTS）	2023 年 9 月	2028 年 9 月	2031 年 9 月

如果要选择 Oracle JDK，目前可选的 LTS 版本为 8、11、17 三个。

1.3 各版本支持时间路线图

Oracle Java SE支持路线图 *†				
释放	GA日期	首要支持，直到	扩展支持，直到	持续支持
7	2011年7月	2019年七月	2022年7月*****	不定
8 **	2014年3月	2022年3月	2030年12月	不定
9（非LTS）	2017年9月	2018年三月	无法使用	不定
10（非LTS）	2018年三月	2018年9月	无法使用	不定
11（LTS）	2018年9月	2023年9月	2026年9月	不定
12（非LTS）	2019年三月	2019年九月	无法使用	不定
13（非LTS）	2019年九月	2020年3月	无法使用	不定
14（非LTS）	2020年3月	2020年9月	无法使用	不定
15（非LTS）	2020年9月	2021年3月	无法使用	不定
16（非LTS）	2021年3月	2021年9月	无法使用	不定
17（LTS）	2021年9月***	2026年9月****	2029年9月****	不定

1.4 各版本介绍

jdk 9

Java 9 提供了超过 150 项新功能特性，包括备受期待的模块化系统、可交互的 REPL 工具：jshell，JDK 编译工具，Java 公共 API 和私有代码，以及安全增强、扩展提升、性能管理改善等。可以说 Java 9 是一个庞大的系统工程，完全做了一个整体改变。

特性太多，查看链接：

https://openjdk.java.net/projects/jdk9/

jdk 10

https://openjdk.java.net/projects/jdk/10/

286: Local-Variable Type Inference **局部变量类型推断** 296: Consolidate the JDK Forest into a Single Repository JDK 库的合并 304: Garbage-Collector Interface 统一的垃圾回收接口 307: Parallel Full GC for G1 为 G1 提供并行的 Full GC 310: Application Class-Data Sharing 应用程序类数据（AppCDS）共享 312: Thread-Local Handshakes ThreadLocal 握手交互 313: Remove the Native-Header Generation Tool (javah) 移除 JDK 中附带的 javah 工具 314: Additional Unicode Language-Tag Extensions 使用附加的 Unicode 语言标记扩展 316: Heap Allocation on Alternative Memory Devices 能将堆内存占用分配给用户指定的备用内存设备 317: Experimental Java-Based JIT Compiler 使用 Graal 基于 Java 的编译器

319: Root Certificates 根证书 322: Time-Based Release Versioning 基于时间定于的发布版本

jdk 11

https://openjdk.java.net/projects/jdk/11/

181: Nest-Based Access Control 基于嵌套的访问控制 309: Dynamic Class-File Constants 动态类文件常量 315: Improve Aarch64 Intrinsics 改进 Aarch64 Intrinsics 318: Epsilon: A No-Op Garbage Collector **Epsilon — 一个 No-Op（无操作）的垃圾收集器** 320: Remove the Java EE and CORBA Modules 删除 Java EE 和 CORBA 模块 321: HTTP Client (Standard) HTTPClient API 323: Local-Variable Syntax for Lambda Parameters 用于 Lambda 参数的局部变量语法 324: Key Agreement with Curve25519 and Curve448 Curve25519 和 Curve448 算法的密钥协议 327: Unicode 10 328: Flight Recorder 飞行记录仪 329: ChaCha20 and Poly1305 Cryptographic Algorithms ChaCha20 和 Poly1305 加密算法 330: Launch Single-File Source-Code Programs 启动单一文件的源代码程序 331: Low-Overhead Heap Profiling 低开销的 Heap Profiling 332: Transport Layer Security (TLS) 1.3 支持 TLS 1.3 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental) **可伸缩低延迟垃圾收集器** 335: Deprecate the Nashorn JavaScript Engine 弃用 Nashorn JavaScript 引擎 336: Deprecate the Pack200 Tools and API 弃用 Pack200 工具和 API

jdk 12

https://openjdk.java.net/projects/jdk/12/

189：Shenandoah: A Low-Pause-Time Garbage Collector (Experimental) 低暂停时间的 GC 230: Microbenchmark Suite 微基准测试套件 325: Switch Expressions (Preview) switch 表达式 334: JVM Constants API JVM 常量 API 340: One AArch64 Port, Not Two 只保留一个 AArch64 实现 341: Default CDS Archives 默认类数据共享归档文件 344: Abortable Mixed

Collections for G1 可中止的 G1 Mixed GC 346: Promptly Return Unused Committed Memory from G1 G1 及时返回未使用的已分配内存

jdk 13

https://openjdk.java.net/projects/jdk/13/

350: Dynamic CDS Archives 动态 CDS 档案 351: ZGC: Uncommit Unused Memory ZGC:取消使用未使用的内存 353: Reimplement the Legacy Socket API 重新实现旧版套接字 API 354: Switch Expressions (Preview) switch 表达式（预览） 355: Text Blocks (Preview) 文本块（预览）

jdk 14

https://openjdk.java.net/projects/jdk/14/

305: Pattern Matching for instanceof (Preview) instanceof 的模式匹配 343: Packaging Tool (Incubator) 打包工具 345: NUMA-Aware Memory Allocation for G1 G1 的 NUMA-Aware 内存分配 349: JFR Event Streaming JFR 事件流 352: Non-Volatile Mapped Byte Buffers 非易失性映射字节缓冲区 358: Helpful NullPointerExceptions 实用的 NullPointerExceptions 359: Records (Preview) 361: Switch Expressions (Standard) Switch 表达式 362: Deprecate the Solaris and SPARC Ports 弃用 Solaris 和 SPARC 端口 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector 删除并发标记扫描（CMS）垃圾回收器 364: ZGC on macOS 365: ZGC on Windows 366: Deprecate the ParallelScavenge + SerialOld GC Combination 弃用 ParallelScavenge + SerialOld GC 组合 367: Remove the Pack200 Tools and API 删除 Pack200 工具和 API 368: Text Blocks (Second Preview) 文本块 370: Foreign-Memory Access API (Incubator) 外部存储器访问 API

jdk 15

https://openjdk.java.net/projects/jdk/15/

339: Edwards-Curve Digital Signature Algorithm (EdDSA) EdDSA 数字签名算法 360: Sealed Classes (Preview) 密封类（预览） 371: Hidden Classes 隐藏类 372: Remove the Nashorn JavaScript Engine 移除 Nashorn JavaScript 引擎 373: Reimplement the Legacy DatagramSocket API 重新实现 Legacy DatagramSocket API 374: Disable and Deprecate Biased Locking 禁用偏向锁定 375: Pattern Matching for instanceof (Second Preview) instanceof 模式匹配（第二次预览） 377: ZGC: A Scalable Low-Latency Garbage Collector ZGC：一个可扩展的低延迟垃圾收集器 378: Text Blocks

文本块 379: Shenandoah: A Low-Pause-Time Garbage Collector Shenandoah:低暂停时间垃圾收集器 381: Remove the Solaris and SPARC Ports 移除 Solaris 和 SPARC 端口 383: Foreign-Memory Access API (Second Incubator) 外部存储器访问 API（第二次孵化版） 384: Records (Second Preview) Records（第二次预览） 385: Deprecate RMI Activation for Removal 废弃 RMI 激活机制

jdk 16

https://openjdk.java.net/projects/jdk/16/

338: Vector API (Incubator) Vector API（孵化器） 347: Enable C++14 Language Features JDK C++的源码中允许使用 C++14 的语言特性 357: Migrate from Mercurial to Git OpenJDK 源码的版本控制从 Mercurial (hg) 迁移到 git 369: Migrate to GitHub OpenJDK 源码的版本控制迁移到 github 上 376: ZGC: Concurrent Thread-Stack Processing ZGC：并发线程处理 380: Unix-Domain Socket Channels Unix 域套接字通道 386: Alpine Linux Port 将 glibc 的 jdk 移植到使用 musl 的 alpine linux 上 387: Elastic Metaspace 弹性元空间 388: Windows/AArch64 Port 移植 JDK 到 Windows/AArch64 389: Foreign Linker API (Incubator) 提供 jdk.incubator.foreign 来简化 native code 的调用 390: Warnings for Value-Based Classes 提供基于值的类的警告 392: Packaging Tool jpackage 打包工具转正 393: Foreign-Memory Access API (Third Incubator) 394: Pattern Matching for instanceof Instanceof 的模式匹配转正 395: Records Records 转正 396: Strongly Encapsulate JDK Internals by Default 默认情况下，封装了 JDK 内部构件 397: Sealed Classes (Second Preview) 密封类

jdk 17

https://openjdk.java.net/projects/jdk/17/

306: Restore Always-Strict Floating-Point Semantics 恢复始终严格的浮点语义

356: Enhanced Pseudo-Random Number Generators 增强型伪随机数生成器

382: New macOS Rendering Pipeline 新的 macOS 渲染管道

391: macOS/AArch64 Port macOS/AArch64 端口

398: Deprecate the Applet API for Removal 弃用 Applet API 后续将进行删除

403: Strongly Encapsulate JDK Internals 强封装 JDK 的内部 API

406: Pattern Matching for switch (Preview) switch 模式匹配（预览）

407: Remove RMI Activation 删除 RMI 激活机制

409: Sealed Classes 密封类转正

410: Remove the Experimental AOT and JIT Compiler 删除实验性的 AOT 和 JIT 编译器

411: Deprecate the Security Manager for Removal 弃用即将删除的安全管理器

412: Foreign Function & Memory API (Incubator) 外部函数和内存 API（孵化特性）

414: Vector API (Second Incubator) Vector API（第二次孵化特性）

415: Context-Specific Deserialization Filters 上下文特定的反序列化过滤器

1.5 JDK 各版本下载链接

https://www.oracle.com/java/technologies/downloads/archive/

image-20220525200441935

1.6 如何学习新特性

对于新特性，我们应该从哪几个角度学习新特性呢？

- 语法层面：
  - 比如 JDK5 中的自动拆箱、自动装箱、enum、泛型
  - 比如 JDK8 中的 lambda 表达式、接口中的默认方法、静态方法
  - 比如 JDK9 中局部变量的类型推断
  - 比如 JDK12 中的 switch
  - 比如 JDK13 中的文本块
- API 层面：
  - 比如 JDK8 中的 Stream、Optional、新的日期时间、HashMap 的底层结构
  - 比如 JDK9 中 String 的底层结构
  - 新的 / 过时的 API
- 底层优化
  - 比如 JDK8 中永久代被元空间替代、新的 JS 执行引擎
  - 比如新的垃圾回收器、GC 参数、JVM 的优化

2. Lambda 表达式

2.1 关于 Java8 新特性简介

Java 8 (又称为 JDK 8 或 JDK1.8) 是 Java 语言开发的一个主要版本。 Java 8 是 oracle 公司于 2014 年 3 月发布，可以看成是自 Java 5 以来最具革命性的版本。Java 8 为 Java 语言、编译器、类库、开发工具与 JVM 带来了大量新特性。

尚硅谷

image-20220525201653599

- 速度更快
- 代码更少(增加了新的语法：**Lambda 表达式**)
- 强大的 **Stream API**

- 便于并行
  - **并行流**就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。相比较串行的流，并行的流可以很大程度上提高程序的执行效率。
  - Java 8 中将并行进行了优化，我们可以很容易的对数据进行并行操作。Stream API 可以声明性地通过 parallel() 与 sequential() 在并行流与顺序流之间进行切换。
- 最大化减少空指针异常：Optional
- Nashorn 引擎，允许在 JVM 上运行 JS 应用
  - 发音“nass-horn”,是德国二战时一个坦克的命名
  - javascript 运行在 jvm 已经不是新鲜事了，Rhino 早在 jdk6 的时候已经存在。现在替代 Rhino，官方的解释是 Rhino 相比其他 JavaScript 引擎（比如 google 的 V8）实在太慢了，改造 Rhino 还不如重写。所以 Nashorn 的性能也是其一个亮点。
  - Nashorn 项目在 JDK 9 中得到改进；在 JDK11 中 `Deprecated`，后续 JDK15 版本中 `remove`。在 JDK11 中取以代之的是 GraalVM。（GraalVM 是一个运行时平台，它支持 Java 和其他基于 Java 字节码的语言，但也支持其他语言，如 JavaScript，Ruby，Python 或 LLVM。性能是 Nashorn 的 2 倍以上。）

2.2 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(new Runnable() {
            @Override
            public void run() {
                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**。

2.3 Lambda 及其使用举例

Lambda 是一个**匿名函数**，我们可以把 Lambda 表达式理解为是**一段可以传递的代码**（将代码像数据一样进行传递）。使用它可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使 Java 的语言表达能力得到了提升。

- 从匿名类到 Lambda 的转换举例 1

```java
//匿名内部类
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello World!");
    }
};
```

```java
//Lambda 表达式
Runnable r1 = () -> System.out.println("Hello Lambda!");
```

image-20220527101737072

- 从匿名类到 Lambda 的转换举例 2

```java
//原来使用匿名内部类作为参数传递
TreeSet<String> ts = new TreeSet<>(new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return Integer.compare(o1.length(), o2.length());
    }
});
```

```java
//Lambda 表达式作为参数传递
TreeSet<String> ts2 = new TreeSet<>(
        (o1, o2) -> Integer.compare(o1.length(), o2.length))
);
```

image-20220527101814203

2.4 语法

Lambda 表达式：在 Java 8 语言中引入的一种新的语法元素和操作符。这个操作符为“->”， 该操作符被称为 Lambda 操作符或箭头操作符。它将 Lambda 分为两个部分：

- 左侧：指定了 Lambda 表达式需要的参数列表
- 右侧：指定了 Lambda 体，是抽象方法的实现逻辑，也即 Lambda 表达式要执行的功能。

语法格式一：无参，无返回值

```java
@Test
public void test1(){
    //未使用Lambda 表达式
    Runnable r1 = new Runnable() {
        @Override
        public void run() {
            System.out.println("我爱北京天安门");
        }
    };

    r1.run();

    System.out.println("***********************");

    //使用Lambda 表达式
    Runnable r2 = () -> {
```

```
        System.out.println("我爱北京故宫");
    };

    r2.run();
}
```

语法格式二：Lambda 需要一个参数，但是没有返回值。

```
@Test
public void test2(){
    //未使用Lambda 表达式
    Consumer<String> con = new Consumer<String>() {
        @Override
        public void accept(String s) {
            System.out.println(s);
        }
    };
    con.accept("谎言和誓言的区别是什么？");

    System.out.println("*******************");

    //使用Lambda 表达式
    Consumer<String> con1 = (String s) -> {
        System.out.println(s);
    };
    con1.accept("一个是听得人当真了，一个是说的人当真了");

}
```

语法格式三：数据类型可以省略，因为可由编译器推断得出，称为“类型推断”

```
@Test
public void test3(){
    //语法格式三使用前
    Consumer<String> con1 = (String s) -> {
        System.out.println(s);
    };
    con1.accept("一个是听得人当真了，一个是说的人当真了");

    System.out.println("*******************");
    //语法格式三使用后
    Consumer<String> con2 = (s) -> {
        System.out.println(s);
    };
    con2.accept("一个是听得人当真了，一个是说的人当真了");
```

```
}
```

语法格式四：Lambda 若只需要一个参数时，参数的小括号可以省略

```
@Test
public void test4(){
    //语法格式四使用前
    Consumer<String> con1 = (s) -> {
        System.out.println(s);
    };
    con1.accept("一个是听得人当真了，一个是说的人当真了");

    System.out.println("******************");
    //语法格式四使用后
    Consumer<String> con2 = s -> {
        System.out.println(s);
    };
    con2.accept("一个是听得人当真了，一个是说的人当真了");


}
```

语法格式五：Lambda 需要两个或以上的参数，多条执行语句，并且可以有返回值

```
@Test
public void test5(){
    //语法格式五使用前
    Comparator<Integer> com1 = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            System.out.println(o1);
            System.out.println(o2);
            return o1.compareTo(o2);
        }
    };

    System.out.println(com1.compare(12,21));
    System.out.println("****************************");
    //语法格式五使用后
    Comparator<Integer> com2 = (o1,o2) -> {
        System.out.println(o1);
        System.out.println(o2);
        return o1.compareTo(o2);
    };
```

```
    System.out.println(com2.compare(12,6));


}
```

语法格式六：当 Lambda 体只有一条语句时，return 与大括号若有，都可以省略

```
@Test
public void test6(){
    //语法格式六使用前
    Comparator<Integer> com1 = (o1,o2) -> {
        return o1.compareTo(o2);
    };

    System.out.println(com1.compare(12,6));

    System.out.println("****************************");
    //语法格式六使用后
    Comparator<Integer> com2 = (o1,o2) -> o1.compareTo(o2);

    System.out.println(com2.compare(12,21));

}

@Test
public void test7(){
    //语法格式六使用前
    Consumer<String> con1 = s -> {
        System.out.println(s);
    };
    con1.accept("一个是听得人当真了，一个是说的人当真了");

    System.out.println("****************************");
    //语法格式六使用后
    Consumer<String> con2 = s -> System.out.println(s);

    con2.accept("一个是听得人当真了，一个是说的人当真了");

}
```

2.5 关于类型推断

在语法格式三 Lambda 表达式中的参数类型都是由编译器推断得出的。Lambda 表达式中无需指定类型，程序依然可以编译，这是因为 javac 根据程序的上下文，在后台推断出了参数的类型。Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的。这就是所谓的“类型推断”。

image-20220527103215741

举例：

```
@Test
public void test() {
    //类型推断1
    ArrayList<String> list = new ArrayList<>();
    //类型推断2
    int[] arr = {1, 2, 3};

}
```

3. 函数式(Functional)接口

3.1 什么是函数式接口

- 只包含一个抽象方法（Single Abstract Method，简称 SAM）的接口，称为函数式接口。当然该接口可以包含其他非抽象方法。
- 你可以通过 Lambda 表达式来创建该接口的对象。（若 Lambda 表达式抛出一个受检异常(即：非运行时异常)，那么该异常需要在目标接口的抽象方法上进行声明）。
- 我们可以在一个接口上使用 `@FunctionalInterface` 注解，这样做可以检查它是否是一个函数式接口。同时 javadoc 也会包含一条声明，说明这个接口是一个函数式接口。
- 在 `java.util.function` 包下定义了 Java 8 的丰富的函数式接口

3.2 如何理解函数式接口

- Java 从诞生日起就是一直倡导“一切皆对象”，在 Java 里面面向对象(OOP)编程是一切。但是随着 python、scala 等语言的兴起和新技术的挑战，Java 不得不做出调整以便支持更加广泛的技术要求，也即 Java 不但可以支持 OOP 还可以支持 OOF（面向函数编程）
  - 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。目前 C++，C#，Python，Scala 等均支持 Lambda 表达式。
- 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。
- 在函数式编程语言当中，函数被当做一等公民对待。在将函数作为一等公民的编程语言中，Lambda 表达式的类型是函数。但是在 Java8 中，有所不同。在 Java8 中，Lambda 表达式是对象，而不是函数，它们必须依附于一类特别的对象类型——函数式接口。
- 简单的说，在 Java8 中，Lambda 表达式就是一个函数式接口的实例。这就是 Lambda 表达式和函数式接口的关系。也就是说，只要一个对象是函数式接口的实例，那么该对象就可以用 Lambda 表达式来表示。

3.3 举例

举例 1：

```java
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executi
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

image-20220527111442115

举例 2：

函数式接口中未使用泛型：

```java
@FunctionalInterface
public interface MyNumber{
    public double getValue();
}
```

函数式接口中使用泛型：

```java
@FunctionalInterface
public interface MyFunc<T>{
    public T getValue(T t);
}
```

image-20220527111621424

作为参数传递 Lambda 表达式：

```java
public String toUpperString(MyFunc<String> mf, String str){
        return mf.getValue(str);
}
```

实例化

作为参数传递 Lambda 表达式：

```java
String newStr = toUpperString(
        (str) -> str.toUpperCase(), "abcdef");
System.out.println(newStr);
```

image-20220527111751485

作为参数传递 Lambda 表达式：为了将 Lambda 表达式作为参数传递，接收 Lambda 表达式的参数类型必须是与该 Lambda 表达式兼容的函数式接口的类型。

3.4 Java 内置函数式接口

3.4.1 之前的函数式接口

之前学过的接口，有些就是函数式接口，比如：

- 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)

3.4.2 四大核心函数式接口

函数式接口	称谓	参数类型	用途
`Consumer<T>`	消费型接口	T	对类型为 T 的对象应用操作，包含方法：`void accept(T t)`
`Supplier<T>`	供给型接口	无	返回类型为 T 的对象，包含方法：`T get()`
`Function<T, R>`	函数型接口	T	对类型为 T 的对象应用操作，并返回结果。结果是 R 类型的对象。包含方法：`R apply(T t)`
`Predicate<T>`	判断型接口	T	确定类型为 T 的对象是否满足某约束，并返回 boolean 值。包含方法：`boolean test(T t)`

3.4.3 其它接口

类型 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
BiConsumer<T,U>	void accept(T t, U u)	接收两个对象用于完成功能

接口名	抽象方法	描述
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

类型 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
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

类型 3：函数型接口

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
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

接口名	抽象方法	描述
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
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

接口名	抽象方法	描述
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

类型 4：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
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

3.4.4 内置接口代码演示

举例 1：

```
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#");
        //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。
        list.forEach(s -> System.out.println(s));
    }
}
```

举例 2：

```
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());
```

```
    }
}
```

举例 3：

```
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));

        //用于删除集合中满足filter 指定的条件判断的。
        //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));
    }
}
```

举例 4：

```
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"));
    }
}
```

3.4.5 练习

练习 1：无参无返回值形式

假如有自定义函数式接口 Call 如下：

```
public interface Call {
    void shout();
}
```

在测试类中声明一个如下方法：

```
public static void callSomething(Call call){
            call.shout();
}
```

在测试类的 main 方法中调用 callSomething 方法，并用 Lambda 表达式为形参 call 赋值，可以喊出任意你想说的话。

```
public class TestLambda {
      public static void main(String[] args) {
            callSomething(()->System.out.println("回家吃饭"));
            callSomething(()->System.out.println("我爱你"));
            callSomething(()->System.out.println("滚蛋"));
            callSomething(()->System.out.println("回来"));
      }
      public static void callSomething(Call call){
            call.shout();
      }
}
interface Call {
    void shout();
}
```

练习 2：消费型接口

代码示例：Consumer 接口

在 JDK1.8 中 Collection 集合接口的父接口 Iterable 接口中增加了一个默认方法：

`public default void forEach(Consumer<? super T> action)`遍历 Collection 集合的每个元素，执行“xxx 消费型”操作。

在 JDK1.8 中 Map 集合接口中增加了一个默认方法：

`public default void forEach(BiConsumer<? super K,? super V> action)`遍历 Map 集合的每对映射关系，执行“xxx 消费型”操作。

案例：

（1）创建一个 Collection 系列的集合，添加一些字符串，调用 forEach 方法遍历查看

（2）创建一个 Map 系列的集合，添加一些(key,value)键值对，调用 forEach 方法遍历查看

示例代码：

```
    @Test
    public void test1(){
        List<String> list = Arrays.asList("hello","java","lambda","
atguigu");
        list.forEach(s -> System.out.println(s));
    }
    @Test
    public void test2(){
        HashMap<Integer,String> map = new HashMap<>();
        map.put(1, "hello");
        map.put(2, "java");
        map.put(3, "lambda");
        map.put(4, "atguigu");
        map.forEach((k,v) -> System.out.println(k+"->"+v));
    }
```

练习 3：供给型接口

代码示例：Supplier 接口

在 JDK1.8 中增加了 StreamAPI，java.util.stream.Stream 是一个数据流。这个类型有一个静态方法：

`public static <T> Stream<T> generate(Supplier<T> s)`可以创建 Stream 的对象。而又包含一个 forEach 方法可以遍历流中的元素：`public void forEach(Consumer<? super T> action)`。

案例：

现在请调用 Stream 的 generate 方法，来产生一个流对象，并调用 Math.random()方法来产生数据，为 Supplier 函数式接口的形参赋值。最后调用 forEach 方法遍历流中的数据查看结果。

```
    @Test
    public void test2(){
        Stream.generate(() -> Math.random()).forEach(num -> System.
out.println(num));
    }
```

练习 4：功能型接口

代码示例：Function<T,R>接口

在 JDK1.8 时 Map 接口增加了很多方法，例如：

public default void replaceAll(BiFunction<? super K,? super V,? extends V> function)按照 function 指定的操作替换 map 中的 value。

public default void forEach(BiConsumer<? super K,? super V> action)遍历 Map 集合的每对映射关系，执行“xxx 消费型”操作。

案例：

（1）声明一个 Employee 员工类型，包含编号、姓名、薪资。

（2）添加 n 个员工对象到一个 HashMap<Integer,Employee>集合中，其中员工编号为 key，员工对象为 value。

（3）调用 Map 的 forEach 遍历集合

（4）调用 Map 的 replaceAll 方法，将其中薪资低于 10000 元的，薪资设置为 10000。

（5）再次调用 Map 的 forEach 遍历集合查看结果

Employee 类：

```
class Employee{
      private int id;
      private String name;
      private double salary;
      public Employee(int id, String name, double salary) {
            super();
            this.id = id;
            this.name = name;
            this.salary = salary;
      }
      public Employee() {
            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;
      }
      public double getSalary() {
            return salary;
      }
```

```
	public void setSalary(double salary) {
		this.salary = salary;
	}
	@Override
	public String toString() {
		return "Employee [id=" + id + ", name=" + name + ", salary=
" + salary + "]";
	}

}
```

测试类：

```
import java.util.HashMap;

public class TestLambda {
	public static void main(String[] args) {
		HashMap<Integer,Employee> map = new HashMap<>();
		Employee e1 = new Employee(1, "张三", 8000);
		Employee e2 = new Employee(2, "李四", 9000);
		Employee e3 = new Employee(3, "王五", 10000);
		Employee e4 = new Employee(4, "赵六", 11000);
		Employee e5 = new Employee(5, "钱七", 12000);

		map.put(e1.getId(), e1);
		map.put(e2.getId(), e2);
		map.put(e3.getId(), e3);
		map.put(e4.getId(), e4);
		map.put(e5.getId(), e5);

		map.forEach((k,v) -> System.out.println(k+"="+v));
		System.out.println();

		map.replaceAll((k,v)->{
			if(v.getSalary()<10000){
				v.setSalary(10000);
			}
			return v;
		});
		map.forEach((k,v) -> System.out.println(k+"="+v));
	}
}
```

练习 5：判断型接口

代码示例：Predicate 接口

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。

`public default void forEach(Consumer<? super T> action)`遍历 Collection 集合的每个元素，执行“xxx 消费型”操作。

案例：

（1）添加一些字符串到一个 Collection 集合中

（2）调用 forEach 遍历集合

（3）调用 removeIf 方法，删除其中字符串的长度<5 的

（4）再次调用 forEach 遍历集合

```
import java.util.ArrayList;

public class TestLambda {
      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");

            list.forEach(str->System.out.println(str));
            System.out.println();

            list.removeIf(str->str.length()<5);
            list.forEach(str->System.out.println(str));
      }
}
```

练习 6：判断型接口

案例：

（1）声明一个 Employee 员工类型，包含编号、姓名、性别，年龄，薪资。

（2）声明一个 EmployeeSerice 员工管理类，包含一个 ArrayList 集合的属性 all，在 EmployeeSerice 的构造器中，创建一些员工对象，为 all 集合初始化。

（3）在 EmployeeSerice 员工管理类中，声明一个方法：ArrayList get(Predicate p)，即将满足 p 指定的条件的员工，添加到一个新的 ArrayList 集合中返回。

（4）在测试类中创建 EmployeeSerice 员工管理类的对象，并调用 get 方法，分别获取：

- 所有员工对象
- 所有年龄超过 35 的员工
- 所有薪资高于 15000 的女员工
- 所有编号是偶数的员工
- 名字是“张三”的员工
- 年龄超过 25，薪资低于 10000 的男员工

示例代码：

Employee 类：

```
public class Employee{
      private int id;
      private String name;
      private char gender;
      private int age;
      private double salary;

      public Employee(int id, String name, char gender, int age, double
 salary) {
            super();
            this.id = id;
            this.name = name;
            this.gender = gender;
            this.age = age;
            this.salary = salary;
      }
      public Employee() {
            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;
      }
      public double getSalary() {
            return salary;
      }
```

```
	public void setSalary(double salary) {
		this.salary = salary;
	}
	@Override
	public String toString() {
		return "Employee [id=" + id + ", name=" + name + ", gender=
" + gender + ", age=" + age + ", salary=" + salary
		+ "]";
	}
}
```

员工管理类：

```
class EmployeeService{
	private ArrayList<Employee> all;
	public EmployeeService(){
		all = new ArrayList<Employee>();
		all.add(new Employee(1, "张三", '男', 33, 8000));
		all.add(new Employee(2, "翠花", '女', 23, 18000));
		all.add(new Employee(3, "无能", '男', 46, 8000));
		all.add(new Employee(4, "李四", '女', 23, 9000));
		all.add(new Employee(5, "老王", '男', 23, 15000));
		all.add(new Employee(6, "大嘴", '男', 23, 11000));
	}
	public ArrayList<Employee> get(Predicate<Employee> p){
		ArrayList<Employee> result = new ArrayList<Employee>();
		for (Employee emp : result) {
			if(p.test(emp)){
				result.add(emp);
			}
		}
		return result;
	}
}
```

测试类：

```
public class TestLambda {
	public static void main(String[] args) {
		EmployeeService es = new EmployeeService();

		es.get(e -> true).forEach(e->System.out.println(e));
		System.out.println();
		es.get(e -> e.getAge()>35).forEach(e->System.out.println
(e));
		System.out.println();
		es.get(e -> e.getSalary()>15000 && e.getGender()=='女').for
Each(e->System.out.println(e));
		System.out.println();
		es.get(e -> e.getId()%2==0).forEach(e->System.out.println
```

```
(e));
            System.out.println();
            es.get(e -> "张三".equals(e.getName())).forEach(e->System.o
ut.println(e));
            System.out.println();
            es.get(e -> e.getAge()>25 && e.getSalary()<10000 && e.getGe
nder()=='男').forEach(e->System.out.println(e));
      }
}
```

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。

4.1 方法引用

当要传递给 Lambda 体的操作，已经有实现的方法了，可以使用方法引用！

方法引用可以看做是 Lambda 表达式深层次的表达。换句话说，方法引用就是 Lambda 表达式，也就是函数式接口的一个实例，通过方法的名字来指向一个方法，可以认为是 Lambda 表达式的一个语法糖。

> 语法糖（Syntactic sugar），也译为糖衣语法，是由英国计算机科学家彼得·约翰·兰达（Peter J. Landin）发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。

4.1.1 方法引用格式

- 格式：使用方法引用操作符 “::” 将类(或对象) 与 方法名分隔开来。
  - 两个:中间不能有空格，而且必须英文状态下半角输入
- 如下三种主要使用情况：
  - 情况 1：对象 :: 实例方法名
  - 情况 2：类 :: 静态方法名
  - 情况 3：类 :: 实例方法名

4.1.2 方法引用使用前提

要求 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

要求 2：

针对情况 1：函数式接口中的抽象方法 a 在被重写时使用了某一个对象的方法 b。如果方法 a 的形参列表、返回值类型与方法 b 的形参列表、返回值类型都相同，则我们可以使用方法 b 实现对方法 a 的重写、替换。

针对情况 2：函数式接口中的抽象方法 a 在被重写时使用了某一个类的静态方法 b。如果方法 a 的形参列表、返回值类型与方法 b 的形参列表、返回值类型都相同，则我们可以使用方法 b 实现对方法 a 的重写、替换。

针对情况 3：函数式接口中的抽象方法 a 在被重写时使用了某一个对象的方法 b。如果方法 a 的返回值类型与方法 b 的返回值类型相同，同时方法 a 的形参列表中有 n 个参数，方法 b 的形参列表有 n-1 个参数，且方法 a 的第 1 个参数作为方法 b 的调用者，且方法 a 的后 n-1 参数与方法 b 的 n-1 参数匹配（类型相同或满足多态场景也可以）

例如：t->System.out.println(t)

() -> Math.random() 都是无参

4.1.3 举例

```
public class MethodRefTest {

	// 情况一：对象 :: 实例方法
	//Consumer 中的void accept(T t)
	//PrintStream 中的void println(T t)
	@Test
	public void test1() {
		Consumer<String> con1 = str -> System.out.println(str);
		con1.accept("北京");

		System.out.println("*******************");
		PrintStream ps = System.out;
		Consumer<String> con2 = ps::println;
		con2.accept("beijing");
	}

	//Supplier 中的T get()
	//Employee 中的String getName()
	@Test
	public void test2() {
		Employee emp = new Employee(1001,"Tom",23,5600);

		Supplier<String> sup1 = () -> emp.getName();
		System.out.println(sup1.get());
```

```
        System.out.println("******************");
        Supplier<String> sup2 = emp::getName;
        System.out.println(sup2.get());

    }

    // 情况二：类 :: 静态方法
    //Comparator 中的int compare(T t1,T t2)
    //Integer 中的int compare(T t1,T t2)
    @Test
    public void test3() {
        Comparator<Integer> com1 = (t1,t2) -> Integer.compare(t1,t
2);
        System.out.println(com1.compare(12,21));

        System.out.println("******************");

        Comparator<Integer> com2 = Integer::compare;
        System.out.println(com2.compare(12,3));

    }

    //Function 中的R apply(T t)
    //Math 中的Long round(Double d)
    @Test
    public void test4() {
        Function<Double,Long> func = new Function<Double, Long>() {
            @Override
            public Long apply(Double d) {
                return Math.round(d);
            }
        };

        System.out.println("******************");

        Function<Double,Long> func1 = d -> Math.round(d);
        System.out.println(func1.apply(12.3));

        System.out.println("******************");

        Function<Double,Long> func2 = Math::round;
        System.out.println(func2.apply(12.6));
    }

    // 情况三：类 :: 实例方法  (有难度)
    // Comparator 中的int comapre(T t1,T t2)
    // String 中的int t1.compareTo(t2)
```

```java
    @Test
    public void test5() {
        Comparator<String> com1 = (s1,s2) -> s1.compareTo(s2);
        System.out.println(com1.compare("abc","abd"));

        System.out.println("******************");

        Comparator<String> com2 = String :: compareTo;
        System.out.println(com2.compare("abd","abm"));
    }

    //BiPredicate 中的boolean test(T t1, T t2);
    //String 中的boolean t1.equals(t2)
    @Test
    public void test6() {
        BiPredicate<String,String> pre1 = (s1,s2) -> s1.equals(s2);
        System.out.println(pre1.test("abc","abc"));

        System.out.println("******************");
        BiPredicate<String,String> pre2 = String :: equals;
        System.out.println(pre2.test("abc","abd"));
    }

    // Function 中的R apply(T t)
    // Employee 中的String getName();
    @Test
    public void test7() {
        Employee employee = new Employee(1001, "Jerry", 23, 6000);


        Function<Employee,String> func1 = e -> e.getName();
        System.out.println(func1.apply(employee));

        System.out.println("******************");
        Function<Employee,String> func2 = Employee::getName;
        System.out.println(func2.apply(employee));
    }

}
```

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。

格式：类名::new

举例：

```java
public class ConstructorRefTest {
      //构造器引用
    //Supplier 中的T get()
    //Employee 的空参构造器: Employee()
    @Test
    public void test1(){

        Supplier<Employee> sup = new Supplier<Employee>() {
            @Override
            public Employee get() {
                return new Employee();
            }
        };
        System.out.println("******************");

        Supplier<Employee>  sup1 = () -> new Employee();
        System.out.println(sup1.get());

        System.out.println("******************");

        Supplier<Employee>  sup2 = Employee :: new;
        System.out.println(sup2.get());
    }

      //Function 中的R apply(T t)
    @Test
    public void test2(){
        Function<Integer,Employee> func1 = id -> new Employee(id);
        Employee employee = func1.apply(1001);
        System.out.println(employee);

        System.out.println("******************");

        Function<Integer,Employee> func2 = Employee :: new;
        Employee employee1 = func2.apply(1002);
        System.out.println(employee1);

    }

      //BiFunction 中的R apply(T t,U u)
    @Test
    public void test3(){
        BiFunction<Integer,String,Employee> func1 = (id,name) -> new Em
ployee(id,name);
        System.out.println(func1.apply(1001,"Tom"));

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

```
        BiFunction<Integer,String,Employee> func2 = Employee :: new;
        System.out.println(func2.apply(1002,"Tom"));

    }

}

package com.atguigu.java2;

/**
 * @author 尚硅谷-宋红康 邮箱: shkstart@126.com
 */
public class Employee {

      private int id;
      private String name;
      private int age;
      private double salary;

      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 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 Employee() {
        System.out.println("Employee().....");
    }

    public Employee(int id) {
        this.id = id;
        System.out.println("Employee(int id).....");
    }

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public Employee(int id, String name, int age, double salary) {

        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee{" + "id=" + id + ", name='" + name + '\''
+ ", age=" + age + ", salary=" + salary + '}';
    }

}
```

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。

格式：数组类型名::new

举例：

```
//数组引用
//Function 中的R apply(T t)
@Test
public void test4(){
    Function<Integer,String[]> func1 = length -> new String[length];
    String[] arr1 = func1.apply(5);
    System.out.println(Arrays.toString(arr1));
```

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

    Function<Integer,String[]> func2 = String[] :: new;
    String[] arr2 = func2.apply(10);
    System.out.println(Arrays.toString(arr2));

}
```

5. 强大的 Stream API

5.1 说明

- 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。
- 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 可以极大提供 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 提供了一种高效且易于使用的处理数据的方式。

5.2 为什么要使用 Stream API

实际开发中，项目中多数数据源都来自于 Mysql，Oracle 等。但现在数据源可以更多了，有 MongDB，Radis 等，而这些 NoSQL 的数据就需要 Java 层面去处理。

5.3 什么是 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。

Stream 和 Collection 集合的区别：**Collection 是一种静态的内存数据结构，讲的是数据，而 Stream 是有关计算的，讲的是计算。**前者是主要面向内存，存储在内存中，后者主要是面向 CPU，通过 CPU 实现计算。

注意：

①Stream 自己不会存储元素。

②Stream 不会改变源对象。相反，他们会返回一个持有结果的新 Stream。

③Stream 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。

5.4 Stream 的操作三个步骤

1- 创建 Stream 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(终端操作) 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。

image-20220514180803311

5.4.1 创建 Stream 实例

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：

- default Stream stream() : 返回一个顺序流
- default Stream parallelStream() : 返回一个并行流

```
@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();
}
```

方式二：通过数组

Java8 中的 Arrays 的静态方法 stream() 可以获取数组流：

- static Stream stream(T[] array): 返回一个流

- public static IntStream stream(int[] array)
- public static LongStream stream(long[] array)
- public static DoubleStream stream(double[] array)

```
@Test
public void test02(){
    String[] arr = {"hello","world"};
    Stream<String> stream = Arrays.stream(arr);
}

@Test
public void test03(){
    int[] arr = {1,2,3,4,5};
    IntStream stream = Arrays.stream(arr);
}
```

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) : 返回一个流

```
@Test
public void test04(){
    Stream<Integer> stream = Stream.of(1,2,3,4,5);
    stream.forEach(System.out::println);
}
```

方式四：创建无限流(了解)

可以使用静态方法 Stream.iterate() 和 Stream.generate(), 创建无限流。

- 迭代 public static Stream iterate(final T seed, final UnaryOperator f)
- 生成 public static Stream generate(Supplier s)

```
// 方式四：创建无限流
@Test
public void test05() {
      // 迭代
      // public static<T> Stream<T> iterate(final T seed, final
      // UnaryOperator<T> f)
      Stream<Integer> stream = Stream.iterate(0, x -> x + 2);
      stream.limit(10).forEach(System.out::println);
```

```
        // 生成
        // public static<T> Stream<T> generate(Supplier<T> s)
        Stream<Double> stream1 = Stream.generate(Math::random);
        stream1.limit(10).forEach(System.out::println);
}
```

5.4.2 一系列中间操作

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”。

1-筛选与切片

方 法	描 述
filter(Predicatep)	接收 Lambda， 从流中排除某些元素
distinct()	筛选，通过流所生成元素的 hashCode() 和 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

2-映 射

方法	描述
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

3-排序

方法	描述
sorted()	产生一个新流，其中按自然顺序排序

方法	描述
sorted(Comparator com)	产生一个新流，其中按比较器顺序排序

```java
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
```

```java
    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);
    }
}
```

5.4.3 终止操作

- 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。

1-匹配与查找

方法	描述
allMatch(Predicate p)	检查是否匹配所有元素
anyMatch(Predicate p)	检查是否至少匹配一个元素

方法	描述
noneMatch(Predicate p)	检查是否没有匹配所有元素
findFirst()	返回第一个元素
findAny()	返回当前流中的任意元素
count()	返回流中元素总数
max(Comparator c)	返回流中最大值
min(Comparator c)	返回流中最小值
forEach(Consumer c)	内部迭代(使用 Collection 接口需要用户去做迭代，称为外部迭代。 相反，Stream API 使用内部迭代——它帮你把迭代做了)

2-规约

方法	描述
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
reduce(BinaryOperator b)	可以将流中元素反复结合起来，得到一个值。返回 Optional

备注：map 和 reduce 的连接通常称为 map-reduce 模式，因 Google 用它来进行网络搜索而出名。

3-收集

方 法	描 述
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，具体方法与实例如下表：

方法	返回类型	作用
toList	List	把流中元素收集到 List

```
List<Employee> emps= list.stream().collect(Collectors.toList());
```

方法	返回类型	作用
toSet	Set	把流中元素收集到 Set

```
Set<Employee> emps= list.stream().collect(Collectors.toSet());
```

方法	返回类型	作用
toCollection	Collection	把流中元素收集到创建的集合

```
Collection<Employee> emps =list.stream().collect(Collectors.toCollection(ArrayList::new));
```

方法	返回类型	作用
counting	Long	计算流中元素的个数

```
long count = list.stream().collect(Collectors.counting());
```

方法	返回类型	作用
summingInt	Integer	对流中元素的整数属性求和

```
int total=list.stream().collect(Collectors.summingInt(Employee::getSalary));
```

方法	返回类型	作用
averagingInt	Double	计算流中元素 Integer 属性的平均值

```
double avg = list.stream().collect(Collectors.averagingInt(Employee::getSalary));
```

方法	返回类型	作用
summarizingInt	IntSummaryStatistics	收集流中 Integer 属性的统计值。如：平均值

```
int SummaryStatisticsiss= list.stream().collect(Collectors.summarizingInt(Employee::getSalary));
```

方法	返回类型	作用
joining	String	连接流中每个字符串

```
String str= list.stream().map(Employee::getName).collect(Collectors.joining());
```

方法	返回类型	作用
maxBy	Optional	根据比较器选择最大值

```
Optional<Emp>max= list.stream().collect(Collectors.maxBy(comparingInt(Employee::getSalary)));
```

方法	返回类型	作用
minBy	Optional	根据比较器选择最小值

```
Optional<Emp> min = list.stream().collect(Collectors.minBy(comparingInt(Employee::getSalary)));
```

方法	返回类型	作用

方法	返回类型	作用
reducing	归约产生的类型	从一个作为累加器的初始值开始，利用 BinaryOperator 与流中元素逐个结合，从而归约成单个值

```
int total=list.stream().collect(Collectors.reducing(0, Employee::getSal
ar, Integer::sum));
```

方法	返回类型	作用
collectingAndThen	转换函数返回的类型	包裹另一个收集器，对其结果转换函数

```
int how= list.stream().collect(Collectors.collectingAndThen(Collectors.
toList(), List::size));
```

方法	返回类型	作用
groupingBy	Map<K, List>	根据某属性值对流分组，属性为 K，结果为 V

```
Map<Emp.Status, List<Emp>> map= list.stream().collect(Collectors.groupi
ngBy(Employee::getStatus));
```

方法	返回类型	作用
partitioningBy	Map<Boolean, List>	根据 true 或 false 进行分区

```
Map<Boolean,List<Emp>> vd = list.stream().collect(Collectors.partitioni
ngBy(Employee::getManage));
```

举例：

```
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 a
pply(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);
    }
}
```

5.5 Java9 新增 API

新增 1：Stream 实例化方法

ofNullable()的使用：

Java 8 中 Stream 不能完全为 null，否则会报空指针异常。而 Java 9 中的 ofNullable 方法允许我们创建一个单元素 Stream，可以包含一个非空元素，也可以创建一个空 Stream。

```
//报NullPointerException
//Stream<Object> stream1 = Stream.of(null);
//System.out.println(stream1.count());

//不报异常，允许通过
Stream<String> stringStream = Stream.of("AA", "BB", null);
System.out.println(stringStream.count());//3

//不报异常，允许通过
List<String> list = new ArrayList<>();
list.add("AA");
list.add(null);
System.out.println(list.stream().count());//2
//ofNullable()：允许值为null
Stream<Object> stream1 = Stream.ofNullable(null);
```

```
System.out.println(stream1.count());//0

Stream<String> stream = Stream.ofNullable("hello world");
System.out.println(stream.count());//1
```

iterator()重载的使用：

```
//原来的控制终止方式:
Stream.iterate(1,i -> i + 1).limit(10).forEach(System.out::println);

//现在的终止方式:
Stream.iterate(1,i -> i < 100,i -> i + 1).forEach(System.out::println);
```

5.6 练习

案例：

现在有两个 ArrayList 集合存储队伍当中的多个成员姓名，要求使用传统的 for 循环（或增强 for 循环）依次进行以 下若干操作步骤：

1. 第一个队伍只要名字为 3 个字的成员姓名；存储到一个新集合中。
2. 第一个队伍筛选之后只要前 3 个人；存储到一个新集合中。
3. 第二个队伍只要姓张的成员姓名；存储到一个新集合中。
4. 第二个队伍筛选之后不要前 2 个人；存储到一个新集合中。
5. 将两个队伍合并为一个队伍；存储到一个新集合中。
6. 根据姓名创建 Person 对象；存储到一个新集合中。
7. 打印整个队伍的 Person 对象信息。

Person 类的代码为：

```
public class Person {
    private String name;
    public Person() {}
    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 + "'}";
    }
}
```

两个队伍（集合）的代码如下：

```
public static void main(String[] args) {
        //第一支队伍
         ArrayList<String> one = new ArrayList<>();
         one.add("迪丽热巴");
         one.add("宋远桥");
         one.add("苏星河");
         one.add("石破天");
         one.add("石中玉");
         one.add("老子");
         one.add("庄子");
         one.add("洪七公");
         //第二支队伍
         ArrayList<String> two = new ArrayList<>();
         two.add("古力娜扎");
         two.add("张无忌");
         two.add("赵丽颖");
         two.add("张三丰");
         two.add("尼古拉斯赵四");
         two.add("张天爱");
         two.add("张二狗");

        // ....编写代码完成题目要求
    }
```

参考答案：

```
public static void main(String[] args) {
        //第一支队伍
         ArrayList<String> one = new ArrayList<>();
         one.add("迪丽热巴");
         one.add("宋远桥");
         one.add("苏星河");
         one.add("石破天");
         one.add("石中玉");
         one.add("老子");
         one.add("庄子");
         one.add("洪七公");
```

```
        //第二支队伍
        ArrayList<String> two = new ArrayList<>();
        two.add("古力娜扎");
        two.add("张无忌");
        two.add("赵丽颖");
        two.add("张三丰");
        two.add("尼古拉斯赵四");
        two.add("张天爱");
        two.add("张二狗");

            // 第一个队伍只要名字为3 个字的成员姓名;
        // 第一个队伍筛选之后只要前3 个人;
        Stream<String> streamOne = one.stream().filter(s -> s.length
() == 3).limit(3);

        // 第二个队伍只要姓张的成员姓名;
        // 第二个队伍筛选之后不要前2 个人;
        Stream<String> streamTwo = two.stream().filter(s -> s.startsWit
h("张")).skip(2);

        // 将两个队伍合并为一个队伍;
        // 根据姓名创建Person 对象;
        // 打印整个队伍的Person 对象信息。
        Stream.concat(streamOne, streamTwo).map(Person::new).forEach(Sy
stem.out::println);

}
```

6. 新语法结构

6.1 Java 的 REPL 工具： jShell 命令

这是 JDK9 的新特性。

产生背景

像 Python 和 Scala 之类的语言早就有交互式编程环境 REPL (read - evaluate - print - loop)了，以交互式的方式对语句和表达式进行求值。开发者只需要输入一些代码，就可以在编译前获得对程序的反馈。而之前的 Java 版本要想执行代码，必须创建文件、声明类、提供测试方法方可实现。

设计理念

即写即得、快速运行

实现目标

Java 9 中终于拥有了 REPL 工具：jShell。利用 jShell 在没有创建类的情况下直接声明变量，计算表达式，执行语句。即开发时可以在命令行里直接运行 Java 的代码，而无需创建 Java 文件，无需跟人解释”public static void main(String[] args)”这句废话。

jShell 也可以从文件中加载语句或者将语句保存到文件中。

jShell 也可以是 tab 键进行自动补全和自动添加分号。

使用举例

调出 jShell

```
C:\Users\Administrator>jshell
|  欢迎使用 JShell -- 版本 9.0.1
|  要大致了解该版本，请键入：/help intro
```

获取帮助

```
jshell> /help intro
|
|  intro
|
|  使用 jshell 工具可以执行 Java 代码，从而立即获取结果。
|  您可以输入 Java 定义 (变量，方法，类，等等)，例如：  int x = 8
|  或 Java 表达式，例如：  x + x
|  或 Java 语句或导入。
|  这些小块的 Java 代码称为 '片段'。
|
|  这些 jshell 命令还可以让您了解和
|  控制您正在执行的操作，例如：  /list
```

基本使用

```
jshell> System.out.println("你好! world");
你好! world

jshell> int i = 10;
i ==> 10

jshell> int j = 20;
j ==> 20

jshell> int k = i + j;
k ==> 30

jshell> System.out.println(k);
30
```

```
jshell> public int add(int m,int n){
   ...> return m + n;
   ...> }
|  已创建 方法 add(int,int)

jshell> int k = add(1,2);
k ==> 3

jshell> System.out.println(k);
3
```

Tips：在 JShell 环境下，语句末尾的“;” 是可选的。但推荐还是最好加上。提高代码可读性。

导入指定的包

```
jshell> import java.util.*;
```

默认已经导入如下的所有包：（包含 java.lang 包）

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

只需按下 Tab 键，就能自动补全代码

```
jshell> Sy
SyncFailedException   SynchronousQueue      System

jshell> System.out
out

jshell> System.out.
append(        checkError()   close()        equals(        flush()
format(        getClass()     hashCode()     notify()       notifyAll()
print(         printf(        println(       toString()     wait(
write(

jshell> System.out.
```

列出当前 session 里所有有效的代码片段

```
jshell> /list

   1 : public int add(int m,int n){
       return m + n;
       }
   2 : int k = add(1,2);
   3 : System.out.println(k);

jshell>
```

查看当前 session 下所有创建过的变量

```
jshell> /var
|     int k = 3

jshell>
```

查看当前 session 下所有创建过的方法

```
jshell> /methods
|     int add(int,int)

jshell>
```

Tips：我们还可以重新定义相同方法名和参数列表的方法，即为对现有方法的修改（或覆盖）。

使用外部代码编辑器来编写 Java 代码

```
jshell> /edit add
|  已修改 方法 add(int,int)

jshell> add(1,2);
a

jshell> int j = 5;
j ==> 5

jshell> /edit j

jshell>
```

从外部文件加载源代码

```
/**
 * Created by songhongkang on 2017/12/27 0020.
 */
public void printHello() {
    System.out.println("马上 2018 年了，尚硅谷祝所有的谷粉元旦快乐！");
}
printHello();
```

使用/open 命令调用：

```
jshell> /open E:\teach\01_Java9\HelloWorld.java
马上2018年了，尚硅谷祝所有的谷粉元旦快乐！

jshell>
```

没有受检异常（编译时异常）

```
jshell> URL url = new URL("http://www.atguigu.com");
url ==> http://www.atguigu.com

jshell>
```

说明：本来应该强迫我们捕获一个 IOException，但却没有出现。因为 jShell 在后台为我们隐藏了。

退出 jShell

```
jshell> /exit
|  再见
```

6.2 异常处理之 try-catch

在 Java 8 之前，我们习惯于这样处理资源的关闭：

```
InputStreamReader reader = null;
try{
    reader = new InputStreamReader(System.in);

    //流的操作
    reader.read();
}catch (IOException e){
    e.printStackTrace();
}finally{
    if(reader != null){
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

java 8 中，可以实现资源的自动关闭，但是要求执行后必须关闭的所有资源必须在 try 子句中初始化，否则编译不通过。如下例所示：

```
try(InputStreamReader reader = new InputStreamReader(System.in)){

}catch (IOException e){
    e.printStackTrace();
}
```

java 9 中，用资源语句编写 try 将更容易，我们可以在 try 子句中使用已经初始化过的资源，此时的资源是 final 的：

```
InputStreamReader reader = new InputStreamReader(System.in);
OutputStreamWriter writer = new OutputStreamWriter(System.out);
try(reader;writer){
     //reader 是final 的，不可再被赋值
//   reader = null;
}catch (IOException e){
     e.printStackTrace();
}
```

6.3 局部变量类型推断

这是 JDK 10 的新特性。

产生背景

开发者经常抱怨 Java 中引用代码的程度。局部变量的显示类型声明，常常被认为是不必须的，给一个好听的名字经常可以很清楚的表达出下面应该怎样继续。

试图通过减少编写 Java 代码相关的仪式，允许开发人员省略通常不必要的局部变量类型声明以增强 Java 语言的体验性。此功能可以这样声明，如

```
var list= new ArrayList<String>();
```

好处

减少了啰嗦和形式的代码，避免了信息冗余，而且对齐了变量名，更容易阅读！

使用举例

```
//1.局部变量的初始化
var list = new ArrayList<>();

//2.增强for 循环中的索引
for(var v : list) {
    System.out.println(v);
}

//3.传统for 循环中
for(var i = 0;i < 100;i++) {
    System.out.println(i);
}
```

适用场景：

场景一：类实例化时

作为 Java 开发者，在声明一个变量时，我们总是习惯了敲打两次变量类型，第一次用于声明变量类型，第二次用于构造器。

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();
```

场景二：返回值类型含复杂泛型结构

变量的声明类型书写复杂且较长，尤其是加上泛型的使用

```
Iterator<Map.Entry<Integer, Student>> iterator = set.iterator();
```

不适用场景

初始值为 null

```
jshell> var s = null;
   错误:
   无法推断本地变量 s 的类型
     (变量初始化程序为 'null')
```

Lambda 表达式

```
jshell> var r = () -> Math.random();
|  错误:
|  无法推断本地变量 r 的类型
|    (lambda 表达式需要显式目标类型)
|  var r = () -> Math.random();
|  ^--------------------------^
```

方法引用

```
shell> var r = System.out::println;
   错误:
   无法推断本地变量 r 的类型
     (方法引用需要显式目标类型)
   var r = System.out::println;
```

为数组静态初始化

```
jshell> var arr= {"a","b","c"};
|  错误:
|  无法推断本地变量 arr 的类型
|    (数组初始化程序需要显式目标类型)
|  var arr= {"a","b","c"};
|  ^--------------------^
```

没有初始化的局部变量声明

```
jshell> var i;
|  错误:
|  无法推断本地变量 i 的类型
|    (无法在不带初始化程序的变量上使用 'var')
```

方法的返回类型

```
jshell> public var getInfo(){
   ...>
   ...>    return "这是一个方法";
   ...> }
   错误:
   此处不允许使用 'var'
```

方法的参数类型

```
jshell> public void swap(var i){
   ...> }
   错误:
   此处不允许使用 'var'
```

构造器的参数类型

```
jshell> public class Student{
   ...>
   ...>    public Student(var name){
   ...>    }
   ...> }
   错误:
   此处不允许使用 'var'
     public Student(var name){
                    ^-^
```

属性

```
jshell> public class Student{
   ...>    var name;
   ...> }
   错误:
   此处不允许使用 'var'
```

catch 块

```
jshell> try{
   ...>    int i=1/0;
   ...> }catch(var e){
   ...>    e.printStackTrace();
   ...> }
   错误:
   此处不允许使用 'var'
   }catch(var e){
```

工作原理

在处理 var 时，编译器先是查看表达式右边部分，也就是所谓的构造器，并将它作为变量的类型，然后将该类型写入字节码当中。

注意：

- var 不是一个关键字

你不需要担心变量名或方法名会与 var 发生冲突，因为 var 实际上并不是一个关键字，而是一个类型名，只有在编译器需要知道类型的地方才需要用到它。除此之外，它就是一个普通合法的标识符。也就是说，除了不能用它作为类名，其他的都可以，但极少人会用它作为类名。

- 这不是 JavaScript

首先我要说明的是，var 并不会改变 Java 是一门静态类型语言的事实。编译器负责推断出类型，并把结果写入字节码文件，就好像是开发人员自己敲入类型一样。

下面是使用 IntelliJ（实际上是 Fernflower 的反编译器）反编译器反编译出的代码：

```
URL codefx = new URL("http://codefx.org");
URLConnection connection = codefx.openConnection();
BufferedReader reader = new BufferedReader( newInputStreamReader(connec
tion.getInputStream()));
```

从代码来看，就好像之前已经声明了这些类型一样。事实上，这一特性只发生在编译阶段，与运行时无关，所以对运行时的性能不会产生任何影响。所以请放心，这不是 JavaScript。

6.4 switch 表达式

jdk12 中预览特性：

1.传统 switch 的弊端

传统的 switch 声明语句(switch statement)在使用中有一些问题：

- 匹配是自上而下的，如果忘记写 break, 后面的 case 语句不论匹配与否都会执行；
- 所有的 case 语句共用一个块范围，在不同的 case 语句定义的变量名不能重复；
- 不能在一个 case 里写多个执行结果一致的条件；
- 整个 switch 不能作为表达式返回值；

Java 12 将会对 switch 声明语句进行扩展，可将其作为增强版的 switch 语句或称为 "switch 表达式"来写出更加简化的代码。

2.何为预览语言？

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新版 Java 中来的，预览语言功能的想法是在 2018 年初被引入 Java 中的，**本质上讲，这是一种引入新特性的测试版的方法。**通过这种方式，能够根据用户反馈进行升级、更改，在极端情况下，如果没有被很好的接纳，则可以完全删除该功能。预览功能的关键在于它们没有被包含在 Java SE 规范中。

3.语法详解

扩展的 switch 语句，不仅可以作为语句（statement），还可以作为表达式（expression），并且两种写法都可以使用传统的 switch 语法，或者使用简化的“case L ->”模式匹配语法作用于不同范围并控制执行流。这些更改将简化日常编码工作，并为 switch 中的模式匹配（JEP 305）做好准备。

- 使用 Java 12 中 Switch 表达式的写法，省去了 break 语句，避免了因少写 break 而出错。
- 同时将多个 case 合并到一行，显得简洁、清晰也更加优雅的表达逻辑分支，其具体写法就是将之前的 case 语句表成了：case L ->，即如果条件匹配 case L，则执行标签右侧的代码，同时标签右侧的代码段只能是表达式、代码块或 throw 语句。
- 为了保持兼容性，case 条件语句中依然可以使用字符 : ，这时 fall-through 规则依然有效的，即不能省略原有的 break 语句，但是同一个 Switch 结构里不能混用 -> 和 : ，否则会有编译错误。并且简化后的 Switch 代码块中定

义的局部变量，其作用域就限制在代码块中，而不是蔓延到整个 Switch 结构，也不用根据不同的判断条件来给变量赋值。

4.代码举例

举例 1：

Java 12 之前

```
/**
 * @author shkstart
 * @create 2019 下午 4:47
 */
public class SwitchTest {
    public static void main(String[] args) {
        int numberOfLetters;
        Fruit fruit = Fruit.APPLE;
        switch (fruit) {
            case PEAR:
                numberOfLetters = 4;
                break;
            case APPLE:
            case GRAPE:
            case MANGO:
                numberOfLetters = 5;
                break;
            case ORANGE:
            case PAPAYA:
                numberOfLetters = 6;
                break;
            default:
                throw new IllegalStateException("No Such Fruit:" + frui
t);
        }
        System.out.println(numberOfLetters);

    }
}
enum Fruit {
    PEAR, APPLE, GRAPE, MANGO, ORANGE, PAPAYA;
}
```

如果有编码经验，你一定知道，switch 语句如果漏写了一个 break，那么逻辑往往就跑偏了，这种方式既繁琐，又容易出错。如果换成 switch 表达式，Pattern Matching 机制能够自然地保证只有单一路径会被执行：

Java 12

```
/**
 * @author shkstart
```

```
 * @create 2019 下午 10:38
 */
public class SwitchTest1 {
    public static void main(String[] args) {
        Fruit fruit = Fruit.GRAPE;
        switch(fruit){
            case PEAR -> System.out.println(4);
            case APPLE,MANGO,GRAPE -> System.out.println(5);
            case ORANGE,PAPAYA -> System.out.println(6);
            default -> throw new IllegalStateException("No Such Fruit:"
 + fruit);
        };
    }
}
```

更进一步，下面的表达式，为我们提供了优雅地表达特定场合计算逻辑的方式：

```
/**
 * @author shkstart
 * @create 2019 下午 10:44
 */
public class SwitchTest2 {
    public static void main(String[] args) {
        Fruit fruit = Fruit.GRAPE;
        int numberOfLetters = switch(fruit){
            case PEAR -> 4;
            case APPLE,MANGO,GRAPE -> 5;
            case ORANGE,PAPAYA -> 6;
            default -> throw new IllegalStateException("No Such Fruit:"
 + fruit);
        };
        System.out.println(numberOfLetters);
    }
}
```

举例 2：

Java 12 之前：

```
public class SwitchTest {

      public static void main(String[] args) {
            Week day = Week.FRIDAY;
            switch (day) {
            case MONDAY:
            case FRIDAY:
            case SUNDAY:
                  System.out.println(6);
                  break;
            case TUESDAY:
```

```
                System.out.println(7);
                break;
            case THURSDAY:
            case SATURDAY:
                System.out.println(8);
                break;
            case WEDNESDAY:
                System.out.println(9);
                break;
        default:
            throw new IllegalStateException("What day is today?" + day);
            }
    }
}

enum Week {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

Java 12：

```
/**
 * @author shkstart
 * @create 2019 下午 11:06
 */
public class SwitchTest1 {

    public static void main(String[] args) {
        Week day = Week.FRIDAY;
        switch (day) {
        case MONDAY,FRIDAY, SUNDAY -> System.out.println(6);

        case TUESDAY -> System.out.println(7);
        case THURSDAY, SATURDAY -> System.out.println(8);
        case WEDNESDAY -> System.out.println(9);
        default -> throw new IllegalStateException("What day is tod
ay?" + day);
        }
    }
}
```

Java 12 中更进一步：

```
/**
 * @author shkstart
 * @create 2019 下午 11:06
 */
public class SwitchTest2 {
    public static void main(String[] args) {
        Week day = Week.FRIDAY;
```

```
        int numLetters = switch (day) {
            case MONDAY, FRIDAY, SUNDAY -> 6;
            case TUESDAY -> 7;
            case THURSDAY, SATURDAY -> 8;
            case WEDNESDAY -> 9;
            default -> throw new IllegalStateException("What day is tod
ay?" + day);
        };
    }
}
```

5.使用总结

这个语法如果做过 Android 开发的不会陌生，因为 Kotlin 家的 when 表达式就是这么干的！

Switch Expressions 或者说起相关的 Pattern Matching 特性，为我们勾勒出了 Java 语法进化的一个趋势，将开发者从复杂繁琐的低层次抽象中逐渐解放出来，以更高层次更优雅的抽象，既降低代码量，又避免意外编程错误的出现，进而提高代码质量和开发效率。

6.展望

Java 11 以及之前版本中，Switch 表达式支持下面类型： byte、char、short、int、Byte、Character、Short、Integer、enum、String，在未来的某个 Java 版本有可能会允许支持 float、double 和 long （以及对应类型的包装类型）。

JDK13 中二次预览特性：

在 JDK 12 中引入了 Switch 表达式作为预览特性。JDK 13 提出了第二个 switch 表达式预览。JEP 354 修改了这个特性，它引入了 yield 语句，用于返回值。这意味着，switch 表达式(返回值)应该使用 yield，switch 语句(不返回值)应该使用 break。

在 JDK 12 中有一个，但是要进行一个更改：要从 switch 表达式中生成一个值 break，要删除 with value 语句以支持 a yield 声明。目的是扩展，switch 以便它可以用作语句或表达式，因此两个表单既可以使用 case … : 带有连贯符号的传统标签，也可以使用新 case … -> 标签，而不需要通过，还有一个新的语句用于从 switch 表达式中产生值。这些更改将简化编码并为模式匹配做好准备。

在以前，我们想要在 switch 中返回内容，还是比较麻烦的，一般语法如下：

```
@Test
public void testSwitch1(){
  String x = "3";
  int i;
  switch (x) {
      case "1":
```

```
                i=1;
                break;
            case "2":
                i=2;
                break;
            default:
                i = x.length();
                break;
        }
        System.out.println(i);
    }
```

在 JDK13 中使用以下语法：

```
    @Test
    public void testSwitch2(){
        String x = "3";
        int i = switch (x) {
            case "1" -> 1;
            case "2" -> 2;
            default -> {
                yield 3;
            }
        };
        System.out.println(i);
    }
```

或者

```
    @Test
    public void testSwitch3() {
        String x = "3";
        int i = switch (x) {
            case "1":
                yield 1;
            case "2":
                yield 2;
            default:
                yield 3;
        };
        System.out.println(i);
    }
```

在这之后，switch 中就多了一个关键字用于跳出 switch 块了，那就是 yield，他用于返回一个值。和 return 的区别在于：return 会直接跳出当前循环或者方法，而 yield 只会跳出当前 switch 块。

JDK14 中转正特性：

这是 JDK 12 和 JDK 13 中的预览特性，现在是正式特性了。

尚硅谷

该特性规定，switch 可以当作语句使用，也可以当作表达式使用。

这可以简化日常的编码方式，也为本版本中预览的模式匹配（JEP 305）特性打下了基础。

具体情况：使用->来替代以前的:+break；另外还提供了 yield 来在 block 中返回值

```
/**
 *  switch 的新特性
 *
 * @author shkstart Email:shkstart@126.com
 * @create 下午 12:10
 */
public class Feature04 {
    //jdk12 之前的用法
    @Test
    public void test1() {

        Week day = Week.FRIDAY;
        switch (day) {
            case MONDAY:
            case TUESDAY:
            case WEDNESDAY:
                System.out.println(1);
                break;
            case THURSDAY:
                System.out.println(2);
                break;
            case FRIDAY:
            case SATURDAY:
                System.out.println(3);
                break;
            case SUNDAY:
                System.out.println(4);
                break;
            default:
                throw new IllegalStateException("What day is today?" +
day);
        }

    }

    //jdk12 新特性：引用 switch 表达式
    @Test
    public void test2(){

        Week day = Week.FRIDAY;
        switch (day){
            case MONDAY,TUESDAY,WEDNESDAY -> System.out.println(1);
```

```
            case THURSDAY -> System.out.println(2);
            case FRIDAY,SATURDAY -> System.out.println(3);
            case SUNDAY -> System.out.println(4);
            default -> throw new IllegalStateException("What day is tod
ay?" + day);
        }

        //使用变量接收switch 表达式的值
        int num = switch (day) {
            case MONDAY, TUESDAY, WEDNESDAY -> 1;
            case THURSDAY -> 2;
            case FRIDAY, SATURDAY -> 3;
            case SUNDAY -> 4;
            default -> throw new IllegalStateException("What day is tod
ay?" + day);
        };

        System.out.println(num);

    }

    //jdk13 新特性：引用了yield 关键字，用于返回指定的数据，结束switch 结构
    @Test
    public void test3(){
        String x = "3";
        int num = switch (x){
            case "1" -> 1;
            case "2" -> 2;
            case "3" -> 3;
            default -> {
                System.out.println("default...");
                yield 4;
            }
        };

        System.out.println(x);
    }

    @Test
    public void test4(){
        String x = "3";
        int num = switch (x){
            case "1":yield 1;
            case "2":yield 2;
            case "3":yield 3;
            default: yield 4;
        };
        System.out.println(num);
    }
```

```
}

enum Week {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

18. switch 的模式匹配

JDK17 的预览特性：

老代码写法：

```
static String formatter(Object o) {
    String formatted = "unknown";
    if (o instanceof Integer i) {
        formatted = String.format("int %d", i);
    } else if (o instanceof Long l) {
        formatted = String.format("long %d", l);
    } else if (o instanceof Double d) {
        formatted = String.format("double %f", d);
    } else if (o instanceof String s) {
        formatted = String.format("String %s", s);
    }
    return formatted;
}
```

模式匹配新写法：

```
static String formatterPatternSwitch(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> o.toString();
    };
}
```

直接在 switch 上支持 Object 类型，这就等于同时支持多种类型，使用模式匹配得到具体类型，大大简化了语法量，这个功能还是挺实用的。

6.5 文本块

JDK13 中：

现实问题

在 Java 中，通常需要使用 String 类型表达 HTML，XML，SQL 或 JSON 等格式的字符串，在进行字符串赋值时需要进行转义和连接操作，然后才能编译该代码，这种表达方式难以阅读并且难以维护。

文本块就是指多行字符串，例如一段格式化后的 xml、json 等。而有了文本块以后，用户不需要转义，Java 能自动搞定。因此，文本块将提高 Java 程序的可读性和可写性。

目标

- 简化跨越多行的字符串，避免对换行等特殊字符进行转义，简化编写 Java 程序。
- 增强 Java 程序中字符串的可读性。

举例

会被自动转义，如有一段以下字符串：

```
<html>
  <body>
      <p>Hello, 尚硅谷</p>
  </body>
</html>
```

将其复制到 Java 的字符串中，会展示成以下内容：

```
"<html>\n" +
"    <body>\n" +
"        <p>Hello, 尚硅谷</p>\n" +
"    </body>\n" +
"</html>\n";
```

即被自动进行了转义，这样的字符串看起来不是很直观，在 JDK 13 中，就可以使用以下语法了：

```
"""
<html>
  <body>
      <p>Hello, world</p>
  </body>
</html>
""";
```

使用“”“作为文本块的开始符和结束符，在其中就可以放置多行的字符串，不需要进行任何转义。看起来就十分清爽了。

如常见的 SQL 语句：

```sql
select employee_id,last_name,salary,department_id
from employees
where department_id in (40,50,60)
order by department_id asc
```

原有方式：

```
String query = "select employee_id,last_name,salary,department_id\n" +
                "from employees\n" +
                "where department_id in (40,50,60)\n" +
                "order by department_id asc";
```

使用新特性：

```
String newQuery = """
        select employee_id,last_name,salary,department_id
        from employees
        where department_id in (40,50,60)
        order by department_id asc
""";
```

具体使用

1.基本使用

- 文本块是 Java 语言中的一种新文字。它可以用来表示任何字符串，并且提供更大的表现力和更少的复杂性。
- 文本块由零个或多个字符组成，由开始和结束分隔符括起来。
  - 开始分隔符是由三个双引号字符（"""），后面可以跟零个或多个空格，最终以行终止符结束。文本块内容以开始分隔符的行终止符后的第一个字符开始。
  - 结束分隔符也是由三个双引号字符（"""）表示，文本块内容以结束分隔符的第一个双引号之前的最后一个字符结束。
- 文本块中的内容可以直接使用"，"，但不是必需的。
- 文本块中的内容可以直接包括行终止符。允许在文本块中使用 \n，但不是必需的。例如，文本块：

```
"""
line1
line2
line3
"""
```

相当于：

```
"line1\nline2\nline3\n"
```

或者一个连接的字符串：

```
"line1\n" +
"line2\n" +
"line3\n"
```

如果字符串末尾不需要行终止符，则结束分隔符可以放在最后一行内容上。例如：

```
"""
line1
line2
line3"""
```

相当于

```
"line1\nline2\nline3"
```

文本块可以表示空字符串，但不建议这样做，因为它需要两行源代码：

```
String empty = """
""";
```

以下示例是错误格式的文本块：

```
String a = """""";   // 开始分隔符后没有行终止符
String b = """ """;  // 开始分隔符后没有行终止符
String c = """
           ";        // 没有结束分隔符
String d = """
           abc \ def
           """;      // 含有未转义的反斜线（请参阅下面的转义处理）
```

在运行时，文本块将被实例化为 String 的实例，就像字符串一样。从文本块派生的 String 实例与从字符串派生的实例是无法区分的。具有相同内容的两个文本块将引用相同的 String 实例，就像字符串一样。

此外，

1. 编译器在编译时会删除掉这些多余的空格。

下面这段代码中，我们用.来表示我们代码中的的空格，而这些位置的空格就是多余的。

```
String html = """
..............<html>
..............    <body>
..............        <p>Hello, world</p>
```

```
..............    </body>
..............</html>
..............""";
```

多余的空格还会出现在每一行的结尾，特别是当你从其他地方复制过来时，更容易出现这种情况，比如下面的代码：

```
String html = """
..............<html>...
..............    <body>
..............        <p>Hello, world</p>....
..............    </body>.
..............</html>...
..............""";
```

这些多余的空格对于程序员来说是看不到的，但是他又是实际存在的，所以如果编译器不做处理，可能会导致程序员看到的两个文本块内容是一样的，但是这两个文本块却因为存在这种多余的空格而导致差异，比如哈希值不相等。

1. 转义字符

允许开发人员使用 \n，\f 和\r 来进行字符串的垂直格式化，使用 \b 和 \t 进行水平格式化。比如下面的代码是合法的：

```
String html = """
              <html>\n
                  <body>\n
                      <p>Hello, world</p>\n
                  </body>\n
              </html>\n
              """;
```

请注意，在文本块内自由使用"是合法的。例如:

```
String story = """
    "When I use a word," Humpty Dumpty said,
    in rather a scornful tone, "it means just what I
    choose it to mean - neither more nor less."
    "The question is," said Alice, "whether you
    can make words mean so many different things."
    "The question is," said Humpty Dumpty,
    "which is to be master - that's all."
""";
```

但是，三个"字符的序列需要进行转义至少一个"以避免模仿结束分隔符：

```
String code =
    """
    String text = \"""
        A text block inside a text block
```

```
\""";
""";
```

1. 文本块连接

可以在任何可以使用字符串的地方使用文本块。例如，文本块和字符串可以相互连接：

```
String code = "public void print(Object o) {" +
              """
                  System.out.println(Objects.toString(o));
              }
              """;
```

但是，涉及文本块的连接可能变得相当笨重。以下面文本块为基础：

```
String code = """
              public void print(Object o) {
                  System.out.println(Objects.toString(o));
              }
              """;
```

假设我们想把上面的 Object 改为来自某一变量，我们可能会这么写：

```
String code = """
              public void print(""" + type + """
               o) {
                  System.out.println(Objects.toString(o));
              }
              """;
```

可以发现这种写法可读性是非常差的，更简洁的替代方法是使用 String :: replace 或 String :: format，比如：

```
String code = """
              public void print($type o) {
                  System.out.println(Objects.toString(o));
              }
              """.replace("$type", type);

String code = String.format("""
              public void print(%s o) {
                  System.out.println(Objects.toString(o));
              }
              """, type);
```

另一个方法是使用 String :: formatted，这是一个新方法，比如：

```
String source = """
                public void print(%s object) {
                    System.out.println(Objects.toString(object));
```

```
                }
                """.formatted(type);
```

JDK14 中：

JDK13 引入的 text blocks 进行第二轮 preview，JDK14 的版本主要增加了两个 escape sequences，分别是\ <line-terminator>与\s escape sequence

现实问题：

- 在 Java 中，通常需要使用 String 类型表达 HTML，XML，SQL 或 JSON 等格式的字符串，在进行字符串赋值时需要进行转义和连接操作，然后才能编译该代码，这种表达方式难以阅读并且难以维护。

目标

- 简化跨越多行的字符串，避免对换行等特殊字符进行转义，简化编写 Java 程序。
- 增强 Java 程序中用字符串表示的其他语言的代码的可读性
- 解析新的转义序列

举例：

```java
/**
 * 文本块(text blocks)的使用
 *
 * @author shkstart Email:shkstart@126.com
 * @create 下午 7:13
 */
public class Feature05 {


    @Test
    public void test1(){
        String text1 = "The Sound of silence\n" +
                "Hello darkness, my old friend\n" +
                "I've come to talk with you again\n" +
                "Because a vision softly creeping\n" +
                "Left its seeds while I was sleeping\n" +
                "And the vision that was planted in my brain\n" +
                "Still remains\n" +
                "Within the sound of silence";

        System.out.println(text1);

        //jdk13 中的新特性:
        String text2 = """
```

```
            The Sound of silence
            Hello darkness, my old friend
            I've come to talk with you again
            Because a vision softly creeping
            Left its seeds while I was sleeping
            And the vision that was planted in my brain
            Still remains
            Within the sound of silence
            """;
    System.out.println();
    System.out.println(text2);

    System.out.println(text1.length());
    System.out.println(text2.length());
}

//html
@Test
public void test2(){
    String html1 = "<html lang=\"en\">\n" +
            "<head>\n" +
            "    <meta charset=\"UTF-8\">\n" +
            "    <title>java14 新特性</title>\n" +
            "</head>\n" +
            "<body>\n" +
            "    <p>hello,atguigu</p>\n" +
            "</body>\n" +
            "</html>";
    //jdk13 中的新特性:
    String html2 = """
            <html lang="en">
            <head>
                <meta charset="UTF-8">
                <title>java14 新特性</title>
            </head>
            <body>
                <p>hello,atguigu</p>
            </body>
            </html>
            """;
}

//json
@Test
public void test3() {
    //jdk13 之前的写法
    String myJson = "{\n" +
            "    \"name\":\"Song Hongkang\",\n" +
            "     \"address\":\"www.atguigu.com\",\n" +
```

```
            "    \"email\":\"shkstart@126.com\"\n" +
            "}";
    System.out.println(myJson);

    //jdk13 的新特性
    String myJson1 = """
            {
                "name":"Song Hongkang",
                 "address":"www.atguigu.com",
                "email":"shkstart@126.com"
            }""";
    System.out.println(myJson1);
}

//sql
@Test
public void test4(){
    String sql = "SELECT id,NAME,email\n" +
            "FROM customers\n" +
            "WHERE id > 4\n" +
            "ORDER BY email DESC";

    //jdk13 新特性:
    String sql1 = """
            SELECT id,NAME,email
            FROM customers
            WHERE id > 4
            ORDER BY email DESC
            """;



}
//jdk14 新特性
@Test
public void test5(){
    String sql1 = """
            SELECT id,NAME,email
            FROM customers
            WHERE id > 4
            ORDER BY email DESC
            """;
    System.out.println(sql1);

    // \:取消换行操作
    // \s:表示一个空格
    String sql2 = """
            SELECT id,NAME,email \
            FROM customers\s\
```

```
                WHERE id > 4 \
                ORDER BY email DESC
                """;
        System.out.println(sql2);
    }
}
```

JDK15 中：文本块功能转正

Text Blocks 首次是在 JDK 13 中以预览功能出现的，然后在 JDK 14 中又预览了一次，终于在 JDK 15 中被确定下来，可放心使用了。

文本块是一种多行字符串文字，它避免了大多数转义序列的需要，以一种可预测的方式自动设置字符串的格式，并在需要时使开发人员可以控制格式，简化编写 Java 程序的任务。

文本块建议的目标是提高 Java 程序中的字符串的可读性，这些字符串表示以非 Java 语言编写的代码。 另一个目标是支持从字符串文本迁移，规定任何新构造都可以表达与字符串文本相同的字符串集，解释相同的转义序列，并且以与字符串文本相同的方式进行操作。OpenJDK 开发人员希望添加转义序列来管理显式空白和换行控件。

6.6 instanceof 的模式匹配

JDK14 中预览特性：

这个特性很有意思，因为它为更为通用的模式匹配打开了大门。模式匹配通过更为简便的语法基于一定的条件来抽取对象的组件，而 instanceof 刚好是这种情况，它先检查对象类型，然后再调用对象的方法或访问对象的字段。

有了该功能，可以减少 Java 程序中显式强制转换的数量，从而提高生产力，还能实现更精确、简洁的类型安全的代码。

Java 14 之前：

```
if (obj instanceof String) {
    String str = (String) obj; // 需要强转
    .. str.contains(..) ..
}else{
     str = ....
}
```

Java 14 新特性：

```
if (!(obj instanceof String str)) {
    .. str.contains(..) .. // 不再需要转换代码，实际发生了转换
} else {
    .. str....
}
```

举例：

```
/**
 * 1. JEP 305: instanceof 的模式匹配（预览）
 *
 * @author shkstart Email:shkstart@126.com
 * @create 上午 11:32
 */
public class Feature01 {
    @Test
    public void test1(){

        Object obj = new String("hello,Java14");
        obj = null;//在使用null 匹配instanceof 时，返回都是false.
        if(obj instanceof String){
            String str = (String) obj;
            System.out.println(str.contains("Java"));
        }else{
            System.out.println("非 String 类型");
        }

        //举例1:
        if(obj instanceof String str){ //新特性：省去了强制类型转换的过程
            System.out.println(str.contains("Java"));
        }else{
            System.out.println("非 String 类型");
        }
    }
}

// 举例2
class InstanceOf{

    String str = "abc";

    public void test(Object obj){
```

```
        if(obj instanceof String str){//此时的 str 的作用域仅限于 if 结构内。
            System.out.println(str.toUpperCase());
        }else{
            System.out.println(str.toLowerCase());
        }

    }

}

//举例3:
class Monitor{
    private String model;
    private double price;

//    public boolean equals(Object o){
//        if(o instanceof Monitor other){
//            if(model.equals(other.model) && price == other.price){
//                return true;
//            }
//        }
//        return false;
//    }


    public boolean equals(Object o){
        return o instanceof Monitor other && model.equals(other.model)
&& price == other.price;
    }

}
```

JDK15 中第二次预览：

在 Java 14 中作为预览语言功能引入的 instanceof 模式匹配，在 Java 15 中处于第二次预览，而没有任何更改。

模式匹配允许程序中的通用逻辑（主要是从对象中的条件提取组件）可以更简洁地表达。Haskell 和 C# 等语言已采用模式匹配来实现简洁和安全性。

旧写法：

```
// 先判断类型
if (obj instanceof String) {
    // 然后转换
    String s = (String) obj;
    // 然后才能使用
}
```

新写法：（自动匹配模式）

```
if (obj instanceof String s) {
    // 如果类型匹配 直接使用
} else {
    // 如果类型不匹配则不能直接使用
}
```

JDK16 中转正特性：

java 的 Instanceof 模式匹配最初是在 java14 中引入的，在 15 中得到增强，现在在 java16 中转正。

6.7 Record

JDK14 中预览特性：

背景

早在 2019 年 2 月份，Java 语言架构师 Brian Goetz，曾经写过一篇文章，详尽的说明了并吐槽了 Java 语言，他和很多程序员一样抱怨"Java 太啰嗦"或有太多的"繁文缛节"，他提到：开发人员想要创建纯数据载体类（plain data carriers）通常都必须编写大量低价值、重复的、容易出错的代码。如：构造函数、getter/setter、equals()、hashCode()以及 toString()等。

以至于很多人选择使用 IDE 的功能来自动生成这些代码。还有一些开发会选择使用一些第三方类库，如 Lombok 等来生成这些方法，从而会导致了令人吃惊的表现（surprising behavior）和糟糕的可调试性（poor debuggability）。

神说要用 record，于是就有了

我们有时候需要编写许多低价值的重复代码来实现一个简单的数据载体类：构造函数，访问器，equals()，hashCode () ，toString ()等。为了避免这种重复代码，Java 14 推出 record。

Java14 也许最令人兴奋，同时也是最令人惊讶的创新就是：Record 类型的引入 。

使用 record 来减少类声明语法，效果类似 lombok 的 @Data 注解，Kotlin 中的 data class。它们的共同点是类的部分或全部状态可以直接在类头中描述，并且这个类中只包含了纯数据而已。

该预览特性提供了一种更为紧凑的语法来声明类。值得一提的是，该特性可以大幅减少定义类似数据类型时所需的样板代码。

- 当你用 **record** 声明一个类时，该类将自动拥有以下功能：
  - 获取成员变量的简单方法，以上面代码为例 name() 和 partner() 。注意区别于我们平常 getter 的写法。

  - 一个 equals 方法的实现，执行比较时会比较该类的所有成员属性
  - 重写 equals 当然要重写 hashCode
  - 一个可以打印该类所有成员属性的 toString 方法。
  - 请注意只会有一个构造方法。
- 和枚举类型一样，记录也是类的一种受限形式。作为回报，记录对象在简洁性方面提供了显著的好处。
  - Enum interface
- 还可以在 Record 声明的类中定义静态字段、静态方法、构造器或实例方法。
- 不能在 Record 声明的类中定义实例字段；类不能声明为 abstract；不能声明显式的父类等。
- 为了在 Java 14 中引入这种新类型，需要在 Java.lang.Class 对象中添加如下两个新方法：
  - RecordComponent[] getRecordComponents()
  - boolean isRecord()

举例：

```
/**
 *  Record 类型的演示
 *
 * @author shkstart Email:shkstart@126.com
 * @create 下午 6:13
 */
public class Feature03 {
    @Test
    public void test1(){
        //测试构造器
        Person p1 = new Person("罗密欧",new Person("zhuliye",null));
        //测试toString()
        System.out.println(p1);
        //测试equals():
        Person p2 = new Person("罗密欧",new Person("zhuliye",null));
        System.out.println(p1.equals(p2));

        //测试hashCode()和equals()
        HashSet<Person> set = new HashSet<>();
        set.add(p1);
        set.add(p2);
```

```java
        for (Person person : set) {
            System.out.println(person);
        }

        //测试name()和partner():类似于getName()和getPartner()
        System.out.println(p1.name());
        System.out.println(p1.partner());

    }

    @Test
    public void test2(){
        Person p1 = new Person("zhuyingtai");

        System.out.println(p1.getNameInUpperCase());

        Person.nation = "CHN";
        System.out.println(Person.showNation());

    }
}

/**
 * @author shkstart Email:shkstart@126.com
 * @create 下午 6:20
 */
public record Person(String name,Person partner) {

    //还可以声明静态的属性、静态的方法、构造器、实例方法

    public static String nation;

    public static String showNation(){
        return nation;
    }

    public Person(String name){
        this(name,null);
    }

    public String getNameInUpperCase(){
        return name.toUpperCase();
    }
    //不可以声明非静态的属性
//    private int id;//报错
}

//不可以将record 定义的类声明为abstract 的
```

```
//abstract record Order(){
//
//}

//不可以给record 定义的类声明显式的父类（非Record 类）
//record Order() extends Thread{
//
//}
```

JDK15 中第二次预览特性：

Records Class 也是第二次出现的预览功能，它在 JDK 14 中也出现过一次了，使用 Record 可以更方便的创建一个常量类，使用的前后代码对比如下。

当你用 Record 声明一个类时，该类将自动拥有以下功能：

- 获取成员变量的简单方法，以上面代码为例 name() 和 partner() 。注意区别于我们平常 getter 的写法。
- 一个 equals 方法的实现，执行比较时会比较该类的所有成员属性
- 重写 equals 当然要重写 hashCode
- 一个可以打印该类所有成员属性的 toString 方法。
- 请注意只会有一个构造方法。

旧写法：

```
class Point {
    private final int x;
    private final int y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    int x() { return x; }
    int y() { return y; }
    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point other = (Point) o;        return other.x == x && other.y
= y;
    }    public int hashCode() {
        return Objects.hash(x, y);
    }    public String toString() {
        return String.format("Point[x=%d, y=%d]", x, y);
    }
}
```

新写法：

```
record Point(int x, int y) { }
```

JDK16 中转正特性：

此特性在 JDK14 中作为预览版本引入，最终到 JDK16 中转正。

record 是一种全新的类型，它本质上是一个 final 类，同时所有的属性都是 final 修饰，它会自动编译出 public get hashcode 、equals、toString 等方法，减少了代码编写量。

```
public record Dog(String name, Integer age) {
}

public class Java14Record {

    public static void main(String[] args) {
        Dog dog1 = new Dog("牧羊犬", 1);
        Dog dog2 = new Dog("田园犬", 2);
        Dog dog3 = new Dog("哈士奇", 3);
        System.out.println(dog1);
        System.out.println(dog2);
        System.out.println(dog3);
    }
}
```

怎样看待记录

你应该如何看待和处理这些新的语言元素呢？记录是一种新的、受限形式的类，用于将数据建模为数据。我们不可能向记录添加任何附加状态；除了记录的组件之外，你不能定义（非静态）字段。记录实际上是建模不可变数据的。你也可以将记录视为元组，但它并不只是其他一些语言所有的那种一般意义上的元组，在那种元组里有一些可以由索引引用的任意组件。在 Java 中，元组元素有实际名称，并且元组类型本身——即记录，也有一个名称，因为名称在 Java 中很重要。

记录不适合哪些场景

有些场景中我们可能会觉得记录用起来并不是很合适。首先，它们并不是任何现有代码的一个样板缩减机制。虽然我们现在有一种非常简洁的方式来定义这些记录，但这并不意味着你的应用程序中的任何数据（如类）都可以轻松地被记录替换，这主要是因为记录存在的一些限制所致。这也不是它真正的设计目标。

记录的设计目标是提供一种将数据建模为数据的好方法。它也不是 JavaBeans 的直接替代品，因为正如我之前提到的，访问器这样的方法不符合 JavaBeans 的 get 标准。另外 JavaBeans 通常是可变的，而记录是不可变的。尽管它们的用途有点像，但记录并不会以某种方式取代 JavaBean。你也不应该将记录视为值类型。

值类型可能会在未来的 Java 版本中作为语言增强引入，其主要关注内存布局和类中数据的有效表示。当然，这两条世界线在未来某一时刻可能会合并在一起，但就目前而言，记录只是表达纯数据类的一种更简洁的方式。

其他内容见：https://www.cnblogs.com/chengxuyuan-blog/p/15232243.html

6.8 密封类

JDK15 的预览特性：

通过密封的类和接口来增强 Java 编程语言，这是新的预览特性。 用于限制超类的使用，密封的类和接口限制其它可能继承或实现它们的其它类或接口。

这个特性的目标包括——允许类或接口的开发者来控制哪些代码负责实现，提供了比限制使用超类的访问修饰符声明方式更多选择,并通过支持对模式的详尽分析而支持模式匹配的未来发展。

在 Java 中，类层次结构通过继承实现代码的重用，父类的方法可以被许多子类继承。 但是，类层次结构的目的并不总是重用代码。有时，其目的是对域中存在的各种可能性进行建模，例如图形库支持的形状类型或金融应用程序支持的贷款类型。当以这种方式使用类层次结构时，我们可能需要限制子类集从而来简化建模。

具体使用： 因为我们引入了 sealed class 或 interfaces，这些 class 或者 interfaces 只允许被指定的类或者 interface 进行扩展和实现。

使用修饰符 sealed，您可以将一个类声明为密封类。密封的类使用 reserved 关键字 permits 列出可以直接扩展它的类。子类可以是最终的，非密封的或密封的。

示例：

```
package com.atguigu.java;

public abstract sealed class Shape permits Circle, Rectangle, Square
{...}
public final class Circle extends Shape {...}
public sealed class Rectangle extends Shape permits TransparentRectangl
e, FilledRectangle {...}
public final class TransparentRectangle extends Rectangle {...}
public final class FilledRectangle extends Rectangle {...}
public non-sealed class Square extends Shape {...}
```

JDK16 二次预览特性：

通过密封类和接口，Java 编程语言得到了增强。它限制哪些类或接口可以限制或实现它们。

Sealed Classes 是在 JDK15 中引入的概念，它表示某个类允许哪些类来继承它：

```
public sealed class SealExample permits Seal1, Seal2{}

public non-sealed class Seal1 extends SealExample {}

public final class Seal2 extends SealExample {}
```

final 表示 Seal2 不能再被继承了。non-sealed 表示可以允许任何类继承。

#########

下面是一段引用：

我们都知道，在 Java 中如果想让一个类不能被继承和修改，这时我们应该使用 `final` 关键字对类进行修饰。不过这种要么可以继承，要么不能继承的机制不够灵活，有些时候我们可能想让某个类可以被某些类型继承，但是又不能随意继承，是做不到的。Java 15 尝试解决这个问题，引入了 `sealed` 类，被 `sealed` 修饰的类可以指定子类。这样这个类就只能被指定的类继承。

而且 `sealed` 修饰的类的机制具有传递性，它的子类必须使用指定的关键字进行修饰，且只能是 `final`、`sealed`、`non-sealed` 三者之一。

#############

下面来谈谈密封类（sealed class）这个特性。请注意，这是 Java 16 中的预览特性，将在 Java 17 中成为最终版本。你需要将--enable-preview 标志传递给编译器调用和 JVM 调用才能在 Java 16 中使用这个特性。该特性允许你控制继承层次结构。

假设你想对一个超类型 Option 建模，其中你只想有 Some 和 Empty 两个子类型。并且你想预防 Option 类型获得任何扩展。例如，你不想在层次结构中允许 Maybe 类型。

因此，你已经详细描述了 Option 类型的所有子类型。如你所知，目前在 Java 中控制继承的唯一工具是通过 final 关键字。这意味着根本不能有任何子类，但这不是我们想要的。有一些解决方法可以在没有密封类的情况下建模这个特性，但有了密封类后就容易多了。

密封类特性带有新的关键字 sealed 和 permits。看看下面的代码段。

```
public sealed class Option<T>
 permits Some, Empty {
 ...
}
public final class Some
 extends Option<String> {
 ...
}
public final class Empty
 extends Option<Void> {
 ...
}
```

我们可以定义要 sealed 的 Option 类。然后，在类声明之后，我们使用 permit 关键字来规定只允许 Some 和 Empty 类扩展 Option 类。然后，我们可以像往常一样将 Some 和 Empty 定义为类。我们希望将这些子类设为 final，以防止进一步继承。现在系统就不能编译其他类来扩展 Option 类了，这是由编译器通过密封类机制强制执行的。

JDK17 转正特性：

密封类（Sealed Classes）是 Java 15 引入的， Java 17 正式转正。

密封类可以对继承或者实现它们的类进行限制。

密封类可以用来增强 Java 编程语言，防止其他类或接口扩展或实现它们。

比如抽象类 `Person` 只允许 `Employee` 和 `Manager` 继承。

```
public abstract sealed class Person
    permits Employee, Manager {

    //...
}
```

另外，任何扩展密封类的类本身都必须声明为 `sealed`、`non-sealed` 或 `final`。

```
public final class Employee extends Person {
}

public non-sealed class Manager extends Person {
}
```

再举例：

```
public abstract sealed class Student
    permits ZhangSan, LiSi, ZhaoLiu {
    ...

}
```

类 Student 被 sealed 修饰，说明它是一个密封类，并且只允许指定的 3 个子类继承。

再举例：

```
package com.example.geometry;

public abstract sealed class Shape
    permits Circle, Rectangle, Square, WeirdShape { ... }

public final class Circle extends Shape { ... }

public sealed class Rectangle extends Shape
    permits TransparentRectangle, FilledRectangle { ... }
public final class TransparentRectangle extends Rectangle { ... }
public final class FilledRectangle extends Rectangle { ... }

public final class Square extends Shape { ... }

public non-sealed class WeirdShape extends Shape { ... }
```

6.9 隐藏类

JDK15 新特性：

该提案通过启用标准 API 来定义无法发现且具有有限生命周期的隐藏类，从而提高 JVM 上所有语言的效率。JDK 内部和外部的框架将能够动态生成类，而这些类可以定义隐藏类。通常来说基于 JVM 的很多语言都有动态生成类的机制，这样可以提高语言的灵活性和效率。隐藏类天生为框架设计的，在运行时生成内部的 class。隐藏类只能通过反射访问，不能直接被其他类的字节码访问。隐藏类可以独立于其他类加载、卸载，这可以减少框架的内存占用。

Hidden Classes 是什么呢？ Hidden Classes 就是不能直接被其他 class 的二进制代码使用的 class。Hidden Classes 主要被一些框架用来生成运行时类，但是这些类不是被用来直接使用的，而是通过反射机制来调用。比如在 JDK8 中引入的 lambda 表达式，JVM 并不会在编译的时候将 lambda 表达式转换成为专门的类，而是在运行时将相应的字节码动态生成相应的类对象。另外使用动态代理也可以为某些类生成新的动态类。

那么我们希望这些动态生成的类需要具有什么特性呢？ 不可发现性。因为我们是为某些静态的类动态生成的动态类，所以我们希望把这个动态生成的类看做是静态类的一部分。所以我们不希望除了该静态类之外的其他机制发现。 访问控制。我们希望在访问控制静态类的同时，也能控制到动态生成的类。 生命周期。动态生成类的生命周期一般都比较短，我们并不需要将其保存和静态类的生命周期一致。

API 的支持 所以我们需要一些 API 来定义无法发现的且具有有限生命周期的隐藏类。这将提高所有基于 JVM 的语言实现的效率。 比如： java.lang.reflect.Proxy 可以定义隐藏类作为实现代理接口的代理类。 java.lang.invoke.StringConcatFactory 可以生成隐藏类来保存常量连接方法； java.lang.invoke.LambdaMetaFactory 可以生成隐藏的 nestmate 类，以容纳访问封闭变量的 lambda 主体；

普通类是通过调用 ClassLoader::defineClass 创建的，而隐藏类是通过调用 Lookup::defineHiddenClass 创建的。这使 JVM 从提供的字节中派生一个隐藏类，链接该隐藏类，并返回提供对隐藏类的反射访问的查找对象。调用程序可以通过返回的查找对象来获取隐藏类的 Class 对象。

7. 新的 API

7.1 Optional 类

这是 JDK8 的新特性

概述

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 类(java.util.Optional) 是一个容器类，它可以保存类型 T 的值，代表这个值存在。或者仅仅保存 null，表示这个值不存在。原来用 null 表示一个值不存在，现在 Optional 可以更好的表达这个概念。并且可以避免空指针异常。

Optional 类的 Javadoc 描述如下：这是一个可以为 null 的容器对象。如果值存在则 isPresent()方法会返回 true，调用 get()方法会返回该对象。

方法

Optional 提供很多有用的方法，这样我们就不用显式进行空值检测。

- **创建 Optional 类对象的方法：**
  - static Optional empty() ：用来创建一个空的 Optional 实例
  - static Optional of(T value) ：用来创建一个 Optional 实例，value 必须非空
  - `static <T> Optional<T> ofNullable(T value)` ：用来创建一个 Optional 实例，value 可能是空，也可能非空
- **判断 Optional 容器中是否包含对象：**

- boolean isPresent() : 判断 Optional 容器中的值是否存在
- 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

- **获取 Optional 容器的对象：**
  - T get(): 如果调用对象包含值，返回该值。否则抛异常。T get()与 of(T value)配合使用
  - 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
  - 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
  - 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

举例

```
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.NoSuchElementExce
ption: 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);
    }
}
```

这是 JDK9-11 的新特性

新增方法	描述	新增的版本
boolean isEmpty()	判断 value 是否为空	JDK 11
ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)	value 非空，执行参数 1 功能；如果 value 为空，执行参数 2 功能	JDK 9
Optional or(Supplier<? extends Optional<? extends T>> supplier)	value 非空，返回对应的 Optional；value 为空，返回形参封装的 Optional	JDK 9
Stream stream()	value 非空，返回仅包含此 value 的 Stream；否则，返回一个空的 Stream	JDK 9
T orElseThrow()	value 非空，返回 value；否则抛异常 NoSuchElementException	JDK 10

7.2 String 存储结构和 API 变更

这是 JDK9 的新特性。

产生背景：

> Motivation The current implementation of the String class stores characters in a char array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most String objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal char arrays of such String objects is going unused.

使用说明：

Description

> We propose to change the internal representation of the String class from a UTF-16 char array to a byte array plus an encoding-flag field. The new String class will store characters encoded either as ISO-8859-1/Latin-1 (one byte per character), or as UTF-16 (two bytes per character), based upon the contents of the string. The encoding flag will indicate which encoding is used.

结论：String 再也不用 char[] 来存储啦，改成了 byte[] 加上编码标记，节约了一些空间。

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    @Stable
    private final byte[] value;
      ...
}
```

拓展：StringBuffer 与 StringBuilder

那 StringBuffer 和 StringBuilder 是否仍无动于衷呢？

> String-related classes such as AbstractStringBuilder, StringBuilder, and StringBuffer will be updated to use the same representation, as will the HotSpot VM's intrinsic string operations.

JDK11 新特性：新增了一系列字符串处理方法

描述	举例
判断字符串是否为空白	" ".isBlank(); // true
去除首尾空白	" Javastack ".strip(); // "Javastack"
去除尾部空格	" Javastack ".stripTrailing(); // " Javastack"
去除首部空格	" Javastack ".stripLeading(); // "Javastack "
复制字符串	"Java".repeat(3);// "JavaJavaJava"
行数统计	"A\nB\nC".lines().count(); // 3

JDK12 新特性：String 实现了 Constable 接口

String 源码：

```
public final class String implements java.io.Serializable, Comparable<S
tring>, CharSequence,Constable, ConstantDesc {
```

java.lang.constant.Constable 接口定义了抽象方法：

```java
public interface Constable {
      Optional<? extends ConstantDesc> describeConstable();
}
```

Java 12 String 的实现源码：

```java
@Override
public Optional<String> describeConstable() {
      return Optional.of(this);
}
```

很简单，其实就是调用 Optional.of 方法返回一个 Optional 类型，Optional 不懂的可以参考 Java 8 的新特性

6.3.2 String#describeConstable 和 resolveConstantDesc

一个非常有趣的方法来自新引入的接口 java.lang.constant.Constable - 它用于标记 constable 类型，这意味着这类型的值是常量，可以在 JVMS 4.4 常量池中定义。

```
Java SE > Java SE Specifications > Java Virtual Machine Specification 下的第 4 章：
Chapter 4. The class File Format

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html
```

String 的源码：

```java
/**
 * Returns an {@link Optional} containing the nominal descriptor for this
 * instance, which is the instance itself.
 *
 * @return an {@link Optional} describing the {@linkplain String} instance
 * @since 12
 */
@Override
public Optional<String> describeConstable() {
 return Optional.of(this);
}


/**
 * Resolves this instance as a {@link ConstantDesc}, the result of which is
 * the instance itself.
 *
 * @param lookup ignored
 * @return the {@linkplain String} instance
 * @since 12
```

```
 */
@Override
public String resolveConstantDesc(MethodHandles.Lookup lookup) {
 return this;
}
```

举例：

```
private static void testDescribeConstable() {
      System.out.println("======test java 12 describeConstable======");
      String name = "尚硅谷 Java 高级工程师";
      Optional<String> optional = name.describeConstable();
      System.out.println(optional.get());
}
```

结果输出：

```
======test java 12 describeConstable======
尚硅谷 Java 高级工程师
```

JDK12 新特性：String 新增方法

1. String#transform(Function)

JDK-8203442 引入的一个小方法，它提供的函数作为输入提供给特定的 String 实例，并返回该函数返回的输出。

```
var result = "foo".transform(input -> input + " bar");
System.out.println(result); // foo bar
```

或者

```
var result = "foo"
 .transform(input -> input + " bar")
 .transform(String::toUpperCase)
System.out.println(result); // FOO BAR
```

对应的源码：

```
/**
* This method allows the application of a function to {@code this}
* string. The function should expect a single String argument
* and produce an {@code R} result.
* <p>
* Any exception thrown by {@code f()} will be propagated to the
* caller.
*
* @param f    functional interface to a apply
*
* @param <R>  class of the result
*
```

```
* @return     the result of applying the function to this string
*
* @see java.util.function.Function
*
* @since 12
*/
public <R> R transform(Function<? super String, ? extends R> f) {
 return f.apply(this);
}
```

传入一个函数式接口 Function，接受一个值，返回一个值，参考：Java 8 新特性之函数式接口。

在某种情况下，该方法应该被称为 map()。

举例：

```
private static void testTransform() {
      System.out.println("======test java 12 transform======");
      List<String> list1 = List.of("Java", " Python", " C++ ");
      List<String> list2 = new ArrayList<>();
      list1.forEach(element -> list2.add(element.transform(String::stri
p)
                                                .transform(String::to
UpperCase)
                                               .transform((e) -> "Hi,
" + e))
                          );
      list2.forEach(System.out::println);
}
```

结果输出：

```
======test java 12 transform======
Hi,JAVA
Hi,PYTHON
Hi,C++
```

示例是对一个字符串连续转换了三遍，代码很简单。如果使用 Java 8 的 Stream 特性，可以如下实现：

```
private static void testTransform1() {
        System.out.println("======test before java 12 ======");
        List<String> list1 = List.of("Java  ", " Python", " C++ ");

        Stream<String> stringStream = list1.stream().map(element -> ele
ment.strip()).map(String::toUpperCase).map(element -> "Hello," + elemen
t);
        List<String> list2 = stringStream.collect(Collectors.toList());
```

```
        list2.forEach(System.out::println);
    }
```

1. String#indent

该方法允许我们调整 String 实例的缩进。

举例：

```
private static void testIndent() {
      System.out.println("======test java 12 indent======");
      String result = "Java\n Python\nC++".indent(3);
      System.out.println(result);
}
```

结果输出：

```
======test java 12 indent======
   Java
    Python
   C++
```

换行符 \n 后向前缩进 n 个空格，为 0 或负数不缩进。

以下是 indent 的核心源码：

```
      /**
     * Adjusts the indentation of each line of this string based on the
 value of
     * {@code n}, and normalizes line termination characters.
     * <p>
     * This string is conceptually separated into lines using
     * {@link String#lines()}. Each line is then adjusted as described
below
     * and then suffixed with a line feed {@code "\n"} (U+000A). The re
sulting
     * lines are then concatenated and returned.
       *  ...略...
     *
     * @since 12
     */
    public String indent(int n) {
        if (isEmpty()) {
            return "";
        }
        Stream<String> stream = lines();
        if (n > 0) {
            final String spaces = " ".repeat(n);
            stream = stream.map(s -> spaces + s);
        } else if (n == Integer.MIN_VALUE) {
            stream = stream.map(s -> s.stripLeading());
```

```
        } else if (n < 0) {
            stream = stream.map(s -> s.substring(Math.min(-n, s.indexOf
NonWhitespace())));
        }
        return stream.collect(Collectors.joining("\n", "", "\n"));
    }
```

其实就是调用了 lines() 方法来创建一个 Stream，然后再往前拼接指定数量的空格。

7.3 JDK11：InputStream 加强

InputStream 终于有了一个非常有用的方法：transferTo，可以用来将数据直接传输到 OutputStream，这是在处理原始数据流时非常常见的一种用法，如下示例。

```
var classLoader = ClassLoader.getSystemClassLoader();
var inputStream = classLoader.getResourceAsStream("javastack.txt");
var javastack = File.createTempFile("javastack2", "txt");

try (var outputStream = new FileOutputStream(javastack)) {
  inputStream.transferTo(outputStream);
}
```

7.4 JDK16：提供基于值的类的警告

这个是什么意思呢？我们知道 java 中对应的 primary 类型都有一个 Object 类型，比如 int 对应的是 Integer。

如果是用 Integer 的构造函数，则我们可以这样构造：

```
Integer integer= new Integer(100);
```

但是在 JDK16 中，这种构造函数已经被废弃了：

```
@Deprecated(since="9", forRemoval = true)
public Integer(int value) {
      this.value = value;
}
```

我们可以直接这样写：

```
Integer integer2= 100;
```

另外：

添加了一个注解，用于标识当前是是基于值的类，比如 Java 8 引入的预防空指针的 Optional 类，现在已经添加了注解标识。

```
@jdk.internal.ValueBased
public final class Optional<T> {
    // ...
}
```

再举例：

```
@jdk.internal.ValueBased
public final class SomeVbc {

    public SomeVbc() {}

    final String ref = "String";

    void abuseVbc() {

        synchronized(ref) {            // OK
            synchronized (this) {      // WARN
            }
        }
    }
}

final class AuxilliaryAbuseOfVbc {

    void abuseVbc(SomeVbc vbc) {

        synchronized(this) {           // OK
            synchronized (vbc) {       // WARN
            }
        }
    }
}
```

7.5 JDK16/17：Vector API

JDK16 的预览特性：

提供了 jdk.incubator.vector 来用于矢量计算。

Vector API 的想法是提供一种向量计算方法，最终实现比传统的标量计算方法（在支持 CPU 体系架构上）提供更高的性能。什么叫做向量计算呢？熟悉 pandas 的朋友可能知道，在 pandas 可以方便的对矩阵进行计算，如果用 java 实现则需要计算矩阵中的每个元素，非常麻烦，这也是 python 的 pandas 库能够流行的原因。

现在 JDK16 也可以做到了，我们一起来看看,先是传统写法：

```
//传统写法
int[] x = {1, 2, 3, 4};
int[] y = {4, 3, 2, 1};
int[] c = new int[x.length];

for (int i = 0; i < x.length; i++) {
```

```
    c[i] =x[i] * y[i];
}
```

如果我们希望两个数组的数字相乘，则只能进行每个元素的遍历。

现在的写法：

```
var vectorA = IntVector.fromArray(IntVector.SPECIES_128, x, 0);
var vectorB = IntVector.fromArray(IntVector.SPECIES_128, y, 0);
var vectorC = vectorA.mul(vectorB);
vectorC.intoArray(c, 0);
```

我们构建两个 Vector 变量，直接调用 Vector 类的 mul 方法即可。

fromArray 中有三个参数，第一个是向量的长度，第二是原数组，第三个是偏移量。因为一个 int 有 4 个字节，所以这里我们使用 SPECIES_128。

再举例：

```
static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_256;

void vectorComputation(float[] a, float[] b, float[] c) {

    for (int i = 0; i < a.length; i += SPECIES.length()) {
        var m = SPECIES.indexInRange(i, a.length);
        // FloatVector va, vb, vc;
        var va = FloatVector.fromArray(SPECIES, a, i, m);
        var vb = FloatVector.fromArray(SPECIES, b, i, m);
        var vc = va.mul(va).
                    add(vb.mul(vb)).
                    neg();
        vc.intoArray(c, i, m);
    }
}
```

JDK17 中二次预览特性：

Vector API 这是一个新的初始迭代孵化器模块，模块包：jdk.incubator.vector，用于表示在运行时可靠地编译到支持的 CPU 架构上的最佳矢量硬件指令的矢量计算，矢量运算可以提供优于等效标量计算的性能，并且在机器学习、人工智能和密码学等领域非常普遍。

本次增强的 API 允许以一种在运行时，可靠地编译为支持的 CPU 架构上的最佳向量指令的方式表达向量计算。

7.6 JDK17：标记删除 Applet API

Applet API 提供了一种将 Java AWT/Swing 控件嵌入到浏览器网页中的方法。不过，目前 Applet 已经被淘汰。大部分人应该和我一样，压根就没有用过 Applet。

Applet API 实际上是无用的，因为所有 Web 浏览器供应商都已删除或透露计划放弃对 Java 浏览器插件的支持。Java 9 的时候，Applet API 已经被标记为启用，Java 17 的时候终于标记为删除了。

具体如下：

```
java.applet.Applet
java.applet.AppletStub
java.applet.AppletContext
java.applet.AudioClip
javax.swing.JApplet
java.beans.AppletInitializer
```

8. 其它结构变化

8.1 JDK 目录结构变化

这是 JDK9 的新特性。

JDK 8 的目录结构

目录	说明
bin 目录	包含命令行开发和调试工具，如 javac，jar 和 javadoc。
include 目录	包含在编译本地代码时使用的 C/C++头文件
lib 目录	包含 JDK 工具的几个 JAR 和其他类型的文件。 它有一个 tools.jar 文件，其中包含 javac 编译器的 Java 类
jre/bin 目	包含基本命令，如 java 命令。 在 Windows 平台上，它包含系统的运

目录	说明
录	行时动态链接库（DLL）。
jre/lib 目录	包含用户可编辑的配置文件，如.properties 和.policy 文件。包含几个 JAR。 rt.jar 文件包含运行时的 Java 类和资源文件。

JDK 9 的目录结构

目录	说明
bin 目录	包含所有命令。 在 Windows 平台上，它继续包含系统的运行时动态链接库。
conf 目录	包含用户可编辑的配置文件，例如以前位于 jre\lib 目录中的.properties 和.policy 文件
include 目录	包含要在以前编译本地代码时使用的 C/C++头文件。 它只存在于 JDK 中
jmods 目录	包含 JMOD 格式的平台模块。 创建自定义运行时映像时需要它。 它只存在于 JDK 中
legal 目录	包含法律声明
lib 目录	包含非 Windows 平台上的动态链接本地库。 其子目录和文件不应由开发人员直接编辑或使用

没有名为 jre 的子目录

8.2 模块化系统: Jigsaw -> Modularity

这是 JDK9 的新特性。

- 谈到 Java 9 大家往往第一个想到的就是 Jigsaw 项目。众所周知，Java 已经发展超过 20 年（95 年最初发布），Java 和相关生态在不断丰富的同时也越来越暴露出一些问题：
  - Java 运行环境的膨胀和臃肿。每次 JVM 启动的时候，至少会有 30～60MB 的内存加载，主要原因是 JVM 需要加载 rt.jar，不管其中的类

是否被 classloader 加载，第一步整个 jar 都会被 JVM 加载到内存当中去（而模块化可以根据模块的需要加载程序运行需要的 class）

  - 当代码库越来越大，创建复杂，盘根错节的“意大利面条式代码”的几率呈指数级的增长。不同版本的类库交叉依赖导致让人头疼的问题，这些都阻碍了 Java 开发和运行效率的提升。
  - 很难真正地对代码进行封装, 而系统并没有对不同部分（也就是 JAR 文件）之间的依赖关系有个明确的概念。每一个公共类都可以被类路径之下任何其它的公共类所访问到，这样就会导致无意中使用了并不想被公开访问的 API。
  - 类路径本身也存在问题: 你怎么知晓所有需要的 JAR 都已经有了, 或者是不是会有重复的项呢?

- 同时，由于兼容性等各方面的掣肘，对 Java 进行大刀阔斧的革新越来越困难，Jigsaw 从 Java 7 阶段就开始筹备，Java 8 阶段进行了大量工作，终于在 Java 9 里落地，一种千呼万唤始出来的意味。
- Jigsaw 项目（后期更名为 Modularity）的工作量和难度大大超出了初始规划。JSR 376 Java 平台模块化系统（JPMS, Java Platform Module System）作为 Jigsaw 项目的核心, 其主体部分被分解成 6 个 JEP(JDK Enhancement Proposals)。
- 作为 java 9 平台最大的一个特性，随着 Java 平台模块化系统的落地，开发人员无需再为不断膨胀的 Java 平台苦恼，例如，您可以使用 jlink 工具，根据需要定制运行时环境。这对于拥有大量镜像的容器应用场景或复杂依赖关系的大型应用等，都具有非常重要的意义。
- 本质上讲，模块(module)的概念，其实就是 package 外再裹一层，也就是说，用模块来管理各个 package，通过声明某个 package 暴露，不声明默认就是隐藏。因此，模块化使得代码组织上更安全，因为它可以指定哪些部分可以暴露，哪些部分隐藏。

实现目标

- 主要目的在于减少内存的开销
- 只须必要模块，而非全部 jdk 模块，可简化各种类库和大型应用的开发和维护
- 改进 Java SE 平台，使其可以适应不同大小的计算设备
- 改进其安全性，可维护性，提高性能

举例：略

8.3 其它语法变化

8.3.1 JDK9 语法改进：UnderScore(下划线)使用的限制

在 java 8 中，标识符可以独立使用“_”来命名：

```
String _ = "hello";
System.out.println(_);
```

但是，在 java 9 中规定“_”不再可以单独命名标识符了，如果使用，会报错：

8.3.2 JDK9/17：AOT

JDK9 中：

产生背景

Oracle 一直在努力提高 Java 启动和运行时性能，希望其能够在更广泛的场景达到或接近本地语言的性能。但是，直到今天，谈到 Java，很多 C/C++ 开发者还是会不屑地评价为启动慢，吃内存。

简单说，这主要是因为 Java 编译产生的类文件是 Java 虚拟机可以理解的二进制代码，而不是真正的可执行的本地代码，需要 Java 虚拟机进行解释和编译，这带来了额外的开销。

使用说明

JIT（Just-in-time）编译器可以在运行时将热点编译成本地代码，速度很快。但是 Java 项目现在变得很大很复杂，因此 JIT 编译器需要花费较长时间才能热身完，而且有些 Java 方法还没法编译，性能方面也会下降。AoT 编译就是为了解决这些问题而生的。

在 JDK 9 中， AOT（JEP 295: Ahead-of-Time Compilation）作为实验特性被引入进来，开发者可以利用新的 jaotc 工具将重点代码转换成类似类库一样的文件。虽然仍处于试验阶段，但这个功能使得 Java 应用在被虚拟机启动之前能够先将 Java 类

编译为原生代码。此功能旨在改进小型和大型应用程序的启动时间，同时对峰值性能的影响很小。

但是 Java 技术供应商 Excelsior 的营销总监 Dmitry Leskov 担心 AoT 编译技术不够成熟，希望 Oracle 能够等到 Java 10 时有个更稳定版本才发布。

JDK17 中：

删除实验性的基于 Java 的提前 (AOT)，因为使用有限，维护它所需的工作量很重要。

作为实验性功能，JDK 9 已与提前编译（jaotc 工具）集成。对于 AOT 编译，jaotc 使用 Java 编写的 Graal 编译器。由于这些实验特性尚未使用，因此需要付出相当大的努力来维护和改进它们。Oracle 发布的 JDK 16 版本没有这些功能，也没有人抱怨。

8.3.3 JDK11：更简化的编译运行程序

看下面的代码。

```
// 编译
javac Javastack.java

// 运行
java Javastack
```

在我们的认知里面，要运行一个 Java 源代码必须先编译，再运行，两步执行动作。而在 Java 11 版本中，通过一个 java 命令就直接搞定了，如以下所示：

```
java Javastack.java
```

一个命令编译运行源代码的注意点：

- 执行源文件中的第一个类, 第一个类必须包含主方法。
- 并且不可以使用其它源文件中的自定义类, 本文件中的自定义类是可以使用的。

8.3.4 JDK14：有用的 NullPointerExceptions

Helpful NullPointerExceptions 可以在我们遇到 NPE 时节省一些分析时间。如下的代码会导致一个 NPE。

```
public static void main(String[] args) {
    Person p = new Person();
      String cityName = p.getAddress().getCity().getName();
}
```

在 Java 11 中，输出将显示 NullPointerException 发生的行号，但不知道哪个方法调用时产生的 null，必须通过调试的方式找到。

```
Exception in thread "main" java.lang.NullPointerException
        at com.atguigu.java.Demo.main(HelpfulNullPointerExceptionsDemo.
java:13)
```

在 Java 17 中，则会准确显示发生 NPE 的精确位置。

```
Exception in thread "main" java.lang.NullPointerException: Cannot invok
e "com.heiz.java17.Address.getCity()" because the return value of "com.
heiz.java17.Person.getAddress()" is null
            at com.atguigu.java.Demo.main(HelpfulNullPointerExceptionsD
emo.java:13)
```

8.3.5 JDK16：弹性元空间

Metaspace 的主要功能是管理类的元数据的内存。引入 Elastic Metaspace 是为了改进 HotSpot JVM 中元空间内存的分配和释放。可以更快地将不需要的内存返回给操作系统，从而减少开销和内存碎片。

Elastic Metaspace 使用较小的块分配内存，并通过将未使用的元空间内存返回给操作系统来提高弹性。它可以提高性能并降低维护成本。

8.3.6 JDK17：JDK 内部元素强封装

很多程序员以损害安全性和可维护性的方式使用 JDK 的内部元素比如一些非 `public` 类、方法和字段。

为了继续提高 JDK 的安全性和可维护性，默认强封装 JDK 的所有内部元素，从而限制对它们的访问，除了关键的内部 API 如 `sun.misc.Unsafe`。鼓励开发人员从内部元素迁移到标准 API。

此外，用户仍然可以选择自 JDK 9 以来的默认的宽松的强封装，这样可以帮助用户毫不费力地升级到未来的 Java 版本。

9. GC 方面新特性

GC 是 java 主要优势之一。 然而, 当 GC 停顿太长, 就会开始影响应用的响应时间。消除或者减少 GC 停顿时长, java 将对更广泛的应用场景是一个更有吸引力的平台。此外, 现代系统中可用内存不断增长,用户和程序员希望 JVM 能够以高效的方式充分利用这些内存, 并且无需长时间的 GC 暂停时间。

JDK10 : 为 G1 提供并行的 Full GC

这个 JEP 的目标只有一个，那就是要把 G1 的 full GC 算法给并行化，从而提高性能。你知道，G1 最大的亮点就是可以尽量的避免 full gc。但毕竟是“尽量”，在有些情况下，G1 就要进行 full gc 了，比如如果它无法足够 快的 回收内存的时候，

它就会强制停止所有的应用 线程 然后清理。在 Java10 之前，一个单线程版的标记-清除-压缩算法被用于 full gc。

所以为了尽量减少 full gc 带来的影响，在 Java10 中，就把之前的那个单线程版的标记-清除-压缩的 full gc 算法改成了支持多个线程同时 full gc。这样也算是减少了 full gc 所带来的停顿。

你可以通过-XX:ParallelGCThreads 这个 参数 来指定用于并行 GC 的线程数。

JDK11：引入的 ZGC

ZGC, 这应该是 JDK11 最为瞩目的特性, 没有之一。

ZGC 是一个并发, 基于 region, 压缩型的垃圾收集器, 只有 root 扫描阶段会 STW(stop the world), 因此 GC 停顿时间不会随着堆的增长和存活对象的增长而变长。

ZGC 的设计目标是：支持 TB 级内存容量，暂停时间低（<10ms），对整个程序吞吐量的影响小于 15%。 将来还可以扩展实现机制，以支持不少令人兴奋的功能，例如多层堆（即热对象置于 DRAM 和冷对象置于 NVMe 闪存），或压缩堆。

JDK11：引入的 Epsilon GC

JDK 上对这个特性的描述是: 开发一个处理内存分配但不实现任何实际内存回收机制的 GC, 一旦可用堆内存用完, JVM 就会退出.

如果有 System.gc()调用, 实际上什么也不会发生(这种场景下和-`XX:+DisableExplicitGC` 效果一样), 因为没有内存回收, 这个实现可能会警告用户尝试强制 GC 是徒劳.

用法 : `-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC`

如果使用选项-XX:+UseEpsilonGC, 程序很快就因为堆空间不足而退出。使用这个选项的原因 :

提供完全被动的 GC 实现, 具有有限的分配限制和尽可能低的延迟开销,但代价是内存占用和内存吞吐量。

这个垃圾收集器实际上不做任何工作，因此允许您精确测量应用程序的内存使用情况。如果您想保持尽可能低的内存操作吞吐量，则很有用。主要用途如下 :

- 性能测试(它可以帮助过滤掉 GC 引起的性能假象)
- 内存压力测试(例如,知道测试用例 应该分配不超过 1GB 的内存, 我们可以使用-Xmx1g –XX:+UseEpsilonGC, 如果程序有问题, 则程序会崩溃)

- 非常短的 JOB 任务(对象这种任务, 接受 GC 清理堆那都是浪费空间)
- VM 接口测试
- Last-drop 延迟&吞吐改进

JDK12：Shenandoah GC：低停顿时间的 GC

6.2.1 背景和设计思路

Shenandoah 垃圾回收器是 Red Hat 在 2014 年宣布进行的一项垃圾收集器研究项目 Pauseless GC 的实现，旨在**针对 JVM 上的内存收回实现低停顿的需求**。该设计将与应用程序线程并发，通过交换 CPU 并发周期和空间以改善停顿时间，使得垃圾回收器执行线程能够在 Java 线程运行时进行堆压缩，并且标记和整理能够同时进行，因此避免了在大多数 JVM 垃圾收集器中所遇到的问题。

据 Red Hat 研发 Shenandoah 团队对外宣称，Shenandoah 垃圾回收器的暂停时间与堆大小无关，这意味着无论将堆设置为 200 MB 还是 200 GB，都将拥有一致的系统暂停时间，不过实际使用性能将取决于实际工作堆的大小和工作负载。

与其他 Pauseless GC 类似，**Shenandoah GC 主要目标是 99.9% 的暂停小于 10ms，暂停与堆大小无关等。**

这是一个实验性功能，不包含在默认（Oracle）的 OpenJDK 版本中。

6.2.5 工作原理

从原理的角度，我们可以参考该项目官方的示意图，其内存结构与 G1 非常相似，都是将内存划分为类似棋盘的 region。整体流程与 G1 也是比较相似的，**最大的区别在于实现了并发的 疏散(Evacuation) 环节，引入的 Brooks Forwarding Pointer 技术使得 GC 在移动对象时，对象引用仍然可以访问。**

Shenandoah GC 工作周期如下所示：

上图对应工作周期如下：

1. Init Mark 启动并发标记 阶段
2. 并发标记遍历堆阶段
3. 并发标记完成阶段
4. 并发整理回收无活动区域阶段
5. 并发 Evacuation 整理内存区域阶段
6. Init Update Refs 更新引用初始化 阶段
7. 并发更新引用阶段
8. Final Update Refs 完成引用更新阶段
9. 并发回收无引用区域阶段

6.2.6 信息延展

不是唯有 GC 停顿可能导致常规应用程序响应时间比较长。具有较长的 GC 停顿时间会导致系统响应慢的问题，但响应时间慢并非一定是 GC 停顿时间长导致的，队列延迟、网络延迟、其他依赖服务延迟和操作提供调度程序抖动等都可能导致响应变慢。使用 Shenandoah 时需要全面了解系统运行情况，综合分析系统响应时间。下面是 jbb15 benchmark 中，Shenandoah GC 相对于其他主流 GC 的表现。

各种 GC 工作负载对比：

GC 暂停相比于 CMS 等选择有数量级程度的提高，对于 GC 暂停非常敏感的场景，价值还是很明显的，能够在 SLA 层面有显著提高。当然，这种对于低延迟的保证，也是以消耗 CPU 等计算资源为代价的，实际吞吐量表现也不是非常明朗，需要看企业的实际场景需求，并不是一个一劳永逸的解决方案。

Shenandoah 开发团队在实际应用中的测试数据：

收 集 器	运行时间	总 停 顿	最大停顿	平均停顿
Shenandoah	387.602s	320ms	89.79ms	53.01ms
G1	312.052s	11.7s	1.24s	450.12ms
CMS	285.264s	12.78s	4.39s	852.26ms
Parallel Scavenge	260.092s	6.59s	3.04s	823.75ms

JDK12：可中断的 G1 Mixed GC

简言之，当 G1 垃圾回收器的回收超过暂停时间的目标，则能中止垃圾回收过程。

G1 是一个垃圾收集器，设计用于具有大量内存的多处理器机器。由于它提高了性能效率，G1 垃圾收集器最终将取代 CMS 垃圾收集器。

该垃圾收集器设计的主要目标之一是满足用户设置的预期的 JVM 停顿时间。

G1 采用一个高级分析引擎来选择在收集期间要处理的工作量，此选择过程的结果是一组称为 GC 回收集（collection set(`CSet`)）的区域。一旦收集器确定了 GC 回收集 并且 GC 回收、整理工作已经开始，这个过程是 without stopping 的，即 G1 收集器必须完成收集集合的所有区域中的所有活动对象之后才能停止；但是如果收集器选择过大的 GC 回收集，此时的 STW 时间会过长超出目标 pause time。

这种情况在 mixed collections 时候比较明显。这个特性启动了一个机制，当选择了一个比较大的 collection set，Java 12 中将把 GC 回收集（混合收集集合）拆分为 mandatory（必需或强制）及 optional 两部分(当完成 `mandatory` 的部分，如果还有剩余时间则会去处理 `optional` 部分)来将 mixed collections 从 without stopping 变为 abortable，以更好满足指定 pause time 的目标。

- 其中必需处理的部分包括 G1 垃圾收集器不能递增处理的 GC 回收集的部分（如：年轻代），同时也可以包含老年代以提高处理效率。
- 将 GC 回收集拆分为必需和可选部分时，垃圾收集过程优先处理必需部分。同时，需要为可选 GC 回收集部分维护一些其他数据，这会产生轻微的 CPU 开销，但小于 1 %的变化，同时在 G1 回收器处理 GC 回收集期间，本机内存使用率也可能会增加，使用上述情况只适用于包含可选 GC 回收部分的 GC 混合回收集合。
- 在 G1 垃圾回收器完成收集需要必需回收的部分之后，如果还有时间的话，便开始收集可选的部分。但是粗粒度的处理，可选部分的处理粒度取决于剩余的时间，一次只能处理可选部分的一个子集区域。在完成可选收集部分的收集后，G1 垃圾回收器可以根据剩余时间决定是否停止收集。如果在处理完必需处理的部分后，剩余时间不足，总时间花销接近预期时间，G1 垃圾回收器也可以中止可选部分的回收以达到满足预期停顿时间的目标。

JDK12：增强 G1，自动返回未用堆内存给操作系统

6.8.1 概述

上面介绍了 Java 12 中增强了 G1 垃圾收集器关于混合收集集合的处理策略，这节主要介绍在 Java 12 中同时也对 G1 垃圾回收器进行了改进，使其能够在空闲时自动将 Java 堆内存返还给操作系统，这也是 Java 12 中的另外一项重大改进。

目前 Java 11 版本中包含的 G1 垃圾收集器暂时无法及时将已提交的 Java 堆内存返回给操作系统。为什么呢？ G1 目前只有在 full GC 或者 concurrent cycle（并发处理周期）的时候才会归还内存，由于这两个场景都是 G1 极力避免的，因此在大多

数场景下可能不会及时归还 committed Java heap memory 给操作系统。除非有外部强制执行。

在使用云平台的容器环境中，这种不利之处特别明显。即使在虚拟机不活动，但如果仍然使用其分配的内存资源，哪怕是其中的一小部分，G1 回收器也仍将保留所有已分配的 Java 堆内存。而这将导致用户需要始终为所有资源付费，哪怕是实际并未用到，而云提供商也无法充分利用其硬件。如果在此期间虚拟机能够检测到 Java 堆内存的实际使用情况，并在利用空闲时间自动将 Java 堆内存返还，则两者都将受益。

6.8.2 具体操作

为了尽可能的向操作系统返回空闲内存，G1 垃圾收集器将在应用程序不活动期间定期生成或持续循环检查整体 Java 堆使用情况，以便 G1 垃圾收集器能够更及时的将 Java 堆中不使用内存部分返还给操作系统。对于长时间处于空闲状态的应用程序，此项改进将使 JVM 的内存利用率更加高效。

而在用户控制下，可以可选地执行 Full GC，以使返回的内存量最大化。

JDK12 的这个特性新增了两个参数分别是 G1 PeriodicGCInterval 及 G1 PeriodicGCSystemLoadThreshold，设置为 0 的话，表示禁用。如果应用程序为非活动状态，在下面两种情况任何一个描述下，G1 回收器会触发定期垃圾收集：

- 自上次垃圾回收完成以来已超过 G1PeriodicGCInterval (`milliseconds`)， 并且此时没有正在进行的垃圾回收任务。如果 G1PeriodicGCInterval 值为零表示禁用快速回收内存的定期垃圾收集。
- 应用所在主机系统上执行方法 getloadavg()，默认一分钟内系统返回的平均负载值低于 G1PeriodicGCSystemLoadThreshold 指定的阈值，则触发 full GC 或者 concurrent GC(如果开启 `G1PeriodicGCInvokesConcurrent`)，GC 之后 Java heap size 会被重写调整，然后多余的内存将会归还给操作系统。如果 G1PeriodicGCSystemLoadThreshold 值为零，则此条件不生效。

如果不满足上述条件中的任何一个，则取消当期的定期垃圾回收。等一个 G1PeriodicGCInterval 时间周期后，将重新考虑是否执行定期垃圾回收。

G1 定期垃圾收集的类型根据 G1PeriodicGCInvokesConcurrent 参数的值确定：如果设置值了，G1 垃圾回收器将继续上一个或者启动一个新并发周期；如果没有设置值，则 G1 回收器将执行一个 Full GC。在每次一次 GC 回收末尾，G1 回收器将调整当前的 Java 堆大小，此时便有可能会将未使用内存返还给操作系统。新的 Java 堆内存大小根据现有配置确定，具体包括下列配置：- XX:MinHeapFreeRatio、-XX:MaxHeapFreeRatio、-Xms、-Xmx。

默认情况下，G1 回收器在定期垃圾回收期间新启动或继续上一轮并发周期，将最大限度地减少应用程序的中断。如果定期垃圾收集严重影响程序执行，则需要考虑整个系统 CPU 负载，或让用户禁用定期垃圾收集。

JDK13：ZGC:取消使用未使用的内存

在 JDK 11 中，Java 引入了 ZGC，这是一款可伸缩的低延迟垃圾收集器，但是当时只是实验性的。号称不管你开了多大的堆内存，它都能保证在 10 毫秒内释放 JVM，不让它停顿在那。但是，当时的设计是它不能把内存归还给操作系统。对于比较关心内存占用的应用来说，肯定希望进程不要占用过多的内存空间了，所以这次增加了这个特性。

JDK 11

JDK 11 is the open-source reference implementation of version 11 of the Java SE Platform as specified by by JSR 384 in the Java Community Process.

JDK 11 reached General Availability on 25 September 2018. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

Features

181: Nest-Based Access Control
309: Dynamic Class-File Constants
315: Improve Aarch64 Intrinsics
318: Epsilon: A No-Op Garbage Collector
320: Remove the Java EE and CORBA Modules
321: HTTP Client (Standard)
323: Local-Variable Syntax for Lambda Parameters
324: Key Agreement with Curve25519 and Curve448
327: Unicode 10
328: Flight Recorder
329: ChaCha20 and Poly1305 Cryptographic Algorithms
330: Launch Single-File Source-Code Programs
331: Low-Overhead Heap Profiling
332: Transport Layer Security (TLS) 1.3
333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)
335: Deprecate the Nashorn JavaScript Engine

在 Java 13 中，JEP 351 再次对 ZGC 做了增强，将没有使用的堆内存归还给操作系统。ZGC 当前不能把内存归还给操作系统，即使是那些很久都没有使用的内存，

也只进不出。这种行为并不是对任何应用和环境都是友好的，尤其是那些内存占用敏感的服务，例如：

1. 按需付费使用的容器环境；
2. 应用程序可能长时间闲置，并且和很多其他应用共享和竞争资源的环境；
3. 应用程序在执行期间有非常不同的堆空间需求，例如，可能在启动的时候所需的堆比稳定运行的时候需要更多的堆内存。

7.4.3 使用细节

ZGC 的堆由若干个 Region 组成，每个 Region 被称之为 **ZPage**。每个 Zpage 与数量可变的已提交内存相关联。当 ZGC 压缩堆的时候，ZPage 就会释放，然后进入 page cache，即 **ZPageCache**。这些在 page cache 中的 ZPage 集合就表示没有使用部分的堆，这部分内存**应该**被归还给操作系统。回收内存可以简单的通过从 page cache 中逐出若干个选好的 ZPage 来实现，由于 page cache 是以 LRU（Least recently used，最近最少使用）顺序保存 ZPage 的，并且按照尺寸（小，中，大）进行隔离，因此逐出 ZPage 机制和回收内存相对简单了很多，**主要挑战是设计关于何时从 page cache 中逐出 ZPage 的策略。**

一个简单的策略就是设定一个超时或者延迟值，表示 ZPage 被驱逐前，能在 page cache 中驻留多长时间。这个超时时间会有一个合理的默认值，也可以通过 JVM 参数覆盖它。Shenandoah GC 用了一个类型的策略，默认超时时间是 5 分钟，可以通过参数**-XX:ShenandoahUncommitDelay = milliseconds** 覆盖默认值。

像上面这样的策略可能会运作得相当好。但是，用户还可以设想更复杂的策略：**不需要添加任何新的命令行选项。**例如，基于 GC 频率或某些其他数据找到合适超时值的启发式算法。**JDK13 将使用哪种具体策略目前尚未确定。**可能最初只提供一个简单的超时策略，使用**-XX:ZUncommitDelay = seconds** 选项，以后的版本会添加更复杂、更智能的策略（如果可以的话）。

uncommit 能力默认是开启的，但是无论指定何种策略，ZGC 都不能把堆内存降到低于 Xms。这就意味着，如果 Xmx 和 Xms 相等的话，这个能力就失效了。-XX:-ZUncommit 这个参数也能让这个内存管理能力失效。

JDK14：弃用 ParallelScavenge 和 SerialOld GC 组合

- 由于维护和兼容性测试的成本，在 JDK 8 时将 Serial+CMS、ParNew+Serial Old 这两个组合声明为废弃（JEP 173），并在 JDK 9 中完全取消了这些组合的支持（JEP214）
- ParallelScavenge + SerialOld GC 的 GC 组合要被标记为 Deprecate 了。
- JDK 官方给出将这个 GC 组合标记为 Deprecate 的理由是：这个 GC 组合需要大量的代码维护工作，并且，这个 GC 组合很少被使用。因为它的使用场景应该是一个很大的 Young 区配合一个很小的 Old 区，这样的话，Old 区用 SerialOldGC 去收集时停顿时间我们才能勉强接受。
- 废弃了 parallel young generation GC 与 SerialOld GC 的组合(-XX:+UseParallelGC 与-XX:-UseParallelOldGC 配合开启)，现在使用-

XX:+UseParallelGC -XX:-UseParallelOldGC 或者-XX:-UseParallelOldGC 都会出现告警如下：

```
Java HotSpot(TM) 64-Bit Server VM warning: Option
UseParallelOldGC was deprecated in version 14.0 and will likely
be removed in a future release.
```

JDK14：删除 CMS 垃圾回收器

- 该来的总会来，自从 G1（基于 Region 分代）横空出世后，CMS 在 JDK9 中就被标记为 Deprecate 了（JEP 291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector）

- CMS 的弊端：

  1.会产生内存碎片，导致并发清除后，用户线程可用的空间不足。

  2.既然强调了并发（Concurrent），CMS 收集器对 CPU 资源非常敏感

  3.CMS 收集器无法处理浮动垃圾

- 上述的这些问题，尤其是碎片化问题，给你的 JVM 实例就像埋了一颗炸弹。说不定哪次就在你的业务高峰期来一次 FGC。当 CMS 停止工作时，会把 Serial Old GC 作为备选方案，而 Serial Old GC 是 JVM 中性能最差的垃圾回收方式，停顿个几秒钟，上十秒都有可能。

- 移除了 CMS 垃圾收集器，如果在 JDK14 中使用-XX:+UseConcMarkSweepGC 的话，JVM 不会报错，只是给出一个 warning 信息。

JDK14：ZGC on macOS 和 windows

- JDK14 之前，ZGC 仅 Linux 才支持。现在 mac 或 Windows 上也能使用 ZGC 了，示例如下：

  ```
  -XX:+UnlockExperimentalVMOptions -XX:+UseZGC
  ```

- ZGC 与 Shenandoah 目标高度相似，在尽可能对吞吐量影响不大的前提下，实现在任意堆内存大小下都可以把垃圾收集的停顿时间限制在十毫秒以内的低延迟。

- 《深入理解 Java 虚拟机》一书中这样定义 ZGC：ZGC 收集器是一款基于 Region 内存布局的，（暂时）不设分代的，使用了读屏障、染色指针和内存多重映射等技术来实现可并发的标记-压缩算法的，以低延迟为首要目标的一款垃圾收集器。

虽然 ZGC 还在试验状态，没有完成所有特性，但此时性能已经相当亮眼，用“令人震惊、革命性”来形容，不为过。

未来将在服务端、大内存、低延迟应用的首选垃圾收集器。

JDK15：ZGC 功能转正

ZGC 是 Java 11 引入的新的垃圾收集器（JDK9 以后默认的垃圾回收器是 G1），经过了多个实验阶段，自此终于成为正式特性。

自 2018 年以来，ZGC 已增加了许多改进，从并发类卸载、取消使用未使用的内存、对类数据共享的支持到改进的 NUMA 感知。此外，最大堆大小从 4 TB 增加到 16 TB。支持的平台包括 Linux、Windows 和 MacOS。

ZGC 是一个重新设计的并发的垃圾回收器，通过减少 GC 停顿时间来提高性能。

但是这并不是替换默认的 GC，默认的 GC 仍然还是 G1；之前需要通过-`XX:+UnlockExperimentalVMOptions`、`-XX:+UseZGC` 来启用 ZGC，现在只需要-`XX:+UseZGC` 就可以。相信不久的将来它必将成为默认的垃圾回收器。

相关的参数有 ZAllocationSpikeTolerance、ZCollectionInterval、ZFragmentationLimit、ZMarkStackSpaceLimit、ZProactive、ZUncommit、ZUncommitDelay ZGC-specific JFR events(ZAllocationStall、ZPageAllocation、ZPageCacheFlush、ZRelocationSet、ZRelocationSetGroup、ZUncommit)也从 experimental 变为 product

JDK15：Shenandoah 垃圾回收算法转正

Shenandoah 垃圾回收算法终于从实验特性转变为产品特性，这是一个从 JDK 12 引入的回收算法，该算法通过与正在运行的 Java 线程同时进行疏散工作来减少 GC 暂停时间。Shenandoah 的暂停时间与堆大小无关，无论堆栈是 200 MB 还是 200 GB，都具有相同的一致暂停时间。

怎么形容 Shenandoah 和 ZGC 的关系呢？异同点大概如下：

- 相同点：性能几乎可认为是相同的
- 不同点：ZGC 是 Oracle JDK 的，根正苗红。而 Shenandoah 只存在于 OpenJDK 中，因此使用时需注意你的 JDK 版本

打开方式：使用-XX:+UseShenandoahGC 命令行参数打开。

Shenandoah 在 JDK12 被作为 experimental 引入，在 JDK15 变为 Production；之前需要通过-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC 来启用，现在只需要-XX:+UseShenandoahGC 即可启用

JDK16：ZGC 并发线程处理

我们知道 ZGC 是 HotSpot JVM 中一种低延时的垃圾回收算法。但是在线程的堆栈处理过程中，总有一个制约因素就是 safepoints。在 safepoints 这个点，java 的线程是要暂停执行的，从而限制了 GC 的效率。

回顾：

我们都知道，在之前，需要 GC 的时候，为了进行垃圾回收，需要所有的线程都暂停下来，这个暂停的时间我们成为 **Stop The World**。

而为了实现 STW 这个操作， JVM 需要为每个线程选择一个点停止运行，这个点就叫做**安全点（Safepoints）**。

而 ZGC 的并发线程堆栈处理可以保证 java 线程可以在 GC safepoints 的同时可以并发执行。它有助于提高所开发的 java 软件应用程序的性能和效率。

10. GraalVM

在 JDK11 中将运行 JS 应用 Nashorn 引擎标记为过时，并在 JDK15 中删除了 Nashorn JavaScript Engine 及 jjs 命令行工具。

在 JDK11 中取以代之的是 GraalVM。GraalVM 是一个运行时平台，它支持 Java 和其他基于 Java 字节码的语言，但也支持其他语言，如 JavaScript，Ruby，Python 或 LLVM。性能是 Nashorn 的 2 倍以上。

在 JDK17 中，移除实验性的 java 版本的 AOT 及 JIT Compiler，具体移除：

```
jdk.aot – the jaotc tool
jdk.internal.vm.compiler – the Graal compiler
jdk.internal.vm.compiler.management – Graal's MBean
```

后续要使用可以使用 Graal 编译器 (GraalVM)进行 JIT 编译。

Graal VM 在 HotSpot VM 基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”的运行平台使用。语言包括：Java、Scala、Groovy、Kotlin；C、C++、JavaScript、Ruby、Python、R 等

11. 小结与展望

随着云计算和 AI 等技术浪潮，当前的计算模式和场景正在发生翻天覆地的变化，不仅对 Java 的发展速度提出了更高要求，也深刻影响着 Java 技术的发展方向。**传统的大型企业或互联网应用，正在被云端、容器化应用、模块化的微服务甚至是函数(FaaS， Function-as-a-Service)所替代。**

Java 虽然标榜面向对象编程，却毫不顾忌的加入面向接口编程思想，又扯出匿名对象之概念，每增加一个新的东西，对 Java 的根本所在的面向对象思想的一次冲击。反观 Python，抓住面向对象的本质，又能在函数编程思想方面游刃有余。**Java 对标 C/C++，以抛掉内存管理为卖点，却又陷入了 JVM 优化的噩梦。**选择比努力更重要，选择 Java 的人更需要对它有更清晰的认识。

Java 需要在新的计算场景下，改进开发效率。这话说的有点笼统，我谈一些自己的体会，Java 代码虽然进行了一些类型推断等改进，更易用的集合 API 等，但仍然给开发者留下了过于刻板、形式主义的印象，这是一个长期的改进方向。

尚硅谷