Java 核心技术笔记

Pionpill [1]

本文档为作者学习《Java 核心技术》[2]一书时的笔记。

2022 年 10 月 14 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

[2]《Core Java》:Cay S. Horstmann 原书第 11 版 2021 第一版

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础 (操作系统，数据结构，计算机网络，编译原理等)，本人在撰写此笔记时已经学过 C++/Python/JavaScript 等语言，且对 Java 语言有一定了解，过于基础的内容本人不会再介绍。

本文文章结构与原书大体相同，分上下两卷。上卷基础部分，基础部分的基础内容不再赘述，下卷深入 Java 语言。但每卷结构都有修改。

原文花费了大段篇幅书写 Java 一些跨时代的特性 (相对之前的高级语言而言)，但这些特性中的绝大部分在现代程序设计语言中已是司空见惯的基本属性，因此，本人会按照自己的思维撰写笔记。

本人的编写及开发环境如下：

- IDE: VSCode 1.62
- JDK11

2022 年 10 月 14 日

目录

第一部分　Java 基础　1

I　Java 基础语法

1　Java 程序设计环境 2

1.1　Java 开发工具包 2

2　Java 基本程序设计结构 3

2.1　Java 程序与注释 3

2.1.1　Java 程序 3

2.1.2　Java 注释 3

2.2　数值型 3

2.2.1　Java 的一些性质 3

2.2.2　数值的基本数据类型 3

2.2.3　数值转换 4

2.2.4　大数 5

2.3　字符串 5

2.3.1　== 比较 5

2.3.2　equals() 5

2.4　输入与输出 6

2.4.1　标准输入 6

2.4.2　文件的输入输出 6

2.5　数组 6

2.5.1　数组基础 6

2.5.2　for each 循环 7

2.5.3　数组拷贝 7

II　面向对象

3　对象与类 8

3.1　包 8

3.1.1　包名 8

3.1.2　类的导入 8

3.1.3　静态导入 8

3.1.4　类路径 9

3.2 JAR 文件 9
3.2.1 创建 JAR 文件 9
3.2.2 清单文件 10
3.2.3 可执行 JAR 文件 11
3.3 文档注释 11
3.3.1 通用注释 11
3.3.2 方法注释 11
3.3.3 包注释 12
3.3.4 注释抽取 12

4 继承 13
4.1 类继承基础 13
4.1.1 多态与强制类型转换 13
4.1.2 抽象类 13
4.2 所有子类的超类: Object 14
4.2.1 Object 类型的变量 14
4.2.2 `equals` 方法 14
4.2.3 `hashCode` 方法 15
4.2.4 `toString` 方法 16
4.3 泛型数组列表 16
4.3.1 声明数组列表 16
4.3.2 访问数组列表元素 17
4.4 对象包装器与自动装箱 17
4.5 参数数量可变的方法 18
4.6 枚举类 19
4.7 反射 19

5 接口，lambda 表达式，内部类 20
5.1 接口 20
5.1.1 接口的概念 20
5.1.2 接口的属性 21
5.1.3 静态和私有方法 21
5.1.4 默认方法 21
5.1.5 回调 22
5.1.6 `Comparator` 接口 22
5.1.7 对象克隆 23
5.2 lambda 表达式 24
5.2.1 lambda 表达式的语法 24
5.2.2 函数式接口 24
5.2.3 方法引用 24

5.2.4 构造器引用 . . . 25
5.2.5 变量作用域 . . . 25
5.2.6 处理 lambda 表达式 . . . 26
5.3 内部类 . . . 26
5.3.1 使用内部类访问对象状态 . . . 26
5.3.2 局部内部类 . . . 28
5.3.3 匿名内部类 . . . 28
5.3.4 静态内部类 . . . 28

III 核心特性

6 异常，断言和日志 . . . 29
6.1 错误处理 . . . 29
6.1.1 异常分类 . . . 29
6.1.2 声明检查型异常 . . . 29
6.1.3 创建异常类 . . . 30
6.2 捕获异常 . . . 30
6.2.1 捕获异常 . . . 30
6.2.2 `try-with-Resources` 语句 . . . 31
6.2.3 处理异常的技巧 . . . 31
6.3 使用断言 . . . 32
6.3.1 断言的概念 . . . 32
6.3.2 启用很禁用断言 . . . 32
6.4 日志 . . . 32
6.4.1 使用日志 . . . 32

7 泛型程序设计 . . . 34
7.1 类型参数的好处 . . . 34
7.2 定义简单泛型 . . . 34
7.2.1 泛型类 . . . 34
7.2.2 泛型方法 . . . 35
7.2.3 类型变量的限定 . . . 35
7.3 泛型代码与虚拟机 . . . 35
7.3.1 类型擦除 . . . 35
7.3.2 转换泛型表达式 . . . 36
7.3.3 转换类型方法 . . . 36
7.4 限制与局限性 . . . 36
7.5 泛型类型的继承规则 . . . 38
7.6 通配符类型 . . . 38
7.6.1 通配符上下界 . . . 38

7.6.2 无限定通配符 38

8 集合 39

8.1 Java 集合框架 39
8.1.1 集合接口与实现分离 39
8.1.2 `Collection` 接口 39
8.1.3 迭代器 39
8.1.4 集合框架中的接口 40
8.2 具体集合 40
8.2.1 链表与数组列表 40
8.3 映射 40
8.3.1 基本映射操作 41
8.3.2 弱散列映射 41
8.3.3 链接散列集与映射 41
8.3.4 标识散列映射 41

9 并发 42

9.1 什么是线程 42
9.2 线程状态 42
9.2.1 新建线程 42
9.2.2 可运行线程 42
9.2.3 阻塞和等待线程 43
9.2.4 终止线程 43
9.3 线程属性 43
9.3.1 中断线程 43
9.3.2 守护线程 44
9.3.3 线程名 45
9.3.4 未捕获异常的处理器 45
9.3.5 线程优先级 45
9.4 同步 45
9.4.1 静态条件详解 46
9.4.2 锁对象 46
9.4.3 条件对象 47
9.4.4 `synchronized` 关键字 48
9.4.5 同步块 49
9.4.6 线程局部变量 49
9.5 任务和线程池 49
9.5.1 `Callable` 与 `Future` 49

第一部分

Java 基础

I Java 基础语法

有关 Java 与 Sun 公司的发展历史这里直接省略，现在 Java 由 Oracle 公司维护，这不影响我们编写 Java 程序，有兴趣请自行查阅资料。

1 Java 程序设计环境

1.1 Java 开发工具包

众所周知，Java 为了实现跨平台提供了许多类型的软件，随之产生了许多术语如下:

表 1.1 Java 术语

术语名	缩写	解释
Java Development Kit	JDK	编写 Java 程序的程序员使用的软件
Java Runtime Environment	JRE	运行 Java 程序的用户使用的软件
Server JRE	-	服务器上运行 Java 程序的软件
Standard Edition	SE	标准版: 桌面或简单应用的 Java 平台
Enterprise Edition	EE	企业版: 复杂服务器应用的 Java 平台
Micro Edition	ME	小型设备的 Java 平台
Open JDK	-	免费开源实现

JDK 的某些远古版本被称为 SDK。我们一般所用的都是 JAVA JDK SD X 其中 JDK SD 省略。X 表示版本，本文默认使用 Java 8 及以上版本[1]。

可以在 Oracle 官网下载对应版本的 JDK: `https://www.oracle.com/java/technologies/downloads`

[1] 笔者用的 Java11

2 Java 基本程序设计结构

2.1 Java 程序与注释

2.1.1 Java 程序

一个 Java 源文件的文件名必须和此文件中的公有类[2]名相同，我们可以使用

```
java sample arg
```

运行对应的 java 程序，注意不需要添加后缀名。其中 args 代表传入的参数。

2.1.2 Java 注释

Java 提供了三种注释方式：

- 单行注释: //
- 界定注释: /* */
- 多行注释: /** */

在多行注释中可以通过 @ + 关键字书写文档，比如 @author Pionpill 表示这段代码由 Pionpill 书写。要养成良好的注释习惯捏。

2.2 数值型

2.2.1 Java 的一些性质

不同于 C++，Java 数字的值不会因平台不同而有变动，也即与机器硬件无关。在 A 机器上存储的整型 X 在 B 机器上也一定会是 X，C++ 在这个过程中可能会将 X 变为某个不知道的值。

Java 的所有函数传参都是传值，而不是传引用。我在 Python 和 JavaScript 笔记中已经多次说明过这两者的区别，相信有经验的 coder 知道这点有多重要。

Java 中除了基本类型 (数值，字符串，布尔值) 不是对象，其他数据类型均为对象。

2.2.2 数值的基本数据类型

下面简单过一下 Java 的基础数据类型。

整型：Java 提供了以下四种整型数:

[2]这也代表着一个 Java 程序中只能由一个类提供对外接口。

表 1.2 Java 整型

类型	存储空间	取值范围
int	4 byte	-20 亿——20 亿
short	2 byte	-32768 ——32767
long	8 byte	-9^{19}——9^{19}
byte	1 byte	-128 ——127

有以下注意点:

- Java 没有 unsigned 形式的整型数。
- 可以给数字字面量加下划线，编译器会自动去除，这只是为了添加可读性。
- 可以通过在字面量后加 l/L 指定长整型。

浮点数：Java 提供了两种浮点类型:

表 1.3 浮点类型

类型	存储空间	有效位
float	4 byte	6-7 位
double	8 byte	15 位

有以下注意点:

- 三个特殊浮点数值: 正无穷大，负无穷大，NAN。
- 可以通过在字面量后加 f/F d/D 指定浮点数类型。

布尔型 (boolean)

注意 Java 中的 boolean 型不是整型就行了[3]。

2.2.3 数值转换

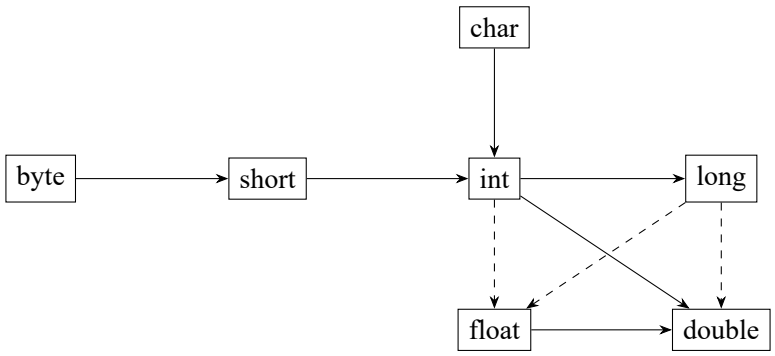

图 2.1 数值类型之间的合法转换

[3]Python 中布尔值就是 0/1。

上图所示，实线代表无信息丢失，虚线代表可能有信息丢失。此外，在进行数值计算时会自动向精度更高的数值类型转换。

2.2.4 大数

在 `java.math` 包中包含两个大数类: BigInteger 和 BigDecimal。这两个类可以处理包含任意长度数字序列的数值。

2.3 字符串

字符串基本操作和其他语言一样，下面讲一下易混淆的 equals 和 == 比较运算。

2.3.1 == 比较

对于基本数据类型，== 比较值是否相同 (因为基本数据类型没有引用捏)。

对于引用数据类型，== 比较引用的地址是否相同。

2.3.2 equals()

Object 中的 `equals` 方法是这样写的:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

也即和 == 相同，对引用数据类型比较地址是否相同。但绝大多数类都会重写 `equals` 方法，在 string 中就这样重写了 `equals` 方法:

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
```

```
    }
    return false;
}
```

可以理解为: 比较内容是否相同。

2.4 输入与输出

2.4.1 标准输入

利用 Java 进行标准输出非常简单，调用 `System.out.print()` 即可，而构建标准输入却显得不那么简单。想要通过控制台进行输入，往往需要执行以下几个步骤:

- 构建 `Scanner` 对象:

```
Scanner in = new Scanner(System.in);
```

- 调用对应的方法:

  Scanner 对象针对不同的数据类型设计了多种方法，例如最简单的 `nextLine()` 读取一行输入，`next()` 读取一个单词，`nextInt()` 读取一个整数。

同时针对读取密码，Java 还设计了 `Console` 类，但它不如 `Scanner` 那么好用。

2.4.2 文件的输入输出

文件的读取同样需要构建 Scanner 对象:

```
Scanner in = new Scanner(Path.of("myfile.txt"),StandardCharsets.UTF_8);
```

文件写入则需要构造 PrintWriter 对象。

```
PrintWriter out = new PrintWriter("myfile.txt",StandardCharsets.UTF_8);
```

在这个过程中可能会因为文件不存在或无法在指定路径创建文件等 IO 错误。

2.5 数组

2.5.1 数组基础

数组与字符串一样是不可变数据类型。其基础的创建语句如下 (其中数字代表数组长度):

```
int[] a = new int[100];
```

Java 提供了一种创建数组的简写形式:

```
int[] a = {1,2,3,4,5};
```

也可以通过这种方式创建匿名数组:

```
new int[] {1,2,3,4,5};
```

多维数组的申明也十分简单:

```
double[][] a = new double[m][n];
```

2.5.2 for each 循环

Java 提供了一种遍历可迭代对象的简单语法:

```
for(int element:a)
    System.out.println(element);
```

这和传统的 `for` 循环结合下标达到的效果是相同的。

2.5.3 数组拷贝

由于 Java 的所有参数传递都是值传递，直接使用下面这种形式拷贝数组仅是一种浅拷贝。

```
int[] a = {1,2,3};
int[] b = a;
```

如果想要深拷贝数组，则需要使用到 `Arrays` 类的 `copyOf` 方法。

```
int[] b = Arrays.copyOf(luckyNumbers, 2*luckyNumbers.length);
```

其中，第二个参数是数组长度。

II 面向对象

3 对象与类

3.1 包

3.1.1 包名

使用包的主要原因是确保类名的唯一性。一般，为了确保包名的绝对唯一性，要用一个因特尔域名以逆序的形式作为包名，然后对于不同的工程使用不同的子包。

从编译器的角度来看，嵌套的包之间没有任何关系。例如，`java.util` 与 `java.util.jar` 包毫无关系。每一个包都是独立的类集合。

3.1.2 类的导入

一个类可以使用所属包中的所有类，以及其他包中的公共类。

可以使用两种方式访问另一个包中的公共类。第一种方式是使用完全限定名；就是包名后面跟着类名:

```
java.time.LocalDate today = java.time.LocalDate.now();
```

这种方式很繁琐，更常见的是使用 `import` 语句引入:

```
import java.time.*;
```

上面这种使用通配符导入所有类对代码的规模没有任何负面影响。不过如果能精确到类名，可以增加代码可读性。此外，* 可以导入目标包下的所有类，但是不能导入文件下的包。

在极个别情况下，仍有可能出现类名冲突，这时候就只能使用完全限定名了:

```
import java.util.*;
import java.sql.*;
var deadline = new java.util.Date();
var today = new java.sql.Date(...);
```

此时也体现了 `var` 声明语句的好处：可以自动判断数据类型，省去了大段代码。

3.1.3 静态导入

有一种 `import` 语句允许导入静态方法和静态字段，而不只是类。

比如，在源文件顶部，添加一条指令:

```
import static java.lang.System.*;
```

```
out.println("Goodbye");
```

另外，还可以导入特定的方法或字段:

```
import static java.lang.System.out;
```

3.1.4 类路径

类路径是所有包含类文件的路径的集合。在 UNIX 环境中，类路径用: 分隔，在 Windows 环境中，则以; 分隔。

设置类路径可以使用 -classpath 选项，也可以通过设置 CLASSPATH 环境变量来指定。

```
java -classpath c:\classdir;,;c:\archives\archive.jar MyProg
set CLASSPATH = c:\classdir;.;c:\archives\archive.jar
```

直到推出 shell 为止，类路径设置均有效。

3.2 JAR 文件

Java 归档文件 (JAR) 文件既可以包含类文件，也可以包含诸如图像和声音等其他类型的文件。此外，JAR 文件是压缩的，它使用了我们熟悉的 ZIP 压缩格式。

3.2.1 创建 JAR 文件

通常，jar 命令的格式如下:

```
jar options file1 file2 ...
```

其中 options 包含的可选项如下:

表 2.1 jar 程序选项

选项	说明
c	创建一个新的或空的存档文件并加入文件。
C	临时改变目录。
e	在清单文件中创建一个入口点。
f	指定 JAR 文件名作为第二个命令行参数。
i	建立索引文件
m	将一个清单文件添加到 JAR 文件中
M	不为条目创建清单
t	显示内容表
u	更新一个已有的 JAR 文件
v	生成详细的输出结果
x	解压文件
0	存储但不进行 ZIP 压缩

3.2.2 清单文件

清单文件用于描述归档文件的特殊特性。清单文件被命名为 MANIFEST.MF，它位于 JAR 文件的一个特殊的 META-INF 子目录中。

复杂的清单文件可能包含更多条目。这些清单条目被分成多个节。第一节被称为主节。他作用于整个 JAR 文件。随后的条目用来指定命名实体的属性，如单个文件，包或 URL。它们必须以一个 Name 条目开始。节于节之间用空行分开。例如:

```
Manifest-Version: 1.0
lines describing this archive

Name:Woozle.class
lines describing this class
Name:com.mycompany.mypkg/
lines describing this package
```

要想编辑清单文件，需要将希望添加到清单文件中的行放到文本文件中，然后运行:

```
jar cfm jarFileName mainfestFileName...
```

要想更新一个已有的 JAR 文件的清单，则需要将添加的部分放置到一个文本文件中，然后执行以下命令:

```
jar ufm MyArchive.jar manifest-additions.mf
```

此外，清单文件的最后一行必须是一个换行符，否则将无法被正确读取。

3.2.3 可执行 JAR 文件

可以使用 jar 命令中的 e 选项指定程序的入口点:

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass files to add
```

或者在清单文件中指定程序的主类:

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

无论使用哪一种方法，用户可以简单地通过下面的命令来启动程序:

```
java -jar MyProgram.jar
```

在 Mac OS 平台中，可以直接通过双击 jar 文件执行程序，但在 Windows 平台中，Java 运行时安装程序将为".jar" 扩展名创建一个文件关联，会用 javaw -jar 命令启动文件。当然也可以通过第三方包安装工具将 jar 文件转换为 exe 文件执行。

3.3 文档注释

JDK 包含一个很有用的工具，叫做 javadoc，它可以由源文件生成一个 HTML 文档。这些文档的内容源于我们在程序中写的注释。

由于最终生成的是 HTML 文件，因此我们在注解中可以加入 html 标签。除了普通的文本，我们还可以写入自由格式文本，标记以 @ 开始，如 @author。下面将重点介绍这些自由格式文本。

3.3.1 通用注释

通用注释含有的标记如下:

- `@since text`
  始于条目，文本可一世引入这个特性的版本的任何描述。
- `@author name`
  作者条目，每个作者条目对应一个人名，可以有多个作者条目
- `@version text`
  版本条目

3.3.2 方法注释

方法中特有的标记如下:

- `@param variable description`
  用于描述参数。
- `@return description`

用于描述返回值。

- `@throws class description`

  用于描述可能抛出的异常。

3.3.3 包注释

要产生包注释，就需要在每一个包目录中添加一个单独的文件。可以有如下两个选择:

- 提供一个名为 package-info.java 的文件，在里面使用 /** */ 文档注释，后面是一个 package 语句。它不能包含更多的代码或注释。
- 提供一个名为 package.html 的文件。会抽取 body 标签内的所有文本。

3.3.4 注释抽取

如果需要要获取 javadoc 文件，则只需执行对应的 javadoc 指令即可。

4 继承

4.1 类继承基础

4.1.1 多态与强制类型转换

多态: 程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在实践中，多态表现为超类可以引用子类对象:

```
Person a = new Employee("Pionpill");
Person b = new Admin("Brandon");
Person c = new Staff("Chicken");
```

不能将超类的引用赋予给子类变量。如何判断多态，可以使用 “is-a” 规则。

此外，多态引用的对象仅能使用超类的方法，而不能使用子类的方法。例如上面 Person 对象 a 只能使用 Person 的方法，如果调用 Employee 独有的方法则会报错。

有时候可能需要将某个类的对象引用转换成另一个类的对象引用。进行强制类型转换的唯一原因是: 要在暂时忽视对象的实际类型之后使用对象的全部功能。

```
Manager boss = (Manager) staff;
```

一般情况下，我们只会在继承层次内进行强制类型转换。并且在将超类转换为子类之前，使用 `instanceof` 进行检查。

```
if (staff instanceof Manager) {
    boss = (Manager) staff;
}
```

虽然极少数情况下会用到强制类型转换，但并不推荐这样使用，如果一定要使用的话，建议先考虑重新设计继承树。

4.1.2 抽象类

假设有抽象基类 Person 的两个子类 Employee 和 Student。抽象基类的所有方法均为抽象方法，两个子类均实现了父类的全部抽象方法。

众所周知，抽象类无法创建实例，但是可以定义一个抽象类的对象变量，这个变量只能引用非抽象子类的对象:

```
Person a = new Employee("Pionpill");
```

下面代码是可行的:

```
var people = new Person[2];
people[0] = new Employee(...);
```

```
people[1] = new Student(...);
for (Person p : people)
    System.out.println(p.getName() + "," + p.getDescription());
```

我们知道，抽象基类的方法是没有定义的，因此也不可能被调用。但由于不能构造抽象类 Person 的对象，所以变量 p 永远不会引用 Person 对象，而是引用其子类 (具体的对象)，而这些对象中都实现了对应的方法。

与此同时，如果省略抽象基类的方法，仅在子类中实现诸如 getDescription() 方法，就不能在变量 p 上调用对应方法了，这是因为编译器只允许调用在类中声明的方法。

4.2 所有子类的超类: Object

当我们定义一个类时如果没有明确地指出它的超类，那么它的直接超类就是 Object 类。同时 Object 是除基本类型外所有类的最顶层超类。

4.2.1 Object 类型的变量

Object 类型的变量可以引用任何类型的对象。

```
Object obj = new Employee("Harry Hacker", 35000);
```

当然，Object 类型的变量只能用于作为各种值的一个泛型容器。要想对其中的内容进行具体的操作，还需要清楚对象的原始类型，并进行强制类型转换。

```
Employee e = (Employee) obj;
```

4.2.2 equals 方法

前文已经提及过 `equals` 方法，这里主要说明一些实现细节。

Java 语言规范要求 `equals` 方法具有下面的特征:

- 自反性: 任何非空引用，`x.equals(x)` 应该返回 true。
- 对称性: 任何非空引用，当且仅当 `x.equals(y)` 返回 true 时，`y.equals(x)` 返回 true。
- 传递性性: 任何三个非空引用，两两相等，那么三个引用应相等。
- 一致性: 任何非空引用，反符调用 `equals` 函数应返回相同的值。
- 任意非空引用，与 null 比较时，都应返回 false。

在实际实现 `equals` 方法的过程中，往往不可避免的使用到 `instanceof`，`getClass` 等操作，但是这些操作往往会违反上述的某些特性，即使是官方库的有些实现也陷入了这样的窘境。

对于上述问题，原书给出了如下建议:

1. 显示参数名为 otherObject。

2. 检查 this 和 otherObject 是否相等:

```
if (this == otherObject) return true;
```

这条语句只是一个优化，因为检查身份要比逐个比较字段开销小。

3. 检查 otherObject 是否为 null，如果是，则返回 false。

```
if(otherObject == null) return false;
```

4. 比较类，如果 `equals` 的语义可以在子类中改变，就使用 getClass 检测。

```
if(getClass() != otherObject.getClass()) return false;
```

如果所有子类都有相同的相等性语义，可以使用 `instanceof` 检测:

```
if(!(otherObject instanceof ClassName)) return false;
```

5. 将 otherObject 强制转换为相应类类型的变量:

```
CLassName other = (ClassName) otherObject;
```

6. 根据相等性概念的要求来比较字段。使用 == 比较基本类型的字段，使用 Object.equals 比较对象字段。

4.2.3 hashCode 方法

散列码是由对象导出的一个整型值。散列码是没有规律的，两个不同对象的散列码基本上不会相同。在 Object 类中定义了 `hashCode` 方法用于生成散列码。

下面是 String 类计算散列码的方法:

```
int hash = 0;
for(int i=0; i<length(); i++)
    hash = 31*hash + charAt(i);
```

在实现类的 `hashCode` 方法时，常常组合类中各个属性的散列码计算出实例的散列码。几乎所有的对象以及基本类型都有 `hashCode` 方法。部分对象和基本类型有静态 `hashCode` 方法。

```
public int hashCode() {
    return 7 * name.hashCode() + 11 * Double.hashCode(salary) + 13 * hireDay.hashCode();
}
```

还有个更好的做法，直接调用 `Objects.hash` 计算散列值:

```
public int hashCode() {
    return Objects.hash(name, salary, hireDay);
}
```

4.2.4 toString 方法

toString 方法返回表示对象值的一个字符串。绝大多数的返回值遵循这样的格式: 类名 + [字段]。比如下面这个例子:

```
public String toString() {
    return getClass().getName() + "[name=" + name + ",salary=" + salary + "]";
}
```

设计子类的程序员应该定义自己的 toString 方法。如果超类已经实现了 toString 方法，则子类只需要调用 super.toString() 就可以了。

```
public String toString() {
    return super.toString() + "[bonus" + bonus + "]";
}
```

toString 方法随处可见的主要原因如下:

- 在字符串的 “+” 操作中连接对象，编译器将自动调用对象的 toString 方法。

```
var p = new Point(10,20);
String message = "The current point is " + p;
```

- println() 函数的参数如果是对象，则会自动调用对应的 toString 方法。

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

> **警告 4.1.** 数组继承了 *Object* 类的 *toString* 方法，数字类型将使用一种古老的格式 *([I...)* 打印。补救的方法是调用静态方法 *Arrays.toString*。如果是多维数组，则需要调用 *Arrays.deepToString* 方法。

在实际项目中，强烈建议给自己定义的每一个类添加 toString 方法。

4.3 泛型数组列表

如果有 C++ 编程经历，一定会因为数组大小不可变而苦恼。即使 Java 允许在运行时确定数组大小，但是并不能动态更改数组。

解决这个问题的最简单方法是使用 Java 中的一个类，名为 ArrayList。在添加和删除元素时，它能够自动地调整数组容量。

ArrayList 是一个有参数类型的泛型类。为了指定数组列表保存的元素对象地类型。需要用一对尖括号括起来追加到 ArrayList 后面。

4.3.1 声明数组列表

声明一个保存 Employee 对象的数组列表的三种语法 (Java10):

```
ArrayList<Employee> staff = new ArrayList<Employee>();
var staff = new ArrayList<Employee>();
```

```
3 ArrayList<Employee> staff = new ArrayList<>();
```

上面第三种声明语法叫菱形语法，虽然在第二个菱形中没有指定对象名，但编译器会检查前面的变量名自动确认对象类型。

> **注意 4.1.** *在 Java 老版本中，会使用 Vector 类实现动态数组。不过，ArrayList 类更加高效，已经没有任何理由再使用 Vector。*

下面介绍几个常用的方法:

- `ArrayList<E>(int initialCapacity)`
  用指定容量构造一个空数组列表。
- `boolean add(E obj)`
  将元素添加到数组列表中。数组列表管理着一个内部的对象引用数组。
- `int size()`
  返回当前存储在数组列表中的元素个数。
- `void ensureCapacity(int capacity)`
  确定数组列表容量。
- `void trimToSize()`
  将数组列表的存储容量削减到当前大小。

4.3.2 访问数组列表元素

数组列表自动扩展容量的便利增加了访问元素语法的复杂程度。其原因是 `ArrayList` 类并不是 Java 程序设计语言的一部分，而是某个人编写并在标准库中提供的一个实用工具类。

- `E set(int index, E obj)`
  将值 obj 放置在数组列表的指定索引位置，返回之前的内容。
- `E get(int index)`
  得到指定索引位置存储的值。
- `void add(int index, E obj)`
  后移元素从而将 obj 插入到指定索引位置。
- `E remove(int index)`
  删除指定索引位置的元素，并将后面的元素前移。返回所删除的元素。

4.4 对象包装器与自动装箱

有时候需要将 `int` 这样的基本类型转换为对象。所有的基本类型都有一个与之对应的类，这些类通常被称为包装器: `Integer, Long, Float, Double, Short, Byte, Character, Boolean`。包装类是不可变的，也是 `final` 类，因此不能派生它们的子类。

假定要定义一个整型数组列表。尖括号中不允许是基本类型，就只能用到 `Integer` 包装器类。

```
var list = new ArrayList<Integer>();
```

幸运的是，ArrayList 提供了自动装箱与自动拆箱操作。使得我们可以直接这样使用:

```
list.add(3);          // 自动转换为: list.add(Integer.valueOf(3));
int n = list.get(i);  // 自动转换为: list.get(i).intValue();
```

自动装箱与自动拆箱适用于许多地方，这使得基本类型和它们的对象包装看起来是一样的。但它们有一个很大的不同: 同一性。== 运算符在基本类型中比较的是值是否相同，而在包装器对象中则是比较内存位置是否相同。因此，下面比较通常会失败。

```
Integer a = 1000;
Integer b = 1000;
a == b;    // false
```

不过上述比较结果也不往往是 false，如果经常出现的值包装到相同的对象中，这种比较就可能成功。这种不确定的结果并不是我们想要的，解决的办法是在比较两个包装器对象时调用 equals 方法。

由于包装器类引用可以为 null。所以自动装箱有可能会抛出一个 NullPointerException 异常。

4.5 参数数量可变的方法

我们知道，print 函数的参数个数没有限制。它是这样定义的:

```
public PrintStream printf(String fmt, Object... args) {
    return format(fmt, args);
}
```

这里的省略号... 是 Java 代码的一部分，它表明这个方法可以接收任意数量的对象实际上 print 方法接收两个参数，一个是格式字符串，另一个是 Object[] 数组，其中保存着所有其他参数 (如果调用者提供的是基本类型，会自动将它们装箱为对象)。

换句话说，对于 printf 的实现者来说，Object... 参数类型与 Object[] 完全一样。编译器会自动转换每个 print 调用，将参数绑定到数组中，并在必要的时候进行自动装箱。

下面是一个计算若干个数值中最大值的函数:

```
public static double max(double... values) {
    double largest = Double.NEGATIVE_INFINITY;
    for(double v:values)
        if (v>largest)
            largest = v;
    return largest;
}
double m = max(4.1, 40, -5);
```

编译器将 new double[] {4.1, 40, -5} 传递给 max 方法。

4.6 枚举类

下面举一个例子:

```
public enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE}
```

实际上，这个声明定义的类型是一个类，它刚好有 4 个实例，不可能构造新的对象。因此在比较枚举类型的值时，并不需要调用 `equals`，直接使用 “==” 就可以了。

如果需要的话，可以为枚举类型增加构造器，方法和字段。当然，构造器只是在构造枚举常量的时候调用。下面是一个示例:

```
public enum Size {
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
    private String abbreviation
    private Size(String abbreviation) {this.abbreviation = abbreviation;}
    public String getAbbreviation() {return abbreviation;}
}
```

枚举的构造器必须是私有的，可以省略 `private` 修饰符。所有枚举类型都是 `Enum` 类的子类。它们继承了这个类的许多方法。其中最有用的一个是 `toString`, 这个方法会返回枚举常量名。`toString` 的逆方法是静态方法 `ValueOf`:

```
Size s = Enum.valueOf(Size.class, "SMALL");
```

此外还有几个常用的方法:

- `int ordinal()`

  返回枚举常量在 `enum` 声明中的位置，位置从 0 开始计数。

- `int compareTo(E other)`

  和字符比较相同，返回负整数，0，正整数。

4.7 反射

反射库提供了一个丰富且精巧的工具集，可以用来编写能够动态操作 Java 代码的程序。能够分析类能力的程序成为反射。反射机制的功能极其强大。

反射是一种功能强大且复杂的机制。主要是开发工具的程序员对它感兴趣，一般应用程序员并不需要考虑反射机制。如果你不需要为其他 Java 程序员构建工具，可以跳过本章剩余部分。

跳过。

5 接口，lambda 表达式，内部类

5.1 接口

5.1.1 接口的概念

在 Java 程序设计语言中，接口不是类，而是对希望符合这个接口的类的一组需求。

下面我们看一个 `Comparable` 接口[1]。

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

这说明，任何实现该接口的类都必须包含 `compareTo` 方法，这个方法有一个泛型参数 `other`，并返回整数。

接口中的所有方法都是 `public`。因此在接口中声明方法时，不必提供关键字 `public`。

接口中不应该提供实例字段和实例方法 (新版 Java 已经可以提供简单的实例方法)。这个任务应该交给实现接口的类来完成。

为了让类实现一个接口，通常需要完成以下几步操作。

1. 将类声明为实现给定的接口。

```
class Employee implements Comparable<Employee>
```

2. 对接口中的所有方法提供定义。

```
class Employee implements Comparable<Employee> {
    public int compareTo(Employee other) {
        return Double.compare(this.salary, other.salary);
    }
}
```

> **警告 5.1.** *在接口申明中可以省略 `public`。但是在实现接口时，必须把方法声明为 `public`，否则编译器会报错。*

既然接口只有一个方法，那么我们为什么不直接实现该方法，而要多一步实现接口的操作呢? 这主要有两个原因:

- 一是在参数类型中我们可以书写接口类型，这样就能确保实参都实现了该接口。方便后续操作。
- 二是 Java 是一种工程化语言，往往具有复杂的继承关系，实现接口方便项目管理。

[1]在旧版 Java 中，使用的是 Object 而不是泛型类型。虽然泛型类型要在之后说明，但现在已经很少用 Object 进行强制类型转换，泛型已经成为绝大多数程序员的首选。

5.1.2 接口的属性

接口不是类，不能使用 new 运算符实例化一个接口。但能却能声明接口变量，接口变量必须引用实现了这个接口的类对象:

```
Comparable x = new Employee("Pionpill");
```

接口在许多方面表现的都与类一样，比如接口变量可以使用 instance 检查，接口可以继承接口。接口中的成员有以下特点:

- 成员方法: 声明时必须为 public，可以省略。实现时必须为 public，但不可以省略。
- 成员属性: 接口中不能包含实例字段,但是可以包含常量。接口中的字段默认为 public static final。

既然现代的接口已经可以实现某些简单的方法，那么抽象类与接口的差别在哪呢。设计接口的最主要原因是 Java 只允许类的单继承，而类在实现接口时可以多继承:

```
class Employee implements Cloneable, Comparable
```

5.1.3 静态和私有方法

在 Java8 中，允许在接口中添加静态方法。理论上讲，这没有任何语法错误，但是这有悖于接口的初衷。目前为止，通常的做法是将静态方法放在伴随类中。在标准库中，经常会有成对出现的接口和实用工具类，如 Collection/Collections。

随着 Java 的更新，在 Java11 标准库中已经将很多静态方法放在接口中，这样一来，这就没有必要再为实用工具方法另外提供一个伴随类。

在 Java9 中，接口中的方法是可以为 private 的。既可以修饰静态方法也可以是实例方法。但作用十分有限。

5.1.4 默认方法

可以为接口方法提供一个默认实现。必须用 default 修饰符标记这样一个方法。

```
public interface Comparable<T> {
    default int compareTo(T other) {
        return 0
    }
}
```

当然这并没有太大的用处，毕竟每一个实现接口的类都会覆盖里面的方法。

不过某些情况下，这会有些用:

- 接口演化问题: 如果随着接口更新，加入了新的方法，对应的实现类就会报错，这时候设置一个默认方法就可以避免这种错误。
- 默认方法可以调用其他方法，这可以简化一些操作。

既然接口可以多继承，那必然会导致同名方法冲突，幸运的是，Java 解决这类冲突的规则十分简单:

- 超类优先。如果超类与接口具有同名方法，那么采取超类中实现的具体方法。
- 接口冲突。如果两个接口提供了相同的默认方法，那么必须覆盖这个方法来解决冲突。

多继承带来的冲突远不止这些，但 Java 的解决思路大致如上，如果具体的实现类与接口冲突，则才需类中的方法，接口有没有提供默认方法并灭有什么区别。这正是 “类优先” 原则。

5.1.5 回调

回调 (callback) 简单来讲是指某一件事发生时进行调用。一般回调的是函数，在 Java 中可以传入回调对象，对象能携带更多的数据。

回调对象必须实现 `java.awt.event` 包的 `ActionListener` 接口:

```
public interface ActionListener {
    void actionPerformed(ActionEvent event);
}
```

5.1.6 `Comparator` 接口

顾名思义,`Comparator` 接口实现了元素的比较。例如 `String` 类实现了 `Comparable<String>`, `String.compareTo` 方法可以按字典顺序比较字符串。

加入我们需要按长度递增的顺序对字符串进行排序，若不是按字典顺序进行排序。则可以使用 `Arrays.sort` 方法的第二个版本。使用数组和比较器作为参数。比较器是实现了 `Comparator` 接口的类的实例。

```
public interface Comparator<T> {
    int compare(T first,T second);
}
```

要按长度比较字符串，可以如下定义一个实现 `Comparator<String>` 的类:

```
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

需要对数组进行排序时，则为 `Arrays.sort` 方法传入对应的对象:

```
String[] friends = {"Peter","Paul","Mike"}
Arrays.sort(friends, new LengthComparator());
```

在下面章节会了解到，用 lambda 表达式可以更方便地进行比较。

5.1.7 对象克隆

众所周知，克隆 (即拷贝) 可以分为浅拷贝与深拷贝，Java 中除了基本数据类型是传值，其他都是传引用，也即传引用时进行浅拷贝。因此，如果我们运行如下代码:

```
var ori = new Employee("Mike",5000);
Employee copy = ori;
copy.setSalary(10000);
```

那么源对象的薪资也会改变。因为浅拷贝两个引用在内存中指向同一块区域。

如果需要深拷贝，则需使用 clone 方法:

```
Employee copy = ori.clone();
copy.setSalary(10);
```

clone 是 Object 的 protected 方法，因此我们无法直接调用这个方法。只有 Employee 类可以克隆 Employee 对象。为什么要这样限制呢，假设我们克隆的字段中包含对其他对象的引用，那么我们没有实现彻底的深拷贝。因此 **clone** 方法需要我们手动改写。

对于每一个类，需要确定:

- 默认的 clone 方法是否满足需求。
- 是否可以在可变的子对象上调用 clone 来修补默认的 clone 方法。
- 是否不该使用 clone。

如果需要实现该方法，则类必须:

- 实现 Cloneable 接口;
- 重新定义 clone 方法，并指定 public 访问修饰符。

即使 clone 的默认实现能够满足要求，还是需要实现 Cloneable 接口，将 clone 定义为 public:

```
class Employee implements Cloneable {
    public Employee clone() throws CloneNotSupportedException{
        return (Employee) super.clone();
    }
}
```

> **警告 5.2.** *在实现 clone 方法的过程中，我们必须考虑这样一个问题，如果我们在父类 Employee 中实现了 clone 方法，且 Employee 派生出了子类 Manager，那么父类的 Clone 方法能很好地对子类进行深拷贝吗? 出于这个原因，Object 类中的 clone 方法被设置为了 protected，且标准库中仅有 5% 的方法实现了 clone 方法。*

5.2 lambda 表达式

5.2.1 lambda 表达式的语法

lambda 表达式可以看作一个函数，他的表现形式为: **参数 -> 表达式**。例如:

```
(String first, String second) -> first.length - second.length()
```

如果表达式过长，可以将代码放进 {} 中，并显示地包含 `return` 语句。

```
(String first, String second) -> {
    if (first.length() > second.length()) return -1;
    else if (first.length() < second.length()) return 1;
    else return 0;
}
```

还有以下注意点:

- 即使没有参数，也必须提供，这和无参数方法一样。
- 如果参数类型通过上下文可以推导出来，那么可以省略不写。
- 如果只有一个参数，那么可以省略小括号。
- 返回值类型无需指定，上下文会自动推导。

5.2.2 函数式接口

对于只有一个抽象方法的接口，需要这种接口的对象时，可以提供一个 lambda 表达式。这种接口称为函数式接口。

例如 `Array.sort` 方法，它的第二个参数需要一个 `Comparator` 实例，`Comparator` 就是只有一个方法的接口，所以可以提供一个 lambda 表达式:

```
Arrays.sort(words, (first,second) -> first.length() - second.length());
```

lambda 表达式可以转换为接口，这一点让 lambda 表达式很有吸引力。具体的语法很简短:

```
var timer = new Timer(1000, event ->
    {
        System.out.println("Hello, World!");
        Toolkit.getDefaultToolKit().beep();
    });
```

与实现了 ActionListener 接口的类相比，这段代码可读性要好很多。

5.2.3 方法引用

有时，lambda 表达式涉及一个方法:

```
var timer = new Timer(1000, event -> System.out.println(event));
```

如果直接把 `println` 方法传递到 `Timer` 构造器就更好了:

```
var timer = new Timer(1000, System.out::println);
```

表达式 System.out.println 是一个方法引用 (method reference)，它指示编译器生成一个函数式接口的实例，覆盖这个接口的抽象方法来调用给定的方法。在上面的例子中，会生成一个 `ActionListener`,它的 `actionPerformed(ActionEvent e)` 方法要调用 `System.out.println(e)`。

再比如:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

需要使用:: 运算符分隔方法名与对象或类名，主要有三种情况:

1. *object::instanceMethod*
2. *class::instanceMethod*
3. *class::staticMethod*

第一种情况下,方法引用等价于向方法传递参数的 lambda 表达式。对于 `System.out::println`,对象是 `System.out`，所以方法表达式等价于 `x -> System.out.println(x)`。

第二种情况下,第一个参数会成为方法的隐式参数。例如,`String::compareToIgnoreCase` 等同于 `(x,y) -> x.compareToIgnoreCase(y)`。

第三种情况下,所有参数都传递到静态方法,`Math::pow` 等价于 `(x,y) -> Math.pow(x,y)`。

只有当 lambda 表达式的体只调用一个方法而不做其他操作时，才能把 lambda 表达式重写为方法引用:

```
s -> s.length() == 0
```

这里有一个方法调用。但是还有一个比较，所以这里不能使用方法引用。

此外，在方法引用中还可以使用到 `this` 与 `super` 关键字。

5.2.4 构造器引用

构造器引用与方法引用很类似，只不过方法名为 `new`。例如 `Person::new` 是 `Person` 构造器的一个引用。

5.2.5 变量作用域

假如我们要写下面这样的代码:

```
public static void repeatMessage(String text, int delay) {
    ActionListener listener = event -> {
        System.out.println(text);
        Toolkit.getDefaultToolKit().beep();
    };
    new Timer(delay, listener).start();
}
```

将 lambda 表达式看作函数，我们就写出了一个闭包。而使用闭包必然会涉及到很多变量问题[2]。为了避免不必要的内存泄漏等玛法，我们在 lambda 表达式中需要遵守如下规定:

- 在 lambda 表达式中只使用不会改变的变量。

5.2.6 处理 lambda 表达式

假设我们需要重复一个动作 n 次，将这个动作和重复次数传递到一个 repeat 方法:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

要接受这个 lambda 表达式，需要选择一个函数式接口，这里我们使用了 `Runnable` 接口(含有一个 `run` 抽象方法):

```
public static void repeat(int n,Runnable action) {
    for (int i = 0; i < n; i++) action.run();
}
```

调用 `action.run()` 时会执行 lambda 表达式的函数体。

5.3 内部类

内部类 (inner class) 是定义在另一个类中的类。这样的好处有:

- 内部类可以对同一个包中的其他类隐藏。
- 内部类方法可以访问定义这个类的作用域中的数据，包括原本私有的数据。

内部类原先对于间接地实现回调非常重要，不过如今 lambda 表达式在这方面做的更好。

5.3.1 使用内部类访问对象状态

这里使用一个不太实用的例子进行说明:

```
public class TalkingClock {
    public static void main(String[] args) {
        TalkingClock clock = new TalkingClock(1000,true);
        clock.start();
        JOptionPane.showMessageDialog(null, "Quit?");
        System.exit(0);
    }
}

class TalkingClock {
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) {
```

[2]见 https://blog.csdn.net/Hunt_bo/article/details/107699137

```
        this.interval = interval;
        this.beep = beep;
    }

    public void start() {
        ActionListener listener = new TimePrinter();
        Timer timer = new Timer(interval, listener);
        timer.start();
    }

    public class TimePrinter implements ActionListener{
        @Override
        public void actionPerformed(ActionEvent e) {
            System.out.println("At the stone, the time is " +
                Instant.ofEpochMilli(e.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }
}
```

这里的 TimePrinter 位于 TalkingClock 类内部。这并不意味着每个 TalkingClock 有一个 TimePrinter 实例字段。如前所示，TimePrinter 对象是由 TalkingClock 类的方法构造的。

TimePrinter 类没有实例字段或者名为 beep 的变量。实际上，beep 指示 TalkingClock 对象中创建这个 TimePrinter 的字段。

一个内部方法可以访问自身的数据字段，也可以访问创建它的外围类对象的数据字段。为此，内部类的对象总有一个隐士 i 你用，指向创建它的外部对象。这个引用在内部类的定义中是不可见的。我们称之为 outer。下面代码等价:

```
if (beep) Toolkit.getDefaultToolKit().beep();
if (outer.beep) Toolkit.getDefaultToolKit().beep();
```

因为 TimePrinter 类没有定义构造器，所以编辑器为它生成了一个无参数构造器:

```
public TimePrinter(TalkingClock clock) {
    outer = clock
}
```

而在构造 TimePrinter 对象时，编译器就会将当前语音时钟的 this 引用传递给这个构造器。

```
ActionListener listener = new TimePrinter(this);
```

关于内部类，还有几个注意点:

- 内部类中声明的所有静态字段都必须是 final，并初始化为一个编译时常量。
- 内部类不能有 static 方法。

注意 5.1. 在编译过程中，内部类会使用 $ 转换为普通的类，保留第一个对外部类的引用。

5.3.2 局部内部类

仔细查看 `TalkingClock` 代码会发现，类型 `TimePrinter` 的名字只出现了一次，也即只被使用了一次。遇到这种情况，可以在一个方法中局部地定义这个类。

```
public void start(int interval, boolean beep) {
    class TimePrinter implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("XXX");
            if (beep) Toolkit.getDefaultToolKit().beep();
        }
    }
    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.start();
}
```

局部类的作用域被限定在块中，不可以有访问说明符。局部类的优势是对块外部的世界完全隐藏，可以访问块中的局部变量。

5.3.3 匿名内部类

使用局部内部类时，通常还可以再进一步。假如只想创建这个类的一个对象，甚至不需要为类指定名字。

```
public void start(int interval, boolean beep) {
    var listener = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.out.println("XXX");
                if (beep) Toolkit.getDefaultToolKit().beep();
            }
        };
    var timer = new Timer(interval, listener);
    timer.start();
}
```

它的含义是: 创建一个类的新对象，这个类实现了 `ActionListener` 接口，需要实现的方法在括号内定义。一般语法如下:

```
new SuperType(construction parameters) {
    inner class methods and data
}
```

其中，`SuperType` 可以是接口。由于匿名内部类没有名字，也就不能提供构造器。

5.3.4 静态内部类

有时候，使用内部类只是为了把一个类隐藏在另一个类的内部，并不需要内部类有外部类对象的引用。为此，可以把内部类声明为 static，这样就不会生成那个引用。

III 核心特性

6 异常，断言和日志

6.1 错误处理

6.1.1 异常分类

在 Java 程序设计语言中，异常对象都是派生于 `Throwable` 类的一个实例对象。

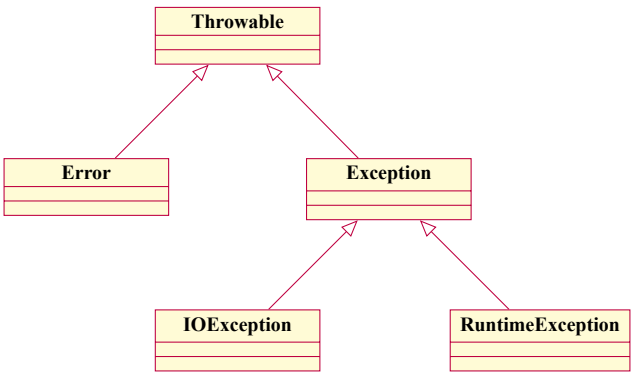

图 6.1 Java 中的异常层次结构

`Exception` 派生的两个分支需要重点关注，一般的规则是：由编程错误导致的异常属于 `RuntimeException`；如果程序本省没问题，但由于 I/O 错误这类问题导致的异常属于其他异常。

Java 语言规范将派生于 `Error` 类或 `RuntimeException` 类的所有异常称为非检查型 (unchecked) 异常，所有其他异常称为检查型 (checked) 异常。编译器将检查你是否为所有的检查型异常提供了异常处理器。

6.1.2 声明检查型异常

如果遇到了无法处理的情况，Java 方法可以抛出一个异常。方法不仅需要告诉编译器将要返回什么值，还要告诉编译器可能发生什么错误。比如在读取文件时，可能文件不存在，或者文件内容为空。

要在方法的首部指出这个方法可能抛出一个异常:

```
1 public FileInputStream(String name) throws FileNotFoundException
```

如果发生了糟糕的情况，构造器将不会重新初始化一个对象，而是抛出对应的错误，并搜索如何处理错误。

不需要声明 Java 的内部错误，即从 Error 继承的异常。任何程序代码都有可能抛出那些异常，比如数组下标问题。

6.1.3 创建异常类

如果有必要，我们可以定义一个派生自 `Exception` 或其子类的异常类。自定义的这个类应该包含两个构造器，一个默认构造器，一个包含详细描述信息的构造器 (超类 `Throwable` 的 `toString` 方法会返回一个字符串，其中包含这个详细信息。)。

```
class FileFormatException extends IOException {
    public FileFormatException() {}
    public FileFormatException(String gripe) {
        super(gripe);
    }
}
```

6.2 捕获异常

6.2.1 捕获异常

如果发生了某个异常，但没有在任何地方捕获这个异常，程序就会终止。想要捕获一个异常，就需要 `try/catch` 语句。异常语句很好理解，不再赘述，下面给出基本语法:

```
try {
    code
}
catch (FileNotFoundException | UnknownHostException e) {
    XXX
}
catch (IOException e) {
    XXX
}
finally {
    XXX
}
```

介绍一个特殊的方法 `initCause` 这个方法是对异常来进行包装的。目的就是为了出了问题的时候追根究底。举个例子，在底层中有一个异常 A 被捕获到后进行了处理，这导致上层抛出了异常 B，虽然此时捕获到了 B 异常，但是只能说明 B 出现了异常，而不知道是什么导致了 B 异常。

如果此时我们使用 `initCause` 方法对异常进行包装，我们就可以通过 `getCause` 方法获得原始的 A 异常。

```
class A {
    try {
        ...
```

```
    } catch(AException a) {
        BException b = new BEexception();
        b.initCause(a);
        throw b;
    }
}
...
class B {
    try {
        ...
    } catch(BException b) {
        b.getCause(); //得到导致B异常的原始异常
    }
}
```

6.2.2 try-with-Resources 语句

try-with-Resources 语句最简单的形式为:

```
try (Resource res = ...) {
    work with res
}
```

try 块退出时，会自动调用 res.close()。下面给出一个典型的例子:

```
try (var in = new Scanner (
    new FileInputStream("/usr/share/dict/words"), StandardCharsets.UTF_8))
{
    while (in.hasNext())
        System.out.println(in.next());
}
```

这个块正常退出时，或者存在异常时，都会自动调用 in.close() 方法，就好像使用了 finally 块一样。

6.2.3 处理异常的技巧

1. 异常不能代替测试: 捕获并抛出异常消耗的资源远高于简单的测试。
2. 不要将语句分装在多个 try 语句块中，这会导致代码块激增。
3. 充分利用异常层次结构，派生合适的异常子类。
4. 不要压制异常，即使异常出现的概率很低。
5. 检查错误时，“苛刻” 要比放任更好。比起返回 null，直接报错往往更好。
6. 不要羞于传递异常，如果不是必要，传递比自己捕获往往更合适。

6.3 使用断言

6.3.1 断言的概念

断言机制允许在代码测试期间向代码中插入一些检查，而在生产代码中会自动删除这些检查。Java 语言引入了关键字 assert。

```
assert condition;
assert condition:expression;
```

其中，表达式将传给 `AssertionError` 对象，以便以后显示。

需要重申的是，断言仅在测试阶段起作用。而且断言往往是针对致命性错误。因此在程序设计中不可以过于依赖断言。

6.3.2 启用很禁用断言

默认情况下，断言是禁用的。可以在运行程序时，使用 `-enableassertions -ea` 选择启用断言。

也可以指定某个类或某个包启用断言:

```
java -ea:MyClass -ea:com.company.lib MyApp
```

也可以使用 `-disableassertions -da` 选择禁用断言。

6.4 日志

最简单的 debug 方式莫过于使用 `System.out.print` 观察程序行为，但是在我们解决完问题后需要删除这些语句，如果出问题了，还需要新增语句。日志 API 可以很好的解决这个繁琐的过程，该 API 优点如下:

- 可以很容易取消全部日志记录，或者仅仅取消某个级别以下的日志，而且很容易打开日志。
- 日志的代码开销很小，灵活度很高。
- 日志记录可以被定向到控制台，文件等等。
- 日志系统有单独的控制器，可以被格式化。

6.4.1 使用日志

简单的日志记录，可以使用全局日志记录器并调用其 `info` 方法:

```
Logger.getGlobal().info("File->Open menu item selected");
```

在合适的地方可以通过调用如下函数取消所有日志:

```
Logger.getGlobal().setLevel(Level.OFF);
```

可以调用 getLogger 方法创建或获取日志记录器:

```
private static final Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

> **注意 6.1.** 未被任何变量引用的日志记录器可能会被垃圾回收。为了防止这种情况发生，要像上面的例子中一样，用静态变量存储日志记录器的引用。

通常有 7 个日志级别: SERVER, WARNING, INFO, CONFIG, FINE, FINER, FINEST

默认情况下，实际只记录前三个级别，也可以设置一个不同的级别:

```
logger.setLevel(level.FINE);
```

这样,FINE 以及所有更高级别的日志都会被记录。此外还可以通过 LEVEL.ALL LEVEL.OFF 开关所有级别的日志记录。或者指定级别:

```
logger.log(level.FINE, message);
```

记录日志的最常见用途是记录那些预料之外的异常。可以使用下面两个便利方法在日志记录中包含异常的描述:

```
void throwing(String className, String methodName, Throwable t);
void log(Level l, String message, Throwable t);
```

Java 的日志还包含过滤器，处理器等等功能。本人觉得脱了项目谈这些过于抽象，这里省略，请读者自行查阅资料。

7 泛型程序设计

7.1 类型参数的好处

在 Java 中增加泛型类之前，泛型程序设计都是通过继承实现的。`ArrayList` 类只维护一个 `Object` 引用的数组:

```
public class ArrayList {
    private Object[] elementData;
    ...
    public Object get(int i) {...}
    public void add(Object o) {...}
}
```

这种方法存在两个问题: 当取值时必须进行强制类型转换；没有错误检查，可以将任何类型的值加入到数组中。

泛型提供了一个更好的解决方案: *类型参数* (type parameter)。`ArrayList` 类有一个类型参数用来指示元素的类型:

```
var files = new ArrayList<String>();
```

这样的好处如下:

- 代码具有更好的可读性。
- 编译器会进行类型检查。

使用泛型编程有多容易，编写泛型类就有多困难。因为编写者需要预判各种各样的类作为类型参数。

7.2 定义简单泛型

7.2.1 泛型类

泛型类就是有一个或多个类型变量的类:

```
public class Pair<T> {
    private T first;
    private T second;

    public Pair() {first = null; second = null;}
    public Pair(T first, T second) {this.first = first; this.second = second;}

    public T getFirst() {return first;}
    public T getSecond() {return second;}

    public void setFirst(T newValue) {first = newValue;}
    public void setSecond(T newValue) {second = newValue;}
}
```

类似的，可以定义多个字段类型:

```
public class Pair<T,U> {...}
```

> **注意 7.1.** *Java 库使用变量 E 表示集合的元素类型，K/V 分别便是键和值。T 表示任意类型。*

7.2.2 泛型方法

在泛型方法中，需要把参数类型放在修饰符后面，返回值类型前面。定义一个带参数类型的方法:

```
public static <T> T getMiddle(T... a) {...}
```

在调用泛型方法时，可以明确参数类型，也可以不写参数类型，因为编译器会自动推导出参数类型。

```
String middle = ArrayAlg.<String>getMiddle("John","Q.");
String middle = ArrayAlg.getMiddle("John","Q.");
```

7.2.3 类型变量的限定

有时，类或方法需要对类型变量加以约束，比如我们希望类型 T 实现了 `compareTo` 方法，可以这样写:

```
public static <T extends Comparable> T min(T[] a) ...
```

如果需要多个限制条件，可以这样写:

```
T extends Comparable & Serializable
```

7.3 泛型代码与虚拟机

虚拟机没有泛型类型对象，所有对象都属于普通类。

7.3.1 类型擦除

无论如何定义一个泛型类型，都会自动提供一个相应的原始类型。这个原始类型的名字就是去掉类型参数后的泛型类型名。类型变量会被擦除，并替换为限定类型 (或者，无限定变量被替换为 Object)。

例如:

```
public class Pair {
    private Object first;
    private Object second;
```

```
4
5       public Pair() {first = null; second = null;}
6       public Pair(Object first, Object second) {this.first = first; this.second = second;}
7
8       public Object getFirst() {return first;}
9       public Object getSecond() {return second;}
10
11      public void setFirst(Object newValue) {first = newValue;}
12      public void setSecond(Object newValue) {second = newValue;}
13  }
```

因为 T 没有被限定，所以被替换为了 `Object`。如果有限定，比如 `Comparable`。那么类型将不是 `Object` 而是 `Comparable`。

7.3.2 转换泛型表达式

编写一个泛型方法调用时，如果擦除了返回类型，编译器会插入强制类型转换。例如:

```
1  Pair<Employee> buddies = ...;
2  Employee buddy = buddies.getFirst();
```

`getFirst` 擦除类型后的返回类型是 `Object`。编译器自动插入转换到 `Employee` 的强制类型转换。

7.3.3 转换类型方法

在类型擦除的过程中，会遇到很多复杂的问题。比如，我在代码中使用了 `super` 调用父类的方法，而类型擦除后，我们无法知道这个类型的父类是谁，也即无法保持多态。

Java 虚拟机会使用名为桥方法的方式重写或者引用等方式保持多态。总而言之，虚拟机帮你避免类型擦除与多态发生冲突。

对于 Java 泛型，我们需要记住以下几个事实:

- 虚拟机中没有泛型，只有普通的类和方法。
- 所有类型参数都会被替代。
- 会合成桥方法保持多态。
- 为了保持安全，必要时插入强制类型转换。

7.4 限制与局限性

不能用基本类型实例化类型参数

基本类型不能作为类型参数，应该使用对应的包装器，如 `double -> Double`。这是因为 `Object` 无法存储 `double` 值。

运行时类型查询只适用于原始类型

虚拟机中的对象总有一个特定的非泛型类型。因此，所有类型查询只产生原始类型:

```
if (a instanceof Pair<String>) //Error
if (stringPair.getClass() == employeePair.getClass()) // equal
```

不允许创建参数化类型的数组

例如:

```
var table = new Pair<String>[10];
```

擦除之后，`table` 类型是 `Pair[]`，转换为 `Object[]`。

数组会记住它的元素类型，如果试图存储其他类型的元素，会抛出错误。

尽管可以通过数组存储的检查，但仍会导致一个类型错误。因此不允许创建参数化类型的数组。

Varargs 警告

varargs 拆开来: var args。即多参数警告。具体为向参数个数可变的方法传递泛型类型的实例。

在传递参数个数可变的实例时，实际上传的就是参数数组。这就违背了之前的规则，但这种情况下，我们只会得到一个警告，而不是错误。

可以使用添加注解 `@SupportWarnings("unchecked")` 的方式消除警告。Java7 中可以使用 `@SafeVarargs`。

不能实例化类型变量

考虑这样一个代码:

```
public Pair() {first = new T(); second = new T();} // Error
```

由于 `T()` 会被转换为 `Object()` 所以上述代码肯定会出问题。

同样的也不能实例化泛型数组。

其他缺陷

还有一些不是很常见的缺陷:

- 不能在静态字段或方法中引用类型变量。
- 不能抛出或捕获泛型类的实例。
- 可以取消对检查型异常的检查。

- 注意擦除后的冲突。

7.5 泛型类型的继承规则

只需要记住，无论 S 与 T 有什么继承关系，Pair<S> 与 Pair<T> 都没有任何关系。也即泛型类型之间没有任何关系。

7.6 通配符类型

7.6.1 通配符上下界

在通配符类型中，允许类型参数发生变化。

```
Pair <? extends Employee>
```

表示任何泛型 `Pair` 类型，它的类型参数是 `Employee` 的子类，如 `Pair<Manager>`。

其中? 代表类型不限制，`extends` 关键字则制定了类型的上界。类似的，还有 `super` 关键字，限制为所有的超类型。

7.6.2 无限定通配符

无限定通配符就是 <?> 代表不确定类型[1]。

[1]参考文献:https://blog.csdn.net/yu_duan_hun/article/details/123975211

8 集合

8.1 Java 集合框架

8.1.1 集合接口与实现分离

> **警告 8.1.** 这一节讲了很多数据结构与算法，这不是 *Java* 原有的特性，只是用 *Java* 语言进行了实现，故省去了很多。

与现代的数据结构类库的常见做法一样，Java 集合类库也将接口与是西安分离。例如队列接口:

```
public interface Queue<E> {
    void add(E element);
    E remove();
    int Size();
}
```

我们可以使用循环数组或者链表实现队列。但使用者并不关心队列的具体实现方法。

8.1.2 `Collection` 接口

在 Java 类库中，集合类的基本接口是 `Collection` 接口。

```
public interface Collection<E> {
    boolean add(E element);
    Iterator<E> iterator();
    ...
}
```

其中，`add` 方法用于向集合中添加元素。如果添加元素确实改变了集合就返回 `true`; 如果集合没有发生变化就返回 `false`。

`iterator` 方法用于返回一个实现了 `Iterator` 接口的对象。可以使用这个迭代器对象依次访问集合中的元素。

8.1.3 迭代器

`Iterator` 接口包含 4 个方法:

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
    void remove();
    default void forEachRemaining(C)
}
```

通过反复调用 `next` 方法，可以逐个访问集合中的每个元素。但是，如果到达了集合的末尾，`next` 方法将抛出一个 `NoSuchElementException`。因此在调用之前要用 `hasNext` 进行判

断。如果迭代器对象还有多个可以访问的元素，这个方法就返回 `true`。如果想要查看集合中的所有元素，就请求一个迭代器，当 `hasNext` 返回 true 时就反复地调用 `next` 方法。

编译器简单地将 “for each” 循环转换为带有迭代器的循环，可以处理任何实现了 `Iterable` 接口的对象，这个接口只包含一个抽象方法:

```
public interface Iterable<E> {
    Iterator<E> iterator();
    ...
}
```

8.1.4 集合框架中的接口

集合有两个基本接口: `Collection` 和 `Map`。我们可以直接插入元素:

```
boolean add(E element)
```

由于映射包含键值对，更常见的，使用如下方法:

```
V put(K key, V value)
V get(K key)
```

8.2 具体集合

8.2.1 链表与数组列表

链表结构比较复杂，Java 中的链表都是双向链表。我们不能对链表 (`LinkedList`) 进行一边读取一边写入的操作，这往往会导致严重的错误。我们需要遵守一个规则: 可以根据需要为一个集合关联多个迭代器，前提是这些迭代器只能读取集合。或者，可以关联一个能同时读写的迭代器。

数组列表相对简单，但是如果对数组列表进行增减操作，会消耗大量的时间。

还有一种快速查询元素的集合叫散列表，我已经在别的笔记写过很多次，这里不写了。此外还有树结构，队列。都是数据结构的内容，这里省略。

8.3 映射

映射用来存放键/值对。如果提供了键就能查找到值。例如，可以存储一个员工记录表，其中键为员工 ID，值为 `Employee` 对象。

8.3.1 基本映射操作

Java 类库为映射提供了两个通用的实现: `HashMap` 和 `TreeMap`。这两个类都实现了 `Map` 接口。散列映射与树映射的差别类似于散列表与树的差别。

映射的基础内容没什么好讲的，操作类似于 Python 中的字典。

8.3.2 弱散列映射

设计 `WeakHashMap` 类是为了解决这样一个有趣的问题。如果有一个值，它对应的键已经不再在程序中的任何地方使用。将会出现什么情况? 假定对某个键的最后一个引用已经消失，那么不再有任何途径可以引用这个值的对象。但是，由于程序中的任何部分不会再有这个键，所以无法从映射中删除这个键值对。

为什么垃圾回收不能删除它呢? 垃圾回收器会跟踪活动的对象。只要映射对象是活动的，其中的所有桶也是活动的，它们就不能被回收。因此，需要由程序负责从长期存活的映射表中删除那些无用的值。或者使用 `WeakHashMap`，当对键的唯一引用来自散列表映射条目时，这个数据结构将被回收。

8.3.3 链接散列集与映射

`LinkedHashSet` 和 `LinkedHashMap` 类会记住插入元素项的顺序。这样就可以避免散列表中的项看起来是随机的。在表中插入元素时，就会并入到双向链表中。

8.3.4 标识散列映射

类 `IdentityHashMap` 有特殊的用途。在这个类中，键的散列值不是用 `hashCode` 函数计算的，而是用 `System.identityHashCode` 方法计算的。这是 `Object.hashCode` 根据对象的内存地址计算散列码时所用的方法。而且，在对两个对象进行比较时，`IdentityHashMap` 类使用 `==`，而不使用 `equals`。

也就是说，不同的键对象即使内容相同，也被视为不同的对象。

9 并发

9.1 什么是线程

进程是资源分配的基本单位，是程序执行的实例。程序运行时系统会创建进程并为它分配资源，然后把该进程放入进程就绪队列，进程调度器选中它的时候就会为它分配 CPU 时间，程序开始真正运行。

线程是一条执行路径，是程序执行时的最小单位，它是进程的一个执行流，是 CPU 调度和分派的基本单位，一个进程可以由很多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。线程由 CPU 独立调度执行，在多 CPU 环境下就允许多个线程同时运行。同样多线程也可以实现并发操作，每个请求分配一个线程来处理。

一个正在运行的软件 (如迅雷) 就是一个进程，一个进程可以同时运行多个任务 (迅雷软件可以同时下载多个文件，每个下载任务就是一个线程), 可以简单的认为进程是线程的集合。

我们需要知道的是，Java 中使用线程可以并发运行，提高程序效率。可以通过如下方式启动一个线程:

```
Thread thread = new Thread(Runnable target) // 构造线程，并调用目标的 run() 方法。
thread.start() // 启用线程，调用 run 方法。这个方法会立即返回。新线程并发运行。
```

9.2 线程状态

线程有如下六种状态:

- New: 新建
- Runnable: 可运行
- Blocked: 阻塞
- Waiting: 等待
- Timed waiting: 计时等待
- Terminated: 终止

要确定一个线程的当前状态，只需要调用 `getState` 方法。

9.2.1 新建线程

当用 `new` 操作符创建新线程时，线程还没有开始运行。他的状态是 *New*。在线程运行之前还有一些基础工作要做。

9.2.2 可运行线程

一旦调用了 `start` 方法，线程就处于可运行状态。可运行的线程可能正在运行也可能不在运行。要由操作系统为线程提供具体的运行时间。(Java 没有将正在运行的线程作为一个单

独的状态。仍然叫可运行状态。)

线程开始运行后，不一定始终保持运行，运行中的线程有时需要暂停，让其他线程有机会，这取决于操作系统的调用方式。抢占式操作系统给每一个可运行的线程一个时间片来执行任务。抢占式操作系统的具体操作请查阅相关资料。

虽然所有的桌面和服务器操作系统都使用抢占式调度。但是，像手机这样的小型设备可能使用协作式调度。在这样的设备中，一个线程只有在调用 yield 方法或者被阻塞或等待时才会失去控制权。

使用下面方法，可以让正在执行的线程向另一个线程交出运行权:

```
static void yield()
```

9.2.3 阻塞和等待线程

当线程处于阻塞或等待状态时，它暂时是不活动的。不运行任何代码，而且消耗最少的资源。要由线程调度器激活这个线程。

- 当一个线程试图获取一个内部的对象锁，而这个锁被其他线程占用，该线程会被阻塞。当所有线程都释放了这个锁，并且线程调度器允许该线程保持这个锁时，它将变成非阻塞状态。
- 当线程等待另一个线程通知调度器出现一个条件时，这个线程会进入等待状态。事实上阻塞和等待状态并没有太大的差别。
- 有几个方法有超时参数，调用这些方法会让线程进入计时等待状态。这一状态将一直保持到超时期满或者接收到适当的通知。

9.2.4 终止线程

线程会由于以下两个原因之一而终止:

- `run` 方法正常退出，线程自然终止。
- 因为一个没有捕获的异常终止了 `run` 方法，使线程意外终止。

9.3 线程属性

9.3.1 中断线程

当线程的 `run` 方法执行方法体中的最后一条语句后再执行 `return` 语句返回时，或者出现了方法中没有捕获的异常时，线程将终止。除了已经废除的 `stop` 方法，没有办法可以强制线程终止。不过，`interrupt` 方法可以用来请求终止一个线程。

当对一个线程调用 `interrupt` 方法时，就会设置线程的中断状态。这是每个线程都有的 `boolean` 标志。每个线程都应该不时地检查这个标志，以判断线程是否被中断。

要想得出是否设置了中断状态，首先调用静态的 `Thread.currentThread` 方法获得当前线程，然后调用 `isInterrupted` 方法:

```
while (!Thread.currentThread().isInterrupted() && ...)
```

但是，如果线程被阻塞，就无法检查中断状态。这里就要引入 `InterruptedException` 异常。

没有任何语言要求被中断的线程应当终止。中断一个线程只是要引起它的注意。被中断的线程可以决定如何相应中断。某些线程非常重要，所以应该处理这个异常，然后再继续执行。但更普遍的是，线程只需要将中断解释为一个终止请求。这种线程的 `run` 方法具有如下形式:

```
Runnable r = () -> {
    try {
        ...
        while (!Thread.currentThread().isInterrupted() && ...) {
            do work...
        }
    } catch (InterruptedException e) {
        // thread was interrupted during sleep or wait
    } finally {
        // cleanup if required
    }
};
```

如果再每次工作迭代之后都调用 `sleep` 方法，`isInterrupted` 检查既没有必要也没有用处。如果设置了中断状态，此时倘若调用 `sleep` 方法，他不会休眠。实际上，它会清除中断状态并抛出 `InterruptedException`。因此，如果循环调用了 `sleep`，不要检测中断状态，而应该捕获 `InterruptException` 异常。

```
Runnable r = () -> {
    try {
        ...
        while (...) {
            ...
            Thread.sleep(delay);
        }
    } catch (InterruptedException e) {
        // thread was interrupted during sleep
    } finally {
        // cleanup if required
    }
};
```

9.3.2 守护线程

可以通过

```
t.setDaemon(true);
```

将一个线程转换为守护线程，这样一个线程并没有什么魔力。守护线程的唯一用途是为其他线程提供服务。当只剩下守护线程时，虚拟机就会退出，因为如果只剩下守护线程，就没必要继续运行程序了。

9.3.3 线程名

默认情况下，线程名为: Thread-2。可以用 setName 方法为线程设置任何名字:

```
var t = new Thread(runnable);
t.setName("Web thread");
```

9.3.4 未捕获异常的处理器

线程的 run 方法不能抛出任何检查型异常，但是，非检查型异常可能会导致线程终止。在这种情况下，线程会死亡。

不过，对于可以传播的异常，并没有任何 catch 子句。实际上，在线程死亡之前，异常会传递到一个用于处理未捕获异常的处理器。

这个处理器必须属于一个实现了 Thread.UncaughtExceptionHandler 接口的类。这个接口只有一个方法:

```
void uncaughtException(Thread t,Throwable e)
```

可以用 setUncaughtExceptionHandler 方法为任何线程安装一个处理器。也可以用 Thread 类的静态方法 setDefaultUncaughtExceptionHandler 为所有线程安装一个默认的处理器。替代处理器可以使用日志 API 将未捕获异常的报告发送到一个日志文件。

9.3.5 线程优先级

一般情况下，一个线程会继承构造它的那个线程的优先级。可以用 setPriority 方法提高或降低任何一个线程的优先级。可以设置为:

- MIN_PRIORITY: 1
- MAX_PRIORITY: 10
- NORM_PRIORITY: 5

线程的优先级相当依赖于操作系统，Windows 有 7 个优先级，Java 会对优先级进行映射。有些 Linux 系统则只有一个优先级。

9.4 同步

在大多数实际的多线程应用中，两个或两个以上的线程需要共享对同一数据的存取。如果两个线程存取同一个对象，并且每个线程分别调用了一个修改该对象状态的方法，会导致

对象被破坏，这种情况通常称为竞态条件。

为了避免多线程破坏共享数据，必须学习如何同步存取。

9.4.1 静态条件详解

当两个线程同时对公共数据进行存取时，公共数据可能会被破坏。例如多个线程执行以下操作:

```
accounts[to] += amount;
```

问题的关键在于原子操作，这个指令可能如下处理:

- 将 `accounts[to]` 加载到寄存器。
- 增加 `amounts`。
- 将结果写回 `accounts[to]`。

假定第一个线程执行前两个步骤，然后它的运行权被抢占。再假设第 2 个线程被唤醒，更新 `accounts` 数组中的同一个元素。然后，第一个线程被幻想并完成其第 3 步。这个动作会抹去第 2 个线程所做的更新，就会出错。

9.4.2 锁对象

有两种机制可防止并发访问代码块。Java 语言提供了一个 `synchronized` 关键字来达到这一目的。Java5 引入了 `ReentrantLock` 类。

用 `ReentrantLock` 保护代码块的基本结构如下:

```
myLock.lock();
try {
    // critical section
} finally {
    myLock.unlock();
}
```

这个结构确保任何时刻只有一个线程进入临界区。一旦一个线程锁定了锁对象，其他任何线程都无法通过 `lock` 语句。当其他线程调用 `lock` 时，它们会暂停，知道第一个线程释放这个锁对象。

> **警告 9.1.** 必须要将 *unlock* 操作包括在 *finally* 子句中，这一点至关重要。如果在临界区的代码抛出一个异常，锁必须释放。否则，其他线程将永远阻塞。使用锁时，就不能使用 *try-with-resources* 语句。他会在首部声明一个新变量。但是如果使用一个锁，可能想使用多个线程共享那个变量。

理论上，给不同的线程加锁，每个线程会维护不同的锁对象，两个线程都不会阻塞。本应该如此，因为线程在操作不同的实例时，线程之间不会相互影响。

我们使用的锁称为重入 (reentrant) 锁，因为线程可以反复获得已拥有的锁。锁有一个持有计数来跟踪对 `lock` 方法的嵌套调用。线程每一次调用 `lock` 后都要调用 `unlock` 来释放锁。

由于这个特性，被一个锁保护的代码可以调用另一个使用相同锁的方法。

9.4.3 条件对象

通常，线程进入临界区后却发现只有满足了某个条件后再能执行。可以使用一个条件对象来管理那些已经进入了一个锁却不能做有用工作的线程。

现在我们考虑这样一种情况:

```
if (bank.getBalance(from) >= amount)
    bank.transfer(from, to, amount);
```

如果我们在判断语句之后，被其他线程占用，随后的操作会产生错误的结果，因此，这里需要加锁:

```
public void transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
        while(accounts[from] < amount) {
            //wait
        }
        // transfer funds
    } finally {
        bankLock.unlock();
    }
}
```

如果账户中没有足够的资金，线程会等待其他线程向账户中添加资金。现在这个线程获得了 bankLock 的排他性访问，因此别的线程没有几乎进行，会一直等待下去。

一个锁对象可以有一个或多个相关联的条件对象。可以用 newCondition 方法获得一个条件对象。

```
class Bank {
    private Condition sufficientFunds;
    ...
    public Bank() {
        ...
        sufficientFunds = bankLock.newCondition();
    }
}
```

如果 transfer 方法发现资金不足，它会调用:

```
sufficientFunds.await();
```

当前线程现在暂停，并放弃锁。这就允许另一个线程执行。等待获得锁的线程和已经调用了 await 方法的线程存在本质上的不同。一旦一个线程调用了 await 方法，它就进入了这个条件的 等待集。当锁可用时，该线程并不会变为可运行状态。实际上，它人人保持非活动状态，直到另一个线程在同一条件上调用 signalAll 方法。

这个调用会重新激活等待这个条件的所有线程。当这些线程从等待集中移出时，它们再次成为可运行的线程，调度器最终将再次将它们激活。同时，它们会尝试重新进入该对象。一旦锁可用，它们中的某个线程将从 `await` 调用返回，得到这个锁，并从之前暂停的地方继续执行。

此时，线程应该再次测试条件。不能保证现在一定满足条件，`signalAll` 方法仅仅是通知等待的线程: 现在有可能满足条件，值得再次检查条件。

最终需要有某个其他线程调用 `signalAll` 方法，这一点至关重要。当一个线程调用 `await` 时，他没有办法重新自行激活。它寄希望于其他线程。如果没有其他线程来重新激活等待的线程，它就永远不再运行了。这将导致 *死锁现象*。

9.4.4 synchronized 关键字

`Lock` 和 `Condition` 接口允许程序员充分控制锁定。但大多数情况下可以使用 Java 语言内置的一种机制。从 1.0 版本开始，Java 中每个对象都有一个内部锁。若果一个方法声明有 `synchronized` 关键字，那么对象的锁将保护整个方法。也就是说，要调用这个方法，线程必须获得内部对象锁。

下面代码等价:

```
public synchronized void method () {
    // method body
}

public void method() {
    this.intrinsicLock.lock();
    try {
        // method body
    } finally {
        this.intrinsicLock.unlock();
    }
}
```

内部对象锁只有一个关联条件。`wait` 方法将一个线程增加到等待集中，`notifyAll/notify` 方法可以接触等待线程的阻塞。换句话说，调用 `wait` 或 `notifyAll` 等价于:

```
intrinsicCondition.await();
intrinsicCondition.signalAll();
```

> **注意 9.1.** *`wait`, `notifyAll`, `notify` 是 `Object` 类的 `final` 方法。`Condition` 方法必须命名为 `await`, `signalAll`, `signal`，从而不会与那些方法发生冲突。*

关于线程锁的做法:

- 优先使用 `java.util.concurrent` 包中的某种机制，它会处理所有的锁。
- 其次使用 `synchronized` 关键字。
- 最后考虑 `Lock/Condition` 结构提供的额外能力。

9.4.5 同步块

每一个 Java 对象都有一个锁。线程可以通过调用同步方法获得锁。还有另一种机制可以获得锁: 进入一个同步块。

```
synchronized (obj) {
    // critical section
}
```

它会获得 obj 的锁，书上讲的比较抽象，建议查看这篇文章: https://blog.csdn.net/csdnnews/article/details/82321777。

9.4.6 线程局部变量

线程在共享变量时存在一定的风险。有时可能要避免共享变量，使用 ThreadLocal 辅助类可以为各个线程提供各自的实例。

9.5 任务和线程池

构造一个新的线程开销有些大，因为这涉及到与操作系统的交互。如果程序中创建了大量生命期很短的线程，那么不应该吧每个任务映射到一个单独的线程，而应该使用线程池。线程池中包含许多准备运行的线程。为线程池提供一个 Runnable 就会有一个线程调用 run 方法。当 run 方法退出时，这个线程不会死亡，而是留在池中准备为下一个请求提供服务。

9.5.1 Callable 与 Future

Runnable 封装一个异步运行的任务，可以把它想象成一个没有参数和返回值的异步方法。Callable 与 Runnable 类似，但是有返回值。Callable 接口是一个参数化的类型，只有一个方法 call。

```
public interface Callable<V> {
    V call() throws Exception;
}
```

类型参数是返回值的类型。例如，Callable<Integer> 表示一个最终返回 Integer 对象的异步计算。

Future 保存异步计算的结果。可以启动一个计算，将 Future 对象交给某个线程，然后忘掉它。这个 Future 对象的所有者在结果计算好之后就可以获得结果。