第一章 绪论

第二章 基本数据类型、数组、枚举

标识符

命名规则：

- 字母、下划线_、美元符号$，阿拉伯数字，长度不限
- 第一个字符非数字
- 不能是关键字、true、false、null
- 区分大小写

基本数据类型

逻辑类型：boolean

可取值：true、false

```
boolean condition = true, x = false;
```

整数类型：byte、short、int、long

int：4个字节，范围$-2^{31} \sim 2^{31}-1$

```
int dec = 114; // 十进制
int bin = 0b1110010; // 二进制使用0b前缀
int oct = 0162; // 八进制使用0前缀
int dex = 0x72; // 十六进制使用0x前缀
```

byte：1个字节，范围$-2^{7} \sim 2^{7}-1$

```
byte x = -12;
```

short：2个字节，范围$-2^{15} \sim 2^{15}-1$

```
short x = 12;
```

long：8个字节，范围$-2^{63} \sim 2^{63}-1$，需要加后缀L

```
long width = 12L;
```

若不加L，编译器会默认把一串数字认为是int类型，若其超过int的表示范围就会报错

字符类型：char

java使用unicode编码，兼容ascii码。 char类型分配2个字节，范围$0 \sim 65525$，在内存中存储的是字符对应的unicode（utf）码

```
char ch1 = 'A', ch2 = '国', ch3 = '\\', ch4 = 'の';
```

可以用int显式转换char类型以查看字符对应的unicode码：

```
public class Main {
    public static void main(String[] args) {
        char ch1 = 'ω';
        int ch2 = 32831;
        System.out.println("\""+ch1+"\""+"的utf码是"+(int)ch1);
        System.out.println("utf码为"+ch2+"的字符是"+"\""+(char)ch2+"\"");
    }
}
```

浮点类型：float、double

float：4个字节，保留8位有效数字，需要加后缀F或f

```
float x = 22.76f, y = 1e-12F;
```

double：8个字节，保留16位有效数字，后缀D或d可以省略

```
double x = 1e12D, y = 12.4444444;
```

基本数据类型的转换

数据类型精度的排序（从低到高）：

```
byte short char int long float double
```

低级别的值赋给高级别的变量时会自动转换位高级别的类型，成为“隐式转换”即“自动类型提升”，如：

```
int a = 10;
long b = a;
```

高等级的值不允许直接赋值给低等级的变量，因为可能导致数据丢失或数值不精确。若要强制赋值需要进行“显式转换”即“强制类型转换”，如：

```
long a = 10;
int b = (int)a;
```

需要注意，若高等级的值超过了低等级变量能表示的范围则会导致数值溢出或截断，如long类型的3000000000转换为int会变为-1294967296，double类型的123456789.123456789转换为float会变为1.23456792E8。

常见的一个用法是将double或float强制转换为int来取其整数部分。

从命令行输入输出数据

输入基本类型数据

可以使用`java.util`包中的Scanner类来从命令行输入基本类型数据。

使用以下方法从缓冲区中读取从命令行输入的基本类型数据：

```
nextInt(), nextFloat(), nextDouble(),
next(),  // 字符串，跳过空白（空格、制表符、换行符），到空白结束
nextLine(),  // 一行文本，到换行符结束
nextShort(), nextLong(), nextByte(), nextBoolean()
```

使用以下方法判断缓冲区中的下一个token是否符合某种类型（has方法不会从缓冲区中删除token）：

```
hasNext(), // 缓冲区是否还有下一个标记，返回值为true或false
hasNextInt(), hasNextFloat(), hasNextDouble(), hasNextShort(), hasNextLong(),
hasNextByte(), hasNextBoolean(), hasNextLine() // 检查缓冲区中下个token是否为指定类型，返回值为true或false
```

示例：

```
import java.util.Scanner; // 导入Scanner类

public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // 创建Scanner对象来读取输入

        // 检查是否有下一个整数并读取
        System.out.print("请输入一个整数：");
        if (scanner.hasNextInt()) {  // 使用 hasNextInt() 进行检查
            int intValue = scanner.nextInt();
            System.out.println("你输入的整数是：" + intValue);
        } else {
            System.out.println("输入的不是一个有效的整数！");
            scanner.next(); // 清除无效输入
        }

        // 检查是否有下一个
        System.out.print("请输入一个字符串：");
        if (scanner.hasNext()) {  // 使用 hasNext() 进行检查
            String stringValue = scanner.next();
            System.out.println("你输入的字符串是：" + stringValue);
        } else {
            System.out.println("没有输入有效的字符串！");
        }

        // 读取一整行文本
        scanner.nextLine(); // 清除缓冲区
        System.out.print("请输入一整行文本：");
        String line = scanner.nextLine();
        System.out.println("你输入的文本是：" + line);

        scanner.close(); // 关闭Scanner以释放资源
    }
}
```

输出基本类型数据

使用`System.out.println()`或者`System.out.print()`输出表达式或串的值，前者输出后自动换行，后者不自动换行。

在括号中，可以使用并置符号`+`将多个表达式或值合并起来输出，如：

```
System.out.println("你输入的文本是："+"\""+input+"\"");
```

若要输出的字符串较长需要拆分为几部分输出，可以使用并置符号加回车实现，如：

```
System.out.println("这是第一部分"+
                   "这是第二部分"+
                   "这是第三部分");
```

也可以使用c语言中printf函数类似的格式控制输出，可用的格式控制符号如下：

格式控制符号	输出类型
%d	int
%f	float，保留6位小数
%c	char
%s	string
%md	int占m列
%m.nf	float占m列，小数保留n位

示例：

```
System.out.printf("你输入的数字是%d", input_int);
```

数组

java中的数组需要先声明后创建，也可在声明的同时创建。

数组的声明

声明数组的格式如下：

```
// 一维数组
类型[] 数组名;
类型 数组名[];

// 二维数组
类型[][] 数组名;
类型[] 数组名[];
```

java采用"数组的数组"来声明多维数组，如数组`[3][4]`是由3个长度为4的数组组成。

java在声明数组时不允许指定数组的大小，数组的大小只能在创建的时候指定。

数组的创建

数组的声明只是指定了数组的名字和元素类型，要向其中存放数据还要为其分配内存空间，即创建数组。

创建数组的格式如下：

```
数组名 = new 元素类型[元素个数];
```

与c语言不同，java允许使用int类型变量来指定数组的长度，但是无法改变已创建的数组的长度。

数组在声明和创建时的元素类型必须相同，否则会报错。

数组的声明和创建同时完成

```
int num[] = new int[5];
// 等号左边：声明一个数组，名字叫num，储存int类型的值
// 等号右边：为这个数组分配5个int类型元素的空间
```

数组的初始化和赋值

数组在创建时可以直接为其赋初值，元素的数量可以代替指定的数组长度，但是 **java不允许在数组声明和初始化时混用指定大小和直接初始化列表** ，如：

```
int[] num = new int[]{1,2,3};

// 错误示范：
// int num[] = new int[3]{1, 2, 3}; // 不能同时指定大小和初始化列表
```

也可以使用初始化列表直接赋值，省略new关键字，简化代码，如：

```
int[] num = {1, 2, 3}; // 数组的长度是3
```

也可以逐个为数组元素赋值，如：

```
int[] num = new int[5];
num[0] = 1;
num[1] = 10;
// 没赋值的部分为默认值0

//二维数组的初始化
int a[][] = {{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1}};
```

使用var类型简化变量的创建

在声明和创建数组时，指定两次数组元素的类型虽然安全但是十分繁琐，因此可以使用var类型，让编译器自动判断变量的类型，简化代码，如：

```
int num[] = new int[5];
// 可以等价替换为
var num = new int[5];
// 再使用初始化列表直接赋值
var num2 = new int[]{1,2,3};

int a[][] = {{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1}};
// 可以等价替换为
var a = new int[][]{{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1}};

// 同理，其他类型也可以使用var
var input1 = 123; // int
var input2 = 3.14; // double
var input3 = 'A';  //char
var input4 = "你好"; // String
```

数组在使用var时不可以省略new和类型[长度]。

使用var类型必须立即对其初始化，否则无法判断其类型。

var的类型推断由赋值表达式的类型决定，使用时应确保表达式的类型是明确的。

var类型仅可用于局部变量，不可用于成员变量、方法参数或返回类型。

数组元素的使用和length的使用

java数组元素的索引从0开始，若访问的索引超过数组长度虽然可以编译通过，但是运行时会报错。

使用`数组名.length`可以返回数组的长度，如var num = new int[5]，则new.length的值为5。

数组的复制

数组的复制分为浅拷贝和深拷贝。浅拷贝只是复制了值的引用，两个变量都指向了同一个内存中的数组，复制后对两个变量的操作是共享的；后者将值又复制了一份（占用双倍的内存空间），复制后对两个变量的操作是独立的。

数组复制的前置要求是两个数组的类型必须相同。

浅拷贝

java的数组是引用型变量，即数组变量存储的是数组对象在内存中的地址，而不是数组元素本身。数组的浅拷贝可以让两个变量指向同一个数组，如：

```
var num1 = new int[]{1,2,3};
var num2 = new int[]{4,5};
num2 = num1;
// 系统将释放分配给{4,5}的内存（因为没有任何变量指向{4,5}），同时让num2指向{1,2,3}
// 由于num1和num2都指向{1,2,3}，对num1或num2做任何修改都会影响另一个变量所指的数组
```

深拷贝

深拷贝将数组中的每一个元素都复制一份，创建了一个新的数组对象。这个新数组与原数组独立，占用新的内存空间，对新数组的修改不会影响原数组。
深拷贝需要手动实现，通过遍历数组元素将其逐一复制到新数组中，这意味着两个数组的长度必须相等。

```
// 第一种方法，手动拷贝
var num1 = new int[]{1, 2, 3};
var num2 = new int[num1.length];
for (int i = 0; i < num1.length; i++) {
    num2[i] = num1[i];
}

// 第二种方法，使用Arrays.copyOf方法
import java.util.Arrays;
var num3 = new int[]{1, 2, 3};
// 使用 Arrays.copyOf 进行深拷贝
var num4 = Arrays.copyOf(num3, num3.length);
// 第一个参数是原数组，第二个参数是新的数组长度，若大于原数组长度会用0或false或null填充，若小于原数组长度会截断

// 第三种方法，使用System.arraycopy方法
int[] src = {1, 2, 3, 4, 5};
int[] dest = new int[7]; // 目标数组有更大的空间
System.arraycopy(src, 1, dest, 2, 3);
// arraycopy的参数为：原数组，原数组索引起始位置，目标数组，目标数组起始位置，复制的元素数量
```

数组的字符串输出

使用Arrays类中的toString方法将数组转换为字符串输出，如：

```
import java.util.Arrays;

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

```
        int[] numbers = {10, 20, 30, 40, 50};
        String[] names = {"Alice", "Bob", "Charlie"};

        System.out.println(Arrays.toString(numbers));
        // 输出: [10, 20, 30, 40, 50]
        System.out.println(Arrays.toString(names));
        // 输出: [Alice, Bob, Charlie]
    }
}
```

数组的排序和二分法查找

使用Arrays类中的sort方法对数组排序，如：

```
import java.util.Arrays;

public class SortExample {
    public static void main(String[] args) {
        int[] numbers = {5, 2, 8, 3, 1};
        Arrays.sort(numbers);
        // 此时numbers为{1, 2, 3, 5, 8}
        System.out.println(Arrays.toString(numbers));
    }
}
```

sort方法支持仅对数组中某个部分进行排序，如：

```
int[] arr = {5, 2, 8, 3, 1};
Arrays.sort(arr, 1, 4); // 对索引1到3的元素排序
// 输出: 5, 2, 3, 8, 1
```

使用Arrays类中的binarySearch方法对 **已排序** 的数组进行二分搜索，能在$O(\log n)$的时间复杂度内完成元素的查找，返回目标元素的索引，若查找失败则返回负值，其绝对值为插入的位置，如：

```
import java.util.Arrays;

public class BinarySearchExample {
    public static void main(String[] args) {
        int[] numbers = {1, 3, 5, 7, 9, 11};
        int index = Arrays.binarySearch(numbers, 7);
        System.out.println("元素 7 的索引: " + index); // 输出: 元素 7 的索引: 3

        index = Arrays.binarySearch(numbers, 4);
        System.out.println("元素 4 的索引: " + index); // 输出: 元素 4 的索引: -3
```

```
    }
}
```

枚举

java的枚举类型用于定义一组常量，如四季、一周的七天，如：

```
// 定义枚举类型 Day
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumExample {
    public static void main(String[] args) {
        // 赋值
        Day today = Day.MONDAY;  // 直接赋值为枚举常量
        System.out.println("今天是: " + today);
        // 今天是: MONDAY

        // 比较
        if (today == Day.MONDAY) {
            System.out.println("星期一开始工作! ");
            // 输出: 星期一开始工作!
        }

        // 使用name()方法输出枚举类型变量的值
        System.out.println("今天是: " + today.name());
        // 输出: 今天是: MONDAY

        // 使用ordinal()方法输出某个常量的索引值，从0开始
        System.out.println("今天是第 " + (today.ordinal() + 1) + " 天"); // 输出:
今天是第 1 天
    }
}
```

第三章 运算符、表达式、语句

运算符和表达式

算术运算符和算术表达式

- 加减乘除 + - * /
- 取余 %

- 自增自减 ++ --，x++表示先使用x后自增，++x表示先自增后使用

算数混合运算的精度

数据类型精度的排序（从低到高）：

```
byte short char int long float double
```

精度计算规则（从上到下）：

- byte，short，char类型参与计算时先转化为int
- 若有double则按double计算
- 若有float则按float计算
- 若最高精度为long则按long计算
- 若最高精度低于int则按int计算
- 不允许把超过byte的int赋值给byte

```
byte x = 97 + 1, y = 1;
// 这个不会报错，因为编译器可以直接计算出x是98，y是1，不超过byte的表示范围

byte z = x + y;
// 这个会报错，因为byte类型在参与计算时会先转为int，x+y得到的int结果不能自动转为byte类型，因为可能导致数据溢
```

关系运算符和关系表达式

运算返回值为boolean型的true或false。

运算符	含义
>	大于
<	小于
>=	大于等于
<=	小于等于
==	等于
!=	不等于

逻辑运算符和逻辑表达式

运算符	含义
&&	与
||	或
!	非

对“与”和“或”运算，存在“短路原则”，若根据前一个表达式的值已经可以判断出整个表达式的真假，则不再计算后边表达式的值。

赋值运算符和赋值表达式

- 赋值 =

```
int a = b = 100;
// 会报错，因为编译器不知道b的类型

int c;
int d = c =100;
// 可以，因为c和d的类型都是明确的，他两个都会被赋值为100

int m = 100;
int n = 100;
// 最好这么写，代码易读
```

位运算符

java的位运算用于对整数类型的位进行直接操作，性能高效，适用于性能和资源有限的环境。

位运算符	含义	解释
&	按位与	都为1则为1
|	按位或	其中一个为1则为1
^	按位异或	不同则为1
~	按位取反	取反
<<	左移	位左移指定的位数，右侧用0填充 相当于乘2
>>	右移	位右移指定的位数，右侧用符号位填充（正0负1） 相当于除2
>>>	无符号右移	右移指定的位数，左侧用0填充

举例：

- a = 0b0101，b = 0b0011
- a & b = 0b0001
- a | b = 0b0111
- a ^ b = 0b0110
- ~a = 0b1010, ~b = 0b1100

设 c = -24 = 0b 1110 1000，对其作左右移的位运算：

左移2位 c << 2

> 0b 1010 0000
> 左边两个1被丢弃，右边补两个0

右移2位 c >> 2

> 0b 1111 1010 右边两个0被丢弃，左边补符号位1（负数）

无符号右移3位 c >>> 3

> 0b 0001 1101 右边3位被丢弃，左边补0

instanceof 运算符

`instanceof`用于判断对象是否是某个类的实例，用来检查对象是否是指定类或其子类、实现类的实例，返回值为boolean类型的true或false。

语法：

```
object instanceof ClassName
```

- 若 object 是 ClassName 类型或其子类、实现类的实例，则返回 true。
- 若 object 不是 ClassName 类型，则返回 false。

运算符的优先级

优先级由高到低排序：

运算符类型	运算符
括号	()
一元运算符	++ -- + -(正负) ! ~
类型转换	(type)
乘除、取余	* / %
加减	+ -

运算符类型	运算符
移位	<< >> >>>
关系	> < >= <= instanceof
相等性	== !=
按位与	&
按位异或	^
按位或	|
逻辑与	&&
逻辑或	||
三元运算符	?:
赋值	= += -+ *= /=

语句概述

java中的语句分为：

- 方法调用语句 `System.out.println("hello");`
- 表达式语句 `x = 23`
- 复合语句

```
{
    x = 123;
    System.out.println("%d", x);
}
```

- 空语句（只有一个分号）
- 控制语句（条件分支、开关、循环）
- package和import语句

条件分支语句 if

if - else if - else：

```
if (表达式) {
    语句
}
else if (表达式){
    语句
}
```

```
else {
    语句
}
```

开关语句 switch

`switch`是单条件多分支的开关语句，语法如下：

```
switch (表达式) {
    case 常量1: {
        语句
        break; // 可选
    }
    case 常量2: {
        语句
    }
    ...
    default: {
        //可选
        语句
    }
}
```

switch会先计算表达式的值，若其与某个case的常量值相等，则从这个case的语句开始依次执行，直到遇到break

switch语句适合与枚举类型搭配使用，如：

```
public class main {
    // 定义枚举类型
    enum color {
        red, green, blue
    }
    public static void main(String[] args) {
        for (color a: color.values()){
            for (color b: color.values()){
                for (color c: color.values()){
                    if(a != b && a != c && b != c){
                        // 输出所有可能的颜色组合
                        System.out.println("a=" + a + ", b=" + b + ", c=" + c);
                    }
                }
            }
        }

    }
}
```

输出：

```
a=red, b=green, c=blue
a=red, b=blue, c=green
a=green, b=red, c=blue
a=green, b=blue, c=red
a=blue, b=red, c=green
a=blue, b=green, c=red
```

循环语句

for循环

```
for (初始化; 条件; 每次循环最后执行){
    语句
}
```

在JDK1.5中新增了“增强型for循环”或“for-each循环”，用于简化对数组或集合的遍历。语法如下：

```
for(声明循环变量: 数组名字){
    语句
}
```

比如：

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

for (int number : numbers) {
    System.out.println(number);
}
```

for循环创建了一个int变量`number`，每次循环都会将数组`numbers`中的下一个元素赋值给`number`，直接体现了遍历的对象，简化了代码，有效防止数组越界，但是仅适用于顺序遍历的情况

while循环

```
while (表达式){
    语句
```

```
}
```

do-while循环

```
do {
    语句
} while (表达式)
```

先执行一次循环体内的程序，然后根据表达式的值决定是否重复执行

break、continue

break直接跳出当前一层的循环，continue让当前循环直接进行下一次循环

第四章 类与对象

类

类，class，是java程序的基本要素。一个java程序由若干个类组成。

```
class 类名 {
    类体
}
```

类体的组成：

- 变量的声明：储存对象的属性
- 方法的定义：对类中的属性进行操作

类的变量

类的变量分为两种：

- 成员变量：在变量声明部分声明的变量，在整个类中都有效
  - 用`static`修饰的成员变量称为类变量（静态变量）
  - 否则为实例变量
- 方法变量：在方法部分声明的变量，在整个方法中有效

> 静态变量是类级别的，只在类被第一次创建对象时初始化，对这个类的所有对象都可见且保持一致，在任意一个对象中修改静态变量都会同步到其他对象中，在程序终止时才被释放。适用于定义常量，如

> MAX_VALUE。

若方法中局部变量的名字与成员变量相同，则成员变量会被隐藏。若要引用成员变量需要加this关键字。

```
class Square {
    // 变量声明->成员变量
    double length;

    // 方法定义
    void EditLength {
        length = scanner.nextDouble();
    }

    double CalcArea {
        // 方法变量
        double area = length*length;
        return area;
    }
}
```

注意，声明类的成员变量时可以直接为其赋初值，但是不可以声明成员变量后再为其赋值，因为赋值操作只能出现在方法中，如：

```
class example {
    int a = 1;
    // 这样是可以的

    int b;
    b = 1;
    // 这样不可以
}
```

类的方法

```
方法声明部分 {
    方法体的内容
}
```

基本的方法声明包括方法名和方法的返回类型，可以是任意java的数据类型或void（不返回数据）。

方法的签名

方法的 **签名** 是方法名、参数列表（类型和顺序）、组合，用于唯一标识一个方法，区分同一个类或不同类中的方法，以确定具体调用哪个方法。

签名不包括方法的返回类型、修饰符、参数名、方法体。

方法的重载 Overload

java允许在一个类中定义多个方法，他们的方法名相同但是 **参数列表不同**，根据传入参数的不同执行不同的方法，称为 **重载**

重载的返回类型可以不同，但是不用于区分重载

```
class MathOperations {
    int add(int a, int b) {
        return a + b;
    }

    // 重载的add方法，参数列表不同
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        System.out.println(math.add(5, 3));
        // 输出: 8
        System.out.println(math.add(5.5, 3.2));
        // 输出: 8.7
    }
}
```

构造方法

构造方法是创建对象时初始化对象的特殊方法，在创建对象时由new关键字自动调用。
构造方法可以用于设置对象的初始状态，执行必要的初始化操作。
构造方法在一个对象被创建时被调用，它 **与类同名**，没有返回类型，可以被重载。

```
class Person {
    // 成员变量
    String name;
    int age;

    // 无参构造方法
    public Person() {
        this.name = "Unknown";
        this.age = 0;
```

```
    }

    // 有参构造方法
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // 打印信息方法
    void PrintInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        // 使用无参构造方法创建对象
        Person person1 = new Person();
        // 使用new关键字时自动运行构造方法
        person1.PrintInfo();  // 输出: Name: Unknown, Age: 0

        // 使用有参构造方法创建对象
        Person person2 = new Person("Alice", 30);
        person2.PrintInfo();  // 输出: Name: Alice, Age: 30
    }
}
```

类方法和实例方法

类方法是用`static`修饰的方法，有以下特性：

- 类方法是类级别的，属于类本身，而不是类的某个实例
- 可以直接通过`ClassName.ClassMethod`调用而不需要实例化
- 不能访问非静态的成员变量和实例方法，只能访问静态变量和静态方法

实例方法没有`static`修饰，有以下特性：

- 实例方法是属于对象的方法，需要通过创建类的实例来调用，如`ObjectName.InstanceMethods`
- 实例方法可以访问实例变量和实例方法，也可以访问类变量和类方法

```
class Calculator {
    // 类方法：不依赖于具体的对象
    public static int add(int a, int b) {
        return a + b;
    }

    // 实例方法：依赖于具体的对象
    public int multiply(int a, int b) {
```

```
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        // 调用类方法：直接通过类名调用，不需要创建对象
        int sum = Calculator.add(5, 10);
        System.out.println("Sum: " + sum);  // 输出：Sum: 15

        // 创建Calculator对象
        Calculator calculator = new Calculator();

        // 调用实例方法：需要通过对象调用
        int product = calculator.multiply(5, 10);
        System.out.println("Product: " + product);  // 输出：Product: 50
    }
}
```

实例方法能对类变量和实例变量操作，但是类方法只能操作类变量

实例方法可以调用类中的其他方法，类方法只能调用类中的类方法

对象

在java中，用类来声明变量，然后用类来给出这个类的一个实例，也就是创建一个对象

构造方法

当程序创建对象时需要使用类的构造方法

若没有编写构造方法，则编译器会默认这个类只有一个构造方法，他没有参数，方法体中没有语句

创建对象

创建对象前需要先声明对象：`类名 对象名;`
此时没有为这个对象分配空间

然后使用new运算符为对象分配变量，即创建对象：`对象名 = new 类名(参数)`
实例化对象后会为成员变量分配空间并赋初值，然后根据成员变量的地址计算出一个称为“引用”的值作为`new 类名(参数)`的运算结果，赋值给对象名

分配给对象的变量习惯性称为对象的实体

使用`类名 对象名 = new 类名(参数);`同时声明并创建变量：

```
class Person {
    // 成员变量
    String name;
```

```
    int age;

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

public class Main {
    public static void main(String[] args) {
        Person person2 = new Person("Alice", 30);
    }
}
```

若两个对象有相同的引用，则二者有相同的实体，如：

```
class Point {
    int x, y;
    Point (int x, int y){
        this.x = x;
        this.y = y;
    }
    void PrintPos(){
        System.out.printf("x = %d, y = %d\n", this.x, this.y);
    }
}

public class Main{
    public static void main(String[] args){
        Point p1 = new Point(1, 2);
        Point p2 = new Point(3, 5);
        p1.PrintPos();
        p2.PrintPos();

        p1 = p2;
        // 此后对p1、p2任意一者的更改都会同步到对方身上，因为p1和p2都指向了同一个对象
        p1.PrintPos();
        p2.PrintPos();
    }
}
```

输出：

```
x = 1, y = 2
x = 3, y = 5
x = 3, y = 5
x = 3, y = 5
```

当程序发现内存中的某个实体不被任何一个对象引用，则会被自动释放内存，称为“垃圾收集机制”。比如上述程序中，p1指向的对象被p2覆盖，没有任何一个对象指向这个实体，那他就会被释放内存。

使用对象

使用点运算符/引用运算符/访问运算符：.来访问对象的变量或方法

```
class example {
    int num1;
    double num2;

    public example(int num1, double num2){
        this.num1 = num1;
        this.num2 = num2;
    }

    void print2(){
        System.out.printf("num2 = %f\n", num2);
    }
}

public class Main {
    public static void main(String[] args) {
        example exam = new example(10, 3.14);
        System.out.printf("num1 = %d\n", exam.num1);
        exam.print2();
    }
}
```

输出：

```
num1 = 10
num2 = 3.140000
```

可变参数

可变参数允许方法接收多个相同类型的参数而无需明确指定参数的数量，语法如下，其中...为省略号语法

```
public void methodName(Type... parameterName) {
    // 方法体
}
```

也可以明确前几个参数的类型和数量，最后输入不确定数量的参数，如：

```
public static int sum(double sum, double... numbers){
    // 方法体
```

```
}
```

实例：

```
public class VarargsExample {
    public static int sum(int... numbers) {
        int total = 0;
        for (int number : numbers) {
            total += number;
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println(sum(1, 2, 3));
        // 输出：6
        System.out.println(sum(10, 20));
        // 输出：30
        System.out.println(sum(5, 5, 5, 5, 5));
        // 输出：25
    }
}
```

编译器实际上为所有传入的输入创建了一个数组，也就是说可以使用数组的方式来访问这些参数，如：

```
public class Main {
    // 定义一个接受可变参数的 sum 方法
    public static int sum(int... numbers) {
        int total = 0;
        for (int i = 0; i < numbers.length; i++) {
            total += numbers[i]; // 通过索引访问数组元素
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println(sum(1, 2, 3));
        // 输出：6
        System.out.println(sum(10, 20));
        // 输出：30
        System.out.println(sum(5, 5, 5, 5, 5));
        // 输出：25
    }
}
```

对象的组合

java中"对象的组合"允许一个类包含其他类的对象作为其成员变量，表示某个类是由其他类组成的，即一个对象拥有另一个对象的实例。

示例：

```
class Address {
    private String city;

    public Address(String city) {
        this.city = city;
    }

    public String getCity() {
        return city;
    }
}

class Person {
    private String name;
    private Address address; // Person 包含 Address 对象

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
        // 这里保存的是address对象的引用而不是副本，因此后续直接修改address对象时会同步到person中
    }

    public void printInfo() {
        System.out.println("Name: " + name + ", City: " + address.getCity());
    }
}

public class Main {
    public static void main(String[] args) {
        Address address = new Address("New York");
        Person person = new Person("John", address);
        person.printInfo();  // 输出: Name: John, City: New York
    }
}
```

static 关键字

static 关键字可以用来修饰变量（类变量-实例变量）和方法（类方法-实例方法），表示它们属于类而不是某个具体的对象实例。

实例方法可以调用实例变量和实例方法，也可以调用类变量和类方法，但是类方法只能调用类变量和类方法，不能调用实例变量和实例方法。

一个类的不同对象实例共享同一个类变量，使用相同的内存空间，直到程序结束运行才释放内存。

示例：

```
class MyClass {
    private static int count = 0; // 静态变量，属于类而不是对象实例

    public MyClass() {
        count++; // 每次创建对象时，count 变量加 1
    }

    public static int getCount() {
        return count; // 静态方法，可以通过类名直接调用
    }
}

public class Main {
    public static void main(String[] args) {
        // 创建 MyClass 的两个实例
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();
        MyClass obj3 = new MyClass();

        // 输出 MyClass 的实例数量
        System.out.println("MyClass 的实例数量: " + MyClass.getCount());
    }
}
```

this 关键字

`this`关键字在构造方法和实例方法中用于引用当前对象的实例变量和方法，但是静态方法中不能使用`this`。

在构造方法中，用于区分构造方法的参数名与实例变量名：

```
public class Person {
    private String name;
    private int age;

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

在实例方法中，用于区分局部变量和实例变量：

```
public class Person {
    private String name;

    public void setName(String name) {
```

```
        this.name = name;
        // this.name 是实例变量，name 是方法参数
    }

    public void printName() {
        System.out.println(this.name);
        // 通过 this 访问实例变量
    }
}
```

在构造方法中，调用当前类的其他构造方法，即重载：

```
public class Person {
    private String name;
    private int age;

    public Person() {
        this("Unknown", 0);  // 调用带有参数的构造方法
    }

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

包 package

`package`是用于组织类、接口和其他包的命名空间机制，类似于文件夹，通过包的层次结构来管理大量的类，避免类名冲突，有效减少类名冲突，并提供访问控制。

包的命名规则（通常情况下）：

- 全部小写，如：`com.example.myapp`
- 以公司或组织的反向网址开头，如：公司网址为`myapp.example.com`，包名为`com.example.myapp`
- 以项目或模块的名称结尾，如：`com.example.myapp.core`

包和文件系统的目录结构是一一对应的。包的层次结构在文件系统中以文件夹形式体现

```
package com.example.utils;

public class Utility {
    类体
}
```

这个包对应的文件系统结构可能是：

```
src/
 └── com/
     └── example/
         └── utils/
             └── Utility.java
```

声明包

在java源文件中，package声明位于除注释外的第一行，一个源文件只能有一个package声明，如：

```
package com.example.myapp; // 声明当前文件为 com.example.myapp 包

public class Person {
    // Person 类属于 com.example.myapp 包，其全限定名为 com.example.myapp.Person
    private String name;
    private int age;

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

    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

导入包

使用import来导入其他包中的类、接口、枚举等。import通常位于package声明之后，类声明之前。

使用import 包名.类名导入某个包的某个类，也可以使用import 包名.*来导入该包下的所有类。

导入包的类之后，可以直接使用类名创建对象，如：

```
import com.example.myapp.Person;

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 25);
        person.displayInfo();
    }
}
```

但是不可以导入两个不同包中同名的类，比如 java.util.Date 和 com.example.package1.Date，只能import其中一个Date，然后使用全限定名来创建另一个Date的实例，如：

```java
import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Date date1 = new Date();
        com.example.package1.Date date2 = new com.example.package1.Date();
    }
}
```

访问权限

Java中常用的访问权限有四种：public、protected、default（默认）和private。

- public：可以被任何其他类访问。
- protected：可以被同一包中的类以及不同包中的子类访问。
- default（默认）：可以被同一包中的类访问，不能被不同包中的类访问。
- private：只能被定义它的类访问。

访问修饰符	同一个类内	同一个包内	子类（不同包）	其他包中的类
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

公有 public

公有变量和公有方法可以被任何其他类访问。

```java
public class Person {
    public String name;  // 公有变量
    public int age;      // 公有变量

    // 构造函数，用于初始化 name 和 age
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // 公共方法，可以从外部类调用
    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

被保护 protected

被保护的变量和方法可以被同一个包中的类访问，也可以被不同包中的子类继承和访问，但不能被其他包中的非子类访问。

```
public class Person {
    protected String name;  // 被保护的变量
    protected int age;      // 被保护的变量

    // 构造函数，用于初始化 name 和 age
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // 被保护的方法，可以在同包和子类中调用
    protected void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

class Employee extends Person {
    public Employee(String name, int age) {
        super(name, age);
    }

    // 可以在子类中调用被保护的方法
    public void showEmployeeInfo() {
        displayInfo();
    }
}
```

默认 default

默认（default）访问权限通常也称为包级访问权限，仅限于同一个包内的类和接口使用，不能被其他包中的类或子类访问。

```
public class Person {
    String name;  // 默认访问权限的变量
    int age;      // 默认访问权限的变量

    // 构造函数，用于初始化 name 和 age
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // 默认访问权限的方法，仅限于同一包内的类调用
    void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        // 同一个包内可以访问默认权限的方法
        person.displayInfo();
    }
}
```

私有 private

私有变量和私有方法只能在定义它们的类中访问，不能从其他类访问。

```
public class Person {
    private String name;  // 私有变量
    private int age;      // 私有变量

    // 构造函数，用于初始化 name 和 age
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // 私有方法，只能在本类内部调用
    private void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    // 公共方法，可以从外部类调用，它调用了私有方法
    public void showInfo() {
        displayInfo();  // 内部调用私有方法
    }
}
```

在Person类中，`name`和`age`是私有变量，只能被Person类中的方法访问。`displayInfo()`是私有方法，只能被Person类中的方法访问。`showInfo()`是公共方法，可以从外部类调用，它调用了私有方法`displayInfo()`，私有方法`displayInfo()`又调用了私有变量`name`和`age`。

基本数据类型的类封装

Java中基本数据类型有8种：byte、short、int、long、float、double、char、boolean。这些基本数据类型没有方法，不能直接调用方法。为了方便操作基本数据类型，Java提供了对应的类封装，这些类封装了基本数据类型，并提供了相应的方法。

有时候需要把数据存入某个地方，而它只接收“对象”，不接收简单的数据，比如Java中的一些集合类（如`ArrayList`），它们只能存放对象，不能直接存放int、char这种数据，所以需要把int装到Integer这种类封装中。

java有“自动装箱和拆箱”的特性，使得基本数据类型和类封装之间可以自动完成转化，如：

```
int num1 = 10;
Integer obj = num1;
// 自动装箱，将 int 类型转换为 Integer 对象
int num2 = obj;
// 自动拆箱，将 Integer 对象转换为 int 类型
```

基本数据类型	封装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

对封装类调用对应的`doubleValue()`、`byteValue()`、`intValue()`、`shortValue()`、`longValue()`方法可以返回该对象的基本类型数据

Character类中的方法可以对字符类型进行判断，如：

```
Character.isDigit(char ch)  // 是否为数字
Character.isLetter(char ch) // 是否为字母
Character.isUpperCase(char ch) // 是否为大写字母
Character.isLowerCase(char ch) // 是否为小写字母
Character.toUpperCase(char ch) // 转换为大写字母
Character.toLowerCase(char ch) // 转换为小写字母
```

jar文件

jar文件是Java Archive的缩写，它是一种压缩文件格式，用于存储Java类文件、资源文件等，可以将有包名的类的字节码文件压缩为jar文件，供其他文件使用`import`引用。

第五章 继承与接口

子类与父类

继承允许子类具有父类的属性和方法，同时有独属于自己的属性和方法。

- 父类（Superclass）：也叫基类，是被继承的类，包含一些通用的属性和方法。
- 子类（Subclass）：也叫派生类，是继承父类的类。它可以继承父类的属性和方法，并且可以扩展或重写父类的方法。

一个子类只能用一个父类，一个父类可以有多个子类。

继承

使用`extends`表示继承关系，语法如下：

```
class 父类 {
    // 父类中的属性和方法
}

class 子类 extends 父类 {
    // 子类中的属性和方法
}
```

重写 override

子类可以重写父类的方法，重写的方法必须与父类的方法 **具有相同的名称、参数列表和返回类型。**

```
class 父类 {
    void 方法1(参数) {
        方法体
    }
}

class 子类 extends 父类 {
    @Override
    // 这是一个注解，表明这个方法是对父类方法的重写
    void 方法1(参数) {
        方法体
    }
}
```

示例：

```
// 父类
class Animal {
    String name; // 父类属性 名字

    public void eat() { // 父类方法
        System.out.println("This animal is eating.");
    }
}

// 子类
```

```
class Dog extends Animal {
    String varieties; // 子类属性 品种
    public void bark() { // 子类方法
        System.out.println("The dog is barking.");
    }

    @Override
    public void eat() { // 重写父类方法
        System.out.println("The dog is eating.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.name = "Buddy";
        myDog.varieties = "Golden Retriever";
        myDog.eat();
        myDog.bark();
    }
}
```

super关键字

在子类中，当子类覆盖了父类的属性或方法时，可以在子类内部使用`super`关键字来调用父类的属性或方法。

super关键字不可以在main()方法中使用

```
class Animal {
    public void makeSound() { // 父类方法
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() { // 重写父类方法
        System.out.println("Dog barks");
    }

    public void callParentSound() {
        // 调用父类被重写的方法
        super.makeSound();
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
        // 输出 "Dog barks"
```

```
        myDog.callParentSound();
        // 调用父类方法，输出 "Animal makes a sound"
    }
}
```

子类的继承性

若子类和父类在同一个包中：

> 子类会自动继承父类的 **非private** 成员变量和方法作为自己的成员变量和方法，被继承的变量和方法的访问权限不变。

若子类和父类不在同一个包中：

访问修饰符	父类和子类在同一个包中	父类和子类不在同一个包中
public	√	√
protected	√	√
默认（无修饰符）	√	×
private	×	×

子类对象的构造过程

当创建一个子类对象时，首先会调用父类的构造方法，然后调用子类的构造方法。

当子类创建对象时，子类中的成员变量和父类中的成员变量都被分配了内存空间（即使是父类的`private`），但是父类的`private`成员变量不作为子类的变量，无法从子类中访问，即子类不继承父类的`private`成员变量。

然后会调用父类中的构造方法，对父类的所有成员变量进行初始化，若没有对其赋值则会赋初值，然后调用子类中的构造方法，对子类的所有成员变量进行初始化。

虽然子类无法访问父类的`private`变量，但是可以在子类中**调用父类的`public`或`protected`方法**来访问父类的`private`变量。

成员变量的隐藏

当子类定义了与父类同名的成员变量时，父类的同名变量会被隐藏。这意味着通过子类对象访问该变量时，实际上访问的是子类的变量，而不是父类的变量。

父类的成员变量被隐藏不是被覆盖，父类的成员变量仍然存在，只是无法通过子类对象访问，但是可以通过父类的实例来访问。

方法的重写 Override

方法的重写是在子类中定义与父类方法签名完全相同的方法，通过子类对象调用该方法时，实际执行的是子类中的实现，而不是父类的实现。

子类方法的权限不能比父类的更严格，若父类方法 public，则子类只能是public。

在方法重写中，可以使用`super`关键字来调用父类的原始方法。

final 关键字

`final`关键字可以用来修饰类、方法和变量。

- `final`类：不能被继承
- `final`方法：不能被子类重写
- `final`变量：初始化后值不能被改变，即常量

对象的上转型对象

将子类对象赋值给父类对象沉稳给对象的上转型

假设a是父类对象，b是子类对象，则`a = b`称为对象的上转型对象。

上转型后，访问父类方法时，若子类中对该方法进行了重写，则会执行子类的方法，若没有重写则会调用父类的版本，但是不可以访问子类独有的方法；访问变量时，即使子类中有同名的变量，访问的也是父类的变量。

```
class A {
    public String name = "Parent";

    public void print() {
        System.out.println("This is Parent class");
    }
}

class B extends A {
    public String name = "Child";

    @Override
    public void print() {
        System.out.println("This is Child class");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B(); // 上转型
        System.out.println(a.name);
        // 调用父类的变量，输出 "Parent"
        a.print();
        // 调用被子类重写的方法，输出 "This is Child class"
    }
}
```

上转型对象和继承的区别

对于上转型对象：

- 将一个子类的对象赋值给一个父类引用

- 在上转型对象中只能访问父类的成员
- 若子类重写了父类的方法则访问被重写的方法
- 不可以访问子类特有的成员，即只能访问转型后引用类型所定义的方法
- 上转型对象可以访问隐藏的变量

常用情况：

```
public class Animal {
    public void eat() {
        System.out.println("This animal is eating.");
    }
}

public class Dog extends Animal {
    public void eat() {
        System.out.println("This dog is eating.");
    }
}

public class Cat extends Animal {
    public void eat() {
        System.out.println("This cat is eating.");
    }
}

public class Main {
    public void feed(Animal animal) {
        // 这个方法接受一个Animal类的参数，调用animal的eat()方法
        animal.eat();
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        feed(dog);
        feed(cat);
        // 传入Animal的子类，方法接收的实际为一个上转型对象
        // 在feed中会调用被子类重写的eat()方法
        // 扩展了feed方法，使其可以接受不同继承自Animal的对象
    }

}
```

多态

同一个方法调用，基于对象的不同，可能会表现出不同的行为，实际调用哪个方法在运行时决定。

多态的两种形式：

- 方法重载：对同一个方法根据不同的参数调用
- 方法重写和上转型：子类重写父类的方法，运行时根据对象的类型决定调用哪个方法

abstract 抽象类

抽象类用于定义一些共同的行为或者规范，具体的实现则交由子类来完成。

抽象类不能被实例化，只能被继承，不能用`final`修饰，不能用`new`创建对象

抽象类用`abstract`修饰，如：

```
abstract class A{
    ...
}
```

抽象类中可以有：

- 抽象方法：只有方法的声明，没有具体实现的方法体
- 非抽象方法：有方法体的方法
- 成员变量
- 构造方法（供子类使用）

一个非抽象类把抽象类当作父类时，必须重写父类中的抽象方法，否则该类也必须被声明为抽象类。

接口

接口定义了类必须实现的行为规范，而不提供具体的实现，即定义多个类之间的公共行为，类可以通过实现接口来遵循这些行为。

接口不能实例化，不能有构造方法，接口中的变量默认为`public static final`

接口使用关键字`interface`定义，类使用`implements`关键字实现接口中的所有方法。

接口中可以有：

- 常量：默认为`public static final`，属于接口本身
- 抽象方法：默认为`public abstract`，声明类必须实现的行为规范，需要在类中重写
- `default`方法：在接口中定义一个方法默认的具体实现，需要有方法体，在子类中可以重写default方法或直接使用default方法，也可用于在不修改其他类的情况下为接口增加新的方法。
- `static`方法：属于接口本身的方法，不能被实现类继承或重写，需要通过接口名直接调用。

```
interface Animal {
    // 常量
    int LEGS = 4;

    // 抽象方法
    void sound();

    // default 方法
    default void sleep() {
```

```
        System.out.println("This animal is sleeping.");
    }

    // static 方法
    static void info() {
        System.out.println("This is the Animal interface.");
    }
}

class Dog implements Animal {

    // 实现接口中的抽象方法
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }

    // 重写接口中的 default 方法（可选）
    @Override
    public void sleep() {
        System.out.println("Dog is sleeping.");
    }
}

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

        dog.sound();
        // 输出 "Dog barks"

        dog.sleep();
        // 输出 "Dog is sleeping."

        // 访问接口中的常量
        System.out.println("Dog has " + Dog.LEGS + " legs.");  // 输出 "Dog has 4
legs."

        // 调用接口中的 static 方法
        Animal.info();  // 输出 "This is the Animal interface."
    }
}
```

接口和继承类的关系

通过类的继承，子类可以**选择性**的实现抽象类中的方法，但是接口要求子类必须实现接口中的**所有方法**，此外实现了接口的类可以有接口中没有的方法。

接口回调

接口回调是指在某个方法中传入一个接口类型的参数，然后在方法内部调用该接口的方法。

接口回调的工作流程：

- 接口定义一套规范，规定必须实现的方法
- 在Task类中设置监听器方法，参数是一个实现了接口的对象
- 在主类中完成接口的具体实现，传递给Task类中设置监听器的方法
- Task类中的任务完成后，通过接口调用回调的方法

```
// 定义回调接口，定义一个任务完成后的操作
interface TaskCompleteListener {
    // 接口中有一个方法
    void TaskComplete();
}

// 在Task类中执行任务，在任务完成时调用接口中的回调方法
class Task {
    // 接口类型的成员变量
    private TaskCompleteListener listener;

    // 设置回调监听器，传入一个实现了 TaskCompleteListener 接口的对象
    public void setListener(TaskCompleteListener listener) {
        this.listener = listener;
    }

    // 执行任务
    public void execute() {
        System.out.println("Executing task...");
        taskComplete();
    }

    // 任务完成后，若设置了回调监听器则调用回调方法
    public void taskComplete() {
        if (listener != null) {
            listener.TaskComplete();
        }
    }
}

// 主类中提供不同的回调实现
public class Main {
    public static void main(String[] args) {
        // 创建任务对象
        Task task1 = new Task();
        Task task2 = new Task();
        Task task3 = new Task();

        // 对接口方法的实现
        task1.setListener(new TaskCompleteListener() {

            @Override
            public void TaskComplete() {
                System.out.println("Task 1 finished!");
            }
```

```
        });
        task2.setListener(new TaskCompleteListener() {
            @Override
            public void TaskComplete() {
                System.out.println("Task 2 finished!");
            }
        });
        task3.setListener(new TaskCompleteListener() {
            @Override
            public void TaskComplete() {
                System.out.println("Task 3 finished!");
            }
        });

        // 执行任务，然后自动执行TaskComplete
        task1.execute();
        task2.execute();
        task3.execute();
    }
}
```

若不使用接口回调，则需要为每个任务创建不同的 Task 实现，这会导致大量的代码重复：

```
class Task1 {
    public void execute() {
        System.out.println("Executing task 1...");
        System.out.println("Task 1 finished!");
    }
}

class Task2 {
    public void execute() {
        System.out.println("Executing task 2...");
        System.out.println("Task 2 finished!");
    }
}

class Task3 {
    public void execute() {
        System.out.println("Executing task 3...");
        sendEmail();
    }

    public void sendEmail() {
        System.out.println("Task 3 finished!");
    }
}
```

与不使用接口回调相比，接口回调的优点是：

- 解耦性：Task类仅负责任务的实现，而任务完成后的操作由外部类通过实现接口来决定。
- 灵活性：通过不同的接口实现为 Task 类设置不同的回调方法，也可以动态地改变回调逻辑，让 Task 类可以重复使用，任务完成后的操作根据需要灵活定义。
- 可扩展性：外部类可以通过实现接口来扩展任务完成后的操作，即新的功能可以通过新的回调实现类来添加，而不需要修改 Task 类。
-

第六章 特殊的类 Lambda表达式

内部类

内部类是在类内部定义的类，可以让某个类的逻辑更加紧凑，其他类无法使用某个类中的内部类来声明对象

在内部类中不可以声明static变量和方法

成员内部类

最常见的内部类，可以访问外部类的所有成员变量和方法（包括private）

通需要过外部类的实例来创建内部类的实例

```
class Outer {
    private String message = "Hello from Outer class";

    class Inner {
        public void printMessage() {
            System.out.println(message);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        // 创建外部类对象
        Outer.Inner inner = outer.new Inner();
        // 创建内部类对象

        inner.printMessage(); // 输出: Hello from Outer class
    }
}
```

静态内部类

可以直接通过外部类的类名来创建，而不需要外部类的实例，仅能访问外部类的static成员变量和方法

```
class Outer {
    private static String staticMessage = "Hello from Static Inner class";
```

```
    static class Inner {
        public void printStaticMessage() {
            System.out.println(staticMessage);
        }
    }
}
```

局部内部类

在一个方法中定义，仅能在这个方法中使用，可以访问外部类的成员变量和方法

```
class Outer {
    private String message = "Hello from Local Inner class";

    public void displayMessage() {
        // 局部内部类
        class Inner {
            public void printMessage() {
                System.out.println(message); // 访问外部类的成员
            }
        }

        Inner inner = new Inner();
        inner.printMessage(); // 调用局部内部类的方法
    }
}
```

匿名类

匿名类是一种不单独定义，没有名字的类，常用于简化代码，特别是在实现接口或继承类时，匿名类可以用于快速定义一个类的实现，而无需显式地创建一个新的类文件。

```
// 定义Runnable接口
public interface Runnable {
    void run();
}

// 主程序类
public class Main {
    public static void main(String[] args) {
        // 使用匿名类实现Runnable接口
        var thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("匿名类的run方法被调用");
            }
        });
```

```
        // 启动线程
        thread.start();
    }
}
```

与不使用匿名类相比：

```
// 定义Runnable接口
public interface Runnable {
    void run();
}

// 显式实现Runnable接口的类
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("MyRunnable类的run方法被调用");
    }
}

// 主程序类
public class Main {
    public static void main(String[] args) {
        // 创建MyRunnable对象
        MyRunnable myRunnable = new MyRunnable();

        // 创建Thread对象并传入MyRunnable对象
        Thread thread = new Thread(myRunnable);

        // 启动线程
        thread.start();
    }
}
```

Lambda表达式

Lambda表达式，即一个匿名方法（函数），允许使用更简洁的语法来表示函数或代码块，语法：

```
(parameters) -> expression
或
(parameters) -> { statements }

箭头左侧参数可以有多个或没有
箭头右侧可以是单一表达式或大括号括起来的代码块
```

```
() -> System.out.println("Hello, Lambda!");
```

```
(str) -> System.out.println(str);

(int a, int b) -> a + b;

(double a, double b) -> {
    double sum = a + b;
    return sum;
};
```

对**只有一个方法**的接口，可以使用Lambda表达式来简化代码，例如：

```
public interface Task {
    void execute();
}

public class Main {
    public static void main(String[] args) {
        // 使用Lambda表达式实现Task接口
        Task task1 = () -> System.out.println("Executing task...");

        // 不使用Lambda表达式实现Task接口
        var task2 = new Task() {
            @Override
            public void execute() {
                System.out.println("Executing task...");
            }
        };

        task1.execute();
        task2.execute();
    }
}
```

异常类

Java的异常类分为Error和Exception两种：

- Error：表示系统错误，通常无法被程序处理，例如内存溢出、堆栈溢出等。
- Exception：表示程序运行时可能出现的错误，分为检查异常（Checked Exception）和运行时异常（Runtime Exception）。

使用`try...catch`语句处理异常：在`try`中的语句发生异常后，`try`立即结束运行，跳转到对应的`catch`部分，如果`catch`没有匹配的异常类型，则继续向上抛出异常。

```
try {
    // 可能发生异常的代码
} catch (ExceptionType1 e1) {
    // 处理ExceptionType1异常的代码
} catch (ArrayIndexOutOfBoundsException e) {
```

```
    System.out.println("数组下标越界异常: " + e.getMessage());
} catch (Exception e){
    System.out.println("产生异常: " + e.getMessage());
} finally {
    // 无论是否发生异常，都会执行的代码
}
```

可以扩展Exception类来自定义异常，例如：

```
// 自定义异常类: 年龄小于18
class InvalidAgeTooYoungException extends Exception {
    public InvalidAgeTooYoungException(String message) {
        super(message);  // 调用父类的构造函数
    }
}

// 自定义异常类: 年龄大于40
class InvalidAgeTooOldException extends Exception {
    public InvalidAgeTooOldException(String message) {
        super(message);  // 调用父类的构造函数
    }
}

public class AgeValidationExample {
    // 检查年龄的方法，抛出不同的自定义异常
    // 使用throws声明可能的异常类型
    public static void validateAge(int age) throws InvalidAgeTooYoungException,
InvalidAgeTooOldException {
        if (age < 18) {
            // 使用throw抛出异常
            // 年龄小于18，抛出 InvalidAgeTooYoungException
            throw new InvalidAgeTooYoungException("年龄必须大于或等于 18");
        } else if (age > 40) {
            // 年龄大于40，抛出 InvalidAgeTooOldException
            throw new InvalidAgeTooOldException("年龄不能超过 40");
        } else {
            System.out.println("年龄有效: " + age);
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(45);  // 测试年龄，触发不同的异常
        } catch (InvalidAgeTooYoungException e) {
            System.out.println("捕获到异常: 太年轻: " + e.getMessage());
        } catch (InvalidAgeTooOldException e) {
            System.out.println("捕获到异常: 太年长: " + e.getMessage());
        } finally {
            System.out.println("判断完毕");
        }
    }
}
```

执行逻辑：

1. 异常类定义：定义两个继承自`Exception`的自定义异常类，接受一个str用于描述异常的具体信息，在构造函数中调用`super(message)`将str传递给父类的构造函数
2. 抛出异常：在`validateAge`方法中，首先用`throws`声明所有可能的异常类型，然后根据不同的年龄使用`throw`抛出不同的异常，并传递对应的描述信息，若无异常则执行else
3. 捕获异常：调用`validateAge`方法，判断是否抛出异常，若抛出则捕获对应的异常并打印异常信息
4. 执行finally的语句

断言语句

用于测试代码的逻辑正确性，确保程序的状态或行为符合预期。
断言失败时会抛出`AssertionError`异常。

```
public class AssertionExample {
    public static void main(String[] args) {
        int age = -5;

        // 简单断言
        assert age >= 0; // 如果 age 小于 0，将抛出 AssertionError

        // 带消息的断言
        assert age >= 0 : "年龄不能为负数，当前年龄: " + age;

        System.out.println("年龄是: " + age);
    }
}
```

断言默认是禁用的，需要通过`java -ea AssertionExample`启用断言。

java反射

java反射允许程序在运行时动态获取类的信息，并且可以操作类的成员，如属性、方法、构造函数甚至是实例化对象。

```
import java.lang.reflect.*;

class Person {
    private String name;

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

    public void greet() {
        System.out.println("Hello, my name is " + name);
    }
```

```
    private void secret() {
        System.out.println("This is a secret method.");
    }
}

public class ReflectionDemo {
    public static void main(String[] args) {
        try {
            // 获取 Person 类的 Class 对象
            Class<?> personClass = Class.forName("Person");

            // 创建实例
            Constructor<?> constructor = personClass.getConstructor(String.class);
            Object personInstance = constructor.newInstance("Alice");

            // 调用 greet 方法
            Method greetMethod = personClass.getMethod("greet");
            greetMethod.invoke(personInstance);

            // 访问私有字段 name
            Field nameField = personClass.getDeclaredField("name");
            nameField.setAccessible(true); // 允许访问私有字段
            System.out.println("Name: " + nameField.get(personInstance));

            // 修改私有字段 name
            nameField.set(personInstance, "Bob");
            greetMethod.invoke(personInstance); // 调用 greet 方法

            // 调用私有方法
            Method secretMethod = personClass.getDeclaredMethod("secret");
            secretMethod.setAccessible(true); // 允许访问私有方法
            secretMethod.invoke(personInstance); // 调用私有方法

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

第七章 面向对象设计的基本原则

开闭原则

“对扩展开放，对修改关闭” 即：

- 允许在不修改现有代码的基础上，通过增加新功能来扩展系统的行为
- 增加新的模块时不需要修改现有的模块

```java
// 抽象类或接口：Shape
interface Shape {
    double getArea();  // 抽象方法
}

// 圆形类
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}

// 矩形类
class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }
}

// 客户端代码
class AreaCalculator {
    public double calculateTotalArea(Shape[] shapes) {
        double totalArea = 0;
        for (Shape shape : shapes) {
            totalArea += shape.getArea();  // 多态调用
        }
        return totalArea;
    }
}
```

多用组合，少用继承

优先使用组合（实现接口）来实现对象间的功能重用和扩展，尽量减少对继承（继承父类）的依赖。

继承的缺点：

- 父类和子类是强耦合关系，对父类的改动会影响所有子类
- 一个子类只能继承一个父类
- 如果父类设计不当或后期扩展需求变化，子类容易出问题
- 继承关系确定后，子类的行为难以动态改变

组合的优点：

- 组合是弱耦合关系，只依赖于接口或对象，而不是类层次结构
- 可以复用多个类的功能，而不需要考虑类的继承关系
- 可以在运行时改变不同组件之间的组合，实现更多的动态行为
- 组合允许扩展功能而无需修改已有代码

第八章 设计模式

针对某一类问题的最佳解决方案，即一个**成功的、可复用的**设计方案

框架

针对某个领域，用于开发的**类的集合**，包括多个设计模式

设计模式的分类

- 行为型模式
  - 策略模式
  - 访问者模式
- 结构型模式
  - 装饰模式
  - 适配器模式
- 创建型模式
  - 工厂方法模式

策略模式

将一系列**平行的算法**封装起来，使他们可以互相替换。

- 策略接口 Strategy：定义若干个抽象方法
- 具体策略 ConcreteStrategy：实现策略接口，给出具体算法
- 上下文 Context：依赖于接口的类，其中包含的方法调用具体策略所实现的策略接口中的方法

优点：

- 每种算法都可以独立变化，客户端不需要知道具体实现
- 添加新的策略时，不需要修改Context的代码，在Context中可以直接引用新的具体策略的实例，符合开闭原则
- Context和ConcreteStrategy是松耦合关系，Context只需要知道它要使用一个实现了Strategy接口类的实例

```java
// 策略接口
interface DiscountStrategy {
    double applyDiscount(double price);
}

// 具体策略1：百分比折扣
class PercentageDiscount implements DiscountStrategy {
    private double percentage;

    public PercentageDiscount(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public double applyDiscount(double price) {
        return price * (1 - percentage / 100);
    }
}

// 具体策略2：固定金额折扣
class FixedAmountDiscount implements DiscountStrategy {
    private double amount;

    public FixedAmountDiscount(double amount) {
        this.amount = amount;
    }

    @Override
    public double applyDiscount(double price) {
        return price - amount;
    }
}

// 上下文
class ShoppingCart {
    private DiscountStrategy discountStrategy;

    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    public double calculateTotal(double price) {
        return discountStrategy.applyDiscount(price);
    }
}

// 使用示例
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setDiscountStrategy(new PercentageDiscount(10)); // 10%折扣
        System.out.println("总价: " + cart.calculateTotal(100)); // 输出: 90.0
```

```
        cart.setDiscountStrategy(new FixedAmountDiscount(15)); // 固定金额折扣
        System.out.println("总价: " + cart.calculateTotal(100)); // 输出: 85.0
    }
}
```

访问者模式

表示一个作用于某对象结构中的各个元素的操作，使用户可以在不改变各个元素的类的前提下定义作用于这些元素的新操作。

- 抽象元素 Element：一个接口，定义接收访问者的`accept`方法
- 具体元素 ConcreteElement：定义自己的行为，实现`accept`方法，将当前元素传递给访问者
- 抽象访问者 Visitor：一个接口，包含对各种具体元素的访问方法
- 具体访问者 ConcreteVisitor：定义对各种具体元素的操作

优点：

- 将操作与对象结构分离，使得在不改变对象结构的情况下，可以增加新的操作
- 增加新操作时，只需创建一个新的访问者类，而不需要修改现有的元素类，避免对现有代码的干扰，遵循开闭原则
- 将所有的处理逻辑都集中在访问者中，便于管理和维护

```
// 访问者接口
interface BillVisitor {
    void visit(ResidentialUser user);
    void visit(CommercialUser user);
}

// 具体访问者：电费计算
class ElectricityBillVisitor implements BillVisitor {
    private double residentialRate; // 居民电价
    private double commercialRate;   // 商业电价

    public ElectricityBillVisitor(double residentialRate, double commercialRate) {
        this.residentialRate = residentialRate;
        this.commercialRate = commercialRate;
    }

    @Override
    public void visit(ResidentialUser user) {
        double amount = user.getConsumption() * residentialRate; // 计算电费
        System.out.println("居民用户电费: ¥" + amount);
    }

    @Override
    public void visit(CommercialUser user) {
        double amount = user.getConsumption() * commercialRate; // 计算电费
        System.out.println("商业用户电费: ¥" + amount);
    }
```

```
}

// 元素接口
interface User {
    void accept(BillVisitor visitor);
    double getConsumption();
}

// 具体元素：居民用户
class ResidentialUser implements User {
    private double consumption; // 用电量

    public ResidentialUser(double consumption) {
        this.consumption = consumption;
    }

    @Override
    public void accept(BillVisitor visitor) {
        visitor.visit(this);
    }

    @Override
    public double getConsumption() {
        return consumption;
    }
}

// 具体元素：商业用户
class CommercialUser implements User {
    private double consumption; // 用电量

    public CommercialUser(double consumption) {
        this.consumption = consumption;
    }

    @Override
    public void accept(BillVisitor visitor) {
        visitor.visit(this);
    }

    @Override
    public double getConsumption() {
        return consumption;
    }
}

// 使用示例
public class Main {
    public static void main(String[] args) {
        double residentialRate = 0.5; // 居民单价
        double commercialRate = 0.8;   // 商业单价

        User residentialUser = new ResidentialUser(100); // 100度电
        User commercialUser = new CommercialUser(150);   // 150度电
```

```
        BillVisitor billVisitor = new ElectricityBillVisitor(residentialRate,
commercialRate);

        residentialUser.accept(billVisitor); // 输出：居民用户电费：¥50.0
        commercialUser.accept(billVisitor);   // 输出：商业用户电费：¥120.0
    }
}
```

1. 定义接口和类

- BillVisitor接口：定义不同用户计算电费的visit方法
- User接口：定义用户的accept和getConsumption方法
- ElectricityBillVisitor类：实现BillVisitor接口，定义对不同类型用户的电费计算方法
- ResidentialUser和CommercialUser类：实现User接口，表示居民和商业用户

2. 创建User类型的上转型对象：居民和商业

```
User residentialUser = new ResidentialUser(100); // 100度电
User commercialUser = new CommercialUser(150);   // 150度电
```

3. 创建访问者实例ElectricityBillVisitor，传入居民和商业的用电单价

```
double residentialRate = 0.5; // 居民单价
double commercialRate = 0.8;   // 商业单价
BillVisitor billVisitor = new ElectricityBillVisitor(residentialRate,
commercialRate);
```

4. 计算电费

分别调用不同用户对象的accept方法，传入访问者对象billVisitor，计算不同用户的电费

```
residentialUser.accept(billVisitor); // 计算居民电费
commercialUser.accept(billVisitor);   // 计算商业电费
```

在ResidentialUser的accept方法中，billVisitor.visit(this)将当前对象ResidentialUser传递给访问者ElectricityBillVisitor，访问者根据不同的用户类型选择不同的单价计算电费。

装饰模式

动态地给对象添加额外的功能。通过组合多个装饰器，可以实现复杂的功能而无需修改原始对象的代码。

- 抽象组件 Component：一个接口，描述被装饰对象的共同特性，定义需要进行装饰的方法
- 具体组件 ConcreteComponent：实现抽象组件接口，表示被装饰的原始对象
- 装饰器 Decorator：实现抽象组件接口，持有一个具体组件的引用。用于装饰具体组件

- 具体装饰器 ConcreteDecorator：装饰器的子类，添加具体的功能

```
// 抽象组件
interface Coffee {
    String getDescription();
    double cost();
}

// 具体组件
class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "简单咖啡";
    }

    @Override
    public double cost() {
        return 5.0;
    }
}

// 抽象装饰器
abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }
}

// 具体装饰器：添加牛奶
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", 牛奶";
    }

    @Override
    public double cost() {
        return coffee.cost() + 1.5; // 牛奶附加费用
    }
}

// 具体装饰器：添加糖
class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }
```

```
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", 糖";
    }

    @Override
    public double cost() {
        return coffee.cost() + 0.5; // 糖附加费用
    }
}

// 使用示例
public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " 费用: ¥" + coffee.cost());

        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " 费用: ¥" + coffee.cost());

        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " 费用: ¥" + coffee.cost());
    }
}
```

```
简单咖啡 费用: ¥5.0
简单咖啡, 牛奶 费用: ¥6.5
简单咖啡, 牛奶, 糖 费用: ¥7.0
```

适配器模式

将一个类的接口转换为对客户友好的另一个接口，使得由于接口不兼容而无法一起工作的类可以一起工作。

- 目标 Target：一个客户使用的接口
- 被适配者 Adaptee：已经存在的接口或抽象类
- 适配器 Adapter：一个类，实现了目标接口并包含被适配者的引用，对被适配者与目标接口进行适配

```
// 目标接口 USBC
interface USBTypeC {
    void chargeWithTypeC();
}

// 被适配者 Micro-USB
class MicroUSBCharger {
    public void chargeWithMicroUSB() {
        System.out.println("使用 Micro-USB 充电中...");
    }
}
```

```
// 适配器
class MicroUSBToTypeCAdapter implements USBTypeC {
    private MicroUSBCharger microUSBCharger;

    public MicroUSBToTypeCAdapter(MicroUSBCharger microUSBCharger) {
        this.microUSBCharger = microUSBCharger;
    }

    @Override
    public void chargeWithTypeC() {
        System.out.println("适配器转换中...");
        microUSBCharger.chargeWithMicroUSB();
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        // 创建一个Micro-USB充电器
        MicroUSBCharger microUSBCharger = new MicroUSBCharger();

        // 使用适配器让它为USB-C接口的手机充电
        USBTypeC chargerAdapter = new MicroUSBToTypeCAdapter(microUSBCharger);
        chargerAdapter.chargeWithTypeC();  // 输出: 适配器转换中... 使用 Micro-USB
充电中...
    }
}
```

工厂方法（虚拟构造）模式

责任链模式

第九章 常用类

String

构造String对象

```
// 使用字符串字面量
String str1 = "Hello, World!";

// 使用new关键字
String str2 = new String("Hello, World!");

// 使用字符数组
char s[] = {'J', 'a', 'v', 'a'};
String str3 = new String(s);
String str4 = new String(s, 1, 3); // 从索引1开始，长度为3的子数组
```

字符串的值储存在**常量池**中，若有两个String类型的值相同，则都指向同一个值，而不是创建两个相同的对象。

```
String s1 = "hello"; // 在常量池中创建"hello"
String s2 = "hello"; // 直接引用常量池中已有的"hello"
System.out.println(s1 == s2);  // true，因为指向同一个对象

String s3 = new String("hello");  // 创建新对象
String s4 = new String("hello");  // 又创建一个新对象
System.out.println(s3 == s4);  // false，因为二者是不同对象

String s5 = "hel" + "lo";  // 编译时会优化为"hello"，从常量池中引用

String s6 = "hel";
String s7 = s6 + "lo";  // 不会直接引用常量池中的"hello"
```

String常用方法

对`String str1 = "Java"`：

- `str1.length()`：返回字符串的长度，即字符个数
- `str1 + str2`：字符串连接，返回String类型
- `str1.equals(str2)`：比较字符串内容是否相等，区分大小写，返回boolean类型
- `str1.indexOf("World")`：返回子串在字符串中首次出现的索引，不存在则返回-1
- `str1.charAt(1)`：返回str1中索引为1的字符
- `str1.substring(5)`：从索引5截取到字符串末尾，返回String类型
- `str1.sunstring(5, 10)`：从索引5截取到索引10（不包括10），返回String类型
- `str1.replace(str2, str3)`：替换str1中所有的str2为str3
- `str1.toUpperCase()`：转换为大写
- `str1.toLowerCase()`：转换为小写
- `str1.startsWith(String prefix)`
- `str1.startsWith(String prefix, int offset)`，偏移值为跳过str1的前offset个字符
    - 判断是否前缀为指定字符串，返回布尔值
- `str1.trim()`：返回一个String对象，其值去掉了str1的**前后的**空格、制表符、换行符等空白字符（不处理内部的）

String对象和基本数据的互相转化

```
// 基本数据类型转换为String
int num = 123;
String str1 = String.valueOf(num);
String str2 = Integer.toString(num);

// String转换为基本数据类型
String str3 = "456";
int num2 = Integer.parseInt(str3);
// Integer.parseInt转换失败（非数字、null、空字符串、超出int范围）时会抛出
NumberFormatException异常
```

Object类 和 toString()方法

`Object`类是java中所有类的根类，所有类都直接或间接继承自`Object`类。
`Object`类提供了一些基本的方法，他们都可以在子类中被重写，如：

- `toString()`：返回对象的字符串表示：`"类名@哈希码值"`
- `equals()`：比较两个对象的**引用**是否相等，通常需要重写为比较内容
- `getClass()`：返回对象的运行时类

其中`toString()`方法通常会被重写为返回对象的详细信息，如：

```
class Person {
    private String name;
    private int age;

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

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

正则表达式

正则表达式（Regular Expression， regex）是一种用于处理字符串的强大工，可以用来搜索、匹配、替换和分割字符串。

java中正则表达式通常通过`java.util.regex`包中的`Pattern`类（表示编译后的正则表达式对象）和`Matcher`类（执行匹配操作）实现。

匹配规则

1. 字符匹配
    - 普通字符：匹配自身，如正则表达式`"cat"`匹配字符串`"cat"`
    - 转义字符：匹配`. * ?`等符号时需要加反斜杠转义，如`"\. \* \?"`
2. 字符类，使用方括号`[]`表示
    - `[abc]`：a或b或c
    - `[a-z]`：从a到z的小写字母
    - `[A-Z]`：从A到Z的大写字母
    - `[0-9]`：从0-9的数字
    - `[^a-c]`：除a-c的小写字母
    - `[a-d[m-p]]`：并集，a-d或m-p
    - `[a-z&&[m-p]]`：交集，a-z中m-p

    - `[a-z&&[^m-p]]`：差集，a-z中除去m-p
3. 预定义字符类：使用时需要加反斜杠转义，如`\. \\d \\S`
    - `.`：除换行符外的任意字符
    - `\d`：数字
    - `\D`：非数字字符
    - `\w`：数字、字母、下划线
    - `\W`：非数字、字母、下划线的字符
    - `\s`：空白字符，如空格、制表符、换行符
    - `\S`：非空白字符
    - `\p{Lower}`：小写字母
    - `\p{Upper}`：大写字母
    - `\p{Alpha}`：字母
    - `\p{Digit}`：数字
    - `\p{Alnum}`：字母或数字
    - `\p{Punct}`：标点符号`! " # $ % & 等`
    - `\p{ASCII}`：ASCII字符
4. 边界匹配：
    - `^`：匹配开头，如`"^cat"`匹配以`"cat"`开头的字符串
    - `$`：匹配结尾
    - `\b`：匹配单词边界（空格、标点符号），如`"\bhello\b"`仅匹配完整的单词`"hello"`
    - `\B`：匹配非单词边界，如`"\Bworld\B"`仅匹配仅作为其他字符一部分出现的`"world"`，即不在单词的开头或结尾处
5. 重复匹配：
    - `char*`：匹配char0次或多次
    - `char+`：匹配char1次或多次
    - `char?`：匹配char0次或1次
    - `char{n}`：匹配char恰好n次
    - `char{n,}`：匹配char至少n次
    - `char{n,m}`：匹配char $[n,m]$ 次
6. 组：将表达式部分封装成组，以便重复或逻辑操作。
    - 捕获组：使用圆括号 `()` 表示捕获组，匹配的内容可以通过 `Matcher.group()` 方法获取。例如，`(abc)` 匹配 `abc` 并将其捕获。
    - 非捕获组：使用 `(?:...)` 表示非捕获组，仅用于分组逻辑，不捕获内容。例如，`(?:abc)` 匹配 `abc`，但不捕获。
    - 向前断言：`(?=...)` 表示正向前瞻匹配。例如，`a(?=b)` 匹配 `a`，但要求后面跟着 `b`。
    - 向后断言：`(?<=...)` 表示正向后顾匹配。例如，`(?<=a)b` 匹配 `b`，但要求前面有 `a`。
7. 逻辑预算
    - 或运算：`|` 表示逻辑或，匹配左右两边的任意一个表达式,如`cat|dog` 匹配 `"cat"` 或 `"dog"`。
    - 贪婪与惰性匹配：正则表达式默认是**贪婪的**，尽可能多地匹配字符。在量词后添加 `?`，可以变为**惰性匹配**，即尽可能少地匹配字符。例如，`.*` 是贪婪的，`.*?` 是惰性的。

常用方法

- `Pattern.compile(String regex)`: 将正则表达式编译为 `Pattern` 对象。
- `Matcher matcher = pattern.matcher(String input)`: 通过 `Pattern` 创建 `Matcher` 对象。
- `matcher.matches(String regex)`: 判断整个字符串是否与正则表达式匹配。
- `matcher.find(String regex)`: 查找字符串中与正则表达式匹配的部分。

- `matcher.group()`: 获取匹配的子字符串。

使用正则表达式匹配字符串的过程

1. 使用 `Pattern.compile(String regex)` 方法将正则表达式编译为 `Pattern` 对象。
2. 通过 `Pattern.matcher()` 方法，传入要匹配的字符串，生成 Matcher 对象
3. 使用 `Matcher` 对象的各种方法来进行匹配操作，如 `matches()`、`find()` 等

使用 `Pattern.matches(regex, input)` 检查整个字符串是否与正则表达式匹配，返回布尔值

```
import java.util.regex.Pattern;

public class RegexExample {
    public static void main(String[] args) {
        String regex = "\\d{3}-\\d{2}-\\d{4}";
        // 三个数字-两个数字-四个数字
        String input = "123-45-6789";  // 要匹配的字符串

        // 使用 matches 方法匹配整个字符串
        if (Pattern.matches(regex, input)) {
            System.out.println("格式正确! ");
        } else {
            System.out.println("格式错误! ");
        }
    }
}
```

使用 `matcher.find()` 查找输入字符串中是否存在匹配的子串
使用 `matcher.group()` 返回匹配的子串

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexExample {
    public static void main(String[] args) {
        String regex = "\\d{3}-\\d{2}-\\d{4}";  // 正则表达式，匹配社会保障号格式
        String input = "我的SSN是123-45-6789。";  // 要匹配的字符串

        // 编译正则表达式
        Pattern pattern = Pattern.compile(regex);
        // 创建匹配器
        Matcher matcher = pattern.matcher(input);

        // 使用 find 方法查找部分匹配
        if (matcher.find()) {
            System.out.println("找到匹配: " + matcher.group());
        } else {
            System.out.println("未找到匹配");
        }
```

```
    }
}
```

使用 `while(matcher.find())` 循环查找所有匹配的子串

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexExample {
    public static void main(String[] args) {
        String regex = "\\d+";  // 匹配一个或多个数字
        String input = "价格分别是10元、20元和30元";  // 要匹配的字符串

        // 编译正则表达式
        Pattern pattern = Pattern.compile(regex);
        // 创建匹配器
        Matcher matcher = pattern.matcher(input);

        // 查找所有匹配的数字
        while (matcher.find()) {
            System.out.println("找到匹配的数字: " + matcher.group());
        }
    }
}
```

StringBuffer

StringBuffer是可变的String对象，可以动态修改内容而不需要创建新对象，可以自动扩展占用的空间（默认为16）

StringBuffer位于`java.lang`包中，不需要手动导入

```
public class StringBufferExample {
    public static void main(String[] args) {
        // 创StringBuffer对象
        var sb = new StringBuffer("Hello");

        // 1. aappend(String str): 在末尾添加字符串
        sb.append(" World");
        System.out.println("After append: " + sb); // 输出: Hello World

        // 2. insert(int offset, String str): 在指定位置插入字符串
        sb.insert(6, "Java ");
        System.out.println("After insert: " + sb); // 输出: Hello Java World

        // 3. delete(int start, int end): 删除指定范围内的字符
        sb.delete(6, 11);
        System.out.println("After delete: " + sb); // 输出: Hello World
```

```
        // 4. reverse(): 反转字符串
        sb.reverse();
        System.out.println("After reverse: " + sb); // 输出: dlroW olleH

        // 5. 转换为String
        String str = sb.toString();
        System.out.println("Converted to String: " + str); // 输出: dlroW olleH

        // 6. 获取当前长度
        System.out.println("Length: " + sb.length()); // 输出: 11

        // 7. 获取当前容量
        System.out.println("Capacity: " + sb.capacity()); // 输出: 16 (默认初始容
量)
    }
}
```

Math

方法名	描述
`Math.abs(x)`	返回 `x` 的绝对值。
`Math.max(x, y)`	返回 `x` 和 `y` 中的最大值。
`Math.min(x, y)`	返回 `x` 和 `y` 中的最小值。
`Math.ceil(x)`	返回大于等于 `x` 的最小整数。
`Math.floor(x)`	返回小于等于 `x` 的最大整数。
`Math.round(x)`	返回最接近 `x` 的整数。
`Math.exp(x)`	返回 `e^x` 的值。
`Math.log(x)`	返回 `x` 的自然对数（底数为 `e`）。
`Math.sqrt(x)`	返回 `x` 的非负平方根。
`Math.pow(x, y)`	返回 `x` 的 `y` 次幂。
`Math.random()`	返回大于等于 0.0 小于 1.0 的随机浮点数。
`Math.random(int n)`	返回大于或等于 0 小于 `n` 的随机整数。
`Math.sin(x)`	返回 `x` 的正弦值。
`Math.cos(x)`	返回 `x` 的余弦值。
`Math.tan(x)`	返回 `x` 的正切值。
`Math.asin(x)`	返回 `x` 的反正弦值。
`Math.acos(x)`	返回 `x` 的反余弦值。
`Math.atan(x)`	返回 `x` 的反正切值。

方法名	描述
`Math.E`	自然对数的底数 `e` 的值。
`Math.PI`	圆周率的值。

Random

```
// 使用系统时间作为种子
Random random = new Random();

// 使用指定的种子
Random randomWithSeed = new Random(long seed);

// 随机整数
// 范围从Integer.MIN_VALUE到Integer.MAX_VALUE
// 即-2,147,483,648到2,147,483,647
int randomInt = random.nextInt();

// [0, bound) 之间的随机整数
int randomIntBounded = random.nextInt(int bound);

// [0.0, 1.0) 之间的随机浮点数
float randomFloat = random.nextFloat();

// [0.0, 1.0) 之间的随机双精度浮点数
double randomDouble = random.nextDouble();
```

Java Swing

用于开发桌面应用程序的Java GUI工具包

第十二章 输入输出流

File类

一个File类的对象可以指向一个文件或一个目录

```
var file = new File(String path);
var file = new File(String parent, String filename);
var file = new File(File parent, String filename); // parent为父目录

var file = new File("C:\\example\\file.txt");
var file = new File("C:\\example", "file.txt");
var parentDir = new File("C:\\example");
var file = new File(parentDir, "file.txt");
```

File类的常用方法

文件：

方法	描述
boolean createNewFile()	创建新文件
boolean delete()	删除文件或空目录
String getName()	获取文件名
boolean exists()	判断文件或目录是否存在
String getAbsolutePath()	获取文件的绝对路径
String getParent()	获取文件的父目录
boolean isDirectory()	判断是否为目录
boolean isFile()	判断是否为文件

目录：

方法	描述
boolean mkdir()	创建目录
String[] list()	获取目录下的所有文件
File[] listFiles()	用File对象返回目录下的所有文件

运行可执行文件

```
// 创建Runtime对象
Runtime ec = Runtime.getRuntime();
// 执行命令
ec.exec("notepad.exe");
```

文件字节流 FileInput/OutputStream

字节流不适用于中文，因为字节流没法很好的处理占用两字节的Unicode字符

文件字节输入流 FileInputStream

构造方法：

- FileInputStream
  - (String name)
  - (File file)
  - name和file是输入流的源，输入流从源中读取数据

建立输入流失败时会抛出IOException异常

以字节为单位读取：

- int read
  - (byte b[])：读取b.length个字节并存入b中
  - (byte b[], int off, int len)：读取len个字节，从b中off的位置存入
  - read方法顺序读取文件，只要不关闭流就会一直读取下去，直到文件末尾，返回-1

```
var fis = new FileInputStream("C:\\example\\file.txt");
int b; // 储存当前读取的字节
while((b = fis.read()) != -1) {
    System.out.print((char)b);
}
fis.close();
```

文件字节输出流 FileOutputStream

构造方法：

- FileOutputStream
  - (String name)
  - (File file)
  - name和file是输出流的源，输出流将数据写入源中
  - 若目标文件不存在会创建该文件，若已存在会刷新该文件（使文件长度为0）
    - (String name, boolean append)
    - (File file, boolean append)
    - 指定文件已存在时是否刷新该文件，若append为true则不刷新文件，在文件末尾追加内容

以字节为单位写入：

- void write(int b)
  - (int b)：写入一个字节的数据
  - (byte b[])：写入b.length个字节的数据
  - (byte b[], int off, int len) ： 从b中off的位置开始写入len个字节到文件
  - FileOutputStream顺序地写入内容，直到流被关闭

```
fos = new FileOutputStream("C:\\example\\output.txt", true); // 设置为 true 时会追加到文件末尾

// 写入一个字节
fos.write(65);  // ASCII值65代表字符 'A'

// 写入字节数组
byte[] bytes = {66, 67, 68};  // 字节数组表示 'B', 'C', 'D'
fos.write(bytes);

// 写入字节数组的一部分
fos.write(bytes, 1, 2);  // 只写入字节数组的从索引 1 开始的 2 个字节，即 'C' 和 'D'
```

close 关闭流

完成对一个文件的操作后应调用`close()`方法关闭输入输出流，解除程序对文件的占用

文件字符流 FileReader/Writer

构造方法：

- FileReader
  - (File file)
  - (String filepath)
- FileWriter
  - (File file)
  - (String filepath)

字符输入输出流使用字符数组读写数据，即以字符为基本单位处理数据

常用方法：

- int read
  - ()：读取一个字符，返回int类型的Unicode字符值，若到文件末尾则返回-1
  - (char b[])：读取b.length个字符并存入b中
  - (char b[], int off, int len)：读取len个字符，从b中off的位置存入
- void write
  - (int n)或(char c)：存入一个字符
  - (char c[])：存入一个字符数组
  - (char c[], int off, int len)：从数组中off的位置开始存入len个字符到文件

```
try (
    FileReader fr = new FileReader("C:\\example\\input.txt");
    FileWriter fw = new FileWriter("C:\\example\\output.txt")
    ) {
    // 读取字符并写入到另一个文件
    int c;  // 用于存储每次读取的字符
    while ((c = fr.read()) != -1) {
        fw.write(c);  // 将字符写入输出文件
    }

    // 使用字符数组读取并写入
    var buffer = new char[1024];  // 缓冲区
    int len;
    while ((len = fr.read(buffer)) != -1) {
        fw.write(buffer, 0, len);  // 将缓冲区中的字符写入输出文件
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

缓冲流 BufferedReader/Writer

缓冲流需要搭配字符流使用，增强了读写文件的能力，且通过缓冲区的方式来减少磁盘操作次数，提高性能

- BufferedReader(Reader in)
- BufferedWriter(Writer out)

常用方法：

- String readLine()：读取一行文本，返回一个字符串，不包含换行符，若到文件末尾则返回null
- void newLine()：写入一个换行符
- write(String str)：写入一个字符串
- write(char buff[], int off, int len)：从buff中off的位置开始写入len个字符到文件
- close()

```
String inputFilePath = "C:\\example\\input.txt";
String outputFilePath = "C:\\example\\output.txt";

// 使用缓冲流读取和写入文件
try (
    BufferedReader br = new BufferedReader(new FileReader(inputFilePath));
    BufferedWriter bw = new BufferedWriter(new FileWriter(outputFilePath))
    // 将缓冲流的创建放在try中可以在程序执行完成try语句后自动关闭流
) {
    String line;
    while ((line = br.readLine()) != null) {
        // 读取一行并写入到输出文件
        bw.write(line);
        bw.newLine();  // 写入换行符
    }
    // 将input中的内容复制到output中
} catch (IOException e) {
    e.printStackTrace();
}
```

随机流 RandomAccessFile

随机流可以对文件中的任意位置进行读写操作

构造方法：

- RandomAccessFile(File file, String mode)，mode可取`r`和`rw`，表示只读或读写权限
- RandomAccessFile(String filepath, String mode)
- 随机流的指向既可以读也可以写

常用方法：

- seek(long a)：将随机流的指向移动到文件的第a个字节处
- getFilePointer()：获取随机流当前指向位置

- read()：读取一个字节，返回int
- readChar()：读取一个字符
- readInt()：读取一个int
- readLine()：读取一行文本
    - 不适用于非ASCII码，即中文字符，会出现乱码
- skipBytes(long n)：跳过n个字节
- write(byte b[])：写入一个字节
    - write操作会覆盖当前位置的数据
- writeChar(char c)：写入一个字符
- writeChars(String s)：写入一个字符串
- writeInt(int i)：写入一个int

```
try (RandomAccessFile file = new RandomAccessFile("example.txt", "rw")) {
    // 先读取文件的前 10 个字节
    var bytes = new byte[10];
    file.read(bytes);
    System.out.println("读取的数据: " + new String(bytes));

    // 将文件指针移动到第 5 个字节位置
    file.seek(5);
    // 从第 5 个字节开始写入 "Hello"
    file.write("Hello".getBytes());

    // 将文件指针移动到文件末尾
    file.seek(file.length());
    // 在文件末尾追加 " World!"
    file.write(" World!".getBytes());

    // 再次读取文件的全部内容
    file.seek(0); // 将文件指针移回到文件开头
    var fullBytes = new byte[(int) file.length()];
    file.read(fullBytes);
    System.out.println("文件的最终内容: " + new String(fullBytes));

} catch (IOException e) {
    e.printStackTrace();
}
```

文件锁 FileLock

在多个进程或线程访问同一个文件时，使用文件锁防止多个程序或线程同时对同一个文件进行写操作，确保在某个时刻只有一个进程或线程能够对文件进行操作，防止并发访问导致数据的不一致或冲突

RandomAccessFile创建的流在进行读写时可以使用文件锁，只要不解除该锁，其他程序就无法访问该文件

```
try {
    RandomAccessFile file = new RandomAccessFile("example.txt", "rw");
    FileChannel channel = file.getChannel();
```

```
        // 尝试获取文件锁
        FileLock lock = channel.tryLock();

        if (lock != null) {
            System.out.println("文件锁已成功获取");

            // 执行文件读写操作
            file.seek(0);  // 移动文件指针到文件开头
            file.write("Hello, world!".getBytes());  // 向文件写入数据

            // 模拟文件操作，暂停 2 秒
            Thread.sleep(2000);
        } else {
            System.out.println("文件已被锁定，无法获取文件锁");
        }

    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    } finally {
        // 显式释放文件锁和关闭资源
        try {
            if (lock != null && lock.isValid()) {
                lock.release();  // 释放文件锁
                System.out.println("文件锁已释放");
            }
            if (channel != null) {
                channel.close();  // 关闭文件通道
            }
            if (file != null) {
                file.close();  // 关闭文件
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

第十四章 JDBC与MySQL数据库

MySQL数据类型

整型

以下是将表示范围改为科学计数法后的表格：

以下是将表示范围改为 **2 的指数幂形式** 并将行内容居中的表格：

类型名	表示范围	用途	占用空间
INT/INTEGER	±（2^{31}）（有符号）	存储标准范围的整数	4 字节

类型名	表示范围	用途	占用空间
BIGINT	±（2^{63}）（有符号）	存储非常大的整数	8 字节

浮点型

类型名	表示范围	用途	占用空间
FLOAT	单精度浮点数（约 7 位有效数字）	存储低精度小数	4 字节
DOUBLE	双精度浮点数（约 15 位有效数字）	存储高精度小数	8 字节
DECIMAL(精度, 标度)	自定义精度和标度，精度为总长度，标度为小数长度	精确小数（如财务数据）	由定义的精度和标度决定

字符串

类型名	表示范围	用途	占用空间
CHAR(n)	定长字符串，最多存储 n 个字符（1 到 255）	存储固定长度的字符串	n 字节
VARCHAR(n)	变长字符串，最多存储 n 个字符（1 到 65535）	存储长度变化的字符串	实际长度 + 2 字节
TEXT	最大 65,535 字节	存储中等大小文本	实际大小 + 2 字节
LONGTEXT	最大 4,294,967,295 字节	存储超大文本	实际大小 + 4 字节
ENUM	从定义列表中选一个值（最多 65,535 个）	存储有限选项值	1 到 2 字节
SET	从定义列表中选多个值（最多 64 个）	存储有限组合值	1 到 8 字节

日期和时间

类型名	表示范围	用途	占用空间
DATE	`1000-01-01` 到 `9999-12-31`	存储日期	3 字节
DATETIME	`1000-01-01 00:00:00` 到 `9999-12-31 23:59:59`	存储日期和时间	8 字节
TIMESTAMP	`1970-01-01 00:00:01 UTC` 到 `2038-01-19 03:14:07 UTC`	记录时间戳，支持时区	4 字节
TIME	`-838:59:59` 到 `838:59:59`	存储时间（时:分:秒）	3 字节
YEAR	`1901` 到 `2155`	存储年份	1 字节

其他

类型名	表示范围	用途	占用空间
BOOL	`TRUE` 或 `FALSE`（内部存储为整数）	存储布尔值	1 字节
JSON	依赖实际数据大小	存储 JSON 格式的结构化数据	依赖实际数据大小

MySQL语句

创建表语法：

```
CREATE TABLE 表名 (
    列名 数据类型 [约束条件],
    ...
) [表选项];
```

在 SQL 中，`CREATE TABLE` 用于创建表。它的语法和参数可以根据不同的数据库（如 MySQL、PostgreSQL、SQL Server）有所不同，但通常支持以下主要参数：

基本语法

```
CREATE TABLE 表名 (
    列名 数据类型 [约束条件],
    ...
) ;
```

常见约束条件：

约束名	功能
PRIMARY KEY	定义主键，唯一标识表中的每一行，不能有重复值且不能为空。
UNIQUE	确保该列的值唯一，但允许存在多个 NULL 值。
NOT NULL	确保该列不能存储 NULL 值。
DEFAULT	为该列设置默认值，当插入数据时未指定该列值时，自动填充默认值。
CHECK	定义条件约束，确保列值符合指定条件（如 `CHECK (age >= 18)`）。
CHARACTER SET	指定字符集，如 `CHARACTER SET utf8`。

综合实现：

```
-- 创建数据库Book
create database Book;

-- 使用数据库Book
use Book;
```

```
-- 创建表Booklist
create table Booklist(
    ISBN varchar(100) primary key,
    name varchar(100) character set utf8,
    price decimal(10,2),
    publishdate date
);

-- 插入数据
insert into Booklist values('9787115428028','余胜军教你学Java',99.99,'2020-01-01');

-- 查询数据
select * from Booklist;

-- 删除表
drop table Booklist;

-- 删除数据库
drop database Book;
```

JDBC

即Java Database Connectivity，Java数据库连接，是用于操作数据库的API

连接MySQL数据库

导入jar包

在IDEA的 文件-项目结构-库 导入`mysql-connector-j-9.1.0.jar`

加载驱动

```
static void loadDriver(){
    try{
        Class.forName("com.mysql.cj.jdbc.Driver");   }
    catch(Exception e){
        System.out.println(e.getMessage());
    }
}
```

连接数据库

```
String uri = "jdbc:mysql://127.0.0.1:3306/new_schema?
useSSL=false&serverTimezone=GMT&characterEncoding=utf-8";
String user="root";
String password="1mTheB055@";
```

```
try{
    Connection con = DriverManager.getConnection(uri, user, password);
}catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

查询 更新数据

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

    String sql;
    try {
        // 建立数据库连接
        Connection con = DriverManager.getConnection(uri, user, password);

        // SQL查询语句
        sql = "SELECT * FROM Booklist WHERE column_name = 'ISBN'";
        // 使用Statement对象执行查询
        // 使用try-with-resources保证自动释放资源
        try (Statement stmt = con.createStatement()) {
            // 执行查询并返回结果集
            try (ResultSet rs = stmt.executeQuery(sql)) {
                printResultSet(rs);
            }
        } catch (SQLException e) {
        e.printStackTrace();
        }

        // SQL更新语句
        // 更新指定 ISBN 的书的价格
        sql = "UPDATE Booklist SET price = 9.99 WHERE ISBN = '9787115428028'";
        try (Statement stmt = con.createStatement()) {
            int rowsUpdated = stmt.executeUpdate(sql);
            // 返回值，即rowsUpdated，为执行UPDATE操作受影响的行数
            if (rowsUpdated > 0) {
                System.out.println("书的价格更新成功！");
            } else {
                System.out.println("没有找到匹配的书，更新失败！");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }

        // SQL更新语句，使用预处理
        // 更新指定 ISBN 的书的价格
        sql = "UPDATE Booklist SET price = ? WHERE ISBN = ?";
        try (PreparedStatement stmt = con.prepareStatement(sql)) {
            // 设置参数
            stmt.setDouble(1, 9.99);  // 设置新的价格
            stmt.setString(2, "9787115428028");  // 设置要更新的书的 ISBN
```

```
            // 执行更新
            int rowsUpdated = stmt.executeUpdate();
            if (rowsUpdated > 0) {
                System.out.println("书的价格更新成功！");
            } else {
                System.out.println("没有找到匹配的书，更新失败！");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }

        con.close(); // 关闭连接

    } catch (SQLException e) {
        e.printStackTrace();
    }
}

static void loadDriver(){
    //加载驱动
    try{
        Class.forName("com.mysql.cj.jdbc.Driver");   }
    catch(Exception e){
        System.out.println(e.getMessage());
    }
}

static void printResultSet(ResultSet rs) throws SQLException {
    while (rs.next()) {
        String ISBN = rs.getString(1);
        String name = rs.getString(2);
        System.out.printf("ISBN: %s, 书名: %s\n", ISBN, name);
    }
}
```

事务

保证数条SQL语句要么全部执行成功，要么全部不执行

```
con.setAutoCommit(False); // 关闭自动提交
stmt = con.createStatement(); // 先关闭自动提交再获取stmt
try{
    int rows1 = stmt.executeUpdate(sql1); // 执行第一条SQL语句
    int rows2 = stmt.executeUpdate(sql2);
    if(rows1 != 0 && rows2 != 0){ // 两条sql语句都执行成功
        con.commit(); // 提交事务
    }else{
        con.rollback(); // 回滚事务
    }
}catch(SQLException e){
```

```
        try{
            con.rollback(); // 发生异常时回滚事务
        }catch(SQLException ex){
            System.out.println(ex.getMessage());
        }
    }
```

第十五章 Java多线程

程序、进程、线程

程序 Program

- 程序是一个静态的文件或代码集合，通常是源代码（.java文件）或者编译后的字节码文件（.class文件）。
- 程序本身并不会执行，只有在运行时被加载到内存中，成为进程的一部分后，才能执行。

进程 Process

- 进程是正在执行的程序的实例，包含程序代码、数据、堆栈、文件描述符等资源。
- 每个进程都有自己独立的内存空间，因此不同的进程之间相互隔离。
- 在 Java 中，启动一个应用程序时，JVM 会为这个应用程序创建一个进程。

举例：当运行一个 Java 程序时，JVM 会启动一个进程来执行这个程序，并为这个进程分配内存。

线程 Thread

- 线程是进程中的一个执行单元，是程序执行的最小单位。
- 每个进程至少有一个主线程（通常是 `main` 方法所运行的线程），而进程中的多个线程可以共享进程的内存空间（如堆和方法区），但每个线程有自己的栈空间。
- 线程是程序执行的基本单位，它在进程内执行任务，多个线程可以并发执行，从而提高程序的执行效率。
- 当程序中所有的线程都被结束后 JVM 才会结束 Java 程序。

举例：在 Java 中，可以创建多个线程来并行执行任务，常见的方法有继承 `Thread` 类或实现 `Runnable` 接口。

对一个n核cpu，每个核都可以独立地处理一个或多个线程（超线程技术），在同一时刻可以同时处理n个（或2n等）线程。
对于每个核心，每个线程都会被在短暂执行后快速切换到下一个线程执行，保证每个线程都有机会被执行。

线程的状态和生命周期

使用`Thread`类及其子类表示线程，使用`getState()`方法获取枚举类型状态`Thread.State`的值：

`Thread`类的构造方法：

- Thread()：通过重写`run()`方法指定任务
- Thread(Runnable target)：通过实现`Runnable`接口的`run()`方法指定任务
- Thread(String name)：创建一个新的线程对象，并指定线程的名称

- Thread(Runnable target, String name)：创建一个新的线程对象，并指定线程的名称和任务

线程的状态：

- NEW 新建
  - 一个`Thread`类或其子类的对象被创建，获得了内存空间和其他资源，单还未调用`start()`方法启动线程
- RUNNABLE 可运行
  - NEW状态的线程调用`start()`方法启动线程
    - 仅NEW状态的线程可以调用`start()`方法，否则触发异常
  - JVM将线程放入可运行队列，等待CPU调度执行
  - JVM将CPU使用权切换给当前RUNNABLE线程时，调用`run()`方法执行线程
    - **需要重写父类中的`run()`方法**
- BLOCKED WAITING TIMED_WAITING 中断
  - 当JVM将CPU使用权切换给其他线程时，当前线程进入BLOCKED状态，等待JVM解除BLOCKED状态重新进入RUNNABLE状态
  - 线程执行期间调用`sleep(int millsecond)`方法会立刻让出CPU使用权指定的时间，并在此期间进入TIMED_WAITING状态，等待时间结束后重新进入RUNNABLE状态
  - 线程执行期间调用`wait()`方法会立刻让出CPU使用权，并在此期间进入WAITING状态，等待其他线程调用`notify()`或`notifyAll()`方法唤醒，唤醒后重新进入RUNNABLE状态
- TERMINATED 死亡
  - 线程执行完`run()`方法，结束运行

```
public class Main {

    public static void main(String[] args) {
        // 创建一个线程并启动
        MyThread myThread = new MyThread();

        // 打印线程初始状态（NEW）
        System.out.println("Thread state after creation: " + myThread.getState());

        // 启动线程
        myThread.start();

        // 打印线程状态（RUNNABLE）
        // 在这里，由于线程的执行非常快，可能无法准确捕捉线程状态
        // 线程状态会经历 RUNNABLE 和其他状态，如 BLOCKED 等
        try {
            Thread.sleep(100); // 主线程休眠，给子线程一些时间
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 打印线程当前状态
        System.out.println("Thread state after start: " + myThread.getState());

        // 等待线程执行完毕
        try {
            myThread.join();  // 等待myThread线程结束
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // 打印线程结束后的状态（TERMINATED）
            System.out.println("Thread state after completion: " +
    myThread.getState());
        }

        // 自定义线程类
        static class MyThread extends Thread {
            @Override
            public void run() {
                try {
                    // 模拟一些任务
                    System.out.println("Thread is running...");
                    Thread.sleep(500);  // 使线程进入TIMED_WAITING状态
                    System.out.println("Thread finished sleeping...");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

输出：

```
Thread state after creation: NEW
Thread is running...
Thread state after start: TIMED_WAITING
Thread finished sleeping...
Thread state after completion: TERMINATED
```

使用Runnable接口创建线程

Runnable接口：

```
public interface Runnable {
    void run();
}
```

示例：

```
public class Main {

    public static void main(String[] args) {
        // 创建一个Runnable对象
```

```
        MyRunnable task = new MyRunnable();

        // 创建一个Thread对象，并将Runnable对象传入
        Thread thread = new Thread(task);

        // 启动线程
        thread.start();

        // 主线程执行其他任务
        System.out.println("Main thread is running...");
    }

    // 实现Runnable接口
    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            // 线程执行的任务
            System.out.println("Thread is running...");
            try {
                Thread.sleep(2000);  // 模拟任务执行，休眠2秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread finished.");
        }
    }
}
```

输出：

```
Main thread is running...
Thread is running...
Thread finished.
```

前两行的顺序可能不一样，因为 **线程的执行顺序是不可预测的，取决于操作系统的调度**

线程run()方法中的局部变量

若多次将同一个实现了Runnable接口的类传递给同一个Thread类或其子类创建线程，则创建的多个线程相互隔离，每个都有独立的局部变量，互不干扰，除非使用了共享的资源，如静态变量

```
public class RunnableExample {
    public static void main(String[] args) {
        // 创建 Runnable 实现
        MyRunnable task = new MyRunnable();

        // 创建多个线程，传入相同的 Runnable 实现
        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);
```

```
        Thread thread3 = new Thread(task);

        // 启动多个线程
        thread1.start();
        thread2.start();
        thread3.start();
    }

    // 实现 Runnable 接口
    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            // 每个线程执行时都有自己独立的局部变量
            int localVar = 0;
            localVar++;
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is running with localVar: " +
localVar);
        }
    }
}
```

输出：

```
Thread-2 is running with localVar: 1
Thread-1 is running with localVar: 1
Thread-0 is running with localVar: 1
```

线程的同步

synchronized 是 Java 中用来控制多线程并发访问共享资源的一种机制，它保证了**同一时刻只有一个线程能够执行**被 synchronized 修饰的代码块或方法，进而避免了多线程同时访问共享资源导致的数据不一致问题。

同步实例方法

同步实例方法会锁定当前对象`this`，确保同一时刻只有一个线程能执行该方法。

当一个线程执行 increment() 方法时，其他线程不能执行同一个对象的任何被 synchronized 修饰的方法，直到当前线程执行方法完毕。

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
```

```
    }
}
```

同步静态方法

如果同步的方法是静态的，锁的对象将是**类本身**，而不是实例对象。

当一个线程执行被 synchronized 修饰的静态方法时，所有当前类的对象中被 synchronized 修饰的静态方法都不能被其他线程执行，直到当前线程执行完毕。

```
class Counter {
    private static int count = 0;

    public synchronized static void increment() {
        count++;
    }

    public synchronized static int getCount() {
        return count;
    }
}
```

同步代码块

```
class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) { // 锁定this
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}
```

increment() 方法中的 synchronized (this) 语句块锁定了 this，即当前对象。只有获得该锁的线程才可以执行 count++ 操作。

指定锁对象

```
class Counter {
    private int count = 0;
    private final Object lock = new Object();  // 定义一个锁对象
```

```
    public void increment() {
        synchronized (lock) { // 锁定lock
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}
```

可以用任意对象作为锁。锁对象可以是任意类型的对象，常见的做法是定义一个专门的锁对象。

这里的 lock 对象是专门用于同步的对象，所有想要访问 increment() 方法的线程都必须先获取 lock 对象的锁。

这样可以更精细地控制哪些代码块需要同步，而不影响其他部分的并发。

第十六章 Java网络基础

URL类

java.net中的URL(Uniform Resource Locator, 统一资源定位符)类提供了使用URL对象获取URL资源的方法。

一个URL对象至少包括：

- 协议：如http、https、ftp等
- 地址：有效的域名或IP地址，如www.baidu.com，39.156.66.10
- 资源：主机上的任意文件

URL类构造方法

```
import java.net.*;

// 使用域名初始化一个URL对象
public URL(String spec) throws MalformedURLException

try {
    var url = new URL("http://www.baidu.com/");
}catch(MalformedURLException e) {
    System.out.println("URL格式错误，Bad URL: " + e);
}

// 指定协议、地址和资源
public URL(String protocol, String host, String file) throws MalformedURLException
```

读取URL中的资源

对URL对象调用`InputStream()`和`openStream()`方法以返回一个输入流，指向URL资源。

受网络等因素影响，读取URL资源可能很慢，因此需要将其放在一个线程中进行，避免阻塞主线程。

```
import java.io.*;
import java.net.*;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        // 启动新线程来读取URL
        String urlString = "https://www.baidu.com/";
        var urlReaderThread = new URLReaderThread(urlString);
        urlReaderThread.start(); // 启动线程
    }

    // 在新的线程中读取URL
    static class URLReaderThread extends Thread {
        private String urlString;

        public URLReaderThread(String urlString) {
            this.urlString = urlString;
        }

        @Override
        public void run() {
            // 执行读取URL的任务
            readFromURL(urlString);
        }
    }

    public static void readFromURL(String urlString) {
        try {
            // 创建URL对象
            URL url = new URL(urlString);

            // 打开输入流
            InputStream inputStream = url.openStream();

            // 使用Scanner读取数据
            Scanner scanner = new Scanner(inputStream, "UTF-8");

            // 按行读取数据
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);  // 打印每一行内容
            }

            // 关闭Scanner和流
            scanner.close();
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

InetAddress类

地址的表示

Internet上的主机使用域名或IP地址表示地址，`java.net`中的`InetAddress`类中含有一个主机的域名和IP地址，如`www.baidu.com/39.156.66.10`。

获取地址

使用`getByName()`方法获取指定主机的地址，getByName是java.net包中InetAddress类的静态方法，可以直接使用`InetAddress.getByName()`调用而不需要创建`InetAddress`的对象。

```
import java.net.*;

public class Main{
    public static void main(String[] args){
        try{
            InetAddress address_1 = InetAddress.getByName("www.baidu.com");
            System.out.println(address_1.toString());
            InetAddress address_2 = InetAddress.getByName("47.103.24.173");
            System.out.println(address_2.toString());
            // 如果对IP地址使用getByName方法，需要该IP地址的DNS服务器设置PTR记录，也就是允许根据IP地址反向查询域名
            // 否则无法根据IP地址获取到域名，这个与DNS服务商有关
        }catch (UnknownHostException e){
            System.out.println("查找失败");
            e.toString();
        }
    }
}
```

输出：

```
www.baidu.com/110.242.68.3
47.103.24.173
```

`InetAddress`类中还有两个实例方法：

- `getHostName()`：获取域名
- `getHostAddress()`：获取IP地址

套接字 Socket

`Socket`，套接字，用于表示网络通信中的一个端点，每个套接字由 **IP地址** 和 **端口号** 组成。

客户端 Socket

在客户端创建`Socket`对象，指定目标服务器的IP地址和端口号。构造方法为：`Socket(String host, int port)`。

```
try{
    var mysocket = new Socket("http://192.168.1.1", 80);
}catch(IOException e){
    // ...
}
```

对`Socket`对象调用`getInputStream()`方法可以获得一个输入流，并从中读取host的信息，调用`getOutputStream()`方法可以获得一个输出流，并通过向这个输出流中写入信息来发送给host

服务器端 ServerSocket

在服务器端建立`ServerSocket`对象，构造方法为`ServerSocket(int port)`，监听指定的端口号。

```
try{
    var myserversocket = new ServerSocket(1880);
}catch(IOException e){
    // 若1880端口被占用则会抛出IOException异常
}
```

对`ServerSocket`对象调用`accept()`方法将`Socket`和`ServerSocket`连接起来：

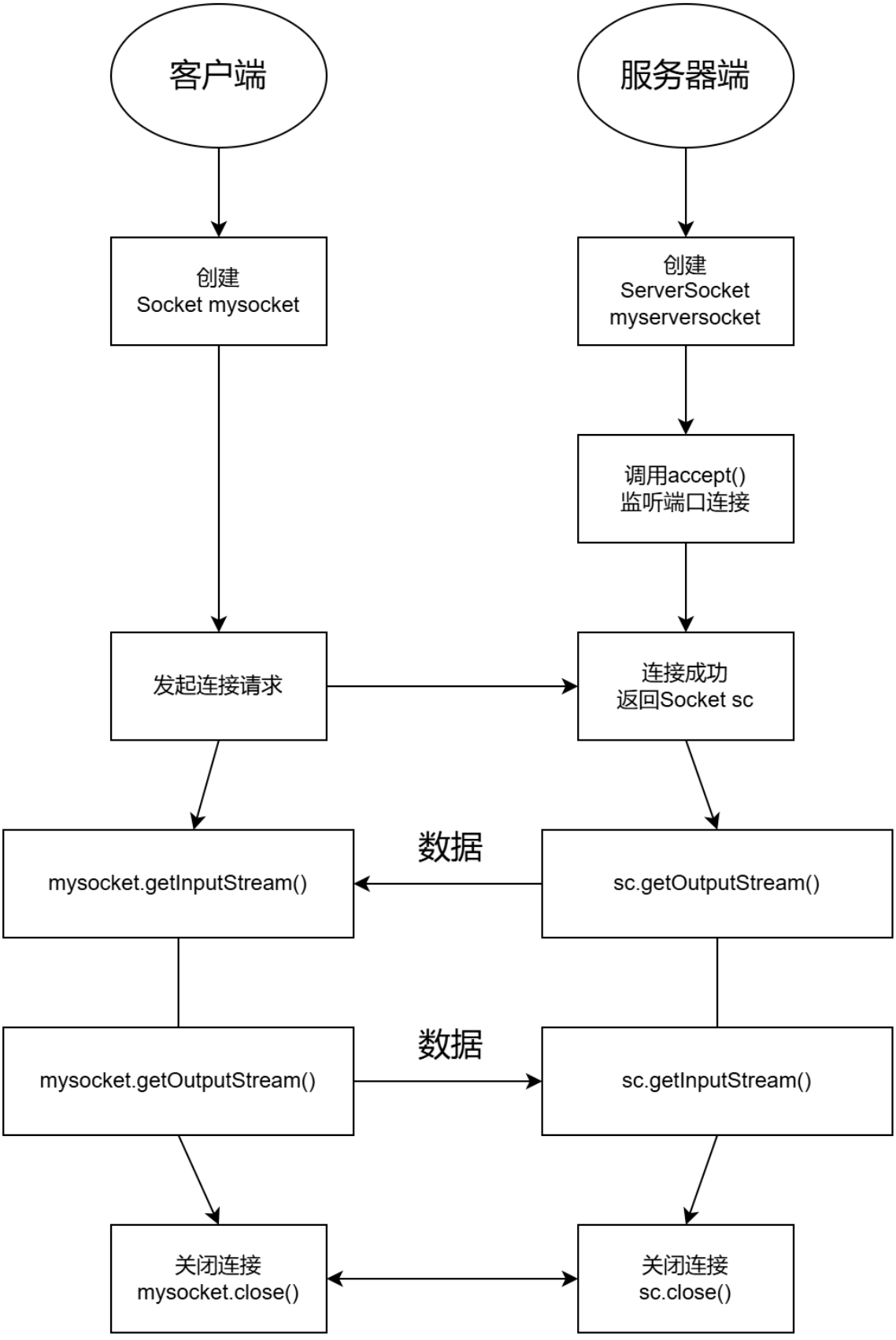
```
try{
    Socket sc = myserversocket.accept();
}catch(IOException e){
    // ...
}
```

建立连接

- 在客户端创建 `Socket mysocket`，在服务器端创建 `ServerSocket myserversocket`
- 调用 `myserversocket.accept()` 方法，等待 `mysocket` 连接
- 若客户端连接，则返回 `myserversocket` 与 `mysocket` 相连接的 `Socket` 对象 `sc`
- `sc.getOutputStream()` 方法获得的输出流指向 `mysocket.getInputStream()` 方法获得的输入流
- `mysocket.getOutputStream()` 方法获得的输入流指向 `sc.getInputStream()` 方法获得的输出流
- `mysocket` 或 `sc` 调用 `getInetAddress()` 方法获得服务器或客户端的IP地址和域名
- `mysocket` 或 `sc` 调用 `close()` 方法关闭套接字连接

`myserversocket.accept()` 方法会堵塞线程的执行，即 `accpet()` 被调用后，如果没有客户端与服务器端连接，`accept()` 就会一直等待，直到有客户端与其连接。

读取数据也会堵塞线程的执行，因为还没有发送数据时，另一端就可能已经开始读取数据了。


客户端
服务器端
创建
Socket mysocket
创建
ServerSocket
myserversocket
调用accept()
监听端口连接
发起连接请求
连接成功
返回Socket sc
数据
mysocket.getInputStream()
sc.getOutputStream()
数据
mysocket.getOutputStream()
sc.getInputStream()
关闭连接
mysocket.close()
关闭连接
sc.close()

程序实现

先启动Server，然后启动Client

Server:

```
import java.io.*;
import java.net.*;

public class Server {
    public static void main(String[] args) {
        try (ServerSocket myServerSocket = new ServerSocket(12345)) {
            System.out.println("服务器已启动，等待客户端连接...");
            Socket sc = myServerSocket.accept(); // 等待客户端连接
            System.out.println("客户端已连接: " + sc.getInetAddress());

            // 创建输入流
            BufferedReader input = new BufferedReader(new
InputStreamReader(sc.getInputStream()));

            // 接收服务器端发送的消息
            String serverMessage;
            while ((serverMessage = input.readLine()) != null) {
                System.out.printf("服务器端收到: %s%n", serverMessage);
            }

            // 关闭连接
            System.out.println("客户端结束连接");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Client:

```
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String[] args) {
        try (Socket mySocket = new Socket("localhost", 12345)) {
            System.out.println("已连接到服务器: " + mySocket.getInetAddress());

            // 创建输入输出流
            OutputStream outputStream = mySocket.getOutputStream();
            PrintWriter output = new PrintWriter(outputStream, true);

            // 客户端发送0到9
            for (int i = 0; i <= 9; i++) {
                output.println(i);
                System.out.printf("客户端发送: %d%n", i);
```

```
                Thread.sleep(1000); // 等待1秒
            }

            // 关闭连接
            mySocket.close();
            System.out.println("客户端关闭连接");

        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Server输出：

```
服务器已启动，等待客户端连接...
客户端已连接：/127.0.0.1
服务器端收到：0
服务器端收到：1
服务器端收到：2
服务器端收到：3
服务器端收到：4
服务器端收到：5
服务器端收到：6
服务器端收到：7
服务器端收到：8
服务器端收到：9
客户端结束连接
```

Client输出：

```
已连接到服务器：localhost/127.0.0.1
客户端发送：0
客户端发送：1
客户端发送：2
客户端发送：3
客户端发送：4
客户端发送：5
客户端发送：6
客户端发送：7
客户端发送：8
客户端发送：9
客户端关闭连接
```

UDP数据报

前文中`Socket`使用的是TCP连接

- TCP（传输控制协议）连接
  - 传输数据前需要建立连接，传输的每个数据包都需要接收端确认，传输结束后需要释放连接
  - 高可靠性，保证数据完整准确地传输
  - 流量、延迟开销大，实时性低
- UDP（用户数据报协议）连接
  - 传输数据无需建立连接，没有数据重传机制，不保证数据到达和按顺序到达，支持广播和多播
  - 实时性高，延迟低，资源消耗低
  - 数据传输不可靠，易出现丢包、重复、乱序

发送UDP数据包

使用`DatagramPacket`类封装数据包，构造方法为：

- `DatagramPacket(byte[] data, int length, InetAddress address, int port)`
  - 将数据`data`发送到`address:port`
- `DatagramPacket(byte[] data, int offset, int length, InetAddress address, int port)`
  - 将数据`data`从`offset`开始的`length`个字节发送到`address:port`

```
import java.io.IOException;
import java.net.*;

public class Main {
    public static void main(String[] args) throws UnknownHostException,
SocketException {
        byte[] data = "你好Java".getBytes(); // 数据内容
        var address = InetAddress.getByName("www.baidu.com"); // 获取目标域名的ip地
址
        var datapack = new DatagramPacket(data, data.length, address, 980); // 打
包数据包

        // 根据datapack获取其数据、ip、端口
        byte[] data2 = datapack.getData();
        InetAddress address2 = datapack.getAddress();
        int port2 = datapack.getPort();

        // 发送datapack
        try(var datasocket = new DatagramSocket()){ // 使用没有参数的构造方法
            datasocket.send(datapack); // 发送数据包
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

接收UDP数据包

- 用构造方法`DatagramSocket(int port)`创建用于接收数据包的类，其中`port`为发送的端口号
- 用构造方法`DatagramPacket(byte[] data, int length)`创建用于接收数据包的类，其中`data`为用于存储接收到的数据的字节数组，`length`为数据包中最多能接收的字节数

    - 数据包中数据长度不超过 **8mb**
- 用`receive(DatagramPacket pack)`方法接收数据包，将数据包存入`pack`中

```
import java.io.IOException;
import java.net.*;

public class Main {
    public static void main(String[] args) {
        var data = new byte[100]; // 存储数据的数组
        int length = 90; // 最大接收的数据大小，不大于数组的长度
        var datapack = new DatagramPacket(data, length); // 用于接收数据包的pack
        try(var datasocket = new DatagramSocket(12345)){ // 通过12345端口接收数据包
            datasocket.receive(datapack); // 接收数据包存入datapack

            // 根据datapack获取其数据、ip、端口
            byte[] data2 = datapack.getData();
            InetAddress address2 = datapack.getAddress();
            int port2 = datapack.getPort();
            int length2 = datapack.getLength();


            String receivedData = new String(datapack.getData(), 0,
datapack.getLength()); // 将接收到的数据转为String
            System.out.println("接收到的数据: " + receivedData);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

`receive()`方法会堵塞线程，直到接收到数据包

广播数据报