第 1 章：Java 编程

1.1 Java 语言入门

错题集

1.1.1 Java 语言特性

1. **面向对象（Object-Oriented）**：所有代码都写在类中，支持封装/继承/多态
2. **跨平台性（Write Once, Run Anywhere）**：Java 程序通过 JVM（Java Virtual Machine）运行，屏蔽了底层操作系统的差异
3. **自动内存管理（垃圾收集机制）**：使用垃圾回收器（Garbage Collector）回收不再使用的对象。
4. 异常处理机制完善：提供 try-catch-finally 结构来捕获异常，防止程序崩溃；同时支持 checked 和 unchecked 异常，提高程序健壮性。
5. 多线程支持良好：可以在同一个程序中创建多条线程，让多个任务并发执行（利用多核 CPU）。
6. 安全性强：Java 提供字节码校验、类加载机制、安全管理器等机制，确保运行环境的安全。

1.1.2 Java 内存问题

- **内存泄漏**：不再使用的对象由于仍然存在引用而导致无法被垃圾回收器回收，导致内存使用量逐渐增加，最终可能导致系统性能下降甚至崩溃。
  - 静态集合类：如果将对象添加到一个静态的集合类（如 ArrayList、HashMap 等）中，并且没有或忘记从集合中移除这些对象，则这些对象将不会被垃圾回收。
  - 未关闭的资源：比如数据库连接、文件流、网络连接等，如果没有正确关闭，相关的对象可能永远不会被垃圾回收。
  - 监听器和回调：注册监听器或回调接口后，若不及时注销，可能会造成内存泄漏。
  - 内部类持有外部类的引用：特别是非静态内部类会隐式地持有其外部类的引用，如果不小心管理，可能导致外部类不能被回收。
- **内存溢出**：当应用程序试图分配超过堆空间所能提供的内存时发生的错误
  - 堆大小不足：如果应用程序需要的内存超出了 JVM 设置的最大堆大小限制。
  - 频繁的大对象创建：例如，大量临时大对象短时间内被创建而未及时释放。
  - 无限递归调用：可能导致栈溢出异常（StackOverflowError）

1.1.3 Java 程序运行机制

Java 程序从源代码到最终执行，主要经历三个阶段：

1. 源代码阶段（写剧本）

开发者首先使用文本编辑器或集成开发环境编写 `.java` 源文件。

例如，创建一个名为 `HelloWorld.java` 的文件：

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

```
}
```

- `HelloWorld.java` 是 Java 源代码文件。
- `public class HelloWorld` 定义了一个名为 `HelloWorld` 的公共类。
- `main` 方法是该 Java 应用程序的入口；JVM 运行该程序时，会从 `main` 方法开始执行。

当一个顶层类（没有被定义在其他类或接口内部的类）被声明为 `public` 时，类名必须与文件名完全一致，并且 Java 区分大小写：

$$\texttt{public class HelloWorld} \quad \longleftrightarrow \quad \texttt{HelloWorld.java}.$$

2. 编译阶段（解读翻译剧本）

Java 编译器 `javac` 将开发者编写的 `.java` 源文件转换为 `.class` 字节码文件。

在终端中执行：

```
javac HelloWorld.java
```

如果编译成功，会生成：

$$\texttt{HelloWorld.java} \xrightarrow{\texttt{javac}} \texttt{HelloWorld.class}.$$

- `HelloWorld.java`：开发者编写的 Java 源代码文件。
- `javac`：Java 编译器，负责检查并编译 Java 源代码。
- `HelloWorld.class`：编译后生成的 Java 字节码文件。
- **字节码不是某一种 CPU 能够直接执行的机器码，而是一种由 JVM 理解执行的中间表示。**
- Java 文件并不一定必须包含 `main` 方法才能成功编译；但是，如果希望把该类作为独立程序运行，通常需要提供 `main` 方法。

Java 程序的标准入口方法为：

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

- `public`：JVM 可以从类的外部访问该方法。
- `static`：JVM 不需要先创建 `HelloWorld` 对象，就可以直接调用该方法。
- `void`：该方法不返回任何结果。
- `String[] args`：用于接收运行程序时传入的命令行参数。

3. 执行阶段

编译成功后，可以在终端中执行：

```
java HelloWorld
```

- `java` 命令启动 Java 虚拟机（Java Virtual Machine, JVM）。
- 命令中填写类名 `HelloWorld`，不填写 `.class` 后缀。
- JVM 通过类加载器**加载** `HelloWorld.class`。
  类**加载过程**总体可以表示为：

$$\text{加载} \rightarrow \text{连接} \rightarrow \text{初始化}.$$

其中，连接阶段又包括：

$$\text{验证} \rightarrow \text{准备} \rightarrow \text{解析}.$$

因此，类加载过程可以展开为以下五个步骤：

1) **加载（Loading）**
   - 类加载器从磁盘、网络或其他位置读取 `.class` 字节码文件。
   - JVM 将字节码中的类信息加载到内存中。
   - JVM 在内存中创建对应的 `java.lang.Class` 对象。
   - 该 `Class` 对象描述类的名称、字段、方法、父类和接口等信息。
2) **验证（Verification）**
   - 检查字节码文件的格式是否合法。
   - 检查字节码是否符合 JVM 规范。
   - 检查代码是否可能破坏 JVM 的安全性。
   - 防止损坏或恶意的字节码被直接执行。
3) **准备（Preparation）**
   - 为类的静态变量分配内存。
   - 将静态变量设置为相应数据类型的默认值。
   - `int` 的默认值是 `0`。
   - `boolean` 的默认值是 `false`。
   - 引用类型的默认值是 `null`。

```
public class Example {
    static int number = 10;
}
```

在准备阶段，JVM 会先为静态变量 `number` 分配内存，并将其设置为默认值：

```
number = 0;
```

此时还不会执行程序中显式写出的 `number = 10`。

4) **解析（Resolution）**

- 将字节码中的符号引用转换为直接引用。
- 确定程序所引用的类、字段和方法的实际目标。
- 例如，字节码中可能只记录某个方法的类名、方法名和参数信息；解析后，JVM 能够找到实际对应的方法。

5) **初始化（Initialization）**

- 执行静态变量的显式赋值语句。
- 执行类中的静态代码块。
- 将静态变量从默认值修改为程序指定的初始值。

对于前面的例子，初始化阶段才会执行：

```
static int number = 10;
```

此时，`number` 才会从默认值 `0` 变为程序指定的值 `10`。

注释

需要区分以下两个概念：

- `.class` 文件：存储在磁盘上的 Java 字节码文件。
- `Class` 对象：JVM 加载某个类后，在内存中创建的、用来描述该类信息的对象。

- 类加载和初始化完成后，JVM 找到 `HelloWorld` 类的 `main` 方法。
- JVM 调用 `main` 方法，开始执行程序中的代码；JVM 可以逐条解释执行字节码，也可以通过即时编译器（Just-In-Time Compiler, JIT）将频繁执行的字节码编译为本地机器码。
- JVM 通过操作系统使用线程、内存、文件、网络和 CPU 等底层资源。

错题集

Java 程序的运行过程可以概括为：

$$\texttt{.java}\ \text{源文件} \xrightarrow{\texttt{javac}} \texttt{.class}\ \text{字节码} \xrightarrow{\text{类加载器}} \text{JVM} \rightarrow \text{操作系统} \rightarrow \text{硬件平台}.$$

Java 通过将源代码编译为平台无关的字节码，再由不同操作系统上的 JVM 执行字节码，实现了**一次编写，到处运行**（Write Once, Run Anywhere）。

1.1.4 Java 垃圾回收机制

1. **标记阶段**：找出哪些对象已经“死亡”（不可达）
2. **清除阶段**：回收这些对象占用的内存空间
3. **整理阶段**（非必须）：移动存活对象，避免内存碎片

1.1.5 Java 基础语法

1. Java 程序的基本结构

一个最简单的 Java 程序如下：

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

- `public class HelloWorld`：定义了一个名为 `HelloWorld` 的公共类。
- `public static void main(String[] args)`：定义了 Java 应用程序的入口方法。
- `System.out.println(...)`：向控制台输出一行内容。
- **花括号** `{}`：用于界定类、方法和控制结构的代码块。
- **分号** `;`：表示一条语句的结束。

Java 程序具有以下基本层次：类 $\supset$ 方法 $\supset$ 语句.

在上面的程序中：

- `HelloWorld` 是类。
- `main` 是定义在类中的方法。
- `System.out.println(...);` 是定义在方法中的语句。

2. `public class` 与 `class` 的区别

在 Java 中，顶层类既可以声明为公共类，也可以不写访问修饰符。

1) `public class`

- 表示公共顶层类。
- 该类可以被其他包中的代码访问。
- 类名必须与源文件名完全一致，并且区分大小写。
- **一个** `.java` **文件中最多只能有一个** `public` 顶层类。

例如，文件名为 `Person.java`：

```
public class Person {
}
```

此时，公共类名和文件名必须满足：

$$\texttt{public class Person} \quad \longleftrightarrow \quad \texttt{Person.java}.$$

2) `class`

如果顶层类前面没有写 public，它具有包级可见性，也称为 package-private。

- **只能被同一个包中的代码访问**。
- 类名不强制要求与源文件名一致。
- 一个 .java 文件中可以**定义多个** package-private 顶层类。

例如，文件名为 Person.java：

```
public class Person {
    // 公共顶层类
}

class Student {
    // package-private 顶层类
}

class Teacher {
    // package-private 顶层类
}

```

其中：

- Person、Student 和 Teacher 都是顶层类。
- Person 是公共顶层类，因此文件必须命名为 Person.java。
- Student 和 Teacher 没有使用 public，因此只在当前包内可见。
- 一个源文件中不能定义两个 public 顶层类。

虽然 Java 允许在一个文件中定义多个非公共顶层类，但实际开发中通常建议：

- 每个 *.java* 文件主要定义一个顶层类。
- 文件名与主要类名保持一致。
- 将不同功能的类分别放入不同的源文件中。

3. 注释的使用

注释是写给开发者阅读的说明文字，通常不会被当作程序代码执行。

Java 主要支持以下三种注释。

1) 单行注释

单行注释以 // 开始，一直持续到当前行末尾。

```
// 这是一条单行注释
System.out.println("Hello World"); // 输出一行文字

```

- 只能作用于当前行。
- 适合编写简短说明。
- 可以单独占一行，也可以放在语句后面。

2) **多行注释**

多行注释以 /* 开始，以 */ 结束。

```
/*
这是一个
多行注释
*/
System.out.println("Hello World");
```

- 注释内容可以跨越多行。
- 适合编写较长的解释。
- 也可以临时注释掉一段代码。
- 普通多行注释通常不能相互嵌套。

3) **文档注释（Javadoc）**

文档注释以 /** 开始，以 */ 结束。

```
/**
 * 表示一个学生。
 *
 * @author Zhang San
 * @version 1.0
 */
public class Student {
}

```

文档注释主要用于：

- 描述类和接口。
- 描述字段和构造方法。
- 描述方法的功能、参数、返回值和异常。
- 通过 javadoc 工具生成 HTML 格式的 API 文档。

常见的 Javadoc 标签包括：

- `@author`：说明作者信息。
- `@version`：说明版本信息。

- `@param`：说明方法参数。
- `@return`：说明方法返回值。
- `@throws` 或 `@exception`：说明方法可能抛出的异常。

带有文档注释的方法示例：

```java
/**
 * 获取用户的年龄。
 *
 * @param userId 用户 ID
 * @return 用户的年龄
 * @throws Exception 当用户不存在时抛出异常
 */
public int getAge(int userId) throws Exception {
    // 方法体
    return 18;
}
```

其中：

- `@param userId` 描述参数 `userId` 的含义。
- `@return` 描述方法返回的结果。
- `@throws Exception` 描述方法可能抛出的异常。

4. **API 文档**：API（Application Programming Interface）文档是 Java 提供的一套标准说明文档，记录了 Java 所有**类**、**接口**、**方法**、**参数**、**返回值**等内容（Link：https://docs.oracle.com/javase/8/docs/api/）

1.2 Java 基本语法

错题集

1.2.1 标识符

1. 标识符的定义

标识符（identifier）是开发者为程序中的组成部分指定的名称，主要用于标识：变量；方法；类；接口；包；常量；参数。

```
int age;
String studentName;

public class Student {
}
```

其中：

- age 是变量名，也是一个标识符；
- studentName 是变量名，也是一个标识符；
- Student 是类名，也是一个标识符；
- int、String 和 public 不是这里新定义的标识符。

2. 标识符的强制命名规则

标识符必须遵守 Java 的语法规则，否则程序无法通过编译。

1) **允许使用的字符**

标识符通常可以包含：

- 英文字母，如 A--Z 和 a--z；
- 数字，如 0--9；
- 下划线 _；
- 美元符号 $；
- *Java 允许的其他 Unicode 字母。*

Java 标识符并不严格局限于 ASCII 英文字母，但实际开发中通常仍推荐使用英文命名。

2) **不能以数字开头**

数字可以出现在标识符中，但不能作为第一个字符。

```
int age1;       // 合法
int student2;   // 合法
int 1age;       // 错误：以数字开头
```

3) **不能包含空格**

一个标识符必须是连续的字符序列。

```
String userName;    // 合法
```

```
String user_name;   // 合法
String user name;   // 错误：包含空格
```

4) **不能包含非法符号**

除**下划线**和**美元**符号外，连字符、井号等符号通常不能出现在标识符中。

```
int studentAge;     // 合法
int student_age;    // 合法
int student-age;    // 错误：包含连字符 -
int student#age;    // 错误：包含 #
```

美元符号虽然可以合法地出现在标识符中：

```
int price$;
int $count;
```

但实际开发中通常不推荐主动使用 `$`，因为：

- 容易降低代码可读性；
- Java 编译器和其他工具生成的名称中可能包含 `$`；
- 容易与工具自动生成的内部名称混淆。

5) **不能使用 Java 关键字或保留字**

例如，不能将 `class`、`public`、`int`、`for` 等关键字作为变量名或方法名。

```
int count;      // 合法
int number;     // 合法
int for;        // 错误：for 是关键字
int class;      // 错误：class 是关键字
```

6) **严格区分大小写**

Java 是大小写敏感的语言：`age`、`Age` 和 `AGE` 是三个不同的变量名。

```
int age;
int Age;
int AGE;
```

7) **单独的下划线不能作为标识符**

在较新的 Java 版本中，单独的 `_` 不能作为变量名，但以下名称仍然合法：

```
int _age;
int student_;
```

3. 标识符的命名规范

命名规范通常不是编译器强制执行的语法规则，但遵守规范可以提高代码的可读性和一致性。

1) 包名

包名通常采用以下规范：

- 所有字母小写；
- 多级包名使用点号分隔；
- 顶级包通常采用公司或组织域名的倒置形式。

例如：

```
package com.example.chapter02;
package org.apache.commons.lang;
```

不推荐：

```
package Com.Example.Chapter02;
```

2) 类名与接口名

类名和接口名通常采用大驼峰命名法（PascalCase）：

- 每个单词的首字母大写；
- 单词之间不使用下划线；
- 名称通常使用名词或名词短语；
- 除通用缩写外，尽量避免难以理解的缩写。

正确示例：

```
public class HelloWorld {
}

public class UserLoginService {
}

public interface PaymentService {
}
```

不推荐：

```
public class helloWorld {
}

public class userlogin {
}
```

```
6
7 public class user_login_service {
8 }
```

3) **变量名**

变量名通常采用小驼峰命名法（camelCase）：

- 第一个单词的首字母小写；
- 后续单词的首字母大写；
- 名称应能够表达变量保存的数据；
- 布尔变量通常以 `is`、`has` 或 `can` 开头。

例如：

```
1 int age;
2 String userName;
3 double interestRate;
4 boolean isStudent;
5 boolean hasPermission;
6 boolean canLogin;
```

不推荐：

```
1 int Age;              // 首字母不应大写
2 String UserName;      // 更适合写为 userName
3 double interest_rate; // Java 中通常不使用下划线命名普通变量
```

4) **方法名**

方法名同样采用小驼峰命名法，但通常**以动词或动词短语开头**，表示该方法执行的动作。

例如：

```
1 void printMessage() {
2 }
3
4 String getUserName() {
5     return "Alice";
6 }
7
8 double calculateInterest() {
9     return 1.0;
10 }
11
```

```
boolean isValid() {
    return true;
}

```

不推荐：

```
void GetUserName() {
}

void UserName() {
}
```

5) **常量名**

常量通常具有以下特点：

- 使用 `final` 声明；
- 所有字母大写；
- 多个单词之间使用下划线连接；
- 这种命名方式也称为 UPPER_SNAKE_CASE。

例如：

```
final int MAX_AGE = 100;
final double PI = 3.1415926;
final int MAX_LOGIN_ATTEMPTS = 5;

```

不推荐：

```
final int max_age = 100;
final int MaxAge = 100;
final int maxAge = 100;
```

4. Java 关键字与保留字

关键字是 Java 语言已经赋予特殊语法意义的单词，因此不能作为普通标识符使用。

常见关键字可以按照用途分类：

- **访问控制：** `public`、`private`、`protected`
- **基本数据类型：** `byte`、`short`、`int`、`long`、`float`、`double`、`char`、`boolean`
- **分支循环：** `if`、`else`、`switch`、`case`、`default`、`while`、`do`、`for`、`break`、`continue`、`return`
- **异常处理：** `try`、`catch`、`finally`、`throw`、`throws`

- **类与对象：** `class`、`interface`、`extends`、`implements`、`new`、`this`、`super`
- **修饰符：** `static`、`final`、`abstract`、`synchronized`、`volatile`、`transient`
- **包管理：** `package`、`import`
- **其他：** `void`、`enum`、`assert`、`instanceof`

还需要注意：

- `goto` 和 `const` 被保留，不能作为标识符使用，但 Java 当前没有使用它们实现对应语法。
- `true`、`false` 和 `null` 是特殊字面量，也不能作为标识符使用。
- 部分较新的 Java 语法还使用受限关键字或上下文关键字；是否可用可能取决于具体语法位置和 Java 版本。

1.2.2 变量

1. 变量的定义与使用

变量（variable）是程序中用于保存数据的命名存储位置。一个变量主要包含三个要素：

- **数据类型：**决定变量可以保存什么类型的数据，以及可执行哪些操作。
- **变量名：**用于在程序中引用该变量，必须符合标识符规则。
- **变量值：**当前保存在变量中的数据。

变量的基本语法为：

```
数据类型 变量名;
数据类型 变量名 = 初始值;
```

例如：

```
int age;
age = 18;
double score = 95.5;
```

其中：

- `int` 和 `double` 是**数据类型**；
- `age` 和 `score` 是**变量名**；
- `18` 和 `95.5` 是**变量值**。

使用变量时需要遵守以下规则：

1) 变量必须**先声明**，之后才能使用。
2) 赋给变量的值必须与变量的**数据类型兼容**。
3) 局部变量必须在第一次读取之前完成初始化（**变量有值才能被读取**）。
4) 变量**只能在其作用域内访问**，同一作用域内不能声明两个同名的局部变量。

例如：

```
public static void main(String[] args) {
    int number = 10;
    System.out.println(number);

    {
        int value = 20;
        System.out.println(value);
    }

    // System.out.println(value); // 错误：value 已超出作用域
}
```

2. 变量的分类

根据声明位置和所属对象，Java 变量主要可以分为以下几类。

1) 实例变量

实例变量声明在类中、方法和代码块之外，并且没有使用 `static` 修饰。

- 属于某个具体对象，**每个对象拥有自己独立的一份实例变量**
- 创建对象时会被自动初始化
- **生命周期通常与对象一致**

```
class Student {
    int age; // 实例变量
    String name; // 实例变量
}
```

2) 静态变量

静态变量声明在类中、方法和代码块之外，并使用 `static` 修饰。

- 属于类本身，而不是某个具体对象，**同一个类的所有对象共享同一份静态变量**
- **类加载时会被自动初始化**
- 通常通过类名访问。

```
class Student {
    static int studentCount;
}

public class Main {
```

```
    public static void main(String[] args) {
        Student.studentCount = 10; // 通过类名访问
        System.out.println(Student.studentCount);
    }
}
```

3) **局部变量**

局部变量声明在方法、构造方法或代码块内部。

- 只能在所属方法或代码块中访问；
- 不会被自动初始化；
- 必须在读取之前显式赋值；
- 离开所属作用域后，变量名不再可用。

```
void printAge() {
    int age = 18;
    System.out.println(age);
}
```

以下代码无法通过编译：

```
void printAge() {
    int age;
    // System.out.println(age); // 错误：局部变量尚未初始化
}
```

4) **参数**

参数是在方法或构造方法声明中定义的变量，其初始值由调用者传入。

```
void printAge(int age) {
    System.out.println(age);
}
```

成员变量包括**实例变量和静态变量，它们会获得默认值**：

- 整数类型：0；
- 浮点类型：0.0；
- 字符类型：'\u0000'；
- 布尔类型：false；
- 引用类型：null。

局部变量没有默认值。

1.2.3 Java 数据类型

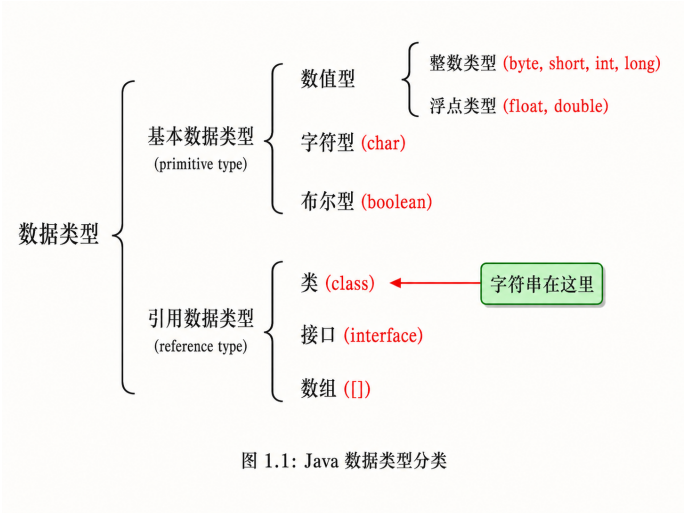


图 1.1: Java 数据类型分类

图 1.1: Java data type classification

- **八种基本数据类型**

1) **整数类型**

Java 提供四种有符号整数类型：

- `byte`：8 位，取值范围为 -2^7 到 2^7-1，即 -128 到 127。
- `short`：16 位，取值范围为 $-2^{15} \approx -32768$ 到 $2^{15}-1 \approx 32767$
- `int`：32 位，取值范围为 -2^{31} 到 $2^{31}-1$，是整数运算中最常用的类型。
- `long`：64 位，取值范围为 -2^{63} 到 $2^{63}-1$。

默认情况下，整数字面量的类型是 `int`。`long` 字面量通常需要添加后缀 `L`：

```
byte small = 100;
short count = 30000;
int age = 18;
long population = 1400000000L;
```

当整数运算结果超过类型的取值范围时，会发生溢出：

```
int value = Integer.MAX_VALUE;
value = value + 1; // 结果变为 Integer.MIN_VALUE
```

2) **浮点类型**

Java 提供两种浮点类型：

- `float`：32 位单精度浮点数，通常约有 6 至 7 位十进制有效数字。
- `double`：64 位双精度浮点数，通常约有 15 至 16 位十进制有效数字。

`float` 字面量必须添加后缀 `F` 或 `f`：

```
float rate = 3.14F;
double pi = 3.141592653589793;
```

浮点数采用有限的二进制位表示，因此许多十进制小数不能被精确保存。因此，**不应直接使用 == 判断两个计算得到的浮点数是否相等**。对金额等需要精确十进制计算的数据，通常使用 `BigDecimal`：

```
double result = 0.1 + 0.2;
System.out.println(result); // 可能输出 0.30000000000000004
```

3) **字符类型**

`char` 是 16 位无符号类型，用于保存一个 UTF-16 代码单元，取值范围为 `'\u0000'` 到 `'\uffff'`。

字符字面量使用单引号（**必须用单引号' ' 包裹，且只能包含一个字符**）：

```
char letter = 'A';
char chinese = '中';
char newline = '\n';
char unicode = '\u0041'; // 'A'
```

常见转义字符包括：

- `\n`：换行；
- `\t`：制表符；
- `\\`：反斜杠；
- `\'`：单引号；
- `\"`：双引号。

`char` 可以参与整数运算：

```
char letter = 'a';
int code = letter;      // 97
char next = (char) (letter + 1); // 'b'
```

需要注意，部分 Unicode 字符需要两个 UTF-16 代码单元表示，因此并非所有可见字符都能存入一个 `char`。

4) **布尔类型**

`boolean` 只有两个可能的值：

```
boolean isStudent = true;
boolean isAdult = false;
```

- 只能使用 `true` 或 `false`；
- **不能使用 `0` 和 `1` 代替；**
- 主要用于条件判断和逻辑运算；
- Java 语言规范没有规定 `boolean` 必须占用一个字节。

逻辑运算示例：

```
boolean result1 = isStudent && isAdult;
boolean result2 = isStudent || isAdult;
boolean result3 = !isStudent;
```

- **三类引用类型**

引用类型变量通常不直接保存对象中的全部数据，而是保存对某个对象的引用；如果当前没有引用任何对象，则**其值可以是** `null`。

1) **类类型（Class Type）**

类类型包括开发者自定义的类，以及 Java 标准库提供的类。

常见的类类型包括：

- `String`；
- `Integer`；
- `ArrayList`；
- 开发者自定义的类，如 `Student`。

例如：

```
String name = "张三";
Student student = new Student();
```

- `String` 和 `Student` 是类类型；
- `name` 和 `student` 是引用变量；

`String` 虽然具有特殊的字符串字面量语法，但它仍然是一个类，不是基本数据类型：

```
String message = "Hello";
```

2) **接口类型（Interface Type）**

接口规定一组方法要求，但通常不直接提供完整的对象实现（可以把接口理解成一份能力规范：接口只规定“对象必须能做什么”，具体“怎么做”由实现接口的类决定）

接口类型的变量可以引用任何实现了该接口的对象（同一个接口变量可以指向不同实现类对象，实际执行哪个版本，取决于当前引用的对象具体是哪个）：

```
interface Animal {
    void makeSound();
```

```
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("汪");
    }
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println("喵");
    }
}

Animal animal1 = new Dog();
Animal animal2 = new Cat();
```

Java 集合框架中也经常使用这种写法：

```
List<String> names = new ArrayList<>();
Map<String, Integer> scores = new HashMap<>();
```

List 和 Map 是接口，ArrayList 和 HashMap 是具体实现类。

3) **数组类型（Array Type）**

数组用于保存多个相同类型的元素。

数组本身是对象，因此数组类型属于引用数据类型。

```
int[] scores = {80, 90, 75};
String[] names = {"Alice", "Bob"};

```

其中：

- int[] 是整数数组类型；
- String[] 是字符串数组类型；
- scores 和 names 都是引用变量；
- 数组创建后长度固定；
- 使用下标访问元素，下标从 0 开始。

也可以使用 new 创建数组：

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

```
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
```

新建数组中的元素会获得默认值：

- 整数元素的默认值为 0；
- 浮点元素的默认值为 0.0；
- 布尔元素的默认值为 false；
- 引用类型元素的默认值为 null。

引用类型的共同特征

- 引用变量可以引用一个对象，也可以保存 null。
- 使用 new 通常会创建新的对象或数组：

```
Student student = new Student();
int[] numbers = new int[5];
```

- 如果引用变量为 null，就不能通过它访问对象成员：

```
Student student = null;
// student.name = "Alice"; // 抛出 NullPointerException
```

- 将一个引用变量赋给另一个引用变量，通常只复制引用，不会复制对象本身：

```
Student student1 = new Student();
Student student2 = student1;
```

此时，student1 和 student2 **引用同一个** Student **对象**。

- 对于引用类型，== 通常判断两个引用是否指向同一个对象；比较对象内容通常使用 equals()：

```
String first = new String("Java");
String second = new String("Java");
System.out.println(first == second);      // false
System.out.println(first.equals(second)); // true
```

- **基本类型与引用类型的区别**
  - 基本类型变量直接表示基本值：

```
int age = 18;
```

```
double score = 95.5;
```

- 引用类型变量表示对对象或数组的引用：

```
String name = "Alice";
int[] scores = {80, 90, 75};
```

- **基本类型不能保存 null，引用类型可以保存 null。**
- 基本类型共有八种：byte、short、int、long、float、double、char 和 boolean；但是，类、接口和数组都可以形成大量不同的引用类型，因此引用类型具体数量不固定。
- 当声明一个基本类型的变量时，该变量本身即保存了其值，并且这个值是直接存储在**栈（Stack）**内存中的；但当你创建一个引用类型的对象时，实际的对象数据被放置在**堆（Heap）**内存中，而栈内存中只保存了一个指向堆中该对象地址的引用（通常所说的指针）。

- **（基本）数据类型转换**

  数值类型之间的转换可以分为自动类型转换和强制类型转换。

  1) **自动类型转换**

  当较小范围的数值类型转换为较大范围的数值类型时，Java 通常可以自动完成转换。

  常见转换顺序为：

$$\texttt{byte} \to \texttt{short} \to \texttt{int} \to \texttt{long} \to \texttt{float} \to \texttt{double}.$$

  char 可以自动转换为：

$$\texttt{char} \to \texttt{int} \to \texttt{long} \to \texttt{float} \to \texttt{double}.$$

  例如：

```
byte small = 10;
int number = small;

char letter = 'A';
int code = letter;

int count = 100;
double result = count;
```

  需要注意：

  - byte 和 short 不能自动转换为 char；
  - char 不能自动转换为 byte 或 short；
  - byte、short 和 char 在进行整数算术运算时，Java 会先执行二元数值提升到 int；

- **整数自动转换为 `float` 或 `double` 时，虽然不会产生编译错误，但仍可能丢失精度**；
- `boolean` 不能与其他基本类型相互转换。

例如：

```
byte a = 10;
byte b = 20;

// byte sum = a + b; // 错误：a + b 的结果类型是 int
int sum = a + b;
```

2) **强制类型转换**

当较大范围的类型转换为较小范围的类型时，必须显式指定目标类型。

语法为：

```
目标类型 变量名 =（目标类型）原值;
```

例如：

```
double value = 3.9;
int integer = (int) value; // 3
```

强制转换可能产生以下问题：

- 浮点数转换为整数时，小数部分会被直接截断，而不是四舍五入
- 较大的整数转换为较小的整数类型时，**可能发生溢出**
- 转换后**可能丢失数值精度**

例如：

```
int number = 128;
byte small = (byte) number; // -128，发生溢出
```

字符和整数也可以显式转换：

```
char letter = 'A';
int code = letter;       // char 自动转换为 int

int value = 66;
char character = (char) value; // 'B'
```

3) **基本类型与 `String` 的转换**

将**基本类型转换为字符串**，常用 `String.valueOf()`：

```
int number = 100;
```

```
2
3 String text1 = String.valueOf(number);
4 String text2 = number + "";
```

在普通字符串拼接中，可以使用 +；只进行类型转换时，String.valueOf() 含义更加明确。

将**字符串转换为基本类型**，需要使用相应包装类的解析方法：

```
1 int number = Integer.parseInt("123"); // 返回int数值
2 long count = Long.parseLong("1000");
3 double price = Double.parseDouble("3.14");
4 boolean flag = Boolean.parseBoolean("true");
```

常见转换方法包括：

- Byte.parseByte();
- Short.parseShort();
- Integer.parseInt();
- Long.parseLong();
- Float.parseFloat();
- Double.parseDouble();
- Boolean.parseBoolean()。

如果字符串不是合法的数值格式，数值解析方法会抛出 NumberFormatException：

```
1 // int number = Integer.parseInt("abc");
2 // 运行时抛出 NumberFormatException
```

valueOf() 通常返回对应的包装类对象：

```
1 Integer number = Integer.valueOf("123"); // 返回 Integer 对象
2 Double price = Double.valueOf("3.14"); // 返回 Double 对象
```

当需要基本类型时，通常直接使用 parseInt()、parseDouble() 等方法。

1.2.4 进制与进制转换

1. 常见进制

进制决定一个数使用哪些数字，以及每一位的权重。

- **二进制：** 逢 2 进 1，只使用 0 和 1。
- **八进制：** 逢 8 进 1，使用 0 到 7。
- **十进制：** 逢 10 进 1，使用 0 到 9。

- **十六进制：**逢 16 进 1，使用 0 到 9 和 A 到 F，其中 A = 10, . . . , F = 15。

Java 整数字面量的写法如下：

```
int binary = 0b10110; // 二进制
int octal = 026;      // 八进制
int decimal = 22;     // 十进制
int hexadecimal = 0x16; // 十六进制
```

上述四个变量保存的值均为十进制的 22。

- 二进制字面量以 `0b` 或 `0B` 开头；
- 八进制字面量以 `0` 开头；
- 十进制字面量没有前缀；
- 十六进制字面量以 `0x` 或 `0X` 开头。

2. 不同进制之间的转换

1) **其他进制转换为十进制**

对于 r 进制数

$$(d_n d_{n-1} \cdots d_1 d_0)_r,$$

其十进制值为

$$\sum_{i=0}^{n} d_i r^i.$$

例如：

$$\begin{aligned}(10110)_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 16 + 4 + 2 = 22\end{aligned}$$

2) **十进制整数转换为其他进制**

使用“除以目标进制、记录余数”的方法：

- 不断将十进制整数除以目标进制；
- 记录每次得到的余数；
- 从最后一个余数开始倒序排列。

例如，将十进制 22 转换为二进制：

$$\begin{aligned}22 \div 2 &= 11 \text{ 余} 0, \\ 11 \div 2 &= 5 \text{ 余} 1, \\ 5 \div 2 &= 2 \text{ 余} 1, \\ 2 \div 2 &= 1 \text{ 余} 0, \\ 1 \div 2 &= 0 \text{ 余} 1.\end{aligned}$$

将余数逆序排列：

$$(22)_{10} = (10110)_2.$$

3) **二进制与八进制转换**

从小数点位置开始，**每三位**二进制数对应一位八进制数：

$$(110\,101)_2 = (65)_8.$$

4) **二进制与十六进制转换**

从小数点位置开始，**每四位**二进制数对应一位十六进制数：

$$(1011\,0101)_2 = (\mathrm{B5})_{16}.$$

不足三位或四位时，在左侧补零：

$$(101)_2 = (0101)_2 = (5)_{16}.$$

3. Java 中的进制转换

将十进制整数转换为其他进制字符串：

```java
int number = 22;

String binary = Integer.toBinaryString(number); // "10110"
String octal = Integer.toOctalString(number);   // "26"
String hex = Integer.toHexString(number);       // "16"
```

也可以使用通用方法：

```java
String binary = Integer.toString(22, 2);
String octal = Integer.toString(22, 8);
String hex = Integer.toString(22, 16);
```

将其他进制字符串转换为十进制整数：

```java
int binary = Integer.parseInt("10110", 2); // 22
int octal = Integer.parseInt("26", 8);     // 22
int hex = Integer.parseInt("16", 16);      // 22
```

其中，`parseInt` 的第二个参数表示字符串所采用的进制。

- 支持的进制范围通常为 2 到 36；
- 转换结果是一个普通的十进制整数值；
- 如果字符串包含该进制不允许的字符，会抛出 `NumberFormatException`。

```java
// Integer.parseInt("102", 2);
```

```
// 错误：二进制字符串中不能出现数字 2
```

4. 有符号整数与补码

Java 的整数类型使用固定长度的二进制补码表示有符号整数。

- 最高位为符号位；
- 最高位为 0 时表示非负数；
- 最高位为 1 时表示负数；
- 正数的补码与普通二进制表示相同；
- 负数的补码可以通过“按位取反后加 1”得到。

以 8 位二进制为例：

$$5 = (0000\,0101)_2.$$

对 5 按位取反：

$$1111\,1010.$$

再加 1：

$$-5 = (1111\,1011)_2.$$

也可以使用以下关系理解 n 位补码：

$$-x \text{ 的补码} = 2^n - x.$$

例如，8 位补码中：

$$2^8 - 5 = 256 - 5 = 251 = (1111\,1011)_2.$$

补码使加法和减法可以使用相同的二进制加法电路。例如：

$$\begin{aligned} 5 + (-5) &= 0000\,0101 + 1111\,1011 \\ &= 1\,0000\,0000. \end{aligned}$$

舍弃超出 8 位的最高进位后，结果为：

$$0000\,0000.$$

需要注意，`Integer.toBinaryString()` 对负数返回 32 位 `int` 的补码表示：

```
System.out.println(Integer.toBinaryString(-5));
// 输出 11111111111111111111111111111011 而不是 -101
```

5. 与二进制相关的位运算：位运算直接对整数的二进制位进行操作

- `&`：按位与；
- `|`：按位或；

- `^`：按位异或；
- `~`：按位取反；
- `<<`：左移；
- `>>`：带符号右移；
- `>>>`：无符号右移。

例如：

```
int a = 5; // 0101
int b = 3; // 0011

int and = a & b; // 0001，即 1
int or = a | b;  // 0111，即 7
int xor = a ^ b; // 0110，即 6

int left = a << 2;  // 20
int right = 20 >> 2; // 5
```

对不发生溢出的非负整数，左移和右移可以近似理解为：

$$x \ll n \approx x \cdot 2^n, \qquad x \gg n \approx \frac{x}{2^n}.$$

例：位移运算

$$x = 5 = (0000\,0101)_2.$$

将 x 左移两位：

$$0000\,0101 \xrightarrow{\text{左移两位}} 0001\,0100,$$
$$5 \ll 2 = 20 = 5 \cdot 2^2.$$

将 20 右移两位：

$$0001\,0100 \xrightarrow{\text{右移两位}} 0000\,0101,$$
$$20 \gg 2 = 5 = \left\lfloor \frac{20}{2^2} \right\rfloor.$$

当除法结果不是整数时，右侧移出的二进制位会被丢弃。例如：

$$7 = (0000\,0111)_2, \qquad 7 \gg 1 = (0000\,0011)_2 = 3.$$

对应的 Java 代码为：

```
int a = 5 << 2;   // 20
int b = 20 >> 2;  // 5
int c = 7 >> 1;   // 3
```

因此，对于非负整数，在左移不发生溢出的情况下：

$$x \ll n = x \cdot 2^n, \qquad x \gg n = \left\lfloor \frac{x}{2^n} \right\rfloor.$$

但对于负数、溢出或无符号右移，不能简单使用上述关系判断结果。

1.2.5 运算符

运算符（operator）用于对一个或多个操作数进行计算、比较、赋值或逻辑判断。

1. 算术运算符

Java 常见的算术运算符包括：

- `+`：加法或正号；
- `-`：减法或负号；
- `*`：乘法；
- `/`：除法；
- `%`：取余；
- `++`：自增；
- `--`：自减。

基本示例：

```
int a = 10;
int b = 3;

int sum = a + b;        // 13
int difference = a - b; // 7
int product = a * b;    // 30
int quotient = a / b;   // 3
int remainder = a % b;  // 1
```

1) **整数除法**

两个整数相除时，结果仍然是整数，小数部分会被直接截断：

```
int result = 10 / 3; // 3
```

如果需要浮点结果，至少有一个操作数必须是浮点类型：

```
double result = 10.0 / 3; // 3.333...
```

2) **除以零**

整数除以零会抛出 `ArithmeticException`：

```
// int result = 10 / 0; // ArithmeticException
```

浮点数除以零遵循 IEEE 754 规则：

```
double a = 10.0 / 0.0; // Infinity
double b = 0.0 / 0.0;  // NaN
```

3) **取余运算**

Java 中满足：

$$a = b\,(a/b) + (a \bmod b),$$

其中整数除法 a/b 向零截断。因此，余数的符号与 **被除数**相同：

```
System.out.println(10 % 3);   // 1
System.out.println(-10 % 3);  // -1
System.out.println(10 % -3);  // 1
System.out.println(-10 % -3); // -1
```

4) **整数运算中的类型提升**

byte、short 和 char 参与算术运算时，通常会先提升为 int：

```
byte x = 10;
byte y = 20;

int sum = x + y;
// byte result = x + y; // 错误：结果类型是 int
```

5) **自增与自减**

++ 使变量增加 1，-- 使变量减少 1。

前缀形式先修改变量，再返回新值；后缀形式先返回旧值，再修改变量：

```
int a = 5;

int b = a++; // b = 5, a = 6
int c = ++a; // a = 7, c = 7
```

可以记为：

- a++：先使用旧值，再自增；
- ++a：先自增，再使用新值。

单独作为一条语句时，两者效果相同：

```
a++;
++a;
```

2. 比较运算符

比较运算的结果一定是布尔值 `true` 或 `false`。

- `>`：大于；
- `<`：小于；
- `>=`：大于等于；
- `<=`：小于等于；
- `==`：等于；
- `!=`：不等于。

例如：

```
int age = 20;

boolean adult = age >= 18; // true
boolean exact = age == 20; // true
boolean other = age != 20; // false
```

Java 不支持连续比较：

```
// if (0 <= age <= 100) { } // 错误

if (0 <= age && age <= 100) {
    System.out.println("范围合法");
}
```

对基本类型使用 == 时，比较的是值：

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

System.out.println(a == b); // true

```

对引用类型使用 == 时，比较的是两个引用是否指向同一个对象，而不是对象内容是否相同：

```
String first = new String("Java");
String second = new String("Java");

System.out.println(first == second);        // false
System.out.println(first.equals(second)); // true
```

因此，比较字符串内容通常使用 `equals()`。

3. 逻辑运算符

逻辑运算符用于组合或取反布尔表达式。

- `!`：取反；
- `&`：左右都为 true 才返回 true；
- `|`：左右有一个为 true，则返回 true；
- `^`：左右不同才返回 true。
- `&&`：若左边为 false，右边不执行；
- `||`：若左边为 true，右边不执行；

例如：

```
int age = 20;
boolean hasId = true;

boolean allowed = age >= 18 && hasId;
boolean denied = !allowed;
```

`&&` 和 `||` 具有短路行为：

- 对于 `&&`，如果左侧为 `false`，右侧不会执行；
- 对于 `||`，如果左侧为 `true`，右侧不会执行。

例如：

```
String text = null;

if (text != null && text.length() > 0) {
    System.out.println(text);
}
```

当 `text == null` 时，右侧的 `text.length()` 不会执行，因此不会抛出 `NullPointerException`。对布尔操作数使用 `&` 和 `|` 时，两侧表达式都会执行，通常优先使用 `&&` 和 `||`。

4. 赋值运算符

基本赋值运算符为：

```
int value = 10;
```

常见复合赋值运算符包括：

- `+=`
- `-=`

- *=
- /=
- %=
- &=、|=、^=
- <<=、>>=、>>>=

例如：

```
int value = 10;

value += 5; // value = 15
value *= 2; // value = 30
```

对于一般变量：

$$x \mathrel{+}= y$$

可以理解为：

$$x = x + y.$$

但是**复合赋值会隐式转换回左侧变量的类型**；因此，**复合赋值也可能发生溢出或精度丢失**：

```
byte value = 10;

value += 5; // 合法
// value = value + 5; // 错误：右侧结果为 int
```

value += 5 大致相当于：

```
value = (byte) (value + 5);
```

5. 条件运算符

条件运算符也称为三元运算符，语法为：

```
条件 ? 条件为真时的表达式，用于简化 if-else 语句：条件为假时的表达式
```

例如：

```
int score = 85;
String result = score >= 60 ? "及格" : "不及格";
```

执行过程为：

- 如果条件为 true，返回问号后、冒号前的表达式；

- 如果条件为 `false`，返回冒号后的表达式；
- 两个结果表达式的类型必须能够形成兼容的结果类型。

三元运算符适合表达简单的二选一结果。对复杂逻辑应使用 `if-else`，以保持可读性。

6. 位运算符

位运算符直接操作整数的二进制位，适用于 `byte`、`short`、`char`、`int` 和 `long`。

- `&`：按位与；
- `|`：按位或；
- `^`：按位异或；
- `~`：按位取反；
- `<<`：左移；
- `>>`：带符号右移；
- `>>>`：无符号右移。

例如：

```
int a = 5; // 0101
int b = 3; // 0011

int and = a & b; // 0001，即 1
int or = a | b;  // 0111，即 7
int xor = a ^ b; // 0110，即 6
```

各个位运算的规则为：

- 按位与：两位都为 1 时结果为 1；
- 按位或：至少一位为 1 时结果为 1；
- 按位异或：两位不同时结果为 1；
- 按位取反：将每一位的 0 和 1 互换。

位移示例：

```
int left = 5 << 2;    // 20
int right = 20 >> 2;  // 5
int unsigned = -8 >>> 1;
```

- `<<`：所有二进制位向左移动，右侧补 0；
- `>>`：所有二进制位向右移动，左侧补符号位；
- `>>>`：所有二进制位向右移动，左侧统一补 0。

对于非负整数，在左移不发生溢出的情况下：

$$x \ll n = x \cdot 2^n, \qquad x \gg n = \left\lfloor \frac{x}{2^n} \right\rfloor.$$

byte、short 和 char 参与位运算时，通常会先提升为 int。

7. 字符串拼接

运算符 + 除了表示数值加法，还可以用于字符串拼接：

```
String first = "Hello";
String second = "World";
String message = first + " " + second; // "Hello World"
```

当 + 的任意一个操作数是字符串时，另一个操作数会被转换为字符串：

```
String message = "Age: " + 18; // "Age: 18"
```

+ 按照从左到右的顺序执行：

```
System.out.println(1 + 2 + "3"); // "33"
System.out.println("1" + 2 + 3); // "123"
```

第一条语句先计算 1 + 2 = 3，再与字符串 "3" 拼接。
第二条语句从第一个操作开始就是字符串拼接。

8. 运算符优先级与结合方向

常见 Java 运算符的优先级从高到低为：() → 算术运算符（++, –, +, -, *, /, %）→ 关系运算符（>, <, >=, <=, ==, !=）→ 逻辑运算符（|,!, &,^, &&, ||）→ 三元运算符（? :）→ 赋值运算符（=, +=, -=, *=, /=, %=）

1) 括号：()；
2) 后缀：expr++、expr--；
3) 一元运算：++、--、+、-、!、~；
4) 乘除取余：*、/、%；
5) 加减：+、-；
6) 位移：<<、>>、>>>；
7) 大小比较：<、>、<=、>=、instanceof；
8) 相等比较：==、!=；
9) 按位与：&；
10) 按位异或：^；
11) 按位或：|；
12) 逻辑与：&&；

13) 逻辑或：||；

14) 条件运算：?:；

15) 赋值运算：=、+=、-= 等。

```
int result = 2 + 3 * 4;    // 14
int other = (2 + 3) * 4;   // 20
```

大多数二元运算符从左向右结合:

```
int result = 20 / 5 * 2; // (20 / 5) * 2 = 8
```

赋值运算符从右向左结合:

```
int a;
int b;

a = b = 10;
```

当表达式较复杂时，应使用括号明确执行顺序，不应过度依赖记忆运算符优先级。

1.2.6 流程控制

流程控制决定程序中各条语句的执行顺序。Java 的流程控制结构主要包括：

- 顺序结构；
- 分支结构；
- 循环结构；
- 跳转语句。

1. 顺序结构

在没有分支、循环或跳转语句的情况下，程序按照代码书写顺序，从上到下依次执行。

```
System.out.println("第一步");
System.out.println("第二步");
System.out.println("第三步");
```

2. 分支结构

分支结构根据条件选择不同的代码路径。Java 中常见的分支语句包括 `if` 和 `switch`。

1) **单分支：`if`**

当条件为 `true` 时执行代码块；条件为 `false` 时跳过代码块。

```
if (条件) {
    // 条件为 true 时执行
```

```
}
```

其中，条件必须是结果为 boolean 的表达式：

```
int score = 85;
if (score >= 60) {
    System.out.println("考试通过");
}
```

Java 不能使用整数代替布尔值：

```
int value = 1;
// if (value) { } // 错误：条件必须是 boolean
```

如果代码块中只有一条语句，可以省略**花括号 ({})**，但通常不推荐：

```
if (score >= 60)
    System.out.println("考试通过");
```

推荐始终保留花括号，以避免后续修改代码时产生错误。

2) **双分支：** `if-else`

当条件为 `true` 时执行 `if` 代码块，否则执行 `else` 代码块。

```
if (条件) {
    // 条件为 true 时执行
} else {
    // 条件为 false 时执行
}
```

例如：

```
int age = 17;

if (age >= 18) {
    System.out.println("已经成年");
} else {
    System.out.println("尚未成年");
}
```

两个代码块中只会执行一个。

3) **多重分支：** `if-else if-else`

多重分支用于依次判断多个条件：

```
if (条件1) {
```

```
    // 条件1成立
} else if (条件2) {
    // 条件1不成立，但条件2成立
} else {
    // 所有条件都不成立
}
```

执行规则如下：

- 条件按照**从上到下**的顺序判断；
- 找到第一个结果为 `true` 的条件后，执行对应代码块；
- 执行完该代码块后，**跳过剩余分支**；
- 最后的 `else` 可省略，用于处理其他所有情况。

例如：

```
int score = 88;

if (score >= 90) {
    System.out.println("A");
} else if (score >= 80) {
    System.out.println("B");
} else if (score >= 70) {
    System.out.println("C");
} else {
    System.out.println("D");
} // only return "B"
```

条件顺序非常重要。范围更严格的条件通常应放在前面：

```
// 错误逻辑：score >= 90 时，第一个条件已经成立
if (score >= 60) {
    System.out.println("及格");
} else if (score >= 90) {
    System.out.println("优秀");
}
```

4) 嵌套分支

一个 `if` 或 `else` 代码块内部可以继续包含其他分支语句。

```
int age = 18;
boolean isStudent = true;
```

```

if (age >= 18) {
    if (isStudent) {
        System.out.println("成年学生");
    } else {
        System.out.println("成年非学生");
    }
} else {
    System.out.println("未成年");
}
```

内层条件只有在程序进入对应的外层分支后才会判断。**当省略花括号时，`else` 会与最近一个尚未匹配的 `if` 配对**。因此，嵌套分支中应使用花括号明确结构。

5) **多路分支：`switch`**

传统的 `switch` 语句根据表达式的值，选择与之相等的 `case` 分支。

```
switch (表达式) {
    case 常量1:
        // 表达式等于常量1时执行
        break;
    case 常量2:
        // 表达式等于常量2时执行
        break;
    default:
        // 没有任何 case 匹配时执行
}
```

例如：

```
int day = 3;

switch (day) {
    case 1:
        System.out.println("星期一");
        break;
    case 2:
        System.out.println("星期二");
        break;
    case 3:
        System.out.println("星期三");
```

```
        break;
    default:
        System.out.println("无效日期");
}
```

传统 switch 需要注意：

- case 后面必须是编译期常量或枚举常量；
- 同一个 switch 中不能出现重复的 case 值；
- default 是可选的；可以写在任意位置，但通常放在最后；
- switch 主要用于离散值的等值匹配，不适合直接表示范围条件。

传统写法中，如果省略 break，程序会继续执行后续 case，这种行为称为穿透（fall-through）；如果不需要穿透，应在对应分支末尾使用 break。

```
int value = 1;

switch (value) {
    case 1:
        System.out.println("A");
    case 2:
        System.out.println("B");
}
// "A"
// "B"
```

在传统等值匹配中，常见的表达式类型包括：

- byte、short、char 和 int；
- 对应的包装类型；
- String；
- 枚举类型。

long、float、double 和 boolean 不能用于传统的 switch 等值匹配；一般选择原则为：

- 范围判断或复杂布尔条件：使用 if-else；
- 一个表达式与多个离散值进行等值匹配：使用 switch。

3. 循环结构

循环结构用于重复执行一段代码。Java 中常见的循环包括：

- while；
- do-while；

- 基本 `for`；
- 增强型 `for`。

循环通常涉及：

- 初始化循环变量；
- 判断循环条件；
- 执行循环体；
- 更新循环变量。

必须保证循环条件最终有机会变为 `false`，否则可能形成无限循环。

1) `while` **循环**

`while` 先判断条件，再决定是否执行循环体：

```
初始化;
while (条件) {
    // 循环体
    更新;
}
```

执行顺序为：

i. 判断条件；
ii. 条件为 `true` 时执行循环体；
iii. 更新相关变量；
iv. 返回条件判断；
v. 条件为 `false` 时结束循环。

例如：

```
int i = 1; // 初始化;
while (i <= 5) {
    System.out.println(i);
    i++; // 更新;
} // "1\n2\n3\n4\n5\n"
```

如果初始条件就是 `false`，循环体一次也不会执行。

2) `do-while` **循环**

`do-while` 先执行一次循环体，再判断条件：

```
初始化;
do {
    // 循环体
```

```
    更新;
} while (条件);
```

需要注意，结尾处 `while` 必须有分号：

```
int i = 1; // 初始化;
do {
    System.out.println(i);
    i++; // 更新;
} while (i <= 5); // "1\n2\n3\n4\n5\n"
```

即使初始条件为 `false`，循环体也至少执行一次。

3) **基本 `for` 循环**

当循环次数或循环变量变化规律较明确时，通常使用基本 `for` 循环。

```
for (初始化; 条件; 更新) {
    // 循环体
}
```

执行顺序为：

i. 执行初始化，只执行一次；
ii. 判断条件；
iii. 条件为 `true` 时执行循环体；
iv. 执行更新语句；
v. 返回条件判断。

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
} // "1\n2\n3\n4\n5\n"
```

在初始化部分声明的变量，其作用域通常只在 `for` 循环内部：

```
for (int i = 0; i < 3; i++) {
    System.out.println(i);
}
// System.out.println(i); // 错误：i 已超出作用域
```

三个部分都可以省略，但两个分号不能省略：

```
for (;;) {
    // 无限循环
}
```

4) **增强型 for 循环**

增强型 for 用于依次访问数组或可迭代对象中的元素：

```
for (元素类型 变量名 : 数组或集合) {
    // 使用当前元素
}
```

例如：

```
int[] numbers = {1, 2, 3};
for (int number : numbers) {
    System.out.println(number);
} // "1\n2\n3\n"
```

其特点为：

- 不需要手动管理下标；
- 适合依次读取所有元素；
- 不能直接获得当前元素的下标；
- 不适合需要逆序、跳跃或按下标修改元素的场景。

对基本类型数组，修改循环变量不会修改数组元素：

```
int[] numbers = {1, 2, 3};
for (int number : numbers) {
    number = 0;
}
```

上述代码只修改局部变量 number，数组中的元素不会变为 0。

5) **嵌套循环**

一个循环体中可以包含另一个完整循环：

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        System.out.println(i + ", " + j);
    }
} // "1, 1\n1, 2\n2, 1\n2, 2\n3, 1\n3, 2\n"
```

执行规则为：**外层循环每执行一次，内层循环都会从头完整执行一遍**。

如果外层循环执行 m 次，内层循环每次执行 n 次，则循环体通常执行 $m \times n$ 次。

4. 跳转语句

跳转语句可以改变循环、分支或方法的正常执行顺序。常见跳转语句包括：

- `break`：break 立即结束所在的最近一层循环/switch
- `continue`：结束最近一层当前一次循环，继续最近一层下一次循环
- `return`：结束整个当前方法，并可返回结果

1) `break`：`break` 用于立即结束

   - 当前所在的最近一层循环；
   - 当前所在的 `switch` 语句。

   例如：

```java
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
} // "1\n2\n3\n4\n"
```

   在嵌套循环中，**普通 `break` 只结束它所在的最近一层循环**：

```java
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (j == 1) {
            break;
        }
    }
} // ""
```

   Java 还支持带标签的 `break`，用于**结束指定的外层语句**：

```java
outer:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (i == 1 && j == 1) {
            break outer;
        }
    }
}
```

   带标签跳转应谨慎使用，以免降低代码可读性。

2) `continue`

   `continue` 用于结束当前这一次循环，跳过循环体中剩余语句，并开始下一次循环。

```java
for (int i = 1; i <= 5; i++) {
```

```
    if (i \% 2 == 0) {
        continue;
    }
    System.out.println(i);
} // "1\n3\n5\n"
```

在不同循环中的后续行为为：

- 在 for 中，转到更新部分，然后重新判断条件；
- 在 while 中，直接重新判断条件；
- 在 do-while 中，转到循环末尾的条件判断。

在嵌套循环中，**普通 continue 只影响最近一层循环**。

带标签的 continue 可以开始指定外层循环的下一次迭代：

```
outer:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (i == j) {
            continue outer;
        }

        System.out.println(i + ", " + j);
    }
}
```

3) return

return 用于**立即结束当前方法**。

在返回类型为 void 的方法中，可以直接写：

```
public static void check(int value) {
    if (value < 0) {
        return;
    }

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

在有返回值的方法中，return 后必须提供与方法返回类型兼容的值：

```
public static int max(int a, int b) {
    if (a >= b) {
```

```
3           return a;
4       }
5
6       return b;
7   }
```

执行 return 后，方法中位于其后的语句不会继续执行。

1.3 Java 数组

错题集

1.3.1 数组概念

数组（array）是一种用于保存多个同类型元素的数据结构。Java 数组本身属于**引用数据类型**。

1. 数组的基本特征

Java 数组具有以下特征：

- 数组对象自身属于引用类型
- 一个数组中的所有元素必须具有**相同的数据类型**
- 数组创建后，长度固定，不能再更改长度（扩容或者缩减长度需要创建新的数组对象）
- 数组中的元素按照固定顺序排列
- 数组使用下标访问元素，下标从 0 开始
- 数组**可以保存基本类型值，也可以保存对象引用**

注释

“静态数组”和“动态数组”：

- Java 原生数组（静态数组）的长度始终是固定的；
- `ArrayList` 等集合才可以在运行期间自动调整容量。

例如：

```
int[] ages = {18, 20, 22, 25};

System.out.println(ages[0]); // 18
System.out.println(ages[3]); // 25
```

对于长度为 n 的数组，其合法下标范围为：

$$0, 1, \ldots, n-1.$$

如果访问非法下标，会抛出 `ArrayIndexOutOfBoundsException`：

```
int[] numbers = new int[3];
// System.out.println(numbers[3]);
// 错误：合法下标只有 0、1、2
```

2. 数组的声明、创建与初始化

- **声明方式**

```
元素类型[] 数组名;
```

例如：

```
int[] numbers;
```

```
String[] names;
```

仅声明数组变量（stack）时，还没有创建数组对象（heap）：

```
int[] numbers;
```

- **使用 new 创建数组对象**：

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

也可以合并为一条语句：

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

- **数组初始化方式**

1) **直接指定元素**

```
int[] ages = {18, 20, 22, 25};
```

数组长度由元素个数自动确定，此处长度为 4。

2) **先指定长度，再逐个赋值**

```
double[] scores = new double[3];

scores[0] = 95.5;
scores[1] = 88.0;
```

未显式赋值的元素**保留默认值**，因此：

```
System.out.println(scores[2]); // 0.0
```

- **元素默认值**：
  - 整数类型：0；
  - 浮点类型：0.0；
  - char：'\u0000'；
  - boolean：false；
  - 引用类型：null。

例如：

```
int[] numbers = new int[3];
String[] names = new String[2];

System.out.println(numbers[0]); // 0
System.out.println(names[0]);   // null
```

- **数组的 length 属性获得数组长度**：

```java
int[] numbers = {10, 20, 30};
System.out.println(numbers.length); // 3
```

注意：

- 数组使用 length 属性，没有括号；
- 字符串长度使用 length() 方法。

```
numbers.length
text.length()
```

3. 数组变量与数组对象

数组变量保存的是数组对象的引用，而不是数组元素本身。

```java
int[] numbers = new int[3];
numbers[0] = 100;
```

可以将其理解为：

- new int[3] 创建一个数组对象；
- numbers 保存对该数组对象的引用；
- numbers[index] 是数组中的一个元素，类型是 int，保存的是数值，不是引用

将一个数组变量赋给另一个数组变量时，复制的是引用：

```java
int[] numbers = {100, 0, 0};
int[] copy = numbers;
copy[1] = 200;
System.out.println(numbers[1]); // 200
```

此时，numbers 和 copy 引用的是同一个数组对象，因此通过任意一个变量修改数组，都会影响同一个数组。

4. 数组的毁灭：当一个数组对象没有任何引用指向它时，就会被 GC 回收

5. 数组的复制

如果需要创建内容独立的新数组，不能只复制引用：

```java
int[] copy = numbers;
```

可以使用 Arrays.copyOf()：

```java
int[] numbers = {10, 20, 30};
int[] copy = Arrays.copyOf(numbers, numbers.length);

copy[0] = 100;

System.out.println(numbers[0]); // 10
System.out.println(copy[0]);    // 100
```

使用该方法需要导入：

```java
import java.util.Arrays;
```

也可以使用：

- System.arraycopy();
- 数组的 clone() 方法；
- 循环逐个复制元素。

对一维**基本类型数组**，上述方法会复制各个基本类型值；对**对象数组**，复制的通常仍然是各个对象的引用，而不是复制每个对象本身，这种复制称为**浅复制**。

6. 数组作为方法参数

Java 的参数传递始终是**值传递**。当数组作为参数传入方法时，传递的是数组引用的副本。该副本仍然指向原来的数组对象，因此方法内部可以修改数组元素：

```java
public static void change(int[] numbers) {
    numbers[0] = 100;
}

int[] values = {1, 2, 3};
change(values);
System.out.println(values[0]); // 100
```

需要区分：

- 如果通过参数修改数组中的某个元素，修改的是双方共同指向的那个数组，所以方法外也能看到变化
- 如果在方法中让参数重新指向另一个数组，只会改变参数变量自己的指向，调用者的变量仍然指向原来的数组

例如：

```java
public static void replace(int[] numbers) {
    numbers = new int[]{9, 9, 9};
```

```
}

int[] values = {1, 2, 3};
replace(values);
System.out.println(values[0]); // 1
```

方法中的 numbers 被改为指向新数组，但调用者的 values 仍然指向原数组。

7. 数组适用场景

- **数据数量固定、需要频繁按下标访问**时，可以使用数组；
- 数据数量经常变化时，通常使用 ArrayList 等集合；
- 需要频繁按顺序插入、删除或查找时，应根据具体需求选择合适的数据结构。

1.3.2 数组基本算法

扩容

Java 数组的长度在创建后不能改变。所谓数组扩容，并不是修改原数组的长度，而是创建一个更大的新数组，将原数组中的元素复制过去，再让变量指向新数组。

1) **使用** `Arrays.copyOf()`

```
import java.util.Arrays;

int[] numbers = {10, 20, 30};
int newLength = numbers.length * 2;
numbers = Arrays.copyOf(numbers, newLength);

System.out.println(Arrays.toString(numbers)); // [10, 20, 30, 0, 0, 0]
```

`Arrays.copyOf(numbers, newLength)` 会：

- 创建一个长度为 `newLength` 的新数组；
- 复制原数组中能够容纳的元素，新增加的位置使用该类型的默认值填充；
- 返回新数组的引用。

对于长度可能为 0 的数组，可以写成：

```
int newLength = Math.max(1, numbers.length * 2);
numbers = Arrays.copyOf(numbers, newLength);
```

2) **使用** `System.arraycopy()`

```
int[] oldArray = {10, 20, 30};
int[] newArray = new int[6];
System.arraycopy(
    oldArray, 0,
    newArray, 0,
    oldArray.length
);
```

其参数依次表示：

```
System.arraycopy(
    源数组, 源数组起始位置,
    目标数组, 目标数组起始位置,
    复制元素数量
);
```

`System.arraycopy()` 适合复制数组中的指定区间，但目标数组必须提前创建。

3) **可变长度集合**

如果程序需要频繁增加或删除元素，通常使用 `ArrayList`，而不是手动扩容数组：

```
ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(10);
numbers.add(20);
numbers.remove(0);
```

`ArrayList` 内部**仍然使用数组**保存元素，只是在**容量不足时自动完成新数组创建和元素复制**。

扩容的复杂度：如果原数组中有 n 个元素，扩容时需要复制这些元素，因此：

$$\text{时间复杂度} = O(n), \qquad \text{额外空间复杂度} = O(n).$$

遍历

按照一定顺序访问数组中的每个元素。常用方式包括普通 `for`、增强型 `for` 和 Stream API。

1) **按下标遍历**

```
int[] scores = {85, 90, 78, 92};
for (int i = 0; i < scores.length; i++) {
    System.out.println(
        "index = " + i + ", value = " + scores[i]
    );
}
```

按下标遍历具有以下特点：

- 可以获得当前元素的下标；
- 可以读取数组元素；
- 可以直接修改数组元素；
- 可以正序、逆序或跳跃访问。

例如，将每个分数增加 5：

```
for (int i = 0; i < scores.length; i++) {
    scores[i] += 5;
}
```

这里直接修改的是数组中的位置 `scores[i]`，因此数组内容会改变。

2) **增强型 `for` 遍历**

```java
for (int score : scores) {
    System.out.println(score);
}
```

每次循环时，当前数组元素的值会复制给循环变量 score；只修改局部变量 score，不会修改数组中的元素。因此，对于基本类型数组：

```java
for (int score : scores) {
    score = 0;
}
```

可以近似理解为：

```java
for (int i = 0; i < scores.length; i++) {
    int score = scores[i];
    score = 0;
}
```

对对象数组，循环变量保存的是对象引用的副本：

- 可以通过该引用修改对象内部的属性；
- 重新给循环变量赋值，不会替换数组中的引用。

增强型 for 适合只需要**依次读取元素**、**不需要下标**的情况。

3) **Stream API***

```java
int sum = Arrays.stream(scores).sum();
int max = Arrays.stream(scores)
                .max()
                .orElse(0);

Arrays.stream(scores)
      .forEach(System.out::println);
```

Stream API 适合执行：

- 求和、最大值和最小值；
- 过滤；
- 映射转换；
- 聚合统计。

Stream 通常不适合需要数组下标，或需要原地修改数组元素的简单循环。

4) **数组与迭代器** *

Java 原生数组没有实现 `Iterable` 接口，也没有提供 `iterator()` 方法，因此不能直接写：

```
int[] numbers = {1, 2, 3};
// numbers.iterator(); // 编译错误
```

但是，可以先将数据转换为支持迭代器的对象，再使用 `Iterator`。

对于包装类型数组，可以先转换为 `List`：

```
Integer[] numbers = {1, 2, 3};

List<Integer> list = Arrays.asList(numbers);
Iterator<Integer> iterator = list.iterator();

while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

查找

用于确定目标元素是否存在，并通常返回其下标。如果没有找到，常用返回值为 `-1`。

1) **线性查找（Linear Search）**

线性查找从数组开头开始，依次比较每个元素。

i. 从下标 0 开始；

ii. 比较 `arr[i]` 和目标值；

iii. 相等时返回当前下标；

iv. 遍历结束后仍未找到，则返回 `-1`。

```
public static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

线性查找：

- 不要求数组有序；
- 最坏时间复杂度为 $O(n)$；
- 额外空间复杂度为 $O(1)$。

2) **二分查找（Binary Search）**使用**左闭右闭**区间 [left, right] 的实现，每次判断有三种情况：

(a) `arr[mid] == target`：找到目标，返回 `mid`；

(b) `arr[mid] < target`：目标只可能在右半部分，因此令 `left = mid + 1`；

(c) `arr[mid] > target`：目标只可能在左半部分，因此令 `right = mid - 1`。

二分查找要求数组已经按照升序排列，每次将查找区间缩小为原来的一半。

- 前提是**数组已经有序**；
- 时间复杂度为 $O(\log n)$；
- 迭代实现的额外空间复杂度为 $O(1)$。

```java
public static int binarySearch(int[] arr, int target) {
    int left = 0;
    int right = arr.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

中间下标写成 `int mid = left + (right - left) / 2;` 而不是 `int mid = (left + right) / 2;` 可以避免 `left + right` 发生整数溢出。

注释

左闭右闭 [left, right] 把 `right` 当作最后一个需要检查的位置；**左闭右开** [left, right) 把 `right` 当作查找范围结束后的第一个位置，因此不检查 `right`。例如，长度为 5 的数组既可以表示为 $[0, 4]$ 也可以表示为 $[0, 5)$。表示的元素相同，只是右边界的定义不同。

3) **使用** `Arrays.binarySearch()`

```java
int[] numbers = {2, 5, 8, 10, 15};
int index = Arrays.binarySearch(numbers, 10); // index = 3
```

使用前**必须保证数组已排序**；如果目标不存在，该方法返回：−(插入位置) − 1.
例如：

```
int result = Arrays.binarySearch(numbers, 9);
```

数字 9 应插入下标 3，因此返回：$-3-1=-4$。

4) **跳跃查找** * **（Jump Search）**

跳跃查找适用于**有序数组**，基本思想是：

i. 按固定步长向后跳跃；
ii. 找到目标可能所在的区间；
iii. 在该区间内执行线性查找。

当步长取约 $\sqrt{n}$ 时，时间复杂度为：$O(\sqrt{n})$。
对普通内存数组而言，二分查找通常更常用。

```
public static int jumpSearch(int[] arr, int target) {
    int n = arr.length;
    int jumpSize = (int) Math.sqrt(n);
    int step = jumpSize;
    int prev = 0;

    // 找到目标值可能存在的区间
    while (arr[Math.min(step, n) - 1] < target) {
        prev = step;
        step += jumpSize;

        if (prev >= n) {
            return -1;
        }
    }
    // 在该区间内进行线性查找
    while (arr[prev] < target) {
        prev++;
        if (prev == Math.min(step, n)) {
            return -1;
        }
    }
    // 检查是否找到目标值
    if (arr[prev] == target) {
        return prev;
```

```
    }

    return -1;
}
```

5) **插值查找** * **（Interpolation Search）**

二分查找固定检查中间位置，插值查找根据目标值的大小，估计目标更可能出现位置，要求：

- **数组已经有序**；
- **数值分布相对均匀**；
- 需要**处理区间两端数值相等**的情况，避免除以零。

分布均匀的理想情况下，平均时间复杂度可达到：$O(\log \log n)$，但最坏时间复杂度仍可能：$O(n)$。

```
public static int interpolationSearch(int[] arr, int target) {
    int low = 0;
    int high = arr.length - 1;

    while (low <= high && target >= arr[low] && target <= arr[high]) {
        // 搜索区间中只剩一个元素
        if (low == high) {
            return arr[low] == target ? low : -1;
        }

        // 根据目标值估计它可能出现的位置
        // target 从最小值 arr[low] 走到最大值 arr[high] 的多少比例
        int pos = low
                  + ((target - arr[low]) * (high - low))
                  / (arr[high] - arr[low]);
        if (arr[pos] == target) {
            return pos;
        }

        if (arr[pos] < target) {
            // 目标值在 pos 的右边
            low = pos + 1;
        } else {
            // 目标值在 pos 的左边
            high = pos - 1;
        }
```

```
    }

    return -1;
}
```

6) 二维有序数组查找

如果二维数组满足：

- 每一行从左到右递增；
- 每一列从上到下递增；

可以从右上角开始查找：

```
public static boolean searchMatrix(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }

    // 从右上角开始查找
    int row = 0;
    int col = matrix[0].length - 1;

    while (row < matrix.length && col >= 0) {
        if (matrix[row][col] == target) {
            return true;
        } else if (matrix[row][col] > target) {
            col--; // 当前值大于目标，向左移动
        } else {
            row++; // 当前值小于目标，向下移动
        }
    }

    return false;
}
```

原理为：

- 当前值大于目标时，目标不可能在当前列下方，因此向左移动；
- 当前值小于目标时，目标不可能在当前行左侧，因此向下移动。

对于 m 行、n 列的矩阵，时间复杂度为：$O(m+n)$。

排序

排序是按照指定顺序重新排列数组元素（下面均以升序排序为例）。

1) **冒泡排序（Bubble Sort）**

冒泡排序反复比较数组中的相邻元素，如果前一个元素大于后一个元素，就交换两者。在一次从左向右的扫描中，较大的元素会不断向右移动。扫描结束后，当前未排序部分中的最大元素会到达最右端。

i) **基本冒泡排序**

假设当前未排序区间为：$[0, \texttt{end}]$，从左向右依次比较 `arr[i]` 与 `arr[i+1]`；
如果 `arr[i] > arr[i+1]` 就交换两个元素。

```
1    public static void bubbleSortBasic(int[] arr) {
2        for (int end = arr.length - 1; end > 0; end--) {
3            for (int i = 0; i < end; i++) {
4                if (arr[i] > arr[i + 1]) {
5                    swap(arr, i, i + 1);
6                }
7            }
8        }
9    }
```

其中：

- `end` 是当前未排序区间的最后一个下标；
- 内层循环比较 `arr[0]` 到 `arr[end]`；
- 每轮结束后，当前最大元素位于 `arr[end]`；
- 因此下一轮不再检查该位置，并令 `end` 减少 1。

例如，对数组：

$$[5, 3, 8, 4, 2]$$

进行排序：

$$\begin{aligned}
&\text{第 1 轮：} \quad [3, 5, 4, 2, 8],\\
&\text{第 2 轮：} \quad [3, 4, 2, 5, 8],\\
&\text{第 3 轮：} \quad [3, 2, 4, 5, 8],\\
&\text{第 4 轮：} \quad [2, 3, 4, 5, 8].
\end{aligned}$$

每一轮都会确定一个最大元素的最终位置。

ii) **提前结束优化**

基本冒泡排序存在一个问题：即使数组已经有序，仍然会继续执行剩余轮次。

可以使用布尔变量 swapped 记录当前一轮是否发生过交换。

```
    public static void bubbleSort(int[] arr) {
        for (int end = arr.length - 1; end > 0; end--) {
            boolean swapped = false;

            for (int i = 0; i < end; i++) {
                if (arr[i] > arr[i + 1]) {
                    swap(arr, i, i + 1);
                    swapped = true;
                }
            }

            if (!swapped) {
                break;
            }
        }
    }
```

执行规则为：

- 每轮开始时令 swapped = false；
- 只要发生一次交换，就令 swapped = true；
- 如果整轮扫描都没有发生交换，说明所有相邻元素都满足升序关系；
- 此时数组已经有序，可以直接结束排序。

例如：

$$[1, 2, 3, 4, 5]$$

第一轮中不会发生任何交换，因此只需扫描一轮即可结束。
该优化使已经有序数组的时间复杂度从 $O(n^2)$ 降为 $O(n)$。

iii) **双向冒泡排序**

双向冒泡排序也称为鸡尾酒排序（Cocktail Shaker Sort）。
普通冒泡排序每轮只从左向右扫描，主要将较大的元素移动到右端。
双向冒泡在每一轮中执行两次扫描：

i. 从左向右扫描，将当前最大元素移动到右端；

ii. 从右向左扫描，将当前最小元素移动到左端。

使用两个边界：

$$[\texttt{left}, \texttt{right}]$$

表示当前尚未排好序的区间。

```java
    public static void cocktailSort(int[] arr) {
        int left = 0;
        int right = arr.length - 1;

        while (left < right) {
            boolean swapped = false;
            // 从左向右：将最大元素移动到 right
            for (int i = left; i < right; i++) {
                if (arr[i] > arr[i + 1]) {
                    swap(arr, i, i + 1);
                    swapped = true;
                }
            }
            right--;

            // 如果没有发生交换，数组已经有序
            if (!swapped) {
                break;
            }
            swapped = false;

            // 从右向左：将最小元素移动到 left
            for (int i = right; i > left; i--) {
                if (arr[i] < arr[i - 1]) {
                    swap(arr, i, i - 1);
                    swapped = true;
                }
            }
            left++;

            // 如果没有发生交换，数组已经有序
            if (!swapped) {
                break;
            }
        }
    }
```

从左向右扫描后：

- 当前未排序区间中的最大元素到达 `right`；
- 该位置已经确定；
- 因此执行 `right--`。

从右向左扫描后：

- 当前未排序区间中的最小元素到达 `left`；
- 该位置已经确定；
- 因此执行 `left++`。

需要特别注意：

```
swapped = false;
```

必须在从右向左扫描之前重新执行。否则，`swapped` 可能仍然保留第一次扫描的 `true`，导致第二次扫描无法正确判断是否发生过交换。

双向冒泡适合数组两端都存在位置偏差较大的元素。例如：

$$[9, 2, 3, 4, 1]$$

从左向右扫描可以较快地把 9 移动到右端，从右向左扫描可以较快地把 1 移动到左端。

但是，双向冒泡并不能保证比较次数减少一半，其平均和最坏时间复杂度仍然是 $O(n^2)$。

复杂度与性质

- 基本冒泡排序：最好、平均和最坏时间复杂度均为 $O(n^2)$；
- 使用提前结束优化后：最好时间复杂度为 $O(n)$，平均和最坏时间复杂度为 $O(n^2)$；
- 双向冒泡排序：最好时间复杂度为 $O(n)$，平均和最坏时间复杂度为 $O(n^2)$；
- 三种实现的额外空间复杂度均为 $O(1)$；
- 冒泡排序是原地排序，不需要创建与输入规模成比例的辅助数组；
- 使用严格比较 `arr[i] > arr[i + 1]` 时，相等元素不会交换，因此冒泡排序是稳定排序。

2) **选择排序（Selection Sort）**

每一轮从未排序部分中找到**最小元素**，再将其与当前起始位置交换。

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
```

```
10
11          if (minIndex != i) {
12              swap(arr, i, minIndex);
13          }
14      }
15  }
```

第 i 轮结束后：`arr[0],...,arr[i]` 已经排好序。

复杂度和性质：

- 最好、平均和最坏时间复杂度均为 $O(n^2)$
- 额外空间复杂度为 $O(1)$
- 通常是不稳定排序（将最小元素与前面的元素直接交换时，可能改变相同元素原有的相对顺序）

3) **插入排序（Insertion Sort）**

把数组分成“左边已经排好序”和“右边还没处理”两部分。每一轮取出右边的第一个元素，把它插入左边合适的位置。

```
1  public static void insertionSort(int[] arr) {
2      for (int i = 1; i < arr.length; i++) {
3          int key = arr[i];
4          int j = i - 1;
5
6          while (j >= 0 && arr[j] > key) {
7              arr[j + 1] = arr[j];
8              j--;
9          }
10
11         arr[j + 1] = key;
12     }
13 }
```

第 i 轮开始时：`arr[0],...,arr[i]` 已经排好序。

复杂度和性质：

- 最好时间复杂度：$O(n)$；
- 平均和最坏时间复杂度：$O(n^2)$；
- 额外空间复杂度：$O(1)$；
- 稳定排序；

- 适合规模较小或接近有序的数组。

4) **归并排序（Merge Sort）**

归并排序采用分治思想：

i. 将数组分成左右两部分；

ii. 分别对左右两部分递归排序；

iii. 将两个有序部分合并为一个有序部分。

```
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // 分治：分别排序左右两部分
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // 合并：将两个有序数组合并
        merge(arr, left, mid, right);
    }
}

private static void merge(int[] arr, int left, int mid, int right) {
    // 创建临时数组
    int[] temp = new int[right - left + 1];
    int i = left;
    int j = mid + 1;
    int k = 0;

    // 比较左右两部分元素，按顺序放入临时数组
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    // 将剩余元素复制到临时数组
    while (i <= mid) {
```

```
32          temp[k++] = arr[i++];
33      }
34      while (j <= right) {
35          temp[k++] = arr[j++];
36      }
37
38      // 将临时数组复制回原数组
39      for (i = 0; i < temp.length; i++) {
40          arr[left + i] = temp[i];
41      }
42  }
```

对数组

$$[5, 3, 8, 4, 2]$$

进行归并排序：

$$\begin{aligned}[5, 3, 8, 4, 2] &\to [5, 3, 8]\ [4, 2] \\ &\to [5, 3]\ [8]\ [4]\ [2] \\ &\to [5]\ [3]\ [8]\ [4]\ [2].\end{aligned}$$

然后逐步合并：

$$\begin{aligned}[5] + [3] &\to [3, 5], \\ [3, 5] + [8] &\to [3, 5, 8], \\ [4] + [2] &\to [2, 4], \\ [3, 5, 8] + [2, 4] &\to [2, 3, 4, 5, 8].\end{aligned}$$

注释

先不断分割，直到每部分只有一个元素；再将相邻的有序部分逐步合并。

复杂度和性质：

- 最好、平均和最坏时间复杂度均为 $O(n \log n)$；
- 额外空间复杂度为 $O(n)$；
- 稳定排序；
- 性能稳定，但需要额外数组。

5) **快速排序（Quick Sort）**

快速排序同样采用分治思想：

i. 选择一个基准值 `pivot`；

ii. 将小于等于基准值的元素放到左侧；

iii. 将大于基准值的元素放到右侧；

iv. 对左右两部分递归排序。

```java
public static void quickSort(int[] arr, int left, int right) {
    if (left < right) {
        // 分区：将数组分为两部分，左边小于pivot，右边大于pivot
        int pivotIndex = partition(arr, left, right);

        // 递归排序左右两部分
        quickSort(arr, left, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, right);
    }
}

private static int partition(int[] arr, int left, int right) {
    // 随机选择pivot，避免最坏情况
    int randomIndex = left + (int)(Math.random() * (right - left + 1));
    swap(arr, randomIndex, right);

    int pivot = arr[right];
    int i = left - 1;

    for (int j = left; j < right; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr, i, j);
        }
    }

    swap(arr, i + 1, right);
    return i + 1;
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

分区结束后：

- 基准值位于最终正确位置；
- 左侧元素均不大于基准值；
- 右侧元素均大于基准值。

对数组

$$[5, 3, 8, 4, 2]$$

选择 5 为基准值：

$$[5, 3, 8, 4, 2] \to [3, 4, 2]\ [5]\ [8].$$

继续递归排序左半部分：

$$[3, 4, 2] \to [2]\ [3]\ [4].$$

最终结果：

$$[2, 3, 4, 5, 8].$$

注释

选择基准值，将较小元素放左边、较大元素放右边，再递归排序左右两部分。

随机选择基准值可降低有序数组等特殊输入导致严重失衡的概率但不能完全消除最坏情况。
复杂度和性质：

- 平均时间复杂度：$O(n \log n)$；
- 最坏时间复杂度：$O(n^2)$；
- 平均递归空间复杂度：$O(\log n)$；
- 最坏递归空间复杂度：$O(n)$；
- 通常是不稳定排序；
- 实际运行中通常具有较好的缓存局部性。

6) **排序算法对比**

算法	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$	不稳定

表 1.1: 常见排序算法复杂度比较

7) **实际开发中的排序**

普通开发中通常直接使用 Java 标准库：

```
int[] numbers = {5, 3, 8, 1};
Arrays.sort(numbers);
```

也可以只排序指定区间：

```
Arrays.sort(numbers, fromIndex, toIndex);
```

其中排序区间为：[fromIndex, toIndex).

去重

输入整数数组，删除重复元素。

1. **有序数组：原地双指针**

输入为有序数组；返回不同元素的数量 k，且前 k 个位置保存去重后的结果。

```
public static int removeDuplicates(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int slow = 0; // slow 保存的是最后一个不重复元素的下标
    for (int fast = 1; fast < nums.length; fast++) {
        if (nums[fast] != nums[slow]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }

    return slow + 1; // 返回新数组长度
}
```

算法： fast 扫描数组，slow 指向最后一个不同元素。发现新元素时，将其写入 slow+1。

$$[1,1,2,2,3] \longrightarrow k = 3, \qquad [1,2,3,\ldots]$$

复杂度：时间 $O(n)$，额外空间 $O(1)$。

2. **无序数组：哈希表**

输入无序数组，返回去重后的列表，并保留元素第一次出现的顺序。

算法：使用两个容器：

- seen：记录已经出现过的元素；

- `result`：按原顺序保存去重结果。

`seen.add(num)` 的返回值表示是否成功加入：

- 返回 `true`：该元素第一次出现，将其加入结果；
- 返回 `false`：该元素已经出现，直接跳过。

```
public static List<Integer> removeDuplicatesUnsorted(
        int[] nums) {
    Set<Integer> seen = new HashSet<>();
    List<Integer> result = new ArrayList<>();

    for (int num : nums) {
        if (seen.add(num)) {
            result.add(num);
        }
    }

    return result;
}
```

对于

$$[3, 1, 3, 2, 1],$$

处理过程为：

当前元素	是否第一次出现	`result`
3	是	[3]
1	是	[3, 1]
3	否，跳过	[3, 1]
2	是	[3, 1, 2]
1	否，跳过	[3, 1, 2]

因此返回：

$$[3, 1, 2].$$

这里由 `HashSet` 判断元素是否重复，由 `ArrayList` 保留第一次出现的顺序。

复杂度： 平均时间 $O(n)$，额外空间 $O(n)$。

最大子数组和

输入整数数组，返回和最大的非空连续子数组之和。

算法： Kadane 算法维护：`dp[i] = max(nums[i], dp[i-1] + nums[i])` then return $\max_i$`(dp[i])`

$$\texttt{currentSum} = \max(\texttt{nums[i]}, \texttt{currentSum} + \texttt{nums[i]})$$

即决定将当前元素加入原子数组，还是从当前元素重新开始。

```java
public static int maxSubArray(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int currentSum = nums[0];
    int maxSum = nums[0];

    for (int i = 1; i < nums.length; i++) {
        currentSum = Math.max(nums[i], currentSum + nums[i]);

        maxSum = Math.max(maxSum, currentSum);
    }

    return maxSum;
}
```

示例：

$$[-2, 1, -3, 4, -1, 2, 1, -5, 4] \longrightarrow [4, -1, 2, 1] \longrightarrow 6$$

复杂度：时间 $O(n)$，额外空间 $O(1)$.

数组旋转

目标：输入数组和整数 k，将数组原地向右旋转 k 位。

算法：三次翻转：

翻转全部 → 翻转前k 个 → 翻转剩余元素.

```java
public static void rotate(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return;
    }

    int n = nums.length;
    k %= n;

    reverse(nums, 0, n - 1);
    reverse(nums, 0, k - 1);
    reverse(nums, k, n - 1);
}
```

```
13
14 private static void reverse(
15         int[] nums, int left, int right) {
16     while (left < right) {
17         int temp = nums[left];
18         nums[left] = nums[right];
19         nums[right] = temp;
20
21         left++;
22         right--;
23     }
24 }
```

示例：

$$[1,2,3,4,5,6,7] \to [7,6,5,4,3,2,1]$$
$$\to [5,6,7,4,3,2,1]$$
$$\to [5,6,7,1,2,3,4].$$

其中 $k = 3$。

复杂度：时间 $O(n)$，额外空间 $O(1)$.

荷兰国旗问题

输入只包含 $0, 1, 2$ 的数组，将其原地排列为：$[0, \ldots, 0, 1, \ldots, 1, 2, \ldots, 2]$。

算法：使用三个指针：

- `low`：下一个 0 应放的位置；
- `mid`：当前检查的位置；
- `high`：下一个 2 应放的位置。

```
1  public static void sortColors(int[] nums) {
2      int low = 0;
3      int mid = 0;
4      int high = nums.length - 1;
5
6      while (mid <= high) {
7          if (nums[mid] == 0) {
8              swap(nums, low, mid);
9              low++;
10             mid++;
11         } else if (nums[mid] == 1) {
12             mid++;
```

```
        } else {
            swap(nums, mid, high);
            high--;
        }
    }
}
```

当 `nums[mid] == 2` 时不能立即执行 `mid++`，因为从右侧交换过来的元素尚未检查。

示例：

$$[2,0,2,1,1,0] \longrightarrow [0,0,1,1,2,2]$$

复杂度：时间 $O(n)$，额外空间 $O(1)$.

1.3.3 多维数组

1. **多维数组的本质**

   Java 的多维数组本质上是**数组的数组**。

   ```
   int[][] matrix = new int[3][4];
   ```

   其中：

   - 外层数组长度为 3，保存 3 个 `int[]` 引用；
   - 每个引用指向一个长度为 4 的一维数组；
   - 所有 `int` 元素默认初始化为 0。

   访问元素时，先选择一行，再选择该行中的元素：

   ```
   matrix[0][0] = 1;
   matrix[1][2] = 10;
   ```

   $$\texttt{matrix.length} = 3, \qquad \texttt{matrix[i].length} = 4.$$

2. **多维数组的初始化**

   1) 静态初始化

   ```
   int[][] matrix = {
       {1, 2, 3},
       {4, 5, 6},
       {7, 8, 9}
   };
   ```

   2) 动态初始化

   ```
   int[][] matrix = new int[3][4];
   ```

3）**先创建外层数组，再分别创建每一行**

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

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

执行

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

后，三行的初始值都是 `null`。必须先创建某一行，才能访问该行元素：

```
matrix[0] = new int[2];
matrix[0][0] = 1;
```

否则会产生 `NullPointerException`。

3. **不规则二维数组**

Java 允许每一行具有不同长度：

```
int[][] irregular = {
    {1, 2},
    {3, 4, 5, 6},
    {7, 8, 9}
};
```

各行长度分别为：

$$2, \quad 4, \quad 3.$$

遍历时必须使用每一行自己的长度：

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

4. **多维数组的遍历**

普通下标遍历：

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
```

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

增强 for 循环：

```
for (int[] row : matrix) {
    for (int value : row) {
        System.out.print(value + " ");
    }
}
```

第一层取出一行 int[]，第二层取出该行中的每个 int。

5. **内存模型**

二维数组变量保存外层数组的引用，外层数组中的每个元素又保存一行数组的引用：

$$\texttt{matrix} \longrightarrow [\texttt{row0}, \texttt{row1}, \texttt{row2}].$$

因此，各行是相互独立的数组对象，可以：

- 具有不同长度；
- 暂时为 null；
- 被重新赋值为新的数组。

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

重新赋值：

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

只会让 arr 指向新的外层数组。原数组若不再被引用，之后可以被垃圾回收。

6. **行优先遍历与访问效率**

通常应先遍历行，再遍历该行中的列：

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        sum += matrix[i][j];
    }
}
```

因为每一行是一个独立的一维数组，连续访问同一行通常具有更好的缓存局部性。

列优先遍历：

```
for (int j = 0; j < matrix[0].length; j++) {
    for (int i = 0; i < matrix.length; i++) {
        sum += matrix[i][j];
    }
}
```

会频繁切换不同的行数组，且只适用于每行长度相同的规则矩阵。

7. **高维数组**

三维数组仍然是数组的数组：

```
int[][][] cube = new int[3][4][5];
```

可以理解为：

$$3\text{ 层} \times 4\text{ 行} \times 5\text{ 列}.$$

访问元素：

```
cube[1][2][3] = 10;
```

8. **应用场景**

- 矩阵：`int[row][col]`；
- 灰度图像：每个元素表示一个像素亮度；
- 彩色图像：`int[height][width][3]`；
- 棋盘：每个元素表示一个格子的状态；
- 数独：使用 9×9 数组保存数字；
- 稀疏矩阵：只记录非零元素的行、列和值。

若 $A \in \mathbb{R}^{m \times n}$，　$B \in \mathbb{R}^{n \times p}$,，则矩阵乘法结果满足：$C \in \mathbb{R}^{m \times p}$，　$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$.

```
public static int[][] multiply(int[][] a, int[][] b) {
    int m = a.length;
    int n = a[0].length;
    int p = b[0].length;
    if (n != b.length) {
        throw new IllegalArgumentException(
            "矩阵维度不匹配"
        );
    }
    int[][] result = new int[m][p];
```

```
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < n; k++) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return result;
}
```

1.3.4 数组工具包

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
```

1. **输出数组：** `toString()` **与** `deepToString()`

   直接输出数组：

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

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

   得到的是数组类型和哈希信息，而不是元素内容。一维数组应使用：

   ```
   System.out.println(Arrays.toString(nums)); // [1, 2, 3]
   ```

   多维数组应使用：

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

   System.out.println(Arrays.toString(matrix)); // [[I@..., [I@...]
   System.out.println(Arrays.deepToString(matrix)); // [[1, 2], [3, 4]]
   ```

   原因是：

   - `toString()` 只读取最外层元素；
   - 二维数组的最外层元素是各行数组的引用；
   - `deepToString()` 会递归读取内部数组。

2. **排序：** `Arrays.sort()` ($O(n \log n)$)

   `sort()` 会直接修改原数组，返回类型为 `void`。

   1) **基本类型数组**

   对于 `int[]`、`long[]` 等基本类型数组，`Arrays.sort()` 按数值升序排列，**直接修改原数组**：

```java
int[] nums = {5, 3, 8, 4, 2};

Arrays.sort(nums);
System.out.println(Arrays.toString(nums)); // [2, 3, 4, 5, 8]

Arrays.sort(nums, 1, 4); // 只排序下标 [1, 4)
System.out.println(Arrays.toString(nums)); // [9, 3, 5, 8, 1]
```

   当前 JDK 的实现说明采用双轴快速排序（Dual-Pivot Quicksort）。它选择两个基准值 $p \le q$，将元素划分为：

$$x < p, \qquad p \le x \le q, \qquad x > q,$$

   再继续排序三个区间。

   基本类型数组不能传入 `Comparator`：

```java
// Arrays.sort(nums, comparator); // 不存在此重载
```

   若需要降序，可以先升序排序，再手动反转数组。该排序不保证稳定性，但相同的基本类型数值没有对象身份，因此通常不影响使用。

$$\text{时间复杂度：} O(n \log n)$$

   2) **对象数组**

   对象数组有两种排序方式。

   1) **自然顺序：** `Comparable`

   元素类型实现了 `Comparable` 时，可以直接调用 `Arrays.sort()`：

```java
String[] words = {"pear", "apple", "orange"};

Arrays.sort(words); // [apple, orange, pear]
```

   自定义类可以通过 `compareTo()` 定义默认顺序：

```java
class Student implements Comparable<Student> {
    String name;
    int score;

    @Override
```

```
    public int compareTo(Student other) {
        return Integer.compare(this.score, other.score);
    }
}
```

2) **自定义顺序：** `Comparator`

排序时可以临时指定比较规则，而不修改类本身：

```
String[] words = {"pear", "fig", "kiwi", "banana"};

Arrays.sort(
    words,
    Comparator.comparingInt(String::length)
); // [fig, pear, kiwi, banana]
```

也可以组合多个条件：

```
Arrays.sort(
    students,
    Comparator
        .comparingInt((Student s) -> s.score)
        .reversed()
        .thenComparing(s -> s.name)
);
```

表示先按分数降序，分数相同时再按姓名升序。

比较结果的含义为：

< 0	第一个对象排在前面
$= 0$	两个对象排序上相等
> 0	第一个对象排在后面

对象数组排序保证稳定：比较结果为 0 的对象会保持原来的相对顺序。例如上例中 `pear` 和 `kiwi` 长度相同，排序后仍保持 `pear` 在前。

当前 JDK 实现使用稳定、自适应的归并排序，能够利用数组中已有的有序片段。

时间复杂度：$O(n \log n)$

3. **并行排序：** `Arrays.parallelSort()`

```
int[] nums = {5, 3, 8, 4, 2};

Arrays.parallelSort(nums);
```

基本思想是：

1) 将大数组划分为多个较小区间；
2) 多个线程同时排序这些区间；
3) 再合并或整理各区间的结果。

并行排序通常使用公共的 Fork/Join 线程池。它不一定比普通排序更快：

- 数组较小时，线程调度成本可能更高；
- 数组很大且 CPU 有多个核心时，才可能明显受益；
- 具体性能应通过实际测试判断，不应死记固定阈值。

其时间复杂度仍可看作：

$$O(n \log n),$$

但并行执行可以缩短实际运行时间。

4. **二分查找：** `Arrays.binarySearch()`

使用前**必须保证数组已经按照相同规则排序**：

```
int[] nums = {1, 3, 5, 7, 9};

int found = Arrays.binarySearch(nums, 5);
System.out.println(found); // 2
```

如果找到目标，返回其下标。
如果没有找到，返回：$-(\text{插入点}) - 1$。
插入点是：将目标插入数组后仍保持有序的位置。
例如查找 4：

$$[1, 3, \square, 5, 7, 9]$$

插入点为 2，因此返回：$-2 - 1 = -3$.

```
int result = Arrays.binarySearch(nums, 4); // result == -3

int insertionPoint = -result - 1; // insertionPoint == 2
```

使用 $-p - 1$ 而不是直接返回 $-p$，是因为插入点可能为 0。如果直接返回 -0，就无法区分：

- 找到了下标 0；
- 没找到，但应插入下标 0。

需要注意：

- **未排序数组上的结果没有意义**；
- 如果存在多个相同元素，不保证返回哪一个；

- 查找时间复杂度为 $O(\log n)$。

5. **比较数组：** `equals()` **与** `deepEquals()`
   数组的 == 比较的是两个变量是否指向同一个数组：

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

System.out.println(a == b);              // false
System.out.println(Arrays.equals(a, b)); // true
```

   `Arrays.equals()` 会检查：

   - 数组长度是否相同；
   - 对应位置的元素是否相等；
   - 元素顺序是否相同。

   对于嵌套数组：

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

System.out.println(Arrays.equals(a, b));     // false
System.out.println(Arrays.deepEquals(a, b)); // true
```

   原因是 `Arrays.equals(a,b)` 只比较最外层元素。二维数组的最外层元素是行数组引用，因此相同行内容但不同的行数组仍会被认为不同。`deepEquals()` 会递归比较所有内部数组。
   对应关系为：

输出	比较
`toString()`	一层
`deepToString()`	递归
`equals()`	一层
`deepEquals()`	递归

   比较时间复杂度最坏为 $O(n)$。

6. **填充数组：** `Arrays.fill()`
   基本类型数组：

```java
int[] nums = new int[5];
Arrays.fill(nums, 10); // [10, 10, 10, 10, 10]
```

   填充部分区间（填充范围同样是：[fromIndex, toIndex)）：

```java
int[] nums = {0, 0, 0, 0, 0};
```

```
Arrays.fill(nums, 1, 4, 9); // [0, 9, 9, 9, 0]
```

对于引用类型，填入的是同一个对象的引用：

```
StringBuilder[] arr = new StringBuilder[3];
Arrays.fill(arr, new StringBuilder()); // 所有元素指向同一个对象（StringBuilder）
arr[0].append("A");
System.out.println(Arrays.toString(arr)); // [A, A, A]

arr[0] = "World"; // 修改的是引用，不是对象内容，修改一个元素不会影响其他元素
```

如果希望每个位置拥有不同对象，应分别创建：

```
StringBuilder[] arr = new StringBuilder[3];
for (int i = 0; i < arr.length; i++) {
    arr[i] = new StringBuilder(); // 每个元素指向不同对象
}
```

也可以使用：

```
Arrays.setAll(
    arr,
    i -> new StringBuilder()
);
```

填充时间复杂度为 $O(n)$。

7. **复制数组**

   (a) `copyOf()` 与 `copyOfRange()`

   `copyOf()` 会**创建一个新数组**：

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

int[] copy = Arrays.copyOf(
    original,
    original.length
);

copy[0] = 100;
System.out.println(Arrays.toString(original)); // [1, 2, 3]
```

   基本类型数组复制的是各元素的数值，所以**两个数组相互独立**。还可以改变数组长度：

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

```
2
3 int[] longer = Arrays.copyOf(original, 5); // [1, 2, 3, 0, 0]
4 int[] shorter = Arrays.copyOf(original, 2); // [1, 2]
```

规则为：

- 新长度较短：截断多余元素；
- 新长度较长：使用默认值补足；
- 基本类型默认值通常为 0；
- 引用类型默认值为 null。

复制部分区间：

```
1 int[] nums = {10, 20, 30, 40, 50};
2
3 int[] part = Arrays.copyOfRange(nums, 1, 4); // [20, 30, 40]
```

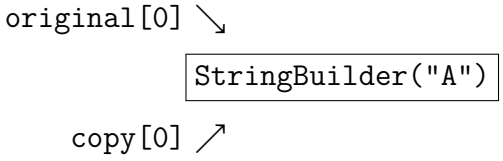
范围为：$[1, 4)$.

(b) 浅拷贝与深拷贝

对于对象数组，copyOf() 创建了新的数组，但数组内部保存的仍是原对象的引用：

```
1  StringBuilder[] original = {
2      new StringBuilder("A"),
3      new StringBuilder("B")
4  };
5
6  StringBuilder[] shallowCopy =
7      Arrays.copyOf(original, original.length);
8
9  shallowCopy[0].append("!");
10
11 System.out.println(original[0]);    // A!
12 System.out.println(shallowCopy[0]); // A!
```

内存关系为：



这叫作**浅拷贝**：

- 外层数组是新的；
- 内部对象仍然共享。

深拷贝必须创建新的内部对象：

```
StringBuilder[] deepCopy = new StringBuilder[original.length];

for (int i = 0; i < original.length; i++) {
    deepCopy[i] = new StringBuilder(
        original[i].toString()
    );
}
```

二维数组也存在同样的问题：

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

int[][] shallow = Arrays.copyOf(matrix, matrix.length);

shallow[0][0] = 100;

System.out.println(matrix[0][0]); // 100
```

因为只复制了外层数组，内部行数组仍然共享。逐行复制可得到独立的二维数组：

```
int[][] deep = new int[matrix.length][];

for (int i = 0; i < matrix.length; i++) {
    deep[i] = Arrays.copyOf(
        matrix[i],
        matrix[i].length
    );
}
```

(c) System.arraycopy()

```
System.arraycopy(
    source, sourceStart,
    destination, destinationStart,
    length
);
```

示例：

```java
int[] source = {1, 2, 3, 4, 5};
int[] destination = new int[5];

System.arraycopy(
    source, 1,
    destination, 0,
    3
); // destination: [2, 3, 4, 0, 0]
```

与 `Arrays.copyOf()` 的区别：

	是否创建数组	主要用途
`Arrays.copyOf()`	自动创建	复制或改变长度（自动创建新的）
`System.arraycopy()`	需要已有目标数组 (destination)	复制到指定位置（直接内存操作，更快）

二者复制引用类型数组时都属于浅拷贝。复制 k 个元素的时间复杂度为：$O(k)$.

8. **数组转列表：** `Arrays.asList()`

对象数组可以转换为固定长度的列表：

```java
Integer[] nums = {1, 2, 3};

List<Integer> list = Arrays.asList(nums);
list.set(0, 10);

System.out.println(Arrays.toString(nums)); // [10, 2, 3]
```

返回的列表由原数组支持，因此：

- **修改列表元素会影响原数组**；
- **修改原数组元素会影响列表**；
- 可以使用 `set()`；
- **不能使用 `add()` 或 `remove()`，因为数组长度固定**。

如果需要**独立**且可改变长度的列表：

```java
List<Integer> list = new ArrayList<>(Arrays.asList(nums));
list.add(4);
```

基本类型数组需要特别注意：

```java
int[] nums = {1, 2, 3};

```

```
List<int[]> list = Arrays.asList(nums); // 错误示范，这是个List of int[]
System.out.println(list.size()); // 1
```

此时整个 int[] 被当作列表中的一个元素，并不是得到：$[1, 2, 3]$. 需要 List<Integer>：

```
List<Integer> list =
    Arrays.stream(nums)
          .boxed()
          .collect(Collectors.toList());
```

9. **数组流：** Arrays.stream()

基本类型数组会得到专用流：

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

IntStream stream = Arrays.stream(nums);
int sum = stream.sum(); // 15
```

常见操作：

```
int max = Arrays.stream(nums).max().getAsInt(); // max
double average = Arrays.stream(nums).average().orElse(0); // mean
// 过滤偶数并转换回数组
int[] even = Arrays.stream(nums).filter(x -> x % 2 == 0).toArray();
```

流操作分成两类：

1) 中间操作：filter()、map()；

2) 终止操作：sum()、max()、toArray()、collect()。

中间操作通常是惰性的，只有执行终止操作时，整个处理过程才真正开始，**流只能使用一次**，需要重新创建：

```
IntStream stream = Arrays.stream(nums);
int sum = stream.sum();
stream.max(); // 错误：流已经被消费
```

需要再次处理时，应重新创建：

```
int sum = Arrays.stream(nums).sum();
int max = Arrays.stream(nums).max().getAsInt();
```

引用类型数组的流操作：**对象数组**会得到 Stream<T>

```
String[] words = {"hello", "java"};
```

```
2
3 List<String> upper = Arrays.stream(words)
4                            .map(String::toUpperCase)
5                            .collect(Collectors.toList());
```

并行流处理：

```
1 long sum = Arrays.stream(nums)
2                  .parallel()
3                  .asLongStream()
4                  .sum();
```

并行流适合：

- 数据规模较大；
- 每个元素的计算相互独立；
- 单个元素的处理成本较高。

小规模数据中，并行调度成本可能使其更慢。

10. **其他常用方法**

1) 按下标生成元素：

```
1 int[] squares = new int[5];
2
3 Arrays.setAll(squares, i -> i * i); // [0, 1, 4, 9, 16]
```

2) 按字典顺序比较数组：

```
1 int r = Arrays.compare(new int[]{1, 2, 3}, new int[]{1, 2, 4}); // result < 0
```

它从左向右寻找第一个不同元素，并根据该位置的大小决定结果。

3) 找到第一个不同位置：

```
1 int i = Arrays.mismatch(new int[]{1, 2, 3}, new int[]{1, 5, 3}); // index == 1
```

如果两个数组完全相同，则返回 -1。

11. **常见错误总结**

- `Arrays.sort()` 会修改原数组；
- `binarySearch()` 前必须先排序；
- 二维数组输出应使用 `deepToString()`；二维数组比较应使用 `deepEquals()`；
- `fill()` 填充对象数组时，所有位置可能指向同一个对象；

- `copyOf()` 复制对象数组时是浅拷贝；
- `asList()` 返回固定长度列表；
- `Arrays.asList(int[])` 会把整个数组当成一个元素；
- Stream 执行终止操作后不能再次使用；
- 并行排序和并行流不保证一定更快。

注释

`Arrays` 类并不是一种新的数组类型，而是一组用于操作普通 Java 数组的静态工具方法。
使用时首先判断操作是否会修改原数组，其次区分基本类型元素与对象引用，最后注意一维数组与嵌套数组需要使用不同的方法。

1.3.5 数组异常报错

数组相关问题主要分为两类：

- **Exception**：通常由错误的索引、空引用或类型转换引起；
- **Error**：通常表示 JVM 资源不足，例如 `OutOfMemoryError`。

1. **数组越界：** `ArrayIndexOutOfBoundsException`
   长度为 n 的数组，合法下标范围为：

$$0 \le \texttt{index} < n.$$

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

int a = arr[-1]; // 越界
int b = arr[3];  // 越界：最大合法下标为 2
```

   最常见的原因是循环条件写成了 `i <= arr.length`：

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

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

   访问外部传入的下标前，可以先检查：

```
if (index >= 0 && index < arr.length) {
```

```
    System.out.println(arr[index]);
} else {
    System.out.println("索引不合法！");
}
```

二分查找中也应避免直接计算：

```
// left + right 可能溢出
int mid = (left + right) / 2;
```

推荐写成：

```
int mid = left + (right - left) / 2;
```

当 left 和 right 均为非负数时，也可以写：

```
int mid = (left + right) >>> 1;
```

后者是另一种安全求中点的方法，但不应简单理解为一定更快。

2. **空指针异常：** NullPointerException

   当数组引用为 null，却尝试读取其长度或元素时，会抛出**空指针异常**：

```
int[] arr = null;

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

   需要注意，**局部变量只声明而未赋值**时：

```
int[] arr;
System.out.println(arr[0]);
```

   这里不是运行时的 NullPointerException，而是编译错误：variable arr might not have been initialized.

   二维数组中，外层数组和每一行需要分别创建：

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

// matrix[0] 仍然是 null
// matrix[0][0] = 1; // NullPointerException

matrix[0] = new int[2];
matrix[0][0] = 1;
```

或者可以完整创建规则矩阵：

```
int[][] matrix = new int[3][4];

matrix[0][0] = 1;
```

安全访问时，应依次检查外层数组、行和下标：

```
if (matrix != null
        && row >= 0 && row < matrix.length && matrix[row] != null
        && col >= 0 && col < matrix[row].length) {
    System.out.println(matrix[row][col]);
}
```

3. **类型转换异常：** ClassCastException
   对象数组中保存的是对象引用。如果对象的实际类型与强制转换的目标类型不一致，就会抛出 ClassCastException：

```
Object[] values = new Object[2];
values[0] = "Hello";

// 实际对象是 String，不能转为 Integer
Integer number = (Integer) values[0];
```

转换前可以使用 instanceof 检查：

```
Object value = values[0];

if (value instanceof String) {
    String text = (String) value;
    System.out.println(text);
}
```

最好的防御方式通常不是频繁强制转换，而是从一开始使用准确的数组类型：

```
String[] words = new String[3];
Integer[] numbers = new Integer[3];
```

4. **数组存储异常：** ArrayStoreException
   Java 数组具有协变性，因此下面的赋值可以编译：

```
Object[] values = new String[3];
```

变量的编译类型是 Object[]，但实际创建的数组仍然是 String[]。

因此只能放入 String 或 null：

```
values[0] = "Hello"; // 正确
values[1] = 100; // ArrayStoreException
```

原因是：

编译器看到 Object[]　　但运行时数组实际是 String[].

该异常与 ClassCastException 的区别是：

- ClassCastException：读取对象后，进行了错误的强制类型转换；
- ArrayStoreException：向对象数组中放入了不兼容的对象。

5. **负数组长度：** NegativeArraySizeException

数组长度不能为负数：

```
int size = -1;

int[] arr = new int[size]; // NegativeArraySizeException
```

如果数组长度来自用户输入或计算结果，应在创建数组前检查：

```
if (size < 0) {
    throw new IllegalArgumentException(
        "数组长度不能为负数"
    );
}

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

6. **泛型数组问题**

Java 不允许直接创建具体泛型类型的数组：

```
List<String>[] lists = new ArrayList<String>[3]; // 编译错误
```

原因是泛型具有类型擦除，而数组在运行时必须知道实际元素类型，两种机制不完全兼容。

通常应使用集合嵌套代替泛型数组：

```
List<List<String>> lists = new ArrayList<>();

lists.add(new ArrayList<>());
lists.get(0).add("Hello");
```

如果确实需要把集合转换为普通数组：

```
List<String> list = Arrays.asList("a", "b", "c");

String[] arr = list.toArray(new String[0]);
```

7. **内存不足：** OutOfMemoryError

   OutOfMemoryError 是 Error，不是普通的 Exception。当 JVM 无法为新数组分配足够内存时，可能抛出：

```
// 不要实际运行
int[] huge = new int[Integer.MAX_VALUE];
```

   常见原因包括：

   - 一次创建过大的数组；重复复制大数组
   - 内存泄漏（长期保存不再需要的数组引用）
   - 无限制增长的缓存或集合

```
// 错误1：创建超大数组
int[] hugeArray = new int[Integer.MAX_VALUE]; // 抛出OutOfMemoryError

// 错误2：内存泄漏
public class MemoryLeak {
    private static List<byte[]> cache = new ArrayList<>();

    public static void addToCache() {
        byte[] data = new byte[1024 * 1024]; // 1MB
        cache.add(data); // 加入静态列表，永远不会被回收
    }

    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {
            addToCache(); // 最终会导致内存溢出
        }
    }
}
```

   常见解决方式：

   - 分批或流式处理数据；
   - 使用弱引用（WeakReference）避免内存泄漏
   - 调整 JVM 堆大小。

```java
// 分批处理大数组
int batchSize = 1000;
for (int i = 0; i < totalSize; i += batchSize) {
    int size = Math.min(batchSize, totalSize - i);
    int[] batch = new int[size];
    // 处理batch...
}

// 使用弱引用（WeakReference）避免内存泄漏
// 当系统进行垃圾回收时，无论内存是否充足，
// 只要弱引用指向的对象没有被其他强引用关联，该对象就会被回收
import java.lang.ref.WeakReference;

public class Cache {
    private static Map<String, WeakReference<byte[]>> cache = new HashMap<>();

    public static void put(String key, byte[] data) {
        cache.put(key, new WeakReference<>(data));
    }

    public static byte[] get(String key) {
        WeakReference<byte[]> ref = cache.get(key);
        return ref != null ? ref.get() : null;
    }
}
```

可以通过 JVM 参数设置最大堆内存：

```
java -Xmx2g Main
```

但增加堆内存只能缓解内存不足，不能修复无限增长的缓存或引用泄漏。

`WeakReference` 也不是通用解决方案：弱引用指向的对象可能随时被回收，并且容器中的键和弱引用对象本身仍可能继续保留。缓存通常更适合使用容量限制和淘汰策略。

8. **数组报错的排查顺序**

遇到数组报错时，可以依次检查：

1) 数组引用是否为 `null`；
2) 下标是否满足 $0 \le$ `index` $<$ `length`；
3) 二维数组的当前行是否已经创建；

4) 对象的实际类型是否与强制转换类型一致；

5) 对象是否能存入数组的实际运行时类型；

6) 数组长度是否为非负数；

7) 数组规模和引用数量是否超过内存容量。

1.4 Java 面向对象编程

错题集

1.4.1 对象概述

Java 是一门面向对象的编程语言。面向对象编程（Object-Oriented Programming, OOP）通过**类**描述一类事物，再根据类创建具体的**对象**，并让不同对象相互协作完成程序功能。

1. 面向对象编程

面向过程和面向对象是两种不同的程序设计思维：

- **面向过程**：关注完成任务需要经过哪些步骤，即“怎么做”；
- **面向对象**：关注任务应该交给哪个对象完成，即“谁来做”。

例如，设计一个购物系统时：

- 面向过程通常将程序拆分为 `login()`、`search()`、`buy()`、`pay()` 等函数；
- 面向对象通常将系统拆分为 `User`、`Product`、`Order`、`Payment` 等类。

面向对象具有以下优点：

- 将复杂问题拆分为多个职责明确的对象；
- 类和对象之间相对独立，便于维护和修改；
- 已经定义的类可以重复使用；
- 程序结构更加接近对现实世界事物的描述。

2. 类与对象

- **类（class）**

  类是对一类具有相同属性和行为的事物的抽象描述。

  例如，所有学生通常都具有姓名、年龄和成绩，并且都可以学习和参加考试，因此可以定义 `Student` 类：

```java
class Student {
    String name;
    int age;
    double score;

    void study() {
        System.out.println(name + "正在学习");
    }

    void exam() {
        System.out.println(
            name + "参加了考试，成绩是" + score
        );
    }
```

```
}
```

Student 只是描述学生应该具有哪些数据和行为，并不表示某个具体学生。

- **对象（object）**

  对象是根据类创建出来的具体实例（instance）。

```
Student s1 = new Student();
Student s2 = new Student();
```

s1 和 s2 引用两个不同的 Student 对象。

每个对象都可以具有自己的属性值：

```
s1.name = "张三";
s1.age = 20;
s1.score = 85.5;

s2.name = "李四";
s2.age = 22;
s2.score = 92.0;
```

修改 s1 所引用对象的属性，不会自动修改 s2 所引用对象的属性。

3. 类的定义

类的基本语法格式为：

```
访问修饰符 class 类名 {
    // 成员变量

    // 构造器

    // 成员方法
}
```

类名通常采用**大驼峰命名法**：

```
Student
BankAccount
ShoppingCart
```

对于普通顶层类，通常写成：

```
public class Student {
}
```

或者省略访问修饰符：

```
class Student {
}
```

普通顶层类不能使用 private 或 protected 修饰。

如果一个顶层类被声明为 public，则源文件名必须与类名相同：

```
// 文件名：Student.java
public class Student {
}
```

4. 类的组成部分

一个普通类主要由成员变量、构造器和成员方法组成。

1) **成员变量**

成员变量用于描述对象的状态或特征：

```
class Student {
    String name;
    int age;
    double score;
}
```

每创建一个 Student 对象，该对象通常都会拥有自己独立的 name、age 和 score。

没有显式赋值时，成员变量会获得默认值：

- 整数类型：0；
- 浮点类型：0.0；
- boolean：false；
- char：'\u0000'；
- 引用类型：null。

2) **成员方法**

成员方法用于描述对象能够执行的行为：

```
void study() {
    System.out.println(name + "正在学习");
}
```

调用成员方法时，**该方法可以访问当前对象中的成员变量**：

```
Student s1 = new Student();
s1.name = "张三";
```

```
3
4 s1.study(); // 张三正在学习
```

3) **构造器**

构造器用于在创建对象时初始化成员变量（如果一个类中没有显式定义任何构造器，Java 编译器会自动提供一个无参数构造器）：

```
1 Student(String name, int age, double score) {
2     this.name = name;
3     this.age = age;
4     this.score = score;
5 }
```

构造器具有以下特点：

- **构造器名称必须与类名相同**；
- 构造器**没有返回值类型**，不能写 `void`；
- **使用 `new` 创建对象时，构造器会自动执行**；
- 一个类**可以定义多个参数不同的构造器**。

下面的方法不是构造器：

```
1 void Student(String name) {
2     this.name = name;
3 }
```

因为它具有返回值类型 `void`，所以只是一个名为 `Student` 的普通方法。

5. `this` 关键字

`this` 表示**当前对象**，**当成员变量和局部变量同名**时，可以使用 `this` 区分二者：

```
1 class Student {
2     String name;
3     int age;
4
5     Student(String name, int age) {
6         this.name = name;
7         this.age = age;
8     }
9 }
```

其中：

- `this.name` 表示当前对象的成员变量；

- `name` 表示构造器接收到的参数；
- `this.name = name` 表示将参数 `name` 赋给当前对象的成员变量。

如果参数名与成员变量名不同，可以省略 `this`：

```
Student(String studentName) {
    name = studentName;
}
```

但实际开发中，通常让参数名与成员变量名保持一致，并使用 `this` 明确表示当前对象的成员。

6. 对象的创建与使用

创建对象的基本语法为：

```
类名 变量名 = new 类名(构造参数);
```

例如：

```
Student s1 = new Student("张三", 20, 85.5);
```

这条语句可以拆分为：

```
Student s1;
s1 = new Student("张三", 20, 85.5);
```

创建对象后，使用点运算符访问对象成员：

```
对象引用.成员变量
对象引用.成员方法()
```

例如：

```
System.out.println(s1.name);
s1.study();
```

完整展示（定义，创建，调用）：

```
class Student {
    String name;
    int age;
    double score;

    Student(String name, int age, double score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }
```

```
11
12      void study() {
13          System.out.println(name + "正在学习");
14      }
15
16      void exam() {
17          System.out.println(
18              name + "参加了考试，成绩是" + score
19          );
20      }
21  }
22
23  public class Main {
24      public static void main(String[] args) {
25          Student s1 = new Student("张三", 20, 85.5);
26          Student s2 = new Student("李四", 22, 92.0);
27          s1.study();
28          s2.exam();
29      }
30  }
```

输出为：

```
1  张三正在学习
2  李四参加了考试，成绩是92.0
```

7. 引用变量与对象

对于：

```
1  Student s1 = new Student();
```

严格来说：

- `new Student()` 创建的是对象；
- `s1` 是引用变量，保存的是对该对象的引用，而不是完整对象本身。

可以近似理解为：

$$\texttt{s1} \longrightarrow \texttt{Student 对象}.$$

日常表达中经常说“s1 是一个对象”，但更准确的说法是：

s1 引用了一个 Student 对象。

引用不等于对象本身。

将一个对象引用赋给另一个引用变量时，复制的是**引用的值**，不会自动复制对象：

```
Student s1 = new Student("张三", 20, 85.5);
Student s2 = s1;
```

此时：

$$\texttt{s1} \longrightarrow \text{Student 对象}, \texttt{s2} \longrightarrow \text{同一个 Student 对象}.$$

因此，通过任意一个引用修改对象，另一个引用都能观察到变化：

```
s2.name = "李四";

System.out.println(s1.name); // 李四
System.out.println(s2.name); // 李四
```

这里并不是修改了两个对象，而是 s1 和 s2 共同引用同一个对象。

注释

引用变量可以理解为对象的“钥匙”。将 s1 赋给 s2，相当于复制了一把能够打开同一栋房子的钥匙，而不是重新建造了一栋房子。

重新给引用变量赋值，只会改变该变量指向哪个对象：

```
Student s1 = new Student("张三", 20, 85.5);
Student s2 = s1;
s2 = new Student("李四", 22, 92.0);
```

执行最后一条语句后：

$$\texttt{s1} \longrightarrow \text{张三对象},$$
$$\texttt{s2} \longrightarrow \text{李四对象}.$$

此时创建了两个不同的对象。

修改 s2 所引用的对象，不会影响 s1 所引用的对象：

```
s2.name = "王五";

System.out.println(s1.name); // 张三
System.out.println(s2.name); // 王五
```

8. null 引用

引用变量可以保存特殊值 null，**表示当前没有引用任何对象**：

```
Student student = null;
```

可以判断引用是否为 null：

```
if (student == null) {
    System.out.println("当前没有学生对象");
}
```

如果通过 null 引用访问成员，会抛出 NullPointerException：

```
Student student = null;

student.study(); // NullPointerException
```

因此，在引用可能为空时，应先进行检查：

```
if (student != null) {
    student.study();
}
```

9. 对象在内存中的分布

初学阶段可以将 Java 运行时内存简化为：

- 栈（Stack）；
- 堆（Heap）；
- 方法区（Method Area）。

1) 栈（Stack）

栈中主要保存方法执行期间产生的局部变量。

```
public static void main(String[] args) {
    int age = 20;
    Student s = new Student();
}
```

其中：

- 局部基本类型变量 age 直接保存数值 20；
- 局部引用变量 s 保存对象引用；
- 方法结束后，对应的局部变量和栈帧会被**移除**。

2) 堆（Heap）

使用 new 创建的对象通常存放在堆中：

```
Student s = new Student("张三", 20, 85.5);
```

可以近似表示为：

栈中的 s ⟶ 堆中的 Student 对象
⟶ name = " 张三"
⟶ age = 20
⟶ score = 85.5

对象中的普通成员变量属于对象的一部分，因此也随着对象保存在堆中。

3) **方法区（Method Area）**

方法区是 JVM 规范中的逻辑概念，用于保存类级别的信息，例如：

- **类名和父类**信息；
- **成员变量**的描述信息；
- **构造器和方法**的描述信息；
- 方法字节码；
- 运行时常量池等。

案例：

```
class Student {
    String name;
    int age;

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

    void study() {
        System.out.println(name + "正在学习");
    }
}
```

方法区中保存的是类似以下**类级别的信息**：

- 类名为 Student；
- 包含 name 和 age 两个成员变量；
- 包含一个构造器和一个 study() 方法；
- 保存这些方法对应的字节码。

具体对象中的 name = " 张三"、age = 20 等实际数据存放在堆中，而不是方法区中。

注释

方法区保存类的“设计信息”，堆保存根据类创建出来的对象。

10. 对象的生命周期与垃圾回收

对象通过 new 创建后存放在堆中：

```
Student s = new Student();
```

当对象不再能够通过任何有效引用访问时，该对象就可能成为垃圾回收器（Garbage Collector, GC）的回收对象。

例如：

```
Student s = new Student();
s = null;
```

执行 s = null 后，如果没有其他引用指向原来的 Student 对象，该对象就不再可达。

也可能由于引用改为指向另一个对象，导致原对象不再可达：

```
Student s = new Student("张三", 20, 85.5);
s = new Student("李四", 22, 92.0);
```

如果没有其他引用指向原来的“张三对象”，该对象就可能被 GC 回收。

需要注意：

- 程序员不能确定对象被回收的准确时间；
- 变量失去引用不等于对象立刻被销毁；
- JVM 会在合适的时间自动执行垃圾回收。

11. 面向对象的分析步骤

面对一个程序设计问题时，可以按照以下步骤分析：

1) 分析问题中涉及哪些重要事物或角色；
2) 将具有共同特征和行为的事物抽象为类；
3) 提取每个类的属性，即对象需要保存哪些数据；
4) 提取每个类的方法，即对象能够完成哪些行为；
5) 分析不同对象之间如何建立联系和相互协作；
6) 根据类创建对象，并调用对象的方法完成任务。

1.4.2 类的成员

类的成员是直接声明在类体中的组成部分，主要包括：

- 属性（Field）；
- 方法（Method）；
- 构造器（Constructor）；
- 代码块（Initializer Block）；
- 内部类（Inner Class）。

属性（Field）

属性也称为**成员变量**，用于保存类或对象的状态。

1. 属性的声明

属性直接声明在类中、方法之外，基本格式为：

```
访问修饰符 [static] [final] 数据类型 属性名 [= 初始值];
```

例如：

```
class Student {
    private String name;          // 实例变量
    private int age = 18;         // 实例变量，显式初始化
    static int totalCount = 0;    // 类变量
    static final String SCHOOL = "ABC School"; // 类常量
}
```

- `static` 表示该属性属于**类变量**；
- `final` 表示该属性**只能被赋值一次**。

2. 实例变量、类变量与局部变量

```
class Student {
    String name;              // 实例变量
    static int totalCount;    // 类变量

    void study(String course) { // course：形参，也是局部变量
        int hours = 2;          // 局部变量
        System.out.println(name + "学习" + course + hours + "小时");
    }
}
```

成员与局部变量的区别：成员变量在 heap 或 method 有默认值，局部变量在 stack 没有默认值。

对比项	实例变量	类变量	局部变量
声明位置	类中，方法外	类中，方法外	方法/构造器/代码块
`static`	无	有 `static`	不能使用
所属对象	每个对象各有一份	整个类共享一份	属于一次方法调用
默认初始化	有	有	没有，使用前必须赋值
生命周期	随对象存在	随类的加载状态存在	随所在作用域存在
内存位置	随对象存放在堆 Heap 中	属于类级别数据，逻辑上归入方法区 Method（具体实现取决于 JVM）	位于方法调用的栈帧 Stack 中

表 1.2: 实例变量、类变量与局部变量的比较

3. 实例变量与类变量

实例变量属于具体对象，每个对象具有独立的数据：

```
class Student {
    String name;
}

Student s1 = new Student();
Student s2 = new Student();
s1.name = "张三";
s2.name = "李四";
```

类变量由同一个类的所有对象共享，通常使用类名访问：

```
class Student {
    String name;
    static int totalCount = 0;

    Student(String name) {
        this.name = name;
        totalCount++;
    }
}
Student s1 = new Student("张三");
Student s2 = new Student("李四");
System.out.println(Student.totalCount); // 2
```

注释

实例变量表示“每个对象自己的数据”，类变量表示“整个类共同拥有的数据”。

4. 成员变量的默认初始化

对象创建后，尚未显式赋值的成员变量会获得默认值：

- 整数类型：0；
- 浮点类型：0.0；
- char：'\u0000'；
- boolean：false；
- 引用类型：null。

```
class DefaultValue {
    int number;
    double price;
    boolean active;
    String name;

    void show() {
        System.out.println(number); // 0
        System.out.println(price);  // 0.0
        System.out.println(active); // false
        System.out.println(name);   // null
    }
}
```

局部变量必须先赋值再使用：

```
void test() {
    int number;
    // System.out.println(number); // 编译错误：number 尚未初始化

    number = 10;
    System.out.println(number);     // 正确
}
```

5. 属性的封装

通常将属性声明为 private，再通过方法控制读取和修改：

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

```
3
4       public int getAge() {
5           return age;
6       }
7
8       public void setAge(int age) {
9           if (age >= 0 && age <= 150) {
10              this.age = age;
11          } else {
12              throw new IllegalArgumentException("年龄不合法");
13          }
14      }
15  }
```

封装的作用：

- 隐藏对象内部实现；
- 防止外部代码随意写入非法数据；
- 将校验、转换等规则集中在方法中；
- 修改内部实现时，尽量不影响外部调用代码。

方法（Method）

方法用于描述类或对象能够执行的行为。Java 中的方法必须声明在类、接口等类型内部，不能独立存在。

1. 方法的声明结构

```
1  访问修饰符 [static] 返回值类型 方法名(参数列表) {
2      方法体;
3      return 返回值;
4  }
```

例如：

```
1  public int add(int a, int b) {
2      int result = a + b;
3      return result;
4  }
```

一个方法主要包含：

- 访问修饰符：控制方法的可见范围；
- 返回值类型：规定方法返回的数据类型；

- 方法名：通常采用小驼峰命名法；
- 参数列表：调用者传入的数据；
- 方法体：方法执行的具体逻辑。

`void` 表示方法不返回结果：

```java
public void sayHello() {
    System.out.println("Hello");
}
```

2. 参数与返回值

根据是否有参数和返回值，可以得到四种基本形式：

```java
// 无参数、无返回值
void sayHi() {
    System.out.println("Hi");
}

// 有参数、无返回值
void printSum(int a, int b) {
    System.out.println(a + b);
}

// 无参数、有返回值
String getName() {
    return "张三";
}

// 有参数、有返回值
double circleArea(double radius) {
    return Math.PI * radius * radius;
}
```

需要注意：

- 非 `void` 方法必须保证所有正常执行路径都能返回相应类型的值；
- `return 值;` 会结束方法并返回结果；
- `void` 方法中可以使用 `return;` 提前结束方法。

3. 实例方法与静态方法

实例方法属于对象，调用时通常**需要对象引用**：

```
class Student {
    String name;

    void study() {
        System.out.println(name + "正在学习");
    }
}

Student s = new Student();
s.name = "张三";
s.study();
```

静态方法属于类，通常**使用类名调用**：

```
class MathUtil {
    static int square(int number) {
        return number * number;
    }
}

int result = MathUtil.square(5);
```

静态方法没有当前对象，因此不能直接使用 this，也不能直接访问实例变量或实例方法。

4. 方法调用

无返回值的方法可以单独调用：

```
student.study();
```

有返回值的方法可以参与赋值、输出或其他表达式：

```
double area = circle.circleArea(5.0);
System.out.println(circle.circleArea(5.0));
double total = circle.circleArea(5.0) + 10;
```

每调用一次方法，当前线程的栈中就会创建相应的栈帧，用于保存参数、局部变量和方法执行状态；方法结束后，该栈帧被移除。

5. 方法重载（Overload）

同一个类中可以定义多个同名方法，只要它们的**参数列表不同**：

```
class Calculator {
    int add(int a, int b) {
```

```
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }

    String add(String text, int number) {
        return text + number;
    }
}
```

参数列表不同可以表现为：

- 参数个数不同；
- 参数类型不同；
- 参数类型的排列顺序不同。

6. 可变参数（Varargs）

可变参数允许方法接收零个或多个同类型实参：

```
void printNames(String... names) {
    for (String name : names) {
        System.out.println(name);
    }
}

printNames();
printNames("张三");
printNames("张三", "李四", "王五");
```

在方法内部，names 按照数组处理。规则：

- 格式为 数据类型... 参数名；
- 一个方法最多只能有一个可变参数；
- 可变参数必须位于参数列表最后；
- 调用时也可以直接传入同类型数组。

7. Java 的参数传递（内存）

Java 只有值传递。 调用方法时，实参的值会复制给形参。

基本类型传递的是数据值的副本：

```
static void change(int number) {
    number = 100;
}

int number = 10;
change(number);
System.out.println(number); // 10
```

引用类型传递的是**引用值的副本**：

```
class Person {
    String name;
}

static void rename(Person person) {
    person.name = "李四";
}

Person p = new Person();
p.name = "张三";
rename(p);
System.out.println(p.name); // 李四
```

形参 person 和实参 p 保存的是两个引用副本，但这两个引用指向同一个对象，因此通过形参修改对象内容时，调用者可以观察到变化。

重新给形参赋值不会改变实参的指向：

```
static void replace(Person person) {
    person = new Person();
    person.name = "王五"; // 把 new Person() 改成“王五”了
}

Person p = new Person();
p.name = "张三";
replace(p);
System.out.println(p.name); // 张三（并没有把p改成“王五”）
```

注释

方法收到的是副本：基本类型参数收到“数据的副本”；引用类型参数收到“引用的副本”。

8. 递归方法

递归是方法在执行过程中直接或间接调用自身：

```
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;                  // 递归出口
    }
    return n * factorial(n - 1); // 向出口推进
}
```

正确的递归通常需要：

- 明确的递归出口；
- 每次调用都向出口推进；
- 问题可以拆成规模更小的同类问题。

递归调用会不断创建新的栈帧。递归层数过深时可能抛出 `StackOverflowError`。

构造器（Constructor）

构造器是类中的特殊成员，用于在创建对象时完成初始化。严格来说，内存分配由 `new` 和 JVM 完成，构造器负责初始化新对象。

1. 构造器的声明

```
访问修饰符 类名(参数列表) {
    初始化语句;
}
```

例如：

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

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

构造器具有以下特点：

- 构造器名称**必须与类名完全相同**；
- 构造器**没有返回值**类型，连 `void` 也不能写；
- 创建对象时会自动调用与实参匹配的构造器；
- 构造器可以重载；
- 构造器**不能使用** `static`、`final`、`abstract` **等修饰符**。

2. 默认构造器与显式构造器

当类中没有声明任何构造器时，编译器会自动提供一个无参构造器：

```
class Animal {
    int legs;
    String name;
}

Animal animal = new Animal();
```

它可以近似理解为：

```
Animal() {
    super();
}
```

只要程序员声明了任意构造器，编译器就不再自动提供无参构造器。

```
class Car {
    Car(String color) {
    }
}

Car c1 = new Car("红色"); // Car c2 = new Car(); // 编译错误：不存在无参构造器
```

需要无参构造器时，应当主动声明：

```
class Car {
    Car() {
    }

    Car(String color) {
    }
}
```

3. 构造器重载

构造器也可以根据不同参数提供多种初始化方式：

```
class Student {
    String name;
    int age;
    String school;

    Student() {
    }

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

    Student(String name, int age, String school) {
        this.name = name;
        this.age = age;
        this.school = school;
    }
}
```

调用哪个构造器由实参列表决定：

```
Student s1 = new Student();
Student s2 = new Student("张三", 20);
Student s3 = new Student("李四", 21, "ABC University");
```

4. 使用 this(...) 调用本类构造器

一个构造器可以使用 this(...) 调用本类中另一个构造器，减少重复代码：

```
class Student {
    String name;
    int age;
    String school;

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

```

```
    Student(String name, int age, String school) {
        this(name, age); // 调用另一个构造器
        this.school = school;
    }
}
```

规则：

- `this(...)` 只能写在构造器中；
- `this(...)` 必须是构造器中的第一条语句；
- 一个构造器**不能直接或间接调用自身**，否则会形成递归构造调用；
- 同一个构造器中不能同时显式写 `this(...)` 和 `super(...)`。

5. 对象的初始化顺序

只考虑当前类、不考虑继承时，实例部分可以按以下顺序理解：

1) **加载类**：JVM 加载 Person 类的字节码；

2) **分配空间**：在堆内存中为 Person 对象分配空间；

3) **属性初始化**

   i. 默认初始化：name=null，age=0；

   ii. 显式初始化：若有 String name=” 默认名”; 则执行；

4) **调用构造器**

   i. 传入参数” 张三”、20；

   ii. 执行构造器内的赋值语句：name=” 张三”，age=20；

5) **返回引用**：栈内存中 p 指向堆中的对象

创建对象后，还可以再次修改对象属性：

```
class InitOrder {
    int number = 10;        // 显式初始化
    InitOrder() {
        number = 30;        // 构造器初始化
    }
}

InitOrder obj = new InitOrder();
obj.number = 40;            // 创建对象后再次赋值
```

最终 `obj.number` 为 40。

6. JavaBean 的基础形式

传统 JavaBean 通常具有以下结构：

- 类通常声明为 `public`；
- 提供公共无参构造器；
- 属性通常声明为 `private`；
- 通过 getter 和 setter 访问属性。

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

    public UserBean() {
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

代码块（Initializer Block）

类体中常见的初始化代码块分为静态代码块和非静态代码块。它们没有名称，也不能被程序员像普通方法一样直接调用。

1. 静态代码块

静态代码块使用 `static` 修饰：

```
class Config {
```

```
    static String version;

    static {
        version = "1.0.0";
        System.out.println("加载配置");
    }
}
```

特点：

- 属于类级别的初始化逻辑
- **在类初始化阶段执行，通常只执行一次**
- 可以直接访问静态成员，**不能直接访问实例成员或使用** `this`

常见用途包括初始化复杂静态数据、读取配置等。对于可能失败或需要释放资源的操作，通常不应随意塞入静态代码块。

2. 非静态代码块

非静态代码块也称为实例初始化块：

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

    {
        age = 18;
        System.out.println("执行非静态代码块");
    }

    Person(String name) {
        this.name = name;
        System.out.println("执行构造器");
    }
}
```

特点：

- **每次创建对象时都会执行，且先于构造器体执行**
- 可以访问实例成员和静态成员
- 适合保存多个构造器都需要执行的公共初始化逻辑

不过，简单的公共初始化通常也可以提取成普通私有方法，这种写法往往更直观。

3. 完整执行顺序

同一个类中的字段初始化表达式和代码块会按照它们在源代码中的出现顺序执行。

```
class InitDemo {
    static int staticNumber = print("1. 静态变量初始化");

    static {
        print("2. 静态代码块");
    }

    int number = print("3. 实例变量初始化");

    {
        print("4. 非静态代码块");
    }

    InitDemo() {
        print("5. 构造器");
    }

    static int print(String message) {
        System.out.println(message);
        return 0;
    }
}

InitDemo d1 = new InitDemo();
InitDemo d2 = new InitDemo();
```

输出顺序为：

```
1. 静态变量初始化
2. 静态代码块
3. 实例变量初始化
4. 非静态代码块
5. 构造器
3. 实例变量初始化
4. 非静态代码块
5. 构造器
```

4. 局部代码块

方法内部也可以使用普通代码块限制局部变量的作用域：

```
void test() {
    {
        int number = 10;
        System.out.println(number);
    }

    // System.out.println(number); // 编译错误：已超出作用域
}
```

局部代码块不是类的成员，也不会自动执行；程序执行到该位置时才会进入代码块。

内部类（Inner Class）

将一个类声明在另一个类或方法内部，可以使相关逻辑集中在一起，并进一步限制类的可见范围。

1. 内部类的分类

常见形式包括：

- 非静态成员内部类；
- 静态嵌套类：使用 `static` 声明的成员类称为**静态嵌套类（static nested class）**
- 局部类；
- 匿名内部类。

2. 非静态成员内部类

非静态成员内部类直接声明在外部类中，并依赖某个外部类对象：

```
class Outer {
    private int value = 10;

    class Inner {
        void show() {
            System.out.println(value);
        }
    }
}
```

在外部类内使用：当作普通类创建对象

```
public void outerMethod() {
    InnerClass inner = new InnerClass(); // 直接创建内部类对象
    inner.innerMethod();
```

```
}
```

在外部类外部使用：必须通过外部类对象创建内部类对象

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
inner.show();
```

特点：

- 可以直接访问外部类的实例成员，包括 private 成员
- 不能包含静态成员（如 static int num;），因为它依赖外部类实例存在
- 可以使用 Outer.this 明确表示关联的外部类对象

例如，当内外部类成员同名时：

```
class Outer {
    int number = 10;

    class Inner {
        int number = 20;

        void show() {
            int number = 30;
            System.out.println(number);            // 30
            System.out.println(this.number);       // 20
            System.out.println(Outer.this.number); // 10
        }
    }
}
```

3. 静态嵌套类

静态嵌套类不依赖外部类对象：

```
class Outer {
    private static int value = 10;
    private int instanceValue = 20;

    static class Nested {
        void show() {
            System.out.println(value);
            // System.out.println(instanceValue); // 不能直接访问
```

```
        }
    }
}
```

创建方式：

```
Outer.Nested nested = new Outer.Nested();
nested.show();
```

特点：

- **不能访问外部类的非静态成员**（因为静态内部类“出生”时，外部类可能还没实例化）
- 可以包含静态成员（如 static void method()），相当于外部类的“独立房客”

4. 局部类

局部类声明在方法或代码块中，只能在其作用域内使用：

```
public class Outer {
    private int outerField=10;
    public void outerMethod(){
        final int num = 1; //可以隐藏这里的final
        class Inner{
            public void innerMethod(){
                //如果要在局部内部类中访问方法中的局部变量，必须是final类型修饰的变量
                System.out.println("访问外部类的属性：" + outerField + num);
            }
        }
        InnerClass innerClass=new InnerClass(); //只能在当前方法中使用
        innerClass.innerMethod();
    }
}
```

局部类访问方法中的局部变量时，该变量必须是 **final 或 effectively final（事实上的 final）**：指变量虽然没有显式写 final，但初始化后没有再次赋值。

5. 匿名内部类

匿名内部类没有显式类名，通常用于只需要一次的简单实现：

```
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("执行任务");
    }
```

```
};

task.run();
```

它表示：

- 定义一个实现 Runnable 的匿名类；
- 立即创建该匿名类的一个对象；
- 将对象引用赋给 task。

匿名内部类也可以继承一个普通类：

```
Animal animal = new Animal() {
    @Override
    void sound() {
        System.out.println("特殊叫声");
    }
};
```

对于只有一个抽象方法的函数式接口，现代 Java 通常优先使用 Lambda：

```
Runnable task = () -> System.out.println("执行任务");
```

特点：

- 没有构造器（因为没有类名），只能创建一个实例
- 必须继承一个父类或实现一个接口

6. 内部类对比

类型	声明位置	与外部类对象的关系	典型创建方式
非静态成员内部类	外部类的类体中	依赖一个外部类对象	`outer.new Inner()`
静态嵌套类	外部类的类体中	不依赖外部类对象	`new Outer.Nested()`
局部类	方法或代码块中	只在声明作用域内可见	在所在作用域内 `new`
匿名内部类	表达式中	没有显式类名	`new 接口或父类 (){...}`

表 1.3: 常见内部类形式比较

7. 为什么使用内部类

- 封装性增强：内部类可以访问外部类的 private 成员，实现“内部操作自由”
- 代码结构清晰：当一个类仅为另一个类服务时，内部类让逻辑更集中
- 简化匿名对象：特别是在事件处理、回调函数中，匿名内部类避免创建多余类

1.4.3 抽象类（Abstract）

抽象类用于表示一种**不完整的父类模型**：父类规定所有子类必须具备哪些行为，但将具体实现交给不同子类完成。

1. 抽象类

使用 abstract 修饰的类称为抽象类：

```
abstract class Animal {
    String name;

    public void eat() {
        System.out.println(name + "正在吃饭");
    }

    public abstract void shout();
}
```

抽象类具有以下特点：

- **不能直接使用 new 创建对象**
- 可以包含成员变量、构造器和普通方法
- **可以包含抽象方法，也可以完全没有抽象方法**
- 主要作为父类使用，为子类提供统一结构

```
Animal animal = new Animal(); // 编译错误
```

2. 抽象方法

抽象方法只规定方法的声明，不提供方法体：

```
public abstract void shout();
```

抽象方法具有以下特点：

- 使用 abstract 修饰；
- 没有方法体，以分号结尾；
- 包含抽象方法的类必须是抽象类；
- 不能使用 private、static 或 final 修饰。

子类实现抽象方法

普通子类必须实现父类中所有未实现的抽象方法

```
class Dog extends Animal {
    @Override
```

```
    public void shout() {
        System.out.println("汪汪汪");
    }
}

class Cat extends Animal {
    @Override
    public void shout() {
        System.out.println("喵喵喵");
    }
}
```

如果子类没有实现全部抽象方法，该子类也必须声明为抽象类：

```
abstract class Bird extends Animal {
}
```

3. 抽象类与多态

虽然不能直接创建抽象类对象，但可以使用抽象类引用指向具体子类对象：

```
Animal dog = new Dog();
Animal cat = new Cat();

dog.shout(); // 汪汪汪
cat.shout(); // 喵喵喵
```

实际调用哪个方法，由引用所指向的具体对象决定。

4. 抽象类的构造器

抽象类可以定义构造器，用于初始化从父类继承的成员：

```
abstract class Person {
    String name;

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

class Student extends Person {
    public Student(String name) {
        super(name);
```

```
12      }
13  }
```

抽象类构造器不能用于直接创建抽象类对象，而是在创建子类对象时通过 `super(...)` 调用。

5. 注意事项

- `abstract class` 与 `final class` 不能同时使用：前者要求被继承，后者禁止被继承
- 抽象方法不能使用 `final`：抽象方法要求重写，`final` 禁止重写
- 抽象类可以提供普通方法，供所有子类直接继承和复用，且多态可用
- 抽象类可以定义构造器，用于初始化从父类继承的成员

注释

抽象类相当于一张不完整的蓝图：
父类规定“必须具备什么功能”，具体子类负责完成“这个功能应该怎样实现”。

1.4.4 接口（Interface）

接口用于定义一组**行为规范**：规定实现类“必须具备哪些功能”，但通常不规定具体如何实现。

1. 接口的定义

使用 interface 关键字定义接口：

```
public interface Flyable {
    void fly();
}
```

接口具有以下特点：

- **不能直接使用 new 创建对象**
- 没有构造器
- **主要用于声明行为规范**
- 类通过 implements 实现接口
- 接口本身可以继承一个或多个接口

```
Flyable f = new Flyable(); // 编译错误
```

2. 接口中的成员

接口中可以包含：

- 常量
- 抽象方法
- 默认方法
- 静态方法
- 私有方法

```
interface Example {
    int MAX_SIZE = 100;        // 常量

    void execute();            // 抽象方法

    default void show() {      // 默认方法
        System.out.println("show");
    }

    static void print() {      // 静态方法
        System.out.println("print");
    }
```

```
13
14      private void helper() {    // 私有方法，JDK 9+
15          System.out.println("helper");
16      }
17  }
```

3. 接口常量

接口中的变量默认都是：public static final

因此：

```
1  interface Config {
2      int MAX_SIZE = 100;
3  }
```

等价于：

```
1  interface Config {
2      public static final int MAX_SIZE = 100;
3  }
```

接口常量属于接口本身，通常通过接口名访问：

```
1  System.out.println(Config.MAX_SIZE);
```

接口中不能定义普通实例变量，并且常量必须在声明时完成初始化。

4. 接口抽象方法

接口中的普通无方法体方法默认都是：public abstract

因此：

```
1  interface Flyable {
2      void fly();
3  }
```

等价于：

```
1  interface Flyable {
2      public abstract void fly();
3  }
```

抽象方法只有声明，没有方法体，以分号结尾。

5. 实现接口

类使用 implements 实现接口：

```java
interface Flyable {
    void fly();
}

class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("鸟正在飞");
    }
}
```

实现接口时需要注意：

- 普通实现类必须实现全部抽象方法，如果没实现全部抽象方法，则必须声明为抽象类
- 实现方法必须声明为 public

```java
abstract class Animal implements Flyable {
    // 可以暂时不实现 fly()
}
```

接口方法默认是 public，因此实现时不能降低访问权限：

```java
class Bird implements Flyable {
    void fly() { } // 编译错误：访问权限不足
}
```

6. 多实现

Java 中的类只能继承一个父类，但可以同时实现多个接口

```java
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("鸭子正在飞");
    }
```

```
14
15      @Override
16      public void swim() {
17          System.out.println("鸭子正在游泳");
18      }
19  }
```

类也可以在继承一个父类的同时实现多个接口：

```
1  class Duck extends Animal
2          implements Flyable, Swimmable {
3  }
```

语法顺序必须是：`extends 父类 implements 接口 1, 接口 2`

7. 接口的多继承

接口使用 `extends` **继承其他接口**，并且可以同时**继承多个接口**：

```
1  interface Flyable {
2      void fly();
3  }
4
5  interface Swimmable {
6      void swim();
7  }
8
9  interface Amphibious extends Flyable, Swimmable {
10     void move();
11 }
```

实现子接口的类，需要实现该接口及其**所有父接口**中的抽象方法

```
1  class Duck implements Amphibious {
2      @Override
3      public void fly() { }
4
5      @Override
6      public void swim() { }
7
8      @Override
9      public void move() { }
10 }
```

8. 接口与多态

接口引用可以指向任意实现该接口的对象：

```
interface Payment {
    void pay(double amount);
}

class Alipay implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("支付宝支付：" + amount);
    }
}

class WeChatPay implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("微信支付：" + amount);
    }
}
```

使用接口类型接收不同实现类对象：

```
Payment payment = new Alipay();
payment.pay(100);

payment = new WeChatPay();
payment.pay(100);
```

调用方只依赖 Payment 接口，可以灵活切换不同实现。

9. 默认方法（JDK 8+）

默认方法使用 default 修饰，具有方法体：

```
interface Vehicle {
    default void start() {
        System.out.println("交通工具启动");
    }
}
```

默认方法的特点：

- 默认是 public；

- 实现类**可以直接继承，也可以选择重写**；
- **通过实现类对象调用**

```
class Car implements Vehicle {
}

Car car = new Car();
car.start();
```

默认方法主要用于在不破坏已有实现类的情况下，为接口增加新的通用功能。

10. 静态方法（JDK 8+）

接口静态方法属于接口本身：

```
interface MathTool {
    static int add(int a, int b) {
        return a + b;
    }
}
```

必须使用接口名调用：

```
int result = MathTool.add(2, 3);
```

接口静态方法不会被实现类继承，不能通过实现类对象调用。

11. 私有方法（JDK 9+）

接口可以使用私有方法复用内部逻辑：

```
interface Logger {
    default void info(String message) {
        print("INFO", message);
    }

    default void error(String message) {
        print("ERROR", message);
    }

    private void print(String level, String message) {
        System.out.println(level + ": " + message);
    }
}
```

私有方法：

- **只能在接口内部调用**，不能被实现类访问或重写
- 主要用于减少默认方法之间的重复代码

12. 默认方法冲突

如果一个类实现的两个接口包含相同的默认方法，实现类必须主动重写：

```
interface A {
    default void show() {
        System.out.println("A");
    }
}

interface B {
    default void show() {
        System.out.println("B");
    }
}

class C implements A, B {
    @Override
    public void show() {
        System.out.println("C");
    }
}
```

也可以指定调用某个接口的默认实现：

```
class C implements A, B {
    @Override
    public void show() {
        A.super.show();
    }
}
```

如果父类方法与接口默认方法冲突，优先使用父类方法（类优先于接口）。

13. 函数式接口

只包含一个抽象方法的接口称为函数式接口，可以使用 `@FunctionalInterface` 检查：

```
@FunctionalInterface
interface Calculator {
```

```
    int calculate(int a, int b);
}
```

函数式接口可以使用 Lambda 表达式实现：

```
Calculator add = (a, b) -> a + b;

System.out.println(add.calculate(2, 3));
```

默认方法、静态方法和私有方法不会计入抽象方法数量。

14. 接口与抽象类的区别

对比项	接口	抽象类
声明方式	`interface`	`abstract class`
对象创建	不能直接实例化	不能直接实例化
构造器	没有构造器	可以有构造器
成员变量	只能定义常量	可以定义实例变量和类变量
方法	抽象、默认、静态和私有方法	抽象方法和普通方法
继承关系	一个类可以实现多个接口	一个类只能继承一个父类
主要作用	**描述对象具备什么能力**	抽取一类对象的共同状态和行为

表 1.4: 接口与抽象类的比较

1.4.5 封装性（Encapsulation）

封装是指将数据和实现细节隐藏起来，仅向外部提供必要且可控的访问方式。

1. 封装的基本思想

封装可以分为广义封装和狭义封装：

- **方法封装**：隐藏功能的具体实现，只暴露方法签名；
- **类封装**：将属性和操作属性的方法组织在同一个类中；
- **包封装**：按功能组织类，并利用包级访问权限限制访问；
- **属性封装**：使用 `private` 隐藏属性，通过方法控制属性的读取和修改。

注释

封装不是单纯地将所有成员设置为 `private`，而是只暴露必要的功能，并隐藏不需要外部知道的实现。

2. 属性封装

属性封装通常分为两个步骤：

1) 使用 `private` 修饰成员变量

2) 提供公共方法读取或修改成员变量

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

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age >= 0 && age <= 150) {
            this.age = age;
        } else {
```

```
            System.out.println("年龄不合法");
        }
    }
}
```

外部代码不能直接访问私有属性：

```
Person person = new Person();

// person.age = -10; // 编译错误
person.setAge(20);
System.out.println(person.getAge());
```

封装的主要好处：

- **安全性**：防止外部代码随意写入非法数据；
- **可控性**：可以在方法中加入校验、转换和日志；
- **可维护性**：内部实现改变时，可以保持对外接口不变；
- **易用性**：调用者只需了解公开方法，不必了解内部细节。

3. 权限修饰符

Java 使用四种访问级别控制类成员的可见范围。

修饰符	本类	同包	不同包子类	其他包
private	✓	×	×	×
缺省	✓	✓	×	×
protected	✓	✓	✓	×
public	✓	✓	✓	✓

表 1.5: Java 权限修饰符的访问范围

- private

  只能在当前类中访问，常用于隐藏属性和内部方法。

```
class Account {
    private double balance;
}
```

- **缺省权限**

  不写访问修饰符时，只允许同一个包中的类访问

```
class PackageTool {
```

```
    void execute() {
    }
}
```

- `protected`

  同包中的类可以访问；不同包中的子类也可以通过继承关系访问。

```
class Animal {
    protected String name;
}
```

  不同包的子类访问 `protected` 成员时，**访问必须建立在继承关系上**，不能将其理解为“任何子类对象都可以随意访问”。

- `public`

  任何包中的代码都可以访问，通常用于对外提供的 API。

访问权限从小到大为：

`private` < 缺省 < `protected` < `public`

程序结构	可以使用的访问权限
顶层类或接口	`public` 或缺省权限
成员变量	四种权限均可使用
成员方法	四种权限均可使用
构造器	四种权限均可使用
成员内部类	四种权限均可使用
局部变量	不能使用

表 1.6: 权限修饰符的适用范围

4. 包级封装：`package` 与 `import`

1) `package`

`package` 用于声明当前类所属的包。

```
package com.example.model;

public class User {
}
```

包的主要作用：

- 按功能组织和管理类；
- 避免不同类之间发生名称冲突；
- 配合缺省权限和 protected 控制访问范围；
- 将大型项目划分为不同模块。

不同包中可以存在同名类：

```
com.example.user.User
com.example.admin.User
```

包名通常采用域名倒置的形式，并全部使用小写字母：

```
com.example.project.model
com.example.project.service
```

包名通常与源代码目录结构对应：

```
package com.example.model;
```

对应目录：

```
src/
   com/
       example/
           model/
               User.java
```

package 声明必须位于源文件的最前面（注释除外），**一个源文件最多只能有一个包声明**。

2) import

import 用于简化其他包中类型名称的书写。

不使用 import：

```
java.util.ArrayList<String> list = new java.util.ArrayList<>();
```

使用 import：

```
import java.util.ArrayList;
ArrayList<String> list = new ArrayList<>();
```

import 不会将类复制到当前包，也不改变类所在的包；只是允许代码使用类的简单名称。

3) **import 的使用方式**

- 单类型导入：

```
import java.util.ArrayList;
```

- 通配符导入：

```
import java.util.*;
```

`*` 只包含当前包中的类型，不会递归导入子包中的类型。

- 静态导入：

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

double area = PI * 4;
double value = sqrt(16);
```

静态导入用于直接使用类中的静态成员，但过度使用可能降低代码可读性。

4) **不需要导入的类型**

以下类型通常不需要显式导入：

- 当前类所在包中的其他类型；
- `java.lang` 包中的类型；
- 使用完整限定名称直接引用的类型。

例如：

```
String name = "Java";
System.out.println(name);
Integer number = 10;
```

`String`、`System` 和 `Integer` 都属于 `java.lang`，因此不需要手动导入。

5) **同名类型冲突**

如果两个包中存在同名类型，不能同时通过简单名称区分它们。

例如：

```
import java.util.Date;
// import java.sql.Date; // 名称冲突
```

此时至少有一个类型需要使用完整限定名称：

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date(0);
```

6) **源文件中的书写顺序**

Java 源文件通常按照以下顺序书写：

```
package com.example.test;
```

```
2
3 import java.util.ArrayList;
4 import static java.lang.Math.PI;
5
6 public class Main {
7
8 }
```

即：package ⟶ import ⟶ 类、接口或枚举的声明

1.4.6 继承性（Inheritance）

继承用于描述类之间的 **is-a** 关系。子类可以复用父类已有的属性和方法，并在此基础上增加或修改自己的功能。

1. 继承的基本语法

Java 使用 `extends` 建立类之间的继承关系：

```
class 父类 {
}

class 子类 extends 父类 {
}
```

例如：

```
class Person {
    String name;

    public void eat() {
        System.out.println(name + "正在吃饭");
    }
}

class Student extends Person {
    String school;

    public void study() {
        System.out.println(name + "正在学习");
    }
}
```

创建子类对象后，可以使用子类的成员，也可以使用从父类继承且当前代码有权访问的成员：

```
Student student = new Student();

student.name = "张三";
student.school = "清华大学";

student.eat();
student.study();
```

构造器不会被子类继承，但创建子类对象时必须先完成父类部分的初始化。

创建子类对象时，必须先初始化父类部分，然后再执行子类构造器。

```
class Person {
    String name;

    public Person(String name) {
        this.name = name;
        System.out.println("父类构造器");
    }
}

class Student extends Person {
    String school;

    public Student(String name, String school) {
        super(name);
        this.school = school;
        System.out.println("子类构造器");
    }
}
```

执行：

```
Student student = new Student("张三", "清华大学");
```

如果子类构造器没有显式写出父类构造器调用，编译器会尝试自动添加：

```
super();
```

因此，如果父类没有无参数构造器，子类必须显式调用父类已有的构造器：

```
class Parent {
    public Parent(String name) {
    }
}

class Child extends Parent {
    public Child() {
        super("默认名称");
    }
}
```

否则会出现编译错误。

2. 继承的主要作用：

- 提取多个类中重复的成员
- 提高代码复用性
- 建立统一的类型体系
- 为方法重写和多态提供基础

3. 继承的基本规则

- Java 类只支持**单继承**
- 一个类只能有**一个直接父类**，但 Java 支持**多层继承**
- 一个父类可以拥有多个直接子类
- 所有类最终都直接或间接继承 `java.lang.Object`

合法：

```
class Animal {
}

class Mammal extends Animal {
}

class Dog extends Mammal {
}
```

非法：

```
class Dog extends Animal, Mammal {
    // 编译错误：类不支持多继承
}
```

一个类虽然只能继承一个父类，但可以同时实现多个接口：

```
class Duck extends Animal implements Flyable, Swimmable {
}
```

4. 子类与父类权限管理

子类对象中包含父类部分和子类部分，但子类能否直接访问父类成员，取决于成员的访问权限。

- `public`：子类通常可以访问；
- `protected`：同包或继承条件下可以访问；
- 缺省权限：只有同包子类可以访问；
- `private`：子类不能直接访问。

```java
class Parent {
    public int publicValue;
    protected int protectedValue;
    int defaultValue;
    private int privateValue;
}

class Child extends Parent {
    public void test() {
        // 可以访问
        System.out.println(publicValue);
        System.out.println(protectedValue);

        // 同包时才可以访问
        System.out.println(defaultValue);

        // 编译错误：不能直接访问父类私有成员
        // System.out.println(privateValue);
    }
}
```

父类的 private 成员仍然属于子类对象中的父类部分，但子类代码不能直接访问，只能通过父类提供的公开或受保护方法间接操作。

5. 成员访问的就近原则

当局部变量、子类成员和父类成员同名时，Java 按照以下顺序查找：

局部变量 ⟶ 当前类成员 ⟶ 父类成员.

```java
class Parent {
    String name = "父类成员";

    public void show() {
        System.out.println("父类方法");
    }
}

class Child extends Parent {
    String name = "子类成员";

```

```
    public void printName() {
        String name = "局部变量";

        System.out.println(name);        // 局部变量
        System.out.println(this.name);   // 子类成员
        System.out.println(super.name); // 父类成员
    }

    @Override
    public void show() {
        System.out.println("子类方法");
    }

    public void printMethod() {
        show();        // 子类方法
        this.show();   // 子类方法
        super.show(); // 父类方法
    }
}
```

其中：

- this 表示当前对象；
- super 表示当前对象中的父类部分。

成员变量不存在多态重写，子类声明同名变量称为字段隐藏(子类定义了与父类同名的成员变量时，并不会替换父类变量，而是父类变量和子类变量同时存在)；方法则可以发生多态重写。

6. 方法重写（Override）

方法重写是指子类重新实现父类中已有的实例方法：

```
class Animal {
    public void shout() {
        System.out.println("动物发出声音");
    }
}

class Dog extends Animal {
    @Override
    public void shout() {
        System.out.println("汪汪汪");
    }
```

```
}
```

使用父类引用指向子类对象时：

```
Animal animal = new Dog();
animal.shout(); // 汪汪汪
```

编译阶段根据引用类型检查方法是否存在，运行阶段根据对象的实际类型选择重写后的方法。

重写方法应满足以下规则：

1) 方法名和参数列表必须相同；
2) 基本类型和 void 返回值必须相同；
3) 引用类型返回值**可以是父类方法返回类型的子类型**，称为协变返回类型；
4) 子类**方法的访问权限不能比父类方法更严格**；
5) 子类**方法不能声明比父类方法范围更大的受检异常**；
6) 建议使用 @Override，让编译器检查是否正确重写。

协变返回值：

```
class Parent {
    public Number getValue() {
        return 1;
    }
}

class Child extends Parent {
    @Override
    public Integer getValue() {
        return 2;
    }
}
```

扩大访问权限：

```
class Parent {
    protected void execute() {
    }
}

class Child extends Parent {
    @Override
    public void execute() {
```

```
9       }
10  }
```

以下成员不能被重写：

- `private` 方法：子类不可见；
- `final` 方法：明确禁止重写；
- `static` 方法：属于类，只能发生静态方法隐藏；
- 构造器：构造器不被继承，因此不能重写。

7. 重写与重载的区别

对比项	重写（Override）	重载（Overload）
发生位置	父类与子类之间	通常发生在同一个类中
方法名称	必须相同	必须相同
参数列表	必须相同	必须不同
返回值	相同或协变返回类型	不能仅靠返回值区分
访问权限	不能缩小	没有此要求
绑定时间	运行期动态绑定	编译期确定
主要作用	修改继承方法的实现	提供同一功能的不同参数版本

表 1.7: 方法重写与方法重载的比较

8. `this` 与 `super`

用途	`this`	`super`
访问成员变量	`this.name`	`super.name`
调用成员方法	`this.show()`	`super.show()`
调用构造器	`this(...)`	`super(...)`
表示对象	当前对象	当前对象中的父类部分

表 1.8: `this` 与 `super` 的比较

需要注意：

- `this(...)` 和 `super(...)` 都必须位于构造器的第一条语句；
- 同一个构造器中不能同时直接写 `this(...)` 和 `super(...)`；
- 构造器之间不能形成循环调用；

- 构造器调用链最终必须到达某个父类构造器。

```
class Student extends Person {
    String school;

    public Student(String name) {
        this(name, "默认学校");
    }

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

9. 继承的设计原则

继承应当表示明确的 **is-a** 关系：学生是人，狗是动物，经理是员工。

不合理的继承：汽车不是发动机，电脑不是键盘。

后者通常应使用组合关系：

```
class Engine {
}

class Car {
    private Engine engine;
}
```

注释

继承表示“子类是一种父类”，组合表示“一个对象拥有另一个对象”。

只有在真正存在类型替代关系时，才应优先考虑继承。

1.4.7 多态性（Polymorphism）

多态指**同一父类引用可以指向不同的子类对象**，调用同一个方法时，根据对象的实际类型执行。

1. 多态的基本形式

多态最常见的形式是：

```
父类类型 引用变量 = new 子类();
```

例如：

```
class Person {
    public void introduce() {
        System.out.println("我是一个人");
    }
}

class Student extends Person {
    @Override
    public void introduce() {
        System.out.println("我是一名学生");
    }

    public void study() {
        System.out.println("学生正在学习");
    }
}
```

使用父类引用指向子类对象：

```
Person person = new Student();
person.introduce(); // 我是一名学生
// person.study();  // 编译错误
```

其中：

- person 的编译时类型是 Person
- person 的运行时类型是 Student
- 编译器只允许访问 Person 中声明的成员；运行时调用的是 Student 重写的方法

注释

父类引用相当于一个通用遥控器，可以控制不同的子类对象，但只能使用父类型中已经定义的“按钮”。

2. 编译时类型与运行时类型

对于：

```
Person person = new Student();
```

可以记为：

编译看左边，运行看右边。

- 编译阶段根据左边的 Person 判断成员是否可以访问；
- 运行阶段根据右边实际创建的 Student 对象决定执行哪个重写方法。

因此：

```
person.introduce(); // Person 中存在，允许调用
// person.study();  // Person 中不存在，编译错误
```

虽然实际对象中存在 study()，但引用类型为 Person，编译器无法用该引用访问子类特有方法。

3. 多态调用的条件

要出现典型的运行时多态效果，通常需要：

1) 存在继承关系或接口实现关系；
2) 父类或接口引用指向子类或实现类对象；
3) 子类重写父类实例方法，从而在运行时表现出不同实现。

严格来说，方法重写不是完成向上转型的必要条件；但如果子类没有重写方法，调用时只会执行继承来的父类实现，无法体现“同一方法，不同实现”的效果。

4. 成员变量与方法的不同表现

实例方法具有运行时多态，成员变量不具有运行时多态。

```
class Person {
    int age = 50;

    public void showAge() {
        System.out.println(age);
    }
}

class Student extends Person {
    int age = 20;

    @Override
    public void showAge() {
```

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

```
Person person = new Student();

System.out.println(person.age); // 50（父类的age）
person.showAge();               // 20
```

原因是：

- 成员变量根据引用的编译时类型选择；
- 重写后的实例方法根据对象的实际类型选择。

此外，**以下成员也不参与运行时多态**：

- 静态 `static` 方法
- 私有 `private` 方法
- 构造器；
- 静态成员变量和实例成员变量

5. 向上转型

将子类引用转换为父类引用称为向上转型，通常由 Java 自动完成：

```
Student student = new Student();
Person person = student;
```

也可以直接写成：

```
Person person = new Student();
```

向上转型后：

- 可以访问父类型中声明的成员；
- 可以动态调用子类重写的方法；
- 不能通过父类引用直接访问子类特有成员。

向上转型不会删除对象中的子类成员，只是父类引用暂时无法直接访问它们。

6. 向下转型

将父类引用强制转换为子类引用称为向下转型：

```
Person person = new Student();
Student student = (Student) person;
student.study();
```

向下转型只能改变引用的类型，不能改变对象本身的真实类型。

下面的转换会在运行时失败：

```
Person person = new Person();

Student student = (Student) person; // ClassCastException
```

因为 person 实际指向的是 Person 对象，而不是 Student 对象。

在向下转型前，可以**使用 instanceof 判断对象的实际类型**：

```
if (person instanceof Student) {
    Student student = (Student) person;
    student.study();
}
```

语法：

```
对象 instanceof 类型
```

返回值为 boolean：

```
Person person = new Student();

System.out.println(person instanceof Student); // true
System.out.println(person instanceof Person); // true
System.out.println(person instanceof Object); // true
```

如果引用为 null：

```
Person person = null;

System.out.println(person instanceof Student); // false
```

7. 接口多态

接口引用也可以指向不同的实现类对象：

```
interface Payment {
    void pay(double amount);
}

class Alipay implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("支付宝支付");
```

```
    }
}

class WeChatPay implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("微信支付");
    }
}
```

```
Payment payment = new Alipay();
payment.pay(100);

payment = new WeChatPay();
payment.pay(100);
```

调用方只依赖 Payment 接口，可以方便地替换不同支付实现。

8. 常见写法

集合框架中也大量使用多态：

```
List<String> list = new ArrayList<>();
```

其中：

- List 是接口类型；
- ArrayList 是具体实现类；
- 调用方依赖 List，可以替换为其他实现。

例如：

```
List<String> list = new LinkedList<>();
```

1.4.8 常见设计模式（Design Patterns）

设计模式是针对常见软件设计问题总结出的 **可复用设计方案**。它不是固定代码，也不是新的语法，而是组织类与对象协作关系的一种思路。
本节介绍四种常见模式：

- 单例模式（Singleton）：控制对象数量；
- 模板方法模式（Template Method）：固定执行流程；
- 工厂模式（Factory）：封装对象创建；
- 代理模式（Proxy）：控制对象访问并增强功能。

单例模式（Singleton）

单例模式保证一个类在程序中只有一个实例，并提供统一的访问入口。
基本实现要点：

1. 使用 `private` 修饰构造器；
2. 在类内部保存唯一实例；
3. 提供静态方法或静态字段访问实例。

饿汉式 类加载时立即创建对象：

```
class Singleton {
    public static final Singleton INSTANCE = new Singleton();

    private Singleton() {}
}
```

使用方式：

```
ConfigManager c1 = ConfigManager.getInstance();
ConfigManager c2 = ConfigManager.getInstance();

System.out.println(c1 == c2); // true
```

特点：

- 实现简单；
- 天然线程安全；
- 类加载时就创建实例，不支持延迟初始化。

懒汉式 第一次使用时才创建对象：

```
public static synchronized Singleton getInstance() {
```

```
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
}
```

这种写法线程安全，但每次调用 getInstance() 都需要获得锁，可能产生额外开销。

枚举单例 最佳实践

```
enum Singleton {
    INSTANCE;

    // 可以添加方法和属性
    public void doSomething() {
        System.out.println("枚举单例");
    }
}
```

调用：

```
Singleton.INSTANCE.doSomething(); // 直接调用

Singleton s1 = Singleton.INSTANCE;
Singleton s2 = Singleton.INSTANCE;

System.out.println(s1 == s2); // true
```

实现同目的，普通单列需要：

```
class Singleton {
    private static final Singleton INSTANCE =
        new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

枚举单例实现简单，并由 JVM 保证实例唯一性。
常见应用：

- 全局配置管理器；
- 日志管理器；
- 缓存管理器；
- 需要共享的资源协调对象。

单例模式不表示整个分布式系统只有一个对象，只保证一个 JVM 或类加载器范围内只有一个实例。

模板方法模式（Template Method）

模板方法模式在父类中定义算法的整体流程，将部分具体步骤交给子类实现。

```
// 父类中定义算法的整体流程
abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    public final void play() {
        initialize();      // 初始化
        startPlay();       // 开始游戏
        endPlay();         // 结束游戏
    }
}

// 具体步骤交给子类实现
class Cricket extends Game {
    void initialize() { System.out.println("Cricket 游戏初始化"); }
    void startPlay() { System.out.println("Cricket 游戏开始"); }
    void endPlay() { System.out.println("Cricket 游戏结束"); }
}
```

适用场景：

- 多个业务具有相同流程，但部分步骤不同；
- 数据处理、文件导入和报表生成流程；
- 框架规定整体流程，用户实现扩展步骤。

注释

模板方法模式是：父类决定“先做什么、再做什么”，子类决定“每一步具体怎么做”。

工厂模式（Factory）

工厂模式将对象的创建过程封装起来，使调用方依赖抽象类型，而不是直接依赖具体实现类。

简单工厂 一个工厂根据参数创建不同对象：

```
// 产品接口：定义所有产品的共同行为
interface Phone {
    void make();
}

// 具体产品1：华为手机
class HuaweiPhone implements Phone {
    @Override
    public void make() {
        System.out.println("生产华为手机");
    }
}

// 具体产品2：苹果手机
class IPhone implements Phone {
    @Override
    public void make() {
        System.out.println("生产苹果手机");
    }
}

// 简单工厂类：负责创建产品
class PhoneFactory {
    // 根据参数创建不同产品
    public static Phone createPhone(String type) {
        if ("huawei".equals(type)) {
            return new HuaweiPhone();
        } else if ("iphone".equals(type)) {
            return new IPhone();
        } else {
            throw new IllegalArgumentException("未知手机类型");
```

```
        }
    }
}

// 测试
public class SimpleFactoryDemo {
    public static void main(String[] args) {
        Phone huawei = PhoneFactory.createPhone("huawei");
        huawei.make(); // 输出：生产华为手机

        Phone iphone = PhoneFactory.createPhone("iphone");
        iphone.make(); // 输出：生产苹果手机
    }
}
```

优点是结构简单，客户端不需要直接创建具体对象。缺点是新增产品时通常需要修改工厂中的判断逻辑，扩展性有限。

简单工厂通常被视为一种常用设计方式，但不属于 GoF 提出的 23 种经典设计模式。

工厂方法 为每一种产品提供对应的工厂：

```
// 产品接口
interface Phone {
    void make();
}

// 具体产品：华为手机
class HuaweiPhone implements Phone {
    @Override
    public void make() {
        System.out.println("生产华为手机");
    }
}

// 具体产品：苹果手机
class IPhone implements Phone {
    @Override
    public void make() {
        System.out.println("生产苹果手机");
    }
```

```
}

// 工厂接口：定义创建产品的方法
interface PhoneFactory {
    Phone createPhone();
}

// 具体工厂1：华为工厂（生产华为手机）
class HuaweiFactory implements PhoneFactory {
    @Override
    public Phone createPhone() {
        return new HuaweiPhone();
    }
}

// 具体工厂2：苹果工厂（生产苹果手机）
class IPhoneFactory implements PhoneFactory {
    @Override
    public Phone createPhone() {
        return new IPhone();
    }
}

// 测试
public class FactoryMethodDemo {
    public static void main(String[] args) {
        PhoneFactory huaweiFactory = new HuaweiFactory();
        Phone huawei = huaweiFactory.createPhone();
        huawei.make(); // 输出：生产华为手机

        PhoneFactory iphoneFactory = new IPhoneFactory();
        Phone iphone = iphoneFactory.createPhone();
        iphone.make(); // 输出：生产苹果手机
    }
}
```

优点：符合“开闭原则”（新增产品只需加工厂类，无需修改原有代码）。
缺点：类数量增多（每个产品对应一个工厂），复杂度提高。

抽象工厂 创建一系列相关或相互依赖的产品族（而非单一产品），通过抽象工厂接口统一管理。

```
// 产品接口1：手机
interface Phone {
    void makePhone();
}

// 产品接口2：充电器（与手机关联）
interface Charger {
    void makeCharger();
}

// 华为产品族
class HuaweiPhone implements Phone {
    @Override
    public void makePhone() {
        System.out.println("生产华为手机");
    }
}
class HuaweiCharger implements Charger {
    @Override
    public void makeCharger() {
        System.out.println("生产华为充电器");
    }
}

// 苹果产品族
class IPhone implements Phone {
    @Override
    public void makePhone() {
        System.out.println("生产苹果手机");
    }
}
class IPhoneCharger implements Charger {
    @Override
    public void makeCharger() {
        System.out.println("生产苹果充电器");
    }
}

```

```
// 抽象工厂接口：定义创建产品族的方法
interface AbstractFactory {
    Phone createPhone();
    Charger createCharger();
}

// 华为工厂（生产华为产品族）
class HuaweiFactory implements AbstractFactory {
    @Override
    public Phone createPhone() {
        return new HuaweiPhone();
    }
    @Override
    public Charger createCharger() {
        return new HuaweiCharger();
    }
}

// 苹果工厂（生产苹果产品族）
class IPhoneFactory implements AbstractFactory {
    @Override
    public Phone createPhone() {
        return new IPhone();
    }
    @Override
    public Charger createCharger() {
        return new IPhoneCharger();
    }
}

// 测试
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        // 使用华为产品族
        AbstractFactory huaweiFactory = new HuaweiFactory();
        huaweiFactory.createPhone().makePhone();      // 输出：生产华为手机
        huaweiFactory.createCharger().makeCharger();  // 输出：生产华为充电器

        // 使用苹果产品族
```

```
        AbstractFactory iphoneFactory = new IPhoneFactory();
        iphoneFactory.createPhone().makePhone();       // 输出：生产苹果手机
        iphoneFactory.createCharger().makeCharger();  // 输出：生产苹果充电器
    }
}
```

三种工厂方式的区别：

模式	核心结构	适用场景
简单工厂	一个工厂创建所有产品	产品种类较少且变化不频繁
工厂方法	每种产品对应一个工厂	产品种类较多，需要持续扩展
抽象工厂	一个工厂创建一组关联产品	需要创建和切换完整产品族

表 1.9: 三种工厂方式的比较

工厂模式的核心价值：

- 分离对象的创建与使用；
- 隐藏复杂的创建过程；
- 让客户端依赖接口或抽象类；
- 便于统一管理对象初始化逻辑。

代理模式（Proxy）

代理模式使用代理对象控制对目标对象的访问，并在不修改目标类的情况下增加额外功能。
代理模式通常包含三个角色：

- 抽象主题：代理类和目标类共同实现的接口；
- 目标对象：真正执行业务逻辑的对象；
- 代理对象：持有目标对象并负责附加操作。

静态代理 代理类与目标类一一对应，编译期确定。

```
// 抽象主题：定义核心业务接口
interface Image {
    void display();
}

// 目标对象：实际加载图片的类（耗时操作）
class RealImage implements Image {
    private String filename;

```

```
10      public RealImage(String filename) {
11          this.filename = filename;
12          loadFromDisk(); // 模拟从磁盘加载图片（耗时）
13      }
14
15      private void loadFromDisk() {
16          System.out.println("加载图片： " + filename);
17      }
18
19      @Override
20      public void display() {
21          System.out.println("显示图片： " + filename);
22      }
23  }
24
25  // 代理对象：控制对RealImage的访问，增加额外功能
26  class ProxyImage implements Image {
27      private RealImage realImage;
28      private String filename;
29
30      public ProxyImage(String filename) {
31          this.filename = filename;
32      }
33
34      @Override
35      public void display() {
36          // 延迟加载：仅在需要时才创建目标对象
37          if (realImage == null) {
38              realImage = new RealImage(filename);
39          }
40          // 附加日志功能
41          System.out.println("日志：准备显示图片");
42          // 调用目标对象的方法
43          realImage.display();
44          System.out.println("日志：图片显示完成");
45      }
46  }
47
48  // 测试
```

```
public class StaticProxyDemo {
    public static void main(String[] args) {
        Image image = new ProxyImage("风景.jpg");

        // 第一次调用：会加载图片
        image.display();
        // 第二次调用：直接显示（无需重复加载）
        image.display();
    }
}
```

优点：简单直观，能明确控制访问逻辑。
缺点：新增目标类时需同步创建代理类，代码冗余（违反“开闭原则”）。

JDK 动态代理 JDK 动态代理在运行时生成代理对象，核心是 InvocationHandler：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

// 抽象主题（主题接口）
interface UserService {
    void login(String username);
    void logout();
}

// 目标对象（被代理类）
class UserServiceImpl implements UserService {
    @Override
    public void login(String username) {
        System.out.println(username + "登录成功");
    }

    @Override
    public void logout() {
        System.out.println("退出登录");
    }
}


```

```
25
26 import java.lang.reflect.InvocationHandler;
27 import java.lang.reflect.Method;
28
29 /**
30  * 实现 InvocationHandler 接口：这是动态代理的核心接口，要求必须实现 invoke 方法。
31  * target 成员变量：存储目标对象（如 UserServiceImpl
      实例），代理需要通过它调用真正的业务方法。
32  * invoke 方法：代理对象的所有方法调用都会进入这个方法（相当于 “中转站”）：
33  * proxy：动态生成的代理对象本身（一般不用）。
34  * method：当前调用的方法（如 login 或 logout）。
35  * args：方法的参数（如 login 方法的 username 参数）。
36  * 执行流程：前置操作 → 调用目标方法 → 后置操作，这就是 “增强” 目标方法的过程
37  */
38 class LogHandler implements InvocationHandler {
39     private Object target;  // 保存目标对象的引用（要代理的对象）
40
41     // 构造方法：传入目标对象
42     public LogHandler(Object target) {
43         this.target = target;
44     }
45
46     // 关键方法：所有代理对象的方法调用都会被转发到这里
47     @Override
48     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
49         // 1. 前置增强：目标方法执行前的操作（如打印日志）
50         System.out.println("日志：" + method.getName() + "方法开始执行");
51
52         // 2. 调用目标对象的实际方法（执行真正的业务逻辑）
53         Object result = method.invoke(target, args);  // 反射调用目标方法
54
55         // 3. 后置增强：目标方法执行后的操作（如打印日志）
56         System.out.println("日志：" + method.getName() + "方法执行结束");
57
58         return result;  // 返回目标方法的执行结果
59     }
60 }
61
62 /**
```

```
 * 步骤 1-2：创建目标对象和处理器，处理器关联目标对象。
 * 步骤 3：Proxy.newProxyInstance() 是生成动态代理对象的关键方法，JVM
   会在运行时动态创建一个实现了 UserService 接口的代理类，并返回其实例。
 * 步骤 4：调用代理对象的 login 和 logout 方法时，不会直接执行目标对象的方法，而是先进入
   LogHandler 的 invoke 方法（这就是代理的 “拦截” 作用）。
 */
public class DynamicProxyDemo {
    public static void main(String[] args) {
        // 1. 创建目标对象（真正做事的对象）
        UserService userService = new UserServiceImpl();

        // 2. 创建代理处理器，传入目标对象（告诉处理器要代理谁）
        LogHandler handler = new LogHandler(userService);

        // 3. 动态生成代理对象（核心！由JVM在运行时创建）
        UserService proxy = (UserService) Proxy.newProxyInstance(
            UserService.class.getClassLoader(),  // 类加载器（用接口的类加载器）
            new Class[]{UserService.class},      // 目标接口（代理对象要实现的接口）
            handler                              // 代理处理器（方法调用的中转站）
        );

        // 4. 调用代理对象的方法（看似调用UserService方法，实际会被代理拦截）
        proxy.login("张三");  // 调用代理的login方法
        proxy.logout();      // 调用代理的logout方法
    }
}
```

其中：

1. 调用 proxy.login(” 张三”) → 被代理拦截，进入 LogHandler.invoke() 方法
2. invoke 方法中：
   - 先打印 “日志：login 方法开始执行”（前置增强）
   - 通过 method.invoke(target, args) 调用目标对象 UserServiceImpl 的 login 方法 → 输出 “张三登录成功”
   - 再打印 “日志：login 方法执行结束”（后置增强）
3. proxy.logout() 的执行流程同上，最终实现了 “无侵入式” 地为目标方法添加日志功能

JDK 动态代理要求目标对象实现接口。如果目标类没有接口，框架通常可以使用基于继承的代理技术，例如 CGLIB。

常见应用：AOP；日志记录；权限校验；性能监控。

四种模式的比较

模式	核心目的	典型场景
单例模式	限制类的实例数量	配置、日志、共享管理器
模板方法	固定流程，允许部分步骤变化	统一业务流程、框架扩展点
工厂模式	分离对象的创建与使用	多种实现类、复杂创建逻辑
代理模式	控制访问并增强目标方法	日志、权限、事务、AOP

表 1.10: 常见设计模式的比较

1.5 Java 常用类

错题集

1.5.1 Object 类

java.lang.Object 是 Java 类体系的根类。如果一个类没有显式继承其他类，编译器会默认让它继承 Object。

```
class Person {
}
```

等价于：

```
class Person extends Object {
}
```

数组也是对象，因此数组类型同样可以赋值给 Object 引用：

```
Object obj1 = new Person();
Object obj2 = new int[]{1, 2, 3};
Object obj3 = "Java";
```

1. 常用方法

方法	作用
equals(Object obj)	判断当前对象与另一个对象是否逻辑相等
hashCode()	返回对象的哈希码，主要供哈希表使用
toString()	返回对象的字符串描述
getClass()	返回对象运行时所属的 Class 对象
clone()	创建对象副本，使用时存在较多限制
wait()、notify()、notifyAll()	用于线程之间的等待与通知
finalize()	已废弃，不应依赖它进行资源释放

表 1.11: Object 类中的常见方法

2. equals() 方法

自定义类如果需要按照属性判断对象是否相等，应重写 equals()：

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

    // 重写equals方法，比较姓名和年龄
    @Override
```

```
    public boolean equals(Object obj) {
        // 1. 自反性：自己和自己肯定相等
        if (this == obj) return true;
        // 2. 空指针判断：null和任何对象都不相等
        if (obj == null) return false;
        // 3. 类型判断：不同类型对象不相等
        if (getClass() != obj.getClass()) return false;
        // 4. 内容比较：强制转换后对比属性
        Person p = (Person) obj;
        return name.equals(p.name) && age == p.age;
    }
}
```

正确的 equals() 应满足以下约定：

- **自反性**：x.equals(x) 必须为 true
- **对称性**：如果 x.equals(y) 为 true，则 y.equals(x) 也必须为 true
- **传递性**：如果 x 等于 y，且 y 等于 z，则 x 必须等于 z
- **一致性**：对象状态不变时，多次比较结果应一致
- 非空对象与 null 比较必须返回 false

3. hashCode() 方法

hashCode() 返回一个 int 类型的哈希码，主要用于 HashMap、HashSet 等哈希数据结构。

通常使用与 equals() 相同的属性计算哈希码：

```
@Override
public int hashCode() {
    return Objects.hash(name, age);
}
```

equals() 与 hashCode() 必须满足：

- 如果两个对象通过 equals() 判断相等，它们的哈希码必须相同
- 哈希码相同的两个对象不一定相等
- 哈希码不同的两个对象一定不相等
- 不同对象可能产生相同哈希码，这种情况称为**哈希冲突**。

重写 equals() 时，通常必须同时重写 hashCode()。否则，在哈希集合中可能出现逻辑错误：

```
Person p1 = new Person("张三", 20);
Person p2 = new Person("张三", 20);

```

```
HashSet<Person> set = new HashSet<>();
set.add(p1);

System.out.println(set.contains(p2));
```

如果只重写 equals() 而没有正确重写 hashCode()，contains(p2) 可能返回 false。
哈希表通常按照以下步骤查找对象：

1) 根据 hashCode() 定位可能的桶；
2) 在桶中使用 equals() 精确比较。

因此：hashCode() 负责快速定位，equals() 负责最终确认。参与哈希码计算的属性不应在对象加入 HashSet 或作为 HashMap 键后随意修改，否则可能无法再次找到该对象。

4. toString() 方法

toString() 用于返回对象的字符串描述。
默认结果通常类似：

```
Person@4e25154f
```

其形式近似为：

类名 + @ + 哈希码的十六进制表示.

为了方便调试和输出，通常应重写：

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

输出对象时，会自动调用 toString()：

```
Person person = new Person("张三", 20);

System.out.println(person);
System.out.println("用户信息：" + person);
```

近似等价于：

```
System.out.println(person.toString());
```

toString() 应返回简洁、有意义的对象信息，但不应输出密码、令牌等敏感数据。

5. getClass() 方法

getClass() 返回对象运行时所属的类：

```
Object obj = new Person("张三", 20);

System.out.println(obj.getClass());                 // class Person
System.out.println(obj.getClass().getName());       // Person
System.out.println(obj.getClass().getSimpleName()); // Person
```

即使引用类型为 Object，getClass() 得到的仍然是对象的实际运行类型。

```
Object obj = "Java";

System.out.println(obj.getClass().getSimpleName()); // String
```

getClass() 是 final 方法，子类不能重写。

6. Object 与多态

由于所有对象都属于 Object 类型，因此 Object **可以接收任意对象**：

```
public static void printObject(Object obj) {
    System.out.println(obj);
}

printObject("Java");
printObject(100);
printObject(new Person("张三", 20));
```

其中：

printObject(100) 会先将基本类型 int 自动装箱为 Integer 对象。

但是，通过 Object 引用只能直接访问 Object 中声明的方法：

```
Object obj = new Person("张三", 20);
// obj.getName(); // 编译错误
```

需要先判断并向下转型：

```
if (obj instanceof Person) {
    Person person = (Person) obj;
    System.out.println(person);
}
```

7. Object 作为容器元素类型

```
ArrayList<Object> list = new ArrayList<>();

list.add("Java");
```

```
list.add(100);
list.add(new Person("张三", 20));
```

这种容器可以存放不同类型的对象，但读取时通常需要类型判断和向下转型：

```
Object element = list.get(0);

if (element instanceof String) {
    String text = (String) element;
}
```

实际开发中，如果集合元素类型明确，应优先使用具体泛型：

```
ArrayList<Person> people = new ArrayList<>();
```

这样可以在编译阶段检查类型，减少强制转换和类型错误。

8. 其他方法 *

- `clone()`
  用于复制对象，但 `Object.clone()` 是 `protected` 方法，并且通常要求类实现 `Cloneable`。
  默认克隆通常属于浅拷贝：对象中的引用类型成员仍可能指向同一个对象。
  实际开发中，通常更推荐使用复制构造器、工厂方法或专门的复制逻辑。
- `wait()`、`notify()`、`notifyAll()`
  用于线程之间的协作，必须在持有该对象监视器的同步代码中调用：

  ```
  synchronized (lock) {
      lock.wait();
  }
  ```

  这些方法属于并发编程知识，通常配合 `synchronized` 使用。
- `finalize()`
  该方法已经废弃，其执行时间不可预测，不能用于关闭文件、数据库连接等资源。
  资源应使用 `try-with-resources` 或显式关闭方法进行管理。

1.5.2 Wrapper

Java 的基本数据类型不是对象，不能直接调用方法，也不能直接作为泛型类型使用。为此，Java 为每一种基本数据类型提供了对应的 **包装类（Wrapper Class）**，使基本类型的数据可以以对象的形式参与面向对象编程。

基本数据类型与包装类

基本数据类型	包装类	继承自
byte	Byte	Number
short	Short	Number
int	Integer	Number
long	Long	Number
float	Float	Number
double	Double	Number
char	Character	无
boolean	Boolean	无

表 1.12: Java 基本数据类型与包装类

包装类的主要作用包括：

- 使基本类型可以作为对象使用；
- 使基本类型可以存储在集合或泛型结构中；
- 提供字符串转换、大小比较、进制转换等工具方法；
- 提供取值范围等常量，如 `Integer.MAX_VALUE` 和 `Integer.MIN_VALUE`。

例如，泛型不能直接使用基本数据类型：

```
List<int> numbers;       // 编译错误
List<Integer> numbers;   // 正确
```

装箱与拆箱

装箱（Boxing） 装箱是指将基本数据类型转换为对应的包装类对象。

```
int num = 10;

Integer numObj1 = Integer.valueOf(num); // 显式装箱
Integer numObj2 = num; // 自动装箱
```

现代 Java 中不建议使用包装类构造器：

```
Integer numObj = new Integer(10); // 不推荐
```

拆箱（Unboxing） 拆箱是指将包装类对象转换为对应的基本数据类型。

```
Integer numObj = 10;

int num1 = numObj.intValue(); // 显式拆箱
int num2 = numObj; // 自动拆箱
```

不同数值包装类提供了相应的拆箱方法：

- byteValue()
- shortValue()
- intValue()
- longValue()
- floatValue()
- doubleValue()

自动装箱与自动拆箱 从 JDK 1.5 开始，编译器可以自动完成基本类型和包装类之间的转换。

```
Integer x = 5;        // 自动装箱
int y = x;            // 自动拆箱

Integer a = 5;
Integer b = 3;
Integer c = a + b;
```

表达式

```
Integer c = a + b;
```

大致经历了以下过程：

1. 将 a 和 b 自动拆箱为 int；
2. 执行整数加法；
3. 将结果自动装箱为 Integer。

可以近似理解为：

```
Integer c = Integer.valueOf(a.intValue() + b.intValue());
```

字符串与基本数据类型之间的转换

字符串转换为基本数据类型　通常使用包装类的 parseXxx() 静态方法：

```
int i = Integer.parseInt("123");
long l = Long.parseLong("1000");
double d = Double.parseDouble("3.14");
boolean flag = Boolean.parseBoolean("true");
```

其中：

- parseInt() 返回 int；
- parseLong() 返回 long；
- parseDouble() 返回 double；
- parseBoolean() 返回 boolean。

若字符串不能转换为合法数字，通常会抛出 NumberFormatException：

```
int num = Integer.parseInt("abc"); // NumberFormatException
```

因此，可以使用异常处理：

```
try {
    int num = Integer.parseInt("123");
    System.out.println(num);
} catch (NumberFormatException e) {
    System.out.println("字符串不是合法整数");
}
```

字符串转换为包装类对象　使用包装类的 valueOf() 方法：

```
Integer i = Integer.valueOf("123");
Double d = Double.valueOf("3.14");
Boolean flag = Boolean.valueOf("true");
```

需要区分：

```
int a = Integer.parseInt("123");       // 返回 int
Integer b = Integer.valueOf("123");    // 返回 Integer
```

基本数据类型转换为字符串　推荐使用 String.valueOf()：

```
String s1 = String.valueOf(123);
String s2 = String.valueOf(3.14);
```

```
String s3 = String.valueOf(true);
```

也可以使用包装类的 `toString()` 方法：

```
String s1 = Integer.toString(123);
String s2 = Double.toString(3.14);
```

字符串拼接也能实现转换：

```
String s = 123 + "";
```

但为了表达清楚，通常更推荐：

```
String s = String.valueOf(123);
```

Integer 缓存机制

Java 会缓存一定范围内的 Integer 对象。通常情况下，默认缓存范围为：

$$-128 \le x \le 127$$

例如：

```
Integer a = 127;
Integer b = 127;

System.out.println(a == b); // true
```

因为 127 位于缓存范围内，a 和 b 可能引用同一个缓存对象。

```
Integer c = 128;
Integer d = 128;

System.out.println(c == d);      // 通常为 false
System.out.println(c.equals(d)); // true
```

不能依赖缓存机制判断数值是否相等。无论数值大小如何，都应使用：

```
c.equals(d)
```

包装类与 null

包装类是引用类型，因此可以保存 null：

```
Integer num = null;
```

而基本数据类型不能保存 null：

```
int num = null; // 编译错误
```

当值为 null 的包装类发生自动拆箱时，会抛出 NullPointerException：

```
Integer numObj = null;
int num = numObj; // NullPointerException
```

其原因是自动拆箱近似等价于：

```
int num = numObj.intValue();
```

但此时 numObj 为 null，无法调用 intValue()。
因此，在拆箱前应进行空值判断：

```
Integer numObj = null;

if (numObj != null) {
    int num = numObj;
}
```

也可以设置默认值：

```
Integer numObj = null;
int num = numObj == null ? 0 : numObj;
```

包装类的常用工具方法

比较大小 （没啥用）

```
int result = Integer.compare(10, 20);
```

返回值规则为：

- 小于零：第一个数较小；
- 等于零：两个数相等；
- 大于零：第一个数较大。

最大值与最小值 （有点用）

```
System.out.println(Integer.MAX_VALUE);
System.out.println(Integer.MIN_VALUE);

System.out.println(Double.MAX_VALUE);
System.out.println(Double.MIN_VALUE);
```

需要注意，Double.MIN_VALUE 表示最小的正数，并不是绝对值最大的负数。

进制转换 （有用）

```
System.out.println(Integer.toBinaryString(10)); // 1010
System.out.println(Integer.toOctalString(10));  // 12
System.out.println(Integer.toHexString(10));    // a
```

也可以按照指定进制解析字符串：

```
int binary = Integer.parseInt("1010", 2);
int octal = Integer.parseInt("12", 8);
int hex = Integer.parseInt("ff", 16);

System.out.println(binary); // 10
System.out.println(octal);  // 10
System.out.println(hex);    // 255
```

包装类与集合

Java 集合和泛型只能存储引用类型，因此不能直接写：

```
List<int> scores; // 编译错误
```

应使用包装类：

```
List<Integer> scores = new ArrayList<>();

scores.add(90); // 自动装箱
scores.add(85);
scores.add(95);

for (int score : scores) {
    // 自动拆箱
    System.out.println(score);
}
```

在以上代码中：

- `scores.add(90)` 将 `int` 自动装箱为 `Integer`；
- 增强型 `for` 循环将 `Integer` 自动拆箱为 `int`。

1.5.3 字符串相关的类

Java 中与字符串处理有关的核心类包括：

- `String`：不可变字符串；
- `StringBuilder`：可变字符串，通常用于单线程环境；
- `StringBuffer`：可变字符串，其许多方法带有同步机制；
- `Character`：基本数据类型 `char` 的包装类。

String 类的基本概念

`String` 是 Java 中最常用的类之一，用于表示一个字符序列。
虽然 Java 为字符串提供了特殊的字面量写法，但 `String` 本质上仍然是一个类（属于引用数据类型），而不是基本数据类型。
字符串使用双引号，字符使用单引号：

```
char letter = 'A';   // char 类型（基本数据类型）
String text = "A";   // String 类型（即使字符串中只有一个字符，它的类型仍然是 String）
String name = "Alice";
```

String 的不可变性

`String` 是一个不可变类，即字符串对象一旦创建，其内容就不能再被修改。
字符数组存储：内部用 `private final char value[]` 存储字符，不可修改。

```
String s1 = "abc";
String s2 = s1 + "d"; // 实际上s1没有变，而是指向新生成的"abcd"对象
```

类似地：

```
String text = "hello";
text.toUpperCase();
System.out.println(text); // hello
```

`toUpperCase()` 返回了一个新字符串（的引用），但变量 `text` 仍然指向原对象。
若希望保存转换结果，必须重新赋值：

```
text = text.toUpperCase();
System.out.println(text); // HELLO
```

因此需要区分：字符串对象本身是 `final` 类不可变，但字符串引用的变量可以改为指向其他对象。

String 为什么设计为不可变 字符串不可变具有以下好处：

- 便于字符串常量池中的对象被安全共享

- 便于在多个线程之间共享字符串
- 字符串的哈希值可以安全缓存
- 适合作为 HashMap 的键
- 避免文件路径、类名、网络地址等重要文本被意外修改

String 对象的创建

使用字符串字面量 推荐使用字符串字面量创建字符串：

```
String str1 = "hello";
String str2 = "hello";
```

对于内容相同的字符串字面量，Java 通常会复用字符串常量池中的同一个对象。

```
System.out.println(str1 == str2); // true
```

使用构造器 也可以使用构造器创建字符串：

```
String str3 = new String("hello");
String str4 = new String("hello");
```

每次显式使用 new，通常都会在堆中创建新的 String 对象：

```
System.out.println(str3 == str4);      // false
System.out.println(str3.equals(str4)); // true
```

字符串常量池

```
String s1 = "Java";
String s2 = "Java";
String s3 = new String("Java");
String s4 = new String("Java");
System.out.println(s1 == s2);      // true
System.out.println(s1 == s3);      // false
System.out.println(s1.equals(s3)); // true
```

避免 NullPointerException 以下代码可能抛出异常：

```
String input = null;
System.out.println(input.equals("yes")); // NullPointerException
```

当比较对象中有一个是固定字符串时，可以将固定字符串写在前面：

```
System.out.println("yes".equals(input)); // false
```

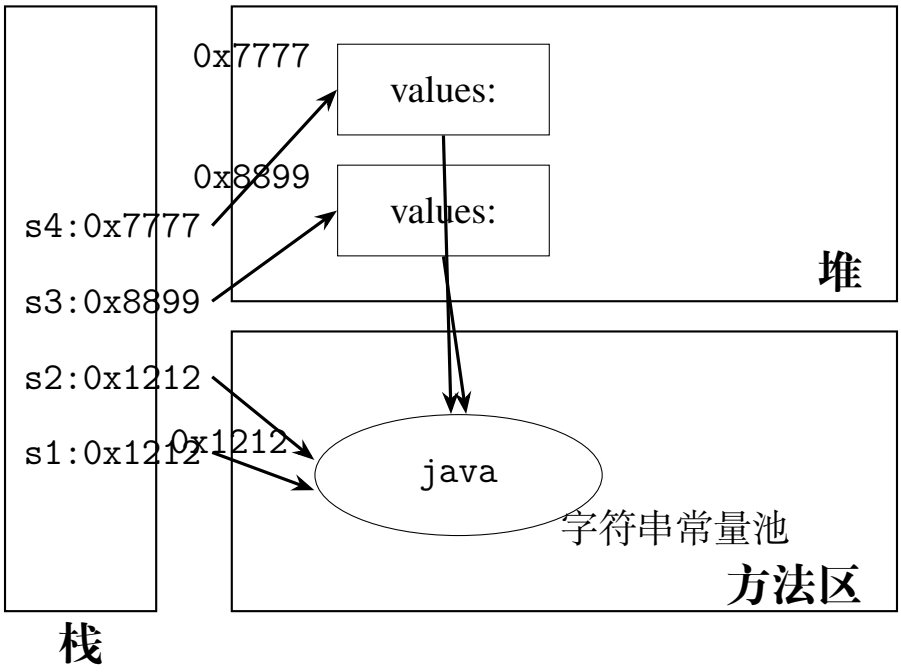


图 1.2: String 对象与字符串常量池关系示意图

也可以使用：

```
import java.util.Objects;
System.out.println(Objects.equals(a, b));
```

Objects.equals() 能够安全处理 null。

intern() 方法 intern() 返回字符串常量池中与当前字符串内容相同的引用。

```
String s1 = new String("hello");
String s2 = s1.intern();
String s3 = "hello";

System.out.println(s1 == s3); // false
System.out.println(s2 == s3); // true
```

字符串拼接

常量之间的拼接 如果参与拼接的内容都是编译期常量，编译器通常会直接计算出最终结果：

```
String s1 = "a" + "b";
String s2 = "ab";

System.out.println(s1 == s2); // true
```

变量参与拼接　只要表达式中包含普通变量，拼接通常需要在运行时完成：

```
String a = "a";
String s1 = a + "b";
String s2 = "ab";
System.out.println(s1 == s2);      // false
System.out.println(s1.equals(s2)); // true
```

即使变量当前保存的是常量池字符串，编译器也不一定能将其视为编译期常量。

final 编译期常量　如果变量是能够在编译期确定的 final 常量，编译器可能继续执行常量折叠：

```
final String a = "a";

String s1 = a + "b";
String s2 = "ab";

System.out.println(s1 == s2); // true
```

但如果值需要在运行时计算，即使使用 final，也不属于编译期常量：

```
final String a = new String("a");

String s1 = a + "b";
String s2 = "ab";

System.out.println(s1 == s2); // false
```

循环中的字符串拼接　在循环中反复使用 + 拼接字符串，可能产生大量临时字符串对象。不推荐：

```
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;
}
```

推荐使用 StringBuilder：

```
StringBuilder builder = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    builder.append(i);
}
String result = builder.toString();
```

对于简单的少量拼接：

```
String fullName = firstName + " " + lastName;
```

直接使用 + 通常更加清晰，不需要手动创建 StringBuilder。

StringBuilder 类

StringBuilder 表示一个可变字符序列。与 String 不同，对 StringBuilder 执行修改操作时，通常直接修改当前对象，而不是每次创建新的字符串对象。

创建 StringBuilder 三种办法：

```
StringBuilder a = new StringBuilder(); // 无初始化
StringBuilder b = new StringBuilder("初始内容"); // 设置初始内容
StringBuilder c = new StringBuilder(100); // 设置初始容量（不表示字符串已有100个字符）
```

StringBuilder 的常用方法

1. append() 在末尾追加内容：

   ```
   StringBuilder builder = new StringBuilder();
   builder.append("Name: ");
   builder.append(", Age: ");
   builder.append(20);
   ```

   append() 返回当前对象本身，因此可以进行链式调用：

   ```
   StringBuilder builder = new StringBuilder();
   builder.append("Name: ")
          .append(", Age: ")
          .append(20);
   ```

2. insert()

   在指定位置插入内容：

   ```
   StringBuilder builder = new StringBuilder("Helo");
   builder.insert(3, "l");
   System.out.println(builder); // Hello
   ```

3. delete()

   删除指定范围内的内容：

   ```
   StringBuilder builder = new StringBuilder("Hello Java");
   builder.delete(5, 10);
   ```

```
System.out.println(builder); // Hello
```

删除范围同样是：[start, end)

4. deleteCharAt()

   删除指定位置的字符：

```
StringBuilder builder = new StringBuilder("Helloo");
builder.deleteCharAt(5);
System.out.println(builder); // Hello
```

5. replace()

   替换指定范围：

```
StringBuilder builder = new StringBuilder("Hello Java");
builder.replace(6, 10, "Python");
System.out.println(builder); // Hello Python
```

6. setCharAt()

   直接修改指定位置的字符：

```
StringBuilder builder = new StringBuilder("Java");
builder.setCharAt(0, 'L');
System.out.println(builder); // Lava
```

7. reverse()

   反转字符序列：

```
StringBuilder builder = new StringBuilder("Java");
builder.reverse();
System.out.println(builder); // avaJ
```

8. toString()

   将 StringBuilder 转换为 String：

```
StringBuilder builder = new StringBuilder("Java");
String result = builder.toString();
```

StringBuffer 类

StringBuffer 同样表示可变字符序列，**其常用方法与 StringBuilder 基本相同**：

```
StringBuffer buffer = new StringBuffer("初始内容");

buffer.append("追加内容");
```

```
buffer.insert(2, "插入");
buffer.delete(0, 2);
buffer.replace(0, 3, "替换");
buffer.reverse();
```

StringBuffer 的许多方法带有同步机制（当多个线程同时访问同一个可变对象时，通过加锁控制访问，避免它们同时修改对象而导致数据混乱），而 StringBuilder 通常不提供同步保证。

String、StringBuilder 与 StringBuffer 的比较

类	是否可变	同步特点	主要使用场景
String	不可变	不涉及修改同步	保存固定文本、参数、返回值和键
StringBuilder	可变	通常不同步	单线程中的大量字符串拼接与修改
StringBuffer	可变	许多方法同步	需要共享可变字符序列的部分多线程场景

表 1.13: 字符串相关类的比较

实际开发中：

- 字符串内容基本不变时，使用 String；
- 单线程中频繁修改字符串时，优先使用 StringBuilder；
- 确实需要同步的可变字符序列时，可以考虑 StringBuffer；但不应仅因为代码位于多线程程序中，就机械地选择 StringBuffer，还需判断该对象是否真的被多个线程共享。

String 的方法

基础办法

1. length()

   返回字符串的长度：

   ```
   String text = "abc";
   System.out.println(text.length()); // 3
   ```

   需要区分：

   ```
   array.length;       // 数组长度
   text.length();      // 字符串长度
   list.size();        // 集合大小
   ```

2. isEmpty()

   判断字符串长度是否为零：

   ```
   String text = "";
   System.out.println(text.isEmpty()); // true
   ```

其含义近似于：

```
text.length() == 0
```

3. `isBlank()`

   从 Java 11 开始，可以使用 `isBlank()` 判断字符串是否为空，或者是否只包含空白字符：

```
System.out.println("".isBlank());      // true
System.out.println("   ".isBlank());   // true
System.out.println("\t\n".isBlank()); // true
System.out.println("abc".isBlank());  // false
```

4. `charAt()` 返回指定索引位置的字符：

```
String text = "abc";
char ch = text.charAt(1);
System.out.println(ch); // b
```

   有效索引范围为：$0 \le i < \texttt{text.length()}$

   若索引越界，会抛出 `StringIndexOutOfBoundsException`。

比较与查找方法　详细见表格

`compareTo()` 按照字典序进行比较：

```
int result = "a".compareTo("b");
System.out.println(result); // 负数
```

返回值规则为：

- 小于 0：当前字符串排在参数字符串前面；
- 等于 0：两个字符串内容相同；
- 大于 0：当前字符串排在参数字符串后面。

通常只判断返回值的正负，不应依赖具体差值：

```
if (a.compareTo(b) < 0) {
    System.out.println("a 排在 b 前面");
}
```

转换与格式化方法　详见表格

这些方法不会修改原字符串，而是返回新的字符串对象。

- `strip()` 对 Unicode 空白字符的识别通常比 `trim()` 更完整。
- `replace()` 的参数按照普通字符序列处理，不是正则表达式。

方法	描述	示例
`equals(obj)`	比较字符串内容是否相等，区分大小写	`"abc".equals("ABC")` → `false`
`equalsIgnoreCase(str)`	比较字符串内容是否相等，忽略大小写	`"abc".equalsIgnoreCase("ABC")` → `true`
`compareTo(str)`	按字典序比较字符串，返回负数、零或正数	`"a".compareTo("b")` → `-1`
`contains(str)`	判断字符串中是否包含指定字符序列	`"abc".contains("b")` → `true`
`indexOf(str)`	返回子串第一次出现的索引；未找到时返回 `-1`	`"abcabc".indexOf("b")` → `1`
`lastIndexOf(str)`	返回子串最后一次出现的索引；未找到时返回 `-1`	`"abcabc".lastIndexOf("b")` → `4`
`startsWith(prefix)`	判断字符串是否以指定前缀开头	`"hello".startsWith("he")` → `true`
`endsWith(suffix)`	判断字符串是否以指定后缀结尾	`"hello".endsWith("lo")` → `true`

表 1.14: String 的比较与查找方法

方法	描述	示例
`toLowerCase()`	将字符串中的字母转换为小写	`"ABC".toLowerCase()` → `"abc"`
`toUpperCase()`	将字符串中的字母转换为大写	`"abc".toUpperCase()` → `"ABC"`
`trim()`	去除字符串首尾的传统空白字符	`" abc ".trim()` → `"abc"`
`strip()`	去除字符串首尾的 Unicode 空白字符	`" abc ".strip()` → `"abc"`
`replace(old, new)`	替换所有匹配的普通字符或字符序列	`"abc".replace("a", "A")` → `"Abc"`
`replaceFirst(regex, value)`	替换第一个符合正则表达式的部分	`"a1b2c3".replaceFirst("\\d", "-")` → `"a-b2c3"`
`replaceAll(regex, value)`	替换所有符合正则表达式的部分	`"a1b2c3".replaceAll("\\d", "")` → `"abc"`
`split(regex)`	按照正则表达式分割字符串，返回字符串数组	`"a,b,c".split(",")` → `["a", "b", "c"]`
`String.join(delimiter, elements)`	使用指定分隔符连接多个字符串	`String.join("-", "a", "b", "c")` → `"a-b-c"`

表 1.15: String 的转换与格式化方法

`split()` 按照正则表达式分割字符串，返回字符串数组

```
String[] parts = "a,b,c".split(",");

for (String part : parts) {
    System.out.println(part);
} // output: "a\nb\nc\n"
```

需要注意，`split()` 的参数是正则表达式。
例如，点号在正则表达式中具有特殊含义，按照普通点号分割时必须转义：

```
String fileName = "archive.tar.gz";
String[] parts = fileName.split("\\.");
```

字符串截取 `substring()`

1. `substring(begin)` 从指定索引开始截取到字符串末尾：

   ```
   String result = "abcde".substring(2);
   System.out.println(result); // cde
   ```

2. `substring(begin, end)` 截取指定区间：

   ```
   String result = "abcde".substring(1, 3);
   System.out.println(result); // bc
   ```

   区间采用左闭右开形式：[beginIndex, endIndex)

字符串与基本类型之间的转换

1. **字符串转换为基本数据类型**，通常使用包装类的解析方法：

   ```
   int number = Integer.parseInt("123");
   long count = Long.parseLong("1000");
   double price = Double.parseDouble("3.14");
   boolean flag = Boolean.parseBoolean("true");
   ```

   如果数字字符串格式不合法，会抛出 `NumberFormatException`：

   ```
   int number = Integer.parseInt("abc");
   // NumberFormatException
   ```

2. **基本数据类型转换为字符串**

   ```
   String s1 = String.valueOf(456);
   String s2 = String.valueOf(3.14);
   ```

```
String s3 = String.valueOf(true);
```

也可以使用字符串拼接：

```
String text = 789 + "";
```

但只进行类型转换时，String.valueOf() 的语义更加明确。

字符串与字符数组之间的转换

1. **字符串转换为字符数组**

```
String text = "abcde";
char[] chars = text.toCharArray();
```

2. **字符数组转换为字符串**

```
char[] data = {'h', 'e', 'l', 'l', 'o'};
String text1 = new String(data);
String text2 = String.valueOf(data);
```

还可以只使用字符数组的一部分：

```
char[] data = {'a', 'b', 'c', 'd', 'e'};
String text = new String(data, 1, 3);
System.out.println(text); // bcd
```

其中：

- 第二个参数表示开始位置；
- 第三个参数表示字符数量。

3. **字符串与字节数组之间的转换**：字符串与字节之间的转换涉及字符编码。

```
import java.nio.charset.StandardCharsets;
String text = "中文";
byte[] bytes = text.getBytes(StandardCharsets.UTF_8);
```

将字节数组转换回字符串：

```
String result = new String(bytes, StandardCharsets.UTF_8);
```

应保证编码和解码使用相同的字符集，不推荐依赖系统默认字符集：

```
byte[] bytes = text.getBytes();
String result = new String(bytes);
```

因为不同操作系统或运行环境的默认字符集可能不同。

正则表达式相关方法

matches() 判断整个字符串是否符合正则表达式：

```
boolean result = "123".matches("\\d+");

System.out.println(result); // true
```

需要注意，matches() 要求整个字符串匹配，而不是只要其中一部分匹配即可。
例如：

```
System.out.println(
    "abc123".matches("\\d+")
); // false
```

若允许前面存在任意字符，可以写为：

```
System.out.println(
    "abc123".matches(".*\\d+")
); // true
```

常见正则表达式包括：

正则表达式	含义
\d	数字字符
\w	字母、数字或下划线
\s	空白字符
.	任意字符
*	前一个模式出现零次或多次
+	前一个模式出现一次或多次
?	前一个模式出现零次或一次

表 1.16: 常见正则表达式符号

在 Java 字符串中，反斜杠本身需要再次转义：

```
"\\d+" // 实际传递给正则表达式引擎的是：\d+
```

1.5.4 时间 API

Java 的时间 API 主要可以分为两代：

- JDK 8 之前：`Date`、`Calendar`、`SimpleDateFormat`
- JDK 8 之后：`java.time` 包中的新时间 API

新项目应优先使用 `java.time` 包。旧时间 API 主要用于阅读旧代码或与旧系统进行兼容。

时间的三种常见语义

在使用时间 API 前，应先明确当前数据表示的是什么。

1. **日期**
   只包含年、月、日，不包含具体时刻：
   ```
   LocalDate birthday = LocalDate.of(2000, 10, 1);
   ```
   常见场景包括生日、纪念日、账单日期。
2. **本地日期时间**
   包含年、月、日、时、分、秒，但没有时区信息：
   ```
   LocalDateTime meeting = LocalDateTime.of(2025, 7, 1, 14, 30);
   ```
   `LocalDateTime` 不包含时区信息，因此它本身不能唯一确定全球时间线上的某个时刻。
3. **时间线上的绝对时刻**
   表示全球时间线上的唯一时刻：
   ```
   Instant instant = Instant.now();
   ```
   常用于日志时间、数据库时间戳和系统之间的数据传输。

JDK 8 之前的旧时间 API

System.currentTimeMillis() `System.currentTimeMillis()` 返回当前时刻对应的 Unix 时间戳，单位为毫秒。Unix 纪元的起点为：1970-01-01T00:00:00Z 其中，`Z` 表示 UTC。

```
long timestamp = System.currentTimeMillis();
System.out.println(timestamp);
```

时间戳本质上是一个 `long` 类型整数，适合存储和比较，但不适合直接展示给用户。

计算程序运行时间 不推荐使用 `currentTimeMillis()` 精确测量程序耗时，因为系统时间可能被人工或操作系统调整。计算经过时间通常使用：

```
long start = System.nanoTime();

doSomething(); // 需要测量的代码
```

```
4
5 long end = System.nanoTime();
6 long elapsedNanos = end - start;
7 System.out.println(elapsedNanos);
```

需要区分：

- currentTimeMillis()：表示当前现实时间；
- nanoTime()：适合测量一段代码经过了多久。

Date 类 java.util.Date 表示时间线上的一个时刻，内部保存从 Unix 纪元开始经过的毫秒数。

```
1 // 创建Date对象
2 Date nowDate = new Date();
3 System.out.println("当前日期：" + nowDate); // 输出类似 Tue Jun 24 14:30:00 CST 2025
4
5 // 获取时间戳
6 long timeStamp = nowDate.getTime();
7 System.out.println("时间戳：" + timeStamp);
8
9 // 从时间戳创建Date对象
10 Date dateFromStamp = new Date(timeStamp);
11 System.out.println("从时间戳创建的日期：" + dateFromStamp);
```

Date 本身不保存时区，但其 toString() 方法又使用系统默认时区显示时间，让时区处理变得麻烦。

Date 的主要问题

- 类名为 Date，实际却同时表示日期和时间
- 月份从 0 开始：1 月是 0，12 月是 11，这简直是” 反人类设计”，程序员经常因为记错而踩坑
- 很多早期方法已经被标记为 @Deprecated
- Date 对象是可变的，这在多线程环境下容易引发问题

例如：

```
1 Date date = new Date();
2 date.setTime(0L);
3 System.out.println(date);
```

调用 setTime() 会直接修改原来的 Date 对象。

SimpleDateFormat SimpleDateFormat 用于完成：

- Date 转换为字符串: 使用 SimpleDateFormat(String pattern) 创建格式化器，pattern 参数定义了

输出格式，比如”yyyy-MM-dd HH:mm:ss”

- 字符串解析为 Date：使用 parse(String source) 方法将字符串解析为 Date 对象，但需要处理 ParseException 异常

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

Date nowDate = new Date();

// 默认格式化（简单但不灵活）
SimpleDateFormat defaultFormatter = new SimpleDateFormat();
System.out.println("默认格式: " + defaultFormatter.format(nowDate)); // 输出类似 25-6-24 下午2:30

// 自定义格式化（灵活但需要记模式）
SimpleDateFormat customFormatter = new SimpleDateFormat("yyyy年MM月dd日 E HH:mm:ss");
System.out.println("自定义格式: " + customFormatter.format(nowDate)); // 输出类似 2025年06月24日 星期二 14:30:00
```

解析字符串需要处理 ParseException：

```
try {
    Date date = formatter.parse("2025-07-01 14:30:00");
    System.out.println(date);
} catch (ParseException e) {
    System.out.println("时间格式错误");
}
```

SimpleDateFormat 的问题

- SimpleDateFormat 是可变类，不能安全地被多个线程同时共享
- 格式字符容易混淆：需要记住各种模式字符（yyyy 表示年，MM 表示月，dd 表示日，HH 表示 24 小时制，hh 表示 12 小时制，mm 表示分钟，ss 表示秒钟）
- 默认解析行为可能过于宽松，且解析和时区处理不够清晰

Calendar 类 Calendar 是一个抽象类，通常通过 getInstance() 获取实例：

```
import java.util.Calendar;

Calendar calendar = Calendar.getInstance(); // 根据本地时区和语言环境返回一个具体的子类

int year = calendar.get(Calendar.YEAR);
```

```
int month = calendar.get(Calendar.MONTH);
int day = calendar.get(Calendar.DAY_OF_MONTH);
```

需要注意：

- Calendar.MONTH 的范围是 0--11(一月对应 0，十二月对应 11)
- Calendar.SUNDAY 的值为 1。

修改日期可以使用：

```
calendar.set(Calendar.DAY_OF_MONTH, 8);
calendar.add(Calendar.HOUR_OF_DAY, 2);
calendar.add(Calendar.MONTH, -2);
```

Calendar 比 Date 功能更丰富，但仍然存在对象可变、字段复杂和月份从零开始等问题。

旧时间 API 的主要问题 见表格

类	主要作用	主要问题
Date	表示一个时间点	对象可变（非线程安全）；时区语义不清晰
Calendar	获取、设置和计算日期字段	对象可变（非线程安全）；月份从零开始
SimpleDateFormat	格式化和解析时间	对象可变（非线程安全）；格式字符容易混淆

表 1.17: JDK 8 之前的主要时间类

JDK 8 的 java.time API

JDK 8 引入了 java.time 包。
新时间 API 的核心设计特点包括：

- **不可变性**（线程安全）：所有新时间类都是不可变的，避免了多线程安全问题，就像给时间上了” 锁”，保证了数据的一致性
- **清晰性**：分离了日期（LocalDate）、时间（LocalTime）和日期时间（LocalDateTime），各司其职，不再像 Date 那样” 一锅炖”
- **实用性**：提供了丰富的方法用于时间计算、格式化和解析，而且时区处理更加清晰

新时间类的方法通常不会修改原对象，而是返回一个新的时间对象。

```
LocalDate date = LocalDate.of(2025, 7, 1);
date.plusDays(1);
System.out.println(date); // 2025-07-01
 // 如果希望保存计算结果，必须接收返回值
date = date.plusDays(1);
System.out.println(date); // 2025-07-02
```

类	表示内容	常见场景
LocalDate	年、月、日	生日、纪念日、账单日期
LocalTime	时、分、秒、纳秒	营业时间、每日会议时间
LocalDateTime	日期和时间，但不包含时区	已知上下文中的本地日期时间
Instant	时间线上的绝对时刻	日志、时间戳、跨系统传输
ZoneId	地区时区规则	Asia/Shanghai 等地区时区
ZonedDateTime	日期、时间和地区时区	跨时区会议、国际业务
Duration	基于秒和纳秒的时间间隔	程序耗时、小时和分钟间隔
Period	基于年、月、日的日期间隔	年龄、日期跨度
DateTimeFormatter	格式化和解析时间	时间对象与字符串转换

表 1.18: java.time 中的核心类

LocalDate

1. **创建 LocalDate** LocalDate 只表示日期：年 – 月 – 日，不包含具体时间和时区。

```
LocalDate today = LocalDate.now(); // 获取当前日期
LocalDate birthday = LocalDate.of(2000, 10, 1); // 创建指定日期:
LocalDate date = LocalDate.parse("2025-07-01"); // 解析标准日期字符串
```

   如果指定的日期不存在，会抛出 DateTimeException：

```
LocalDate date = LocalDate.of(2025, 2, 29); // DateTimeException
```

2. **获取日期字段** 获取日期组件

```
LocalDate date = LocalDate.of(2025, 7, 1);

int year = date.getYear();
int month = date.getMonthValue();
int day = date.getDayOfMonth();
DayOfWeek week = date.getDayOfWeek();
int dayOfYear = date.getDayOfYear();
```

3. **修改日期字段** 修改已经设置好的日期

```
LocalDate date = LocalDate.of(2025, 7, 1);

LocalDate changed = date.withYear(2026)
                        .withMonth(12)
                        .withDayOfMonth(25);

```

```
System.out.println(date);    // 2025-07-01
System.out.println(changed); // 2026-12-25
```

4. **日期计算** 计算新的日期

```
LocalDate date = LocalDate.of(2025, 7, 1);

LocalDate tomorrow = date.plusDays(1);
LocalDate thirtyDaysLater = date.plusDays(30);
LocalDate nextWeek = date.plusWeeks(1);
LocalDate nextMonth = date.plusMonths(1);
LocalDate lastYear = date.minusYears(1);
```

5. **日期比较** 比较日期先后

```
LocalDate a = LocalDate.of(2025, 7, 1);
LocalDate b = LocalDate.of(2025, 8, 1);

System.out.println(a.isBefore(b)); // true
System.out.println(a.isAfter(b));  // false
System.out.println(a.isEqual(b));  // false
```

6. **其他常用方法**

```
LocalDate date = LocalDate.of(2024, 2, 1);

System.out.println(date.isLeapYear());    // true
System.out.println(date.lengthOfMonth()); // 29
System.out.println(date.lengthOfYear());  // 366
```

LocalTime

LocalTime 只表示一天中的时间：时 : 分 : 秒.纳秒，不包含日期和时区。

1. **创建 LocalTime** 创建对象

```
LocalTime now = LocalTime.now(); // 获取当前时间，输出类似 14:30:00.123456789

LocalTime lunch = LocalTime.of(12, 30); // 指定时、分创建时间，输出 12:30

LocalTime precise = LocalTime.of(12, 30, 15, 500_000_000); //
    指定时、分、秒、纳秒创建时间
```

2. **获取时间字段** 时、分、秒、纳

```
int hour = precise.getHour();
int minute = precise.getMinute();
int second = precise.getSecond();
int nano = precise.getNano();
```

3. **修改和计算时间** 计算时间

```
LocalTime time = LocalTime.of(10, 30, 20);

LocalTime startOfHour = time.withMinute(0) // 将分钟改为0
                            .withSecond(0) // 将秒钟改为0
                            .withNano(0); // 将纳秒改为0

LocalTime oneHourLater = time.plusHours(1); // 时间计算：1小时后

LocalTime thirtyMinutesAgo = time.minusMinutes(30); // 时间计算：30分钟前
```

跨越零点：LocalTime 不包含日期，时间计算会在一天内循环：

```
LocalTime time = LocalTime.of(23, 30);
LocalTime result = time.plusHours(2);
System.out.println(result); // 01:30
```

结果中无法看出日期已经进入第二天，所以如果日期变化很重要，应使用 LocalDateTime。

LocalDateTime

LocalDateTime 是 LocalDate 和 LocalTime 的组合。

1. **创建 LocalDateTime** 创建对象

```
LocalDateTime now = LocalDateTime.now(); // 获取当前日期时间
LocalDateTime meeting = LocalDateTime.of(2025, 7, 1, 10, 30); // 创建指定日期时间

LocalDate date = LocalDate.of(2025, 7, 1);
LocalTime time = LocalTime.of(10, 30);
LocalDateTime meeting = LocalDateTime.of(date, time); //
    从LocalDate和LocalTime创建
```

2. **提取/拆分日期和时间** 从 LocalDateTime 提取 LocalDate 和 LocalTime 对象；继承了 LocalDate 和 LocalTime 的所有 get 方法。

```
LocalDate date = meeting.toLocalDate(); // 拆分日期
LocalTime time = meeting.toLocalTime(); // 拆分时间
```

```
3
4  LocalDateTime dateTime = LocalDateTime.of(2025, 7, 1, 14, 30, 20, 500_000_000);
5
6  // 获取日期字段
7  int year = dateTime.getYear();
8  int month = dateTime.getMonthValue();
9  int day = dateTime.getDayOfMonth();
10 int dayOfYear = dateTime.getDayOfYear();
11
12 Month monthEnum = dateTime.getMonth();
13 DayOfWeek week = dateTime.getDayOfWeek();
14
15 // 获取时间字段
16 int hour = dateTime.getHour();
17 int minute = dateTime.getMinute();
18 int second = dateTime.getSecond();
19 int nano = dateTime.getNano();
```

3. **日期时间修改和计算** 继承了 `LocalDate` 和 `LocalTime` 的所有 with 方法及所有 plus 和 minus 方法。

```
1 LocalDateTime changed = meeting.plusDays(3)
2                                .withHour(14)
3                                .withMinute(0)
4                                .withSecond(0)
5                                .withNano(0);
```

Instant

`Instant` 表示时间线上的一个绝对时刻，常用于日志记录、时间戳存储和不同系统之间的时间传输。`Instant` 本身不包含地区时区，其默认字符串形式使用 UTC，末尾的 `Z` 表示零时区偏移。
其内部可以理解为：

- 从 Unix 纪元开始经过的秒数；
- 当前秒内的纳秒调整值。

Unix 纪元的起点为：1970-01-01T00:00:00Z

1. **创建 Instant**

```
1 Instant nowInstant = Instant.now(); // 获取当前瞬时点，输出为 UTC 时间
2 Instant fromMillis = Instant.ofEpochMilli(1_700_000_000_000L); // 从毫秒时间戳创建
3 Instant fromSeconds = Instant.ofEpochSecond(1_700_000_000L); // 从秒级时间戳创建
```

```
Instant parsed = Instant.parse("2025-07-01T06:00:00Z"); // 解析 ISO-8601 字符串
```

2. **获取时间戳**

```
Instant instant = Instant.now();

long milliseconds = instant.toEpochMilli(); // 获取毫秒时间戳
long seconds = instant.getEpochSecond();     // 获取纪元秒数
int nano = instant.getNano();                // 获取当前秒内的纳秒部分
```

3. **Instant 与 Date 转换**

```
Date oldDate = new Date();

Instant instantFromDate = oldDate.toInstant(); // Date -> Instant
Date newDate = Date.from(instantFromDate);      // Instant -> Date
```

4. **时间计算**

Instant 是不可变类，计算方法不会修改原对象，而是返回新的 Instant 对象。

```
Instant now = Instant.now();

Instant oneHourLater = now.plusSeconds(3600);  // 1 小时后
Instant tenMinutesAgo = now.minusSeconds(600); // 10 分钟前
```

计算两个瞬时点之间的时间间隔：

```
Instant now = Instant.now();
Instant earlier = now.minusSeconds(3600); // 1 小时前

Duration duration = Duration.between(earlier, now);
System.out.println(duration.toMinutes()); // 60
```

5. **比较 Instant**

```
Instant first = Instant.parse("2025-07-01T06:00:00Z");
Instant second = Instant.parse("2025-07-01T07:00:00Z");

System.out.println(first.isBefore(second)); // true
System.out.println(first.isAfter(second));  // false
System.out.println(first.equals(second));   // false
```

6. **转换为指定时区的日期时间**

```
Instant instant = Instant.parse("2025-07-01T06:00:00Z");

ZonedDateTime shanghaiTime = instant.atZone(ZoneId.of("Asia/Shanghai"));

System.out.println(shanghaiTime); // 2025-07-01T14:00+08:00[Asia/Shanghai]
```

需要记录全球唯一时刻时，应优先使用 Instant，而不是不包含时区的 LocalDateTime。

DateTimeFormatter

DateTimeFormatter 用于在时间对象和字符串之间进行转换，是新时间 API 中对应于 SimpleDateFormat 的格式化工具。DateTimeFormatter 是不可变且线程安全的，因此可以安全地作为静态常量在多个线程之间共享。

1. **使用预定义标准格式**

   Java 提供了多种预定义的 ISO 格式：

   ```
   LocalDateTime now = LocalDateTime.now();

   DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
   String text = now.format(formatter); // 时间对象 -> 字符串
   System.out.println(text); // 输出类似 2025-06-24T14:30:00.123
   ```

   只格式化日期：

   ```
   LocalDate date = LocalDate.of(2025, 7, 1);

   String text = date.format(DateTimeFormatter.ISO_LOCAL_DATE);
   System.out.println(text); // 2025-07-01
   ```

2. **使用本地化格式**

   本地化格式会根据指定的语言环境选择显示方式：

   ```
   ZonedDateTime now = ZonedDateTime.now(ZoneId.of("Asia/Shanghai"));

   DateTimeFormatter formatter =
       DateTimeFormatter
           .ofLocalizedDateTime(FormatStyle.LONG)
           .withLocale(Locale.CHINA);

   String text = now.format(formatter);
   System.out.println(text); // 输出形式取决于 JDK 和语言环境
   ```

3. **使用自定义格式**

```java
LocalDateTime now = LocalDateTime.of(2025, 7, 1, 14, 30, 20);

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("uuuu/MM/dd HH:mm:ss");
String text = now.format(formatter);
System.out.println(text); // 2025/07/01 14:30:20
```

4. **解析字符串**

```java
String text = "2025-12-25T18:00:00";

LocalDateTime dateTime =
    LocalDateTime.parse(
        text,
        DateTimeFormatter.ISO_LOCAL_DATE_TIME
    );
System.out.println(dateTime); // 2025-12-25T18:00
```

解析自定义格式：

```java
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("uuuu-MM-dd HH:mm:ss");
LocalDateTime dateTime = LocalDateTime.parse("2025-07-01 14:30:20", formatter);
```

格式不符合要求时，会抛出 DateTimeParseException：

```java
try {
    LocalDate date =
        LocalDate.parse(
            "2025/07/01",
            DateTimeFormatter.ISO_LOCAL_DATE
        );
} catch (DateTimeParseException e) {
    System.out.println("日期格式错误");
}

```

5. **格式化带时区的日期时间**

```java
ZonedDateTime tokyoTime = ZonedDateTime.now(ZoneId.of("Asia/Tokyo"));

DateTimeFormatter fmtr = DateTimeFormatter.ofPattern("uuuu-MM-dd HH:mm:ss z");
String text = tokyoTime.format(fmtr);
System.out.println(text); // 输出类似 2025-06-24 15:30:00 JST
```

常见格式字符如下：

字符	含义	示例
uuuu	年	2025
MM	月	07
dd	日	01
HH	24 小时制小时	14
hh	12 小时制小时	02
mm	分钟	30
ss	秒	20
SSS	毫秒部分	123
a	上午或下午	AM/PM
XXX	UTC 偏移量	+08:00
z	时区名称	CST

表 1.19: DateTimeFormatter 常见格式字符

需要特别注意：

- MM 表示月份，mm 表示分钟；
- HH 表示 24 小时制，hh 表示 12 小时制；
- 严格日期处理中通常使用 uuuu 表示年份；
- 格式中包含英文月份或星期时，应显式指定 Locale。

ZoneId 与 ZonedDateTime

ZoneId 表示地区时区规则，ZonedDateTime 表示包含日期、时间和时区的完整日期时间。常见时区 ID 采用“地区/城市”的形式：

- Asia/Shanghai
- Europe/Paris
- America/New_York

地区时区不仅包含当前 UTC 偏移量，还可能包含夏令时和历史时区变化规则。

1. **获取 ZoneId**

```
ZoneId defaultZone = ZoneId.systemDefault(); // 获取系统默认时区
ZoneId shanghaiZone = ZoneId.of("Asia/Shanghai"); // 获取指定地区时区

Set<String> allZoneIds = ZoneId.getAvailableZoneIds(); // 获取所有可用的时区 ID
System.out.println("共有 " + allZoneIds.size() + " 个时区定义");
allZoneIds.stream().limit(10).forEach(System.out::println); // 打印前 10 个时区
```

2. **创建 ZonedDateTime**

```
ZoneId zone = ZoneId.of("Asia/Shanghai");
ZonedDateTime now = ZonedDateTime.now(zone); // 获取指定时区的当前日期时间
```

将 LocalDateTime 与时区结合：

```
LocalDateTime localDateTime = LocalDateTime.of(2025, 7, 1, 14, 0);
ZonedDateTime zonedDateTime =
    ZonedDateTime.of(
        localDateTime,
        ZoneId.of("Asia/Shanghai")
    ); // 为本地日期时间指定时区

// 也可以这么写
ZonedDateTime zonedDateTime = localDateTime.atZone(ZoneId.of("Asia/Shanghai"));
```

Instant 与 ZonedDateTime 转换：

```
Instant instant = Instant.parse("2025-07-01T06:00:00Z");

ZonedDateTime shanghaiTime =
    ZonedDateTime.ofInstant(
        instant,
        ZoneId.of("Asia/Shanghai")
    ); // Instant -> ZonedDateTime

Instant result = shanghaiTime.toInstant(); // ZonedDateTime -> Instant
```

3. **转换时区**

withZoneSameInstant() 保持绝对时刻不变，只改变该时刻在另一个时区中的显示时间。

```
ZonedDateTime beijingTime =
    ZonedDateTime.of(
        2025, 7, 1, 14, 0, 0, 0,
        ZoneId.of("Asia/Shanghai")
    );

ZonedDateTime tokyoTime =
   beijingTime.withZoneSameInstant(ZoneId.of("Asia/Tokyo"));
System.out.println("东京时间: " + tokyoTime);
```

4. **计算不同时区中的同一时刻**

```
LocalDateTime meetingTime = LocalDateTime.of(2025, 7, 1, 14, 0);

ZonedDateTime meetingInBeijing = meetingTime.atZone(ZoneId.of("Asia/Shanghai"));
ZonedDateTime meetingInNewYork =
    meetingInBeijing.withZoneSameInstant(ZoneId.of("America/New_York"));

System.out.println("北京会议时间：" + meetingInBeijing);

System.out.println("纽约对应时间：" + meetingInNewYork);
```

5. **计算两个地区的 UTC 偏移量差**

   不应直接使用两个对象的 getHour() 相减，因为这样无法正确处理跨日和夏令时。

```
ZoneOffset offset = ZoneOffset.of("+08:00"); // 手动创建固定 UTC 偏移量

int beijingOffset = meetingInBeijing.getOffset().getTotalSeconds();
int newYorkOffset = meetingInNewYork.getOffset().getTotalSeconds();
int offsetDifferenceHours = (beijingOffset - newYorkOffset) / 3600;

System.out.println("当前偏移量相差：" + offsetDifferenceHours + " 小时");
```

需要区分：

- withZoneSameInstant()：保持绝对时刻不变；
- withZoneSameLocal()：保持本地日期时间不变，但代表的绝对时刻发生变化。

跨时区转换通常应使用 withZoneSameInstant()。

长期业务时间应优先保存地区时区 ZoneId，而不是只保存当前的固定偏移量。

Duration 与 Period

Duration 和 Period 都用于表示时间间隔，但二者的计算单位不同：

- Duration：基于秒和纳秒；
- Period：基于年、月、日。

1. **创建 Duration**

```
Duration tenSeconds = Duration.ofSeconds(10); // 10 秒
Duration thirtyMinutes = Duration.ofMinutes(30); // 30 分钟
Duration twoHours = Duration.ofHours(2); // 2 小时
Duration oneDay = Duration.ofDays(1); // 1 天，即 24 小时
```

2. **计算两个时间点之间的间隔**

```
LocalTime startTime = LocalTime.of(9, 0);
LocalTime endTime = LocalTime.of(18, 30);

Duration workDuration = Duration.between(startTime, endTime);

long hours = workDuration.toHours();
long minutes = workDuration.toMinutes() % 60;

System.out.println("工作时长：" + hours + " 小时 " + minutes + " 分钟");
```

3. **处理跨越零点的时间**

   单独使用两个 LocalTime 时，系统不知道结束时间是否属于第二天：

```
LocalTime eveningTime = LocalTime.of(23, 0);
LocalTime nextMorning = LocalTime.of(6, 30);

Duration sleepDuration = Duration.between(eveningTime, nextMorning);

if (sleepDuration.isNegative()) {
    sleepDuration = sleepDuration.plusDays(1); // 将结束时间视为第二天
}
System.out.println(sleepDuration.toHours()); // 7
```

   更严谨的写法是加入日期信息：

```
LocalDate date = LocalDate.of(2025, 7, 1);
LocalDateTime start = LocalDateTime.of(date, LocalTime.of(23, 0));
LocalDateTime end =cLocalDateTime.of(date.plusDays(1), LocalTime.of(6, 30));

Duration sleepDuration = Duration.between(start, end);
System.out.println(sleepDuration.toMinutes()); // 450
```

4. **计算两个 Instant 之间的间隔**

```
Instant start = Instant.now();
Instant end = start.plusSeconds(90);

Duration duration = Duration.between(start, end);

System.out.println(duration.toSeconds()); // 90
```

5. 使用 Period 计算日期间隔

```
LocalDate birthDate = LocalDate.of(1990, 5, 15);
LocalDate today = LocalDate.now();

Period age = Period.between(birthDate, today);

System.out.println(
    "年龄: "
    + age.getYears() + " 岁 "
    + age.getMonths() + " 月 "
    + age.getDays() + " 天"
);
```

6. 计算特定日期的倒计时

```
LocalDate today = LocalDate.now();
LocalDate christmas = LocalDate.of(today.getYear(), 12, 25);

if (christmas.isBefore(today)) {
    christmas = christmas.plusYears(1); // 今年已过则计算明年
}
Period untilChristmas = Period.between(today, christmas);

System.out.println(
    "距离圣诞节还有: "
    + untilChristmas.getMonths() + " 月 "
    + untilChristmas.getDays() + " 天"
);
```

7. 计算两个日期之间的总天数

Period.getDays() 只返回 Period 中的“日”部分，并不是两个日期之间的总天数。

计算总天数应使用 ChronoUnit：

```
LocalDate start = LocalDate.of(2025, 7, 1);
LocalDate end = LocalDate.of(2025, 7, 10);

long totalDays = ChronoUnit.DAYS.between(start, end);
System.out.println(totalDays); // 9
```

类	计算单位	常见用途
`Duration`	秒、纳秒	程序运行时间、小时和分钟间隔、瞬时点间隔
`Period`	年、月、日	年龄、合同期限、日历日期间隔

表 1.20: `Duration` 与 `Period` 的区别

TemporalAdjusters

`TemporalAdjuster` 是日期调整器接口，`TemporalAdjusters` 是提供常用日期调整器的工具类。使用它们可以避免手动编写复杂的日期计算逻辑。

1. **获取当月第一天和最后一天**

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

// 当月第一天
LocalDate firstDay = today.with(TemporalAdjusters.firstDayOfMonth());
// 当月最后一天
LocalDate lastDay = today.with(TemporalAdjusters.lastDayOfMonth());
```

2. **获取下一个指定星期**

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

LocalDate nextFriday =
    today.with(
        TemporalAdjusters.next(
            DayOfWeek.FRIDAY
        )
    ); // 严格寻找下一个星期五
```

如果当天就是星期五：

- `next(FRIDAY)` 返回下周五；
- `nextOrSame(FRIDAY)` 返回当天。

```java
LocalDate friday =
    today.with(
        TemporalAdjusters.nextOrSame(
            DayOfWeek.FRIDAY
        )
    );
```

3. **获取当月第 n 个指定星期**

```
LocalDate firstTuesday =
    today.with(
        TemporalAdjusters.firstInMonth(
            DayOfWeek.TUESDAY
        )
    ); // 当月第一个星期二

LocalDate secondTuesday =
    today.with(
        TemporalAdjusters.firstInMonth(
            DayOfWeek.TUESDAY
        )
    ).plusWeeks(1); / 手动获取当月第二个星期二

    today.with(
        TemporalAdjusters.dayOfWeekInMonth(
            2,
            DayOfWeek.TUESDAY
        )
    ); // 更直接
```

4. **自定义日期调整器**

以下调整器用于计算下一个工作日，只处理周六和周日，不包含法定节假日和调休：

```
// 自定义调整器 - 计算下一个工作日
TemporalAdjuster nextWorkingDay = temporal -> {
    LocalDate date = LocalDate.from(temporal);
    DayOfWeek day = date.getDayOfWeek();
    int daysToAdd;
    if (day == DayOfWeek.FRIDAY) {
        daysToAdd = 3; // 周五 -> 下周一
    } else if (day == DayOfWeek.SATURDAY) {
        daysToAdd = 2; // 周六 -> 下周一
    } else {
        daysToAdd = 1; // 其他 -> 次日
    }
    return date.plusDays(daysToAdd);
};

```

```
LocalDate nextWorkday = today.with(nextWorkingDay);
System.out.println("下一个工作日： " + nextWorkday);
);

```

新旧 API 对比与互操作性

特性	旧 API	新 API
对象可变性	Date 和 Calendar 可变	核心时间类不可变
线程安全	多数类不能安全共享	核心时间类线程安全
时区处理	时区概念与日期时间类分离，语义不清晰	使用 ZoneId 和 ZonedDateTime
日期时间分离	Date 同时包含日期和时间	使用不同类型表示不同时间语义
月份	Calendar 使用 0--11	使用符合习惯的 1--12
星期	使用数字常量，周日为 1	使用 DayOfWeek 枚举
格式化与解析	SimpleDateFormat 非线程安全	DateTimeFormatter 线程安全
时间计算	使用 set() 和 add()	使用 with()、plus() 和 minus()

表 1.21: 新旧 Java 时间 API 的核心差异

新旧 API 互转 实际项目中可能需要与旧代码兼容，此时可以进行新旧时间类型转换。

1. **Date 与 Instant**

```
// Date与Instant互转
Date oldDate = new Date();
Instant instant = oldDate.toInstant(); // Date -> Instant
Date newDate = Date.from(instant);     // Instant -> Date
```

2. **Calendar 与 ZonedDateTime**

```
// Calendar与ZonedDateTime互转
Calendar calendar = Calendar.getInstance();
// Calendar -> ZonedDateTime
ZonedDateTime zonedDateTime = convert(calendar);
// ZonedDateTime -> Calendar
Calendar newCalendar = GregorianCalendar.from(zonedDateTime);

public static ZonedDateTime convert(Calendar calendar){
        if(calendar==null){
            return null;
```

```
        }
        //1.获取Calendar的时间戳（毫秒数）
        Instant instant=Instant.ofEpochMilli(calendar.getTimeInMillis());
        //2.获取Calendar的时区
        TimeZone timeZone=calendar.getTimeZone();
        ZoneId zoneId=timeZone.toZoneId();
        //3.转化为ZonedDateTime
        return ZonedDateTime.ofInstant(instant,zoneId);
    }
```

3. **java.sql.Date 与 LocalDate**

```
LocalDate localDate = LocalDate.now();
// LocalDate -> java.sql.Date
java.sql.Date sqlDate = java.sql.Date.valueOf(localDate);
// java.sql.Date -> LocalDate
LocalDate result = sqlDate.toLocalDate();
```

4. **Timestamp 与 LocalDateTime**

```
LocalDateTime localDateTime = LocalDateTime.now();
// LocalDateTime -> Timestamp
java.sql.Timestamp timestamp = java.sql.Timestamp.valueOf(localDateTime);
// Timestamp -> LocalDateTime
LocalDateTime result = timestamp.toLocalDateTime();
```

只有在与旧系统或旧接口交互时，才应进行新旧 API 转换；同一业务流程中应尽量使用同一套时间 API。

1.5.5 比较器

Java 中的基本类型和部分内置类已经定义了比较规则，但自定义对象应该按照价格、年龄、成绩还是名称排序，需要由程序员明确指定。
Java 提供了两套对象比较机制：

- `Comparable`：自然排序，由对象自身定义默认比较规则；
- `Comparator`：定制排序，在类的外部临时指定比较规则。

Comparable：自然排序

`Comparable` 表示对象自身具备比较能力。类实现该接口后，就具有一种固定的自然排序规则。

1. **Comparable 接口** 定义对象默认的比较规则
   推荐使用泛型接口：

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

   `compareTo()` 的返回值规则为：

   - 返回负数：当前对象小于参数对象；
   - 返回 0：两个对象在排序意义上相等；
   - 返回正数：当前对象大于参数对象。

   比较结果只需要关注正数、零和负数，不能假设方法一定返回 1 或 -1。

2. **实现 Comparable** 让商品按照价格进行自然排序

```java
public class Goods implements Comparable<Goods> {

    private String name;
    private double price;

    public Goods(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
```

```
        return price;
    }

    @Override
    public int compareTo(Goods other) {
        return Double.compare(this.price, other.price); // 按价格升序
    }

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

在自然排序中：

- 价格较低的商品是较小对象；
- 价格较高的商品是较大对象；
- 价格相同时，compareTo() 返回 0。

3. **使用自然排序** Arrays.sort() 会自动调用对象的 compareTo() 方法

```
Goods[] goods = {
    new Goods("西游记", 80),
    new Goods("红楼梦", 100),
    new Goods("水浒传", 120),
    new Goods("三国演义", 140)
};

Arrays.sort(goods); // 按 Goods 的自然顺序排序
for (Goods good : goods) {
    System.out.println(good);
}
```

对于集合，也可以使用自然排序：

```
List<Goods> goodsList = new ArrayList<>(Arrays.asList(goods));

Collections.sort(goodsList); // 使用 Comparable 自然排序
goodsList.sort(null);        // null 表示使用自然排序
```

Arrays.sort() 和 List.sort() 都会直接修改原数组或原集合中的元素顺序。

4. **Comparable 与 equals 的一致性**
   Java 建议自然排序与 `equals()` 保持一致：

$$a.\texttt{compareTo}(b) = 0 \iff a.\texttt{equals}(b) = \texttt{true}$$

   这不是语法上的强制要求，但如果二者不一致，可能导致 TreeSet 和 TreeMap 出现不符合预期的行为。例如，若商品只按照价格比较：

```
Goods a = new Goods("商品 A", 100);
Goods b = new Goods("商品 B", 100);

System.out.println(a.compareTo(b)); // 0: 排序意义上相等
System.out.println(a.equals(b));    // false: 默认比较对象地址
```

   此时，应根据业务需要决定：

   - 是否在 `compareTo()` 中继续比较名称；
   - 是否重写 `equals()` 和 `hashCode()`；
   - 是否接受“价格相同即排序相等”的规则。

   需要注意，BigDecimal 是一个经典例外：

```
BigDecimal a = new BigDecimal("1.0");
BigDecimal b = new BigDecimal("1.00");

System.out.println(a.compareTo(b)); // 0
System.out.println(a.equals(b));    // false
```

5. **Java 内置类的自然排序**
   Java 中很多类已经实现了 Comparable：

类	自然排序规则
String	按 UTF-16 字符值进行字典序比较
Integer	按整数数值大小比较
Double	按浮点数值大小比较，并处理 NaN 等特殊值
Character	按字符对应的 UTF-16 值比较
Date	按时间先后比较，较晚的时间更大
LocalDate	按日期先后比较，较晚的日期更大
BigInteger	按任意精度整数的数值大小比较

表 1.22: 常见类的自然排序规则

6. **Comparable 的局限性** 一个类通常只能定义一套自然排序规则.

如果 Goods 已经按照价格实现了 Comparable<Goods>，就不能再同时定义另一套“按名称自然排序”的 compareTo() 方法。
当一个类需要多种排序方式时，应使用 Comparator。

Comparator：定制排序

Comparator 用于在类的外部定义比较规则，无需修改被比较对象的代码。
它**适用于临时排序、多种排序方式以及无法修改第三方类的情况**。

1. **Comparator 接口** 比较两个独立对象

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

compare(o1, o2) 的返回值规则为：

- 返回负数：o1 小于 o2；
- 返回 0：二者在当前排序规则中相等；
- 返回正数：o1 大于 o2。

Comparator 继承的 equals() 方法**通常不需要重写**。它只有一个用于比较的抽象方法，因此属于函数式接口。

2. **匿名内部类** 按商品名称进行定制排序

```
Arrays.sort(goods, new Comparator<Goods>() {
    @Override
    public int compare(Goods g1, Goods g2) {
        return g1.getName().compareTo(g2.getName()); // 按名称字典序排序
    }
});
```

此时即使 Goods 的自然排序是按价格，本次排序仍会临时改为按名称。

3. **Lambda 表达式** 简化 Comparator 的创建

```
Arrays.sort(goods, (g1, g2) -> {
    return g1.getName().compareTo(g2.getName()); // 按名称排序
});
```

只有一条返回语句时，可以进一步简化：

```
Arrays.sort(goods, (g1, g2) -> g1.getName().compareTo(g2.getName()));
```

也可以先保存比较器：

```
Comparator<Goods> byName = (g1, g2) -> g1.getName().compareTo(g2.getName());

Arrays.sort(goods, byName); // 使用指定比较器排序
```

4. **Comparator 工厂和组合方法**

   Java 8 为 Comparator 提供了大量工厂和组合方法：

方法	作用
comparing(keyExtractor)	根据对象的某个属性创建比较器
comparingInt(keyExtractor)	根据 int 属性创建比较器，避免装箱
comparingLong(keyExtractor)	根据 long 属性创建比较器，避免装箱
comparingDouble(keyExtractor)	根据 double 属性创建比较器，避免装箱
reversed()	反转当前比较器的顺序
thenComparing()	当前比较结果相同时，继续按下一规则比较
naturalOrder()	使用对象的自然排序规则
reverseOrder()	使用自然排序的逆序
nullsFirst(comparator)	将 null 对象排在非空对象之前
nullsLast(comparator)	将 null 对象排在非空对象之后

表 1.23: Comparator 的常用方法

常见使用方式：

```
// 按名称升序
Arrays.sort(goods, Comparator.comparing(Goods::getName));

// 按价格升序，使用 double 专用比较器
Arrays.sort(goods, Comparator.comparingDouble(Goods::getPrice));

// 按价格降序
Arrays.sort(goods, Comparator.comparingDouble(Goods::getPrice).reversed());

// 先按价格升序，价格相同再按名称升序
Arrays.sort(goods, Comparator.comparingDouble(Goods::getPrice)
                             .thenComparing(Goods::getName));

// 使用自然排序
Arrays.sort(goods, Comparator.naturalOrder());

// 使用自然排序的逆序
Arrays.sort(goods, Comparator.reverseOrder());
```

```
19
20 // 将 null 商品排在前面
21 Arrays.sort(goods, Comparator.nullsFirst(Comparator.naturalOrder()));
22
23 // 将 null 商品排在后面
24 Arrays.sort(goods, Comparator.nullsLast(Comparator.naturalOrder()));
```

5. **Comparator 的适用场景**
   - 类没有实现 Comparable；
   - 无法修改第三方类的源代码；
   - 一个类需要多种排序方式；
   - 排序规则需要动态改变；
   - 只在某个局部场景中临时使用比较规则；
   - 不适合将某种比较方式定义为对象的自然顺序。

Comparable 与 Comparator 的区别

特性	Comparable	Comparator
排序类型	自然排序	定制排序
接口位置	由被比较类自身实现	在类的外部单独定义
核心方法	compareTo(T other)	compare(T o1, T o2)
排序规则数量	通常只有一套	可以创建多套不同规则
是否修改原类	需要修改类并实现接口	不需要修改被比较类
使用方式	Arrays.sort(array)	Arrays.sort(array, comparator)
适用场景	类具有明确、固定的自然顺序	临时、多变或多级排序

表 1.24: Comparable 与 Comparator 的区别

比较器的高级技巧

1. **使用泛型** 避免强制类型转换和运行时类型错误
   不推荐使用原始类型：

```
1 Comparator comparator = (o1, o2) -> {
2     Goods g1 = (Goods) o1;
3     Goods g2 = (Goods) o2;
4     return g1.getName().compareTo(g2.getName());
5 };
```

   推荐使用泛型：

```
Comparator<Goods> comparator = (g1, g2) -> g1.getName().compareTo(g2.getName());
```

2. **自定义复杂比较逻辑**

   先按字符串长度排序，长度相同再按字典序排序：

```
Comparator<String> byLengthThenAlphabet = (s1, s2) -> {
    int lengthResult = Integer.compare(s1.length(), s2.length());

    if (lengthResult != 0) {
        return lengthResult; // 长度不同时按长度排序
    }

    return s1.compareTo(s2); // 长度相同时按字典序排序
};
```

   使用工厂方法也可以写成：

```
Comparator<String> byLengthThenAlphabet =
    Comparator.comparingInt(String::length)
              .thenComparing(Comparator.naturalOrder());
```

3. **处理 null 对象**

```
Comparator<String> nullsFirst =
    Comparator.nullsFirst(String::compareTo); // null 排在前面

Comparator<String> nullsLast =
    Comparator.nullsLast(String::compareTo); // null 排在后面
```

   处理对象内部可能为 null 的属性：

```
Comparator<Goods> byNullableName =
    Comparator.comparing(
        Goods::getName,
        Comparator.nullsLast(String::compareTo)
    ); // 名称为 null 时排在最后
```

4. **组合多个比较器**

```
Comparator<Student> comparator =
    Comparator.comparingInt(Student::getAge)
              .thenComparing(
                  Comparator.comparingDouble(Student::getScore)
                            .reversed()
```

```
            )
            .thenComparing(Student::getName);
```

比较顺序为：

(a) 先按年龄升序；

(b) 年龄相同按成绩降序；

(c) 年龄和成绩都相同再按名称升序。

比较器的常见陷阱

1. **不要使用减法直接比较数值**

   不推荐：

```
@Override
public int compareTo(Student other) {
    return this.age - other.age; // 可能发生整数溢出
}
```

   推荐：

```
@Override
public int compareTo(Student other) {
    return Integer.compare(this.age, other.age); // 安全比较整数
}
```

   浮点数不能通过强制转换或减法比较：

```
return (int) (this.price - other.price); // 错误：可能丢失小数
```

   应使用：

```
return Double.compare(this.price, other.price); // 正确处理浮点数
```

2. **保证比较规则具有一致性**

   一个正确的比较器应满足：

   - 自反性：`compare(a, a)` 应返回 `0`；
   - 反对称性：如果 `a > b`，则应有 `b < a`；
   - 传递性：如果 `a > b` 且 `b > c`，则应有 `a > c`；
   - 同一组对象多次比较时，结果应保持一致。

   违反这些规则可能导致排序结果错误，甚至抛出：

   `Comparison method violates its general contract!`

3. **compare 返回 0 不一定表示 equals 为 true**

对于比较器：

```
Comparator<Goods> byPrice =
    Comparator.comparingDouble(Goods::getPrice);

Goods a = new Goods("商品 A", 100);
Goods b = new Goods("商品 B", 100);

System.out.println(byPrice.compare(a, b)); // 0
System.out.println(a.equals(b));           // false
```

在普通数组或列表排序中，这通常只是表示两个元素排序位置相同。
但在 TreeSet 和 TreeMap 中，比较结果为 0 会被视为相同键或相同元素：

```
TreeSet<Goods> set = new TreeSet<>(
    Comparator.comparingDouble(Goods::getPrice)
);

set.add(new Goods("商品 A", 100));
set.add(new Goods("商品 B", 100));

System.out.println(set.size()); // 1
```

如果希望保留两个商品，应加入第二级比较：

```
Comparator<Goods> byPriceThenName =
    Comparator.comparingDouble(Goods::getPrice)
              .thenComparing(Goods::getName);
```

4. **避免在比较方法中执行复杂操作**
   排序过程中，比较器可能被调用很多次。
   不应在 compare() 中执行：
   - 数据库查询；
   - 文件读写；
   - 网络请求；
   - 复杂且重复的计算；
   - 会修改对象状态的操作。

   如果比较键计算成本很高，应提前计算并保存结果。
5. **不要随意修改排序字段**
   对象加入 TreeSet 或作为 TreeMap 的键以后，不应再修改参与比较的字段：

```
TreeSet<Student> students = new TreeSet<>();
Student student = new Student("张三", 20, 90);
students.add(student);
// 如果之后将 age 改为其他值，TreeSet 内部顺序可能被破坏
```

参与排序和索引的字段最好保持不可变。

1.5.6 工具类

Java 提供了很多已经封装好的类，用于完成系统操作、数学计算和高精度数值运算。
本节主要介绍：

- `System`：系统输入输出、时间、属性和程序控制；
- `Math`：常见数学运算；
- `BigInteger`：任意精度整数运算；
- `BigDecimal`：任意精度十进制小数运算。

其中，`System` 和 `Math` 是典型的工具类，主要通过静态属性和静态方法使用，不需要创建对象。`BigInteger` 和 `BigDecimal` 则是不可变的数值类，需要创建对象后调用方法进行运算。

System

`System` 位于 `java.lang` 包中，用于访问系统输入输出流、系统时间、系统属性和 JVM 运行环境。其构造器是私有的，因此不能创建 `System` 对象，只能通过类名调用静态成员。

1. **标准输入输出流** 进行控制台输入、正常输出和错误输出
   `System` 提供了三个标准流：

属性	类型	作用
`System.in`	`InputStream`	标准输入流，默认从键盘读取数据
`System.out`	`PrintStream`	标准输出流，输出普通信息
`System.err`	`PrintStream`	标准错误流，输出错误或警告信息

表 1.25: System 的标准流

```
System.out.println("普通输出"); // 输出并换行
System.out.print("不换行输出"); // 输出但不换行

System.err.println("错误信息"); // 输出到标准错误流
```

不同终端或 IDE 可能用不同颜色显示 `System.err`，显示颜色和顺序并不是 Java 语言保证的。
使用 `System.in` 读取一个字节：

```
try {
    System.out.println("请输入一个字符：");// 使用System.out输出信息
    int data = System.in.read(); // 使用System.in读取输入，读取一个字节
    System.out.println("输入的字符：" + (char) data);
} catch (IOException e) {
    System.err.println("读取失败：" + e.getMessage()); //
    使用System.err输出错误信息
}
```

通常也可以使用 Scanner 包装 System.in：

```
Scanner scanner = new Scanner(System.in);
System.out.print("请输入姓名：");
String name = scanner.nextLine();
System.out.println("你好，" + name);
```

2. **currentTimeMillis()** 获取当前毫秒时间戳

   System.currentTimeMillis() 返回当前时刻与 Unix 纪元之间相差的毫秒数。

   Unix 纪元起点为：1970-01-01T00:00:00Z

```
long timestamp = System.currentTimeMillis(); // 获取当前毫秒时间戳
System.out.println(timestamp);
```

   将时间戳转换为 Instant：

```
long timestamp = System.currentTimeMillis();
Instant instant = Instant.ofEpochMilli(timestamp); // 毫秒时间戳 -> Instant
System.out.println(instant);
```

3. **nanoTime()** 测量程序运行经过的时间

   currentTimeMillis() 表示现实世界中的系统时间，系统时间可能被人工或操作系统调整。

   测量一段代码运行了多久时，应优先使用 System.nanoTime()：

```
long start = System.nanoTime(); // 记录开始时间

for (int i = 0; i < 10_000_000; i++) {
    Math.sqrt(i);
}

long end = System.nanoTime(); // 记录结束时间
long elapsedNanos = end - start;
double elapsedMillis = elapsedNanos / 1_000_000.0;
System.out.println("运行时间：" + elapsedMillis + " 毫秒");
```

   nanoTime() 的返回值本身没有日期含义，只能通过两次返回值相减计算经过时间。

4. **exit(int status)** 强制终止当前 JVM

```
System.out.println("程序开始");
boolean fatalError = true;

if (fatalError) {
    System.err.println("发生严重错误，程序终止");
```

```
    System.exit(1); // 非零状态码通常表示异常退出
}
System.out.println("Test."); // 这行代码不会执行
```

状态码通常约定为：

- 0：正常退出；
- 非 0：异常退出。

System.exit() 会终止整个 JVM，不应将它当作普通方法的 return 使用。
在服务器、Web 应用和公共类库中，通常不应随意调用 System.exit()。

5. **gc() 请求 JVM 执行垃圾回收**

```
for (int i = 0; i < 10_000; i++) {
    new Object(); // 临时对象之后可能成为垃圾
}

System.gc(); // 建议 JVM 执行垃圾回收
```

System.gc() 只是向 JVM 发出垃圾回收建议，JVM 可立即执行、稍后执行，也可忽略该请求。
正常程序不应频繁手动调用 System.gc()，垃圾回收通常应交给 JVM 自动管理。

6. **getProperty() 获取系统属性**

```
String javaVersion = System.getProperty("java.version"); // Java 版本
String javaHome = System.getProperty("java.home");       // Java 安装目录
String osName = System.getProperty("os.name");           // 操作系统名称
String osVersion = System.getProperty("os.version");     // 操作系统版本
String userName = System.getProperty("user.name");       // 当前用户名称
String userHome = System.getProperty("user.home");       // 用户主目录
String userDir = System.getProperty("user.dir");         // 当前工作目录
```

常见系统属性如下：
获取全部系统属性：

```
Properties properties = System.getProperties();
properties.list(System.out); // 输出全部系统属性
```

获取环境变量：

```
String path = System.getenv("PATH"); // 获取 PATH 环境变量
Map<String, String> environment = System.getenv(); // 获取全部环境变量
```

系统属性与环境变量不是同一个概念：

属性键	含义
java.version	当前 Java 版本
java.home	Java 运行环境安装目录
os.name	操作系统名称
os.version	操作系统版本
user.name	当前用户名称
user.home	当前用户主目录
user.dir	当前程序工作目录
file.separator	当前系统的文件路径分隔符
line.separator	当前系统的换行符

表 1.26: 常见 Java 系统属性

- 系统属性通过 System.getProperty() 获取；
- 环境变量通过 System.getenv() 获取。

7. **arraycopy()** 高效复制数组内容（数组讲过）

   方法结构为：

```
System.arraycopy(
    source, sourcePosition,
    destination, destinationPosition,
    length
);
```

   例如：

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

// 将 source[1] 开始的 3 个元素复制到 destination[0]
System.arraycopy(source, 1, destination, 0, 3);
System.out.println(Arrays.toString(destination)); // [2, 3, 4, 0, 0]
```

   arraycopy() 可以在同一个数组中复制重叠区间：

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

System.arraycopy(numbers, 0, numbers, 1, 4);
System.out.println(Arrays.toString(numbers)); // [1, 1, 2, 3, 4]
```

   复制对象数组时，复制的是对象引用，不会自动创建对象副本。

Math

Math 位于 java.lang 包中，提供常见数学运算的静态方法。

不需要创建 Math 对象：

```
double result = Math.sqrt(16);
```

常用数学常量包括：

```
double pi = Math.PI; // 圆周率
double e = Math.E;   // 自然对数底数
```

1. **绝对值、最大值和最小值** 完成基础数值运算

   ```
   double absolute = Math.abs(-12.5); // 12.5

   double max = Math.max(3.8, 5.0); // 5.0
   double min = Math.min(3.8, 5.0); // 3.8

   int intMax = Math.max(100, 200);     // 200
   long longMin = Math.min(1000L, 500L); // 500
   ```

   abs()、max() 和 min() 均提供多种基本数值类型的重载方法。

2. **幂、开方、指数和对数**

   ```
   double squareRoot = Math.sqrt(16); // 4.0
   double cubeRoot = Math.cbrt(27);   // 3.0

   double power = Math.pow(2, 10); // 1024.0

   double exponential = Math.exp(1); // e 的 1 次幂
   double naturalLog = Math.log(2);  // ln(2)
   double commonLog = Math.log10(100); // 2.0
   ```

   常用方法包括：

3. 三**角函数** 参数和返回值通常使用弧度

   ```
   double degrees = 45.0;
   double radians = Math.toRadians(degrees); // 角度 -> 弧度

   double sinValue = Math.sin(radians); // 正弦
   double cosValue = Math.cos(radians); // 余弦
   double tanValue = Math.tan(radians); // 正切
   ```

方法	作用
`sqrt(a)`	计算平方根
`cbrt(a)`	计算立方根
`pow(a, b)`	计算 a^b
`exp(a)`	计算 e^a
`log(a)`	计算自然对数 $\ln a$
`log10(a)`	计算以 10 为底的对数

表 1.27: `Math` 的幂和对数方法

反三角函数：

```
double asin = Math.asin(sinValue); // 返回弧度
double acos = Math.acos(cosValue);
double atan = Math.atan(tanValue);

double resultDegrees = Math.toDegrees(asin); // 弧度 -> 角度
```

计算圆的周长和面积：

```
double radius = 5.0;

double circumference = 2 * Math.PI * radius; // 2r
double area = Math.PI * Math.pow(radius, 2);  // r²
```

4. **取整方法**

```
double number = 3.14;
double ceiling = Math.ceil(number); // 4.0: 向正无穷方向取整
double floor = Math.floor(number);  // 3.0: 向负无穷方向取整
long rounded = Math.round(number);  // 3: 四舍五入
```

对于负数：

```
double number = -3.14;
System.out.println(Math.ceil(number));  // -3.0
System.out.println(Math.floor(number)); // -4.0
System.out.println(Math.round(number)); // -3
```

需要注意：

- `ceil()` 返回大于或等于参数的最小整数值；
- `floor()` 返回小于或等于参数的最大整数值；
- `Math.round(double)` 返回 `long` `Math.round(float)` 返回 `int`。

5. **随机数** 生成指定范围的伪随机数

   `Math.random()` 返回：$[0.0, 1.0)$

   范围内的 `double`：

```java
double random = Math.random(); // [0.0, 1.0)
```

   生成 `0--9` 的随机整数：

```java
int number = (int) (Math.random() * 10); // [0, 9]
```

   生成 `1--10` 的随机整数：

```java
int number = (int) (Math.random() * 10) + 1; // [1, 10]
```

   生成闭区间 $[min, max]$ 中的随机整数：

```java
public static int randomInt(int min, int max) {
    if (min > max) {
        throw new IllegalArgumentException("min 不能大于 max");
    }

    return (int) (Math.random() * (max - min + 1)) + min;
}
```

   `Math.random()` 不适合生成密码、验证码密钥或安全令牌。安全相关随机数应使用 `SecureRandom`。

```java
SecureRandom random = new SecureRandom();
int code = random.nextInt(1_000_000); // [0, 999999]
```

6. **精确整数运算** 检测整数溢出

   普通整数运算发生溢出时不会自动抛出异常：

```java
int value = Integer.MAX_VALUE;
System.out.println(value + 1); // -2147483648: 发生溢出
```

   可以使用 `Math` 的精确运算方法：

```java
int sum = Math.addExact(100, 200);          // 300
int difference = Math.subtractExact(10, 3); // 7
int product = Math.multiplyExact(20, 30);   // 600
int next = Math.incrementExact(100);        // 101
int previous = Math.decrementExact(100);    // 99
```

   发生溢出时会抛出 `ArithmeticException`：

```java
int value = Math.addExact(Integer.MAX_VALUE, 1); // ArithmeticException
```

将 long 精确转换为 int：

```java
long value = 1000L;
int result = Math.toIntExact(value); // 1000
```

BigInteger

Java 的 int 和 long 都有固定的取值范围。
当整数超过 long 的范围时，可以使用 BigInteger 表示任意精度整数。
BigInteger 位于 java.math 包中：

```java
import java.math.BigInteger;
```

BigInteger 是不可变类，所有运算方法都会返回新的对象。

1. **创建 BigInteger**

```java
BigInteger big1 = new BigInteger("12345678901234567890"); // 字符串构造
BigInteger big2 = new BigInteger("-98765432109876543210");

BigInteger fromLong = BigInteger.valueOf(1234567890L); // long -> BigInteger

BigInteger zero = BigInteger.ZERO; // 0
BigInteger one = BigInteger.ONE;   // 1
BigInteger ten = BigInteger.TEN;   // 10
```

按照指定进制解析字符串：

```java
BigInteger binary = new BigInteger("1010", 2); // 二进制 1010 -> 10
BigInteger hex = new BigInteger("FF", 16);     // 十六进制 FF -> 255
```

大整数通常应使用字符串构造，因为数值字面量本身可能已经超出基本类型范围：

```java
BigInteger number = new BigInteger("999999999999999999999999999999999999");
```

2. **算术运算** 通过方法调用代替运算符

```java
BigInteger a = new BigInteger("12345678901234567890");
BigInteger b = new BigInteger("98765432109876543210");

BigInteger sum = a.add(b);            // 加法
BigInteger difference = a.subtract(b); // 减法
BigInteger product = a.multiply(b);  // 乘法
BigInteger quotient = a.divide(b);   // 整数除法
```

```
BigInteger remainder = a.remainder(b); // 取余

BigInteger[] result = a.divideAndRemainder(b);
BigInteger quotient2 = result[0];  // 商
BigInteger remainder2 = result[1]; // 余数
```

运算	方法	示例
加法	add(BigInteger value)	a.add(b)
减法	subtract(BigInteger value)	a.subtract(b)
乘法	multiply(BigInteger value)	a.multiply(b)
除法	divide(BigInteger value)	a.divide(b)
取余	remainder(BigInteger value)	a.remainder(b)
商和余数	divideAndRemainder(BigInteger value)	返回数组 [商，余数]
幂运算	pow(int exponent)	a.pow(10)

表 1.28: BigInteger 的算术运算

BigInteger 不能直接使用普通算术运算符：

```
// BigInteger result = a + b; // 编译错误
BigInteger result = a.add(b); // 正确
```

3. **remainder() 与 mod()**

remainder() 的结果符号**可能与被除数相同**：

```
BigInteger a = BigInteger.valueOf(-17);
BigInteger b = BigInteger.valueOf(5);
System.out.println(a.remainder(b)); // -2
```

mod() 主要用于模运算，要求**模数必须为正数**，并返回非负结果：

```
System.out.println(a.mod(b)); // 3
```

4. **比较和绝对值**

```
BigInteger a = new BigInteger("1234567890");
BigInteger b = new BigInteger("9876543210");
BigInteger negative = new BigInteger("-1234567890");

System.out.println(a.compareTo(b)); // 负数: a < b
System.out.println(b.compareTo(a)); // 正数: b > a
System.out.println(a.compareTo(a)); // 0
System.out.println(a.equals(b)); // false
```

```
9
10 BigInteger absolute = negative.abs(); // 1234567890
11 BigInteger max = a.max(b);            // 返回较大值
12 BigInteger min = a.min(b);            // 返回较小值
```

和其他 compareTo() 方法一样，比较时只应判断返回值的正负，不应依赖它一定返回 -1 或 1。

5. **最大公约数和符号**

```
1 BigInteger base = new BigInteger("12");
2 BigInteger a = BigInteger.valueOf(48);
3 BigInteger b = BigInteger.valueOf(18);
4
5 BigInteger gcd = a.gcd(b); // 最大公约数：6
6
7 int sign = base.signum(); // 正数返回 1，0 返回 0，负数返回 -1
```

6. **转换为基本类型**

```
1 BigInteger number = BigInteger.valueOf(1000);
2
3 int intValue = number.intValue();
4 long longValue = number.longValue();
5 double doubleValue = number.doubleValue();
```

普通转换方法可能发生截断。

需要保证数值能够完整放入目标类型时，应使用精确转换：

```
1 int exactInt = number.intValueExact();
2 long exactLong = number.longValueExact();
```

超出目标类型范围时会抛出 ArithmeticException。

BigDecimal

float 和 double 使用二进制浮点数表示数据，无法精确表示部分十进制小数：

```
1 double a = 0.1;
2 double b = 0.2;
3 System.out.println(a + b); // 0.30000000000000004
```

金融、统计和其他需要精确十进制计算的场景，应使用 BigDecimal。BigDecimal 位于 java.math 包中：

```
1 import java.math.BigDecimal;
2 import java.math.RoundingMode;
```

BigDecimal 同样是不可变类，运算方法不会修改原对象，而是返回新的对象。

1. **创建 BigDecimal 优先使用字符串或 valueOf()**

   推荐方式：

```
BigDecimal price = new BigDecimal("19.99"); // 推荐：字符串构造
BigDecimal rate = BigDecimal.valueOf(0.035); // 推荐：valueOf()

BigDecimal count = new BigDecimal(100);      // 从 int 创建
BigDecimal total = BigDecimal.valueOf(100L); // 从 long 创建

BigDecimal zero = BigDecimal.ZERO;
BigDecimal one = BigDecimal.ONE;
BigDecimal ten = BigDecimal.TEN;
```

   不推荐直接使用 double 构造：

```
BigDecimal bad = new BigDecimal(0.1);
System.out.println(bad); // 并不是精确的 0.1，而是保存了 double 0.1 的实际二进制值
```

   正确写法：

```
BigDecimal good1 = new BigDecimal("0.1");
BigDecimal good2 = BigDecimal.valueOf(0.1);
System.out.println(good1); // 0.1
System.out.println(good2); // 0.1
```

   对于十进制小数字面量，应优先使用字符串构造或 BigDecimal.valueOf()，不要直接使用 new BigDecimal(double)。

2. **算术运算**

运算	方法	示例
加法	add(BigDecimal value)	a.add(b)
减法	subtract(BigDecimal value)	a.subtract(b)
乘法	multiply(BigDecimal value)	a.multiply(b)
除法	divide(divisor, scale, roundingMode)	a.divide(b, 2, HALF_UP)
幂运算	pow(int exponent)	a.pow(3)

表 1.29: BigDecimal 的算术运算

```
BigDecimal a = new BigDecimal("10.0");
BigDecimal b = new BigDecimal("3.0");

```

```
BigDecimal sum = a.add(b);        // 13.0
BigDecimal difference = a.subtract(b); // 7.0
BigDecimal product = a.multiply(b);     // 30.00
BigDecimal power = a.pow(3);            // 1000.000
```

原对象不会发生改变：

```
BigDecimal value = new BigDecimal("10");
value.add(BigDecimal.ONE);
System.out.println(value); // 10
```

必须接收返回值：

```
value = value.add(BigDecimal.ONE);
System.out.println(value); // 11
```

3. **除法** 无限循环小数必须指定舍入规则
   能够得到有限小数时，可以直接除：

```
BigDecimal result = new BigDecimal("1").divide(new BigDecimal("4"));
System.out.println(result); // 0.25
```

无法得到有限小数时，直接除会抛出异常：

```
BigDecimal r = new BigDecimal("1").divide(new BigDecimal("3"));
// ArithmeticException
```

应指定小数位数和舍入模式：
常见舍入模式如下：

舍入模式	含义	示例
RoundingMode.HALF_UP	最接近值；刚好一半时远离零	$2.5 \to 3$，$-2.5 \to -3$
RoundingMode.HALF_EVEN	最接近值；刚好一半时取偶数	$2.5 \to 2$，$3.5 \to 4$
RoundingMode.UP	远离零方向舍入	$2.1 \to 3$，$-2.1 \to -3$
RoundingMode.DOWN	向零方向截断	$2.9 \to 2$，$-2.9 \to -2$
RoundingMode.CEILING	向正无穷方向舍入	$2.1 \to 3$，$-2.1 \to -2$
RoundingMode.FLOOR	向负无穷方向舍入	$2.9 \to 2$，$-2.1 \to -3$
RoundingMode.UNNECESSARY	不允许发生舍入，需要舍入时抛出异常	精确结果才能成功

表 1.30: BigDecimal 的常用舍入模式

```
BigDecimal result = new BigDecimal("1").divide(new BigDecimal("3"), 2, // 保留2位
                                               RoundingMode.HALF_UP); // 四舍五入
System.out.println(result); // 0.33
```

其中：

- 2 表示结果保留两位小数；
- HALF_UP 表示通常所说的四舍五入。

4. **setScale() 设置小数位数**

```
BigDecimal value = new BigDecimal("3.14159");
BigDecimal result = value.setScale(2, RoundingMode.HALF_UP);
System.out.println(result); // 3.14
```

若缩小小数位数时会丢失数据，但没有提供舍入模式，则可能抛出异常：

```
BigDecimal value = new BigDecimal("3.14159");
BigDecimal result = value.setScale(2); // ArithmeticException
```

5. **compareTo() 与 equals() 比较规则不同**

compareTo() 比较数值大小，不考虑末尾零导致的 scale 差异：

```
BigDecimal a = new BigDecimal("2.0");
BigDecimal b = new BigDecimal("2.00");

System.out.println(a.compareTo(b)); // 0: 数值相等
```

equals() 同时比较数值和 scale：

```
System.out.println(a.equals(b)); // false
```

因此，判断金额数值是否相等时，通常使用：

```
boolean equal = a.compareTo(b) == 0; // 而不是 a.equals(b);
```

new BigDecimal("2.0") 和 new BigDecimal("2.00") 数值上相等，但通过 equals() 不相等。

6. **precision 与 scale**

```
BigDecimal number = new BigDecimal("123.4500");
System.out.println(number.precision()); // 7: 有效数字总数
System.out.println(number.scale());     // 4: 小数位数
```

去除末尾多余的零：

```
BigDecimal normalized = number.stripTrailingZeros();
System.out.println(normalized); // 123.45
```

7. **MathContext 控制整体有效数字和舍入规则**

```
// 设置整体舍入规则：保留 4 位有效数字
```

```
MathContext context = new MathContext(4, RoundingMode.HALF_UP);

BigDecimal a = new BigDecimal("10");
BigDecimal b = new BigDecimal("3");
BigDecimal result = a.divide(b, context);
System.out.println(result); // 3.333
```

MathContext 中的精度表示有效数字总数，不是固定的小数位数。

BigInteger 与 BigDecimal 的区别

特性	BigInteger	BigDecimal
数据类型	任意精度整数	任意精度十进制小数
是否不可变	是	是
典型构造	字符串、valueOf(long)	字符串、valueOf(double)
除法	整数除法	可以保留小数，可能需要舍入模式
主要场景	阶乘、密码学、大整数计算	金额、利率和精确十进制计算
主要注意事项	不能直接使用普通运算符	避免使用 double 构造；注意 scale 和舍入规则

表 1.31: BigInteger 与 BigDecimal 的区别

四类工具的选择

类	核心功能	常见场景
System	系统输入输出、时间、属性和 JVM 控制	控制台 IO、时间戳、系统信息、数组复制
Math	常见数学运算	开方、幂、三角函数、取整和随机数
BigInteger	任意精度整数	大整数、阶乘、密码学和科学计算
BigDecimal	任意精度十进制小数	金额、利率、税率和精确小数计算

表 1.32: 常用工具类的选择

1.6 Java 异常处理

错题集

1.6.1 异常概述

异常（Exception）是程序运行过程中出现的不正常情况，例如访问不存在的数组位置、对空引用调用方法、读取不存在的文件或数据库连接失败。
异常不等于语法错误或普通逻辑错误：

- 语法错误：代码不符合 Java 语法，无法通过编译；
- 逻辑错误：程序可以运行，但结果不符合预期；
- 异常：程序运行过程中出现特殊情况，破坏正常执行流程。

当异常发生时，JVM 会创建对应的异常对象，其中通常包含异常类型、错误信息和方法调用栈。

异常体系结构

Java 中所有可以被抛出和捕获的对象都继承自 `Throwable`。异常体系的基本结构为：

```
Throwable
  Error
  Exception
      RuntimeException
      其他 Exception
```

1. **Throwable** 所有错误和异常的共同父类，`Throwable` 的两个主要子类为：
   - `Error`
   - `Exception`

   常用方法包括：

```java
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage()); // / by zero
    System.out.println(e.toString());   // 异常类型与错误信息
    e.printStackTrace();                // 输出完整异常调用栈
}
```

2. **Error** 表示严重的系统或 JVM 错误
   常见的 `Error` 包括：
   - `OutOfMemoryError`：JVM 无法继续分配所需内存；
   - `StackOverflowError`：方法调用栈溢出；
   - `NoClassDefFoundError`：运行时无法找到需要的类定义。

   例如，无限递归可能导致栈溢出：

```
public static void recursive() {
    recursive(); // 无限递归，最终可能抛出 StackOverflowError
}
```

Error 通常表示应用程序难以恢复的严重问题，一般不应通过普通业务代码捕获并继续运行。

3. **Exception** 表示程序**可以处理**或传播的异常情况

   Exception 主要分为：

   - 运行时异常：RuntimeException 及其子类；
   - 受检异常：除 RuntimeException 以外的 Exception 子类。

   常见运行时异常：

   ```
   NullPointerException
   ArrayIndexOutOfBoundsException
   ArithmeticException
   ClassCastException
   NumberFormatException
   ```

   常见受检异常：

   ```
   IOException
   SQLException
   ClassNotFoundException
   ```

   Java 根据编译器是否强制要求处理，将可抛出的对象分为受检异常和非受检异常。

类别	判断标准	常见代表
受检异常	Exception 的子类，但不是 RuntimeException 的子类	IOException、SQLException、ClassNotFoundException
非受检异常	RuntimeException 及其子类，以及所有 Error	NullPointerException、ArithmeticException、OutOfMemoryError

表 1.33: 受检异常与非受检异常

   (a) **受检异常** 编译器强制要求捕获或声明

   以下代码不能直接通过编译：

   ```
   FileInputStream input = new FileInputStream("test.txt"); //
       FileNotFoundException 是受检异常，必须处理
   ```

   可以使用 try-catch 捕获：

```java
try {
    FileInputStream input =
        new FileInputStream("test.txt");
} catch (FileNotFoundException e) {
    System.out.println("文件不存在：" + e.getMessage());
}
```

也可以通过 throws 声明：

```java
public static void readFile()
        throws FileNotFoundException {

    FileInputStream input =
        new FileInputStream("test.txt");
}
```

(b) **非受检异常** 编译器不强制要求处理

```java
int[] numbers = {1, 2, 3};
System.out.println(numbers[5]); // 编译可以通过，运行时抛出数组越界异常
```

运行时异常通常反映程序中的代码缺陷，更应该通过参数检查和程序逻辑提前避免，而不是对所有代码都机械地添加 try-catch。“编译时异常”是受检异常的习惯说法，并不是异常在编译阶段发生，而是编译器在编译阶段强制检查是否已经处理。

常见运行时异常

1. **NullPointerException** 对空引用调用实例成员

```java
String text = null;
System.out.println(text.length()); // java.lang.NullPointerException
```

常见原因包括：

- 对 null 调用方法；
- 访问 null 对象的成员变量；
- 对 null 包装类进行自动拆箱；
- 访问尚未初始化的引用变量。

避免方式：

```java
if (text != null) {
    System.out.println(text.length());
}
```

比较固定字符串时，可以将常量写在前面：

```
if ("yes".equals(text)) {
    System.out.println("确认");
}
```

2. **ArrayIndexOutOfBoundsException** 数组索引越界

```
int[] numbers = {10, 20, 30};
System.out.println(numbers[3]); // java.lang.ArrayIndexOutOfBoundsException
```

数组的有效索引范围为：$0 \leq i <$ `array.length`

避免方式：

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

也可以使用增强型 `for` 循环：

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

3. **ArithmeticException** 非法整数算术运算

最典型的情况是整数除以零：

```
int result = 10 / 0; // java.lang.ArithmeticException: / by zero
```

避免方式：

```
int divisor = 0;
if (divisor != 0) {
    int result = 10 / divisor;
}
```

需要注意，浮点数除以零通常不会抛出 `ArithmeticException`：

```
System.out.println(10.0 / 0.0); // Infinity
System.out.println(0.0 / 0.0);  // NaN
```

4. **ClassCastException** 强制转换为不兼容类型

```
Object object = "Hello";
Integer number = (Integer) object; // java.lang.ClassCastException
```

转换前可以使用 `instanceof` 判断：

```java
if (object instanceof Integer) {
    Integer number = (Integer) object;
}
```

使用模式匹配可以同时判断和转换：

```java
if (object instanceof String text) {
    System.out.println(text.length());
}
```

5. **NumberFormatException** 字符串不能转换为目标数字

```java
String text = "123abc";
int number = Integer.parseInt(text); // java.lang.NumberFormatException
```

合法示例：

```java
int number = Integer.parseInt("123");       // 123
double price = Double.parseDouble("19.99"); // 19.99
```

可以捕获转换失败：

```java
try {
    int number = Integer.parseInt(text);
} catch (NumberFormatException e) {
    System.out.println("不是合法整数：" + text);
}
```

6. **InputMismatchException** Scanner 输入类型不匹配

```java
Scanner scanner = new Scanner(System.in);
System.out.print("请输入整数：");
int number = scanner.nextInt(); // 输入 abc 时抛出 InputMismatchException
```

可以先判断输入类型：

```java
if (scanner.hasNextInt()) {
    int number = scanner.nextInt();
    System.out.println(number);
} else {
    System.out.println("输入的不是整数");
    scanner.next(); // 清除错误输入
}
```

7. **IllegalArgumentException** 方法参数不合法

```
public static void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("年龄不能小于 0");
    }
}
```

NumberFormatException 本身也是 IllegalArgumentException 的子类。

8. **IllegalStateException** 对象当前状态不允许执行操作

```
public void start() {
    if (started) {
        throw new IllegalStateException("任务已经启动");
    }
    started = true;
}
```

常见受检异常

主要受检异常包括文件、IO、数据库和动态类加载异常。

1. **FileNotFoundException** 文件不存在或无法打开

```
try {
    FileInputStream input = new FileInputStream("missing.txt");
} catch (FileNotFoundException e) {
    System.out.println("文件未找到：" + e.getMessage());
}
```

FileNotFoundException 是 IOException 的子类。

它不仅可能表示文件不存在，也可能表示路径是目录、权限不足或文件无法打开。

2. **IOException** 输入输出操作失败

```
try (FileInputStream input = new FileInputStream("test.txt")) {
    int data;
    while ((data = input.read()) != -1) {
        System.out.print((char) data);
    }
} catch (IOException e) {
    System.out.println("读取失败：" + e.getMessage());
}
```

使用 try-with-resources 可以在代码块结束后自动关闭资源。

3. **SQLException** 数据库操作失败

```
try {
    Connection connection = DriverManager.getConnection(url, username, password);
    Statement statement = connection.createStatement();
    statement.executeQuery(
        "SELECT * FROM wrong_table"
    );
} catch (SQLException e) {
    System.out.println("数据库错误：" + e.getMessage());
}
```

常见原因包括：

- 数据库连接失败；
- SQL 语句错误；
- 数据表或字段不存在；
- 权限不足；
- 违反数据库约束。

4. **ClassNotFoundException** 动态加载类时找不到类

```
try {
    Class.forName(
        "com.example.NotFoundClass"
    );
} catch (ClassNotFoundException e) {
    System.out.println(
        "类未找到：" + e.getMessage()
    );
}
```

它通常发生在使用反射或动态类加载时，与 `NoClassDefFoundError` 不完全相同。

异常的传播过程

当某个方法发生异常且没有在当前方法中捕获时，异常会沿着方法调用栈向上传播。

```
public static void methodC() {
    int result = 10 / 0; // 异常从这里产生
}

public static void methodB() {
    methodC(); // 异常继续向上传播
```

```
}

public static void methodA() {
    methodB(); // 异常继续向上传播
}

public static void main(String[] args) {
    methodA(); // 如果仍未捕获，程序终止并打印堆栈
}
```

堆栈信息通常会显示：

- 异常类型；
- 异常消息；
- 异常发生的类名；
- 异常发生的方法名；
- 对应的源代码行号；
- 方法之间的调用顺序。

阅读堆栈时，通常先寻找第一条属于自己项目代码的 `at ...` 信息。

异常处理的基本方式

Java 处理异常主要有两种方式：

- `try-catch-finally`：在当前位置捕获并处理异常；
- `throws`：当前方法不处理，交给调用者继续处理。

try-catch-finally 的标准结构

1. **try-catch-finally 的标准结构**
   `try` 块用于放置可能发生异常的代码；`catch` 块用于捕获并处理指定类型的异常；`finally` 块用于执行资源释放等善后代码。

```
try {
    // 可能抛出异常的代码
} catch (ExceptionType1 e) {
    // 处理 ExceptionType1 类型的异常
} catch (ExceptionType2 e) {
    // 处理 ExceptionType2 类型的异常
} finally {
    // 无论是否发生异常，通常都会执行
}
```

执行流程可以理解为：

- 如果 try 中没有异常，则跳过 catch；
- 如果 try 中发生异常，则寻找匹配的 catch；
- 如果存在 finally，通常最后都会执行；
- 如果异常没有被当前方法捕获，则继续向调用者传播。

(a) **try-catch** 在当前位置捕获并处理异常

```
try {
    int number = Integer.parseInt("abc"); // 可能抛出 NumberFormatException
} catch (NumberFormatException e) {
    System.out.println("数字格式不正确");
    System.out.println(e.getMessage()); // For input string: "abc"
}
```

捕获异常后，程序不会因为该异常直接终止，而是可以继续执行后续代码：

```
try {
    int result = 10 / 0; // 可能抛出 ArithmeticException
    System.out.println(result);
} catch (ArithmeticException e) {
    System.out.println("除数不能为 0");
}

System.out.println("程序继续执行");
```

(b) **多重 catch** 针对不同异常分别处理

当一段代码可能抛出多种异常时，可以写多个 catch 块。

```
try {
    int[] nums = {1, 2, 3};
    int result = 10 / 0; // 可能抛出 ArithmeticException
    int value = nums[3]; // 可能抛出 ArrayIndexOutOfBoundsException

    System.out.println(result + ", " + value);
} catch (ArithmeticException e) {
    System.out.println("处理算术异常: " + e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("处理数组越界: " + e.getMessage());
} catch (Exception e) {
    System.out.println("处理其他异常: " + e.getMessage());
```

```
}
```

需要注意：

- 多个 catch 只会执行其中第一个匹配的分支；
- 子类异常应写在父类异常前面；
- 如果先捕获 Exception，后面的具体异常将无法到达。

错误写法：

```
try {
    int result = 10 / 0;
} catch (Exception e) {
    System.out.println("通用异常");
} catch (ArithmeticException e) {
    System.out.println("算术异常"); // 编译错误：无法到达
}
```

(c) **finally** 用于执行资源释放等善后操作

finally 通常用于关闭文件流、释放数据库连接、释放锁等操作。

```
FileInputStream input = null;

try {
    input = new FileInputStream("test.txt");
    int data = input.read(); // 读取文件
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (input != null) {
        try {
            input.close(); // 释放资源
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

finally 的特点：

- 没有异常时会执行；
- 异常被 catch 捕获后会执行；
- 即使 try 或 catch 中有 return，也会先执行；

- 但如果 JVM 被强制终止，例如 System.exit()，则不一定执行。

finally 与 return 的执行顺序：如果 try 或 catch 中有 return，Java 会先保存返回值，再执行 finally，最后返回之前保存的结果。

```
public static int testReturn() {
    int result = 10;

    try {
        result = 20;
        return result; // 先保存返回值 20
    } finally {
        result = 50; // 修改局部变量，但不影响已保存的返回值
    }
}

System.out.println(testReturn()); // 20
```

如果 catch 中有 return，也会先执行 finally：

```
public static int testReturn() {
    int result = 10;

    try {
        result = 20;
        int value = 10 / 0; // 触发异常
        return result;
    } catch (Exception e) {
        return 30; // 先保存返回值 30
    } finally {
        result = 50; // 不影响已经保存的 30
    }
}

System.out.println(testReturn()); // 30
```

强烈不建议在 finally 中写 return，因为它会覆盖 try 或 catch 中的返回值，甚至可能吞掉原本要抛出的异常。

不推荐：

```
public static int testReturn() {
    try {
```

```
        return 20;
    } finally {
        return 50; // 覆盖 try 中的 return
    }
}

System.out.println(testReturn()); // 50
```

2. **捕获最具体的异常类型，且不要使用空 catch 块**

   不推荐：

```
try {
    int number = Integer.parseInt(text);
} catch (Exception e) {
    System.out.println("发生错误");
}
```

   推荐：

```
try {
    int number = Integer.parseInt(text);
} catch (NumberFormatException e) {
    System.out.println("不是合法整数：" + text);
}
```

   空的 catch 会隐藏问题，使程序表现异常但没有任何错误线索。

   错误写法：

```
try {
    readFile();
} catch (IOException e) {
    // 什么都不做
}
```

   至少应记录或重新抛出异常：

```
try {
    readFile();
} catch (IOException e) {
    e.printStackTrace(); // 打印异常调用栈
}
```

   实际项目中通常使用日志框架记录异常对象，而不是简单丢弃异常。

3. **try-with-resources** 自动关闭资源

   从 Java 7 开始，可以使用 try-with-resources 自动关闭实现了 AutoCloseable 的资源。

```
try (FileInputStream input = new FileInputStream("test.txt")) {
    int data = input.read(); // 使用资源
} catch (IOException e) {
    e.printStackTrace();
}
```

   代码块执行结束后，input 会被自动关闭，不需要手动写 finally。

   多个资源也可以一起声明：

```
try (
    FileInputStream input = new FileInputStream("input.txt");
    FileOutputStream output = new FileOutputStream("output.txt")
) {
    int data = input.read();
    output.write(data);
} catch (IOException e) {
    e.printStackTrace();
}
```

   处理文件流、网络连接、数据库连接等资源时，现代 Java 更推荐使用 try-with-resources，而不是手动在 finally 中关闭。

throws throws 用在方法声明上，表示当前方法可能抛出某些异常，但当前方法不负责处理，而是交给调用者处理。不是所有异常都应在当前方法中捕获。当前方法无法合理处理时，可以通过 throws 继续传播。

1. **基本语法** 在方法参数列表之后声明可能抛出的异常类型

```
public 返回类型 方法名(参数列表) throws 异常类型1, 异常类型2 {
    // 可能抛出异常的代码
}
```

   例如：

```
public static void readFile() throws IOException {
    FileInputStream input = new FileInputStream("t.txt"); // 可能抛出 IOException
    input.read(); // 可能抛出 IOException
}
```

   throws 只是声明当前方法可能抛出异常，并没有真正处理异常。

2. **调用者处理** 调用带有受检异常声明的方法时，需要继续捕获或声明

```java
public static void main(String[] args) {
    try {
        readFile(); // 调用可能抛出 IOException 的方法
    } catch (IOException e) {
        System.out.println("文件读取失败: " + e.getMessage());
    }
}
```

也可以继续向上声明：

```java
public static void main(String[] args) throws IOException {
    readFile(); // main 方法继续把 IOException 抛给 JVM
}
```

3. **throws 与 try-catch 配合使用** 底层方法声明异常，上层方法统一处理

```java
public static String readFirstLine(String path) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(path));
    return reader.readLine(); // 可能抛出 IOException
}

public static void main(String[] args) {
    try {
        String line = readFirstLine("test.txt");
        System.out.println(line);
    } catch (IOException e) {
        System.out.println("读取文件失败: " + e.getMessage());
    }
}
```

这种写法适合：

- 底层方法只负责完成具体任务，或者调用者更清楚如何处理异常
- 多个底层异常需要在上层统一处理

4. **多个异常声明** 一个方法可以同时声明多个异常

```java
public static void processFile(String path) throws IOException, SQLException {
    // 文件操作可能抛出 IOException
    // 数据库操作可能抛出 SQLException
}
```

如果多个异常有父子关系，可以声明它们的共同父类：

```
public static void processFile(String path) throws Exception {
    // 可以抛出多种 Exception 子类
}
```

在实际开发中，**不建议随意写过大的异常范围**，否则调用者难以判断具体可能发生什么问题。

5. **重写方法的异常规则** 子类重写方法时：
   - 不能扩大父类方法的受检异常范围
   - 父类方法没有抛出异常，子类重写时也不能抛出异常
   - 父类方法抛出多个异常，子类重写时只能抛出其中的部分或不抛

```
class Parent {
    public void read() throws IOException {
    }
}

class Child extends Parent {
    @Override
    public void read() throws FileNotFoundException {
        // 正确：FileNotFoundException 是 IOException 的子类
    }
}
```

错误示例：

```
class Child extends Parent {
    @Override
    public void read() throws Exception {
        // 编译错误：Exception 比 IOException 范围更大
    }
}
```

运行时异常不受这个规则严格限制：

```
class Child extends Parent {
    @Override
    public void read() throws RuntimeException {
        // 可以声明运行时异常
    }
}
```

throw throw 用在方法体内部，表示程序员主动创建并抛出一个异常对象。
throws 是声明异常，throw 是真正抛出异常。

1. **基本语法** 手动创建并抛出异常对象

```
throw new 异常类型("异常信息");
```

例如：

```
throw new IllegalArgumentException("参数不合法");
```

2. **throw 后面的代码不可达**

```
public static void test() {
    throw new RuntimeException("发生异常");
    // System.out.println("这行代码无法执行"); // 编译错误：无法到达
}
```

执行到 throw 后，当前方法的流程会中断，异常会交给当前方法的 catch 或调用者处理。

3. **throw 受检异常时需要配合 throws 或 try-catch**
如果手动抛出的是受检异常，当前方法必须捕获或声明：

```
public static void readConfig(String path) throws IOException {
    if (path == null || path.isBlank()) {
        throw new IOException("配置文件路径不能为空");
    }
}
```

如果抛出的是运行时异常，编译器不强制要求处理：

```
public static void readConfig(String path) {
    if (path == null || path.isBlank()) {
        throw new IllegalArgumentException("配置文件路径不能为空");
    }
}
```

4. **异常包装** 捕获底层异常后，转换成更适合当前业务语义的异常

```
public static void loadUserData() {
    try {
        readFile();
    } catch (IOException e) {
        throw new RuntimeException("加载用户数据失败", e);
    }
}
```

这里第二个参数 e 会保留原始异常原因，方便之后通过异常链定位问题。

区别	throws	throw
使用位置	方法声明上	方法体内部
作用	声明当前方法可能抛出的异常	主动抛出一个具体异常对象
后面跟的内容	异常类型	异常对象
数量	可以声明多个异常类型	一次只能抛出一个异常对象
是否真正抛出	不一定真正发生异常	执行到该语句时一定抛出异常
常见场景	当前方法无法合理处理异常	参数校验失败、业务规则不满足

表 1.34: throws 与 throw 的区别

总结

异常处理的基本原则

- 能预防的运行时异常，优先通过代码逻辑避免；
- 能恢复的异常，在当前位置捕获并处理；
- 当前方法无法合理处理的异常，使用 throws 继续传播；
- 捕获异常时优先捕获具体异常类型；
- 不要写空的 catch 块；
- 异常信息要清楚，方便定位问题；
- 资源释放优先使用 try-with-resources；
- finally 中只做清理工作，不要写复杂业务逻辑或 return。

常见异常及避免方法 见表格

异常类型	常见避免方法
NullPointerException	判空检查、合理初始化、使用 Objects.requireNonNull()
ArrayIndexOutOfBoundsException	检查下标范围、使用 length、使用增强型 for 循环
ArithmeticException	除法前检查除数是否为零
ClassCastException	转换前使用 instanceof
NumberFormatException	转换前检查字符串格式，捕获转换异常
InputMismatchException	使用 hasNextInt() 等方法检查输入类型
FileNotFoundException	检查路径、文件是否存在以及访问权限
SQLException	检查连接配置、SQL 语句、表结构和约束

表 1.35: 常见异常及避免方法

1.6.2 自定义异常类

自定义异常类是根据业务场景自己定义的异常类型，表达 Java 内置异常无法描述的错误。例如：

- 年龄不能为负数；
- 用户名长度不符合要求；
- 订单状态不允许修改；
- 余额不足，无法完成支付。

相比直接使用 `Exception` 或 `RuntimeException`，自定义异常可以让异常语义更清晰，也方便调用者进行针对性捕获和处理。

为什么需要自定义异常

Java 内置异常适合描述通用错误，但不一定能准确表达业务含义。

1. **内置异常的局限性** 异常类型太通用，业务含义不够清楚

   ```
   1 throw new RuntimeException("年龄不能为负数");
   ```

   这段代码虽然能抛出异常，但调用者只能知道发生了运行时异常，无法通过异常类型直接判断这是年龄错误。使用自定义异常后：

   ```
   1 throw new AgeException("年龄不能为负数", age);
   ```

   调用者可以直接捕获 `AgeException`，异常类型本身就表达了错误含义。

2. **自定义异常的优势**

   - 异常名称更贴近业务；
   - 错误信息更容易理解；
   - 可以保存额外的错误数据；
   - 可以针对不同业务异常分别捕获；
   - 代码可读性和可维护性更强。

   自定义异常的核心价值不是“制造更多异常类”，而是让异常类型准确表达业务错误。

自定义异常类的三步构建法

自定义异常类通常分为三步：

1. 继承异常基类；
2. 编写构造方法；
3. 在业务代码中使用。

1. **继承异常基类** 根据异常性质选择继承 `Exception` 或 `RuntimeException`

   ```
   1 // 受检异常（编译时异常）：调用者必须捕获或声明
   2 class AgeException extends Exception {
   ```

```
    // 代码...
}

// 运行时异常：编译器不强制处理
class UsernameException extends RuntimeException {
    // 代码...
}
```

继承选择原则：

- 外部环境问题、调用者必须感知的问题：继承 Exception；
- 参数错误、状态错误、业务规则错误：通常继承 RuntimeException。

例如，文件不存在、网络失败这类问题通常可以设计为受检异常；参数非法、用户名格式错误这类问题通常可以设计为运行时异常。

2. **编写构造方法** 通常保留常见异常构造器

自定义异常类本质上是一个普通类，需要继承异常父类，并调用父类构造方法保存异常信息。

```
public class AgeException extends Exception {
    //  serialVersionUID用于序列化，类似报警系统的"型号编号"
    private static final long serialVersionUID = 123456789L;
    private final int invalidAge; // 保存非法年龄

    public AgeException() {
        super(); // 无参构造
    }

    public AgeException(String message) {
        super(message); // 保存错误信息
    }

    public AgeException(String message, int invalidAge) {
        super(message); // 保存错误信息
        this.invalidAge = invalidAge;
    }

    public AgeException(String message, Throwable cause) {
        super(message, cause); // 保存错误信息和原始异常
    }

    public int getInvalidAge() {
```

```
        return invalidAge;
    }
}
```

常见构造方法包括：

- 无参构造；
- 只接收错误信息；
- 接收错误信息和原始异常；
- 接收业务字段，用于保存额外错误数据。

3. **使用自定义异常** 在业务规则不满足时主动抛出

```
public class UserService {

    public void register(String name, int age) throws AgeException {
        if (age < 0 || age > 150) {
            throw new AgeException("年龄必须在 0-150 之间", age); // 第三种构造器
        }
        System.out.println("用户" + name + "注册成功，年龄：" + age);
    }

    public static void main(String[] args) {
        UserService service = new UserService();
        try {
            service.register("张三", -5); // 可能抛出 AgeException
        } catch (AgeException e) {
            // 调用父类getMessage()
            System.out.println("注册失败：" + e.getMessage());
            System.out.println("无效年龄：" + e.getInvalidAge());
        }
    }
}
```

这里的流程是：

- register() 中发现年龄非法；
- 使用 throw 主动抛出 AgeException；
- 方法声明 throws AgeException；
- 调用者使用 try-catch 捕获并处理。

受检自定义异常与运行时自定义异常

自定义异常既可以设计成受检异常，也可以设计成运行时异常。

类型	继承方式	特点
受检自定义异常	`extends Exception`	调用者必须使用 try-catch 捕获或使用 throws 声明
运行时自定义异常	`extends RuntimeException`	编译器不强制处理，通常表示参数错误或业务规则错误

表 1.36: 受检自定义异常与运行时自定义异常

1. **受检自定义异常** 适合调用者必须处理的问题

```java
public class FileParseException extends Exception {

    private static final long serialVersionUID = 1L;

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

    public FileParseException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

使用示例：

```java
public static void parseFile(String path) throws FileParseException {
    if (path == null || path.isBlank()) {
        throw new FileParseException("文件路径不能为空");
    }
}
```

调用者必须处理：

```java
try {
    parseFile("");
} catch (FileParseException e) {
    System.out.println("文件解析失败：" + e.getMessage());
}
```

或者继续 throws。

2. **运行时自定义异常** 适合参数非法或业务规则不满足

```java
public class UsernameException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    private final String username;

    public UsernameException(String message, String username) {
        super(message);
        this.username = username;
    }

    public String getUsername() {
        return username;
    }
}
```

使用示例：

```java
public static void checkUsername(String username) {
    if (username == null || username.length() < 3) {
        throw new UsernameException("用户名长度不能小于 3", username);
    }
}
```

因为 UsernameException 是运行时异常，方法不需要强制写 throws。

如果调用者必须知道并处理这个异常，可以设计为受检异常；如果它主要表示程序参数或业务状态不合法，通常设计为运行时异常。

serialVersionUID

serialVersionUID 是序列化版本号，用于标识类的序列化版本。
异常类默认实现了 Serializable，因此自定义异常类通常建议显式声明 serialVersionUID。

1. **基本写法**

```java
public class AgeException extends Exception {
    private static final long serialVersionUID = 1L;
    public AgeException(String message) {
        super(message);
    }
}
```

2. **作用**

   - 标识类的序列化版本；
   - 避免编译器警告；
   - 当异常对象需要序列化传输或保存时，保证版本一致性。

   普通学习阶段可以先记住固定写法：

```
private static final long serialVersionUID = 1L;
```

异常链

异常链用于在抛出新异常时保留原始异常原因。
如果只抛出新的异常信息，可能会丢失底层异常的堆栈信息。

1. **包装原始异常** 使用带 cause 的构造方法

```
public class UserServiceException extends RuntimeException {

    private static final long serialVersionUID = 1L;

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

    public UserServiceException(String message, Throwable cause) {
        super(message, cause); // 保留原始异常
    }
}
```

2. **使用异常链** 将底层异常转换为业务异常

```
public static void loadUserData() {
    try {
        readFile(); // 可能抛出 IOException
    } catch (IOException e) {
        throw new UserServiceException("加载用户数据失败", e);
    }
}
```

   这里的 UserServiceException 表达业务语义，而第二个参数 e 保留了原始的 IOException。

3. **查看原始异常**

```
try {
```

```
    loadUserData();
} catch (UserServiceException e) {
    System.out.println(e.getMessage()); // 加载用户数据失败
    System.out.println(e.getCause());   // 原始 IOException
}
```

包装异常时应尽量保留原始异常对象，否则会丢失真正出错位置的调用栈。

自定义异常的命名规范

1. **类名应以 Exception 结尾**

   推荐：

```
AgeException
UsernameException
OrderStateException
PaymentException
DataValidationException
```

   不推荐：

```
AgeError
BadAge
WrongUsername
MyException
```

2. **名称应表达具体业务含义**

   - `AgeException`：年龄相关异常；
   - `UsernameException`：用户名相关异常；
   - `OrderStateException`：订单状态异常；
   - `InsufficientBalanceException`：余额不足异常。

   不要为了自定义而自定义。异常类名称应该能让人一眼看出具体业务错误。

自定义异常的处理策略

1. **能恢复的异常，在当前位置处理**（谁创建谁处理：创建数据的方法处理异常）

```
try {
    registration.register(username, age);
} catch (UsernameException e) {
    System.out.println("请重新输入用户名：" + e.getUsername());
}
```

2. **当前方法无法处理的异常，继续向上传播**

```java
public void registerUser(String username, int age)
        throws AgeException, UsernameException {

    registration.register(username, age); // 当前方法不处理，继续抛出
}
```

3. **底层异常可以转换为业务异常**

```java
public User loadUser(String path) {
    try {
        return readUserFromFile(path);
    } catch (IOException e) {
        throw new UserServiceException("读取用户信息失败", e);
    }
}
```

4. **异常信息应包含有效上下文**

不推荐：

```java
throw new AgeException("年龄错误", age);
```

推荐：

```java
throw new AgeException("年龄必须在 0-150 之间，实际值：" + age, age);
```

异常信息最好包含：

- 哪个业务操作失败；
- 哪个参数不合法；
- 实际输入值是什么；
- 合法范围或期望格式是什么。

1.7 Java 多线程编程

错题集

1.7.1 线程概念

线程是进程内部的一条执行路径。一个进程至少包含一个线程，多个线程可以在同一进程中分工执行多个任务。线程是 CPU 调度的基本单位；进程是系统分配资源的基本单位。

线程的定义

1. **基本概念**
   线程是进程中的执行单元。一个 Java 程序启动后，默认会先创建 main 主线程。

```java
public class ThreadDemo {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()); // main
    }
}
```

2. **线程的特点**
   - **一个进程至少包含一个线程**；
   - 多个线程可以共享同一进程的内存资源；
   - 每个线程有自己独立的运行栈和程序计数器；
   - 线程比进程更轻量，创建和切换成本更低。
3. **线程之间共享与独立**
   - 共享：堆、方法区、成员变量、静态变量；
   - 独立：虚拟机栈、程序计数器、局部变量。

单线程与多线程

1. **单线程** 只有一条执行路径，任务按顺序执行

```java
boilWater();       // 烧水
prepareNoodles(); // 准备泡面
washDishes();      // 洗碗
```

   - 逻辑简单；
   - 执行顺序明确；
   - 如果某一步等待，后续任务也必须等待。
2. **多线程** 多条执行路径，多个任务可以同时推进

```java
Thread thread = new Thread(() -> {
    System.out.println("准备调料"); // 子线程任务
});
thread.start(); // 启动新线程
```

需要注意：

- `start()`：启动新线程；
- `run()`：普通方法调用，不会启动新线程；
- 多线程可以提高效率，但会带来线程安全问题。

CPU 与线程

1. **单核 CPU 通过时间片轮转实现并发**
   单核 CPU **同一时刻只能执行一个线程**。多个线程快速切换，像同时运行，实际轮流执行。
2. **多核 CPU 可以实现真正并行**
   多核 CPU 可以让多个线程在不同核心上同时执行。
3. **并发与并行**
   - 并发（Concurrent）：多个任务在同一时间段内交替执行；
   - 并行（Parallel）：多个任务在同一时刻真正同时执行。

   并发强调“轮流推进”，并行强调“同时执行”。

线程调度

线程启动后，具体什么时候执行由操作系统和 JVM 调度决定。

1. **时间片**
   CPU 会给线程分配一小段执行时间，时间片用完后可能切换到其他线程。
2. **上下文切换**
   CPU 从一个线程切换到另一个线程时，需要保存和恢复线程状态。
3. **执行顺序不确定**
   多线程程序中，线程的执行顺序通常不固定。

```
Thread t1 = new Thread(() -> {
    System.out.println("线程 1");
});

Thread t2 = new Thread(() -> {
    System.out.println("线程 2");
});

t1.start();
t2.start();
```

上面代码中，线程 1 和 线程 2 谁先输出不确定。

1.7.2 线程创建使用

Java 中可以通过 Thread 类创建和管理线程。线程要执行的任务写在 run() 方法中，真正启动线程必须调用 start() 方法。

调用 start() 才会启动新线程；直接调用 run() 只是普通方法调用。

线程的核心组成

1. **线程对象** 使用 Thread 对象表示一个线程

```
Thread thread = new Thread(); // 创建线程对象
```

2. **线程体** 线程要执行的代码放在 run() 方法中

```
@Override
public void run() {
    System.out.println("线程任务执行中");
}
```

3. **启动线程** 使用 start() 启动新线程

```
thread.start(); // 启动线程，JVM 会自动调用 run()
```

方式一：继承 Thread 类

继承 Thread 类的步骤：

1. 定义类继承 Thread；
2. 重写 run() 方法；
3. 创建线程对象；
4. 调用 start() 方法。

```
// 继承Thread类，创建一个"写代码"线程
class CodeWriter extends Thread {
    @Override
    public void run() {
        System.out.println("写代码线程开始工作...");
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println("正在写第" + i + "行代码...");
                Thread.sleep(500); // 模拟写代码需要时间
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        System.out.println("写代码线程完成工作！");
    }
}

// 继承Thread类，创建一个"测试代码"线程
class CodeTester extends Thread {
    @Override
    public void run() {
        System.out.println("测试代码线程开始工作...");
        try {
            for (int i = 1; i <= 3; i++) {
                System.out.println("正在测试第" + i + "个功能...");
                Thread.sleep(700); // 模拟测试需要时间
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("测试代码线程完成工作！");
    }
}

// 主类，演示多线程的使用
public class ThreadInheritanceDemo {
    public static void main(String[] args) {
        System.out.println("主线程：程序员开始工作...");

        // 创建两个线程对象
        CodeWriter writer = new CodeWriter();
        CodeTester tester = new CodeTester();

        // 启动线程
        writer.start();
        tester.start();

        // 主线程可以继续做其他事情
        System.out.println("主线程：程序员去泡咖啡...");
        try {
            Thread.sleep(1000);
```

```
            System.out.println("主线程：咖啡泡好了！");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 等待两个子线程完成
        try {
            writer.join(); // 等待写代码线程完成
            tester.join(); // 等待测试代码线程完成
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("主线程：程序员完成所有工作，回家睡觉！");
    }
}
```

特点：

- 写法简单，可以直接使用 this 或 getName()；
- 受 Java 单继承限制，**继承了 Thread 后不能再继承其他类**；
- 线程任务和线程对象耦合较高。

方式二：实现 Runnable 接口

实现 Runnable 接口的步骤：

1. 定义类实现 Runnable；
2. 重写 run() 方法；
3. 创建任务对象；
4. 将任务对象传入 Thread；
5. 调用 start() 方法。

```
// 实现Runnable接口，创建一个"写代码"任务
class CodeWriterTask implements Runnable {
    @Override
    public void run() {
        System.out.println("写代码任务开始执行...");
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println("正在写第" + i + "行代码...");
```

```
                Thread.sleep(500); // 模拟写代码需要时间
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("写代码任务完成！");
    }
}

// 实现Runnable接口，创建一个"测试代码"任务
class CodeTesterTask implements Runnable {
    @Override
    public void run() {
        System.out.println("测试代码任务开始执行...");
        try {
            for (int i = 1; i <= 3; i++) {
                System.out.println("正在测试第" + i + "个功能...");
                Thread.sleep(700); // 模拟测试需要时间
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("测试代码任务完成！");
    }
}

// 主类，演示使用Runnable接口创建多线程
public class RunnableDemo {
    public static void main(String[] args) {
        System.out.println("主线程：程序员开始工作...");
        // 创建任务对象
        CodeWriterTask writerTask = new CodeWriterTask();
        CodeTesterTask testerTask = new CodeTesterTask();
        // 创建线程对象，并将任务传递给线程
        Thread writerThread = new Thread(writerTask);
        Thread testerThread = new Thread(testerTask);
        // 启动线程
        writerThread.start();
        testerThread.start();
```

```
        // 主线程可以继续做其他事情
        System.out.println("主线程：程序员去泡咖啡...");
        try {
            Thread.sleep(1000);
            System.out.println("主线程：咖啡泡好了！");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 等待两个子线程完成
        try {
            writerThread.join();
            testerThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("主线程：程序员完成所有工作，回家睡觉！");
    }
}
```

特点：

- 避免单继承限制；
- 任务和线程对象分离；
- 多个线程可以共享同一个任务对象；
- 实际开发中更常用。

Thread 与 Runnable 对比

对比项	继承 Thread	实现 Runnable
继承限制	受单继承限制	不影响继承其他类
任务复用	线程和任务绑定	任务可以被多个线程共享
代码耦合	较高	较低
使用方式	线程类自己调用 start()	包装成 Thread 后调用 start()
推荐程度	简单示例可用	更推荐

表 1.37: Thread 与 Runnable 的区别

start() 与 run() 的区别

1. **start()** 启动新线程

```
Thread thread = new Thread(() -> {
```

```
    System.out.println(Thread.currentThread().getName());
});
thread.start(); // 新线程执行
```

2. **run() 只是普通方法调用**

```
Thread thread = new Thread(() -> {
    System.out.println(Thread.currentThread().getName());
});
thread.run(); // main 线程中普通调用，不会启动新线程
```

同一个线程对象只能调用一次 start()。重复调用会抛出 IllegalThreadStateException。

Thread 常用方法

方法	作用
start()	启动线程
run()	定义线程任务
getName()	获取线程名称
setName(String name)	设置线程名称
currentThread()	获取当前正在执行的线程对象
sleep(long millis)	让当前线程休眠指定毫秒数
yield()	提示当前线程让出 CPU
join()	等待指定线程执行结束
isAlive()	判断线程是否还活着
getState()	获取线程状态
setPriority(int priority)	设置线程优先级
setDaemon(boolean on)	设置守护线程

表 1.38: Thread 常用方法

1. **线程命名与获取**

```
// 获取主线程
Thread mainThread = Thread.currentThread();
System.out.println(mainThread.getName()); // main

// 创建并命名子线程（构造器）
Thread thread1 = new Thread(() -> {
    System.out.println(Thread.currentThread().getName());
}, "子线程A");
thread1.start();
```

```
10
11 // 创建并命名子线程（setName）
12 Thread thread2 = new Thread(() -> {
13     System.out.println(Thread.currentThread().getName());
14 });
15 thread2.setName("子线程B");
16 thread2.start();
```

2. **sleep()** 让当前线程休眠

```
1 try {
2     Thread.sleep(1000); // 当前线程休眠 1 秒
3 } catch (InterruptedException e) {
4     e.printStackTrace();
5 }
```

3. **yield()** 线程让步

```
1 Thread.yield(); // 提示当前线程让出 CPU
```

yield() 只是给调度器一个提示，不保证其他线程一定立刻执行。

4. **join()** 等待线程执行结束

```
1 Thread thread = new Thread(() -> {
2     System.out.println("子线程执行");
3 });
4
5 thread.start();
6 thread.join(); // 当前线程等待 thread 执行完毕
7
8 System.out.println("主线程继续执行");
```

5. **isAlive()** 判断线程是否存活

```
1 Thread thread = new Thread(() -> {
2     System.out.println("子线程执行");
3 });
4
5 System.out.println(thread.isAlive()); // false
6
7 thread.start();
8
9 System.out.println(thread.isAlive()); // true
```

线程优先级

线程优先级用于提示调度器优先执行某些线程。

```
Thread low = new Thread(() -> {
    System.out.println("低优先级线程");
});

Thread high = new Thread(() -> {
    System.out.println("高优先级线程");
});

low.setPriority(Thread.MIN_PRIORITY);  // 1
high.setPriority(Thread.MAX_PRIORITY); // 10

low.start();
high.start();
```

常用优先级常量：

- `Thread.MIN_PRIORITY`：最低优先级，值为 1；
- `Thread.NORM_PRIORITY`：默认优先级，值为 5；
- `Thread.MAX_PRIORITY`：最高优先级，值为 10。

线程优先级只影响被调度的概率，不保证线程执行顺序。

守护线程

守护线程是为其他线程服务的后台线程。当程序中只剩守护线程时，JVM 会结束。

```
Thread daemonThread = new Thread(() -> {
    while (true) {
        System.out.println("守护线程运行中");
    }
});

daemonThread.setDaemon(true); // 必须在 start() 之前设置
daemonThread.start();
```

需要注意：

- `setDaemon(true)` 必须在 `start()` 前调用；
- 守护线程依赖普通用户线程；

- 垃圾回收线程就是典型守护线程。

线程生命周期

Java 官方线程状态主要包括：

状态	含义
NEW	新建状态，线程对象已创建但未启动
RUNNABLE	可运行状态，可能正在运行，也可能等待 CPU
BLOCKED	阻塞状态，等待获得锁
WAITING	无限等待状态，需要其他线程唤醒
TIMED_WAITING	计时等待状态，如 sleep()、限时 join()
TERMINATED	终止状态，线程执行结束

表 1.39: 线程生命周期状态

```
Thread thread = new Thread(() -> {
    System.out.println(Thread.currentThread().getState()); // RUNNABLE
});

System.out.println(thread.getState()); // NEW

thread.start();

System.out.println(thread.getState()); // RUNNABLE 或其他运行中状态
```

售票案例

多个线程共享同一个任务对象，可以模拟多个窗口卖同一批票。

```
class TicketSeller implements Runnable {

    private int tickets = 100; // 共享票数

    @Override
    public void run() {
        while (tickets > 0) {
            System.out.println(
                Thread.currentThread().getName() + " 售出第 " + tickets + " 张票"
            );

            tickets--;
        }
```

```
    }
}

public class TicketDemo {
    public static void main(String[] args) {
        TicketSeller seller = new TicketSeller();

        new Thread(seller, "窗口1").start();
        new Thread(seller, "窗口2").start();
        new Thread(seller, "窗口3").start();
    }
}
```

该代码可能出现重复售票或票数异常，原因是多个线程同时修改共享变量。解决方法需要使用线程同步。

补充：Callable 与 FutureTask

Runnable 没有返回值，也不能直接抛出受检异常。如果线程任务需要返回结果，可以使用 Callable 和 FutureTask。

```
Callable<Integer> task = () -> {
    return 100; // 返回计算结果
};

FutureTask<Integer> futureTask = new FutureTask<>(task);

Thread thread = new Thread(futureTask);

thread.start();

Integer result = futureTask.get(); // 获取线程执行结果

System.out.println(result); // 100
```

总结

- 线程任务写在 run() 方法中；
- 启动线程必须调用 start()；
- 继承 Thread 写法简单，但受单继承限制；
- 实现 Runnable 更灵活，更推荐；

- 多个线程可以共享同一个 Runnable 任务对象；
- sleep() 让当前线程休眠；
- yield() 提示当前线程让出 CPU；
- join() 等待其他线程执行结束；
- 线程优先级不保证执行顺序；
- 守护线程必须在启动前设置；
- 多线程共享数据时可能出现线程安全问题；
- 需要返回值的线程任务可以使用 Callable 和 FutureTask。

1.8 Java 枚举类

第 2 章：团队合作与版本控制

2.1 Maven

错题集

2.1.1 Maven 概述

Maven 是 Java 项目的**依赖管理**和**构建管理**工具。
Maven 不是 Java 语法，而是 Java 项目开发中的工程化工具。

为什么需要 Maven

传统方式需要手动下载、复制和管理 jar 包，问题主要有：

1. **jar 包数量多**：复杂项目可能依赖几十甚至上百个 jar 包；
2. **来源不规范**：第三方网站下载的 jar 包可能版本错误或内容不完整；
3. **依赖冲突**：不同 jar 包可能依赖同一个库的不同版本。

Maven 可以根据配置自动下载依赖，并按照规则处理大部分依赖冲突。

Maven 的核心功能

1. **依赖管理**
   在配置文件中声明依赖坐标，Maven 自动下载对应 jar 包。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

2. **构建管理**
   Maven 可以统一执行项目构建流程：

   清理 -> 编译 -> 测试 -> 打包 -> 部署

   常用命令：

```
mvn clean
mvn compile
mvn package
```

2.1.2 基于 IDEA 创建 Maven 工程

Maven 工程强调**标准化**：项目信息要规范，目录结构要固定。

Maven 项目的 GAVP

GAVP 是 Maven 项目的唯一标识：

- `GroupId`：组织或公司名，如 `com.dsbigdata`；
- `ArtifactId`：项目或模块名，如 `student-manager`；
- `Version`：版本号，如 `1.0.0`、`1.0.0-SNAPSHOT`；（在 `pom.xml` 修改）

- Packaging：打包方式，默认是 jar。

Packaging	含义
jar	普通 Java SE 工程
war	Java Web 工程
pom	父工程，管理子模块依赖

GroupId、ArtifactId、Version 是必填项；Packaging 可选，默认值为 jar。

创建 Maven Java SE 工程

在 IDEA 中创建普通 Maven 工程：

1. File -> New -> Project；
2. 选择 Maven，不要选 Maven Archetype；
3. 填写 GroupId、ArtifactId、Version；
4. 选择项目保存路径，点击 Finish；
5. 等待 Maven 插件和依赖下载完成。

如果右侧 Maven 视图中能看到：

```
Lifecycle
Plugins
Dependencies
```

说明 Maven 工程创建成功。

创建 Maven Java Web 工程

Java Web 工程需要将打包方式改为 war。
在 pom.xml 中加入（在 Version 之后一行）：

```
<packaging>war</packaging>
```

基本步骤：

1. 先按照创建 Maven Java SE 工程流程进行创建
2. 修改 pom.xml，设置 packaging 为 war
3. 在 src/main 下创建 web 目录
4. 配置 WEB-INF/web.xml（在 web 目录下）
5. 刷新 Maven，使配置生效

推荐手动创建 Web 工程，更容易理解 Maven Web 项目结构。插件创建虽然快，但不同 IDEA 版本可能不稳定。

Maven 工程目录结构

Maven 项目结构是标准化的，每个目录都有固定用途。

```
pom.xml                  # 项目的配置文件：管理依赖、插件、打包方式、项目信息
src
|-- main                  # 正式代码和资源
|   |-- java              # Java 源代码
|   |   `-- com/dsbigdata/mavenweb
|   |       |-- controller  # 控制层，处理请求
|   |       |-- service     # 业务逻辑层
|   |       |-- dao         # 数据访问层
|   |       `-- model       # 实体类、数据模型
|   |
|   |-- resources         # 配置文件和资源文件
|   |   |-- log4j.properties
|   |   |-- spring-mybatis.xml
|   |   `-- static
|   |       |-- css
|   |       |-- js
|   |       `-- images
|   |
|   `-- webapp            # Web 工程相关文件
|       |-- WEB-INF       # 浏览器不能直接访问
|       |   `-- web.xml   # Web 配置文件
|       `-- index.html      # 默认首页
|
|-- test                   # 测试代码和资源
|   |-- java              # 单元测试代码
|   `-- resources           # 测试配置文件
|
`-- target                 # Maven 自动生成的编译、打包结果
```

Java 代码放在 src/main/java；配置文件放在 src/main/resources；Web 文件放在 src/main/webapp。

POM 核心信息配置

pom.xml 中的核心项目信息主要包括 modelVersion、groupId、artifactId、version 和 packaging。

```
<modelVersion>4.0.0</modelVersion>
```

```
<groupId>com.companyname.project-group</groupId>
<artifactId>project</artifactId>
<version>1.0.0</version>

<packaging>jar</packaging>
```

各标签含义：

- `modelVersion`：Maven 模型版本，固定写 `4.0.0`；
- `groupId`：公司或组织的唯一标识；
- `artifactId`：项目或模块名称；
- `version`：项目版本号；
- `packaging`：打包方式，默认是 `jar`。
  - `jar`：普通 Java 项目；
  - `war`：Java Web 项目；
  - `pom`：父工程，用于管理子模块。

2.1.3 工程构建

工程构建是把源代码、配置文件和依赖库，转换成可运行程序的过程，最终生成 `jar` 或 `war` 包。

构建流程

Maven 构建过程可以理解为一条自动化流水线：

```
编译 -> 测试 -> 打包 -> 安装 -> 部署
```

- **编译**：将 `.java` 文件编译为 `.class` 文件；
- **测试**：执行 `src/test/java` 下的测试代码；
- **打包**：生成 `jar` 或 `war` 文件；
- **安装**：将打包结果放入本地仓库；
- **部署**：将打包结果上传到远程仓库。

Maven 构建的重点是自动按顺序执行流程，减少手动操作错误。

常用构建命令

构建命令之间的关系 Maven 命令本质上是生命周期中的不同阶段。
默认生命周期常用顺序为：

```
(validate) -> compile -> test -> package -> install -> deploy
```

命令	功能描述
`mvn compile`	编译项目源代码，生成 `target` 文件夹，里面放 `.class` 文件
`mvn test`	先编译源代码，再执行 `src/test/java` 里的测试代码，输出测试结果
`mvn package`	先编译、测试，再把项目打包成 `jar` 或 `war` 文件，放在 `target` 里
`mvn clean`	清理项目构建生成的文件，比如 `target` 文件夹、打包好的 `jar` 包
`mvn install`	先执行 `package` 打包，再把打包好的 `jar/war` 包放到**本地仓库**
`mvn deploy`	先执行 `install`，再把 `jar/war` 包放到**远程仓库（私服）**
`mvn site`	生成项目的文档站点，包含 API 文档、测试报告等

执行后面的命令，会**自动执行前面的**命令：

- `mvn compile`：只编译主代码；
- `mvn test`：先编译，再测试；
- `mvn package`：先编译、测试，再打包；
- `mvn install`：先打包，再安装到本地仓库；
- `mvn deploy`：先安装，再上传到远程仓库。

`clean` 属于清理生命周期，和上面默认生命周期不是一条线，通常放在前面组合使用：

```
mvn clean package
mvn clean install
```

`site` 用于生成项目文档站点，和编译、测试、打包不是同一类核心构建流程。
后面的命令会自动带出前面的步骤；`clean` 是先清理旧结果，再重新构建。

生命周期、命令、插件的关系　三者关系可以概括为：

- **生命周期**：规定构建流程；
- **命令**：生命周期中的具体阶段；
- **插件**：真正执行命令的工具。

例如：

- `compile` 通常由 `maven-compiler-plugin` 执行；
- `package` 会根据项目类型调用 jar 或 war 打包插件。

生命周期定流程，命令是步骤，插件负责真正干活。

war 包插件版本问题

如果 Web 工程打包时报 `maven-war-plugin` 版本不兼容，可以在 `pom.xml` 中指定插件版本：

```
<build>
```

```
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.2.2</version>
        </plugin>
    </plugins>
</build>
```

修改后重新执行：

```
mvn clean package
```

IDEA 可视化构建

在 IDEA 右侧 Maven 视图中，展开 Lifecycle “文件夹”，双击对应命令即可执行构建。
构建结果查看底部 Run 窗口：

- BUILD SUCCESS：构建成功；
- BUILD FAILURE：构建失败，需要查看报错信息。

2.1.4 Maven 依赖管理

Maven 依赖管理用于自动处理项目需要的第三方 jar 包。
主要作用：

- 自动下载 jar 包
- 统一管理依赖版本
- 减少手动导包和版本冲突问题

Maven 依赖都写在 pom.xml 中，不需要手动复制 jar 包到项目里。

添加依赖

依赖写在 dependencies 标签中。一个依赖至少需要三个坐标：

- groupId：组织或公司名；
- artifactId：项目或模块名；
- version：版本号。

```
<dependencies>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
```

```
        <version>1.2.17</version>
        <!-- 依赖范围（optional）-->
    </dependency>
</dependencies>
```

添加依赖的本质是告诉 Maven：我要哪个 jar 包。

依赖信息查询方式：maven 仓库信息官网 https://mvnrepository.com/ 或者 mavensearch 插件搜索。

统一管理依赖版本

如果多个地方都要使用同一个版本号，可以先在 properties 中声明，再通过 ${} 引用。

```
<properties>
    <junit.version>4.12</junit.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
    </dependency>
</dependencies>
```

这样以后升级版本，只需要改 properties 中的一处。

依赖范围 scope

scope 用来控制 jar 包在哪些阶段生效。示例：

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

不要所有依赖都默认用 compile；测试依赖用 test，容器已提供的依赖用 provided。

scope	作用
compile	默认范围，编译、测试、运行都有效
test	只在测试阶段有效，例如 junit
provided	编译、测试有效，运行时由外部环境提供，例如 servlet-api
runtime	测试、运行有效，编译时不需要，例如 JDBC 驱动实现
system	引入本地 jar 包，不推荐使用
import	配合 dependencyManagement 导入依赖管理配置

依赖下载失败处理

如果依赖变红，或构建时报依赖找不到，常见处理方式：

1. 检查网络和 Maven 仓库配置；
2. 检查依赖坐标是否写错；
3. 删除本地仓库中的 .lastUpdated 文件：.lastUpdated 表示依赖上一次下载失败，不删除可能会导致 Maven 一直不重新下载
4. 在 IDEA 中重新刷新 Maven 项目：右键项目 -> Maven -> Reload Project

Build 配置补充

build 标签用于修改 Maven 默认构建行为。
常见用途：

- 修改最终打包名称；
- 指定额外资源文件参与打包；
- 配置构建插件。

修改打包名称：

```
<build>
    <finalName>my-project</finalName>
</build>
```

如果 src/main/java 中有 XML 文件，默认不会被打包。可以通过 resources 指定打包规则：

```
<build>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
```

```
                <include>**/*.xml</include>
            </includes>
        </resource>
    </resources>
</build>
```

配置 Java 编译插件：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
    </plugins>
</build>
```

总结

- 依赖写在 dependencies 中；
- 版本号可以用 properties 统一管理；
- scope 控制依赖在哪些阶段生效；
- 下载失败先检查坐标、仓库和 .lastUpdated；
- build 用于定制打包名称、资源文件和插件。