JavaSE

前言

程序开发

程序：我们想要实现的功能的集合，就是程序。（网易云、微信）

通过编程语言进行编程，让计算机实现我想要的功能。

为什么学习Java

1. 大数据领域中，很多的组件都是使用Java开发的，学习Java，可以更好的帮助我们进一步了解大数据的底层实现。
2. 扩充我们的编码能力。在大数据领域中，如果需要实现数据分析，通常情况下，我们使用SQL就可以完成大多数的功能。但是在某些场景下，原生的SQL语言并不能覆盖我们的需求，那么我们可以通过UDF的方式/java编程的方式去解决。
3. 学习java语言，不要有太大的负担。因为大数据中，大多数的数据处理组件都提供了SQL的API，可以使用SQL实现大多数的需求。

java语言概述

- 是SUN(Stanford University Network，斯坦福大学网络公司) 1995年推出的一门**高级**编程语言。
  - 什么是高级编程语言
  - 从计算机的角度出发，能够直接被计算机执行的语言属于低级语言（汇编语言、机器语言）
  - 高级语言一般是需要先经过**编译**过程转换成低级语言，然后再在计算机上运行。（javac）
- 是一种面向Internet的编程语言。Java一开始富有吸引力是因为Java程序可以在Web浏览器中运行。这些Java程序被称为Java小程序(applet)。applet使用现代的图形用户界面与Web用户进行交互。 applet内嵌在HTML代码中。
- 随着Java技术的不断成熟，已经成为Web应用程序、大数据开发等方面的首选开发语言。 HDFS、MapReduce、Hive、Kafka、elasticsearch等都是采用Java开发的

Java 主要特性

1. 跨平台性：Java代码可以编译成平台无关的字节码，然后在任何支持Java虚拟机（JVM）的计算机上运行，无需修改代码，从而实现了跨平台性。
2. 面向对象编程（OOP）：Java是一种面向对象的编程语言，支持封装、继承和多态等OOP特性，使得程序的设计和实现更加灵活和可维护。
3. 垃圾回收机制：Java有垃圾回收机制，可以自动管理程序的内存分配和释放，减轻了程序员的负担，避免了内存泄漏和内存溢出等问题。

4. 异常处理机制：Java提供了强大的异常处理机制，可以让程序更加健壮和可靠，及时处理程序中的错误和异常情况。
5. 多线程支持：Java提供了多线程支持，可以实现并发编程，充分利用多核CPU和多任务处理器，提高程序的性能和响应速度。
6. 安全性：Java具有内置的安全特性，如安全沙箱、数字签名、加密等，可以保护程序免受恶意攻击和安全漏洞的影响。
7. 高效性：Java的执行效率非常高，因为它使用了即时编译技术（JIT）和逃逸分析技术等，可以将频繁使用的代码编译成本地机器指令，从而提高程序的执行效率。
8. 丰富的类库和工具：Java拥有大量的标准库和第三方库，可以方便地实现各种应用程序的功能，同时Java还有丰富的开发工具和集成开发环境（IDE），如Eclipse、Idea等，可以提高开发效率和质量。

java 开发环境搭建

JDK = JRE + 开发工具集(例如Javac编译工具等)

JRE = JVM + Java SE标准类库

一、java基础

1. 第一个java程序

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

1. java 的程序，都是在 ".java"后缀的文件中编写。这些文件就是所谓的java源码。
2. java源码的组成，必然包括 class（类），比如上述例子的 Main 就是一个类。
3. java的程序入口是 `main(String[]): void` 方法 

    1. 其中 `main(String[]) : void` 叫做方法签名。
    2. 方法的定义必须包含： 方法名、参数列表（可为空）、返回值类型。
    3. java的程序入口是固定的，定义方式是： `public static void main(String[] args)`，我们唯一能修改的就是 args这个参数名称，其他部分必须固定，否则java无法运行。
4. java中是大小写敏感的。另外，java的每个语句都必须以 “;” 作为结尾。
5. 大括号是成对出现的，缺一不可。大括号的作用：规定了代码的作用域。
6. 每个源文件当中只能包含一个public 类，并且这个public修饰的类必须声明于 相同名字的 java源码文件中。

6. java程序想要执行起来，必须先进行编译，之后再运行。

```shell
# 编译
javac  Main.java
# 运行
java Main
```

7. 每次修改代码后，必须要对源码重新编译，生成新的class文件，才会生效。如果是使用idea，那么这个过程是自动的。
8. java源码中，代码的缩进和换行不是必须的，只是为了更加方便人类阅读，更清晰的看到代码的层级结构

2. Java基础语法

Java注释

在Java的编程过程中我们需要对一些程序进行注释说明，方便自己他人更好阅读和维护。

单行注释： //

- 一行解释

多行注释： /* */

- 有更多文字的描述

文档注释： /** */

- 针对类或者方法的注释

关键字

- 被Java语言赋予了**特殊含义**，用作专门**用途**的**字符串**(单词)
- 类似SQL中的select，from，where，group by，在我们写的代码避免使用这些关键字
- Java区分大小写，关键字也一样。

标识符

- Java 对各种**变量、方法和类**等要素**命名**时使用的**字符序列**称为标识符
- 技巧:凡是自己可以起名字的地方都叫标识符。
- 定义合法标识符规则: （强制性的）
  - 由26个英文字母大小写，0-9 ，_或 $ 以及中文组成;（一般不使用中文作为标识符）
  - 数字不可以开头；
  - 不可以使用关键字和保留字，；
  - Java中严格区分大小写，长度无限制；但能包含关键字和保留字
  - 标识符不能包含空格。
- 规范(约定俗成，非强制，但是遵从会更好)
  - 规范其实就是一种标准，大家按照相同的标准开发，后续与别人的交流更加清晰，简单
  - 一般公司内都会有相应的Java开发规范

Java变量

- 什么是变量(存储数据值的容器)

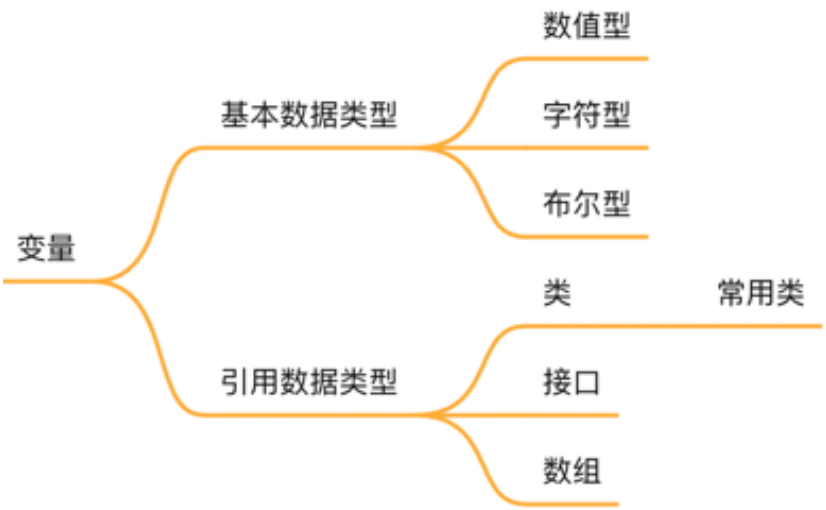
  在数学中，我们学过y=x+1; x表示自变量，y表示应变量

在Java中变量指的是内存当中的一块存储区域，在这个区域中，变量的值可以同类型范围内不断变化。

- 怎么定义变量：

变量类型 变量名(标识符) = 存储值，比如int i=1 这里int就是变量类型 i表示变量名 1表存储值

- 变量的规则:
  - Java中每个变量**必须先声明，后使用**，声明和赋值可以同时进行，也可以分开进行。（前向引用）
  - 使用变量名来访问这块区域（内存区域）的数据
  - 变量的作用域:其定义所在的一对{ }内
  - 变量只有在其作用域内才有效
  - 同一个作用域内，不能定义重名的变量
- 变量分类



  - 基本数据类型
    - 数值型：
      - 整型：byte(1字节),short，int,long(8字节)
      - 浮点型：float,double（3.1415926）
    - 字符型：char 单引号括起来
    - 布尔类型：boolean ，取值只能是true或false
  - 引用数据类型(类[字符串]，接口 数组 枚举)
    - 对象：所有的对象都属于引用数据类型。
      - 字符串：String 表示字符串，通常用双引号括起来，比如String message=“我是中国人”;
    - 数组：数组指的就是一组**相关类型**的变量**集合**，并且这些变量可以按照统一的方式进行操作。数组本身属于**引用数据类型**。

基本数据类型转换

- 自动类型转换：从容器小的类型，转成容器大的类型，这个过程可以由Java语言自动进行

- 强制类型转换：从容器大的类型，转成容器小的类型，需开发者强制转换，而且可能造成精度丢失(精度溢出)
  在可控范围内追求高性能或者是多态(面向对象)

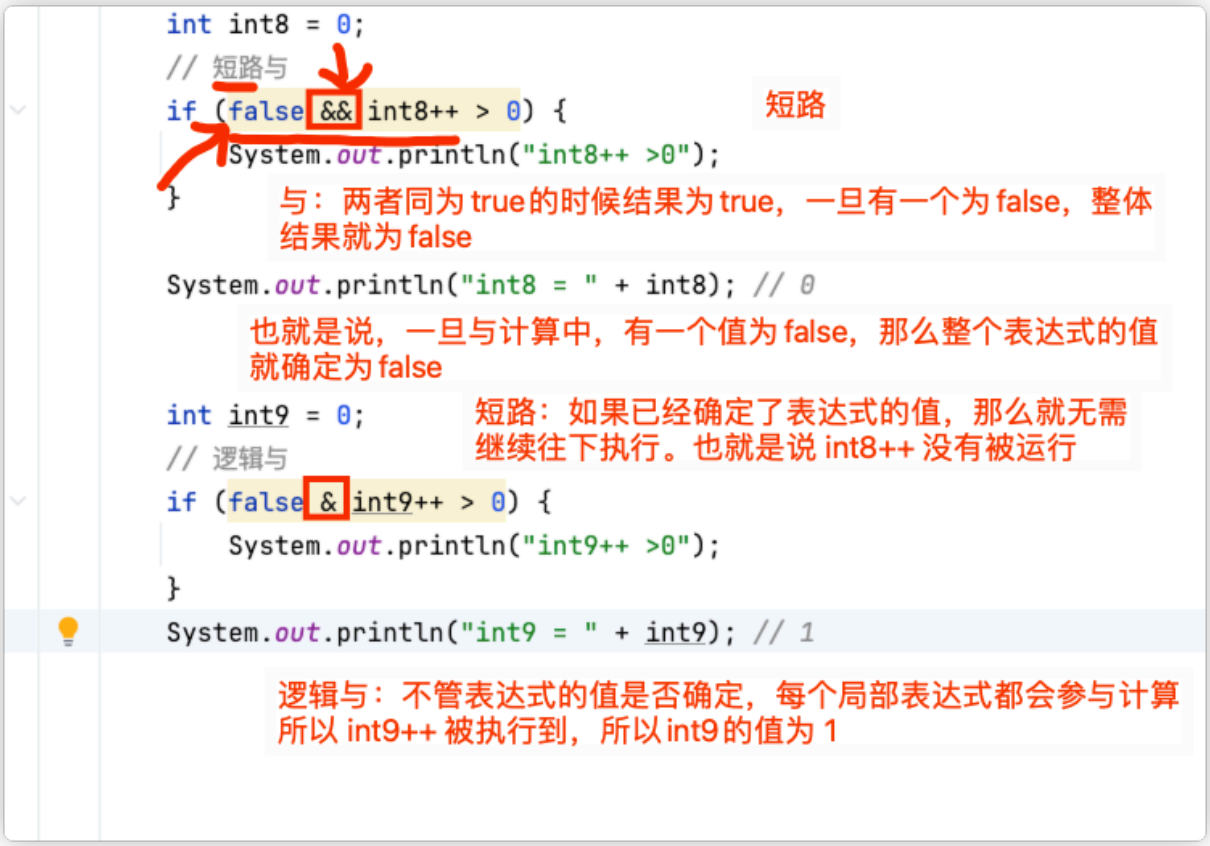
Java运算

运算符：一种特殊的符号，用以表示数据的运算、赋值和比较等

我们学过数据，数学中也有运算符，比如加减乘除，这种运算符叫做算数运算符

Java运算分类：

- 赋值运算符 (比如 int i=10,这里的等号就是赋值运算符)
- 算术运算符(+,-,++,--,* / %)
  - int a=2
  - int b=++a;a先自增为3,然后赋值给b，b==3；
  - int c=a++;a先赋值给c，c==2，然后自增，为3
- 比较运算符(关系运算符) (> ,<,==,>=,<=)
  - ==与equals的联系和区别
    - == 便是的是判断两个基本数据类型是否相等，对于引用数据类型，比较的是他们在内存中的地址值
    - equals 是一个方法，这个方法定义如何比较两个应用数据类型对应对象的值的过程。
- 逻辑运算符（与或非，异或）重点掌握
  - 与： 两者同为真时结果为真
    或：两者有一个为真则结果为真
    非：取反
    异或：a和b不同的时候结果为真
  - 双&时，如果左边为真，右边参与运算，反之，则右边不参与运算。（短路,**高效**）



- 位运算符（了解即可）
  - 6 &3 = 2

    0110 420
    0011 21
    0010 结果
- 三元运算符

Java流程控制

- 顺序结构
  - 程序按照顺序，从前往后执行
- 分支结构
  - if
  - if else
  - if else if
- 循环结构
  - 程序按照顺序，从前往后执行，完成一个循环后，判断是否满足跳出循环的条件，不满足则继续执行
  - for
  - while
  - do-while
- break和continue

- 都可以对提前结束循环
- break为结束当前作用域的循环
- continue为结束本次循环，进行下一次循环

二、面向对象基础

1. 什么是面向对象

- 通俗来讲，面向对象和面向过程都是一种**编程思想**。
- 面向过程：面向过程强调的是功能本身，以函数（SQL中sum、min、avg）作为最小单位，考虑的是**怎么做**
- 面向对象：面向对象，是将功能封装进对象里面，强调的是具备了功能的对象，一般来说以类或者对象为最小单位，考虑的是是**谁（对象）去做**
- **万事万物皆对象**：当我持有了对象（的引用），就相当于拥有了属性和功能，我们通过操作这些功能/访问这些属性，达到完成需求的目的。

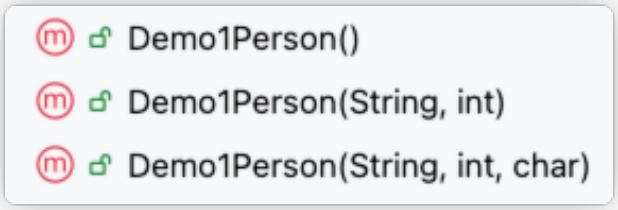
2. 面向对象编程语法

```
Person zhangsan = new Person();
```

这里的Person是类，zhangsan是对象的引用。

类在Java中属于一等公民。

- 类：类是对一类事物的描述，是抽象的、概念上的定义（人类）
  - 属性：身高、体重、年龄、姓名等（名词性的描述），一般也会称之为 成员变量
  - 行为：游泳、吃饭、sayHello等（动作类），一般也会称之为 成员方法（函数）
- 对象：对象是实际存在的该类事物的每个个体，因而也称为实例(instance)。
- 属性和方法的定义
  - 属性，其实就是这个类的变量
  - 方法，定义方式：`[修饰符]  返回值类型  方法名(形式参数列表){方法体}`
    - 返回值类型可以是基本数据类型，比如int、double、boolen，也可以是引用数据类型，比如String、Person
    - 形式参数列表：可以有多个参数，每个参数必须要有参数的类型和参数的名称
    - 方法体：功能的具体实现
- 构造器：什么是构造器？有什么作用
  - 用来创建对象，给对象进行初始化的方法就叫做构造器，也叫做构造方法。
  - Java语言中，每个类都至少有一个构造器 （如果你**没有定义构造器**，系统会默认帮你定义一个**无参构造器**）
  - 一旦显式（手动）定义了构造器，则系统不再提供默认构造器 ，如果此时还需要使用无参构造器，必须自己显示定义出来
  - 一个类可以创建多个重载的构造器（名称一致，参数列表不同）



    - 父类的构造器不可被子类继承 （继承）
    - 默认构造器的修饰符与所属类的修饰符一致

类（静态）属性、类（静态）方法

- 静态属性
    - **共享数据**：多个对象可以共享同一个变量，不需要为每个对象都创建一份。
    - **节省内存**：因为只在类加载时创建一次，不随对象实例化而重复创建。
    - **全局变量的效果**：在类的内部，可以实现类似全局变量的功能。
- 静态方法
    - **可以不创建对象就调用**：直接通过 `类名.方法名()` 调用。
    - **常用于工具类和辅助方法**：比如 `Math` 类里的 `Math.sqrt()`、`Math.random()`。
    - **不能访问非静态成员**：因为非静态成员属于对象，而静态方法不依赖对象。
- 静态代码块
    - 在类加载的时候，会进行一次性的执行的代码，就是静态代码块

```
static {
    System.out.println("这个是一个静态代码块，在类加载的时候就会被执行");
    System.out.println("静态代码块通常的作用是优先进行一次初始化的设置");
    PI = 3.14159265358979323846;
}
```

- 单例设计模式
    - 私有化构造器
    - 提供静态方法获取单例对象

3. 面向对象的三大特性

1. 封装

封装是指一种将抽象性函数式接口的实现细节部分包装、隐藏起来的方法。

- 我们去使用洗衣机，把衣服放进洗衣机，按下开关，选择洗涤模式就可以了。（我们不知道洗衣机内部的构造）
- 开车，需要插上车钥匙，启动车辆就可以开车了。（我们也不知道汽车的发动机内部的构造和原理）
- 封装：隐藏对象内部的复杂性，只对外公开简单的接口，便于外界调用，从而提升系统整体的可扩展性和可维护性。通俗来讲，就是**把该隐藏的东西隐藏起来**，把该暴露的东西暴露出来。这个就是封装的设计思想。

- 权限修饰符是针对类的成员（类的属性和方法）来使用的，也可以对类本身（public、默认）进行修饰。

修饰符	同类（同文件）	同包	子类（不同包）	其他包
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
默认（无修饰符）	✅	✅	❌	❌
`private`	✅	❌	❌	❌

- public： 所有的地方都可以访问到，包括其他包小的类、其他工程等情况。
- protected：同一个包下的所有代码可以访问到。如果是其他包，则只能是该类的子类才能访问。
- 默认：同一个包下的所有代码可以访问到。
- private：只能在当前类中访问，其他地方都无法访问。

2. 继承

继承表示的是 子类 is a 父类（哺乳动物和猫科动物都是动物， 动物是一种生物）

1. 为什么要有继承：
    1. 减少代码的冗余，提高代码的复用性（相同的属性或者行为，可以抽取出来，放在父类中，不需要每个子类都定义一次）
    2. 提高代码的扩展性：子类可以有自己特有的 属性 和 方法，也可以重写父类中的方法，实现子类的特异性。（丰富了父类的能力）

3. 多态

一个对象，可以有多种形态，这个就叫做多态

1. 代码中体现为： 父类的引用，指向子类的对象 `Animal animal = new Dog("旺财",2,"金毛")`
2. 为什么要用多态：实现 高内聚，低耦合

4. 抽象类

- 抽象类是一种特殊的类，它不能被用于创建对象（不能实例化），只能被继承。
- 抽象类可以包含抽象方法和非抽象方法
    - 抽象方法：每个子类可能都有不同的实现方式
    - 非抽象方法：子类的行为一致，可以认为是一种默认实现行为，当然子类也可以重写这些方法
- 抽象方法是一种没有具体实现逻辑的方法，它只有方法签名，没有方法体。
- 抽象类可以有构造函数，但是不能用于实例化，这个构造函数实际上只是方便子类使用（super调用）
- 抽象类可以被继承（主要作用也是用来被继承），子类可以重写父类的方法，也可以添加自己的属性和方法，但是如果子类不是抽象类的话，所有的父类中的抽象方法都要在这个子类中实现。（规范子类的行为）
- **抽象类**是**半成品模板**，通过抽象方法强制子类实现关键逻辑，同时支持代码复用。

5. 接口

- 接口中的方法也只有方法签名，没有方法体。本质上接口中方法都是**抽象**，在Java8新增了接口的默认实现，用default关键字。
- 接口也不能被实例化，只能被其他类 实现 （implement）
- 一个类只能继承一个父类，但是可以实现多个接口（单继承、多实现）
- 实现了接口的类，必须实现接口中所有的抽象方法，否则这个类就必须是抽象类（但凡是具体的子类，就不能拥有抽象方法。）
- 接口可以被继承，子接口可以添加自己的方法和常量（接口中的所有变量都自带final关键字，也就是无法被修改）
- 接口中不能包含 ”抽象方法、默认方法、静态方法“ 以外的方法。
- 接口中不能有构造器。（因为不能被实例化）

6. 扩展：函数式接口

函数式接口是一种特殊的接口。(spark、flink)

Java中，函数式接口指的是 **仅包含一个抽象方法** 的接口（可以有默认方法或者静态方法）。函数式接口可以用**Lambda表达式**或者方法引用来简化实现。

Function和Consumer是最常用的Java 预定义的函数式接口，Supplier（生成元素，没有输入，但是能获取输出）

三、集合

1. 什么是集合

Collection本身只是一个接口，定义了集合需要包含的一系列方法，包括获取数据的大小、是否包含某含某个元素，增删元素的方法等。

2. List

> 元素有序，且可重复的集合

- ArrayList：内部就是一个数组，只是基于这个数组，做了大量的工作：比如自动扩容，比如添加、删除元素的方法
- LinkedList：内部是使用双向链表实现，内部的结构叫做Node，这个Node包括前指针、后指针和元素本身，相应的添加了一些针对起始、结束位置的操作，addFirst、removeLast
  - linkedList可以实现很多的数据结构，比如：
    - 队列：先进先出（add往last添加，get的时候从first获取） `push`、`pop`
    - 栈：先进后出 `addFirst`、`removeLast`

3. Set

> 元素无序（跟添加的顺序无关），并且不可重复的集合

- HashSet：按照hash算法来存储集合中的元素，具有良好的存、取和删除、查找的性能。
- LinkedHashSet：双向链表的方式保证了添加顺序信息，相当于有序的HashSet
  - 性能比HashSet低（元素少的时候可以忽略不计）
- TreeSet：通过红黑树作为底层数据结构，优化了排序的功能。https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

4. Map

> map是双列数据，键值对的模式

- 所有的key拿出来的话就是一个 Set
- 所有的value拿出来的话就是一个集合
- **HashMap**：基于哈希表实现，允许null作为key和value的值，不保证元素的迭代顺序，非线程安全。适合在单线程环境下高效地进行插入、删除和查找操作。
  - HashMap内部是一个数组，每个数组元素都是一个链表或红黑树（Java 8之后），每个链表或树节点都是一个键值对。HashMap使用key的hashCode来确定该键值对在数组中的索引位置，并在链表或树节点中查找值。
- LinkedHashMap：继承自HashMap，使用哈希表加双向链表实现，可以按照元素的添加顺序进行迭代。因为LinkedHashMap内部使用双向链表维护元素的添加顺序。适合需要按照元素添加顺序进行迭代的场景。
- TreeMap：底层是红黑排序树的map结构
- Properties：继承自Hashtable，唯一的区别就是Properties只能存储字符串类型的键值对。通常用来存储应用程序的配置信息，例如数据库连接信息、日志级别等。

5. 集合的工具类

`java.util.Collections`

- 排序（reverse、shuffle、sort）
- 最值（max、min）
- 交换元素（swap）
- 等等（替换元素、创建空集合、交集等）

6. 泛型

> 在定义集合类的时候不知道存储的元素的具体类型，所以用一个英文代替，这个就做泛型，在使用的时候，使用者指定这个类型为具体的类。相当于在使用的时候，使用者明确了容器中要存放的元素类型，那么当我放一个不是这种类型元素进去时，Java就会检测出来。

泛型是 Java 5 引入的一项重要特性，它提供了**编译时类型安全检查**机制，并允许程序员在编写代码时使用**类型参数**（type parameters）。

泛型的本质是**参数化类型**，即所操作的数据类型被指定为一个参数，在使用时再传入具体的类型。

泛型可分为：

- 泛型类 `class Box<T>`
- 泛型方法 `public static <E> void printArray(E[] array)`
- 泛型接口 `public interface Comparable<T>`

泛型通配符（了解）

- 无界通配符 `public static void printList(List<?> list)`
- 上界通配符 `public static double sumOfList(List<? extends Number> list)`
- 下界通配符 `public void addNumbers(List<? super Integer> list)`

有界类型参数：

普通的泛型参数没有限定，任何的类型都可以传递。如果我们想限定参数范围，则可以使用有界类型参数。

`public static <T extends Comparable<T>> int compareObjects(T a, T b)`

限定了传入来的a和b两个对象对应的类T，必须是Comparable的子类，即 T 类必须实现Comparable接口

四、异常处理

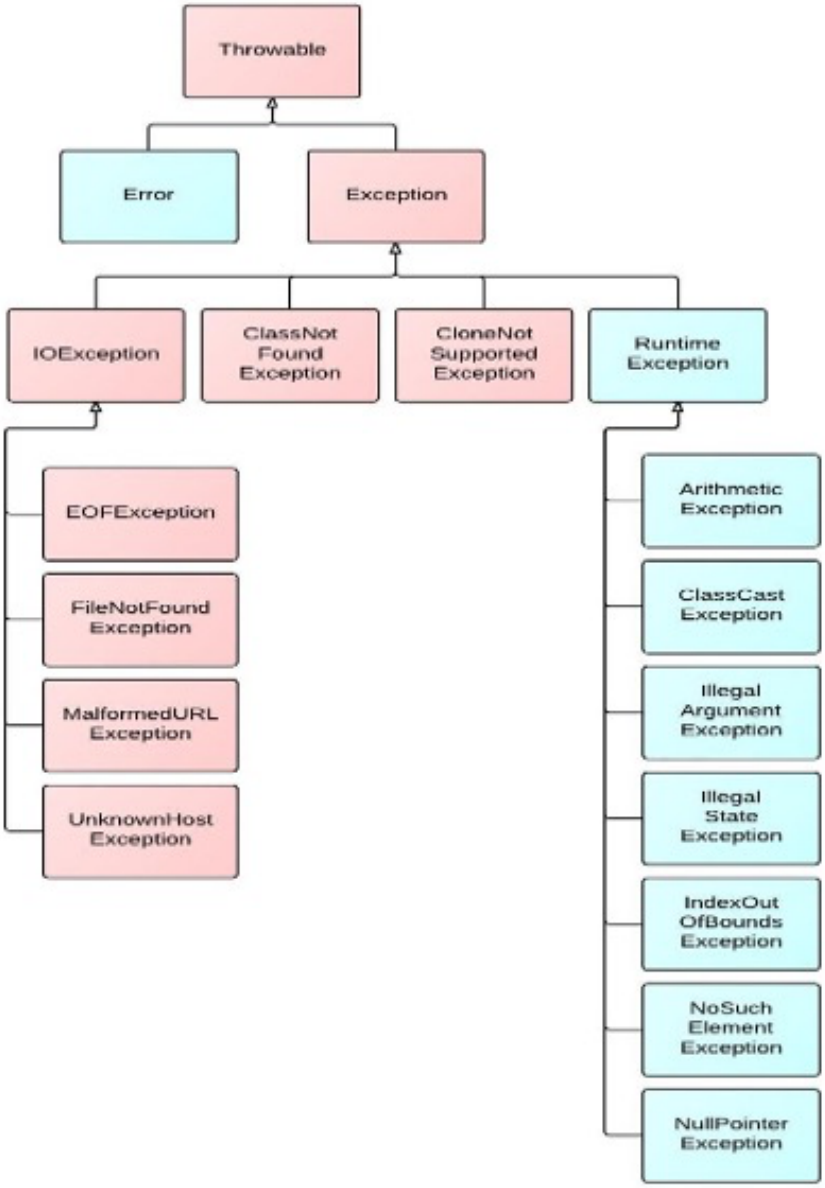
1. 异常的定义

在程序执行过程中，可能会发生一些非预期的事件，这些事件会中断程序的正常运行。这些事件就被叫做异常（Exceptions）

2. 异常的分类

- Error（错误）：
  - 表示程序无法处理的严重问题（类比 windows的蓝屏），通常是JVM运行环境有关，比如(堆)内存溢出`OutOfMemoryError`、栈内存溢出`StackOverflowError`
  - 这种error通常不处理（无法处理）
- Exception（异常）：
  - 表示程序本身可以处理的问题。又分为两大类
    1. 检查型异常（checked）：编译器在编译运行之前就会检查的检查的异常，如果不处理，编译器不允许编译。 比如：`FileNotFoundException`

2. 非受检异常（unchecked）：编译器不强制处理的异常，通常是`RuntimeException`及其子类。通常来说就是程序逻辑错误（bug），应在编程的时候尽可能避免。



3. 异常要怎么处理

1. 自己处理
    1. Try-catch
    2. try-finally
    3. Try-catch-finally
    4. Try with resources (jdk7新增的语法糖)
2. 交给别人处理
    1. throws 异常类(Exception)
    2. 交给调用者
    3. 最外层调用者是jvm（printStackTrace，栈的调用链）

4. 自定义异常

异常本质上也是一个类，跟其他类的区别在于 需要继承 Exception

作用： 用于控制业务逻辑，如果不想让程序继续往下运行，可以通过 throw 抛出去一个异常。

五、注解和反射

1. 注解（Annotation）

定义：注解，也称之为元数据（Metadata），它是一种附加到代码上（方法、接口、类、字段、参数、包）的特殊标记，用于为代码提供额外的信息。这些信息可以在编译时被编译器使用，也可以在运行时被JVM或其他程序（通过反射机制）读取和处理。

注解本身并不影响代码的执行逻辑。

作用：

1. 提供信息给编译器/人类：相当于用来做一些辅助标识，如果不满足注解所定义的规则，会给出来一些提示。
2. 编译时和部署时产生一些影响：软件工具可以处理注解信息，用来生成代码（比如Lombok通过注解@Data自动生成private属性的getter和setter方法）
3. 运行时产生一些影响：程序可以在运行时通过反射读取注解信息，从而实现特定的逻辑。（常见于Spring、JUnit等框架中）

1.1 注解的分类

java中的注解主要可以分为3类

1. 内置注解：java自带的一些注解。常见的有 `Override`,`FunctionInterface`,`Deprecated`,`SuppressWarnings`

- override：用于标记一个方法是重写父类或接口中的方法，作用域：方法
- FunctionInterface：用于标记一个接口是函数式接口（只包含一个抽象方法的接口），作用域：接口
- Deprecated：标记这个方法已经过时了，不建议使用，在未来的新版本发布的时候，可能会删除这个方法
- SuppressWarnings：抑制警告。 比如抑制过时警告：`@SuppressWarnings({"deprecation"})`

2. 元注解

专门用来注解其他注解的注解。它们定义了自定义注解的行为。

- Target：指定被修饰的注解可以应用于哪些java元素（类、方法、字段、接口）
- Documented：文档化。表明注解应该被保留到 javadoc生成的API文档中。
- Retention：保留策略，表示注解应该在哪个级别被保留下来（生效）
    - source：源码级别，编译的时候就会丢弃它（class文件中不会存在这个注解）。主要用于代码分析工具。
    - class（默认）：注解会被保留到class文件中，但是在运行期不会保留。主要是用于字节码操作工具。
    - runtime：运行时也能读取到这些注解，这是最常用的保留策略，用于在运行时对程序进一步处理。（很多的框架都是利用这个机制，Spring全家桶）

1.2 自定义注解及注解的使用（反射）

自定义注解

1. 通过 @interface 定义一个注解
2. 定义注解的使用范围（`java.lang.annotation.ElementType`）和保留策略
3. （可选）可以定义这个注解的额外属性（如果有额外的业务逻辑的话）

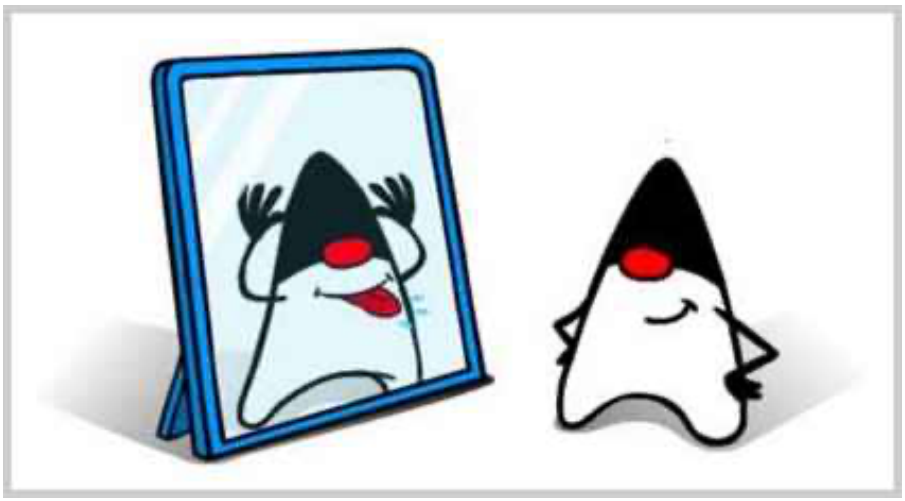
使用

通常是结合反射（和切面）来实现自己的业务逻辑。

1.3 注解的常用使用场景

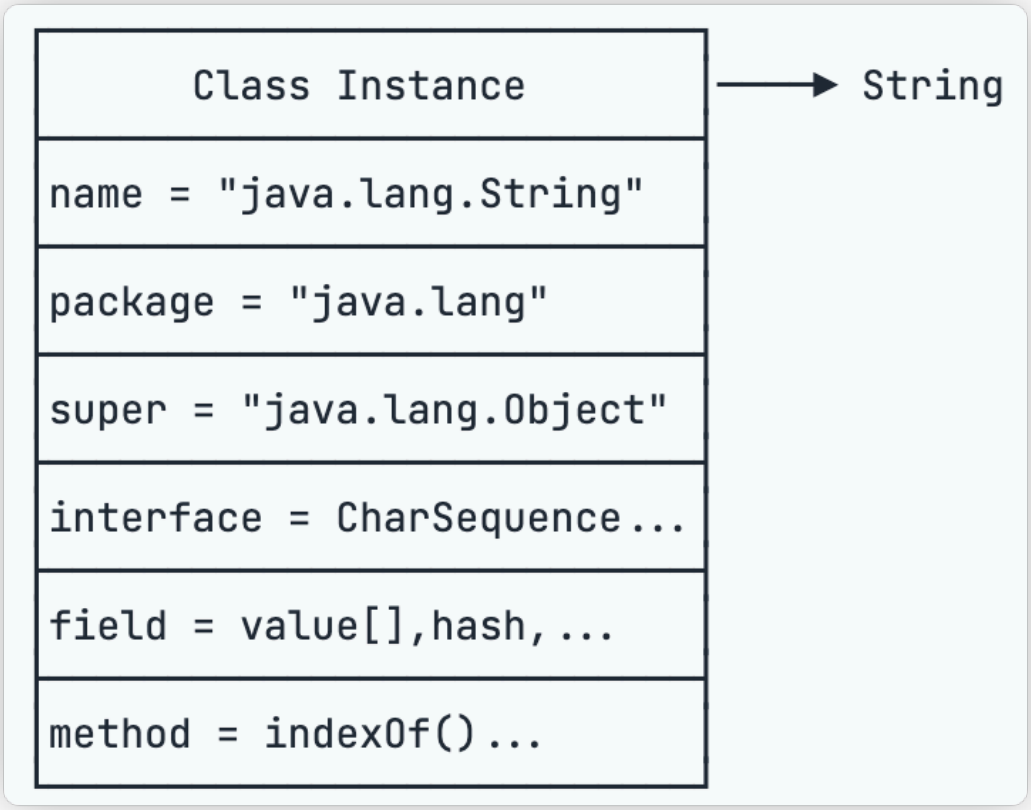
- 编译器的指令识别：override、Deprecated、SuppressWarnings （静态识别，编译前）
- 代码生成： lombok （编译期间）
- 框架配置和依赖注入：Spring框架等 （运行时）
- 文档生成相关：Swagger（编译期间）

2. 反射



反射允许程序在**运行时**动态的获取自身的信息以及操作任意类的属性和方法。换句话说，对于任意一个类，我们在运行时都能够知道这个类的所有属性和方法（包含private的），都能够调用他的任意方法和属性，这个就是java的反射机制。

2.1 Class对象



Class是一个对象，是我们反射机制的入口，JVM为每个加载到内存的 `.class` 文件创建一个对应的 `java.lang.Class` 类的对象（实例），通过这个class对象，我们就可以访问到封装了该类的运行时的所有信息（相当于看到了镜子）

2.2 获取 class 对象的方式

4中方式：

1. **直接使用 .class获取**
2. 通过调用 对象的.getClass方法
3. **使用 Class.forName("全限定名") 来获取class对象。 Class.forName也常用来让jvm加载某个类**
4. 类加载器：classloader （仅作为了解）

2.3 通过反射获取类的信息

包括类名、包、父类、接口、方法、属性等

2.4 通过反射实例化对象、调用方法

```
public class Demo6_DynamicObject {
    public static void main(String[] args) throws InvocationTargetException,
InstantiationException, IllegalAccessException, NoSuchMethodException {

        Class<ReflectablePerson> class1 = ReflectablePerson.class;
        Constructor<?>[] declaredConstructors = class1.getDeclaredConstructors();
        for (Constructor<?> declaredConstructor : declaredConstructors) {
```

```
            if (declaredConstructor.getParameterTypes().length == 2) {
                // 两个参数的构造函数
                /*
                   public ReflectablePerson(String name, int age) {
                        this.name = name;
                        this.age = age;
                    }
                 */
                ReflectablePerson zhangsan =
(ReflectablePerson)declaredConstructor.newInstance("zhangsan", 18);
                zhangsan.introduce();
                // 如果想调用 私有化方法，可以通过反射的方式
                Method getInformation = class1.getDeclaredMethod("getInformation");
                getInformation.setAccessible(true); // 让这个私有化方法可以访问（绕过了权限）
                Object result = getInformation.invoke(zhangsan);
                // zhangsan.getInformation();
                System.out.println(result);

            }
        }
    }
}
```

这种方式就是大多数框架底层的原理。

代理模式（静态代理和动态代理）

六、IO

I：input 输入

O：output 输出

IO是指程序与外部设备（如文件、内存、网络）之间进行数据交换的过程。

分类：

- 字节流：以字节为数据处理单位，适用于一些特殊文件（如图片、音视频、二进制文件等）
- 字符流：以字符为数据处理单位，适用于一些文本类型的数据（如 txt、java）
- 缓冲流：提高读写效率，在流的基础上增加缓冲区 （用了装饰器设计模式）
- NIO：新一代IO。通常采用 Paths、FIles等工具类进行操作。

七、网络编程

大数据： 分布式系统（通过网络来连接）

1. 网络基础概念

- **IP地址**：网络中唯一标识一台主机的地址，如 192.168.1.100
- **端口号**：主机上用于区分不同进程的逻辑编号，范围0~65535
- **协议**：通信规则，常见有 TCP（面向连接，可靠）、UDP（无连接，速度快）

2. Java 网络编程常用类

- `InetAddress`：表示IP地址
- `Socket`：客户端套接字，负责发起连接和通信
- `ServerSocket`：服务器端套接字，负责监听端口、接收连接
- `URL`/`HttpURLConnection`：HTTP协议通信

3. Socket 通信原理（基于TCP的）

- Socket（套接字）是网络通信的端点，分为客户端和服务器端
- 通信流程：
  1. 服务器端创建 ServerSocket，监听端口
  2. 客户端创建 Socket，连接服务器
  3. 双方通过输入输出流进行数据交换（对等传输）
  4. 通信结束后关闭连接

4. 注意事项

网络编程中，需要用到ip、域名、端口等概念，如果本身不了解这些概念，需要自行上网学习。

当我们要启动服务的时候，需要确保要使用的端口，在当前服务器没有其他应用程序占用。

还有，网络通信通常要控制超时时间，比如1分钟（网络攻防）

八、多线程

1. 进程和线程

- **进程 (Process)**：是操作系统进行资源分配和调度的基本单位。简单来说，一个正在运行的应用程序就是一个进程。每个进程都有自己独立的内存空间。
- **线程 (Thread)**：是进程中的一个执行单元（执行路径），是 CPU 调度和分派的基本单位。一个进程可以包含多个线程，它们共享进程的内存空间。

2. 为什么使用多线程

- 提高cpu的利用率：四核八线程的CPU，如果是单线程的应用，cpu利用率就很低。
- 提高程序的响应速度：比如音乐播放器，可以同时提供歌词、音乐播放、搜索音乐等功能。

多线程的缺点：

- 编码复杂，可能造成逻辑错误（线程安全问题）
- 多线程的切换是需要代价 （并发和并行的概念）

3. 创建线程的方式

1. 继承 `Thread` 类，重写 run方法

要注意，使用的时候，是调用thread的 start 方法，而不是run方法。

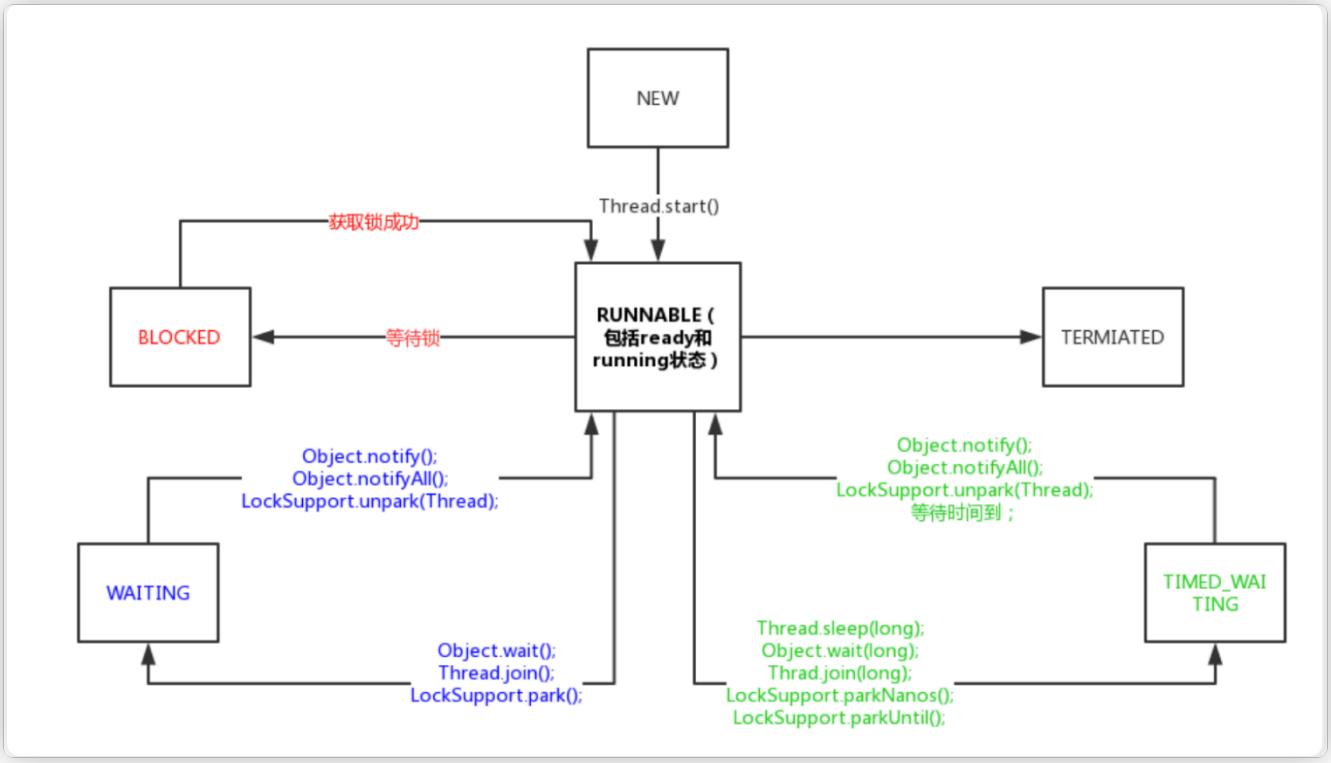
2. 继承 `Runnable` 接口 ，重写 run 方法，使用的时候通过 `new Thread(实例).start()` 做一层包装。

这种方式更常用一点，因为解决了无法多继承的问题

4. 线程的生命周期

Java 线程在其生命周期中会经历以下几种状态：

1. **新建 (NEW)**：当使用 `new` 关键字创建了一个 `Thread` 对象后，线程就处于新建状态。此时它仅仅是一个 Java 对象，操作系统还没有为其分配资源。
2. **就绪 (RUNNABLE)**：当调用线程对象的 `start()` 方法后，线程进入就绪状态。此时，线程已经获取了除 CPU 之外的所有必要资源，等待 CPU 的调度执行。处于就绪状态的线程位于“可运行线程池”中。
3. **运行 (RUNNING)**：当 CPU 调度器选中一个处于就绪状态的线程时，该线程进入运行状态，开始执行其 `run()` 方法中的代码。
4. **阻塞 (BLOCKED / WAITING / TIMED_WAITING)**：线程在运行过程中，可能会因为某些原因暂时放弃 CPU 的使用权，进入阻塞状态。阻塞状态可以细分为：
    - **BLOCKED**：线程等待获取一个监视器锁（例如，进入 `synchronized` 同步块/方法时）。
    - **WAITING**：线程无限期等待另一个线程执行特定操作。例如，调用了 `Object.wait()`、`Thread.join()` 或 `LockSupport.park()`。
    - **TIMED_WAITING**：线程在指定的时间内等待。例如，调用了 `Thread.sleep(long millis)`、`Object.wait(long timeout)`、`Thread.join(long millis)`、`LockSupport.parkNanos()` 或 `LockSupport.parkUntil()`。
5. **终止 (TERMINATED)**：当线程的 `run()` 方法执行完毕，或者因未捕获的异常而退出时，线程进入终止状态。此时线程已经结束执行。

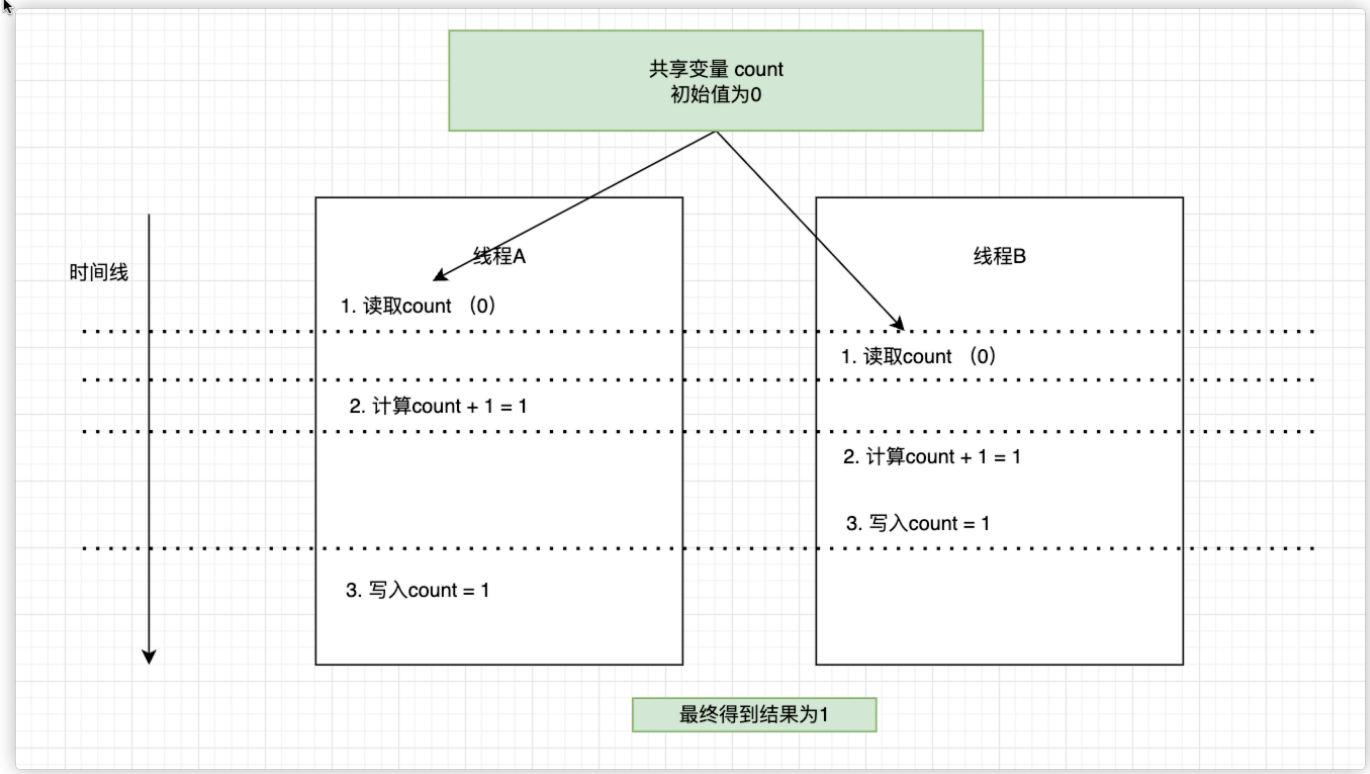


5. 线程安全问题

当多个线程同时访问和修改共享资源的时候，如果没有采取适当的（同步）措施，可能会导致数据不一致或者状态混乱的问题，这就是线程安全问题。

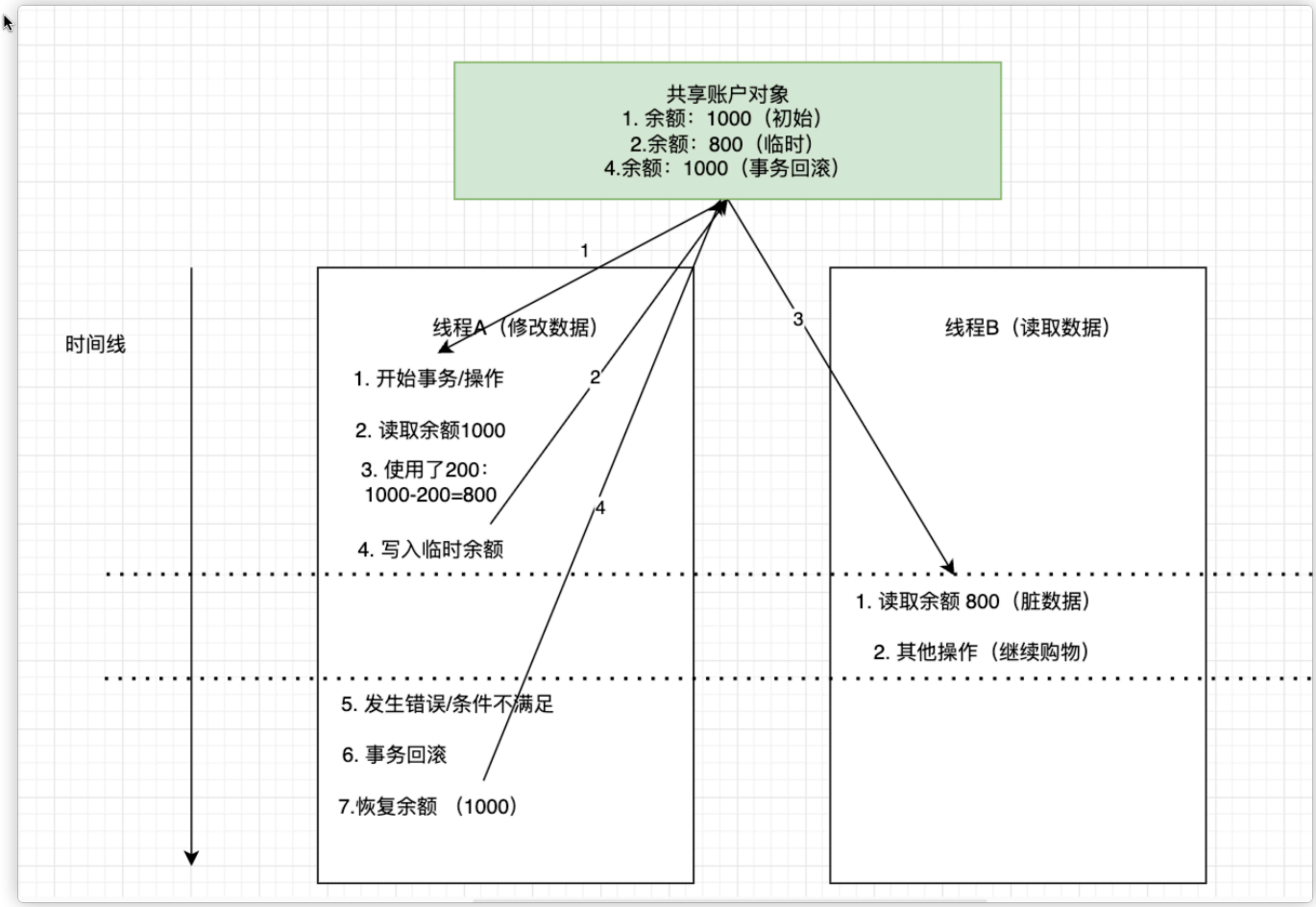
1. 竞态条件：当多个线程同时访问和操作同一个**共享数据**时，最终的结果取决于这些线程执行操作的特定顺序。如果执行顺序不可控，就可能导致非预期的结果，这种情况称为竞态条件。 最经典的例子就是对共享变量进行 "读取-修改-写入" 操作。

> ⚠ **Warning**
>
> **场景：** 两个线程（线程A 和 线程B）同时对一个共享变量 `count` (初始值为 0) 进行自增操作 (`count++`)。我们期望的最终结果是 2。
>
> **`count++` 的分解步骤：**
>
> 1. 读取 `count` 的当前值。
> 2. 将读取到的值加 1。
> 3. 将计算后的新值写回 `count`。



2. 数据脏读：一个线程读取到另外一个线程修改，但是尚未提交的数据。

> ⚠ **Warning**
>
> **场景：** 线程A 正在修改一个共享对象（例如，一个用户的账户信息），它先修改了余额，但后续可能还有其他操作，或者整个操作可能因为某个条件不满足而需要回滚。在线程A完成所有操作并“提交”（逻辑上确认这些修改有效）之前，线程B 读取了这个对象的余额。



6. 线程同步

既然多线程会出现线程安全问题，那怎么解决呢?

方式一：通过`synchronized`关键字对共享变量增加保护，只允许同一时刻、只有一个线程可以执行特定的操作（对共享资源的操作），内部其实也是锁，只不过这个锁我们看不到

可以是同步方法，也可以是同步代码块，区别在于 过`synchronized`关键字修饰的范围

方式二：通过锁机制，保证只允许同一时刻、只有一个线程可以执行特定的操作（对共享资源的操作），内部其实也是锁，只不过这个锁我们看不到

ReentrantLock（可重入锁）

两种方式的区别：

1. 灵活程度上来说，lock更灵活
2. 编程来说，`synchronized`更简单一点（底层是jvm帮我们管理和释放锁）

3. 性能上来说：locak通常会比`synchronized`的方式更优，尤其是高竞争环境下

7. 线程安全类

HashMap，ArrayList，HashSet这些都是线程不安全的。

java提供了对应的线程安全类，都在`java.util.concurrent`包下。

比如

- `java.util.concurrent.ConcurrentHashMap`：线程安全的HashMap，性能不如HashMap
- `java.util.concurrent.CopyOnWriteArrayList`
- `java.util.concurrent.CopyOnWriteArraySet`

8. 线程池

频繁地创建和销毁线程会带来显著的开销。线程池通过维护一组预先创建好的线程，来管理和复用线程，从而：

- **降低资源消耗**：复用已有线程，减少线程创建和销毁的开销。
- **提高响应速度**：任务到达时，无需等待线程创建即可立即执行。
- **提高线程的可管理性**：可以统一管理、分配、调优和监控线程。
- **控制并发数**：可以限制系统中并发线程的数量，防止资源耗尽。

`Executors` 提供了一些便捷的工厂方法来创建不同类型的线程池：

- **`newFixedThreadPool(int nThreads)`**：创建固定大小的线程池。
  - 核心线程数和最大线程数相等，都是 `nThreads`。
  - 使用无界的 `LinkedBlockingQueue` 作为任务队列。
  - 当所有线程都在忙时，新任务会在队列中等待。
  - 适用于需要限制并发线程数量的场景。
- **`newCachedThreadPool()`**：创建可缓存的线程池。
  - 核心线程数为 0，最大线程数为 `Integer.MAX_VALUE`。
  - 使用 `SynchronousQueue` 作为任务队列（不存储元素，直接传递）。
  - 当有新任务时，如果有空闲线程则复用，否则创建新线程。
  - 空闲线程超过 60 秒会被回收。
  - 适用于执行大量、耗时短的异步任务。
  - **注意**：可能创建大量线程，导致资源耗尽。
- **`newSingleThreadExecutor()`**：创建单线程的执行器。
  - 只有一个工作线程。
  - 使用无界的 `LinkedBlockingQueue`。

    - 保证所有任务按提交顺序串行执行。
    - 适用于需要保证任务顺序执行的场景。
- **`newScheduledThreadPool(int corePoolSize)`**：创建支持定时及周期性任务执行的线程池。
    - 核心线程数为 `corePoolSize`，最大线程数为 `Integer.MAX_VALUE`。
    - 使用 `DelayedWorkQueue`。