Java基础知识

概念

Java的特点

- 跨平台性
  - Java编译器将源代码编译成字节码（bytecode），该字节码可以在任何安装了Java虚拟机（JVM）的系统上运行。
- 面向对象
  - Java是一门严格的面向对象编程语言，几乎一切都是对象。面向对象编程（OOP）特性使得代码更易于维护和重用，包括类（class）、对象（object）、继承（inheritance）、多态（polymorphism）、抽象（abstraction）和封装（encapsulation）。
- 垃圾回收机制
  - Java有自己的垃圾回收机制，自动管理内存和回收不再使用的对象。这样，开发者不需要手动管理内存，从而减少内存泄漏和其他内存相关的问题。

Java为什么是跨平台的

- 我们编写的Java源码，编译后会生成一种 .class 文件，称为字节码文件。Java虚拟机就是负责将字节码文件翻译成特定平台下的机器码然后运行。也就是说，只要在不同平台上安装对应的JVM，就可以运行字节码文件，运行我们编写的Java程序。
- 跨平台的是Java程序，不是JVM。JVM是用C/C++开发的，是编译后的机器码，不能跨平台，不同平台下需要安装不同版本的JVM。

JVM、JDK、JRE三者关系

- JVM是Java虚拟机，是Java程序运行的环境。它负责将Java字节码（由Java编译器生成）解释或编译成机器码，并执行程序。JVM提供了内存管理、垃圾回收、安全性等功能，使得Java程序具备跨平台性。
- JDK是Java开发工具包，是开发Java程序所需的工具集合。它包含了JVM、编译器（javac）、调试器（jdb）等开发工具，以及一系列的类库（如Java标准库和开发工具库）。JDK提供了开发、编译、调试和运行Java程序所需的全部工具和环境。
- JRE是Java运行时环境，是Java程序运行所需的最小环境。它包含了JVM和一组Java类库，用于支持Java程序的执行。JRE不包含开发工具，只提供Java程序运行所需的运行环境。

为什么Java解释和编译都有?

- 编译：
  - javac将Java源文件编译为字节码文件（.class文件）
  - JIT将字节码中的热门代码编译为机器码，直接在CPU运行，并会保存在进程内存中（code cache）
- 解释性
  - JVM逐条解释代码，边翻译边生成机器码上CPU运行，运行完不会保存
- JIT编译的启动速度慢，但运行一段时间后速度较快；JVM解释启动快，但遇到重复代码较多时速度较慢。

编译型语言和解释型语言的区别?

- 编译型语言：在程序执行之前，整个源代码会被编译成机器码或者字节码，生成可执行文件。执行时直接运行编译后的代码，速度快，但跨平台性较差。
- 解释型语言：在程序执行时，逐行解释执行源代码，不生成独立的可执行文件。通常由解释器动态解释并执行代码，跨平台性好，但执行速度相对较慢。

数据类型

Java 数据类型一览表

分类	数据类型	占用内存（字节）	取值范围 / 精度	默认值	说明
整数型	byte	1	-128 ~ 127	0	最小的整数类型，适合节省内存
	short	2	-32,768 ~ 32,767	0	比 byte 大一档的整数
	int	4	-2^31 ~ 2^31-1	0	默认的整数类型
	long	8	-2^63 ~ 2^63-1	0L	表示长整型，需加 `L` 后缀
浮点型	float	4	约 ±3.40282347E+38F（6~7 位小数精度）	0.0f	单精度浮点，需加 `f` 后缀
	double	8	约 ±1.79769313486231570E+308（15~16 位精度）	0.0d	双精度浮点，默认浮点类型
字符型	char	2	单个 Unicode 字符（'\u0000' ~ '\uffff'）	'\u0000'	支持存储中文、英文及符号
布尔型	boolean	1（规范未固定）	true / false	false	表示逻辑值
引用型	类（Class）	-	-	null	由类创建的对象
	接口（Interface）	-	-	null	实现多态的契约

分类	数据类型	占用内存（字节）	取值范围 / 精度	默认值	说明
	数组（Array）	-	-	null	存放相同类型数据的集合
	枚举（Enum）	-	-	null	固定常量的集合
	注解（Annotation）	-	-	null	用于元数据描述

> 💡 注意：
>
> - **基本数据类型** 存储在栈上，速度快。
> - **引用数据类型** 存储在堆上，变量保存对象的地址（引用）。
> - boolean 的存储大小在 JVM 规范中未固定，但实际实现通常用 1 个字节或更小的位来表示。

数据类型转换方式

- 自动类型转换（隐式转换）：当目标类型的范围大于源类型时，Java会自动将源类型转换为目标类型，不需要显式的类型转换。例如，将int转换为long、将float转换为double等。
- 强制类型转换（显式转换）：当目标类型的范围小于源类型时，需要使用强制类型转换将源类型转换为目标类型。这可能导致数据丢失或溢出。例如，将long转换为int、将double转换为int等。语法为：目标类型 变量名 = (目标类型) 源类型。
- 字符串转换：Java提供了将字符串表示的数据转换为其他类型数据的方法。例如，将字符串转换为整型int，可以使用Integer.parseInt()方法；将字符串转换为浮点型double，可以使用Double.parseDouble()方法等。
- 数值之间的转换：Java提供了一些数值类型之间的转换方法，如将整型转换为字符型、将字符型转换为整型等。这些转换方式可以通过类型的包装类来实现，例如Character类、Integer类等提供了相应的转换方法。

类型互转会出现什么问题

基本数据类型

- 小范围到大范围（向上转型）：没有问题，一般是安全的
- 大范围到小范围（向下转型）：需要进行强制类型转换，且容易丢失高位或者精度

对象引用转换

- 向上转型：子类的对象赋值给父类的引用，是安全的
- 向下转型：父类的对象赋值给子类的引用，需要手动进行，且存在风险，如果实例实际上不是子类的实例，则会抛出ClassCastException异常，解决方式是需要使用 instanceof 检查：

```
if (animal instanceof Dog) {
Dog dog = (Dog) animal;
} // 只有确认animal是Dog的实例时才进行转型 }
```

为什么用bigDecimal 不用double

- double使用二进制表示，只能表示1/2^n的任意和，会有精度问题

装箱和拆箱

- 装箱：基本数据类型 → 对应的包装类对象
  - 自动装箱：编译器自动完成，例如`Integer i=5;`
  - 手动装箱：`Integer b = Integer.valueOf(5);`
- 拆箱：包装类对象 → 对应的基本数据类型
  - 自动拆箱：编译器自动完成，例如：`Integer a = 5;int c = a; // Integer → int（自动拆箱）`
  - 手动拆箱：调用包装类的取值方法：`int d = a.intValue();`
- 记忆口诀
  - “装进盒子是装箱，从盒子里取出来是拆箱” 基本类型进盒子，成了对象；对象开盒子，变回基本类型。

装箱装箱弊端

- 当循环里有自动拆箱时，会有额外对象生成

Java为什么要有Integer

泛型中的应用

- Java中泛型只能用引用类型，不能用基本数据类型

转换中的应用

- 在Java中，基本类型和引用类型不能直接进行转换，必须使用包装类来实现。例如，将一个int类型的值转换为String类型，必须首先将其转换为Integer类型，然后再转换为String类型。

工具方法支持

- 包装类封装了很多实用的静态方法和常量，方便类型转换、比较、解析，比如Integer.parseInt("123")、Double.isNaN(x)

那为什么还要保留int类型

- 包装类是引用类型，对象的引用和对象本身是分开存储的，而对于基本类型数据，变量对应的内存块直接存储数据本身。
- 因此，基本类型数据在读写效率方面，要比包装类高效。除此之外，在64位JVM上，在开启引用压缩的情况下，一个Integer对象占用16个字节的内存空间，而一个int类型数据只占用4字节的内存空间，前者对空间的占用是后者的4倍。
- 也就是说，不管是读写效率，还是存储效率，基本类型都比包装类高效。

说一下 integer的缓存

- Java的Integer类内部实现了一个静态缓存池，用于存储特定范围内的整数值对应的Integer对象。
- 默认情况下，这个范围是-128至127。当通过Integer.valueOf(int)方法创建一个在这个范围内的整数对象时，并不会每次都生成新的对象实例，而是复用缓存中的现有对象，会直接从内存中取出，不需要新建一个对象。

```
//==比地址
Integer a = Integer.valueOf(100);
Integer b = Integer.valueOf(100);
System.out.println(a == b); // 输出 true，因为复用了缓存池中的对象
Integer a = Integer.valueOf(200);
Integer b = Integer.valueOf(200);
System.out.println(a == b); // 输出 false，因为创建了两个不同的对象
```

面向对象

怎么理解面向对象?

- 面向对象是一种编程方法，将现实的具体事物抽象为对象，对象封装数据和行为，通过消息（方法调用）彼此协作

面向对象三大特性

- 封装：封装是指将对象的属性（数据）和行为（方法）结合在一起，对外隐藏对象的内部细节，仅通过对象提供的接口与外界交互。封装的目的是增强安全性和简化编程，使得对象更加独立。
- 继承：继承是一种可以使得子类自动共享父类数据结构和方法的机制。它是代码复用的重要手段，通过继承可以建立类与类之间的层次关系，使得结构更加清晰。
- 多态：多态是指允许不同类的对象对同一消息作出响应。即同一个接口，使用不同的实例而执行不同操作。多态性可以分为编译时多态（重载）和运行时多态（重写）。它使得程序具有良好的灵活性和扩展性。

多态体现在哪几个方面?

- 方法重载：
  - 方法重载是指同一类中可以有多个同名方法，它们具有不同的参数列表（参数类型、数量或顺序不同）。虽然方法名相同，但根据传入的参数不同，编译器会在编译时确定调用哪个方法。
  - 示例：对于一个 add 方法，可以定义为 add(int a, int b) 和 add(double a, double b)。
- 方法重写：
  - 方法重写是指子类能够提供对父类中同名方法的具体实现。在运行时，JVM会根据对象的实际类型确定调用哪个版本的方法。这是实现多态的主要方式。
  - 示例：在一个动物类中，定义一个 sound 方法，子类 Dog 可以重写该方法以实现 bark，而 Cat 可以实现 meow。
- 接口与实现： 多态也体现在接口的使用上，多个类可以实现同一个接口，并且用接口类型的引用来调用这些类的方法。这使得程序在面对不同具体实现时保持一贯的调用方式。
  - 示例：多个类（如 Dog, Cat）都实现了一个 Animal 接口，当用 Animal 类型的引用来调用 makeSound 方法时，会触发对应的实现。
- 向上转型和向下转型：
  - 在Java中，可以使用父类类型的引用指向子类对象，这是向上转型。通过这种方式，可以在运行时期采用不同的子类实现。
  - 向下转型是将父类引用转回其子类类型，但在执行前需要确认引用实际指向的对象类型以避免 ClassCastException。

多态解决了什么问题

- 多态可以提高代码的扩展性和复用性，是很多设计模式、设计原则、编程技巧的代码实现基础。

面向对象的设计原则

- 单一职责原则（S）：一个类只负责一项职责（一个变化原因）
- 开闭原则（O）：对扩展开放，对修改关闭
- 里氏替换原则（L）：所有父类都可以被子类替换
- 接口隔离原则（I）：多个专用接口优于一个万能大接口
- 依赖倒置原则（D）：高层模块依赖抽象，底层模块也依赖于抽象

重载与重写有什么区别?

- 重载（Overloading）指的是在同一个类中，可以有多个同名方法，它们具有不同的参数列表（参数类型、参数个数或参数顺序不同），编译器根据调用时的参数类型来决定调用哪个方法。
- 重写（Overriding）指的是子类可以重新定义父类中的方法，方法名、参数列表和返回类型必须与父类中的方法一致，通过@override注解来明确表示这是对父类方法的重写。
- 重载是指在同一个类中定义多个同名方法，而重写是指子类重新定义父类中的方法。

抽象类和普通类区别

对比维度	抽象类 (abstract class)	普通类 (concrete class)
是否可实例化	❌ 不能直接创建对象	✅ 可以直接 `new` 对象
是否可包含抽象方法	✅ 可以（未实现的方法，只有方法签名）	❌ 不允许包含抽象方法
是否必须实现所有方法	❌ 子类实现抽象类时，必须实现其所有抽象方法（除非子类也是抽象类）	✅ 自己实现了所有方法
构造方法	✅ 可以有构造方法，用于子类调用	✅ 可以有构造方法
成员	可包含成员变量、已实现方法、抽象方法	可包含成员变量和已实现方法
继承限制	只能单继承，但可实现多个接口	同样只能单继承，可实现多个接口
修饰符	必须用 `abstract` 修饰	无需 `abstract` 修饰
用途	定义通用模板，部分方法延迟到子类实现	定义具体可用的类，提供完整实现

Java抽象类和接口的区别是什么

抽象类

- 抽象类用于描述类的共同特性和行为，可以有成员变量、构造方法和具体方法。适用于有明显继承关系的场景。
- 关键字：extends
- 抽象类可以有定义与实现，方法可在抽象类中实现
- 访问修饰符：抽象类中成员变量默认default，可在子类中被重新定义，也可被重新赋值；抽象方法被abstract修饰，不能被private、static、synchronized和native等修饰，必须以分号结尾，不带花括号。
- 抽象类可以包含实例变量和静态变量。

接口

- 接口用于定义行为规范，可以多实现，只能有常量和抽象方法（Java 8 以后可以有默认方法和静态方法）。适用于定义类的能力或功能。
- 关键字：implements
- 接口只有定义，不能有方法的实现，java 1.8中可以定义default方法体
- 访问修饰符：接口成员变量默认为public static final，必须赋初值，不能被修改；其所有的成员方法都是public、abstract的。
- 接口只能包含常量（即静态常量）。

接口里面可以定义哪些方法

抽象方法

- 抽象方法是接口的核心部分，所有实现接口的类都必须实现这些方法。抽象方法默认是 public 和 abstract，这些修饰符可以省略。

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

默认方法

- 默认方法是在 Java 8 中引入的，允许接口提供具体实现。实现类可以选择重写默认方法

```
public interface Animal {
    void makeSound();
    default void sleep() {
        System.out.println("Sleeping...");
    }
}
```

静态方法

- 静态方法也是在 Java 8 中引入的，它们属于接口本身，可以通过接口名直接调用，而不需要实现类的对象。

```
public interface Animal {
    void makeSound();
    static void staticMethod() {
        System.out.println("Static method in interface");
    }
}
```

私有方法

- 私有方法是在 Java 9 中引入的，用于在接口中为默认方法或其他私有方法提供辅助功能。这些方法不能被实现类访问，只能在接口内部使用。

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

    default void sleep() {
        System.out.println("Sleeping...");
        logSleep();
    }

    private void logSleep() {
        System.out.println("Logging sleep");
    }
}
```

接口可以包含构造函数吗

- 在接口中，不可以有构造方法,在接口里写入构造方法时，编译器提示：Interfaces cannot have constructors，因为接口不会有自己的实例的，所以不需要有构造函数。
- 为什么呢？构造函数就是初始化class的属性或者方法，在new的一瞬间自动调用，那么问题来了Java的接口，都不能new 那么要构造函数干嘛呢？根本就没法调用

Java中的静态变量和静态方法

静态变量

- 在类中使用static关键字声明的变量。它们属于类而不是任何具体的对象。
- 所有该类的实例共享同一个静态变量。如果一个实例修改了静态变量的值，其他实例也会看到这个更改。
- 静态变量在类被加载时初始化，只会对其进行一次分配内存。
- 静态变量可以直接通过类名访问，也可以通过实例访问，但推荐使用类名。

静态方法

- 静态方法也属于类，而不是任何具体的对象。
- 静态方法可以在没有创建类实例的情况下调用。对于静态方法来说，不能直接访问非静态的成员变量或方法，因为静态方法没有上下文的实例。
- 静态方法可以直接调用其他静态变量和静态方法，但不能直接访问非静态成员。
- 静态方法不支持重写（Override），但可以被隐藏（Hide）（子类声明了一个与父类静态方法完全相同签名的静态方法）。

非静态内部类和静态内部类的区别

- 非静态内部类依赖于外部类的实例，而静态内部类不依赖于外部类的实例。
- 非静态内部类可以访问外部类的实例变量和方法，而静态内部类只能访问外部类的静态成员。
- 非静态内部类不能定义静态成员，而静态内部类可以定义静态成员。
- 非静态内部类在外部类实例化后才能实例化，而静态内部类可以独立实例化。
- 非静态内部类可以访问外部类的私有成员，而静态内部类不能直接访问外部类的私有成员，需要通过实例化外部类来访问。

非静态内部类可以直接访问外部方法，编译器是怎么做到的

- 非静态内部类可以直接访问外部方法是因为编译器在生成字节码时会为非静态内部类维护一个指向外部类实例的引用。
- 这个引用使得非静态内部类能够访问外部类的实例变量和方法。编译器会在生成非静态内部类的构造方法时，将外部类实例作为参数传入，并在内部类的实例化过程中建立外部类实例与内部类实例之间的联系，从而实现直接访问外部方法的功能。

关键字

final的作用

- 修饰类：当final修饰一个类时，表示这个类不能被继承，是类继承体系中的最终形态。例如，Java 中的String类就是用final修饰的，这保证了String类的不可变性和安全性，防止其他类通过继承来改变String类的行为和特性。
- 修饰方法：用final修饰的方法不能在子类中被重写。比如，java.lang.Object类中的getClass方法就是final的，因为这个方法的行为是由 Java 虚拟机底层实现来保证的，不应该被子类修改。
- 修饰变量：当final修饰基本数据类型的变量时，该变量一旦被赋值就不能再改变。例如，final int num = 10;，这里的num就是一个常量，不能再对其进行重新赋值操作，否则会导致编译错误。对于引用数据类型，final修饰意味着这个引用变量不能再指向其他对象，但对象本身的内容是可以改变的。例如，final StringBuilder sb = new StringBuilder("Hello");，不能让sb再指向其他StringBuilder对象，但可以通过sb.append(" World");来修改字符串的内容。

深拷贝和浅拷贝

区别

浅拷贝

- 浅拷贝是指只复制对象本身和其内部的值类型字段，但不会复制对象内部的引用类型字段。换句话说，浅拷贝只是创建一个新的对象，然后将原对象的字段值复制到新对象中，但如果原对象内部有引用类型的字段，只是将引用复制到新对象中，两个对象指向的是同一个引用对象。

深拷贝

- 深拷贝是指在复制对象的同时，将对象内部的所有引用类型字段的内容也复制一份，而不是共享引用。换句话说，深拷贝会递归复制对象内部所有引用类型的字段，生成一个全新的对象以及其内部的所有对象。

实现深拷贝的方式

实现 Cloneable 接口并重写 clone() 方法

- 在 clone() 方法中，通过递归克隆引用类型字段来实现深拷贝。

使用序列化和反序列化

- 通过将对象序列化为字节流，再从字节流反序列化为对象来实现深拷贝。要求对象及其所有引用类型字段都实现 Serializable 接口。

手动递归复制

- 针对特定对象结构，手动递归复制对象及其引用类型字段。适用于对象结构复杂度不高的情况。

对象

java创建对象有哪些方式

new关键字

```
MyClass obj = new MyClass();
```

使用Class类的newInstance()方法

- 通过反射机制，可以使用Class类的newInstance()方法创建对象。

```
MyClass obj = (MyClass) Class.forName("com.example.MyClass").newInstance();
```

使用Constructor类的newInstance()方法

- 同样是通过反射机制，可以使用Constructor类的newInstance()方法创建对象。

```
Constructor<MyClass> constructor = MyClass.class.getConstructor();
MyClass obj = constructor.newInstance();
```

使用clone()方法

- 如果类实现了Cloneable接口，可以使用clone()方法复制对象。

```
MyClass obj1 = new MyClass();
MyClass obj2 = (MyClass) obj1.clone();
```

使用反序列化

- 通过将对象序列化到文件或流中，然后再进行反序列化来创建对象。

```
// SerializedObject.java
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("object.ser"));
out.writeObject(obj);
out.close();

// DeserializedObject.java
ObjectInputStream in = new ObjectInputStream(new FileInputStream("object.ser"));
MyClass obj = (MyClass) in.readObject();
in.close();
```

New出的对象什么时候回收

- 通过过关键字new创建的对象，由Java的垃圾回收器（Garbage Collector）负责回收。

垃圾回收方法

引用计数法

- 某个对象的引用计数为0时，表示该对象不再被引用，可以被回收。

可达性分析算法

- 从根对象（如方法区中的类静态属性、方法中的局部变量等）出发，通过对象之间的引用链进行遍历，如果存在一条引用链到达某个对象，则说明该对象是可达的，反之不可达，不可达的对象将被回收。

终结器

- 如果对象重写了finalize()方法，垃圾回收器会在回收该对象之前调用finalize()方法，对象可以在finalize()方法中进行一些清理操作。然而，终结器机制的使用不被推荐，因为它的执行时间是不确定的，可能会导致不可预测的性能问题。

如何获取私有对象

使用公共访问器方法（getter 方法）

- 如果类的设计者遵循良好的编程规范，通常会为私有成员变量提供公共的访问器方法（即 getter 方法），通过调用这些方法可以安全地获取私有对象。

反射机制

- 反射机制允许在运行时检查和修改类、方法、字段等信息，通过反射可以绕过 private 访问修饰符的限制来获取私有对象。

```
import java.lang.reflect.Field;

class MyClass {
    private String privateField = "私有字段的值";
}

public class Main {
    public static void main(String[] args) throws NoSuchFieldException,
IllegalAccessException {
        MyClass obj = new MyClass();
        // 获取 Class 对象
        Class<?> clazz = obj.getClass();
        // 获取私有字段
```

```
        Field privateField = clazz.getDeclaredField("privateField");
        // 设置可访问性
        privateField.setAccessible(true);
        // 获取私有字段的值
        String value = (String) privateField.get(obj);
        System.out.println(value);
    }
}
```

反射

什么是反射

反射概念

- Java 反射机制是在运行状态中，对于任意一个类，都能够知道这个类中的所有属性和方法，对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 Java 语言的反射机制。

特性

- 运行时类信息访问：反射机制允许程序在运行时获取类的完整结构信息，包括类名、包名、父类、实现的接口、构造函数、方法和字段等。
- 动态对象创建：可以使用反射API动态地创建对象实例，即使在编译时不知道具体的类名。这是通过Class类的newInstance()方法或Constructor对象的newInstance()方法实现的。
- 动态方法调用：可以在运行时动态地调用对象的方法，包括私有方法。这通过Method类的invoke()方法实现，允许你传入对象实例和参数值来执行方法。
- 访问和修改字段值：反射还允许程序在运行时访问和修改对象的字段值，即使是私有的。这是通过Field类的get()和set()方法完成的。

反射用到的地方

实例化Bean

- 将程序中所有XML或properties配置文件加载入内存
- Java类里面解析xml或者properties里面的内容，得到对应实体类的字节码字符串以及相关的属性信息
- 通过 clazz.getDeclaredConstructor().newInstance() 创建实例
- 动态配置实例的属性

注解

- 通过 clazz.isAnnotationPresent() / clazz.getAnnotation() 获取注解实例
- 调用注解方法拿到属性值（比如 @RequestMapping 的路径、@Value 的配置值）

注解

讲一讲Java注解的原理

- 注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。
- 我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

注解的底层实现

定义一个注解

- 注解本质上是一种特殊的接口，它继承自 java.lang.annotation.Annotation 接口，所以注解也叫声明式接口

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Task {
    String name();
    int priority() default 5;
}
```

- 根据注解的作用范围，Java注解分为：
  - 源码级：仅存在源码中，编译后丢弃`@Retention(RetentionPolicy.SOURCE)`
  - 类文件级别注解：保留在`.class`文件中，但运行时不可见`@Retention(RetentionPolicy.CLASS)`
  - 运行时注解：保留在`.class`文件中，并且可以通过反射在运行时访问`@Retention(RetentionPolicy.RUNTIME)`
- 只有运行时注解使用反射机制进行解析

在类上使用注解

```
@Task(name = "ExampleTask", priority = 10)
public class Example {
    // 类的业务逻辑...
}
```

触发反射API读取注解

```
public class AnnotationReader {
    public static void main(String[] args) {
        Class<Example> cls = Example.class;

        // 触发：检查注解是否存在
        if (cls.isAnnotationPresent(Task.class)) {
            // 触发：获取注解实例
            Task task = cls.getAnnotation(Task.class);

            // 使用注解值
            System.out.printf(
                "Task name: %s, priority: %d%n",
                task.name(),
                task.priority()
            );
        } else {
            System.out.println("No Task annotation on class Example.");
```

```
        }
    }
}
```

- 调用 isAnnotationPresent(Task.class) 时，JVM 会查找 RuntimeVisibleAnnotations 属性。
- 调用 getAnnotation(Task.class) 时，JVM 调用 AnnotationParser 解析字节码，生成一个动态代理实例。

Java编译器在生成的.class文件中保存注解信息，并进行分类

- RuntimeVisibleAnnotations： 存储运行时可见的注解信息。
- RuntimeInvisibleAnnotations： 存储运行时不可见的注解信息。
- RuntimeVisibleParameterAnnotations 和RuntimeInvisibleParameterAnnotations： 存储方法参数上的注解信息。
- 类加载阶段，.class 文件中携带的注解字节数据随着字节码一起进入内存。
- 当 isAnnotationPresent 或 getAnnotation 被调用时，JVM 才会：
  - 从 RuntimeVisibleAnnotations 属性提取原始字节流
  - 用 sun.reflect.annotation.AnnotationParser 生成属性名→属性值的映射
  - 基于 Proxy 动态创建一个实现 Task 接口的注解实例
- 注解实例会被缓存，下次读取同一个注解不再重复解析。

总结

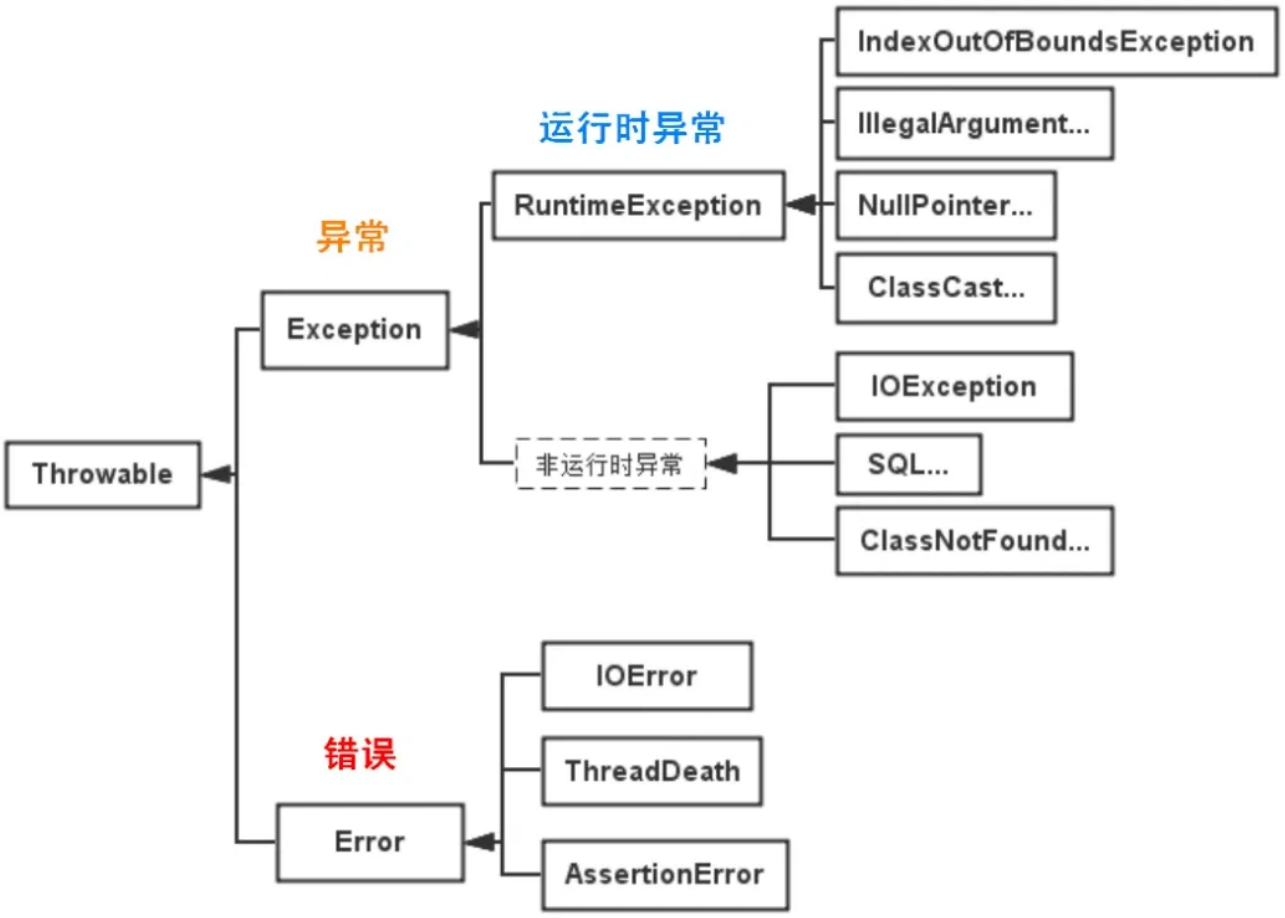
- 因此，注解解析的底层实现主要依赖于 Java 的反射机制和字节码文件的存储。通过 @Retention 元注解可以控制注解的保留策略，当使用 RetentionPolicy.RUNTIME 时，可以在运行时通过反射 API 来解析注解信息。在 JVM 层面，会从字节码文件中读取注解信息，并创建注解的代理对象来获取注解的属性值。

Java注解的作用域

- 类级别作用域：用于描述类的注解，通常放置在类定义的上面，可以用来指定类的一些属性，如类的访问级别、继承关系、注释等。
- 方法级别作用域：用于描述方法的注解，通常放置在方法定义的上面，可以用来指定方法的一些属性，如方法的访问级别、返回值类型、异常类型、注释等。
- 字段级别作用域：用于描述字段的注解，通常放置在字段定义的上面，可以用来指定字段的一些属性，如字段的访问级别、默认值、注释等。

异常

介绍一下Java异常



- Error（错误）：表示运行时环境的错误。错误是程序无法处理的严重问题，如系统崩溃、虚拟机错误、动态链接失败等。通常，程序不应该尝试捕获这类错误。例如，OutOfMemoryError、StackOverflowError等。
- Exception（异常）：表示程序本身可以处理的异常条件。异常分为两大类：
  - 非运行时异常：这类异常在编译时期就必须被捕获或者声明抛出。它们通常是外部错误，如文件不存在（FileNotFoundException）、类未找到（ClassNotFoundException）等。非运行时异常强制程序员处理这些可能出现的问题，增强了程序的健壮性。
  - 运行时异常：这类异常包括运行时异常（RuntimeException）和错误（Error）。运行时异常由程序错误导致，如空指针访问（NullPointerException）、数组越界（ArrayIndexOutOfBoundsException）等。运行时异常是不需要在编译时强制捕获或声明的。

异常处理方式

try-catch语句

```
public void readFile(String path) {
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(path));
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
    } catch (FileNotFoundException e) {
        System.err.println("文件未找到: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("读取出错: " + e.getMessage());
    } finally {
        if (reader != null) {
```

```
            try {
                reader.close();
            } catch (IOException ignored) { }
        }
    }
}
```

- finally语句无论是否发生异常都会执行
- try块中的代码将按顺序执行，如果抛出异常，将在catch块中进行匹配和处理，然后程序将继续执行catch块之后的代码。如果没有匹配的catch块，异常将被传递给上一层调用的方法。

throws关键字

- 在方法签名上使用 throws，把受检异常抛给调用者处理，简化当前方法逻辑。
- 只有“非运行时异常”需要在方法签名中声明，运行时异常可以不声明
- 调用方必须显式捕获或继续声明抛出

throw语句

- 在代码中利用 throw 手动抛出一个异常实例，常用于校验参数或状态。

```
public int divide(int a, int b) {
    if (b == 0) {
        throw new IllegalArgumentException("除数不能为零");
    }
    return a / b;
}
```

- throw 后必须是 Throwable 的子类实例
- 用于主动报告错误或非法状态

抛出异常为什么不用throws

- Unchecked Exceptions：未检查异常（unchecked exceptions）是继承自RuntimeException类或Error类的异常，编译器不强制要求进行异常处理。因此，对于这些异常，不需要在方法签名中使用throws来声明。示例包括NullPointerException、ArrayIndexOutOfBoundsException等。
- 异常被捕获和处理：另一种常见情况是，在方法内部捕获了可能抛出的异常，并在方法内部处理它们，而不是通过throws子句将它们传递到调用者。这种情况下，方法可以处理异常而无需在方法签名中使用throws。

try{return “a”} fianlly{return “b”}这条语句返回啥

- 当执行到 return "a" 时，JVM 会先把要返回的值（即字符串常量 "a"）暂存起来，然后进入 finally 块。
- 在 finally 中又执行了 return "b"，这会直接覆盖之前暂存的返回值。方法最后返回的，就是 "b"。

Object

== 与 equals 有什么区别

- 对于字符串来说，==比较值，equals比较地址
- 对于非字符串来说，如果没有进行重写，==和equals作用相同，都是比较地址

hashcode和equals方法有什么关系

- 在 Java 中，对于重写 equals 方法的类，通常也需要重写 hashCode 方法，并且需要遵循以下规定：
  - 若equals方法返回true，则hashCode值必须相等
  - 若hashCode值相等，equals方法不一定，如果不相等，则为哈希冲突

String、StringBuffer、StringBuilder的区别和联系

特性	String	StringBuffer	StringBuilder
可变性	不可变（Immutable），每次修改生成新对象	可变（Mutable），在原对象上直接修改	可变（Mutable），在原对象上直接修改
线程安全性	天然线程安全	同步（方法级 `synchronized`），线程安全	非同步，线程不安全
性能	拼接时不断创建新对象，性能最差	有同步锁开销，性能中等	无同步锁开销，性能最好
使用场景	字符串常量、少量拼接、用作 Map Key （静态字符串）等	多线程环境下需要频繁修改字符串（多线程动态字符串）	单线程或方法内部频繁拼接（单线程动态字符串）
联系	实现 `CharSequence`、`Appendable` 接口，基于字符数组实现，提供类似 `length()`、`charAt()`、`substring()` 等 API	实现 `CharSequence`、`Appendable` 接口，基于字符数组实现，提供类似 `length()`、`charAt()`、`substring()` 等 API	实现 `CharSequence`、`Appendable` 接口，基于字符数组实现，提供类似 `length()`、`charAt()`、`substring()` 等 API

Java新特性

Java8新特性

新特性	描述
Lambda 表达式	支持函数式编程，简化匿名内部类和回调
函数式接口	使用 `@FunctionalInterface` 标记仅含一个抽象方法的接口
Stream API	声明式集合处理，支持链式调用及并行流（`java.util.stream`）
Optional 类	包装可能为 `null` 的值，避免显式的空指针检查
默认方法 & 静态方法	接口中可定义带实现的默认方法和静态方法，增强向后兼容性
重复注解	使用 `@Repeatable` 支持对同一个程序元素多次使用同一注解

新特性	描述
CompletableFuture	异步编程增强，支持链式、非阻塞的任务组合与回调
并行数组操作	`Arrays.parallelSort`、`parallelPrefix` 等方法支持多核并行处理

函数式接口

内容

- 函数式接口是只包含一个抽象方法的接口。Lambda 表达式必须绑定到一个函数式接口上
- 可以有默认方法和静态方法和从Object类继承来的public方法
- 通常用 @FunctionalInterface 注解标识（可选，但推荐）

Java内置的四个函数式接口

接口名	参数类型	返回类型	作用说明
`Function<T, R>`	T	R	接收一个参数，返回一个结果。常用于数据转换、映射等操作。
`Consumer<T>`	T	void	接收一个参数，不返回结果。常用于打印、保存、处理数据等副作用操作。
`Supplier<T>`	无	T	不接收参数，返回一个结果。常用于延迟加载、生成数据等。
`Predicate<T>`	T	boolean	接收一个参数，返回布尔值。常用于条件判断、过滤等。

Lambda 表达式

内容

- Lambda 表达式它是一种简洁的语法，用于创建匿名函数，主要用于简化函数式接口（只有一个抽象方法的接口）的使用。
- 传统的匿名内部类实现方式代码较为冗长，而 Lambda 表达式可以用更简洁的语法实现相同的功能。

```
//以前的匿名内部类
public class AnonymousClassExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Running using anonymous class");
            }
        });
        t1.start();
    }
}
//lambda表达式
public class LambdaExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> System.out.println("Running using lambda
expression"));
```

```
        t1.start();
    }
}
```

- 虽然有很多好处，但是增加了调试困难。因为 Lambda 表达式是匿名的，在调试时很难定位具体是哪个 Lambda 表达式出现了问题。

语法

- (parameters) -> expression：当 Lambda 体只有一个表达式时使用，表达式的结果会作为返回值。
- (parameters) -> { statements; }：当 Lambda 体包含多条语句时，需要使用大括号将语句括起来，若有返回值则需要使用 return 语句。

Java中stream的API介绍一下

- Stream 是对集合对象功能的增强，它允许你以声明式方式处理数据。
- 它提供了一种高效且易于使用的数据处理方式，特别适合集合对象的操作，如过滤、映射、排序等。Stream API 不仅可以提高代码的可读性和简洁性，还能利用多核处理器的优势进行并行处理。

```
//选出长度大于三的串
List<String> originalList = Arrays.asList("apple", "fig", "banana", "kiwi");
List<String> filteredList = originalList.stream()
                                        .filter(s -> s.length() > 3)
                                        .collect(Collectors.toList());
//求数组中元素的和
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .mapToInt(Integer::intValue)
                 .sum();
```

Java流的并行API

基本概念

- 并行流（ParallelStream）就是将源数据分为多个子流对象进行多线程操作，然后将处理的结果再汇总为一个流对象，底层是使用通用的 fork/join 池来实现，即将一个任务拆分成多个“小任务”并行计算，再把多个“小任务”的结果合并成总的计算结果

底层机制

- 使用 Java 的 ForkJoinPool.commonPool（默认线程数为 CPU 核心数）
- 自动将数据拆分为多个任务
- 每个任务在不同线程中执行
- 最终将结果合并

使用场景

- CPU密集的任务而不是IO密集的任务，因为IO密集的任务会导致线程阻塞等待资源返回，CPU利用率低

completableFuture怎么用的

- CompletableFuture实现了两个接口：Future、CompletionStage。
- Future用于表示异步计算的结果，只能通过阻塞或者轮询的方式获取结果，而且不支持设置回调方法，Java 8之前若要设置回调一般会使用guava的ListenableFuture，回调的引入又会导致臭名昭著的回调地狱（下面的例子会通过ListenableFuture的使用来具体进行展示）。
- CompletableFuture对Future进行了扩展，可以通过设置回调的方式处理计算结果，同时也支持组合操作，支持进一步的编排，同时一定程度解决了回调地狱的问题。
    - 回调地狱：在处理多个异步任务时，每个任务都依赖前一个任务的结果，导致回调函数层层嵌套，这种结构不仅难读、难维护，还容易出错

```
doA(resultA -> {
    doB(resultB -> {
        doC(resultC -> {
            // 处理最终结果，嵌套很多，导致回调地狱
        });
    });
});
```

- CompletableFuture使用链式组合，避免嵌套，提供了 thenApply、thenCompose、thenAccept 等方法，可以像流水线一样组合异步任务
- CompletionStage提供了三种方法（CompletableFuture是它的具体实现类）：
    - thenApply —— 对结果进行转换（同步）
        - 接收上一个任务的结果，进行处理并返回一个新的值。
        - 返回的是一个新的 CompletableFuture，但处理逻辑是同步的。
    - thenCompose —— 链式异步任务（扁平化）
        - 接收上一个任务的结果，并返回一个新的 CompletableFuture（异步任务）。
    - thenAccept —— 消费结果（无返回值）
        - 接收上一个任务的结果，进行处理，但不返回新值。

```
public static void main(String[] args) {
        CompletableFuture<Void> workflow = getUserAsync("user123")
            .thenCompose(user -> getOrdersAsync(user))                  // 获取订单
            .thenCompose(orders -> getLogisticsAsync(orders))          // 获取物流
            .thenAccept(logistics -> {
                // ✅ 处理最终结果
                System.out.println("物流信息：" + logistics);
            })
            .exceptionally(ex -> {
                // ❌ 集中处理异常
                System.err.println("处理失败：" + ex.getMessage());
                return null;
            });

        workflow.join(); // 等待流程执行完毕
        executor.shutdown();
    }
```

Java21新特性

- 虚拟线程：这是 Java 21 引入的一种轻量级并发的新选择。它通过共享堆栈的方式，大大降低了内存消耗，同时提高了应用程序的吞吐量和响应速度。可以使用静态构建方法、构建器或ExecutorService来创建和使用虚拟线程。
- Scoped Values（范围值）：提供了一种在线程间共享不可变数据的新方式，避免使用传统的线程局部存储，促进了更好的封装性和线程安全，可用于在不通过方法参数传递的情况下，传递上下文信息，如用户会话或配置设置。

序列化

怎么把一个对象从一个jvm转移到另一个jvm

将对象序列化

- 把对象转成字节流（可以是 JSON、XML 或 Java 原生二进制），通过网络/文件传到另一 JVM，再反序列化成新的对象。这可以通过 Java 的 ObjectOutputStream 和 ObjectInputStream 来实现。对象类需要实现 Serializable接口

远程过程调用

- 通过远程方法调用，将对象的调用和数据传到另一个 JVM。

持久化中转

- 对象先持久化到数据库/缓存系统（如 Redis、Kafka），另一 JVM 读取并反序列化

消息队列

- 对象转成消息（序列化），通过 RabbitMQ、ActiveMQ、Kafka 等中间件在 JVM 之间传送

序列化和反序列化让你自己实现你会怎么做

Java的原生序列化有很多弊端

- 无法跨语言： Java 序列化目前只适用基于 Java 语言实现的框架，其它语言大部分都没有使用 Java 的序列化框架，也没有实现 Java 序列化这套协议。因此，如果是两个基于不同语言编写的应用程序相互通信，则无法实现两个应用服务之间传输对象的序列化与反序列化。
- 容易被攻击：Java 序列化是不安全的，我们知道对象是通过在 ObjectInputStream 上调用 readObject() 方法进行反序列化的，这个方法其实是一个神奇的构造器，它可以将类路径上几乎所有实现了 Serializable 接口的对象都实例化。这也就意味着，在反序列化字节流的过程中，该方法可以执行任意类型的代码，这是非常危险的。
- 序列化后的流太大：序列化后的二进制流大小能体现序列化的性能。序列化后的二进制数组越大，占用的存储空间就越多，存储硬件的成本就越高。如果我们是进行网络传输，则占用的带宽就更多，这时就会影响到系统的吞吐量。

使用json

- 使用Fastjson

将对象转为二进制字节流

对象类需要实现Serializable接口

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

序列化为字节数组

```
// 序列化：对象 -> 字节数组
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(p);
oos.flush();
byte[] bytes = bos.toByteArray();
```

- 创建输出流：先用 ByteArrayOutputStream（写到内存）或者 FileOutputStream（写到文件）作为基础流。
- 包装为对象输出流：使用 ObjectOutputStream 将普通输出流封装成可以处理对象的流。
- 写出对象：调用 writeObject(Object obj)，将对象转换成字节序列并写入底层流。
- 关闭资源：调用 close() 释放系统资源。（或者try-with-resources自动关闭）

反序列化为对象

```
// 反序列化：字节数组 -> 对象
ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
ObjectInputStream ois = new ObjectInputStream(bis);
Person restored = (Person) ois.readObject();
```

- 确保类可用：反序列化时需要该类的 .class 文件
- 创建输入流：先用 ByteArrayInputStream（内存读取）或 FileInputStream（文件读取）作为基础流。
- 包装为对象输入流：使用 ObjectInputStream 将普通输入流封装成可以读取对象的流。
- 读取对象：调用 readObject() 方法，将字节序列转回 Java 对象。需要强制类型转换 (YourClass)。
- 关闭资源：调用 close() 释放系统资源。

设计模式

单例模式

- 无论在代码的哪个位置调用，都得到同一个对象。
- 将构造方法用private修饰，这样能够防止外部实例化。
- 通过getInstance()方法获取唯一实例

```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {}  // 私有构造方法，防止外部实例化

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();  // 第一次调用时创建
        }
        return instance;
    }
}
```

单例模式多线程下出现的问题

不使用volatile：指令重排序

- 编译器和CPU为了提高执行效率而做的优化：它会在不改变单线程语义的前提下，调整指令的执行顺序，让处理器的流水线、缓存等硬件资源发挥到极致。
- 当创建单例的实例时，会执行：`instance = new Singleton();`，分为三步：
  - 分配内存空间
  - 调用构造函数初始化对象
  - 将内存地址赋值给 instance 引用
- 指令重排序可能会变成
  - 分配内存空间
  - 将内存地址赋值给 instance
  - 调用构造函数初始化对象
- 这样就导致另一个线程第一次检查 instance != null 时，可能拿到的是一个未初始化的对象。

不用synchronized

- 竞态创建：多个线程同时通过 null 判断，重复执行构造，产生多个实例，违反单例唯一性。
- 半初始化可见：new 并非原子：分配内存 → 赋值给引用 → 执行构造。指令重排序可能让“赋值给引用”先于“执行构造”，其他线程读到非 null 的引用，却拿到未完全构造的对象。
- 可见性缺失：一个线程把 instance 设为非 null 后，其他线程未必立刻可见，可能继续创建或读到旧状态。

解决方式：volatile+双重检查锁（DCL）

```java
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {           // 第一次检查
            synchronized (Singleton.class) {
                if (instance == null) {   // 第二次检查
                    instance = new Singleton();
                }
            }
```

```
        }
        return instance;
    }
}
```

- volatile 的作用
  - 禁止指令重排序：对象在创建过程中可能会被编译器或 CPU 重排序，导致其他线程看到一个“未完全初始化”的对象。
  - 保证可见性：一个线程对 instance 的修改，对其他线程立即可见。

> 没有 volatile，即使用 synchronized 也可能遇到“半初始化”的情况。

- synchronized 的作用
  - 保证互斥访问：同一时间只有一个线程能进入临界区（创建实例的代码）。
  - 避免重复创建实例：确保第一次初始化后不会被并发干扰。

I/O

BIO、NIO、AIO区别是什么

BIO（Blocking IO）

- 就是传统的 java.io 包，它是基于流模型实现的，交互的方式是同步、阻塞方式，也就是说在读入输入流或者输出流时，在读写动作完成之前，线程会一直阻塞在那里，它们之间的调用是可靠的线性顺序。优点是代码比较简单、直观；缺点是 IO 的效率和扩展性很低，容易成为应用性能瓶颈。

NIO（Non-Blocking IO）

- Java 1.4 引入的 java.nio 包，提供了 Channel、Selector、Buffer 等新的抽象，可以构建多路复用的、同步非阻塞 IO 程序，同时提供了更接近操作系统底层高性能的数据操作方式。Selector代替了线程本身轮询IO事件，避免了阻塞同时减少了不必要的线程消耗；非阻塞的核心就是通道和缓冲区，当IO事件就绪时，可以通过写到缓冲区，保证IO的成功，而无需线程阻塞式地等待。
  - Channel（通道）
    - 类似流（Stream），但 Channel 同时支持读和写。
  - Buffer（缓冲区）
    - 数据读写通过 Buffer 进行。
  - Selector（选择器）
    - 核心组件，用于监听多个通道的事件，如连接就绪、读就绪、写就绪等。通过多路复用机制实现一个线程管理多个通道。用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。
- 实现NIO的框架
  - Netty 的 I/O 模型是基于非阻塞 I/O 实现的，底层依赖的是 NIO 框架的多路复用器 Selector。采用 epoll 模式后，只需要一个线程负责 Selector 的轮询。当有数据处于就绪状态后，需要一个事件分发器（Event Dispather），它负责将读写事件分发给对应的读写事件处理器（Event Handler）。

AIO（Asynchronous IO）

- 是 Java 1.7 之后引入的包，是 NIO 的升级版本，提供了异步非堵塞的 IO 操作方式，所以人们叫它 AIO（Asynchronous IO），异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。

其他

有一个学生类，想按照分数排序，再按学号排序，应该怎么做?

```
@Override
    public int compareTo(Student other) {
        // 1) 分数降序
        int byScoreDesc = Integer.compare(other.score, this.score);
        if (byScoreDesc != 0) return byScoreDesc;

        // 2) 学号升序（注意：如果学号是“纯数字且不补零”，可考虑按数值比较）
        return this.id.compareTo(other.id);
    }
```

- 实现Comparable接口
- 重写compareTo方法
- Integer.compare：前大于后返回正数，等于返回0，小于返回负数

```
List<Student> list = Arrays.asList(
    new Student("2023003", 90),
    new Student("2023001", 95),
    new Student("2023002", 90),
    new Student("2023004", 80)
);
// 直接排序（使用天然顺序）
Collections.sort(list);
System.out.println(list);
// 输出顺序：score 降序，同分按 id 升序
```

Native方法解释一下

```
public class NativeExample {
    public native void nativeMethod();
}
```

- 在Java类中，native方法看起来与其他方法相似，只是其方法体由native关键字代替，没有实际的实现代码。

实现Native方法步骤

- 生成NI头文件：使用javah工具从你的Java类生成C/C++的头文件，这个头文件包含了所有native方法的原型。
- 编写本地代码：使用C/C++编写本地方法的实现，并确保方法签名与生成的头文件中的原型匹配。
- 编译本地代码：将C/C++代码编译成动态链接库（DLL，在Windows上），共享库（SO，在Linux上）
- 加载本地库：在Java程序中，使用System.loadLibrary()方法来加载你编译好的本地库，这样JVM就能找到并调用native方法的实现了。

集合概念

数组与集合区别，用过哪些?

	数组	集合
大小	固定长度，一旦创建长度无法改变	动态长度，可以根据需要动态增加或减少元素
元素	基本数据类型或对象	对象
访问方式	直接索引访问	通过迭代器或者方法

一些Java集合

- ArrayList： 动态数组，实现了List接口，支持动态增长。
- LinkedList： 双向链表，也实现了List接口，支持快速的插入和删除操作。
- HashMap： 基于哈希表的Map实现，存储键值对，通过键快速查找值。
- HashSet： 基于HashMap实现的Set集合，用于存储唯一元素。
- TreeMap： 基于红黑树实现的有序Map集合，可以按照键的顺序进行排序。
- LinkedHashMap： 基于哈希表和双向链表实现的Map集合，保持插入顺序或访问顺序。
- PriorityQueue： 优先队列，可以按照比较器或元素的自然顺序进行排序。

说说Java集合

```
Collection
├── List
|   ├── ArrayList
|   ├── LinkedList
|   └── Vector
├── Set
|   ├── HashSet
|   ├── LinkedHashSet
|   └── TreeSet
└── Queue
    ├── LinkedList
    ├── PriorityQueue
    └── ArrayDeque

Map
├── HashMap
├── LinkedHashMap
├── TreeMap
└── ConcurrentHashMap
```

ArrayList

- 可变容量，数组实现，扩容时，将原数组复制，支持随机访问，增删性能较差

LinkedList

- 双向链表，增删更快，随机读取慢

HashSet

- 通过HashMap实现，HashMap的Key即HashSet存储的元素，所有Key都是用相同的Value，一个名为PRESENT的Object类型常量。使用Key保证元素唯一性，但不保证有序性。由于HashSet是HashMap实现的，因此线程不安全。

LinkedHashSet

- 继承自HashSet，通过LinkedHashMap实现，使用额外的双向链表维护元素插入顺序。

TreeSet

- 基于红黑树，通过TreeMap实现的，自动排序

HashMap

- JDK1.8 之前由数组+链表组成的，JDK1.8 以后当链表长度大于阈值（默认为 8）时，将链表转化为红黑树

LinkedHashMap

- 继承自 HashMap，额外维护一个双向链表记录顺序

TreeMap

- 红黑树（自平衡的排序二叉树）

ConcurrentHashMap

- Node数组+链表+红黑树实现，线程安全的（jdk1.8以前Segment锁，1.8以后volatile + CAS 或者 synchronized）

PriorityQueue

- 小根堆，根据优先级排序

ArrayDeque

- 基于循环数组

Java中线程安全的集合

ConcurrentHashMap

- 分段锁机制（JDK8 后使用 CAS + synchronized）

CopyOnWriteArrayList

- 写时复制机制，适合读多写少场景。每次写操作都会复制整个数组，读操作不会被写操作阻塞，写操作通过ReentrantLock加锁来确保原子性，volatile 保证新数组对所有线程立即可见。

CopyOnWriteArraySet

- 基于 CopyOnWriteArrayList 实现的线程安全 Set

ConcurrentLinkedQueue

- 非阻塞队列，基于链表。使用 CAS 实现高并发入队/出队

ConcurrentLinkedDeque

- 双端非阻塞队列，支持头尾操作

BlockingQueue

类名	特点
ArrayBlockingQueue	有界阻塞队列，数组实现
LinkedBlockingQueue	可选容量，链表实现
PriorityBlockingQueue	带优先级的阻塞队列
DelayQueue	延迟队列，元素按时间出队
SynchronousQueue	不存储元素，直接交接

Collections和Collection的区别

- Collections是工具类，提供了一系列方法对集合进行操作
- Collection是接口，定义了集合应该具备的功能

集合的遍历

- for循环
- for-each
- 迭代器

```
Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String element = iterator.next();
    System.out.println(element);
}
```

- ListIterator列表迭代器:是迭代器的子类，可以双向访问列表并在迭代过程中修改元素。

```
ListIterator<String> listIterator= list.listIterator();
while(listIterator.hasNext()) {
    String element = listIterator.next();
    System.out.println(element);
}
```

- forEach方法： Java 8引入了 forEach 方法，可以对集合进行快速遍历。

```
list.forEach(element -> System.out.println(element));
```

- Stream API： Java 8的Stream API提供了丰富的功能，可以对集合进行函数式操作，如过滤、映射等。

```
list.stream().forEach(element -> System.out.println(element));
```

list可以一边遍历一边修改元素吗

- for循环：可以修改
- for-each：可以修改值，不能改变结构，例如删除
- 迭代器可以删除，修改值需要通过迭代器的set方法，不能用list的set方法

把ArrayList变成线程安全有哪些方法

- 使用Collections类的synchronizedList方法将ArrayList包装成线程安全的List：

```
List<String> synchronizedList = Collections.synchronizedList(arrayList);
```

- 使用CopyOnWriteArrayList类代替ArrayList，它是一个线程安全的List实现：
- 使用Vector（×）

ArrayList的扩容机制说一下

- 计算新的容量：一般情况下，新的容量会扩大为原容量的1.5倍（在JDK 10之后，扩容策略做了调整），然后检查是否超过了最大容量限制。
- 创建新的数组：根据计算得到的新容量，创建一个新的更大的数组。
- 将元素复制：将原来数组中的元素逐个复制到新数组中。
- 更新引用：将ArrayList内部指向原数组的引用指向新数组。
- 完成扩容：扩容完成后，可以继续添加新元素。
- 之所以扩容是 1.5 倍，是因为 1.5 可以充分利用移位操作，减少浮点数或者运算时间和运算次数。

```
// 新容量计算
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

CopyonWriteArraylist是如何实现线程安全的

使用volatile关键字修饰数组

- 保证当前线程对数组对象重新赋值后，其他线程可以及时感知到。

```
private transient volatile Object[] array;
```

写操作加锁

- 加了一把互斥锁ReentrantLock以保证线程安全。

```
public boolean add(E e) {
    //获取锁
    final ReentrantLock lock = this.lock;
    //加锁
    lock.lock();
    try {
```

```
            //获取到当前List集合保存数据的数组
            Object[] elements = getArray();
            //获取该数组的长度（这是一个伏笔，同时len也是新数组的最后一个元素的索引值）
            int len = elements.length;
            //将当前数组拷贝一份的同时，让其长度加1
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            //将加入的元素放在新数组最后一位，len不是旧数组长度吗，为什么现在用它当成新数组的最后一个元素的下标？建议自行画图推演，就很容易理解。
            newElements[len] = e;
            //替换引用，将数组的引用指向给新数组的地址
            setArray(newElements);
            return true;
        } finally {
            //释放锁
            lock.unlock();
        }
    }
```

- 现在我们来看读操作，读是没有加锁的，所以读是一直都能读

```
public E get(int index) {
    return get(getArray(), index);
}
```

遍历map

for-each循环和entrySet方法

```
// 使用for-each循环和entrySet()遍历Map
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
}
```

for-each循环和keySet()方法

- 只遍历键

```
// 使用for-each循环和keySet()遍历Map的键
for (String key : map.keySet()) {
    System.out.println("Key: " + key + ", Value: " + map.get(key));
}
```

迭代器

```
// 使用迭代器遍历Map
Iterator<Entry<String, Integer>> iterator = map.entrySet().iterator();
    while (iterator.hasNext()) {
        Entry<String, Integer> entry = iterator.next();
        System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
    }
```

Lambda表达式和forEach()方法

```
map.forEach((key, value) -> System.out.println("Key: " + key + ", Value: " + value));
```

使用Stream API

```
// 使用Stream API遍历Map
map.entrySet().stream().forEach(entry -> System.out.println("Key: " + entry.getKey() + ",
Value: " + entry.getValue()));
```

哈希冲突的解决方案有哪些

- 链接法：使用链表或其他数据结构来存储冲突的键值对，将它们链接在同一个哈希桶中。
- 开放寻址法：在哈希表中找到另一个可用的位置来存储冲突的键值对，而不是存储在链表中。常见的开放寻址方法包括线性探测、二次探测和双重散列。
- 再哈希法（Rehashing）：当发生冲突时，使用另一个哈希函数再次计算键的哈希值，直到找到一个空槽来存储键值对。
- 哈希桶扩容：当哈希冲突过多时，可以动态地扩大哈希桶的数量，重新分配键值对，以减少冲突的概率。

Hashmap的put过程

- 计算key的hashcode
- 使用哈希函数确定数组索引
- 若桶为空，直接创建结点将值放入
- 若不为空，且key已存在，直接覆盖value
- 若不为空，key不存在，则插入到链表或者红黑树中
- 判断是否树化：链表长度大于8且数组长度大于等于64
- 判断是否扩容：若元素数量超过现在的容量*负载因子，则扩容。负载因子默认0.75。容量乘2。

Hashmap调用get方法一定安全吗

get方法

- 获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。

安全性

- 空指针异常（NullPointerException）：如果你尝试用 null 作为键调用 get 方法，而 HashMap 没有被初始化（即为 null），那么会抛出空指针异常。不过，如果 HashMap 已经初始化，使用 null 作为键是允许的，因为 HashMap 支持 null 键。
- 线程安全：HashMap 本身不是线程安全的。如果在多线程环境中，没有适当的同步措施，同时对 HashMap 进行读写操作可能会导致不可预测的行为。例如，在一个线程中调用 get 方法读取数据，而另一个线程同时修改了结构（如增加或删除元素），可能会导致读取操作得到错误的结果或抛出 ConcurrentModificationException。

HashMap为啥String适合做Key呢?

- 用 string 做 key，因为 String对象是不可变的，一旦创建就不能被修改，这确保了Key的稳定性。如果Key是可变的，可能会导致hashCode和equals方法的不一致，进而影响HashMap的正确性。

为什么使用红黑树而不是AVL树

- 平衡二叉树追求的是一种 “完全平衡” 状态：任何结点的左右子树的高度差不会超过 1，优势是树的结点是很平均分配的。当增删元素时，基本会破坏结构，导致左旋和右旋。
- 红黑树不追求这种完全平衡状态，而是追求一种 “弱平衡” 状态：整个树最长路径不会超过最短路径的 2 倍。优势是虽然牺牲了一部分查找的性能效率，但是能够换取一部分维持树平衡状态的成本，调整的频率。

重写HashMap的equal和hashcode方法需要注意什么?

- 如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。
- 如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。

重写HashMap的equal方法不当会出现什么问题

- 重写了equals方法，不重写hashCode方法时，可能会出现equals方法返回为true，而hashCode方法却返回false，这样的一个后果会导致在hashmap等类中存储多个一模一样的对象，导致出现覆盖存储的数据的问题，这与hashmap只能有唯一的key的规范不符合。

列举HashMap在多线程下可能会出现的问题

- JDK1.7中的 HashMap 使用头插法插入元素，在多线程的环境下，扩容的时候有可能导致环形链表的出现，形成死循环。因此，JDK1.8使用尾插法插入元素，在扩容时会保持链表元素原本的顺序，不会出现环形链表的问题。
- 多线程同时执行 put 操作，如果计算出来的索引位置是相同的，那会造成前一个 key 被后一个 key 覆盖，从而导致元素的丢失。此问题在JDK 1.7和 JDK 1.8 中都存在。

HashMap的扩容机制介绍一下

- hashMap默认的负载因子是0.75，即如果hashmap中的元素个数超过了总容量75%，则会触发扩容，扩容分为两个步骤：
    - 第1步是对哈希表长度的扩展（2倍）
    - 第2步是将旧哈希表中的数据放到新的哈希表中。
- 扩容后，元素要么留在原位置，要么为原位置加旧容量
    - 若元素哈希值和旧容量取与等于0，则留在原桶
    - 若不为零，则一定为旧容量，则等于旧桶+旧容量

HashMap的大小为什么是2的n次方大小呢

- 按位取与更快（与取模相比）
- 扩容时，元素的索引只高一个bit位
- 哈希分布均衡。配合 JDK 的扰动函数（高位异或低位等混合），低位随机性更好，元素分布更均匀，冲突概率下降。如果容量不是 2 的幂，hash % capacity 的分布可能受哈希值低位模式影响，容易集中冲突。

往HashMap里存20个元素，扩容几次?

- 初始容量16
- 当插入到16*0.75=12时，第13个元素进行扩容，容量为32
- 当插入到20时，20<32*0.75=24，不扩容

说说hashmap的负载因子

- HashMap 负载因子 loadFactor 的默认值是 0.75，当 HashMap 中的元素个数超过了容量的 75% 时，就会进行扩容。
- 默认负载因子为 0.75，是因为它提供了空间和时间复杂度之间的良好平衡。
- 负载因子太低会导致大量的空桶浪费空间，负载因子太高会导致大量的碰撞，降低性能。0.75 的负载因子在这两个因素之间取得了良好的平衡。

Hashmap和Hashtable

- HashMap线程不安全，效率高一点，可以存储null的key和value，null的key只能有一个，null的value可以有多个。默认初始容量为16，每次扩充变为原来2倍。创建时如果给定了初始容量，则扩充为2的幂次方大小。底层数据结构为数组+链表，插入元素后如果链表长度大于阈值（默认为8），先判断数组长度是否小于64，如果小于，则扩充数组，反之将链表转化为红黑树，以减少搜索时间。
- HashTable线程安全，效率低一点，其内部方法基本都经过synchronized修饰，不可以有null的key和value。默认初始容量为11，每次扩容变为原来的2n+1。创建时给定了初始容量，会直接用给定的大小。底层数据结构为数组+链表。它基本被淘汰了，要保证线程安全可以用ConcurrentHashMap。

ConcurrentHashMap怎么实现的

JDK 1.7及以前

- 使用Segment数组将表分段加锁
- Segment 是一种可重入锁（ReentrantLock）
- 一个 ConcurrentHashMap 里包含一个 Segment 数组，一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素。

JDK 1.8

- 通过 volatile + CAS 或者 synchronized 来实现的线程安全
- 添加元素时，若容器为空，则使用 volatile 加 CAS 来初始化
    - 若不为空，若对应的桶为空，则利用 CAS 设置该节点
    - 若桶不为空，则使用 synchronized ，然后，遍历桶中的数据，并替换或新增节点到桶中，最后再判断是否需要转为红黑树，这样就能保证并发访问时的线程安全了。
- CAS：Compare-And-Swap（比较并交换）一种无锁（lock-free）同步原语，用来在多线程环境下安全地更新共享变量。检查内存中某个变量的当前值是否等于预期值（expected）。如果相等，就把它更新为新值（new value）。如果不相等，说明有其他线程修改过了，当前操作失败，通常会重新读取并重试。

分段锁怎么加锁的

- 在 ConcurrentHashMap 中，将整个数据结构分为多个 Segment，每个 Segment 都类似于一个小的 HashMap，每个 Segment 都有自己的锁，不同 Segment 之间的操作互不影响，从而提高并发性能。

- 在 ConcurrentHashMap 中，对于插入、更新、删除等操作，需要先定位到具体的 Segment，然后再在该 Segment 上加锁，而不是像传统的 HashMap 一样对整个数据结构加锁。这样可以使得不同 Segment 之间的操作并行进行，提高了并发性能。

分段锁是可重入的吗

可重入锁

- 同一个线程在持有锁的情况下，可以再次获取该锁而不会被阻塞。这样可以避免一个方法在调用另一个需要同一把锁的方法时发生自我死锁。

分段锁是可重入锁

- 因为直接继承了 ReentrantLock，所以具备所有可重入特性。
- 当一个线程在 Segment 里调用方法时，如果这个方法内部又需要获取该 Segment 的锁，线程可以直接“重入”。
- 如果 put() 过程中需要触发扩容（resize），扩容内部也会访问受同一锁保护的结构，没有可重入特性就会自己卡住自己

已经用了synchronized，为什么还要用CAS呢

- ConcurrentHashMap使用这两种手段来保证线程安全主要是一种权衡的考虑，在某些操作中使用synchronized，还是使用CAS，主要是根据锁竞争程度来判断的。
- CAS
  - 适合一次性原子更新单个变量/引用 的简单操作（比如空桶插入、初始化、计数更新）。
  - 开销低：直接用 CPU 原子指令完成，不阻塞线程。
  - 成功时几乎零等待，能最大化发挥“多桶并行”的优势。
- synchronized
  - 用在 需要修改复杂数据结构 的场景（比如链表插入/删除、红黑树旋转、扩容时搬迁桶）。
  - 能在一个临界区内保护多个字段/节点的一致性。
  - 避免中间态被其他线程看到，保证结构性不变量。
  - 若冲突，线程直接阻塞，会有上下文切换的开销。

ConcurrentHashMap用了悲观锁还是乐观锁

CAS乐观锁

- 场景
  - 空桶插入：桶是 null 时，直接用 CAS 把它原子地替换成新节点。
  - 延迟初始化表：多个线程第一次访问时，用 CAS 确保只初始化一次。
- 特点
  - 不先加锁，先假设不会冲突（乐观）。
  - 冲突才自旋重试，不会阻塞线程。

synchronized悲观锁

- 场景
  - 桶里已有节点，需要修改链表或红黑树结构时。
  - 扩容迁移某个桶的所有节点时。

- 特点
    - 先加锁再修改，保证一个线程在临界区独占资源。
    - 避免复杂结构在更新过程中被其他线程破坏。

说一下HashMap和Hashtable、ConcurrentMap的区别

- HashMap线程不安全，效率高一点，可以存储null的key和value，null的key只能有一个，null的value可以有多个。默认初始容量为16，每次扩充变为原来2倍。创建时如果给定了初始容量，则扩充为2的幂次方大小。底层数据结构为数组+链表，插入元素后如果链表长度大于阈值（默认为8），先判断数组长度是否小于64，如果小于，则扩充数组，反之将链表转化为红黑树，以减少搜索时间。
- HashTable线程安全，效率低一点，其内部方法基本都经过synchronized修饰，不可以有null的key和value。默认初始容量为11，每次扩容变为原来的2n+1。创建时给定了初始容量，会直接用给定的大小。底层数据结构为数组+链表。它基本被淘汰了，要保证线程安全可以用ConcurrentHashMap。
- ConcurrentHashMap是Java中的一个线程安全的哈希表实现，它可以在多线程环境下并发地进行读写操作，而不需要像传统的HashTable那样在读写时加锁。ConcurrentHashMap的实现原理主要基于分段锁和CAS操作。它将整个哈希表分成了多Segment（段），每个Segment都类似于一个小的HashMap，它拥有自己的数组和一个独立的锁。在ConcurrentHashMap中，读操作不需要锁，可以直接对Segment进行读取，而写操作则只需要锁定对应的Segment，而不是整个哈希表，这样可以大大提高并发性能。

Set集合有什么特点？如何实现key无重复的?

- set集合特点：Set集合中的元素是唯一的，不会出现重复的元素。
- set实现原理：Set集合通过内部的数据结构（如哈希表、红黑树等）来实现key的无重复。当向Set集合中插入元素时，会先根据元素的hashCode值来确定元素的存储位置，然后再通过equals方法来判断是否已经存在相同的元素，如果存在则不会再次插入，保证了元素的唯一性。

有序的Set是什么？记录插入顺序的集合是什么?

- 有序的 Set 是TreeSet和LinkedHashSet。TreeSet是基于红黑树实现，保证元素的自然顺序。LinkedHashSet是基于双重链表和哈希表的结合来实现元素的有序存储，保证元素添加的自然顺序
- 记录插入顺序的集合通常指的是LinkedHashSet，它不仅保证元素的唯一性，还可以保持元素的插入顺序。当需要在Set集合中记录元素的插入顺序时，可以选择使用LinkedHashSet来实现。