

Java八股文

Java讲义

1.基础部分

面向对象三大特性

1.封装

2.继承

3.多态

重载与重写的区别

抽象类和接口的区别

final修饰的类、方法、字段分别起什么作用

Object类的一些方法

IO

泛型

异常处理

集合类

LIST

ArrayList

LinkedList

Vector

CopyOnWriteArrayList

Map

HashMap

Hashtable

ConcurrentHashMap

Set

多线程与线程安全

线程的创建方式

线程池

手写线程池

同步方式

悲观锁

乐观锁CAS

volatile

happens-before

ThreadLocal

JVM

[内存分区](#)
[线程私有](#)
[线程共享](#)
[垃圾回收机制](#)
[如何判断对象是否可回收](#)
[引用计数法](#)
[可达性分析法](#)
[垃圾收集算法](#)
[标记清除法](#)
[复制算法](#)
[标记整理法](#)
[垃圾回收器](#)
[GC调优](#)
[类加载机制](#)
[JVM常见命令](#)

Java讲义

1.基础部分

面向对象三大特性

Java 是一门面向对象编程语言，面向对象编程的三大特性是封装、继承和多态。

1.封装

封装是面向对象编程的一种基本概念，它是将数据和操作数据的方法放在一个单元中，使其成为一个独立的实体，对外部程序隐藏其实现细节，同时通过公共接口提供对该实体的访问。在 Java 中，使用类来实现封装。例如：

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

```
    }

    public int getAge() {
        return age;
    }
}
```

在这个示例中，Person 类封装了两个私有字段 name 和 age，外部程序无法直接访问它们。同时，提供了两个公共方法 getName 和 getAge，用于获取 name 和 age 的值。

2. 继承

继承是指子类可以继承父类的属性和方法，使得代码复用性更高。在 Java 中，使用 extends 关键字实现继承。例如：

```
public class Student extends Person {
    private String school;

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

    public String getSchool() {
        return school;
    }
}
```

在这个示例中，Student 类继承了 Person 类，并新增了一个私有字段 school。同时，它也继承了 Person 类中的 name 和 age 字段以及 getName 和 getAge 方法。

3. 多态

多态是指同一种类型的对象，在不同的情况下可以呈现不同的形态，即同一个方法在不同对象上可以有不同的行为。在 Java 中，多态可以通过方法重载和方法重写来实现。例如：

```
public class Animal {
    public void makeSound() {
        System.out.println("Animal is making sound.");
    }
}

public class Dog extends Animal {
```

```
    @Override
    public void makeSound() {
        System.out.println("Dog is barking.");
    }
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat is meowing.");
    }
}
```

在这个示例中，Animal 类定义了一个 makeSound 方法，而 Dog 类和 Cat 类都继承了 Animal 类，并重写了 makeSound 方法。当程序调用不同对象的 makeSound 方法时，会呈现不同的形态：

```
scssCopy codeAnimal animal = new Animal();
animal.makeSound(); // 输出 "Animal is making sound."

Dog dog = new Dog();
dog.makeSound(); // 输出 "Dog is barking."

Cat cat = new Cat();
cat.makeSound(); // 输出 "Cat is meowing."
```

总的来说，Java 的面向对象编程具有封装、继承和多态三大特性，它们使得程序更易于维护和扩展，也使得程序更加灵活和可读性更高。封装可以隐藏实现细节，减少外部程序对对象的直接操作，增加了代码的安全性和可维护性；继承可以实现代码的复用，减少代码的重复编写，提高了代码的可维护性和可扩展性；多态可以使程序更具有弹性，不同对象可以调用相同的方法但表现出不同的行为，使得程序更加灵活和适应不同的场景。

重载与重写的区别

Java中的重载（Overloading）和重写（Overriding）是两个非常重要的概念，它们经常用于面向对象编程中。虽然它们的名字很相似，但它们的含义和使用方式是不同的。

重载（Overloading）是指在同一个类中定义多个同名但参数列表不同的方法，这些方法可以有不同的返回值类型，但不能仅仅是返回类型不同。编译器根据方法的参数类型和个数来确定调用哪个方法。重载可以让程序员方便地调用一个方法，而无需关心参数类型和个数的细节。重载方法的关键在于方法的参数列表，不同的参数列表可以区分不同的方法。例如：

```
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public double add(double a, double b) {  
        return a + b;  
    }  
  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

在上面的示例中，Calculator 类定义了三个同名但参数列表不同的 add 方法。第一个 add 方法接收两个整数作为参数，第二个 add 方法接收两个浮点数作为参数，第三个 add 方法接收三个整数作为参数。这三个方法都可以被称为重载方法，它们的方法名相同，但参数列表不同。

重写（Overriding） 是指在子类中重新定义一个父类中已经存在的方法，方法名、返回值类型和参数列表都要与父类中的方法相同。重写方法的目的是为了改变方法的实现方式，但是它的方法签名必须和父类中的方法相同。如果方法的签名不同，那么就是另一个方法，而不是重写。例如：

```
public class Animal {  
    public void move() {  
        System.out.println("Animal is moving");  
    }  
}  
  
public class Dog extends Animal {  
    @Override  
    public void move() {  
        System.out.println("Dog is running");  
    }  
}
```

在上面的示例中，Animal 类定义了一个 move 方法，它输出 "Animal is moving"。Dog 类继承自 Animal 类，并重写了 move 方法，它输出 "Dog is running"。在这里，Dog 类中的 move 方法就是一个重写方法。

下面是重载和重写的区别：

- 方法名相同，但参数列表不同，是重载；方法名、参数列表和返回值类型相同，是重写。
- 重载是一个类中的方法，而重写是子类中的方法。
- 重载的目的是为了提供多个不同的方法实现，而重写的目的是为了改变方法的实现方式。
- 编译器根据方法的参数类型和个数来确定调用哪个方法，而重写是在运行时动态绑定的，即调用哪个方法取决于实对象的类型，即多态性。

特性	区别
重载	方法名相同，但参数列表不同。提供多个不同的方法实现。
重写	方法名、参数列表和返回值类型相同。改变方法的实现方式。在运行时动态绑定，即调用哪个方法取决于实对象的类型，即多态性。

下面是一个示例代码，演示了重载和重写的用法：

```

public class OverloadAndOverrideDemo {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        Cat cat = new Cat();

        // 调用重载方法
        System.out.println("调用重载方法：");
        System.out.println("2 + 3 = " + Calculator.add(2, 3));
        System.out.println("2.5 + 3.5 = " + Calculator.add(2.5, 3.5));
        System.out.println("1 + 2 + 3 = " + Calculator.add(1, 2, 3));

        // 调用重写方法
        System.out.println("\n调用重写方法：");
        animal.move();
        dog.move();
        cat.move();
    }
}

class Calculator {
    public static int add(int a, int b) {
        return a + b;
    }

    public static double add(double a, double b) {
        return a + b;
    }
}

```

```

    }

    public static int add(int a, int b, int c) {
        return a + b + c;
    }
}

class Animal {
    public void move() {
        System.out.println("Animal is moving");
    }
}

class Dog extends Animal {
    @Override
    public void move() {
        System.out.println("Dog is running");
    }
}

class Cat extends Animal {
    @Override
    public void move() {
        System.out.println("Cat is walking");
    }
}

```

输出结果如下：

```

pythonCopy code
调用重载方法：
2 + 3 = 5
2.5 + 3.5 = 6.0
1 + 2 + 3 = 6

调用重写方法：
Animal is moving
Dog is running
Cat is walking

```

在上面的示例中，我们定义了 Calculator、Animal、Dog 和 Cat 四个类，其中 Calculator 定义了三个重载方法，Animal 定义了一个 move 方法，而 Dog 和 Cat 类分别重写了 move 方法。在 OverloadAndOverrideDemo 类中，我们首先调用了 Calculator 类的三个重载方法，然后分别创建了一个 Animal、Dog 和 Cat 的对象，并调用了它们的 move 方法。从输出结果中可以看到，Calculator 类的三个重载方法都可以被正确地调用，而

Animal、Dog 和 Cat 类的 move 方法都正确地重写了父类 Animal 中的 move 方法，输出了不同的结果。

抽象类和接口的区别

抽象类和接口是 Java 中两种不同的语言机制，它们都用于描述抽象类型。抽象类和接口有很多相似之处，但也有很多不同之处。下面将详细介绍抽象类和接口的区别。

1. 定义方式不同

抽象类是用 `abstract` 关键字定义的类，它可以有抽象方法和非抽象方法。抽象方法是没有实现的方法，必须在子类中实现。非抽象方法是有实现的方法，子类可以选择是否覆盖它们。

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

接口是用 `interface` 关键字定义的类型，它只包含抽象方法和常量。接口中的方法都是抽象方法，不能有实现。常量是接口中定义的 `public static final` 变量。

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

2. 实现方式不同

抽象类是用 `extends` 关键字扩展的，一个类只能继承一个抽象类。如果一个类继承了一个抽象类，它必须实现所有的抽象方法。如果一个类没有实现所有的抽象方法，那么它必须被声明为抽象类。

```
public class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Dog is barking.");  
    }  
}
```

```
    }  
}
```

接口是用 `implements` 关键字实现的，一个类可以实现多个接口。如果一个类实现了一个接口，它必须实现接口中的所有方法。

```
public class Dog implements Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Dog is barking.");  
    }  
}
```

3. 成员变量的区别

抽象类中可以定义成员变量，可以是任何数据类型，可以有初始值，也可以没有初始值。

```
public abstract class Animal {  
    private String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public abstract void makeSound();  
}
```

接口中定义的变量都是常量，即使用 `public static final` 修饰，不能被修改。

```
public interface Animal {  
    int LEGS = 4;  
  
    void makeSound();  
}
```

4. 构造方法的区别

抽象类可以定义构造方法，可以被子类调用，用于初始化父类的成员变量。

```

public abstract class Animal {
    private String name;

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

    public String getName() {
        return name;
    }

    public abstract void makeSound();
}

```

接口不能定义构造方法，因为接口没有实现。

5. 默认方法和静态方法的区别

Java 8 引入了默认方法和静态方法，它们可以在接口中定义方法的实现。默认方法使用 `default` 关键字定义，静态方法使用 `static` 关键字定义。

默认方法可以被实现类继承或覆盖，也可以被实现类调用。静态方法只能在接口中调用。

```

public interface Animal {
    default void eat() {
        System.out.println("Animal is eating.");
    }

    static void sleep() {
        System.out.println("Animal is sleeping.");
    }

    void makeSound();
}

```

6. 适用场景不同

抽象类适合用于描述一些有共同特征的类，其中一部分方法是相同的，一部分方法是不同的，而接口适合用于描述一些没有任何关联的类，其中所有的方法都是抽象的。

抽象类	接口
用 <code>abstract</code> 关键字定义	用 <code>interface</code> 关键字定义

可以有抽象方法和非抽象方法	只包含抽象方法和常量
用 <code>extends</code> 关键字扩展	用 <code>implements</code> 关键字实现
可以定义成员变量	只能定义常量
可以定义构造方法	不能定义构造方法
可以有默认方法和静态方法	可以有默认方法和静态方法
适合用于描述有共同特征的类	适合用于描述没有任何关联的类

final修饰的类、方法、字段分别起什么作用

final 修饰的类

使用 `final` 修饰的类是不能被继承的，它是一个不可变的类。这种类通常会将其所有的成员变量都声明为 `final`，并且不提供修改这些成员变量的方法。这样做的目的是为了确保这个类的实例的状态是不可变的。

```
public final class ImmutableList {
    private final int value;

    public ImmutableList(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

在上面的示例中，`ImmutableList` 类是一个不可变的类，它的 `value` 成员变量是 `final` 的，而且只有一个构造方法可以设置它的值。因此，一旦创建了一个 `ImmutableList` 实例，它的状态就不会再改变了。

final 修饰的成员变量

使用 `final` 修饰的成员变量是一个常量，它的值不能被修改。`final` 成员变量必须在声明时或在构造方法中进行初始化。如果一个类中的某个成员变量是不会改变的，那么可以将它声明为 `final`，这样可以提高代码的可读性和可维护性。

```
public class Person {
    private final String name;
```

```
private final int age;

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

public String getName() {
    return name;
}

public int getAge() {
    return age;
}
}
```

在上面的示例中，`Person` 类中的 `name` 和 `age` 成员变量都是 `final` 的，它们在构造方法中被初始化，之后就不能再被修改了。

final 修饰的方法

使用 `final` 修饰的方法是不能被子类重写的，它是一个不可变的方法。如果一个类中的某个方法是不会被子类重写的，那么可以将它声明为 `final`，这样可以提高代码的性能和安全性。

```
public class ParentClass {
    public final void print() {
        System.out.println("ParentClass");
    }
}

public class ChildClass extends ParentClass {
    // 无法重写 print 方法
}
```

在上面的示例中，`ParentClass` 类中的 `print` 方法是 `final` 的，因此它不能被 `ChildClass` 类重写。

Object类的一些方法

clone()

`clone()` 方法用于创建并返回当前对象的一个副本，即一个新的对象，该对象的所有字段与原对象相同。在 Java 中，通过实现 `Cloneable` 接口来启用对象克隆，否则调用

`clone()` 方法会抛出 `CloneNotSupportedException` 异常。例如：

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

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class CloneDemo {
    public static void main(String[] args) throws CloneNotSupportedException {
        Person p1 = new Person("Alice", 25);
        Person p2 = (Person) p1.clone();
        System.out.println(p1.getName() + ", " + p1.getAge()); // 输出 "Alice, 25"
        System.out.println(p2.getName() + ", " + p2.getAge()); // 输出 "Alice, 25"
        System.out.println(p1 == p2); // 输出 "false"
    }
}
```

在上面的示例中，我们定义了一个 `Person` 类，并实现了 `Cloneable` 接口和 `clone()` 方法。在 `main()` 方法中，我们首先创建了一个 `Person` 对象 `p1`，然后通过 `clone()` 方法创建了一个副本 `p2`。最后，我们分别输出了 `p1` 和 `p2` 的姓名和年龄，以及判断它们是否相等。可以看到，`p1` 和 `p2` 的字段值相同，但它们是不同的对象。

wait()和notify()

`wait()` 和 `notify()` 方法用于线程的同步，它们通常与 `synchronized` 关键字一起使用。
`wait()` 方法会使当前线程进入等待状态，直到另一个线程调用该对象的 `notify()` 方法，或者指定的时间过去。`notify()` 方法用于唤醒一个正在等待该对象的线程。例如：

```

public class WaitNotifyDemo {
    public static void main(String[] args) {
        Data data = new Data();

        new Thread(() -> {
            synchronized (data) {
                try {
                    System.out.println("Thread 1 is waiting");
                    data.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread 1 got data: " + data.getData());
            }
        }).start();

        new Thread(() -> {
            synchronized (data) {
                System.out.println("Thread 2 is producing data");
                data.setData("Hello World");
                data.notify();
            }
        }).start();
    }
}

class Data {
    private String data;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }
}

```



在上面的示例中，我们首先定义了一个 `Data` 类，它有一个私有字段 `data` 和对应的 getter 和 setter 方法。然后，我们创建了两个线程，一个线程等待 `Data` 对象的 `notify`，另一个线程生产数据并调用 `notify`。可以看到，当第二个线程调用 `notify()` 方法后，第一个线程被唤醒，并获取到了新的数据。

equals()

`equals()` 方法用于比较两个对象是否相等。在 Java 中，如果一个类没有重写 `equals()` 方法，则默认使用 `Object` 类的 `equals()` 方法，即比较两个对象的引用是否相等。如果需要比较两个对象的内容是否相等，则必须重写 `equals()` 方法。例如：

```

public class EqualsDemo {
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 25);
        Person p2 = new Person("Alice", 25);
        System.out.println(p1 == p2); // 输出 "false"
        System.out.println(p1.equals(p2)); // 输出 "true"
    }
}

class Person {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public int get

```

什么是反射

反射是指程序在运行时能够访问、检测和修改它本身状态或行为的一种能力。在 Java 中，反射通常指的是程序在运行时能够访问、检测和修改 Java 类的属性、方法和构造器，而不需要在编译时确定这些信息。

Java 中的反射机制主要由以下几个类和接口组成：



- Class 类：表示一个类或接口，在运行时可以获取该类或接口的信息。
- Constructor 类：表示一个构造器。
- Method 类：表示一个方法。
- Field 类：表示一个字段。

反射的使用方法示例：

```

public class ReflectionDemo {
    public static void main(String[] args) throws Exception {
        // 获取 Class 对象
        Class<?> clazz = Class.forName("com.example.Person");

        // 创建对象

```

```
Constructor<?> constructor = clazz.getConstructor(String.class, int.class);
Object obj = constructor.newInstance("Alice", 25);

// 调用方法
Method method = clazz.getMethod("getName");
Object result = method.invoke(obj);
System.out.println(result); // 输出 "Alice"

// 修改字段
Field field = clazz.getDeclaredField("age");
field.setAccessible(true); 
field.setInt(obj, 30);
System.out.println(obj); // 输出 "Person{name='Alice', age=30}"
}

}

class Person {
    private String name;
    private int age;

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

    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;
    }

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

在上面的示例中，我们首先使用 `Class.forName()` 方法获取 `Person` 类的 `Class` 对象。然后，我们使用 `clazz.getConstructor()` 方法获取 `Person` 类的带有 `String` 和 `int` 参数的构造方法，并使用 `newInstance()` 方法创建了一个 `Person` 对象。接着，我们使用 `clazz.getMethod()` 方法获取 `Person` 类的 `getName()` 方法，并使用 `invoke()` 方法调用了该方法。最后，我们使用 `clazz.getDeclaredField()` 方法获取 `Person` 类的 `age` 字段，并使用 `setInt()` 方法修改了该字段的值。可以看到，通过反射，我们可以在运行时获取、创建、调用和修改对象的信息，从而实现一些动态的功能。

反射的优缺点

反射机制具有以下优点：

- 通过反射机制，程序可以在运行时获取类的信息，并动态地创建对象、调用方法和修改属性等，提高了程序的灵活性和可扩展性。
- 反射机制可以让程序在运行时动态地加载和使用类，不需要在编译时将所有需要用到的类都预加载进来，减少了代码的冗余和编译时的工作量。

反射机制也具有以下缺点：

- 反射机制的性能较差，因为它需要在运行时动态地获取类的信息，而这些信息在编译时是无法确定的，需要进行很多的判断和转换，这会导致程序运行速度变慢。
- 反射机制破坏了程序的封装性，因为它可以访问和修改类的私有成员，包括字段、方法和构造器等，这可能导致程序的安全性和稳定性受到影响。

反射能破坏单例模式吗？

反射机制可以破坏单例模式，因为它可以访问和修改类的私有成员，包括构造器。如果单例模式的构造器是私有的，那么反射机制可以通过 `setAccessible` 方法来访问它，并创建多个实例。为了避免这种情况，可以在单例模式的构造器中增加判断，如果已经存在一个实例，就抛出异常或返回原有的实例。以下是一个示例代码，演示了如何使用反射机制创建多个单例实例：

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
  
    private Singleton() {  
    }  
}
```

```

        public static Singleton getInstance() {
            return instance;
        }
    }

    public class ReflectionDemo {
        public static void main(String[] args) throws Exception {
            // 使用反射机制创建多个实例
            Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();
            constructor.setAccessible(true);
            Singleton instance1 = constructor.newInstance();
            Singleton instance2 = constructor.newInstance();

            // 输出实例的哈希码
            System.out.println("instance1: " + instance1.hashCode());
            System.out.println("instance2: " + instance2.hashCode());
        }
    }
}

```

在上面的示例中，我们定义了 Singleton 类作为单例模式的实现，它只能创建一个实例。在 ReflectionDemo 类中，我们使用反射机制获取 Singleton 类的构造器，并调用 newInstance 方法创建两个实例 instance1 和 instance2。由于 Singleton 类的构造器是私有的，我们需要使用 setAccessible 方法来设置访问权限。最后，我们输出了实例的哈希码，可以看到 instance1 和 instance2 的哈希码是不同的，说明它们是不同的对象。这就说明了反射机制可以破坏单例模式。

IO

字符流

Java 的字节流分为输入流和输出流，分别用于读取和写入二进制数据，可以处理任意类型的数据，包括图像、音频、视频等。

Java 中的字节流主要有以下四个类：

- InputStream：所有字节输入流的基类
- OutputStream：所有字节输出流的基类
- FileInputStream：从文件中读取字节流
- FileOutputStream：写入字节到文件中

字节流

Java 的字符流是以字符为单位进行读取和写入的，可以处理文本数据，包括普通的文本文件、XML 文件、HTML 文件等。与字节流不同，字符流可以自动处理字符编码和换行

符等问题。

Java 中的字符流主要有以下四个类：

- Reader：所有字符输入流的基类
- Writer：所有字符输出流的基类
- FileReader：从文件中读取字符流
- FileWriter：写入字符到文件中

二者的区别

Java的字节流和字符流的主要区别在于，字节流是以字节为单位进行读取和写入的，可以处理任意类型的数据，包括图像、音频、视频等，而字符流是以字符为单位进行读取和写入的，可以处理文本数据，包括普通的文本文件、XML 文件、HTML 文件等。与字节流不同，字符流可以自动处理字符编码和换行符等问题。在处理文本数据时，通常使用字符流。在处理二进制数据时，通常使用字节流。

泛型

Java 中的泛型是指在定义类、接口、方法时使用类型参数来代替具体的类型，这使得代码更加通用和可重用。使用泛型可以提高代码的安全性和可读性，同时也可以减少代码的重复编写。

泛型类

泛型类是指在定义类时使用类型参数，例如：

```
public class Box<T> {  
    private T content;  
  
    public Box(T content) {  
        this.content = content;  
    }  
  
    public T getContent() {  
        return content;  
    }  
}
```

在这个示例中，Box 类定义了一个类型参数 T，它可以代表任何一种类型。Box 类有一个私有字段 content，它的类型是 T。Box 类有一个构造方法和一个公共方法 getContent，它们都使用了类型参数 T。

使用泛型类时，需要在类名后面加上类型参数，例如：

```
Box<String> box = new Box<>("hello");
String content = box.getContent();
```

在这个示例中，我们创建了一个 Box 类型的对象 box，并将它的类型参数设置为 String。然后我们调用 box 的 getContent 方法，获取它的 content 字段的值。由于 box 的类型参数是 String，因此 getContent 方法的返回值类型也是 String。

泛型方法

泛型方法是指在定义方法时使用类型参数，例如：

```
public class Utils {
    public static <T> T pick(T[] array) {
        int index = (int) (Math.random() * array.length);
        return array[index];
    }
}
```

在这个示例中，Utils 类定义了一个泛型方法 pick，它有一个类型参数 T，它可以代表任何一种类型。pick 方法接收一个 T 类型的数组，并返回一个 T 类型的值。pick 方法使用了 Math.random 方法生成一个随机数，然后根据随机数选取数组中的一个元素作为返回值。

使用泛型方法时，需要在方法名前面加上类型参数，例如：

```
String[] words = {"hello", "world", "java"};
String word = Utils.pick(words);
```

在这个示例中，我们定义了一个 String 类型的数组 words，然后调用 Utils 类的 pick 方法，获取数组中的一个随机元素。由于 pick 方法是一个泛型方法，因此它的类型参数会根据传入的参数类型自动推断出来。

泛型擦除

在 Java 中，泛型是通过类型擦除来实现的。类型擦除是指在编译时将泛型类型替换为它的边界类型或 Object 类型，然后在运行时使用 Object 类型来表示泛型类型。例如：

```
public class Box<T> {  
    private T content;  
  
    public Box(T content) {  
        this.content = content;  
    }  
  
    public T getContent() {  
        return content;  
    }  
}
```

在这个示例中，Box 类中的类型参数 T 在编译时会被替换为 Object 类型。因此，Box 类的字节码文件中并不包含 T 类型的信息。

类型边界

Java 中的泛型类型参数可以有类型边界，例如：

```
public class Box<T extends Number> {  
    private T content;  
  
    public Box(T content) {  
        this.content = content;  
    }  
  
    public T getContent() {  
        return content;  
    }  
}
```

在这个示例中，Box 类的类型参数 T 有一个类型边界，它必须是 Number 类型或 Number 的子类。这个类型边界可以保证 Box 类中的 content 字段的类型是 Number 类型或 Number 的子类型。

通配符

Java 中的通配符可以用来表示泛型类型参数的上界或下界。通配符有两种形式：上界通配符和下界通配符。

上界通配符 (<? extends T>)

上界通配符可以用来表示泛型类型参数的上界，例如：

```
public class Box<T extends Number> {  
    private List<? extends T> list;   
  
    public Box(List<? extends T> list) {  
        this.list = list;  
    }  
  
    public double sum() {  
        double sum =
```

异常处理

Java的异常处理模块可以让程序员捕获和处理程序运行时出现的异常。异常处理模块包括以下几个关键字和语句：

- try：用于包含可能抛出异常的代码块。
- catch：用于捕获抛出的异常并进行处理。
- finally：用于包含一些无论是否抛出异常都必须执行的代码。
- throw：用于手动抛出异常。
- throws：用于声明方法可能抛出的异常。

异常处理模块的基本语法如下：

```
try {  
    // 可能抛出异常的代码块  
} catch (ExceptionType1 e1) {  
    // 处理 ExceptionType1 类型的异常  
} catch (ExceptionType2 e2) {  
    // 处理 ExceptionType2 类型的异常  
} finally {  
    // 无论是否抛出异常都必须执行的代码  
}
```

在这个语法中，try 语句块中的代码可能会抛出异常。如果抛出了异常，就会进入与异常类型匹配的 catch 语句块中进行处理。如果没有抛出异常，就会跳过 catch 语句块。无论是否抛出异常，finally 语句块中的代码都会被执行。



Java 中的异常可以分为两种类型：受检异常和非受检异常。受检异常必须在方法声明中用 throws 关键字声明，或者在方法内部使用 try-catch 语句块进行处理。非受检异常不需要声明或处理，它们通常表示程序内部出现了严重的错误，例如 NullPointerException、ArrayIndexOutOfBoundsException 等等。

以下是一个使用异常处理模块的示例：

```
public class ExceptionDemo {
    public static void main(String[] args) {
        try {
            int num1 = Integer.parseInt(args[0]);
            int num2 = Integer.parseInt(args[1]);
            int result = divide(num1, num2);
            System.out.println("Result: " + result);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Usage: java ExceptionDemo <num1> <num2>");
        } catch (NumberFormatException e) {
            System.out.println("Invalid input: " + e.getMessage());
        } catch (ArithmaticException e) {
            System.out.println("Division by zero: " + e.getMessage());
        } finally {
            System.out.println("Program finished.");
        }
    }

    public static int divide(int num1, int num2) {
        return num1 / num2;
    }
}
```

在这个示例中，我们定义了一个 divide 方法，用于计算两个数的商。在 main 方法中，我们使用 try-catch 语句块来捕获和处理可能抛出的异常。如果输入的参数不足两个，就会抛出 ArrayIndexOutOfBoundsException 异常；如果输入的参数无法转换为整数，就会抛出 NumberFormatException 异常；如果除数为零，就会抛出 ArithmaticException 异常。无论是否抛出异常，finally 语句块中的代码都会被执行。

finally和return的优先级

```
public static int test() {
    try {
        return 1;
    } catch (Exception e) {
        return 2;
    } finally {
```

```
        return 3;  
    }  
}
```

在这个示例中，try语句块中的代码将返回值设为1。但是由于finally语句块中的代码也包含一个return语句，因此它会覆盖try语句块中的return语句，方法会直接返回3。因此，调用test()方法会返回3，而不是1或2。



需要注意的是，finally语句块中的return语句只会覆盖try或catch语句块中的return语句，而不会影响方法中其他地方的返回值。因此，在方法中同时出现多个return语句时，应该格外注意finally语句块中的return语句是否会影响方法的正常执行。

ERROR

Java中的Error是Throwable类的一个子类，表示程序运行时出现的严重错误。与Exception不同，Error通常不能被捕获和处理，它们通常表示程序内部出现了严重的错误，例如OutOfMemoryError、StackOverflowError等等。以下是Java中一些常见的Error子类：

- OutOfMemoryError：当JVM无法分配足够的内存时，会抛出OutOfMemoryError异常。这通常是由程序中使用了太多的内存，或者JVM分配的内存不足导致的。
- StackOverflowError：当JVM堆栈空间不足时，会抛出StackOverflowError异常。这通常是由于方法递归调用层数过多导致的。

虽然Error通常不能被捕获和处理，但是可以通过一些手段来避免出现这些错误。例如，可以通过JVM的参数来增加堆栈空间大小、限制内存使用或者使用垃圾回收器等。如果遇到了Error异常，通常需要对程序的设计和实现进行仔细的检查和优化，以避免类似的错误再次发生。

Exception

在Java中，Exception是Throwable类的一个子类，表示程序运行时出现的异常情况。Java中的Exception又分为运行时异常和检查时异常两种类型。

运行时异常

运行时异常是指Java程序在运行时发生的异常，通常是由于程序中的逻辑错误或者其他未知的原因导致的。在Java中，运行时异常通常是RuntimeException及其子类。例如：

- NullPointerException：当程序试图访问一个空对象时，就会抛出NullPointerException异常。

- `ArrayIndexOutOfBoundsException`：当程序试图访问一个不存在的数组元素时，就会抛出`ArrayIndexOutOfBoundsException`异常。
- `ArithmaticException`：当程序试图进行除以零等不可能完成的算术运算时，就会抛出`ArithmaticException`异常。

通常情况下，运行时异常不需要在程序中进行显式的捕获和处理。因为这些异常通常是由程序中的错误导致的，需要开发者自己对代码进行检查和修正。

检查时异常

检查时异常是指Java程序在编译时就能够被发现的异常，通常是由于程序中的外部条件不满足或者其他已知的原因导致的。在Java中，检查时异常通常是Exception及其子类，但不包括RuntimeException及其子类。例如：

- `FileNotFoundException`：当程序试图打开一个不存在的文件时，就会抛出`FileNotFoundException`异常。
- `IOException`：当程序试图读写一个文件或者网络连接时，可能会发生IO异常。

由于检查时异常通常是由于程序中的外部条件不满足或者其他已知的原因导致的，因此通常需要在程序中进行显式的捕获和处理。在Java中，可以使用try-catch语句块来捕获和处理检查时异常。例如：

```
try {
    FileInputStream fis = new FileInputStream("file.txt");
    // 读取文件
} catch (FileNotFoundException e) {
    System.out.println("File not found: " + e.getMessage());
}
```

在这个示例中，我们使用try-catch语句块来捕获`FileNotFoundException`异常。如果文件不存在，就会进入catch语句块中进行处理。如果文件存在，就会跳过catch语句块。

需要注意的是，检查时异常通常需要在程序中进行显式的捕获和处理，否则程序将无法通过编译。因此，在使用Java中的类库时，需要仔细查看文档，了解可能抛出的检查时异常，并进行相应的处理。

集合类

LIST

Java中的List接口是一个有序的集合，它可以存储重复的元素。List接口继承自Collection接口，并添加了一些额外的方法。常用的List实现类包括：

- ArrayList：基于数组实现的动态数组，支持随机访问。它的插入和删除操作效率较低，但是随机访问元素的效率很高。
- LinkedList：基于链表实现的双向链表，支持快速的插入和删除操作。它的随机访问效率较低，但是迭代操作效率很高。
- Vector：与ArrayList类似，但是是线程安全的。
- CopyOnWriteArrayList是Java中的一个并发集合类，它是线程安全的，适用于读多写少的场景。它的实现方式是在修改元素时，先将原有的数组复制一份，然后对复制后的数组进行修改，最后将原有的数组替换为新的数组。这种实现方式可以避免在迭代时出现ConcurrentModificationException异常。



▼ ArrayList

ArrayList是Java中的一个集合类，它实现了List接口，可以存储任意类型的对象，并支持随机访问。ArrayList是基于动态数组实现的，可以自动扩容以适应存储的元素数量的变化。在Java中，ArrayList是非线程安全的，如果多个线程同时访问同一个ArrayList对象，可能会导致数据不一致的问题，需要进行同步处理。

扩容机制

ArrayList初始化容量为10，在添加元素时，会自动检查当前的容量是否足够，如果不足够，就会进行扩容。扩容的具体实现方式如下：

1. 如果当前元素数量小于等于10，则新容量为原容量加上5。
2. 如果当前元素数量大于10，则新容量为原容量的1.5倍。

在扩容时，ArrayList会创建一个新的数组，并将原有的元素复制到新数组中。这个过程会消耗一定的时间和空间，因此建议在使用ArrayList时，尽可能预估数据规模，避免多次扩容。

ConcurrentModificationException

当多个线程同时对同一个ArrayList对象进行修改时，可能会导致数据不一致的问题。为了避免这种情况，Java在ArrayList内部维护了一个modCount变量，用于记录ArrayList被修改的次数。当多个线程同时对ArrayList进行修改时，可能会导致



modCount的值发生变化，从而导致迭代时抛出`ConcurrentModificationException`异常。

为了避免这种情况，可以在迭代时使用`Iterator`对象，并在修改`ArrayList`时使用`Iterator`的`remove()`方法，而不是直接调用`ArrayList`的`remove()`方法。

▼ Linked List

Java中的`LinkedList`是一个双向链表，实现了List、Deque、Queue等接口。它与`ArrayList`一样可以存储任意类型的对象，并支持随机访问。相比于`ArrayList`，`LinkedList`的插入和删除操作效率更高，但是随机访问元素的效率较低。

底层实现原理

`LinkedList`的底层实现是一个双向链表。每个节点包含了一个存储的元素以及指向前一个节点和后一个节点的指针。因此，在插入和删除元素时，只需要修改相应的指针即可，不需要像数组一样进行元素的移动。

add方法的过程

`LinkedList`的`add`方法会在链表的末尾添加一个新的节点。具体过程如下：

1. 创建一个新的节点，将元素存储到节点中。
2. 将新节点的`prev`指针指向当前链表的最后一个节点。
3. 将当前链表最后一个节点的`next`指针指向新节点。
4. 将链表的`size`加1。

get方法的过程

`LinkedList`的`get`方法会获取链表中指定位置的元素。具体过程如下：

1. 如果待查找位置小于0或者大于等于链表的`size`，就抛出`IndexOutOfBoundsException`异常。
2. 如果待查找位置小于链表`size`的一半，则从链表的头部开始向后查找。
3. 如果待查找位置大于等于链表`size`的一半，则从链表的尾部开始向前查找。
4. 当找到指定位置的节点时，返回其存储的元素。

▼ Vector

Java中的Vector是一个向量集合，它是线程安全的，可以存储任意类型的对象，并支持随机访问。Vector实现了List接口，可以像ArrayList一样使用，但是由于Vector是线程安全的，因此在多线程环境下使用时不需要进行额外的同步处理。

底层实现原理

Vector的底层实现方式与ArrayList类似，都是基于动态数组实现的。当Vector容量不足以存放新元素时，它会自动扩容，具体的扩容方式与ArrayList相同。Vector的默认容量为10。

与ArrayList的比较

Vector与ArrayList类似，都是动态数组实现的集合类，它们的主要区别在于线程安全性和同步处理。在单线程环境下，ArrayList的效率通常比Vector高，因为Vector的每个操作都需要进行同步处理，而同步处理需要消耗一定的时间和资源。因此，在单线程环境下，建议使用ArrayList。

在多线程环境下，由于Vector是线程安全的，因此不需要进行额外的同步处理，这使得Vector在多线程环境下更加适用。

ConcurrentModificationException

当多个线程同时对同一个Vector对象进行修改时，可能会导致数据不一致的问题。为了避免这种情况，Java在Vector内部维护了一个modCount变量，用于记录Vector被修改的次数。当多个线程同时对Vector进行修改时，可能会导致modCount的值发生变化，从而导致迭代时抛出ConcurrentModificationException异常。

为了避免这种情况，可以在迭代时使用Iterator对象，并在修改Vector时使用Iterator的remove()方法，而不是直接调用Vector的remove()方法。

例子如下：

```
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        Vector<Integer> vector = new Vector<>();
        vector.add(1);
        vector.add(2);
        vector.add(3);
        vector.add(4);
        vector.add(5);

        Runnable r = () -> {
            for (int i = 0; i < vector.size(); i++) {
```

```

        System.out.println(Thread.currentThread().getName() + " : " + vector.get(i));
    }
    // 模拟在迭代过程中修改vector
    if (i == 2) {
        vector.add(6);
        vector.remove(3);
    }
}
};

Thread t1 = new Thread(r);
Thread t2 = new Thread(r);

t1.start();
t2.start();
}
}

```

在上面的代码中，我们创建了一个包含5个整数的Vector，然后创建两个线程分别遍历这个Vector。当第一个线程遍历到第三个元素时，它会添加一个新元素6，并删除第四个元素。这种修改操作可能导致第二个线程在其迭代期间抛出并发修改异常。【当然，只是可能，不一定会发生】

▼ CopyOnWriteArrayList

CopyOnWriteArrayList是Java中的一个并发集合类，它是线程安全的，适用于读多写少的场景。 CopyOnWriteArrayList实现了List接口，可以存储任意类型的对象，并支持随机访问。

底层实现原理

CopyOnWriteArrayList的底层实现是一个数组，每次对集合进行修改时，都会对原有的数组进行一次复制，并将新元素添加到新的数组中。这样可以避免多个线程同时修改同一个数组时出现数据不一致的问题。

优点

CopyOnWriteArrayList的主要优点在于它的线程安全性和高效性。由于CopyOnWriteArrayList采用了写时复制的机制，因此在读取元素时不需要进行任何同步操作，这使得CopyOnWriteArrayList的读取效率非常高。此外，由于

CopyOnWriteArrayList是线程安全的，因此在多线程环境下使用时不需要进行额外的同步处理，这使得CopyOnWriteArrayList在多线程环境下更加适用。

缺点

CopyOnWriteArrayList的主要缺点在于它的写操作效率较低。每次对集合进行写操作时，都需要对原有的数组进行一次复制，这样会消耗一定的时间和空间。

Map

Java的Map接口是一种键值对的数据结构，它可以存储任意类型的键和值，并将它们关联起来。Map接口是Java集合框架中的一部分，它定义了一组操作，可以对键值对进行添加、删除、修改和查询等操作。Map接口有多个实现类，包括HashMap、hashtable、concurrentHashMap等。

▼ HashMap

Java的HashMap是一种散列表，它存储的对象是键值对(key-value)。HashMap的实现是非常高效的，它可以在O(1)时间复杂度内完成基本的插入、删除和查找操作。

底层实现原理

JDK7

在JDK7中，HashMap的底层实现是一个数组，每个数组元素都是一个链表。当插入一个键值对时，HashMap会根据键的hash值计算出一个数组下标，并将键值对存储到对应下标的链表中。当多个键的hash值相同时，它们会被存储在同一个链表中。

JDK8

在JDK8中，HashMap的底层实现和JDK7有所不同。JDK8中，HashMap的底层实现是一个数组，每个数组元素都是一个链表或红黑树。当链表中的元素数量达到8时，HashMap会将链表转换为红黑树。当红黑树中的元素数量小于等于6时，HashMap会将红黑树转换为链表。

put方法的流程

将调用hash(Key)方法得到Key的哈希值hash后，通过(n - 1) & hash方法获得Key值对应的索引位置，放入到数组对应的位置。如果出现哈希冲突，则用链表解决哈希冲

突，当链表长度过长的时候，转成红黑树。

对于Key值的hash的原理

在HashMap中，每个键的hash值是由键的hashCode()方法返回的。HashMap会将这个hash值与数组的长度 - 1进行按位与运算，来得到这个键在数组中的位置。由于数组的长度是2的n次方，因此这个运算得到的结果就是0到数组长度-1之间的一个整数，刚好落在数组下标上。

扩容机制

当HashMap中的元素数量达到数组长度的0.75倍时，HashMap会自动进行扩容。扩容的过程包括以下几个步骤：

1. 创建一个新的数组，新数组的长度是原数组长度的2倍。
2. 将原数组中的元素重新计算hash值，并根据hash值存储到新数组中的对应位置。
3. 将新数组设置为HashMap的底层数组。

为什么数组长度是2的n次方

当数组长度不为2的n次幂的时候， hashCode 值与数组长度减一做与运算的时候，会出现重复的数据，因为不为2的n次幂的话，对应的二进制数肯定有一位为0，这样，不管你的hashCode 值对应的该位，是0还是1，最终得到的该位上的数肯定是0，这带来的问题是HashMap上的数组元素分布不均匀，而数组上的某些位置，永远也用不到。

负载因子LOAD_FACTOR 的作用

负载因子LOAD_FACTOR指的是HashMap中元素数量与数组长度之比。当HashMap中的元素数量达到数组长度的0.75倍时，HashMap会自动进行扩容。这个0.75就是负载因子。负载因子的大小会影响HashMap的性能。如果负载因子过小，那么HashMap会占用较多的空间；如果负载因子过大，那么HashMap的性能可能会受到影响。JDK注释中有说明，根据泊松分布，0.75是一个非常优秀的值。【不需要了解泊松分布的原理，超纲了】

为什么线程不安全？

本质上是没有加锁，不同步导致的。

JDK7采用头插法，造成多线程插入元素时、如果发生扩容，可能会导致形成环，造成死循环。

在JDK7中，当多个线程同时进行插入元素操作时，如果需要进行扩容操作，可能会出现环形链表的情况，从而导致死循环。

```
//假设线程A和线程B同时向hashMap 数组的0位置插入一个值，分别是a和b，理想状态下应该是  
[0]: b -> a ->null  
  
/**  
但是在多线程情况下，元素b还没有完全地把next指针指向a，也就是说完成了一半，  
B线程插入的时候，进行扩容了，那所有的元素都需要进行再哈希，假设说变成了这样  
*/  
  
[0]: a -> b  
  
/**  
那此时b的next指针指向a，就出现了死循环了  
*/
```

因此，JDK8移除了头插法。

▼ hashtable

Java的Hashtable是一种散列表，它和HashMap一样，存储的对象也是键值对(key-value)。Hashtable的实现也是非常高效的，它可以在O(1)时间复杂度内完成基本的插入、删除和查找操作。Hashtable的底层实现和HashMap类似，都是一个数组，每个数组元素都是一个链表。当插入一个键值对时，Hashtable会根据键的hash值计算出一个数组下标，并将键值对存储到对应下标的链表中。当多个键的hash值相同时，它们会被存储在同一个链表中。

线程安全

Hashtable是Java中的一个线程安全的集合类，它是通过同步（synchronized）来保证线程安全的。Hashtable的每个方法都是同步的，也就是说，同一时刻只能有一个线程对Hashtable进行修改。这种同步机制虽然可以保证线程安全，但也会带来一定的性能损失。

为什么Hashtable不建议使用

虽然Hashtable是线程安全的，但它的同步机制是通过synchronized关键字来实现的，这会带来一定的性能损失。同时，由于Hashtable的扩容机制和HashMap类似，它也存在着和HashMap一样的rehashing问题，即在扩容时需要重新计算hash值，这会导致效率降低。因此，在Java 1.2之后，推荐使用ConcurrentHashMap来代替Hashtable，它的性能更好，而且也是线程安全的。

▼ ConcurrentHashMap

ConcurrentHashMap是Java中的一个并发集合类，它是线程安全的，适用于高并发的场景。ConcurrentHashMap实现了Map接口，可以存储任意类型的对象，并支持随机访问。

底层实现原理

ConcurrentHashMap的底层实现是一个数组，每个数组元素都是一个链表或红黑树。当链表中的元素数量达到8时，ConcurrentHashMap会将链表转换为红黑树。当红黑树中的元素数量小于等于6时，ConcurrentHashMap会将红黑树转换为链表。ConcurrentHashMap中通过分段锁的方式来实现线程安全。

JDK7和JDK8的底层实现

在JDK7中，ConcurrentHashMap的底层实现是一个数组，每个数组元素都是一个链表。

在JDK7中，ConcurrentHashMap的线程安全是通过分段锁来实现的。具体来说，ConcurrentHashMap将整个Map分成了一个个Segment，每个Segment都是一个独立的散列表，可以独立地进行加锁和解锁操作。在进行修改操作时，只需要锁定当前要操作的Segment，而不需要锁定整个HashMap，这样就大大提高了并发能力。当读取

操作时，不需要加锁，这样可以保证读取操作的高效性。相比于Hashtable，ConcurrentHashMap的性能更好，而且也是线程安全的。

在JDK8中，ConcurrentHashMap的底层实现和JDK7有所不同。JDK8中，ConcurrentHashMap的底层实现是一个数组，每个数组元素都是一个链表或红黑树。当链表中的元素数量达到8时，ConcurrentHashMap会将链表转换为红黑树。当红黑树中的元素数量小于等于6时，ConcurrentHashMap会将红黑树转换为链表。

在JDK8中，ConcurrentHashMap摒弃了Segment，采用了更加高效的方式来实现线程安全。具体就是使用CAS + synchronized 保证线程安全。当插入节点的时候，首先会检查插入的位置是否为空，如果为空，则尝试cas插入头节点。否则，说明节点不为空，这个时候会使用synchronized 锁住头节点，插入节点。

下面是关键源码的解析(一般面试不会让你背源码，只是为了更好理解，所以面试回答上面一段话即可)

```
//JDK8的 put方法
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode()); //key值扰动，提高散列性，也就是高16位和低16位的异或
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0) //数组为空，初始化
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 如果指定的桶为空，使用CAS操作添加元素，即尝试占据桶的位置
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break; // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            // 如果当前桶的第一个元素的hash值为MOVED，表示正在进行扩容，则协助扩容
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            // 如果当前桶不为空，且当前线程持有桶的头节点f的锁，则在桶中添加元素
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        .....
                    }
                }
            }
        }
    }
}
```

Set

Java的Set接口是Java集合框架中的一个接口，它继承自Collection接口，用于存储不重复的元素。Set接口中没有定义新的方法，和Collection接口中的方法一样，只是Set中的元素不能重复。Set接口有很多实现类，如HashSet、TreeSet等。其中，HashSet是基于哈希表的实现，TreeSet是基于红黑树的实现。

HashSet

HashSet是一个无序、不重复的集合，它可以存储null元素。HashSet的底层实现是一个哈希表，它是通过hash值来确定元素在集合中的位置的。当元素被添加到HashSet中时，HashSet会计算元素的hash值，然后根据hash值将元素存储到相应的桶中。如果两个元素的hash值相同，那么它们会被存储在同一个桶中，这就是哈希冲突。当需要在HashSet中查找元素时，HashSet会根据元素的hash值快速定位到元素所在的桶，然后在桶中查找元素。

TreeSet

TreeSet是一个有序、不重复的集合，它可以存储null元素。TreeSet的底层实现是一个红黑树，它是一种自平衡的二叉搜索树。当元素被添加到TreeSet中时，TreeSet会根据元素的大小将元素插入到红黑树中的相应位置。当需要在TreeSet中查找元素时，TreeSet会根据元素的大小快速定位到元素所在的位置，然后在红黑树中查找元素。

多线程与线程安全

线程的创建方式

在Java中，线程的创建方式有三种：

1. 继承Thread类并重写run()方法
2. 实现Runnable接口并实现run()方法

3. 实现 Callable 接口并实现 call() 方法

第一种方式是最简单的，只需要创建一个类，继承 Thread 类并重写 run() 方法，然后创建该类的实例并调用 start() 方法即可。线程开始运行后，会自动调用 run() 方法执行线程的任务。

```
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        // 线程要执行的任务  
    }  
}  
  
MyThread t = new MyThread();  
t.start(); // 开始执行线程
```

第二种方式是比较常用的一种方式，它可以避免由于 Java 不支持多重继承而导致的继承局限性。实现 Runnable 接口并实现 run() 方法，然后创建该类的实例并将其作为参数传递给 Thread 类的构造方法即可。线程开始运行后，会自动调用 run() 方法执行线程的任务。

```
public class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        // 线程要执行的任务  
    }  
}  
  
MyRunnable r = new MyRunnable();  
Thread t = new Thread(r);  
t.start(); // 开始执行线程
```

第三种方式是比较复杂的一种方式，它可以返回线程执行结果。实现 Callable 接口并实现 call() 方法，然后创建该类的实例并将其作为参数传递给 FutureTask 类的构造方法，再将 FutureTask 实例作为参数传递给 Thread 类的构造方法即可。线程开始运行后，会自动调用 call() 方法执行线程的任务，并返回执行结果。

```
public class MyCallable implements Callable<String> {  
    @Override  
    public String call() throws Exception {  
        // 线程要执行的任务  
        return "执行结果";  
    }  
}
```

```
MyCallable c = new MyCallable();
FutureTask<String> task = new FutureTask<>(c);
Thread t = new Thread(task);
t.start(); // 开始执行线程
String result = task.get(); // 获取线程执行结果
```

以上三种方式都可以创建线程，但第一种方式和第二种方式更为常用。

线程池

线程池的作用

线程池是为了解决线程生命周期开销过大的问题而设计的，它可以在程序启动时创建大量的线程并放入线程池中等待使用，这样可以避免反复创建和销毁线程所带来的开销。使用线程池可以提高程序的效率和稳定性，减少系统资源的消耗。

JDK提供的线程池

Java中提供了三种线程池：FixedThreadPool、CachedThreadPool和ScheduledThreadPool。

1. FixedThreadPool：该线程池中的线程数量是固定的，当线程池中的所有线程都在工作时，新的任务会被放入等待队列中。该线程池适用于需要限制线程数量的场景，如服务器端的并发请求处理。
2. CachedThreadPool：该线程池中的线程数量不固定，当线程池中的所有线程都在工作时，会创建新的线程来处理新的任务。当线程池中的线程闲置时间超过指定时间时，线程会被销毁。该线程池适用于执行许多短期异步任务的程序，如网络服务器。
3. ScheduledThreadPool：该线程池可以执行定时任务和周期任务，适用于需要定时执行任务的场景，如定时器、任务调度等。

线程池	描述
FixedThreadPool	线程数量固定，当线程池中的所有线程都在工作时，新的任务会被放入等待队列中。适用于需要限制线程数量的场景，如服务器端的并发请求处理。
CachedThreadPool	线程数量不固定，当线程池中的所有线程都在工作时，会创建新的线程来处理新的任务。当线程池中的线程闲置时间超过指定时间时，线程会被销毁。

	毁。适用于执行许多短期异步任务的程序，如网络服务器。
<u>ScheduledThreadPool</u>	可以执行 <u>定时任务和周期任务</u> ，适用于需要定时执行任务的场景，如定时器、任务调度等。

线程池的核心参数

线程池的核心参数包括以下几个：

1. corePoolSize : 线程池中的核心线程数量，即线程池中能够同时运行的线程数量。
2. maximumPoolSize : 线程池中的最大线程数量，即线程池中允许的最大线程数量。
3. keepAliveTime : 线程闲置超时时间，当线程池中的线程数量大于核心线程数量时，多余的线程会被销毁，直到线程池中的线程数量等于核心线程数量。如果线程池中的线程数量仍然大于核心线程数量，多余的线程会被销毁，直到线程池中的线程数量等于最大线程数量或线程闲置时间超过指定的时间。
4. unit : 线程闲置超时时间的单位。
5. workQueue : 线程池中的任务队列，用于保存等待执行的任务。

参数	说明
<u>corePoolSize</u>	线程池中的核心线程数量
<u>maximumPoolSize</u>	线程池中的最大线程数量
<u>keepAliveTime</u>	线程闲置超时时间
<u>unit</u>	线程闲置超时时间的单位
<u>workQueue</u>	线程池中的任务队列

线程池的工作流程

线程池的工作流程可以分为以下几个步骤：

1. 当有任务需要执行时，首先会判断线程池中的线程数量是否达到了核心线程数量。如果没有达到，会创建一个新的线程来执行任务；如果已经达到了核心线程数量，任务会被加入到任务队列中等待执行。
2. 当任务被加入到任务队列中后，线程池中的线程会不断地从任务队列中取出任务并执行。如果任务队列中没有任务，线程会进入等待状态。

3. 如果任务队列中的任务数量达到了一定的数量或者任务等待的时间超过了设定的时间，线程池会创建新的线程来执行任务，直到线程数量达到了最大线程数量。

4. 当任务执行完毕后，线程会返回线程池，等待下一次的任务执行。

线程池中的线程可以分为两种：核心线程和非核心线程。核心线程是线程池中的基础线程数量，只有在核心线程都在工作时，新的任务才会被加入到任务队列中等待执行。非核心线程是当任务数量达到一定数量或等待时间超过设定时间时，线程池会创建的新线程。当线程池中的线程数量超过了核心线程数量时，非核心线程可以被销毁，以节省系统资源。

线程池中的任务队列有两种：有界队列和无界队列。有界队列是指队列中只能存放一定数量的任务，当队列中的任务数量达到了一定数量时，新的任务将无法加入队列；无界队列是指队列中可以存放任意数量的任务，当队列中的任务数量达到了一定数量时，新的任务将会一直加入队列中等待执行。

线程池的工作流程可以有效地提高系统的性能和稳定性，避免了多次创建和销毁线程所带来的开销，并且可以根据需要动态地调整线程数量，以适应不同的系统负载。

手写线程池

以下是一个简单的手写线程池的示例代码：

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class MyThreadPool {

    private final int nThreads; // 线程池中线程的数量
    private final PoolWorker[] threads; // 线程池中的线程
    private final BlockingQueue<Runnable> queue; // 任务队列

    /**
     * 构造方法，初始化线程池和任务队列
     * @param nThreads 线程池中线程的数量
     */
    public MyThreadPool(int nThreads) {
        this.nThreads = nThreads;
        queue = new LinkedBlockingQueue<>();
        threads = new PoolWorker[nThreads];
        for (int i = 0; i < nThreads; i++) {
            threads[i] = new PoolWorker();
            threads[i].start(); // 启动线程池中的线程
        }
    }
}
```

```

    /**
     * 提交任务到线程池中
     * @param task 要执行的任务
     */
    public void execute(Runnable task) {
        synchronized (queue) {
            queue.add(task); // 将任务添加到任务队列中
            queue.notify(); // 唤醒一个正在等待的线程
        }
    }

    /**
     * 线程池中执行任务的线程类
     */
    private class PoolWorker extends Thread {
        public void run() {
            Runnable task;
            while (true) {
                synchronized (queue) {
                    while (queue.isEmpty()) {
                        try {
                            queue.wait(); // 等待任务队列非空
                        } catch (InterruptedException e) {
                            System.err.println("线程被中断");
                            return;
                        }
                    }
                    task = queue.poll(); // 取出一个任务
                }
                try {
                    task.run(); // 执行任务
                } catch (Throwable t) {
                    System.err.println("执行任务时出现异常: " + t);
                }
            }
        }
    }
}

```

使用方式如下：

```

public static void main(String[] args) {
    MyThreadPool pool = new MyThreadPool(5);
    for (int i = 0; i < 10; i++) {
        pool.execute(() -> {
            System.out.println(Thread.currentThread().getName() + "执行任务");
        });
    }
}

```

以上代码创建了一个线程池，大小为5，然后提交了10个任务给线程池执行。每个任务只是简单地输出当前线程的名称。

同步方式

悲观锁

悲观锁是一种传统的同步方式，它的基本思想是，对于共享数据的访问持保守态度，认为每次操作共享数据都会发生冲突，因此需要加锁来保证共享数据的一致性和完整性。在悲观锁的机制下，如果一个线程获得了锁，其他线程就必须等待这个线程释放锁后才能访问共享数据。悲观锁的实现方式包括 synchronized 关键字和 ReentrantLock 类。

synchronized

1. 使用方式

Java中的synchronized关键字可以用来修饰方法或代码块，以达到同步访问共享数据的目的。使用synchronized修饰方法时，相当于将整个方法加锁，只有一个线程能够执行该方法；使用synchronized修饰代码块时，需要指定一个锁对象，只有获得该锁对象的线程才能够执行代码块。

synchronized 修饰普通方法和静态方法的区别在于锁的对象不同。



对于 synchronized 修饰的普通方法，锁的对象是调用该方法的对象（即 this），每个对象都有一个锁，因此不同的对象调用同一个 synchronized 方法时，它们之间不会互相阻塞。

对于 synchronized 修饰的静态方法，锁的对象是该方法所在的类对象，因为每个类只有一个类对象，不同的线程调用同一个 synchronized 静态方法时，它们之间会互相阻塞。

以下是一个简单的示例代码：

```
public class SynchronizedDemo {  
    private static int count1 = 0;  
    private int count2 = 0;
```

```

public synchronized void incrementCount1() {
    count1++;
}

public static synchronized void incrementCount1Static() {
    count1++;
}

public synchronized void incrementCount2() {
    count2++;
}

public static void main(String[] args) {
    SynchronizedDemo demo1 = new SynchronizedDemo();
    SynchronizedDemo demo2 = new SynchronizedDemo();

    // 调用普通方法，锁的对象是调用该方法的对象
    Thread thread1 = new Thread(() -> {
        for (int i = 0; i < 100000; i++) {
            demo1.incrementCount1();
        }
    });
    Thread thread2 = new Thread(() -> {
        for (int i = 0; i < 100000; i++) {
            demo1.incrementCount1();
        }
    });

    thread1.start();
    thread2.start();

    try {
        thread1.join();
        thread2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("普通方法 count1: " + demo1.count1); // 输出 200000
    System.out.println("普通方法 count2: " + demo1.count2); // 输出 100000

    // 调用静态方法，锁的对象是类对象
    Thread thread3 = new Thread(() -> {
        for (int i = 0; i < 100000; i++) {
            SynchronizedDemo.incrementCount1Static();
        }
    });

    Thread thread4 = new Thread(() -> {
        for (int i = 0; i < 100000; i++) {
            SynchronizedDemo.incrementCount1Static();
        }
    });
}

```

```
});  
  
thread3.start();  
thread4.start();  
  
try {  
    thread3.join();  
    thread4.join();  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
  
System.out.println("静态方法 count1: " + SynchronizedDemo.count1); // 输出 200000  
}  
}
```

以上代码分别调用了两个线程来分别增加实例变量 count1 和 count2 的值，以及静态变量 count1 的值。结果输出如下：

```
普通方法 count1: 200000  
普通方法 count2: 100000  
静态方法 count1: 200000
```

可以看到，普通方法的 count1 变量的值为 200000，而 count2 变量的值为 100000，这是因为两个线程分别调用了不同的对象的方法，它们之间不会互相阻塞。而静态方法的 count1 变量的值也为 200000，这是因为两个线程调用的是同一个类对象的静态方法，它们之间会互相阻塞。

2. 锁升级的过程

在Java中，synchronized锁有四种状态：

- 无锁状态：表示当前对象没有被锁定，可以被任意线程访问
- 偏向锁状态：表示当前对象被一个线程访问，但是没有竞争，此时该线程获得偏向锁，并将对象头中的线程ID记录下来
- 轻量级锁状态：表示当前对象被多个线程竞争，但是竞争的时间很短暂，此时使用CAS尝试去获取锁，不会造成线程阻塞。
- 重量级锁状态：表示当前对象被多个线程竞争，竞争的时间较长，此时使用操作系统提供的互斥量实现锁，将该对象加入到等待队列中，等待唤醒

3. Java对象头的Mark Word

64位虚拟机						
锁状态	56bit		1bit	4bit	1bit (是否偏向锁)	2bit (锁标志位)
	25bit	31bit				
无锁	unused	对象 hashCode	Cms_free	对象分代年龄	0	01
偏向锁	threadId(54bit)(偏向锁的线程ID)	Epoch(2bit)	Cms_free	对象分代年龄	1	01
轻量级锁	指向栈中锁的记录的指针					00
重量级锁	指向重量级锁的指针					10
GC 标志	空					11

Java 中的锁升级是通过对象头（Object Header）中的标志位实现的。Java 对象头包含了一些元数据信息，如对象的哈希码、GC 信息以及锁标志等。锁标志用于记录当前对象的锁状态，可以用 2 位二进制位表示。

锁状态一般分为以下 4 种：

1. 无锁状态（标志位为“01”）：对象头中的锁标志位为 01 表示当前对象没有被锁定，可以被任意线程访问。
2. 偏向锁状态（标志位为“01”）：当一个线程获取了偏向锁并成功执行了同步块后，该对象的锁标志位会变为“01”，同时记录下线程 ID。此时，如果该线程再次访问同步块，无需重新获取锁，直接进入同步块即可。这种状态下，锁的获取和释放都非常快。
3. 轻量级锁状态（标志位为“00”）：当多个线程竞争同一个偏向锁时，偏向锁就会升级为轻量级锁状态。此时，虚拟机会在当前线程的栈帧中创建一个 Lock Record (锁记录)，记录下当前对象的哈希值和指向 Lock Record 的指针。接着，虚拟机会使用 CAS 操作尝试将对象头中的锁标志位设置为“00”，并将 Lock Record 中的锁标志位设置为“01”。如果 CAS 操作成功，说明当前线程成功获取到了锁，可以执行同步块；否则，说明当前对象已经被其他线程锁定，当前线程需要尝试使用其他方式获取锁。
4. 重量级锁状态（标志位为“10”）：如果轻量级锁获取失败，说明竞争非常激烈，Java 虚拟机会将锁升级为重量级锁。此时，虚拟机会使用操作系统提供的互斥量 (Mutex) 来实现线程的阻塞和唤醒，以避免线程的空转和 CPU 资源的浪费。这种状态下，锁的获取和释放都比较慢。

锁状态	标志位	描述
无锁状态	01	对象没有被锁定，可以被任意线程访问
偏向锁状态	01	当一个线程获取了偏向锁并成功执行同步块后，锁标志位变为“01”，记录线程ID，后续该线程再次访问同步块无需重新获取锁
轻量级锁状态	00	当多个线程竞争同一个偏向锁时，偏向锁升级为轻量级锁状态。虚拟机在当前线程的栈帧中创建 Lock Record 记录当前对象的哈希值和指向 Lock Record 的指针。使用 CAS 操作尝试将对象头中的锁标志位设置为“00”，将 Lock Record 中的锁标志位设置为“01”。
重量级锁状态	10	轻量级锁获取失败，锁升级为重量级锁。虚拟机使用操作系统提供的互斥量实现线程的阻塞和唤醒，以避免线程的空转和 CPU 资源的浪费。



3. 实现原理 (monitor)

Java中的synchronized关键字是基于monitor机制实现的。每个Java对象都有一个monitor对象，用于实现对象的同步机制。monitor对象包括两个队列：入口队列和等待队列。当一个线程获得了对象的锁时，它就可以进入对象的monitor，并将monitor中的Owner变量设置为该线程。如果有其他线程试图访问被锁定的对象，它们就会被放入入口队列中，等待锁的释放。当锁被释放时，monitor就会从入口队列中取出一个线程，将该线程的Owner设置为该线程，并允许该线程进入monitor。如果入口队列为空，就会从等待队列中取出一个线程，并将该线程的Owner设置为该线程，并允许该线程进入monitor。

ReentrantLock

ReentrantLock是Java中的一个可重入锁，和synchronized一样，它可以用来实现线程间的同步。和synchronized相比，ReentrantLock提供了更多的灵活性和扩展性。下面是ReentrantLock的一些特点和使用方法：

1. 可重入性

ReentrantLock是可重入的，也就是说，一个线程可以重复获得已经获得的锁。

2. 公平锁和非公平锁

ReentrantLock可以创建公平锁和非公平锁。公平锁是指多个线程按照申请锁的顺序来获取锁，而非公平锁则允许抢占，即先申请的不一定先获得锁。在默认情况下，**ReentrantLock是非公平锁。**

3. 可中断性

ReentrantLock支持可中断锁，在等待锁的过程中，可以响应中断。这和synchronized不同，synchronized在等待锁的过程中不能被中断。

4. 超时性

ReentrantLock可以设置超时时间，在等待锁的过程中，如果超过了指定的时间还没有获得锁，则会放弃等待。

5. 实现原理

ReentrantLock是Java中的一种可重入锁，其实现是基于AQS（AbstractQueuedSynchronizer）的。在ReentrantLock中，AQS充当了同步器的角色，具体实现了ReentrantLock的同步机制。

核心内部类 Sync：ReentrantLock的同步器，继承自AQS。该类中定义了ReentrantLock的加锁和解锁方法，以及公平锁和非公平锁的实现。内部类实现保证了ReentrantLock是可重入的。

核心代码如下：

```
// 非公平锁
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread(); // 获取当前线程
    int c = getState(); // 获取当前锁的状态
    if (c == 0) { // 如果当前锁的状态为0，即没有线程持有锁
        if (compareAndSetState(0, acquires)) { // 利用CAS操作尝试获取锁
            setExclusiveOwnerThread(current); // 设置当前线程为锁的拥有者
            return true; // 获取锁成功
        }
    }
    else if (current == getExclusiveOwnerThread()) { // 如果当前线程已经持有锁
        int nextc = c + acquires; // 增加锁的重入次数
        if (nextc < 0) // 如果锁的重入次数超过了最大值
            throw new Error("Maximum lock count exceeded"); // 抛出异常
        setState(nextc); // 更新锁的状态
    }
}
```

```

        return true; // 获取锁成功
    }
    return false;
}

// 公平锁
protected boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread(); // 获取当前线程
    int c = getState(); // 获取当前锁的状态
    if (c == 0) { // 如果当前锁的状态为0, 即没有线程持有锁
        if (!hasQueuedPredecessors() && compareAndSetState(0, acquires)) {
            // 如果当前线程没有等待队列中的前驱节点 则 利用CAS操作尝试获取锁
            setExclusiveOwnerThread(current); // 设置当前线程为锁的拥有者
            return true; // 获取锁成功
        }
    }
    else if (current == getExclusiveOwnerThread()) { // 如果当前线程已经持有锁
        int nextc = c + acquires; // 增加锁的重入次数
        if (nextc < 0) // 如果锁的重入次数超过了最大值
            throw new Error("Maximum lock count exceeded"); // 抛出异常
        setState(nextc); // 更新锁的状态
        return true; // 获取锁成功
    }
    return false;
}

```

非公平锁在尝试获取锁时，首先判断锁是否被占用。如果锁未被占用，则尝试通过 CAS (Compare And Set) 操作获取锁，并设置当前线程为锁的拥有者。如果CAS操作成功，则获取锁成功。如果锁已被占用，则尝试检查当前线程是否已经持有锁。如果当前线程已经持有锁，则增加锁的重入次数，并返回获取锁成功。否则，获取锁失败。

公平锁在尝试获取锁时，首先判断当前线程是否有等待队列中的前驱节点。如果当前线程没有等待队列中的前驱节点，则尝试通过CAS操作获取锁，并设置当前线程为锁的拥有者。如果CAS操作成功，则获取锁成功。如果当前线程有等待队列中的前驱节点，则获取锁失败。

下面是synchronized和ReentrantLock 的比较

特性	synchronized	ReentrantLock
锁类型	内置锁	显式锁
锁的可重入性	可重入	可重入
锁的公平性	不保证公平性，可能会出现线程饥饿现象	可以选择非公平锁或公平锁

锁的粒度	粗粒度	可以通过创建多个锁实现细粒度控制
获取锁的方式	会自动获取和释放锁	需要手动获取和释放锁

乐观锁CAS

基本概念

CAS (Compare And Swap) 是一种乐观锁技术，用于实现多线程环境下的同步操作。CAS操作包括三个参数：内存位置（V）、预期原值（A）和新值（B）。当且仅当V的值等于A时，CAS操作才会将V的值更新为B；否则，CAS操作将不会做任何操作。CAS操作是一种原子操作，因此可以保证多线程环境下的数据安全性。

实现原理

在Java中，CAS操作是通过sun.misc.Unsafe类中的compareAndSwapObject()等方法实现的。这些方法使用了机器级别的原子操作指令，可以保证CAS操作的原子性。

CAS操作的实现原理可以简单地描述为：

1. 比较内存位置V的值是否等于预期原值A。
2. 如果V的值等于A，则将V的值更新为新值B。
3. 如果V的值不等于A，则CAS操作失败，需要重新进行比较和更新操作。

使用场景

CAS操作常用于实现无锁算法，如自旋锁和读写锁等。在多线程环境下，使用CAS操作可以避免锁的竞争，提高程序的并发性能。

CAS操作的一些使用场景包括：

- 实现线程安全的计数器。
- 实现线程安全的集合类，如ConcurrentHashMap等。
- 实现无锁算法，如ABA问题的解决方案。



悲观锁和乐观锁的比较

悲观锁：适用于竞争力度比较大的场景，但是申请悲观锁往往带来比较大的开销，而且线程的阻塞也会带来额外的开销

乐观锁：适用于竞争不激烈的场景，但是如果竞争比较大，可能会造成CPU空转，降低CPU的利用率。

volatile

1. volatile的作用

volatile关键字的主要作用是保证变量的可见性和有序性。

可见性

可见性是指一个线程对共享变量的修改对其他线程立即可见，而不是在未来的某个时刻才可见。如果一个变量被volatile修饰，那么对该变量的写操作将立即刷新到主内存中，而对该变量的读操作将直接从主内存中读取。

一个说明可见性的例子

```
public class VisibilityDemo {  
    private static boolean flag = true;  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread t1 = new Thread(() -> {  
            while (flag) {  
                // do something  
            }  
        });  
        t1.start();  
  
        Thread.sleep(1000); // 等待1秒钟  
  
        flag = false; // 更改flag的值  
  
        t1.join();  
    }  
}
```

在上面的代码中，我们定义了一个静态变量flag，并在一个线程中不断地检查该变量的值。在主线程中，我们等待1秒钟，然后将flag的值更改为false。如果Java的可见性机制正常工作的话，那么当flag的值被更改为false时，检查flag值的线程应该会立即感知到这

个变化，并退出循环。然而，实际上这个程序会一直运行下去，因为检查flag值的线程没有感知到flag的值已经被更改了。

这种现象是由于缓存导致的。当flag的值被更改为false时，这个变化可能只是发生在主内存中，而并没有立即被写入到检查flag值的线程所在的CPU缓存中。这意味着，即使flag的值已经被更改了，检查flag值的线程仍然会从CPU缓存中读取flag的旧值，而不是从主内存中读取flag的新值。由于flag的值始终是true，检查flag值的线程将永远不会退出循环。

为了解决这个问题，我们可以使用volatile关键字来修饰flag变量，即将第一行代码改为：

```
private static volatile boolean flag = true;
```

这样，当flag的值被更改时，Java虚拟机会保证该变化会立即被写入主内存中，并从主内存中读取flag的新值。这样一来，检查flag值的线程就可以感知到flag的值已经被更改了，并退出循环。

有序性

有序性是指代码在执行过程中的顺序性，即代码的执行顺序必须与编写时的顺序相同，否则会产生意料之外的结果。**volatile关键字可以保证代码执行的顺序性，因为JVM会保证volatile修饰的变量的读写操作是有序的。**

最经典的一个说明有序性的例子就是单例模式的双检锁了。

```
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;  
    }  
}
```

在双重检查锁定（Double-Checked Locking）的单例模式实现中，为了确保线程安全和避免不必要的锁定开销，会在实例化对象的代码块前后进行两次检查。如果在第一次检查时，发现实例化对象已经存在，就可以直接返回该对象，避免了锁定代码块。如果没有检查到实例化对象，就需要加锁并创建实例化对象。这样可以避免多个线程同时进入实例化对象的代码块，造成重复实例化对象的问题。

然而，双重检查锁定的实现中存在一个潜在的问题，即指令重排序。如果实例化对象的代码块中存在多条指令，那么在没有足够保障的情况下，可能会发生指令重排序。指令重排序可能会导致某些线程在获取到的实例化对象为空或不正确。虽然JVM规范中要求在数据竞争的情况下，指令不会被重排序，但是实际上并不是所有的JVM都会严格按照规范执行。

为了避免指令重排序问题，需要使用volatile关键字。使用volatile关键字可以确保该字段对于所有线程的可见性，即所有线程都能够读到该字段的最新值。同时，volatile关键字也可以保证字段的读写操作按照顺序执行，避免指令重排序问题。

1. 可见性的实现原理

volatile关键字通过强制刷新缓存和让编译器不对代码进行优化，从而实现了可见性。

在多线程环境下，每个线程都有自己的缓存，如果一个线程修改了某个变量，那么这个修改可能只是修改了线程的缓存，而没有立即写入主内存。而volatile修饰的变量，JVM会在对其进行写操作时，将其立即刷新到主内存中，从而让其他线程可以立即看到这个修改。

- 当一个线程修改了共享变量的值后，该变化会立即被写回主内存中，而不是先缓存在线程的工作内存中。同时，当其他线程需要访问这个共享变量时，它们会直接从主内存中读取最新的值，而不是从线程的工作内存中读取。
- 线程在执行 volatile 变量的读操作或写操作时，会自动将前面的操作发生的修改值立即刷回主内存中，以保证多个线程之间操作的有序性。

为了更好地理解可见性是如何实现的，我们可以先了解一下Java内存模型（Java Memory Model, JMM）。在JMM中，所有的共享变量都存储在主内存中，每个线程都有自己的工作内存，工作内存中保存了该线程使用到的共享变量的副本。线程对共享变量的所有操作都必须在自己的工作内存中进行，而不能直接读写主内存中的变量。

当一个线程需要读取共享变量时，它会从主内存中读取该变量的值，并将其复制到自己的工作内存中；当一个线程需要写入共享变量时，它会先将值写入到自己的工作内存中，然后再将这个值刷新到主内存中。这样就会出现一个问题，就是当一个线程修改了共享变量

的值后，其他线程可能无法立即看到这个变化，因为它们可能还在使用自己工作内存中的共享变量副本。

volatile关键字的作用就是告诉JVM，每次访问该变量时都必须从主内存中读取，而不是从工作内存中读取，同时在写操作时也必须立即将结果刷新到主内存中，保证多线程之间的可见性。这样，当一个线程修改了共享变量的值后，其他线程就能立即看到这个变化了。

2. 有序性如何实现

在Java内存模型中，由于CPU为了提高执行效率，可能会对指令进行重排序，包括编译器和处理器级别的重排序。在单线程中，这种重排序并不会导致问题，但在多线程中，由于多个线程可以同时访问共享变量，重排序可能会导致数据的不一致性和并发问题。

而volatile关键字可以保证被修饰变量的可见性和有序性。上一部分我们已经详细说明了volatile的可见性是如何实现的，现在我们重点讲解volatile是如何保证有序性的。

在JVM底层，volatile会将代码的执行顺序限制在其前后插入内存屏障（Memory Barrier，又称内存栅栏），内存屏障是一种CPU指令，用于限制指令重排，确保指令的顺序性。具体来说，当CPU执行到内存屏障时，它会确保前面的所有指令都已经执行完成，后面的指令则必须等待内存屏障执行完毕才能开始执行。

在Java中，内存屏障由JVM实现，可以分为以下几种类型：

1. Load Barrier（读屏障）：保证在读取操作执行后，后续的读写操作都必须在它之后执行；
2. Store Barrier（写屏障）：保证在写入操作执行前，前面的读写操作都必须在它之前执行；
3. Full Barrier（全屏障）：同时包含读写屏障，保证读写操作之间的顺序关系。

在使用volatile修饰的变量进行读操作时，会执行一个Load Barrier，确保volatile变量的读操作不会被其后面的读写操作重排到它前面。同样，在使用volatile修饰的变量进行写操作时，会执行一个Store Barrier，确保volatile变量的写操作不会被其前面的读写操作重排到它后面。

因此，通过内存屏障的限制，volatile可以保证被修饰变量的读写顺序与程序代码中的顺序一致，从而实现有序性。

比如在上述的DCL模式中：

由于instance是volatile变量，线程A在执行完instance = new Singleton()操作之后，会将

新对象的引用立即写入主内存中。而线程B在调用getInstance方法时，会先读取instance的值，由于instance是volatile变量，它会在读取操作前插入一个Load Barrier，以确保读取操作在所有之前的写入操作之后执行。因此，线程B能够读取到线程A已经创建并写入主内存中的单例对象

happens-before

happens-before是Java内存模型中用来描述多线程之间执行顺序的规则，它可以用来保证程序的正确性和可预测性。简单来说，如果操作A happens-before 操作B，那么操作A在时间上发生在操作B之前。

happens-before规则的作用是，保证在一个线程中观察到的结果对于另一个线程是可见的，并且避免出现不确定的结果。这些规则描述了在一个线程中执行的操作如何与在另一个线程中执行的操作相关联，以及这些操作之间可能存在的任何可能的竞争条件。这些规则可以帮助我们理解程序中多个线程之间的交互，并确保这些交互不会产生问题。

JVM的happens-before规则如下：

1. 程序顺序规则：一个线程内，按照程序代码顺序，书写的前面的操作先行发生于书写的后面的操作。
2. 锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作。
3. volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作。
4. 传递性规则：如果操作A先行发生于操作B，操作B先行发生于操作C，那么操作A先行发生于操作C。
5. 线程启动规则：Thread对象的start()方法先行发生于此线程的每个动作。
6. 线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，可以通过Thread.join()方法是否结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。
7. 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。
8. 对象终结规则：一个对象的初始化完成先行发生于它的finalize()方法的开始。
9. 先行发生原则的传递性：如果操作A先行发生于操作B，操作B先行发生于操作C，那么操作A先行发生于操作C。

ps：这些规则不用全部背诵，面试不会让你背诵的，过一遍即可。

ThreadLocal

概念以及使用场景

ThreadLocal是Java中的一个线程间数据传递的工具类，它可以使线程之间的数据隔离，即每个线程中都可以有自己的一个数据副本。

在多线程场景下，有这样一种需求：在同一个线程中，希望能够共享一些数据，而不是将数据暴露到其他线程中。这时可以使用ThreadLocal来实现。

ThreadLocal最常见的使用场景就是在Web应用中，将HttpServletRequest和HttpServletResponse等对象存储在ThreadLocal中，以便于它们在整个请求的处理过程中都可以访问到。

以下是一个简单的ThreadLocal使用样例：

```
public class UserContext {  
    private static final ThreadLocal<User> currentUser = new ThreadLocal<>();  
  
    public static User getCurrentUser() {  
        return currentUser.get();  
    }  
  
    public static void setCurrentUser(User user) {  
        currentUser.set(user);  
    }  
  
    public static void clear() {  
        currentUser.remove();  
    }  
}
```

在这个样例中，我们定义了一个名为UserContext的类，其中包含了三个静态方法。其中，getCurrentUser()方法用于获取当前线程绑定的User对象，setCurrentUser()方法用于将指定的User对象绑定到当前线程上，clear()方法用于清除当前线程绑定的User对象。

在这个样例中，我们通过ThreadLocal实现了每个线程中独立的User对象，而不需要考虑线程安全问题。每个线程中只需要调用UserContext类的setCurrentUser()方法来绑定当前线程的User对象即可。这样，在程序的任何地方都可以通过UserContext类的

`getCurrentUser()`方法来获取当前线程绑定的User对象，而不需要传递User对象作为参数。

例如，可以这样使用：

```
// 在某个请求的处理函数中将当前用户绑定到当前线程  
UserContext.setCurrentUser(currentUser);  
  
// 在程序的任何地方都可以通过UserContext类来获取当前用户  
User currentUser = UserContext.getCurrentUser();
```

这样，就可以方便地实现每个线程中独立的User对象，而不需要考虑线程安全问题。

实现原理

ThreadLocal的基本思想是，在每个线程中都创建一个独立的变量副本。当多个线程同时访问这个变量时，每个线程都使用自己的变量副本，线程之间互不干扰，从而避免了线程同步和互斥的问题。 ThreadLocal实现这个功能的关键是要确保每个线程都拥有自己的变量副本，而不会相互干扰。**ThreadLocal的实现原理主要包括两个部分：ThreadLocal类的实现和Thread类的实现。**

ThreadLocal类的实现：

ThreadLocal类的核心是一个ThreadLocalMap对象，它用来存储每个线程的变量副本。 ThreadLocalMap是一个自定义的Map，它的键是ThreadLocal对象，值是线程的变量副本。每个ThreadLocal对象都会在调用ThreadLocal的get()或set()方法时，被存储到当前线程的ThreadLocalMap对象中。

ThreadLocalMap对象是在每个线程中单独创建的，因此每个线程都有自己独立的ThreadLocalMap对象。 当多个线程同时访问同一个ThreadLocal对象时，它们分别使用自己的ThreadLocalMap对象，从而保证了线程之间的隔离。

Thread类的实现：

Thread类中有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，它用来存储当前线程中所有的ThreadLocal对象及其对应的变量副本。 当调用ThreadLocal的get()方法时，Thread类会首先获取当前线程中的ThreadLocalMap对象，然后根据当前ThreadLocal对象在ThreadLocalMap中的索引位置获取对应的变量副本。当调用ThreadLocal的set()方法时，Thread类也会先获取当前线程中的ThreadLocalMap对象，然后根据当前ThreadLocal对象在ThreadLocalMap中的索引位置存储变量副本。



内存泄漏及解决方案

不正确地使用ThreadLocal很容易导致内存泄漏的出现，因为每个线程有一个独立的ThreadLocalMap，我们使用ThreadLocal的时候，set()进去的数据实际上是存储在ThreadLocalMap里面的，换句话说，就算threadLocal被回收，只要线程不被回收，ThreadLocalMap中的数据就会一直存在。而在实际开发过程中，往往会使用线程池，也就是线程是不会被GC回收的，因此容易导致内存泄漏的出现。（当然，如果线程被回收了，ThreadLocalMap肯定也就没了，自然不会有内存泄漏的问题）。

解决方案：使用完ThreadLocal以后，要调用remove()方法。

下面是一个代码示例

```
static ThreadLocal<int[]> currentUser1 = new ThreadLocal<>();

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(100);
    for (int i = 0; i < 100; i++) {
        executorService.execute(new Thread(()->{
            System.out.println("任务");
            currentUser1.set(new int[1024 * 1024]);
            currentUser1.remove(); // 如果不加这一行代码，可能会造成内存泄露，导致内存溢出
        }));
    }
}
```

JVM

内存分区

JVM的内存分为以下几个区域：

线程私有

程序计数器

程序计数器是一块较小的内存区域，用于存储当前线程正在执行的Java方法的JVM指令地址。由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式实现

的，因此每个线程都需要有独立的程序计数器，各个线程之间互不干扰，独立存储。

虚拟机栈

虚拟机栈也是一块线程私有的内存区域，其描述的是Java方法执行的内存模型：每个方法执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个线程在执行Java方法时，都会随之创建一个栈帧，并且为该线程所独享。在Java虚拟机规范中，对于Java虚拟机栈的规定是非常严格的：每个线程的Java虚拟机栈容量可以固定，也可以是动态扩展的，但在创建线程时可以设置Java虚拟机栈容量的大小。如果线程请求的栈深度大于Java虚拟机所允许的深度，则抛出StackOverflowError异常；如果Java虚拟机栈可以动态扩展，当扩展时无法申请到足够的内存时，则抛出OutOfMemoryError异常。

本地方法栈

本地方法栈与虚拟机栈所发挥的作用是非常相似的，区别是虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为虚拟机使用到的Native方法服务。在Java虚拟机规范中，对于本地方法栈的规定与虚拟机栈的规定是非常类似的。

线程共享

堆



Java堆是Java虚拟机所管理的内存中最大的一块。由于现在的垃圾收集器都采用分代收集算法，所以Java堆中还可以细分为新生代和老年代。新生代分为Eden空间、From Survivor空间和To Survivor空间。大多数情况下，对象都会在Eden区中被创建出来，当Eden区没有足够的空间进行分配时，虚拟机将发起一次Minor GC。

方法区

方法区也称为永久代，用于存储已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap（非堆），目的是与Java堆区分开来。方法区的大小可以通过启动参数“-XX:PermSize”和“-XX:MaxPermSize”来调整。

直接内存

直接内存并不是虚拟机运行时数据区的一部分，但是这部分内存也被频繁地使用。直接内存的使用方式是通过Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。这样能显著提高性能，因为避免了在Java堆和Native堆之间来回复制数据。但是需要注意的是，直接内存的分配不会受到Java堆大小的限制，但是会受到本机总内存大小以及处理器寻址空间的限制。

垃圾回收机制

Java的垃圾回收机制是指Java虚拟机（JVM）自动管理计算机内存的过程。在JVM中，所有的对象都被分配在堆中。当对象不再被引用时，JVM会自动回收这些对象所占用的内存。垃圾回收机制是Java语言的一项重要特性，它可以大大降低程序员的工作量，减少内存泄漏和内存溢出等问题的发生。

在Java中，垃圾回收机制是通过GC（Garbage Collection）来实现的。JVM中有一个垃圾回收器，它会定期扫描堆中的对象，标记哪些对象是“可回收”的，然后释放它们所占用的内存。Java中的垃圾回收机制是基于引用计数的，即当一个对象的引用计数为0时，就会被回收。但是，在实际开发中，引用计数机制并不完美，因为它无法处理循环引用的情况。因此，Java也采用了其他的垃圾回收算法，如标记-清除算法、复制算法、标记-整理算法等。

如何判断对象是否可回收

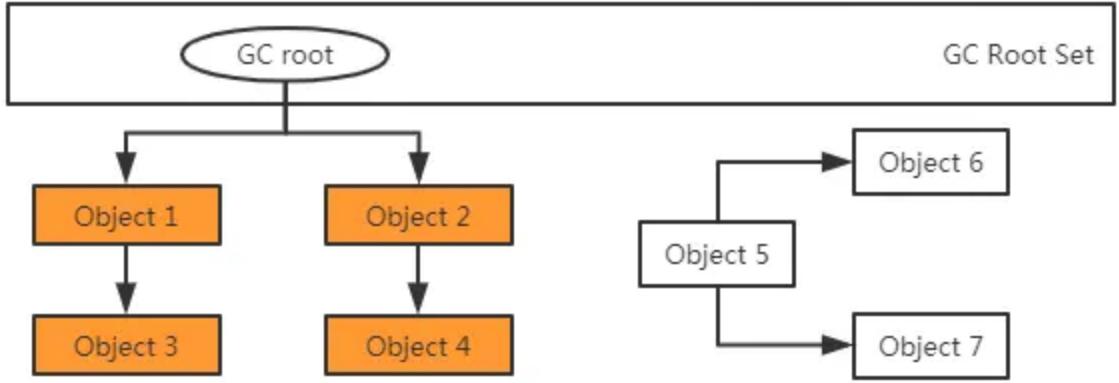
引用计数法

引用计数法是一种最简单的垃圾回收算法。它的基本思路是为每个对象维护一个引用计数器，当有一个新的引用指向该对象时，计数器加一；当一个引用失效时，计数器减一。当某个对象的引用计数器变为零时，就表明这个对象已经成为垃圾，可以被回收。

但是，引用计数法存在一个很明显的问题，就是无法处理循环引用的情况。例如，如果对象A引用对象B，对象B引用对象A，那么这两个对象的引用计数器都不为零，尽管它们已经不再被任何其他对象所引用，也就是说，它们不应该被回收。

可达性分析法

可达性分析法是目前主流的垃圾回收算法。它的基本思路是通过一系列的称为GC Roots的对象作为起始点，从这些节点开始向下搜索，搜索过程中所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，说明这个对象已经不可用，可以被回收。



GC Roots包括以下几种：



1. 虚拟机栈中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中JNI引用的对象

通过这些GC Roots可以标记出所有的活动对象，未被标记的对象则被判定为垃圾对象，可以被回收。

可达性分析法相较于引用计数法的最大优点是可以解决循环引用的问题，因为只有被GC Roots直接或间接引用的对象才会被认为是活动对象，而循环引用会形成一个孤岛，无法被GC Roots找到，因此也不会影响垃圾回收。

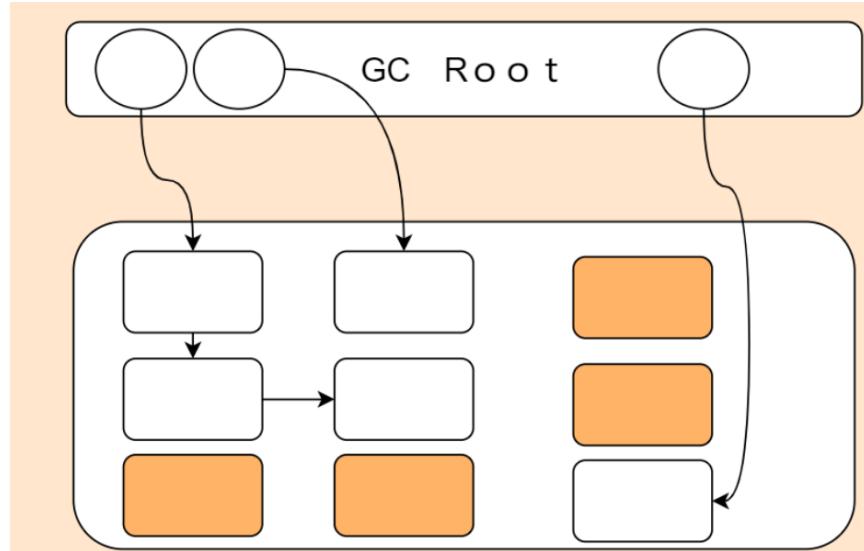
垃圾收集算法

垃圾收集算法是指在进行垃圾回收时，如何确定哪些对象是垃圾对象，并进行回收的具体算法。常见的垃圾收集算法包括**标记清除法、复制算法和标记整理法**。

标记清除法

标记清除法是最基础的垃圾收集算法。其基本思路是分两个阶段：**标记**和**清除**。**标记阶段**从根节点开始遍历，标记所有可达对象，未被标记的对象即为垃圾对象。**清除阶段**将所有垃圾对象从堆中清除。

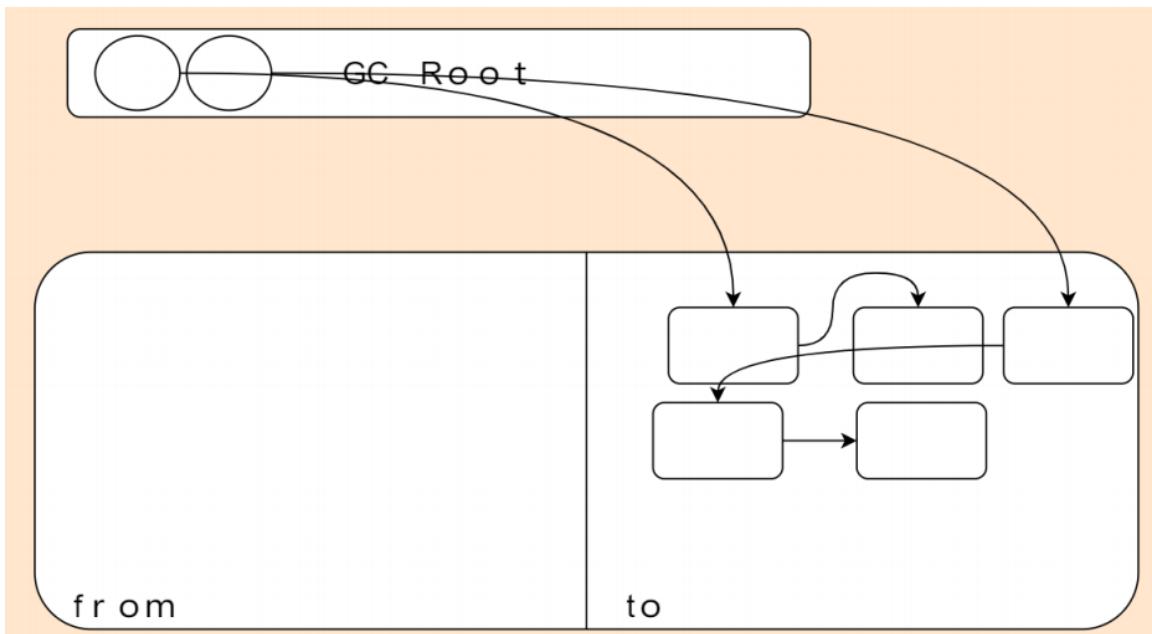
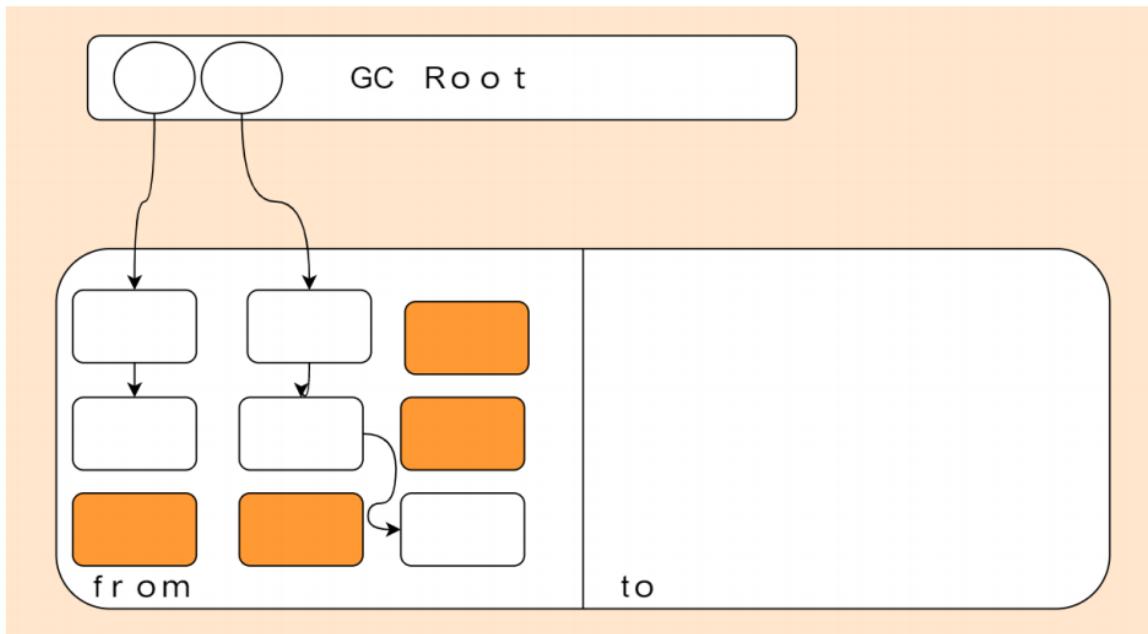
标记清除法的主要优点是实现简单。但是，由于其是基于“标记-清除”的思路实现的，因此会存在“碎片化”问题。即，由于被清除的垃圾对象在堆中会留下一块空间，而这些空间在下一次分配内存时可能无法被充分利用，从而导致内存空间的浪费。



复制算法

复制算法是一种适用于新生代的垃圾收集算法。其基本思路是将新生代分为两个相等大小的区域，一半为From区域，一半为To区域。在进行垃圾回收时，将所有存活对象从From区域复制到To区域，然后清除From区域中的所有对象。这样做的好处是避免了碎片化问题，同时也能够保证分配内存的效率。

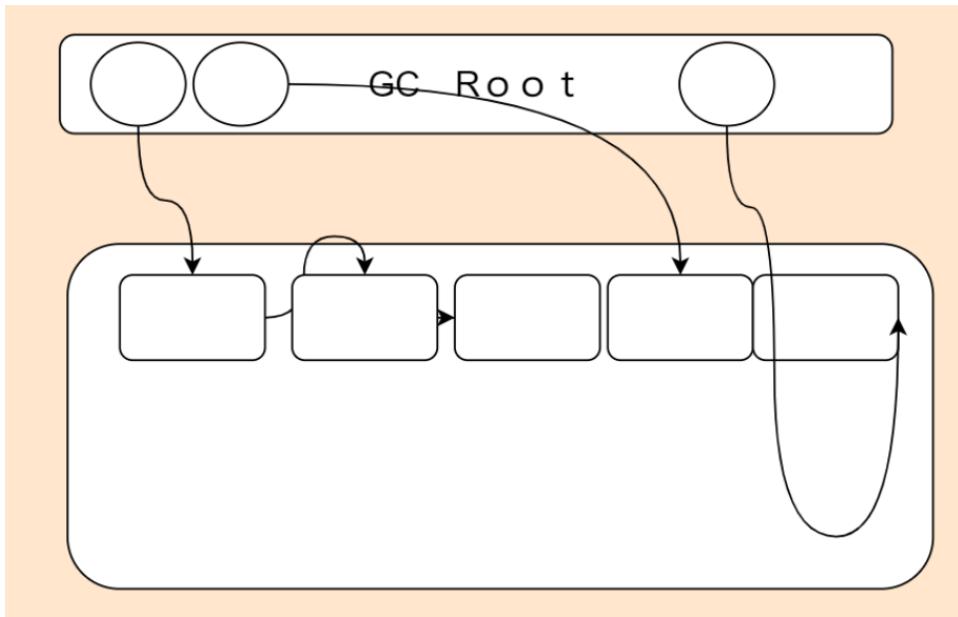
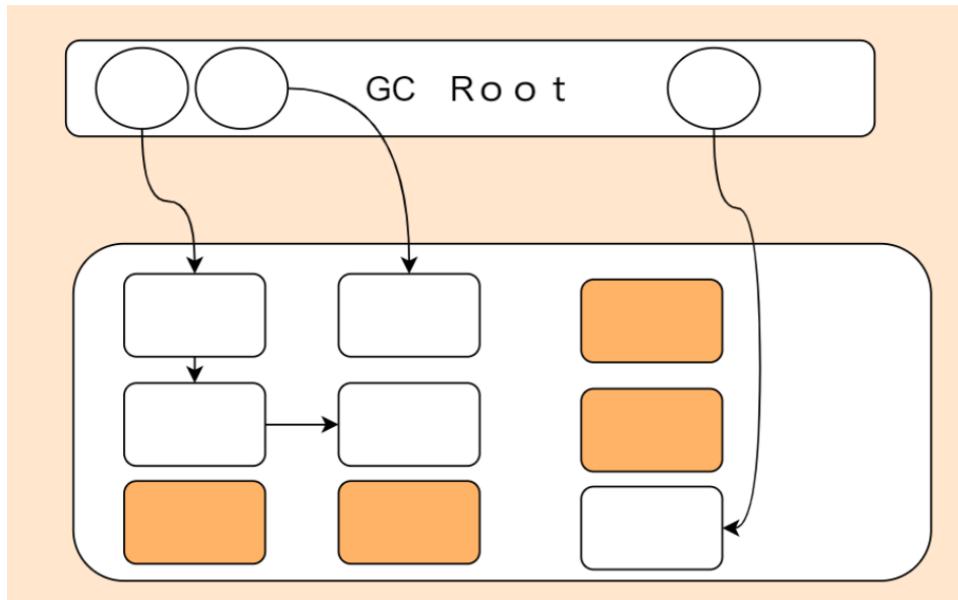
复制算法的主要缺点是只能适用于新生代的垃圾回收，对于老年代的垃圾回收并不适用。原因是老年代中的对象存活率较高，复制算法的效率会受到很大的影响。



标记整理法

标记整理法是一种适用于老年代的垃圾收集算法。其基本思路是在标记阶段与标记清除法相同，但是在清除阶段会将所有存活对象向一端移动，然后清除掉边界以外的所有对象。这样做的好处是避免了碎片化问题，并且也能够保证分配内存的效率。

标记整理法的主要缺点是在对象移动的过程中，需要调整所有指向这些对象的指针，因此其效率较低。



垃圾收集算法	优点	缺点
标记清除法	实现简单	碎片化问题，内存空间浪费
复制算法	解决了碎片化问题，保	只适用于新生代垃圾回收，对老年代的垃圾回收效

	证了分配内存的效率	率低（因为老年代的对象存活时间久）
标记整理法	解决了碎片化问题，保证了分配内存的效率	对存活对象进行移动操作，效率较低

垃圾回收器

STW

在执行垃圾回收时，Java虚拟机需要暂停所有的应用线程，以确保垃圾回收器能够正常工作。这种暂停被称为“STW（Stop The World）”事件。

STW事件的发生会导致应用程序暂停，可能会对应用程序的性能和可用性产生负面影响。因此，开发人员需要在编写应用程序时考虑垃圾回收的影响，并在必要时进行优化。

STW事件通常发生在以下情况下：

- 在执行Full GC时，Full GC需要清除整个堆空间，因此需要STW事件来暂停所有的应用线程。
- 在进行Young GC时，虚拟机会将所有存活的对象复制到Survivor区域中，同时清空Eden区域和From Survivor区域。这也需要STW事件来暂停所有的应用线程。

由于Full GC时间比较长，因此我们要尽可能减少Full GC的发生。开发人员可以通过优化代码，减少内存泄漏等问题来减少Full GC的发生。

常见垃圾收集器

Java虚拟机中的垃圾收集器主要分为两种：基于分代假设的收集器和基于全堆搜索的收集器。其中，分代假设是指大部分对象的生命周期都是短暂的，而只有少部分对象是长期存活的。因此，将堆内存分为新生代和老年代，对不同代的对象采用不同的收集算法，可以提高垃圾回收的效率。

Serial收集器

原理：

- Serial收集器是一种单线程垃圾收集器，采用复制算法实现。
- 会将年轻代分为一个较大的Eden区和两个较小的Survivor区，并将对象按照年龄大小放入不同的区域。



- 在进行垃圾回收时，将所有存活的对象复制到另一个 Survivor 区或者年龄大于等于2的对象放入老年带。

特点：

- 实现简单，内存利用率高。
- 适合于小型应用或者客户端应用。
- 无法处理大对象。
- 暂停时间较长，因为是单线程执行垃圾回收。

应用场景：

- 对于小型应用或者客户端应用，可以使用Serial收集器进行垃圾回收。
- 适合于新生代的垃圾回收。

ParNew收集器

原理

- ParNew收集器是Serial收集器的多线程并行版本，采用复制算法实现。
- 会将年轻代分为一个较大的 Eden 区和两个较小的 Survivor 区，并将对象按照年龄大小放入不同的区域。
- 在进行垃圾回收时，将所有存活的对象复制到另一个 Survivor 区或者年龄大于等于2的对象放入老年带。

特点

- 与Serial收集器相比，具有更好的吞吐量，可以利用多核CPU。
- 可以与CMS收集器配合使用，实现对整个Java堆的垃圾回收。

应用场景

- 适合于对暂停时间要求不高，但需要高吞吐量的场景，比如后台数据处理等。
- 适合于新生代和老年带的垃圾回收。

Parallel Scavenge收集器

原理：

- Parallel Scavenge 收集器是一种并行的新生代垃圾收集器，采用复制算法实现。
- 与 ParNew 收集器类似，将年轻代分为一个较大的 Eden 区和两个较小的 Survivor 区。
- 在进行垃圾回收时，会使用多线程并行地执行垃圾回收，提高垃圾回收的吞吐量。

特点：

- 可以利用多核CPU，提高垃圾回收效率。
- 适合于大规模应用，对吞吐量要求高的场景。
- 暂停时间较长，因为需要进行大量的垃圾回收操作。

应用场景：

- 适合于对暂停时间要求不高，但需要高吞吐量的场景，比如后台数据处理等。
- 适合于新生代的垃圾回收。

Serial Old收集器

原理：

- Serial Old收集器是Serial收集器的老年代版本，采用标记整理算法实现。
- 在进行垃圾回收时，先标记出所有存活的对象，然后将所有存活的对象移到一端，将未标记的对象全部清理掉。

特点：

- 实现简单，内存利用率高。
- 适合于小型应用或者客户端应用。
- 无法处理大对象。
- 暂停时间较长，因为是单线程执行垃圾回收。

应用场景：

- 对于小型应用或者客户端应用，可以使用Serial Old收集器进行垃圾回收。
- 适合于老年代的垃圾回收。

Parallel Old收集器

原理：

- Parallel Old收集器是Parallel Scavenge收集器的老年代版本，采用标记整理算法实现。
- 在进行垃圾回收时，先标记出所有存活的对象，然后将所有存活的对象移到一端，将未标记的对象全部清理掉。

特点：

- 可以利用多核CPU，提高垃圾回收效率。
- 适合于大规模应用，对吞吐量要求高的场景。
- 暂停时间较长，因为需要进行大量的垃圾回收操作。

应用场景：

- 适合于对暂停时间要求不高，但需要高吞吐量的场景，比如后台数据处理等。
- 适合于老年代的垃圾回收。

CMS收集器

原理：

CMS（Concurrent Mark Sweep）收集器是一种并发垃圾收集器，它采用标记清除算法实现。与其他垃圾收集器不同，CMS垃圾收集器在垃圾回收过程中不需要暂停应用程序。这种并发垃圾回收器的出现，解决了应用程序暂停时间过长的问题，提高了应用程序的响应速度。

下面是CMS垃圾收集器的工作流程：

- **初始标记（CMS initial mark）**

初始标记是CMS垃圾收集器的第一步，其目的是标记所有直接与根对象相关联的对象。在执行初始标记时，应用程序线程会暂停，直到标记完成。这个过程是短暂的，不会对应用程序的响应速度造成太大的影响。

- **并发标记（CMS concurrent mark）**

并发标记是CMS垃圾收集器的第二步，其目的是标记所有存活的对象。在执行并发标记时，应用程序可以继续执行，不需要暂停。同时，CMS垃圾收集器使用多个线程来执行

标记操作，提高标记的效率。

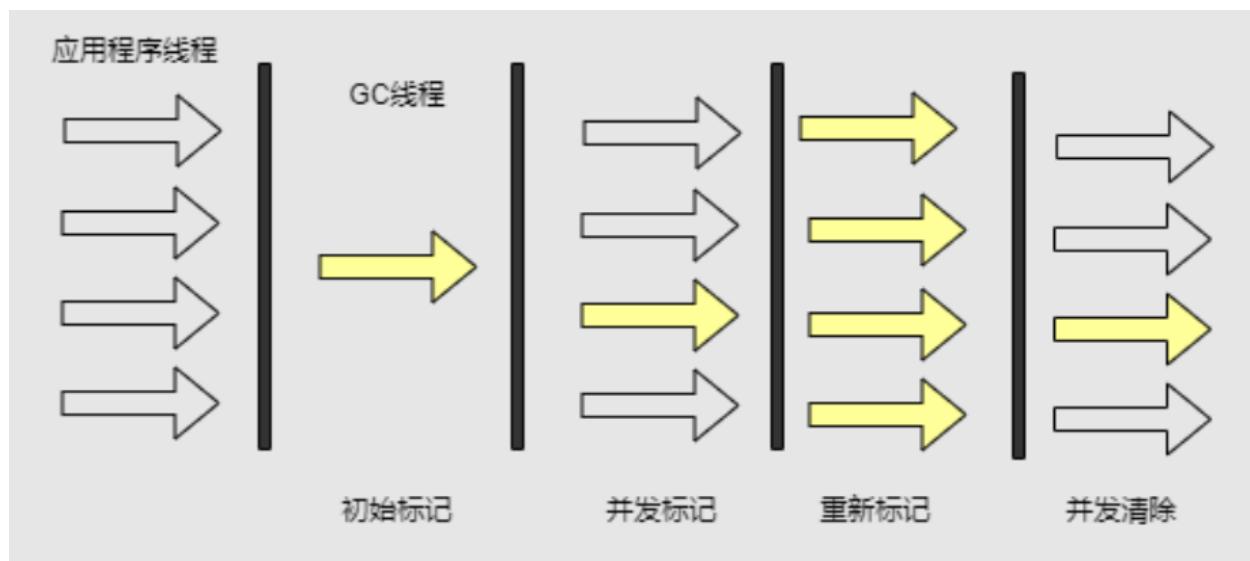
- **重新标记 (CMS remark)**

重新标记是CMS垃圾收集器的第三步，其目的是标记在并发标记期间，由于应用程序的运行，而导致标记状态发生变化的对象。在执行重新标记时，应用程序线程会暂停，直到标记完成。这个过程通常比初始标记更快，但仍可能对应用程序的响应速度造成一定的影响。

- **并发清除 (CMS concurrent sweep)**

并发清除是CMS垃圾收集器的最后一步，其目的是清除所有未被标记的对象。在执行并发清除时，应用程序可以继续执行，不需要暂停。同时，CMS垃圾收集器使用多个线程来执行清除操作，提高清除的效率。

需要注意的是，并发清除阶段可能会导致内存碎片化问题，因此CMS垃圾收集器通常需要与其他垃圾收集器（如Parallel Scavenge收集器）配合使用。同时，由于CMS垃圾收集器的并发特性，其性能会受到多个因素的影响，需要开发人员仔细考虑其使用场景。



特点：

- 可以最大限度地减少垃圾回收的暂停时间，对响应时间要求高的应用非常有用。
- 无法处理浮动垃圾，可能会导致内存碎片。

应用场景：

- 适合于对暂停时间要求高、响应时间要求高的应用，比如实时系统、交互式应用等。

- 适合于堆内存较大、应用运行时间较长的场景。

G1收集器

G1 (Garbage First) 收集器是一种以吞吐量和响应时间为 目标 的垃圾收集器。它的特点是：在进行垃圾回收时，可以同时兼顾高吞吐量和短暂停顿时间。G1收集器主要针对堆内存较大、数据量较大的应用程序进行垃圾回收。

原理

G1收集器的基本原理是将堆内存分为多个区域（Region），每个区域的大小可以动态调整。在进行垃圾回收时，G1收集器会根据垃圾分布情况，优先清理垃圾占比较高的区域，以最大限度地提高垃圾回收效率。同时，G1收集器还会根据用户设定的目标时间，动态地调整垃圾回收的策略，以保证系统的响应时间。

下面是G1垃圾收集器的工作流程：

- **初始标记 (Initial Mark)**

初始标记是G1垃圾收集器的第一步，其目的是标记所有直接与根对象相关联的对象，并标记出应用程序停顿时间预测值。在执行初始标记时，应用程序线程会暂停，直到标记完成。这个过程是短暂的，不会对应用程序的响应速度造成太大的影响。

- **并发标记 (Concurrent Marking)**

并发标记是G1垃圾收集器的第二步，其目的是标记所有存活的对象。在执行并发标记时，应用程序可以继续执行，不需要暂停。同时，G1垃圾收集器使用多个线程来执行标记操作，提高标记的效率。

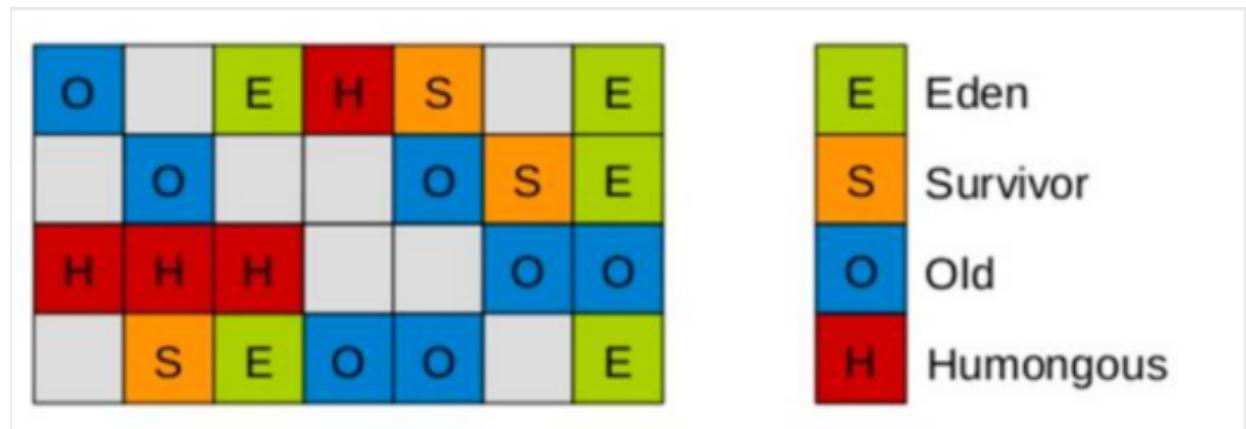
- **最终标记 (Final Marking)**

最终标记是G1垃圾收集器的第三步，其目的是标记在并发标记期间，由于应用程序的运行，而导致标记状态发生变化的对象。在执行最终标记时，应用程序线程会暂停，直到标记完成。这个过程通常比初始标记更快，但仍可能对应用程序的响应速度造成一定的影响。

- **筛选回收 (Live Data Counting and Evacuation)**

筛选回收是G1垃圾收集器的最后一步，其目的是清理所有未被标记的对象，并且将存活的对象复制到一组新的区域中。在执行筛选回收时，应用程序可以继续执行，不需要暂

停。同时，G1垃圾收集器使用多个线程来执行清除和复制操作，提高清除和复制的效率。



特点

G1收集器的主要特点有：

- 可以兼顾高吞吐量和短暂停顿时间
- 可以动态调整区域的大小
- 可以根据用户设定的目标时间，动态地调整垃圾回收的策略
- 可以避免碎片化问题，同时也不会牺牲分配内存的效率。

应用场景

G1收集器主要适用于堆内存较大、数据量较大的应用程序。由于G1收集器可以兼顾高吞吐量和短暂停顿时间，因此适用于对系统响应时间要求较高但对吞吐量要求也较高的场景。同时，G1收集器还可以避免碎片化问题，因此适用于需要长时间运行的应用程序。

GC调优

GC (Garbage Collection) 调优是指对Java应用程序的垃圾回收机制进行调优，以达到更好的性能和更少的内存使用。在Java应用程序中，GC是一项重要的功能，它可以自动地回收不再使用的内存空间，从而避免内存泄漏和内存溢出的问题。但是，如果垃圾回收机制不得当，就可能会影响应用程序的性能和响应速度。

假设我们的一个Java应用程序，出现了频繁的GC，导致程序响应变慢甚至出现了内存溢出的情况。我们需要对该应用程序进行GC调优，使其运行更加稳定和高效。大概思路有以下几点：

1. 增大堆内存的大小，或者修改新生代和老年代的比例。
2. 选择更加适合的垃圾收集器，或者调节垃圾收集器的参数。
3. 通过内存分析工具，判断是否存在内存泄漏或者内存占用过大的代码。

类加载机制

Java虚拟机的类加载机制是指将Java类的.class文件加载到Java虚拟机中，并将其转换为在Java虚拟机中使用的Java类的过程。类加载机制是Java虚拟机的一个核心概念，也是Java语言的一个基础特性。Java虚拟机的类加载机制由三个步骤组成：加载、连接、初始化。

加载

类加载的第一步是加载，即通过类的完全限定名来获取该类的二进制字节流。这个字节流可以从本地文件系统、网络或其他来源获取。字节流中包含了该类的所有信息，包括类的名称、方法、字段、注解等。当Java虚拟机加载一个类时，它会先检查该类是否已经被加载过。如果该类还没有被加载过，Java虚拟机就会使用类加载器加载该类。

连接

类加载的第二步是连接，即将类的二进制字节流转换为Java虚拟机中可以使用的Java类。连接过程包括三个阶段：验证、准备和解析。

- 验证阶段：在验证阶段，Java虚拟机会确保该类的二进制字节流符合Java虚拟机规范，并且不会危害Java虚拟机的安全。
- 准备阶段：在准备阶段，Java虚拟机会为该类中的所有静态变量分配内存，并设置默认值。
- 解析阶段：在解析阶段，Java虚拟机会将类中的符号引用转换为直接引用。

初始化

类加载的第三步是初始化，即执行类的初始化代码。在Java虚拟机执行类的初始化代码时，会对静态变量进行赋值，并执行静态代码块中的代码。类的初始化是在Java虚拟机

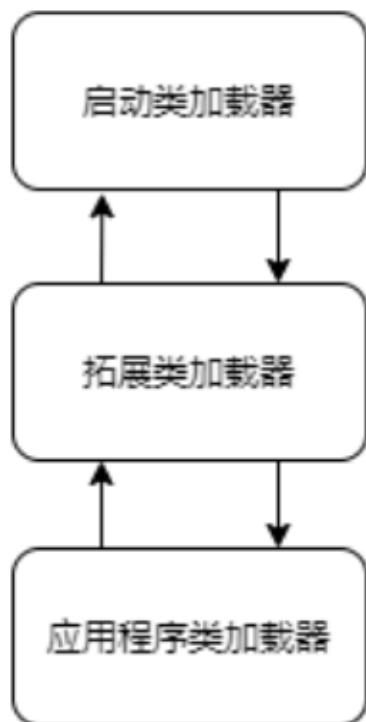
第一次使用该类时执行的。当Java虚拟机需要使用某个类时，如果该类还没有被初始化，Java虚拟机就会触发该类的初始化。

类加载器的分类

在Java虚拟机中，类加载器分为四种类型：Bootstrap ClassLoader、Extension ClassLoader、System ClassLoader和用户自定义类加载器。其中，Bootstrap ClassLoader是Java虚拟机的内置类加载器，用于加载Java虚拟机运行时需要的类；Extension ClassLoader和System ClassLoader是Java虚拟机的标准类加载器，用于加载Java平台的核心类库和应用程序类；用户自定义类加载器则是由应用程序自己实现的。

双亲委派机制

在Java虚拟机中，类加载器之间采用了一种叫做双亲委派机制的方式来避免类的重复加载。当Java虚拟机需要加载一个类时，它会先将该请求委派给父类加载器。如果父类加载器无法加载该类，Java虚拟机才会将该请求委派给子类加载器。通过双亲委派机制，Java虚拟机可以确保同一个类只会被加载一次，避免了类的重复加载和冲突。



启动类加载器

C++实现的。负责加载Java_HOME/lib目录中的类。

拓展类加载器

Java_HOME/lib/ext包下的类

应用程序类加载器

用户类路径上的指定的类库。



双亲委派机制的工作原理如下：

1. 当一个类需要被加载时，先向其父类加载器发起加载请求。
2. 如果父类加载器可以加载该类，则直接返回该类的Class对象。
3. 如果父类加载器无法加载该类，则将该请求转发给自身的加载器进行加载。
4. 如果自身的加载器可以加载该类，则返回该类的Class对象。
5. 如果自身的加载器无法加载该类，则将请求再次转发给父类加载器的请求，重复上述步骤，直到Bootstrap ClassLoader为止。

通过这种机制，Java的类加载器可以保证JVM中只存在唯一的类定义，防止类的重复加载和冲突，同时也可以提高类加载器的效率。

双亲委派机制的打破

然而，在某些情况下，我们需要打破双亲委派机制，即在某些情况下，我们需要自己来加载类定义，而不是委派给父类加载器。

例如，假设我们在Web应用程序中需要使用一个名为"foo.jar"的类库，而Tomcat服务器的公共类路径下已经存在了不同版本的"foo.jar"，为了避免类库冲突，我们需要在Web应用程序中使用特定版本的"foo.jar"。这时，我们可以通过打破双亲委派机制来实现这个目标，具体方法是创建一个新的类加载器，并使用这个类加载器来加载特定版本的"foo.jar"，从而避免与Tomcat服务器中其他版本的"foo.jar"发生冲突。

需要注意的是，打破双亲委派机制可能会带来一些安全风险，因此在使用时需要谨慎考虑，并且尽可能使用其他方式来解决类库冲突问题。

热部署【拓展】

热部署是指在应用程序运行过程中，对Java类进行修改，并且在不重启应用程序的情况下使这些修改生效的过程。

Java中的类加载器（ClassLoader）可以加载指定目录下的字节码文件，并将其转换成Java类的实例。因此，在实现热部署时，可以通过自定义类加载器，将新的类定义加载到JVM中，从而实现动态更新代码。

需要注意的是，在实现热部署时，要避免类的冲突和重复加载。因此，可以采用一些技术手段，如使用版本号或时间戳来标识类定义的版本，避免类的重复加载。同时，为了避免类的冲突，可以采用不同的类加载器加载不同的类定义，从而保证类的唯一性。

JVM常见命令

1.jstack

功能：生成当前Java进程中所有线程的堆栈跟踪信息，用于诊断线程相关的问题。

使用方法：在命令行中输入 `jstack [pid]`，其中pid是Java进程的进程ID，可以使用jps命令查看。

2.jstat

功能：用于监控JVM的各种统计信息，包括堆内存使用情况、GC信息、类加载器信息等。

使用方法：在命令行中输入 `jstat [options] [pid] [interval] [count]`，其中options是指定需要监控的统计信息，pid是Java进程的进程ID，interval是监控间隔，count是监控次数。

3.info

功能：显示Java进程的系统属性和JVM参数信息。

使用方法：在命令行中输入 `jinfo [options] [pid]`，其中options可以指定需要显示的信息，pid是Java进程的进程ID。

4.jps

功能：列出当前所有正在运行的Java进程的进程ID和进程名。

使用方法：在命令行中输入 `jps [options]`，其中options可以指定需要显示的信息。

5.jmap

功能：生成Java进程中的内存映像文件，用于分析内存问题。

使用方法：在命令行中输入 `jmap [options] [pid]`，其中options可以指定需要生成的内存映像文件的格式和内容，pid是Java进程的进程ID。

命令	功能
<code>jstack</code>	生成当前Java进程中所有线程的堆栈跟踪信息，用于诊断线程相关的问题。
<code>jstat</code>	用于监控JVM的各种统计信息，包括堆内存使用情况、GC信息、类加载器信息等。
<code>jinfo</code>	

	显示Java进程的系统属性和JVM参数信息。
jps	列出当前所有正在运行的Java进程的进程ID和进程名。
jmap	生成Java进程中的内存映像文件，用于分析内存问题。