面向对象进阶

接口、内部类

目录

Contents

接口是Java实现**多继承**的一种机制，一个类只能继承一个父类，但如果需要一个类从**多个类**中继承抽象方法，就明显无法实现，所以就出现了**接口**的概念。一个类只可以继承一个父类，但却可以**实现多个接口**。

接口的定义与特点

- 接口的格式如下：

```
接口用关键字interface来定义
public interface 接口名 {
        // 常量
        // 抽象方法
}
```

- JDK8之前接口中只能是**抽象方法**和**常量**。
- **接口不能实例化**。
- 接口中的成员都是public修饰的，写不写都是，因为规范的目的是为了公开化。

接口中成员的特点

接口不是被实例化的，所以**成员变量**不可以是实例变量，他们必须是**类变量**。这就是为什么它们被自动修饰为static。static意味着变量属于整个类而不是类的某个实例，所以不管接口有多少个实现，变量只有一个副本。

- **成员变量**：

  只能是**常量**

  默认修饰符：public static final

- **成员方法**：

  只能是**抽象方法**

  默认修饰符：public abstrat

```
interface MyInterface {
    // 这个变量其实是 public static final int MAX = 10;
    int MAX = 10;
}
```

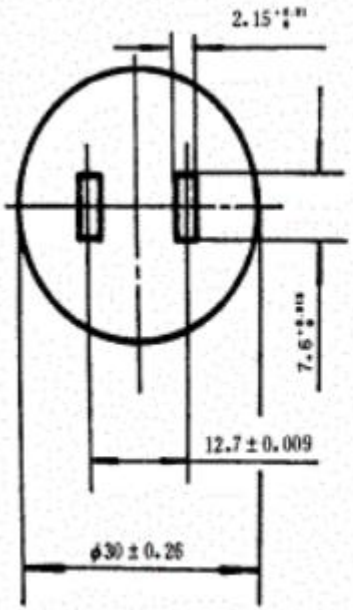
什么是接口

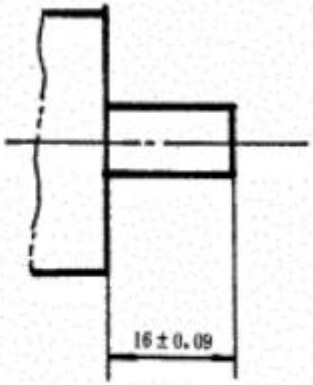
- 接口也是一种规范。

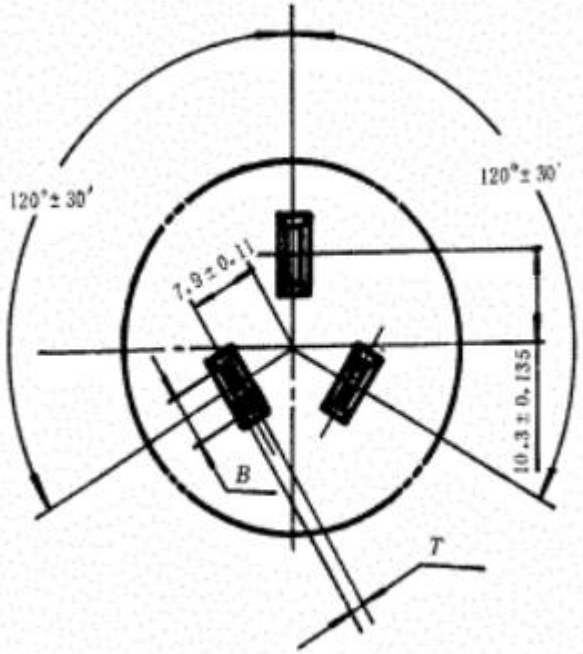
中华人民共和国国家标准

家用和类似用途单相插头插座
型式、基本参数和尺寸　　GB 1002—1996

原因是我们国家的标准委员会已经定义了插头插座的标准，所有的厂商必须严格的按照这种标准去执行。
如果你尝试修改这个规范定义的数据这个是不允许的。







目录

Contents

接口的用法：

- 接口是用来被类实现（implements）的，实现接口的类称为实现类。
- 实现类可以理解成所谓的子类。

```
修饰符 class 实现类 implements 接口1, 接口2, 接口3 , ... {
}
```

实现的关键字：implements

- 从上面可以看出，接口可以被类**单**实现，也可以被类**多**实现。

接口实现的注意事项：

- 一个类实现接口，必须重写完**全部接口的全部抽象方法**，否则这个类需要定义成抽象类。

目录

Contents

练习

- 编写带有**接口**和**抽象类**的标准Javabean类

猫Cat	属性：名字，年龄	行为：吃小鱼，会跑步
狗Dog	属性：名字，年龄	行为：吃骨头，会跑步
鸟Bird	属性：名字，年龄	行为：吃虫子，会飞

提示：

1. **抽象类Animal**：包含共有属性和方法，使用抽象方法表示共有但由子类具体实现的行为。
2. 接口使用：定义**Runnable和Flyable接口**表示可跑步和可飞行的行为。
3. 类实现：让Cat和Dog实现Runnable，Bird实现Flyable。
4. 方法实现：在子类中覆写（Override）Animal中的抽象方法和接口中声明的所有方法。

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

```
public interface Flyable {
    public void fly();
}
```

```
public abstract class Animal {
    private String name;
    private int age;
    public abstract void eat();

    // 无参数构造函数
    public Animal() { }

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // getter和setter方法
.....
}
```

```
public class Dog extends Animal implements Runnable{
    @Override
    public void eat() {
        System.out.println(getName() + "吃骨头");
    }
    @Override
    public void run() {
        System.out.println(getName() + "会跑步");
    }
    public Dog() {
    }
    public Dog(String name, int age) {
        super(name, age);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        // 创建Dog实例并测试方法
        Dog dog = new Dog("小狗", 5);
        dog.eat();
        dog.run();
    }
}
```

目录

Contents

- 接口
  - 接口概述、特点
  - 接口的基本使用：被实现
  - 接口的应用场景：模拟
  - 补充知识：接口与接口的多继承
  - 补充知识：JDK8开始接口新增的方法
  - 补充知识：使用接口的注意事项
- 内部类
- 常用API

基本小结

- 类和类的关系：单继承。
- 类和接口的关系：多实现。
- 接口和接口的关系：多继承，一个接口可以同时继承多个接口。

接口多继承的作用

- 规范合并，整合多个接口为同一个接口，便于子类实现。

```
public interface C extends B , A{

}
```

目录

Contents

- 接口
  - 接口概述、特点
  - 接口的基本使用：被实现
  - 接口的应用场景：模拟
  - 补充知识：接口与接口的多继承
  - 补充知识：JDK8开始接口新增的方法
  - 补充知识：使用接口的注意事项
- 内部类
- 常用API

第一种：默认方法

- 类似之前写的普通实例方法：必须用default修饰
- 默认会public修饰。需要用接口的实现类的对象来调用

```
default void run(){
    System.out.println("--开始跑--");
}
```

第二种：静态方法

- 默认会public修饰，必须static修饰。
- **注意：接口的静态方法必须用本身的接口名来调用。**

```
static void inAddr(){
    System.out.println("学习Java!");
}
```

第三种：私有方法

- 就是私有的实例方法:，必须使用private修饰，从JDK9才开始有的。
- 只能在本类中被其他的默认方法或者私有方法访问。

```
private void go(){
    System.out.println("--准备--");
}
```

目录

Contents

- 接口
  - 接口概述、特点
  - 接口的基本使用：被实现
  - 接口的应用场景：模拟
  - 补充知识：接口与接口的多继承
  - 补充知识：JDK8开始接口新增的方法
  - 补充知识：使用接口的注意事项
- 内部类
- 常用API

接口的注意事项

1、接口不能创建对象

2、一个类实现多个接口，多个接口的规范不能冲突

```
interface Identity {    String getDetails();}

interface Description {    int getDetails();}
// 这个类尝试实现上面的两个接口，但会失败，因为Java不允许一个方法有两个不同的返回类型。
class Person implements Identity, Description {
    @Override
    public String getDetails() {        return "This is a person.";    }
    // 编译错误：方法的返回类型不兼容
    @Override
    public int getDetails() {        return 42;    }
}
```

接口的注意事项

3、一个类继承了父类，同时又实现了接口，父类中和接口中有同名方法，**默认用父类的**。

4、一个类实现了多个接口，多个接口中存在同名的默认方法，可以不冲突，这个类重写该方法即可。

```
interface Identity {    String getDetails();}

interface Description { String getDetails();}

class Person implements Identity, Description {
    @Override
    public String getDetails() {        return "This is a person.";    }
}
```

接口的注意事项

5、一个接口继承多个接口，是没有问题的，如果多个接口中存在规范冲突则不能多继承。

```
interface A {
    void action();
}

interface B {
    int action(); // 注意这里，返回了一个 int 类型结果
}

// 下面这行代码将会导致编译错误，因为接口 A 和 B 有相同的方法签名但返回类型不同
interface C extends A, B {
    // 无法同时继承接口 A 和 B，因为它们存在规范冲突
}
```

目录

Contents

- 接口
- 内部类
  - 内部类概述[了解]
  - 内部类之一：静态内部类[了解]
  - 内部类之二：成员内部类[了解]
  - 内部类之三：局部内部类[了解]
  - 内部类之四：匿名内部类概述[重点]
  - 匿名内部类常见使用形式
  - 匿名内部类真实使用场景演示
- 常用API

内部类

- 内部类就是定义在一个**类里面的类**，里面的类可以理解成（寄生），外部类可以理解成（宿主）。

```
public class People{
    // 内部类
    public class Heart{
    }
}
```

内部类的使用场景

- 场景：当一个事物的内部，还有一个部分需要一个完整的结构进行描述时。

基本作用

- 内部类通常可以方便访问外部类的成员，包括私有的成员。
- 内部类提供了更好的封装性，内部类本身就可以用private ，protectecd等修饰，封装性可以做更多控制。

内部类的分类

- 静态内部类
- 成员内部类（非静态内部类）
- 局部内部类
- 匿名内部类

目录

Contents

- 接口
- 内部类
  - 内部类概述[了解]
  - 内部类之一：静态内部类[了解]
  - 内部类之二：成员内部类[了解]
  - 内部类之三：局部内部类[了解]
  - 内部类之四：匿名内部类概述[重点]
- 常用API

什么是静态内部类？

- 有static修饰，属于外部类本身。
- 它的特点和使用与普通类是完全一样的，类有的成分它都有，只是位置在别人里面而已。

```
public class Outer{
        // 静态成员内部类
        public static class Inner{
        }
}
```

静态内部类创建对象的格式：

格式：外部类名.内部类名 对象名 = new 外部类名.内部类构造器;

范例：Outer.Inner in = new Outer.Inner();

静态内部类的访问拓展：

静态内部类只能访问外部类的**静态变量和静态方法**

如果要访问外部类的**实例成员**需要创建对象

```
public class Car {
    String carName;
    static class Engine{
        String engneName;
        public void show(){
            System.out.println(carName);
            Car c =new Car();
            System.out.println(c.carName);
        }
    }
}
```

静态内部类的访问拓展：

```
public class Car {
    String carName;
    static class Engine{
        public static void staticShow(){
            System.out.println("静态内部类的静态方法");
        }
        public void show(){
            Car c =new Car();
            System.out.println(c.carName);
        }
    }
}
```

调用静态方法的格式：

外部类名.内部类名.方法名();

```
public class Test {
    public static void main(String[] args) {
        Car.Engine.staticShow();
    }
}
```

目录

Contents

- 接口
- 内部类
  - 内部类概述
  - 内部类之一：静态内部类[了解]
  - 内部类之二：成员内部类[了解]
  - 内部类之三：局部内部类[了解]
  - 内部类之四：匿名内部类概述[重点]
- 常用API

什么是成员内部类？

- 无static修饰，写在成员位置，属于外部类的成员。
- JDK16之前，成员内部类中不能定义静态成员，JDK 16开始也可以定义静态成员了。

```
public class Outer {
    // 成员内部类
    public class Inner {

    }
}
```

成员内部类创建对象的格式：

格式：外部类名.内部类名 对象名 = new 外部类构造器.new 内部类构造器();

范例：Outer.Inner in = new Outer().new Inner();

成员内部类的访问拓展：

1、成员内部类中是否可以直接访问外部类的静态成员？

- **可以，外部类的静态成员只有一份可以被共享访问。**

2、成员内部类的实例方法中是否可以直接访问外部类的实例成员？

- **可以的，因为必须先有外部类对象，才能有成员内部类对象，所以可以直接访问外部类对象的实例成员。**

```
public class Car {
    String carName;
    class Engine{
        public void show(){
            System.out.println(carName);
        }
    }
}
```

1. 成员内部类是什么样的、有什么特点？

- 无static修饰，属于外部类的对象。
- 可以直接访问外部类的静态成员，实例方法中可以直接访问外部类的实例成员。

2. 成员内部类如何创建对象？

- 外部类名.内部类名 对象名 = new 外部类构造器.new 内部类构造器();

练习 • 成员内部类

- 请观察如下代码，写出合适的代码对应其注释要求输出的结果。

```
class People{
    private int heartbeat = 150;
    public class Heart{
        private int heartbeat = 110;
        public void show(){
            int heartbeat = 78;
System.out.println(heartbeat); // 78
System.out.println(this.heartbeat); // 110
System.out.println(People.this.heartbeat); // 150
        }
    }
}
```

注意：在成员内部类中访问所在外部类对象，格式：外部类名.this。

目录

Contents

- 接口
- 内部类
  - 内部类概述
  - 内部类之一：静态内部类[了解]
  - 内部类之二：成员内部类[了解]
  - 内部类之三：局部内部类[了解]
  - 内部类之四：匿名内部类概述[重点]
- 常用API

局部内部类

- 局部内部类放在方法、代码块、构造器等执行体中。

目录

Contents

- 接口
- 内部类
  - 内部类概述
  - 内部类之一：静态内部类[了解]
  - 内部类之二：成员内部类[了解]
  - 内部类之三：局部内部类[了解]
  - 内部类之四：匿名内部类概述[重点]
- 常用API

匿名内部类：

- 本质上是一个隐藏了名字的内部类。
- **作用**：方便创建子类对象，最终目的是为了简化代码编写。

格式：

```
new 类名或者接口名() {
    重写方法;
};
```

继承/实现

方法重写

创建对象

```
Employee a = new Employee() {
    public void work() {
    }
};
a. work();
```

特点总结：

- 匿名内部类是一个**没有名字**的内部类，同时也代表一个对象。
- 匿名内部类产生的对象类型，相当于是当前new的那个的类型的**子类类型**。

```
public abstract class Animal {
    public abstract void run();
}
```

```
public class Dog extends Animal{
    @Override
    public void run(){
        System.out.println("狗跑步");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Animal d= new Dog();//多态
        d.run();//狗跑步
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Animal d = new Animal() {
            @Override
            public void run() {
                System.out.println("狗跑步");
            }
        };
        d.run();
    }
}
```

匿名内部类

- 但是，如果此处的Dog类只使用一次，那么将其编写为独立的一个类岂不是很麻烦？
- 这个时候就引入了匿名内部类

练习 使用匿名内部类改写

```
// 定义Task接口
interface Task {
    void execute();
}

// 实现Task接口的类
class PrintTask implements Task {
    @Override
    public void execute() {
        System.out.println("任务执行了！ ");
    }
}

public class Test {
    public static void main(String[] args) {
        Task task = new PrintTask();

        task.execute();
    }
}
```

```
// 定义Task接口保持不变
interface Task {
    void execute();
}

public class Test {
    public static void main(String[] args) {
        // 使用匿名类创建Task对象
        Task task = new Task() {
            @Override
            public void execute() {
                System.out.println("任务执行了！ ");
            }
        };
        task.execute();
    }
}
```

目录

Contents

- 接口
- 内部类
- 常用API
  - API概述
  - Object类：toString方法
  - Object类：equals方法

什么是API?

- API(Application Programming interface) 应用程序编程接口。
- **简单来说：就是Java帮我们已经写好的一些方法，我们直接拿过来用就可以了。**

目录

Contents

- 接口
- 内部类
- 常用API
  - API概述
  - Object类：toString方法
  - Object类：equals方法

Object类的作用：

- 一个类要么默认继承了Object类，要么间接继承了Object类，Object类是Java中的**祖宗类**。
- Object作为所有类的父类，提供了很多**常用的方法**给每个子类对象拿来使用。

Object类的常用方法：

方法名	说明
public String **toString()**	默认是返回当前对象在堆内存中的地址信息:类的全限名@内存地址
public boolean equals(Object o)	默认是比较当前对象与另一个对象的地址是否相同，相同返回true，不同返回false

Object的toString方法：

方法名	说明
public String toString()	默认是返回当前对象在堆内存中的地址信息:类的全限名@内存地址

```
package api_object;

class Student {
    String name;
}
```

```
package api_object;
public class Test{
    public static void main(String[] args) {
        Student s=new Student();
        s.name = "xiaoming";
        System.out.println(s.toString());;
    }
}
```

```
"C:\Program Files\Java\jdk-21
api_object.Student@2f4d3709
```

Object的toString方法：

方法名	说明
public String toString()	默认是返回当前对象在堆内存中的地址信息:类的全限名@内存地址

```
class Student {
    String name;

    @Override
    public String toString() {
        return "名字："+name;
    }
}
```

```
"C:\Program Files\J
名字：xiaoming
```

Object的toString方法：

方法名	说明
public String toString()	默认是返回当前对象在堆内存中的地址信息:类的全限名@内存地址

问题引出

- 开发中直接输出对象，默认输出**对象的地址**其实是毫无意义的。
- 开发中输出对象变量，更多的时候是希望看到**对象的内容数据**而不是对象的地址信息。

toString存在的意义

- 父类toString()方法存在的意义就是**为了被子类重写**，以便返回**对象的内容信息**，而不是地址信息！！

1．Object的toString方法的基本作用是什么，存在的意义是什么？

- **基本作用：给子类继承，子类对象调用可以返回自己的地址。**
- **意义：让子类重写，以便返回子类对象的内容。**

目录

Contents

- 接口
- 内部类
- 常用API
  - API概述
  - Object类：toString方法
  - Object类：equals方法

Object的equals方法：

方法名	说明
public boolean equals(Object o)	默认是比较当前对象与另一个对象的地址是否相同，相同返回true，不同返回false

问题思考

- 直接比较两个对象的地址是否相同完全可以用“==”替代equals。
- 同时，开发中很多业务情况下，更想判断2个对象的内容是否一样。

equals存在的意义

- 为了被子类重写，以便子类自己来定制比较规则（比如比较对象内容）。

1．Object的equals方法的基本作用，存在的意义是什么？

- **基本作用：默认是与另一个对象比较地址是否一样**
- **存在的意义：让子类重写，以便比较对象的内容是否相同。**

包装类

- 包装类就是把**基本类型的数据**包装成**对象**。

基本数据类型	对应的包装类（引用数据类型）
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean

自动装箱：基本数据类型可以自动转换为包装类型。

```
Integer i = 10；//自动装箱。编译器添加：
Integer i = Integer.valueOf(10);
```

编译器会自动通过调用Integer类的**valueOf**方法，将int值转换为Integer对象，完成了自动装箱。

自动拆箱：包装类型可以自动转换为基本数据类型。

```
int a = i;  // 这里实际上做的是
int a = i.intValue();
```

编译器会自动通过调用Integer类的**intValue**方法将Integer对象转换为int值，完成了自动拆箱。

方法名	说明
Integer.valueOf()	默认情况下，Java会缓存**-128到127**之间的整数值，valueOf()会返回缓存的对象而不是创建新对象 这个范围之外的值会创建新的Integer对象

```
Integer i1 = 10;
Integer i2 = 10;
System.out.println(i1 == i2);       // true，因为都指向缓存池中的同一个对象
System.out.println(i1.equals(i2));  // true，比较值

Integer i3 = 200;
Integer i4 = 200;
System.out.println(i3 == i2);       // false，因为超出缓存范围，是两个不同对象
System.out.println(i3.equals(i4));  // true，比较值
```

将基本类型的数据转换为字符串类型的三种方法：

1. 使用`String.valueOf()`方法。这是一种非常通用的方法，可以用于所有基本数据类型。

```
int i = 100;
String str = String.valueOf(i);
```

2. 使用`Integer.toString()`方法。这种方法在将int类型转为string类型时常用。

```
int i = 100;
String str = Integer.toString(i);
```

3. 使用`+`运算符。在java中，可以直接使用`+`运算符将**数字变量**与**字符串**进行相连，从而达到转换效果。

```
int i = 100;
String str = "" + i;
```

将字符串类型的数值转换为基本类型的数据：

1. 使用包装类的parse方法：各种基本数据类型的包装类，比如Integer、Double等，都有parse方法。

```
String str = "123";
int i = Integer.parseInt(str);
```

2. 使用包装类的valueOf方法：将字符串转换为基本类型，但返回的是基本类型的包装类，因此需要进行自动拆箱。

```
String str = "123";
int i = Integer.valueOf(str);
```

实际类型转换过程是：字符串"123"转换为Integer对象，然后通过自动拆箱获得int数值。

String

当创建一个字符串对象，如 `String str = "123";`这种情况，Java会先检查**字符串池**(String Pool)中是否已经存在值为"123"的String对象。

字符串池(string pool)是Java**堆内存**的一部分，主要用于减少在运行时创建新的字符串对象的数量。它维护着一个由**字符串组成**的池，字符串池中的字符串可以被所有的线程共享。

String

具体步骤如下:

- 1. 首先，JVM会在堆内存的字符串池中查找是否有"123"这个值的字符串对象。
- 2. 如果找到了，那么str只是一个指向该对象的引用，**不会重新创建新的对象**。
- 3. 如果在字符串池中没有找到相应的值，那么JVM就会在**字符串池中创建一个值为"123"的字符串对象**，并让str指向新创建的这个对象。

如果我们使用**new关键字创建字符串对象**，如 `String str = new String("123")`，那么就会在堆内存中新建一个对象，不论**字符串池**中是否存在相同值的对象。

StringBuilder

- StringBuilder代表**可变**字符串对象，相当于是一个容器，它里面装的字符串是可以改变的，就是用来操作字符串的。
- 好处：StringBuilder比String更适合做字符串的修改操作，效率会更高，代码也会更简洁。

构造器	说明
public StringBuilder()	创建一个空白的可变的字符串对象，不包含任何内容
public StringBuilder(String str)	创建一个指定字符串内容的可变字符串对象

方法名称	说明
public StringBuilder append(任意类型)	添加数据并返回StringBuilder对象本身
public StringBuilder reverse()	将对象的内容反转
public int length()	返回对象内容长度
public String toString()	通过toString()就可以实现把StringBuilder转换为String

为什么操作字符串建议使用StringBuilder，而不用String?

```java
public class Test {
    public static void main(String[] args) {
        String s = "";
        for (int i = 0; i < 1000000; i++) {
            s = s + "abc";
        }
        System.out.println(s);
    }
}
```


```java
public class Test {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < 1000000; i++) {
            sb.append("abc");
        }
        System.out.println(sb);
    }
}
```

在Java中，String是不可变的，也就是说，一旦创建了一个String对象，你就**不能改变它的内容**。每次对String类型的值进行修改，实际上都是在**内存中创建了一个新的String对象**。这样在大量拼接或修改字符串时，其实会产生很多临时的无用对象，这将会消耗大量的内存，影响程序性能。

而StringBuilder在Java中是一个**可变的对象**。使用StringBuilder进行字符串的拼接操作，会**在原有对象的基础上进行修改**，不会像**String那样频繁创建和销毁对象**，从而降低了内存开销，提高了效率。

为什么操作字符串建议使用StringBuilder，而不用String?

```
public class Test {
    public static void main(String[] args) {
        String s = "";
        for (int i = 0; i < 1000000; i++) {
            s = s + "abc";
        }
        System.out.println(s);
    }
}
```


```
public class Test {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < 1000000; i++) {
                sb.append("abc");
        }
        System.out.println(sb);
    }
}
```

- 对于字符串相关的操作，如频繁的拼接、修改等，建议用StringBuidler，效率更高!
- 注意：如果操作字符串较少，或者不需要操作，以及定义字符串变量，还是建议用String。

- StringBuffer与StringBuilder

注意：

- StringBuffer的用法与StringBuilder功能相同
- 但 StringBuilder是线程不安全的
- StringBuffer是线程安全的，可以被多线程安全地使用

补充内容

- String

方法名	说明
charAt(int index)	返回指定索引处的 char 值。
length()	返回此字符串的长度。

```
public class Main {
    public static void main(String[] args) {
        String str="Hello, Java!";
        // 遍历字符串并打印索引和对应字符
        for (int i=0; i < str.length(); i++) {
        System.out.println("索引 " + i + " 的字符是: " + str.charAt(i));
        }
    }
}
```