

- [final,static,this,super 关键字总结](#)
  - [final 关键字](#)
  - [static 关键字](#)
  - [this 关键字](#)
  - [super 关键字](#)
  - [参考](#)
- [static 关键字详解](#)
  - [static 关键字主要有以下四种使用场景](#)
    - [修饰成员变量和成员方法\(常用\)](#)
    - [静态代码块](#)
    - [静态内部类](#)
    - [静态导包](#)
  - [补充内容](#)
    - [静态方法与非静态方法](#)
    - [static{}静态代码块与{}非静态代码块\(构造代码块\)](#)
    - [参考](#)

## final,static,this,super 关键字总结

---

### final 关键字

**final关键字**，意思是最终的、不可修改的，最见不得变化，用来修饰类、方法和变量，具有以下特点：

1. **final**修饰的类不能被继承，**final**类中的所有成员方法都会被隐式的指定为**final**方法；
2. **final**修饰的方法不能被重写；
3. **final**修饰的变量是常量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能让其指向另一个对象。

说明：使用final方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的Java实现版本中，会将final方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的Java版本已经不需要使用final方法进行这些优化了）。类中所有的private方法都隐式地指定为final。

### static 关键字

**static 关键字主要有以下四种使用场景：**

1. **修饰成员变量和成员方法**：被 **static** 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用。被**static** 声明的成员变量属于静态成员变量，静态变量 存放在 Java 内存区域的方法区。调用格式：**类名.静态变量名 类名.静态方法名()**
2. **静态代码块**：静态代码块定义在类中方法外，静态代码块在非静态代码块之前执行(静态代码块—>非静态代码块—>构造方法)。该类不管创建多少对象，静态代码块只执行一次。
3. **静态内部类（static修饰类的话只能修饰内部类）**：静态内部类与非静态内部类之间存在一个最大的区别：非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内

部类却没有。没有这个引用就意味着：1. 它的创建是不需要依赖外围类的创建。2. 它不能使用任何外围类的非static成员变量和方法。

4. **静态导包(用来导入类中的静态资源, 1.5之后的新特性)**: 格式为: `import static` 这两个关键字连用可以指定导入某个类中的指定静态资源, 并且不需要使用类名调用类中静态成员, 可以直接使用类中静态成员变量和成员方法。

## this 关键字

this关键字用于引用类的当前实例。 例如:

```
class Manager {
    Employees[] employees;

    void manageEmployees() {
        int totalEmp = this.employees.length;
        System.out.println("Total employees: " + totalEmp);
        this.report();
    }

    void report() { }
}
```

在上面的示例中, this关键字用于两个地方:

- this.employees.length: 访问类Manager的当前实例的变量。
- this.report () : 调用类Manager的当前实例的方法。

此关键字是可选的, 这意味着如果上面的示例在不使用此关键字的情况下表现相同。 但是, 使用此关键字可能会使代码更易读或易懂。

## super 关键字

super关键字用于从子类访问父类的变量和方法。 例如:

```
public class Super {
    protected int number;

    protected showNumber() {
        System.out.println("number = " + number);
    }
}

public class Sub extends Super {
    void bar() {
        super.number = 10;
        super.showNumber();
    }
}
```

在上面的例子中，Sub 类访问父类成员变量 number 并调用其父类 Super 的 `showNumber ()` 方法。

### 使用 this 和 super 要注意的问题：

- 在构造器中使用 `super()` 调用父类中的其他构造方法时，该语句必须处于构造器的首行，否则编译器会报错。另外，`this` 调用本类中的其他构造方法时，也要放在首行。
- `this`、`super`不能用在static方法中。

### 简单解释一下：

被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享。而 `this` 代表对本类对象的引用，指向本类对象；而 `super` 代表对父类对象的引用，指向父类对象；所以，**this和super是属于对象范畴的东西，而静态方法是属于类范畴的东西。**

## 参考

- <https://www.codejava.net/java-core/the-java-language/java-keywords>
- <https://blog.csdn.net/u013393958/article/details/79881037>

## static 关键字详解

---

### static 关键字主要有以下四种使用场景

- 修饰成员变量和成员方法
- 静态代码块
- 修饰类(只能修饰内部类)
- 静态导包(用来导入类中的静态资源，1.5之后的新特性)

### 修饰成员变量和成员方法(常用)

被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用。被static 声明的成员变量属于静态成员变量，静态变量 存放在 Java 内存区域的方法区。

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已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），目的应该是与 Java 堆区分开来。

HotSpot 虚拟机中方法区也常被称为“永久代”，本质上两者并不等价。仅仅是因为 HotSpot 虚拟机设计团队用永久代来实现方法区而已，这样 HotSpot 虚拟机的垃圾收集器就可以像管理 Java 堆一样管理这部分内存了。但是这并不是一个好主意，因为这样更容易遇到内存溢出问题。

调用格式：

- 类名.静态变量名
- 类名.静态方法名()

如果变量或者方法被 `private` 则代表该属性或者该方法只能在类的内部被访问而不能在类的外部被访问。

测试方法：

```
public class StaticBean {

    String name;
    //静态变量
    static int age;

    public StaticBean(String name) {
        this.name = name;
    }
    //静态方法
    static void sayHello() {
        System.out.println("Hello i am java");
    }
    @Override
    public String toString() {
        return "StaticBean{" +
            "name=" + name + ",age=" + age +
            "}";
    }
}
```

```
public class StaticDemo {

    public static void main(String[] args) {
        StaticBean staticBean = new StaticBean("1");
        StaticBean staticBean2 = new StaticBean("2");
        StaticBean staticBean3 = new StaticBean("3");
        StaticBean staticBean4 = new StaticBean("4");
        StaticBean.age = 33;
        System.out.println(staticBean + " " + staticBean2 + " " + staticBean3 + "
" + staticBean4);
        //StaticBean{name=1,age=33} StaticBean{name=2,age=33}
        StaticBean{name=3,age=33} StaticBean{name=4,age=33}
        StaticBean.sayHello();//Hello i am java
    }

}
```

## 静态代码块

静态代码块定义在类中方法外, 静态代码块在非静态代码块之前执行(静态代码块 —> 非静态代码块 —> 构造方法)。该类不管创建多少对象, 静态代码块只执行一次。

静态代码块的格式是

```
static {
    语句体;
}
```

一个类中的静态代码块可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果静态代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。



静态代码块对于定义在它之后的静态变量，可以赋值，但是不能访问。

## 静态内部类

静态内部类与非静态内部类之间存在一个最大的区别，我们知道非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：

1. 它的创建是不需要依赖外围类的创建。
2. 它不能使用任何外围类的非static成员变量和方法。

Example（静态内部类实现单例模式）

```
public class Singleton {

    //声明为 private 避免调用默认构造方法创建对象
    private Singleton() {
    }

    // 声明为 private 表明静态内部该类只能在该 Singleton 类中被访问
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getUniqueInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

当 Singleton 类加载时，静态内部类 SingletonHolder 没有被加载进内存。只有当调用 `getUniqueInstance()` 方法从而触发 `SingletonHolder.INSTANCE` 时 SingletonHolder 才会被加载，此时初始化 INSTANCE 实例，并且 JVM 能确保 INSTANCE 只被实例化一次。

这种方式不仅具有延迟初始化的好处，而且由 JVM 提供了对线程安全的支持。

## 静态导包

格式为：import static

这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法

```
//将Math中的所有静态资源导入，这时候可以直接使用里面的静态方法，而不用通过类名进行调用
//如果只想导入单一某个静态方法，只需要将*换成对应的方法名即可

import static java.lang.Math.*; //换成import static java.lang.Math.max;具有一样的效果

public class Demo {
    public static void main(String[] args) {

        int max = max(1,2);
        System.out.println(max);
    }
}
```

## 补充内容

### 静态方法与非静态方法

静态方法属于类本身，非静态方法属于从该类生成的每个对象。如果您的方法执行的操作不依赖于其类的各个变量和方法，请将其设置为静态（这将使程序的占用空间更小）。否则，它应该是非静态的。

Example

```
class Foo {
    int i;
    public Foo(int i) {
        this.i = i;
    }

    public static String method1() {
        return "An example string that doesn't depend on i (an instance variable)";
    }

    public int method2() {
        return this.i + 1; //Depends on i
    }
}
```

```
}
```

你可以像这样调用静态方法：`Foo.method1()`。如果您尝试使用这种方法调用 `method2` 将失败。但这样可行

```
Foo bar = new Foo(1);
bar.method2();
```


总结：

- 在外部调用静态方法时，可以使用“类名.方法名”的方式，也可以使用“对象名.方法名”的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
- 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制

### `static{} 静态代码块与 {} 非静态代码块(构造代码块)`

相同点：都是在JVM加载类时且在构造方法执行之前执行，在类中都可以定义多个，定义多个时按定义的顺序执行，一般在代码块中对一些static变量进行赋值。

不同点：静态代码块在非静态代码块之前执行(静态代码块 -> 非静态代码块 -> 构造方法)。静态代码块只在第一次new执行一次，之后不再执行，而非静态代码块在每new一次就执行一次。非静态代码块可在普通方法中定义(不过作用不大)；而静态代码块不行。

 **修正 (参见: [issue #677](#))**：静态代码块可能在第一次new的时候执行，但不一定只在第一次new的时候执行。比如通过 `Class.forName("ClassDemo")` 创建 Class 对象的时候也会执行。

一般情况下,如果有些代码比如一些项目最常用的变量或对象必须在项目启动的时候就执行的时候,需要使用静态代码块,这种代码是主动执行的。如果我们想要设计不需要创建对象就可以调用类中的方法,例如: Arrays类, Character类, String类等,就需要使用静态方法,两者的区别是 静态代码块是自动执行的而静态方法是被调用的时候才执行的。

Example:

```
public class Test {
    public Test() {
        System.out.print("默认构造方法! --");
    }

    //非静态代码块
    {
        System.out.print("非静态代码块! --");
    }

    //静态代码块
    static {
        System.out.print("静态代码块! --");
    }
}
```

```
private static void test() {
    System.out.print("静态方法中的内容! --");
    {
        System.out.print("静态方法中的代码块! --");
    }
}

public static void main(String[] args) {
    Test test = new Test();
    Test.test();//静态代码块! --静态方法中的内容! --静态方法中的代码块! --
}
}
```

上述代码输出:

静态代码块! --非静态代码块! --默认构造方法! --静态方法中的内容! --静态方法中的代码块! --

当只执行 `Test.test();` 时输出:

静态代码块! --静态方法中的内容! --静态方法中的代码块! --

当只执行 `Test test = new Test();` 时输出:

静态代码块! --非静态代码块! --默认构造方法! --

非静态代码块与构造函数的区别是: 非静态代码块是给所有对象进行统一初始化, 而构造函数是给对应的对象初始化, 因为构造函数是可以多个的, 运行哪个构造函数就会建立什么样的对象, 但无论建立哪个对象, 都会先执行相同的构造代码块。也就是说, 构造代码块中定义的是不同对象共性的初始化内容。

## 参考

- <https://blog.csdn.net/chen13579867831/article/details/78995480>
- <https://www.cnblogs.com/chenssy/p/3388487.html>
- <https://www.cnblogs.com/Qian123/p/5713440.html>