JAVA基础加强

2018年8月29日 9:52

1. java基础加强概述

在JDK5中，JDK中加入了大量的新特性，这些新特性大大的提升了JAVA语言的能力，使java可以开发更加复杂的代码，我们需要学习并掌握这些新的特性，在后续开发一些复杂java代码时，都会用得到。

增强for循环 自动封箱拆箱 静态导入 可变参数 枚举 泛型 注解

**单例设计模式

增强for循环

2018年8月29日 10:15

for的两种用法

普通for实现循环执行代码

增强for实现遍历实现了Iterable接口的类

案例：

```
package cn.tedu.newfeature;

import java.util.ArrayList;
import java.util.List;

/**
 * 增强for循环
 */
public class Demo01 {
	public static void main(String[] args) {
		//1.普通for - 实现循环执行代码
		for(int i=0;i<5;i++){
			System.out.println("hello world~");
		}

		//2.增强for - 遍历集合
		List<String> list = new ArrayList<>();
		list.add("aaa");
		list.add("bbb");
		list.add("ccc");
		list.add("ddd");
		list.add("eee");
		for(String str : list){
			System.out.println(str);
		}
	}
```

```
}
```

自动封箱拆箱

2018年8月29日 10:19

java中的八种基本数据类型 都有对应的包装类

byte short int long float double char boolean

Byte Short Integer Long Float Double Character Boolean

在JDK5之前，我们需要自己手动的来进行基本数据类型和 对应的 包装数据类型之间的类型转换

在JDK5之后，编译会自动帮我们进行这样的转换，所以，在编写代码时基本数据类型和其包装类 可以混用 会自动进行封箱拆箱的操作

案例：

```
package cn.tedu.newfeature;

/**
 * 自动封箱拆箱
 */
public class Demo02 {
    public static void main(String[] args) {
        //--普通转换
        Integer ix1 = new Integer(1);
        int ix2 = ix1.intValue();

        //--自动封箱拆箱转换
        Integer ix3 = 1;
        int ix4 = new Integer(2);
    }
}
```

静态导入

2018年8月29日 10:25

在使用类的静态成员时，需要类名来调用，而通过静态导入，可以省略类名直接使用静态成员，但是这种方式会造成程序的可阅读性降低，不推荐使用

案例：

```
package cn.tedu.newfeature;

import static java.lang.Math.floor;
import static java.lang.Math.ceil;

/**
 * 静态导入
 */
public class Demo03 {
	public static void main(String[] args) {
		//double d1 = Math.floor(3.1415926);
		double d1 = floor(3.1415926);
		System.out.println(d1);
		//double d2 = Math.ceil(3.1415926);
		double d2 = ceil(3.1415926);
		System.out.println(d2);
	}
}
```

可变参数

2018年8月29日 10:31

可变参数是JDK5中提供的一套新的机制，可以使方法接受动态数量的参数，大大提高了java方法的灵活性

语法结构：

public void sumx(int ... nums){}

一个方法中只能有一个可变参数，且只能出现在参数列表的最后一个位置
可变参数在方法的内部相当于一个数组，可以按照访问数组的方式进行访问
如果直接传入一个数组，这个数组会被拆开得到数组内部的元素，当做可变参数的值传递

案例：

```
package cn.tedu.newfeature;

/**
 * 可变参数
 */
public class Demo04 {
    //--可变参数的声明
    public static int sumx1(int ...nums) {
        int sum = 0;
        for(int i = 0;i<nums.length;i++){
            sum += nums[i];
        }
        return sum;
    }
    //--同一个方法内可变参数只能出现一次
    //--且只能出现在 方法参数列表的最后一个位置
    public static int sumx2(int x,int ... nums){
        int sum = 0;
        for(int i = 0;i<nums.length;i++){
            sum += nums[i];
        }
        return sum * x;
    }

    public static void main(String[] args) {

        //--可变参数可以接受0到n个参数 在其内部相当于是一个数组
        int r2 = sumx1();
        int r3 = sumx1(2);
```

```
        int r4 = sumx1(2,3);
        int r5 = sumx1(2,3,4,5,76,7,3,54,567,34,546);
        System.out.println(r5);

        //--可变参数也可以直接接受一个数组 这个过程中会将数组拆分 得到元素作为可变参数的值进行传递
        int [] arr = {2,3,4,5};
        int r6 = sumx1(arr);
        System.out.println(r6);
    }
}
```

枚举

2018年8月29日 10:47

在jdk5之前，是通过抽象类 私有化构造方法 并在类的内部提供公有 静态 常量的 当前类对象来实现枚举的
而在jdk5中，java提供了新的关键字叫enum来实现枚举类型，大大简化了这个过程

案例：

```
package cn.tedu.newfeature;

/**
 * 枚举
 */
public class Demo05 {
	public static void main(String[] args) {
	}
}

enum WeekEnum05{
	Mon,Tue,Wed,Thu,Fri,Sat,Sun
}

class Week05{
	private Week05() {
	}
	public static final Week05 Mon = new Week05();
	public static final Week05 Tue = new Week05();
	public static final Week05 Wed = new Week05();
	public static final Week05 Thu = new Week05();
	public static final Week05 Fri = new Week05();
	public static final Week05 Sat = new Week05();
	public static final Week05 Sun = new Week05();
}
```

在枚举中 可以包含 普通属性 私有的构造方法 普通方法 抽象方法

```
package cn.tedu.newfeature;

/**
 * 枚举
 */
```

```
public class Demo06 {
	public static void main(String[] args) {
		System.out.println(Week06.Fri.getDay());
		System.out.println(Week06.Fri.isRestDay());
		System.out.println(WeekEnum06.Fri.getDay());
		System.out.println(WeekEnum06.Fri.isRestDay());
	}
}

//在JDK5之后，有了enmu关键字 ，便于实现枚举
//--包含成员属性
//--包含私有构造方法
//--包含普通方法
//--包含抽象方法
enum WeekEnum06{
	Mon(1){
		@Override
		public boolean isRestDay() {
			return false;
		}
	},Tue(2){
		@Override
		public boolean isRestDay() {
			return false;
		}
	},Wed(3){
		@Override
		public boolean isRestDay() {
			return false;
		}
	},Thu(4){
		@Override
		public boolean isRestDay() {
			return false;
		}
	},Fri(5){
		@Override
		public boolean isRestDay() {
			return false;
		}
	},Sat(6){
		@Override
		public boolean isRestDay() {
			return true;
```

```
        }
    },Sun(7){
        @Override
        public boolean isRestDay() {
            return true;
        }
    };
    private int day = 0;
    private WeekEnum06(int day){
        this.day = day;
    }
    public int getDay(){
        return day;
    }
    public abstract boolean isRestDay();
}

//在JDK5之前，手动实现枚举效果
//--包含成员属性
//--包含私有构造方法
//--包含普通方法
//--包含抽象方法
abstract class Week06{
    public static final Week06 Mon = new Week06(1){
        @Override
        public boolean isRestDay() {
            return false;
        }
    };
    public static final Week06 Tue = new Week06(2){
        @Override
        public boolean isRestDay() {
            return false;
        }
    };
    public static final Week06 Wed = new Week06(3){
        @Override
        public boolean isRestDay() {
            return false;
        }
    };
    public static final Week06 Thu = new Week06(4){
        @Override
        public boolean isRestDay() {
```

```
            return false;
        }
    };
    public static final Week06 Fri = new Week06(5){
        @Override
        public boolean isRestDay() {
            return false;
        }
    };
    public static final Week06 Sat = new Week06(6){
        @Override
        public boolean isRestDay() {
            return true;
        }
    };
    public static final Week06 Sun = new Week06(7){
        @Override
        public boolean isRestDay() {
            return true;
        }
    };

    private int day = 0;
    private Week06(int day) {
        this.day = day;
    }
    public int getDay(){
        return this.day;
    }
    public abstract boolean isRestDay();
}
```

泛型

2018年8月29日 11:20

1. 泛型的概念

Jdk5中提供的新的特性，来代表不确定的类型

2. 集合泛型

泛型最常见的使用场景，在集合类上限定集合类中能够存储的数据的类型，这种泛型的使用场景称之为集合泛型，但是其实集合泛型也就是一种类上定义的泛型，没有什么特别的

3. 泛型的使用

要么两边都没有，要么一边有一边没有，要么两边都有，如果两边都有必须一模一样，泛型中没有继承的关系

```
List list1 = new ArrayList();
List<String> list2 = new ArrayList();
List list3 = new ArrayList<String>();
List<String> list4 = new ArrayList<String>();
List<Object> list5 = new ArrayList<String>();//报错
List<String> list6 = new ArrayList<Object>();//报错
```

4. 自定义泛型

自定义泛型 分为 方法上的泛型 和 类上的泛型

a. 方法上的泛型

可以在方法上定义泛型，作用范围为当前方法，泛型需要先声明再使用，方法上的泛型可以在方法的返回值前通过尖括号声明，通常声明为一个大写字母，也可以同时声明多个泛型，在中间用逗号分隔即可。

方法上的泛型不需要明确指定具体类型，而是会在使用该方法时，自动推断出泛型的具体类型。

案例：上帝案例

```
package cn.tedu.newfeature;

class Person{}
class Dog{}
class Cat{}

class God{
    public <T> T kill(T t){
        System.out.println("上帝弄死了"+t);
        return t;
    }

}

public class Demo09 {
    public static void main(String[] args) {
        God god = new God();
        Person p = god.kill(new Person());
    }
}
```

a. 类上的泛型

可以在类上定义泛型，作用范围为当前类的内部，泛型需要先声明再使用，类上的泛型可以在类名后通过尖括号声明，通常声明为一个大写字母，也可以同时声明多个泛型，在中间用逗号分隔即可。

类上的泛型需要在创建引用或对象时，明确指定具体类型，如果不指定则默认取值为Object.

静态方法中无法使用类上定义的泛型,如果需要在静态方法中使用泛型，只能在静态方法上声明，在静态方法内使用

**集合泛型其实没有任何特殊的地方，本质上就是一个类上定义的泛型

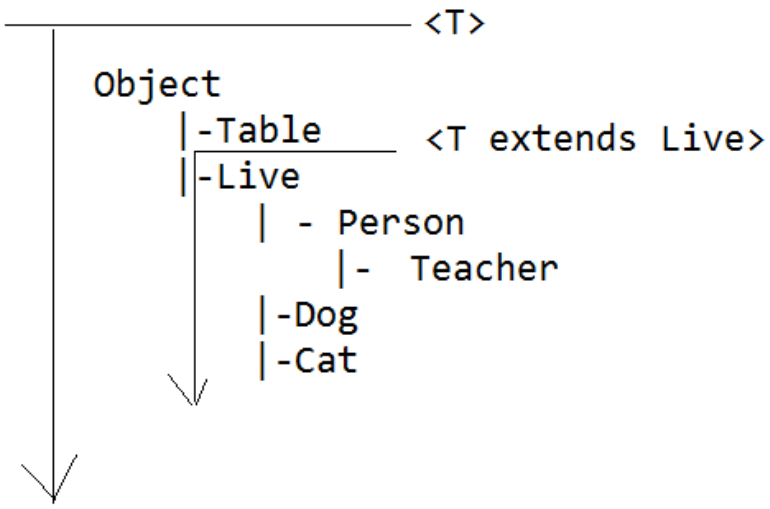
案例：

```
package cn.tedu.newfeature;
```

```
import java.util.ArrayList;
import java.util.List;

class Person{}
class Dog{}
class Cat{}

class God<T>{
	public static <T> void xiaban(T t){

	}

	public T kill(T t){
		System.out.println("上帝弄死了"+t);
		return t;
	}

	public T save(T t){
		System.out.println("上帝弄活了"+t);
		return t;
	}
}

public class Demo09 {
	public static void main(String[] args) {
		God<Person> god = new God<Person>();
		Person p = god.kill(new Person());
		p = god.save(p);
		//god.save(new Dog());
	}
}
```

b. 泛型的上边界

可以在声明泛型时通过extends关键字声明泛型的上边界，从此泛型的取值就只能是指定的上边界类型或其子孙类型。如果不指定上边界，则泛型的默认上边界是Object。



案例：

```
package cn.tedu.newfeature;

class Live{}
class Person extends Live{}
class Teacher extends Person{}
class Dog extends Live{}
class Cat extends Live{}
class Table{}

class God<T extends Live>{
    public static <T> void xiaban(T t){

    }

    public T kill(T t){
        System.out.println("上帝弄死了"+t);
        return t;
    }

    public T save(T t){
        System.out.println("上帝弄活了"+t);
        return t;
    }
}

public class Demo09 {
    public static void main(String[] args) {
        God<Person> personGod = new God<Person>();
        Person p = personGod.kill(new Person());
```

```
        p = personGod.save(p);

        God<Dog> dogGod = new God<Dog>();
        Dog dog = dogGod.kill(new Dog());
        dogGod.save(dog);

        //God<Table> tableGod = new God<>();
        //Table tab = tableGod.kill(new Table());
        //tab = tableGod.save(tab);
    }
}
```

c. 泛型通配符

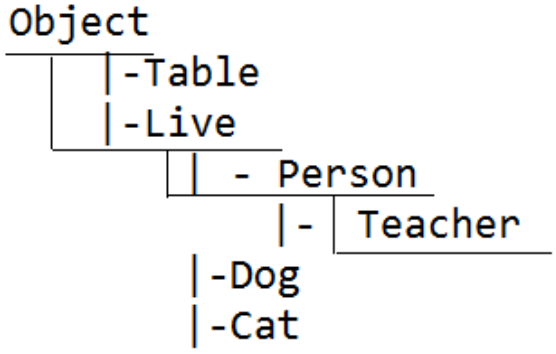
泛型在使用的过程中是没有继承关系的，所以在一个带有泛型的引用需要先后指向不同泛型具体值的对象时，应该声明怎样的泛型类型呢？此时些什么都不行，就可以写一个问号表示未确定，这个？就称之为泛型通配符

```
List <?> list = null;
list = new ArrayList<String>();
list = new ArrayList<Integer>();
```

泛型通配符 也可以通过上边界 或 下边界 限定取值范围

d. 泛型通配符的下边界

在使用泛型通配符时，可以通过super关键字限定泛型统配符的取值范围 为指定类型获取祖先类型，这种用法就称之为指定了泛型通配符的下边界



案例：

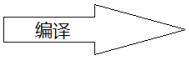
```
package cn.tedu.newfeature;

import java.util.ArrayList;
import java.util.List;

/**
 * 泛型
 */
public class Demo08 {
	public static void main(String[] args) {
		List <? super Teacher> list = null;
		list = new ArrayList<Object>();
		list = new ArrayList<Person>();
		list = new ArrayList<Teacher>();
		list = new ArrayList<BigTeacher>();
		list = new ArrayList<Dog>();
		list = new ArrayList<Cat>();
		list = new ArrayList<Table>();
	}

}
```

5. 泛型原理 - 泛型擦除

泛型是在编译过程中起作用的，在编译的过程中泛型的所有信息都会被擦除掉，在编译完成生成的.class文件中，没有任何和泛型相关的信息。这个过程称之为泛型擦除。

泛型擦除的原理非常的简单，在编译器编译类的过程中，一旦遇到泛型，会将所有的泛型都转换为该泛型的上边界类型，并在需要的地方加上必要的类型检查和类型强转。

```java
class God<T extends Live>{
    public T kill(T t){
        System.out.println("上帝弄死了"+t);
        return t;
    }

    public T save(T t){
        System.out.println("上帝弄活了"+t);
        return t;
    }
}

public class Demo09 {
    public static void main(String[] args) {
        God<Person> personGod = new God<Person>();
        Person p = personGod.kill(new Person());
        p = personGod.save(p);
    }
}
```




```java
class God{
    public Live kill(Live t){
        System.out.println("上帝弄死了"+t);
        return t;
    }

    public Live save(Live t){
        System.out.println("上帝弄活了"+t);
        return t;
    }
}

public class Demo09 {
    public static void main(String[] args) {
        God personGod = new God();
        Person p =(Person)personGod.kill(new Person());
        p = (Person)personGod.save(p);
    }
}
```

注解

2018年8月29日 15:09

1. 注解概述

注释:给人看的提示信息

注解:给程序看的提示信息

jdk自带的注解：

@Override

提示编译器，当前方法要重写父类方法，如果没有重写成功，请报错

@Deprecated

提示编译器，当前方法已经被过时，当有人调用此方法时，警告方法过时

@SuppressWarnings

提示编译器，不要再提示指定类型的警告了

自定义注解：

也可以自己开发特定功能的注解，来实现特定的功能

目前，在企业级开发中利用注解实现轻量级的配置是非常流行的做法

2. 自定义注解的开发

自定义注解开发需要经过三个步骤：

声明一个注解类描述一个注解

使用注解

反射注解

a. 声明注解：

i. 声明注解类

想要声明一个注解，可以通过@interface关键字来声明注解类

ii. 使用元注解修饰注解类

注解类创建出来后，还可以使用元注解来进行修饰

所谓元注解就是jdk提供的用来声明注解类功能的注解：

@Target

声明当前注解可以用在哪些个地方

如果不声明则默认可以用在任意位置

可以在Target元注解的属性中 通过ElementType来指定可以使用的位置

@Retention

声明当前注解可以被保留到什么时候

如果不声明则默认只会在源文件中存在，编译过后就被丢弃

可以在Retention元注解的属性中 通过RetentionPolicy来指定可以保留的位置

RetentionPolicy.SOURCE

保留到源代码阶段，编译时丢弃，通常是给编译器看的

RetentionPolicy.CLASS

保留到.class文件中，类加载时丢弃，通常是给类加载器看的

RetentionPolicy.RUNTIME

保留到字节码阶段，到运行阶段都能看到，通常是给程序看的

需要在运行阶段反射操作的注解，必须配置为此项

@Documented

声明当前注解是否可以被文档工具提取到自动生成的doc文档中

如果不声明则默认注解不会被文档工具提取到doc文档中

@Inherited

声明当前注解是否具有继承性，即，如果父类用这个注解声明后，子类是否自动继承这个注解

如果不声明，则默认注解没有继承性

iii. 注解中还可以声明属性

在注解中声明属性的过程非常类似于在接口中定义方法，只不过 注解中声明的属性 必须是public的或者不写访问权限修饰符 默认就是public的。

注解中的属性只能是 八种基本数据类型 Class类型 String类型 其他注解类型 及这些类型的一维数组 类型

声明的属性必须在使用注解的过程中，在注解后通过小括号在其中为这些属性赋值

可以在为注解声明属性的过程中，通过default关键字 为注解的属性设定默认值，有默认值的属性 可以在注解使用的过程中不设置值，如果不设置值，默认采用默认值

如果注解中只有一个属性要赋值，且这个属性的名字是value，则在使用注解指定value属性值时，"value="可以省略

如果注解中某一个属性要赋值，而这个属性是数组类型，但给如的数组中只有一个值，则包裹这个值的{}可以省略

b. 使用注解

在注解可以声明的位置，使用注解即可，注意要求赋值的属性要给正确值。

c. 反射注解

Class Constructor Method Field 类上都提供了操作注解相关的方法，通过这些方法就可以反射注解，来控制程序的运行

**注意，只有RetentionPolicy.RUNTIME注解才可以被反射

boolean	**isAnnotationPresent**(Class<? extends Annotation> annotationClass) 如果指定类型的注释存在于此元素上，则返回 true，否则返回 false。
<A extends Annotation> A	**getAnnotation**(Class<A> annotationClass) 如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null。
Annotation[]	**getAnnotations**() 返回此元素上存在的所有注释。
Annotation[]	**getDeclaredAnnotations**() 返回直接存在于此元素上的所有注释。

案例：警察罚款

```
package cn.tedu.annotation;

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

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface Level{
    String value() default "交警";
}
```

```
@Level("警督")
public class Police {
    public static void main(String[] args) {
        System.out.println("敬个礼，您好，您超速了，罚款200元~~");
        Class clz = Police.class;
        if(clz.isAnnotationPresent(Level.class)){
            Level l = (Level) clz.getAnnotation(Level.class);
            if("协警".equals(l.value())){
                System.out.println("少罚点，罚50得了~~");
            }else if("交警".equals(l.value())){
                System.out.println("抽根烟，200给您，分就别扣了~~");
            }else if("刑警".equals(l.value())){
                System.out.println("交200，赶紧走~~");
            }else{
                System.out.println("交200，不够还有~~");
            }
        }else{
            System.out.println("打一顿，扭送派出所~~");
        }
    }
}
```

单例设计模式

2018年8月29日 16:33

1. 单例设计模式概述

所谓的设计模式就是前人在开发java程序时总结的套路，目前java公认有23中设计模式，单例设计模式就是其中的一种

装饰设计模式
单例设计模式

有的时候，希望程序中某一个类的对象只能有一个，此时就需要用到单例设计模式。

2. 单例设计模式的开发

a. 饿汉式单例设计模式

私有化构造方法
在类的内部创建该类的对象，保存到静态的成员属性中
提供静态的获取该对象的方法

优点：
开发简单，没有多线程并发安全问题
缺点：
类一加载，对象就被创建，一直驻留在内存中，理论在类加载到第一次使用之间，对象浪费内存
案例：

```
class 曹县领导人{
    private static 曹县领导人 ldr = new 曹县领导人("新三胖");

    private String name;

    public static 曹县领导人 getInstance(){
```

```
        return ldr;
    }

    private 曹县领导人(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

b. 懒汉式单例设计模式

私有化构造方法

在类的内部提供保存该类的对象的属性，但初值为null

提供静态的获取该类的对象的方法，在该方法中判断，如果类中的保存的该类的对象的属性为null，则创建该类的对象并存入该属性中，并返回该对象，如果不为null，则直接得到类中保存该类的对象并返回

要注意，要在获取对象的方法上，加同步代码块，保证多线程环境下的并发安全。

优点：

开发简单，在真正用到时才创建对象，不会浪费内存

缺点：

需要使用同步代码块同步方法，在多线程场景下效率低下

案例：

```
class 曹县领导人{
    private static 曹县领导人 ldr = null;

    private String name;

    private 曹县领导人(String name) {
        this.name = name;
    }
```

```
    public static synchronized 曹县领导人 getInstance(){
        if(ldr == null){
            ldr = new 曹县领导人("鑫三胖");
        }
        return ldr;
    }

    public String getName() {
        return name;
    }
}
```