Java泛型详解

JayDroid 关注
2 2019.05.12 02:05:57 字数 1,337 阅读 24,536

2516326-5475e88a458a09e4.png

一，打破砂锅问到底

> 泛型存在的意义?
> 泛型类，泛型接口，泛型方法如何定义?
> 如何限定类型变量?
> 泛型中使用的约束和局限性有哪些?
> 泛型类型的继承规则是什么?
> 泛型中的通配符类型是什么?
> 如何获取泛型的参数类型?
> 虚拟机是如何实现泛型的?
> 在日常开发中是如何运用泛型的?

JayDroid 关注
总资产6 (约0.56元)

无法在Android Studio 3.6上启用Gradle的Offline Mode
阅读 794

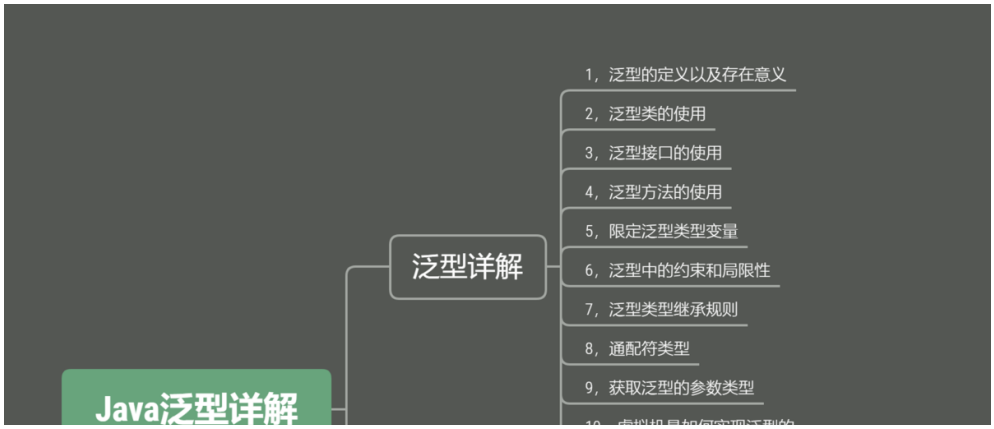
LeetCode_0001_两数之和
阅读 511

Java泛型详解

JayDroid 关注 赞赏支持

推荐阅读

闭关修炼21天，“啃完”283页pdf，我终于4面拿下字节跳动offer
阅读 94,189



Java泛型详解.png

二，晓之以理动之以码

1，泛型的定义以及存在意义

泛型，即“参数化类型”。就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。
例如：GenericClass<T>{}

> 一些常用的泛型类型变量：
> E：元素（Element），多用于java集合框架
> K：关键字（Key）
> N：数字（Number）
> T：类型（Type）
> V：值（Value）

如果要实现不同类型的加法，每种类型都需要重载一个add方法

```
package com.jay.java.泛型.needGeneric;

/**
 * Author: Jay On 2019/5/9 16:06
 * <p>
 * Description: 为什么使用泛型
 */
public class NeedGeneric1 {

    private static int add(int a, int b) {
        System.out.println(a + "+" + b + "=" + (a + b));
        return a + b;
    }

    private static float add(float a, float b) {
        System.out.println(a + "+" + b + "=" + (a + b));
        return a + b;
    }

    private static double add(double a, double b) {
        System.out.println(a + "+" + b + "=" + (a + b));
        return a + b;
    }

}
```


```
    private static <T extends Number> double add(T a, T b) {
        System.out.println(a + "+" + b + "=" + (a.doubleValue() + b.doubleValue()));
        return a.doubleValue() + b.doubleValue();
    }

    public static void main(String[] args) {
        NeedGeneric1.add(1, 2);
        NeedGeneric1.add(1f, 2f);
        NeedGeneric1.add(1d, 2d);
        NeedGeneric1.add(Integer.valueOf(1), Integer.valueOf(2));
        NeedGeneric1.add(Float.valueOf(1), Float.valueOf(2));
        NeedGeneric1.add(Double.valueOf(1), Double.valueOf(2));
    }
}

```

取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易现"java.lang. ClassCast Exception"异常。

```
package com.jay.java.泛型.needGeneric;

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

/**
 * Author: Jay On 2019/5/9 16:23
 * <p>
 * Description: 为什么要使用泛型
 */
public class NeedGeneric2 {
    static class C{

    }
    public static void main(String[] args) {
        List list=new ArrayList();
        list.add("A");
        list.add("B");
        list.add(new C());
        list.add(100);
        //1.当我们将一个对象放入集合中，集合不会记住此对象的类型，当再次从集合中取出此对象时，改对象的编
        //2.因此，//1处取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现"java.lang.C
        for (int i = 0; i < list.size(); i++) {
//            System.out.println(list.get(i));
            String value= (String) list.get(i);
            System.out.println(value);
        }
    }
}

```

所以使用泛型的意义在于

1,适用于多种数据类型执行相同的代码（代码复用）

2, 泛型中的类型在使用时指定，不需要强制类型转换（类型安全，编译器会检查类型）

2，泛型类的使用

定义一个泛型类：public class `GenericClass<T>` {}

```
package com.jay.java.泛型.DefineGeneric;

/**
 * Author: Jay On 2019/5/9 16:49
 * <p>
 * Description: 泛型类
 */

```

```
public class GenericClass<T> {
    private T data;

    public T getData() {
        return data;
    }

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

    public static void main(String[] args) {
        GenericClass<String> genericClass=new GenericClass<>();
        genericClass.setData("Generic Class");
        System.out.println(genericClass.getData());
    }
}
```

3，泛型接口的使用

定义一个泛型接口：public interface `GenericIntercace<T>` {}

```
/**
 * Author: Jay On 2019/5/9 16:57
 * <p>
 * Description: 泛型接口
 */
public interface GenericIntercace<T> {
     T getData();
}
```

实现泛型接口方式一：public class `ImplGenericInterface1<T>` implements `GenericIntercace<T>`

```
/**
 * Author: Jay On 2019/5/9 16:59
 * <p>
 * Description: 泛型接口实现类-泛型类实现方式
 */
public class ImplGenericInterface1<T> implements GenericIntercace<T> {
    private T data;

    private void setData(T data) {
        this.data = data;
    }

    @Override
    public T getData() {
        return data;
    }

    public static void main(String[] args) {
        ImplGenericInterface1<String> implGenericInterface1 = new ImplGenericInterface1<
        implGenericInterface1.setData("Generic Interface1");
        System.out.println(implGenericInterface1.getData());
    }
}
```

实现泛型接口方式二：public class ImplGenericInterface2 implements `GenericIntercace<String>` {}

```
/**
 * Author: Jay On 2019/5/9 17:01
 * <p>
 * Description: 泛型接口实现类-指定具体类型实现方式
```

```
 */
public class ImplGenericInterface2 implements GenericIntercace<String> {
    @Override
    public String getData() {
        return "Generic Interface2";
    }

    public static void main(String[] args) {
        ImplGenericInterface2 implGenericInterface2 = new ImplGenericInterface2();
        System.out.println(implGenericInterface2.getData());
    }
}
```

4，泛型方法的使用

定义一个泛型方法： private static <T> T genericAdd(T a, T b) {}

```
/**
 * Author: Jay On 2019/5/10 10:46
 * <p>
 * Description: 泛型方法
 */
public class GenericMethod1 {
    private static int add(int a, int b) {
        System.out.println(a + "+" + b + "=" + (a + b));
        return a + b;
    }

    private static <T> T genericAdd(T a, T b) {
        System.out.println(a + "+" + b + "="+a+b);
        return a;
    }

    public static void main(String[] args) {
        GenericMethod1.add(1, 2);
        GenericMethod1.<String>genericAdd("a", "b");
    }
}
```

```
/**
 * Author: Jay On 2019/5/10 16:22
 * <p>
 * Description: 泛型方法
 */
public class GenericMethod3 {

    static class Animal {
        @Override
        public String toString() {
            return "Animal";
        }
    }

    static class Dog extends Animal {
        @Override
        public String toString() {
            return "Dog";
        }
    }

    static class Fruit {
        @Override
        public String toString() {
            return "Fruit";
        }
    }

    static class GenericClass<T> {
```

```

        public void show01(T t) {
            System.out.println(t.toString());
        }

        public <T> void show02(T t) {
            System.out.println(t.toString());
        }

        public <K> void show03(K k) {
            System.out.println(k.toString());
        }
    }

    public static void main(String[] args) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        Fruit fruit = new Fruit();
        GenericClass<Animal> genericClass = new GenericClass<>();
        //泛型类在初始化时限制了参数类型
        genericClass.show01(dog);
//        genericClass.show01(fruit);

        //泛型方法的参数类型在使用时指定
        genericClass.show02(dog);
        genericClass.show02(fruit);

        genericClass.<Animal>show03(animal);
        genericClass.<Animal>show03(dog);
        genericClass.show03(fruit);
//        genericClass.<Dog>show03(animal);
    }
}
```

5，限定泛型类型变量

1,对类的限定：public class `TypeLimitForClass<T extends List & Serializable>{}`
2,对方法的限定：public static `<T extends Comparable<T>>` T getMin(T a, T b) {}

```
/**
 * Author: Jay On 2019/5/10 16:38
 * <p>
 * Description: 类型变量的限定-方法
 */
public class TypeLimitForMethod {

    /**
     * 计算最小值
     * 如果要实现这样的功能就需要对泛型方法的类型做出限定
     */
//    private static <T> T getMin(T a, T b) {
//        return (a.compareTo(b) > 0) ? a : b;
//    }

    /**
     * 限定类型使用extends关键字指定
     * 可以使类，接口，类放在前面接口放在后面用&符号分割
     * 例如: <T extends ArrayList & Comparable<T> & Serializable>
     */
    public static <T extends Comparable<T>> T getMin(T a, T b) {
        return (a.compareTo(b) < 0) ? a : b;
    }

    public static void main(String[] args) {
        System.out.println(TypeLimitForMethod.getMin(2, 4));
        System.out.println(TypeLimitForMethod.getMin("a", "r"));
    }
}
```

```java
/**
 * Author: Jay On 2019/5/10 17:02
 * <p>
 * Description: 类型变量的限定-类
 */
public class TypeLimitForClass<T extends List & Serializable> {
    private T data;

    public T getData() {
        return data;
    }

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

    public static void main(String[] args) {
        ArrayList<String> stringArrayList = new ArrayList<>();
        stringArrayList.add("A");
        stringArrayList.add("B");
        ArrayList<Integer> integerArrayList = new ArrayList<>();
        integerArrayList.add(1);
        integerArrayList.add(2);
        integerArrayList.add(3);
        TypeLimitForClass<ArrayList> typeLimitForClass01 = new TypeLimitForClass<>();
        typeLimitForClass01.setData(stringArrayList);
        TypeLimitForClass<ArrayList> typeLimitForClass02 = new TypeLimitForClass<>();
        typeLimitForClass02.setData(integerArrayList);

        System.out.println(getMinListSize(typeLimitForClass01.getData().size(), typeLimi

    }

    public static <T extends Comparable<T>> T getMinListSize(T a, T b) {
        return (a.compareTo(b) < 0) ? a : b;
    }
```

6，泛型中的约束和局限性

1,不能实例化泛型类
2,静态变量或方法不能引用泛型类型变量，但是静态泛型方法是可以的
3,基本类型无法作为泛型类型
4,无法使用instanceof关键字或==判断泛型类的类型
5,泛型类的原生类型与所传递的泛型无关，无论传递什么类型，原生类是一样的
6,泛型数组可以声明但无法实例化
7,泛型类不能继承Exception或者Throwable
8,不能捕获泛型类型限定的异常但可以将泛型限定的异常抛出

```java
/**
 * Author: Jay On 2019/5/10 17:41
 * <p>
 * Description: 泛型的约束和局限性
 */
public class GenericRestrict1<T> {
    static class NormalClass {

    }

    private T data;

    /**
     * 不能实例化泛型类
     * Type parameter 'T' cannot be instantiated directly
     */
    public void setData() {
        //this.data = new T();
```

```
    }

    /**
     * 静态变量或方法不能引用泛型类型变量
     * 'com.jay.java.泛型.restrict.GenericRestrict1.this' cannot be referenced from a sta
     */
//    private static T result;

//    private static T getResult() {
//        return result;
//    }

    /**
     * 静态泛型方法是可以的
     */
    private static <K> K getKey(K k) {
        return k;
    }

    public static void main(String[] args) {
        NormalClass normalClassA = new NormalClass();
        NormalClass normalClassB = new NormalClass();
        /**
         * 基本类型无法作为泛型类型
         */
//        GenericRestrict1<int> genericRestrictInt = new GenericRestrict1<>();
        GenericRestrict1<Integer> genericRestrictInteger = new GenericRestrict1<>();
        GenericRestrict1<String> genericRestrictString = new GenericRestrict1<>();
        /**
         * 无法使用instanceof关键字判断泛型类的类型
         * Illegal generic type for instanceof
         */
//        if(genericRestrictInteger instanceof GenericRestrict1<Integer>){
//            return;
//        }

        /**
         * 无法使用“==”判断两个泛型类的实例
         * Operator '==' cannot be applied to this two instance
         */
//        if (genericRestrictInteger == genericRestrictString) {
//            return;
//        }

        /**
         * 泛型类的原生类型与所传递的泛型无关，无论传递什么类型，原生类是一样的
         */
        System.out.println(normalClassA == normalClassB);//false
        System.out.println(genericRestrictInteger == genericRestrictInteger);//
        System.out.println(genericRestrictInteger.getClass() == genericRestrictString.ge
        System.out.println(genericRestrictInteger.getClass());//com.jay.java.泛型.restric
        System.out.println(genericRestrictString.getClass());//com.jay.java.泛型.restrict

        /**
         * 泛型数组可以声明但无法实例化
         * Generic array creation
         */
        GenericRestrict1<String>[] genericRestrict1s;
//        genericRestrict1s = new GenericRestrict1<String>[10];
        genericRestrict1s = new GenericRestrict1[10];
        genericRestrict1s[0]=genericRestrictString;
    }

}
```

```
/**
 * Author: Jay On 2019/5/10 18:45
 * <p>
 * Description: 泛型和异常
 */
public class GenericRestrict2 {

    private class MyException extends Exception {
```

```
    }

    /**
     * 泛型类不能继承Exception或者Throwable
     * Generic class may not extend 'java.lang.Throwable'
     */
//    private class MyGenericException<T> extends Exception {
//    }
//
//    private class MyGenericThrowable<T> extends Throwable {
//    }

    /**
     * 不能捕获泛型类型限定的异常
     * Cannot catch type parameters
     */
    public <T extends Exception> void getException(T t) {
//        try {
//
//        } catch (T e) {
//
//        }
    }

    /**
     *可以将泛型限定的异常抛出
     */
    public <T extends Throwable> void getException(T t) throws T {
        try {

        } catch (Exception e) {
            throw t;
        }
    }
}
```

7，泛型类型继承规则

1,对于泛型参数是继承关系的泛型类之间是没有继承关系的
2,泛型类可以继承其它泛型类，例如: public class ArrayList<E> extends AbstractList<E>
3,泛型类的继承关系在使用中同样会受到泛型类型的影响

```
/**
 * Author: Jay On 2019/5/10 19:13
 * <p>
 * Description: 泛型继承规则测试类
 */
public class GenericInherit<T> {
    private T data1;
    private T data2;

    public T getData1() {
        return data1;
    }

    public void setData1(T data1) {
        this.data1 = data1;
    }

    public T getData2() {
        return data2;
    }

    public void setData2(T data2) {
        this.data2 = data2;
    }

    public static <V> void setData2(GenericInherit<Father> data2) {

    }
```

```
28
29      public static void main(String[] args) {
30  //        Son 继承自 Father
31          Father father = new Father();
32          Son son = new Son();
33          GenericInherit<Father> fatherGenericInherit = new GenericInherit<>();
34          GenericInherit<Son> sonGenericInherit = new GenericInherit<>();
35          SubGenericInherit<Father> fatherSubGenericInherit = new SubGenericInherit<>();
36          SubGenericInherit<Son> sonSubGenericInherit = new SubGenericInherit<>();
37
38          /**
39           * 对于传递的泛型类型是继承关系的泛型类之间是没有继承关系的
40           * GenericInherit<Father> 与GenericInherit<Son> 没有继承关系
41           * Incompatible types.
42           */
43          father = new Son();
44  //        fatherGenericInherit=new GenericInherit<Son>();
45
46          /**
47           * 泛型类可以继承其它泛型类，例如: public class ArrayList<E> extends AbstractList<E>
48           */
49          fatherGenericInherit=new SubGenericInherit<Father>();
50
51          /**
52           *泛型类的继承关系在使用中同样会受到泛型类型的影响
53           */
54          setData2(fatherGenericInherit);
55  //        setData2(sonGenericInherit);
56          setData2(fatherSubGenericInherit);
57  //        setData2(sonSubGenericInherit);
58
59      }
60
61      private static class SubGenericInherit<T> extends GenericInherit<T> {
62
63      }
64
```

8，通配符类型

1, `<? extends Parent>` 指定了泛型类型的上届

2, `<? super Child>` 指定了泛型类型的下届

3, `<?>` 指定了没有限制的泛型类型

通配符测试类结构.png

```java
/**
 * Author: Jay On 2019/5/10 19:51
 * <p>
 * Description: 泛型通配符测试类
 */
public class GenericByWildcard {
    private static void print(GenericClass<Fruit> fruitGenericClass) {
        System.out.println(fruitGenericClass.getData().getColor());
    }

    private static void use() {
        GenericClass<Fruit> fruitGenericClass = new GenericClass<>();
        print(fruitGenericClass);
        GenericClass<Orange> orangeGenericClass = new GenericClass<>();
        //类型不匹配,可以使用<? extends Parent> 来解决
//        print(orangeGenericClass);
    }

    /**
     * <? extends Parent> 指定了泛型类型的上届
     */
    private static void printExtends(GenericClass<? extends Fruit> genericClass) {
        System.out.println(genericClass.getData().getColor());
    }

    public static void useExtend() {
        GenericClass<Fruit> fruitGenericClass = new GenericClass<>();
        printExtends(fruitGenericClass);
        GenericClass<Orange> orangeGenericClass = new GenericClass<>();
        printExtends(orangeGenericClass);

        GenericClass<Food> foodGenericClass = new GenericClass<>();
        //Food是Fruit的父类，超过了泛型上届范围，类型不匹配
//        printExtends(foodGenericClass);

        //表示GenericClass的类型参数的上届是Fruit
        GenericClass<? extends Fruit> extendFruitGenericClass = new GenericClass<>();
        Apple apple = new Apple();
        Fruit fruit = new Fruit();
        /*
         * 道理很简单，？ extends X  表示类型的上界，类型参数是X的子类，那么可以肯定的说，
         * get方法返回的一定是个X（不管是X或者X的子类）编译器是可以确定知道的。
         * 但是set方法只知道传入的是个X，至于具体是X的那个子类，不知道。
         * 总结：主要用于安全地访问数据，可以访问X及其子类型，并且不能写入非null的数据。
         */
//        extendFruitGenericClass.setData(apple);
//        extendFruitGenericClass.setData(fruit);

        fruit = extendFruitGenericClass.getData();

    }

    /**
     * <? super Child> 指定了泛型类型的下届
     */
    public static void printSuper(GenericClass<? super Apple> genericClass) {
        System.out.println(genericClass.getData());
    }

    public static void useSuper() {
        GenericClass<Food> foodGenericClass = new GenericClass<>();
        printSuper(foodGenericClass);

        GenericClass<Fruit> fruitGenericClass = new GenericClass<>();
        printSuper(fruitGenericClass);

        GenericClass<Apple> appleGenericClass = new GenericClass<>();
        printSuper(appleGenericClass);

        GenericClass<HongFuShiApple> hongFuShiAppleGenericClass = new GenericClass<>();
        // HongFuShiApple 是Apple的子类，达不到泛型下届，类型不匹配
//        printSuper(hongFuShiAppleGenericClass);

```

```
        GenericClass<Orange> orangeGenericClass = new GenericClass<>();
        // Orange和Apple是兄弟关系，没有继承关系，类型不匹配
//        printSuper(orangeGenericClass);

        //表示GenericClass的类型参数的下界是Apple
        GenericClass<? super Apple> supperAppleGenericClass = new GenericClass<>();
        supperAppleGenericClass.setData(new Apple());
        supperAppleGenericClass.setData(new HongFuShiApple());
        /*
         * ？ super X 表示类型的下界，类型参数是X的超类（包括X本身），
         * 那么可以肯定的说，get方法返回的一定是个X的超类，那么到底是哪个超类？不知道，
         * 但是可以肯定的说，Object一定是它的超类，所以get方法返回Object。
         * 编译器是可以确定知道的。对于set方法来说，编译器不知道它需要的确切类型，但是X和X的子类可以安全
         * 总结：主要用于安全地写入数据，可以写入X及其子类型。
         */
//        supperAppleGenericClass.setData(new Fruit());

        //get方法只会返回一个Object类型的值。
        Object data = supperAppleGenericClass.getData();
    }

    /**
     * <?> 指定了没有限定的通配符
     */
    public static void printNonLimit(GenericClass<?> genericClass) {
        System.out.println(genericClass.getData());
    }

    public static void useNonLimit() {
        GenericClass<Food> foodGenericClass = new GenericClass<>();
        printNonLimit(foodGenericClass);
        GenericClass<Fruit> fruitGenericClass = new GenericClass<>();
        printNonLimit(fruitGenericClass);
        GenericClass<Apple> appleGenericClass = new GenericClass<>();
        printNonLimit(appleGenericClass);

        GenericClass<?> genericClass = new GenericClass<>();
        //setData 方法不能被调用， 甚至不能用 Object 调用;
//        genericClass.setData(foodGenericClass);
//        genericClass.setData(new Object());
        //返回值只能赋给 Object
        Object object = genericClass.getData();

    }

}
```

9，获取泛型的参数类型

Type是什么

这里的Type指java.lang.reflect.Type, 是Java中所有类型的公共高级接口, 代表了Java中的所有类型. Type体系中类型的包括：数组类型(GenericArrayType)、参数化类型(ParameterizedType)、类型变量(TypeVariable)、通配符类型(WildcardType)、原始类型(Class)、基本类型(Class), 以上这些类型都实现Type接口.

> 参数化类型,就是我们平常所用到的泛型List、Map；
> 数组类型,并不是我们工作中所使用的数组String[] 、byte[]，而是带有泛型的数组，即T[] ；
> 通配符类型, 指的是<?>, <? extends T>等等
> 原始类型, 不仅仅包含我们平常所指的类，还包括枚举、数组、注解等；
> 基本类型, 也就是我们所说的java的基本类型，即int,float,double等

```
public interface ParameterizedType extends Type {
    // 返回确切的泛型参数，如Map<String, Integer>返回[String, Integer]
    Type[] getActualTypeArguments();

    //返回当前class或interface声明的类型，如List<?>返回List
    Type getRawType();

    //返回所属类型. 如,当前类型为O<T>.I<S>，则返回O<T>. 顶级类型将返回null
    Type getOwnerType();
}
```

```
/**
 * Author: Jay On 2019/5/11 22:41
 * <p>
 * Description: 获取泛型类型测试类
 */
public class GenericType<T> {
    private T data;

    public T getData() {
        return data;
    }

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

    public static void main(String[] args) {
        GenericType<String> genericType = new GenericType<String>() {};
        Type superclass = genericType.getClass().getGenericSuperclass();
        //getActualTypeArguments 返回确切的泛型参数，如Map<String, Integer>返回[String, Inte
        Type type = ((ParameterizedType) superclass).getActualTypeArguments()[0];
        System.out.println(type);//class java.lang.String
    }
}
```

10，虚拟机是如何实现泛型的

Java泛型是Java1.5之后才引入的，为了向下兼容。Java采用了C++完全不同的实现思想。Java中的泛型更多的看起来像是编译期用的
Java中泛型在运行期是不可见的，会被擦除为它的上级类型。如果是没有限定的泛型参数类型，就会被替换为Object.

```
GenericClass<String> stringGenericClass=new GenericClass<>();
GenericClass<Integer> integerGenericClass=new GenericClass<>();
```

C++中GenericClass<String>和GenericClass<Integer>是两个不同的类型
Java进行了类型擦除之后统一改为GenericClass<Object>

```
/**
 * Author: Jay On 2019/5/11 16:11
 * <p>
 * Description:泛型原理测试类
 */
public class GenericTheory {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("Key", "Value");
        System.out.println(map.get("Key"));
        GenericClass<String, String> genericClass = new GenericClass<>();
        genericClass.put("Key", "Value");
        System.out.println(genericClass.get("Key"));
    }
```

```

    public static class GenericClass<K, V> {
        private K key;
        private V value;

        public void put(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public V get(V key) {
            return value;
        }
    }

    /**
     * 类型擦除后GenericClass2<Object>
     * @param <T>
     */
    private class GenericClass2<T> {

    }

    /**
     * 类型擦除后GenericClass3<ArrayList>
     * 当使用到Serializable时会将相应代码强制转换为Serializable
     * @param <T>
     */
    private class GenericClass3<T extends ArrayList & Serializable> {

    }
}

```

对应的字节码文件

```
 public static void main(String[] args) {
        Map<String, String> map = new HashMap();
        map.put("Key", "Value");
        System.out.println((String)map.get("Key"));
        GenericTheory.GenericClass<String, String> genericClass = new GenericTheory.Gener
        genericClass.put("Key", "Value");
        System.out.println((String)genericClass.get("Key"));
    }
```

三，学以致用

1，泛型解析JSON数据封装

api返回的json数据

```
{
    "code":200,
    "msg":"成功",
    "data":{
        "name":"Jay",
        "email":"10086"
    }
}
```

BaseResponse .java

```
/**
 * Author: Jay On 2019/5/11 20:48
```

```
 * <p>
 * Description: 接口数据接收基类
 */
public class BaseResponse {

    private int code;
    private String msg;

    public int getCode() {
        return code;
    }

    public void setCode(int code) {
        this.code = code;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

UserResponse.java

```
/**
 * Author: Jay On 2019/5/11 20:49
 * <p>
 * Description: 用户信息接口实体类
 */
public class UserResponse<T> extends BaseResponse {
    private T data;

    public T getData() {
        return data;
    }

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

2，泛型+反射实现巧复用工具类

```
/**
 * Author: Jay On 2019/5/11 21:05
 * <p>
 * Description: 泛型相关的工具类
 */
public class GenericUtils {

    public static class Movie {
        private String name;
        private Date time;

        public String getName() {
            return name;
        }

        public Date getTime() {
            return time;
        }

        public Movie(String name, Date time) {
            this.name = name;
            this.time = time;
```

```
        }

        @Override
        public String toString() {
            return "Movie{" + "name='" + name + '\'' + ", time=" + time + '}';
        }
    }

    public static void main(String[] args) {
        List<Movie> movieList = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            movieList.add(new Movie("movie" + i, new Date()));
        }
        System.out.println("排序前:" + movieList.toString());

        GenericUtils.sortAnyList(movieList, "name", true);
        System.out.println("按name正序排: " + movieList.toString());

        GenericUtils.sortAnyList(movieList, "name", false);
        System.out.println("按name逆序排: " + movieList.toString());
    }

    /**
     * 对任意集合的排序方法
     * @param targetList 要排序的实体类List集合
     * @param sortField  排序字段
     * @param sortMode   true正序, false逆序
     */
    public static <T> void sortAnyList(List<T> targetList, final String sortField, final
        if (targetList == null || targetList.size() < 2 || sortField == null || sortFiel
            return;
        }
        Collections.sort(targetList, new Comparator<Object>() {
            @Override
            public int compare(Object obj1, Object obj2) {
                int retVal = 0;
                try {
                    // 获取getXxx()方法名称
                    String methodStr = "get" + sortField.substring(0, 1).toUpperCase() +
                    Method method1 = ((T) obj1).getClass().getMethod(methodStr, null);
                    Method method2 = ((T) obj2).getClass().getMethod(methodStr, null);
                    if (sortMode) {
                        retVal = method1.invoke(((T) obj1), null).toString().compareTo(me
                    } else {
                        retVal = method2.invoke(((T) obj2), null).toString().compareTo(me
                    }
                } catch (Exception e) {
                    System.out.println("List<" + ((T) obj1).getClass().getName() + ">排序
                    e.printStackTrace();
                }
                return retVal;
            }
        });
    }
}
```

3，Gson库中的泛型的使用-TypeToken

```
/**
 * Author: Jay On 2019/5/11 22:11
 * <p>
 * Description: Gson库中的泛型使用
 */
public class GsonGeneric {
    public static class Person {
        private String name;
        private int age;

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

```
        }

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

    public static void main(String[] args) {
        Gson gson = new Gson();
        List<Person> personList = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            personList.add(new Person("name" + i, 18 + i));
        }
        // Serialization
        String json = gson.toJson(personList);
        System.out.println(json);
        // Deserialization
        Type personType = new TypeToken<List<Person>>() {}.getType();
        List<Person> personList2 = gson.fromJson(json, personType);
        System.out.println(personList2);
    }
}
```

测试代码

Java泛型详解

泛型 泛型由来 泛型字面意思不知道是什么类型，但又好像什么类型都是。看前面用到的集合都有泛型的影子。 以Array...

向日花开 阅读 1,219 评论 2 赞 6

Java泛型详解

最近项目组在进行泛型代码编写时遇到很多困难，讨论下来发现大家对这个概念都是一知片解，然而在我们的项目开发过程中，又...

Caprin 阅读 4,151 评论 0 赞 46

Java泛型详解大纲

由于博客的特殊显示原因，尖括号用（）代替 泛型概述 Java泛型（generics）是JDK 5中引入的一个新特性...

冯匿 阅读 166 评论 0 赞 0

Java泛型详解

2.6 Java泛型详解 Java泛型是JDK5中引入的一个新特性，允许在定义类和接口的时候使用类型参数(type...

jianhuih 阅读 221 评论 0 赞 1

java泛型详解1

为什么要使用泛型？ 一般的类和方法，只能使用具体的类型；要么是基本类型，要么是自定义类。如果我有这种需求：可以应用...

糖醋豆腐脑 阅读 32 评论 0 赞 0