终于搞懂 Java 中的泛型啦！

> 本文已收录于：[github.com/danmuking/a...](http://cxyroad.com/”https://github.com/danmuking/all-in-one”)（持续更新）

前言

哈喽，大家好，我是 **DanMu**。今天想和大家聊聊 Java 中的泛型。

什么是泛型?

Java 泛型（Generics） 是 JDK 5 中引入的一个新特性。它允许我们通过预先定义模板，为多种不同的数据类型执行相同的逻辑，实现更好的代码复用。
Java 编译器实现了对泛型参数进行检测，并且运行我们通过泛型参数来指定传入的对象类型，比如`ArrayList<Integer> list = new ArrayList<Integer>()`就指定了这个 ArrayList 中只能存放 Integer 对象，如果传入其他类型的对象就会报错。

为什么需要泛型?

引入泛型的意义在于：

* **适用于多种数据类型执行相同的代码**（代码复用）

如果没有泛型，即使是通用的逻辑，也需要针对每种类型单独进行一次重载，比如下面这个例子：

```
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;
}
```

这不是耽误我赚个小目标的进度嘛。为了帮助我们更快的赚到小目标，Java 贴心的为我们设计了泛型方法，现在我们可以用泛型的方式来重写一下上面的代码：

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

在这端代码中，使用了一个泛型`<T extends Number>`来代替了上面出现的所有数字类型，在使用时，我们只需要指定 T 的类型，编译器就会自动的帮助我们将方法中的 T 转换为对应的类型，因此**只需要一个函数就能实现所有数字类型的加法。**

* 泛型中的类型可以在使用时指定，不需要强制类型转换（**类型安全**，编译器会**检查类型**）

比如下面这个例子：

```
List list = new ArrayList();
list.add(”联系时长”);
```

```
list.add(2.5);
list.add(”的练习生”);

```

在上述 list 中，list 中的所有元素都是Object类型，因此，我们在取出集合元素时需要手动进行强制类型转化，将Object 转换到具体的目标类型，不仅麻烦而且很容易出现`java.lang.ClassCastException`异常。
通过引入泛型，它能够为我们进行类型的约束，提供编译前的检查：

```
List<String> list = new ArrayList<String>(); // list中只能放String, 不能放其它类型的元素

```

如何使用泛型?

泛型主要的使用方式有三种：**泛型类、泛型接口、泛型方法。**

泛型类：

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
//在实例化泛型类时，必须指定T的具体类型
public class Generic<T>{

    private T key;

    public Generic(T key) {
        this.key = key;
    }

    public T getKey(){
        return key;
    }
}

```

如何实例化泛型类：

```
Generic<Integer> genericInteger = new Generic<Integer>(123456);
```

泛型接口：

```
public interface Generator<T> {
    public T method();
}
```

实现泛型接口，不指定类型：

```
class GeneratorImpl<T> implements Generator<T>{
    @Override
    public T method() {
        return null;
    }
}
```

实现泛型接口，指定类型：

```
class GeneratorImpl<T> implements Generator<String>{
    @Override
    public String method() {
        return ”hello”;
    }
}
```

泛型方法：

```
public static < E > void printArray( E[] inputArray )
{
     for ( E element : inputArray ){
        System.out.printf( "%s ", element );
     }
     System.out.println();
}
```

使用：

```
// 创建不同类型数组：Integer, Double 和 Character
Integer[] intArray = { 1, 2, 3 };
String[] stringArray = { "Hello", "World" };
printArray(intArray);
printArray(stringArray);
```

> **注意:**`public static <E> void printArray(E[] inputArray )`一般被称为静态泛型方法;在 Java 中泛型只是一个占位符，必须在传递类型后才能使用。*类在实例化时才能真正的传递类型参数*，由于静态方法的加载先于类的实例化，也就是说类中的泛型还没有传递真正的类型参数，静态的方法的加载就已经完成了，所以静态泛型方法是没有办法使用类上声明的泛型的。只能使用自己声明的

泛型的上下限

有时候我们不希望某个泛型能够被所有类型使用，比如这个例子：

```
private static void printText(T a) {
    System.out.println(a.text);
}
```

```
// 使用
add(1);

```

在方法中将会输出传入对象的 text 属性，但是如果当传入对象没有 text 属性或者 text 属性不可以直接访问，程序就会报错。OMG！这就很糟糕了，并且这段代码可以很容易的通过编译检查，只有在运行时错误才会被发现。为了减少这种情况的发生，我们需要一种机制，能够将泛型允许接受的对象类型限制在一定的范围内。
在 Java 中提供了 extends 和 super 两个关键字来限制泛型的范围：
extends 关键字声明了类型的上界，表示参数化的类型可能是所指定的类型，或者是此类型的子类

```
class Info<T extends Number>{    // 此处泛型只能是数字类型
    private T var ;        // 定义泛型变量
    public void setVar(T var){
        this.var = var ;
    }
    public T getVar(){
        return this.var ;
    }
    public String toString(){    // 直接打印
        return this.var.toString() ;
    }
}
public class demo1{
    public static void main(String args[]){
        Info<Integer> i1 = new Info<Integer>() ;        // 声明Integer的泛型对象
        Info<List> i1 = new Info<List>() ;              // 报错
    }
}

```

super 关键字声明了类型的下界，表示参数化的类型可能是指定的类型，或者是此类型的父类

```
public class demo2 {
    public static void main(String[] args) {
```

```
        List<? super Number> number = new ArrayList<Number>();
        number.add(314);
String a = ”test”;
// number.add(a); 报错
        getUperNumber(number);

    }
    // <?> 无限制通配符
    public static void getUperNumber(List<?> data) {
          System.out.println(”data :” + data.get(0));
        }
}
```

如果需要进行多种条件的限制，可以用 & 将多个条件连接起来：

```
public class Client {
    //工资低于2500元的上斑族并且站立的乘客车票打8折
    public static <T extends Staff & Passenger> void discount(T t){
        if(t.getSalary()<2500 && t.isStanding()){
            System.out.println(”恭喜你！您的车票打八折！”);
        }
    }
    public static void main(String[] args) {
        discount(new Me());
    }
}
```

点，不迷路
=======

> 好了，以上就是这篇文章的全部内容了，如果你能看到这里，**非常感谢你的支持！**
> 如果你觉得这篇文章写的还不错， 求**点赞** 求**** 求**分享** 对暖男我来说真的 **非常有用！！！**
> 白嫖不好，创作不易，各位的支持和认可，就是我创作的最大动力，我们下篇文章见！
> 如果本篇博客有任何错误，请批评指教，不胜感激 ！

> 最后推荐我的**IM项目
DiTing**（[github.com/danmuking/D...](http://cxyroad.com/
”https://github.com/danmuking/DiTing-Go”)），致力于成为一个初学者友好、易于上手的 IM 解决方案，希望能给你的学习、面试带来一点帮助，如果人才你喜欢，给个Star叭！

参考资料

[Java 基础 – 泛型机制详解](http://cxyroad.com/
”https://pdai.tech/md/java/basic/java-basic-x-generic.html”)

原文链接: https://juejin.cn/post/7383100103001587722