Java 语言高级特性前言

java 知识是作为 Android 开发的语言基础，虽然现在我们已经推出了 kotlin，但是基于以下原因我们还是需要好好牢牢掌握 java：

1）SDK 还是改成 java，kotlin 也需要编译成为 java 运行；

2）目前大量的第三方库和继承与前任的代码都是 java 所写的；

3）Java 语言应用不仅仅在 Android，就是在后台开发中也是一个最流行的语言；

4）大公司面试都要求我们有扎实的 Java 语言基础。所以，请大家不要轻视提高自己 Java 基础的机会，请大家认真学习，做好笔记，争取取得更大的进步。

Java 中的泛型

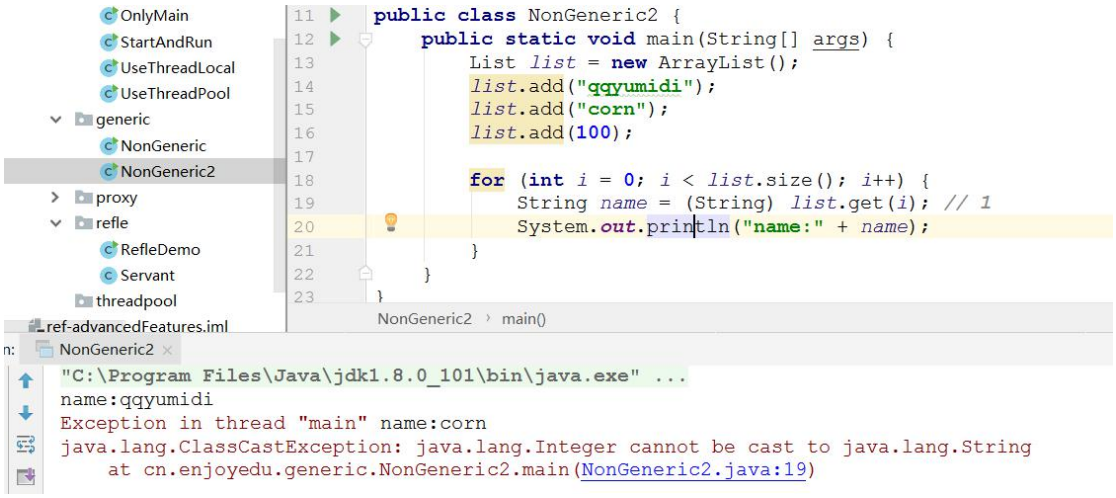
为什么我们需要泛型？

通过两段代码我们就可以知道为何我们需要泛型

```
public int addInt(int x,int y){
    return x+y;
}

public float addFloat(float x,float y){
    return x+y;
}
```

实际开发中，经常有数值类型求和的需求，例如实现 int 类型的加法，有时候还需要实现 long 类型的求和，如果还需要 double 类型的求和，需要重新在重载一个输入是 double 类型的 add 方法。



定义了一个 List 类型的集合，先向其中加入了两个字符串类型的值，随后加入一个 Integer 类型的值。这是完全允许的，因为此时 list 默认的类型为 Object 类型。在之后的循环中，由于忘记了之前在 list 中也加入了 Integer 类型的值或其他编码原因，很容易出现类似于//1 中的错误。因为编译阶段正常，而运行时会出现“java.lang.ClassCastException”异常。因此，导致此类错误编码过程中不易发现。

在如上的编码过程中，我们发现主要存在两个问题：

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，改对象的编译类型变成了 Object 类型，但其运行时类型任然为其本身类型。

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.ClassCastException”异常。

所以泛型的好处就是：

- 适用于多种数据类型执行相同的代码
- 泛型中的类型在使用时指定，不需要强制类型转换

泛型类和泛型接口

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？

顾名思义，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）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

引入一个类型变量 T（其他大写字母都可以，不过常用的就是 T，E，K，V 等等），并且用<>括起来，并放在类名的后面。泛型类是允许有多个类型变量的。

```
 */
public class NormalGeneric<T> {
    private T data;

    public NormalGeneric() {
    }

    public NormalGeneric(T data) {
        this();
        this.data = data;
    }
}

public class NormalGeneric2<T,K> {
    private T data;
    private K result;

    public NormalGeneric2() {
    }
```

泛型接口与泛型类的定义基本相同。

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

而实现泛型接口的类，有两种实现方法：

1、未传入泛型实参时：

```
public class ImplGenerator<T> implements Generator<T> {

    private T data;
```

在 new 出类的实例时，需要指定具体类型：

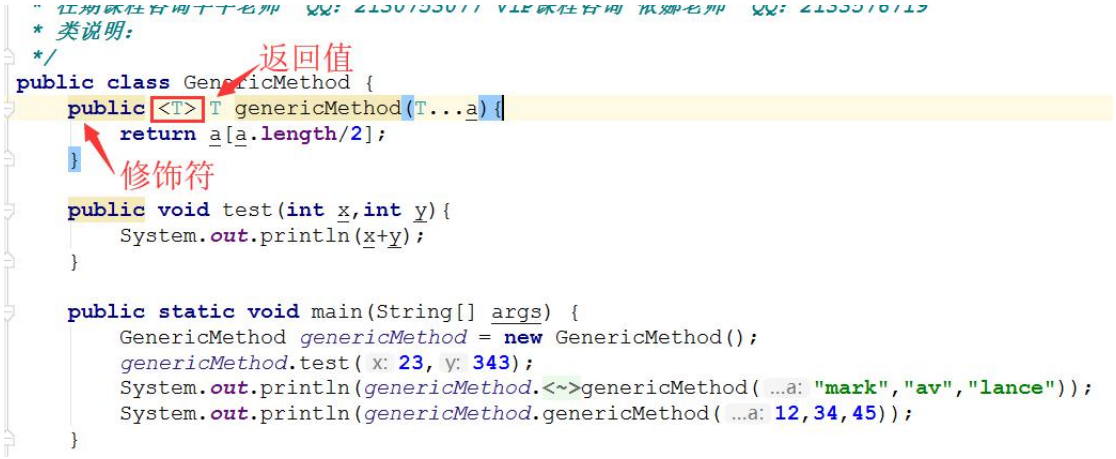
```
public static void main(String[] args) {
    ImplGenerator<String> implGenerator = new
```

2、传入泛型实参

```
public class ImplGenerator2 implements Generator<String> {
    @Override
    public String next() {
        return "OK";
    }
```

在 new 出类的实例时，和普通的类没区别。

泛型方法




```
/**
 * 类说明:
 */
public class GenericMethod {
    public <T> T genericMethod(T...a){
        return a[a.length/2];
    }

    public void test(int x,int y){
        System.out.println(x+y);
    }

    public static void main(String[] args) {
        GenericMethod genericMethod = new GenericMethod();
        genericMethod.test( x: 23, y: 343);
        System.out.println(genericMethod.<~>genericMethod( ...a: "mark","av","lance"));
        System.out.println(genericMethod.genericMethod( ...a: 12,34,45));
    }
```

泛型方法，是在调用方法的时候指明泛型的具体类型，泛型方法可以在任何地方和任何场景中使用，包括普通类和泛型类。注意泛型类中定义的普通方法和泛型方法的区别。

普通方法：

```java
public class Generic<T>{
    private T key;

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

    //虽然在方法中使用了泛型，但是这并不是一个泛型方法。
    //这只是类中一个普通的成员方法，只不过他的返回值是在声明泛型类已经声明过的泛型。
    //所以在这个方法中才可以继续使用 T 这个泛型。
    public T getKey(){
        return key;
    }
```

泛型方法

```java
/**
 * 这才是一个真正的泛型方法。
 * 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声明了一个泛型T
 * 这个T可以出现在这个泛型方法的任意位置.
 * 泛型的数量也可以为任意多个
 *    如: public <T,K> K showKeyName(Generic<T> container){
 *        ...
 *        }
 */
public <T> T showKeyName(Generic<T> container){
    System.out.println("container key :" + container.getKey());
    //当然这个例子举的不太合适，只是为了说明泛型方法的特性。
    T test = container.getKey();
    return test;
}
```

限定类型变量

有时候，我们需要对类型变量加以约束，比如计算两个变量的最小，最大值。

```java
public static <T> T min(T a,T b){
    if(a.comapareTo(b)>0) return a; else return b;
}
```

请问，如果确保传入的两个变量一定有 compareTo 方法？那么解决这个问题的方案就是将 T 限制为实现了接口 Comparable 的类

```java
public static <T extends Comparable> T min(T a,T b){
    if(a.compareTo(b)>0) return b; else return a;
}
```

T **extends** Comparable 中

T 表示应该绑定类型的子类型，Comparable 表示绑定类型，子类型和绑定类型可以是类也可以是接口。

如果这个时候，我们试图传入一个没有实现接口 Comparable 的类的实例，将会发生编译错误。

```java
public static <T extends Comparable> T min(T a,T b){
    if(a.compareTo(b)>0) return b; else return a;
}

public static <T extends Comparable & Serializable> T max(T a, T b){
    if(a.compareTo(b)>0) return a; else return b;
}

static class Test{}

public static void main(String[] args) {
    System.out.println(ArrayAlg.min( a: "mark", b: "av"));
    System.out.println(ArrayAlg.min( a: 22, b: 12));
    ArrayAlg.min(new Test(),new Test());
    System.out.println(ArrayAlg.max( a: 22, b: 12));

}
```

同时 extends 左右都允许有多个，如 T,V **extends** Comparable & Serializable

注意限定类型中，只允许有一个类，而且如果有类，这个类必须是限定列表的第一个。

这种类的限定既可以用在泛型方法上也可以用在泛型类上。

泛型中的约束和局限性

现在我们有泛型类

```java
public class Restrict<T> {
```

不能用基本类型实例化类型参数

```java
// Restrict<double> 这种不允许
Restrict<Double> restrict = new Restrict<>();
```

运行时类型查询只适用于原始类型

```java
//if(restrict instanceof Restrict<Double>){}这种不允许
//if(restrict instanceof Restrict<T>){}这种不允许
Restrict<String> restrictString = new Restrict<>();
System.out.println(restrict.getClass()==restrictString.getClass());
System.out.println(restrict.getClass().getName());
```

泛型类的静态上下文中类型变量失效

```java
//静态域或者方法里不能引用类型变量
// private static T instance;
//静态方法 本身是泛型方法就行
// private static <T> T getInstance(){}
```

不能在静态域或方法中引用类型变量。因为泛型是要在对象创建的时候才知道是什么类型的，而对象创建的代码执行先后顺序是 static 的部分，然后才是构造函数等等。所以在对象初始化之前 static 的部分已经执行了，如果你在静态部分引用的泛型，那么毫无疑问虚拟机根本不知道是什么东西，因为这个时候类还没有初始化。

不能创建参数化类型的数组

```
Restrict<Double>[] restrictArray;//可以
// Restrict<Dou...[] ... = new Restrict<Double>[10]; 不允许
```

Variable 'restrictArray' is never used

不能实例化类型变量

```
    //不能实例化类型变量
//    public Restrict() {
//        this.data = new T();
//    }
```

不能捕获泛型类的实例

```
    //泛型类不能extends Exception/Throwable
    //private class Problem<T> extends Exception{}
    //不能捕获泛型类对象
//    public <T extends Throwable> void doWork(T t){
//        try{
//
//        }catch(T e){
//            //do sth..
//        }
//    }
```

但是这样可以：

```
public <T extends Throwable> void doWork(T t) throws T{
    try{

    }catch(Throwable e){
        //do sth..
        throw t;
    }
}
```

泛型类型的继承规则

现在我们有一个类和子类

```
public class Employee {

public class Worker extends Employee {
```

有一个泛型类

```
public class Pair<T>
```

请问 Pair<Employee>和 Pair<Worker>是继承关系吗？

答案：不是，他们之间没有什么关系

```
Employee employee = new Worker();
Pair<Employee> employeePair2 = new  Pair<Worker>();
```

但是泛型类可以继承或者扩展其他泛型类，比如 List 和 ArrayList

```
    Pair<Employee> pair = new ExtendPair<>();
}

/*泛型类可以继承或者扩展其他泛型类，比如List和ArrayList*/
private static class ExtendPair<T> extends Pair<T>{

}
```

通配符类型

正是因为前面所述的，Pair<Employee>和 Pair<Worker>没有任何关系，如果我们有一个泛型类和一个方法

```
public static void print(GenericType<Fruit> p){
    System.out.println(p.getData().getColor());
}

public class GenericType<T> {
```

现在我们有继承关系的类

```
public class Fruit {

public class Orange extends Fruit {
}

public class Apple extends Fruit {

public class HongFuShi extends Apple {
}
```

则会产生这种情况：

```
public static void use(){
    GenericType<Fruit> a = new GenericType<>();
    print(a);
    GenericType<Orange> b = new GenericType<>();
    // print(b);这样不允许
}
```

为解决这个问题，于是提出了一个通配符类型 ？

有两种使用方式：

？ extends X　表示类型的上界，类型参数是 X 的子类

？ super X　表示类型的下界，类型参数是 X 的超类

这两种 方式从名字上来看，特别是 super，很有迷惑性，下面我们来仔细辨析这两种方法。

？ extends X

表示传递给方法的参数，必须是 X 的子类（包括 X 本身）

```
public static void print2(GenericType<? extends Fruit> p){
    System.out.println(p.getData().getColor());
}

public static void use2(){
    GenericType<Fruit> a = new GenericType<>();
    print2(a);
    GenericType<Orange> b = new GenericType<>();
    print2(b);
    GenericType<? extends Fruit> c = b;
```

但是对泛型类 GenericType 来说，如果其中提供了 get 和 set 类型参数变量的方法的话，set 方法是不允许被调用的，会出现编译错误

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

    public T getData() {
        return data;
    }

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

GenericType<? extends Fruit> c = a;
Apple apple = new Apple();
Fruit fruit = new Fruit();
// 这样不允许
c.setData(apple);
c.setData(fruit);
```

get 方法则没问题，会返回一个 Fruit 类型的值。

```
Fruit y = c.getData();
```

为何？

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。

? super X

表示传递给方法的参数，必须是 X 的超类（包括 X 本身）

```
public static void printSuper(GenericType<? super Apple> p){
    System.out.println(p.getData());
}

public static void useSuper(){
    GenericType<Fruit> fruitGenericType = new GenericType<>();
    GenericType<Apple> appleGenericType = new GenericType<>();
    GenericType<HongFuShi> hongFuShiGenericType = new GenericType<>();
    GenericType<Orange> orangeGenericType = new GenericType<>();
    printSuper(fruitGenericType);
    printSuper(appleGenericType);
    printSuper(hongFuShiGenericType);
    printSuper(orangeGenericType);
```

但是对泛型类 GenericType 来说，如果其中提供了 get 和 set 类型参数变量的方法的话，set 方法可以被调用的，且能传入的参数只能是 X 或者 X 的子类

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

    public T getData() {
        return data;
    }

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

// 这样不允许
x.setData(new Fruit());
x.setData(new Apple());
x.setData(new HongFuShi());
Object z = x.getData();//唯一可行的赋值
```

get 方法只会返回一个 Object 类型的值。

```
Fruit y = c.getData();
```

为何？

？ 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 的子类可以安全的转型为 X。

总结：主要用于安全地写入数据，可以写入 X 及其子类型。

无限定的通配符 ？

表示对类型没有什么限制，可以把？看成所有类型的父类，如 Pair< ?>；

比如：

ArrayList<T> al=new ArrayList<T>(); 指定集合元素只能是 T 类型

ArrayList<?> al=new ArrayList<?>();集合元素可以是任意类型，这种没有意义，一般是方法中，只是为了说明用法。

在使用上：

？ getFirst() ： 返回值只能赋给 Object，；

void setFirst(?) ： setFirst 方法不能被调用， 甚至不能用 Object 调用；

虚拟机是如何实现泛型的？

泛型思想早在 C++语言的模板（Template）中就开始生根发芽，在 Java 语言处于还没有出现泛型的版本时，只能通过 Object 是所有类型的父类和类型强制转换两个特点的配合来实现类型泛化。，由于 Java 语言里面所有的类型都继承于 java.lang.Object，所以 Object 转型成任何对象都是有可能的。但是也因为有无限的可能性，就只有程序员和运行期的虚拟机才知道这个 Object 到底是个什么类型的对象。在编译期间，编译器无法检查这个 Object 的强制转型是否成功，如果仅仅依赖程序员去保障这项操作的正确性，许多 ClassCastException 的风险就会转嫁到程序运行期之中。

泛型技术在 C#和 Java 之中的使用方式看似相同，但实现上却有着根本性的分歧，C#里面泛型无论在程序源码中、编译后的 IL 中（Intermediate Language，中间语言，这时候泛型是一个占位符），或是运行期的 CLR 中，都是切实存在的，List＜int＞与 List＜String＞就是两个不同的类型，它们在系统运行期生成，有自己的虚方法表和类型数据，这种实现称为类型膨胀，基于这种方法实现的泛型称为真实泛型。

Java 语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此，对于运行期的 Java 语言来说，ArrayList＜int＞与 ArrayList＜String＞就是同一个类，所以泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型称为伪泛型。

将一段 Java 代码编译成 Class 文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都不见了，程序又变回了 Java 泛型出现之前的写法，泛型类型都变回了原生类型

```
public static String method(List<String> stringList){
    System.out.println("List");
    return "OK";
}

public static Integer method(List<Integer> integerList){
    System.out.println("List");
    return 0;
}
```

上面这段代码是不能被编译的，因为参数 List＜Integer＞和 List＜String＞编译之后都被擦除了，变成了一样的原生类型 List＜E＞，擦除动作导致这两种方法的特征签名变得一模一样。

由于 Java 泛型的引入，各种场景（虚拟机解析、反射等）下的方法调用都可能对原有的基础产生影响和新的需求，如在泛型类中如何获取传入的参数化类型等。因此，JCP 组织对虚拟机规范做出了相应的修改，引入了诸如 Signature、LocalVariableTypeTable 等新的属性用于解决伴随泛型而来的参数类型的识别问题，Signature 是其中最重要的一项属性，它的作用就是存储一个方法在字节码层面的特征签名[3]，这个属性中保存的参数类型并不是原生类型，而是包括了参数化类型的信息。修改后的虚拟机规范要求所有能识别 49.0 以上版本的 Class 文件的虚拟机都要能正确地识别 Signature 参数。

另外，从 Signature 属性的出现我们还可以得出结论，擦除法所谓的擦除，仅仅是对方法的 Code 属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。