第3章 Java深入泛型与注解面试题汇总

3.1 泛型是什么，泛型擦除呢？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《架构设计中必不可少的泛型-Java泛型的定义与原理》

这道题想考察什么？

泛型

考察的知识点

泛型的特点和优缺点以及泛型擦除

考生应该如何回答

泛型就是一种就是一种不确定的数据类型。在Java中有着重要的地位，在面向对象编程及各种设计模式中都有非常广泛的应用。

泛型的优点

我们为什么需要使用泛型：

1. 适用于多种数据类型执行相同的代码，例如两个数据相加：

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

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

不同的类型，我们就需要增加不同的方法，但是使用泛型那我们的代表将变为：

```
public <T> T addInt(T x,T y){
    return x+y;
}
```

2. 编译检查，例如下面代码

```
List<String> list = new ArrayList();
list.add(10);//①
list.add("享学");
String name = list.get(2);//②
```

因为我们指定了List泛型类型为String，因此在代码1处编译时会报错。而在代码2处，不再需要做类型强转。

泛型的缺点

1. 静态域或者方法里不能引用泛型变量，因为泛型是在new对象的时候才知道，而类的构造方法是在静态变量之后执行。
2. 不能捕获泛型类对象

泛型擦除

Jdk中实现的泛型实际上是伪泛型，例如泛型类 Fruit<T> ，编译时 T 会被擦除，成为 Object。但是泛型擦除会带来一个复杂的问题：

桥方法

有如下代码：

```
public class Parent<T> {

    public void setSrc(T src){

    }
}
public class Child extends Parent<String>{
    @Override
    public void setSrc(String src) {
        super.setSrc(src);
    }
}
```

Parent类是一个泛型类，在经过编译时泛型擦除后其中`setSrc(T)`将会变为`setSrc(Object)`；而Child类继承与Parent并且指定了泛型类型为String。那么经过编译后这两个类应该变为：

```
public class Parent {

    public void setSrc(Object src){

    }
}
public class Child extends Parent{
    @Override
    public void setSrc(String src) {
        super.setSrc(src);
    }
}
```

父类存在`setSrc(Object)`，而子类则是`setSrc(String)`。这明显是两个不同的方法，按照Java的重写规则，子类并没有重写父类的方法，而是重载。

所以实际上子类中存在两个`setSrc`方法。一个自己的，一个是继承自父类的：

```
public void setSrc(String src)
public void setSrc(Object src)
```

那么当我们：

```
Parent o = new Child();
o.setSrc("1");
```

此时o实际类型是Child，静态类型是Parent。按照Java规则，会调用父类中的`setSrc(Object)`,如：

```
public class A{
    public void setValue(Object value){
        System.out.println("Object");
    }
}
public class B extends A{
        public void setValue(String value){
```

```
            System.out.println("String");
        }
}

public static void main(String[] args) {
    A a = new  B();
    a.setValue("1");
    a.setValue(11);
}
```

上诉代码会输出两次"Object"。然而在泛型中却不符合此规则，因为 Java 编译器帮我们处理了这种情况，在泛型中引入了"Bridge Method"—桥方法。通过查看Child.class的字节码文件：

```
public void setSrc(java.lang.String);
    descriptor: (Ljava/lang/String;)V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: invokespecial #2                  // Method Parent.setSrc:
(Ljava/lang/Object;)V
         5: return
      LineNumberTable:
        line 4: 0
        line 6: 5

public void setSrc(java.lang.Object);
    descriptor: (Ljava/lang/Object;)V
    flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: checkcast     #3                  // class java/lang/String
         5: invokevirtual #4                  // Method setSrc:
(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 1: 0
}
```

可以看到 Child 类中有两个 `setSrc` 方法，一个参数为 String 类型，一个参数为 Object 类型，参数为 Object 类型。而参数为Object的 `setSrc` 方法可以在flags中看到**ACC_BRIDGE**和**ACC_SYNTHETIC** 。其中**ACC_BRIDGE**用于说明这个方法是由编译生成的桥接方法，**ACC_SYNTHETIC**说明这个方法是由编译器生成，并且不会在源代码中出现。

在 `setSrc(Object)` 桥方法可以看到实际上会使用 `checkcast` 先进行类型转换检查，然后执行 `invokevirtual` 调用 `setSrc(String)` 方法，这样就避免了我们还能调用到父类的方法。

3.2 List<String>能否转为List<Object>

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《架构设计中必不可少的泛型-Java泛型的定义与原理》

这道题想考察什么？

1. 是否熟悉泛型的继承模式？
2. 是否了解 Java 泛型的真正实现机制？
3. 是否泛型解决多态的问题，利用“桥方法”

考察的知识点

1. 泛型的继承模式？
2. Java 泛型的真正实现机制
3. 泛型解决多态的问题，利用“桥方法”

考生应该如何回答

Java的泛型是伪泛型，编译时会进行泛型擦除（**《3.1 泛型是什么，泛型擦除呢？》**）。

因此List<Number>和 List<Integer> 最终的类型都被擦除了，无论是List<String> 还是 List<Object> 都是List类型。

既然存在泛型擦除，但是下面的代码无法通过编译检查：

```
List<String> strs = new ArrayList<Integer>();
List<Object> objects = strs;
```

编译器会帮我我们检查明显的代码问题，因此上述代码会报错，这是编译器的行为，但是如果我们将代码改为：

```
List<String> strs = (List)new ArrayList<Integer>();
List<Object> objects = (List)strs;
```

注意，每条语句我们增加了强转声明。此时编译器能够成功完成编译。**因此List<String>其实能够强转为List<Object>**。但是存在隐患：

```
List<String> strs = (List)new ArrayList<Integer>();
List<Object> objects = (List)strs;
objects.add(123);
String str =  strs.get(0);
```

上述代码使用objects（List<Object>）向集合中增加整型数据：123。然后通过 strs获取数据时，因为其类型为List<String>，但是真实数据类型为整型。此时就会发生运行时异常：

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer
cannot be cast to class java.lang.String
```

3.3 Java的泛型中super 和 extends 有什么区别？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《架构设计中必不可少的泛型-Java泛型的定义与原理》

这道题想考察什么？

掌握PECS原则，灵活运用泛型

考察的知识点

泛型上下边界

考生应该如何回答

在Java的泛型语法中，可以使用super和extends关键字指定泛型的上下边界。

extends

`? extends T`为上界通配符，也就是说限制类型只能是T 或者 T 的派生类，比如我们存在代表水果的Fruit类，代表苹果的Apple类以及代表香蕉的Banana类。苹果与香蕉都是水果，因此：

```
class Fruit{}
class Apple extends Fruit{}
class Banana extends Fruit{}
```

那么下面我们使用List集合作为盘子来装水果：

```
List<? extends Fruit> plates = new ArrayList<>();
```

但是当我们希望往plates中放入苹果或者香蕉时会发现，plates中无法放入任何元素，只能从plates中取出元素。

```
List<? extends Fruit> plates = new ArrayList<>();
plates.add(new Apple());  //Error
plates.add(new Banana()); //Error

Fruit fruit1 = paltes.get();
Object fruit1 = paltes.get();
Apple fruit1 = paltes.get(); //Error
```

其实原因在于，编译器只知道List中是Fruit或者其派生类，但是具体类型无从得知，可能是香蕉也可能是苹果甚至其他水果类。所以在add时，编译器无法判断你给的类型是不是能够和容器类型匹配，因此对于上界，不能往里存，只能往外取。

super

`? super T` 为通配符下界，也就是说限制类型只能是T 或者T的超类。

```
List<? super Fruit> plates = new ArrayList<>();
Fruit fruit = plates.get(0);//Error
Apple apple = plates.get(0);//Error
Object object = plates.get(0);

plates.add(new Apple());
plates.add(new Banana());
```

下界<? super T>不影响往里存，但往外取只能放在Object对象里。因为List<? super Fruit> 代表该容器元素是Fruit或者Fruit的超类。向容器中存储数据，只需要数据类型是Fruit的派生类即可，因为苹果是水果Fruit，香蕉也是水果Fruit。但是取数据时，无法得知取出来的数据到底时什么类型，所以只能使用Object来表示。

PECS原则

PECS原则即Producer Extends Consumer Super ，生产使用extends，消费使用super。结合上下界的特点可知：

- 经常读取数据，使用Extends；
- 经常加入数据，使用Super；

3.4 注解是什么？有哪些使用场景？（滴滴）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《架构设计中必不可少的泛型-Java泛型的定义与原理》

这道题想考察什么？

Java基础，高级语言特性

考察的知识点

注解与其应用场景

考生如何回答

Java 注解（Annotation）又称 Java 标注，是 JDK5.0 引入的一种注释机制。 注解是元数据的一种形式，提供有关于程序但不属于程序本身的数据。注解本身没有特殊意义，对它们注解的代码的操作没有直接影响。

按照**@Retention** 元注解定义的注解保留级，注解可以一般常见于以下场景使用：

SOURCE

`RetentionPolicy.SOURCE` ，作用于源码级别的注解，在类中使用`SOURCE`级别的注解，其编译之后的class中会被丢弃。可提供给Lint 检查、APT等场景使用。

Lint

在Android开发中，`support-annotations`与`androidx.annotation`中均有提供`@IntDef`注解，此注解的定义如下：

```
@Retention(SOURCE)  //源码级别注解
@Target({ANNOTATION_TYPE})
public @interface IntDef {
    int[] value() default {};

    boolean flag() default false;

    boolean open() default false;
}
```

> Java中Enum(枚举)的实质是特殊单例的静态成员变量，在运行期所有枚举类作为单例，全部加载到内存中。比常量多5到10倍的内存占用。

此注解的意义在于能够取代枚举，实现如方法入参限制。

如：我们定义方法`test`，此方法接收参数`teacher`需要在：**Lance**、**Alvin**中选择一个。如果使用枚举能够实现为:

```
public enum Teacher{
    LANCE,ALVIN
}

public void test(Teacher teacher) {

}
```

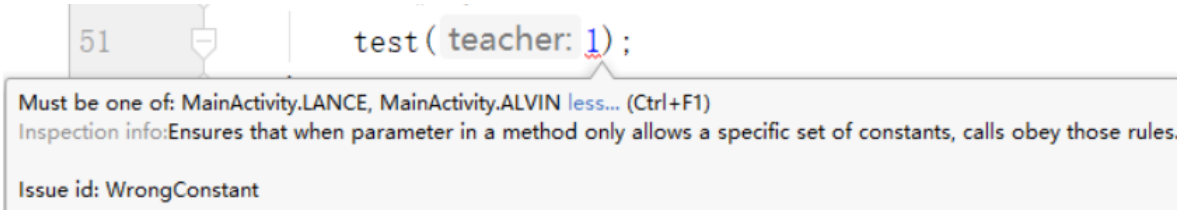
而现在为了进行内存优化，我们现在不再使用枚举，则方法定义为：

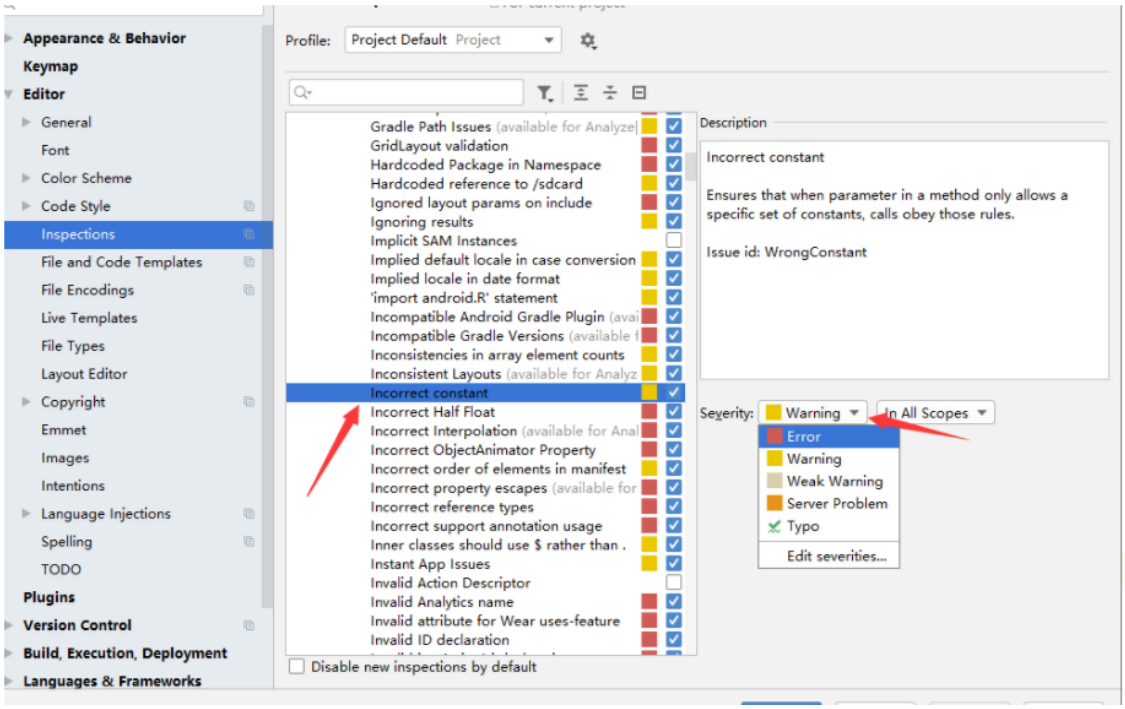
```
public static final int LANCE = 1;
public static final int ALVIN = 2;

public void test(int teacher) {

}
```

然而此时，调用`test`方法由于采用基本数据类型int，将无法进行类型限定。此时使用@IntDef增加自定义注解：

```
public static final int LANCE = 1;
public static final int ALVIN = 2;

@IntDef(value = {LANCE, ALVIN}) //限定为LANCE，ALVIN
@Target(ElementType.PARAMETER)  //作用于参数的注解
@Retention(RetentionPolicy.SOURCE) //源码级别注解
public @interface Teacher {
}

public void test(@Teacher int teacher) {

}
```

此时，我们再去调用`test`方法，如果传递的参数不是`LANCE`或者`ALVIN`则会显示 **Inspection** 警告(编译不会报错)。



> 可以修改此类语法检查级别：
>
> 
>

APT注解处理器

SOURCE另一种更常见的应用场景是结合APT使用。APT全称为："Anotation Processor Tools"，意为注解处理器。顾名思义，其用于处理注解。编写好的Java源文件，需要经过`javac`的编译，翻译为虚拟机能够加载解析的字节码Class文件。注解处理器是 javac 自带的一个工具，用来在编译时期扫描处理注解信息。你可以为某些注解注册自己的注解处理器。 注册的注解处理器由`javac`调起，并将注解信息传递给注解处理器进行处理。

> 注解处理器是对注解应用最为广泛的场景。在Glide、EventBus3、Butterknifer、Tinker、ARouter等等常用框架中都有注解处理器的身影。但是你可能会发现，这些框架中对注解的定义并不是`SOURCE`级别，更多的是`CLASS`级别，其实：**CLASS包含了SOURCE，RUNTIME包含SOURCE、CLASS。**所以CLASS是包含了SOURCE的场景，RUNTIME则包含了所有保留级的注解

> 使用场景。所以对于APT来说，不管使用何种保留时都可以。

CLASS

定义为`CLASS`的注解，会保留在class文件中，但是会被虚拟机忽略(即无法在运行期反射获取注解)。此时完全符合此种注解的应用场景为字节码操作。如：AspectJ、热修复Roubust等框架。

> 在Android开发中，保留在class，但是会在dex被抛弃

RUNTIME

注解保留至运行期，意味着我们能够在运行期间结合反射技术获取注解中的所有信息。如Retofit，借助反射获取获取用户定义在注解中的请求配置信息，基于获取的这些请求配置完成对Request请求的构建。