面向对象设计五大原则（单一功能、开闭原则、里氏替换、接口隔离以及依赖反转）是现代软件设计，开发和维护过程中必须考虑的，而且积极重要的原则。这其中第三大原则里氏替换原则（Liskov Substitution principle）是对[子类型](https://zh.wikipedia.org/wiki/%25E5%25AD%2590%25E7%25B1%25BB%25E5%259E%258B)的特别定义，由芭芭拉·利斯科夫（Barbara Liskov）在1987年在一次会议演说中首先提出。

今天我们结合里氏替换原则，来具体讨论一下其在Java，Scala这两大语言的泛型编程和设计中起到的举足轻重的作用。

**概念引入**

在此之前，有必要让我们重温一下几个概念，里氏替换原则和变型(Variance).

里氏替换原则说，如果类型A是类型B的子类，这里我们用A <: B来表示, 那么基类B的对象可以被子类A的对象替换而不改变任何程序的期望特性。

变型包含三个概念，协变(Covariance)，逆变(Contravariance)和不变(Invariance)

如果A <: B，也就是说A是B的子类, 而且类T[A] <: T[B]，那么类T就是协变的。譬如在Java中， Interger是Number 的子类，同时Integer[] 数组是Number[]数组的子类，所以Java中数组是协变的；而如果T[B] :< T[A], 那么类型T就是逆变的；如果T[A] 和 T[B] 没有关系，那么T就是不变的。

为了符合里氏替换原则，有几条规则在现代的面向对象编程语言设计必须遵守，这些规定分为两类，一类是合同原则，合同原则讨论了子类和父类之间必须遵守的合同和限制, 第二类，也是我们今天要讨论的，变型原则；此原则对于函数的签名上了三个规则：

1. 函数的输入参数是逆变的
2. 函数的返回参数是协变的
3. 函数的子类函数不可以抛出新的异常，除非被抛出的异常是基类函数抛出异常的子类

那么变型原则和我们的泛型编程有什么关系？那就先从Java泛型来进行分析。

**Java泛型**

之前提到，在Java语言里面数组Array是协变的，协变就意味着我们可以很好的利用面向对象的参数多态(parametric polymorphism), 但是同时我们发现，协变的Array会给我造成不少麻烦。譬如看如下程序：

Integer[] ints = {1,3,4,6,2};

Object[] objs = ints;

objs[0] = "1";

objs[1] = true;

objs[2] = new Object();

这段Java代码是完全可以编译成功，但是一旦这样代码进入运行阶段所能造成的问题是难以估量的。为什么，因为这样的协变数组是类型不安全的。

既然Java的Array已经是协变了，已经没有回头路了，我们怎么可以既可以使用参数化类型(parameterized types)，又能达到类型安全呢？于是Java 5 给了我们泛型Generics。在引入泛型以后，Collection包里面的数据类型就都可以使用泛型达到类型安全，这样我们的写代码时候就不用担心会有人在List，Set，Queue，Map里面乱放东西了，从此高枕无忧。

但是，你是不是感觉少了点什么？少了什么呢？对的，就是多态，OOP最重要的是什么，是多态。由于Java参数化类型是不变的，譬如List<String> 和 List<Object>没有任何关系，即使String是Object的子类。这里所说的多态不是子类继承父类的多态，而是参数多态（parametric polymorphism）。在Java中有参数多态吗？有，参数多态是通过泛型的上下界和通配符来实现的。

**Java泛型的上下界和通配符**

上界通常我们使用关键词extends, 譬如

public void findNumber(List<? extends Number> numbers)

假设我们有一个List<Integer> integers，由于Integer:<Number，这样我们就可以把integers这个list传给上述方法，也可以放List<Long>等，实现了参数多态。你会注意到，上界其实就是我们之前所提到的协变。

下界我们使用关键词super，譬如

public void addToIntegers(List<? super Integer> integers)

在这里，我们可以把List<Number>做传参，因为Number :> Integer,这样我们同样实现了参数多态，同样我们可以看到定义下界给予了List逆变的功能。

这时候你可能会问，那这样，我们不就是会遇到和Array一样的类型安全问题了吗？其实不会，Java在编译阶段就会给予错误提示。譬如下面代码

public void findNumber(List<? extends Number> numbers) {

Integer i = 1;

numbers.add(i); // compilation error

numbers.add(null);

numbers.forEach(System.out::println);

}

第三行代码，当我们要numbers加一个i元素时，或者说，当我们要mutate这个List的时候，编译出错。因为编译器并不能确定，传入的List是什么具体类型，可以是Integer，可以是Number，也可以是Long。这样就防止类型安全问题发生。但是我们看到当我们插入null时，编译成功了，为什么？因为null类型在java的类型系统里面是最低的，null是任何类的子类。

同样，当我们使用super时，为了达到类型安全，编译器会给我们做什么限制呢，如下代码所示：

public void addToIntegers(List<? super Integer> integers){

Integer i = 1;

integers.add(i);

for(Object obj : integers) {...}

for(Integer obj : integers) {...} // compilation error

}

当我们想按照Integer取出元素的时候，编译错误。此时编译器并不确定传入的List是具体那个类型，所以会自动按照最安全类型去Cast，我们这边定义了下界，而没有定义上界，（Java里面不能同时定义上下界,Scala可以）所以编译器直接按照最高对象Object返回给我们，这就是为什么第五行不能编译通过。

总而言之，当我们定义上界的时候，我们只能从List里面取东西，也就是Get；当我们定义下界的时候，我们只能往List里面加东西，也就是put，当我们取的时候，我们获得的类型是Object。

这就是传说中的**Get And Put Principle**，这个原则在Java泛型编程的时候是必须要注意和遵守的。但是我认为这并不是一个原则，而是一个方法让我们去克服Java泛型设计上的缺陷。

仔细看一下，就会发现一些不够优雅的地方。其一，根据里氏替换原则，方法的传入参数应该是逆变的，而在第一个例子，我们看到，Java并不阻止我们把协变类型作为传入参数；其二，方法的返回参数应该是协变的，同样，Java也不会阻止我们把逆变类型作为方法的返回参数。

同样我们看一下当我们创建自己的泛型,把协变类型作为方法的传入参数时候，Java编译器同样也没有阻止我们。

class BirdContainer<T extends Bird> {

private T bird;

public T process(T bird){

//...

return bird;

}

}

**Scala 参数化类型**

上述提到的Java泛型的问题，在Scala的参数化类型中有了很好的提高。在此值得提一下，Scala中所谓的的参数化类型即Java中的泛型。

首先，数组在Scala中是不变的，在编译阶段，Scala编译器会给出错误提示，如下代码所示：

scala> var ints = Array(1,2,3)

scala> var objs:Array[AnyVal] = ints

<console>:12: error: type mismatch;

found : Array[Int]

required: Array[AnyVal]

Note: Int <: AnyVal, but class Array is invariant in type T.

当我们尝试吧一个Int数据复制给一个AnyVal类型数据时候，编译器告诉我们说Int 是AnyVal的子类，但是Array类是不变的，所以编译失败，这样就避免我们在Java会遇到的问题。

而至于List，在Scala中是协变，又由于List是immutable不可变的，由此达到了类型安全。

那么通过协变，不可变来达到类型安全的List是怎么一个行为呢？我们先创建一个Int类型的listInt，然后在这个listInt插入一个char类型的，这时候注意到，插入操作和Java的List不同之处在于这里返回一个新的List，而且这个list的类型因插入的数据类型变化而变化。譬如这里我们插入一个char类型，返回的是一个AnyVal的List，我们且命名listAnyVal。因为list是不可变的，所以，原来的那个listInt并没有被修改。很好的在协变的条件下保证List的类型安全。我们继续在listAnyVal插入一个String类型的数据，同样这时候返回一个新的List，我们发现这个返回的List的数据类型是Any。

val listInt : List[Int] = List(1,2,3)

val listAnyVal : List[AnyVal] = listInt.::('a’)

val listAny : List[Any] = listAnyVal.::("string")

仔细检查list查入操作的返回类型会发现，AnyVal是Int和Char类型共有的超类，而Any是AnyVal和String类型所共有的超类。那么Scala编译器是怎么来知道改怎么返回类型的呢？

在深入了解之前，需要先了解一下Scala相对于Java所不同的的申明端变型(declaration-site variance)

**Scala申明端变型**

这是Scala相对于Java的泛型很大的提高之处。

Scala支持Java一样的使用端变型(use-site variance)，只要把<? extends T> 改成 [\_ <: T]，他同时支持申明端变型。顾名思义，类型的协变特性，在类型的申明阶段就已经定义了。你只需要把+符放在类型参数前面，申明此类型是协变类型参数，或者使用-符合，申明是逆变类型参数；如果什么都不放，就和Java一样，是不变的类型化参数。

譬如如下的Scala的List

sealed abstract class List[+A] extends AbstractSeq[A] …

我们看到，这样，List就是协变了。

如果我们需要定义和使用自己的协变类型，可以先在定义阶段申明其协变类型，然后直接使用：

class CovariantFoo[+T] { ... }

val covariantFoo = new CovariantFoo[Int]()

而同样的操作在Java需要先定义其泛型，然后在使用的时候申明其协变特性：

public class Foo<T> { ... }

Foo<? extends Integer> covariantFoo = new Foo<Integer>();

那么差别在哪里？差别在于，申明端的协变把设计的任务给了API设计者的一边，而使用端的协变，把设计重任放到了API使用者的那边，当然如果使用者可以很好的理解变型，做很好的设计的话，两者一样好。

但是，就像Scala的创造者Martin Odersky在Programming in Scala书里提到，变型并不是一个简单的任务，使用者通常会用错，他们也通常会认为通配符和泛型太过于复杂。而在申明端协变逆变的情况下，你把意图告诉给编译器，而编译器就会做很好的检查并确认是否出错。

那么Scala编译器具体做了什么呢？在这方面，Scala编译器做到了文章一开始提到的变型原则三个要求，其中第三个不在泛型和参数化类型讨论范围，我们暂且不讨论。在此我们主要讨论前两个。

为了检查一下scala编译器到底是不是这么工作的，我们来看一下如下一段代码:

trait Container[+T] {

def isEmpty : Boolean

def head : T

def tail : Container[T]

def prepend (elem : T) : Container[T]

}

为了不和List混淆，我们且命名之为Container，其功能和List类似，我们使用+符号，把他定义为协变类型。他有四个方法，isEmpty检查是否为空，head为当前的T，tail为连在head上的Container，最后一个方法是插入一个T类型，返回一个Container。但是我们会看到，此时编译器报错了，错误为：Covariant type T occurs in the contra variant position in type T of value elem。 基本就是说，在类型T的elem位置上面，编译器期望是一个逆变类型，而当前T类型是协变的。

这就我们上述的第一原则，由于elem这个位置是方法的传入参数，传入参数必须是逆变的。同样道理，在一个方法的返回位置，你如果放一个逆变类型，编译器也会同样报错。

是的，Java编译器没有做到的，Scala编译器做到了。

那么我们怎么在Scala来实现这个prepend方法，由不失去协变特性呢？像Java一样，Scala也可以使用上下界。

trait Container[+T] {

def isEmpty : Boolean

def head : T

def tail : Container[T]

def prepend [U >: T] (elem : U) : Container[U]

}

在此，prepend方法接受一个参数化类型，这个类型是T的超类，也就是说我们定义了一个下限。而根据前面Java的例子我们知道，下限是逆变，方法传入参数逆变是符合里氏替换原则的，所以编译通过。这时候此方法接受一个T类型的超类，而且返回超类的参数化类型，即Container[U]。

借此机会，结合List的定义和List的插入方法::的实现，我们来理解一下之前看到的，为什么返回类型都是原来List类型，和插入类型的共有超类，如下所示：

sealed abstract class List[+A] extends AbstractSeq[A] …

def ::[B >: A] (x: B): List[B] =

new scala.collection.immutable.::(x, this)

假设，有一个List[Int]，此时A就是Int类型，这时，我们对此List做一个插入操作，插入一个Char类型，注意，这个时候，Char并不足以替代B，因为插入方法申明说B必须是A的超类，也就是Int的超类，返回也是返回一个List[B]，那么这是编译器就会在类型系统中找到Char和Int最低共有超类，那就是AnyVal。也就为什么，我们最终会得到一个List[AnyVal]，值得注意的是，返回的list同样是一个immutable的list。

至此，我们讨论的变型都是在面向对象范围内的。而Scala作为一个函数式编程语言，同时也是一个纯粹的面向对象语言，函数也是一个对象，既然是对象，就存在父类和子类。那么接下来，我们来看一下变型在函数作为对象中的扩展。

**Scala函数类型与变型**

熟悉Java的Lambda编程的都知道，lambda只不过是一个对于Functional Interface的实现，是一个匿名的方程，类似的在Scala里面也有lambda表达式，与Java不同的是Scala的lambda表达式用符号=>表示，每一个表达式是一个trait FunctionN的实现，根据表达式的参数个数不同而不同；譬如函数(x: Int) => x + 1，其类型是Int => Int，是如下代码的缩写：

new Function1[Int, Int] {

def apply(x: Int): Int = x + 1

}

查看一下源代码，Function1的定义如下：

trait Function1[-S, +T] extends scala.AnyRef { def apply(x: S): T }

可以看到Function1是一个参数化类型的trait，继承超类AnyRef，其定义最大化的遵守了里氏替换原则，为什么？他有两个参数，S和T，根据其符号，我们知道S是逆变，T是协变的。Function1只有一个抽象方法apply，apply传入参数是S，返回类型是T。

那么function1作为一个对象，其子类型在高阶函数的情况下是怎么一个行为呢？

假设我们有一个高阶函数func

def func(f:Null => AnyRef) = 1

此高阶函数传入参数是一个函数f，这个函数的类型是 Null => AnyRef，其返回一个Int类型。根据里氏替换原则，任何属于类型Null => AnyRef的子类型函数都可以传给高阶函数func。

譬如类型是 Null => List[Int]的f1:

scala> val f1 = (n:Null) => List(1)

f1: Null => List[Int] = $$Lambda$1142/943079062@35c95869

scala> var i1 = func(f1)

i1: Int = 1

为符合里氏替换原则，我们需要传入参数逆变，返回参数协变，对此我们对f1类型表达式左右参数相对于类型 Null => AnyRef分别做一下比较。

传入参数都是Null， Null可以是其本身的父类，所以传入参数逆变，符合规则；返回参数List是AnyRef的子类，即List:<AnyRef，所以返回参数协变，也符合规则，所以上述代码能编译通过。

按照同样的推理，f2: String => String也能被传给高阶方程，因为String :> Null, 而且String :< AnyRef,所以类型 String => String :< Null => AnyRef, 如下面代码所示：

scala> val f2 = (n:String) => "hello world"

f2: String => String = $$Lambda$1143/1534507153@611a91df

scala> val i2 = func(f2)

i2: Int = 1

但是，函数f3:Int => String传给func的时候，编译出错：

scala> var f3 = (n:Int) => "hello world"

f3: Int => String = $$Lambda$1145/555754759@62878d7c

scala> val i3 = func(f3)

<console>:13: error: type mismatch;

found : Int => String

required: Null => AnyRef

val i3 = func(f3)

这里编译器报了类型匹配错误，因为在Scala类型系统里，Int并不是Null的超类型，传入参数逆变条件不符合，因此里氏替换原则变型要求不能达到，编译出错。这在很大程度上帮助开发人员能够在更早的阶段发现问题。

**小结**

我们从里氏替换原则出发；分析了Java数组的变型特性，类型不安全特征，到Java的泛型编程，泛型的变型，以及Java泛型编程中的一些注意点和其不优雅之处；然后我们分析了Scala，作为一个函数编程语言和面向对象编程语言的结合体，其数组类型的安全特性，其相对于Java在参数化类型，泛型编程，通过完全遵守里氏替换原则的条件下，在设计和应用方面的显著提高；最后，我们再次结合里氏替换原则，分析了Scala的函数作为对象和在高阶函数使用的时候，其类型变化和变型。