Kotlin的泛型

简介

与java一样，kotlin也支持泛型，用法和java泛型差别不大，kotlin特色是型变支持。

基本用法：

定义类：

跟java相同，定义在类后面的尖括号：

```
open class Basket<T>{

}
```

定义方法：

定义在fun 关键字和 方法名之间。

```
//java
  public <S> void  testFunction(S s){
        //todo
    }
//kotlin
  fun <S> testFunction(s:S){
        //todo
    }
```

以声明一个水果篮为例，在构造方法中声明了泛型，里面提供一个list支持set和get操作：

```
open class Basket<T> {

    var content: T? = null

    fun set(fruit: T) {
        content = fruit
    }

    fun get(): T? {
        return content
    }
}
```

定义一个水果类：

```
open class Fruit {
    open fun desc() {
        println("它是水果")
    }
}
```

使用：

```
fun main(args: Array<String>) {
    val fruit1 = Fruit()
    val basket = Basket(fruit1)
}
```

与java的尖括号语法不同，如果我在类的构造方法中指定了类型的话，在kotlin中可省略不写，其可帮我们自动推断。

从泛型类派生子类：

我现在写一个小水果篮子继承自果篮类：

```
class SmallBasket : Basket<Fruit>()
```

注意点：

与java不同的是，无论是通过显示指定还是让系统推断。kotlin要求始终为泛型参数明确地指定类型，所以上面参数我指定为水果类。 而在java中，以下两种都是允许的：

```
public  class SmallBasketJ extends BasketJ<String> {
   }
```

或

```
public  class SmallBasketJ extends BasketJ{

   }
```

型变：

回顾一下，java的泛型是不支持型变的，如何理解这句话呢？ 首先这行代码是没有问题的：

```
String string = new String("sss");
Object object = string;
```

因为string是Object的子类，子类可以协变为父类，但是在泛型中：

```
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // 不允许
```

因此，Java 禁止这样以保证运行时的安全，因为如果上面的代码允许被编译通过那么：

```
//这里我们把一个整数放入一个字符串列表
objs.add(1);
//报 ClassCastException
String s = strs.get(0);
```

所以泛型不支持型变的设计保证了其是“类型安全的”，但是通过通配符，可以让他们有“型变”的能力，具体为：

java通配符上限：

```
<? extends E>
```

表示此方法接受 E 或者 E 的 一些子类型对象的集合，而不只是 E 自身。 这意味着我们可以安全地从其中（该集合中的元素是 E 的子类的实例）读取 E，但不能写入， 因为我们不知道什么对象符合那个未知的 E 的子类型。 反过来，该限制可以让Collection<String>表示为Collection<? extends Object>的子类型。 简而言之，带 extends 限定（上界）的通配符类型使得类型是协变的（covariant）。

```
interface Collection<E> …… {
  void addAll(Collection<? extends E> items);
}
```

我们可以往Collection中添加E类型或者它的任意子类，这是泛型的协变。 我们用一个图来表示就是“正三角漏斗”，顶部就是我们的E：

意味着从泛型中取出(out)对象是安全的(一定是我们的E类型)，但传入对象并不知道具体类型（可能是E或者它的子类）。

java通配符下限：

```
<? super T>
```

与通配符上限相反，限制传入的参数下限是T(即T或者它的父类)当我们用下限修饰符去修饰的话，将对象传给泛型对象是安全的，如Collections.copy方法：

```
    public static <T> void copy(List<? super T> dest, List<? extends T>
src) {
        。。。。。。
    }
```

第一个参数使用通配符下限，限制了目标list只能是T或者T它的父类，而第二个参数拷贝源限制了参数只能是T或者T的子类，这样就保证了类型的合法。

```
        List<Apple> source = Arrays.asList(new Apple());
        //Object 也是最终父类 也ok
//        List<Object> destination = Arrays.asList(new Object());
        List<Fruit> destination = Arrays.asList(new Fruit());
        Collections.copy(destination,source);
```

我们上面的例子，我们可以完成这样的业务类型：

把包含苹果的List放入原有包含Fruit的List(或者Object的List).

通配符上限保证了传入参数的安全，如下“倒三角漏斗”所示：

我们可以往漏斗中放入E类型和任何它的父类,这就是泛型的逆变，意味着向其中传入(in)对象是安全的，但就不能保证取出来的参数的类型（可能是T，也可能是它的父类）

结论速记：

- 通配符上限-extends-正三角-取出安全-out
- 通配符下限-super-倒三角-存入安全-in

Kotlin型变：

无论java的通配符上限还是下限，都多少有缺陷，要么存不安全，要么取不安全，而在kotlin中，就解决了这个问题，让out：“纯输出”，让in“纯输入”。

在此之前，我们借助上面java的通配符的 (in) 和 (out) 的操作来理解一个概念： 我们称只能从中读取的对象为生产者，并称那些你只能写入的对象为消费者。

Kotlin声明处型变：

out： （协变注解）生产者：

一般原则是：当一个类 C 的类型参数 T 被声明为 out 时，它就只能出现在 C 的成员的输出-位置，但回报是 C<Base> 可以安全地作为 C<Derived>的超类。 简而言之，他们说类 C 是在参数 T 上是协变的，或者说 T 是一个协变的类型参数。 你可以认为 C 是 T 的生产者，而不是 T 的消费者。

还是水果篮和水果的例子，我定义了一个水果篮类，构造方法传入了T的示例，仅提供了get方法：

```
class Basket2<out T>(private val content: T) {

    fun get(): T {
        return content
    }
}
```

那么我们之前在java中不能实现的泛型型变，现在就是ok的了：

```
var basketFruit: Basket2<Fruit> = Basket2(Fruit())
var basketApple: Basket2<Apple> = Basket2(Apple())
 //ok的 符合协变的规则
basketFruit = basketApple
```

in：（逆变注解）消费者：。它使得一个类型参数逆变：只可以被消费而不可以被生产，我们以Compareble为例：

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    // 我们可以将 x 赋给类型为 Comparable <Double> 的变量
    val y: Comparable<Double> = x // OK！因为 y可以接受Double或者它的任意父类，即“逆变了”
}
```

我们再回到篮子和水果的例子，我定义一个水果篮类，用in修饰：

```
class Basket3<in T> {

    fun set(param: T) {
        println(param)
    }
}
```

那么我们现在可以逆变了：

```
var basket3Apple = Basket3<Apple>()
var basket3Fruit = Basket3<Fruit>()
//ok的 符合逆变
basket3Apple = basket3Fruit
```

结论：

- 如果泛型T（或其他字母）只出现在该类的返回值中声明，那么该泛型形参即可使用out修饰
- 如果泛型T（或其他字母）只出现在该类的方法的形参声明中，那么泛型形参可使用int修饰

Kotlin使用处型变：类型投影

声明时型变虽然方便，但它有一个限制：要么该类的所有方法都只用泛型声明返回值类型（此时可用out声明型变）：要么所有方法都只用泛型声明形参类型（此时可用in声明型变）。如果一个类中有 的方法使用泛型声明返回值类型，有的方法使泛型声明形参类型，那么该类就不能使用声明处型变。典型的例子就是Kotlin 的Array类，它无法使用声明处型变，该类在T 上既不能协变也是不能逆变的。

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { …… }
    fun set(index: Int, value: T) { …… }
}
```

假如写下了如下方法，把一个数组复制到另外一个数组：

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

尝试着按照这种方式调用：

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
//   ^ 其类型为 Array<Int> 但此处期望 Array<Any>
```

再次回到老问题：T是不型变的，因此 Array<Int>和Array<Any>都不是彼此的子类，如果在from参数中要求Sting，我实际却传入了int就会报ClassCastException，那么我们想避免这样的事情发生我们可以这么做：

```
fun copy(from: Array<out Any>, to: Array<Any>) { ..... }
```

我们说from不仅仅是一个数组，而且是一个受限制(投影)的数组，我们只可以调用返回类型为T的方法，上面我们就只能调用get（）。这便是我们的使用处型变的方法。

再例如： 我定义一个类型为Number的Array：

```
var numArr: Array<Number> = arrayOf(1,2,3,4,5)
numArr.set(0,2)//1.正常
var intArr:Array<Int> = arrayOf(1,2,3)//2.正常
numArr = intArr//3.报错 不支持声明处型变
```

那么我现在在Number上面加上一个out

```
var numArr: Array<out Number> = arrayOf(1,2,3,4,5)
numArr.set(0,2)//1.报错
var intArr:Array<Int> = arrayOf(1,2,3)//2.正常
numArr = intArr//3.正常
```

我用out修饰了Number，意味着它可以接受协变，代价就是只能出不能添加。

上面的例子中 out 的定义就叫类型投影

依然以Array为例：

我写一个填充到数组的方法，指定类型为String

```
fun fill(dest: Array<String>,value:String){
    if(dest.size>0){
        dest[0] = value
    }
}
```

此时我这么调用就会报错：

```
var arr1:Array<CharSequence> = arrayOf("a","b",StringBuilder("test"))
fill(arr1,"test") //报错
println(arr1.contentToString())
```

此时我在声明处添加 in ，表示可以接受String的任何父类，就可以编译通过了：

```
fun fill(dest: Array<in String>,value:String){
。。。。。。
}
```

再例如，刚刚上面的的例子：

```
var intArr:Array<Int> = arrayOf(1,2,3)
var number:Array<Number> = arrayOf(1,2,3)
intArr = number //报错：不支持声明时逆变
```

我们加上这个限制以后，就能逆变了：

```
var intArr:Array<in Int> = arrayOf(1,2,3)
```

星投影

表示不知道类型实参的任何信息

```
var list:Array<*> =  arrayOf("test","kotiln",1,2)
list[0]="1"//报错 无法被写入
```

所以：

- 星号投影不能写入，只能读取
- <*>等价于java中的<?>

设定类型形参上限

单个形参

kotlin不仅允许在使用通配符时设定形参上限，而且可以在定义类型形参时设定上限，用于表示给该类型的实际类型要么是该上限类型，要么是它的子类。

回顾一下上面篮子的例子

```
class Basket2<out T>(private val content: T) {

    fun get(): T {
        return content
    }
}
```

我们改一改，让它只能放水果：

```
class Basket2<T:Fruit>(private val content: T) {

    fun get(): T {
        return content
    }
}
```

乍一看，它们好像没有什么区别。。。。

```
var basket2Fruit: Basket2<Fruit> = Basket2(Fruit())
var basket2Apple: Basket2<Apple> = Basket2(Apple())
```

以上两行代码在两种修饰符下都可以执行，但是，我们知道，out是可以让泛型协变的，即：

```
basket2Fruit = basket2Apple //out ok ，设定形参上限报错
```

用out是ok的，代价是只能作为生产者输出了，而用形参上限，我们却可以跟它提供一个set方法：

```
class Basket2<T : Fruit>(private var content: T) {

    fun set(fruit: T) {
        content = fruit
    }

    fun get(): T {
        return content
    }
}
```

这样我们就能保证这个篮子中只能放入Fruit和它的子类了，也能从里面取出Fruit，但是此时，它不能型变。

多个形参

kotlin允许为类型设定多个形参上限，在尖括号外用 where语句： 先定义两个接口或者父类：

```
interface Eatable {
    fun eat()
}

interface Color
```

如果想限定参数的必须实现上面两个接口可以这么写：

```
class Basket<T> where T : Eatable, T : Color {
。。。。。。
}
```

- 对于泛型的使用如果我们没有型变需要，有存有取，可以优先使用形参上限来限制参数。

具体化类型参数

kotlin允许在内联函数(inline修饰)使用refied修饰泛型参数，这样可将泛型参数变成一个具体的类型参数，很适用于我们需要用Class做参数的情形： 例如，我们要从某个List找某个指定类型的元素：

```
fun<T> findData(clazz:Class<T>):T?{
    .....
}
//使用
findData(Integer:class.java)
```

那么这么写就能省略class参数了：

```
fun inline <refied T> findData():T{
    .....
}
//使用
findData<Int>()
```

是不是优雅许多?

参考资料：

https://www.kotlincn.net/docs/reference/generics.html