Scala核心编程

-泛型、上下界、视图界定 、上下文界定

尚硅谷研究院

- 泛型

基本介绍

1) 如果我们要求函数的参数可以接受任意类型。可以使用泛型，这个类型可以代表任意的数据类型。
2) 例如 List，在创建 List 时，可以传入整型、字符串、浮点数等等任意类型。那是因为 List 在 类定义时引用了泛型。比如在Java中：**public interface** List<E> **extends** Collection<E>

- 泛型

Scala泛型应用案例1

要求:

1) 编写一个Message类
2) 可以构建Int类型的Message, String类型的Message.
3) 要求使用泛型来完成设计,(说明：不能使用Any)

```
object GenericUse {
  def main(args: Array[String]): Unit = {
    val mes1 = new StrMessage[String]("10")
    println(mes1.get)
    val mes2 = new IntMessage[Int](20)
    println(mes2.get)
  }}

// 在 Scala 定义泛型用[T]， s 为泛型的引用
abstract class Message[T](s: T) {
  def get: T = s
}
// 子类扩展的时候，约定了具体的类型
class StrMessage[String](msg: String) extends Message(msg
class IntMessage[Int](msg: Int) extends Message(msg)
```

- 泛型

Scala泛型应用案例2

➢ 要求

1) 请设计一个EnglishClass (英语班级类)，在创建EnglishClass的一个实例时，需要指定[班级开班季节(spring,autumn,summer,winter)、班级名称、班级类型]
2) 开班季节只能是指定的，班级名称为String, 班级类型是(字符串类型 "高级班", "初级班"..) 或者是 Int 类型(1, 2, 3 等)
3) 请使用泛型来完成本案例.

```
// Scala 枚举类型
object SeasonEm extends Enumeration {
  type SeasonEm = Value //自定义SeasonEm，是Value类型,这样才能使用
  val spring, summer, winter, autumn = Value
}
```

- 泛型

Scala泛型应用案例2

➢ 代码

```
object GenericUse2 {
  def main(args: Array[String]): Unit = {

    val class1 = new EnglishClass[SeasonEm, String, String](SeasonEm.spring, "001班", "高级班")
    println(class1.classSeason + " " + class1.className + " " + class1.classType)

    val class2 = new EnglishClass[SeasonEm, String, Int](SeasonEm.spring, "002班", 1)
    println(class2.classSeason + " " + class2.className + " " + class2.classType)
}}
// Scala 枚举类型
object SeasonEm extends Enumeration {
  type SeasonEm = Value //自定义SeasonEm，是Value类型,这样才能使用
  val spring, summer, winter, autumn = Value
}
// 定义一个泛型类
class EnglishClass[A, B, C](val classSeason: A, val className: B, val classType: C)
```

- 泛型

Scala泛型应用案例3

➢ 要求

1) 定义一个函数，可以获取各种类型的 List 的中间index的值
2) 使用泛型完成

```
def getMidEle[A](l: List[A])={
    l(l.length/2)
}
```

- 泛型

Scala泛型应用案例3

➢ 代码

```
object GenericUse3 {
  def main(args: Array[String]): Unit = {
    // 定义一个函数，可以获取各种类型的 List 的中间index的值
    val list1 = List("jack",100,"tom")
    val list2 = List(1.1,30,30,41)

    println(getMidEle(list1))

  }
  // 定义一个方法接收任意类型的 List 集合
  def getMidEle[A](l: List[A])={
    l(l.length/2)
  }
}
```

- 类型约束-上界(Upper Bounds)/下界(lower bounds)

上界(Upper Bounds)介绍和使用

➢ **java中上界**

在 Java 泛型里表示某个类型是 A 类型的子类型，使用 extends 关键字，这种形式叫 upper bounds(上限或上界)，语法如下：

```
<T extends A>
//或用通配符的形式：
<? extends A>
```

- 类型约束-上界(Upper Bounds)/下界(lower bounds)

上界(Upper Bounds)介绍和使用

➢ **scala中上界**

在 scala 里表示某个类型是 A 类型的子类型，也称上界或上限，使用 **<:** 关键字，语法如下：

```
[T <: A]
//或用通配符:
[_ <: A]
```

Scala 的那些奇怪的符号 上

● 类型约束-上界(Upper Bounds)/下界(lower bounds)

上界(Upper Bounds)介绍和使用

➢ **scala中上界应用案例-要求**

1) 编写一个通用的类，可以进行Int之间、Float之间、等实现了Comparable接口的值直接的比较.//java.lang.Integer
2) 分别使用**传统方法**和**上界的方式**来完成，体会上界使用的好处.

```
class CompareInt(n1: Int, n2: Int) {
  def greater = if(n1 > n2) n1 else n2
}
class CompareComm[T <: Comparable[T]](obj1: T, obj2: T) {
    def greater = if(obj1.compareTo(obj2) > 0) obj1 else obj2
}
//映射转换 Predef.scala
```

- 类型约束-上界(Upper Bounds)/下界(lower bounds)

上界(Upper Bounds)介绍和使用

➢ scala中上界应用案例-代码

```
object UpperBoundsDemo {
  def main(args: Array[String]): Unit = {
    //常规方式
    /*
    val compareInt = new CompareInt(-10, 2)
    println("res1=" + compareInt.greater)
    val compareFloat = new CompareFloat(-10.0f, -20.0f)
    println("res2=" + compareFloat.greater)*/
    /*val compareComm1 = new CompareComm(20, 30)
    println(compareComm1.greater)*/
    val compareComm2 = new CompareComm(Integer.valueOf(20), Integer.valueOf(30))
    println(compareComm2.greater)
    val compareComm3 =
      new CompareComm(java.lang.Float.valueOf(20.1f), java.lang.Float.valueOf(30.1f))
    println(compareComm3.greater)
    val compareComm4 = new CompareComm[java.lang.Float](201.9f, 30.1f)
    println(compareComm4.greater)
  }
}
/*class CompareInt(n1: Int, n2: Int) {
  def greater = if(n1 > n2) n1 else n2
}
```

- 类型约束-上界(Upper Bounds)/下界(lower bounds)

上界(Upper Bounds)介绍和使用

➢ scala中上界课程测试题(理解上界含义)

```
object LowerBoundsDemo {
  def main(args: Array[String]): Unit = {
    biophony(Seq(new Bird, new Bird)) //?
    biophony(Seq(new Animal, new Animal)) //?
    biophony(Seq(new Animal, new Bird)) //?
    biophony(Seq(new Earth, new Earth)) //?
  }
  def biophony[T <: Animal](things: Seq[T]) = things map (_.sound)
}
class Earth { //Earth 类
  def sound(){ //方法
    println("hello !")
  }}
class Animal extends Earth{
  override def sound() ={ //重写了Earth的方法sound()
    println("animal sound")
  }}
class Bird extends Animal{
  override def sound()={ //将Animal的方法重写
    print("bird sounds")
```

- 类型约束-上界(Upper Bounds)/下界(lower bounds)

下界(Lower Bounds)介绍和使用

➢ **Java中下界**

在 Java 泛型里表示某个类型是 A类型的父类型，使用 super 关键字

```
<T super A>
//或用通配符的形式：
<? super A>
```

- 类型约束-上界(Upper Bounds)/下界(lower bounds)

下界(Lower Bounds)介绍和使用

➢ **scala中下界**

在 scala 的下界或下限，使用 **>:** 关键字，语法如下：

```
[T >: A]
//或用通配符:
[_ >: A]
```

类型约束-上界(Upper Bounds)/下界(lower bounds)

下界(Lower Bounds)介绍和使用

- scala中下界应用实例

```
object LowerBoundsDemo {
  def main(args: Array[String]): Unit = {

    biophony(Seq(new Earth, new Earth)).map(_.sound())

    biophony(Seq(new Animal, new Animal)).map(_.sound())
    biophony(Seq(new Bird, new Bird)).map(_.sound())

    val res = biophony(Seq(new Bird))

    val res2 = biophony(Seq(new Object))
    val res3 = biophony(Seq(new Moon))
    println("\nres2=" + res2)
    println("\nres3=" + res2)

  }
  def biophony[T >: Animal](things: Seq[T]) = things
}
```

```
class Earth { //Earth 类
  def sound(){ //方法
    println("hello !")
  }
}
class Animal extends Earth{
  override def sound() ={ //重写了Earth的方法sound()
    println("animal sound")
  }
}
class Bird extends Animal{
  override def sound()={ //将Animal的方法重写
    print("bird sounds")
  }
}
class Moon {}
```

Scala 的那些奇怪的符号 上

- 类型约束-上界(Upper Bounds)/下界(lower bounds)

下界(Lower Bounds)介绍和使用

➢ scala中下界的使用小结

```
def biophony[T >: Animal](things: Seq[T]) = things
```

1) 对于下界，可以传入任意类型
2) 传入和Animal直系的，是Animal父类的还是父类处理，是Animal子类的按照Animal处理
3) 和Animal无关的，一律按照Object处理
4) 也就是下界，可以随便传，只是处理是方式不一样
5) 不能使用上界的思路来类推下界的含义

```
scala> biophony(Seq(new Bird, new Bird))
res2: Seq[Animal] = List(Bird@7486b455, Bird@660acfb)
```

- 类型约束-视图界定(View bounds)

视图界定基本介绍

<% 的意思是 “view bounds”(视界)，它比<:适用的范围更广，除了所有的子类型，还允许隐式转换类型。

def method [A <% B](arglist): R = ... 等价于:
def method [A](arglist)(implicit viewAB: A => B): R = ...
或等价于:
implicit def conver(a:A): B = …

<% 除了方法使用之外，class 声明类型参数时也可使用：
class A[T <% Int]

- 类型约束-视图界定(View bounds)

视图界定应用案例1

```
object ViewBoundsDemo {
  def main(args: Array[String]): Unit = {
   //方式1
    val compareComm1 = new CompareComm(20, 30)
    println(compareComm1.greater)
    //同时，也支持前面学习过的上界使用的各种方式,看后面代码
  }
}
class CompareComm[T <% Comparable[T]](obj1: T, obj2: T) {
  def greater = if(obj1.compareTo(obj2) > 0) obj1 else obj2
}
```

- 类型约束-视图界定(View bounds)

视图界定应用案例1

```
object ViewBoundsDemo {
  def main(args: Array[String]): Unit = {
    val compareComm1 = new CompareComm(20, 30) //
    println(compareComm1.greater)

    val compareComm2 = new CompareComm(Integer.valueOf(20), Integer.valueOf(30))
    println(compareComm2.greater)

    val compareComm4 = new CompareComm[java.lang.Float](201.9f, 30.1f)
    println(compareComm4.greater)
    //上面的小数比较，在视图界定的情况下，就可以这样写了
    val compareComm5 =
      new CompareComm(201.9f, 310.1f)
    println(compareComm5.greater)
  }
}

/**
  * <% 视图界定 view bounds
  *   会发生隐式转换
  */
class CompareComm[T <% Comparable[T]](obj1: T, obj2: T) {
```

- 类型约束-视图界定(View bounds)

视图界定应用案例2

说明: 使用视图界定的方式，比较两个Person对象的年龄大小。

```
val p1 = new Person("tom", 10)
val p2 = new Person("jack", 20)
val compareComm2 = new CompareComm2(p1, p2)
println(compareComm2.getter)



class Person(val name: String, val age: Int) extends Ordered[Person] {
  override def compare(that: Person): Int = this.age - that.age
  override def toString: String = this.name + "\t" + this.age}
class CompareComm2[T <% Ordered[T]](obj1: T, obj2: T) {
  def getter = if (obj1 > obj2) obj1 else obj2
  def geatter2 = if (obj1.compareTo(obj2) > 0) obj1 else obj2
}
```

- 类型约束-视图界定(View bounds)

视图界定应用案例3

说明: 自己写隐式转换结合视图界定的方式，比较两个Person对象的年龄大小。

```
// 隐式将Student -> Ordered[Person2]//放在object MyImplicit 中
 implicit def person22OrderedPerson2(person: Person2) = new Ordered[Person2]{
   override def compare(that: Person2): Int = person.age - that.age
 }

 val p1 = new Person2("tom", 110)
 val p2 = new Person2("jack", 20)
 import MyImplicit._
 val compareComm3 = new CompareComm2(p1, p2)
 println(compareComm3.geatter)

class Person2(val name: String, val age: Int)  {
  override def toString = this.name + "\t" + this.age
}
class CompareComm3[T <% Ordered[T]](obj1: T, obj2: T) {
  def geater = if (obj1 > obj2) obj1 else obj2
}
```

- 类型约束-上下文界定(Context bounds)

基本介绍

与 view bounds 一样 context bounds(上下文界定)也是**隐式参数的语法糖**。为语法上的方便， 引入了”上下文界定”这个概念

类型约束-上下文界定(Context bounds)

ordered和ordering区别.

上下文界定应用实例

要求：使用上下文界定+隐式参数的方式，比较两个Person对象的年龄大小

要求：使用Ordering实现比较

```
object ContextBoundsDemo {
  implicit val personComparetor = new Ordering[Person] {
    override def compare(p1: Person, p2: Person): Int =
      p1.age - p2.age
  }

  def main(args: Array[String]): Unit = {
    val p1 = new Person("mary", 30)
    val p2 = new Person("smith", 35)
    val compareComm4 = new CompareComm4(p1,p2)
    println(compareComm4.geatter)
    val compareComm5 = new CompareComm5(p1,p2)
    println(compareComm5.geatter)
    val compareComm6 = new CompareComm6(p1,p2)
    println(compareComm6.geatter)
  }}
```

```
//方式1
class CompareComm4[T: Ordering](obj1: T, obj2: T)(implicit compareto
    def geatter = if (comparetor.compare(obj1, obj2) > 0) obj1 else obj2
}
//方式2,将隐式参数放到方法内
class CompareComm5[T: Ordering](o1: T, o2: T) {
    def geatter = {
        def f1(implicit cmptor: Ordering[T]) = cmptor.compare(o1, o2)
        if (f1 > 0) o1 else o2
    }}
//方式3,使用implicitly语法糖，最简单(推荐使用)
class CompareComm6[T: Ordering](o1: T, o2: T) {
  def geatter = {
    //这句话就是会发生隐式转换，获取到隐式值 personComparetor
    val comparetor = implicitly[Ordering[T]]
    println("CompareComm6 comparetor" + comparetor.hashCode())
    if(comparetor.compare(o1, o2) > 0) o1 else o2
  }}
//一个普通的Person类
class Person(val name: String, val age: Int) {
  override def toString = this.name + "\t" + this.age
}
```

- 协变、逆变和不变

基本介绍

1) Scala的协变(+)，逆变(-)，协变covariant、逆变contravariant、不可变invariant

2) 对于一个带类型参数的类型，比如 List[T]，如果对A及其子类型B，满足 List[B]也符合List[A]的子类型，那么就称为***covariance(协变)***，如果 List[A]是 List[B]的子类型，即与原来的父子关系正相反，则称为***contravariance(逆变)***。如果一个类型支持协变或逆变，则称这个类型为variance(翻译为可变的或变型)，否则称为***invariance(不可变的)***

3) ***在Java里，泛型类型都是invariant，比如 List<String> 并不是 List<Object> 的子类型。而scala支持，可以在定义类型时声明(用加号表示为协变，减号表示逆变)，***
如: trait List[+T] // 在类型定义时声明为协变这样会把List[String]作为List[Any]的子类型。

- 协变、逆变和不变

应用实例

在这里引入关于这个符号的说明，在声明Scala的泛型类型时，***“+”表示协变，而“-”表示逆变***

C[+T]：如果A是B的子类，那么C[A]是C[B]的子类，称为协变

C[-T]：如果A是B的子类，那么C[B]是C[A]的子类，称为逆变

C[T]：无论A和B是什么关系，C[A]和C[B]没有从属关系。称为不变.

```
val t: Temp[Super] = new Temp[Sub]("hello world1")
class Temp3[A](title: String) { //Temp3[+A] //Temp[-A]
  override def toString: String = {
    title
  }}
//支持协变
class Super
class Sub extends Super
```

谢谢！ 欢迎收看！