# Scala

## Scala面试题

### 介绍scala

#### (★★★)scala 语言有什么特点，什么是函数式编程？有什么优点

一致性: 尽管Scala融合了静态类型系统、面向对象、函数式编程等语言特性，但却很少能看出融合的痕迹。Scala是我见到融合最多语言特性而又不显得杂乱的编程语言之一。

类型安全：Scala创始人是教授，他先带领创建了Java 5编译器，而后觉得Java有太多羁绊而发明了Scala。 Scala编译器和类型系统非常强大，它的目标是尽量把软件错误消灭在编写过程中。 Scala类型系统是图灵完备的，甚至可以在编译期间解决问题。

面向对象: Scala是面向对象的编程语言，所有的变量和方法都封装在对象中，可以把信息封装起来供外部使用。(第2章)

函数式编程：Scala同时又是函数式编程语言，函数可以独立存在，可以定义一个函数作为另一个函数的返回值，也可以接受函数作为函数的参数。这给组合函数带来了很大的便利。如何把面向对象编程形容成搭积木的话，函数式编程就像拼线条，更灵活和更有创意。(第3章)

异步编程: 由于函数式编程提倡变量不可变，使异步编程变得非常容易。同时Scala提供的Future(第5章)， 和akka类库(第9-11章)，使得异步编程变得非常容易。

基于JVM: Scala会被编译成为jvm bytecode,所以Scala能无缝集成已有的Java类库。你可以非常自然的使用已经存在的非常庞大且稳定的Java类库，比如小巧好用的apache.common.\*, 或者Java上的各种工具类库。

函数式编程的优点：

在函数式编程中，由于数据全部都是不可变的，所以没有并发编程的问题，是多线程安全的。可以有效降低程序运行中所产生的副作用，对于快速迭代的项目来说，函数式编程可以实现函数与函数之间的热切换而不用担心数据的问题，因为它是以函数作为最小单位的，只要函数与函数之间的关系正确即可保证结果的正确性。

函数式编程的表达方式更加符合人类日常生活中的语法，代码可读性更强。实现同样的功能函数式编程所需要的代码比面向对象编程要少很多，代码更加简洁明晰。函数式编程广泛运用于科学研究中，因为在科研中对于代码的工程化要求比较低，写起来更加简单，所以使用函数式编程开发的速度比用面向对象要高很多，如果是对开发速度要求较高但是对运行资源要求较低同时对速度要求较低的场景下使用函数式会更加高效。

参考网站：<https://www.cnblogs.com/zhenw/p/6160931.html>

#### (★)编译好的scala程序，运行时还需要scala环境吗

并不需要，编译好的scala文件打包好后，①可以使用java方式运行打包好的scala程序，在控制台上直接使用java指令进行运行，注意的是，需要使用-D参数导入scala的jar包即可。②在控制台中可以直接使用scala命令来运行。

### 常量与变量

#### (★★)scala 为什么设计 var 和val

一想到这两个的区别，大多数人第一反应就是，var 修饰的变量可改变，val 修饰的变量不可改变；但真的如此吗？事实上，var 修饰的对象引用可以改变，val 修饰的则不可改变，但对象的状态却是可以改变的。

参考博客：<https://blog.csdn.net/a1234h/article/details/77962536>

### 数据类型与数据结构

#### (★)map和tuple 的区别

(1) 默认情况下Map构造的是不可变的集合，里面的内容不可修改，一旦修改就变成新的Map，原有的Map内容保持不变；

(2) Map的实例是调用工厂方法模式apply来构造Map实例，而需要主要的是Map是接口，在apply中使用了具体的实现；

(3) Map中存放的时键值对；

(4) Tuple中可以有很多不同类型的数据；

(5) Tuple的另外一个非常重要的使用是作为函数的返回值，在Tuple中返回若干个值

### 类，对象，继承，特质

#### (★★★)解释下: case class,  apply方法?

什么是样例类：进过scala编译器优化后，被更好的用于模式匹配规则的类。①case class的每个参数默认以val存在，除非显示的声明为var。②自动产生伴生对象，且伴生对象会自动产生apply方法来构建对象。③伴生对象自动产生unapply方法，提取主构造器的参数进行模式匹配。④自动产生copy方法，来构建一个与现有值相同的新对象。 ⑤class中自动产生hashcode，toString，equals方法。⑥继承了scala.Product,scala.Serializable两个特质。（这些特质可通过反编译 .class文件获取）  
 apply是一种比较普遍用法。主要用来解决复杂对象的初始化问题。在scala中class和object都可以定义自己的apply方法，类名() 调用的是Object下的apply()方法，对象() 调用的是apply()方法。

#### (★★)scala case class 有什么重要

Case Class是样例类，能够被优化以用于模式匹配。

●构造器中的参数如果不被声明为var的话，默认是val类型

●自动创建伴生对象，实现apply方法，可以不直接显示地new对象

●伴生对象实现unapply方法，从而可以将case class应用于模式匹配

●实现toString、hashCode、copy、equals方法

### 函数与方法

#### (★★)scala 伴生对象有什么作用

与类名相同并且用object修饰的对象叫做伴生对象类和其伴生对象之间可以相互访问私有的方法和属性。

#### (★★★)为什么说scala是函数式编程？

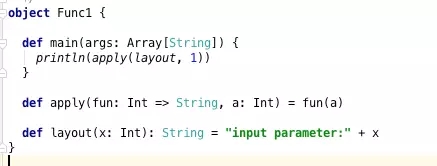
函数是scala中最重要的部分，有人愿意称“函数式scala中的一等公民”。也就是因为函数才使得scala更加简洁、优雅、又耐人寻味，总之，函数式编程在scala中是一等一的重要，使用scala开发的各类框架都存在这大量函数，如果想去了解、阅读Spark源码，函数式编程这一关必须要过,在scala中提供了大量应用例如:map，flatmap，reduce……

将函数作为值,匿名函数,函数的高级用法,将函数作为参数,将函数作为返回值

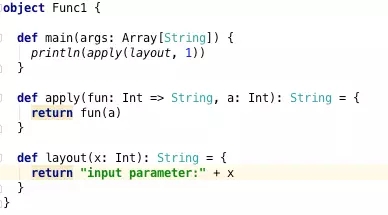
#### (★★★)什么是scala的高阶函数？

Scala高阶函数有两种：

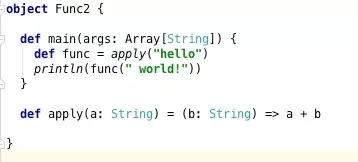
1、将函数作为另一个函数的的参数



比如上面的代码，apply函数有两个参数，fun和a，其中fun是一个函数，更形象一点的写法如下：



2、返回结果是一个函数



上面代码，apply函数返回的结果是一个函数，该函数接受一个参数b，返回的结果是apply函数的参数和参数b两个字符拼接。

在Scala中，高阶函数有很多，比如map、filter等函数，它们都是使用了匿名函数作为参数。

关于函数：将可以做为参数传递到方法中的表达式称之为"函数"。

高阶函数：接受函数字面量作为参数的函数。

Scala中的高阶函数包括：作为值的函数、匿名函数、闭包和柯里化等等。

作为值的函数：将函数赋值给变量的函数。

匿名函数：没有将函数赋值给变量的函数。

闭包：是一个函数，该函数的返回值依赖于声明在函数外部的一个或是多个变量。

柯里化：将原来接受两个参数的方法变成接受一个参数方法的过程。

#### (★★)说一下scala的下划线有哪些作用？

1. 方法转化为函数。空格后下划线: 例：def m1(x:Int, y:Int) = x+y; val f1 = m1 \_

2.集合汇总的每一个元素。例：val list=List(1,2,3,4); val list1=list.map(\_ +1)

3.获取元组Tuple中的元素。例：val t =(“aa”,1, 3.4); t.\_1; t.\_2;t.\_3

4.模式匹配。例： case \_ => 0 表示匹配所有的情况。

5.占位符。例：模式匹配中： case List(\_, \_\*) =>1 表示匹配元素个数大于1的List

6.通配符。例：包导入时，import scala.collection.mutable.\_ 表示该包下所有的类

7.初始化变量。例： var name:String = \_ 值为该类型的默认值：String为null，Int为0，Double为0.0等等。。。。

8.集合中的二元操作。例： val list = List(1,2,3,5); list.reduce(\_ + \_)

### 线程

#### (★★★)scala 并发编程是怎么弄得，你对 actor 模型怎么理解有何优点

Scala很好地解决了java并发编程的问题，要在scala中进行并发编程，有以下几种途径可以实现：

1 actor消息模型、akka actor并发模型。

2 Thread、Runnable

3 java.util.concurennt

4 第三方开源并发框架如Netty，Mina

java中的并发主要是通过线程来实现，各线程采用共享资源的机制来实现程序的并发，这里面临竞争资源的问题，虽然采用锁机制可以避免竞争资源的问题，但会存在死锁问题，要开发一套健壮的并发应用程序具有一定的难度。而scala的并发模型相比于java它更简单，它采用消息传递而非资源共享来实现程序的并发，消息传递正是通过Actor来实现的。

参考网站：<https://blog.csdn.net/qq_36330643/article/details/74452229>

#### (★★★)scala akka 框架有没有接触过，有什么重要

Akka适用场景非常广泛，这里根据一些已有的使用案例来总结一下，Akka能够在哪些应用场景下投入生产环境：

事务处理（Transaction Processing）：

在线游戏系统、金融/银行系统、交易系统、投注系统、社交媒体系统、电信服务系统。

后端服务（Service Backend）：

任何行业的任何类型的应用都可以使用，比如提供REST、SOAP等风格的服务，类似于一个服务总线，Akka支持纵向&横向扩展，以及容错/高可用（HA）的特性。

并行计算（Concurrency/Parallelism）：

任何具有并发/并行计算需求的行业，基于JVM的应用都可以使用，如使用编程语言Scala、Java、Groovy、JRuby开发。

仿真：

Master/Slave架构风格的计算系统、计算网格系统、MapReduce系统。

通信Hub（Communications Hub）：

电信系统、Web媒体系统、手机媒体系统。

复杂事件流处理（Complex Event Stream Processing）：

Akka本身提供的Actor就适合处理基于事件驱动的应用，所以可以更加容易处理具有复杂事件流的应用。

参考网站：<http://www.aboutyun.com/thread-14776-1-1.html>

### 隐式转换与隐式参数

#### (★★★)Scala的隐式转换了解么？

1.当方法中的参数的类型与目标类型不一致时

2.当对象调用类中不存在的方法或成员时，编译器会自动将对象进行隐式转换

1）显示定义规则

在使用带有隐式参数的函数时，如果没有明确指定与参数类型匹配相同的隐式值，编译器不会通过额外的隐式转换来确定函数的要求。

2）作用域规则

不管是隐式值，隐式对象，隐式类或隐式转换函数，都必须在当前的作用域使用才能起作用！

     3）无歧义规则

 所谓无歧义值的是，不能存在多个隐式转换是代码合法，如代码中不应该存在两个隐式转换函数能够同时使某一类型转换为另一类型，也不应该存在相同的两个隐式值，主构造函数参数类型以及成员方法等同的两个隐式类。

     4）一次性转换规则

  隐式转换从源类型到目标类型只会经过一次转换，不会经过多次隐式转换达到。

#### (★★)什么是Scala的隐式转换？

关于隐式转换：

①通过隐式转换，可以在编写程序时不用书写那些冗长的、过于细节的代码，让编译器在编译时自动推导出这些信息来。

②使用隐式转换可以极大地减少代码量。

使用方式：

①将方法或是变量标识为隐式的（implicit）

②将方法的参数列表标记为隐式的（implicit）

③将类标记为隐式的（implicit）

隐式转换的时机：

①当方法中的参数类型与目标类型不一致时

②对象调用类中不存在的方法或是成员时，编译器会自动将对象进行隐式转换

### 编程题

#### (★★★)使用Scala进行排序、冒泡，快排，归并

答：冒泡：

object BubbleSort {

def main(args: Array[String]) {

val arr = Array(3, 12, 43, 23, 7, 1, 2, 0)

var temp = 0

for (i <- 0 until arr.length - 1) {

for (j <- 0 until (arr.length - i - 1) {

if (arr(j) > arr(j + 1) {

temp = arr(j)

arr(j) = arr(j + 1)

arr(j + 1) = temp

}

}

}

for (num <- arr) {

print(num + "\t")

}

}

}

快排：

object QuickSort {

def quickSort(arr:Array[Int],low:Int,high:Int):Unit = {

var i = 0

var j = 0

var temp = 0

var t = 0

if (low > high) {

return

}

i = low

j = high

// temp就是基准位

temp = arr(low)

while (i < j) {

// 先看右边，依次往左递减

while (temp <= arr(j) && i < j) {

j -= 1

}

// 再看左边，依次往右递增

while (temp >= arr(i) && i < j) {

i += 1

}

if (i < j) {

t = arr(j)

arr(j) = arr(i)

arr(i) = t

}

}

// 最后将基准位与i和j相等位置的数字交换

arr(low) = arr(i)

arr(i) = temp

// 递归调用左半数组

quickSort(arr,low,j - 1)

// 递归调用右半数组

quickSort(arr,j + 1,high)

}

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

val arr:Array[Int] = Array(10,7,2,4,7,62,3,4,2,1,8,9,19)

quickSort(arr,0,arr.length - 1)

for (elem <- arr) {

print(elem + "\t")

}

}

}

归并：

object MergeSort {

def mergedSort[T](less: (T, T) => Boolean)(list: List[T]): List[T] = {

// 找到第一个最小的值，然后递归的进行比较（less只是一个def函数）

def merged(xList: List[T], yList: List[T]): List[T] = {

(xList, yList) match {

case (Nil, \_) => yList

case (x :: xTail, y :: yTail) => {

if (less(x, y) x :: merged(xTail, yList)

else

y :: merged(xList, yTail)

}

}

}

val n = list.length / 2

if (n == 0) list

else {

// splitAt 从n个元素开始切分，把数据分为2份

val (x, y) = list splitAt n

// 合并左右的数组

merged(mergedSort(less)(x), mergedSort(less)(y)

}

}

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

val list = List(1, 2, 3, 7, 12, 20, 23, 43)

println(mergedSort(x: Int, y: Int) => x < y)(list)

}

}