

# 尚硅谷大数据技术之 Scala

(作者：尚硅谷大数据研发部)

## 第 1 章 Scala 入门

### 1.1 概述

#### 1.1.1 为什么学习 Scala



- 1) Spark—新一代内存级大数据计算框架，是大数据的重要内容。
- 2) Spark就是使用Scala编写的。因此为了更好的学习Spark，需要掌握Scala这门语言。
- 3) Spark的兴起，带动Scala语言的发展！

让天下没有难学的技术

## 1.1.2 Scala 发展历史

### Scala发展历史

联邦理工学院的马丁·奥德斯基（Martin Odersky）于2001年开始设计Scala。

马丁·奥德斯基是编译器及编程的狂热爱好者，长时间的编程之后，希望发明一种语言，能够让写程序这样的基础工作变得高效，简单。所以当接触到JAVA语言后，对JAVA这门便携式，运行在网络，且存在垃圾回收的语言产生了极大的兴趣，所以决定将函数式编程语言的特点融合到JAVA中，由此发明了两种语言（Pizza & Scala）。



Pizza和Scala极大地推动了Java编程语言的发展。

- JDK5.0 的泛型、增强for循环、自动类型转换等，都是从Pizza引入的新特性。
- JDK8.0 的类型推断、Lambda表达式就是从Scala引入的特性。

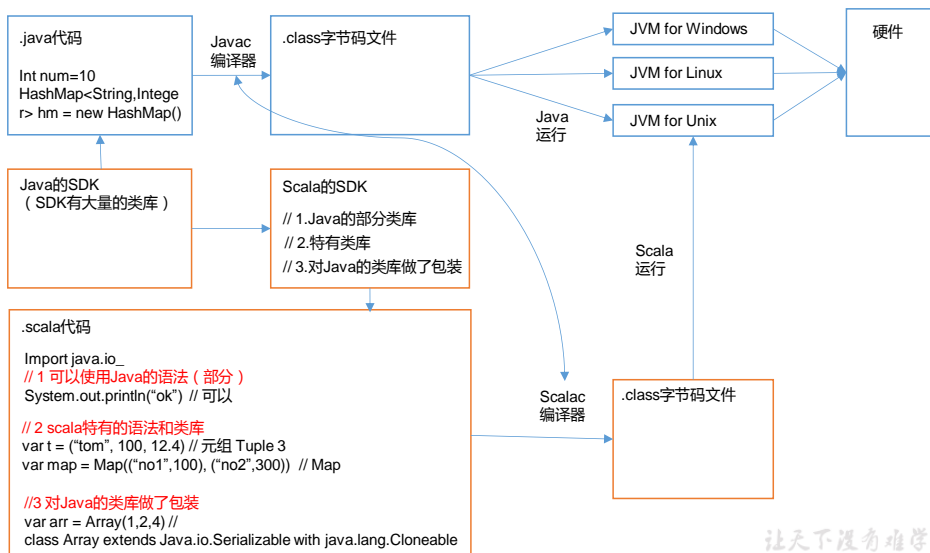
JDK5.0和JDK8.0的编辑器就是马丁·奥德斯基写的，因此马丁·奥德斯基一个人的战斗力抵得上一个Java开发团队。

让天下没有难学的技术

## 1.1.3 Scala 和 Java 关系

一般来说，学 Scala 的人，都会 Java，而 Scala 是基于 Java 的，因此我们需要将 Scala 和 Java 以及 JVM 之间的关系搞清楚，否则学习 Scala 你会蒙圈。

### Scala和Java及JVM关系图



让天下没有难学的技术

## 1.1.4 Scala 语言特点

### Scala语言特点



Scala是一门以Java虚拟机（JVM）为运行环境并将**面向对象**和**函数式编程**的最佳特性结合在一起的**静态类型编程语言**（静态语言需要提前编译的如：Java、c、c++等，动态语言如：js）。

1) Scala是一门**多范式**的编程语言，Scala支持**面向对象**和**函数式编程**。（多范式，就是多种编程方法的意思。有面向过程、面向对象、泛型、函数式四种程序设计方法。）

2) Scala源代码（.scala）会被编译成Java字节码（.class），然后运行于JVM之上，**并可以调用现有的Java类库，实现两种语言的无缝对接。**

3) Scala单作为一门语言来看，非常的**简洁高效**。

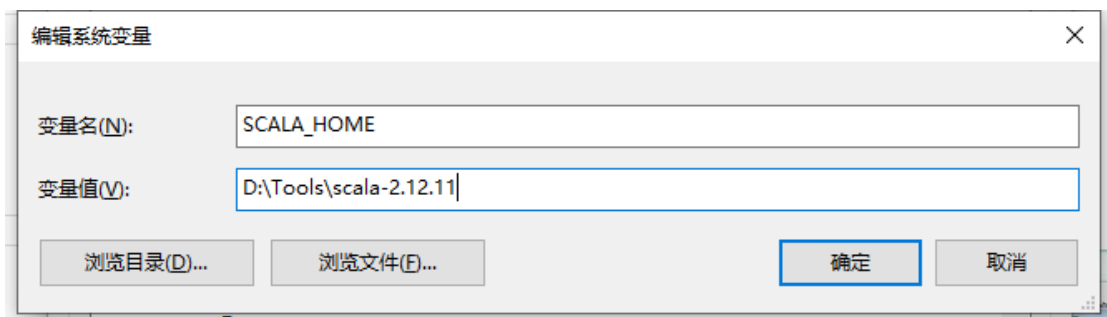
4) Scala在设计时，马丁·奥德斯基是参考了Java的设计思想，可以说Scala是源于Java，同时马丁·奥德斯基也加入了自己的思想，将**函数式编程语言的特点融合到JAVA中**，因此，对于学习过Java的同学，只要在学习Scala的过程中，搞清楚Scala和Java相同点和不同点，就可以快速的掌握Scala这门语言。

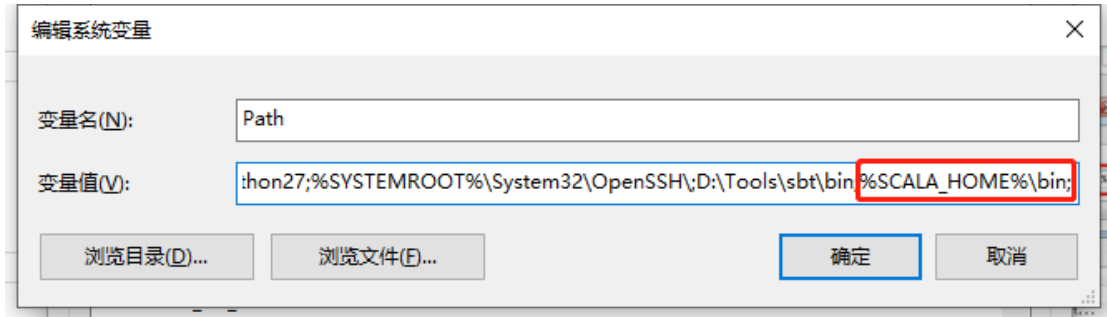
让天下没有难学的技术

## 1.2 Scala 环境搭建

### 1) 安装步骤

- (1) 首先确保 JDK1.8 安装成功
- (2) 下载对应的 Scala 安装文件 scala-2.12.11.zip
- (3) 解压 scala-2.12.11.zip，我这里解压到 D:\Tools
- (4) 配置 Scala 的环境变量





注意 1：解压路径不能有任何中文路径，最好不要有空格。

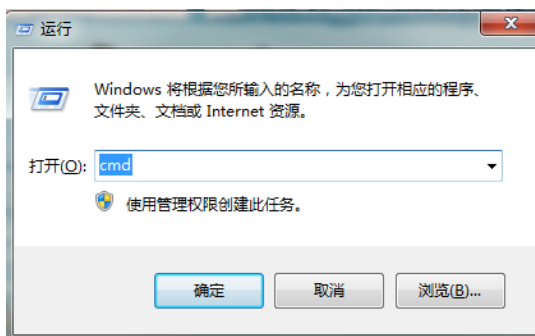
注意 2：环境变量要大写 SCALA\_HOME

## 2) 测试

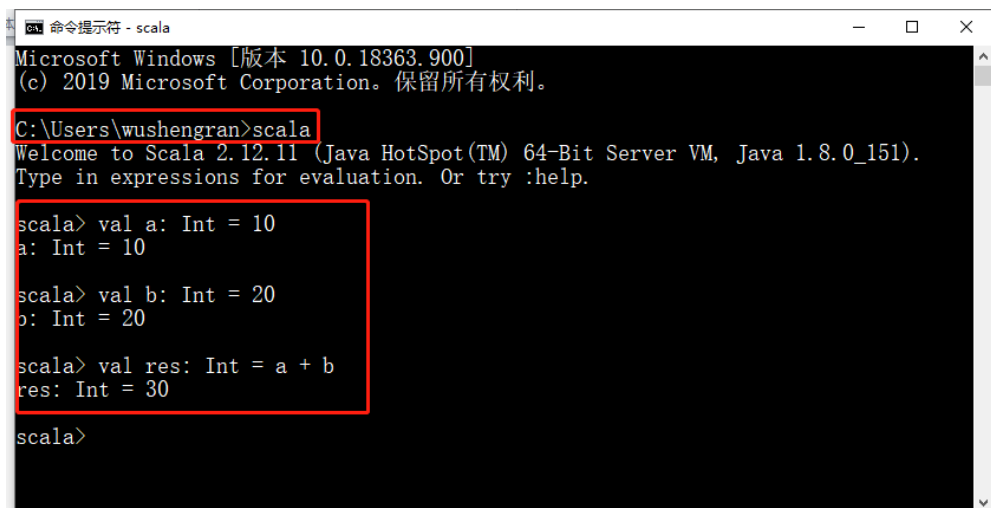
需求：计算两数 a 和 b 的和。

步骤

(1) 在键盘上同时按 win+r 键，并在运行窗口输入 cmd 命令



(2) 输入 Scala 并按回车键，启动 Scala 环境。然后定义两个变量，并计算求和。



更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

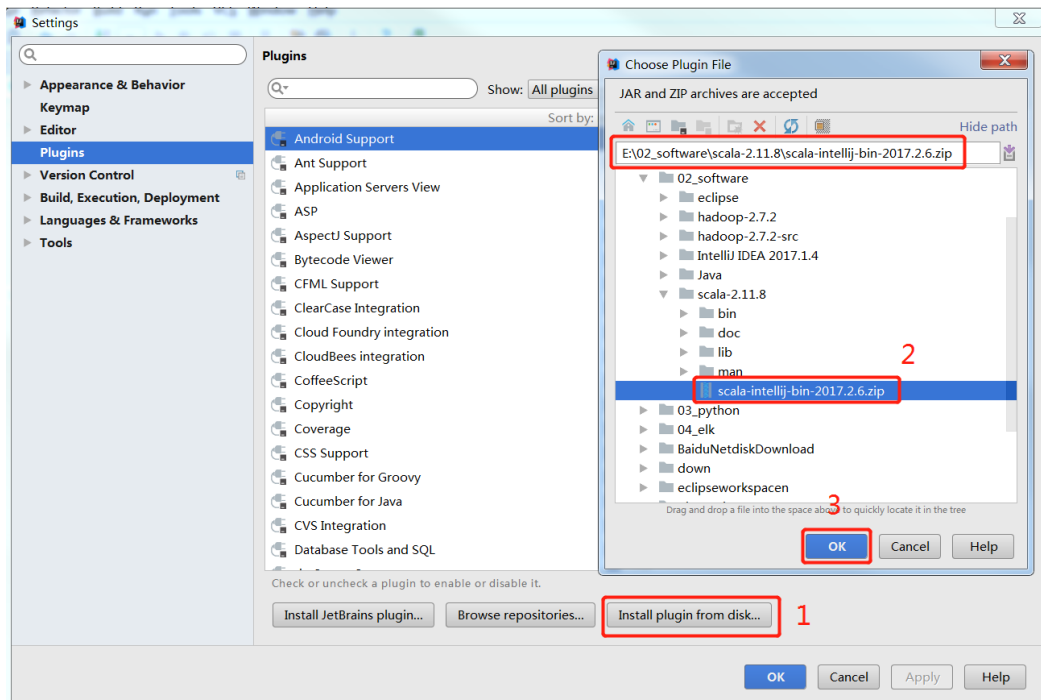
### 1.3 Scala 插件安装

默认情况下 IDEA 不支持 Scala 的开发，需要安装 Scala 插件。

#### 1) 插件离线安装步骤

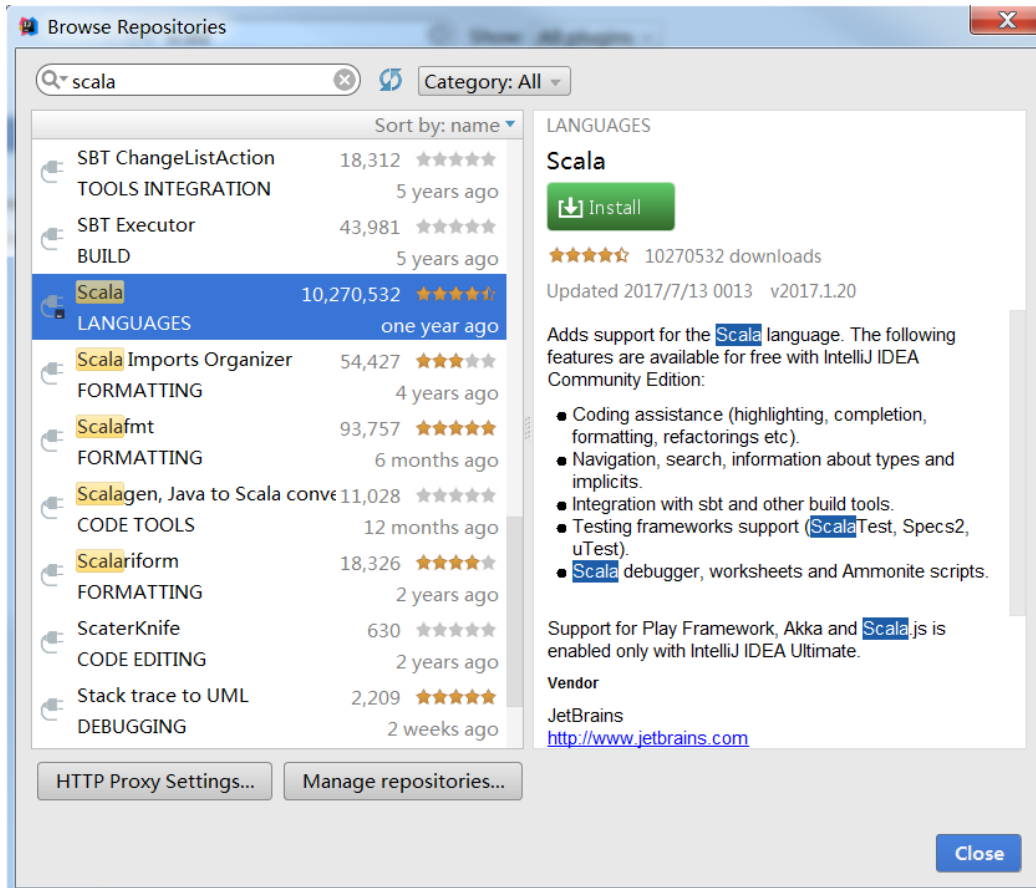
(1) 建议将该插件 `scala-intellij-bin-2017.2.6.zip` 文件，放到 Scala 的安装目录 `D:\Tools\scala-2.12.11` 下，方便管理。

(2) 打开 IDEA，在左上角找到 `File->在下拉菜单中点击 Setting...->点击 Plugins->点击右下角 Install plugin from disk...`，找到插件存储路径 `D:\Tools\scala-2.12.11\scala-intellij-bin-2017.2.6.zip`，最后点击 `ok`。

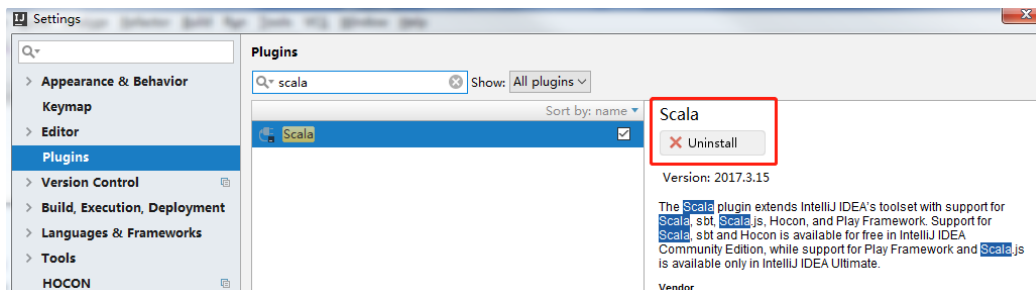


#### 2) 插件在线安装 (可选)

(1) 在搜索插件框里面输入 `Scala`->点击 `Install`->点击 `ok`->点击 `apply`。



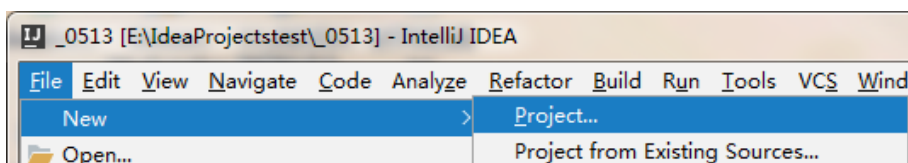
(2) 重启 IDEA , 再次来到 Scala 插件页面 , 已经变成 Uninstall。



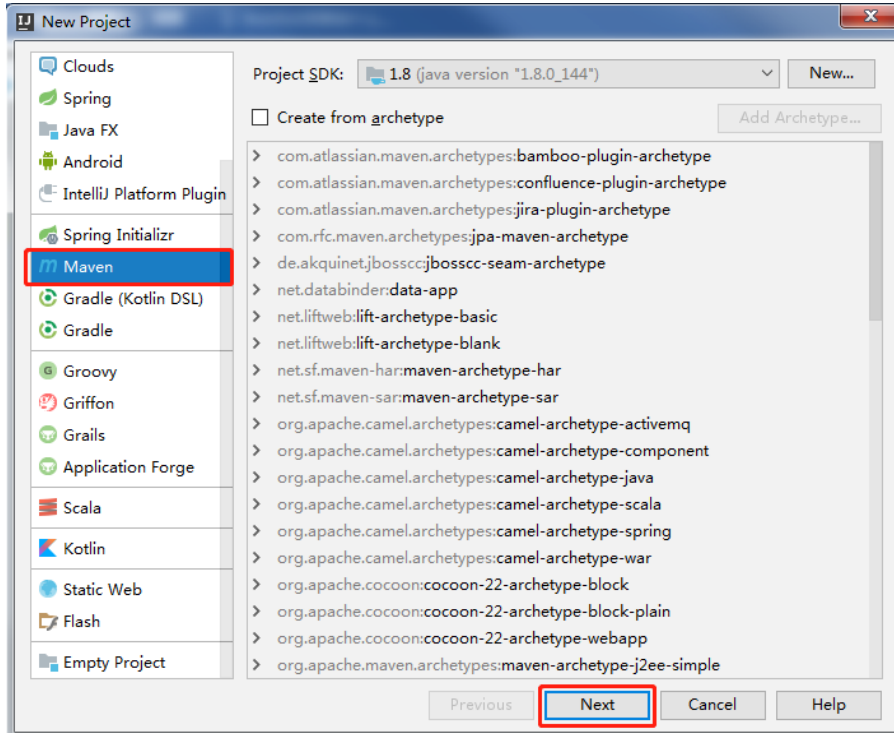
## 1.4 HelloWorld 案例

### 1.4.1 创建 IDEA 项目工程

1) 打开 IDEA->点击左侧的 File->选择 New->选择 Project...

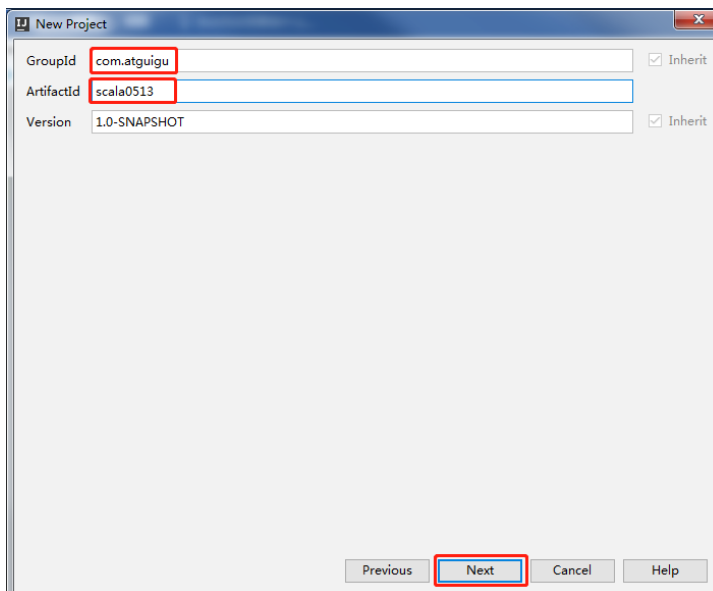


2) 创建一个 Maven 工程 , 并点击 next

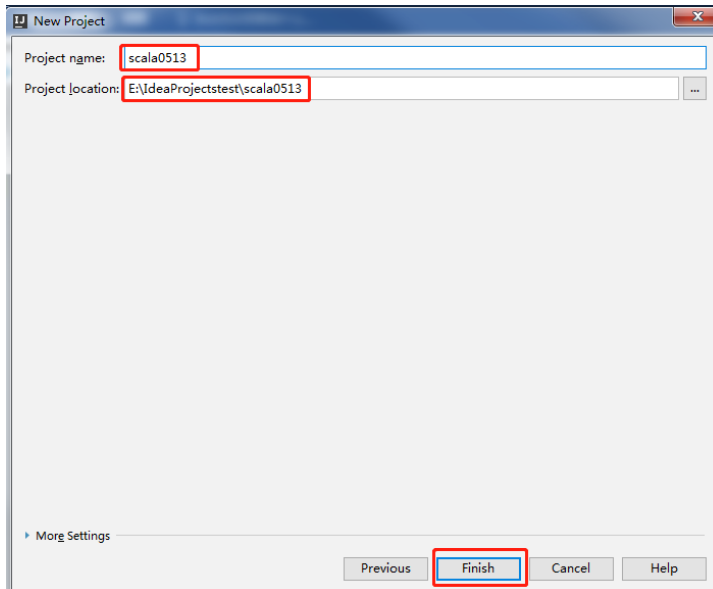


3 ) groupId 输入 com.atguigu->ArtifactId 输入 scala->点击 next->点击 Finish

注意：工程存储路径一定不能有中文和空格。

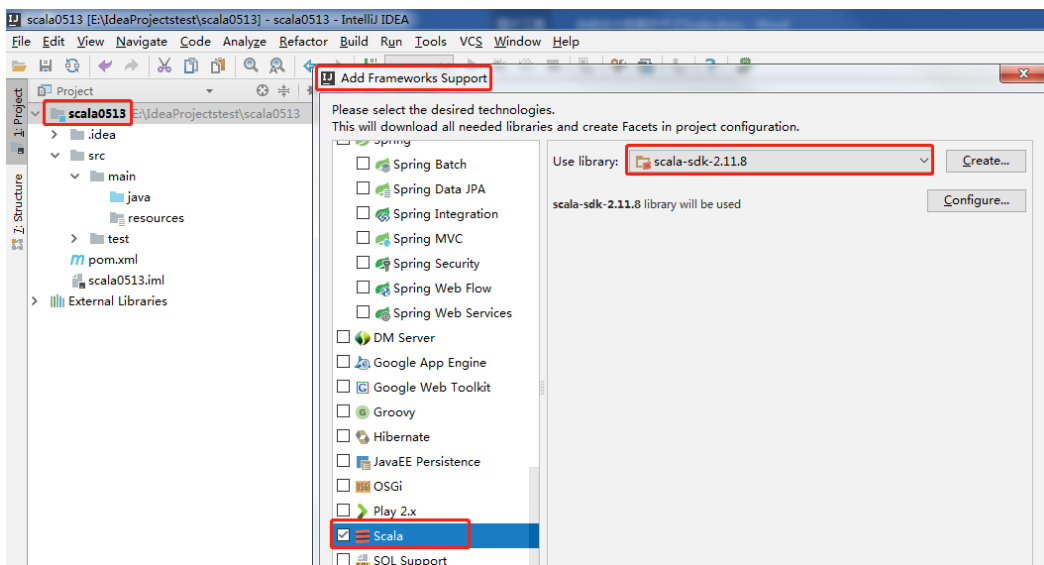


4 ) 指定项目工作目录空间



5) 默认下，Maven 不支持 Scala 的开发，需要引入 Scala 框架。

在 scala0513 项目上，点击右键-> Add Framework Support... ->选择 Scala->点击 OK



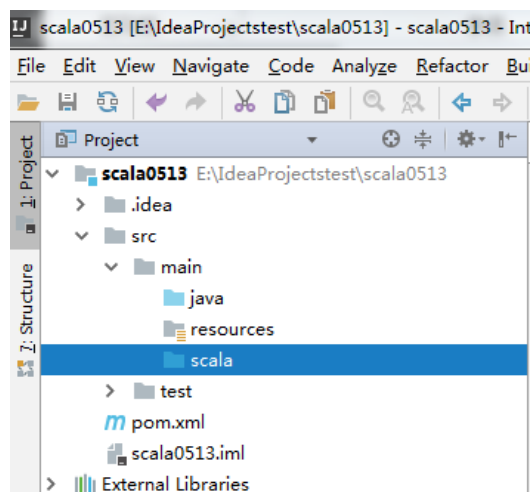
**注意：**如果是第一次引入框架，Use library 看不到，需要选择你的 Scala 安装目录，然后工具就会自动识别，就会显示 user library。

6) 创建项目的源文件目录

右键点击 main 目录->New->点击 Directory -> 写个名字 ( 比如 scala )。

右键点击 scala 目录->Mark Directory as->选择 Sources root，观察文件夹颜色发生变化。



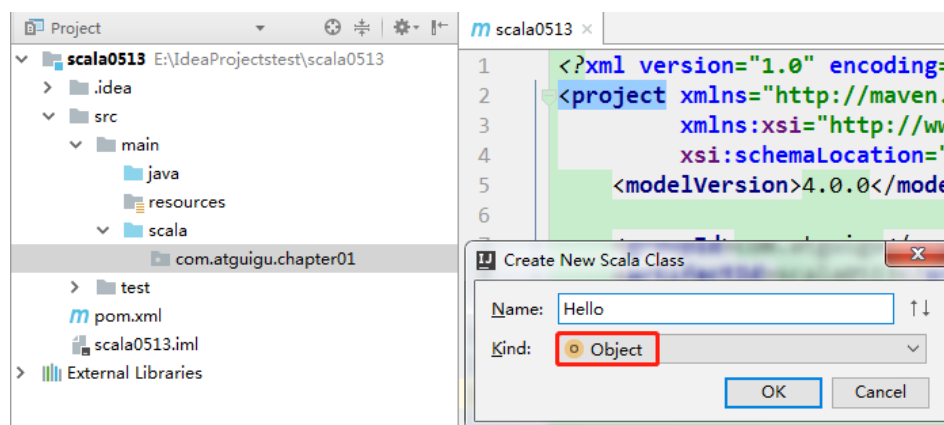


7) 在 scala 包下，创建包 com.atguigu.chapter01 包名和 Hello 类名，

右键点击 scala 目录->New->Package->输入 com.atguigu.chapter01->点击 OK。

右键点击 com.atguigu.chapter01->New->Scala Class->Kind 项选择 Object->Name 项输入

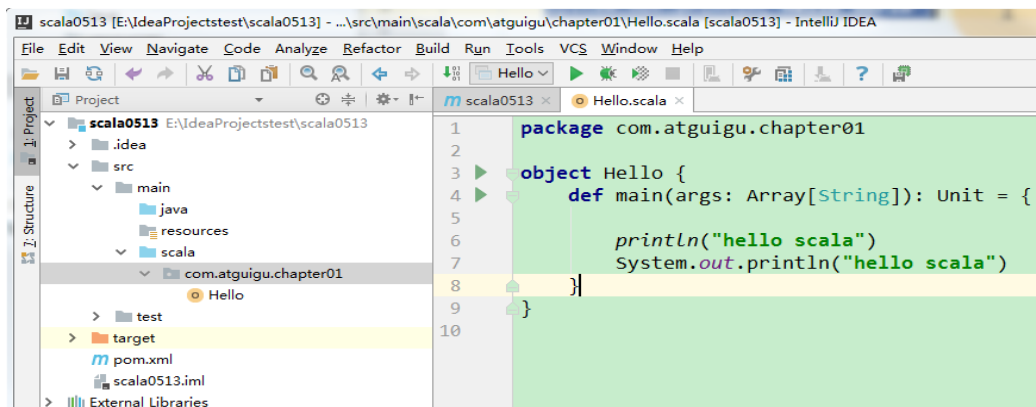
Hello。



8) 编写输出 Hello Scala 案例

在类中输入 main，然后回车可以快速生成 main 方法；

在 main 方法中输入 println("hello scala")



运行后，观察控制台打印输出：

hello scala

hello scala

说明：Java 中部分代码也是可以在 Scala 中运行。

## 1.4.2 class 和 object 说明

对第一个程序进行说明

```
// main方法名
// 小括号表示参数列表
//     参数声明方式: java -> 类型 参数名
//                     scala -> 参数名 : 类型
// public修饰符: scala中没有public关键字，如果不声明访问权限，那么就是公共的。
// static修饰符: scala中没有静态语法，所以没有static关键字。
// void关键字: 表示返回值，但是不遵循面向对象语法，所以scala中没有，但是有Unit类型，表示没有返回值
// scala中: 方法名（参数列表）: 返回值类型
// scala中声明方法必须采用关键字def声明
// scala中方法实现赋值给方法声明，所以中间需要等号连接

// Scala是一个完全面向对象的语言，所以没有静态语法，为了能调用静态语法（模仿静态语法），
//     采用伴生对象单例的方式调用方法
def main(args: Array[String]): Unit = {
```

```
package com.atguigu.chapter01

object Hello {

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

        println("hello scala")

    }

}
```

Scala完全面向对象，故Scala去掉了Java中非面向对象的元素，如static关键字，void类型

#### 1) static

Scala无static关键字，由object实现类似静态方法的功能（类名.方法名）。

class关键字和Java中的class关键字作用相同，用来定义一个类；

#### 2) void

对于无返回值的函数，Scala定义其返回值类型为Unit类

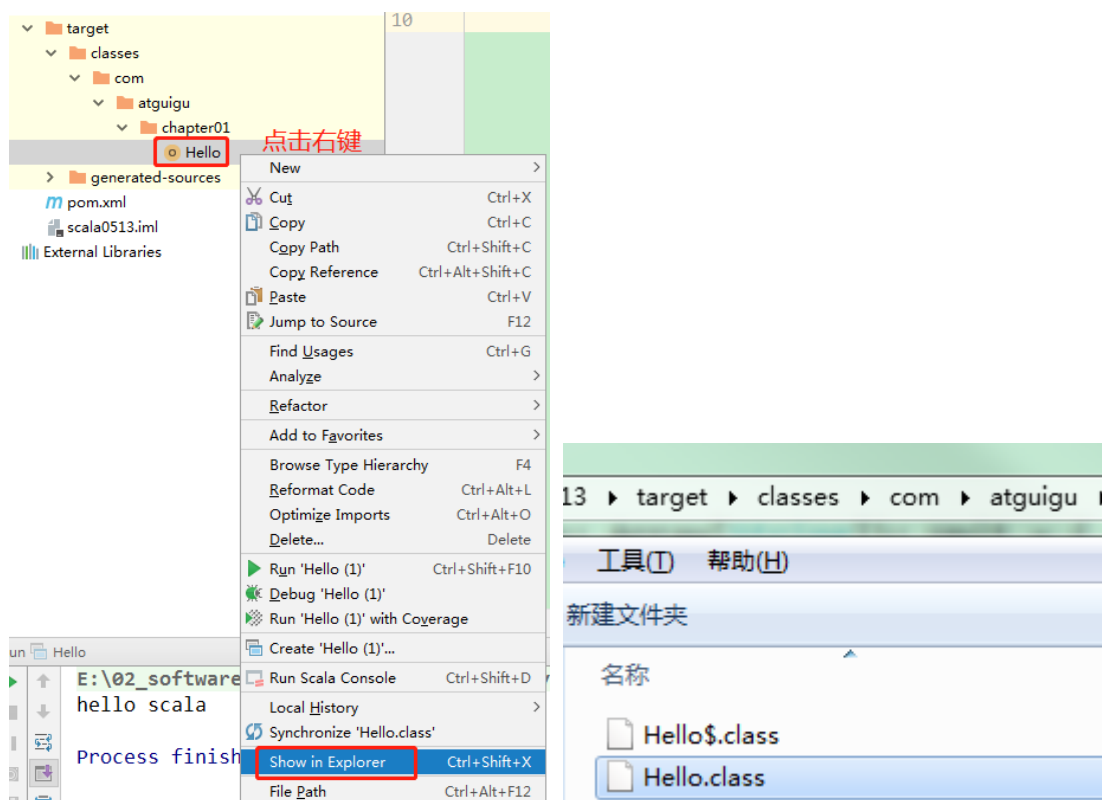
让天下没有难学的技术

## 1.4.3 Scala 程序反编译

1) 在项目的 target 目录 Hello 文件上点击右键->Show in Explorer->看到 object 底层生成

Hello\$.class 和 Hello.class 两个文件

2) 采用 Java 反编译工具 jd-gui.exe 反编译代码，将 Hello.class 拖到 jd-gui.exe 页面



### 1) Hello源代码

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("hello,scala")
  }
}
```

(1) Object编译后生成Hello\$.class和Hello.class两个文件

### 2) Hello.class类

```
package com.atguigu.chapter01

public final class Hello
{
  public static void main(String[] paramArrayOfString)
  {
    Hello$.MODULE$.main(paramArrayOfString);
  }
}
```

(2) Hello中有个main函数，调用 Hello\$ 类的一个静态对象 MODULE\$

### 3) Hello\$.class类

```
public final class Hello$
{
  public static final MODULE$;

  static
  {
    new ();
  }

  public void main(String[] args)
  {
    Predef..MODULE$.println("hello,scala");
  }

  private Hello$()
  {
    MODULE$ = this;
  }
}
```

(3) Hello\$.MODULE\$. 对象是静态的，通过该对象调用 Hello\$的主函数

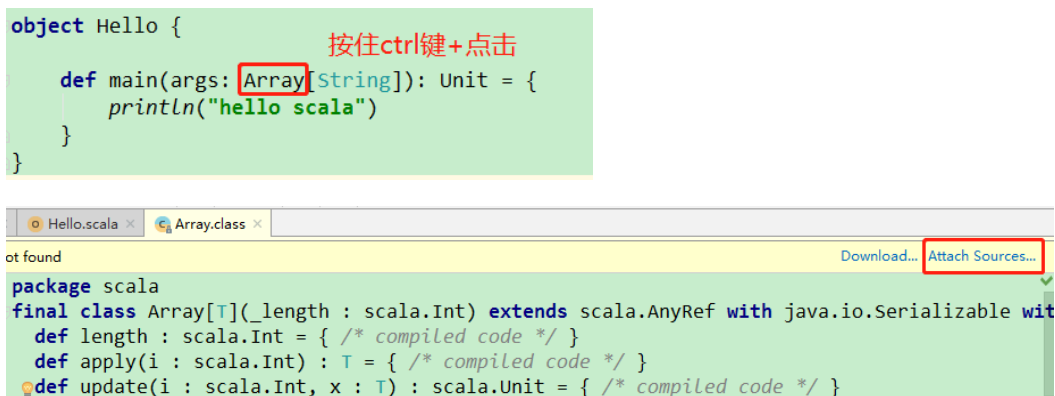
让天下没有难学的技术

## 1.5 关联 Scala 源码

在使用 Scala 过程中，为了搞清楚 Scala 底层的机制，需要查看源码，下面看看如何关联和查看 Scala 的源码包。

### 1) 查看源码

例如查看 Array 源码。按住 ctrl 键->点击 Array->右上角出现 Attach Sources...



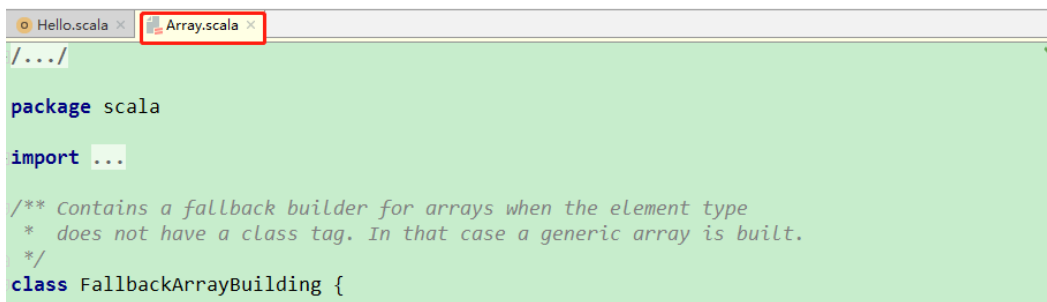
### 2) 关联源码

(1) 将我们的源码包 scala-sources-2.12.11.tar.gz 拷贝到 D:\Tools\scala-2.12.11\lib 文件夹下，并解压为 scala-sources-2.12.11 文件夹

(2) 点击 Attach Sources...->选择 D:\Tools\scala-2.12.11\lib\scala-sources-2.12.11，这个

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

文件夹，就可以看到源码了



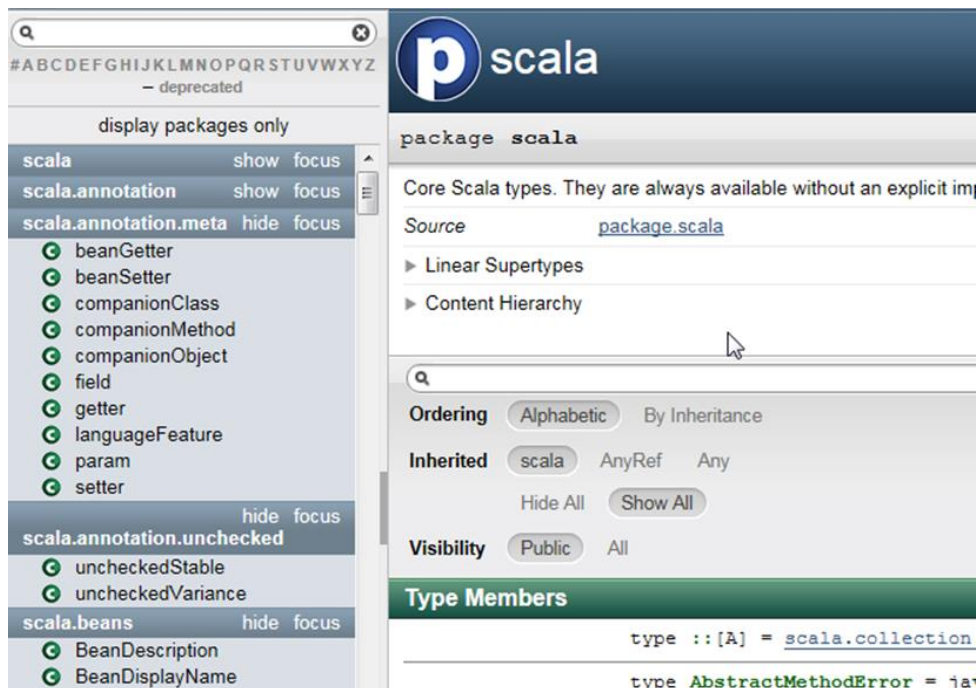
```
package scala

import ...

/** Contains a fallback builder for arrays when the element type
 *  does not have a class tag. In that case a generic array is built.
 */
class FallbackArrayBuilding {
```

## 1.6 官方编程指南

- 1) 在线查看：<https://www.scala-lang.org/>
- 2) 离线查看：解压 scala-docs-2.11.8.zip，可以获得 Scala 的 API 操作。



## 第 2 章 变量和数据类型

### 2.1 注释

Scala 注释使用和 Java 完全一样。

注释是一个程序员必须要具有的良好编程习惯。将自己的思想通过注释先整理出来，再用代码去体现。

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

## 1) 基本语法

(1) 单行注释：`//`

(2) 多行注释：`/* */`

(3) 文档注释：`/**`

`*`

`*/`

## 2) 案例实操

```
package com.atguigu.chapter02

object TestNotes {

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

        // (1) 单行注释：//
        println("dalang")

        // (2) 多行注释：/* */
        /*
        println("dalang")
        println("jinlian")
        */

        // (3) 文档注释：/**
        /**
        /**/
        /**
        * println("qingge")
        * println("qingge")
        * println("qingge")
        */
    }
}
```

## 3) 代码规范

(1) 使用一次 `tab` 操作，实现缩进，默认整体向右边移动，用 `shift+tab` 整体向左移

(2) 或者使用 `ctrl + alt + L` 来进行格式化

(3) 运算符两边习惯性各加一个空格。比如：`2 + 4 * 5`。

(4) 一行最长不超过 80 个字符，超过的请使用换行展示，尽量保持格式优雅

## 2.2 变量和常量（重点）

常量：在程序执行的过程中，其值不会被改变的变量

0) 回顾：Java 变量和常量语法

变量类型 变量名称 = 初始值                      int a = 10

final 常量类型 常量名称 = 初始值                  final int b = 20

1) 基本语法

var 变量名 [: 变量类型] = 初始值                  var i: Int = 10

val 常量名 [: 常量类型] = 初始值                  val j: Int = 20

**注意：能用常量的地方不用变量**

2) 案例实操

(1) 声明变量时，类型可以省略，编译器自动推导，即类型推导

(2) 类型确定后，就不能修改，说明 Scala 是强数据类型语言。

(3) 变量声明时，必须要有初始值

(4) 在声明/定义一个变量时，可以使用 var 或者 val 来修饰，var 修饰的变量可改变，

val 修饰的变量不可改。

```
package com.atguigu.chapter02

object TestVar {

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

        // (1) 声明变量时，类型可以省略，编译器自动推导，即类型推导
        var age = 18
        age = 30

        // (2) 类型确定后，就不能修改，说明 Scala 是强数据类型语言。
        //      age = "tom" // 错误

        // (3) 变量声明时，必须要有初始值
        //      var name // 错误
    }
}
```

// (4) 在声明/定义一个变量时, 可以使用 `var` 或者 `val` 来修饰, `var` 修饰的变量可改变, `val` 修饰的变量不可改。

```
var num1 = 10    // 可变
val num2 = 20    // 不可变

num1 = 30    // 正确
//num2 = 100 // 错误, 因为 num2 是 val 修饰的
}
```

(5) `var` 修饰的对象引用可以改变, `val` 修饰的对象则不可改变, 但对象的状态 (值)

却是可以改变的。(比如: 自定义对象、数组、集合等等)

```
object TestVar {

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

    // p1 是 var 修饰的, p1 的属性可以变, 而且 p1 本身也可以变
    var p1 = new Person()
    p1.name = "dalang"
    p1 = null

    // p2 是 val 修饰的, 那么 p2 本身就不可变 (即 p2 的内存地址不能变),
    // 但是, p2 的属性是可以变, 因为属性并没有用 val 修饰。
    val p2 = new Person()
    p2.name = "jinlian"
    // p2 = null // 错误的, 因为 p2 是 val 修饰的
  }

}

class Person {
  var name: String = "jinlian"
}
```

## 2.3 标识符的命名规范

Scala 对 **各种变量、方法、函数** 等命名时使用的字符序列称为 **标识符**。即: 凡是自己可以起名字的地方都叫标识符。

### 1) 命名规则

Scala 中的标识符声明, **基本和 Java 是一致的**, 但是细节上会有所变化, 有以下三种规则:

(1) 以字母或者下划线开头, 后接字母、数字、下划线

[更多 Java - 大数据 - 前端 - python 人工智能资料下载, 可百度访问: 尚硅谷官网](#)



(2) 以操作符开头, 且只包含操作符 (+ - \* / # !等)

(3) 用反引号 `...` 包括的任意字符串, 即使是 Scala 关键字 (39 个) 也可以

- package, import, class, **object**, **trait**, extends, **with**, type, for
- private, protected, abstract, **sealed**, final, **implicit**, lazy, override
- try, catch, finally, throw
- if, else, **match**, case, do, while, for, return, **yield**
- **def**, **val**, **var**
- this, super
- new
- true, false, null

## 2) 案例实操

需求: 判断 hello、Hello12、1hello、h-b、x h、h\_4、\_ab、Int、\_、+\*-/#!、+\*-/#!1、if、

`if`, 这些名字是否合法。

```
object TestName {  
    def main(args: Array[String]): Unit = {  
  
        // (1) 以字母或者下划线开头, 后接字母、数字、下划线  
        var hello: String = "" // ok  
        var Hello12: String = "" // ok  
        var 1hello: String = "" // error 数字不能开头  
  
        var h-b: String = "" // error 不能用-  
        var x h: String = "" // error 不能有空格  
        var h_4: String = "" // ok  
        var _ab: String = "" // ok  
        var Int: String = "" // ok 因为在 Scala 中 Int 是预定义的字符,  
        不是关键字, 但不推荐  
  
        var _: String = "hello" // ok 单独一个下划线不可以作为标识符,  
        因为_被认为是一个方法  
        println(_)  
  
        // (2) 以操作符开头, 且只包含操作符 (+ - * / # !等)  
        var +*-/#! : String = "" // ok  
        var +*-/#!1 : String = "" // error 以操作符开头, 必须都是操作
```

符

```
// (3) 用反引号 `...` 包括的任意字符串, 即使是 Scala 关键字 (39 个)
也可以
var if : String = "" // error 不能用关键字
var `if` : String = "" // ok 用反引号 `...` 包括的任意字符串,
包括关键字
}
```

## 2.4 字符串输出

### 1) 基本语法

- (1) 字符串, 通过+号连接
- (2) printf 用法: 字符串, 通过%传值。
- (3) 字符串模板 (插值字符串): 通过\$获取变量值

### 2) 案例实操

```
package com.atguigu.chapter02

object TestCharType {

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

    var name: String = "jinlian"
    var age: Int = 18

    // (1) 字符串, 通过+号连接
    println(name + " " + age)

    // (2) printf 用法字符串, 通过%传值。
    printf("name=%s age=%d\n", name, age)

    // (3) 字符串, 通过$引用
    // 多行字符串, 在 Scala 中, 利用三个双引号包围多行字符串就可以实现。
    // 输入的内容, 带有空格、\t 之类, 导致每一行的开始位置不能整洁对齐。
    // 应用 scala 的 stripMargin 方法, 在 scala 中 stripMargin 默认
    // 是"|"作为连接符, // 在多行换行的行头前面加一个"|"符号即可。
    val s =
      """
        |select
        |  name,
        |  age
        |from user
      """
  }
```

```
        |where name="zhangsan"
        """".stripMargin
println(s)

//如果需要变量进行运算，那么可以加${}
val s1 =
    s"""
        |select
        |    name,
        |    age
        |from user
        |where name="$name" and age=${age+2}
        """".stripMargin
println(s1)

val s2 = s"name=$name"
println(s2)
    }
}
```

## 2.5 键盘输入

在编程中，需要接收用户输入的数据，就可以使用键盘输入语句来获取。

### 1) 基本语法

StdIn.readLine()、StdIn.readShort()、StdIn.readDouble()

### 2) 案例实操

需求：可以从控制台接收用户信息，【姓名，年龄，薪水】。

```
import scala.io.StdIn

object TestInput {

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

        // 1 输入姓名
        println("input name:")
        var name = StdIn.readLine()

        // 2 输入年龄
        println("input age:")
        var age = StdIn.readShort()

        // 3 输入薪水
        println("input sal:")
        var sal = StdIn.readDouble()

        // 4 打印
        println("name=" + name)
```

```
println("age=" + age)
println("sal=" + sal)
}
```

## 2.6 数据类型（重点）

### 回顾：Java数据类型

Java基本类型：char、byte、short、int、long、float、double、boolean

Java引用类型：（对象类型）

由于Java有基本类型，而且基本类型不是真正意义的对象，即使后面产生了基本类型的包装类，但是仍然存在基本数据类型，所以Java语言并不是真正意义的面向对象。

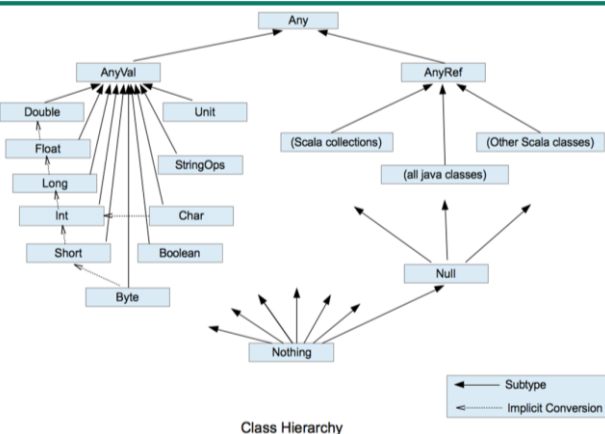
Java基本类型的包装类：Character、Byte、Short、Integer、Long、Float、Double、Boolean

注意：Java中基本类型和引用类型没有共同的祖先。

让天下没有难学的技术

### Scala数据类型

- 1) Scala中一切数据都是对象，都是Any的子类。
- 2) Scala中数据类型分为两大类：数值类型（AnyVal）、引用类型（AnyRef），不管是值类型还是引用类型都是对象。
- 3) Scala数据类型仍然遵守，低精度的值类型向高精度值类型，自动转换（隐式转换）
- 4) Scala中的StringOps是对Java中的String增强
- 5) Unit：对应Java中的void，用于方法返回值的位置，表示方法没有返回值。Unit是一个数据类型，只有一个对象就是()。Void不是数据类型，只是一个关键字
- 6) Null是一个类型，只有一个对象就是null。它是所有引用类型（AnyRef）的子类。
- 7) Nothing，是所有数据类型的子类，主要用在函数没有明确返回值时使用，因为这样我们可以把抛出的返回值，返回给任何的变量或者函数。



让天下没有难学的技术

```
//Scala中一切数据都是对象，都是Any的子类
10 //和new User() 意义一样
```

## 2.7 整数类型（Byte、Short、Int、Long）

Scala 的整数类型就是用于存放整数值，比如 12，30，3456 等等。

### 1) 整型分类

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

数据类型	描述
Byte [1]	8 位有符号补码整数。数值区间为 -128 到 127
Short [2]	16 位有符号补码整数。数值区间为 -32768 到 32767
Int [4]	32 位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long [8]	64 位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807 = 2 的(64-1)次方-1

## 2) 案例实操

(1) Scala 各整数类型有固定的表示范围和字段长度，不受具体操作的影响，以保证

Scala 程序的可移植性。

```
object TestDataType {  
    def main(args: Array[String]): Unit = {  
  
        // 正确  
        var n1:Byte = 127  
        var n2:Byte = -128  
  
        // 错误  
        // var n3:Byte = 128  
        // var n4:Byte = -129  
    }  
}
```

(2) Scala 的整型，默认为 Int 型，声明 Long 型，须后加 'l' 或 'L'

```
object TestDataType {  
    def main(args: Array[String]): Unit = {  
  
        var n5 = 10  
        println(n5)  
  
        var n6 = 9223372036854775807L  
        println(n6)  
    }  
}
```

(3) Scala 程序中变量常声明为 Int 型，除非不足以表示大数，才使用 Long

## 2.8 浮点类型 ( Float、Double )

Scala 的浮点类型可以表示一个小数，比如 123.4f，7.8，0.12 等等。

### 1) 浮点型分类

数据类型	描述
Float [4]	32 位, IEEE 754 标准的单精度浮点数
Double [8]	64 位 IEEE 754 标准的双精度浮点数

### 2) 案例实操

Scala 的浮点型常量默认为 Double 型，声明 Float 型常量，须后加 'f' 或 'F'。

```
object TestDataType {  
    def main(args: Array[String]): Unit = {  
  
        // 建议，在开发中需要高精度小数时，请选择 Double  
        var n7 = 2.2345678912f  
        var n8 = 2.2345678912  
  
        println("n7=" + n7)  
        println("n8=" + n8)  
    }  
}
```

//运行的结果

```
n7=2.2345679  
n8=2.2345678912
```

## 2.9 字符类型 ( Char )

### 1) 基本说明

字符类型可以表示单个字符，字符类型是 Char。

### 2) 案例实操

(1) 字符常量是用单引号 ' ' 括起来的单个字符。

(2) \t ：一个制表位，实现对齐的功能

(3) \n ：换行符

(4)\\ :表示\

(5)\":表示"

```
object TestCharType {

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

        //(1) 字符常量是用单引号 ' ' 括起来的单个字符。
        var c1: Char = 'a'
        println("c1=" + c1)
        //注意：这里涉及自动类型提升，其实编译器可以自定判断是否超出范围，
        //不过 idea 提示报错
        var c2:Char = 'a' + 1
        println(c2)

        //(2) \t : 一个制表位，实现对齐的功能
        println("姓名\t年龄")

        //(3) \n : 换行符
        println("西门庆\n潘金莲")

        //(4) \\ : 表示\
        println("c:\\岛国\\avi")

        //(5) \" : 表示"
        println("同学们都说：\"大海哥最帅\"")

    }

}
```

## 2.10 布尔类型：Boolean

### 1) 基本说明

(1) 布尔类型也叫 Boolean 类型，Boolean 类型数据只允许取值 true 和 false

(2) boolean 类型占 1 个字节。

### 2) 案例实操

```
object TestBooleanType {

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

        var isResult : Boolean = false
        var isResult2 : Boolean = true

    }

}
```

## 2.11 Unit 类型、Null 类型和 Nothing 类型（重点）

### 1) 基本说明

数据类型	描述
Unit	表示无值，和其他语言中 void 等同。用作不返回任何结果的方法的结果类型。Unit 只有一个实例值，写成 ()。
Null	null，Null 类型只有一个实例值 null
Nothing	Nothing 类型在 Scala 的类层级最低端；它是任何其他类型的子类型。  当一个函数，我们确定没有正常的返回值，可以用 Nothing 来指定返回类型，这样有一个好处，就是我们可以把返回的值（异常）赋给其它的函数或者变量（兼容性）

### 2) 案例实操

（1）Unit 类型用来标识过程，也就是没有明确返回值的函数。

由此可见，Unit 类似于 Java 里的 void。Unit 只有一个实例——**()**，这个实例也没有实

质意义

```
object TestSpecialType {  
    def main(args: Array[String]): Unit = {  
        def sayOk : Unit = { // unit 表示没有返回值，即 void  
        }  
        println(sayOk)  
    }  
}
```

（2）Null 类只有一个实例对象，Null 类似于 Java 中的 null 引用。**Null 可以赋值给任**

**意引用类型（AnyRef），但是不能赋值给值类型（AnyVal）**

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



```
//null 可以赋值给任意引用类型 ( AnyRef ) , 但是不能赋值给值类型 ( AnyVal )
var cat = new Cat();
cat = null    // 正确

var n1: Int = null // 错误
println("n1:" + n1)

}
}

class Cat {
}
```

( 3 ) Nothing , 可以作为没有正常返回值的方法的返回类型 , 非常直观的告诉你这个方法不会正常返回 , 而且由于 Nothing 是其他任意类型的子类 , 他还能跟要求返回值的方法兼容。

```
object TestSpecialType {

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

    def test() : Nothing={
      throw new Exception()
    }
    test
  }
}
```

## 2.12 类型转换

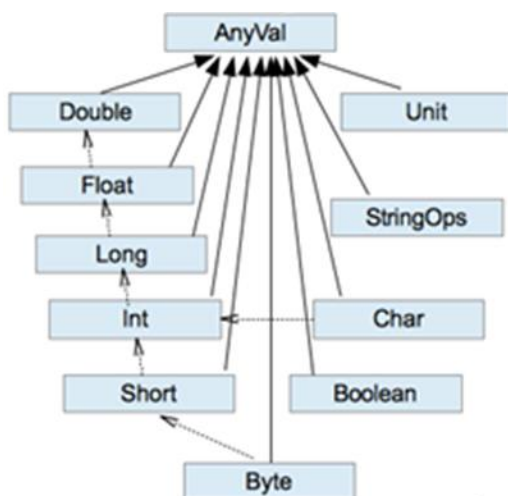
扩展 Java 面试题 ( 隐式类型转换 ) :

```
public static void main(String[] args) {  
    byte b = 10;  
    test(b);  
}  
  
public static void test(byte b) {  
    System.out.println("bbbb");  
}  
  
public static void test(short s) {  
    System.out.println("sssss");  
}  
  
public static void test(char c) {  
    System.out.println("cccc");  
}  
  
public static void test(int i) {  
    System.out.println("iiii");  
}
```

### 2.12.1 数值类型自动转换

当 Scala 程序在进行赋值或者运算时，精度小的类型自动转换为精度大的数值类型，这

个就是自动类型转换（隐式转换）。数据类型按精度（容量）大小排序为：



#### 1) 基本说明

(1) 自动提升原则：有多种类型的数据混合运算时，系统首先自动将所有数据转换成精度大的那种数据类型，然后再进行计算。

(2) 把精度大的数值类型赋值给精度小的数值类型时，就会报错，反之就会进行自动类型转换。

(3) (byte, short) 和 char 之间不会相互自动转换。

(4) byte, short, char 他们三者可以计算，在计算时首先转换为 int 类型。

## 2) 案例实操

```
object TestValueTransfer {  
    def main(args: Array[String]): Unit = {  
  
        // (1) 自动提升原则：有多种类型的数据混合运算时，系统首先自动将所有  
        // 数据转换成精度大的那种数值类型，然后再进行计算。  
        var n = 1 + 2.0  
        println(n) // n 就是 Double  
  
        // (2) 把精度大的数值类型赋值给精度小的数值类型时，就会报错，反之就会  
        // 进行自动类型转换。  
        var n2 : Double = 1.0  
        // var n3 : Int = n2 // 错误，原因不能把高精度的数据直接赋值和低  
        // 精度。  
  
        // (3) (byte, short) 和 char 之间不会相互自动转换。  
        var n4 : Byte = 1  
        // var c1 : Char = n4 // 错误  
        var n5 : Int = n4  
  
        // (4) byte, short, char 他们三者可以计算，在计算时首先转换为 int  
        // 类型。  
        var n6 : Byte = 1  
        var c2 : Char = 1  
        // var n : Short = n6 + c2 // 当 n6 + c2 结果类型就是 int  
        // var n7 : Short = 10 + 90 // 错误  
    }  
}
```

注意：Scala 还提供了非常强大的隐式转换机制（隐式函数，隐式类等），我们放在高级部分专门用一个章节来讲解。

## 2.12.2 强制类型转换

### 1) 基本说明

自动类型转换的逆过程，将精度大的数值类型转换为精度小的数值类型。使用时要加上强制转函数，但可能造成精度降低或溢出，格外要注意。

```
Java : int num = (int)2.5
Scala : var num : Int = 2.7.toInt
```

### 2) 案例实操

(1) 将数据由高精度转换为低精度，就需要使用到强制转换

(2) 强转符号只针对于最近的操作数有效，往往会使用小括号提升优先级

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

    // (1) 将数据由高精度转换为低精度，就需要使用到强制转换
    var n1: Int = 2.5.toInt // 这个存在精度损失

    // (2) 强转符号只针对于最近的操作数有效，往往会使用小括号提升优先级
    var r1: Int = 10 * 3.5.toInt + 6 * 1.5.toInt // 10 * 3 + 6 * 1
    = 36
    var r2: Int = (10 * 3.5 + 6 * 1.5).toInt // 44.0.toInt =
    44

    println("r1=" + r1 + " r2=" + r2)
  }
}
```

## 2.12.3 数值类型和 String 类型间转换

### 1) 基本说明

在程序开发中，我们经常需要将基本数值类型转成 String 类型。或者将 String 类型转成基本数值类型。

### 2) 案例实操

(1) 基本类型转 String 类型（语法：将基本类型的值+"" 即可）

(2) String 类型转基本数值类型（语法：s1.toInt、s1.toFloat、s1.toDouble、s1.toByte、

s1.toLong、s1.toShort )

```
object TestStringTransfer {  
    def main(args: Array[String]): Unit = {  
  
        // (1) 基本类型转 String 类型 (语法: 将基本类型的值+"" 即可)  
        var str1 : String = true + ""  
        var str2 : String = 4.5 + ""  
        var str3 : String = 100 + ""  
  
        // (2) String 类型转基本数值类型 (语法: 调用相关 API)  
        var s1 : String = "12"  
  
        var n1 : Byte = s1.toByte  
        var n2 : Short = s1.toShort  
        var n3 : Int = s1.toInt  
        var n4 : Long = s1.toLong  
    }  
}
```

### (3) 注意事项

在将 String 类型转成基本数值类型时, 要确保 String 类型能够转成有效的数据, 比如我们可以把"123", 转成一个整数, 但是不能把"hello"转成一个整数。

var n5:Int = "12.6".toInt 会出现 NumberFormatException 异常。

### 扩展面试题

```
object TestType {  
    def main(args: Array[String]): Unit = {  
  
        //00000000 00000000 00000000 10000010  
        //var n:Int = 128  
        var n:Int = 130  
        ~~~~~  
        var b:Byte = n.toByte  
        ~~~~~  
  
        //10000000 为当前字节范围最小值, 约定-128  
        //负数补码: 符号位不变, 其它位按位取反 + 1  
        println(b)  
    }  
}
```

## 第 3 章 运算符

Scala 运算符的使用和 Java 运算符的使用基本相同，只有个别细节上不同。

### 3.1 算术运算符

#### 1) 基本语法

运算符	运算	范例	结果
+	正号	+3	3
-	负号	b=4; -b	-4
+	加	5+5	10
-	减	6-4	2
*	乘	3*4	12
/	除	5/5	1
%	取模(取余)	7%5	2
+	字符串相加	"He"+"llo"	"Hello"

(1) 对于除号 `/`，它的整数除和小数除是有区别的：整数之间做除法时，只保留整数部分而舍弃小数部分。

(2) 对一个数取模 `a%b`，和 Java 的取模规则一样。

#### 2) 案例实操

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

    // (1) 对于除号 /，它的整数除和小数除是有区别的：整数之间做除法
    // 时，只保留整数部分而舍弃小数部分。
    var r1: Int = 10 / 3 // 3
    println("r1=" + r1)

    var r2: Double = 10 / 3 // 3.0
    println("r2=" + r2)

    var r3: Double = 10.0 / 3 // 3.3333
    println("r3=" + r3)
    println("r3=" + r3.formatted("%.2f")) // 含义：保留小数点 2
    // 位，使用四舍五入
  }
}
```

```
// (2) 对一个数取模 a%b, 和 Java 的取模规则一样。
var r4 = 10 % 3 // 1
println("r4=" + r4)
}
```

## 3.2 关系运算符 (比较运算符)

### 1) 基本语法

运算符	运算	范例	结果
==	相等于	4==3	false
!=	不等于	4!=3	true
<	小于	4<3	false
>	大于	4>3	true
<=	小于等于	4<=3	false
>=	大于等于	4>=3	true

### 2) 案例实操

#### (1) 需求 1:

```
object TestRelation {
  def main(args: Array[String]): Unit = {
    // 测试: >, >=, <=, <, ==, !=
    var a: Int = 2
    var b: Int = 1

    println(a > b)      // true
    println(a >= b)     // true
    println(a <= b)     // false
    println(a < b)      // false
    println("a==b" + (a == b)) // false
    println(a != b)     // true
  }
}
```

#### (2) 需求 2: Java 和 Scala 中关于==的区别

Java:

==比较两个变量本身值, 即两个对象在内存中的首地址;

equals 比较字符串中所包含的内容是否相同。

更多 Java -大数据 -前端 -python 人工智能资料下载, 可百度访问: 尚硅谷官网

```
public static void main(String[] args) {  
    String s1 = "abc";  
    String s2 = new String("abc");  
  
    System.out.println(s1 == s2);  
  
    System.out.println(s1.equals(s2));  
}
```

输出结果：

```
false  
true
```

Scala：==更加类似于 Java 中的 equals，参照 jd 工具

```
def main(args: Array[String]): Unit = {  
    val s1 = "abc"  
  
    val s2 = new String("abc")  
  
    println(s1 == s2)  
    println(s1.eq(s2))  
}
```

输出结果：

```
true  
false
```

### 3.3 逻辑运算符

#### 1) 基本语法

用于连接多个条件（一般来讲就是关系表达式），最终的结果也是一个 Boolean 值。

假定：变量 A 为 true，B 为 false

运算符	描述	实例
&&	逻辑与	(A && B) 运算结果为 false
	逻辑或	(A    B) 运算结果为 true
!	逻辑非	!(A && B) 运算结果为 true

#### 2) 案例实操

```
object TestLogic {  
    def main(args: Array[String]): Unit = {  
  
        // 测试：&&、||、!  
        var a = true
```



```

var b = false

println("a&&b=" + (a && b))    // a&&b=false
println("a||b=" + (a || b))    // a||b=true
println("!(a&&b)=" + (!(a && b))) // !(a&&b)=true
}
}

扩展避免逻辑与空指针异常
isEmpty(String s){
    //如果按位与，s 为空，会发生空指针
    return s!=null && !"".equals(s.trim());
}

```

## 3.4 赋值运算符

### 1) 基本语法

赋值运算符就是将某个运算后的值，赋给指定的变量。

运算符	描述	实例
=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C *= A 等于 C = C * A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2
&=	按位与后赋值	C &= 2 等于 C = C & 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
=	按位或后赋值	C  = 2 等于 C = C   2

注意：Scala 中没有++、--操作符，可以通过+=、-=来实现同样的效果；

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

## 2) 案例实操

```
object TestAssignment {  
    def main(args: Array[String]): Unit = {  
        var r1 = 10  
  
        r1 += 1 // 没有++  
        r1 -= 2 // 没有--  
    }  
}
```

## 3.5 位运算符

### 1) 基本语法

下表中变量 a 为 60, b 为 13。

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 , 二进制解释: 0000 1100
	按位或运算符	(a   b) 输出结果 61 , 二进制解释: 0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49 , 二进制解释: 0011 0001
~	按位取反运算符	(~a) 输出结果 -61 , 二进制解释: 1100 0011 , 在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240 , 二进制解释: 0011 0000
>>	右移动运算符	a >> 2 输出结果 15 , 二进制解释: 0000 1111
>>>	无符号右移	a >>> 2 输出结果 15, 二进制解释: 0000 1111

## 2) 案例实操

```
object TestPosition {  
    def main(args: Array[String]): Unit = {  
  
        // 测试: 1000 << 1 => 10000  
        var n1 : Int = 8  
    }  
}
```

```
n1 = n1 << 1
println(n1)
}
}
```

### 3.6 Scala 运算符本质

在 Scala 中其实是没有运算符的，所有运算符都是方法。

- 1) 当调用对象的方法时，点.可以省略
- 2) 如果函数参数只有一个，或者没有参数，()可以省略

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

    // 标准的加法运算
    val i: Int = 1.+(1)

    // (1) 当调用对象的方法时，.可以省略
    val j: Int = 1 + (1)

    // (2) 如果函数参数只有一个，或者没有参数，()可以省略
    val k: Int = 1 + 1

    println(1.toString())
    println(1 toString())
    println(1 toString)
  }
}
```

## 第 4 章 流程控制

### 4.1 分支控制 if-else

让程序有选择的执行，分支控制有三种：单分支、双分支、多分支

#### 4.1.1 单分支

- 1) 基本语法

```
if (条件表达式) {
  执行代码块
}
```

说明：当条件表达式为 true 时，就会执行{ }的代码。

- 2) 案例实操

需求：输入人的年龄，如果该同志的年龄小于 18 岁，则输出“童年”

```
object TestIfElse {  
    def main(args: Array[String]): Unit = {  
  
        println("input age:")  
        var age = StdIn.readShort()  
  
        if (age < 18){  
            println("童年")  
        }  
    }  
}
```

## 4.1.2 双分支

### 1) 基本语法

```
if (条件表达式) {  
    执行代码块 1  
} else {  
    执行代码块 2  
}
```

### 2) 案例实操

需求：输入年龄，如果年龄小于 18 岁，则输出“童年”。否则，输出“成年”。

```
object TestIfElse {  
    def main(args: Array[String]): Unit = {  
  
        println("input age:")  
        var age = StdIn.readShort()  
  
        if (age < 18){  
            println("童年")  
        }else{  
            println("成年")  
        }  
    }  
}
```

## 4.1.3 多分支

### 1) 基本语法

```
if (条件表达式 1) {  
    执行代码块 1  
}  
else if (条件表达式 2) {  
    执行代码块 2  
}
```

```

}

.....
else {
    执行代码块 n
}

```

## 2) 案例实操

(1) 需求 1：需求：输入年龄，如果年龄小于 18 岁，则输出“童年”。如果年龄大于等于 18 且小于等于 30，则输出“中年”，否则，输出“老年”。

```

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

        println("input age")
        var age = StdIn.readInt()

        if (age < 18){
            println("童年")
        }else if (age >= 18 && age < 30){
            println("中年")
        }else{
            println("老年")
        }
    }
}

```

(2) 需求 2：Scala 中 **if else 表达式其实是有返回值的**，具体返回值取决于满足条件的代码体的最后一行内容。

```

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

        println("input age")
        var age = StdIn.readInt()

        val res :String = if (age < 18){
            "童年"
        }else if (age >= 18 && age < 30){
            "中年"
        }else{
            "老年"
        }

        println(res)
    }
}

```

(3) 需求 3：Scala 中返回值类型不一致，取它们共同的祖先类型。

```

object TestIfElse {

```

```
def main(args: Array[String]): Unit = {  
    println("input age")  
    var age = StdIn.readInt()  
  
    val res:Any = if (age < 18){  
        "童年"  
    }else if(age>=18 && age<30){  
        "中年"  
    }else{  
        100  
    }  
  
    println(res)  
}
```

(4) 需求4: Java 中的三元运算符可以用 if else 实现

如果大括号{}内的逻辑代码只有一行,大括号可以省略。如果省略大括号,if 只对最近的一行逻辑代码起作用。

```
object TestIfElse {  
    def main(args: Array[String]): Unit = {  
        // Java  
        // int result = flag?1:0  
  
        // Scala  
        println("input age")  
        var age = StdIn.readInt()  
        val res:Any = if (age < 18) "童年" else "成年"  
        "不起作用"  
  
        println(res)  
    }  
}
```

## 4.2 嵌套分支

在一个分支结构中又完整的嵌套了另一个完整的分支结构,里面的分支的结构称为内层。

分支外面的分支结构称为外层分支。嵌套分支不要超过3层。

### 1) 基本语法

```
if(){  
    if(){  
  
    }else{
```

```
}  
}
```

## 2) 案例实操

需求：如果输入的年龄小于 18，返回“童年”。如果输入的年龄大于等于 18，需要再判

断：如果年龄大于等于 18 且小于 30，返回“中年”；如果其他，返回“老年”。

```
object TestIfElse {  
  def main(args: Array[String]): Unit = {  
  
    println("input age")  
    var age = StdIn.readInt()  
  
    val res :String = if (age < 18){  
      "童年"  
    }else {  
      if(age>=18 && age<30){  
        "中年"  
      }else{  
        "老年"  
      }  
    }  
  
    println(res)  
  }  
}
```

## 4.3 Switch 分支结构

在 Scala 中没有 Switch，而是使用模式匹配来处理。

模式匹配涉及到的知识点较为综合，因此我们放在后面讲解。

## 4.4 For 循环控制

Scala 也为 for 循环这一常见的控制结构提供了非常多的特性，这些 for 循环的特性被称为 for 推导式或 for 表达式。

### 4.4.1 范围数据循环 ( To )

#### 1) 基本语法

```
for(i <- 1 to 3){  
  print(i + " ")  
}  
println()
```

(1) i 表示循环的变量，<- 规定 to

(2) i 将会从 1-3 循环, 前后闭合

## 2) 案例实操

需求: 输出 5 句 "宋宋, 告别海狗人参丸吧"

```
object TestFor {  
    def main(args: Array[String]): Unit = {  
        for(i <- 1 to 5){  
            println("宋宋, 告别海狗人参丸吧"+i)  
        }  
    }  
}
```

## 4.4.2 范围数据循环 ( Until )

### 1) 基本语法

```
for(i <- 1 until 3) {  
    print(i + " ")  
}  
println()
```

(1) 这种方式和前面的区别在于 i 是从 1 到 3-1

(2) 即使前闭合后开的范围

## 2) 案例实操

需求: 输出 5 句 "宋宋, 告别海狗人参丸吧"

```
object TestFor {  
    def main(args: Array[String]): Unit = {  
        for(i <- 1 until 5 + 1){  
            println("宋宋, 告别海狗人参丸吧" + i)  
        }  
    }  
}
```

## 4.4.3 循环守卫

### 1) 基本语法

```
for(i <- 1 to 3 if i != 2) {  
    print(i + " ")  
}  
println()
```

说明:

更多 Java -大数据 -前端 -python 人工智能资料下载, 可百度访问: 尚硅谷官网



(1) 循环守卫，即循环保护式（也称条件判断式，守卫）。保护式为 true 则进入循环体内部，为 false 则跳过，类似于 continue。

(2) 上面的代码等价

```
for (i <- 1 to 3) {  
  if (i != 2) {  
    print(i + " ")  
  }  
}
```

## 2) 案例实操

需求：输出 1 到 5 中，不等于 3 的值

```
object TestFor {  
  
  def main(args: Array[String]): Unit = {  
  
    for (i <- 1 to 5 if i != 3) {  
      println(i + "宋宋")  
    }  
  }  
}
```

## 4.4.4 循环步长

### 1) 基本语法

```
for (i <- 1 to 10 by 2) {  
  println("i=" + i)  
}
```

说明：by 表示步长

### 2) 案例实操

需求：输出 1 到 10 以内的所有奇数

```
for (i <- 1 to 10 by 2) {  
  println("i=" + i)  
}
```

输出结果

```
i=1  
i=3  
i=5  
i=7  
i=9
```

## 4.4.5 嵌套循环

### 1) 基本语法

```
for(i <- 1 to 3; j <- 1 to 3) {  
    println(" i =" + i + " j = " + j)  
}
```

说明：没有关键字，所以范围后一定要加 **;** 来隔断逻辑

### 2) 基本语法

上面的代码等价

```
for (i <- 1 to 3) {  
    for (j <- 1 to 3) {  
        println("i =" + i + " j=" + j)  
    }  
}
```

## 4.4.6 引入变量

### 1) 基本语法

```
for(i <- 1 to 3; j = 4 - i) {  
    println("i=" + i + " j=" + j)  
}
```

**说明：**

(1) for 推导式一行中有多个表达式时，所以要加 **;** 来隔断逻辑

(2) for 推导式有一个不成文的约定：当 for 推导式仅包含单一表达式时使用圆括号，

当包含多个表达式时，一般每行一个表达式，并用花括号代替圆括号，如下

```
for {  
    i <- 1 to 3  
    j = 4 - i  
} {  
    println("i=" + i + " j=" + j)  
}
```

### 2) 案例实操

上面的代码等价于

```
for (i <- 1 to 3) {  
    var j = 4 - i  
    println("i=" + i + " j=" + j)  
}
```

## 4.4.7 循环返回值

### 1) 基本语法

```
val res = for(i <- 1 to 10) yield i  
println(res)
```

说明：将遍历过程中处理的结果返回到一个新 Vector 集合中，使用 yield 关键字。

**注意：开发中很少使用。**

### 2) 案例实操

需求：将原数据中所有值乘以 2，并把数据返回到一个新的集合中。

```
object TestFor {  
    def main(args: Array[String]): Unit = {  
        var res = for(i <- 1 to 10) yield {  
            i * 2  
        }  
        println(res)  
    }  
}
```

输出结果：

Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

## 4.4.8 倒序打印

1) 说明：如果想倒序打印一组数据，可以用 reverse。

2) 案例实操：

需求：倒序打印 10 到 1

```
for(i <- 1 to 10 reverse){  
    println(i)  
}
```

## 4.5 While 和 do..While 循环控制

While 和 do..While 的使用和 Java 语言中用法相同。

### 4.5.1 While 循环控制

#### 1) 基本语法

循环变量初始化

```
while (循环条件) {  
  
    循环体(语句)  
  
    循环变量迭代  
  
}
```

说明：

(1) 循环条件是返回一个布尔值的表达式

(2) while 循环是先判断再执行语句

(3) 与 for 语句不同，while 语句没有返回值，即整个 while 语句的结果是 Unit 类型()

(4) 因为 while 中没有返回值，所以当要用该语句来计算并返回结果时，就不可避免的使用变量，而变量需要声明在 while 循环的外部，那么就等同于循环的内部对外部的变量造成了影响，所以不推荐使用，而是推荐使用 for 循环。

## 2) 案例实操

需求：输出 10 句 "宋宋，喜欢海狗人参丸"

```
object TestWhile {  
  
    def main(args: Array[String]): Unit = {  
  
        var i = 0  
  
        while (i < 10) {  
            println("宋宋，喜欢海狗人参丸" + i)  
            i += 1  
        }  
    }  
}
```

## 4.5.2 do..while 循环控制

### 1) 基本语法

循环变量初始化;

```
do{
```

[更多 Java](#) -[大数据](#) -[前端](#) -[python](#) [人工智能资料下载](#)，可百度访问：尚硅谷官网

循环体(语句)

循环变量迭代

} while(循环条件)

说明

(1) 循环条件是返回一个布尔值的表达式

(2) do..while 循环是先执行，再判断

## 2) 案例实操

需求：输出 10 句 "宋宋，喜欢海狗人参丸"

```
object TestWhile {  
    def main(args: Array[String]): Unit = {  
        var i = 0  
        do {  
            println("宋宋，喜欢海狗人参丸" + i)  
            i += 1  
        } while (i < 10)  
    }  
}
```

## 4.6 循环中断

### 1) 基本说明

Scala 内置控制结构特地**去掉了 break 和 continue**，是为了更好的适应[函数式编程](#)，推荐使用函数式的风格解决 break 和 continue 的功能，而不是一个关键字。Scala 中使用 breakable 控制结构来实现 break 和 continue 功能。

### 2) 案例实操

需求 1：采用异常的方式退出循环

```
def main(args: Array[String]): Unit = {  
    try {  
        for (elem <- 1 to 10) {  
            println(elem)  
            if (elem == 5) throw new RuntimeException  
        }  
    }  
}
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
    }  
  } catch {  
    case e =>  
  }  
  println("正常结束循环")  
}
```

需求 2：采用 Scala 自带的函数，退出循环

```
import scala.util.control.Breaks  
  
def main(args: Array[String]): Unit = {  
  Breaks.breakable(  
    for (elem <- 1 to 10) {  
      println(elem)  
      if (elem == 5) Breaks.break()  
    }  
  )  
  
  println("正常结束循环")  
}
```

需求 3：对 break 进行省略

```
import scala.util.control.Breaks._  
  
object TestBreak {  
  def main(args: Array[String]): Unit = {  
    breakable {  
      for (elem <- 1 to 10) {  
        println(elem)  
        if (elem == 5) break  
      }  
    }  
  
    println("正常结束循环")  
  }  
}
```

需求 4：循环遍历 10 以内的所有数据，奇数打印，偶数跳过（continue）

```
object TestBreak {  
  def main(args: Array[String]): Unit = {  
    for (elem <- 1 to 10) {  
      if (elem % 2 == 1) {  
        println(elem)  
      } else {  
        println("continue")  
      }  
    }  
  }  
}
```

## 4.7 多重循环

### 1) 基本说明

(1) 将一个循环放在另一个循环体内,就形成了嵌套循环。其中,for,while,do...while 均可以作为外层循环和内层循环。【**建议一般使用两层,最多不要超过3层**】

(2) 设外层循环次数为 m 次,内层为 n 次,则内层循环体实际上需要执行  $m*n$  次。

### 2) 案例实操

需求:打印出九九乘法表

```
object TestWhile {
    def main(args: Array[String]): Unit = {
        for (i <- 1 to 9) {
            for (j <- 1 to i) {
                print(j + "*" + i + "=" + (i * j) + "\t")
            }
            println()
        }
    }
}
```

输出结果:

```
1*1=1
1*2=2      2*2=4
1*3=3      2*3=6      3*3=9
1*4=4      2*4=8      3*4=12      4*4=16
1*5=5      2*5=10     3*5=15      4*5=20      5*5=25
1*6=6      2*6=12     3*6=18      4*6=24      5*6=30      6*6=36
1*7=7      2*7=14     3*7=21      4*7=28      5*7=35      6*7=42      7*7=49
1*8=8      2*8=16     3*8=24      4*8=32      5*8=40      6*8=48      7*8=56      8*8=64
1*9=9      2*9=18     3*9=27      4*9=36      5*9=45      6*9=54      7*9=63      8*9=72      9*9=81
```

## 第5章 函数式编程

### 1) 面向对象编程

解决问题,分解对象,行为,属性,然后通过对象的关系以及行为的调用来解决问题。

对象:用户

行为:登录、连接 JDBC、读取数据库

属性:用户名、密码

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问:尚硅谷官网

Scala 语言是一个完全面向对象编程语言。万物皆对象

对象的本质：对数据和行为的一个封装

## 2) 函数式编程

解决问题时，将问题分解成一个一个的步骤，将每个步骤进行封装（函数），通过调用这些封装好的步骤，解决问题。

例如：请求->用户名、密码->连接 JDBC->读取数据库

Scala 语言是一个完全函数式编程语言。万物皆函数。

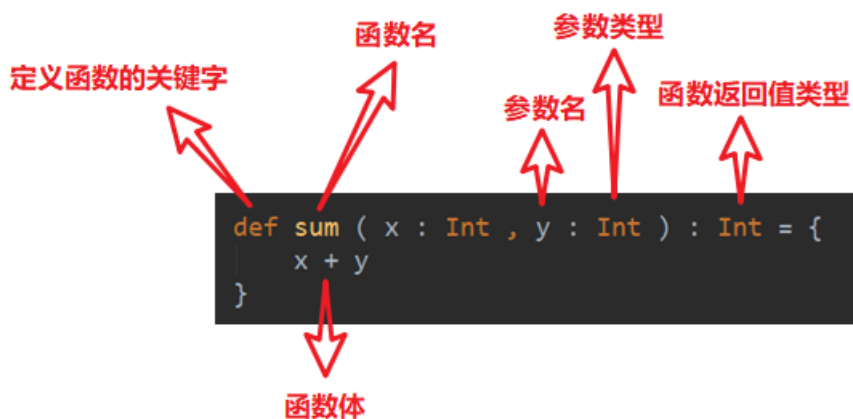
函数的本质：函数可以当做一个值进行传递

3) 在 Scala 中函数式编程和面向对象编程完美融合在一起了。

## 5.1 函数基础

### 5.1.1 函数基本语法

#### 1) 基本语法



#### 2) 案例实操

需求：定义一个函数，实现将传入的名称打印出来。

```
object TestFunction {

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

    // (1) 函数定义
    def f(arg: String): Unit = {
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网



```
println(arg)
}

// (2) 函数调用
// 函数名 (参数)
f("hello world")
}
}
```

### 5.1.2 函数和方法的区别

#### 1) 核心概念

(1) 为完成某一功能的程序语句的集合，称为函数。

(2) 类中的函数称之为方法。

#### 2) 案例实操

(1) Scala 语言可以在任何的语法结构中声明任何的语法

(2) 函数没有重载和重写的概念；方法可以进行重载和重写

(3) Scala 中函数可以嵌套定义

```
object TestFunction {

    // (2) 方法可以进行重载和重写，程序可以执行
    def main(): Unit = {

    }

    def main(args: Array[String]): Unit = {
        // (1) Scala 语言可以在任何的语法结构中声明任何的语法
        import java.util.Date
        new Date()

        // (2) 函数没有重载和重写的概念，程序报错
        def test(): Unit = {
            println("无参，无返回值")
        }
        test()

        def test(name: String): Unit = {
            println()
        }

        // (3) Scala 中函数可以嵌套定义
        def test2(): Unit = {
```

```
def test3(name:String):Unit={  
    println("函数可以嵌套定义")  
}  
}  
}  
}
```

### 5.1.3 函数定义

#### 1) 函数定义

(1) 函数 1：无参，无返回值

(2) 函数 2：无参，有返回值

(3) 函数 3：有参，无返回值

(4) 函数 4：有参，有返回值

(5) 函数 5：多参，无返回值

(6) 函数 6：多参，有返回值

#### 2) 案例实操

```
package com.atguigu.chapter05  
  
object TestFunctionDeclare {  
  
    def main(args: Array[String]): Unit = {  
  
        // 函数 1：无参，无返回值  
        def test1(): Unit = {  
            println("无参，无返回值")  
        }  
        test1()  
  
        // 函数 2：无参，有返回值  
        def test2(): String = {  
            return "无参，有返回值"  
        }  
        println(test2())  
  
        // 函数 3：有参，无返回值  
        def test3(s:String):Unit={  
            println(s)  
        }  
        test3("jinlian")  
    }  
}
```

```
// 函数 4：有参，有返回值
def test4(s:String):String={
    return s+"有参，有返回值"
}
println(test4("hello "))

// 函数 5：多参，无返回值
def test5(name:String, age:Int):Unit={
    println(s"$name, $age")
}
test5("dalang",40)
}
```

### 5.1.4 函数参数

#### 1) 案例实操

(1) 可变参数

(2) 如果参数列表中存在多个参数，那么可变参数一般放置在最后

(3) 参数默认值，一般将有默认值的参数放置在参数列表的后面

(4) 带名参数

```
object TestFunction {

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

        // (1) 可变参数
        def test(s: String*): Unit = {
            println(s)
        }

        // 有输入参数：输出 Array
        test("Hello", "Scala")

        // 无输入参数：输出 List()
        test()

        // (2) 如果参数列表中存在多个参数，那么可变参数一般放置在最后
        def test2(name: String, s: String*): Unit = {
            println(name + ", " + s)
        }

        test2("jinlian", "dalang")

        // (3) 参数默认值
        def test3(name: String, age: Int = 30): Unit = {
```

```
println(s"$name, $age")
}

// 如果参数传递了值，那么会覆盖默认值
test3("jinlian", 20)

// 如果参数有默认值，在调用的时候，可以省略这个参数
test3("dalang")

// 一般情况下，将有默认值的参数放置在参数列表的后面
def test4( sex : String = "男", name : String ) : Unit = {
    println(s"$name, $sex")
}
// Scala 函数中参数传递是，从左到右
//test4("wusong")

// ( 4 ) 带名参数
test4(name="ximenqing")
}
```

### 5.1.5 函数至简原则（重点）

函数至简原则：能省则省

#### 1) 至简原则细节

- ( 1 ) return 可以省略，Scala 会使用函数体的最后一行代码作为返回值
- ( 2 ) 如果函数体只有一行代码，可以省略花括号
- ( 3 ) 返回值类型如果能够推断出来，那么可以省略（:和返回值类型一起省略）
- ( 4 ) 如果有 return，则不能省略返回值类型，必须指定
- ( 5 ) 如果函数明确声明 unit，那么即使函数体中使用 return 关键字也不起作用
- ( 6 ) Scala 如果期望是无返回值类型，可以省略等号
- ( 7 ) 如果函数无参，但是声明了参数列表，那么调用时，小括号，可加可不加
- ( 8 ) 如果函数没有参数列表，那么小括号可以省略，调用时小括号必须省略
- ( 9 ) 如果不关心名称，只关心逻辑处理，那么函数名（def）可以省略

#### 2) 案例实操

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
object TestFunction {  
  
    def main(args: Array[String]): Unit = {  
  
        // (0) 函数标准写法  
        def f(s: String): String = {  
            return s + " jinlian"  
        }  
        println(f("Hello"))  
  
        // 至简原则:能省则省  
  
        //(1) return 可以省略, Scala 会使用函数体的最后一行代码作为返回值  
        def f1(s: String): String = {  
            s + " jinlian"  
        }  
        println(f1("Hello"))  
  
        //(2) 如果函数体只有一行代码, 可以省略花括号  
        def f2(s:String):String = s + " jinlian"  
  
        //(3) 返回值类型如果能够推断出来, 那么可以省略(:和返回值类型一起省略)  
        def f3(s: String) = s + " jinlian"  
        println(f3("Hello3"))  
  
        //(4) 如果有 return, 则不能省略返回值类型, 必须指定。  
        def f4():String = {  
            return "ximenqing4"  
        }  
        println(f4())  
  
        //(5) 如果函数明确声明 unit, 那么即使函数体中使用 return 关键字也不起作用  
        def f5(): Unit = {  
            return "dalang5"  
        }  
        println(f5())  
  
        //(6) Scala 如果期望是无返回值类型, 可以省略等号  
        // 将无返回值的函数称之为过程  
        def f6() {  
            "dalang6"  
        }  
        println(f6())  
  
        //(7) 如果函数无参, 但是声明了参数列表, 那么调用时, 小括号, 可加可不加  
        def f7() = "dalang7"  
        println(f7())  
    }  
}
```

```
println(f7)

// (8) 如果函数没有参数列表, 那么小括号可以省略, 调用时小括号必须省略
def f8 = "dalang"
//println(f8())
println(f8)

// (9) 如果不关心名称, 只关心逻辑处理, 那么函数名 (def) 可以省略
def f9 = (x:String)=>{println("wusong")}

def f10(f:String=>Unit) = {
    f("")
}

f10(f9)
println(f10((x:String)=>{println("wusong")}))
}
```

## 5.2 函数高级

### 5.2.1 高阶函数

在 Scala 中, 函数是一等公民。怎么体现的呢?

对于一个函数我们可以: **定义函数**、**调用函数**

```
object TestFunction {

    def main(args: Array[String]): Unit = {
        // 调用函数
        foo()
    }

    // 定义函数
    def foo():Unit = {
        println("foo...")
    }
}
```

但是其实函数还有更高阶的用法

#### 1) 函数可以作为值进行传递

```
object TestFunction {

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

        // (1) 调用 foo 函数, 把返回值给变量 f
        //val f = foo()
        val f = foo
        println(f)
    }
}
```

```
// (2) 在被调用函数 foo 后面加上 _ , 相当于把函数 foo 当成一个整体 ,
传递给变量 f1
val f1 = foo _

foo()
f1()

// (3) 如果明确变量类型 , 那么不使用下划线也可以将函数作为整体传递给
变量
var f2: () => Int = foo
}

def foo(): Int = {
  println("foo...")
  1
}
}
```

## 2) 函数可以作为参数进行传递

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

  // (1) 定义一个函数 , 函数参数还是一个函数签名 ; f 表示函数名称 ; (Int, Int)
  表示输入两个 Int 参数 ; Int 表示函数返回值
  def f1(f: (Int, Int) => Int): Int = {
    f(2, 4)
  }

  // (2) 定义一个函数 , 参数和返回值类型和 f1 的输入参数一致
  def add(a: Int, b: Int): Int = a + b

  // (3) 将 add 函数作为参数传递给 f1 函数 , 如果能够推断出来不是调用 , _
  可以省略
  println(f1(add))
  println(f1(add _))
  // 可以传递匿名函数
}
}
```

## 3) 函数可以作为函数返回值返回

```
def main(args: Array[String]): Unit = {
  def f1() = {
    def f2() = {

    }
    f2 _
  }

  val f = f1()
  // 因为 f1 函数的返回值依然为函数 , 所以可以变量 f 可以作为函数继续调用
  f()
  // 上面的代码可以简化为
  f1() ()
}
```

```
}
```

## 5.2.2 匿名函数

### 1) 说明

没有名字的函数就是匿名函数。

`(x:Int)=>{函数体}`

`x`：表示输入参数类型；`Int`：表示输入参数类型；函数体：表示具体代码逻辑

### 2) 案例实操

需求 1：传递的函数有一个参数

传递匿名函数至简原则：

(1) 参数的类型可以省略，会根据形参进行自动的推导

(2) 类型省略之后，发现只有一个参数，则圆括号可以省略；其他情况：没有参数和参数超过 1 的永远不能省略圆括号。

(3) 匿名函数如果只有一行，则大括号也可以省略

(4) 如果参数只出现一次，则参数省略且后面参数可以用 `_` 代替

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

    // (1) 定义一个函数：参数包含数据和逻辑函数
    def operation(arr: Array[Int], op: Int => Int) = {
        for (elem <- arr) yield op(elem)
    }

    // (2) 定义逻辑函数
    def op(ele: Int): Int = {
        ele + 1
    }

    // (3) 标准函数调用
    val arr = operation(Array(1, 2, 3, 4), op)
    println(arr.mkString(","))

    // (4) 采用匿名函数
    val arr1 = operation(Array(1, 2, 3, 4), (ele: Int) => {
        ele + 1
    })
}
```



```
println(arr1.mkString(", "))

// (4.1) 参数的类型可以省略, 会根据形参进行自动的推导;
val arr2 = operation(Array(1, 2, 3, 4), (ele) => {
    ele + 1
})
println(arr2.mkString(", "))

// (4.2) 类型省略之后, 发现只有一个参数, 则圆括号可以省略; 其他情况: 没有参数和参数超过 1 的永远不能省略圆括号。
val arr3 = operation(Array(1, 2, 3, 4), ele => {
    ele + 1
})
println(arr3.mkString(", "))

// (4.3) 匿名函数如果只有一行, 则大括号也可以省略
val arr4 = operation(Array(1, 2, 3, 4), ele => ele + 1)
println(arr4.mkString(", "))

// (4.4) 如果参数只出现一次, 则参数省略且后面参数可以用_代替
val arr5 = operation(Array(1, 2, 3, 4), _ + 1)
println(arr5.mkString(", "))
}
```

需求 2: 传递的函数有两个参数

```
object TestFunction {

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

        def calculator(a: Int, b: Int, op: (Int, Int) => Int): Int = {
            op(a, b)
        }

        // (1) 标准版
        println(calculator(2, 3, (x: Int, y: Int) => {x + y}))

        // (2) 如果只有一行, 则大括号也可以省略
        println(calculator(2, 3, (x: Int, y: Int) => x + y))

        // (3) 参数的类型可以省略, 会根据形参进行自动的推导;
        println(calculator(2, 3, (x, y) => x + y))

        // (4) 如果参数只出现一次, 则参数省略且后面参数可以用_代替
        println(calculator(2, 3, _ + _))
    }
}
```

### 扩展练习

练习 1: 定义一个匿名函数, 并将它作为值赋给变量 fun。函数有三个参数, 类型分别更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) [人工智能资料下载](#), 可百度访问: [尚硅谷官网](#)

为 Int, String, Char, 返回值类型为 Boolean。

要求调用函数 `fun(0, "", '0')` 得到返回值为 false, 其它情况均返回 true。

练习 2: 定义一个函数 `func`, 它接收一个 Int 类型的参数, 返回一个函数 (记作 `f1`)。

它返回的函数 `f1`, 接收一个 String 类型的参数, 同样返回一个函数 (记作 `f2`)。函数 `f2` 接收一个 Char 类型的参数, 返回一个 Boolean 的值。

要求调用函数 `func(0)("")('0')` 得到返回值为 false, 其它情况均返回 true。

## 5.2.3 高阶函数案例

需求: 模拟 Map 映射、Filter 过滤、Reduce 聚合

```
object TestFunction {

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

    // (1) map 映射
    def map(arr: Array[Int], op: Int => Int) = {
      for (elem <- arr) yield op(elem)
    }

    val arr = map(Array(1, 2, 3, 4), (x: Int) => {
      x * x
    })
    println(arr.mkString(","))

    // (2) filter 过滤。有参数, 且参数再后面只使用一次, 则参数省略且
    后面参数用_表示
    def filter(arr: Array[Int], op: Int => Boolean) = {
      var arr1: ArrayBuffer[Int] = ArrayBuffer[Int]()
      for (elem <- arr if op(elem)) {
        arr1.append(elem)
      }
      arr1.toArray
    }
    var arr1 = filter(Array(1, 2, 3, 4), _ % 2 == 1)
    println(arr1.mkString(","))

    // (3) reduce 聚合。有多个参数, 且每个参数再后面只使用一次, 则参
    数省略且后面参数用_表示, 第 n 个_代表第 n 个参数
    def reduce(arr: Array[Int], op: (Int, Int) => Int) = {
```

```

var init: Int = arr(0)

for (elem <- 1 until arr.length) {
    init = op(init, elem)
}
init
}

//val arr2 = reduce(Array(1, 2, 3, 4), (x, y) => x * y)
val arr2 = reduce(Array(1, 2, 3, 4), _ * _)
println(arr2)
}
}

```

## 5.2.4 函数柯里化&闭包

闭包：函数式编程的标配

### 1) 说明

**闭包**：如果一个函数，访问到了它的外部（局部）变量的值，那么这个函数和他所处的

环境，称为闭包

**函数柯里化**：把一个参数列表的多个参数，变成多个参数列表。

### 2) 案例实操

#### (1) 闭包

```

object TestFunction {
    def main(args: Array[String]): Unit = {
        def f1()={
            var a:Int = 10
            def f2(b:Int)={
                a + b
            }
            f2 _
        }

        // 在调用时，f1 函数执行完毕后，局部变量 a 应该随着栈空间释放掉
        val f = f1()

        // 但是在此处，变量 a 其实并没有释放，而是包含在了 f2 函数的内部，形
        // 成了闭合的效果
        println(f(3))

        println(f1()(3))
    }
}

```

闭包：方法在栈中，方法弹出栈，变量也会消失，所以在执行最内层函数时，外层函数的变量已经没了，但是最内层函数依赖的变量会打包储存在堆中，称为闭包

递归：f(5)先入栈，f(4)入栈最后f(1)入栈，最后先进后出（这种递归如果数值过大可能栈溢出）

尾递归：原理（最外层函数的返回值不需要计算，这样栈中每次调用方法都是覆盖）

```
// 函数柯里化，其实就是将复杂的参数逻辑变得简单化，函数柯里化一定存在闭包
def f3() (b:Int)={
    a + b
}

println(f3() (3))
}
```

### 5.2.5 递归

#### 1) 说明

一个函数/方法在函数/方法体内又调用了本身，我们称之为递归调用

#### 2) 案例实操

```
object TestFunction {

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

        // 阶乘
        // 递归算法
        // 1) 方法调用自身
        // 2) 方法必须要有跳出的逻辑
        // 3) 方法调用自身时，传递的参数应该有规律
        // 4) scala 中的递归必须声明函数返回值类型

        println(test(5))
    }

    def test(i : Int) : Int = {
        if (i == 1) {
            1
        } else {
            i * test(i - 1)
        }
    }
}
```

### 5.2.6 控制抽象

#### 1) 值调用：把计算后的值传递过去

```
object TestControl {

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

        def f = ()=>{
            println("f...")
        }
    }
}
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
        10
    }

    foo(f())
}

def foo(a: Int):Unit = {
    println(a)
    println(a)
}
}
```

## 2) 名调用：把代码传递过去

```
object TestControl {

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

        def f = ()=>{
            println("f...")
            10
        }

        foo(f())
    }

    //def foo(a: Int):Unit = {
    def foo(a: =>Int):Unit = {//注意这里变量 a 没有小括号了
        println(a)
        println(a)
    }
}
```

输出结果：

```
f...
10
f...
10
```

注意：Java 只有值调用；Scala 既有值调用，又有名调用。

## 3) 案例实操

```
object TestFunction {

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

        // (1) 传递代码块
        foo({
            println("aaa")
        })

        // (2) 小括号可以省略
        foo{
            println("aaa")
        }
    }
}
```

```
def foo(a: =>Unit):Unit = {  
    println(a)  
    println(a)  
}  
}
```

自定义一个 While 循环

```
object TestFunction {  
  
    def main(args: Array[String]): Unit = {  
  
        var i:Int = 1  
        myWhile(i <= 10){  
            println(i)  
            i +=1  
        }  
    }  
  
    def myWhile(condition: =>Boolean) (op: =>Unit):Unit={  
  
        if (condition){  
            op  
            myWhile(condition) (op)  
        }  
    }  
}
```

## 5.2.7 惰性加载

### 1) 说明

当函数返回值被声明为 **lazy** 时，函数的执行将被推迟，直到我们首次对此取值，该函数才会执行。这种函数我们称之为惰性函数。

### 2) 案例实操

```
def main(args: Array[String]): Unit = {  
  
    lazy val res = sum(10, 30)  
    println("-----")  
    println("res=" + res)  
}  
  
def sum(n1: Int, n2: Int): Int = {  
    println("sum 被执行。。。")  
    return n1 + n2  
}
```

输出结果：

```
-----  
sum 被执行。。。 
```

```
res=40
```

注意：lazy 不能修饰 var 类型的变量

## 第 6 章 面向对象

Scala 的面向对象思想和 Java 的面向对象思想和概念是一致的。

Scala 中语法和 Java 不同，补充了更多的功能。

### 6.1 Scala 包

#### 1) 基本语法

package 包名

#### 2) Scala 包的三大作用（和 Java 一样）

（1）区分相同名字的类

（2）当类很多时，可以很好的管理类

（3）控制访问范围

#### 6.1.1 包的命名

##### 1) 命名规则

只能包含数字、字母、下划线、小圆点，但不能用数字开头，也不要使用关键字。

##### 2) 案例实操

```
demo.class.exec1 //错误，因为 class 关键字  
demo.12a         //错误，数字开头
```

##### 3) 命名规范

一般是小写字母+小圆点

**com.公司名.项目名.业务模块名**

##### 4) 案例实操

```
com.atguigu.oa.model  
com.atguigu.oa.controller  
com.sohu.bank.order
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

## 6.1.2 包说明（包语句）

### 1) 说明

Scala 有两种包的管理风格，一种方式和 Java 的包管理风格相同，每个源文件一个包（包名和源文件所在路径不要求必须一致），包名用 “.” 进行分隔以表示包的层级关系，如 com.atguigu.scala。另一种风格，通过嵌套的风格表示层级关系，如下

```
package com{
  package atguigu{
    package scala{

    }
  }
}
```

第二种风格有以下特点：

- (1) 一个源文件中可以声明多个 package
- (2) 子包中的类可以直接访问父包中的内容，而无需导包

### 2) 案例实操

```
package com {

  import com.atguigu.Inner //父包访问子包需要导包

  object Outer {
    val out: String = "out"

    def main(args: Array[String]): Unit = {
      println(Inner.in)
    }
  }

  package atguigu {

    object Inner {
      val in: String = "in"

      def main(args: Array[String]): Unit = {
        println(Outer.out) //子包访问父包无需导包
      }
    }
  }
}

package other {
```



}

### 6.1.3 包对象

在 Scala 中可以为每个包定义一个同名的包对象，定义在包对象中的成员，作为其对应包下所有 class 和 object 的共享变量，可以被直接访问。

#### 1) 定义

```
package object com{
    val shareValue="share"
    def shareMethod()={}
}
```

#### 1) 说明

(1) 若使用 Java 的包管理风格, 则包对象一般定义在其对应包下的 package.scala 文件中, 包对象名与包名保持一致。



(2) 如采用嵌套方式管理包, 则包对象可与包定义在同一文件中, 但是要保证包对象与包声明在同一作用域中。

```
package com {

    object Outer {
        val out: String = "out"

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

}

package object com {
```

```
val name: String = "com"
}
```

### 6.1.4 导包说明

1) 和 Java 一样，可以在顶部使用 import 导入，在这个文件中的所有类都可以使用。

2) 局部导入：什么时候使用，什么时候导入。**在其作用范围内都可以使用**

3) 通配符导入：import java.util.\_

4) 给类起名：import java.util.{ArrayList=>JL}

5) 导入**相同包的**多个类：import java.util.{HashSet, ArrayList}

6) 屏蔽类：import java.util.{ArrayList => \_, \_}

7) 导入包的绝对路径：new \_root\_.java.util.HashMap

```
package java {
  package util {
    class HashMap {
      }
  }
}
```

#### 说明

import com.atguigu.Fruit	引入 com.atguigu 包下 Fruit ( class 和 object )
import com.atguigu._	引入 com.atguigu 下的所有成员
import com.atguigu.Fruit._	引入 Fruit(object)的所有成员
import com.atguigu.{Fruit,Vegetable}	引入 com.atguigu 下的 Fruit 和 Vegetable
import com.atguigu.{Fruit=>Shuiguo}	引入 com.atguigu 包下的 Fruit 并更名为 Shuiguo
import com.atguigu.{Fruit=>Shuiguo,_}	引入 com.atguigu 包下的所有成员,并将 Fruit 更名为 Shuiguo
import com.atguigu.{Fruit=>_,_}	引入 com.atguigu 包下屏蔽 Fruit 类
new _root_.java.util.HashMap	引入的 Java 的绝对路径

## 2) 注意

Scala 中的三个默认导入分别是

```
import java.lang._
```

```
import scala._
```

```
import scala.Predef._
```

## 6.2 类和对象

类：可以看成是一个模板

对象：表示具体的事物

### 6.2.1 定义类

#### 1) 回顾：Java 中的类

如果类是 public 的，则必须和文件名一致。

一般，一个.java 有一个 public 类

**注意：Scala 中没有 public，一个.scala 中可以写多个类。**

#### 1) 基本语法

```
[修饰符] class 类名 {
```

```
    类体
```

```
}
```

说明

(1) Scala 语法中，类并不声明为 public，所有这些类都具有公有可见性（即默认就是 public）

(2) 一个 Scala 源文件可以包含多个类

#### 2) 案例实操

```
package com.atguigu.chapter06
```

```
// (1) Scala 语法中，类并不声明为 public，所有这些类都具有公有可见性（即默认就是 public）
class Person {

}

// (2) 一个 Scala 源文件可以包含多个类
class Teacher{

}
```

## 6.2.2 属性

属性是类的一个组成部分

### 1) 基本语法

**[修饰符] var|val 属性名称 [: 类型] = 属性值**

**注：**Bean 属性（@BeanProperty），可以自动生成规范的 setXxx/getXxx 方法

### 2) 案例实操

```
package com.atguigu.scala.test

import scala.beans.BeanProperty

class Person {

    var name: String = "bobo" //定义属性

    var age: Int = _ // _表示给属性一个默认值

    //Bean 属性（@BeanProperty）
    @BeanProperty var sex: String = "男"
    //val 修饰的属性不能赋默认值，必须显示指定
}

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

        var person = new Person()
        println(person.name)

        person.setSex("女")
        println(person.getSex)
    }
}
```

## 6.3 封装

封装就是把抽象出的数据和对数据的操作封装在一起，数据被保护在内部，程序的其它部分只有通过被授权的操作（成员方法），才能对数据进行操作。Java 封装操作如下，

- （1）将属性进行私有化
- （2）提供一个公共的 set 方法，用于对属性赋值
- （3）提供一个公共的 get 方法，用于获取属性的值

Scala 中的 `public` 属性，底层实际为 `private`，并通过 `get` 方法（`obj.field()`）和 `set` 方法（`obj.field_=(value)`）对其进行操作。所以 Scala 并不推荐将属性设为 `private`，再为其设置 `public` 的 `get` 和 `set` 方法的做法。但由于很多 Java 框架都利用反射调用 `getXXX` 和 `setXXX` 方法，有时候为了和这些框架兼容，也会为 Scala 的属性设置 `getXXX` 和 `setXXX` 方法（通过 `@BeanProperty` 注解实现）。

### 6.1.5 访问权限

#### 1) 说明

在 Java 中，访问权限分为：`public`，`private`，`protected` 和默认。在 Scala 中，你可以通过类似的修饰符达到同样的效果。但是使用上有区别。

- （1）Scala 中属性和方法的默认访问权限为 `public`，但 Scala 中无 `public` 关键字。
- （2）`private` 为私有权限，只在类的内部和伴生对象中可用。
- （3）`protected` 为受保护权限，Scala 中受保护权限比 Java 中更严格，同类、子类可以访问，同包无法访问。
- （4）`private[包名]` 增加包访问权限，包名下的其他类也可以使用

#### 2) 案例实操

```
package com.atguigu.scala.test
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
class Person {  
    private var name: String = "bobo"  
    protected var age: Int = 18  
    private[test] var sex: String = "男"  
  
    def say(): Unit = {  
        println(name)  
    }  
}  
  
object Person {  
    def main(args: Array[String]): Unit = {  
        val person = new Person  
        person.say()  
        println(person.name)  
        println(person.age)  
    }  
}  
  
class Teacher extends Person {  
    def test(): Unit = {  
        this.age  
        this.sex  
    }  
}  
  
class Animal {  
    def test: Unit = {  
        new Person().sex  
    }  
}
```

### 6.2.3 方法

#### 1) 基本语法

```
def 方法名(参数列表) [ : 返回值类型] = {  
  
    方法体  
  
}
```

#### 2) 案例实操

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
class Person {  
    def sum(n1:Int, n2:Int) : Int = {  
        n1 + n2  
    }  
}  
  
object Person {  
    def main(args: Array[String]): Unit = {  
        val person = new Person()  
        println(person.sum(10, 20))  
    }  
}
```

## 6.2.4 创建对象

### 1) 基本语法

**val | var 对象名 [ : 类型 ] = new 类型()**

### 2) 案例实操

(1) val 修饰对象，不能改变对象的引用（即：内存地址），可以改变对象属性的值。

(2) var 修饰对象，可以修改对象的引用和修改对象的属性值

(3) 自动推导变量类型不能多态，所以多态需要显示声明

```
class Person {  
    var name: String = "canglaoshi"  
}  
  
object Person {  
    def main(args: Array[String]): Unit = {  
        //val 修饰对象，不能改变对象的引用（即：内存地址），可以改变对象属  
        性的值。  
        val person = new Person()  
        person.name = "bobo"  
  
        // person = new Person() // 错误的  
  
        println(person.name)  
    }  
}
```

## 6.2.5 构造器

和 Java 一样，Scala 构造对象也需要调用构造方法，并且可以有任意多个构造方法。

Scala 类的构造器包括：**主构造器和辅助构造器**

### 1) 基本语法

```
class 类名(形参列表) { // 主构造器

    // 类体

    def this(形参列表) { // 辅助构造器

    }

    def this(形参列表) { //辅助构造器可以有多个...

    }

}
```

说明：

(1) 辅助构造器，函数的名称 `this`，可以有多个，编译器通过参数的个数及类型来区分。

(2) 辅助构造方法不能直接构建对象，必须直接或者间接调用主构造方法。

(3) 构造器调用其他另外的构造器，要求被调用构造器必须提前声明。

### 2) 案例实操

(1) 如果主构造器无参数，小括号可省略，构建对象时调用的构造方法的小括号也可以省略。

```
// (1) 如果主构造器无参数，小括号可省略
//class Person () {
class Person {

    var name: String = _

    var age: Int = _

    def this(age: Int) {
```



```
    this()
    this.age = age
    println("辅助构造器")
  }

  def this(age: Int, name: String) {
    this(age)
    this.name = name
  }

  println("主构造器")
}

object Person {

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

    val person2 = new Person(18)
  }
}
```

## 6.2.6 构造器参数

### 1) 说明

Scala 类的主构造器函数的形参包括三种类型：未用任何修饰、var 修饰、val 修饰

- (1) 未用任何修饰符修饰，这个参数就是一个局部变量
- (2) var 修饰参数，作为类的成员属性使用，可以修改
- (3) val 修饰参数，作为类只读属性使用，不能修改

### 2) 案例实操

```
class Person(name: String, var age: Int, val sex: String) {
}

object Test {

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

    var person = new Person("bobo", 18, "男")

    // (1) 未用任何修饰符修饰，这个参数就是一个局部变量
    // printf(person.name)

    // (2) var 修饰参数，作为类的成员属性使用，可以修改
    person.age = 19
    println(person.age)
  }
}
```

```
// (3) val 修饰参数，作为类的只读属性使用，不能修改
// person.sex = "女"
println(person.sex)
}
}
```

## 6.4 继承和多态

### 1) 基本语法

**class 子类名 extends 父类名 { 类体 }**

(1) 子类继承父类的**属性和方法**

(2) scala 是单继承

### 2) 案例实操

(1) 子类继承父类的**属性和方法**

(2) 继承的调用顺序：父类构造器->子类构造器

```
class Person(nameParam: String) {
    var name = nameParam
    var age: Int = _

    def this(nameParam: String, ageParam: Int) {
        this(nameParam)
        this.age = ageParam
        println("父类辅助构造器")
    }

    println("父类主构造器")
}

class Emp(nameParam: String, ageParam: Int) extends Person(nameParam, ageParam) {
    var empNo: Int = _

    def this(nameParam: String, ageParam: Int, empNoParam: Int) {
        this(nameParam, ageParam)
        this.empNo = empNoParam
        println("子类的辅助构造器")
    }
}
```

```
println("子类主构造器")
}

object Test {
  def main(args: Array[String]): Unit = {
    new Emp("z3", 11, 1001)
  }
}
```

### 3) 动态绑定

Scala 中属性和方法都是动态绑定，而 Java 中只有方法为动态绑定。

### 案例实操(对比 Java 与 Scala 的重写)

#### Scala

```
class Person {
  val name: String = "person"

  def hello(): Unit = {
    println("hello person")
  }
}

class Teacher extends Person {

  override val name: String = "teacher"

  override def hello(): Unit = {
    println("hello teacher")
  }
}

object Test {
  def main(args: Array[String]): Unit = {
    val teacher: Teacher = new Teacher()
    println(teacher.name)
    teacher.hello()

    val teacher1: Person = new Teacher
    println(teacher1.name)
    teacher1.hello()
  }
}
```

#### Java

```
class Person {

  public String name = "person";
  public void hello() {
    System.out.println("hello person");
  }
}

class Teacher extends Person {
```

```

public String name = "teacher";

@Override
public void hello() {
    System.out.println("hello teacher");
}

}

public class TestDynamic {
    public static void main(String[] args) {

        Teacher teacher = new Teacher();
        Person teacher1 = new Teacher();

        System.out.println(teacher.name);
        teacher.hello();

        System.out.println(teacher1.name);
        teacher1.hello();
    }
}

```

结果对比

Scala

```

teacher
hello teacher
teacher
hello teacher

```

Java

```

teacher
hello teacher
person
hello teacher

```

## 6.5 抽象类

### 6.5.1 抽象属性和抽象方法

#### 1) 基本语法

- (1) 定义抽象类：abstract class Person{} //通过 abstract 关键字标记抽象类
- (2) 定义抽象属性：val|var name:String //一个属性没有初始化，就是抽象属性
- (3) 定义抽象方法：def hello():String //只声明而没有实现的方法，就是抽象方法

#### 案例实操

```

abstract class Person {

    val name: String

    def hello(): Unit

}

```

```
class Teacher extends Person {  
    val name: String = "teacher"  
    def hello(): Unit = {  
        println("hello teacher")  
    }  
}
```

## 2) 继承&重写

(1) 如果父类为抽象类，那么子类需要将抽象的属性和方法实现，否则子类也需声明为抽象类

(2) 重写非抽象方法需要用 `override` 修饰，重写抽象方法则可以不加 `override`。

(3) 子类中调用父类的方法使用 `super` 关键字

(4) 子类对抽象属性进行实现，父类抽象属性可以用 `var` 修饰；

子类对非抽象属性重写，父类非抽象属性只支持 `val` 类型，而不支持 `var`。

因为 `var` 修饰的为可变变量，子类继承之后就可以直接使用，没有必要重写

## 6.5.2 匿名子类

### 1) 说明

和 Java 一样，可以通过包含带有定义或重写的代码块的方式创建一个匿名的子类。

### 2) 案例实操

```
abstract class Person {  
    val name: String  
    def hello(): Unit  
}  
  
object Test {  
    def main(args: Array[String]): Unit = {  
        val person = new Person {  
            override val name: String = "teacher"  
            override def hello(): Unit = println("hello teacher")  
        }  
    }  
}
```

}

## 6.6 单例对象（伴生对象）

Scala语言是**完全面向对象**的语言，所以并没有静态的操作（即在Scala中没有静态的概念）。但是为了能够和Java语言交互（因为Java中有静态概念），就产生了一种特殊的对象来**模拟类对象**，该对象为**单例对象**。若单例对象名与类名一致，则称该单例对象这个类的**伴生对象**，这个类的所有“静态”内容都可以**放置在它的伴生对象**中声明。

### 6.6.1 单例对象语法

#### 1) 基本语法

```
object Person{
    val country:String="China"
}
```

#### 2) 说明

- （1）单例对象采用 **object** 关键字声明
- （2）单例对象对应的类称之为**伴生类**，伴生对象的名称应该和伴生类名一致。
- （3）单例对象中的属性和方法都可以通过伴生对象名（类名）直接调用访问。

#### 3) 案例实操

```
//（1）伴生对象采用 object 关键字声明
object Person {
    var country: String = "China"
}

//（2）伴生对象对应的类称之为伴生类，伴生对象的名称应该和伴生类名一致。
class Person {
    var name: String = "bobo"
}

object Test {
    def main(args: Array[String]): Unit = {
        //（3）伴生对象中的属性和方法都可以通过伴生对象名（类名）直接调用访问。
        println(Person.country)
    }
}
```

## 6.6.2 apply 方法

### 1) 说明

- (1) 通过伴生对象的 **apply 方法**，实现不使用 `new` 方法创建对象。
- (2) 如果想让主构造器变成私有的，可以在 `()` 之前加上 `private`。
- (3) `apply` 方法可以重载。
- (4) Scala 中 **`obj(arg)`** 的语句实际是在调用该对象的 **apply** 方法，即 `obj.apply(arg)`。用以统一面向对象编程和函数式编程的风格。
- (5) 当使用 `new` 关键字构建对象时，调用的其实是类的构造方法，当直接使用类名构建对象时，调用的其实是伴生对象的 `apply` 方法。

### 2) 案例实操

```
object Test {  
    def main(args: Array[String]): Unit = {  
  
        // (1) 通过伴生对象的 apply 方法，实现不使用 new 关键字创建对象。  
        val p1 = Person()  
        println("p1.name=" + p1.name)  
  
        val p2 = Person("bobo")  
        println("p2.name=" + p2.name)  
    }  
}  
  
// (2) 如果想让主构造器变成私有的，可以在 () 之前加上 private  
class Person private(cName: String) {  
    var name: String = cName  
}  
  
object Person {  
    def apply(): Person = {  
        println("apply 空参被调用")  
        new Person("xx")  
    }  
  
    def apply(name: String): Person = {  
        println("apply 有参被调用")  
        new Person(name)  
    }  
}
```

```
//注意：也可以创建其它类型对象，并不一定是伴生类对象
```

```
}
```

**扩展：在 Scala 中实现单例模式**

## 6.7 特质 (Trait)

Scala 语言中，采用特质 trait (特征) 来代替接口的概念，也就是说，多个类具有相同的特质 (特征) 时，就可以将这个特质 (特征) 独立出来，采用关键字 trait 声明。

Scala 中的 trait 中即可以有抽象属性和方法，也可以有具体的属性和方法，一个类可以混入 (mixin) 多个特质。这种感觉类似于 Java 中的抽象类。

Scala 引入 trait 特征，第一可以替代 Java 的接口，第二个也是对单继承机制的一种补充。

### 6.7.1 特质声明

#### 1) 基本语法

```
trait 特质名 {  
    trait 主体  
}
```

#### 2) 案例实操

```
trait PersonTrait {  
  
    // 声明属性  
    var name:String = _  
  
    // 声明方法  
    def eat():Unit={  
  
    }  
  
    // 抽象属性  
    var age:Int  
  
    // 抽象方法  
    def say():Unit  
}
```

通过查看字节码，可以看到特质=抽象类+接口



## 6.7.2 特质基本语法

一个类具有某种特质（特征），就意味着这个类满足了这个特质（特征）的所有要素，所以在使用时，也采用了 **extends 关键字**，如果有多个特质或存在父类，那么需要采用 **with 关键字** 连接。

### 1) 基本语法：

**没有父类**：class 类名 extends 特质 1 with 特质 2 with 特质 3 ...

**有父类**：class 类名 extends 父类 with 特质 1 with 特质 2 with 特质 3...

### 2) 说明

- (1) 类和特质的关系：使用继承的关系。
- (2) 当一个类去继承特质时，第一个连接词是 extends，后面是 with。
- (3) 如果一个类在同时继承特质和父类时，应当把父类写在 extends 后。

### 3) 案例实操

- (1) 特质可以同时拥有抽象方法和具体方法
- (2) 一个类可以混入 (mixin) 多个特质
- (3) 所有的 Java 接口都可以当做 Scala 特质使用
- (4) **动态混入**：可灵活的扩展类的功能

**(4.1) 动态混入：创建对象时混入 trait，而无需使类混入该 trait**

(4.2) 如果混入的 trait 中有未实现的方法，则需要实现

```
trait PersonTrait {  
  
    // (1) 特质可以同时拥有抽象方法和具体方法  
    // 声明属性  
    var name: String = _  
  
    // 抽象属性  
    var age: Int
```

```
// 声明方法
def eat(): Unit = {
    println("eat")
}

// 抽象方法
def say(): Unit
}

trait SexTrait {
    var sex: String
}

// (2) 一个类可以实现/继承多个特质
// (3) 所有的 Java 接口都可以当做 Scala 特质使用
class Teacher extends PersonTrait with java.io.Serializable {

    override def say(): Unit = {
        println("say")
    }

    override var age: Int = _
}

object TestTrait {

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

        val teacher = new Teacher

        teacher.say()
        teacher.eat()

        // (4) 动态混入：可灵活的扩展类的功能
        val t2 = new Teacher with SexTrait {
            override var sex: String = "男"
        }

        //调用混入 trait 的属性
        println(t2.sex)
    }
}
```

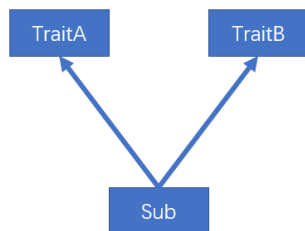
### 6.7.3 特质叠加

由于一个类可以混入 (mixin) 多个 trait，且 trait 中可以有具体的属性和方法，若混入的特质中具有相同的方法(方法名，参数列表，返回值均相同)，必然会出现继承冲突问题。

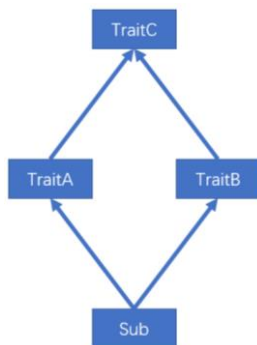
冲突分为以下两种：

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

第一种，一个类（Sub）混入的两个 trait（TraitA，TraitB）中具有相同的具体方法，且两个 trait 之间没有任何关系，解决这类冲突问题，直接在类（Sub）中重写冲突方法。



第二种，一个类（Sub）混入的两个 trait（TraitA，TraitB）中具有相同的具体方法，且两个 trait 继承自相同的 trait（TraitC），及所谓的“钻石问题”，解决这类冲突问题，Scala 采用了**特质叠加**的策略。



所谓的**特质叠加**，就是将混入的多个 trait 中的冲突方法叠加起来，案例如下，

```
trait Ball {  
  def describe(): String = {  
    "ball"  
  }  
}  
  
trait Color extends Ball {  
  override def describe(): String = {  
    "blue-" + super.describe()  
  }  
}  
  
trait Category extends Ball {  
  override def describe(): String = {  
    "foot-" + super.describe()  
  }  
}  
  
class MyBall extends Category with Color {  
  override def describe(): String = {  
    "my ball is a " + super.describe()  
  }  
}
```

```

}

object TestTrait {
  def main(args: Array[String]): Unit = {
    println(new MyBall().describe())
  }
}

```

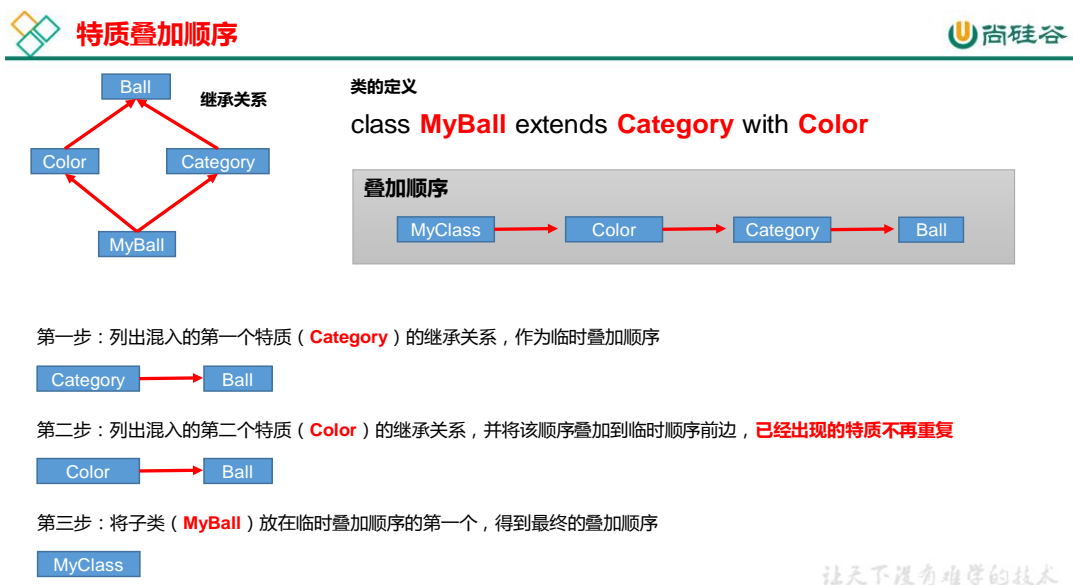
结果如下：

```
my ball is a blue-foot-ball
```

## 6.7.4 特质叠加执行顺序

**思考：**上述案例中的 `super.describe()` 调用的是父 trait 中的方法吗？

当一个类混入多个特质的时候，scala 会对所有的特质及其父特质按照一定的顺序进行排序，而此案例中的 `super.describe()` 调用的实际上是排好序后的下一个特质中的 `describe()` 方法。，排序规则如下：



**结论：**

（1）案例中的 `super` 不是表示其父特质对象，而是表示上述叠加顺序中的下一个特质，即，**MyClass 中的 `super` 指代 Color，Color 中的 `super` 指代 Category，Category 中的 `super` 指代 Ball。**

（2）如果想要调用某个指定的混入特质中的方法，可以增加约束：`super[]`，例如

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

super[Category].describe()。

## 6.7.5 特质自身类型

### 1) 说明

自身类型可实现依赖注入的功能。

### 2) 案例实操

```
class User(val name: String, val age: Int)

trait Dao {
  def insert(user: User) = {
    println("insert into database :" + user.name)
  }
}

trait APP {
  _: Dao =>

  def login(user: User): Unit = {
    println("login :" + user.name)
    insert(user)
  }
}

object MyApp extends APP with Dao {
  def main(args: Array[String]): Unit = {
    login(new User("bobo", 11))
  }
}
```

## 6.7.6 特质和抽象类的区别

1. 优先使用特质。一个类扩展多个特质是很方便的，但却只能扩展一个抽象类。
2. 如果你需要构造函数参数，使用抽象类。因为抽象类可以定义带参数的构造函数，而特质不行（有无参构造）。

## 6.8 扩展

### 6.8.1 类型检查和转换

#### 1) 说明

(1) obj.isInstanceOf[T]：判断 obj 是不是 T 类型。

(2) obj.asInstanceOf[T]：将 obj 强转成 T 类型。

(3) classOf 获取对象的类名。

## 2) 案例实操

```
class Person{  
}  
  
object Person {  
  def main(args: Array[String]): Unit = {  
  
    val person = new Person  
  
    // (1) 判断对象是否为某个类型的实例  
    val bool: Boolean = person.isInstanceOf[Person]  
  
    if ( bool ) {  
      // (2) 将对象转换为某个类型的实例  
      val p1: Person = person.asInstanceOf[Person]  
      println(p1)  
    }  
  
    // (3) 获取类的信息  
    val pClass: Class[Person] = classOf[Person]  
    println(pClass)  
  }  
}
```

## 6.8.2 枚举类和应用类

### 1) 说明

枚举类：需要继承 Enumeration

应用类：需要继承 App

### 2) 案例实操

```
object Test {  
  def main(args: Array[String]): Unit = {  
  
    println(Color.RED)  
  }  
}  
  
// 枚举类  
object Color extends Enumeration {  
  val RED = Value(1, "red")  
  val YELLOW = Value(2, "yellow")  
  val BLUE = Value(3, "blue")  
}
```

```
// 应用类
object Test20 extends App {
    println("xxxxxxxxxxxx");
}
```

### 6.8.3 Type 定义新类型

#### 1) 说明

使用 type 关键字可以定义新的数据类型名称，本质上就是类型的一个别名

#### 2) 案例实操

```
object Test {

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

        type S=String
        var v:S="abc"
        def test():S="xyz"

    }

}
```

## 第 7 章 集合

### •7.1 集合简介

1) Scala 的集合有三大类：序列 Seq、集 Set、映射 Map，所有的集合都扩展自 Iterable 特质。

2) 对于几乎所有的集合类，Scala 都同时提供了**可变**和**不可变**的版本，分别位于以下两个包

不可变集合：scala.collection.**immutable**

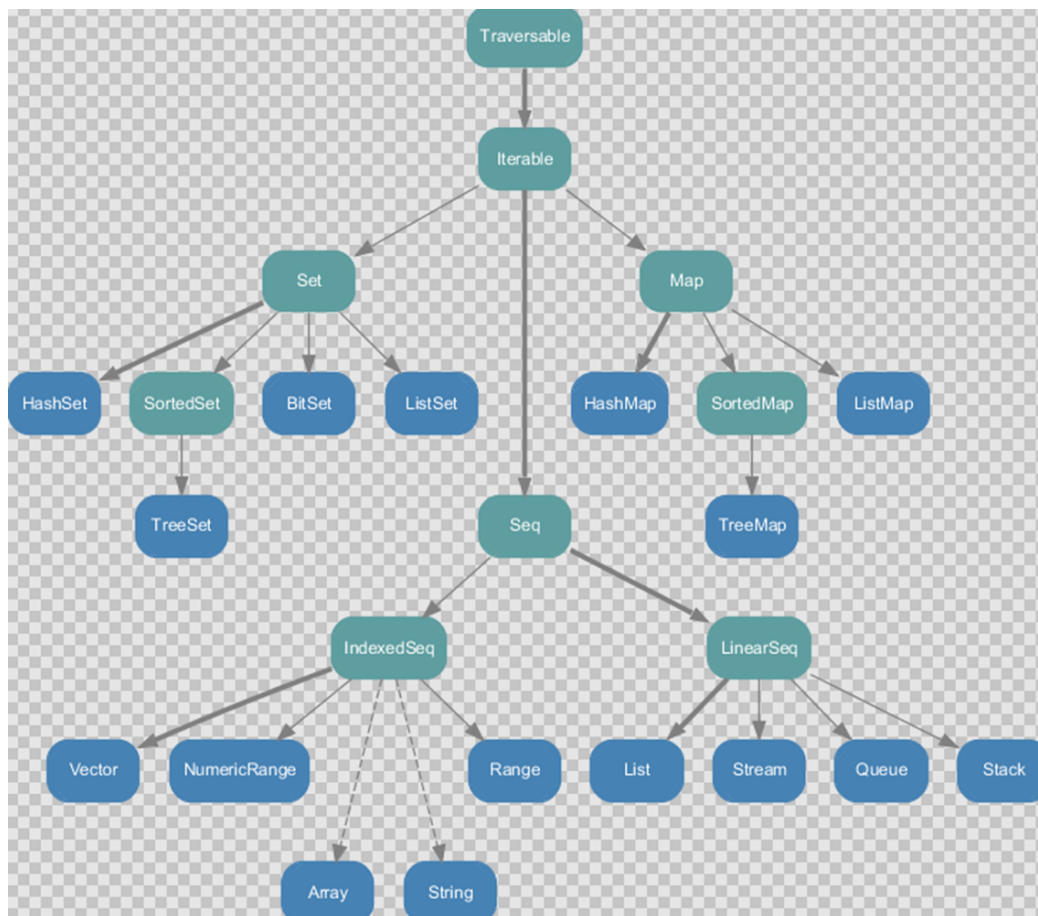
可变集合： scala.collection.**mutable**

3) Scala 不可变集合，就是指该集合对象不可修改，每次修改就会返回一个新对象，而不会对原对象进行修改。类似于 java 中的 String 对象

4) 可变集合，就是这个集合可以直接对原对象进行修改，而不会返回新的对象。类似于 java 中 StringBuilder 对象

建议：在操作集合的时候，不可变用符号，可变用方法

### 7.1.1 不可变集合继承图



- 1) Set、Map 是 Java 中也有的集合
- 2) Seq 是 Java 没有的，我们发现 List 归属到 Seq 了，因此这里的 List 就和 Java 不是同一个概念了
- 3) 我们前面的 for 循环有一个 1 to 3，就是 IndexedSeq 下的 Range
- 4) String 也是属于 IndexedSeq
- 5) 我们发现经典的数据结构比如 Queue 和 Stack 被归属到 LinearSeq(线性序列)
- 6) 大家注意 Scala 中的 Map 体系有一个 SortedMap，说明 Scala 的 Map 可以支持排序
- 7) IndexedSeq 和 LinearSeq 的区别：

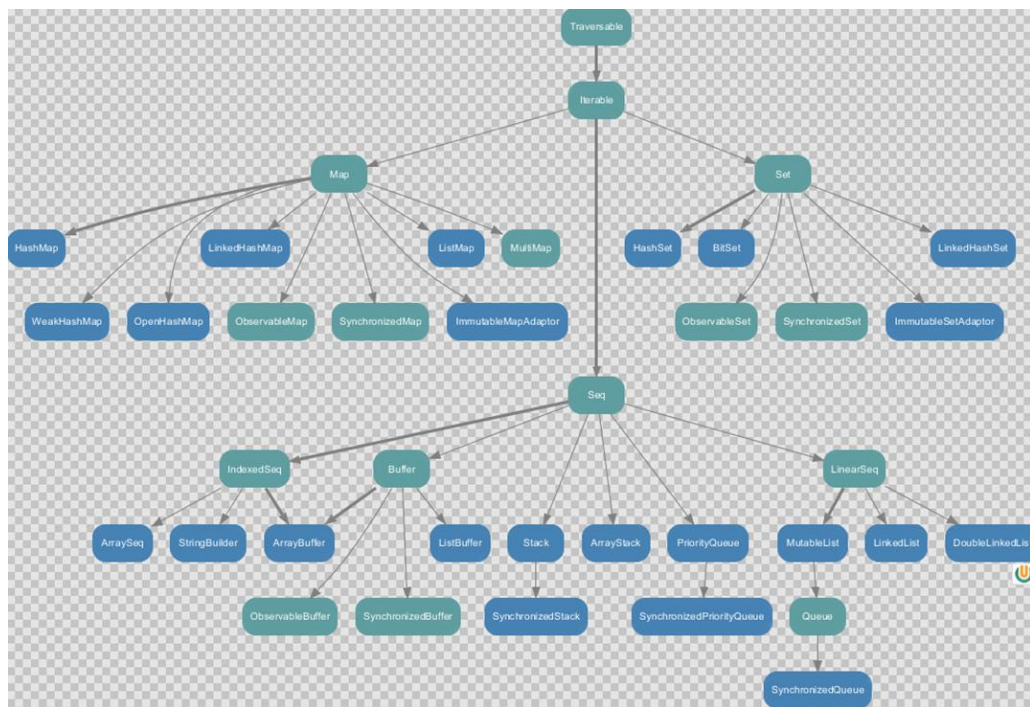
(1) **IndexedSeq** 是通过索引来查找和定位，因此速度快，比如 String 就是一个索引集



合，通过索引即可定位

(2) **LinearSeq** 是线型的，即有头尾的概念，这种数据结构一般是通过遍历来查找

## 7.1.2 可变集合继承图



## 7.2 数组

### 7.2.1 不可变数组

#### 1) 第一种方式定义数组

定义：`val arr1 = new Array[Int](10)`

(1) `new` 是关键字

(2) `[Int]`是指定可以存放的数据类型，如果希望存放任意数据类型，则指定 `Any`

(3) `(10)`，表示数组的大小，确定后就不可以变化

#### 2) 案例实操

```
object TestArray{  
    def main(args: Array[String]): Unit = {  
  
        // (1) 数组定义  
        val arr01 = new Array[Int](4)  
    }  
}
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
println(arr01.length) // 4

// (2) 数组赋值
// (2.1) 修改某个元素的值
arr01(3) = 10
// (2.2) 采用方法的形式给数组赋值
arr01.update(0,1)

// (3) 遍历数组
// (3.1) 查看数组
println(arr01.mkString(","))

// (3.2) 普通遍历
for (i <- arr01) {
    println(i)
}

// (3.3) 简化遍历
def printx(elem:Int): Unit = {
    println(elem)
}
arr01.foreach(printx)
// arr01.foreach((x)=>{println(x)})
// arr01.foreach(println(_))
arr01.foreach(println)

// (4) 增加元素 (由于创建的是不可变数组, 增加元素, 其实是产生新的数组)
println(arr01)
val ints: Array[Int] = arr01 :+ 5
println(ints)
}
```

### 3) 第二种方式定义数组

```
val arr1 = Array(1, 2)
```

(1) 在定义数组时, 直接赋初始值

(2) 使用 apply 方法创建数组对象

### 4) 案例实操

```
object TestArray{

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

        var arr02 = Array(1, 3, "bobo")
        println(arr02.length)
        for (i <- arr02) {
```

```
        println(i)
    }
}
}
```

## 7.2.2 可变数组

### 1) 定义变长数组

```
val arr01 = ArrayBuffer[Any](3, 2, 5)
```

(1) [Any]存放任意数据类型

(2) (3, 2, 5)初始化好的三个元素

(3) ArrayBuffer 需要引入 scala.collection.mutable.ArrayBuffer

### 2) 案例实操

(1) ArrayBuffer 是有序的集合

(2) 增加元素使用的是 append 方法(), 支持可变参数

```
import scala.collection.mutable.ArrayBuffer

object TestArrayBuffer {

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

        // (1) 创建并初始赋值可变数组
        val arr01 = ArrayBuffer[Any](1, 2, 3)

        // (2) 遍历数组
        for (i <- arr01) {
            println(i)
        }
        println(arr01.length) // 3
        println("arr01.hash=" + arr01.hashCode())

        // (3) 增加元素
        // (3.1) 追加数据
        arr01.+=(4)
        // (3.2) 向数组最后追加数据
        arr01.append(5, 6)
        // (3.3) 向指定的位置插入数据
        arr01.insert(0, 7, 8)
        println("arr01.hash=" + arr01.hashCode())

        // (4) 修改元素
```

```
arr01(1) = 9 //修改第 2 个元素的值
println("-----")

for (i <- arr01) {
    println(i)
}
println(arr01.length) // 5
}
```

## 7.2.3 不可变数组与可变数组的转换

### 1) 说明

arr1.toBuffer //不可变数组转可变数组

arr2.toArray //可变数组转不可变数组

(1) arr2.toArray 返回结果才是一个不可变数组, arr2 本身没有变化

(2) arr1.toBuffer 返回结果才是一个可变数组, arr1 本身没有变化

### 2) 案例实操

```
object TestArrayBuffer {

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

        // (1) 创建一个空的可变数组
        val arr2 = ArrayBuffer[Int]()

        // (2) 追加值
        arr2.append(1, 2, 3)
        println(arr2) // 1,2,3

        // (3) ArrayBuffer ==> Array
        // (3.1) arr2.toArray 返回的结果是一个新的定长数组集合
        // (3.2) arr2 它没有变化
        val newArr = arr2.toArray
        println(newArr)

        // (4) Array ==> ArrayBuffer
        // (4.1) newArr.toBuffer 返回一个变长数组 newArr2
        // (4.2) newArr 没有任何变化, 依然是定长数组
        val newArr2 = newArr.toBuffer
        newArr2.append(123)

        println(newArr2)
    }
}
```

```
}
```

## 7.2.4 多维数组

### 1) 多维数组定义

```
val arr = Array.ofDim[Double](3,4)
```

说明：二维数组中有三个一维数组，每个一维数组中有四个元素

### 2) 案例实操

```
object DimArray {  
    def main(args: Array[String]): Unit = {  
  
        //(1) 创建了一个二维数组，有三个元素，每个元素是，含有 4 个元素一维  
        数组()  
        val arr = Array.ofDim[Int](3, 4)  
        arr(1)(2) = 88  
  
        //(2) 遍历二维数组  
        for (i <- arr) { //i 就是一维数组  
  
            for (j <- i) {  
                print(j + " ")  
            }  
  
            println()  
        }  
    }  
}
```

## 7.3 列表 List

### 7.3.1 不可变 List

#### 1) 说明

(1) List 默认为不可变集合

(2) 创建一个 List (数据有顺序，可重复)

(3) 遍历 List

(4) List 增加数据

(5) 集合间合并：将一个整体拆成一个一个的个体，称为扁平化

(6) 取指定数据

(7) 空集合 Nil

## 2) 案例实操

```
object TestList {  
  
  def main(args: Array[String]): Unit = {  
  
    // (1) List 默认为不可变集合  
    // (2) 创建一个 List (数据有顺序, 可重复)  
    val list: List[Int] = List(1, 2, 3, 4, 3)  
  
    // (7) 空集合 Nil  
    val list5 = 1::2::3::4::Nil  
  
    // (4) List 增加数据  
    // (4.1) :: 的运算规则从右向左  
    // val list1 = 5::list  
    val list1 = 7::6::5::list  
    // (4.2) 添加到第一个元素位置  
    val list2 = list.+:(5)  
  
    // (5) 集合间合并: 将一个整体拆成一个一个的个体, 称为扁平化  
    val list3 = List(8, 9)  
    // val list4 = list3::list1  
    val list4 = list3::list1  
  
    // (6) 取指定数据  
    println(list(0))  
  
    // (3) 遍历 List  
    // list.foreach(println)  
    // list1.foreach(println)  
    // list3.foreach(println)  
    // list4.foreach(println)  
    list5.foreach(println)  
  }  
}
```

## 7.3.2 可变 ListBuffer

### 1) 说明

(1) 创建一个可变集合 ListBuffer

(2) 向集合中添加数据

### (3) 打印集合数据

#### 2) 案例实操

```
import scala.collection.mutable.ListBuffer

object TestList {

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

        // (1) 创建一个可变集合
        val buffer = ListBuffer(1, 2, 3, 4)

        // (2) 向集合中添加数据
        buffer.+=(5)
        buffer.append(6)
        buffer.insert(1, 2)

        // (3) 打印集合数据
        buffer.foreach(println)

        // (4) 修改数据
        buffer(1) = 6
        buffer.update(1, 7)

        // (5) 删除数据
        buffer.-(5)
        buffer.-=(5)
        buffer.remove(5)
    }
}
```

## 7.4 Set 集合

默认情况下，Scala 使用的是不可变集合，如果你想使用可变集合，需要引用

scala.collection.mutable.Set 包

### 7.4.1 不可变 Set

#### 1) 说明

(1) Set 默认是不可变集合，数据无序

(2) 数据不可重复

(3) 遍历集合

#### 2) 案例实操

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
object TestSet {  
    def main(args: Array[String]): Unit = {  
  
        // (1) Set 默认是不可变集合，数据无序  
        val set = Set(1,2,3,4,5,6)  
  
        // (2) 数据不可重复  
        val set1 = Set(1,2,3,4,5,6,3)  
  
        // (3) 遍历集合  
        for(x<-set1){  
            println(x)  
        }  
    }  
}
```

## 7.4.2 可变 mutable.Set

### 1) 说明

- (1) 创建可变集合 mutable.Set
- (2) 打印集合
- (3) 集合添加元素
- (4) 向集合中添加元素，返回一个新的 Set
- (5) 删除数据

### 2) 案例实操

```
object TestSet {  
    def main(args: Array[String]): Unit = {  
  
        // (1) 创建可变集合  
        val set = mutable.Set(1,2,3,4,5,6)  
  
        // (3) 集合添加元素  
        set += 8  
  
        // (4) 向集合中添加元素，返回一个新的 Set  
        val ints = set.+(9)  
        println(ints)  
        println("set2=" + set)  
  
        // (5) 删除数据  
        set -= (5)
```



```
// (2) 打印集合
set.foreach(println)
println(set.mkString(", "))
}
}
```

## 7.5 Map 集合

Scala 中的 Map 和 Java 类似，**也是一个散列表**，它存储的内容也是键值对 (key-value)

映射

### 7.5.1 不可变 Map

1) 说明

- (1) 创建不可变集合 Map
- (2) 循环打印
- (3) 访问数据
- (4) 如果 key 不存在，返回 0

2) 案例实操

```
object TestMap {

  def main(args: Array[String]): Unit = {
    // Map
    // (1) 创建不可变集合 Map
    val map = Map( "a"->1, "b"->2, "c"->3 )

    // (3) 访问数据
    for (elem <- map.keys) {
      // 使用 get 访问 map 集合的数据，会返回特殊类型 Option(选项)：
      // 有值 (Some)，无值(None)
      println(elem + "=" + map.get(elem).get)
    }

    // (4) 如果 key 不存在，返回 0
    println(map.get("d").getOrElse(0))
    println(map.getOrElse("d", 0))

    // (2) 循环打印
    map.foreach((kv)=>{println(kv)})
  }
}
```

```
scala> map.get("a")
res12: Option[Int] = Some(1)
```

```
scala> map.get("a").get
res13: Int = 1
```

```
scala> map.get("a").getOrElse(11)
res14: Int = 1 // 用了getOrElse不管存不存在返回值是Int
```

```
scala> map.get("d").getOrElse(11)
res15: Int = 11
```

## 7.5.2 可变 Map

### 1) 说明

- (1) 创建可变集合
- (2) 打印集合
- (3) 向集合增加数据
- (4) 删除数据
- (5) 修改数据

### 2) 案例实操

```
object TestSet {  
    def main(args: Array[String]): Unit = {  
  
        // (1) 创建可变集合  
        val map = mutable.Map( "a"->1, "b"->2, "c"->3 )  
  
        // (3) 向集合增加数据  
        map.+=( "d"->4)  
  
        // 将数值 4 添加到集合，并把集合中原值 1 返回  
        val maybeInt: Option[Int] = map.put("a", 4)  
        println(maybeInt.getOrElse(0))  
  
        // (4) 删除数据  
        map.-= ("b", "c")  
  
        // (5) 修改数据  
        map.update("d", 5)  
        map("d") = 5  
  
        // (2) 打印集合  
        map.foreach((kv)=>{println(kv)})  
    }  
}
```

## 7.6 元组

### 1) 说明

元组也是可以理解为一个容器，可以存放各种相同或不同类型的数据。说的简单点，就

是将多个无关的数据封装为一个整体，称为**元组**。

注意：元组中最大只能有 22 个元素。

## 2) 案例实操

(1) 声明元组的方式：(元素 1, 元素 2, 元素 3)

(2) 访问元组

(3) Map 中的键值对其实就元组,只不过元组的元素个数为 2，称之为对偶

```
object TestTuple {  
  
    def main(args: Array[String]): Unit = {  
  
        // (1) 声明元组的方式：(元素 1, 元素 2, 元素 3)  
        val tuple: (Int, String, Boolean) = (40, "bobo", true)  
  
        // (2) 访问元组  
        // (2.1) 通过元素的顺序进行访问，调用方式：_顺序号  
        println(tuple._1)  
        println(tuple._2)  
        println(tuple._3)  
  
        // (2.2) 通过索引访问数据  
        println(tuple.productElement(0))  
  
        // (2.3) 通过迭代器访问数据  
        for (elem <- tuple.productIterator) {  
            println(elem)  
        }  
  
        // (3) Map 中的键值对其实就元组,只不过元组的元素个数为 2，称之为  
        对偶  
        val map = Map("a" -> 1, "b" -> 2, "c" -> 3)  
        val map1 = Map(("a", 1), ("b", 2), ("c", 3))  
  
        map.foreach(tuple => {println(tuple._1 + "=" + tuple._2)})  
    }  
}
```

## 7.7 集合常用函数

### 7.7.1 基本属性和常用操作

1) 说明

(1) 获取集合长度

(2) 获取集合大小

(3) 循环遍历

(4) 迭代器

(5) 生成字符串

(6) 是否包含

## 2) 案例实操

```
object TestList {  
    def main(args: Array[String]): Unit = {  
        val list: List[Int] = List(1, 2, 3, 4, 5, 6, 7)  
  
        // (1) 获取集合长度  
        println(list.length)  
  
        // (2) 获取集合大小, 等同于 length  
        println(list.size)  
  
        // (3) 循环遍历  
        list.foreach(println)  
  
        // (4) 迭代器  
        for (elem <- list.iterator) {  
            println(elem)  
        }  
  
        // (5) 生成字符串  
        println(list.mkString(", "))  
  
        // (6) 是否包含  
        println(list.contains(3))  
    }  
}
```

## 7.7.2 衍生集合

### 1) 说明

(1) 获取集合的头

(2) 获取集合的尾 (不是头的就是尾)

(3) 集合最后一个数据

(4) 集合初始数据 (不包含最后一个)

(5) 反转

(6) 取前 (后) n 个元素

(7) 去掉前 (后) n 个元素

(8) 并集

(9) 交集

(10) 差集

(11) 拉链

(12) 滑窗

## 2) 案例实操

```
object TestList {  
  
  def main(args: Array[String]): Unit = {  
  
    val list1: List[Int] = List(1, 2, 3, 4, 5, 6, 7)  
    val list2: List[Int] = List(4, 5, 6, 7, 8, 9, 10)  
  
    // (1) 获取集合的头  
    println(list1.head)  
  
    // (2) 获取集合的尾 (不是头的就是尾)  
    println(list1.tail)  
  
    // (3) 集合最后一个数据  
    println(list1.last)  
  
    // (4) 集合初始数据 (不包含最后一个)  
    println(list1.init)  
  
    // (5) 反转  
    println(list1.reverse)  
  
    // (6) 取前 (后) n 个元素
```

```
println(list1.take(3))
println(list1.takeRight(3))

// (7) 去掉前 (后) n 个元素
println(list1.drop(3))
println(list1.dropRight(3))

// (8) 并集
println(list1.union(list2))

// (9) 交集
println(list1.intersect(list2))

// (10) 差集
println(list1.diff(list2))

// (11) 拉链 注:如果两个集合的元素个数不相等,那么会将同等数量的数据进行拉链,多余的数据省略不用
println(list1.zip(list2))

// (12) 滑窗
list1.sliding(2, 5).foreach(println)
}
```

### 7.7.3 集合计算简单函数

#### 1) 说明

- (1) 求和
- (2) 求乘积
- (3) 最大值
- (4) 最小值
- (5) 排序

#### 2) 实操

```
object TestList {

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

    val list: List[Int] = List(1, 5, -3, 4, 2, -7, 6)

    // (1) 求和
    println(list.sum)
```

```
// (2) 求乘积
println(list.product)

// (3) 最大值
println(list.max)

// (4) 最小值
println(list.min)

// (5) 排序
// (5.1) 按照元素大小排序
println(list.sortBy(x => x))

// (5.2) 按照元素的绝对值大小排序
println(list.sortBy(x => x.abs))

// (5.3) 按元素大小升序排序
println(list.sortWith((x, y) => x < y))

// (5.4) 按元素大小降序排序
println(list.sortWith((x, y) => x > y))
}
```

(1) sorted

对一个集合进行自然排序，通过传递隐式的 Ordering

(2) sortBy

对一个属性或多个属性进行排序，通过它的类型。

(3) sortWith

基于函数的排序，通过一个 comparator 函数，实现自定义排序的逻辑。

## 7.7.4 集合计算高级函数

1) 说明

(1) 过滤

遍历一个集合并从中获取满足指定条件的元素组成一个新的集合

(2) **转化/映射 (map)**

将集合中的每一个元素映射到某一个函数

(3) 扁平化

(4) 扁平化+映射 注：**flatMap** 相当于先进行 map 操作，在进行 flatten 操作

集合中的每个元素的子元素映射到某个函数并返回新集合

(5) **分组(group)**

**按照指定的规则对集合的元素进行分组**

(6) 简化 (归约)

(7) 折叠

## 2) 实操

```
object TestList {  
    def main(args: Array[String]): Unit = {  
        val list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)  
        val nestedList: List[List[Int]] = List(List(1, 2, 3), List(4,  
5, 6), List(7, 8, 9))  
        val wordList: List[String] = List("hello world", "hello  
atguigu", "hello scala")  
  
        // (1) 过滤  
        println(list.filter(x => x % 2 == 0))  
  
        // (2) 转化/映射  
        println(list.map(x => x + 1))  
  
        // (3) 扁平化  
        println(nestedList.flatten)  
  
        // (4) 扁平化+映射 注：flatMap 相当于先进行 map 操作，在进行 flatten  
操作  
        println(wordList.flatMap(x => x.split(" ")))  
  
        // (5) 分组  
        println(list.groupBy(x => x % 2))  
    }  
}
```

## 3) Reduce 方法

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网



Reduce 简化（归约）：通过指定的逻辑将集合中的数据进行聚合，从而减少数据，最终获取结果。

#### 案例实操

```
object TestReduce {  
  
  def main(args: Array[String]): Unit = {  
  
    val list = List(1,2,3,4)  
  
    // 将数据两两结合，实现运算规则  
    val i: Int = list.reduce( (x,y) => x-y )  
    println("i = " + i)  
  
    // 从源码的角度，reduce 底层调用的其实就是 reduceLeft  
    //val i1 = list.reduceLeft((x,y) => x-y)  
  
    // ((4-3)-2-1) = -2  
    val i2 = list.reduceRight((x,y) => x-y)  
    println(i2)  
  }  
}
```

#### 4) Fold 方法

Fold 折叠：化简的一种特殊情况。

##### (1) 案例实操：fold 基本使用

```
object TestFold {  
  
  def main(args: Array[String]): Unit = {  
  
    val list = List(1,2,3,4)  
  
    // fold 方法使用了函数柯里化，存在两个参数列表  
    // 第一个参数列表为：零值（初始值）  
    // 第二个参数列表为：简化规则  
  
    // fold 底层其实为 foldLeft  
    val i = list.foldLeft(1)((x,y)=>x-y)  
  
    val i1 = list.foldRight(10)((x,y)=>x-y)  
  
    println(i)  
    println(i1)  
  }  
}
```

##### (2) 案例实操：两个集合合并

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
object TestFold {

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

    // 两个 Map 的数据合并
    val map1 = mutable.Map("a"->1, "b"->2, "c"->3)
    val map2 = mutable.Map("a"->4, "b"->5, "d"->6)

    val map3: mutable.Map[String, Int] = map2.foldLeft(map1)
    {
      (map, kv) => {
        val k = kv._1
        val v = kv._2

        map(k) = map.getOrElse(k, 0) + v

        map
      }
    }

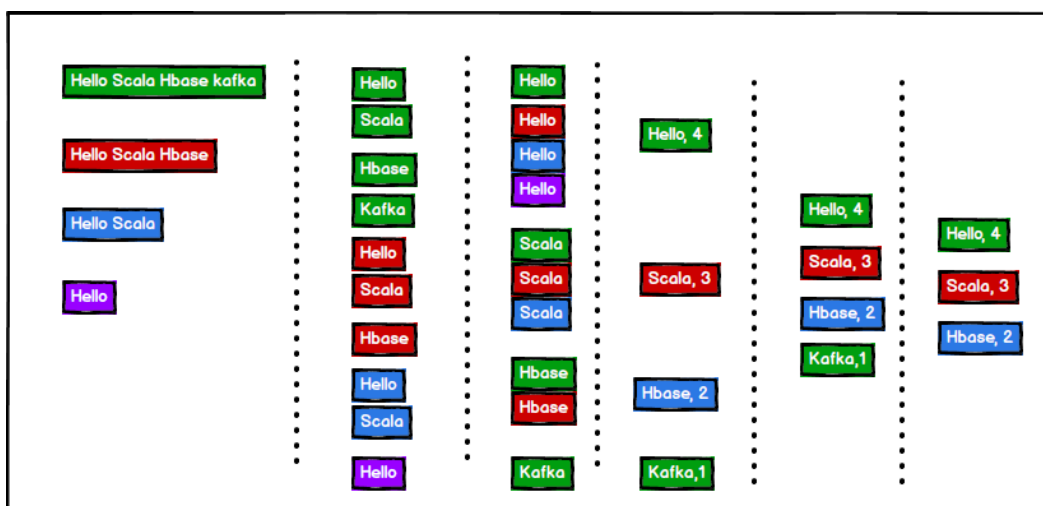
    println(map3)
  }
}
```

## 7.7.5 普通 WordCount 案例

### 1) 需求

单词计数：将集合中出现的相同的单词，进行计数，取计数排名前三的结果

### 2) 需求分析



### 3) 案例实操

```
object TestWordCount {

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

```
// 单词计数：将集合中出现的相同的单词，进行计数，取计数排名前三的结果
val stringList = List("Hello Scala Hbase kafka", "Hello
Scala Hbase", "Hello Scala", "Hello")

// 1) 将每一个字符串转换成一个一个单词
val wordList: List[String] =
stringList.flatMap(str=>str.split(" "))
//println(wordList)

// 2) 将相同的单词放在一起
val wordToWordsMap: Map[String, List[String]] =
wordList.groupBy(word=>word)
//println(wordToWordsMap)

// 3) 对相同的单词进行计数
// (word, list) => (word, count)
val wordToCountMap: Map[String, Int] =
wordToWordsMap.map(tuple=>(tuple._1, tuple._2.size))

// 4) 对计数完成后的结果进行排序（降序）
val sortList: List[(String, Int)] =
wordToCountMap.toList.sortWith {
  (left, right) => {
    left._2 > right._2
  }
}

// 5) 对排序后的结果取前 3 名
val resultList: List[(String, Int)] = sortList.take(3)

println(resultList)
}
```

## 7.7.6 复杂 WordCount 案例

### 1) 方式一

```
object TestWordCount {

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

    // 第一种方式（不通用）
    val tupleList = List(("Hello Scala Spark World ", 4), ("Hello
Scala Spark", 3), ("Hello Scala", 2), ("Hello", 1))

    val stringList: List[String] = tupleList.map(t=>(t._1 + "
") * t._2)

    //val words: List[String] =
stringList.flatMap(s=>s.split(" "))
    val words: List[String] = stringList.flatMap(_.split(" "))
  }
}
```

```
//在 map 中，如果传进来什么就返回什么，不要用_省略
val groupMap: Map[String, List[String]] =
words.groupBy(word=>word)
//val groupMap: Map[String, List[String]] =
words.groupBy(_)

// (word, list) => (word, count)
val wordToCount: Map[String, Int] = groupMap.map(t=>(t._1,
t._2.size))

val wordCountList: List[(String, Int)] =
wordToCount.toList.sortWith {
  (left, right) => {
    left._2 > right._2
  }
}.take(3)

//tupleList.map(t=>(t._1 + " ") * t._2).flatMap(_.split("
")).groupBy(word=>word).map(t=>(t._1, t._2.size))
println(wordCountList)
}
}
```

## 2) 方式二

```
object TestWordCount {

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

    val tuples = List(("Hello Scala Spark World", 4), ("Hello
Scala Spark", 3), ("Hello Scala", 2), ("Hello", 1))

    // (Hello,4), (Scala,4), (Spark,4), (World,4)
    // (Hello,3), (Scala,3), (Spark,3)
    // (Hello,2), (Scala,2)
    // (Hello,1)
    val wordToCountList: List[(String, Int)] = tuples.flatMap
{
  t => {
    val strings: Array[String] = t._1.split(" ")
    strings.map(word => (word, t._2))
  }
}

    // Hello, List((Hello,4), (Hello,3), (Hello,2), (Hello,1))
    // Scala, List((Scala,4), (Scala,3), (Scala,2))
    // Spark, List((Spark,4), (Spark,3))
    // Word, List((Word,4))
    val wordToTupleMap: Map[String, List[(String, Int)]] =
wordToCountList.groupBy(t=>t._1)

    val stringToInts: Map[String, List[Int]] =
wordToTupleMap.mapValues {
  datas => datas.map(t => t._2)
}
    stringToInts
  }
```

```
/*
    val    wordToCountMap:    Map[String,    List[Int]]    =
wordToTupleMap.map {
    t => {
        (t._1, t._2.map(t1 => t1._2))
    }
}

    val    wordToTotalCountMap:    Map[String,    Int]    =
wordToCountMap.map(t=>(t._1, t._2.sum))
println(wordToTotalCountMap)
*/
}
```

## 7.8 队列

### 1) 说明

Scala 也提供了队列 ( Queue ) 的数据结构，队列的特点就是先进先出。进队和出队的方法分别为 enqueue 和 dequeue。

### 2) 案例实操

```
object TestQueue {

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

        val que = new mutable.Queue[String]()

        que.enqueue("a", "b", "c")

        println(que.dequeue())
        println(que.dequeue())
        println(que.dequeue())
    }

}
```

## 7.9 并行集合

### 1) 说明

Scala 为了充分使用多核 CPU，提供了并行集合（有别于前面的串行集合），用于多核环境的并行计算。

### 2) 案例实操

```
object TestPar {

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

```
val result1 = (0 to 100).map{case _ => Thread.currentThread.getName}
val result2 = (0 to 100).par.map{case _ => Thread.currentThread.getName}

println(result1)
println(result2)
}
```

## 第 8 章 模式匹配

Scala 中的模式匹配类似于 Java 中的 switch 语法

```
int i = 10
switch (i) {
    case 10 :
        System.out.println("10");
        break;
    case 20 :
        System.out.println("20");
        break;
    default :
        System.out.println("other number");
        break;
}
```

但是 scala 从语法中补充了更多的功能，所以更加强大。

### 8.1 基本语法

模式匹配语法中，采用 match 关键字声明，每个分支采用 case 关键字进行声明，当需要匹配时，会从第一个 case 分支开始，如果匹配成功，那么执行对应的逻辑代码，如果匹配不成功，继续执行下一个分支进行判断。如果所有 case 都不匹配，那么会执行 case \_ 分支，类似于 Java 中 default 语句。

```
object TestMatchCase {

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

        var a: Int = 10
        var b: Int = 20
        var operator: Char = 'd'

        var result = operator match {
            case '+' => a + b
            case '-' => a - b
            case '*' => a * b
            case '/' => a / b
            case _ => "illegal"
        }
    }
}
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
    }  
    println(result)  
  }  
}
```

#### 1) 说明

(1) 如果所有 case 都不匹配，那么会执行 case \_ 分支，类似于 Java 中 default 语句，若此时没有 case \_ 分支，那么会抛出 MatchError。

(2) 每个 case 中，不需要使用 break 语句，自动中断 case。

(3) match case 语句可以匹配任何类型，而不只是字面量。

(4) => 后面的代码块，直到下一个 case 语句之前的代码是**作为一个整体执行**，可以使用{}括起来，也可以不括。

## 8.2 模式守卫

#### 1) 说明

如果想要表达匹配某个范围的数据，就需要在模式匹配中增加条件守卫。

#### 2) 案例实操

```
object TestMatchGuard {  
  def main(args: Array[String]): Unit = {  
    def abs(x: Int) = x match {  
      case i: Int if i >= 0 => i  
      case j: Int if j < 0 => -j  
      case _ => "type illegal"  
    }  
    println(abs(-5))  
  }  
}
```

## 8.3 模式匹配类型

### 8.3.1 匹配常量

#### 1) 说明

Scala 中，模式匹配可以匹配所有的字面量，包括字符串，字符，数字，布尔值等等。

## 2) 实操

```
object TestMatchVal {  
    def main(args: Array[String]): Unit = {  
        println(describe(6))  
    }  
  
    def describe(x: Any) = x match {  
        case 5 => "Int five"  
        case "hello" => "String hello"  
        case true => "Boolean true"  
        case '+' => "Char +"  
    }  
}
```

### 8.3.2 匹配类型

#### 1) 说明

需要进行类型判断时，可以使用前文所学的 `isInstanceOf[T]` 和 `asInstanceOf[T]`，也可使用模式匹配实现同样的功能。

#### 2) 案例实操

```
object TestMatchClass {  
    def describe(x: Any) = x match {  
        case i: Int => "Int"  
        case s: String => "String hello"  
        case m: List[_] => "List"  
        case c: Array[Int] => "Array[Int]"  
        case something => "something else " + something  
    }  
  
    def main(args: Array[String]): Unit = {  
  
        //泛型擦除  
        println(describe(List(1, 2, 3, 4, 5)))  
  
        //数组例外，可保留泛型  
        println(describe(Array(1, 2, 3, 4, 5, 6)))  
        println(describe(Array("abc")))  
    }  
}
```



### 8.3.3 匹配数组

#### 1) 说明

scala 模式匹配可以对集合进行精确的匹配，例如匹配只有两个元素的、且第一个元素为 0 的数组。

#### 2) 案例实操

```
object TestMatchArray {  
    def main(args: Array[String]): Unit = {  
        for (arr <- Array(Array(0), Array(1, 0), Array(0, 1, 0),  
Array(1, 1, 0), Array(1, 1, 0, 1), Array("hello", 90))) { // 对一个数组集合进行遍历  
  
            val result = arr match {  
                case Array(0) => "0" //匹配 Array(0) 这个数组  
  
                case Array(x, y) => x + "," + y //匹配有两个元素的数组，然后将元素值赋给对应的 x,y  
  
                case Array(0, _) => "以 0 开头的数组" //匹配以 0 开头和数组  
  
                case _ => "something else"  
            }  
  
            println("result = " + result)  
        }  
    }  
}
```

### 8.3.4 匹配列表

#### 1) 方式一

```
object TestMatchList {  
    def main(args: Array[String]): Unit = {  
  
        //list 是一个存放 List 集合的数组  
        //请思考，如果要匹配 List(88) 这样的只含有一个元素的列表，并原值返回，应该怎么写  
        for (list <- Array(List(0), List(1, 0), List(0, 0, 0), List(1, 0, 0), List(88))) {  
  
            val result = list match {
```

```

        case List(0) => "0" //匹配 List(0)
        case List(x, y) => x + "," + y //匹配有两个元素的 List
        case List(0, _) => "0 ..."
        case _ => "something else"
    }

    println(result)
}
}
}

```

## 2) 方式二

```

object TestMatchList {

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

        val list: List[Int] = List(1, 2, 5, 6, 7)

        list match {
            case first :: second :: rest => println(first + "-" +
second + "-" + rest)
            case _ => println("something else")
        }
    }
}

```

## 8.3.5 匹配元组

```

object TestMatchTuple {

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

        //对一个元组集合进行遍历
        for (tuple <- Array((0, 1), (1, 0), (1, 1), (1, 0, 2))) {

            val result = tuple match {
                case (0, _) => "0 ..." //是第一个元素是 0 的元组
                case (y, 0) => "" + y + "0" // 匹配后一个元素是 0 的对
偶元组

                case (a, b) => "" + a + " " + b
                case _ => "something else" //默认
            }

            println(result)
        }
    }
}

```

### 扩展案例

```
object TestGeneric {
```

```
def main(args: Array[String]): Unit = {  
    //特殊的模式匹配1 打印元组第一个元素  
    for (elem <- Array(("a", 1), ("b", 2), ("c", 3))) {  
        println(elem._1)  
    }  
    for ((word,count) <- Array(("a", 1), ("b", 2), ("c", 3))) {  
        println(word)  
    }  
    for ((word,_) <- Array(("a", 1), ("b", 2), ("c", 3))) {  
        println(word)  
    }  
    for (("a",count) <- Array(("a", 1), ("b", 2), ("c", 3))) {  
        println(count)  
    }  
    println("-----")  
    //特殊的模式匹配2 给元组元素命名  
    var (id,name,age): (Int, String, Int) = (100, "zs", 20)  
    println((id,name,age))  
    println("-----")  
    //特殊的模式匹配3 遍历集合中的元组,给 count * 2  
    var list: List[(String, Int)] = List(("a", 1), ("b", 2), ("c", 3))  
    //println(list.map(t => (t._1, t._2 * 2)))  
    println(  
        list.map(  
            case (word,count)=>(word,count*2)  
        )  
    )  
    var list1 = List(("a", ("a", 1)), ("b", ("b", 2)), ("c", ("c", 3)))  
    println(  
        list1.map(  
            case (groupkey, (word,count))=>(word,count*2)  
        )  
    )  
}
```

## 8.3.6 匹配对象及样例类

### 1) 基本语法

```
class User(val name: String, val age: Int)
```

```
object User{

    def apply(name: String, age: Int): User = new User(name, age)

    def unapply(user: User): Option[(String, Int)] = {
        if (user == null)
            None
        else
            Some(user.name, user.age)
    }
}

object TestMatchUnapply {
    def main(args: Array[String]): Unit = {
        val user: User = User("zhangsan", 11)
        val result = user match {
            case User("zhangsan", 11) => "yes"
            case _ => "no"
        }

        println(result)
    }
}
```

## 小结

- `val user = User("zhangsan",11)`，该语句在执行时，实际调用的是 `User` 伴生对象中的 `apply` 方法，因此不用 `new` 关键字就能构造出相应的对象。
- 当将 `User("zhangsan", 11)` 写在 `case` 后时[`case User("zhangsan", 11) => "yes"`]，会默认调用 `unapply` 方法(对象提取器)，**user 作为 unapply 方法的参数**，`unapply` 方法将 `user` 对象的 `name` 和 `age` 属性提取出来，与 `User("zhangsan", 11)` 中的属性值进行匹配
- `case` 中对象的 `unapply` 方法(提取器)返回 `Some`，且所有属性均一致，才算匹配成功，属性不一致，或返回 `None`，则匹配失败。
- 若只提取对象的一个属性，则提取器为 `unapply(obj:Obj):Option[T]`  
若提取对象的多个属性，则提取器为 `unapply(obj:Obj):Option[(T1,T2,T3...)]`  
若提取对象的可变个属性，则提取器为 `unapplySeq(obj:Obj):Option[Seq[T]]`

## 2) 样例类

(1) 语法：

```
case class Person (name: String, age: Int)
```

(2) 说明

① 样例类仍然是类，和普通类相比，只是其自动生成了伴生对象，并且伴生对象中自动提供了一些常用的方法，如 **apply**、**unapply**、**toString**、**equals**、**hashCode** 和 **copy**。

② 样例类是为模式匹配而优化的类，因为其默认提供了 **unapply** 方法，因此，样例类可以直接使用模式匹配，而无需自己实现 **unapply** 方法。

③ 构造器中的每一个参数都成为 **val**，除非它被显式地声明为 **var**（不建议这样做）

(3) 实操

上述匹配对象的案例使用样例类会节省大量代码

```
case class User(name: String, age: Int)

object TestMatchUnapply {
  def main(args: Array[String]): Unit = {
    val user: User = User("zhangsan", 11)
    val result = user match {
      case User("zhangsan", 11) => "yes"
      case _ => "no"
    }

    println(result)
  }
}
```

## 8.4 变量声明中的模式匹配

```
case class Person(name: String, age: Int)

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

    val (x, y) = (1, 2)
    println(s"x=$x,y=$y")

    val Array(first, second, _) = Array(1, 7, 2, 9)
    println(s"first=$first,second=$second")

    val Person(name, age) = Person1("zhangsan", 16)
    println(s"name=$name,age=$age")
  }
}
```

## 8.5 for 表达式中的模式匹配

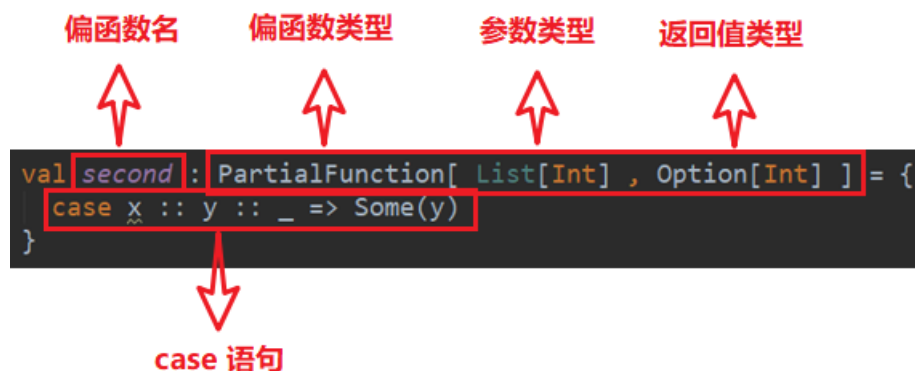
```
object TestMatchFor {  
  
  def main(args: Array[String]): Unit = {  
  
    val map = Map("A" -> 1, "B" -> 0, "C" -> 3)  
    for ((k, v) <- map) { //直接将 map 中的 k-v 遍历出来  
      println(k + " -> " + v) //3 个  
    }  
    println("-----")  
  
    //遍历 value=0 的 k-v ,如果 v 不是 0,过滤  
    for ((k, 0) <- map) {  
      println(k + " --> " + 0) // B->0  
    }  
  
    println("-----")  
    //if v == 0 是一个过滤的条件  
    for ((k, v) <- map if v >= 1) {  
      println(k + " ---> " + v) // A->1 和 c->33  
    }  
  }  
}
```

## 8.6 偏函数中的模式匹配(了解)

偏函数也是函数的一种,通过偏函数我们可以方便的对输入参数做更精确的检查。例如该偏函数的输入类型为 List[Int],而我们需要的是第一个元素是 0 的集合,这就是通过模式匹配实现的。

### 1) 偏函数定义

```
val second: PartialFunction[List[Int], Option[Int]] = {  
  case x :: y :: _ => Some(y)  
}
```



注：该偏函数的功能是返回输入的 List 集合的第二个元素

## 2) 偏函数原理

上述代码会被 scala 编译器翻译成以下代码，与普通函数相比，只是多了一个用于参数检查的函数——isDefinedAt，其返回值类型为 Boolean。

```
val second = new PartialFunction[List[Int], Option[Int]] {  
  
    //检查输入参数是否合格  
    override def isDefinedAt(list: List[Int]): Boolean = list match  
    {  
        case x :: y :: _ => true  
        case _ => false  
    }  
  
    //执行函数逻辑  
    override def apply(list: List[Int]): Option[Int] = list match  
    {  
        case x :: y :: _ => Some(y)  
    }  
}
```

## 3) 偏函数使用

偏函数不能像 second(List(1,2,3))这样直接使用，因为这样会直接调用 apply 方法，而应该调用 applyOrElse 方法，如下

```
second.applyOrElse(List(1,2,3), (_: List[Int]) => None)
```

applyOrElse 方法的逻辑为 if (isDefinedAt(list)) apply(list) else default。如果输入参数满足条件，即 isDefinedAt 返回 true，则执行 apply 方法，否则执行 default 方法，default 方法为参数不满足要求的处理逻辑。

## 4) 案例实操

### (1) 需求

将该 List(1,2,3,4,5,6,"test")中的 Int 类型的元素加一，并去掉字符串。

```
def main(args: Array[String]): Unit = {  
    val list = List(1,2,3,4,5,6,"test")  
    val list1 = list.map {  
        a =>  
            a match {
```

```

    case i: Int => i + 1
    case s: String => s + 1
  }
}
println(list1.filter(a=>a.isInstanceOf[Int]))
}

```

## (2) 实操

方法一：

```
List(1,2,3,4,5,6,"test").filter(_._isInstanceOf[Int]).map(_._asInstanceOf[Int] + 1).foreach(println)
```

方法二：

```
List(1, 2, 3, 4, 5, 6, "test").collect { case x: Int => x + 1 }.foreach(println)
```

## 第9章 异常

语法处理上和 Java 类似，但是又不尽相同。

### 9.1 Java 异常处理

```

public class ExceptionDemo {

    public static void main(String[] args) {

        try {
            int a = 10;
            int b = 0;
            int c = a / b;
        } catch (ArithmeticException e) {
            // catch 时，需要将范围小的写到前面
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("finally");
        }
    }
}

```

注意事项

(1) Java 语言按照 try—catch—finally 的方式来处理异常

(2) 不管有没有异常捕获，都会执行 finally，因此通常可以在 finally 代码块中释放资源。

源。

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网



(3) 可以有多个 catch，分别捕获对应的异常，这时需要把范围小的异常类写在前面，把范围大的异常类写在后面，否则编译错误。

## 9.2 Scala 异常处理

```
def main(args: Array[String]): Unit = {  
    try {  
        var n= 10 / 0  
    } catch {  
        case ex: ArithmeticException=>{  
            // 发生算术异常  
            println("发生算术异常")  
        }  
        case ex: Exception=>{  
            // 对异常处理  
            println("发生了异常 1")  
            println("发生了异常 2")  
        }  
    } finally {  
        println("finally")  
    }  
}
```

1) 我们将可疑代码封装在 try 块中。在 try 块之后使用了一个 catch 处理程序来捕获异常。如果发生任何异常，catch 处理程序将处理它，程序将不会异常终止。

2) Scala 的异常的工作机制和 Java 一样，但是 Scala 没有“checked (编译期)”异常，即 Scala 没有编译异常这个概念，异常都是在运行的时候捕获处理。

3) 异常捕捉的机制与其他语言中一样，如果有异常发生，catch 子句是按次序捕捉的。因此，在 catch 子句中，越具体的异常越要靠前，越普遍的异常越靠后，如果把越普遍的异常写在前，把具体的异常写在后，在 Scala 中也不会报错，但这样是非常不好的编程风格。

4) finally 子句用于执行不管是正常处理还是有异常发生时都需要执行的步骤，一般用于对象的清理工作，这点和 Java 一样。

5) 用 throw 关键字，抛出一个异常对象。所有异常都是 Throwable 的子类型。throw 表达式是有类型的，就是 Nothing，因为 Nothing 是所有类型的子类型，所以 throw 表达式可

以用在需要类型的地方

```
def test():Nothing = {
    throw new Exception("不对")
}
```

6) java 提供了 throws 关键字来声明异常。可以使用方法定义声明异常。它向调用者函数提供了此方法可能引发此异常的信息。它有助于调用函数处理并将该代码包含在 try-catch 块中，以避免程序异常终止。在 Scala 中，可以使用 throws 注解来声明异常

```
def main(args: Array[String]): Unit = {
    f11()
}

@throws(classOf[NumberFormatException])
def f11()={
    "abc".toInt
}
```

## 第 10 章 隐式转换

当编译器第一次编译失败的时候，会在当前的环境中查找能让代码编译通过的方法，用于将类型进行转换，实现二次编译

### 10.1 隐式函数

<https://www.cnblogs.com/xia520pi/p/8745923.html>

#### 1) 说明

隐式转换可以在不需改任何代码的情况下，扩展某个类的功能。

#### 2) 案例实操

需求：通过隐式转化为 Int 类型增加方法。

```
class MyRichInt(val self: Int) {
    def myMax(i: Int): Int = {
        if (self < i) i else self
    }

    def myMin(i: Int): Int = {
        if (self < i) self else i
    }
}

object TestImplicitFunction {
    // 使用 implicit 关键字声明的函数称之为隐式函数
```

```
implicit def convert(arg: Int): MyRichInt = {
    new MyRichInt(arg)
}

def main(args: Array[String]): Unit = {
    // 当想调用对象功能时，如果编译错误，那么编译器会尝试在当前作用域范
    围内查找能调用对应功能的转换规则，这个调用过程是由编译器完成的，所以称之为隐
    式转换。也称之为自动转换
    println(2.myMax(6))
}
```

## 10.2 隐式参数

普通方法或者函数中的参数可以通过 **implicit** 关键字声明为隐式参数，调用该方法时，就可以传入该参数，编译器会在相应的作用域寻找符合条件的隐式值。

### 1) 说明

- (1) 同一个作用域中，相同类型的隐式值只能有一个
- (2) 编译器按照隐式参数的类型去寻找对应类型的隐式值，与隐式值的名称无关。
- (3) 隐式参数优先于默认参数

### 2) 案例实操

```
object TestImplicitParameter {

    implicit val str: String = "hello world!"

    def hello(implicit arg: String="good bey world!"): Unit = {
        println(arg)
    }

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

## 10.3 隐式类

在 Scala2.10 后提供了隐式类，可以使用 **implicit** 声明类，隐式类的非常强大，同样可以扩展类的功能，在集合中隐式类会发挥重要的作用。

### 1) 隐式类说明

(1) 其所带的构造参数有且只能有一个

(2) 隐式类必须被定义在“类”或“伴生对象”或“包对象”里，即隐式类不能是**顶级的**。

## 2) 案例实操

```
object TestImplicitClass {  
    implicit class MyRichInt(arg: Int) {  
        def myMax(i: Int): Int = {  
            if (arg < i) i else arg  
        }  
        def myMin(i: Int) = {  
            if (arg < i) arg else i  
        }  
    }  
    def main(args: Array[String]): Unit = {  
        println(1.myMax(3))  
    }  
}
```

## 10.4 隐式解析机制

### 1) 说明

(1) 首先会在当前代码作用域下查找隐式实体（隐式方法、隐式类、隐式对象）。（**一般是这种情况**）

(2) 如果第一条规则查找隐式实体失败，会继续在隐式参数的类型的作用域里查找。

类型的作用域是指与**该类型相关联的全部伴生对象**以及**该类型所在包的包对象**。

### 2) 案例实操

```
package com.atguigu.chapter10  
  
import com.atguigu.chapter10.Scala05_Transform4.Teacher  
  
// (2) 如果第一条规则查找隐式实体失败，会继续在隐式参数的类型的作用域里查找。  
// 类型的作用域是指与该类型相关联的全部伴生模块，  
object TestTransform extends PersonTrait {  
    def main(args: Array[String]): Unit = {
```

```
// (1) 首先会在当前代码作用域下查找隐式实体
val teacher = new Teacher()
teacher.eat()
teacher.say()
}

class Teacher {
  def eat(): Unit = {
    println("eat...")
  }
}

trait PersonTrait {
}

object PersonTrait {
  // 隐式类 : 类型 1 => 类型 2
  implicit class Person5(user:Teacher) {

    def say(): Unit = {
      println("say...")
    }
  }
}
```

## 第 11 章 泛型

### 11.1 协变和逆变

#### 1) 语法

```
class MyList[+T]{ //协变
```

```
}
```

```
class MyList[-T]{ //逆变
```

```
}
```

```
class MyList[T] //不变
```

#### 2) 说明

协变：Son 是 Father 的**子类**，则 MyList[Son] 也作为 MyList[Father]的 **"子类"**。

逆变：Son 是 Father 的**子类**，则 MyList[Son]作为 MyList[Father]的 **"父类"**。

不变：Son 是 Father 的**子类**，则 MyList[Father]与 MyList[Son] **"无父子关系"**。

### 3) 实操

```
//泛型模板
//class MyList<T>{}
//不变
//class MyList[T]{}
//协变
//class MyList[+T]{}
//逆变
//class MyList[-T]{}

class Parent{}
class Child extends Parent{}
class SubChild extends Child{}

object Scala_TestGeneric {
  def main(args: Array[String]): Unit = {
    //var s:MyList[Child] = new MyList[SubChild]
  }
}
```

## 11.2 泛型上下限

### 1) 语法

```
Class PersonList[T <: Person]{ //泛型上限
}
```

```
Class PersonList[T >: Person]{ //泛型下限
}
```

### 2) 说明

泛型的上下限的作用是对传入的泛型进行限定。

### 3) 实操

```
class Parent{}
class Child extends Parent{}
class SubChild extends Child{}

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

    //test(classOf[SubChild])
    //test[Child](new SubChild)
  }
}
```

```
//泛型通配符之上限
//def test[A <: Child] (a:Class[A]): Unit = {
//  println(a)
//}

//泛型通配符之下限
//def test[A >: Child] (a:Class[A]): Unit = {
//  println(a)
//}

//泛型通配符之下限 形式扩展
def test[A >: Child] (a:A): Unit = {
  println(a.getClass.getName)
}
}
```

## 11.3 上下文限定

### 1) 语法

**def f[A : B](a: A) = println(a)** //等同于 **def f[A](a:A)(implicit arg:B[A])=println(a)**

### 2) 说明

上下文限定是将泛型和隐式转换的结合产物，以下两者功能相同，使用上下文限定[A :

Ordering]-后，方法内无法使用隐式参数名调用隐式参数，需要通过 **implicitly[Ordering[A]]**

获取隐式变量，如果此时无法查找到对应类型的隐式变量，会发生出错误。

```
implicit val x = 1
val y = implicitly[Int]
val z = implicitly[Double]
```

### 3) 实操

```
def f[A:Ordering] (a:A,b:A) =implicitly[Ordering[A]].compare(a,b)
def f[A] (a: A, b: A) (implicit ord: Ordering[A]) = ord.compare(a, b)
```

## 第 12 章 总结

### 12.1 开发环境

要求掌握必要的 Scala 开发环境搭建技能。

### 12.2 变量和数据类型

掌握 var 和 val 的区别

掌握数值类型 ( Byte、Short、Int、Long、Float、Double、Char ) 之间的转换关系

## 12.3 流程控制

掌握 if-else、for、while 等必要的流程控制结构，掌握如何实现 break、continue 的功能。

## 12.4 函数式编程

掌握高阶函数、匿名函数、函数柯里化、闭包、函数参数以及函数至简原则。

## 12.5 面向对象

掌握 Scala 与 Java 继承方面的区别、单例对象( 伴生对象 )、构造方法、特质的用法及功能。

## 12.6 集合

掌握常用集合的使用、集合常用的计算函数。

## 12.7 模式匹配

掌握模式匹配的用法

## 12.8 下划线

掌握下划线不同场合的不同用法

## 12.9 异常

掌握异常常用操作即可

## 12.10 隐式转换

掌握隐式方法、隐式参数、隐式类，以及隐式解析机制

## 12.11 泛型

掌握泛型语法

# 第 13 章 IDEA 快捷键

1) 快速生成程序入口：main

```
输入 main->回车
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网



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

2) 自动补全变量：.var

输入 1.var->回车

```
val i: Int = 2
```

3) 快速打印：.sout

输入 1.sout->回车

```
println(1)
```

4) 快速生成 for 循环：遍历对象.for

输入 1 to 3.for

```
for (elem <- 1 to 3) {  
}
```

5) 查看当前文件的结构：Ctrl + F12

6) 格式化当前代码：Ctrl + Shift + L

7) 自动为当前代码补全变量声明：Ctrl + Shift + V

更多请查看截图：

### IntelliJ Idea 常用快捷键列表

Ctrl+Shift + Enter, 语句完成

"! ", 否定完成, 输入表达式时按 "!" 键

Ctrl+E, 最近的文件

Ctrl+Shift+E, 最近更改的文件

Shift+Click, 可以关闭文件

Ctrl+[ OR ], 可以跑到大括号的开头与结尾

Ctrl+F12, 可以显示当前文件的结构

Ctrl+F7, 可以查询当前元素在当前文件中的引用, 然后按 F3 可以选择

Ctrl+N, 可以快速打开类

Ctrl+Shift+N, 可以快速打开文件

Alt+Q, 可以看到当前方法的声明

Ctrl+P, 可以显示参数信息

Ctrl+Shift+Insert, 可以选择剪贴板内容并插入

Alt+Insert, 可以生成构造器/Getter/Setter等

Ctrl+Alt+V, 可以引入变量。例如: new String(); 自动导入变量定义

Ctrl+Alt+T, 可以把代码包在一个块内, 例如: try/catch

Ctrl+Enter, 导入包, 自动修正

Ctrl+Alt+L, 格式化代码

Ctrl+Alt+I, 将选中的代码进行自动缩进编排, 这个功能在编辑 JSP 文件时也可以工作

Ctrl+Alt+O, 优化导入的类和包

Ctrl+R, 替换文本

Ctrl+F, 查找文本

Ctrl+Shift+Space, 自动补全代码

Ctrl+空格, 代码提示 (与系统输入法快捷键冲突)

Ctrl+Shift+Alt+N, 查找类中的方法或变量

Alt+Shift+C, 最近的更改

Alt+Shift+Up/Down, 上/下移一行

Shift+F6, 重构 - 重命名  
Ctrl+X, 删除行  
Ctrl+D, 复制行  
Ctrl+/或Ctrl+Shift+/, 注释 (//或者/\*\*/)  
Ctrl+J, 自动代码 (例如: serr)  
Ctrl+Alt+J, 用动态模板环绕  
Ctrl+H, 显示类结构图 (类的继承层次)  
Ctrl+Q, 显示注释文档  
Alt+F1, 查找代码所在位置  
Alt+1, 快速打开或隐藏工程面板  
Ctrl+Alt+left/right, 返回至上次浏览的位置  
Alt+left/right, 切换代码视图  
Alt+Up/Down, 在方法间快速移动定位  
Ctrl+Shift+Up/Down, 向上/下移动语句  
F2 或 Shift+F2, 高亮错误或警告快速定位  
Tab, 代码标签输入完成后, 按 Tab, 生成代码  
Ctrl+Shift+F7, 高亮显示所有该文本, 按 Esc 高亮消失  
Alt+F3, 逐个往下查找相同文本, 并高亮显示  
Ctrl+Up/Down, 光标中转到第一行或最后一行下  
Ctrl+B/Ctrl+Click, 快速打开光标处的类或方法 (跳转到定义处)  
Ctrl+Alt+B, 跳转到方法实现处  
Ctrl+Shift+Backspace, 跳转到上次编辑的地方  
Ctrl+O, 重写方法  
Ctrl+Alt+Space, 类名自动完成  
Ctrl+Alt+Up/Down, 快速跳转搜索结果  
Ctrl+Shift+J, 整合两行  
Alt+F8, 计算变量值  
Ctrl+Shift+V, 可以将最近使用的剪贴板内容选择插入到文本  
Ctrl+Alt+Shift+V, 简单粘贴  
Shift+Esc, 不仅可以把焦点移到编辑器上, 而且还可以隐藏当前 (或最后活动的) 工具窗口  
F12, 把焦点从编辑器移到最近使用的工具窗口  
Shift+F1, 要打开编辑器光标字符处使用的类或者方法 Java 文档的浏览器

Ctrl+W, 可以选择单词继而语句继而行继而函数  
Ctrl+Shift+W, 取消选择光标所在词  
Alt+F7, 查找整个工程中使用地某一个类、方法或者变量的位置

Ctrl+I, 实现方法  
Ctrl+Shift+U, 大小写转化  
Ctrl+Y, 删除当前行

Shift+Enter, 向下插入新行  
psvm/sout, main/System.out.println(); Ctrl+J, 查看更多  
Ctrl+Shift+F, 全局查找  
Ctrl+F, 查找/Shift+F3, 向上查找/F3, 向下查找  
Ctrl+Shift+S, 高级搜索  
Ctrl+U, 转到父类  
Ctrl+Alt+S, 打开设置对话框  
Alt+Shift+Insert, 开启/关闭列选择模式  
Ctrl+Alt+Shift+S, 打开当前项目/模块属性  
Ctrl+G, 定位行  
Alt+Home, 跳转到导航栏  
Ctrl+Enter, 上插一行  
Ctrl+Backspace, 按单词删除  
Ctrl+"+/-", 当前方法展开、折叠  
Ctrl+Shift+"+/-", 全部展开、折叠  
【调试部分、编译】  
Ctrl+F2, 停止  
Alt+Shift+F9, 选择 Debug  
Alt+Shift+F10, 选择 Run  
Ctrl+Shift+F9, 编译  
Ctrl+Shift+F10, 运行  
Ctrl+Shift+F8, 查看断点  
F8, 步过  
F7, 步入  
Shift+F7, 智能步入  
Shift+F8, 步出

Alt+Shift+F8, 强制步过  
Alt+Shift+F7, 强制步入  
Alt+F9, 运行至光标处  
Ctrl+Alt+F9, 强制运行至光标处  
F9, 恢复程序  
Alt+F10, 定位到断点  
Ctrl+F8, 切换行断点  
Ctrl+F9, 生成项目  
Alt+1, 项目  
Alt+2, 收藏  
Alt+6, TODO  
Alt+7, 结构  
Ctrl+Shift+C, 复制路径  
Ctrl+Alt+Shift+C, 复制引用, 必须选择类名  
Ctrl+Alt+Y, 同步  
Ctrl+~, 快速切换方案 (界面外观、代码风格、快捷键映射等菜单)  
Shift+F12, 还原默认布局  
Ctrl+Shift+F12, 隐藏/恢复所有窗口  
Ctrl+F4, 关闭  
Ctrl+Shift+F4, 关闭活动选项卡  
Ctrl+Tab, 转到下一个拆分器  
Ctrl+Shift+Tab, 转到上一个拆分器

**【重构】**  
Ctrl+Alt+Shift+T, 弹出重构菜单  
Shift+F6, 重命名  
F6, 移动  
F5, 复制  
Alt+Delete, 安全删除  
Ctrl+Alt+N, 内联

**【查找】**  
Ctrl+F, 查找  
Ctrl+R, 替换  
F3, 查找下一个  
Shift+F3, 查找上一个