

尚硅谷大数据技术之 Scala

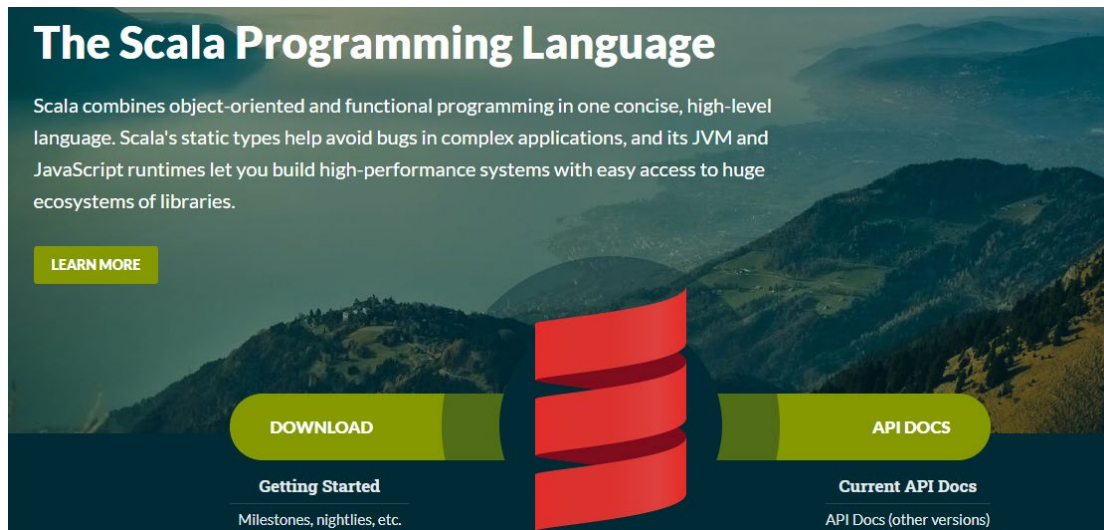
版本：V4.0



作者：尚硅谷大数据研发部

第1章 Scala 入门

1.1 概述



1.1.1 什么是 Scala

从英文的角度来讲，Scala 并不是一个单词，而是 Scalable Language 两个单词的缩写，表示可伸缩语言的意思。从计算机的角度来讲，Scala 是一门完整的软件编程语言，那么连在一起就表示 Scala 是一门可伸缩的软件编程语言。之所以说它是可伸缩，是因为这门语言体现了面向对象，函数式编程等多种不同的语言范式，且融合了不同语言新的特性。

Scala 编程语言是由联邦理工学院洛桑（EPFL）的 Martin Odersky 于 2001 年基于 Funnel 的工作开始设计并开发的。由于 Martin Odersky 之前的工作是开发通用 Java 和 Javac（Sun 公司的 Java 编译器），所以基于 Java 平台的 Scala 语言于 2003 年底/2004 年初发布。

截至到 2020 年 8 月，Scala 最新版本为 2.13.3，支持 JVM 和 JavaScript

Scala 官网：<https://www.scala-lang.org/>

1.1.2 为什么学习 Scala

在之前的学习中，我们已经学习了很长时间的 Java 语言，为什么此时要学习一门新的语言呢？主要基于以下几个原因：

- 1) 大数据主要的批处理计算引擎框架 Spark 是基于 Scala 语言开发的
- 2) 大数据主要的流式计算引擎框架 Flink 也提供了 Scala 相应的 API

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) 大数据领域中函数式编程的开发效率更高，更直观，更容易理解

1.1.3 Java and Scala

Martin Odersky 是狂热的编译器爱好者，长时间的编程后，希望开发一种语言，能够让写程序的过程变得简单，高效，所以当接触到 Java 语言后，感受到了这门语言的魅力，决定将函数式编程语言的特性融合到 Java 语言中，由此产生了 2 门语言（Pizza & Scala），这两种语言极大地推动了 Java 语言的发展

- JDK1.5 的泛型，增强 for 循环，自动类型转换等都是从 Pizza 语言引入的新特性
- JDK1.8 的类型推断， λ （lambda）表达式是从 Scala 语言引入的新特性

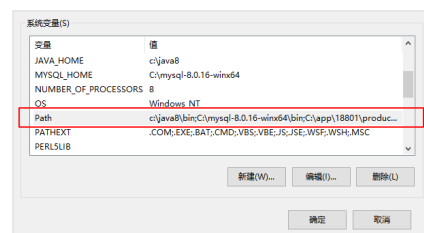
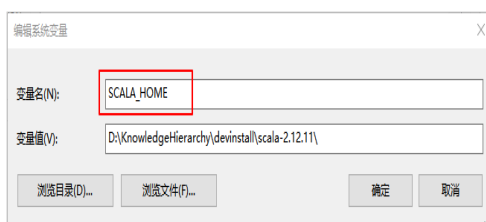
由上可知，Scala 语言是基于 Java 开发的，所以其编译后的文件也是字节码文件，并可以运行在 JVM 中。

1.2 快速上手

1.2.1 Scala 环境安装

- 1) 安装 JDK 1.8（略）
- 2) 安装 Scala2.12

- 解压文件：scala-2.12.11.zip，解压目录要求无中文无空格
- 配置环境变量



- 3) 环境测试

如果出现如下窗口内容，表示环境安装成功

```
C:\WINDOWS\system32\cmd.exe - scala
Microsoft Windows [版本 10.0.17763.1158]
(c) 2018 Microsoft Corporation. 保留所有权利。

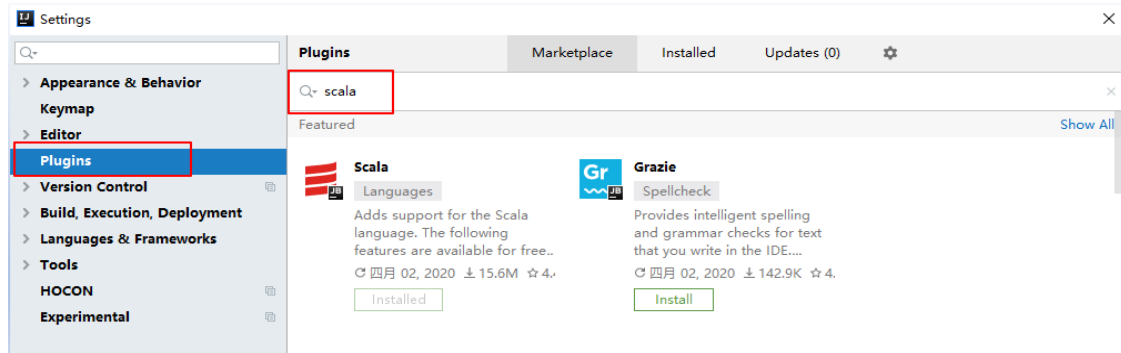
D:\Users\18801>scala
Welcome to Scala 2.12.11 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_111).
Type in expressions for evaluation. Or try :help.

scala> 10 + 10
res0: Int = 20

scala>
```

1.2.2 Scala 插件安装

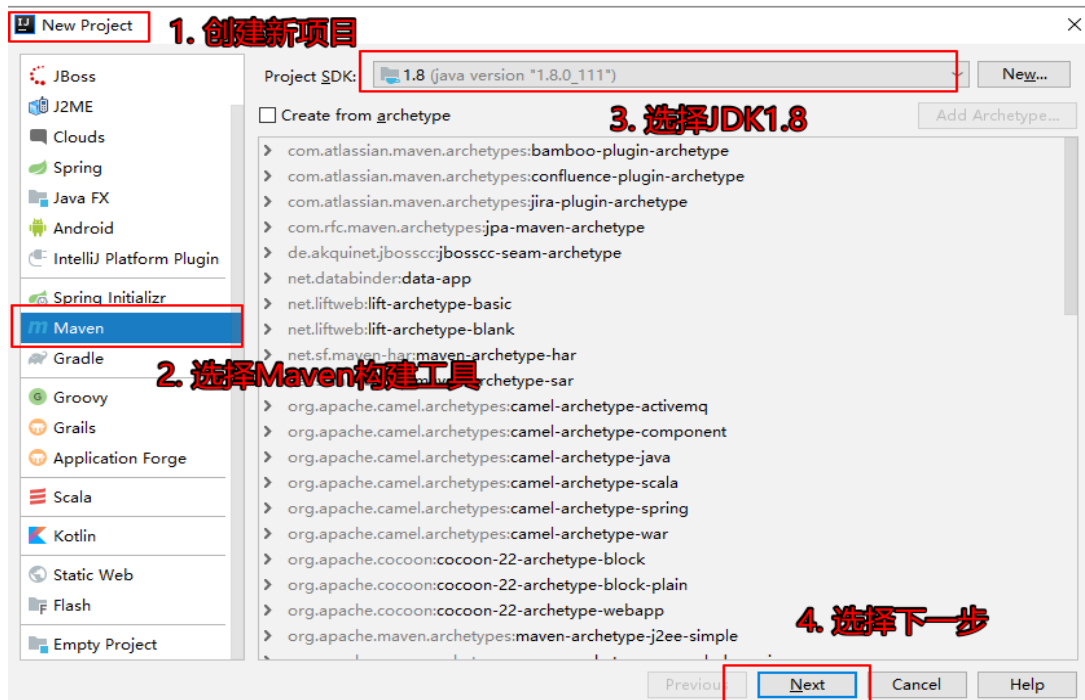
默认情况下 IDEA 不支持 Scala 的开发，需要安装 Scala 插件。

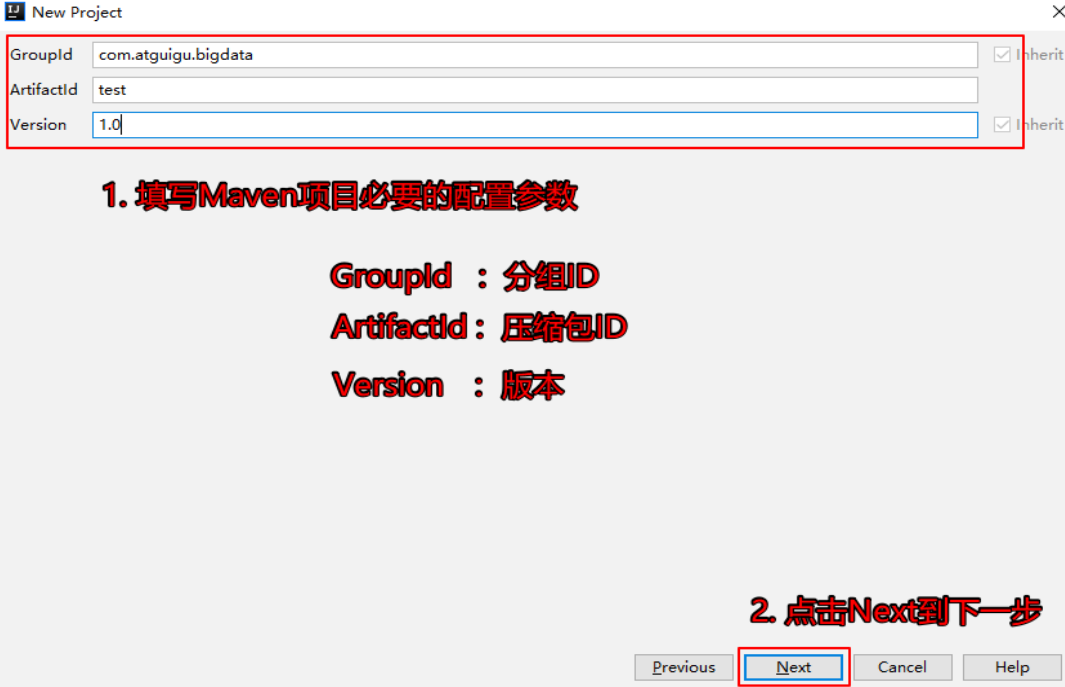


如果下载慢的，请访问网址：<https://plugins.jetbrains.com/plugin/1347-scala/versions>

1.2.3 Hello Scala 案例

1) 创建 Maven 项目



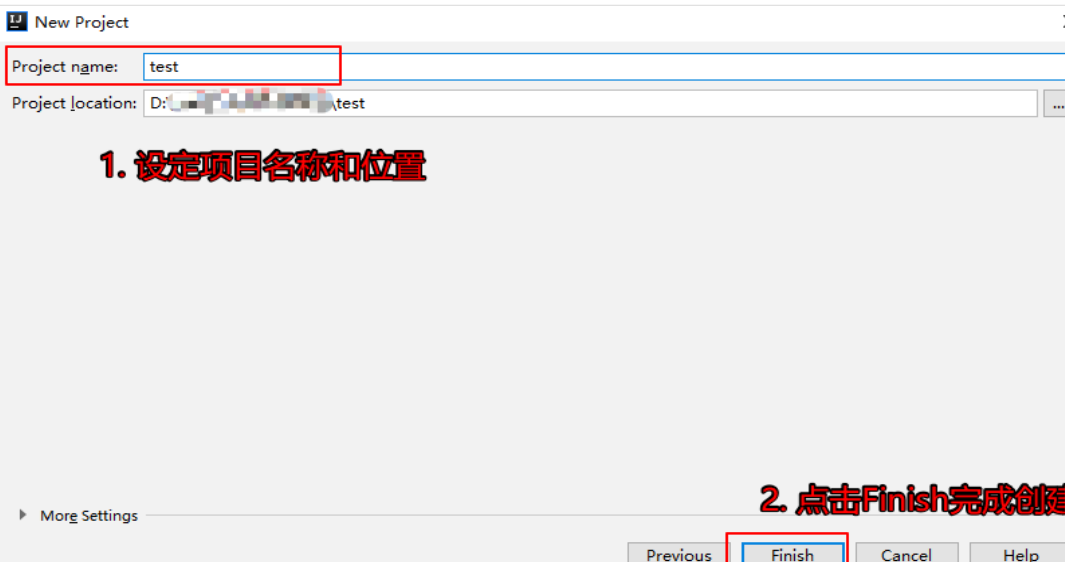


1. 填写Maven项目必要的配置参数

GroupId : 分组ID
ArtifactId : 压缩包ID
Version : 版本

2. 点击Next到下一步

Previous **Next** Cancel Help



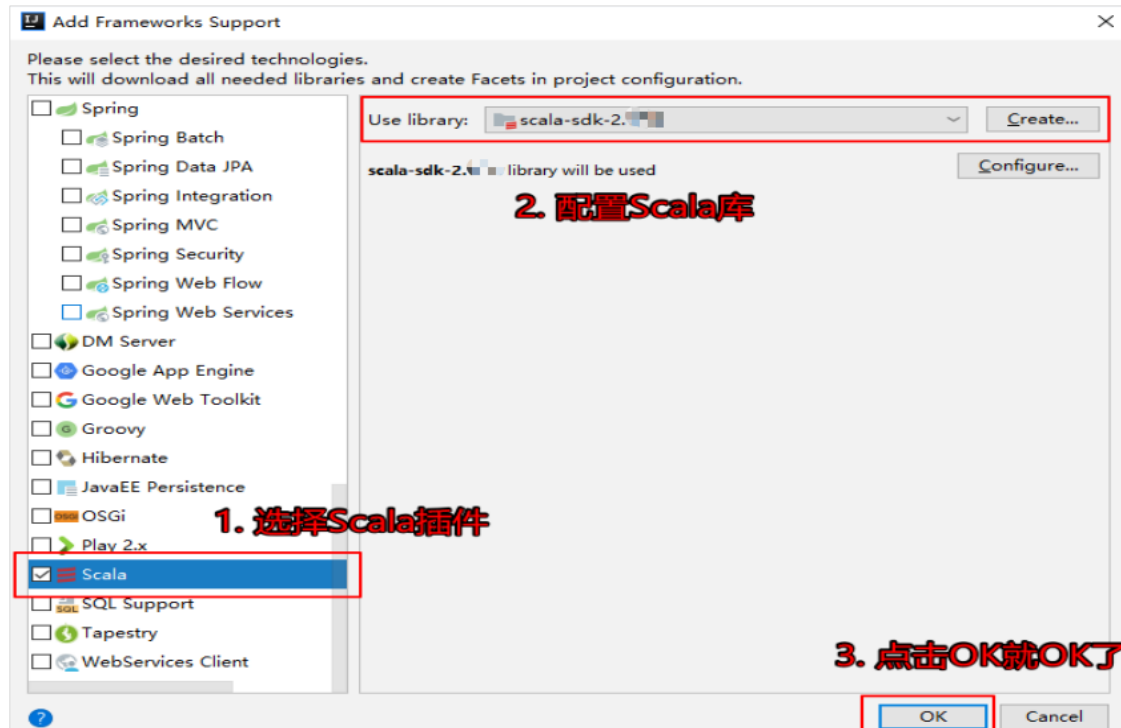
1. 设定项目名称和位置

2. 点击Finish完成创建

More Settings Previous **Finish** Cancel Help

2) 增加 Scala 框架支持

默认情况，IDEA 中创建项目时不支持 Scala 的开发，需要添加 Scala 框架的支持。



3) 创建类

在 main 文件目录中创建 Scala 类：com.atguigu.bigdata.scala.HelloScala

```
package com.atguigu.bigdata.scala

object HelloScala {
    def main(args: Array[String]): Unit = {
        System.out.println("Hello Scala")
        println("Hello Scala")
    }
}
```

4) 代码解析

- object
- def
- args : Array[String]
- Unit
- System.out.println
- println

如果只是通过代码来进行语法的解析，并不能了解其真正的实现原理。scala 语言是基于 Java 语言开发的，所以也会编译为 class 文件，那么我们可以通过反编译指令 javap

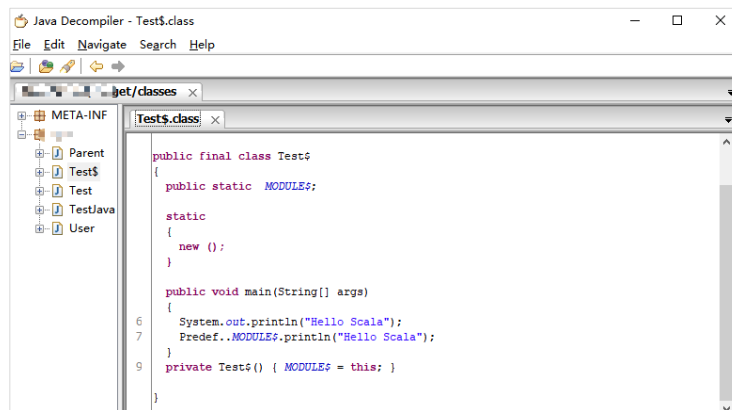
```
javap -c -l 类名
```

```
D:\workspace\idea\classes\classes-test\target\classes\com>javap -c -l Test$
警告: 二进制文件Test$包含com.Test$
Compiled from "Test.scala"
public final class com.Test$ {
    public static com.Test$ MODULE$;

    public static {};
    Code:
        0: new           #2              // class com/Test$
        3: invokespecial #12             // Method "<init>":()V
        6: return

    public void main(java.lang.String[]);
    Code:
        0: getstatic     #21             // Field java/lang/System.out:Ljava/io/PrintStream;
        3: ldc           #23             // String Hello Scala
        5: invokevirtual #29             // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8: getstatic     #34             // Field scala/Predef$.MODULE$:Lscala/Predef$;
        11: ldc           #23             // String Hello Scala
        13: invokevirtual #37             // Method scala/Predef$.println:(Ljava/lang/Object;)V
        16: return
   LineNumberTable:
        line 6: 0
        line 7: 8
    LocalVariableTable:
        Start Length Slot Name   Signature
        0      17     0  this   Lcom/Test$;
        0      17     1  args   [Ljava/lang/String;
```

或反编译工具 jd-gui.exe 查看 scala 编译后的代码。



通过对比和 java 语言之间的关系，来掌握具体代码的实现原理



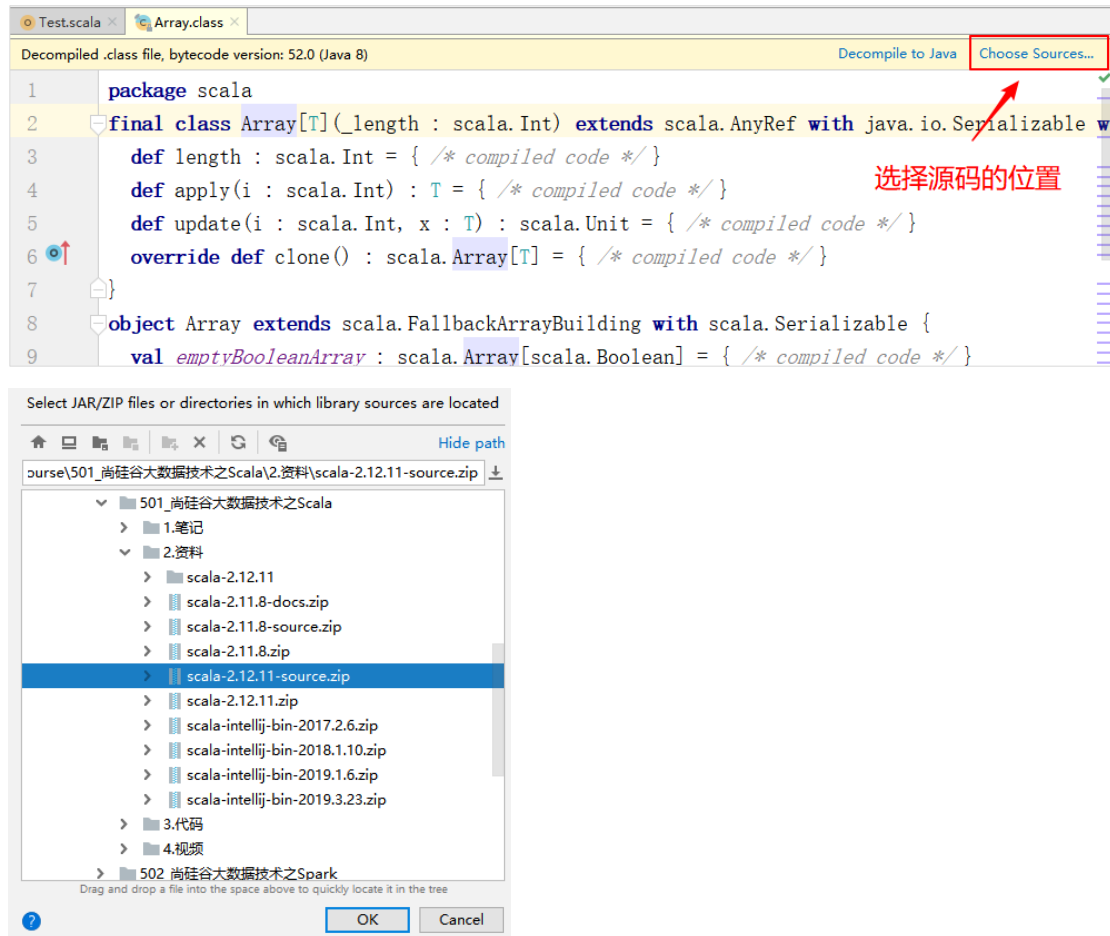
思考两个问题：

➤ 设置 path，classpath 环境变量的作用？

➤ IDEA 中哪里是 classpath？

5) 源码关联

在使用 Scala 过程中，为了搞清楚 Scala 底层的机制，需要查看源码，那么就需要关联和查看 Scala 的源码包。



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。

2.1.1 单行注释

```
package com.atguigu.bigdata.scala

object ScalaComment{
    def main(args: Array[String]): Unit = {
        // 单行注释
    }
}
```

2.1.2 多行注释

```
package com.atguigu.bigdata.scala

object ScalaComment{
    def main(args: Array[String]): Unit = {
        /*
            多行注释
        */
    }
}
```

2.1.3 文档注释

```
package com.atguigu.bigdata.scala
/**
 * doc 注释
 */
object ScalaComment{
    def main(args: Array[String]): Unit = {
    }
}
```

2.2 变量

变量是一种使用方便的占位符，用于引用计算机内存地址，变量创建后会占用一定的内存空间。基于变量的数据类型，操作系统会进行内存分配并且决定什么将被储存在保留内存中。因此，通过给变量分配不同的数据类型，你可以在这些变量中存储整数，小数或者字母。

2.2.1 语法声明

变量的类型在变量名之后等号之前声明。

```
object ScalaVariable {
    def main(args: Array[String]): Unit = {
        // var | val 变量名 : 变量类型 = 变量值
        // 用户名称
        var username : String = "zhangsan"
        // 用户密码
        val userpswd : String = "000000"
    }
}
```

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](#)

```
}  
}
```

变量的类型如果能够通过**变量值**推断出来，那么可以省略类型声明，这里的省略，并不是不声明，而是由 Scala 编译器在编译时自动声明编译的。

```
object ScalaVariable {  
  def main(args: Array[String]): Unit = {  
    // 因为变量值为字符串，又因为 Scala 是静态类型语言，所以即使不声明类型  
    // Scala 也能在编译时正确的判断出变量的类型，这体现了 Scala 语言的简洁特性。  
    var username = "zhangsan"  
    val userpswd = "000000"  
  }  
}
```

2.2.2 变量初始化

Java 语法中变量在**使用前**进行初始化就可以，但是 Scala 语法中是不允许的，必须**显示**进行初始化操作。

```
object ScalaVariable {  
  def main(args: Array[String]): Unit = {  
    var username // Error  
    val username = "zhangsan" // OK  
  }  
}
```

2.2.3 可变变量

值可以改变的变量，称之为**可变变量**，但是变量类型无法发生改变, Scala 中可变变量使用关键字 **var** 进行声明

```
object ScalaVariable {  
  def main(args: Array[String]): Unit = {  
    // 用户名称  
    var username : String = "zhangsan"  
    username = "lisi" // OK  
    username = true // Error  
  }  
}
```

2.2.4 不可变变量

值一旦初始化后无法改变的变量，称之为**不可变变量**。Scala 中不可变变量使用关键字 **val** 进行声明，**类似于** Java 语言中的 **final** 关键字

```
object ScalaVariable {  
  def main(args: Array[String]): Unit = {  
    // 用户名称  
    val username : String = "zhangsan"  
    username = "lisi" // Error  
    username = true // Error  
  }  
}
```



思考两个问题：

- val 和 var 两个修饰符，哪一个会推荐使用？
- Java 中的字符串为何称之为不可变字符串？

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

2.3 标识符

Scala 可以使用两种形式的标志符，字符数字和符号。

- 字符数字使用字母或是下划线开头，后面可以接字母或是数字，符号"\$"在 Scala 中也看作为字母。然而以"\$"开头的标识符为保留的 Scala 编译器产生的标志符使用，应用程序应该避免使用"\$"开始的标识符，以免造成冲突。
- Scala 的命名规范采用和 Java 类似的 camel 命名规范，首字符小写，比如 toString。类名的首字符还是使用大写。此外也应该避免使用以下划线结尾的标志符以避免冲突。
- Scala 内部实现时会使用转义的标志符，比如:-> 使用 \$colon\$minus\$greater 来表示这个符号。

```
// 和 Java 一样的标识符命名规则
val name = "zhangsan" // OK
val name1 = "zhangsan0" // OK
//val lname = "zhangsan0" // Error
val name$ = "zhangsan1" // OK
val $name = "zhangsan2" // OK
val name_ = "zhangsan3" // OK
val _name = "zhangsan4" // OK
val $ = "zhangsan5" // OK
val _ = "zhangsan6" // OK
//val l = "zhangsan6" // Error
//val true = "zhangsan6" // Error

// 和 Java 不一样的标识符命名规则
val + = "lisi" // OK
val - = "lisi" // OK
val * = "lisi" // OK
val / = "lisi" // OK
val ! = "lisi" // OK
//val @ = "lisi" // Error
val @@ = "lisi" // OK
//val # = "lisi" // Error
val ## = "lisi" // OK
val % = "lisi" // OK
val ^ = "lisi" // OK
val & = "lisi" // OK
//val ( = "lisi" // Error
//val ( = "lisi" // Error
//val ) = "lisi" // Error
//val = = "lisi" // Error
val == = "lisi" // OK
//val [ = "lisi" // Error
//val ] = "lisi" // Error
//val : = "lisi" // Error
val :: = "lisi" // OK
//val ; = "lisi" // Error
//val ' = "lisi" // Error
//val " = "lisi" // Error
val "" = "lisi" // OK
val < = "lisi" // OK
val > = "lisi" // OK
val ? = "lisi" // OK
val | = "lisi" // OK
```

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](#)

```
val \ = "lisi" // OK
//val ` = "lisi" // Error
val ~ = "lisi" // OK
val :-> = "wangwu" // OK
val :-< = "wangwu" // OK
// 切记，能声明和能使用是两回事
```

Scala 中的标识符也不能是**关键字**或**保留字**，那么 Scala 中有多少关键字或保留字呢？

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	null
object	override	package	private
protected	return	sealed	super
this	throw	trait	try
true	type	val	var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		



思考两个问题：

- 如何在 Java 语言中访问 Scala 对象？
- 如果变量就想使用特定含义的关键字怎么办？

2.4 字符串

在 Scala 中，字符串的类型实际上就是 Java 中的 String 类，它本身是没有 String 类的。

在 Scala 中，String 是一个不可变的字符串对象，所以该对象不可被修改。这就意味着你如果修改字符串就会产生一个新的字符串对象。

```
object ScalaString {
  def main(args: Array[String]): Unit = {
    val name : String = "scala"
    val subname : String = name.substring(0,2)
  }
}
```

2.4.1 字符串连接

```
object ScalaString {
  def main(args: Array[String]): Unit = {
    // 字符串连接
```

```
println("Hello " + name)
}
}
```

2.4.2 传值字符串

```
object ScalaString {
  def main(args: Array[String]): Unit = {
    // 传值字符串 (格式化字符串)
    printf("name=%s\n", name)
  }
}
```

2.4.3 插值字符串

```
object ScalaString {
  def main(args: Array[String]): Unit = {
    // 插值字符串
    // 将变量值插入到字符串
    println(s"name=${name}")
  }
}
```

2.4.4 多行字符串

```
object ScalaString {
  def main(args: Array[String]): Unit = {
    // 多行格式化字符串
    // 在封装 JSON 或 SQL 时比较常用
    // | 默认顶格符
    println(
      s"""
        | Hello
        | ${name}
      """.stripMargin)
  }
}
```

2.5 输入输出

2.5.1 输入

- 从屏幕（控制台）中获取输入

```
object ScalaIn {
  def main(args: Array[String]): Unit = {
    // 标准化屏幕输入
    val age : Int = scala.io.StdIn.readInt()
    println(age)
  }
}
```

- 从文件中获取输入

```
object ScalaIn {
  def main(args: Array[String]): Unit = {
    // 请注意文件路径的位置
    scala.io.Source.fromFile("input/user.json").foreach(
      line => {
        print(line)
      }
    )
    scala.io.Source.fromFile("input/user.json").getLines()
  }
}
```

2.5.2 输出

Scala 进行文件写操作，用的都是 java 中的 I/O 类

```
object ScalaOut {  
  def main(args: Array[String]): Unit = {  
    val writer = new PrintWriter(new File("output/test.txt" ))  
    writer.write("Hello Scala")  
    writer.close()  
  }  
}
```

2.5.3 网络

Scala 进行网络数据交互时，采用的也依然是 java 中的 I/O 类

```
object TestServer {  
  def main(args: Array[String]): Unit = {  
    val server = new ServerSocket(9999)  
    while ( true ) {  
      val socket: Socket = server.accept()  
      val reader = new BufferedReader(  
        new InputStreamReader(  
          socket.getInputStream,  
          "UTF-8"  
        )  
      )  
      var s : String = ""  
      var flg = true  
      while ( flg ) {  
        s = reader.readLine()  
        if ( s != null ) {  
          println(s)  
        } else {  
          flg = false  
        }  
      }  
    }  
  }  
}  
  
...  
  
object TestClient {  
  def main(args: Array[String]): Unit = {  
    val client = new Socket("localhost", 9999)  
    val out = new PrintWriter(  
      new OutputStreamWriter(  
        client.getOutputStream,  
        "UTF-8"  
      )  
    )  
    out.print("hello Scala")  
    out.flush()  
    out.close()  
    client.close()  
  }  
}
```



思考一个问题：java 的序列化怎么回事？

1. java 将内存中的对象存储到磁盘文件中，要求对象必须实现可序列化接口
2. 在网络中想要传递对象，这个对象需要序列化。

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

2.6 数据类型

Scala 与 Java 有着相同的数据类型，但是又有不一样的地方

2.6.1 Java 数据类型

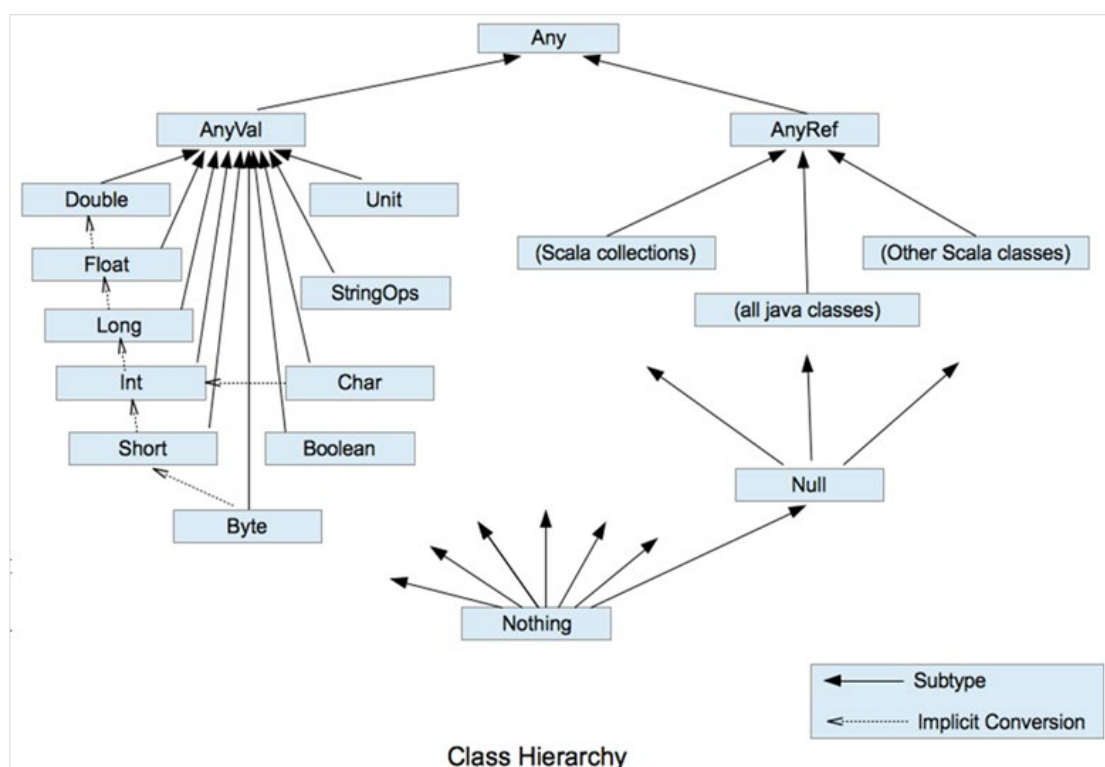
Java 的数据类型包含基本类型和引用类型

- 基本类型：byte,short,char,int,long,float,double,boolean
- 引用类型：Object，数组，字符串，包装类，集合，POJO 对象等

2.6.2 Scala 数据类型

Scala 是完全面向对象的语言，所以不存在基本数据类型的概念，有的只是任意值对象类型

(AnyVal) 和任意引用对象类型(AnyRef)



数据类型	描述
Byte	8位有符号补码整数。数值区间为 -128 到 127
Short	16位有符号补码整数。数值区间为 -32768 到 32767
Int	32位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long	64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807
Float	32 位, IEEE 754 标准的单精度浮点数
Double	64 位 IEEE 754 标准的双精度浮点数
Char	16位无符号Unicode字符, 区间值为 U+0000 到 U+FFFF
String	字符序列
Boolean	true或false
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 或空引用
Nothing	Nothing类型在Scala的类层级的最底端; 它是任何其他类型的子类型。
Any	Any是所有其他类的超类
AnyRef	AnyRef类是Scala里所有引用类(reference class)的基类

2.7 类型转换

2.7.1 自动类型转化（隐式转换）

```
object ScalaDataType {
  def main(args: Array[String]): Unit = {
    val b : Byte = 10
    val s : Short = b
    val i : Int = s
    val lon : Long = i
  }
}
```



思考一个问题：如下代码是否正确？

```
val c : Char = 'A' + 1
println(c)
```

2.7.2 强制类型转化

● Java 语言

```
int a = 10
byte b = (byte)a
```

● Scala 语言

```
var a : Int = 10
Var b : Byte = a.toByte
// 基本上 Scala 的 AnyVal 类型之间都提供了相应转换的方法。
```

2.7.3 字符串类型转化

scala 是完全面向对象的语言，所有的类型都提供了 toString 方法，可以直接转换为字符串

```
lon.toString
```

任意类型都提供了和字符串进行拼接的方法

```
val i = 10
val s = "hello " + i
```


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 算数运算符

假定变量 A 为 10, B 为 20

运算符	描述	实例
+	加号	A + B 运算结果为 30
-	减号	A - B 运算结果为 -10
*	乘号	A * B 运算结果为 200
/	除号	B / A 运算结果为 2
%	取余	B % A 运算结果为 0

3.2 关系运算符

假定变量 A 为 10, B 为 20

运算符	描述	实例
==	等于	(A == B) 运算结果为 false
!=	不等于	(A != B) 运算结果为 true
>	大于	(A > B) 运算结果为 false
<	小于	(A < B) 运算结果为 true
>=	大于等于	(A >= B) 运算结果为 false
<=	小于等于	(A <= B) 运算结果为 true



思考一个问题：如下代码执行结果如何？

```
val a = new String("abc")
val b = new String("abc")

println(a == b)
println(a.equals(b))
println(a.eq(b))
```

3.3 赋值运算符

运算符	描述	实例
=	简单的赋值运算，指定右边操作数赋值给左边的操作数。	C = A + B 将 A + B 的运算结果赋值给 C
+=	相加后再赋值，将左右两边的操作数相加后再赋值给左边的操作数。	C += A 相当于 C = C + A
-=	相减后再赋值，将左右两边的操作数相减后再赋值给左边的操作数。	C -= A 相当于 C = C - A
*=	相乘后再赋值，将左右两边的操作数相乘后再赋值给左边的操作数。	C *= A 相当于 C = C * A
/=	相除后再赋值，将左右两边的操作数相除后再赋值给左边的操作数。	C /= A 相当于 C = C / A
%=	求余后再赋值，将左右两边的操作数求余后再赋值给左边的操作数。	C %= A is equivalent to C = C % A
<<=	按位左移后再赋值	C <<= 2 相当于 C = C << 2
>>=	按位右移后再赋值	C >>= 2 相当于 C = C >> 2
&=	按位与运算后赋值	C &= 2 相当于 C = C & 2
^=	按位异或运算符后再赋值	C ^= 2 相当于 C = C ^ 2
=	按位或运算后赋值	C = 2 相当于 C = C 2



思考一个问题：为什么在上面的运算符中没有看到 ++， --？

++运算有歧义，容易理解出现错误，所以 scala 中没有这样的语法，所以采用 +=的方式来代替。

3.4 逻辑运算符

假定变量 A 为 1，B 为 0

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

3.5 位运算符

如果指定 A = 60; 及 B = 13; 两个变量对应的二进制为

```
A = 0011 1100
B = 0000 1101
```

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：1111 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



思考一个问题：位运算有啥用？你用过吗或你接触过吗？

3.6 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。

- scala 是完全面向对象的语言，所以数字其实也是对象
- 当调用对象的方法时，点.可以省略
- 如果函数参数只有一个，或者没有参数，()可以省略

```
object ScalaOper {
  def main(args: Array[String]): Unit = {
    val i : Int = 10
    val j : Int = i.+(10)
    val k : Int = j +(20)
    val m : Int = k + 30
    println(m)
  }
}
```

第4章 流程控制

Scala 程序代码和所有编程语言代码一样，都会有特定的执行流程顺序，默认情况下是顺序执行，上一条逻辑执行完成后才会执行下一条逻辑，执行期间也可以根据某些条件执行不同的分支逻辑代码。

4.1 分支控制

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

IF...ELSE 语句是通过一条或多条语句的执行结果（**true** 或者 **false**）来决定执行的代码块

```
if(布尔表达式) {  
    // 如果布尔表达式为 true 则执行该语句块  
}
```

如果布尔表达式为 **true** 则执行大括号内的语句块，否则跳过大括号内的语句块，执行大括号之后的语句块。

```
object ScalaBranch {  
    def main(args: Array[String]): Unit = {  
        val b = true  
        if ( b ) {  
            println("true")  
        }  
    }  
}
```

4.1.2 双分支

```
if(布尔表达式) {  
    // 如果布尔表达式为 true 则执行该语句块  
} else {  
    // 如果布尔表达式为 false 则执行该语句块  
}
```

如果布尔表达式为 **true** 则执行接着的大括号内的语句块，否则执行 **else** 后的大括号内的语句块。

```
object ScalaBranch {  
    def main(args: Array[String]): Unit = {  
        val b = true  
        if ( b ) {  
            println("true")  
        } else {  
            println("false")  
        }  
    }  
}
```

4.1.3 多分支

```
if(布尔表达式 1) {  
    // 如果布尔表达式 1 为 true，则执行该语句块  
} else if ( 布尔表达式 2 ) {
```

```
// 如果布尔表达式 2 为 true，则执行该语句块
}...
} else {
    // 上面条件都不满足的场合，则执行该语句块
}
```

实现一个小功能：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“青年”，如果年龄大于 30 小于等于 50，则输出“中年”，否则，输出“老年”。

```
object ScalaBranch {
    def main(args: Array[String]): Unit = {
        val age = 30
        if ( age < 18 ) {
            println("童年")
        } else if ( age <= 30 ) {
            println("青年")
        } else if ( age <= 50 ) {
            println("中年")
        } else {
            println("老年")
        }
    }
}
```

实际上，**Scala 中的表达式都是有返回值的**，所以上面的小功能还有其他的实现方式

```
object ScalaBranch {
    def main(args: Array[String]): Unit = {
        val age = 30
        val result = if ( age < 18 ) {
            "童年"
        } else if ( age <= 30 ) {
            "青年"
        } else if ( age <= 50 ) {
            "中年"
        } else {
            "老年"
        }
        println(result)
    }
}
```



思考一个问题：怎么没有讲三元运算符？

Scala 语言中没有三元运算符的，使用 if 分支判断来代替三元运算符

4.2 循环控制

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。编程语言提供了更为复杂执行路径的多种控制结构。循环语句允许我们多次执行一个语句或语句组

Scala 语言提供了以下几种循环类型

循环类型	描述
while 循环	运行一系列语句，如果条件为true，会重复运行，直到条件变为false。
do...while 循环	类似 while 语句区别在于判断循环条件之前，先执行一次循环的代码块。
for 循环	用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。

4.2.1 for 循环

1) 基本语法

```
for ( 循环变量 <- 数据集 ) {  
    循环体  
}
```

这里的数据集可以是任意类型的数据集合，如字符串，集合，数组等，这里我们还没有讲到集合，数组语法，请大家不要着急，先能演示例子，后面咱们详细讲。

```
object ScalaLoop {  
    def main(args: Array[String]): Unit = {  
        for ( i <- Range(1,5) ) { // 范围集合  
            println("i = " + i )  
        }  
        for ( i <- 1 to 5 ) { // 包含 5  
            println("i = " + i )  
        }  
        for ( i <- 1 until 5 ) { // 不包含 5  
            println("i = " + i )  
        }  
    }  
}
```

2) 循环守卫

循环时可以增加条件来决定是否继续循环体的执行,这里的判断条件我们称之为循环守卫

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

3) 循环步长

scala 的集合也可以设定循环的增长幅度，也就是所谓的步长 step

```
object ScalaLoop {  
    def main(args: Array[String]): Unit = {  
        for ( i <- Range(1,5,2) ) {  
            println("i = " + i )  
        }  
        for ( i <- 1 to 5 by 2 ) {  
            println("i = " + i )  
        }  
    }  
}
```

4) 循环嵌套

```
object ScalaLoop {  
    def main(args: Array[String]): Unit = {  
        for ( i <- Range(1,5); j <- Range(1,4) ) {
```

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


```
}
```

一种特殊的 **while** 循环就是，先执行循环体，再判断循环条件是否成立

```
do {  
    循环体  
} while ( 循环条件表达式 )
```

2) while 循环

```
object ScalaLoop {  
    def main(args: Array[String]): Unit = {  
        var i = 0  
        while ( i < 5 ) {  
            println(i)  
            i += 1  
        }  
    }  
}
```

3) do...while 循环

```
object ScalaLoop {  
    def main(args: Array[String]): Unit = {  
        var i = 5  
        do {  
            println(i)  
        } while ( i < 5 )  
    }  
}
```

4.2.3 循环中断

scala 是完全面向对象的语言，所以无法使用 **break**，**continue** 关键字这样的方式来中断，或继续循环逻辑，而是采用了函数式编程的方式代替了循环语法中的 **break** 和 **continue**

```
object ScalaLoop {  
    def main(args: Array[String]): Unit = {  
        scala.util.control.Breaks.breakable {  
            for ( i <- 1 to 5 ) {  
                if ( i == 3 ) {  
                    scala.util.control.Breaks.break  
                }  
                println(i)  
            }  
        }  
    }  
}
```

4.2.4 嵌套循环

循环中有循环，就是嵌套循环。通过嵌套循环可以实现特殊的功能，比如说九九乘法表，这里自己练习吧，之前不是刚写了个九层妖塔的例子吗？，难不成你不会？



第5章 函数式编程

在之前 Java 课程的学习中，我们一直学习的就是面向对象编程，所以解决问题都是按照面向对象的方式来处理的。比如用户登陆等业务功能，但是接下来，我们会学习函数式编程，采用函数式编程的思路来解决问题。scala 编程语言将函数式编程和面向对象编程完美地融合在一起了。

- 面向对象编程

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

- 函数式编程

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功能按照指定的步骤，解决问题。

5.1 基础函数编程

5.1.1 基本语法

```
[修饰符] def 函数名 ( 参数列表 ) [:返回值类型] = {  
    函数体  
}  
  
private def test( s : String ) : Unit = {  
    println(s)  
}
```

5.1.2 函数&方法

- scala 中存在方法与函数两个不同的概念，二者在语义上的区别很小。scala 方法是类的一部分，而函数是一个对象，可以赋值给一个变量。换句话说在类中定义的函数即是方法。scala 中的方法跟 Java 的类似，方法是组成类的一部分。scala 中的函数则是一个完整的对象。
- Scala 中的方法和函数从语法概念上来讲，一般不好区分，所以简单的理解就是：方法也是函数。只不过类中声明的函数称之为方法，其他场合声明的就是函数了。类中的方法是有重载和重写的。而函数可就没有重载和重写的概念了，但是函数可以嵌套声明使用，方法就没有这个能力了，千万记得哟。

5.1.3 函数定义

1) 无参，无返回值

```
object ScalaFunction {  
    def main(args: Array[String]): Unit = {  
        def fun1(): Unit = {  
            println("函数体")  
        }  
    }  
}
```

```
    }  
    fun1()  
  }  
}
```

2) 无参，有返回值

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun2(): String = {  
      "zhangsan"  
    }  
    println( fun2() )  
  }  
}
```

3) 有参，无返回值

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun3( name:String ): Unit = {  
      println( name )  
    }  
    fun3("zhangsan")  
  }  
}
```

4) 有参，有返回值

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun4(name:String): String = {  
      "Hello " + name  
    }  
    println( fun4("zhangsan") )  
  }  
}
```

5) 多参，无返回值

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun5(hello:String, name:String): Unit = {  
      println( hello + " " + name )  
    }  
    fun5("Hello", "zhangsan")  
  }  
}
```

6) 多参，有返回值

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun6(hello:String, name:String): String = {  
      hello + " " + name  
    }  
    println( fun6("Hello", "zhangsan") )  
  }  
}
```

5.1.4 函数参数

1) 可变参数

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun7(names:String*): Unit = {  
      println(names)  
    }  
  }  
}
```

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

```
}
fun7()
fun7( "zhangsan" )
fun7( "zhangsan", "lisi" )
}
```

可变参数不能放置在参数列表的前面，一般放置在参数列表的最后

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    // Error
    //def fun77(names:String*, name:String): Unit = {

    //}
    def fun777( name:String, names:String* ): Unit = {
      println( name )
      println( names )
    }
  }
}
```

2) 参数默认值

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun8( name:String, password:String = "000000" ): Unit = {
      println( name + "," + password )
    }
    fun8("zhangsan", "123123")
    fun8("zhangsan")
  }
}
```

3) 带名参数

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun9( password:String = "000000", name:String ): Unit = {
      println( name + "," + password )
    }
    fun9("123123", "zhangsan" )
    fun9(name="zhangsan")
  }
}
```

5.1.5 函数至简原则

所谓的至简原则，其实就是 Scala 的作者为了开发人员能够大幅度提高开发效率。通过编译器的动态判定功能，帮助我们将函数声明中能简化的地方全部都进行了简化。也就是说将函数声明中那些能省的地方全部都省掉。所以这里的**至简原则**，简单来说就是：**能省则省**。

1) 省略 return 关键字

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun1(): String = {
      return "zhangsan"
    }
    def fun11(): String = {
      "zhangsan"
    }
  }
}
```

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) 省略花括号

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun2(): String = "zhangsan"  
  }  
}
```

3) 省略返回值类型

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun3() = "zhangsan"  
  }  
}
```

4) 省略参数列表

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun4 = "zhangsan"  
    fun4// OK  
    fun4()// (ERROR)  
  }  
}
```

5) 省略等号

如果函数体中有明确的 **return** 语句，那么返回值类型不能省略

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun5(): String = {  
      return "zhangsan"  
    }  
    println(fun5())  
  }  
}
```

如果函数体返回值类型明确为 **Unit**，那么函数体中即使有 **return** 关键字也不起作用

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun5(): Unit = {  
      return "zhangsan"  
    }  
    println(fun5())  
  }  
}
```

如果函数体返回值类型声明为 **Unit**，但是又想省略，那么此时就必须连同等号一起省略

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun5() {  
      return "zhangsan"  
    }  
    println(fun5())  
  }  
}
```

6) 省略名称和关键字

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    () => {  
      println("zhangsan")  
    }  
  }  
}
```

```
}
```

5.2 高阶函数编程

所谓的高阶函数，其实就是将函数当成一个类型来使用，而不是当成特定的语法结构。

5.2.1 函数作为值

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun1(): String = {
      "zhangsan"
    }
    val a = fun1
    val b = fun1 _
    val c : ()=>Unit = fun1
    println(a)
    println(b)
  }
}
```

5.2.2 函数作为参数

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun2( i:Int ): Int = {
      i * 2
    }
    def fun22( f : Int => Int ): Int = {
      f(10)
    }
    println(fun22(fun2))
  }
}
```

5.2.3 函数作为返回值

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun3( i:Int ): Int = {
      i * 2
    }
    def fun33( ) = {
      fun3 _
    }
    println(fun33()(10))
  }
}
```

5.2.4 匿名函数

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun4( f:Int => Int ): Int = {
      f(10)
    }
    println(fun4((x:Int)=>{x * 20}))
    println(fun4((x)=>{x * 20}))
    println(fun4((x)=>x * 20))
    println(fun4(x=>x * 20))
    println(fun4(_ * 20))
  }
}
```

5.2.7 控制抽象

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun7(op: => Unit) = {
      op
    }
    fun7{
      println("xx")
    }
  }
}
```

5.2.5 闭包

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun5() = {
      val i = 20
      def fun55() = {
        i * 2
      }
      fun55 _
    }
    fun5() ()
  }
}
```



思考一个问题: 没有使用外部变量还能称之为闭包吗?

5.2.6 函数柯里化

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun6(i: Int)(j: Int) = {
      i * j
    }
  }
}
```

5.2.7 递归函数

```
object ScalaFunction {
  def main(args: Array[String]): Unit = {
    def fun8(j: Int): Int = {
      if ( j <= 1 ) {
        1
      } else {
        j * fun8(j-1)
      }
    }
    println(fun8(5))
  }
}
```



思考两个问题:

- 递归常用吗?
- 递归会出问题吗?

5.2.9 惰性函数

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。

```
object ScalaFunction {  
  def main(args: Array[String]): Unit = {  
    def fun9(): String = {  
      println("function...")  
      "zhangsan"  
    }  
    lazy val a = fun9()  
    println("-----")  
    println(a)  
  }  
}
```



思考两个问题:

- 函数到底是什么？实现原理？
- 如果一台机器想让另外一台机器执行函数功能怎么办？

第6章 面向对象编程

Scala 是一门**完全**面向对象的语言，摒弃了 Java 中很多不是面向对象的语法。虽然如此，但其面向对象思想和 Java 的面向对象思想还是一致的

6.1 基础面向对象编程

6.1.1 包

1) 基本语法

Scala 中基本的 package 包语法和 Java 完全一致

```
package com.atguigu.bigdata.scala
```

2) 扩展语法

Java 中 package 包的语法比较单一，Scala 对此进行扩展

- Scala 中的包和类的物理路径没有关系
- package 关键字可以嵌套声明使用

```
package com
package atguigu {
  package bigdata {
    package scala {
      object ScalaPackage {
        def test(): Unit = {
          println("test...")
        }
      }
    }
  }
}
```

- 同一个**源码文件**中子包可以直接访问父包中的内容，而无需 import

```
package com
package atguigu {
  package bigdata {
    class Test {
    }
  }
  package scala {
    object ScalaPackage {
      def test(): Unit = {
        new Test()
      }
    }
  }
}
```

- Scala 中 package 也可以看作对象，并声明属性和函数

```
package com
package object atguigu {
  val name : String = "zhangsan"
  def test(): Unit = {
    println( name )
  }
}
```

```
}  
package atguigu {  
  package bigdata {  
    package scala {  
      object ScalaPackage {  
        def test(): Unit = {  
        }  
      }  
    }  
  }  
}
```

6.1.2 导入

1) 基本语法

Scala 中基本的 import 导入语法和 Java 完全一致

```
import java.util.List  
import java.util._ // Scala 中使用下划线代替 Java 中的星号
```

2) 扩展语法

Java 中 import 导入的语法比较单一，Scala 对此进行扩展

- Scala 中的 import 语法可以在任意位置使用

```
object ScalaImport{  
  def main(args: Array[String]): Unit = {  
    import java.util.ArrayList  
    new ArrayList()  
  }  
}
```

- Scala 中可以导包，而不是导类

```
object ScalaImport{  
  def main(args: Array[String]): Unit = {  
    import java.util  
    new util.ArrayList()  
  }  
}
```

- Scala 中可以在同一行中导入相同包中的多个类，简化代码

```
import java.util.{List, ArrayList}
```

- Scala 中可以屏蔽某个包中的类

```
import java.util._  
import java.sql.{Date=>_, Array=>_, _ }
```

- Scala 中可以给类起别名，简化使用

```
import java.util.{ArrayList=>AList}  
  
object ScalaImport{  
  def main(args: Array[String]): Unit = {  
    new AList()  
  }  
}
```

- Scala 中可以使用类的绝对路径而不是相对路径

```
import _root_.java.util.ArrayList
```

- 默认情况下，Scala 中会导入如下包和对象

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

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

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



思考一个问题：导入对象？

6.1.3 类

面向对象编程中类可以看成是一个模板，而对象可以看成是根据模板所创建的具体事物

1) 基本语法

```
// 声明类：访问权限 class 类名 { 类主体内容 }
class User {
    // 类的主体内容
}
// 对象：new 类名(参数列表)
new User()
```

2) 扩展语法

Scala 中一个源文件中可以声明多个公共类

6.1.4 属性

1) 基本语法

```
class User {
    var name : String = _ // 类属性其实就是类变量
    var age : Int = _ // 下划线表示类的属性默认初始化
}
```

2) 扩展语法

Scala 中的属性其实在编译后也会生成方法

```
class User {
    var name : String = _
    val age : Int = 30
    private val email : String = _
    @BeanProperty var address : String = _
}
```

6.1.5 访问权限

Scala 中的访问权限和 Java 中的访问权限类似，但是又有区别：

```
private : 私有访问权限
private[包名] : 包访问权限
protected : 受保护权限，不能同包
             : 公共访问权限
```



思考一个问题：你会调用 java 中的 clone 方法吗？

6.1.6 方法

Scala 中的类的方法其实就是函数，所以声明方式完全一样，但是必须通过使用对象进行调用

```
object ScalaMethod{
    def main(args: Array[String]): Unit = {
        val user = new User
        user.login("zhangsan", "000000")
    }
}
class User {
```

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 login( name:String, password:String ): Boolean = {  
    false  
}  
}
```



思考两个问题：还记得方法的重写和重载吗？你真的明白吗？

6.1.7 对象

Scala 中的对象和 Java 是类似的

```
val | var 对象名 [: 类型] = new 类型()  
var user : User = new User()
```

6.1.8 构造方法

和 Java 一样，Scala 中构造对象也需要调用类的构造方法来创建。并且一个类中可以有任意多个不相同的构造方法。这些构造方法可以分为 2 大类：[主构造函数](#)和[辅助构造函数](#)。

```
class User() { // 主构造函数  
    var username : String = _  
    def this( name:String ) { // 辅助构造函数，使用 this 关键字声明  
        this() // 辅助构造函数应该直接或间接调用主构造函数  
        username = name  
    }  
    def this( name:String, password:String ) {  
        this(name) // 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  
    }  
}
```

6.2 高阶面向对象编程

6.2.1 继承

和 Java 一样，Scala 中的继承也是单继承，且使用 `extends` 关键字。

```
class Person {  
}  
class User extends Person {  
}
```

构造对象时需要考虑构造方法的执行顺序

6.2.2 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），才能对数据进行访问。

- 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



思考一个问题：真的有必要吗？

6.2.3 抽象

- Scala 将一个不完整的类称之为抽象类。

```
abstract class Person {  
}
```

- Scala 中如果一个方法只有声明而没有实现，那么是抽象方法，因为它不完整。

```
abstract class Person {  
    def test():Unit  
}
```

- Scala 中如果一个属性只有声明没有初始化，那么是抽象属性，因为它不完整。

```
abstract class Person {  
    var name:String  
}
```

- 子类如果继承抽象类，必须实现抽象方法或补全抽象属性，否则也必须声明为抽象的，因为依然不完整。

```
abstract class Person {  
    var name:String  
}  
class User extends Person {  
    var name : String = "zhangsan"  
}
```

6.2.4 单例对象

- 所谓的单例对象，就是在程序运行过程中，指定类的对象只能创建一个，而不能创建多个。这样的对象可以由特殊的设计方式获得，也可以由语言本身设计得到，比如 object 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，然后通过伴生对象名称直接调用
- 如果类名和伴生对象名称保持一致，那么这个类称之为伴生类。Scala 编译器可以通过伴生对象的 apply 方法创建伴生类对象。apply 方法可以重载，并传递参数，且可由 Scala 编译器自动识别。所以在使用时，其实是可以省略的。

```
class User { // 伴生类  
}  
object User { // 伴生对象  
    def apply() = new User() // 构造伴生类对象  
}  
...  
val user1 = new User() // 通过构造方法创建对象  
val user2 = User.apply() // 通过伴生对象的 apply 方法构造伴生类对象  
val user3 = User() // scala 编译器省略 apply 方法，自动完成调用
```



思考一个问题: Thread 线程中 wait 方法和 sleep 方法的区别?

6.2.5 特质

Scala 将多个类的相同特征从类中剥离出来，形成一个独立的语法结构，称之为“特质”（特征）。这种方式在 Java 中称之为接口，但是 Scala 中没有接口的概念。所以 scala 中没有 interface 关键字，而是采用特殊的关键字 trait 来声明特质，如果一个类符合某一个特征（特质），那么就可以将这个特征（特质）“混入”到类中。这种混入的操作可以在声明类时使用，也可以在创建类对象时动态使用。

1) 基本语法

```
trait 特质名称
class 类名 extends 父类(特质1) with 特质2 with 特质3
trait Operator {

}
trait DB{

}
class MySQL extends Operator with DB{

}
```



思考一个问题: 特质到底是什么？为什么又可以使用 extends，又可以使用 with？

2) 动态混入

```
object ScalaTrait{
  def main(args: Array[String]): Unit = {
    val mysql = new MySQL with Operator
    mysql.insert()
  }
}
trait Operator {
  def insert(): Unit = {
    println("insert data...")
  }
}
class MySQL {

}
```

3) 初始化叠加

```
object ScalaTrait{
  def main(args: Array[String]): Unit = {
    val mysql = new MySQL
  }
}
trait Operator {
  println("operator...")
}
trait DB {
  println("db...")
}
class MySQL extends DB with Operator{
  println("mysql...")
}
```

}

4) 功能叠加

```
object ScalaTrait {
  def main(args: Array[String]): Unit = {
    val mysql: MySQL = new MySQL
    mysql.operData()
  }
}

trait Operate{
  def operData():Unit={
    println("操作数据。。")
  }
}

trait DB extends Operate{
  override def operData(): Unit = {
    print("向数据库中。。")
    super.operData()
  }
}

trait Log extends Operate{

  override def operData(): Unit = {
    super.operData()
  }
}

class MySQL extends DB with Log {

}
```



思考一个问题: scala 中的 super 是什么?

6.2.6 扩展

● 类型检查和转换

```
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)
  }
}
```



思考一个问题: 字符串真的不可变吗?

● 枚举类和应用类

```
object 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](#)


```
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 AppTest extends App {
    println("application");
}
```

- Type 定义新类型

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

```
object Test {
    def main(args: Array[String]): Unit = {
        type S = String
        var v : S = "abc"
    }
}
```

第7章 集合

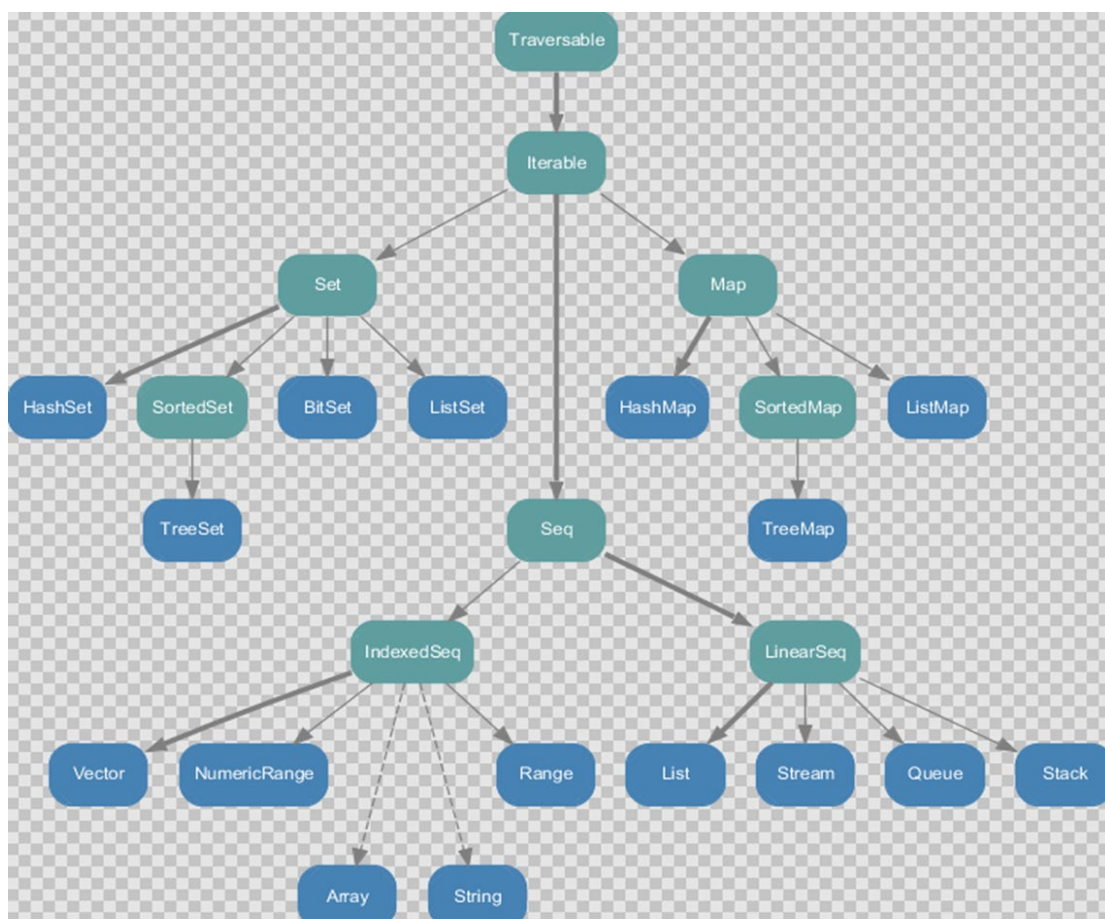
7.1 简介

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。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。

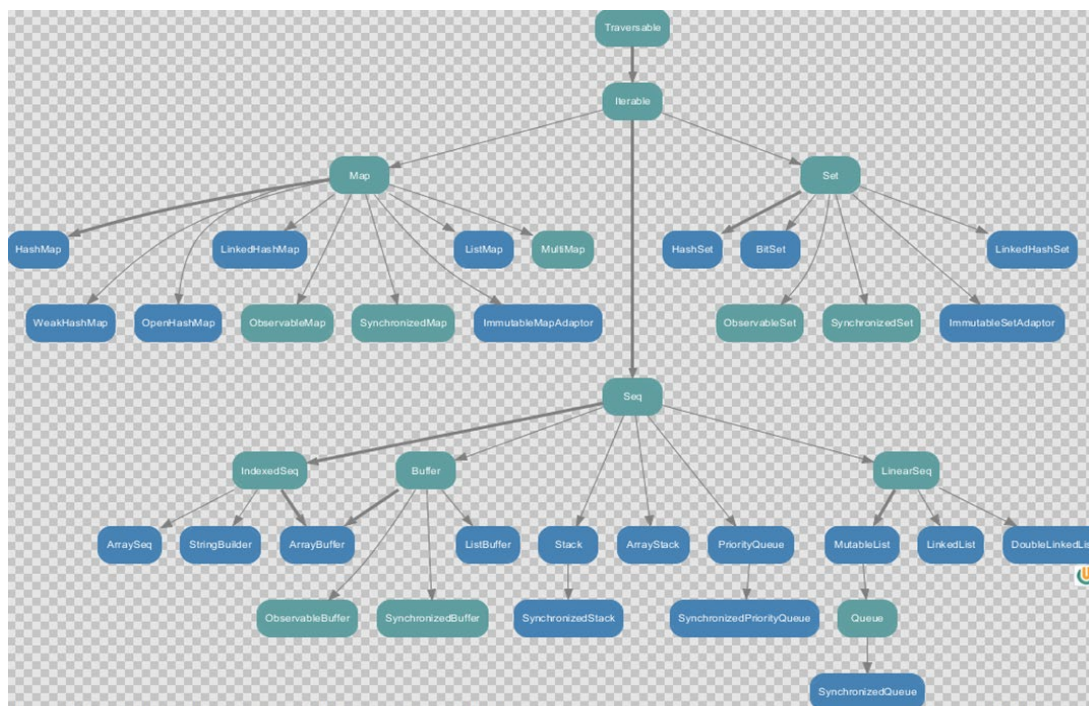
可变集合可以在适当的地方被更新或扩展。这意味着你可以修改，添加，移除一个集合的元素。而不可变集合类，相比之下，永远不会改变。不过，你仍然可以模拟添加，移除或更新操作。但是这些操作将在每一种情况下都返回一个新的集合，同时使原来的集合不发生改变，所以这里的不可变并不是变量本身的值不可变，而是变量指向的那个内存地址不可变

可变集合和不可变集合，在 scala 中该如何进行区分呢？我们一般可以根据集合所在包名进行区分：

➤ scala.collection.immutable



➤ scala.collection.mutable



7.2 数组

7.2.1 不可变数组

1) 基本语法

```

object ScalaCollection{
  def main(args: Array[String]): Unit = {
    // (1) 数组定义
    val arr01 = new Array[Int](4)
    println(arr01.length) // 4

    // (2) 数组赋值
    // (2.1) 修改某个元素的值
    arr01(3) = 10
    val i = 10
    arr01(i/3) = 20
    // (2.2) 采用方法的形式修改数组的值
    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)})
  }
}

```

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

```
    arr01.foreach(println(_))
    arr01.foreach(println)
  }
}
```

2) 基本操作

```
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    // 创建数组的另外一种方式
    val arr1 = Array(1,2,3,4)
    val arr2 = Array(5,6,7,8)
    // 添加数组元素，创建新数组
    val arr3: Array[Int] = arr1 :+ 5
    println( arr1 eq arr3 ) // false

    val arr4: Array[Int] = arr1 ++: arr2
    // 添加集合
    val arr5: Array[Int] = arr1 ++ arr2

    arr4.foreach(println)
    println("*****")
    arr5.foreach(println)
    println("*****")
    // 多维数组
    var myMatrix = Array.ofDim[Int](3,3)
    myMatrix.foreach(list=>list.foreach(println))
    // 合并数组
    val arr6: Array[Int] = Array.concat(arr1, arr2)
    arr6.foreach(println)

    // 创建指定范围的数组
    val arr7: Array[Int] = Array.range(0,2)
    arr7.foreach(println)

    // 创建并填充指定数量的数组
    val arr8: Array[Int] = Array.fill[Int](5)(-1)
    arr8.foreach(println)
  }
}
```

7.2.2 可变数组

1) 基本语法

```
import scala.collection.mutable.ArrayBuffer
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    val buffer = new ArrayBuffer[Int]
    // 增加数据
    buffer.append(1,2,3,4)
    // 修改数据
    buffer.update(0,5)
    buffer(1) = 6
    // 删除数据
    val i: Int = buffer.remove(2)
    buffer.remove(2,2)
    // 查询数据
    println(buffer(3))
    // 循环集合
    for ( i <- buffer ) {
      println(i)
    }
  }
}
```

```
}
```

2) 基本操作

```
import scala.collection.mutable.ArrayBuffer
object ScalaCollection{
    def main(args: Array[String]): Unit = {
        val buffer1 = ArrayBuffer(1,2,3,4)
        val buffer2 = ArrayBuffer(5,6,7,8)

        val buffer3: ArrayBuffer[Int] = buffer1 += 5
        println( buffer1 eq buffer3 ) // true

        // 使用 ++ 运算符会产生新的集合数组
        val buffer4: ArrayBuffer[Int] = buffer1 ++ buffer2
        // 使用 += 运算符会更新之前的集合, 不会产生新的数组
        val buffer5: ArrayBuffer[Int] = buffer1 += buffer2
        println( buffer1 eq buffer4 ) // false
        println( buffer1 eq buffer5 ) // true
    }
}
```

7.2.3 可变数组和不可变数组转换

```
import scala.collection.mutable
import scala.collection.mutable.ArrayBuffer
object ScalaCollection{
    def main(args: Array[String]): Unit = {
        val buffer = ArrayBuffer(1,2,3,4)
        val array = Array(4,5,6,7)

        // 将不可变数组转换为可变数组
        val buffer1: mutable.Buffer[Int] = array.toBuffer
        // 将可变数组转换为不可变数组
        val array1: Array[Int] = buffer.toArray
    }
}
```

7.3 Seq 集合

7.3.1 不可变 List

1) 基本语法

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

        // Seq 集合
        val list = List(1,2,3,4)

        // 增加数据
        val list1: List[Int] = list :+ 1
        println(list1 eq list)
        list1.foreach(println)
        val list2: List[Int] = 1 +: list
        list2.foreach(println)
        println("*****")
        val list3: List[Int] = list.updated(1,5)
        println(list eq list3)
        List3.foreach(println)
    }
}
```

2) 基本操作

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

        // Seq 集合
        val list1 = List(1,2,3,4)
        // 空集合
        val list2: List[Nothing] = List()
        val nil = Nil
        println(list2 eq nil)

        // 创建集合
        val list3: List[Int] = 1::2::3::Nil
        val list4: List[Int] = list1 :: Nil

        // 连接集合
        val list5: List[Int] = List.concat(list3, list4)
        list5.foreach(println)

        // 创建一个指定重复数量的元素列表
        val list6: List[String] = List.fill[String](3) ("a")
        list6.foreach(println)
    }
}
```

7.3.2 可变 List

1) 基本语法

```
import scala.collection.mutable.ListBuffer
object ScalaCollection{
    def main(args: Array[String]): Unit = {
        // 可变集合
        val buffer = new ListBuffer[Int]()
        // 增加数据
        buffer.append(1,2,3,4)
        // 修改数据
        buffer.update(1,3)
        // 删除数据
        buffer.remove(2)
        buffer.remove(2,2)
        // 获取数据
        println(buffer(1))
        // 遍历集合
        buffer.foreach(println)
    }
}
```

2) 基本操作

```
import scala.collection.mutable.ListBuffer
object ScalaCollection{
    def main(args: Array[String]): Unit = {

        // 可变集合
        val buffer1 = ListBuffer(1,2,3,4)
        val buffer2 = ListBuffer(5,6,7,8)

        // 增加数据
        val buffer3: ListBuffer[Int] = buffer1 :+ 5
        val buffer4: ListBuffer[Int] = buffer1 += 5
        val buffer5: ListBuffer[Int] = buffer1 ++ buffer2
        val buffer6: ListBuffer[Int] = buffer1 ++= buffer2

        println( buffer5 eq buffer1 )
    }
}
```

```
println( buffer6 eq buffer1 )

val buffer7: ListBuffer[Int] = buffer1 - 2
val buffer8: ListBuffer[Int] = buffer1 -= 2
println( buffer7 eq buffer1 )
println( buffer8 eq buffer1 )
}
}
```

7.3.3 可变集合和不可变集合转换

```
import scala.collection.mutable
import scala.collection.mutable.ListBuffer
object ScalaCollection{
  def main(args: Array[String]): Unit = {

    val buffer = ListBuffer(1,2,3,4)
    val list = List(5,6,7,8)

    // 可变集合转变为不可变集合
    val list1: List[Int] = buffer.toList
    // 不可变集合转变为可变集合
    val buffer1: mutable.Buffer[Int] = list.toBuffer

  }
}
```

7.4 Set 集合

7.4.1 不可变 Set

1) 基本语法

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

    val set1 = Set(1,2,3,4)
    val set2 = Set(5,6,7,8)

    // 增加数据
    val set3: Set[Int] = set1 + 5 + 6
    val set4: Set[Int] = set1.+(6,7,8)
    println( set1 eq set3 ) // false
    println( set1 eq set4 ) // false
    set4.foreach(println)
    // 删除数据
    val set5: Set[Int] = set1 - 2 - 3
    set5.foreach(println)

    val set6: Set[Int] = set1 ++ set2
    set6.foreach(println)
    println("*****")
    val set7: Set[Int] = set2 ++: set1
    set7.foreach(println)
    println(set6 eq set7)

  }
}
```

2) 基本操作

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

    val set1 = Set(1,2,3,4)
    val set2 = Set(5,6,7,8)
```

```
// 增加数据
val set3: Set[Int] = set1 + 5 + 6
val set4: Set[Int] = set1.+(6,7,8)
println( set1 eq set3 ) // false
println( set1 eq set4 ) // false
set4.foreach(println)
// 删除数据
val set5: Set[Int] = set1 - 2 - 3
set5.foreach(println)

val set6: Set[Int] = set1 ++ set2
set6.foreach(println)
println("*****")
val set7: Set[Int] = set2 ++: set1
set7.foreach(println)
println(set6 eq set7)
}
}
```

7.4.2 可变 Set

1) 基本语法

```
import scala.collection.mutable
object ScalaCollection{
  def main(args: Array[String]): Unit = {

    val set1 = mutable.Set(1,2,3,4)
    val set2 = mutable.Set(5,6,7,8)

    // 增加数据
    set1.add(5)
    // 添加数据
    set1.update(6,true)
    println(set1.mkString(","))
    // 删除数据
    set1.update(3,false)
    println(set1.mkString(","))

    // 删除数据
    set1.remove(2)
    println(set1.mkString(","))

    // 遍历数据
    set1.foreach(println)
  }
}
```

2) 基本操作

```
import scala.collection.mutable
object ScalaCollection{
  def main(args: Array[String]): Unit = {

    val set1 = mutable.Set(1,2,3,4)
    val set2 = mutable.Set(4,5,6,7)

    // 交集
    val set3: mutable.Set[Int] = set1 & set2
    println(set3.mkString(","))
    // 差集
    val set4: mutable.Set[Int] = set1 &~ set2
    println(set4.mkString(","))
  }
}
```



```
}  
}
```

7.5 Map 集合

Map(映射)是一种可迭代的键值对 (key/value) 结构。所有的值都可以通过键来获取。

Map 中的键都是唯一的。

7.5.1 不可变 Map

1) 基本语法

```
object ScalaCollection{  
  def main(args: Array[String]): Unit = {  
  
    val map1 = Map( "a" -> 1, "b" -> 2, "c" -> 3 )  
    val map2 = Map( "d" -> 4, "e" -> 5, "f" -> 6 )  
  
    // 添加数据  
    val map3 = map1 + ("d" -> 4)  
    println(map1 eq map3) // false  
  
    // 删除数据  
    val map4 = map3 - "d"  
    println(map4.mkString(", "))  
  
    val map5: Map[String, Int] = map1 ++ map2  
    println(map5 eq map1)  
    println(map5.mkString(", "))  
  
    val map6: Map[String, Int] = map1 ++: map2  
    println(map6 eq map1)  
    println(map6.mkString(", "))  
  
    // 修改数据  
    val map7: Map[String, Int] = map1.updated("b", 5)  
    println(map7.mkString(", "))  
  
    // 遍历数据  
    map1.foreach(println)  
  }  
}
```

2) 基本操作

```
object ScalaCollection{  
  def main(args: Array[String]): Unit = {  
  
    val map1 = Map( "a" -> 1, "b" -> 2, "c" -> 3 )  
    val map2 = Map( "d" -> 4, "e" -> 5, "f" -> 6 )  
  
    // 创建空集合  
    val empty: Map[String, Int] = Map.empty  
    println(empty)  
    // 获取指定 key 的值  
    val i: Int = map1.apply("c")  
    println(i)  
    println(map1("c"))  
  
    // 获取可能存在的 key 值  
    val maybeInt: Option[Int] = map1.get("c")  
    // 判断 key 值是否存在
```

```
    if ( !maybeInt.isEmpty ) {  
        // 获取值  
        println(maybeInt.get)  
    } else {  
        // 如果不存在，获取默认值  
        println(maybeInt.getOrElse(0))  
    }  
  
    // 获取可能存在的 key 值，如果不存在就使用默认值  
    println(map1.getOrElse("c", 0))  
}  
}
```

7.5.2 可变 Map

1) 基本语法

```
import scala.collection.mutable  
object ScalaCollection{  
    def main(args: Array[String]): Unit = {  
  
        val map1 = mutable.Map( "a" -> 1, "b" -> 2, "c" -> 3 )  
        val map2 = mutable.Map( "d" -> 4, "e" -> 5, "f" -> 6 )  
  
        // 添加数据  
        map1.put("d", 4)  
        val map3: mutable.Map[String, Int] = map1 + ("e" -> 4)  
        println(map1 eq map3)  
        val map4: mutable.Map[String, Int] = map1 += ("e" -> 5)  
        println(map1 eq map4)  
  
        // 修改数据  
        map1.update("e", 8)  
        map1("e") = 8  
  
        // 删除数据  
        map1.remove("e")  
        val map5: mutable.Map[String, Int] = map1 - "e"  
        println(map1 eq map5)  
        val map6: mutable.Map[String, Int] = map1 -= "e"  
        println(map1 eq map6)  
        // 清除集合  
        map1.clear()  
    }  
}
```

2) 基本操作

```
import scala.collection.mutable  
object ScalaCollection{  
    def main(args: Array[String]): Unit = {  
  
        val map1 = mutable.Map( "a" -> 1, "b" -> 2, "c" -> 3 )  
        val map2 = mutable.Map( "d" -> 4, "e" -> 5, "f" -> 6 )  
  
        val set: Set[(String, Int)] = map1.toSet  
        val list: List[(String, Int)] = map1.toList  
        val seq: Seq[(String, Int)] = map1.toSeq  
        val array: Array[(String, Int)] = map1.toArray  
  
        println(set.mkString(", "))  
        println(list.mkString(", "))  
        println(seq.mkString(", "))  
        println(array.mkString(", "))  
    }  
}
```

```
println(map1.get("a"))
println(map1.getOrElse("a", 0))

println(map1.keys)
println(map1.keySet)
println(map1.keysIterator)
println(map1.values)
println(map1.valuesIterator)
}
}
```

7.6 元组

在 Scala 语言中，我们可以将多个无关的数据元素封装为一个整体，这个整体我们称之为：元素组合，简称元组。有时也可将元组看成容纳元素的容器，其中最多只能容纳 22 个

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

    // 创建元组，使用小括号
    val tuple = (1, "zhangsan", 30)

    // 根据顺序号访问元组的数据
    println(tuple._1)
    println(tuple._2)
    println(tuple._3)
    // 迭代器
    val iterator: Iterator[Any] = tuple.productIterator

    // 根据索引访问元素
    tuple.productElement(0)

    // 如果元组的元素只有两个，那么我们称之为对偶元组，也称之为键值对
    val kv: (String, Int) = ("a", 1)
    val kv1: (String, Int) = "a" -> 1
    println( kv eq kv1 )

  }
}
```



思考两个问题:

- 只能放 22 个数据，是不是太少了呢？
- 函数的参数最多能放多少个呢？

7.7 队列

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。

```
import scala.collection.mutable
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    val que = new mutable.Queue[String]()
    // 添加元素
    que.enqueue("a", "b", "c")
    val que1: mutable.Queue[String] = que += "d"
    println(que eq que1)
    // 获取元素
  }
}
```

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



思考一个问题:

- kafka 中如何保证消费数据的有序?

7.8 并行

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。

```
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    val result1 = (0 to 100).map{x => Thread.currentThread.getName}
    val result2 = (0 to 100).par.map{x => Thread.currentThread.getName}

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



思考三个问题:

- 并发 & 并行?
- 什么是线程安全问题? 如何解决?
- ThreadLocal 能解决线程安全问题吗? 共享数据

7.9 常用方法

1) 常用方法

```
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    val list = List(1,2,3,4)

    // 集合长度
    println("size =>" + list.size)
    println("length =>" + list.length)
    // 判断集合是否为空
    println("isEmpty =>" + list.isEmpty)
    // 集合迭代器
    println("iterator =>" + list.iterator)
    // 循环遍历集合
    list.foreach(println)
    // 将集合转换为字符串
    println("mkString =>" + list.mkString(","))
    // 判断集合中是否包含某个元素
    println("contains =>" + list.contains(2))
    // 取集合的前几个元素
    println("take =>" + list.take(2))
    // 取集合的后几个元素
    println("takeRight =>" + list.takeRight(2))
    // 查找元素
    println("find =>" + list.find(x => x % 2 == 0))
  }
}
```

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

```
// 丢弃前几个元素
println("drop =>" + list.drop(2))
// 丢弃后几个元素
println("dropRight =>" + list.dropRight(2))
// 反转集合
println("reverse =>" + list.reverse)
// 去重
println("distinct =>" + list.distinct)
}
}
```

2) 衍生集合

```
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    val list = List(1,2,3,4)
    val list1 = List(1,2,3,4)
    val list2 = List(3,4,5,6)

    // 集合头
    println("head => " + list.head)
    // 集合尾
    println("tail => " + list.tail)
    // 集合尾迭代
    println("tails => " + list.tails)
    // 集合初始值
    println("init => " + list.init)
    // 集合初始值迭代
    println("inits => " + list.inits)
    // 集合最后元素
    println("last => " + list.last)
    // 集合并集
    println("union => " + list.union(list1))
    // 集合交集
    println("intersect => " + list.intersect(list1))
    // 集合差集
    println("diff => " + list.diff(list1))
    // 切分集合
    println("splitAt => " + list.splitAt(2))
    // 滑动（窗口）
    println("sliding => " + list.sliding(2))
    // 滚动（没有重复）
    println("sliding => " + list.sliding(2,2))
    // 拉链
    println("zip => " + list.zip(list1))
    // 数据索引拉链
    println("zipWithIndex => " + list.zipWithIndex)
  }
}
```

3) 计算函数

```
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    val list = List(1,2,3,4)
    val list1 = List(3,4,5,6)

    // 集合最小值
    println("min => " + list.min)
    // 集合最大值
    println("max => " + list.max)
    // 集合求和
    println("sum => " + list.sum)
    // 集合乘积
  }
}
```

```
println("product => " + list.product)
// 集合简化规约
println("reduce => " + list.reduce((x:Int,y:Int)=>{x+y}))
println("reduce => " + list.reduce((x,y)=>{x+y}))
println("reduce => " + list.reduce((x,y)=>x+y))
println("reduce => " + list.reduce(_+_))
// 集合简化规约(左)
println("reduceLeft => " + list.reduceLeft(_+_))
// 集合简化规约(右)
println("reduceRight => " + list.reduceRight(_+_))
// 集合折叠
println("fold => " + list.fold(0)(_+_))
// 集合折叠(左)
println("foldLeft => " + list.foldLeft(0)(_+_))
// 集合折叠(右)
println("foldRight => " + list.foldRight(0)(_+_))
// 集合扫描
println("scan => " + list.scan(0)(_+_))
// 集合扫描(左)
println("scanLeft => " + list.scanLeft(0)(_+_))
// 集合扫描(右)
println("scanRight => " + list.scanRight(0)(_+_))
}
}
```

4) 功能函数

```
object ScalaCollection{
  def main(args: Array[String]): Unit = {
    val list = List(1,2,3,4)

    // 集合映射
    println("map => " + list.map(x=>{x*2}))
    println("map => " + list.map(x=>x*2))
    println("map => " + list.map(_*2))
    // 集合扁平化
    val list1 = List(
      List(1,2),
      List(3,4)
    )
    println("flatten =>" + list1.flatten)
    // 集合扁平映射
    println("flatMap =>" + list1.flatMap(list=>list))
    // 集合过滤数据
    println("filter =>" + list.filter(_%2 == 0))
    // 集合分组数据
    println("groupBy =>" + list.groupBy(_%2))
    // 集合排序
    println("sortBy =>" + list.sortBy(num=>num)(Ordering.Int.reverse))
    println("sortWith =>" + list.sortWith((left, right) => {left < right}))
  }
}
```

7.10 案例实操 - WordCount TopN

7.10.1 数据准备

```
Hello Scala
Hello Spark
Hello Hadoop
```

7.10.2 功能实现

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

    val list: List[String] = Source.fromFile("input/word.txt").getLines().toList

    val wordList: List[String] = list.flatMap(_.split(" "))

    val word2OneList: List[(String, Int)] = wordList.map((_, 1))

    val word2ListMap: Map[String, List[(String, Int)]] = word2OneList.groupBy(_._1)

    val word2CountMap: Map[String, Int] = word2ListMap.map(
      kv => {
        (kv._1, kv._2.size)
      }
    )
    println(word2CountMap)
  }
}
```

7.10.3 小练习 1 - 另外一种 WordCount

```
val dataList = List(
  ("Hello Scala", 4), ("Hello Spark", 2)
)
```

7.10.4 小练习 2 - 不同省份的商品点击排行

数据在资料中：data.txt

第8章 模式匹配

8.1 简介

Scala 中的模式匹配类似于 Java 中的 switch 语法,但是 scala 从语法中补充了更多的功能,可以按照指定的规则对数据或对象进行匹配,所以更加强大。

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

8.2 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。如果不存在 case _ 分支,那么会发生错误。

```
object ScalaMatch{
  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"
    }
    println(result)
  }
}
```

8.3 匹配规则

8.3.1 匹配常量

```
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

```
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  
}
```

8.3.3 匹配数组

```
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

```
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)  
}  
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

```
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)  
}
```

8.3.6 匹配对象

```
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)
    }
}

val user: User = User("zhangsan", 11)
val result = user match {
    case User("zhangsan", 11) => "yes"
    case _ => "no"
}
```

8.3.7 样例类

- 样例类就是使用 **case** 关键字声明的类
- 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**。
- 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。
- 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）

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

object ScalaCaseClass {
    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 应用场景

8.4.1 变量声明

```
object ScalaMatch {
    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) = Person("zhangsan", 16)
        println(s"name=$name,age=$age")
    }
    case class Person(name: String, age: Int)
}
```

8.4.2 循环匹配

```
object ScalaMatch {
    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
        }
```

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(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
}
}
```

8.4.3 函数参数

```
object ScalaMatch {
    def main(args: Array[String]): Unit = {
        val list = List(
            ("a", 1), ("b", 2), ("c", 3)
        )
        val list1 = list.map {
            case (k, v) => {
                (k, v*2)
            }
        }
        println(list1)
    }
}
```

8.5 偏函数

所谓的偏函数，其实就是对集合中符合条件的数据进行处理函数

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 Int，但是我们只考虑数值为 1 的时候，数据该如何处理，其他不考虑。

8.5.1 基本语法

```
// 声明偏函数
val pf: PartialFunction[Int, String] = { case 1 => "one" }

...
// 应用偏函数
println(List(1, 2, 3, 4).collect(pf))
```

8.5.2 案例实操

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。

● 不使用偏函数

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

● 使用偏函数

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


第9章 异常

9.1 简介

Scala 异常语法处理上和 Java 类似，但是又不尽相同。

Java 异常：

```
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");
}
```

9.2 基本语法

```
object ScalaException {
    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")
            }
        } finally {
            println("finally")
        }
    }
}
```

Scala 中的异常不区分所谓的编译时异常和运行时异常，也无需显示抛出方法异常，所以 Scala 中没有 **throws** 关键字。



思考三个问题：

- 什么是编译时异常和运行时异常？
- 分别举几个例子？
- 如果 Java 程序调用 scala 代码，如何明确异常？

增加注解 @throws(Exception)

第10章 隐式转换

10.1 简介

在之前的类型学习中，我们已经学习了自动类型转换，精度小的类型可以自动转换为精度大的类型，这个转换过程无需开发人员参与，由**编译器**自动完成，这个转换操作我们称之为**隐式转换**。

在其他的场合，隐式转换也起到了非常重要的作用。如 Scala 在程序编译错误时，可以通过隐式转换中**类型转换机制**尝试进行**二次编译**，将本身错误无法编译通过的代码通过类型转换后编译通过。慢慢地，这也形成了一种**扩展功能**的转换机制。这个听着很抽象，不好理解，不急，咱慢慢体会。

10.2 隐式函数

```
object ScalaImplicit {
  def main(args: Array[String]): Unit = {
    implicit def transform(d: Double): Int = {
      d.toInt
    }
    var d: Double = 2.0
    val i: Int = d
    println(i)
  }
}
```



思考一个问题:如果有多个相同转换规则怎么办?

10.3 隐式参数 & 隐式变量

```
object ScalaImplicit {
  def main(args: Array[String]): Unit = {
    def transform(implicit d: Double) = {
      d.toInt
    }
    implicit val dd: Double = 2.0
    println(transform)
  }
}
```

10.4 隐式类

在 Scala2.10 后提供了隐式类，可以使用 `implicit` 声明类，隐式类非常强大，同样可以扩展类的功能，在集合的数据处理中，隐式类发挥了重要的作用。

- 其所带的构造参数有且只能有一个
- 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。

```
object ScalaImplicit {
  def main(args: Array[String]): Unit = {
    val emp = new Emp()
  }
}
```

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](#)

```
emp.insertUser()
}
class Emp {
}
implicit class User( emp : Emp) {
  def insertUser(): Unit = {
    println("insert user...")
  }
}
```

10.5 隐式机制

所谓的隐式机制，就是一旦出现编译错误时，编译器会从哪些地方查找对应的隐式转换规则

- 当前代码作用域
- 当前代码上级作用域
- 当前类所在的包对象
- 当前类（对象）的父类（父类）或特质（父特质）

其实最直接的方式就是直接导入。

第11章 泛型

11.1 简介

Scala 的泛型和 Java 中的泛型表达的含义都是一样的，对处理的数据类型进行约束，但是 Scala 提供了更加强大的功能

```
class Test[A] {  
    private var elements: List[A] = Nil  
}
```

11.2 泛型转换

Scala 的泛型可以根据功能进行改变

11.2.1 泛型不可变

```
object ScalaGeneric {  
    def main(args: Array[String]): Unit = {  
  
        val test1 : Test[User] = new Test[User] // OK  
        val test2 : Test[User] = new Test[Parent] // Error  
        val test3 : Test[User] = new Test[SubUser] // Error  
  
    }  
    class Test[T] {  
    }  
    class Parent {  
    }  
    class User extends Parent{  
    }  
    class SubUser extends User {  
    }  
}
```

11.2.2 泛型协变

```
object ScalaGeneric {  
    def main(args: Array[String]): Unit = {  
  
        val test1 : Test[User] = new Test[User] // OK  
        val test2 : Test[User] = new Test[Parent] // Error  
        val test3 : Test[User] = new Test[SubUser] // OK  
  
    }  
    class Test[+T] {  
    }  
    class Parent {  
    }  
    class User extends Parent{  
    }  
    class SubUser extends User {  
    }  
}
```

11.2.3 泛型逆变

```
object ScalaGeneric {  
    def main(args: Array[String]): Unit = {  
  
        val test1 : Test[User] = new Test[User] // OK  
  
    }  
}
```

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


```
val test2 : Test[User] = new Test[Parent] // OK
val test3 : Test[User] = new Test[SubUser] // Error

}
class Test[-T] {
}
class Parent {
}
class User extends Parent{
}
class SubUser extends User {
}
}
```

11.3 泛型边界

Scala 的泛型可以根据功能设定类树的边界

```
object ScalaGeneric {
  def main(args: Array[String]): Unit = {
    val parent : Parent = new Parent()
    val user : User = new User()
    val subuser : SubUser = new SubUser()
    test[User](parent) // Error
    test[User](user)   // OK
    test[User](subuser) // OK
  }
  def test[A]( a : A ): Unit = {
    println(a)
  }
  class Parent {
  }
  class User extends Parent{
  }
  class SubUser extends User {
  }
}
```

11.3.1 泛型上限

```
object ScalaGeneric {
  def main(args: Array[String]): Unit = {
    val parent : Parent = new Parent()
    val user : User = new User()
    val subuser : SubUser = new SubUser()
    test[Parent](parent) // Error
    test[User](user)     // OK
    test[SubUser](subuser) // OK
  }
  def test[A<:User]( a : A ): Unit = {
    println(a)
  }
  class Parent {
  }
  class User extends Parent{
  }
  class SubUser extends User {
  }
}
```

11.3.2 泛型下限

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

```
val parent : Parent = new Parent()
val user : User = new User()
val subuser : SubUser = new SubUser()
test[Parent](parent) // OK
test[User](user) // OK
test[SubUser](subuser) // Error
}
def test[A>:User]( a : A ): Unit = {
  println(a)
}
class Parent {
}
class User extends Parent{
}
class SubUser extends User {
}
```

11.4 上下文限定

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]之后，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。

```
object ScalaGeneric {
  def main(args: Array[String]): Unit = {
    def f[A : Test](a: A) = println(a)
    implicit val test : Test[User] = new Test[User]
    f( new User() )
  }
  class Test[T] {
  }
  class Parent {
  }
  class User extends Parent{
  }
  class SubUser extends User {
  }
}
```

第12章 正则表达式

12.1 简介

正则表达式(regular expression)描述了一种字符串匹配的模式（pattern），可以用来检查一个串是否含有某种子串、将匹配的子串替换或者从某个串中取出符合某个条件的子串等。

表达式	匹配规则
^	匹配输入字符串开始的位置。
\$	匹配输入字符串结尾的位置。
.	匹配除“\r\n”之外的任何单个字符。
[...]	字符集。匹配包含的任一字符。例如，“[abc]”匹配“plain”中的“a”。
[^...]	反向字符集。匹配未包含的任何字符。例如，“[^abc]”匹配“plain”中“p”，“l”，“i”，“n”。
\A	匹配输入字符串开始的位置（无多行支持）
\z	字符串结尾(类似\$，但不受处理多行选项的影响)
\Z	字符串结尾或行尾(不受处理多行选项的影响)
re*	重复零次或更多次
re+	重复一次或更多次
re?	重复零次或一次
re{ n}	重复n次
re{ n,}	
re{ n, m}	重复n到m次
a b	匹配 a 或者 b
(re)	匹配 re,并捕获文本到自动命名的组里
(?: re)	匹配 re,不捕获匹配的文本，也不给此分组合分配组号
(?> re)	贪婪子表达式
\w	匹配字母或数字或下划线或汉字
\W	匹配任意不是字母，数字，下划线，汉字的字符
\s	匹配任意的空白符,相等于 [\t\n\r\f]
\S	匹配任意不是空白符的字符
\d	匹配数字，类似 [0-9]
\D	匹配任意非数字的字符
\G	当前搜索的开头
\n	换行符
\b	通常是单词分界位置，但如果在字符类里使用代表退格
\B	匹配不是单词开头或结束的位置

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

\t	制表符
\Q	开始引号: \Q(a+b)*3\E 可匹配文本 "(a+b)*3"。
\E	结束引号: \Q(a+b)*3\E 可匹配文本 "(a+b)*3"。

12.2 基本语法

```
object ScalaRegex {
  def main(args: Array[String]): Unit = {
    // 构建正则表达式
    val pattern = "Scala".r
    val str = "Scala is Scalable Language"

    // 匹配字符串 - 第一个
    println(pattern findFirstIn str)

    // 匹配字符串 - 所有
    val iterator: Regex.MatchIterator = pattern findAllIn str
    while ( iterator.hasNext ) {
      println(iterator.next())
    }

    println("*****")
    // 匹配规则: 大写, 小写都可
    val pattern1 = new Regex("(S|s)cala")
    val str1 = "Scala is scalable Language"
    println((pattern1 findAllIn str1).mkString(","))
  }
}
```

12.2 案例实操

● 手机号正则表达式验证方法

```
object ScalaRegex {
  def main(args: Array[String]): Unit = {
    // 构建正则表达式
    println(isMobileNumber("18801234567"))
    println(isMobileNumber("11111111111"))
  }
  def isMobileNumber(number: String): Boolean = {
    val regex = "^(13[0-9])|(14[5,7,9])|(15[^4])|(18[0-9])|(17[0,1,3,5,6,7,8]))[0-9]{8}$".r
    val length = number.length
    regex.findFirstMatchIn(number.slice(length-11,length)) != None
  }
}
```

● 提取邮件地址的域名部分

```
object ScalaRegex {
  def main(args: Array[String]): Unit = {
    // 构建正则表达式
    val r = "[_A-Za-z0-9-]+(?:\\.[_A-Za-z0-9-\\+]+)*(@(\\[A-Za-z0-9-]+(?:\\.[_A-Za-z0-9-\\+]+)*|(?\\.[_A-Za-z]{2,})) ?"
    println(r.replaceAll("abc.edf+jianli@gmail.com    hello@gmail.com.cn",
      (m => "*****" + m.group(2))))
  }
}
```

第13章 代码实践

学习了这么长时间的语法，咱们可不要忘记咱们学习这门语言的目的呀，咱们是为了学习后面的分布式计算引擎 Spark 和 Flink 框架。在学习后面那些比较牛的框架之前，咱们就动手自己来写写分布式计算功能。