Scala文档： https://docs.scala-lang.org/zh-cn/

Scala中文手册： https://www.itbook.team/book/scala/SCALAZhongWenShouCe/KaiShiShenQiDeSCALABianChengZhiLv.html

1.数据类型

键盘输入：

```
var line = StdIn.readLine()
println("line = " + line)
```

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语言并不是真正意思的面向对象。

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。

Scala数据类型

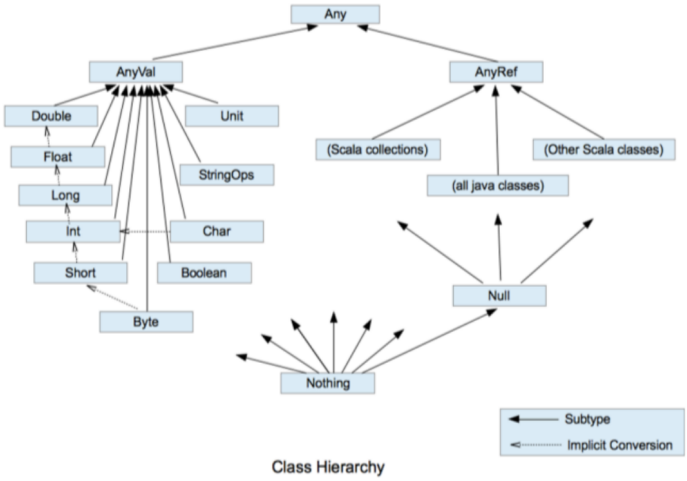
1）Scala中一切数据都是对象，都是Any的子类。

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



Class Hierarchy

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，主要用在一个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。


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

Boolean类型

取值只允许为true和false，占用1个字节

Unit类型，Null类型和Nothing类型

例如：

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

2.运算符

Java 和 Scala 中关于==的区别

Java：

- ==比较两个变量本身的值，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。

Scala：

- ==更加类似于 Java 中的 equals，可以直接用来比较字符串内容是否相同

没有++和--

只能按照如下格式写：

r1 += 1 // 没有++

r1 -= 2 // 没有--

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

3.流程控制

to和until

1 to 3等价于1.to(3)，**前闭后闭**

1 until 3等价于1.until(3)，**前闭后开**

引入变量

（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。

以下3种写法等价：

```scala
for (i <- 1 to 10){
  val j = 10 - i
  println("i = " + i + ", j = " + j)
}

for (i <- 1.to(10); j = 10 - i){
  println("i = " + i + ", j = " + j)
}

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

循环中断

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。

```scala
def test_breakable(): Unit = {
  import scala.util.control.Breaks._
  breakable({
    for (i <- 1 to 10) {
      if (i == 5) {
        break //可加括号，可不加
      }
      println(i)
    }
  }) //breakable这个小括号可以单独省略，花括号也可以单独省略，但是小括号和花括号不能同时省略
}
```

我们不妨看看内部是怎么实现的?

4.函数式编程

- 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。
- 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。

函数和方法的区别

- 为完成某一功能的程序语句的集合，称为函数。**类中的函数**称之**方法**。
- 函数没有重载和重写的概念；**方法**可以进行**重载和重写**。
- Scala 中函数可以嵌套定义。

函数参数

- 可变参数：s : String*
- 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
- 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

```
def tt_cc(s: String*): Unit = {
  println(s)
}
tt_cc("a", "b", "c")
tt_cc("a")

输出:
WrappedArray(a, b, c)
WrappedArray(a)
```

函数至简原则

- 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**
- 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**
- 如果不关心名称，只关心逻辑处理，那么函数名（def）可以省略

```scala
// (1) 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(): Unit = {
  println("atguigu")
}
//两种方式均可
f7()
f7

// (2) 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: Unit = {
  println("atguigu")
}
// f8(): 直接报错
f8

// (3) 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(name: String): Unit = {
  println(name)
}

// 上面的f9，可以用匿名函数[lambda表达式]实现
val chenchi = (name: String) => { println(name) }
chenchi("hello spark")
```

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，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

函数传递其实就两个点：要么加_，要么明确函数类型。

2）函数可以作为参数进行传递

这里可以看到，传递add函数给f1，加不加_均可，因为已经生命类型了。

函数声明：**如果无参是()=>Unit这种，前面的()并不是Unit的对象，因为有参数也得用一个小括号包起来，后面返回的必须是Unit，这是一种类型。中间的=>，其实就是lambda表达式。**

函数声明只需要类型，不需要变量名！！！

3）函数可以作为函数返回值返回

这已经类似于闭包和柯里化了。

示例一：基本使用

```scala
def f(n: Int): Int = {
  println("f调用")
  n + 1
}
def fun(): Int = {
  println("fun调用")
  1
}

val result: Int = f(123)
println(result)

// 1. 函数作为值进行传递
val f1: Int=>Int = f
val f2 = f _

println(f1)
println(f1(12))
println(f2)
println(f2(35))

val f3: ()=>Int = fun
val f4 = fun _
println(f3)
println(f3())
println(f4)
println(f4())
println("====================================")

// 2. 函数作为参数进行传递
// 定义二元计算函数
def dualEval(op: (Int, Int)=>Int, a: Int, b: Int): Int = {
  op(a, b)
}

def add(a: Int, b: Int): Int = {
  a + b
}

```

```scala
println(dualEval(add, 12, 35))
println(dualEval((a, b) => a + b, 12, 35))
println(dualEval(_ + _, 12, 35))
println("========================================")

// 3. 函数作为函数的返回值返回
def f5(): Int=>Unit = {
  def f6(a: Int): Unit = {
    println("f6调用 " + a)
  }
  f6    // 将函数直接返回
}

val f6 = f5()
println(f6)
println(f6(25)) //这里打印()，是因为函数返回值为Unit，Unit取值只有一个()，表示空值

println(f5()(25)) //这里打印()，是因为函数返回值为Unit，Unit取值只有一个()，表示空值
```

结果如下：

示例二：自定义map，filter和reduce

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

def filter(arr: Array[Int], op: Int=>Boolean): Array[Int] = {
  for (elem <- arr if op(elem)) yield elem
}

def reduce(arr: Array[Int], op: (Int, Int)=>Int): Int = {
  if(arr.length == 0) throw new UnsupportedOperationException("数组不能为空")
  var result = arr(0)
  for (i <- 1 until arr.length) {
    result = op(result, arr(i))
  }
  result
}


def main(args: Array[String]): Unit = {
  val arr = Array(1, 2, 3, 4, 5)
  println(map(arr, _ * 2).mkString(","))
  println(filter(arr, _ % 2 == 0).mkString(","))
  println(reduce(arr, _ + _))
}
```

结果如下：

匿名函数（lambda表达式）

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。

匿名函数**不用定义返回值类型**，例如：(x: Int): Int => {xxx}不行，标准写法是def test(x: Int): Int = {}，不要弄混。

匿名函数需要有变量名，和函数声明不同！！！

传递匿名函数至简原则：

- 参数的类型可以省略，会根据形参进行自动的推导

- 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。
- 匿名函数如果只有一行，则大括号也可以省略
- 如果参数只出现一次，则参数省略且后面参数可以用_代替

```scala
//一个参数
//  (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)
```

```scala
println(arr5.mkString(","))


//两个参数
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, _ + _))
```

```scala
// 定义一个函数，以函数作为参数输入
def f(func: String => Unit): Unit = {
  func("atguigu")
}

f((name: String) => {
  println(name)
})

println("========================")

// 匿名函数的简化原则
//      (1) 参数的类型可以省略，会根据形参进行自动的推导
f((name) => {
  println(name)
})

//      (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。
f( name => {
  println(name)
})

//      (3) 匿名函数如果只有一行，则大括号也可以省略
f( name => println(name) )

//      (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
f( println(_) )

//      (5) 如果可以推断出，当前传入的println是一个函数体，而不是调用语句，可以直接省略下划线
f( println )

println("========================")

// 实际示例，定义一个”二元运算“函数，只操作1和2两个数，但是具体运算通过参数传入
def dualFunctionOneAndTwo(fun: (Int, Int)=>Int): Int = {
  fun(1, 2)
}

val add = (a: Int, b: Int) => a + b
```

```scala
val minus = (a: Int, b: Int) => a - b

println(dualFunctionOneAndTwo(add))
println(dualFunctionOneAndTwo(minus))

// 匿名函数简化
println(dualFunctionOneAndTwo((a: Int, b: Int) => a + b))
println(dualFunctionOneAndTwo((a: Int, b: Int) => a - b))

println(dualFunctionOneAndTwo((a, b) => a + b))
println(dualFunctionOneAndTwo( _ + _))
println(dualFunctionOneAndTwo( _ - _))

println(dualFunctionOneAndTwo((a, b) => b - a))
println(dualFunctionOneAndTwo( -_ + _))
```

结果如下：

闭包和函数柯里化（ClosureAndCurrying）

闭包是函数式编程的标配。

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。**（会把内部函数和调用的外部变量一起打包放到堆内存，而不是栈空间，所以不会释放）**

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。**（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）**

```scala
//闭包：初始形态
def f1(a: Int): Int=>Int = {
  def f2(b: Int): Int = {
    a + b
  }
  f2
}
println(f1(1)(2))

//闭包：简化1
def f3(a: Int): Int=>Int = {
  (b: Int) => { //匿名函数
    a + b
  }
}
println(f3(1)(2))

//闭包：简化2
def f4(a: Int): Int=>Int = {
  b => a + b //匿名函数
}
println(f4(1)(2))

//闭包：简化3
def f5(a: Int): Int=>Int = a + _ //匿名函数，变量使用一次
println(f5(1)(2))

//柯里化
def f6(a: Int)(b: Int): Int = {
  a + b
}
println(f6(1)(2))
```

运行结果：

递归（含尾递归）

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。

调用：

```
println(fact(5))  //120
println(tail_fact(5, 1))  //120
println(tailFact(5))  //120
```

控制抽象

如何自己实现一个while循环?

注意：这里的花括号不能省略，是因为有多行代码，之前的breakable那个介意省略，是因为只有一个for循环！

惰性加载

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。

结果如下：

如果去掉lazy，执行结果如下：

5.面向对象

导包说明

Scala 中的三个默认导入分别是

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

类和对象

注意：Scala 中没有 public（默认就是public），一个.scala 中可以写多个类。

注意：Bean 属性（@BeanPropetry），可以自动生成规范的 setXxx/getXxx 方法。

运行结果如下：

封装

Java 封装操作如下：

- 将属性进行私有化
- 提供一个公共的 set 方法，用于对属性赋值
- 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）。

注意：只有成员变量才能初始化为占位符，局部变量不可以！

访问权限

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。

- Scala 中属性和方法的**默认访问权限为 public**，但 Scala 中**无 public 关键字**。
- private 为私有权限，只在**类的内部和伴生对象**中可用。
- 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**。
- **private[包名]**增加包访问权限，包名下的其他类也可以使用。

先定义一个父类和一个子类：

然后去调用该父类和子类，查看访问权限：

运行结果如下：

帮我解释一下scala中的private和private[this]，我看很多源码都还有private[this]，但是听说最新的已经把private[this]这种写法取消了，是真的吗?

在 Scala 中，private 和 private[this] 的主要区别在于**访问控制的粒度**：private[this]修饰的成员只能在**当前实例**中访问，其他实例无法访问。

对于 private[this] 成员，Scala 不会生成 getter和setter 方法，而是直接将字段访问优化为直接访问。这种优化减少了方法调用的开销，性能会略高。

截至 Scala 2.13 和 Scala 3，private[this]**没有被废弃**，仍然是有效的语法，你可能听到的“废弃”指的是社区开发者或某些风格指南建议减少使用private[this]，因为：

- 对于大多数场景，直接使用 private 已经足够。
- Scala 3 引入了更强大的访问控制特性（如 private 的限定修饰符），可以替代某些使用 private[this] 的场景。

构造器：

Scala 类的构造器包括：**主构造器**和**辅助构造器**。

```
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...}
}
```

（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（**代码写在前面**）。

（4）如果主构造器无参数，**小括号可省略【类和函数一样，都是有小括号的，只是可以省略】**，构建对象时调用的构造方法的**小括号也可以省略【和函数类似，函数声明没加小括号，调用一定不能加；但是类声明没加小括号，调用也是可加可不加】**。

运行结果如下：

（5）构造器参数包括三种类型：

- **未用任何修饰符修饰**，这个参数就是一个**局部变量**
- **var 修饰参数**，作为类的**成员属性**使用，可以修改
- **val 修饰参数**，作为类**只读属性**使用，不能修改

继承和多态

scala 是单继承，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。

动态绑定的结果区别：

抽象类

- **定义抽象类：**abstract class Person{} //通过 abstract 关键字标记抽象类
- **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
- **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
- 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
- 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（但最好都加）。

- 子类中调用父类的方法使用 super 关键字

可以创建匿名子类：

运行结果如下：

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，对应的类叫做伴生类，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。

伴生类和伴生对象就是一体化的，private的属性和方法都可以随便访问。

apply方法：

- 通过伴生对象的 apply 方法，实现不使用 new 方法创建对象。
- 如果想让主构造器变成私有的，可以在()之前加上 private。
- apply 方法可以重载。
- 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。
- 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，调用的其实时伴生对象的 apply 方法。

运行结果如下：

我们可以利用单例对象（伴生对象）的特性，来实现Java中的单例模式（设计模式）：

eq 是专门用于比较引用相等性的，无论什么类型，eq 都是直接比较内存地址。

运行结果如下：

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（跟scala中的抽象类一样）**，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。

通过查看字节码，可以看到**特质=抽象类+接口**

在scala2里面，trait没有构造方法，所以后面不能像类一样加()。

使用方式：

- **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...
- 注意：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**。

所有的 Java 接口都可以当做 Scala 特质使用：

```
class Teacher extends PersonTrait with java.io.Serializable
```

动态混入：

创建对象时混入 trait（用的时候再继承），而无需使类混入 trait。

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：

- 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。

我们不妨分析一下执行顺序：

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：

（我把你当兄弟，你却当我爸爸）

结论：

- 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。
- 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**super[Category].describe()**。

特质自身类型：

自身类型可以实现**依赖注入**的功能。那么什么叫做依赖注入呢?

依赖注入 (Dependency Injection, DI) 是一种设计模式，用于将对象所依赖的其他对象（例如服务、组件等）以外部方式传递给它，而不是由对象自己创建或查找依赖项。这种模式有助于解耦组件、提高代码的可测试性和灵活性。

```java
// 服务接口
public interface GreetingService {
    void sayHello();
}

// 服务实现
public class EnglishGreetingService implements GreetingService {
    @Override
    public void sayHello() {
        System.out.println("Hello!");
    }
}

// 使用服务的类
public class Greeter {
    private GreetingService greetingService;

    // 构造函数注入
    public Greeter(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public void greet() {
        greetingService.sayHello();
    }
}

// 主类
public class Main {
    public static void main(String[] args) {
        // 手动注入依赖
        GreetingService service = new EnglishGreetingService();
        Greeter greeter = new Greeter(service); // 注入依赖
        greeter.greet(); // 输出: Hello!
    }
}
```

自身类型是一种机制，允许在特质中指定它期望被混入的具体类型。这种方式确保了特质可以访问指定类型的成员和方法，而不需要显式地继承该类型。

怎么理解**被混入（Mixin）**？

在 Scala 中，“混入”指的是将一个特质 (trait)添加到一个（类或者特质）中，使这个（类或者特质）获得特质中的方法或特性。

例如：

```
class MyApp extends MySQLDatabase with UserService
```

这里，MyApp类混入了MySQLDatabase和UserService，因此MyApp获得了两者提供的功能。

特质可以通过自身类型声明它只能与什么类或者特质一起使用。

例如：

```
trait UserService {
  self: Database => // 表示 UserService 依赖于 Database
  def getUser(id: Int): String = query(s"SELECT * FROM users WHERE id = $id")
}
```

- UserService 的自身类型是 **Database**，意味着它期望UserService只能与实现了Database的类或特质一起使用。
- 这种设计强制了编译时的类型约束。

运行结果如下：

特质和抽象类的区别：

- 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。
- 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（无参构造，这也仅限于scala2及以前版本）。

类型检查和转换

- obj.isInstanceOf[T]：判断 obj 是不是 T 类型。
- obj.asInstanceOf[T]：将 obj 强转成 T 类型。
- classOf 获取对象的类名。

```scala
class Person

object Person extends App {
    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) 获取包名+类名
    val pClass: Class[Person] = classOf[Person]
    println(pClass)
}
```

枚举类

```scala
object Test {
    def main(args: Array[String]): Unit = {
        println(Color.RED)
    }
}
// 枚举类
object Color extends Enumeration {
    val RED = Value(1, "red") //按照1去存，但是调用的时候是red
    val YELLOW = Value(2, "yellow")
    val BLUE = Value(3, "blue")
}
```

type定义新类型

使用 type 关键字可以定义新的数据数据类型名称，本质上就是类型的一个别名。

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

这里我们可以看一下Predef.scala里面关于String的声明：

样例类和普通类：

6.集合

建议：在操作集合的时候，不可变用符号，可变用方法。

我们放在一起进行记忆：

数组、列表、Set、Map都分为**可变**和**不可变**两类。

- 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**。
  - 以不可变以数组为例：指的是数组的长度不可变，数组中的元素的值是可以改变的（数组引用不可变是由val修饰的）
- 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**。

不可变我们可以理解为开辟完空间，就不能变化了。

需要记住的一条准则：**不可变的都是赋值给一个新的集合，可变的都是在原集合上进行操作。**

不可变集合

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。

```scala
//----创建
//数组两种
val arr = new Array[Int](5) //初始化都为0
val arr = Array(1, 2, 3, 4, 5)
//列表两种
//列表不能new，因为被sealed关键字修饰了
val list = 17 :: 28 :: 59 :: 16 :: Nil
val list = List(1, 2, 3, 4, 5)
//Set一种
val set = Set(13, 23, 53, 12, 13, 23, 78)
//Map一种
val map = Map("a" -> 13, "b" -> 25, "hello" -> 3) //map本质里面是元组，所以可以写成Map(("a", 1), ("b", 2), ("c", 3))，但是没必要记

//----遍历查询：都记住两个即可
for (elem <- arr) println(elem) //增强for循环
arr.foreach( println ) //foreach方法

//----增
//数组(分前后)      --以:结尾的运算符，调用的顺序是从右到左
arr :+ 4 //后
4 +: arr //前
//列表(分前后)
list :+ 10 //后
10 +: list //前
//Set(无顺序)
set + 129
//Map(无顺序)
m + ("c" -> 6) //m + (("c", 6))也可以，没必要记

//----删(删不了)

//----改
arr(0) = 12 //数组能改，其他都改不了

//----查
arr(0) //数组下标
list(0) //列表下标
set.contains(0) //查找是否包含
```

```
39 m.contains(0) => m("a") //查找是否包含，并访问(这种不是很优雅，可以用getOrElse，可变不可变都可用getOrElse)
```

可变集合

```scala
//----创建
//数组两种
val arr = new ArrayBuffer[Int]() //包括后面的这些括号都是可加可不加的，别忘了class不管定义加不加括号，new的对象都是可加可不加，这点跟函数不一样
val arr = ArrayBuffer(23, 57, 92)
//列表两种
val list = new ListBuffer[Int]()
val list = ListBuffer(12, 53, 75)
//Set两种
val set = new mutable.HashSet[Int]() / mutable.LinkedHashSet[Int]() / mutable.TreeSet[Int]() / mutable.BitSet()
val set = mutable.Set(13, 23, 53, 12, 13, 23, 78)
//Map两种
val m = new mutable.HashMap[String, Int]() / mutable.LinkedHashMap[String, Int]() / mutable.TreeMap[String, Int]() / mutable.WeakHashMap[String, Int]() / mutable.ListMap[String, Int]()
val m = mutable.Map("a" -> 13, "b" -> 25, ("hello", 3))

//----遍历查询：都记住两个即可
for (elem <- arr) println(elem) //增强for循环
arr.foreach( println ) //foreach方法

//----增
arr/list.append(36)  //appendAll
arr/list.prepend(11, 76) //prependAll
arr/list.insert(1, 13, 59) //insertAll
set.add(10)
m.put("d", 9)

//----删
arr.remove(0) //index，传两个参数多的是删除的长度
list.remove(0) //index，传两个参数多的是删除的长度
set.remove(4) //elem
m.remove("a") //key

//----改
arr(0)=1  arr.update(0, 1)
list(0)=1 list.update(0, 1)
//set没改需求
m.update("a", 5) //等价于put

```

```
//----查
arr(0) //数组下标
list(0) //列表下标
set.contains(0) //查找是否包含
m.contains(0) => m("a") //查找是否包含，并访问(这种不是很优雅，可以用getOrElse，可变不可变都可用getOrElse)
```

可变不可变互相转换

toArray toBuffer

toList toBuffer

toSet

toMap

++运算符

++操作两边的集合均不发生变化，会生成一个新集合。

++可以接受任意集合类型，不管什么类型，不管可变可不变

元组

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，元组中最大只能有 22 个元素。

关于集合操作

衍生集合

简单函数

复杂函数

flatMap 相当于先进行 map 操作，在进行 flatten 操作。

WordCount实现

这里面groupBy里面为什么不能直接传_?

对于groupBy函数来说，**groupBy(_)等价于groupBy _，相当于函数赋值**，会产生歧义。

注：mapValues那一步用map(kv => (kv._1, kv._2.length))也可以实现。

mapValues的作用：mapValues creates a Map with the same keys from this Map but transforming each key's value using the function f.

队列和栈

7.模式匹配

基本语法

```
val y: String = 1 match {
  case 1 => "one"
  case 2 => "two"
  case 3 => "three"
  case _ => "other"
}
```

（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。

模式守卫

匹配类型

匹配变量声明

匹配对象（样例类）

关于匹配对象说明：

- 当将Student("alice", 18)写在 case 后时，会默认调用 **unapply 方法(对象提取器)**，**student 作为 unapply 方法的参数**，unapply 方法将 student 对象的 name 和 age 属性提取出来，与 Student("alice", 18)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]] 。

关于样例类说明：

- 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）。

偏函数

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。

该偏函数的功能是返回输入的 List 集合的第二个元素。
上述代码会被编译器编译成如下代码：

```
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)
    }
}
```

8.异常处理

先看下Java中的异常：

```java
public static void main(String[] args) {
    try {
        int n = 10 / 0;
    } 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");
    }
}
```

Scala中的异常：

```scala
def main(args: Array[String]): Unit = {
  try{
    val n = 10 / 0
  } catch {
    case e: ArithmeticException => {
      println("发生算术异常")
    }
    case e: Exception => {
      println("发生一般异常")
    }
  } finally {
    println("处理结束")
  }
}
```

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**。

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")
}
```

9.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。

隐式函数

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。

说明：

- 同一个作用域中，相同类型的隐式值只能有一个
- 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
- 隐式参数优先于默认参数

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）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。

隐式解析规则

（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。

例如：

```
spark.read.text("afs://yinglong.afs.baidu.com:9902/user/feed-bjh/output/bjh_pay_chapter_order_data/20230320").map(row => {
  val splits = row.getString(0).split("\t")
  val article_nid = splits(0)
  val set_nid = splits(1)
  val single_order_num = splits(8).toLong
  val set_order_num = splits(9).toLong
  val is_explosive = splits(10).toInt

  (article_nid, set_nid, single_order_num, set_order_num, is_explosive)
}).toDF("article_nid", "set_nid", "single_order_num", "set_order_num", "is_explosive").createOrReplaceTempView("result")
```

前面必须加上**import spark.implicits._**才不会报错。

10.泛型

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”。

泛型上下限

Class PersonList[T <: Person] //泛型上限

Class PersonList[T >: Person] //泛型下限

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]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。

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)
```