✓ 为什么scala

scala不是一门纯粹的面向对象的语言，他强调函数式编程，函数式编程的两大核心理念是函数是一等的，以及程序中应该将值直接映射到输出，尽量的减少在函数中对程序的副作用。

- 一等函数

  函数可以做为别的函数的参数、返回值，赋值给变量

- 直接映射到输出

  操作直接将输入值映射到输出值，而不是当场修改数据。

「变量定义」：变量分为两种，一周是val类型，一种是var类型，val类型一旦被初始化就不能修改其值，var类型在整个生命周期之内都可以被重新赋值。

「函数定义」：def开始，接下来是函数名，之后吃参数列表，紧跟一个冒号，之后是返回类型，接下来是等号，花括号之内是函数体

```
def max(x: int, y:Int): Int = {
 if(x > y)
   x
 else
   y
}
```

「函数字面量」：用圆括号括起来一组带名字的参数，一个又箭头和函数体

```
(x: Int, y: Int) => x + y
```

「foreach,跟for遍历」

```
args.foreach(arg => println(arg))

for(arg <- args)
  println(arg)

for(i <- 0 to 2)
  println(i)
//for括号中，左边是变量，右边是数组
```

方法调用：在scala中并没有操作符重载，1 + 2被解释成(1).+(2)调用Int的+方法

「从文件读取文本行」：

```
import scala.io.Source
if(args.length > 0){
  for(line <- Source.fromFile(args(0)).getLines())
    println(line.length + " " + line)
}
else
  Console.err.println("please enter filename")
```

✓ 类和对象

「类的定义」

```
class ClassName(){
} //类的A定义

val a = new ClassName() //实例化

a.method //调用字段方法
```

若是定义的返回值位uint，其作用就是强调函数的副作用，要是没有 `return` 语句，将会将函数中最后的结果做为返回值

「单例对象」

当有同名类时，单例对象位伴生对象，可以访问 `伴生类` 的所有成员变量，其可以存放静态方法，类中不能有静态方法

```
object ObjectName{
}
```

若无 `伴生类` 则位孤立对象，其中可以用maim方法

```
object Summer{
  def main(args: Array[String]) = {
    for(arg <- args)
      println("hello world")
  }
}

```

直接使用scala命令进行编译，时间会比较长，因为他会遍历jar所有的包，使用 `fsc` 进行编译，第一次时间比较长，之后会创建一个 `守护进程` 若是程序没有改动，则时间上会很快

✓ 基础类型跟操作

- 「字面量」：是表达源码中一个固定值的方法

- 「字符串字面量」：这里需要注意的是若是有 `"""` 括起来的字符串将不会进行转译
- 「符号字面量」： `'ident` 这样的形势，用来再动态语言中当作标识符的场合
- 「字符串插值」： `s" $name"` 将回去寻找变量name，对字符串中的 `name` 进行插值，还可以使用f插值如 `f"{math.Pi}%.5f"` 进行printf风格的插值，row插值将不会对 `\` 进行转译
- 「操作符即方法」

  调用1 + 3 这样的方法，会调用 `1.+(3)` ，同时，任何方法都是操作符， `s.indexOf('o')` 可以写成 `s indexOf 'o'`

  前缀标识符：+ - ! ~ 会表示成 `p.unary_!` 这样的类似形式g

  后缀标识符是那些不接受参数且再调用时没有英文句点圆括号的方法，可以再调用的时候省区圆括号，如 `s.length` 返回字符串 `s` 的长度
- 「无符号右移」： `>>>` 左移跟无符号右移都会自动填充0
- 「对象相等」：直接使用 `==` 的方法，可以直接比较对象的内容上相等。
- 「结合性」：任何 `:` 的方法都由他右边的对象调用，传入左侧的操作元。 `a:::b` 将会是 `b.:::(a)`

✓ 函数式对象

如果一个类没有定义体，并不需要给出花括号，因为没有显示的构造方法，若需要检查一些前置条件的话，需要用到 `require` 这个方法

```
class Rational(n: Int, d: Int){
  require(d != 0)  //检查分母不能为0
  override def toString = n + "/" + d //没有参数的话可以直接写 =
}
```

`n, d` 做为类参数，scala编译器会自动采集到这两个参数，并创建一个主构造方法，同时接受这两个参数。

重写 `toString` 方法：因为编写一个类之后，新建他会打印初该类的一些默认信息，要是没有从写 `toString` 方法的话会导致打印的只是地址

```
class Rational(n: Int, d: Int){
  override def toString = n + "/" + d
}
```

「辅助构造方法」：相当于多态，以 `def this` 打头，但是辅助构造放法必须调用 `主构造方法` 或者调用同一个类中的 `另外一个构造方法`，但这个构造方法最终还是会调用主构造方法，主构造方法是类的单一入口，因为只有主构造方法才能调用 `超类`

```
class Rational(n: Int, d: Int){
  def this(n: Int) = this(n, 1)
}
```

「变量定义」：先申明为 `val或var` 之后定义标识符，最后定义类型，若是val需要给出初始化参数（正常情况下），但是要是标识符以下划线结尾，需要再标识符与冒号之间加上 `空格`

```
val n: Int = 1
val n_ : Int = 1
```

「隐式转换」：重载了操作符之后就可以进行运算了，比如 r * 2这样的方法没问题，操作符的左结核性会调用r对应的 * 方法，但是使用2 * r 这样的方法并不行，因为Int类型并没有定义 * r的方法，若想调用，需要加入隐式转换

```
implicit def intToRational(x: Int) = new Rational(x)
```

该方法将int转换为Rational类型，之后在调用Rational的 * 方法

Rational程序例子：

```
class Rational(d: Int, n: Int){
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)
  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
      )

  def + (i: Int):Rational =
    new Rational(numer + i * denom, denom)

  def - (that: Rational): Rational =
    new Rational(numer * that.denom - that.numer * denom,
      denom * that.denom)

  def - (i: Int): Rational =
    new Rational(numer - i * denom, denom)

```

```
  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  def * (i: Int): Rational =
    new Rational(numer * i, denom)

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)

  override def toString = numer + "/" + denom
  private def gcd(a: Int, b: Int): Int =
    if(b == 0) a else gcd(b, a % b)

}
```

✓ 内建的控制结构

- **「if表达式」**

  首先测试条件，根据条件是否满足来执行两个不同代码分支中的一个

  ```
  println(if (!args.isEnpty) args(0) else "default.txt")
  ```

- **「for循环」**

  ```
  for (file <- filesName)
    println(file)
  ```

  像 `file <- filesName` 这样的语法是生成器，遍历filsname中的所有元素

- **「过滤」**

  ```
  for (file <- filesName if file.getName.endWith(".scala"))
    println(file)
  ```

  使用if子句进行过滤

- **「嵌套循环」**

  使用多个 `<-` 子句，将会先遍历外层迭代器，在根据条件遍历内层迭代器，还可以在中途绑定变量。

  ```
  for (
    file <- filesName
    if file.getName.endWith(".scala")
    line <- fileLines(file)
      trimed = line.trim
    if trimed.matchs(pattern)
  )println(file + ":" line.trim)
  ```

- **「产出一个新的集合」**

  在for循环的后面加上`yield`关键字将会产生一个集合，如`for`子句`yield`代码体

```
def scalaFiles =
  for{
       file <- filesName
       if file.getName.endsWith(".scala")
    }yield file
```

- **「捕获处理异常」**

  try：代码体

  cache：捕获处理异常，且里面用cace进行匹配

  finally：不管咋样都要执行的代码

- **「没有break跟continue」**

  通过使用变量`var`来进行控制，要是实在是需要break支持，可以在`scala.util.control.Braks._`中找到

✓ 函数跟闭包

- **「函数字面量」**

  函数字面量存在于源码中，在运行时编译成类，并在运行时实例化为函数值，所以函数值对象形式存在于运行时

```
(x: Int) => x + 1
```

  上述例子时函数字面量，左侧时参数列表，右侧是操作，多条语句的话，使用`{}`进行组合，`=>`的意思表示将左侧的内容转换为右侧的内容。

```
someNumbers.filter(x => x > 0)
```

  接受一个函数做为入参，并对列表中的每个元素调用这个函数。同时由于知道了`someNumbers`是整数，所以x必定是整数，不用再申明类型

- **「占位符语法」**

用占位符来表示一个或多个参数，只要满足每个参数只在函数字面量中出现一次就行，第一个下划线代表第一个参数，第二个下划线代表第二个参数，第三个下划线代表第三个参数，以此类推。

```
someNumber.filter(_ > 0)  //变量是啥都不管
```

- **「部分应用函数」**

  可以用`_`代替独立的参数列表，但是要与主函数之间加上空格。

```
def sum(a: Int, b: Int, c: Int): Int = a + b + c
val a = sum _  //_ 表示的是三个参数的参数列表 可以通过a(1, 2, 3)调用
val b = sum(1, _: Int, 3)  //可以调用b(2)
someNumbers.foreach(println _)
someNumbers.foreach(println)
```

  最后的这两种形式，可以再明确需要函数的地方给出。

- **「闭包」**

  函数字面量创建出来的函数值被称为闭包，该名称源于“捕获”其自由变量从而“闭合”该函数字面量的操作。没有自由变量的字面量成为闭合语，比如`(x: Int => x + 1)`。任何带有自由变量的函数字面量称为开放语，比如`(x: Int => x + more)`

- **「特殊函数的调用形式」**

  - 重复参数

```
def echo(args: String*) =
  for (arg <- args) println(arg)
echo(arr: _*)  //传入一个String类型的数组
```

  - 缺省值参数，带名参数

    缺省值参数就是将参数提前赋值，与python中一致。

    带名参数的作用就是再调用过程中，参数的顺序可以被打乱。

- **「尾递归」**

  在函数的尾部自己调用自己，这样的话就是尾递归，尾递归会自动的优化成`while`循环的形式，减少栈的开销。但是只能对那些直接尾递归的函数进行优化，要是间接的调用尾递归并不会进行优化

```scala
def isEven(x: Int): Boolean =
  if (x == 0) true else isOdd(x - 1)

def isodd(x: Int): Boolean =
  if (x == 0) false else isEven(x - 1)  //这样的相互递归不会优化，最后一步调用的是另外的一个函数值，并不会进行优化
```

✓ 抽象控制

- **「减少代码重复」**

  利用一等函数与占位符消除代码的重复

```scala
object FileMatcher{
  private def filesHere = (new java.io.File(".")).listFiles
  private def filesMatching(matcher: String => Boolean) =
    for (file <- filesHere; if matcher(file.getName))
      yield file

  def filesEnding(query: String) =
    filesMatching(_.endsWith(query))

  def filesContaining(query: String) =
    filesMatching(_.contains(query))

  def filesRegex(query: String) =
    filesMatching(_.matches(query))
}

```

- **「柯里化」**

  支持多个参数列表编写的一种形式，可以通过传入多组参数来应用他，也可以部分应用，然后绑定到变量上，通过变量进行调用

```scala
def curriedSum(x: Int)(y: Int) = x + y
val second = curriedSum(1)_
val twoPlis = second(2)  //1 + 2 的值，通过部分绑定调用
```

- **「编写新的控制结构」**

  通过一等函数，与 `柯里化` ，将某个控制模式简化。

```
def withPrintWriter(flie: File)(op: PrintWriter => Unit){
  val writer = new printWriter(file)
  try{
    op(writer)
  } finally{
  writer.close()
  }
}

val file = new File("data.txt")
withPrintWriter(file){wirter => writer.println(new java.util.Data)} //单个参数可以使用花括号
```

- **「传名参数 - 传值参数」**

  传值参数中表达式的值将先于函数被调用，而传名参数将会在函数中调用该名字的地方进行调用。其实就是一个参数求值先后的问题。

```
def byNameAssert(predicate: => Boolean) = {
  if (assertionsEnabled && !predicate) //传名参数，运行到这里才会进行求值
    throw new AssertionError
}

def boolAssert(predicate: Boolean) = //传值参数，其实在这里predicate就进行求值了
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

✓ 组合跟继承

- **「抽象成员」**

  申明为一个没有实现方法的成员，包含其的类为抽象类，具体需要在 `class` 之前加上 `abstract`

- **「无参方法」**

  当函数没有参数列表的时候，参数列表的 `()` 都可以省略不写，但是在当其不仅仅表示一个属性，而是将会对某些 `var` 产生副作用的时候，将建议写上圆括号

```
abstract class Element{
  def contents: Array[String] //抽象方法，因此需要抽象类
  def height: Int = contents.length //参数列表省略，调用的时候也可以不使用参数列表
  def width: Int = if (height == 0) 0 else contents(0).height
}
```

- **「扩展类」**

  通过 `extends` 关键字进行类继承，但是只能继承私非私有成员，不能继承私有成员。在重写 `overide` 的过程中，字段跟方法不能同名。

- **「定义参数化字段」**

  在类参数前加上 `val` 则同时定义参数跟字段。

```
class ArrayElement(
  val contents: Array[String]
)extends Element

class ArrayElement(cons: Array[String]) extends Element{  //相当于类定义的这个形式
  val contents: Array[String] = cons
}
```

- **「调用超类构造方法」**

  只需要将参数放入扩展类后面的圆括号中即可

```
class ArrayElement(s: String) extends ArrayElement(Array(s))
```

- **「多态跟动态绑定」**

  多态就是多种形态，基类可以指向不同的子类来进行实例化，同时要是有不同的子类中有同名函数，将会根据实例化的子类来决定具体调用哪个函数，这样的方法叫做动态绑定。这两个就是一个概念，字面上理解就好，现实应用中也特别的自然。

- **「申明final成员」**

  要是不想某个方法或者字段被重写，可以通过在成员前面添加 `final` 字段来达到这样的效果，这样子类中可以继承使用，但是不能被重写。

- **「定义工厂对象」**

  工厂对象包含创建其他对象的方法。工厂方法使用单例对象来定义创建其他类，最直接的放在父类中，并通过引入来进行从命名。

✓ 特质

- **「简述」**

  使用关键字 `trait` 进行特质的使用，同时可以使用 `extends` 跟 `with` 将特质混入到类中，其中使用 `extends` 进行特质的混入会隐式的继承特质中的超类，但是在特质中，不能有任何的类参数（传入类的主构造方法的参数），且与 `super` 是进行的动态绑定

```scala
class Frog extends Animal with Philosophical with HasLegs  //通过extends继承某个类，同时可以在用with引入多个特质
trait Philosophical  //特质申明样子

trait Nopoint(x: Int, y: Int)  //错误，不能进行编译
```

- **「Ordered特质」**

  是一个进行大小比较的特质，使用需要进行两步走，第一是混入 `Ordered[C]`，其中C是你要进行比较的元素类。第二件事是定义一个用来比较两个对象的 `compare` 方法，该方法比较两个接受者，要是返回 `0` 则两个对象相等，要是返回负值则接收者比入参小，要是为正值，则接受者比入参大。

```scala
class Rational(n: Int, d: Int) extends Ordered[Rational]{
  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)
}
```

- **「作为可叠加修改的特质」**

  就是特质可以为类提供可叠加的修改，及特质修改类的方法，而且允许你将你这些修改叠加起来。且叠加的优先级重要程度是越靠近右边的方法最先被调用，如果方法调用 `super` 则最左侧的方法最先被调用。

```scala
abstract class IntQueue{
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue{
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) = { buf += x}
}

trait Doubling extends IntQueue{
  abstract override def put(x: Int) = {super.put(2 * x)}  //对于可叠加的特质，特地申明为abstract override
}

trait Incrementing extends IntQueue{
  abstract override def put(x: Int) = {super.put(x + 1)}
}

trait Filtering extends IntQueue{
  abstract override def put (x: Int) = {
    if(x >= 0) super.put(x)
  }
}

val queue = (new BasicIntQueue with Filtering with Incrementing)  //先调increasing 在调filter
```

如果是在类中调用了super方法，将会调用最近的一个方法，如果除了最后一个方法之外，所有方法都super，那最终结果就是叠加到一个的行为（叠加）

✓ 包和引入

- **「打包」**

  使用 `package` 关键字进行打包

```
package bobsrockets{
  package navigation{

  }
}
```

  顶层包通过 `__root__` 进行引入

- **「引入」**

  通过 `import` 进行相关包以及函数的引用，同时还可以使用重命名的方法，通过 `<原名> => <新名>` 这样的方法进行重命名

```
import Fruits.{Apple => Ap, Orange}
import Fruits.{Pear => _, _}  //引入除了Pear的所有包
```

- **「包对象」**

  每个包都会有一个对象，任何被放在包对象里的定义都会被当做这个包本身成员

```
package object bobsdelights{
  def showFruit(fruit: Fruit) = {
    import furit._
    println(name + "s are " + colro)
  }
}

package printmenu
import bobsdelights.showFruit //直接当做成员来使用
```

✓ 断言跟测试

- **「`assert` 断言」**

如果 `condition` 不满足，则 `assert(condition)` 抛出 `AssertionError` 。同时还存在另外一个版本， `assert(condition, explanation)` 首先检查条件，要是条件不满足，则给定 `explanation` 的 `AssertionError`

- **「`scala` 测试」**

`ScalaTest` 的核心概念是套件 `suite` ，即测试集合。所谓测试，可以是任何带有名称，可以启动，并且要么成功，要么成功，要么失败。

`ScalaTest` 提供了风格特质 `(style trait)` ，这些特质扩展了 `suite` 并提供了不同的测试风格。

```
import org.scalatest.FunSuite
import Element.elem

class ElementSuite extends FunSuite{
  test("elem result shuld have passed width"){  //圆括号内表示的是测试名称
    val ele = elem('x', 2, 3)
    assert(ele.width == 2)  //测试内容
  }
}
```

还可以强调预期结果与实际结果的差

```
assertResult(2){
  ele.width
}
```

如果想检查摸个方法抛出某个预期的异常可以用 `assertThrows`

```
assertThrowe[IllegalArgumentException]{
  elem('x', -2, 3)
}

val caught =
  intercept[ArithmeticException]{
    1 / 0
  }
assert(caught.getMessage == "/ by zero")
```

如何执行一个测试

```
scala -cp scalatest.jar TVsetSpec.scala  //编译
scala -cp scakatest.jar org.scalatest.run TVsetSpec.scala  //执行
```

✓ 样例类跟模式匹配

- **「样例类」**

笼统来说，就是将想要匹配的类加上一个 `case` 关键字。

首先他会将我们的方法加上一个工厂方法，可以通过 `Var(“x”)` 这样的类型来进行类创建而不是 `new Var(“x”)` 。

其次是参数列表中的参数都可以隐式的加上一个 `val` 前缀，这样的话就可以当做字段来进行处理。

再次编译器会帮我们自然的实现 `toString, hashCode, equals` 方法

最后编译器还会帮我们做一个 `copy` 方法，该方法可以进行一些参数的修改

```
abstract class Expr
case class Var(name: String) extends Expr
case class number(num: Double) extends Expr
case class UpOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr

//使用实例
val v = Var("x")
val op = BinOp("+", number(1), v)
v.name
println(op)
op.copy(operator = "-")
```

- **「模式匹配」**

具体形式是 `选择器 match {可选分支}` ，其中可选分支包括至少一个 `case` ，每一个可选分支都包括一个模式以及一个或多个表达式，如果模式匹配了，这些表达式就会被求值。 `=>` 将模式与表达式分开。

`match switch` 的区别

- `match` 是一个表达式，总是会得到一个值
- `scala` 的分支不会贯穿到下一个 `case` ，若是匹配的话，会隐式的加上一个 `break` 进行退出
- 若是啥都不匹配的话会抛出异常，所以最后需要加上一个通配 `case`

```
def simplifyTop(expr: Expr): Expr = expr match{
  case UnOp("-", UnOp("-", e)) => e  //双重取负
  case BinOp("+", e, number(0)) => e  // + 0
  case BinOp("*", e, number(1)) => 1  // * 1
  case _ => exrp
}
```

- **「模式种类」**

所有的模式匹配跟表达式类型十分的相似

- **「通配模式」**

  `(_)` 会匹配任何对象，用来忽略你并不关心的局部

```
expr match{
  case BinOp(_, _, _) => println(expr + "is a binary operation") //只关心是一个二源操作
  case _ => println("something else") //覆盖值处理
}
```

- **「常量模式」**

  只匹配自己

```
expr match{
  case 5 => "five"
  case Nil => "the empty list" //只能匹配空列表
  case _ => "something else"
}
```

- **「变量模式」**

  变量模式可以匹配任何对象，这点跟通配模式相同，但是变量模式会将匹配的对象绑定到变量上，这样就可以使用这个变量进行下一步的处理

```
expr match{
  case 0 => "zero"
  case somethingElse => "not zero" + somethingElse //这里将somethingElse用作一些处理，但是这还是局部变量
}
```

  `scala` 中一般会将**「小写」**开头的字面量当做**「变量匹配」**，要是想要继续使用小写开头字面量当做常量的话可以使用 `this.pi, obj.pi` 等进行处理

- **「构造方法模式」**

  最有用的模式，由一个名称加上一个圆括号以及圆括号中的模式组成。这个模式将首先检查被匹配的对象是否是以这个名称命名的样例实例，在检查这个对象的够着方法参数是否匹配这些额外给出模式。

  这样意味着可以进行深度匹配，不仅仅会检查顶层，还会进一步检查对象内容是否匹配这些额外的模式要求

```
expr match{
  case BinOp("+", e, number(0)) => println("a deep match") //首先匹配BinOp，接着匹配参数列表，最后匹配number(0)，一共有三层模式匹配
}
```

- **「序列模式」**

  可以跟`Lsis, Array`这些序列进行模式匹配，可以用`_*`作为模式的最后一个元素，表示的是还可以匹配任意长度的元素。

```
expr match{
  case List(0, _*) => println("this is a list with beginning 0")
  case _ =>
}
```

- **「元组模式」**

  这里跟序列模式是一样的，只不过将`List`换成了圆括号

- **「带类型的模式」**

  使用该模式来替代类型测试跟类型转换

```
expr match{
  case s: String => s.length
  case m:Map[_, _] => m.size
  case _ => -1
}

//不好的类型转换跟测试
if(x.isInstanceOf[String]){  //类型判断
  val s = x.asInstanceOf[String]  //类型转换
  s.length
}
```

  - 擦除类型

    无法检查`Map中Int到Int`的映射，因为采用的都是一个叫做擦除式泛型，但是数组是例外

```
expr match{
  case m: Map[Int, Int] => true  //error Map不支持具体类型检查
  case a: Array[String] => "yes"  //true 对Array做了特殊处理
  case _ =>
}
```

  - 变量绑定

    除了独自存在的变量模式之外，我们还可以对任何其他模式添加变量，之后在对变量做一些其他的处理，通过`@`进行比那辆的绑定

```
expr match{
  case UpOp("abs", e @ UpOp("abs", _)) => e  //变量绑定
  case _ =>
}
```

- **「模式守卫」**

  模式守卫是任何的布尔表达式，如果纯在模式守卫，这个匹配需要在模式守卫为 `true` 的时候才会成功

```
expr match{
  case m: Int if n> 0 => println(m)  //打印大于 0 的数
  case _ =>
}
```

- **「密封类」**

  在做模式匹配的过程中，如果想要编译器帮你确保已经列出所有的类了，通常将会申明为密封类 `sealed` ，密封类中所有的子类都需要在密封类所在的文件当中，这样我们只需要关心我们已知的子类就可以了

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class number(num: Double) extends Expr
case class UpOp(operation: String, arg: Expr) extends Expr
case class BinOp(operation: String, left: Expr, right: Expr) extends Expr
```

- **「`Option` 类型」**

  `Option` ，为一个标准的类型表示可选值，这样的值可以有两种形式 `Some(x), None` ，主要是传递参数的时候进行使用，因为有些时候必须需要判断一个对象是否为空，但是在 `scala` 中 `Null` 并不是一个合法元素，因此不能进行判断，使用 `Option` 类型很好的解决了这个问题。

```
def show(x: Option[String]) = x match{
  case Some(x) => s
  case None => "?"  //Option类型可以有Null
}
```

- **「偏函数」**

  它只对会作用于指定类型的参数或指定范围值的参数实施计算，超出它的界定范围之外的参数类型和值它会忽略

  `PartialFunction` 特质规定了两个要实现的方法： `apply` 和 `isDefinedAt` ， `isDefinedAt` 用来告知调用方这个偏函数接受参数的范围，可以是类型也可以是值，如果是指定的范围，则返回 `true` 否者的话返回 `false` 。 `apply` 方法用来描述对已接受的值如何处理。

  大多数的时候服务的是case，case序列得到的其实就是一个偏函数

  自己定义一个偏函数

```scala
new PartialFunction[List[Int], Int]{
  def apply(xs: List[Int]) = xs match{
    case x :: y :: _ => y
  }
  def isDefinedAt(xs: List[Int]) xs match{
    case x :: y :: _ => true
    case _ => false
  }
}
```

✓ 使用列表