# 面向对象

Scala的类与Java、C++的类比起来更简洁，学完之后你会更爱Scala！！！

对象： 使用object关键字修饰的

类： 使用class关键字修饰的 new Person() 实例对象

new 类： 类的实例（对象）

## 对象

### 单例对象

Scala中没有静态方法和静态字段，没有static，

java中，没有关键字修饰的方法，只能用new class（）.方法

so 对于一个class来说，所有的方法和成员变量在实例被 new 出来之前都无法访问

虽然可以在class中定义main方法，然并卵…

但是可以使用object这个语法结构来达到同样的目的

用object关键字修饰的对象是单例的，称为单例对象，静态对象。

|  |
| --- |
| *//单例对象* **object** ScalaSingleton {  **def** saySomething(msg: String) = {  *println*(msg)  } } **object** test {  **def** main(args: Array[String]): Unit = {  ScalaSingleton.*saySomething*(**"singleton...."**)  *println*(ScalaSingleton)  *println*(ScalaSingleton)  *// 输出结果:  // cn.demo.ScalaSingleton$@28f67ac7  // cn.demo.ScalaSingleton$@28f67ac7* } } |

### 伴生对象

伴生对象是一种特殊的单例对象。是一种相对概念，需要满足两个条件：

条件1：在同一个源文件中,

条件2：对象名和类名相同

这样的单例对象，被称作是这个类的伴生对象。类被称为是这个单例对象的伴生类。

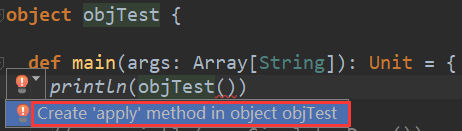
结论：**类和伴生对象之间可以相互访问私有的方法和属性**

|  |
| --- |
| **class** Dog {  **val** *id* = 1  **private var** *name* = **"xiaoqing"   def** printName(): Unit ={  *//在Dog类中可以访问伴生对象Dog的私有属性  println*(Dog.*CONSTANT* + *name* )  } }  */\*\*  \* 伴生对象  \*/* **object** Dog {   *//伴生对象中的私有属性* **private val** *CONSTANT* = **"汪汪汪 : "   def** main(args: Array[String]) {  **val** p = **new** Dog  *//访问私有的字段name* p.*name* = **"123"** p.printName()  } } |

### apply方法

通常我们会在类的伴生对象中定义apply方法，当遇到对象名(参数1,...参数n)时apply方法会被调用

正常情况下，对象调用时是不能带参数的，但是如果能找到对应的apply方法，就能调用成功。



当使用对象（参数列表）来调用对象时，会去对象中找对应参数的apply方法，如果找到就执行相应的逻辑，如果找不到，就报错。

注意：只能找到和参数列表对应的apply方法。

要和对象区分开来

ApplyDemo // 对象

ApplyDemo() // ApplyDemo.apply() 方法

该语法的目的：**不需通过new关键字，更方便的完成类和实例对象的初始化。**

|  |
| --- |
| **object** ApplyDemo {  def apply(msg:String) = { print(s"主食 油泼面，小菜：$msg")  }  def apply(i:Int):Int = { i \* i  }  **def** main(args: Array[String]) {  *//调用了Array伴生对象的apply方法  //def apply(x: Int, xs: Int\*): Array[Int]  //arr1中只有一个元素5* **val** arr1 = *Array[Int]*(5)  *println*(arr1.toBuffer)  println(ApplyDemo("海参炒面"))  println(ApplyDemo.apply("油炸煎饼"))  println(ApplyDemo(1))  } } |

apply方法，是object上的方法。

1. 一般情况下，对象没有构造方法， 不能加参数

|  |
| --- |
| **object** Test2{  **def** main(args: Array[String]): Unit = {  Test2 *// 这是一个对象，对象上不能有参数* Test2() *// 添加参数之后，就报错了* } } |

1. 但是，如果对象（参数列表），实际上调用的是**对象上的apply方法**，具体会根据参数列表的个数，参数的类型去找对象上对应的apply方法，如果没找到，就报错。返回值类型取决于apply的返回类型
2. apply方法的作用，3.1，快速的创建类的实例对象

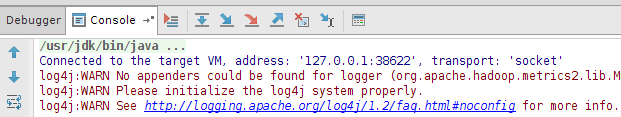
|  |
| --- |
| *// apply 方法的主要作用是创建类的实例对象* **def** apply(): ApplyDemo = **new** ApplyDemo() |

3.2，可以方便的对数组进行初始化。 *val arr = Array(1, 3, 5, 6)*

4，apply方法相当于java中的方法重载，可以定义对个apply方法，具体的参数类型，参数个数，以及返回值都是可以自定义的。

### IDEA调试程序

IDEA Debug  调试程序



Show Execution Point (Alt+F10)   显示断点位置

Step Over(F8)  下一步

Step Into (F7)   进入到代码，类似于eclipse中的F5

Force Step Into (Alt+Shift+F7)   强制进入代码，会走所有的代码流程（不常用）

Step out (Shift+F8)  跳出当前的方法（最后会回到原debug位置）

Drop Frame

Run to Cursor (Alt+F9)  调到下一个断点

### 应用程序对象

Scala程序都必须从一个对象的main方法开始，可以通过扩展App特质，不写main方法。

|  |
| --- |
| **object** AppObjectDemo **extends** App{  *//不用写main方法  println*(**"I love you Scala"**) } |

## 类

### 类的定义

在Scala中，类并不用声明为public。

Scala源文件中可以包含多个类，所有这些类都具有公有可见性。

var 修饰的变量, 这个变量对外提供getter setter方法  
val 修饰的变量,是只读属性 对外提供了getter方法,没有setter（相当于java中用final修饰的变量）

|  |
| --- |
| **class** Student {  **val** *id* = 666  *// \_ 表示一个占位符, 编译器会根据变量的具体类型赋予相应初始值  // 注意: 使用\_ 占位符是, 变量类型必须指定* **var** *name*: String = \_  *//用var修饰的变量既有getter又有setter* **var** *age*: Int = 20  }  **object** Test{  **val** *name*: String = **"zhangsan"  def** main(args: Array[String]): Unit = {   *// 调用空参构造器,* **val** student = **new** Student()  student.*name* = **"laowang"** *// 类中使用val修饰的变量不能更改  // student.age = 20   println*(**s"student.name ====== $**{student.*name*} **$**{student.*age*}**"**)  *println*(**"Test.name ======"** + Test.*name*)  } } |

### 构造器

1，主构造器

2，辅助构造器

同java中的构造方法

构造器分为两类：主构造器，辅助构造器

主构造器直接在类名后面定义。

每个类都有主构造器，主构造器的参数直接放置类名后面，与类交织在一起。

如果没有定义构造器, 类会有一个默认的空参构造器  
辅助构造器自定义，使用def this关键字，而且必须调用主构造器，或者其他的辅助构造器

注意：主构造器会执行类定义中的所有语句

|  |
| --- |
| */\*\*  \*每个类都有主构造器，主构造器的参数直接放置类名后面，与类交织在一起  \*/* **class** Person(**val** name: String, **val** age: Int){  *//主构造器会执行类定义中的所有语句  println*(**"执行主构造器"**)   **private var** *gender* = **"male"** *//用this关键字定义辅助构造器* **def this**(name: String, age: Int, gender: String){  *//每个辅助构造器必须以主构造器或其他的辅助构造器的调用开始*  **this**(name, age)  *println*(**"执行辅助构造器"**)  **this**.*gender* = gender  }  **var** *account1*:String=\_   **def this**(name:String,age:Int,gender:String){  **this**(name,age)  **this**.*gender*=gender  }  **def this**(name:String,age:Int,gender:String,account:String){  **this**(name,age,gender)  **this**.*account1*=account  }   *println*(**"尼玛,这里还是主构造器"**)  }  **object** Person {  **def** main(args: Array[String]): Unit = {  **val** s = **new** Person (**"laoduan"**, 38)  *println*(**s"$**{s.name} **$**{s.age}**"**)   **val** s1 = **new** Person (**“dingding"**, 18, **"female"**)  *println*(**s"$**{s1.*gender*}**"**)  **val** *p2* =**new** Person(**"xx2"**,12,**"female"**,**"9527"**) *println*(**s"$**{*p2*.age}**,$**{*p2*.*account1*}**"**)  } } |

scala的构造器：

1. 有两类构造器：主构造器，辅助构造器
2. 构造器的定义位置，

主构造器和类交织在一起，**class** Student2(**val** name: String, **var** age: Int)

1. 辅助构造器是一个特殊的方法，定义在类中 **def this**(name:String,age:Int,gender:String)
2. 辅助构造器，第一行必须调用主构造器（或者其他的辅助构造器）
3. 辅助构造器的参数不能和主构造器的参数完全一致（参数个数，参数类型，参数顺序）
4. 可以定义空参的辅助构造器，但是主构造器的参数必须进行初始化赋值
5. 作用域：辅助构造器的作用域，只在方法中，主构造器的作用域是类中除了成员属性和成员方法之外的所有范围（可以通过反编译查看源码）

### 访问权限

#### 成员变量的访问权限

默认权限是public 任何地方都可以访问

private 作用域 类和其伴生对象中

private [this] ，作用域为当前类中，伴生对象中无效

private [packageName]  指定包及其子包有效

|  |
| --- |
| */\* \* private var age \* age 在这个类中是有getter setter方法的 \* 但是前面如果加上了private 修饰, 也就意味着, age只能在这个类的内部以及其伴生类对象中可以访问修改 \* 其他外部类不能访问 \* \*/* **class** Student3 **private** (**val** name: String, **private var** age: Int) {   **var** *gender*: String = \_   *// 辅助构造器, 使用def this  // 在辅助构造器中必须先调用类的主构造器* **def this**(name: String, age:Int, gender: String){  **this**(name, age)  **this**.*gender* = gender  }  *// private[this]关键字标识该属性只能在类的内部访问, 伴生类不能访问* **private**[**this**] **val** *province*: String = **"北京市"   def** getAge = 18 }  *// 类的伴生对象* **object** Student3 {  **def** main(args: Array[String]): Unit = {  *// 伴生对象可以访问类的私有方法和属性* **val** s3 = **new** Student3(**"Angelababy"**, 30)  s3.age = 29  *println*(**s"$**{s3.age}**"**)  *// println(s"${s3.province}") 伴生类不能访问* } } |

#### 方法的访问权限

通用于主构造器，辅构造器，以及普通方法

默认权限是共有的

private 作用域为类和其伴生对象

private [this] ，作用域为当前类中，伴生对象中无效

private [packageName]  指定包及其子包有效 包名的写法，直接写报名，不需要层级路径

主构造器上一样适用于该方法的访问权限

private [cn.edu360.day03] **错误的**

private [day03] **正确的**

|  |
| --- |
| */\* \* private 加在主构造器前面标识这个主构造器是私有的, 外部不能访问这个构造器 \* \*/* **class** Student2 **private** (**val** name: String, **var** age: Int) {  **var** *gender*: String = \_   *// 辅助构造器, 使用def this  // 在辅助构造器中必须先调用类的主构造器* **def this**(name: String, age:Int, gender: String){  **this**(name, age)  **this**.*gender* = gender  } } **object** Student2 {  **def** main(args: Array[String]): Unit = {  **val** s1 = **new** Student2(**"laoYang"**, 18, **"male"**)  *println*(**s"$**{s1.*gender*}**"**)  } } |

#### 类包的访问权限

private 作用域为当前包及其子包  同 private [this]

private [packageName] 作用域为指定包及其子包

|  |
| --- |
| */\* \* private[包名] class 放在类最前面, 是修饰类的访问权限, 也就是说类在某些包下不可见或不能访问 \* private[edu360] class 代表student4在edu360包下及其子包下可以见, 同级包中不能访问 \* \*/* **private**[**this**] **class** Student4(**val** name: String, **private var** age: Int) {  **var** *xx*: Int = \_ } **object** Student4{   **def** main(args: Array[String]): Unit = {  **val** s = **new** Student4(**"张三"**, 20)   *println*(**s"$**{s.name}**"**)  } } |

#### 访问权限总结：

对于成员属性 ，成员方法；（主构造器，辅助构造器，普通的方法）

public：默认的访问权限

private：在类及其伴生对象中有效，其他地方无效

private [this] ：只在类的范围内有效，伴生对象中无效，其他地方无效

private [package] : 在指定包及其 子包范围内有效

package: 只能是一个具体的包名，不能是一个路径

private [scala29] 正确的

private [scala29.day04] 错误的

对于类：

public : 默认的权限 ，公有的

private ,private[this] 在当前包及其子包范围内有效

private [package] 在指定包及其子包范围内有效

### 抽象类

在Scala中, 使用**abstract**修饰的类称为抽象类. 在抽象类中可以定义属性、未实现的方法（抽象方法）和具体实现的方法。

|  |
| --- |
| */\* \* abstract修饰的类是一个抽象类 \* \*/* **abstract class** Animal {  *println*(**"Animal's constructor ...."**)  *// 定义一个name属性* **val** *name*: String = **"animal"** *// 没有任何实现的方法* **def** sleep()  *// 带有具体的实现的方法* **def** eat(f: String): Unit = {  *println*(**s"$**f**"**)  }} |

类和对象，该怎么选择？

class object

优先使用object，object本质上拥有了的类的所有特性，object中没有构造器，也没有参数

如果需要封装数据，或者需要有构造器的时候，才会使用class。

一般情况下，使用object + 伴生的class

## 特质Trait

scala中没有interface implements

Trait(特质)相当于 java的接口。比接口功能更强大。特质中可以定义属性和方法的实现。

Scala的类只能够继承单一父类，但是可以实现（继承，混入）多个特质（Trait）**使用的关键字是 with和extends**

特质不能有任何的类参数，即传递给类的主构造器的参数。

trait PointTest(x: Int, y: Int) // 编译不过

|  |
| --- |
| **trait** T1 {  *// 定义普通方法，有方法实现* **def** youcanfly()={  *println*(**"tai feng lai le you can fly"**)  } }  **trait** T2 {  *// 定义一个属性* **val** *className*: String = **"NB大神班"** *// 定义一个没有实现的方法,默认就是抽象方法* **def** teacherSay(name: String)   *// 定义带有具体的实现的方法* **def** doSomething() = {  *println*(**"群主开始发红包了..."**)  } } |

**动态混入特质。**

|  |
| --- |
| **object** test{   **def** main(args: Array[String]): Unit = {   *// 动态混入特征，让类有了特质的方法* **val** t1 = **new** Teacher **with** T1  *println*(t1.youcanfly())   *// 动态混入特质不能使用extends关键字,可同时混入多个特质* **val** t = **new** Teacher() **with** T1 **with** T2{  *// 如果特质中有抽象方法，则必须重写该抽象方法，可以不使用override关键字* **def** teacherSay(name:String)={  *println*(**s"最高face，$**{name}**"**)  }  *// 重写一个有具体的实现的方法，必须使用关键字override* **override****def** doSomething() = {  *println*(**"群主:抢到红包继续接龙..."**)  }  }  *println*(t.teach(**"laozhao"**))  *println*(t.doSomething)  println(t.youcanfly())  } } **class** Teacher{ } |

**比较：scala的trait和java中的interface的异同？**

1,java的interface只定义方法名称和参数列表，不能定义方法体。而trait则可以定义方法体。

2,在java中实现接口用implements，而在scala中，实现trait用extends和with。

3,java的interface和scala的trait的最大区别是，scala可以在一个class实例化的时候动态混入trait。

**用特质还是用抽象类？？**

1，优先使用特质。一个类可以扩展多个特质，但却只能扩展一个抽象类。

2，如果需要构造器，使用抽象类。因为抽象类可以定义带参数的构造器，而特质不行。

## 继承

### 扩展类

在Scala中扩展类的方式和Java一样都是使用extends关键字

### 重写方法

在Scala中重写一个非抽象的方法必须使用override修饰符

### 示例

|  |
| --- |
| *// 定义一个抽象类* **abstract class** Animal(**val** age: Int) {  *println*(**"Animal`s main constructor invoked"** + age)  *//定义一个抽象方法* **def** run()  **def** breath(): Unit = {  *println*(**"呼吸氧气"**)  } }  *// 定义一个特质，定义一个普通方法* **trait** Fightable {  **def** fight(): Unit = {  *println*(**"用嘴咬"**)  } }  *// 定义一个特质，一个抽象方法* **trait** Flyable { *// 定义一个抽象方法* **def** fly() }  定义一个bird类,实现多个特质  *//在scala中，不论是继承还是实现特质，第一个都用extends关键字* **class** Bird **extends** Flyable **with** Fightable {  **override def** fly(): Unit = {  *println*(**"用翅膀飞"**)  } }  **object** Bird {  **def** main(args: Array[String]): Unit = {  **val** b = **new** Bird  b.fly()  } }  定义一个类，继承类，并实现多个trait  **class** Monkey(age: Int) **extends** Animal(age) **with** Flyable **with** Fightable {  *//重写抽象的方法, 可以不加override关键字* **def** run(): Unit = { *// super()  println*(**"跳着跑"**)  }   *//重写非抽象的方法，必须加override关键字* **override def** breath(): Unit = {  *println*(**"猴子呼吸"**)  }   **override def** fly(): Unit = {  *println*(**"乘着筋斗云飞"**)  }   **override def** fight(): Unit = {  *println*(**"用棒子打"**)  }  *println*(**"monkey`s main constructor invoked"** + age) }  **object** Monkey {  **def** main(args: Array[String]): Unit = {  **val** a: Animal = **new** Monkey(100)  a.breath()  } } |

### 总结

特质和抽象类的总结：

1. 单继承，多实现
2. 继承父类，使用extends，实现（混入）特质 使用 with和extends
3. 当类没有父类，那么实现的第一个特质，使用extends关键字，其他地方都使用with
4. 不论是继承父类，还是实现特质，都必须实现抽象方法，实现抽象方法可以不用override关键字，但是如果要重写一个普通方法，必须加上override关键字
5. scala的面向对象，具备了面向对象的封装继承 和多态 三大特性。
6. 特质可以在类实例化的时候动态混入，使用的关键字是with，如果有抽象方法必须实现。动态混入了特质的对象，

|  |
| --- |
| *// 在类实例化的时候，混入特质* **val** t: Teachers **with** TraitDemo **with** TraitDemo2 = **new** Teachers() **with** TraitDemo **with** TraitDemo2 {  **override def** say(): Unit = {  *println*(**"明天放假，so happy"**)  } }  *\* 三大特性 封装 继承 多态 \* 多态的满足条件？ \* 1，继承父类 或者实现接口 \* 2，重写父类或者接口的方法 \* 3，父类引用指向子类对象，或者接口引用指向实现类 \* scala 的面向 对象*  **val** a:Animal = **new** Dog() a.eat() a.weap(**"xxoo"**) |

## 样例类/样例对象

样例类：使用case关键字 修饰的类，重要的特征就是支持模式匹配,多例

样例object：使用case关键字修饰的对象，支持模式匹配，单例

case class 和 class的一些区别：

case class在初始化的时候，不用new，而普通类初始化时必须要new。

case class 重写了toString方法。

默认实现了equals和hashCode

case class 实现了序列化接口 **with** Serializable

case class **支持模式匹配**（最重要的特征），所有case class 必须要有参数列表

有参数用case class，无参用case object

case class，和 case object,可以当作消息进行传递

|  |
| --- |
| */\* \* 样例类,使用case 关键字 修饰的类, 其重要的特征就是支持模式匹配 \* \*/* **case class** Message(msg: String)  */\*\*  \* 样例object, 不能封装数据, 其重要特征就是支持模式匹配  \*/* **case object** CheckHeartBeat  **object** TestCaseClass **extends** App{  *// 可以不使用new 关键字创建实例* **val** *msg* = *Message*(**"hello"**)  *println*(*msg*.msg) } |

# 模式匹配

候选人：

要求： 标准： 至少要满足一个条件，进一步的动作

跑车

大波浪

颜值

学历

xxx

pass，满足一个条件，

Scala有一个十分强大的模式匹配机制，可以应用到很多场合：如switch语句、类型检查等。

并且Scala还提供了样例类，对模式匹配进行了优化，可以快速进行匹配

模式匹配就是match 和 一系列的case 语法实现

其中，每一个case 里面就是一个匿名函数 =>

模式匹配的基本关键字 就是 match case

## 匹配字符串

|  |
| --- |
| **import** scala.util.Random **object** CaseDemo01 **extends** App{  **val** *arr* = *Array*(**"YoshizawaAkiho"**, **"YuiHatano"**, **"AoiSola"**)  **val** *name* = *arr*(Random.nextInt(*arr*.length))  *name* **match** {  **case "YoshizawaAkiho"** => *println*(**"xx老师..."**)  **case "YuiHatano"** => *println*(**"oo老师..."**)  **case** \_ => *println*(**"真不知道你们在说什么..."**)  } } |

## 匹配类型

|  |
| --- |
| **import** scala.util.Random **object** CaseDemo02 **extends** App{ *//val v = if(x >= 5) 1 else if(x < 2) 2.0 else "hello"* **val** *arr* = *Array*(**"hello"**, 1, 2.0, CaseDemo2)  **val** *v* = *arr*(Random.nextInt(arr.length))  *println*(*v*)  *v* **match** {  **case** x: Int => *println*(**"Int "** + x)  **case** y: Double **if**(y >= 0) => *println*(**"Double "**+ y) *// if 守卫*  **case** z: String => *println*(**"String "** + z)  **case** CaseDemo02 => {  *println*(**"case demo 2"**)  *//throw new Exception("not match exception")* }  **case** \_ => **throw new** Exception(**"not match exception"**)  } } |

**注意：**case y: Double if(y >= 0) => ...

模式匹配的时候还可以添加守卫条件。如不符合守卫条件，将掉入case \_中

## 匹配数组、元组、集合

|  |
| --- |
| **object** CaseDemo03 **extends** App{   **val** *arr* = *Array*(1, 3, 5)  *arr* **match** {  **case** *Array*(1, x, y) => *println*(x + **" "** + y)  **case** *Array*(0) => *println*(**"only 0"**)  **case** *Array*(0, \_\*) => *println*(**"0 ..."**)  **case** \_ => *println*(**"something else"**)  }   **val** *lst* = *List*(3, -1)  *lst* **match** {  **case** 0 *:: Nil* => *println*(**"only 0"**)  **case** x *::* y *:: Nil* => *println*(**s"x: $**x **y: $**y**"**)  **case** 0 *::* tail => *println*(**"0 ..."**)  **case** \_ => *println*(**"something else"**)  }   **val** *tup* = (2, 3, 5)  *tup* **match** {  **case** (2, x, y) => *println*(**s"1, $**x **, $**y**"**)  **case** (\_, z, 5) => *println*(z)  **case** \_ => *println*(**"else"**)  } } |

## 样例类

在Scala中样例类是一中特殊的类，可用于模式匹配。case class是多例的，后面要跟构造参数，case object是单例的，无需参数

|  |
| --- |
| **import** scala.util.Random  **case class** SubmitTask(id: String, name: String) **case class** HeartBeat(time: Long) **case object** CheckTimeOutTask  **object** CaseDemo04 **extends** App{  **val** *arr* = *Array*(CheckTimeOutTask, *HeartBeat*(12333), *SubmitTask*(**"0001"**, **"task-0001"**))   *arr*(Random.nextInt(*arr*.length)) **match** {  **case** *SubmitTask*(id, name) => {  *println*(**s"$**id**, $**name**"**)  }  **case** *HeartBeat*(time) => {  *println*(time)  }  **case** CheckTimeOutTask => {  *println*(**"check"**)  }  } } |

## Option类型

在Scala中Option类型样例类用来表示可能存在或也可能不存在的值(Option的子类有Some和None)。Some包装了某个值，None表示没有值

|  |
| --- |
| **object** OptionDemo {  **def** main(args: Array[String]) {  **val** map = *Map*(**"a"** -> 1, **"b"** -> 2)  **val** v = map.get(**"b"**) **match** {  **case** *Some*(i) => i  **case** None => 0  }  *println*(v)  *//更好的方式* **val** v1 = map.getOrElse(**"c"**, 0)  *println*(v1)  } } |

## 偏函数

被包在大括号内没有match的一组case语句是一个偏函数，它是PartialFunction[A, B]的一个实例，A代表输入参数类型，B代表返回类型，常用作输入模式匹配

|  |
| --- |
| **object** PartialFuncDemo {  **def** func1(num: String) : Int = num **match** {  **case "one"** => 1  **case "two"** => 2  **case** \_ => -1  }  **def** func2: PartialFunction[String, Int] = {  **case "one"** => 1  **case "two"** => 2  **case** \_ => -1  }  **def** main(args: Array[String]) {  *println*(*func1*(**"one"**))  *println*(*func2*(**"one"**))  } } |

偏函数本质上是由多个case语句组成的针对每一种可能的参数分别进行处理的一种“结构较为特殊”的函数，就是一个参数的函数。（Function1）

|  |
| --- |
| **object** PartialFunctionDemo2 {  **def** f:PartialFunction[Any,Int]={  **case** i:Int => i \*10  **case \_** => i \*10  }  **def** main(args: Array[String]): Unit = {   **val** arr = *Array*(1,3,5,**"seven"**)  *// arr.map{case t:Int =>t\*10}* **val** collect: Array[Int] = arr.collect {  **case** t: Int  => t \* 10  }  println(collect)   arr.collect(*f*).foreach(*println*)  } } |

偏函数最常用的就是方法中要求传入偏函数的类型。