1.编译scala为java文件解析

在编写完成scala文件后，会生成.class文件

```
object Test{
    def Test(args:Array[String]):Unit{
        println("hello")
    }
}
```

使用反编译软件对.class文件进行编译、补全后代码如下：

```
public class Test{
    public static void main(String[] paramArrayOfString){
        Test$.MODULE$.main(paramArrayOfString);
    }
}
final class Tst${
    public static final Test$ MODULE$;
    static{
        MODULE$ = new Test$();
    }
    public void main(String[] args){
        System.out.println("hello")
    }
}
```

可以见到，对scala进行编译后，其代码依旧是java形式。

进而推导得到：

scala的main不是scala的真正的入口，main只是被封装后的一个类

2.scala语言输出的三种形式

1.字符串通过+号连接 (类似java)

```
object printdemo{
    def main(args:Array[String]):Unit = {
        var str1 : String = "hello"
        var str2 : String = "world"

        println(str1 + str2)
    }
}
```

2.printf用法(类似C语言) 字符串通过%传值

```
object printdemo{
    def main(args:Array[String]):Unit ={
     var name:String = "tom"
     var age:Int = 10
     var sal:Float = 10.2f
        printf("名字=%s 年龄=%d 薪水=%.2f",name,age,sal)
    }
}
```

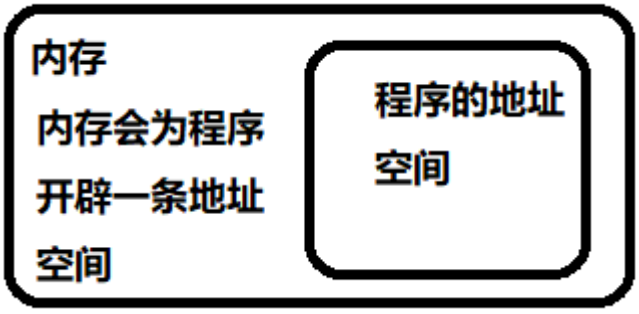
3.字符串通过$引用(类似PHP)

```
object printdemo{
    def main(args:Array[String]):Unit ={
        var name : String = "tom"
        var age : Int = 10
        println(s"姓名=$name 年龄=${age + 1}")
    }
}
```

3.生成文档说明

使用scaladoc -d d:/hello.scala　即可生成对应的文档

4.变量



1.变量是放在栈还是堆里?

java中，对象放在堆里，基本数据类型放在栈里

但是实际上，现代编译器都会做一个分析，叫逃逸分析

当一个变量满足以下条件之一时，变量会被放在堆里：

1.生命周期很长

2.被多个对象引用

同样的，当一个对象只是临时的时，也有可能会放在栈里

2.类型推导

当存在一个变量被定义时

```
var num = 10
//此时，num被自动识别为Int
```

3.var和val

var是可变的

val是不可变的

为什么要设计var和val两个类型?

1.通常情况下，需要描述多个对象时，我们会new一个对象，然后只修改、获取其本身的属性，而不是再new一个对象

例如:

```
package test.day01

object test02 {
  def main(args: Array[String]): Unit = {
    var dog:Dog = new Dog
    dog.name = "小狗"

    dog.age = 1

    println(dog.name,dog.age)

    dog = new Dog

    println(dog.name,dog.age)
  }
  class Dog{
    var age : Int = _;

    var name:String = "";
  }

}
```

```
//执行结果
小狗,1
 ,0
```

为此，我们就需要使用val，**val没有线程安全问题，因此效率较高**

```
package test.day01

object test02 {
  def main(args: Array[String]): Unit = {
    val dog:Dog = new Dog
    dog.name = "小狗"

    dog.age = 1

    println(dog.name,dog.age)
  }
  class Dog{
    var age : Int = _;

    var name:String = "";
  }

}
```

scala的设计者推荐我们尽可能使用val

2.如果对象需要改变，则使用var

例如：

```
var a = 10
a = 15
```

4.在反编译状态下，var和val的状态是怎样的?

```
//存在一个变量a和常量b
var a = 1;
val b = 2;
```

反编译后：

```
private Int a = 1;
private final Int b = 1
```

可以见得，scala的val反编译后**被final修饰，所以其不可被修改**

为什么被java 的final修饰后，变量就变得无法修改了呢?

因为在java中，**final修饰的变量不提供set方法，只提供get方法，所以不能通过公开的set方法对final修饰的变量进行更改**

5.数据类型与结构

scala和java有相同数据类型（也有不同的），scala中数据类型都是对象，也就是scala中没有java中的原生类型

scala数据类型分为 **AnyVal（值类型）和AnyRef**（引用类型）

不论是 AnyVal 还是 AnyRef 都是对象

如何体现数据类型是对象?

```
//存在一个Int类型的num
num : Int = 10

//进入源码中查看
final abstract class Int private extends AnyVal{}
```

可以见到，Int类型是被class修饰，且继承于AnyVal的

而java的int类型并不是被class修饰的

意味着scala的Int类型是一个类，在分门别类的思想下，可以知道Int就是一个对象，因为只有对象才有"种类"

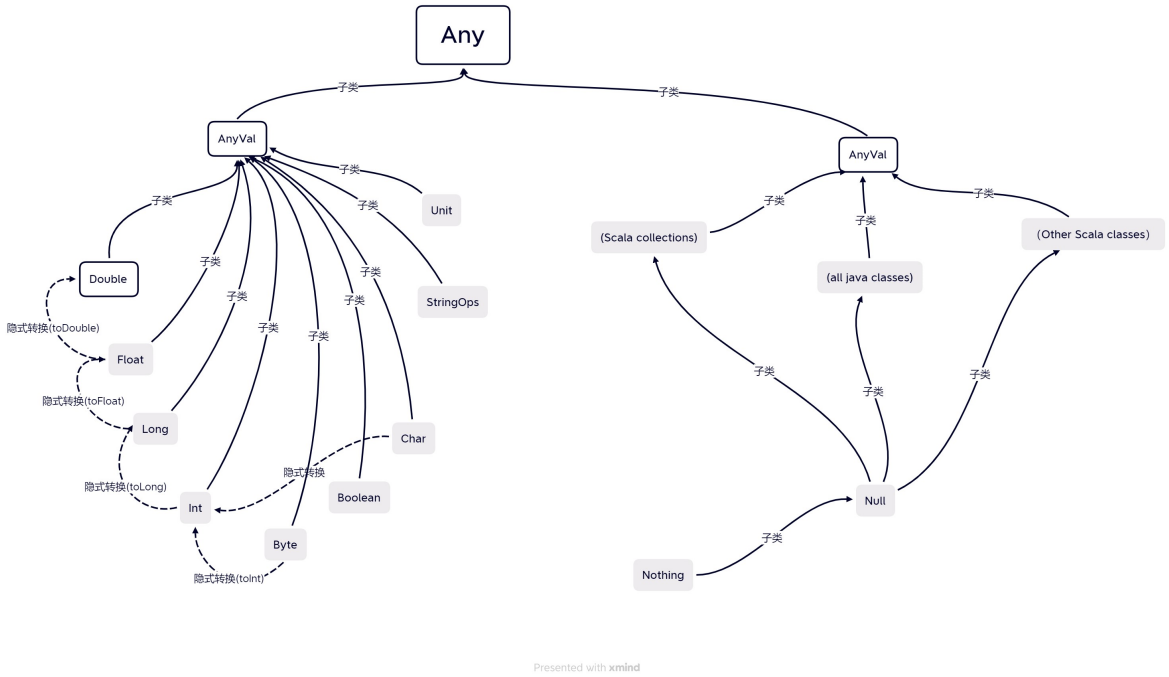
且因为Int是一个类，因此他的一个实例，就可以使用很多方法

题外:
　在scala中，如果一个方法没有形参，则可以省略()

例如:

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

sayHi
```

在scala中，**一切皆为对象**



可以见的，**Null是所有AnyRef的子类，而Nothing是所有类的子类**(包括Any、AnyVal等)，这样一来Nothing就可以存在于任何对象里

最典型的就是抛异常，如果Nothing不是对象的子类，那么就无法通过throw抛出Nothing异常

总结：

1.在scala中有一个跟类型Any，Any是所有类的父类

2.scala中一切皆为对象，分为两大类，AnyVal(值类型)和AnyRef(引用类型)，他们都是Any子类

3.Null类型是scala的特别类型，Null类型只有一个类型：null，他是一个bottom class，是所有AnyRef类型的子类

4.Nothing 类型也是 bottom class，他是所有类型(AnyVal、AnyRef)的子类，在开发中，通常可以将Nothing的值返回给任意变量或者函数

Nothing案例：

```
object TypedemoO1{
    def main(args:Array[String]) : Unit={

    }
    //比如开发中，存在一个方法，一定会存在异常中断时，这时就可以返回Nothing
    //即当我们用Nothing做返回值时，就是明确说明该方法没有正常返回值
    def sayHello:Nothing={
        throw new Exception("抛出Nothing异常")
    }
}
```

```
结果:
Connected to the target VM, address: '127.0.0.1:59043', transport: 'socket'
Exception in thread "main" java.lang.Exception: 抛出Nothing异常
    at test.day02.test01$.throws(test01.scala:13)
    at test.day02.test01$.main(test01.scala:9)
    at test.day02.test01.main(test01.scala)
Disconnected from the target VM, address: '127.0.0.1:59043', transport: 'socket'
```

返回值也可以不用Nothing，可以是任何类型如Unit，因为Nothing是Unit的子类

在scala中，依然遵守 低精度值 向高精度 的值自动转换(implicit conversion)隐式转换

需要注意的是，scala的+-或其他运算符是直接定义在基本类型里的，也就是说运算符是基本类型的方法，而Any和AnyVal等超类，是没有定义自己的运算方法的，所以超类无法进行计算。

1.整数类型

1.scala各整数类型有固定的表数范围和字段长度，不受具体OS的影响，以保证scala程序的可移植性

2.scala的整型　常量/字面量 都默认为Int，声明 Long 型 常量/字面量 须后加 'l' 或 'L'

```
val b = 10000
//Int型
val a = 10000L
//Long型
```

3.scala程序中变量常声明为Int型，除非不足以表示大数时才用Long

当没有使用 'l' 或 'L' 声明Long类型，但整数长度大于Int时，编译会报错

`Integer number is out of range for type Int`

```
var i = 15151515151515151515151515151551515151515
```

此时编译不会通过，因为Int不支持如此之长的整数，需要使用 'l' 或 'L'指定Long类型

```
var i = 15151515151515151515151515151551515151515L
```

2.浮点类型

与整数类型类似，scala默认的浮点类型时双精度浮点型(Double)，要声明 Float 需要在 常量/字面量 后加上 'f' 或 'F'

```
var a = 10.1
//Double型

var b = 10.1f
//Float型
```

编译规则也是一样的

浮点型常量拥有两种表达形式：

1.十进制形式

`5.12、.512等`

2.科学计数法

`5.12e2、5.12E-2等`

3.字符类型

scala的字符Char会将整数对照着unicode表进行输出，unicode码表包括了ASCII码：

```
var char1 : Char = 97
println("char1="char1)
```

`结果：A`

字符Char 也可以隐式转换成 Int

但是，Int 无法转换成 Char

如：

```
var char : Char = 'a'+1
var c2 : Char = 97 + 1
//这两种写法不能够通过
var c3 : Char = 98
//这种写法可以通过
```

为什么char = 98可以通过，而另外两种却不能呢？

如果直接将整数 98 赋给Char类型，那么 Char 类型将自动解释为Unicode字符码，因为Scala将字符与整数进行了隐式映射

如果将字符型和整形进行算术计算，那么Scala则需要根据语法规范执行类型检查，在Scala中，字符类型被视为无符号整数，且可以进行算术计算，但是由于类型推断的限制，使得不能直接将结果赋给字符类型

详细地说，表达式'a' + 1 进行了字符类型和整数类型之间的加法运算，根据语言规范，加法运算两边类型应该一致，这种情况下scala会将字符类型 隐式转换成为 整数类型，因此运算后的结果是整数类型，而整数类型是无法赋给一个字符类型的，所以编译不能通过

解决方法：

可以调用Int的方法 .toChar进行类型的隐式转换即可

```
var a :Char = ('a' + 1).toChar
```

总之，直接将整数赋给Char类型会解释为对应的unicode字符；而在进行运算后，需要进行类型检查和显示转换问题

字符类型存储到计算机中，需要将字符对应的码值（整数）找出来

存储：字符 --> 码值 --> 二进制 --> 存储

读取：二进制 --> 码值 --> 字符 -->读取

字符和码值的对应关系是通过字符编码表决定的（已经规定好的），这一点和java一致

4.布尔类型

1.scala中，**Boolean 数据类型只允许取 ture 和 false**，不接受如0、1、'真'、'假' 等进行返回或判断

2.Boolean 类型只占一个字节

3.Boolean 类型适用于逻辑运算，一般用于流程控制

5.特殊类型——Unit、Null、Nothing

1.Unit 表示无值，和其他语言中的void相同，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，写作();

2.Null null，Null类型只有一个实例值 null;

3.Nothing Nothing类型在Scala的类层级的最低端，他是所有类型的子类。当一个函数在确定没有正常的返回值时（一定会报错的函数），就可以用Nothing来指定返回类型，这样可以把返回的异常赋给其他函数或者变量，使其能够兼容报错的函数

三个数据类型的实例：

Unit：

```
object Unit{
    def main(args:Array[String]):Unit={
        val res = sayHello
        println(res)
    }
    def sayHello():Unit={

    }
}
```

`输出结果:()`

`意味着Unit输出的结果为一个空`

Null:

```
object Null{
    def main(args:Array[String]):Unit={
        val dog : Dog = null
        val a : Char = null
        //此时，在运行时会报错，报错内容如下
    }
}
class Dog{
}
```

`an expression of type Null is ineligible for implicit conversion`

`val a : Char = null`

表示无法将 null 隐式转换为 Char

这是因为 Null 类只有一个实例对象：null，类似于java中的null引用，null可以赋值给任意引用类型（AnyRef），但是不能赋值给值类型（AnyVal 及 AnyVal 的子类，如Int、Float、Char等）

总结一：

1.对3进行开方后，再对其进行平方，最后的结果与原值应该差多少?

知识点：精度损失

```
object EndP{
    class main(args:Array[String]):Unit{
        var i = 3
        var j = math.sqrt(i)
        println(math.pow(j,2))
    }
}
```

`结果：2.9999999999999996`

可以知道，相较于最开始的3，精度损失了0.00000000004

2.Scala语言的sdk是什么?

3.Scala程序的编写、编译、运行步骤，能否一步执行

`答：先编写，再编译，最后运行；可以一步执行`

4.Scala程序编写规则

5.如何检测一个变量是val 还是var?

答：`为变量进行赋值，如果能被修改则是var，否则为val`

6.Scala 允许用数字取乘一个字符串，使用"crazy"*3。这个操作做了什么？再scaladoc中怎么招这个操作

`答：crazy会被重复输出三次，可以推断出，字符串也拥有运算符，运算符可以对字符串进行一些控制，如"crazy" + "3"则会输出crazy3，`

7.（10 max 2） 的含义是什么？max方法定义在哪个类中?

9.BigInt计算 2 的 1024次方

————————————————————————————————————

6.自动类型转换(隐式转换)

自动类型转换 也叫 隐式转换

当Scala程序再进行赋值或者运算时，**精度小的类型自动转换为精度大的数据类型**，这个就是自动类型转换

这种特性java也有，scala的 自动类型转换 和java几乎一样

细节特性：

1.在有多种数据类型混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后进行计算

2.当我们把精度（容量）最大的数据类型赋值给精度（容量）小的数据类型的时候，会发生报错，反之则会进行 自动类型转换

3.（byte，short）和 char 之间不会相互自动转换

4.byte，short，char 三者可以计算，如char+char，在计算时首先转换为Int 类型

5.遵循 自动提升原则，表达式结果的类型自动提升为 操作数中最大的数据类型

7.强制类型转换

强制类型转换 可以理解为 显式转换

强制类型转换 是主动的，而不是自动的

强制类型转换 是 自动类型转换 的逆过程，将容量大的数据类型转换为容量小的数据类型，使用时要加上强制转换函数，**但可能造成精度降低或溢出**，需要格外注意

在java中，强制类型转换操作如下：

```
int num = (int)2.5
```

在scala中，强制类型转换操作如下：

```
var num:Int = 2.7.toInt
```

强制类型转换细节：

1.当数据从大到小时，就需要强制转换

2.强转符号只针对于最近的操作有效，往往会使用小括号提升优先度

3.通常Char类型可以保存Int类型的常量值，但不能保存Int类型的变量值，此时需要强转

4.Byte 和 Short 类型在进行运算时，当作Int类型处理

8.关系运算符（比较运算符）

```
var a = 9
var b = 8
println(a>b) //true
println(a<b) //false
println(a>=b)//true
println(a<=b)//false
println(a==b)//false
println(a!=b)//true
val f:Boolean = a>b//true
```

使用细节：

1.关系运算符的结果都是boolean，要么时true，要么是false（不存在0、1等）

2.关系运算符组成的表达式，我们叫关系表达式

3.比较运算符是==而非=

4.如果两个浮点数进行比较，必须保证数据类型一致

9.逻辑运算符

运算符	描述	实例
&&	逻辑与	A && B
||	逻辑或	A || B
!	逻辑非	!(A && B)

逻辑运算符的返回结果都是Boolean类型，即只有两个值：true 和 false

10.赋值运算符

运算符	描述	实例
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2
&=	按位与后赋值	C &= 2 等于 C = C& 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
|=	按位或后赋值	C |= 2 等于 C = C | 2

11.顺序控制

顺序控制注意事项：

scala中定义变量时采用合法的**向前引用**

即在使用变量前，变量必须已经被创建了

```
val a = 1
val b = 2
val c = a + b
//此时是有效的

val a = 1
val c = a + b
val b = 2
//此时报错，因为b的创建晚于b的引用
```

但是函数的顺序是可以不遵守

1.if-else

if-else

```
if(i==1){
    printf("如果i==1，则执行这里面的代码")
}else{
    printf("如果没有满足i==2，则执行这里的代码")
}
```

执行原理：
scala的 if 表达式源码定位于 scala.Predef对象中，代码如下：

```
object Predef{
...
def ifThenElse[T](cond: Boolean, thenp: => T, elsep: => T): T =
    if (cond) thenp else elsep
...
}
```

可以看到，if是通过ifThenElse来完成的，方法主要接受三个参数：条件表达式 cond、当条件为真时执行的代码块 thenp 和条件为假时执行的代码块 elsep。此方法使用了类型参数T，使得返回值可以根据实际情况判断

在编译过程中，scala会将if表达式转换为对应的字节码指令

```
val x = if(condition){

}else{

}
```

编译器会将其转换为类似以下的字节码指令

```
val x = ifThenElse(condition,{

},{

})
```

2.for

for

```
//遍历区间
for(i <- 1 to 5){
    println(i)
}

//遍历集合
for(li <- list){
    println(li)
}

//带有条件筛选的遍历
for(li <- list if li < 10){
    println(li)
}

//带有生成器和条件的遍历
for{
    i <- 1 to 3
    j < - 1 to 3 if i != j
}{
    println(i,j)
```

```
}
```

for支持使用yield 关键字来生成新的集合或者结果

以下代码会生成一个包含1到5的每个元素乘以2的新集合

```
val d = for(i <- 1 to 5) yield i * 2
println(i)
```

for循环返回值

`Vector(2,4,6,8,10)`

```
val c = for(i <- 1 to 5)yield{
    if(i % 2 == 0){
        i
    }else{
        "不是偶数"
    }
}
```

以上案例能够体现scala的一个重要的语法特点，就是将一个集合的每个数据进行处理，并返回给新的集合

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

i可以是一个代码块，这就意味着我们可以对i进行处理

3.while

while

```
while(true){

}
```

12.类和对象

假设存在一个类 CheckTest()

```
class CheckTest{
    int sum = 0;
}
```

此时实例化一次类CheckTest

```
val a1 = new CheckTest
```

然后再一次实例化一个类

```
val a2 = new CheckTest
```

最后为a1的sum赋值

```
a1.sum = 1
```

需要注意的是：此时是有两个sum变量，修改其中一个sum不会影响到另一个sum，因为他们两个不属于同一个对象

另外需要注意的是：虽然a1和a2都被val修饰了，但是不可修改的特性仅仅限制a1和a2，使其不能再次赋值为其他对象而已；而对于对象的属性，则是可以进行改动的。用一段话来说就是 我的final不是我对象的final

保证对象实例的值 在对象的整个生命周期都有效，这个我们称之为保证对象的状态，这是保证对象健壮性的重要方法之一，做法如下：

第一步：

通过private关键字将字段变为私有以阻止外界对他的直接访问

```
class CheckTest{
    private var sum = 0
}
```

此时使用a1进行调用就会发生报错，编译不通过

```
val a1 = new CheckTest
a1.sum = 1//报错
```

13.单例对象(Singleton对象)

scala比java更为面向对象的特点之一就是 scala不能定义静态成员，而是以单例对象代替之。

定义单例对象时，除了用object关键字替换了class关键字外，单例对象的定义与类的定义别无二致

```
object Check{
    class main(args:Array[String]):Unit={
    }
    val a = 1
    if(a == 1)
    ...
}
```

当单例对象与某个类共享一个名称时，单例对象就被称为是这个类的伴生对象。

类和他的伴生对象必须定义在同一个源文件中

类被称为这个单例对象的伴生类

类和他的伴生对象可以互相访问对方的私有成员/属性

定义单例对象并不需要定义类型（仅限于scala的抽象层次），单例对象拓展了父类，并且可以混入特质，一次你可以使用类型调用单例对象的方法，胡总和使用类型的实例变量代指单例对象，并把它传递给需要类型参数的方法

一般类和单例对象的差别是，单例对象不带参数，而类**可以**带参数。单例对象不是用new关键字实例化的，所以没有办法和机会为他传递实例化参数。

每个单例对象都被时限为虚构类的实例，并指向静态的变量，因此单例对象与java的静态类有着相同的初始化语义

下列是单例对象反编译后的结果，从代码中可以很清晰地看到单例对象的特性

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

    }
}
```

反编译后：

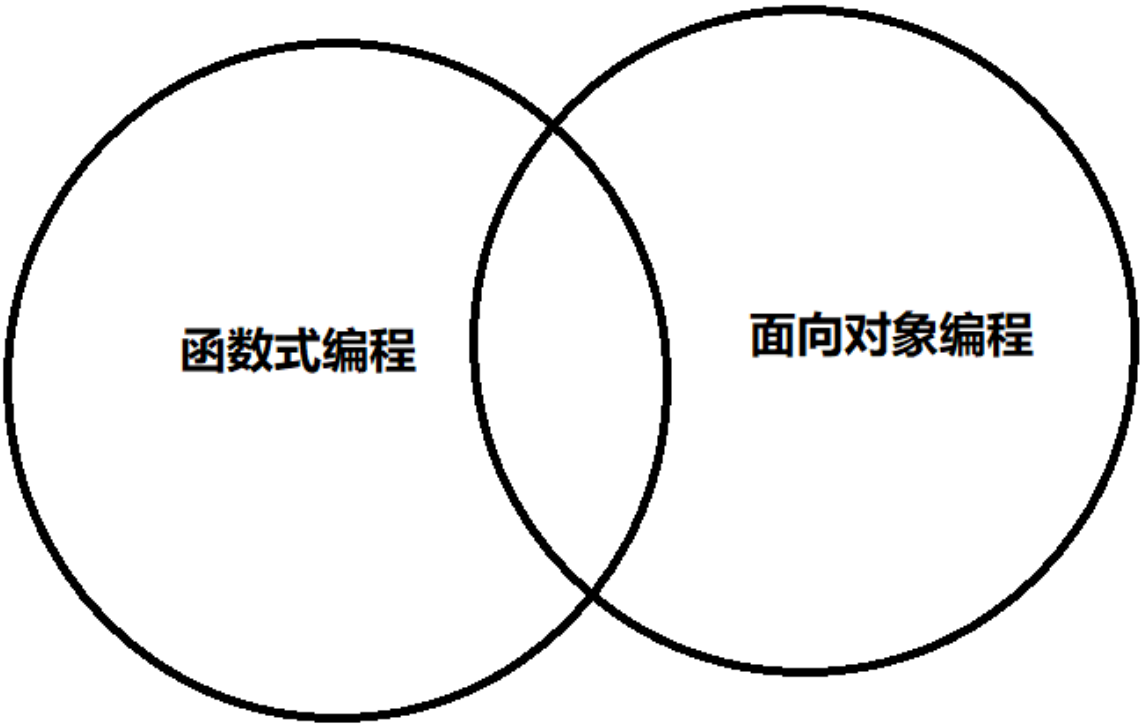
```
public final class SLdemo$ {
  public static SLdemo$ MODULE$;

  public void main(String[] args) {}

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

可以见得，在拥有单例对象时，编译后的java会在主类上加上final关键字，此时意味着单例对象已经成型后就不能再被修改，这大大提高了安全性，但也降低了灵活性

由于单例对象的不可修改性，所以单例对象也不需要参数

14.函数式编程

1.函数式编程是和面向对象编程融合在一起，学习函数式编程需要oop知识



学习顺序：函数式编程 -->面向对象编程 -->函数式高级编程

2.函数式基础编程

scala方法和函数的关系

方法转换为函数

```
object model {
    def main(args: Array[String]):Unit={
        //使用方法
        //先创建一个对象
        val dog = new Dog
        println(dog.sum(10,20))

        //方法转换成函数
        val f1 = dog.sum _
        println(f1)
    }
}

class Dog{
    //方法
    def sum(n1:Int,n2:Int):Int={
        n1+n2
    }
}
```

执行结果

`30`

`<function2>`

由此可知，函数的返回类型是function

深入：

进入到function2.scala中查看源码，可以发现

```
trait Function2[@scala.specialized -T1, @scala.specialized -T2,@scala.specialized
+R] extends scala.AnyRef{
    def apply(v1: T1,v2 : T2):R

    @scala.annotation.unspecialized
    defcurried : scala.Function1[T1,scala.Function[T2, R]]={}

    @scala.annotation.unspecialized
    def tupled : scala.Function1[Scala.Tuple2[T1,T2],R]={}

    override def toString() : java.lang.String = {}
}
```

可以见的，Function2的基础是Function1，看到 Function2 继承自 AnyRef 也就是 引用基类，可以推断实现的 Function1 也是继承自 AnyRef

Function2中有三个形参，分别是 -T1 , -T2 , +R，即两个**逆变类型参数** (-T)，一个**协变类型参数** (+R)

逆变类型参数 (-T):

逆变意味着类型参数可以在类型层次结构中向上转换。例如：如果类型A是类型B的子类，则Function1[B,C,T]是Fcuntion[A,C,T]的子类型，逆变类型参数在函数参数位置使用较多

进一步说明：

假设A是B的子类，那么**定义A时也可以被视为定义B，当需要使用B类型时，传入A类型也能正常执行**

比如Dog是Animal的子类，当需要指Animal时，把Dog传过去也能正常运行，**因为Dog也是Animal**

协变类型参数 (+R):

协变意味着类型参数可以在类型层次结构中向下转换。例如：如果A类型是B类型的父类，则Funtion1[T,C,A]是Funtion1[T,C,B]的子类型。协变类型参数在函数返回值的位置使用较多

进一步说明：

协变也是同理，假设A是B的父类，那么定义A时也被视为定义B，**当需要B类型时，使用A也能照常运行**

而**@scala.annotation.specialized**是一个注解，告诉编译器对类型参数进行特殊化处理，以提高性能，他会针对特定的基本类型优化代码，**以避免装箱和拆箱操作。**

@scala.annotation.unspecialized也是一个注解，告诉编译器不对特定的方法进行特殊化处理，即不对代码进行优化，通常用于希望避免过度优化而导致编译时间过长的情况

而代码中的 apply 方法是一个特殊的方法，它可以被调用时省略方法名。实际上当调用函数一样调用一个对象时，scala会自动查找并调用这个对象的apply方法。

apply可以理解为对象的初始化方法，只要对象被调用，scala就会开始寻找对象中的apply方法

```
object ApplyDemo02 {
  def apply(): Unit={
    println("apply被调用")
  }
  def main(args: Array[String]): Unit = {
    ApplyDemo02()
  }

}
```

`apply方法被调用`

经典的apply案例：

```
object ApplyDemo01 {
    def apply[A,B](func:A => B)(args: A*): Seq[B] = {
        args.map(func)
    }
    def main(args:Array[Stirng]):Unit={
        val addOne = (x:Int) => x + 1
        val result = ApplyDemo(addOne)(1,2,3))
        println(result)
    }
}
```

`Vector(2,3,4)`

可以见到，在调用了类时，apply方法直接被调用了，也就是可以认为，apply方法是类的形参，也是类一个独立的代码块、方法体

思考：

函数会不会自带一个apply?

```
object ApplyDemo03{
    def main(args:Array[String]):Unit={
        ApplyDemo01()
    }
}
```

此时发生报错， 编译不通过

`ApplyDemo03.type does not take parameters`

此时我们可以得知：

函数不会自动添加apply，scala虽然会扫描apply方法，但是不是强制的

继续思考：

能不能使用其他方式代替apply?

```
object ApplyDemo04{
    def main(args:Array[String]):Unit={
        val m = new Monkey
        val t = m.sum _

        println(t(1,2)
    }
}
class monkey{
    def sum(n1:Int,n2:Int):Int={
        n1+n2
    }
}
```

方法函数化确实能够达到类似的效果，但他的本质还是变量 t 引用 m的方法而已，并没有达到代替apply的目的

代替apply可以使用以下两种方法：

1.自定义方法名

2.使用辅助对象

```
class ApplyDemo05(val name: String)

object ApplyDemo05 {
  def apply(name: String): ApplyDemo05 = {
    new ApplyDemo05(name)
  }

  def main(args: Array[String]): Unit = {
    val obj = ApplyDemo05("example")  // 直接调用 apply 方法来创建对象
    println(obj)
  }
}
```

总结：

1.在scala中，方法和函数几乎可以等同（比如定义、使用、运行机制都是一样的），只是函数的使用更加灵活

2.函数式编程时从编程的方式（范式）的角度来谈的：函数式编程把函数当作一等公民（函数是一等公民，和变量一样，既可以作为函数的参数使用，也可以将函数赋值给一个变量，函数的创建不用依赖于类或者对象，而java中，函数的创建要依赖于类、抽象类或者接口），充分利用函数、支持的函数多种使用方式

3.面向对象编程是以对象为基础的编程方式

4.在scala中函数式编程和面向对象编程融合在一起了

此处的toString方法仅仅用于指定包的地址

1.1函数的定义：

基本语法：

```
def 函数名([参数名：参数类型],...)[[:返回类型]=]{
    语句...
    return 返回值
}
```

1.函数声明关键字为def（definition）

2.[参数名:参数类型],...:表示函数的输入（就是参数列表），可以没有。如果有，多个参数使用逗号相隔

3.函数中的语句：表示为了实现某一个功能代码块

4.函数可以有返回值，也可以没有

5.返回值形式1：：返回值类型=

6.返回值形式2：=表示返回值类型不确定，使用类型推导完成

7.返回值形式3：表示没有返回值，return不生效

8.如果没有return，默认以执行到最后一行的结果作为返回值

1.2递归的使用

递归就是一个函数调用了函数自己

```
def test(n:Int){
    if(n>2){
        test(n-1)
    }
    println("n=" + n)
}
```

这一段函数在`test(n-1)`处调用了自己，所以这就是递归

要搞清楚什么是递归，就要先搞清楚什么是迭代

迭代法，也叫穷举、枚举法，更多地用在输出数组总和上，图解如下：

 image-20230703144916182

也就是从下标0开始，一直迭代到最后一个下标

在迭代中，首先拿到前n项的和，再和下一项进行相加，就得到了当前项的和

具体代码如下

```
object DieDai{
    def main(args:Array[String]):Unit={
        sum(List(1,2,3,4,5,6))
    }
    def sum(a:List[Int]):Unit={
        var b = 0
        for(i <- a){
            b += i
            println(b)
        }
    }
}
```

`1,3,6,10,15`

而递归算法则相反

再递归中，想要知道第10项的总和，就要知道第10项的数，和前9项的总和

想要知道前9项的总和，就要知道第9项的数，和前8项的总和

想要知道前8项的总和，就要知道第8项的数，和前7项的总和

.....

总而言之，想要知道第n项的总和，就要先知道前n-1项的总和

 image-20230703151101045

具体实现如下：

```
object DiGui{
    def main(args:Array[String])=Unit{
        sum(6)
    }
    def sum(a:Int)=Unit={
        if(a > 2){
            sum(a-1)
        }
        println(a)
    }
}
```

`2,3,4,5,6`

可以见的，以-1的方式，从小到大依次输出了

也可以反过来证明上图，先将总数(6)压入栈底，然后将6-1(5)压入栈，依次类推，直到2为止，随后依次取出，由于2最后压入栈，所以2最先出栈，随后3、4、5、6

还可以反推：每存在一个函数被执行时，就会有一个栈被开辟

递归也可以认为是一种枚举

递归与迭代的不同点：

迭代是人为控制(栈在人脑中)

递归是完全有机器执行(栈在机器中)

迭代能够受人的影响，效率低下但可靠性高

递归不一定受人的影响，当递归无法被控制时，栈会变得越来越高从而使得内存无法承受(栈溢出)，效率高但可靠性低于迭代

函数注意事项和细节：

1.函数的形参列表可以是多个，如果函数没有形参，调用时可以不带()

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

2.形参列表和返回值列表的数据类型可以是值类型(AnyVal)和引用类型(AnyRef)

3.scala中的函数可以根据函数体最后一行代码自行推断函数返回值类型

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

    }
    def sum(){
        1+1
    }
}
```

此时sum的类型被自动推断为Int类型

4.由于scala能够自行推断，所以在省略return关键字的场合，返回值类型也可以省略

5.如果函数明确使用return关键字，那么函数返回就不能使用自行推断了，这时要明确写成：返回类型=，如果不写的话，即使有return 返回值也会为(),因为不写的话，就**向函数表明该函数没有返回值**

```
object test{
    def main(args:Array[String]):Unit={
        sum
    }
    def sum():Int={
        return 1+1
    }
}
```

2

```
object test{
    def main(args:Array[String]){
        sum
    }
    def sum(){
        return 1+1
    }
}
```

()

6.如果函数明确声明无返回值（声明Unit），那么即使函数体内有return，那么也不会有返回值

```
object test{
    def main(args:Array[String]){

    }
    def sum():Unit={
        return 1+1
    }
}
```

()

7.如果明确函数无返回值或不确定返回值类型，那么返回值类型可以省略

8.scala语法中任何语法结构都可以嵌套其他语法结构，即 函数中可以再次声明、定义函数，类中可以再声明、定义类，方法中可以再次声明、定义方法

```
object test{
    def main(args:Array[String]){
        def a(){
            def b(){
                def c(){
                    ...
                }
            }
        }
    }
}
```

9.scala函数的形参，再声明函数时，直接赋初始值（默认值），这时在调用函数时，如果没有指定实参，就会使用默认值。如果指定了实参，实参会覆盖默认值

```
object test{
    def main(args:Array[String]):Unit={
        a()//注意，a是有形参的，不论给不给值都需要加上()
        b(2)
    }
    def a(b:Int = 1){
        println(b)
    }
    def c(d:Int = 1){
        println(c)
    }
}
```

`1,2`

10.如果函数存在多个参数，每一个参数都可以设置默认值，那么这个时候，传递的参数到底是覆盖默认值，还是赋给没有默认值的参数？这是不确定的（默认按照声明顺序[从左到右]）。这种情况下，可以采用带名参数

11.scala函数的形参默认是val的，因此不能再函数中进行修改

```
//如同
object test{
    def main(args:Array[String]):Unit={
    }
    def a(val a : Int = 1){
}
}
```

过程

将函数的返回类型为Unit的函数称之为过程(procedure)，如果明确函数没有返回值，那么等号可以省略

```
//f10没有返回值，可以使用Unit来说明
//这时，这个函数我们也叫做过程
def f10(name:String):Unit={
    println(name)
}
```

惰性函数

了解惰性函数之前，要先了解惰性计算

惰性计算

　尽可能地延迟表达式求值

惰性计算是许多编程语言的特性

在定义了惰性的变量/函数后，只有在变量/函数被调用或引用时才会开始计算/加载

一般来讲，可以将耗时的计算推迟到绝对需要的时候，其次，可以先创建无限个变量/集合/函数，然后让他们等待调用，这可以使资源得到更好地利用(在被调用之前，不会占用计算的资源)

在java中没有提供惰性，但有拥有惰性特性的API

scala的惰性：

scala的惰性实际上是对 lazy 关键字修饰的变量/函数进行一个标记，拥有这个标记的变量在未被访问时不做任何动作，当被第一次访问时被计算，之后返回计算的结果，随后将实际值缓存起来，在以后的调用将直接交予缓存中的值(未改变的情况下)而非重新计算

```
object Lazytest{
    def main(args:Array[String]):Unit={
        val LazyValue:Int={
            println("C
            ")
        }

    }
}
```

属性、成员变量

　1.属性的定义方法与变量相同

```
var a : Int = 1
```

属性的定义默认的权限是私有的(private)，只是会对外暴露两个公开的方法对属性进行读写

　2.属性的定义类型可以为任意类型，包含值类型或引用类型

```
var a : Null =null
```

　3.Scala中声明一个属性，必须**要初始化**，然后根据初始化数据的类型自动推断，属性类型可以省略(这点和java不同)

　4.如果赋值为null，一定要加类型，因为不加类型，那么该属性的类型就是Null类型

2.Lambda

基础lambda

```
val f2 = (n1:Int,n2:Int)=> n1 + n2
```

15.异常处理

scala提供了 try 和 catch 块来处理异常。

try块用于包含可能出错的代码

catch块用于处理try块中发生的异常

try..catch不受数量控制

语法上与java类似，只在小范围地方有区别

以下是**java异常**处理：

```
try{
    //可能会报错的代码
    float a = 1/0
}catch(Exception e){
    //处理报错
    e.printStackTrace();
    System.out.println("分母不为0");
}finally{
    //catch内不包含的错误，可以统一在这里处理掉。finally始终执行
    System.out.println("有没有被处理的异常，将在这里被统一处理");
}
```

结果

```
java.lang.ArithmeticException: / by zero
    at test.day03.JavaExceptionDemo.main(JavaExceptionDemo.java:6)
分母不为0
有没有被处理的异常，将在这里被统一处理
```

Java异常处理的特点

1.java语言按照try - catch - catch - finally 的方式处理异常

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

3.可以有多个catch，分别捕获不同的异常，这时需要按照 先范围小，后返回大的规则书写，范围小的异常类写在前面。否则会提示 "Exception 'java.lang.xxxxxx' has already been caught"。finally经常用来释放资源

scala异常：

```
try{
    val r = 1/0
}catch{
    case ex:Exception => println("捕获异常")
}finally{
    println("finally被执行")
}
```

```
捕获异常
finally
```

可以见得，scala异常与java异常的区别不仅仅是语法上的区别，还有返回值的区别

在java中，报错后即使使用了catch，异常依旧会被输出

而scala中，报错后使用了catch，只有case中的内容会被输出

scala异常说明：

1.在scala中只有一个catch

2.catch中有多个case，每个case匹配一种异常

3.=>是关键符号，表示后面是对该异常的处理代码块

4.finally是最终要处理的，且始终会被执行

小结：

1.将可能报错的代码封装在try块中，在try块之后使用了一个catch处理程序来捕获异常，如果发生任何异常，catch处理程序将处理它，程序不会异常终止

2.scala的一场工作机制与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的

3.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

```scala
def main(args:Array[String]):Unit={
    val res = test()
    println(res.toString)
}
def test():Nothing={
    throw new Exception("发生错误")
}
```

```
Exception in thread "main" java.lang.Exception: 发生错误
    at test.day03.throwAble$.test(throwAble.scala:9)
    at test.day03.throwAble$.main(throwAble.scala:5)
    at test.day03.throwAble.main(throwAble.scala)
```

需要注意的是，这种手动抛出异常也需要进行处理，否则代码将会停止工作

```scala
def main(args:Array[String]):Unit={
    try{
        val res = test()

    }catch{
        case ex:Exception => println("捕获异常")
    }finally{
        println("111")
    }
    println("代码继续执行")
}
def test():Nothing={
    throw new Exception("发生异常")
}
```

```
异常捕获
111
代码继续执行
```

可见，当捕获异常后，不会影响程序的执行

如果想要打印异常内容，可以使用getMessage

```
def main(args:Array[String]):Nothing={
    try{
        val res = test()
    }catch{
        case ex:Exception => println("捕获异常=》" + ex.getMessage)
    }finally{
        println("111")
    }
    println("代码继续执行")
}
def test():Nothing={
    throw new Exception("发生异常")
}
```

```
异常捕获=》发生错误
111
代码继续执行
```

一般来讲，finally里写的是对try{}中的资源的分配

比如在做大数据处理时，很多数据都是从文件中读取的，对文件指针的引用或者关闭，都会写在这里面

16.面向对象高级内容

1.类与对象

如何创建对象：

基本语法

```
val 或者 var 对象名 [: 类型] = new 类型()
```

说明：

1.如果不希望改变对象的引用（即内存地址），就应该声明为val性质的，否则声明为var，scala设计者推荐使用val，因为在程序中，我们一般只改变对象属性的值，而不改变对象的引用

2.scala在声明对象变量时，可以根据创建对象的类型自动判断，所以类型声明可以省略，**但当类型和new对象类型有继承关系（即多态）时，就必须写上类型声明**

如果希望在new对象的时候，将子类对象交给父类的引用，这时需要写上类型

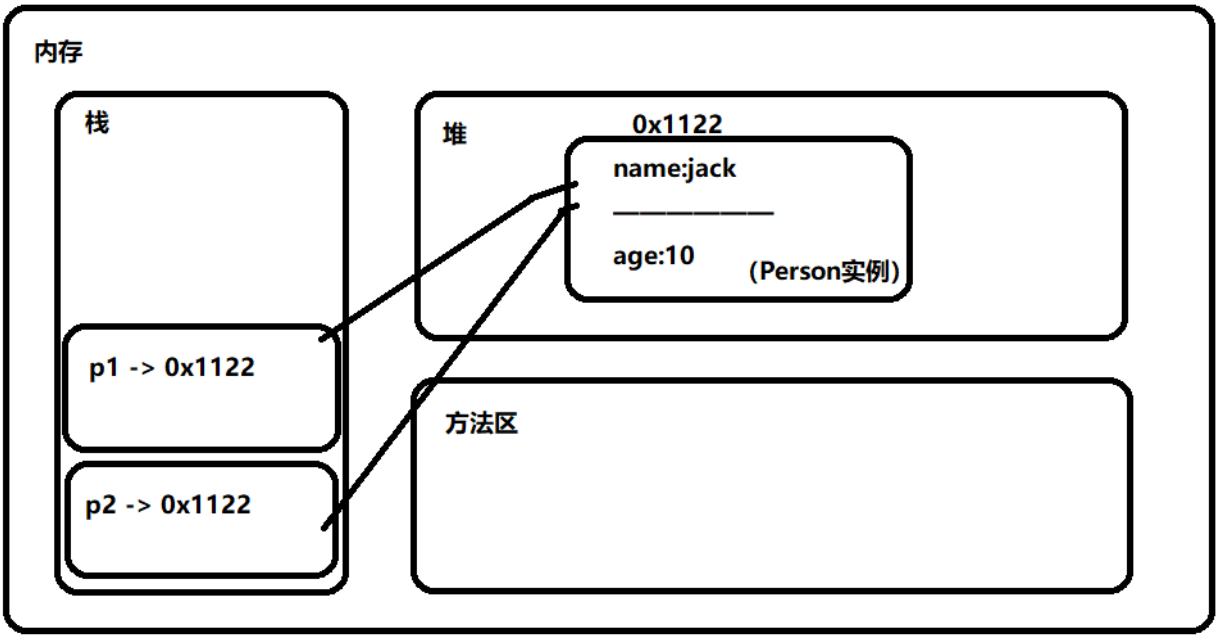
```
object ObjTest{
    def main(args:Array[String]):Unit={
        val emp = new Emp
        val emp2 : Person = new Emp
        //要将子类对象交给父类引用，需要在定义变量时声明类型
    }
}
class Person{

}
class Emp extends Person{

}
```

2.类和对象的内存分配机制

scala的内存分配机制和java一致

存在以下scala

```
def main(args:Array[String]):Unit={
    val p1 = new Person
    p1.name = "jack"
    p1.age = 30
    val p2 = p1
    //分析scala中的对象在内存中的布局
    println("p1.name=" + p1.name + "p1.hashCode=" + p1.hashCode)
    println("p2.name=" + p2.name + "p2.hashCode=" + p2.hashCode)
}
```



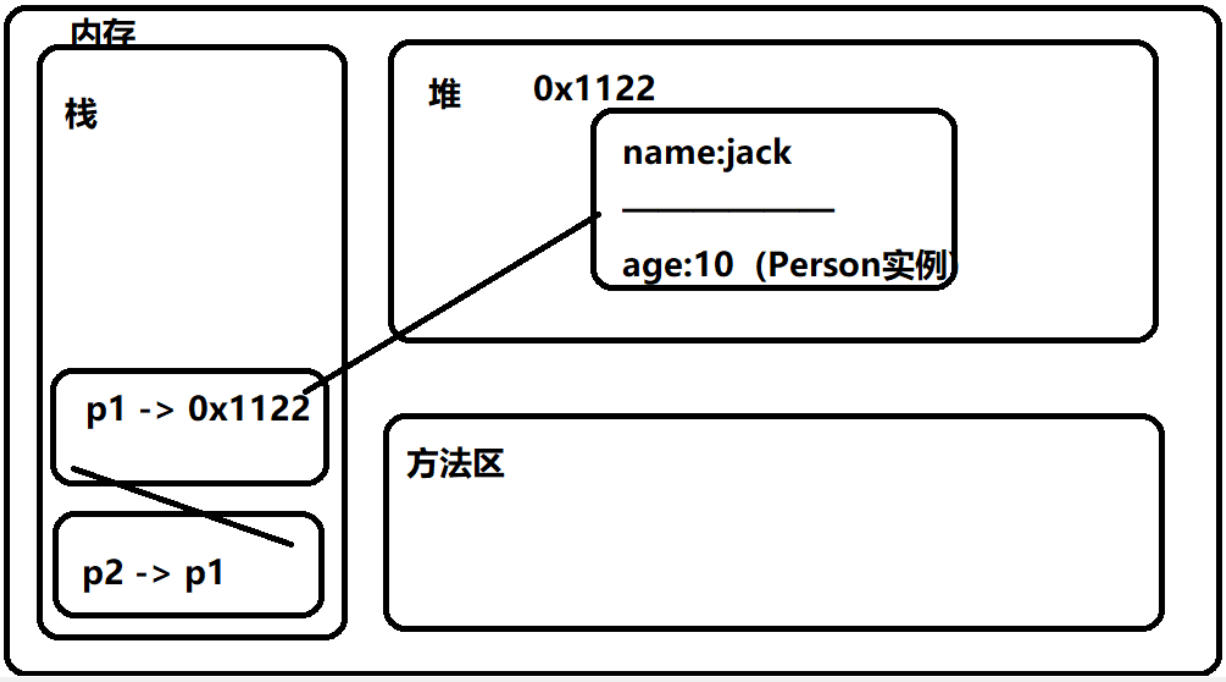
证明：

```
object ObjTest{
    def main(args:Array[String]):Unit={
        var p1 = new Emp
        var p2 : Person = p1

        p1.name = "jack"
        p1.age = 10

        println(p1 == p2)
        println(p1.name + p1.age)
        println(p2.name + p2.age)
        println(p1.hashCode())
        println(p2.hashCode())
    }
}
class Person{
    var name = ""
    var age : Int = _
}
class Emp extends Person{

}
```

结果

```
true
jack10
jack10
2114694065
2114694065
```

可见，p2 引用了 p1



2.方法

scala的方法其实就是函数，方法的声明规则与函数的声明规则一致

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

```
class Dog{
    private var sal : Double = _
    var food : String = _

    def cal(n1:Int,n2:Int):Int={
        return n1 + n2
    }
}
```

方法的调用机制原理

1.当scala开始执行时，先在栈区开辟一个main栈，main栈式最后被销毁的

2.当scala程序在执行到一个方法时，总会开辟一个新的栈

3.每个栈都是一个独立的空间，变量（基本数据类型）是独立的，互相不会影响

4.当方法执行完毕后，该方法开辟的栈就会被jvm机制回收（并不是真的被回收了，只是用不到了而已）

栈：

栈存在一个栈顶，类似于一个指针或者索引，当往上走的时候指针会跟着往上走，用完了栈后指针又会往下移动，当我们说栈被回收后，实际上就是栈顶往下移动了，以后使用不到了。当第二次开辟一个栈时，栈顶会将原先的数据进行一个覆盖。

真正的销毁栈工作不一定会做

方法练习1：

```
object MethodDemo01{
    /**
     * 编写类（MethodExec）,编程一个方法，方法不需要参数，在方法中打印一个10*8的矩形
     * 在main中调用该方法
     */
     def main(args:Array[String]):Unit={

     }
     def MethodExec():Unit={
        var i = 10
        wihle(i!=0){
            println("*"*8)
            i-=1
        }
     }
}
```

结果

```
********
********
********
********
********
********
********
********
********
********
```

方法练习2：

```
object MethodDemo{
    def main(args:Array[String]):Unit={
        /**
       * 修改上一个程序，编写一个方法中，方法不需要参数，计算该矩形的面积
       * 并将其作为方法返回值。在main方法中调用该方法，接收返回的面积值并打印
       * 结果保留小数点2位
       * @param args
       */

    }
    def MethodExec():Unit={
        val width = 8
        val height = 10
```

```
        val Mj : Double = width * height
        println(f"$Mj%.2f")
    }
}
```

结果

```
80.00
```

方法练习3：

```
object MethodDemo03{
  /**
   * 修改上一个程序，编写一个方法
   * 提供m和n两个参数，方法中打印一个m*n的矩形
   * 再编写一个方法计算该矩形的面积，将其作为返回值
   * main中调用该方法，接收返回的面积值并打印
   * @param args
   */
   def main(args:Array[String]):Unit={
    var a =
   }
    def MethodExec(m:Int,n:Int):Unit={
        var i = m
        while(i != 0){
            println("*"*n)
            i -= 1
        }
        Mj(m,n)
        def Mj(len:Int,width:Int):Unit={
            println(len*width)
        }
    }
}
```

结果

```
********
********
********
********
********
********
********
********
********
********
80
```

方法联系4：

```
object MethodDemo04{
  /**
   * 编写方法，判断一个个数odd是奇数还是偶数
   * @param args
   */
    def main(args:Array[String]):Unit={
        println(MethodExec(4))
        println(MethodExec(5))
    }
    def MethodExec(odd:Int):Boolean={
        if(odd % 2 == 0){
            true
        }else{
            false
        }
    }
}
```

```
true
flase
```

17.构造器

java构造器回顾

java构造器基本语法：

在java中，构造器的方法名必须和类名同名

构造器没有返回值

```
[修饰符]方法名(参数列表){
    构造方法体
}
```

构造器(construr)又叫构造方法，是类的一种特殊的方法。它的主要作用**是完成对新对象的初始化**

java构造器的特点：

1.java中一个类可以定义多个不同的构造方法，叫构造方法重载

2.如果程序员**没有定义构造方法**，系统会自动生成一个默认的无参构造方法，比如Person(){}

3.一旦定义了自己的构造方法，默认的构造方法(无参构造)就会被覆盖，不能再使用默认的无参构造了，除非显示地定义一次

```
```

```
//第一个无参构造：利用构造器设置所有人的属性初始值为18
//第二个带name和age两个参数的构造：使得每次创建Person对象时同时初始化对象的age属性和name属性
class Person{
    public String name;
    public int age;

    public String getInfo(){
        return name + "\t" + age;
    }
    public Person(){
        age = 18;
    }
    public Person(String name,int age){
        this.name = name;
        this.age = age;
    }
}
```

什么需求下要使用构造器：

在创建如人类的对象时，就直接指定这个对象的初始年龄和姓名，这时可以使用构造器

Scala构造器

和java一样，scala构造对象也需要调用构造方法，并且可以拥有任意个构造方法

即scala构造器也支持重载

scala类的构造器包括：主构造器、辅助构造器

Scala构造器的基本语法：

```
class 类名(形参列表){
//主构造器
    def this(形参列表){
    //辅助构造器
    }
    def this(形参列表){
    //辅助构造器
    }
    ...//辅助构造器可以拥有任意多个
}
```

1.辅助构造器 函数的名称this，可以有多个，编译器通过**不同参数**来区分

不同参数可以是类型不一样，也可以是个数不一样

```
class test(a : Int){
    def this(a:String){

    }
    def this(a:Int,b:Int){

    }
}
```

构造器的案例：

```
object ConDemo{
    def main(args:Array[String]):Unit={
        val p1 = new Person("阿甘",19)
        println(p1)
        println(p1.hashCode())
    }
}
class Person(inName:String,inAge:Int){
    var age : Int = inAge
    var name : String = inName

    override def toString : String = {
        "name = " + this.name + ",age = " + this.age
    }
    override def hashCode():Int={
        10101
    }
}
```

```
name = 阿甘,age = 19
10101
```

scala构造器的注意事项和细节：

1.scala构造器作用是完成对新对象的初始化，构造器没有返回值

2.主构造器的声明直接放置在类名之后[用于反编译]

3.主构造器会执行类定义中的所有语句（体会到scala函数式编程和面向对象编程融合到一起，即：构造器也是方法，方法也是函数）

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

```
object ConDemo02{
    def main(args:Array[String]):Unit={
        val a = new A
    }
```

```
}
class B {
    println("b--")
}
class A extends B{
    println("a--")
    def this(name:String){
        this
        pritnln("A this")
    }
}
```

```
b--
a--
```

```
object ConDemo03{
    def main(args:Array[String]):Unit={

    }
}
class B {
    println("b--")
}
class A extends B{
    println("a--")
    def this(name:String){
        this
        println("this")
    }
}
```

```
b--
a--
this
```

以此可以看出执行流程：

父类构造器 =》 主构造器 =》辅助构造器

18.属性的高级部分

构造器参数

1.scala类的主构造器的形参没有用任何修饰符修饰，那么这个参数就是**局部变量**

2.如果参数使用val关键字声明，那么scala会将参数作为类的私有的只读属性使用

3.如果参数使用var关键字声明，那么scala会将参数作为类的**成员属性**使用，并会提供属性对应的xxx()[类似getter]/xxx_$eq()[类似setter]方法，即这时的**成员属性是私有的，但是可读可写**

scalaBean属性

javaBean规范了定义Java属性，像是getXXX（）和setXXX（）的方法，许多java工具（框架）都依赖这个命名习惯，为了java的互操作性，将scala字段加@BeanProperty时，会自动生成规范的setXXX/getXXX方法，这时可以使用对象.setXXX()和对像的.getXXX()来调用属性

注意：给某个属性加入@BeanProperty注解后，会生成getXXX和setXXX的方法，并且对原来底层自动生成类似xxx(),xxx_$eq()方法，没有冲突，二者可以共存

```
import scala.beans.BeanProperty
class Car{
    @BeanProperty var name : String = null
}
```

对象创建流程：

```
class Person{
    var age : Short = 90
    var name : String = _
    def this(n : String,a:Int){
        this()
        this.name = n
        this.age = a
    }
}
var p : Person = new Person("小倩",20)
```

1.加载类信息（加载属性信息和方法信息，会加载到方法区）

2.在内存中（堆中）开辟空间，空间大小取决于属性大小

3.使用父类构造器（主构造器和辅助构造器）进行初始化

4.使用主构造器对数学进行初始化[age:90,name null]

5.使用辅助构造器对属性进行再次初始化[age:20,name "小倩"]

6.将开辟的对象的地址赋给 p 这个引用

19.面向对象三大特性

面向对象的三大特性：**封装**、**继承**和**多态**

面向对象-抽象

在定义一个类的时候，实际上就是把事物的共有属性和行为提取出来，形参一个物理模型（模板），这种**研究问题的方法称之为抽象**

即存在一个生物，不论这个生物什么，都具备有 以下属性：体重、高度、年龄等，这时候就被称作抽象

面向对象-封装

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

封装的实现步骤

1.将属性进行私有化

2.提供一个公共的set方法，用于对属性判断并赋值

```
def setXxx(参数名：类型):Unit={
    //加入数据验证的业务逻辑
    属性 = 参数名
}
```

3.提供一个公共的get方法，用于获取属性的值

```
def getXxx()[:返回类型]={
    return 属性
}
```

封装的注意事项

1.scala中为了简化代码的开发，当声明属性时，本身就自动提供了对应的setter/getter方法，如果属性声明为private的，那么自动生成的setter/getter方法也是private的，如果属性省略访问权限修饰符，那么自动生成的setter/getter方法时public的

```
class Cat(Age:Int){
    private var age : Int = 0
    var name : String = ""
    private val age2 : Int = 0
    val name2 : String = ""
}
```

2.因此我们如果只是对一个属性进行简单的set和get，只要声明一下该属性（属性使用默认访问修饰符）不用专门的getset，会默认创建，访问时，**直接对象.变量**这样也是为了保持访问一致性

3.从形式上看dog.food直接访问属性，其实底层仍然是访问的方法

4.有了上面的特性，目前很多新的框架，在进行了反射时，也支持对属性的直接反射

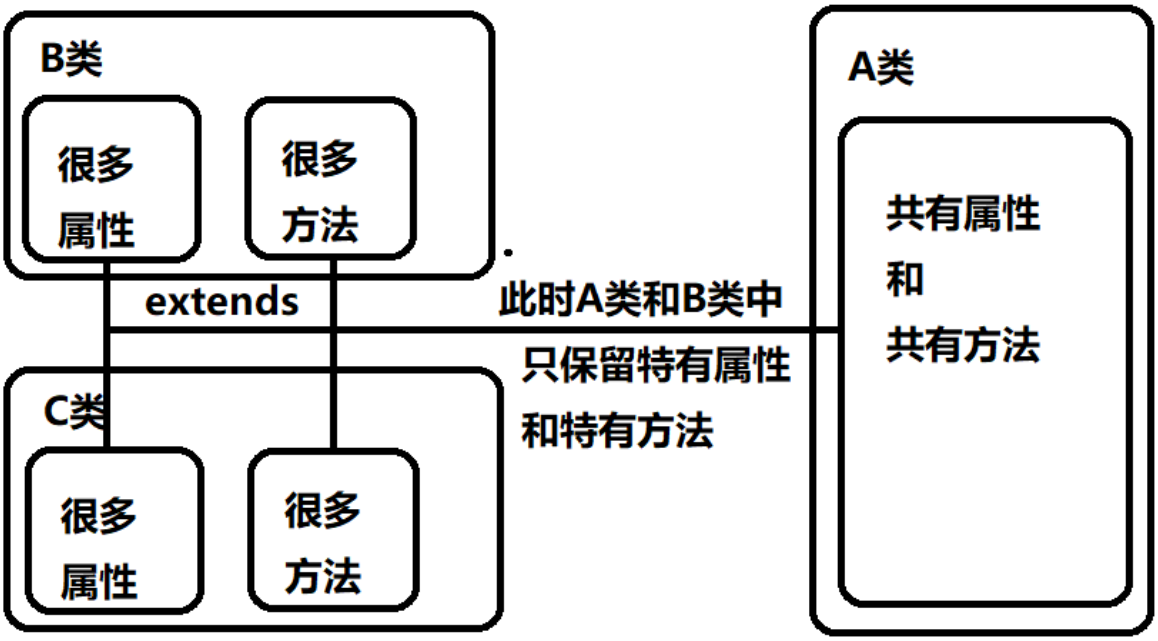
面向对象-继承

java继承回顾

```
class 子类名 extends 父类名{
    类体
}
```

继承可以解决代码的复用，当多个类存在相同的属性和方法时，可以使子类直接拿到父类相同的属性和方法，而不用重写定义，只需要通过extends语句来声明继承即可

和java一样，scala也支持单继承



scala继承

scala的继承语法和java完全一样

```
class 子类名 extends 父类名{
    类体
}
```

快速案例

```
object ExtendsDemo01{
    def main(args:Array[String]):Unit={
        val stu = new Student
        stu.studying
        stu.showInfo
    }
    class Person{
        var name : String = _
        var age : Int = _
        def showInfo() : Unit={
            println("学生姓名")
            println("名字" + this.name)
        }
    }
    class Student extends Person{
        def studying() : Unit = {
            println(this.name + "学习中")
        }
    }
}
```

```
null学习中
学生姓名
姓名null
```

scala继承带来了哪些便利?

1.代码的**复用性**提高了

2.代码的**扩展性**和**维护性**提高了

当修改父类时，对应的子类就会继承相应的方法和属性

子类继承了声明？怎么继承的?

子类继承了所有的属性，只是私有的属性不能直接访问，需要通过**公共的方法**取访问【debug代码验证时可以看到】

面向对象-方法重写

scala明确规定重写一个非抽象方法需要用**override**修饰符，调用超类的方法使用**super**关键字

```
class Person{
    var name : String = "tom"
    def printName(){
        println("Person printName()" + name)
    }
}
```

```
class Emp extends Person{
//这里需要显式的使用override
    override def printName(){
        println("Emp printName()" + name)
        super.printName()
    }
}
```

第二个类重写了父类的printName方法

面向对象-覆写字段

1.def只能重写另一个def（即：方法只能重写另一个方法）

2.val只能重写另一个val 属性或**重写不带参数的def**

```
class AAA{
    val a = _
    def b:Int{}
}
class BBB extends AAA{
    override val a = _
    override val b:Int = 1
}
```

思考：

为什么val不能重写var?

因为var拥有两个暴露的get和set方法，而val只有一个暴露的get方法，当重写后就会发生逻辑混乱，主要体现在设置是设置的var的属性，而调用则调用了val的属性，这样会造成数据的设置和数据的获取不一致

20.类型检查和转换

基本介绍

要测试某个对象是否属于某个给定的类，可以用isInstanceOf方法。用asInstanceOf方法将引用转换为子类的引用。classOf获取对象的类名

1.classOf[String]就如同 Java 的 String.class

2.obj.isInstanceOf[T] 就如同Java的 obj instanceof T判断 obj 是不是 T 类型

3.obj.asInstanceOf[T] 就如同Java的 (T)obj 将 obj 强转成 T 类型

```
//获取对象类型
println(classOf[String])
val s = "zhangsan"
println(s.getClass.getName)//这种是java中反射方式得到的类型
println(s.isInstanceOf[String])
println(s.asInstanceOf[String])//将s显示转换成String

var p = new Person2
val e = new Emp
p = e//将子类对象赋给父类
p.name = "xxx"
println(e.name)
p.asInstanceOf(Empl.sayHi())
```

类型检查和转换的最大价值在于：可以判断传入对象的类型，然后转成对应的子类对象，进行相关操作，这里也体现出多态的特点

21.超类

回顾Java中超类的构造

```
class A{
    public A(){
        System.out.println("A()");
    }
    public A(String name){
        System.out.println("A(String name)" + name);
    }
}
class B extends A{
    public B(){
        //这里会隐式调用super();就是无参的父类构造器A()
        System.out.println("B()");
    }
    public B(String name){
        super(name);
        System.out.println("B(String name)" + name);
```

```
    }
}
```

从代码可以看出：在Java中，创建子类对象时，子类的构造器总是去调用一个父类的构造器（显式或隐式调用）

java中，子构造器的辅助构造器B()里，会默认拥有一个super()，用于调用父构造器的辅助构造器。

而父构造器也有一个隐藏的super，用于调用Object，但是因为Object没有任何输出，所以不能看到

scala对应的操作如下：

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

    }
    class A{
        println("class A")//为了演示B函数会隐式调用super()
        def A():Unit={
            println("A()")
        }
        def A(name : String):Unit={
            println("A(name : String)")
        }
    }
    class B extends A{
        def B():Unit={
            println("B()")
        }
        def B(name : String):Unit={
            println("B(name:String)")
        }
    }
}
```

结果：

```
class A
B()
```

22.抽象类

基本介绍

scala中，通过abstract关键字标记不能被实例化的类。**方法不用标记abstract**，只要省略掉方法体即可

```
abstract class CCC{
    def a()
}
```

抽象类可以拥有抽象字段，抽象字段就是没有初始值的字段

```
abstract class CCC{//抽象类
    var name : String//抽象字段
    def a()//抽象方法
}
```

抽象字段和抽象方法只能在抽象类中定义

抽象类的价值更多是在于设计，是设计者设计好后，让子类继承并实现抽象类（即：实现抽象类的抽象方法）

实例化抽象类

```
object AbstractClassDetail01{
    def main(args:Array[String]):Unit={
        val animal = new Animal03{
            override def sayHello():Unit={
                println("say hello ")
            }
        }
        animal.sayHello()
    }
}

abstract class Animal03{
    def sayHello()
}
```

上列将抽象方法sayHello手动实现是可以的，但是如下列

```
object AbstractClassDetail01{
    def main(args:Array[String]):Unit={
        val animal = new Animal03
        animal.sayHello()
    }
}
abstract class Animal03{
    def sayHello()
}
```

这种就不行，无法直接实现抽象方法，因为抽象方法里没有东西让你实现

抽象类细节：

1.抽象类不能被实例化

2.抽象类不一定要包含abstract方法，也就是说，抽象类可以没有abstract方法

3.一旦类包含了抽象方法或者抽象属性，则这个类必须声明为abstract

4.抽象方法不能有主体，不允许使用abstract修饰

5.如果一个类继承了抽象类，则它必须实现抽象类的所有抽象方法和**抽象属性**，除非它自己也声明为abstract类

```
object AbstractClassDetail01{
    def main(args:Array[String]):Unit={

    }
}
abstract class Animal03{
    def sayHello()
    var a:Int
}
abstract class Temp{
    val animal = new Animal
}
```

6.**抽象方法和抽象属性**不能用private、final修饰，因为这写关键字都与**重写/实现**相违背、相冲突

7.抽象类中可以有实现的方法

8.子类重写抽象方法不需要override，写上也不会错

```
object AbstractClassDetail01{

}
abstract class Animal03{
    def sayHello()
    var a : Int
}
abstract class Temp extends Animal03{
    def sayHello():Unit{
        println("hello")
    }
    override var a : Int = 1
}
```

23.匿名子类

基本介绍

和Java一样，可以通过包含带有定义或重写的代码块的方式创建一个匿名的子类

回顾Java匿名子类的使用

```
abstract class A2{
 abstract public void cry();
}
A2 obj = new A2(){
    @Override
    public void cry(){
        System.out.println("okokk")
    }
}
```

scala的匿名子类

```
abstract class Monster{
    var name : String
    def cry()
}
var monster = new Monster{
    override var name:String = "张三"
    override def cry():Unit={
        println("啊啊啊")
    }
}
```

小结练习

```
object End01{
  /**
   * @练习1
   * 编写Computer类，包含cpu、内存、硬盘等属性，getDetails方法用于返回Computer的详细信息
   * 编写PC子类，继承Computer类，添加特有属性【品牌brand】
   * 编写NotePad子类，继承Computer类，添加特有属性【颜色color】
   * 编写Test Object，在main方法中创建PC和NotePad对象，分别为对象中特有的属性赋值，以及
   * 从Computer类继承的属性值赋值，并使用方法打印输出信息
   */
    def main(args:Array[String]):Unit={
        val pc = new PC
        pc.Cpu = "intel Core i9 12th"
        pc.Nc = "金士顿 16G"
        pc.PD = "三星 512G"
        pc.brand = "宏基"
        pc.getDetails
```

```
        val notePad = new NotePad
        notePad.Cpu = "AMD radon 540x"
        notePad.Nc = "三星 8G"
        notePad.PD = "迅捷 1T"
        notePad.color = "灰色"
        notePad.getDetail
    }
}
class Computer{
    var Cpu:String = ""
    var Nc :String = ""
    var PD :String = ""
    def getDetails():Unit{
        println("Cpu =>" + Cpu + ",Nc => " + Nc + ",PD => " + PD)
    }
}
class PC extends Computer{
    var brand = ""
    def getDetails():Unit{
        println("Cpu =>" + Cpu + ",Nc => " + Nc + ",PD => " + PD + ",brand => " +
brand)
    }
}
class NotePad extends Computer{
    var color = ""
    def getDetails():Unit{
        println("Cpu =>" + Cpu + ",Nc => " + Nc + ",PD => " + PD + ",color => " +
color)
    }
}
```

```
Cpu => intel Core i9 12th,Nc => 金士顿 16G,PD => 三星 512G,brand => 宏基
Cpu => AMD radon 540x,Nc => 三星 8G,PD => 迅捷 1T,color => 灰色
```

24.静态属性和静态方法

回顾java

```
public static 返回值类型 方法名（参数列表）{方法体}
```

java中的静态方法并不是通过对象调用的，而是通过类对象调用的，所以静态操作并不是面向对象的

scala中静态的概念-伴生对象

scala语言是完全面向对象（**万物皆对象**）的语言，所以没有静态的操作（即在scala中没有静态的概念）但是为了能够和java语言交互，就产生了一中特殊的对象来**模拟类对象**，我们称之为**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**中声明和调用

```
class ScalaPersson{
 var name : String = _
    //说明 class ScalaPerson是伴生类
}

object ScalaPersson{
    var sex:Boolean = true
    //说明 object ScalaPerson是伴生对象
}
```

当在同一个文件中，有一个class和一个object的名字相同，这时候我们称为伴生类、伴生对象

我们将非静态的内容写到伴生类中

我们将静态的内容写到伴生对象中

class即伴生类 编译后底层生成 ScalaPerson类 ScalaPerson.class

object即伴生对象 编译后底层生成 ScalaPerson$类 ScalaPerson$.class

对于伴生对象的内容，我们可以直接通过ScalaPersoon.属性或者方法

在底层里

    println(ScalaPerson.sex)等价于ScalaPerson$.MODULE$.sex()

25.Apply方法

提出疑问：

  在我们使用一些方法时，如

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

时，为什么不需要new对象就能够创建一个List呢?

回顾java中，想要达到这种效果，要么用反射机制，要么就只能new了

实际上这是因为有apply方法的存在

apply是一个方法，定义在类下，定义好后可以通过类名 + () 直接调用apply，如下

```
object ApplyDemo02{
    def apply(){
        println("apply被调用")
    }
    def main(args:Array[String]):Unit={
        ApplyDemo02()
    }
}
```

```
apply被调用
```

可见，当使用类名 + ()时，scala应该会自动扫描apply，从而达到调用的效果

apply方法与伴生实例

```
object ApplyDemo03{
    def main(args:Array[String]):Unit={
        val pig1 = new Pig("佩奇")
        val pig2 = new Pig("快速")
        val pig3 = new pig
        println(pig1.name)
        println(pig2.name)
        println(pig3().name)
    }
}
class Pig(pName:String){
    var name = pName
}
object Pig{
    def apply(pName:String):Pig = new Pig(pName)
    def apply():Pig = new Pig("匿名")
}
```

```
佩奇
快速
匿名
```

26.特质*

scala的特质就是java的接口

java接口的定义

```
interface 接口名
```

实现接口

```
class 实现类类名 implements 接口名1，接口2
```

1.在java中，一个类可以实现多个接口

2.在java中，接口之间支持多继承

3.接口中属性都是常量

4.接口中的方法都是抽象的

scala接口的介绍

从面向对象来看，接口并不属于面向对象的范畴，scala是纯面向对象的语言

在scala中没有接口

scala语言中，**采用了特质 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

```
trait 特质名{
    trait体
}
```

实现一个特质

```
object T1 extends Serializable{}
```

所有的java接口都可以当作特质来使用

```
trait Serializable extends Any with java.io.Serializable{}
```

```
object T2 extends Cloneable{}
```

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

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 ...
```

可以把特质看作是继承的一种补充

scala的继承是单继承，也就是只能有一个父类，保证了代码的纯洁性，比C++中的多继承更简洁

但对子类功能的拓展有一定影响，所以我们认为

scala引入trait特质，第一可以替代java的接口，第二也是对单继承机制的一种补充

scala提供了特质，特质可以同时拥有抽象方法和具体方法，一个类可以实现/继承多个特质

```
object TraitTest{
    def main(args:Array[String]):Unit={
        val sheep = new Sheep
        sheep.sayHi()
    }
}

trait TraitTest01{
    //抽象方法
    def sayHi()

    //实现普通方法
    def sayHello():Unit = {
        println("say Hello-")
    }
}
class Sheep extends TraitTest01{
    override def sayHi():Unit={
        println("say hi -")
    }
}
```

```
say hi -
```

当特质中只有抽象方法的时候，和java的机制是完全一致的

反编译后：

```
public class Sheep implements TraitTest01{
    public void sayHi(){
        Predef..MODULE$.println("say hi -")
    }
}
```

当一个特质中有抽象方法和非抽象方法时

1.一个特质在底层对应两个类

TraitTest01.class 接口

2.还对应 TraitTest01$class.class，会生成一个TraitTest01$class抽象类，由抽象类实现非抽象方法方法

当特质中由接口和抽象类时

class Sheep extends TraitTest01在底层对应

class Sheep implements TraitTest01

当在 Sheep 类中要使用TraitTest01的实现的方法，就通过TraitTest01$class来实现

带有特质的对象，哈可以动态混入

1.除了可以在类声明时继承特质以外，还可以在构建对象时混入特质，扩展目标类的功能

2.此种方式也可以应用于对抽象类功能进行扩展

3.动态混入是scala特有的方式（java中没有动态混入），可在不修改类声明/定义的情况下，扩展类的功能，非常的灵活，**耦合度低** ocp原则（闭合原则、修改源码关闭、扩展功能开放）

4.动态混入可以在不影响原有的继承关系的基础上，给指定的类扩展功能

5.演示：

```
object MixinDemo01{
    def main(args:Array[String]):Unit={
        val oracleDB = new OracleDB with Operate3
        oracleDB.insert(100)

        val mySQL = new MySQL with Operate3
        mySQL.insert(200)

    }
}
trait Operate3{
    def insert(id:Int):Unit={
        println("插入数据 + " + id)
    }
}
class OracleDB{

}
abstract class MySQL3{

}
```

可以见得，所谓的**动态混入**就是在**创建对象时 实现特质**

思考：如果抽象类中由抽象方法，如何动态混入？

```
object MixinDemo02{
    def main(args:Array[String]):Unit={
        val mySQL3_1 = new MySQL3_1 with Operate3{
            override def say():Unit={
                println("say")
            }
        }
    }
}
abstract class MySQL3_1{
    def say()
}
```

创建对象有几种方式

1.new 对象

2.apply 创建对象

3.匿名子类的方式创建

4.动态混入

27.叠加特质

基本介绍

构建一个对象的同时 混入多个特质，我们叫做**叠加特质**

叠加特质的**特质声明顺序从左到右**，方法执行的顺序从右到左

```
object MixinDemo02{
    def main(args:Array[String]):Unit={
        val mySQL3_1 = new MySQL3_1 with Operate3 with Operate2{
            override def say():Unit={
                println("say")
            }
        }
    }
}
abstract class MySQL3_1{
    def say()
}
trait Operate3{
    def insert(id:Int):Unit={
        println("插入数据 + " + id)
```

```
    }
}
trait Operate2{
    def delect(id:Int):Unit={
        println("删除数据 + " + id)
    }
}
```

```
插入数据 = 100
删除数据 + 200
```

叠加特质的细节：

1.特质的声明顺序从左到右

2.scala在执行叠加对象的方法时，会首先从后面的特质（从右向左）开始执行

3.scala中特质中如果调用super，并不是表示调用父特质的方法，而是向前面（左边）继续查找特质，如果找不到，才回去父特质查找

4.如果想要调用具体特质的方法，可以指定：super[特质].xxx(...).其中的泛型必须是该特质的直接超类类型

```
trait File4 extends Data4{
    println("File4")
    override def insert(id:Int):Unit={
        println("向文件")
        super[Date].insert(id)
    }
}
```

28.富接口

富接口：即该特质中既有抽象方法又有非抽象方法

```
trait Operate{
    def insert(id:Int)//抽象方法
    def pageQuery(pageno:Int,pagesize:Int):Unit={
        println("分页查询")
    }
}
```

29.自身类型特质

说明：

自身类型：主要是为了解决特质的循环依赖问题，同时可以确保特质在不扩展某个类的情况下，依然可以做到**限制混入该特质的类的类型**

举例说明自身类型特质，以及**如何使用自身类型**特质

循环依赖问题：

在一个类里，两个特质互相引用

限制混入该特质的类的类型：

设计了 self-type 限制

等于是主动告诉编译器"我是谁"

如果没有"我"，那么以后的代码不能执行

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

  }
}
trait Logger{
  this:Exception =>
  def log():Unit={
    println(getMessage)
  }
}
```

这里面，getMessage是Exception类下的，如果去掉了this:Exception，则getMessage无法使用

Logger就是自身类型特质，当这里做了自身类型后，那么等价于 trait Logger extends Exception，要求混入该特质的对象的类也是Exception的子类

```
object selfType {
  def main(args: Array[String]): Unit = {
  }
}
trait Logger{
  this:Exception =>
  def log():Unit={
    println(getMessage)
  }
}
class Console extends Exception with Logger
class Console extends Logger//报错
```

29.内部类

1.嵌套类

基本介绍

在scala中，你几乎可以在任何语法结构中内嵌任何语法结构，如在类中可以再定义一个类，这样的类是嵌套类，其他语法结构也是一样

嵌套类就**类似于java中的内部类**

java中，类共有五大成员，请说明是哪五大成员

1.属性

2.方法

3.内部类

4.构造器

5.代码块

java内部类简单回顾

在java中，一个类的内部又完整嵌套了另一个完整的类结构，被嵌套的类称之为内部类（inner class），嵌套其他类的类称为**外部类**

内部类最大的特点就是**可以直接访问私有属性**，并且可以体现类与类之间的包含关系

```
class Outer{//外部类
    class Inner{//内部类

    }
}
class Other{//外部其他类
}
```

java内部类的分类

从定义在外部类的**成员位置**上来看

1.成员内部类（没用static修饰）

2.和静态内部类（使用static修饰）

定义在外部类**局部位置**上（比如方法内）来看

1.分为局部内部类（有类名）

2.匿名内部类（没有类名）

```
public class innerClassDemo01_java{
    public static void main(String[] args){
        //创建一个外部类对象
        OuterClass outer1 = new OuterClass();
        //创建一个外部类对象
        OuterClass outer2 = new OuterClass();
        /**
         *创建java成员内部类
         *说明在java中，将成员内部类当作一个属性，因此使用下面的方式来创建outer1.new
InnerClass();
         */
        OuterClass.InnerClass ic1 = OuterClass.new InnerClass();
        OuterClass.InnerClass ic2 = OuterClass.new InnerClass();
        /**
         *下面的方法调用说明在java中，内部类只和类型相关，也就是说只要是
         * OuterClass.InnerClass 类型的对象就可以传给形参 InnerClass ic
         */
        ic1.test(ic2)
        ic2.test(ic1)

        /**
         * 创建java静态内部类
         * 因为在java中静态内部类是和类相关的，使用new OuterClass.StaticInnerClass()
         */
        OuterClass.StaticInnerClass staticInnerClass = new
OuterClass.StaticInnerClass();
    }
}
```

```
class OuterClass{//外部类
    class InnerClass{//成员内部类
        public void test(InnerClass ic){
            System.out.prinln(ic)
        }
    }
    static class StaticInnerClass{
         //静态内部类
    }
}
```

scala嵌套类的使用

```
class ScalaOuterClass{
    class scalaInnerClass{
        //成员内部类
        def innerClass(ic:ScalaInnerClass):Unit={
            println(ic)
        }
    }
}
object ScalaOuterClass{
    //伴生对象
    class scalaStaticInnerClass{
        //静态内部类
    }
}

val outer1:ScalaOuterClass = new ScalaOuterClass()
val outer2:ScalaOuterClass = new ScalaOuterClass()

//Scala创建内部类的方式和Java不一样，将new关键字放置在前，使用 对象.内部类的方式创建

val inner1 = new outer1.ScalaInnerClass()
val inner2 = new outer2.ScalaInnerClass()

inner1.innerClass(inner1)//必须为inner1
inner2.innerClass(inner2)//必须为inner2

//创建静态内部类对象
val staticInner = new ScalaOuterClass.ScalaStaticInnerClass()
println(staticInner)
```

```
外部类被调用
外部类被调用
成员内部类被调用
成员内部类被调用
test.day05.ScalaOuterClass$ScalaInnerClass@7e0babb1
test.day05.ScalaOuterClass$ScalaInnerClass@6debcae2
静态内部类被调用
```

在默认情况下，scala的内部类的实例和创建该内部类的外部对象直接关联

内部类中访问外部类的属性

方式一：使用 **外部类名.this.属性名**

```
object InnerClassDemo02{
    def main(args:Array[String]){
        val out1 = new OuterClass
        val inner1 = new out1.InnerClass

    }
}
class OuterClass {
    private val id : Int = _
    class InnerClass{
        OuterClass.this.id = 1
        println(id)
    }
}
```

30.类型投影

在scala的内部类内容中，我们知道，在一个内部类的方法被调用时，只能传入这个内部类所属的参数，那么想要传入其他内部类的参数时，就需要加上#

```
object InnerClassDemo02{
    def main(args:Array[String]){
        val out1 = new OuterClass
        val out2 = new OuterClass
        vla inner1 = new out1.InnerClass
        val inner2 = new out2.InnerClass
        inner1.test(inner2)
    }
}
class OuterClass{
    class InnerClass{
        def test(ic : InnerClass#OuterClass){
            println(ic)
        }
    }
}
```

31.隐式转换和隐式值

隐式函数基本介绍

隐式转换函数是以**implicit**关键字声明的带有**单个参数**的函数，这种函数将会**自动**应用，将值从一种类型转换为另一种类型

```
implicit def f1 (d:Double):Int={
    d.toInt
}
//Double 是输入类型，Int是转换后的类型
```

```
//反编译
private final int f1$1(double d){
    return (int)d;
}
```

隐式转换函数可以拥有多个，但不能拥有同一个形参相同、返回值相同的隐式转换函数

```
implicit def f(f:Float):Int={
    f.toInt
}
implicit def d(d:Double):Int={
    d.toInt
}
implicit def d(d:Double):Int={//报错}
```

因为当出现两个完全相同的隐式函数时，编译器并不知道要调用其中哪一个

隐式转换的注意事项和细节

1.隐式转换函数的函数名可以是任意的，**隐式转换与函数名称**无关，只与函数签名（函数参数类型和返回值类型）有关

2.隐式函数可以有多个（即：隐式函数列表），但是需要保证在当前环境下，只有一个隐式函数能够被识别

隐式转换丰富类库功能

如果需要为一个类增加一个方法，可以通过隐式转换来实现（动态增加功能）比如为MySQL类增加一个delete方法

在实际项目中，要增加新功能就意味着改变源代码，这违反了软件开发的ocp原则

在这种情况下，可以通过**隐式转换函数给类动态添加新功能**

```
object ImplicitDemo01{
    def main(args:Array[String]):Unit={
        implicit def addDelete(mysql:MySQL):DB={
            new DB
        }
        val mysql = new MySQL
        mysql.insert()
        mysql.delete()//此时myslq的delete已经可以使用了
    }
}
class MySQL{
    def insert():Unit={
        println("insert")
    }
}
class DB{
    def delete():Unit={
        println("delete")
    }
}
```

隐式值（隐式变量）

隐式值也叫**隐式变量**，将某个形参**变量标记为implicit**,所以编译器会在方法省略隐式参数的情况下去搜索作用域内的隐式值作为缺省参数

```
object ImplicitDemo02{
    def main(args:Array[String]):Unit={
        implicit val str1 : String = "jack"//隐式值//name就是隐式参数
        def hello(implicit name : String):Unit = {
            println(name + "hello")
        }
        hello
    }
}
```

```
jackhlello
```

简单理解就是

当变量和形参中都有implicit时，相同类型自动填充，即自动将变量丢进已经调用的方法中

但是要注意，当变量和形参中都有implicit时，**相同类型会自动填充**，这个操作不会在意形参名及形参个数的

```
object ImplicitDemo02{
    def main(args:Array[String]):Unit={
        implicit val str1:String = "jack"//隐式值
        //name和id就是隐式参数
        def hello(implicit name:String,id:String):Unit={
            println(name + id)
        }
        hello
    }
}
```

```
jackjack
```

在上面的案例可以看到，连id也被自动填充为jack了

而且，当存在两个隐式值时，不论方法里是不是有两个或以上形参，编译均会报错

可以得知，在默认情况下，隐式值只能有一个，且被应用于所有隐式参数

隐式类

在scala2.10后提供了隐式类，可以使用implicit声明类，隐式类的功能非常强大，同样可以扩展类的功能，比前面使用隐式转换丰富类库功能更加方便，在集合中隐式类会发挥重要的作用

隐式类使用有如下几个特点：

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，即隐式类不能时顶级的(top-level objects)

3.隐式类不能是case class

4.作用域内不能有与之相同名称的标识符

```
implicit class DB1(var m:MYSQL1){
    def addSuffix():String={
        m + "scala"
    }
}
```

```
private static final ImplicitClassDemo01$DB1$1 DB1$2(MySQL1 m){
    return new ImplicitClassDemo01$DB1$1(m)
}
```

隐式转换的机制：

```scala
object TmplicitDemo04{
    def main(args:Array[String]):Unit={
        implicit def f1(d:Double):Int={
            d.toInt
        }
    def test1(n1:Int):Unit={
        println("ok")
    }
    test(10.1)
  }
}
```

```scala
  private static final void test1$1(int n1) {
    scala.Predef$.MODULE$.println("ok");
  }
  public void main(String[] args) {
    test1$1(f1$1(10.1D));
  }
```

32.数据结构

集合

1.scala支持**可变和不可变集合**，不可变集合可以安全的并发访问

2.两个主要的包

不可变集合：scala.conllection.immutable

可变集合：scala.collection.mutable

3.scala默认不可变集合

4.scala集合有三大类，序列(Seq)，集(Set)，映射(Map)，所有集合都扩展来自Iterable特质，在Scala中集合有可变和不可变两种

Java数据结构：

```scala
public class JavaCollection{
    public static void main(String[] args){
        //不可变集合类似java的数组
        int[] nums = new int[3];
        nums[2] = 11;
```

```
        String[] names = {"bj","sh"};
        System.out.println(nums + " " + names);

        //可变集合举例
        ArrayList a1 = new ArrayList<String>();
        a1.add("zs");
        a1.add("zs2");
        System.out.println(a1 + " " + a1.hashCode());
        a1.add("zs3");
        System.out.println(a1 + " " + a1.hashCode());
    }
}
```

```
[I@511d50c0 [Ljava.lang.String;@60e53b93
[zs, zs2] 242625
[zs, zs2, zs3] 7642233
```

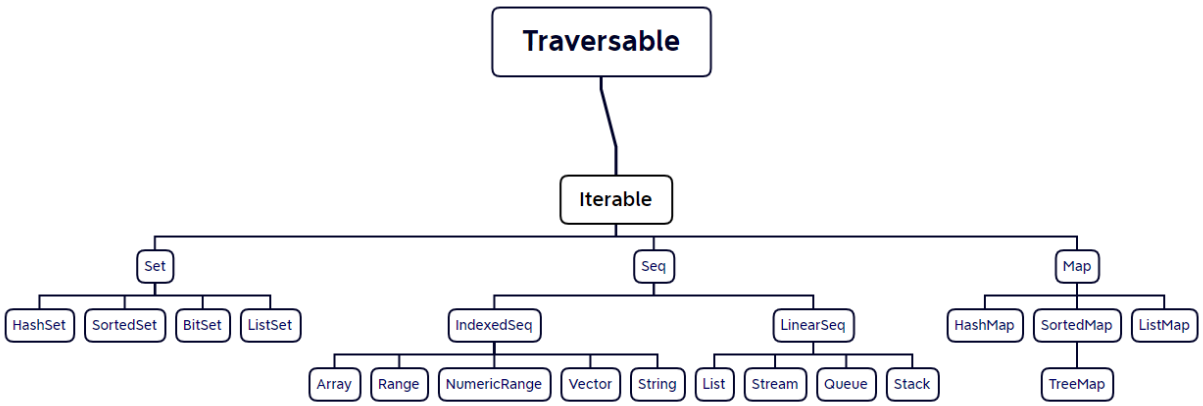
不可变：**大小不可变，内容可变**（改变大小会发生越界错误），**本身不可变**（所有的运算只是返回值，本身完全不进行变化）

可变：**大小可变，内容可变，本身也是可以动态变化**

当内容更改后地址也会发生变化

说明其实在底层，原内容并没有被改变，而是创建一个新实例，然后将原内容和新增的内容复制过去，最后更改指向为新实例

scala不可变集合继承图：



小解：

1.Set、Map是Java中也有的集合

2.Seq是java没有的，而List被归属到Seq中去了，因此这里的List和java的List不一样

3.for循环中的1 to 3 就是IndexedSeq下的Vector

4.String也属于IndexedSeq

5.这里经典的数据结构比如Queue和Stack被归属到LinerSeq中去了

6.Map体系中有一个SortedMap，说明Scala的Map可以支持排序

7.IndexMap和LinearSeq 的区别

[IndexSeq]是通过索引来查找定位的，因此速度较快，比如String就是一个索引集合，通过索引进行定位

[LinearSeq]是线性的，即有头有尾的概念，这种数据结构一般通过遍历查找，他的价值在于应用到一些具体的应用场景

scala中，核心在Seq(序列)

Seq（序列）最大的特点就是有序的

在LinearSeq（线性序列）中，应用较多的是Queue（队列）和Stack（栈）

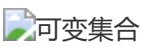
而IndexSeq（索引序列）中，应用较多的是Range（范围）、Array（数组）、String（字符串）、Vector（向量）和NumericRange（数值范围）

访问时，使用IndexedSeq（索引序列）较快

而LinearSeq较慢，因为底层是使用链表实现的，而hash则是数值加链表的形式，所以hash更快

```
object CollectionDemo01{
    def main(args:Array[String]):Unit={
        vla str = "hello"//字符串在scala中就是一个char的集合 IndexedSeq
        for(item <- str){
            println(item)
        }
    }
}
```

可变集合：



可变集合比不可变集合更加丰富

在Seq集合中，增加了 **Buffer** 集合，将来开发中，常用的就有ArrayBuffer和ListBuffer

如果涉及到线程安全，可以选择使用 **Synchronized** 类型的集合

Array（数组）：

定长数组（声明泛型）

定义数组：

```
val arr01 = new Array[Int](4)
println(arr01.lenth)

println("arr01(0) = " + arr01(0))
for(i <- arr01){
    println(i)
}

arr01(3) = 10
```

可以见得，数组（Array）在定义时，需要需要new对象，且需要指定泛型，也需要指定数组长度（超长会越界）

在更改值时，在数组括号内输入下标即可修改/访问相应位置的值

```
object ArrayDemo01{
    def main(args:Array[String]):Unit={
        val arr = new Array[Int](5)
        arr(0) = 1
        println(arr(0))
        for(i<-arr){
            println(i)
        }
    }
}
```

```
1
1
0
0
0
0
```

1.创建了一个Array对象

2.[Int]表示泛型，即该数组中只能存储Int

3.[Any]表示可以存储任何类型

4.没有赋值的情况下，默认值为0

5.数组长度不可变，但内容可以修改

赋值数组

在定义数组时直接赋值

这里使用了apply的特性

```
var arr02 = Array(1,3,"hello")
for(i <- arr02){
println(i)
}
```

```
objet ArrayDemo02{
    def main(args:Array[String]):Unit={
        val arr = Array(1,3,"hello")
        for(i <- arr){
            println(i)
        }
    }
}
```

```
1
3
hello
```

通过下标遍历：

```
object ArrayDemo03{
    def main(args:Array[String]):Unit={
        val arr = Array(1,3,"hello","bye")
        for(i <- 0 until(arr.length)){
            println(arr(i))
        }
    }
}
```

```
1
3
hello
bye
```

变长数组（声明泛型）

变长数组是可变集合

所以我们使用ArrayBuffer[Int]

```
val arr2 = ArrayBuffer[Int]()
//追加元素
arr2.append(7)
```

```
object ArrayBufferDemo01{
    def main(args:Array[String]):Unit={
        val arr = ArrayBuffer[Int]()
        arr.append(7)
        arr.append(9)
        for(i <- arr){
            println(i)
        }
    }
}
```

```
7
9
```

可变集合流程（创建、查询、修改、删除）

集合一般只能在两个包里：import scala.collection.immutable、import scala.collection.mutable

除了append（添加）以外，还有大量其他方法

arr(0)：修改数组

```
arr(0) = 1
```

append：向末尾追加元素（可多个）

```
arr.append(1)
arr.append(1,2,3,4)
```

clear：清空所有元素

```
arr.clear
```

clone：完全拷贝当前数组，并可以将拷贝后的数组赋给新变量

```
val a = arr.clone
```

appendAll：将一个数组内的所有元素完全追加进来

```
arr.appendAll(brr)
//是将brr追加进arr
```

drop：返回删除第n个元素的集合，改变集合本身

```
arr.drop(1)
```

remove：删除目标下标的元素，这个行为操作数组本身

```
arr.remove(1)
```

dropRight：从右边开始计算个数的删除

```
arr.dropRight(1)
```

dropWihle：循环删除直到不满足条件

```
arr.dropWihle( _ < 3)//删除开头下标在小于3的所有值
```

distinct：去重

```
arr.distinct
```

init：除了集合中最后一个元素外，其他的都正常输出

```
arr.init
```

inits：包括最后一个元素，全部以ArrayBuffer()的形式增量输出

```
arr.inits
```

```
arr2 = ArrayBuffer(7, 1, 1, 9, 1, 2, 3)
arr2 = ArrayBuffer(7, 1, 1, 9, 1, 2)
arr2 = ArrayBuffer(7, 1, 1, 9, 1)
arr2 = ArrayBuffer(7, 1, 1, 9)
arr2 = ArrayBuffer(7, 1, 1)
arr2 = ArrayBuffer(7, 1)
arr2 = ArrayBuffer(7)
arr2 = ArrayBuffer()
```

copyToArray：将数组缓冲区内的元素复制到另一个数组中

这个方法有三个重载的方法：

```
copyToAray(arr:Array[A]):Unit
```

```
copyToArray(arr:Array[A],start:Int):Unit
```

```
copyToArray(arr:Array[A],start:Int,len:Int):Unit
```

```
arr.copyToArray(brr,0,1)
//将arr的内容拷贝到brr中，从0开始，长度为1
```

combinations：生成一个长度为(n: Int)的新数组，这个数组会将原来的数组全部组合都存储

例如存在一个数组

```
val arr = ArrayBuffer[Int](7,1,1,8)
for (i <- arr.combinations(2)){
    println(i)
}
```

```
ArrayBuffer(7, 1)
ArrayBuffer(7, 8)
ArrayBuffer(1, 1)
ArrayBuffer(1, 8)
```

ensuring：如果符合条件判断，则正常输出，否则抛出assertion failed异常

```
val arr = ArrayBuffer[Int](1,2)
println(arr.length.ensuring(_ > 1))//正常输出1，2
println(arr.length.encuring(_ > 3))//抛出异常assertion failed
```

diff：排除掉集合中与目标集合中一样的内容（一次仅排除靠前的一例）

```
val arr = ArrayBuffer[Int](7,1,1,8)
val brr = Array(1,7,3)
println(arr.diff(brr))
```

```
ArrayBuffer(1, 8)
```

filter：排除掉不满足条件的元素

```
val arr = ArrayBuffer[Int](7,1,1,8)
arr.filter(_ > 2)
```

```
ArrayBuffer(7,8)
```

变长和定长数组互相转换

```
arr1.toBuffer//定长转可变数组
arr2.toArray//可变数组转定长
```

1.arr2.toArray返回的结果才是一个定长数组，arr2本身不变

2.arr1.toBuffer返回结果才是一个可变数组，arr1本身不变

```
object ArrayToBufferOrRan{
    def main(args:Array[String]):Unit={
        val arr1 = Array(1,2,3)
        val arr2 = ArrayBuffer[Int](1,2,3)

        val arr1_1 = arr1.toBuffer
        val arr2_1 = arr2.toArray

        for(i <- arr1){
            println("arr1 => " + i)
        }
        for(j <- arr2){
            println("arr2 => " +j)
        }
        for(n <- arr1_1){
            println("arr1_1 => " + n)
        }
        for(m <- arr2_1){
            println("arr2_1 => " + m)
        }
    }
}
```

```
arr1 => 1
arr1 => 2
arr1 => 3
arr2 => 1
arr2 => 2
arr2 => 3
arr1.toBuffer => 1
arr1.toBuffer => 2
arr1.toBuffer => 3
arr2.toArray => 1
arr2.toArray => 2
arr2.toArray => 3
```

toArray是如何实现的?

```
def toArray[B >: A ClassTag]:Array[B] = {
    if(isTraversableAgain){
        val result = new Array[B](size)
        copyToArray(result,0)
        result
    }
    else toBuffer.toArray
}
```

实际上就是new了一个新的Array，然后把原来的数组从下标为0开始拷贝进新数组里

Tuple（元组）

基本介绍

元组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，就是多个无关的数据封装成一个整体称为元组

但是元组中最大只能由22个元素

这是受限于构造元组的方法所限，以下是部分元组Function

```
Tuple1
Tuple2
Tuple3
...
```

也就是说，如果存在只有一个元素的元组，那么本质上是调用了Tuple1这个方法，元素则作为形参被转入而已

同样的，当拥有21个元素，那么就是调用了Tuple21这个方法，元素被作为形参传入，所以最大22个元素是因为元素方法最多只有Tuple22

那么如何定义元组呢？依据上面的理论，我们可以得知，元组无非就是一个类，而我们想要使用元组，就和调用类一样就行

```
object TupleDemo01{
    def main(args:Array[String]):Unit={
        val tuple = Tuple2(1,2)
        println(tuple)
    }
}
```

```
(1,2)
```

进入Tuple2中

```
final case class Tuple2[@specialized(Int, Long, Double, Char, Boolean)
+T1,@specialized(Int, Long, Double, Char, Boolean) +T2](_1: T1, _2: T2)
  extends Product2[T1, T2]{
  override def toString() = "(" + _1 + "," + _2 + ")"
  def swap: Tuple2[T2,T1] = Tuple2(_2, _1)

}
```

可以看到，Tuple2重写了toString，这使得其使用println打印时会将形参列表（元素）分割，然后用“（”作为前缀，每两个形参（元素）之间使用“，”分割，最后使用“）”作为后缀结尾

再回头看形参，两个形参被@specialized修饰，意味优化的，将输入的内容进行优化处理，输入的类型包括Int、Long、Double、Char、Boolean的子类（协变类）

现在可以总结：

Tuple接收类型，最后通过toString进行返回

```
public void main(String[] args) {
    Tuple2.mcII.sp sp = new Tuple2.mcII.sp(1, 2);
    scala.Predef$.MODULE$.println(sp);
  }
```

在反编译后可以看到，本质上是new了一个Tuple2，这是前面所述的又一证明

Tuple还有另一种写法：

```
val tuple1 = (1,2,3,"1")
println(a)
//val tuple1:(Int,Int,Int,String)
```

这种定义方式也能顺利通过，乍一看好像和Tuple4没有关系，有关系的只是类型是4个，但进行反编译后可以看到

```
  public void main(String[] args) {
    Tuple4 a = new Tuple4(BoxesRunTime.boxToInteger(1),
BoxesRunTime.boxToInteger(2), BoxesRunTime.boxToInteger(3), "1");
    scala.Predef$.MODULE$.println(a);
  }
```

它还是使用的Tuple

以此还可以看到一个现象：

当传入的元素只有单一类型时和传入的元素有多个类型时，编译方式也不一样

元组的访问

看到源码，可以发现swap方法，实际上这也是获取元组的元素的方法

即

```
tuple._1
```

```
//顺序号访问
object TupleDemo02{
    def main(args:Array[String]):Unit={
        val tuple = Tuple2(1,"你好")
        println(tuple._2)
    }
}
```

```
你好
```

元组的顺序号访问从1开始，元组的索引访问从0开始

```
//索引访问
object TupleDemo02{
    def main(args:Array[String]):Unit={
        val tuple = Tuple2(1,"你好")
        println(tuple.productElement(1))
    }
}
```

```
你好
```

元组的遍历不能使用传统的遍历，需要使用元组的迭代器

```
for(item <- tuple.productIterator){
    println("item=" + item)
}
```

```
1
2
```

List（列表）

Scala 中的 List 和 Java List 不一样，在Java中List是一个接口，真正存放数据的是ArrayList，而 Scala 的 List 可以直接存放数据，就是一个 Object，默认情况下 Scala的 List不可变，List属于序列Seq

```
val List = scala.collection.immtable.Liss
object List extends SeqFactory[List]
```

创建list

```
val list01 = List(1,2,3)//创建时直接分配元素
println(list01)
vla list02 = Nil//空列表
println(list02)
```

```
List(1, 2, 3)
List()
```

遍历list

```
for(i <- list01){
    println(i)
}
```

```
1
2
3
```

获取目标下标的值（列表的下标也是从0开始）

```
println(list01(1))
```

```
2
```

list本身无法被修改，想要新增数据、删除数据，都需要返回一个新list

修改

```
println(list01.updated(1,6))//索引从0开始
```

```
List(1, 6, 3)
```

删除:删除n个以前的所有值，然后将剩余的值返回为一个新列表

```
println(list01.drop(1))//索引从1开始
```

```
List(2, 3)
```

追加（末尾）

```
println(list01 :+ 5)
```

```
List(1, 2, 3, 5)
```

添加（开头）

```
println(5 +: list01)
```

```
List(5, 1, 2, 3)
```

将元素添加到头部

```
println(5 :: list01)
```

```
List(5, 1, 2, 3)
```

连接两个列表

```
println(list03 ::: list01)
```

```
List(5, 5, 6, 1, 2, 3)
```

连接两个列表2

```
println(list01 ++ list03)
```

```
List(1, 2, 3, 5, 5, 6)
```

list嵌套

```
println(list01 :: Nil)
```

```
List(List(1, 2, 3))
```

```
println(4 :: 5 :: 6 :: list01 :: Nil)
```

```
List(4, 5, 6, List(1, 2, 3))
```

1.::表示向集合中新建集合、添加元素

2.运算时，集合对象一定要放置在最右边

3.运算规则：从左向右

4.:::运算符表示将集合中的每个元素加入到空集合中去

Queue（队列）

1.队列是一个有序列表，在底层可以使用数组或者是链表来实现的

2.其输入和输出要准寻先入先出的原则（和栈相反），即：先存入的先取出

3.scala中，设计者直接提供队列类型使用

4.scala中，可变集合和不可变集合都有Queue，但一般来讲，我们通常使用可变集合的队列

队列的创建与追加

```
val que = Queue[int](1,2,3)
que += 1
println(que)
```

```
Queue(1, 2, 3, 1)
```

出队（从Queue的头部取出一个元素，类似于弹出，会导致Queue本身的变化）

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

```
Queue = 1
```

入队（从Queue的尾部插入一个元素，类似于插入，会导致Queue本身的变化）

```
println(que.enqueue(3))
```

```
Queue(1, 2, 3, 3)
```

返回队首（第一个元素）的元素

```
println(que.head)
```

```
1
```

返回队尾（最后一个元素）的元素

```
println(que.last)
```

```
3
```

返回队列的尾部（除了第一个以外的所有元素）

```
println(que.tail)
```

```
Queue(2,3)
```

```
println(que.tail.tail)
```

```
Queue(3)
```

Map

java的Map回顾

HashMap 是一个散列表，他存储的内容是键值对(k-v)映射，Java中的HashMap是无序的

```
public class TestJavaMap{
    public static void main(String[] args){
        HashMap<String,Integer> him = new HashMap();
        him.put("no1",100);
        him.put("no2",200);
        System.out.println(him);
        System.out.println(him.get("no2"));
    }
}
```

```
{no2=200, no1=100}
200
```

scala的map和java的map很类似，也是一个散列表，它存储的内容也是键值对（k-v）映射，scala中**不可变的map是有序的，可变的map是无序的**

map中元素是Tuple2类型

默认情况下。map是不可变map

不可变map：

```
val map1 = Map("Alice" -> 10,"Bob" -> 20)
```

注意：不可变map不能修改值、增加键值对、删除键值对等

可变map：

```
val map2 = mutable.Map("Alice" -> 10)
```

可变map——新增键值对

put

```
val map = mutable.Map("Alice" ->  40,"Bob" -> 20)
map.put("Sel",90)
println(map)
```

```
Map(Alice -> 40, Bob -> 20, Sel -> 90)
```

新增键值对2

```
map("new") = 30
println(map)
```

```
Map(Bob -> 20, Alice -> 10, new -> 30)
```

新增键值对3

```
map += ("EE" -> 1,"FF" -> 3)
println(map)
```

```
Map(Bob -> 20, EE -> 1, Alice -> 10, FF -> 3, new -> 30)
```

不可变map默认有序

可变map输出和声明顺序不一致

创建空的map

```
val map3 = new scala.collection.mutable.HashMap[String,Int]
println(map3)
```

```
Map()
```

map取值

```
println(map1("Alice"))
```

```
10
```

1.如果key存在，则返回对应的值

2.如果key不存在，则抛出异常

3.在java中，如果key不存在则返回null

contains：对目标key进行判断，存在则true，否则为false

```
println(map1.contains("Alice"))
```

```
true
```

get：获取key的value

```
println(map1.get("Alice"))
println(map1.get("Alice").get)
```

```
Some(10)
10
```

getOrElse：对获取的目标key进行判断，存在则返回key对应的value，否则返回第二个参数（可以自定义返回值）

```
println(map1.getOrElse("a","不存在"))
println(map1.getOrElse("Alice","不存在"))
```

```
不存在
90
```

遍历

遍历key：三种方式

```
prinltn(map1.keys)
println(map1.keySet)
for(i <- map1.keysIterator){
    println(i)
}
```

```
Set(Bob, EE, Alice, FF, new)
Bob
EE
Alice
FF
new
Set(Bob, EE, Alice, FF, new)
```

遍历value：两种方式

```
println(map1.values)
println(map1.valuesIterator)
```

```
HashMap(20, 1, 10, 3, 30)
<iterator>
```

Set（集）

java中set回顾

```
HashSet hs = new HashSet<String>();
hs.add("hello")
hs.add("bye")
```

java中，HashSet是实现Set接口实现的一个实体类，数据是以hash表的形式存放的，里面的**不能包含重复数据**，Set接口是一种不包含重复元素的collection，HashSet中的数据源也是没有顺序的

scala中的set

默认情况下，scala使用的是不可变集合，如果想要用可变集合，需要引用scala.collection.mutable.Set

Set不可变集合的创建

```
val set = Set(1,2,3)
println(set)
```

Set可变集合的创建

```
import scala.collecion.mutable.Set
val mutableSet = Set(1,2,3)
```

可变集合添加数据

```
setd1.add(5)
```

可变集合修改数据

由于set的值不可重复、无序，且set没有下标索引或者key进行限制，所以set不需要有更新的方法，如果需要达到更新的目的，只需要删除原来的，新增一个元素即可

Synchronized（线程安全的集合）

带有Synchronized就是带有线程安全的集合，一般操作和其他不可变线程差不多

并行集合

对一个集合进行多线程操作

```
(1 to 5).par.foreach(println(_))
```

.par就是为当前操作添加一个线程

32.match（模式匹配）

scala中没有java中的switch，但有类似的，且更加强大

模式匹配中，采用match关键字声明，每个分支使用case关键字声明，需要匹配时，从第一个case开始，如果匹配成功，则执行对应的代码，否则继续执行下一个分支进行判断，如果所有的case都不匹配，则执行case_，类似于java中的default

```
//java switch
int i = 1;
switch(i){
    case 0:
        break;
    case 1:
        break;
    default:
        break;
}
```

```
//scala match
val oper="*"
val n1 = 20
val n2 = 10
var res = 0
oper match{
    case '+' => res = n1 + n2
    case '-' => res = n1 - n2
    case '*' => res = n1 * n2
    case '/' => res = n1 / n2
    case _ => println("oper error")
}
println("res = " + res)
```

```
200
```

33.类型约束

上界(upper bounds)

java中的上界：
在java泛型里表示某个类型是A类型的子类型，使用extends关键字，这种形式叫做 upper bounds （上限或上界），语法如下

```
<T extends A>
    //T是A的子类型
//或用通配符的形式
<? extends A>
```

scala中的上界：

在scala里表示某个类型是A类型的子类型，也称为上界或上限，使用 <: 关键字，语法如下

```
[T <: A]
//或使用通配符
[_ <: A]
```

上界应用案例

1.定编写一个通用的类，可以进行Int之间、Float之间等实现了Comparable接口的值直接的比较

2.分别使用传统方法和上界的方式来完成，体会上界使用的好处

```
//传统方法
class CompareInt(n1:Int,n2:Int){
    def greater = if(n1 > n2)n1 else n2
}
//上界方法
calss CompareComm[T <: Comparable[T]](obj1:T,obj2:T){
    def greater = if(obj1.compareTo(obj2)>0) obj1 else obj2
}
```

```
val compareComm = new CompareComm[Integer](1,3)
//由于传入的类型是Comparable(Java的接口)的子类型，所以不能使用Int等AnyVal的类型
println(compareComm.greater)
```

```
3
```

Comparable类型：

Integer

java.lang.Float等

下界（lower bounds）

java泛型里表示某个类型是A类型的父类型，使用super关键字

```
<T super A>
//或用通配符表示
<? super A>
```

scala中下界

在scala中的下界或下限，使用 >: 关键字，语法如下：

```
[T >: A]
//或用通配符
[_ >: A]
```

视图界定（view bounds）

scala中，视图界定的关键字是 <% ，它比 <: 适用的范围更广，除了所有的子类型，还允许使用隐式转换类型

```
def method [A <% B](arglist):R = ...
//等价于
def method [A](arglist)(implicit viewAB:A => B):R = ...
//或等价于
implicit def conver(a:A):B = ...
```

<% 除了方法使用外，class声明类型参数时也可以使用

```
class A[T <% Int]
```

视图界定案例

```
object ViewBoundsDemo{
    def main(args:Array[String]):Unit={
        vla compareComm1 = new CompareComm(20,30)
        println(compareComm1.greater)
        //同时也支持前面学习过的上界使用的各种方式
    }
}
class CompareComm[T <% Comparable[T]](obj1:T,obj2:T){
    def greater = if(obj1.compareTo(obj2) > 0)obj1 else obj2
}
```

上下文界定（Context bounds）

和view bounds一样，context bounds（上下文界定）也是隐式参数的语法糖。

为语法上的方便，引入了“上下文界定”这个概念

上下文界定应用实例

```
//方式1
class CompareCommg[T:Ordering](obj1:T,obj2:T)(implicit comparetor:Ordering[T]){
    def greatter = if(comparetor.compare(obj1,obj2) > 0) obj1.else obj2
}

//方式2
class CompareComm9[T:Ordering](o1:T,o2:T){
    def greatter = {
        def f1(implicit cmptor:Ordering[T]) = cmptor.compare(o1,o2)
        if(f1 > 0)o1 else o2
    }
}

//方式3：使用implicitly语法糖，最简单（推荐使用）
class CompareComm[T:Ordering](o1:T,o2:T){
    def greatter = {
        //这句话就是会发生隐式转换，获取到隐式转换值personComparetor
        val comparetor = limplicity[Ordering[T]]
        println("CompareComm comparetor" + comparetor.hashCode())
        if(comparetor.compare(o1,o2) > 0) o1 else o2
    }
}

//一个普通的Person类
class Person(vla name:String,val age:Int){
    override def toString = this.name + "\t" + this.age
}
```

34.协变、逆变和不变

1.scala的协变（+），逆变（-），协变covariant、逆变contravariant、不可变invariant

2.对于一个带类型参数的类型，比如List[T]，如果对A及其子类型B，满足List[B]也符合List[A]的子类型，那么就称为**covariance（协变）**，如果List[A]时List[B]的子类型，即与原来的父子关系相反，则称为**contravariance（逆变）**。如果一个类型支持**协变或逆变**，那么这个类型就被称为variance（可变或变型），否则称为**invariance（不变型）**

3.在Java里，泛型类型都是invariant，比如List并不是List的子类型。而在scala支持，可以在定义类型的时候声明（用加号（+）表示协变，减号（-）表示逆变

如：trait List[+T]//在类型定义时声明为协变这样会把List[String]作为List[Any]的子类型

应用实例：

在这里引入关于符号说明，在声明scala 的泛型类型时，“+”表示协变，而“-”表示逆变

C[+T]：如果A是B的子类，那么C[A]是C[B]的子类，称为协变

C[-T]：如果A是B的子类，那么C[B]是C[A]的子类，称为逆变

C[T]：无论A和B是什么关系，C[A]和C[B]没有从属关系，称为不变

```
//不变情况下，只有左右类型相同才行
val t:Temp[Super] = new Temp[Sub]("hello 协变")
val t:Temp[Sub] = new Temp[Super]("hello 逆变")
val t:Temp[Sub] = new Temp[Sub]("hello 不变")


class Temp1[+A](titile:String){
    override def toString:String={
        title
    }
}

class Temp2[-A](title:String){
    override def toString:String={
        titile
    }
}

class Temp3[A](title:String){//Temp3[+A]//Temp[-A]
    override def toString:String={
        title
    }
}


 //支持协变
class Super
class Sub extends Super
```