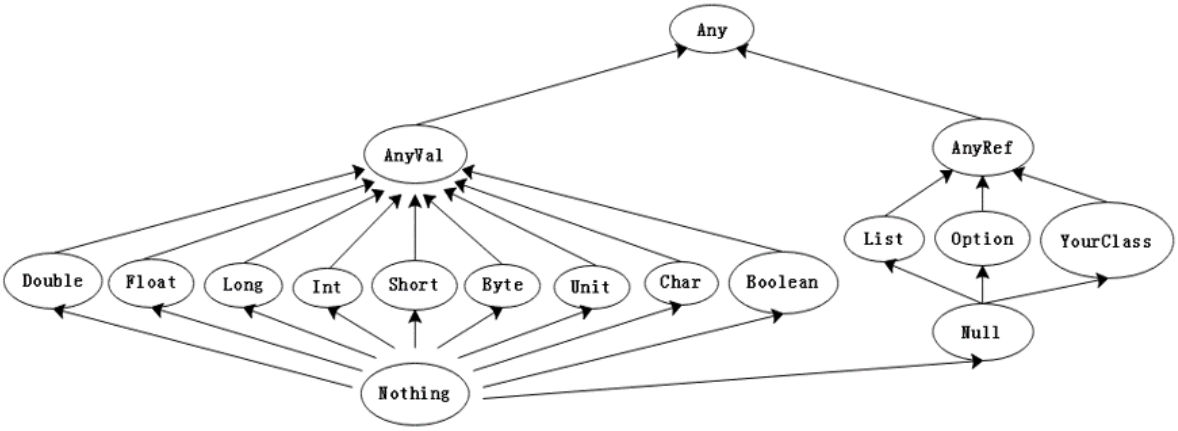
Scala程序设计

一、Scala基础

1.1 数据类型



Byte	**8位有符号补码整数。数值区间为 -128 到 127**
Short	16位有符号补码整数。数值区间为 -32768 到 32767
Int	32位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long	64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807
Float	32 位, IEEE 754 标准的单精度浮点数
Double	64 位 IEEE 754 标准的双精度浮点数
Char	16位无符号Unicode字符, 区间值为 U+0000 到 U+FFFF
String	字符序列
Boolean	true或false
Unit	表示无值，和其他语言中void等同。用作不返回任何结果的方法的结果类型。Unit只有一个实例值，写成()。
Null	null 或空引用
Nil	长度为0的List
None	Option的两个子类之一，另一个是Some，用于安全的函数返回值
Nothing	Nothing类型在Scala的类层级的最底端；它是任何其他类型的子类型。
Any	Any是所有其他类的超类
AnyRef	AnyRef类是Scala里所有引用类(reference class)的基类

比较特殊的None，是Option的两个子类之一，另一个是Some，用于安全的函数返回值。

scala推荐在可能返回空的方法使用Option[X]作为返回类型。如果有值就返回Some[X],否则返回None

```
def get(key: A): Option[B] = {
  if (contains(key))
    Some(getValue(key))
  else
    None
}
```

1.2变量和常量的声明

var修饰变量，val修饰常量

定义变量或常量的时候，也可以写上返回的类型，一般省略

```
var a  = 10 // 变量a = 10，自动寻找整型类型
var b: Int = 10 // 变量b = 10，声明整型类型
val pi = 3.1415926  // 常量pi = 3.1415926
```

如果在没有指明数据类型的情况下声明变量或常量必须要给出其初始值，否则将会报错。

scala还可以同时多个变量声明

```
val xmax, ymax = 100  // xmax, ymax都声明为100
```

1.3 scala运算符

一个运算符是一个符号，用于告诉编译器来执行指定的数学运算和逻辑运算。

Scala 含有丰富的内置运算符，包括以下几种类型：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符（选学）
- 赋值运算符

1.3.1 算术运算符

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

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
```

```
        var c = 25;
        var d = 25;
        println("a + b = " + (a + b) );
        println("a - b = " + (a - b) );
        println("a * b = " + (a * b) );
        println("b / a = " + (b / a) );
        println("b % a = " + (b % a) );
        println("c % a = " + (c % a) );

    }
}
```

1.3.2比较运算符

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

1.3.3逻辑运算符

运算符	描述	实例
&&	逻辑与	(A && B) 运算结果为 false
||	逻辑或	(A || B) 运算结果为 true
!	逻辑非	!(A && B) 运算结果为 true

1.3.4赋值运算符

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

1.4 类与对象

1.4.1 类声明

用class关键字创建类

```
// 创建Person类
class Person(){
  var name = “zhangsan”
  var age = 20
}
```

1.4.2 创建对象

用new关键字创建对象

```
/**
*class默认实现了gettter和setter方法
**/
object Test {
  def main(args: Array[String]): Unit = {
    var zhangsan = new Person
    println(zhangsan.name + ":" + zhangsan.age) //class默认实现了gettter和setter方
法
  }
}
```

1.4.2 构造器

①默认构造器

class中如果有参数的传入，那么这个构造器就是这个类的默认构造

```
// class中如果有参数的传入，那么这个构造器就是这个类的默认构造
class Person(xname: String, xage: Int) {
  var name = xname
  var age = xage
}

object Test {
  def main(args: Array[String]): Unit = {
    var zhangsan = new Person("zhangsan",20)
    println(zhangsan.name + ":" + zhangsan.age)
  }

}
```

②重写构造器

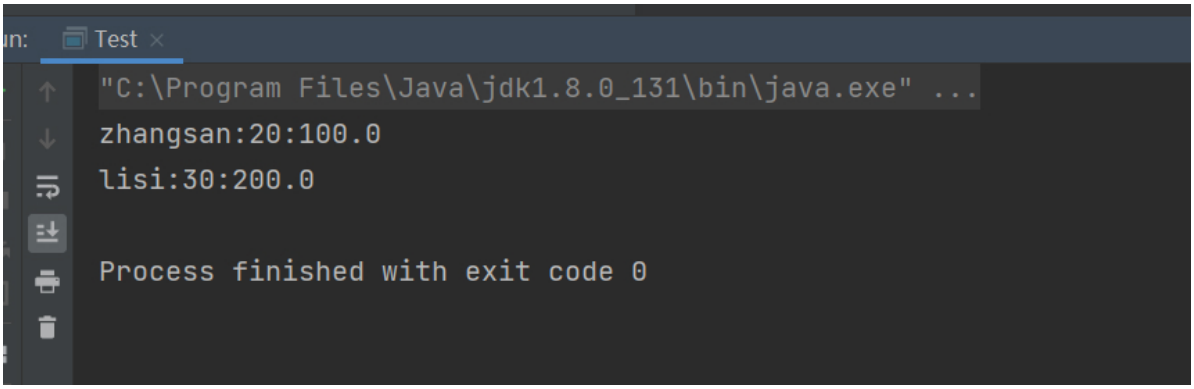
重写构造器，必须调用类默认的构造器

```
class Person(xname: String, xage: Int) {
  var name = xname
  var age = xage
  var money:Float = 100
  //重写构造器
  def this(xname:String,xage:Int,xmoney:Float){
    this(xname,xage) // 这一行注释掉直接报错
    money = xmoney
  }
}

object Test {
  def main(args: Array[String]): Unit = {
    var zhangsan = new Person("zhangsan",20)   // 位置参数
    var lisi = new Person(xname = "lisi",xage = 30,xmoney = 200) // 关键字参数
    println(zhangsan.name + ":" + zhangsan.age + ":" +zhangsan.money)
    println(lisi.name + ":" + lisi.age + ":" +lisi.money)
  }

}
```

运行结果



1.5 访问修饰符

Scala 访问修饰符基本和Java的一样，分别有：private，protected，public。

如果没有指定访问修饰符，默认情况下，Scala 对象的访问级别都是 public。

1.5.1 私有(private)成员

Scala 中的 private 限定符，比 Java 更严格，在嵌套类情况下，外层类甚至不能访问被嵌套类的私有成员。

用 private 关键字修饰，带有此标记的成员仅在包含了成员定义的类或对象内部可见，同样的规则还适用内部类。

```
class Outer{
    class Inner{
        private def f(){
            println("f")
        }
        class InnerMost{
            f() // 正确
        }
    }
    (new Inner).f() //错误
}
```

(new Inner).f( ) 访问不合法是因为 **f** 在 Inner 中被声明为 private，而访问不在类 Inner 之内。

但在 InnerMost 里访问 **f** 就没有问题的，因为这个访问包含在 Inner 类之内。

Java 中允许这两种访问，因为它允许外部类访问内部类的私有成员。

1.5.2 保护(protect)成员

在 scala 中，对保护（Protected）成员的访问比 java 更严格一些。因为它只允许保护成员在定义了该成员的的类的子类中被访问。而在java中，用 protected关键字修饰的成员，除了定义了该成员的类的子类可以访问，同一个包里的其他类也可以进行访问。

```
package p {
    class Super {
        protected def f() {println("f")}
    }
    class Sub extends Super {
        f()
    }
    class Other {
        (new Super).f() //错误
    }
}
```

上例中，Sub 类对 f 的访问没有问题，因为 f 在 Super 中被声明为 protected，而 Sub 是 Super 的子类。相反，Other 对 f 的访问不被允许，因为 other 没有继承自 Super。而后者在 java 里同样被认可，因为 Other 与 Sub 在同一包里。

1.5.3 公开(public)成员

```
class Outer {
   class Inner {
      def f() { println("f") }
      class InnerMost {
         f() // 正确
      }
   }
   (new Inner).f() // 正确因为 f() 是 public
}
```

Scala 中，如果没有指定任何的修饰符，则默认为 public。这样的成员在任何地方都可以被访问。

1.5.4 作用域保护

Scala中，访问修饰符可以通过使用限定词强调。格式为:

```
private[x]

或

protected[x]
```

这里的x指代某个所属的包、类或单例对象。如果写成private[x],读作"这个成员除了对[...]中的类或[...]中的包中的类及它们的伴生对像可见外，对其它所有类都是private。

这种技巧在横跨了若干包的大型项目中非常有用，它允许你定义一些在你项目的若干子包中可见但对于项目外部的客户却始终不可见的东西。

```
package bobsrockets{
    package navigation{
        private[bobsrockets] class Navigator{
         protected[navigation] def useStarChart(){}
         class LegOfJourney{
             private[Navigator] val distance = 100
             }
            private[this] var speed = 200
            }
        }
```

```
        package launch{
        import navigation._
        object Vehicle{
        private[launch] val guide = new Navigator
        }
    }
}
```

上述例子中，类 Navigator 被标记为 private[bobsrockets] 就是说这个类对包含在 bobsrockets 包里的所有的类和对象可见。

比如说，从 Vehicle 对象里对 Navigator 的访问是被允许的，因为对象 Vehicle 包含在包 launch 中，而 launch 包在 bobsrockets 中，相反，所有在包 bobsrockets 之外的代码都不能访问类 Navigator。

1.6 伴生类和伴生对象

object不能传递参数，object里面的属性和方法都是静态的(拟静态),类似于java中的工具类（类名.方法）

```
//下面是错误代码，代码直接标红
//object是一个单例对象（静态类），主函数入口是放在object下的
object Test(a:Int){      // object不能传递参数
  def main(args: Array[String]): Unit = {
  }
}
```

伴生类和伴生对象：在一个scala文件中，如果class和object的名字一样，则互为伴生类和伴生对象，他们可以直接访问到互相的私有成员变量

```
class Person(xname:String,xage:Int){
  val name = xname
  val age = xage
  private var money = 100 // class Person的私有成员 money

  //获取身高
  def test()={
    println("object Person的私有成员height = " + Person.height)  // object Person是静态的
  }
}


object Person {
  private var height = 180
  def main(args: Array[String]): Unit = {
    val zhangsan = new Person(xname = "zhangsan",xage = 40)
    zhangsan.test()
  }
}
```

1.7 总结

注意点：

- 建议类名首字母大写，方法首字母小写，类和方法命名建议符合驼峰命名法
- scala中的object是单例对象，相当于java中的工具类，可以看成是定义静态的方法的类。object不可以传参数。另：Trait不可以传参数。
- scala中的class类默认可以传参数，默认的参数就是默认的构造函数。
- 重写构造函数的时候，必须要调用默认的构造函数
- class类属性自带getter，setter方法
- 使用object时，不用new，使用class时要new。
- 如果在同一个scala文件中，如果class和object的名字一样，则互为伴生类和伴生对象，他们可以直接访问到互相的私有成员变量

二、键盘标准输入

编程中可以通过键盘输入语句来接收用户输入的数据。（类似java中的scanner对象）

在scala中只需要导入对应的包，甚至不需要实例化对象

```
import scala.io.StdIn

object Test {
  def main(args: Array[String]): Unit = {
    println("请输入姓名：")
    var name = StdIn.readLine() // 接收字符串
    println("请输入年龄：")
    var age = StdIn.readInt() // 接收整型
    printf("您的名字是%s,年龄是%d,", name, age)  //printf是格式化输出
  }
}
```

三、分支结构

```
import scala.io.StdIn

object Test2{
  def main(args: Array[String]): Unit = {
    println("请输入你的成绩")
    var score:Int = StdIn.readInt() // 传入整型
    /**
     * if else 分支结构
     *  与条件 &&
     *  或条件 ||
     *  非条件 !
     */
    if(score >= 0 && score < 60){
      println("不及格")
    }else if(score >= 60 && score <= 80){
      println("及格")
    }else if(score >80 && score <=100){
      println("优秀")
    }else{
```

```
      println("????")
    }
  }
}
```

四、for循环

for(临时变量 <- 序列)

4.1 快速创建数组

① 1 to 10 左闭右闭

生成 Range(1,2,3,4,5,6,7,8,9,10)

② 1 until 10 左闭右开

生成Range(1,2,3,4,5,6,7,8,9)

```
object Test{
  def main(args: Array[String]): Unit = {
    println(1 to 10) // Range 1 to 10
    println(1 until 10) // Range 1 until 10
    // 1 to 10  Range(1,2,3,4,5,6,7,8,9,10)
    for (i <- 1 to 10){print(i)} // 12345678910
    println()
    // 1 until 10 Range(1,2,3,4,5,6,7,8,9)
    for(i <- 1 until(10)){print(i)} // 123456789
  }
}
```

③ 带步长的Range

起点 to/until (终点,步长)

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     * 带步长的range
     */
    for(i <- 1 to (10,2)){  // 1,3,5,7,9
      println(i)
    }
  }
}
```

scala的for多循环条件都可以放在一行

4.2循环体中加入判断条件

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     * 循环体中(直接)加入判断条件
     */
    for(i <- 1 to 10; if i>5 ; if i % 2 ==0 ){
      println(i)
    }
  }
}
```

等同于

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     * 补充常规写法
     */
    for(i <- 1 to 10){
      if(i > 5 && i % 2 ==0){
        println(i)
      }
    }
  }
}
```

4.3 双重for循环

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     * scala风格双重for循环
     */
    for(i <- 1 to 10;j <- 1 to 10){
        println(i + " : "  + j)
    }
  }
}
```

等同于

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     * 常规双重for循环
     */
    for(i <- 1 to 10){
      for(j <- 1 to 10){
        println(i + " : "  + j)
      }
    }
  }
}
```

经典九九乘法表

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     *  九九乘法表
     */
    for(i <- 1 to 9; j <- 1 to i){
      print(j + "*" + i + " = " + j*i + "\t")
      if(i==j){
        println() // 换行
      }
    }
  }
}
```

等同于

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     *  九九乘法表（常规写法）
     */
    for(i <- 1 to 9){
      println()
      for(j <- 1 to i){
        printf("%d*%d=%d\t",j,i,j*i)
      }
    }
  }
}
```

scala中不能写count++ count-- 即不支持自增，自减，用赋值运算符代替count+=1

4.4 yield

```
object Test{
  def main(args: Array[String]): Unit = {
    /**
     * for循环用yield关键字返回一个集合   for{子句} yield{变量或者表达式}，原来的集合不会被改变，
     * 只会通过你的for/yield构建出一个新的集合
     */
    println(for(i <- 1 to 10) yield {i+1})  //Vector(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
    var list = for(i <- 1 to 10 ; if(i>5)  ) yield {i*10}
    for(w <- list){
      println(w)
    }
  }
}
```

五、while do...while

5.1 while 和 do...while

do...while 循环与 while 循环类似，但是 do...while 循环会确保至少执行一次循环。

```
/**
 * while循环 和 do while
 */

object Test4 {
  def main(args: Array[String]): Unit = {
//    var index = 0
//    while(index < 100){
//      printf("这是第%d次循环\n",index)
//      index += 1
//    }

    var index = 0
    do{
      printf("这是第%d次循环\n",index)
      index += 1
    }while(index < 100)
  }
}
```

5.2 Scala break 语句

Scala 语言中默认是没有 break 语句，但是你在 Scala 2.8 版本后可以使用另外一种方式来实现 *break* 语句。当在循环中使用 **break** 语句，在执行到该语句时，就会中断循环并执行循环体之后的代码块。

```
// 导入包
import util.control.Breaks._

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

    // 实现break功能    当 i==2 时 break
    breakable{
      for (i <- 1 to 10){
        if (i==2) break()
        println(i)
      }
    }
  }
}
```

```
//导入包
import util.control.Breaks._

object ContinueDemo {
  def main(args: Array[String]): Unit = {
    for (i <- 1 to 10) {
      // 实现continue功能    当 i==2 时 continue
      breakable {
        if (i == 2) break()
```

```
        println(i)
      }
    }
  }
}
```

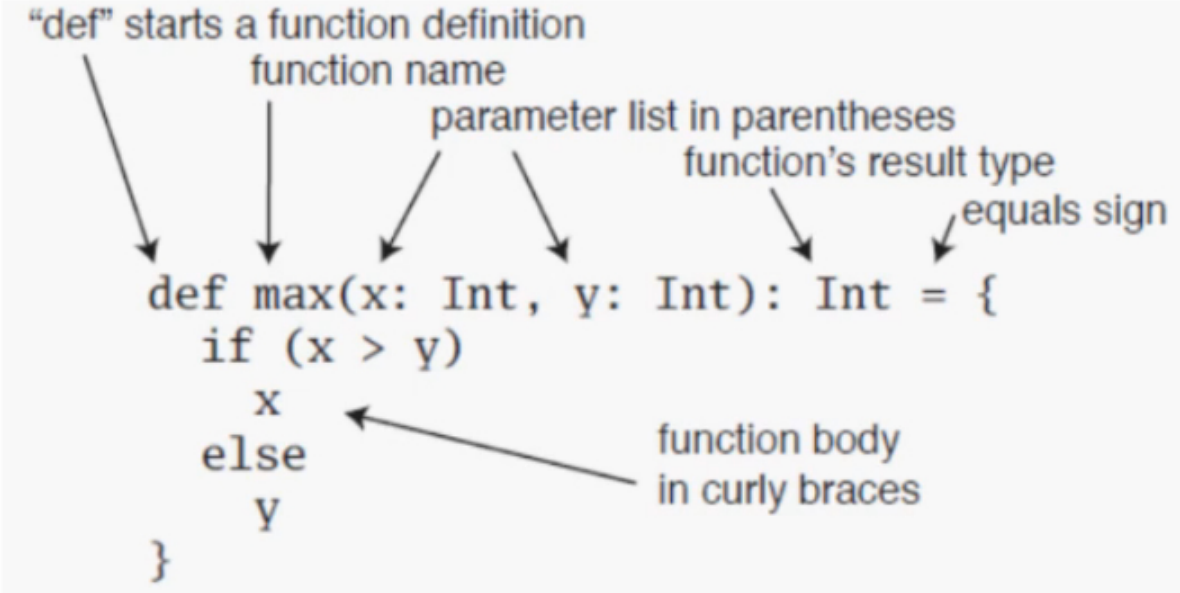
上面2个例子的区别其实就是`breakable`的范围大小，在循环内，就是continue，在循环外就是break

六、Scala函数

6.1 函数定义

有参函数

无参函数



Scala使用def关键字告诉编译器这是一个方法

```
object Test1 {
  def main(args: Array[String]): Unit = {
    //def 关键字
   def fun(a:Int,b:Int): Unit={
     println(a+b)
   }
   fun(1,1)

   def fun1(a:Int,b:Int):Int = a+b
   println(fun1(1,2))
  }
}
```

:Unit 无返回值

```
object Test1 {
  def main(args: Array[String]): Unit = {

    // 自定义一个求最大值的方法，比较两个参数的大小
    // def getMax(a:Int,b:Int):Unit ={  // 报错
    def getMax(a:Int,b:Int):Int ={
      if(a>b){
```

```
        return a
      }else{
        return b
      }
    }
  }
}
```

return关键字可以省略，scala自动将函数的最后一行作为返回值

```
object Test1 {
  def main(args: Array[String]): Unit = {

    // 自定义一个求最大值的方法，比较两个参数的大小
    def getMax(a:Int,b:Int):Int ={
      if(a>b){
        a
      }else{
        b
      }
      // 111  // 因为这里是最后一行，scala认为它才是返回值
    }
    println(getMax(20,30))  // 调用函数
  }
}
```

只有一句语句时，可以省略**{}**

```
object Test1 {
  def main(args: Array[String]): Unit = {

    // 自定义一个求最大值的方法，比较两个参数的大小
    def getMax(a:Int,b:Int):Int ={
      if(a>b)  // 只有一行时，可以省略 {}
        a
      else
        b
    }
    println(getMax(20,30))  // 调用函数
  }
}
```

函数可以省略返回类型，会进行类型自动推断（隐式）。显式调用return时不能省略返回类型

```
object Test1 {
  def main(args: Array[String]): Unit = {
    // 自定义一个求最大值的方法，比较两个参数的大小
    def getMax(a:Int,b:Int) = {
      if(a>b) {
        // return a  // 报错，return必须指定返回值类型
        a
      } else
        b
    }
    println(getMax(20,30))  // 调用函数
  }
}
```

没有 = 的话，返回值就是空，= 的作用是赋值，一般在无返回值的函数中省略 =

```
object Test1 {
  def main(args: Array[String]): Unit = {
    // 自定义一个求最大值的方法，比较两个参数的大小
    def getMax(a:Int,b:Int){  // 没有等号时
      println("函数执行了吗") // 输出成功
      if(a>b) {
        // return a  // 报错，return必须指定返回值类型
        a
      } else
        b
    }
    println(getMax(20,30))  // 结果为 ()
  }
}
```

方法的参数是用val定义的，即方法内只能使用参数而不能修改

```
// 报错
object Test1 {
  def main(args: Array[String]): Unit = {
    def plus(a:Int) = {
      a  = 10 // 报错reassignment to val
    }
  }
}
```

6.2 递归函数

- 递归函数：关键点在于递归的定义，终止条件（栈溢出）
- 递归函数不能省略方法的返回类型

```
object Test2 {
  def main(args: Array[String]): Unit = {

    /**
     * 100 * 99 * 98 *.....
     */

    // 错误写法  无限递归，因为scala会将最后一行设为返回值，一直在跑num * f1(num-1)
```

```
//    def f1(num:Int):Int= {
//      if(num==1)
//        num
//      num * f1(num-1)
//    }

    /**
     * 第一种解决办法
     */
    def f1(num:Int):Int={  //递归函数不能省略方法的返回类型
      if(num==1)
        num
      else
        num * f1(num-1)
    }

    /**
     * 第二种解决办法
     */
    def f2(num:Int):Int={ //递归函数不能省略方法的返回类型
      if(num==1)
        return num
      num * f2(num-1)
      }


    println(f1(5))
    println(f2(5))

  }
}
```

6.3 包含参数默认值的函数

- 默认值的函数中，如果传入的参数个数与函数定义相同，则传入的数值会覆盖默认值。
- 如果不想覆盖默认值，传入的参数个数小于定义的函数的参数，则需要指定参数名称

```
object Test3 {
  def main(args: Array[String]) = {
    /**
     * 包含参数默认的函数
     * @param a 默认值为5
     * @param b 默认值为10
     * @return
     */
    def f1(a:Int = 5, b:Int = 10): Int = {
      a + b
    }

    println(f1()) // a=5 + b=10
    println(f1(50,100)) // a=50 + b=100

    println(f1(100)) // a=100 + b=10    位置参数
    println(f1(b=100)) // a=5 + b=100 关键字参数

  }
```

```
}
```

6.4 可变长度个数的函数

```
/**
 * 可变参数个数的函数
 */
object Test4 {
  def main(args: Array[String]): Unit = {

    /**
     * 可变参数个数的函数
     * 注意：多个参数逗号分开
     */

    // Int* 是不定长的数组  java中 int ...a   python中 *a
    // 以元祖的形式返回
    def getSum(a:Int*)={
      var sum:Int = 0
      for(i <- a){
        sum += i
      }
      sum
    }
    println(getSum(1,2,3,4,5))

  }
}
```

6.5 匿名函数

一共如下三种：

1.有参匿名函数

2.无参匿名函数

3.有返回值的匿名

- 匿名函数用 => 定义
- 可以将匿名函数返回给val定义的值
- 匿名函数不能显式声明函数的返回类型
- 和高阶函数结合使用

```
object Test5{
  def main(args: Array[String]): Unit = {

    /**
     * 匿名函数 =>
     * 没有名字的函数
     */
    val f3 = (a:Int,b:Int) => {println((a+b))}   //有参匿名函数（无返回值）
    f3(1,5)
    val f4 = () => {println("无参匿名函数")} // 无参匿名函数（无返回值）
    f4()
    val f5 = (a:Int,b:Int) => {a+b} // 有参匿名函数（返回值）
```

```
    println(f5(1,5))
  }
}
```

1.匿名函数不允许设置默认值

2.匿名函数不允许声明返回类型

3.匿名函数加上return后，变成无返回值的函数，（建议不写return）

```
        - 实际上出现return后，相当于匿名函数声明返回值为Unit
```

4.匿名函数不能使用可变长度个数的参数

6.6 嵌套函数

```
object Test6 {
  def main(args: Array[String]): Unit = {
    /**
     * 嵌套函数
     * 函数里面写其他函数
     */
    def outer(a:Int,b:Int)={  // 外层函数
      def inner(c:Int)={   // 嵌套函数
        (a+b)*c
      }
      println(inner(10))
    }
    outer(1,2)
  }
}
```

6.7 偏应用函数

Scala 偏应用函数是一种表达式，你不需要提供函数需要的所有参数，只需要提供部分，或不提供所需参数。

```
object Test7 {
  def main(args: Array[String]): Unit = {

    def log(date:Date,s:String)={
      println("date is" + date + ",log is " + s)  // 打印时间
    }
    val date = new Date()
    log(date,"log1")  //date参数是不变的，只是改变s参数
    log(date,"log2")
    log(date,"log3")

    println()
    /**
     * 偏应用函数
     * Scala 偏应用函数是一种表达式，你不需要提供函数需要的所有参数，只需要提供部分，或不提供所需参数。
     *
     */
    // 想要调用log，以上变化的是第二个参数，可以用偏应用函数处理
    // date是实参，_表示占位，需要传入一个参数
```

```
    val logWithDate = log(date,_:String)   // _:Sting 是要接收的值
    logWithDate("log1")
    logWithDate("log2")
    logWithDate("log3")

  }
}
```

6.8 高阶函数（理解为主，能看懂就好）

```
object Test8 {
  def main(args: Array[String]): Unit = {

    /**
     * 高阶函数
     * 1.函数的参数是函数。 函数的格式是f:(A)=>B,其中A为f的接收类型参数,B为f的返回类型
     * 2.函数的返回类型是函数
     * 3.两者结合
     */
    def f1(a:Int,f:(Int,Int)=>Int)={
      // ctrl+alt+v
      val result = f(1, 2)
      a * result
    }
    //调用函数
    def f2(a:Int,b:Int):Int={ a + b }
//    f1(5,f2(1,2))  // 报错 其实实际上是 f1(5,3)
    println(f1(2,f2))
  }
}
```

6.8.1高阶函数+匿名函数

```
object Test9 {
  def main(args: Array[String]): Unit = {

    def f1(a: Int, f: (Int, Int) => Int) = {
      // ctrl+alt+v
      val result = f(1, 2)
      a * result
    }
    /**
     * 匿名函数+高阶函数
     */
    println(f1(5,(a:Int,b:Int)=>{a+b}))  // 匿名函数
  }
}
```

当匿名函数作为高阶函数的参数时

并且匿名函数只有一个参数时 可以省略参数a的数据类型或括号

```
object Test9 {
  def main(args: Array[String]): Unit = {

    def f1(a: Int, f: (Int) => Int) = {
```

```
      // ctrl+alt+v
      val result = f(1)
      a * result
    }
    /**
     * 高阶函数+匿名函数
     * 当匿名函数作为高阶函数的参数时，并且匿名函数只有一个参数时，可以省略 参数a的数据类型或括号
     */
    println(f1(5,a=>{a*10}))


  }
}
```

6.8.2 函数的返回类型是函数

```
object Test10 {
  def main(args: Array[String]): Unit = {
    /**
     * 函数的返回类型是函数
     */
    def f1(a:Int,b:Int):(String,String)=>String ={    // 返回值是函数
      def returnFuntion(c:String,d:String)={a + " "+ b + " "+ c + " " + d}  // 嵌套函数
      returnFuntion //返回一个函数
    }

    // 在idea中 alt+Enter快捷键  显式显示类型
    val function: (String, String) => String = f1(1,2)    //function接收的是一个函数
    println(function("一","二"))
  }
}
```

6.8.3 高阶函数的参数是函数，返回值也是函数

```
object Test11 {
  def main(args: Array[String]): Unit = {
    def f1(f: (Int, Int) => Int): (Int, Int) => Int = {
      f
    }

    /**
     *  参数是函数，返回类型也是函数
     */
    println( f1((a: Int, b: Int) => {a + b})(100, 200)  )


    /**
     * 附上完整版以供理解
     */
    def f2(a:Int,b:Int):Int= {return a+b}
    val f3 = f1(f2)
    println(f3(100,200))

    /**
```

```
     * 简化版
     */
    // _ 类似java中的 * 通配符
    // 变量只使用一次的时候可以简写成如下
    println(f1(_+_)(100,200))


  }
}
```

6.9 柯里化函数

可以理解为高阶函数的简化

函数柯里化：只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。

我们之前学习的一个函数，实现相加:

```
def add(x:Int,y:Int)=x+y
```

那么我们应用的时候，应该是这样用：add(1,2)

现在我们把这个函数变一下形

```
def add(x:Int)(y:Int) = x + y
```

在调用时

```
val result = add(1)  // 返回一个result，那result的值应该是一个匿名函数：(y:Int)=>1+y
// 继续调用result
val sum = result(2)
```

完整例子

```
object Test12 {
  def main(args: Array[String]): Unit = {
    /**
     * 柯里化函数，可以理解为高阶函数的简化
     */
    def f1(a:Int,b:Int)(c:Int,d:Int) = {
      a+b+c+d
    }
    println(f1(1,2)(3,4))
  }
}
```

6.10 闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

闭包通常来讲可以简单的认为是可以访问一个函数里面局部变量的另外一个函数。

```
// 如下面有一个匿名函数
val multiplier = (i:Int) => i * 10
// 函数体内有一个变量 i，它作为函数的一个参数。
```

```
val multiplier = (i:Int) => i * factor
// 在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形式参数，在 multiplier 函数被调用时，i 被赋予一个新的值。然而，factor不是形式参数，而是自由变量
```

```
var factor = 3
val multiplier = (i:Int) => i * factor
// 这里我们引入一个自由变量 factor，这个变量定义在函数外面。
// 这样定义的函数变量 multiplier 成为一个"闭包"，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数。
```

可以理解为，存在自由变量的函数就是闭包

完整实例

```
object Test {
   def main(args: Array[String]) {
      println( "muliplier(1) value = " +  multiplier(1) )
      println( "muliplier(2) value = " +  multiplier(2) )
   }
   var factor = 3
   val multiplier = (i:Int) => i * factor
}
```

七. Scala字符串

7.1 String

类型为 String (java.lang.String)

在 Scala 中，字符串的类型实际上是 Java String，它本身没有 String 类。

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

创建字符串实例

```
var s1 = "Hello World!";
var s2:String = "Hello World!";
```

7.2 StringBuilder可变

String 对象是不可变的，如果你需要创建一个可以修改的字符串，可以使用 String Builder 类

```
object Test_String {
  def main(args: Array[String]): Unit = {
    /**
     * String
     */
    val s1:String = "abcd"
    println(s1.indexOf("a")) //0
    println(s1.indexOf(98)) //1
    val s2 = "ABCD"
    println(s1==s2)  // false
```

```
    println(s1.compareToIgnoreCase(s2))  //0

//    val result = s1 + "b"  // 在内存中创建一个新对象
    /**
     * StringBuilder
     */
    val s3 = new StringBuilder
    s3.append("abc")
    println(s3) // abc
    s3 += 'd'
    s3 + 'e'
    println(s3) // abcde
//    s3 + "fi" // 添加失败
//    println(s3) // abcde
    /**
     * StringBuilder对象 += char类型不能 += String类型，String类型要用append添加
     */
    s3.append("fi")
    println(s3)
  }
}
```

7.3 String方法

字符串长度方法，我们可以使用 length() 方法来获取字符串长度：

```
object Test {
   def main(args: Array[String]) {
      var palindrome = "www.runoob.com";
      var len = palindrome.length();
      println( "String Length is : " + len );
   }
}
```

字符串连接，String 类中使用 concat() 方法来连接两个字符串：

```
string1.concat(string2);
//或者用加号连接
string1 = string1 + string2
```

String方法表，需要时查看

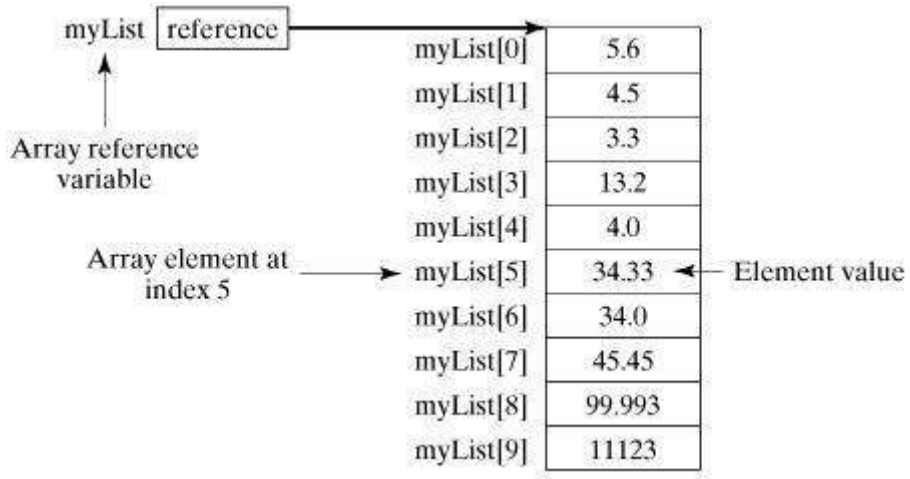
序号	方法及描述
1	**char charAt(int index)** 返回指定位置的字符
2	**int compareTo(Object o)** 比较字符串与对象
3	**int compareTo(String anotherString)** 按字典顺序比较两个字符串
4	**int compareToIgnoreCase(String str)** 按字典顺序比较两个字符串，不考虑大小写
5	**String concat(String str)** 将指定字符串连接到此字符串的结尾
6	**boolean contentEquals(StringBuffer sb)** 将此字符串与指定的 StringBuffer 比较。
7	**static String copyValueOf(char[] data)** 返回指定数组中表示该字符序列的 String
8	**static String copyValueOf(char[] data, int offset, int count)** 返回指定数组中表示该字符序列的 String
9	**boolean endsWith(String suffix)** 测试此字符串是否以指定的后缀结束
10	**boolean equals(Object anObject)** 将此字符串与指定的对象比较
11	**boolean equalsIgnoreCase(String anotherString)** 将此 String 与另一个 String 比较，不考虑大小写
12	**byte getBytes()** 使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中
13	**byte[] getBytes(String charsetName** 使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中
14	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** 将字符从此字符串复制到目标字符数组
15	**int hashCode()** 返回此字符串的哈希码
16	**int indexOf(int ch)** 返回指定字符在此字符串中第一次出现处的索引
17	**int indexOf(int ch, int fromIndex)** 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索
18	**int indexOf(String str)** 返回指定子字符串在此字符串中第一次出现处的索引
19	**int indexOf(String str, int fromIndex)** 返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
20	**String intern()** 返回字符串对象的规范化表示形式
21	**int lastIndexOf(int ch)** 返回指定字符在此字符串中最后一次出现处的索引
22	**int lastIndexOf(int ch, int fromIndex)** 返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索
23	**int lastIndexOf(String str)** 返回指定子字符串在此字符串中最右边出现处的索引
24	**int lastIndexOf(String str, int fromIndex)** 返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索

序号	方法及描述
25	**int length()** 返回此字符串的长度
26	**boolean matches(String regex)** 告知此字符串是否匹配给定的正则表达式
27	**boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** 测试两个字符串区域是否相等
28	**boolean regionMatches(int toffset, String other, int ooffset, int len)** 测试两个字符串区域是否相等
29	**String replace(char oldChar, char newChar)** 返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的
30	**String replaceAll(String regex, String replacement** 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串
31	**String replaceFirst(String regex, String replacement)** 使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串
32	**String[] split(String regex)** 根据给定正则表达式的匹配拆分此字符串
33	**String[] split(String regex, int limit)** 根据匹配给定的正则表达式来拆分此字符串
34	**boolean startsWith(String prefix)** 测试此字符串是否以指定的前缀开始
35	**boolean startsWith(String prefix, int toffset)** 测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
36	**CharSequence subSequence(int beginIndex, int endIndex)** 返回一个新的字符序列，它是此序列的一个子序列
37	**String substring(int beginIndex)** 返回一个新的字符串，它是此字符串的一个子字符串
38	**String substring(int beginIndex, int endIndex)** 返回一个新字符串，它是此字符串的一个子字符串
39	**char[] toCharArray()** 将此字符串转换为一个新的字符数组
40	**String toLowerCase()** 使用默认语言环境的规则将此 String 中的所有字符都转换为小写
41	**String toLowerCase(Locale locale)** 使用给定 Locale 的规则将此 String 中的所有字符都转换为小写
42	**String toString()** 返回此对象本身（它已经是一个字符串！）
43	**String toUpperCase()** 使用默认语言环境的规则将此 String 中的所有字符都转换为大写
44	**String toUpperCase(Locale locale)** 使用给定 Locale 的规则将此 String 中的所有字符都转换为大写
45	**String trim()** 删除指定字符串的首尾空白符
46	**static String valueOf(primitive data type x)** 返回指定类型参数的字符串表示形式

八、Scala集合

8.1 数组

Scala 语言中提供的数组是用来存储固定大小的同类型元素，数组对于每一门编辑应语言来说都是重要的数据结构之一。在Scala中，长度不可变。



8.1.1 创建数组

两种方式

```
object Test1 {
  def main(args: Array[String]): Unit = {
    /**
     *  创建数组的两种方式
     *  1. new Array[String](3)
     *  2. 直接Array
     */

    // 创建 类型为Int，长度为3的数组
    val arr1 = new Array[Int](3)
    // 赋值
    arr1(0) = 100
    arr1(1) = 200
    arr1(2) = 300
    println(arr1(0))

    // 创建String 类型的数组，并且直接赋值
    var arr2 = Array[String]("s100","s200","s300")

    // 最简单的写法
    var arr3 = Array(1,2,3)

  }
}
```

8.1.2 数组遍历

①for

```
object Test1 {
  def main(args: Array[String]): Unit = {
    /**
```

```
     *  遍历数组
     *  1. for
     *  2.foreach
     */
    // 创建String 类型的数组，并且直接赋值
    var arr2 = Array[String]("s100","s200","s300")

    // for 遍历
    for (i <-  arr2){
      println(i)
    }

  }
}
```

②foreach

数组名.foreach(函数)

```
object Test1 {
  def main(args: Array[String]): Unit = {
    /**
     *  遍历数组
     *  1. for
     *  2.foreach
     */

    var arr2 = Array[String]("s100","s200","s300")

    // foreach 遍历
    // foreach 将数组中的元素一个个提取出来
    // 语法：   数组名.foreach(函数)
    arr2.foreach(i => {println(i)})

  }
}
```

代码简写

```
object Test1 {
  def main(args: Array[String]): Unit = {
    /**
     *  遍历数组
     *  1. for
     *  2.foreach
     */

    var arr2 = Array[String]("s100","s200","s300")


    //代码简写：第一次
    arr2.foreach(i => println(i))

    //代码简写：第二次
    arr2.foreach(println(_))

    //代码简写：第三次
```

```
    arr2.foreach(println)

  }
}
```

8.1.3 二维数组

用定义的方式

```
object Test2 {
  def main(args: Array[String]): Unit = {
    /**
     * 二维数组
     */
    val array = new Array[Array[Int]](3)
    array(0) = Array(1,2)
    array(1) = Array(3,4)
    array(2) = Array(5,6)

    array.foreach(println)  // 结果是 [I@2d209079,[I@2752f6e2,[I@e580929

    array.foreach(x => {
      x.foreach(println)
      println("---------")
    })
  }
}
```

用Array.ofDim方式初始化

```
object Test2 {
  def main(args: Array[String]): Unit = {
    /**
     * 二维数组
     */
    val array = Array.ofDim[Int](3,2)  // 初始化

    var x = 1
    for(i <- 0 until  3 ; j <- 0 until 2 ) {
      array(i)(j) =  x
      x += 1
    }

    array.foreach(println)  // 结果是 [I@3d8c7aca,[I@5ebec15,[I@21bcffb5

    array.foreach(x => {
      x.foreach(println)
      println("---------")
    })
  }
}
```

练习

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

```
      var myList = Array(1.9, 2.9, 3.4, 3.5)

      // 输出所有数组元素
      for ( x <- myList ) {
         println( x )
      }

      // 计算数组所有元素的总和
      var total = 0.0;
      for ( i <- 0 to (myList.length - 1)) {
         total += myList(i);
      }
      println("总和为 " + total);

      // 查找数组中的最大元素
      var max = myList(0);
      for ( i <- 1 to (myList.length - 1) ) {
         if (myList(i) > max) max = myList(i);
      }
      println("最大值为 " + max);

   }
}
```

8.1.4 数组方法

下表中为 Scala 语言中处理数组的重要方法，使用它前我们需要使用 **import Array._** 引入包。

序号	方法和描述
1	**def apply(x: T, xs: T*): Array[T]** 创建指定对象 T 的数组, T 的值可以是 Unit, Double, Float, Long, Int, Char, Short, Byte, Boolean。
2	**def concat[T](xss: Array[T]*): Array[T]** 合并数组
3	**def copy(src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int): Unit** 复制一个数组到另一个数组上。相等于 Java's System.arraycopy(src, srcPos, dest, destPos, length)。
4	**def empty[T]: Array[T]** 返回长度为 0 的数组

5	**def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]**
	返回指定长度数组，每个数组元素为指定函数的返回值。
	以上实例数组初始值为 0，长度为 3，计算函数为**a=>a+1**：
	`scala> Array.iterate(0,3)(a=>a+1)`
	`res1: Array[Int] = Array(0, 1, 2)`

6	**def fill[T](n: Int)(elem: => T): Array[T]** 返回数组，长度为第一个参数指定，同时每个元素使用第二个参数进行填充。
7	**def fill[T](n1: Int, n2: Int)(elem: => T): Array[Array[T]]** 返回二数组，长度为第一个参数指定，同时每个元素使用第二个参数进行填充。
8	**def ofDim[T](n1: Int): Array[T]** 创建指定长度的数组
9	**def ofDim[T](n1: Int, n2: Int): Array[Array[T]]** 创建二维数组
10	**def ofDim[T](n1: Int, n2: Int, n3: Int): Array[Array[Array[T]]]** 创建三维数组

11	**def range(start: Int, end: Int, step: Int): Array[Int]** 创建指定区间内的数组，step 为每个元素间的步长
12	**def range(start: Int, end: Int): Array[Int]** 创建指定区间内的数组

13	**def tabulate[T](n: Int)(f: (Int)=> T): Array[T]**
	返回指定长度数组，每个数组元素为指定函数的返回值，默认从 0 开始。
	以上实例返回 3 个元素：
	`scala> Array.tabulate(3)(a => a + 5)`
	`res0: Array[Int] = Array(5, 6, 7)`

14	**def tabulate[T](n1: Int, n2: Int)(f: (Int, Int) => T): Array[Array[T]]** 返回指定长度的二维数组，每个数组元素为指定函数的返回值，默认从 0 开始。

```
import Array._
object Test3 {
  def main(args: Array[String]): Unit = {
    /**
     * 简单介绍几个Array方法
     */

    // concat() 拼接函数
    val arr1 = Array(1,2,3)
    val arr2 = Array(4,5,6)
    val arr3 = Array.concat(arr1,arr2)  // 拼接两个Array
    arr3.foreach(println)

    // fill() 函数
    var arr4 = Array.fill(4)("hello scala")
    arr4.foreach(println)
  }
}
```

8.1.5 可变长度数组

导入包

```
import Array._
import scala.collection.mutable.ArrayBuffer

object Test3 {
  def main(args: Array[String]): Unit = {
    /**
     * 可变长度数组
     */
    val arr1 = Array(1,2,3)

    val arr2 = ArrayBuffer[String]("a","b","c")

    arr2.append("e","f","d") // 可以添加多个元素
    arr2.+=("end")  // 最后追加元素
    arr2.+=:("start") //开头追加元素
    arr2.foreach(print)
  }
```

8.2 list

8.2.1 创建list

Scala 列表类似于数组，它们所有元素的类型都相同，但是它们也有所不同：列表是不可变的，值一旦被定义了就不能改变，其次列表 具有递归的结构（也就是链接表结构）而数组不是。。

```
object ListTest {
  def main(args: Array[String]): Unit = {
    // 字符串列表
    val site: List[String] = List("Runoob", "Google", "Baidu")

    // 整型列表
    val nums: List[Int] = List(1, 2, 3, 4)

    // 空列表
```

```
    val empty: List[Nothing] = List()

    // 二维列表
    val dim: List[List[Int]] =
      List(
        List(1, 0, 0),
        List(0, 1, 0),
        List(0, 0, 1)
      )
  }
}
```

其中**Nil**是长度为0的List

也可以通过**Nil**和**::**来构造(相对比较麻烦)

```
object ListTest {
  def main(args: Array[String]): Unit = {
    /**
     * 1.Nil 也可以表示为一个空列表。
     * 2.可以通过 Nil 和 :: 构造列表
     */
    // 字符串列表
    val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

    // 整型列表
    val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

    // 空列表
    val empty = Nil

    // 二维列表
    val dim = (1 :: (0 :: (0 :: Nil))) ::
      (0 :: (1 :: (0 :: Nil))) ::
      (0 :: (0 :: (1 :: Nil))) :: Nil

  }
}
```

8.2.2 list遍历

通过**for**和**foreach**

```
object ListTest {
  def main(args: Array[String]): Unit = {
    /**
     * list遍历
     */

    //创建
    val list = List(1,2,3,4,5)

    //遍历
    for(i <- list) {println(i)}   // 方式一
    list.foreach{x=>println(x)}   // 方式二
    list.foreach{println}         // 方式三
  }
```

```
}
```

8.2.3 list方法

①List.filter()

用于过滤元素

```
object ListTest {
  def main(args: Array[String]): Unit = {
    /**
     * 过滤元素
     */
    //创建
    val list = List(1,2,3,4,5)

    //fitter方法   list.filter(匿名函数,其中返回值是布尔值)
    val list2 = list.filter(x=>{
      if(x>3) true  // 过滤大于
      else false
    })

    list2.foreach(println)

  }
}
```

②List.count()

用于计数

```
object ListTest {
  def main(args: Array[String]): Unit = {
    /**
     * 根据条件计数
     */
    //创建
    val list = List(1,2,3,4,5)

    //count计数
    val i = list.count(x=>{true})  // 5
    println(i)

    //根据条件 count计数
    val j = list.count(x=>{if(x>2)true else false}) // 3
    println(j)

  }
}
```

③List.map() 是一个双射

1 to 1，一一对应关系

map是一个映射函数，将一个集合映射成另一个集合

```
object ListTest2 {
```

```
  def main(args: Array[String]): Unit = {
    /**
     * map方法
     * 将一个集合映射成另一个集合
     */
    val ls = List(20, 45, 67)

    val ls2 = ls.map(x => {
      x * 2
    })

    ls2.foreach(println)

  }
}
```

上述例子也可以用yield实现

```
object ListTest2 {
  def main(args: Array[String]): Unit = {
    /**
     * map方法
     * 将一个集合映射成另一个集合
     */
    val ls = List(20, 45, 67)

    //也可以用yield实现
    val ls2 = for(x <- ls) yield {x*2}

    ls2.foreach(println)
  }
}
```

map是一个双射(一一对应)，可以从下面例子中看出来

```
object ListTest2 {
  def main(args: Array[String]): Unit = {
    /**
     * map方法
     * 将一个集合映射成另一个集合
     */
    var ls = List[String]("hello world","hello java","hello scala")
    // 用map函数实现一个切分
    var strLs:List[Array[String]] = ls.map(x => {x.split(" ")})

    // 结果是
    /*
    * [
    *   ["hello","world"],
    *   ["hello","java"],
    *   ["hello","scala"]
    * ]
    *
    * 其中 "hello world"   -->  ["hello","world"]
    *     "hello java"    -->   ["hello","java"]
    *      "hello scala"  -->   ["hello","java"]
```

```
    * */
    strLs.foreach(x=>x.foreach(println))
  }
}
```

④ List.flatMap()　只是一个满射

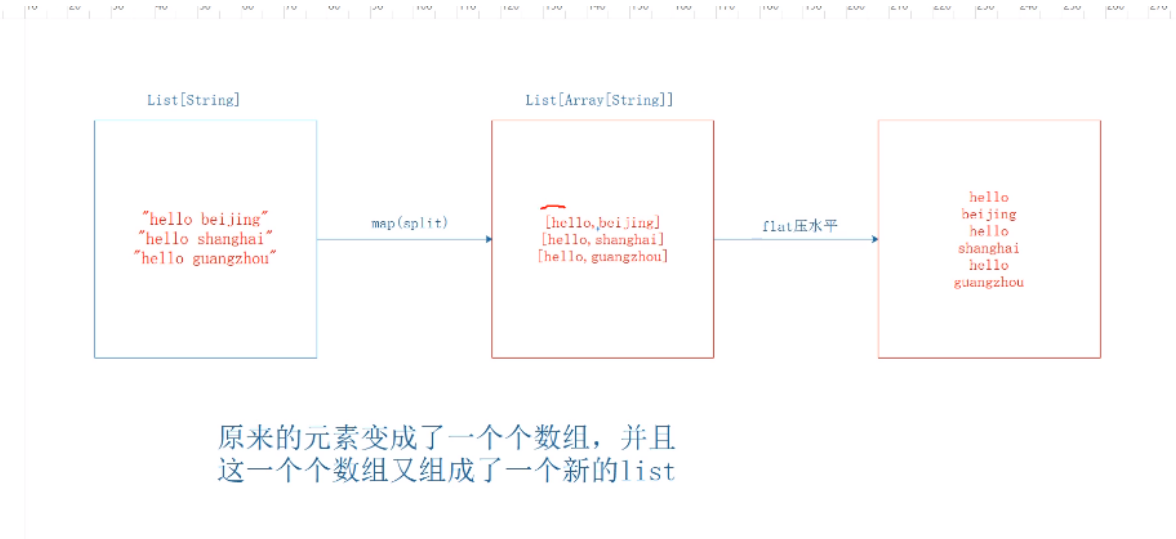
1 to n 或者 n to n，不一定是一一对应关系

它只是一个满射，可以从下面的例子看出来

```
package com.ListLeeson

object ListTest3 {
  def main(args: Array[String]): Unit = {
    /**
     * flatMap:把结果铺平
     * 将map完的结果，flat压平，最终形成 一进多出
     */
    var ls = List[String]("hello world","hello java","hello scala")

    val strLs = ls.flatMap( x=>{x.split(" ")})
    /*
    hello
    world
    hello
    java
    hello
    scala
     */
    strLs.foreach(println)
  }
}
```



⑤ List.head，List.tail

- `head` 返回列表第一个元素
- `tail` 返回一个列表，包含除了第一元素之外的其他元素

```
object ListTest4 {
  def main(args: Array[String]): Unit = {
    /**
     * head 返回第一个元素
     * tail 返回一个列表，包含除了第一个元素以外的所有元素
     */
    val ls = List[String]("一","二","三","四")

    println(ls.head)
    println(ls.tail)
    /* 结果
              一
              List(二, 三, 四)
     */
  }
}
```

⑥List.isEmpty

- `isEmpty` 在列表为空时返回true

```
object ListTest4 {
  def main(args: Array[String]): Unit = {
    /**
     * isEmpty
     */
    val ls1 = List()  // 空列表
    val ls2 = List(1,2,3,4,5) // 非空列表
    println(ls1.isEmpty)  // true
    println(ls2.isEmpty)  // false
  }
}
```

⑦List.fill()

我们可以使用 List.fill() 方法来创建一个指定重复数量的元素列表

```
object Test {
   def main(args: Array[String]) {
      val site = List.fill(3)("Runoob") // 重复 Runoob 3次
      println( "site : " + site  )

      val num = List.fill(10)(2)         // 重复元素 2, 10 次
      println( "num : " + num  )
   }
}
```

⑧List.reverse

List.reverse 用于将列表的顺序反转

```
object Test {
   def main(args: Array[String]) {
      val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      println( "site 反转前 : " + site )

      println( "site 反转后 : " + site.reverse )
   }
}
```

⑨List.tabulate()

List.tabulate() 方法是通过给定的函数来创建列表。

方法的第一个参数为元素的数量，可以是二维的，第二个参数为指定的函数，我们通过指定的函数计算结果并返回值插入到列表中，起始值为 0

```
object Test {
   def main(args: Array[String]) {
      // 通过给定的函数创建 5 个元素
      val squares = List.tabulate(6)(n => n * n)
      println( "一维 : " + squares  )

      // 创建二维列表
      val mul = List.tabulate( 4,5 )( _ * _ )
      println( "多维 : " + mul  )
   }
}
```

⑩连接列表

你可以使用 **:::** 运算符或 **List.:::()** 方法或 **List.concat()** 方法来连接两个或多个列表。

```
object Test {
   def main(args: Array[String]) {
      val site1 = "Runoob" :: ("Google" :: ("Baidu" :: Nil))
      val site2 = "Facebook" :: ("Taobao" :: Nil)

      // 使用 ::: 运算符
      var fruit = site1 ::: site2
      println( "site1 ::: site2 : " + fruit )

      // 使用 List.:::() 方法
      fruit = site1.:::(site2)
      println( "site1.:::(site2) : " + fruit )

      // 使用 concat 方法
      fruit = List.concat(site1, site2)
      println( "List.concat(site1, site2) : " + fruit  )


   }
}
```

List常用方法表

序号	方法及描述

1	**def +:(elem: A): List[A]**
	为列表预添加元素
	`scala> val x = List(1)`
	`x: List[Int] = List(1)`
	`scala> val y = 2 +: x`
	`y: List[Int] = List(2, 1)`
	`scala> println(x)`
	`List(1)`

2	**def ::(x: A): List[A]** 在列表开头添加元素
3	**def :::(prefix: List[A]): List[A]** 在列表开头添加指定列表的元素

4	**def :+(elem: A): List[A]**
	复制添加元素后列表。
	`scala> val a = List(1)`
	`a: List[Int] = List(1)`
	`scala> val b = a :+ 2`
	`b: List[Int] = List(1, 2)`
	`scala> println(a)`
	`List(1)`

5	**def addString(b: StringBuilder): StringBuilder** 将列表的所有元素添加到 StringBuilder
6	**def addString(b: StringBuilder, sep: String): StringBuilder** 将列表的所有元素添加到 StringBuilder，并指定分隔符
7	**def apply(n: Int): A** 通过列表索引获取元素
8	**def contains(elem: Any): Boolean** 检测列表中是否包含指定的元素
9	**def copyToArray(xs: Array[A], start: Int, len: Int): Unit** 将列表的元素复制到数组中。
10	**def distinct: List[A]** 去除列表的重复元素，并返回新列表
11	**def drop(n: Int): List[A]** 丢弃前n个元素，并返回新列表
12	**def dropRight(n: Int): List[A]** 丢弃最后n个元素，并返回新列表
13	**def dropWhile(p: (A) => Boolean): List[A]** 从左向右丢弃元素，直到条件p不成立
14	**def endsWith[B](that: Seq[B]): Boolean** 检测列表是否以指定序列结尾
15	**def equals(that: Any): Boolean** 判断是否相等

16	**def exists(p: (A) => Boolean): Boolean**
	判断列表中指定条件的元素是否存在。
	判断l是否存在某个元素:
	`scala> l.exists(s => s == "Hah")`
	`res7: Boolean = true`

17	**def filter(p: (A) => Boolean): List[A]**
	输出符号指定条件的所有元素。
	过滤出长度为3的元素:
	`scala> l.filter(s => s.length == 3)`
	`res8: List[String] = List(Hah, WOW)`

18	**def forall(p: (A) => Boolean): Boolean** 检测所有元素。 例如：判断所有元素是否以"H"开头： scala> l.forall(s => s.startsWith("H")) res10: Boolean = false
19	**def foreach(f: (A) => Unit): Unit** 将函数应用到列表的所有元素
20	**def head: A** 获取列表的第一个元素
21	**def indexOf(elem: A, from: Int): Int** 从指定位置 from 开始查找元素第一次出现的位置
22	**def init: List[A]** 返回所有元素，除了最后一个

23	**def intersect(that: Seq[A]): List[A]** 计算多个集合的交集
24	**def isEmpty: Boolean** 检测列表是否为空
25	**def iterator: Iterator[A]** 创建一个新的迭代器来迭代元素
26	**def last: A** 返回最后一个元素
27	**def lastIndexOf(elem: A, end: Int): Int** 在指定的位置 end 开始查找元素最后出现的位置
28	**def length: Int** 返回列表长度
29	**def map[B](f: (A) => B): List[B]** 通过给定的方法将所有元素重新计算
30	**def max: A** 查找最大元素
31	**def min: A** 查找最小元素
32	**def mkString: String** 列表所有元素作为字符串显示
33	**def mkString(sep: String): String** 使用分隔符将列表所有元素作为字符串显示

34	**def reverse: List[A]** 列表反转
35	**def sorted[B >: A]: List[A]** 列表排序
36	**def startsWith[B](that: Seq[B], offset: Int): Boolean** 检测列表在指定位置是否包含指定序列
37	**def sum: A** 计算集合元素之和
38	**def tail: List[A]** 返回所有元素，除了第一个
39	**def take(n: Int): List[A]** 提取列表的前n个元素
40	**def takeRight(n: Int): List[A]** 提取列表的后n个元素
41	**def toArray: Array[A]** 列表转换为数组
42	**def toBuffer[B >: A]: Buffer[B]** 返回缓冲区，包含了列表的所有元素
43	**def toMap[T, U]: Map[T, U]** List 转换为 Map

44	**def toSeq: Seq[A]** List 转换为 Seq
45	**def toSet[B >: A]: Set[B]** List 转换为 Set
46	**def toString(): String** 列表转换为字符串

8.2.4 可变长度List

```
import scala.collection.mutable.ListBuffer
object ListTest5 {
  def main(args: Array[String]): Unit = {
    /**
     * 可变长度List
     */
    val lsBuffer = ListBuffer[Int](1,2,4,5,6,7,8)
    lsBuffer.append(59,60) // 在结尾任意追加多个元素
    lsBuffer.+=(100) // 在结尾追加一个元素
    lsBuffer.+=:(-100) // 在开头追加一个元素

    lsBuffer.foreach(println)
  }
}
```

8.3 Set

Scala Set(集合)是没有重复的对象集合，所有的元素都是唯一的。

Scala 集合分为可变的和不可变的集合。

默认情况下，Scala 使用的是不可变集合，如果你想使用可变集合，需要引用 **scala.collection.mutable.Set** 包

8.3.1 创建Set

注意：Set集合自动去重

```
object Test1 {
  def main(args: Array[String]): Unit = {
    /**
     * 创建set
     */
    val set1 = Set(1,2,3,4,4)  // set会自动去重
    val set2 = Set(1,2,5)
    println(set1) // Set(1, 2, 3, 4)
  }
}
```

8.3.2 遍历Set

同样使用**for**和**foreach**

```
object Test2 {
  def main(args: Array[String]): Unit = {
    /**
     * for  和   foreach
```

```
     */
    val set1 = Set(1,2,3,4)

    // for
    for (s <- set1){println(s)}

    // foreach
    set1.foreach(s=>println(s))
  }
}
```

8.3.3 连接Set

你可以使用 ++ 运算符或 **Set.++()** 方法来连接两个集合。如果元素有重复的就会移除重复的元素。

```
object Test {
   def main(args: Array[String]) {
      val site1 = Set("Runoob", "Google", "Baidu")
      val site2 = Set("Faceboook", "Taobao")

      // ++ 作为运算符使用
      var site = site1 ++ site2
      println( "site1 ++ site2 : " + site )

      //  ++ 作为方法使用
      site = site1.++(site2)
      println( "site1.++(site2) : " + site )
   }
}
```

8.3.4 Set方法举例

- 1.交集： `intersect`,`&`
- 2.差集： `diff`, `&~`
- 3.子集： `subsetOf`
- 4.最大： `max`
- 5.最小： `min`
- 6.转成数组： `toList`
- 7.转成字符串： `mkString`

```
object Test3 {
  def main(args: Array[String]): Unit = {
    val set1 = Set(1,2,3,4)
    val set2 = Set(1,2,3)

    // 并集
    println(Set(1,2,3).union(Set(2,3,5))) // Set(1, 2, 3, 5)
    println("~~~~~~~~~~~~")

    // 交集
    val set3 = set1.intersect(set2)
    val set4 = set1.&(set2)
    println(set3)  // Set(1, 2, 3)
    println(set4)  // Set(1, 2, 3)
    println("~~~~~~~~~~~~")
```

```
    // 差集
    set1.diff(set2).foreach(println)  // 4
    set1.&~(set2).foreach(println)  // 4
    println("~~~~~~~~~~~~")

    // 子集
    println(set1.subsetOf(set2)) // false  set1 不是 set2 的子集
    println(set2.subsetOf(set1)) // true  set2 是 set1 的子集
    println("~~~~~~~~~~~~")

    // 最大值和最小值
    println(set1.max) // 4
    println(set1.min) // 1
    println("~~~~~~~~~~~~")

    // 转成Array和List
    println(set1.toArray) //[I@7a5d012c
    set1.toArray.foreach(println)  // 1 2 3 4
    println(set1.toList)  // List(1, 2, 3, 4)
    println("~~~~~~~~~~~~")

    //转成字符串
    println(set1.mkString) // 1234
    println(set1.mkString("\t")) // 1   2   3   4
  }
}
```

常用set方法：https://www.runoob.com/scala/scala-sets.html

8.3.5 可变长度Set

```
import scala.collection.mutable.Set
object Test4 {
  def main(args: Array[String]): Unit = {
    val set1 = Set[Int](1,2,3,4,5)
    set1.add(100)
    set1.+=(200)
//    set1.+=:(-100) // 报错
    set1.+=(1,200,300)
    set1.foreach(println)  // 无序

  }
}
```

8.4 Map

Map(映射)是一种可迭代的键值对（key/value）结构。

8.4.1 创建Map

- `Map(1 -> "shanghai")`
- `Map((1,"shanghai"))`

创建`map`时，相同的`key`被后面相同的`key`顶替掉，只保留一个

```
object Test1 {
  def main(args: Array[String]): Unit = {
```

```
    /**
     * 创建Map
     */
    val map1 = Map("1"->"shanghai",2->"guangzhou",("3","beijing"))
    println(map1) // Map(1 -> shanghai, 2 -> guangzhou, 3 -> beijing)

    // 空Map
    // 空哈希表，键为字符串，值为整型
    var A:Map[Char,Int] = Map()
    // isEmpty  在 Map 为空时返回true
    println(A.isEmpty)
  }
}
```

定义 Map 时，需要为键值对定义类型。如果需要添加 key-value 对，可以使用 + 号，如下所示

```
    map1 += ('I' -> 1)
    map1 += ('J' -> 5)
    map1 += ('K' -> 10)
    map1 += ('L' -> 100)
```

8.4.2 获取Map的值

Map 中的键都是唯一的。

所有的值都可以通过键来获取。

Map 也叫哈希表（Hash tables）。

- `map.get(key)`
- `map.get(key).getOrElse("no value")` 这种方式是如果`map`中没有对应项则赋值为`getOrElse()`里面的值

```
object Test1 {
  def main(args: Array[String]): Unit = {
    /**
     * 创建Map
     */
    val map1 = Map("1"->"shanghai",2->"guangzhou",("3","beijing"))
    println(map1) // Map(1 -> shanghai, 2 -> guangzhou, 3 -> beijing)

    // 获取值
    /**
     * map.get(key)
     * map.get(key).getOrElse("no value") 这种方式是如果map中没有对应项则赋值为
getOrElse里面的值
     */
    println(map1.get("1"))  // 根据key找value
    val result = map1.get(8).getOrElse("no value")
    println(result)
  }
}
```

8.4.2 遍历Map

两种方式`for`, `foreach`

```
object Test2 {
  def main(args: Array[String]): Unit = {
    val map1 = Map("1"->"shanghai",2->"guangzhou",("3","beijing"))  // 创建map

    // 遍历map
    map1.foreach(println) // (1,shanghai)   (2,guangzhou)   (3,beijing)

    // 遍历map的key和value
    map1.foreach(x=>{println(x._1+" : " + x._2)})  // x._1 是key  x._2 是 value
  }
}
```

遍历keys

```
object Test2 {
  def main(args: Array[String]): Unit = {
    val map1 = Map("1" -> "shanghai", 2 -> "guangzhou", ("3", "beijing")) // 创建
map
    /**
     * 遍历keys
     */

    val keys: Iterable[Any] = map1.keys  // 获取所有key
    // 遍历key
    keys.foreach(x=>{println(x+" : " + map1.get(x))})

  }
}
```

遍历values

```
object Test2 {
  def main(args: Array[String]): Unit = {
    val map1 = Map("1" -> "shanghai", 2 -> "guangzhou", ("3", "beijing")) // 创建
map
    /**
     * 遍历keys
     */

    val keys: Iterable[Any] = map1.keys  // 获取所有key
    // 遍历key
    keys.foreach(x=>{println(x+" : " + map1.get(x))})

  }
}
```

8.4.3 Map合并

你可以使用 ++ 运算符或 **Map.++()** 方法来连接两个 Map，Map 合并时会移除重复的 key。

```
object Test {
   def main(args: Array[String]) {
      val colors1 = Map("red" -> "#FF0000",
                        "azure" -> "#F0FFFF",
                        "peru" -> "#CD853F")
      val colors2 = Map("blue" -> "#0033FF",
                        "yellow" -> "#FFFF00",
                        "red" -> "#FF0000")

      //  ++ 作为运算符
      var colors = colors1 ++ colors2
      println( "colors1 ++ colors2 : " + colors )

      //  ++ 作为方法
      colors = colors1.++(colors2)
      println( "colors1.++(colors2) : " + colors )

   }
}
```

8.4.4 Map方法

① Map.filter()

用于过滤，留下符合条件的记录

```
object Test3 {
  def main(args: Array[String]): Unit = {
    val map1 = Map("age" -> 18, "money" -> 200)

    // Map.filter(匿名函数) 过滤方法
    val map2 = map1.filter(x => {
      if (x._2 > 20)
        true
      else
        false
    })

    println(map2)  // Map(money -> 200)
  }
}
```

②Map.count()

统计符合条件的记录数

```
object Test3 {
  def main(args: Array[String]): Unit = {
    val map1 = Map("age" -> 18, "money" -> 200)
    /**
     *  Map.count(匿名函数)
     */
    // 全计数
```

```
    println(map1.count(x=>{true}))  // 2

    //条件计数
    val result = map1.count(x => {
      if (x._2 > 20)
        true
      else
        false
    })

    println(result)  // 1
  }
}
```

③Map.contains()

`map` 中是否包含某个 `key`

```
object Test3 {
  def main(args: Array[String]): Unit = {
    val map1 = Map("age" -> 18, "money" -> 200)

    /**
     * Map.contains(key) 判断map是否包含某个key
     */
    println(map1.contains("age"))  // true
    println(map1.contains("name"))  // false
  }
}
```

if + Map.contains() 组合使用

```
object Test {
   def main(args: Array[String]) {
      val sites = Map("runoob" -> "http://www.runoob.com",
                      "baidu" -> "http://www.baidu.com",
                      "taobao" -> "http://www.taobao.com")

      if( sites.contains( "runoob" )){
           println("runoob 键存在，对应的值为 :"  + sites("runoob"))
      }else{
           println("runoob 键不存在")
      }
      if( sites.contains( "baidu" )){
           println("baidu 键存在，对应的值为 :"  + sites("baidu"))
      }else{
           println("baidu 键不存在")
      }
      if( sites.contains( "google" )){
           println("google 键存在，对应的值为 :"  + sites("google"))
      }else{
           println("google 键不存在")
      }
   }
}
```

④Map.exist()

符合条件的记录存在与否

```
object Test3 {
  def main(args: Array[String]): Unit = {
    val map1 = Map("age" -> 18, "money" -> 200,"score"->400)
    // exists 存在与否
    val flag = map1.exists(x=>{
      if (x._2>20){   // 不一定遍历完，只要满足条件就结束
        println(x._2)
        true
      }
      else
        false
    })

    println(flag)  // 结果 200 true
  }
}
```

Map方法表

序号	方法及描述
1	**def ++(xs: Map[(A, B)]): Map[A, B]** 返回一个新的 Map，新的 Map xs 组成
2	**def -(elem1: A, elem2: A, elems: A*): Map[A, B]** 返回一个新的 Map, 移除 key 为 elem1, elem2 或其他 elems。
3	**def --(xs: GTO[A]): Map[A, B]** 返回一个新的 Map, 移除 xs 对象中对应的 key
4	**def get(key: A): Option[B]** 返回指定 key 的值
5	**def iterator: Iterator[(A, B)]** 创建新的迭代器，并输出 key/value 对
6	**def addString(b: StringBuilder): StringBuilder** 将 Map 中的所有元素附加到StringBuilder，可加入分隔符
7	**def addString(b: StringBuilder, sep: String): StringBuilder** 将 Map 中的所有元素附加到StringBuilder，可加入分隔符
8	**def apply(key: A): B** 返回指定键的值，如果不存在返回 Map 的默认方法

9	**def clear(): Unit** 清空 Map
10	**def clone(): Map[A, B]** 从一个 Map 复制到另一个 Map
11	**def contains(key: A): Boolean** 如果 Map 中存在指定 key，返回 true，否则返回 false。
12	**def copyToArray(xs: Array[(A, B)]): Unit** 复制集合到数组
13	**def count(p: ((A, B)) => Boolean): Int** 计算满足指定条件的集合元素数量
14	**def default(key: A): B** 定义 Map 的默认值，在 key 不存在时返回。
15	**def drop(n: Int): Map[A, B]** 返回丢弃前n个元素新集合
16	**def dropRight(n: Int): Map[A, B]** 返回丢弃最后n个元素新集合
17	**def dropWhile(p: ((A, B)) => Boolean): Map[A, B]** 从左向右丢弃元素，直到条件p不成立

18	**def empty: Map[A, B]** 返回相同类型的空 Map
19	**def equals(that: Any): Boolean** 如果两个 Map 相等(key/value 均相等)，返回true，否则返回false
20	**def exists(p: ((A, B)) => Boolean): Boolean** 判断集合中指定条件的元素是否存在
21	**def filter(p: ((A, B))=> Boolean): Map[A, B]** 返回满足指定条件的所有集合
22	**def filterKeys(p: (A) => Boolean): Map[A, B]** 返回符合指定条件的不可变 Map
23	**def find(p: ((A, B)) => Boolean): Option[(A, B)]** 查找集合中满足指定条件的第一个元素
24	**def foreach(f: ((A, B)) => Unit): Unit** 将函数应用到集合的所有元素
25	**def init: Map[A, B]** 返回所有元素，除了最后一个

26	**def isEmpty: Boolean** 检测 Map 是否为空
27	**def keys: Iterable[A]** 返回所有的key/p>
28	**def last: (A, B)** 返回最后一个元素
29	**def max: (A, B)** 查找最大元素
30	**def min: (A, B)** 查找最小元素
31	**def mkString: String** 集合所有元素作为字符串显示
32	**def product: (A, B)** 返回集合中数字元素的积。
33	**def remove(key: A): Option[B]** 移除指定 key

34	**def retain(p: (A, B) => Boolean): Map.this.type** 如果符合满足条件的返回 true
35	**def size: Int** 返回 Map 元素的个数
36	**def sum: (A, B)** 返回集合中所有数字元素之和
37	**def tail: Map[A, B]** 返回一个集合中除了第一元素之外的其他元素
38	**def take(n: Int): Map[A, B]** 返回前 n 个元素
39	**def takeRight(n: Int): Map[A, B]** 返回后 n 个元素
40	**def takeWhile(p: ((A, B)) => Boolean): Map[A, B]** 返回满足指定条件的元素
41	**def toArray: Array[(A, B)]** 集合转数组

42	**def toBuffer[B >: A]: Buffer[B]** 返回缓冲区，包含了 Map 的所有元素
43	**def toList: List[A]** 返回 List，包含了 Map 的所有元素
44	**def toSeq: Seq[A]** 返回 Seq，包含了 Map 的所有元素
45	**def toSet: Set[A]** 返回 Set，包含了 Map 的所有元素
46	**def toString(): String** 返回字符串对象