Scala是一门多范式的编程语言，一种类似java的编程语言实现可伸缩的语言并集成面向对象编程和函数式编程的各种特性。

Scala语言的特点：面向对象编程、面向函数式编程、静态类型语言（变量的数据类型在编译阶段确定，动态类型语言是变量的数据类型在运行阶段确定如Python）、基于JVM（Scala程序文件.scala需要编译成Java的.class文件在JVM上运行，将.scala编译成的.class文件反编译后会产生对应的.scala的.java文件），Scala语言执行速度快，代码简洁如Python，大数据生态圈Spark和Kafka的源码是由Scala语言编写

scalac .scala文件 编译指定的.scala文件

IntelliJ IDEA的Scala插件离线下载地址<http://plugins.jetbrains.com/plugin/1347-scala>

IntelliJ IDEA的插件在线下载安装File->Settings->Plugins

Scala语句结束不需要分号

定义变量（Scala定义变量除了抽象类或者接口时不需要初始化，其他变量必须初始化），定义变量可以不指定变量的数据类型，系统会根据变量的初始化值推断变量的数据类型，数据类型确定后不能改变：

var 变量名=初始化值

var 变量名:数据类型=初始化值

Scala的值类型：Byte、Char、Short、Int、Long、Float、Double，类似Java的基本类型的对应的包装类

定义常量（Scala的常量相当于Java中final修饰的变量），Scala的常量类型为值类型（相当于Java的基本数据类型对应的包装类）时常量的值不能修改，常量类型为引用类型时引用不可变，引用的内容可变,[lazy修饰的val常量的值在需要的时候才会赋予指定的常量名，在一定程度上节约资源]：

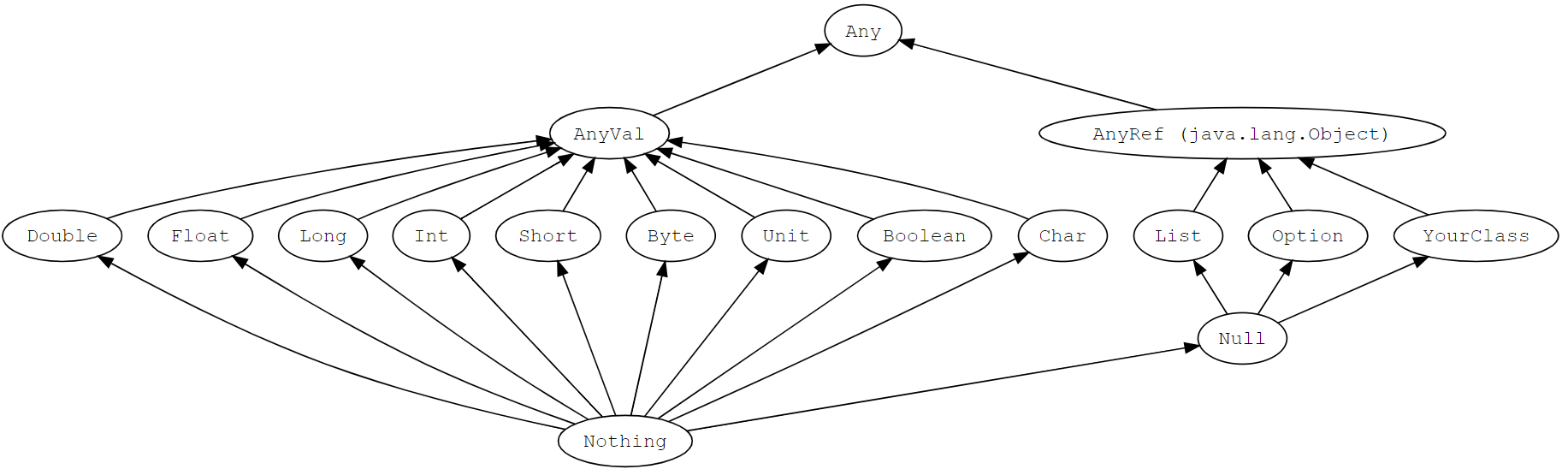
[lazy] val 常量名=初始化值

[lazy] val 常量名:数据类型=初始化值

Scala官方推荐使用val，val比var安全，val在垃圾回收时比var更快

数据类型：

值类型是类类型，无基本数据类型和包装类之分（相当于Java中的包装类）



Any相当于Java中的Object超类，AnyVal值类型，Unit类型相当于Java中的Void类型有一个对象()，AnyRef引用类型，Option类有两个子类Some和None，YourClass自定义类类型属于引用类型，Null类型有一个对象null，Nothing类型无法创建相应对象和没有相应对象（为了完善Scala概念做平滑过渡，异常有可能会返回Nothing类型）

Scala的运算符都是方法的重载或方法的调用（即 变量 运算符 变量 与变量.运算符(变量)结果一致），Scala中没有++和--运算符可以用+=和-=代替

Scala表达式是一个语句块包含一条或多条语句，表达式有返回值，返回值是表达式中最后一条语句的执行结果不用return返回

表达式无结果或最后一条语句为赋值语句或输出语句返回Unit类型的对象()

条件表达式包含if/else的语句块，块表达式为{一条或多条语句}

循环：

for循环（ “<-”表示循环赋值， “数值 to/until 数值”表示从数值到数值包含/不包含右边界数值,yield表示从集合中取出的所有变量通过表达式后重新封装成集合）：for(变量名 <- 表达式或数组或集合)[yield 表示式]

for循环嵌套：for(变量名 <- 表达式或数组或集合[;变量名 <- 表达式或数组或集合])

while循环：while(条件语句){表达式}

do while循环：do{表达式} while (条件语句)

方法：

def 方法名(变量名:变量类型[\*]=默认值[,变量名:变量类型默认值]):返回类型=方法体，\*表示可变长参数即指定参数的个数可变（可变参数位置必须在方法参数的最后一个），方法会将传入的参数封装成指定变量名的数组或集合

def 方法名(变量名:变量类型=默认值)[(变量名:变量类型=默认值)，参数列表]:返回类型=方法体

方法体其实为快表达式，返回类型除递归方法必须指明外其他方法的返回类型系统可以根据方法体的返回值推断类型即可以省略

方法名(对应变量名位置的参数)或方法名(变量名=变量值[,变量名=变量值])调用方法，调用多个参数的方法传入的参数个数少于方法参数个数默认从第一个开始向后依次传入

def 方法名[()]:返回类型=方法体 无参的方法可以使用不带()的方法名直接调用，定义无参方法时不带()调用时也不能带()

函数：

(变量名:变量值[,变量名:类型])=>表达式 调用函数时使用系统返回的res信息作为暂时函数名

val 函数名=(变量名:类型[,变量名:类型])=>表达式 常用的函数定义

val 函数名=((变量名:类型[,变量名:类型])=>表达式)

val 函数名=(\_:类型)操作符[(\_:类型)]=>表达式

val 函数名:(类型[,类型])=>返回类型=(\_操作符\_)

val 函数名:(类型[,类型])=>返回类型=(变量名[,变量名])=>表达式

Scala中函数其实是创建的Function2的对象，常用的函数定义等价于：

val 函数名=new Function2[参数类型,返回类型]{def apply(变量名:类型,变量名:类型):返回类型=表达式}

val 函数名=()=>表达式 无参函数的定义，函数名()调用“()”不能省略，省略代表函数对象本身

Scala方法与函数的区别：

方法与函数的定义语法不同、方法一般定义在类、特质或者Object中共享所在类、特质或者Object中的属性，函数可以被调用存放到一个变量中作为参数传递给其他的方法或者函数也可以作为返回值

方法与函数的联系：

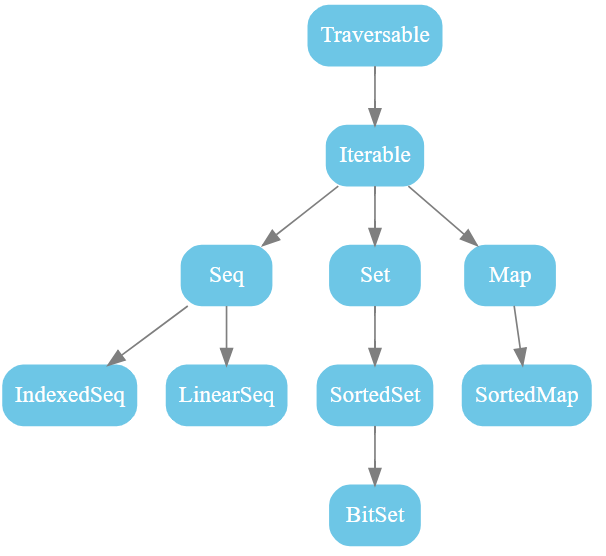
函数可以作为参数传递给方法，方法可以转换成函数，方法作为参数传入方法或函数时系统自动将传入的方法转换成函数

方法名 \_ 将指定方法显示的转换成函数

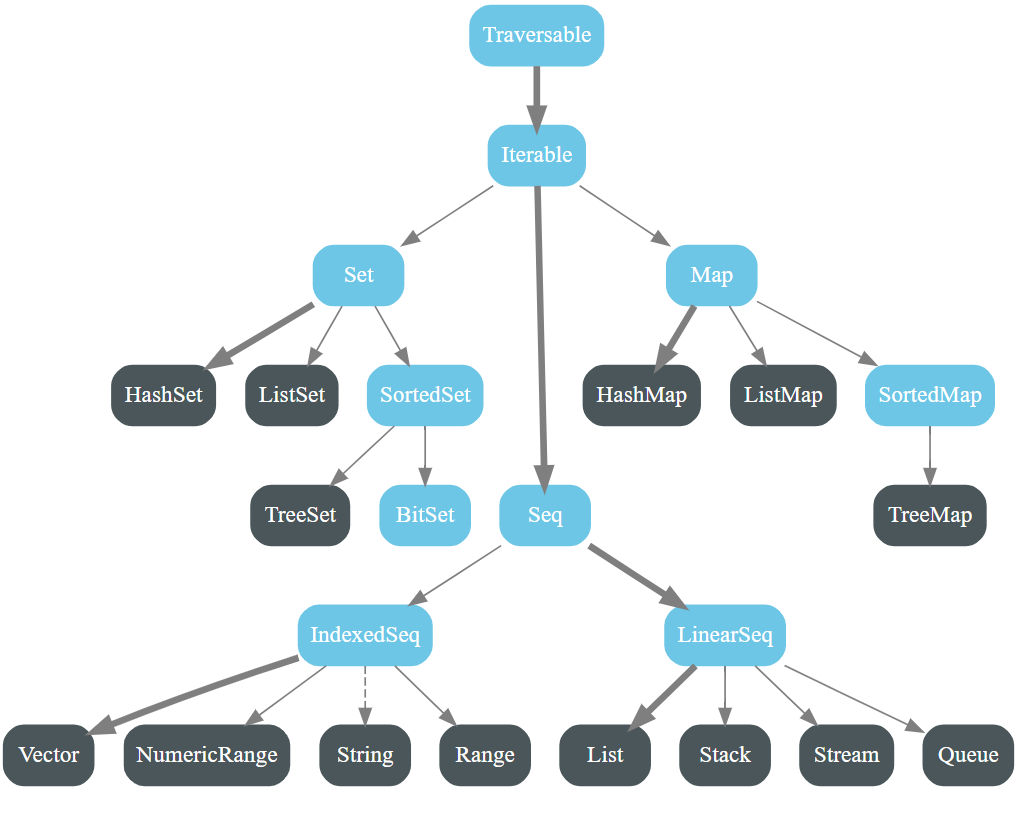
集合是存放多个同一种数据类型对象的容器，Scala的集合分为可变集合和不可变集合（可以模拟修改产生新的集合）

Traversable与Iterable为Scala的特质（与Java的接口相类似）

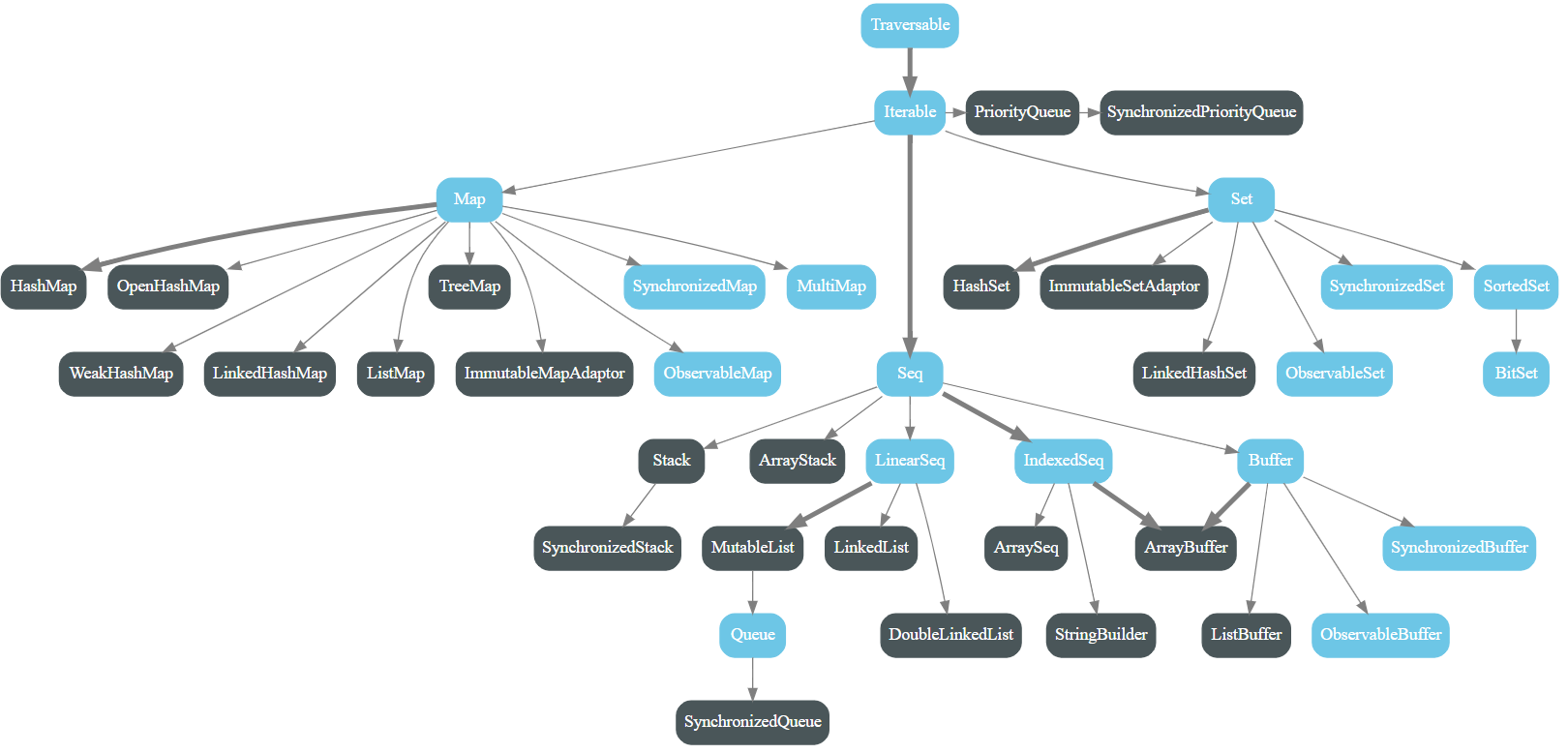
Scala的集合



Scala的不可变集合



Scala的可变集合



可变集合需要引入可变集合的类，import scala.collection.mutable.集合名类名Buffer

定长数组：

val 数组名=Array(值[,值])

val 数组名=Array[类型](值[,值])

val 数组名=new Array[类型](数值) 定义指定类型长度的数组

数组名(下标)访问数组指定元素，下标从0开始

变长数组：

var 数组名=ArrayBuffer(值[,值])

var 数组名=ArrayBuffer[类型](值[,值])

var 数组名=new ArrayBuffer[类型](数值) 定义变长数组，初始的数组长度为0，由于是变长数组数值无意义可省略

数组名.map(遍历的元素名=>表达式)与数组名.map(使用\_代替遍历的元素的表达式)等价

数组名.filter(过滤的元素名=>表达式)与数组名.filter(使用\_代替过滤的元素的表达式)等价

定长Map集合：

val Map集合名=Map(键key->值[,键key->值])

val Map集合名=Map((键key,值)[,(键key,值)])

Map集合名(键key)获取指定键key的值

元组是可以包含不同种类型的对象的集合

val 元组名=(值[,值]) 定义元组

元组名.\_数值 取出指定元组的第数值个元素，数值从1开始，元组中的元素不可修改

val 元组名,(元素名[,元素名])=(值[,值]) 定义的元组可以通过元素名访问对应的值

val 元组名=new Tuple数值(值[,值]) 创建指定数值个数的元组，元组最大个数为22

元组名.productIterator.foreach(变量名=>表达式)遍历指定元组

数组.zip(数组名) 指定的数组与指定的数组名进行拉链操作形成新的元组

数组.zipAll(数组名,变量,变量名) 指定的数组与指定的数组名进行拉链操作形成新的元组，数组与数组名的长度不一致用变量和变量名的值代替

元组名.unzip 解开拉链形式集合类型的元组

Nil是没有元素的空List的集合

Set集合（元素无序不重复的）：

val Set集合名=Set(值[,值])

集合常用的函数：

sum/max/min 集合元素的所有值相加和/最大元素/最小元素

filter(变量名=>表达式) 过滤集合中所有符合表达式的元素

flatten 变平或压平，将集合中包含的所有集合类型的元素中的元素整合在一起形成新的集合

map 对集合中的元素进行映射遍历处理

flattenMap 对集合中的元素进行映射遍历变平处理

forall(变量名=>表达式) 遍历集合中的所有元素是否全部符合表达式

foreach(变量名=>表达式) 遍历集合中的所有元素进行表达式操作

reduceLeft(\_操作符\_) 集合中的元素从左向右将第一个元素与其后一个元素做操作符操作后的结果再与后一个做操作符操作，以此类推直到所有元素都做完操作返回结果

reduceRight(\_操作符\_) 集合中的元素从右向左将第一个元素与其前一个元素做操作符操作后的结果再与前一个做操作符操作，以此类推直到所有元素都做完操作返回结果

foldLeft(初始值)(\_操作符\_) 集合中的元素从左向右将初始值元素与第一个元素做操作符操作后的结果与后一个元素做操作符操作，以此类推直到所有元素都做完操作返回结果

foldRight(初始值)(\_操作符\_) 集合中的元素从右向左将初始值元素与第一个元素做操作符操作后的结果与前一个元素做操作符操作，以此类推直到所有元素都做完操作返回结果

Scala的类一般形式：

class 类名 {//类名默认类型为public

val 属性名:属性类型=初始化值//系统会自动生成get方法

var 属性名:属性类型=初始化值//系统会自动生成get和set方法（系统生成的get和set方法名默认与属性名一致的方法，系统自动生成的get和set方法不能覆盖重写但可以自定义get和set方法名的方法）

private var 属性名:属性类型//系统会自动生成private修饰的get和set方法

private[this] var 属性名:属性类型=初始化值//系统不会生成get和set方法，只有当前对象可以访问该属性

def this(参数名:参数类型[,参数名:参数类型]){//自定义辅助构造函数，类默认有自带的无参主构造函数this()，主构造函数会执行类中定义的所有语句

this()//自定义辅助构造函数需要调用主构造函数或其他辅助构造函数

}

}

class 类名([var] 属性名:属性类型) {//带有参数的主构造函数，属性名会作为类的属性并生成get和set方法，[不带var修饰属性名仅作为主构造函数的参数不作为类的属性]

def this(参数名:参数类型[,参数名:参数类型]){//自定义辅助构造函数需要调用带有参数的主构造函数或其他辅助构造函数

this(属性名)

}

}

class 类名 private () {//定义私有的主构造函数

def this(参数名:参数类型[,参数名:参数类型]){//自定义辅助构造函数需要调用私有的主构造函数或其他辅助构造函数

this()

}

}

object 类名 {//单例对象，相当于Java中的静态类，可以在单例对象中定义工具函数和常量，单例对象不能带参数，单例对象的类名与其他类名重复存在时，单例对象成为同类名的类的伴生对象，同类名的类成为单例对象的伴生类，伴生对象与伴生类之间可以使用伴生对象或者伴生类的类名互相访问彼此私有属性或者方法，伴生对象需要与半生类定义在同一文件中

}

println(s"$参数名") 在输出信息双引号之间使用$参数名是获取指定参数名的参数值

object 类名 {//主方法main一般定义在单例对象中使用

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

}

}

object 类名 extends App{//主方法main也可以在继承了应用程序类App的单例对象中使用，在应用程序对象中直接写应用程序逻辑不用定义main方法

}

object 伴生对象类名(属性名:属性类型[,属性名:属性类型]) {

def apply()=new 伴生类类名(属性名:属性类型[,属性名:属性类型])//伴生对象的注入方法，使用val 对象名=半生类名(属性名:属性类型[,属性名:属性类型])不用new可以通过伴生对象的注入方法进行创建伴生类的对象

def unapply(arg: 伴生类类名): Option[(属性名:属性类型[,属性名:属性类型])] = {//提取指定伴生类的属性值

}

}

class 子类名(override val 属性名:属性类型[,val 属性名:属性类型]) extends 父类名(属性名){//子类继承父类的所有属性和方法，重写父类的非抽象方法或属性需要使用override，重写父类的抽象方法可以不使用override，final修饰的类、方法和属性不能被继承或重写

}

对象名.isInstanceOf[类名] 判断指定对象是否属于指定的类

对象名.asInstanceOf[类名] 将指定的对象转换成指定类的对象

classOf[类名] 获取指定类的信息

abstract class 类名 {//定义抽象类，抽象类的抽象属性和方法可以不初始化值和实现，类只能继承一个超类但可以实现多个特质（类似Java中的接口）

var 属性名:属性类型

def 方法名:返回类型

}

泛型：

[B<:A] UpperBound上界：B类型的上界是A类型即B类型的父类是A类型

[B:>A] LowerBound下界：B类型的下界是A类型即B类型的子类是A类型

[B<%A] ViewBound：B类型要转换成A类型，需要一个隐式转换函数

[B:A] ContextBound需要一个隐式转换值

[-A,+B]

[-A]逆变，作为参数类型，如果A是B的子类，那么C[B]是C[A]的子类

[+B]协变，作为返回类型，如果A是B的子类，那么C[A]是C[B]的子类

trait 特质名{//定义特质，泛型使用[]定义，特质中可以有抽象的方法和实现的方法

}

class to extends 特质名 with 特质名 {//类继承一个特质使用extends，继承多于一个特质后用with，类继承特质需要实现特质中的抽象方法

}

var 对象名=new 类名 [with 特质名] 在创建类对象的时候也可以加入多个特质，特质的顺序从最右边的特质向左执行

case class 类名(属性名:属性类型[,属性名:属性类型])//定义样例类，样例类默认带有apply方法，构造函数的参数默认是public val修饰，样例类没有参数也需要带()，样例类对象的比较是基于对象中的值或者结果比较不是基于对象的引用

val 对象名=样例类对象.copy(属性名=属性值[,属性名=属性值])//对样例类对象进行复制，复制的对象与样例类对象比较是相等的，可以不完全复制包含的属性

模式匹配（类似Java中的switch）

匹配的对象 match{//匹配条件中的常量名默认大写，小写会被匹配默认为变量名，变量名的匹配会被匹配对象重新赋值即总是匹配成功

[case 匹配条件=>表达式]//不需要break语句，默认break

case \_=>表达式//\_相当于Java中的default，必须有默认default的匹配

}

Scala的通配符为\_表示任意一个字符

样例类匹配：匹配的对象为多个样例类继承同一个超类的类对象，匹配条件为每个样例类的类信息

类型匹配：匹配条件为Scala的值类型

Random.nextInt(数值) 获取数值内的随机数

val 函数名=new PartialFunction[参数类型,返回类型] {//定义偏函数，可以在调用函数时通过isDefinedAt方法查看是否符合函数条件

override def isDefinedAt(参数名: 参数类型): 返回类型 = 判断条件

override def apply(参数名: 参数类型): 返回类型 = 函数体

}

val 函数名:PartialFunction[参数类型,返回类型] ={//简洁定义偏函数，自动将判断条件封装到isDefinedAt方法中

case 参数名:参数类型 条件判断=>函数体

}

val 函数名=偏函数名 [orElse 偏函数] 组合多个偏函数成一个偏函数

val 函数名:(参数类型=>返回类型)=偏函数名 [andThen 偏函数] 串联多个偏函数成一个函数流水线

sealed 类名{//定义密封类或特质，密封类的子类只能定义在密封类的定义文件中，密封类子类的子类不受密封类的限制可以在非密封类的定义文件中定义，避免继承的滥用和模式匹配（匹配的对象为多个类继承了同一个密封类的类对象，匹配条件为密封类的子类的类信息，不需要\_）

}

Option类是密封的抽象类，有Some和None（类似Java中的Null）两个子类，Option通过模式匹配避免Java中的空指针异常，数据类型不确定或者返回值不确定可以通过Option类进行匹配避免空值

Map对象.getOrElse(键值key,默认值)通过Option类获取Map对象中指定键值key的值比Map对象 (键值key)安全不会返回空值

字符串插值器是Scala2.10版本之后创建字符串的机制，允许字符串中存在变量

val 字符串名=s"${表达式}$变量名" 可以将字符串中${表达式}和$变量名以及特殊字符转换为相应的值

val 字符串名=f"$变量名%数值.数值f" 可以将字符串中$变量名转换成相应的值并格式化

val 字符串名=raw"a\nb" 可以将字符串中$变量名转换为相应的值但不转换特殊字符

val 资源名=Source.fromFile("文件路径名") 获取指定文件路径名的资源

val 资源名=Source.fromURL("URL地址") 获取指定URL的网络资源

val 迭代器名=资源名.getLines() 将指定资源按行转换成迭代器

val 迭代器名=资源名.buffered 将指定资源按字符转换成迭代器

迭代器名.hasNext 查看迭代器是否有下一个值

迭代器名.head 查看迭代器当前位置的字符

迭代器.next() 迭代器后移一个值

资源名.close() 将指定资源关闭

val 资源写对象名=new PrintWriter("文件路径名") Scala封装Java的写操作，可以调用Java的写对象进行写操作

资源写对象.close()

val 正则表达式名="正则表达式规则".r 定义正则表达式

val 正则表达式名=new Regex("正则表达式规则") 定义正则表达式

val 正则表达式名="""正则表达式规则""" 定义可以包含斜杠引号等特殊符合的正则表达式

正则表达式.findAllIn("字符串") 返回字符串所有匹配正则表达式的迭代器

val 变量名=正则表达式.findFirstIn("字符串") 返回字符串首个正则表达式匹配的内容

val 变量名=正则表达式.findPrefixOf("字符串") 查看字符串是否能匹配正则表达式

val 变量名=正则表达式.replaceFirstIn("字符串","替换字符串") 将字符串中首个正则表达式匹配内容替换为指定字符串

val 变量名=正则表达式.replaceAllIn("字符串","替换字符串") 将字符串中所有正则表达式匹配内容替换为指定字符串

高阶函数：函数的参数或返回值为函数

集合对象.map(函数名或匿名函数或带\_表达式) 高阶函数map可以将集合对象中的每个元素作为参数传递个指定的函数

闭包：函数的返回值依赖于函数外的一个或者多个变量即函数体引用了函数外部定义的变量，函数定义中引用的外部变量值会在函数调用后做相应改变

柯里化：将多个参数的方法变成接收一个单一参数的函数，返回一个接收余下参数的新函数

def 方法名(参数名:参数类型[,参数名:参数类型]):返回值类型=方法体 对应的柯里化方法为def 方法名(参数名:参数类型)[( 参数名:参数类型)]:返回值类型=方法体

柯里化的过程：

def 方法名(参数名:参数类型):单一参数函数的参数类型=>单一参数函数的返回类型=[(参数名:参数类型)]=>方法体

val 单一参数函数名=方法名(参数值)

val 方法结果名=单一参数函数名[(参数值)]

系统中柯里化的方法有集合对象.foldLeft(初始值)(带\_表达式)

Scala方法可以嵌套，Scala方法有多态性（类似Java的泛型）传入的参数不确定类型和返回的结果也不确定类型，但可以使用[泛型名]指定一个泛型

隐式转换：Scala中存在的类型默认转换

implicit class 类名(属性名:属性值){//定义隐式类，隐式类只能定义在类、Trait和Object内部，同一作用域隐式类不能与其他方法、属性和对象重名，使用隐式类进行隐式转换时需要引入隐式类import 类路径名.\_

}

implicit class 类名(属性名:属性值)(implicit 属性名:属性值){//隐式类的构造函数只能有一个非隐式参数，但可以将属性名分开进行定义

}

递归方法需要指定返回类型

implicit def 方法名(属性名:属性类型[,属性名:属性类型])=方法体 定义隐式转换函数

编译不通过是会自动寻找隐式转换，找不到相应的隐式转换会报错，类型与期望的类型不一致、对象访问一个对象本身没有的方法、对象调用的方法的参数与对象对应类的方法对应参数不一致编译器会自动进行隐式转换

def 方法名(属性名:属性类型[,属性名:属性类型])(implicit 属性名:属性类型)=方法体 定义带有隐式参数的方法，调用隐式参数的方法时，方法作用域内需要有对应隐式参数类型的对象

Actor：并行计算模型，并行是多个任务真正意义上的同时进行（计算机多核时才能并行），并发是多个任务宏观上是同时进行实际是任意时刻只有一个任务进行（计算机单核同时运行多个任务），Actor可以接收消息到自己的消息队列（先进先出）中和处理消息队列中消息（创建新的Actor、指定下一个消息的处理和给其他Actor发送消息），Scala的新版本Actor放入了Akka中

任务调度：基于线程（线程有相应任务时进行执行，没有任务时阻塞状态），基于事件（有任务时调用需要的线程进行执行，没有任务时将线程挂起操作）

Java多线程编程中拥有共享数据锁模型使用synchronized进行程序加锁，线程内部代码顺序执行，存在死锁问题

Scala的Actor编程中无共享数据无synchronized加锁，Actor内部代码顺序执行，不存在死锁问题

创建Actor需要引入相应的包import scala.actors.Actor

class 类名 extends Actor{//自定义Actor的类需要继承Actor，类似Java中的Thread

override def act(): Unit = {//实现Actor的抽象方法act()，类似Java中的run()

while (true){

receive{//接收消息，进行匹配做出相应响应

[case 消息匹配条件=>相应响应的快表达式]

}

}

}

}

var 对象名=new 类名 创建自定义Actor对象

对象.start() 启动Actor对象

val 返回值变量名=对象 !/!?/!! 消息 传递指定消息到指定Actor对象,发送无返回值的异步消息/等待返回值的同步消息/有返回值Future[Any]的异步消息，异步消息不阻塞程序继续执行，同步消息阻塞程序必须完成消息的响应才能继续执行

class 类名 extends Actor{//自定义Actor的类需要继承Actor，类似Java中的Thread

override def act(): Unit = {//实现Actor的抽象方法act()，类似Java中的run()

loop{

react{//react比receive更高效实现线程的重复利用，类似Java中的线程池

[case 消息匹配条件=>{

相应响应的快表达式

sender ! 返回消息值 Actor对象处理完消息响应时返回发送消息者的返回信息

}]

}

}

}

}

消息的类型可以是任意的一般定义消息类型为样例类

对象.isSet 检测Actor对象处理消息的返回结果是否可用

Akka：

Spark的RPC是通过Akka类库实现的，Akka用sca1a语言开发，基于actor并发模型实现

Akka具有高可靠、高性能、可扩展等特点，使用Akka可以轻松实现分布式RPC功能

Actor是Akka中最核心的概念，它是一个封装了状态和行为的对象，Actor之间可以通过交换消息的方式进行通信，每个 Actor都有自己的收件箱(Mailbox)，通过Actor能够简化锁及线程管理，可以非常容易地开发出正确地并发程序和并行系统

Actor的特性：

提供了一种髙级抽象，能够简化在并发(Concurrency)/并行(Para1e1ism)应用场景下的编程开发，提供了异步非阻的、高性能的事件驱动编程模型，超级轻量级事件处理(每GB堆内存几百万Actor)

ActorSystem：

在Akka中，ActorSystem是一个重量级的结构，他需要分配多个线程, ActorSystem通常是一个单例对象，ActorSystem可以创建多个Actor

Actor：

在Akka中，Actor负责通信，在Actor中一些重要的生命周期方法：

prestart()方法:方法在 Actor对象构造方法执行后执行，整个Actor生命周期中仅执行一次，receive()方法:方法在Actor的preStart方法执行完成后执行，用于接收消息，会被反复执行