李振超 18603071634

github

https://github.com/zhenchao125/scala1128.git

git下载镜像

https://npm.taobao.org/mirrors/git-for-windows/

前言：

Scala以实用主义为指导，面试问得不多，特有的东西如柯里化、闭包是面试重点。

不用太纠结底层，会用、会看代码就可以。

一、介绍

scala最重要的：

函数式变成

面向对象

集合

模式匹配，很有用

学scala的目的

为了更好地使用spark、flink，kafka等大数据组件。

1.与Java的关系

- 都编译为.class字节码文件
  - scala编译器，将scala代码编译成.class字节码文件。
- 都运行在JVM上
- SDK软件开发工具包，可以无缝对接
  - 可以使用Java的部分类库，但是基本上，我们能在Java上用到的，scala都可以用
  - scala的特有类库的

2.动态语言和静态语言

2.1静态语言

需要进行编译

例如：c --编译--> 可执行文件 --> 操作系统

使用IDEA等编程软件，可以进行错误提示。

2.2动态语言

边解释边执行

例如：js/bash --> 操作系统

所以语法错误，只能在执行到该行代码时，才可以被发现。编程过程中，没有错误提示。

3.为什么选择Scala?

目的：为了更好地使用spark、flink，kafka等大数据组件。

scala的优点：

1. 函数式编程
2. 不变性，例如数组长度不变，多线程不容易出现bug

二、环境搭建

特别注意：scala-2.11.8.zip，解压文件目录，不能有任何中文，且不出现空格。

lib，scala-library.jar，重要的。

1.windows配置

- 配置scala前提：先配置好Java
- 解压scala压缩包
- 配置环境变量
  - %SCALA_HOME%，添加scala文件路径
  - PATH=%SCALA_HOME%/bin

repl交互式解释器: read eval print loop

读取代码，执行代码，打印结果，循环代码

2.在IDEA中的使用

2.1添加开发插件

1.plugins，添加scala开发插件。安装完后重启IDEA。

添加成功的标志，创建maven工程，在external libraries，能看到scala的类包。

2.添加框架支持

在项目处，右键，选择add frameword support

2.2看反编译文件的方法

1.用jd,gui，但是有bug，会将`$`翻译为`.`

2.用IDEA的decompile自带的，scala to java插件。

scala to java，可将scala代码，转换成java代码

2.3编译文件的解析

创建Object类，编译后会生成2个类

（1）Scala中如果使用object关键字声明类，在编译时，会同时生成两个类：当前类，当前类$

（2）使用当前类$的目的在于模拟静态语法，可以通过类名直接访问方法。

（3）Scala将当前类$这个对象称之为“伴生对象”，伴随着类所产生的对象，这个对象中的方法可以直接使用。

```
//Hello类
//1. object在底层会生成两个类 Hello , Hello$
//2. Hello中有个main函数，调用 Hello$ 类的一个静态对象 MODULES$
public final class Hello
{
    public static void main(String[] paramArrayOfString)
    {
        Hello$.MODULE$.main(paramArrayOfString);
    }
}


//Hello$类，伴生类
//有static代码块
public final class Hello$
{
    public static final  MODULE$;

    static
    {
          new ();
    }
```

```
    public void main(String[] args)
    {
          //3. Hello$.MODULE$. 对象是静态的，通过该对象调用Hello$的main函数
          Predef..MODULE$.println("hello,scala");
    }

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

三、变量与数据类型

1.变量和常量

1.1变量赋值

```
var a: Int = 100
//可以进行类型推导，不用加: Int
var a = 100
```

注意：

1.声明的时候，必须进行初始化，不能延后初始化。

2.声明变量的习惯：当使用到该变量时，再进行声明

1.2常量赋值

```
val a: Int = 100
```

注意：

1.声明的时候，必须进行初始化，不能延后初始化。

2.不能重新赋值

实际开发中，能用常量的地方，绝对不要用变量

1.3类型推导（疑问）

1. 如果在声明的时候没有指定类型, 则会根据初始化的值自动进行类型的推导

2. 不要理解成动态类型。
3. 类型推导不是万能, 有些情况不能推导。什么情况不能推导？（疑问）

4.在if判断中，类型推导是分支语句的返回值的共同父类

例如:

```
val a: Double =10

if (m > n){
  Math.spar
} else {

}
```

2.标识符的命名规范

1.按照java的规范(数字字母)

- 下划线的使用要注意, 在scala中下划线有很多特殊的含义

2.可以使用所谓的的运算符作为标识符

- `+-*/` ...
- 至少使用2个运算符作为标识符（只使用1个运算符，虽然可以进行赋值，但是会产生歧义和使用困难）
- 运算符不能和字母混用（原因，可参考运算符）

3.如果有必要, 其实还可以使用任意的字符作为标识符

- 需要用反引号括起来
- 例如，用空格作为标识符，需要用飘号`

```
//需要用飘号 ``
val `      ` = 20
println(`      `)
//打印结果，20
```

3.字符串输出

3.1输出方式

格式化输出

printf用法：字符串，通过%传值。

```
// 1. 使用java的输出
System.out.println("abc");
// 2. scala的输出
println("ahc")
// 3. 格式化输出
val a = 20
//        println("a = " + a)
// 参考的传统的c语言
printf("a = %d %s %.2f", a, "1128班", math.Pi)
```

3.2字符串插值

3.2.1 s插值（重要）

作用：通过$引用变量和常量，不需要像java一样，用字符串拼接大法

```
val a = 30
val r = s"a = $(a * 10)"
printf("a = $a")

//a值乘以10
printf("a = $(a * 10)")
//等同于
printf(r)
```

3.2.2 raw插值

作用：直接可以写出转义字符，不需要再加转义字符的转义字符了

```
val r =raw"\r \n \t"
//等同于
val m="\\r \\n \\t"

println(r)
//结果为 \r \n \t
```

3.2.3 字符串模版

作用：不需要\n，也可以显示多行字符串

```
//例如SQL语句，会有大量的空行 \n，很影响可读性。
//可以使用3个引号，来实现多行字符串的显示
val sql = """
    select
    *
    from ads
    where id >10
"""

//打印效果
```

```
    select
    *
    from ads
    where id >10
```

`|` 用来标识每一行的开头位置，避免每行不对位的麻烦事。

在字符串后面，加上`.stripMargin`，省去`|`前面的字符。

注意，`|`前面的字符会被省去

```
val sql = """
    |select
    |*
    |from ads
    |where id >10
""".stripMargin

//打印效果
select
*
from ads
where id >10
```

4.读取数据

4.1直接用System.in

使用System.in直接读, 做一些封装

```
//1.用InputStreamReader读取单个字节
//2.用BufferedReader，字节缓冲流，来存储读取到的单个字节
//3.readLine，一行一行打印
val reader = new BufferedReader(new InputStreamReader(System.in))
val line: String = reader.readLine()
println(line)
```

4.2 Scanner

jdk 1.5之后, Scanner

```
println("请输入你的银行卡密码: ")
val scanner = new Scanner(System.in)
val line2: String = scanner.nextLine()
println(line2)
```

4.3 scala方式

可以直接打出提示语句，并等待数据键入。

```
//不再需要单独System.out.println("提示信息")，一行代码搞定提示信息，和读取键入数据
//StdIn.readXXX()
val line3: String = StdIn.readLine("请输入你的银行卡密码: ")
println(line3)
```

5.数据类型关系

5.1复习Java数据类型

Java将任意类型，分为了2类，基本数据类型和引用数据类型。

- 基本数据类型

  ```
  八大基本数据类型:
  byte short int long char(0-65535)
  float double
  boolean
  ```

- 引用数据类型(对象类型)

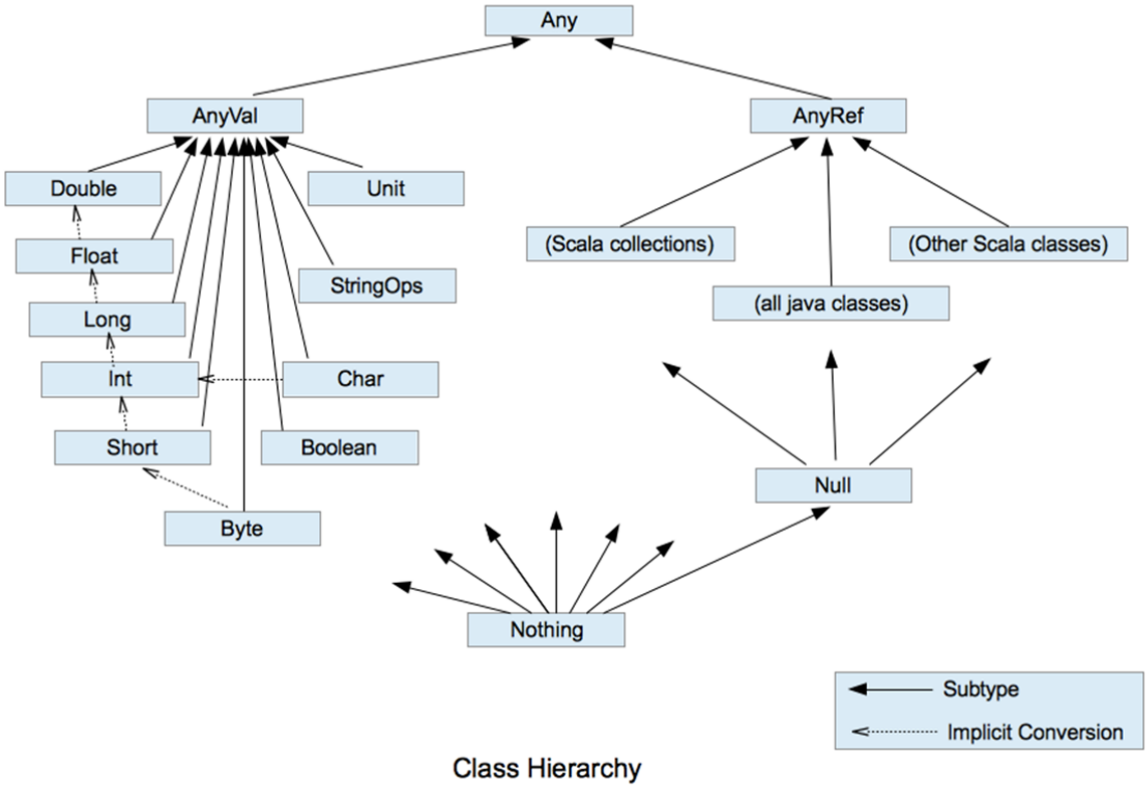
  所有的类型都是引用类型. (String)

  基本类型的包装类

Object，只是引用数据类型的共同父类。并不是任意类的父类。

Scala的Any，是所以任意类的父类，是Object的父类，层级更高

5.2关系图解释



Class Hierarchy

```
Any
    AnyVal   数值类型
        对应着java的基本类型

    AnyRef   引用类型
```

关注3个特殊的类

Unit

Null

Nothing

5.3 AnyVal

8个Java的包装类，加上StringOps、Unit

5.3.1 StringOps

可以当成String用，但是与隐式转换有关。

是String的1个增强类，是String++

与Java String类的区别

Java中的String，是final类，无法继承、增强。

Scala中的String，可以增强

5.3.2 Unit

Unit表示类型，值为`()`

作用：替换Java中的void。因为Scala中，没有void关键字了。

特点：给Unit类型的变量\常量赋值，都是无效的。值只能是`()`

`Int -> 1, 2 ...`

`Unit -> ()`

5.4 AnyRef

去掉AnyVal，剩下的都是AnyRef

相当于Java中的Object，以及所有的类

scala中的所有集合

5.4.1 Null

是所有AnyRef的子类，是AnyRef最底层的类

Null类型，只有1个值`null`

作用：替换Java中的null

`java: User a = null`

`scala: Null 类型, 它有一个值 null`

5.4.2 Nothing

是scala中，所有类型的子类，是所有类型最底层的类型

Nothing，没有任何对象

作用：一般用于辅助类型的推导。例如，当代码异常结束，则抛出值nothing

6. 类型转换

6.1自动类型转换

和Java相同

从范围小的->范围大的 才会自动转换

`Byte -> Short->Int->Long->Float->Double`

`Char -> Int->....`

6.2强制类型转换

注意：超过数值类型范围的值，强转会出现问题。例如128转Byte问题

```
val a: Short=128
println(a)
//打印结果 128
val b: Byte=a.toByte
println(b)
//打印结果 -128
//因为Byte的取值范围是-128~127，128超过取值范围。在原码、反码、补码的计算中，出现了问题。
```

6.3 String之间的转换

- 值-> string `.toString`
- String->值 `.toInt`

7.源代码

可查看`scala原码目录/src/library-aux`

查看`Any`代码

与Java不同的点

Java中，==和equals不等价

但是在Scala中，`==`就是`equals`

四、运算符

在scala中, 本质上是没有运算符的，运算符的本质其实是方法名

```
val a:Int =10
val b:Int =20
//当作运算符
val c:Int = a + b
//加法的完整写法
//+号就是1个方法名
val dLInt = a.+(b)
```

1.为什么方法名会写成运算符的形式?

运算符实际是方法名，那么为什么使用中可以写成不像方法名的形式呢?

1.1调用方法写法的省略

1.可以省略点`.`

2.如果参数只有1个或者0个，则可以省略`()`

变量名规则的原因——变量名不能是运算符和字符混合。

1.2什么时候可以成为运算符?

`.+()`，由于调用方法写法的省略，变成`+`

1.3方法和运算符的区别

1. 调用方法必须有`.`, 运算符是省略点的
2. 调用方法是没有优先级. 就是按照调用顺序来执行
3. 运算符有优先级，写成方法形式则无优先级

1.4运算符优先级

运算符是否有优先级，取决于写法

- 方法形式的写法，没有优先级

```
var a = 10.+(4).*(10).-(5)
//结果为136，从左到右计算
```

- 运算符形式的写法，有优先级

```
var a =10+4*10-5
//结果为45，与java运算符优先级相同
```

结合性，左结合、右结合

2. 3个判断相等

2.1复习java的相等于

==，比较hashcode

equals，可以重写方法，比较`值`

有区别

hashcode和内存地址，不是一回事

System.identityHashCode，是原始的hashcode，是用**内存地址**来作为值计算的

String类型的对象，用hashcode计算时，计算的实际是**对象的值**，而不是对象的内存地址值。

2.2 scala的相等于

==与equasl，**等价**，比较`值`

eq，比较`地址值`

但是重写时，需要使用equasl，和java的判断相等规则相同

3. 逻辑运算符

与Java完全相同

4. 赋值运算符

赋值语句作为1个语句结构，它的值为Unit类型`()`

赋值号比较特殊，看不到定义方法

4.1 赋值运算符的返回值

java赋值语句，返回的是左边变量的值

```
int a =3
int b =5
System.out.println(a = b)
  //打印值为 5
```

scala赋值语句，返回的是Unit类型`()`

```
int a =3
int b =5
println(a = b)
//打印值为 ()
println(a)
//打印值为 5
println(a -= 1)
//打印值为()
```

4.2 自增、自减

Scala中没有`++`、`--`操作符，需要通过`+=`、`-=`来实现同样的效果

```
a += 1
a -= 10
```

5. 位运算符

一般用于算法开发，提升算法性能，其他的情况下用得不多。

笔试题

问题1：如何最高效的计算，2^10

方法：用位运算符

位运算规律：`1: 0000 0001` `2: 0000 0010`

结论：计算2^n，等同于`1 << n`

问题2：如何最快速地判断，1个数是奇数还是偶数

方法：按位与，判断最后1位是1还是0

位运算规律： 奇数`xxxx xxx1` 偶数`xxxx xxx0`

步骤： 判断奇数，(n & 1) ==1

6.运算符的其他差异

6.1没有三元运算符（疑问）

java中的三元运算符，有潜规则，笔试会出错（疑问）

java的三元运算符

```
判断条件 ? ture值 : false值
```

scala可用if替代

```
if(判断条件) ture值 else false值
```

五、流程控制

在scala中，任何的**语法结构**都有值。

例如，if else的值，是执行语句的最后一行的值。

例如，2个变量赋值，`a =b`的值是Unit类型，`()`

1.顺序

2.分支

2.1复习java

- if else
- switch

2.1.1 switch的问题

```
switch(值){
    case 常量:
        break;
    case 常量:

}
```

存在的问题：

- 限制太多
  - `值`的类型是有限制: `byte, char, int, short, enum, String(1.7新增)`
  - `case`后面必须是常量
- 需要解决`case`穿透问题
  - 忘记添加`break`, 则会导致`case`穿透

2.2 Scala分支

- if else
- 模式匹配

2.3 if else

值：if语句的值，是执行分支中的最后一行代码的值。

替换三元运算符

原写法：`(判断语句) ? true值 : false值`

现写法：`if(判断语句) true值 else false值`

例如

```
if(m > n){
  m
  n
}
//if的值为n

if(m > n){
  m
} else {
  n
}
//若判断条件为true，则if的值为m，反之为n
```

2.4 模式匹配（单独拿出来，重点记录）

函数式编程，都会有的

3. 循环

3.1 复习Java

- for
- while
- do... while

3.2 Scala循环

- while
- do ... while
- for

3.3 while（略）

与Java没有区别

值：循环语句的值，是Unit类型

3.4 do ... while（略）

与Java没有区别

3.5 for

不是1种循环，而是1种遍历。本质是，遍历1个“集合”中所有的元素。

虽然word笔记、以及大多数教材中，都把for作为循环来解释。但是从遍历，来解释能更清楚。

3.5.1 增强for循环

java写法：`for(循环值类型 c : 集合){}`

scala写法：不用写循环值类型

```
for(c <- string){
  println(c)
}
//c，是string字符串中的每1个字符
```

3.5.2 范围数据循环方式

例如，1-100所有值的集合

2种循环方式

- `A to B`，[A, B]
- `A until B`，[A, B)

```
val a = 0.to(99)
// 前闭后闭，[1, 100]
val b = 0.until(100)
// 前闭后开，[1, 100)
```

3.5.3 遍历

1-100集合的写法

```
for(i <- 0 to 100)
for(i <- 0 until 101)
```

3.5.4 按步长遍历

```
//第1种写法
for(i <- 0 to 100 by 2)
//第2种写法
for(i <- 0 to (100, 2))
//实际上，会生成2个集合，第1个集合是1-100，第2个集合是1-100步长为2

//倒写
100 to 1 by -1
//100到1，倒序
```

3.5.5 循环守卫

类似于java中的continue，满足了某种条件才可以执行后面的代码。保护式为true则进入循环体内部，为false则跳过。

优点：可以更加直观看到哪些值进入循环体。

```
for(n <- 1 to 100 if n % 2 == 0)
//上面语句中，保护式为if n % 2 == 0
//只有守卫（判断语句）通过的，才可以进入到循环中

//与java中的语句，等同于
for(n <- 1 to 100){
  if(n % 2 ==0){
    //很多逻辑，但仅仅对偶数才执行。
    //缺点：不能很好的看出哪些值可以进入到循环中
  }
}
```

3.5.6 退出循环的方式

没有break关键字。

退出循环的原理：抛出异常，用try...catch接收异常，执行后面的代码

使用`Breaks.break`抛出异常，`Breaks.breakable`来接收异常。

需要导入Breaks包：`import scala.util.control.Breaks._`，这样写之后，就不需要用`Breaks.break`了，只需要写`break`

```
try {
   for (i <- 2 until n) {
       if (n % i == 0) {
            isPrime = false
            throw new ....
       }
   }
} catch {
     case e =>
}

//这种写法很麻烦，需要try...catch
```

使用`Breaks.breakable()`

内部try抛出了异常，不用我们再手动写try...catch

```
breakable ( // 也是个方法，内部其实try了抛出的异常
    for (i <- 2 until n) {
         if (n % i == 0) {
            isPrime = false
             break // 抛出异常. 其实是一个方法，内部在抛异常
          }
      }
 )
```

提出：return也是抛出异常（疑问）

3.5.7引入变量

在很多情况下，定义的变量，如果没有指定var，那么都是val。（scala是不变性）

```
for(i <- 1 to 100; j = 10; k =30)
//虽然i一直在变化，但是i是val，每次都是1个新的val，也就是说i的地址值每次都会变
//j和k，都是val。但每次循环，地址值不会变。和i不同
//不能在循环体中给i/j/k赋值。
```

for推导式有一个不成文的约定：当for推导式仅包含单一表达式时使用圆括号，当包含多个表达式时，一般每行一个表达式，并用花括号代替圆括号

```
//简略写法
for (i <- 1 to 3 ; j = 4 - i) {...}

//约定写法
for {
    i <- 1 to 3
j = 4 - i
} { .... }
```

3.5.8 循环嵌套

所有的代码都在内循环

优点：可以突出循环中的代码逻辑

```
//2种写法
//九九乘法表为例
//多层for循环
for(i <- 1 to 9){
    for(j <- 1 to i){
         print(s"$j * $i = ${i * j}\t")
     }
    println()
 }

//循环嵌套：所有的代码都在内循环
for(i <- 1 to 9 ; j <- 1 to i){
     print(s"$j * $i = ${i * j}\t")
     if(i == j) println()
 }
```

3.5.9for推导-循环返回值

将遍历过程中处理的结果返回到一个数组中，使用yield关键字

```
val s: IndexedSeq[Int] = for (i <- 1 to 10) yield i * i
//for遍历[1,10]，并将每个值计算平方，存入到数组中。
```

3.5.9.1 yield关键字

作用：记住每次迭代中的有关值，并逐一存入到一个数组中。

与map作用类似，一进一出。进来1个`i`，出去1个`i*i`

使用要求：

- yield后面的代码，一定要可以返回1个值。
- 可以实现多行代码，但最后一行一定要能返回1个值。
- 一般的，yield后面的代码逻辑不复杂，不推荐写多行代码（业务代码）

六、函数式编程

0.函数与方法的区别

函数只能定义在方法中

0.1定义方式不一样

1.方法定义

- 有`def`
- 只有具名方法，没有匿名方法
- 可以重载/重写

```
def 方法名(参数类别): 返回值类型 = { //方法的实现 }

def foo(a: Int, b:Int) = {
     println("xxx")
        }
```

2.函数定义

- 没有`def`
- 有具名函数，也有匿名函数
- 不能重载/重写

```
(参数列表) => { //函数体 }
//匿名函数
(a: Int, b:Int) => a + b
//具名函数，f就是函数名
val f2:(Int, Int) => Int =(x: Int, y: Int) => x+y
```

0.2省略规则不一样

使用时，圆括号是否可以省略

1.方法的使用

- 在只有1个以下的参数时，可以省略圆括号()

```
// Object01是对象名，foo2是该对象中的方法。那么在foo2只有1个参数时，可以有以下3种写法
Object01 foo2 10 //对象名 方法名 参数
Object01.foo2(10) //对象名.方法名(参数)
this foo2 10 //对象名用this替代 方法名 参数
```

2.函数的使用

- 不能省圆括号。
- 省圆括号后，调用函数，还是调用的自己，并没有运行。

```
// f1是函数，则调用的不能省略圆括号
val f1 = () => println("无参函数...") //定义了1个具名函数
f1() //不能省略圆括号
f1 //这是具名函数f1本身，并不能执行函数体内的代码
```

0.3使用规则不一样

1.函数，可以作为值传递和作为返回值返回。但方法不行。

换句话说，在给高阶函数（方法）传递参数的时候，**只能传函数**，不能传方法。

```
//例如
val f1 = (a: Int, b: Int => Int) =>  { println(a);println(b)} //高阶函数
val f2 =(c: Int) => c +15 //f2函数
f1(3,f2) //函数作为参数
def f3(a: Int) :Int = a + 15
f1(3,f3) //方法作为参数，理论上是不可以的
//但是实际使用中，也可以通过编译并运行。原因见下面，scala编译器优化问题
```

但是实际使用中，为什么也可以把方法作为方法的参数值传递呢？因为scala把方法转成了函数。

2. scala编译器优化，方法自动转成函数

在使用中，**不产生歧义**的情况下，scala会自动的根据需要把方法转成函数

转换语法：在方法名后加`_`，可以转成函数。

```
方法：add10
函数：add10 _
```

3.scala自动方法转函数，有歧义的情况

```
main {
  high1(abc) //此时会报错，有歧义。歧义的原因：scala不知道是否要不要将abc转成函数。
//歧义1：abc是转换成函数传入high1方法中
//歧义2：abc执行后得到的值传入high1方法中
  high1(abc()) //此时也会报错，因为abc方法，在定义时省略了圆括号，所以使用时也不能省略
  high1(abc _)//可以执行，此时方法转成了函数
}

def abc={println('abc')}
def high1(op: () => Unit) =op
```

1.定义函数（方法）

函数和方法的概念（淡化标记）（需要补充）

讲课中淡化了函数和方法的区别，认为函数=方法。但是函数和方法在某些方面是有区别的，例如重载和重写，只在方法中有，在函数中没有。

1.1 定义语法

与java有区别

```
def add(a: Int, b: Int): Int = {
    //具体实现
}
```

1.2 返回值

没有return，则自动把最后一行的值返回

有return，返回return的值。代码执行到return，就结束函数。此时，返回值类型不能推导。

1.3 语法省略规则（疑问

1.在定义函数时，函数体只有1行代码，则可以省略`{ }`

```
def add(a: Int, b: Int) = a + b
```

2.在定义函数时，若返回值类型可以推断出来，则可以省略，scala编译器会根据最后一行得到的值的类型进行自动推导。

- 若函数中使用了return，则不能省略返回值类型，因为不能进行返回值的类型推导。

```
//可以进行自动推导
def add(a: Int, b: Int) = {
          a + b
      }

//无法进行自动推导，此时会推断为Nothing。需要写返回值类型。
def add(a: Int, b: Int) = {
          return a + b
      }
```

3.在调用无参函数时，没有参数，则可以省略`()`

4.在定义无参函数时，该函数没有参数，则可以省略`()`。

- 若定义无参函数时，省略`()`，则在调用该函数时，必须省略`()`。

```
//定义有参函数
def add(a: Int =100){
   a + 200
}

//定义无参函数，不省略()
def foo() = {
         println("foo...")
     }

//定义无参函数时，省略()
def foo1 = {
         println("foo...")
     }

//main函数
 def main(args: Array[String]) {
   //使用foo1函数，不能加()
   foo1

   //使用foo函数，可以加也可以不加()
   foo()

   //使用add，有默认值，但不可省略()
   add()
 }
```

5.在定义函数时，若函数返回值声明为Unit，不论函数内部return的值是什么，在使用该函数时，得到的返回值只能是Unit类型

6.若函数返回值是Unit，则可以省略返回值类型`: Unit`。若返回值不是Unit，则不能省略返回值类型。

```
//这样的函数称为“过程，纯函数”（疑问）什么是纯函数？好像这里记录有错误
def foo() = {
  //函数返回值Unit
}
```

7.如果不关心名称，只关系逻辑处理，那么函数名（def）可以省略

2. 其他特征

2.1 形参默认类型

参数默认为val，不能修改。若需要修改参数，则应该自己重新声明一个新的变量。

```
def foo(a: Int) = {
  println(a)
  a = 100 //错误，a默认为val
  var b = a //定义新的变量，去修改新定义的变量
  b = 100
}
```

2.2 函数内的函数

在函数内，可以定义函数。

- 需要注意，在函数中可以先使用内函数，再定义内函数。
- 但是若在内函数和调用内函数之间插入1行代码，则会出错（举例见下面的代码）。

所以注意：在函数内定义的函数，最好先定义，再使用，避免出错。

```
def add(a: Int){
  foo() //内部函数，可以先使用，再定义
  //var a: Int = 10  但是若增加这1行代码，则前面的foo()不可使用
  def foo(){   //
    println("foo...")
  }

  foo() //先定义，再使用
}
```

2.3 参数默认值

定义函数的时候，可以给函数设置默认值。

当传递参数的时候，没有给这个参数传值，则可以使用默认值。

利用这个特性，可以不用写方法重载。（注意，函数没有重载和重写，方法才有重载和重写）

```
def add(a: Int = 100, b: Int){}
//位置参数，可以使用默认值
def add1(a: Int, b: Int = 100){}
//使用默认值，需要使用命名参数
```

2.4 传递参数

传递参数时，默认使用**位置参数。**

一般的，**命名参数**和**默认值**会配合起来使用

```
def add1(a: Int = 100, b: Int) = a + b
// 位置参数，所有的语言中，默认都是按照位置来传参!!!
add1(100, b = 1000)
// 命名参数，就可以按照自己喜欢的顺序,想怎么传就怎么传
add1(b = 2000, a = 1000)
```

2.5 可变参数

定义时，可变参数必须写在函数参数的最后1位

传递参数时，不能**直接**传递数组给可变参数。但可以通过将数组展开，来给可变参数传参。（Java可以把数组传给可变参数）

可变参数，会被当作1个数组（集合）来使用

```
def foo(ss: Int*) = {}
//ss，是1个数组（集合）

val arr: Array[Int] = Array(1,2,3,4)
//arr:_*，可以展开数组，传递给可变参数。
foo(arr:_*)
foo(ss[0],ss[1],ss[2],ss[3])
//foo(arr) 不能直接传数组
```

使用技巧：

可变参数，可以传递0个或多个参数，那么怎样保证使用函数时，必须传递n个以上的参数呢?

例如：保证至少传递1个参数

```
def fool(firtst: Int, s: Int*) = {}
//在可变参数前面，加上1个参数first，即可保证传递1个以上参数
//n个参数，类推，加上n个参数即可
```

默认值，scala源码怎么做的？（疑问）

```
待补充
```

3. 高阶函数

更准确的说法，是高阶方法。更高大上的说法，是高阶算子。

定义：参数中有函数或者返回值是函数的函数就是高阶函数

判定方法：只看参数中的函数的参数类型和返回值类型是否正确

```
def foo(a: Int, f: Int => Int) = {}
//要求参数f是1个函数，且f函数的有1个参数，参数类型为Int，返回值为Int
//其中，当参数 f函数，只有1个参数时，可以省略括号()

def foo(a: Int, f: (Int, Int) => Int) = {}
//要求参数f是1个函数，且f函数的有2个参数，参数类型为Int，返回值为Int
```

3.1 匿名函数和具名函数

3.1.1 具名函数

有名字的函数

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

3.1.2 匿名函数

与具名函数比较，形式上的区别：将`=`，换为胖箭头`=>`

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

- `=>`前的部分，是匿名函数的参数列表副本
- `=>`后的部分，是匿名函数的实现

作用：一般用于向高阶函数传递参数

返回值类型: 一般开推导出来，不能直接写返回值类型。可以通过匿名函数的字面量，来规定返回值类型。

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

val f =(x: Int, y: Int) => x+y

val f2:(Int, Int) => Int =(x: Int, y: Int) => x+y
//"(x: Int, y: Int) => x+y" ，函数字面量，函数的符号表达
//"(Int, Int) => Int" ，函数类型
//"f2"，函数名
```

> **重要**：永远不要在匿名函数中使用`return`，会出现bug，很难解决

literal（字面量）：值的符号表达

3.1.2.1 匿名函数的传递

匿名函数，作为值存储在变量中

增加1个常量`val f`来接收匿名函数的返回值。此时就可以用`f`来使用该匿名函数了。

```
//前提：参数中有匿名函数的高阶函数
def calc(a: Int, b: Int, op: (Int, Int) => Int) = op(a,b)

//匿名函数的写法
(x: Int, y: Int) => x+y

//创建变量来接收匿名函数的返回值
//优点：相比上面的写法，可以在多个地方使用；
//
val f: (Int, Int) => Int = (x: Int, y: Int) => x + y

//传递f，可复用
calc(3, 5, f)
//传递匿名函数，不可复用
calc(3, 5, (x: Int, y: Int) => x+y)
```

3.1.2.2 匿名函数的省略写法

定义时匿名函数，省略参数类型的要求：有上下文环境，或者接收变量已经定义了参数类型

```
//完整的匿名函数字面量定义
val f: (Int, Int) => Int = (x: Int, y: Int) => x + y

//已经写死了变量f1的参数类型时，也可以省略匿名函数的参数类型
val f1: (Int, Int) => (x, y) => x + y
```

使用时匿名函数，省略参数理解，看下面代码中的（1、2、3）顺次理解到最精炼的写法`_`，不需要写参数。

```
//调用匿名函数的高阶函数
def calc(a: Int, b: Int, op: (Int, Int) => Int) = op(a,b)

def main(args: Array[String]): Unit = {
  //1.匿名函数的常用场景，给高阶函数calc传值
    calc(3, 5, (x: Int, y: Int) => x+y)
  //2.有上下文环境，匿名函数的参数类型可以省略，scala可以推导出来
    calc(3, 5, (x, y) => x+y)
  //3.如果参数只使用一次，则可以用 _ 来替换每个参数，此时 => 也需要省略。最精炼的写法。
  //第1个_，表示第1个参数（3）
  //第2个_，表示第1个参数（5）
    calc(3, 5, _ * _)

  }
```

3.2 参数式函数和返回值式函数

3.2 高阶函数的具体使用（疑问）

对集合的使用，并限定为Int类型数组。后面再扩展到泛型（疑问）

foreach

map

reduce

filter

3.2.1 foreach

作用：遍历数组，得到元素。对元素做一些操作（通过函数告诉foreach）

思路：对元素做操作，只有调用者知道要做什么操作。

```
def foreach(arr: Array[Double], op: Double => Unit) = {
        // 可以遍历，但是你得给我个函数，我遍历到元素之后，我去调用这个函数
        for (ele <- arr) {
            op(ele)
        }
    }
```

3.2.2 filter

作用：过滤出符合条件的元素，并返回这些元素。

思路：用循环守卫，yield返回符合条件的元素至新数组中

```
def filter(arr: Array[Int], condition: Int => Boolean) = {
        // for 推导
        for (ele <- arr if condition(ele)) yield ele
    }
```

3.2.3 map

作用：映射，一进一出。注意与foreach的用法区别开，区别点在于“映射”。foreach并不需要返回值，实现一一对应的映射。

思路：用`yield 映射操作`，返回新数组。

```
def map(arr: Array[Int], op: Int => Double): Array[Double] = {
        // for推导
        for (ele <- arr) yield op(ele)
    }
```

3.2.4 reduce

作用：聚合，多进一出。

思路：创建临时变量，保存循环聚合时的结果值。需要将上1次的聚合结果，和这1次的元素，进行操作。

注意：第1次的聚合，需要取第1个元素作为lastResult。那么for循环，也要改为`1 until arr.length`

```
def reduce(arr: Array[Int], op: (Int, Int) => Int): Int = {
        if(arr.length <= 0) return 0
        // 聚合操作
        // 表示上一次的聚合结果
        var lastResult = arr(0)
        for (i <- 1 until arr.length) {
            lastResult = op(lastResult, arr(i))
        }
        lastResult
    }
```

4. 函数柯里化

4.1 闭包（疑问）

定义：1个**匿名函数**和这个**匿名函数所处的环境**，理解匿名函数和匿名函数所处的环境，见下面的代码解释。

作用：闭包，可以延长外部局部变量的生命周期。相当于函数保存了数值。（之前，函数只能保存代码）可以实现在函数的尾部访问到函数的局部遍历

特点：同1个闭包，内部变量的生命周期是持续存在的。

```
//闭包
def foo(a: Int) = {
  (b: Int) => a + b
}
//函数柯里化
def foo(a: Int)(b: Int) = a + b
//注意和有1个参数列表的函数区别开
def foo1(a: Int, b: Int) = a + b

//拿到闭包，理解闭包
//匿名函数："(b: Int) => a + b"
//这个匿名函数所处的环境："foo函数，参数a=10"这个环境
```

```
val f: Int => Int = foo(10)
//执行闭包，延长了外部局部变量a的生命周期
f(20) //结果为 30，f闭包中变量a=10，会保存在该f闭包中。
//f(20) 等同于
foo(10)(20)
```

4.2 柯里化（疑问）

定义：将1个参数列表的多个参数，变成多个参数列表的过程，叫作对函数柯里化

```
//val和def都可以，因为scala中函数是一等公民，可以将函数复制给常量
//1个参数列表，接收多个参数的函数
val sum = (x: Int, y: Int, z: Int) => x + y + z

//只有1个参数的函数
//闭包
val sum1 = (x: Int) => {
    y: Int => {
      z: Int => {
        x + y + z
      }
    }
  }
//等价于
//柯里化函数，复制给常量sum2
val sum2 = (x: Int) => (y: Int) => (z: Int) => x + y + z
//柯里化函数sum3
def sum3(x: Int)(y: Int)(z: Int) = x + y + z
```

理论基础：闭包

具体应用：一般只会柯里化出来2个参数列表。

第1个参数列表，是必须填入的参数

第2个参数列表，可以不传入参数，使用隐式，如implicit

4.2.1 柯里化函数获取闭包

如果只写前1个参数列表，会报错，

1.写上函数的参数类型

2.未传入参数的参数列表补上_

（疑问）这里的问题原因，和解决方法的原理，与方法和函数的区别有关！（疑问）

```scala
def foo(a: Int)(b: Int) = a + b

val f1 = foo(3) //会报错
//柯里化函数获取闭包的方式
//1.将未传入参数的参数列表补上(_)
val f1 = foo(3)(_)
//2.写上函数的参数类型
val f1: Int => Int = foo(3)

// '_' 的用法
val f2 = foo(_) //等同于
val f2 :Int => Int => Int =(a: Int) => (b: Int) => a + b
```

4.2.2 柯里化函数有默认值

```scala
def foo(a: Int)(b: Int= 100) = a + b

foo(22)() //结果为122
foo(22) //错误，会报错
```

5. 控制抽象

Scala中可以自己定义类似于if-else，while的流程控制语句

5.1 名调用和值调用

5.1.1 值调用

定义：先计算参数表达式的值，再应用到函数内部

```scala
//举例1，传入参数的值，是“4+6”的计算结果，10
def add(a: Int) = a+2

add(4 + 6) //传递给add函数的参数，是10，也就是“4+6”计算后的值
```

5.1.2 名调用

定义：将未计算的参数表达式直接应用到函数内部

名调用的标志：函数的参数为`op: => Unit`，没有圆括号

一般如果接受的是一个无参的函数的情况，可以改成名调用

```scala
//举例1，传入参数的值，是“4+6”
def add(a: Int) = a+2

add(4 + 6) //传递给add函数的参数，是“4+6”
```

```
//举例2
def foo(a: () => Int) = {
    a()
    a()
    a()
}
// 使用名调用进行改造

def foo1(a: => Int) = {
    a
    a
    a
}

//使用时
//值调用，不能省略参数的圆括号()
foo(()=>{
      println("hhh")
      10
    })

//名调用，可以省略圆括号()，只写匿名函数a的代码即可
//这就是自定义while的原理，为什么自定义的while使用时可以像while一样写。
foo1{
      println("hhh")
      10
    }
```

5.1.2.1 自定义while（疑问）

while是1个柯里化函数

```
//while可以理解为1个这样的柯里化函数
while()({})

//自定义while，使用方式和while一样
def myWhile(condition: => Boolean)(op: => Unit) {
  if(condition) {
    op
    //再次调用myWhile时，是在调用myWhile(condition)的闭包，condition匿名函数的变量得到了保留
    myWhile(condition)(op)
  }
}

//不用柯里化的写法（疑问）


//为什么这样写是错的？会形成死循环
def myWhile(condition: => Boolean,op: => Unit) {
    if (condition) {
       op
      //再次调用myWhile时，condition匿名函数中的变量没有得到保留
       myWhile(condition,op)
   }
```

```
}
```

5.1.2.2 名调用，启动多线程

传递一段代码，让这段代码，在一个子线程中执行

```
def runInThread(code: => Unit) = {
    new Thread() {
      override def run(): Unit = code
    }.start()
}

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

  //main线程
        println(Thread.currentThread().getName)
  //子线程1
        runInThread {
            println(Thread.currentThread().getName)
        }
  //子线程2
        runInThread {
            println(Thread.currentThread().getName)
        }
}
```

6. 惰性求值

定义：对1个常量，当被声明为lazy时，直到第一次对此取值时才去求值。一旦计算一次后，以后再使用，就不用再计算了。

限制：只能用在`val`常量

```
//a 就会惰性求值
lazy val a = 10
```

lazy，是一种介于val和def的状况

- val，赋值时，直接执行
- lazy，赋值时，不执行，延迟至第一次使用才执行求值
- def，调用一次，执行1次

val与lazy，只是加载的时间不同

```
//赋值时就已经计算出结果
//在main函数外，这是1个object的常量，会先加载？（疑问）
val a = {
    println("a...")
    10
```

```
}

//赋值时未计算结果，延迟至第一次使用
lazy val b = {
    println("b...")
    20
}

//每次调用，计算结果
def c = {
    println("c...")
    30
}

def main(args: Array[String]): Unit = {
        // layz其实是介于 val和def之间的状况
  //就算不在main函数中，调用常量a，也会在运行该object时，计算val的值，并赋值。
  //打印出：a...
        println(a) // 10
        println(a) // 10
        println(b) //因为b是惰性求值，所以在这里对b取值时，才计算b的值
        // b...
        //20
        println(b) //运行1次后，就不再求值了
        //20
        println(c) //函数，每次都需要计算
        //c...
        //30
        println(c)
        //c...
        //30
    }
```

7. 递归

在函数式编程语言中，递归非常重要。很多算法，都会用到递归。

写递归的技巧：定义函数时，先写注释。一定要明确，这个函数干什么，每个参数的作用是什么。

注意：递归函数的返回值，不能类型推导。因为什么时候结束，不知道。

问题：普通递归，容易出现栈内存溢出

解决方法：尾递归。scala提供了一种尾递归优化，可避免栈内存溢出stackOverflow

7.1 尾递归

IDEA，提示会变成1个循环圈，与递归的提示暴风圈，不同。

原理：scala有尾递归优化，在编译时，会优化为迭代算法，循环。

注意java区别：java没有尾递归优化。

需求：需要有`累加器(结果累计器)`，**尾递归**的**难点**，就在**累加器的设计**上

```
 // 计算n!，n! = n * (n-1)!
//尾递归，其中add就是累加器，用于传递结果值给下一个迭代对象中。
@tailrec
def factorialTail(n: Int, add:Long): Long = {
    if(n == 1) add
    else factorialTail(n - 1, add * n)
}

//递归
def factorial(n: Int): Long = {
    if(n == 1) 1
    else n * factorial(n - 1)
}
```

8.部分应用函数

用占位符`_`来表示没有传入的参数，返回的是1个函数。实际上返回的是1个闭包。

作用：在现有函数的基础上，得到1个新的函数。

注意：

1.占位符，只能是`_`，不能加其他的东西，如`_ + 10`

2.如果匿名函数原封不动的返回，则不能化简为`_`，需要写为`x => x`

```
val f:Double => Double = Math.pow(_ , 2)
//可以直接用函数f，计算平方了。不用再每次写Math.pow(x ,2)
val f1:Double => Double = Math.pow(_ +10 , 2)//报错!
//因为部分应用函数中，占位符只能是 _，不能加其他的东西
```

如何判断是不是尾递归

1.用尾递归标签，`@tailrec`，检测是不是尾递归

2.IDEA提示为1个循环圈

函数相对与对象只封装了代码, 没有封装数据

语法糖?

函数式变成标配

高阶函数

闭包

柯里化

函数、方法，是有区分的，但实际上使用时，不用去区分

面向对象的理解

理解对象

用对象进行封装

对**数据**和**行为**的封装

理解类

为了更加方便的去封装对象

为什么要写代码、程序?

1.存数据

2.对数据做各种运算，得到想要的结果

面向对象

定义类

```
class User1
```

属性

属性只的默认初始化值:

- 数字 0
- 布尔型 false
- 引用型 null

scala的变量，在声明时必须赋值。没有默认值的变量属性，赋值为`_`

```
class User1{
    // 类的具体的内容
    // 只读属性
    val sex = "male"
    // 可读可写属性    _ 表示给name初始化默认值 null
    var name: String = _
    var age = 10
}

class User2(var name: String, val age: Int, sex: String) {
  def eat = println(sex)
}
```

属性的3种情况

1. var，可变的变量
2. val，不可变的常量
3. 无var/val，在类中被使用时，可以成为私有属性

```
class User2(var name: String, val age: Int, sex: String) {
  def eat = println(sex) //sex在类中被使用
}
```

构造器参数的特征

1.在类中定义的属性，默认为私有

2.私有属性，在编译时，会自动添加公共的getter和setter方法。

```
//可查看类编译后的成员，通过cmd，javap -p 类文件名（编译后的文件）
public class com.atguigu.scala1128.day04.obj.User2 {
  private java.lang.String name;
  private final int age;
  public java.lang.String name();   // getter
  public void name_$eq(java.lang.String);  // setter
  public int age();
  public com.atguigu.scala1128.day04.obj.User2(java.lang.String, int);
}
```

3.getter和setter的使用有约定

在访问属性的时候, 约定是访问公共的`getter`方法

```
println(user.name)
其实是访问的
public java.lang.String name();
```

在修改属性值的时候, 会默认访问`setter方法`

```
user.name = "zs"
其实访问的是
public void name_$eq(java.lang.String)
```

主构造中，若参数没有`val/var`，那么这个参数**在有些情况下**会成为私有属性，不会自动添加公共的getter和setter方法，会添加final的setter方法。

- 对"有些情况"的理解
    - 至少这个私有参数，在类的内部中，有被用到

scala编译器提供的setter和getter问题

scala自动提供的setter和getter，会有问题

问题：提供的getter和stter，不符合标准的java bean规范。会影响到一些java类的使用

标准的java bean写法getter和setter

```
java bean:
public String getName(){  }
public void SetName(String name){  }
```

由于scala的生态不完善，scala会大量的使用专门为java准备的那些类库，这些类库在底层一般要用到标准java bean规范（例如getter和setter）

此时，需要添加java的标准java bean

冲突：但是添加标准java bean，还要scala做啥?

解决：为了方便的得到标准java bean。scala提供了1个注解`@BeanProperty`，来解决标准java bean问题。添加了注解的参数，会自动添加标准java bean的getter和setter。

```
class User2(@BeanProperty var name: String, @BeanProperty val age: Int,
@BeanProperty sex: String)
```

注意：若参数没有var/val且在类内部也没有被使用，添加了注解@BeanPropety，也不会有该参数的getter和setter方法，只有内部使用了这个参数，才会有final getter的只读方法。

总结：什么时候要加`@BeanProperty`?

需要用到java类库时，加上注解。如果全是scala类库，就不需要加了。

构造器的特征

1.主构造器：默认有空构造器, 但是一旦有参构造方法时，就没有无参构造器了。

若需要无参构造器，则需要手动添加无参构造方法。

```
class User3(var name: String) {
    var age: Int = _
      //代码块
    println("hahaha")
    // 定义一个无参辅助构造器
    def this() = {
    // 注意: 首行必须是调用自己的主构造器
    this("lisi")
    }
}
```

在使用中，`对象名.属性名`，实际上使用的是方法，而不是直接取到的值。

构造器中的代码块，在scala编译后，会放入到类的构造器中

构造器重载

scala支持构造器的重载

构造器的分类

scala整体分为2种构造器

1.主构造器

紧跟着类名

2.辅助构造器

和主构造器构成了重载关系

功能相比比较弱

与主构造的区别

- 辅助构造器，仅仅只是1个函数
- 主构造器中的变量，可以成为属性，但辅助构造器种的变量则不行
- 辅助构造器，首行必须是调用自己的主构造器

```
// 定义一个无参辅助构造器
def this() = {

// 注意: 首行必须是调用自己的主构造器
this("lisi")
}

def this(age: Int) = {
    this("lisi")
    this.age = age
}
def this(a: Int) = {
    this()
}
```

理解scala的面向对象和函数式融合

怎样看 `var age:Int = _` ?

```
class User2(var name: String){
  var age:Int = _
}
```

这个问题，体现了scala将面向对象编程和函数式编程的融合。

- 从面向对象的角度，User2是1个类，age是里面的1个属性
- 从函数式编程的角度，User2，是1个构造方法，age是里面的1个变量

> 注意：scala方法的定义，有先后顺序
>
> 在主构造器重的方法，先定义的方法，不能调用后新定义的方法
>
> java的方法，没有先后顺序

包

包的声明

1. 支持和 `java` 一样的声明方式(基本这种使用)

   ```
   package com.atguigu.scala1128.day04.pack
   ```

2. 支持多个 `package` 语句(很少碰到)

   ```
   //以下2个package拼接
   package com.atguigu.scala1128.day04.pack
   package a.b

   class PackDemo {}
   ```

   以上的类，会放在 `com.atguigu.scala1128.day04.pack.a.b.PackDemo`

3. 包语句(很少碰到)

   ```
   package com.atguigu.scala1128.day04.pack

   package c{  // c其实是子包
       class A
     //class A编译后，会放在com.atguigu.scala1128.day04.pack.c，这个包下
   }
   ```

包的导入

1. 导入和`java`一样, 在文件最顶层导入, 整个文件的任何位置都可以使用(掌握)

```
import java.util.HashMap
```

2. 在`scala`中其实在代码任何位置都可以导入**(掌握)**

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

    import java.io.FileInputStream
    // 只能在main函数中使用
    val is = new FileInputStream("c:/users.json")
}
```

3. 导入类的时候, 防止和现有的冲突, 可以给**类**起别名

```
import java.io.{FileInputStream => JFI}
```

4. 如何批量导入**(掌握)**

```
import java.io._  // 导入java.io包下所有的类   (java是*)
```

5. 屏蔽某个类

   屏蔽批量导入中的某个类

   将屏蔽的类起别名`_`，取不到的名字

```
import java.io.{FileInputStream => _, _}  //屏蔽 FileInputStream
```

6. 静态导入

   java

   中的静态导入，只能导入静态成员

```
import static java.lang.Math.*

public static void main(){
    pow(2,3) //不需要写Math.pow()
}
```

   scala

   可以导入任意成员，因为scala没有静态概念

```
import java.lang.Math._
```

7. `scala`还支持导入对象的成员(掌握)

```
val u = new User
// 把对象u的成员导入
import u._
foo()
eat()
```

公共方法的处理

公共方法，如`Math.pow()`，java工具类中的静态方法。

`java`可创建工具类, 在工具类中写静态方法. 因为`java`中所有的方法都需要依附于类或者对象

`scala`可创建一个**包对象**Package Object的, 将来在这个包内, 使用包对内的方法的时候, 就像使用自己定义的方法一样。

条件：包对象的名字，要和包名相同

举例：在`com.atguigu.scala1128.day04`包中，创建了Package Object

```
package com.atguigu.scala1128.day04

package object pack {
    def foo1() = {
        println("foo...")
    }
    def eat1() = {
        println("eat...")
    }
}
```

在`com.atguigu.scala1128.day04.pack`包下所有的类可以直接使用这些方法`方法名`，可以不使用`对象名.方法名`。

默认导入

有3个默认导入的包

```
java.lang._
scala._
predef._
```

继承

面向对象的3大特征

封装

继承

多态

有了继承，自然就有多态

```
class A
class B extends A
```

方法的覆写（重写）

`java`中方法的覆写规则:两同两小一大

两同

- 方法名
- 参数列表

两小

- 返回值类型: 子类的返回值类型应该等于或小于父类的返回值类型
- 抛的异常: 子类的方法抛的异常要小于父类抛的异常

一大

- 访问权限: 子类要大于父类访问权限

在`scala`中遵守同样的规则!!!

与java的不同点

1. 在`java`中`override`注解(1.6)是可选的.
2. 在`scala`中, `override`是一个关键字, 必须添加

> 注意：和方法的重载，作区分

多态

定义：一个对象的编译时类型和运行时类型不一致

1.编译时类型，`=`左边

- 编译的时候, 是否可以通过要看编译时类型.

2.运行时类型，`=`右边

- 创建对象的时候，使用的类型就是运行时类型
- 运行的时候, 方法的具体表现要看运行时类型

结论：编译看左，运行看右

具体结果，看运行时类型

编译是否通过，看编译时类型

属性的多态（scala独有）

在java中，属性没有多态，方法才有多态

```
public static
```

在scala中，属性可以覆写，也具有多态（因为属性的本质），要加override

属性的本质：scala中属性，也是1个方法

```
class A {
  def b = {
    println("abc")
    10
  }
}

class B extend A {
override def b =1000 //正常的方法覆写了方法
override val b =1000 //此时，常量b覆写了方法b。常量覆写方法的要求：被覆写发方法不能有参数
}
```

属性覆写的规则：

val只能覆写val、没有参数的def

var只能覆写抽象var、抽象字段（属性）

继承的构造器处理

注意

1. 子类的辅构造器，必须先调用自己的主构造器，不能主动去调用父类的构造器
2. 只有主构造器才有权利去调用父类的构造器
3. super只能出现在普通函数中，不能出现在构造器中。

执行顺序

java

静态代码块 -> 构造代码块 -> 构造器

scala，没有静态概念

主构造器，辅构造器

动态绑定机制？什么东西（疑问）

抽象类

java中的抽象类

1.用`abstract`修饰类,就是抽象类

2.抽象类中可以有普通类中的所有成员

3.抽象类还可以有抽象方法

- 抽象方法：只有方法签名（返回值类型，参数列表，方法名），没有方法的实现

4.抽象类不能直接创建对象，必须创建子类的对象

scala中的抽象类

和java抽象类的特点一样，使用上与java没有区别，按着java的抽象类使用就可以。

独有特点

- 有抽象字段（抽象属性）

`???`，是1个Nothing类型的变量名，意味着该方法待实现

```
abstract class A{
  //抽象方法，没有方式实现
  def eat(): Unit

  //抽象字段（抽象属性），没有赋值
  var age: Int
}
```

权限修饰符

java权限修饰符

1. 用在外部上:
   - `public`
     1. 所有地方都可以找到这个类
     2. 这个类的类名要文件名保持一致(`.java`)
   - 默认
2. 用在类的内部成员(属性, 方法, 内部类)上

   四种修饰符
   - `public`

     所有地方都可以访问
   - `protected`

     同包和子父类中可以访问

     在子类中访问`protected`的方法的时候:

     `super.foo();`
   - 默认(`friendly`)

     同包中访问

- `private`

  只能在当前类中访问

protected的理解（有误解，易错）

之前的理解有问题，易错

使用范围：同包和子父类中使用。

注意：子父类非同包的使用方法，特别是子类对父类的访问方式。

子类中，可以访问父类的protected的方法，但是要注意访问的方式。在子类中，直接创建父类对象，调用父类的protected方法是错误的，访问不到的。

```
class Child extend Father {

  main {}
  Father f = new Father;
  f.foo() //错误的
  super.foo();
}
```

scala权限修饰符

用在外部类上

scala没有`public`关键字

public

默认为public

`private`

只能在当前包使用

用在类的内部成员（属性、方法、内部类）上

public

默认为public

`protected`

这个限制更加严格，只能在子父类中访问，非子父类在同包中不能访问。

与java相同，子类对父类的protected方法访问，需要用`super.foo()`，也就是增加`super.`来访问。

private

可以控制访问，精细化控制

`private[允许控制的包名]`，注意包名，不能写全路径

单例对象

定义：单例对象是一种特殊的类，有且只有一个实例。

复习java

饿汉式，一开始就创建对象

懒汉式，类似于lazy，在使用时才创建对象

特点：涉及到多线程之后，代码更复杂，需要双重判断

scala单例对象写法

单例对象很简单，类似于独立对象的写法。

注意：object后面不能加上参数，因为会翻译成静态成员，静态成员是没有构造方法的。

写法：

```
object 对象名{
  //代码，属性，方法，具体可执行的代码
}

//不能加参数
object 对象名(val a =3 等参数){} //错误的，不能写参数
```

单例对象的类型

独立对象

当一个单例对象没有与类具有相同的名称，这个单例对象被称作是独立对象。简而言之，不是伴生对象的单例对象，就是独立对象。

例如，以下的SingDog，就是独立对象

```
object SingDog{}
```

实际上：**独立对象也有伴生类**，但是这个伴生类是**scala编译器提供的。**特点和下面的伴生对象一致，例如`伴生对象的成员，都会成为静态成员`

伴生对象

当一个单例对象与类具有相同的名称时，这个单例对象被称作是这个类的伴生对象。

例如，以下的TwoDog，就是伴生对象

```
object TwoDog{} //伴生对象
class TwoDog{} //伴生类
```

伴生对象和伴生类

称为伴生对象的条件

1.`class`的名字和`object`的名字相同

2.都在1个`.scala`文件中

特性：他们可以互相访问对方的私有成员

理解：伴生对象，相当于伴生类，单独getInstance()了1个对象。

java角度看反编译

scala字节码反编译为java，从java角度看scala代码

- 伴生对象的成员，都会成为**静态成员**
- 伴生类中的成员，都会成为**非静态成员**

理解：伴生对象接收了类中的静态成员——解决java的静态成员问题

为了编程语法更加符合面向对象的思想。面向对象，所有的成员，都应该依赖于1个对象。但是java的静态类却不依赖于1个对象。

java中的静态类，破坏了面向对象的思想。因为静态代码，不需要对象，就可以执行。

scala的伴生对象，解决这个问题。更加符合面向对象，是真正的纯面向对象。

理解：`伴生类的对象`和`伴生对象`

伴生对象，实际是`伴生类$`的类，而不是`伴生类`

伴生对象 extend 伴生类，才是`伴生类`，等同于`new 伴生类`

```
class Student(val name: String,val age: Int){

}

object Student extends Student("AA", 10) //此时是Student类

object Student {} //此时是 Student$ 类，而不是Student类
```

静态工厂

封装创建对象的细节，用工厂类去创建对象。

需要把构造器私有，屏蔽外界对类的直接创建，强制使用静态工厂。

`主构造器私有`**的问题**

```
class Human private(val color: String) {}
```

问题：工厂类也无法创建Human了。

解决方法：使用伴生对象和伴生类，他们的特点是互相可以访问对象的私有成员。

```
class Human private(val color: String) { //主构造器私有
  println(s"创造了$color 种人")
  override def toString: String = s"$color 种人"
}

object Human {
  val humans: mutable.Map[String, Human] = mutable.Map[String, Human](
    "黄" -> new Human("黄"),
    "白" -> new Human("白"),
    "黑" -> new Human("黑"))

  //若无预先创建的human，则创建新的Human并加入到map集合中
  def makeHuman(color: String) = humans.getOrElseUpdate(color, new Human(color))
}
```

apply——函数式编程和面向对象编程的融合

函数的特点：可以被调用

apply的作用：scala中，对象也可以像函数一样被调用

潜规则：对象被调用时，默认找apply方法

```
//函数调用:
函数名(参数)  //等价于 函数名.apply(参数)
// 对象调用
对象名(参数)   // 等价于 调用对象的 apply方法
```

注意：1.函数也可以用apply调用，但是方法不能apply调用。方法需要转成函数，才能使用apply调用。

2.**伴生对象的apply**很有用，通常做法是返回**伴生类的对象**。写法上，省略了1个`new`。（注意伴生对象和伴生类的对象的区别，以及他们apply方法的区别）

据说在后面有用不就省略了1个new吗？（疑问）（推测）让伴生对象和伴生类的对象产生关系。

3.普通类的apply，根据实际情况写。

实际情况是指，普通类中的哪些业务逻辑，用函数式编程方便，就可以把业务逻辑卸载apply方法中。

伴生类和伴生对象`apply`的区别

伴生对象，找伴生对象的apply方法

伴生类的对象，找伴生类的apply方法

特质trait

特质 (Traits)： 用于在类 (Class)之间共享程序接口 (Interface)和字段 (Fields)。类似于java8的接口。

与抽象类的区别（待补充）

特质的构造器比抽象类弱

特质只有1个默认的无参构造器；

抽象类可以有有参构造器，*且可以写很多辅助构造器*（待验证）

```
trait A {}
//不能写成有参构造器
trait A(val c =10) {} //错误的
abstract class A(val c =10){} //抽象类可以有有参构造器
```

成员

抽象类能有的成员, 那么特质都可有

1. 属性
2. 方法
3. 抽象属性（scala独有的，抽象类和特质都有）
4. 抽象方法
5. 构造器(主/辅)

> `java`接口:
>
> 1. 1.8之前: 是抽象方法和常量的集合
> 2. 从1.8开始, 默认方法

特点

特质实现了java最纯粹的接口

scala代码

```
//特质
trait Usb {
    println("Usb  的构造器")
    //抽象字段(属性)
    val name: String
    //常量
    val id = "1234567"
    //变量
    var cid = "abcdef"
    //已实现的方法（不是$init$，不要弄混！）
    def init() = {
        println(s"$name 开始初始化")
    }
    // 抽象方法
    def insert(): String
    def work(): Unit
}

//实现类
class HuaweiUSB extends Usb {
    //同名常量
    override val id = "444"
    //同名抽象字段
    override val name: String = "华为 USB"
    //实现类的代码块，特质的init方法（不是$init$，不要弄混！）
    init()
    override def insert(): String = {
        println(s"$name 开始插入设备")
        "ok"
    }

    override def work(): Unit = {
        println(s"$name 开始工作")
    }
}
```

java编译后

```
public abstract class Usb$class
{
  public static void init(Usb $this)
  {
    Predef...
  }
//特质的构造器（自造词）
  public static void $init$(Usb $this)
  {
    Predef..MODULE$.println("Usb  的构造器");
   //常量，id赋值
  $this.com$atguigu$scala1128$day05$traitdemo$Usb$_setter_$id_$eq("1234567");
```

```
    //变量，cid赋值
    $this.cid_$eq("abcdef");
  }
}

//实现类反编译
public class HuaweiUSB implements Usb
{
  private final String id;
  private final String name;
  private String cid;

    ...

  public void init() { Usb.class.init(this); }
//实现类的构造器
  public HuaweiUSB()
  {
  //第一行，就调用特质的构造器（自造词）
    Usb.class.$init$(this);
    //重写的常量id，重新赋值
    this.id = "444";
    //重写的抽象字段name，赋值
    this.name = "华为 USB";
    //实现类中的代码块
    init();
  }
}
```

特质中的代码块

- 代码块，会进入到类的构造器中。

实现类的写法

- 特质中的所有成员，如果在实现类中有同名成员出现，那么都需要加上`override`关键字，进行重写(覆写)

有非抽象成员的特质，编译后的`trait类$class`抽象类和`trait`接口

- 特质编译为java后，interface接口，成为了最纯粹的抽象类，因为这个接口中都是抽象类成员；
- abstract抽象类帮助interface接口，接收了不纯粹的抽象类成员，和普通类的区别就是没有构造方法。

trait和抽象类的区别

trait

- 可以多混入

抽象类

- 只能单继承

java角度看反编译

特质只有抽象的成员时

- 编译会生成1个interface接口。

特质有非抽象的成员时，如赋值的常量、变量、实现的方法

- 编译会生成2个字节码文件，1个`trait类`的interface接口。
- 另1个字节码文件，是`trait类$class`这个抽象类。
  - 其中`trait类$class`类，都有1个方法`$init$`，这个是`特质的构默认构造器`
  - 特质中的非抽象成员，都会放在`trait类$class`这个抽象类中。

> 对特质构造器的解释，见下文

理解特质构造器（了解即可）

特质是没有对象的，所说的特质构造器，指的是特质有非抽象成员时，反编译中，出现在`trait类$class`类中的，`$init$`方法。

产生的条件：特质中有非抽象成员

特点：反编译的文件中，有`$init$`方法。

方法体内容：除了实现的方法外的，非抽象成员，比如变量常量、代码块

谁创建的：scala编译器自动创建

使用的地方：实现类的构造器，第一行（除super()之外的））

叠加冲突

问题：2个特质，出现同名方法的问题

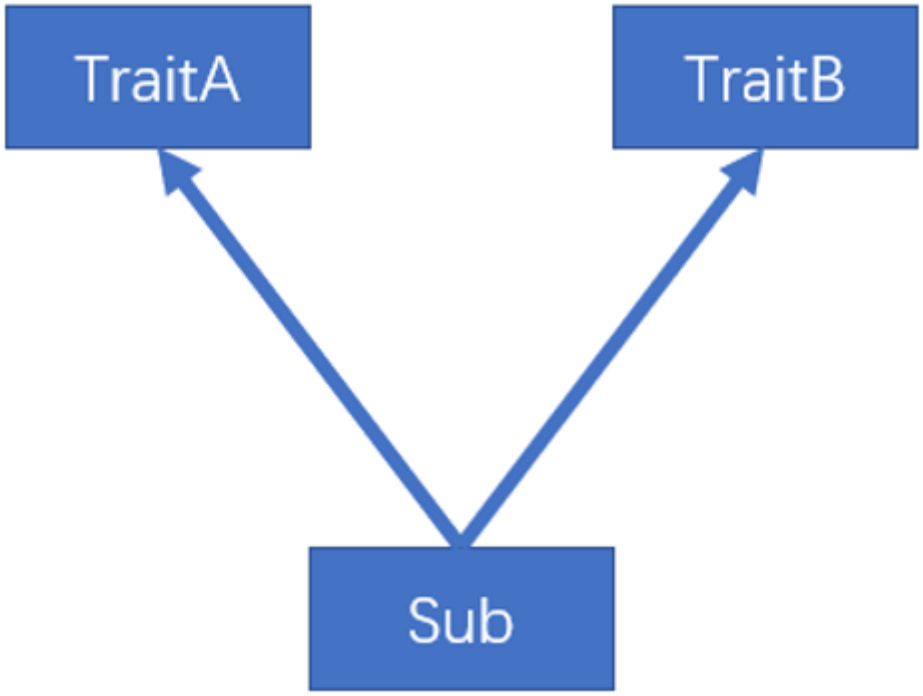
1.有同名抽象方法

可以在实现类中，实现1次即可。

2.有同名的已实现的方法

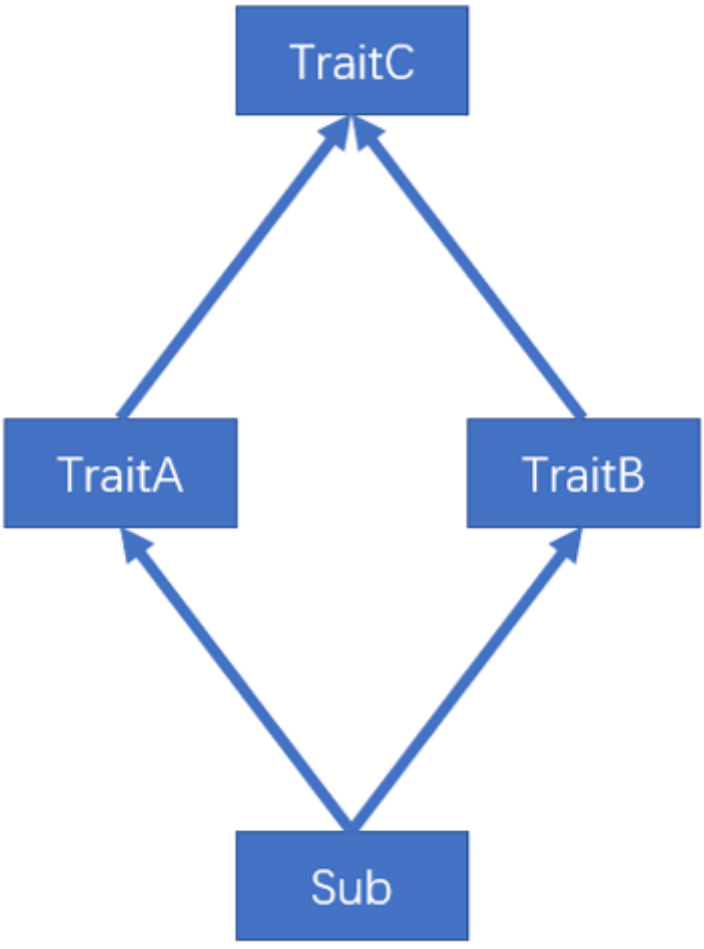
调用该方法会报错，有冲突。需要处理

处理方法1：在实现类中，该冲突的方法重写（覆写）



处理方法2：使用“钻石问题”解法。在这2个特质的共同父类中，写该方法，2个子特质类，变成重写共同父类的方法。

此时选择混入顺序中，**排最后的特质**的方法。



举例：

```
//特质T1和特质T2，都有实现的方法foo()，且T1和T2有共同父类T12。
//此时，foo()方法，选择排最后的特质，T1中的foo方法
//因为T2先初始化，T1后初始化，所以T1中的foo方法，覆盖了T2中的foo方法。
class C12 extends T2 withi T1 {}

//此时，foo()方法，依然是T1的foo方法
//因为java初始化会先初始化父类，所以T2初始化时，已经先初始化了T12
class C12 extends T2 withi T1 with T12 {}

//初始化叠加顺序，T12 -> T2 -> T1 -> C12
//若给foo()，添加super.foo()，那么此时，super指的是叠加顺序中的前一个类，而不是真正的父类。
```

理解super指的是谁

`super.foo()`，不是真正的父类，而是按照叠加顺序，找的前一个加载的类。

问题需求：如果要避免因叠加顺序问题，需要找到了兄弟类，而不是找父类。

解决方法：`super[T12].foo()`，明确是定super指的是T12类，而不是按照初始化的叠加顺序的兄弟类T2。

> 开发环境中，需要避免这种叠加冲突。

特质继承类

特质可以继承类

目的：特质为了使用继承类中的代码

1.特质直接继承需要的类

```
class A{}
trait B extends A{} //目的，为了调用A中的一些代码
class C extends B{}

class A{}
trait B{}
class C extends A with B{}

//出错，类C隐含着继承了2个类M和A。因为java只能单继承。
//若要编译通过，则M类应该是A类的子类
class M{}
class A{}
trait B{}
class C extends M with B{} //编译错误
```

2.使用自身类型（selftype）

```
class A{ def foo() = println("A...foo")}
trait B{
  //s就是A的对象，s是1个变量名
  s: A =>
  //也可以写成
  _: A =>

  def eat() {
    this.foo()
  }
}
class C extends A with B{}

//等价于，特质B直接继承A
class A{}
trait B extends A{}
class C extends B{}
```

动态叠加

在`java`中所有的继承关系都应该在定义类是确定好.

`scala`支持特质的动态叠加. 在创建对象的时候, 可以临时只针对整个对象来叠加特质

作用：相当于省略了实现类。特质中有待实现的成员，还需要实现。

举例：

```
main{
  val h =new H with F1 //特质叠加，可以叠加多个
  h.foo() //叠加后，可以使用叠加特质的已实现方法。
}

class H
trait F1 {
  def foo() = println("f1 foo...")
}
```

补充知识

类型的判断和强转

有替代：可以用模式匹配来实现类型判断和强转

注意：强转只能用于引用类型，不能用于值类型

有不能强转的情况，比如值类型，Int。但是课上演示时发现，并没有报错。可以正常强转（疑问）

原因，Any中有了这2个方法

```
isInstanceOf[]
```

```
asInstanceOf[]

object Extra1 {
    def main(args: Array[String]): Unit = {
        val a:A = new B
        // java中判断类型:   a instanceof B
        if (a.isInstanceOf[B]) {  // 判断a是否为B的对象
            val b = a.asInstanceOf[B] // a转换B的类型
            b.foo()
        }
    }
}

class A
class B extends A{
    def foo() = println("foo...")
}

//强转只能用于引用类型，不能用于值类型
```

总结

使用时，把trait当作1个接口使用即可

枚举类

可以借鉴的操作：用java来创建枚举类，然后在scala中使用。

原因：java的枚举类好用。

模拟枚举类

sealed，密封类。它的子类将来只能出现在当前的这个文件中。

不写main方法，也可以直接执行代码（一般不使用，只是省了main方法而已）

补充知识

App类

继承App，App中有main方法，可以直接执行代码。

只做了解，**spark中永远不要使用**，会出问题。spark官方文档中，有说明不要使用App。

类的别名

使用的地方比较多。

实际打印类名的时候，还是会打印出原来的类名。

```
type T =

//比如 Predef.scala中
//scala将java中的String，起别名为String。所以我们可以在scala中，用String来调用java的String
type String = java.lang.String
```

内部类

内部类，怎样取外部类的属性值?

java的方法：`外部类.this.属性名`

scala的方法：使用自身类型`that.属性名`，that默认就是自身类型`that =>`

类型投影

内部类的方法，如果参数是内部类

java和scala，都默认该内部类，必须是同一个外部类对象中的内部内

但是scala可以跳过必须是同一个外部类对象的限制

使用类型投影

```
def foo(obj: Outer#Inner)
```

隐式转换

前言：scala源码不好理解的地方，如果不掌握这个，看spark源码会非常痛苦。必须要学好。

一共包含3个部分

1. 使用隐式转换函数
2. 隐式类(对隐式转换函数的简化), 在scala`2.1.x`版本才有的功能
3. 隐式参数和隐式值

隐式转换函数

写法：函数添加 `implicit` ，参数类型和返回值类型要正确。

注意：隐式转换函数**在作用域内**，只能有1个，如果有多个相同类型的隐式函数，那么语法编译会报错。

```
// 在作用域内只能有一个，如果有多个 Double=> Int 那么这个时候，隐式转换就不会发生
// 不关心函数名，只看函数的参数类型是否是Double，且返回值类型是Int，那么就会调用这个函数。
implicit def double2Int(d: Double): Int = d.toInt

val a: Int = 10.1 // Double=> Int
```

应用举例：

java的File类的缺点，不能直接读取文件信息，需要使用数据流读取。

scala.io.Source，替换了java中读取文件的那一套代码

```
object ImplicitDemo2 {
    def main(args: Array[String]): Unit = {
        // 读取文本文件中的内容  java: IO
        implicit def file2RichFile(file: File) = new RichFile(file)

        val content = new File("C:\\....路径").readContent
        println(content)
    }
}

class RichFile(file: File) {

    // 这个方法要真正的去封装读 文件 内容的代码
    def readContent: String = {
      //mkString，将文字拼接起来
        Source.fromFile(file, "utf-8").mkString
    }
}
```

重要功能

对已有的类，做功能上的扩展。（替代了，java中的增强类了。）

应用举例：

实现解析语句"3 days ago"，得到1个3天前的日期。

```
object Implicit03 {
  def main(args: Array[String]): Unit = {
    //隐式转换，将Int转换为增强类RichInt
```

```
    implicit def Int2RichI (i: Int) = new RichInt(i)
    val ago = "ago"
    val later = "later"
    val i1 = 3
    //使用函数时，可以省略点，函数只有1个参数时，可以省略括号
    println(3 days ago)
    println(4.days(later))
  }
}

//需要转入的信息
class RichInt(day: Int) {
  //增强类的函数
  def days(when: String)= {
    if ("ago" == when) LocalDate.now().minusDays(day).toString
    else {LocalDate.now().plusDays(day).toString}
  }
}
3 days ago
//可以分解为
3.days(ago)
//Int增强类，增加1个新的方法days，传入参数ago或者later
```

有用的日期类

`LocalDate`

取今天的日期，前后N的日期，很方便。

```
LocalDate.now() //现在的时间
LocalDate.minusDays(day: Int) //往前推day天
LocalDate.plusDays(day: Int) //往后推day天
```

不用在Date，DateFormat来自己写代码了。

隐式转换函数在什么时候起作用?

1.赋值给没有任何关系的其他类时

```
//A和B类型，没有任何关系
val a: A = new A
val b:B = a   // 会找隐式转换（A => B），如果有所需要的隐式转换，则编译成功
```

2.对象调用了自己没有的方法时，会找隐式转换，如果找到新的对象有该方法，则成功。

```
implicit def a(file:File) = new RichFile(file)
val content = new File("..").readContent //File类没有readContent方法，找隐式转换（File => RichFile）
//RichFile中有一个readContent方法
```

只有在需要的时候才会隐式转换，以下问题可参考理解

- 当隐式转换的转换类型，和原类型都有同名的方法时，会用哪个方法?
    - 会优先使用原类型的。
    - 因为对象可以调用自己的方法，不会找隐式转换。

怎样写隐式转换?

- 写一个隐式转换函数
    - 函数名没有限制, 随意
    - 参数一定是你原来的类型
    - 返回值一定是你新定义的类型
    - `implicit def a(file:File) = new RichFile(file)`
- 写一个自定义类型(能够完成以前的类没有的功能)
    - 自定义一个类: 主构造接受已有的类型
    - 在定义一个你需要的方法

```
class RichFile(file: File){
    def readContent = {}
}
```

隐式类

用`implicit`修饰的类就是隐式类

优点：可以省略隐式转换函数，是隐式转换函数的升级版

使用限制：不能写在顶级类中，只能**写在内部类中**任意位置，但是要注意，在main方法中，隐式类要写在使用隐式转换代码的前面。

有使用限制的原因：`implicit`不能用在顶级类

```
implicit class RichFile(file: File) {
    // 这个方法要真正的去封装读 文件 内容的代码
    def readContent: String = {
        Source.fromFile(file, "utf-8").mkString
    }
}

object implicitDemo {
  def main(args: Array[String]): Unit = {
    //不再需要下面的隐式转换函数，就可以实现其隐式转换的功能了
  implicit def file2RichFile(file: File) = new RichFile(file)
  }
}
```

隐式参数和隐式值

隐式参数和隐式值，通常配合使用。

```
//隐式值，注意作用域
implicit val aaa: Int = 100

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

// a是隐式参数，将来调用的时候根据需要,可以不传
// 让scala自动帮助我们传递，找一个隐式值
def foo(implicit a: Int) = {
    println(a)
}
```

隐式值需要注意的

参数名没有影响，只看类型。

同个作用域中，只有1个类型的隐式值。

当函数没有传入参数时，则会自动传入隐式参数

函数使用隐式值时，必须省略圆括号，否则编译会报错。

隐式参数需要注意的

参数列表中，有implicit，则参数都是隐式的。

如果函数中，有非隐式的参数，则应使用函数**柯里化**。

- 隐式参数和非隐式出纳室放在不同的参数列表中。
    - 使用两个参数列表
    - 第一个是必须要传的参数
    - 第二个里面是隐式参数

隐式参数和默认值的区别

隐式参数和默认值，不是一回事

情形：当隐式参数有默认值

- 省略圆括号，表示在使用隐式值。
- 加上圆括号，表示在使用默认值。

```
def main(args: Array[String]): Unit = {
    implicit val a1: Int = 100
    implicit val s1: String = "well done"

    foo //使用了a1、s1，2个隐式值
```

```
  //foo 会报错，使用隐式值，不加圆括号
    foo1 //省略圆括号，使用了a1，1个隐式值
    foo1() //加上圆括号，使用了默认值，200
    foo2() //使用了默认值，300
    //foo2 使用默认值时，不能省略圆括号

  }
  //隐式参数
  def foo(implicit a: Int, b: String) = println(a)
  //隐式参数带默认值
  def foo1(implicit a: Int = 200) = println(a)
  //默认值
  def foo2(a: Int = 300) = println(a)
```

实际上：不会有人，在使用隐式参数时加上默认值。

隐式转换函数，隐式类，隐式值 的查找路径位置

查找顺序

1.在当前作用域中查找

2.在相关类的伴生对象中查找

- 理解，什么是相关类
  - 在发生隐式转换的那一行中，所出现的类，都是相关类。
  - 如果有泛型，泛型也是相关的。

```
val a: A =new A //只有1个相关类，A
val b: B =new A //有2个相关类，A、B
```

拓展-隐式转换使用广泛，String为例

String，也有隐式转换

有隐式转换，是scala的String，比java强的原因

String的隐式转换位置：在默认导入类，`scala.Predef`中

```
val s: String ="abc"
s.head //可以使用String中没有的函数，说明String有隐式转换
```

集合（最重要的，没有之一）

集合框架

复习java的集合框架

- 数组
- List，Set，Map

缺点：这些集合之间，没有太多关联。同种作用的方法，其方法名都不一样。

scala集合特点（待完善）

1.所有的集合框架，有一些共同的方法。

2.提供了很多高阶算子（函数），可以操作集合中的元素。

3.所有的集合，都是从3个类继承得到的

分为2大类：可变集合、不可变集合

- 不可变集合，优点：没有多线程问题。在分布式中，使用不可变集合，可以减少很多Bug。
- 集合可变还是不可变，要看这个集合继承的是可变父类，还是不可变父类。与集合类型（数组、列表、集合）无关。
- scala默认是不可变的，例如`Seq`是不可变的
- 需要可变集合时，需要加上`mutable`，例如`mutable.Seq`是不可变的

Iterable，表示子类都可以用for循环

Seq，有顺序的

IndexedSeq，有下标的

LinearSeq，无下标的，如链表

Set

SortedSet，类似于java的TreeSet

Map

SortedMap

是否可变，要看他们的后代。

不可变家族

Seq使用最多的，List

Set使用最多的，HashSet。其中ListSet可用List替代。

Map使用最多的，HashMap

可变家族

WeakHashMap，可用来做缓存。不用的缓存会回收。

通用特点

在scala中，几乎所有的集合之间可以用互相转换。包括可变转不可变

通用函数

常规函数

```
val list1 = List(30, 50, 70, 60, 10, 20)
//         （1）获取集合长度
//length和size，是同一个函数
println(list1.length)
//         （2）获取集合大小
println(list1.size)
//         （3）循环遍历
list1.foreach(println)
//         （4）迭代器
println("----------")
val it: Iterator[Int] = list1.iterator
println(it.hasNext)
println(it.isEmpty)
while (it.hasNext) {
    println(it.next())  // 返回刚刚跳过的那个元素
}
//         （5）生成字符串，mkString(分隔符)
println(list1.mkString("-> "))
//         （6）是否包含，isContains()
println(list1.contains(30))
```

理解迭代器

和java中的一样

理解：可将迭代器理解为指针，指针指向元素地址的开头位置。

优点：非容器式集合。节省内存，指针只会指向数据，并不会复制数据。

迭代器的方法

1.`hasNext`，判断有没有下一个元素，有元素则返回true，不会移动指针

2.`isEmpty`=`!hasNext`，判断有没有下一个元素，没有元素则返回true，不会移动指针

3.`next`，返回刚刚跳过去的元素

注意：1个迭代器，只能迭代1次。迭代器不能往回走。

取值函数

```
//  （1）获取集合的头head
```

```
// println(list1.head)
// （2）获取集合的尾（不是头就是尾）tail
// println(list1.tail)
// （3）集合最后一个数据 last
// println(list1.last)
// （4）集合初始数据（不包含最后一个）
// println(list1.init)
// （5）反转
// println(list1.reverse)
// （6）取前（后）n个元素
// println(list1.take(3))
// println(list1.take(30))
// println(list1.takeRight(2))
// （7）去掉前（后）n个元素
// println(list1.drop(2))
// println(list1.dropRight(3))
// （8）并集
// println(list1.union(list2))
// （9）交集
// println(list1.intersect(list2))
// （10）差集
// println(list1.diff(list2))
// （11）拉链
// val list3: List[(Int, Int)] = list1.zip(list2)  // 多出会忽略
//  val list3 = list1.zipAll(list2, -1, "fff")  // 配置默认值
//        val list3= list1.zipWithIndex  // 元素和索引进行拉链
//        println(list3)
// （12）滑窗
val list1 = List(30, 50, 70, 60, "abc")
        val list2 = List("30", "50", "7", "6", "10", "2", "abc")
        val it: Iterator[List[String]] = list2.sliding(3, 1)
        println(it.toList)
```

交并差集

- 可重复集合
    - 可以使用交并差集，但用的不多。
    - 运算符，使用有限制，可重复集合

拉链

将2个集合的元素，按顺序一一结合在一起，用元组保存。

```
list1.zip(list2) //多余的元素会被忽略

list1.zipAll(list2, -1, -2)
//若list1多余了，则会给list1中多出来的的元素配 -2
//若list2多余了，则会给list2中多出来的元素配 -1

//和自己的索引进行拉链
list1.zipWithIndex
//如果集合没有索引，则用顺序进行拉链。但是顺序可能没有意义。
```

滑窗

滑窗结束的标志：窗口接触到了最后1个元素

```
//窗口大小a，滑窗间隔b
.sliding(a, b)

val list = List(30, 40, 50, 60, 70)
list.sliding(2, 2) //返回迭代器
//则迭代模拟
第1次，(30, 40)
第2次，(50, 60)
第3次，(70) //接触到最后1个元素，滑窗停止
```

值计算函数

```
val list1 = List(30, 50, 70, 60, 10, 20)
//         （1）求和
println(list1.sum)
//         （2）求乘积
println(list1.product)
//         （3）最大值
println(list1.max)
//         （4）最小值
println(list1.min)
```

排序是比较难的

集合高级操作

foreach

省略循环，达到遍历的目的

返回值：没有

需要传递1个函数

map

一进一出，将集合中的每个元素经过算子后，放入到新的集合中。

返回值：有

应用：对数据结构进行调整。

举例：List集合存放数字，需求返回1个元组的List集合，存放(数字本身, 数字的位数)

Map集合，需求返回map中的key值的集合。

```
待补充
_._1 //代表key
```

filter

一进1出，或0出。满足条件的输出，不满足的去掉

返回值：有

满足条件的输出，不满足的取消

```
x => x > 20
// _ 替代了 x => x
_ > 20
```

flatten

拍平，扁平化。类似于Hive中的行转列，降维。

1次只能拍平1个维度

元组不能拍平

List[Array[Int]]，拍平成为List[Int]

List[String]，拍平成为List[char]

举例：将英文语句List，变成英文单词List

思路map->flatten

```

```

map+flatten配合使用的语法

flatMap

一进多出（0，1，>1）

要求：传入的函数必须能返回1个集合

一般做法：使用flatMap，而不是用map->flatten

groupBy

多进少出

注意：匿名函数，如果返回值没有改变，`x => x`，则不能使用`_`代替

mapValues

只有map，元组可以使用

key原封不动，只对value做map

与map比较：省略了key

reduce

(结果值, 本次运算值)

聚合，化简

多进1出，只有1个返回值

返回值类型，和原元素类型一致

原理：第一次计算，用第一个元素和第二个元素。后面的计算，用上一次计算的结果值和下一个元素

(待插入图片)

reduceRight

(本次运算值, 结果值)

与reduce作用相同，但是计算顺序为从右到左

folderLeft-`/:`

(结果值, 本次运算值)

与reduce的区别：第一次计算，`zero`零值和第一个元素。`zero`零值的类型会决定最终聚合的结果值类型

返回值类型，由`zero`零值的类型决定，和原元素类型可以不一致，

folderRight-`:\`

(本次运算值, 结果值)

与folderLeft作用相同，但是计算顺序为从右到左

reduce和folder的区别

reduce的结果值类型和元素类型必须一致。

folder的结果值类型可以和元素类型不同。

reduce能做的事情，folder肯定能做到。但是folder能做到的事情，reduce不一定能做到。

举例：将List[Int]，聚合为1个字符串

```
val list: List[Int] = List(30, 50, 70, 60, 10)
//需求 转为 "305070..."
list.foldleft("")(_+_) //原元素类型为Int，结果值为String，零值""为String
```

folder用运算符的好处

可以更加直观地看到零值

```
list1.foldLeft(start) //函数的原始使用形式
list1./:(start) //运算符形式
start /: list1 //运算符形式简化
```

scan

folder的增强版，可以看到变化的过程

返回值：1个集合，比原集合长度+1，多了零值

```
val list1 = List(30, 50, 70, 60, 10, 20)

val list2: List[Int] = list1.scanLeft(0)(_ + _)
//List(0, 30, 80, 150, 210, 220, 240)
//记录每一次折叠的过程，比原集合多了1个元素，多了零值

val list2 = list1.scanRight(0)(_ + _)
//List(240, 210, 160, 90, 30, 20, 0)
//与scanLeft记录顺序相反
```

排序sort

前言：本节排序内容，不涉及任何的具体排序算法（冒泡，选择，插入，希尔，快速，归并）

重点掌握Ordering和sortBy

3个排序函数

注意：不管排序的是可变还是不可变结合，都不会修改原集合，返回的是1个新的排序后的集合

复习java排序

`java`中的排序, 涉及到元素的比较大小

1. 元素自己可以和其他的兄弟比较 1 < 2

   让类实现`Comparable接口`

2. 找一个比较器进行比较两个元素大小`c.compare(a. b)`优先级高

   `Comparator`

   - `compare(a, b)`

     如果返回值小于零 a < b

     返回值等于 a == b

返回值大于零 a > b

让类实现排序功能

java写法

实现Comparable

scala写法

1.类实现比较（了解即可）

类实现Ordered

继承自Comparable

优点：可以使用运算符，来比较类的大小

缺点：不够灵活，不能灵活改变升序或者降序。需要去更改自定义类型的定义。

```
class User1(val age: Int, val name: String) extends Ordered[User1] {

    override def toString: String = s"[age = $age, name = $name]"

  //重写compare方法，实现排序
    override def compare(that: User1): Int = this.age - that.age
}
```

2.使用比较器（必须会写）

使用比较器Ordering

继承自Comparator

优点：1.已经内置了常用类型（Int，Boolean等）的比较器，没有内置的类，需要自己定义比较器。.

2.降序排序只需要用`.reverse`即可。

sorted有隐式值，不传参数，会自动传比较器。

```
def sorted[B >: A](implicit ord: Ordering[B]): Repr =
```

举例

```
def main(args: Array[String]): Unit = {
  //定义User2的比较器
   implicit val ord: Ordering[User2] = new Ordering[User2] {
       override def compare(x: User2, y: User2): Int =
           if (x.age > y.age) 1
           else if (x.age == y.age) 0
           else -1
   }

  //创建List，进行元素的比较
   val list1 = List(new User2(20, "a"), new User2(10, "c"), new User2(15, "b"))
```

```
    println(list1.sorted)
    }
}

//User2类
class User2(val age: Int, val name: String) {
    override def toString: String = s"[age = $age, name = $name]"
}
```

sortBy（重要）

可以解决99.99%的排序问题

理解：sortBy(排序指标)(排序比较器)

多个排序指标，用元组。最多可支持9个排序指标。若排序指标大于9个，则需要自己写Ordering.Tuple

升序转降序，直接在排序指标前加`-`

举例：

```

```

sortWith

day07的最后1个案例，是最难的（复习）

如何选择高级算子?

1.看这些集合的输入和输出个数，一进一出或者一进多出。

经典案例-wordcount

java式做法

```
待补充
```

scala式做法

```
val list = List("hello world", "hello world", "hello beautiful hello")
//flatMap -> groupBy -> map
val list1 = list.flatMap(x => x.split(" "))
      .groupBy(x => x)
      .map(kv => {
        (kv._1, kv._2.size)
      })
 println(list1)
```

数组

定长数组-`Array`

`Array`

本质：java数组，长度不能变，元素可以换

创建数组

1.直接给元素值

```
val arr: Array[Int] = Array[Int](10, 20, 30)
```

2.使用`new`创建

```
//定义数组长度，此时不能省略数组的泛型，否则会推导为Nothing
//例如，下面的数组，不能忽略泛型定义[Int]
new Array[Int](length)
//等价于
Array.ofDim(length) //这个一般用于多维数组
Array.ofDim(1D-length)(2D-length) //二维数组
```

添加元素-`:+` `+:` `++` `:+=`

可以用运算符（实际是函数），在ArrayOps中，定义了这些运算符

1. `:+` 用来在数组的尾部添加元素，返回1个新数组
2. `+:` 用来在数组的头部添加元素，返回1个新数组
3. `++`合并2个数组，合并后的元素顺序，左数组到右数组，返回1个新数组

```
val arr = Array(30, 50, 70)
val arr2 = Array(130, 150, 170)
//尾部添加
arr :+ 100 //此时产生了新数组
//头部添加
100 +: arr //这里有运算符的结合性
//2个数组合并
arr ++ arr2 //产生新数组，元素顺序为(arr, arr2)，也就是(30, 50, 70, 130, 150, 170)
arr2 ++ arr // (130, 150, 170, 30, 50, 70)
```

4.`:+=`数组尾部增加1个元素，不返回数组

5.`+:=`数组头部增加1个元素，不返回数组

让定长数组，也可以增加元素。等价于`:+`之后，再赋值。

本质：生成了1个新的数组，将结果复制到了新数组中

```
var arr1 = Array(30, 50, 70)

arr1 :+= 100 //等价于 arr1 = arr1 :+ 100，arr1原数组销毁，生成新的数组，地址值改变
//此时，如果arr1是val，那么就会报错。
//本质是生成新数组，将新数组的地址值赋值给了变量arr1。
```

运算符的结合性（可考虑将本节内容移至运算符）

通常有左结合和右结合

规则：如果1个运算符，以`:`结尾，那么就是右结合。

举例：`100 +: arr`，等同于`arr.+:(100)`

作用：写起来，非常的直观。新的元素顺序，一目了然。`100 +: arr`，元素顺序就是`100, arr`中的元素

```
a + b 左结合
a = 30 右结合
```

可变数组-`ArrayBuffer`

创建数组

```
// 创建可变数组
val buffer = ArrayBuffer(10, 20, 30, 40, 10.3)
// 创建一个空的ArrayBuffer
new ArrayBuffer[Int]()
// 创建一个空的ArrayBuffer
ArrayBuffer[Int]()
```

添加元素-`+:` `:+` `+=` `+=:` `:+=` `++=`

1.**添加元素，原数组不变化**，返回1个新数组

`+:`，头部增加元素，返回1个新数组。

`:+`，尾部增加元素，返回1个新数组。

```
var arr = ArrayBuffer(10, 20)
100 +: arr //100, 10, 20
arr :+ 100 //10, 20, 100
```

2.**添加元素，原数组变化**

`+=`，在可变数组的尾部增加元素(没有产生新的集合)，等价于 append() <-这是java的写法

`+=:` 在可变数组的头部增加元素(没有删除原来的数组)

`:+=` 在可变数组的尾部增加元素(删除原来的数组)

- 相当于`:+`得到新数组后再赋值给原来的变量，此时数组地址值改变，原数组丢失。
- 与`+=:`做出区别，有没有删除原数组

```
var buffer = ArrayBuffer(10, 20)
buffer += 100  // 等价于 buffer.append(100)

100 +=: buffer // 100, 10, 20 buffer地址值未变化
buffer :+= 100 // 10, 20, 100 buffer地址值有变化
// 等价于 buffer = buffer :+ 100，buffer :+ 100返回1个新数组
```

3.**将1个数组追加到另1个数组**

`++=` 或`++=:`1个数组追加到另1个数组，只有1个数组发生追加变化。

```
//以下2个数组追加的结果，是一样的
arr1 ++= arr2
//arr1变化，元素顺序是arr1, arr2
arr1 ++=: arr2
//arr2变化，元素顺序是arr1, arr2
//一定要注意，元素顺序和哪个数组变化，一定要区分开
```

多维数组

和java完全一样，底层就是`java`的多维数组

本质都是假的, 都是用一维数组模拟出来!

List

不可变List-`List`

创建List

```
// 1. 得到List集合
val list1 = List(10, 20, 30)
//val list2: List[Int] = list1 :+ 100
val list2: List[Int] = 100 +: list1
```

2.创建空List（待补充）

```
val list3 =List[Int]() //注意，圆括号不能删除，否则会变成函数

Nil //空集合
```

增加元素-`::` `:::` `++`

`:+` `+:`，这个在List中用得不多，不做记录。

```
list :+ 100
100 +: list
//这个用得不多
```

List专有运算符

1. `::`，中置运算符（用得最多的），返回1个新List
2. `:::`，把2个List的元素合并到一起，返回1个新List

执行顺序：`:`结尾，是右结合

```
//以 : 结尾，所以是右结合
list1 :: list2 //将list1，作为整体，存入list2
//::: 和 ++，是等价的
val list3 = list1 ++ list2
//等价于
val list3: List[Int] = list1 ::: list2  // list2.:::(lit1)
```

创建空集合

```
val list3 =List[Int]() //注意，圆括号不能删除，否则会变成函数

Nil //空集合
```

空集合传入元素

```
10 :: 20 :: 30 :: Nil
//等价于
List(10, 20, 30)
```

默认对象实现的含义

```
val seq = Seq(10, 20, 30) //等价于 List(10, 20, 30)
```

可变ListBuffer-`ListBuffer`

```
不能使用 ::  :::
```

元组-`Tupple`

元组的作用

函数的返回值，不能是2个不同类型的值。多个同类型的值，可以用数组存放。

作用：专门用来封装**数据类型不同**的数据，放在一起。

替换了java中为了接收不同类型返回值而创建的java bean。

举例1

(a, b) 对偶，(k, v)kv形式的

元组存放函数的多个类型返回值，在spark中，用得非常非常多。

取元素，可用模式匹配

```
val m = 10
val n = 3
// 同时计算出来 m / n 和m % n取余
//元组接收2个返回值
val r: (Int,Int) = /%(m, n)

val (a, b):(Int,Int) = /%(10, 3)
println(a) //使用了模式匹配
println(b) //使用了模式匹配
```

```
//如果不想要某个返回值，可以用 _
val (a, _) = /%(10, 3)

//有2个返回值的函数
def /%(m: Int, n: Int): (Int, Int) = {
        (m / n, m % n)
    }
```

举例2

元组的嵌套，用得非常非常多

```
val t = (1, (10, (20, (30, 40))))
println(t._2._2._2._2)  //取出40
```

元组的特点

1.`Tuple1 - Tuple22`，最多可以封装22个数据

2.元组不可变，不能修改，不能增加删除。

3.元素类型是确定的，也不能修改。

4.元组永远没有遍历的需求。

元组的操作

创建元组

有4种写法

`()` `->` `→`

```
val t1 = Tuple2("a", 10)
//等价于
val t1 = ("a", 10)
//等价于
val t1 = "a" -> 10
//等价于
val t1 = "a" → 10
```

取出元素-`._n`

1.`._n`，取第n个元素

2.模式匹配，可参考<元组的作用-举例1>

```
// /%是举例1中，我自定义的1个函数
// 同时计算出来 m / n 和m % n取余
val (a, b):(Int,Int) = /%(10, 3)
//用 ._n ，来取得第n个元素
t1._1 //取得"a"
.
val (a, b):(Int,Int) = /%(10, 3)
println(a) //使用了模式匹配
println(b) //使用了模式匹配
```

Set

Set的特点

- 集合内元素无序。
- 元素不能重复。

回顾：java的`TreeSet`，有序，是因为自动排序

理解有序和无序

- 有序
  - 取出顺序和放入顺序一致 叫做有序
- 无序
  - 遍历顺序和存入顺序不一致.

Set的操作

创建Set

Set默认为不可变，若要可变Set，则要加上`mutable`

```
val set1 = Set(10,20,30,50)
val set2 = Set(1,20,30,1) //结果为1,20,30
```

元素操作-`+` `-` `+=` `-=`

1.添加元素-`+`

2.删除元素-`-`

3.添加元素并赋值-`+=`

4.删除元素并赋值-`-=`

```
val set = Set(10, 50, 30, 10, 40, 10)
//Set(10, 50, 30, 40)
val set1 = set + 1
//Set(10, 1, 50, 40, 30)，Set无序，所以"1"插入的位置不确定
val set2 = set - 10
//Set(50, 30, 40)

import scala.collection.mutable
//可变Set
val set1: mutable.Set[Int] = mutable.Set(10, 20, 30, 20)
set1 += 100
set1 -= 20
println(set1) //Set(30, 100, 10)
```

关系操作-`++` `|` `&` `&~` `--`

1.并集-`++` `|`

对应函数：`union()`

2.交集-`&`

对应函数：intersect()

3.差集-`&~` `--`

对应函数：diff()

```
val set1 = Set(10,20,30,50)
val set2 = Set(1,20,30,5)
// 并
println(set1 ++ set2)  // ++ 所有集合通用
println(set1 | set2)
println(set1.union(set2))
// 交集
println(set1.intersect(set2))  // java
println(set1 & set2)
// 差集
println(set1 &~ set2)
println(set1 -- set2)
println(set1.diff(set2))
```

经典使用-去重

思路：得到Set，再转换成需要的集合

做法：其他集合用`.toSet`，即可得到Set。

得到去重的结果，以List为例

```
list.toSet.toList //即可得到去重后的List
```

Map

Map的特点

1.在`scala`中, 是使用元组来表示键值。

2.遍历Map，得到的就是元组

3..key不能重复，新的value值会替换旧的value值。

4.获取元素最好使用`map.getOrElse(key, defaultVlaue)`

5.可变`Map`多了一个`map.getOrElseUpdate(key, defaultVlaue)`

Map操作

创建map

```
//Map默认为不可变
var map1 = Map(("a",10), ("b",20), ("c",30))

//可变map，需要加上mutable
import scala.collection.mutable
var map2 =mutable.Map("a" → 10, "b" → 20, "c" → 30)
```

for遍历

```
val map = Map(("a", 97), ("b", 98), ("c", 98))
//使用通常的
for (kv <- map) {  // map中存储的是元组，所以，遍历出来的kv就是元组
  println(kv._1)
  println(kv._2)
        }

//使用模式匹配
for((k,v) <- map) {
  println(k+" "+v)
}

//遍历过滤
//只要value值为98的元组
for((k,98) <- map) {
  println(k+" "+v)
}

//只要key值
for((k, _) <- map){
  println(k)
}
//等同于
for (k <- map.keySet) {
```

```
  println(k)
}
```

取出所有key

```
val set = map.keySet
```

获取元素

```
map(key)  //若key不存在，汇报异常

map.getOrElse(key, 默认值defaultVlaue) //若key不存在，会返回默认值，不会报异常

//只能用在可变map，不可变map没有该方法
map.getOrUpdate(key, 默认值defaultVlaue) //若key不存在，会返回默认值，并且把这个(key-默认值)键值对存入map中，不会报异常
```

队列

FIFO，先进先出

入队

出队

栈

FILO，先进后出

入栈

出栈

并行

.par

打印时，会

spark中不使用。

用了分布式，就不用并行了。

惰性数据结构

使用的使用，才生成

作用：节省内存

stream

使用`.toStream`获取集合的流

特点：1次求出1个值

强制求值，`.force`

`#::`，在stream的头部添加

Option

表示1个对象，要么有，要么没有。

源码中使用得多

作用：避免空指针。提示使用者，这里可能会有空指针。

解决java中，null空指针的问题。

常用函数：

判断是否有对象（不是null）

得到对象

其他的函数，类似于集合

举例

```
def foo(): User = {
  null 或者 User对象
}
//此时，foo函数，可能会返回1个对象，或者null。需要在使用这个方法时，进行健壮性的判断。
```

模式匹配

是对`java`中的`switch`的升级，功能远远大于`switch`

特点

1.不用写break，避免了java的case穿透问题

2.模式匹配时，若没有匹配到，则会抛出异常

3.匹配所有

可用`_`和变量名，如`aa`来实现匹配所有。

本质上，这2种方式是一样的。

区别在于，是否能使用匹配到的值。

注意：变量名，首字母应该是小写字母，否则会被认为是常量。

```
// _ 匹配所有
//匹配到的值用不了
case _ => 代码

//aa是变量名，可以匹配所有
//可以使用匹配到任何的值
case aa => println(aa)
```

4.匹配的值是常量还是变量

判断标准：看首字母是大写字母还是小写字母

常量，首字母大写

变量，首字母小写

```
val b = 120
val B:Int= 130
val a = 110 //匹配case b，打印 110bbbb
a match {
    case B =>
        //B是常量，是前面B =130这个常量。
        println(B + "BBBB")
    case b =>
        // b是一个新定义的只能在当前的case中使用的变量
        println(b + "bbbb")
    case _ =>
}
```

5.匹配有顺序，从上至下，匹配到后就不会再往下匹配

```
operation match {
  //因为 _ 下划线匹配所有，所以后面的case就不会匹配到了
  case _ =>
    println("你的运算符有问题")

  //后面的，都匹配不到
  case + =>
    println(a + b)
  case ...
}
```

6.模式匹配的值

scala中，任何语法都有值。

模式匹配的返回值，是匹配成功后的执行代码的结果值。

返回值类型，是所有匹配的执行代码的结果值类型的共同父类。

```
//s是该模式匹配的返回值，由于返回值可能有Int和String和异常，所以类型推导为Any。
//s的值为200，因为4匹配到case 4，执行case 4后面的代码，返回执行代码的最后一行的值，即200
val s:Any = 4 match {
case 3 =>
    100
case 4 =>
    200
case 5 =>
    ""
}
```

匹配类型

被匹配的值，需要被定义为匹配值类型的共同父类，否则会报编译错误，提示不符合类型的case匹配是多余的。

```
//被匹配对象s，为Int类型，则case匹配中的"a"会报编译错误，因为永远也不会匹配到。
//如果将s的类型改为Any，则可以编译通过
val s:Int = 3
s match {
    case 3 =>
        100
    case 4 =>
        200
    case "a" =>
        300
}
```

普通匹配类型

作用：替代isInstanceOf和asInstanceOf，完成类型判断和类型转换。

被匹配的值，需要被定义为匹配值类型的共同父类，否则会报编译错误，提示不符合类型的匹配值是多余的匹配语句。

```
val a: Any = 99 //匹配到case a: Int
a match {
  //注意，这里的a是match的局部变量a，不是val a。
    case a: Int =>
        println(a to 110)
    case b: Boolean =>
        println("是一个boolean: " + b)
}
```

匹配带守卫

可以添加守卫，增加过滤条件

```
val a: Any = 99
a match {
        // 只匹配大于等于100的整数. 守卫
    case a: Int if a >= 100 && a <= 110 =>
        println(a to 110)
    case b: Boolean =>
        println("是一个boolean: " + b)
}
```

匹配带泛型

带泛型的类型，无法匹配泛型。

例如List，匹配时，无法关心List[T]中的泛型T。需要在匹配中写成List[_]，表示不关心List中的泛型，因为无法关心，做不到。

```
val a: Any = Map(1->2)
        a match {
          //scala的Array用的java的数组，new Int[]，所以可以使用泛型（可见后文<泛型擦除>）
            case a: Array[Int] =>
                println("Array[Int]...")

            // 如果是带泛型的类型，泛型匹配不出来。需要改为 _
            case a: List[_] =>
                println("List[_]....")
          //改成 case m: Map[String, Boolean]，也会匹配上，因为Map是带泛型的类，Map的泛型无法匹配，所以这里写泛型毫无意义
            case m: Map[_, _] =>
                println("Map[_, _]...")

        }
```

泛型擦除

泛型的作用：在写代码的时候，类型更加安全。

泛型起作用的位置：只出现在编译时，编译成字节码之后，泛型就不存在了。也就是说**泛型只存在于源代码**中。

为什么数组的泛型，Array[T]中的T，可以匹配？（存疑）

scala的数组, 本质就是java的数组。 new int[] new double[]

匹配内容

本质：匹配对象

只能匹配有顺序的集合，常用的如下

数组的内容

List的内容

元组的内容

无法匹配没有顺序的，如Set

匹配数组的内容

```
//匹配4个元素的数组
case Array(a, b, c, d)
//匹配某位元素的值为特定值
case Array(10, b, c, d)
//匹配数组，不关心其中的元素
case Array(10, b, _, d)
//匹配数组，只关心前面的n个元素，后面的元素不管有多少个都不关心
case Array(a, b ,_*)

//匹配数组，关系你前面的n个元素，后面的元素不匹配，但也需要获取到
case Array(a, rest@_*) //rest只是1个变量名，可以替换的
//等价于
case a +: rest //有问题，Array不太适合用中置运算符
//中置运算符，原理来自 100 +: arr，将在数组arr前面增加1个元素100
//中置运算符，在Array中，使用得情况不好，出问题。List可以使用中置运算符方法。
```

匹配元组的内容

将嵌套得比较深的元组，方便地取出其中的元素。

```
val t = (1,(2,(3,(4,5))))
println(t._2._2._2._2) //取出最里面的元素 5
//等同于模式匹配
t match {
    case (_,(_,(_,(_,a)))) =>
        println(a)
}
```

同样的，将map嵌套中的元素，方便地取出。

```
val map = Map("a" -> 97, "b" -> 98)
map.foreach(kv => {
    kv match {
        case (k, v) => println(v)
    }
})
```

匹配对象

可以解释匹配内容的原理

匹配对象的本质

1.是去调用这个对象的 unapply 方法, 把需要匹配的值传给这个方法

2.unapply方法, 返回值必须是 Option, 如果返回的Some,则表示匹配成功, 然后把Some中封装的值赋值case语句中 如果返回的是None, 表示匹配失败, 继续下一个case语句

本质：调用对象的 `unapply` 方法，把需要匹配的值传递给这个方法。

判断是否匹配成功：成功返回Some(值)，并拆包出值，不成功返回None，进行下一个case判断。

对象的unapply方法：参数为被匹配的值，返回值必须为Optioin

最终的返回值：如果匹配成功，返回Some(值)，并拆包出值，应用到匹配的代码中。

```
object objectName对象名{
  def unapply(T): Option[T] = {
    //Some包装结果值
    if ... Some(result结果值)
    else None
  }
}
//n，是传给该对象的unapply方法的参数值
//a，是unapply方法的返回值，result结果值
n match {
  //注意：case后面跟的是对象，如果是伴生类和伴生对象，那么就是使用伴生对象中的unapply方法
  case objectName对象名(a) =>
}
```

匹配对象中的伴生对象

匹配对象，就是调用伴生对象的unapply方法

```
class Person(val name: String, val age: Int)

object Person {
    def unapply(p: Person) = {
        if (p == null) None
        else Some((p.name, p.age))
    }
}

object MatchObject2 {
    def main(args: Array[String]): Unit = {
        val p = new Person("zs", 20)

        // 通过模式匹配的方式把  name和age来匹配出来
        p match {
            case Person(name, age) =>
                println(name)
                println(age)
            case _ =>
        }
    }
}
```

痛点：每次使用匹配对象，都需要写一段伴生对象和伴生类的代码。

```
//使用样另类
case class
```

什么时候调用apply，什么时候调用unapply?

`对象()` 等价于 `对象.apply()`

`case 对象()=>` 等价于 `对象.unapply`

样例类

case class

作用：简化模式匹配对象的使用

相当于将伴生类和伴生对象结合在了一起。

自动在伴生对象中，实现了很多方法。其中unapply方法，默认返回伴生类的新对象。

自动实现的方法包括：apply、unapply、equals、hascode

使用场景

1.模式匹配

```
//写法1：匹配对象中的伴生对象

//写法2：
```

2.替代java bean

3.作为进程间通信的通讯协议

spark常用

Akka，scala的ARP通讯工具

发送端，发送样例类对象

接收端，可以直接用case来拿到数据。

所以样例类，变成了一个通讯协议。只有符合样例类的case，才可以解析到数据。

序列化-反序列化，是钝化更专业的说法。

匹配序列

unapply，匹配多元素集合

unapplySeq，匹配序列

unapply和unapplySeq，都需要返回Option

Match3中的，`names:_*`，这是什么？（疑问）

```
//举例：使用样例类和不使用样例类的区别
待补充
```

在变量声明时，使用模式匹配

```
//使用模式匹配，不用写match case
val Array(_, _, a, _*) =foo
val Array(_, _, a, rest@_*) =foo
//可直接使用a
println(a)

def foo = Array(10,20,30,40,50,)
```

在for循环中，使用模式匹配

```
Match6
```

偏函数

偏好，只选择偏好的数进行计算。

只有1个算子支持偏函数，collect

collect 等价于 filter + map

PartialFunction[传入类型, 返回值类型]

偏函数的特点：

1.是一个普通的函数

2.偏函数PartialFunction的方法isDefinedAt，只在算子collect中起作用。所以说，只有collect算子支持偏函数。

3.偏函数PartialFunction的方法isDefinedAt，是偏函数的特点。

```
trait PartialFunction[-A, +B] extends (A => B) { self =>
```

偏函数的定义

方式1（仅限于理解，一般不使用）

最原理的方式

```
val list = List(10,"aaa", true, 20, 30)
    //创建偏函数
    val f = new PartialFunction[Any, Int] {
      //判断哪些元素符合要求，可以计算。也就是偏好是什么
      //作用相当于filter
      //isDefinedAt，只用在collect方法中
      override def isDefinedAt(x: Any): Boolean = x match {
        case _:Int => true
        case _ => false
      }
      //符合要求的元素，使用apply进行计算。
      //作用相当于map
      override def apply(v1: Any): Int = v1 match {
        case a: Int => a * 2
      }
    }
    //collect传入偏函数，即可对集合进行处理
    val list2 = list.collect(f)
    println(list2)
```

方式2（常用）

用大括号`{}`写的case语句，就是偏函数

```
val list = List(10,"aaa", true, 20, 30)
//这里，collect只有1个参数，可以省略小括号
val list3 = list.collect({  //这里大括号{}内的，就是偏函数了
      case a: Int => a * 2
    })
    println(list3)
```

偏函数，最常见的应用

1.可以给变量名起一个有意义的名字

ele._1，无法展示意义

age，可以展示出数据的意义，是年龄

```
Match9
避免了反复元组取值，一堆下划线。
```

2.简化函数写法，将偏函数当作普通函数使用

使用偏函数，是因为偏函数的写法2-大括号中有case，这样的写法方便。

```
Match9 Match10
这里，将偏函数，作为1个普通函数来使用。使用偏函数，是因为偏函数的写法2-大括号中有case，这样的写法方便。
记住，只有collect才会使用偏函数中的灵魂方法，isDefineAt()
```

异常处理

复习java异常

特点：

1.增强了代码的健壮性。

2.使代码变得臃肿

scala异常特点

所有的异常，都可以处理，也可以不处理

处理

1.try...catch...finally

和java不一样的地方：返回值是否包含finally。

scala整个的返回值，是try或者catch中的值。但是有return，会返回return的值，包括在finally中的return。

java的返回值是，是try\catch\finally中的值，会返回finally的值。

2.抛出异常值类型

使用`@throws注解`

3.主动抛出异常

throw new 异常类

泛型

要求：能看懂就可以，因为像泛型的3种变化，我们写不出来。

泛型的介绍

泛型类

泛型函数

区别：作用域范围

scala的泛型，为什么是[]？java的泛型，使用的是`<>`

scala原生支持xml，标签都是用`<>`，所以尖括号使用在了xml标签上。

泛型的上界

注意：泛型不能进行隐式转换。隐式转换的类型符合泛型要求，也没有用。

例如，上界为Ordered，Int有隐式转换变为Ordered的子类，但是Int不符合上界为Ordered。

泛型的下届

有问题，和常规理解有不同，需要看视频（疑问）

泛型的3个变化

不变，默认的

协变，[+T]

逆变，[-T]

常用集合的泛型变化类型

数组Array[T]，是不变的

列表List[+T]，是协变的

Set[+T]，是协变

Map[+T, T]，key是协变，value是不变

上下文界定（泛型）

implicitly，召唤隐式参数

用上下文泛型，会丢失隐式参数的参数名，无法直接调用。需要用implicitly，召唤隐式参数。

[T:Ordering]，这里的Ordering不要加泛型，不能写[T:Ordering[T]]，这样是多重泛型，错误写法。

视图界定

已经过时，但是Spark中，有些地方用到了。需要了解。

IDEA会自动提示，将视图界定，更改为隐式参数。

```
ViewBound
```

其他零碎知识

中置运算符

1 + 2，在中间的运算符，+

一元运算符

后置

1 toString

不写参数，省略`.`

前置

`+5`，`-5`，`!false`取反，`~2`按位取反

能定义前置的运算符，只有前面这4个，`+ - ! ~`

前置运算符，需要在定义函数时，使用`unary_`作为方法名的前缀

```
//例如： 实现 !n，返回10-n的值
//unary_作为前缀，加上!
//注意，能作为前置运算符的，只有4个符号。
def unary_! : Int = 10 -n
```

apply

任何对象都可以使用

其中，对象指的是，伴生对象、普通对象、类的对象

update

添加下标的类，默认`对象(index) = value`，会调用update函数。

user(0) = 100

总结scala中，几个默认使用的函数。

面试

scala问得不多，也不会很难，还是以Java面试为主

scala特有的东西是重点，如柯里化，闭包

其他的，如集合知识，还是问的Java

补充

literal（字面量）、常量、变量

字面量（literal）是用于表达源代码中一个固定值的表示法（notation）

具体举例：由字母，数字等构成的字符串或者数值，字面量只能作为右值出现（右值是指等号右边的值）

```
int a = 123
//a为左值，123为右值
```

常量和变量都属于变量

常量，赋过值后不能再改变的变量

变量，可以再进行赋值操作。

IDEA快捷键

查看方法的栈 ctrl + alt + h

撤销 ctrl + z

返回撤销操作 ctrl + shift + z

获取对象内存地址值

System.identityHashCode()

哈希值的本质

默认情况下，现有内存地址值，才有哈希值。先有hash值，才有hashCode

若更改了哈希值的计算，则可能就不是拿内存地址值计算的了

可查看类编译后的成员

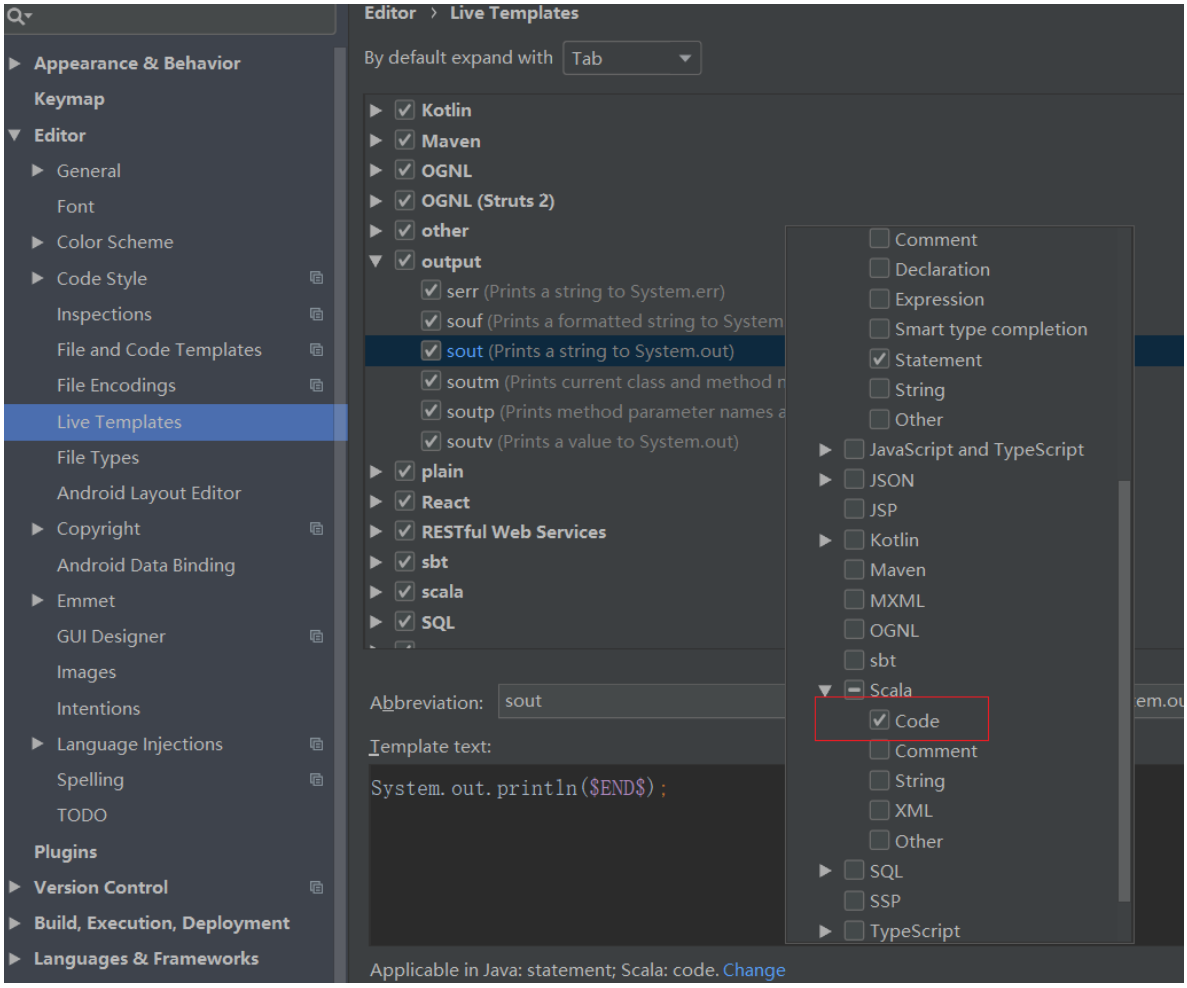
通过cmd，javap -p 类文件名（编译后的文件）

使用问题

IDEA中配置scala后，无法正常使用短命令，如sout

问题：sout会显示为StringIndexOutOfBoundsException，而不提示为System.out.println()

解决方法：



IDEA设置settings--Live Templates。选择Applicable in，勾选Scala--code