**第二课：Scala面向对象彻底精通及Spark源码阅读**

杭州-Frank

# Scala中的类、object实战详解

Scala是纯面向对象语言。谈面向对象，一定要谈类，类把数据和代码封装起来。

## 类

跟Java一样，Scala也是用关键字class来定义类。示例如下：

scala> class HiScala{

| private var name = "Spark"

| def sayName(){println(name)}

| def getName = name

| }

defined class HiScala

以上代码，定义了一个名称为HiScala的类，默认情况下是public级别，所以public关键字可以不写。

* 定义了一个属性name，为可变变量，访问级别为private，在类的外面不能被访问！
* 定义一个函数：sayName
* 定义一个函数：getName

在Scala中，变量与类中的方法是同等级的，可以直接相互赋值。

## 创建类实例

scala> val scal = new HiScala

scal: HiScala = HiScala@1c655221

//此时，scal就是HiScala类的一个实例。但是，在Scala中我们一边不会用new来创建类的实例，而是用apply工厂方法模式来创建。

//调用scal的sayName方法，用于打印出name成员的值。

scala> scal.sayName()

Spark

//由于sayName没有传参数，所以可以把括号去掉，这样更简洁。

scala> scal.sayName

Spark

//调用getName方法，访问name成员的值

scala> scal.getName

res2: String = Spark

//此时确实返回了name的值。

//该示例中name是私有的，所以不能直接访问scal实例的name，如下访问就出错了。

scala> scal.name

<console>:10: error: variable name in class HiScala cannot be accessed in HiScala

scal.name

^

## get与set

Scala的get和set跟Java的get和set有很大的差异。

在Scala中，如果给变量前定义了private，那么Scala解释器会给这个变量自动生成private的get和set方 法；

如果变量前没有定义了private，那么解释器会给这个变量自动生成public的get和set方法，这样就可以直接访问该类的实例对象的成员变量，实际访问的是它的set和get方法。

//我们重新改造上面的类，把属性name前面的private访问级别去掉。

scala> class HiScala{

| var name = "Spark" //去掉了private关键字

| def sayName(){println(name)}

| def getName = name

| }

defined class HiScala

scala> val scal = new HiScala

scal: HiScala = HiScala@1e6d1014

//访问name属性, 值为Spark

scala> scal.name

res4: String = Spark

//此时虽然是访问属性name，但其实不是直接访问var指定的变量name，而是Scala解释器自动给name生成public级别的get和set方法。

//修改name属性

scala> scal.name="Scala"

scal.name: String = Scala

//再次访问name属性, 值已经变为Scala

scala> scal.name

res5: String = Scala

## 自定义的get和set

scala> class Person {

| private var myName = "Flink"

| def name = this.myName //自定义get方法

| def name\_=(newName : String){ //自定义set方法，注意下划线和等号之间没有空格！

| myName = newName

| println("Hi " + myName)

| }

| }

defined class Person

// luck为Person 类的实例

scala> val luck = new Person

luck: Person = Person@6a5fc7f7

//通过自定义的get方法，访问到了myName属性

scala> luck.name

res0: String = Flink

//调用自定义的set方法，修改了myName属性

scala> luck.name = "Spark"

Hi Spark

luck.name: String = Spark

//再次访问myName属性，发现属性值已经变成Spark

scala> luck.name

res2: String = Spark

//修改上面的示例，仅仅暴露属性的get方法，没有为其复写set方法。但是提供一个名为update的方法用来修改该属性的值。

scala> class Person {

| private var myName = "Flink"

| def name = this.myName //get方法

| def update(newName : String){

| myName = newName

| println("Hi " + myName)

| }

| }

defined class Person

// luck为Person 类的实例

scala> val luck = new Person

luck: Person = Person@7c469c48

//通过自定义的get方法，访问myName属性值，初始值为Flink

scala> luck.name

res4: String = Flink

//想直接通过set方法来修改myName，会提示出错，因为没有复写set方法。

scala> luck.name="Hadoop"

<console>:9: error: value name\_= is not a member of Person

luck.name="Hadoop"

^

//通过额外提供的update方法来修改属性

scala> luck.update("Hadoop")

Hi Hadoop

//再次查看myName属性值，发现已经变成了Hadoop

scala> luck.name

res6: String = Hadoop

## private[this]

private[this]至关重要的，在Spark源码中随处可见。

代表属性或方法为对象私有！在类私有的基础上更强一层的控制。

//定义一个类:Person

scala> class Person {

| private var myName = "Flink"

| def name = this.myName

| def update(newName : String){

| myName = newName

| println("Hi " + myName)

| }

| def talk(p:Person) = {

| println("hello:"+p.name)

| }

| }

defined class Person

//定义 p1为Person对象

scala> val p1=new Person

p1: Person = Person@14555e0a

//定义 p2为Person对象

scala> val p2=new Person

p2: Person = Person@1b2abca6

//把p1的myName 属性值改为p1

scala> p1.update("p1")

Hi p1

//把p2的myName 属性值改为p2

scala> p2.update("p2")

Hi p2

//查看p1的myName 属性值，此时已经改为p1

scala> p1.name

res14: String = p1

//查看p2的myName 属性值，此时已经改为p2

scala> p2.name

res15: String = p2

//调用p1的talk方法，传入p2对象，打印出p2的myName 属性值

scala> p1.talk(p2)

hello:p2

//我们修改一下代码，看看下面代码：

scala> class Person {

| private[this] var name = "Flink"

| def update(newName : String){

| name = newName

| println("Hi " + name)

| }

| def talk(p:Person) = {

| println("hello:"+p.name)

| }

| }

<console>:15: error: value name is not a member of Person

println("hello:"+p.name)

^

//此时，在定义类时就报错！因为talk方法中p参数是Person类的对象，而name属性被限制为private[this]，所以只能在Person类内部使用，Person类的对象无权使用。也就是说在Person类中，update方法中可以使用name，但是talk方法访问对象p的name是不允许的！

// private[this]改成private之后，下面的写法就能正常定义，代码如下：

scala> class Person {

| private var name = "Flink"

| def update(newName : String){

| name = newName

| println("Hi " + name)

| }

| def talk(p:Person) = {

| println("hello:"+p.name)

| }

| }

defined class Person

scala>

## 构造器的重载

scala> class Person {

| private[this] var name = "Flink"

| private[this] var age = 10

| def update(newName : String){

| name = newName

| println("Hi " + name)

| }

|

|

| //重载的构造器，首先调用默认的构造器

| def this(name:String){

| this()

| this.name=name

| }

|

| //重载的构造器，调用上面已经存在的构造器

| def this(name:String, age:Int){

| this(name)

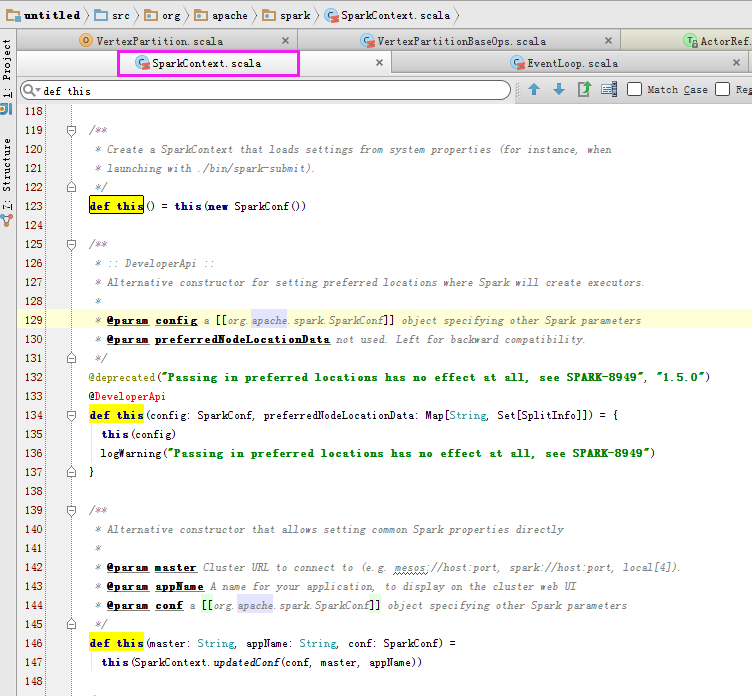
| this.age=age

| }

| }

defined class Person

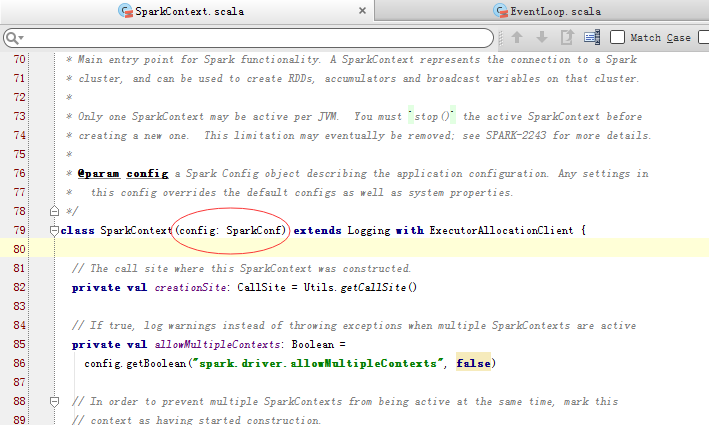
在Spark源码中，构造器重载非常常见，如下图所示：



（图一）

从图一的源代码中可以看出，SparkContext类重写了有很多构造器，在创建SparkContext实例对象时，传递不同的参数来调用相应的构造器。

在Spark中，与类名放在一起的构造器为默认构造器，如下所示：



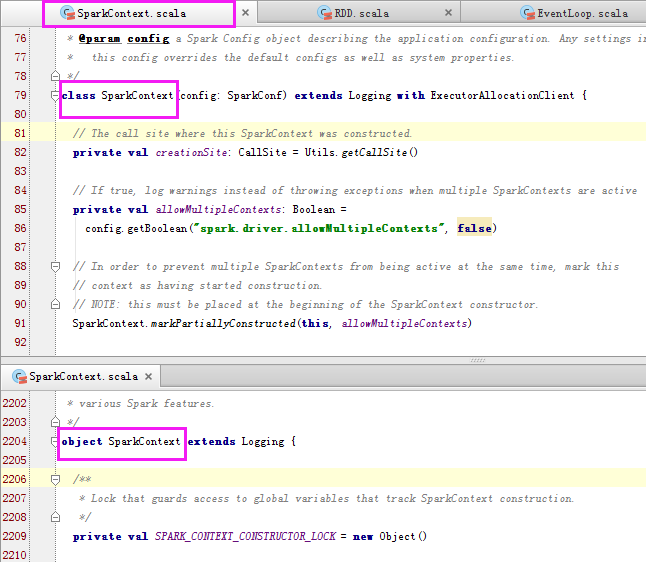
（图二）

默认构造器可以带参数也可以不带参数，在图二所示的源代码SparkContext类中，默认构造器带有一个参数。初始化类成员的工作，都放在默认构造器中完成。在SparkContext类中，creationSite、allowMultipleContexts等变量，就是在SparkContext类的默认构造器中完成。

## object

定义一个类同名的object对象，里面存放静态的成员或者方法，把该object对象称之为该类的**伴生对象**。

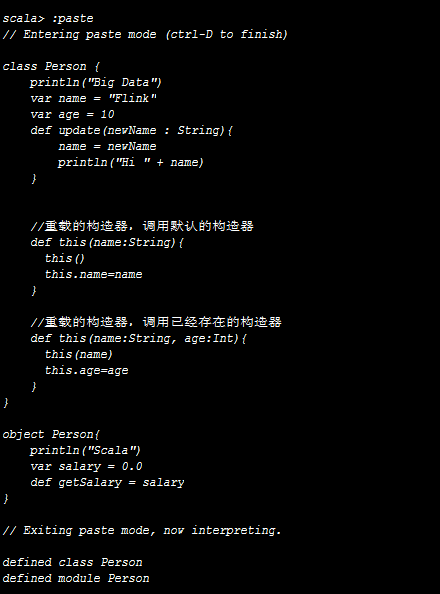
查看SparkContext源码，如下图三所示：



（图三）

在图三中，SparkContext object为SparkContext类的伴生对象，里面存放一系列静态的成员和静态的方法。而且，当我们第一次调用SparkContext时，SparkContext伴生对象会被执行一次，仅此一次。

下面是一个示例：



（图4）

上图中object Person是class Person的伴生对象，class Person称之为object Person的伴生类。伴生对象适合在里面定义一些工具方法，以及存放一些全局唯一的常量，节省空间。

//第一次调用：Person伴生对象的getSalary方法

//此时，我们发现：Person伴生对象被初始化，打印出字符串Scala。

scala> Person.getSalary

Scala

res34: Double = 0.0

//第二次调用：Person伴生对象的getSalary方法

//此时，我们发现：Person伴生对象不再需要初始化，没有打印出字符串Scala。

scala> Person.getSalary

res35: Double = 0.0

以上示例中确实验证了：伴生对象会被执行一次，仅此一次。

Scala中，我们在定义类的对象时， 一般都不会用new类名，再传入参数的方式来定义。而是直接用类名，或者类名加参数的方式。例如，我们重温下面的示例：

我们构造一个数组，其实是调用Array的伴生对象object Array的apply方法。

scala> val array=Array(1,2,3)

array: Array[Int] = Array(1, 2, 3)

scala> val array=Array.apply(1,2,3)

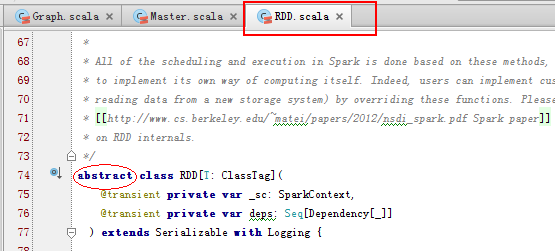
array: Array[Int] = Array(1, 2, 3)

以上两种定义数组的方式效果是一样的。apply方法就是当前类的伴生对象的工厂方法。延伸一下，Java水平比较高才编程人员，在构造Java对象时，一般来说不会直接new一个类，而是通过工厂方法模式来创建。而Scala语言中，天生就支持这次模式，所以在具体类对象构造时，一般都是在伴生类的伴生对象的apply方法中去实现！这样可以控制对象的生成。

# Scala中的抽象类、接口实战详解

## abstract

抽象类使用abstract关键字，以Spark源码RDD类为示例，如下图5所示：



（图5）

子类去继承抽象类，跟Java一样，也是使用extends关键字。如JdbcRDD就继承了RDD抽象类，如下：

class JdbcRDD[T: ClassTag](

sc: SparkContext,

getConnection: () => Connection,

sql: String,

lowerBound: Long,

upperBound: Long,

numPartitions: Int,

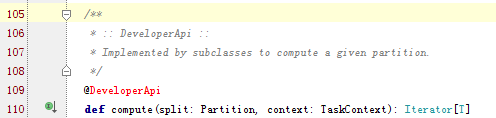
mapRow: (ResultSet) => T = JdbcRDD.resultSetToObjectArray \_)

**extends** RDD[T](sc, Nil) with Logging {

那么JdbcRDD就可以使用父类RDD所有它可以使用的属性和方法。

当然，子类也可以覆盖父类的属性和方法。注意，这里面有个前提，父类的属性和方法没有加final。

一个抽象类，那么里面肯定是定义了某个方法，但是没有又实现体，只有方法的说明。比如：

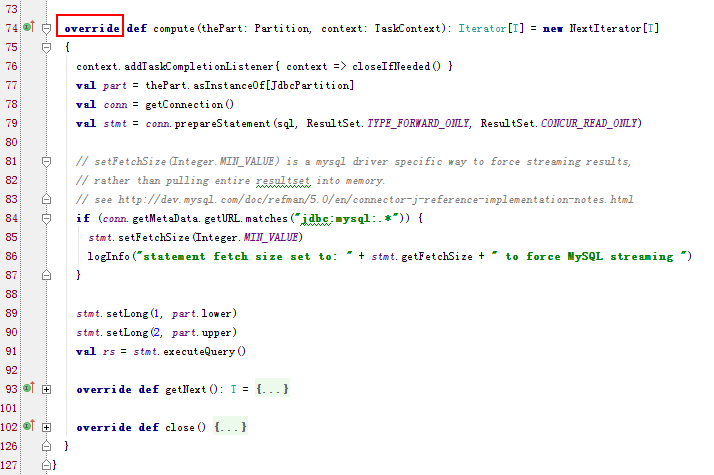


（图六）

上图代码块中，compute就是RDD抽象类的一个方法，只有说明，没有实现体。

## override

在上面JdbcRDD子类中，因为JdbcRDD继承了RDD抽象类，那么在JdbcRDD中一定要复写compute方法。如下所示：



使用override关键字表示复写父类的该方法。除了覆盖父类的方法，也可以使用override来覆盖父类的val属性。父类在定义属性时没有给具体的值，就是抽象属性，在子类中，子类必须覆盖该属性。

## trait

相当于Java的接口。在trait定义的类中可以定义抽象方法，但是又没有具体的实现。子类可以使用extends关键字来继承该父类。

以Spark源码中的RDD类为示例，RDD继承了Serializable和Logging两个父类。一个子类如果继承多个父类，那么继承的第一个父类前使用extends关键字，后面的父类使用with关键字。RDD继承的这两个父类，在定义时都使用trait关键字来定义。Scala语法中，不允许一个子类同时继承多个抽象父类，但是允许同时继承多个trait父类。

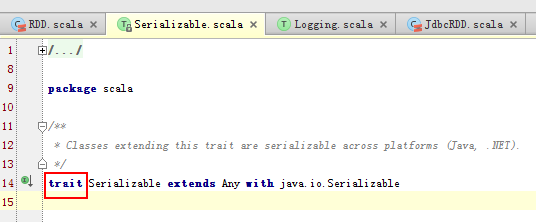
abstract class RDD[T: ClassTag](

@transient private var \_sc: SparkContext,

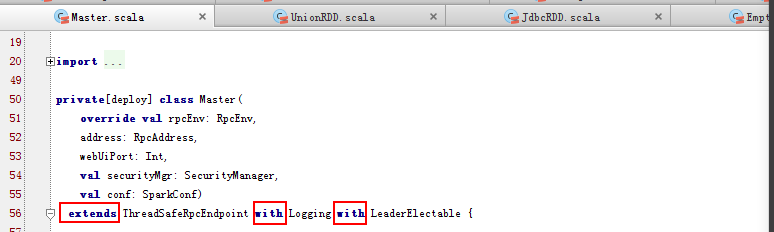
@transient private var deps: Seq[Dependency[\_]]

) extends Serializable with Logging {

下面是Serializable类的源码：



trait的用途，更多的是作为工具方法的容器。我们把通用的功能，放在trait中。一个子类，如果需要把很多通用的功能都混进来，那么就需要继承多个trait父类，继承的第一个父类使用extends，后面的父类使用with。Spark源码Master类就是一个示例，继承了多个父类。



# 综合案例及Spark源码解析

阅读Spark源码中的SparkContext类，熟悉第一课和本课的知识点。