Show Submission Credentials

P4. Introduction to Scala A basic introduction of Scala for Java programmer

41 days 23 hours left

✔ Introduction to Scala

Introduction to Scala

Introduction to Scala

Information

Learning Objectives

- Discuss the differences between Object-oriented and Functional programming, and how Scala is both.
- Discuss the differences between class, case class, object, companion object and trait.
- Recall basic functional operations and their use cases - map, flatMap, filter, and reduce.
- Recall the basic language syntax - loops, if, function definition, pattern matching.

Most people think Scala has a steep learning curve. That's true if you want to become an expert. For a beginner with basic Java skills, one can treat Scala as a version of Java with less verbose syntax, which is not as hard to learn. This primer attempts to give you a quick introduction to the Scala language, which could be useful in the Spark project and the ETL of the group project.

Why Scala?

Scala is becoming a mainstream programming language. It is widely used in industry and open-source software, and its adoption is growing fast. Here is a link about Scala popularity analysis (http://appliedscala.com/blog/2016/scala-popularity/). You can also check StackOverFlow's 2019 Developer Survey (https://insights.stackoverflow.com/survey/2019) - it ranks first as the top paid programming language in the US. Scala has concise syntax (compared to Java), and is suitable for dealing with concurrency. These are some of the reasons why Twitter and LinkedIn use Scala in their core products. Many popular open-source systems are written in Scala, such as Apache Spark.

What is Scala?

Scala is a "Hybrid" language. First of all, like Java, Scala is an object-oriented language running in a JVM. It means that Java and Scala classes can be freely mixed, no matter whether they reside in different projects or in the same project. On the other hand, Scala is also a functional language. As you get familiar with Scala, you will find that it is easy to program compact code that is easy to reason about. Here is a good talk (https://www.youtube.com/watch?v=3jg1AheF4n0) given by the designer of Scala, Martin Odersky.

Environment Set Up

This is a hands-on, example-based primer. You can either install Scala on your local machine, on a VM, or launch a zeppelin cluster (Refer to the Zeppelin for Apache Spark primer), and follow along in the Scala shell, notebook, IDE or text editor.

Installing and running the Scala Shell

- Download and install (https://www.scala-lang.org/download/) Scala 2 on your operating system.
- Make sure that the **scala/bin** directory is in the PATH environment variable.
- Open a command shell in your operating system.
- Type `scala` followed by the `Enter` key.
- Use `:q` to quit and `:paste` to paste code snippets.

Setting up a Dev Environment

Java IDEs such as IntelliJ and Eclipse offer Scala support. You can find the Intellij plugin for Scala here (https://www.jetbrains.com/help/idea/discover-intellij-idea-for-scala.html). As for build tools, you can use Maven to manage packages and build your projects. If you are using Scala in the Spark Project, we recommend using Maven to manage your dependencies; we have provided skeleton code for you to do so.

Hello World

As the first example, we use the classic “Hello, world!” program to demonstrate the differences between Scala and Java.

Scala version:

```
object HelloWorldScala {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Java version:

```
public class HelloWorldJava {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

As you can see, the structure of the Scala version is very similar to the Java version. Let's go through the differences one by one.

Singleton Object

You might have noticed that the main method is not declared as `static` here. This is because static members (methods or fields) do not exist in Scala. Rather than defining static members, the Scala programmer declares these members in **singleton objects**.

Keyword `object` declared a singleton object, which is a class with a **single** instance. The declaration above thus declares both a class called `HelloWorldScala` and an instance of that class, also called `HelloWorldScala` . This instance is lazily initialized and it will be created the first time it is used.

Classes

Apart from singletons, you can also create multiple objects by defining classes in Scala.

Here is an example for a point class:

```
class Point(var x: Int = 0, var y: Int = 0) {

  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
  }

  override def toString: String =
    s"($x, $y)"
}

val point1 = new Point
point1.x  //0
println(point1)  //(0, 0)

val point2 = new Point(2, 3)
point2.x  //2
println(point2)  //(2, 3)

val point3 = new Point(y = 100)
point3.move(50, -30)
println(point3)  //(50, 70)
```

The `Point` class has four members: the variables `x` and `y` and the methods `move` and `toString`. The constructor is defined in the class signature and it can have optional parameters by providing default values. In this `Point` class, `x` and `y` have the default value of 0 unless specified in the constructor.

Companion Objects

The object with the same name as a class or a trait (trait is the interface in Scala) and is defined in the same source file as the associated file or the trait is defined as companion object. An analog to a companion object in Java is having a class with static variables or methods. In Scala you would move the static variables and methods to its companion object.

In Scala, everything inside the companion object does not belong to a specific runtime instance of the class, but is available from a static context. Meanwhile, any variable or method inside a class is available to a specific instance, but not available from a static context.

```
class Main {
    def nonStaticMethod() {
        println("nonStaticMethod");
    }
}

// companion object
object Main {
    // a "static" variable
    val STATIC_FINAL_CONSTANT = "const"
    // a "static" method
    def staticMethod() {
        println("staticMethod");
    }
}

var mainInstance : Main = new Main();
mainInstance.nonStaticMethod(); // not Main.nonStaticMethod()

Main.staticMethod(); // not mainInstance.staticMethod()
```

The mechanism of companion object makes a clear separation of static and non-static attributes and methods in a class.

`public` by Default

In Scala, a class is not explicitly declared as public. A Scala source file can contain multiple classes, and all of them have public visibility. Methods are also public **by default**.

`Unit` v/s `void`

Instead of `void` in Java, use `Unit` as the analog of void in Scala. The example is a concise version of this:

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

In Scala, the **type** of a variable or function is always written **after** the name of the variable or function.

Standard Output Method

To print a value, use `print` or `println` function. There is also a `printf` function with a C-style format string:

```
printf("Hello, %s! You are %d years old.\n", "Fred", 42)
```

You can also add variables to strings as follows:

```
val name = "Fred"
val age = 42
s"Hello, ${name}! You are ${age} years old."
```

Semicolon

In Scala, semicolons are only required if you have multiple statements in the same line.

Array

`Array[String]` represents a String Array in Scala. You can create an array in Scala as follows:

```
val nums = new Array[Int](10) // Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
val s = Array("Hello", "World")
```

Use `()` instead of `[]` to access elements

```
print(s(0)) // Hello
```

Declaring Values and Variables

In the Spark example, `val` occurs before every variable name. A value declared with val is a constant and you cannot modify its value. To declare a variable whose values can vary, use a `var`.

```
var counter = 0 // declare an Int
counter = 1 // OK, it can be changed as a var

val language = "Scala" // declare a constant String
language = "Java" // error: reassignment to val
```

In Scala, you are encouraged to use a `val` unless you really need to change the values. Note that you don't need to specify the type of a value or variable. It is inferred from the type of the expression with which you initialize it. There will be an error if you try to declare a value or variable without initializing it.

However, you can specify the type if necessary. The type name follows the variable name with a colon. For example:

```
val greeting: String = "hello!"
```

Danger

Since `var` is mutable, it should be used with caution in parallel. Let's look at an example:

```
object Increment {
  var num = 0
  def inc(): Unit = {
    num = num + 1
  }
}

class IncrementThread extends Runnable {
  def run(): Unit = {
    for (i <- 1 to 1000) {
      Increment.inc()
    }
  }
}

object Threading {
  def main(args: Array[String]): Unit = {
    val threads = for (i <- 1 to 5) yield new Thread(new IncrementThread)
    for (thread <- threads) thread.start()
    for (thread <- threads) thread.join()
    print(Increment.num)
  }
}
```

The output of this program is indeterminate. The result can be some value which is smaller than 5000 (eg. 4984). This is because **race condition** could happen to the variable `num` due to lack of synchronization.

In Scala, `val` is preferred in parallel programming since it is immutable. When doing RDD manipulations in Spark programming, You should especially be cautious with using `var` when composing lambda functions, which will be executed in parallel when passed to higher-order functions.

Types

In contrast to Java, all values in Scala are objects. There is no distinction between primitive types and class types in Scala. You can invoke methods on numbers, for example:

```
1.toString()
```

Here is the documentation for the Scala class hierarchy (https://docs.scala-lang.org/tour/unified-types.html).

Block Expressions and Assignments

Similar to Java, a block statement is a sequence of statements enclosed in `{}`. However, it has a value in Scala, which is the value of the last expression.

```
val x = 5
val y = 7
val distance = {
  val dx = x - 2;
  val dy = y - 3;
  math.sqrt(dx * dx + dy * dy)
}
// distance: Double = 5.0
```

The value of a `{}` block is the last expression in it, e.g. `sqrt(dx * dx + dy * dy)` in the example above. The variables `dx` and `dy`, which were only needed as intermediate values in the computation, are hidden and not accessible from the rest of the program outside the block.

Control Flow

`if` expression

Unlike Java, almost all constructs have values in Scala. This is a feature of functional languages, where programs are viewed as computing a value. Yes, an `if` expression has a value. For example:

```
val s = if (x > 0) 1 else -1
# in Java, it will be:
# int s = x > 0 ? 1 : -1
```

In this example, the expression has the type `Int` because both branches are of type `Int`. You can understand the `if` in Scala as the combination of the `?` and `:` operator in Java.

Another example:

```
if (x > 0) "positive" else -1
```

If the branches are of different types, the expression has a common **supertype**, `Any`. `Any` is a basic type, which is something similar to `Object` in java.

If the `else` branch is omitted, the type of the expression will be set to `Any`, and the value of the `else` branch will equal to `Unit`. For example:

```
if (x > 0) 1
if (x > 0) 1 else ()
```

The expressions above are the same, and `()` is a `Unit` as mentioned above.

Loops

Scala has the same `while` and `do while` loops as Java.

```
while (n > 0) {
  r = r * n
  n -= 1
}

do {
  r = r * n
  n -= 1
} while (n > 0)
```

The above examples set `r` to the product of 1, 2, ..., n. The type of these loops are `Unit`, written as `()` as they result in no value. Loops differentiate Scala from purely functional languages which mostly rely on recursion to perform the same work.

But `for` has a different semantic from its Java counterpart.

```
for (i <- 1 to n) r = r * i
```

The call `1 to n` returns a **Range (http://www.scala-lang.org/api/2.12.0/scala/collection/immutable/Range.html)** of the numbers from 1 to n (inclusive). The construct `for (i <- expr)` makes the variable `i` traverse all values of the expression to the right of the `<-`.

Use the `until` method instead of the `to` method, if you don't want to include the upper bound.

```
val s = "Hello"
var sum = 0
for (i <- 0 until s.length) // Last value for i is s.length - 1
    sum += s(i)
```

You will notice that loops are not used in Scala as often as in other languages. You can often process the values in a sequence by applying a function on all of them, like a **map function** in MapReduce.

Note: the `break` and `continue` statements do not exist in Scala.

Functions

To define a function, you specify the function's name, parameters, and body like this.

```
def fac(n : Int) = {
  var r = 1
  for (i <- 1 to n) r = r * i
  r
}
```

You must specify the **types** of all the parameters. However, as long as the function is **not recursive**, you do not need to specify the return type.

You can still specify the return type if you think it's necessary. Similar to declaring a variable, you can add the return type with a colon at the end of the function signature.

```
def half(n: Int): Double = {
  n * 0.5
}
```

There is no need for the `return` keyword in this example. The last expression of the block becomes the value that the function returns. It is possible to use `return` as in Java, to exit a function immediately, but that is not commonly done in Scala.

Remember to add `=` after parameters, except for a function without return value, like the HelloWorld example. The Scala compiler determines the return type from the type of the expression to the right of the `=` symbol.

Higher-Order Functions

In a functional programming language, functions are first-class citizens that can be passed around and manipulated just like other data types.

```
import scala.math._
val num = 3.14
val fun = ceil _
```

The `_` turns the `ceil` method into a function. You can call the function as `fun(num)` and the value of this expression will be `4.0`.

The only difference is that `fun` is a variable containing a function, not a fixed function. Your can give `fun` to another function:

```
Array(3.14, 1.42, 2.0).map(fun) // Array(4.0, 2.0, 2.0)
```

The `map` method, which is a higher-order function that accepts another function, applies it to all the values in an array, and returns an array with the function values.

Many methods in Scala accept functions as parameters. You can find out more about functions that accept and return functions here (https://docs.scala-lang.org/tour/higher-order-functions.html).

Pure Functions

Pure function is a concept in Functional Programming. A pure function is a function where the return value is only determined by its input values without any observable side effects.

For example, the function

```
def addOne(x: Int): Int = x + 1
```

only takes the given parameter, add by one and then return, without printing, asking for user inputs or modifying program states.

Given the same input, a pure function will always produce the same output, and it does neither rely on any external states nor produce any side effects. With these features, pure functions are able to produce active parallel replication in parallel applications.

This idea is applied to RDD transformations (which you will learn about in the Spark project) in a Spark job. If you view your transformations as pure functions mapping from one RDD to another, then you can rest assured that the resulting RDD will have the elements from the

source RDD processed only once. Spark also uses this mechanism to guarantee efficient data processing and fault tolerance.

Anonymous Functions

In Scala, you don't have to give a name to each function, just like you don't have to give a name to each number. Here is an anonymous function that multiplies a number by 3:

```
(x: Double) => 3 * x
```

The `=>` converts any `Double x` into a `Double 3 * x`. You can view this as a mapping `x` to `3 * x`. You can pass it to another function:

```
Array(3.14, 1.42, 2.0).map((x: Double) => 3 * x)
// Array(9.42, 4.26, 6.0)
```

When you pass an anonymous function to another function or method, Scala helps you by deducing types when possible, so you can do this:

```
Array(3.14, 1.42, 2.0).map((x) => 3 * x)
```

As a special bonus, for a function that has just one parameter, you can omit the `()` around the parameter:

```
Array(3.14, 1.42, 2.0).map(x => 3 * x)
```

If a parameter occurs only once on the right-hand side of the `=>`, you can replace it with a single underscore:

```
Array(3.14, 1.42, 2.0).map(3 * _)
```

Keep in mind that these shortcuts only work when the parameter types are **known**. You can specify the types in cases where more than one can be inferred:

```
val f = _+_ // error: missing parameter type for expanded function

val f = (_: Int) + (_: Int) // f: (Int, Int) => Int
```

If you want to have more than one statement in the function literal, you can form a block with curly braces and put one statement on each line. As before, the value returned by the function is the result of evaluating the last expression.

```
Array(3.14, 1.42, 2.0).map(x => {
  println(s"x: $x")
  3 * x
})
```

Information

Aside: partially applied functions

The underscore has multiple uses in Scala. In the `ceil _` example above, the underscore is not a placeholder for a single parameter but a parameter list. `ceil _` is an example of a partially applied function, which is an expression where you give some (or none) of the required arguments.

Passing Java Methods as Parameters

Since Java and Scala are compatible with each other, you can pass Java methods into Scala higher-order functions. Here is an example.

We have the Java Class `JavaClass` with a static method `square`, which takes in a integer and returns its squared value.

```
public class JavaClass {
    public static int square(int value) {
        return value * value;
    }
}
```

And we can pass this java static method as a parameter into a Scala higher order function.

```
val arr1 = Array(1, 2, 3, 4, 5)
val arr2 = arr1.map(JavaClass.square)
// arr2: Array[Int] = Array(1, 4, 9, 16, 25)
```

This is a very simple way to take advantage of both Scala's functional programming mechanism and Java's fluent syntax.

Pattern Matching

Scala provides `case` statements, which are optimized to work with pattern matching. Pattern matching looks similar to `switch` in Java but it does provide more functionality. It can be considered as a way of doing dynamic dispatch in object-oriented programming and allows you to call different versions of functions based on the dynamic type of the arguments.

Here is how you can calculate the nth Fibonacci number with `case` statements in Scala:

```
def fibonacci(n : Int) : Int = n match {
  case 0 | 1 => n
  case _ => fibonacci(n-1) + fibonacci(n-2)
}
```

You can also use `case` directly by passing it as an argument into a higher order function. For example, assume you have a list of tuples:

```
val l = List(("a",1),("b",2),("c",3),("d",4))
```

You can use pattern matching like this to convert each tuple into a single tab-delimited string:

```
l.map { case (k, v) => s"$k\t$v" }
```

Which is equivalent to

```
l.map(tuple => s"${tuple._1}\t${tuple._2}")
```

Hints: Spark also implements the `map` method in its RDD API. In a Spark program, you can replace `l` with a Spark RDD: `rdd.map { case (k, v) => s"$k\t$v" }`. The `case` statement matches a tuple.

Information

Note on tuples

Just like in Python, a Scala tuple is an immutable collection enclosed by parenthesis. For example, `(1, 3.14, "Fred")` is a tuple of type `(Int, Double, java.lang.String)`. But its indices start at 1, not 0.

```
scala> val t = (1, 3.14, "Fred")
t: (Int, Double, String) = (1, 3.14, Fred)

scala> val second = t._2
second: Double = 3.14
```

Information

Scala's `match` versus Java's `switch` statements

If you are coming from a Java background, you will probably have noticed some similarity between these two statements. However, there are some nuances of match expressions that you should take note of: 1. Expressions do not fall through into the next case. 2. A `MatchError` exception is thrown if none of the patterns match. You should add a default case to make sure that all cases are covered. 3. `match` is an expression. It should always result in a value.

Case Classes

Case classes help to minimize the amount of boilerplate code when pattern matching on objects. In general, you can pattern match on objects by adding just a `case` keyword to each class.

Defining a Case Class

Here is how you would instantiate objects of a Book `case class`.

```
scala> case class Book(isbn : String)

scala> val ctci = Book("978-0984782864")
scala> val csapp = Book("978-0134092669")
```

Notice how `new` was not required to instantiate this class. Case classes have an `apply` method to take care of object construction.

Comparing between Objects

Instances of case classes are compared by structure, not by reference.

```
scala> val ctci1 = Book("978-0984782864")
ctci1: Book = Book(978-0984782864)

scala> val ctci2 = Book("978-0984782864")
ctci2: Book = Book(978-0984782864)

scala> val eq = ctci1 == ctci2
eq: Boolean = true
```

Pattern Matching on Case Classes

```
scala> abstract class Item
defined class Item

scala> case class Book(isbn : String) extends Item
defined class Book

scala> case class Magazine(issn : String) extends Item
defined class Magazine

scala> val item  = Book("978-0984782864")
item: Book = Book(978-0984782864)

scala> def printSerialNumber(item : Item) = {
     | item match {
     | case Book(b) => println("ISBN: " + b)
     | case Magazine(m) => println("ISSN: " + m)
     | case _ =>
     | }
     | }

scala> printSerialNumber(item)
ISBN: 978-0984782864
```

In this example, we have a library catalog that loans out books and magazines, which is an abstract super class called `Item` which are implemented with 2 separate case classes. `printSerialNumber` takes in an `Item` object and does pattern matching on the case classes `Book` and `Magazine`.

Options

Scala has an `Option` type for optional values. They are either an instance of `scala.Some` or the `None` object. Pattern matching is a good way to unpack the `Option` type. Scala has a number of built-in operations that utilize options. One example is the `get` method below.

```
scala> val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")
capitals: scala.collection.immutable.Map[String,String] =
Map(France -> Paris, Japan -> Tokyo)

scala> capitals get "France"
res0: Option[String] = Some(Paris)

scala> capitals get "North Pole"
res1: Option[String] = None
```

Information

Option vs null

It can be argued that Scala's options improves code readability since it is clear that a variable of type `Option[String]` is an optional `String`, rather than a variable of type `String` that might be null. With the use of options in Scala, an error that would throw a null pointer exception at runtime in Java would become a compilation error.

Higher Order Functions Revisited

In the Functions section, you might have noticed the use of `map` to modify elements of an array. This and other higher-order functions can also be found in other functional languages like ML and Haskell. Spark also relies on such methods to manipulate data.

Mapping with `map` and `flatMap`

`map` builds a new collection by applying a function to all elements of this array.

`flatMap` builds a new collection by applying a function to all elements of this array and using the elements of the resulting collection.

Suppose you have some `l`:

```
val l = List(("a", 1), ("b", 2), ("c", 3), ("d", 4))
```

When `map` returns a list of lists, `flatMap` returns a single list with all element lists combined. You can see the differences between `flatMap` and `map` from the following:

`map`

```
l.map{case (k, v) => List((k, v), (k, v + 10))}
// List[List[(String, Int)]] = List(List((a,1), (a,11)), List((b,2), (b,12)), Li
st((c,3), (c,13)), List((d,4), (d,14)))
```

`flatMap`

```
l.flatMap{case (k, v) => List((k, v), (k, v + 10))}
// List[(String, Int)] = List((a,1), (a,11), (b,2), (b,12), (c,3), (c,13), (d,
4), (d,14))
```

Combining with `reduce` and folding

`reduce` takes a function to reduce the collection to one element.

```
l.reduce{(x, y)=> (x._1 + y._1, x._2 + y._2)}
// (String, Int) = (abcd, 10)
```

Folding (`/:` and `:\`) combines the elements of the collection with an operator. `/:` stands for the fold left operation and `:\` fold right. Each operation involves a start value, a list and an operation. In the following example, fold left is used to combine the elements of a list into a string.

```
val l = List("i", "love", "cloud", "computing")
("" /: l ) (_ + " " + _) // " i love cloud computing"
```

The arguments for fold right appear in a different order. This is how you can produce a result similar to the above with the fold right operation.

```
(l :\ "") (_ + " " + _) // "i love cloud computing "
```

Note: You will also learn about the `flatMap` and `reduce` in the Spark primer. When you know how to use these methods in Scala, it's easy to use them in Spark.

Filtering

`filter` takes a list `l` of type `List[T]` and a function `f` of type `T -> Boolean` . It results in a list of all elements `x` in `l` where `f(x)` evaluated to true.

```
val nums = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

nums.filter(_ % 2 == 0) // List(2, 4, 6, 8, 10)
```

Note: Scala does not have `++` or `--` operators to add and remove one from a variable. Instead, simply use `+= 1` or `-= 1` .

Learn more about Scala from the official documentation (https://www.scala-lang.org/documentation/getting-started.html).

References

Odersky, Martin, Lex Spoon, and Bill Venners. Programming in Scala. Walnut Creek, CA: Artima Press, 2016.