EBook Gratuito

APPRENDIMENTO

Scala Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#scala

Sommario

Di ... 1

Capitolo 1: Iniziare con Scala Language ... 2

Osservazioni ... 2

Versioni ... 2

Examples ... 3

Ciao mondo definendo un metodo "principale" ... 3

Ciao mondo estendendo l'app ... 4

Inizializzazione ritardata ... 4

Inizializzazione ritardata ... 4

Ciao mondo come una sceneggiatura ... 5

Utilizzo di Scala REPL ... 5

Scala Quicksheet ... 6

Capitolo 2: accattivarsi ... 8

Sintassi ... 8

Examples ... 8

Un moltiplicatore configurabile come funzione al curry ... 8

Gruppi di parametri multipli di tipi diversi, che calcolano i parametri di posizioni arbit ... 8

Esecuzione di una funzione con un singolo gruppo di parametri ... 8

accattivarsi ... 9

accattivarsi ... 9

Quando usare Currying ... 10

Un uso del mondo reale di Currying ... 11

Capitolo 3: annotazioni ... 13

Sintassi ... 13

Parametri ... 13

Osservazioni ... 13

Examples ... 13

Usando un'annotazione ... 13

Annotazione del costruttore principale ... 13

Creare le tue annotazioni ... 14

Capitolo 4: Classi di casi 16
Sintassi 16
Examples 16
Case Classality 16
Manufatti di codice generati 16
Nozioni di base di Case Class 18
Classi di casi e immutabilità 18
Crea una copia di un oggetto con determinate modifiche 19
Classi di casi a singolo elemento per sicurezza di tipo 19
Capitolo 5: Classi di tipo 21
Osservazioni 21
Examples 21
Classe di tipo semplice 21
Estendere una classe di tipo 22
Aggiungi le funzioni della classe tipo ai tipi 23
Capitolo 6: Classi e oggetti 25
Sintassi 25
Examples 25
Istanziare istanze di classe 25
Classe di istanziazione senza parametro: {} vs () 26
Singleton & Companion Objects 27
Oggetti Singleton 27
Companion Objects 27
Oggetti 28
Controllo del tipo di istanza 28
Costruttori 30
Costruttore primario 30
Costruttori ausiliari 31
Capitolo 7: collezioni 32
Examples 32
Ordina una lista 32
Crea una lista contenente n copie di x 33

Lista e trucchi di vettore ... 33
Tagliere di raccolta mappe ... 34
Mappa e filtra su una collezione ... 35
Carta geografica ... 35
Moltiplicare i numeri interi per due ... 35
Filtro ... 35
Controllo dei numeri di coppia ... 36
Altri esempi di mappe e filtri ... 36
Introduzione alle Collezioni Scala ... 36
Tipi percorribili ... 37
piega ... 38
Per ciascuno ... 39
Ridurre ... 39
Capitolo 8: Collezioni parallele ... 41
Osservazioni ... 41
Examples ... 41
Creazione e utilizzo di raccolte parallele ... 41
insidie ... 41
Capitolo 9: Combinatori parser ... 43
Osservazioni ... 43
Examples ... 43
Esempio di base ... 43
Capitolo 10: Configurare Scala ... 44
Examples ... 44
Su Linux tramite dpkg ... 44
Installazione di Ubuntu tramite download e configurazione manuale ... 44
Mac OSX tramite Macports ... 45
Capitolo 11: enumerazioni ... 46
Osservazioni ... 46
Examples ... 46
Giorni della settimana usando Enumerazione Scala ... 46

Utilizzo di tratti sigillati e oggetti del caso ... 47
Utilizzo di tratti tratti e casi e di tutti i valori-macro ... 48
Capitolo 12: Espressioni regolari ... 50
Sintassi ... 50
Examples ... 50
Dichiarazione di espressioni regolari ... 50
Ripetizione della corrispondenza di un motivo in una stringa ... 51
Capitolo 13: estrattori ... 52
Sintassi ... 52
Examples ... 52
Estrattori a tupla ... 52
Estrattori Case Class ... 53
Unapply - Estrattori personalizzati ... 53
Notazione Infix dell'estrattore ... 54
Estrattori Regex ... 55
Estrattori trasformativi ... 55
Capitolo 14: Funzione ordine superiore ... 57
Osservazioni ... 57
Examples ... 57
Utilizzo dei metodi come valori di funzione ... 57
Funzioni di ordine elevato (funzione come parametro) ... 58
Argomenti valutazione pigra ... 58
Capitolo 15: funzioni ... 60
Osservazioni ... 60
Differenza tra funzioni e metodi: ... 60
Examples ... 60
Funzioni anonime ... 60
Sottolinea la stenografia ... 61
Funzioni anonime senza parametri ... 61
Composizione ... 61
Relazione con le funzioni parziali ... 62

Capitolo 16: Funzioni definite dall'utente per Hive 63
Examples 63
Una semplice UDF di Hive in Apache Spark 63
Capitolo 17: Funzioni parziali 64
Examples 64
Composizione 64
Utilizzo con `collect` 64
Sintassi di base 65
Utilizzo come funzione totale 66
Utilizzo per estrarre le tuple in una funzione mappa 66
Capitolo 18: Futures 68
Examples 68
Creare un futuro 68
Consumare un futuro di successo 68
Consumare un futuro fallito 68
Mettere insieme il futuro 69
Sequencing and traversing Futures 69
Combina più Futures - Per Comprensione 70
Capitolo 19: Gestione degli errori 72
Examples 72
Provare 72
O 72
Opzione 73
Pattern Matching 73
Utilizzando la mappa e getOrElse 73
Usando la piega 73
Conversione in Java 73
Gestione degli errori originati nei futures 74
Usando le clausole try-catch 74
Convertire le eccezioni in entrambi i tipi di opzioni 75
Capitolo 20: Gestione XML 76
Examples 76

Beautify o Pretty-Print XML 76
Capitolo 21: I flussi 77
Osservazioni 77
Examples 77
Utilizzo di un flusso per generare una sequenza casuale 77
Stream infiniti tramite ricorsione 77
Stream infinito autoreferenziale 78
Capitolo 22: impliciti 79
Sintassi 79
Osservazioni 79
Examples 79
Conversione implicita 79
Parametri impliciti 80
Classi implicite 81
Risolvere i parametri impliciti usando 'implicitamente' 82
Implicita nella REPL 82
Capitolo 23: Iniezione di dipendenza 84
Examples 84
Cake Pattern con classe di implementazione interna 84
Capitolo 24: Interoperabilità Java 85
Examples 85
Conversione di collezioni Scala in raccolte Java e viceversa 85
Array 85
Conversioni di tipo Scala e Java 86
Interfacce funzionali per le funzioni di Scala - scala-java8-compat 87
Capitolo 25: Interpolazione a stringa 89
Osservazioni 89
Examples 89
Hello String Interpolation 89
Interpolazione di stringhe formattate usando l'interpolatore f 89
Utilizzo dell'espressione in stringhe letterali 89
Interpolatori di stringhe personalizzati 90

Interpolatori a stringa come estrattori ... 91
Interpolazione di stringa grezza ... 91
Capitolo 26: Invocazione Dinamica ... 93
introduzione ... 93
Sintassi ... 93
Osservazioni ... 93
Examples ... 93
Campo di accesso ... 93
Metodo chiamate ... 94
Interazione tra accesso al campo e metodo di aggiornamento ... 94
Capitolo 27: JSON ... 96
Examples ... 96
JSON con spray-json ... 96
Rendi disponibile la libreria con SBT ... 96
Importa la libreria ... 96
Leggi JSON ... 96
Scrivi JSON ... 96
DSL ... 96
Read-Write in Case Classes ... 97
Formato personalizzato ... 97
JSON con Circe ... 98
JSON con il gioco di parole ... 98
JSON con json4s ... 101
Capitolo 28: Lavorare con Gradle ... 104
Examples ... 104
Impostazione di base ... 104
Crea il tuo plugin Gradle Scala ... 104
Scrivere il plugin ... 105
Utilizzando il plugin ... 108
Capitolo 29: Lavorare con i dati in uno stile immutabile ... 110
Osservazioni ... 110

Il valore e i nomi delle variabili dovrebbero essere nel caso di cammello inferiore 110
Examples 110
Non è solo val vs. var 110
val e var 110
Collezioni immutabili e mutevoli 111
Ma non posso usare l'immutabilità in questo caso! 111
"Perché dobbiamo mutare?" 112
Creazione e compilazione della mappa dei result 112
Implementazione mutevole 112
Piegando in soccorso 112
Risultato intermedio 113
Più ragionevole la ragionevolezza 113
Capitolo 30: Le tuple 114
Osservazioni 114
Examples 114
Creazione di una nuova tupla 114
Tuple all'interno delle collezioni 114
Capitolo 31: Libreria di continuazioni 116
introduzione 116
Sintassi 116
Osservazioni 116
Examples 116
Le callback sono continui 116
Creazione di funzioni che portano a continuazione 117
Capitolo 32: Macro 119
introduzione 119
Sintassi 119
Osservazioni 119
Examples 119
Annotazione Macro 119
Metodo Macro 120
Errori nelle macro 121

Capitolo 33: Mentre cicli 123
Sintassi 123
Parametri 123
Osservazioni 123
Examples 123
Mentre cicli 123
Cicli Do-While 123
Capitolo 34: Migliori pratiche 125
Osservazioni 125
Examples 125
Mantienilo semplice 125
Non imballare troppo in una sola espressione 125
Preferisci uno stile funzionale, in modo ragionevole 126
Capitolo 35: monadi 127
Examples 127
Definizione di Monade 127
Capitolo 36: Operatori in Scala 129
Examples 129
Operatori integrati 129
Sovraccarico dell'operatore 129
Precedenza dell'operatore 130
Capitolo 37: Opzione Classe 132
Sintassi 132
Examples 132
Opzioni come raccolte 132
Utilizzo dell'opzione anziché di null 132
Nozioni di base 133
Esempio con la mappa 134
Opzioni per le comprensioni 134
Capitolo 38: Pacchi 136
introduzione 136

Examples ... 136
Struttura del pacchetto ... 136
Pacchetti e file ... 136
Convocazione di denominazione del pacchetto ... 137
Capitolo 39: Pattern Matching ... 138
Sintassi ... 138
Parametri ... 138
Examples ... 138
Match Pattern semplice ... 138
Pattern Matching With Stable Identifier ... 139
Pattern Matching su un Seq ... 140
Guardie (se espressioni) ... 141
Pattern matching con classi di casi ... 141
Abbinamento su un'opzione ... 142
Tratti sigillati che corrispondono al modello ... 142
Pattern Matching con Regex ... 143
Pattern binder (@) ... 143
Tipi di corrispondenza del modello ... 144
Pattern Matching compilato come tableswitch o lookupswitch ... 145
Abbinamento di più pattern contemporaneamente ... 145
Pattern Matching su tuple ... 146
Capitolo 40: Per le espressioni ... 148
Sintassi ... 148
Parametri ... 148
Examples ... 148
Basic For Loop ... 148
Di base per la comprensione ... 148
Nested For Loop ... 149
Monadico per le comprensioni ... 149
Scorrere le raccolte utilizzando un ciclo For ... 150
Desugaring per le comprensioni ... 150
Capitolo 41: Programmazione a livello di codice ... 152

Examples ... 152
Introduzione alla programmazione a livello di carattere ... 152
Capitolo 42: Quasiquotes ... 154
Examples ... 154
Creare un albero di sintassi con quasiquotes ... 154
Capitolo 43: ricorsione ... 155
Examples ... 155
Ricorsione di coda ... 155
Ricorsione regolare ... 155
Ricorsione di coda ... 155
Ricorsione senza pila con trampolino (scala.util.control.TailCalls) ... 156
Capitolo 44: Riflessione ... 158
Examples ... 158
Caricamento di una lezione utilizzando la riflessione ... 158
Capitolo 45: Scala.js ... 159
introduzione ... 159
Examples ... 159
console.log in Scala.js ... 159
Funzioni di freccia grassa ... 159
Classe semplice ... 159
collezioni ... 159
Manipolazione del DOM ... 159
Utilizzo con SBT ... 160
Dipendenza da Sbt ... 160
In esecuzione ... 160
In esecuzione con la compilazione continua: ... 160
Compilare a un singolo file JavaScript: ... 160
Capitolo 46: scaladoc ... 161
Sintassi ... 161
Parametri ... 161
Examples ... 162

Semplice metodo Scaladoc ... 162
Capitolo 47: scalaz ... 163
introduzione ... 163
Examples ... 163
ApplyUsage ... 163
FunctorUsage ... 163
ArrowUsage ... 164
Capitolo 48: Scopo ... 165
introduzione ... 165
Sintassi ... 165
Examples ... 165
Ambito pubblico (predefinito) ... 165
Un ambito privato ... 165
Un ambito specifico del pacchetto privato ... 166
Scopo privato dell'oggetto ... 166
Ambito protetto ... 166
Ambito protetto da pacchetto ... 166
Capitolo 49: Se espressioni ... 168
Examples ... 168
Basic If Expressions ... 168
Capitolo 50: sincronizzato ... 169
Sintassi ... 169
Examples ... 169
sincronizzare su un oggetto ... 169
sincronizzare implicitamente su questo ... 169
Capitolo 51: Singoli tipi di metodo astratto (tipi SAM) ... 170
Osservazioni ... 170
Examples ... 170
Lambda Sintassi ... 170
Capitolo 52: Sovraccarico dell'operatore ... 171
Examples ... 171
Definizione degli operatori Infix personalizzati ... 171

Definizione di operatori unari personalizzati ... 171
Capitolo 53: Symbol letterali ... 173
Osservazioni ... 173
Examples ... 173
Sostituzione di stringhe nelle clausole case ... 173
Capitolo 54: Test con ScalaCheck ... 175
introduzione ... 175
Examples ... 175
Scalacheck con scalatest e messaggi di errore ... 175
Capitolo 55: Test con ScalaTest ... 178
Examples ... 178
Ciao World Spec Test ... 178
Cheat sheet Test Spec ... 178
Include la libreria ScalaTest con SBT ... 179
Capitolo 56: Tipi di auto ... 180
Sintassi ... 180
Osservazioni ... 180
Examples ... 180
Semplice esempio di auto-tipo ... 180
Capitolo 57: Tipo di inferenza ... 181
Examples ... 181
Inferenza di tipo locale ... 181
Tipo di inferenza e generici ... 181
Limitazioni all'inferenza ... 181
Prevenire il non inferire ... 182
Capitolo 58: Tipo Parametrizzazione (Generics) ... 184
Examples ... 184
Il tipo di opzione ... 184
Metodi parametrizzati ... 184
Collezione generica ... 185
Definire la lista di Ints ... 185
Definire una lista generica ... 185

Capitolo 59: Tipo Varianza ... 186
Examples ... 186
covarianza ... 186
invarianza ... 186
controvarianza ... 187
Covarianza di una collezione ... 187
Covarianza su un tratto invariante ... 188
Capitolo 60: Tratti ... 189
Sintassi ... 189
Examples ... 189
Modifica impilabile con tratti ... 189
Nozioni di base sui tratti ... 190
Risolvere il problema del diamante ... 190
linearizzazione ... 192
Capitolo 61: Unità di gestione (misure) ... 194
Sintassi ... 194
Osservazioni ... 194
Examples ... 194
Digita gli alias ... 194
Classi di valore ... 194
Capitolo 62: Var, Val e Def ... 196
Osservazioni ... 196
Examples ... 196
Var, Val e Def ... 196
var ... 196
val ... 197
DEF ... 197
funzioni ... 198
Lazy val ... 198
Quando usare 'pigro' ... 199
Sovraccarico Def ... 200

Parametri nominati 200

Titoli di coda **202**

Di

Capitolo 1: Iniziare con Scala Language

Osservazioni

Scala è un moderno linguaggio di programmazione multi-paradigma progettato per esprimere schemi di programmazione comuni in modo conciso, elegante e sicuro per il tipo. Integra perfettamente funzionalità di linguaggi orientati agli oggetti e funzionali .

La maggior parte degli esempi forniti richiede un'installazione di Scala funzionante. Questa è la pagina di installazione di Scala , e questo è l'esempio di 'Come configurare Scala' . scalafiddle.net è una buona risorsa per l'esecuzione di piccoli esempi di codice sul web.

Versioni

Versione	Data di rilascio
2.10.1	2013/03/13
2.10.2	2013/06/06
2.10.3	2013/10/01
2.10.4	2014/03/24
2.10.5	2015/03/05
2.10.6	2015/09/18
2.11.0	2014/04/21
2.11.1	2014/05/21
2.11.2	2014/07/24
2.11.4	2014/10/30
2.11.5	2014/01/14
2.11.6	2015/03/05
2.11.7	2015/06/23
2.11.8	2016/03/08
2.11.11	2017/04/19
2.12.0	2016/11/03

Versione	Data di rilascio
2.12.1	2016/12/06
2.12.2	2017/04/19

Examples

Ciao mondo definendo un metodo "principale"

Inserisci questo codice in un file denominato `HelloWorld.scala` :

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
  }
}
```

Dimostrazione dal vivo

Per compilarlo in bytecode eseguibile da JVM:

```
$ scalac HelloWorld.scala
```

Per eseguirlo:

```
$ scala Hello
```

Quando il runtime di Scala carica il programma, cerca un oggetto chiamato `Hello` con un metodo `main` . Il metodo `main` è il punto di ingresso del programma ed è eseguito.

Nota che, diversamente da Java, Scala non ha bisogno di nominare oggetti o classi dopo il file in cui si trovano. Invece, il parametro `Hello` passato nel comando `scala Hello` riferisce all'oggetto da cercare che contiene il metodo `main` da eseguire. È perfettamente possibile avere più oggetti con i metodi principali nello stesso file `.scala` .

L'array `args` conterrà gli argomenti della riga di comando dati al programma, se presenti. Ad esempio, possiamo modificare il programma in questo modo:

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
    for {
      arg <- args
    } println(s"Arg=$arg")
  }
}
```

Compilalo:

```
$ scalac HelloWorld.scala
```

E poi eseguirlo:

```
$ scala HelloWorld 1 2 3
Hello World!
Arg=1
Arg=2
Arg=3
```

Ciao mondo estendendo l'app

```
object HelloWorld extends App {
  println("Hello, world!")
}
```

Dimostrazione dal vivo

Estendendo il tratto `App` , puoi evitare di definire un metodo `main` esplicito. L'intero corpo dell'oggetto `HelloWorld` viene considerato come "il metodo principale".

2.11.0

> ## Inizializzazione ritardata
>
> Secondo la documentazione ufficiale , l' `App` utilizza una funzionalità denominata *inizializzazione ritardata* . Ciò significa che i campi dell'oggetto vengono inizializzati *dopo* che è stato chiamato il metodo principale.

2.11.0

> ## Inizializzazione ritardata
>
> Secondo la documentazione ufficiale , l' `App` utilizza una funzionalità denominata *inizializzazione ritardata* . Ciò significa che i campi dell'oggetto vengono inizializzati *dopo* che è stato chiamato il metodo principale.
>
> `DelayedInit` è ora **deprecato** per uso generale, ma è *ancora supportato* per `App` come caso speciale. Il supporto continuerà fino a quando non verrà decisa e implementata una funzione sostitutiva.

Per accedere agli argomenti della riga di comando durante l'estensione `App` , utilizzare `this.args` :

```
object HelloWorld extends App {
  println("Hello World!")
  for {
    arg <- this.args
  } println(s"Arg=$arg")
}
```

Quando si usa `App` , il corpo dell'oggetto verrà eseguito come metodo `main` , non è necessario sovrascrivere il `main` .

Ciao mondo come una sceneggiatura

Scala può essere usato come linguaggio di scripting. Per dimostrare, crea `HelloWorld.scala` con il seguente contenuto:

```
println("Hello")
```

Eseguilo con l'interprete della riga di comando (`$` è il prompt della riga di comando):

```
$ scala HelloWorld.scala
Hello
```

Se ometti `.scala` (ad esempio se hai semplicemente digitato `scala HelloWorld`) il runner cercherà un file `.class` compilato con bytecode invece di compilare e quindi eseguire lo script.

> **Nota:** se scala viene utilizzato come linguaggio di scripting, non è possibile definire alcun pacchetto.

Nei sistemi operativi che utilizzano `bash` o terminali di shell simili, gli script Scala possono essere eseguiti utilizzando un 'preambolo di shell'. Creare un file denominato `HelloWorld.sh` e inserire quanto segue come contenuto:

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("Hello")
```

Le parti tra `#!` e `!#` è il 'preambolo della shell', ed è interpretato come uno script di bash. Il resto è Scala.

Una volta salvato il file sopra, devi concederne le autorizzazioni 'eseguibili'. Nella shell puoi fare questo:

```
$ chmod a+x HelloWorld.sh
```

(Nota che questo dà il permesso a tutti: leggi su chmod per sapere come impostarlo per gruppi di utenti più specifici.)

Ora puoi eseguire lo script in questo modo:

```
$ ./HelloWorld.sh
```

Utilizzo di Scala REPL

Quando esegui `scala` in un terminale senza parametri aggiuntivi, apre un interprete REPL (Read-

Eval-Print Loop):

```
nford:~ $ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala>
```

Il REPL consente di eseguire Scala in un foglio di lavoro: il contesto di esecuzione viene mantenuto ed è possibile provare manualmente i comandi senza dover creare un intero programma. Ad esempio, digitando `val poem = "As halcyons we shall be"` sarebbe simile a questo:

```
scala> val poem = "As halcyons we shall be"
poem: String = As halcyons we shall be
```

Ora possiamo stampare il nostro `val` :

```
scala> print(poem)
As halcyons we shall be
```

Nota che `val` è immutabile e non può essere sovrascritto:

```
scala> poem = "Brooding on the open sea"
<console>:12: error: reassignment to val
       poem = "Brooding on the open sea"
```

Ma nel REPL è *possibile* ridefinire un valore `val` (che causerebbe un errore in un normale programma Scala, se è stato eseguito nello stesso ambito):

```
scala> val poem = "Brooding on the open sea"
poem: String = Brooding on the open sea
```

Per il resto della sessione REPL questa variabile appena definita ombreggia la variabile definita in precedenza. REPL sono utili per vedere rapidamente come funzionano gli oggetti o altro codice. Sono disponibili tutte le funzionalità di Scala: puoi definire funzioni, classi, metodi, ecc.

Scala Quicksheet

Descrizione	Codice
Assegna valore int immutabile	`val x = 3`
Assegna un valore int mutabile	`var x = 3`
Assegna valore immutabile con tipo esplicito	`val x: Int = 27`
Assegna il valore valutato pigramente	`lazy val y = print("Sleeping in.")`
Associare una funzione a un nome	`val f = (x: Int) => x * x`

Descrizione	Codice
Associare una funzione a un nome con tipo esplicito	`val f: Int => Int = (x: Int) => x * x`
Definire un metodo	`def f(x: Int) = x * x`
Definire un metodo con digitazione esplicita	`def f(x: Int): Int = x * x`
Definisci una classe	`class Hopper(someParam: Int) { ... }`
Definisci un oggetto	`object Hopper(someParam: Int) { ... }`
Definisci un tratto	`trait Grace { ... }`
Ottieni il primo elemento della sequenza	`Seq(1,2,3).head`
Se interruttore	`val result = if(x > 0) "Positive!"`
Ottieni tutti gli elementi della sequenza tranne prima	`Seq(1,2,3).tail`
Passa attraverso un elenco	`for { x <- Seq(1,2,3) } print(x)`

Ciclo annidato	`for {`
	`x <- Seq(1,2,3)`
	`y <- Seq(4,5,6)`
	`} print(x + ":" + y)`

Per ogni elemento della lista eseguire la funzione	`List(1,2,3).foreach { println }`
Stampa su standard	`print("Ada Lovelace")`
Ordina una lista alfanumericamente	`List('b','c','a').sorted`

Leggi Iniziare con Scala Language online: https://riptutorial.com/it/scala/topic/216/iniziare-con-scala-language

Capitolo 2: accattivarsi

Sintassi

- aFunction (10) _ // Utilizzo di '_' Indica al compilatore che tutti i parametri nel resto dei gruppi di parametri verranno ritentati.
- nArityFunction.curried // Converte una funzione n-arity in una versione al curry equivalente
- anotherFunction (x) (_: String) (z) // Esecuzione di un parametro arbitrario. Ha bisogno del suo tipo esplicitamente dichiarato.

Examples

Un moltiplicatore configurabile come funzione al curry

```
def multiply(factor: Int)(numberToBeMultiplied: Int): Int = factor * numberToBeMultiplied

val multiplyBy3 = multiply(3)_     // resulting function signature Int => Int
val multiplyBy10 = multiply(10)_ // resulting function signature Int => Int

val sixFromCurriedCall = multiplyBy3(2) //6
val sixFromFullCall = multiply(3)(2)    //6

val fortyFromCurriedCall = multiplyBy10(4) //40
val fortyFromFullCall = multiply(10)(4)    //40
```

Gruppi di parametri multipli di tipi diversi, che calcolano i parametri di posizioni arbitrarie

```
def numberOrCharacterSwitch(toggleNumber: Boolean)(number: Int)(character: Char): String =
  if (toggleNumber) number.toString else character.toString

// need to explicitly specify the type of the parameter to be curried
// resulting function signature Boolean => String
val switchBetween3AndE = numberOrCharacterSwitch(_: Boolean)(3)('E')

switchBetween3AndE(true) // "3"
switchBetween3AndE(false) // "E"
```

Esecuzione di una funzione con un singolo gruppo di parametri

```
def minus(left: Int, right: Int) = left - right

val numberMinus5 = minus(_: Int, 5)
val fiveMinusNumber = minus(5, _: Int)

numberMinus5(7)    //  2
fiveMinusNumber(7) // -2
```

accattivarsi

Definiamo una funzione di 2 argomenti:

```
def add: (Int, Int) => Int = (x,y) => x + y
val three = add(1,2)
```

Currying `add` trasforma in una funzione che prende **un** `Int` e restituisce una **funzione** (da **un** `Int` a un `Int`)

```
val addCurried: (Int) => (Int => Int) = add2.curried
//                ^~~ take *one* Int
//                          ^~~~ return a *function* from Int to Int

val add1: Int => Int = addCurried(1)
val three: Int = add1(2)
val allInOneGo: Int = addCurried(1)(2)
```

È possibile applicare questo concetto a qualsiasi funzione che richiede più argomenti. Eseguendo una funzione che accetta più argomenti, la trasforma in una serie di applicazioni di funzioni che prendono **un** argomento:

```
def add3: (Int, Int, Int) => Int = (a,b,c) => a + b + c + d
def add3Curr: Int => (Int => (Int => Int)) = add3.curried

val x = add3Curr(1)(2)(42)
```

accattivarsi

Currying, secondo Wikipedia ,

> è la tecnica di tradurre la valutazione di una funzione che prende più argomenti per valutare una sequenza di funzioni.

Concretamente, in termini di tipi di scala, nel contesto di una funzione che prende due argomenti, (ha arity 2) è la conversione di

```
val f: (A, B) => C // a function that takes two arguments of type `A` and `B` respectively
                   // and returns a value of type `C`
```

a

```
val curriedF: A => B => C // a function that take an argument of type `A`
                          // and returns *a function*
                          // that takes an argument of type `B` and returns a `C`
```

Quindi per le funzioni di arity-2 possiamo scrivere la funzione curry come:

```
def curry[A, B, C](f: (A, B) => C): A => B => C = {
  (a: A) => (b: B) => f(a, b)
```

```
}
```

Uso:

```
val f: (String, Int) => Double = {(_, _) => 1.0}
val curriedF: String => Int => Double = curry(f)
f("a", 1)        // => 1.0
curriedF("a")(1) // => 1.0
```

Scala ci offre alcune funzionalità linguistiche che aiutano con questo:

1. È possibile scrivere funzioni curry come metodi. così `curriedF` può essere scritto come:

```
def curriedFAsAMethod(str: String)(int: Int): Double = 1.0
val curriedF = curriedFAsAMethod _
```

2. È possibile annullare la ricorrenza (ovvero passare da `A => B => C` a `(A, B) => C`) utilizzando un metodo di libreria standard: `Function.uncurried`

```
val f: (String, Int) => Double = Function.uncurried(curriedF)
f("a", 1) // => 1.0
```

Quando usare Currying

Il Currying *è la tecnica per tradurre la valutazione di una funzione che prende più argomenti per valutare una sequenza di funzioni, ognuna con un singolo argomento* .

Questo è normalmente utile quando ad esempio:

1. diversi argomenti di una funzione sono calcolati **in momenti diversi** . *(Esempio 1)*
2. diversi argomenti di una funzione sono calcolati **da diversi livelli dell'applicazione** . *(Esempio 2)*

Esempio 1

Supponiamo che il reddito annuale totale sia una funzione composta dal reddito e da un bonus:

```
val totalYearlyIncome:(Int,Int) => Int =  (income, bonus) => income + bonus
```

La versione al curry della suddetta funzione 2-arity è:

```
val totalYearlyIncomeCurried: Int => Int => Int = totalYearlyIncome.curried
```

Nota nella definizione di cui sopra che il tipo può essere anche visualizzato / scritto come:

```
Int => (Int => Int)
```

Supponiamo che la porzione di reddito annuale sia nota in anticipo:

```
val partialTotalYearlyIncome: Int => Int = totalYearlyIncomeCurried(10000)
```

E ad un certo punto della linea il bonus è noto:

```
partialTotalYearlyIncome(100)
```

Esempio 2

Supponiamo che la produzione automobilistica implichi l'applicazione di ruote per auto e carrozzeria:

```
val carManufacturing:(String,String) => String = (wheels, body) => wheels + body
```

Queste parti sono applicate da diverse fabbriche:

```
class CarWheelsFactory {
  def applyCarWheels(carManufacturing:(String,String) => String): String => String =
          carManufacturing.curried("applied wheels..")
}

class CarBodyFactory {
  def applyCarBody(partialCarWithWheels: String => String): String =
partialCarWithWheels("applied car body..")
}
```

Si noti che il `CarWheelsFactory` sopra `CarWheelsFactory` la funzione di produzione dell'auto e applica solo le ruote.

Il processo di produzione dell'auto assumerà quindi la seguente forma:

```
val carWheelsFactory = new CarWheelsFactory()
val carBodyFactory   = new CarBodyFactory()

val carManufacturing:(String,String) => String = (wheels, body) => wheels + body

val partialCarWheelsApplied: String => String  =
carWheelsFactory.applyCarWheels(carManufacturing)
val carCompleted = carBodyFactory.applyCarBody(partialCarWheelsApplied)
```

Un uso del mondo reale di Currying.

Quello che abbiamo è un elenco di carte di credito e vorremmo calcolare i premi per tutte quelle carte che la società della carta di credito deve pagare. I premi stessi dipendono dal numero totale di carte di credito, in modo che la società li adegui di conseguenza.

Abbiamo già una funzione che calcola il premio per una singola carta di credito e tiene conto del totale delle carte emesse dalla società:

```
case class CreditCard(creditInfo: CreditCardInfo, issuer: Person, account: Account)

object CreditCard {
```

```
  def getPremium(totalCards: Int, creditCard: CreditCard): Double = { ... }
}
```

Ora un approccio ragionevole a questo problema sarebbe quello di mappare ogni carta di credito a un premio e ridurlo a una somma. Qualcosa come questo:

```
val creditCards: List[CreditCard] = getCreditCards()
val allPremiums = creditCards.map(CreditCard.getPremium).sum //type mismatch; found : (Int,
CreditCard) ⇒ Double required: CreditCard ⇒ ?
```

Comunque il compilatore non gradirà questo, perché `CreditCard.getPremium` richiede due parametri. Applicazione parziale al salvataggio! Possiamo parzialmente applicare il numero totale di carte di credito e utilizzare questa funzione per mappare le carte di credito ai loro premi. Tutto quello che dobbiamo fare è curry la funzione `getPremium` cambiandola per usare più liste di parametri e siamo a posto.

Il risultato dovrebbe essere simile a questo:

```
object CreditCard {
  def getPremium(totalCards: Int)(creditCard: CreditCard): Double = { ... }
}

val creditCards: List[CreditCard] = getCreditCards()

val getPremiumWithTotal = CreditCard.getPremium(creditCards.length)_

val allPremiums = creditCards.map(getPremiumWithTotal).sum
```

Leggi accattivarsi online: https://riptutorial.com/it/scala/topic/1636/accattivarsi

Capitolo 3: annotazioni

Sintassi

- @AnAnnotation def someMethod = {...}
- @AnAnnotazione class someClass {...}
- @AnnotatioWithArgs (annotation_args) def someMethod = {...}

Parametri

Parametro	Dettagli
@	Indica che il token che segue è un'annotazione.
SomeAnnotation	Il nome dell'annotazione
constructor_args	(facoltativo) Gli argomenti passati all'annotazione. Se nessuno, le parentesi non sono necessarie.

Osservazioni

Scala-lang fornisce un elenco di annotazioni standard e i relativi equivalenti Java .

Examples

Usando un'annotazione

Questa annotazione di esempio indica che il seguente metodo è `deprecated` .

```
@deprecated
def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Questo può anche essere scritto in modo equivalente come:

```
@deprecated def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Annotazione del costruttore principale

```
/**
 * @param num Numerator
```

```
 * @param denom Denominator
 * @throws ArithmeticException in case `denom` is `0`
 */
class Division @throws[ArithmeticException](/*no annotation parameters*/) protected (num: Int,
denom: Int) {
    private[this] val wrongValue = num / denom

    /** Integer number
     *  @param num Value */
    protected[Division] def this(num: Int) {
      this(num, 1)
    }
}
object Division {
  def apply(num: Int) = new Division(num)
  def apply(num: Int, denom: Int) = new Division(num, denom)
}
```

Il modificatore di visibilità (in questo caso `protected`) dovrebbe venire dopo le annotazioni nella stessa riga. Nel caso in cui l'annotazione accetti i parametri opzionali (come in questo caso `@throws` accetta una causa opzionale), è necessario specificare un elenco di parametri vuoto per l'annotazione: `()` prima dei parametri del costruttore.

Nota: è possibile specificare più annotazioni, anche dello stesso tipo (annotazioni ripetute).

Allo stesso modo con una classe di caso senza metodo di produzione ausiliario (e la causa specificata per l'annotazione):

```
case class Division @throws[ArithmeticException]("denom is 0") (num: Int, denom: Int) {
    private[this] val wrongValue = num / denom
}
```

Creare le tue annotazioni

Puoi creare le tue annotazioni Scala creando classi derivate da scala.annotation.StaticAnnotation o scala.annotation.ClassfileAnnotation

```
package animals
// Create Annotation `Mammal`
class Mammal(indigenous:String) extends scala.annotation.StaticAnnotation

// Annotate class Platypus as a `Mammal`
@Mammal(indigenous = "North America")
class Platypus{}
```

Le annotazioni possono quindi essere interrogate utilizzando l'API di riflessione.

```
scala>import scala.reflect.runtime.{universe ⇒ u}

scala>val platypusType = u.typeOf[Platypus]
platypusType: reflect.runtime.universe.Type = animals.reflection.Platypus

scala>val platypusSymbol = platypusType.typeSymbol.asClass
platypusSymbol: reflect.runtime.universe.ClassSymbol = class Platypus
```

```
scala>platypusSymbol.annotations
List[reflect.runtime.universe.Annotation] = List(animals.reflection.Mammal("North America"))
```

Leggi annotazioni online: https://riptutorial.com/it/scala/topic/3783/annotazioni

Capitolo 4: Classi di casi

Sintassi

- case class Foo () // Le classi di casi senza parametri devono avere una lista vuota
- case class Foo (a1: A1, ..., aN: AN) // Crea una classe case con i campi a1 ... aN
- caso oggetto Barra // Crea una classe caso singleton

Examples

Case Classality

Una funzione fornita gratuitamente dalle classi case è un metodo di `equals` generato automaticamente che controlla l'uguaglianza di valore di tutti i singoli campi membro invece di controllare semplicemente l'uguaglianza di riferimento degli oggetti.

Con le classi ordinarie:

```
class Foo(val i: Int)
val a = new Foo(3)
val b = new Foo(3)
println(a == b)// "false" because they are different objects
```

Con classi di casi:

```
case class Foo(i: Int)
val a = Foo(3)
val b = Foo(3)
println(a == b)// "true" because their members have the same value
```

Manufatti di codice generati

Il modificatore di `case` fa sì che il compilatore Scala generi automaticamente un codice standard di codice comune per la classe. L'implementazione manuale di questo codice è noiosa e fonte di errori. La seguente definizione della classe del caso:

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

... verrà generato automaticamente il seguente codice:

```
class Person(val name: String, val age: Int)
  extends Product with Serializable
{
  def copy(name: String = this.name, age: Int = this.age): Person =
    new Person(name, age)

  def productArity: Int = 2
```

```
  def productElement(i: Int): Any = i match {
    case 0 => name
    case 1 => age
    case _ => throw new IndexOutOfBoundsException(i.toString)
  }

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productPrefix: String = "Person"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[Person]

  override def hashCode(): Int = scala.runtime.ScalaRunTime._hashCode(this)

  override def equals(obj: Any): Boolean = this.eq(obj) || obj match {
    case that: Person => this.name == that.name && this.age == that.age
    case _ => false
  }

  override def toString: String =
    scala.runtime.ScalaRunTime._toString(this)
}
```

Il modificatore di `case` genera anche un oggetto associato:

```
object Person extends AbstractFunction2[String, Int, Person] with Serializable {
  def apply(name: String, age: Int): Person = new Person(name, age)

  def unapply(p: Person): Option[(String, Int)] =
    if(p == null) None else Some((p.name, p.age))
}
```

Quando applicato a un `object` , il modificatore di `case` ha effetti simili (anche se meno drammatici). Qui i guadagni primari sono un'implementazione di `toString` e un valore `hashCode` coerente tra i processi. Nota che gli oggetti case (correttamente) usano l'uguaglianza di riferimento:

```
object Foo extends Product with Serializable {
  def productArity: Int = 0

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productElement(i: Int): Any =
    throw new IndexOutOfBoundsException(i.toString)

  def productPrefix: String = "Foo"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[this.type]

  override def hashCode(): Int = 70822 // "Foo".hashCode()

  override def toString: String = "Foo"
}
```

È ancora possibile implementare manualmente metodi che sarebbero altrimenti forniti dal

modificatore di `case` sia nella classe stessa che nel suo oggetto associato.

Nozioni di base di Case Class

Rispetto alle classi regolari, la notazione delle classi di casi offre diversi vantaggi:

- Tutti gli argomenti del costruttore sono `public` e accessibili su oggetti inizializzati (normalmente questo non è il caso, come dimostrato qui):

```
case class Dog1(age: Int)
val x = Dog1(18)
println(x.age) // 18 (success!)

class Dog2(age: Int)
val x = new Dog2(18)
println(x.age) // Error: "value age is not a member of Dog2"
```

- Fornisce un'implementazione per i seguenti metodi: `toString` , `equals` , `hashCode` (basato sulle proprietà), `copy` , `apply` e `unapply` :

```
case class Dog(age: Int)
val d1 = Dog(10)
val d2 = d1.copy(age = 15)
```

- Fornisce un comodo meccanismo per la corrispondenza del modello:

```
sealed trait Animal // `sealed` modifier allows inheritance within current build-unit
only
case class Dog(age: Int) extends Animal
case class Cat(owner: String) extends Animal
val x: Animal = Dog(18)
x match {
    case Dog(x) => println(s"It's a $x years old dog.")
    case Cat(x) => println(s"This cat belongs to $x.")
}
```

Classi di casi e immutabilità

Il compilatore Scala prefixa ogni argomento nella lista dei parametri per default con `val` . Ciò significa che, per impostazione predefinita, le classi dei casi sono immutabili. Ad ogni parametro viene assegnato un metodo di accesso, ma non esistono metodi di mutatore. Per esempio:

```
case class Foo(i: Int)

val fooInstance = Foo(1)
val j = fooInstance.i       // get
fooInstance.i = 2           // compile-time exception (mutation: reassignment to val)
```

La dichiarazione di un parametro in una case case come `var` sovrascrive il comportamento predefinito e rende la classe case mutabile:

```
case class Bar(var i: Int)

val barInstance = Bar(1)
val j = barInstance.i       // get
barInstance.i = 2           // set
```

Un'altra istanza in cui una classe case è "mutabile" è quando il valore nella classe case è mutabile:

```
import scala.collection._

case class Bar(m: mutable.Map[Int, Int])

val barInstance = Bar(mutable.Map(1 -> 2))
barInstance.m.update(1, 3)                  // mutate m
barInstance                                 // Bar(Map(1 -> 3)
```

Si noti che la "mutazione" che si sta verificando qui è nella mappa a cui `m` indica, non a `m` stesso. Quindi, se qualche altro oggetto avesse `m` come membro, vedrebbe anche il cambiamento. Si noti come nel seguente esempio, la modifica `instanceA` modifica anche l' `instanceB` :

```
import scala.collection.mutable

case class Bar(m: mutable.Map[Int, Int])

val m = mutable.Map(1 ->2)
val barInstanceA = Bar(m)
val barInstanceB = Bar(m)
barInstanceA.m.update(1,3)
barInstanceA  // Bar = Bar(Map(1 -> 3))
barInstanceB  // Bar = Bar(Map(1 -> 3))
m  // scala.collection.mutable.Map[Int,Int] = Map(1 -> 3)
```

Crea una copia di un oggetto con determinate modifiche

Le classi dei casi forniscono un metodo di `copy` che crea un nuovo oggetto che condivide gli stessi campi di quello vecchio, con alcune modifiche.

Possiamo usare questa funzione per creare un nuovo oggetto da uno precedente che abbia alcune delle stesse caratteristiche. Questa semplice classe di casi per dimostrare questa caratteristica:

```
case class Person(firstName: String, lastName: String, grade: String, subject: String)
val putu = Person("Putu", "Kevin", "A1", "Math")
val mark = putu.copy(firstName = "Ketut", lastName = "Mark")
// mark: People = People(Ketut,Mark,A1,Math)
```

In questo esempio possiamo vedere che i due oggetti condividono caratteristiche simili (`grade = A1` , `subject = Math`), tranne dove sono stati specificati nella copia (`firstName` e `lastName`).

Classi di casi a singolo elemento per sicurezza di tipo

Al fine di raggiungere la sicurezza del tipo a volte vogliamo evitare l'uso di tipi primitivi sul nostro dominio. Ad esempio, immagina una `Person` con un `name` . In genere, dovremmo codificare il `name` come `String` . Tuttavia, non sarebbe difficile mescolare uno `String` che rappresenta una `Person` 's `name` con una `String` che rappresenta un messaggio di errore:

```
def logError(message: ErrorMessage): Unit = ???
case class Person(name: String)
val maybeName: Either[String, String] = ??? // Left is error, Right is name
maybeName.foreach(logError) // But that won't stop me from logging the name as an error!
```

Per evitare tali problemi puoi codificare i dati in questo modo:

```
case class PersonName(value: String)
case class ErrorMessage(value: String)
case class Person(name: PersonName)
```

e ora il nostro codice non verrà compilato se mescoliamo `PersonName` con `ErrorMessage` o anche una `String` ordinaria.

```
val maybeName: Either[ErrorMessage, PersonName] = ???
maybeName.foreach(reportError) // ERROR: tried to pass PersonName; ErrorMessage expected
maybeName.swap.foreach(reportError) // OK
```

Ma questo comporta un piccolo sovraccarico di runtime dato che ora dobbiamo box / unbox `String` s to / from i loro contenitori `PersonName` . Per evitare ciò, è possibile creare classi di valore `PersonName` e `ErrorMessage` :

```
case class PersonName(val value: String) extends AnyVal
case class ErrorMessage(val value: String) extends AnyVal
```

Leggi Classi di casi online: https://riptutorial.com/it/scala/topic/1022/classi-di-casi

Capitolo 5: Classi di tipo

Osservazioni

Per evitare problemi di serializzazione, in particolare in ambienti distribuiti (es. Apache Spark), è buona pratica implementare il tratto `Serializable` per le istanze di classe type.

Examples

Classe di tipo semplice

Una classe di tipo è semplicemente un `trait` con uno o più parametri di tipo:

```
trait Show[A] {
  def show(a: A): String
}
```

Invece di estendere una classe di tipo, viene fornita un'istanza implicita della classe type per ogni tipo supportato. Il posizionamento di queste implementazioni nell'oggetto associato della classe del tipo consente di eseguire la risoluzione implicita senza importazioni speciali:

```
object Show {
  implicit val intShow: Show[Int] = new Show {
    def show(x: Int): String = x.toString
  }

  implicit val dateShow: Show[java.util.Date] = new Show {
    def show(x: java.util.Date): String = x.getTime.toString
  }

  // ..etc
}
```

Se si desidera garantire che un parametro generico passato a una funzione abbia un'istanza di una classe di tipo, utilizzare i parametri impliciti:

```
def log[A](a: A)(implicit showInstance: Show[A]): Unit = {
  println(showInstance.show(a))
}
```

Puoi anche utilizzare un contesto :

```
def log[A: Show](a: A): Unit = {
  println(implicitly[Show[A]].show(a))
}
```

Chiama il metodo di `log` sopra come qualsiasi altro metodo. Non riuscirà a compilare se non è possibile trovare un'implementazione implicita `Show[A]` per `A` si passa al `log`

```
log(10) // prints: "10"
log(new java.util.Date(1469491668401L) // prints: "1469491668401"
log(List(1,2,3)) // fails to compile with
                 // could not find implicit value for evidence parameter of type
Show[List[Int]]
```

Questo esempio implementa la classe `Show` type. Questa è una classe di tipo comune utilizzata per convertire istanze arbitrarie di tipi arbitrari in `String` s. Anche se ogni oggetto ha un metodo `toString` , non è sempre chiaro se `toString` sia definito o meno in un modo utile. Con l'uso della classe `Show` type, puoi garantire che tutto ciò che è passato al `log` abbia una conversione ben definita in `String` .

Estendere una classe di tipo

Questo esempio discute l'estensione della classe del tipo sottostante.

```
trait Show[A] {
  def show: String
}
```

Per rendere una classe *che* controlli (ed è scritta in Scala) estendi la classe del tipo, aggiungi un implicito al suo oggetto compagno. Cerchiamo di mostrare come possiamo ottenere la classe `Person` da questo esempio per estendere `Show` :

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}
```

Possiamo rendere questa classe estendere `Show` aggiungendo un implicito all'oggetto companion di `Person` :

```
object Person {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}
```

Un oggetto complementare *deve trovarsi nello stesso file* della classe, quindi è necessario che sia la `class Person` sia la `class Person object Person` nello stesso file.

Per creare una classe che non controlli, o che non sia scritta in Scala, estendi la classe del tipo, aggiungi un implicito all'oggetto complementare della classe type, come mostrato nell'esempio della classe di tipo semplice .

Se non controlli né la classe né la classe del tipo, crea un implicito come sopra ovunque e `import` . Utilizzando il metodo `log` sull'esempio della classe Simple Type :

```
object MyShow {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
```

```
  }
}

def logPeople(persons: Person*): Unit = {
  import MyShow.personShow
  persons foreach { p => log(p) }
}
```

Aggiungi le funzioni della classe tipo ai tipi

L'implementazione di Scala delle classi di tipi è piuttosto dettagliata. Un modo per ridurre la verbosità è introdurre le cosiddette "Classi di operazione". Queste classi comprimono automaticamente una variabile / valore quando vengono importate per estendere la funzionalità.

Per illustrare questo, cerchiamo innanzitutto di creare una classe di tipo semplice:

```
// The mathematical definition of "Addable" is "Semigroup"
trait Addable[A] {
  def add(x: A, y: A): A
}
```

Successivamente implementeremo il tratto (creare un'istanza della classe del tipo):

```
object Instances {

  // Instance for Int
  // Also called evidence object, meaning that this object saw that Int learned how to be
added
  implicit object addableInt extends Addable[Int] {
    def add(x: Int, y: Int): Int = x + y
  }

  // Instance for String
  implicit object addableString extends Addable[String] {
    def add(x: String, y: String): String = x + y
  }

}
```

Al momento sarebbe molto complicato utilizzare le nostre istanze Addable:

```
import Instances._
val three = addableInt.add(1,2)
```

`1.add(2)` semplicemente scrivere write `1.add(2)` . Quindi creeremo una "Operation Class" (anche chiamata "Ops Class") che si occuperà sempre di un tipo che implementa `Addable` .

```
object Ops {
  implicit class AddableOps[A](self: A)(implicit A: Addable[A]) {
    def add(other: A): A = A.add(self, other)
  }
}
```

Ora possiamo usare la nostra nuova funzione `add` come se fosse parte di `Int` e `String` :

```
object Main {

  import Instances._ // import evidence objects into this scope
  import Ops._       // import the wrappers

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

    println(1.add(5))
    println("mag".add("net"))
    // println(1.add(3.141)) // Fails because we didn't create an instance for Double

  }
}
```

Le classi "Ops" possono essere create automaticamente da macro nella libreria simulacrum :

```
import simulacrum._

@typeclass trait Addable[A] {
  @op("|+|") def add(x: A, y: A): A
}
```

Leggi Classi di tipo online: https://riptutorial.com/it/scala/topic/3835/classi-di-tipo

Capitolo 6: Classi e oggetti

Sintassi

- `class MyClass{} // curly braces are optional here as class body is empty`
- `class MyClassWithMethod {def method: MyClass = ???}`
- `new MyClass() //Instantiate`
- `object MyObject // Singleton object`
- `class MyClassWithGenericParameters[V1, V2](vl: V1, i: Int, v2: V2)`
- `class MyClassWithImplicitFieldCreation[V1](val v1: V1, val i: Int)`
- `new MyClassWithGenericParameters(2.3, 4, 5)` o con un tipo diverso: `new MyClassWithGenericParameters[Double, Any](2.3, 4, 5)`
- `class MyClassWithProtectedConstructor protected[my.pack.age](s: String)`

Examples

Istanziare istanze di classe

Una classe in Scala è un "progetto" di un'istanza di classe. Un'istanza contiene lo stato e il comportamento definiti da tale classe. Per dichiarare una classe:

```
class MyClass{}  // curly braces are optional here as class body is empty
```

Un'istanza può essere istanziata usando una `new` parola chiave:

```
var instance = new MyClass()
```

o:

```
var instance = new MyClass
```

Le parentesi sono facoltative in Scala per la creazione di oggetti da una classe che ha un costruttore senza argomenti. Se un costruttore di classi accetta argomenti:

```
class MyClass(arg : Int)       // Class definition
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   // not allowed
```

Qui `MyClass` richiede un argomento `Int` , che può essere usato solo internamente alla classe. non è possibile accedere ad `arg` al di fuori di `MyClass` meno che non sia dichiarato come campo:

```
class MyClass(arg : Int){
    val prop = arg  // Class field declaration
}

var obj = new MyClass(2)
obj.prop     // legal statement
```

In alternativa, può essere dichiarato pubblico nel costruttore:

```
class MyClass(val arg : Int)   // Class definition with arg declared public
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   //arg is now visible to clients
```

Classe di istanziazione senza parametro: {} vs ()

Diciamo che abbiamo una classe MyClass senza argomenti del costruttore:

```
class MyClass
```

In Scala possiamo istanziarlo usando la seguente sintassi:

```
val obj = new MyClass()
```

O possiamo semplicemente scrivere:

```
val obj = new MyClass
```

Ma, se non prestata attenzione, in alcuni casi la parentesi opzionale potrebbe produrre un comportamento inaspettato. Supponiamo di voler creare un'attività che dovrebbe essere eseguita in un thread separato. Di seguito è riportato il codice di esempio:

```
val newThread = new Thread { new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    }

newThread.start   // prints no output
```

Potremmo pensare che questo codice di esempio, se eseguito, stamperà `Performing task.` , ma con nostra sorpresa, non stamperà nulla. Vediamo cosa sta succedendo qui. Se guardi più da vicino, abbiamo usato le parentesi graffe `{}` , subito dopo la `new Thread` . Ha creato una classe anonima che estende `Thread` :

```
val newThread = new Thread {
  //creating anonymous class extending Thread
}
```

E poi nel corpo di questa classe anonima, abbiamo definito il nostro compito (ancora una volta creando una classe anonima che implementa l'interfaccia `Runnable`). Quindi potremmo aver pensato che abbiamo usato il costruttore `public Thread(Runnable target)` ma di fatto (ignorando facoltativo `()`) abbiamo usato `public Thread()` costruttore `public Thread()` con niente definito nel metodo body of `run()` . Per correggere il problema, dobbiamo utilizzare le parentesi anziché le parentesi graffe.

```
val newThread = new Thread ( new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    )
```

In altre parole, qui `{}` e `()` non sono *intercambiabili* .

Singleton & Companion Objects

Oggetti Singleton

Scala supporta membri statici, ma non nello stesso modo di Java. Scala fornisce un'alternativa a questo chiamato *Singleton Objects* . Gli oggetti Singleton sono simili a una classe normale, tranne per il fatto che non possono essere istanziati utilizzando la `new` parola chiave. Di seguito è riportata una classe di esempio singleton:

```
object Factorial {
    private val cache = Map[Int, Int]()
    def getCache = cache
}
```

Si noti che abbiamo usato la parola chiave `object` per definire l'oggetto singleton (invece di 'class' o 'trait'). Poiché gli oggetti singleton non possono essere istanziati, non possono avere parametri. L'accesso a un oggetto singleton ha il seguente aspetto:

```
Factorial.getCache() //returns the cache
```

Si noti che questo sembra esattamente come accedere a un metodo statico in una classe Java.

Companion Objects

In oggetti Singleton Scala può condividere il nome di una classe corrispondente. In uno scenario di questo tipo, l'oggetto singleton viene definito come un *oggetto companion* . Ad esempio, sotto la classe `Factorial` è definito, e sotto di esso è definito un oggetto companion (chiamato anche `Factorial`). Per convenzione gli oggetti companion sono definiti nello stesso file della loro classe companion.

```
class Factorial(num : Int) {

  def fact(num : Int) : Int = if (num <= 1) 1 else (num * fact(num - 1))

  def calculate() : Int = {
    if (!Factorial.cache.contains(num)) {    // num does not exists in cache
      val output = fact(num) // calculate factorial
      Factorial.cache += (num -> output)     // add new value in cache
    }
```

```
    Factorial.cache(num)
  }
}

object Factorial {
  private val cache = scala.collection.mutable.Map[Int, Int]()
}

val factfive = new Factorial(5)
factfive.calculate  // Calculates the factorial of 5 and stores it
factfive.calculate  // uses cache this time
val factfiveagain = new Factorial(5)
factfiveagain.calculate  // Also uses cache
```

In questo esempio utilizziamo una `cache` privata per memorizzare fattoriale di un numero per risparmiare tempo di calcolo per numeri ripetuti.

Qui `object Factorial` è un oggetto compagno e `class Factorial` è la corrispondente classe compagno. Gli oggetti e le classi Companion possono accedere ai rispettivi membri `private` . Nell'esempio sopra la classe `Factorial` sta accedendo al membro della `cache` privata dell'oggetto associato.

Si noti che una nuova istanziazione della classe continuerà a utilizzare lo stesso oggetto compagno, quindi qualsiasi modifica alle variabili membro di tale oggetto verrà trasferita.

Oggetti

Mentre le classi sono più simili a progetti, gli oggetti sono statici (cioè già istanziati):

```
object Dog {
    def bark: String = "Raf"
}

Dog.bark() // yields "Raf"
```

Sono spesso usati come compagni di una classe, ti permettono di scrivere:

```
class Dog(val name: String) {

}

object Dog {
    def apply(name: String): Dog = new Dog(name)
}

val dog = Dog("Barky") // Object
val dog = new Dog("Barky") // Class
```

Controllo del tipo di istanza

Verifica del tipo : `variable.isInstanceOf[Type]`

Con la corrispondenza del modello (non così utile in questa forma):

```
variable match {
  case _: Type => true
  case _ => false
}
```

Sia `isInstanceOf` che pattern matching stanno controllando solo il tipo dell'oggetto, non il suo parametro generico (nessuna reificazione del tipo), ad eccezione degli array:

```
val list: List[Any] = List(1, 2, 3)             //> list  : List[Any] = List(1, 2, 3)

val upcasting = list.isInstanceOf[Seq[Int]]     //> upcasting  : Boolean = true

val shouldBeFalse = list.isInstanceOf[List[String]]
                                                //> shouldBeFalse  : Boolean = true
```

Ma

```
val chSeqArray: Array[CharSequence] = Array("a") //> chSeqArray  : Array[CharSequence] =
Array(a)
val correctlyReified = chSeqArray.isInstanceOf[Array[String]]
                                               //> correctlyReified  : Boolean = false


val stringIsACharSequence: CharSequence = ""    //> stringIsACharSequence  : CharSequence = ""

val sArray = Array("a")                         //> sArray  : Array[String] = Array(a)
val correctlyReified = sArray.isInstanceOf[Array[String]]
                                                //> correctlyReified  : Boolean = true

//val arraysAreInvariantInScala: Array[CharSequence] = sArray
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
//Note: String <: CharSequence, but class Array is invariant in type T.
//You may wish to investigate a wildcard type such as `_ <: CharSequence`. (SLS 3.2.10)
//Workaround:
val arraysAreInvariantInScala: Array[_ <: CharSequence] = sArray
                                                //> arraysAreInvariantInScala  : Array[_ <:
CharSequence] = Array(a)


val arraysAreCovariantOnJVM = sArray.isInstanceOf[Array[CharSequence]]
                                                //> arraysAreCovariantOnJVM  : Boolean = true
```

Digitare casting : `variable.asInstanceOf[Type]`

Con la corrispondenza del modello :

```
variable match {
  case _: Type => true
}
```

Esempi:

```
  val x = 3                                       //> x  : Int = 3
  x match {
    case _: Int => true//better: do something
```

```
    case _ => false
  }                                             //> res0: Boolean = true

  x match {
    case _: java.lang.Integer => true//better: do something
    case _ => false
  }                                             //> res1: Boolean = true

  x.isInstanceOf[Int]                           //> res2: Boolean = true

  //x.isInstanceOf[java.lang.Integer]//fruitless type test: a value of type Int cannot also be
a Integer

  trait Valuable { def value: Int}
  case class V(val value: Int) extends Valuable

  val y: Valuable = V(3)                        //> y  : Valuable = V(3)
  y.isInstanceOf[V]                             //> res3: Boolean = true
  y.asInstanceOf[V]                             //> res4: V = V(3)
```

Nota: si tratta solo del comportamento sulla JVM, su altre piattaforme (JS, native) il tipo di casting / controllo potrebbe comportarsi diversamente.

Costruttori

Costruttore primario

In Scala il costruttore principale è il corpo della classe. Il nome della classe è seguito da un elenco di parametri, che sono gli argomenti del costruttore. (Come con qualsiasi funzione, un elenco di parametri vuoto può essere omesso.)

```
class Foo(x: Int, y: String) {
    val xy: String = y * x
    /* now xy is a public member of the class */
}

class Bar {
    ...
}
```

I parametri di costruzione di un'istanza non sono accessibili al di fuori del suo corpo del costruttore a meno che non siano contrassegnati come membro di istanza dalla parola chiave `val` :

```
class Baz(val z: String)
// Baz has no other members or methods, so the body may be omitted

val foo = new Foo(4, "ab")
val baz = new Baz("I am a baz")
foo.x // will not compile: x is not a member of Foo
foo.xy // returns "abababab": xy is a member of Foo
baz.z // returns "I am a baz": z is a member of Baz
val bar0 = new Bar
val bar1 = new Bar() // Constructor parentheses are optional here
```

Qualsiasi operazione da eseguire quando un'istanza di un oggetto viene creata un'istanza viene scritta direttamente nel corpo della classe:

```
class DatabaseConnection
    (host: String, port: Int, username: String, password: String) {
    /* first connect to the DB, or throw an exception */
    private val driver = new AwesomeDB.Driver()
    driver.connect(host, port, username, password)
    def isConnected: Boolean = driver.isConnected
    ...
}
```

Si noti che è considerata una buona pratica inserire il minor numero possibile di effetti collaterali nel costruttore; al posto del codice precedente, si dovrebbe considerare di `connect` e `disconnect` metodi in modo che il codice del consumatore sia responsabile della pianificazione dell'IO.

Costruttori ausiliari

Una classe può avere costruttori aggiuntivi chiamati "costruttori ausiliari". Questi sono definiti dalle definizioni del costruttore nel formato `def this(...) = e` , dove `e` deve richiamare un altro costruttore:

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

// usage:
new Person("Grace Hopper").fullName // returns Grace Hopper
new Person("Grace", "Hopper").fullName // returns Grace Hopper
```

Ciò implica che ogni costruttore può avere un modificatore diverso: solo alcuni possono essere disponibili pubblicamente:

```
class Person private(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

new Person("Ada Lovelace") // won't compile
new Person("Ada", "Lovelace") // compiles
```

In questo modo puoi controllare in che modo il codice del consumatore può istanziare la classe.

Leggi Classi e oggetti online: https://riptutorial.com/it/scala/topic/2047/classi-e-oggetti

Capitolo 7: collezioni

Examples

Ordina una lista

Supponendo che il seguente elenco possiamo ordinare una varietà di modi.

```
val names = List("Kathryn", "Allie", "Beth", "Serin", "Alana")
```

Il comportamento predefinito di `sorted()` consiste nell'utilizzare `math.Ordering` , che per le stringhe restituisce un ordinamento lessografico :

```
names.sorted
// results in: List(Alana, Allie, Beth, Kathryn, Serin)
```

`sortWith` consente di fornire il proprio ordine utilizzando una funzione di confronto:

```
names.sortWith(_.length < _.length)
// results in: List(Beth, Allie, Serin, Alana, Kathryn)
```

`sortBy` ti consente di fornire una funzione di trasformazione:

```
//A set of vowels to use
val vowels = Set('a', 'e', 'i', 'o', 'u')

//A function that counts the vowels in a name
def countVowels(name: String) = name.count(l => vowels.contains(l.toLower))

//Sorts by the number of vowels
names.sortBy(countVowels)
//result is: List(Kathryn, Beth, Serin, Allie, Alana)
```

Puoi sempre invertire una lista, o una lista ordinata, usando `reverse:

```
names.sorted.reverse
//results in: List(Serin, Kathryn, Beth, Allie, Alana)
```

Gli elenchi possono anche essere ordinati utilizzando il metodo Java `java.util.Arrays.sort` e il suo wrapper Scala `scala.util.Sorting.quickSort`

```
java.util.Arrays.sort(data)
scala.util.Sorting.quickSort(data)
```

Questi metodi possono migliorare le prestazioni quando si ordinano raccolte più grandi, se è possibile evitare le conversioni di raccolta e di disimballaggio / boxing. Per una discussione più dettagliata sulle differenze di prestazioni, leggi su Scala Collection ordinata, sortWith e sortBy

Performance .

Crea una lista contenente n copie di x

Per creare una raccolta di `n` copie di qualche oggetto `x` , utilizzare il metodo di riempimento . Questo esempio crea un `List` , ma può funzionare con altre raccolte per le quali il `fill` ha senso:

```
// List.fill(n)(x)
scala > List.fill(3)("Hello World")
res0: List[String] = List(Hello World, Hello World, Hello World)
```

Lista e trucchi di vettore

> Ora è preferibile utilizzare `Vector` anziché `List` perché le implementazioni hanno prestazioni migliori. Le caratteristiche di prestazione possono essere trovate qui . `Vector` può essere utilizzato ovunque sia utilizzato `List` .

Creazione di liste

```
List[Int]()          // Declares an empty list of type Int
List.empty[Int]      // Uses `empty` method to declare empty list of type Int
Nil                  // A list of type Nothing that explicitly has nothing in it

List(1, 2, 3)        // Declare a list with some elements
1 :: 2 :: 3 :: Nil   // Chaining element prepending to an empty list, in a LISP-style
```

Prendi l'elemento

```
List(1, 2, 3).headOption // Some(1)
List(1, 2, 3).head       // 1

List(1, 2, 3).lastOption // Some(3)
List(1, 2, 3).last       // 3, complexity is O(n)

List(1, 2, 3)(1)         // 2, complexity is O(n)
List(1, 2, 3)(3)         // java.lang.IndexOutOfBoundsException: 4
```

Prepend gli elementi

```
0 :: List(1, 2, 3)       // List(0, 1, 2, 3)
```

Aggiungi elementi

```
List(1, 2, 3) :+ 4       // List(1, 2, 3, 4), complexity is O(n)
```

Iscriviti (Concatena) Elenchi

```
List(1, 2) ::: List(3, 4) // List(1, 2, 3, 4)
List.concat(List(1,2), List(3, 4)) // List(1, 2, 3, 4)
List(1, 2) ++ List(3, 4)  // List(1, 2, 3, 4)
```

Operazioni comuni

```
List(1, 2, 3).find(_ == 3)                     // Some(3)
List(1, 2, 3).map(_ * 2)                       // List(2, 4, 6)
List(1, 2, 3).filter(_ % 2 == 1)               // List(1, 3)
List(1, 2, 3).fold(0)((acc, i) => acc + i * i) // 1 * 1 + 2 * 2 + 3 * 3 = 14
List(1, 2, 3).foldLeft("Foo")(_ + _.toString)  // "Foo123"
List(1, 2, 3).foldRight("Foo")(_ + _.toString) // "123Foo"
```

Tagliere di raccolta mappe

> Nota che questo riguarda la creazione di una raccolta di tipo `Map` , che è distinta dal metodo della `map` .

Creazione della mappa

```
Map[String, Int]()
val m1: Map[String, Int] = Map()
val m2: String Map Int = Map()
```

Una mappa può essere considerata una raccolta di `tuples` per la maggior parte delle operazioni, in cui il primo elemento è la chiave e il secondo è il valore.

```
val l = List(("a", 1), ("b", 2), ("c", 3))
val m = l.toMap                               // Map(a -> 1, b -> 2, c -> 3)
```

Ottieni elemento

```
val m = Map("a" -> 1, "b" -> 2, "c" -> 3)

m.get("a")  // Some(1)
m.get("d")  // None
m("a")      // 1
m("d")      // java.util.NoSuchElementException: key not found: d

m.keys      // Set(a, b, c)
m.values    // MapLike(1, 2, 3)
```

Aggiungi elemento (i)

```
Map("a" -> 1, "b" -> 2) + ("c" -> 3)              // Map(a -> 1, b -> 2, c -> 3)
Map("a" -> 1, "b" -> 2) + ("a" -> 3)              // Map(a -> 3, b -> 2)
Map("a" -> 1, "b" -> 2) ++ Map("b" -> 3, "c" -> 4) // Map(a -> 1, b -> 3, c -> 4)
```

Operazioni comuni

Nelle operazioni in cui si verifica un'iterazione su una mappa (`map` , `find` , `forEach` , ecc.), Gli elementi della raccolta sono `tuples` . Il parametro della funzione può utilizzare gli accessor tuple (`_1` , `_2`) o una funzione parziale con un blocco di casi:

```
m.find(_._1 == "a")  // Some((a,1))
```

```
m.map {
  case (key, value) => (value, key)
}                    // Map(1 -> a, 2 -> b, 3 -> c)
m.filter(_._2 == 2)  // Map(b -> 2)
m.foldLeft(0){
  case (acc, (key, value: Int)) => acc + value
}                    // 6
```

Mappa e filtra su una collezione

Carta geografica

"Mappatura" su una raccolta utilizza la funzione `map` per trasformare ogni elemento di quella raccolta in un modo simile. La sintassi generale è:

```
val someFunction: (A) => (B) = ???
collection.map(someFunction)
```

Puoi fornire una funzione anonima:

```
collection.map((x: T) => /*Do something with x*/)
```

Moltiplicare i numeri interi per due

```
// Initialize
val list = List(1,2,3)
// list: List[Int] = List(1, 2, 3)

// Apply map
list.map((item: Int) => item*2)
// res0: List[Int] = List(2, 4, 6)

// Or in a more concise way
list.map(_*2)
// res1: List[Int] = List(2, 4, 6)
```

Filtro

`filter` viene utilizzato quando si desidera escludere o "filtrare" determinati elementi di una raccolta. Come con la `map` , la sintassi generale assume una funzione, ma quella funzione deve restituire un `Boolean` :

```
val someFunction: (a) => Boolean = ???
collection.filter(someFunction)
```

Puoi fornire direttamente una funzione anonima:

```
collection.filter((x: T) => /*Do something that returns a Boolean*/)
```

Controllo dei numeri di coppia

```
val list = 1 to 10 toList
// list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Filter out all elements that aren't evenly divisible by 2
list.filter((item: Int) => item % 2==0)
// res0: List[Int] = List(2, 4, 6, 8, 10)
```

Altri esempi di mappe e filtri

```
case class Person(firstName: String,
                  lastName: String,
                  title: String)

// Make a sequence of people
val people = Seq(
  Person("Millie", "Fletcher", "Mrs"),
  Person("Jim", "White", "Mr"),
  Person("Jenny", "Ball", "Miss") )


// Make labels using map
val labels = people.map( person =>
  s"${person.title}. ${person.lastName}"
)

// Filter the elements beginning with J
val beginningWithJ = people.filter(_.firstName.startsWith("J"))

// Extract first names and concatenate to a string
val firstNames = people.map(_.firstName).reduce( (a, b) => a + "," + b )
```

Introduzione alle Collezioni Scala

Il framework delle collezioni Scala, secondo i suoi autori , è progettato per essere facile da usare, conciso, sicuro, veloce e universale.

Il framework è costituito da tratti Scala che sono progettati per essere blocchi per la creazione di collezioni. Per maggiori informazioni su questi blocchi, leggi la panoramica ufficiale delle collezioni Scala .

Queste raccolte incorporate sono separate nei pacchetti immutabili e mutabili. Per impostazione predefinita, vengono utilizzate le versioni immutabili. Costruire un `List()` (senza importare nulla) costruirà una lista *immutabile* .

Una delle funzionalità più potenti del framework è l'interfaccia coerente e di facile utilizzo tra le raccolte affini. Ad esempio, sommando tutti gli elementi in una raccolta è uguale per Elenchi, Set, Vettori, Seq e Matrici:

```
val numList = List[Int](1, 2, 3, 4, 5)
numList.reduce((n1, n2) => n1 + n2)  // 15

val numSet = Set[Int](1, 2, 3, 4, 5)
numSet.reduce((n1, n2) => n1 + n2)   // 15

val numArray = Array[Int](1, 2, 3, 4, 5)
numArray.reduce((n1, n2) => n1 + n2) // 15
```

Questi tipi affini ereditano dal tratto `Traversable` .

> Ora è preferibile utilizzare `Vector` anziché `List` perché le implementazioni hanno prestazioni migliori. Le caratteristiche di prestazione possono essere trovate qui . `Vector` può essere utilizzato ovunque sia utilizzato `List` .

Tipi percorribili

Classi di raccolta che hanno l'attrezzo `Traversable` trait `foreach` ed ereditano molti metodi per eseguire operazioni comuni alle collezioni, che funzionano tutte in modo identico. Le operazioni più comuni sono elencate qui:

- Mappa : `map` , `flatMap` e `collect` nuove collezioni applicando una funzione a ciascun elemento della raccolta originale.

```
List(1, 2, 3).map(num => num * 2) // double every number = List(2, 4, 6)

// split list of letters into individual strings and put them into the same list
List("a b c", "d e").flatMap(letters => letters.split(" ")) // = List("a", "b", "c", "d", "e")
```

- Le conversioni - `toList` , `toArray` e molte altre operazioni di conversione cambiano la raccolta corrente in un tipo più specifico di raccolta. Questi sono in genere metodi preceduti da 'to' e il tipo più specifico (cioè 'tolist' converte in una `List`).

```
val array: Array[Int] = List[Int](1, 2, 3).toArray // convert list of ints to array of ints
```

- Le informazioni sulla dimensione - `isEmpty` , `nonEmpty` , `size` e `hasDefiniteSize` sono tutti metadati relativi al set. Ciò consente alle operazioni condizionali sulla raccolta o al codice di determinare la dimensione della raccolta, incluso se è infinita o discreta.

```
List().isEmpty // true
List(1).nonEmpty // true
```

- Recupero degli elementi : `head` , `last` , `find` e le loro varianti di `Option` vengono utilizzate per recuperare il primo o l'ultimo elemento o trovare un elemento specifico nella raccolta.

```
val list = List(1, 2, 3)
list.head // = 1
list.last // = 3
```

- Le operazioni di recupero della sub-raccolta - `filter` , `tail` , `slice` , `drop` e altre operazioni

consentono di scegliere parti della raccolta per operare ulteriormente.

```
List(-2, -1, 0, 1, 2).filter(num => num > 0) // = List(1, 2)
```

- Operazioni di suddivisione : `partition` , `splitAt` , `span` e `groupBy` dividono la raccolta corrente in parti diverse.

```
// split numbers into < 0 and >= 0
List(-2, -1, 0, 1, 2).partition(num => num < 0) // = (List(-2, -1), List(0, 1, 2))
```

- Test Element - `exists` , `forall` e `count` vengono usati per controllare le operazioni di questa raccolta per vedere se soddisfa un predicato.

```
List(1, 2, 3, 4).forall(num => num > 0) // = true, all numbers are positive
List(-3, -2, -1, 1).forall(num => num < 0) // = false, not all numbers are negative
```

- Folds - `foldLeft` (`/:` `foldRight` , `foldRight` (`:\`), `reduceLeft` e `reduceRight` vengono utilizzati per applicare le funzioni binarie agli elementi successivi nella raccolta. Vai qui per gli esempi di piega e vai qui per ridurre gli esempi .

piega

Il metodo `fold` esegue iterazioni su una raccolta, utilizzando un valore dell'accumulatore iniziale e applicando una funzione che utilizza ciascun elemento per aggiornare correttamente l'accumulatore:

```
val nums = List(1,2,3,4,5)
var initialValue:Int = 0;
var sum = nums.fold(initialValue){
  (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 15 because 0+1+2+3+4+5 = 15
```

Nell'esempio precedente, è stata fornita una funzione anonima a `fold()` . Puoi anche usare una funzione con nome che accetta due argomenti. Sopportando questo nel mio, l'esempio sopra può essere riscritto così:

```
def sum(x: Int, y: Int) = x+ y
val nums = List(1, 2, 3, 4, 5)
var initialValue: Int = 0
val sum = nums.fold(initialValue)(sum)
println(sum) // prints 15 because 0 + 1 + 2 + 3 + 4 + 5 = 15
```

La modifica del valore iniziale influirà sul risultato:

```
initialValue = 2;
sum = nums.fold(initialValue){
 (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 17 because 2+1+2+3+4+5 = 17
```

Il metodo `fold` ha due varianti: `foldLeft` e `foldRight` .

`foldLeft()` itera da sinistra a destra (dal primo elemento della raccolta fino all'ultimo in quell'ordine). `foldRight()` da destra a sinistra (dall'ultimo elemento al primo elemento). `fold()` scorre da sinistra a destra come `foldLeft()` . Infatti `fold()` chiama effettivamente `foldLeft()` internamente.

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
```

`fold()` , `foldLeft()` e `foldRight()` restituiranno un valore che ha lo stesso tipo con il valore iniziale che prende. Tuttavia, a differenza di `foldLeft()` e `foldRight()` , il valore iniziale assegnato a `fold()` può essere solo dello stesso tipo o di un supertipo del tipo della raccolta.

In questo esempio l'ordine non è rilevante, quindi puoi cambiare `fold()` in `foldLeft()` o `foldRight()` e il risultato rimarrà lo stesso. L'utilizzo di una funzione sensibile all'ordine altererà i risultati.

In caso di dubbio, preferisci `foldLeft()` su `foldRight()` . `foldRight()` è meno performante.

Per ciascuno

`foreach` è insolito tra gli iteratori delle collezioni in quanto non restituisce un risultato. Invece applica una funzione a ciascun elemento che ha solo effetti collaterali. Per esempio:

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x.foreach { println }
1
2
3
```

La funzione fornita a `foreach` può avere qualsiasi tipo di ritorno, ma il risultato verrà scartato . In genere `foreach` viene utilizzato quando sono desiderati effetti collaterali. Se si desidera trasformare i dati, prendere in considerazione l'utilizzo di `map` , `filter` , a `for comprehension` o un'altra opzione.

Esempio di risultati di scarto

```
def myFunc(a: Int) : Int = a * 2
List(1,2,3).foreach(myFunc) // Returns nothing
```

Ridurre

I metodi `reduceRight` `reduce()` , `reduceLeft()` e `reduceRight` sono simili alle pieghe. La funzione passata per ridurre prende due valori e ne produce un terzo. Quando si opera in un elenco, i primi due valori sono i primi due valori nell'elenco. Il risultato della funzione e il successivo valore nell'elenco vengono quindi riapplicati alla funzione, producendo un nuovo risultato. Questo nuovo risultato viene applicato con il successivo valore dell'elenco e così via fino a quando non ci sono più elementi. Il risultato finale viene restituito.

```
val nums = List(1,2,3,4,5)
sum = nums.reduce({ (a, b) => a + b })
println(sum) //prints 15

val names = List("John","Koby", "Josh", "Matilda", "Zac", "Mary Poppins")

def findLongest(nameA:String, nameB:String):String = {
  if (nameA.length > nameB.length) nameA else nameB
}

def findLastAlphabetically(nameA:String, nameB:String):String = {
  if (nameA > nameB) nameA else nameB
}

val longestName:String = names.reduce(findLongest(_,_))
println(longestName) //prints Mary Poppins

//You can also omit the arguments if you want
val lastAlphabetically:String = names.reduce(findLastAlphabetically)
println(lastAlphabetically) //prints Zac
```

Ci sono alcune differenze nel modo in cui le funzioni di riduzione funzionano rispetto alle funzioni di piega. Loro sono:

1. Le funzioni di riduzione non hanno valore dell'accumulatore iniziale.
2. Le funzioni di riduzione non possono essere chiamate su liste vuote.
3. Le funzioni di riduzione possono solo restituire il tipo o il supertipo dell'elenco.

Leggi collezioni online: https://riptutorial.com/it/scala/topic/686/collezioni

Capitolo 8: Collezioni parallele

Osservazioni

Le raccolte parallele facilitano la programmazione parallela nascondendo i dettagli di parallelizzazione di basso livello. Ciò facilita l'utilizzo di architetture multi-core. Esempi di collezioni parallele includono `ParArray` , `ParVector` , `mutable.ParHashMap` , `immutable.ParHashMap` e `ParRange` . Un elenco completo può essere trovato nella documentazione .

Examples

Creazione e utilizzo di raccolte parallele

Per creare una raccolta parallela da una collezione sequenziale, chiama il metodo `par` . Per creare una collezione sequenziale da una raccolta parallela, chiamare il metodo `seq` . Questo esempio mostra come si trasforma un `Vector` normale in un `ParVector` e poi di nuovo:

```
scala> val vect = (1 to 5).toVector
vect: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> val parVect = vect.par
parVect: scala.collection.parallel.immutable.ParVector[Int] = ParVector(1, 2, 3, 4, 5)

scala> parVect.seq
res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
```

Il metodo `par` può essere concatenato, consentendo di convertire una collezione sequenziale in una raccolta parallela e eseguire immediatamente un'azione su di essa:

```
scala> vect.map(_ * 2)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 4, 6, 8, 10)

scala> vect.par.map(_ * 2)
res2: scala.collection.parallel.immutable.ParVector[Int] = ParVector(2, 4, 6, 8, 10)
```

In questi esempi, il lavoro viene effettivamente suddiviso in più unità di elaborazione e quindi ricongiunto dopo il completamento del lavoro, senza richiedere l'intervento dello sviluppatore.

insidie

Non utilizzare raccolte parallele quando gli elementi della raccolta devono essere ricevuti in un ordine specifico.

Le raccolte parallele eseguono operazioni contemporaneamente. Ciò significa che tutto il lavoro è diviso in parti e distribuito a diversi processori. Ogni processore non è consapevole del lavoro svolto da altri. Se l' *ordine della raccolta* è importante, il lavoro elaborato in parallelo non è deterministico. (L'esecuzione dello stesso codice due volte può produrre risultati diversi).

Operazioni non associative

Se un'operazione è non associativa (se l'ordine di esecuzione è importante), il risultato su una raccolta parallela sarà non deterministico.

```
scala> val list = (1 to 1000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10...

scala> list.reduce(_ - _)
res0: Int = -500498

scala> list.reduce(_ - _)
res1: Int = -500498

scala> list.reduce(_ - _)
res2: Int = -500498

scala> val listPar = list.par
listPar: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8,
9, 10...

scala> listPar.reduce(_ - _)
res3: Int = -408314

scala> listPar.reduce(_ - _)
res4: Int = -422884

scala> listPar.reduce(_ - _)
res5: Int = -301748
```

Effetti collaterali

Le operazioni che hanno effetti collaterali, come `foreach` , potrebbero non essere eseguite come desiderato su raccolte parallele a causa delle condizioni di gara. Evita questo utilizzando funzioni che non hanno effetti collaterali, come `reduce` o `map` .

```
scala> val wittyOneLiner = Array("Artificial", "Intelligence", "is", "no", "match", "for",
"natural", "stupidity")

scala> wittyOneLiner.foreach(word => print(word + " "))
Artificial Intelligence is no match for natural stupidity

scala> wittyOneLiner.par.foreach(word => print(word + " "))
match natural is for Artificial no stupidity Intelligence

scala> print(wittyOneLiner.par.reduce{_ + " " + _})
Artificial Intelligence is no match for natural stupidity

scala> val list = (1 to 100).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...
```

Leggi Collezioni parallele online: https://riptutorial.com/it/scala/topic/3882/collezioni-parallele

Capitolo 9: Combinatori parser

Osservazioni

Casi di ParseResult

A `ParseResult` disponibile in tre versioni:

- Successo, con un marcatore per l'inizio della partita e il prossimo personaggio da abbinare.
- Fallimento, con un indicatore dell'avvio di dove è stata tentata la partita. In questo caso il parser retrocede in quella posizione, dove sarà quando l'analisi continua.
- Errore, che interrompe l'analisi. Nessun backtracking o ulteriore parsing.

Examples

Esempio di base

```
import scala.util.parsing.combinator._

class SimpleParser extends RegexParsers {
  // Define a grammar rule, turn it into a regex, and apply it the input.
  def word: Parser[String] = """[A-Z][a-z]+""".r ^^ { _.toString }
}

object SimpleParser extends SimpleParser {
  val parseAlice = parse(word, "Alice went to Alamo Square.")
  val parseBarb = parse(word, "barb went Upside Down.")
}

//Successfully finds a match
println(SimpleParser.parseAlice)
//Fails to find a match
println(SimpleParser.parseBarb)
```

L'output sarà il seguente:

```
[1.6] parsed: Alice
res0: Unit = ()

[1.1] failure: string matching regex `[A-Z][a-z]+' expected but `b' found

barb went Upside Down.
^
```

`[1.6]` nell'esempio di `Alice` indica che l'inizio della partita è in posizione `1` e il carattere di pugno che rimane per abbinare inizia nella posizione `6` .

Leggi Combinatori parser online: https://riptutorial.com/it/scala/topic/3730/combinatori-parser

Capitolo 10: Configurare Scala

Examples

Su Linux tramite dpkg

Nelle distribuzioni basate su Debian, incluso Ubuntu, il modo più semplice è usare il file di installazione `.deb` . Vai al sito web di Scala . Scegli la versione che desideri installare, scorri verso il basso e cerca `scala-xxxdeb` .

È possibile installare la scala deb dalla riga di comando:

```
sudo dpkg -i scala-x.x.x.deb
```

Per verificare che sia installato correttamente, nel prompt dei comandi del terminale:

```
which scala
```

La risposta restituita dovrebbe essere equivalente a quanto inserito nella variabile PATH. Per verificare che scala funzioni:

```
scala
```

Questo dovrebbe avviare il REPL di Scala e segnalare la versione (che, a sua volta, dovrebbe corrispondere alla versione scaricata).

Installazione di Ubuntu tramite download e configurazione manuale

Scarica la tua versione preferita da Lightbend con `curl` :

```
curl -O http://downloads.lightbend.com/scala/2.xx.x/scala-2.xx.x.tgz
```

Decomprimere il file `tar` su `/usr/local/share` o `/opt/bin` :

```
unzip scala-2.xx.x.tgz
mv scala-2.xx.x /usr/local/share/scala
```

Aggiungi il `PATH` a `~/.profile` o `~/.bash_profile` o `~/.bashrc` includendo questo testo in uno di questi file:

```
$SCALA_HOME=/usr/local/share/scala
export PATH=$SCALA_HOME/bin:$PATH
```

Per verificare che sia installato correttamente, nel prompt dei comandi del terminale:

```
which scala
```

La risposta restituita dovrebbe essere equivalente a quanto inserito nella variabile `PATH` . Per verificare che `scala` :

```
scala
```

Questo dovrebbe avviare il REPL di Scala e segnalare la versione (che, a sua volta, dovrebbe corrispondere alla versione scaricata).

Mac OSX tramite Macports

Su computer Mac OSX con MacPorts installati, apri una finestra di terminale e digita:

```
port list | grep scala
```

Questo elencherà tutti i pacchetti relativi a Scala disponibili. Per installarne uno (in questo esempio la versione 2.11 di Scala):

```
sudo port install scala2.11
```

(Il `2.11` potrebbe cambiare se si desidera installare una versione diversa.)

Tutte le dipendenze verranno installate automaticamente e il parametro `$PATH` aggiornato. Per verificare tutto ha funzionato:

```
which scala
```

Questo ti mostrerà il percorso per l'installazione di Scala.

```
scala
```

Questo aprirà Scala REPL e segnalerà il numero di versione installato.

Leggi Configurare Scala online: https://riptutorial.com/it/scala/topic/2921/configurare-scala

Capitolo 11: enumerazioni

Osservazioni

L'approccio con `sealed trait` e `case objects` è preferito perché l'enumerazione di Scala ha alcuni problemi:

1. Le enumerazioni hanno lo stesso tipo dopo la cancellazione.
2. Il compilatore non si lamenta del fatto che "Match non è esaustivo", se il caso è mancato fallirà in runtime `scala.MatchError` :

```
def isWeekendWithBug(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sun | WeekDays.Sat => true
}

isWeekendWithBug(WeekDays.Fri)
scala.MatchError: Fri (of class scala.Enumeration$Val)
```

Confrontare con:

```
def isWeekendWithBug(day: WeekDay): Boolean = day match {
  case WeekDay.Sun | WeekDay.Sat => true
}

Warning: match may not be exhaustive.
It would fail on the following inputs: Fri, Mon, Thu, Tue, Wed
def isWeekendWithBug(day: WeekDay): Boolean = day match {
                                              ^
```

Una spiegazione più dettagliata è presentata in questo articolo su Enumerazione di Scala .

Examples

Giorni della settimana usando Enumerazione Scala

Le enumerazioni Java-like possono essere create estendendo Enumeration .

```
object WeekDays extends Enumeration {
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

def isWeekend(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sat | WeekDays.Sun => true
  case _ => false
}

isWeekend(WeekDays.Sun)
res0: Boolean = true
```

È anche possibile aggiungere un nome leggibile dall'uomo per i valori in un'enumerazione:

```
object WeekDays extends Enumeration {
      val Mon = Value("Monday")
      val Tue = Value("Tuesday")
      val Wed = Value("Wednesday")
      val Thu = Value("Thursday")
      val Fri = Value("Friday")
      val Sat = Value("Saturday")
      val Sun = Value("Sunday")
}

println(WeekDays.Mon)
>> Monday

WeekDays.withName("Monday") == WeekDays.Mon
>> res0: Boolean = true
```

Fai attenzione al comportamento non tipicamente errato, in cui diverse enumerazioni possono valutare come lo stesso tipo di istanza:

```
object Parity extends Enumeration {
   val Even, Odd = Value
}

WeekDays.Mon.isInstanceOf[Parity.Value]
>> res1: Boolean = true
```

Utilizzo di tratti sigillati e oggetti del caso

Un'alternativa all'estensione `Enumeration` sta utilizzando oggetti caso `sealed` :

```
sealed trait WeekDay

object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
}
```

La parola chiave `sealed` garantisce che il tratto `WeekDay` non possa essere esteso in un altro file. Ciò consente al compilatore di fare alcune ipotesi, incluso che tutti i valori possibili di `WeekDay` sono già elencati.

Uno svantaggio è che questo metodo non consente di ottenere un elenco di tutti i valori possibili. Per ottenere una tale lista deve essere fornita esplicitamente:

```
val allWeekDays = Seq(Mon, Tue, Wed, Thu, Fri, Sun, Sat)
```

Le classi di casi possono anche estendere un tratto `sealed` . Pertanto, oggetti e case case possono essere mescolati per creare gerarchie complesse:

```
sealed trait CelestialBody

object CelestialBody {
  case object Earth extends CelestialBody
  case object Sun extends CelestialBody
  case object Moon extends CelestialBody
  case class Asteroid(name: String) extends CelestialBody
}
```

Un altro svantaggio è che non c'è modo di accedere a un nome di variabile dell'enumerazione di un oggetto `sealed` , o cercare da esso. Se è necessario un tipo di nome associato a ciascun valore, deve essere definito manualmente:

```
  sealed trait WeekDay { val name: String }

  object WeekDay {
      case object Mon extends WeekDay { val name = "Monday" }
      case object Tue extends WeekDay { val name = "Tuesday" }
      (...)
  }
```

O semplicemente:

```
  sealed case class WeekDay(name: String)

  object WeekDay {
      object Mon extends WeekDay("Monday")
      object Tue extends WeekDay("Tuesday")
      (...)
  }
```

Utilizzo di tratti tratti e casi e di tutti i valori-macro

Questa è solo un'estensione della variante del tratto sigillato in cui una macro genera un set con tutte le istanze in fase di compilazione. Ciò omette lo svantaggio che uno sviluppatore può aggiungere un valore all'enumerazione ma dimentica di aggiungerlo al set allElements.

Questa variante diventa particolarmente utile per grandi enumerazioni.

```
import EnumerationMacros._

sealed trait WeekDay
object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
  val allWeekDays: Set[WeekDay] = sealedInstancesOf[WeekDay]
}
```

Per far funzionare questo è necessario questa macro:

```
import scala.collection.immutable.TreeSet
import scala.language.experimental.macros
import scala.reflect.macros.blackbox

/**
A macro to produce a TreeSet of all instances of a sealed trait.
Based on Travis Brown's work:
http://stackoverflow.com/questions/13671734/iteration-over-a-sealed-trait-in-scala
CAREFUL: !!! MUST be used at END OF code block containing the instances !!!
*/
object EnumerationMacros {
  def sealedInstancesOf[A]: TreeSet[A] = macro sealedInstancesOf_impl[A]

  def sealedInstancesOf_impl[A: c.WeakTypeTag](c: blackbox.Context) = {
    import c.universe._

    val symbol = weakTypeOf[A].typeSymbol.asClass

    if  (!symbol.isClass || !symbol.isSealed)
      c.abort(c.enclosingPosition, "Can only enumerate values of a sealed trait or class.")
    else {

      val children = symbol.knownDirectSubclasses.toList

      if (!children.forall(_.isModuleClass)) c.abort(c.enclosingPosition, "All children must
be objects.")
      else c.Expr[TreeSet[A]] {

        def sourceModuleRef(sym: Symbol) =
Ident(sym.asInstanceOf[scala.reflect.internal.Symbols#Symbol
          ].sourceModule.asInstanceOf[Symbol]
        )

        Apply(
          Select(
            reify(TreeSet).tree,
            TermName("apply")
          ),
          children.map(sourceModuleRef(_))
        )
      }
    }
  }
}
```

Leggi enumerazioni online: https://riptutorial.com/it/scala/topic/1499/enumerazioni

Capitolo 12: Espressioni regolari

Sintassi

- re.findAllIn (s: CharSequence): MatchIterator
- re.findAllMatchIn (s: CharSequence): Iterator [Match]
- re.findFirstIn (s: CharSequence): Option [String]
- re.findFirstMatchIn (s: CharSequence): Option [Match]
- re.findPrefixMatchIn (s: CharSequence): Opzione [Match]
- re.findPrefixOf (s: CharSequence): Option [String]
- re.replaceAllIn (s: CharSequence, replacer: Match => String): String
- re.replaceAllIn (s: CharSequence, replacement: String): String
- re.replaceFirstIn (s: CharSequence, replacement: String): String
- re.replaceSomeIn (s: CharSequence, replacer: Match => Option [String]): String
- re.split (s: CharSequence): Array [String]

Examples

Dichiarazione di espressioni regolari

Il metodo `r` implicitamente fornito tramite scala.collection.immutable.StringOps produce un'istanza di scala.util.matching.Regex dalla stringa dell'oggetto. La sintassi delle stringhe a tre virgolette di Scala è utile qui, dato che non devi sfuggire alle barre rovesciate come faresti in Java:

```
val r0: Regex = """(\d{4})-(\d${2})-(\d{2})""".r     // :)
val r1: Regex = "(\\d{4})-(\\d{2})-(\\d{2})".r // :(
```

scala.util.matching.Regex implementa un'API di espressioni regolari idiomatiche per Scala come wrapper su java.util.regex.Pattern e la sintassi supportata è la stessa. Detto questo, il supporto di Scala per i letterali di stringa multi-linea rende il flag `x` sostanzialmente più utile, consentendo commenti e ignorando gli spazi bianchi del pattern:

```
val dateRegex = """(?x:
   (\d{4}) # year
  -(\d{2}) # month
  -(\d{2}) # day
)""".r
```

Esiste una versione sovraccaricata di `r` , `def r(names: String*): Regex` che consente di assegnare nomi di gruppi alle acquisizioni del modello. Questo è un po 'fragile dato che i nomi sono dissociati dalle acquisizioni e dovrebbero essere utilizzati solo se l'espressione regolare verrà utilizzata in più posizioni:

```
"""(\d{4})-(\d{2})-(\d{2})""".r("y", "m", "d").findFirstMatchIn(str) match {
  case Some(matched) =>
    val y = matched.group("y").toInt
```

```
    val m = matched.group("m").toInt
    val d = matched.group("d").toInt
    java.time.LocalDate.of(y, m, d)
  case None => ???
}
```

Ripetizione della corrispondenza di un motivo in una stringa

```
val re = """\((.*?)\)""".r

val str =
"(The)(example)(of)(repeating)(pattern)(in)(a)(single)(string)(I)(had)(some)(trouble)(with)(once)"


re.findAllMatchIn(str).map(_.group(1)).toList
res2: List[String] = List(The, example, of, repeating, pattern, in, a, single, string, I, had,
some, trouble, with, once)
```

Leggi Espressioni regolari online: https://riptutorial.com/it/scala/topic/2891/espressioni-regolari

Capitolo 13: estrattori

Sintassi

- val extractor (extractedValue1, _ / * secondo valore estratto estrapolato * /) = valueToBeExtracted
- valueToBeStruttura abbinata {case extractor (extractedValue1, _) => ???}
- val (tuple1, tuple2, tuple3) = tupleWith3Elements
- oggetto Foo {def unapply (foo: Foo): Option [String] = Some (foo.x); }

Examples

Estrattori a tupla

`x` e `y` sono estratti dalla tupla:

```
val (x, y) = (1337, 42)
// x: Int = 1337
// y: Int = 42
```

Per ignorare un valore, utilizzare _ :

```
val (_, y: Int) = (1337, 42)
// y: Int = 42
```

Per decomprimere un estrattore:

```
val myTuple = (1337, 42)
myTuple._1  // res0: Int = 1337
myTuple._2  // res1: Int = 42
```

Nota che le tuple hanno una lunghezza massima di 22, e quindi `._1` da `._22` a `._22` (supponendo che la tupla abbia almeno quella dimensione).

Gli estrattori di tupla possono essere usati per fornire argomenti simbolici per le funzioni letterali:

```
val persons = List("A." -> "Lovelace", "G." -> "Hopper")
val names = List("Lovelace, A.", "Hopper, G.")

assert {
  names ==
    (persons map { name =>
      s"${name._2}, ${name._1}"
    })
}

assert {
  names ==
```

```
    (persons map { case (given, surname) =>
      s"$surname, $given"
    })
}
```

Estrattori Case Class

Una classe di casi è una classe con un codice standard di codice standard che è incluso automaticamente. Un vantaggio di questo è che Scala semplifica l'uso di estrattori con classi di casi.

```
case class Person(name: String, age: Int)  // Define the case class
val p = Person("Paola", 42)  // Instantiate a value with the case class type

val Person(n, a) = p  // Extract values n and a
// n: String = Paola
// a: Int = 42
```

In questo frangente, sia `n` che `a` sono `val` nel programma e sono accessibili come tali: si dice che siano stati "estratti" da p. continuando:

```
val p2 = Person("Angela", 1337)

val List(Person(n1, a1), Person(_, a2)) = List(p, p2)
// n1: String = Paola
// a1: Int = 42
// a2: Int = 1337
```

Qui vediamo due cose importanti:

- L'estrazione può avvenire a livelli "profondi": è possibile estrarre le proprietà degli oggetti nidificati.
- Non tutti gli elementi *devono* essere estratti. Il jolly _ carattere indica che quel particolare valore può essere qualsiasi cosa, e viene ignorato. Nessun `val` è stato creato.

In particolare, questo può semplificare la corrispondenza tra le raccolte:

```
val ls = List(p1, p2, p3)  // List of Person objects
ls.map(person => person match {
  case Person(n, a) => println("%s is %d years old".format(n, a))
})
```

Qui, abbiamo un codice che usa l'extractor per verificare esplicitamente che la `person` sia un oggetto `Person` e immediatamente estrae le variabili che ci interessano: `n` e `a` .

Unapply - Estrattori personalizzati

È possibile scrivere un'estrazione personalizzata implementando il metodo `unapply` e restituendo un valore di tipo `Option` :

```
class Foo(val x: String)
```

```
object Foo {
  def unapply(foo: Foo): Option[String] = Some(foo.x)
}

new Foo("42") match {
  case Foo(x) => x
}
// "42"
```

Il tipo restituito di `unapply` potrebbe essere qualcosa di diverso da `Option` , a condizione che il tipo restituito fornisca i metodi `get` e `isEmpty` . In questo esempio, la `Bar` viene definita con tali metodi e riappare in modo `unapply` un'istanza di `Bar` :

```
class Bar(val x: String) {
  def get = x
  def isEmpty = false
}

object Bar {
  def unapply(bar: Bar): Bar = bar
}

new Bar("1337") match {
  case Bar(x) => x
}
// "1337"
```

Il tipo restituito di `unapply` può anche essere un `Boolean` , che è un caso speciale che non ha i requisiti `get` ed `isEmpty` sopra. Tuttavia, si noti in questo esempio che `DivisibleByTwo` è un oggetto, non una classe, e non accetta un parametro (e quindi tale parametro non può essere associato):

```
object DivisibleByTwo {
  def unapply(num: Int): Boolean = num % 2 == 0
}

4 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// yes

3 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// no
```

Ricorda che `unapply` va nell'oggetto compagno di una classe, non nella classe. L'esempio sopra sarà chiaro se capisci questa distinzione.

Notazione Infix dell'estrattore

Se una classe del caso ha esattamente due valori, il suo estrattore può essere utilizzato nella notazione infisso.

```
case class Pair(a: String, b: String)
val p: Pair = Pair("hello", "world")
val x Pair y = p
//x: String = hello
//y: String = world
```

Qualsiasi estrattore che restituisce una tupla da 2 può funzionare in questo modo.

```
object Foo {
    def unapply(s: String): Option[(Int, Int)] = Some((s.length, 5))
}
val a Foo b = "hello world!"
//a: Int = 12
//b: Int = 5
```

Estrattori Regex

Un'espressione regolare con parti raggruppate può essere utilizzata come estrattore:

```
scala> val address = """(.+):(\d+)""".r
address: scala.util.matching.Regex = (.+):(\d+)

scala> val address(host, port) = "some.domain.org:8080"
host: String = some.domain.org
port: String = 8080
```

Si noti che quando non è abbinato, viene `MatchError` un `MatchError` in fase di runtime:

```
scala> val address(host, port) = "something not a host and port"
scala.MatchError: something not a host and port (of class java.lang.String)
```

Estrattori trasformativi

Il comportamento dell'estrattore può essere utilizzato per ricavare valori arbitrari dal loro input. Ciò può essere utile negli scenari in cui si desidera essere in grado di agire sui risultati di una trasformazione nel caso in cui la trasformazione abbia esito positivo.

Considerare come esempio i vari formati di nome utente utilizzabili in un ambiente Windows :

```
object UserPrincipalName {
  def unapply(str: String): Option[(String, String)] = str.split('@') match {
    case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
    case _ => None
  }
}

object DownLevelLogonName {
  def unapply(str: String): Option[(String, String)] = str.split('\\') match {
    case Array(d, u) if u.length > 0 && d.length > 0 => Some((d, u))
    case _ => None
  }
}
```

```
def getDomain(str: String): Option[String] = str match {
  case UserPrincipalName(_, domain) => Some(domain)
  case DownLevelLogonName(domain, _) => Some(domain)
  case _ => None
}
```

In effetti è possibile creare un estrattore che mostri entrambi i comportamenti allargando i tipi che può abbinare:

```
object UserPrincipalName {
  def unapply(obj: Any): Option[(String, String)] = obj match {
    case upn: UserPrincipalName => Some((upn.username, upn.domain))
    case str: String => str.split('@') match {
      case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
      case _ => None
    }
    case _ => None
  }
}
```

In generale, gli estrattori sono semplicemente una conveniente riformulazione del pattern `Option` , applicato ai metodi con nomi come `tryParse` :

```
UserPrincipalName.unapply("user@domain") match {
  case Some((u, d)) => ???
  case None => ???
}
```

Leggi estrattori online: https://riptutorial.com/it/scala/topic/930/estrattori

Capitolo 14: Funzione ordine superiore

Osservazioni

Scala fa di tutto per trattare i metodi e le funzioni in modo sintatticamente identico. Ma sotto il cofano, sono concetti distinti.

Un metodo è codice eseguibile e non ha alcuna rappresentazione di valore.

Una funzione è un'istanza di oggetto reale di tipo `Function1` (o un tipo simile di un'altra arity). Il suo codice è contenuto nel suo metodo di `apply` . Effettivamente, agisce semplicemente come un valore che può essere passato in giro.

Per inciso, la capacità di trattare le funzioni come valori *è* esattamente ciò che si intende per linguaggio che supporta le funzioni di ordine superiore. Le istanze di funzione sono l'approccio di Scala all'implementazione di questa funzione.

Una funzione reale di ordine superiore è una funzione che accetta un valore di funzione come argomento o restituisce un valore di funzione. Ma in Scala, poiché tutte le operazioni sono metodi, è più generale pensare a metodi che ricevono o restituiscono parametri di funzione. Quindi la `map` , come definita in `Seq` potrebbe essere pensata come una "funzione di ordine superiore" a causa del fatto che il suo parametro è una funzione, ma non è letteralmente una funzione; è un metodo.

Examples

Utilizzo dei metodi come valori di funzione

Il compilatore Scala convertirà automaticamente i metodi in valori di funzione allo scopo di passarli in funzioni di ordine superiore.

```
object MyObject {
  def mapMethod(input: Int): String = {
    int.toString
  }
}

Seq(1, 2, 3).map(MyObject.mapMethod) // Seq("1", "2", "3")
```

Nell'esempio sopra, `MyObject.mapMethod` non è una chiamata di funzione, ma viene invece passata alla `map` come valore. Infatti, la `map` *richiede* un valore di funzione passato ad esso, come si può vedere nella sua firma. La firma per la `map` di un `List[A]` (un elenco di oggetti di tipo `A`) è:

```
def map[B](f: (A) ⇒ B): List[B]
```

La parte `f: (A) => B` indica che il parametro per questa chiamata di metodo è una funzione che accetta un oggetto di tipo `A` e restituisce un oggetto di tipo `B` `A` e `B` sono definiti arbitrariamente. Tornando al primo esempio, possiamo vedere che `mapMethod` accetta un `Int` (che corrisponde ad `A`)

e restituisce una `String` (che corrisponde a `B`). Quindi `mapMethod` è un valore di funzione valido per passare alla `map` . Potremmo riscrivere lo stesso codice come questo:

```
Seq(1, 2, 3).map(x:Int => int.toString)
```

Questo inline il valore della funzione, che può aggiungere chiarezza per le funzioni semplici.

Funzioni di ordine elevato (funzione come parametro)

Una funzione di ordine superiore, al contrario di una funzione di primo ordine, può avere una delle tre forme:

- Uno o più dei suoi parametri è una funzione e restituisce un valore.
- Restituisce una funzione, ma nessuno dei suoi parametri è una funzione.
- Entrambi i precedenti: uno o più dei suoi parametri è una funzione e restituisce una funzione.

```
object HOF {
    def main(args: Array[String]) {
    val list =
List(("Srini","E"),("Subash","R"),("Ranjith","RK"),("Vicky","s"),("Sudhar","s"))
    //HOF
     val fullNameList= list.map(n => getFullName(n._1, n._2))

     }

    def getFullName(firstName: String, lastName: String): String = firstName + "." +
lastName
    }
```

Qui la funzione mappa assume come parametri un parametro `getFullName(n._1,n._2)` . Questo è chiamato **HOF (funzione di ordine superiore).**

Argomenti valutazione pigra

Scala supporta la valutazione lazy per gli argomenti delle funzioni usando notazione: `def func(arg: => String)` . Tale argomento di funzione potrebbe richiedere un oggetto `String` regolare o una funzione di ordine superiore con tipo di restituzione `String` . Nel secondo caso, l'argomento della funzione verrebbe valutato sull'accesso valore.

Si prega di vedere l'esempio:

```
def calculateData: String = {
  print("Calculating expensive data! ")
  "some expensive data"
}

def dumbMediator(preconditions: Boolean, data: String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
```

```
    case false => None
  }
}

def smartMediator(preconditions: Boolean, data: => String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

smartMediator(preconditions = false, calculateData)

dumbMediator(preconditions = false, calculateData)
```

`smartMediator` chiamata `smartMediator` restituirà None value e stamperà il messaggio `"Applying mediator"` .

`dumbMediator` chiamata `dumbMediator` restituirebbe Nessun valore e stamperà il messaggio `"Calculating expensive data! Applying mediator"` .

La valutazione pigra potrebbe essere estremamente utile quando si desidera ottimizzare un sovraccarico di costosi calcoli di argomenti.

Leggi Funzione ordine superiore online: https://riptutorial.com/it/scala/topic/1642/funzione-ordine-superiore

Capitolo 15: funzioni

Osservazioni

Scala ha funzioni di prima classe.

Differenza tra funzioni e metodi:

Una funzione non è un metodo in Scala: le funzioni sono un valore e possono essere assegnate come tali. I metodi (creati usando `def`), d'altra parte, devono appartenere a una classe, un tratto o un oggetto.

- Le funzioni sono compilate in una classe che estende un tratto (come `Function1`) in fase di compilazione e sono istanziate a un valore in fase di esecuzione. I metodi, d'altra parte, sono membri della loro classe, tratto o oggetto e non esistono al di fuori di questo.
- Un metodo può essere convertito in una funzione, ma una funzione non può essere convertita in un metodo.
- I metodi possono avere la parametrizzazione dei tipi, mentre le funzioni no.
- I metodi possono avere valori di default dei parametri, mentre le funzioni no.

Examples

Funzioni anonime

Le funzioni anonime sono funzioni definite ma non assegnate a un nome.

La seguente è una funzione anonima che accetta due interi e restituisce la somma.

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

L'espressione risultante può essere assegnata a un `val` :

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

Le funzioni anonime vengono principalmente utilizzate come argomenti per altre funzioni. Ad esempio, la funzione `map` su una raccolta prevede un'altra funzione come argomento:

```
// Returns Seq("FOO", "BAR", "QUX")
Seq("Foo", "Bar", "Qux").map((x: String) => x.toUpperCase)
```

I tipi di argomenti della funzione anonima possono essere omessi: i tipi vengono dedotti automaticamente :

```
Seq("Foo", "Bar", "Qux").map((x) => x.toUpperCase)
```

Se c'è un solo argomento, le parentesi attorno a quell'argomento possono essere omesse:

```
Seq("Foo", "Bar", "Qux").map(x => x.toUpperCase)
```

Sottolinea la stenografia

C'è una sintassi ancora più breve che non richiede nomi per gli argomenti. Lo snippet sopra riportato può essere scritto:

```
Seq("Foo", "Bar", "Qux").map(_.toUpperCase)
```

_ rappresenta gli argomenti della funzione anonima in modo posizionale. Con una funzione anonima che ha più parametri, ogni occorrenza di _ farà riferimento a un argomento diverso. Ad esempio, le due espressioni seguenti sono equivalenti:

```
// Returns "FooBarQux" in both cases
Seq("Foo", "Bar", "Qux").reduce((s1, s2) => s1 + s2)
Seq("Foo", "Bar", "Qux").reduce(_ + _)
```

Quando si utilizza questa stenografia, qualsiasi argomento rappresentato da _ posizionale può essere fatto riferimento una sola volta e nello stesso ordine.

Funzioni anonime senza parametri

Per creare un valore per una funzione anonima che non accetta parametri, lascia vuoto l'elenco dei parametri:

```
val sayHello = () => println("hello")
```

Composizione

La composizione delle funzioni consente di far funzionare due funzioni e di essere viste come una singola funzione. Espresso in termini matematici, data una funzione `f(x)` e una funzione `g(x)` , la funzione `h(x) = f(g(x))` .

Quando una funzione viene compilata, viene compilata su un tipo correlato a `Function1` . Scala fornisce due metodi nell'implementazione `Function1` relativa alla composizione: `andThen` e `compose` . Il metodo di `compose` adatta alla definizione matematica di cui sopra in questo modo:

```
val f: B => C = ...
val g: A => B = ...

val h: A => C = f compose g
```

The `andThen` (penso che `h(x) = g(f(x))`) abbia un sentimento più simile a 'DSL':

```
val f: A => B = ...
val g: B => C = ...

val h: A => C = f andThen g
```

Una nuova funzione anonima viene allocato con che è chiuso su `f` e `g` . Questa funzione è associata alla nuova funzione `h` in entrambi i casi.

```
def andThen(g: B => C): A => C = new (A => C){
  def apply(x: A) = g(self(x))
}
```

Se uno dei due `f` o `g` funziona tramite un effetto collaterale, quindi chiamando `h` causerà tutti gli effetti collaterali di `f` e `g` per accadere nell'ordine. Lo stesso vale per qualsiasi cambiamento di stato mutabile.

Relazione con le funzioni parziali

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

Ogni singola funzione `PartialFunction` è anche una `Function1` . Questo è contro-intuitivo in senso matematico formale, ma si adatta meglio alla progettazione orientata agli oggetti. Per questo motivo `Function1` non deve fornire un metodo `isDefinedAt` `true` `isDefinedAt` .

Per definire una funzione parziale (che è anche una funzione), utilizzare la seguente sintassi:

```
{ case i: Int => i + 1 } // or equivalently { case i: Int ⇒ i + 1 }
```

Per ulteriori dettagli, dare un'occhiata a PartialFunctions .

Leggi funzioni online: https://riptutorial.com/it/scala/topic/477/funzioni

Capitolo 16: Funzioni definite dall'utente per Hive

Examples

Una semplice UDF di Hive in Apache Spark

```
import org.apache.spark.sql.functions._

// Create a function that uses the content of the column inside the dataframe
val code = (param: String) => if (param == "myCode") 1 else 0
// With that function, create the udf function
val myUDF = udf(code)
// Apply the udf to a column inside the existing dataframe, creating a dataframe with the
additional new column
val newDataframe = aDataframe.withColumn("new_column_name", myUDF(col(inputColumn)))
```

Leggi Funzioni definite dall'utente per Hive online:
https://riptutorial.com/it/scala/topic/8241/funzioni-definite-dall-utente-per-hive

Capitolo 17: Funzioni parziali

Examples

Composizione

Le funzioni parziali sono spesso utilizzate per definire una funzione totale in parti:

```
sealed trait SuperType
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val pfA: PartialFunction[SuperType, Int] = {
  case A => 5
}

val pfB: PartialFunction[SuperType, Int] = {
  case B => 10
}

val input: Seq[SuperType] = Seq(A, B, C)

input.map(pfA orElse pfB orElse {
  case _ => 15
}) // Seq(5, 10, 15)
```

In questo utilizzo, le funzioni parziali vengono tentate in ordine di concatenazione con il metodo `orElse` . In genere, viene fornita una funzione parziale finale che corrisponde a tutti i casi rimanenti. Collettivamente, la combinazione di queste funzioni agisce come una funzione totale.

Questo schema viene in genere utilizzato per separare le preoccupazioni in cui una funzione può effettivamente agire come un dispatcher per percorsi di codice disparati. Questo è comune, ad esempio, nel metodo di ricezione di un attore Akka .

Utilizzo con `collect`

Mentre le funzioni parziali sono spesso usate come sintassi conveniente per le funzioni totali, includendo una corrispondenza jolly finale (`case _`), in alcuni metodi, la loro parzialità è la chiave. Un esempio molto comune in Scala idiomatica è il metodo `collect` , definito nella libreria delle collezioni Scala. Qui, le funzioni parziali consentono alle funzioni comuni di esaminare gli elementi di una raccolta per mapparle e / o filtrarle in modo che si verifichino in una sintassi compatta.

Esempio 1

Supponendo che abbiamo una funzione radice quadrata definita come funzione parziale:

```
val sqRoot:PartialFunction[Double,Double] = { case n if n > 0 => math.sqrt(n) }
```

Possiamo invocarlo con il combinatore di `collect` :

```
List(-1.1,2.2,3.3,0).collect(sqRoot)
```

eseguendo efficacemente la stessa operazione di:

```
List(-1.1,2.2,3.3,0).filter(sqRoot.isDefinedAt).map(sqRoot)
```

Esempio 2

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case class A(value: Int) extends SuperType
case class B(text: String) extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A(5), B("hello"), C, A(25), B(""))

input.collect {
  case A(value) if value < 10   => value.toString
  case B(text) if text.nonEmpty => text
} // Seq("5", "hello")
```

Ci sono diverse cose da notare nell'esempio sopra:

- Il lato sinistro di ogni modello corrisponde in modo efficace agli elementi da elaborare e includere nell'output. Qualsiasi valore che non ha un `case` corrispondente viene semplicemente omesso.
- Il lato destro definisce l'elaborazione specifica del caso da applicare.
- La corrispondenza del modello lega la variabile per l'uso nelle istruzioni di guardia (le clausole `if`) e nella parte destra.

Sintassi di base

Scala ha un particolare tipo di funzione chiamata una funzione parziale , che si estende normali funzioni - significa che una `PartialFunction` istanza può essere utilizzato ovunque `Function1` è previsto. Le funzioni parziali possono essere definite in modo anonimo utilizzando la sintassi del `case` utilizzata anche nella corrispondenza del modello :

```
val pf: PartialFunction[Boolean, Int] = {
  case true => 7
}

pf.isDefinedAt(true) // returns true
pf(true) // returns 7

pf.isDefinedAt(false) // returns false
pf(false) // throws scala.MatchError: false (of class java.lang.Boolean)
```

Come visto nell'esempio, non è necessario definire una funzione parziale sull'intero dominio del suo primo parametro. Si presuppone che un'istanza `Function1` standard sia *totale* , ovvero che sia definita per ogni argomento possibile.

Utilizzo come funzione totale

Le funzioni parziali sono molto comuni in Scala idiomatica. Sono spesso usati per la loro sintassi basata su un `case` conveniente per definire le funzioni totali sui tratti :

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A, B, C)

input.map {
  case A => 5
  case _ => 10
} // Seq(5, 10, 10)
```

Questo salva la sintassi aggiuntiva di un'istruzione `match` in una normale funzione anonima. Confrontare:

```
input.map { item =>
  item match {
    case A => 5
    case _ => 10
  }
} // Seq(5, 10, 10)
```

Viene anche utilizzato frequentemente per eseguire una scomposizione dei parametri utilizzando la corrispondenza del modello, quando una tupla o una classe case viene passata a una funzione:

```
val input = Seq("A" -> 1, "B" -> 2, "C" -> 3)

input.map { case (a, i) =>
   a + i.toString
} // Seq("A1", "B2", "C3")
```

Utilizzo per estrarre le tuple in una funzione mappa

Queste tre funzioni della mappa sono equivalenti, quindi usa la variazione che il tuo team trova più leggibile.

```
val numberNames = Map(1 -> "One", 2 -> "Two", 3 -> "Three")

// 1. No extraction
numberNames.map(it => s"${it._1} is written ${it._2}" )

// 2. Extraction within a normal function
numberNames.map(it => {
    val (number, name) = it
    s"$number is written $name"
})

// 3. Extraction via a partial function (note the brackets in the parentheses)
numberNames.map({ case (number, name) => s"$number is written $name" })
```

La funzione parziale **deve corrispondere a tutti gli input** : ogni caso che non corrisponde genererà un'eccezione in fase di esecuzione.

Leggi Funzioni parziali online: https://riptutorial.com/it/scala/topic/1638/funzioni-parziali

Capitolo 18: Futures

Examples

Creare un futuro

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutureDivider {
    def divide(a: Int, b: Int): Future[Int] = Future {
        // Note that this is integer division.
        a / b
    }
}
```

Molto semplicemente, il metodo di `divide` crea un futuro che si risolverà con il quoziente di `a b` .

Consumare un futuro di successo

Il modo più semplice per consumare un futuro di *successo* - o meglio, ottenere il valore all'interno del futuro - è utilizzare il metodo della `map` . Supponiamo che alcuni codici chiamino il metodo di `divide` dell'oggetto `FutureDivider` dall'esempio "Creating a Future". Come dovrebbe apparire il codice per ottenere il quoziente di `a b` ?

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(quotient)
        }
    }
}
```

Consumare un futuro fallito

A volte il calcolo in un futuro può creare un'eccezione, che farà fallire il futuro. Nell'esempio "Creating a Future", cosa accadrebbe se il codice chiamante passasse `55` e `0` al metodo `divide` ? Avrebbe gettato un `ArithmeticException` dopo aver provato a dividere per zero, ovviamente. Come sarebbe gestito nel codice che consuma? Ci sono in realtà una manciata di modi per affrontare i fallimenti.

Gestire l'eccezione con il `recover` e la corrispondenza del modello.

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)
```

```
        eventualQuotient recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Gestire l'eccezione con il `failed` proiezione, dove l'eccezione diventa il valore del futuro:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        // Note the use of the dot operator to get the failed projection and map it.
        eventualQuotient.failed.map {
            ex => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Mettere insieme il futuro

Gli esempi precedenti hanno dimostrato le caratteristiche individuali di un futuro, gestendo casi di successo e fallimento. Di solito, tuttavia, entrambe le funzionalità vengono gestite in modo molto più preciso. Ecco l'esempio, scritto in modo più ordinato e più realistico:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(s"Quotient: $quotient")
        } recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Sequencing and traversing Futures

In alcuni casi è necessario calcolare una quantità variabile di valori su Futures separati. Si supponga di avere una `List[Future[Int]]` , ma invece una `List[Int]` deve essere elaborata. Quindi la domanda è come trasformare questa istanza di `List[Future[Int]]` in un `Future[List[Int]]` . Per questo scopo c'è il metodo di `sequence` sull'oggetto compagno `Future` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.sequence(listOfFuture)
```

In generale, la `sequence` è un operatore comunemente noto nel mondo della programmazione funzionale che trasforma `F[G[T]]` in `G[F[T]]` con restrizioni a `F` e `G`

Esiste un operatore alternativo chiamato `traverse` , che funziona in modo simile ma prende una funzione come argomento aggiuntivo. Con la funzione di identità `x => x` come parametro si

comporta come l'operatore di `sequence` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.traverse(listOfFuture)(x => x)
```

Tuttavia, l'argomento extra consente di modificare ogni istanza futura all'interno della `listOfFuture` . Inoltre, il primo argomento non deve essere una lista di `Future` . Pertanto è possibile trasformare l'esempio come segue:

```
def futureOfList: Future[List[Int]] = Future.traverse(List(1,2,3))(Future(_))
```

In questo caso l' `List(1,2,3)` viene passato direttamente come primo argomento e la funzione di identità `x => x` viene sostituita con la funzione `Future(_)` per avvolgere in modo simile ciascun valore `Int` in un `Future` . Un vantaggio di questo è che l' `List[Future[Int]]` intermedio `List[Future[Int]]` può essere omesso per migliorare le prestazioni.

Combina più Futures - Per Comprensione

La *comprensione* è un modo compatto per eseguire un blocco di codice che dipende dal risultato positivo di più futuri.

Con `f1, f2, f3` tre `Future[String]` che conterranno le stringhe `one, two, three` rispettivamente,

```
val fCombined =
    for {
        s1 <- f1
        s2 <- f2
        s3 <- f3
    } yield (s"$s1 - $s2 - $s3")
```

`fCombined` sarà una `Future[String]` contenente la stringa `one - two - three` una volta che tutti i futures sono stati completati con successo.

Si noti che qui viene assunto un implicito ExectionContext.

Inoltre, tieni a mente che per la comprensione è solo uno zucchero sintattico per un metodo flatMap, quindi la costruzione di oggetti futuri all'interno del corpo eliminerebbe l'esecuzione simultanea di blocchi di codice racchiusi dai futures e condurrebbe al codice sequenziale. Lo vedi nell'esempio:

```
val result1 = for {
  first <- Future {
    Thread.sleep(2000)
    System.currentTimeMillis()
  }
  second <- Future {
    Thread.sleep(1000)
    System.currentTimeMillis()
  }
} yield first - second
```

```
val fut1 = Future {
  Thread.sleep(2000)
  System.currentTimeMillis()
}
val fut2 = Future {
  Thread.sleep(1000)
  System.currentTimeMillis()
}
val result2 = for {
  first <- fut1
  second <- fut2
} yield first - second
```

Il valore racchiuso dall'oggetto `result1` sarebbe sempre negativo mentre il `result2` sarebbe positivo.

Per maggiori dettagli sulla *comprensione* e sulla `yield` in generale, vedi http://docs.scala-lang.org/tutorials/FAQ/yield.html

Leggi Futures online: https://riptutorial.com/it/scala/topic/3245/futures

Capitolo 19: Gestione degli errori

Examples

Provare

Usando Prova con `map` , `getOrElse` e `flatMap` :

```
import scala.util.Try

val i = Try("123".toInt)      // Success(123)
i.map(_ + 1).getOrElse(321)   // 124

val j = Try("abc".toInt)      // Failure(java.lang.NumberFormatException)
j.map(_ + 1).getOrElse(321)   // 321

Try("123".toInt) flatMap { i =>
  Try("234".toInt)
    .map(_ + i)
}                             // Success(357)
```

Uso di `Try` con corrispondenza del modello:

```
Try(parsePerson("John Doe")) match {
    case Success(person) => println(person.surname)
    case Failure(ex) => // Handle error ...
}
```

O

Diversi tipi di dati per errore / successo

```
def getPersonFromWebService(url: String): Either[String, Person] = {

    val response = webServiceClient.get(url)

    response.webService.status match {
        case 200 => {
            val person = parsePerson(response)
            if(!isValid(person)) Left("Validation failed")
            else Right(person)
        }

        case _ => Left(s"Request failed with error code $response.status")
    }
}
```

Pattern matching su entrambi i valori

```
getPersonFromWebService("http://some-webservice.com/person") match {
    case Left(errorMessage) => println(errorMessage)
```

```
    case Right(person) => println(person.surname)
}
```

Converti un valore in Opzione

```
val maybePerson: Option[Person] = getPersonFromWebService("http://some-
webservice.com/person").right.toOption
```

Opzione

L'uso di valori `null` è fortemente sconsigliato, a meno che non si interagisca con il codice Java legacy che si aspetta `null` . Invece, `Option` dovrebbe essere usato quando il risultato di una funzione potrebbe essere qualcosa (`Some`) o nothing (`None`).

Un blocco try-catch è più appropriato per la gestione degli errori, ma se la funzione potrebbe legittimamente non restituire nulla, `Option` è appropriato da usare e semplice.

`Option[T]` può essere `Some(value)` (contiene un valore di tipo `T`) o `None` :

```
def findPerson(name: String): Option[Person]
```

Se nessuna persona viene trovata, `None` può essere restituita. Altrimenti, viene restituito un oggetto di tipo `Some` contenente un oggetto `Person` . Quello che segue sono i modi per gestire un oggetto di tipo `Option` .

Pattern Matching

```
findPerson(personName) match {
    case Some(person) => println(person.surname)
    case None => println(s"No person found with name $personName")
}
```

Utilizzando la mappa e getOrElse

```
val name = findPerson(personName).map(_.firstName).getOrElse("Unknown")
println(name) // Prints either the name of the found person or "Unknown"
```

Usando la piega

```
val name = findPerson(personName).fold("Unknown")(_.firstName)
// equivalent to the map getOrElse example above.
```

Conversione in Java

Se è necessario convertire un tipo di `Option` in un tipo Java null-enabled per l'interoperabilità:

```
val s: Option[String] = Option("hello")
s.orNull           // "hello": String
s.getOrElse(null)  // "hello": String

val n: Option[Int] = Option(42)
n.orNull           // compilation failure (Cannot prove that Null <:< Int.)
n.getOrElse(null)  // 42
```

Gestione degli errori originati nei futures

Quando `exception` viene lanciata dall'interno di un `Future` , puoi (dovresti) utilizzare `recover` per gestirlo.

Per esempio,

```
def runFuture: Future = Future { throw new FairlyStupidException }

val itWillBeAwesome: Future = runFuture
```

... genererà `Exception` dall'interno del `Future` . Ma visto che siamo in grado di prevedere che un `Exception` di tipo `FairlyStupidException` con una probabilità alta, siamo in grado di gestire questo specifico caso in modo elegante:

```
val itWillBeAwesomeOrIllRecover = runFuture recover {
    case stupid: FairlyStupidException =>
         BadRequest("Another stupid exception!")
}
```

Come puoi vedere il metodo dato per `recover` è una `PartialFunction` sul dominio di tutti i `Throwable` , quindi puoi gestire solo alcuni tipi e lasciare che il resto entri nell'etere della gestione delle eccezioni a livelli più alti nello stack `Future` .

Si noti che questo è simile all'esecuzione del seguente codice in un contesto non `Future` :

```
def runNotFuture: Unit = throw new FairlyStupidException

try {
    runNotFuture
} catch {
    case e: FairlyStupidException => BadRequest("Another stupid exception!")
}
```

È molto importante gestire le eccezioni generate all'interno di `Future` perché per la maggior parte del tempo sono più insidiose. Di solito non riescono a scappare, perché corrono in un contesto di esecuzione e thread diversi, e quindi non ti chiedono di correggerli quando accadono, specialmente se non noti nulla nei log o il comportamento del applicazione.

Usando le clausole try-catch

Oltre ai costrutti funzionali come `Try` , `Option` e `Either` per la gestione degli errori, Scala supporta anche una sintassi simile a quella di Java, utilizzando una clausola try-catch (con un potenziale

blocco finally). La clausola catch è un pattern match:

```
try {
  // ... might throw exception
} catch {
  case ioe: IOException => ... // more specific cases first
  case e: Exception => ...
  // uncaught types will be thrown
} finally {
  // ...
}
```

Convertire le eccezioni in entrambi i tipi di opzioni

Per convertire le eccezioni in tipi `Either` o `Option` , è possibile utilizzare i metodi forniti in `scala.util.control.Exception`

```
import scala.util.control.Exception._

val plain = "71a"
val optionInt: Option[Int] = catching(classOf[java.lang.NumberFormatException]) opt {
plain.toInt }
val eitherInt = Either[Throwable, Int] = catching(classOf[java.lang.NumberFormatException])
either { plain.toInt }
```

Leggi Gestione degli errori online: https://riptutorial.com/it/scala/topic/910/gestione-degli-errori

Capitolo 20: Gestione XML

Examples

Beautify o Pretty-Print XML

L'utilità `PrettyPrinter` produrrà documenti XML "carina". Il seguente frammento di codice consente di stampare xml non formattato:

```
import scala.xml.{PrettyPrinter, XML}
val xml = XML.loadString("<a>Alana<b><c>Beth</c><d>Catie</d></b></a>")
val formatted = new PrettyPrinter(150, 4).format(xml)
print(formatted)
```

Questo produrrà il contenuto usando una larghezza di pagina di `150` e una costante di indentazione di `4` caratteri spazi bianchi:

```
<a>
    Alana
    <b>
        <c>Beth</c>
        <d>Catie</d>
    </b>
</a>
```

È possibile utilizzare `XML.loadFile("nameoffile.xml")` per caricare xml da un file anziché da una stringa.

Leggi Gestione XML online: https://riptutorial.com/it/scala/topic/1453/gestione-xml

Capitolo 21: I flussi

Osservazioni

Gli stream vengono valutati pigramente, nel senso che possono essere utilizzati per implementare i generatori, che forniranno o "genereranno" un nuovo elemento del tipo specificato su richiesta, piuttosto che prima del fatto. Ciò garantisce che vengano eseguiti solo i calcoli necessari.

Examples

Utilizzo di un flusso per generare una sequenza casuale

`genRandom` crea un flusso di numeri casuali che ha una possibilità su quattro di terminare ogni volta che viene chiamato.

```
def genRandom: Stream[String] = {
  val random = scala.util.Random.nextFloat()
  println(s"Random value is: $random")
  if (random < 0.25) {
    Stream.empty[String]
  } else {
    ("%.3f : A random number" format random) #:: genRandom
  }
}

lazy val randos = genRandom  // getRandom is lazily evaluated as randos is iterated through

for {
  x <- randos
} println(x) // The number of times this prints is effectively randomized.
```

Nota il costrutto `#::` , che *ricorre pigramente* : poiché sta anteponendo il numero casuale corrente a uno stream, non valuta il resto del flusso finché non viene iterato.

Stream infiniti tramite ricorsione

Si possono costruire flussi che si riferiscono a se stessi e quindi diventano infinitamente ricorsivi.

```
// factorial
val fact: Stream[BigInt] = 1 #:: fact.zipWithIndex.map{case (p,x)=>p*(x+1)}
fact.take(10)  // (1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
fact(24)       // 620448401733239439360000

// the Fibonacci series
val fib: Stream[BigInt] = 0 #:: fib.scan(1:BigInt)(_+_)
fib.take(10)  // (0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
fib(124)      // 36726740705505779255899443

// random Ints between 10 and 99 (inclusive)
def rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(10)  // (20, 95, 14, 44, 42, 78, 85, 24, 99, 85)
```

In questo contesto, la differenza tra Var, Val e Def è interessante. Come `def` ogni elemento viene ricalcolato ogni volta che viene referenziato. Come valore `val` ogni elemento viene mantenuto e riutilizzato dopo che è stato calcolato. Questo può essere dimostrato creando un effetto collaterale con ogni calcolo.

```
// def with extra output per calculation
def fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!!!!!!!!!!! 120
fact(4)  // !!!!!!!!!! 24
fact(7)  // !!!!!!!!!!!!!!!!!!!!!!!!!!!! 5040

// now as val
val fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!! 120
fact(4)  // 24
fact(7)  // !! 5040
```

Questo spiega anche perché il numero casuale `Stream` non funziona come `val` .

```
val rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(5)  // (79, 79, 79, 79, 79)
```

Stream infinito autoreferenziale

```
// Generate stream that references itself in its evaluation
lazy val primes: Stream[Int] =
  2 #:: Stream.from(3, 2)
    .filter { i => primes.takeWhile(p => p * p <= i).forall(i % _ != 0) }
    .takeWhile(_ > 0) // prevent overflowing

// Get list of 10 primes
assert(primes.take(10).toList == List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29))

// Previously calculated values were memoized, as shown by toString
assert(primes.toString == "Stream(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ?)")
```

Leggi I flussi online: https://riptutorial.com/it/scala/topic/3702/i-flussi

Capitolo 22: impliciti

Sintassi

- implicito val x: T = ???

Osservazioni

Le classi implicite consentono di aggiungere metodi personalizzati a tipi esistenti, senza dover modificare il loro codice, arricchendo così i tipi senza dover controllare il codice.

L'uso di tipi impliciti per arricchire una classe esistente viene spesso definito come un modello "arricchisci la mia biblioteca".

Restrizioni su classi implicite

1. Le classi implicite possono esistere solo all'interno di un'altra classe, oggetto o tratto.
2. Le classi implicite possono avere solo un parametro di costruttore primario non implicito.
3. Potrebbe non esserci un'altra definizione di oggetto, classe, tratto o membro della classe all'interno dello stesso ambito che ha lo stesso nome della classe implicita.

Examples

Conversione implicita

Una conversione implicita consente al compilatore di convertire automaticamente un oggetto di un tipo in un altro tipo. Ciò consente al codice di trattare un oggetto come un oggetto di un altro tipo.

```
case class Foo(i: Int)

// without the implicit
Foo(40) + 2    // compilation-error (type mismatch)

// defines how to turn a Foo into an Int
implicit def fooToInt(foo: Foo): Int = foo.i

// now the Foo is converted to Int automatically when needed
Foo(40) + 2    // 42
```

La conversione è a senso unico: in questo caso non è possibile convertire `42` indietro a `Foo(42)` . Per fare ciò, è necessario definire una seconda conversione implicita:

```
implicit def intToFoo(i: Int): Foo = Foo(i)
```

Si noti che questo è il meccanismo mediante il quale un valore float può essere aggiunto ad un valore intero, per esempio.

> Le conversioni implicite dovrebbero essere usate con parsimonia perché offuscano ciò che sta accadendo. È consigliabile utilizzare una conversione esplicita tramite una chiamata al metodo, a meno che non si ottenga un guadagno di leggibilità tangibile dall'utilizzo di una conversione implicita.

Non vi è alcun impatto significativo sulle prestazioni delle conversioni implicite.

Scala importa automaticamente una varietà di conversioni implicite in `scala.Predef` , incluse tutte le conversioni da Java a Scala e `scala.Predef` . Questi sono inclusi di default in qualsiasi compilazione di file.

Parametri impliciti

I parametri impliciti possono essere utili se un parametro di un tipo deve essere definito una volta nell'ambito e quindi applicato a tutte le funzioni che utilizzano un valore di quel tipo.

Una normale chiamata di funzione è simile a questa:

```
// import the duration methods
import scala.concurrent.duration._

// a normal method:
def doLongRunningTask(timeout: FiniteDuration): Long = timeout.toMillis

val timeout = 1.second
// timeout: scala.concurrent.duration.FiniteDuration = 1 second

// to call it
doLongRunningTask(timeout) // 1000
```

Ora diciamo che abbiamo alcuni metodi che hanno tutti una durata di timeout e vogliamo chiamare tutti quei metodi che usano lo stesso timeout. Possiamo definire il timeout come variabile implicita.

```
// import the duration methods
import scala.concurrent.duration._

// dummy methods that use the implicit parameter
def doLongRunningTaskA()(implicit timeout: FiniteDuration): Long = timeout.toMillis
def doLongRunningTaskB()(implicit timeout: FiniteDuration): Long = timeout.toMillis

// we define the value timeout as implicit
implicit val timeout: FiniteDuration = 1.second

// we can now call the functions without passing the timeout parameter
doLongRunningTaskA() // 1000
doLongRunningTaskB() // 1000
```

Il modo in cui funziona è che il compilatore scalac cerca un valore nell'ambito che è **contrassegnato come implicito** e il **cui tipo corrisponde a** quello del parametro implicito. Se ne trova uno, lo applicherà come parametro implicito.

> Si noti che questo non funzionerà se si definiscono due o anche più impliciti dello stesso tipo nell'ambito.

Per personalizzare il messaggio di errore, utilizzare l'annotazione `implicitNotFound` sul tipo:

```
@annotation.implicitNotFound(msg = "Select the proper implicit value for type M[${A}]!")
case class M[A](v: A) {}

def usage[O](implicit x: M[O]): O = x.v

//Does not work because no implicit value is present for type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
implicit val first: M[Int] = M(1)
usage[Int]     //Works when `second` is not in scope
implicit val second: M[Int] = M(2)
//Does not work because more than one implicit values are present for the type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
```

Un timeout è un caso d'uso normale per questo, o per esempio in Akka ActorSystem è (il più delle volte) sempre lo stesso, quindi di solito è passato implicitamente. Un altro caso d'uso sarebbe la progettazione di librerie, più comunemente con librerie FP che si basano su typeclass (come scalaz , gatti o estasi).

> Generalmente è considerata una cattiva pratica l'utilizzo di parametri impliciti con tipi di base come *Int* , *Long* , *String* ecc. Poiché creerà confusione e renderà il codice meno leggibile.

Classi implicite

Le classi implicite rendono possibile aggiungere nuovi metodi a classi precedentemente definite.

La classe `String` non ha alcun metodo senza `withoutVowels` . Questo può essere aggiunto in questo modo:

```
object StringUtil {
  implicit class StringEnhancer(str: String) {
    def withoutVowels: String = str.replaceAll("[aeiou]", "")
  }
}
```

La classe implicita ha un singolo parametro costruttore (`str`) con il tipo che si desidera estendere (`String`) e contiene il metodo che si desidera "aggiungere" al tipo (senza `withoutVowels`). I nuovi metodi definiti possono ora essere utilizzati direttamente sul tipo avanzato (quando il tipo avanzato è in ambito implicito):

```
import StringUtil.StringEnhancer // Brings StringEnhancer into implicit scope

println("Hello world".withoutVowels) // Hll wrld
```

Sotto il cofano, le classi implicite definiscono una conversione implicita dal tipo avanzato alla classe implicita, come questa:

```
implicit def toStringEnhancer(str: String): StringEnhancer = new StringEnhancer(str)
```

Le classi implicite sono spesso definite come classi Value per evitare la creazione di oggetti runtime e quindi la rimozione del sovraccarico di runtime:

```
implicit class StringEnhancer(val str: String) extends AnyVal {
    /* conversions code here */
}
```

Con la definizione migliorata sopra, non è necessario creare una nuova istanza di `StringEnhancer` ogni volta che viene richiamato il metodo `withoutVowels` .

Risolvere i parametri impliciti usando 'implicitamente'

Assumendo un elenco di parametri impliciti con più di un parametro implicito:

```
case class Example(p1:String, p2:String)(implicit ctx1:SomeCtx1, ctx2:SomeCtx2)
```

Ora, supponendo che una delle istanze implicite non sia disponibile (`SomeCtx1`) mentre tutte le altre istanze implicite necessarie sono in-scope, per creare un'istanza della classe deve essere fornita un'istanza di `SomeCtx1` .

Questo può essere fatto preservando l'una l'altra istanza implicita nell'ambito usando la parola chiave `implicitly` :

```
Example("something","somethingElse")(new SomeCtx1(), implicitly[SomeCtx2])
```

Implicita nella REPL

Per visualizzare tutti gli `implicits` nell'ambito durante una sessione REPL:

```
scala> :implicits
```

Per includere anche le conversioni implicite definite in `Predef.scala` :

```
scala> :implicits -v
```

Se uno ha un'espressione e desidera visualizzare l'effetto di tutte le regole di riscrittura applicabili ad esso (compresi gli impliciti):

```
scala> reflect.runtime.universe.reify(expr) // No quotes. reify is a macro operating directly
on code.
```

(Esempio:

```
scala> import reflect.runtime.universe._
scala> reify(Array("Alice", "Bob", "Eve").mkString(", "))
resX: Expr[String] = Expr[String](Predef.refArrayOps(Array.apply("Alice", "Bob",
"Eve")(Predef.implicitly)).mkString(", "))
```

)

Leggi impliciti online: https://riptutorial.com/it/scala/topic/1732/impliciti

Capitolo 23: Iniezione di dipendenza

Examples

Cake Pattern con classe di implementazione interna.

```
//create a component that will be injected
trait TimeUtil {
    lazy val timeUtil = new TimeUtilImpl()

    class TimeUtilImpl{
        def now() = new DateTime()
    }
}

//main controller is depended on time util
trait MainController {
    _ : TimeUtil => //inject time util into main controller

    lazy val mainController = new MainControllerImpl()

    class MainControllerImpl {
        def printCurrentTime() = println(timeUtil.now()) //timeUtil is injected from TimeUtil
trait
    }
}

object MainApp extends App {
    object app extends MainController
       with TimeUtil //wire all components

    app.mainController.printCurrentTime()
}
```

Nell'esempio sopra, ho dimostrato come iniettare TimeUtil in MainController.

La sintassi più importante è l'auto-annotazione (`_: TimeUtil =>`) che consiste nell'iniettare `TimeUtil` in `MainController` . In altre parole, `MainController` dipende da `TimeUtil` .

Uso la classe interiore (ad es. `TimeUtilImpl`) in ogni componente perché, a mio parere, è più facile per i test in quanto possiamo `TimeUtilImpl` in giro la classe interiore. Ed è anche più facile tracciare da dove viene chiamato il metodo quando il progetto diventa più complesso.

Infine, collego tutti i componenti. Se hai familiarità con Guice, questo è equivalente a `Binding`

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/scala/topic/5909/iniezione-di-dipendenza

Capitolo 24: Interoperabilità Java

Examples

Conversione di collezioni Scala in raccolte Java e viceversa

Quando è necessario passare una raccolta in un metodo Java:

```
import scala.collection.JavaConverters._

val scalaList = List(1, 2, 3)
JavaLibrary.process(scalaList.asJava)
```

Se il codice Java restituisce una raccolta Java, è possibile trasformarla in una raccolta Scala in un modo simile:

```
import scala.collection.JavaConverters._

val javaCollection = JavaLibrary.getList
val scalaCollection = javaCollection.asScala
```

Si noti che questi sono decoratori, quindi si limitano a racchiudere le raccolte sottostanti in un'interfaccia di raccolta Scala o Java. Pertanto, le chiamate `.asJava` e `.asScala` non copiano le raccolte.

Array

Gli array sono normali array JVM con un tocco che sono trattati come invarianti e hanno costruttori speciali e conversioni implicite. Costruiscile senza la `new` parola chiave.

```
val a = Array("element")
```

Ora `a` ha tipo `Array[String]` .

```
val acs: Array[CharSequence] = a
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
```

Sebbene `String` sia convertibile in `CharSequence` , `Array[String]` non è convertibile in `Array[CharSequence]` .

Puoi utilizzare una `Array` come altre raccolte, grazie a una conversione implicita a `TraversableLike` `ArrayOps` :

```
val b: Array[Int] = a.map(_.length)
```

La maggior parte delle collezioni Scala (`TraversableOnce`) hanno un `toArray` metodo di prendere un

implicito `ClassTag` per costruire la matrice risultato:

```
List(0).toArray
//> res1: Array[Int] = Array(0)
```

Ciò semplifica l'uso di qualsiasi `TraversableOnce` nel tuo codice Scala e poi lo passa al codice Java che si aspetta un array.

Conversioni di tipo Scala e Java

Scala offre conversioni implicite tra tutti i principali tipi di raccolta nell'oggetto JavaConverters.

Le seguenti conversioni di tipi sono bidirezionali.

Tipo Scala	Tipo Java
Iterator	java.util.Iterator
Iterator	java.util.Enumeration
Iterator	java.util.Iterable
Iterator	java.util.Collection
mutable.Buffer	java.util.List
mutable.Set	java.util.Set
mutable.Map	java.util.Map
mutable.ConcurrentMap	java.util.concurrent.ConcurrentMap

Alcune altre collezioni Scala possono anche essere convertite in Java, ma non hanno una conversione al tipo Scala originale:

Tipo Scala	Tipo Java
Seq	java.util.List
mutable.Seq	java.util.List
Impostato	java.util.Set
Carta geografica	java.util.Map

Riferimento :

Conversioni tra le raccolte Java e Scala

Interfacce funzionali per le funzioni di Scala - scala-java8-compat

Un kit di compatibilità Java 8 per Scala.

La maggior parte degli esempi viene copiata da Readme

Convertitori tra scala.FunctionN e java.util.function

```
import java.util.function._
import scala.compat.java8.FunctionConverters._

val foo: Int => Boolean = i => i > 7
def testBig(ip: IntPredicate) = ip.test(9)
println(testBig(foo.asJava))  // Prints true

val bar = new UnaryOperator[String]{ def apply(s: String) = s.reverse }
List("cod", "herring").map(bar.asScala)    // List("doc", "gnirrih")

def testA[A](p: Predicate[A])(a: A) = p.test(a)
println(testA(asJavaPredicate(foo))(4))  // Prints false
```

Convertitori tra scale.Option e le classi java.util Facoltativo, OpzionaleDouble, OpzionaleInt e OpzionaleLungo.

```
import scala.compat.java8.OptionConverters._

    class Test {
      val o = Option(2.7)
      val oj = o.asJava        // Optional[Double]
      val ojd = o.asPrimitive  // OptionalDouble
      val ojds = ojd.asScala   // Option(2.7) again
    }
```

Convertitori da collezioni Scala a Java 8 Stream

```
import java.util.stream.IntStream

import scala.compat.java8.StreamConverters._
import scala.compat.java8.collectionImpl.{Accumulator, LongAccumulator}


  val m = collection.immutable.HashMap("fish" -> 2, "bird" -> 4)
  val parStream: IntStream = m.parValueStream
  val s: Int = parStream.sum
  // 6, potientially computed in parallel
  val t: List[String] = m.seqKeyStream.toScala[List]
  // List("fish", "bird")
  val a: Accumulator[(String, Int)] = m.accumulate // Accumulator[(String, Int)]

  val n = a.stepper.fold(0)(_ + _._1.length) +
    a.parStream.count // 8 + 2 = 10

  val b: LongAccumulator = java.util.Arrays.stream(Array(2L, 3L, 4L)).accumulate
  // LongAccumulator
  val l: List[Long] = b.to[List] // List(2L, 3L, 4L)
```

Leggi Interoperabilità Java online: https://riptutorial.com/it/scala/topic/2441/interoperabilita-java

Capitolo 25: Interpolazione a stringa

Osservazioni

Questa funzione esiste in Scala 2.10.0 e versioni successive.

Examples

Hello String Interpolation

L'interpolatore **s** consente l'utilizzo di variabili all'interno di una stringa.

```
val name = "Brian"
println(s"Hello $name")
```

stampa "Ciao Brian" sulla console quando è in esecuzione.

Interpolazione di stringhe formattate usando l'interpolatore f

```
val num = 42d
```

Stampa due cifre decimali per `num` usando *f*

```
println(f"$num%2.2f")
42.00
```

Stampa `num` usando la notazione scientifica usando *e*

```
println(f"$num%e")
4.200000e+01
```

Stampa `num` in esadecimale con *a*

```
println(f"$num%a")
0x1.5p5
```

Altre stringhe di formato possono essere trovate su
https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

Utilizzo dell'espressione in stringhe letterali

Puoi usare parentesi graffe per interpolare espressioni in stringhe letterali:

```
def f(x: String) = x + x
val a = "A"
```

```
s"${a}"    // "A"
s"${f(a)}" // "AA"
```

Senza le parentesi, scala interpolerebbe solo l' *identificatore* dopo `$` (in questo caso `f`). Poiché non vi è alcuna conversione implicita da `f` a una `String` questa è un'eccezione in questo esempio:

```
s"$f(a)"  // compile-time error (missing argument list for method f)
```

Interpolatori di stringhe personalizzati

È possibile definire interpolatori di stringa personalizzati oltre a quelli incorporati.

```
my"foo${bar}baz"
```

Viene espanso dal compilatore per:

```
new scala.StringContext("foo", "baz").my(bar)
```

`scala.StringContext` non ha il `my` metodo, quindi può essere fornito dalla conversione implicita. Un interpolatore personalizzato con lo stesso comportamento come il comando incorporato `s` interpolatore sarebbe quindi calcolato come segue:

```
implicit class MyInterpolator(sc: StringContext) {
  def my(subs: Any*): String = {
    val pit = sc.parts.iterator
    val sit = subs.iterator
    // Note parts.length == subs.length + 1
    val sb = new java.lang.StringBuilder(pit.next())
    while(sit.hasNext) {
      sb.append(sit.next().toString)
      sb.append(pit.next())
    }
    sb.toString
  }
}
```

E l'interpolazione del `my"foo${bar}baz"` sarebbe stata la seguente:

```
new MyInterpolation(new StringContext("foo", "baz")).my(bar)
```

Si noti che non esiste alcuna restrizione sugli argomenti o il tipo di ritorno della funzione di interpolazione. Questo ci porta lungo un percorso oscuro in cui la sintassi di interpolazione può essere utilizzata in modo creativo per costruire oggetti arbitrari, come illustrato nel seguente esempio:

```
case class Let(name: Char, value: Int)

implicit class LetInterpolator(sc: StringContext) {
  def let(value: Int): Let = Let(sc.parts(0).charAt(0), value)
```

```
}

let"a=${4}" // Let(a, 4)
let"b=${"foo"}" // error: type mismatch
let"c=" // error: not enough arguments for method let: (value: Int)Let
```

Interpolatori a stringa come estrattori

È anche possibile utilizzare la funzione di interpolazione delle stringhe di Scala per creare estrattori elaborati (pattern matcher), come forse il più famoso impiegato nelle API quasiquotes dei macro Scala.

Dato che `n"p0${i0}p1"` desugars a `new StringContext("p0", "p1").n(i0)` , forse non sorprende che la funzionalità estrattore sia abilitata fornendo una conversione implicita da `StringContext` a una classe con struttura `n` di un tipo che definisce una `unapply` o `unapplySeq` metodo.

Ad esempio, si consideri il seguente estrattore che estrae i segmenti del percorso costruendo un'espressione regolare dalle parti `StringContext` . Possiamo quindi delegare la maggior parte del sollevamento pesante al metodo `unapplySeq` fornito dalla scala.util.matching.Regex risultante:

```
implicit class PathExtractor(sc: StringContext) {
  object path {
    def unapplySeq(str: String): Option[Seq[String]] =
      sc.parts.map(Regex.quote).mkString("^", "([^/]+)", "$").r.unapplySeq(str)
  }
}

"/documentation/scala/1629/string-interpolation" match {
  case path"/documentation/${topic}/${id}/${_}" => println(s"$topic, $id")
  case _ => ???
}
```

Si noti che l'oggetto `path` potrebbe anche definire un metodo `apply` per comportarsi come un normale interpolatore.

Interpolazione di stringa grezza

È possibile utilizzare l'interpolatore **raw** se si desidera che una stringa venga stampata così com'è e senza alcuna escape di valori letterali.

```
println(raw"Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Con l'uso dell'interpolatore **raw** , dovresti vedere quanto segue stampato nella console:

```
Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde
```

Senza l'interpolatore **raw** , `\n` e `\t` sarebbero stati sfuggiti.

```
println("Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

stampe:

```
Hello World In English And French
English:       Hello World
French:        Bonjour Le Monde
```

Leggi Interpolazione a stringa online: https://riptutorial.com/it/scala/topic/1629/interpolazione-a-stringa

Capitolo 26: Invocazione Dinamica

introduzione

Scala consente di utilizzare l'invocazione dinamica quando si chiamano metodi o si accede a campi su un oggetto. Invece di avere questo costruito in profondità nel linguaggio, questo si ottiene attraverso regole di riscrittura simili a quelle delle conversioni implicite, abilitate dal tratto marker [`scala.Dynamic`] [Scaladoc dinamico]. Ciò consente di emulare la capacità di aggiungere dinamicamente proprietà a oggetti presenti in linguaggi dinamici e altro ancora. [Scaladoc dinamico]: http://www.scala-lang.org/api/2.12.x/scala/Dynamic.html

Sintassi

- la classe Foo estende la dinamica
- foo.field
- foo.field = valore
- foo.method (args)
- foo.method (namedArg = x, y)

Osservazioni

Per dichiarare i sottotipi di `Dynamic` , è necessario abilitare le `dynamics` caratteristiche del linguaggio importando `scala.language.dynamics` o l' `-language:dynamics` compiler. Gli utenti di questa `Dynamic` che non definiscono i propri sottotipi non devono abilitarlo.

Examples

Campo di accesso

Questo:

```
class Foo extends Dynamic {
  // Expressions are only rewritten to use Dynamic if they are not already valid
  // Therefore foo.realField will not use select/updateDynamic
  var realField: Int = 5
  // Called for expressions of the type foo.field
  def selectDynamic(fieldName: String) = ???
  def updateDynamic(fieldName: String)(value: Int) = ???
}
```

consente un accesso semplice ai campi:

```
val foo: Foo = ???
foo.realField // Does NOT use Dynamic; accesses the actual field
foo.realField = 10 // Actual field access here too
foo.unrealField // Becomes foo.selectDynamic(unrealField)
```

```
foo.field = 10  // Becomes foo.updateDynamic("field")(10)
foo.field = "10" // Does not compile; "10" is not an Int.
foo.x() // Does not compile; Foo does not define applyDynamic, which is used for methods.
foo.x.apply() // DOES compile, as Nothing is a subtype of () => Any
// Remember, the compiler is still doing static type checks, it just has one more way to
// "recover" and rewrite otherwise invalid code now.
```

Metodo chiamate

Questo:

```
class Villain(val minions: Map[String, Minion]) extends Dynamic {
  def applyDynamic(name: String)(jobs: Task*) = jobs.foreach(minions(name).do)
  def applyDynamicNamed(name: String)(jobs: (String, Task)*) = jobs.foreach {
    // If a parameter does not have a name, and is simply given, the name passed as ""
    case ("", task) => minions(name).do(task)
    case (subsys, task) => minions(name).subsystems(subsys).do(task)
  }
}
```

consente le chiamate ai metodi, con e senza parametri denominati:

```
val gru: Villain = ???
gru.blu() // Becomes gru.applyDynamic("blu")()
// Becomes gru.applyDynamicNamed("stu")(("fooer", ???), ("boomer", ???), ("", ???),
//          ("computer breaker", ???), ("fooer", ???))
// Note how the `???` without a name is given the name ""
// Note how both occurrences of `fooer` are passed to the method
gru.stu(fooer = ???, boomer = ???, ???, `computer breaker` = ???, fooer = ???)
gru.ERR("a") // Somehow, scalac thinks "a" is not a Task, though it clearly is (it isn't)
```

Interazione tra accesso al campo e metodo di aggiornamento

Leggermente controintuitivamente (ma anche l'unico modo corretto per farlo funzionare), questo:

```
val dyn: Dynamic = ???
dyn.x(y) = z
```

è equivalente a:

```
dyn.selectDynamic("x").update(y, z)
```

mentre

```
dyn.x(y)
```

è ancora

```
dyn.applyDynamic("x")(y)
```

È importante essere consapevoli di questo, altrimenti potrebbe essere nascosto di nascosto e

causare strani errori.

Leggi Invocazione Dinamica online: https://riptutorial.com/it/scala/topic/8296/invocazione-dinamica

Capitolo 27: JSON

Examples

JSON con spray-json

spray-json offre un modo semplice per lavorare con JSON. Utilizzando formati impliciti, tutto accade "dietro le quinte":

Rendi disponibile la libreria con SBT

Per gestire `spray-json` con dipendenze della libreria gestita SBT :

```
libraryDependencies += "io.spray" %% "spray-json" % "1.3.2"
```

Si noti che l'ultimo parametro, il numero di versione (`1.3.2`), potrebbe essere diverso in diversi progetti.

La libreria `spray-json` è ospitata su repo.spray.io .

Importa la libreria

```
import spray.json._
import DefaultJsonProtocol._
```

Il protocollo JSON predefinito `DefaultJsonProtocol` contiene i formati per tutti i tipi di base. Per fornire funzionalità JSON per i tipi personalizzati, utilizzare i builder di convenienza per formati o formati di scrittura in modo esplicito.

Leggi JSON

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = """{ "foo": "bar" }""".parseJson // JsValue = {"foo":"bar"}

res.convertTo[Map[String, String]] // Map(foo -> bar)
```

Scrivi JSON

```
val values = List("a", "b", "c")
values.toJson.prettyPrint // ["a", "b", "c"]
```

DSL

DSL non è supportato.

Read-Write in Case Classes

L'esempio seguente mostra come serializzare un oggetto classe case nel formato JSON.

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = jsonFormat2(Address)
implicit val personFormat = jsonFormat2(Person)

// serialize a Person
Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = fred.toJson.prettyPrint
```

Ciò risulta nel seguente JSON:

```
{
  "name": "Fred",
  "address": {
    "street": "Awesome Street 9",
    "city": "SuperCity"
  }
}
```

Quella JSON può, a sua volta, essere deserializzata di nuovo in un oggetto:

```
val personRead = fredJsonString.parseJson.convertTo[Person]
//Person(Fred,Address(Awesome Street 9,SuperCity))
```

Formato personalizzato

Scrivi un `JsonFormat` personalizzato se è richiesta una serializzazione speciale di un tipo. Ad esempio, se i nomi dei campi sono diversi in Scala rispetto a JSON. Oppure, se diversi tipi di calcestruzzo sono istanziati in base all'input.

```
implicit object BetterPersonFormat extends JsonFormat[Person] {
    // deserialization code
    override def read(json: JsValue): Person = {
        val fields = json.asJsObject("Person object expected").fields
        Person(
            name = fields("name").convertTo[String],
            address = fields("home").convertTo[Address]
        )
    }
```

```
    // serialization code
    override def write(person: Person): JsValue = JsObject(
        "name" -> person.name.toJson,
        "home" -> person.address.toJson
    )
}
```

JSON con Circe

Circe fornisce codec derivati in fase di compilazione per en / decode json in classi case. Un semplice esempio si presenta così:

```
import io.circe._
import io.circe.generic.auto._
import io.circe.parser._
import io.circe.syntax._

case class User(id: Long, name: String)

val user = User(1, "John Doe")

// {"id":1,"name":"John Doe"}
val json = user.asJson.noSpaces

// Right(User(1L, "John Doe"))
val res: Either[Error, User] = decode[User](json)
```

JSON con il gioco di parole

play-json utilizza formati impliciti come altri framework JSON

Dipendenza SBT: `libraryDependencies += ""com.typesafe.play" %% "play-json" % "2.4.8"`

```
import play.api.libs.json._
import play.api.libs.functional.syntax._ // if you need DSL
```

`DefaultFormat` contiene i formati defaul per leggere / scrivere tutti i tipi di base. Per fornire funzionalità JSON per i tuoi tipi, puoi utilizzare i builder di convenienza per formati o formati di scrittura in modo esplicito.

Leggi JSON

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = Json.parse("""{ "foo": "bar" }""") // JsValue = {"foo":"bar"}

res.as[Map[String, String]]                  // Map(foo -> bar)
res.validate[Map[String, String]]            //JsSuccess(Map(foo -> bar),)
```

Scrivi JSON

```
val values = List("a", "b", "c")
```

```
Json.stringify(Json.toJson(values))          // ["a", "b", "c"]
```

DSL

```
val json = parse("""{ "foo": [{"foo": "bar"}]}""")
(json \ "foo").get                    //Simple path: [{"foo":"bar"}]
(json \\ "foo")                       //Recursive path:List([{"foo":"bar"}], "bar")
(json \ "foo")(0).get                  //Index lookup (for JsArrays): {"foo":"bar"}
```

Preferisce come sempre pattern matching contro `JsSuccess` / `JsError` e cercare di evitare `.get` , `array(i)` chiama.

Leggi e scrivi su case class

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = Json.format[Address]
implicit val personFormat = Json.format[Person]

// serialize a Person
val fred = Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = Json.stringify(Json.toJson(Json.toJson(fred)))

val personRead = Json.parse(fredJsonString).as[Person] //Person(Fred,Address(Awesome Street
9,SuperCity))
```

Proprio formato

Puoi scrivere il tuo JsonFormat se hai bisogno di una serializzazione speciale del tuo tipo (ad esempio, dai nomi diversi ai campi in scala e Json o istanziati diversi tipi di calcestruzzo in base all'input)

```
case class Address(street: String, city: String)

// create the formats and provide them implicitly
implicit object AddressFormatCustom extends Format[Address] {
  def reads(json: JsValue): JsResult[Address] = for {
    street <- (json \ "Street").validate[String]
    city <- (json \ "City").validate[String]
  } yield Address(street, city)

  def writes(x: Address): JsValue = Json.obj(
    "Street" -> x.street,
    "City" -> x.city
  )
}
// serialize an address
val address = Address("Awesome Street 9", "SuperCity")
val addressJsonString = Json.stringify(Json.toJson(Json.toJson(address)))
//{"Street":"Awesome Street 9","City":"SuperCity"}

val addressRead = Json.parse(addressJsonString).as[Address]
//Address(Awesome Street 9,SuperCity)
```

Alternativa

Se il json non corrisponde esattamente ai campi della classe case (`isAlive` in caso di classe vs `is_alive` in json):

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Boolean)

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").read[Boolean]
    ) (User.apply _)
}
```

Json con campi opzionali

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Option[Boolean])

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").readNullable[Boolean]
    ) (User.apply _)
}
```

Leggere i timestamp di JSON

Immagina di avere un oggetto Json, con un campo timestamp Unix:

```
{
  "field": "example field",
  "date": 1459014762000
}
```

soluzione:

```
case class JsonExampleV1(field: String, date: DateTime)
object JsonExampleV1{
  implicit val r: Reads[JsonExampleV1] = (
    (__ \ "field").read[String] and
      (__ \ "date").read[DateTime](Reads.DefaultJodaDateReads)
    )(JsonExampleV1.apply _)
}
```

Leggere classi caso personalizzate

Ora, se avvolgi i tuoi identificatori di oggetto per la sicurezza del tipo, questo ti piacerà. Vedi il seguente oggetto JSON:

```
{
  "id": 91,
  "data": "Some data"
}
```

e le classi di casi corrispondenti:

```
case class MyIdentifier(id: Long)

case class JsonExampleV2(id: MyIdentifier, data: String)
```

Ora devi solo leggere il tipo primitivo (Long) e mappare il tuo idenfier:

```
object JsonExampleV2 {
  implicit val r: Reads[JsonExampleV2] = (
      (__ \ "id").read[Long].map(MyIdentifier) and
    (__ \ "data").read[String]
    )(JsonExampleV2.apply _)
}
```

codice all'indirizzo https://github.com/pedrorijo91/scala-play-json-examples

JSON con json4s

json4s usa formati impliciti come altri framework json.

Dipendenza SBT:

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.4.0"
//or
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.4.0"
```

importazioni

```
import org.json4s.JsonDSL._
import org.json4s._
import org.json4s.native.JsonMethods._

implicit val formats = DefaultFormats
```

`DefaultFormats` contiene i formati predefiniti per leggere / scrivere tutti i tipi di base.

Leggi JSON

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = parse("""{ "foo": "bar" }""")           // JValue = {"foo":"bar"}
```

```
res.extract[Map[String, String]]                  // Map(foo -> bar)
```

Scrivi JSON

```
val values = List("a", "b", "c")
compact(render(values))         // ["a", "b", "c"]
```

DSL

```
json \ "foo"       //Simple path: JArray(List(JObject(List((foo,JString(bar))))))
json \\ "foo"      //Recursive path: ~List([{"foo":"bar"}], "bar")
(json \ "foo")(0)  //Index lookup (for JsArrays): JObject(List((foo,JString(bar))))
("foo" -> "bar") ~ ("field" -> "value")    // {"foo":"bar","field":"value"}
```

Leggi e scrivi su case class

```
import org.json4s.native.Serialization.{read, write}

case class Address(street: String, city: String)
val addressString = write(Address("Awesome stree", "Super city"))
// {"street":"Awesome stree","city":"Super city"}

read[Address](addressString) // Address(Awesome stree,Super city)
//or
parse(addressString).extract[Address]
```

Leggere e scrivere liste eterogenee

Per serializzare e deserializzare un elenco eterogeneo (o polimorfico), è necessario fornire specifici suggerimenti di tipo.

```
trait Location
case class Street(name: String) extends Location
case class City(name: String, zipcode: String) extends Location
case class Address(street: Street, city: City) extends Location
case class Locations (locations : List[Location])

implicit val formats = Serialization.formats(ShortTypeHints(List(classOf[Street],
classOf[City], classOf[Address])))

val locationsString = write(Locations(Street("Lavelle Street"):: City("Super city","74658")))

read[Locations](locationsString)
```

Proprio formato

```
class AddressSerializer extends CustomSerializer[Address](format => (
  {
    case JObject(JField("Street", JString(s)) :: JField("City", JString(c)) :: Nil) =>
      new Address(s, c)
  },
  {
    case x: Address => ("Street" -> x.street) ~ ("City" -> x.city)
  }
```

```
  ))

implicit val formats = DefaultFormats + new AddressSerializer
val str = write[Address](Address("Awesome Stree", "Super City"))
// {"Street":"Awesome Stree","City":"Super City"}
read[Address](str)
// Address(Awesome Stree,Super City)
```

Leggi JSON online: https://riptutorial.com/it/scala/topic/2348/json

Capitolo 28: Lavorare con Gradle

Examples

Impostazione di base

1. Crea un file chiamato `SCALA_PROJECT/build.gradle` con questi contenuti:

```
group 'scala_gradle'
version '1.0-SNAPSHOT'

apply plugin: 'scala'

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "https://repo.typesafe.com/typesafe/maven-releases"
    }
}

dependencies {
    compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.6'
}


task  "create-dirs" << {
    sourceSets*.scala.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each { it.mkdirs() }
}
```

2. Esegui `gradle tasks` per vedere le attività disponibili.

3. Esegui `gradle create-dirs` per creare una `gradle create-dirs src/scala, src/resources .`

4. Esegui `gradle build` per creare il progetto e scaricare le dipendenze.

Crea il tuo plugin Gradle Scala

Dopo aver esaminato l'esempio di **installazione di base** , potresti trovarti a ripetere la maggior parte di esso in ogni singolo progetto Scala Gradle. Odora come codice boilerplate ...

Cosa succede se, invece di applicare il plugin Scala offerto da Gradle, è possibile applicare il proprio plugin Scala, che sarebbe responsabile della gestione di tutte le logiche di build comuni, estendendo, allo stesso tempo, il plug-in già esistente.

Questo esempio trasformerà la precedente build logic in un plugin Gradle riutilizzabile.

Fortunatamente, in Gradle, puoi facilmente scrivere plug-in personalizzati con l'aiuto dell'API Gradle, come indicato nella documentazione . Come linguaggio di implementazione, puoi usare lo

stesso Scala o anche Java. Tuttavia, la maggior parte degli esempi che puoi trovare in tutti i documenti sono scritti in Groovy. Se hai bisogno di più esempi di codice o vuoi capire cosa si nasconde dietro il plugin Scala, ad esempio, puoi controllare il repository gradle github .

Scrivere il plugin

Requisiti

Il plug-in personalizzato aggiungerà le seguenti funzionalità quando applicato a un progetto:

- un oggetto di proprietà `scalaVersion` , che avrà due proprietà predefinite di override
  - major = "2.12"
  - minor = "0"
- una funzione `withScalaVersion` , che si applica a un nome di dipendenza, aggiungerà la versione scala maggiore per garantire la compatibilità binaria (l'operatore sbt `%%` potrebbe suonare un campanello, altrimenti vai qui prima di procedere)
- un'attività `createDirs` per creare l'albero delle directory necessario, esattamente come nell'esempio precedente

Linea guida di implementazione

1. crea un nuovo progetto gradle e aggiungi quanto segue a `build.gradle`

```
apply plugin: 'scala'
apply plugin: 'maven'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile gradleApi()
    compile "org.scala-lang:scala-library:2.12.0"
}
```

Note :

- l'implementazione del plugin è scritta in Scala, quindi abbiamo bisogno del plugin Gradle Scala
- per utilizzare il plugin da altri progetti, viene utilizzato il plugin Gradle Maven; questo aggiunge l'attività di `install` utilizzata per salvare il jar del progetto nell'archivio locale Maven
- `compile gradleApi()` aggiunge il `gradle-api-<gradle_version>.jar` al classpath

2. crea una nuova classe di Scala per l'implementazione del plugin

```
package com.btesila.gradle.plugins

import org.gradle.api.{Plugin, Project}
```

```
class ScalaCustomPlugin extends Plugin[Project] {
    override def apply(project: Project): Unit = {
        project.getPlugins.apply("scala")
    }
}
```

Note :

- per implementare un plugin, estendere semplicemente il tratto del `Plugin` di tipo `Project` e sovrascrivere il metodo `apply`
- all'interno del metodo apply, si ha accesso all'istanza di `Project` cui è applicato il plugin e si può usarlo per aggiungere la build logic ad esso
- questo plugin non fa altro che applicare il plugin Gradle Scala già esistente

3. aggiungi la proprietà dell'oggetto `scalaVersion`

In primo luogo, creiamo una classe `ScalaVersion` , che conterrà le due proprietà della versione

```
class ScalaVersion {
  var major: String = "2.12"
  var minor: String = "0"
}
```

Una cosa interessante dei plugin Gradle è il fatto che puoi sempre aggiungere o sovrascrivere proprietà specifiche. Un plugin riceve questo tipo di input utente tramite `ExtensionContainer` collegato a un'istanza di `Project` gradle. Per maggiori dettagli, controlla questo .
Aggiungendo il seguente al metodo `apply` , stiamo praticamente facendo questo:

- se non esiste una proprietà `scalaVersion` definita nel progetto, ne aggiungiamo una con i valori predefiniti
- altrimenti, otteniamo quello esistente come istanza di `ScalaVersion` , per usarlo ulteriormente

```
var scalaVersion = new ScalaVersion
if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
   project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
else
   scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]
```

Ciò equivale a scrivere quanto segue nel file di build del progetto che applica il plugin:

```
ext {
    scalaVersion.major = "2.12"
    scalaVersion.minor = "0"

}
```

4. aggiungi la libreria `scala-lang` alle dipendenze del progetto, usando lo `scalaVersion`

```
project.getDependencies.add("compile", s"org.scala-lang:scala-
```

```
library:${scalaVersion.major}.${scalaVersion.minor}")
```

Ciò equivale a scrivere quanto segue nel file di build del progetto che applica il plugin:

```
compile "org.scala-lang:scala-library:2.12.0"
```

5. aggiungere la funzione `withScalaVersion`

```
val withScalaVersion = (lib: String) => {
    val libComp = lib.split(":")
    libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
    libComp.mkString(":")
}
project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)
```

6. infine, creare l'attività `createDirs` e aggiungerla al progetto
   Implementare un'attività Gradle estendendo `DefaultTask` :

```
class CreateDirs extends DefaultTask {
  @TaskAction
  def createDirs(): Unit = {
    val sourceSetContainer =
this.getProject.getConvention.getPlugin(classOf[JavaPluginConvention]).getSourceSets

    sourceSetContainer forEach { sourceSet =>
      sourceSet.getAllSource.getSrcDirs.forEach(file => if (!file.getName.contains("java"))
file.mkdirs())
    }
  }
}
```

Nota : `SourceSetContainer` contiene informazioni su tutte le directory di origine presenti nel progetto. Quello che fa il plugin Gradle Scala è aggiungere i set sorgente extra a quelli Java, come puoi vedere nei documenti del plugin .

Aggiungere l'attività `createDir` al progetto aggiungendo questo al metodo `apply` :

```
project.getTasks.create("createDirs", classOf[CreateDirs])
```

Alla fine, la tua classe `ScalaCustomPlugin` dovrebbe apparire così:

```
class ScalaCustomPlugin extends Plugin[Project] {
  override def apply(project: Project): Unit = {
    project.getPlugins.apply("scala")

    var scalaVersion = new ScalaVersion
    if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
      project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
    else
      scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]
```

```
    project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")

    val withScalaVersion = (lib: String) => {
      val libComp = lib.split(":")
      libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
      libComp.mkString(":")
    }
    project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)

    project.getTasks.create("createDirs", classOf[CreateDirs])
  }
}
```

Installazione del progetto plugin sul repository Maven locale

Questo è veramente facile eseguendo `gradle install`
È possibile verificare l'installazione andando nella directory del repository locale, di solito in `~/.m2/repository`

Come fa Gradle a trovare il nostro nuovo plugin?

Ogni plugin Gradle ha un `id` che viene utilizzato nell'istruzione `apply` . Ad esempio, scrivendo quanto segue nel file di build, si traduce in un trigger di Gradle per trovare e applicare il plug-in con id `scala` .

```
apply plugin: 'scala'
```

Allo stesso modo, vorremmo applicare il nostro nuovo plugin nel modo seguente,

```
apply plugin: "com.btesila.scala.plugin"
```

il che significa che il nostro plugin avrà l'ID `com.btesila.scala.plugin` .

Per impostare questo ID, aggiungi il seguente file:
src / main / resources / META-INF / Gradle-plugin / com.btesil.scala.plugin.properties

```
implementation-class=com.btesila.gradle.plugins.ScalaCustomPlugin
```

Successivamente, esegui di nuovo `gradle install` .

Utilizzando il plugin

1. crea un nuovo progetto Gradle vuoto e aggiungi il seguente al file di build

```
buildscript {
    repositories {
        mavenLocal()
```

```
        mavenCentral()
    }

    dependencies {
        //modify this path to match the installed plugin project in your local repository
        classpath 'com.btesila:working-with-gradle:1.0-SNAPSHOT'
    }
}

repositories {
    mavenLocal()
    mavenCentral()
}


apply plugin: "com.btesila.scala.plugin"
```

2. run `gradle createDirs` - ora dovresti avere tutte le directory sorgente generate
3. sostituisci la versione di scala aggiungendo questo al file di build:

```
ext {
    scalaVersion.major = "2.11"
    scalaVersion.minor = "8"

}
println(project.ext.scalaVersion.major)
println(project.ext.scalaVersion.minor)
```

4. aggiungere una libreria di dipendenze compatibile con binario con la versione di Scala

```
dependencies {
    compile withScalaVersion("com.typesafe.scala-logging:scala-logging:3.5.0")
}
```

Questo è tutto! Ora puoi usare questo plugin su tutti i tuoi progetti senza ripetere lo stesso vecchio standard.

Leggi Lavorare con Gradle online: https://riptutorial.com/it/scala/topic/3304/lavorare-con-gradle

Capitolo 29: Lavorare con i dati in uno stile immutabile

Osservazioni

Il valore e i nomi delle variabili dovrebbero essere nel caso di cammello inferiore

> I nomi costanti dovrebbero essere nella custodia del cammello superiore. Cioè, se il membro è definitivo, immutabile e appartiene a un oggetto pacchetto o un oggetto, può essere considerato una costante
>
> Il metodo, il valore e i nomi delle variabili dovrebbero essere in un caso di cammello inferiore

Fonte: http://docs.scala-lang.org/style/naming-conventions.html

Questo compila:

```
val (a,b) = (1,2)
// a: Int = 1
// b: Int = 2
```

ma questo non:

```
val (A,B) = (1,2)
// error: not found: value A
// error: not found: value B
```

Examples

Non è solo val vs. var

`val` e `var`

```
scala> val a = 123
a: Int = 123

scala> a = 456
<console>:8: error: reassignment to val
       a = 456

scala> var b = 123
b: Int = 123

scala> b = 321
```

```
b: Int = 321
```

- `val` riferimenti `val` sono immutabili: come una variabile `final` in `Java` , una volta inizializzata non puoi cambiarla
- `var` riferimenti `var` sono riassegnabili come una semplice dichiarazione di variabile in Java

Collezioni immutabili e mutevoli

```
  val mut = scala.collection.mutable.Map.empty[String, Int]
  mut += ("123" -> 123)
  mut += ("456" -> 456)
  mut += ("789" -> 789)

  val imm = scala.collection.immutable.Map.empty[String, Int]
  imm + ("123" -> 123)
  imm + ("456" -> 456)
  imm + ("789" -> 789)

  scala> mut
    Map(123 -> 123, 456 -> 456, 789 -> 789)

  scala> imm
    Map()

scala> imm + ("123" -> 123) + ("456" -> 456) + ("789" -> 789)
    Map(123 -> 123, 456 -> 456, 789 -> 789)
```

La libreria standard di Scala offre sia strutture di dati immutabili che mutevoli, non il riferimento ad esso. Ogni volta che una struttura di dati immutabile viene "modificata", viene prodotta una nuova istanza invece di modificare la raccolta originale sul posto. Ogni istanza della raccolta può condividere una struttura significativa con un'altra istanza.

Collezione Mutevole e Immutabile (Documentazione ufficiale di Scala)

Ma non posso usare l'immutabilità in questo caso!

Prendiamo come esempio una funzione che richiede 2 `Map` e restituisce una `Map` contenente ogni elemento in `ma` e `mb` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int]
```

Un primo tentativo potrebbe iterare attraverso gli elementi di una delle mappe usando `for ((k, v) <- map)` e in qualche modo restituire la mappa unita.

```
def merge2Maps(ma: ..., mb: ...): Map[String, Int] = {

  for ((k, v) <- mb) {
    ???
  }

}
```

Questa primissima mossa aggiunge immediatamente un vincolo: **una mutazione al di fuori di quella `for` ora è *necessario*** . Questo è più chiaro quando si deseleziona il `for` :

```
// this:
for ((k, v) <- map) { ??? }

// is equivalent to:
map.foreach { case (k, v) => ??? }
```

"Perché dobbiamo mutare?"

`foreach` si basa su effetti collaterali. Ogni volta che vogliamo che qualcosa succeda all'interno di un `foreach` abbiamo bisogno di "side-effect something", in questo caso potremmo mutare un `var result` variabile `var result` o possiamo usare una struttura dati mutabile.

Creazione e compilazione della mappa dei `result`

Supponiamo che la `ma` e `mb` siano `scala.collection.immutable.Map` , potremmo creare la mappa dei `result` da `ma` :

```
val result = mutable.Map() ++ ma
```

Quindi iterate tramite `mb` aggiungendo i suoi elementi e se la `key` dell'elemento corrente su `ma` esiste già, sovrascriviamo quella di `mb` .

```
mb.foreach { case (k, v) => result += (k -> v) }
```

Implementazione mutevole

Fin qui tutto bene, "abbiamo dovuto usare collezioni mutevoli" e una corretta implementazione potrebbe essere:

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  val result = scala.collection.mutable.Map() ++ ma
  mb.foreach { case (k, v) => result += (k -> v) }
  result.toMap // to get back an immutable Map
}
```

Come previsto:

```
scala> merge2Maps(Map("a" -> 11, "b" -> 12), Map("b" -> 22, "c" -> 23))
  Map(a -> 11, b -> 22, c -> 23)
```

Piegando in soccorso

Come possiamo sbarazzarci di `foreach` in questo scenario? Se tutto ciò che dobbiamo fare è fondamentalmente scorrere gli elementi della raccolta e applicare una funzione mentre si accumula il risultato sull'opzione potrebbe essere `.foldLeft` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  mb.foldLeft(ma) { case (result, (k, v)) => result + (k -> v) }
  // or more concisely mb.foldLeft(ma) { _ + _ }
}
```

In questo caso il nostro "risultato" è il valore accumulato a partire da `ma` , lo `zero` della `.foldLeft` .

Risultato intermedio

Ovviamente questa soluzione immutabile sta producendo e distruggendo molte istanze di `Map` durante la piegatura, ma vale la pena ricordare che quelle istanze non sono un clone completo della `Map` accumulato ma che invece condividono una struttura significativa (dati) con l'istanza esistente.

Più ragionevole la ragionevolezza

È più facile ragionare sulla semantica se è più dichiarativa come l'approccio `.foldLeft` . L'utilizzo di strutture di dati immutabili potrebbe aiutare a rendere più facile ragionare la nostra implementazione.

Leggi Lavorare con i dati in uno stile immutabile online:
https://riptutorial.com/it/scala/topic/6298/lavorare-con-i-dati-in-uno-stile-immutabile

Capitolo 30: Le tuple

Osservazioni

Perché le tuple sono limitate alla lunghezza 23?

Le tuple vengono riscritte come oggetti dal compilatore. Il compilatore ha accesso a `Tuple1` attraverso `Tuple22` . Questo limite arbitrario è stato deciso dai progettisti di linguaggi.

Perché le lunghezze di tuple contano da 0?

Una `Tuple0` è equivalente a `Unit` .

Examples

Creazione di una nuova tupla

Una tupla è una raccolta eterogenea di valori compresi tra due e ventidue. Una tupla può essere definita usando le parentesi. Per le tuple di dimensione `2` (anche chiamata 'coppia') c'è una sintassi delle frecce.

```
scala> val x = (1, "hello")
x: (Int, String) = (1,hello)
scala> val y = 2 -> "world"
y: (Int, String) = (2,world)
scala> val z = 3 → "foo"     //example of using U+2192 RIGHTWARD ARROW
z: (Int, String) = (3,foo)
```

`x` è una tupla di dimensione due. Per accedere agli elementi di una tupla usare `._1` , attraverso `._22` . Ad esempio, possiamo usare `x._1` per accedere al primo elemento della tupla `x` . `x._2` accede al secondo elemento. Più elegantemente, puoi usare gli estrattori di tuple .

La sintassi della freccia per la creazione di tuple di dimensione due viene principalmente utilizzata in Maps, che sono raccolte di coppie `(key -> value)` :

```
scala> val m = Map[Int, String](2 -> "world")
m: scala.collection.immutable.Map[Int,String] = Map(2 -> world)

scala> m + x
res0: scala.collection.immutable.Map[Int,String] = Map(2 -> world, 1 -> hello)

scala> (m + x).toList
res1: List[(Int, String)] = List((2,world), (1,hello))
```

La sintassi per la coppia nella mappa è la sintassi della freccia, rendendo chiaro che 1 è la chiave e a è il valore associato a quella chiave.

Tuple all'interno delle collezioni

Le tuple sono spesso utilizzate all'interno delle raccolte ma devono essere gestite in un modo specifico. Ad esempio, dato il seguente elenco di tuple:

```
scala> val l = List(1 -> 2, 2 -> 3, 3 -> 4)
l: List[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Può sembrare naturale aggiungere gli elementi insieme usando la tupla-disimballaggio implicita:

```
scala> l.map((e1: Int, e2: Int) => e1 + e2)
```

Tuttavia, ciò comporta il seguente errore:

```
<console>:9: error: type mismatch;
 found   : (Int, Int) => Int
 required: ((Int, Int)) => ?
              l.map((e1: Int, e2: Int) => e1 + e2)
```

Scala non può decomprimere implicitamente le tuple in questo modo. Abbiamo due opzioni per correggere questa mappa. Il primo è usare gli accessori posizionali `_1` e `_2` :

```
scala> l.map(e => e._1 + e._2)
res1: List[Int] = List(3, 5, 7)
```

L'altra opzione è usare un'istruzione `case` per decomprimere le tuple usando la corrispondenza del modello:

```
scala> l.map{ case (e1: Int, e2: Int) => e1 + e2}
res2: List[Int] = List(3, 5, 7)
```

Queste restrizioni si applicano a qualsiasi funzione di ordine superiore applicata a una raccolta di tuple.

Leggi Le tuple online: https://riptutorial.com/it/scala/topic/4971/le-tuple

Capitolo 31: Libreria di continuazioni

introduzione

Lo stile di passaggio continuo è una forma di flusso di controllo che implica il passaggio alle funzioni del resto del calcolo come argomento di "continuazione". La funzione in questione richiama in seguito tale continuazione per continuare l'esecuzione del programma. Un modo per pensare a una continuazione è come una chiusura. La libreria di continuazioni Scala porta continuazioni delimitate nella forma dei primitivi che si `shift` / `reset` al linguaggio.

libreria di continuazioni: https://github.com/scala/scala-continuations

Sintassi

- reset {...} // Le continuazioni si estendono fino alla fine del blocco di reset che lo racchiude
- shift {...} // Crea una continuazione che affermi dopo la chiamata, passandola alla chiusura
- A @cpsParam [B, C] // Un calcolo che richiede una funzione A => B per creare un valore di C
- @cps [A] // Alias per @cpsParam [A, A]
- @suspendable // Alias per @cpsParam [Unit, Unit]

Osservazioni

`shift` e `reset` sono strutture di flusso di controllo primitive, come `Int.+` è un'operazione primitiva e `Long` è un tipo primitivo. Sono più primitivi di quanto in quelle continuazioni delimitate possano essere effettivamente usati per costruire quasi tutte le strutture di flusso di controllo. Non sono molto utili "out-of-the-box", ma brillano davvero quando vengono utilizzati nelle librerie per creare API ricche.

Continuazioni e monadi sono anche strettamente collegate. Le continuazioni possono essere fatte nel monad di continuazione e le monadi sono continuazioni perché la loro operazione `flatMap` prende una continuazione come parametro.

Examples

Le callback sono continui

```
// Takes a callback and executes it with the read value
def readFile(path: String)(callback: Try[String] => Unit): Unit = ???

readFile(path) { _.flatMap { file1 =>
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}
```

L'argomento della funzione su `readFile` è una continuazione, in quanto `readFile` richiama per continuare l'esecuzione del programma dopo che ha svolto il suo lavoro.

Al fine di contenere ciò che può facilmente diventare un inferno di callback, usiamo la libreria di continuazioni.

```
reset { // Reset is a delimiter for continuations.
  for { // Since the callback hell is relegated to continuation library machinery.
        // a for-comprehension can be used
    file1 <- shift(readFile(path1)) // shift has type (((A => B) => C) => A)
    // We use it as (((Try[String] => Unit) => Unit) => Try[String])
    // It takes all the code that occurs after it is called, up to the end of reset, and
    // makes it into a closure of type (A => B).
    // The reason this works is that shift is actually faking its return type.
    // It only pretends to return A.
    // It actually passes that closure into its function parameter (readFile(path1) here),
    // And that function calls what it thinks is a normal callback with an A.
    // And through compiler magic shift "injects" that A into its own callsite.
    // So if readFile calls its callback with parameter Success("OK"),
    // the shift is replaced with that value and the code is executed until the end of reset,
    // and the return value of that is what the callback in readFile returns.
    // If readFile called its callback twice, then the shift would run this code twice too.
    // Since readFile returns Unit though, the type of the entire reset expression is Unit
    //
    // Think of shift as shifting all the code after it into a closure,
    // and reset as resetting all those shifts and ending the closures.
    file2 <- shift(readFile(path2))
  } processFiles(file1, file2)
}

// After compilation, shift and reset are transformed back into closures
// The for comprehension first desugars to:
reset {
  shift(readFile(path1)).flatMap { file1 => shift(readFile(path2)).foreach { file2 =>
processFiles(file1, file2) } }
}
// And then the callbacks are restored via CPS transformation
readFile(path1) { _.flatMap { file1 => // We see how shift moves the code after it into a
closure
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}  // And we see how reset closes all those closures
// And it looks just like the old version!
```

Creazione di funzioni che portano a continuazione

Se lo `shift` viene chiamato al di fuori di un blocco di `reset` delimitante, può essere utilizzato per creare funzioni che creano continuazioni all'interno di un blocco di `reset` . È importante notare che il tipo di `shift` non è solo `(((A => B) => C) => A)` , in realtà è `(((A => B) => C) => (A @cpsParam[B, C]))` . Tale annotazione segna dove sono necessarie le trasformazioni CPS. Le funzioni che chiamano `shift` senza `reset` hanno il tipo di ritorno "infetto" con quell'annotazione.

All'interno di un blocco di `reset` , il valore di `A @cpsParam[B, C]` sembra avere un valore di `A` , anche se in realtà è solo una finzione. La continuazione necessaria per completare il calcolo ha tipo `A => B` , quindi il codice che segue un metodo che restituisce questo tipo deve restituire `B` `C` è il tipo di

ritorno "reale", e dopo la trasformazione di CPS la chiamata di funzione ha il tipo `C`

Ora, l'esempio, preso dallo Scaladoc della biblioteca

```
val sessions = new HashMap[UUID, Int=>Unit]
def ask(prompt: String): Int @suspendable = // alias for @cpsParam[Unit, Unit]. @cps[Unit] is
also an alias. (@cps[A] = @cpsParam[A,A])
  shift {
    k: (Int => Unit) => {
      println(prompt)
      val id = uuidGen
      sessions += id -> k
    }
  }

def go(): Unit = reset {
  println("Welcome!")
  val first = ask("Please give me a number") // Uses CPS just like shift
  val second = ask("Please enter another number")
  printf("The sum of your numbers is: %d\n", first + second)
}
```

Qui, `ask` memorizzerà la continuazione in una mappa, e in seguito qualche altro codice può recuperare quella "sessione" e passare il risultato della query all'utente. In questo modo, `go` può effettivamente utilizzare una libreria asincrona mentre il suo codice sembra un normale codice imperativo.

Leggi Libreria di continuazioni online: https://riptutorial.com/it/scala/topic/8312/libreria-di-continuazioni

Capitolo 32: Macro

introduzione

I macro sono una forma di metaprogrammazione del tempo di compilazione. Alcuni elementi del codice Scala, come annotazioni e metodi, possono essere fatti per trasformare un altro codice quando vengono compilati. Le macro sono normali codici Scala che operano su tipi di dati che rappresentano un altro codice. Il plugin [Macro Paradise] [] estende le capacità dei macro oltre il linguaggio di base. [Macro Paradise]: http://docs.scala-lang.org/overviews/macros/paradise.html

Sintassi

- def x () = macro x_impl // x è una macro, dove x_impl è usato per trasformare il codice
- def macroTransform (annottees: Any *): Any = macro impl // Utilizzare nelle annotazioni per renderli macro

Osservazioni

Le macro sono una funzione linguistica che deve essere abilitata, importando `scala.language.macros` o con l'opzione del compilatore `-language:macros` . Solo le definizioni macro richiedono questo; il codice che utilizza le macro non ha bisogno di farlo.

Examples

Annotazione Macro

Questa semplice annotazione macro emette l'elemento annotato così com'è.

```
import scala.annotation.{compileTimeOnly, StaticAnnotation}
import scala.reflect.macros.whitebox.Context

@compileTimeOnly("enable macro paradise to expand macro annotations")
class noop extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any = macro linkMacro.impl
}

object linkMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._

    c.Expr[Any](q"{..$annottees}")
  }
}
```

L'annotazione `@compileTimeOnly` genera un errore con un messaggio che indica che il plug-in del compilatore `paradise` deve essere incluso per utilizzare questa macro. Le istruzioni per includere questo tramite SBT sono qui .

Puoi utilizzare la macro sopra descritta in questo modo:

```
@noop
case class Foo(a: String, b: Int)

@noop
object Bar {
  def f(): String = "hello"
}

@noop
def g(): Int = 10
```

Metodo Macro

Quando un metodo è definito come una macro, il compilatore prende il codice che viene passato come argomento e lo trasforma in un AST. Quindi richiama l'implementazione della macro con tale AST e restituisce un nuovo AST che viene quindi ricollegato al suo sito di chiamata.

```
import reflect.macros.blackbox.Context

object Macros {
  // This macro simply sees if the argument is the result of an addition expression.
  // E.g. isAddition(1+1) and isAddition("a"+1).
  // but !isAddition(1+1-1), as the addition is underneath a subtraction, and also
  // !isAddition(x.+), and !isAddition(x.+(a,b)) as there must be exactly one argument.
  def isAddition(x: Any): Boolean = macro isAddition_impl

  // The signature of the macro implementation is the same as the macro definition,
  // but with a new Context parameter, and everything else is wrapped in an Expr.
  def isAddition_impl(c: Context)(expr: c.Expr[Any]): c.Expr[Boolean] = {
    import c.universe._ // The universe contains all the useful methods and types
    val plusName = TermName("+").encodedName // Take the name + and encode it as $plus
    expr.tree match { // Turn expr into an AST representing the code in isAddition(...)
      case Apply(Select(_, `plusName`), List(_)) => reify(true)
      // Pattern match the AST to see whether we have an addition
      // Above we match this AST
      //             Apply (function application)
      //            /     \
      //         Select  List(_) (exactly one argument)
      // (selection ^ of entity, basically the . in x.y)
      //      /          \
      //    _              \
      //               `plusName` (method named +)
      case _                                      => reify(false)
      // reify is a macro you use when writing macros
      // It takes the code given as its argument and creates an Expr out of it
    }
  }
}
```

È anche possibile avere macro che accettano `Tree` s come argomenti. Come la `reify` viene usata per creare `Expr` s, l'interpolatore `q` (per quasi una stringa) ci consente di creare e decostruire l' `Tree` . Nota che potremmo aver usato `q` sopra (`expr.tree` è, sorpresa, anche un `Tree` stesso), ma non per scopi dimostrativi.

```
// No Exprs, just Trees
def isAddition_impl(c: Context)(tree: c.Tree): c.Tree = {
  import c.universe._
  tree match {
    // q is a macro too, so it must be used with string literals.
    // It can destructure and create Trees.
    // Note how there was no need to encode + this time, as q is smart enough to do it itself.
    case q"${_} + ${_}" => q"true"
    case _              => q"false"
  }
}
```

Errori nelle macro

Le macro possono attivare avvisi ed errori del compilatore attraverso l'uso del loro `Context` .

Diciamo che siamo particolarmente zelanti quando si tratta di codice errato, e vogliamo contrassegnare ogni istanza di debito tecnico con un messaggio informativo del compilatore (non pensiamo a quanto sia pessima questa idea). Possiamo usare una macro che non fa nulla se non emettere un messaggio del genere.

```
import reflect.macros.blackbox.Context

def debtMark(message: String): Unit = macro debtMark_impl
def debtMarkImpl(c: Context)(message: c.Tree): c.Tree = {
  message match {
    case Literal(Constant(msg: String)) => c.info(c.enclosingPosition, msg, false)
    // false above means "do not force this message to be shown unless -verbose"
    case _                              => c.abort(c.enclosingPosition, "Message must be a
string literal.")
    // Abort causes the compilation to completely fail. It's not even a compile error, where
    // multiple can stack up; this just kills everything.
  }
  q"()" // At runtime this method does nothing, so we return ()
}
```

Inoltre, invece di usare `???` per contrassegnare il codice non implementato, possiamo creare due macro, `!!!` e `?!?` , che hanno lo stesso scopo, ma emettono avvisi del compilatore. `?!?` causerà l'emissione di un avviso e `!!!` causerà un errore definitivo.

```
import reflect.macros.blackbox.Context

def ?!? : Nothing = macro impl_?!?
def !!! : Nothing = macro impl_!!!

def impl_?!?(c: Context): c.Tree = {
  import c.universe._
  c.warning(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
  // If someone were to shadow the scala package, scala.Predef.??? would not work, as it
  // would end up referring to the scala that shadows and not the actual scala.
  // ROOTPKG is the very root of the tree, and acts like it is imported anew in every
  // expression. It is actually named _root_, but if someone were to shadow it, every
  // reference to it would be an error. It allows us to safely access ??? and know that
  // it is the one we want.
}
```

```
def impl_!!!(c: Context): c.Tree = {
  import c.universe._
  c.error(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
}
```

Leggi Macro online: https://riptutorial.com/it/scala/topic/3808/macro

Capitolo 33: Mentre cicli

Sintassi

- while (boolean_expression) {block_expression}
- do {block_expression} while (boolean_expression)

Parametri

Parametro	Dettagli
`boolean_expression`	Qualsiasi espressione che valuterà per `true` o `false` .
`block_expression`	Qualsiasi espressione o insieme di espressioni che verranno valutate se `boolean_expression` `true` .

Osservazioni

La differenza principale tra i cicli `while` e `do-while` è se eseguono l' `block_expression` prima che controllino se devono eseguire il ciclo.

Poiché i cicli `while` e `do-while` si basano su un'espressione per valutare `false` da terminare, spesso richiedono che lo stato mutabile venga dichiarato all'esterno del ciclo e quindi modificato all'interno del ciclo.

Examples

Mentre cicli

```
var line = 0
var maximum_lines = 5

while (line < maximum_lines) {
  line = line + 1
  println("Line number: " + line)
}
```

Cicli Do-While

```
var line = 0
var maximum_lines = 5

do {
  line = line + 1
```

```
  println("Line number: " + line)
} while (line < maximum_lines)
```

Il ciclo `do` / `while` è usato raramente nella programmazione funzionale, ma può essere utilizzato per aggirare la mancanza di supporto per il costrutto `break` / `continue` , come visto in altre lingue:

```
if(initial_condition) do if(filter) {
  ...
} while(continuation_condition)
```

Leggi Mentre cicli online: https://riptutorial.com/it/scala/topic/650/mentre-cicli

Capitolo 34: Migliori pratiche

Osservazioni

> Preferisci vals, oggetti immutabili e metodi senza effetti collaterali. Raggiungi per loro prima. Usa vars, oggetti mutabili e metodi con effetti collaterali quando hai una necessità specifica e giustificazione per loro.

- *Programmazione in Scala* , di Odersky, Spoon e Venners

Ci sono più esempi e linee guida in questa presentazione di Odersky.

Examples

Mantienilo semplice

Non complicare eccessivamente compiti semplici. Il più delle volte ti serviranno solo:

- tipi di dati algebrici
- ricorsione strutturale
- api simili a monadi (`map` , `flatMap` , `fold`)

Ci sono un sacco di cose complicate in Scala, come ad esempio:

- `Cake pattern` o `Reader Monad` per iniezione di dipendenza.
- Passare valori arbitrari come argomenti `implicit` .

Queste cose non sono chiare per i nuovi arrivati: evita di usarle prima di capirle. L'utilizzo di concetti avanzati senza una reale necessità offusca il codice, rendendolo *meno* manutenibile.

Non imballare troppo in una sola espressione.

- Trova nomi significativi per le unità di calcolo.
- Utilizzare `for` comprensioni o `map` per combinare insieme i calcoli.

Diciamo che hai qualcosa del genere:

```
if (userAuthorized.nonEmtpy) {
  makeRequest().map {
    case Success(respone) =>
      someProcessing(..)
      if (resendToUser) {
        sendToUser(...)
      }
    ...
  }
}
```

Se tutte le funzioni restituiscono `Either` o di un altro `Validation` -come tipo, è possibile scrivere:

```
for {
  user     <- authorizeUser
  response <- requestToThirdParty(user)
  _        <- someProcessing(...)
} {
  sendToUser
}
```

Preferisci uno stile funzionale, in modo ragionevole

Di default:

- Usa `val` , non `var` , ove possibile. Ciò consente di sfruttare senza problemi un certo numero di utility funzionali, compresa la distribuzione del lavoro.
- Usa `recursion` e `comprehensions` s, non cicli.
- Usa collezioni immutabili. Questo è un correttivo per l'utilizzo di `val` quando possibile.
- Concentrati sulle trasformazioni di dati, sulla logica in stile CQRS e non su CRUD.

Ci sono buoni motivi per scegliere lo stile non funzionale:

- `var` può essere utilizzato per lo stato locale (ad esempio, all'interno di un attore).
- `mutable` offre prestazioni migliori in determinate situazioni.

Leggi Migliori pratiche online: https://riptutorial.com/it/scala/topic/4376/migliori-pratiche

Capitolo 35: monadi

Examples

Definizione di Monade

Informalmente, una monade è un contenitore di elementi, indicato come `F[_]` , ricco di 2 funzioni: `flatMap` (per trasformare questo contenitore) e `unit` (per creare questo contenitore).

Esempi di librerie comuni includono `List[T]` , `Set[T]` e `Option[T]` .

Definizione formale

Monad `M` è un tipo parametrico `M[T]` con due operazioni `flatMap` e `unit` , come ad esempio:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

Queste funzioni devono soddisfare tre leggi:

1. *Associatività* : `(m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)`
   Cioè, se la sequenza è invariata puoi applicare i termini in qualsiasi ordine. Quindi, applicando `m f` , e quindi applicando il risultato a `g` otterrà lo stesso risultato dell'applicazione di `f` a `g` applicando `m` a quel risultato.
2. *Unità sinistra* : `unit(x) flatMap f == f(x)`
   Cioè, l'unità monad di `x` flat-mapped su `f` equivale a applicare `f` a `x` .
3. *Unità destra* : `m flatMap unit == m`
   Questa è una "identità": qualsiasi unità monad flat-mapped restituirà una monade equivalente a se stessa.

Esempio :

```
val m = List(1, 2, 3)
def unit(x: Int): List[Int] = List(x)
def f(x: Int): List[Int] = List(x * x)
def g(x: Int): List[Int] = List(x * x * x)
val x = 1
```

1. *Associatività* :

```
(m flatMap f).flatMap(g) == m.flatMap(x => f(x) flatMap g) //Boolean = true
//Left side:
List(1, 4, 9).flatMap(g) // List(1, 64, 729)
//Right side:
 m.flatMap(x => (x * x) * (x * x) * (x * x)) //List(1, 64, 729)
```

2. *Unità sinistra*

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. *Unità giusta*

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```

Le collezioni standard sono Monade

La maggior parte delle raccolte standard sono monade (`List[T]` , `Option[T]`) o monad (`Either[T]` , `Future[T]`). Queste raccolte possono essere facilmente combinate insieme `for` comprensione (che è un modo equivalente di scrivere trasformazioni `flatMap`):

```
val a = List(1, 2, 3)
val b = List(3, 4, 5)
for {
  i <- a
  j <- b
} yield(i * j)
```

Quanto sopra è equivalente a:

```
a flatMap {
  i => b map {
    j => i * j
  }
}
```

Poiché una monade conserva la *struttura dei dati* e agisce solo sugli elementi all'interno di quella struttura, possiamo disporre di infinite strutture monadiche a catena, come mostrato qui in una comprensione preliminare.

Leggi monadi online: https://riptutorial.com/it/scala/topic/4112/monadi

Capitolo 36: Operatori in Scala

Examples

Operatori integrati

Scala ha i seguenti operatori integrati (metodi / elementi di linguaggio con regole di precedenza predefinite):

genere	Simbolo	Esempio
Operatori aritmetici	`+ - * / %`	`a + b`
Operatori relazionali	`== != > < >= <=`	`a > b`
Operatori logici	`&& & || | !`	`a && b`
Operatori bit-saggio	`& | ^ ~ << >> >>>`	`a & b` , `~a` , `a >>> b`
Operatori di assegnazione	`= += -= *= /= %= <<= >>= &= ^= |=`	`a += b`

Gli operatori di Scala hanno lo stesso significato di Java

Nota : metodi che terminano con `:` bind a destra (e associativo a destra), quindi la chiamata con `list.::(value)` può essere scritta come `value :: list` con sintassi dell'operatore. (`1 :: 2 :: 3 :: Nil` è uguale a `1 :: (2 :: (3 :: Nil))`)

Sovraccarico dell'operatore

In Scala puoi definire i tuoi operatori:

```
class Team {
   def +(member: Person) = ...
}
```

Con quanto sopra definito puoi usarlo come:

```
ITTeam + Jack
```

o

```
ITTeam.+(Jack)
```

Per definire gli operatori unari è possibile `unary_` con `unary_` . Ad esempio `unary_!`

```
class MyBigInt {
```

```
   def unary_! = ...
}

var a: MyBigInt = new MyBigInt
var b = !a
```

Precedenza dell'operatore

Categoria	Operatore	Associatività
Postfix	`() []`	Da sinistra a destra
unario	`! ~`	Da destra a sinistra
moltiplicativo	`* / %`	Da sinistra a destra
Additivo	`+ -`	Da sinistra a destra
Cambio	`>> >>> <<`	Da sinistra a destra
relazionale	`> >= < <=`	Da sinistra a destra
Uguaglianza	`== !=`	Da sinistra a destra
Bitwise e	`&`	Da sinistra a destra
Xor bit a bit	`^`	Da sinistra a destra
Bitwise o	`|`	Da sinistra a destra
Logico e	`&&`	Da sinistra a destra
Logico o	`||`	Da sinistra a destra
assegnazione	`= += -= *= /= %= >>= <<= &= ^= | =`	Da destra a sinistra
Virgola	`,`	Da sinistra a destra

La programmazione in Scala fornisce il seguente schema basato sul 1 ° carattere nell'operatore. Es. `>` È il 1 ° carattere nell'operatore `>>>` :

Operatore
(tutti gli altri caratteri speciali)
`* / %`
`+ -`
`:`

Operatore
= !
< >
&
^
|
(tutte le lettere)
(tutti gli operatori di assegnazione)

L'unica eccezione a questa regola riguarda gli *operatori di assegnazione* , ad esempio `+=` , `*=` , ecc. Se un operatore termina con un carattere uguale (=) e non è uno degli operatori di confronto `<=` , `>=` , `==` o `!=` , quindi la precedenza dell'operatore è la stessa del semplice compito. In altre parole, inferiore a quello di qualsiasi altro operatore.

Leggi Operatori in Scala online: https://riptutorial.com/it/scala/topic/6604/operatori-in-scala

Capitolo 37: Opzione Classe

Sintassi

- class Alcuni [+ T] (valore: T) estende l'opzione [T]
- oggetto Nessuno estende l'opzione [Nothing]
- Opzione [T] (valore: T)

  Costruttore per creare un `Some(value)` o `None` secondo il valore fornito.

Examples

Opzioni come raccolte

`Option` hanno alcune utili funzioni di ordine superiore che possono essere facilmente comprese visualizzando opzioni come *raccolte con zero o un elemento* - dove `None` si comporta come la raccolta vuota, e `Some(x)` si comporta come una raccolta con un singolo elemento, `x` .

```
val option: Option[String] = ???

option.map(_.trim) // None if option is None, Some(s.trim) if Some(s)
option.foreach(println) // prints the string if it exists, does nothing otherwise
option.forall(_.length > 4) // true if None or if Some(s) and s.length > 4
option.exists(_.length > 4) // true if Some(s) and s.length > 4
option.toList // returns an actual list
```

Utilizzo dell'opzione anziché di null

In Java (e in altre lingue), l'utilizzo di `null` è un modo comune per indicare che non esiste alcun valore collegato a una variabile di riferimento. In Scala, l'uso `Option` è preferito rispetto all'utilizzo di `null` . `Option` include valori che *potrebbero* essere `null` .

`None` è una sottoclasse di `Option` racchiude un riferimento null. `Some` è una sottoclasse di `Option` racchiude un riferimento non nullo.

Avvolgere un riferimento è facile:

```
val nothing = Option(null) // None
val something = Option("Aren't options cool?") // Some("Aren't options cool?")
```

Questo è un codice tipico quando si chiama una libreria Java che potrebbe restituire un riferimento null:

```
val resource = Option(JavaLib.getResource())
// if null, then resource = None
```

```
// else resource = Some(resource)
```

Se `getResource()` restituisce un valore `null` , la `resource` sarà un oggetto `None` . Altrimenti sarà un `Some(resource)` oggetto `Some(resource)` . Il modo migliore per gestire `Option` è l'utilizzo di funzioni di ordine superiore disponibili all'interno del tipo di `Option` . Ad esempio, se si desidera verificare se il proprio valore non è `None` (simile alla verifica del `value == null`), si utilizzerà la funzione `isDefined` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isDefined) {  // resource is `Some(_)` type
  val r: Resource = resource.get
  r.connect()
}
```

Allo stesso modo, per verificare la presenza di un riferimento `null` puoi farlo:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isEmpty) { // resource is `None` type.
  System.out.println("Resource is empty! Cannot connect.")
}
```

È preferibile che si tratti l'esecuzione condizionale sul valore avvolto di `Option` (senza utilizzare il metodo 'eccezionale' `Option.get`) trattando l' `Option` come monade e utilizzando `foreach` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
resource foreach (r => r.connect())
// if r is defined, then r.connect() is run
// if r is empty, then it does nothing
```

Se è richiesta un'istanza `Resource` (rispetto a un'istanza `Option[Resource]` , è comunque possibile utilizzare `Option` per proteggere da valori null. Qui il metodo `getOrElse` fornisce un valore predefinito:

```
lazy val defaultResource = new Resource()
val resource: Resource = Option(JavaLib.getResource()).getOrElse(defaultResource)
```

Il codice Java non gestirà prontamente l' `Option` di Scala, quindi quando si passano i valori al codice Java è buona norma scartare `Option` , passando a un valore `null` o ad un valore predefinito appropriato, ove appropriato:

```
val resource: Option[Resource] = ???
JavaLib.sendResource(resource.orNull)
JavaLib.sendResource(resource.getOrElse(defaultResource)) //
```

Nozioni di base

`Option` è una struttura di dati che contiene un singolo valore o nessun valore. `Option` può essere pensata come una raccolta di zero o di un elemento.

L'opzione è una classe astratta con due figli: `Some` e `None` .

`Some` contengono un singolo valore e `None` non contiene alcun valore.

`Option` è utile in espressioni che altrimenti utilizzerebbero `null` per rappresentare la mancanza di un valore concreto. Questo protegge contro una `NullPointerException` e consente la composizione di molte espressioni che potrebbero non restituire un valore utilizzando combinatori come `Map` , `FlatMap` , ecc.

Esempio con la mappa

```
val countries = Map(
  "USA" -> "Washington",
  "UK" -> "London",
  "Germany" -> "Berlin",
  "Netherlands" -> "Amsterdam",
  "Japan" -> "Tokyo"
)

println(countries.get("USA")) // Some(Washington)
println(countries.get("France")) // None
println(countries.get("USA").get) // Washington
println(countries.get("France").get) // Error: NoSuchElementException
println(countries.get("USA").getOrElse("Nope")) // Washington
println(countries.get("France").getOrElse("Nope")) // Nope
```

`Option[A]` è **sigillata** e quindi non può essere estesa. Quindi la semantica è stabile e può essere invocata.

Opzioni per le comprensioni

`Option` s hanno un metodo `flatMap` . Ciò significa che possono essere utilizzati in a comprensione. In questo modo possiamo sollevare funzioni regolari per lavorare su `Option` s senza doverle ridefinire.

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = Option(2)

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = Some(3)
```

Quando uno dei valori è un `None` il risultato finale del calcolo sarà `None` .

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = None

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = None
```

Nota: questo modello si estende più in generale per i concetti denominati `Monad` s. (Ulteriori informazioni dovrebbero essere disponibili sulle pagine relative a comprensione e `Monad` s)

In generale non è possibile mescolare diverse monadi in a comprensione. Ma poiché `Option` può essere facilmente convertita in `Iterable` , possiamo facilmente mescolare `Option` s e `Iterable` chiamando il metodo `.toIterable` .

```
val option: Option[Int] = Option(1)
val iterable: Iterable[Int] = Iterable(2, 3, 4, 5)

// does NOT compile since we cannot mix Monads in a for comprehension
// val myResult = for {
//   optionValue <- option
//   iterableValue <- iterable
//} yield optionValue + iterableValue

// It does compile when adding a .toIterable on the option
val myResult = for {
  optionValue <- option.toIterable
  iterableValue <- iterable
} yield optionValue + iterableValue
// myResult: Iterable[Int] = List(2, 3, 4, 5)
```

Una piccola nota: se avessimo definito la nostra comprensione, l'altro modo di comprensione sarebbe stato compilato poiché la nostra opzione sarebbe stata convertita implicitamente. Per questo motivo è utile aggiungere sempre questa `.toIterable` (o la funzione corrispondente a seconda della raccolta che si sta utilizzando) per coerenza.

Leggi Opzione Classe online: https://riptutorial.com/it/scala/topic/2293/opzione-classe

Capitolo 38: Pacchi

introduzione

I pacchetti in Scala gestiscono gli spazi dei nomi in programmi di grandi dimensioni. Ad esempio, la `connection` del nome può verificarsi nei pacchetti `com.sql` e `org.http` . È possibile utilizzare rispettivamente `com.sql.connection` e `org.http.connection` per accedere a ciascuno di questi pacchetti.

Examples

Struttura del pacchetto

```
package com {
    package utility {
        package serialization {
            class Serializer
            ...
        }
    }
}
```

Pacchetti e file

La clausola del pacchetto non è direttamente associata al file in cui è stata trovata. È possibile trovare elementi comuni della clausola del pacchetto in file diversi. Ad esempio, le clausole del pacchetto qui sotto possono essere trovate nel file math1.scala e nel file math2.scala.

File math1.scala

```
package org {
    package math {
        package statistics {
            class Interval
        }
    }
}
```

File math2.scala

```
package org {
    package math{
        package probability {
            class Density
        }
    }
}
```

File study.scala

```
import org.math.probability.Density
import org.math.statistics.Interval

object Study {

    def main(args: Array[String]): Unit = {
        var a = new Interval()
        var b = new Density()
    }
}
```

Convocazione di denominazione del pacchetto

I pacchetti di Scala dovrebbero seguire le convenzioni di denominazione dei pacchetti Java.
I nomi dei pacchetti sono scritti in lettere minuscole per evitare conflitti con i nomi delle classi o delle interfacce. Le aziende utilizzano il loro nome di dominio Internet inverso per iniziare i nomi dei pacchetti, ad esempio,

```
io.super.math
```

Leggi Pacchi online: https://riptutorial.com/it/scala/topic/8231/pacchi

Capitolo 39: Pattern Matching

Sintassi

- selettore corrisponde a partialFunction
- selettore match {elenco di alternative di casi) // Questa è la forma più comune di quanto sopra

Parametri

Parametro	Dettagli
selettore	L'espressione il cui valore è associato al modello.
alternative	un elenco di alternative minime di `case` .

Examples

Match Pattern semplice

Questo esempio mostra come abbinare un input a più valori:

```
def f(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Unknown!"
}

f(2)  // "Two"
f(3)  // "Unknown!"
```

Dimostrazione dal vivo

Nota: _ è la *caduta attraverso* o caso *di default,* ma non è necessario.

```
def g(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
}

g(1)  // "One"
g(3)  // throws a MatchError
```

Per evitare di lanciare un'eccezione, qui è la migliore pratica di programmazione funzionale per gestire il caso predefinito (`case _ => <do something>`). Si noti che la corrispondenza su *una classe di case* può aiutare il compilatore a produrre un avvertimento se manca un caso. Lo stesso vale per i tipi definiti dall'utente che estendono un tratto sigillato. Se la corrispondenza è totale,

potrebbe non essere necessario un caso predefinito

È anche possibile confrontarsi con valori che non sono definiti in linea. Questi devono essere *identificatori stabili* , che si ottengono usando un nome in maiuscolo o racchiudendo i backtick.

Con `One` e `two` definiti da qualche altra parte o passati come parametri di funzione:

```
val One: Int = 1
val two: Int = 2
```

Possono essere abbinati nel modo seguente:

```
def g(x: Int): String = x match {
  case One => "One"
  case `two` => "Two"
}
```

A differenza di altri linguaggi di programmazione come Java, ad esempio, non vi è alcuna caduta. Se un blocco di casi corrisponde a un input, viene eseguito e la corrispondenza è finita. Pertanto il caso meno specifico dovrebbe essere l'ultimo blocco del caso.

```
def f(x: Int): String = x match {
  case _ => "Default"
  case 1 => "One"
}

f(5) // "Default"
f(1) // "Default"
```

Pattern Matching With Stable Identifier

Nella corrispondenza del modello standard, l'identificatore utilizzato ombreggia qualsiasi identificatore nell'ambito di inclusione. A volte è necessario abbinare la variabile dello scope che racchiude.

La seguente funzione di esempio prende un carattere e un elenco di tuple e restituisce un nuovo elenco di tuple. Se il carattere esiste come primo elemento in una delle tuple, il secondo elemento viene incrementato. Se non esiste ancora nella lista, viene creata una nuova tupla.

```
def tabulate(char: Char, tab: List[(Char, Int)]): List[(Char, Int)] = tab match {
  case Nil => List((char, 1))
  case (`char`, count) :: tail => (char, count + 1) :: tail
  case head :: tail => head :: tabulate(char, tail)
}
```

Quanto sopra dimostra la corrispondenza del modello in cui l'input del metodo, `char` , viene mantenuto "stabile" nella corrispondenza del modello: ovvero, se si chiama `tabulate('x', ...)` , la prima dichiarazione case verrà interpretata come:

```
case('x', count) => ...
```

Scala interpreterà qualsiasi variabile demarcata con un segno di spunta come identificatore stabile: interpreterà anche qualsiasi variabile che inizia con una lettera maiuscola nello stesso modo.

Pattern Matching su un Seq

Per verificare un numero preciso di elementi nella collezione

```
def f(ints: Seq[Int]): String = ints match {
  case Seq() =>
      "The Seq is empty !"
  case Seq(first) =>
      s"The seq has exactly one element : $first"
  case Seq(first, second) =>
      s"The seq has exactly two elements : $first, $second"
  case  s @ Seq(_, _, _) =>
      s"s is a Seq of length three and looks like ${s}"  // Note individual elements are not
bound to their own names.
  case s: Seq[Int] if s.length == 4 =>
      s"s is a Seq of Ints of exactly length 4"  // Again, individual elements are not bound
to their own names.
  case _ =>
      "No match was found!"
}
```

Dimostrazione dal vivo

Per estrarre il (i) primo (i) elemento (i) e mantenere il resto come una raccolta:

```
def f(ints: Seq[Int]): String = ints match {
  case Seq(first, second, tail @ _*) =>
      s"The seq has at least two elements : $first, $second. The rest of the Seq is $tail"
  case Seq(first, tail @ _*) =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  // alternative syntax
  // here of course this one will never match since it checks
  // for the same thing as the one above
  case first +: tail =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  case _ =>
      "The seq didn't match any of the above, so it must be empty"
}
```

In generale, qualsiasi forma che può essere usata per costruire una sequenza può essere usata per modellare la corrispondenza con una sequenza esistente.

Si noti che durante l'utilizzo di `Nil` e `::` funzionerà quando il modello corrisponde a una sequenza, lo converte in una `List` e può avere risultati imprevisti. Vincolati a `Seq( ...)` e `+:` per evitare questo.

Nota che mentre usi `::` non funzionerà per `WrappedArray` , `Vector` ecc., Vedi:

```
scala> def f(ints:Seq[Int]) = ints match {
     | case h :: t => h
     | case _ => "No match"
```

```
     | }
f: (ints: Seq[Int])Any

scala> f(Array(1,2))
res0: Any = No match
```

E con `+:`:

```
scala> def g(ints:Seq[Int]) = ints match {
     | case h+:t => h
     | case _ => "No match"
     | }
g: (ints: Seq[Int])Any

scala> g(Array(1,2).toSeq)
res4: Any = 1
```

Guardie (se espressioni)

Le espressioni case possono essere combinate con le espressioni if per fornire una logica extra durante la corrispondenza dei pattern.

```
def checkSign(x: Int): String = {
    x match {
      case a if a < 0 => s"$a is a negative number"
      case b if b > 0 => s"$b is a positive number"
      case c => s"$c neither positive nor negative"
    }
}
```

È importante assicurarsi che le tue guardie non creino una corrispondenza non esaustiva (il compilatore spesso non lo prenderà):

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case None    => doSomethingIfNone
}
```

Questo genera un `MatchError` su numeri dispari. È necessario tenere conto di tutti i casi o utilizzare una casella di ricerca con caratteri jolly:

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case _ => doSomethingIfNoneOrOdd
}
```

Pattern matching con classi di casi

Ogni classe case definisce un estrattore che può essere usato per catturare i membri della classe case quando il pattern matching:

```
case class Student(name: String, email: String)

def matchStudent1(student: Student): String = student match {
    case Student(name, email) => s"$name has the following email: $email" // extract name and
email
}
```

Vengono applicate tutte le normali regole di corrispondenza del modello: puoi utilizzare guardie ed espressioni costanti per controllare la corrispondenza:

```
def matchStudent2(student: Student): String = student match {
    case Student("Paul", _) => "Matched Paul" // Only match students named Paul, ignore email
    case Student(name, _) if name == "Paul" => "Matched Paul" // Use a guard to match students
named Paul, ignore email
    case s if s.name == "Paul" => "Matched Paul" // Don't use extractor; use a guard to match
students named Paul, ignore email
    case Student("Joe", email) => s"Joe has email $email" // Match students named Joe, capture
their email
    case Student(name, email) if name == "Joe" => s"Joe has email $email" // use a guard to
match students named Joe, capture their email
    case Student(name, email) => s"$name has email $email." // Match all students, capture
name and email
}
```

Abbinamento su un'opzione

Se stai abbinando su un tipo di opzione :

```
def f(x: Option[Int]) = x match {
    case Some(i) => doSomething(i)
    case None    => doSomethingIfNone
}
```

Questo è funzionalmente equivalente all'uso di `fold` o `map` / `getOrElse` :

```
def g(x: Option[Int]) = x.fold(doSomethingIfNone)(doSomething)
def h(x: Option[Int]) = x.map(doSomething).getOrElse(doSomethingIfNone)
```

Tratti sigillati che corrispondono al modello

Quando pattern corrisponde a un oggetto il cui tipo è un tratto sigillato, Scala verificherà in fase di compilazione che tutti i casi siano "esaustivamente abbinati":

```
sealed trait Shape
case class Square(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape


def matchShape(shape: Shape): String = shape match {
    case Square(height, width) => "It's a square"
    case Circle(radius)        => "It's a circle"
    //no case for Point because it would cause a compiler warning.
```

```
}
```

Se in seguito viene aggiunta una nuova `case class` per `Shape` , tutte le istruzioni di `match` su `Shape` inizieranno a generare un avviso del compilatore. Ciò facilita il completo refactoring: il compilatore avviserà lo sviluppatore di tutto il codice che deve essere aggiornato.

Pattern Matching con Regex

```
val emailRegex: Regex = "(.+)@(.+)\\.(.+)".r

"name@example.com" match {
  case emailRegex(userName, domain, topDomain) => println(s"Hi $userName from $domain")
  case _ => println(s"This is not a valid email.")
}
```

In questo esempio, la regex tenta di abbinare l'indirizzo email fornito. In caso `userName` , `userName` e `domain` vengono estratti e stampati. `topDomain` viene estratto, ma in questo esempio non viene eseguito nulla. Chiamare `.r` su una stringa `str` equivale a `new Regex(str)` . La funzione `r` è disponibile tramite una conversione implicita .

Pattern binder (@)

Il segno `@` associa una variabile a un nome durante una corrispondenza di modello. La variabile associata può essere l'intero oggetto abbinato o parte dell'oggetto abbinato:

```
sealed trait Shape
case class Rectangle(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape

(Circle(5): Shape) match {
  case Rectangle(h, w) => s"rectangle, $h x $w."
  case Circle(r) if r > 9 => s"large circle"
  case c @ Circle(_) => s"small circle: ${c.radius}"  // Whole matched object is bound to c
  case Point => "point"
}
```

```
> res0: String = small circle: 5
```

L'identificatore associato può essere utilizzato nei filtri condizionali. Così:

```
case Circle(r) if r > 9 => s"large circle"
```

può essere scritto come:

```
case c @ Circle(_) if c.radius > 9 => s"large circle"
```

Il nome può essere associato solo a una parte del modello abbinato:

```
Seq(Some(1), Some(2), None) match {
  // Only the first element of the matched sequence is bound to the name 'c'
```

```
  case Seq(c @ Some(1), _*) => head
  case _ => None
}
```

```
> res0: Option[Int] = Some(1)
```

Tipi di corrispondenza del modello

La corrispondenza del modello può anche essere utilizzata per verificare il tipo di un'istanza, piuttosto che utilizzare `isInstanceOf[B]` :

```
val anyRef: AnyRef = ""

anyRef match {
  case _: Number       => "It is a number"
  case _: String       => "It is a string"
  case _: CharSequence => "It is a char sequence"
}
//> res0: String = It is a string
```

L'ordine dei casi è importante:

```
anyRef match {
  case _: Number       => "It is a number"
  case _: CharSequence => "It is a char sequence"
  case _: String       => "It is a string"
}
//> res1: String = It is a char sequence
```

In questo modo è simile a una classica istruzione "switch", senza la funzionalità fall-through. Tuttavia, puoi anche creare pattern match e "estrai" dal tipo in questione. Per esempio:

```
case class Foo(s: String)
case class Bar(s: String)
case class Woo(s: String, i: Int)

def matcher(g: Any):String = {
  g match {
    case Bar(s) => s + " is classy!"
    case Foo(_) => "Someone is wicked smart!"
    case Woo(s, _) => s + " is adventerous!"
    case _ => "What are we talking about?"
  }
}

print(matcher(Foo("Diana")))  // prints 'Diana is classy!'
print(matcher(Bar("Hadas")))  // prints 'Someone is wicked smart!'
print(matcher(Woo("Beth", 27)))   // prints 'Beth is adventerous!'
print(matcher(Option("Katie")))  // prints 'What are we talking about?'
```

Si noti che nel caso `Foo` e `Woo` usiamo il carattere di sottolineatura (`_`) per "abbinare una variabile non associata". Ciò significa che il valore (in questo caso `Hadas` e `27` , rispettivamente) non è associato a un nome e quindi non è disponibile nel gestore per quel caso. Questa è una utile stenografia per abbinare il valore "qualsiasi" senza preoccuparsi di quale sia quel valore.

Pattern Matching compilato come tableswitch o lookupswitch

L'annotazione `@switch` indica al compilatore che l'istruzione `match` può essere sostituita con una singola istruzione `tableswitch` a livello bytecode. Questa è una piccola ottimizzazione che può rimuovere confronti non necessari e carichi variabili durante il runtime.

L'annotazione `@switch` funziona solo per le corrispondenze con costanti letterali e identificatori `final val` . Se il pattern match non può essere compilato come un `tableswitch` / `lookupswitch` , il compilatore solleverà un avvertimento.

```
import annotation.switch

def suffix(i: Int) = (i: @switch) match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

I risultati sono gli stessi di un normale pattern match:

```
scala> suffix(2)
res1: String = "2nd"

scala> suffix(4)
res2: String = "4th"
```

Dalla documentazione di Scala (2.8+) - `@switch` :

> Un'annotazione da applicare a un'espressione di corrispondenza. Se presente, il compilatore verificherà che la corrispondenza sia stata compilata su un tableswitch o su un interruttore di ricerca e genera un errore se invece compila in una serie di espressioni condizionali.

Dalla specifica Java:

- tableswitch : "Accedi alla tabella di salto per indice e salta"
- lookupswitch : "Accedi alla tabella di salto per corrispondenza e salto"

Abbinamento di più pattern contemporaneamente

Il `|` può essere utilizzato per avere una singola istruzione di dichiarazione del caso contro più input per ottenere lo stesso risultato:

```
def f(str: String): String = str match {
  case "foo" | "bar" => "Matched!"
  case _ => "No match."
}

f("foo")  // res0: String = Matched!
f("bar")  // res1: String = Matched!
```

```
f("fubar")  // res2: String = No match.
```

Si noti che mentre la corrispondenza dei **valori in** questo modo funziona bene, la seguente corrispondenza dei **tipi** causerà problemi:

```
sealed class FooBar
case class Foo(s: String) extends FooBar
case class Bar(s: String) extends FooBar

val d = Foo("Diana")
val h = Bar("Hadas")

// This matcher WILL NOT work.
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) | Bar(s) => print(s)  // Won't work: s cannot be resolved
    case Foo(_) | Bar(_) => _         // Won't work: _ is an unbound placeholder
    case _ => "Could not match"
  }
}
```

Se nel secondo caso (con _) non hai bisogno del valore della variabile non associata e vuoi solo fare qualcos'altro, stai bene:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(_) | Bar(_) => "Is either Foo or Bar."  // Works fine
    case _ => "Could not match"
  }
}
```

Altrimenti, hai lasciato a spaccare i tuoi casi:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) => s
    case Bar(s) => s
    case _ => "Could not match"
  }
}
```

Pattern Matching su tuple

Dato il seguente `List` di tuple:

```
val pastries = List(("Chocolate Cupcake", 2.50),
                    ("Vanilla Cupcake", 2.25),
                    ("Plain Muffin", 3.25))
```

La corrispondenza del modello può essere utilizzata per gestire ogni elemento in modo diverso:

```
pastries foreach { pastry =>
  pastry match {
```

```
    case ("Plain Muffin", price) => println(s"Buying muffin for $price")
    case p if p._1 contains "Cupcake" => println(s"Buying cupcake for ${p._2}")
    case _ => println("We don't sell that pastry")
  }
}
```

Il primo caso mostra come abbinare una stringa specifica e ottenere il prezzo corrispondente. Il secondo caso mostra un uso dell'estrazione if e tuple per confrontarsi con gli elementi della tupla.

Leggi Pattern Matching online: https://riptutorial.com/it/scala/topic/661/pattern-matching

Capitolo 40: Per le espressioni

Sintassi

- per il corpo di {clauses}
- per {clausole} cedere il corpo
- per (clausole) del corpo
- per (clausole) cedere il corpo

Parametri

Parametro	Dettagli
per	Parola chiave richiesta per utilizzare un ciclo / comprensione for
clausole	L'iterazione e i filtri su cui lavora per.
dare la precedenza	Usalo se vuoi creare o "cedere" una collezione. L'utilizzo di `yield` farà sì che il tipo di reso di `for` sia una raccolta anziché `Unit` .
corpo	Il corpo dell'espressione for, eseguito su ogni iterazione.

Examples

Basic For Loop

```
for (x <- 1 to 10)
  println("Iteration number " + x)
```

Ciò dimostra iterando una variabile, `x` , da `1` a `10` e facendo qualcosa con quel valore. Il tipo di ritorno di questo `for` comprensione è `Unit` .

Di base per la comprensione

Ciò dimostra un filtro su un ciclo for e l'uso della `yield` per creare una "comprensione della sequenza":

```
for ( x <- 1 to 10 if x % 2 == 0)
  yield x
```

L'output per questo è:

```
scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

A per la comprensione è utile quando è necessario creare una nuova collezione basata sull'iterazione e sui suoi filtri.

Nested For Loop

Questo mostra come puoi scorrere su più variabili:

```
for {
  x <- 1 to 2
  y <- 'a' to 'd'
} println("(" + x + "," + y + ")")
```

(Si noti che `to` qui è un metodo operatore infisso che restituisce un range inclusivo . Si veda la definizione qui .)

Questo crea l'output:

```
(1,a)
(1,b)
(1,c)
(1,d)
(2,a)
(2,b)
(2,c)
(2,d)
```

Nota che questa è un'espressione equivalente, usando parentesi invece di parentesi:

```
for (
  x <- 1 to 2
  y <- 'a' to 'd'
) println("(" + x + "," + y + ")")
```

Per ottenere tutte le combinazioni in un singolo vettore, possiamo `yield` il risultato e impostarlo su un valore `val` :

```
val a = for {
  x <- 1 to 2
  y <- 'a' to 'd'
} yield "(%s,%s)".format(x, y)
// a: scala.collection.immutable.IndexedSeq[String] = Vector((1,a), (1,b), (1,c), (1,d),
(2,a), (2,b), (2,c), (2,d))
```

Monadico per le comprensioni

Se hai diversi oggetti di tipi monadici , possiamo ottenere combinazioni dei valori usando un 'per comprensione':

```
for {
   x <- Option(1)
   y <- Option("b")
   z <- List(3, 4)
```

```
} {
    // Now we can use the x, y, z variables
    println(x, y, z)
    x  // the last expression is *not* the output of the block in this case!
}

// This prints
// (1, "b", 3)
// (1, "b", 4)
```

Il tipo di ritorno di questo blocco è `Unit` .

Se gli oggetti sono dello *stesso* tipo monadico `M` (es. `Option`), allora usando `yield` restituirà un oggetto di tipo `M` invece di `Unit` .

```
val a = for {
   x <- Option(1)
   y <- Option("b")
} yield {
    // Now we can use the x, y variables
    println(x, y)
    // whatever is at the end of the block is the output
    (7 * x, y)
}

// This prints:
// (1, "b")
// The val `a` is set:
// a: Option[(Int, String)] = Some((7,b))
```

Si noti che la parola chiave `yield` *non può* essere utilizzata nell'esempio originale, dove esiste un mix di tipi monadici (`Option` ed `List`). Provare a farlo produrrà un errore di mancata corrispondenza di tipo in fase di compilazione.

Scorrere le raccolte utilizzando un ciclo For

Questo dimostra come stampare ogni elemento di una mappa

```
val map = Map(1 -> "a", 2 -> "b")
for (number <- map) println(number) // prints (1,a), (2,b)
for ((key, value) <- map) println(value) // prints a, b
```

Questo dimostra come stampare ogni elemento di una lista

```
val list = List(1,2,3)
for(number <- list) println(number) // prints 1, 2, 3
```

Desugaring per le comprensioni

`for` comprensione in Scala sono solo zucchero sintattico . Queste comprensioni sono implementate usando i `withFilter` , `foreach` , `flatMap` e `map` dei loro tipi di soggetto. Per questo motivo, solo i tipi che hanno questi metodi definiti possono essere utilizzati in termini `for`

comprensione.

A `for` comprensione del seguente modulo, con pattern `pN` , generatori `gN` e condizioni `cN` :

```
for(p0 <- x0 if g0; p1 <- g1 if c1) { ??? }
```

... `withFilter` le chiamate nidificate usando `withFilter` e `foreach` :

```
g0.withFilter({ case p0 => c0  case _ => false }).foreach({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).foreach({
    case p1 => ???
  })
})
```

Considerando che un espressione `for` / `yield` della seguente forma:

```
for(p0 <- g0 if c0; p1 <- g1 if c1) yield ???
```

... ridurrà lo zucchero alle chiamate annidate usando `withFilter` e sia `flatMap` che `map` :

```
g0.withFilter({ case p0 => c0  case _ => false }).flatMap({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).map({
    case p1 => ???
  })
})
```

(Nota che la `map` è usata nella comprensione più interna, e `flatMap` è usato in ogni comprensione esterna.)

A `for` comprensione può essere applicato a qualsiasi tipo implementando i metodi richiesti dalla rappresentazione de-zuccherata. Non ci sono restrizioni sui tipi di ritorno di questi metodi, purché siano componibili.

Leggi Per le espressioni online: https://riptutorial.com/it/scala/topic/669/per-le-espressioni

Capitolo 41: Programmazione a livello di codice

Examples

Introduzione alla programmazione a livello di carattere

Se consideriamo un elenco eterogeneo, in cui gli elementi della lista hanno tipi diversi ma conosciuti, potrebbe essere desiderabile essere in grado di eseguire collettivamente gli elementi della lista senza scartare le informazioni sul tipo degli elementi. L'esempio seguente implementa un'operazione di mapping su un semplice elenco eterogeneo.

Poiché il tipo di elemento varia, la classe di operazioni che possiamo eseguire è limitata a qualche forma di proiezione del tipo, quindi definiamo un tratto `Projection` `type Apply[A]` astratto `type Apply[A]` calcolando il *tipo* di risultato della proiezione e `def apply[A](a: A): Apply[A]` calcolando il *valore* risultante della proiezione.

```
trait Projection {
  type Apply[A] // <: Any
  def apply[A](a: A): Apply[A]
}
```

Nell'implementazione del `type Apply[A]` stiamo programmando a livello di tipo (al contrario del livello di valore).

Il nostro tipo di lista eterogenea definisce un'operazione della `map` parametrizzata dalla proiezione desiderata e dal tipo di proiezione. Il risultato dell'operazione della mappa è astratto, varierà in base alla classe e alla proiezione e, naturalmente, deve essere ancora una `HList` :

```
sealed trait HList {
  type Map[P <: Projection] <: HList
  def map[P <: Projection](p: P): Map[P]
}
```

Nel caso di `HNil` , la lista eterogenea vuota, il risultato di qualsiasi proiezione sarà sempre se stesso. Qui dichiariamo `trait HNil` come una comodità in modo che possiamo scrivere `HNil` come un tipo al posto di `HNil.type` :

```
sealed trait HNil extends HList
case object HNil extends HNil {
  type Map[P <: Projection] = HNil
  def map[P <: Projection](p: P): Map[P] = HNil
}
```

`HCons` è la lista eterogenea non vuota. Qui affermiamo che quando si applica un'operazione di mappa, il tipo di testa risultante è quello che risulta dall'applicazione della proiezione al valore

della testa (`P#Apply[H]`), e che il tipo di coda risultante è quello che risulta dalla mappatura del proiezione sulla coda (`T#Map[P]`), che è noto per essere una `HList` :

```
case class HCons[H, T <: HList](head: H, tail: T) extends HList {
  type Map[P <: Projection] = HCons[P#Apply[H], T#Map[P]]
  def map[P <: Projection](p: P): Map[P] = HCons(p.apply(head), tail.map(p))
}
```

La proiezione più ovvia è quella di eseguire una qualche forma di operazione di avvolgimento - l'esempio seguente produce un'istanza di `HCons[Option[String], HCons[Option[Int], HNil]]` :

```
HCons("1", HCons(2, HNil)).map(new Projection {
  type Apply[A] = Option[A]
  def apply[A](a: A): Apply[A] = Some(a)
})
```

Leggi Programmazione a livello di codice online:
https://riptutorial.com/it/scala/topic/3738/programmazione-a-livello-di-codice

Capitolo 42: Quasiquotes

Examples

Creare un albero di sintassi con quasiquotes

Usa quasiquotes per creare un `Tree` in una macro.

```
object macro {
  def addCreationDate(): java.util.Date = macro impl.addCreationDate
}

object impl {
  def addCreationDate(c: Context)(): c.Expr[java.util.Date] = {
    import c.universe._

    val date = q"new java.util.Date()" // this is the quasiquote
    c.Expr[java.util.Date](date)
  }
}
```

Può essere arbitrariamente complesso ma verrà convalidato per la corretta sintassi di scala.

Leggi Quasiquotes online: https://riptutorial.com/it/scala/topic/4032/quasiquotes

Capitolo 43: ricorsione

Examples

Ricorsione di coda

Usando la ricorsione regolare, ogni chiamata ricorsiva spinge un'altra voce nello stack di chiamate. Quando la ricorsione è completata, l'applicazione deve far scattare ogni voce fino in fondo. Se ci sono molte chiamate ricorsive, può finire con uno stack enorme.

Scala rimuove automaticamente la ricorsione nel caso in cui trovi la chiamata ricorsiva in posizione di coda. L'annotazione (`@tailrec`) può essere aggiunta alle funzioni ricorsive per garantire che venga eseguita l'ottimizzazione delle chiamate tail. Il compilatore mostra quindi un messaggio di errore se non è in grado di ottimizzare la ricorsione.

Ricorsione regolare

Questo esempio non è ricorsivo di coda perché quando viene effettuata la chiamata ricorsiva, la funzione deve tenere traccia della moltiplicazione che deve fare con il risultato dopo il ritorno della chiamata.

```
def fact(i : Int) : Int = {
   if(i <= 1) i
   else i * fact(i-1)
}

println(fact(5))
```

La chiamata alla funzione con il parametro darà come risultato una pila simile a questa:

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 * 1)))))
(* 5 (* 4 (* 3 (* 2))))
(* 5 (* 4 (* 6)))
(* 5 (* 24))
120
```

Se proviamo ad annotare questo esempio con `@tailrec` , otterremo il seguente messaggio di errore: `could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position`

Ricorsione di coda

Nella ricorsione di coda, si eseguono prima i calcoli e quindi si esegue la chiamata ricorsiva, passando i risultati del passaggio corrente al successivo passo ricorsivo.

```
def fact_with_tailrec(i : Int) : Long = {
   @tailrec
   def fact_inside(i : Int, sum: Long) : Long = {
      if(i <= 1) sum
      else fact_inside(i-1,sum*i)
   }
   fact_inside(i,1)
}

println(fact_with_tailrec(5))
```

Al contrario, la traccia dello stack per il fattoriale ricorsivo della coda ha il seguente aspetto:

```
(fact_with_tailrec 5)
(fact_inside 5 1)
(fact_inside 4 5)
(fact_inside 3 20)
(fact_inside 2 60)
(fact_inside 1 120)
```

C'è solo la necessità di tenere traccia della stessa quantità di dati per ogni chiamata a `fact_inside` perché la funzione sta semplicemente restituendo il valore che ha ottenuto fino in cima. Ciò significa che anche se viene chiamato `fact_with_tail 1000000` , è necessaria solo la stessa quantità di spazio di `fact_with_tail 3` . Questo non è il caso con la chiamata non ricorsiva alla coda e in quanto tali valori di grandi dimensioni possono causare un overflow dello stack.

Ricorsione senza pila con trampolino (scala.util.control.TailCalls)

È molto comune ottenere un errore `StackOverflowError` mentre si chiama la funzione ricorsiva. La libreria standard di Scala offre TailCall per evitare l'overflow dello stack utilizzando oggetti heap e continuazioni per memorizzare lo stato locale della ricorsione.

Due esempi dallo scaladoc di TailCalls

```
import scala.util.control.TailCalls._

def isEven(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(true) else tailcall(isOdd(xs.tail))

def isOdd(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(false) else tailcall(isEven(xs.tail))

// Does this List contain an even number of elements?
isEven((1 to 100000).toList).result

def fib(n: Int): TailRec[Int] =
  if (n < 2) done(n) else for {
    x <- tailcall(fib(n - 1))
    y <- tailcall(fib(n - 2))
  } yield (x + y)
```

```
// What is the 40th entry of the Fibonacci series?
fib(40).result
```

Leggi ricorsione online: https://riptutorial.com/it/scala/topic/3889/ricorsione

Capitolo 44: Riflessione

Examples

Caricamento di una lezione utilizzando la riflessione

```
import scala.reflect.runtime.universe._
val mirror = runtimeMirror(getClass.getClassLoader)
val module = mirror.staticModule("org.data.TempClass")
```

Leggi Riflessione online: https://riptutorial.com/it/scala/topic/5824/riflessione

Capitolo 45: Scala.js

introduzione

`Scala.js` è una porta di `Scala` che compila in `JavaScript` , che alla fine funzionerà all'esterno della `JVM` . Ha vantaggi come forte tipizzazione, ottimizzazione del codice in fase di compilazione, piena interoperabilità con le librerie JavaScript.

Examples

console.log in Scala.js

```
println("Hello Scala.js") // In ES6: console.log("Hello Scala.js");
```

Funzioni di freccia grassa

```
val lastNames = people.map(p => p.lastName)
// Or shorter:
val lastNames = people.map(_.lastName)
```

Classe semplice

```
class Person(val firstName: String, val lastName: String) {
  def fullName(): String =
    s"$firstName $lastName"
}
```

collezioni

```
val personMap = Map(
  10 -> new Person("Roger", "Moore"),
  20 -> new Person("James", "Bond")
)
val names = for {
  (key, person) <- personMap
  if key > 15
} yield s"$key = ${person.firstName}"
```

Manipolazione del DOM

```
import org.scalajs.dom
import dom.document

def appendP(target: dom.Node, text: String) = {
  val pNode = document.createElement("p")
  val textNode = document.createTextNode(text)
  pNode.appendChild(textNode)
```

```
  target.appendChild(pNode)
}
```

Utilizzo con SBT

Dipendenza da Sbt

```
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1" // (Triple %%%)
```

In esecuzione

```
sbt run
```

In esecuzione con la compilazione continua:

```
sbt ~run
```

Compilare a un singolo file JavaScript:

```
sbt fastOptJS
```

Leggi Scala.js online: https://riptutorial.com/it/scala/topic/9426/scala-js

Capitolo 46: scaladoc

Sintassi

- Passa sopra metodi, campi, classi o pacchetti.
- Inizia con `/**`
- Ogni riga ha una procedura `*` iniziale con i commenti
- Termina con `*/`

Parametri

Parametro	Dettagli
Classe specifica	_
`@constructor detail`	Spiega il principale costruttore della classe
Metodo specifico	_
`@return detail`	Dettagli su ciò che viene restituito sul metodo.
Metodo, Costruttore e / o Tag di classe	_
`@param x detail`	Dettagli sul parametro value `x` su un metodo o costruttore.
`@tparam x detail`	Dettagli sul parametro type `x` su un metodo o costruttore.
`@throws detail`	Quali eccezioni possono essere lanciate.
uso	_
`@see detail`	Riferimenti ad altre fonti di informazione.
`@note detail`	Aggiunge una nota per le condizioni pre o post, o altre restrizioni o aspettative importanti.
`@example detail`	Fornisce codice di esempio o documentazione di esempio correlata.
`@usecase detail`	Fornisce una definizione di metodo semplificata per quando la definizione del metodo completo è troppo complessa o rumorosa.
Altro	_
`@author detail`	Fornisce informazioni sull'autore di quanto segue.

Parametro	Dettagli
`@version detail`	Fornisce la versione a cui appartiene questa parte.
`@deprecated detail`	Contrassegna la seguente entità come deprecata.

Examples

Semplice metodo Scaladoc

```
/**
  * Explain briefly what method does here
  * @param x Explain briefly what should be x and how this affects the method.
  * @param y Explain briefly what should be y and how this affects the method.
  * @return Explain what is returned from execution.
  */
def method(x: Int, y: String): Option[Double] = {
  // Method content
}
```

Leggi scaladoc online: https://riptutorial.com/it/scala/topic/4518/scaladoc

Capitolo 47: scalaz

introduzione

Scalaz è una libreria Scala per la programmazione funzionale.

Fornisce strutture dati puramente funzionali per integrare quelle della libreria standard di Scala. Definisce un insieme di classi di tipi fondamentali (es. `Functor` , `Monad`) e istanze corrispondenti per un gran numero di strutture di dati.

Examples

ApplyUsage

```
import scalaz._
import Scalaz._

scala> Apply[Option].apply2(some(1), some(2))((a, b) => a + b)
res0: Option[Int] = Some(3)

scala> val intToString: Int => String = _.toString

scala> Apply[Option].ap(1.some)(some(intToString))
res1: Option[String] = Some(1)

scala> Apply[Option].ap(none)(some(intToString))
res2: Option[String] = None

scala> val double: Int => Int = _ * 2

scala> Apply[List].ap(List(1, 2, 3))(List(double))
res3: List[Int] = List(2, 4, 6)

scala> :kind Apply
scalaz.Apply's kind is X[F[A]]
```

FunctorUsage

```
import scalaz._
import Scalaz._
scala> val len: String => Int = _.length
len: String => Int = $$Lambda$1164/969820333@7e758f40

scala> Functor[Option].map(Some("foo"))(len)
res0: Option[Int] = Some(3)

scala> Functor[Option].map(None)(len)
res1: Option[Int] = None

scala> Functor[List].map(List("qwer", "adsfg"))(len)
res2: List[Int] = List(4, 5)
```

```
scala> :kind Functor
scalaz.Functor's kind is X[F[A]]
```

ArrowUsage

```
import scalaz._
import Scalaz._
scala> val plus1 = (_: Int) + 1
plus1: Int => Int = $$Lambda$1167/11131119649@6a6bfd97

scala> val plus2 = (_: Int) + 2
plus2: Int => Int = $$Lambda$1168/9243295480@6bbe050f

scala> val rev = (_: String).reverse
rev: String => String = $$Lambda$1227/1278001332@72685b74

scala> plus1.first apply (1, "abc")
res0: (Int, String) = (2,abc)

scala> plus1.second apply ("abc", 2)
res1: (String, Int) = (abc,3)

scala> rev.second apply (1, "abc")
res2: (Int, String) = (1,cba)

scala> plus1 *** rev apply(7, "abc")
res3: (Int, String) = (8,cba)

scala> plus1 &&& plus2 apply 7
res4: (Int, Int) = (8,9)

scala> plus1.product apply (1, 2)
res5: (Int, Int) = (2,3)

scala> :kind Arrow
scalaz.Arrow's kind is X[F[A1,A2]]
```

Leggi scalaz online: https://riptutorial.com/it/scala/topic/9893/scalaz

Capitolo 48: Scopo

introduzione

Scope on Scala definisce da dove si può accedere a un valore (`def` , `val` , `var` o `class`).

Sintassi

- dichiarazione
- dichiarazione privata
- [questa] dichiarazione privata
- dichiarazione privata [fromWhere]
- dichiarazione protetta
- protetta [daWhere] dichiarazione

Examples

Ambito pubblico (predefinito)

Per impostazione predefinita, l'ambito è `public` , il valore è accessibile da qualsiasi luogo.

```
package com.example {
  class FooClass {
    val x = "foo"
  }
}

package an.other.package {
  class BarClass {
    val foo = new com.example.FooClass
    foo.x // <- Accessing a public value from another package
  }
}
```

Un ambito privato

Quando l'ambito è privato, è possibile accedervi solo dalla classe corrente o da altre istanze della classe corrente.

```
package com.example {
  class FooClass {
    private val x = "foo"
    def aFoo(otherFoo: FooClass) {
      otherFoo.x // <- Accessing from another instance of the same class
    }
  }
  class BarClass {
    val f = new FooClass
```

```
    f.x // <- This will not compile
  }
}
```

Un ambito specifico del pacchetto privato

È possibile specificare un pacchetto in cui è possibile accedere al valore privato.

```
package com.example {
  class FooClass {
    private val x = "foo"
    private[example] val y = "bar"
  }
  class BarClass {
    val f = new FooClass
    f.x // <- Will not compile
    f.y // <- Will compile
  }
}
```

Scopo privato dell'oggetto

L'ambito più restrittivo è l'ambito *"oggetto-privato"* , che consente di accedere a tale valore solo dalla stessa istanza dell'oggetto.

```
class FooClass {
  private[this] val x = "foo"
  def aFoo(otherFoo: FooClass) = {
    otherFoo.x // <- This will not compile, accessing x outside the object instance
  }
}
```

Ambito protetto

L'ambito protetto consente l'accesso al valore da qualsiasi sottoclasse della classe corrente.

```
class FooClass {
  protected val x = "foo"
}
class BarClass extends FooClass {
  val y = x // It is a subclass instance, will compile
}
class ClassB {
  val f = new FooClass
  f.x // <- This will not compile
}
```

Ambito protetto da pacchetto

L'ambito protetto del pacchetto consente di accedere al valore solo da qualsiasi sottoclasse in un pacchetto specifico.

```
package com.example {
  class FooClass {
    protected[example] val x = "foo"
  }
  class ClassB extends FooClass {
    val y = x // It's in the protected scope, will compile
  }
}
package com {
  class BarClass extends com.example.FooClass {
    val y = x // <- Outside the protected scope, will not compile
  }
}
```

Leggi Scopo online: https://riptutorial.com/it/scala/topic/9705/scopo

Capitolo 49: Se espressioni

Examples

Basic If Expressions

In Scala (a differenza di Java e della maggior parte delle altre lingue), `if` è **un'espressione** invece di *un'istruzione* . Indipendentemente da ciò, la sintassi è identica:

```
if(x < 1984) {
   println("Good times")
} else if(x == 1984) {
   println("The Orwellian Future begins")
} else {
   println("Poor guy...")
}
```

L'implicazione di `if` essere un'espressione è che è possibile assegnare il risultato della valutazione dell'espressione a una variabile:

```
val result = if(x > 0) "Greater than 0" else "Less than or equals 0"
\\ result: String = Greater than 0
```

Sopra vediamo che l'espressione `if` viene valutata e il `result` è impostato su quel valore risultante.

Il tipo restituito di un'espressione `if` è il **supertipo** di tutti i rami logici. Ciò significa che per questo esempio il tipo restituito è una `String` . Poiché non tutti `if` le espressioni restituiscono un valore (come ad esempio un `if` dichiarazione che non ha `else` di logica ramo), è possibile che il tipo di ritorno è `Any` :

```
val result = if(x > 0) "Greater than 0"
// result: Any = Greater than 0
```

Se non è possibile restituire alcun valore (ad esempio se vengono utilizzati solo effetti collaterali come `println` all'interno dei rami logici), il tipo di ritorno sarà `Unit` :

```
val result = if(x > 0) println("Greater than 0")
// result: Unit = ()
```

`if` espressioni in Scala sono simili a come funziona l' operatore ternario in Java . A causa di questa somiglianza, non esiste un operatore di questo tipo in Scala: sarebbe ridondante.

> Graffe possono essere omessi in `if` espressione se il contenuto è una singola riga.

Leggi Se espressioni online: https://riptutorial.com/it/scala/topic/4171/se-espressioni

Capitolo 50: sincronizzato

Sintassi

- `objectToSynchronizeOn.synchronized { /* code to run */}`
- `synchronized {/* code to run, can be suspended with wait */}`

Examples

sincronizzare su un oggetto

`synchronized` è un costrutto di concorrenza di basso livello che può aiutare a impedire che più thread accedano alle stesse risorse. Introduzione per JVM utilizzando il linguaggio Java .

```
anInstance.synchronized {
  // code to run when the intristic lock on `anInstance` is acquired
  // other thread cannot enter concurrently unless `wait` is called on `anInstance` to suspend
  // other threads can continue of the execution of this thread if they `notify` or
`notifyAll` `anInstance`'s lock
}
```

In caso di `object` s potrebbe sincronizzarsi sulla classe dell'oggetto, non sull'istanza singleton.

sincronizzare implicitamente su questo

```
 /* within a class, def, trait or object, but not a constructor */
 synchronized {
   /* code to run when an intrisctic lock on `this` is acquired */
   /* no other thread can get the this lock unless execution is suspended with
    * `wait` on `this`
    */
 }
```

Leggi sincronizzato online: https://riptutorial.com/it/scala/topic/3371/sincronizzato

Capitolo 51: Singoli tipi di metodo astratto (tipi SAM)

Osservazioni

I singoli metodi astratti sono tipi, introdotti in Java 8 , che hanno esattamente un membro astratto.

Examples

Lambda Sintassi

NOTA: Questo è disponibile solo in Scala 2.12+ (e nelle recenti versioni 2.11.x con i `-Xexperimental -Xfuture` compilatore `-Xexperimental -Xfuture`)

Un tipo SAM può essere implementato utilizzando un lambda:

2.11.8

```
trait Runnable {
  def run(): Unit
}

val t: Runnable = () => println("foo")
```

Il tipo può facoltativamente avere altri membri non astratti:

2.11.8

```
trait Runnable {
  def run(): Unit
  def concrete: Int = 42
}

val t: Runnable = () => println("foo")
```

Leggi Singoli tipi di metodo astratto (tipi SAM) online:
https://riptutorial.com/it/scala/topic/3664/singoli-tipi-di-metodo-astratto--tipi-sam-

Capitolo 52: Sovraccarico dell'operatore

Examples

Definizione degli operatori Infix personalizzati

Negli operatori di Scala (come `+` , `-` , `*` , `++` , ecc.) Sono solo metodi. Ad esempio, `1 + 2` può essere scritto come `1.+(2)` . Questi tipi di metodi sono chiamati *"operatori infissi"* .

Ciò significa che i metodi personalizzati possono essere definiti sui tuoi tipi, riutilizzando questi operatori:

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
}
```

Questi operatori definiti come metodi possono essere utilizzati in questo modo:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val b = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))

// could also be written a.+(b)
val sum = a + b
```

Nota che gli operatori infissi possono avere un solo argomento; l'oggetto prima che l'operatore chiamerà il proprio operatore sull'oggetto dopo l'operatore. Qualsiasi metodo Scala con un singolo argomento può essere utilizzato come operatore infisso.

> Questo dovrebbe essere usato con il parcimony. Generalmente è considerato una buona pratica solo se il proprio metodo fa esattamente ciò che ci si aspetterebbe da quell'operatore. In caso di dubbio, usa un nome più prudente, come `add` invece di `+` .

Definizione di operatori unari personalizzati

Gli operatori unari possono essere definiti anteponendo l'operatore a `unary_` . Gli operatori `unary_+` sono limitati a `unary_+` , `unary_-` , `unary_!` e `unary_~` :

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
```

```
  def unary_- = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) * -1

    new Matrix(rows, cols, newData)
  }
}
```

L'operatore unario può essere utilizzato come segue:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val negA = -a
```

> Questo dovrebbe essere usato con il parcimony. Sovraccaricare un operatore unario con una definizione che non è quella che ci si aspetterebbe può portare alla confusione del codice.

Leggi Sovraccarico dell'operatore online: https://riptutorial.com/it/scala/topic/2271/sovraccarico-dell-operatore

Capitolo 53: Symbol letterali

Osservazioni

Scala ha un concetto di **simboli** - le stringhe che sono *internate* , ovvero: due simboli con lo stesso nome (la stessa sequenza di caratteri), al contrario delle stringhe, si riferiranno allo stesso oggetto durante l'esecuzione.

I simboli sono una caratteristica di molte lingue: Lisp, Ruby ed Erlang e altro, tuttavia in Scala sono di uso relativamente piccolo. Buona caratteristica di avere comunque.

Uso:

Ogni letterale inizio con una singola citazione ' , seguito da una o più cifre, lettere o sotto-punteggi _ è un simbolo letterale. Il primo carattere è un'eccezione in quanto non può essere una cifra.

Buone definizioni:

```
'ATM
'IPv4
'IPv6
'map_to_operations
'data_format_2006

// Using the `Symbol.apply` method

Symbol("hakuna matata")
Symbol("To be or not to be that is a question")
```

Cattive definizioni:

```
'8'th_division
'94_pattern
'bad-format
```

Examples

Sostituzione di stringhe nelle clausole case

Supponiamo di avere più origini dati che includono *database, file, prompt* e *elenco argomenti* . A seconda della fonte scelta cambiamo il nostro approccio:

```
def loadData(dataSource: Symbol): Try[String] = dataSource match {
  case 'database => loadDatabase() // Loading data from database
  case 'file =>  loadFile() // Loading data from file
  case 'prompt => askUser() // Asking user for data
  case 'argumentList => argumentListExtract() // Accessing argument list for data
  case _ => Failure(new Exception("Unsupported data source"))
}
```

Potremmo usare molto bene `String` al posto di `Symbol` . Non l'abbiamo fatto, perché nessuna delle caratteristiche delle stringhe è utile in questo contesto.

Questo rende il codice più semplice e meno soggetto a errori.

Leggi Symbol letterali online: https://riptutorial.com/it/scala/topic/6419/symbol-letterali

Capitolo 54: Test con ScalaCheck

introduzione

ScalaCheck è una libreria scritta in Scala e utilizzata per test automatici basati su proprietà di programmi Scala o Java. ScalaCheck è stato originariamente ispirato alla libreria Haskell QuickCheck, ma si è anche avventurato in proprio.

ScalaCheck non ha dipendenze esterne oltre al runtime di Scala e funziona alla grande con sbt, lo strumento di compilazione Scala. Inoltre è completamente integrato nei framework di test ScalaTest e specs2.

Examples

Scalacheck con scalatest e messaggi di errore

Esempio di utilizzo dello scalacheck con scalatest. Di seguito abbiamo quattro test:

- "show pass esempio" - passa
- "mostrare semplice esempio senza messaggio di errore personalizzato" - proprio messaggio non inviato, senza dettagli, `&&` operatore booleano viene utilizzato
- "mostra un esempio con messaggi di errore sull'argomento" - messaggio di errore sull'argomento (`"argument" |:` :) metodo Props.all viene utilizzato al posto di `&&`
- "mostra un esempio con messaggi di errore sul comando" - messaggio di errore sul comando (`"command" |:` :) Il metodo Props.all viene utilizzato al posto di `&&`

```
import org.scalatest.prop.Checkers
import org.scalatest.{Matchers, WordSpecLike}

import org.scalacheck.Gen._
import org.scalacheck.Prop._
import org.scalacheck.Prop

object Splitter {
  def splitLineByColon(message: String): (String, String) = {
    val (command, argument) = message.indexOf(":") match {
      case -1 =>
        (message, "")
      case x: Int =>
        (message.substring(0, x), message.substring(x + 1))
    }
    (command.trim, argument.trim)
  }

  def splitLineByColonWithBugOnCommand(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim + 2, argument.trim)
  }

  def splitLineByColonWithBugOnArgument(message: String): (String, String) = {
```

```
    val (command, argument) = splitLineByColon(message)
    (command.trim, argument.trim + 2)
  }
}

class ScalaCheckSpec extends WordSpecLike with Matchers with Checkers {

  private val COMMAND_LENGTH = 4

  "ScalaCheckSpec " should {
```

```
    "show pass example" in {
      check {
        Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
          (chars, expArgument) =>
            val expCommand = new String(chars.toArray)
            val line = s"$expCommand:$expArgument"
            val (c, p) = Splitter.splitLineByColon(line)
            Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
        }

      }
    }
```

```
"show simple example without custom error message " in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        c === expCommand && expArgument === p
    }

  }
}
```

```
"show example with error messages on argument" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

```
"show example with error messages on command" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnCommand(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
```

```
    }

  }
}
```

L'output (frammenti):

```
[info] - should show example // passed
[info] - should show simple example without custom error message  *** FAILED ***
[info]    (ScalaCheckSpec.scala:73)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:73)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info] - should show example with error messages on argument *** FAILED ***
[info]    (ScalaCheckSpec.scala:86)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:86)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "" but got "2"
[info]       argument
[info] - should show example with error messages on command *** FAILED ***
[info]    (ScalaCheckSpec.scala:99)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:99)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "2" but got ""
[info]       command
```

Leggi Test con ScalaCheck online: https://riptutorial.com/it/scala/topic/9430/test-con-scalacheck

Capitolo 55: Test con ScalaTest

Examples

Ciao World Spec Test

Creare una classe di test nella directory `src/test/scala` , in un file denominato `HelloWorldSpec.scala` . Metti questo nel file:

```
import org.scalatest.{FlatSpec, Matchers}

class HelloWorldSpec extends FlatSpec with Matchers {

  "Hello World" should "not be an empty String" in {
      val helloWorld = "Hello World"
      helloWorld should not be ("")
  }
}
```

- Questo esempio utilizza `FlatSpec` e `Matchers` , che fanno parte della libreria ScalaTest.
- `FlatSpec` consente di `FlatSpec` test nello stile **Behaviour Driven Development (BDD)** . In questo stile, una frase viene utilizzata per descrivere il comportamento previsto di una determinata unità di codice. Il test conferma che il codice aderisce a tale comportamento. Vedere la documentazione per ulteriori informazioni .

Cheat sheet Test Spec

Impostare

I test seguenti utilizzano questi valori per gli esempi.

```
val helloWorld = "Hello World"
val helloWorldCount = 1
val helloWorldList = List("Hello World", "Bonjour Le Monde")
def sayHello = throw new IllegalStateException("Hello World Exception")
```

Controlla il tipo

Per verificare il tipo per una data `val` :

```
helloWorld shouldBe a [String]
```

Notare che qui le parentesi sono usate per ottenere il tipo `String` .

Uguale controllo

Per testare l'uguaglianza:

```
helloWorld shouldEqual "Hello World"
helloWorld should === ("Hello World")
helloWorldCount shouldEqual 1
helloWorldCount shouldBe 1
helloWorldList shouldEqual List("Hello World", "Bonjour Le Monde")
helloWorldList === List("Hello World", "Bonjour Le Monde")
```

Controllo non uguale

Per testare la disuguaglianza:

```
helloWorld should not equal "Hello"
helloWorld !== "Hello"
helloWorldCount should not be 5
helloWorldList should not equal List("Hello World")
helloWorldList !== List("Hello World")
helloWorldList should not be empty
```

Controllo della lunghezza

Per verificare la lunghezza e / o la dimensione:

```
helloWorld should have length 11
helloWorldList should have size 2
```

Controllo delle eccezioni

Per verificare il tipo e il messaggio di un'eccezione:

```
val exception = the [java.lang.IllegalStateException] thrownBy {
  sayHello
}
exception.getClass shouldEqual classOf[java.lang.IllegalStateException]
exception.getMessage should include ("Hello World")
```

Include la libreria ScalaTest con SBT

Utilizzando SBT per gestire la dipendenza della libreria , aggiungere questo a `build.sbt` :

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.0"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.0" % "test"
```

Maggiori dettagli possono essere trovati sul sito di ScalaTest .

Leggi Test con ScalaTest online: https://riptutorial.com/it/scala/topic/5506/test-con-scalatest

Capitolo 56: Tipi di auto

Sintassi

- trait Type {selfId => / *altri membri possono fare riferimento a* `selfId` *nel caso in cui* `this` *significhi qualcosa /}*
- trait Type {selfId: OtherType => / * altri membri possono usare `selfId` e sarà di tipo `OtherType` * /
- tratto Tipo {selfId: OtherType1 con OtherType2 => / * `selfId` è di tipo `OtherType1` e `OtherType2` * /

Osservazioni

Spesso usato con il motivo a torta.

Examples

Semplice esempio di auto-tipo

I tipi di auto possono essere usati in tratti e classi per definire i vincoli sulle classi concrete con cui è mescolato. È anche possibile utilizzare un identificatore diverso per `this` usando questa sintassi (utile quando l'oggetto esterno deve essere referenziato da un oggetto interno).

Supponi di voler memorizzare alcuni oggetti. Per questo, si creano interfacce per lo storage e si aggiungono valori a un contenitore:

```
trait Container[+T] {
  def add(o: T): Unit
}

trait PermanentStorage[T] {
  /* Constraint on self type: it should be Container
   * we can refer to that type as `identifier`, usually `this` or `self`
   * or the type's name is used. */
  identifier: Container[T] =>

  def save(o: T): Unit = {
    identifier.add(o)
    //Do something to persist too.
  }
}
```

In questo modo quelli non si trovano nella stessa gerarchia di oggetti, ma `PermanentStorage` non può essere implementato senza implementare anche `Container` .

Leggi Tipi di auto online: https://riptutorial.com/it/scala/topic/4639/tipi-di-auto

Capitolo 57: Tipo di inferenza

Examples

Inferenza di tipo locale

Scala ha un potente meccanismo di inferenza del tipo integrato nella lingua. Questo meccanismo è definito come "Inferenza di tipo locale":

```
val i = 1 + 2                 // the type of i is Int
val s = "I am a String"       // the type of s is String
def squared(x : Int) = x*x    // the return type of squared is Int
```

Il compilatore può inferire il tipo di variabili dall'espressione di inizializzazione. Allo stesso modo, il tipo di metodo restituito può essere omesso, poiché sono equivalenti al tipo restituito dal corpo del metodo. Gli esempi precedenti sono equivalenti alle seguenti, dichiarazioni di tipo esplicite:

```
val i: Int = 1 + 2
val s: String = "I am a String"
def squared(x : Int): Int = x*x
```

Tipo di inferenza e generici

Il compilatore Scala può anche dedurre parametri di tipo quando vengono chiamati metodi polimorfici o quando vengono create istanze di classi generiche:

```
case class InferedPair[A, B](a: A, b: B)

val pairFirstInst = InferedPair("Husband", "Wife")  //type is InferedPair[String, String]

// Equivalent, with type explicitly defined
val pairSecondInst: InferedPair[String, String]
                      = InferedPair[String, String]("Husband", "Wife")
```

La precedente forma di inferenza del tipo è simile all'operatore Diamond , introdotto in Java 7.

Limitazioni all'inferenza

Esistono scenari in cui l'inferenza del tipo di Scala non funziona. Ad esempio, il compilatore non può inferire il tipo di parametri del metodo:

```
def add(a, b) = a + b  // Does not compile
def add(a: Int, b: Int) = a + b // Compiles
def add(a: Int, b: Int): Int = a + b // Equivalent expression, compiles
```

Il compilatore non può inferire il tipo di ritorno dei metodi ricorsivi:

```
// Does not compile
def factorial(n: Int) = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
// Compiles
def factorial(n: Int): Int = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
```

Prevenire il non inferire

Basato su questo post del blog .

Supponi di avere un metodo come questo:

```
  def get[T]: Option[T] = ???
```

Quando si tenta di chiamarlo senza specificare il parametro generico, `Nothing` viene inferito, che non è molto utile per un'implementazione effettiva (e il suo risultato non è utile). Con la seguente soluzione del `NotNothing` contesto vincolato può impedire con il metodo senza specificare il tipo previsto (in questo esempio `RuntimeClass` è esclusa anche da per `ClassTags` non `Nothing` , ma `RuntimeClass` è dedotta):

```
@implicitNotFound("Nothing was inferred")
sealed trait NotNothing[-T]

object NotNothing {
  implicit object notNothing extends NotNothing[Any]
  //We do not want Nothing to be inferred, so make an ambigous implicit
  implicit object `\n The error is because the type parameter was resolved to Nothing` extends
NotNothing[Nothing]
  //For classtags, RuntimeClass can also be inferred, so making that ambigous too
  implicit object `\n The error is because the type parameter was resolved to RuntimeClass`
extends NotNothing[RuntimeClass]
}

object ObjectStore {
  //Using context bounds
  def get[T: NotNothing]: Option[T] = {
    ???
  }

  def newArray[T](length: Int = 10)(implicit ct: ClassTag[T], evNotNothing: NotNothing[T]):
Option[Array[T]] = ???
}
```

Esempio di utilizzo:

```
object X {
  //Fails to compile
  //val nothingInferred = ObjectStore.get

  val anOption = ObjectStore.get[String]
  val optionalArray = ObjectStore.newArray[AnyRef]()

  //Fails to compile
  //val runtimeClassInferred = ObjectStore.newArray()
}
```

Leggi Tipo di inferenza online: https://riptutorial.com/it/scala/topic/4918/tipo-di-inferenza

Capitolo 58: Tipo Parametrizzazione (Generics)

Examples

Il tipo di opzione

Un bell'esempio di un tipo parametrizzato è il tipo di opzione . È essenzialmente solo la seguente definizione (con molti altri metodi definiti sul tipo):

```
sealed abstract class Option[+A] {
  def isEmpty: Boolean
  def get: A

  final def fold[B](ifEmpty: => B)(f: A => B): B =
    if (isEmpty) ifEmpty else f(this.get)

  // lots of methods...
}

case class Some[A](value: A) extends Option[A] {
  def isEmpty = false
  def get = value
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

Possiamo anche vedere che questo ha un metodo parametrico, `fold` , che restituisce qualcosa di tipo `B`

Metodi parametrizzati

Il tipo di ritorno di un metodo può dipendere dal *tipo* del parametro. In questo esempio, `x` è il parametro, `A` è il *tipo* di `x` , che è noto come *parametro type* .

```
def f[A](x: A): A = x

f(1)          // 1
f("two")      // "two"
f[Float](3)   // 3.0F
```

Scala utilizzerà l' inferenza di tipo per determinare il tipo di ritorno, che vincola i metodi che possono essere chiamati sul parametro. Pertanto, è necessario prestare attenzione: il seguente è un errore in fase di compilazione poiché `*` non è definito per ogni tipo `A` :

```
def g[A](x: A): A = 2 * x  // Won't compile
```

Definire la lista di Ints

```
trait IntList { ... }

class Cons(val head: Int, val tail: IntList) extends IntList { ... }

class Nil extends IntList { ... }
```

ma cosa succede se abbiamo bisogno di definire la lista di Boolean, Double etc?

Definire una lista generica

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: [T], val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

class Nil[T] extends List[T] {
  def isEmpty: Boolean = true

  def head: Nothing = throw NoSuchElementException("Nil.head")

  def tail: Nothing = throw NoSuchElementException("Nil.tail")
}
```

Leggi Tipo Parametrizzazione (Generics) online: https://riptutorial.com/it/scala/topic/782/tipo-parametrizzazione--generics-

Capitolo 59: Tipo Varianza

Examples

covarianza

Il simbolo `+` indica un parametro di tipo come *covariante* - qui diciamo che " `Producer` is covariant on `A` ":

```
trait Producer[+A] {
  def produce: A
}
```

Un parametro di tipo covariante può essere pensato come un tipo di "output". Contrassegnare `A` come covariante afferma che `Producer[X] <: Producer[Y]` condizione che `X <: Y` Ad esempio, un `Producer[Cat]` è un `Producer[Animal]` valido `Producer[Animal]` , poiché tutti i gatti prodotti sono anche animali validi.

Un parametro di tipo covariante non può apparire in posizione controvariante (input). L'esempio seguente non verrà compilato poiché stiamo affermando che `Co[Cat] <: Co[Animal]` , ma `Co[Cat]` ha `def handle(a: Cat): Unit` che non può gestire alcun `Animal` come richiesto da `Co[Animal]` !

```
trait Co[+A] {
  def produce: A
  def handle(a: A): Unit
}
```

Un approccio per gestire questa restrizione consiste nell'usare parametri di tipo delimitati dal parametro di tipo covariante. Nel seguente esempio, sappiamo che `B` è un supertipo di `A` Pertanto, data `Option[X] <: Option[Y]` per `X <: Y` , sappiamo che l' `Option[X]` 's ha `def getOrElse[B >: X](b: => B): B` può accettare qualsiasi supertipo di `X` - che include i supertipi di `Y` come richiesto `Option[Y]` :

```
trait Option[+A] {
  def getOrElse[B >: A](b: => B): B
}
```

invarianza

Di default tutti i parametri di tipo sono invarianti - dato il `trait A[B]` , diciamo che " `A` è invariante su `B` ". Ciò significa che date due parametrizzazioni `A[Cat]` e `A[Animal]` , non asseriamo nessuna relazione sub / superclasse tra questi due tipi - non ritiene che `A[Cat] <: A[Animal]` né che `A[Cat] >: A[Animal]` indipendentemente dal rapporto tra `Cat` e `Animal` .

Le annotazioni di varianza ci forniscono un mezzo per dichiarare tale relazione e impongono regole sull'uso dei parametri di tipo in modo che la relazione rimanga valida.

controvarianza

Il simbolo `-` contrassegna un parametro di tipo come *controverso* - qui diciamo che "Il `Handler` è controverso su `A` ":

```
trait Handler[-A] {
  def handle(a: A): Unit
}
```

Un parametro di tipo controvariante può essere pensato come un tipo di "input". Contrassegnare `A` come controvariante afferma che `Handler[X] <: Handler[Y]` fornito `X >: Y` Ad esempio, un `Handler[Animal]` è un `Handler[Cat]` valido `Handler[Cat]` , poiché un `Handler[Animal]` deve anche gestire i gatti.

Un parametro di tipo controvariante non può apparire nella posizione covariante (uscita). L'esempio seguente non verrà compilato poiché stiamo affermando che `Contra[Animal] <: Contra[Cat]` , tuttavia un `Contra[Animal]` ha `def produce: Animal` che non è garantito per produrre gatti come richiesto da `Contra[Cat]` !

```
trait Contra[-A] {
   def handle(a: A): Unit
   def produce: A
}
```

Attenzione però: ai fini della risoluzione dell'overloading, la contravarianza inverte anche in modo controintuitivo la specificità di un tipo sul parametro di tipo controvariante - `Handler[Animal]` è considerato "più specifico" di `Handler[Cat]` .

Poiché non è possibile sovraccaricare i metodi sui parametri di tipo, questo comportamento diventa generalmente problematico solo quando si risolvono gli argomenti impliciti. Nell'esempio seguente `ofCat` non verrà mai utilizzato, poiché il tipo di ritorno `ofAnimal` è più specifico:

```
implicit def ofAnimal: Handler[Animal] = ???
implicit def ofCat: Handler[Cat] = ???

implicitly[Handler[Cat]].handle(new Cat)
```

Questo comportamento è attualmente previsto per cambiare in dotty , ed è perché (ad esempio) `scala.math.Ordering` è invariante sul suo parametro tipo `T` Una soluzione alternativa consiste nel rendere invariabile la classe di tipizzazione e digitare la parametrizzazione della definizione implicita nell'evento in cui si desidera che venga applicata alle sottoclassi di un determinato tipo:

```
trait Person
object Person {
  implicit def ordering[A <: Person]: Ordering[A] = ???
}
```

Covarianza di una collezione

Poiché le raccolte sono tipicamente covarianti nel loro tipo di elemento *, una raccolta di un sottotipo può essere passata dove è previsto un super tipo:

```
trait Animal { def name: String }
case class Dog(name: String) extends Animal

object Animal {
  def printAnimalNames(animals: Seq[Animal]) = {
    animals.foreach(animal => println(animal.name))
  }
}

val myDogs: Seq[Dog] = Seq(Dog("Curly"), Dog("Larry"), Dog("Moe"))

Animal.printAnimalNames(myDogs)
// Curly
// Larry
// Moe
```

Potrebbe non sembrare magico, ma il fatto che un `Seq[Dog]` sia accettato da un metodo che si aspetta un `Seq[Animal]` è l'intero concetto di un tipo di tipo superiore (qui: `Seq`) che è covariante nel suo parametro tipo.

* Un controesempio è il set della libreria standard

Covarianza su un tratto invariante

C'è anche un modo per avere un singolo metodo per accettare un argomento covariante, invece di avere l'intero tratto covariante. Questo potrebbe essere necessario perché vorresti usare `T` in una posizione controvariante, ma mantenerlo comunque covariante.

```
trait LocalVariance[T]{
  /// ??? throws a NotImplementedError
  def produce: T = ???
  // the implicit evidence provided by the compiler confirms that S is a
  // subtype of T.
  def handle[S](s: S)(implicit evidence: S <:< T) = {
    // and we can use the evidence to convert s into t.
    val t: T = evidence(s)
    ???
  }
}

trait A {}
trait B extends A {}

object Test {
  val lv = new LocalVariance[A] {}

  // now we can pass a B instead of an A.
  lv.handle(new B {})
}
```

Leggi Tipo Varianza online: https://riptutorial.com/it/scala/topic/1651/tipo-varianza

Capitolo 60: Tratti

Sintassi

- tratto ATIGHT {...}
- classe AClass (...) estende ATrait {...}
- classe AClass estende BClass con ATrait
- classe AClass estende ATrait con BTrait
- classe AClass estende ATrait con BTrait con CTrait
- classe ATrait estende BTrait

Examples

Modifica impilabile con tratti

È possibile utilizzare i tratti per modificare i metodi di una classe, utilizzando i tratti in modo impilabile.

Il seguente esempio mostra come i tratti possono essere impilati. L'ordine dei tratti è importante. Usando un diverso ordine di tratti, si ottiene un comportamento diverso.

```
class Ball {
  def roll(ball : String) = println("Rolling : " + ball)
}

trait Red extends Ball {
  override def roll(ball : String) = super.roll("Red-" + ball)
}

trait Green extends Ball {
  override def roll(ball : String) = super.roll("Green-" + ball)
}

trait Shiny extends Ball {
  override def roll(ball : String) = super.roll("Shiny-" + ball)
}

object Balls {
  def main(args: Array[String]) {
    val ball1 = new Ball with Shiny with Red
    ball1.roll("Ball-1") // Rolling : Shiny-Red-Ball-1

    val ball2 = new Ball with Green with Shiny
    ball2.roll("Ball-2") // Rolling : Green-Shiny-Ball-2
  }
}
```

Nota che `super` è usato per invocare `roll()` metodo `roll()` in entrambi i tratti. Solo in questo modo possiamo ottenere modifiche impilabili. In caso di modifica impilabile, l'ordine di invocazione del metodo è determinato dalla *regola di linearizzazione* .

Nozioni di base sui tratti

Questa è la versione più basilare di un tratto in Scala.

```
trait Identifiable {
  def getIdentifier: String
  def printIndentification(): Unit = println(getIdentifier)
}

case class Puppy(id: String, name: String) extends Identifiable {
  def getIdentifier: String = s"$name has id $id"
}
```

Dal momento che nessuna classe super è dichiarata per tratto `Identifiable` , per impostazione predefinita si estende dalla classe `AnyRef` . Poiché in `Identifiable` viene fornita alcuna definizione per `getIdentifier` , la classe `Puppy` deve implementarla. Tuttavia, `Puppy` eredita l'implementazione di `printIdentification` da `Identifiable` .

Nella REPL:

```
val p = new Puppy("K9", "Rex")
p.getIdentifier  // res0: String = Rex has id K9
p.printIndentification()  // Rex has id K9
```

Risolvere il problema del diamante

Il problema dei diamanti , o ereditarietà multipla, è gestito da Scala usando i Tratti, che sono simili alle interfacce Java. I tratti sono più flessibili delle interfacce e possono includere metodi implementati. Questo rende tratti simili ai mixin in altre lingue.

Scala non supporta l'ereditarietà di più classi, ma un utente può estendere più tratti in una singola classe:

```
trait traitA {
  def name = println("This is the 'grandparent' trait.")
}

trait traitB extends traitA {
  override def name = {
    println("B is a child of A.")
    super.name
  }

}

trait traitC extends traitA {
  override def name = {
    println("C is a child of A.")
    super.name
  }
}

object grandChild extends traitB with traitC
```

```
grandChild.name
```

Qui `grandChild` eredita sia da `traitB` che da `traitC` , che a loro volta ereditano entrambi da `traitA` . L'output (sotto) mostra anche l'ordine di precedenza quando si risolvono quali implementazioni del metodo vengono chiamate per prime:

```
C is a child of A.
B is a child of A.
This is the 'grandparent' trait.
```

Si noti che, quando viene utilizzato `super` per invocare metodi in `class` o `trait` , la regola di linearizzazione entra in gioco per decidere la gerarchia delle chiamate. L'ordine di linearizzazione per `grandChild` sarà:

> grandChild -> traitC -> traitB -> traitA -> AnyRef -> Any

Di seguito è un altro esempio:

```
trait Printer {
  def print(msg : String) = println (msg)
}

trait DelimitWithHyphen extends Printer {
  override def print(msg : String) {
    println("-------------")
    super.print(msg)
  }
}

trait DelimitWithStar extends Printer  {
  override def print(msg : String) {
    println("*************")
    super.print(msg)
  }
}

class CustomPrinter extends Printer with DelimitWithHyphen with DelimitWithStar

object TestPrinter{
  def main(args: Array[String]) {
    new CustomPrinter().print("Hello World!")
  }
}
```

Questo programma stampa:

```
*************
-------------
Hello World!
```

La linearizzazione per `CustomPrinter` sarà:

> CustomPrinter -> DelimitWithStar -> DelimitWithHyphen -> Printer -> AnyRef -> Any

linearizzazione

In caso di modifica impilabile , Scala organizza le classi e i tratti in un ordine lineare per determinare la gerarchia delle chiamate al metodo, che è nota come *linearizzazione* . La regola di linearizzazione viene utilizzata *solo* per quei metodi che prevedono l'invocazione del metodo tramite `super()` . Consideriamo questo con un esempio:

```
class Shape {
  def paint (shape: String): Unit = {
    println(shape)
  }
}

trait Color extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Color ")
  }
}

trait Blue extends Color {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Blue ")
  }
}

trait Border extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Border ")
  }
}

trait Dotted extends Border {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Dotted ")
  }
}

class MyShape extends Shape with Dotted with Blue {
  override def paint (shape : String) {
    super.paint(shape)
  }
}
```

La linearizzazione avviene da *dietro in avanti* . In questo caso,

1. La prima `Shape` sarà linearizzata, che assomiglia a:

   ```
   Shape -> AnyRef -> Any
   ```

2. Quindi `Dotted` è linearizzato:

   ```
   Dotted -> Border -> Shape -> AnyRef -> Any
   ```

3. Il prossimo in linea è il `Blue` . La linearizzazione di Normally `Blue` sarà:

   ```
   Blue -> Color -> Shape -> AnyRef -> Any
   ```

perché, nella linearizzazione di `MyShape` fino ad ora (*Passo 2*), `Shape -> AnyRef -> Any` è già apparso. Quindi, è ignorato. Pertanto, la linearizzazione `Blue` sarà:

```
Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any
```

4. Infine, verrà aggiunto `Circle` e l'ordine di linearizzazione finale sarà:

   Cerchio -> Blu -> Colore -> Punteggiato -> Bordo -> Forma -> AnyRef -> Qualsiasi

Questo ordine di linearizzazione decide l'ordine di invocazione dei metodi quando `super` è usato in qualsiasi classe o tratto. Viene invocata la prima implementazione del metodo da destra, nell'ordine di linearizzazione. Se viene `new MyShape().paint("Circle ")` , verrà stampato:

```
Circle with Blue Color with Dotted Border
```

Maggiori informazioni sulla linearizzazione possono essere trovate qui .

Leggi Tratti online: https://riptutorial.com/it/scala/topic/1056/tratti

Capitolo 61: Unità di gestione (misure)

Sintassi

- class Meter (val meter: Double) estende AnyVal
- type Meter = Double

Osservazioni

Si consiglia di utilizzare classi di valore per unità o una libreria dedicata per loro.

Examples

Digita gli alias

```
type Meter = Double
```

Questo semplice approccio presenta seri inconvenienti per la gestione delle unità, poiché ogni altro tipo di `Double` è compatibile con esso:

```
type Second = Double
var length: Meter = 3
val duration: Second = 1
length = duration
length = 0d
```

Tutte le compilazioni di cui sopra, quindi in questo caso le unità possono essere utilizzate solo per marcare i tipi di input / output per i lettori del codice (solo l'intento).

Classi di valore

```
case class Meter(meters: Double) extends AnyVal
case class Gram(grams: Double) extends AnyVal
```

Le classi di valore forniscono un modo sicuro per codificare le unità, anche se richiedono un numero leggermente maggiore di caratteri per utilizzarle:

```
var length = Meter(3)
var weight = Gram(4)
//length = weight //type mismatch; found : Gram required: Meter
```

Estendendo `AnyVal` s, non vi è alcuna penalità di runtime per utilizzarli, a livello di JVM, quelli sono normali tipi primitivi (`Double` s in questo caso).

Nel caso in cui si desideri generare automaticamente altre unità (come `Velocity` aka `MeterPerSecond`

), questo approccio non è il migliore, anche se ci sono librerie che possono essere utilizzate anche in questi casi:

- Squants
- unità
- ScalaQuantity

Leggi Unità di gestione (misure) online: https://riptutorial.com/it/scala/topic/5966/unita-di-gestione--misure-

Capitolo 62: Var, Val e Def

Osservazioni

Poiché `val` sono semanticamente statici, vengono inizializzati "sul posto" ovunque compaiano nel codice. Questo può produrre un comportamento sorprendente e indesiderato se usato in classi e tratti astratti.

Per esempio, diciamo che vorremmo creare un tratto chiamato `PlusOne` che definisca un'operazione di incremento su un `Int` avvolto. Poiché `Int` s è immutabile, il valore più uno è noto all'inizializzazione e non verrà mai modificato in seguito, quindi semanticamente è un valore `val` . Tuttavia, definirlo in questo modo produrrà un risultato inaspettato.

```
trait PlusOne {
    val i:Int

    val incr = i + 1
}

class IntWrapper(val i: Int) extends PlusOne
```

Non importa quale sia il valore `i` si costruisce `IntWrapper` con, chiamando `.incr` sul oggetto restituito sarà sempre tornare 1. Questo perché la val `incr` viene inizializzato *nel tratto,* prima della classe che estende, e in quel momento `i` ha solo il valore di default `0` . (In altre condizioni, potrebbe essere popolato con `Nil` , `null` o un valore predefinito simile).

La regola generale, quindi, è di evitare l'uso di `val` su qualsiasi valore che dipende da un campo astratto. Invece, usa `lazy val` , che non valuta fino a quando non è necessario, o `def` , che valuta ogni volta che viene chiamato. Si noti tuttavia che se il valore `lazy val` è forzato a valutare da una `val` prima che l'inizializzazione sia completata, si verificherà lo stesso errore.

Un violino (scritto in Scala-Js, ma vale lo stesso comportamento) può essere trovato qui.

Examples

Var, Val e Def

var

Una `var` è una variabile di riferimento, simile alle variabili in linguaggi come Java. Oggetti diversi possono essere assegnati liberamente a una `var` , purché l'oggetto dato abbia lo stesso tipo con cui la `var` stata dichiarata:

```
scala> var x = 1
x: Int = 1
```

```
scala> x = 2
x: Int = 2

scala> x = "foo bar"
<console>:12: error: type mismatch;
 found   : String("foo bar")
 required: Int
       x = "foo bar"
       ^
```

Nota nell'esempio sopra il tipo di `var` stato dedotto dal compilatore dato il primo assegnamento di valore.

val

Una `val` è un riferimento costante. Pertanto, un nuovo oggetto non può essere assegnato a una `val` che è già stata assegnata.

```
scala> val y = 1
y: Int = 1

scala> y = 2
<console>:12: error: reassignment to val
       y = 2
         ^
```

Tuttavia, l'oggetto a cui punta `val` *non* è costante. Quell'oggetto può essere modificato:

```
scala> val arr = new Array[Int](2)
arr: Array[Int] = Array(0, 0)

scala> arr(0) = 1

scala> arr
res1: Array[Int] = Array(1, 0)
```

DEF

Una `def` definisce un metodo. Un metodo non può essere riassegnato a.

```
scala> def z = 1
z: Int

scala> z = 2
<console>:12: error: value z_= is not a member of object $iw
       z = 2
       ^
```

Negli esempi precedenti, `val y` e `def z` restituiscono lo stesso valore. Tuttavia, una `def` viene valutata *quando viene chiamata* , mentre una `val` o `var` viene valutata *quando viene assegnata* .

Ciò può comportare un comportamento diverso quando la definizione ha effetti collaterali:

```
scala> val a = {println("Hi"); 1}
Hi
a: Int = 1

scala> def b = {println("Hi"); 1}
b: Int

scala> a + 1
res2: Int = 2

scala> b + 1
Hi
res3: Int = 2
```

funzioni

Poiché le funzioni sono valori, possono essere assegnate a `val` / `var` / `def` s. Tutto il resto funziona allo stesso modo di sopra:

```
scala> val x = (x: Int) => s"value=$x"
x: Int => String = <function1>

scala> var y = (x: Int) => s"value=$x"
y: Int => String = <function1>

scala> def z = (x: Int) => s"value=$x"
z: Int => String

scala> x(1)
res0: String = value=1

scala> y(2)
res1: String = value=2

scala> z(3)
res2: String = value=3
```

Lazy val

`lazy val` è una funzione linguistica in cui l'inizializzazione di una `val` è ritardata fino a quando non viene acceduta per la prima volta. Dopo quel punto, si comporta come una normale `val` .

Per usarlo aggiungi la parola chiave `lazy` prima di `val` . Ad esempio, utilizzando il REPL:

```
scala> lazy val foo = {
     |   println("Initializing")
     |   "my foo value"
     | }
foo: String = <lazy>

scala> val bar = {
     |   println("Initializing bar")
```

```
     |   "my bar value"
     | }
Initializing bar
bar: String = my bar value

scala> foo
Initializing
res3: String = my foo value

scala> bar
res4: String = my bar value

scala> foo
res5: String = my foo value
```

Questo esempio dimostra l'ordine di esecuzione. Quando viene dichiarato il valore `lazy val` , tutto ciò che viene salvato nel valore `foo` è una chiamata a funzione lenta che non è stata ancora valutata. Quando il regolare `val` è impostato, vediamo il `println` chiamata esecuzione e il valore viene assegnato a `bar` . Quando valutiamo `foo` la prima volta vediamo `println` execute - ma non quando viene valutato la seconda volta. Allo stesso modo, quando viene valutata la `bar` , non vediamo `println` execute - solo quando è dichiarata.

Quando usare 'pigro'

1. **L'inizializzazione è computazionalmente costosa e l'utilizzo di `val` è raro.**

   ```
   lazy val tiresomeValue = {(1 to 1000000).filter(x => x % 113 == 0).sum}
   if (scala.util.Random.nextInt > 1000) {
     println(tiresomeValue)
   }
   ```

   `tiresomeValue` richiede molto tempo per il calcolo e non è sempre utilizzato. Rendendolo un valore `lazy val` risparmia calcoli superflui.

2. **Risoluzione delle dipendenze cicliche**

   Diamo un'occhiata a un esempio con due oggetti che devono essere dichiarati contemporaneamente durante l'istanziazione:

   ```
   object comicBook {
     def main(args:Array[String]): Unit = {
       gotham.hero.talk()
       gotham.villain.talk()
     }
   }

   class Superhero(val name: String) {
     lazy val toLockUp = gotham.villain
     def talk(): Unit = {
       println(s"I won't let you win ${toLockUp.name}!")
     }
   }
   ```

```
class Supervillain(val name: String) {
  lazy val toKill = gotham.hero
  def talk(): Unit = {
    println(s"Let me loosen up Gotham a little bit ${toKill.name}!")
  }
}

object gotham {
  val hero: Superhero = new Superhero("Batman")
  val villain: Supervillain = new Supervillain("Joker")
}
```

Senza la parola chiave `lazy` , i rispettivi oggetti non possono essere membri di un oggetto. L'esecuzione di tale programma comporterebbe una `java.lang.NullPointerException` . Usando `lazy` , il riferimento può essere assegnato prima che venga inizializzato, senza timore di avere un valore non inizializzato.

Sovraccarico Def

Puoi sovraccaricare un `def` se la firma è diversa:

```
def printValue(x: Int) {
  println(s"My value is an integer equal to $x")
}

def printValue(x: String) {
  println(s"My value is a string equal to '$x'")
}

printValue(1)  // prints "My value is an integer equal to 1"
printValue("1") // prints "My value is a string equal to '1'"
```

Funziona allo stesso modo se all'interno di classi, tratti, oggetti o meno.

Parametri nominati

Quando si richiama una `def` , i parametri possono essere assegnati esplicitamente per nome. Ciò significa che non è necessario ordinarli correttamente. Ad esempio, definire `printUs()` come:

```
// print out the three arguments in order.
def printUs(one: String, two: String, three: String) =
   println(s"$one, $two, $three")
```

Ora può essere chiamato in questi modi (tra gli altri):

```
printUs("one", "two", "three")
printUs(one="one", two="two", three="three")
printUs("one", two="two", three="three")
printUs(three="three", one="one", two="two")
```

Ciò comporta che `one, two, three` vengano stampati in tutti i casi.

Se non tutti gli argomenti sono nominati, i primi argomenti sono abbinati per ordine. Nessun argomento posizionale (non denominato) può seguire un nome:

```
printUs("one", two="two", three="three") // prints 'one, two, three'
printUs(two="two", three="three", "one") // fails to compile: 'positional after named
argument'
```

Leggi Var, Val e Def online: https://riptutorial.com/it/scala/topic/3155/var--val-e-def

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Scala Language	4444, Andy Hayden, Ani Menon, Community, David G., David Portabella, dk14, Donald.McLean, Gabriele Petronella, Grzegorz Oledzki, implicitdef, isaias-b, J Atkin, Jean, Jonathan, mammothbane, marcospereira, Marek Skiba, mdarwin, Nathaniel Ford, NeoWelkin, Nicofisi, Priya, rolve, Shoe, sschaef, Thomas Andrews, Tyler James Harden, Ven, Vogon Jeltz
2	accattivarsi	Adamos Loizou, alphaloop, Amr Gawish, dimitrisli, Luka Jacobowitz, Nathaniel Ford, rjsvaljean, Suma, vise890
3	annotazioni	Gábor Bakos, Nathaniel Ford, Thomas Matecki
4	Classi di casi	Andy Hayden, Dan Simon, dk14, Gábor Bakos, HTNW, J Cracknell, keegan, made raka teja, Marc Grue, Nathaniel Ford, pedrorijo91, Rumoku, ScientiaEtVeritas, suj1th, Suma
5	Classi di tipo	Arseniy Zhizhelev, Daniel C. Sobral, Gábor Bakos, gregghz, Nathaniel Ford, TomTom, Yawar
6	Classi e oggetti	Aamir, Gábor Bakos, mdarwin, mirosval, MSmedberg, Nathaniel Ford, ScientiaEtVeritas, steve, Sudhir Singh, Tzach Zohar, vivek
7	collezioni	Anton, Camilo Sampedro, deepkimo, Donald.McLean, doub1ejack, EdgeCaseBerg, Filippo Vitale, George, implicitdef, ipoteka, Jason, John Starich, Mr D, Nathaniel Ford, raam86, Shastick, Suma, Tundebabzy, Vasiliy Levykin
8	Collezioni parallele	Nathaniel Ford, Shuklaswag
9	Combinatori parser	Nathaniel Ford
10	Configurare Scala	Hristo Iliev, Matas Vaitkevicius, Nathaniel Ford, Rjk
11	enumerazioni	Andy Hayden, Cortwave, Daniel Schröter, Gábor Bakos, implicitdef, ipoteka, Nathaniel Ford, phantomastray, Red Mercury
12	Espressioni regolari	dmitry, J Cracknell, Nathaniel Ford
13	estrattori	Andy Hayden, Dan Hulme, Dan Simon, Gábor Bakos, gilad hoch, Idloj, J Cracknell, jwvh, knutwalker, Łukasz, Martin Seeler, Michael Ahlers, Nathaniel Ford, Suma, W.P. McNeill

14	Funzione ordine superiore	acjay, ches, Nathaniel Ford, nukie, Rajat Jain, Srini
15	funzioni	Aravindh S, Archeg, Camilo Sampedro, ches, corvus_192, Dawny33, Gábor Bakos, Gabriele Petronella, implicitdef, ipoteka , Jean, jwvh, michael_s, Nathaniel Ford, raam86, rjsvaljean, ScientiaEtVeritas, Shastick, stefanobaghino, Sven Koschnicke, vise890, wheaties
16	Funzioni definite dall'utente per Hive	Camilo Sampedro
17	Funzioni parziali	acjay, Akash Sethi, David Leppik, dimitrisli, jwvh, Suma, Tzach Zohar
18	Futures	isaias-b, kevin628, Nathaniel Ford, nukie, Shastick
19	Gestione degli errori	Andy Hayden, Graham, John Starich, made raka teja, mnoronha, Nathaniel Ford, Simon, Suma, tacos_tacos_tacos, Tzach Zohar
20	Gestione XML	Nathaniel Ford, Rockie Yang, vsnyc
21	I flussi	jwvh, Nathaniel Ford, Oleg Pyzhcov
22	impliciti	Andy Hayden, dimitrisli, Gábor Bakos, HTNW, implicitdef, ipoteka, Jose Antonio Jimenez Saez, Michael Zajac, Nathaniel Ford, nattyddubbs, Simon, spiffman, Suma, Timo, vsminkov
23	Iniezione di dipendenza	Hoang Ong
24	Interoperabilità Java	Andrzej Jozwik, Dan Hulme, Gábor Bakos, mvn, the21st, thekingofkings
25	Interpolazione a stringa	Andy Hayden, Ayberk, Brian, implicitdef, J Cracknell, Nadim Bahadoor
26	Invocazione Dinamica	HTNW
27	JSON	ipoteka, John, Muki, Nathaniel Ford, pedrorijo91, suj1th, void, Wogan, zoitol
28	Lavorare con Gradle	Bianca Tesila, Nathaniel Ford, Rjk
29	Lavorare con i dati in uno stile immutabile	Filippo Vitale
30	Le tuple	corvus_192, evan.oman, Lawsy, Nathaniel Ford

31	Libreria di continuazioni	dmitry, HTNW
32	Macro	gregghz, HTNW, Nathaniel Ford
33	Mentre cicli	J Cracknell, Nathaniel Ford
34	Migliori pratiche	corvus_192, ipoteka, Nathaniel Ford, RamenChef, Sarvesh Kumar Singh, Shuklaswag
35	monadi	ipoteka, Nathaniel Ford
36	Operatori in Scala	Gábor Bakos, Shaido, Suminda Sirinath S. Dharmasena
37	Opzione Classe	Bruce Lowe, CPS, earldouglas, evan.oman, Governa, John Starich, Matthew Scharley, Nathaniel Ford, R Pieters, ScientiaEtVeritas, suj1th, Tzach Zohar, Vasiliy Levykin
38	Pacchi	Alex Javarotti, Nathaniel Ford, NetanelRabinowitz
39	Pattern Matching	Ali Dehghani, Andrzej Jozwik, Andy Hayden, CPS, Dan Simon, Daniel Werner, Filippo Vitale, Gábor Bakos, implicitdef, insan-e, jilen, jozic, JRomero, Justin Bailey, Louis F., mammothbane, Matt, Nadim Bahadoor, Nathaniel Ford, Peter Neyens, Sergio, Shastick, Shoe, Simon, Suma, T.Grottker, user6062072, vdebergue, vsminkov, Yagüe
40	Per le espressioni	Andy Hayden, J Cracknell, jwvh, LivingRobot, Nathaniel Ford, ScientiaEtVeritas
41	Programmazione a livello di codice	J Cracknell
42	Quasiquotes	gregghz
43	ricorsione	Dmitry Bystritsky, Gábor Bakos, jilen, jwvh, michael_s, ScientiaEtVeritas, teldosas
44	Riflessione	Sachin Janani
45	Scala.js	Camilo Sampedro
46	scaladoc	Camilo Sampedro, Gábor Bakos, Nathaniel Ford
47	scalaz	chengpohi
48	Scopo	Camilo Sampedro
49	Se espressioni	corvus_192, Nathaniel Ford, ScientiaEtVeritas
50	sincronizzato	Gábor Bakos

51	Singoli tipi di metodo astratto (tipi SAM)	Gábor Bakos, Gabriele Petronella, Nathaniel Ford
52	Sovraccarico dell'operatore	corvus_192, implicitdef, inzi, mnoronha, Nathaniel Ford, Simon
53	Symbol letterali	ZbyszekKr
54	Test con ScalaCheck	Andrzej Jozwik
55	Test con ScalaTest	Nadim Bahadoor, Nathaniel Ford
56	Tipi di auto	Gábor Bakos, irundaia
57	Tipo di inferenza	Gábor Bakos, Nathaniel Ford, suj1th
58	Tipo Parametrizzazione (Generics)	akauppi, Andy Hayden, Eero Helenius, Nathaniel Ford, vivek
59	Tipo Varianza	acjay, J Cracknell, Reactormonk
60	Tratti	André Laszlo, Andy Hayden, Donald.McLean, Louis F., Nathaniel Ford, Rumoku, Sudhir Singh, Vogon Jeltz
61	Unità di gestione (misure)	Gábor Bakos
62	Var, Val e Def	Aamir, John Starich, jwvh, linkhyrule5, Nathaniel Ford, Shastick, Shuklaswag, stefanobaghino, ZbyszekKr