EBook Gratuito

APPRENDIMENTO functional-programming

Free unaffiliated eBook created from **Stack Overflow contributors.**

#functional-programming

Sommario

Di

Capitolo 1: Iniziare con la programmazione funzionale

Osservazioni

La programmazione funzionale è un paradigma di programmazione che modella i calcoli (e quindi i programmi) come la valutazione delle funzioni matematiche. Ha le sue radici nel calcolo lambda, che è stato sviluppato da Alonzo Church nella sua ricerca sulla computabilità.

La programmazione funzionale ha alcuni concetti interessanti:

- **Funzioni di ordine superiore**
- **Purezza**
- **ricorsione**
- **Pigrizia**
- **Trasparenza referenziale**
- **accattivarsi**
- **funtori**
- **monadi**
- **Memoizzazione e ottimizzazione della chiamata di coda**
- **Test delle unità funzionali**

Alcuni esempi di linguaggi di programmazione funzionale sono Lisp , Haskell , Scala e Clojure , ma anche altri linguaggi, come Python , R e Javascript permettono di scrivere (parti di) i tuoi programmi in uno stile funzionale. Anche in Java , la programmazione funzionale ha trovato il suo posto con Lambda Expressions e Stream API introdotte in Java 8.

Examples

Funzioni pure

Le funzioni pure sono autonome e non hanno effetti collaterali. Dato lo stesso insieme di input, una funzione pura restituirà sempre lo stesso valore di output.

La seguente funzione è pura:

```
function pure(data) {
    return data.total + 3;
}
```

Tuttavia, questa funzione non è pura in quanto modifica una variabile esterna:

```
function impure(data) {
    data.total += 3;
    return data.total;
```

```
}
```

Esempio:

```
data = {
    total: 6
};

pure(data);   // outputs: 9
impure(data); // outputs: 9 (but now data.total has changed)
impure(data); // outputs: 12
```

Funzioni di ordine superiore

Le funzioni di ordine superiore assumono altre funzioni come argomenti e / o le restituiscono come risultati. Formano gli elementi costitutivi della programmazione funzionale. La maggior parte delle lingue funzionali ha una qualche forma di funzione di filtro, per esempio. Questa è una funzione di ordine superiore, prendendo come argomento un elenco e un predicato (funzione che restituisce true o false).

Funzioni che non fanno mai riferimento a nessuna di queste `first-order functions` come `first-order functions` .

```
function validate(number,predicate) {
    if (predicate) {    // Is Predicate defined
        return predicate(number);
    }
    return false;
}
```

Qui "predicato" è una funzione che testerà per alcune condizioni che coinvolgono i suoi argomenti e restituirà vero o falso.

Un esempio di chiamata per quanto sopra è:

```
validate(someNumber, function(arg) {
    return arg % 10 == 0;
    }
);
```

Un requisito comune è quello di aggiungere numeri all'interno di un intervallo. Usando le funzioni di ordine superiore possiamo estendere questa capacità di base, applicando una funzione di trasformazione su ciascun numero prima di includerlo nella somma.

Si desidera aggiungere tutti gli interi all'interno di un determinato intervallo (utilizzando Scala)

```
def sumOfInts(a: Int, b: Int): Int = {
  if(a > b) 0
  else a + sumOfInts(a+1, b)
}
```

Vuoi aggiungere quadrati di tutti gli interi all'interno di un determinato intervallo

```
def square(a: Int): Int = a * a

def sumOfSquares(a: Int, b: Int): Int = {
  if(a > b) 0
  else square(a) + sumOfSquares(a + 1, b)
}
```

Si noti che queste cose hanno 1 cosa in comune, che si desidera applicare una funzione su ogni argomento e quindi aggiungerle.

Consente di creare una funzione di ordine superiore per fare entrambe le cose:

```
def sumHOF(f: Int => Int, a: Int, b: Int): Int = {
  if(a > b) 0
  else f(a) + sumHOF(f, a + 1, b)
}
```

Puoi chiamarlo così:

```
def identity(a: Int): Int = a

def square(a: Int): Int = a * a
```

Si noti che `sumOfInts` e `sumOfSquare` possono essere definiti come:

```
def sumOfInts(a: Int, b: Int): Int = sumHOF(identity, a, b)

def sumOfSquares(a: Int, b: Int): Int = sumHOF(square, a, b)
```

Come si può vedere da questo semplice esempio, le funzioni di ordine superiore forniscono soluzioni più generalizzate e riducono la duplicazione del codice.

> Ho usato Scala per esempio - di Martin Odersky come riferimento.

accattivarsi

Il Currying è il processo di trasformazione di una funzione che utilizza più argomenti in una sequenza di funzioni ognuna con un solo parametro. Il currying è correlato, ma non uguale a, all'applicazione parziale.

Consideriamo la seguente funzione in JavaScript:

```
var add = (x, y) => x + y
```

Possiamo usare la definizione di currying per riscrivere la funzione add:

```
var add = x => y => x + y
```

Questa nuova versione accetta un singolo parametro, `x` , e restituisce una funzione che accetta un singolo parametro, `y` , che alla fine restituirà il risultato dell'aggiunta di `x` e `y` .

```
var add5 = add(5)
var fifteen = add5(10) // fifteen = 15
```

Un altro esempio è quando abbiamo le seguenti funzioni che inseriscono le parentesi attorno alle stringhe:

```
var generalBracket = (prefix, str, suffix) => prefix + str + suffix
```

Ora, ogni volta che usiamo `generalBracket` dobbiamo passare tra parentesi:

```
var bracketedJim = generalBracket("{", "Jim", "}") // "{Jim}"
var doubleBracketedJim = generalBracket("{{", "Jim", "}}") // "{{Jim}}"
```

Inoltre, se passiamo nelle stringhe che non sono parentesi, la nostra funzione restituisce comunque un risultato errato. Risolviamo questo:

```
var generalBracket = (prefix, suffix) => str => prefix + str + suffix
var bracket = generalBracket("{", "}")
var doubleBracket = generalBracket("{{", "}}")
```

Si noti che sia la `bracket` che il `doubleBracket` ora sono funzioni in attesa del loro parametro finale:

```
var bracketedJim = bracket("Jim") // "{Jim}"
var doubleBracketedJim = doubleBracket("Jim") // "{{Jim}}"
```

Immutabilità

Nei linguaggi tradizionali orientati agli oggetti, `x = x + 1` è un'espressione semplice e legale. Ma nella programmazione funzionale, è *illegale* .

Le variabili non esistono nella programmazione funzionale. I valori memorizzati sono ancora chiamati variabili solo a causa della cronologia. In realtà, sono *costanti* . Una volta che `x` assume un valore, è quel valore per la vita.

Quindi, se una *variabile* è una *costante* , allora come possiamo cambiare il suo valore?

La programmazione funzionale riguarda le modifiche ai valori in un record facendo una copia del record con i valori modificati.

Ad esempio, invece di fare:

```
var numbers = [1, 2, 3];
numbers[0] += 1; // numbers = [2, 2, 3];
```

Tu fai:

```
var numbers = [1, 2, 3];
var newNumbers = numbers.map(function(number) {
    if (numbers.indexOf(number) == 0)
        return number + 1
    return number
});
console.log(newNumbers) // prints [2, 2, 3]
```

E non ci sono loop nella programmazione funzionale. Usiamo le funzioni di ricorsione o di ordine superiore come `map` , `filter` e `reduce` per evitare il looping.

Creiamo un semplice ciclo in JavaScript:

```
var acc = 0;
for (var i = 1; i <= 10; ++i)
    acc += i;
console.log(acc); // prints 55
```

Possiamo ancora fare meglio cambiando la vita di `acc` da global a local:

```
function sumRange(start, end, acc) {
    if (start > end)
        return acc;
    return sumRange(start + 1, end, acc + start)
}
console.log(sumRange(1, 10, 0)); // 55
```

Nessuna variabile o loop significa codice più semplice, più sicuro e più leggibile (specialmente quando si esegue il debug o il test - non è necessario preoccuparsi del valore di `x` dopo un numero di istruzioni, non cambierà mai).

Leggi Iniziare con la programmazione funzionale online: https://riptutorial.com/it/functional-programming/topic/3184/iniziare-con-la-programmazione-funzionale

Capitolo 2: Loop con funzioni ricorsive ricorsive e tail

introduzione

Come già sapete, per motivi di immutabilità non è possibile elaborare i dati utilizzando per cicli e cicli continui. Quindi abbiamo funzioni ricorsive da salvare.

Examples

non ricorsivo (dove l'immutabilità non è una preoccupazione)

```
function sum(numbers) {
    var total = 0;
    for (var i = numbers.length - 1; i >= 0; i--) {
        total += numbers[i];
    }
    return total;
}
```

È un codice procedurale con mutazioni (sul `total`).

ricorsivo per il salvataggio

```
function sum(numbers) {
    if(numbers.length == 0) {
        return 0;
    }
    return numbers[0] + sum(numbers.slice(1));
}
```

questa è la versione ricorsiva. non c'è alcuna mutazione, ma stiamo creando uno stack di chiamate come sotto, che utilizza memoria extra.

somma ([10, 5, 6, 7]);

10 + somma ([5, 6, 7]);

10 + 5 + somma ([6, 7]);

10 + 5 + 6 + somma ([7]);

10 + 5 + 6 + 7 + somma ([]);

10 + 5 + 6 + 7 + 0;

coda ricorsiva per ottimizzare

```
function sum(numbers) {
    return tail_sum(numbers, 0);
}

function tail_sum(numbers, acc) {
    if(numbers.length == 0) {
        return acc;
    }
    return tail_sum(numbers.slice(1), acc + numbers[0]);
}
```

nella versione ricorsiva della coda, il valore di ritorno della funzione non ha bisogno di aspettare fino alla fine per fare i suoi calcoli, quindi non c'è uno stack enorme qui; solo due livelli.

somma ([10, 5, 6, 7]);

tail_sum ([10, 5, 6, 7], 0);

tail_sum ([5, 6, 7], 10);

tail_sum ([6, 7], 15);

tail_sum ([7], 21);

tail_sum ([], 28);

28;

Leggi Loop con funzioni ricorsive ricorsive e tail online: https://riptutorial.com/it/functional-programming/topic/10819/loop-con-funzioni-ricorsive-ricorsive-e-tail

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con la programmazione funzionale	AJW, Avinash Anand, Community, Huy Vo, Keelan, MauroPorrasP, Mr Tsjolder, rasmeister, rst_ack, tomturton, Zoyd
2	Loop con funzioni ricorsive ricorsive e tail	Dariush Alipour