La programmazione funzionale

La programmazione funzionale è un tipo di programmazione **dichiarativa**. Un approccio funzionale implica la composizione del problema come **set di funzioni da eseguire**. È necessario definire con attenzione l'**input** e l'**output** di ogni funzione.

---

in sintesi

- La programmazione funzionale o FP è un modo di pensare alla costruzione di software basata su alcuni principi fondamentali
- I concetti di programmazione funzionale si concentrano sui risultati, non sul processo
- L'obiettivo di qualsiasi linguaggio FP è quello di imitare le funzioni matematiche
- Alcuni dei più importanti linguaggi di programmazione funzionale: 1) Haskell 2) SM 3) Clojure 4) Scala 5) Erlang 6) Clean
- Una "funzione pura" è una funzione i cui input sono dichiarati come input e nessuno di essi deve essere nascosto. Gli output sono anche dichiarate come output.
- Dati immutabili significa che dovresti essere in grado di creare facilmente strutture di dati invece di modificare quelle già esistenti
- Ti consente di evitare problemi di confusione ed errori nel codice
- Il codice funzionale non è facile, quindi è difficile da capire per il principiante
- FP utilizza i dati immutabili mentre OOP utilizza i dati mutabili

---

funzioni matematiche

- le funzioni sono funzioni **matematiche**: per un dato input restituiscono sempre lo stesso risultato;
- le funzioni **non modificano i dati ricevuti in input**, ma restituiscono sempre nuovi valori;
- le funzioni possono essere **passate come parametro** e **restituite** da altre funzioni, e possono essere **combinate** tra di loro.

Una funzione in matematica può essere scritta in questo modo:

- `f(X) = Y`

L'istruzione può essere letta come "Una funzione F, che accetta X come argomento e restituisce Y come output." X e Y possono essere qualsiasi numero.

Ne discende che:

- Una funzione matematica deve sempre avere un argomento.
- Una funzione matematica deve sempre restituire un valore.
- Una funzione matematica dovrebbe agire solo sui suoi argomenti di ricezione (cioè X) e non su elementi esterni.
- Per una data X, ci sarà solo una Y.

---

funzioni pure e non pure

Le funzioni pure sono le funzioni che restituiscono lo stesso output per il medesimo input.

Alcune funzioni sono definite non pure: sono funzioni che producono **effetti collaterali**.

Ad esempio una funzione senza parametri che abbia come valore di ritorno la data corrente

Ne discende che:

- Una funzione senza parametri non è pura o non ha senso
- Una funzione senza un output non è pura

Esempio

```
function pura(a,b)
{
        return a+b; //ritorno la somma di a e b, senza modificare i
parametri in ingresso
}
```

```
var y;
function nonPura()
{
        y = y + 100; //modifico il valore di una variabile esterna al
blocco...
}
```

Funzioni pure

Una "funzione pura" è una funzione i cui input sono dichiarati come input e nessuno di essi deve essere nascosto. Gli output sono anche dichiarati come output.

Le funzioni pure agiscono sui propri parametri. Non è efficace se non restituisce nulla. Inoltre, offre lo stesso output per i parametri indicati

Funzioni impure

Le impure funziona esattamente al contrario. Hanno input o output nascosti; si chiama impure. Le funzioni impure non possono essere utilizzate o testate separatamente in quanto hanno dipendenze.

Trasparenza referenziale

Le **funzioni pure** garantiscono la trasparenza referenziale: la capacità di **sostituire una espressione con il suo risultato**, mantenendo la correttezza dell'applicazione.

I programmi funzionali dovrebbero eseguire operazioni proprio come se fosse per la prima volta. Quindi, saprai cosa potrebbe essere successo o meno durante l'esecuzione del programma e i suoi effetti collaterali. Nel termine FP si chiama **trasparenza referenziale**.

Nei programmi funzionali le variabili una volta definite *non cambiano il loro valore in tutto il programma*. I programmi funzionali non hanno dichiarazioni di assegnazione. Se dobbiamo memorizzare qualche valore, definiamo invece nuove variabili. Questo elimina qualsiasi possibilità di effetti collaterali perché qualsiasi variabile può essere sostituita con il suo valore effettivo in qualsiasi punto di esecuzione. Lo stato di qualsiasi variabile è costante in ogni istante.

Sfruttando la **trasparenza referenziale** c'è la possibilità di utilizzare una tecnica detta *memoization*, che consiste nel mettere in cache risultati di funzioni che hanno un certo costo computazionale.

```
var funzioneMemoizzata = memoize(funzioneConCalcoliPesanti);
```

Dati immutabili

Dati immutabili significa che dovresti essere in grado di creare facilmente strutture di dati invece di modificare quelle già esistenti.

Nella programmazione funzionale, non possiamo modificare una variabile dopo che è stata inizializzata. Siamo in grado di creare nuove variabili, ma non possiamo modificare le variabili esistenti e questo aiuta davvero a mantenere lo stato durante il runtime di un programma. Una volta creata una variabile e impostato il suo valore, possiamo avere piena fiducia sapendo che il valore di quella variabile non cambierà mai.

Funzione di prima classe

Le funzioni di prima classe vengono trattate come variabili di prima classe. Le variabili di prima classe possono essere passate a funzioni come parametro, possono essere restituite da funzioni o memorizzate in strutture di dati. Le funzioni di ordine superiore sono le funzioni che accettano altre funzioni come argomenti e possono anche restituire funzioni.

---

closure

La closure è una funzione interna che può accedere alle variabili della funzione genitore, anche dopo che la funzione genitore è stata eseguita. Funzioni di ordine superiore

Le funzioni di ordine superiore accettano altre funzioni come argomenti o le restituiscono come risultati.

Le funzioni di ordine superiore consentono applicazioni parziali o curry. Questa tecnica applica una funzione ai suoi argomenti uno alla volta, poiché ogni applicazione restituisce una nuova funzione che accetta l'argomento successivo.

Ricorsione

Non ci sono loop "for" o "while" nei linguaggi funzionali. L'iterazione nei linguaggi funzionali viene implementata attraverso la ricorsione. Le funzioni ricorsive si richiamano ripetutamente fino a quando non

raggiunge il caso base.

```
var serieFibonacci = function (n)
{
  if (n===1)
  {
    return [0, 1];
  }
  else
  {
    var s = serieFibonacci(n - 1);
    s.push(s[s.length - 1] + s[s.length - 2]);
    return s;
  }
};

 console.log(serieFibonacci(10));
```

Fumzione Ricorsiva

```
function fattorializza(num){
   if (num === 0 || num === 1){return 1;}
     return (num * fattorializza(num – 1));
   }

var result = fattorializza(14);

console.log(result);
```

Ricorsione con ciclo for

```
function fattorializza(num) {
  if (num === 0 || num === 1){
    return 1;
  }
  for (var i = num-1; i >= 1; i-- ) {
     num *= i;
   }
 return num;
}
console.log(fattorializza(6));
```

Composizione delle funzioni

La composizione delle funzioni combina 2 o più funzioni per crearne una nuova. Stati condivisi

Gli stati condivisi sono un concetto importante nella programmazione OOP. Fondamentalmente, sta aggiungendo proprietà agli oggetti. Ad esempio, se un disco rigido è un oggetto, capacità di archiviazione e dimensioni del disco possono essere aggiunte come proprietà.

Effetti collaterali

Gli effetti collaterali sono eventuali cambiamenti di stato che si verificano al di fuori di una funzione chiamata. L'obiettivo principale di qualsiasi linguaggio di programmazione FP è ridurre al minimo gli effetti collaterali, separandoli dal resto del codice software. Nella programmazione FP È fondamentale rimuovere gli effetti collaterali dal resto della logica di programmazione.

---

modularità

Il design modulare aumenta la produttività. I piccoli moduli possono essere codificati rapidamente e hanno maggiori possibilità di riutilizzo, il che sicuramente porta a uno sviluppo più rapido dei programmi. Oltre a ciò, i moduli possono essere testati separatamente, il che consente di ridurre il tempo impiegato per il test e il debug dell'unità.

---

manutenibilità

Manutenibilità è un termine semplice che significa che la programmazione FP è più facile da mantenere in quanto non è necessario preoccuparsi di modificare accidentalmente qualcosa al di fuori di una determinata funzione.

I vantaggi della programmazione funzionale

- Le funzioni pure sono più facili da capire perché non cambiano alcuno stato e dipendono solo dall'input che viene loro dato. Qualunque output producano è il valore di ritorno che danno. La loro firma della funzione fornisce tutte le informazioni su di essi, ad esempio il loro tipo di ritorno e i loro argomenti.
- La capacità dei linguaggi di programmazione funzionale di trattare le funzioni come valori e di trasmetterle a funzioni come parametri rendono il codice più leggibile e facilmente comprensibile.
- Il test e il debug è più semplice. Poiché le funzioni pure accettano solo argomenti e producono output, non producono alcun cambiamento, non accettano input o producono output nascosti. Usano valori immutabili, quindi diventa più facile controllare alcuni problemi nei programmi scritti che usano funzioni pure.
- Viene utilizzato per implementare la concorrenza / il parallelismo perché le funzioni pure non cambiano le variabili o altri dati al di fuori di essa.
- Adotta una valutazione lenta che evita una valutazione ripetuta perché il valore viene valutato e memorizzato solo quando è necessario.
- Ti consente di evitare problemi di confusione ed errori nel codice
- Più facile da testare ed eseguire test unitari ed eseguire il debug del codice FP.
- Elaborazione e concorrenza parallele
- Implementazione di hot code e tolleranza agli errori

- Offre una migliore modularità con un codice più breve
- Aumento della produttività dello sviluppatore
- Supporta funzioni nidificate
- Costrutti funzionali come Lazy Map & Lists, ecc.
- Permette un uso efficace del Lambda Calculus

Limiti della programmazione funzionale

- A volte la scrittura di funzioni pure può ridurre la leggibilità del codice.
- Scrivere programmi in stile ricorsivo invece di usare i loop può essere un po' complicato.
- Scrivere funzioni pure è facile, ma combinarle con il resto dell'applicazione e le operazioni di I / O è un compito difficile.
- Valori immutabili e ricorsione possono portare a una riduzione delle prestazioni.
- Il paradigma di programmazione funzionale non è facile, quindi è difficile da capire per il principiante
- Difficile sapere con esattezza come molti oggetti evolvono durante la codifica
- Ha bisogno di molta progettazione e di un grande ambiente di sviluppo e runtime
- Il riutilizzo è molto complicato e necessita di un costante refactoring
- Gli oggetti potrebbero non rappresentare correttamente il problema

---

Linguaggi funzionali

- Haskell (Facebook anti-spam system)
- SML
- Clojure
- Scala
- Erlang (Whatsapp)
- Clean
- F#
- ML/OCaml Lisp / Scheme
- XSLT
- SQL
- Mathematica

Imperative

```
const arr = [1, 2, 3, 4, 5, 6 ,7 ,8 ,9]

function getOdds(arr){
   let odds = [ ];    for(let i = 0; i < arr.length + 1; i++){
          if ( i % 2 !== 0 ){
             odds.push( i )
```

```
            };
          };
     return odds
   };
   console.log(getOdds(arr))
// stampa [1, 3, 5, 7, 9]
```

Functional

```
function getOdds2(arr){
   return arr.filter(num => num % 2 !== 0)
}
console.log(getOdds2(arr))
// stampa [ 1, 3, 5, 7, 9 ]

// sintassi abbreviata (lambda)
const getOdds3 = arr => arr.filter(num => num % 2 !== 0)
console.log(getOdds3(arr))
// stampa [ 1, 3, 5, 7, 9 ]
```