UNIVERSITÀ DEGLI STUDI DI
CASSINO E DEL LAZIO MERIDIONALE

DIPARTIMENTO DI INGEGNERIA ELETTRICA
E DELL'INFORMAZIONE "MAURIZIO SCARANO"

CORSO DI LAUREA IN INGEGNERIA INFORMATICA
E DELLE TELECOMUNICAZIONI

Assessment delle performance di Elixir nell'ambito IOT.

Relatore:
Prof. Ciro D'Elia

Candidato:
Nico Fiorini

ANNO ACCADEMICO 2022/2023

Questa è una dedica

La perfezione non è il nostro obbiettivo ma la nostra tendenza
Omar Palermo

Abstract

L' industria del software si trova a fronteggiare la necessità di sviluppare software sempre più scalabili e performanti per fronteggiare l'aumento degli utenti e di servizi che ne fanno utilizzo. In questo contesto, Elixir, un linguaggio di programmazione funzionale e concorrente basato su Erlang, emerge come una scelta promettente per la costruzione di sistemi altamente affidabili e reattivi, semplificando di molto lo sviluppo di software concorrenziale.

Questo studio si propone di analizzare le caratteristiche di Elixir e le sue performance attraverso una serie di esperimenti empirici esplorando diversi aspetti delle performance mettendo in rilievo vantaggi e svantaggi nell'adottarlo.

I risultati di questa ricerca forniranno una comprensione approfondità delle capacità di Elixir in termini di prestazioni e affidabilità consentendo agli sviluppatori di fare una scelta pensata alle esigenze dei loro progetti.

Indice

1 Introduzione **1**

2 Caratteristiche di Elixir **3**
2.1 Introduzione 3
2.2 Il paradigma funzionale 4
2.3 Struttura di un progetto Elixir 5
2.3.1 Il tool Mix 5
2.3.2 Moduli 5
2.4 Basi dichiarative 6
2.5 Concorrenza 6
2.6 Supervision Tree 6

3 Performance - Test sperimentali **7**
3.1 Introduzione 7

4 Conclusioni **8**

Bibliografia **9**

Capitolo 1

Introduzione

Elixir è un linguaggio di programmazione dinamico e funzionale sviluppato nel 2012 da José Valim, con l'obbiettivo di favorire una maggiore scalabilità e produttività nella macchinia virtuale di Erlang, mantenendo al contempo la compatibilità con l'ecosistema di Erlang[1]. Elixir si è affermato come una promettente scelta nell'industria del software,specialmente in contesti dove è richiesta scalabilità, tolleranza agli errori e reattività grazie al suo approccio concorrenziale.

In particolare, Elixir può risultare vantaggioso nel campo dell'IoT per diversi motivi:

1. **Concorrenza**: Nell'ambito dell'IoT, la gestione simultanea di dispositivi è essenziale. Elixir, grazie alla sua capacità di gestire facilmente la concorrenza, consente il monitoraggio e il controllo efficiente di numerosi dispositivi contemporaneamente.

2. **Fault Tolerance**: Data la natura degli ambienti IoT, dove i dispositivi possono guastarsi improvvisamente, Elixir offre strumenti per la supervisione e la gestione degli errori, garantendo la continuità delle operazioni anche in caso di fallimenti.

3. **Sviluppo Rapido e Manutenzione**: Elixir è un linguaggio moderno che offre una sintassi efficiente e snella, oltre a strumenti di sviluppo come Mix per la gestione delle dipendenze e l'ambiente interattivo iex. La presenza di un package manager (Hex)[**Hex63:online**] e la possibilità di generare automaticamente la documentazione facilitano il processo di sviluppo e manutenzione del codice.

Il trattato esplora Elixir concentrandosi su due aspetti principali: la semplicità e le performance. Si analizzano i punti di forza di un linguaggio funzionale

e come questi sono sfruttati in Elixir, con un focus sulla concorrenza. Nella scelta di un linguaggio, la semplicità è fondamentale e deve essere accessibile a tutti i programmatori. Tuttavia, l'efficienza è altrettanto importante, quindi vengono condotti test empirici per valutare le performance di Elixir.

In particolare il lavoro effettuato è così ripartito:

- Nel capitolo 2 si discute del linguaggio funzionale, esaminando le astrazioni offerte da Elixir per lo svilluppo di codice affidabile, si tratta la concorrenza e come la Erlang VM si occupa della gestione dei processi.
- Nel capitolo 3 si spiega il lavoro sperimentale svolto e i risultati ottenuti (continuare)

Capitolo 2

Caratteristiche di Elixir

2.1 Introduzione

In questo capitolo, esamineremo le caratteristiche distintive di Elixir, un linguaggio di programmazione funzionale e concorrente che sfrutta appieno la potenza della piattaforma OTP (Open Telecom Platform), non si vuole coprire ogni dettaglio del linguaggio, ma mettere in evidenza le caratteristiche fondamentali per iniziare a capire come pensare il codice con questo linguaggio.

Elixir, scritto in Erlang ed eseguito sulla macchina virtuale Erlang (BEAM), eredita gli obiettivi di Erlang, ma apporta miglioramenti significativi per rendere il linguaggio più appetibile e moderno.

Erlang, nato nel 1986, è stato progettato per semplificare lo sviluppo di software concorrente e robusto. Elixir si basa su queste fondamenta solide, offrendo un'API più pulita e astrazioni avanzate che consentono ai programmatori di ragionare a un livello più elevato, facilitando la scrittura di codice concorrente in modo intuitivo.

Una delle massime principali di Erlang e, di conseguenza, di Elixir, è "Let it crash" (Lascia che si schianti), che riflette l'approccio alla gestione degli errori nei sistemi concorrenti, incoraggiando la gestione degli errori tramite il rilancio e la supervisione anziché il blocco del processo.

Per capire come lavorare con questo linguaggio, bisogna affrontare un po' di questioni e farsi un po' di domande. Bisogna capire come la macchina virtuale Beam affronta la concorrenza, Elixir in particolare è un linguaggio orientato alla concorrenza e le astrazioni che fornisce sono proprio per far sì che si programmi in modo concorrenziale portando ad avere un codice responsivo e gestendo bene i processi attraverso il meccanismo di Supervision, il software diventa anche robusto. Un altro punto da affrontare è l'immutabilità dei dati,

è un concetto chiave in Elixir ed Erlang, è proprio questa caratteristica che ci semplifica la programmazione concorrenziale.

2.2 Il paradigma funzionale

Come già accennato Elixir è un linguaggio funzionale, dove il concetto di funzione ricopre il ruolo di protagonista, i dati sono immutabili e il codice è dichiarativo.

Questo modo di vedere le cose deriva dal Lambda calcolo o λ-calcolo [3] un sistema formale definito da Alonzo Church nel 1936, sviluppato per definire formalmente le funzioni e il loro calcolo.

In un paradigma basato su stati come la programmazione ad oggetti spesso si hanno variabili condivise mutabili, ovvero, più parti del codice possono riferirsi alla stessa variabile, e questo complica la programmazione multithreading dovendosi preoccupare di meccanismi come il blocco sincronizzato o il locking per evitare le race condition tra più parti del codice, e non è immediato scrivere del codice concorrenziale sicuro e spesso si riscontrano comportamenti indeterminati. In un paradigma funzionale si predilogono le variabili immutabili che aggirano questo problema riducendo il rischio di scrivere codice concorrenziale non sicuro.

Cambiare paradigma non è immediato, un paradigma si può dire che definisce il modo di pensare al problema, nella programmazione ad oggetti per esempio si definiscono le cosidette classi, pensando al problema come oggetti che possono comportarsi in un determinato modo attraverso le funzioni definite su di esso. Perciò si pensa ad un oggetto che ha un comportamento e che cambia il suo stato nel tempo, un modo di sviluppare intuitivo ma non sempre ottimale per la risoluzione di problemi. Nella programmazione funzionale si cambia prospettiva, ovvero si ha un input, si passa l'input alla funzione e si ottiene la trasformazione dell'input ottenendo l'output.

In poche parole un linguaggio funzionale assume che scrivere un software complesso sia più facile nel momento in cui il codice ha queste proprietà:

- I dati sono immutabili
- Le funzioni sono pure, ovvero, il risultato di una funzione dipende soltanto dai suoi parametri in input.
- Le funzioni non generano effetti oltre il suo valore restituito.

Con queste proprietà si ha più controllo del flusso del programma, anche se non sempre possono essere soddisfatte.

2.3 Struttura di un progetto Elixir

Elixir è un linguaggio moderno, e come ogni linguaggio moderno che si rispetti fornisce un tool per la creazione e configurazione di progetti, questo tool si chiama **Mix**.

2.3.1 Il tool Mix

È possibile creare un progetto con il comando:

```
mix new <nome-progetto>
```

Verrà creata una struttura per il progetto, e nel progetto saranno presenti una cartella **lib** dove andrà il codice, una cartella **test** per creare degli Unit Test, ma soprattutto viene creato un file **mix.exs** per la configurazione del progetto.

Come sappiamo Elixir fornisce anche un ambiente interattivo (**iex**) per testare il nostro codice, ed è consentito avviare questo ambiente nel dominio della nostra applicazione con il comando:

```
iex -S mix
```

Si può compilare il progetto con:

```
mix compile
```

Con Mix possiamo includere e scaricare facilmente anche librerie esterne attraverso il package manager.[2]

2.3.2 Moduli

Elixir organizza il codice in Moduli, che permettono di definire le funzioni dentro dei namespace,permettendoci così di separare le responsabilità raggruppando le funzioni.

Ci sono varie cose che si possono definire dentro un modulo, si possono definire delle **struct** ma cosa più importante si possono definire i cosiddetti **Behaviour**, un modo per definire un interfaccia Api, Elixir fornisce delle astrazioni proprio attraverso questi. Ciò che si vuole evidenziare ora è che il progetto è definito in moduli, ed è il modo che Elixir fornisce per organizzare il codice.

2.4 Basi dichiarative

Come già accennato, Elixir adotta un approccio dichiarativo nella definizione delle funzioni. Questo si contrappone all'approccio imperativo, che si concentra su "come posso risolvere questo problema?", mentre quello dichiarativo si pone la domanda "come posso definire un problema?".

Nell'esempio 2.1 è presentato un approccio imperativo al problema "somma dei primi n elementi", mentre nell'esempio 2.2 è presentato l' approccio dichiarativo con Elixir.

```
int sum_first_n(n){
  int sum=0;
    for(int i=1;i++;i<=n){
      sum+=i;
    }
    return sum;
}
```

Esempio 2.1: Somma N elementi

```
defmodule Sum do
  def sum_recursive(0), do: 0
  def sum_recursive(n), do: n + sum_recursive(n - 1)
end
```

Esempio 2.2: Somma N elementi

2.5 Concorrenza

2.6 Supervision Tree

Capitolo 3

Performance - Test sperimentali

3.1 Introduzione

Capitolo 4

Conclusioni

Bibliografia

[1] *Elixir (programming language) - Wikipedia*. https://en.wikipedia.org/wiki/Elixir_(programming_language).

[2] *HexDocs*. https://hexdocs.pm/.

[3] *Lambda calcolo - Wikipedia*. https://it.wikipedia.org/wiki/Lambda_calcolo.