PCD Module 3.4

Programmazione reattiva

Programmazione reattiva

Sia le CPS che le Promises richiedono che le computazioni asincrone facciano recapitare il risultato in un colpo. È comune però, ad esempio, nelle applicazioni la necessità di gestire **flussi** asincroni di dati o eventi. Questo è l'obiettivo della **programmazione reattiva**.

Tale paradigma è orientato ai flussi di dati e la propagazione dei cambiamenti, andando a facilitare la gestione di flussi asincroni di dati ed eventi. Relazionato con il pattern Observer e la programmazione guidata dagli eventi. Sta avendo molta attenzione nello sviluppo di applicazioni web responsive e big data.

Linguaggi e framework

- FrTime
- MS Reactive Extension
- Reactive Js - Flapjax, Bacon.js, AngularJs
- Reactive Dart
- Scala.React
- RxJava
- RxSwift...

Astrazioni

La RP astrae eventi variabili nel tempo per un loro consumo da parte del programmatore. Il programmatore può quindi definire elementi che reagiranno ad ogni evento in arrivo variabile nel tempo.

Le astrazioni sono valori di prima classe e possono essere passati o composti dal programma.

Due tipi di astrazioni reattive:

- **flussi di eventi**: modellano valori variabili nel tempo continui e discreti. Astrazione asincrona dal flusso progressivo di dati sequenziali e intermittenti in arrivo da un evento ricorrente (es. eventi del mouse)
- **comportamenti (segnali)**: valori continui nel tempo; astrazioni che rappresentano un flusso fluido e ininterrotto di dati in arrivo da un evento regikare (es. timer)

Esempio in Flapjax

```
var timer = timerB(100); // timer ogni 100ms - *comportamento*
var seconds = liftB( // altra var. comportamentale, memorizza tempo in sec.
    function (time){
        return Math.floor(time / 1000);
    }, timer );

insertDomB(seconds, 'timer-div'); // seconds è inserita nel DOM
// il valore del timer è continuamente aggiornato e mostrato nella pagina
```

Questo esempio crea un elemento DOM in una pagina web che mostra un timer in secondi.

Punto chiave: via completamente dichiarativa di gestire eventi e flussi.

Lifting

- I valori delle espressioni che dipendono da *valori reattivi* devono essere reattivi a loro volta.
- Una variabile che viene assegnata con qualche espressione che coinvolge _comportamenti o flussi di eventi diventa una variabile reattiva a sua volta, dal momento che l'aggiornamento da diversi comportamenti/flussi di eventi interessa la variabile stessa
- Il processo di convertire una normale variabile in una reattiva è chiamato **lifting**.
- Per ricomputare tutte le espressioni reattive una volta che un flusso di eventi o comportamento è eseguito, molte librerie costruiscono un **albero delle dipendenze**: quando un'espressione cambia, le espressioni dipendenti sono ricalcolate e i loro valori aggiornati.

Lifting implicito e esplicito

- Alcune librerie eseguono implicitamente il lifting (es. Bacon.js)
- Altre hanno bisogno che il programmatore esegua il lifting manualmente (es. React.js)
- Altre ancora fanno entrambe le cose (Flapjax)

Comporre flussi di eventi e comportamenti

Composizione reattiva astratta: permette di evitare l'incubo delle callback. Ad esempio, invece di avere tre callback separate per 3 eventi del mouse, si possono comporre come un singolo flusso id eventi che risponde a tutti e 3 gli eventi. Molte librerie forniscono questo tipo di supporto (Bacon.js con operatore combinazione, Flapjax con mergeE).

```
var saveTimer = timerE(10000); //10 seconds
var saveClicked = extractEventE('save-button','click');
var save = mergeE(saveTimer,saveClicked);
save.mapE(doSave); //save received data
```

Estensioni reattive

Per integrare aspetti di programmazione reattiva nei linguaggi principali si usano le **estensioni reattive (Rx)**. Rx è una libreria per comporre programmi asincroni e basati sugli eventi utilizzando collezioni osservabili.

Proprietà principali:

- **asincrona e basata sugli eventi**: la missione di Rx è semplificare i modelli di programmazione, un aspetto chiave per sviluppare programmi reattivi, dove l'asincronismo è essenziale.
- **composizione**: semplifica la composizione di computazioni asincrone
- **collezioni osservabili**: sfruttando la conoscenza attiva di modelli di programmazione come Linq, vedendo le computazioni asincrone come fonti di dati osservabili. Un mouse, ad esempio, diventa un db di eventi generati dal mouse, composti da diversi combinatori Linq.

Originalmente introdotte in .NET.

Principi

- Con Rx è possibile
  - rappresentare più *flussi di dati asincroni* in arrivo da diverse fonti
  - sottoscrivere al flusso di eventi utilizzando `IObserver<T>` (e `IOBservable<T>`
- Poichè le sequenze osservabili sono flussi di dati, possiamo eseguire query tramite operatori standard di Linq implementati in `Observable`. Inoltre è possibile filtrare, aggregare, comporre eventi multipli facilmente.
- Esistono un numero di altri operatori reattivi specifici per i flussi che permettono di scrivere query potenti. Cancellazioni, eccezioni e sincronizzazioni sono gestiti tramite i metodi di estensione di Rx.

`IObservable`

Rx va a completare e interoperare facilmente con flussi di dati sincroni `IEnumerable<T>` e computazioni asincrone a valore singolo `Task<T>`.

Interfaccia observer e observable

```
interface IObservable<T> {
  IDisposable Subscribe(IObserver<T> observer);
}
interface IObserver<T> {
  void OnNext(T value);
  void OnError(Exception error);
  void OnCompleted();
}
interface IDisposable {
  void Dispose();
}
```

Esempio

```
IObservable<int> source = Observable.Range(1, 10);
IDisposable subscription = source.Subscribe(
    x => Console.WriteLine("OnNext: {0}", x),
    ex => Console.WriteLine("OnError: {0}", ex.Message),
    () => Console.WriteLine("OnCompleted"));
Console.WriteLine("Press ENTER to unsubscribe...");
Console.ReadLine();
subscription.Dispose();
```

Quando un observer sottoscrive una sequenza osservabile, il thread che chiama il metodo `Subscribe` può essere diverso da quello in cui la sequenza viene eseguita fino al completamento. Di conseguenza, la chiamata Subscribe è asincrona in quanto il chiamante non viene bloccato fino al completamento l'osservazione della sequenza.

Ponti

Con eventi .NET esistenti

```
var lbl = new Label();
var frm = new Form { Controls = { lbl } };
IObservable<EventPattern<MouseEventArgs>>
      move = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
move.Subscribe(evt => {
  lbl.Text = evt.EventArgs.Location.ToString();
});
```

Il flusso di eventi `mouse-move` è convertito in una sequenza osservabile. Ogni volta che viene eseguito `mouse-move`, il sottoscrittore riceve una notifica `OnNext`. Possiamo quindi esaminare il valore `EventArgs` di tale notifica e avere la posizione del `mouse-move`.

Con fonti asincrone esistenti

```
Stream inputStream = Console.OpenStandardInput();
var read =
  Observable.FromAsyncPattern<byte[], int, int, int>
                           (inputStream.BeginRead, inputStream.EndRead);
byte[] someBytes = new byte[10];
IObservable<int> source = read(someBytes, 0, 10);
IDisposable subscription = source.Subscribe(
        x => Console.WriteLine("OnNext: {0}", x),
        ex => Console.WriteLine("OnError: {0}", ex.Message),
        () => Console.WriteLine("OnCompleted"));

Console.ReadKey();
```

`BeginRead` e `EndRead` per un oggetto `Stream` che usa il pattern `IAsyncResult` sono convertiti in una funzione che ritorna una sequenza osservabile.

Queryzzare sequenze osservabili tramite operatori Linq

```
var s1 = Observable.Range(1, 3);
var s2 = Observable.Range(1, 3);
```

Per:

- **trasformare dati**: tramite aggregazione e proiezione
  - *aggregazione*: `s1.Concat(s2).Subscribe(Console.WriteLine)`. Risultato: 1,2,3,1,2,3
  - *unione*: `s1.Merge(s2).Subscribe(Console.WriteLine)`. Risultato: 1,1,2,2,3,3
  - *proiezione*: `var seqNum = Observable.Range(1, 5); var seqString = from n in seqNum select new string('*', (int)n); seqString.Subscribe( str =>` Mappa un flusso di numeri in un flusso di stringhe.
  - **comporre dati**: tramite zip e selectMany
  - **chiedere dati**: tramite where e any

RxJava

Implementazione Java VM di Rx. Estende il pattern observer per supportare sequenze di dati/eventi e aggiunge operatori che permettono di comporre sequenze assieme dichiarativamente, astraendo problematiche riguardanti threading a basso livello, sincronizzazione, thread-safety, strutture dati concorrenti, IO bloccante. Supporta Java 5+ e linguaggi basati su JVM (es. Kotlin e Scala).

Operatori RX

- **creazione di observable**
- **trasformazione di observable**: trasformano elementi emessi da un observable
  - `Map`
  - `Scan`
  - `FlatMap`
  - `Buffer`
  - `GroupBy`
  - `Window`
- **filtro di observable**: emettono selettivamente oggetti da una fonte osservabile
  - `Filter`
  - `Skip`
  - `Take`
  - `Debounce`
  - `Distinct`

    - `ElementAt`
    - `First`
    - `IgnoreElements`
    - `Last`
    - `Sample`
    - `SkipLast`
    - `TakeLast`

- **combinazione di observable**: lavorano con più fonti osservabili per creare un singolo observable

    - `Merge`
    - `Zip`
    - `And/Then/When`
    - `Join`
    - `CombineLatest`
    - `StartWith`
    - `Switch`

- **operatori condizionali e booleani**: valutano uno o più osservabili o elementi emessi dagli osservabili
- **operatori matematici e di aggregazione**: operano sull'intera sequenza di oggetti emessi da un observable
- **operatori di contropressione**: strategie per "coping" con osservabili che produce elementi più rapidamente di quanto li consumano gli osservatori

Diagrammi di Marble

Vedi slide

Sfide

Nè la programmazione reattiva, nè le estensioni possono essere considerate la pallotola d'argento della programmazione asincrona. Sono effettive per gestire i flussi di dati/eventi in uno stile funzionale, ma non come modello di programmazione asincrona general-purpose.

La sfida oggi è come integrare tutti questi approcci e tecniche:

- sync + async + push + pull
- exploiting concurrency
- devising models that work also in the case of ds