PCD Module 3.3

Programmazione asincrona

Programmazione asincrona

Stile di programmazione molto importante ai giorni d'oggi. Riguarda l'esecuzione e la gestione di computazioni/richieste/processi asincroni, astraendoli dai thread. Recentemente supportato da tutti i principali framework e piattaforme. Include una varietà di meccanismi ed architetture, inclusa la programmazione guidata dagli eventi. È una problematica ancora in corso d'opera.

Task e futures

Future: fornendo una computazione asincrona (task), un oggetto rappresentante il risultato della future o lo stato della computazione è creato e ritornato immediatamente. L'oggetto permette di:

- controllare lo stato del task (poll)
- bloccarlo quando il risultato del task viene richiesto
- cancellare il task in esecuzione quando possibile
- catturando errori/eccezioni relativi al task in esecuzione

Esempi: Java Task Executor, MS Task-based Async Pattern, Apple's GCD, Android's AsyncTask.

Esempio: JTE

Task richiamabili negli executor che rappresentano una computazione in differita, completati con qualche tipo di risultato. Il metodo `call` incapsula il comportamento del task che fa qualche tipo di computazione e ritorna qualche tipo di valore V. (`V call() throws Exception`)

Inviato ad un executor service per mezzo del metodo `submit` , ritornante una Future. Le Future forniscono medoti per vedere quando il task è stato completato/cancellato/risolto/eliminato.
(`Future<T> submit(Callable<T> task)`)

Esempio: MS TAP

Basato sui tipi `Task` e `Task<TResult>` del namespace `System.Threading.Tasks` , utilizzati per rappresentare operazioni arbitrarie asincrone.

TAP utilizza un singolo metodo per rappresentare l'inizializzazione e il completamento di una operazione asincrona. Il metodo ritorna un `Task` o un `Task<TResult>` in base alla risposta (`void` o

`TResult`).

Implementazione tramite compilatori C# o VB, manualmente o tramite aiuto del compilatore.

Implementazione manuale

```
public static Task<int> ReadTask(this Stream stream, byte[] buffer,
                                 int offset, int count, object state)
{
    var tcs = new TaskCompletionSource<int>();
    stream.BeginRead(buffer, offset, count, ar =>
    {
        try { tcs.SetResult(stream.EndRead(ar)); }
        catch (Exception exc) { tcs.SetException(exc); }
    }, state);
    return tcs.Task;
}
```

Con `async` e `await`

Metodi che possono essere sospesi tramite `async` e `await` rilasciando il controllo al chiamante. Un metodo `await` può essere a sua volta aspettato dai metodi che lo chiamano.

```
private async Task SumPageSizesAsync() {
    byte[] urlContents = await client.GetByteArrayAsync(url);
    // ...
}
```

Prima di TAP

- **APM**, Async Programming Model, con metodi `begin` e `end`, deprecato
- **EAP**, Event-based Async Pattern, metodi con suffisso `async`, eventi, tipi delegati e tipi derivati da `EventArg`, deprecato
- **TAP**, Task-based Async Pattern, un solo metodo per rappresentare inizializzazione e completamento di op. asincrona.

Continuation Passing Style e event-driven

Ripensare a computazioni e modelli di programmazione per renderli completamente basati sulla programmazione asincrona.

Eseguendo una computazione asincrona, specifichiamo le funzioni che devono essere chiamate come **continuazione** della computazione quando la computazione asincrona è completata o ha avuto errori. La continuazione ha un parametro, ovvero i lvalore di ritorno passato dalla computazione asincrona.

```
//   pseudo-definition
function myAsyncFunc(param1, ..., paramN, continuation){
  // when the result of myAsyncFunc is ready, then
  continuation(result) is called
}
// non blocking invocation
myAsync(...,cont_function(value) {...})
```

CPS in generale

È uno stile di programmazione in cui il controllo è passato esplicitamente nella forma di **continuazione**.

Una funzione scritta in CPS prende un argomento in più, ovvero una funzione di un argomento. Quando la funzione CPS ha computato il valore del risultato, lo ritorna chiamando la funzione di continuazione con il valore come argomento. Invocando una CPS, la funzione chiamante fornisce una procedura che deve essere invocata con il valore di ritorno della subroutine. (tipo callback objc)

Callback come continuazioni

Callback: continuazioni chiamate quando il risultato di una computazione asincrona è pronta. Esempio in JS:

```
function loadUserPic(userId, ret) {
  findUserById(userId, function(user) {
    loadPic(user.picId, ret);
  });
}

loadUserPic('john', function(pic) {
  ui.show(pic);
});
```

Modello event-loop

Modello esecutivo

Chi invoca le continuazioni/callback? Due possibilità:

- un thread di controllo separato, in esecuzione concorrentemente al thread di controllo che ha eseguito la richiesta, concorrentemente. Problemi: inversione del controllo, race condition
- lo stesso thread logico di controllo che ha eseguito la richiesta: **modello event loop**, adottato da web app moderne e nella programmazione event-driven

Architettura event-loop e programmazione event-driven

Comportamento organizzato come insieme di **event handlers** che incapsulano la computazione per essere eseguita quando un evento è ricevuto.

Viene usata una coda degli eventi per tenere traccia di eventi generati dall'ambiente o dall'event handler.

Comportamento concettualmente rappresentato da un loop:

```
loop {
    Event ev = evQueue.remove()
    Handler handler = selectHandler(ev)
    execute(handler)
}
```

Esecuzione della computazione asincrona atomica: gli eventi accadono finchè un handler è in esecuzione e in coda nella coda eventi.

Osservazioni

- Gli event handler sono pensati per essere eseguiti senza blocchi (primitive/chiamate/richieste). Un comportamento bloccante deve essere sostituito da una richiesta asincrona o computazione.
- L'event loop è embeddato dentro al runtime che esegue il codice: gli sviluppatori non lo vedono, non hanno bisogno di creare i loop - basta specificare come scegliere ed eseguire gli event handlers. Forti similitudini con l'event loop degli attori.
- La programmazione guidata dagli eventi è chiamata anche **programmazione senza stack delle chiamate**: la gestione degli eventi non è una chiamata alla procedura.

Callback come continuazioni: esempi

- Gli eventi sono impliciti e si riferiscono al completamento con successo o errore di richieste/computazioni asincrone
- Gli event handler sono rappresentati dalle funzioni di continuazione specificate quando viene invocata una richiesta/computazione asincrona
- Le closure sono usate per definire il contesto da usare quando si processano gli eventi.

Esempi Javascript su slide

L'inferno delle callback

Programmazione guidata dagli eventi: benefici

- Uso di un singolo thread per coordinare e gestire molteplici task asincroni (eventualmente eseguito da thread esterni)
- Non ci sono race a basso livello (no stati condivisi)
- Non ci sono deadlock a basso livello (handler async non bloccano mai)

... e problemi

- **spaghetti asincroni**: computazione generale spezzata in handler asincroni. Va a minare la modularità e la struttura del programma e rende difficile comprendere il codice.
- **piramide della dannazione**: composizione basata su CPS può portare a callback/continuazioni innestate. Ciò aumenta la complessità e porta a una leggibilità del codice povera, poca riusabilità, poca estendibilità.

Piramide della maledizione

Callback innestate. Esempio di chat con tre chiamate asincrone:

```
registerToChatOnServer(username,function(rooms){
    joinAvailableRoom(rooms, function(roomname){
        sendChatToAll(roomname, msg, function(reply){
            showChatReply(reply);
        })
    })
});
```

Promises

Una soluzione parziale: le promises

Rappresentano l'eventuale completamento e risultato di una singola operazione asincrona. Incapsulano azioni asincrone, agendo più come un valore ritornato dal risultato di una computazione con la differenza che il valore potrebbe non essere sempre disponibile. Una promise può essere rigettata o risolta una sola volta. Permette di appiattire l'innestamento delle callback.

In Javascript

Definite nelle standard, chiamate anche **thenables**. Supportate da diversi framework (Q, When.js, jQuery Promises, ecc)

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...
  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
}
else {
    reject(Error("It broke"));
  }
});

promise.then(function(result) {
  console.log(result); // "Stuff worked!"
}, function(err) {
  console.log(err); // Error: "It broke"
});
```

Incatenamento di promises:

```
var promise = new Promise(function(resolve, reject) {
  resolve(1);
});
promise.then(function(val) {
  console.log(val); // 1
  return val + 2;
}).then(function(val) {
  console.log(val); // 3
});
```

Esempi con Q:

```
Q.fcall(registerToChatOnServer)
 .then(joinAvailableRoom)
 .then(sendChat)
 .then(function (reply){
        showChatReply(reply)
    },function (error){
}) .done();

// configuring the promise..
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
console.error("Failed!", error); });
```

Comporre le promise

In molte librerie le promise sono astrazioni ad alto livello. Possono quindi essere passate tra funzioni.

Un programmatore può esegure un numero di operazioni ad alto livello sulle promise come, ad esempio, comporre un gruppo di chiamate asincrone indipendenti. Esempio di Fibonacci in Q:

```
fibpromise = Q.all([ computeFibonacci(n-1), computeFibonacci(n-2) ]);

fibpromise.spread(function (result1, result2) {
    //resolver for both operations..
    console.log(result1 + result2);
}), function(err){
    //error occurred
});
```

Per reagire a `fibpromise` attacchiamo un resolver, tramite metodo `spread`, che permette agli argomenti di essere inviati al resolver.

Altri framework

- **AngularJS**
- **Dart**
- AmbientTalk
- Argus
- Fignale

Le promise non sono la pallottola d'argento

Sono una soluzione elegante a risolvere il problema dell'innestamento, ma non tutto può essere *promesso*:

- incremental processing
- event processing & streams • UIlogic
- temporal logic

Esempi concreti:

- Live search
- Processing streams of pushed data (SSE, WebSockets...)
- Incremental processing if large data sets as streams

Meccanismi async/await

Dart

Integra async/await in un linguaggio basato sull'event loop. Integra Java + JS, può anche essere tradotto in puro Javascript. Competitor: Typescript di Microsoft.

Dart è basato su event loop e meccanismi relativi, ma una estensione recente propone meccanismi async/await e stream reattivi per semplificare la programmazione asincrona.

Event Loop

Un'app Dart ha un singolo event loop con due code:

- **event queue**, contiene tutti gli eventi esterni (IO, eventi mouse, grafici, timer, ecc)
- **microtask queue**, contiene gli eventi relativi ai task da eseguire, ordinati dal programma stesso. È necessario poichè il codice per gestire gli eventi qualche volta ha bisogno di completare un task successivamente, ma prima di ripassare il controllo all'event loop.

Diagramma ed esempi su slide.

Introduzione di Async/Await

- **async**: funzione il cui corpo è marcato con `async`. Quando chiamiamo un'async, ritorna una Future e il corpo della funzione è programmato per essere eseguito più tardi. Quando viene eseguito, la Future viene completata. Permette di usare le await nella funzione.
- **await**: `await funzione`, sospende la funzione attualmente in esecuzione finchè il risultato non è pronto (la Future non è completa). Il risultato della funzione await è il completamento della Future.

```
import "dart:html";
main() async {
    var context = querySelector("canvas").context2D;
    var running = true;    // Set false to stop game.
    while (running) {
        var time = await window.animationFrame;
        context.clearRect(0, 0, 500, 500);
        context.fillRect(time % 450, 20, 50, 50);
    }
}
```