

Il debutto di Java 8: qualcosa di nuovo

Parecchie novità ci aspettano nella prossima versione di Java. L'edizione 8 sarà un punto di svolta per l'evoluzione del linguaggio?



Arriva la release 8 dell'ambiente di sviluppo che si preannuncia rivoluzionaria come fu per Java 5, che nel 2004 interruppe la pacifica evoluzione dei numeri di release dopo 1.2, 1.3, 1.4, oppure Java 7 che si prese ben quattro anni di pausa rispetto a Java 6. La novità più rilevante è una serie di costrutti dedicati a rendere più snello il linguaggio e aiutare gli sviluppatori ad approfittare delle macchine sempre più parallele a disposizione, dato che la corsa all'incremento della frequenza di clock ha lasciato da parecchio tempo il posto all'aumento del numero di core. Vediamo allora in dettaglio che cosa ci aspetta.

Il contenuto del JDK 8

La novità principale è il progetto Lambda, nome in codice JSR 335, che importa nel linguaggio dei costrutti per la creazione di blocchi funzionali che in forme più o meno simili abbiamo già visto nel C++11 e soprattutto in C#. Arriviamo per gradi al problema che si vuole risolvere: cominciamo a considerare una categoria particolare di classi, quelle con un singolo metodo non astratto, che si usano per gestire callback, come in questo caso:

```
foo.doSomething(new  
CallbackHandler() {  
    public void callback(Context  
c) {  
        System.out.println(c.v());  
    }  
});
```

di queste cinque righe di codice l'unica che preme allo sviluppatore, quella

che effettivamente svolge lavoro applicativo, è la `println`. Tutte le altre sono rumore causato dalla sintassi e servono a fare contento il compilatore.

Questi costrutti si potranno esprimere in modo molto più compatto, per esempio così:

```
CallbackHandler cb =  
    (Context c) -> System.out.  
println(c.v());
```

Il tipo dei parametri può essere dedotto automaticamente dal compilatore. Possiamo arrivare a creare un funtore in modo molto sintetico, per esempio così

```
x -> x+1
```

Questo artificio sintattico rende molto più lineare la programmazione con tutte quelle interfacce che hanno un singolo metodo astratto (Sam), come ad esempio *Runnable*, *Callable*, *EventHandler*, *Comparator*. Questo ingresso dei lambda nella sintassi dei linguaggi a oggetti più popolari, è stato sintetizzato provocatoriamente così: *basta programmare con*

«La programmazione con blocchi funzionali permette di ottenere maggiori prestazioni in un mondo di Cpu multicore»

i concetti del 1970, passiamo a quelli del 1930. Ma non si tratta di un'involuzione, anzi. L'uso di interfacce funzionali si presta, infatti, molto bene a un mucchio di situazioni applicative, specialmente ove il flusso di controllo naturale è la gestione di eventi.

Ma c'è dell'altro: la programmazione con blocchi funzionali permette anche di continuare la rincorsa verso prestazioni maggiori in un mondo in cui le velocità di clock cominciano a saturare e il mondo è pieno di Cpu multicore, dai computer fino ai tablet e ai telefoni. Consideriamo per esempio gli strumenti che Google e coloro che fanno data mining usano per analizzare un mare di dati e estrarre informazioni. Invece di usare un gigantesco database per valutare funzioni complesse in un insieme sterminato di oggetti, si usano centinaia o anche migliaia di macchine con agenti che valutano costrutti molto semplici su un singolo elemento ed emettono un valore, per esempio la presenza di parole chiave o la lista dei termini contenuti. Questi valori vengono poi aggregati in un imbuto che colleziona statistiche.

La prima fase di questa operazione, il calcolo di un valore a partire da un record, si chiama *map*, la seconda fase, l'aggregazione di valori, si chiama *reduce*. Questi termini sono familiari a chi ha sviluppato soluzioni con Apache Hadoop, un motore open source che realizza parte delle funzioni su cui fonda l'indicizzazione di Google.

Il principio che sta alla base dell'uso di *map* e *reduce* invece di un insieme

più sofisticato di processi, per esempio quello tipico dei costrutti Sql, è che map e reduce sono processi facilmente parallelizzabili, perché sono operazioni elementari che non hanno bisogno di contesto.

Chi ci segue, sa che i costrutti dei linguaggi funzionali sono simili a quello che abbiamo illustrato nel fatto che un requisito della programmazione funzionale è che le funzioni siano blocchi che agiscono esclusivamente sull'insieme dei loro parametri restituendo un valore, senza nessun altro effetto secondario sull'ambiente operativo.

Questi fattori stanno alla base dell'architettura delle nuove interfacce applicative delle collection di Java.

Multithreading nelle collection

Mostriamo in pratica quello che intendiamo dire con questo esempio, in cui si calcola il massimo voto di laurea nel 2010 facendo uso di map, reduce e di blocchi lambda. L'esempio dovrebbe chiarire dove sta il punto:

```
Collection<Student> students =
...;
double max =
students.filter(Students s ->
s.gradYear == 2010))
.map(Students s ->
s.score })
.reduce(0.0,
Math::max);
```

In questo esempio si vedono all'opera molti elementi interessanti di quella che è chiamata *Stream Api* delle *Collection*, cominciamo a smontarlo per capire. In primo luogo, vediamo i blocchi funzionali, per esempio

```
Students s -> s.gradYear ==
2010))
```

Il senso di questa funzione è: a uno studente si associa il fatto che si sia laureato nel 2010, il valore di ritorno della funzione è semplicemente il valore di verità di questo predicato.

Vediamo inoltre che alla collection *students* vengono applicati dei metodi a cascata, da cui si deduce che l'output di *.filter* è una collection, quasi certamente un sottoinsieme della collection iniziale.

All'output di *filter* si applica la *map*, con un altro blocco funzionale

```
Students s -> s.score
```

Che associa a ogni studente il voto di laurea, contenuto nella variabile *score* di un oggetto di classe *Student*. Il risultato di questa operazione è una collection di numeri, che contengono i voti. Infine, la fase finale, la riduzione di *reduce*, applica a ogni elemento della collezione di numeri il calcolo di un massimo preso dalla libreria *Math* con un valore iniziale di zero, come termine di paragone.

Si tratta di un modo di costruire catene di funzioni affine a quello reso popolare da JQuery fra i programmatori di interfacce utente, o a quello familiare ai programmatori Unix, avvezzi a costruire *pipe* di comandi shell per costruire operazioni complesse.

Il vantaggio, però non è solo sintattico e non si limita solo a stuzzicare l'occhio del programmatore.

Il costrutto *map* si applica a un singolo oggetto, senza che sia importante l'ordine in cui gli oggetti sono processati, mentre *reduce*, applica la stessa funzione di riduzione a tutti gli oggetti di una collezione, di nuovo senza preoccuparsi dell'ordine.

Ne segue che le operazioni di *map* e *reduce* possono essere spezzettate dal compilatore in modo da usare tutte le

Cpu a disposizione per processare blocchi di oggetti dividendo una collezione in parecchi elementi in collezioni più piccole e parallelizzando il calcolo, questo permette di avere il beneficio di una programmazione parallela senza doversi preoccupare eccessivamente delle primitive di sincronizzazione tipiche del calcolo parallelo.

Insomma, una stream di elementi della collezione si può dividere in rivoli che possono essere processati indipendentemente e parallelamente, garantendo prestazioni migliori.

Un motore di scripting

Nashorn è un motore JavaScript scritto per girare direttamente sulla macchina virtuale DaVinci che sta al cuore delle release correnti di Java.

Nashorn significa rinoceronte in tedesco. Per il gruppo di sviluppatori che ha lavorato al progetto, il nome è nato come un gioco di parole su Rhino, il motore JavaScript della Mozilla Foundation.

Il motore di scripting con il timbro di Oracle dovrebbe essere leggero e veloce quanto è possibile esserlo su una macchina virtuale Java, di sicuro è un progetto più moderno e radicale di Rhino, quindi ci aspettiamo risultati lusinghieri dai benchmark.

Ci aspettiamo anche da Nashorn che renda JavaScript un linguaggio di primo livello sulla macchina virtuale, dando impulso alle soluzioni JavaScript server side e alla realizzazione di widget che usano JavaFX, la libreria client per la realizzazione di oggetti visuali interattivi, di cui Java 8 include la versione 3.0, predisposta per il multi touch.

Un esempio di uso di Nashorn per creare un display Lcd con JavaFX, si trova all'indirizzo blogs.oracle.com/nashorn/entry/repost_taming_the_nashorn_first. L'articolo spiega in dettaglio come creare e lanciare in esecuzione l'applicazione di esempio attivando la macchina JavaScript con il comando *jjs*.

Supporto tablet e cellulari

Un'altra novità importante di Java 8 è il supporto nativo per i dispositivi portatili, come i telefoni e i tablet, con tutto il corredo di funzioni per attivare il posizionamento geografico e la macchina fotografica.

Certo, Java sui telefoni significa per quasi tutti Android, un sistema su cui

JDK 8: LE PRINCIPALI NOVITÀ

Project Jigsaw	Sistema di moduli per le applicazioni Java e per la piattaforma Java
Project Lambda	Closures e funzionalità connesse nel linguaggio Java {JSR 335}
	Operazioni parallele nelle Api relative alle collections {filter/map/reduce}
Oracle JVM Convergence	Migrazione delle funzioni di analisi delle prestazioni e profiling da JRockit, inclusi Mission Control e Flight Recorder
JavaFX 3.0	Java client di nuova generazione, Multi-touch
JavaScript	Nuova generazione dell'engine JavaScript-on-JVM {Project Nashorn}
	Interoperabilità JavaScript/Java sotto JVM
Supporto a device	Camera, Location, Bussola e Accelerometro
Produttività sviluppo	Annotations onTypes {JSR 308}
API e altri aggiornamenti	Miglioramenti a Security, Date/Time {JSR 310}, Networking, Internationalization, Accessibility, Packaging/Installation

```
Collection<Student> students = ...;

max = students. filter (new Predicate<Student>() {
    public boolean op(Student s) {
        return s.gradYear > 2011;
    }
}) .map (new Extractor<Student, Double>() {
    public Double extract(Student s) {
        return s.score;
    }
}) .reduce (0.0, new Reducer<Double, Double>() {
    public Double reduce(Double max, Double score) {
        return Math.max(max, score);
    }
});
```

La novità più grossa riguarda il modo in cui si potranno trattare le Collection di Java, con un approccio simile al web mining, ideale per le Cpu multicore

la macchina virtuale standard di Oracle è stata soppiantata da una macchina ad hoc. Java su Android, quindi è presente solo come sintassi e architettura di base. Comunque la mossa ha senso per supportare applicazioni in Java per i tablet e per organizzare eventualmente un rientro in gioco.

Annotation on types

Ci sono innovazioni interessanti anche per quanto riguarda la sintassi del linguaggio, un'estensione delle annotazioni che riguardano le variabili e la possibilità di creare metodi di default nelle interfacce. Le annotazioni, estese secondo i dettami della specifica JSR308 permettono di dare indicazioni al compilatore e verificare assunzioni sull'uso delle variabili, come ad esempio l'intenzione di creare una lista senza modificarne il contenuto

```
class UnmodifiableList<T>
implements @ReadOnly List<@
ReadOnly T> { ... }
```

La sintassi del linguaggio è stata modificata in modo compatibile con il passato per consentire un uso più esteso delle annotazioni. In Java 7 le annotazioni possono essere scritte solo sui parametri formali di una dichiarazione e nella dichiarazione di classi, package, metodi, campi e variabili locali.

In Java 8 sono consentite annotazioni in ogni uso di un tipo e sulla dichiarazione dei parametri.

La specifica non indica esplicitamente quali annotazioni sono gestite. Le annotazioni, come `ReadOnly` nell'esempio precedente, possono essere gestite dal compilatore o da futuri plug-in del compilatore. Il meccanismo è specificato solo nella sua struttura lasciando libertà di implementazione a evoluzioni future del compilatore e di strumenti accessori per gli sviluppatori.

L'uso delle annotazioni può rendere il codice più snello, per esempio sottintendendo dei test senza doverli esplicitare, come in questo frammento:

```
myString = (@NonNull String)
myObject;
```

Defender methods

L'altra importante estensione al linguaggio è legata al miglioramento che riguarda i lambda e il loro uso con le funzioni streaming delle collection, di tratta dei *defender methods*, o *virtual extension methods*: un meccanismo che consente di creare metodi di default nelle interfacce, senza impedirne l'estensione.

Cominciamo dall'inizio: esce Java 8, con i nuovi lambda, quindi ci viene voglia di scrivere codice come questo:

```
List<T> lista;

lista.forEach(...);
```

con un blocco lambda fra i puntini. Sfortunatamente non possiamo farlo, perché `forEach` non è nell'interfaccia `List` e nemmeno nell'interfaccia

`Collection` da cui questa deriva. Si potrebbe modificare una di queste interfacce, ma si perderebbe la compatibilità con tutto il codice esistente.

L'estensione della sintassi delle interfacce che uscirà con Java 8 consente di aggiungere semplicemente in fondo al codice dell'interfaccia `Set`

```
public interface Set<T> extends
Collection<T> {
    public int size();
    // The rest of the existing
Set methods
    public void forEach(Block<T>
b)
        default
Collections.<T>forEach();
}
```

```
class Collections {
    ...
    public static<T> void
forEach(Set<T> set,
Block<T> block) {...}
}
```

Un modo di vedere queste novità è che Java diventa un pochino più simile a un linguaggio dinamico, come Python. Abbiamo già visto un'estensione importante dell'interfaccia delle collection, con l'introduzione della *Stream Api* di cui abbiamo parlato prima.

In definitiva

Java 8 sarà ricco di innovazioni importanti, alcune delle quali sono destinate a cambiare forma alla programmazione con insiemi, liste e collezioni. Si tratta di novità importanti e benvenute, che non comporteranno problemi per il codice esistente, pur trasformando un blocco fondamentale del linguaggio.

L'aggiunta dei blocchi lambda a Java ha richiesto un'importante revisione del linguaggio, che è tutto sommato un atto dovuto. Se anche il C++, uno dei linguaggi più ardui da innovare ha recepito costrutti tipici di Lisp, Javascript e Python, è giusto che anche Java non limiti i programmatori.

La programmazione con blocchi funzionali anonimi non è l'unica pratica che Java importa da Javascript, perché la piattaforma Java si arricchisce di un Javascript ufficiale, da cui ci si attendono ottime prestazioni, sarà interessante vedere se arriverà una versione su piattaforma Java di node.js, per esempio. •

