marco.canducci@studio.unibo
06/05/2018

Assignment #2
RegEx Matchings Search

Gruppo di lavoro

Il progetto è stato sviluppato cooperativamente da tre studenti: il sottoscritto Canducci Marco (matr. 833069), Cavina Eugenio (matr. 833893) e Gondolini Monica (matr. 855202). Durante le primissime fasi di sviluppo si è approfondito personalmente un argomento a testa: ogni componente del gruppo ha cercato di inquadrare ed impostare il problema seguendo una delle tre metodologie richieste dal testo dell'*assignment*. Dopo aver abbozzato le tre diverse analisi, si è però proseguito nello sviluppo in modo totalmente cooperativo, trovandosi di persona ed incentivando l'attiva partecipazione di ogni componente alle discussioni riguardanti analisi del problema, scelte tecnologiche e soluzioni implementative per ognuno dei tre esercizi.

Esercizio 1: Tasks & Executors

Analisi del problema e soluzioni proposte

Proposta n.1: "MainNioWalk.java"

In una prima analisi si è partiti dal presupposto che la navigazione del *FileSystem* sarebbe probabilmente risultata più *IO-bound* che *CPU-bound*: di conseguenza si è momentaneamente soprasseduto sulla sua parallelizzazione.

Per quanto riguarda il resto del programma, si vuole anticipare sin da subito che saranno utilizzati due *ExecutorService*, sui quali saranno riversati *task* con scopi diversi:

1. ***searchInFileExecutor***: in esso vengono lanciati i *SearchInFileTask*, aventi il compito di analizzare un *file* di testo e restituire un *Report* contenente il numero di *match* trovati ed il *path* del *file* esaminato. **NB**: Essendo un requisito del problema la visualizzazione di ogni nuovo *Report* appena disponibile, si è deciso di ***wrappare* questo *searchInFileExecutor* in un *ExecutorCompletionService***, grazie al quale è possibile effettuare la chiamata bloccante ***take()***, con lo scopo di ottenere il primo risultato appena pronto (senza abortire gli altri *task* e senza dover fare *polling!*). Per ulteriori informazioni si veda: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html.

2. ***searchInTextExecutor***: utilizzato unicamente nel caso si decida di permettere la suddivisione dei *file* di grosse dimensioni. Al suo interno vengono lanciati i *task* di tipo *searchInTextTask*, il cui compito è quello di analizzare il contenuto di una stringa e restituire un il numero di *match* trovati.

Riprendendo il discorso dalla navigazione del *FileSystem*: ogni qualvolta venissero incontrati dei *file* testuali verrà lanciato un nuovo *SearchInFileTask* (in particolare, se abilitata la divisione dei file, un *SearchInFileTaskSplit*) all'interno del *searchInFileCompletionService*. **NB**: Il **numero *N* di *file* trovati** (e quindi dei *task* lanciati) deve essere memorizzato.

Terminata la navigazione, viene richiamata una *shutdown()* sul *searchInFileExecutor* poiché nessuno dovrà più poter aggiungere task al suo interno (nè direttamente, nè attraverso il suo *wrapper*: *searchInFileCompletionService*).

Arrivato a questo punto, il *main thread* "*master*" otterrà iterativamente (**per N volte**) il primo risultato utile (appena pronto) interrogando il *searchInFileCompletionService* con una ***take()*** bloccante. In questo modo si evita di fare esplicitamente ***polling*** sul *searchInFileExecutor*, poiché, si presume, tale polling risulta incapsulato all'interno del *completion service*.

Per ogni *Report* ottenuto viene quindi stampato a video il nome del *file* con i corrispondenti *match* trovati (se presenti) e le statistiche aggiornate.

Un ultimo dettaglio: quando e da chi viene quindi utilizzato il *searchInTextExecutor?* E soprattutto, perché quest'ultimo non necessita di un suo *completion service*?

Tale *ExecutorService* viene istanziato dal *Main* ed utilizzato dai *SearchInFileTaskSplit* solamente nel caso venga abilitato il parametro *SPLIT_FILES* all'interno del file *Config.java*.
I *task SearchInFileTaskSplit* estendono i *SearchInFileTask* e suddividono il file in stringhe di dimensione massima *MAX_CHUNK_LEGTH* se necessario.
Ognuna di queste sotto-stringhe viene quindi assegnata ad un *SearchInTextTask*, dopodiché il *SearchInFileTask* attende che **tutti i *searchInTextTask* lanciati abbiano finito** il conteggio dei *match*, quindi somma tutti i singoli risultati e restituisce il totale nel *Report* in uscita. In questo caso quindi *wrappare* l'*executor* in un *completion service* risulterebbe inutile poiché sarebbe comunque necessario attendere la terminazione di tutti i *task* lanciati.

Proposta n.2: "MainFullTasks.java"

La seconda soluzione proposta rimane pressoché identica alla prima, eccezion fatta per la navigazione del FileSystem, che non risulta più sequenziale, ma viene qui parallelizzata con l'utilizzo di una **Fork/Join**.

Per il conteggio dei *file* testuali e per lanciare i corrispondenti *SearchInFileTask* viene infatti implementato ed utilizzato un ***SearchInFolderRecursiveTask*** (in precedenza tale compito era invece svolto nello *stream* generato dalla funzione di libreria *java.nio.file.Files.walk()*).

Possibili miglioramenti

Ci si ritiene particolarmente soddisfatti della suddivisione del lavoro pulita ed efficace tra i vari *Task* ed *Executor*; si ipotizza pertanto che miglioramenti rilevanti si possano invece apportare sia nella lettura dei *file* (si veda il capitolo a fine *report* intitolato "*Lettura e dimensione dei file*") che nella loro suddivisione in *chunk* più piccoli: sarebbe utile poter suddividere il loro contenuto in modo *safe* rispetto all'espressione regolare che si sta valutando.

Esercizio 2: Event Loop

In questa seconda parte di assignment si è scelto di utilizzare *JavaScript* e *NodeJS*, il cui approfondimento è stato ritenuto utile per questo ed altri corsi (quello dedicato alle tecnologie *web* in particolare), oltre che per numerosi ambiti applicativi.

Analisi del problema e soluzioni proposte

Proposta n.1: "index-callback.js"

Per prima cosa si è studiato l'esempio fornito nel laboratorio dedicato alle richieste HTTP con *NodeJS*, cercando di comprenderlo ed utilizzarlo come base di partenza. Si è quindi sviluppata una prima soluzione, la quale prevede l'esecuzione di una *callback* "*processLink()*" per ogni *response* ricevuta da richieste HTTP precedenti, il cui compito consiste nel:

1. Controllare che lo *status code* della *response* sia "OK" (codice 200).
2. Controllare che la *response* ricevuta corrisponda ad una risorsa di tipo testuale.
3. Ottenere e pulire tutti i *link* presenti nella pagina.
4. Per ogni *link*, controllare che non siano già state effettuate richieste su di lui e che il numero di richieste totali non abbia già superato il numero *MAX_REQUESTS. N*el caso tali condizioni fossero rispettate, effettuare una richiesta **asincrona** (**e parallela**) su tale link/*url* grazie al modulo *request*.
5. Cercare il numero di *match* presenti nella pagina corrente e, nel caso ve ne fossero, stampare su *standard output* tale numero, insieme all'url della pagina.
6. Aggiornare le statistiche e stamparle su *standard output*.

Sono risultati particolarmente comodi il supporto di *JavaScript* alle ***closure*** e la sequenzialità che caratterizza l'**Event Loop**: tutti i controlli effettuati nella *callback* sono stati implementati in maniera semplice e pulita, potendo accedere direttamente alle variabili dello *scope* padre direttamente dalla *callback* e senza doversi preoccupare di corse critiche o problemi di sincronizzazione.

Proposta n.2: "index-promise-rpn.js"

Nonostante la prima soluzione risultasse già sintetica e ben leggibile, si è voluto comunque realizzarne anche una versione "*promise*-ificata" sfruttando la libreria ***request-promise-native*** (https://www.npmjs.com/package/request-promise-native): grazie ad essa risulta possibile effettuare una richiesta ottenendo in cambio una *promise* nativa di *JavaScript*. Come si può intuire, questa seconda soluzione risulta molto simile alla prima (poiché essa comprendeva una sola *callback*); durante la sua implementazione si è però potuto avere un assaggio di come le *promise* possano risolvere un'eventuale ***pyramid of doom*** e di come spesso semplifichino **la gestione degli errori**.

Possibili miglioramenti

Nel caso si fosse voluto parallelizzare non solo l'invio delle richieste *HTTP* per tutti i *link* presenti in una pagina, ma anche la ricerca dei *match*, si sarebbe potuta realizzare una soluzione che utilizzasse ***child processes*** (https://nodejs.org/api/child_process.html) o anche ***web workers***, tenendo però in considerazione il fatto che questi ultimi risultano al momento supportati solo grazie a moduli di terze parti, come ad esempio:

- https://www.npmjs.com/package/webworker-threads
- https://www.npmjs.com/package/workerpool
- https://www.npmjs.com/package/tiny-worker

Esercizio 3: Reactive programming

Analisi del problema e soluzione proposta

Per sfruttare appieno la programmazione reattiva si è voluta sperimentare l'integrazione tra il *framework* di *RxJava* e *JavaFX*, con l'obiettivo molto interessante di realizzare un ***binding*** diretto tra componenti grafici e flussi di dati (***variabili reattive***).

Per prima cosa, alla pressione del tasto "*Search*" sulla *GUI*, vengono raccolti in una lista tutti i *file* contenuti nella cartella di partenza ed in tutte le sue sottocartelle, fino al massimo livello di annidamento *MAX_DEPTH*.

Tale lista diviene quindi il punto di partenza per la creazione del principale flusso di dati "*hot*" (di tipo ***event stream*** poiché discreto nel tempo), generabile in modo sequenziale o parallelo a discrezione dell'utente. Tale flusso sarà composto da una successione di oggetti di tipo *Report*, contenenti ognuno il percorso di un *file* analizzato ed il corrispondente numero di *match* trovati.

Sfruttando operazioni come ***map***, ***scan***, ***zip*** *e* ***filter***, vengono creati nuovi flussi osservabili partendo da quello iniziale o dalla combinazione di flussi intermedi; nella fattispecie, vengono creati degli ***event streams*** che rappresentano in ogni momento:

- Il numero di *file* già analizzati
- Il numero di *file* all'interno dei quali sono stati riscontrati dei *match*
- La percentuale di *file* analizzati dove sono stati trovati dei *match* (combinando i due flussi precedenti)
- Il numero totale di *match* trovati in tutti i *file* già analizzati
- Ecc.

Come ultima cosa viene effettuato il ***binding*** tra i componenti di *output* della *GUI* (*label*, *textArea*, ecc.) e gli opportuni ***event streams***.

NB: Sia che la computazione venga effettuata in modo sequenziale, sia che venga effettuata in modo parallelo, ci si è assicurati che la ricerca dei *match* all'interno dei *file*, sia le successive operazioni sui vari stream, non vengano portata avanti dal ***Java FX Application Thread***, ma da un *thread* separato. Nel caso anche la navigazione del *FileSystem* dovesse

essere ritenuta onerosa in termini computazionali sarà necessario pensare ad una soluzione atta ad affidare ad un *thread* separato anch'essa.

Parallelizzazione in ReactiveX

Un modo idiomatico per parallelizzare il lavoro in *ReactiveX* consiste nell'utilizzare l'operatore ***flatmap*** in combinazione con uno ***Scheduler***: così facendo, la *flatmap* permette di suddividere in modo efficace uno *stream* di eventi in uno *stream* di sotto-*stream* (per i dettagli implementativi si faccia riferimento al file *Controller.java*).

In alternativa, da *RxJava2.0.5* sono stati introdotti gli operatori ***parallel()*** e ***sequential()***, grazie ai quali è possibile parallelizzare e ri-sequenzializzare il flusso di esecuzione in maniera simile a quanto avveniva già nativamente in Scala.

Hot vs Cold Observables

In una implementazione iniziale della soluzione il flusso sorgente di dati "*reportsSource*" veniva ingenuamente generato in modo *cold* poiché non si era ancora a conoscenza del fatto che ad ogni nuovo *subscriber* tale flusso veniva completamente rigenerato. Questo comportamento causava ovviamente enormi sprechi a livello computazionale, oltre ad aggiornamenti visibilmente sequenziali su tutti gli elementi della GUI (veniva prima aggiornata un'etichetta incrementalmente fino all'ultimo valore, poi l'etichetta successiva, e così via). Dopo aver corretto la nostra implementazione (rendendo *hot* la sorgente iniziale) tutto ha iniziato a funzionare come previsto, ottenendo etichette ed aree di testo che si aggiornavano contemporaneamente all'arrivo di ogni nuovo risultato sullo stream a cui erano state *bind*-ate.

Possibili miglioramenti

Dopo aver risolto l'esercizio, si è cercato di collegare direttamente lo *stream* generato dalla funzione della libreria di *NewIO* (***java.nio.file.Files.Walk()***) al flusso osservabile di *RxJava* (***io.reactivex.observables.ConnectableObservable***) utilizzando la seguente libreria esterna: https://github.com/akarnokd/RxJava2Jdk8Interop#stream-to-rxjava.
Purtroppo, anche per problemi di tempo, non si è riusciti ad approfondire e verificare la fattibilità di questa trasformazione; ci si è quindi dovuti "accontentare" di generare il *ConnectableObservable* di RxJava partendo direttamente da una **lista** che raccogliesse prima tutti i risultati piuttosto che direttamente dallo *stream* di partenza.

Scelte e problematiche comuni

Parametri per la navigazione di alberi e grafi

Durante l'attraversamento di un albero, essendovi una naturale struttura gerarchica, si è ritenuto opportuno poter configurare il **livello massimo di profondità** raggiungibile rispetto al nodo di partenza. Nelle soluzioni del primo e del terzo esercizio (che lavorano su *FileSystem*) è stato pertanto introdotto il parametro *MAX_DEPTH* che indica la massima profondità delle sottocartelle esplorabili partendo dalla cartella iniziale.

Durante l'attraversamento di un grafo si è data invece una maggiore importanza al **numero massimo di nodi raggiungibili**, piuttosto che ad un ipotetica lontananza dal nodo iniziale. Nelle diverse soluzioni del secondo esercizio (dove si effettua *crawling* su pagine *web*) si è quindi utilizzato il parametro *MAX_REQUESTS* per indicare il numero massimo di richieste *http* effettuabili. Si vuole però sottolineare che per allontanarsi troppo dalla pagina di partenza si è deciso di implementare un approccio di navigazione ***Breadth-First*** piuttosto che *Depth-First*.

Filtraggio dei file di testo

Filtrare correttamente i *file* a contenuto testuale (eg: txt, html, css, xml, etc.) non è risultato un compito né semplice, né pulito o esatto.

- Lavorando su *FileSystem* (Esercizi 1 e 3), il metodo adottato è stato quello di controllare che il tipo di un *file* iniziasse con la parola "*text*": si è utilizzata la funzione di libreria ***Path.probeContentType(Path path)*** fornita da ***java.nio***.
- Lavorando sul *web* (Esercizio 2) si è invece deciso di implementare **un doppio controllo**, di cui il primo di natura preventiva: per prima cosa vengono esclusi tutti gli *url* le cui estensioni corrispondono a note tipologie di *file* non testuali (eg: jpeg, jpg, gif, png, pdf, mp4, etc.), dopodiché, una volta effettuata la richiesta e ricevuta risposta, ci si assicura che nell'*header* il *content-type* inizi con la stringa "*text/html*".

Lettura e dimensione dei file

Problematica per la quale è stato necessario effettuare una ben precisa scelta di *design*.
Le due alternative erano quelle di:

1. Considerare tutto il contenuto dei *file*
2. Suddividere i *file* in righe

Si è optato per la prima alternativa poiché non si voleva escludere a priori la possibilità di riconoscere con l'espressione regolare caratteri speciali come l'andata a capo (molto utile ad esempio per contare le righe di un documento).

D'altro canto si è consapevoli del fatto che la scelta compiuta preclude ogni tipo di politica atta a tamponare il problema della lettura di *file* di grosse dimensioni, poiché tale scelta presuppone che tutto il *file* sia presente in memoria per poter essere processato.

Si vuole infine precisare che nel primo esercizio (come da consegna) si è introdotta la possibilità di suddividere il lavoro legato a *file* di grosse dimensioni su più *Task*; tale funzionalità però, oltre ad introdurre il problema della **suddivisione non *safe* del file** rispetto all'espressione regolare, non attenua nemmeno il problema sopracitato riguardante la memoria di sistema disponibile. Infatti, nella soluzione presentata il *file* viene comunque letto nella sua interezza prima di essere *splittato* e suddiviso sui vari sotto-*task*. In una soluzione più elaborata si sarebbe potuto **mappare direttamente il *buffer* di caratteri** in lettura da disco sul *buffer* accettato dalla funzione di *matching* (*pattern.matcher()*) coordinando i sotto-*task* in modo tale che non partisse la computazione di molte porzioni di file contemporaneamente, e facendo quindi in modo che non fosse più necessaria la lettura dell'intero file prima di poterne analizzare i vari *chunk*.