Alma Mater Studiorum · Università di Bologna

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

campus di Cesena

Relazione PCD Assignment02: Parte 2

Brini Tommaso

Buizo Manuel

Rambaldi Riccardo

Anno Accademico 2023-2024

Indice

Capitolo 1

Analisi del problema

La seconda parte dell'assignment consisteva nello sviluppo di un sistema in grado di generare un report sulle occorrenze di una determinata parola all'interno delle pagine web, partendo da un indirizzo web specificato e considerando le pagine collegate ad esso fino a un certo livello di profondità.

Il problema appena descritto doveva essere affrontato in tre approcci diversi: il primo consisteva nell'implementare la soluzione sfruttando la programmazione asincrona ad eventi (**Event-Loop**), il secondo sfruttando la programmazione reattiva (**React**) e infine il terzo tramite l'utilizzo di **Virtual Threads**.

Per prima cosa quindi ci siamo concentrati su l'implementazione di una classe/libreria che fornisca un metodo asincrono getWordOccurrences, il quale accetterà come parametri l'indirizzo di partenza, la parola da cercare e la profondità massima, restituisca il conteggio delle occorrenze trovate.

Abbiamo poi implementato una GUI che utilizzando le diverse implementazioni asincrone permetta all'utente in maniera intuitiva e interattiva di specificare i parametri e di visualizzare i risultati in modo chiaro e real-time.

In ciascuna delle tre versioni, l'obiettivo principale è stato prestare particolare attenzione all'ottimizzazione e allo sfruttamento delle specificità di ogni approccio utilizzato, privilegiando un design e un'implementazione che massimizzino l'efficienza, la scalabilità del sistema e il riuso di codice.

Capitolo 2

Strategia risolutiva

Per affrontare il problema della ricerca delle occorrenze di una parola all'interno delle pagine web, abbiamo seguito il pattern MVC (Model-View-Controller). Cosi facendo, garantiamo una chiarra separazione delle responsabilità tra Model, View e Controller, facilitando manutenibilità e estendibilità del codice. Il Model si occupa della logica di gestione e recupero dei dati dalle richieste al web, la View gestisce la presentazione grafica dei risultati e il Controller coordina le interazione tra le prime due componenti. Inoltre, abbiamo provato a utilizzare dei meccanismi di caching temporanea in modo da evitare di ripetere operazioni costose su pagine web già analizzate, in quanto un eccessivo numero di richieste al web veniva inevitabilmente bloccato dal server. In particolare, abbiamo memorizzato l'url della pagina di partenza, evitando di rianalizzarla ogni volta che veniva incontrata a profondità diverse. In questo modo, abbiamo notato un complessivo miglioramento delle prestazioni del sistema. Per effettuare test massivi su profondità più elevate senza preoccuparci del blocco di richieste da parte del server web, abbiamo caricato l'intera Javadoc su un server locale e fatto partire ricerche dall'index iniziale. In questo modo, abbiamo testato l'architettura generale in maniera più approfondita e più rapidamente.

Per fare ciò, abbiamo implementato il Searcher con due tipi di logica diversi. La logica Local seleziona tutti gli href che non iniziano con *http*, aggiungendo al loro path il proprio path locale; la logica Remote invece seleziona solo gli href che iniziano con *http* e che effettivamente puntano a un'altra pagina web.

Entrambe le logiche controllano che i link trovati indirizzino a una pagina web corretta e testuale in modo da poter proseguire con la ricerca delle occorrenze della parola in input, escludendo tutti i link a immagini o documenti di altro tipo.

Capitolo 3

Architettura proposta

Il progetto è organizzato in diversi package ognuno contenenti specifiche responsabilità all'interno dell'intera applicazione.

- **Core**: Contiene l'effettiva libreria per la ricerca delle occorrenze della parola data a partire dalla pagina indicata e dai suoi link collegati. Contiene l'implementazione effettiva del Searcher e delle sue due logiche, Local o Remote. Implementa la funzionalità *History*, che permette di tenere traccia della cronologia delle ricerche precedenti, salvate in un file json specifico e ricaricate dalla GUI a ogni diverso lancio.

- **GUI**: Contiene tutti i componenti grafici dell'interfaccia utente (GUI), progettati per rendere l'applicazione reattiva e efficiente. Include caselle di input di testo per specificare indirizzo web, parola da ricercare e profondità massima a cui arrivare, una area di testo di output per visualizzare i risultati in modo chiaro e incrementale. Inoltre, contiene una sezione in cui possiamo visualizzare le performance ottenute sulle ricerche salvate nella history, analizzandole attraverso tempi di esecuzione, profondità e tipo diverso di logica del worker.

- **Web**: contiene l'implementazione del server locale che carica la Javadoc su cui effettuare test massimivi di ricerca. Inoltre, contiene l'implementazione del clientService che si occupa di effettuare e gestire le richieste del corpo delle pagine web. Offre due modalità di connessione: con **Vertx**, per la gestione asincrona degli eventi, può dare problemi quando le richieste sono eccessive in quanto si blocca il thread di vertx stesso; con **Jsoup** per l'analisi e l'estrazione asincrona del contenuto delle pagine

web, preferito in quanto gestisce meglio le eccessive richieste.

- **Worker**: contiene l'effettiva implementazione dei componenti attivi che si occupano della logica di ricerca. Propone tre approcci differenti, ognuno dei quali ottimizzati per sfruttare al meglio gli aspetti specifici della tecnica utilizzata. Discussi singolarmente nei successivi capitoli.

Questa architettura mira a garantire una chiara separazione delle responsabilità tra i diversi componenti dell'applicazione, consentendo una facile manutenzione, estensione e ottimizzazione delle varie funzionalità. Ogni componente è progettato per sfruttare al meglio le caratteristiche specifiche della tecnologia, garantendo prestazioni ottimali.

Capitolo 4

Event Loop

Per implementare l'approccio basato su Event-Loop, abbiamo utilizzato la libreria Vert.x per gestire in modo efficente le operazioni asincrone e la scalabilità delle varie richieste HTTP. Inizialmente, abbiamo pensato di aggiungere ogni nuovo evento (quindi ogni nuovo link trovato) all'event-bus di Vert.x, in modo che venisse poi eseguito in maniera asincrona. Successivamente, per gestire al meglio la fine della ricerca e la chiusura dell'applicazione, abbiamo optato sull'utilizzo del concetto di **Future** per gestire in modo efficiente le operazioni asincrone e la concorrenza.
Viene eseguito un blocco di codice all'interno di executeBlocking() di Vert.x per eseguire operazioni di lunga durata senza bloccare l'Event Loop principale. L'operazione di ricerca delle occorrenze della parola nella pagina corrente viene eseguita in modo asincrono all'interno di un Future, restituendo una lista di risultati. Per ogni risultato trovato, viene avviato un insieme di operazioni asincrone parallele per esplorare i link trovati in ogni pagina "figlia", fino a raggiungere la profondità massima.
L'utilizzo di CompletableFuture permette di gestire in modo efficiente la concorrenza delle operazioni asincrone e di attendere il completamento di tutte le operazioni "figlie" prima di continuare. La fine della ricerca verrà segnalata alla GUI tramite l'opportuno metodo.

L'implementazione dell'Event-Loop offre un modo efficiente per gestire le richieste di ricerca in modo asincrono e parallelo, sfruttando al meglio le caratteristiche di Vert.x e il concetto di Future. L'approccio ricorsivo e parallelo consente di esplorare efficacemente le pagine web collegate e di trovare le occorrenze della parola specificata fino alla profondità desiderata.

```java
@Override
public void addEventUrl(final SearchResponse response) {
    this.onResponseView(response);

    final Future<Void> promise = this.vertx.executeBlocking(() -> {
        final List<SearchResponse> responses = this.searcher().search(response);
        responses.forEach(this::onResponseView);

        final List<CompletableFuture<Void>> nestedFutures = responses.stream()
            .flatMap(res -> IntStream.range(1, res.maxDepth())
                    .mapToObj(i -> CompletableFuture.runAsync(() -> {
                        this.onResponseView(res);
                        final List<SearchResponse> list = this.searcher().search(res);
                        list.forEach(this::onResponseView);
                        System.out.println(list.size());
                        System.out.println("Depth: " + res.currentDepth());
                    })).toList();

        final CompletableFuture<Void> allNestedFutures = CompletableFuture.allOf(nestedFutures.toArray(new CompletableFuture[0]));
        allNestedFutures.join();
        return null;
    });

    promise.onComplete(handler -> {
        this.onFinishListener();
    });
}
```

Figura 4.1: Event-Loop

Capitolo 5

Virtual Thread

L'approccio con i Virtual Thread è un altra tecnica efficace e scalabile per gestire in modo concorrente le richieste di ricerca delle occorrenze di una parola nelle pagine web.
Come nel precedente approccio a Event-Loop, anche in questo caso abbiamo fatto uso di Future per gestire le operazioni asincrone e concorrenti. Nello specifico, in questo caso abbiamo utilizzato CompletableFuture. La creazione di un nuovo Virtual Thread viene eseguita attraverso il metodo *runAsync()* delle CompletableFuture, che consente di eseguire il blocco di codice in un thread separato in modo asincrono. Specifichiamo un Executor per controllare il pool di Virtual Threads eseguiti, in modo da gestire la concorrenza.

L'elaborazione delle richieste avviene all'interno dei Virtual Thread, consentendo l'esecuzione delle operazioni in modo concorrente e parallelo.

Viene eseguita una ricerca delle occorrenze della parola nella pagina corrente, e successivamente vengono avviate operazioni parallele per esplorare ulteriormente i link "figli". Una volta completate tutte le operazioni asincrone, viene notificata la fine della ricerca alla GUI tramite l'opportuno metodo. E' importante notare come l'uso delle CompletableFuture consenta di gestire il comportamento di tutte le operazioni "figlie" prima di eseguire il metodo di chiusura della ricerca, garantendo una terminazione ordinata del processo.

L'approccio parallelo e concorrente garantito dai Virtual Thread consente di sfruttare al massimo le risorse del sistema e di mantenere il processo di ricerca reattivo anche in presenza di un grande volume di richieste.

```java
@Override
public void addEventUrl(final SearchResponse response) {
    this.onResponseView(response);
    final CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
        final List<SearchResponse> responses = this.searcher().search(response);
        responses.forEach(this::onResponseView);

        final List<CompletableFuture<Void>> nestedFutures = responses.stream() Stream<SearchResponse>
                .flatMap(res -> IntStream.range(1, res.maxDepth())
                        .mapToObj(i -> CompletableFuture.runAsync(() -> {
                            this.onResponseView(res);
                            final List<SearchResponse> list = this.searcher().search(res);
                            list.forEach(this::onResponseView);
                            System.out.println(list.size());
                            System.out.println("Depth: " + res.currentDepth());
                        }, this.executor))
                ) Stream<CompletableFuture<...>>
                .toList();

        final CompletableFuture<Void> allNestedFutures = CompletableFuture.allOf(nestedFutures.toArray(new CompletableFuture[0]));
        allNestedFutures.join();
    }, this.executor);

    future.thenRun(this::onFinishListener);
}
```

Figura 5.1: Virtual-Thread

Capitolo 6

Reactive Programming

L'ultimo approccio implementato è la tecnica della programmazione reattiva. Per fare ciò, abbiamo utilizzato componenti reattivi quali Flowable e Schedulers di RxJava.

Viene utilizzato Flowable.just() per creare un'istanza di Flowable contenente l'indirizzo iniziale, che rappresenta il punto di partenza del processo di ricerca. Abbiamo utilizzato Schedulers.computation per eseguire il flusso di operazioni in quanto è ottimizzato per operazioni computazionalmente intensive.
Utilizzando metodi come *map* e *flatMap* vengono eseguite operazioni di ricerca per trovare le occorrenze della parola nella pagina corrente e nei sui link collegati. Una volta terminate tutte le operazioni di ricerca, viene comunicato alla Gui la fine della ricerca.
L'utilizzo di Flowable e Schedulers consente di gestire in modo efficiente le sequenze di eventi e di eseguire le operazioni in thread dedicati, garantendo prestazioni ottimali e reattività dell'applicazione.

Il modello reattivo consente di gestire in modo efficiente le operazioni di ricerca e di esplorazione delle pagine web, consentendo di mantenere una user experience fluida e reattiva anche in presenza di un grande volume di richieste.

```java
@Override
public void addEventUrl(final SearchResponse response) {
    this.onResponseView(response);
    this.searchObservable = Flowable.just(response).subscribeOn(Schedulers.computation());

    IntStream.range(0, response.maxDepth()).forEach(i ->
            this.searchObservable = this.searchObservable.map(res -> {
                final List<SearchResponse> list = this.searcher().search(res);
                list.forEach(this.searchReportSubject::onNext);
                list.forEach(this::onResponseView);
                return list;
            }).flatMap(Flowable::fromIterable));
    this.searchObservable.doOnComplete(this::onFinishListener).subscribe();
}
```

Figura 6.1: React

Capitolo 7

Prove di performance

Oltre alla GUI realizzata per vedere in modo incrementale tutti i link che vengono visitati ad ogni profondità, è stata realizzata un'area di analisi che permette di verificare, a parità di profondità, le varie performance ottenute (in termini di tempo) al variare della strategia scelta.

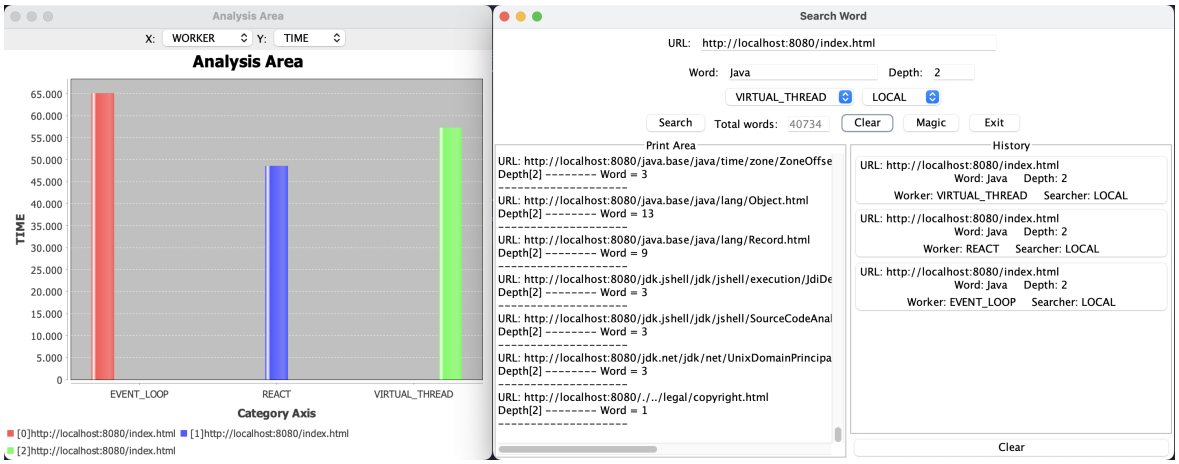

Figura 7.1: GUI realizzata e area di analisi performance

Dalla figura si può notare che il modello reattivo è molto più veloce a parità di profondità rispetto all'event-loop (di 15 secondi) e ai virtual thread (di quasi 10 secondi).