ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

23: Esecuzione alternativa

AMBIENTI

Abbiamo analizzato quattro diversi paradigmi, ovvero quattro diversi modi di organizzare il codice per ottenere diverse caratteristiche funzionali nella risoluzione di un problema.

La concorrenza ci permette di sfruttare meglio le risorse.

La distribuzione ci permette di espanderci oltre il singolo nodo di calcolo.

La reattività ci permette di minimizzare la latenza di risposta.

Il modello ad attori ci permette di astrarre oltre l'asincronia e la distribuzione.

In tutti questi paradigmi, tuttavia, abbiamo sempre lavorato nel consueto ambiente di esecuzione di Java:

Il programma viene tradotto in bytecode ed eseguito dalla JVM.

La JVM applica le strategie legate alla modalità JIT per bilanciare l'efficienza di esecuzione.

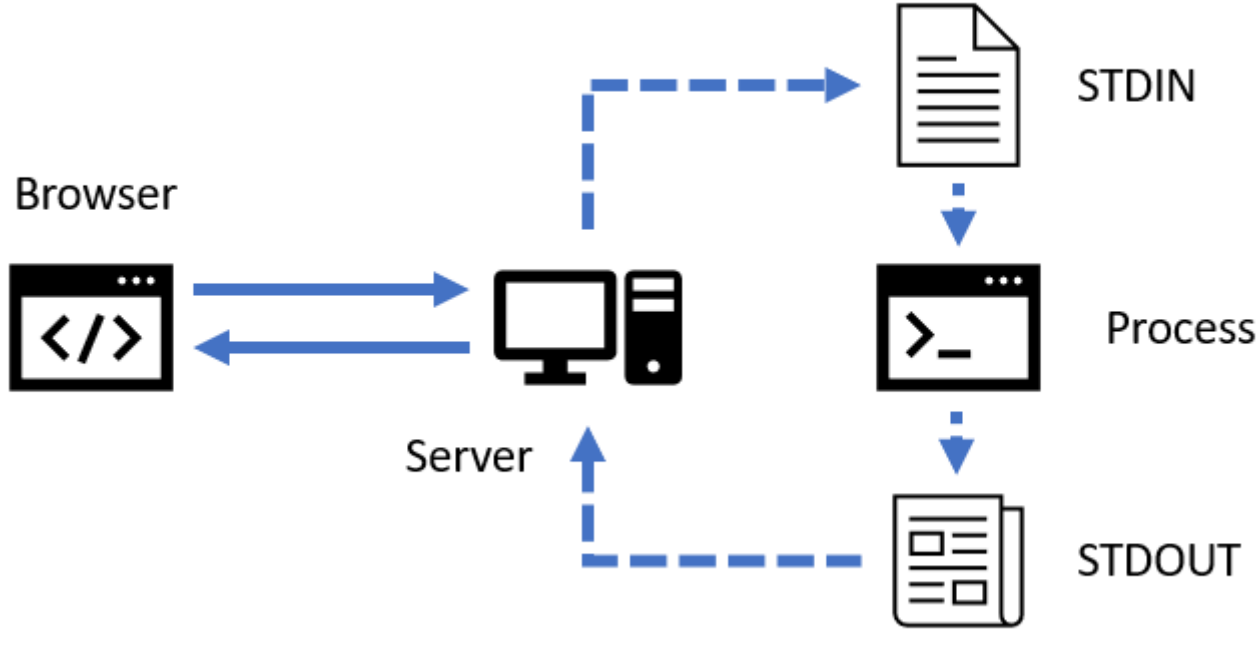
Questo però ha un costo in termini di tempo di avvio del programma.

Quando Java e la JVM sono stati inventati, questa scelta era vantaggiosa: la performance, a regime, raggiungeva livelli da accettabili a ottimi, e il codice era facilmente portabile.

Oggi le cose sono cambiate.

SERVER(LESS)

I primi siti web "dinamici" funzionavano nel modo seguente:



Questa modalità di funzionamento venne formalizzata nello standard
Common Gateway Interface

Molto presto diventa evidente la penalità di performance dovuta all'avvio di un processo esterno al web server.

Nasce l'application server, per mantenere sempre pronte alla risposta le applicazioni.

Java e la JVM quindi nascono in una situazione in cui è normale per un server rimanere in esecuzione, in attesa di richieste.

L'efficienza si persegue gestendo più applicazioni sulla stessa macchina, che ne condividono le risorse.

Dopo vent'anni però la situazione cambia:

AWS Lambda

Oggi, un servizio cloud cosiddetto *serverless* permette di acquisire le risorse necessarie all'esecuzione di una sola richiesta, pagando unicamente i millisecondi usati per costruire la risposta.

Il tempo di startup e la dimensione di un application server, modalità tipiche di erogazione di molti tipi di applicazioni, diventano uno svantaggio.

Un server non è più sempre in attesa delle richieste: esiste per il tempo strettamente necessario ad erogare una sola risposta.

Per la JVM è un problema particolarmente pressante: sebbene le prestazioni a regime siano di ottimo livello, lo diventano solo dopo diversi secondi di attività.

La strategia della JIT si trova in direzione opposta rispetto a questa situazione tecnologica.

GRAALVM

Il progetto GraalVM nasce come progetto di riscrittura del compilatore Java in Java stesso, per poi focalizzare sulla sola parte di frontend del compilatore.

In seguito a questo cambio di rotta, diventa un sistema completo che include:

- un nuovo compilatore JIT
- un compilatore ahead-of-time per Java
- una specifica di codice intermedio ("Truffle")

- un back-end che usa il sistema LLVM ed un runtime minimale per ottenere un eseguibile nativo da una rappresentazione Truffle
- strumentazione agnostica rispetto al linguaggio per il debug e l'ottimizzazione

GraalVM è in grado di compilare un qualsiasi bytecode da un linguaggio sulla JVM in un eseguibile nativo.

Attraverso Truffle, può supportare anche altri linguaggi.

La compilazione ahead-of-time, in un linguaggio come Java, ha sicuramente i suoi problemi: in particolare, il sistema di Classloading e le feature di reflection devono essere limitate per rendere prevedibile il codice disponibile al runtime.

Quello che si ottiene però è una sostenuta riduzione della dimensione dell'eseguibile finale (che comprende anche le parti necessarie della libreria standard) ma sopratutto del tempo di avvio

JAVA TO CONTAINER

Attraverso un framework che supporta GraalVM, possiamo adattare un nostro precedente esempio per osservare come ottenerne un eseguibile nativo già impacchettato in un container.

Scegliamo per questo esempio il framework Micronaut

Rispetto a VertX, Micronaut è un framework molto diverso: è particolarmente *opinionated* e propone una impostazione precisa di progetto, scegliendo all'interno dell'insieme delle tecnologie supportate le implementazioni delle funzioni necessarie.

In realtà questo approccio è dettato dalla necessità di selezionare ed integrare specificamente quei componenti che sono utilizzabili con GraalVM e compatibili con il processo di compilazione ahead-of-time.

Questo approccio è particolarmente evidente nel modo consigliato di preparazione di un progetto: un'apposita area del sito permette di generare un progetto d'esempio scegliendo i parametri tecnologici di cui si ha bisogno.

Al repository
https://hg.sr.ht/~michelemauro/app2020-mnaut
trovate il progetto d'esempio che andiamo ad analizzare.

Gran parte del progetto è analoga all'esempio TicTacToe che abbiamo già visto nelle precedenti lezioni. Il codice di dominio è stato copiato praticamente identico, a meno di riposizionamento del package.

La parte più interessante è l'interfacciamento con il framework, in particolare la dichiarazione delle rotte.

Micronaut ha in questo un approccio molto differente da VertX; in modo decisamente più tradizionale, usa delle *annotazioni*.

Il server si può avviare con `./gradlew run` per lanciarlo in modalità normale, eseguito sulla JVM.

Lanciando invece `./gradlew dockerBuildNative` si lancia la costruzione dell'immagine docker.

```
@Controller("/")
public class Server {

  private static final String GAME_NOT_FOUND = ...
  private static final String JOIN_FORM = ...
  private static final String WAIT_FOR_ANOTHER = ...

  static final ObjectMapper mapper = new ObjectMapper();
```

Tutte le classi annotate con `@Controller` vengono raccolte all'avvio del server e la struttura delle API esposte viene costruita analizzando il loro contenuto.

Questo lavoro viene fatto al runtime se il framework viene avviato normalmente (cioè come applicazione sulla JVM) o al momento della compilazione se viene generato un eseguibile nativo.

```
@Get(value = "/", produces = MediaType.TEXT_HTML)
public String welcome() {
  return JOIN_FORM;
}
```

Le annotazioni indicano i metodi che devono servire le varie richieste, specificando il pattern di URL a cui rispondono, i formati trattati e l'interfacciamento con la richiesta.

```
@Get(value = "/game/{playerId}",
  produces = MediaType.TEXT_HTML)
public HttpResponse< String > gameStatus(
  @PathVariable String playerId) {

  boolean open = gameServer.open(playerId);
  if (!open)
    return HttpResponse.notFound().body(GAME_NOT_FOUND);

  String result = gameServer.status(playerId)
    .map((res) -> render(playerId, res.idx, res.status))
    .orElse(String.format(WAIT_FOR_ANOTHER, playerId));
  return HttpResponse.ok().body(result);
}
```

Il pattern può contenere variabili, che possono essere trasformate direttamente in parametri del metodo. L'annotazione ha accesso al nome lessicale del parametro, e quindi può riconoscerlo dentro il pattern.

La risposta è costruita a partire dal valore di ritorno del metodo; se complessa deve essere contenuta in un oggetto adeguatamente espressivo.

```
@Post(
  value = "/game{/playerId}",
  produces = MediaType.TEXT_HTML,
  consumes = MediaType.APPLICATION_FORM_URLENCODED)
public HttpResponse< String > game(
  @PathVariable @Nullable String playerId,
  @Nullable Integer move) {

  if (playerId == null) {
    GameLocation location = ...;
    return HttpResponse.status(HttpStatus.TEMPORARY_REDIRECT)
      .header("Location", location.game)
      .body("");
  }
```

In questo caso la situazione è più complessa: il linguaggio del pattern non può esprimere come rotte differenti quella con e quella senza il parametro `playerId`, quindi dobbiamo gestire nel metodo la sua presenza.

Notate che abbiamo anche annotato il corrispondente parametro del metodo come potenzialmente nullo.

Nel caso in cui il parametro `playerId` manchi, stiamo rispondendo ad una richiesta `POST /game`.

Se invece è presente, stiamo rispondendo ad una richiesta `POST /game/123456`.

```
  boolean open = gameServer.open(playerId);
  if (!open) return HttpResponse.notFound()
    .body(GAME_NOT_FOUND);

  String result = ...
  return HttpResponse.ok().body(result);
}
```

Allo stesso modo per l'API che risponde in formato JSON.

```
@Post(value = "/game{/playerId}",
  produces = MediaType.APPLICATION_JSON)
public HttpResponse< String > gameApi(
  @PathVariable @Nullable String playerId,
  @Nullable Integer move) {

  if (playerId == null) {
    GameLocation location = ...;
    return HttpResponse.ok()
      .body(location.toJson(mapper));
  }
```

```
  boolean open = gameServer.open(playerId);
  if (!open) return HttpResponse.notFound()
    .body(GAME_NOT_FOUND);
  String result = gameServer.status(playerId)...
  return HttpResponse.ok().body(result);
}
```

Una volta completata la costruzione dell'immagine docker, questa può essere avviata:

```
docker run --rm -p8080:8080 tictactoe:latest
```

ALTRI RUNTIME

Esistono altri approcci per perseguire obiettivi simili a quelli di GraalVM, seguendo però strade differenti.

TornadoVM è un plugin per varie implementazioni di JVM che permette di scrivere codice Java in grado di essere eseguito su GPU, FPGA ed altri hardware eterogenei, accedendo a paradigmi di calcolo di solito dominio esclusivo di linguaggi di basso livello o molto specializzati.

KotlinNative si appoggia ad LLVM per produrre un eseguibile direttamente dal codice Kotlin, senza passare per la JVM.

La struttura del linguaggio e del suo compilatore permettono questo passaggio. Il risultato è un eseguibile nativo, compilato ahead-of-time time con un approccio però diverso da GraalVM.