

ALOM



ASYNCHRONISM

PROBLÉMATIQUES :

Exécuter des traitements longs sans bloquer
l'utilisateur 

Exécuter des traitements parallèlement plutôt que
séquentiellement

- Envois de mails 
- Impression de documents 

Ne pas bloquer l'utilisateur si l'imprimante n'a plus de
papier !

THE COST OF I/O :

The cost of I/O

L1-cache	3 cycles
L2-cache	14 cycles
RAM	250 cycles
Disk	41 000 000 cycles
Network	240 000 000 cycles

THE COST OF I/O :

Action	Latency	# of cycles	Human Time
1 Cycle CPU (3GHz Clock)	0.3 ns	1	1 s
Level 1 cache access	0.9 ns	3	3 s
Level 2 cache access	2.8 ns	9	9 s
Level 3 cache access	12.9 ns	43	43 s

THE COST OF I/O :

Action	Latency	# of cycles	Human Time
RAM access	70 - 100 ns	233 - 333	3.5 - 5.5 m
NVMe SSD	7 - 150 μ s	23k - 500k	6.5 h - 5.5 d
Rotational disk	1 - 10 ms	3 - 30 M	1 - 10 months
Internet: SF to NYC	40 ms	130 M	4.2 years



CONCURRENCE



Comment exécuter plusieurs choses en même temps ?

LE CPU

Un CPU exécute un seul process à la fois

Le système d'exploitation switch entre les process
pour leur donner le CPU

LES CPUS MULTI-COEURS



Exécution de plusieurs process en parallèle

Comment exécuter des choses en parallèle dans un même process ?

MULTITHREADING

Permet l'exécution de plusieurs tâches (threads) dans un même programme

Les threads se partagent la mémoire du process: 

EN JAVA

Un des premiers langages à rendre le multithreading
"facile" pour les développeurs

La gestion du multithreading se fait à l'aide des classes

```
java.util.concurrent.*
```

TOMCAT 🐱 (RAPPEL)

- instancie un `Thread` java par connexion HTTP
- voyez par vous-même dans votre code:

```
System.out.println(Thread.currentThread()).
```
- les `Threads` sont maintenus en vie dans un `pool`
- le nombre de thread est paramétrable pour booster 🚀 Tom (par défaut)
- si pas de thread dispo, les requêtes sont 'mises en attente'



- Les beans Spring sont des singletons par défaut!
- La mémoire entre les threads est partagée!
- Ne pas utiliser d'attributs de classe dans un bean spring : les valeurs seraient partagées entre tous les threads/requêtes HTTP !

INSTANCIER DES THREADS

implémenter l'interface Runnable...



```
1 @FunctionalInterface
2 public interface Runnable {
3     public abstract void run();
4 }
```

carbon
carbon.now.sh

INSTANCIER DES THREADS

... en java 8 avec une lambda



```
1 new Thread(() -> {  
2     System.out.println("Hello from thread " + Thread.currentThread().getName());  
3 }).start();
```

carbon
carbon.now.sh

⚠ à bien utiliser la méthode `start`

la méthode `run` exécuterait le code donné dans le thread "courant"



Comment récupérer un résultat ?

FUTURE, CALLABLE ET EXECUTORSERVICE

depuis Java ☕ 5 ! (si, si !)

- `Future<T>` représente un résultat asynchrone
- `Callable<T>` représente une tâche retournant un résultat
- `ExecutorService` gère l'exécution de tâches asynchrones `Runnable` ou `Callable<T>`

EXECUTORSERVICE

Exécution d'un Callable



```
1 // création d'un executorService
2 ExecutorService executorService = Executors.newFixedThreadPool(2);
3 // exécution d'une tâche dans un thread dédié avec récupération future d'un résultat
4 Future<String> futureResult = executorService.submit(() -> {return "test";});
5 // attente de la fin de la tâche (attente bloquante dans le thread "main")
6 String result = futureResult.get();
7 System.out.println(result);
```

carbon
carbon.now.sh



EXECUTORSERVICE

Exécution de plusieurs `Callable`

```
● ● ●  
1 ExecutorService executorService = Executors.newFixedThreadPool(2);  
2  
3 // création de callables  
4 Callable<String> a = () -> "a";  
5 Callable<String> b = () -> "b";  
6 Callable<String> c = () -> "c";  
7  
8 // exécution (cet appel est bloquant jusqu'a la fin des 3 callable!)  
9 List<Future<String>> futures = executorService.invokeAll(Arrays.asList(a, b, c));  
10 for(Future<String> future : futures){  
11     System.out.println(future.get());  
12 }
```

COMPLETABLEFUTURE

depuis Java ☕ 8

- `CompletableFuture<T>` est une Future
- Propose des méthodes de chaînage (proche des Promesses en ECMAScript)
- Propose des méthodes `static` pour exécuter des `Runnable/Callable` sans `ExecutorService`

COMPLETABLEFUTURE



```
1 // avec un runnable
2 CompletableFuture.runAsync(() -> System.out.println("Hello from Thread" + Thread.currentThread().getName()));
3
4 // pour récupérer un résultat (avec un Supplier)
5 CompletableFuture<String> futureString =
6     CompletableFuture.supplyAsync(() -> "Hello from Thread" + Thread.currentThread().getName());
7 System.out.println(futureString.get());
8
9 // pour lister la taille du pool par défaut
10 System.out.println(ForkJoinPool.getCommonPoolParallelism());
```

carbon
carbon.now.sh



COMPLETABLEFUTURE & STREAMS

Gérer le chargement asynchrone d'une liste

COMPLETABLEFUTURE

Taille du pool de threads par défaut : Nb CPU - 1

Permet de laisser un CPU disponible au Thread principal et à l'OS

Si un seul CPU dispo, pas de pool de Thread, un Thread sera créé pour chaque tâche

Attention au contexte d'exécution (docker) et à ce pool partagé

AVEC LES STREAMS JAVA 8

Possibilité d'exécuter des streams Java en parallèle -
Proche de l'asynchronisme

En séquentiel :



```
1 var pokemonTypesIds = List.of(1,2,3);  
2  
3 var pokemonTypesSync = pokemonTypesIds.stream()  
4     .map(pokemonTypeService::getPokemonType)  
5     .collect(Collectors.toList());
```

carbon
carbon.now.sh



AVEC LES STREAMS JAVA 8

En synchrone, chaque opération se fait dans le thread principal

```
1 2019-03-01 08:43:31 --- [main] RestTemplate : HTTP GET http://localhost:8080/pokemon-types/1
2 2019-03-01 08:43:31 --- [main] RestTemplate : Accept=[application/json, application/*+json]
3 2019-03-01 08:43:31 --- [main] RestTemplate : Response 200 OK
4 2019-03-01 08:43:31 --- [main] RestTemplate : Reading to [com.miage.altea.tp.pokemon_ui.pokemonTypes.bo.PokemonType]
5 2019-03-01 08:43:32 --- [main] RestTemplate : HTTP GET http://localhost:8080/pokemon-types/2
6 2019-03-01 08:43:32 --- [main] RestTemplate : Accept=[application/json, application/*+json]
7 2019-03-01 08:43:32 --- [main] RestTemplate : Response 200 OK
8 2019-03-01 08:43:32 --- [main] RestTemplate : Reading to [com.miage.altea.tp.pokemon_ui.pokemonTypes.bo.PokemonType]
9 2019-03-01 08:43:33 --- [main] RestTemplate : HTTP GET http://localhost:8080/pokemon-types/3
10 2019-03-01 08:43:33 --- [main] RestTemplate : Accept=[application/json, application/*+json]
11 2019-03-01 08:43:33 --- [main] RestTemplate : Response 200 OK
12 2019-03-01 08:43:33 --- [main] RestTemplate : Reading to [com.miage.altea.tp.pokemon_ui.pokemonTypes.bo.PokemonType]
```


AVEC LES STREAMS JAVA 8

En parallèle



```
1 var pokemonTypesIds = List.of(1,2,3);  
2  
3 var pokemonTypesSync = pokemonTypesIds.parallelStream()  
4     .map(pokemonTypeService::getPokemonType)  
5     .collect(Collectors.toList());
```

carbon
carbon.now.sh

AVEC LES STREAMS JAVA 8

En parallèle, chaque opération se fait dans le pool de thread. Le thread principal assure lui aussi sa part du travail



```
1 2019-03-01 08:45:56 --- [commonPool-worker-3] RestTemplate : HTTP GET http://localhost:8080/pokemon-types/3
2 2019-03-01 08:45:56 --- [main] RestTemplate : HTTP GET http://localhost:8080/pokemon-types/2
3 2019-03-01 08:45:56 --- [commonPool-worker-5] RestTemplate : HTTP GET http://localhost:8080/pokemon-types/1
4 2019-03-01 08:45:56 --- [main] RestTemplate : Accept=[application/json, application/*+json]
5 2019-03-01 08:45:56 --- [commonPool-worker-5] RestTemplate : Accept=[application/json, application/*+json]
6 2019-03-01 08:45:56 --- [commonPool-worker-3] RestTemplate : Accept=[application/json, application/*+json]
7 2019-03-01 08:45:56 --- [commonPool-worker-5] RestTemplate : Response 200 OK
8 2019-03-01 08:45:56 --- [main] RestTemplate : Response 200 OK
9 2019-03-01 08:45:56 --- [commonPool-worker-3] RestTemplate : Response 200 OK
10 2019-03-01 08:45:56 --- [commonPool-worker-5] RestTemplate : Reading to [PokemonType]
11 2019-03-01 08:45:56 --- [main] RestTemplate : Reading to [PokemonType]
12 2019-03-01 08:45:56 --- [commonPool-worker-3] RestTemplate : Reading to [PokemonType]
```

carbon
carbon.now.sh



QUEL INTÉRÊT POUR NOUS?

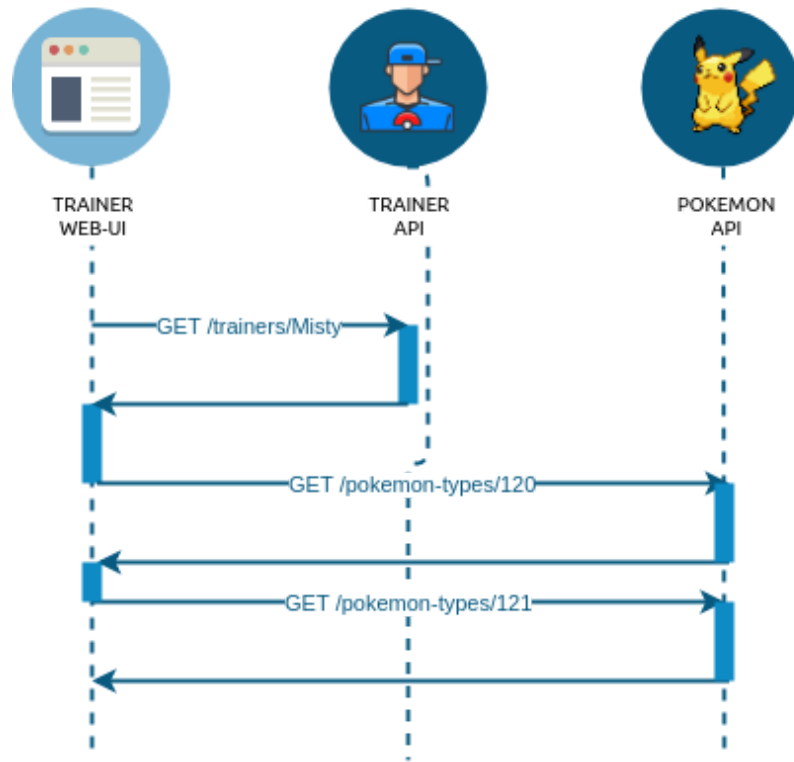


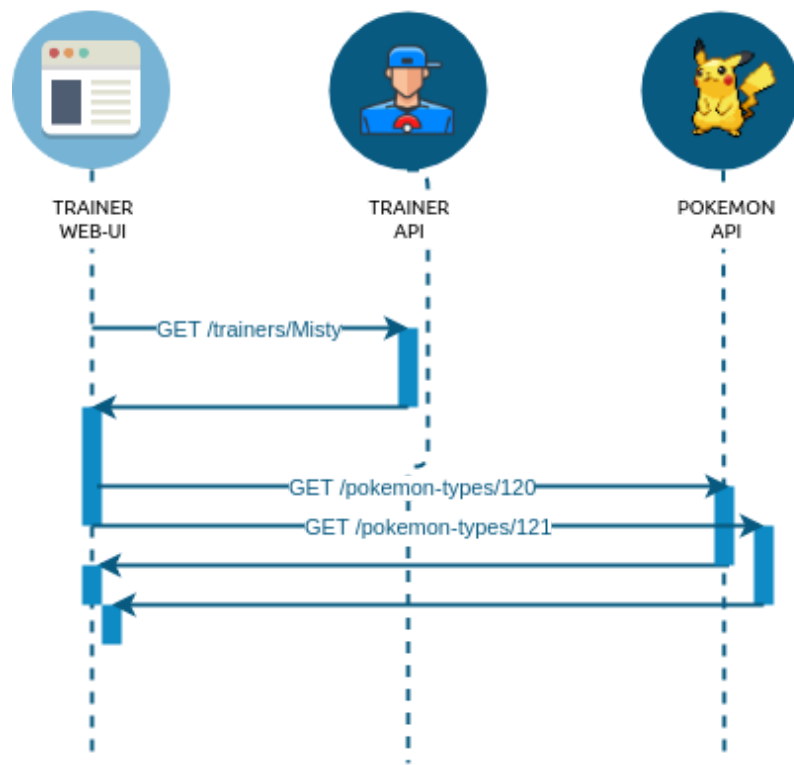
Je n'ai pas besoin de Threads, Tomcat 🐱 est déjà multithreadé

QUEL INTÉRÊT POUR NOUS?

Tomcat alloue un thread pour chaque requête entrante. Nous avons besoin de threads supplémentaires pour :

- Composition d'appels d'API
- Exécution de tâches longues
- Parallélisme au lieu de séquençement





GAIN DE TEMPS DE TRAITEMENT GLOBAL !



SPRING ASYNC

Exécution de tâches asynchrones avec l'annotation

`@Async`

```
@Configuration
@EnableAsync
public class AsyncConfig {
}
```

```
@Async
void doSomething() {
    // this will be run asynchronously
}
```

```
@Async
Future<PokemonType> doSomethingThatReturns(int i) {
    // this will be run asynchronously
}
```


SPRING ASYNC

Configuration du pool de threads

```
spring.task.execution.pool.max-size=16  
spring.task.execution.pool.queue-capacity=100  
spring.task.execution.pool.keep-alive=10s
```

SCHEDULING

Déclencher des tâches en fonction d'expressions crontab, ou d'un délai.

```
@Configuration
@EnableScheduling
public class SchedulingConfig {
}
```

```
@Scheduled(cron="*/5 * * * * MON-FRI")
void doSomethingOnWeekDays() {
    // something that should run on weekdays only
}

@Scheduled(fixedRate = 5, timeUnit = TimeUnit.SECONDS)
public void doSomethingEveryFiveSeconds() {
    // something that should run periodically
}
```

FIN DU COURS

Cours suivant :

High-Availability & micro-services patterns