Java8 est réactif !

Parmi les nombreuses évolutions que nous propose Java8, l'une d'entre-elles attire particulièrement notre attention. Il s'agit de la présence de la classe CompletableFuture<>.

Par Philippe PRADOS - 2014
www.prados.fr

Mine de rien, cette classe va bouleverser les applications Java. De nouvelles architectures seront proposées, de nouveaux frameworks vont apparaître pour remplacer les anciens, etc. C'est une classe majeure de Java8. Elle peut avoir autant d'impact que l'on été les annotations.

Architecture réactive

L'objectif des architectures réactives[1] est de ne plus utiliser des threads pour chaque traitement concurrent. Au lieu de multiplier les threads, exploitons les API asynchrones des OS. Il est bien plus efficace de distribuer la puissance des processeurs sur la base d'événement que sur une base temporelle. En fait, il y a deux sortes de threads : les soft threads qui simulent du multi-tâches en interrompant les traitements périodiquement, et les hards threads portés par les cœurs du processeur (ou l'hyper-threading dans un seul cœur). Une architecture réactive propose de n'utiliser que des hard-threads.

Le rapport avec CompletableFutures ?

La classe **`Future<>`** propose de déclencher un traitement en tâche de fond et de diffuser le résultat plus tard, dans le futur, via la méthode **`get()`**. Le développeur possède alors une référence vers un résultat futur. Il peut l'interroger pour savoir si le résultat est disponible, ou bien le demander immédiatement, car il en a besoin. La méthode **`get()`** bloque alors le thread actuel jusqu'à la fin du traitement asynchrone.

La classe **`CompletableFuture<>`** hérite de **`Future<>`** mais offre un nouveau paradigme. Il est maintenant possible de valoriser un **`Future<>`** par d'autres moyens que la fin du traitement en tâche de fond. Cela change beaucoup de choses. En effet, il est alors possible d'utiliser des API asynchrones pour débloquer un **`Future<>`**.

La classe **`CompletableFuture<>`** permet de généraliser ce modèle de développement. Cette classe est très (trop) riche et difficile à appréhender au premier abord. En fait, il y a quatre familles de méthodes : écriture, lecture, transformation et terminaison.

Les méthodes d'écritures permettent de valoriser un **`CompletableFuture<>`** avec une valeur (**`complete`**) ou une exception (**`completeExceptionaly`**).

La méthode **`supplyAsync()`** lance un traitement en tâche de fond et retourne immédiatement un **`CompletableFuture<>`** qui sera valorisé plus tard.

Les méthodes de lectures récupèrent les valeurs en mode bloquant ou non. Si une valeur est déjà disponible, elle est immédiatement retournée. Sinon, le thread courant bloque. Si une exception valorise le futur, elle est propagée lors d'un **`get()`**.

Utiliser des méthodes bloquantes n'est pas très réactif.

---

[1] http://blog.octo.com/tag/reactive/

Les méthodes de transformation se chargent de réagir à la valorisation d'un **CompletableFuture<>** pour produire une autre valeur. Cela permet d'enchaîner les transformations au fur et à mesure que les valeurs sont disponibles.

Les méthodes de terminaisons permettent de déclencher un traitement lors de la valorisation du futur.

Tout est synthétisé dans le tableau. Notez que chaque méthode possède une version avec le suffixe async pour être exécutée en tâche de fond, et une autre recevant une instance Executor pour exploiter un pool de thread spécifique.

Comment utiliser tout cela ? Partons d'un **CompletableFuture<>** un peu absurde qui retourne 42 en tâche de fond.

```
CompletableFuture<Integer> cf=CompletableFuture.supplyAsync(() -> 42);
```

Remarquez l'utilisation d'une Closure de Java8.

Il est maintenant possible de transformer le résultat en chaîne de caractères puis d'afficher le résultat.

```
cf.whenComplete((x,e) -> Integer.valueOf(x))
  .thenAccept((x) -> out.println(x));
```

La classe **CompletableFuture<>** permet la composition de traitement.

Le code est optimisé pour éviter autant que possible d'enchaîner les transformations sur un autre thread si ce n'est pas nécessaire. La transformation et la terminaison peuvent être effectuées dans le même thread présent dans le pool.

La version complètement asynchrone est très proche.

```
cf.whenCompleteAsync((x, e) -> Integer.valueOf(x))
  .thenAcceptAsync((x) -> out.println(x));
```

Ainsi, la conversion de l'entier en chaîne, puis l'affichage de la chaîne de caractères, s'effectuent dans des threads différents. En fait, le code utilise un pool de thread unique **ForkJoinPool.commonPool()**. Il est initialisé avec le nombre de cœurs disponible sur la plate-forme.

La composition permet également d'attendre deux **CompletableFuture<>** de types différents.

```
CompletableFuture<String> future=CompletableFuture.supplyAsync(
  () -> grosTraitementTresLong());
CompletableFuture<Integer> other=CompletableFuture.supplyAsync(() -> 42);
future.thenAcceptBoth(other,(x,y) -> { out.println(x+" "+y); } );
```

Ou le premier **CompletableFuture<>** parmi deux de même type.

```
CompletableFuture<Integer> future=CompletableFuture.supplyAsync(() -> 33);
CompletableFuture<Integer> other=CompletableFuture.supplyAsync(() -> 42);
future.acceptEither(other,(x) -> { out.println(x); } );
```

Google utilise cette approche lors de l'interrogation du moteur de recherche. La même demande est effectuée sur plusieurs serveurs. Le premier qui répond a gagné.

Il est possible d'effectuer une transformation à partir de deux **CompletableFuture<>**. Cela produit un nouveau **CompletableFuture<>** permettant d'enchaîner d'autres traitements.

```
CompletableFuture<String> future=CompletableFuture.supplyAsync(() -> "hello");
CompletableFuture<Integer> other=CompletableFuture.supplyAsync(() -> 42);
```

```
CompletableFuture<String> result=future.thenCombine(other,(x,y) -> x+y );
```

Toutes ces combinaisons permettent de rédiger des pipelines complexes de traitements asynchrones.

Utilisation avec une API bloquante

Imaginons un enchaînement de traitement et de transformation faisant intervenir des API bloquantes. L'idée est d'obtenir un flux sur une page Web. Tout d'abord, proposons un **`CompletableFuture<Reader>`** avec le flux.

```
// Implémentation asynchrone avec exception
CompletableFuture<Reader> getAsyncURL(URL url) {
  final CompletableFuture<Reader> result = new CompletableFuture<Reader>();


  Executors.callable(() ->

  {

    // Exécuté dans un autre thread

    URLConnection con;

    try {

      con = url.openConnection();

      result.complete(new InputStreamReader(con.getInputStream(),
        getEncoding(con)));

    } catch (Exception e) {

      result.completeExceptionally(e);

    }

  });

  return result;

}
```

Puis transformons le flux en chaîne de caractères dès que la page HTML est disponible.

```
CompletableFuture<String> page = getURLAsync(url)
  .thenApply((in) -> {
    try {

      return IOUtils.toString(in);

    } catch (Exception e) {

      return "";

  }

});
```

Notez la capture de l'exception pour transformer le résultat en une chaîne vide. Nous traiterons cela plus loin.

Il est alors possible de demander le contenu de la page. Utilisons un appel bloquant, même si ce n'est pas conseillé.

```
// Lecture bloquante du flux transformé en String

System.out.println(page.get());
```

Tant que le pool de thread n'est pas saturé, nous pouvant demander des pages en parallèle. Mais s'il est saturé, impossible d'avoir d'autres traitements en tâche de fond complémentaire. Il faut attendre la fin de l'un d'entre eux.

Utilisation d'une API non bloquante

Maintenant, rendons cela réactif.

Nous utilisons l'API AsyncHTTPClient. Cette API n'utilise pas les closures. Elle n'est pas encore compatible Java 8. Nous devons encore utiliser une inner classe avec deux méthodes **`onCompleted`** et **`onThrowable`**. Nous mappons juste ces méthodes vers leurs équivalents de **`CompletableFuture<>`**.

```
// Implémentation asynchrone avec exception
CompletableFuture<Reader> getURLAsync(URL url) throws IOException {

  CompletableFuture<Reader> rc = new CompletableFuture<Reader>();

  AsyncCompletionHandler handler=new AsyncCompletionHandler<Response>(){

    @Override

    public Response onCompleted(Response response) throws Exception{

      rc.complete(new InputStreamReader(response.getResponseBodyAsStream(),

      getEncoding(response.getContentType())));

    return response;

    }


    @Override

    public void onThrowable(Throwable t){

      rc.completeExceptionally(t);

    }

  };

  asyncHttpClient.prepareGet(url.toString()).execute(handler);

  return rc;

}
```

Et voilà. Notre application est maintenant non bloquante lors de la lecture de la page HTML. Elle peut gérer une multitude de connexions sans avoir à ajouter de soft-threads. C'est techniquement la couche Netty qui est prévenue de l'arrivée des réponses par l'OS, en mode asynchrone. Lorsque la page est disponible, le **`CompletableFuture<>`** est valorisé. Cela déclenche la suite des traitements. Il n'y a plus de limitation artificielle à cause de la taille du pool de thread.

Gestion des exceptions

Nous avons vu comment combiner des **`CompletableFuture<>`**. Mais comment gérer les erreurs lors des transformations ?

C'est un vrai problème. Lors des discutions sur Java8, ils n'ont pas réussi à se mettre d'accord sur la gestion des exceptions lors de l'utilisation des closures. Du coup, toutes les méthodes de **`CompletableFuture<>`** refusent d'utiliser une closure envoyant une exception.

Ce n'est pas étonnant, car les traitements sont exécutés sur des threads différents. Il n'y a rien à capturer dans le thread courant.

Il y a alors plusieurs approches possibles, inspirées du modèle fonctionnel, pour résoudre cette difficulté.

Le modèle fonctionnel est le suivant : une transformation doit produire un résultat quoi qu'il arrive.

Nous avons alors trois approches possibles :

- Retourner une valeur par défaut
- Retourner une instance Optional
- Retourner une instance Try

La première approche, la plus simple, consiste à capturer l'exception et la convertir en une valeur particulière. Par exemple, un reader contenant le message de l'exception ou une chaîne vide.

```
// Implementation monadic (un seul type de résultat)

CompletableFuture<Reader> getURLMonadic(URL url) {

  final Executor executor = ForkJoinPool.commonPool();

  return CompletableFuture.supplyAsync(() ->

  {

    URLConnection con;

    try {

      con = url.openConnection();

      return new InputStreamReader(con.getInputStream(), getEncoding(con));

    } catch (Exception e) {

      return new StringReader("");

    }

  });

}
```

La deuxième approche consiste à utiliser un type très courant en programmation fonctionnelle : **`Optional`**. C'est un type qui permet de s'affranchir du **`null`**. L'idée est d'avoir un type qui porte soit une valeur, soit rien.

```
// Implémentation asynchrone avec Optional
CompletableFuture<Optional<Reader>> getURLOptional(URL url) {

  return CompletableFuture.supplyAsync(() ->

  {

    try {

      URLConnection con = url.openConnection();

      return Optional.of(new InputStreamReader(con.getInputStream(),
        getEncoding(con)));

    } catch (IOException e) {

      return Optional.empty();

    }
```

```
  });
}
```

Une transformation peut alors s'écrire comme ceci :

```
CompletableFuture<Optional<String>> pageOrEmpty = getURLOptional(url)
  .thenApply((in) -> {
    try {
      if (!in.isPresent()) return Optional.empty();
      return Optional.of(IOUtils.toString(in.get()));
    } catch (Exception e) {
      return Optional.empty();
    }
  });
```

in est du type **`Optional<Reader>`**. Il est possible qu'il n'y ait pas de valeur. La transformation retourne une instance **`Optional<String>`**.

La dernière approche consiste à utiliser une librairie équivalente à celle proposée par Twitter pour Scala : une classe **`Try<>`** qui porte soit une valeur, soit une exception. Cette classe n'existe pas dans Java8. Yohan Legat de Zenika propose une implémentation ici : http://goo.gl/5SmBhs.

```
// Implémentation asynchrone avec exception
CompletableFuture<Try<Reader>> getURLTry(URL url) {
  return CompletableFuture.supplyAsync(Try.of(() ->
  {
    URLConnection con = url.openConnection();
    return new InputStreamReader(con.getInputStream(), getEncoding(con));
  }));
}
```

La transformation est alors proche de l'approche **`Optional<>`**. Une méthode pouvant générer une exception est invoquée via **`Try.of(closure)`**. Cela permet de transformer le résultat en instance **`Try`** avec la valeur ou l'exception.

```
CompletableFuture<Try<String>> pageOrException=getURLTry(url)
  .thenApply(
    Try.of(
      (in) -> IOUtils.toString(in.getOrThrow())
    )
);
```

La méthode **`getOrThrow()`** propage l'exception si nécessaire.

Try.of()

Pour les lecteurs intéressés, la méthode **`Try.of()`** est codé ainsi :

```
public static <T> Supplier<Try<T>> of(ThrowingSupplier<T> function) {
    return () -> {
        try {
            T result = function.get();
            return new Success<>(result);
        } catch (RuntimeException e) {
            throw e; // we don't want to wrap runtime exceptions
        } catch (Exception e) {
            return new Failure<>(e);
        }
    };
}
```

La classe **`ThrowingSupplier<>`** est une classe équivalente à la classe **`Supplier<>`** de Java8, mais acceptant une closure pouvant produire une exception.

```
@FunctionalInterface
public interface ThrowingSupplier<T> {
    T get() throws new Exception;
}
```

Lors de l'invocation de **`Try.of()`** avec une closure sans paramètre et pouvant emmètre une exception, le compilateur va produire une instance **`ThrowingSupplier<>`** correspondant. Le code suivant :

```
Try.of((x) ->
  {
    if (true) throw new Exception();
    return 0;
  });
```

est compilé ainsi :

```
Try.of(new ThrowingSupplier<Integer>()
  {
    Integer get() throws Exception
    {
       if (true) throw new Exception() ;
       return 0 ;
    }
  }) ;
```

Cela produit au final une instance **`Supplier<>`** initialisée avec une closure. Le code de **`Try.of()`** revient à ceci :

```
public static <T> Supplier<Try<T>> of(ThrowingSupplier<T> function) {
    return new Supplier<Try<T>>() {
        public Try<T> get() {
            try {
                T result = function.get();
                return new Success<>(result);
            } catch (RuntimeException e) {
              throw e; // we don't want to wrap runtime exceptions
            } catch (Exception e) {
               return new Failure<>(e);
            }
        }
   };
}
```

Le compilateur génère une sous-classe de **`Supplier<Try<T>>`** et valorise l'unique méthode **`get()`**.

L'écriture avec les closure est quand même plus sympathique.

Pour conclure

La classe classe **`CompletableFuture<>`** ouvre la porte à la programmation réactive sous Java 8. Nous pensons que ce modèle de développement va se généraliser dans un avenir très proche. Des outils comme node.js ou des langages comme Scala proposent d'utiliser uniquement ce modèle événementiel.

Philippe PRADOS article@prados.fr
Équipe Réactive – OCTO Technology

Entrée	**Méthode**	**Paramètre princ.**	**Sortie**
Écriture			
Valorise un futur avec une valeur.			
`CF<T>,t`	`complete`	`T`	`CF<T>`
Valorise un futur avec une exception.			
`CF<T>,e`	`completeExceptionaly`	`e`	`CF<T>`
Valorise en tâche de fond va la création d'un CF.			
	`CF.supplyAsync`	`()->T`	`CF<T>`
Lecture			
Bloque le thread jusqu'à obtention de la valeur ou d'une exception. Attention, bloquant !			
`CF<T>`	`get`		`T`
Bloque le thread jusqu'à obtention de la valeur, d'une exception ou l'expiration d'un délai. Attention, bloquant !			
`CF<T>`	`getTimeOut`	`delais`	`T`
Retourne la valeur disponible ou une valeur par défaut.			
`CF<T>`	`getNow`	`default`	`T`
Se synchrone sur l'un d'eux et retourne sa valeur en Object.			

`CF<T>`
`CF<U>`
…
	`anyOf`		`CF<Object>`

Se synchrone sur plusieurs CF. Il faut les consulter individuellement ensuite.			

`CF<T>`
`CF<U>`
…
	`allOf`		`CF<Void>`

Transformation			

Transforme en valeur depuis une valeur ou une exception			
CF<T>	whenComplete	(T,e) -> CF<T>	CF<T>
Transforme un résultat en un autre.			
CF<T>	thenApply	(T) -> U	CF<U>
Transforme une exception en valeur.			
CF<T>	exceptionally	(e)->T	CF<T>
Applique une transformation avec l'un ou l'autre.			
CF<T> CF<T>	applyEither	(T)-> U	CF<U>
Combine deux résultats pour en faire un troisième.			
CF<T> CF<U>	thenCombine	(T,U) -> V	CF<V>
Transforme une donnée via un traitement retournant un CF.			
CF<T>	thenCompose	T -> CF<U>	CF<U>
Terminaison			
Exécute après une valorisation ou une exception.			
CF<T>	handle	(T,e) -> {…}	
Exécute dès la valeur disponible.			
CF<T>	thenAccept	(T)-> {…}	
Exécute après l'un et l'autre.			
CF<T> CF<U>	thenAcceptBoth	(T,U) -> {…}	
Exécute après l'un ou l'autre.			
CF<T> CF<T>	acceptEither	(T) -> {…}	