Futures

Jean-Luc Falcone

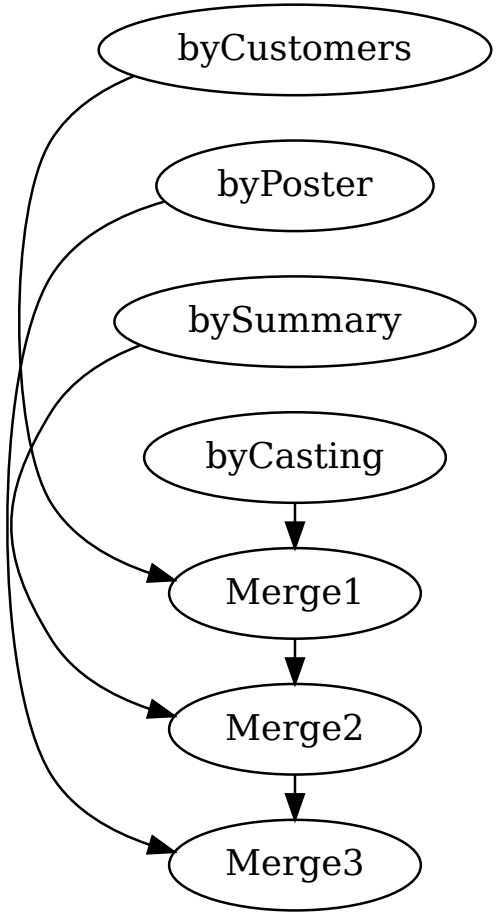
HEPIA - 2013

Problème

Dans une application de vente de DVDs, on aimerait suggérer des films similaires à celui que le client consulte.
On veut pour cela combiner quatres méthodes différentes:

1. Films similaires selon les autres clients
2. Films similaires selon le casting
3. Films similaires selon le résumé
4. Films similaires selon l'affiche

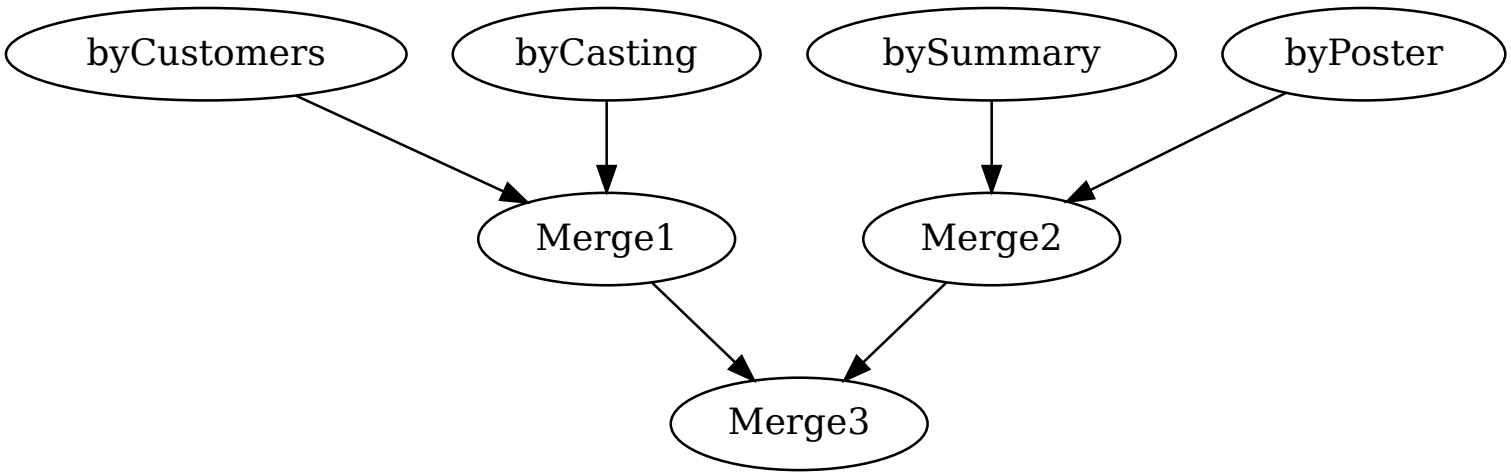
Version séquentielle



Implémentation séquentielle

```
def sequential( m: Movie ): SimilarMovies = {
  val s1 = byCustomers(m)
  val s2 = bySummary(m)
  val s3 = byCasting(m)
  val s4 = byPoster(m)
  val m1 = merge( s1, s2 )
  val m2 = merge( s3, m1 )
  merge( s4, m2 )
}
```

Version parallèle



Les Futures

La méthode future:

- exécute un bloc de code de manière parallèle:
- retourne une référence vers un résultat à venir (`Future[T]`)

```
import scala.concurrent._
import ExecutionContext.Implicits.global

val f1: Future[Int] = future{ 2+2 }

val f2: Future[Map[String,Int]] =
  future{ wordCount(text,stopWords) }
```

Attendre le résultat

On peut attendre le résultat avec la méthode `Await.result`:

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import scala.concurrent.duration._

val f2: Future[Map[String,Int]] =
  future{ wordCount(text,stopWords) }

val result: Map[String,Int] =
  Await.result( f2, 1.minute )
```

Modèle d'exécution: *Threads*

Les *threads* sont considérés comme des processus léger.
Cependant:

- Leur création est couteuse
- Commuter d'un *thread* à un autre est couteux

Modèle d'exécution: *Threads pool*

On peut instantier un thread par CPU (coeur) et leur faire exécuter un ensemble de tâches.

```
Callable<MovieResult> bySimilarity( final Movie m ) {
  return new Callable<MovieResult>{
    public MovieResult call() { /* ... */ }
  };
}

ExecutorService pool =
  Executors.newFixedThreadPool(poolSize);

Future<MovieResult> fs1 =
 pool.submit( bySimilarity(movie) );
MovieResult mr = fs1.get();
```

Threads pool et code bloquant

```
Callable<MovieResult> merge(
  final Future<MovieResult> f1,
  final Future<MovieResult> f2
) {
  return new Callable<MovieResult> {
    MovieResult call() {
      return doMerge( f1.get, f2.get );
    }
  };
}
```

Threads pool et code bloquant: Utilisation

```
Future<MovieResult> fs1 = pool.execute( byCustomers(m) );
Future<MovieResult> fs2 = pool.execute( byCasting(m) );

Future<MovieResult> m1 = pool.execute( merge( fs1, fs2 ) );
```

Call-back asynchrone (et parallèle)

Sauver le résultat dans un fichier.

```
val f2 = future{ wordCount(text,stopWords) }

f2.foreach{ res =>
  storeWordCount( res, "napoleon.histogram.txt" )
}
```

Transformation asynchrone (et parallèle)

Obtenir les 20 mots les plus employés (par fréquence décroissante).

```
val f2 = future{ wordCount(text,stopWords) }

val mostCommon: Future[List[String]] = f2.map{ res =>
  res.toList
    .sortBy( _._2 )
    .reverse
    .take(20)
    .map( _._1 )
}
```

Combiner des futures

```
val f1 = future{ 1+1 }
val f2 = future{ 3*4 }
val f3 = future{ 10-2 }

val f4: Future[Int] = for{
  x1 <- f1
  x2 <- f2
  x3 <- f3
} yield x1*x2 - x3
```

Implémentation parallèle

```
def parallel( m: Movie ): Future[SimilarMovies] = {
    val fs1 = future{ byCustomers(m) }
    val fs2 = future{ bySummary(m) }
    val fs3 = future{ byCasting(m) }
    val fs4 = future{ byPoster(m) }

    val fm1 = for(  s1 <- fs1; s2 <- fs2 )
      yield merge(s1,s2)
    val fm2 = for(  s3 <- fs3; s4 <- fs4 )
      yield merge(s3,s4)

    for(  m1 <- fm1; m2 <- fm2 )
    yield merge(m1,m2)
}
```

Performances

Machine	Cores	Sequentiel	Parallel
Laptop	2	4.9 (1x)	2.9 (1.7x)
Desktop	4	4.9 (1x)	1.6 (3x)
Baobab	16	4.9 (1x)	1.6 (3x)