

Le Modèle d'Acteurs

Jean-Luc Falcone

11 juin 2013

Problème: gérer un compteur de métriques

On aimerait gérer un compteur de métriques pour surveiller l'exécution d'un système distribué. Par exemple:

- Nombre de requêtes
- Temps dévolu à un certain calcul
- Accès au cache
- ...

Resources partagées

Pour mettre à jour une valeur, il faut le faire de manière atomique.

Implémentation classique (Java)

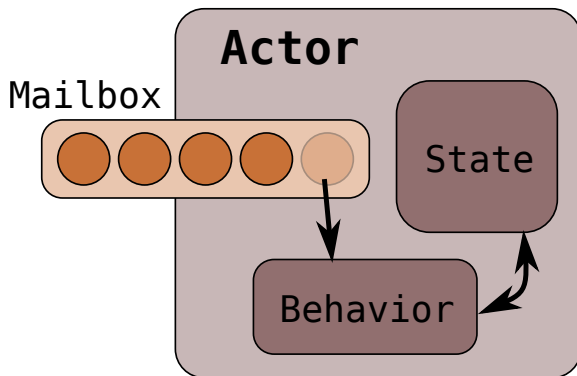
```
class Metrics {  
    private final Map<String,Long> metrics =  
        new HashMap<>();  
  
    public synchronized  
    void increment( String key, long l )  
  
    public synchronized long get( String key )  
  
    public synchronized long reset( String key )  
    public synchronized long resetAll()  
  
    public synchronized  
    Map<String,Long> currentSnapshot()  
}
```

Implémentation classique, suite (Java)

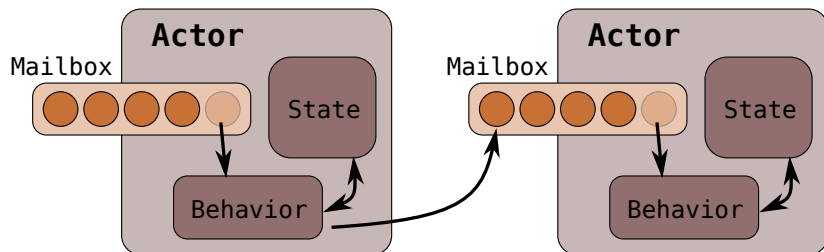
```
public synchronized
void increment( String key, long l ) {
    if( ! metrics.containsKey( key ) )
        metrics.put( key, l );
    else
        metrics.put( key, l + metrics.get(key) );
}

public synchronized Map<String,Long> currentSnapshot {
    final Map<String,Long> m = new HashMap<>();
    for( k: metrics.keySet() ) {
        m.put( k, metrics.get(k) );
    }
    return m;
}
```

Acteur



Deux acteurs



Composants

Messages	Peut être de n'importe quel type. Immutable
Boîtes-aux-lettres	Gérée à l'extérieur de l'acteur.
Etat	Unique à l'acteur. Peut être mutable .
Comportement	Réaction vis-à-vis des messages entrants.

Comportement

Le comportement d'un acteur est uniquement **réactif**.

Il peut consister en (non-exclusif):

- Mettre à jour l'état
- Répondre à l'expéditeurs
- Envoyer d'autres messages à d'autres acteurs
- Créer d'autres acteurs
- Se terminer
- Terminer le système en entier

Implémentations

Il existe plusieurs implémentations du modèle d'acteurs en Scala (ou dans d'autres langages).

Il est recommandé d'utiliser la librairie **Akka**.

Messages

```
object MetricMessages {  
  // Commands  
  case class Increment( key: String, l: Long )  
  case class Reset( key: String )  
  case object ResetAll  
  
  // Queries  
  case class Get( key: String )  
  case object CurrentSnapshot  
  
  // Results  
  case class Metric( key: String, value: Long )  
  case class Snapshot( map: Map[String,Long] )  
}
```

Acteur Définition

```
class MetricsActor extends Actor {  
  
  private val metrics = new mutable.HashMap[String,Long]  
  
  def receive = {  
    case Increment(k,l) => /* ... */  
    case Get(k) => /* ... */  
    case Reset(k) => /* ... */  
    case ResetAll => /* ... */  
    case CurrentSnapshot => /* ... */  
  }  
  
}
```

Acteur Définition

```
def receive = {  
  
  case Increment(k,l) => {  
    val before = metrics.getOrElse(k,0)  
    metrics += k -> (before + 1 )  
  }  
  
  case Reset(k) => metrics += k -> 0  
  
}
```

Exemple d'utilisation

```
val system = ActorSystem("MyHugeSystem")

val metrics = system.actorOf( Props[MetricsActor] )

metrics ! Increment("foo", 2)
metrics ! Increment("bar", 1)
metrics ! Increment("foo", 3)
metrics ! ResetAll
```

Répondre à un acteur

```
def receive = {  
  
  case Get(k) =>  
    sender ! Metric( k, metrics.getOrElse(k,0) )  
  
  case CurrentSnapshot =>  
    sender ! SnapShot( Map( metrics:_* ) )  
  
}
```

Exemple d'utilisation depuis un acteur

```
class ReportingActor extends Actor {  
  
  def receive = {  
    case WriteReport( metrics ) =>  
      metrics ! CurrentSnapshot  
  
    case SnapShot( map ) => {  
      //Process and Write Report  
    }  
  }  
}
```

Exemple d'utilisation en dehors d'un acteur

```
import akka.pattern.ask

val foo: Future[Long] =
  ( metrics ? Get("foo") ) map {
    case Metric( "foo", value ) => value
  }

val snapshot: Future[Map[String,Int]] =
  ( metrics ? CurrentSnapshot ) map {
    case SnapShot( map ) => map
  }
```


Cas complexe

