3.8 Persistence

Objective

Akka persistence enables **stateful actors** to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster.

Key Concept

The key concept behind Akka persistence is that **only changes** to an actor's internal state are persisted but never its current state directly (except for optional **snapshot**)

- The changes are only appended to storage and immutated.
- Can be either the full history or starting from a snapshot
- from eventsourced lib
- Event Sourcing by Martin Fowler

Event Sourcing的概念： (version control system is the popular case)

> The key to Event Sourcing is that we guarantee that all changes to the domain objects are initiated by the event objects. This leads to a number of facilities that can be built on top of the event log:
>
> Complete Rebuild: We can discard the application state completely and rebuild it by re-running the events from the event log on an empty application.
>
> Temporal Query: We can determine the application state at any point in time. Notionally we do this by starting with a blank state and rerunning the events up to a particular time or event. We can take this further by considering multiple time-lines (analogous to branching in a version control system).
>
> Event Replay: If we find a past event was incorrect, we can compute the consequences by reversing it and later events and then replaying the new event and later events. (Or indeed by throwing away the application state and replaying all events with the correct event in sequence.) The same technique can handle events received in the wrong sequence - a common problem with systems that communicate with asynchronous messaging.

Dependencies

```
"com.typesafe.akka" %% "akka-persistence" % "2.4.2"
```

built-in persistence plugins, including in-memory heap based journal, local file-system based snapshot-store and LevelDB based journal. For LevelDB, requires additional jar files:

```
"org.iq80.leveldb"            % "leveldb"          % "0.7"
"org.fusesource.leveldbjni"   % "leveldbjni-all"   % "1.8"
```

LevelDB

LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.

Architecture

PersistentActor

Is a persistent, stateful actor to be able to persist events to a journal. When a persistent actor is restarted, journaled messages are replayed to that actor to recover internal state.

PersistentView (deprecated from 2.4.2)

A view is a persistent, stateful actor that receives journaled messages that have been written by another persistent actor. A view itself does not journal new messages, instead, it updates internal state only from a persistent actor’s replicated message stream.

AtLeastOnceDelivery

To send messages with at-least-once delivery semantics to destination. You can mix-in AtLeastOnceDelivery trait to your PersistentActor on the sending side.

AsyncWriteJournal

A journal stores the sequence of messages sent to a persistent actor. Journal maintains highestSequenceNr that is increased on each message. The storage backend of a journal is pluggable. The persistence extension comes with a “leveldb” journal plugin, which writes to the local filesystem.

Sanpshot store

A snapshot store persists snapshots of a persistent actor’s or a view’s internal state.

Event Sourcing

Def: **Capture all changes to an application state as a sequence of events.**

1. Receive a (non-persistent) command
2. Validate command see if it can be applied to the current state. **Validation** can mean anything, from simple inspection of a command's message up to a conversation with external services.
3. If validation succeeds, events are generated from the command.
4. These events are then persisted
5. After successful persistence, change the actor's state.
6. Only persisted events are replayed during recovery.
7. Event sourced actors may process commands that do not change state such s **query** command.

p.s. 有command才有event

`PersistentActor` trait

An actor that extends this trait uses the persist method to **persist** and handle events. The behavior of a PersistentActor is defined by implementing **receiveRecover** and **receiveCommand**.

```scala
import akka.actor._
import akka.persistence._
case class Cmd(data: String)
case class Evt(data: String)
case class ExampleState(events: List[String] = Nil) {
  def updated(evt: Evt): ExampleState = copy(evt.data :: events)
  def size: Int = events.length
  override def toString: String = events.reverse.toString
}
class ExamplePersistentActor extends PersistentActor {
  override def persistenceId = "sample-id-1"
  var state = ExampleState()
  def numEvents = state.size

  def updateState(event: Evt): Unit =
    state = state.updated(event)

  val receiveRecover: Receive = {
    case evt: Evt                                => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  val receiveCommand: Receive = {
    case Cmd(data) =>
       persist(Evt(s"${data}-${numEvents}"))(updateState)
       persist(Evt(s"${data}-${numEvents + 1}")) { event =>
         updateState(event)
         context.system.eventStream.publish(event)
       }
    case "snap"  => saveSnapshot(state)
    case "print" => println(state)
  }
}
```

The persistent actor's *receiveRecover* method defines how state is updated during recovery by handling **Evt** and **SnapshotOffer** messages. The persistent actor's *receiveCommand* method is a command handler.

In this example, two events are persisted by calling persist with an event (or a sequence of events) as first argument and an event handler as second argument.

The persist method persists events asynchronously and the event handler is executed for successfully persisted events.

The sender of a persisted event is the sender of the corresponding command. This allows event handlers to reply to the sender of a command (not shown).

The main responsibility of an event handler is changing persistent actor state using event data and notifying others about successful state changes by publishing events.

It's also possible to switch between different command handlers during normal processing and recovery with context.become() and context.unbecome(). To get the actor into the same state after recov- ery you need to take special care to perform the same state transitions with become and unbecome in the receiveRecover method as you would have done in the command handler. Note that when using become from receiveRecover it will still only use the receiveRecover behavior when replaying the events. When replay is completed it will use the new behavior.

Identifiers

A persistent actor must have an identifier that doesn't change across different actor incarnations. The identifier must be defined with the persistenceId method.

`override def persistenceId = "my-stable-persistence-id"`

Recovery

By default, a persistent actor is automatically recovered on start and on restart by replaying journaled messages. **New messages sent to a persistent actor during recovery do not interfere with replayed messages.**

Accessing the **sender()** for replayed messages will always result in a deadLetters reference, as the original sender is presumed to be long gone. If you indeed have to notify an actor during recovery in the future, store its ActorPath explicitly in your persisted events.

Recovery Customization

```
override def recovery = Recovery(toSequenceNr = 457L)
```

```
`override def recovery = Recovery.none`
```

Recovery status

A persistent actor can query its own recovery status via the methods

```
def recoveryRunning: Boolean
def recoveryFinished: Boolean
```

Sometimes there is a need for performing **additional initialization** when the recovery has completed before processing any other message sent to the persistent actor.

```
override def receiveRecover: Receive = {
  case RecoveryCompleted =>
  // perform init after recovery, before any other messages
  //...
  case evt               => //...
}

override def receiveCommand: Receive = {
  case msg => //...
}
```

Relaxed local consistency requirements and high throughput use-cases

遇到需要處理大量events或滿足high throughput的use cases.

Relaxed local consistency: 低要求的consistency

The persistAsync method provides a tool for implementing high-throughput persistent actors. It will not stash incoming Commands while the Journal is still working.

```scala
class MyPersistentActor extends PersistentActor {

  override def persistenceId = "my-stable-persistence-id"

  override def receiveRecover: Receive = {
    case _ => // handle recovery here
  }

  override def receiveCommand: Receive = {
    case c: String => {
      sender() ! c
      persistAsync(s"evt-$c-1") { e => sender() ! e }
      persistAsync(s"evt-$c-2") { e => sender() ! e }
    }
  }
}

// usage
persistentActor ! "a"
persistentActor ! "b"

// possible order of received messages:
// a
// b
// evt-a-1
// evt-a-2
// evt-b-1
// evt-b-2
```

Internal stash

persistent actor會有自己的stash，這跟你做的stash是不一樣的！

The persistent actor has a **private stash** for internally caching incoming messages during *Recovery* or the persist method persisting events.

You should be careful to not send more messages to a persistent actor than it can keep up with, otherwise OutOfMemoryError will occur. You can define a maximum stash capacity in the mail configuration:

```
akka.actor.default-mailbox.stash-capacity=10000
```

Note that the stash capacity is per actor. If you have many persistent actors. The default overflow strategy is the ThrowOverflowExceptionStrategy, which discards the current received message and throws a StashOverflowException, causing actor restart if default suspervision strategy is used. You can override the **internalStashOverflowStrategy** method to return DiscardToDeadLetterStrategy or ReplyToStrategy for any "individual" persistent actor.

```
akka.persistence.internal-stash-overflow-strategy=
  "akka.persistence.ThrowExceptionConfigurator"
or
  "akka.persistence.DiscardConfigurator"
```

Failures

If persistence of an event fails, **onPersistFailure** will be invoked (logging the error by default), and **the actor will unconditionally be stopped.** *It is better to stop the actor and after a back-off timeout start it again. The akka.pattern.BackoffSupervisor actor is provided to support such restarts.*

```scala
val childProps = Props[MyPersistentActor]
val props = BackoffSupervisor.props(
  Backoff.onStop(
    childProps,
    childName = "myActor",
    minBackoff = 3.seconds,
    maxBackoff = 30.seconds,
    randomFactor = 0.2))
context.actorOf(props, name = "mySupervisor")
```

Message Deletion

Deleting messages in event sourcing based applications is typically used in conjunction with snapshotting, i.e. after a snapshot has been successfully stored, a **deleteMessages(toSequenceNr)** up until the sequence number of the data held by that snapshot can be issued to safely delete the previous events while still having access to the accumulated state during replays - by loading the snapshot. (最後一句看不懂/-)

The result of the deleteMessages request is signaled to the persistent actor with a DeleteMessagesSuccess message if the delete was successful or a DeleteMessagesFailure message if it failed.

Persistence status handling

Persisting, deleting, and replaying messages can either succeed or fail.

Method	**Success**	**Failure/Rejection**	**After failure handler invoked**
persist / persistAsync	persist handler invoked	onPersistFailure/onPersistRejected	Actor is stopped.
recovery	RecoveryCompleted	onRecoveryFailure	Actor is stopped
deleteMessages	DeleteMessagesSuccess	DeleteMessagesFailure	No automatic actions

Safely shutting down persistent actors

With normal Actors it is often acceptable to use the special PoisonPill message to signal to an Actor that it should stop itself once it receives this message – in fact this message is handled automatically by Akka, leaving the target actor no way to refuse stopping itself when given a poison pill. This can be dangerous when used with PersistentActor due to the fact that *incoming commands are stashed while the persistent actor is awaiting confirmation from the Journal that events have been written when persist() was used.* **Actor may receive and (auto)handle the PoisonPill before it processes the other messages which have been put into its stash.**

```
Consider using explicit shut-down messages instead of PoisonPill when working with persistent actors.
```

```scala
/** Explicit shutdown message */
case object Shutdown

class SafePersistentActor extends PersistentActor {
  override def persistenceId = "safe-actor"

  override def receiveCommand: Receive = {
    case c: String =>
      println(c)
      persist(s"handle-$c") { println(_) }
    case Shutdown =>
      context.stop(self)
  }

  override def receiveRecover: Receive = {
    case _ => // handle recovery here
  }
}
```

```scala
// UN-SAFE, due to PersistentActor's command stashing:
persistentActor ! "a"
persistentActor ! "b"
persistentActor ! PoisonPill
// order of received messages:
// a
//   # b arrives at mailbox, stashing;        internal-stash = [b]
// PoisonPill is an AutoReceivedMessage, is handled automatically
// !! stop !!
// Actor is stopped without handling `b` nor the `a` handler!
```

```scala
// SAFE:
persistentActor ! "a"
persistentActor ! "b"
persistentActor ! Shutdown
// order of received messages:
// a
//   # b arrives at mailbox, stashing;        internal-stash = [b]
//   # Shutdown arrives at mailbox, stashing; internal-stash = [b, Shutdown]
// handle-a
//   # unstashing;                            internal-stash = [Shutdown]
// b
// handle-b
//   # unstashing;                            internal-stash = []
// Shutdown
// -- stop --
```

Snapshots

Persistent actors can save snapshots of internal state by calling the saveSnapshot method. If saving of a snapshot succeeds, the persistent actor receives a **SaveSnapshotSuccess** message, otherwise a **SaveSnapshotFailure** message

```scala
var state: Any = _

override def receiveCommand: Receive = {
  case "snap"                             => saveSnapshot(state)
  case SaveSnapshotSuccess(metadata)         => // ...
  case SaveSnapshotFailure(metadata, reason) => // ...
}
```

During recovery, the persistent actor is offered a previously saved snapshot via a **SnapshotOffer** message from which it can initialize internal state.

```scala
var state: Any = _

override def receiveRecover: Receive = {
  case SnapshotOffer(metadata, offeredSnapshot) => state = offeredSnapshot
  case RecoveryCompleted                        =>
  case event                                    => // ...
}
```

The replayed messages that follow the SnapshotOffer message, if any, are younger than the offered snapshot. They finally recover the persistent actor to its current (i.e. latest) state. (年輕的意思比較靠近現在)

In general, a persistent actor is only offered a snapshot if that persistent actor has previously saved one or more snapshots and at least one of these snapshots matches the **SnapshotSelectionCriteria** that can be specified for recovery.

```scala
override def recovery = Recovery(fromSnapshot = SnapshotSelectionCriteria(
  maxSequenceNr = 457L,
  maxTimestamp = System.currentTimeMillis))
```

If not specified, they default to **SnapshotSelectionCriteria.Latest** which selects the latest (= youngest) snapshot.

Snapshot store

In order to use snapshots, **a default snapshot-store (akka.persistence.snapshot-store.plugin) must be configured**, or the PersistentActor can pick a snapshot store explicitly by overriding def **snapshotPluginId: String**. Note that Cluster Sharding is using snapshots, so if you use Cluster Sharding you need to define a snapshot store plugin. 用到cluster sharding就必須要提供store plugin

Snapshot deletion

A persistent actor can delete individual snapshots by calling the **deleteSnapshot** method with the sequence number of when the snapshot was taken. To bulk-delete a range of snapshots matching SnapshotSelectionCriteria, persistent actors should use the **deleteSnapshots** method.

Note **deleteSnapshot** will delete all snapshots with the same sequence number.

Method	Success	Failure message
saveSnapshot(Any)	SaveSnapshotSuccess	SaveSnapshotFailure
deleteSnapshot(Long)	DeleteSnapshotSuccess	DeleteSnapshotFailure
deleteSnapshots(SnapshotSelectionCriteria)	DeleteSnapshotsSuccess	DeleteSnapshotsFailure

At-Least-Once Delivery

利用persistence可以做到at-least-once的能力！ 送出去但還沒被confirmed的message就persistant起來

To send messages with at-least-once delivery semantics to destinations you can mix-in **AtLeastOnceDelivery trait** to your PersistentActor on the sending side. It takes care of re-sending messages when they have not been confirmed within a configurable timeout.

The state of the sending actor, including which messages have been sent that have not been confirmed by the recepient must be persistent so that it can survive a crash of the sending actor or JVM. **It is your responsibility to persist the intent that a message is sent and that a confirmation has been received.**

你必須自己做persistence, AtLeastOnceDelivery trait並不會幫你做。還要注意，在at-least-once機制下，接收端的收到messages order就不再保證一定會跟送出時相同，因為有re-send的緣故！

Relationship between deliver and confirmDelivery

送message的方式: To send messages to the destination path, use the **deliver** method after you have persisted the intent to send the message.

Confirm message的方式: The destination actor must send back a confirmation message. When the sending actor receives this confirmation message you should persist the fact that the message was delivered successfully and then call the **confirmDelivery** method.

如何做到confirm: Deliver requires a **deliveryIdToMessage** function to pass the provided deliveryId into the message so that the correlation between deliver and confirmDelivery is possible. **The deliveryId must do the round trip.**

如果你要客製化deliveryId的話：

You must then retain a **mapping** between the internal deliveryId (passed into the deliveryIdToMessage function) and your custom correlation id (passed into the message). You can do this by storing such mapping in a Map(correlationId -> deliveryId) from which you can retrieve the deliveryId to be passed into the confirmDelivery method once the receiver of your message has replied with your custom correlation id.

注意AtLeastOnceDelivery本身並不會自己保留尚未confirmed messages的狀態。

The AtLeastOnceDelivery trait has a state consisting of unconfirmed messages and a sequence number. It does not store this state itself. **You must persist events corresponding to the deliver and confirmDelivery invocations from your PersistentActor so that the state can be restored by calling the same methods during the recovery phase of the PersistentActor.** During recovery, calls to deliver will not send out messages, those will be sent later if no matching confirmDelivery will have been performed.

如果要做snapshot，必須連同AtLeastOnceDeliverySnapshot一起做成snapshot。

Support for snapshots is provided by **getDeliverySnapshot** and **setDeliverySnapshot**. The **AtLeastOnceDeliverySnapshot** contains the full delivery state, including unconfirmed messages. If you need a custom snapshot for other parts of the actor state you must also include the AtLeastOnceDeliverySnapshot.

The interval between redelivery: **redeliverInterval** (config: akka.persistence.at-least-once-delivery.redeliver-interval)

The maximum number of messages that will be sent at each redelivery burst: ** redeliveryBurstLimit** (akka.persistence.at-least-once-delivery.redelivery-burst-limit)

After a number of delivery attempts a AtLeastOnceDelivery.UnconfirmedWarning message will be sent to self. The re-sending will still continue, but you can choose to call confirmDelivery to cancel the re-sending.

最大可以保留的uncomfirmed messages:

The AtLeastOnceDelivery trait holds messages in memory until their successful delivery has been confirmed. The maximum number of unconfirmed messages that the actor is allowed to hold in memory is defined by the maxUnconfirmedMessages method.

Persistent FSM

[TBD]

Homework

1. Design a actor represents a account in a bank whihc can receive "withdraw", "deposit", "transfer" and "lookup", 4 commands. These commands will convert to events to modify internal state, "balance".
2. Consider current rate for USD and NTD, for example, user can deposit in USD, but balance in NTD so you need to compute a price in NTD first.