基本认识

Akka Persistence 使有状态的 actor 能持久化它们内部的状态，以至于在 JVM crash 后重启或者被其监管者重启之后，内部状态不会丢失，可以重新恢复。

Akka Persistence 背后关键的概念是：只有一个 actor 的状态发生改变的时候才会持久化，不会直接持久化当前的状态。

有状态的 actor 之可以被恢复，是通过重放存储的状态的改变来实现的。这个改变可以是完整的改变历史。
或者也可以从 snapshot 恢复 actor，这可以显著地减少恢复时间。

Akka Persistence 的依赖：
"com.typesafe.akka" %% "akka-persistence" % "2.5.11"

Akka Persistence 带来了一些内置的持久化插件，包括基于堆内存、基于本地文件的 snapshot 存储、基于 LevelDB。
如果使用基于 LevelDB，需要引入下面的依赖：
libraryDependencies += "org.fusesource.leveldbjni" % "leveldbjni-all" % "1.8"

架构

PersistentActor：
是一个持久化的、有状态的 actor。
它能以线程安全的方式持久化 events 到 journal。
当 persistent actor 启动或者重启时，journaled messages 被重演，以至于 persistent actor 可以恢复它内部的状态。

AtLeastOnceDelivery：
以 at-least-once 发送策略将消息发送到目的 actor。

Event Sourcing

Event Sourcing 背后的基本思想是很简单的：

> 一个 persistent actor 接收到一个 command，这个 command 首先被校验是否能应用到当前 actor 的状态。
> 如果校验成功了，events 会从 command 中生成，events 代表了 command 的影响。
> 然后这些 events 被持久化，当成功持久化之后这些 events 被用来改变 actor 的状态。

一、persist *

Akka persistence 使用 PersistentActor 特质来支持 Event Sourcing。
继承自这个特质的 actor 使用 persist 方法持久化和处理 events。
一个 PersistentActor 的行为通过 receiveRecover 和 receiveCommand 方法来定义。

persist 方法的第二个参数 event handler 只会处理成功持久化的 events。
成功持久化的 events 在 persistent actor 内部会被作为单独的消息发送到 persistent actor，这会触发 event handler 的执行。
而发送 events 的发送者和 command 的发送者是一样的，这允许 event handler 可以回应 command 的发送者。

event handler 的主要责任是：使用 event data 改变 persistent actor 的状态，并且通过 publish events 来通知其它 actor 它的状态成功改变了。

between the persist call and the execution(s) of the associated event handler，persistent actor 不会接受未来的 command。
收到的 message 被 stashed，直到 persist 方法执行完成。

> 如果一个 event 的持久化行为失败，onPersistFailure 方法会被调用(默认打印错误日志)，并且 actor 将无条件终止。
> 如果一个 event 在它被持久化之前拒绝了(例如由于序列化错误)，onPersistRejected 方法会被调用(默认打印警告日志)，并且 actor 继续正常处理下一个 message。

二、identifier

一个 persistent actor 必须有一个 identifier，identifier 通过使用 persistentId 方法定义，这个 identifier 必须是**全局唯一**的。
示例：
override def persistenceId = "my-stable-persistence-id"

三、recover *

默认情况下，persistent actor 在启动或者重启时，通过重演 journaled message 来恢复。
在恢复期间收到的新的 message 不会干扰重演的 message，新 message 会被 stashed，在 recovery 过程结束之后这些新 message 被接收。

在同一时间并发恢复的 actor 的数量不能超过系统或者后端存储的负载。如果超过了限制，想要恢复的 actor 必须等其他 actors 恢复完成。
可以使用下面的属性配置`同一时间并发恢复的 actor 的数量`：
akka.persistence.max-concurrent-recoveries = 50

注意：访问重演 message 的 sender 会返回一个 deadLetters，因为原始的 sender 被认为是很长时间之前的。如果想在恢复期间通知重演 message 的 sender，可以存储它的 ActorPath。

自定义 Recovery：

应用可以通过在 recovery 方法返回一个自定义的 Recovery 来自定义恢复是如何执行的。

1. 使用 SnapshotSelectionCriteria.None 来跳过加载 snapshot 并加载所有的 events。

```
override def recovery =
  Recovery(fromSnapshot = SnapshotSelectionCriteria.None)
```

2. 设置一个恢复的上界，使 actor 只能恢复到过去的某一个点，而不是恢复到最新的状态。

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

3. 禁用恢复

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

Recovery status

使用下面的方法查询恢复的状态：

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

有时候需要在恢复完成之后，并在处理其他接受到的消息之前，执行额外的初始化。persistent actor 将接受一个特殊的 RecoveryCompleted 消息，此消息刚刚好在恢复完成之后，处理其他消息之前被处理。

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

当 actor 启动时，如果从 journal 恢复 actor 的状态时发生了错误，onRecoveryFailure 方法被调用(默认打印错误日志)，并且 actor 被终止。针对加载 snapshot 时发生了错误的情况，处理行为和上面一样。

四、internal stash

persistent actor 有一个私有的 stash，它被用来内部缓存在 recovery 或者在 persist events 期间收到的消息。

应该小心不要发送过多的消息到一个 persistent actor，否则 stashed messages 将会没有边界的增长。

通过定义一个最大的 stash 容量是非常明智，可以避免 OOM：

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

需要注意：stash 容量是针对每一个 actor 的。

persistent actor 定义了三种策略，用来处理当 stash 容量超过限制的情况：

1. 默认的策略是：ThrowOverflowExceptionStrategy，抛弃当前接收到的消息，抛出 StashOverflowException 异常，causing actor restart if the default supervision strategy is used。
2. 你也可以覆盖 internalStashOverflowStrategy 方法，返回 DiscardToDeadLetterStrategy 或者 ReplyToStrategy。

五、原子地持久化多个 events

使用 persist 或者 persistAsync 方法可以原子地持久化一个 event。
使用 persistAll 或者 persistAllAsync 方法可以原子地持久化多个 events。

有的 journals 不支持原子地持久化多个 events，会拒绝 persistAll 命令。
这种情况，**onPersistRejected** 方法会被调用，携带着 **UnsupportedOperationException 异常**。

六、删除 message *

有时候需要删除到指定顺序号的所有 messages。deleteMessages 方法提供了这个功能。

deleteMessages 方法请求的结果会产生 DeleteMessagesSuccess 消息(如果方法执行成功)，或者 DeleteMessagesFailure 消息(如果方法执行失败)。

七、安全地 shutdown persistent actor

当从外部 shutdown persistent actor 时需要给与特别的关注。
对于普通的 actor，我们可以使用 PoisonPill 消息来终止 actor。但是这种机制不能应用到 persistent actor。
因为 PoisonPill 是 AutoReceivedMessage，如果使用 PoisonPill 消息来终止 persistent actor，persistent actor may receive and (auto)handle the PoisonPill before it processes the other messages which have been put into its stash。

shutdown persistent actor 的推荐方式：使用显示的 shut-down message，而不是 PoisonPill。

举例：

```
/** 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
  }
}
```

```
// 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!

// 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 --
```

Snapshot

当使用 actors 构建你的领域模型时，可能会注意到某些 actors 会很容易就积累相当长的 event 日志，并且会经历很长的 recovery 时间。针对这种情况，可以使用 snapshot 来减少 recovery 的时间。

一、保存 snapshot *

persistent actor 可以通过调用 saveSnapshot 方法保存 actor 内部状态的 snapshot。

> 如果 snapshot 保存成功了，persistent actor 会接受到一个 SaveSnapshotSuccess 消息；
> 否则，会接收到一个 SaveSnapshotFailure 消息。

示例：

```
override def receiveCommand: Receive = {
  case SaveSnapshotSuccess(metadata)         ⇒ // ...
  case SaveSnapshotFailure(metadata, reason) ⇒ // ...
}
```

metadata 是 SnapshotMetadata 类型：

```
final case class SnapshotMetadata(persistenceId: String, sequenceNr: Long, timestamp: Long = 0L)
```

二、从 snapshot revocery actor *

在 recovery 期间，persistent actor 通过接收到一个 SnapshotOffer 消息来接收之前保存的 snapshot，通过它可以初始化 actor 内部的状态。

示例：

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

一般来说，如果 persistent actor 之前保存了一个或者多个 snapshot，persistent actor 只会接收到一个 snapshot。
并且保存的多个 snapshot 中，至少有一个匹配 SnapshotSelectionCriteria，那被指定用来做 recovery。

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

如果没有指定 SnapshotSelectionCriteria，则默认的 SnapshotSelectionCriteria.Latest 将会选择最新的(即最年轻的)snapshot。

注意：
为了使用 snapshot，必须在配置文件中配置 akka.persistence.snapshot-store.plugin，
或者在 PersistentActor 中通过覆盖 def snapshotPluginId: String 方法来显示地选择一个 snapshot store。

三、删除 snapshot *

persistent actor 可以通过使用携带者顺序号的 deleteSnapshot 方法来删除某个独立的 snapshot。
或者可以使用 deleteSnapshots 方法删除匹配 SnapshotSelectionCriteria 的多个 snapshots。

四、snapshot status handing

保存或者删除 snapshot 可能成功也可能失败，这些信息通过状态消息发送到 persistent actor。

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

如果 failure message 遗留下来未被 actor 处理，则针对每一个收到的 failure message 会打印一个警告日志。
对于 success message，没有默认的行为，可以不用处理这类 message。

一、AtLeastOnceDelivery 特质介绍

为了使用`At-Least-Once`传输语义把消息发送到目的 actor。你可以在 PersistentActor 中混入 AtLeastOnceDelivery 特质。
这个特质保证：在配置的超时时间之内如果没有收到接收方回应的确认，则会重新发送消息。

为了保证在 Actor 崩溃或者 JVM 崩溃之后，已经发送但是还未收到确认的消息能重新发送，必须将这些状态持久化。
AtLeastOnceDelivery 特质自己不会持久化任何东西，这是你的责任去持久化这个意图：一个消息被发送，并且一个确认已经被接收。

> 使用 deliver 方法将消息发送到目的 actor。 【重要】
> 当 sending actor 收到了目的 actor 发送的确认消息，你应该持久化这个事实：消息已经被成功传输。然后调用 confirmDelivery 方法。

> 如果 persistent actor 当前不在 recovery，这个 deliver 方法将发送消息到目的 actor；
> 当 persistent actor 在 recovery 期间时，已经发送的消息将会被缓存起来直到使用 confirmDelivery 方法证实它们。
> 一旦 recovery 完成，如果还有消息未被证实，persistent actor 将会在发送其他消息之前重新发送它们。

> Deliver 需要一个 deliveryIdToMessage 函数将提供的 deliverId 传递到消息中(即发送到目的 actor 的消息中携带着 deliverId)，以至于 deliver 方法和 confirmDelivery 方法可以联系起来。
> 目的 actor 回应的确认消息中也包装着相同的 deliverId。
> 然后 sending actor 将会把这个 deliverId 作为 confirmDelivery 方法的参数调用 confirmDelivery 方法，最终完成这个 deliver 过程。
> 由 persistence 模块生成的 deliverId 是一个严格单调递增的顺序号。

示例：

```
case class Msg(deliveryId: Long, s: String)
case class Confirm(deliveryId: Long)

sealed trait Evt
case class MsgSent(s: String) extends Evt
case class MsgConfirmed(deliveryId: Long) extends Evt

class MyPersistentActor(destination: ActorSelection) extends PersistentActor with AtLeastOnceDelivery {
  override def persistenceId: String = "persistence-id"

  override def receiveCommand: Receive = {
    case s: String           ⇒ persist(MsgSent(s))(updateState)
    case Confirm(deliveryId) ⇒ persist(MsgConfirmed(deliveryId))(updateState)
  }

  override def receiveRecover: Receive = {
    case evt: Evt ⇒ updateState(evt)
  }

  def updateState(evt: Evt): Unit = evt match {
    case MsgSent(s) ⇒ deliver(destination)(deliveryId ⇒ Msg(deliveryId, s))
    case MsgConfirmed(deliveryId) ⇒ confirmDelivery(deliveryId)
  }
}

class MyDestination extends Actor {
  def receive = {
```

```
      case Msg(deliveryId, s) ⇒
        // ...
        sender() ! Confirm(deliveryId)
    }
}
```

二、AtLeastOnceDelivery 特质的配置

每次重新传输之间的间隔可以通过 akka.persistence.at-least-once-delivery.redeliver-interval 进行配置。
或者通过覆盖 redeliverInterval 方法来自定义。

如果想限制每次重新传输的消息最大数量，比如如果有非常多的未证实的 message(比如目的 actor 很长时间不可用)，这种限制机制可以避免每次重传时产生大量的消息造成系统压力。
可以通过 redeliveryBurstLimit 方法自定义每次重传的消息最大数量。
或者通过 akka.persistence.at-least-once-delivery.redelivery-burst-limit 属性进行配置。

我们可以控制经过多少次 re-sending 之后，self 收到 AtLeastOnceDelivery.UnconfirmedWarning 消息。在收到这个 warn 消息之后，re-sending 仍会继续，但是我们可以通过调用 confirmDelivery 方法**取消 re-sending**。
在收到 warn 消息之前 re-sending 的次数可以通过 warnAfterNumberOfUnconfirmedAttempts 方法进行定义；
或者可以通过 akka.persistence.at-least-once-delivery.warn-after-number-of-unconfirmed-attempts 属性进行配置。

AtLeastOnceDelivery 特质将 messages 保存在内存中，直到它们成功的 delivery 得到了 confirm。
persistent actor 允许在内存中保存的 unconfirmed messages 的数量通过 maxUnconfirmedMessages 方法进行定义。
或者通过 akka.persistence.at-least-once-delivery.max-unconfirmed-messages 属性进行配置。
如果内存中 unconfirmed message 的数量超过了限制，deliver 方法将不会接受更多的 messages，并且它会抛出 AtLeastOnceDelivery.MaxUnconfirmedMessagesExceededException 异常。

一、store plugin 介绍

Akka Persistence 后端使用的存储插件是可插拔的。
在 https://akka.io/community/?_ga=2.133275661.2085462024.1519903771-1924259398.1501156402 可以看到社区中已经可用的存储插件。

默认情况下，存储插件会应用到所有的 persistent actors，不过也可以单独为 persistent actor 配置存储插件。例如：

```
trait ActorWithOverridePlugins extends PersistentActor {
  override def persistenceId = "123"
  override def journalPluginId = "akka.persistence.chronicle.journal"
  override def snapshotPluginId = "akka.persistence.chronicle.snapshot-store"
}
```

如果 persistent actor 没有覆盖 journalPluginId 和 snapshotPluginId 方法，则 akka persistence extence 将会使用 application.conf 文件里配置的存储插件：

```
akka.persistence.journal.plugin = ""
akka.persistence.snapshot-store.plugin = ""
```

默认情况下，存储插件是根据需要按需启动的。有些场景中预先把存储插件启动好是比较有益的。
为了做到这样，在 akka.extensions 配置下面添加 akka.persistence.Persistence。然后使用 akka.persistence.journal.auto-start-journals 和 akka.persistence.snapshot-store.auto-start-snapshot-stores 指定你希望启动的插件的 ids。
示例：

```
akka {
  extensions = [akka.persistence.Persistence]
  persistence {
    journal {
      plugin = "akka.persistence.journal.leveldb"
      auto-start-journals = ["akka.persistence.journal.leveldb"]
    }
    snapshot-store {
      plugin = "akka.persistence.snapshot-store.local"
      auto-start-snapshot-stores = ["akka.persistence.snapshot-store.local"]
    }
  }
}
```

二、Local LevelDB journal plugin

LevelDB journal plugin 的配置实体是 akka.persistence.journal.leveldb。它写 message 到本地的 LevelDB 中。
使用下面的配置开启这个 plugin：

```
akka.persistence.journal.plugin = "akka.persistence.journal.leveldb"
```

基于 LevelDB 的 plugin 需要下面额外的依赖：

```
"org.fusesource.leveldbjni"    % "leveldbjni-all"    % "1.8"
```

LevelDB 默认的存储路径是：当前工作目录下以`journal`命名的目录。

存储路径可以通过下面的配置进行修改：akka.persistence.journal.leveldb.dir = "target/journal"

使用 LevelDB journal plugin 有一个特点：删除操作不会把 message 从 journal 中删除，而是针对每一个要删除的 message 添加一个 "tombstone"。
因此，就会发现一个问题，需要我们自己处理不断增长的 journal 的容量。为了处理这种问题，LevelDB 提供了一个特殊的 journal 压缩功能。
可以通过下面的配置进行配置：

```
# Number of deleted messages per persistence id that will trigger journal compaction
akka.persistence.journal.leveldb.compaction-intervals {
    persistence-id-1 = 100
    persistence-id-2 = 200
    # ...
    persistence-id-N = 1000
    # use wildcards to match unspecified persistence ids, if any
    "*" = 250
}
```

三、Local snapshot store

Local snapshot plugin 的配置实体是 akka.persistence.snapshot-store.local。它写 snapshot 到本地文件系统中。
使用下面的配置开启这个 plugin：

```
akka.persistence.snapshot-store.plugin = "akka.persistence.snapshot-store.local"
```

默认的存储路径是：当前工作目录下以`snapshots`命名的目录。
存储路径可以通过下面的配置进行修改：akka.persistence.snapshot-store.local.dir = "target/snapshots"

自定义序列化

snapshot 和 persistent messages 的序列化使用 Akka 的 Serialization 设施进行配置。
(详细内容看`Akka-Actor`笔记`序列化`一节的内容)