ReceiveActor API

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

Creating Actors

**NOTE**

Since Akka.NET enforces parental supervision every actor is supervised and (potentially) the supervisor of its children, it is advisable that you familiarize yourself with [Actor Systems](https://getakka.net/articles/concepts/actor-systems.html) and [Supervision and Monitoring](https://getakka.net/articles/concepts/supervision.html) and it may also help to read [Actor References, Paths and Addresses](https://getakka.net/articles/concepts/addressing.html).

Defining an Actor class

In order to use the Receive() method inside an actor, the actor must inherit from ReceiveActor. Inside the constructor, add a call to Receive<T>(Action<T> handler) for every type of message you want to handle:

Here is an example:

public class MyActor: ReceiveActor

{

private readonly ILoggingAdapter log = Context.GetLogger();

public MyActor()

{

Receive<string>(message => {

log.Info("Received String message: {0}", message);

Sender.Tell(message);

});

Receive<SomeMessage>(message => {...});

}

}

Props

Props is a configuration class to specify options for the creation of actors, think of it as an immutable and thus freely shareable recipe for creating an actor including associated deployment information (e.g. which dispatcher to use, see more below). Here are some examples of how to create a Props instance

Props props1 = Props.Create(typeof(MyActor));

Props props2 = Props.Create(() => new MyActorWithArgs("arg"));

Props props3 = Props.Create<MyActor>();

Props props4 = Props.Create(typeof(MyActorWithArgs), "arg");

The second variant shows how to pass constructor arguments to the Actor being created, but it should only be used outside of actors as explained below.

Recommended Practices

It is a good idea to provide static factory methods on the ReceiveActor which help keeping the creation of suitable Props as close to the actor definition as possible.

public class DemoActor : ReceiveActor

{

private readonly int \_magicNumber;

public DemoActor(int magicNumber)

{

\_magicNumber = magicNumber;

Receive<int>(x =>

{

Sender.Tell(x + \_magicNumber);

});

}

public static Props Props(int magicNumber)

{

return Akka.Actor.Props.Create(() => new DemoActor(magicNumber));

}

}

system.ActorOf(DemoActor.Props(42), "demo");

Another good practice is to declare local messages (messages that are sent in process) within the Actor, which makes it easier to know what messages are generally being sent over the wire vs in process.:

public class DemoActor : UntypedActor

{

protected override void OnReceive(object message)

{

switch (message)

{

case DemoActorLocalMessages.DemoActorLocalMessage1 msg1:

// Handle message here...

break;

case DemoActorLocalMessages.DemoActorLocalMessage2 msg2:

// Handle message here...

break;

default:

break;

}

}

class DemoActorLocalMessages

{

public class DemoActorLocalMessage1

{

}

public class DemoActorLocalMessage2

{

}

}

}

Creating Actors with Props

Actors are created by passing a Props instance into the ActorOf factory method which is available on ActorSystem and ActorContext.

// ActorSystem is a heavy object: create only one per application

ActorSystem system = ActorSystem.Create("MySystem");

IActorRef myActor = system.ActorOf<MyActor>("myactor");

Using the ActorSystem will create top-level actors, supervised by the actor system's provided guardian actor, while using an actor's context will create a child actor.

public class FirstActor : ReceiveActor

{

IActorRef child = Context.ActorOf<MyActor>("myChild");

// plus some behavior ...

}

It is recommended to create a hierarchy of children, grand-children and so on such that it fits the logical failure-handling structure of the application, see [Actor Systems](https://getakka.net/articles/concepts/actor-systems.html).

The call to ActorOf returns an instance of IActorRef. This is a handle to the actor instance and the only way to interact with it. The IActorRef is immutable and has a one to one relationship with the Actor it represents. The IActorRef is also serializable and network-aware. This means that you can serialize it, send it over the wire and use it on a remote host and it will still be representing the same Actor on the original node, across the network.

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with $$, but it may contain URL encoded characters (eg. %20 for a blank space). If the given name is already in use by another child to the same parent an InvalidActorNameException is thrown.

Actors are automatically started asynchronously when created.

Actor API

If the current actor behavior does not match a received message, it's recommended that you call the unhandled method, which by default publishes a new Akka.Actor.UnhandledMessage(message, sender, recipient) on the actor system's event stream (set configuration item Unhandled to on to have them converted into actual Debugmessages).

In addition, it offers:

* Self reference to the IActorRef of the actor
* Sender reference sender Actor of the last received message, typically used as described in [Reply to messages](https://getakka.net/articles/actors/receive-actor-api.html#reply-to-messages).
* SupervisorStrategy user overridable definition the strategy to use for supervising child actors

This strategy is typically declared inside the actor in order to have access to the actor's internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the Sender reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).

* Context exposes contextual information for the actor and the current message, such as:
  + factory methods to create child actors (ActorOf)
  + system that the actor belongs to
  + parent supervisor
  + supervised children
  + lifecycle monitoring
  + hotswap behavior stack as described in [HotSwap](https://getakka.net/articles/actors/receive-actor-api.html#becomeunbecome)

The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

public override void PreStart()

{

}

protected override void PreRestart(Exception reason, object message)

{

foreach (IActorRef each in Context.GetChildren())

{

Context.Unwatch(each);

Context.Stop(each);

}

PostStop();

}

protected override void PostRestart(Exception reason)

{

PreStart();

}

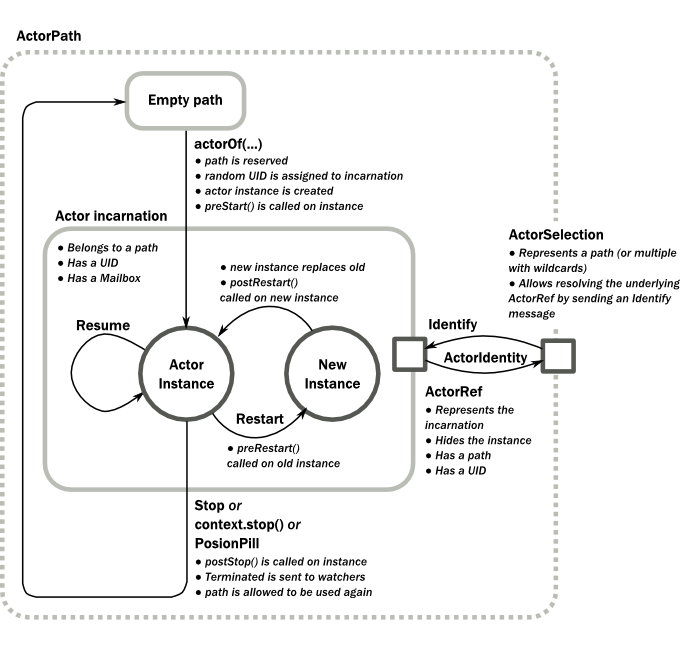
protected override void PostStop()

{

}

The implementations shown above are the defaults provided by the ReceiveActor class.

Actor Lifecycle



A path in an actor system represents a "place" which might be occupied by a living actor. Initially (apart from system initialized actors) a path is empty. When ActorOf() is called it assigns an incarnation of the actor described by the passed Props to the given path. An actor incarnation is identified by the path and a UID. A restart only swaps the Actor instance defined by the Props but the incarnation and hence the UID remains the same.

The lifecycle of an incarnation ends when the actor is stopped. At that point the appropriate lifecycle events are called and watching actors are notified of the termination. After the incarnation is stopped, the path can be reused again by creating an actor with ActorOf(). In this case the name of the new incarnation will be the same as the previous one but the UIDs will differ.

An IActorRef always represents an incarnation (path and UID) not just a given path. Therefore if an actor is stopped and a new one with the same name is created an IActorRef of the old incarnation will not point to the new one.

ActorSelection on the other hand points to the path (or multiple paths if wildcards are used) and is completely oblivious to which incarnation is currently occupying it. ActorSelection cannot be watched for this reason. It is possible to resolve the current incarnation's ActorRef living under the path by sending an Identify message to the ActorSelection which will be replied to with an ActorIdentity containing the correct reference (see [Identifying Actors via Actor Selection](https://getakka.net/articles/actors/receive-actor-api.html#identifying-actors-via-actor-selection)). This can also be done with the resolveOne method of the ActorSelection, which returns a Task of the matching IActorRef.

Lifecycle Monitoring aka DeathWatch

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the Terminated message dispatched by the other actor upon termination (see [Stopping Actors](https://getakka.net/articles/actors/receive-actor-api.html#stopping-actors)). This service is provided by the DeathWatch component of the actor system.

Registering a monitor is easy (see fourth line, the rest is for demonstrating the whole functionality):

public class WatchActor : ReceiveActor

{

private IActorRef child = Context.ActorOf(Props.Empty, "child");

private IActorRef lastSender = Context.System.DeadLetters;

public WatchActor()

{

Context.Watch(child); // <-- this is the only call needed for registration

Receive<string>(s => s.Equals("kill"), msg =>

{

Context.Stop(child);

lastSender = Sender;

});

Receive<Terminated>(t => t.ActorRef.Equals(child), msg =>

{

lastSender.Tell("finished");

});

}

}

It should be noted that the Terminated message is generated independent of the order in which registration and termination occur. In particular, the watching actor will receive a Terminated message even if the watched actor has already been terminated at the time of registration.

Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the Terminated message.

It is also possible to deregister from watching another actor's liveliness using Context.Unwatch(target). This works even if the Terminated message has already been enqueued in the mailbox; after calling unwatch no Terminated message for that actor will be processed anymore.

Start Hook

Right after starting the actor, its PreStart method is invoked.

protected override void PreStart()

{

child = Context.ActorOf(Props.Empty);

}

This method is called when the actor is first created. During restarts it is called by the default implementation of PostRestart, which means that by overriding that method you can choose whether the initialization code in this method is called only exactly once for this actor or for every restart. Initialization code which is part of the actor's constructor will always be called when an instance of the actor class is created, which happens at every restart.

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see [Supervision and Monitoring](https://getakka.net/articles/concepts/supervision.html). This restart involves the hooks mentioned above:

* The old actor is informed by calling PreRestart with the exception which caused the restart and the message which triggered that exception; the latter may be None if the restart was not caused by processing a message, e.g. when a supervisor does not trap the exception and is restarted in turn by its supervisor, or if an actor is restarted due to a sibling's failure. If the message is available, then that message's sender is also accessible in the usual way (i.e. by calling the Sender property). This method is the best place for cleaning up, preparing hand-over to the fresh actor instance, etc. By default it stops all children and calls PostStop.
* The initial factory from the ActorOf call is used to produce the fresh instance.
* The new actor's PostRestart method is invoked with the exception which caused the restart. By default the PreStart is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the PostRestart hook returns. The message that triggered the exception will not be received again. Any message sent to an actor while it is being restarted will be queued to its mailbox as usual.

**WARNING**

Be aware that the ordering of failure notifications relative to user messages is not deterministic. In particular, a parent might restart its child before it has processed the last messages sent by the child before the failure. See Discussion: [Message Ordering for details](https://getakka.net/articles/concepts/message-delivery-reliability.html#discussion-message-ordering).

Stop Hook

After stopping an actor, its PostStop hook is called, which may be used e.g. for deregistering this actor from other services. This hook is guaranteed to run after message queuing has been disabled for this actor, i.e. messages sent to a stopped actor will be redirected to the DeadLetters of the ActorSystem.

Identifying Actors via Actor Selection

As described in Actor References, Paths and Addresses, each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an ActorSelection with the result:

// will look up this absolute path

Context.ActorSelection("/user/serviceA/actor");

// will look up sibling beneath same supervisor

Context.ActorSelection("../joe");

**NOTE**

It is always preferable to communicate with other Actors using their IActorRef instead of relying upon ActorSelection. Exceptions are: sending messages using the At-Least-Once Delivery facility, initiating first contact with a remote system. In all other cases ActorRefs can be provided during Actor creation or initialization, passing them from parent to child or introducing Actors by sending their ActorRefs to other Actors within messages.

The supplied path is parsed as a System.URI, which basically means that it is split on / into path elements. If the path starts with /, it is absolute and the look-up starts at the root guardian (which is the parent of "/user"); otherwise it starts at the current actor. If a path element equals .., the look-up will take a step "up" towards the supervisor of the currently traversed actor, otherwise it will step "down" to the named child. It should be noted that the .. in actor paths here always means the logical structure, i.e. the supervisor.

The path elements of an actor selection may contain wildcard patterns allowing for broadcasting of messages to that section:

// will look all children to serviceB with names starting with worker

Context.ActorSelection("/user/serviceB/worker\*");

// will look up all siblings beneath same supervisor

Context.ActorSelection("../\*");

Messages can be sent via the ActorSelection and the path of the ActorSelectionis looked up when delivering each message. If the selection does not match any actors the message will be dropped.

To acquire an IActorRef for an ActorSelection you need to send a message to the selection and use the Sender reference of the reply from the actor. There is a built-in Identify message that all Actors will understand and automatically reply to with a ActorIdentity message containing the IActorRef. This message is handled specially by the actors which are traversed in the sense that if a concrete name lookup fails (i.e. a non-wildcard path element does not correspond to a live actor) then a negative result is generated. Please note that this does not mean that delivery of that reply is guaranteed, it still is a normal message.

public class Follower : ReceiveActor

{

private readonly IActorRef \_probe;

private string identifyId = "1";

private IActorRef \_another;

public Follower(IActorRef probe)

{

\_probe = probe;

var selection = Context.ActorSelection("/user/another");

selection.Tell(new Identify(identifyId), Self);

Receive<ActorIdentity>(identity =>

{

if (identity.MessageId.Equals(identifyId))

{

var subject = identity.Subject;

if (subject == null)

{

Context.Stop(Self);

}

else

{

\_another = subject;

Context.Watch(\_another);

\_probe.Tell(subject, Self);

}

}

});

Receive<Terminated>(t =>

{

if (t.ActorRef.Equals(\_another))

{

Context.Stop(Self);

}

});

}

}

You can also acquire an IActorRef for an ActorSelection with the ResolveOne method of the ActorSelection. It returns a Task of the matching IActorRef if such an actor exists. It is completed with failure akka.actor.ActorNotFound if no such actor exists or the identification didn't complete within the supplied timeout.

Remote actor addresses may also be looked up, if *remoting* is enabled:

Context.ActorSelection("akka.tcp://app@otherhost:1234/user/serviceB");

Messages and Immutability

**IMPORTANT**

Messages can be any kind of object but have to be immutable. Akka can’t enforce immutability (yet) so this has to be by convention.

Here is an example of an immutable message:

public class ImmutableMessage

{

public ImmutableMessage(int sequenceNumber, List<string> values)

{

SequenceNumber = sequenceNumber;

Values = values.AsReadOnly();

}

public int SequenceNumber { get; }

public IReadOnlyCollection<string> Values { get; }

}

Send messages

Messages are sent to an Actor through one of the following methods.

* Tell() means fire-and-forget, e.g. send a message asynchronously and return immediately.
* Ask() sends a message asynchronously and returns a Future representing a possible reply.

Message ordering is guaranteed on a per-sender basis.

**NOTE**

There are performance implications of using Ask since something needs to keep track of when it times out, there needs to be something that bridges a Task into an IActorRef and it also needs to be reachable through remoting. So always prefer Tell for performance, and only Ask if you must.

In all these methods you have the option of passing along your own IActorRef. Make it a practice of doing so because it will allow the receiver actors to be able to respond to your message, since the Sender reference is sent along with the message.

Tell: Fire-forget

This is the preferred way of sending messages. No blocking waiting for a message. This gives the best concurrency and scalability characteristics.

// don’t forget to think about who is the sender (2nd argument)

target.Tell(message, Self);

The sender reference is passed along with the message and available within the receiving actor via its Senderproperty while processing this message. Inside of an actor it is usually Self who shall be the sender, but there can be cases where replies shall be routed to some other actor—e.g. the parent—in which the second argument to Tell would be a different one. Outside of an actor and if no reply is needed the second argument can be null; if a reply is needed outside of an actor you can use the ask-pattern described next.

Ask: Send-And-Receive-Future

The ask pattern involves actors as well as Tasks, hence it is offered as a use pattern rather than a method on ActorRef:

var tasks = new List<Task>();

tasks.Add(actorA.Ask("request", TimeSpan.FromSeconds(1)));

tasks.Add(actorB.Ask("another request", TimeSpan.FromSeconds(5)));

Task.WhenAll(tasks).PipeTo(actorC, Self);

This example demonstrates Ask together with the Pipe Pattern on tasks, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: Ask produces a Task, two of which are awaited until both are completed, and when that happens, a new Result object is forwarded to another actor.

Using Ask will send a message to the receiving Actor as with Tell, and the receiving actor must reply with Sender.Tell(reply, Self) in order to complete the returned Task with a value. The Ask operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

**WARNING**

To complete the Task with an exception you need send a Failure message to the sender. This is not done automatically when an actor throws an exception while processing a message.

try

{

var result = operation();

Sender.Tell(result, Self);

}

catch (Exception e)

{

Sender.Tell(new Failure { Exception = e }, Self);

}

If the actor does not complete the task, it will expire after the timeout period, specified as parameter to the Askmethod, and the Task will be cancelled and throw a TaskCancelledException.

For more information on Tasks, check out the [MSDN documentation](https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx).

**WARNING**

When using task callbacks inside actors, you need to carefully avoid closing over the containing actor’s reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time.

Forward message

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a 'mediator'. This can be useful when writing actors that work as routers, load-balancers, replicators etc. You need to pass along your context variable as well.

target.Forward(result, Context);

Receive messages

To receive a message you should create a Receive handler in a constructor.

Receive<string>(ms => Console.WriteLine("Received message: " + msg));

Handler priority

If more than one handler matches, the one that appears first is used while the others are ignored.

Receive<string>(s => Console.WriteLine("Received string: " + s)); //1

Receive<string>(s => Console.WriteLine("Also received string: " + s)); //2

Receive<object>(o => Console.WriteLine("Received object: " + o)); //3

**Example**  
The actor receives a message of type string. Only the first handler is invoked, even though all three handlers can handle that message.

Using predicates

By specifying a predicate, you can choose which messages to handle.

Receive<string>(s => s.Length > 5, s => Console.WriteLine("Received string: " + s);

The handler above will only be invoked if the length of the string is greater than 5.

If the predicate do not match, the next matching handler will be used.

Receive<string>(s => s.Length > 5, s => Console.WriteLine("1: " + s)); //1

Receive<string>(s => s.Length > 2, s => Console.WriteLine("2: " + s)); //2

Receive<string>(s => Console.WriteLine("3: " + s)); //3

**Example**  
The actor receives the message "123456". Since the length of is 6, the predicate specified for the first handler will return true, and the first handler will be invoked resulting in "1: 123456" being written to the console.

**NOTE**

Note that even though the predicate for the second handler matches, and that the third handler matches all messages of type string only the first handler is invoked.

**Example**  
If the actor receives the message "1234", then "2: 1234" will be written to the console.

**Example**  
If the actor receives the message "12", then "3: 12" will be written on the console.

Predicates can be specified before the action handler or after. These two declarations are equivalent:

Receive<string>(s => s.Length > 5, s => Console.WriteLine("Received string: " + s));

Receive<string>(s => Console.WriteLine("Received string: " + s), s => s.Length > 5);

Unmatched messages

If the actor receives a message for which no handler matches, the unhandled message is published to the EventStream wrapped in an UnhandledMessage. To change this behavior override Unhandled(object message)

protected override void Unhandled(object message)

{

//Do something with the message.

}

Another option is to add a handler last that matches all messages, using ReceiveAny().

ReceiveAny

To catch messages of any type the ReceiveAny(Action<object> handler) overload can be specified.

Receive<string>(s => Console.WriteLine("Received string: " + s);

ReceiveAny(o => Console.WriteLine("Received object: " + o);

Since it handles everything, it must be specified last. Specifying handlers it after will cause an exception.

ReceiveAny(o => Console.WriteLine("Received object: " + o);

Receive<string>(s => Console.WriteLine("Received string: " + s); //This will cause an exception

**NOTE**

Note that Receive<object>(Action<object> handler) behaves the same as ReceiveAny() as it catches all messages. These two are equivalent:

ReceiveAny(o => Console.WriteLine("Received object: " + o);

Receive<object>(0 => Console.WriteLine("Received object: " + o);

Non generic overloads

Receive has non generic overloads:

Receive(typeof(string), obj => Console.WriteLine(obj.ToString()) );

Predicates can go before or after the handler:

Receive(typeof(string), obj => ((string) obj).Length > 5, obj => Console.WriteLine(obj.ToString()) );

Receive(typeof(string), obj => Console.WriteLine(obj.ToString()), obj => ((string) obj).Length > 5 );

And the non generic Func

Receive(typeof(string), obj =>

{

var s = (string)obj;

if (s.Length > 5)

{

Console.WriteLine("1: " + s);

return true;

}

return false;

});

Reply to messages

If you want to have a handle for replying to a message, you can use Sender, which gives you an IActorRef. You can reply by sending to that IActorRef with Sender.Tell(replyMsg, Self). You can also store the IActorReffor replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or task context) then the sender defaults to a 'dead-letter' actor ref.

Receive<string>(() =>

{

var result = calculateResult();

// do not forget the second argument!

Sender.Tell(result, Self);

})

Receive timeout

The IActorContext SetReceiveTimeout defines the inactivity timeout after which the sending of a ReceiveTimeout message is triggered. When specified, the receive function should be able to handle an Akka.Actor.ReceiveTimeout message.

**NOTE**

Please note that the receive timeout might fire and enqueue the ReceiveTimeout message right after another message was enqueued; hence it is not guaranteed that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass in nullto SetReceiveTimeout to switch off this feature.

public class MyActor : ReceiveActor

{

private ILoggingAdapter log = Context.GetLogger();

public MyActor()

{

Receive<string>(s => s.Equals("Hello"), msg =>

{

Context.SetReceiveTimeout(TimeSpan.FromMilliseconds(100));

});

Receive<ReceiveTimeout>(msg =>

{

Context.SetReceiveTimeout(null);

throw new Exception("Receive timed out");

return;

});

}

}

Stopping actors

Actors are stopped by invoking the Stop method of a ActorRefFactory, i.e. ActorContext or ActorSystem. Typically the context is used for stopping child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. stop may return before the actor is stopped.

public class MyStoppingActor : ReceiveActor

{

private IActorRef child;

public MyStoppingActor()

{

Receive<string>(s => s.Equals("interrupt-child"), msg =>

{

Context.Stop(child);

});

Receive<string>(s => s.Equals("done"), msg =>

{

Context.Stop(Self);

});

}

}

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the DeadLetters of the ActorSystem, but that depends on the mailbox implementation.

Termination of an actor proceeds in two steps: first the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the internal termination notifications from its children until the last one is gone, finally terminating itself (invoking PostStop, dumping mailbox, publishing Terminated on the DeathWatch, telling its supervisor). This procedure ensures that actor system sub-trees terminate in an orderly fashion, propagating the stop command to the leaves and collecting their confirmation back to the stopped supervisor. If one of the actors does not respond (i.e. processing a message for extended periods of time and therefore not receiving the stop command), this whole process will be stuck.

Upon ActorSystem.Terminate, the system guardian actors will be stopped, and the aforementioned process will ensure proper termination of the whole system.

The PostStop hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

protected override void PostStop()

{

// clean up resources here ...

}

**NOTE**

Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an InvalidActorNameException. Instead, watch the terminating actor and create its replacement in response to the Terminated message which will eventually arrive.

PoisonPill

You can also send an actor the Akka.Actor.PoisonPill message, which will stop the actor when the message is processed. PoisonPill is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Use it like this:

myActor.Tell(PoisonPill.Instance, Sender);

Graceful Stop

GracefulStop is useful if you need to wait for termination or compose ordered termination of several actors:

var manager = system.ActorOf<Manager>();

try

{

await manager.GracefulStop(TimeSpan.FromMilliseconds(5), "shutdown");

// the actor has been stopped

}

catch (TaskCanceledException)

{

// the actor wasn't stopped within 5 seconds

}

...

public class Manager : ReceiveActor

{

private IActorRef worker = Context.Watch(Context.ActorOf<Cruncher>("worker"));

public Manager()

{

Receive<string>(s => s.Equals("job"), msg =>

{

worker.Tell("crunch");

});

Receive<Shutdown>(\_ =>

{

worker.Tell(PoisonPill.Instance, Self);

Context.Become(ShuttingDown);

});

}

private void ShuttingDown(object message)

{

Receive<string>(s => s.Equals("job"), msg =>

{

Sender.Tell("service unavailable, shutting down", Self);

});

Receive<Shutdown>(\_ =>

{

Context.Stop(Self);

});

}

}

When GracefulStop() returns successfully, the actor’s PostStop() hook will have been executed: there exists a happens-before edge between the end of PostStop() and the return of GracefulStop().

In the above example a "shutdown" message is sent to the target actor to initiate the process of stopping the actor. You can use PoisonPill for this, but then you have limited possibilities to perform interactions with other actors before stopping the target actor. Simple cleanup tasks can be handled in PostStop.

**WARNING**

Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use after GracefulStop() returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a Terminated message, i.e. not for top-level actors.

Become/Unbecome

Upgrade

Akka supports hotswapping the Actor’s message loop (e.g. its implementation) at runtime. Use the Context.Become method from within the Actor. The hotswapped code is kept in a Stack which can be pushed (replacing or adding at the top) and popped.

**WARNING**

Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor using Context.Become:

public class HotSwapActor : ReceiveActor

{

public HotSwapActor()

{

Receive<string>(s => s.Equals("foo"), msg =>

{

Become(Angry);

});

Receive<string>(s => s.Equals("bar"), msg =>

{

Become(Happy);

});

}

private void Angry(object message)

{

Receive<string>(s => s.Equals("foo"), msg =>

{

Sender.Tell("I am already angry?");

});

Receive<string>(s => s.Equals("bar"), msg =>

{

Become(Happy);

});

}

private void Happy(object message)

{

Receive<string>(s => s.Equals("foo"), msg =>

{

Sender.Tell("I am already happy :-)");

});

Receive<string>(s => s.Equals("bar"), msg =>

{

Become(Angry);

});

}

}

This variant of the Become method is useful for many different things, such as to implement a Finite State Machine (FSM). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use Unbecome, instead always the next behavior is explicitly installed.

The other way of using Become does not replace but add to the top of the behavior stack. In this case care must be taken to ensure that the number of "pop" operations (i.e. Unbecome) matches the number of "push" ones in the long run, otherwise this amounts to a memory leak (which is why this behavior is not the default).

public class Swapper : ReceiveActor

{

public class Swap

{

public static Swap Instance = new Swap();

private Swap() { }

}

private ILoggingAdapter log = Context.GetLogger();

public Swapper()

{

Receive<Swap>(swap1 =>

{

log.Info("Hi");

BecomeStacked(() =>

{

Receive<Swap>(swap2 =>

{

log.Info("Ho");

UnbecomeStacked();

});

});

});

}

}

...

static void Main(string[] args)

{

var system = ActorSystem.Create("MySystem");

var swapper = system.ActorOf<Swapper>();

swapper.Tell(Swapper.SWAP);

swapper.Tell(Swapper.SWAP);

swapper.Tell(Swapper.SWAP);

swapper.Tell(Swapper.SWAP);

swapper.Tell(Swapper.SWAP);

swapper.Tell(Swapper.SWAP);

Console.ReadLine();

}

Stash

The IWithUnboundedStash interface enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking Context.BecomeStacked() or Context.UnbecomeStacked();, all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally. An actor that implements IWithUnboundedStash will automatically get a deque-based mailbox.

Here is an example of the IWithUnboundedStash interface in action:

public class ActorWithProtocol : ReceiveActor, IWithUnboundedStash

{

public IStash Stash { get; set; }

public ActorWithProtocol()

{

Receive<string>(s => s.Equals("open"), open =>

{

Stash.UnstashAll();

BecomeStacked(() =>

{

Receive<string>(s => s.Equals("write"), write =>

{

// do writing...

});

Receive<string>(s => s.Equals("close"), write =>

{

Stash.UnstashAll();

Context.UnbecomeStacked();

});

ReceiveAny(\_ => Stash.Stash());

});

});

ReceiveAny(\_ => Stash.Stash());

}

}

Invoking Stash() adds the current message (the message that the actor received last) to the actor's stash. It is typically invoked when handling the default case in the actor's message handler to stash messages that aren't handled by the other cases. It is illegal to stash the same message twice; to do so results in an IllegalStateException being thrown. The stash may also be bounded in which case invoking Stash() may lead to a capacity violation, which results in a StashOverflowException. The capacity of the stash can be configured using the stash-capacity setting (an Int) of the mailbox's configuration.

Invoking UnstashAll() enqueues messages from the stash to the actor's mailbox until the capacity of the mailbox (if any) has been reached (note that messages from the stash are prepended to the mailbox). In case a bounded mailbox overflows, a MessageQueueAppendFailedException is thrown. The stash is guaranteed to be empty after calling UnstashAll().

Note that the stash is part of the ephemeral actor state, unlike the mailbox. Therefore, it should be managed like other parts of the actor's state which have the same property. The IWithUnboundedStash interface implementation of PreRestart will call UnstashAll(), which is usually the desired behavior.

Killing an Actor

You can kill an actor by sending a Kill message. This will cause the actor to throw a ActorKilledException, triggering a failure. The actor will suspend operation and its supervisor will be asked how to handle the failure, which may mean resuming the actor, restarting it or terminating it completely. See [What Supervision Means](https://getakka.net/articles/concepts/supervision.html#what-supervision-means) for more information.

Use Kill like this:

// kill the 'victim' actor

victim.Tell(Akka.Actor.Kill.Instance, ActorRef.NoSender);

Actors and exceptions

It can happen that while a message is being processed by an actor, that some kind of exception is thrown, e.g. a database exception.

What happens to the Message

If an exception is thrown while a message is being processed (i.e. taken out of its mailbox and handed over to the current behavior), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don't want a system to livelock (so consuming a lot of cpu cycles without making progress).

What happens to the mailbox

If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox will be there as well.

What happens to the actor

If code within an actor throws an exception, that actor is suspended and the supervision process is started (see Supervision and Monitoring). Depending on the supervisor’s decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated.

Initialization patterns

The rich lifecycle hooks of Actors provide a useful toolkit to implement various initialization patterns. During the lifetime of an IActorRef, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the IActorRef.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the IActorRef is created. The following sections provide patterns for different initialization needs.

Initialization via constructor

Using the constructor for initialization has various benefits. First of all, it makes it possible to use readonly fields to store any state that does not change during the life of the actor instance, making the implementation of the actor more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

Initialization via PreStart

The method PreStart() of an actor is only called once directly during the initialization of the first instance, that is, at creation of its ActorRef. In the case of restarts, PreStart() is called from PostRestart(), therefore if not overridden, PreStart() is called on every incarnation. However, overriding PostRestart() one can disable this behavior, and ensure that there is only one call to PreStart().

One useful usage of this pattern is to disable creation of new ActorRefs for children during restarts. This can be achieved by overriding PreRestart():

protected override void PreStart()

{

// Initialize children here

}

// Overriding postRestart to disable the call to preStart() after restarts

protected override void PostRestart(Exception reason)

{

}

// The default implementation of PreRestart() stops all the children

// of the actor. To opt-out from stopping the children, we

// have to override PreRestart()

protected override void PreRestart(Exception reason, object message)

{

// Keep the call to PostStop(), but no stopping of children

PostStop();

}

Please note, that the child actors are *still restarted*, but no new IActorRef is created. One can recursively apply the same principles for the children, ensuring that their PreStart() method is called only at the creation of their refs.

For more information see [What Restarting Means](https://getakka.net/articles/concepts/supervision.html#what-restarting-means).

Initialization via message passing

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization message, and use Become() or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

public class Service : ReceiveActor

{

private string \_initializeMe;

public Service()

{

Receive<string>(s => s.Equals("init"), init =>

{

\_initializeMe = "Up and running";

Become(() =>

{

Receive<string>(s => s.Equals("U OK?") && \_initializeMe != null, m =>

{

Sender.Tell(\_initializeMe, Self);

});

});

});

}

}

If the actor may receive messages before it has been initialized, a useful tool can be the Stash to save messages until the initialization finishes, and replaying them after the actor became initialized.