cell Documentation

Release 0.0.3

Ask Solem
Contributors

January 19, 2017

Contents

Contents:

CHAPTER 1

cell - Actor framework

Version 0.0.3

1.1 Synopsis

cell is an actor framework for Kombu and celery .

1.2 What is an Actor

The actor model was first proposed by Carl Hewitt in 1973 [1] and was improved, among others, by Gul Agha [2].

An Actor is an entity (a class in cell), that has a mailbox and a behaviour. Actors communicate between each other only by exchanging messages. Upon receiving a message, the behaviour of the actor is executed, upon which the actor can send a number of messages to other actors, create a number of actors or change its internal state.

Cell supports:

- distributed actors (actors are started on celery workers)
- Remoting: Communicating with actors running on other hosts
- Routers: it supports round-robin, direct and broadcast delivery of actor messages. You can create custom routers on top of it, implementing Actors as routers (joiner, collector, gatherer).

1.3 Why should I use it?

In a nutshell:

- Horizontal scalability with actors across multiple nodes
- You get asynchronous message passing for free
- If you are already using celery, all comes for free, no additional setup is required
- Control over the tasks distribution
- More flexible configurations of nodes
- Well known abstraction
- Easy learning curve (check teh 30 sec video to get you started)

1.3.1 if you are a celery user:

- You can use Actors, instead of task-based classes:

(You can program with classes and not tasks)

- Stateful execution. You can link actors together and their execution, creating complex workflows.

You can control the execution per actor/not per worker.

- Better control over work distribution (You can target the same worker for a given task):

```
adder.send.add(2, 2)
adder.send.add(2, 2)
```

1.3.2 If you are a general Pythonist

Having a framework for distributed actor management in your toolbox is a must, bacause:

- simplify the distributed processing of tasks.
- vertical scalability:
- Fair work distribution, load balancing, sticky routing

1.4 Installation

You can install *cell* either via the Python Package Index (PyPI) or from source.

To install using *pip*,:

```
$ pip install cell
```

To install using *easy_install*,:

```
$ easy_install cell
```

If you have downloaded a source tarball you can install it by doing the following,:

```
$ python setup.py build
# python setup.py install # as root
```

1.5 Quick how-to

If you are too impatient to start, here are the 3 quick steps you need to run 'Hello, world!' in cell: (You can also check the Demo video)

- Define an Actor

```
from cell.actors import Actor

class GreetingActor(Actor):
    class state:
        def greet(self, who='world'):
            print 'Hello %s' % who
```

- Start celery with an amqp broker support

```
>>> celery worker -b 'pyamqp://guest@localhost'
```

- Invoke a method on an actor instance:

```
from cell.agents import dAgent
from kombu import Connection
from examples.greeting import GreetingActor

connection = Connection('amqp://guest:guest@localhost:5672//')
agent = dAgent(connection)
greeter = agent.spawn(GreetingActor)
greeter.call('greet')
```

The full source code of the example from `examples` module. To understand what is going on check the *Getting started* section.

1.6 Getting Help

1.6.1 Mailing list

Join the carrot-users mailing list.

1.7 Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at http://github.com/celery/cell/issues/

1.8 Contributing

Development of *cell* happens at Github: http://github.com/celery/cell

You are highly encouraged to participate in the development. If you don't like Github (for some reason) you're welcome to send regular patches.

1.9 License

This software is licensed under the *New BSD License*. See the *LICENSE* file in the top distribution directory for the full license text.

1.10 Copyright

Copyright (C) 2011-2013 GoPivotal, Inc.

CHAPTER 2

Getting started examples

2.1 Hello World

Here we will demonstrate the main cell features starting with a simple Hello World example. To understand what is going on, we will break the example into pieces.

2.1.1 Defining an actor

Actors are implemented by extending the `Actor` class and then defining a series of supported methods. The supported methods should be encapsulated in the Actor's internal `state` class. Our first actor implements one method called greet.

```
from cell.actors import Actor

class GreetingActor(Actor):
    class state:
        def greet(self, who='world'):
            print 'Hello %s' % who
```

2.1.2 Starting an actor

Cell targets distributed actors management. Therefore, creating an actor means spawning an actor consumer remotely by sending a command to a celery worker agent. That is why before spawning an actor, you need to have a celery worker running:

```
$ celery worker
```

Actors are created via `spawn()` method, called on an instance of `dAgent` (distributed agent). Agents are components responsible for creating and stopping actors on celery workers. Each celery worker has embedded agent

component `dAgent` (distributed agent) listening for commands. `spawn()` is invoked with a class of type `Actor` or its derivative and starts an actor of that type remotely (in the celery worker). The method returns a proxy (an instance of `ActorProxy`) for the remotely started actor. The proxy holds the unique identifier, which ensures the messages can be delivered unambiguously to the same remote actor.

The code below starts a `GreetingActor` and then invokes its py:meth:*greeting* method.

```
from cell.agents import dAgent
from kombu import Connection

connection = Connection()
# STEP 1: Create an agent
agent = dAgent(connection)
# STEP 2: Pass the actor type to spawn method
greeter = agent.spawn(GreetingActor)

# STEP 3: Use actor proxy to call methods on the remote actor
greeter.send('greet')
```

2.1.3 Calling a method

The cell actor model comes with few build-in delivery policies. Here we demonstrate direct delivery - the message is send to a particular actor instance. (Look at the end of the section for the other delivery options.)

```
actor.send('greet')
```

The `greet()` method has been executed and you can verify that by looking at the workers console output. The remote actor of type GreetingActor you created earlier handle the method. The message 'Hello world' should be printed on the celery worker console. Note that the call is asynchronous and since 'great' is a void method no result will be returned. For getting results back and invoking methods synchronously check Getting a result back section

The basic Actor API expose three more methods for sending a message:

- `send()` - sends to a particular actor instance
- `throw()` - sends to an actor instance of the same type
- `scatter()` - sends to all actor instances of the same type

For more information on the above options, see the Delivery options section

2.1.4 Calling a method with parameters

Here is an example how to call the method `greet()` with an argument `who`

```
actor.send('greet', {'who':'everyone'})
```

2.1.5 Getting a result back

Let's add another method to the `GreetingActor` class `how_are_you()` that returns a result.

```
from cell.actors import Actor

class GreetingActor(Actor):
    class state:
        def greet(self, who='world'):
            print 'Hello %s' % who
```

```
        def how_are_you(self):
            return 'Fine!'
```

We can get the result in two ways:

- using a **blocking call** (set the nowait parameter to True), it blocks the execution until a result is delivered or a timeout is reached:

```
result = actor.send('greet', {'who':'everyone'}, nowait=True)
```

d.. warning:: If you are using blocking calls, greenlets should be enabled in the celery worker:

Greenlets can be enabled either by using eventlet or using gevent:

```
>>> celery worker -P eventlet -c 100
```

or

```
>>> celery worker -P gevent -c 100
```

You can read more about concurrency in celery in here

- using a **non-blocking call** (set the nowait parameter to False, the default), it returns an an *AsyncResult* instance.

AsyncResult can be used to check the state of the result, get the return value or if the method failed, the exception and traceback).

```
result = actor.send('greet', {'who':'everyone'}, nowait=False)
```

The `result()` returns the result if it is ready or wait for the result to complete

```
result = actor.send('greet', {'who':'everyone'}, nowait=False)
print result.result
```

See cell.result for the complete result object reference.

2.1.6 Stopping an actor

We can stop an actor if we know its id.

```
agent.kill(actor.id)
```

`agents.dAgent.kill()` is a broadcast command sent to all agents. If an agents doesn't have in its registry the given actor.id, it will dismiss the command, otherwise it will gently kill the actor and delete it from its registry.

2.1.7 Where to go from here

If you want to learn more you should explore the examples in the `examples` module in the cell codebase and/or study the *User Guide*.

2.2 Adder

Actor that can add one to a given number. Adder actor can be used to implement a Counter.

```
from kombu import Connection
connection = Connection()
agent = dAgent(connection)

class Adder(Actor):
    class state():
        def add_one(self, i):
            print 'Increasing %s with 1' % i
            return i + 1

if __name__=='__main__':
    import examples.adder
    adder = agent.spawn(Adder)

    adder.call('add-one', {'i':10})
```

2.3 Chat-users

```
from cell.actors import Actor
from cell.agents import dAgent

connection = Connection()

class User(Actor):
    class state():

        def post(self, msg):
            print msg

    def post(self, msg):
        msg = 'Posting on the wall: %s' % msg
        self.scatter('post', {'msg': msg})

    def message_to(self, actor, msg):
        a = User(id = actor, connection = self.connection)
        msg = 'Actor %s is sending you a message: %s' %(self.id, msg)
        a.call('post', {'msg':msg})

    def connect(self):
        if not agent:
            agent = dAgent(self.connection)
        return self.agent.spawn(self)

if __name__=='__main__':
    import examples.chat
    rumi = examples.chat.User(connection).connect()
    rumi.post('Hello everyone')

    ask = examples.chat.User(connection).connect()
    ask.post('Hello everyone')
    rumi.message_to(ask.id, 'How are you?')
    ask.message_to(rumi.id, 'Fine.You?')
```

2.4 Map-reduce

```
import celery
from cell.actors import Actor
from cell.agents import dAgent

my_app = celery.Celery(broker='pyamqp://guest@localhost//')
agent = dAgent(connection=my_app.broker_connection())


class Aggregator(Actor):

    def __init__(self, barrier=None, **kwargs):
        self.barrier = barrier
        super(Aggregator, self).__init__(**kwargs)

    class state(Actor.state):
        def __init__(self):
            self.result = {}
            super(Aggregator.state, self).__init__()

        def aggregate(self, words):
            for word, n in words.iteritems():
                self.result.setdefault(word, 0)
                self.result[word] += n

            self.actor.barrier -= 1
            if self.actor.barrier <= 0:
                self.print_result()

        def print_result(self):
            for (key, val) in self.result.iteritems():
                print "%s:%s" % (key, val)


class Reducer(Actor):

    class state(Actor.state):
        def __init__(self):
            self.aggregator = None
            super(Reducer.state, self).__init__()

        def on_agent_ready(self):
            self.aggregator = Aggregator(connection=self.actor.connection)

        def count_lines(self, line, aggregator):
            words = {}
            for word in line.split(" "):
                words.setdefault(word, 0)
                words[word] += 1
            self.aggregator.id = aggregator
            self.aggregator.call('aggregate', {'words': words})

    def on_agent_ready(self):
            self.state.on_agent_ready()


class Mapper(Actor):
```

```
    class state(Actor.state):
        REDUCERS = 10

        def on_agent_ready(self):
            self.pool = []
            for i in range(self.REDUCERS):
                reducer = self.actor.agent.spawn(Reducer)
                self.pool.append(reducer)

        def count_document(self, file):
            with open(file) as f:
                lines = f.readlines()
                count = 0
                self.aggregator = agent.spawn(Aggregator, barrier=len(lines))
                for line in lines:
                    reducer = self.pool[count % self.REDUCERS]
                    reducer.cast('count_lines',
                                 {'line': line,
                                  'aggregator': self.aggregator.id})

    def on_agent_ready(self):
        self.state.on_agent_ready()

if __name__ == '__main__':
    import examples.map_reduce
    file = "map_reduce_test.txt"
    mapper = agent.spawn(examples.map_reduce.Mapper)
    mapper.call('count_document', {'file': file})
```

CHAPTER 3

User Guide

"We are all merely Actors" Ask Solem

3.1 Basics

Quick Cheat Sheet for Agents

- `a.spawn(cls, kwarg=value)` start a remote actor instance
- `a.select(cls)` returns a remote actor instance if actor of type cls is already started
- `a.kill(id)` stops an actor by id

Quick Cheat Sheet for Actors

- `a.send.method(kwarg=value, nowait=False)` invoke method on the remote actor instance asynchronously returns AsyncResult
- `a.send.method(kwarg=value, nowait=False)` invoke method on a remote actor instance synchronously
- `a.throw.method(kwarg=value)` invoke method on a remote actor of type a instance asynchronously
- `a.scatter.method(kwarg=value)` invoke method on all remote actors of type a

3.2 Create an Actor

To implement a new actor extend `Actor` class. Actor behaviour (all supported methods) should be implemented in the internal `state` class.

```
class Logger(Actor):
    class state(Actor.state):
        def log(self, msg):
            print msg
```

Then the actor can be started (spawned) remotely using `cell.agents.dAgent.spawn()`

```
from kombu import Connection
from cell.agents import dAgent

agent = dAgent.Connection()
logger_ref = agent.spawn(Logger)
```

3.3 Actors and ActorProxy

We do not create instances of actors directly, instead we ask an `dAgent` to spawn (instantiate) an Actor of given type on a remote celery worker.

```
logger_ref = agent.spawn(Logger)
```

The returned object (logger_ref) is not of Logger type like our actor, it is not even an Actor. It is an instance of `ActorProxy`, which is a wrapper (proxy) around an actor: The actual actor can be deployed on a different machine on different celery worker.

`ActorProxy` transparently and invisibly to the client sends messages over the wire to the correct worker(s). It wraps all method defined in the *Actor.state* internal class.

3.4 Select an existing actor

If you know that an actor of the type you need is already spawned, but you don't know its id, you can get a proxy for it as follows:

```
from examples.logger import Logger
try:
    logger = agent.select(Logger)
except KeyError:
    logger = agent.spawn(Logger)
```

In the above example we check if an actor is already spawned in any of the workers. If Logger is found in any of the workers, the `agents.Agent.select()` will throw an exception of type `KeyError`.

3.5 Actor Delivery Types

Here we create two actors to use throughout the section.

```
logger1 = agent.spawn(Logger)
logger2 = agent.spawn(Logger)
```

logger1 and logger2 are ActorProxies for the actors started remotely as a result of the spawn command. The actors are of the same type (Logger), but have different identifiers.

Cell supports few sending primitives, implementing the different delivery policies:

- direct (using `send()`) - sends a message to a concrete actor (to the invoker)

```
logger1.send('log', {'msg':'the quick brown fox ...'})
```

The message is delivered to the remote counterpart of logger1.

- round-robin (using `throw()`) - sends a message to an arbitrary actor with the same actor type as the invoker

```
logger1.throw('log', {'msg':'the quick brown fox ...'})
```

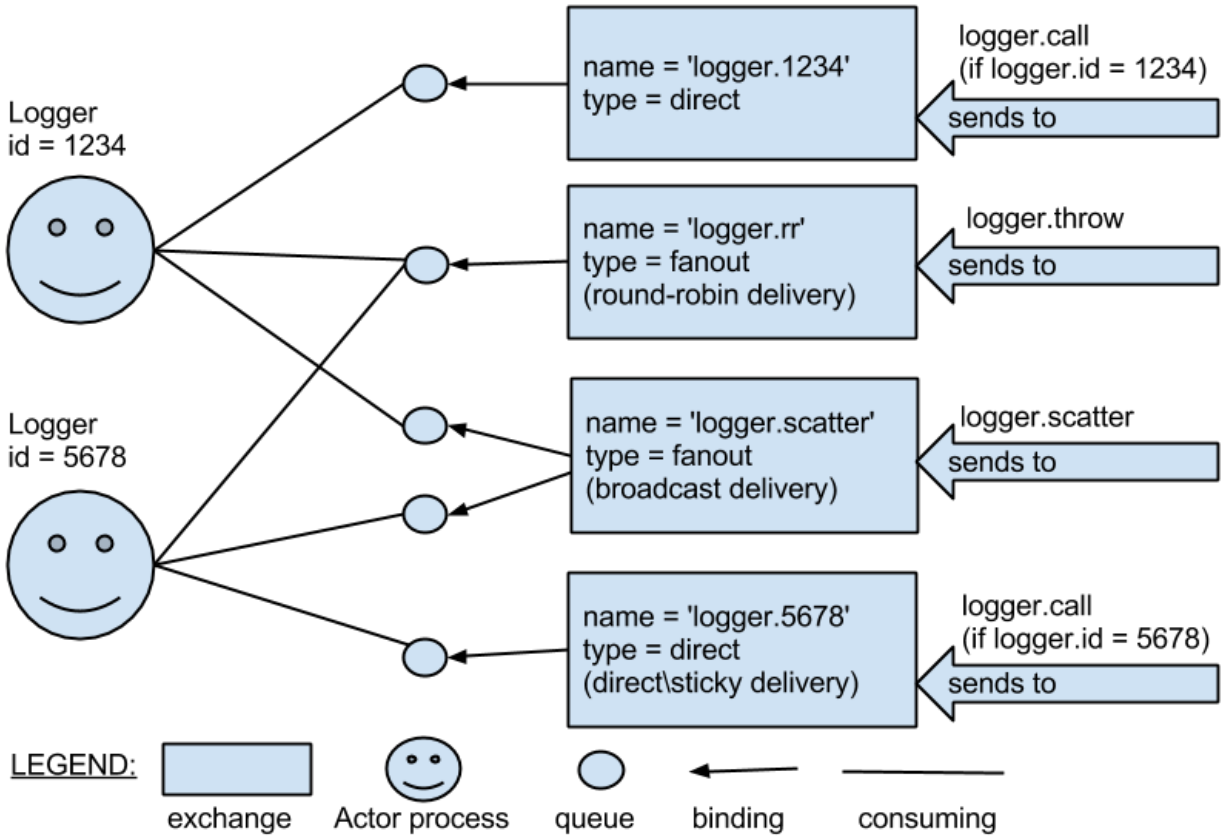
The message is delivered to the remote counterparts of either logger1, or logger2.

- broadcast (using `scatter()`) - sends a message to all running actors, having the same actor type as the invoker

```
logger1.scatter('log', {'msg':'the quick brown fox ...'})
```

The message is delivered to both logger 1 and logger 2. All running actors, having a type Logger will receive and process the message.

The picture below sketches the three delivery options for actors. It shows what each primitive does and how each delivery option is implemented in terms of transport entities (exchanges and queues). Each primitive (depending on its type and the type of the actor) has an exchange and a routing key to use when sending.



3.6 Emit method or how to bind actors together

In addition to the inboxes (all exchanges, explained in the Actor Delivery Types), each actor also have an outbox. Outboxes are used when we want to bind actors together. An example is forwarding messages 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. They are also useful for returning result back to the caller.

How it works? The `emit()` method explicitly send a message to its actor outbox. By default, no one is listening to the actor outbox. The binding can be added and removed dynamically when needed by the application. (See `add_binding()` and `remove_binding()` for more information) The **|forward|** operator is a handy wrapper around the `add_binding()` method. Fir example The code below binds the outbox of logger1 to the inbox of logger2 (logger1 **|forward|** logger 2) Thus, all messages that are send to logger1 (via `emit()`) will be received by logger 2.

```
logger1 = agent.spawn(Logger)
logger2 = agent.spawn(Logger)

logger1 |forward| logger2
logger1.emit('log', {'msg':'I will be printed from logger2'})
logger1 |stop_forward| logger2
logger1.emit('log', {'msg':'I will be printed from logger1'})
```

3.7 Type your method calls

Passing actor methods as a string is often not a convenient option. That's why we provide an API to call the method directly. The ActorProxy class returns a partial when any of the Actor API methods is called (call, send, throw, scatter) actor.api_method.state_method_to_be_called returns a partial application of actor.api_method with firtsh argument set to method_to_be_called. Therefore, the following pair of executions on the instance of :py:class: *ActorProxy* are teh same Note that if the state_method_to_be_called does not exist in the :py:class: *ActorProxy.state* an exception (`AttributeError()`) will be thrown.

```
logger.send.log({'msg':'Hello'}, nowait=False)
logger.send('log', {'msg':'Hello'}, nowait=False)

logger.call.log({'msg':'Hello'}, nowait=False)
logger.call('log', {'msg':'Hello'}, nowait=False)

logger.throw.log({'msg':'Hello'}, nowait=False)
logger.throw('log', {'msg':'Hello'}, nowait=False)

logger.scatter.log({'msg':'Hello'}, nowait=False)
logger.scatter('log', {'msg':'Hello'}, nowait=False)

# throws `AttributeError` on the sender
# and the message is not sent
logger.scatter.my_log({'msg':'Hello'}
```

3.8 Getting the result back

Cell supports several types of return patterns:

- fire and forget - whenever the nowait parameter is set to True, apply to all `cell.actors.Actor()`) methods

- returning future - apply only to `cell.Actors.call()`. It returns `cell.Actors.AsyncResults` instance when invoked with nowait is False. The result can be accessed when needed via `cell.Actors.AsyncResults.result()`.
- blocking call (returning the result) - apply to `cell.Actors.send()` and `cell.Actors.throw()` when invoked with nowait = False
- generator - apply to `cell.Actors.scatter()` when invoked with nowait = False

3.8.1 ScatterGatherFirstCompletedActor

Here is how you can send a broadcast message to all actors of a given type and wait for the first message that is received.

```
from cell.agents import first_reply
first_reply(actor.scatter('greet', limit=1)
```

You can implement you own `first_reply()` function. Remember that the scatter method returns generator. Then all you need to do is call its next() method only once:

```
def first_reply(replies, key):
    try:
        return replies.next()
    except StopIteration:
        raise KeyError(key)
```

3.8.2 Collect all replies

if limit is not specifies, the

```
# returns a generator
res = actor.scatter('greet', timeout = 5)

# Iterate over all the results until a timeout is reached
for i in res:
    print i
```

3.9 Request-reply pattern

Note: When using actors, always start celery with greenlet support enabled!

(see **'Greenlets in celery'_** for more information)

Depending on your environment and requirements, you can start green workers in one of these ways:

When greenlet is enbaled, each method is executed in its own greenlet.

Actor model prescribes that an actor should not block and wait for the result of another actor. Therefore, the result should always be passed via callback. However, if you are not a fen of the CPS (continuation passing style) and want to preserve your control flow, you can use greenlets.

Below, the two options (callbacks and greenlets) are explained in more details:

- **via greenlets**

Warning: To use this option, GREENLETS SHOULD BE ENABLED IN THE CELERY WORKERS running the actors. If not, a deadlock is possible.

Below is an example of Counter actor implementation. To count to a given target, the Counter calls the Incrementer inc method in a loop. The Incrementer advance the number by one and returned the incremented value. The loop continues until the final count target is reached.

```
class Incrementer(Actor):
    class state:
        def inc(self, n)
            return n + 1

    def inc(self, n):
        self.send('inc', {'n':n}, nowait=False)

class Counter(Actor):
    class state:
        def count_to(self, target)
            incrementer = self.agent.spawn(Incrementer)
            next = 0
            while target:
                print next
                next = incrementer.inc(next)
                target -= 1
```

The actors (Counter and Incrementer) can run in the same worker or can run in a different workers and the above code will work in both cases.

What will happen if celery workers are not greenlet enabled?

If the actors are in the same worker and this worker is not started with a greenlet support the Counter worker will be blocked, waiting for the result of the Incrementer, preventing the Incrementer from receiving commands and therefore causing a dealock. If the worker supports greenlets, only the Counter greenlet will block, allowing the worker execution flow to continue.

- **via actor outboxes**

```
class Incrementer(Actor):
    class state:
        def inc(self, i, token=None):
            print 'Increasing %s with one' % i
            res = i + 1
            # Emit sends messages to the actor outbox
            # The actor outbox is bound to the Counter inbox
            # Thus, the message is send to teh Counter
            # and its count message is invoked.
            self.actor.emit('count', {'res': res, 'token': token})
            return res

    def inc(self, n):
        self.send('inc', {'n':n}, nowait=False)

class Counter(Actor):
    class state:
        def __init__(self):
        self.targets = {}
        self.adder = None

    # Here we bind the outbox of Adder to the inbox of Counter.
```

```
    # All messages emitted to Adder are delegated to the Counter inbox.
    def on_agent_ready(self):
        ra = Adder(self.actor.connection)
        self.adder = self.actor.agent.add_actor(ra)
        self.adder |forward| self.actor

    def count(self, res, token):
        if res < target:
            self.adder.throw('add_one', {'i': res, 'token': token})
        else:
            print 'Done with counting'

    def count_to(self, target):
        self.adder.throw('add_one', {'i': 0, 'token': token})

def on_agent_ready(self):
    self.state.on_agent_ready()
```

The above example uses the outbox of an actor to send back the result. All operations are asynchronous. Note that as a result of asynchrony, the counting might not be in order. Different measures should be takes to preserve the order. For example, a token can be assigned to each request and used to order the results.

3.10 Scheduling Actors in celery

- when greenlets are disabled

All messages are handled by the same thread and processed in order of delivery. Thus, it is up to the broker in what order the messages will be delivered and processed. If one actor blocks, the whole thread will be blocked.

- when greenlets are enabled

Each message is processed in a separate greenlet. If one greenlet/actor blocks, the execution is passed to the next greenlet and the (system) thread as a whole is not blocked.

3.11 Receiving arbitrary messages

An actor message has a name (label) and arguments. Each message name should have a corresponding method in the Actor's internal state class. Otherwise, an error code is returned as a result of the message call. However, if fire and forget called is used (call with nowait argument set to True), no error code will be returned. You can find the error expecting the worker log or the the worker console.

If you want to implement your own pattern matching on messages and/or want to accept generic method names, you can override the `default_receive()` method.

3.12 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.

3.12.1 What happens to the Message

If an exception is thrown while a message is being processed (i.e. taken out of its queue 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 queue. 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.

3.12.2 What happens to the actor

Actor is not stopped, either restarted, it can continue receiving other messages.

CHAPTER 4

Change history

4.1 0.0.1

release-date TBA

- Initial version.

CHAPTER 5

API Reference

Release 0.0

Date January 19, 2017

5.1 cell.actors

5.2 cell.agents

5.3 cell.results

cell.result

class `cell.results.`**`AsyncResult`**(*ticket*, *actor*)

`Error`
alias of `CellError`

exception `NoReplyError`
No reply received within time constraint

`AsyncResult.`**`gather`**(*propagate=True*, ***kwargs*)

`AsyncResult.`**`get`**(***kwargs*)
What kind of arguments should be pass here

`AsyncResult.`**`result`**(***kwargs*)

`AsyncResult.`**`to_python`**(*reply*, *propagate=True*)
Extracts the value out of the reply message.

Parameters **`reply`** – In the case of a successful call the reply message will be:

```
{'ok': return_value, **default_fields}
```

Therefore the method returns: return_value, **default_fields

If the method raises an exception the reply message will be:

```
{'nok': [repr exc, str traceback], **default_fields}
```

:keyword propagate - Propagate exceptions raised instead of returning a result representation of the error.

5.4 cell.exceptions

cell.exceptions

exception `cell.exceptions.`**`CellError`**(*exc=None*, *traceback=None*)
Remote method raised exception.

`exc` = None

`traceback` = None

exception `cell.exceptions.`**`Next`**
Used in a gather scenario to signify that no reply should be sent, to give another agent the chance to reply.

exception `cell.exceptions.`**`NoReplyError`**
No reply received within time constraint

exception `cell.exceptions.`**`NotBoundError`**
Object is not bound to a connection.

5.5 cell.presence

5.6 cell.models

5.7 cell.g

5.8 cell.g.eventlet

5.9 cell.utils

cl.utils

`cell.utils.`**`force_list`**(*obj*)

`cell.utils.`**`flatten`**(*it*)

`cell.utils.`**`instantiate`**(*name*, **args*, ***kwargs*)
Instantiate class by name.

See `get_cls_by_name()`.

class `cell.utils.`**`cached_property`**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
Cached property descriptor.

Caches the return value of the get method on first call.

Examples:

```
        @cached_property
        def connection(self):
            return Connection()

        @connection.setter  # Prepares stored value
        def connection(self, value):
            if value is None:
                raise TypeError('Connection must be a connection')
            return value

        @connection.deleter
        def connection(self, value):
            # Additional action to do at del(self.attr)
            if value is not None:
                print('Connection {0!r} deleted'.format(value)
```

deleter(*fdel*)

setter(*fset*)

5.10 cell.bin.cell

5.11 cell.bin.base

cell.bin.base

class cell.bin.base.**Option**(**opts*, ***attrs*)

Instance attributes: _short_opts : [string] _long_opts : [string]

action : string type : string dest : string default : any nargs : int const : any choices : [string] callback : function callback_args : (any*) callback_kwargs : { string : any } help : string metavar : string

ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count', 'callback', 'help', 'versio

ALWAYS_TYPED_ACTIONS = ('store', 'append')

ATTRS = ['action', 'type', 'dest', 'default', 'nargs', 'const', 'choices', 'callback', 'callback_args', 'callback_kwargs', 'help

CHECK_METHODS = [<function _check_action at 0x7f9f80076050>, <function _check_type at 0x7f9f800760c8>, <function

CONST_ACTIONS = ('store_const', 'append_const')

STORE_ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count')

TYPED_ACTIONS = ('store', 'append', 'callback')

TYPES = ('string', 'int', 'long', 'float', 'complex', 'choice')

TYPE_CHECKER = {'int': <function check_builtin at 0x7f9f80071c80>, 'float': <function check_builtin at 0x7f9f80071c80

check_value(*opt*, *value*)

convert_value(*opt*, *value*)

get_opt_string()

process(*opt*, *value*, *values*, *parser*)

take_action(*action*, *dest*, *opt*, *value*, *values*, *parser*)

takes_value()

class `cell.bin.base.`**Command**

Parser
alias of `OptionParser`

args = ''

create_parser(*prog_name*)

execute_from_commandline(*argv=None*)
Execute application from command line.

Parameters **argv** – The list of command line arguments. Defaults to `sys.argv`.

exit(*v=0*)

exit_status(*msg*, *status=0*, *fh=<open file '<stderr>', mode 'w'>*)

exit_usage(*msg*)

get_options()
Get supported command line options.

handle_argv(*prog_name*, *argv*)
Parses command line arguments from `argv` and dispatches to *run()*.

Parameters

- **prog_name** – The program name (`argv[0]`).
- **argv** – Command arguments.

option_list = ()

parse_options(*prog_name*, *arguments*)
Parse the available options.

prog_name = None

run(**args*, ***options*)

usage()
Returns the command-line usage string for this app.

version = '0.0.3'

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

C

U

V