Knit: A new tool for Releases and Upgrades

Paul Joseph Davis
Sr. Software Engineer
@davisp
Summary

• Hot Code Loading
• Releases
• Upgrades
• Appups and Relups
• Using Upgrades
• More about Knit
• How things Break
Hot Code Loading
You’ve Probably Used Hot Code Loading

Eshell V5.8.2  (abort with ^G)
1> c(foo).
{ok, foo}
Hot Code Loading Constraints

• Only allowed two versions of a module in the VM
• Processes with code from v1 are killed automatically when v3 is loaded
• Processes run new code by calling exported functions
Example 1 - A Successful Upgrade

-module(good_upgrade).
-export([start/0, loop/0]).

start() ->
  erlang:spawn(?MODULE, loop, []).

loop() ->
  Vsn = lists:keyfind(vsn, 1, ?MODULE:module_info(attributes)),
  io:format("Version: ~p~n", [Vsn]),
  timer:sleep(2000),
  ?MODULE:loop(). % Notice the use of ?MODULE
Example 1 - A Successful Upgrade

Eshell V5.8.2 (abort with ^G)
1> c(good_upgrade).
   {ok,good_upgrade}
2> good_upgrade:start().
   <0.38.0>
   {vsn,[285322158962536634385124857288843166172]}
   {vsn,[285322158962536634385124857288843166172]}
3> c(good_upgrade).
   {ok,good_upgrade}
   Vsn: {vsn,[243367076262672122378804240543149085496]}
   Vsn: {vsn,[243367076262672122378804240543149085496]}
4> c(good_upgrade).
   {ok,good_upgrade}
   Version: {vsn,[160372567835089398502372253338826710031]}
   Version: {vsn,[160372567835089398502372253338826710031]}
Example 2 - Upgrade Failure

-module(bad_upgrade).
-export([[start/0, loop/0]]).

start() ->
  erlang:spawn(?MODULE, loop, []). 

loop() ->
  Vsn = lists:keyfind(vsn, 1, ?MODULE:module_info(attributes)),
  io:format("Version: ~p~n", [Vsn]),
  timer:sleep(2000),
  loop(). % No more ?MODULE
Example 2 - Upgrade Failure

Eshell V5.8.2 (abort with ^G)
1> c(bad_upgrade).
   {ok,bad_upgrade}
2> bad_upgrade:start().
   <0.38.0>
   {vsn,[181013074981266123478501823959170679836]}
   {vsn,[181013074981266123478501823959170679836]}
3> c(bad_upgrade).
   {ok,bad_upgrade}
   {vsn,[168525046126506918599002166162913726653]}
   {vsn,[168525046126506918599002166162913726653]}
4> erlang:monitor(process, pid(0, 38, 0)).
   #Ref<0.0.0.109>
5> flush().
   ok
   {vsn,[168525046126506918599002166162913726653]}
   {vsn,[168525046126506918599002166162913726653]}
6> c(bad_upgrade).
   {ok,bad_upgrade}
7> flush().
   Shell got {'DOWN',#Ref<0.0.0.109>,process,<0.38.0>,killed}
   ok
Hot Code Loading in Production

• Ops duty, 3am Saturday morning.
• Fires are burning.
• I need a log message!

laptop $ vim apps/app/src/foo.erl
laptop $ rebar compile
laptop $ scp apps/app/ebin/foo.beam prod1:/opt/relname/lib/app-vsn/ebin/foo.beam
laptop $ ssh prod1
prod1 $ remsh
Eshell V5.8.2 (abort with ^G)
1> nl(foo).
   abcast
2>
Be Careful!

• l/1 vs nl/1 - “The problem came back!”
• nl/1 and node reboots
• Upgrades can un-patch code
• What code is this server running?!
• Behavior changes are a bit harder
• code_change/3 not called for l/1, nl/1x
Which Processes Might Die?

- erlang:check_old_code/1
  - check_old_code(Module::atom()) -> boolean()
- erlang:check_process_code/2,3
  - check_process_code(Pid::pid(), Module::atom()) -> boolean()

find_old_code() ->
  AllPids = processes(),
  AllMods = [M || {M, F} <- code:all_loaded(), F /= preloaded],
  lists:flatmap(fun(Pid) ->
      FiltFun = fun(Mod) -> check_process_code(Pid, Mod) end,
      case lists:filter(FiltFun, AllMods) of
      [] -> [];
      BadMods -> [{Pid, process_info(Pid, registered_name)}, BadMods]
      end
    end, lists:sort(AllPids)).
Releases
What’s a Release?

• Generally: A tarball containing everything to run an Erlang Application (capital A)
  • Although not necessarily…
  • Optional Erlang VM
  • Optional Application specific data and utilities
• A set of compiled applications that contain a single Erlang Application (capital A)
• More Generally: Compiled Erlang modules with extra metadata as a single file
Contents of a Release

. |
  | └─ erts-5.8.2
  |
  | └─ lib
  |   | └─ $app1-$appvsn1
  |   |   | └─ ebin
  |   |   | └─ priv
  |   | └─ $app2-$appvsn2
  |   |   | └─ ebin
  |   |   | └─ include
  |   | └─ ... 
  |
  └─ releases
      | └─ $relvsn
      |   | └─ $relname.rel
      |   | └─ $relname.script
      |   | └─ $relname.boot
      | └─ RELEASES
      └─ start_erl.data
Important Files

- lib/
- erts-$vsn/
- releases/
- releases/RELEASERS - Textual Erlang term describing each release the node has run or unpacked
- releases/start_erl.data - Text file containing “$ertsvsn $relvsn”
  - 5.8.2 0.0.1
- releases/$relvsn/$relname.rel - Description of the release
- releases/$relvsn/$relname.boot - Binary Erlang term describing how to start the release
Generating a Release

• systools - Very low level library interface
• reltool - Slightly higher library interface
• rebar - Command line interface to reltool using reltool.config
• relx - Replaces reltool and systools
• knit - rebar style reltool.config command line interface (for now)
Upgrades
What’s an Upgrade?

• Turn a VM running a release at version A and turn it into a release running version B
  • Upgrade or downgrade
  • No requirement for linearity
• In practice its mostly just upgrades
  • Make another upgrade if something isn’t working
  • Upgrade failure causes the node to reboot
Contents of an Upgrade

```
lib
| app1-appvsn1
| | ebin
| | priv
| app2-appvsn2
| | ebin
| | include
| ...
releases
| relvsn2
| | relname.boot
| | start.boot
| | relup
| relname-$relvsn.rel
```
Important Files

• lib/
• releases/$relname-$relvsn.rel
• releases/$relvsn/start.boot
• releases/$relvsn/$relname.boot
• releases/$relvsn/relup
What’s a relup?

• An Erlang term “script” that contains the instructions to effect the upgrade for the entire release
• Compiled from app ups
• Uses only the “low-level” instruction set
relup format

{Vsn,
  [{UpFromVsn, Descr, Instructions}, ...],
  [{DownToVsn, Descr, Instructions}, ...]}. 
What’s an appup?

• An Erlang term “script” that contains instructions to effect an upgrade for a single application
• Can contain either “high-level” or “low-level” instructions
• High-level instructions compiled by systools into low-level instructions
• Not quite a direct expansion
Appups and Relups
Instructions

• Both a high and low level instruction set
  • High level is roughly a macro (C pre-processor, not Lisp)
• Handles adding, reloading, and removing code from the Erlang VM
• Various other instructions related to modifying application state, running arbitrary functions, upgrading the VM itself
Example: High-Level instruction

{
    update,
    Mod, % Module name as an atom
    ModType, % dynamic or static, usually dynamic
    Timeout, % time limit to suspend a processes running Mod
    Change, % soft or {advanced, Extra}
    PrePurge, % soft_purge or brutal_purge
    PostPurge, % soft_purge or brutal_purge
    DepMods % list of modules as atoms this module depends on
}

© 2014 IBM Corporation
Example: Corresponding low-level instrs.

{suspend,[Mod]}},
{load,{DepMod1,PrePurge,PostPurge}}},
{load,{DepMod2,PrePurge,PostPurge}}},
{load,{DepModN,PrePurge,PostPurge}}},
{load,{Mod,PrePurge,PostPurge}}},
{code_change,up,[[Mod,[]]]],
{resume,[Mod]]},
{resume,[Mod]]}},
Other Instructions to be Aware Of

- **point_of_no_return** - VM reboots on error after this
  - There are limits on what can happen before this instruction
- **{apply, M, F, A}** - Run an arbitrary function
- **{sync_nodes, Id, Nodes}** - Synchronize the upgrade on a set of nodes
- **{add_application, Application}**
- **{remove_application, Application}**
- **{restart_application, Application}** - Nuke everything and let supervisors restart
- **restart_emulator** - Nuke things harder
- **restart_new_emulator** - Upgrade the Erlang VM
How to Create an appup

- Manually - Steep learning curve
- rebar - Easy-ish without any ability to affect the generated appup
- rebar/manual hybrid - Generate base template with rebar, tweak by hand
- knit - Generates appups based on a set of module attributes
Knit’s Module Attributes

- **knit_priority** - Knit specific, allows for rough ordering of modules
- **knit_extra** - Passed to code_change/3 for behaviors
- **knit_depends** - Set the module dependencies
- **knit_timeout** - Set a timeout for the upgrade
- **knit_purge** - Set Pre/PostPurge strategies
- **knit_apply** - Call a function as part of the upgrade, can control when the function is run
Creating a relup

- systools:make_relup/4 - Not really
- rebar/relx/knit
  - Once you have appups created the command line tools are roughly equal
  - rebar takes a few more manual/scripted steps than the others
Applying an Upgrade
Preparation

• Extract a release somewhere
• Start it
• Copy an upgrade tarball to its `releases` directory
• Run three releaseHandler functions
release_handler

release_handler:unpack_release(RelNameWithVsn).
release_handler:install_release(RelVsn).
release_handler:makePermanent(RelVsn).
release_handler:unpack_release/1

- Extract and validate RelNameWithVsn.rel from `releases/$RelNameWithVsn.rel`
- Expands the upgrade tarball over top of the running release
  - Uses keep_old_files so it doesn’t clobber existing files
- Updates `releases/RELEASERS` with the new release information
release_handler:install_release/1

- Updates each application’s version, description, and environment
- Applies the relup script
- Notifies each application of environment configuration changes
  - via each application’s \{mod, \{Mod, Args\}\} from $appname.app
  - Runs Mod:config_change/3
    - Mod:config_change(Changed, New, Removed)
- Marks the release as installed
  - A node reboot at this point reverts to the previous version
release_handler:makePermanent/1

• Updates `releases/start_erl.data`
• Updates `releases/RELEASES` updating the current release statuses
• Updates `init`'s command line arguments to reflect the new values for -boot and -config if they changed
More about Knit
Knit Stuff

- [https://github.com/davisp/knit](https://github.com/davisp/knit)
- Still **very** alpha, mostly a test bed for ideas on how to generate appups
- README goal is “Just type knit” for 80% of use cases
- Still depends on reltool.config which is non-trivial. Considering replacing this approach
- Removes a lot of reltools/systools knobs in the interest of simplicity
- Upgrade tarballs could be slimmed down considerably
- Considering injecting extra tooling to help with applying upgrades
How Stuff Breaks
No Receive Timeout

wait_forThing() ->
  receive
    {thing, Thing} -> do(Thing)
  end.

-export([wait_for_thing/0]).
wait_forThing() ->
  receive
    {thing, Thing} -> do(Thing)
  after 60000 ->
    ?MODULE:wait_for_thing()
  end.
Sharing Records

• Sharing between modules or processes
• Don’t
• Use modules that wrap access
  • Yes, it’s a bit icky accessor/mutator style code
  • But it makes upgrades so much easier
• Very close internally do dictating no records in .hrl files
  • But legacy code…
Messages in the ether

• Old versions of records and messages can exist for a surprisingly long time
• Ordering of code loading can cause surprises
Anonymous Functions

• Are the devil…
• You don’t have to be executing them for them to break upgrades
• API design
  • For callbacks allow either \{M, F, A\}
  • Or at least \{Fun, Acc\} so that Fun can be specified as `fun Module:Function/2`  
• Probably the most common cause of broken upgrades
Supervision Tree Changes

• The dynamic child specification complicates things
• Much harder to automatically create the necessary appup/relup instructions automatically
• knit_apply should make these possible without more direct intervention
• Luckily its not a super common requirement (hopefully)
Don’t spam release_handler (the process)

• release_handler does some heavy weight operations in process
• Using release_handler:which_releases/0 to get the current release version is not a good idea
• Generally it just makes applying upgrades painfully slow
RPC Protocol Upgrades

• One of the harder upgrades to make
• Requires special attention when relying code on foreign nodes
  • We still haven’t played with sync_nodes internally
• Not entirely sure if there’s a knit specific solution
Questions?

(Also, we’re hiring)
Links

• http://learnyousomeerlang.com/release-is-the-word
• http://learnyousomeerlang.com/relups
• http://www.erlang.org/doc/man/relup.html
• http://www.erlang.org/doc/man/appup.html
• http://www.erlang.org/doc/man/reltool.html
• http://www.erlang.org/doc/man/systools.html