Leo-Babel

Contents

1 Introduction

Leo-Babel is very different from Ctrl-B (execute-script). Ctrl-B executes a script using the Leo-Editor process and it gives the script full access to the Leo-Editor code and data in RAM. Leo-Babel executes a script in a sub-process of the

Leo-Editor process. Consequently, the script has no access to the Leo-Editor address space and hence no direct access to the Leo-Editor code or data in RAM. However, a Leo-Babel script can use Leo-Bridge to access and change any Leo-Editor file.

Leo-Babel is modeled on Emacs-Babel. However, Leo-Babel currently supports only two script languages, while Emacs-Babel supports over 50 script languages.

Emacs-Babel does **NOT** "live stream" the script's standard out or standard error while the script executes. Instead Emacs-Babel only displays the script's output after the script terminates. Leo-Babel does "live stream" the script's standard out and standard error while the script executes. This is Leo-Babel's only significant improvement over Emacs-Babel.

Emacs-Babel does **NOT** provide a convenient way to kill a script that misbehaves (perhaps by running much too long). Leo-Babel provides a convenient way to kill a script that misbehaves.

Emacs-Babel allows a script to be treated as a function. Leo-Babel does **not** do this.

Emacs-Babel allows "interpreter sessions." One interpreter session can be used to execute a series of scripts. Leo-Babel does **not** implement sessions.

2 Babel Root

When command babel-exec-p is executed, the currently selected node is the "Babel Root." It's body is the "Babel Parameters Script" which can be empty.

You can put comments in the "Babel Root" body just begin your comments with an ampersand "@" in column 1, this begins a Leo-Editor comment section.

3 Babel Parameters Script

The Babel Parameters Script is executed with the following objects available:

1. babel - The Babel API object.
2. b - The Babel API object.
3. c - The Leo-Editor commander for the Leo-Editor file containing the Babel Root node.
4. g - The Leo-Editor globals.
5. p - The position of the Babel Root node.

The Babel Parameters Script can define the following parameters that affect Babel Script execution:

1. babel_script
2. babel_results
3. babel_node_creation
4. babel_python
5. babel_shell
6. babel_redirect_stdout
7. babel_script_args

The current working directory for the Babel Parameters Script is the working directory for the Babel Script node. See section "Current working directory for a node".

3.1 babel_script

If the script in the Babel Root body defines babel_script, then the specified node is used as the root of the script subtree; else, the first child of the Babel Root node is used as the root of the script subtree.

3.2 babel_results

If the script in the Babel Root body defines babel_results, then the specified node is used as the root of the results subtree; else, the second child of the Babel Root node is used as the root of the results subtree.

3.3 Node Position or UNL

babel_script and babel_results can be either a Leo-Editor (commander, node position) pair or a UNL.

The commander, node position pair can be any iterable, for example a tuple or a list.

If the UNL contains a file pathname part, it can refer to any Leo-Editor file. If the UNL does **NOT** contain a file pathname part, then it refers to the Leo-Editor file containing the UNL.

3.4 babel_node_creation

If babel_node_creation is not defined, then the default for Babel node creation applies.

If babel_node_creation is False, then Leo-Babel does not create its three results nodes for each script run. But it does still display all the resuls data in the log pane.

If babel_node_creation is True, then Leo-Babel creates three results nodes for each script run.

3.5 Python Interpreter

If babel_python is not defined, then the default program for interpreting Python language scripts is used.

If babel_python is defined, then the specified program is used for interpreting Python language scripts.

The program specified must exist somewhere on the path specified by the environment variable PATH or the absolute path to the program must be specified.

Examples:

babel_python = 'python2'

The Python 2 program is used to interpret a Python language script.

babel_python = 'python3'

The Python 3 program is used to interpret a Python language script.

3.6 Shell Interpreter

If babel_shell is not defined, then the default program for interpreting "shell" language scripts is used.

If babel_shell is defined, then the specified program is used for interpreting shell language scripts.

The program specified must exist somewhere on the path specified by the environment variable PATH or the absolute path to the program must be specified.

Examples:

babel_shell = 'bash'

The Bourne shell.

babel_shell = 'sh'

The POSIX standard shell interpreter chosen by your Linux distribution.

babel_shell = 'zsh'

The Z shell.

3.7 Redirect Stdout to Stderr

If the script in the Babel Root body defines babel_redirect_stdout, it specifies whether or not stdout is redirected to stderr. By default, stdout is **NOT** redirected to stderr.

babel_redirect_stdout

- False –> Do not redirect stdout. This is the default, if babel_redirect_stdout does not exist.
- True –> Redirect stdout to stderr

3.8 Babel Script arguments

If the Babel Parameters Script defines babel_script_args, then these arguments are passed to the Babel Script as command line arguments. So if babel_script_args is defined, then it must be a list of strings.

The first command line argument is always the file pathname of the script file. The babel_script_args begin with the second command line argument. For Python scripts the babel_script_args are sys.argv[1:]. For Bash scripts the babel_script_args are $@.

3.9 Splitting a Large Babel Parameters Script into several nodes

A Babel Parameters Script can be split into a subtree of nodes using any one of several schemes.

A section reference in the Babel Root node can refer to the third child of the Babel Root node. This third child can be the root of the script subtree.

If babel_script and babel_results are used to place the script and results nodes outside the subtree rooted by the Babel Root node, then the Babel Parameters Script can occupy the subtree rooted by the Babel Root node.

3.10 babel - The Babel API object

When the Babel Parameters Script is executed, "babel" is defined in the global dictionary and it provides access to the Babel API.

3.10.1 babel.unl2pos()

Univeral Node Locator to Leo-Editor Commander, Position List - babel.unl2pos()

```
Call:
    cmdrUnl, posList = babel.unl2pos(unl, cmdr=None)

Arguments:
    unl: Universal Node Locator
    cmdr:  Optional Leo-Editor commander for the file containing the node(s)
        specified by unl. Default:  None

Returns:
    cmdrUnl: Commander for the file containing the position(s) in posList.
    posList:   A list containing in tree order all the positions
        that satisfy the UNL.
        [] (empty list) --> No position satisfies the UNL

Exceptions:
    ValueError

        If unl contains a file pathname part and cmdr is not None,
        then ValueError is raised because both the pathname part
        and the cmdr specify files. This is either redundant or contradictory.

        If unl does NOT contain a file pathname and cmdr is None,
        then ValueError is raised because there is no specification
        of the target file.
```

A UNL consists of an optional protocol prefix, an optional file pathname part, and a required node path part.

If the optional protocol prefix is present, then it must be "unl://". If the optional protocol prefix is present, then the UNL must be "UNL quoted". If the optional protocol prefix is **NOT** present, then the UNL must **NOT** be "UNL quoted".

In order to resolve the specified UNL, babel.unl2pos() opens the specified Leo-Editor file if it is not already open, and it leaves it open. Hence, if in Leo-Editor file X you pass babel.unl2pos() a UNL for Leo-Editor file Y, this always leaves with files X and Y open in Leo-Editor.

3.11 UNL Quoted

“UNL Quoting” a string replaces " " (space) with %20.

Note carefully, “UNL Quoting” differs from “URL Quoting”. “URL Quoting” a string replaces " " (space) with %20, ‘\t’ (tab) with %09, and “”’ (single quote) with %27.

3.12 Debugging a Babel Parameter Script

A Babel Parameter Script is executed without writing it to disk as a “script” file. To aid debugging when a Babel Parameter Script raises an exception, Leo-Babel writes the script with line numbers to the Leo-Editor Log pane. Then it re-raises the exception. The exception message almost always contains a line number which matches the line numbers Leo-Babel writes.

4 UNL’s

The Leo-Editor core provides some Universal Node Locators (UNL’s) support. The Leo-Babel plugin provides additional UNL support.

4.1 Pound Sign (#) Caution

Using a pound sign (#) in a file name can screw up the UNL support provided by both Leo-Editor and Leo-Babel. It is a limitation of the current design that you should not use the pound sign (#) in any file name that appears in any UNL that you use.

This limitation results from UNL support assuming that the first pound sign in a UNL begins the required node part.

Hence, if there is a pound sign in a file name, then UNL support thinks the node part begins with this first pound sign.

4.2 Leo-Editor UNL Support

4.2.1 Ctrl-left-click (command open-url-under-cursor)

If you Ctrl-left-click (command open-url-under-cursor) on a UNL in a node body containing the protocol prefix, the Leo-Editor core changes focus to the specified node. If the specified node is in another Leo-Editor file, then if necessary, Leo-Editor opens this Leo-Editor file. This functionality has nothing to do with Leo-Babel.

4.2.2 open-url command

If you put a UNL with the protocol prefix in the first line of the body of a node, select that node, and execute the open-url command. This selects the node specified by the UNL. Again this support is in the Leo-Editor core and has nothing to do with Leo-Babel.

4.2.3 p.get_UNL()

```
p - Leo-Editor node position
g - Leo-Editor globals

p.get_UNL(with_file=True, with_proto=False, with_index=True)

Example for one position:

g.es(p.get_UNL(False, False, False))
g.es(p.get_UNL(False, False, True))
g.es(p.get_UNL(False, True, False))
g.es(p.get_UNL(False, True, True))
g.es(p.get_UNL(True, False, False))
g.es(p.get_UNL(True, False, True))
g.es(p.get_UNL(True, True, False))
g.es(p.get_UNL(True, True, True))

8 lines of output for the above 8 lines of code:

Root-->space " " tab " " single quote "'"
Root:0-->space " " tab " " single quote "'":0
unl:///tmp/unl.leo#Root-->space%20"%20"%20tab%20" "%20single%20quote%20"'"
unl:///tmp/unl.leo#Root:0-->space%20"%20"%20tab%20" "%20single%20quote%20"'":0
/tmp/unl.leo#Root-->space " " tab " " single quote "'"
/tmp/unl.leo#Root:0-->space " " tab " " single quote "'":0
unl:///tmp/unl.leo#Root-->space%20"%20"%20tab%20" "%20single%20quote%20"'"
unl:///tmp/unl.leo#Root:0-->space%20"%20"%20tab%20" "%20single%20quote%20"'":0
```

Leo-Babel does **NOT** support UNL's produced by "with_index=True".

4.3 Leo-Babel UNL Support

4.3.1 Leo-Babel Help Pop-Up Menu

The UNL on the status line does **NOT** contain the protocol, so after copying and pasting it into a node body you need to add the UNL protocol prefix:

"unl://" and you need to "UNL quote" all spaces by replacing each with %20. Consequently, for convenience Leo-Babel provides the "copy UNL to clipboard" command which provides a "UNL quoted" UNL with the UNL protocol prefix.

These UNL's provided by Leo-Babel always specify the Leo-Editor file containing the specified node. Hence, if you want you can put the Babel Root in File A, the Script Root in File B, and the Results Root in File C.

I recommend always using UNL's that contain the protocol prefix and the file pathname. But if you prefer using UNL's with other formats, then you can obtain them from p.get_UNL() by specifying the appropriate function parameters. Caution: Leo-Babel does **NOT** support UNL's produced by "with_index=True". That is, UNL's with child indices.

4.3.2 babel.un2pos()

Leo-Editor does not provide a convenient function for going from a UNL to (Leo-Editor commander, position list) pair, so for the convenience of Babel Parameter Scripts, Leo-Babel provides babel.unl2pos().

5 Current working directory for a node

The current working directory for a Leo-Editor node is determined as follows. Set the current working directory to the directory containing the Leo-Editor file. Scan from the root down to the target node. Each time an @path directive is encountered, set the current working directory as specified. When the target node is reached, the current working directory is the node's current working directory.

Note that multiple @path nodes allow relative paths to be used conveniently.

An @path directive can be in either the headline or the body–but only the first @path in a body is honored. The rest are ignored.

6 Babel Script

Leo-Editor "sections" and @others allow the script to be split into the whole subtree rooted by the Script Root node.

All directive lines (lines beginning with @) and comments are filtered out before the script is executed.

The script is written to a temporary file and the appropriate interpreter is invoked to execute the script file in a subprocess of the Leo-Editor process. The

current working directory for the script is the current working directory for the currently selected node.

Leo-Babel ignores all headlines.

The script is written to the same file used by Ctrl-B. The default path is $HOME/.leo/scriptFile.py.

You can specify the file to use with the following "Debugging" settings option:

```
@string script_file_path = <pathname>
```

Example:

```
@string script_file_path = /sec/tmp/leoScript.py
```

- Use / to as the path delimiter, regardless of platform.
- The filename should end in .py.
- For Ctrl-B this setting has effect only if the write_script_file setting is True. Currently leoSettings.leo contains:

  @bool write_script_file = True

So by default a script file is written.

The current working directory for the script is the working directory for the Babel Script node.

7 Results

Both the headline and body of the results subtree root are ignored. For each execution of the script the results are: 1) A new "Results Instance" root is the first child of the "results" subtree root. The "Results Instance" headline is the elapsed time of the script execution and the time of script completion. 2) The first child of the Results Instance root has headline "stdout" and body equal to the standard output of the script. 3) The second child of the Results Instance root has headline "stderr" and body equal to the standard error output of the script.

When the script terminates, the new Results Instance root is the selected node.

8 Language

The current language directive (@language) determines the script language.

Currently the only languages allowed are:

- @language python

- @language shell

9 Babel Kill

While Leo-Babel is executing a script, a pop-up window offers the option of killing the Leo-Babel subprocess. This pop-up window is produced by a Python script running in a second sub-process of the Leo-Editor process. When the kill option is selected by clicking the Yes button or by entering carriage return, the pop-up window disappears, it kills the script process (by sending it signal SIGHUP), and the kill process terminates. When the script process terminates normally, the kill window disappears and its process terminates.

The kill window attempts to kill the script process by sending SIGHUP. This usually kills the script process, but the script may explicitly handle SIGHUP without terminating.

9.1 Emacs-Babel Limitation

Emacs-Babel provides no way to kill a script process.

10 Live Streaming Stdout and Stderr

While the script executes, the script's stdout and stderr outputs are printed to Leo-Editor's Log tab.

When the script terminates, the script process's termination code, the script's wall clock elapsed time (hours:minutes:seconds) and termination time are printed to Leo-Editor's Log tab.

Completion Example:

```
0 Subprocess Termination Code
00:00:01 Elapsed Time. 2017-07-05 15:18:37 End Time
```

10.1 stdout, stderr, and completion Default colors

- stdout - green (#00ff00)
- stderr - purple (#A020F0)
- completion - gold (#FFD700)

10.2 Order of stdout and stderr lines in the log pane

The order of stdout and stderr lines in the log pane may not be time order. The log pane output is generated by polling once per second. If there is both stdout and stderr output between polls, then the order of the stdout and stderr lines in the log pane is determined by the order in which stdout and stderr are polled and the timing of the output relative to these polls.

10.3 Customizing Colors

If you want to customize these colors then define Leo-Editor settings Leo-Babel-stdout, Leo-Babel-stderr, Leo-Babel-completion. See the Leo-Editor Settings section.

11 Leo-Babel Node Creation

When the script terminates, Leo-Babel by default inserts three nodes into the Leo-Editor file. By default the second child of the Babel Root node is the root of the "results" subtree. Both the headline and body of the results subtree root are ignored.

11.1 First Node

A new "Results Instance" root is created as the first child of the "results" subtree root. The "Results Instance Root" headline is the elapsed time of the script execution and the time of script completion. Its body contains the script's process termination code. All this information was previously printed to the log pane.

11.2 Second Node

The second node created is the stdout node and it is inserted as the first child of the Results Instance Root. Its headline is "stdout" and its body contains all the stdout output by the script.

11.3 Third Node

The third node created is the stderr node and it is inserted as the second child of the Results Instance Root. Its headline is "stderr" and its body contains all the stderr output by the script.

11.4 Emacs-Babel Limitation

Emacs-Babel only captures stdout. For Emacs-Babel the only way to capture stderr for a script X is to have script X itself redirect stderr to stdout.

12 Leo-Editor Settings

In an @settings subtree in leoMySettings.leo (applies to all your Leo-Editor files) or in a particular Leo-Editor file (applies to just this one Leo-Editor file), add one node per setting with the setting in the headline.

12.1 Customizing Colors

Examples of color settings:

- @color Leo-Babel-stdout = #c8ffbe
- @color Leo-Babel-stderr = #ffc0cc
- @color Leo-Babel-completion = #ffee8b

The default colors are:

```
* stdout 00ff00  green
* stderr A020F0  purple
* completion message FFD700  gold
```

12.2 Node Creation Default

Parameter name: Leo-Babel-Node-Creation-Default

```
* False --> by default, no results nodes are added.
* True --> by default, results nodes are added.
```

Example:

@bool Leo-Babel-Node-Creation-Default = False

If Leo-Babel-Node-Creation-Default is not defined, then Leo-Babel creates results nodes.

This default can be overridden for an individual Babel script by setting babel_node_creation True/False in the Babel Parameters Script.

12.3 Python Interpreter Default

Parameter Name: Leo-Babel-Python

This parameter specifies the program used to interpret a Python language script. The program must exist on the path specified by the PATH environment variable, or the absolute path to the program must be specified.

If Leo-Babel-Python is **NOT** specified, then the default Python interpreter is "python3."

Examples:

@string Leo-Babel-Python = python2

The Python 2 interpreter.

@string Leo-Babel-Python = python3

The Python 3 interpreter.

This default can be overridden for an individual Babel script by setting babel_python in the Babel Parameters Script.

12.4 Shell Interpreter Default

Parameter Name: Leo-Babel-Shell

This parameter specifies the default program used to interpret a shell language script. The program must exist on the path specified by the PATH environment variable, or the absolute path to the program must be specified.

If Leo-Babel-Shell is **NOT** specified, then the default shell interpreter is "bash." Examples:

@string Leo-Babel-Shell = bash

The Bourne shell.

@string Leo-Babel-Shell = sh

The POSIX standard shell interpreter chosen by your Linux distribution.

@string Leo-Babel-Shell = zsh

The Z shell.

This default can be overridden for an individual Babel script by setting babel_shell in the Babel Parameters Script.

13 Supported Python Release

Leo-Babel only works when Python 3 interprets the Leo-Editor code and Python 3 interprets babel_kill.py.

14 Why Use Leo-Babel

I use Leo-Editor as my Personal Information Manager (PIM). Hence, for example, I have many Leo-Editor files containing many Bash scripts along with Descriptions of what they do. Whenever I want to use the command line to do something that I have done before, I search my appropriate Leo-Editor file, copy the commands to the clipboard, open a terminal, and paste the commands into the terminal. This works very well, and it has the advantage of maximum simplicity for the environment of the executing script.

By making a script a Leo-Babel script, I gain some imposed structure and uniformity and automatic logging of every run of the script.

15 Shortcut Advice

A plugin should not bind any keys. That is, set any shortucts. So Leo-Babel limits itself to defining two commands:

- babel-exec-p
- babel-menu-p

If you don't want to use UNL's, then there is no need to make using babel-menu more convenient by assigning it a key binding.

Your key binding(s) can be any sequence that you do not want to use for something else. You can see all the current key bindings by executing Alt-x, print-bindings. You should set your key bindings in the appropriate place in your leoMySettings.leo.

Here is what I use:

```
Headline:
@command babel-exec @key=Shift-Ctrl-B

Body:
c.k.simulateCommand('babel-exec-p')

Headline:
@command babel-menu @key=Shift-Ctrl-H

Body:
c.k.simulateCommand('babel-menu-p')
```

16 Leo-Babel Reports Failed Dependencies

Leo-Babel uses several Python libraries. If you have not installed a Python package that Leo-Babel needs, then the Leo-Babel plugin initialization fails and this error message is output to the Log Pane and to the console:

```
loadOnePlugin: can not load enabled plugin: leo.plugins.leo_babel.babel
```

This occurs when an import statement raises exception ImportError.

Leo-Babel reports the name of each module whose attempted import raises an ImportError exception. These reports are sent to the console and in red to the Log Pane.

17 How to start a terminal using Leo-Babel

The command line required depends on the terminal emulator that you use. Here is an example command line for terminal emulator xfce4-terminal:

```
xfce4-terminal -x ledger -f '/pri/git/Ledger4/data/journal.txt'
```

Since the terminal emulator immediately changes its parent process, the Leo-Babel script execution immediately finishes.

18 sudo works fine, except when several are pasted from the clipboard

When you need to do a series of root-priveleged commands using Leo-Babel, and you want to launch them all at once, the straightforward strategy works:

```
sudo command1
sudo command2
sudo command3
```

Only the first sudo pops up a window asking for a password and it waits for the operator to enter the password.

If you put a series of command lines starting with sudo into a Bash script, this also works fine.

But if you copy a series of command lines to the clipboard and paste them into a terminal, this does **NOT** work because the first sudo consumes the next line as the user's password. But the following does work when these lines are copied to the clipboard and then pasted into a terminal:

```
    gksudo command1
    sudo command2
    sudo command3
```

gksudo pops up a window that lets you enter your password. The sudo's see that the process is already root-priveleged, so they don't request your password.

Alternatively, you can put your commands in a loop, this forces bash to wait for the first command to terminate before executing the second command:

```
for xx in 1
do
    sudo beep
    sudo beep
done
```

19 sudo: no tty present and no askpass program specified

If you try to use sudo in a Leo-Babel script, you may get this error message printed to stderr:

```
sudo: no tty present and no askpass program specified
```

On Ubuntu 16.04, I eliminated this error as follows:

1. I created /etc/sudo.conf containing:

   ```
   # Sudo askpass:
   #
   # An askpass helper program may be specified to provide a graphical
   # password prompt for "sudo -A" support.  Sudo does not ship with
   # its own askpass program but can use the OpenSSH askpass.
   #
   # Use the OpenSSH askpass
   #Path askpass /usr/X11R6/bin/ssh-askpass
   #
   # Use the Gnome OpenSSH askpass
   Path askpass /usr/bin/ssh-askpass
   ```

2. I installed ssh-askpass. You can install any one of the three packages that contain ssh-askpass.

20 Leo-Babel.pdf

You may find Leo-Babel.pdf helpful. Its table of contents allows jumping directly to any section listed in the table of contents. The contents are otherwise the same as the help displayed by Alt-P and then clicking “babel”. You can find Leo-Babel.pdf in the doc subdirectory of the directory in which Leo-Babel is installed on your system.

21 Examples of Leo-Babel Use

For examples of Leo-Babel use look in the examples subdirectory of the directory in which Leo-Babel is installed on your system.