Λlisue (lambdalisue)
#vimconf2018

Effective **Modern** Vim scripting

https://bit.ly/lambdalisue-vimconf-2018

About me

Λlisue (Alisue, 有末, ありすえ)

- Engineer at **Fixpoint, Inc.**
  - Frontend engineer (TypeScript, PostCSS)
  - Software engineer (Python 3, Go)
- Vim activities
  - Plugins (gina.vim, gista.vim)
  - Patch (patch-8.0.1361)
  - Others (jupyter-vim-binding)

About me

Λlisue (Alisue, 有末, ありすえ)

- Engineer at **Fixpoint, Inc.**
  - Frontend engineer (TypeScript, PostCSS)
  - Software engineer (Python 3, Go)
- Vim a[illegible]ies
  - Plugins (gina.vim, gista.vim)
  - Patch (patch-8.0.1361)
  - Others (jupyter-vim-binding)

Dark Vimmer

Dark Vimmer?

至 高 ノ 暗 黒 美 無

- Dark powered Vim plugins by the dark Vim maestro
  - deoplete.nvim, denite.nvim, dein.vim, etc...
- Tons of Vim plugins
  - I'm using more than 100 Vim plugins
- Use Vim because of Vim plugins
  - File operations? I use Shougo/vimfiler.vim
  - Refactoring? I use thinca/qfreplace
  - Git? I use lambdalisue/gina.vim

https://www.irasutoya.com/2017/10/blog-post_613.html

Fall into the dark side

欲 望 ヲ 解 キ 放 チ 漆 黒 ニ 染 マ レ

- There are tons of Vim plugins
  - More than 5,000 plugins in vim.org
  - More than 17,000 plugins in vimawesome.com
  - Potentially more plugins exist in github.com
- But there is **NO BEST** plugin for you
  - Everybody use Vim differently
  - Some plugins are too old
  - Some plugins are too new

https://www.irasutoya.com/2015/09/blog-post_477.html

https://www.irasutoya.com/2016/03/blog-post_46.html

Create your **own** plugin

https://bit.ly/lambdalisue-vimconf-2018

Purpose & Agenda

Purpose

Learn how to create a Vim plugin in modern way

Agenda

1. Hello World
   - Learn basics through a minimal Vim plugin
2. Synchronous script runner
   - Learn how to make a real plugin
3. Asynchronous script runner
   - Learn the modern way through rewriting

Hello World ●

Synchronous script runner ●

Asynchronous script runner ●

How to make a Vim plugin

- Create **plugin/{plugin}.vim**
  - Automatically sourced on Vim start-up
- Create **autoload/{plugin}.vim**
  - Add autoload functions
  - Automatically sourced when used
- Create other requirements
  - **doc/{plugin}.txt**
  - **README.md, LICENSE**
  - syntax, indent, after, etc...

Hello World

https://github.com/lambdalisue/vim-amake/tree/hello_world

- Add **~/vim-amake** to runtimepath
    - Add `set runtimepath+=~/vim-amake`
- Create **~/vim-amake** directory with
    - **plugin/amake.vim**
    - **autoload/amake.vim**

```
vim-amake/
  |-- plugin/
        |-- amake.vim
  |-- autoload/
        |-- amake.vim
```

```
$ echo "set runtimepath+=~/vim-amake" >> ~/.vimrc
$ mkdir ~/vim-amake && cd ~/vim-amake
$ mkdir plugin autoload
$ touch plugin/amake.vim autoload/amake.vim
```

Hello World

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#hello_world()
```

Hello World

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#hello_world()
```

Source guard
finish sourcing this file when **g:loaded_amake** exists

Hello World

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#hello_world()
```

Source guard
finish sourcing this file when **g:loaded_amake** exists

Command definition
Define **Amake** command which execute **amake#hello_world()** function (autoload function)

Hello World

autoload/amake.vim

```vim
function! amake#hello_world() abort
  echo "Hello World"
endfunction
```

Hello World

autoload/amake.vim

```
function! amake#hello_world() abort
  echo "Hello World"
endfunction
```

Autoload function definition
Autoload function **hoge** in **autoload/foo/bar.vim** become **foo#bar#hoge**.
This function **echo** "Hello World"

Hello World

autoload/amake.vim

```
function! amake#hello_world() abort
  echo "Hello World"
endfunction
```

Autoload function definition
Autoload function **hoge** in **autoload/foo/bar.vim** become **foo#bar#hoge**.
This function **echo** "Hello World"

Abort as soon as an error is detected
Vim does not abort function even an error is detected in default. The **abort** keyword change this behavior to abort the function on errors.

Hello World

autoload/amake.vim

```
function! amake#hello_world() abort
  echo "Hello World"
endfunction
```

Autoload function definition
Autoload function **hoge** in **autoload/foo/bar.vim** become **foo#bar#hoge**.
This function **echo** "Hello World"

abort as soon as an error is detected
Vim does not abort function even an error is detected in default. The **abort** keyword change this behavior to abort the function on errors.

```
:Amake
Hello World
```

Hello World ●

Synchronous script runner ●

Asynchronous script runner ●

Synchronous script runner

https://github.com/lambdalisue/vim-amake/tree/sync

- **:Amake** executes a script file synchronously
  - Execute an external program and **wait**
  - Open a new buffer with results
  - Inferior copy of thinca/vim-quickrun
- Steps
  - Function to invoke an external program
  - Function to create a runner of a particular filetype
  - Runner to build command to execute a script file
  - Function to open a new buffer with particular contents
  - Tie up all aboves together

Invoke an external program

autoload/amake/process.vim

```vim
function! amake#process#call(args) abort
  let args = map(
        \ a:args[:],
        \ { _, v -> shellescape(v) },
        \)
  let output = system(join(args))
  return split(output, '\r\?\n')
endfunction
```

Invoke an external program

autoload/amake/process.vim

```
function! amake#process#call(args) abort
  let args = map(
        \ a:args[:],
        \ { _, v -> shellescape(v) },
        \)
  let output = system(join(args))
  return split(output, '\r\?\n')
endfunction
```

Enclose items with single quotes
It encloses items of **a:args** with single quotes.
It is required because **system()** require a string.
["echo", "Hello World"] -> ["'echo'", "'Hello World'"]

Invoke an external program

autoload/amake/process.vim

```
function! amake#process#call(args) abort
  let args = map(
        \ a:args[:],
        \ { _, v -> shellescape(v) },
        \)
  let output = system(join(args))
  return split(output, '\r\?\n')
endfunction
```

Enclose items with single quotes
It encloses items of **a:args** with single quotes.
It is required because **system()** require a string.
["echo", "Hello World"] -> ["'echo'", "'Hello World'"]

Shallow copy of a list by slice
The **map()** modify a list inplace so create a shallow copy of a list by slice syntax.

Invoke an external program

autoload/amake/process.vim

```
function! amake#process#call(args) abort
  let args = map(
        \ a:args[:],
        \ { _, v -> shellescape(v) },
        \)
  let output = system(join(args))
  return split(output, '\r\?\n')
endfunction
```

Enclose items with single quotes
It encloses items of **a:args** with single quotes.
It is required because **system()** require a string.
["echo", "Hello World"] -> ["'echo'", "'Hello World'"]

Shallow copy of a list by slice
The **map()** modify a list inplace so create a shallow copy of a list by slice syntax.

Lambda function
Vim 8.0 introduced a lambda function syntax.
The **map()** pass key and value so use _ to indicate that we won't use key in the function.

Invoke an external program

autoload/amake/process.vim

```
function! amake#process#call(args) abort
  let args = map(
        \ a:args[:],
        \ { _, v -> shellescape(v) },
        \)
  let output = system(join(args))
  return split(output, '\r\?\n')
endfunction
```

Enclose items with single quotes
It encloses items of **a:args** with single quotes.
It is required because **system()** require a string.
["echo", "Hello World"] -> ["'echo'", "'Hello World'"]

Shallow copy of a list by slice
The **map()** modify a list inplace so create a shallow copy of a list by slice syntax.

Lambda function
Vim 8.0 introduced a lambda function syntax.
The **map()** pass key and value so use _ to indicate that we won't use key in the function.

```
:echo amake#process#call(['echo', 'Hello World'])
['Hello World']
```

Create a runner of a particular filetype

autoload/amake/runner.vim

```vim
function! amake#runner#new(filetype) abort
  let namespace = substitute(a:filetype, '\W', '_', 'g')
  let funcname = printf(
        \ 'amake#runner#%s#new',
        \ namespace,
        \)
  try
    return call(funcname, [])
  catch /:E117: [^:]\+: amake#runner#[^#]\+#new/
    throw printf(
          \ 'amake: Runner is not found: %s',
          \ a:filetype,
          \)
  endtry
endfunction
```

Create a runner of a particular filetype

autoload/amake/runner.vim

```vim
function! amake#runner#new(filetype) abort
  let namespace = substitute(a:filetype, '\W', '_', 'g')
  let funcname = printf(
        \ 'amake#runner#%s#new',
        \ namespace,
        \)
  try
    return call(funcname, [])
  catch /:E117: [^:]\+: amake#runner#[^#]\+#new/
    throw printf(
          \ 'amake: Runner is not found: %s',
          \ a:filetype,
          \)
  endtry
endfunction
```

Create an autoload function name
Replace non word characters to _ then use it as a namespace in **amake#runner#{namespace}#new**
e.g. 'foo-bar' -> amake#runner#foo_bar#new

Create a runner of a particular filetype

autoload/amake/runner.vim

```
function! amake#runner#new(filetype) abort
  let namespace = substitute(a:filetype, '\W', '_', 'g')
  let funcname = printf(
        \ 'amake#runner#%s#new',
        \ namespace,
        \)
  try
    return call(funcname, [])
  catch /:E117: [^:]\+: amake#runner#[^#]\+#new/
    throw printf(
          \ 'amake: Runner is not found: %s',
          \ a:filetype,
          \)
  endtry
endfunction
```

Create an autoload function name
Replace non word characters to _ then use it as a namespace in **amake#runner#{namespace}#new**
e.g. 'foo-bar' -> amake#runner#foo_bar#new

Catch E117 and re-throw
Vim throw **E117** with a function name so catch that error with a particular function name and re-throw a new error with user-friendly message.

Create a runner of a particular filetype

autoload/amake/runner.vim

```vim
function! amake#runner#new(filetype) abort
  let namespace = substitute(a:filetype, '\W', '_', 'g')
  let funcname = printf(
        \ 'amake#runner#%s#new',
        \ namespace,
        \)
  try
    return call(funcname, [])
  catch /:E117: [^:]\+: amake#runner#[^#]\+#new/
    throw printf(
          \ 'amake: Runner is not found: %s',
          \ a:filetype,
          \)
  endtry
endfunction
```

Create an autoload function name
Replace non word characters to _ then use it as a namespace in a**make#runner#{namespace}#new**
e.g. 'foo-bar' -> amake#runner#foo_bar#new

Catch E117 and re-throw
Vim throw **E117** with a function name so catch that error with a particular function name and re-throw a new error with user-friendly message.

```
:call amake#runner#new('vim')
E605: Exception not caught: amake: Runner is not found:
vim
```

Runners

autoload/amake/runner/vim.vim

```
function! amake#runner#vim#new() abort
  return { 'build_args': funcref('s:build_args') }
endfunction

function! s:build_args(filename) abort
  let cmd = printf(
        \ 'source %s',
        \ fnameescape(a:filename),
        \)
  return [
        \ 'nvim', '-n', '--headless',
        \ '--cmd', cmd, '--cmd', 'quit',
        \]
endfunction
```

Runners

autoload/amake/runner/vim.vim

```
function! amake#runner#vim#new() abort
  return { 'build_args': funcref('s:build_args') }
endfunction

function! s:build_args(filename) abort
  let cmd = printf(
        \ 'source %s',
        \ fnameescape(a:filename),
        \)
  return [
        \ 'nvim', '-n', '--headless',
        \ '--cmd', cmd, '--cmd', 'quit',
        \]
endfunction
```

Return a runner object
A runner object has **build_args** method which is a reference of the **s:build_args()**.

Runners

autoload/amake/runner/vim.vim

```vim
function! amake#runner#vim#new() abort
  return { 'build_args': funcref('s:build_args') }
endfunction

function! s:build_args(filename) abort
  let cmd = printf(
        \ 'source %s',
        \ fnameescape(a:filename),
        \)
  return [
        \ 'nvim', '-n', '--headless',
        \ '--cmd', cmd, '--cmd', 'quit',
        \]
endfunction
```

Return a runner object
A runner object has **build_args** method which is a reference of the **s:build_args()**.

Script local function
A function starts from **s:** is a script local function which is only available from the script. Like private function in other language.

Runners

autoload/amake/runner/vim.vim

```
function! amake#runner#vim#new() abort
  return { 'build_args': funcref('s:build_args') }
endfunction

function! s:build_args(filename) abort
  let cmd = printf(
        \ 'source %s',
        \ fnameescape(a:filename),
        \)
  return [
        \ 'nvim', '-n', '--headless',
        \ '--cmd', cmd, '--cmd', 'quit',
        \]
endfunction
```

Return a runner object
A runner object has **build_args** method which is a reference of the **s:build_args()**.

Script local function
A function starts from **s:** is a script local function which is only available from the script. Like private function in other language.

```
:echo amake#runner#new('vim')
{'build_args': function('<80><fd>R213_build_args') }
```

Runners

autoload/amake/runner/python.vim

```vim
function! amake#runner#python#new() abort
  return { 'build_args': { f -> ['python', f] } }
endfunction
```

autoload/amake/runner/javascript.vim

```vim
function! amake#runner#javascript#new() abort
  return { 'build_args': { f -> ['node', f] } }
endfunction
```

Runners

autoload/amake/runner/python.vim

```vim
function! amake#runner#python#new() abort
  return { 'build_args': { f -> ['python', f] } }
endfunction
```

autoload/amake/runner/javascript.vim

```vim
function! amake#runner#javascript#new() abort
  return { 'build_args': { f -> ['node', f] } }
endfunction
```

```
:echo amake#runner#new('python')
{'build_args': function('<lambda>6') }

:echo amake#runner#new('javascript')
{'build_args': function('<lambda>7') }
```

Invoke a runner

autoload/amake/runner.vim

```vim
function! amake#runner#run(runner, filename) abort
  let args = a:runner.build_args(a:filename)
  let output = amake#process#call(args)
  return {
        \ 'args': args,
        \ 'output': output,
        \}
endfunction
```

Invoke a runner

autoload/amake/runner.vim

```
function! amake#runner#run(runner, filename) abort
  let args = a:runner.build_args(a:filename)
  let output = amake#process#call(args)
  return {
        \ 'args': args,
        \ 'output': output,
        \}
endfunction
```

Build command arguments by a runner
Invoke **build_args** method of a runner to create command arguments to execute **a:filename**

Invoke a runner

autoload/amake/runner.vim

```
function! amake#runner#run(runner, filename) abort
  let args = a:runner.build_args(a:filename)
  let output = amake#process#call(args)
  return {
        \ 'args': args,
        \ 'output': output,
        \}
endfunction
```

Build command arguments by a runner
Invoke **build_args** method of a runner to create command arguments to execute **a:filename**

Invoke command arguments and get results
Invoke the **args** by a function previously created and get results as **output**

Invoke a runner

autoload/amake/runner.vim

```
function! amake#runner#run(runner, filename) abort
  let args = a:runner.build_args(a:filename)
  let output = amake#process#call(args)
  return {
        \ 'args': args,
        \ 'output': output,
        \}
endfunction
```

Build command arguments by a runner
Invoke **build_args** method of a runner to create command arguments to execute **a:filename**

Invoke command arguments and get results
Invoke the **args** by a function previously created and get results as **output**

Return a result object
Result object has **args** and **output** attribute

Invoke a runner

autoload/amake/runner.vim

```vim
function! amake#runner#run(runner, filename) abort
  let args = a:runner.build_args(a:filename)
  let output = amake#process#call(args)
  return {
        \ 'args': args,
        \ 'output': output,
        \}
endfunction
```

Build command arguments by a runner
Invoke **build_args** method of a runner to create command arguments to execute **a:filename**

Invoke command arguments and get results
Invoke the **args** by a function previously created and get results as **output**

Return a result object
Result object has **args** and **output** attribute

```vim
:let r = amake#runner#new('python')
:echo amake#runner#run(r, 'test.py')
{'args': ['python', 'test.py'], 'output': ['Hello World']}
```

Open a buffer

autoload/amake/buffer.vim

```vim
function! amake#buffer#new(bufname, content) abort
  execute 'new' fnameescape(a:bufname)
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
```

Open a buffer

autoload/amake/buffer.vim

```
function! amake#buffer#new(bufname, content) abort
  execute 'new' fnameescape(a:bufname)
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
```

Open a new buffer
Execute **new** command with correctly escaped **a:bufname** to open a new buffer

Open a buffer

autoload/amake/buffer.vim

```
function! amake#buffer#new(bufname, content) abort
  execute 'new' fnameescape(a:bufname)
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
```

Open a new buffer
Execute **new** command with correctly escaped **a:bufname** to open a new buffer

Replace contents of the buffer
Buffer may exist prior to the function call so **setlocal modifiable** and remove contents by **silent %delete _** before **setline()**. The _ is a black-hole register which is used to discard

Open a buffer

autoload/amake/buffer.vim

```
function! amake#buffer#new(bufname, content) abort
  execute 'new' fnameescape(a:bufname)
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
```

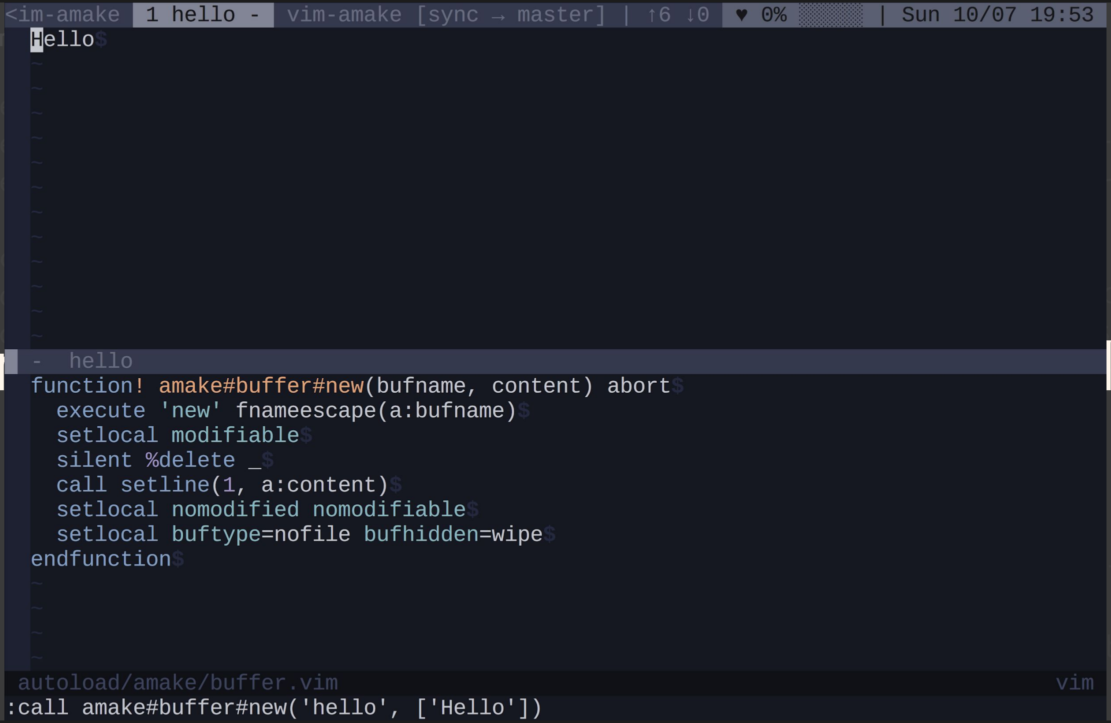
Open a new buffer
Execute **new** command with correctly escaped **a:bufname** to open a new buffer

Replace contents of the buffer
Buffer may exist prior to the function call so **setlocal modifiable** and remove contents by **silent %delete _** before **setline()**. The _ is a black-hole register which is used to discard

Configure local options

nomodified	Turn off modified flag
nomodifiable	Make the buffer non modifiable
buftype=nofile	Tell Vim that the buffer is not file
bufhidden=wipe	Wipeout the buffer when hidden

Open a buffer



```
<im-amake  1 hello -  vim-amake [sync → master] | ↑6 ↓0  ♥ 0%  | Sun 10/07 19:53
Hello
-  hello
function! amake#buffer#new(bufname, content) abort
  execute 'new' fnameescape(a:bufname)
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
autoload/amake/buffer.vim                                                    vim
:call amake#buffer#new('hello', ['Hello'])
```

Tie up

autoload/amake.vim

```vim
function! amake#run() abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output)
endfunction
```

plugin/amake.vim

```vim
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#run()
```

Tie up

autoload/amake.vim

```
function! amake#run() abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output)
endfunction
```

Create a runner of a current filetype
&filetype is a filetype of a current buffer

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#run()
```

Tie up

autoload/amake.vim

```
function! amake#run() abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output)
endfunction
```

Create a runner of a current filetype
&filetype is a filetype of a current buffer

Execute a current buffer with a runner
expand('%:p') is an absolute path of a current buffer

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#run()
```

Tie up

autoload/amake.vim

```
function! amake#run() abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output)
endfunction
```

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#run()
```

Create a runner of a current filetype
&filetype is a filetype of a current buffer

Execute a current buffer with a runner
expand('%:p') is an absolute path of a current buffer

Create an unique buffer name
Add **amake://** prefix and use **args** of **result** object to make an unique **bufname** of the command

Tie up

autoload/amake.vim

```
function! amake#run() abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output)
endfunction
```

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#run()
```

Create a runner of a current filetype
&filetype is a filetype of a current buffer

Execute a current buffer with a runner
expand('%:p') is an absolute path of a current buffer

Create an unique buffer name
Add **amake://** prefix and use **args** of **result** object to make an unique **bufname** of the command

Open a new buffer
Use an unique **bufname** and **output** of **result** object to open a new result buffer

Tie up

autoload/amake.vim

```
function! amake#run() abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output)
endfunction
```

plugin/amake.vim

```
if exists('g:loaded_amake')
  finish
endif
let g:loaded_amake = 1

command! Amake call amake#run()
```

Create a runner of a current filetype
&filetype is a filetype of a current buffer

Execute a current buffer with a runner
expand('%:p') is an absolute path of a current buffer

Create an unique buffer name
Add **amake://** prefix and use **args** of **result** object to make an unique **bufname** of the command

Open a new buffer
Use an unique **bufname** and **output** of **result** object to open a new result buffer

Replace Amake command
Invoke the **amake#run()** function in **Amake** command

Tie up

```
autoload/amake.vim

function! amake#run()
  let runner = amake#r
  let result = amake#r
  let bufname = printf
  call amake#buffer#ne
endfunction


plugin/amake.vim

if exists('g:loaded_am
  finish
endif
let g:loaded_amake = 1

command! Amake call am
```

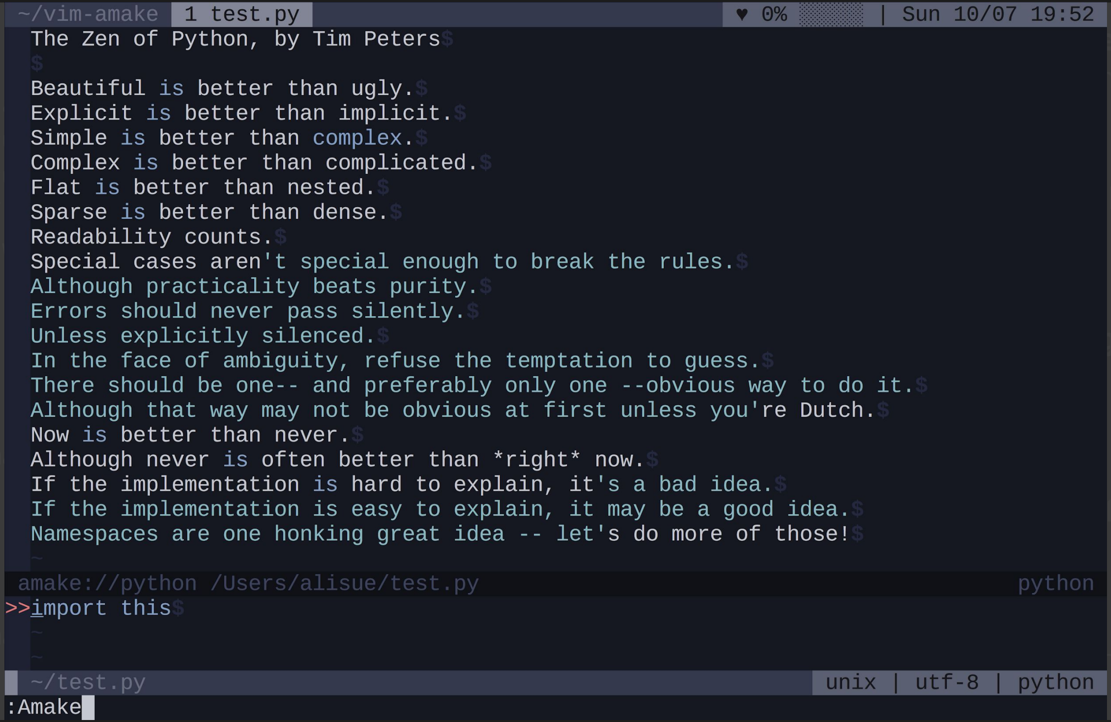
etype
t buffer

runner
h of a current buffer

gs of result object to
command

put of result object

n in Amake command



```
~/vim-amake  1 test.py                                          ♥ 0%     | Sun 10/07 19:52
  The Zen of Python, by Tim Peters$
  $
  Beautiful is better than ugly.$
  Explicit is better than implicit.$
  Simple is better than complex.$
  Complex is better than complicated.$
  Flat is better than nested.$
  Sparse is better than dense.$
  Readability counts.$
  Special cases aren't special enough to break the rules.$
  Although practicality beats purity.$
  Errors should never pass silently.$
  Unless explicitly silenced.$
  In the face of ambiguity, refuse the temptation to guess.$
  There should be one-- and preferably only one --obvious way to do it.$
  Although that way may not be obvious at first unless you're Dutch.$
  Now is better than never.$
  Although never is often better than *right* now.$
  If the implementation is hard to explain, it's a bad idea.$
  If the implementation is easy to explain, it may be a good idea.$
  Namespaces are one honking great idea -- let's do more of those!$
  ~
 amake://python /Users/alisue/test.py                                                  python
>>import this$
  ~
  ~
  ~/test.py                                                         unix | utf-8 | python
:Amake
```

- Hello World
- Synchronous script runner
- **Asynchronous script runner**

Asynchronous script runner

https://github.com/lambdalisue/vim-amake/tree/async

- **:Amake** executes a script file asynchronously
  - Execute an external program then return
  - Open a new buffer with results once the program terminated
  - Inferior copy of vim-quickrun with a job runner
- Steps
  - Learn Vital.vim, System.Job, and Async.Promise
  - Write a function to start an external program and return a Promise
  - Tie up all functions powered by Promise

Vital.vim

https://github.com/vim-jp/vital.vim

- Provides modern module system
  - Embed modules into a plugin
  - **:Vitalize . +{Module}** to install/update
- Provides tons of useful modules
  - DateTime
  - Random
  - HTTP client
  - etc...
- Most of modules are well tested
  - With vim-themis, a modern Vim testing framework

```vim
let s:DateTime = vital#vital#import('DateTime')

let utc = s:DateTime.timezone(0)
let dt1 = s:DateTime.now()
let dt2 = dt1.to(utc)
echo printf('NOW: %s', dt1.to_string())
echo printf('UTC: %s', dt2.to_string())
```

```
NOW: 2018-10-07 22:05:39 +0900
UTC: 2018-10-07 13:05:39 +0000
```

Vim.Buffer

https://github.com/vim-jp/vital.vim

- Official vital module
- Utility module for handling buffer
- Support Vim and Neovim
    - Support Vim 8.0.0027 or above
    - Support Neovim 0.2.0 or above

Vim.Buffer usage

```vim
let s:Buffer = vital#vital#import('Vim.Buffer')

" Open 'foo' with a default opener
call s:Buffer.open('foo')

" Open 'foo' with 'botright split ++enc=utf8 ++ff=dos'
call s:Buffer.open('foo', {
    \ 'opener': 'split',
    \ 'mods': 'botright',
    \ 'cmdarg': '++enc=utf8 ++ff=dos',
    \})
```

Use Vim.Buffer

1. Install **vim-jp/vital.vim** as a Vim plugin
2. Open Vim in **~/vim-amake**
3. Initialize vital.vim of vim-amake
   - **:Vitalize --name=amake .**
4. Tell vital.vim to bundle Vim.Buffer
   - **:Vitalize . +Vim.Buffer**
5. **Vim.Buffer** is embedded under **autoload/vital**
6. Dependencies of **Vim.Buffer** are embedded automatically

```
autoload/
 |-- amake/
 |-- vital/
   |-- _amake/
   |-- Data/
     |-- Dict.vim
     |-- List.vim
   |-- Vim/
     |--  Buffer.vim
     |--  Guard.vim
     |--  Type.vim
   |-- Prelude.vim
   |-- _amake.vim
   |-- amake.vim
   |-- amake.vital
 |-- amake.vim
```

Use Vim.Buffer

autoload/amake/buffer.vim

```
let s:Buffer = vital#amake#import('Vim.Buffer')

function! amake#buffer#new(bufname, content, opener) abort
  call s:Buffer.open(a:bufname, {
        \ 'opener': a:opener,
        \})
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
```

Use Vim.Buffer

autoload/amake/buffer.vim

```
let s:Buffer = vital#amake#import('Vim.Buffer')

function! amake#buffer#new(bufname, content, opener) abort
  call s:Buffer.open(a:bufname, {
        \ 'opener': a:opener,
        \})
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
```

Import Vim.Buffer
Use **vital#amake#import()** function to import a vital module and bind the module into a script local variable

Use Vim.Buffer

autoload/amake/buffer.vim

```
let s:Buffer = vital#amake#import('Vim.Buffer')

function! amake#buffer#new(bufname, content, opener) abort
  call s:Buffer.open(a:bufname, {
        \ 'opener': a:opener,
        \})
  setlocal modifiable
  silent %delete _
  call setline(1, a:content)
  setlocal nomodified nomodifiable
  setlocal buftype=nofile bufhidden=wipe
endfunction
```

Import Vim.Buffer
Use **vital#amake#import()** function to import a vital module and bind the module into a script local variable

Use Vim.Buffer.open to open a buffer
Vim.Buffer module provides **open** method to open a buffer. See **:help Vim.Buffer** for detail.

Use Vim.Buffer

autoload/amake.vim

```vim
function! amake#run(opener) abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output, a:opener)
endfunction
```

plugin/amake.vim

```vim
command! -nargs=? Amake call amake#run(<q-args>)
```

Use Vim.Buffer

autoload/amake.vim

```
function! amake#run(opener) abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output, a:opener)
endfunction
```

plugin/amake.vim

```
command! -nargs=? Amake call amake#run(<q-args>)
```

Allow opener argument
Use **opener** argument to switch the way to open a buffer

Use Vim.Buffer

autoload/amake.vim

```
function! amake#run(opener) abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  call amake#buffer#new(bufname, result.output, a:opener)
endfunction
```

Allow opener argument
Use **opener** argument to switch the way to open a buffer

plugin/amake.vim

```
command! -nargs=? Amake call amake#run(<q-args>)
```

Allow 0 or 1 argument in the command
-nargs=? allow 0 or 1 argument of the command and **<q-args>** is expanded to quoted arguments

Use Vim.Buffer

autoload/amake.vim

```
function! amake#run(opene
  let runner = amake#runr
  let result = amake#runr
  let bufname = printf('a
  call amake#buffer#new(b
endfunction
```

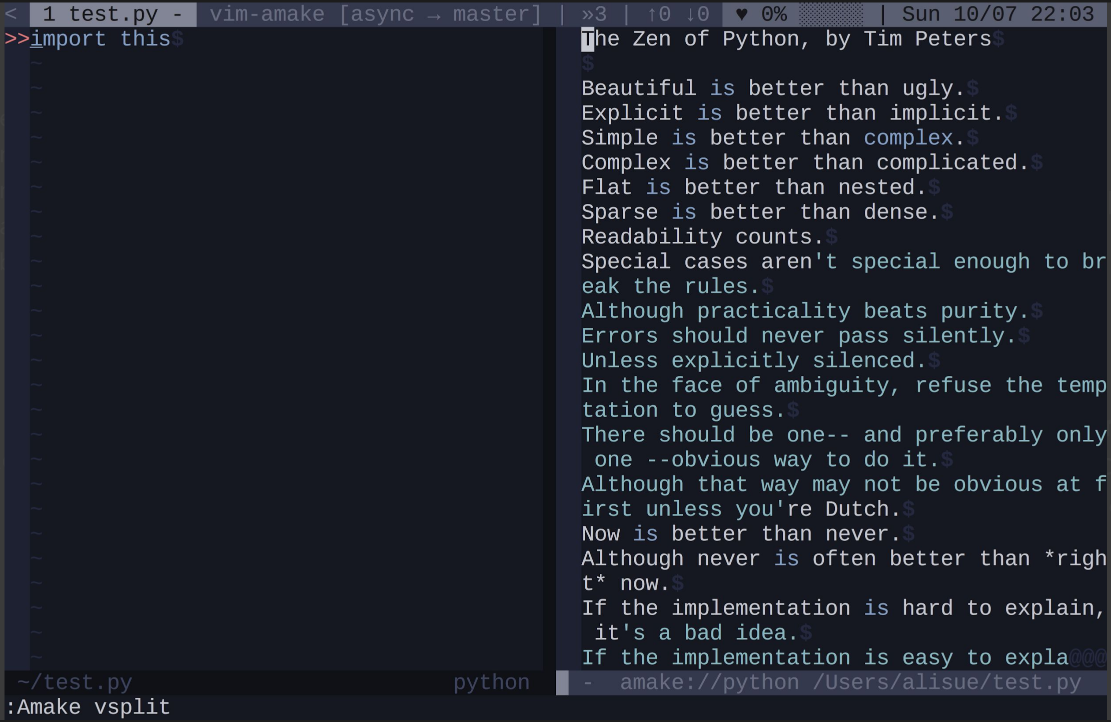
plugin/amake.vim

```
command! -nargs=? Amak
```

he way to open a

mmand

of the command and
arguments



```
<  1 test.py -  vim-amake [async → master] | »3 | ↑0 ↓0  ♥ 0%         | Sun 10/07 22:03
>>import this$                                  The Zen of Python, by Tim Peters$
~                                               $
~                                               Beautiful is better than ugly.$
~                                               Explicit is better than implicit.$
~                                               Simple is better than complex.$
~                                               Complex is better than complicated.$
~                                               Flat is better than nested.$
~                                               Sparse is better than dense.$
~                                               Readability counts.$
~                                               Special cases aren't special enough to br
~                                               eak the rules.$
~                                               Although practicality beats purity.$
~                                               Errors should never pass silently.$
~                                               Unless explicitly silenced.$
~                                               In the face of ambiguity, refuse the temp
~                                               tation to guess.$
~                                               There should be one-- and preferably only
~                                                one --obvious way to do it.$
~                                               Although that way may not be obvious at f
~                                               irst unless you're Dutch.$
~                                               Now is better than never.$
~                                               Although never is often better than *righ
~                                               t* now.$
~                                               If the implementation is hard to explain,
~                                                it's a bad idea.$
~                                               If the implementation is easy to expla@@@
 ~/test.py                         python      -  amake://python /Users/alisue/test.py
:Amake vsplit
```

Job on Vim and Neovim

Job in Vim 8

```
function! s:job_cb(rs, channel, msg) abort
  call add(a:rs, a:msg)
endfunction

let out = []
let exit = []
let job_options = {
      \ 'out_cb': funcref('s:job_cb', [out]),
      \ 'exit_cb': funcref('s:job_cb', [exit]),
      \}
let args = ['python', '-c', 'import this']
call job_start(args, job_options)
sleep 100m
echo printf('Exit: %d', exit[0])
echo join(out, "\n")
```

Job in Neovim

```
function! s:job_cb(job_id, data, event) abort dict
  if a:event ==# 'stdout'
    let self.stdout[-1] .= a:data[0]
    call extend(self.stdout, a:data[1:])
  else
    let self.exitval = a:data
  endif
endfunction

let job_options = {
      \ 'stdout': [''],
      \ 'on_stdout': funcref('s:job_cb'),
      \ 'on_exit': funcref('s:job_cb'),
      \}
let args = ['python', '-c', 'import this']
let job = jobstart(args, job_options)
call jobwait([job])
echo printf('Exit: %d', job_options.exitval)
echo join(job_options.stdout, "\n")
```

Job on Vim and Neovim

Job in Vim 8

```
function! s:job_cb(rs, channel, msg) abort
  call add(a:rs, a:msg)
endfunction

let out = []
let exit = []
let job_options = {
      \ 'out_cb': funcref('s:job_cb', [out]),
      \ 'exit_cb': funcref('s:job_cb', [exit]),
      \}
let args = ['python', '-c', 'import this']
call job_start(args, job_options)
sleep 100m
echo printf('Exit: %d', exit[0])
echo join(out, "\n")
```

Job in Neovim

```
function! s:job_cb(job_id, data, event) abort dict
  if a:event ==# 'stdout'
    let self.stdout[-1] .= a:data[0]
    call extend(self.stdout, a:data[1:])
  else
    let self.exitval = a:data
  endif
endfunction

let job_options = {
      \ 'stdout': [''],
      \ 'on_stdout': funcref('s:job_cb'),
      \ 'on_exit': funcref('s:job_cb'),
      \}
let args = ['python', '-c', 'import this']
let job = jobstart(args, job_options)
call jobwait([job])
echo printf('Exit: %d', job_options.exitval)
echo join(job_options.stdout, "\n")
```

Job on Vim and Neovim

Job in Vim 8

```
function! s:job_cb(rs, channel, msg) abort
  call add(a:rs, a:msg)
endfunction

let out = []
let exit = []
let job_options = {
      \ 'out_cb': funcref('s:job_cb', [out]),
      \ 'exit_cb': funcref('s:job_cb', [exit]),
      \}
let args = ['python', '-c', 'import this']
call job_start(args, job_options)
sleep 100m
echo printf('Exit: %d', exit[0])
echo join(out, "\n")
```

Job in Neovim

```
function! s:job_cb(job_id, data, event) abort dict
  if a:event ==# 'stdout'
    let self.stdout[-1] .= a:data[0]
    call extend(self.stdout, a:data[1:])
  else
    let self.exitval = a:data
  endif
endfunction

let job_options = {
      \ 'stdout': [''],
      \ 'on_stdout': funcref('s:job_cb'),
      \ 'on_exit': funcref('s:job_cb'),
      \}
let args = ['python', '-c', 'import this']
let job = jobstart(args, job_options)
call jobwait([job])
echo printf('Exit: %d', job_options.exitval)
echo join(job_options.stdout, "\n")
```

Job on Vim and Neovim

Job in Vim 8

```
function! s:job_cb(rs, channel, msg) abort
  call add(a:rs, a:msg)
endfunction

let out = []
let exit = []
let job_options = {
      \ 'out_cb': funcref('s:job_cb', [out]),
      \ 'exit_cb': funcref('s:job_cb', [exit]),
      \}
let args = ['python', '-c', 'import this']
call job_start(args, job_options)
sleep 100m
echo printf('Exit: %d', exit[0])
echo join(out, "\n")
```

Job in Neovim

```
function! s:job_cb(job_id, data, event) abort dict
  if a:event ==# 'stdout'
    let self.stdout[-1] .= a:data[0]
    call extend(self.stdout, a:data[1:])
  else
    let self.exitval = a:data
  endif
endfunction

let job_options = {
      \ 'stdout': [''],
      \ 'on_stdout': funcref('s:job_cb'),
      \ 'on_exit': funcref('s:job_cb'),
      \}
let args = ['python', '-c', 'import this']
let job = jobstart(args, job_options)
call jobwait([job])
echo printf('Exit: %d', job_options.exitval)
echo join(job_options.stdout, "\n")
```

Job on Vim and Neovim

Job in Vim 8

```
function! s:job_cb(rs, channel, msg) abort
  call add(a:rs, a:msg)
endfunction

let out = []
let exit = []
let job_options = {
      \ 'out_cb': funcref('s:job_cb', [out]),
      \ 'exit_cb': funcref('s:job_cb', [exit]),
      \}
let args = ['python', '-c', 'import this']
call job_start(args, job_options)
sleep 100m
echo printf('Exit: %d', exit[0])
echo join(out, "\n")
```

Job in Neovim

```
function! s:job_cb(job_id, data, event) abort dict
  if a:event ==# 'stdout'
    let self.stdout[-1] .= a:data[0]
    call extend(self.stdout, a:data[1:])
  else
    let self.exitval = a:data
  endif
endfunction

let job_options = {
      \ 'stdout': [''],
      \ 'on_stdout': funcref('s:job_cb'),
      \ 'on_exit': funcref('s:job_cb'),
      \}
let args = ['python', '-c', 'import this']
let job = jobstart(args, job_options)
call jobwait([job])
echo printf('Exit: %d', job_options.exitval)
echo join(job_options.stdout, "\n")
```

System.Job

https://github.com/lambdalisue/vital-Whisky

- External vital module
    - Non official vital module
- Support Vim and Neovim
    - Support Vim 8.0.0027 or above
    - Support Neovim 0.2.0 or above
- Tested in Windows/Linux/Mac
    - AppVeyor for Windows
    - Travis for Linux
    - Develop under Mac

Job with System.Job

```vim
function! s:on_stdout(data) abort dict
  let self.stdout[-1] .= a:data[0]
  call extend(self.stdout, a:data[1:])
endfunction

function! s:on_exit(data) abort dict
  let self.exitval = a:data
endfunction

let Job = vital#vital#import('System.Job')
let args = ['python', '-c', 'import this']
let job = Job.start(args, {
      \ 'stdout': [''],
      \ 'on_stdout': funcref('s:on_stdout'),
      \ 'on_exit': funcref('s:on_exit'),
      \})
call job.wait()
echo printf('Exit: %d', job.exitval)
echo join(job.stdout, "\n")
```

Async.Promise

https://github.com/vim-jp/vital.vim

- Official vital module
- Follows spec of ECMAScript
  - Promise.finally (ECMAScript)
  - Promise.wait (Original feature)
- Works on Vim and Neovim
  - Support Vim 8.0 or above
  - Works on Neovim 0.2.0 or above

Usage of Async.Promise

```vim
let s:Promise = vital#vital#import('Async.Promise')

function! s:executor(delay, resolve, reject) abort
  if float2nr(reltimefloat(reltime())) % 2 is# 0
    call timer_start(a:delay, { -> a:resolve() })
  else
    call timer_start(a:delay, { -> a:reject() })
  endif
endfunction

let timer = s:Promise.new(
      \ funcref('s:executor', [1000]),
      \)
call timer
      \.then({ -> execute('echo "Success"', '') })
      \.catch({ -> execute('echo "Fail"', '') })
```

Invoke a process asynchronously

autoload/amake/process.vim

```vim
let s:Job = vital#amake#import('System.Job')
let s:Promise = vital#amake#import('Async.Promise')

function! amake#process#open(args) abort
  return s:Promise.new(funcref('s:executor', [a:args]))
endfunction

function! s:executor(args, resolve, reject) abort
  let ns = {
        \ 'resolve': a:resolve, 'reject': a:reject,
        \ 'stdout': [''], 'stderr': [''],
        \}
  call s:Job.start(a:args, {
        \ 'on_stdout': funcref('s:on_receive', [ns.stdout]),
        \ 'on_stderr': funcref('s:on_receive', [ns.stderr]),
        \ 'on_exit': funcref('s:on_exit', [ns]),
        \})
endfunction
```

Invoke a process asynchronously

autoload/amake/process.vim

```
let s:Job = vital#amake#import('System.Job')
let s:Promise = vital#amake#import('Async.Promise')

function! amake#process#open(args) abort
  return s:Promise.new(funcref('s:executor', [a:args]))
endfunction

function! s:executor(args, resolve, reject) abort
  let ns = {
        \ 'resolve': a:resolve, 'reject': a:reject,
        \ 'stdout': [''], 'stderr': [''],
        \}
  call s:Job.start(a:args, {
        \ 'on_stdout': funcref('s:on_receive', [ns.stdout]),
        \ 'on_stderr': funcref('s:on_receive', [ns.stderr]),
        \ 'on_exit': funcref('s:on_exit', [ns]),
        \})
endfunction
```

Create a new Promise instance
Create a new Promise instance with **a:args** binded function of **s:executor**. **Async.Promise.new** calls the given function immediately

Invoke a process asynchronously

autoload/amake/process.vim

```
let s:Job = vital#amake#import('System.Job')
let s:Promise = vital#amake#import('Async.Promise')

function! amake#process#open(args) abort
  return s:Promise.new(funcref('s:executor', [a:args]))
endfunction

function! s:executor(args, resolve, reject) abort
  let ns = {
        \ 'resolve': a:resolve, 'reject': a:reject,
        \ 'stdout': [''], 'stderr': [''],
        \}
  call s:Job.start(a:args, {
        \ 'on_stdout': funcref('s:on_receive', [ns.stdout]),
        \ 'on_stderr': funcref('s:on_receive', [ns.stderr]),
        \ 'on_exit': funcref('s:on_exit', [ns]),
        \})
endfunction
```

Create a new Promise instance
Create a new Promise instance with **a:args** binded function of **s:executor**. **Async.Promise.new** calls the given function immediately

Create a namespace variable
Vim script does not have pointers so use a Dict to pass a reference of variables

Invoke a process asynchronously

autoload/amake/process.vim

```
let s:Job = vital#amake#import('System.Job')
let s:Promise = vital#amake#import('Async.Promise')

function! amake#process#open(args) abort
  return s:Promise.new(funcref('s:executor', [a:args]))
endfunction

function! s:executor(args, resolve, reject) abort
  let ns = {
        \ 'resolve': a:resolve, 'reject': a:reject,
        \ 'stdout': [''], 'stderr': [''],
        \}
  call s:Job.start(a:args, {
        \ 'on_stdout': funcref('s:on_receive', [ns.stdout]),
        \ 'on_stderr': funcref('s:on_receive', [ns.stderr]),
        \ 'on_exit': funcref('s:on_exit', [ns]),
        \})
endfunction
```

Create a new Promise instance
Create a new Promise instance with **a:args** binded function of **s:executor**. **Async.Promise.new** calls the given function immediately

Create a namespace variable
Vim script does not have pointers so use a Dict to pass a reference of variables

Start an external program
Call **System.Job.start()** to start an external program with given callbacks. **ns.stdout**, **ns.stderr**, and **ns** are bound to the each callbacks here

Invoke a process asynchronously

autoload/amake/process.vim

```vim
" ...continue from previous

function! s:on_receive(bs, data) abort
  let a:bs[-1] .= a:data[0]
  call extend(a:bs, a:data[1:])
endfunction

function! s:on_exit(ns, exitval) abort
  let data = {
        \ 'stdout': a:ns.stdout,
        \ 'stderr': a:ns.stderr,
        \ 'exitval': a:exitval,
        \}
  if a:exitval is# 0
    call a:ns.resolve(data)
  else
    call a:ns.reject(data)
  endif
endfunction
```

Invoke a process asynchronously

autoload/amake/process.vim

```
" ...continue from previous

function! s:on_receive(bs, data) abort
  let a:bs[-1] .= a:data[0]
  call extend(a:bs, a:data[1:])
endfunction

function! s:on_exit(ns, exitval) abort
  let data = {
        \ 'stdout': a:ns.stdout,
        \ 'stderr': a:ns.stderr,
        \ 'exitval': a:exitval,
        \}
  if a:exitval is# 0
    call a:ns.resolve(data)
  else
    call a:ns.reject(data)
  endif
endfunction
```

Extend newline split string list
System.Job uses Neovim way to receive data so extend given **data** as a newline split string list to the given **bs** (list variable)

Invoke a process asynchronously

autoload/amake/process.vim

```
" ...continue from previous

function! s:on_receive(bs, data) abort
  let a:bs[-1] .= a:data[0]
  call extend(a:bs, a:data[1:])
endfunction

function! s:on_exit(ns, exitval) abort
  let data = {
        \ 'stdout': a:ns.stdout,
        \ 'stderr': a:ns.stderr,
        \ 'exitval': a:exitval,
        \}
  if a:exitval is# 0
    call a:ns.resolve(data)
  else
    call a:ns.reject(data)
  endif
endfunction
```

Extend newline split string list
System.Job uses Neovim way to receive data so extend given **data** as a newline split string list to the given **bs** (list variable)

Create result data object
To resolve/reject with process result, create data object with **a:ns.stdout**, **a:ns.stderr**, and **a:exitval**

Invoke a process asynchronously

`autoload/amake/process.vim`

```vim
" ...continue from previous

function! s:on_receive(bs, data) abort
  let a:bs[-1] .= a:data[0]
  call extend(a:bs, a:data[1:])
endfunction

function! s:on_exit(ns, exitval) abort
  let data = {
        \ 'stdout': a:ns.stdout,
        \ 'stderr': a:ns.stderr,
        \ 'exitval': a:exitval,
        \}
  if a:exitval is# 0
    call a:ns.resolve(data)
  else
    call a:ns.reject(data)
  endif
endfunction
```

Extend newline split string list
System.Job uses Neovim way to receive data so extend given **data** as a newline split string list to the given **bs** (list variable)

Create result data object
To resolve/reject with process result, create data object with **a:ns.stdout**, **a:ns.stderr**, and **a:exitval**

Resolve/Reject the promise
Invoke **a:ns.resolve** or **a:ns.reject** to terminate the promise based on the **exitval** of the process with **data** object

Invoke a process asynchronously

```
:let p = amake#process#open(['echo', 'Hello'])
:call p.then({ v -> execute('echo v', '') })
  {'exitval': 0, 'stdout': ['Hello'], 'stderr': ['']}
```

autoload/amake/process.vim

```
" ...continue from previous

function! s:on_receive(bs, data) abort
  let a:bs[-1] .= a:data[0]
  call extend(a:bs, a:data[1:])
endfunction

function! s:on_exit(ns, exitval) abort
  let data = {
        \ 'stdout': a:ns.stdout,
        \ 'stderr': a:ns.stderr,
        \ 'exitval': a:exitval,
        \}
  if a:exitval is# 0
    call a:ns.resolve(data)
  else
    call a:ns.reject(data)
  endif
endfunction
```

Extend newline split string list
System.Job uses Neovim way to receive data so extend given **data** as a newline split string list to the given **bs** list variable

Create result data object
To resolve/reject with process result, create data object with **a:ns.stdout**, **a:ns.stderr**, and **a:exitval**

Resolve/Reject the promise
Invoke **a:ns.resolve** or **a:ns.reject** to terminate the promise based on the **exitval** of the process with **data** object

Tie up

autoload/amake/runner.vim

```
function! amake#runner#run(runner, filename) abort
  let args = a:runner.build_args(a:filename)
  let result = amake#process#open(args)
  let result.args = args
  return result
endfunction
```

Tie up

autoload/amake/runner.vim

```
function! amake#runner#run(runner, filename) abort
  let args = a:runner.build_args(a:filename)
  let result = amake#process#open(args)
  let result.args = args
  return result
endfunction
```

Return a promise with args

Return a promise object from **amake#process#open** with **args** attribute so that users can build a buffer name like previous version

Tie up

autoload/amake.vim

```vim
function! amake#run(opener) abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  let options = {
        \ 'opener': empty(a:opener) ? 'new' : a:opener,
        \}
  let Open = { c -> amake#buffer#new(bufname, c, options) }
  call result
        \.then({ v -> Open(v.stdout) })
        \.catch({ v -> Open(v.stdout + [''] + v.stderr) })
endfunction
```

Tie up

autoload/amake.vim

```
function! amake#run(opener) abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  let options = {
        \ 'opener': empty(a:opener) ? 'new' : a:opener,
        \}
  let Open = { c -> amake#buffer#new(bufname, c, options) }
  call result
        \.then({ v -> Open(v.stdout) })
        \.catch({ v -> Open(v.stdout + [''] + v.stderr) })
endfunction
```

Add callbacks for success/fail
Callback given to **then()** is called when the promise success and callback given to **catch()** is called when the promise fail. It opens a new buffer via **Open** with different contents

Tie up

autoload/amake.vim

```
function! amake#run(opener) abort
  let runner = amake#runner#new(&filetype)
  let result = amake#runner#run(runner, expand('%:p'))
  let bufname = printf('amake://%s', join(result.args, ' '))
  let options = {
        \ 'opener': empty(a:opener) ? 'new' : a:opener,
        \}
  let Open = { c -> amake#buffer#new(bufname, c, options) }
  call result[illegible]
        \.then({[illegible]->[illegible]pen([illegible]s[illegible]out)
        \.catch([illegible]v ->[illegible]Open([illegible]tdout[illegible]['[illegible]+ v.s[illegible]
endfunction
```

Create a temporary utility function
Open opens a new buffer with pre-constructed **buf**[illegible]**me** and gi[illegible] s/con[illegible]ts

Add callbacks for success/fail
Callback given to **then()** is called when the promise success and callback given to **catch()** is called when the promise fail. It opens a new buffer via **Open** with different contents

:Amake

Try it by yourself!

Step up

- Visit **https://github.com/lambdalisue/vim-amake**
  - MIT License
  - Fork it
- Visit **https://github.com/vim-jp/vital.vim**
  - Tons of useful vital modules you should know
- Visit **https://github.com/lambdalisue/vital-Whisky**
  - Useful vital modules for asynchronous programming

Take home message

Fall into
the dark side

欲 望 ヲ 解 キ 放 チ 漆 黒 ニ 染 マ ・ レ

https://bit.ly/lambdalisue-vimconf-2018