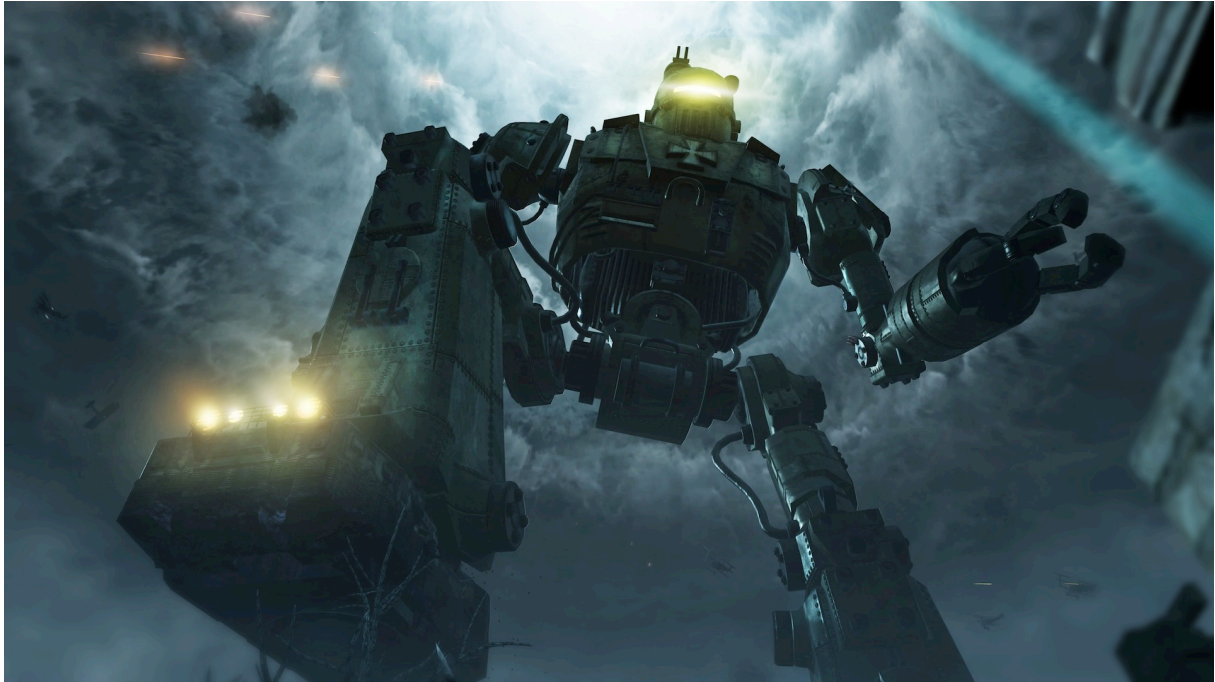


---

## Robo

build unknown

Simple YAML-based task runner written in Go.



## Features

- Not super slow
- Not super obscure
- No dependencies
- Variables
- Simple

## Installation

From gobinaries.com:

```
1 $ curl -sf https://gobinaries.com/tj/robo | sh
```

From source:

```
1 $ go get github.com/tj/robo
```

---

## Usage

Command-line usage.

## Listing tasks

Output tasks:

```
1 $ robo
2
3   aws - amazon web services cli
4   circle.open - open the repo in circle ci
5   events - send data to the "events" topic
6   push - push image from the current directory
```

## Task help

Output task help:

```
1 $ robo help events
2
3   Usage:
4
5       events [project-id] [rate]
6
7   Description:
8
9       send data to the "events" topic
10
11  Examples:
12
13      Send 25 events a second to gy2d
14      $ robo events gy2d 25
```

## Running tasks

Regardless of task type (shell, exec, script) any additional arguments given will be passed.

For example suppose you have the following task:

```
1 aws:
2   exec: ssh tools aws
```

You may then interact with the AWS cli as you would normally:

---

```
1 $ robo aws help
2 $ robo aws ec2 describe-instances
```

## Configuration

Task configuration.

## Commands

The most basic task simply runs a shell command:

```
1 hello:
2   summary: some task
3   command: echo world
```

You may also define multi-line commands with YAML's `|`:

```
1 hello:
2   summary: some task
3   command: |
4     echo hello
5     echo world
```

Commands are executed via `sh -c`, thus you may use shell features and positional variables, for example:

```
1 hello:
2   command: echo "Hello ${1:-there}"
```

Yields:

```
1 $ robo hello
2 Hello there
3
4 $ robo hello Tobi
5 Hello there Tobi
```

## Exec

The exec alternative lets you replace the robo image without the fork & shell, however note that shell features are not available (pipes, redirection, etc).

```
1 hello:
```

---

```
2  summary: some task
3  exec: echo hello
```

Any arguments given are simply appended.

## Scripts

Shell scripts may be used instead of inline commands:

```
1  hello:
2    summary: some task
3    script: path/to/script.sh
```

If the script is executable, it is invoked directly, this allows you to use `#!/`:

```
1  $ echo -e '#!/usr/bin/env ruby\nputs "yo"' > yo.rb
2  $ chmod +x yo.rb
3  $ cat > robo.yml
4  yo:
5    summary: yo from rb
6    script: yo.rb
7  ^C
8  $ robo yo
9  yo
```

Script paths are relative to the *config* file, not the working directory.

## Usage

Tasks may optionally specify usage parameters, which display upon help output:

```
1  events:
2    summary: send data to the "events" topic
3    exec: docker run -it events
4    usage: "[project-id] [rate]"
```

## Examples

Tasks may optionally specify any number of example commands, which display upon help output:

```
1  events:
2    summary: send data to the "events" topic
3    exec: docker run -it events
4    usage: "[project-id] [rate]"
5    examples:
```

---

```
6   - description: Send 25 events a second to gy2d
7     command: robo events gy2d 25
```

## Variables

Robo supports variables via the text/template package. All you have to do is define a map of `variables` and use `{{ }}` to refer to them.

Here's an example:

```
1 stage:
2   summary: Run commands against stage.
3   exec: ssh {{.hosts.stage}} -t robo
4
5 prod:
6   summary: Run commands against prod.
7   exec: ssh {{.hosts.prod}} -t robo
8
9 variables:
10  hosts:
11    prod: bastion-prod
12    stage: bastion-stage
```

The variables section does also interpolate itself with its own data via `{{ .var }}` and allows shell like command expressions via `$(echo true)` to be executed first providing the output result as a variable. Note that variables are interpolated first and then command expressions are evaluated. This will allow you to reduce repetitive variable definitions and declarations.

```
1 hash:
2   summary: echos the git {{ .branch }} branch's git hash
3   command: echo {{ .branch }} {{ .githash }}
4
5 variables:
6   branch: master
7   githash: $(git rev-parse --short {{ .branch }})
```

Along with your own custom variables, robo defines the following variables:

```
1 $ robo variables
2
3   robo.file: /Users/amir/dev/src/github.com/tj/robo/robo.yml
4   robo.path: /Users/amir/dev/src/github.com/tj/robo
5
6   user.home: /Users/amir
7   user.name: Amir Abushareb
8   user.username: amir
```

---

## Environment

Tasks may define `env` key with an array of environment variables, this allows you to re-use robo configuration files, for example:

```
1 // aws.yml
2 dev:
3   summary: AWS commands in dev environment
4   exec: aws
5   env: ["AWS_PROFILE=eng-dev"]
6
7 stage:
8   summary: AWS commands in stage environment
9   exec: aws
10  env: ["AWS_PROFILE=eng-stage"]
11
12 prod:
13   summary: AWS commands in prod environment
14   exec: aws
15   env: ["AWS_PROFILE=eng-prod"]
```

You can also override environment variables:

```
1 $ cat > robo.yml
2 home:
3   summary: overrides $HOME
4   exec: echo $HOME
5   env: ["HOME=/tmp"]
6 ^C
7 $ robo home // => /tmp
```

Variables can also be used to set env vars.

```
1 $ cat > robo.yml
2 aws-stage:
3   summary: AWS stage
4   exec: aws
5   env: ["AWS_PROFILE={{.aws.profile}}"]
6 variables:
7   aws:
8     profile: eng-stage
9 ^C
10 $ robo aws-stage ...
```

Note that you cannot use shell features in the environment key.

---

## Setup / Cleanup

Some tasks or even your entire robo configuration may require steps upfront for setup or afterwards for a cleanup. The keywords `before` and `after` can be embedded into a task or into the overall robo configuration. It has the same executable syntax as a task: `script`, `exec` and `command`. Defining it on a task level causes the steps to be executed before (respectively after) the task. Global before or after steps are invoked for every task in the configuration. All steps get interpolated the same way tasks and variables are interpolated.

```
1 before:
2   - command: echo "global before {{ .foo }}"
3 after:
4   - script: /global/after-script.sh
5
6 foo:
7   before:
8     - command: echo "local before {{ .foo }}"
9     - exec: git pull -r
10  after:
11    - command: echo "local after"
12    - exec: git reset --hard HEAD
13  exec: git status
14
15 variables:
16   foo: bar
```

## Templates

Task `list` and `help` output may be re-configured, for example if you prefer to view usage information instead of the summary:

```
1 templates:
2   list: |
3     {{range .Tasks}}  {{cyan .Name}} - {{.Usage}}
4     {{end}}
```

Or perhaps something more verbose:

```
1 templates:
2   list: |
3     {{range .Tasks}}
4       name: {{cyan .Name}}
5       summary: {{.Summary}}
6       usage: {{.Usage}}
7     {{end}}
```

---

## Global tasks

By default `./robo.yml` is loaded, however if you want global tasks you can simply alias to something like:

```
1 alias segment='robo --config ~/.robo.yml'
```

## Robo chaining

You can easily use Robo to chain Robo, which is useful for multi-environment setups. For example:

```
1 prod:
2   summary: production tasks
3   exec: robo --config production.yml
4
5 stage:
6   summary: stage tasks
7   exec: robo --config stage.yml
```

Or on remote boxes:

```
1 prod:
2   summary: production tasks
3   exec: ssh prod-tools -t robo --config production.yml
4
5 stage:
6   summary: stage tasks
7   exec: ssh stage-tools -t robo --config stage.yml
```

You can also use robo's builtin variables `robo.path`, for example if you put all robofiles in together:

```
1 |
2 dev.yml |
3 prod.yml |
4 root.yml |
5 stage.yml
```

And you would like to call `dev`, `prod` and `stage` from `root`:

```
1 dev:
2   summary: Development commands
3   exec: robo --config {{ .robo.path }}/dev.yml
4
5 stage:
6   ...
```

---

## Composition

You can compose multiple commands into a single command by utilizing robo's built-in `robo.file` variable:

```
1 one:
2   summary: echo one
3   command: echo one
4
5 two:
6   summary: echo two
7   command: echo two
8
9 all:
10  summary: echo one two
11  command: |
12    robo -c {{ .robo.file }} one
13    robo -c {{ .robo.file }} two
```

```
1 $ robo all
2 one
3 two
```

## Why?

We generally use Makefiles for project specific tasks, however the discoverability of global tasks within a large team is difficult unless there's good support for self-documentation, which Make is bad at.

I'm aware of the million other solutions (Sake, Thor, etc) but I just wanted something fast without dependencies.

## License

MIT