Taskfile: A Tasklist Compiler

Release

Sam Kleinman

November 11, 2012

Contents

Taskfile is aggregates task information from text files using GNU Make. Taskfile supports task tracking for a diverse collection of workflows and tools.

1 Contents

1.1 Working with Taskfile

Synopsis

Above all, Taskfile aims to provide a means to create task lists with as little conceptual overhead as possible. As long as you can find a way to include unique markers to identify your tasks and your work exists in "`grep`"-able text format, you should be able to use Taskfile without much modification. This document discusses usage patterns how to integrate Taskfile into practical workflows.

Patterns

Once you've *configured* Taskfile, running the `make` command in the directory where the makefile lives is usually all you need to do. The makefile provides a "`make todo`" operation that prints the task list, or you can view the output using an interface like Geektool, Ikiwki or preferred text editor. Ikiwiki, and Emacs' ' Markdown Mode provide linking ability ability that facilitates "moving backwards" from the task on the "todo" output to the embeded task in a file.

There are two major approaches to organizing a Taskfile system:

1. Do work in files and insert task items that Taskfile (i.e. `grep`) will pickup. Embedding tasks is simple and there's no real downside, though you can end up with task items in inopportune places if you're careless.
2. Maintain a collection of project-specific "tracking" or notes files for task planning that contain a few notes and some "`TODO`" option.

Both modalities are equivalent and Taskfile doesn't "prefer" one over the other. There is no need to work in a "pure" system: you can mix "embeded tasks" within a notes file, or just use "tracking files" that are *only* tasks.

Taskfile includes the source location of the file that contains the task item. Thus, the real implication of the embeded versus tracking is that sometimes tracking files make it difficult to trace back from the tasklist to the place where you need to do work. In practice, combination of both modes often proves most optimal.

For day-to-day and moment-to-moment work you may choose to keep a view of the of your task list or task lists open at all times either in an editor window or as a rendered page in a browser, and then use this to either jump to the relevant file to begin work. Many text editors have file searching functionality that makes it possible to find all references of a string within a file, [1] that you may find helpful.

Components

Projects

In the context of Taskfile, a project represents a class of non-overlapping tasks that Taskfile will aggregate into separate lists. In other project management systems, projects often refer to smaller groupings of tasks, where a project might be a document, or a release of a piece of software, or some other logical grouping. In Taskfile, you may use projects to filter the tasks that you're currently working on with from tasks marked "future" or "frozen." Conversely, you may use project separation to separate tasks for personal and side projects from your works projects based on keyword or source file path.

The goal of Tasklist is to provide an aggregate view into all of your tasks so that you'll be able to see at a glance what tasks require your attention without relying on your memory to remember tasks. Usually this means "make all tasks

[1] In emacs this is "occur" mode. TextMate also has or had a "TODO" mode that performs a similar function.

visible all the time," but it sometimes makes sense to seperate some conceptually distinct tasks. Used judiciously, projects are great for keeping things organized.

There are two features of the way Taskfile handles projects that are worth noting:

1. Projects make it difficult to track and follow tasks back to their original location in the source file. Sometimes this doesn't matter.
2. Project separation is easy to configure, but requires some manual intervention in the makefile itself. See "*Setup and Configuration*" and "*Taskfile Internals*" for more information.

Keywords

Keywords are unique strings of characters that identify tasks. There are no formal limitation on what can be a keyword, but they should be distinct, the default behavior is for keywords to be case sensitive but this is simple to disable. The default keywords are:

- `TODO`
- `FIXME`
- `EDIT`
- `ONGOING`
- `FUTURE`
- `FROZEN`

Specify these keywords in regular expression syntax, [2] and keywords can be quite specific, both in terms of the letters used and their position in the line. The best keywords contain characters that are unlikely to appear naturally in the source text files. Case sensitivity helps reduce collisions, but certain letter combinations are incredibly uncommon in some languages (i.e. "`tk`" and "`q`" followed by most letters.)

You may also consider requiring that your keywords the keyword appear at the beginning of a line (which is in the default makefile.) You may also wish to require that your TODO item take the form of a comment in whatever syntax your textfiles are in.

Integration

The "`make todo`" output of the task list is good for most rudimentary tasklist viewing; however, more serious operations may require a more interactive tasklist. In most cases, whatever tools you use to edit your source files work fine with the output of Taskfile, and it's easy to modify some common tools to provide support for the taskfile output.

Ikiwiki provided the initial inspiration for, and hosting of, Taskfile and the default configuration maintains compatibility with this approach. Just make sure that Taskfile's output has an extension that Ikiwiki can parse and ends up in a location that Ikiwiki will build.

If you use Emacs, the following modes and features may be useful for interacting with Taskfile:

- Markdown Mode

  The latest versions of markdown mode, include an automatic wiki-link following feature that allows you to travel from the current file to the linked file within the file by overloading the "Enter" key.

- Occur Mode

[2] Unsurprisingly, perhaps, Taskfile uses "`grep -E`" to find and filter tasks from the source files.

Recent versions of Emacs include occur, which and searches and indexes textfiles. Use `occur` within the source files, to find instances of keywords within a file. Occur cites line numbers and makes it easy to jump to specific line numbers.

- Auto-Revert Mode or Revbufs

  Because Taskfile generates the todo files outside of Emacs, use a system like auto revert mode or `revbufs` to get emacs to refresh the buffer from the disk when you update.

- Compile Mode

  Emacs includes compile mode that provides an easy method to run, rerun and monitor make and make-like processes within emacs.

Note: Most text editors contain some or all of these features, with different interfaces and names. If you use another text editor, consider *contributing* documentation to Taskfile to explain these functions and possible configurations.

Internal Approach

Taskfile operates by scanning a directory tree for files that contain or begin with a `TODO` *keywords* and copying *only* those `TODO` lines to a "cache." Todo lists are then built from this mirrored "cache tree." GNU Make's dependency checking allows Taskfile to only scan or rebuild those files when the source files change.

Depending on the number of files and the number of lines in the file, the initial creation of a crash can take several seconds; however, incremental rebuilds of the list should complete in less than a second.

Taskfile's predecessor was a simple shell script. Although this script used many of same basic operations it had no dependency checking and had to aggregate all of the data on every run, was more difficult to customize, and was not a feasible solution for checking projects with large numbers of files or a large amount of data.

The primary limitation of Taskfile at present is the fact that many deployments will require some duplication of the Taskfile makefile to track different project trees create different outputs. While the duplication is a concern, the fact that users must tweak and maintain unique makefiles is a larger concern. Future distributions of Taskfile will include a "meta-maker" that will guide some Taskfile customization.

See Also:

"*Taskfile Internals*"

1.2 Setup and Configuration

This document addresses basic setup and configuration of Taskfile. For more in-depth information on using taskfile for day-to-day see "*Taskfile Internals*" for descriptions of all components within Taskfile.

Installing

To install Taskfile:

1. Clone the taskfile repository. For example:

   ```
   git clone http://cyborginstitute.com/git/taskfile.git
   ```

2. Copy the primary `taskfile.make` into your default notes directory where you keep your text files [3] (e.g. "`~/notes`") and rename it as needed. For example:

   ```
   cp taskfile/taskfile.make ~/notes/makefile
   ```

3. (Optional.) Copy the `taskfile.project` to another project folder (e.g. "`~/projects/`") and rename it as needed. For example:

   ```
   cp taskfile/taskfile.proejct ~/projects/makefile
   ```

Congratulations! You have now installed Taskfile. Continue reading, and consult "*Taskfile Internals*" for more information regarding your Taskfile system.

Configuration

The default configuration is sufficient for most basic uses: if all the project files you use are in a single folder, and you do not need to generate multiple lists, you may only have to modify the following variables at the beginning of the makefile, which control the input and output of the taskfile build system:

- `EXTENSION`,
- `SOURCE`,

Including whitespace and comments, the makefile is under 200 lines, and only 20-30 lines, or so, are relevant to the operation of Taskfile. Complete documentation of all taskfile elements is in the *internals* page.

Customizing

There are many possible customizations. However, there are two major classes of customization for Taskfile systems:

1. Creating different aggregation selections and outputs, to separate work domains that don't share any contextual overlap.

   Currently, by default, Taskfile supports a single `todo.mdwn` output. However, there are two additional outputs possible in the default taskfile, if you un-comment and modify several lines. Using these targets and variables as an example you can create any number of unique aggregations.

   Consider the following: In addition to adding targets to build a secondary tasklist, you must also ensure that those items on that secondary items do not end up on your primary list (unless you want them to.)

2. Modifying the output of the taskfile output to enhance capability with your own preferred text file editing system.

   Currently, Taskfile produces Markdown output that allows for a double square bracket "wiki link" syntax (i.e. `[[link]]`) to the page that contains the original source of the task item. Modify the transformation in the `sed` expression in the end of the `$(OUTPUT)` target. Alternatively, you can add an additional expression (e.g. "`-e s/^TODO/TASK/`") to the end of this statement (before "`| sort -u >> $@`".)

Extending

Because Taskfile is just a makefile, and a reasonably simple makefile at that, there are a number of options and directions that you may chose to take if you want to extend Taskfile. This section contains a list of possible extensions and enhancements to Taskfile:

[3] Because Taskfile recurses into all sub-directories, you can place the makefile anywhere, as long as you have tasks below this point in the hierarchy. At the same time you may want to chose carefully because Taskfile must scan (and create a mirror every directory (and potentially every file) within this directory tree every time you refresh your list. Use symbolic links and groups of folders to limit your taskfile as needed.

- **Additional output formats:**

  Make exists to generate output according to custom specifications, so it's trivial to add new output formats to a makefile, assuming you have generic converters. Consider the following "extension," which uses Multi-Markdown to convert the standard markdown output of Taskfile to PDF.

```
taskfile: [...] $(OUTPUT_FILE_NAME).pdf

$(OUTPUT_FILE_NAME).tex:$(OUTPUT)
      mmd2LaTeX.pl $<
$(OUTPUT_FILE_NAME).pdf:$(OUTPUT_FILE_NAME).tex
      pdflatex $(OUTPUT_FILE_NAME).tex
```

- **Integrate into emacs** (*or other text editor:*)

  There are a number of functions and keybindings in the `taskfile.el` that you may find helpful. These functions make it possible for you to:

  - Regenerate your taskfile inside of Emacs, using `compilation-mode`.
  - Change a task state to "`DONE`"
  - Open the tasklist (i.e. `OUTPUT`) from a key binding.
  - Open a "flow" buffer for ad hoc tasks.

  See Also:

  Integrating Taskfile with Emacs

- **Git Integration:**

  Run Taskfile as part of a `pre-commit` hook to update the taskfile before committing the repository.

  Conversely, you may want to exclude your Taskfile output from version control because it's always possible to generate the taskfile.

- **Scheduling with Cron:**

  Because re-generating the Taskfile output is efficient, it's safe to run as a cron task.

While the initial distribution of Taskfile should be as simple and "base" as possible, we can include any good and appropriately licensed extension in the default distribution. See "*Contribute to Taskfile* for more.

1.3 Taskfile Internals

This document describes the internal operation and components of the Taskfile makefile. Use this document as a reference in support of the higher level discussion of *Taskfile customization*.

Variables

`MAKEFLAGS`
: Defines the default flags for **make**. For Taskfile, this is "`-j4`" but you can use more or less depending on your system.

Location and Output Variables

`SOURCE`
: Specifies the top level of the file system tree where all files that *might* include task items. Consider using a path

like "~/wiki", "~/deft", "~/notes", "~/writing" or wherever your working text and project planning files live.

CACHE
The directory where Taskfile stores the cached extraction of task items from the SOURCE files. Ensure that the Taskfile cache is not tracked in version control. Consider using ".git/taskfile-cache" or place the CACHE path in your .gitignore or other similar exclusion file.

OUTPUT_FILE_NAME
Typically this is "todo" and represents the name of the file where Taskfile writes it output.

This variable does not include the file extension.

EXTENSION
Specifies the file extension for the of the files that contain the task list. This extension is also added. to the file name of the output file, and should not contain a "." character.

Taskfile can only look for todo items with one extension, while this might present as a limitation, it ensures that Taskfile will only scan files that might have task items.

OUTPUT
The full path, including file name, of the file where Taskfile will write the tasklist.

The default value for this variable is "$(SOURCE)/$(OUTPUT_FILENAME).$(EXTENSION)" and Taskfile will set this value if you do not set another value.

EXTRA_OUTPUT_DIR
The default value for this variable is $(SOURCE)/$(OUTPUT_FILENANE), although you can override this value.

PROJECTS_OUTPUT
Defines a target to build for project specific output.

Typically a distinct makefile (e.g. a separate Taskfile instance) builds these projects specific lists. This and PROJECTS_MAKEFILE simply provide pointers to the other Taskfile instance to provide a centralized user interface.

PROJECTS_MAKECONTEXT
The path to a "projects" taksfile's directory context. Called as "make -C $(PROJECTS_MAKECONTEXT)". This line is, however, commented out in the distributed version of taskfile.

Patterns and Filters

KEYWORDS
Defines the search pattern for a grep command that finds the items Taskfile aggregates as "task items." Taskfile calls this grep in the following form, with the KEYWORDS variable:

```
grep -E "($(KEYWORDS)).*"
```

The default expression is:

```
^TODO|^DEV|^FIXME|^WRITE|^EDIT|^FUTURE|^FROZEN|WORK
```

Modify the expression as needed.

Note: Taskfile sorts the output of grep when making the task list, which may impact how your organize your Taskfile query. Furthermore, regular expressions that anchor the search pattern (i.e. "^" for the beginning of the line, or "$" for the end of the line,) often show better performance, depending on the structure of your search content.

FUTURE_FILTER
Holds a name for the "future" filter, used to create a secondary list of non- or less-actionable items.

Note: The default distribution of tasklist has no enabled targets that build the FUTURE_FILTER task lists. If you want to build a future task list, you will need to un-comment these sections. See the "*Setup and Configuration*" document for more information on this process.

FUTURE_KEYWORDS
Holds a list of keywords, passed as in "KEYWORDS" to `grep`. Used to

---

Note: There are no active targets that build the "future" versions of the task list, as future task lists require some measure of customization. See the "*Setup and Configuration*" document for more information on this process.

However, items that use one of these keywords are not included in the primary tasklist.

WORK_FILTER
As FUTURE_FILTER, WORK_FILTER makes it possible to build *tasklists* from work-based tasks. Whereas the $(FUTURE_OUTPUT) (using FUTURE_FILTER) builds a separate tasklist, based on a special *keywords*, $(WORK_OUTPUT) using WORK_FILTER creates a separate tasklist by filtering out some items based on their location in SOURCE.

Note: There are no active targets that build "work" tasklists in default taskfile, because work tasklists require some measure of customization. However, no items in locations that include the WORK_FILTER term will appear in the primary taskfile.

See Also:

$(WORK_OUTPUT)

Consider the "*Setup and Configuration*" document for instructions regarding configuring the work tasklists.

Project Variables

FUTURE_OUTPUT
Defines the location for the "future" task list. In the default distribution of Tasklist, this variable has a value of:

```
$(EXTRA_OUTPUT_DIR)/$(FUTURE_FILTER).$(EXTENSION)
```

In most cases you will not need or want to modify this value. This *variable* expands to a path of "`~/wiki/todo/future.mdwn`" given the default configuration.

See Also:

The following variables, for documentation of the default value of this *variables*:

- EXTRA_OUTPUT_DIR
- FUTURE_FILTER
- EXTENSION

Additionally, consider the commented $(FUTURE_OUTPUT) target in the default distribution for an idea of the *future tasklist's* implementation.

Note: There are no active targets that build the "future" versions of the task list, as future task lists require some measure of customization. See the "*Setup and Configuration*" document for more information on this process.

WORK_OUTPUT
Defines the location for the file where taskfile writes the *work tasklist*. The default value for this variable is as follows:

```
$(SOURCE)/$(WORK_FILTER)/$(OUTPUT_FILE_NAME).$(EXTENSION)
```

In most cases you will not need or want to modify this value. Given the default values, this expands to "`~/wiki/work/todo.mdwn`" in the default configuration.

See Also:

The following variables, for documentation of the default values for these *variables*:

- `SOURCE`
- `WORK_FILTER`
- `OUTPUT_FILE_NAME`
- `EXTENSION`

Additionally, consider the commented `$(WORK_OUTPUT)` target in the default distribution for an idea of the *work tasklist's* implementation.

Note: There are no active targets that build the "future" versions of the task list, as future task lists require some measure of customization. See the "*Setup and Configuration*" document for more information on this process.

EXTRA_OUTPUT_DIR
Defines a directory within the `SOURCE` directory that holds. additional outputs and dependent files. The `$(FUTURE_OUTPUT)` builds into this directory, and a number of "template" files are in this directory. The `SOURCES` does not include items from this directory.

In the default configuration `EXTRA_OUTPUT_DIR` has the following value:

```
$(SOURCE)/$(OUTPUT_FILENANE)
```

If this variable isn't defined in the beginning section of the taskfile, Taskfile will provide a default.

NAME
This *variable* only appers in the projects taskfile. This value forms the basis of the projects-specific taskfile output, and contributes to several other variables.

OUTPUT_DIR
This *variable* only appers in the projects taskfile.

Default Variables

Taskfile supplies default values for the following values that are necessary for Taskfile operation, if you do not define custom values at the beginning of the file.

OUTPUT
Unless set at the beginning of the file, the value of `OUTPUT` is "`$(SOURCE)/$(OUTPUT_FILENAME).$(EXTENSION)`".

See Also:

`OUTPUT` and thehe following variables that affect the value of `OUTPUT` in this default configuration:

- SOURCE
- OUTPUT_FILENAME
- EXTENSION

EXTRA_OUTPUT_DIR

Unless set at the beginning of the file, the value of EXTRA_OUTPUT_DIR is "$(SOURCE)/$(OUTPUT_FILENANE)".

See Also:

EXTRA_OUTPUT_DIR and the following variables that affect the value of EXTRA_OUTPUT_DIR in the default configuration:

- SOURCE
- OUTPUT_FILENAME

TMPL_DIR

The TMPL_DIR *variable* only appers in the project-specific default taskfile. In the default setting this path should match EXTRA_OUTPUT_DIR in the main Taskfile.

OUTPUT_FILE_NAME

Unless set at the beginning of the file, the value of OUTPUT_FILE_NAME is "todo"

See Also:

EXTRA_OUTPUT_DIR.

Computed Variables

The following variables use computed forms to generate lists or functions which underpin the operation of the targets that produce the tasklist.

SOURCES

Generates a list files that end with the EXTENSION. Excludes the output filneame and some temporary files. Taskfile computes SOURCES using the find command and filters the results with grep. The value of this variable is:

```
$(shell find $(SOURCE) -name "*$(EXTENSION)" -not \( -name ".\#*" \) | grep -v "$(OUTPUT_FILE_NA
```

SOURCEDIR

Returns a list of all directories, with recursive resolution that may contain source files. SOURCEDIR only appears in the CACHE_DIRS variable. It has the following value:

```
$(shell find $(SOURCE) -type d -not \( -name ".*" -prune \) -not \( -name "$(OUTPUT_FILE_NAME)"
```

CACHE_DIRS

Using GNU Make's string substitution function, CACHE_DIRS generates a list of directories but substitutes the path of the top level SOURCE directory for the name of the CACHE directory in the value of SOURCEDIR. The actual value as specified is:

```
$(subst $(SOURCE),$(CACHE),$(SOURCEDIR))
```

This variable ensures Taskfile creates all required directories in the task cache before attempting to write files.

CACHE_INDEX_FILES

Using a nested string substitution, CACHE_INDEX_FILES replaces CACHE with SOURCE, and ".:var:*EXTENSION*" with ".:var:*OUTPUT_FILE_NAME*" for all of the files in the SOURCES directory that have end with the EXTENSION. For instance, given the default configuration and a file in SOURCES such

as "~/wiki/shopping.mdwn", this will become ".git/tasklist-build/shopping.todo". The code itself is:

```
$(subst $(SOURCE),$(CACHE),$(subst .$(EXTENSION),.$(OUTPUT_FILE_NAME),$(wildcard $(SOURCES)/*.$(
```

CLEAN_UP_DELETED_FILES

Defines a shell function/loop for use in the cleanup routines that deletes files in the CACHE ` directory if they do not exist in the SOURCE directory.

In some cases, if you delete or move a file within the SOURCE hierarchy, stale tasks remain on the list. Use clean to run this routine.

The code that implements this function, formatted for easy reading, is as follows:

```
for item in `find $(CACHE)/ -name "*$(OUTPUT_FILE_NAME)"` ;
  do
    temp=`echo $$item | sed -e "s/$(OUTPUT_FILE_NAME)/$(EXTENSION)/" -e "s@$(CACHE)@$(SOURCE)@"`

    if [[ ! -f "$$temp" ]]
      then
        echo "rm $$item"
        rm $$item
    fi
done
```

See Also:

The "clean" target. Additionally this shell operation uses the following Make variables:

- CACHE
- OUTPUT_FILE_NAME
- EXTENSION
- CACHE
- SOURCE

Targets

User Interface

These targets provide an interface and outputs for Taskfile. While these targets do not write data to the cache or output, some have dependencies that may trigger a rebuild.

help

Returns a brief help text that lists the available build targets and a brief overview of their use.

todo

Prints the todo list to the terminal with cat.

This target depends on OUTPUT, so will rebuild the todo list if it is out of date

todo-work

Prints the work-specific todo list to the terminal with cat.

This target depends on WORK_OUTPUT, so will rebuild the work-specific todo list if it is out of date.

Note: The default distribution disables this target by default.

---

todo-future
Prints the aggregated future-todo list to the terminal with `cat`.

This target depends on `WORK_OUTPUT`, so will rebuild the aggregated future-todo list if it is out of date.

Note: The default distribution disables this target by default.

Meta Targets

These targets provide dependency groupings for task list to support basic operation and configuration, but do not build output directly.

all
Provides a single interface to build or rebuild all of the Tasklist output files and their dependencies.

This is the default target for the Taskfile makefile.

See Also:

`all` depends on the following targets:

- `$(SOURCES)`
- `$(CACHE)/.setup`
- `$(CACHE_INDEX_FILES)`
- `$(CACHE)/$(OUTPUT_FILE_NAME).list`
- `$(OUTPUT)`

The following dependencies are not enabled by default:

- `$(FUTURE_OUTPUT)`
- `$(WORK_OUTPUT)`

setup
Runs a sub-make process that builds the `$(CACHE)/.setup` target. This creates all of the required directories and template files for the Tasklist process.

Core Aggregation

This group of targets does the actual core "work" of Taskfile: by creating the cache, collecting the task items, and aggregating the core output list.

$(CACHE)/.setup
This simple configuration target creates all required cache directories, and touches several template files that makes it possible to build the Tasklists without error.

The target creates the following directories:

- `CACHE` (to hold taskfile's cache.)
- `CACHE_DIRS` (to mirror the directory structure so that later targets don't attempt to write to impossible paths.)
- `EXTRA_OUTPUT_DIR` (to hold template files and special output.)

And creates empty files (with the `touch` utility:)

$(EXTRA_OUTPUT_DIR)/tmpl.$(WORK_FILTER)
Ensures that the "tmpl" file for the work-output exists. Taskfile inserts the contents of this file into the beginning of the work-output file before the task items. Taskfile does not generate work-output unless you edit the Makefile to uncomment the relevant targets.

$(EXTRA_OUTPUT_DIR)/tmpl.$(FUTURE_FILTER)
Ensures that the "tmpl" file for the future-output exists. Taskfile interest the contents of this file into the beginning of the future-output file before the task items. Taskfile does not generate future-output unless you edit the Makefile to uncomment the relevant targets.

$(CACHE)/.setup
Taskfile creates this folder to satisfy the dependency checking for the `setup` target.

The build target is just an empty placeholder file.

$(CACHE)/%.$(OUTPUT_FILENAME)
This target depends on `$(SOURCE)/%.$(EXTENSION)`, and is responsible for creating the cache. The cache is a mirror of all the source directory tree, except that only lines that contain a match for the regular expression specified in `KEYWORDS`.

The "`%`" character acts as a wildcard, and when used in both the target and the destination, this target ensures that Taskfile updates the cache whenever a file that matches the dependency (i.e. all files in the `SOURCE` directory hierarchy,) is rebuilt into a cache target.

Because of the structure of this operation, this target ensures that Taskfile only parses those files that end with `EXTENSION`, and that all files in the cache have a distinct extension.

Note: This target suppresses normal output and instead prints "`Caching: $(CACHE)/.$(EXTENSION)`".

$(CACHE)/$(OUTPUT_FILENAME).list
This target depends on `CACHE_INDEX_FILES`, which holds a list of files that the "`$(CACHE)/%.$(OUTPUT_FILENAME)`" generates.

The target performs the following three actions:

- Removes the previous version of "`$(CACHE)/$(OUTPUT_FILENAME).list`".
- Outputs the entire contents of every file in the cache.
- Performs a series of transformations to modify the output of "`grep`" to provide "back links" in the aggregated list that points back to the original source file.

$(OUTPUT)
This target depends on the "`$(CACHE)/$(OUTPUT_FILENAME).list`" output.

The target performs the following two actions:

- Copies the content of `$(EXTRA_OUTPUT_DIR)/tmpl.$(OUTPUT_FILE_NAME)` into the new `OUTPUT` file. This provides any header material.
- Performs a series of transformations on the content of the "`$(CACHE)/$(OUTPUT_FILENAME).list`" file to remove any items that match the `FUTURE_FILTER`, `FUTURE_KEYWORDS`, or `WORK_FILTER`.

  Finally Taskfile sorts the output and writes it to the new `OUTPUT` file.

Advanced Aggregation

Uncomment and customize these targets as necessary in `taskfile.make` file included in this distribution to provide these advanced aggregation features.

$(FUTURE_OUTPUT)

This target builds the file described by the variable FUTURE_OUTPUT. It depends on the $(OUTPUT) target.

Procedurally, this target is similar to the $(OUTPUT) target. $(FUTURE_OUTPUT) has the following components:

- Copies the content of $(EXTRA_OUTPUT_DIR)/tmpl.$(FUTURE_FILTER) into the new FUTURE_OUTPUT file. This provides any header material.
- Selects all of the lines that match FUTURE_KEYWORDS in the file built by the target $(CACHE)/$(OUTPUT_FILE_NAME).list.

  Taskfile sorts these lines before writing them to the output file.
- Performs a series of transformations on the content of the "$(CACHE)/$(OUTPUT_FILENAME).list" file. All transformations occur in the target file, the content in the $(CACHE)/$(OUTPUT_FILENAME).list file is not modified. These transformations to remove any items that match the WORK_FILTER, and clean up potential formatting errors.

$(WORK_OUTPUT)

This target builds the file described by the variable WORK_OUTPUT. It depends on the $(OUTPUT) target.

Procedurally, this target is similar to the $(OUTPUT) and $(FUTURE_OUTPUT) targets and has the following components:

- Copies the content of $(EXTRA_OUTPUT_DIR)/tmpl.$(WORK_FILTER) into the new WORK_OUTPUT file. This provides header data.
- Selects all of the lines that match WORK_KEYWORDS in the file built by the target $(CACHE)/$(OUTPUT_FILE_NAME).list.

  Taskfile sorts these lines before writing them to the output file.
- Performs a series of transformations on the content imported in from "$(CACHE)/$(OUTPUT_FILENAME).list". All transformations occur in the target file, the content in the $(CACHE)/$(OUTPUT_FILENAME).list file is not modified.

$(PROJECTS_OUTPUT)

This target, which builds the $(SOURCE)/projects.$(EXTENSION) file, calls a sub-make in the context of the directory specified by the PROJECTS_MAKECONTEXT. This assumes that, when active, there is a projects-specific Taskfile located in the PROJECTS_MAKECONTEXT.

The taskfile.projects provides an example of such a file.

Cleaning Aggregation

These targets are useful for forcing Taskfile to delete certain files that have grown stale or that you would like to regenerate during the next build.

clean

The clean target will delete the generated output, remove stale files from the cache, and run the setup routine. In short, this target does everything that you need short of deleting the CACHE directory to get a good build.

Runs the command specified by the CLEAN_UP_DELETED_FILES variable.

clean removes the following files directly:

- OUTPUT
- FUTURE_OUTPUT
- WORK_OUTPUT

- PROJECTS_OUTPUT
- $(CACHE)/$(OUTPUT_FILE_NAME).list

When the clean operation has finished, this target runs a sub-make using the setup (in silent mode.)

clean-output
The clean-output target removes the following files:

- OUTPUT
- FUTURE_OUTPUT
- WORK_OUTPUT
- PROJECTS_OUTPUT
- $(CACHE)/$(OUTPUT_FILE_NAME).list

When the clean operation has finished, this target runs a sub-make using the setup (in silent mode.)

Use clean-output as a less intensive version of the clean process because of the omission of the CLEAN_UP_DELETED_FILES procedure.

clean-setup
The clean-setup target removes the .setup file created by the $(CACHE)/.setup target.

clean-cache
The clean-cache removes the CACHE directory. When the clean operation has finished, this target runs a sub-make using the setup (in silent mode.)

clean-dirty
The clean-dirty removes the CACHE directory.

clean-all
The clean target removes the following files:

- CACHE
- OUTPUT
- FUTURE_OUTPUT
- WORK_OUTPUT
- PROJECTS_OUTPUT
- $(CACHE)/$(OUTPUT_FILE_NAME).list

Building this target will remove all files created by Taskfile.

Dependencies

$(SOURCE)/%.$(EXTENSION)
Used by the $(CACHE)/%.$(OUTPUT_FILENAME).

$(SOURCE)/%.$(EXTENSION) provides a matching dependency for all the files specified by the SOURCES variable.

1.4 Integrating Taskfile with Emacs

This document provides an overview of the taskfile.el included in the root of this repository. The organization of this file mirrors the organization of taskfile.el

Taskfile

Variables

`taskfile-location`
Defines the full path to your main taskfile. `taskfile-open` uses this variable.

`taksfile-flow-location`
Defines the full path to the *flow* file. `taskfile-flow` uses this variable.

`taskfile-compile-command`
Defines the make invocation used to rebuild your taskfile. `taskfile-compile` uses this variable.

Functions

`taskfile-mark-done`
This interactive function takes a standard Taskfile task entry (in the source format) and transforms it into a Markdown list-entry prefixed by the string "`DONE`". For example, the `taskfile-mark-done` takes following task:

```
TODO work on Taskfile Project
```

and transforms it into:

```
- DONE work on Taskfile Project
```

`taskfile-open`
Opens the buffer that holds the default taskfile. Define the full path to your taskfile in `taskfile-location`.

`taskfile-open` opens the task list in read-only mode, to prevent unintended editing, and enables Visual Line Mode

`taskfile-flow`
Opens the buffer that holds the *flow* file. Define the full path to this file in `taskfile-flow-location`.

`taskfile-compile`
Runs Emacs' "`compile`" command using the `make` invocation defined in `taskfile-compile-command`. Use this to open `compile-mode` [4]

Deft Mode

Deft is a note-taking and notes organization mode for emacs. If you do not have an existing note taking solution, you may find deft useful. The `taskfile.el` provides a few additional functions on top of deft that you may find helpful either in conjunction with deft, or on their own.

Variables

The following variables define aspects of deft operation.

`deft-extension`
Sets the file extension that Deft uses for its files. Typically this should reflect the value of `EXTENSION`.

[4] http://emacswiki.org/emacs/CompilationMode

deft-directory
: Sets the directory in that Deft looks for files. Typically this should either be the value of `SOURCE` or a sub-directory of the directory defined by `SOURCE`.

deft-text-mode
: Specifies a major-mode to use as the default mode for. Typically this should be `markdown-mode` or `rst-mode` but any available major-mode in your emacs installation, preferably one that Taskfile's regular expressions can parse, will work.

deft-auto-save-interval
: Specifies the interval in seconds that deft buffers will automatically write their contents to disk. Typically the best value for this setting is `nil` to prevent this behavior entirely.

Functions

deft-file-make-slug
: This is a helper function used by `tychoish-deft-create` to generate a reasonable lower-case and hyphen separated file name.

tychoish-deft-create
: Prompts the user to enter the name of a new file, with a filneame computed from the user input using `deft-file-make-slug`.

Keybindings

C-c d o
: Calls `deft`.

  Mnemonic: "*deft open.*"

C-c d n
: Calls `tychoish-deft-create`.

  Mnemonic: "*deft new.*"

C-c d d
: Opens the `deft-directory` in a `dired` buffer.

  Mnemonic: "*deft directory.*"

C-c t t
: Calls `taskfile-open`.

  Mnemonic: "*taskfile tasks.*"

C-c t c
: Calls `taskfile-compile`.

  Mnemonic: "*taskfile compile.*"

C-c t f
: Calls `taskfile-flow`

  Mnemonic: "*taskfile flow.*"

Occur Customizations

At the end of `taskfile.el` there are a number of modifications to occur culled from the Emacs Wiki Occur Page that may make occur more easy for you to use. You may choose to omit these customizations if they conflict or disrupt your current workflow.

1.5 Contribute to Taskfile

Source Code

Taskfile source code is available from the following git hosting providers:

- taskfile git repository (Cyborg Institute.)
- taskfile on Github

Feel free to clone or fork at your leisure. Issue a pull request on GitHub or send me an email/IM/IRC message if you want to send a patch, or would like me to pull from another repository changes back into "maineline." Feel free to suggest changes to either the code or the documentation.

Contrib

The `contrib/` directory in the source tree is for enhancements and extensions to Taskfile. While the core makefiles are (and should be) basic and unadorned for easy use, customization, and compatibility, all submissions to `contrib/` are totally appropriate no matter how specialized.

Bugs/Issues

Use the taskfile issue tracker (on GitHub) to browse current bugs/issues/questions, or open a new issue.

Feel free to send me an email if you want to log an issue, but don't want to fuss with the bug tracker yourself (yet.)

1.6 Glossary

aggregation The process of consolidating information or data from a wide number of input sources into a single manageable output.

flow In Taskfile operation, most source files are normal files that contain content and notes, as well as potential actionable task items. The flow file, is an exception, and operates as a scratch space for ad hoc tasks and actionable items that don't merit their own file, or for situations when you need to capture a large number of notes and tasks at once.

future list A *tasklist* filtered from the primary `$(OUTPUT)` according to the `FUTURE_FILTER`. Use to segregate non-actionable tasks that are beyond the current horizon, or in a dependent or frozen state.

keyword A string that you can use to identify a task item in a plain text file. Taskfile's default configuration assumes that keywords are strings of all-capital letters at the beginning of a line.

target In Make terminology a (build) target refers to the output generated according to the process defined by the makefile.

tasklist The output of the Taskfile build process. The `$(OUTPUT)` generates this file in the default build.

variable The values that configure the machine and environmental specific operation of Taskfile. Full documentation is available in the *variables section of the internals reference*.

work list A *tasklist* filtered from the primary `$(OUTPUT)` according to the `WORK_FILTER` (typically based on filename.) Used to segregate non-actionable tasks or different spheres of work from each other.

- *genindex*

2 Resources

- taskfile git repository
- taskfile on Github
- taskfile issue tracker

The latest version of this manual is also available for download in ePub and PDF formats:

- Taskfile Manual, ePub
- Taskfile Manual, PDF

3 Overview

Taskfile, using GNU Make, compiles a tasklist from one or more directories of files, using keywords (i.e. "`TODO`", "`FIXME`", "`FROZEN`") to identify tasks and then generate a task list. Basically, you focus on your own work, create tasks as you need to, and run "`make`" every now and then. While the approach is exceedingly simple, there are a number of practical advantages that this approach provides:

- Task planning can transpire in-parallel with actual work on code or writing, without needing to switch to a task management systems.
- Complete interoperation with number of existing tools and systems, including:
  - Ikiwki
  - Sphinx
  - Git
  - File systems and text files.
  - Emacs' markdown mode, occur, and deft.
  - Pretty much anything else you want.
- Using GNU Make, makes it possible for the aggregation operation to be *very* efficient and robust, so you can use it against large collections of files.
- The implementation of Taskfile is nearly trivial. Dozens of make/shell lines do everything that you need, so it's easy to improve, extend, and tweak how the system works.

In truth this "project," if I may be so bold, is more about the text and documentation that surrounds the code than the other way around. Thus, consider this documentation set a primmer on creating your own Taskfile-based system.

Index

Symbols

A

C

D

E

F

H

K

M

N

O

P

S

T

V

W