Design: OpenTelemetry Go MultiMod Releaser

A proposal for Module versioning compliance and flexible release tooling improvement to OpenTelemetry-Go Repository

By Eddy Lin (AWS)
Reviewed by: Anthony Mirabella (AWS), Alolita Sharma (AWS)
GitHub Issue: https://github.com/open-telemetry/opentelemetry-go/issues/1446

1. Introduction

As outlined in the requirement docs for improving the OpenTelemetry-Go module versioning and release tooling, it is necessary to allow for flexibility in specifying sets of modules which should maintain consistent versioning. For example, as certain sets of modules (for example, tracing modules) become stable and reach versions v1+, other modules may still be experimental and must maintain a v0 versioning tag.

This design doc will propose a way to allow specifying certain sets of modules whose versions must increment in lockstep and adding verification to ensure the tagging is consistent with the semantic import conventions specified in the versioning specifications. This will be done for both the opentelemetry-go and opentelemetry-go-contrib repositories.

What is the current process of releasing and versioning in the repository?
Currently, releasing is done semi-automatically, needing an approver or maintainer to run several scripts in a given sequence. To maintain consistency, the new releasing workflow will be similar but provide the option for flexible module versioning, as well as provide built-in verification steps to ensure the versioning specified is consistent with conventions.

Existing opentelemetry-go Release Process (EXPANDABLE)

Currently, releasing is done semi-automatically, requiring a maintainer to run several scripts in a given sequence, as described below:

1. First, a pre-release phase includes running a script to create a branch which updates **ALL module versions** in the go.mod files within the opentelemetry-go repo, as well as all versions listed in the dependencies for each go.mod file. The changes are manually verified, and the Changelog is updated.

2. In between the pre-release and tagging is a PR for merging the branch created in the pre-release step to the main branch. The tags from the next step are then created at the commit on the main branch where that PR was merged.
3. Then, a script ./tag.sh is run to automatically tag all modules with the same new tag (which must match the one specified earlier).
4. Lastly, the release is created and verified, allowing the contrib repository to have a matching new release.

2. Goals

Our goal is to improve the current process for preparing the opentelemetry-go modules for release:

- First, we create a way to declare sets of modules that must maintain consistent versions using a human-readable versions.yaml file in the root directory of the core and contrib repos. A verification step is then performed to check that versions.yaml is valid.
- Using the above file, we created a Go script to replace the current pre-release and tagging script so that we may increment versions of all modules in a chosen set, rather than all modules within the repository.

3. Design Overview

In order to allow the flexibility of specifying different sets of modules that must maintain consistent versioning, steps 1 and 3 in the current release process listed above (which update the versions of modules to their new specified version) have been altered to take in a set of specified modules whose versions will be incremented in lockstep. At a high level, the proposal has the following steps:

1. To specify the sets of modules that must maintain consistent versions, a human-readable **YAML file** will be created in the repo's root that groups all modules. These modules are listed by their import paths, such as go.opentelemetry.io/otel, rather than as their relative file paths, as the YAML file will contain both core and contrib modules.
2. Verification of the YAML file's module versions and dependencies using a new script must be performed prior to running the pre-release and tagging scripts to update module versions.
3. The pre-release and tagging script have been updated to allow for incrementing the versions of only specified sets of modules (rather than incrementing the versions of ALL modules in the repo).

3.1 Example Module Sets YAML file

 # Copyright statement

module-sets:

stable-v1:
  version: v1.0.0-RC1
  modules:
    - go.opentelemetry.io/otel
    - go.opentelemetry.io/otel/bridge/opencensus
    - go.opentelemetry.io/otel/bridge/opentracing
    - go.opentelemetry.io/otel/example/jaeger
    - go.opentelemetry.io/otel/example/namedtracer
    - go.opentelemetry.io/otel/example/opencensus
    - go.opentelemetry.io/otel/example/otel-collector
    - go.opentelemetry.io/otel/example/passthrough
    - go.opentelemetry.io/otel/example/prometheus
    - go.opentelemetry.io/otel/example/zipkin
    - go.opentelemetry.io/otel/exporters/otlp/otlptrace
    - go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc
    - go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracehttp
    - go.opentelemetry.io/otel/exporters/trace/jaeger
    - go.opentelemetry.io/otel/exporters/trace/zipkin
    - go.opentelemetry.io/otel/exporters/stdout
    - go.opentelemetry.io/otel/oteltest
    - go.opentelemetry.io/otel/trace
    - go.opentelemetry.io/otel/sdk
 experimental-metrics:
  version: v0.20.0
  modules:
    - go.opentelemetry.io/otel/exporters/metric/prometheus
    - go.opentelemetry.io/otel/internal/metric
    - go.opentelemetry.io/otel/metric
    - go.opentelemetry.io/otel/sdk/export/metric
    - go.opentelemetry.io/otel/sdk/metric
excluded-modules:
- go.opentelemetry.io/otel/internal/tools

In the module-sets section, each module set to be specified is named (such as “stable-v1” and “experimental-metrics” in the example above) and contains two fields: version (with semver conventions and a leading “v”) and modules (a list of all modules within the set, specified by their module import paths).

In the excluded-modules, modules are specified that will not maintain any version.

3.2 Source file Organization

In addition to the important source files below, there will also be test files (e.g. _test.go and data files), as described in [Testing Plan: Go MultiMod Releaser App](#).

- multimod/
  - README.md # README for the multimod tool

- main.go # creates the Cobra app binary
- cmd/ # handles CLI commands, subcommands, and flags using Cobra
  - root.go # define global flags
  - verify.go
  - prerelease.go
  - sync.go
  - tag.go
- internal/
  - common/ # shared functions for multimod app
    - commontest/ # package containing shared functions used for testing
    - conversions.go # convert between Mod import path, file path, tag names
    - tools.go # miscellaneous shared funcs
    - versions.go # structs/funcs for reading in version files
  - verify/
    - verify.go
  - prerelease/
    - prerelease.go
  - sync/
    - sync.go
  - tag/
    - tag.go

4. Releasing Workflow

NOTE: This workflow describes the general process for creating a release, but some of the function names and command line calls have been slightly changed. For the most up-to-date usage, find the MultiMod releaser README. At the time of writing, the most up-to-date README can be found here to avoid duplication.

Within this file, a description of each command's flag options and functionality is described.

5. Detailed Implementation

5.1 Global Cobra Flags

Flags:
-h, --help help for versions
-v, --versioning-file string Path to versioning file that contains definitions of all module sets. If unspecified, defaults to versions.yaml in the Git repo root. (default "/Users/edwali/Documents/GitHub/opentelemetry-go-build-tools/versions.yaml")
At the heart of the MultiMod app are the source files in multimod/internal directory. In the sections below, the important functionality is highlighted or listed for each of the directories. To make it easier to view the overall picture, error checking statements have been removed.

5.2 Common Functions

package common // import "go.opentelemetry.io/build-tools/multimod/internal/common"

Package common provides helper functions for reading and parsing Module
versioning files, as well as common functions used by multiple commands in
the Cobra application.
Listed below are several exported and important unexported functions from the common
directory.

versions.go

```
// excludedModules functions as a set containing all module paths that are excluded
// from versioning.
type excludedModulesSet map[ModulePath]struct{}

// ModuleSetMap maps the name of a module set to a ModuleSet struct.
type ModuleSetMap map[string]ModuleSet

// ModuleSet holds the version that the specified modules within the set will have.
type ModuleSet struct {
  Version string       `mapstructure:"version"`
  Modules []ModulePath `mapstructure:"modules"`
}

// ModulePath holds the module import path, such as "go.opentelemetry.io/otel".
type ModulePath string

// ModuleInfoMap is a mapping from a module's import path to its ModuleInfo struct.
type ModuleInfoMap map[ModulePath]ModuleInfo

// ModuleInfo is a reverse of the ModuleSetMap, to allow for quick lookup from module
// path to its set and version.
type ModuleInfo struct {
  ModuleSetName string
  Version       string
}

// ModuleFilePath holds the file path to the go.mod file within the repo,
// including the base file name ("go.mod").
type ModuleFilePath string

// ModulePathMap is a mapping from a module's import path to its file path.
type ModulePathMap map[ModulePath]ModuleFilePath

// ModuleTagName is the simple file path to the directory of a go.mod file used for Git tagging.
```

```
// For example, the opentelemetry-go/sdk/metric/go.mod file will have a ModuleTagName
"sdk/metric".
type ModuleTagName string

// versionConfig is needed to parse the versions.yaml file with viper.
type versionConfig struct {
  ModuleSets      ModuleSetMap `mapstructure:"module-sets"`
  ExcludedModules []ModulePath `mapstructure:"excluded-modules"`
}
// readVersioningFile reads in a versioning file (typically given as versions.yaml) and returns
// a versionConfig struct.
func readVersioningFile(versioningFilename string) (versionConfig, error) {}

// BuildModulePathMap creates a map with module paths as keys and go.mod file paths as
values.
func (versionCfg versionConfig) BuildModulePathMap(root string) (ModulePathMap, error) {}
```

tools.go

```
// IsStableVersion returns true if modSet.Version is stable (i.e. version major greater than
// or equal to v1), else false.
func IsStableVersion(v string) bool {}

// GetAllModuleSetNames returns the name of all module sets given in a versioningFile.
func GetAllModuleSetNames(versioningFile string, repoRoot string) ([]string, error) {}

// UpdateGoModFiles updates the go.mod files in modFilePaths by updating all modules listed in
// newModPaths to use the newVersion given.
func UpdateGoModFiles(modFilePaths []ModuleFilePath, newModPaths []ModulePath,
newVersion string) error {}

// RunGoModTidy takes a ModulePathMap and runs "go mod tidy" at each module file path.
func RunGoModTidy(modPathMap ModulePathMap) error {}
```

conversions.go

```
// ModulePathsToTagNames returns a list of tag names from a list of module's import paths.
func ModulePathsToTagNames(modPaths []ModulePath, modPathMap ModulePathMap,
repoRoot string) ([]ModuleTagName, error) {}
```

git.go

```
// CommitChangesToNewBranch creates a new branch, commits to it, and returns to the original
worktree.
func CommitChangesToNewBranch(branchName string, commitMessage string, repo
*git.Repository) error {}

// GetWorktree returns the worktree of a repo.
func GetWorktree(repo *git.Repository) (*git.Worktree, error) {}
```

```
// VerifyWorkingTreeClean returns nil if the working tree is clean or an error if not.
func VerifyWorkingTreeClean(repo *git.Repository) error {}
```

module_versioning.go

```
// ModuleVersioning holds info about modules listed in a versioning file.
type ModuleVersioning struct {
  ModSetMap  ModuleSetMap
  ModPathMap ModulePathMap
  ModInfoMap ModuleInfoMap
}

// NewModuleVersioning returns a ModuleVersioning struct from a versioning file and repo root.
func NewModuleVersioning(versioningFilename string, repoRoot string) (ModuleVersioning,
error) {}
```

module_set_release.go

```
// ModuleSetRelease contains info about a specific set of modules in the versioning file to be
updated.
type ModuleSetRelease struct {
  ModuleVersioning
  ModSetName string
  ModSet     ModuleSet
  TagNames   []ModuleTagName
}

// NewModuleSetRelease returns a ModuleSetRelease struct by specifying a specific set of
modules to update.
func NewModuleSetRelease(versioningFilename, modSetToUpdate, repoRoot string)
(ModuleSetRelease, error) }

// ModSetVersion gets the version of the module set to update.
func (modRelease ModuleSetRelease) ModSetVersion() string {}

// ModSetPaths gets the import paths of all modules in the module set to update.
func (modRelease ModuleSetRelease) ModSetPaths() []ModulePath {}

// ModuleFullTagNames gets the full tag names (including the version) of all modules in the
module set to update.
func (modRelease ModuleSetRelease) ModuleFullTagNames() []string {}

// CheckGitTagsAlreadyExist checks if Git tags have already been created that match the
specific module tag name
// and version number for the modules being updated. If the tag already exists, an error is
returned.
func (modRelease ModuleSetRelease) CheckGitTagsAlreadyExist(repo *git.Repository) error {}
```

5.3 Commontest

package commontest // import "go.opentelemetry.io/build-tools/multimod/internal/common/commontest"

Package commontest provides exported shared helper functions for testing.

commontest.go

```
// WriteGoModFiles is a helper function to dynamically write go.mod files used for testing.
// This func is duplicated from the commontest package to avoid a cyclic dependency.
func WriteGoModFiles(modFiles map[string][]byte) error {}

// RemoveAll attempts to remove a directory and all nested subdirectories,
// taking in a testing instance and providing a Fatal to stop tests if failed.
func RemoveAll(t *testing.T, dir string) {}
```

5.4 Verify

Types and Funcs

```
type verification struct {
  common.ModuleVersioning
}

// dependencyMap keeps track of all modules' dependencies.
type dependencyMap map[common.ModulePath][]common.ModulePath
```

Run

```
func Run(versioningFile string) {
   repoRoot, err := tools.FindRepoRoot()

   v, err := newVerification(versioningFile, repoRoot)

   v.verifyAllModulesInSet()

   v.verifyVersions()

   v.verifyDependencies() // only provides warnings, not errors
}

// verifyAllModulesInSet checks that every module (as defined by a go.mod file) is contained in exactly
// one module set, unless it is excluded.
func (v verification) verifyAllModulesInSet() error {}

// verifyVersions checks that module set versions conform to versioning semantics.
func (v verification) verifyVersions() error {}
```

```
// verifyDependencies checks that dependencies between modules conform to versioning
semantics.
func (v verification) verifyDependencies() error {}
```

5.5 Sync

Types and Funcs

```
// sync holds fields needed to update one module set at a time.
type sync struct {
  OtherModuleSetRelease common.ModuleSetRelease
  MyModuleVersioning    common.ModuleVersioning
}
```

Run

```
func Run(myVersioningFile string, otherVersioningFile string, otherRepoRoot string,
otherModuleSetNames []string, allModuleSets bool, skipModTidy bool) {
  myRepoRoot, err := tools.FindRepoRoot()

  if allModuleSets {
     otherModuleSetNames, err = common.GetAllModuleSetNames(otherVersioningFile,
otherRepoRoot)
  }

  repo, err := git.PlainOpen(myRepoRoot)

  common.VerifyWorkingTreeClean(repo)

  for _, moduleSetName := range otherModuleSetNames {
     s, err := newSync(myVersioningFile, otherVersioningFile, moduleSetName, myRepoRoot,
otherRepoRoot)

     s.updateAllGoModFiles()

     modSetUpToDate, err := s.checkModuleSetUpToDate(repo)
     if modSetUpToDate {
        log.Println("Module set already up to date. Skipping...")
        continue
     } else {
        log.Println("Updating versions for module set...")
     }

     if skipModTidy {
        log.Println("Skipping go mod tidy...")
     } else {
        common.RunGoModTidy(s.MyModuleVersioning.ModPathMap)
     }
```

```
      s.commitChangesToNewBranch(repo)
      }
  }
}
```

5.6 Prerelease

Types and Funcs

```
// prerelease holds fields needed to update one module set at a time.
type prerelease struct {
  common.ModuleSetRelease
}
```

Run

```
func Run(versioningFile string, moduleSetNames []string, allModuleSets bool, skipModTidy
bool) {
  repoRoot, err := tools.FindRepoRoot()

  if allModuleSets {
    moduleSetNames, err = common.GetAllModuleSetNames(versioningFile, repoRoot)
  }

  repo, err := git.PlainOpen(repoRoot)

  common.VerifyWorkingTreeClean(repo)

  for _, moduleSetName := range moduleSetNames {
    p, err := newPrerelease(versioningFile, moduleSetName, repoRoot)

    modSetUpToDate, err := p.checkModuleSetUpToDate(repo)

    if modSetUpToDate {
      log.Println("Module set already up to date (git tags already exist). Skipping...")
      continue
    } else {
      log.Println("Updating versions for module set...")
    }

    p.updateAllVersionGo()

    p.updateAllGoModFiles()

    if skipModTidy {
      log.Println("Skipping go mod tidy...")
    } else {
```

```
            common.RunGoModTidy(p.ModuleVersioning.ModPathMap)
        }

        p.commitChangesToNewBranch(repo)
    }
}
```

5.7 Tag

Types and Funcs

```
type tagger struct {
    common.ModuleSetRelease
    CommitHash plumbing.Hash
    Repo       *git.Repository
}
```

Run

```
func Run(versioningFile, moduleSetName, commitHash string, deleteModuleSetTags bool) {

    repoRoot, err := tools.FindRepoRoot()

    t, err := newTagger(versioningFile, moduleSetName, repoRoot, commitHash,
deleteModuleSetTags)

    // if delete-module-set-tags is specified, then delete all newModTagNames
    // whose versions match the one in the versioning file. Otherwise, tag all
    // modules in the given set.
    if deleteModuleSetTags {
        t.deleteModuleSetTags()

        fmt.Println("Successfully deleted module tags")
    } else {
        t.tagAllModules()
    }
}

func newTagger(versioningFilename, modSetToUpdate, repoRoot, hash string,
deleteModuleSetTags bool) (tagger, error) {
    modRelease, err := common.NewModuleSetRelease(versioningFilename, modSetToUpdate,
repoRoot)

    repo, err := git.PlainOpen(repoRoot)

    fullCommitHash, err := getFullCommitHash(hash, repo)

    modFullTagNames := modRelease.ModuleFullTagNames()
```

```
    if deleteModuleSetTags {
        verifyTagsOnCommit(modFullTagNames, repo, fullCommitHash)
    } else { // if tags already exist, return an error
        if err = modRelease.CheckGitTagsAlreadyExist(repo); err != nil {
            return tagger{}, fmt.Errorf("CheckGitTagsAlreadyExist failed: %v", err)
        }
    }

    return tagger{
        ModuleSetRelease: modRelease,
        CommitHash:       fullCommitHash,
        Repo:             repo,
    }, nil
}
```

Improvements

Several features are in the pipeline for development and deployment. The most recent developments can be found in the “Issues” and “Pull Requests” sections of the [OpenTelemetry-Go-Build-Tools repository](#).

After the base functionalities described above were implemented, a “sync” subcommand was added as described here: [Design: Go MultiMod Releaser Sync and Multi-Set Prerelease](#).