Contributing

Prometheus uses GitHub to manage reviews of pull requests.

- If you are a new contributor see: Steps to Contribute
- If you have a trivial fix or improvement, go ahead and create a pull request, addressing (with `@...`) a suitable maintainer of this repository (see MAINTAINERS.md) in the description of the pull request.
- If you plan to do something more involved, first discuss your ideas on our mailing list. This will avoid unnecessary work and surely give you and us a good deal of inspiration. Also please see our non-goals issue on areas that the Prometheus community doesn't plan to work on.
- Relevant coding style guidelines are the Go Code Review Comments and the *Formatting and style* section of Peter Bourgon's Go: Best Practices for Production Environments.
- Be sure to sign off on the DCO

Steps to Contribute

Should you wish to work on an issue, please claim it first by commenting on the GitHub issue that you want to work on it. This is to prevent duplicated efforts from contributors on the same issue.

Please check the `help-wanted` label to find issues that are good for getting started. If you have questions about one of the issues, with or without the tag, please comment on them and one of the maintainers will clarify it. For a quicker response, contact us over IRC.

For quickly compiling and testing your changes do:

```
make test         # Make sure all the tests pass before you commit and push :)
```

We use `golangci-lint` for linting the code. If it reports an issue and you think that the warning needs to be disregarded or is a false-positive, you can add a special comment `//nolint:linter1[,linter2,...]` before the offending line. Use this sparingly though, fixing the code to comply with the linter's recommendation is in general the preferred course of action.

Pull Request Checklist

- Branch from the master branch and, if needed, rebase to the current master branch before submitting your pull request. If it doesn't merge cleanly with master you may be asked to rebase your changes.
- Commits should be as small as possible, while ensuring that each commit is correct independently (i.e., each commit should compile and pass tests).

- If your patch is not getting reviewed or you need a specific person to review it, you can @-reply a reviewer asking for a review in the pull request or a comment, or you can ask for a review on IRC channel #prometheus on irc.freenode.net (for the easiest start, join via Riot).
- Add tests relevant to the fixed bug or new feature.

Dependency management

The Prometheus project uses Go modules to manage dependencies on external packages. This requires a working Go environment with version 1.12 or greater installed.

All dependencies are vendored in the `vendor/` directory.

To add or update a new dependency, use the `go get` command:

```
# Pick the latest tagged release.
go get example.com/some/module/pkg

# Pick a specific version.
go get example.com/some/module/pkg@vX.Y.Z
```

Tidy up the `go.mod` and `go.sum` files and copy the new/updated dependency to the `vendor/` directory:

```
# The GO111MODULE variable can be omitted when the code isn't located in GOPATH.
GO111MODULE=on go mod tidy

GO111MODULE=on go mod vendor
```

You have to commit the changes to `go.mod`, `go.sum` and the `vendor/` directory before submitting the pull request.

API Implementation Guidelines

Naming and Documentation

Public functions and structs should normally be named according to the file(s) being read and parsed. For example, the `fs.BuddyInfo()` function reads the file `/proc/buddyinfo`. In addition, the godoc for each public function should contain the path to the file(s) being read and a URL of the linux kernel documentation describing the file(s).

Reading vs. Parsing

Most functionality in this library consists of reading files and then parsing the text into structured data. In most cases reading and parsing should be separated into different functions/methods with a public `fs.Thing()` method and a private `parseThing(r Reader)` function. This provides a logical separation and allows parsing to be tested directly without the need to read from the filesystem. Using a `Reader` argument is preferred over other data types such as `string` or `*File` because it provides the most flexibility regarding the data source. When a set of files in a directory needs to be parsed, then a `path` string parameter to the parse function can be used instead.

/proc and /sys filesystem I/O

The `proc` and `sys` filesystems are pseudo file systems and work a bit differently from standard disk I/O.
Many of the files are changing continuously and the data being read can in some cases change between subsequent reads in the same file. Also, most of the files are relatively small (less than a few KBs), and system calls to the `stat` function will often return the wrong size. Therefore, for most files it's recommended to read the full file in a single operation using an internal utility function called `util.ReadFileNoStat`. This function is similar to `ioutil.ReadFile`, but it avoids the system call to `stat` to get the current size of the file.

Note that parsing the file's contents can still be performed one line at a time. This is done by first reading the full file, and then using a scanner on the `[]byte` or `string` containing the data.

```
data, err := util.ReadFileNoStat("/proc/cpuinfo")
if err != nil {
    return err
}
reader := bytes.NewReader(data)
scanner := bufio.NewScanner(reader)
```

The `/sys` filesystem contains many very small files which contain only a single numeric or text value. These files can be read using an internal function called `util.SysReadFile` which is similar to `ioutil.ReadFile` but does not bother to check the size of the file before reading.

```
data, err := util.SysReadFile("/sys/class/power_supply/BAT0/capacity")
```