Unexpected module path

A user working on their project, `my-go-project`, might run into an error during `go get -u` as such:

```
$ cd my-go-project
$ go get -u ./...
[...]
go: github.com/golang/lint@v0.0.0-20190313153728-d0100b6bd8b3: parsing go.mod:
unexpected module path "golang.org/x/lint"
[...]
Exit code 1
```

`golang.org/x/lint` is a module whose git repository and module name used to be `github.com/golang/lint` before migrating to the git repo `golang.org/x/lint` and renaming its module name to `golang.org/x/lint`. The Go tool currently stumbles trying to understand the old module name at the new git repository: golang/go#30831.

This was surfaced to `my-go-project` because `my-go-project` or one of its transitive dependencies has a route in the module graph to the old `github.com/golang/lint` module name.

For example, if `my-go-project` itself relies on the old `github.com/golang/lint` module name:

```
$ GO111MODULE=on go mod graph
my-go-project github.com/golang/lint@v0.0.0-20180702182130-06c8688daad7
```

Or, perhaps `my-go-project` depends on an old version of `google.golang.org/grpc` which depends on the old `github.com/golang/lint` module name:

```
$ GO111MODULE=on go mod graph
my-go-project google.golang.org/grpc@v1.16.0
google.golang.org/grpc@v1.16.0 github.com/golang/lint@v0.0.0-20180702182130-
06c8688daad7
```

Finally, perhaps `my-go-project` depends on another dependency that requires an old version of `google.golang.org/grpc`, which in turn depends on the old `github.com/golang/lint` module name:

```
$ GO111MODULE=on go mod graph
my-go-project some/dep@v1.2.3
...
another/dep@v1.4.2 google.golang.org/grpc@v1.16.0
google.golang.org/grpc@v1.16.0 github.com/golang/lint@v0.0.0-20180702182130-
06c8688daad7
```

Removing References To The Name

Until the Go tool is updated to understand a module which has changed its module path (tracking in golang/go#30831), the solution to this is to update the graph so that there are no more paths to the old module name.

Using the examples above, we'll explore updating the graph so that there are no more paths to `github.com/golang/lint`.

Fixing the first example is simple, the only link is from `my-go-project` - which the user controls! Replacing the old location with the new in the `go.mod` - `github.com/golang/lint@v0.0.0-20180702182130-06c8688daad7` with `golang.org/x/lint v0.0.0-20190301231843-5614ed5bae6f` - removes the link from the graph:

```
$ GO111MODULE=on go mod graph
my-go-project golang.org/x/lint@v0.0.0-20190301231843-5614ed5bae6f
```

Fixing the second example involves more steps but is essentially the same process: `google.golang.org/grpc@v1.16.0` provides the link to `github.com/golang/lint`, so `google.golang.org/grpc` should update its `go.mod` from `github.com/golang/lint@v0.0.0-20180702182130-06c8688daad7` to `golang.org/x/lint v0.0.0-20190301231843-5614ed5bae6f` (this thankfully already happened in `v1.17.0`). Then, `my-go-project` should update its `go.mod` to include the new version of `google.golang.org/grpc`, so that we now have:

```
$ GO111MODULE=on go mod graph
my-go-project google.golang.org/grpc@v1.17.0
google.golang.org/grpc@v1.17.0 golang.org/x/lint@v0.0.0-20181026193005-c67002cb31c3
```

Fixing the third example is similar to the second: update to a newer version of `another/dep` which brings in the newer version of `google.golang.org/grpc` which does not contain a reference to `github.com/golang/lint`.

Hooray! Problems solved - there are no more paths to `github.com/golang/lint` for the Go tool to consider, so it does not trip up on this problem during `go get -u`.

A Harder Problem: Removing Trailing History

This is all well and good, and should satisfy most user's problems.

However, there is one situation that ends up being quite a bit more involved: when there are cycles in the module dependency graph. Consider this module dependency graph:

Module Dependency Graph With A Cycle

And, let's imagine that `some/lib` used to depend on `github.com/golang/lint`.

Let's look at this module dependency graph with versions included:

```
$ go mod graph
my-go-lib some/lib@v1.7.0
some/lib@v1.7.0 some-other/lib@v2.5.3
some/lib@v1.7.0 golang.org/x/lint@v0.0.0-20181026193005-c67002cb31c3
some-other/lib@v2.5.3 some/lib@v1.6.0
some/lib@v1.6.0 some-other/lib@v2.5.0
some/lib@v1.6.0 golang.org/x/lint@v0.0.0-20181026193005-c67002cb31c3
some-other/lib@v2.5.0 some/lib@v1.3.1
some/lib@v1.3.1 some-other/lib@v2.4.8
some/lib@v1.3.1 golang.org/x/lint@v0.0.0-20181026193005-c67002cb31c3
some-other/lib@v2.4.8 some/lib@v1.3.0
```

```
some/lib@v1.3.0 some-other/lib@v2.4.7
some/lib@v1.3.0 github.com/golang/lint@v0.0.0-20180702182130-06c8688daad7
```

Visualized with golang.org/x/exp/cmd/modgraphviz:

A Module Dependency Graph With Trailing History

Here we see that even though the last several versions of `some/lib` correctly depend on `golang.org/x/lint`, the fact that `some/lib` and `some-other/lib` share a cycle mean that there's very likely to be a path far back in time.

The reason such paths occur is because the process of bumping versions is usually individually atomic: when `some/lib` bumps its version of `some-other/lib` and release a new version of itself, the latest version of `some-other/lib` still depends on the previous version of `some/lib`. That is, no individual bump of either of these libraries will be enough to remove the chain into history.

To remove the chain into history and remove the old `github.com/golang/lint` reference from the graph for good, both libraries have to bump their versions of each other at the same time.

Atomically Version Bumping Two Libraries

The solution to removing `github.com/golang/lint` is to first make sure `some/lib` doesn't depend on `github.com/golang/lint`, and then to bump both `some/lib` and `some-other/lib` to non-existent future versions of each other. We want this kind of a graph:

```
my-go-lib some/lib@v1.7.1
some/lib@v1.7.1 some-other/lib@v2.5.4
some/lib@v1.7.1 golang.org/x/lint@v0.0.0-20181026193005-c67002cb31c3
some-other/lib@v2.5.4 some/lib@v1.7.1
```

A Module Dependency Graph Without Trailing History

Since `some/lib` and `some-other/lib` depend on each other at the same version, there's no path backwards in time to a point where `github.com/golang/lint` is provided.

Here are the steps to achieve this atomic version bump, assuming `some/lib` is at `v1.7.0` and `some-other/lib` is at `v2.5.3`:

1. Verify that the error does in fact exist:
    1. Run `GO111MODULE=on go get -u ./...` in `some/lib` and `some-other/lib`.
    2. In both repos you should observe the error go: `github.com/golang/lint@v0.0.0-20190313153728-d0100b6bd8b3: parsing go.mod: unexpected module path "golang.org/x/lint"`.
2. Verify that the latest version of `some/lib` depends on `golang.org/x/lint` instead of `github.com/golang/lint`. It would be a shame to remove the historical trails but keep the broken dependency to `github.com/golang/lint`!
3. Bump both libs to non-existent future versions of each other using alpha tags (which are safer since go modules won't consider alpha versions as newer when evaluating the latest released version of a module):
    1. `some/lib` changes its `some-other/lib` dependency from `v2.5.3` to `v2.5.4-alpha`.

    2. `some/lib` tags the commit `v1.7.1-alpha` and pushes the commit and tag.
    3. `some-other/lib` changes its `some/lib` dependency from `v1.6.0` to `v1.7.1-alpha` .
    4. `some-other/lib` tags the commit `v2.5.4-alpha` and pushes the commit and tag.
4. Verify results whilst things are still in an alpha state:
    1. `GO111MODULE=on go build ./...` && `go test ./...` in `some/lib` .
    2. `GO111MODULE=on go build ./...` && `go test ./...` in `some-other/lib` .
    3. `GO111MODULE=on go mod graph` in both repos and assert that there's no path to `github.com/golang/lint` .
    4. Note: `go get -u` still will not work because - as mentioned above - alpha versions aren't considered when evaluating latest versions.
5. If everything looks good, continue by once again bumping to non-existent future versions of each other:
    1. `some/lib` changes its `some-other/lib` dependency from `v2.5.4-alpha` to `v2.5.4`
    2. `some/lib` tags the commit `v1.7.1` and pushes the commit and tag.
    3. `some-other/lib` changes its `some/lib` dependency from `v1.7.1-alpha` to `v1.7.1` .
    4. `some-other/lib` tags the commit `v2.5.4` and pushes the commit and tag.
6. Verify that the error no longer exists:
    1. Run `GO111MODULE=on go get -u ./...` in `some/lib` and `some-other/lib` .
    2. No parsing `go.mod: unexpected module path "golang.org/x/lint"` error should occur.
7. Currently, the `go.sum` s of `some/lib` and `some-other/lib` are incomplete. This is due to the fact that we depended upon future, non-existent versions of modules, so we were not able to generate go.sum entries until the process was finished. So let's fix this:
    1. `GO111MODULE=on go mod tidy` in `some/lib` .
    2. Commit, tag the commit `v1.7.2` , and push both commit and tag.
    3. `GO111MODULE=on go mod tidy` in `some-other/lib` .
    4. Commit, tag the commit `v2.5.5` , and push both commit and tag.
8. Finally, let's make sure that `my-go-project` depends on these new versions of `some/lib` and `some-other/lib` which do not have long historical tails:
    1. Change the `my-go-project` `go.mod` entry from `some/lib v1.7.0` to `some/lib 1.7.2` .
    2. Test by running `GO111MODULE=on go get -u ./...` in `my-go-project` .

Note that between steps 5.b and 5.d, users are broken: a version of `some/lib` has been released that depends on a non-existent version of `some-other/lib` . Therefore, this process should ideally been done real-time so that step 5.d is finished very soon after step 5.b, creating as small a window of breakage as possible.

Larger Cycles

This example explained the process for removing historical trails when there exists a cycle involving two packages in a graph, but what about if there are cycles involving more packages? For example, consider the following graphs:

Module Dependency Graph With Four Related Cycles

Module Dependency Graph With One Four Vertex Cycle

Each of these graphs involve cycles (the latter example) or interconnected modules (the former example) involving four modules, instead of the simple two module example we saw earlier. The process is largely the same, though, but this time in step 3 and 5 we're going to bump all four modules to non-existent future versions of each other, and similarly in steps 4 and 6 we're going to test all four modules, and in step 7 fix the go.sum of all four modules.

More generally, the process above holds for any group of interconnected modules involving any n modules: each major step just involves n modules acting in coordination.