Go Proposal Process: Enabling Experiments

Go Proposals, Part 5

Russ Cox

September 23, 2019

research.swtch.com/proposals-experiment

[*I've been thinking a lot recently about the Go proposal process, which is the way we propose, discuss, and decide changes to Go itself. Like nearly everything about Go, the proposal process is an experiment, so it makes sense to reflect on what we've learned and try to improve it. This post is the fifth in a series of posts about what works well and, more importantly, what we might want to change.*]

Communicating a proposed change precisely and clearly is difficult, on both sides of the communication. Technical details are easy to write and to read incorrectly without realizing it, and implications are easy to misunderstand. After all, this is why our programs are filled with bugs. The best way I know to address this problem for a large change is to implement it and try it. (See my GopherCon talk, "Experiment, Simplify, Ship.")

Being able to try out a possible new feature, whether it is in the design draft or the final proposal stage, is extremely helpful for understanding the feature. Understanding the feature is in turn critical for being able to give meaningful, concrete feedback. Anything we can do to help everyone (including the authors) understand proposals better sooner is a way to improve the overall process.

Prototypes

Multiple contributors at the summit brought up the Go modules proposal as an example of how much it helped to have a working prototype: being able to learn about modules by trying vgo was very helpful for them, instead of having to imagine the experience by reading documents alone. There is a balance to be struck here. It certainly helped to have vgo available for the initial public discussion, but other problems were caused by waiting until then to discuss the ideas publicly. We published the design drafts last summer without working prototypes specifically to avoid that mistake, of discussing ideas too late. But, when we get farther along, especially with generics, it will also be important to make working prototypes available for experimentation well before we reach the final proposal decision.

Short Experiments

We had already recognized the need for experimenting before making a final decision, which motivated the procedure we introduced for language changes starting in Go 1.13. In short, that procedure is: have an initial discussion about whether to move forward; if so, check in the implementation at the start of the three-month development cycle; have a final discussion at the end of the development cycle; if the feature is not ready yet, remove it for the freeze and the release; repeat if needed. This three-month window worked reasonably well for small features like signed shift counts. For larger features, it is clear that three months is too short and a different approach providing a longer window is needed. We spent a while at the summit talking about possible ways to make features available on a longer-term experimental basis, and the various concerns that must be balanced.

Longer, Opt-In Experiments

For `vgo` the way to opt in to experimenting was to download and run a separate command, not the go command. And `vgo` could "compile out" use of modules by preparing a `vendor` directory. For `try`, Robert Griesemer wrote a simple converter, `tryhard`, that looked for opportunities to add `try` expressions; we intended to have a `tryhard -u` that removed them as well, so that people who wanted to experiment with `try` could write code using it and "compile" that down to pure Go when publishing it. A separate command is heavy-weight but has the significant benefit of being independent of the underlying toolchain, the same as non-experimental tools like `goyacc` and `stringer`.

There is also a mechanism for experiments within the main toolchain. The environment variable `GOEXPERIMENT` can be set during the toolchain build (that is, during `all.bash` or `make.bash`) to enable unfinished or experimental features in that toolchain. This mechanism restricts the use of these features to developers who build the Go toolchain itself from source, which is not most users. Indeed, `GOEXPERIMENT` is intended mainly for use by the developers of those in-progress features, typically invisible implementation details, not semantic language changes. (For example, use of register `RBP` to hold a frame pointer on x86-64 was added as an experiment flag until we were sure it was robust enough to enable by default.)

As a lighter-weight mechanism, people at the contributor summit raised the idea of opting in to an experimental feature with a line in in `go.mod` or with a Python-like special import (`import _ "experimental/try"`).

Restricting Experiments

The biggest question about experimental features is how to restrict them—that is, how to contain their impact—to ensure the overall ecosystem does not depend on them before they are stable. On the one hand, you want to make it possible for people to try the feature in real-world use cases, meaning production uses. So you want to be using an otherwise release-quality toolchain where the only change is that the feature is added. (Separate tools and the `GOEXPERIMENT` settings both make this possible.) On the other hand, any production usage creates a reliance on the feature that will translate into breakage if the feature is changed or removed. The ability to gather production experience with the feature and the stability of the usual Go compatibility promise are in direct conflict. Even if users understand that there is no compatibility for an experimental feature, it still hurts them when their code breaks.

A critical aspect of containing experimental features is thinking about how they interact with dependencies. Is it okay for a dependency to opt in to using an experimental feature? If the feature is removed, that might break packages that didn't even realize they depended on it. And what about tools? Do editing environments or tools like `gopls` have to understand the experimental feature as well? It gets complicated fast.

We especially want to avoid breaking people who don't even know they were using the feature. Since the feature is experimental and *will* break, that means trying to prevent a situation where people are using it without knowing, or where people are coerced into using it by an important dependency. Avoiding this problem is the main reason that we have used heavier weight mechanisms like separate tools or the `GOEXPERIMENT` flag to limit the scope of experiments in Go.

At the contributor summit, a few contributors with Rust experience said that a while back many Rust crates simply required the use of Rust's unstable toolchain, which has in-progress features enabled. They also said that the situation has improved, in part because of attention to the problem and in part because some of the most important in-progress features were completed and moved into the stable toolchain.

One problem we had in Go along similar lines was in the introduction of experimental vendoring support in Go 1.5. For that release, vendoring had to be enabled using the `GO15VENDOREXPERIMENT=on` environment variable. The Go 1.6 release changed the default to be opt-out, and Go 1.7 removed the setting entirely. But that meant projects with a significant number of developers had to tell each developer to opt in in order for the project to use it, which made it harder to try and adopt than we realized. Understanding this problem is one of the reasons that modules have defaulted to an "automatic" mode triggered by the presence of a `go.mod` file for the past couple releases. Although the `GO111MODULE` variable allows finer-grained control, it can be ignored by most users. Want to try modules? Create a `go.mod` (which you needed to do anyway).

What To Do

I don't see a silver bullet here. We will probably have to decide for each change what the appropriate experiment mechanism is.

It is critically important to allow users to experiment with proposed features, to better understand them and to find problems early. Significant changes should continue to be backed by prototypes.

On the other hand, it is equally (if not more) important that experimental features not become unchangeable de facto features due to dependency network effects. A lightweight mechanism (like `import _ "experimental/try"`) may be appropriate when a feature is near final and we are willing to support the current semantics in all future toolchains. Before then, such a mechanism is inappropriate: all it takes is one important dependency to make the feature impossible to change.

The most likely answer to the right way to experiment is "it depends." A trivial, well understood change like binary number literals is probably fine to do using the "short" release cycle mechanism. Larger changes like modules or `try` (or generics!) probably need external tools or very careful use of `GOEXPERIMENT` to avoid unwanted network effects.

Even so, we should be careful to remember to provide for some experimentation mechanism for any non-trivial change, well before that change leaves the design draft stage and becomes a formal proposal.

Next

Again, this is the fourth post in a series of posts thinking and brainstorming about the Go proposal process. Everything about these posts is very rough. The point of posting this series—thinking out loud instead of thinking quietly—is so that anyone who is interested can join the thinking.

I encourage feedback, whether in the form of comments on these posts, comments on the newly filed issues, mail to rsc@golang.org, or your own blog posts (please leave links in the comments). Thanks for taking the time to read these and think with me.

It's been a month since the last post, because I was away for three weeks with my kids before school started and then spent last week catching up. I have at least two more posts planned.

The next post is about overall representation in the proposal process and the Go language effort more broadly: who is here, who is missing, and what can we do about it?