The New Era of Go Package Management

sam boyer

@sdboyer

Hi! Here we are, the last session of the ‘con. I hope you’ve all had as much fun as I have.

I’m here to talk about package management, as we’re now solidly on the road toward of new official tooling for this, and it’s going to be a real sea change.

Still, package management might seem like an odd topic for a closing keynote, as at first glance, it seems kinda narrow. I think, though, it speaks to something pretty universal. So, I want to start by making sure we’re all on the same page with what package management is fundamentally about, and why it matters.

1. We write code

OK, so, someone writes code.

2. We use others' code

@sdboyer

And, because writing code is not an academic exercise, we generally don't want to do everything from scratch. So, we rely on others' code

3. ow

@sdboyer

And, over time, this becomes painful.

* Ensuring what we test in dev is the same as what we run in prod
* Managing updates from upstream
* Especially when those updates break our code
* Or when our deps disagree about what version of other deps to use
* Ensuring our users can build with all the same versions of dependencies that we tested with

Package (dependency) management *matters*

@sdboyer

we HAVE to deal with these problems. everyone writing software does. and it’s always a drag, because these are *never* the issues we actually want to deal with. fixing issues with dependencies is inevitably yak-shaving.

No, like…*more* than that

But this is about more than just that. To address the pain, we're creating new tools that we'll use to interact with other people in the community. That means new types of interactions, new expectations of each other, and new community norms.

Tools -> Social Systems

@sdboyer

When we talk about building a new dependency management tool, we're really talking about making new rules for everybody to interact with through that tool.

Invented 1876

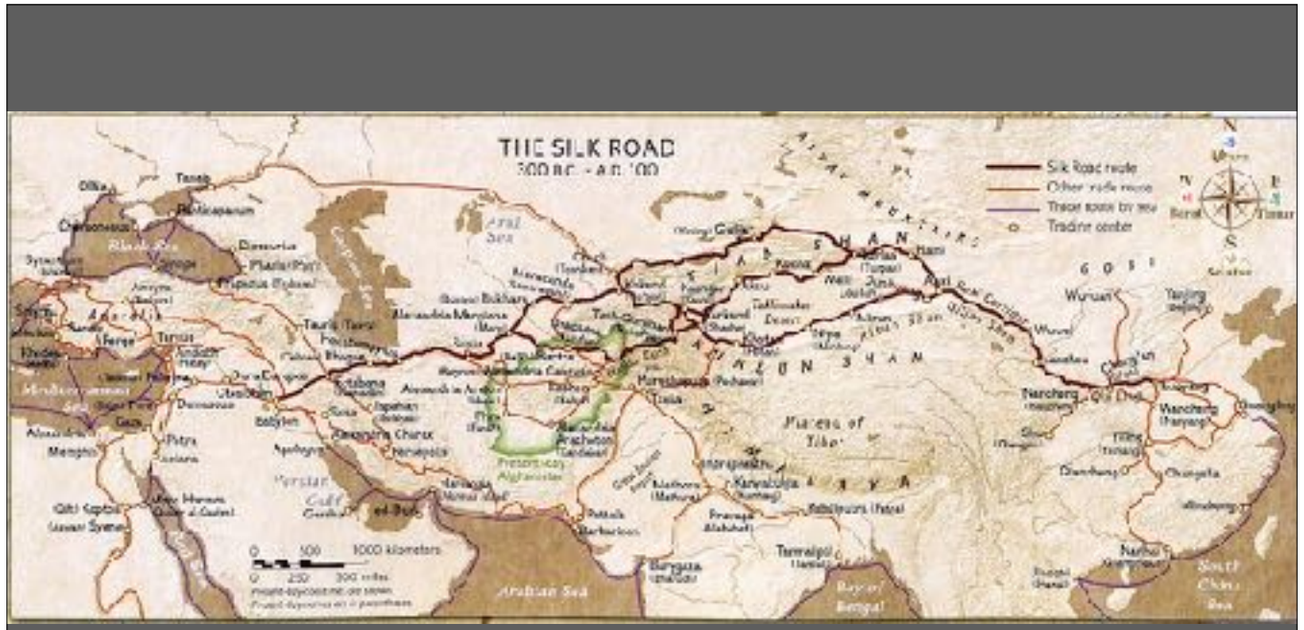
Developing the telephone changed how fast and how far we could communicate with other people. Coordination at a distance was suddenly a thing; information could pass from home to home without people ever needing to leave. It didn’t take off right away, but this contributed

And, of course, when the telephone moved from the wall to our pockets, social protocols changed more. I'm an older millennial, so I remember actually setting up places to meet, instead of figuring it out on the fly.

Dependency managers might be the most boring social tech ever…

and that’s a low bar, because an app that just sent “Yo” existed - but they’re still a social system, because they mediate and coordinate the interactions and relationships between people writing software.

In fact, I’d like to go even a bit further, with another analogy, and talk about pidgin languages. These are languages that evolve when people with no common language need to interact. No one speaks them natively, and they’re often transient - not much of a foundation for culture, like language usually is. A common place they evolved was trade routes



Like the Silk Road. Many pidgins were used there, ad hoc.

When I think about the history of Go as a community and software ecosystem, it seems to me that we've been without common language, hacking together transient pidgins when we wanted to share our work. When looking at upstream go code, typically at whatever random revision I happened to grab, I often quite literally have the sensation of floating, unmoored from any guidance the upstream author might offer as to whether this version even really works or not (WORK ON THIS) - "YOU'RE ON YOUR OWN"

Now, as we move towards official tooling, it's rather like these pidgins can go away, and we can all start speaking a real language with each other. And that's exciting, not just because the pain goes away, but because I've no idea how much better this already-great community and ecosystem will get when we're all speaking the same language to trade.

Can read talk title as "old era of pkg mgmt, new era of pkg mgmt." But the better reading is, "the era of Go before robust pkg mgmt, and the era of Go after."

WE'VE BEEN WITHOUT LANGUAGE

OK so I’m perilously close to claiming that package management will make the world a better place, reminiscent of HBO’s Silicon Valley. Time to pump the brakes.

So, to bring us back down to Earth, let’s look back a bit,

towards the fundamental issues and history that brought us to where we are today.

Roots of the problem

- GOPATH allows only one version
- No reproducibility
- Releases, ~pointless
- Updates, a crapshoot

@sdboyer

* GOPATH only allowing one version at a time, and being global, meant that everyone's deps mixed together
* First and foremost was reproducibility.
* go get didn't care about releases (tags), which meant that there was little incentive to make them. For years, people just didn't really at all (and many still don't)
* This disincentive meant that, even if we addressed the multitenancy and build reproducibility problems, we still tended to get fairly random versions of our dependencies - whatever the tip of the project happened to be when we last updated.

* Dependencies aren’t free, and you should be thoughtful about pulling them in
* Don’t break your API, you’ll make your dependers’ lives hell
* Even with the best dependency management tools, these are still good principles, and it’s a valuable thing that they’ve infused Go’s culture

Rise of the tools

- **2013:** Godep, gom, et al. - 2013
- **2014:** glide, gopkg.in, et al. - 2014
- **2015:** gb, govendor, et al. - 2015

@sdboyer

* Tools took aim at some or all of these problems. And they made the situation was made enormously better, though still not “solved”.
* They also often made slightly different tradeoffs, resulting in tools that were incompatible in subtle, or sometimes significant ways.

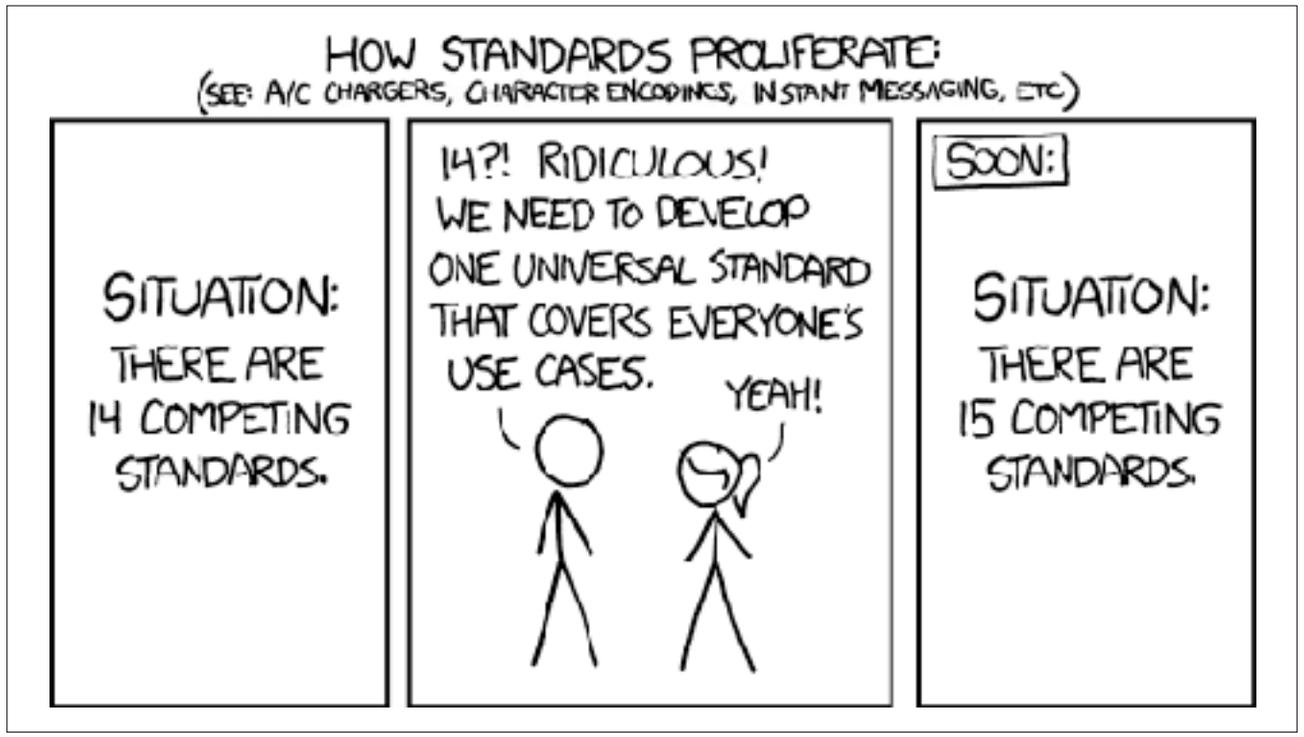
* Godep was the simplest, and remains the most widely used solution today, addressing the problem of reproducibility. But GOPATH manipulation was still a pain. (also, import rewriting, for encapsulation)
* gopkg.in provided something complementary and lightweight that allowed for some diversity of required versions within GOPATH, by simply providing name aliases
* gom and glide were more explicitly modeled after tools from other langs, like bundler and npm, by allowing the user to interact with released versions
* govendor was especially adept at encapsulating

`vendor/` on the scene

- Go 1.5: `vendor/` added, off by default (Aug, 2015)
- Go 1.6: `vendor/` on by default (Feb, 2016)
- Go 1.7: `vendor/` always on (Aug, 2016)

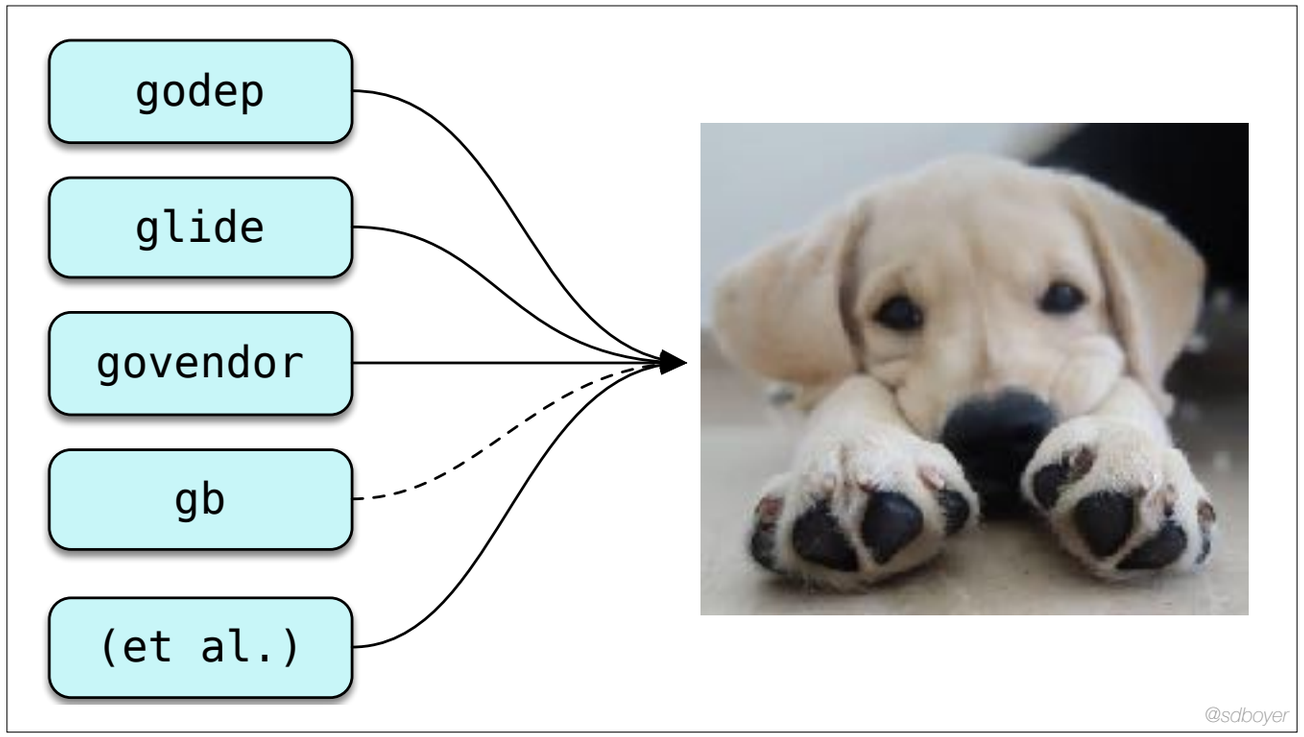
@sdboyer

* The major tools adapted quickly, because vendor provided desperately needed encapsulation.
* But it introduced its own problems - among others, nested vendor directories that duplicated packages could create unnecessary, largely unresolvable type errors
* We in the space had seen this coming for a while, and had been building bridges between the authors of existing tools.
* vendor was a move with unintended side effects, and not terribly well integrated into the tooling. not something we wanted to repeat



Ordinarily, this sort of situation - complex problem, lots of existing approaches, lots of opinions - just results in more new “standards” cluttering things up. But that is not where we are today - which is, itself, a remarkable testament to the awesomeness of the Go community.

Instead, drawing on the energy and momentum of the discussions already ongoing in 2016, Peter Bourgon organized a committee (list the people) plus an advisory group (list the people), that was charged with considering all the things, writing up design docs, and implementing a prototype tool.



And this tool would

* cover the crucial use cases of existing tools, either directly or through equivalent alternatives, but not become a kitchen sink
* allow us to validate some hypotheses about workflow and the system would work at scale
* serve as a working prototype, carving a path for what the community wanted to see in the toolchain
* minimize new rules and assumptions to remain maximally compatible with existing toolchain assumptions
* provide flexible, working code so that the toolchain implementation needn't start from scratch

of course, it would also
* keep everyone happy, all the time, no exceptions
* reduce the gestation period for all dog breeds to a mere forty-eight hours, thereby ensuring a neverending supply of puppies that just can't deal with their paws being too big for their bodies

I kid, but the point is, a lot of considerations to balance. And in January, 2017 we opened it up

dep - the “official experiment”. Those words are carefully chosen - I’ll explain more about them in a minute. But, dep is where we are today.

Oh! Also, there was one other priority we had:

We wanted to be as inclusive of the community as we possibly could, every step of the way. So, before I go any further, I'd like ask folks to stand if you have

* contributed code to dep
* commented on any dep issues,
* participated in discussions in #vendor on slack, or on the package management google group
* responded to any of the dep mgmt surveys

We have come a very, very long way, and we have a LOT to be proud of.

`dep` **fundamentals**

- Borrows from others, but is tailored to Go
- Imports are queen
- Two-file system: `Gopkg.toml, Gopkg.lock`
- Project-oriented
- Semver tagging
- `vendor/`-centric - (almost) no GOPATH

@sdboyer

this isn't the place for a dep deep dive, though Carolyn will be doing a lightning talk on its use during community day.

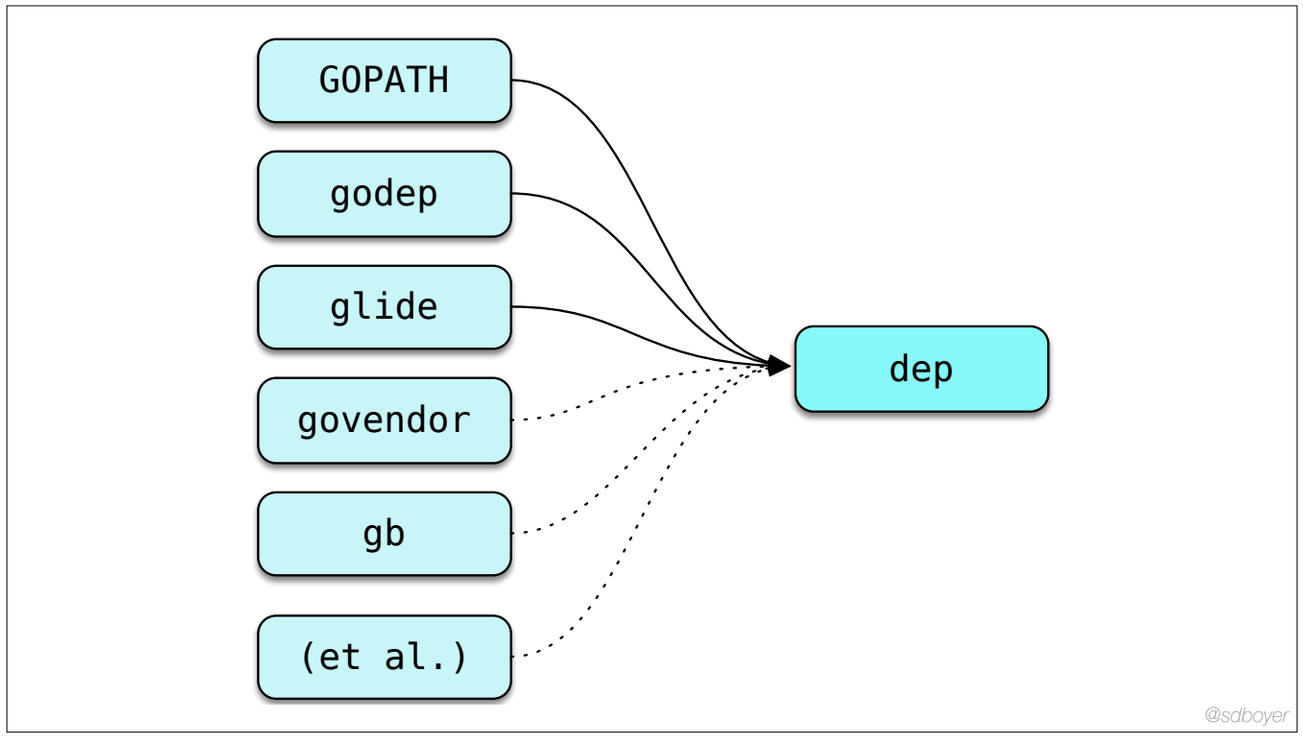
* "We have drunk deeply from the well of other languages' experience…"
    * Past several years have seen package management really turn into its own, cross-lang problem domain. We've borrowed and learned a lot from that, but dep is very much built for Go's unique constraints
    * BUILDING ON A LEGACY, LEARNED FROM MANY LANGUAGES
* Best example of that are that imports and static analysis are still determining what's required - not some metadata file
* We use the two-file system - manifest and lock, or (names). you edit the manifest to declare rules on your direct deps, and dep picks out
* dep is oriented towards projects, which are trees of packages rooted by a Gopkg.toml file. Project roots should almost always be repository roots.

```
dep status

dep init

dep ensure
```

@sdboyer

dep really has only three commands. Keeping things simple helps with the transition towards the toolchain.

* status tells you about the state of your dependencies on disk, if there are updates available, etc.
* init sets up a new project, generating a Gopkg.toml/lock. On existing projects, it'll auto-convert glide and Godeps (and we're adding more), or can read from GOPATH.


GOPATH
godep
glide
govendor
gb
(et al.)
dep
@sdboyer

dep status

dep init

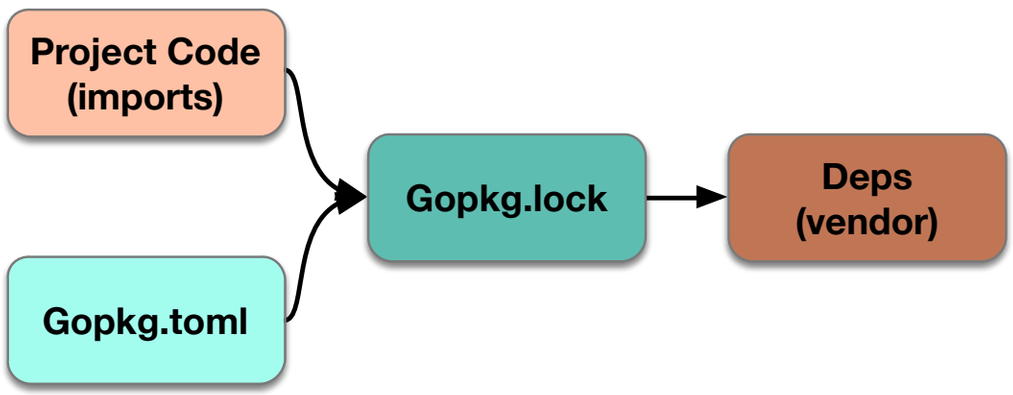
dep ensure

@sdboyer

ok, let’s actually spend some time on this, because this is the where you’ll be spending your time. (“ask permission” for this)

ensure is the real workhorse. it’s what you’ll use, all the time, for most things.

`ensure` keeps states in sync



@sdboyer

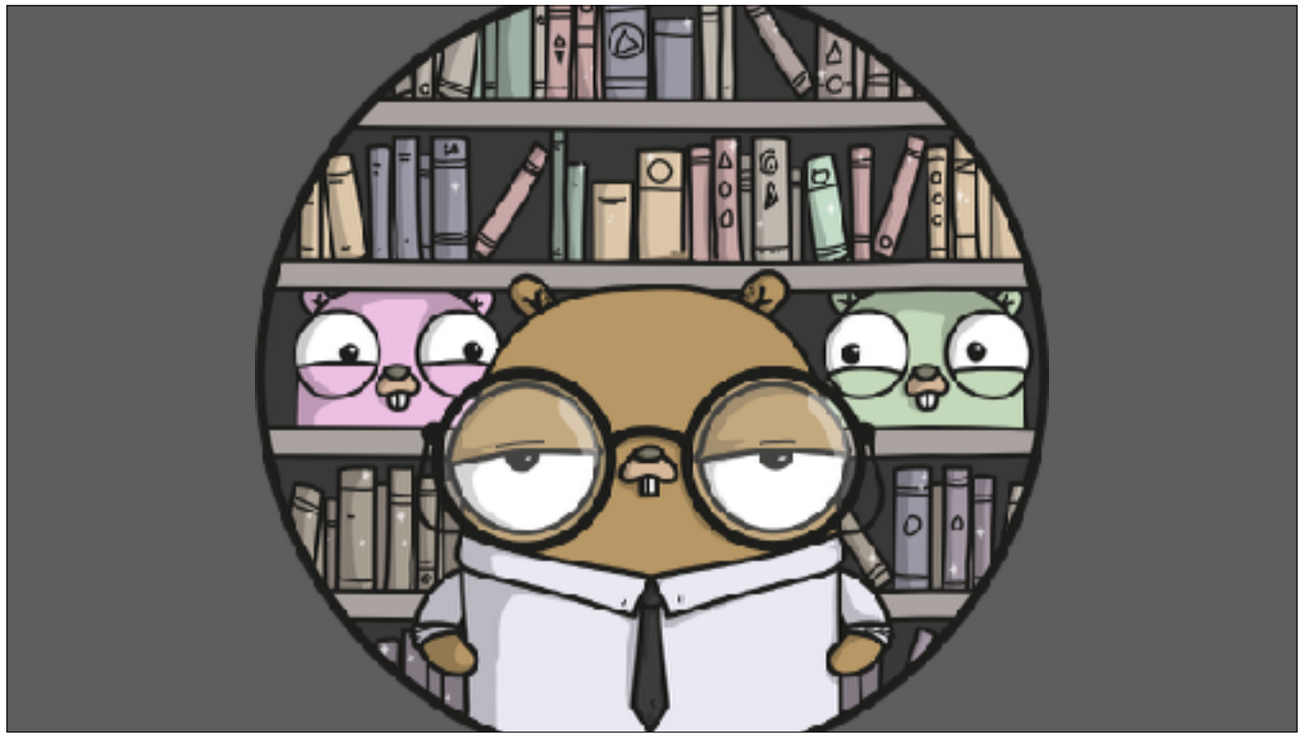
outtro: The tool generally cannot (barring a bug, or explicit user request) exit 0 with these states out of sync.

“Sync-based” tool

@sdboyer

We refer to this general pattern a sync-based tool, to distinguish it from tools that require the user to maintain the relationship between these states. (side note - these principles are leaking back out; a lot of changes in npm5 were inspired by these principles)

It’s another way of saying it’s opinionated. Quite opinionated. Opinionated enough, in fact, that I wrote Ashley McNamara a letter, asking her to help us anthropomorphize a bit.



I would like to introduce you all to Digby.

Digby is a package-managing gopher. He’s kinda part librarian, part accountant - he really likes things being well-ordered and in their proper place, and can get a little finicky about it at times. And hot DAMN, he keeps the house in order.

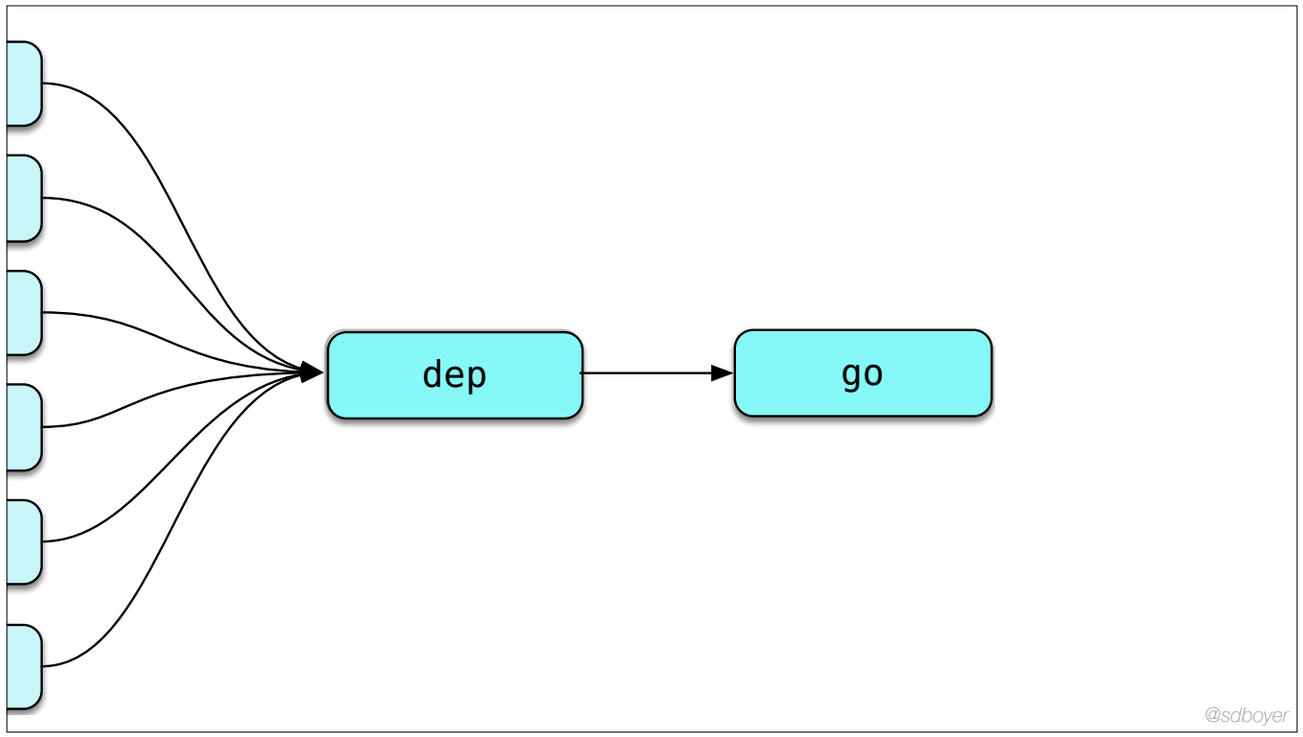
“Hey **`dep`**, **`ensure`** everything’s shipshape, kthx”

@sdboyer

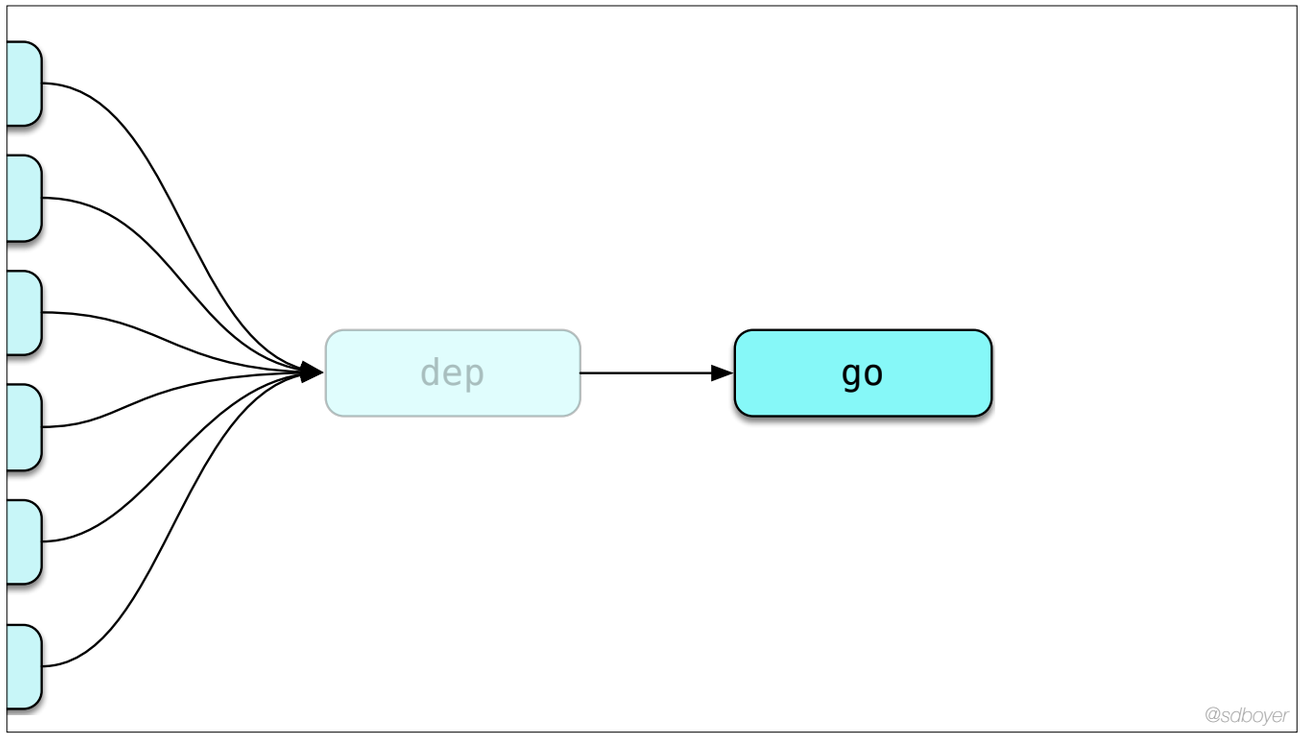
hey dep, ensure that all my imports are satisfied, that my build is reproducible, that all my dependencies are placed in my vendor directory

OK, so that's a snapshot of dep, and dep is at the center of this larger process right now. But dep is not the end goal, here.

* Goal is for dep to not need to exist, because its relevant behavior has been absorbed into the toolchain



* This is where the meaning of “official experiment” really becomes clear: we are focused on creating a clear mapping from dep, to the toolchain.
* That doesn’t necessarily mean one-for-one conversion - so, there may not be a “go ensure”.



But, either way, dep is a stepping stone - one that we hope to phase out when the toolchain itself is ready.

This does highlight something, though.

* As if the task at hand wasn’t challenging enough, this is also the first time the Go team has had to contemplate large swathes of code from the community.
* Pkg Mgmt would be a major undertaking if the Go team was doing it alone
* But the inputs here are coming from the community. No code of appreciable size has ever moved from the community into stdlib or the toolchain

Key insights from `dep`

- Two-file system
- Imports are queen
- Still sync-based
- Semver tagging
- `vendor/`, sorta

@sdboyer

(OTHER WORDS: influence? parallels? heritage? Learned from dep, lessons from dep, etc. key insights from dep)

* Still the same basic two-file system idea
* Imports still determine what's required
* We're still looking at minimizing the command set using the basic principles of being "sync-based." What's not decided is whether that exists as a separate command, analogous to dep ensure, or if it's made implicit in existing go commands - test, list, run, build, etc.

* vendor will likely stick around - it remains the only true defense against upstream disappearance, e.g. left-pad - but the optimal design introduces a third space for upstream packages you're *not* hacking on, used directly by the compiler and e.g. guru, goimports

TODOs

- Multi-project workflow
- Semver suggestion tool
- Registries
- Editor integration patterns
- Security model
- Performance!
- Better failure feedback
- Private/enterprise patterns

@sdboyer

* Multi-project is one of the single most-requested features
* Lots of advantages from a central registry - yank (without left-pad-ing) buggy or insecure versions; can pre-compute metadata. Can make it something enterprises set up for themselves internally, too
*

TODOs…for **YOU**

- Tag your projects with semver
- Convert projects to dep (yes, it's ready!)
- Maybe jump in and contribute to dep - word is, we're *super* friendly!
- Hackathon, tomorrow!
- Updates: sdboyer.io/dep-status

Dave Cheney's written some great blog posts (no surprise) about tagging with semver. Basically, try to follow the Go1 compatibility guarantee, at least until our tool can help you.

Convert your projects! Give us feedback!

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, ***FITNESS FOR A PARTICULAR PURPOSE*** *AND…*

The MIT License

Our communities can only exist because we make no promises to each other that our software works.

It has to be this way, because software is difficult and complex, and that underlying reality is never going to change.

Dep mgmt matters because, when done well, it puts some structure around this chaotic mess. And when it's done really well, taming that complexity becomes a more collective task, rather than one we each have to manage ourselves.

I believe the tools we've built live up to that standard. And I'm excited to see how the community and Go ecosystem grows as a result.

* Thank people, groups
* Call back to Russ’ keynote, wrt Go 2 - and let’s hope that the interaction between the pm team and the Go team can serve as a model for future collaboration on other major features