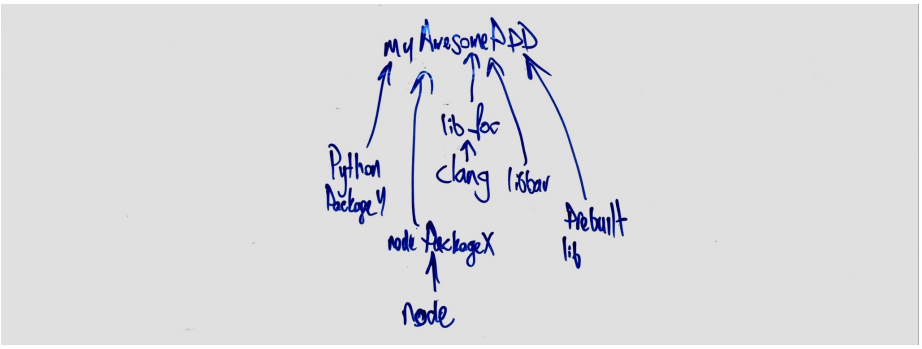
Learn you some build

Sevki

Sun Jan 10 19:09:28 CES 2016

Abstract

How to build native build rules



Build targets are;

- Concurrent
- Declerative
- nodes on a directed acyclic graph (no import cycles, 1-way communication)

So what does this mean for build targets?

Every target you create must exist in a vacuum; which means that your target may only make assumptions about what is in its vacuum, and nothing more. If it's not there, it doesn't exist.

You are free to make assumptions about what is installed from your dependencies a.k.a. your vacuum, at first glance this may seem a bit limiting, but this can compose incredibly versatile workflows.

For instance imagine you have two targets, `foo` and `foo_Clang`, both targets have one source file `foo.c`, `foo_Clang` has one dependency `clang` while `foo` has none.

As mentioned before, children are how you manipulate the vacuum. In this case, `clang` installs a binary in the under `bin` directory in the `foo_Clang` vacuum.

In the `foo_Clang` vacuum calling `CC` will now invoke the `clang` binary installed by that dependency, instead of what ever it is the host machine os had defined in environment variables, mechanics of how this works will be explained in the `Installs()` sections.

This gives you the one place to make assumptions; dependencies. Imagine creating a `os_package` type that installs binaries that your target needs in it's vacuum. And given that these are vacuums, you can have multiple versions of the same compiler running in parallel, because vacuums.

Ok enough chatter; time to create a target type;

First thing first, let's get you familiar with the target interface.

```
// Target defines the interface that rules must implement for becoming build targets.
type Target interface {
    GetName() string
    GetDependencies() []string

    Hash() []byte
    Build(###Context) error
    Installs() map[string]string // map[dst]src
}
```

There is nothing in this interface that isn't absolutely necessary, if it's not needed it will be removed.

** Convenience methods (how's that for irony after that last sentence) `GetName()` return's the name, for the target, that's it.

`GetDependencies()` returns the list of dependencies for a target.

Let's build the first bits

```
package js

import (
    "crypto/sha1"
    "fmt"
    "log"

    "io"
    "path/filepath"
    "strings"
    "time"

    "sevki.org/build"
    "sevki.org/build/ast"
)
```

```
func init() {
    if err := ast.Register("npm_package", NpmPackage{}); err != nil {
        log.Fatal(err)
    }
}

type NpmPackage struct {
    Name         string   `npm_package:"name"`
    Version      string   `npm_package:"version"`
    Dependencies []string `npm_package:"deps"`
}

func (npm ###NpmPackage) GetName() string {
    return npm.Name
}

func (npm ###NpmPackage) GetDependencies() []string {
    return npm.Dependencies
}
```

these are pretty straight forward. And since now we know what the structure of the target will be we can write a `BUILD` file.

```
npm_package(
    name="express"
    version="4.13.3"
)
```

`Hash() []byte`

This is where your target should produce a hash. There are two things to remember when you are writing the hash function;

- You have two hash functions, `Build()` and `Hash()`. If you think of your compilation step as a hash function, for file `foo.c` your c compiler should always produce the same `foo.o`. So your `Hash()` function should be as deterministic as your `Build()` function.
- Whenever build is producing wrong builds, or you have to blow away the `/tmp/build` directory before a build, it means there is a discrepancy between your two hash functions. Fix the `Hash()` never blow the directory away.
- After this step is complete, you shouldn't do anything to change the deterministic nature of the hash produced by this function in the build step, the correctness of your target relies on it.

While build does know where your files are, because it has to make their paths absolute, it will leave you the choice of how to extract a meaningful hash from

them. For instance, `cc_library` hashes the `$CC --version`. Build assumes the author of the target type knows more about how it should be built then the build. So if you have to get the current weather report for hell like so

```
func Hash() []byte {
    res, _ := http.Get("http://www.yr.no/place/Norway/Nord-Tr%C3%B8ndelag/Stj%C3%B8rdal/Hell
    h := sha1.New()
    io.Copy(h, res.Body)
    return h.Sum(nil)
}
```

you are more than welcome to do it.

let's build the hash function for `npm`

```
func (npm *NpmPackage) Hash() []byte {
    h := sha1.New()
    io.WriteString(h, npm.Name)
    if npm.Version == "" {
        io.WriteString(h, npm.Version)
    } else {
        fmt.Fprintf(h, "%d", time.Now().UnixNano())
    }
    return h.Sum(nil)
}
```

as you can see in this we are hashing the version and name of the package, and if the package doesn't have a version we make sure that it's not cached. USE A VERSION YOU HIPPIES!

`Build(*Context) error`

This is the only step of our target that is cached. So this is where it all gets interesting.

So this is the most basic example I could find on npmjs.org

```
$ npm install express
```

let's start by implementing this.

```
func (npm *NpmPackage) Build(c *build.Context) error {
    if npm.Version == "" {
        return fmt.Errorf("NPM package %s failed to install, no version string")
    }
    params := []string{}
    params = append(params, "install")
    params = append(params, fmt.Sprintf("%s@%s", npm.Name, npm.Version))
```

first we check and fail if the version isn't there, second thing we do is to create a `params` array and append `install` and `express@4.13.3` to the array. Make sure that all the flags and arguments you are passing to Exec are their own string items, do not try to be smart and join all the params with `\s` and pass them all in to exec as a single item string array. The params array, essentially ends up being the `os.Args` array on the executed program.

Context

So far, everything we have done has been pretty much go, now's the part I've been going on about in introduction about vacuums. We want to isolate each target build from other's (unless otherwise specified), we do this by using contexts. Each build function gets a context object (it's vacuum). Contexts provide a bunch of functions that are bound to that node, for instance `c.Println()` will write to that node's output stream, since these are usually run in parallel, if all the compilations wrote to `STDOUT` at the same time, it would be indecipherable garbage output. Lets start by writing to that output

```
    c.Println(strings.Join(append([]string{"npm"}, params...), " "))
```

while that's not entirely necessary it's good for debugging later on. The same problem with `STDOUT` becomes a problem when build needs to shell out. If all the workers before they started working called `os.Chdir()` there would be no predicting what would happen, so you need to use `Exec()` which can be used in different directories in parallel. So the last bit of our `npm` build function will be

```
    if err := c.Exec("npm", nil, params); err != nil {
    c.Println(err.Error())
    return fmt.Errorf(err.Error())
    }
    return nil
}
```

First argument exec takes is the name of the application to exec, which in this case is going to be npm, pretty straight forward. Second argument it will get is the string array of environment variables, for the C/C++ target types for instance, this involves adding the `lib` and `include` directories to `LIBRARY_PATH` and `C_INCLUDE_PATH` environment variables respectively, for target types that are expecting binaries to be installed, adding `bin` to `PATH` environment variable would be the way to go.

** Installs() map[string]string

This function returns a map of things to be installed from dependencies to the dependent target. key of the map is the destination, the value it returns is the source of installation.

```
func (npm *NpmPackage) Installs() (installs map[string]string) {
```

```
    path := filepath.Join("node_modules", npm.Name)
    installs[path] = path
    return
}
```

the order of setting the key and the value is not very prominent in this example, so let's give an example where it is;

Destination is always the key and source is always the value, couple of reasons for that, firstly while you can create 2 symlinks with one source, you can't create one symlink with to sources (atleast in unix), another reason why destination is the key is that it makes life easier when looking up sources while making the paths absolute.

So for `cc_library` this would look like this;

```
func (cl *CLib) Installs() map[string]string {
    exports := make(map[string]string)
    libName := fmt.Sprintf("%s.a", cl.Name)
    exports[filepath.Join("lib", libName)] = libName
    return exports
}
```