Understanding the npm dependency model

2016-08-24 ◉ javascript

Currently, npm is *the* package manager for the frontend world. Sure, there are alternatives, but for the time being, npm seems to have won. Even tools like Bower are being pushed to the wayside in favor of the One True Package Manager, but what's most interesting to me is npm's relatively novel approach to dependency management. Unfortunately, in my experience, it is actually not particularly well understood, so consider this an attempt to clarify how exactly it works and how it affects **you** as a user or package developer.

First, the basics

At a high level, npm is not too dissimilar from other package managers for programming languages: packages depend on other packages, and they express those dependencies with *version ranges*. npm happens to use the semver versioning scheme to express those ranges, but the way it performs version resolution is mostly immaterial; what matters is that packages can depend on ranges rather than specific versions of packages.

This is rather important in any ecosystem, since locking a library to a specific set of dependencies could cause significant problems, but it's actually much less of a problem in npm's case compared to other, similar package systems. Indeed, it is often safe for a library author to pin a dependency to a specific version without affecting dependent packages or applications. The tricky bit is determining *when* this is safe and when it's not, and this is what I so frequently find that people get wrong.

Dependency duplication and the dependency tree

Most users of npm (or at least most package authors) eventually learn that, unlike other package managers, npm installs a *tree* of dependencies. That is, every package installed gets its own set of dependencies rather than forcing every package to share the same canonical set of packages. Obviously, virtually every single package manager in existence has to model a

dependency tree at some point, since that's how dependencies are expressed by programmers.

For example, consider two packages, `foo` and `bar`. Each of them have their own set of dependencies, which can be represented as a tree:

```
foo
├── hello ^0.1.2
└── world ^1.0.7

bar
├── hello ^0.2.8
└── goodbye ^3.4.0
```

Imagine an application that depends on *both* `foo` and `bar`. Obviously, the `world` and `goodbye` dependencies are totally unrelated, so how npm handles them is relatively uninteresting. However, consider the case of `hello`: both packages require conflicting versions.

Most package managers (including RubyGems/Bundler, pip, and Cabal) would simply barf here, reporting a version conflict. This is because, in most package management models, **only one version of any particular package can be installed at a time**. In that sense, one of the package manager's primary responsibilities is to figure out a set of package versions that will satisfy every version constraint simultaneously.

In contrast, npm has a somewhat easier job: it's totally okay with installing different versions of the same package because each package gets its own set of dependencies. In the aforementioned example, the resulting directory structure would look something like this:

```
node_modules/
├── foo/
│   └── node_modules/
│       ├── hello/
│       └── world/
└── bar/
    └── node_modules/
        ├── hello/
        └── goodbye/
```

Notably, the directory structure very closely mirrors the actual dependency tree. The above diagram is something of a simplification: in practice, each transitive dependency would have its own `node_modules` directory and so on, but the directory structure can get pretty messy pretty quickly. (Furthermore, npm 3 performs some optimizations to attempt to share dependencies when it can, but those are ultimately unnecessary to actually understanding the model.)

This model is, of course, extremely simple. The obvious effect is that every package gets its own little sandbox, which works absolutely marvelously for utility libraries like `ramda`, `lodash`, or `underscore`. If `foo` depends on `ramda@^0.19.0` but `bar` depends on `ramda@^0.22.0`, they can both coexist completely peacefully without any problems.

At first blush, this system is *obviously* better than the alternative, flat model, so long as the underlying runtime supports the required module loading scheme. However, it is not without drawbacks.

The most apparent downside is a significant increase in code size, given the potential for many, many copies of the same package, all with different versions. An increase in code size can often mean more than just a larger program—it can have a significant impact on performance. Larger programs just don't fit into CPU caches as easily, and merely having to page a program in and out can significantly slow things down. That's mostly just a tradeoff, though, since you're sacrificing performance, not program correctness.

The more insidious problem (and the one that I see crop up quite a lot in the npm ecosystem without much thought) is how dependency isolation can affect cross-package communication.

Dependency isolation and values that pass package boundaries

The earlier example of using `ramda` is a place where npm's default dependency management scheme really shines, given that Ramda just provides a bunch of plain ol' functions. Passing these around is totally harmless. In fact, mixing functions from two different versions of Ramda would be totally okay! Unfortunately, not all cases are nearly that simple.

Consider, for a moment, `react`. React components are very much *not* plain old data; they are complex values that can be extended, instantiated, and rendered in a variety of ways. React

represents component structure and state using an internal, private format, using a mixture of carefully arranged keys and values and some of the more powerful features of JavaScript's object system. This internal structure might very well change between React versions, so a React component defined with `react@0.3.0` likely won't work quite right with `react@15.3.1`.

With that in mind, consider two packages that define their own React components and export them for consumers to use. Looking at their dependency tree, we might see something like this:

```
awesome-button
└── react ^0.3.0

amazing-modal
└── react ^15.3.1
```

Given that these two packages use wildly different versions of React, npm would give each of them their own copy of React, as requested, and packages would happily install. However, if you tried to use these components together, they wouldn't work at all! A newer version of React simply cannot understand an old version's component, so you would get a (likely confusing) runtime error.

What went wrong? Well, dependency isolation works great when a package's dependencies are purely implementation details, never observable from outside of a package. However, as soon as a package's dependency becomes exposed as part of its *interface*, dependency isolation is not only subtly wrong, it can cause complete failure at runtime. These are cases when traditional dependency management are much better—they will tell you as soon as you attempt to install two packages that they just don't work together, rather than waiting for you to figure that out for yourself.

This might not sound *too* bad—after all, JavaScript is a very dynamic language, so static guarantees are mostly few and far between, and your tests should catch these problems should they arise—but it can cause unnecessary issues when two packages *can* theoretically work together fine, but because npm assigned each one its own copy of a particular package (that is, it wasn't quite smart enough to figure out it could give them both the same copy), things break down.

Looking outside of npm specifically and considering this model when applied to other languages, it becomes increasingly clear that this won't do. This blog post was inspired by a Reddit thread discussing the npm model applied to Haskell, and this flaw was touted as a reason why it couldn't possibly work for such a static language.

Due to the way the JavaScript ecosystem has evolved, it's true that most people can often get away with this subtle potential for incorrect behavior without any problems. Specifically, JavaScript tends to rely on duck typing rather than more restrictive checks like `instanceof`, so objects that satisfy the same protocol will still be compatible, even if their implementations aren't *quite* the same. However, npm actually provides a robust solution to this problem that allows package authors to explicitly express these "cross-interface" dependencies.

Peer dependencies

Normally, npm package dependencies are listed under a `"dependencies"` key in the package's `package.json` file. There is, however, another, less-used key called `"peerDependencies"`, which has the same format as the ordinary dependencies list. The difference shows up in how npm performs dependency resolution: rather than getting its own copy of a peer dependency, a package expects that dependency to be provided by its dependent.

This effectively means that peer dependencies are effectively resolved using the "traditional" dependency resolution mechanism that tools like Bundler and Cabal use: there must be one canonical version that satisfies everyone's constraint. Since npm 3, things are a little bit less straightforward (specifically, peer dependencies are not automatically installed unless a dependent package explicitly depends on the peer package itself), but the basic idea is the same. This means that package authors must make a choice for each dependency they install: should it be a normal dependency or a peer dependency?

This is where I think people tend to get a little lost, even those familiar with the peer dependency mechanism. Fortunately, the answer is relatively simple: is the dependency in question visible in *any place* in the package's interface?

This is sometimes hard to see in JavaScript because the "types" are invisible; that is, they are dynamic and rarely explicitly written out. However, just because the types are dynamic does not mean they are not there at runtime (and in the heads of various programmers), so the rule still holds: if the type of a function in a package's public interface somehow depends on a dependency, it should be a peer dependency.

To make this a little more concrete, let's look at a couple of examples. First off, let's take a look at some simple cases, starting with some uses of `ramda`:

```
import { merge, add } from 'ramda'

export const withDefaultConfig = (config) =>
  merge({ path: '.' }, config)

export const add5 = add(5)
```

The first example here is pretty obvious: in `withDefaultConfig`, `merge` is used purely as an implementation detail, so it's safe, and it's not part of the module's interface. In `add5`, the example is a little trickier: the result of `add(5)` is a partially-applied function created by Ramda, so technically, a Ramda-created value is a part of this module's interface. However, the contract `add5` has with the outside world is simply that it is a JavaScript function that adds five to its argument, and it doesn't depend on any Ramda-specific functionality, so `ramda` can safely be a non-peer dependency.

Now let's look at another example using the `jpeg` image library:

```
import { Jpeg } from 'jpeg'

export const createSquareBuffer = (size, cb) =>
  createSquareJpeg(size).encode(cb)

export const createSquareJpeg = (size) =>
  new Jpeg(Buffer.alloc(size * size, 0), size, size)
```

In this case, the `createSquareBuffer` function invokes a callback with an ordinary Node.js `Buffer` object, so the `jpeg` library is an implementation detail. If that were the only function exposed by this module, `jpeg` could safely be a non-peer dependency. However, the `createSquareJpeg` function violates that rule: it returns a `Jpeg` object, which is an opaque value with a structure defined exclusively by the `jpeg` library. Therefore, a package with the above module *must* list `jpeg` as a peer dependency.

This sort of restriction works in reverse, too. For example, consider the following module:

```
import { writeFile } from 'fs'

export const writeJpeg = (filename, jpeg, cb) =>
  jpeg.encode((image) => fs.writeFile(filename, image, cb))
```

The above module does not even *import* the `jpeg` package, yet it implicitly depends on the `encode` method of the `Jpeg` interface. Therefore, despite not even explicitly using it anywhere in the code, a package containing the above module should include `jpeg` as a peer dependency.

They key is to carefully consider what contract your modules have with their dependents. If those contracts involve other packages in any way, they should be peer dependencies. If they don't, they should be ordinary dependencies.

Applying the npm model to other programming languages

The npm model of package management is more complicated than that of other languages, but it provides a real advantage: implementation details are kept as implementation details. In other systems, it's quite possible to find yourself in "dependency hell", when you personally know that the version conflict reported by your package manager is not a real problem, but because the package system must pick a single canonical version, there's no way to make progress without adjusting code in your dependencies. This is extremely frustrating.

This sort of dependency isolation is not the most advanced form of package management in existence—indeed, far from it—but it's definitely more powerful than most other mainstream systems out there. Of course, most other languages could not adopt the npm model simply by changing the package manager: having a global package namespace can prevent multiple versions of the same package being installed at a *runtime* level. The reason npm is able to do what it does is because Node itself supports it.

That said, the dichotomy between peer and non-peer dependencies is a little confusing, especially to people who aren't package authors. Figuring out which packages need to go in which group is not always obvious or trivial. Fortunately, other languages might be able to help.

Returning to Haskell, its strong static type system would potentially allow this distinction to be detected entirely automatically, and Cabal could actually report an error when a package used in an exposed interface was not listed as a peer dependency (much like how it currently prevents importing a transitive dependency without explicitly depending on it). This would allow helper function packages to keep on being implementation details while still maintaining strong interface safety. This would likely take a lot of work to get just right—managing the global nature of typeclass instances would likely make this much more complicated than a naïve approach would accommodate—but it would add a nice layer of flexibility that does not currently exist.

From the perspective of JavaScript, npm has demonstrated that it can be a capable package manager, despite the monumental burden placed upon it by the ever-growing, ever-changing JS ecosystem. As a package author myself, I would implore other users to carefully consider the peer dependencies feature and work hard to encode their interfaces' contracts using it—it's a commonly misunderstood gem of the npm model, and I hope this blog post helped to shed at least a little more light upon it.

← *Using types to unit-test in Haskell* *Climbing the infinite ladder of abstraction* →