npm-packlist

Get a list of the files to add from a folder into an npm package.

These can be handed to tar like so to make an npm package tarball:

```
const packlist = require('npm-packlist')
const tar = require('tar')
const packageDir = '/path/to/package'
const packageTarball = '/path/to/package.tgz'

packlist({ path: packageDir })
  .then(files => tar.create({
    prefix: 'package/',
    cwd: packageDir,
    file: packageTarball,
    gzip: true
  }, files))
  .then(_ => {
    // tarball has been created, continue with your day
  })
```

This uses the following rules:

1. If a `package.json` file is found, and it has a `files` list, then ignore everything that isn't in `files`. Always include the readme, license, notice, changes, changelog, and history files, if they exist, and the package.json file itself.
2. If there's no `package.json` file (or it has no `files` list), and there is a `.npmignore` file, then ignore all the files in the `.npmignore` file.
3. If there's no `package.json` with a `files` list, and there's no `.npmignore` file, but there is a `.gitignore` file, then ignore all the files in the `.gitignore` file.
4. Everything in the root `node_modules` is ignored, unless it's a bundled dependency. If it IS a bundled dependency, and it's a symbolic link, then the target of the link is included, not the symlink itself.
5. Unless they're explicitly included (by being in a `files` list, or a `!negated` rule in a relevant `.npmignore` or `.gitignore`), always ignore certain common cruft files:
    1. .npmignore and .gitignore files (their effect is in the package already, there's no need to include them in the package)
    2. editor junk like `.*.swp`, `._*` and `.*.orig` files
    3. `.npmrc` files (these may contain private configs)
    4. The `node_modules/.bin` folder
    5. Waf and gyp cruft like `/build/config.gypi` and `.lock-wscript`
    6. Darwin's `.DS_Store` files because wtf are those even
    7. `npm-debug.log` files at the root of a project

> You can explicitly re-include any of these with a `files` list in `package.json` or a negated ignore file rule.

Only the `package.json` file in the very root of the project is ever inspected for a `files` list. Below the top level of the root package, `package.json` is treated as just another file, and no package-specific semantics are applied.

Interaction between `package.json` and `.npmignore` rules

For simplicity, it is best to use *either* a `files` list in `package.json` *or* a `.npmignore` file, and not both. If you only use one of these methods, you can skip this documentation section.

The `files` list in `package.json` is used to direct the exploration of the tree. In other words, that's all the walker will ever look at when exploring that level.

In some cases this can lead to a `.npmignore` file being ignored. If a *directory* is listed in `files`, then any rules in a root or nested `.npmignore` files will be honored.

For example, with this package.json:

```
{
  "files": [ "dir" ]
}
```

a `.npmignore` file at `dir/.npmignore` (and any subsequent sub-directories) will be honored. However, a `.npmignore` at the root level will be skipped.

Conversely, with this package.json:

```
{
  "files": ["dir/subdir"]
}
```

a `.npmignore` file at `dir/.npmignore` will not be honored.

Any specific file matched by a glob or filename in the package.json `files` list will be included, and cannot be excluded by any `.npmignore` files in nested directories, or by a `.npmignore` file in the root package directory, unless that root `.npmignore` file is also in the `files` list.

The previous (v1) implementation used in npm 6 and below treated `package.json` as a special sort of "reverse ignore" file. That is, it was parsed and handled as if it was a `.npmignore` file with `!` prepended to all of the globs in the `files` list. In order to include children of a directory listed in `files`, they would *also* have `/**` appended to them.

This is tricky to explain, but is a significant improvement over the previous (v1) implementation used in npm 6 and below, with the following beneficial properties:

- If you have `{"files":["lib"]}` in `package.json`, then the walker will still ignore files such as `lib/.DS_Store` and `lib/.foo.swp`. The previous implementation would include these files, as they'd be matched by the computed `!lib/**` ignore rule.
- If you have `{"files":["lib/a.js","lib/b.js"]}` in `package.json`, and a `lib/.npmignore` containing `a.js`, then the walker will still include the two files indicated in `package.json`, and ignore the `lib/.npmignore` file. The previous implementation would mark these files for inclusion, but then

exclude them when it came to the nested `.npmignore` file. (Ignore file semantics dictate that a "closer" ignore file always takes precedence.)

- A file in `lib/pkg-template/package.json` will be included, and its `files` list will not have any bearing on other files being included or skipped. When treating `package.json` as just Yet Another ignore file, this was not the case, leading to difficulty for modules that aim to initialize a project.

In general, this walk should work as a reasonable developer would expect. Matching human expectation is tricky business, and if you find cases where it violates those expectations, please let us know.

API

Same API as ignore-walk, just hard-coded file list and rule sets.

The `Walker` and `WalkerSync` classes take a `bundled` argument, which is a list of package names to include from node_modules. When calling the top-level `packlist()` and `packlist.sync()` functions, this module calls into `npm-bundled` directly.