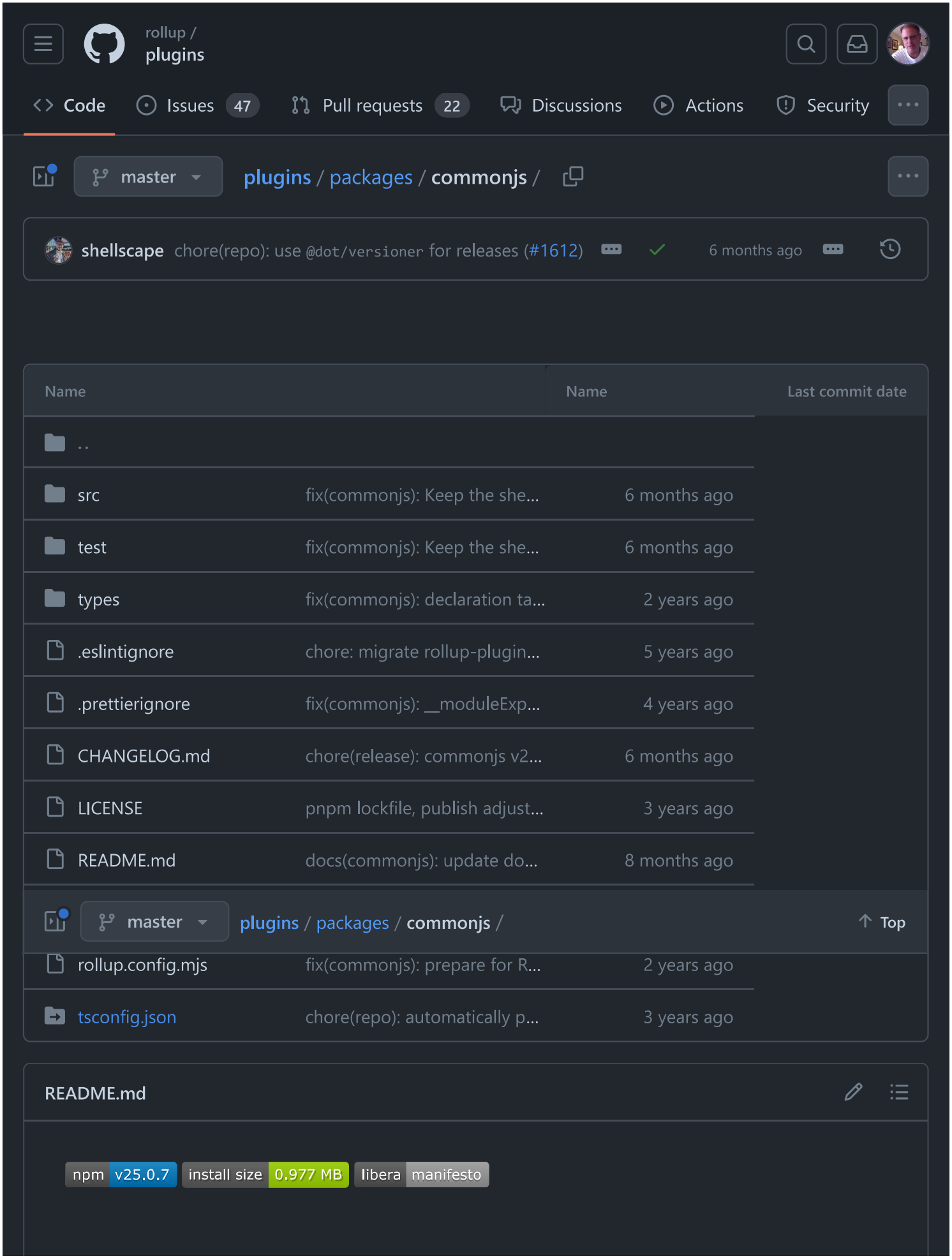
rollup /
plugins
Code
Issues 47
Pull requests 22
Discussions
Actions
Security
master
plugins / packages / commonjs /
shellscape chore(repo): use @dot/versioner for releases (#1612)
6 months ago
Name
Name
Last commit date
..
src fix(commonjs): Keep the she... 6 months ago
test fix(commonjs): Keep the she... 6 months ago
types fix(commonjs): declaration ta... 2 years ago
.eslintignore chore: migrate rollup-plugin... 5 years ago
.prettierignore fix(commonjs): __moduleExp... 4 years ago
CHANGELOG.md chore(release): commonjs v2... 6 months ago
LICENSE pnpm lockfile, publish adjust... 3 years ago
README.md docs(commonjs): update do... 8 months ago
master
plugins / packages / commonjs /
Top
rollup.config.mjs fix(commonjs): prepare for R... 2 years ago
tsconfig.json chore(repo): automatically p... 3 years ago
README.md
npm v25.0.7
install size 0.977 MB
libera manifesto

@rollup/plugin-commonjs

🍣 A Rollup plugin to convert CommonJS modules to ES6, so they can be included in a Rollup bundle

Requirements

This plugin requires an LTS Node version (v14.0.0+) and Rollup v2.68.0+. If you are using `@rollup/plugin-node-resolve`, it should be v13.0.6+.

Install

Using npm:

```
npm install @rollup/plugin-commonjs --save-dev
```

Usage

Create a `rollup.config.js` configuration file and import the plugin:

```
import commonjs from '@rollup/plugin-commonjs';

export default {
  input: 'src/index.js',
  output: {
    dir: 'output',
    format: 'cjs'
  },
  plugins: [commonjs()]
};
```

Then call `rollup` either via the CLI or the API.

When used together with the node-resolve plugin

Options

`strictRequires`

Type: `"auto" | boolean | "debug" | string[]`
Default: `"auto"`

By default, this plugin will try to hoist `require` statements as imports to the top of each file. While this works well for many code bases and allows for very efficient ESM output, it does not perfectly capture CommonJS semantics as the initialisation order of required modules will be different. The resultant side effects can include log statements being emitted in a different order, and some code that is dependent on the initialisation order of polyfills in require statements may not work. But it is especially problematic when there are circular `require` calls between CommonJS modules as those often rely on the lazy execution of nested `require` calls.

Setting this option to `true` will wrap all CommonJS files in functions which are executed when they are required for the first time, preserving NodeJS semantics. This is the safest setting and should be used if the generated code does not work correctly with `"auto"`. Note that `strictRequires: true` can have a small impact on the size and performance of generated code, but less so if the code is minified.

The default value of `"auto"` will only wrap CommonJS files when they are part of a CommonJS dependency cycle, e.g. an index file that is required by some of its dependencies, or if they are only required in a potentially "conditional" way like from within an if-statement or a function. All other CommonJS files are hoisted. This is the recommended setting for most code bases. Note that the detection of conditional requires can be subject to race conditions if there are both conditional and unconditional requires of the same file, which in edge cases may result in inconsistencies between builds. If you think this is a problem for you, you can avoid this by using any value other than `"auto"` or `"debug"`.

`false` will entirely prevent wrapping and hoist all files. This may still work depending on the nature of cyclic dependencies but will often cause problems.

You can also provide a picomatch pattern, or array of patterns, to only specify a subset of files which should be wrapped in functions for proper `require` semantics.

`"debug"` works like `"auto"` but after bundling, it will display a warning containing a list of ids that have been wrapped which can be used as picomatch pattern for fine-tuning or to avoid the potential race conditions mentioned for `"auto"`.

`dynamicRequireTargets`

Type: `string | string[]`
Default: `[]`

Note: In previous versions, this option would spin up a rather comprehensive mock environment that was capable of handling modules that manipulate `require.cache`. This is no longer supported. If you rely on this e.g. when using request-promise-native, use version 21 of this plugin.

Some modules contain dynamic `require` calls, or require modules that contain circular dependencies, which are not handled well by static imports. Including those modules as `dynamicRequireTargets` will simulate a CommonJS (NodeJS-like) environment for them with support for dynamic dependencies. It also enables `strictRequires` for those modules, see above.

Note: In extreme cases, this feature may result in some paths being rendered as absolute in the final bundle. The plugin tries to avoid exposing paths from the local machine, but if you are `dynamicRequirePaths` with paths that are far away from your project's folder, that may require replacing strings like `"/Users/John/Desktop/foo-project/"` -> `"/"`.

Example:

```
commonjs({
  dynamicRequireTargets: [
    // include using a glob pattern (either a string or an array of strings)
    'node_modules/logform/*.js',

    // exclude files that are known to not be required dynamically, this allow
    '!node_modules/logform/index.js',
    '!node_modules/logform/format.js',
    '!node_modules/logform/levels.js',
    '!node_modules/logform/browser.js'
  ]
});
```

`dynamicRequireRoot`

Type: `string`
Default: `process.cwd()`

To avoid long paths when using the `dynamicRequireTargets` option, you can use this option to specify a directory that is a common parent for all files that use dynamic require statements. Using a directory higher up such as `/` may lead to unnecessarily long paths in the generated code and may expose directory names on your machine like your home directory name. By default it uses the current working directory.

`exclude`

Type: `string | string[]`
Default: `null`

A picomatch pattern, or array of patterns, which specifies the files in the build the plugin should *ignore*. By default, all files with extensions other than those in `extensions` or `".cjs"` are ignored, but you can exclude additional files. See also the `include` option.

`include`

Type: `string | string[]`
Default: `null`

A picomatch pattern, or array of patterns, which specifies the files in the build the plugin should operate on. By default, all files with extension `".cjs"` or those in `extensions` are included, but you can narrow this list by only including specific files. These files will be analyzed and transpiled if either the analysis does not find ES module specific statements or `transformMixedEsModules` is `true`.

`extensions`

Type: `string[]`
Default: `['.js']`

For extensionless imports, search for extensions other than .js in the order specified. Note that you need to make sure that non-JavaScript files are transpiled by another plugin first.

`ignoreGlobal`

Type: `boolean`
Default: `false`

If true, uses of `global` won't be dealt with by this plugin.

`sourceMap`

Type: `boolean`
Default: `true`

If false, skips source map generation for CommonJS modules. This will improve performance.

`transformMixedEsModules`

Type: `boolean`
Default: `false`

Instructs the plugin whether to enable mixed module transformations. This is useful in scenarios with modules that contain a mix of ES `import` statements and CommonJS `require` expressions. Set to `true` if `require` calls should be transformed to imports in mixed modules, or `false` if the `require` expressions should survive the transformation. The latter can be important if the code contains environment detection, or you are coding for an environment with special treatment for `require` calls such as ElectronJS. See also the "ignore" option.

`ignore`

Type: `string[] | ((id: string) => boolean)`
Default: `[]`

Sometimes you have to leave require statements unconverted. Pass an array containing the IDs or an `id => boolean` function.

`ignoreTryCatch`

Type: `boolean | 'remove' | string[] | ((id: string) => boolean)`
Default: `true`

In most cases, where `require` calls to external dependencies are inside a `try-catch` clause, they should be left unconverted as it requires an optional dependency that may or may not be installed beside the rolled up package. Due to the conversion of `require` to a static `import` - the call is hoisted to the top of the file, outside of the `try-catch` clause.

- `true`: All external `require` calls inside a `try` will be left unconverted.
- `false`: All external `require` calls inside a `try` will be converted as if the `try-catch` clause is not there.
- `remove`: Remove all external `require` calls from inside any `try` block.
- `string[]`: Pass an array containing the IDs to left unconverted.
- `((id: string) => boolean|'remove')`: Pass a function that control individual IDs.

Note that non-external requires will not be ignored by this option.

`ignoreDynamicRequires`

Type: `boolean` Default: false

Some `require` calls cannot be resolved statically to be translated to imports, e.g.

```
function wrappedRequire(target) {
  return require(target);
}
wrappedRequire('foo');
wrappedRequire('bar');
```

When this option is set to `false`, the generated code will either directly throw an error when such a call is encountered or, when `dynamicRequireTargets` is used, when such a call cannot be resolved with a configured dynamic require target.

Setting this option to `true` will instead leave the `require` call in the code or use it as a fallback for `dynamicRequireTargets`.

`esmExternals`

Type: `boolean | string[] | ((id: string) => boolean)` Default: `false`

Controls how to render imports from external dependencies. By default, this plugin assumes that all external dependencies are CommonJS. This means they are rendered as default imports to be compatible with e.g. NodeJS where ES modules can only import a default export from a CommonJS dependency:

```
// input
const foo = require('foo');

// output
import foo from 'foo';
```

This is likely not desired for ES module dependencies: Here `require` should usually return the namespace to be compatible with how bundled modules are handled.

If you set `esmExternals` to `true`, this plugins assumes that all external dependencies are ES modules and will adhere to the `requireReturnsDefault` option. If that option is not set, they will be rendered as namespace imports.

You can also supply an array of ids to be treated as ES modules, or a function that will be passed each external id to determine if it is an ES module.

`defaultIsModuleExports`

Type: `boolean | "auto"`
Default: `"auto"`

Controls what is the default export when importing a CommonJS file from an ES module.

- `true`: The value of the default export is `module.exports`. This currently matches the behavior of Node.js when importing a CommonJS file.

```
// mod.cjs
exports.default = 3;
```

```
import foo from './mod.cjs';
console.log(foo); // { default: 3 }
```

- `false`: The value of the default export is `exports.default`.

```
// mod.cjs
exports.default = 3;
```

```
import foo from './mod.cjs';
console.log(foo); // 3
```

- `"auto"`: The value of the default export is `exports.default` if the CommonJS file has an `exports.__esModule === true` property; otherwise it's `module.exports`. This makes it possible to import the default export of ES modules compiled to CommonJS as if they were not compiled.

```
// mod.cjs
exports.default = 3;
```

```
// mod-compiled.cjs
exports.__esModule = true;
exports.default = 3;
```

```
import foo from './mod.cjs';
import bar from './mod-compiled.cjs';
console.log(foo); // { default: 3 }
console.log(bar); // 3
```

`requireReturnsDefault`

Type: `boolean | "namespace" | "auto" | "preferred" | ((id: string) => boolean | "auto" | "preferred")`
Default: `false`

Controls what is returned when requiring an ES module from a CommonJS file. When using the `esmExternals` option, this will also apply to external modules. By default, this plugin will render those imports as namespace imports, i.e.

```
// input
const foo = require('foo');

// output
import * as foo from 'foo';
```

This is in line with how other bundlers handle this situation and is also the most likely behaviour in case Node should ever support this. However there are some situations where this may not be desired:

- There is code in an external dependency that cannot be changed where a `require` statement expects the default export to be returned from an ES module.
- If the imported module is in the same bundle, Rollup will generate a namespace object for the imported module which can increase bundle size unnecessarily:

```
// input: main.js
const dep = require('./dep.js');
console.log(dep.default);

// input: dep.js
export default 'foo';

// output
var dep = 'foo';

var dep$1 = /*#__PURE__*/ Object.freeze({
  __proto__: null,
  default: dep
});

console.log(dep$1.default);
```

For these situations, you can change Rollup's behaviour either globally or per module. To change it globally, set the `requireReturnsDefault` option to one of the following values:

- `false`: This is the default, requiring an ES module returns its namespace. This is the only option that will also add a marker `__esModule: true` to the namespace to support interop patterns in CommonJS modules that are transpiled ES modules.

```
// input
const dep = require('dep');
console.log(dep);

// output
import * as dep$1 from 'dep';

function getAugmentedNamespace(n) {
  if (n.__esModule) return n;
  var f = n.default;
  if (typeof f == 'function') {
    var a = function a() {
      if (this instanceof a) {
        return Reflect.construct(f, arguments, this.constructor);
      }
      return f.apply(this, arguments);
    };
    a.prototype = f.prototype;
  } else a = {};
  Object.defineProperty(a, '__esModule', { value: true });
  Object.keys(n).forEach(function (k) {
    var d = Object.getOwnPropertyDescriptor(n, k);
    Object.defineProperty(
      a,
      k,
      d.get
        ? d
        : {
            enumerable: true,
            get: function () {
              return n[k];
            }
          }
    );
  });
  return a;
}

var dep = /*@__PURE__*/ getAugmentedNamespace(dep$1);

console.log(dep);
```

- `"namespace"`: Like `false`, requiring an ES module returns its namespace, but the plugin does not add the `__esModule` marker and thus creates more efficient code. For external dependencies when using `esmExternals: true`, no additional interop code is generated.

```
// output
import * as dep from 'dep';

console.log(dep);
```

- `"auto"`: This is complementary to how `output.exports`: `"auto"` works in Rollup: If a module has a default export and no named exports, requiring that module returns the default export. In all other cases, the namespace is returned. For external dependencies when using `esmExternals: true`, a corresponding interop helper is added:

```
// output
import * as dep$1 from 'dep';

function getDefaultExportFromNamespaceIfNotNamed(n) {
  return n && Object.prototype.hasOwnProperty.call(n, 'default') && Object
    ? n['default']
    : n;
}

var dep = getDefaultExportFromNamespaceIfNotNamed(dep$1);

console.log(dep);
```

- `"preferred"`: If a module has a default export, requiring that module always returns the default export, no matter whether additional named exports exist. This is similar to how previous versions of this plugin worked. Again for external dependencies when using `esmExternals: true`, an interop helper is added:

```
// output
import * as dep$1 from 'dep';

function getDefaultExportFromNamespaceIfPresent(n) {
  return n && Object.prototype.hasOwnProperty.call(n, 'default') ? n['defa
}

var dep = getDefaultExportFromNamespaceIfPresent(dep$1);
```

```
console.log(dep);
```

- `true`: This will always try to return the default export on require without checking if it actually exists. This can throw at build time if there is no default export. This is how external dependencies are handled when `esmExternals` is not used. The advantage over the other options is that, like `false`, this does not add an interop helper for external dependencies, keeping the code lean:

```
// output
import dep from 'dep';

console.log(dep);
```

To change this for individual modules, you can supply a function for `requireReturnsDefault` instead. This function will then be called once for each required ES module or external dependency with the corresponding id and allows you to return different values for different modules.

Using with @rollup/plugin-node-resolve

Since most CommonJS packages you are importing are probably dependencies in `node_modules`, you may need to use @rollup/plugin-node-resolve:

```
// rollup.config.js
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';

export default {
  input: 'main.js',
  output: {
    file: 'bundle.js',
    format: 'iife',
    name: 'MyModule'
  },
  plugins: [commonjs(), resolve()]
};
```

Usage with symlinks

Symlinks are common in monorepos and are also created by the `npm link` command. Rollup with `@rollup/plugin-node-resolve` resolves modules to their real paths by default. So `include` and `exclude` paths should handle real paths rather than symlinked paths (e.g. `../common/node_modules/**` instead of `node_modules/**`). You may also use a regular expression for `include` that works regardless of base path. Try this:

```
commonjs({
  include: /node_modules/
});
```

Whether symlinked module paths are realpathed or preserved depends on Rollup's `preserveSymlinks` setting, which is false by default, matching Node.js' default behavior. Setting `preserveSymlinks` to true in your Rollup config will cause `import` and `export` to match based on symlinked paths instead.

Strict mode

ES modules are *always* parsed in strict mode. That means that certain non-strict constructs (like octal literals) will be treated as syntax errors when Rollup parses modules that use them. Some older CommonJS modules depend on those constructs, and if you depend on them your bundle will blow up. There's basically nothing we can do about that.

Luckily, there is absolutely no good reason *not* to use strict mode for everything — so the solution to this problem is to lobby the authors of those modules to update them.

Inter-plugin-communication

This plugin exposes the result of its CommonJS file type detection for other plugins to use. You can access it via `this.getModuleInfo` or the `moduleParsed` hook:

```
function cjsDetectionPlugin() {
  return {
    name: 'cjs-detection',
    moduleParsed({
      id,
      meta: {
        commonjs: { isCommonJS }
      }
    }) {
```

```
        console.log(`File ${id} is CommonJS: ${isCommonJS}`);
      }
    };
  }
```

Meta

CONTRIBUTING

LICENSE (MIT)