Documentation

Split Modules

Run Command

```
// Root of the project
jscodeshift -t <SCRIPT_NAME> <ENTRY_DIRECTORY/FILE> --targetNam
```

Exports

The script is a `jscodeshift` codemod. Its exports a parser and a default function that are used to run the script for every module in the entry directory. The default parser used is `babel/parser` configured to support typescript modules. The default exported function takes in the file name and path and runs the script if the module contains the target name.

processExports

Takes filePath and calls all the methods in the correct sequential manner. It servers as the entry point for the script to control all the individual transformations and processes. It calls:

- getModuleTypes
- getExportNames
- checkIfExportsAreObjects
- modifyImportSources
- separateExportSpecifiers
- splitImportDeclarations
- replaceVariableDeclarations
- replaceObjectMemberExpressions

- getHelperGraph
- splitObjectExports
- splitNonObjectCode
- splitTypesCode
- createHelperModules

It also handles the global variables and phase-3 of the script ie processing the data and writing to utils-exports, utils-re-exports and utils-types JSONs.

getModuleTypes

Gets the TSTypeAliasDeclarations defined at the module scope. It is used for retrieving non-exported Module level type declarations

```
declare type ColorType = {
}
declare type ColorUnion = string | number
```

getExportNames

Takes in path of a module and returns the exports, re-exports and type exports of that module. It optionally performs operations based on whether the module is the main target file or one of its dependencies using the indexFile flag.

It processes Named, Default and '*' exports.

- For direct exports from the target module, it calls `addFileExports` to store the details of the export in fileExports global object.
- For exported types, it similarly adds the details in exportedTypes object.
- For each re-export, it calls the `getReExports` method for processing the re-exports of the module.

It returns the exports, re-exports and exported types.

getReExports

Processes re-exports of the kind exports ... from ‘source’ of the given file. This method recursively get all the re-exports of the file by calling getExportNames internally for each ‘*’ export.

checkIfExportsAreObjects

get the export kind(objects or non-objects) of each direct export(returned from getExportNames). Using this data it separates the exports into `objectExports` and `nonObjectExports` for separate processing.

modifyImportSources

The method modifies the import path of the target file relative to the new modules that are going to be created by the script. It ensures consistency in imports without breaking the code.

separateExportSpecifiers

This method separates the exports with multuple specifiers into individual export declarations. This is done since the code analyses each export individually and hence needs them separated.

splitImportDeclarations

Similar to separateExportSpecifiers, This method separates the imports with multiple specifiers into individual imports. It is done given the code takes the entire import declaration while searching for a dependency and hence each import needs to be separated

replaceVariableDeclarations

This separates Variable declarations similar to splitImportDeclarations. This is also done to ensure that while analysing a dependency, only the declared variable is included and not the variables declared along with it.

replaceObjectMemberExpressions

Finds object expression like READER.say, READER.accounts and replace these with aliases like say_from_READER, accounts_from_READER. This is done to ensure that naming conflicts dont arise when importing these dependency properties.

getHelperGraph

This runs the phase-2 of the script. It gets all the exported items from the target module(the to-be created modules) and runs the graph algorithm devised to calculate for each helper function whether it is shared and thus needs a separate module or it is used only by 1 export and can be merged in the export's module itself. The algorithm is as follows:

- For each export, a recursive function is called that gets the 'helper' utilities it depends upon. Each helper utility stores an object with **'count' and 'parents'** properties.
- The **count** stores how many exports are dependent on the utility. The **parents** is a set that stores the names of the utilities (exports, non-exports) that directly depends on it and thus have a connecting edge.
- Once the algorithm runs for all the exports, A utility is deemed to have a separate module if **count > 1 and parents > 1.** This is the only necessary condition for the utility to have a new module otherwise it can be put **merged in export's module if count = 1 or in the parent's module if parents = 1.**

This internally calls 'getHelperCount' function that the algorithm for a single export and updates the count, parents property of all dependent utilities in that export's subtree.

splitObjectExports

This function splits the objects exports into individual modules. It internally calls:

- **getObjectExportProps:** Calls this method to get the properties of the object along with the property value
- **addToExportModule:** This method is called to run the recursive AST analyser function to get the dependent imports, helpers and other exports and adds all the required code in that properties code in the correct manner.
- **createIndependentModules:** It adds the necessary code to that property and then create its module in the newModules sub directory

splitNonObjectCode

This function splits the non objects exports into individual modules. It internally calls:

- **addToExportModule:** This is called to run the recursive AST analyser function to get the dependent imports, helpers and other exports and adds all the required code in that exports code in the correct manner.
- **createIndependentModules:** It adds the necessary code to that export's and then create its module in the newModules sub directory

splitTypesCode

Similar to splitNonObjectCode, It splits the type exports specifically into individual modules. It also internally calls:

- **addToExportModule:** This is called to run the recursive AST analyser function to get the dependent imports, helpers and other exports and adds all the required code in that exports code in the correct manner.
- **createIndependentModules:** It adds the necessary code to that export's and then create its module in the newTypes sub directory

createHelperModules

This, like createIndependentModules, creates helper modules in the helperModules sub directory for those helper utilities that have been marked as shared.

Writing data to JSON files

The data collected is processed and stored in the utils-exports, utils-re-exports and utils-types JSON files. This data stores the exports, re-exports and exported types for all the target files processed by the script. This is further used by Import-resolution script to modify imports from these target modules and map it to the correct exporting module.

Import Resolution

Run command

```
// Root of the project
jscodeshift -t <SCRIPT_NAME> <ENTRY_DIRECTORY/FILE>
```

Exports

The script is a `jscodeshift` codemod. Its exports a parser and a default function that are used to run the script for every module in the entry directory. The default parser used is `babel/parser` configured to support typescript modules.

Initialisation

Each run of the script reads data from the utils-exports, utils-re-exports, utils-types and tsconfig JSON files. The first 3 are created after running the split modules script. The script then calls the following functions in the order:

- splitImportDeclarations
- splitExportDeclarations
- modifyWildcardImports
- modifyWildcardExports
- getImportedModules

splitImportDeclarations

Splits the import declarations with multiple into individual declarations if the import source belongs to util exports, util re-exports or util types. This is done given that imports need to be mapped individually to the correct exporting modules created after splitting.

splitExportDeclarations

Similarly splits the export declarations into individual declarations if the export source belongs to util exports, util re-exports or util types. This is also done so that these re-exports can be correctly mapped to the correct exporting source module

modifyWildcardImports

This resolves the namespace/wildcard imports of the form import (type) '*" as Obj from '...' . These imports need to be replaced by individual import declarations so that they can be later mapped to the correct exporting source module. So the wildcard imports are replaced with a list of individual imports, all put together in the original imported namespace.

```
import * as say from './index'
// modified to
import {func1} from './newModules/func1'
import {func2} from './newModules/func2'
const say = {
    func1: func1,
    func2: func2
}
```

modifyWildcardExports

This resolved the namespace/wildcard exports of the form export (type) '*" from '...' . These exports need to be replaced by individual export declarations so that they can be later mapped to the correct exporting source module. So the wildcard exports are replaced with a list of individual exports.

```
export * from './index'
// modified to
export {func1} from './newModules/func1'
export {func2} from './newModules/func2'
```

getImportedModules

This function is called after imports/re-exports splitting and wildcard imports/re-exports resolution. This function now analyses and modifies each import, re export declaration sequentially. It uses the JSON file data to correctly map the imports from one of the target files to the correct sources. It calls internally many methods to map type imports, imports that are re-exports of the source module and imports that are direct exports of the source.

- **modifyImport:** Modifies imports/re-exports that are directly exported from a target module
- **evaluateReExport:** Modifies imports/re-exports that are re-exported from a target module
- **evaluateTypeExport:** Method for handling type imports/re-exports specifically

Utility functions

There are some utility functions as well that are used by one of the above functions:

- **replaceFileName:** Replaces the filename in the file path with the new source
- **resolveSrcPath:** Given an import source and the path of the module, returns the absolute path of the imported module. It returns null for node modules and library imports. It uses **tsconfig** for resolving alias imports
- **replaceObjectReferences:** Identifies the properties of an utility object imported from a target module that are used in the current module. These properties are then imported independently and assigned to the object at the top level of the module.
- **create(Named | Default) import/export:** Functions for creating import/export declarations with given type, kind, specifier and source.