Understanding CommonJS and ES Modules in Node.js: Differences, Similarities, and Limitations

In Node.js, modules are used to organize and reuse code. They allow us to break down large programs into smaller, more manageable pieces. Two of the most common module formats used in Node.js are CommonJS (CJS) and ECMAScript (ES) modules. In this article, we will discuss the differences and similarities between CommonJS and ES modules, how they work, and how Node.js loads them.

CommonJS Modules

CommonJS (CJS) is the module format used by Node.js. It was created to provide a standard for structuring JavaScript code in a modular way. The main idea behind CJS is to use the require() function to load modules.

To create a module in CJS format, we use the module.exports object. This object contains the values that we want to export from the module. For example:

```
// math.js
module.exports.add = function(a, b) {
  return a + b;
};
module.exports.multiply = function(a, b) {
  return a * b;
};

// main.js
const math = require('./math');
console.log(math.add(2, 3)); // 5
console.log(math.multiply(2, 3)); // 6
```

In this example, we define a math module that exports two functions: add and multiply. In the main.js file, we require the math module and use its exported functions.

CommonJS Module Wrapper Function and Arguments

In CommonJS modules, the module code is wrapped in an IIFE (mmediately Invoked Function Expression) that is passed several arguments, including exports, require, module, __filename, and __dirname. This function is sometimes called the "module wrapper" or "module function."

Here is a brief explanation of each of these arguments:

- **exports:** This is an object that is used to export values from the module. Any values that are added to the exports object will be available to other modules that require this module.
- **require:** This is a function that is used to import values from other modules. When a module requires another module using the require function, Node.js will load and execute the required module and return its exports object.
- **module:** This is an object that represents the current module. It contains information about the module, such as its ID, filename, and whether it has been loaded or not. We can use the module object to export values from the module or modify its properties.
- **__filename:** This is a string that contains the absolute path of the current module file.
- **__dirname:** This is a string that contains the absolute path of the directory that contains the current module file.

The module wrapper function provides a way to create a new scope for the module code and to avoid polluting the global scope with module-specific variables and functions. The exports, require, module, __filename, and __dirname arguments are passed to the module wrapper function by Node.js when it loads and executes the module, allowing the module code to interact with the module system and the file system in a controlled and predictable way.

Execution and Returning Exports

In CommonJS modules, the module code is executed synchronously when it is loaded by Node.js. This means that the module code is executed and its exports object is populated before any other code in the application is executed.

When the module code is executed, it can modify the exports object by adding properties or methods to it. These properties and methods become the public interface of the module and can be accessed by other modules that require this module.

For example, consider the following CommonJS module:

```
// math.js
function add(x, y) {
  return x + y;
}

exports.add = add;
```

In this module, the add function is defined and then added to the exports object using the exports.add syntax. When another module requires this module, Node.js will load and execute the math.js module, and the exports object will be populated with the add function. Other modules can then use the add function by requiring the math.js module and accessing the add property of the exports object:

```
// index.js
const math = require('./math.js');

console.log(math.add(2, 3)); // outputs 5
```

In this example, the index.js module requires the math.js module using the require function, and then uses the add function by accessing the add property of the exports object.

Caching

CommonJS modules are cached after they are loaded by Node.js, which means that subsequent requests for the same module will return the cached version of the module rather than loading and executing the module again.

The module cache is an object that is maintained by Node.js and is used to store loaded modules. When a module is loaded by Node.js, its exports object is added to the module cache, along with other metadata about the module, such as its ID, filename, and dependencies.

When another module requires the same module, Node.js checks the module cache to see if the module has already been loaded. If the module is in the cache, Node.js returns the cached version of the module, rather than loading and executing the

module again. This can help to improve the performance of the application by reducing the number of times that modules need to be loaded and executed.

However, it's important to note that the module cache is only effective within a single Node.js process. If the same module is required by different Node.js processes, each process will have its own module cache, and the module will need to be loaded and executed separately by each process.

Additionally, it's important to be aware of the potential issues that can arise from the module cache, such as when a module's state is modified by one module and then reused by another module. To avoid these issues, it's important to design modules that are stateless and that do not modify their own exports object.

Node.js Module Resolution and Interface

When we require a module, Node.js looks for it in the following order:

1. Core modules (built-in modules provided by Node.js)
2. Files or folders with the same name as the module in the current directory
3. If no file found try to find folder with index.js in it
4. Folders in the node_modules directory, starting with the current folder and moving up the folder hierarchy until the module is found

Once the module is found, Node.js reads and executes the code in the module. The module exports an object that contains its public interface. This interface can consist of functions, objects, or values that are accessible to other modules that require this module.

ES Modules

ECMAScript (ES) modules were introduced in ES6 as a new way to structure JavaScript code in a modular way. ES modules use the import and export statements to define and use modules.

In contrast to CommonJS modules, which have no required file extension, ES modules are identified by their file extension. In most cases, ES modules use the .mjs extension, although this can vary depending on the configuration of the environment. When using ES modules in Node.js, we need to specify the .mjs file extension when importing ES modules in our code. However, if we define "type": "module" in package.json, we can use the .js extension for our ES modules instead.

This can make it easier to use ES modules in our code without having to rename all of our files to use the .mjs extension.[1]

ES modules have a different syntax than CJS modules. To create a module in ES format, we use the export keyword to export values from the module. For example:

```
// math.mjs
export function add(a, b) {
  return a + b;
}
export function multiply(a, b) {
  return a * b;
}

// main.mjs
import * as math from './math.mjs';
console.log(math.add(2, 3)); // 5
console.log(math.multiply(2, 3)); // 6
```

In this example, we define a math module using the export keyword to export the add and multiply functions. In the main.mjs file, we use the import statement to import the entire math module and access its exported functions using the math object.

ES Module Interface and Scope

In ES modules, each module is treated as a separate file with its own scope, and there is no need to use a wrapper function to create a new scope for the module code.

In ES modules, we use the import and export statements to define the module's interface and to control which values are exposed to other modules. We can use the import statement to import values from other modules, and the export statement to export values from the current module. The scope of an ES module is determined by

[1] It's important to note that not all Node.js features are available when using ES modules, and some Node.js modules may not work correctly with ES modules. Additionally, the ECMAScript modules implementation in Node.js is still experimental and subject to change, so it's important to keep this in mind when using ES modules in Node.js.

its top-level declarations, which are not automatically added to the global scope like they are in CommonJS modules.

ES modules are designed to be more modular and lightweight than CommonJS modules, and they do not rely on a wrapper function to provide access to the module system and file system. Instead, they use a combination of static analysis and dynamic loading to manage module dependencies and to ensure that modules are loaded only when they are needed. This makes ES modules more efficient and easier to reason about than CommonJS modules, especially in large-scale applications.

Comparing CommonJS and ES Modules

There are several differences and similarities between CommonJS and ES modules. Let's explore them in more detail:

1. **Syntax:** The syntax for defining and using modules is different between the two formats. CJS uses require() and module.exports, while ES uses import and export statements.
2. **Path resolution:** The way that Node.js resolves module paths is different between CJS and ES modules.

Loading, Scope, and Compatibility Differences

For CJS, Node.js looks for the module in the file system using the require() function. It searches for modules in the current directory and in the node_modules directory, following a specific order.

For ES modules, Node.js uses the import statement to load modules. The file path is resolved relative to the current file, and Node.js uses the file extension to determine the module type.

1. **Loading:** CJS modules are loaded synchronously, meaning that the module is loaded and executed before the rest of the code is executed. ES modules are loaded asynchronously, meaning that the module is loaded and executed when it is needed.
2. **Scope:** CJS modules have a single shared scope, which means that all the code in the module shares the same scope. ES modules have a local scope, which means that each module has its own scope and does not share variables with other modules.
3. **Compatibility:** CJS modules are only compatible with Node.js, while ES modules can be used in both Node.js and browsers.

4. **Migration:** Migrating from CJS to ES modules can be difficult because of the differences in syntax and path resolution. However, tools like Babel can be used to transpile ES modules to CJS modules for compatibility with Node.js.

Things you can't do with ES modules

While ES modules have several advantages over CommonJS modules, there are also some limitations to their functionality. Here are some things that cannot be done with ES modules:

1. **Dynamic loading:** ES modules do not support dynamic loading of modules at runtime, unlike CommonJS modules. This means that modules cannot be loaded conditionally or lazily.
2. **Top-level async/await:** ES modules do not support using the async/await syntax at the top level of a module. This is because the top-level code in a module is executed synchronously.
3. **CommonJS interop:** ES modules do not support the require() function, which is used to load CommonJS modules. This can make it difficult to integrate ES modules with existing CommonJS code.
4. **Circular dependencies:** ES modules do not support circular dependencies, which occur when two or more modules depend on each other. This can be a problem if a project has a complex module structure.
5. **Mutable exports:** ES modules have immutable exports, which means that the value of an exported object cannot be changed once it has been exported. This can be a problem if a module needs to export a mutable object.

__dirname and __filename are not defined In ES modules

In ES modules, the __dirname and __filename variables are not defined. These variables are commonly used in CommonJS modules to get the directory name and file name of the current module.

The reason for this is that ES modules are designed to be loaded asynchronously, which means that the module is not loaded until it is needed. This makes it difficult to determine the directory name and file name of the module at the time of module loading, which is when __dirname and __filename are typically used in CommonJS modules.

Instead, ES modules use the import.meta.url property to get the URL of the current module. This property returns a string containing the full URL of the module, including the file name and path.

While it may be inconvenient to not have access to __dirname and __filename in ES modules, it is not a major limitation. There are alternative ways to get the directory and file name of the current module in ES modules, such as using the URL constructor to parse the import.meta.url property.

The way to get around this in ES modules is by using the `fileURLToPath` method imported from the `url` module, and passing `import.meta.url` as an argument to get the `__filename`, and to get the `__dirname` to use the `path.dirname` method, like in this example:

```
import path from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);

const __dirname = path.dirname(__filename);
```

Conclusion

In conclusion, CommonJS and ES modules are two different module formats used in Node.js. CommonJS is the module format used by Node.js, and it uses the require() function to load modules. ES modules were introduced in ES6 and use the import and export statements to define and use modules.

While there are differences in syntax, path resolution, loading, scope, compatibility, and migration, both formats serve the same purpose of organizing and reusing code. It is important to understand the differences between the two formats and to choose the appropriate format based on the requirements of the project.