הבנת CJS Modules ו-ES Modules ב-Node.js
הבדלים, קווי דמיון ומגבלות

ב-Node.js, מודולים משמשים לארגון ושימוש חוזר בקוד. הם מאפשרים לנו לפרק תוכניות גדולות לפיסות קטנות וניתנות לניהול יעיל יותר. שניים מהפורמטים הנפוצים ביותר של מודולים בשימוש ב-Node.js הם Common JS Modules (להלן CJS) ו-ECMAScript Modules (להלן ES).
במאמר זה, נדון בהבדלים ובדמיון בין מודולי CJS ו-ES, כיצד הם פועלים וכיצד Node.js טוענת אותם.

מודולי CJS

CJS הוא פורמט המודול המשמש את Node.js. הוא נוצר כדי לספק תקן לבניית קוד JavaScript בצורה מודולרית. הרעיון המרכזי מאחורי CJS הוא להשתמש בפונקציה require כדי לטעון מודולים.

כדי ליצור מודול אנו משתמשים באובייקט module.exports. אובייקט זה מכיל את הערכים שאנו רוצים לייצא מהמודול. לדוגמה:

```
// 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
```

בדוגמה זו, אנו מגדירים מודול מתמטי המייצא שתי פונקציות: חיבור וכפל. בקובץ main.js, אנו משתמשים במתודת require על מנת לייבא את מודול המתמטיקה ומשתמשים בפונקציות המיוצאות ממנו.

פענוח וממשק (Module Resolution and Interface) של מודול Node.js

כאשר אנו דורשים מודול באמצעות מתודת require, אז Node.js מחפש אותו בסדר הבא:

1. מודולי ליבה (מודולים מובנים מסופקים על ידי Node.js)
2. קבצים או תיקיות בעלי שם זהה למודול בספרייה הנוכחית
3. אם לא נמצא קובץ נסה למצוא תיקייה עם index.js בתוכה
4. תיקיות בספריית node_modules, החל מהתיקייה הנוכחית ומעבר בהיררכיית התיקיות עד למציאת המודול

ברגע שהמודול נמצא, Node.js קורא ומבצע את הקוד במודול. המודול מייצא אובייקט המכיל את הממשק הציבורי שלו. ממשק זה יכול להיות מורכב מפונקציות, אובייקטים או ערכים הנגישים למודולים אחרים הדורשים מודול זה.

Module Wrapper Function וארגומנטים

במודולי CJS, קוד המודול עטוף בפונקצית IIFE (ר"ת Immediately Invoked Function Expression) עם מספר ארגומנטים:
exports, require, module, __filename ו-__dirname. פונקציה זו נקראת לפעמים "module wrapper" או "module function".

```
(function (exports, require, module, __filename, __dirname) {
  // Module code lives here ...
});
```

להלן הסבר קצר על כל אחד מהארגומנטים הללו:

- **exports:** זהו אובייקט המשמש לייצוא ערכים מהמודול. כל הערכים שנוספו לאובייקט הייצוא יהיו זמינים למודולים אחרים הדורשים מודול זה.
- **require:** זוהי פונקציה המשמשת לייבוא ערכים ממודולים אחרים. כאשר מודול דורש מודול אחר המשתמש בפונקציה require, Node.js יטען ויבצע את המודול הנדרש ויחזיר את אובייקט הייצוא שלו.
- **module:** זהו אובייקט המייצג את המודול הנוכחי. הוא מכיל מידע על המודול, כגון המזהה שלו, שם הקובץ והאם הוא נטען או לא. אנו יכולים להשתמש באובייקט המודול כדי לייצא ערכים מהמודול או לשנות את המאפיינים שלו.
- **__filename:** זוהי מחרוזת המכילה את הנתיב המוחלט של קובץ המודול הנוכחי.
- **__dirname:** זוהי מחרוזת המכילה את הנתיב המוחלט של הספרייה המכילה את קובץ המודול הנוכחי.

ה-wrapper function של המודול מספקת דרך ליצור סקופ חדש עבור קוד כל מודול ולהימנע מלזהם את הסקופ הגלובלי עם משתנים ופונקציות ספציפיות למודול. ארגומנטי ה-exports, require, module, __filename ו__dirname מועברים ל-wrapper function על ידי Node.js כאשר היא טוענת ומפעילה את המודול, מה שמאפשר לקוד המודול לקיים אינטראקציה עם מערכת המודול ומערכת הקבצים בצורה מבוקרת וניתנת לחיזוי.

Execution and Returning Exports

במודולי CJS, קוד המודול מבוצע באופן סינכרוני כאשר הוא נטען על ידי Node.js. המשמעות היא שקוד המודול מבוצע ואובייקט הייצוא שלו מאוכלס לפני ביצוע קוד אחר באפליקציה.

כאשר קוד המודול מבוצע, הוא יכול לשנות את אובייקט הייצוא על ידי הוספת מאפיינים או שיטות אליו. מאפיינים ושיטות אלו הופכים לממשק הציבורי של המודול וניתן לגשת אליהם על ידי מודולים אחרים הדורשים מודול זה.

לדוגמה, נבחן את המודול הבא:

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

exports.add = add;
```

במודול זה, פונקציית ה-add מוגדרת ולאחר מכן מתווספת לאובייקט ה-exports באמצעות תחביר exports.add. כאשר מודול אחר דורש (require) מודול זה, Node.js יטען ויבצע את המודול math.js, ואובייקט הייצוא יאוכלס בפונקציית ה-add. לאחר מכן מודולים אחרים יכולים להשתמש בפונקציית add על ידי דרישת המודול math.js וגישה למאפיין ה-add של אובייקט הייצוא:

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

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

בדוגמה זו, מודול index.js דורש את מודול math.js באמצעות הפונקציה require, ולאחר מכן משתמש בפונקציה add על ידי גישה למאפיין ה-add של אובייקט הייצוא.

שמירה במטמון (Caching)

מודולי CJS נשמרים במטמון לאחר טעינתם על ידי Node.js, מה שאומר שבקשות עוקבות עבור אותו מודול יחזירו את הגרסה השמורה של המודול במקום לטעון ולהפעיל את המודול שוב.

מטמון המודול הוא אובייקט שמתוחזק על ידי Node.js ומשמש לאחסון מודולים טעונים. כאשר מודול נטען על ידי Node.js, אובייקט הייצוא שלו מתווסף למטמון המודול, יחד עם מטא נתונים אחרים על המודול, כגון המזהה, שם הקובץ והתלות שלו.

כאשר מודול אחר דורש (requires) את אותו מודול, Node.js בודק את מטמון המודול כדי לראות אם המודול כבר נטען. אם המודול נמצא במטמון, Node.js מחזיר את הגרסה השמורה של המודול,

במקום לטעון ולהפעיל את המודול שוב. זה יכול לעזור לשפר את הביצועים של האפליקציה על ידי הפחתת מספר הפעמים שיש לטעון ולהפעיל מודולים.

עם זאת, חשוב לציין כי מטמון המודול יעיל רק בתוך תהליך Node.js בודד. אם אותו מודול נדרש על ידי תהליכי Node.js שונים, לכל תהליך יהיה מטמון מודול משלו, והמודול יצטרך להיטען ולהפעיל בנפרד על ידי כל תהליך.

בנוסף, חשוב להיות מודעים לבעיות הפוטנציאליות שעלולות לנבוע ממטמון המודול, כגון כאשר הסטייט של המודול משתנה על ידי מודול אחד ולאחר מכן נעשה בו שימוש חוזר על ידי מודול אחר. כדי להימנע מבעיות אלו, חשוב לעצב מודולים שהם חסרי סטייט ושאינם משנים את אובייקט הייצוא שלהם.

ES Modules

מודולי ES הוצגו ב-ES6 כדרך חדשה להשתמש בקוד JavaScript בצורה מודולרית. מודולי ES משתמשים בהצהרות import ו-export כדי להגדיר ולהשתמש במודולים

בניגוד למודולי CJS, שאין להם סיומת קובץ נדרשת, מודולי ES מזוהים לפי סיומת הקובץ שלהם. ברוב המקרים, מודולי ES משתמשים בסיומת .mjs, אם כי זה יכול להשתנות בהתאם לתצורת הסביבה. בעת שימוש במודולי ES ב-Node.js, עלינו לציין את סיומת הקובץ .mjs בעת ייבוא מודולי ES בקוד שלנו. עם זאת, אם נגדיר "type": "module" ב-package.json, נוכל להשתמש בסיומת .js עבור מודולי ה-ES שלנו במקום זאת. זה יכול להקל על השימוש במודולי ES בקוד שלנו מבלי לשנות את שמות כל הקבצים שלנו כדי להשתמש בסיומת .mjs.[1]

למודולי ES יש תחביר שונה ממודולי CJS. כדי ליצור מודול בפורמט ES, אנו משתמשים במילת המפתח export כדי לייצא ערכים מהמודול. לדוגמה:

```
// 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
```

[1] חשוב לציין שלא כל התכונות של Node.js זמינות בעת שימוש במודולי ES, וייתכן שחלק ממודולי Node.js לא יפעלו כהלכה עם מודולי ES. בנוסף, יישום מודולי ECMAScript ב-Node.js עדיין ניסיוני ונתון לשינויים, לכן חשוב לזכור זאת בעת שימוש במודול ES ב-Node.js.

בדוגמה זו, אנו מגדירים מודול מתמטי באמצעות מילת המפתח export כדי לייצא את פונקציות ההוספה והכפל. בקובץ main.mjs, אנו משתמשים במשפט import כדי לייבא את כל מודול המתמטיקה ולגשת לפונקציות המיוצאות שלו באמצעות האובייקט המתמטי.

ממשק וסקופ של מודולי ES

במודולי ES, כל מודול מטופל כקובץ נפרד עם סקופ משלו, ואין צורך להשתמש בפונקציית wrapper כדי ליצור סקופ חדש עבור קוד המודול. במודולי ES, אנו משתמשים בהצהרות ה-export ו-import כדי להגדיר את ממשק המודול וכדי לשלוט אילו ערכים חשופים למודולים אחרים. אנו יכולים להשתמש במשפט ה-import כדי לייבא ערכים ממודולים אחרים, ובמשפט ה-export לייצא ערכים מהמודול הנוכחי. הסקופ של מודול ES נקבע על ידי ההצהרות ברמה העליונה שלו, שאינן מתווספות אוטומטית לסקופ הגלובלי כמו במודולי CJS.

מודולי ES מתוכננים להיות מודולריים וקלים יותר ממודולי CJS, והם אינם מסתמכים על פונקציית wrapper כדי לספק גישה למערכת המודולים ולמערכת הקבצים. במקום זאת, הם משתמשים בשילוב של ניתוח סטטי וטעינה דינמית כדי לנהל תלות של מודולים וכדי להבטיח שהמודולים נטענים רק כאשר הם נחוצים. זה הופך את מודולי ES ליעילים יותר וקלים יותר להבנה מאשר מודולי CJS, במיוחד ביישומים בקנה מידה גדול.

הבדלי טעינה, Scope ותאימות

עבור מודלי CJS סביבת Node.js תחפש את המודול במערכת הקבצים באמצעות הפונקציה require. היא תחפש מודולים בספרייה הנוכחית ובספריית node_modules, לפי סדר מסוים.

עבור מודולי ES, סביבת Node.js תשתמש במשפט import כדי לטעון מודולים. נתיב הקובץ מפוענח ביחס לקובץ הנוכחי, ו-Node.js משתמש בסיומת הקובץ כדי לקבוע את סוג המודול, לכן חובה לציין את סיומת הקובץ כאשר מייבאים קבצים מקומיים.

1. **טעינה:** מודולי CJS נטענים באופן סינכרוני, כלומר המודול נטען ומבוצע לפני ביצוע שאר הקוד. מודולי ES נטענים באופן אסינכרוני, כלומר המודול נטען ומבוצע בעת הצורך.
2. **Scope:** למודולי CJS יש סקופ משותף אחד, כלומר כל הקוד במודול חולק את אותו סקופ. למודולי ES יש סקופ מקומי, כלומר לכל מודול יש סקופ משלו ואינו חולק משתנים עם מודולים אחרים.
3. **תאימות:** מודולי CJS תואמים רק ל-Node.js, בעוד שניתן להשתמש במודולי ES גם ב-Node.js וגם בדפדפנים.

דברים שלא ניתן לעשות עם מודולי ES

בעוד שלמודולי ES יש כמה יתרונות על פני מודולי CJS, יש גם כמה מגבלות לפונקציונליות שלהם. הנה כמה דברים שלא ניתן לעשות עם מודולי ES:

1. **טעינה דינמית:** מודולי ES אינם תומכים בטעינה דינמית של מודולים בזמן ריצה, בניגוד למודולי CJS. המשמעות היא שלא ניתן לטעון מודולים באופן מותנה או lazy loading.
2. **async/await בטופ לבל:** מודולי ES אינם תומכים בשימוש בתחביר async/await בטופ לבל של המודול. הסיבה לכך היא שהקוד בטופ לבל במודול מבוצע באופן סינכרוני.

3. **תלות מעגלית:** מודולי ES אינם תומכים בתלות מעגלית, המתרחשות כאשר שני מודולים או יותר תלויים זה בזה. זו יכולה להיות בעיה אם לפרויקט יש מבנה מודול מורכב.
4. **exports ניתנים לשינוי (Mutable):** למודולי ES יש exports בלתי ניתנים לשינוי (immutable), מה שאומר שלא ניתן לשנות את הערך של אובייקט מיוצא לאחר הייצוא. זו יכולה להיות בעיה אם מודול צריך לייצא אובייקט שצריך לשנות.

__dirname ו-__filename אינם מוגדרים במודולי ES

במודולי ES, המשתנים __dirname ו__filename אינם מוגדרים. משתנים אלה נמצאים בשימוש נפוץ במודולי CJS כדי לקבל את שם הספרייה ושם הקובץ של המודול הנוכחי.

הסיבה לכך היא שמודולי ES מתוכננים להיטען באופן אסינכרוני, כלומר המודול אינו נטען עד לרגע בו הוא נצרך. זה מקשה על קביעת שם הספרייה ושם הקובץ של המודול בזמן טעינת המודול.

למרות שזה עשוי להיות לא נוח שאין גישה ל-__dirname ו-__filename במודולי ES, אין זו מגבלה גדולה. ישנן דרכים חלופיות לקבל את הספרייה ואת שם הקובץ של המודול הנוכחי במודולי ES, כגון שימוש בקונסטרקטור URL כדי לנתח את המאפיין import.meta.url.

מודולי ES משתמשים במאפיין import.meta.url כדי לקבל את כתובת האתר של המודול הנוכחי. מאפיין זה מחזיר מחרוזת המכילה את כתובת האתר המלאה של המודול, כולל שם הקובץ והנתיב.

הדרך לעקוף את זה במודולי ES היא על ידי שימוש במתודה `fileURLToPath` המיובאת מהמודול `url`, והעברת `import.meta.url` כארגומנט כדי לקבל את ה- `__filename`, וכדי להביא את `__dirname` ניתן להתשמש במתודה `path.dirname`, כמו בדוגמה זו:

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

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

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

סיכום

לסיכום, מודולי CJS ו-ES הם שני פורמטים שונים של מודולים המשמשים ב-Node.js. CJS הוא פורמט המודול המשמש את Node.js, והוא משתמש בפונקציה require כדי לטעון מודולים. מודולי ES הוצגו ב-ES6 ומשתמשים בהצהרות import ו-export כדי להגדיר ולהשתמש במודולים.

בעוד שיש הבדלים בתחביר, בפיענוח הנתיב, טעינה, סקופ ותאימות, שני הפורמטים משרתים את אותה מטרה של ארגון ושימוש חוזר בקוד. חשוב להבין את ההבדלים בין שני הפורמטים ולבחור את הפורמט המתאים בהתאם לדרישות הפרויקט.