Question: Ứng dụng nodejs chạy bao nhiêu core & thread của cpu?

Ứng dụng Node.js chạy trên **một luồng đơn (single thread) của một core CPU** theo mô hình **event-driven, non-blocking I/O**. Tuy nhiên, nó có thể tận dụng nhiều core bằng cách sử dụng các phương pháp sau:

1. Mặc định: Chạy trên một core, một thread

- JavaScript trong Node.js chạy trên một luồng duy nhất trong tiến trình chính.
- Node.js sử dụng **libuv** để xử lý I/O bất đồng bộ bằng một **thread pool** (thường có 4 thread mặc định) để xử lý các tác vụ như:
  - File system (đọc/ghi file)
  - Mã hóa/giải mã
  - Network requests (DNS lookup, SSL handshake)

2. Tận dụng nhiều core bằng Cluster Module

- Module cluster giúp chạy nhiều instance của ứng dụng trên các core khác nhau.
- Mỗi instance là một tiến trình con (worker process) giao tiếp với tiến trình chính (master process).

Ví dụ chạy ứng dụng trên tất cả các core:

```
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
    const numCPUs = os.cpus().length;
    console.log(`Số core CPU: ${numCPUs}`);

    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('exit', (worker) => {
        console.log(`Worker ${worker.process.pid} đã thoát`);
    });
} else {
    require('./app'); // Chạy ứng dụng của bạn
}
```

3. Sử dụng Worker Threads để chạy đa luồng

- Từ Node.js 10 trở lên, module worker_threads cho phép tạo nhiều thread chạy song song.

Ví dụ tạo thread xử lý công việc nặng:

```
const { Worker, isMainThread, parentPort } = require('worker_threads');

if (isMainThread) {
    const worker = new Worker(__filename);
    worker.on('message', (msg) => console.log(`Message from worker: ${msg}`));
} else {
    parentPort.postMessage('Hello from worker');
}
```

Tóm lại

- Mặc định, Node.js chỉ chạy trên **1 core, 1 thread chính**.
- Có thể mở rộng ra **nhiều core** bằng cluster hoặc worker_threads.
- Nếu ứng dụng có tác vụ CPU nặng, nên dùng worker_threads hoặc các dịch vụ microservice.

Question: Tôi có 1 cpu với cấu hình 4 core và 16 thread. Ứng dụng nodejs sẽ chạy như thế nào trên cpu này

Với CPU có **4 core và 16 thread**, Node.js sẽ chạy như sau:

1. Mặc định: Node.js chạy trên 1 core, 1 thread chính

- Khi bạn chạy một ứng dụng Node.js **bình thường** (không dùng cluster hoặc worker_threads), nó chỉ sử dụng **1 thread chính trên 1 core** của CPU.
- Các tác vụ **I/O, file system, networking** sẽ được xử lý bởi **thread pool** của **libuv** (mặc định có 4 thread, có thể tăng lên bằng biến môi trường UV_THREADPOOL_SIZE).

☞ **Hiệu suất bị giới hạn bởi 1 core của CPU, dù CPU có 16 thread.**

---

2. Nếu dùng Cluster Module: Chạy trên 4 core (4 process)

- Module cluster giúp bạn nhân bản tiến trình (process) của ứng dụng để tận dụng **4 core** của CPU.
- Mỗi tiến trình **sử dụng 1 thread chính**, nhưng do có 4 process nên ứng dụng sẽ **tận dụng được 4 core**.

Ví dụ:

```
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
    const numCPUs = os.cpus().length; // Trả về 4 (vì có 4 core vật lý)
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }
} else {
    require('./app'); // Chạy ứng dụng
}
```

☞ Lúc này, Node.js sẽ tận dụng 4 core, nhưng mỗi core vẫn chỉ chạy 1 thread chính của Node.js.

3. Nếu dùng Worker Threads: Chạy đa luồng trên tất cả 16 thread

- worker_threads giúp bạn tạo nhiều **thread** trong cùng một process.
- Bạn có thể tạo tối đa **16 thread**, giúp tận dụng toàn bộ **16 thread của CPU**.

Ví dụ:

```
const { Worker, isMainThread, parentPort } = require('worker_threads');

if (isMainThread) {
    for (let i = 0; i < 16; i++) { // Tạo 16 worker
        const worker = new Worker(__filename);
        worker.on('message', (msg) => console.log(`Worker ${i}: ${msg}`));
    }
} else {
    parentPort.postMessage('Hello from worker');
}
```

☞ Lúc này, ứng dụng có thể tận dụng toàn bộ 16 thread của CPU.

☞ Phù hợp cho các tác vụ tính toán nặng như AI, xử lý dữ liệu lớn.

Tóm tắt

Cách chạy	Số core sử dụng	Số thread sử dụng
Mặc định	1 core	1 thread
Cluster	4 core	4 thread (4 process)

Cách chạy	Số core sử dụng	Số thread sử dụng
Worker Threads	4 core	16 thread (1 process, đa luồng)

📌 **Nếu ứng dụng của bạn là web server (Express, NestJS, Fastify)**, dùng cluster là đủ.
📌 **Nếu ứng dụng cần xử lý nặng (AI, video encoding, crypto)**, dùng worker_threads để tận dụng 16 thread.

Question: Javascript hoạt động như thế nào trên trình duyệt

JavaScript trên trình duyệt hoạt động dựa trên mô hình **Single Thread (luồng đơn)** với **Event Loop (vòng lặp sự kiện)** để xử lý các tác vụ bất đồng bộ một cách hiệu quả. Dưới đây là cách JavaScript hoạt động trên trình duyệt:

---

1. Kiến trúc JavaScript Runtime trên trình duyệt

Một trình duyệt hiện đại như Chrome, Firefox, Edge... cung cấp **môi trường runtime** cho JavaScript bao gồm:

(1) Call Stack (Ngăn xếp lời gọi)

- Là nơi thực thi các lệnh đồng bộ.
- JavaScript chạy từng dòng mã một, theo mô hình **LIFO (Last In, First Out)**.

(2) Web APIs (APIs của trình duyệt)

- Cung cấp các tính năng như:
  - setTimeout, setInterval (Timer)
  - fetch, XMLHttpRequest (HTTP request)
  - DOM API (Truy cập HTML)
  - console.log, localStorage
  - Geolocation, WebRTC, WebSockets...

(3) Callback Queue (Hàng đợi Callback)

- Các tác vụ bất đồng bộ sẽ được đưa vào hàng đợi chờ **Event Loop** xử lý.

(4) Event Loop (Vòng lặp sự kiện)

- Kiểm tra **Call Stack** có rảnh không. Nếu rảnh, nó lấy các tác vụ từ **Callback Queue** đưa vào **Call Stack** để thực thi.

---

2. Cách JavaScript thực thi mã (Single Thread)

Ví dụ với đoạn mã sau:

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout");
}, 2000);

console.log("End");
```

Trình tự thực thi:

1. **Call Stack** chạy console.log("Start") → in "Start".
2. **setTimeout** được gửi đến **Web APIs** và hẹn giờ 2 giây.
3. **console.log("End")** chạy ngay → in "End".
4. Sau 2 giây, **setTimeout callback** được đẩy vào **Callback Queue**.
5. **Event Loop** kiểm tra Call Stack rảnh → đẩy callback vào Call Stack → in "Timeout".

📌 **Kết quả in ra:**

```
Start
End
Timeout
```

3. JavaScript bất đồng bộ với Web APIs

Các tác vụ như **AJAX, setTimeout, Event Listener** không chạy trực tiếp trong Call Stack mà được trình duyệt xử lý.

Ví dụ với **fetch API**:

```
console.log("Start");

fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(response => response.json())
  .then(data => console.log(data));
```

```
console.log("End");
```

📌 **Thứ tự in ra:**

```
Start
End
{ id: 1, title: "delectus aut autem", completed: false }
```

Lý do:

- fetch được gửi đến Web APIs, không chặn JavaScript.
- console.log("End") chạy ngay.
- Khi fetch hoàn thành, nó đưa callback vào Callback Queue.
- Event Loop đẩy callback vào Call Stack và in ra dữ liệu JSON.

---

4. Microtask Queue (Promise, MutationObserver)

Có 2 loại hàng đợi trong JavaScript:

- **Macrotask Queue**: chứa setTimeout, setInterval, setImmediate, I/O, MessageChannel.
- **Microtask Queue**: chứa Promises, process.nextTick (Node.js), MutationObserver**.**

💡 Microtasks luôn được ưu tiên trước Macrotasks.
Ví dụ:

```
console.log("Start");

setTimeout(() => console.log("setTimeout"), 0);

Promise.resolve().then(() => console.log("Promise"));

console.log("End");
```

📌 **Kết quả in ra:**

```
Start
End
Promise
setTimeout
```

- Promise chạy trước setTimeout vì nó nằm trong **Microtask Queue**.
- setTimeout nằm trong **Macrotask Queue** nên chạy sau.

---

5. JavaScript có thật sự chạy đơn luồng?

JavaScript **bản chất** là **single-threaded**, nhưng trình duyệt cung cấp Web APIs để xử lý bất đồng bộ song song.
Nhờ **Event Loop**, JavaScript có thể: ✅ Chạy I/O song song (fetch, AJAX).
✅ Không bị block bởi các tác vụ chậm.
✅ Xử lý hàng nghìn sự kiện UI mà không bị treo.

Tóm lại

- JavaScript trên trình duyệt chạy trên **Call Stack (Single Thread)**.
- Bất đồng bộ nhờ **Web APIs** và **Event Loop**.
- **Callback Queue & Microtask Queue** quyết định thứ tự thực thi.
- **Promise** chạy trước **setTimeout** nhờ Microtask Queue.

Question: Có thể thêm promise trong setTimeout không? và ngược lại.

Có thể sử dụng **Promise bên trong setTimeout** và cũng có thể **đặt setTimeout bên trong Promise**. Tuy nhiên, cách chúng hoạt động sẽ khác nhau do cơ chế **Microtask Queue** và **Macrotask Queue** của JavaScript.

1. Đặt Promise trong setTimeout

```
setTimeout(() => {
    console.log("setTimeout bắt đầu");

    new Promise((resolve) => {
        console.log("Promise trong setTimeout");
        resolve();
    }).then(() => console.log("Promise resolved"));

    console.log("setTimeout kết thúc");
}, 1000);
```

📌 **Kết quả in ra (sau 1 giây):**

```
setTimeout bắt đầu
Promise trong setTimeout
setTimeout kết thúc
Promise resolved
```

🔍 **Giải thích:**

- Khi setTimeout chạy, nó thực thi tất cả lệnh bên trong.
- console.log("Promise trong setTimeout") chạy ngay.
- Promise.resolve() được đẩy vào **Microtask Queue**.
- console.log("setTimeout kết thúc") chạy.
- Sau đó, **Microtask Queue** được xử lý, nên "Promise resolved" in ra cuối cùng.

💡 **Promise trong setTimeout sẽ được giải quyết ngay lập tức nếu không có await hoặc tác vụ async khác.**

---

2. Đặt setTimeout trong Promise

```
new Promise((resolve) => {
    console.log("Promise bắt đầu");

    setTimeout(() => {
        console.log("setTimeout trong Promise");
        resolve("Promise hoàn thành");
    }, 1000);
}).then((message) => console.log(message));

console.log("Promise kết thúc");
```

📌 **Kết quả in ra ngay lập tức:**

```
Promise bắt đầu
Promise kết thúc
(setTimeout chờ 1 giây)
setTimeout trong Promise
Promise hoàn thành
```

🔍 **Giải thích:**

- "Promise bắt đầu" in ra đầu tiên.
- setTimeout được gửi vào **Macrotask Queue** và hẹn giờ 1 giây.
- "Promise kết thúc" in ra ngay vì đoạn code tiếp theo của Promise vẫn là đồng bộ.
- Sau 1 giây, setTimeout thực thi "setTimeout trong Promise".
- resolve("Promise hoàn thành") gọi then(), đưa callback vào **Microtask Queue**.
- "Promise hoàn thành" in ra.

💡 **Promise chỉ hoàn thành sau khi setTimeout kết thúc.**

---

3. Promise vs setTimeout – Cái nào chạy trước?

```
setTimeout(() => console.log("setTimeout"), 0);

Promise.resolve().then(() => console.log("Promise"));
```

📌 **Kết quả in ra:**

```
Promise
setTimeout
```

🔍 **Giải thích:**

- setTimeout(..., 0) được gửi vào **Macrotask Queue**.
- Promise.resolve().then(...) được đưa vào **Microtask Queue**.
- **Microtask Queue được ưu tiên trước Macrotask Queue**, nên "Promise" in ra trước.

Tóm lại

Cách kết hợp	Kết quả
Promise trong setTimeout	Promise sẽ chạy **sau** setTimeout nhưng trước khi setTimeout kết thúc.
setTimeout trong Promise	setTimeout trì hoãn việc resolve Promise.
Promise vs setTimeout(0)	Promise luôn chạy trước vì **Microtask Queue** ưu tiên hơn **Macrotask Queue**.

Question: Hoisting trong javascript là gì? cách phòng tránh

Hoisting trong JavaScript là gì?

Hoisting là cơ chế trong JavaScript giúp **đưa khai báo biến và hàm lên đầu phạm vi (scope) trước khi mã được thực thi**. Điều này có nghĩa là bạn có thể sử dụng biến hoặc hàm **trước khi khai báo chúng trong code** mà không gặp lỗi.

---

1. Hoisting với **var**, **let**, **const**

(1) var bị hoisted nhưng giá trị là undefined

```
console.log(a); // undefined
var a = 5;
console.log(a); // 5
```

🔍 Giải thích:

- JavaScript **hoisting** var a; lên đầu, nhưng không hoisting giá trị 5.
- Do đó, console.log(a) in undefined thay vì báo lỗi.

Thực chất mã trên được máy tính hiểu như sau:

```
var a;
console.log(a); // undefined
a = 5;
console.log(a); // 5
```

(2) let và const bị hoisted nhưng không thể sử dụng trước khi khai báo

```
console.log(b); // ✗ ReferenceError: Cannot access 'b' before initialization
let b = 10;
```

💡 Lý do:

- let và const cũng bị **hoisted**, nhưng chúng nằm trong **Temporal Dead Zone (TDZ)**, nơi mà biến **không thể truy cập trước khi khai báo**.
- Điều này giúp tránh lỗi logic như với var.

Tương tự với const:

```
console.log(c); // ✕ ReferenceError
const c = 20;
```

2. Hoisting với function declaration và function expression

(1) Function Declaration bị hoisted đầy đủ

```
sayHello(); // ✓ "Hello!"

function sayHello() {
    console.log("Hello!");
}
```

🔍 Giải thích:

- **Toàn bộ** function được hoisted lên đầu, nên có thể gọi trước khi khai báo.

---

(2) Function Expression không bị hoisted đầy đủ

```
greet(); // ✕ TypeError: greet is not a function

var greet = function() {
    console.log("Hi!");
};
```

🔍 Giải thích:

- var greet được hoisted lên đầu với giá trị mặc định undefined.
- Khi gọi greet(), nó vẫn chưa có giá trị là function, nên bị lỗi.

💡 Tương tự với let và const, nhưng lỗi sẽ là ReferenceError do TDZ:

```
greet(); // ✕ ReferenceError

let greet = function() {
    console.log("Hi!");
};
```

3. Cách phòng tránh lỗi do hoisting

✅ 1. Luôn khai báo biến trước khi sử dụng

Thay vì:

```
console.log(name); // ❌ undefined
var name = "Alice";
```

Hãy viết:

```
var name = "Alice";
console.log(name); // ✅ "Alice"
```

✅ 2. Dùng let và const thay vì var

- Tránh var để không bị lỗi undefined khi truy cập biến trước khai báo.
- let và const giúp phát hiện lỗi sớm do chúng không thể truy cập trước khi khai báo.

Ví dụ tốt:

```
let age = 25;
console.log(age); // ✅ 25
```

✅ 3. Định nghĩa function trước khi gọi (đối với function expression)

Thay vì:

```
sayHi(); // ❌ TypeError
var sayHi = function() {
    console.log("Hi!");
};
```

Hãy viết:

```
const sayHi = function() {
    console.log("Hi!");
};
sayHi(); // ✅ "Hi!"
```

Tóm lại

Loại khai báo	Bị hoisting?	Giá trị khi hoisting
var	✅ Có	undefined
let	✅ Có	❌ Lỗi (TDZ)
const	✅ Có	❌ Lỗi (TDZ)
Function Declaration	✅ Có	Toàn bộ function được hoisted
Function Expression (dùng var)	✅ Có	undefined (lỗi khi gọi trước khai báo)
Function Expression (dùng let/const)	✅ Có	❌ Lỗi (TDZ)

📌 **Nên dùng let, const và khai báo trước khi sử dụng để tránh lỗi hoisting!** 🚀

Question: var, let, const là gì? so sánh

1. var – Biến kiểu cũ (nên tránh dùng)

Đặc điểm:

- **Hoisting**: Bị hoisted với giá trị undefined.
- **Function Scope**: Chỉ giới hạn trong hàm chứa nó.
- **Không có Block Scope**: Không bị giới hạn trong {}.
- **Có thể gán lại (reassign) và khai báo lại (redeclaration)**.

Ví dụ:

```
console.log(a); // ✅undefined (do hoisting)
var a = 10;

if (true) {
    var b = 20;
}
console.log(b); // ✅20 (không có block scope)

var c = 30;
var c = 40; // ✅Không báo lỗi
console.log(c); // 40
```

🔍 **Lý do tránh dùng var**:

- Có thể vô tình ghi đè biến (redeclare).
- Không có block scope → Dễ gây lỗi logic.

2. let – Biến có thể thay đổi giá trị (mutable)

Đặc điểm:

- **Hoisting**: Bị hoisted nhưng không thể sử dụng trước khi khai báo (**TDZ – Temporal Dead Zone**).
- **Block Scope**: Chỉ tồn tại trong {} chứa nó.
- **Có thể gán lại (reassign) nhưng không thể khai báo lại (redeclaration)**.

Ví dụ:

```
console.log(x); // ❌ ReferenceError (TDZ)
let x = 10;

if (true) {
    let y = 20;
}
console.log(y); // ❌ ReferenceError (block scope)

let z = 30;
z = 40; // ✅ Cho phép gán lại
console.log(z); // 40

let z = 50; // ❌ SyntaxError (không thể khai báo lại)
```

🔍 **Khi nào dùng let?**

- Khi biến cần thay đổi giá trị nhưng không muốn cho phép khai báo lại.

---

3. const – Biến hằng số (immutable)

Đặc điểm:

- **Hoisting**: Bị hoisted nhưng nằm trong **TDZ**.
- **Block Scope**: Giống let, chỉ tồn tại trong {} chứa nó.
- **Không thể gán lại (reassign) và không thể khai báo lại (redeclaration)**.

Ví dụ:

```
console.log(a); // ✗ ReferenceError (TDZ)
const a = 10;

if (true) {
    const b = 20;
}
console.log(b); // ✗ ReferenceError (block scope)

const c = 30;
c = 40; // ✗ TypeError (không thể gán lại)

const d; // ✗ SyntaxError (const phải gán giá trị ngay lập tức)
```

🔍 **Lưu ý quan trọng**:

- const chỉ ngăn chặn thay đổi tham chiếu của biến, nhưng **đối tượng và mảng có thể thay đổi nội dung**.

```
const obj = { name: "Alice" };

obj.name = "Bob"; // ✓ Hợp lệ (thay đổi nội dung object)
console.log(obj); // { name: "Bob" }

obj = { name: "Charlie" }; // ✗ TypeError (không thể gán lại obj)
```

🔍 **Khi nào dùng const?**

- Khi biến không cần thay đổi giá trị.
- Dùng const càng nhiều càng tốt để tránh lỗi logic.

---

4. So sánh tổng hợp

Đặc điểm	var	let	const
Hoisting	✓ Có (giá trị undefined)	✓ Có (TDZ)	✓ Có (TDZ)
Scope	Function Scope	Block Scope	Block Scope

Đặc điểm	var	let	const
Có thể khai báo lại (redeclare)?	✅ Có	❌ Không	❌ Không
Có thể gán lại (reassign)?	✅ Có	✅ Có	❌ Không
Bắt buộc gán giá trị khi khai báo?	❌ Không	❌ Không	✅ Có
Dùng cho hằng số?	❌ Không	❌ Không	✅ Có
Ảnh hưởng đến hiệu suất?	Không đáng kể	Không đáng kể	Không đáng kể

---

5. Nên dùng var, let hay const?

✅ **Luôn ưu tiên const** – Nếu biến không thay đổi giá trị.
✅ **Dùng let** – Nếu cần thay đổi giá trị của biến.
❌ **Tránh dùng var** – Vì dễ gây lỗi, không có block scope.

💡 **Quy tắc chung**:

1. **Mặc định dùng const** để giảm lỗi.
2. **Chỉ dùng let khi thực sự cần thay đổi giá trị**.
3. **Tránh var hoàn toàn**.

📌 **Tóm lại**:

- **const**: Dùng cho hằng số hoặc biến không thay đổi.
- **let**: Dùng khi cần thay đổi giá trị của biến.
- **var**: Không nên dùng, vì dễ gây lỗi.

🚀 **Code sạch hơn, ít lỗi hơn khi dùng let & const đúng cách!**

Question: Scope trong JavaScript là gì?

Scope (phạm vi) trong JavaScript là **vùng mà một biến hoặc hàm có thể được truy cập**. Nó quyết định biến có thể được sử dụng ở đâu trong code.

---

1. Các loại Scope trong JavaScript

JavaScript có 3 loại scope chính:

1. **Global Scope** (Phạm vi toàn cục)
2. **Function Scope** (Phạm vi hàm)
3. **Block Scope** (Phạm vi khối)

Ngoài ra, còn có: 4. **Lexical Scope** (Phạm vi từ vựng) 5. **Module Scope** (Phạm vi module)

---

2. Chi tiết các loại Scope

(1) Global Scope – Phạm vi toàn cục

📌 Biến được khai báo ngoài mọi hàm hoặc khối sẽ có phạm vi toàn cục.
📌 Có thể truy cập ở mọi nơi trong code.

```
var globalVar = "Tôi là biến toàn cục";

function test() {
    console.log(globalVar); // ✅ Có thể truy cập
}

test();
console.log(globalVar); // ✅ Có thể truy cập
```

🔍 **Lưu ý**:

- Biến toàn cục có thể bị thay đổi ở bất kỳ đâu, dễ gây lỗi.
- Trong trình duyệt, biến toàn cục trở thành thuộc tính của **window**.

```
console.log(window.globalVar); // "Tôi là biến toàn cục"
```

(2) Function Scope – Phạm vi hàm

📌 Biến khai báo bằng var trong một hàm chỉ có thể truy cập bên trong hàm đó.
📌 Không thể truy cập từ bên ngoài hàm**.**

```
function myFunction() {
    var localVar = "Tôi là biến trong hàm";
    console.log(localVar); // ✅ Có thể truy cập
}
```

```
myFunction();
console.log(localVar); // ✗ ReferenceError (không thể truy cập ngoài hàm)
```

🔍 **Lưu ý**:

- var có **function scope**, nghĩa là nếu khai báo trong hàm thì chỉ dùng trong hàm đó.

---

(3) Block Scope – Phạm vi khối (Chỉ áp dụng cho let và const)

📌 Biến khai báo bằng **let** hoặc **const** trong **{}** chỉ có thể truy cập trong khối đó.
📌 Không thể truy cập từ bên ngoài khối.

```
if (true) {
    let blockVar = "Tôi chỉ tồn tại trong khối";
    console.log(blockVar); // ✓ Có thể truy cập
}

console.log(blockVar); // ✗ ReferenceError (ngoài phạm vi khối)
```

🔍 **Lưu ý**:

- let và const có **block scope**.
- var **không có block scope**, có thể gây lỗi logic:

```
if (true) {
    var testVar = "Dùng var trong if";
}
console.log(testVar); // ✓ "Dùng var trong if" (vẫn truy cập được)
```

⚠ **Luôn ưu tiên let và const để tránh lỗi!**

---

(4) Lexical Scope – Phạm vi từ vựng (Scope lồng nhau)

📌 Hàm con có thể truy cập biến của hàm cha, nhưng ngược lại thì không.
📌 JavaScript sử dụng cơ chế "Scope Chain" để tìm biến từ trong ra ngoài.

```
function outerFunction() {
    let outerVar = "Biến trong hàm ngoài";
```

```
    function innerFunction() {
        console.log(outerVar); // ✅ Có thể truy cập
    }

    innerFunction();
}
outerFunction();
```

🔍 **Lưu ý**:

- innerFunction() có thể truy cập outerVar, nhưng outerFunction() không thể truy cập biến của innerFunction().

⚠ **Scope Chain** (Chuỗi phạm vi) hoạt động như sau:

1. **Tìm biến trong phạm vi hiện tại.**
2. **Nếu không tìm thấy, kiểm tra phạm vi cha.**
3. **Tiếp tục cho đến phạm vi toàn cục.**
4. **Nếu không có biến nào khớp, báo lỗi ReferenceError.**

---

(5) Module Scope – Phạm vi trong module

📌 **Trong JavaScript ES6, mỗi file .js là một module riêng biệt.**
📌 **Các biến khai báo bằng let, const, hoặc var trong module sẽ không bị rò rỉ ra phạm vi toàn cục.**

```
// file1.js
export let message = "Xin chào";

// file2.js
import { message } from './file1.js';
console.log(message); // ✅ "Xin chào"
```

🔍 **Lưu ý**:

- Biến trong một module **không thể truy cập từ file khác** trừ khi được export.
- **Dùng để tránh xung đột biến giữa các file.**

---

3. Tóm tắt so sánh các loại Scope

Loại Scope	Mô tả	Dùng với
Global Scope	Biến có thể truy cập ở mọi nơi	var, let, const
Function Scope	Biến chỉ tồn tại trong một hàm	var
Block Scope	Biến chỉ tồn tại trong {}	let, const
Lexical Scope	Hàm con có thể truy cập biến của hàm cha	Tất cả
Module Scope	Biến chỉ tồn tại trong module hiện tại	let, const

---

4. Nên dùng loại scope nào?

✅ Dùng **const** hoặc **let** thay vì **var** để tránh lỗi phạm vi.
✅ Dùng **block scope (**let**,** const**)** để biến không bị rò rỉ ra ngoài.
✅ Dùng **module scope** để tránh xung đột biến giữa các file.
✅ Luôn hiểu rõ **lexical scope** để tránh lỗi khi làm việc với function trong function.

🚀 **Nắm vững scope giúp code sạch hơn, ít lỗi hơn, và dễ bảo trì hơn!**