Chapter 11: Tích Hợp TypeScript với JavaScript Hiện Có và Declaration Files

Mô tả tổng quát

Chapter này tập trung vào cách tích hợp TypeScript vào các dự án JavaScript hiện có một cách hiệu quả, đồng thời giải thích cách sử dụng các thư viện JavaScript trong TypeScript thông qua **Declaration Files** (`.d.ts`). Đây là các file đặc biệt giúp TypeScript hiểu được kiểu dữ liệu của code JavaScript, từ đó cung cấp khả năng kiểm tra kiểu (type checking) và hỗ trợ lập trình viên khi làm việc với các thư viện bên ngoài hoặc code JavaScript cũ. Nội dung bao gồm các chiến lược tích hợp, cách sử dụng file `.d.ts` , và cách tạo chúng.

Tiêu đề Chapter

Làm Việc Chung: TypeScript với JavaScript và Declaration Files

---

Tóm tắt lý thuyết chính

1. Chiến lược tích hợp TypeScript vào dự án JavaScript

Việc chuyển đổi hoàn toàn một dự án JavaScript lớn sang TypeScript có thể tốn nhiều thời gian và công sức. Vì vậy, cách tiếp cận **từ từ (gradual adoption)** thường là lựa chọn tối ưu, đặc biệt với các dự án đã có codebase lớn.

Giải thích chi tiết:

- **Tại sao cần tích hợp từ từ?** TypeScript yêu cầu định nghĩa kiểu (type) cho các biến, hàm, và đối tượng, điều này có thể phức tạp nếu dự án JavaScript hiện có không có thông tin kiểu. Thay vì chuyển đổi toàn bộ dự án cùng lúc, bạn có thể bắt đầu bằng cách thêm TypeScript vào các phần mới hoặc từng phần nhỏ, giúp giảm rủi ro lỗi và dễ dàng kiểm soát quá trình chuyển đổi.
- **Lợi ích**: Giảm thiểu gián đoạn, tận dụng được các lợi ích của TypeScript (như kiểm tra kiểu) mà không cần chỉnh sửa toàn bộ code JavaScript cũ.

Cách thực hiện:

- **Bắt đầu với module mới**: Viết các file mới bằng TypeScript (`.ts` hoặc `.tsx`) và để các file JavaScript (`.js`) cũ hoạt động song song.
- **Chuyển đổi dần**: Từng bước chuyển các file `.js` sang `.ts` bằng cách thêm kiểu dữ liệu, bắt đầu từ các module nhỏ hoặc ít phụ thuộc.
- **Cấu hình** `tsconfig.json` : Sử dụng các tùy chọn trong `tsconfig.json` để hỗ trợ tích hợp.

Cấu hình `tsconfig.json` **hỗ trợ JavaScript:**

- `allowJs: true` : Cho phép TypeScript biên dịch cả file JavaScript (`.js` và `.jsx`). Điều này giúp bạn giữ nguyên các file `.js` trong dự án mà không cần chuyển đổi ngay.
  - **Ví dụ**: Nếu bạn có file `utils.js` , TypeScript sẽ nhận diện và biên dịch nó cùng với file `.ts` .
- `checkJs: true` : Bật kiểm tra kiểu cho file JavaScript. Bạn có thể dùng **JSDoc** trong file `.js` để cung cấp thông tin kiểu, giúp TypeScript hiểu code JavaScript.
  - **Ví dụ JSDoc**:

```
// file: legacyUtils.js
/**
 * Adds two numbers.
 * @param {number} a The first number.
 * @param {number} b The second number.
 * @returns {number} The sum of a and b.
 */
function add(a, b) {
  return a + b;
}
module.exports = { add };
```

  - **Giải thích**: JSDoc là cách chú thích trong JavaScript để mô tả kiểu dữ liệu. TypeScript sẽ đọc các chú thích này để kiểm tra kiểu mà không cần chuyển file sang `.ts` .
- `noEmit: true` : Chỉ kiểm tra kiểu mà không tạo file output JavaScript. Thường dùng khi bạn sử dụng công cụ khác như **Babel** để biên dịch code.

    - **Khi nào dùng?** Nếu dự án đã có quy trình build riêng (như Webpack hoặc Babel), bạn có thể dùng TypeScript chỉ để kiểm tra kiểu.
- `outDir` và `rootDir`: Quản lý cấu trúc thư mục khi biên dịch. Ví dụ, đặt `outDir: "./dist"` để lưu file output trong thư mục `dist`, và `rootDir: "./src"` để chỉ định thư mục nguồn.

Ví dụ cấu hình `tsconfig.json`:

```
{
  "compilerOptions": {
    "allowJs": true,
 "checkJs": true,
 "outDir": "./dist",
 "rootDir": "./src",
 "noEmit": false,
 "target": "es2016",
 "module": "commonjs",
 "strict": true
  },
  "include": ["src"]
}
```

---

2. Sử dụng thư viện JavaScript trong TypeScript

Khi sử dụng các thư viện JavaScript như **jQuery**, **Lodash**, hoặc **Express** trong TypeScript, bạn cần cung cấp thông tin kiểu để TypeScript hiểu cách hoạt động của thư viện. Nếu không, TypeScript sẽ coi các thành phần của thư viện là kiểu `any`, làm mất đi lợi ích kiểm tra kiểu.

Vấn đề:

- TypeScript không thể tự động suy ra kiểu từ code JavaScript.
- Nếu thiếu thông tin kiểu, TypeScript sẽ báo lỗi hoặc mặc định dùng kiểu `any`, dẫn đến việc mất kiểm soát kiểu.

Giải pháp: Declaration Files (`.d.ts`):

- **File `.d.ts` là gì?** Đây là file chứa các khai báo kiểu (type declarations) cho code JavaScript, không chứa logic thực thi, chỉ mô tả "hình dạng" (shape) của thư viện hoặc code JavaScript.
- **Ví dụ**: File `lodash.d.ts` mô tả các hàm, tham số, và giá trị trả về của thư viện **Lodash**.

Tìm và cài đặt Declaration Files từ DefinitelyTyped:

- **DefinitelyTyped** (https://definitelytyped.org/) là kho lưu trữ cộng đồng chứa các file `.d.ts` chất lượng cao cho hàng ngàn thư viện JavaScript phổ biến.
- Các file này được xuất bản dưới dạng package `@types` trên npm.
- **Lệnh cài đặt**:

```
npm install --save-dev @types/package-name
# Ví dụ cho Lodash:
npm install --save-dev @types/lodash
```

- **Cách hoạt động**: Sau khi cài đặt, TypeScript tự động nhận diện các file `.d.ts` trong thư mục `node_modules/@types` .

Ví dụ sử dụng Lodash:

```
// file: example.ts
import _ from 'lodash';

const numbers = [1, 2, 3, 4];
const shuffled = _.shuffle(numbers); // TypeScript biết kiểu của shuffle (hà
console.log(shuffled); // Ví dụ output: [3, 1, 4, 2]
```

- **Giải thích**: Nhờ `@types/lodash` , TypeScript biết rằng `_.shuffle` nhận một mảng và trả về một mảng cùng kiểu, giúp tránh lỗi khi sử dụng.

3. Viết Declaration Files cơ bản

Nếu một thư viện JavaScript không có file `.d.ts` trên DefinitelyTyped hoặc bạn làm việc với code JavaScript nội bộ, bạn cần tự viết file `.d.ts`.

Khai báo biến, hàm, class (Ambient Declarations):

- Sử dụng từ khóa `declare` để thông báo rằng các thành phần này tồn tại trong code JavaScript và cung cấp kiểu cho chúng.
- **Ví dụ**:

```
// file: my-legacy-library.d.ts
declare const GLOBAL_API_KEY: string;

declare function processData(data: any): { success: boolean; result?: an

declare class LegacyWidget {
  constructor(options: any);
  render(elementId: string): void;
  destroy(): void;
}
```

- **Giải thích**: File `.d.ts` trên mô tả một biến `GLOBAL_API_KEY` kiểu `string`, một hàm `processData` nhận tham số `any` và trả về một object, và một class `LegacyWidget` với các phương thức. TypeScript sẽ sử dụng thông tin này để kiểm tra kiểu mà không cần import (nếu là global).

Khai báo Module:

- Nếu thư viện JavaScript là module (CommonJS, ES Module, AMD), bạn cần khai báo module trong file `.d.ts`.
- **Ví dụ**:

```ts
// file: custom-module-lib.d.ts
declare module 'custom-module-lib' {
  export function greet(name: string): string;
  export const MAX_USERS: number;
}
```

- **Sử dụng trong file `.ts`:**

```ts
// file: app.ts
import { greet, MAX_USERS } from 'custom-module-lib';

console.log(greet("Developer")); // TypeScript biết greet trả về string
console.log(MAX_USERS); // TypeScript biết MAX_USERS là number
```

- **Giải thích**: File `.d.ts` định nghĩa "hình dạng" của module `custom-module-lib`, giúp TypeScript kiểm tra kiểu khi import.

Shorthand Ambient Module Declarations:

- Nếu bạn chỉ muốn khai báo module mà không cung cấp chi tiết kiểu (tạm thời dùng `any`):

```ts
// file: untyped-module.d.ts
declare module 'untyped-module';
```

- **Khi nào dùng?** Khi chưa có thời gian viết file `.d.ts` đầy đủ hoặc muốn nhanh chóng bỏ qua lỗi import. Tuy nhiên, nên hạn chế vì mất lợi ích kiểm tra kiểu.

---

4. Tạo `.d.ts` từ file `.ts` (Declaration Generation)

Khi viết thư viện bằng TypeScript, bạn nên cung cấp file `.d.ts` để người dùng (dù dùng TypeScript hay JavaScript) có thể tận dụng thông tin kiểu.

Cách thực hiện:

- Bật tùy chọn `declaration: true` trong `tsconfig.json` để TypeScript tự động tạo file `.d.ts` khi biên dịch.
- **Cấu hình mẫu:**

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "declaration": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  },
  "include": ["src"]
}
```

- **Ví dụ file nguồn:**

```
// file: src/math-lib.ts
export function sum(a: number, b: number): number {
  return a + b;
}
export const VERSION = "1.0.0";
```

- **Kết quả sau biên dịch:**
  - `dist/math-lib.js` : File JavaScript đã biên dịch.
  - `dist/math-lib.d.ts` :

```
export declare function sum(a: number, b: number): number;
export declare const VERSION: string;
```

- **Giải thích**: File `.d.ts` chứa các khai báo kiểu, giúp người dùng thư viện biết cách sử dụng hàm `sum` và hằng `VERSION` mà không cần xem code nguồn.

Code ví dụ chính (Tổng hợp và mở rộng)

1. Sử dụng thư viện có `@types` (ví dụ: `date-fns`)

Cài đặt:

```
npm install date-fns
npm install --save-dev @types/date-fns
```

Code TypeScript:

```
// file: dateExample.ts
import { format, addDays } from 'date-fns';

const today = new Date();
const threeDaysLater = addDays(today, 3);

console.log("Hôm nay (mặc định):", format(today, "PPP"));
console.log("3 ngày sau (mặc định):", format(threeDaysLater, "dd/MM/yyyy HH:
```

- **Giải thích**: Nhờ `@types/date-fns`, TypeScript biết kiểu của `format` và `addDays`, giúp đảm bảo bạn truyền đúng định dạng ngày tháng và tham số.

2. Viết file `.d.ts` cho module JavaScript

File JavaScript:

```
// file: legacy-lib/string-utils.js
function reverseString(str) {
  return str.split('').reverse().join('');
}
const GREETING_PREFIX = "Hello, ";
module.exports = { reverseString, GREETING_PREFIX };
```

File Declaration:

```
// file: src/types/string-utils.d.ts
declare module 'legacy-lib/string-utils' {
  export function reverseString(str: string): string;
  export const GREETING_PREFIX: string;
}
```

Sử dụng trong TypeScript:

```
// file: appUsingLegacy.ts
import { reverseString, GREETING_PREFIX } from '../legacy-lib/string-utils';

const original = "TypeScript";
const reversed = reverseString(original);
console.log(`${GREETING_PREFIX}${reversed}`); // Output: "Hello, tpircSepyT"
```

- **Giải thích**: File `.d.ts` cung cấp kiểu cho module JavaScript, giúp TypeScript kiểm tra đúng kiểu của `reverseString` (nhận và trả về `string`) và `GREETING_PREFIX` (kiểu `string`).

3. Tích hợp file JavaScript với `allowJs` và JSDoc

Cấu hình `tsconfig.json`:

```
{
  "compilerOptions": {
    "allowJs": true,
    "checkJs": true,
    "outDir": "./dist",
    "module": "commonjs",
    "target": "es2016",
    "strict": true
  },
  "include": ["src"]
}
```

File JavaScript:

```
// file: src/js-calculator.js
/**
 * @param {number} x
 * @param {number} y
 * @returns {number}
 */
function multiply(x, y) {
  return x * y;
}
module.exports = { multiply };
```

File TypeScript:

```
// file: src/ts-consumer.ts
import { multiply } from './js-calculator';

const product = multiply(5, 7); // TypeScript biết product là number
console.log("Product from JS:", product); // Output: "Product from JS: 35"

// const errorProduct = multiply(5, "7"); // Lỗi: Argument of type 'string'
```

- **Giải thích**: JSDoc trong `js-calculator.js` cung cấp thông tin kiểu, giúp TypeScript kiểm tra tham số và giá trị trả về của `multiply`.

Danh sách bài tập

1. Trắc nghiệm: File `.d.ts` dùng để làm gì?

Tiêu đề: Mục đích của Declaration Files

Mô tả: Chọn đáp án đúng nhất mô tả công dụng của file `.d.ts`.

Câu hỏi: Trong TypeScript, file `.d.ts` (Declaration File) chủ yếu được sử dụng để làm gì?

- A. Chứa code JavaScript đã được biên dịch từ TypeScript.
- B. Cung cấp thông tin kiểu (type definitions) cho code JavaScript hiện có, giúp TypeScript hiểu và làm việc với code đó.
- C. Cấu hình các tùy chọn của trình biên dịch TypeScript cho một dự án.
- D. Chứa các unit test cho code TypeScript.

Đáp án: B

Giải thích: File `.d.ts` chỉ chứa khai báo kiểu, không chứa logic thực thi. Chúng giúp TypeScript hiểu code JavaScript, từ đó cung cấp kiểm tra kiểu và gợi ý code trong editor.

---

2. Code: Viết file `.d.ts` cho hàm JavaScript đơn giản

Tiêu đề: Thực hành viết Declaration File

Mô tả:
Giả sử bạn có file JavaScript `legacy-math.js`:

```
// file: legacy-math.js
function calculateArea(shape, dimensions) {
  if (shape === "circle" && typeof dimensions === 'number') {
    return Math.PI * dimensions * dimensions;
  }
  if (shape === "rectangle" && typeof dimensions === 'object' && dimensions.
    return dimensions.width * dimensions.height;
  }
  return -1; // Error or unknown shape
}
module.exports = { calculateArea };
```

Viết file `legacy-math.d.ts` để cung cấp kiểu cho hàm `calculateArea`. Hàm nhận:

- `shape: "circle"`, `dimensions: number` (bán kính).
- `shape: "rectangle"`, `dimensions: { width: number, height: number }`.
- Hàm trả về `number`.

Giải pháp mẫu:

```
// file: src/types/legacy-math.d.ts
declare module 'legacy-math' {
  export function calculateArea(shape: "circle", radius: number): number;
  export function calculateArea(shape: "rectangle", dimensions: { width: num
}
```

- **Giải thích**: Sử dụng **function overloads** để mô tả hai cách gọi khác nhau của `calculateArea`. TypeScript sẽ kiểm tra đúng kiểu dựa trên tham số `shape`.

---

3. Code: Tìm và cài đặt declaration file cho thư viện `axios`

Tiêu đề: Sử dụng DefinitelyTyped

Mô tả:

1. Cài đặt thư viện `axios` và `@types/axios`:

```
npm install axios
npm install --save-dev @types/axios
```

2. Viết code TypeScript sử dụng `axios.get()` để gọi API công khai `https://jsonplaceholder.typicode.com/todos/1` và log dữ liệu trả về.

Giải pháp mẫu:

```
// file: apiCaller.ts
import axios, { AxiosResponse } from 'axios';

interface Todo {
  userId: number;
  id: number;
  title: string;
  completed: boolean;
}

async function fetchTodo(): Promise<void> {
  try {
    const response: AxiosResponse<Todo> = await axios.get('https://jsonplace
    const todoData: Todo = response.data;

    console.log("Todo Fetched:");
    console.log(`ID: ${todoData.id}`);
    console.log(`Title: ${todoData.title}`);
    console.log(`Completed: ${todoData.completed}`);
  } catch (error: any) {
    if (axios.isAxiosError(error)) {
      console.error("Axios error:", error.message);
      if (error.response) {
        console.error("Status:", error.response.status);
        console.error("Data:", error.response.data);
      }
```

```
    } else {
      console.error("An unexpected error occurred:", error);
    }
  }
}

fetchTodo();
```

- **Giải thích**: Nhờ `@types/axios`, TypeScript biết kiểu của `response` và `data`. Interface `Todo` định nghĩa cấu trúc dữ liệu trả về, giúp code an toàn và dễ bảo trì.

---

4. Code: Tạo `.d.ts` từ file `.ts`

Tiêu đề: Thực hành Declaration Generation

Mô tả:

1. Tạo file `src/geometry.ts`:

```
// file: src/geometry.ts
export interface ShapeInfo {
  name: string;
  sides: number;
  area?: number;
}

export function getShapeDetails(info: ShapeInfo): string {
  let details = `Shape: ${info.name}, Sides: ${info.sides}`;
  if (info.area) {
    details += `, Area: ${info.area}`;
  }
  return details;
}
```

2. Cấu hình `tsconfig.json` với `declaration: true`.
3. Chạy `tsc` và kiểm tra file `.d.ts` trong thư mục `outDir`.

Giải pháp mẫu (`tsconfig.json`):

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "declaration": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  },
  "include": ["src"]
}
```

File `.d.ts` được tạo:

```
// file: dist/geometry.d.ts
export interface ShapeInfo {
  name: string;
  sides: number;
  area?: number;
}
export declare function getShapeDetails(info: ShapeInfo): string;
```

- **Giải thích**: File `.d.ts` chứa khai báo kiểu của interface và hàm, giúp người dùng thư viện hiểu cách sử dụng mà không cần xem code nguồn.

---

5. Thảo luận: Khi nào nên tự viết file `.d.ts` và khi nào nên dựa vào `@types`?

Tiêu đề: Chiến lược sử dụng Declaration Files

Mô tả: Thảo luận khi nào nên dùng package `@types` từ DefinitelyTyped và khi nào cần tự viết file `.d.ts`.

Gợi ý:

- **Dùng `@types`**:
  - Thư viện phổ biến (như Lodash, Axios, Express) có sẵn trên DefinitelyTyped.
  - Được cộng đồng kiểm thử, đảm bảo chất lượng.
  - Tiết kiệm thời gian, không cần viết lại từ đầu.
- **Tự viết `.d.ts`**:
  - Thư viện nội bộ hoặc code JavaScript tự viết.
  - Thư viện nhỏ, ít phổ biến, không có `@types`.
  - Muốn kiểm soát hoàn toàn định nghĩa kiểu.
  - Đóng góp cho DefinitelyTyped (nếu bạn viết file `.d.ts` chất lượng, có thể chia sẻ lên DefinitelyTyped).

Giải thích:

- **Lợi ích của `@types`**: Tiết kiệm thời gian, đáng tin cậy, được cập nhật thường xuyên bởi cộng đồng.
- **Lợi ích của tự viết**: Phù hợp với nhu cầu cụ thể, đặc biệt khi làm việc với code nội bộ hoặc thư viện không có sẵn file `.d.ts`.
- **Mẹo**: Nếu tự viết, bắt đầu với khai báo đơn giản (dùng `any` nếu cần) và dần cải thiện chi tiết kiểu khi có thời gian.