Chapter 5: Kết Hợp Kiểu - Union Types, Intersection Types và Type Guards

Mô tả tổng quát

Chương này giúp bạn nắm vững các công cụ mạnh mẽ trong TypeScript để làm việc với các kiểu dữ liệu phức tạp một cách an toàn và hiệu quả. Bạn sẽ học cách sử dụng:

- **Union Types (`|`)**: Cho phép một biến thuộc một trong nhiều kiểu dữ liệu khác nhau.
- **Intersection Types (`&`)**: Kết hợp nhiều kiểu thành một kiểu mới chứa tất cả thuộc tính của các kiểu gốc.
- **Type Guards**: Các kỹ thuật như `typeof`, `instanceof`, `in`, và user-defined type guards để kiểm tra và thu hẹp kiểu dữ liệu tại runtime.

Những khái niệm này rất quan trọng khi xây dựng ứng dụng thực tế, đặc biệt trong các dự án ReactJS hoặc NextJS sử dụng TypeScript, nơi bạn cần xử lý dữ liệu từ API hoặc các nguồn không xác định trước.

Tiêu đề chương học

Kết Hợp Kiểu: Union, Intersection và Type Guards

Tóm tắt lý thuyết chính

1. Union Types (`|`)

Union Types cho phép một biến, tham số hoặc giá trị trả về thuộc một trong nhiều kiểu dữ liệu được chỉ định. Toán tử `|` được sử dụng để định nghĩa union type.

Ví dụ đơn giản:

```typescript
let identifier: string | number;

identifier = "user-123"; // OK
console.log(identifier.toUpperCase()); // OK, TypeScript biết identifier là

identifier = 404; // OK
// console.log(identifier.toUpperCase()); // Lỗi: 'toUpperCase' không tồn tạ
```

Giải thích:

- Khi làm việc với union types, TypeScript chỉ cho phép truy cập các thuộc tính/phương thức **chung** của tất cả các kiểu trong union.
- Để sử dụng các thuộc tính/phương thức riêng của một kiểu cụ thể, bạn cần **thu hẹp kiểu** (narrowing) bằng type guards (sẽ giải thích bên dưới).
- Union types rất hữu ích khi xử lý dữ liệu từ API, nơi một giá trị có thể là `string`, `number`, hoặc thậm chí `null`.

2. Làm việc với Union Types: Thu hẹp kiểu (Narrowing)

Thu hẹp kiểu là quá trình TypeScript xác định một kiểu cụ thể hơn trong một khối mã dựa trên các kiểm tra điều kiện. Dưới đây là các kỹ thuật phổ biến:

a. Sử dụng `typeof`

Hữu ích khi kiểm tra các kiểu nguyên thủy (`string`, `number`, `boolean`, `undefined`, v.v.).

Ví dụ:

```typescript
function printId(id: string | number): void {
    if (typeof id === "string") {
        // TypeScript biết id là string
        console.log("ID (string):", id.toUpperCase());
    } else {
        // TypeScript biết id là number
```

```
        console.log("ID (number):", id.toFixed(0));
    }
}

printId("abc-123"); // ID (string): ABC-123
printId(101.56);   // ID (number): 102
```

Giải thích:

- `typeof` trả về kiểu của biến tại runtime (`"string"` , `"number"` , v.v.).
- Trong khối `if` , TypeScript tự động thu hẹp kiểu của `id` dựa trên kết quả kiểm tra.
- Ứng dụng thực tế: Xử lý các tham số linh hoạt trong hàm hoặc dữ liệu từ form nhập liệu.

b. Sử dụng `instanceof`

Hữu ích khi làm việc với các đối tượng được tạo từ class.

Ví dụ:

```
class Dog {
    bark() { console.log("Woof!"); }
}
class Cat {
    meow() { console.log("Meow!"); }
}

function makeSound(animal: Dog | Cat): void {
    if (animal instanceof Dog) {
        animal.bark(); // animal là Dog
    } else {
        animal.meow(); // animal là Cat
    }
}

makeSound(new Dog()); // Woof!
makeSound(new Cat()); // Meow!
```

Giải thích:

- `instanceof` kiểm tra xem một đối tượng có phải là instance của một class hay không.
- TypeScript sử dụng thông tin này để thu hẹp kiểu trong khối `if` hoặc `else`.
- Ứng dụng thực tế: Xử lý các đối tượng phức tạp trong các ứng dụng có nhiều loại thực thể (ví dụ: trong game hoặc hệ thống quản lý).

c. Sử dụng toán tử `in`

Kiểm tra sự tồn tại của một thuộc tính trong đối tượng.

Ví dụ:

```
interface Admin {
    isAdmin: true;
    manageUsers(): void;
}
interface RegularUser {
    isUser: true;
    viewContent(): void;
}

function performAction(user: Admin | RegularUser): void {
    if ("manageUsers" in user) {
        user.manageUsers(); // user là Admin
    } else {
        user.viewContent(); // user là RegularUser
    }
}
```

Giải thích:

- Toán tử `in` kiểm tra xem một thuộc tính có tồn tại trong đối tượng không.
- TypeScript sử dụng kết quả này để thu hẹp kiểu.
- Ứng dụng thực tế: Xử lý các đối tượng người dùng với vai trò khác nhau (admin, user) trong hệ thống xác thực.

3. Discriminated Unions (Tagged Unions)

Discriminated Unions là một pattern mạnh mẽ để làm việc với union types. Mỗi kiểu trong union có một thuộc tính chung (gọi là **discriminant** hoặc **tag**) với giá trị là một literal type cụ thể (thường là chuỗi hoặc số).

Ví dụ:

```
interface Circle {
    kind: "circle"; // Discriminant
    radius: number;
}
interface Square {
    kind: "square"; // Discriminant
    sideLength: number;
}
interface Rectangle {
    kind: "rectangle"; // Discriminant
    width: number;
    height: number;
}

type Shape = Circle | Square | Rectangle;

function getArea(shape: Shape): number {
    switch (shape.kind) {
        case "circle":
            return Math.PI * shape.radius ** 2;
        case "square":
            return shape.sideLength ** 2;
        case "rectangle":
            return shape.width * shape.height;
        default:
            const _exhaustiveCheck: never = shape;
            return _exhaustiveCheck;
    }
}

console.log(getArea({ kind: "circle", radius: 5 })); // ~78.54
```

```
console.log(getArea({ kind: "square", sideLength: 4 })); // 16
console.log(getArea({ kind: "rectangle", width: 3, height: 6 })); // 18
```

Giải thích:

- Thuộc tính `kind` là discriminant, giúp TypeScript phân biệt các kiểu trong union.
- `switch` statement sử dụng `kind` để thu hẹp kiểu, cho phép truy cập các thuộc tính cụ thể (`radius` , `sideLength` , v.v.).
- Biến `_exhaustiveCheck` kiểu `never` đảm bảo rằng tất cả các trường hợp đã được xử lý. Nếu thêm một kiểu mới vào `Shape` mà không cập nhật `switch` , TypeScript sẽ báo lỗi.
- Ứng dụng thực tế: Xử lý phản hồi API với các trạng thái khác nhau (success, error, loading).

4. Intersection Types (`&`)

Intersection Types kết hợp nhiều kiểu thành một kiểu mới chứa tất cả thuộc tính của các kiểu gốc. Toán tử `&` được sử dụng để định nghĩa intersection type.

Ví dụ:

```
interface Draggable {
    drag(): void;
}
interface Resizable {
    resize(): void;
}

type UIWidget = Draggable & Resizable;

let widget: UIWidget = {
    drag: () => console.log("Dragging..."),
    resize: () => console.log("Resizing...")
};

widget.drag(); // Dragging...
widget.resize(); // Resizing...
```

Giải thích:

- `UIWidget` yêu cầu đối tượng phải có cả phương thức `drag` và `resize`.
- Nếu các kiểu giao nhau có thuộc tính trùng tên nhưng khác kiểu, TypeScript có thể báo lỗi hoặc kết quả là kiểu `never`.
- Ứng dụng thực tế: Xây dựng các thành phần giao diện (UI components) trong React, nơi một component có thể có nhiều tính năng (draggable, resizable, clickable, v.v.).

5. Type Guards

Type Guards là các kỹ thuật giúp TypeScript thu hẹp kiểu dữ liệu trong một phạm vi mã nhất định. Ngoài `typeof`, `instanceof`, và `in`, bạn có thể tự định nghĩa type guards.

User-defined Type Guards

Sử dụng từ khóa `is` để định nghĩa một hàm kiểm tra kiểu trả về boolean, với cú pháp: `parameter is Type`.

Ví dụ:

```
interface Bird {
    fly(): void;
    layEggs(): void;
}
interface Fish {
    swim(): void;
    layEggs(): void;
}

function isBird(pet: Fish | Bird): pet is Bird {
    return (pet as Bird).fly !== undefined;
}

function moveAnimal(pet: Fish | Bird) {
    if (isBird(pet)) {
        pet.fly(); // pet là Bird
    } else {
        pet.swim(); // pet là Fish
```

```
    }
    pet.layEggs(); // Có thể gọi trực tiếp vì cả Bird và Fish đều có layEggs
}

let sparrow: Bird = { fly: () => console.log("Sparrow flying"), layEggs: ()
let salmon: Fish = { swim: () => console.log("Salmon swimming"), layEggs: ()

moveAnimal(sparrow); // Sparrow flying, Sparrow laying eggs
moveAnimal(salmon); // Salmon swimming, Salmon laying eggs
```

Giải thích:

- Hàm `isBird` trả về `true` nếu `pet` có phương thức `fly`, và TypeScript thu hẹp kiểu của `pet` thành `Bird`.
- Cú pháp `pet is Bird` là **type predicate**, giúp TypeScript hiểu kết quả của hàm.
- Ứng dụng thực tế: Kiểm tra kiểu dữ liệu phức tạp từ dữ liệu API hoặc các nguồn không rõ ràng.

Code ví dụ tổng hợp

Dưới đây là các ví dụ minh họa tất cả các khái niệm trên, được mở rộng để dễ hiểu hơn:

```
// ---- Union Types và typeof ----
function processValue(value: string | number | boolean): void {
    if (typeof value === "string") {
        console.log(`String value: ${value.toUpperCase()}`);
    } else if (typeof value === "number") {
        console.log(`Number value: ${value.toFixed(2)}`);
    } else {
        console.log(`Boolean value: ${value}`);
    }
}

processValue("Hello TypeScript"); // String value: HELLO TYPESCRIPT
processValue(3.14159); // Number value: 3.14
processValue(true); // Boolean value: true
```

```typescript
// ---- Discriminated Unions ----
type NetworkResponse =
    | { status: "success"; data: { userId: string; items: string[] } }
    | { status: "error"; errorCode: number; errorMessage: string };

function handleNetworkResponse(response: NetworkResponse): void {
    if (response.status === "success") {
        console.log("Data received:", response.data.items.join(", "));
    } else {
        console.error(`Error ${response.errorCode}: ${response.errorMessage}
    }
}

handleNetworkResponse({ status: "success", data: { userId: "u1", items: ["it
// Data received: item1, item2
handleNetworkResponse({ status: "error", errorCode: 500, errorMessage: "Inte
// Error 500: Internal Server Error

// ---- Intersection Types ----
type Person = { name: string; age: number };
type Employee = { employeeId: string; department: string };

type EmployeeProfile = Person & Employee;

let employee1: EmployeeProfile = {
    name: "Alice Wonderland",
    age: 30,
    employeeId: "E123",
    department: "Engineering"
};

console.log(`${employee1.name} (ID: ${employee1.employeeId}) works in ${empl
// Alice Wonderland (ID: E123) works in Engineering.

// ---- User-defined Type Guard với 'in' ----
interface Car {
    drive(): void;
    fuelType: "gasoline" | "diesel" | "electric";
}
interface Bicycle {
    pedal(): void;
    gearCount: number;
```

```
}

function isCar(vehicle: Car | Bicycle): vehicle is Car {
    return "drive" in vehicle;
}

function operateVehicle(vehicle: Car | Bicycle): void {
    if (isCar(vehicle)) {
        vehicle.drive();
        console.log(`This car uses ${vehicle.fuelType}.`);
    } else {
        vehicle.pedal();
        console.log(`This bicycle has ${vehicle.gearCount} gears.`);
    }
}

let myCar: Car = { drive: () => console.log("Car driving..."), fuelType: "el
let myBike: Bicycle = { pedal: () => console.log("Bicycle pedaling..."), gea

operateVehicle(myCar); // Car driving..., This car uses electric.
operateVehicle(myBike); // Bicycle pedaling..., This bicycle has 18 gears.
```

Danh sách bài tập

Bài tập 1: Trắc nghiệm - Type Guard với `in`

Tiêu đề: Type Guard với `in`

Mô tả: Chọn toán tử phù hợp để kiểm tra sự tồn tại của thuộc tính trong một đối tượng, thường dùng trong type guards.

Câu hỏi: Để kiểm tra xem một đối tượng `obj` có thuộc tính `propName` hay không trong TypeScript, bạn sẽ sử dụng toán tử nào?

- A. `typeof obj.propName === "undefined"`
- B. `obj instanceof propName`
- C. `"propName" in obj`

- D. `obj.hasOwnProperty(propName)`

Đáp án: C

Giải thích:

- Toán tử `in` kiểm tra sự tồn tại của thuộc tính trong đối tượng (bao gồm cả thuộc tính kế thừa).
- `hasOwnProperty` chỉ kiểm tra thuộc tính trực tiếp, không được ưu tiên trong type guards.
- `typeof` dùng để kiểm tra kiểu, không phù hợp ở đây.
- `instanceof` dùng để kiểm tra instance của class.

Bài tập 2: Code - Hàm xử lý `string` hoặc `string[]`

Tiêu đề: Thực hành với Union Type và `typeof` / `Array.isArray`

Mô tả: Viết hàm `logItems` nhận tham số `items` kiểu `string | string[]`:

- Nếu `items` là `string`, in ra chuỗi đó.
- Nếu `items` là `string[]`, in từng phần tử trên một dòng.

Giải pháp mẫu:

```
function logItems(items: string | string[]): void {
    if (typeof items === "string") {
        console.log("Item:", items);
    } else if (Array.isArray(items)) {
        console.log("Items in array:");
        items.forEach(item => console.log("- " + item));
    }
}

logItems("Một mục duy nhất"); // Item: Một mục duy nhất
logItems(["Táo", "Chuối", "Cam"]); // Items in array: - Táo, - Chuối, - Cam
```

Giải thích:

- `typeof items === "string"` kiểm tra `items` là chuỗi.
- `Array.isArray(items)` là type guard kiểm tra `items` là mảng.
- Hàm xử lý cả hai trường hợp một cách rõ ràng và an toàn.

Bài tập 3: Code - Intersection Type cho `FullStackDeveloper`

Tiêu đề: Thực hành với Intersection Type

Mô tả:

1. Tạo hai interface: `FrontendDeveloper` (có `frontendTech: string[]`, `buildUI()`), `BackendDeveloper` (có `backendTech: string[]`, `setupServer()`).
2. Tạo type alias `FullStackDeveloper` là intersection của hai interface trên.
3. Tạo đối tượng `devProfile` kiểu `FullStackDeveloper` và in thông tin.

Giải pháp mẫu:

```
interface FrontendDeveloper {
    frontendTech: string[];
    buildUI(): void;
}

interface BackendDeveloper {
    backendTech: string[];
    setupServer(): void;
}

type FullStackDeveloper = FrontendDeveloper & BackendDeveloper;

let devProfile: FullStackDeveloper = {
    frontendTech: ["React", "TailwindCSS", "TypeScript"],
    backendTech: ["Node.js", "Express", "PostgreSQL"],
    buildUI: () => console.log("Building user interface..."),
    setupServer: () => console.log("Setting up server environment...")
};
```

```
console.log("Full-stack Developer Profile:");
console.log("Frontend Technologies:", devProfile.frontendTech.join(", "));
console.log("Backend Technologies:", devProfile.backendTech.join(", "));
devProfile.buildUI(); // Building user interface...
devProfile.setupServer(); // Setting up server environment...
```

Giải thích:

- `FullStackDeveloper` yêu cầu đối tượng phải có tất cả thuộc tính và phương thức của cả `FrontendDeveloper` và `BackendDeveloper`.
- Ví dụ này mô phỏng hồ sơ của một lập trình viên full-stack, thường gặp trong các dự án thực tế.

Bài tập 4: Code - Discriminated Unions cho API Response

Tiêu đề: Thực hành với Discriminated Unions

Mô tả: Tạo các interface:

- `ApiSuccessResponse<T>`: `status: "success"`, `data: T`.
- `ApiErrorResponse`: `status: "error"`, `message: string`, `code: number`. Tạo type alias `ApiResponse<T>` là union của hai interface trên. Viết hàm `handleApiResponse<T>` để xử lý:
- Nếu `status` là `"success"`, in `response.data`.
- Nếu `status` là `"error"`, in `response.message` và `response.code`.

Giải pháp mẫu:

```
interface ApiSuccessResponse<T> {
    status: "success";
    data: T;
}

interface ApiErrorResponse {
```

```
    status: "error";
    message: string;
    code: number;
}

type ApiResponse<T> = ApiSuccessResponse<T> | ApiErrorResponse;

function handleApiResponse<T>(response: ApiResponse<T>): void {
    switch (response.status) {
        case "success":
            console.log("API Success! Data:", response.data);
            break;
        case "error":
            console.error(`API Error (Code: ${response.code}): ${response.me
            break;
    }
}

type User = { id: number; name: string };
let successfulUserData: ApiResponse<User> = {
    status: "success",
    data: { id: 1, name: "Alice" }
};

let failedAuthResponse: ApiResponse<User> = {
    status: "error",
    message: "Authentication Failed",
    code: 401
};

handleApiResponse(successfulUserData); // API Success! Data: { id: 1, name:
handleApiResponse(failedAuthResponse); // API Error (Code: 401): Authenticat
```

Giải thích:

- `status` là discriminant giúp phân biệt `ApiSuccessResponse` và `ApiErrorResponse`.
- Hàm `handleApiResponse` sử dụng `switch` để xử lý từng trường hợp.
- Pattern này rất phổ biến khi làm việc với API trong các ứng dụng web.

Bài tập 5: Code - User-defined Type Guard `isStringArray`

Tiêu đề: Thực hành với User-defined Type Guard

Mô tả: Viết hàm `isStringArray` kiểm tra một biến `unknown` có phải là `string[]` hay không. Hàm cần:

1. Kiểm tra `value` là mảng (`Array.isArray`).
2. Kiểm tra tất cả phần tử là `string` (`value.every`). Thử nghiệm với các giá trị khác nhau.

Giải pháp mẫu:

```
function isStringArray(value: unknown): value is string[] {
    return Array.isArray(value) && value.every(item => typeof item === "stri
}

function processStringArray(data: unknown): void {
    if (isStringArray(data)) {
        console.log("Processing string array:", data.map(s => s.toUpperCase(
    } else {
        console.log("Input is not a string array:", data);
    }
}

let testValue1: unknown = ["hello", "world", "typescript"];
let testValue2: unknown = ["hello", 123, "world"];
let testValue3: unknown = "this is a string";
let testValue4: unknown = [1, 2, 3];
let testValue5: unknown = null;

processStringArray(testValue1); // Processing string array: [ 'HELLO', 'WORL
processStringArray(testValue2); // Input is not a string array: [ 'hello', 1
processStringArray(testValue3); // Input is not a string array: this is a st
processStringArray(testValue4); // Input is not a string array: [ 1, 2, 3 ]
processStringArray(testValue5); // Input is not a string array: null
```

Giải thích:

- `isStringArray` sử dụng `Array.isArray` và `every` để kiểm tra kiểu.
- Type predicate `value is string[]` giúp TypeScript thu hẹp kiểu trong khối `if`.
- Ví dụ này mô phỏng việc kiểm tra dữ liệu từ nguồn không rõ (như API).

Kết luận

Chương này cung cấp các công cụ thiết yếu để làm việc với kiểu dữ liệu phức tạp trong TypeScript. **Union Types**, **Intersection Types**, và **Type Guards** là nền tảng để xây dựng các ứng dụng an toàn và dễ bảo trì. Hãy thực hành các bài tập trên để nắm vững các khái niệm này, đặc biệt trong các dự án ReactJS hoặc NextJS sử dụng TypeScript.