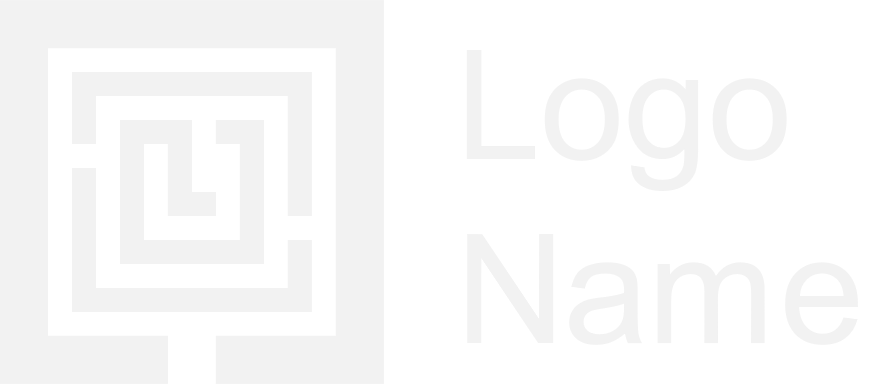
|  |
| --- |
| Typescript  doccument  Vue 3 |
| CLS-5 Street Address  City, ST ZIP Code  Phone  Email |



### **Tham khảo**

Tài liệu tham khảo:

1. <https://www.typescriptlang.org/docs/>
2. https://levunguyen.com/laptrinhjavascript/2021/03/04/khai-bao-bien-trong-typescript/

### **Lợi ích và khó khăn**

### **Cách sử dụng**

#### *Variable:*

*<scope>* <name>?: *<type> =* <value>

Ex: *const* props?: *object =* {

Prop1?: *‘hi’*

}

// khai báo biến trong vue 3 ts thông qua ref

*const* refForm = ref<props>({

Prop1?: *‘hi’*

})

Scope: có các kiểu const, var, let

Type có các loại :

* Nguyên hàm: string, number, Boolean
* Mảng: array
  + Ta có thể khai báo một mảng chứa kiểu nào đó thông qua cú pháp: Array<type> => Array<number>, Array<string>
* Any
* Function
* Object
* Interface
* Class
* Null và undefined
* Unknown: Loại unknownđại diện cho bất kỳ giá trị. Điều này tương tự với anyloại, nhưng an toàn hơn vì không hợp pháp để làm bất cứ điều gì với một unknowngiá trị

##### Static:

Các thành viên tĩnh của một lớp được truy cập bằng cách sử dụng tên lớp và ký hiệu dấu chấm mà không cần tạo một đối tượng, ví dụ: <ClassName>.<StaticMember>.

Được khai báo bằng từ khóa ‘static’

Biến static sẽ được truy suất thông qua class không phải thông qua biểu hiện của class là đối tượng

*class* Circle {

  static pi: *number* = 3.14;

  static calculateArea(*radius*:*number*) {

      return this.pi \* *radius* \* *radius*;

  }

}

Circle.pi; // returns 3.14

Circle.calculateArea(5); // returns 78.5

//error

*const* objCircle = new Circle()

objCircle.pi

##### Enum

Enum được cài đặt thông qua cấu trúc:

1. Enum số:

*enum* Direction {

  Up = <number>,

  Down,

  Left,

  Right,

}

Cách gọi: Direction.up

<number>: là giá trị của ‘Up’ và là chỉ số giá trị ban đầu mặc định và sẽ tác động đến các giá trị của các thành phần tiếp theo

enum Direction {

Up,

Down,

Left,

Right,

}

Thì mặc định giá trị khởi đầu là 0. Tất cả các thành viên sau được tự động tăng từ thời điểm đó. Nói cách khác, Direction.Up có giá trị 0, Down  1, Left 2 và Right  3.

enum Direction {

Up = 1,

Down,

Left,

Right,

}

Thì mặc định giá trị khởi đầu là 1. Tất cả các thành viên sau được tự động tăng từ thời điểm đó. Nói cách khác, Direction.Up có giá trị 1, Down  2, Left 3 và Right  4.

Các hành vi tăng tự động hữu ích với trường hợp chúng ta không quan tâm đến chính các giá trị thành viên, nhưng quan tâm đến mỗi giá trị khác biệt với các giá trị khác thành viên trong cùng một enum

\*\*Lưu ý: *Các enum không có bộ khởi tạo cần phải được đặt trước hoặc phải xuất hiện sau các enum số được khởi tạo với các hằng số hoặc các thành viên enum không đổi khác. Nói cách khác, những điều sau đây không được phép*:

enum E {

A = getSomeValue(),

B,

Enum member must have initializer.Enum member must have initializer.

}

Ở đây enum B chưa có bộ khởi tạo mà phía trước ngay cạnh nó lại không có emun hằng số mà là một enum A biến đổi thông qua function.

enum E {

X = 5,

A = getSomeValue(),

B,

Enum member must have initializer.Enum member must have initializer.

}

* B lúc này vẫn báo lỗi dù phía trước nó đã có x là một hằng số, nhưng do phía trước liên tiếp nó vẫn không phải là hằng số

enum E {

A = getSomeValue(),

X = 5,

B,

}

* B lúc này hợp lệ và nhận giá trị là 6

1. Enum chuỗi

enum Direction {

Up = "UP",

Down = "DOWN",

Left = "LEFT",

Right = "RIGHT",

}

Trong một chuỗi enum, mỗi thành viên phải được khởi tạo liên tục bằng một chuỗi ký tự hoặc với một thành viên chuỗi enum khác.

1. Enum không đồng nhất

enums có thể được trộn lẫn với các thành viên chuỗi và số

\*\*\*Khuyên: không nên dùng cách này

Enum trong TypeScript là một cấu trúc dữ liệu giúp cho việc định nghĩa và sử dụng các giá trị cố định trong mã. Enum được sử dụng rất nhiều trong các trường hợp sau:

1. Định nghĩa các trạng thái: Bạn có thể sử dụng enum để định nghĩa các trạng thái của một đối tượng hoặc một hệ thống, ví dụ như trạng thái của một task hoặc một hệ thống trạng thái.
2. Xác định các giá trị hạn chế: Bạn có thể sử dụng enum để xác định các giá trị hạn chế cho một thuộc tính hoặc một biến.
3. Định nghĩa các kiểu dữ liệu: Bạn có thể sử dụng enum để định nghĩa các kiểu dữ liệu riêng cho một đối tượng hoặc một hệ thống, ví dụ như kiểu dữ liệu của một đối tượng hoặc một trạng thái.

##### 3. As const:

Giúp bạn gán một giá trị cho một biến với kiểu dữ liệu được giữ nguyên và không thể thay đổi sau đó

*const* items = [1, 2, 3] as const;

Thử:

items.push(3)

Property 'push' does not exist on type 'readonly [1, 2, 3]'.

Test object

##### 4. Tuple:

* *Sử dụng Tuple để chứa 2 hoặc nhiều giá trị có kiểu dữ liệu khác nhau*

|  |  |
| --- | --- |
| 1  2  3  4  5 | var empId: number = 1;  var empName: string = "Steve";  // Tuple type variable  var employee: [number, string] = [1, "Steve"];  var employee: [number, string][];  employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]]; |

* Chúng ta truy cập mảng Tuple như sau:

|  |  |
| --- | --- |
| 1  2  3 | var employee: [number, string] = [1, "Steve"];  employee[0]; // returns 1  employee[1]; // returns "Steve" |

* Chúng ta sử dụng push để thêm phần tử cho Tuple.

|  |  |
| --- | --- |
| 1  2  3 | var employee: [number, string] = [1, "Steve"];  employee.push(2, "Bill");  console.log(employee); //Output: [1, 'Steve', 2, 'Bill'] |

##### 5. Union:

* Cho phép chúng ta sử dụng nhiều kiểu dữ liệu cho một biến

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let code: (string | number);  code = 123; // OK  code = "ABC"; // OK  code = false; // Compiler Error  let empId: string | number;  empId = 111; // OK  empId = "E111"; // OK  empId = true; // Compiler Error |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function displayType(code: (string | number))  {  if(typeof(code) === "number")  console.log('Code is number.')  else if(typeof(code) === "string")  console.log('Code is string.')  }  displayType(123); // Output: Code is number.  displayType("ABC"); // Output: Code is string.  displayType(true); //Compiler Error: Argument of type 'true' is not assignable to a parameter of type string | number |

##### 6. Any:

* Any sử dụng khi ta không biết kiểu dữ liệu là gì.
* Thường được sử dụng khi chúng ta gọi một webservice bên ngoài hoặc dịch vụ của nhà phát triển thứ 3.

|  |  |
| --- | --- |
| 1  2  3 | let something: any = "Hello World!";  something = 23;  something = true; |

**Chúng ta có thể tạo mảng với any như sau:**

|  |  |
| --- | --- |
| 1  2  3 | let arr: any[] = ["John", 212, true];  arr.push("Smith");  console.log(arr); //Output: [ 'John', 212, true, 'Smith' ] |

##### 7. Void:

* Void đại diện cho giá trị trả về của các hàm không trả về giá trị. Đó là kiểu suy luận bất cứ khi nào một hàm không có bất kỳ returncâu lệnh nào hoặc không trả về bất kỳ giá trị rõ ràng nào từ các câu lệnh trả về đó.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function sayHi(): void {  console.log('Hi!')  }  let speech: void = sayHi();  console.log(speech); //Output: undefined |

##### 8. Never:

*Never có nghĩa giá trị đó sẽ không xảy ra. Never được sử dụng khi ta chắc chắn việc gì đó không xảy ra. Ví dụ sau chúng ta viết function nó sẽ không bao giờ trả về lỗi vì while luôn true nên vòng lặp while chạy vô hạn không bao giờ bị lỗi.*

as *never*

*window.location = `http: ...` =>* Type 'string' is not assignable to type 'Location | (string & Location)'

window.location *= `http: ...`* as *never => ok*

window.location *= `http: ...`* as *any => ok*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function throwError(errorMsg: string): never {  throw new Error(errorMsg);  }  function keepProcessing(): never {  while (true) {  console.log('I always does something and never ends.')  }  } |

##### \*9. Type Aliases:

*Cơ bản nó là* một bí danh của một đối tượng, khi bạn muốn sử dụng ở nhiều nơi cho các loại cấu trúc phức tạp*, một*tên*cho bất kỳ*loại*nào*

<https://flow.org/en/docs/types/aliases/>

type MyObject<A, B, C, string, number > = {

foo: A,

bar: B,

baz: C,

property: string,  
method(): number,

};

type ID = number | string;

var val: MyObject<number, boolean, string> = {

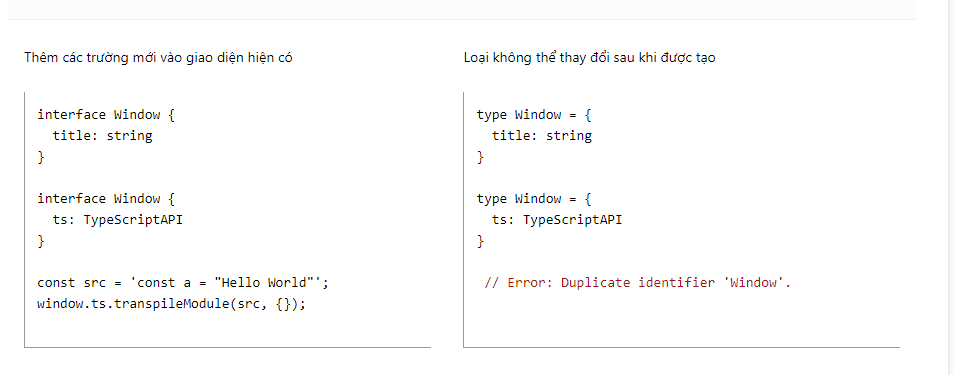
foo: 1,

bar: true,

baz: 'three',

};

export *type* EmailLabel = 'personal' | 'company' | 'important' | 'private'



Ta thấy type không thể khai báo đè còn interface thì cho phép

##### \*10. Declare:

Declare dùng để định nghĩa một biến, modul, class,.. đã được định nghĩa ở bên ngoài tệp Typescript hiện tại, chẳng hạn như trong một tệp khác được import vào

##### \*11. Type Assertion:loại xác nhận

Dùng xác nhận loại thông tin mà ts không thể biết được,để định nghĩa rõ hơn hoặc thiếu định nghĩa cho các biến bằng cách trích xuất ta có thể dùng type assertion để định nghĩa kiểu loại cho nó:

Loại *HTMLCanvasElement*

*const* myCanvas = document.getElementById("main\_canvas") as *HTMLCanvasElement*;

mặc định nó sẽ trả về kiểu dữ liệu HTMLElement nhưng ta muốn xác nhận kiểu dữ liệu cụ thể hơn là *HTMLCanvasElement*

hoặc bạn có thể viết như sau:

*const* myCanvas = <*HTMLCanvasElement*>document.getElementById("main\_canvas");

Một số trường hợp hay gặp khác như sau:

*const* { globalState, isShowLoading, isHideLoading } = inject('myGlobals')

=> ở đây ta muốn trích xuất các thành phần con từ myGlobals nhưng hệ thống sẽ báo Property 'globalState' does not exist on type 'unknown'. Vậy nên cần khai báo kiểu dữ liệu cho các biến nói trên ta không thể khai báo như bình thường:

*const* { globalState: any, isShowLoading, isHideLoading } = inject('myGlobals')

lúc này cũ pháp như vậy lỗi, lúc này ta cần loại xác nhận để hỗ trợ xác nhận các biến được trích xuất, ta khai báo như sau:

*const* { globalState, isShowLoading, isHideLoading } = inject('myGlobals') as { globalState: GlobalState; isShowLoading: IsShowLoading ;isHideLoading: IsHideLoading }

##### \*12. general:

Other important TypeScript types

| Type | Explanation |
| --- | --- |
| unknown | the top type. |
| never | the bottom type. |
| object literal | eg { property: Type } |
| void | a subtype of undefined intended for use as a return type. |
| T[] | mutable arrays, also written Array<T> |
| [T, T] | tuples, which are fixed-length but mutable |
| (t: T) => U | functions |

#### *interface:*

##### Kiểu cơ bản

*interface* Props {

popperInlineEnd?: *Boolean | Number*

tag?: *string*

contentContainerTag?: *string*

isRtl?: *Boolean*

[propName: *string*]?: *any => Không quy định số lượng*

}

Ex:

*const* refForm = ref<Props>()

ex:

\\ ta có 1 đối tượng interface hoặc một đối tượng bí danh (aliases)

*type* objectAliases = {

Prop1: *string*

  Prop2: *number*

  Prop3: *boolean*

}

*interface* expArr {

  property?: *objectAliases*[]

=> biểu thị cho thuộc tính property là một mảng các đối tượng *objectAliases*

}

##### Kiểu không quy định số lượng

*interface* myArray {

[propName: *string*]?: *any => Không quy định số lượng*

}

Ex:

let myArray: StringArray;

myArray = ["Bob", "Fred"];

##### \*Kiểu lấy giá trị là key của một interface khác

*interface* Status {

  Verified: *string*

  Rejected: *boolean*

  Pending: *number*

}

*interface* Transition {

status: keyof Status

}

*interface* Transition1 {

status: Status [keyof Status]

}

Ex:

*const* lastTransitions: Transition[] = [

  {

    status: 'Verified',

  },

  {

    status: 'Rejected',

  },

  {

    status: 'Pending',

  },

]

*const* lastTransitions: Transition1[] = [

  {

    status: *string*,

  },

  {

    status: *boolean*,

  },

  {

    status: *number*,

  },

]

*const* position = ['relative', 'absolute', 'static', 'fixed', 'sticky'] as const

position?: typeof position[number]

##### Kiểu function

*interface* SearchFunc {

  (*source*: *string*, *subString*: *string*): *boolean*;

}

=>Kiểu property trên là kiểu function và giá trị output của function là Boolean hoặc void ,…

*let* mySearch: SearchFunc;

 mySearch = *function* (*source*: *string*, *subString*: *string*): *boolean* {

*let* result = *source*.search(*subString*);

   return result > -1;

 };

\*\*\*note: tích hợp vào emit

##### Tham số chỉ đọc: readonly

Không thể thay đổi giá trị chỉ được phép đọc

interface Citizen {

  name: *string*;

  readonly SSN: *number*;

}

*let* personObj: Citizen  = { SSN: 110555444, name: 'James Bond' }

personObj.name = 'Steve Smith'; // OK

personObj.SSN = '333666888'; // Compiler Error

\*\* khai báo 2 phương thức trong interface

*interface* DescribableFunction {

  description: *string*

  (*someArg*: *number*): *boolean*

  (*someArg1*: *string*, *someArg2*: *number*): *boolean*

}

*function* doSomething(*fn*: DescribableFunction) {

  console.log(`${*fn*.description} returned ${fn(6)}${fn('s', 1)}`)

}

N=> Như ta thấy ở trên các method được định nghĩa là 1 function không tên nên nó sẽ được gọi thông qua tên parameter, các method này sẽ giống như overloading phụ thuộc vào biến truyền vào nó sẽ thuộc method nào. Để quy định tên ta làm như sau:

*interface* DescribableFunction {

  description: *string*

  methd1(*someArg*: *number*): *boolean*

  methd2(*someArg1*: *string*, *l*: *number*): *boolean*

}

*function* doSomething(*fn*: DescribableFunction) {

  console.log(`${*fn*.description} returned ${*fn*.methd1(6)}${*fn*.methd2('s', 1)}`)

}

\*\*\*Một cách tổ chức function khác thông qua từ khóa new:

Ta cũng có thể khai báo một khởi tạo đối tượng bên trong interface

*class* SomeObject {

  objName: *string*

  objValue: *number*

*constructor*(*objName*: *string*, *objValue*: *number*) {

    this.objName = *objName*

    this.objValue = *objValue*

  }

  // method

  getObj(): *void* {

    console.log(`${this.objName}:${this.objValue}`)

  }

}

// ---cut---

*interface* SomeConstructor {

  new (*nameObj*: *string*, *objValue*: *number*): SomeObject

}

*function* fn(*Ctor*: SomeConstructor) {

  return new *Ctor*('hello', 123)

}

fn(SomeObject).getObj()

##### Kế thừa interface: Thường được dung cho các interface global cha

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | interface IPerson {  name: string;  gender: string;  }  interface IEmployee extends IPerson {  empCode: number;  }  let empObj:IEmployee = {  empCode:1,  name:"Bill",  gender:"Male"  } |

##### Class kế thừ interface: implement interface

Cũng như các ngôn ngữ khác ts cũng hỗ trợ cho một class cài đặt interface

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | interface IEmployee {  empCode: number;  name: string;  getSalary:(number)=>number;  }  class Employee implements IEmployee {  empCode: number;  name: string;  constructor(code: number, name: string) {  this.empCode = code;  this.name = name;  }  getSalary(empCode:number):number {  return 20000;  }  }  let emp = new Employee(1, "Steve"); |

##### Generic interface:

Ts interface cũng cho phép cài đặt generic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | interface KeyPair<T, U> {  key: T;  value: U;  }  let kv1: KeyPair<number, string> = { key:1, value:"Steve" }; // OK  let kv2: KeyPair<number, number> = { key:1, value:12345 }; // OK |

Giúp linh hoạt kiểu dữ liệu hơn cho interface

##### \*Intersection interface: (&)

Interface cũng cho phép ta kết hợp các interface hiện có để tạo thành các inteface mới có tất các thành viên của các interface được kết hợp

*interface* Colorful {

  color: *string*

}

*interface* Circle {

  radius: *number*

}

*type* ColorfulCircle = Colorful & Circle

lúc này ColorfulCircle sẽ có tất cả các thuộc tính của cả 2 inteface colorful và circle

#### *Function:*

Cấu trúc function

*let* fst: (*a*: *any*, *b*: *any*) *=>* *any* = (*a*, *b*) *=>* *a*;

// or more precisely:

*let* fst: <T, U>(*a*: T, *b*: U) *=>* T = (*a*, *b*) *=>* *a*;

Function void: là function sẽ không return mà chỉ thực hiện chức năng nào đó

*const* nameFunc = (*par*: *any*, *par2*: *boolean*) *=>* {

  console.log(*par*, *par2*)

}

nameFunc('bach', true)

##### \*Function calback**:**

function calback: ta có thể truyền function là params của function khác

*const* nameFunc = (*par*: *any*, par2: (*fpar*: *boolean*) *=>* *string*) *=>* {

  console.log(*par*, par2(false))

}

*const* fnCallback = (*fpar*: *boolean*) *=>* {

  return `${*fpar*}`

}

nameFunc('bach', fnCallback)

* Lúc này ta truyền function chưa thực thi, nếu bạn muốn truyền param cho function trước ngay lúc gắn params thì có thể làm như sau:

*const* nameFunc = (*par*: *any*, par2: () *=>* *string*) *=>* {

  console.log(*par*, par2())

}

*const* fnCallback = (*fpar*: *boolean*) *=>* {

  return *function* sss() {

    return `${*fpar*}`

  }

}

nameFunc('bach', fnCallback(false))

* Lúc này ta có thể gán giá trị trước cho function fnCallback, vào function nameFunc ta chỉ cần thực thi, ở đây ta thấy cấu trúc mới của ‘closure’ ta sẽ nói sau.

*const* fnCallback = (*fpar*: *boolean*) *=>* {

  return *function* sss() {

    return `${*fpar*}`

  }

}

*const* resetEvent : EventSourceFunc = (*info*, successCallback) *=>* {

  event.value = JSON.parse(JSON.stringify(props.event))

  nextTick(() *=>* {

    refForm.value?.resetValidation()

  })

}

##### **\*Closure:**

Glosure là một tập hợp bao gồm một hàm và môi trường nơi hàm đó đươc khai báo.

Glosure sẽ có thể truy cập biến ở 3 phạm vi khác nhau:

1. Biến toàn cục ( global)
2. Biến được khai báo ở ngoài hàm (outer)
3. Biến cục bộ được khai báo ở bên trong hàm closure (local)

Các biến này được lưu trữ trong bộ nhớ Heap của JavaScript và không bị thu hồi sau khi hàm thoát khỏi phạm vi của nó, do đó các hàm trong JavaScript có thể truy cập và sử dụng các biến này trong các lần gọi hàm sau đó.

Closure thường được sử dụng để giữ lại trạng thái của một đối tượng hoặc một biến trong suốt vòng đời của một ứng dụng. Nó cũng được sử dụng trong các pattern như module pattern, trong đó các biến hoặc phương thức được bao bọc trong một hàm và trả về để truy cập từ các phần khác của ứng dụng.

*function* counter() {

*let* count = 0

  return *function* () {

    count++

    console.log(count)

  }

}

*const* myCounter = counter()

myCounter() // 1

myCounter() // 2

myCounter() // 3

* Đặc điểm của closure :
  + **Hàm closures có thể truy cập tới biến của hàm chứa nó, dù cho hàm đó đã return:** 
    - Thông thường, khi một hàm đã **return** thì biến cục bộ trong hàm đó cũng được giải phóng. (đáng lẽ biến cout sẽ được giải phóng vùng nhớ)
    - Nhưng với closure trong JS/TS thì khác, bạn vẫn có thể truy cập đến những biến cục bộ đó ngay cả khi outer function đã thực hiện xong. (ex: lúc này biến cout=0 sẽ được lưu ở head của js)

*function* adder(*n*: *number*) {

*const* intro = 'This answer is '

*const* local = *n*

  return *function* (*number*: *number*) {

*const* result = *number* + local

    console.log(intro + result)

  }

}

*const* adder2 = adder(2)

adder2(10)

Trong ví dụ trên, hàm closures là một hàm không tên function(number). Hàm closures này sử dụng biến cục bộ của outer function là intro và local.

Khi mình gọi hàm adder(2), hàm này thực hiện và kết quả trả về được gán vào biến adder2. Nói cách khác, adder2 chứa **nested function** được trả về từ việc gọi hàm adder(2).

Sau đó, mình gọi adder2(10) và kết quả trả về là 12

Chứng tỏ, hàm closures vẫn có thể truy cập tới biến cục bộ của outer function là intro, local ngay cả khi hàm outer adder2 đã thực hiện xong.

##### **Tham số Rest:**

*Một function có thể có 1 hoặc nhiều tham số. Sẽ có những trường hợp ta không biết được sẽ truyền bao nhiêu tham số trong function. Để giải quyết vấn đề này TypeScript cung cấp cho chung ta tham số Rest với khai báo là 3 dấu … trong function.*

function Greet(greeting: string, ...names: string[]) {

return greeting + " " + names.join(", ") + "!";

}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"

L\*\*Lưu ý: tham số rest không được đặt ở tham số đầu tiên của function

|  |  |
| --- | --- |
| 1  2  3 | function Greet(...names: string[], greeting: string) { // Compiler Error  return greeting + " " + names.join(", ") + "!";  } |

##### **\*Tham số destructuring:**

Cú pháp phá hủy cấu trúc

*function* sum({ *a*, *b*, *c* }: { a: *number*; b: *number*; c: *number* }) {

  console.log(*a* + *b* + *c*);

}

// Same as prior example

*type* ABC = { a: *number*; b: *number*; c: *number* };

*function* sum({ *a*, *b*, *c* }: ABC) {

  console.log(*a* + *b* + *c*);

}

##### **\*Khả năng của function:**

Loại trả về void không nhất thiết là không trả về thức gì đó, Nói cách khác đây là một kiểu hàm ngữ cảnh với voidkiểu trả về ( type vf = () => void), khi thực hiện có thể trả về bất kỳ giá trị nào khác, nhưng nó sẽ bị bỏ qua

*interface* voidFunc {

  (): *void*

}

Hoặc

*type* voidFunc = () *=>* *void*

*const* f1: voidFunc = () *=>* {

  return true

}

*const* f2: voidFunc = () *=>* true

*const* f3: voidFunc = *function* () {

  return 1

}

*const* v1 = f1()

*const* v2 = f2()

*const* v3 = f3()

Dù trả về giá trị nhưng vẫn hợp lệ và các biến được gán vẫn mang kiểu giá trị void (v1,v2,v3)

#### *class:*

*Một Class thì gồm các phần như constructor, thuộc tính và phương thức.*

class Employee {

empCode: number;

empName: string;

constructor(code: number, name: string) {

this.empName = name;

this.empCode = code;

}

getSalary() : number {

return 10000;

}

}

let emp = new Employee(100,"Steve");

##### Kế thừa: **extends**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Person {  name: string;    constructor(name: string) {  this.name = name;  }  }  class Employee extends Person {  empCode: number;    constructor(empcode: number, name:string) {  super(name);  this.empCode = empcode;  }    displayName():void {  console.log("Name = " + this.name + ", Employee Code = " + this.empCode);  }  }  let emp = new Employee(100, "Bill");  emp.displayName(); // Name = Bill, Employee Code = 100 |

##### Super:

class Car {

name: string;

constructor(name: string) {

this.name = name;

}

run(speed:number = 0) {

console.log("A " + this.name + " is moving at " + speed + " mph!");

}

}

*Để truy cập đến class cha từ class con để gọi hàm và biến ta sử dụng từ khoá super*

*Ta có thể kế thừ khởi tạo từ cha*

constructor(name: string) {

super(name);

}

Ta cũng có thể gọi method được cài đặt từ cha:

run(speed = 150) {

console.log('A Mercedes started')

super.run(speed);

}

##### Cài đặt interface:

*Chúng ta có thể implements một hoặc nhiều Interface cho Class. Ví dụ Class Employee sau đây.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | interface IPerson {  name: string;  display():void;  }  interface IEmployee {  empCode: number;  }  class Employee implements IPerson, IEmployee {  empCode: number;  name: string;    constructor(empcode: number, name:string) {  this.empCode = empcode;  this.name = name;  }    display(): void {  console.log("Name = " + this.name + ", Employee Code = " + this.empCode);  }  }  let per:IPerson = new Employee(100, "Bill");  per.display(); // Name = Bill, Employee Code = 100  let emp:IEmployee = new Employee(100, "Bill");  emp.display(); //Compiler Error: Property 'display' does not exist on type 'IEmployee' |

##### Override method:

*Chúng ta có thể override lại method của lớp cha giống như trong Java.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | class Car {  name: string;    constructor(name: string) {  this.name = name;  }    run(speed:number = 0) {  console.log("A " + this.name + " is moving at " + speed + " mph!");  }  }  class Mercedes extends Car {    constructor(name: string) {  check this.name err=> 'super' must be called before accessing 'this' in the constructor of a derived class.  super(name);  check this.name ss=> Mercedes-Benz GLA  }    run(speed = 150) {  console.log('A Mercedes started')  super.run(speed);  }  }  class Honda extends Car {    constructor(name: string) {  super(name);  }    run(speed = 100) {  console.log('A Honda started')  super.run(speed);  }  }  let mercObj = new Mercedes("Mercedes-Benz GLA");  let hondaObj = new Honda("Honda City")  mercObj.run(); // A Mercedes started A Mercedes-Benz GLA is moving at 150 mph!  hondaObj.run(); // A Honda started A Honda City is moving at 100 mph! |

##### Abstract class: Abstract

1. Mục đích: Abstract class được tạo ra để định nghĩa một bộ khung (framework) cho các lớp con, trong khi class thường được tạo ra để biểu diễn một đối tượng hoặc một tập hợp các thuộc tính và phương thức.
2. Sử dụng abstract method: Abstract class có thể chứa các abstract method, trong khi class thường không chứa abstract method. Abstract method là một phương thức không có thực thi và yêu cầu lớp con của nó định nghĩa lại phương thức đó.
3. Tạo đối tượng: Chúng ta không thể tạo đối tượng từ một abstract class, trong khi chúng ta có thể tạo đối tượng từ một class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | abstract class Person {  name: string;    constructor(name: string) {  this.name = name;  }  display(): void{  console.log(this.name);  }  abstract find(string): Person;  }  class Employee extends Person {  empCode: number;    constructor(name: string, code: number) {  super(name); // must call super()  this.empCode = code;  }  find(name:string): Person {  // execute AJAX request to find an employee from a db  return new Employee(name, 1);  }  }  let emp: Person = new Employee("James", 100);  emp.display(); //James  let emp2: Person = emp.find('Steve'); |

*Class mà extends một abstract class phải bắt buộc sử dụng từ khóa super trong constructor của mình.*

##### Overloading:

Cho phép cài đặt phương thức của class cùng tên nhưng nhiều kiểu chức năng, nhiều loại tham số khác nhau.

Overloading trong ts sẽ bắt buộc bao gồm 2 phần đó là phần overloading signatures và phần implement signature

\*\*\* lưu lỹ phần kiểu dữ liệu của phần overloading thực thi phải bao quát được tất cả overloading signatures

Khi đó ta sẽ có phương thức đã được nạp chồng

Ví dụ:

class Greeter {

message: string;

constructor(message: string) {

this.message = message;

}

// Overload signatures

greet(person: string): string;

greet(persons: string[]): string[];

// Implementation signature

greet(person: unknown): unknown {

if (typeof person === 'string') {

return `${this.message}, ${person}!`;

} else if (Array.isArray(person)) {

return person.map(name => `${this.message}, ${name}!`);

}

throw new Error('Unable to greet');

}

}

##### Phân biệt interface, abstract class và class

* Interface là thành phần khác biệt nhất so với 2 thành phần còn lại, interface chỉ là một cấu trúc, nó cung cấp và khai báo những property hay method nó chỉ định cho class cần thực hiện, nó không cung cấp bất kỳ mã cũng như chi tiết thực thi cho các phương thức

*interface* animal {

  run(*e*: 'updateOtp', *val*: *string*): *void*

}

Run ở đây chỉ là struct chưa được cài đặt

Khi muốn sử dụng thì nó sẽ chỉ định cho class thực hiện thông qua ‘implement’

*class* dog implements animal {

  run(*e*: *number*, *val*: *string*) {

    console.log(*e*, *val*)

  }

}

Lúc này ta sẽ cài đặt phương thức ở class

* Abstract class là một lớp trừu tượng, là một class nhưng không thể thực thi như class không thể tạo đối tượng để thực thi, Một abstract class là một class mà không thể được khởi tạo trực tiếp, nó chỉ có thể được kế thừa bởi một class con. Abstract class cung cấp một cấu trúc chung cho các class con và có thể chứa một số phương thức abstract (không có thực thi) hoặc phương thức có thực thi
* Class: Một class là một mô hình hóa của một đối tượng trong thế giới thực. Nó chứa cấu trúc của một đối tượng và các phương thức để thực hiện các hành động trên đối tượng.

#### *export:*

export *const* useCalendar

export *const* useCalendar = (*event*: Ref<Event | NewEvent>) *=>* {

...

return {

    refCalendar,

    calendarOptions,

  }

}

#### *cycle hook:*

##### a. Computed:

*const* isAppRtl = computed({

   get() {

    return config.app.isRtl.value

    },

  set(*value*: typeof config.app.isRtl.value) {

    config.app.isRtl.value = *value*

    localStorage.setItem(`${config.app.title}-isRtl`, *value*.toString())

    \_setAppDir(*value* ? 'rtl' : 'ltr')

  },

})

*const* startDateTimePickerConfig = computed(() *=>* {

*const* config: Options = { enableTime: true, dateFormat: 'Y-m-d H:i' }

  if (event.value.end)

    config.maxDate = event.value.end

  return config

})

##### b. Mounted:

*const* updatePopper = async () *=>* {

...

}

onMounted(updatePopper)

 // 👉 onMounted

  onMounted(() *=>* {

    ...

  })

##### c. Watch:

<https://www.vuemastery.com/blog/vues-watch-vs-watcheffect-which-should-i-use/>

* Loại đơn

watch(prop, *val* *=>* {

. . .

  }, { immediate: true, deep: true })

* Loại kép

watch([prop1, prop2], *([newProp1, newProp2], [prevProp1, prevProp2])* *=>* {

. . .

stopWatch () => Dừng watchEffect HERE

  }, { immediate: true, deep: true })

##### d. WatchEffect:

Ta cũng có thể có nhiều watchEffect, mỗi cái sẽ watch 1 số biến cụ thể:

watchEffect(() => {

console.log(state.count)

})

watchEffect(() => {

console.log(state.double)

})

##### e. Setup: thay thế cho created

\*\*\*Note: - Khi không có template ta sẽ sử dụng dạng `optional api`

* Khi có template ta sẽ sử dụng composition api (setup lang=`ts`)

#### *key word:*

##### 1. StructuredClone: Clone một struct

structuredClone(data)

##### 2. Ref:

*Ref nhận một giá trị bên trong và trả về một đối tượng ref phản ứng và có thể thay đổi. Đối tượng ref có một thuộc tính duy nhất .value trỏ đến giá trị bên trong*

* Cách khai báo:

const count = ref(0);

* Muốn thay đổi giá trị của count ta phải thông qua ‘value’

count.value++;

* Ta có thể khai báo kiểu dữ liệu cho ref

ref<type>(value)

*type: có thể là interface, class, number, string, Boolean, Object, …*

* Muốn gọi sử dụng biến ref

Setup => return biến ref

return { count};

template:

<button @click="increaseCount">count is: {{ count }}</button>

*Ref hữu ích cho các nguyên hàm (chuỗi, số, boolean, v.v.)*

* Mở rộng:

*const* items = ['Item 1', 'Item 2', 'Item 3', 'Item 4'] as const

*const* select = ref<typeof items[number]>()

Quy định kiểu ref cho “select” là các loại được quy định trong mảng “items”

=> selected chỉ nhận các giá trị:

"Item 1" | "Item 2" | "Item 3" | "Item 4" | *undefined*

##### 3. Reactive:

*Phản ứng trả về một bản sao phản ứng của đối tượng. Chuyển đổi phản ứng là "deep"—nó ảnh hưởng đến tất cả các thuộc tính lồng nhau. Trong quá trình triển khai dựa trên*[*Proxy ES2015*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy)*, proxy được trả về không bằng với đối tượng ban đầu. Bạn nên làm việc độc quyền với proxy phản ứng và tránh dựa vào đối tượng ban đầu.*

* Cách khai báo:

const state = reactive({

count: 0

});

* Muốn thay đổi giá trị của count ta không cần phải thông qua ‘value’ như ref:

state.count++;

*reactive hữu ích cho các đối tượng và mảng.*

* Muốn gọi sử dụng biến reactive

Setup => return biến reactive

return { state};

template:

<button @click="increaseCount">count is: {{ state.count }}</button>

* Lưu ý:

*Reactive và ref có thể kết hợp lẫn nhau*

*Đối với ref khi ta truyền một đối tượng vào ref nó sẽ trả về một đối tượng đã được chuyển qua reactive*

const state = ref({

count: 0

});

Lúc này để thay đổi giá trị biến:

state.value.count++;

***\*\*Ngược lại để đổi từ reactive sang ref trong trường hợp sau***

* Đối tượng reactive:

const state = reactive({

count: 0

});

Khi ta muốn setup return các giá trị của đối tượng reactive thông qua detructoring => …state

return { ...state };

nhằm mục đích sử dụng

<button @click="increaseCount">count is: {{ count }}</button>

Thay vì state.count

Lúc này giá trị count sẽ không tồn tại, không thể trả giá trị của đối tượng reactive qua setup => để khắc phục ta thực hiện việc đổi reactive qua ref thông qua “toRef”

return { ...toRefs(state };

Lúc này trên template ta có thể sử dụng giá trị của reactive (count)

<button @click="increaseCount">count is: {{ count }}</button>

##### 4. Prop: defineProps

*const* props = withDefaults(defineProps<Type>(), {

=>Type kiểu dữ liệu cho prop thường là 1 interface

  property: ‘value’, value’=> giá trị mặc định cho prop

})

*Ex:*

*interface* Props {

  collapsed?: *boolean*

  noActions?: *boolean*

  actionCollapsed?: *boolean*

  actionRefresh?: *boolean*

  actionRemove?: *boolean*

  title?: *string*

}

*const* props = withDefaults(defineProps<Props>(), {

  collapsed: false,

  noActions: false,

  actionCollapsed: false,

  actionRefresh: false,

  actionRemove: false,

  title: undefined,

})

##### 5. Emit: defineEmits

*interface* Emit {

  (*e*: 'collapsed', *isContentCollapsed*: *boolean*): *void*

  (*e*: 'refresh', hideOverlay: () *=>* *void*): *void*

  (*e*: 'trash'): *void*

}

*const* emit = defineEmits<Emit>()

 emit('collapsed', isContentCollapsed.value)

 emit('refresh', hideOverlay)

 emit('trash')

or

defineEmits<{

  (*e*: 'change', *id*: *string*): *void*

}>()

##### 6. DefineExpose:

##### 7. ToRaw:

*Trả về một raw là nguyên bản của proxy do vue tạo ra*

*const* foo = reactive({ a: 1 })

Lúc này foo là một proxy: Proxy {a: 1}

Để lấy nguyên bản của nó là {a: 1}:

toRaw(foo) => {a:1}

Tương tự

*const* foo2 = ref({ a: 1 }) => nó là 1 ref và trả về 1 proxy thông qua value

toRaw(foo2.value) =>{a:1}

##### 8. Provide / inject:

* Sử dụng ở app global

app.provide('$msg', 'hello')

* Sử dụng cơ bản:

Tại nơi cung cấp dịch vụ:

import { provide } from 'vue'

provide('$msg', 'North Pole')

Tại nơi sử dụng dịch vụ:

import { inject } from 'vue'

const $msg = inject('$msg')

*Cấu hình nên property global*

##### 9. Toán tử in:

*Toán tử này cho phép bạn kiểm tra xem một giá trị có nằm trong một object hay không*

*interface* Fish { swim: () *=>* *void* }

*interface* Bird { fly: () *=>* *void* }

*function* move(*animal*: Fish | Bird) {

  if ('swim' in *animal*)

    return *animal*.swim()

  return *animal*.fly()

}

*const* dogs: Fish = {

  swim: () *=>* console.log('swins'),

}

move(dogs)

##### 10. Toán tử is:

Vị ngữ: parameterName is Type

Khẳng định kiểu trả về của một đối tượng là một type của một parameter

*interface* Fish { swim: () *=>* *void* }

*interface* Bird { fly: () *=>* *void* }

*function* getSmallPet(): Fish | Bird {

*const* cat: Fish = {

    swim: () *=>* console.log('swim'),

  }

*const* dogs: Bird = {

    fly: () *=>* console.log('fly'),

  }

  return dogs

}

*function* isFish(*pet*: Fish | Bird): *pet* is Fish {

  return (*pet* as Fish).swim !== undefined

}

// ---cut---

// Both calls to 'swim' and 'fly' are now okay.

*const* pet = getSmallPet()

if (isFish(pet))

  pet.swim()

else

  pet.fly()

* Khẳng định function isFish sẽ trả về kiểu dữ liệu Fish của interface pet, giúp thu hẹp biến pet thành một loại cụ thể là fish

##### \*11. Toán tử Postfix: (!)

Viết ! sau bất kỳ biểu thức nào thực sự là một kiểu khẳng định rằng giá trị không phải là null hoặc undefined:

*function* liveDangerously(*x*?: *number* | *null*) {

  // No error

  console.log(*x*!.toFixed())

}

##### \*12. ReadonlyArray:

Là 1 interface khai báo cho thể loại mảng giúp cho mảng này không bị thay đổi

##### \*13. keyof: Lấy giá trị key

*type* Point = { x: *number*; y: *number* };

*type* P = keyof Point;

*const* params: P = // sẽ là giá trị ‘x’ hoặc ‘y’

*type* Arrayish = { [*n*: *number*]: *unknown* };

*type* A = keyof Arrayish;

*const* params: A = // sẽ là number

##### \*14. typeof: Lấy kiểu giữ liệu

// Prints "string"

console.log(typeof "Hello world");

//dùng cách này để định nghĩa mảng kiểu giữ liệu cho thuộc tính

*const* position = ['relative', 'absolute', 'static', 'fixed', 'sticky'] as const

position?: typeof position[number]

*const* MyArray = [

  { name: 'Alice', age: 15 },

  { name: 'Bob', age: 23 },

  { name: 'Eve', age: 38 },

]

*type* Person = typeof MyArray[number]

=> *type* Person {

  name: *string*

  age: *number*

}

*type* Age = typeof MyArray[number]['age']

*type* Age = *number*

##### \*15. Conditional type:

Là một biểu thức điều kiện với cấu trúc điều kiện ? đúng: sai

*interface* Animal {

  live(): *void*

}

*interface* Dog extends Animal {

  woof(): *void*

}

*type* Example1 = Dog extends Animal ? *number* : *string => kiểm tra Dog có kế thừa Animal không*

*type* Example1 = *number*

*type* Example2 = *RegExp* extends Animal ? *number* : *string*

*type* Example2 = *string*

Giúp cho việc viết tổng quát tốt hơn

Một ví dụ cho rút gọn overloading

*interface* IdLabel {

  id: *number* /\* some fields \*/

}

*interface* NameLabel {

  name: *string* /\* other fields \*/

}

*function* createLabel(*id*: *number*): IdLabel

*function* createLabel(*name*: *string*): NameLabel

*function* createLabel(*nameOrId*: *string* | *number*): IdLabel | NameLabel

*function* createLabel(*nameOrId*: *string* | *number*): IdLabel | NameLabel {

  throw 'unimplemented'

}

Khi ta biết chắc từ loại 1 string và 1 number và một trường tổng quát string| number ta có thể tóm gọm phương thức

): IdLabel | NameLabel =>

*type* NameOrId<T extends *number* | *string*> = T extends *number*

  ? IdLabel

  : NameLabel;

Phụ thuộc vào T là gì sẽ so sánh xem có kế thừa number không để quy định kiểu IdLabel hay NameLabel

Tiếp tục ta chuyển đổi params với generic là

<T extends number | string> => T là number hoặc string

((*nameOrId*: *string* | *number)*)=> (name: T)

*function* createLabel<T extends *number* | *string*>(*idOrName*: T): NameOrId<T> {

  throw "unimplemented";

}

#### *Quy tắc chuyển đổi dữ liệu:*

#### *Mẫu dữ liệu:*

#### *Enum:*

*enum* ErrorCodes {

  BAD\_REQUEST = 400,

  UNAUTHORIZED = 401,

  FORBIDDEN = 403,

  NOT\_FOUND = 404,

  INTERNAL\_SERVER\_ERROR = 500,

}

*function* handleError(*errorCode*: *number*) {

  switch (*errorCode*) {

    case ErrorCodes.BAD\_REQUEST:

      console.error('Bad request')

      break

    case ErrorCodes.UNAUTHORIZED:

      console.error('Unauthorized')

      break

    case ErrorCodes.FORBIDDEN:

      console.error('Forbidden')

      break

    case ErrorCodes.NOT\_FOUND:

      console.error('Not found')

      break

    case ErrorCodes.INTERNAL\_SERVER\_ERROR:

      console.error('Internal server error')

      break

    default:

      console.error('Unknown error')

  }

}

#### *eeee*



Optional api và composition api

Vue 3

|  |  |  |
| --- | --- | --- |
| stt | Optional api | Composition api |
| 1 | export default {  name: 'MyComponent',  components: { ... },  props: { ... },  data() { return { ... } },  computed: {},  mounted() { ... }  methods: {}  } | <script setup>  import { ref, reactive, computed } from 'vue';  // presentation  const salute = ref('Hello world');  const greeting = computed(() => {  return `${salute.value}, I am ${persona.name} and have ${persona.age} years old`.  });  </script> |
| 2 | Các logic code sẽ bị phân tán vào các option (data, computed, methods, watch ...),  khó kiểm soát, việc kéo lên kéo xuống rất tốn thời gian đối với file lớn | Chỉ cần sử dụng tag <script setup>  nhóm lại những logic lại với nhau |
| 3 | Sự hỗ trợ của TypeScript đối với Option API luôn luôn là điểm hạn chế cố hữu từ lâu do chúng ta phải quá phụ thuộc vào các magic của Vue | Tương thích tốt hơn với TS,   chúng ta sẽ viết lại một cách gần gũi hơn với JavaScript thuần bằng cách sử dụng các hàm và khiến cho chúng ta dễ dàng kiểm soát hơn |
| 4 | Bị giới hạn trong phạm vi của Vue Component, và từ đó cần phải sử dụng đến mixin và slot rất dài dòng | có thể tách được các hàm ra và tái sử dụng chúng theo cách ta muốn một cách dễ dàng |
| 5 | Các component nhỏ | Các component Lớn |
| 6 | Các component không thao tác với template | Các component thao tác nhiều với template |
| 7 | Để sử dụng và thao tác dễ dàng hơn các content trong params setup    setup(*props*, { *slots* }) | Thao tác thông qua các define, và interface import vv (getCurrentInstance) |
| 8 |  | Ưu tiên |

Mẫu compositionAPI:

<script lang="ts" setup>

import { inject } from 'vue'

import type { I18nLanguage } from '@layouts/types'

// Khai báo interface

*interface* Props {

  collapsed?: *boolean*

}

*interface* Emit {

  (*e*: 'collapsed', *isContentCollapsed*: *boolean*): *void*

  (*e*: 'refresh', hideOverlay: () *=>* *void*): *void*

  (*e*: 'trash'): *void*

}

// Khai báo prop

*const* props = withDefaults(defineProps<Props>(), {

  collapsed: false,

})

// Khai báo định nghĩa

*const* emit = defineEmits<Emit>() // định nghĩa đối tượng emit

*const* $msg = inject('$msg')// định nghĩa đối tượng global

// mảng

*const* i18nCompLanguages: I18nLanguage[] = [

  {

    label: 'English',

    i18nLang: 'en',

  },

]

// function

*const* handleLangChange = (*lang*: *string*) *=>* {

  isAppRtl.value = *lang* === 'ar'

  emit('collapsed', isContentCollapsed.value)

  emit('refresh', hideOverlay)

}

</script>

<!-- template -->

<template>

  <*NavBarI18n*

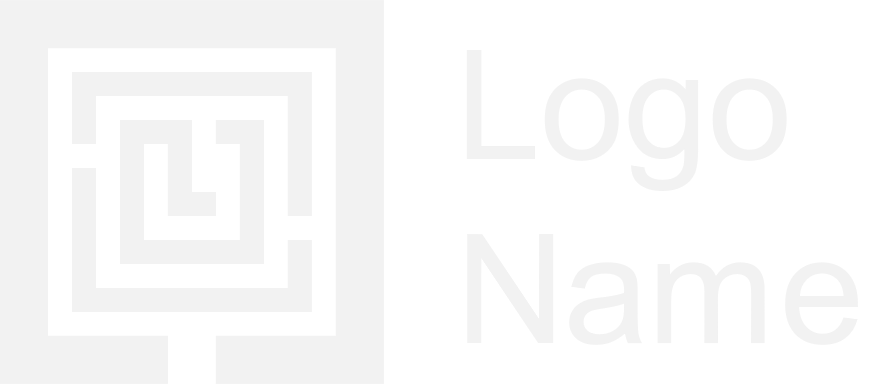
    :languages="i18nCompLanguages"

    @change="handleLangChange"

  />

</template>

|  |
| --- |
| setting global property  and lodash js  Vue 3 |
| CLS-5 Street Address  City, ST ZIP Code  Phone  Email |



## \*Seting Global Property

## 

1. Global property
   1. B1: Khởi tạo interface cho biến global

Ta có interface ‘GlobalStateReadOnly’ để định nghĩa cấu trúc thành phần các global thể loại chỉ được phép đọc

*interface* GlobalStateReadOnly {

  isRenderReadOnly: *boolean*

  isLoadingReadOnly: *boolean*

}

Ta có interface ‘GlobalState’ để định nghĩa cấu trúc thành phần các global

*interface* GlobalState {

  isRender?: *boolean*

  isLoading?: *boolean*

  [*key*: *string*]: *any*

}

* 1. B2: khơi tạo giá trị mặc định cho biến global

Biến chỉ đọc sử dụng interface ‘GlobalStateReadOnly’

*const* globalStateReadOnly = reactive<GlobalStateReadOnly>({

  isRenderReadOnly: false,

  isLoadingReadOnly: false,

})

Biến global bình thường sẽ dùng interface ‘GlobalState’

*const* globalState = reactive<GlobalState>({

  isRender: false,

  isLoading: false,

})

* 1. B3: tạo hàm ‘changeValue’ cho biến global mặc định

\*\*\* lưu ý global read only sẽ không chứa hàm ‘changeValue’

*const* changeValue = (*key*: *string*, *value*: *any*) *=>* {

  globalState[*key*] = *value*

}

Để custom thì interface ‘GlobalState’ phải chứa trường custom name động

  [*key*: *string*]: *any*

* 1. B4: export

export *const* myGlobalsReadOnly = readonly({

  ...toRefs(globalStateReadOnly),

})

export *const* myGlobals = ({

  ...toRefs(globalState),

  changeValue,

})

* 1. B5: gọi ở global

Tại main.js ta thực hiện ‘provide’ các global

app.provide('myGlobals', myGlobals)

app.provide('myGlobalsReadOnly', myGlobalsReadOnly)

* 1. Sử dụng
* Khai báo interface
  + Tạo interface:
    - Cách1

*interface* globalState {

  isRender: *boolean*

}

*interface* IsShowLoading {

  (): *void*

}

*type* IsHideLoading = IsShowLoading

*const* { globalState, isShowLoading, isHideLoading } = inject('myGlobals') as { globalState: GlobalState; isShowLoading: IsShowLoading ;isHideLoading: IsHideLoading }

* + - Cách 2

*interface* globaleInterface {

  isRender: *boolean*

  isLoading: *boolean*

  isHideLoading(): *void*

  isShowLoading(): *void*

  changeValue(*key*: *any*, *value*: *any*): *void*

}

*const* myGlobal = inject('myGlobals') as globaleInterface

*const* { isRender, changeValue, isShowLoading } = myGlobal

\*\* hạn chế không thể gọi được tại các file ngoài khu vực app

Để khắc phục điều đó => ta sẽ sử dụng thông qua global public

\*\*\* các biến global chỉ được sử dụng trong các trường hợp đặc biệt là dùng nhiều nơi và dữ liệu đơn giản, ít thay đổi. Khi dữ liệu lớn cần quản lý hay thay đổi ta sẽ sử dụng store để lưu trữ và quản lý, không nên lạm dụng quá nhiều biến global. Vì rất khó quản lý và bảo trì.

## \*Seting Lodash

Lodash sẽ được custom theo cấu trúc của provide/inject

Tại file propertyts ta sẽ cung cấp thêm nguồn provide

export *const* globals = ({

  ...toRefs(globalState),

  changeValue,

  \_, \\ lodash

})

Khi sử dụng

Khai báo interface cho lodash

*interface* globaleInterface {

  \_: \_.LoDashStatic

}

Thêm loại dữ liệu là interface \_.LoDashStatic để cung cấp extention gợi ý khi code từ interface.

Inject từ provide globals => chứa lodash

*const* global = inject('globals') as globaleInterface

gọi property ‘\_’ lodash để sử dụng

*const* { \_ } = global

ex:

console.log(\_.isArray(['12']))

hoặc

console.log(window.\_.isArray(['12']))

\*\*\* hoặc ta có thể sử dụng lodashUtil để không cần khai báo interface mỗi lần gọi, và đặc biệt lodashUtil sẽ được gọi ở tất cả các file ts, còn provide chỉ được sử dụng trong app

import { \_ } from '@/utils/LodashUtil'

console.log(\_.isArray(['12']))

đơn giản hơn, và đặc biệt lodash sẽ được gọi từ global không phải khởi tạo lại thư viện.

*const* obj = {

  name: 'John',

  age: null,

  age1: 1,

  address: {

    city: 'New York',

    state: null,

    state3: 'New York1',

    state4: 55,

    state5: {

      a: null,

      b: '',

      c: undefined,

      d: 5656,

      f: 'adsada',

    },

  },

}

*function* liveDangerously(*x*?: *number* | *null*) {

  // No error

  console.log(*x*!.toFixed())

}

// xóa null object

static omitByDeep: *any* = (*objs*: *any*) *=>* {

    return \_.transform(*objs*, (*result*: *any*, *value*, *key*) *=>* {

      if (\_.isObject(*value*)) {

*const* newVal = ObjectUtil.omitByDeep(*value*)

        if (!\_.isEmpty(newVal))

*result*[*key*] = newVal

      }

      else if (\_.isNumber(*value*) || 0 || (!\_.isNil(*value*) && !\_.isEmpty(*value*))) {

*result*[*key*] = *value*

      }

    })

  }

*const* result = omitByDeep(obj)

## \*Seting css/scss

#### @import

#### @USe:

[@use](https://viblo.asia/u/use) khá giống [@import](https://viblo.asia/u/import) nhưng có những sự khác biệt quan trọng sau:

* File sẽ chỉ dc gọi vào một lần duy nhất bất kể bạn sử dụng [@use](https://viblo.asia/u/use) file đó bao nhiêu lần
* Các biến bắt đầu bằng underscore (\_) hoặc hyphen (-) sẽ được coi là biến local (private)
* Tất cả những variables, hay functions private sẽ có namespace riêng khi tao sử dụng [@use](https://viblo.asia/u/use) tới file đó, ví dụ:

@use 'buttons'; // creates a `buttons` namespace

@use 'forms'; // creates a `forms` namespace

#### @forward

Ta có

#### Sự khác nhau

#### CÁCh chuyển Đổi

B1: Tạo một module scss trong thư mục styles

Quy tắc tạo biến:

$color-border-default: red;

B2: Import/use/forward để sử dụng ở những nơi cần

Để sử dụng biến

border: 1px solid $color-border-default;

nhưng đối với các thư viện sử dụng cấu hình config biến là kiểu css thì ta không thể sử dụng theo cách trên

ví dụ:

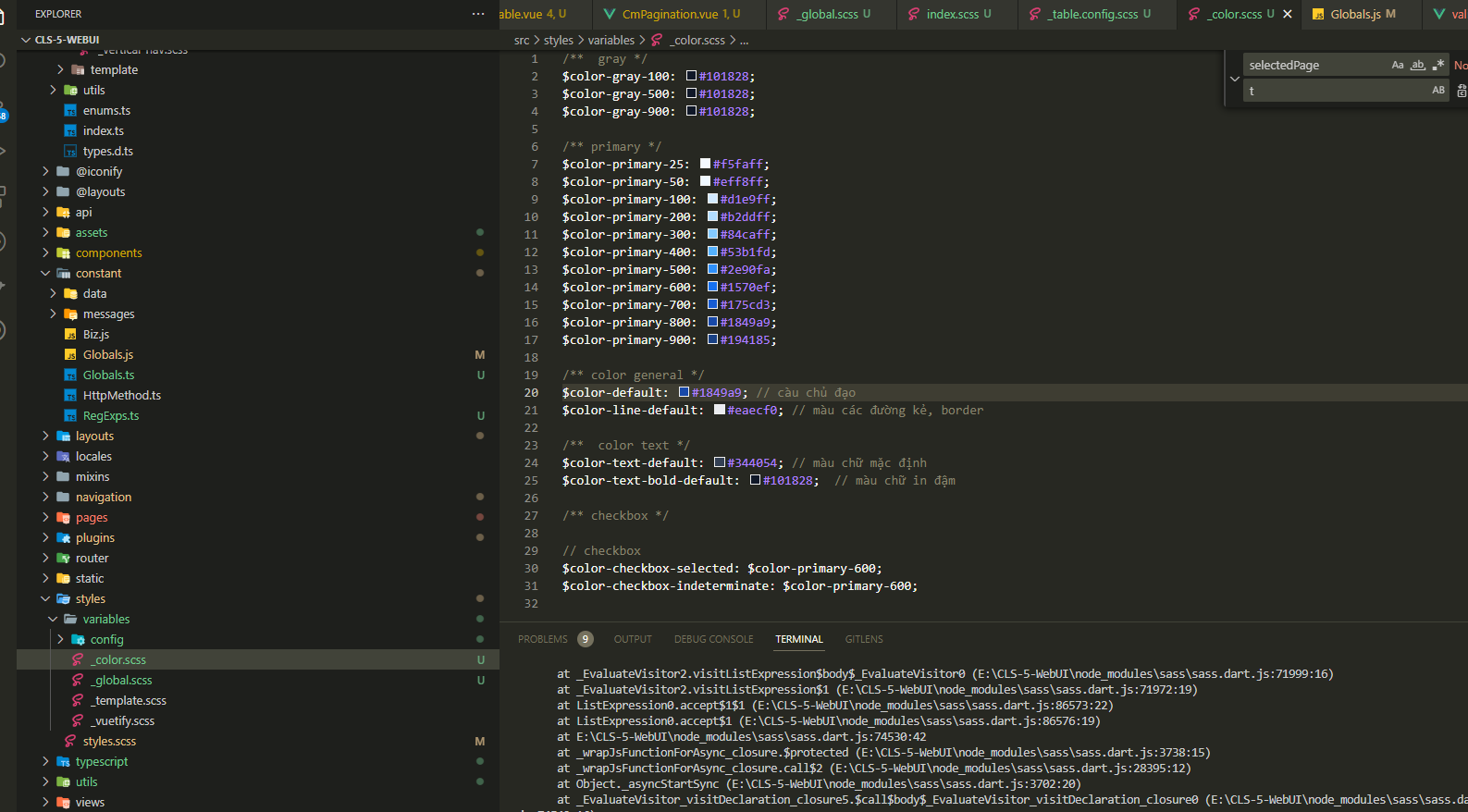
--easy-table-border: 1px solid $color-border-default;

* Lúc này –-easy-table-border sẽ không hiểu biến $color-border-default là gì và nó sẽ nhận giá trị chuỗi

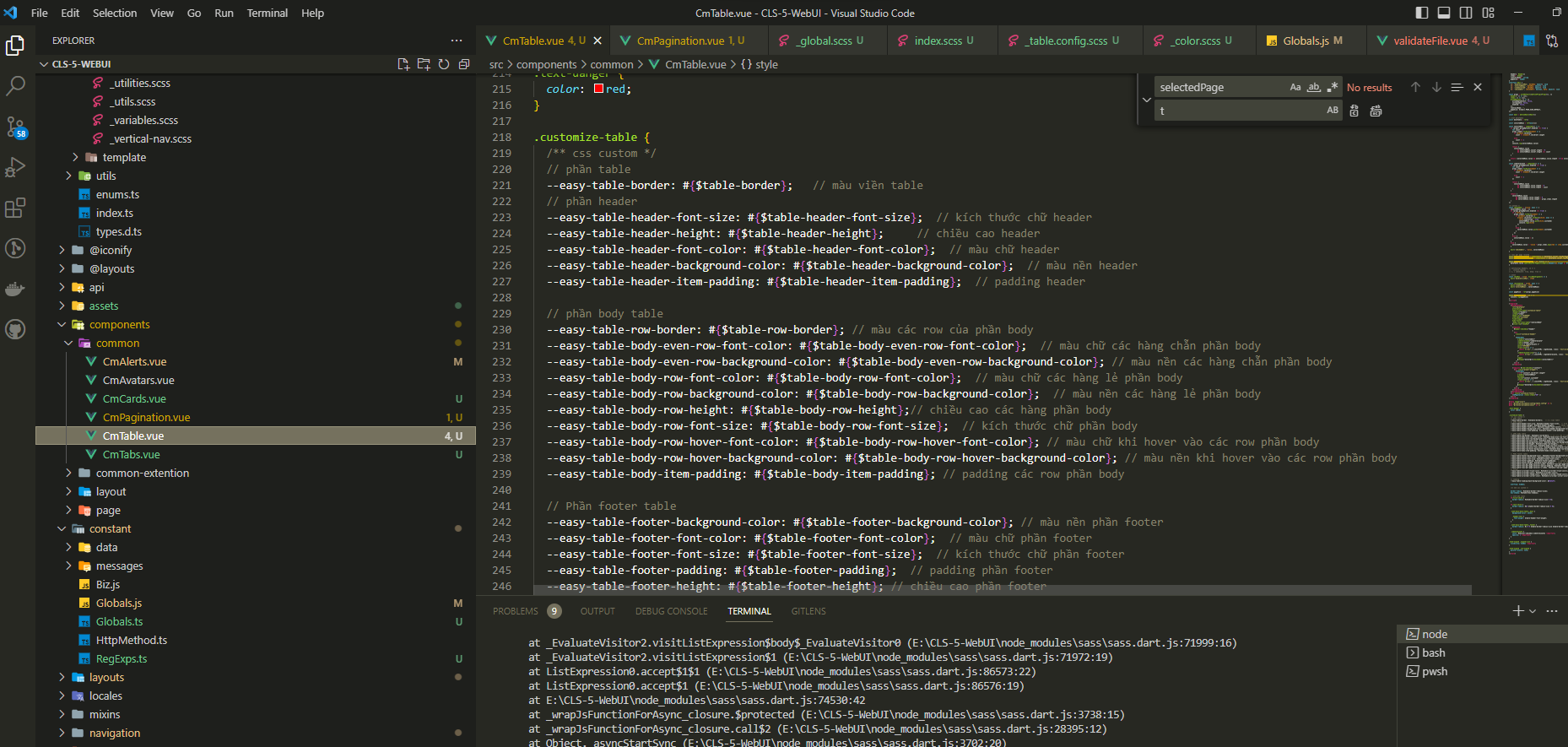
Để khắc phục điều đó ta có cú pháp gán biến trong sass như sau:

--easy-table-border: #{1px solid #{$color-border-default}};

* Lúc này nó #{variable} nên nó sẽ đọc các biến và sử lý trước khi gán cho biến –-easy-table-border

 Text

Description automatically generated Text

Description automatically generated

Cơ bản nhất ví dụ ta sẽ tạo ra 2 loại table: basic và table group

Lúc này ở cả 2 component table basic và group ta đều kế thừa biến từ config table, ngoài ra trong quá trình config ở conponent ta vẫn có thể import file global và color để sử dụng các dịch vụ chung ngoài các thành phần của table config:

Dễ hiểu hơn:

Trong table sẽ có một số thành phần như checkbox thì css màu săc của checkbox ta sẽ sử dụng ở file color chứ không khởi tạo thêm config cho checkbox ở file config table.

\*\*Note: tất cả các api scss của component đều được cài đặt ở style component

Tên có thể loại scss hoặc css, nhưng ở file config table thì phải là loại scss