```typescript
function returnItem<T>(para: T): T {
    return para
}
```

可以看到，泛型给予开发者创造灵活、可重用代码的能力

8.2. 使用方式

泛型通过 `<>` 的形式进行表述，可以声明：

- 函数
- 接口
- 类

8.2.1. 函数声明

声明函数的形式如下：

```typescript
function returnItem<T>(para: T): T {
    return para
}
```

定义泛型的时候，可以一次定义**多个类型参数**，比如我们可以同时定义泛型 `T` 和 泛型 `U`：

```typescript
function swap<T, U>(tuple: [T, U]): [U, T] {
    return [tuple[1], tuple[0]];
}

swap([7, 'seven']); // ['seven', 7]
```

8.2.2. 接口声明

声明接口的形式如下：

```typescript
interface ReturnItemFn<T> {
    (para: T): T
}
```

那么当我们想传入一个number作为参数的时候，就可以这样声明函数:

```typescript
const returnItem: ReturnItemFn<number> = para => para
```

8.2.3. 类声明

使用泛型声明类的时候，既可以作用于类本身，也可以作用与类的成员函数

下面简单实现一个元素同类型的栈结构，如下所示:

```typescript
class Stack<T> {
    private arr: T[] = []

    public push(item: T) {
        this.arr.push(item)
    }

    public pop() {
        this.arr.pop()
    }
}
```

使用方式如下:

```typescript
const stack = new Stacn<number>()
```

如果上述只能传递 `string` 和 `number` 类型，这时候就可以使用 `<T extends xx>` 的方式猜实现**约束泛型**，如下所示:

```ts
type Params = string | number

class Stack<T extends Params> {
    private arr: T[] = []

    public push(item: T) {
        this.arr.push(item)
    }

    public pop() {
        this.arr.pop()
    }
}

const stack = new Stack<boolean>()
```

类型“boolean”不满足约束“Params”。 ts(2344)

查看问题 (⌥F8)　没有可用的快速修复

除了上述的形式，泛型更高级的使用如下：

例如要设计一个函数，这个函数接受两个参数，一个参数为对象，另一个参数为对象上的属性，我们通过这两个参数返回这个属性的值

这时候就设计到泛型的索引类型和约束类型共同实现

8.2.4. 索引类型、约束类型

索引类型 `keyof T` 把传入的对象的属性类型取出生成一个联合类型，这里的泛型 U 被约束在这个联合类型中，如下所示：

TypeScript | 复制代码

```ts
function getValue<T extends object, U extends keyof T>(obj: T, key: U) {
  return obj[key] // ok
}
```

上述为什么需要使用泛型约束，而不是直接定义第一个参数为 `object` 类型，是因为默认情况 `object` 指的是 `{}`，而我们接收的对象是各种各样的，一个泛型来表示传入的对象类型，比如 `T extends object`

使用如下图所示：

```
function getValue<T extends object, U extends keyof T>(obj: T, key: U) {
    return obj[key] // ok
  }

const a = {
    name: 'huihui',
    age: 18
}

getValue(a,)
```

getValue(obj: { name: string; age: number; }, key: "name" | "age"): string | number

8.2.5. 多类型约束

例如如下需要实现两个接口的类型约束：

TypeScript | 复制代码

```typescript
interface FirstInterface {
  doSomething(): number
}

interface SecondInterface {
  doSomethingElse(): string
}
```

可以创建一个接口继承上述两个接口，如下

TypeScript | 复制代码

```typescript
interface ChildInterface extends FirstInterface, SecondInterface {

}
```

正确使用如下：

```typescript
class Demo<T extends ChildInterface> {
  private genericProperty: T

  constructor(genericProperty: T) {
    this.genericProperty = genericProperty
  }
  useT() {
    this.genericProperty.doSomething()
    this.genericProperty.doSomethingElse()
  }
}
```

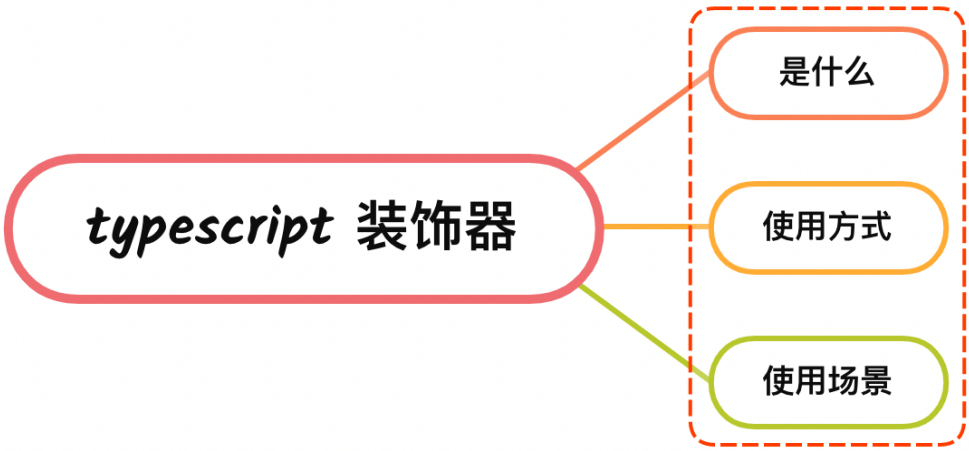
通过泛型约束就可以达到多类型约束的目的

8.3. 应用场景

通过上面初步的了解，后述在编写 `typescript` 的时候，定义函数，接口或者类的时候，不预先定义好具体的类型，而在使用的时候在指定类型的一种特性的时候，这种情况下就可以使用泛型

灵活的使用泛型定义类型，是掌握 `typescript` 必经之路

9. 说说你对 TypeScript 装饰器的理解？应用场景？



9.1. 是什么

装饰器是一种特殊类型的声明，它能够被附加到类声明，方法， 访问符，属性或参数上

是一种在不改变原类和使用继承的情况下，动态地扩展对象功能

同样的，本质也不是什么高大上的结构，就是一个普通的函数，`@expression` 的形式其实是 `Object.defineProperty` 的语法糖

`expression` 求值后必须也是一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入

9.2. 使用方式

由于 `typescript` 是一个实验性特性，若要使用，需要在 `tsconfig.json` 文件启动，如下：

TypeScript | 复制代码

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

`typescript` 装饰器的使用和 `javascript` 基本一致

类的装饰器可以装饰：

- 类
- 方法/属性
- 参数
- 访问器

9.2.1. 类装饰

例如声明一个函数 `addAge` 去给 Class 的属性 `age` 添加年龄.

```typescript
function addAge(constructor: Function) {
  constructor.prototype.age = 18;
}

@addAge
class Person{
  name: string;
  age!: number;
  constructor() {
    this.name = 'huihui';
  }
}

let person = new Person();

console.log(person.age); // 18
```

上述代码，实际等同于以下形式：

```typescript
Person = addAge(function Person() { ... });
```

上述可以看到，当装饰器作为修饰类的时候，会把构造器传递进去。 `constructor.prototype.age` 就是在每一个实例化对象上面添加一个 `age` 属性

9.2.2. 方法/属性装饰

同样，装饰器可以用于修饰类的方法，这时候装饰器函数接收的参数变成了：

- target：对象的原型
- propertyKey：方法的名称
- descriptor：方法的属性描述符

可以看到，这三个属性实际就是 `Object.defineProperty` 的三个参数，如果是类的属性，则没有传递第三个参数

如下例子：

TypeScript | 复制代码

```typescript
// 声明装饰器修饰方法/属性
function method(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  console.log(target);
  console.log("prop " + propertyKey);
  console.log("desc " + JSON.stringify(descriptor) + "\n\n");
  descriptor.writable = false;
};

function property(target: any, propertyKey: string) {
  console.log("target", target)
  console.log("propertyKey", propertyKey)
}

class Person{
 @property
 name: string;
 constructor() {
   this.name = 'huihui';
 }

 @method
 say(){
   return 'instance method';
 }

 @method
 static run(){
   return 'static method';
 }
}

const xmz = new Person();

// 修改实例方法say
xmz.say = function() {
 return 'edit'
}
```

输出如下图所示：

```
target ▸{constructor: f, say: f}                                    index.ts:12
propertyKey name                                                    index.ts:13
▸{constructor: f, say: f}                                           index.ts:5
prop say                                                            index.ts:6
desc {"writable":true,"enumerable":false,"configurable":true}       index.ts:7

class Person {                                                      index.ts:5
    constructor() {
        this.name = 'xiaomuzhu';
    }
    say() {
        return 'instance method';
    }
    static run() {
        return 'static method';
    }
}
prop run                                                            index.ts:6
desc {"writable":true,"enumerable":false,"configurable":true}       index.ts:7

▸Uncaught TypeError: Cannot assign to read only property 'say' of object '#<Person>'   index.ts:37
    at index.ts:37
```

9.2.3. 参数装饰

接收3个参数，分别是：

- target ： 当前对象的原型
- propertyKey ： 参数的名称
- index：参数数组中的位置

TypeScript | 复制代码

```typescript
function logParameter(target: Object, propertyName: string, index: number) {
  console.log(target);
  console.log(propertyName);
  console.log(index);
}

class Employee {
  greet(@logParameter message: string): string {
      return `hello ${message}`;
  }
}
const emp = new Employee();
emp.greet('hello');
```

输入如下图：

```
▼Object ℹ
  ▶constructor: class Employee
  ▶greet: ƒ greet(message)
  ▶[[Prototype]]: Object
greet
0
```

9.2.4. 访问器装饰

使用起来方式与方法装饰一致，如下：

TypeScript | 复制代码

```typescript

function modification(target: Object, propertyKey: string, descriptor: PropertyDescriptor) {
  console.log(target);
  console.log("prop " + propertyKey);
  console.log("desc " + JSON.stringify(descriptor) + "\n\n");
};

class Person{
 _name: string;
 constructor() {
   this._name = 'huihui';
 }

 @modification
 get name() {
   return this._name
 }
}
```

9.2.5. 装饰器工厂

如果想要传递参数，使装饰器变成类似工厂函数，只需要在装饰器函数内部再函数一个函数即可，如下：

TypeScript | 复制代码

```typescript
function addAge(age: number) {
  return function(constructor: Function) {
    constructor.prototype.age = age
  }
}

@addAge(10)
class Person{
  name: string;
  age!: number;
  constructor() {
    this.name = 'huihui';
  }
}

let person = new Person();
```

9.2.6. 执行顺序

当多个装饰器应用于一个声明上，将由上至下依次对装饰器表达式求值，求值的结果会被当作函数，由下至上依次调用，例如如下：

TypeScript | 复制代码

```typescript
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("f(): called");
    }
}

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}

// 输出
f(): evaluated
g(): evaluated
g(): called
f(): called
```

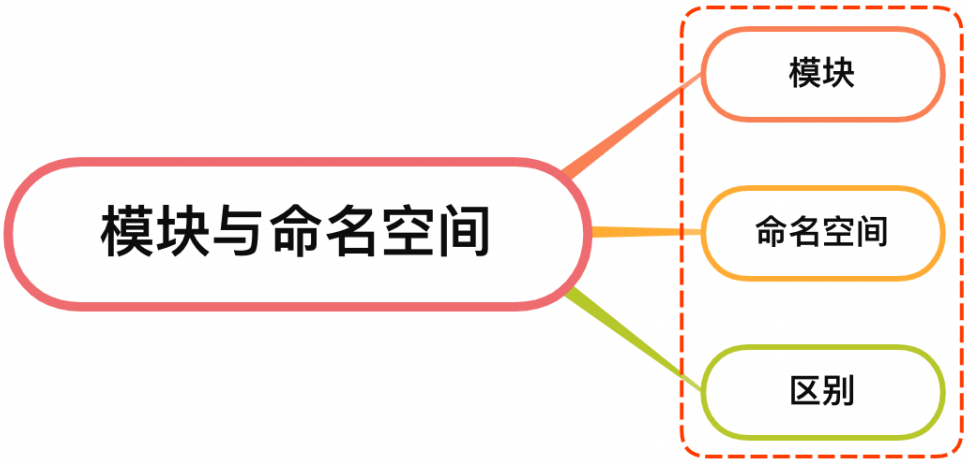
9.3. 应用场景

可以看到，使用装饰器存在两个显著的优点：

- 代码可读性变强了，装饰器命名相当于一个注释
- 在不改变原有代码情况下，对原来功能进行扩展

后面的使用场景中，借助装饰器的特性，除了提高可读性之后，针对已经存在的类，可以通过装饰器的特性，在不改变原有代码情况下，对原来功能进行扩展

10. 说说对 TypeScript 中命名空间与模块的理解？区别？



10.1. 模块

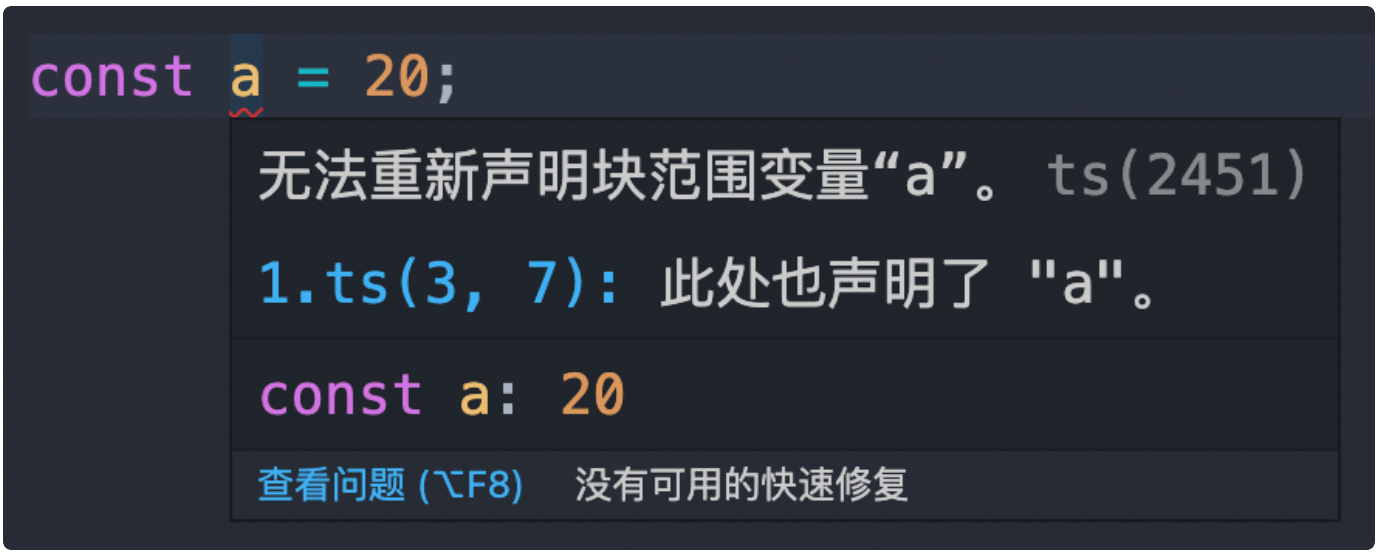
`TypeScript` 与 `ECMAScript` 2015 一样，任何包含顶级 `import` 或者 `export` 的文件都被当成一个模块

相反地，如果一个文件不带有顶级的 `import` 或者 `export` 声明，那么它的内容被视为全局可见的

例如我们在在一个 `TypeScript` 工程下建立一个文件 `1.ts`，声明一个变量 `a`，如下：

```typescript
const a = 1
```

然后在另一个文件同样声明一个变量 `a`，这时候会出现错误信息



提示重复声明 `a` 变量，但是所处的空间是全局的

如果需要解决这个问题，则通过 `import` 或者 `export` 引入模块系统即可，如下：

```typescript
const a = 10;

export default a
```

在 `typescript` 中， `export` 关键字可以导出变量或者类型，用法与 `es6` 模块一致，如下：

```typescript
export const a = 1
export type Person = {
    name: String
}
```

通过 `import` 引入模块，如下：

```typescript
import { a, Person } from './export';
```

10.2. 命名空间

命名空间一个最明确的目的就是解决重名问题

命名空间定义了标识符的可见范围，一个标识符可在多个名字空间中定义，它在不同名字空间中的含义是互不相干的

这样，在一个新的名字空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其他名字空间中

`TypeScript` 中命名空间使用 `namespace` 来定义，语法格式如下：

```typescript
namespace SomeNameSpaceName {
   export interface ISomeInterfaceName {      }
   export class SomeClassName {      }
}
```

以上定义了一个命名空间 `SomeNameSpaceName` ，如果我们需要在外部可以调用 `SomeNameSpaceName` 中的类和接口，则需要在类和接口添加 `export` 关键字

使用方式如下：

```typescript
SomeNameSpaceName.SomeClassName
```

命名空间本质上是一个对象，作用是将一系列相关的全局变量组织到一个对象的属性，如下：

```typescript
namespace Letter {
  export let a = 1;
  export let b = 2;
  export let c = 3;
  // ...
  export let z = 26;
}
```

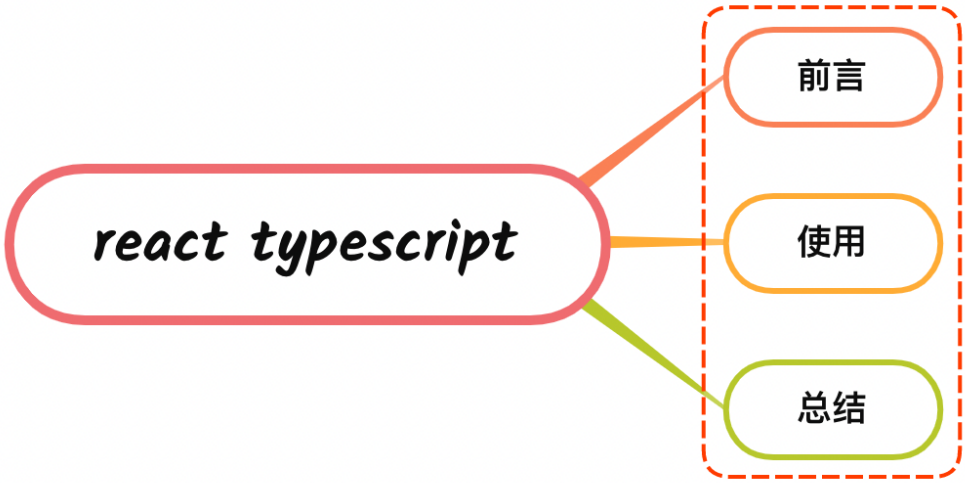
编译成 `js` 如下：

```javascript
var Letter;
(function (Letter) {
    Letter.a = 1;
    Letter.b = 2;
    Letter.c = 3;
    // ...
    Letter.z = 26;
})(Letter || (Letter = {}));
```

10.3. 区别

- 命名空间是位于全局命名空间下的一个普通的带有名字的 JavaScript 对象，使用起来十分容易。但就像其它的全局命名空间污染一样，它很难去识别组件之间的依赖关系，尤其是在大型的应用中
- 像命名空间一样，模块可以包含代码和声明。 不同的是模块可以声明它的依赖
- 在正常的TS项目开发过程中并不建议用命名空间，但通常在通过 d.ts 文件标记 js 库类型的时候使用命名空间，主要作用是给编译器编写代码的时候参考使用

11. 说说如何在 React 项目中应用 TypeScript?



11.1. 前言

单独的使用 `TypeScript` 并不会导致学习成本很高，但是绝大部分前端开发者的项目都是依赖于框架的

例如与 `Vue` 、 `React` 这些框架结合使用的时候，会有一定的门槛

使用 `TypeScript` 编写 `React` 代码，除了需要 `TypeScript` 这个库之外，还需要安装 `@types/react` 、 `@types/react-dom`

```bash
npm i @types/react -s

npm i @types/react-dom -s
```

至于上述使用 `@types` 的库的原因在于，目前非常多的 `JavaScript` 库并没有提供自己关于 `TypeScript` 的声明文件

所以， `ts` 并不知道这些库的类型以及对应导出的内容，这里 `@types` 实际就是社区中的 `DefinitelyTyped` 库，定义了目前市面上绝大多数的 `JavaScript` 库的声明

所以下载相关的 `JavaScript` 对应的 `@types` 声明时，就能够使用使用该库对应的类型定义

11.2. 使用方式

在编写 `React` 项目的时候，最常见的使用的组件就是：

- 无状态组件

- 有状态组件
- 受控组件

11.2.1. 无状态组件

主要作用是用于展示 `UI`，如果使用 `js` 声明，则如下所示：

JSX | 复制代码

```jsx
import * as React from "React";

export const Logo = (props) => {
  const { logo, className, alt } = props;

  return <img src={logo} className={className} alt={alt} />;
};
```

但这时候 `ts` 会出现报错提示，原因在于没有定义 `porps` 类型，这时候就可以使用 `interface` 接口去定义 `porps` 即可，如下：

TSX | 复制代码

```tsx
import * as React from "React";

interface IProps {
  logo?: string;
  className?: string;
  alt?: string;
}

export const Logo = (props: IProps) => {
  const { logo, className, alt } = props;

  return <img src={logo} className={className} alt={alt} />;
};
```

但是我们都知道 `props` 里面存在 `children` 属性，我们不可能每个 `porps` 接口里面定义多一个 `children`，如下：

```typescript
interface IProps {
  logo?: string;
  className?: string;
  alt?: string;
  children?: ReactNode;
}
```

更加规范的写法是使用 `React` 里面定义好的 `FC` 属性，里面已经定义好 `children` 类型，如下：

```tsx
export const Logo: React.FC<IProps> = (props) => {
  const { logo, className, alt } = props;

  return <img src={logo} className={className} alt={alt} />;
};
```

- React.FC 显式地定义了返回类型，其他方式是隐式推导的
- React.FC 对静态属性：displayName、propTypes、defaultProps 提供了类型检查和自动补全
- React.FC 为 children 提供了隐式的类型（ReactElement | null）

11.2.2. 有状态组件

可以是一个类组件且存在 `props` 和 `state` 属性

如果使用 `TypeScript` 声明则如下所示：

```tsx
import * as React from "React";

interface IProps {
  color: string;
  size?: string;
}
interface IState {
  count: number;
}
class App extends React.Component<IProps, IState> {
  public state = {
    count: 1,
  };
  public render() {
    return <div>Hello world</div>;
  }
}
```

上述通过泛型对 `props` 、 `state` 进行类型定义，然后在使用的时候就可以在编译器中获取更好的智能提示

关于 `Component` 泛型类的定义，可以参考下 React 的类型定义文件 `node_modules/@types/React/index.d.ts` ，如下所示：

```typescript
class Component<P, S> {
  readonly props: Readonly<{ children?: ReactNode }> & Readonly<P>;

  state: Readonly<S>;
}
```

从上述可以看到， `state` 属性也定义了可读类型，目的是为了防止直接调用 `this.state` 更新状态

11.2.3. 受控组件

受控组件的特性在于元素的内容通过组件的状态 `state` 进行控制

由于组件内部的事件是合成事件，不等同于原生事件，

例如一个 `input` 组件修改内部的状态，常见的定义的时候如下所示：

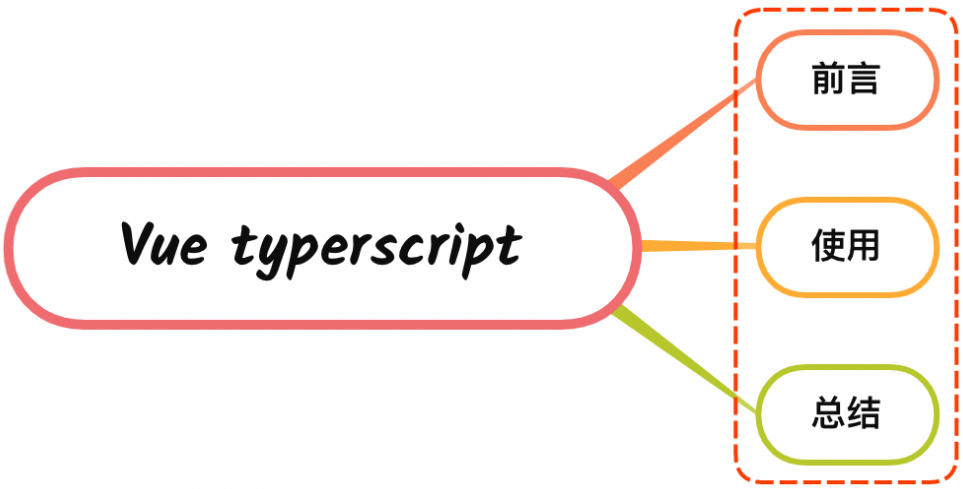
```typescript
private updateValue(e: React.ChangeEvent<HTMLInputElement>) {
    this.setState({ itemText: e.target.value })
}
```

常用 `Event` 事件对象类型：

- ClipboardEvent<T = Element> 剪贴板事件对象
- DragEvent<T = Element> 拖拽事件对象
- ChangeEvent<T = Element> Change 事件对象
- KeyboardEvent<T = Element> 键盘事件对象
- MouseEvent<T = Element> 鼠标事件对象
- TouchEvent<T = Element> 触摸事件对象
- WheelEvent<T = Element> 滚轮事件对象
- AnimationEvent<T = Element> 动画事件对象
- TransitionEvent<T = Element> 过渡事件对象

`T` 接收一个 `DOM` 元素类型

12. 说说如何在Vue项目中应用TypeScript?



12.1. 前言

与link类似

在 `VUE` 项目中应用 `typescript`，我们需要引入一个库 `vue-property-decorator`，

其是基于 `vue-class-component` 库而来，这个库 `vue` 官方推出的一个支持使用 `class` 方式来开发 `vue` 单文件组件的库

主要的功能如下：

- methods 可以直接声明为类的成员方法
- 计算属性可以被声明为类的属性访问器
- 初始化的 data 可以被声明为类属性
- data、render 以及所有的 Vue 生命周期钩子可以直接作为类的成员方法
- 所有其他属性，需要放在装饰器中

12.2. 使用

vue-property-decorator 主要提供了多个装饰器和一个函数:

12.2.1. @Component

`Component` 装饰器它注明了此类为一个 `Vue` 组件，因此即使没有设置选项也不能省略

如果需要定义比如 `name` 、 `components` 、 `filters` 、 `directives` 以及自定义属性，就可以在 `Component` 装饰器中定义，如下：

```vue
import {Component,Vue} from 'vue-property-decorator';
import {componentA,componentB} from '@/components';

 @Component({
    components:{
        componentA,
        componentB,
    },
    directives: {
        focus: {
            // 指令的定义
            inserted: function (el) {
                el.focus()
            }
        }
    }
})
export default class YourCompoent extends Vue{

}
```

12.2.2. computed、data、methods

这里取消了组件的data和methods属性，以往data返回对象中的属性、methods中的方法需要直接定义在Class中，当做类的属性和方法

TypeScript | 复制代码

```typescript
@Component
export default class HelloDecorator extends Vue {
    count: number = 123 // 类属性相当于以前的 data

    add(): number { // 类方法就是以前的方法
        this.count + 1
    }

    // 获取计算属性
    get total(): number {
      return this.count + 1
    }

    // 设置计算属性
    set total(param:number): void {
      this.count = param
    }
}
```

12.2.3. @props

组件接收属性的装饰器，如下使用：

JavaScript | 复制代码

```
import {Component,Vue,Prop} from vue-property-decorator;

@Component
export default class YourComponent extends Vue {
    @Prop(String)
    propA:string;

    @Prop([String,Number])
    propB:string|number;

    @Prop({
     type: String, // type: [String , Number]
     default: 'default value', // 一般为String或Number
      //如果是对象或数组的话。默认值从一个工厂函数中返回
      // defatult: () => {
      //     return ['a','b']
      // }
     required: true,
     validator: (value) => {
        return [
          'InProcess',
          'Settled'
        ].indexOf(value) !== -1
     }
    })
    propC:string;
}
```

12.2.4. @watch

实际就是 `Vue` 中的监听器，如下：

```vue
import { Vue, Component, Watch } from 'vue-property-decorator'

@Component
export default class YourComponent extends Vue {
  @Watch('child')
  onChildChanged(val: string, oldVal: string) {}

  @Watch('person', { immediate: true, deep: true })
  onPersonChanged1(val: Person, oldVal: Person) {}

  @Watch('person')
  onPersonChanged2(val: Person, oldVal: Person) {}
}
```

12.2.5. @emit

`vue-property-decorator` 提供的 `@Emit` 装饰器就是代替 `Vue` 中的事件的触发 `$emit`，如下：

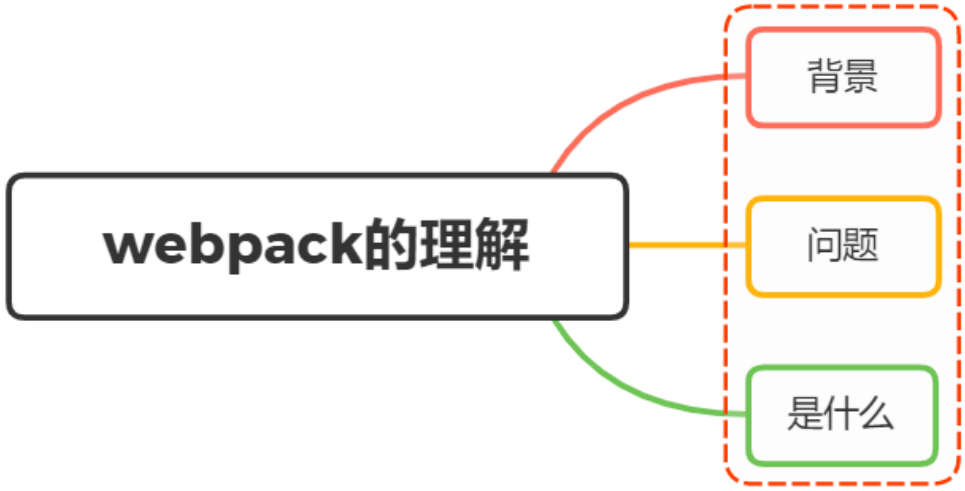
```typescript
import {Vue, Component, Emit} from 'vue-property-decorator';
    @Component({})
    export default class Some extends Vue{
        mounted(){
            this.$on('emit-todo', function(n) {
                console.log(n)
            })
            this.emitTodo('world');
        }
        @Emit()
        emitTodo(n: string){
            console.log('hello');
        }
    }
```

12.3. 总结

可以看到上述 `typescript` 版本的 `vue class` 的语法与平时 `javascript` 版本使用起来还是有很大的不同，多处用到 `class` 与装饰器，但实际上本质是一致的，只有不断编写才会得心应手

Webpack面试真题（10题）

1. 说说你对webpack的理解？解决了什么问题？



1.1. 背景

`Webpack` 最初的目标是实现前端项目的模块化，旨在更高效地管理和维护项目中的每一个资源

1.1.1. 模块化

最早的时候，我们会通过文件划分的形式实现模块化，也就是将每个功能及其相关状态数据各自单独放到不同的 `JS` 文件中

约定每个文件是一个独立的模块，然后再将这些 `js` 文件引入到页面，一个 `script` 标签对应一个模块，然后调用模块化的成员

HTML 复制代码

```
<script src="module-a.js"></script>
<script src="module-b.js"></script>
```

但这种模块弊端十分的明显，模块都是在全局中工作，大量模块成员污染了环境，模块与模块之间并没有依赖关系、维护困难、没有私有空间等问题

项目一旦变大，上述问题会尤其明显

随后，就出现了命名空间方式，规定每个模块只暴露一个全局对象，然后模块的内容都挂载到这个对象中

```javascript
window.moduleA = {
  method1: function () {
    console.log('moduleA#method1')
  }
}
```

这种方式也并没有解决第一种方式的依赖等问题

再后来，我们使用立即执行函数为模块提供私有空间，通过参数的形式作为依赖声明，如下

```javascript
// module-a.js
(function ($) {
  var name = 'module-a'

  function method1 () {
    console.log(name + '#method1')
    $('body').animate({ margin: '200px' })
  }

  window.moduleA = {
    method1: method1
  }
})(jQuery)
```

上述的方式都是早期解决模块的方式，但是仍然存在一些没有解决的问题。例如，我们是用过 `script` 标签在页面引入这些模块的，这些模块的加载并不受代码的控制，时间一久维护起来也十分的麻烦

理想的解决方式是，在页面中引入一个 `JS` 入口文件，其余用到的模块可以通过代码控制，按需加载进来

除了模块加载的问题以外，还需要规定模块化的规范，如今流行的则是 `CommonJS` 、 `ES Modules`

1.2. 问题

从后端渲染的 `JSP` 、 `PHP` ，到前端原生 `JavaScript` ，再到 `jQuery` 开发，再到目前的三大框架 `Vue` 、 `React` 、 `Angular`

开发方式，也从 `javascript` 到后面的 `es5` 、 `es6、7、8、9、10` ，再到 `typescript` ，包括编写 `CSS` 的预处理器 `less` 、 `scss` 等