《前端高手进阶》

朱德龙 前中兴软创主任工程师

— 拉勾教育出品 —

第11讲：为什么说 JavaScript 不适合大型项目？

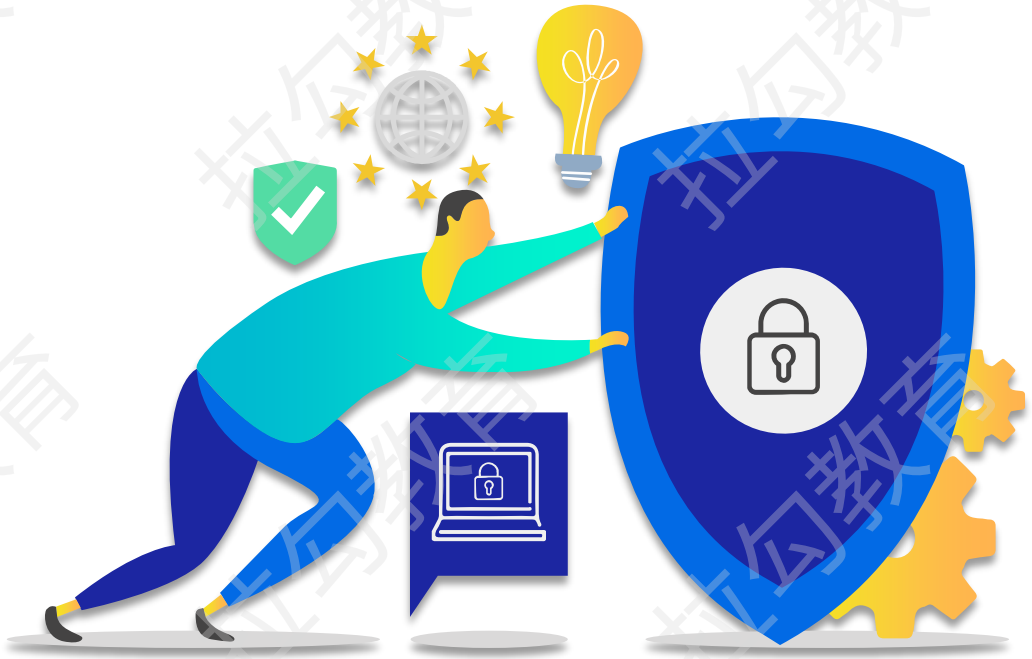
前言

随着前端快速发展，JavaScript 语言的设计缺陷在大型项目中逐渐显

第 10 课时提到的模块问题就是其中之一

但庆幸的是，ES6 模块在原生层面解决了这个问题

不同环境下的兼容性问题也可以**由工具转化代码**来解决



类型问题

1. 类型声明

```
var c = 0
...
function fn() {
  ...
  c = 30;
}
fn();
```

类型问题

2. 动态类型

动态类型是指在运行期间才做数据类型检查的语言

即动态类型语言编程时，不用给任何变量指定数据类型

```
function printId(user) {
 return user.id
}
```

类型问题

3. 弱类型

```
var tmp = []
...
tmp = null
...
// tmp 到底会变成什么?
```

类型问题

TypeScript

它是基于 JavaScript 的语法糖

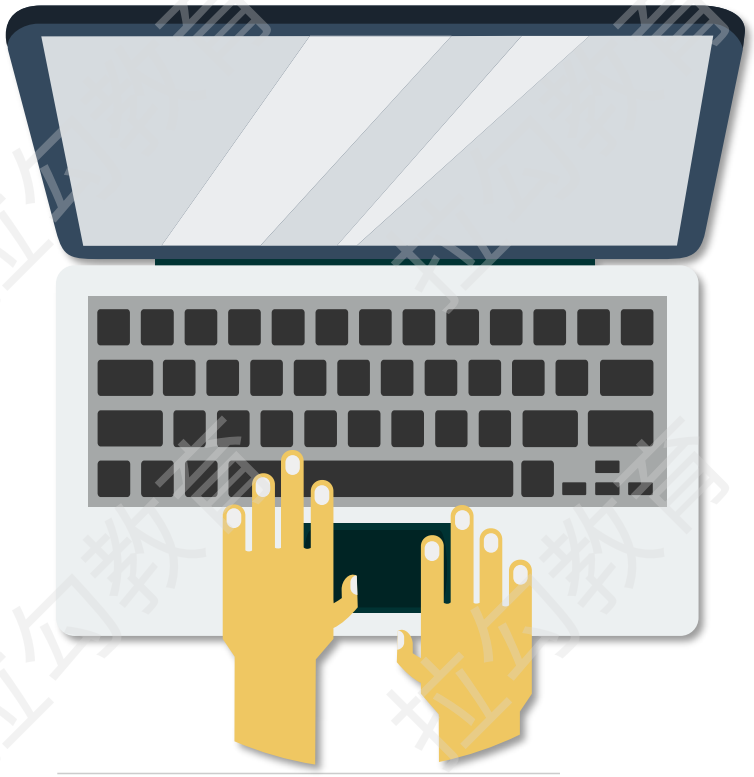
也就是说 TypeScript 代码没有单独的运行环境

需要编译成 JavaScript 代码之后才能运行

从它的名字不难看出，它的核心特性是类型“Type”

具体工作原理就是**在代码编译阶段进行类型检测**

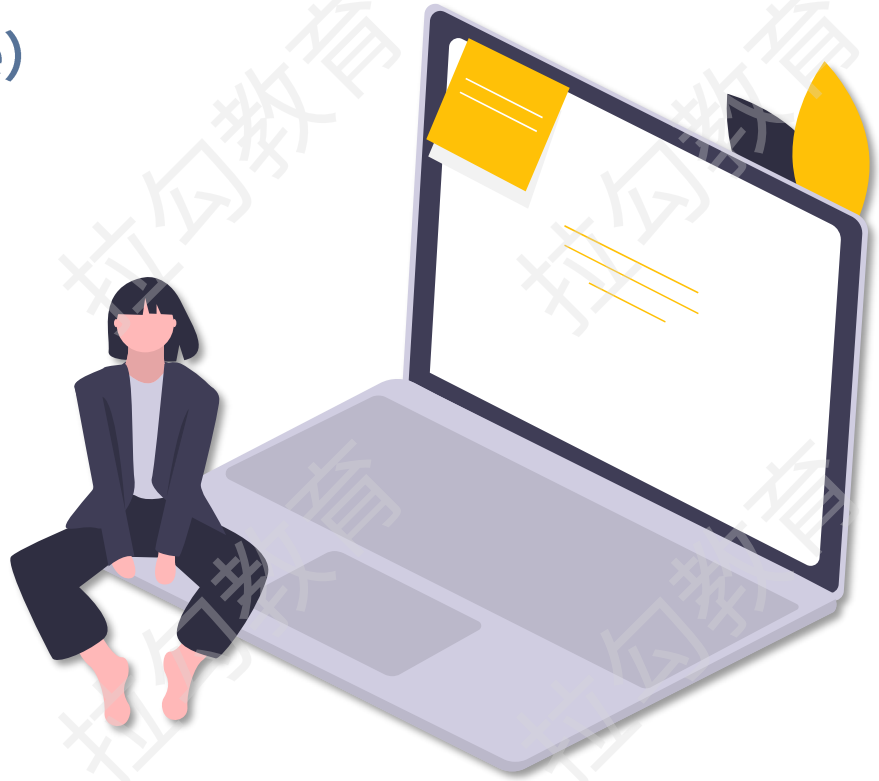
这样就能在代码部署运行之前及时发现问题



类型与接口

TypeScript 让 JavaScript 变成了**静态强类型**、**变量**需要严格声明的语言

为此定义了两个重要概念：**类型（type）**和**接口（interface）**



类型与接口

TypeScript 在 JavaScript 原生类型的基础上进行了**扩展**

但为了和基础类型对象进行区分，采用了小写的形式

类型之间可以互相组合形成**新的类型**

1. 元组

```
let x: [string, number];
```

2. 枚举

- 数字枚举
- 字符串枚举
- 异构枚举

```
enum example {
  No = 0,
  Yes = "YES",
}
```

2. 枚举

```
enum example{
   No = 0,
   Yes = "YES",

console.log(example.No)
// 编译成
var example;
(function (example) {
   example[example["No"] = 0] = "No";
```

2. 枚举

```
  example[example["No"] = 0] = "No";
  example["Yes"] = "YES";
})(example || (example = {}));
console.log(example.No);
////////////
const enum example {
  No = 0,
  Yes = "YES",
}
```

2. 枚举

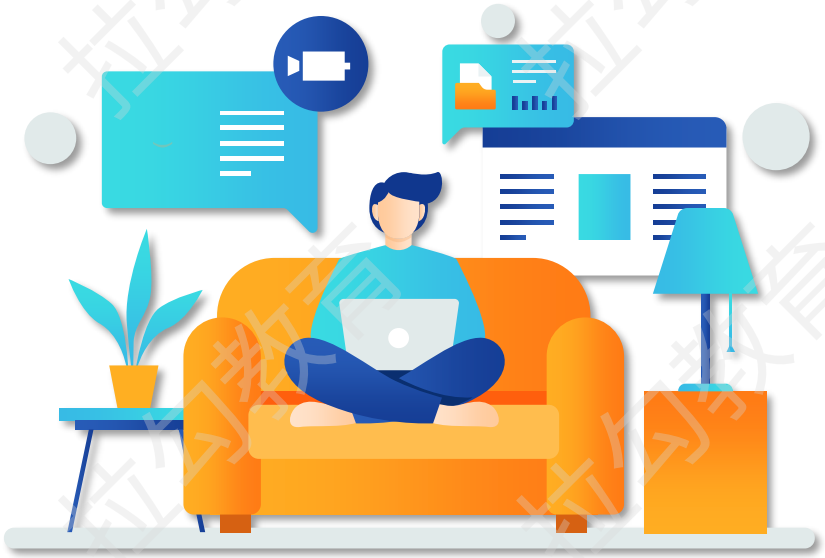
```
console.log(example.No);
////////////
const enum example {
  No = 0,
  Yes = "YES",
}
console.log(example.No)
// 编译成
console.log(0 /* No */);
```

3. any

any 类型代表可以是任何一种类型，所以会跳过类型检查

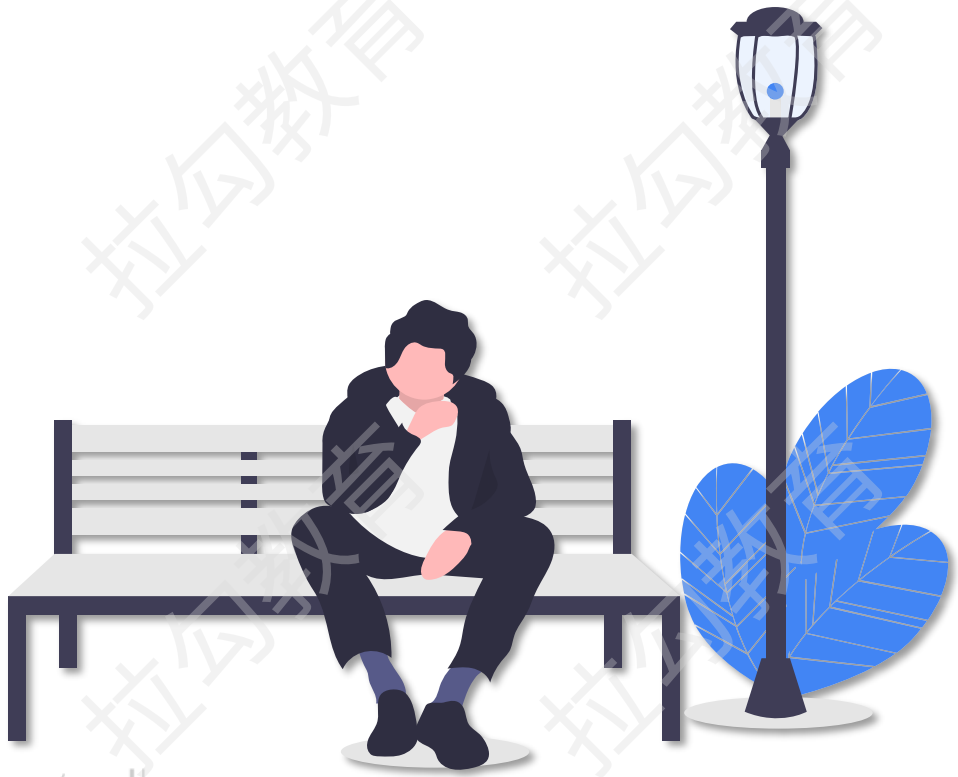
相当于让变量或返回值又变成**弱类型**

因此建议尽量减少 any 类型的使用



4. void

void 表示没有任何类型，常用于描述**无返回值**的函数



5. never

```
let u: 'a'|'b'
//...
if(u === 'a'){
  //...
} else if (u === 'b') {
  //...
}
```

5. never

```
let u: 'a'|'b'|'c'
//...
if(u === 'a') {
  //...
} else if (u === 'b') {
  //...
} else {
  let trmp: never = u // Type '"c"' is not assignable to type 'never'.
}
```

5. never

```
/* 声明 */
interface IA {
  id: string
}
type TA = {
  id: string
}
```

5. never

```
/* 继承 */
interface IA2 extends IA {
  name: string
}
type TA2 = TA & { name: string }
/* 实现 */
class A implements IA {
```

5. never

```
/* 实现 */
class A implements IA {
  id: string = ''
}
class A2 implements TA {
  id: string = ''
}
```

类型抽象

拉勾教育
互联网人实战大学

泛型是对类型的一种抽象

一般用于函数，能让调用者动态地指定部分数据类型

这一点和 any 类型有些像

对于类型的定义具有不确定性，可以指代多种类型

但最大区别在于泛型可以**对函数成员或类成员产生约束关系**

类型抽象

拉勾教育
互联网人实战大学

```
function useState<S>(initialState: S | (() => S)): [S, Dispatch<SetStateAction<S>>];
```

类型抽象

在使用泛型的时候，可以通过尖括号来手动指定泛型变量的类型

这个指定操作称之为**类型断言**

也可以不指定，让 TypeScript 自行推断类型

```
const [id, setId] = useState<string>('');
```

类型组合

交叉

将多个类型合并为一个类型，操作符为 “&”

```
type Admin = Student & Teacher
```

类型组合

联合

表示符合多种类型中的任意一个，不同类型通过操作符“|”连接

```
type A = {
  a: string
}
type B = {
  b: number
}
type AorB = A | B
```

类型组合

联合

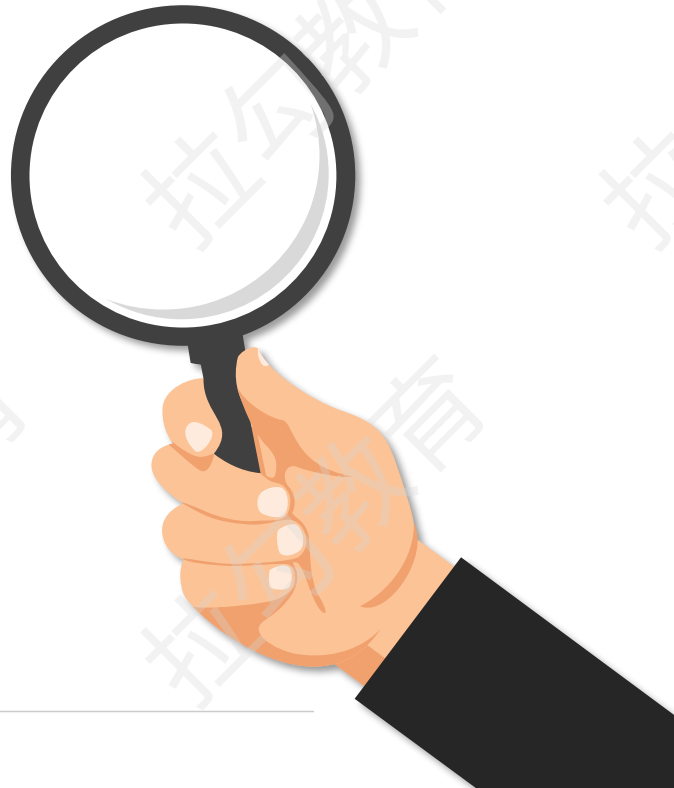
```
let v: AorB
// ...
if ((<A>v).a) {
  //...
} else {
  (<B>v).b
  //...
}
```

类型引用

索引

让 TypeScript 编译器检查出使用了动态属性名的类型

需要通过**索引类型查询**和**索引类型访问**来实现



类型引用

索引

```
function getValue<T, K extends keyof T>(o: T, name: K): T[K] {
    return o[name]; // o[name] is of type T[K]
}
let com = {
    name: 'lagou',
    id: 123
}
let id: number = getValue(com, 'id')
let no = getValue(com, 'no') //?é???Argument of type '"no"' is
not assignable to parameter of type '"id" | "name"'
```

类型引用

映射

从已有类型中创建新的类型

```
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
};
interface task {
  title: string;
  description: string;
  status: string;
}
type simpleTask = Pick<task, 'title' | 'description'>// {title:
string;description: string}
```

实践：编写类型声明


```
const debounce = <T extends Function, U, V extends
any[]>(func: T, wait: number = 0) => {
  let timeout: number | null = null
  let args: V
  function debounced(...arg: V): Promise<U> {
    args = arg
    if(timeout) {
      clearTimeout(timeout)
      timeout = null
    }
```

实践：编写类型声明


```
// 以 Promise 的形式返回函数执行结果
return new Promise((res, rej) => {
  timeout = setTimeout(async () => {
    try {
      const result: U = await func.apply(this, args)
      res(result)
    } catch(e) {
      rej(e)
    }
  }, wait)
```

实践：编写类型声明


```
  }
}

// 允许取消
function cancel() {
  clearTimeout(timeout)
  timeout = null
}

// 允许立即执行
function flush(): U {
  cancel()
```

实践：编写类型声明


```
  }
  // 允许立即执行
  function flush(): U {
    cancel()
    return func.apply(this, args)
  }
  debounced.cancel = cancel
  debounced.flush = flush
  return debounced
}
```

总结

讲述了如何通过 TypeScript 来解决 JavaScript 的类型问题

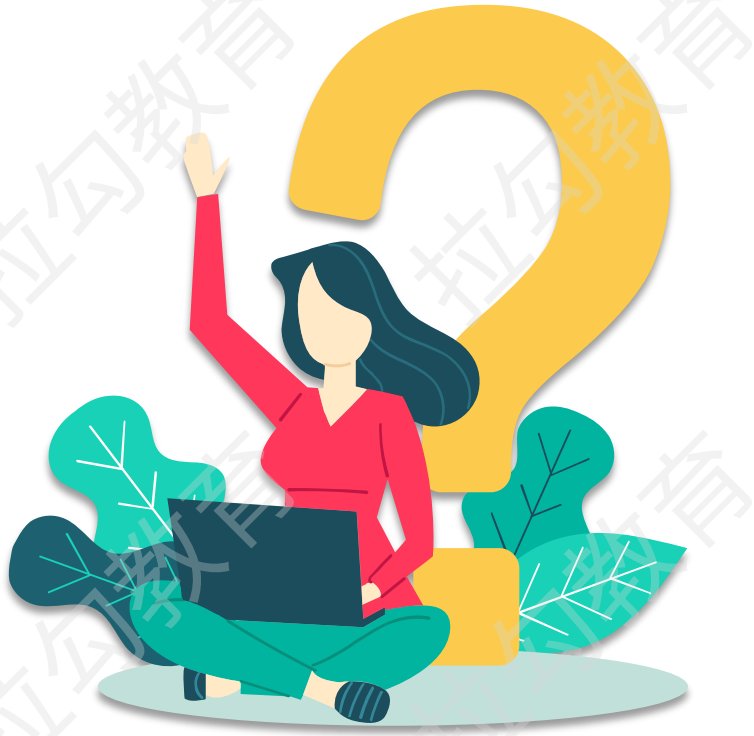
TypeScript 在原有的基础类型上进行了扩展

重点需要掌握**如何通过泛型来对类型进行抽象**

如何通过组合及引用来对已有的类型创建新的类型

总结

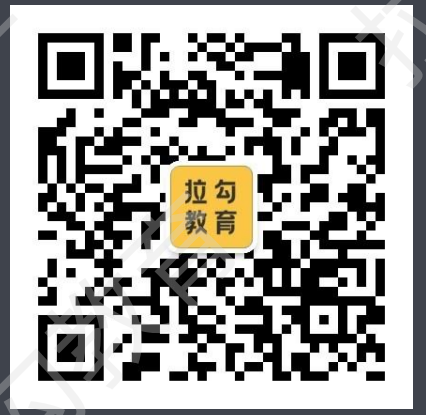
TypeScript 如何对运行时的数据进行校验？



Next：第12讲《浏览器如何执行 JavaScript 代码？》

拉勾教育

—互联网人实战大学—



下载「拉勾教育App」
获取更多内容

L / A / G / O / U