36 | Flow：通过Flow类看JS的类型检查

2022-12-10 石川 来自北京

《JavaScript进阶实战课》

课程介绍 >

讲述：石川

时长 11:34　大小 10.56M

你好，我是石川。

前面我们讲了除了功能性和非功能性测试外，代码的质量检查和风格检查也能帮助我们发现和避免程序中潜在的问题，今天我们再来看看另外一种发现和避免潜在问题的方法——代码类型的检查。说到类型检查，TypeScript 可能是更合适的一门语言，但既然我们这个专栏主讲的是 JavaScript，所以今天我们就通过 Flow ——这个 **JavaScript 的语言扩展**，来学习 JavaScript 中的类型检查。

如果你有 C 或 Java 语言的开发经验，那么对类型注释应该不陌生。我们拿 C 语言最经典的 Hello World 来举个例子，这里的 int 就代表整数，也就是函数的类型。那么，我们知道在 JavaScript 中，是没有类型注释要求的。而通过 Flow，我们既可以做类型注释，也可以对注释和未注释的代码做检查。

```c
#include <stdio.h>

int main() {
  printf("Hello World! \n");
  return 0;
}
```


它的**工作原理**很简单，总结起来就 3 步。

1. 给代码加类型注释；
2. 运行 Flow 工具来分析代码类型和相关的错误报告；
3. 当问题修复后，我们可以通过 Babel 或其它自动化的代码打包流程来去掉代码中的类型注释。

这里，你可能会想，为什么我们在第 3 步会删除注释呢？这是因为 Flow 的语言扩展本身并没有改变 JavaScript 本身的编译或语法，所以它只是我们**代码编写阶段的静态类型检查**。

为什么需要类型

在讲 Flow 前，我们先来熟悉下类型检查的使用场景和目的是什么？类型注释和检查最大的应用场景是较为大型的复杂项目开发。在这种场景下，严格的类型检查可以避免代码执行时**由于类型的出入而引起的潜在问题**。

另外，我们也来看看 Flow 和 TypeScript 有什么共同点和区别。这里，我们可以简单了解下。先说说**相同点**：

- 首先，TypeScript 本身也是 JavaScript 的一个扩展，顾名思义就是“类型脚本语言”；
- 其次，TypeScript 加 TSC 编译器的代码注释和检查流程与 Flow 加 Babel 的模式大同小异；
- 对于简单类型的注释，两者也是相似的；
- 最后，它们的应用场景和目的也都是类似的。

再看看 Flow 和 TypeScript 这两者的**区别**：

- 首先，在对于相对高阶的类型上，两者在语法上有所不同，但要实现转换并不难；

- 其次，TypeScript 是早于 ES6，在 2012 年发布的，所以它虽然名叫 TypeScript，但在当时除了强调类型外，也为了弥补当时 ES5 中没有 class、for/of 循环、模块化或 Promises 等不足而增加了很多功能；
- 除此之外，TypeScript 也增加了很多自己独有的枚举类型（enum）和命名空间（namespace）等关键词，所以它可以说是一个独立的语言了。而 Flow 则更加轻量，它建立在 JavaScript 的基础上，加上了类型检查的功能，而不是自成一派，但改变了 JavaScript 语言本身。

安装和运行

有了一些 Flow 相关的基础知识以后，下面我们正式进入主题，来看看 Flow 的安装和使用。

和我们之前介绍的 JavaScript 之器中的其它工具类似，我们也可以通过 NPM 来安装 Flow。这里，我们同样也使用了 `-g`，这样的选项可以让我们通过命令行来运行相关的程序。

```
npm install -g flow-bin
```

在使用 Flow 做代码类型检查前，我们需要通过下面的命令在项目所在地文件目录下做初始化。通过初始化，会创建一个以 .flowconfig 为后缀的配置文件。虽然我们一般不需要修改这个文件，但是对于 Flow 而言，可以知道我们项目的位置。

```
npm run flow --init
```

在第一次初始化之后，后续的检查可以直接通过 `npm run flow` 来执行。

```
npm run flow
```

Flow 会找到项目所在位置的所有 JavaScript 代码，但只会对头部标注了 `// @flow` 的代码文件做类型检查。这样的好处是**对于已有项目，我们可以对文件逐个来处理，按阶段有计划地加上类型注释。**

前面，我们说过即使对于没有注释的代码，Flow 也可以进行检查，比如下面的例子，因为我们没有把 for 循环中的 `i` 设置为本地变量，就可能造成对全局的污染，所以在没有注释的情况下，也会收到报错。一个简单的解决方案就是在 for 循环中加一段 `for (let i = 0)` 这样的代码。

复制代码

```
// @flow
let i = { x: 0, y: 1 };
for(i = 0; i < 10; i++) {
  console.log(i);
}
i.x = 1;
```

同样，下面的例子在没有注释的情况下也会报错。虽然 Flow 开始不知道 `msg` 参数的类型，但是看到了长度 `length` 这个属性后，就知道它不会是一个数字。但是后面在函数调用过程中传入的实参却是数字，明显会产生问题，所以就会报错。

复制代码

```
// @flow
function msgSize(msg) {
  return msg.length;
}
let message = msgSize(10000);
```

类型注释的使用

上面，我们讲完了在没有注释的情况下的类型检查。下面，我们再来看看类型注释的使用。当你声明一个 JavaScript 变量的时候，可以通过一个冒号和类型名称来增加一个 Flow 的类型注释。比如下面的例子中，我们声明了数字、字符串和布尔的类型。

复制代码

```
// @flow
let num: number = 32;
let msg: string = "Hello world";
let flag: boolean = false;
```

同上面未注释的例子一样，即使在没有注释的情况下，Flow 也可以通过变量声明赋值来判断值的类型。**唯一的区别是在有注释的情况下，Flow 会对比注释和赋值的类型，如果发现两者间有出入，便会报错。**

函数参数和返回值的注释与变量的注释类似，也是通过冒号和类型名称。在下面的例子中，我们把参数的类型注释为字符串，返回值的类型注释为了数字。当我们运行检查的时候，虽然函数本身可以返回结果，但是会报错。这是因为我们期待的返回值是字符串，而数组长度返回的结果却是数字。

复制代码

```
// 普通函数
function msgSize(msg: string): string {
  return msg.length;
}
console.log(msgSize([1,2,3]));
```

不过有一点需要注意的是 JavaScript 和 Flow 中 `null` 的类型是一致的，但是 JavaScript 中的 `undefined` 在 Flow 中是 `void`。而且针对函数没有返回值的情况，我们也可以用 `void` 来注释。

如果你想允许 `null` 或 `undefiend` 作为合法的变量或参数值，只需要在类型前**加一个问号的前缀**。比如在下面的例子中，我们使用了 `?string`，这个时候，虽然它不会对 `null` 参数本身的类型报错，但会报错说 `msg.length` 是不安全的，这是因为 `msg` 可能是 `null` 或 `undefined` 这些没有长度的值。为了解决这个报错，我们可以使用一个判断条件，只有在判断结果是**真值**的情况下，会再返回 `msg.length`。

复制代码

```
// @flow

function msgSize(msg: ?string): number {
  return msg.length;
}
console.log(msgSize(null));

function msgSize(msg: ?string): number {
  return msg ? msg.length : -1;
}
console.log(msgSize(null));
```

复杂数据类型的支持

到目前为止，我们学习了几个原始数据类型：字符串、数字、布尔、null 和 undefined 的检查，并且也学习了 Flow 在变量声明赋值、函数的参数和返回值中的使用。下面我们来看一些 Flow 对其它更加复杂的数据类型检查的支持。

类

首先我们先来看一下类。类的关键词 `class` 不需要额外的注释，但是我们可以对里面的属性和方法做类型注释。比如在下面的例子中，`prop` 属性的类型是数字。方法 `method` 的参数是字符串，返回值我们可以定义为数字。

复制代码

```
// @flow
class MyClass {
  prop: number = 42;
  method(value: string): number { /* ... */ }
}
```

对象和类型别名

Flow 中的对象类型看上去很像是对象字面量，区别是 Flow 的属性值是类型。

复制代码

```
// @flow
var obj1: { foo: boolean } = { foo: true };
```

在对象中，如果一个属性是可选的，我们可以通过下面的问号的方式来代替 `void` 和 `undefined`。如果没有注释为可选，那就默认是必须存在的。如果我们想改成可选，同样需要加一个问号。

复制代码

```
var obj: { foo?: boolean } = {};
```

Flow 对函数中没有标注的额外的属性是不会报错的。如果我们想要 Flow 来严格执行只允许明确声明的属性类型，可以通过增加以下**竖线**的方式声明相关的对象类型。

复制代码

```
// @flow
function method(obj: { foo: string }) {
  // ...
}
method({
  foo: "test", // 通过
  bar: 42      // 通过
});

{| foo: string, bar: number |}
```


对于过长的类型对象参数，我们可以通过**自定义类型名称**的方式将参数类抽象提炼出来。

复制代码

```
// @flow
export type MyObject = {
  x: number,
  y: string,
};

export default function method(val: MyObject) {
  // ...
}
```

我们可以像导出一个模块一样地导出类型。其它的模块可以用导入的方式来引用类型定义。但是，这里需要注意的是，导入类型是 Flow 语言的一个延伸，不是一个实际的 JavaScript 导入指令。类型的导入导出只是被 Flow 作为类型检查来使用的，在最终执行的代码中，会被删除。最后需要注意的是，和创建一个 type 比起来，更简洁的方式是**直接定义一个 MyObject 类，用来作为类型。**

我们知道在 JavaScript 中，对象有时被用来当做字典或字符串到值的映射。属性的名称是后知的，没法声明成一个 Flow 类型。但是，我们还是可以通过 Flow 来描述数据结构。假设你有一个对象，它的属性是城市的名称，值是城市的位置。我们可以将数据类型通过下面的方式声明。

复制代码

```
// @flow
var cityLocations : {[string]: {long:number, lat:number}} = {
    "上海": { long: 31.22222, lat: 121.45806 }
};
```

```
export default cityLocations;
```


数组

数组中的同类元素类型可以在角括号中说明。一个有着固定长度和不同类型元素的数组，叫做**元组（tuple）**。在元组中，元素的类型在用逗号相隔的方括号中声明。

我们可以通过解构（destructuring）赋值的方式，加上 Flow 的类型别名功能来使用元组。

如果我们希望函数能够接受一个任意长度的数组作为参数时就不能使用元组了，这时我们需要用的是 `Array<mixed>`。`mixed` 表示数组的元素可以是任意类型。

复制代码

```
// @flow

function average(data: Array<number>) {
 // ...
}

let tuple: [number, boolean, string] = [1, true, "three"];
let num  : number  = tuple[0]; // 通过
let bool : boolean = tuple[1]; // 通过
let str  : string  = tuple[2]; // 通过

function size(s: Array<mixed>): number {
    return s.length;
}
console.log(size([1,true,"three"]));
```

如果我们的函数对数组中的元素进行检索和使用，Flow 检查会使用类型检查或其他测试来确定元素的类型。如果你愿意放弃类型检查，你也可以使用 any 而不是 mixed，它允许你对数组的值做任何处理，而**不需要确保这些值是期望的类型**。

函数

我们已经了解了如何通过添加类型注释来指定函数的参数类型和返回值的类型。但是，在高阶函数中，当函数的参数之一本身是函数时，我们也需要能够指定该函数参数的类型。要用 Flow 表示函数的类型，需要写出用逗号分隔、再用括号括起来的每个参数的类型，然后用箭头表示，最后键入函数的返回类型。

下面是一个期望传递回调函数的示例函数。这里注意下，我们是如何为回调函数的类型定义类型别名的。

复制代码

```
// @flow
type FetchTextCallback = (?Error, ?number, ?string) => void;
function fetchText(url: string, callback: FetchTextCallback) {
  // ...
}
```

总结

虽然使用类型检查需要很多额外的工作量，当你开始使用 Flow 的时候，也可能会扫出来大量的问题，但这是很正常的。一旦你掌握了类型规范，就会发现它可以避免很多的潜在问题，比如函数中的输入和输出值类型与期待的参数或结果不一致。

另外，除了我们介绍的数字、字符串、函数、对象和数组等这几种核心的数据类型，类型检查还有很多用法，你也可以通过Flow 的官网了解更多。

思考题

在类型检查中，有两种思想，一种是可靠性（soundness），一种是完整性（completeness）。可靠性是检查任何可能会在运行时发生的问题，完整性是检查一定会在运行时发生的问题。第一种思想是“宁可误杀，也不放过”；而后者的问题是有时可能会让问题“逃脱”。那么你知道 Flow 或 TypeScript 遵循的是哪种思想吗？你觉得哪种思想更适合实现？

欢迎在留言区分享你的看法、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。我们下节课再见！

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。