ASTON

Интенсив-курс по React JS

astondevs.ru

Занятие 6. Flow & Typescript

1. Плюсы и минусы
2. Flow
3. Typescript

Плюсы и минусы

Плюсы:

1. Удобная отладка(легче отловить и исправить ошибки)
2. Сложнее допустить ошибку
3. Более понятный код

Минусы:

1. Больше кода
2. Больше времени на написание кода
3. Порог вхождения

Flow vs Typescript

Typescript имеет компилятор и компилирует .ts файлы в .js, в то время как flow скорее как умный линтер.

Чтобы он начал работать в начале файла надо написать

// @flow

Typescript разработан Microsoft, flow разработан Facebook.

Typescript более популярен, соответственно имеет большее комьюнити и быстрее решаются какие-то проблемы и больше уже типизированных библиотек. Если для библиотеки не написаны типы, то при импорте вам придется типизировать и ее. Подключение к проекту - установить через npm i или yarn add и затем создать файл конфига.

Далее рассмотрим основные возможности и примеры описания типов. Тк базовые возможности не отличаются фактически, то дальше не буду разделять flow и Typescript в примерах.

Typescript

Объявляем тип через « : ». Если присваиваем значение другого типа - ошибка.

```
let a: number = 5;
a = 'aaaaa';  //error
```

Поддерживаются базовые типы js. В том числе и составные.

```
const myObj: (a: number) = {
  a: 10
}
myObj.a = 'aaaaa';  //error

const myArr: number[] = [1, 2, 3];
myArr.push('aaaaa'); //error
```

Также задаем типы параметров функции и возвращаемых значений.

```
const myFunc = (a: number, str: string): string => {
  return a + str;
}
```

Typescript

Есть возможность задавать необязательные параметры, а также использовать rest параметры. Если функция ничего не возвращает, То пишем void.

```
const myFunc = (str: string, bool?: boolean, ...nums: number[]): void => {}
```

Также есть возможность написать тип как any, но это скорее даже анти-паттерн, так как может повлечь “потерю типов” .
Также есть возможность сделать oneOf type.

```
let a: string | number = 5;
a = '5';  //no error
```

Typescript

Type, Interface.

У нас есть возможность создавать свои типы. Это способствует удобству при чтение кода.

Сразу понятно для чего создан тип ApiResponse и понятно для чего его использовать.

После чего в саге можем делать запрос с типом:

```
type Animal {
  name: string
}

type Bear = Animal & {
  honey: boolean
}

const bear = getBear()
bear.name
bear.honey
```

```
interface Point {
  x: number;
  y: number;
}

function printCoords(pt: Point) {
  console.log(`Значение координаты 'x': ${pt.x}`);
  console.log(`Значение координаты 'y': ${pt.y}`);
}

printCoords({ x: 3, y: 7 });
```

Typescript

Таблица сравнения типов и интерфейсов

Аспект	Тип	Интерфейс
Может описывать функции	Да	Да
Может описывать конструкторы	Да	Да
Может описывать кортежи	Да	Да
Может расширяться с помощью интерфейсов	Иногда	Да
Может расширяться с помощью классов	Нет	Да
Может быть реализован с помощью классов	Иногда	Да
Может пересекаться (intersect) с другими типами/интерфейсами	Да	Иногда
Может объединяться с другими типами/интерфейсами	Да	Нет
Может использоваться для создания связанных (mapped) типов	Да	Нет
Может связываться с помощью связанных типов	Да	Да
Имеет расширенное представление в сообщениях об ошибках	Да	Нет
Может быть дополненным (augmented)	Нет	Да
Может быть рекурсивным	Иногда	Нет

Typescript

Можем типизировать как функциональные, так и классовые компоненты. Можем даже методы жизненного цикла типизировать.

```
type MyProps = {
  title: string
}
type MyState = {
  name: string
}
class MyClassComponent extends Component<MyProps, MyState> {
  this.state = {name: ''};

  componentDidMount(): void {
  }

  shouldComponentUpdate(nextProps: MyProps, nextState: MyState): boolean {
    return true;
  }
```

Typescript

Примитивы: string, number, boolean, Array

- **string** представляет строковые значения, например, 'Hello World'
- **number** предназначен для чисел, например, 42. JS не различает целые числа и числа с плавающей точкой (или запятой), поэтому не существует таких типов, как int или float - только number
- **boolean** - предназначен для двух значений: true и false
- **Array** - Для определения типа массива [1, 2, 3] можно использовать синтаксис number[]; такой синтаксис подходит для любого типа (например, string[] - это массив строк и т.д.). Также можно встретить Array<number>, что означает то же самое.

```
let tokenKey = “Hello world”
let userId = 1234 .
```

```
let loteryNumbers: number[]
loteryNumbers.push(45)

let luckyNumbers = [1, 2, 3, 4]

luckyNumbers.push("hello world")
```

Typescript

Функции

В JS функции, в основном, используются для работы с данными. TS позволяет определять типы как для входных (input), так и для выходных (output) значений функции.

```
function greet(name: string) {
  console.log(`Hello, ${name.toUpperCase()}!`);
}
```

Также можно аннотировать тип возвращаемого функцией значения:

```
function getFavouriteNumber(): number {
  return 26;
}
```

Typescript

Generics.

Также у нас есть возможность делать как бы переменные в типах. Это похоже на шаблоны из некоторых других ЯП.

Это помогает делать более универсальные типы.

```
const myFunc = <T>(arg: T, str: string): string => {
  return str + arg;
}

myFunc<number>(18, 'age: ');        //age: 18
myFunc<boolean>(true, 'success: '); //success: true
```

Это могут быть функции:

Typescript

Типы объекта

Объектный тип - это любое значение со свойствами. Для его определения мы просто перечисляем все свойства объекта и их типы.

```
function printCoords(pt: { x: number; y: number })
  console.log(`Значение координаты 'x': ${pt.x}`);
  console.log(`Значение координаты 'y': ${pt.y}`);
}

printCoords({ x: 3, y: 7 });
```

Опциональные свойства

Для определения свойства в качестве опционального используется символ ? после названия свойства:

```
function printName(obj: { first: string; last?: string }) {
  // ...
}
// Обе функции скомпилируются без ошибок
printName({ first: "John" });
printName({ first: "Jane", last: "Air" });
```

Typescript

Объединения (unions)

Объединение - это тип, сформированный из 2 и более типов, представляющий значение, которое может иметь один из этих типов. Типы, входящие в объединение, называются членами (members) объединения.

```
function printId(id: number | string) {
  console.log(`Ваш ID: ${id}`);
}

// OK
printId(101);
// OK
printId("202");
// Ошибка
printId({ myID: 22342 });
// Argument of type '{ myID: number }' is not assignable to parameter of type
```

Typescript

Утверждение типа (type assertion)

В некоторых случаях мы знаем о типе значения больше, чем TS.

Например, когда мы используем document.getElementById, TS знает лишь то, что данный метод возвращает какой-то HTMLElement, но мы знаем, например, что будет возвращен HTMLCanvasElement.

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

```
const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas");
```

Typescript

Защитник типа typeof

Оператор typeof возвращает одну из следующих строк:

- "string"
- "number"
- "bigint"
- "boolean"
- "symbol"
- "undefined"
- "object"
- "function"

```
function printAll(strs: string | string[] | null) {
  if (typeof strs === "object") {
    for (const s of strs) {
      // Object is possibly 'null'.
      // Потенциальным значением обьекта является 'null'
      console.log(s);
    }
  } else if (typeof strs === "string") {
    console.log(strs);
  } else {
    // ...
  }
}
```

Typescript

Сужение типов с помощью оператора in

В JS существует оператор для определения наличия указанного свойства в объекте - оператор in. TS позволяет использовать данный оператор для сужения потенциальных типов.

Например, в выражении 'value' in x, где 'value' - строка, а x - объединение, истинная ветка сужает типы x к типам, которые имеют опциональное или обязательное свойство value, а ложная ветка сужает типы к типам, которые имеют опциональное или не имеют названного свойства:

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    return animal.swim();
  }

  return animal.fly();
}
```

Typescript

Исключающие объединения (discriminated unions)

Предположим, что мы пытаемся закодировать фигуры, такие как круги и квадраты. Круги "следят" за радиусом, а квадраты - за длиной стороны.

```
interface Shape {
  kind: "circle" | "square";
  radius?: number;
  sideLength?: number;
}
```

Тип never

Для представления состояния, которого не должно существовать, в TS используется тип never.

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