TypeScript 语言

课程概述

1. 解决JavaScript自由类型系统的问题，来大大提高代码的可靠程度
2. 课程概要
    a. 强类型与弱类型
    b. 静态类型与动态类型
    c. JS自由类型系统的问题
    d. Flow 静态类型检查方案
    e. TypeScript 语言规范与基本应用

强类型与弱类型（类型安全）

强类型：

1. 实参的类型必须与形参的类型一致，有更强的类型越苏
2. 不允许有隐式类型转换

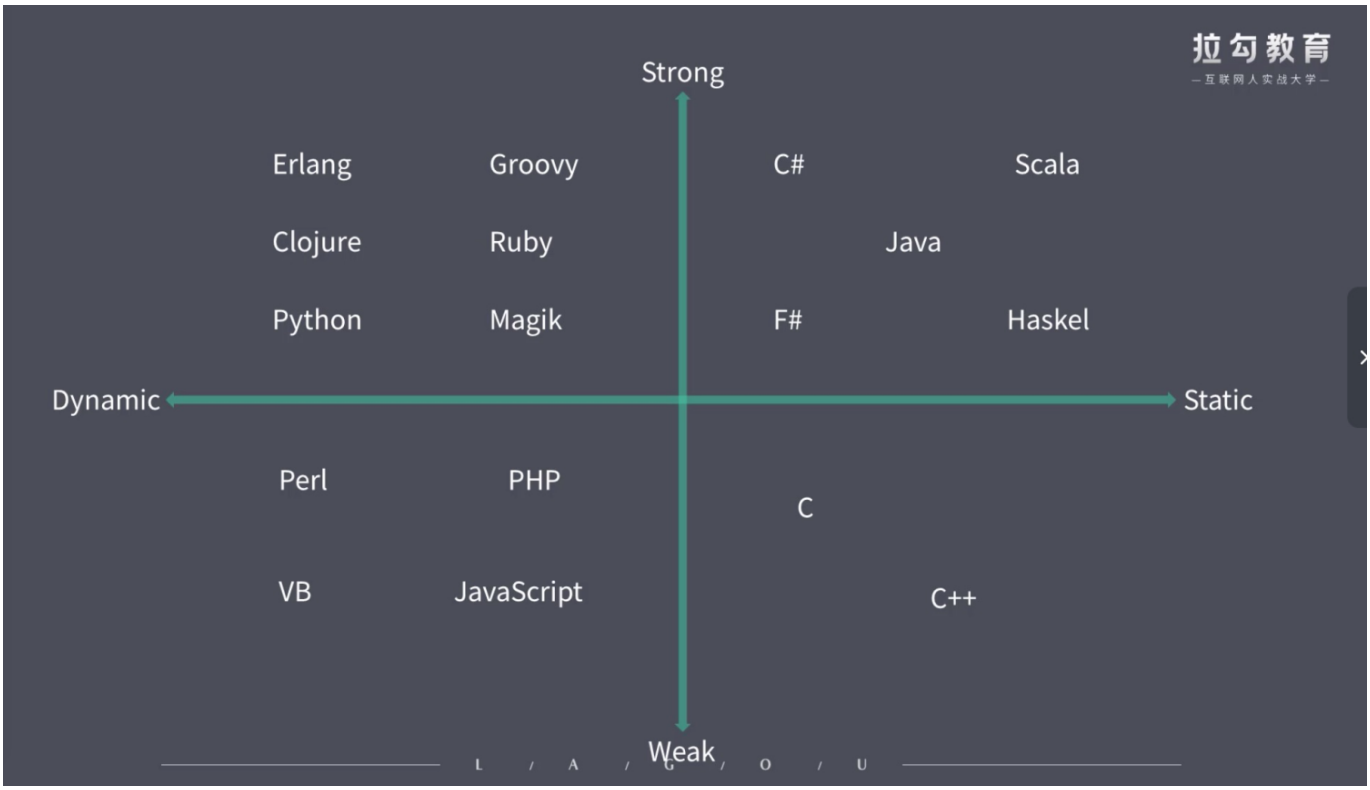
弱类型：

1. 实参的类型不需要与形参的类型一致
2. 可以有任意的隐式类型转换

静态类型与动态类型（类型检查）

静态类型：变量在声明时，就已经确定了类型，后期不允许修改

动态类型：只有当运行时，才能明确类型；变量是没有类型的，变量存放的值是有类型的



JavaScript 类型系统特征

> 弱类型 且 动态类型

1. JavaScript没有编译环节
2. 早期的JavaScript是一种脚本语言，一次只有几百行，加上了类型限制完全没有必要
3. 大规模应用下，这种『优势』就变成了短板

弱类型的问题

1. 执行时才能确定变量类型，如果undefined当做了对象、函数来使用，就会报错
2. 相加函数中，被传入了一个字符串、一个数字，结果就出错
3. 对象只接受字符串作为键。当然，Symbol可以优化它
4. 这种就属于“君子约定”

```
const obj = {};

setTimeout(() => {
  obj.foo(); // TypeError: obj.foo is not a function
}, 10000);
```

```
6
7 // ========================================
8
9 function sum (a, b) {
10   return a + b;
11 }
12
13 console.log(sum(100, 100)); // 200
14 console.log(sum(100, '100')); // 100100
15
16 // ========================================
17
18 const obj = {};
19
20 obj[true] = 100;
21 console.log(obj['true']); // 100
```

强类型的优势

1. 错误更早暴露：可以在代码运行之前，就发现问题，避免运行后在debugger找错而浪费开发时间
2. 代码更智能，编码更准确：由于不确定变量类型，编辑器无法准确给予语法提示
3. 重构更牢靠：可以给予提示，甚至自动帮助我们修改过来
4. 减少不必要的类型判断：

Flow 概述

> JavaScript的类型检查器，使用类型注解

Flow 快速上手

```
1 # 安装flow工具
2 yarn add flow-bin -D
3 # 初始化flow配置
4 yarn flow init
5 # 运行flow服务
6 yarn flow
```

```
1 // @flow
2
```

```
function sum (a: number, b: number) {
  return a + b;
}

sum(100, 100);

sum( '100', '100');
```

Flow 编译移除注解

使用工具在开发完成后，帮助我们移除注解

1. 使用flow-remove-types

```
yarn add flow-remove-types -D

yarn flow-remove-types 源目录 -d 目标目录
```

2. 使用babel, 如下第一行安装过后，配置.babelrc文件

```
yarn add @babel/core @babel/cli @babel/preset-flow

yarn babel 源目录 -d 目标目录
```

```
{
    "presets": ["@babel/preset-flow"]
}
```

Flow 开发工具插件

1. VS code中的Flow Language Suport
2. 其他编辑器的插件可以查看：https://flow.org/en/docs/editors

Flow 类型推断

> type inference

根据变量执行的运算，来推断变量的类型

```
// @flow
```

```
2
3 function square(n) {
4   return n * n; // n 报错
5 }
6
7 square('100');
```

Flow 类型注解

> type annotations

```
1 // @flow
2 function square(n: number) {
3   return n * n;
4 }
5
6 let num: number = '100';
7
8 function foo(): number {
9   return 100;
10 }
```

Flow 原始类型

> Primitive Types

```
1 // @flow
2 const a: string = 'foobar';
3 const b: number = Infinity; // NaN // 100
4 const c: boolean = false; // true
5 const d: null = null;
6 const e: void = undefined;
7 const f: symbol = Symbol();
```

Flow 数组类型

```
1 // @flow
```

```
const arrr1: Array<number> = [1, 2, 3, 4]; // 全部由数字组成的数组
const arr2: number[] = [1, 2, 3]; // 全部由数字组成的数组
// 元组
const foo: [string, number] = ['a', 2]; // 包含两位元素的数组，第一位字符串、第二位数字
```

Flow 对象类型

```
// @flow

const obj1: { foo: string, bar: number } = {foo: '1', bar: 2};
const obj2: { foo?: string, bar: number } = {bar: 2};
const obj3: { [string]: string } = {};

obj3.key1 = 'value1';
obj3.key2 = '100';
```

Flow 函数类型

```
// @flow

function foo (callback: (string, number) => void) {
  callback('string', 100);
}

foo(function (str, num) {
  // str => string
  // num => number
  // 不能有返回值
})
```

Flow 特殊类型

```
// @flow

```

```
// 字面量类型
const a: 'foo' = 'foo'; // 只能存放'foo'
const type: 'success' | 'warning' | 'danger' = 'success'; // 三者都可以

// 联合类型
const b: string | number = '100'; // 100
type StringOrNumber = string | number;
const c: StringOrNumber = '1';

// maybe 类型
const gender: ?number = null; // number | null | void
```

Flow Mixed 与 Any

它们都是表示接受任意类型，Mixed是强类型，Any是弱类型

```
// @flow

// mixed类型，表示所有类型都可以，它是强类型的
function passMixed (value: mixed) {
  return value * value; // 语法报错；可以用if判断分别作处理
}

passMixed(null);
passMixed(100);
passMixed("100");
passMixed(undefined);
passMixed({});
passMixed([]);

// any类型，也可以接受所有类型，它是弱类型的
function passAny(value: any) {
  return value * value; // 语法不报错
}

passAny('string'); // 实际运行时会报错
```

Flow 类型小结

官方文档：https://flow.org/en/docs/types
一个更优秀的第三方手册：https://www.saltycrane.com/cheat-sheets/flow-type/latest/

Flow 运行环境 API

运行环境的API，以及内置对象的类型声明

- https://github.com/facebook/flow/blob/master/lib/core.js
- https://github.com/facebook/flow/blob/master/lib/dom.js
- https://github.com/facebook/flow/blob/master/lib/bom.js
- https://github.com/facebook/flow/blob/master/lib/cssom.js
- https://github.com/facebook/flow/blob/master/lib/node.js

TypeScript 概述

> JavaScript的超集: JS、类型系统、ES6+，经过编译后会形成JavaScript

1. 即使不用TS的类型系统等功能，TypeScript也能帮助我们更好的使用ECMAScript的新特性，此时不用babel都可以，TS最低能编译到ES3
2. 功能更为强大，生态也更健全、更完善
3. TypeScript --- 前端领域中的第二语言
4. 缺点一：语言本身多了很多概念，比如：接口、泛型、枚举；缺点二：项目初期，TypeScript会增加一些成本
5. TypeScript属于「渐进式」，你不用它，单纯写JS语法也是可以的

TypeScript 快速上手

1. 先安装typescript --- yarn add typescript -D
2. 手动编译.ts，-- yarn tsc xxx.ts

```
const hello = (name: string) => {
  console.log(`Hello, ${name}`);
}

hello('TYpeScript');
```

TypeScript 配置文件

1. yarn tsc --init 来初始化配置信息，会生成一个tsconfig.json文件

2. 建议把sourcemap, outDir, rootDir 都打开
3. 然后，就可以直接使用yarn tsc 来编译.ts啦
4. "strictNullChecks": true 可以用来检查变量是否为空

TypeScript 原始类型

string、number、boolean、null、undefined、symbol

```
const a: string = 'foobar';
const b: number = 100; // NaN Infinity
const c: boolean = true; // false
const d: null = null;
const e: void = undefined;
const f: undefined = undefined;

// 报错的原因：tsconfig.json中"target": "es5"的配置不能识别到
// ES6中的Symbol、Promise等
const g: symbol = Symbol(); // 会报错
```

TypeScript 标准库声明

解决上面的Symbol报错的方法是，修改tsconfig.json中的的lib，修改为["ES2015", "DOM"]

TypeScript 中文错误消息

1. VSCODE中，设置中搜索typescript locale， 将其设置为zh-CN
2. 命令行中，yarn tsc --locale zh-CN 可以将错误消息修改为中文

TypeScript 作用域问题

```
// 使用ES6的模块化，添加如下代码

export {}
```

TypeScript Object 类型

1. 可以赋值对象、数组、function等，也就是除了原始类型以外的其他值

```
export {}

const foo: object = function() {};
```

```
4
5 const obj: { foo: number, bar: string } = {
6   foo: 123,
7   bar: 'string',
8 }
```

TypeScript 数组类型

```
1 const arr1: Array<number> = [1, 2, 3, 4];
2 const arr2: number[] = [1, 2, 3, 4];
3
4 function sum (...args: number[]) {
5   return args.reduce((prev, next) => prev + next, 0);
6 }
```

TypeScript 元组类型

1. 明确数量和类型的类型

```
1 const tuple: [number, string] = [27, 'wubo'];
2 const [age, name] = tuple;
```

TypeScript 枚举类型

1. 用某几个值代表某个对象
2. 枚举内的值，如果是数值的话，从上往下自增，如果第一个属性没有设置数字值则默认为0

```
1 enum PostStatus {
2   Draft,
3   Unpublished,
4   Published
5 }
6
7 const post = {
8   title: 'Hello TypeScript',
9   content: 'TypeScript is a typed superset of JavaScript.',
10  status: PostStatus.Draft
```

```
}

PostStatus[0]; // Draft
```

TypeScript 函数类型

1. 形参数量与实参数量一致

```
export {}
// 参数b可以不传
// 参数a默认值100
function func1 (a: number = 100, b?: number): string {
  return 'func1';
}
func1(10);

const func2: (a: number, b: number) => string = function (a: number, b: number): string {

  return 'func2';
}
```

TypeScript 任意类型

1. 不会进行语法检查，所以尽量不要用它

```
export {}

function stringify (value: any) {
  return JSON.stringify(value);
}

stringify('string');
stringify(100);
stringify(true);

let foo: any = 'string';
foo = 100;
foo.bar();
```

TypeScript 隐式类型推断

1. 根据变量的使用情况，来推断变量类型
2. 但是，我们更建议给变量设置明确的类型

```
let age = 18;

age = 'string'; // 报错, 因为ts已经把age推断为了number类型

let foo; // 被推断为any类型
```

TypeScript 类型断言

1. 变量 as 类型;

```
export {}

// 假定这个nums来自一个明确的接口
const nums = [110, 120, 119, 112];

// 我们已经明确了res一定是数字，但是ts会认为res为 number | undefined
const res = nums.find(i => i > 0);

// 所以直接把res当做数字进行计算时，就会报错
const square = res * res; // 报错

// 解决方式
const num1 = res as number;
const square1 = num1 * num1;

const num2 = <number>res; // JSX 下不能使用
```

TypeScript 接口

1. 是一种约定，比如约定一个对象中有哪些成员，以及它们的类型
2. 实现了接口的变量，必须包含接口中规定的成员

```ts
export {}

interface Post {
  title: string;
  content: string;
}

function printPost (post: Post) {
  console.log(post.title, post.content);
}

printPost({
  title: 'Hello TypeScript',
  content: 'TypeScript',
})

const post: Post = { // 报错
  title: '123'
}
```

TypeScript 接口补充

1. 接口中还有可选成员、只读成员

```ts
export {}

interface Post {
  title: string
  content: string
  subtitle?: string // 可选成员
  readonly summary: string // 只读成员
}

const hello: Post = {
  title: 'Hello TypeScript',
  content: 'A JavaScript superset',
  summary: 'A javascript',
};

```

```
// hello.summary = 'other'; //报错

// ---------------------------

interface Cache { // 成员的值必须是对象
  [prop: string]: string
}

const cache: Cache = {};
cache.foo = 'foo';
cache.bar = 'bar';
cache.num = '100';
// cache.num2 = 100; // 报错
```

TypeScript 类的基本使用

1. 描述异类具体事务的抽象特征
2. ts中，属性必须先进行声明，而且一定要赋值

```
export {}

// 属性必须进行赋值
class Person {
  name: string;
  age: number = 27;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  sayHi (msg: string): string {
    return `I am ${this.name}, ${this.age}`;
  }
}
```

TypeScript 类的访问修饰符

1. private： 私有；只能在此类内访问，且不可被继承

2. public：默认；随意访问
3. protected：受保护的，只允许在此类和子类中访问，可以被继承

```ts
export {}

// 属性必须进行赋值
class Person {
  name: string;
  private age: number = 27;
  protected gender: boolean;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
    this.gender = true;
  }

  sayHi (msg: string): string {
    return `I am ${this.name}, ${this.age}`;
  }
}

class Student extends Person {
  private constructor(name: string, age: number) {
    super(name, age);
    console.log(this.gender); // 可以访问到
  }

  static create (name: string, age: number) {
    return new Student(name, age);
  }
}

const tom = new Person('tom', 18);
console.log(tom.name);
// console.log(tom.age); // 报错
// console.log(tom.gender); // 报错

// const bob = new Student('bob', 20); // 报错,因为constructor是privat
```

```
  e的
const jack = Student.create('jack', 18);
```

TypeScript 类的只读属性

1. 用readonly修饰

```
// 只读属性初始化后，不能再修改
class Person {
  name: string;
  private age: number = 27;
  protected readonly gender: boolean;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
    this.gender = true;
  }

  sayHi (msg: string): string {
    return `I am ${this.name}, ${this.age}`;
  }
}
```

TypeScript 类与接口

1. 建议一个接口只规范一种动作

```
export {}

interface Eat {
  eat (food: string): void
}

interface Run {
  run (distance: number): void
}

class Person implements Eat,Run {
```

```
  eat (food: string) {
    console.log(`优雅的进餐: ${food}`);
  }

  run (distance: number) {
    console.log(`直立行走: ${distance}`);
  }
}

class Animal implements Eat,Run {
  eat (food: string) {
    console.log(`呼噜呼噜的吃: ${food}`);
  }

  run (distance: number) {
    console.log(`爬行: ${distance}`);
  }
}
```

TypeScript 抽象类

1. 抽象类包含具体的实现，接口只有声明没有实现
2. 抽象类只能被继承，不能被new实例化
3. 继承了抽象类的类，必须实现抽象类中的抽象方法

```
export {}

abstract class Animal {
  eat (food: string): void {
    console.log(`呼噜呼噜的吃: ${food}`);
  }

  abstract run (distance: number): void
}

class Dog extends Animal {
  run (distance: number): void {
    console.log(`四脚爬行 ${distance}`);
  }
```

```
}

const d = new Dog();
d.eat('食物');
d.run(100);
```

TypeScript 泛型

1. 声明时不指定具体的类型，而在调用时在传递具体的类型。
2. 把我们定义时不能确定的类型，变成一个参数T，在我们使用时来传递这样一个参数

```
// Array就是一个泛型，我们在定义arr时，再具体指定了是number
function createNumberArray (length: number, value: number): number[]
{
  const arr = Array<number>(length).fill(value);

  return arr;
}

function createArray<T> (length: number, value: T) : T[] {
  const arr = Array<T>(length).fill(value);

  return arr;
}

const res = createArray<string>(3, 'foo');
const res2 = createArray<number>(3, 100);
```

TypeScript 类型声明

1. 我们经常要用到npm模块，而他们又不一定是用typescript编写的， 语法是declare
2. 解决的就是，一些代码在定义时没有定义类型，就需要我们自己来重新进行类型声明

```
import { camelCase } from 'lodash'
import qs from 'query-string'; // 这个包默认就有自己的声明.d.ts文件

declare function camelCase (input: string): string

qs.parse('?key=value&key2=value2');
```

```
7 const res = camelCase('hello typed');
```

随堂测试

TypeScript语言随堂测试

成绩单

答题人	吴博	您的得分	35/45	答对题数	7/9

答题解析 全部题目 错题集

您的姓名：*

您的回答：吴博

您的手机号码：*

您的回答：17671686923

1. 下面哪些方式在`JavaScript`中可以实现隐式类型转换（） [分值：5]

您的回答：A 减号 - ¦ C 乘法 * ¦ D 除法 / ✗

正确答案为：A 减号 -|B 加号 +|C 乘法 *|D 除法 /

2. 下列关于语言类型的说法正确的是（） [分值：5]

您的回答：A 强类型语言，要求实参必须跟形参的类型保持一致 ¦ B 强类型不允许随意的隐式类型转换，弱类型允许 ¦ C 弱类型在语言层面不会限制实参的类型 ¦ D 动态类型 运行的时候才知道它的类型 且可以随意修改类型 ✓ （得分：5）

3. 下列哪个指令是创建.flowconfig文件的命令（） [分值：5]

您的回答：B `flow init` ✓ （得分：5）

4. Flow 中下列哪些表示数组类型的方法是正确的() [分值：5]

您的回答：A `const arr:Array<number> = [1,2,3]` ¦ B `const arr:[number] = [1,2,3]` ¦ C `const arr:number[] = [1,2,3]` ✗

正确答案为：A `const arr:Array<number> = [1,2,3]`|C `const arr:number[] = [1,2,3]`

5. Flow 中函数中的参数和返回值都可以设置类型注解（） [分值：5]

您的回答：A 对 ✓ （得分：5）

6. Flow 下列可以表示任意类型的是（） [分值：5]

您的回答：B any ¦ C mixed ✓ （得分：5）

7. 下列有关`TypeScript`的说法正确的是（） [分值：5]

您的回答：A `TypeScript`工具的安装方式可以是`npm install typescript -g` ¦ B 编译`ts`文件的命令是`tsc filename.ts` ¦ C`TypeScript`中接口类型声明的关键字是`interface` ✓ （得分：5）

8. 下列说法正确的是（） [分值：5]

您的回答：B `ts`中接口的成员有可选成员，只读成员，动态成员 ¦ D `ts`中使用class关键字定义类 ✓ （得分：5）

9. 下列关于访问修饰符说法正确的是（） [分值：5]

您的回答：A,`private`关键字定义的是私有成员，只能在类的内部访问 ¦ B,`public`定义的成员是公共成员，在类的内部，外部，子类都可以访问 ¦ C,`protected`定义的是受保护的成员，在自身属性和子类中都可以访问的到。 ✓ （得分：5）