TypeScript

1. 类型注解和编译时类型检查
2. 基于类的面向对象编程
3. 泛型
4. 接口
5. 声明文件

类型注解和编译时类型检查

定义变量后，可以通过冒号来指定类型注解

```
// Hello.vue
let name = "xx"; // 类型推论
let title: string = "开课吧"; // 类型注解
name = 2;// 错误
title = 4;// 错误
```

数组类型

```
let names: string[];
names = ['Tom'];//或Array<string>
```

任意类型

```
let foo:any = 'xx'
foo = 3

//any类型也可用于数组
let list: any[] = [1, true, "free"];
list[1] = 100;
```

函数中使用类型

```
function greeting(person: string): string {
  return 'Hello, ' + person;
}
//void类型，常用于没有返回值的函数
function warnUser(): void { alert("This is my warning message"); }
```

可选参数：参数名后面加上问号，变成可选参数

```
function sayHello(name:string, age?:number): string {
  console.log(name, age)
}
```

参数默认值

```
function sayHello(name:string, age:number=20): string {
  console.log(name, age)
}
```

*函数重载：以参数数量或者参数类型区分

```
// 函数重载：多个同名函数，通过参数数量或者类型不同或者返回值不同
function info(a: { name: string }): string;
function info(a: string): object;
function info(a: any): any {
  if (typeof a === "object") {
    return a.name;
  } else {
    return { name: a };
  }
}
info({ name: "jerry" });
info("jerry");
```

```
function add(a: number, b: number): string;
function add(a: string, b: string): string;
function add(a: any, b: any): string {
  if (typeof a === "number") {
    return "数字相加: " + (a + b);
  } else {
    return "字符串拼接: " + (a + b);
  }
}
console.log(add(1,1));
console.log(add('foo','bar'));
```

```
//Vue项目@Props传递的参数@Props({}) 类型，是否必传，默认值，检验
  @Prop() private msg!: string; // 属性msg必填项，字符串类型
  @Prop({ default: "匿名" }) private foo?: string; // 属性foo必填项，字符串类型
export interface PropOptions<T=any> {
  type?: PropType<T>;
  required?: boolean;
  default?: T | null | undefined | (() => T | null | undefined);
  validator?(value: T): boolean;
}
```

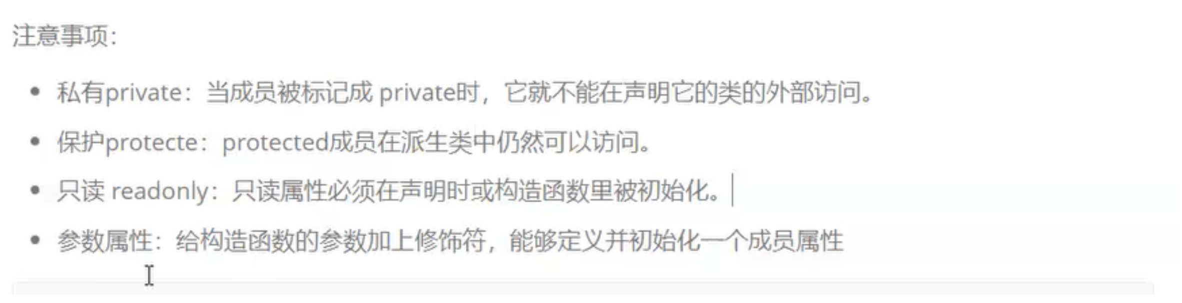
```
 // 普通的属性相当于组件data
  private features: Feature[] = [];

  // 生命周期
  async created() {
    //...
    const result = await getData<Feature>();
    this.features = result.data;
  }

  // 计算属性
  get featureCount() {
    return this.features.length;
  }
```

多态

```
//重写父类方法
class Shape {
  readonly foo: string = "foo";
  protected area: number;

  constructor(public color: string, width: number, height: number) {
    this.area = width * height;
  }

  shoutout() {
    return (
      "I'm " + this.color + " with an area of " + this.area + " cm squared."
    );
  }
}

class Square extends Shape {
  constructor(color: string, side: number) {
    super(color, side, side);
    console.log(this.color);
  }
  shoutout() {
    return "我是" + this.color + " 面积是" + this.area + "平方厘米";
  }
}
const square: Square = new Square("blue", 2);
console.log(square.shoutout());
```

存取器getter 和 setter

```
class Employee {
  private firstName: string = "Mike";
  private lastName: string = "James";

  get fullName(): string {
    return this.firstName + " " + this.lastName;
  }
  set fullName(newName: string) {
    console.log("您修改了用户名");
    this.firstName = newName.split(" ")[0];
    this.lastName = newName.split(" ")[1];
  }
}
const employee = new Employee();

employee.fullName = "Bob Smith";
```


注意事项：

- 私有private：当成员被标记成 private时，它就不能在声明它的类的外部访问。
- 保护protecte：protected成员在派生类中仍然可以访问。
- 只读 readonly：只读属性必须在声明时或构造函数里被初始化。
- 参数属性：给构造函数的参数加上修饰符，能够定义并初始化一个成员属性

类型约束 ---接口

```
// 接口约束结构
interface Person {
  firstName: string;
  lastName: string;
  sayHello(): string; // 要求实现方法
}
// 类实现接口
class Greeter implements Person {
  constructor(public firstName = "", public lastName = "") {}
  sayHello() {
    return "Hello, " + this.firstName + " " + this.lastName;
  }
}
function greeting2(person: Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}
const user = { firstName: "Jane", lastName: "User", sayHello: () => "lalala" };
const user2 = new Greeter("a", "b");
console.log(user);
console.log(greeting2(user2));
```

泛型

泛型是指在定义函数，接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性

```
interface Result<T> {
  ok: 0 | 1;
  data: T[];
}
// 泛型函数
function getData<T>(): Promise<Result<T>> {
  const data: any[] = [
    { id: 1, name: "类型注解", version: "2.0" },
    { id: 2, name: "编译型语言", version: "1.0" }
  ];
  return Promise.resolve({ ok: 1, data } as Result<T>);
}
```