TypeScript基础知识

1.类型注解

```
let str: string;
str = 'string'; // ok
str = 1; // error

let bol = false; // 类型推论
```

2.类型基础

```
let anyData: any; // 任意类型
anyData = 'str';
anyData = 1;
anyData = true;

let arr = string[]; // 字符串数组或者Array<string>
arr = ['xh', 'zs'];

let obj: object; // 不是基本类型就是object
obj = { // ok
  prop: 1
}
obj = 1; //error
// 推荐使用 类型
type Prop = {
  prop: number
}
let obj: Prop;

let arr = any[]; // 任意类型数组
arr = [1, 'xh', true];

function add(arg: string): string { // 参数必须为string类型,返回值必须为string类型
  return 'hello' + arg;
}

function fn(): void { } // 无返回值的函数
```

3.类型断言

```
// 更具体的类型
const anyData: any = 'I am xh';
const len = (anyData as string).length; // number类型不会执行
```

4.联合类型

```
// 希望某个变量或参数的类型是多种类型其中之一
let data: string | number;
data = 'xh';
data = 1;
```

5.类型+交叉类型

```
type First = {
  name: string
}
type Second = {
  sex: number
}
type FirstAndSecond = First & Second;
let data: FirstAndSecond;
data = { // ok
  name: 'xh',
  sex: 1
}
```

6.接口

```
interface Person {
    name: string,
    sex: number
}
```

7.函数

```
// 参数一旦声明，就要求传递，且类型需符合
// 默认值用=
// 可选参数加?
function add (a: string, b = 'xh', c?: string): number {
  return a + b;
}
```

8.类

```
class Person {
  private _name = 'xh'; // 私有属性，不能在类的外部访问
  protected sex = 1; // 保护属性，可以在子类中访问

  // 参数属性:构造函数参数加修饰符，能够定义为成员属性
  constructor(public face = "帅") {};

  // 方法也有修饰符
  private someMethod() {};

  // 存取器:属性方式访问，可添加额外逻辑，控制读写性
  get name() {
    return this._name;
  }
  set name(val) {
    this._name = val;
  }
}
```

9.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。以此增加代码通用性。

```
// 使用泛型
interface Result<T> {
  ok: 0 | 1;
  data: T;
}
// 泛型方法
function getResult<T>(data: T): Result<T> {
  return {ok:1, data};
}
// 用尖括号方式指定T为string
getResult<string>('hello');
// 用类型推断指定T为number
getResult(1);
```

10.装饰器

装饰器是工厂函数，它能访问和修改装饰目标

- 类装饰器

```
//类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数。
function log(target: Function) {
// target是构造函数
    console.log(target === Foo); // true
    target.prototype.log = function() {
      console.log(this.name);
    }
}
@log
class Person {
  name = 'xh'
}
const person = new Person();
person.log();
```

- 方法装饰器

```
// 这里通过修改descriptor.value扩展了bar方法
function log(target: any, name: string, descriptor: any) {
    const baz = descriptor.value;
    descriptor.value = function(val: string) {
        console.log('dong~~');
        baz.call(this, val);
    }
    return descriptor;
}
class Person {
```

```
  @log
  setBar(val: string) {
    this.bar = val
} }
let person = new Person();
person.setBar('lalala')
```

- 属性装饰器

```
// 属性装饰器
function setName(target, name) {
    target[name] = 'xh'
}
class Person {
  @setName name!:string;
}
let person = new Person();
console.log(person.name);
```

- 可接收参数装饰器

```
function setName(param:string) {
    return function (target, name) {
        target[name] = param
    }
}
class Person {
  @setName('xh') name!:string;
}
let person = new Person();
console.log(person.name);
```

TypeScript + Vue的使用

```
vue-property-decorator

import {
  Component,
  Vue,
  Prop,
  Emit,
  Watch
} from 'vue-property-decorator';
import {
```

```
  namespace,
  State
} from 'vuex-class';
const wxCommon = namespace('wxCommon');

@Component({
  name: 'Home'
})
export default class Home extends Vue {
  @State counter;

  @wxCommon.State('wxConfigData')
  wxConfigData!: WxState;

  features: string[] = [];

  fValue = '';

  @Prop() msg?: string;

  created() {
    this.features = ['1', '33', '32'];
  }

  get foo() {
    return this.features.length;
  }

  @Emit('showit')
  addFeature(e: KeyboardEvent) {
    const inp = e.target as HTMLInputElement;
    this.features.push(inp.value);
  }

  @Watch('features')
  changeValue(val: string[], val2: string[]): void {
    window.console.log(val.length, val2.length);
  }

  @Watch('fValue')
  changeValue2(val: string): void {
    window.console.log(val);
  }
}
```