数据类型定义

有哪些

boolean 布尔

number 数字, 整数和浮点

string 字符串

array 数组

tuple 元组

enum 枚举

any 任意

null和undefined, 其他类型的子类型

怎么定义

```
let str: boolean
```

```
let str: string
```

```
let arr0: Array < string >
let arr1: string[]
let arr2: any[]
```

```
let arr: [string, number, booealn]
```

```

enum Flag {
    success = 1,
        error = 2,
        warn = 3,
}
let f: Flag = Flag.success;

```

```
enum Color {
    red,
    green,
    blue,
    alpha,
}
let c: Color = Color.red;


enum Color {
    red,
    green = 5,
    blue,
    alpha,
}
let c: Color = Color.blue;
console.log(c); // 6
```

```
let aany: any = false
aany = 'www'
console.log(aany); // www
```

```
let num: number;
console.log(num); // undefined, 报错
let num1: undefined;
console.log(num1); // undefined
let num2: number | undefined;
console.log(num2); // undefined
let num4: number | undefined | null;

```

```
function lg(): void {
    console.log('void');
}
lg();
```

```ts
let err: never;
err = (() => {
    throw new Error('错误');
})();
// 很少用到
// 备注:throw new Error()会中断后面的执行
```

函数

声明

```ts
function fnum(): number {
    return 123;
}

let func2 = function(): number {
    return 123;
};

let func3 = function(name: string, age: number): string {
    return `${name} is ${age}`;
};
```

可选参数

可选参数必须放到最后, ts不建议放到前面

```ts
let func4 = function(name: string, age ? : number): string {
    return `${name} is ${age ?? 'unknown'}`;
};
```

默认参数

```ts
let func5 = function(name: string, age: number = 21): string {
    return `${name} is ${age}`;
};
```

剩余参数

```
let func6 = function(...result: number[]): number {
    return result.reduce((a: number, b: number) => a + b, 0);
};
```

重载

```
function getInfo0(name: string): string;

function getInfo0(age: number): number;

function getInfo0(str: any): any {
    if (typeof str === 'string') {
        return 'string:' + str;
    } else if (typeof str === 'number') {
        return 'number:' + str;
    }
    return;
}

function getInfo1(name: string): string;

function getInfo1(name: string, age: number): string;

function getInfo1(name: any, age ? : any): any {
    if (!age) {
        return 'name:' + name;
    } else {
        return 'name + age:' + name + age;
    }
}
```

类

```
// es5
function Person(name, age) {
    this.name = 'Meow';
```

```
    this.age = '12';
}
Person.prototype.country = 'China'
// 对象冒充
function Me0() {
    Person.call(this)
}
// 原型链继承
function Me1() {

}
Me1.prototype = new Person()
// 组合继承
function Me2() {
    Person.call(this, name, age)
}
Me2.prototype = Person.prototype
```

```
// ts
// 类只定义了下面的类型但没有在构造函数中创建会报错
// name: string
// 可以通过下面这样定义来解决
// name: string|undefined
class Person0 {
    constructor(name: string) {
        this.name = name;
    }
    name: string;
    run(): void {
        console.log(this.name);
    }
}
```

```
// ts继承
class Person1 extends Person0 {
    constructor(name: string) {
```

```
        super(name);
    }
    age: number = 20;
    run(): void {
        console.log(this.name + ' is running');
    }
}
```

```
// 修饰符 public protected private 不加默认是public
// static 关键字创建静态方法，不需要实例化就可以使用，但静态方法只能访问静态方法和变量
```

```
// 抽象类
// 想要继承抽象类必须实现其中的所有抽象方法
abstract class Animal {
    constructor(name: string) {
        this.name = name;
    }
    name: string;
    abstract eat(): any;
}
class Dog extends Animal {
    constructor(name: string) {
        super(name);
    }
    eat() {
        console.log('什么都吃');
    }
}
```

接口

```
// 属性接口
interface FullName {
    firstName: string;
    lastName: string;
    // 可选属性
```

```
    age ? : number;
}

function printName(name: FullName) {
    console.log(
        `${name.firstName} ${name.lastName}${name.age ? ' is ' + name.age : ''}`
    );
}
printName({ firstName: 'rock', lastName: 'meow' });
printName({ firstName: 'rock', lastName: 'meow', age: 21 });
// 传入的对象如果定义在内部则必须和接口定义的一模一样，在外部则可以包含多余的属性，怀疑可能是外
```

```
// 函数接口
interface encrypt {
    (key: string, value: string): string;
}
let md5: encrypt = function(key: string, value: string): string {
    return `${key}: ${value}`;
};
```

```
// 可索引接口，对数组和对象的约束(不常用)
interface UserArr {
    [index: number]: string;
}
const arr: UserArr = ['aa', 'bb', 'cc', 'dd', 'ee'];
interface UserObj {
    [index: string]: string;
}
const obj: UserObj = { name: 'meow', age: '21' };
```

```
// 类类型接口:对类的约束 和 抽象类有些相似
interface Animal {
    name: string;
    eat(food ? : string): void;
```

```
}
class Dog implements Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    eat(food ? : string) {
        console.log(`${this.name} eat ${food || 'everything'}`);
    }
}
```

```
// 接口扩展：接口可以继承接口
interface Person2 {
    eat(): void;
}
interface Hobby {
    hobby(): void;
}
interface Chinese extends Person2 {
    work(): void;
}
class Reactor implements Chinese {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    eat(): void {
        console.log(`${this.name} eat food`);
    }
    work(): void {
        console.log(`${this.name} write React`);
    }
}
class Meow extends Reactor implements Hobby {
    constructor(name: string) {
        super(name);
    }
    hobby(): void {
```

```
        this.work();
        console.log(`${this.name} play game`);
    }
}
```

泛型

泛型的理解: 在ts提供的全部类型之外添加了一种新的类型, 但并不是真正的添加了新的类型, 而是这种类型可以代表ts中的那些类型或是自定义的类等, 像是一个变量标识符的感觉, 在使用泛型的位置靠前的地方加上<T>来声明, 也可以是其他字母等, 但T较多, 可能代表的是
要实现泛型接口, 那么这个类也必须是一个泛型类

```
// 定义
function getData<T>(value: T): T {
    return value;
}
console.log(getData<number>(123));
console.log(getData<string>('abc'));
function getString<T>(key: string, value: T): string {
    return `${key}:${value}`;
}
console.log(getString<number>('meow', 123));
```

```
// 接口泛型
interface ConfigFn {
    <T>(value1: T, value2: T): T;
}
let getConfig0: ConfigFn = function <T>(value1: T, value2: T): T {
    return value1 > value2 ? value1 : value2;
};
```

```
// 泛型类
class Tem<T> {
    list: T[] = [];
}
```

```ts
// 类作为参数
class User {
    username: string;
    password: string | undefined;
    constructor(username: string, password?: string) {
        this.username = username;
        if (password) {
            this.password = password;
        }
    }
}
class DB {
    addUser(user: User): boolean {
        // 数据库操作
        console.log(user);
        return true;
    }
}
class DBT<T> {
    add(item: T): boolean {
        // 数据库操作
        console.log(item);
        return true;
    }
}
new DB().addUser(new User('meow'));
// 错误写法,不能用
// new DBT<User>().add('aaa');
new DBT<User>().add(new User('meow', '123'));
```

命名空间

```ts
// 多个同名不冲突
namespace A{
    let sum = 10;
}
```

```
namespace B{
    let sum = 20;
}
// 10
let num = A.sum;
```

装饰器

```
// 类装饰器
// 普通装饰器
function logClass(item: any) {
    item.prototype.api = '111';
    item.prototype.func = () => {
        console.log('miku');
    };
}
@logClass
class Decorator {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    getName(): string {
        return this.name;
    }
}
let decorator: any = new Decorator('meow');
console.log(decorator.api);
// 装饰器工厂
function logClassFactory(api: string) {
    return function (item: any) {
        item.prototype.api = api;
        item.prototype.func = () => {
            console.log('miku39');
        };
    };
}
@logClassFactory('http://www.baidu.com')
class DecoratorF {
```

```
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    getName(): string {
        return this.name;
    }
}
let decoratorF: any = new DecoratorF('meoww');
console.log(decoratorF.api);
// 重载构造函数
function overLoad(proto: any) {
    return class extends proto {
        url: string = 'http://www.baidu.com';
        getUrl(): string {
            console.log('meow ---', this.url);
            return this.url;
        }
    };
}
@overLoad
class HttpUrl {
    url: string;
    constructor(url: string) {
        this.url = url;
    }
    getUrl(): string {
        console.log(this.url);
        return this.url;
    }
}
new HttpUrl('http://www.4399.com').getUrl();
```

```
// 属性装饰器
function logProperty(params: any) {
    return function (target: any, attr: any) {
        target[attr] = params;
    };
```

```
}
class logPropertyC {
    @logProperty('miku')
    name: string | undefined;
    constructor() {}
    getName(): string {
        return this.name ?? '';
    }
}
console.log(new logPropertyC().getName());
```

```
// 方法装饰器
function funcDecorator(params: any) {
    console.log(params, '---');
    return function (target: any, name: any, desc: any) {
        let oldFunc = desc.value;
        desc.value = function (...args: any[]) {
            console.log(args);
            args = args.map((value) => value + '');
            oldFunc.apply(this, args);
        };
    };
}
class funcClass {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    @funcDecorator('999')
    get(...params: any[]) {
        console.log(this.name + ' ' + params.join(' '));
    }
}
new funcClass('meow').get('aaa', 123, '444', false);
```

```
// 方法参数装饰器
```

```ts
// 没什么用，不如用类装饰器
function paramsDecorator(params: any) {
    return function (target: any, funcName: any, paramsIndex: any) {
        console.log(params);
        console.log(target, funcName, paramsIndex);
    };
}
class paramsClass {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    get(age: number, @paramsDecorator('ccc') end: any) {
        console.log(`${this.name} is ${age}, ${end}`);
    }
}
new paramsClass('meow').get(21, '@#$#@');
```

装饰器执行顺序

定义在后面的先执行，即靠近的先执行

属性装饰器

方法装饰器

方法参数装饰器2

方法参数装饰器1

类装饰器2

类装饰器1