类的定义

下面是使用 TS 约束属性并实例化对象

```
class User {
    name: string
    age: number
    (n: string, a: number) {
        this.name = n
        this.age = a;
    }

    info(): string {
        return `${this.name}的年龄是 ${this.age}`
    }
}

const hd = new User('后盾人', 12);
const xj = new User('向军', 18)

```

通过约束数组的类型为`User`，使其成员只能是 User 类型对象

```
...
const users: User[] = [hd, xj];
console.log(users);
...

```

#修饰符

下面我们来掌握在 JS 类和 TS 结合使用

#public

下面来介绍第一个访问修饰符 public，指**公开**的属性或方法

- 默认情况下属性是 public （公开的），即可以在类的内部与外部修改和访问
- 不明确设置修饰符即为 public

```
class User {
    public name: string
    public age: number
    (n: string, a: number) {
        this.name = n
        this.age = a;
    }

    public info(): string {
        return `${this.name}的年龄是 ${this.age}`
    }
}

const hd = new User('后盾人', 12);
const xj = new User('向军', 18)

hd.name = "后盾人网站"

for (const key in hd) {
    if (hd.(key)) {
        console.log(key);
    }
}

```

#protected

protected 修饰符指**受保护**的，只允许在父类与子类使用，不允许在类的外部使用

```
class Hd {
    protected name: string
    (name: string) {
        this.name = name
    }
}
class User extends Hd {
    (name: string) {
```

```
        super(name)
    }

    public info(): string {
        return `你好 ${this.name}`
    }
}

const hd = new User('后盾人');

console.log(hd.info());
console.log(hd.name); //属性是 protected 不允许访问

```

#private

private 修饰符指**私有**的，不允许在子类与类的外部使用

父类声明 private 属性或方法子类不允许覆盖

```
class Hd {
    private name: string
    constructor(name: string) {
        this.name = name
    }
    private info(): void { }
}
class User extends Hd {
    constructor(name: string) {
        super(name)
    }

    public info(): void {

    }
}

```

子类不能访问父类的 private 属性或方法

```
class Hd {
    private name: string
    constructor(name: string) {
        this.name = name
    }
}
class User extends Hd {
    constructor(name: string) {
        super(name)
    }

    public info(): string {
        return `你好 ${this.name}` //属性是 private 不允许子类访问
    }
}

```

子类更改父类方法或属性的访问修饰符有些限制的

- 父类的 private 不允许子类修改
- 父类的 protected 子类可以修改为 protected 或 public
- 父类的 public 子类只能设置为 public

```
class Hd {
    private name: string
    (name: string) {
        this.name = name
    }
    public info(): void { }
}
class User extends Hd {
    (name: string) {
        super(name)
    }
    protected info(): string {
        return 'houdunren.com'
    }
}
```

```
16
```

#readonly

readonly 将属性定义为只读，不允许在类的内部与外部进行修改

- 类似于其他语言的 const 关键字

```
class Axios {
    readonly site: string = '
https://houdunren.com/api
    readonly site: string = '
    (site?: string) {
        this.site = site || this.site
    }
    public get(url: string): any[] {
        console.log(`你正在请求 ${this.site + '/' + url}`)
        return []
    }
}

const instance = new Axios('
https://www.houdunwang.com
const instance = new Axios('
instance.get('users')

```

#constructor

构造函数是初始化实例参数使用的，在 TS 中有些细节与其他程序不同

我们可以在构造函数 constructor 中定义属性，这样就不用在类中声明属性了，可以简化代码量

- 必须要在属性前加上 public、private、readonly 等修饰符才有效

```
class User {
    (
        public name: string,
        public age: number
    ) {}

    public info(): string {
```

```
        return `${this.name}的年龄是 ${this.age}`
    }
}

const hd = new User('后盾人', 12);

```

#static

static 用于定义静态属性或方法，属性或方法是属于构造函数的

- 静态属性是属于构造函数的，不是对象独有的，所以是所有对象都可以共享的
- 有关原型与 class 的原理，向军大叔已经在后盾人 (opens new window)上录制了，有时间学习一下，可以了解原理

#语法介绍

下面是 static 使用的语法

```
class Site {
    static url: string = 'houdunren.com'

    static getSiteInfo() {
        return '我们不断更新视频教程在' + Site.url
    }
}
console.log(Site.getSiteInfo());

```

#单例模式

当把 construct 定义为非 public 修饰符后，就不能通过这个类实例化对象了。

```
class User {
    protected () {

    }
}

const hd = new User();//报错
```

```
8
```

我们可以利用这个特性再结合 static 即可实现单例模式，即只实例化一个对象

```
class User {
    static instance: User | null = null;
    protected () { }

    public static make(): User {
        if (User.instance == null) User.instance = new User;

        return User.instance;
    }
}

const hd = User.make();
console.log(hd);

```

#get/set

使用 get 与 set 访问器可以动态设置和获取属性，类似于 vue 或 laravel 中的计算属性

- 我会在后盾人 (opens new window)发布坦克大战游戏项目(你看到文档的时候可能已经发布了）里面的多处用到 get、set
- 在后盾人 (opens new window)已经发布了 JS 中 class 的详细知识，建议有时间看一下

```
class User {
    private _name
    (name: string) {
        this._name = name
    }
    public get name() {
        return this._name;
    }
    public set name(value) {
        this._name = value
    }
}

```

```
const hd = new User('向军')
hd.name = '李四'
console.log(hd.name);

```

因为 get 与 set 是新特性所以编译时要指定 ES 版本

```
tsc 1.ts -w -t es5

```

#abstract

抽象类定义使用 abstract 关键字，抽象类除了具有普通类的功能外，还可以定义抽象方法

- 抽象类可以不包含抽象方法，但抽象方法必须存在于抽象类中
- 抽象方法是对方法的定义，子类必须实现这个方法
- 抽象类不可以直接使用，只能被继承
- 抽象类类似于类的模板，实现规范的代码定义

```
class Animation {
    protected getPos() {
        return { x: 100, y: 300 }
    }
}

class Tank extends Animation {
    public move(): void {

    }
}

class Player extends Animation {
    public move: void{

}

```

上例中的子类都有 move 方法，我们可以在抽象方法中对其进行规范定义

- 抽象方法只能定义，不能实现，即没有函数体
- 子类必须实现抽象方法

```
abstract class Animation {
    abstract move(): void
    protected getPos() {
        return { x: 100, y: 300 }
    }
}

class Tank extends Animation {
    public move(): void {

    }
}

class Player extends Animation {
    public move(): void {

    }
}

```

子类必须实现抽象类定义的抽象属性

```
abstract class Animation {
    abstract move(): void
    abstract name: string
    protected getPos() {
        return { x: 100, y: 300 }
    }
}
class Tank extends Animation {
    name: string = '坦克'
    public move(): void {

    }
}

class Player extends Animation {
    name: string = '玩家'
```

```
17
18      public move(): void {
19
20      }
21  }
22
```

抽象类不能被直接使用，只能被继承

```
1  abstract class Animation {
2      abstract move(): void
3      protected getPos() {
4          return { x: 100, y: 300 }
5      }
6  }
7  const hd = new Animation(); //报错，不能通过抽象方法创建实例
8
```

#Interface

接口用于描述类和对象的结构

- 使项目中不同文件使用的对象保持统一的规范
- 使用接口也会支有规范更好的代码提示

#抽象类

下面是抽象类与接口的结合使用

```
1  interface AnimationInterface {
2      name: string
3      move(): void
4  }
5  abstract class Animation {
6      protected getPos(): { x: number; y: number } {
7          return { x: 100, y: 300 }
8      }
9  }
10
11 class Tank extends Animation implements AnimationInterface {
```

```
    name: string = '敌方坦克'
    public move(): void {
        console.log(`${this.name}移动`)
    }
}

class Player extends Animation {
    name: string = '玩家'
    public move(): void {
        console.log(`${this.name}坦克移动`)
    }
}
const hd = new Tank()
const play = new Player()
hd.move()
play.move()

```

#对象

下面使用接口来约束对象

```
interface UserInterface {
    name: string;
    age: number;
    isLock: boolean;
    info(other:string): string,
}

const hd: UserInterface = {
    name: '后盾人',
    age: 18,
    isLock: false,
    info(o:string) {
        return `${this.name}已经${this.age}岁了,${o}`
    },
}

console.log(hd.info());
```

```
18
```

如果尝试添加一个接口中不存在的函数将报错，移除接口的属性也将报错。

```
const hd: UserInterface = {
                ...
    houdurnen() { }  //“houdurnen”不在类型“UserInterface”中
}

```

如果有额外的属性，使用以下方式声明，这样就可以添加任意属性了

```
interface UserInterface {
    name: string;
    age: number;
    isLock: boolean;
    [key:string]:any
}

```

#接口继承

下面定义游戏结束的接口 `PlayEndInterface` ，AnimationInterface 接口可以使用 `extends` 来继承该接口

```
interface PlayEndInterface {
    end(): void
}
interface AnimationInterface extends PlayEndInterface {
    name: string
    move(): void
}

class Animation {
    protected getPos(): { x: number; y: number } {
        return { x: 100, y: 300 }
    }
}

```

```
class Tank extends Animation implements AnimationInterface {
    name: string = '敌方坦克'
    public move(): void {
        console.log(`${this.name}移动`)
    }
    end() {
        console.log('游戏结束');
    }
}

class Player extends Animation implements AnimationInterface {
    name: string = '玩家'
    public move(): void {
        console.log(`${this.name}坦克移动`)
    }
    end() {
        console.log('游戏结束');
    }
}
const hd = new Tank()
const play = new Player()
hd.move()
play.move()

```

对象可以使用实现多个接口，多个接口用逗号连接

```
interface PlayEndInterface {
    end(): void
}
interface AnimationInterface {
    name: string
    move(): void
}

class Animation {
    protected getPos(): { x: number; y: number } {
        return { x: 100, y: 300 }
    }
```

```
}

class Tank extends Animation implements AnimationInterface, PlayEndInterface {
    name: string = '敌方坦克'
    public move(): void {
        console.log(`${this.name}移动`)
    }
    end() {
        console.log('游戏结束');
    }
}

class Player extends Animation implements AnimationInterface, PlayEndInterface {
    name: string = '玩家'
    public move(): void {
        console.log(`${this.name}坦克移动`)
    }
    end() {
        console.log('游戏结束');
    }
}
const hd = new Tank()
const play = new Player()
hd.move()
play.move()

```

#函数

下面使用 UserInterface 接口约束函数的参数与返回值

- 会根据接口规范提示代码提示
- 严格约束参数类型，维护代码安全

函数参数

下面是对函数参数的类型约束

```
interface UserInterface {
    name: string;
    age: number;
    isLock: boolean;
```

```
}

function lockUser(user: UserInterface, state: boolean): UserInterface {
    user.isLock = state;
    return user;
}

let user: UserInterface = {
    name: '后盾人', age: 18, isLock: false
}

lockUser(user, true);
console.log(user);

```

函数声明

使用接口可以约束函数的定义

```
interface Pay {
    (price: number): boolean
}
const getUserInfo: Pay = (price: number)=>true

```

#构造函数

下面的代码我们发现需要在多个地方使用对 user 类型的定义

```
class User {
    info: { name: string, age: number }
    (user: { name: string, age: number }) {
        this.info = user
    }
}
const hd = new User({ name: '后盾人', age: 18 })
console.log(hd);

```

使用 interface 可以优化代码，同时也具有良好的代码提示

```
interface UserInterface {
    name: string,
    age: number
}
class User {
    info: UserInterface
    (user: UserInterface) {
        this.info = user
    }
}
const hd = new User({ name: '后盾人', age: 18 })
console.log(hd);

```

#数组

对数组类型使用接口进行约束

```
const hd: UserInterface = {
    name: '后盾人',
    age: 18,
    isLock: false
}

const xj: UserInterface = {
    name: '向军',
    age: 16,
    isLock: false
}

const users: UserInterface[] = [];
users.push(hd, xj)
console.log(users);

```

#枚举

下面是使用枚举设置性别

```
enum SexType {
    BOY, GIRL
}

interface UserInterface {
    name: string,
    sex: SexType
}

const hd: UserInterface = {
    name: '后盾人',
    sex: SexType.GIRL
}
console.log(hd); //{ name: '后盾人', sex: 1 }

```

#案例

下面是 index.ts 文件的内容，通过 interface 接口来限制支付宝与微信支付的规范

```
interface PayInterace {
    handle(price: number): void
}

class AliPay implements PayInterace {
    handle(price: number): void {
        console.log('支付宝付款');
    }
}
class WePay implements PayInterace {
    handle(price: number): void {
        console.log('微信付款');
    }
}

//支付调用
function pay(type: string, price: number): void {
    let pay: PayInterace
```

```
    if (type == 'alipay') {
        pay = new AliPay()
    } else {
        pay = new WePay()
    }
    pay.handle(price)
}

```

然后执行编译

```
tsc index.ts -w

```

界面处理 index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>后盾人</title>
        <script src="index.js" defer></script>
    </head>
    <body>
        <button onclick="pay('alipay',100)">支付宝</button>
        <button onclick="pay('wepay',200)">微信付款</button>
    </body>
</html>

```

#type

type 与 interface 非常相似都可以描述一个对象或者函数，使用 type 用于定义类型的别名，是非常灵活的类型定义方式。

- type 可以定义基本类型别名如联合类型，元组
- type 与 interface 都是可以进行扩展
- 使用 type 相比 interface 更灵活

- 如果你熟悉其他编程语言，使用 interface 会让你更亲切
- 使用类(class) 时建议使用接口，这可以与其他编程语言保持统一
- 决定使用哪个方式声明类型，最终还是看公司团队的规范

#基本使用

下面是使用 type 声明对象类型

```
type User = {
    name: string,
    age: number
}
const hd: User = { name: '后盾人', age: 18 }

```

上面已经讲解了使用 interface 声明函数，下面来看使用 type 声明函数的方式

```
type Pay = (price: number) => boolean
const wepay: Pay = (price: number) => {
    console.log(`微信支付${price}`);
    return true;
}

wepay(100)

```

#类型别名

type 可以为 number、string、boolean、object 等基本类型定义别名，比如下例的 IsAdmin。

```
//基本类型别名
type IsAdmin = boolean

//定义联合类型
type Sex = 'boy' | 'girl'
```

```
type User = {
    isAdmin: IsAdmin,
    sex: Sex
}
const hd: User = {
    isAdmin: true,
    sex: "boy"
}

//声明元组
const users: [User] = [hd]

```

#索引类型

type 与 interface 在索引类型上的声明是相同的

```
interface User {
    [key: string]: any
}

type UserTYpe = {
    [key: string]: any
}

```

#声明继承

typescript 会将同名接口声明进行合并

```
interface User {
    name: string
}
interface User {
    age: number
}
const hd: User = {
    name: '后盾人',
```

```
    age: 18
}

```

interface 也可以使用 extends 继承

```
interface Admin {
    role: string
}
interface User extends Admin {
    name: string
}
const hd: User = {
    role: 'admin',
    name: '后盾人',
}

```

interface 也可以 extends 继承 type

```
type Admin = {
    role: string
}
interface User extends Admin {
    name: string
}
const hd: User = {
    role: 'admin',
    name: '后盾人',
}

```

type 与 interface 不同，存在同名的 type 时将是不允许的

```
type User {
    name: string
}
type User {
    age: number
```

```
}

```

不过可以使用& 来进行 interface 的合并

```
interface Name {
    name: string
}
interface Age {
    age: number
}
type User = Name & Age

```

下面是 type 类型的声明合并

```
type Admin = {
    role: string,
    isSuperAdmin: boolean
}
type Member = {
    name: string
}

type User = Admin & Member;

const hd: User = {
    isSuperAdmin: true,
    role: 'admin',
    name: '后盾人'
}

```

下面声明的是满足任何一个 type 声明即可

```
type Admin = {
    role: string,
    isSuperAdmin: boolean
}
```

```ts
type Member = {
    name: string
}

type User = Admin | Member;

const hd: User = {
    role: 'admin',
    name: '后盾人'
}

```

#implements

class 可以使用 implements 来实现 type 或 interface

```ts
type Member = {
    name: string
}

class User implements Member {
    name: string = '后盾人'
}
```