javascript es6

定义方法

let

- 只在作用域中生效，var在作用域外也能访问，类似于全局定义
- 去除的了变量提升的特性，规定调用前一定要先定义
- 死区，在块级作用域内，使用let声明之前，该变量是不可用的

var

- 会有变量提升，在定义变量定义之前引用，会显示对象已经定义，

const

- 是一个常量，比let多的一个特性只读，声明后不运行修改，这个不允许只是不允许修改const定义的内存地址

最佳实践，不用var，主用const，涉及要改变内存地址的用let

解构

数组解构

数组中是直接使用数组下标进行位置进行解构

对象解构

对象中是没有所谓下标，所以需要对应对象中的属性名来进行解构

```
const obj = { age: 18 };

const { name: objname = "jack" } = obj; //:后面跟的是重命名,=后面跟的是默认值
console.log(objname);
```

模板字符串函数

```
const { log } = console;
```

```
2
3 const name = "tom";
4 const sex = false;
5 // 模板字符串函数会将传入的字符串根据你加入的变量进行分割成几段，在函数内进行组
合
6 function myTagFunc(strings, name, gender) {
7   const sex = gender ? "man" : "woman";
8   return strings[0] + name + strings[1] + sex + strings[2];
9 }
10 const result = myTagFunc`hey, ${name} is a ${sex}`;
11 log(result);
```

字符串扩展犯法

```
1 const message = "Error: foo is not defined.";
2
3 message.startsWith("Error")  //查找字符串开头是否以Error开头
4 message.endsWith(".")    //查找字符串结尾是否以.结尾
5 message.includes("foo")  //查找字符串中间是否包含foo
```

参数默认值

```
1 const { log } = console;
2
3 function foo(bar, enable = true)
4     // enable = enable || true  这种形式，对于传一个false进来一样会使用默认值
5     // enable = enable === undefined ? true : enable  上面的方法应该这样写
6     log('foo invoked - enable: ')  //默认值的参数必须放后面
7     log(enable)
```

剩余参数

```
1 function foo(first, ...args) {
2   console.log(args);
3 }
4 foo(1, 2, 3, 4);
```

对象字面量的增强

```
1 const bar = '345'
2
3 const obj = {
4     foo: 123,
5     // bar: bar 变量定义中属性名和属性值相同的情况下,可以直接省略后面的
6     bar,
7     // method1: function () {
8     //     console.log('method111')
```

```
    // }
     // 在函数定义时，上面可省略成下面的格式
     method1 () {
         console.log('method111')
     },
     // 传统是不支持在变量名中定义函数调用,es6中加上[]就可以执行任意表达式
     [Math.random()]: 123
}
```

proxy 方法 监听方法

proxy有着defineProperty所没有的功能

下面图片是proxy所能支持的监听的回调，以及回调的使用方法

handler 方法	触发方式
get	读取某个属性
set	写入某个属性
has	in 操作符
deleteProperty	delete 操作符
getProperty	Object.getPropertypeOf()
setProperty	Object.setPrototypeOf()
isExtensible	Object.isExtensible()
preventExtensions	Object.preventExtensions()
getOwnPropertyDescriptor	Object.getOwnPropertyDescriptor()
defineProperty	Object.defineProperty()
ownKeys	Object.keys()、Object.getOwnPropertyNames()、Object.getOwnPropertySymbols()
apply	调用一个函数
construct	用 new 调用一个函数

```
const person = {
  name: "zsf",
  age: 25,
};

// 对于proxy相当于就是一个监听函数,监听对象的查看和删除,并给出对于的回调
const personProxy = new Proxy(person, {
  get(target, property) {
    // 给变量做一个判断给设置默认值
     return property in target ? target[property] : "default";
   },
   // target  是指列表里面的所有内容  property 是指引用的字段名称 value 是设置的属性值
   set(target, property, value) {
       if (property === 'age') {
           if(!Number.isInteger(value)) {
               throw new TypeError(`${value} is not an int`)
           }
```

```
        }
        target[property] = value
    }
});
console.log(personProxy.xxx);
```

Reflect

```
const obj = {
    name: "zsd",
    age: 18,
};

// Reflect 出现的原因是因为，在一下属性的调用的太乱,既有应用in delete 字符,又有调用keys方式的,格式不统一
console.log("name" in obj);
// console.log(delete obj["age"]);
console.log(Object.keys(obj));

// 下面Reflect的调用就显得格式统一,不会有杂乱无章的感觉
// Reflect 里面支持的方式名称总共有13个,在proxy的图里面handler那一行
console.log(Reflect.has(obj, "name"));
console.log(Reflect.deleteProperty(obj, "age"));
console.log(Reflect.ownKeys(obj, "name"));
```

prototype 原型

```
function Person(name) {
    // this表示当前作用域下的this
    this.name = name;
    // 这是对象定义的方法,这种定义能在创建实例后，通过.来调用
    // 并且对象定义,只是局限于自己实例化的对象,会产生效果
    this.Introduce = function () {
        console.log("这是一个对象方法");
    };
}
// 在 prototype 上定义方法,相当于在对象的所有实例上都加上这个方法
// 每个对象在创建时都会包含 prototype 这个原型对象
Person.prototype.say = function () {
    console.log(`hi, my name is ${this.name}`);
};


// 在对象定义的属性和方法中prototype是没有办法直接进行重写属性或方法的
```

```
function Aclass(){
    this.Property = 1;
    this.Method = function(){
        console.log(1);
    }
    }
// 这里执行对象属性的重写是可以执行,但是不会生效
// 体现出对象定义的方式要比原型定义的优先级高
// 原型定义一般是定义一些通用方法和属性的
Aclass.prototype.Property = 2;
Aclass.prototype.Method = function(){
    console.log(2);
}
var obj = new Aclass();
console.log(obj.Property);
obj.Method();



// 子类重写父类的属性和方法
function AClass() {
  this.Property = 1;
  this.Method = function () {
    console.log(1);
  };
}
function AClass2() {
  this.Property2 = 2;
  this.Method2 = function () {
    console.log(2);
  };
}
//通过原型的方式把 Aclass父类中的方式类似继承的方式加入到了 AClass2中
AClass2.prototype = new AClass();
// 因为是通过prototype原型的方式继承的,所以可以通过原型来重写
AClass2.prototype.Property = 3;
AClass2.prototype.Method = function () {
  console.log(4);
};
var obj = new AClass2();
```

```
console.log(obj.Property);
obj.Method();

Person.Run = function () {
  console.log("这是一个类的方法");
};
```

class类

```
class Person {
  // 这就是一个类
  // constructor 这个函数,相当于就是__init__
  constructor(name) {
    this.name = name;
  }
  // 这里就相当于定义的一个方法
  say() {
    console.log(`hi, my name is ${this.name}`);
   }
   // 此处是静态方法,这个里面是不能使用this指向的
   static create(name) {
     return new Person(name);
   }
}
const tom = Person.create("tom");
tom.say();
const p = new Person("t");
p.say();

```

类的继承

```
class Person {
  // 这就是一个类
  // constructor 这个函数,相当于就是__init__
  constructor(name) {
    this.name = name;
  }
  // 这里就相当于定义的一个方法
  say() {
    console.log(`hi, my name is ${this.name}`);
```

```
  }
}
// extends 是继承关键字,后面跟的是继承的类
class Student extends Person {
  constructor(name, number) {
    // 这里的super就是代指父类的意思,将name值传给父类
    super(name);
    this.number = number;
  }
  hello() {
    // 通过super来调用父类的say方法
    super.say();
    console.log(`my school number is ${this.number}`);
  }
}
const s = new Student("jack", "100");
s.hello();
```

Set

```
const arr = [1, 2, 3, 4, 1, 2, 3];
//Set和数组类似,但是Set中重复的值会被忽略
// 下面写法就是将数组中的重复值通过Set去掉,然后展开返回一个数组
const result = [...new Set(arr)];
console.log(result);
```

Map

```
const m = new Map();
const tom = { name: "tom" };
m.set(tom, 90);
console.log(m);
console.log(m.get(tom));

// 分别打印出键和值
m.forEach((value, key) => {
  console.log(value, key);
});
```

Symbol

```
// 通过for然后加上字符串就可以产生多个一样的标识
const s1 = Symbol.for("foo");
const s2 = Symbol.for("foo");
console.log(s1 === s2);
```

```
5
6  const a = {
7    // 通过下面这个字段就能重新定义,对象的toString标签
8    // 最多用于对象的私有属性的属性名
9    [Symbol.toStringTag]: "XObject",
10 };
11 // 这里打印出来叫对象的toString标签
12 // 默认是 [object Object]
13 console.log(a.toString());
14 // 这个属性专门用于获取Symbol的属性名
15 console.log(Object.getOwnPropertySymbols(a));
```

可迭代接口

```
1  // 必须要挂载一个Iterator 才能进行迭代
2
3  // Set 函数对象中包含了Symbol.iterator 这个对象,因为有这个对象所以才能被迭代
4  const t = new Set([1, 2]);
5  const inter = t[Symbol.iterator]();
6  // 函数在被循环的时候，都会执行Symbol.iterator 方法,然后通过方法里的next进行返回值
7  console.log(inter.next());
8  console.log(inter.next());
9
10 //在对象中只要添加一个Symbol.iterator 方法，就能进行迭代了
11 const obj = {
12   store: ["foo", "bar", "baz"],
13   // 此处实现可迭代接口Iteralble
14   [Symbol.iterator]: function () {
15     let index = 0;
16     const self = this;
17     // 迭代接口  Iterator
18     return {
19       // 必须要有一个可以迭代的next方法
20       next: function () {
21         const result = {
22           value: self.store[index],
23           // done表示是否为结尾,为true就是停止
24           done: index >= self.store.length,
25         };
26         index++;
27         // 实现迭代结果接口  IterationResult
```

```
        return result;
      },
    };
  },
};

for (let o of obj) {
  console.log(o);
}
```

this 调用域的问题

```
var num = 20;
const obj = {
    num: 10,
    func: num => {
        this.num += 5;
        //1、浏览器-25 箭头函数没有this，此处this指向window,而浏览器全局变量都会挂载到window对象下
        //1、node-NaN 箭头函数没有this，this指向全局。node环境没有window,全局this指向空对象{},
        //  全局变量不会挂载到全局this,所以node环境this.num 是undefined，undefinde + 5 = NaN
        console.log(this.num);
        console.log(this)
        num += 5;
        // 2、45 此处为func入参加5的结果，无疑义
        console.log(num);
        var num = 30;
        return function() {
            this.num += 4;
            // 3、浏览器-29
            //  此处闭包函数在全局作用于下调用，
            // 相当于:
            // const func1 = obj.func(40);
            //  func1();
            // this同func，指向window,全局变量num挂载到window下
            //  调用func：window.num+5，  此处window.num+4，20 + 5 + 4所以是29；
            // 3、node-NaN 原因同1，全局变量未挂载到window
            console.log(this.num);
            num += 10;
            // 4、40 此处num为闭包函数外缓存的num，初始值未30，此处+10，所以是40，无疑义
            console.log(num)
        }
    }
}
obj.func(40)()
```

ES2017

```
const obj = {
```

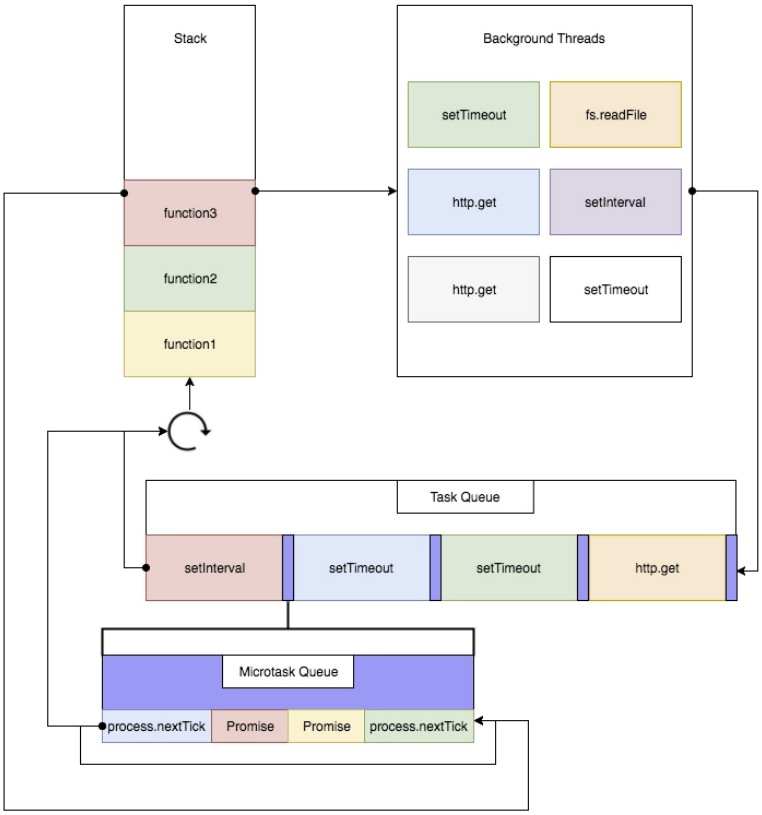
```
  foo: "value1",
  bar: "value2",
};
// 打印出对象的内容
console.log(Object.values(obj));
// 将键值对输出成列表
console.log(Object.entries(obj));
const p1 = {
  lastname: "E",
  firstname: "lei",
  // 可以设置fullname来对字符串进行重新定义,调用方法跟属性调用一样
  get fullname() {
    return this.firstname + this.lastname;
  },
};
console.log(p1.fullname);
// 在对象的复制中,会把get属性当作一个普通的属性进行复制，所以值不会改变
// assign 就是用后面的对象,覆盖前面的对象,共同值被后面的覆盖,没有的值就添加
const p2 = Object.assign({}, p1);
p2.firstname = "zce";
console.log(p2);

// getOwnPropertyDescriptors 这函数就是为了正确的取出get或set中的值的对象
const descriptors = Object.getOwnPropertyDescriptors(p1);
// console.log(w);
// 将包含get或set值的对象进行合并
const p3 = Object.defineProperties({}, descriptors);
p3.firstname = "zce";
console.log(p3.fullname);

const books = {
  html: 5,
  css: 16,
  javascript: 128,
};
for (const [name, conut] of Object.entries(books)) {
  // padEnd 结尾 总长度,不足用-补足   padStart 开头  总长度  不足用0补足
  console.log(`${name.padEnd(16, "-
")}|${conut.toString().padStart(3, "0")}`);
}
```

单线程 同步模式，异步模式说明

Promise

```
Promise.resolve('foo') // 把一个字符串或者其他的,作为一个成功的Promise对象进行返回

Promise.reject()// 创建一个状态为失败的Promise对象

Promise.all() // 在队列中执行多个Promise的对象,所有对象执行完成后,才会结束

Promise.race() // 在队列中执行多个Promise的对象,第一个结束后,这个任务就结束了

//宏任务 宏任务执行完后,会添加到队列的末尾 settimeout 属于宏任务 大部分异步都属于宏任务

//微任务 会在当前任务执行完后,立即执行下一个子任务不会重新排队 Promise属于微任务
```

浏览器的Event loop



这张图将浏览器的Event Loop完整的描述了出来，我来讲执行一个JavaScript代码的具体流程：

1. 执行全局Script同步代码，这些同步代码有一些是同步语句，有一些是异步语句（比如setTimeout等）；
2. 全局Script代码执行完毕后，调用栈Stack会清空；
3. 从微队列microtask queue中取出位于队首的回调任务，放入调用栈Stack中执行，执行完后microtask queue长度减1；
4. 继续取出位于队首的任务，放入调用栈Stack中执行，以此类推，直到直到把microtask queue中的所有任务都执行完毕。注意，如果在执行microtask的过程中，又产生了microtask，那么会加入到队列的末尾，也会在这个周期被调用执行；
5. microtask queue中的所有任务都执行完毕，此时microtask queue为空队列，调用栈Stack也为空；
6. 取出宏队列macrotask queue中位于队首的任务，放入Stack中执行；
7. 执行完毕后，调用栈Stack为空；
8. 重复第3-7个步骤；
9. 重复第3-7个步骤；
10. ......

可以看到，这就是浏览器的事件循环Event Loop

这里归纳3个重点：

1. 宏队列macrotask一次只从队列中取一个任务执行，执行完后就去执行微任务队列中的任务；
2. 微任务队列中所有的任务都会被依次取出来执行，知道microtask queue为空；
3. 图中没有画UI rendering的节点，因为这个是由浏览器自行判断决定的，但是只要执行UI rendering，它的节点是在执行完所有的microtask之后，下一个macrotask之前，紧跟着执行UI render。

TypeScript

- 类型安全

- 分为 强类型 弱类型 他们的区别就是，会不会隐式的进行类型转换
  - 静态类型就是变量定义是确定了类型，就不允许更改
  - 动态类型是代码执行时动态给定的，可以随意更改

flow

```
安装flow
npm install flow-bin --dev
安装编译移除注解,因为带着注解是没有办法用node运行的
npm install flow-remove-types --dev

关闭vscode的语法校验，设置 => 搜索 javascript validate 关闭就好了


指定运行之后的文件存放位置
npm flow-remove-types . -d dist


babel配合插件时间注解移除
@babel/core 是babel的核心模块 @babel/preset-flow 包含了转换注解的插件
npm install @babel/core @babel/cli @babel/preset-flow --dev
配置
在.babelrc配置文件中添加
{ "presets": ["@babel/preset-flow"]}
//编译模块中的文件
npx babel src -d dist

开发插件 Flow language Support 插件


a: flow原始类型
a: string 存储字符串
a: number 数字类型,NaN也算是数字类型 Infinity 无穷大，赋值时使用
a: boolean = false 和 true
```

```
a: null = null
a: void = undefined
a: symbol = Symbol()

flow 数组类型

arr1: Array<number> 表示全部由数字组成的数组
arr1: number[] 表示全部由数字组成的数组
// 元组
arr1: [string, number] = 这样写表示，这个数组只能设置两个参数,一个字符串一个数字


 flow 对象类型
 obj1: { foo: string, bar: number} = {foo: 'string, bar:100} 表示这个数组必须要有这两个键值对，而且数据类型一致
 obj2: { foo?: string, bar: number} 这样表示foo这个键值对可有可无
 obj3: { [string]:string } = {} 这样表示键值对里的键必须是字符串


 flow 函数类型

 function foo (callback: (string, number) => void){
 callback('string',100)
 }
 这样表示这个回调函数必须要有两个参数,一个字符串一个数字，不要有返回值

 flow 特殊类型
 type: 'success' | 'warning' | 'danger' = 'success'
 type: 'string' | 'number' = '100'
 这个表示，输入的答案必须是这三个里面的值，不然报错

 别名
 type StringOrNumber = string | number 设置了这个别名其他地方就可以直接使用
 b: StringOrNumber = 'string'

 function passMixed (value: mixed){
 // 这里的强类型需要加类型判断，不然会报错
 if (typeof value === 'string'){

 }
 }
```

```
function passMixed (value: any){
// 这里的弱类型，可以随便使用

}

flow 文档链接
https://www.saltycrane.com/cheat-sheets/flow-type/latest


```

相关文档的声明文件

```
- 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

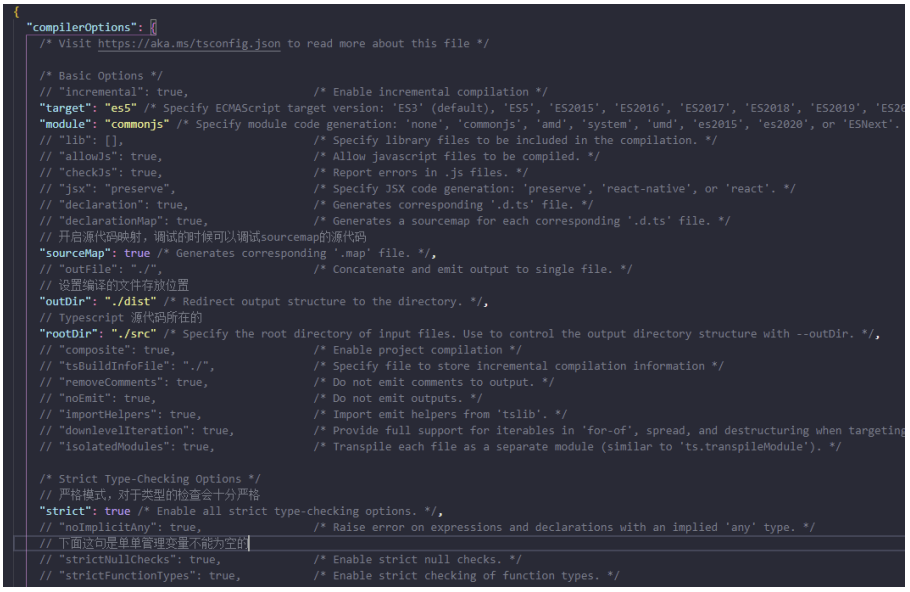
typescript的定义和概念以及优缺点

- Javascript 是一种轻量级的解释性脚本语言，一般用于html页面的开发
- Typescript 是javascript的超集，包含了javascript的所有元素，可以载入javascript的代码运行，并扩展了javascript的语法
- TypeScript 可以使用javaScript中额所有代码和编码概念，TypeScript是为了是javaScript的开发变得更加容易而创建的
- TypeScript的优势
- 更好的协作，更强的生产力，更快捷的项目重构，代码质量更好，更清晰

- 1. TypeScript的优点

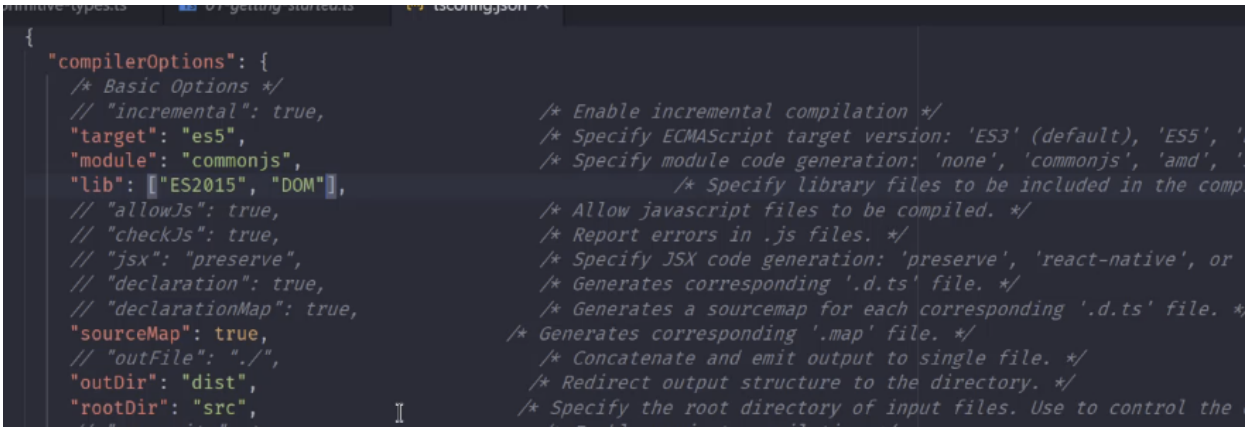
    - 1. 有更好的代码错误检查，更健全的代码规范
    - 2. 编写的代码更加健壮，使得代码质量更好更清晰
    - 3. 使用typescript工具来重构变得更加容易和快捷
    - 4. 干净的 ECMAScript 6 代码，自动完成和动态输入等因素有助于提高开发人员的工作效率
- 2. TypeScript的缺点
    - 1. 学习成本提高，需要理解接口，泛型，类，枚举等新的概念
    - 2. 短期可能会怎加一些开发成本，笔记要多写一些类型的定义
    - 3. 集成到构建流程需要些工作量

```
npm install typescript --dev
//执行typescript进行编译
npx tsc 01.ts

设置配置文件
npx tsc --init

非严格模式下，string number，boolean 都是可以为空，严格模式下就会报错
```



配置文件一些常用的东西

使用sysbol promise和一些es6的语法，编译成es5就会报错，需要将标准库改成es2015 的，因为单独改一个会consolog会报错，所以要加上Dom的标准库



中文错误信息

```
Typescript 里面
npx tsc --locale zh-CN 就会显示中文的错误信息

vscode中 设置 搜索 typescript locale 选择zh-CN vscode中的错误提示也会是中文的
```

作用域问题

定义其他文件中的存在的变量名，会报重复变量定义

```
在模块中添加一个exprot {} 就会创建一个单独的作用域
```

枚举类型

```
enum poststatus {
Draft = 0,
Unpublished = 1,
```

```
Published = 2
}
```

类型断言

```
const num1 = res as number
```

接口

```
export {}; // 确保跟其他示例没有成员冲突

interface Post {
  title: string;
  content: string;
  subtitle?: string; //表示这个变量可有可无
  readonly summary: string;  // 只读,在变量赋值后,就不能更改了
}

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


printPost({ title: "ptrhon", content: "java" });

// 这样写就可以设置随意的键值了
interface Cache {
  // prop这里是一个格式，不是代表某一个
  [prop: string] : string
}
```

class 类

```
export {}

class Person {
  name: string
  age: number

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

```

```
    sayHi (msg: string): void {
      console.log(`I am ${this.name},${msg}`)
    }
}
```

类与接口

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

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

class Person implements Eat, Run {  // 这里用接口相当于定义了一个模型,你在这个类中需要用模型中的方法,不然会报错
    eat (food: string): void {
      console.log(`优雅的进餐: ${food}`)
    }

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

抽象类

```
abstract class Person  {  // abstract 抽象类关键字，被定义成抽象类就只能够被继承，不能创建实例了
    eat (food: string): void {
      console.log(`优雅的进餐: ${food}`)
    }
    abstract run (distance: number): void // 抽象类方法,定义了抽象类方法，继承类必须定义这个方法
}

class Dog extends Person {
    run(distance: number): void {
      throw new Error("Method not implemented.")
    }

}
```

泛型

```ts
function createNumberArray (length: number, value: number): number[] {
  const arr = Array<number>(length).fill(value)
  return arr
}

function createStingArray (length: number, value: string): string[] {
  const arr = Array<string>(length).fill(value)
  return arr
}
// 泛型  跟上面对比一下就能看出，泛型相当于是用一个占位符占住位置，等待后面输入的类型来确定类型
function createArray<T> (length: number, value: T): T[] {
  const arr = Array<T>(length).fill(value)
  return arr
}

const res = createArray<string>(3,'val')
```

类型声明

```ts
// 安装一个例子模块
npm install lodash --dev

import { camelCase } from 'lodash'


// 这是lodash的类型声明插件，一般插件在使用的时候都会提示你安装这个类似的插件
// npm install @types/lodash --dev

// 这里是进行变量声明，就是声明你应该输入一些什么类型的数据
// 这是在插件中没有进行变量声明的插件，可以这样手动声明

// declare function camelCase (input: string): string

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