TypeScript

安装、编译

```
cnpm install -g typescript

tsc helloword.ts
```

```
1、 tsc --init 生成tscconfig.json  改"outDir":'./js'

2、vscode 任务-运行任务-监视
```

进入类型的世界：理解原始类型与对象类型

原始类型的类型标注

```
const name1: string = "zhangzf99";
const age: number = 10;
const bool: boolean = true;
const undef: undefined = undefined;
const nul: null = null;
const obj: object = { name1 };
const bigint1: bigint = 90909907986756746545n;
const bigint2: bigint = BigInt(90909907986756746545);
const symbolVar: symbol = Symbol("unique");
```

null 和 undefined

在 js 中，null -> 这里有值，但是个空值，undefined -> 这里没有值 在 ts 中，null 和 undefined 类型都是有具体意义的类型。也就是说，它们作为类型时，表示的是一个有意义的具体类型值。这两者在没有开启 strictNullChecks 检查的情况下，会**被视作其他类型的子类型**，比如 string 类型会被认为包含了 null 与 undefined 类型：

```
const temp1: null = null;
const temp2: undefined = undefined;
const temp3: string = null; // 仅在关闭 strictNullChecks 时成立，下同
const temp4: string = undefined;
```

void

ts 的类型标注中也有 void，用于描述一个内部没有 return 语句，或者没有显示 return 一个值的函数的返回值。

```
function func1() {}
function func2() {
  return;
}
function func3() {
  return undefined;
}
```

func1 与 func2 的返回值都会被隐式推导为 void，只有显示返回了 undefined 值的 func3 其返回值才会被推导为了 undefined。但在实际的代码执行中，func1 与 func2 的返回值均是 undefined。

> 虽然 func3 的返回值类型会被推导为undefined，但是你仍然可以使用 void 类型进行标注，因为在类型层面 func1、func2、func3 都表示“没有返回一个有意义的值”。

```
function func1() {}
function func2() {
  return;
}
function func3(): void {
  return undefined;
}
```

数组的类型标注

```
const arr1: string[] = ["111", "222"];
const arr2: Array<number> = [111, 2];
```

数组是我们在日常开发大量使用的数据结构，但在某些情况下，使用**元组**来代替数组更加妥当，比如一个数组中只存放固定长度的变量，但我们进行了超出长度的访问：

```
console.log(arr1[999]); // undefined
```

这种是不符合预期的，因为我们能确定这个数组中只有三个成员，并希望在越界访问时给出类型报错。这时我们可以使用元组类型进行类型注释：

```
const arr3: [string, number, null] = ["111", 11, null];
console.log(arr3[999]); // 报错 长度为 "3" 的元组类型 "[string, number, null]" 在索引 "999" 处没有元素。
```

在这种情况下，对数组合法边界内的索引访问（即 0、1、2）将精准地获得对应位置上的类型。同时元组也支持了再某一位置上的可选成员：

```
const arr4: [string, number?, boolean?] = ["111", 11];
// 鼠标放在arr4上 const arr4: [string, (number | undefined)?, (boolean |
undefined)?]
console.log(arr4.length); // 2
```

对于标记为可选的成员，在 --strictNullCheckes 配置下会被视为一个 string | undefined 的类型。此时数组长度属性也会发生变化，比如上面的元组 arr4， 其长度类型为 1 | 2 | 3。

可能会觉得元组的可读性实际上并不好。比如对于 [string, number, boolean]来说，你并不能直接知道这三个元素都代表什么，还不如使用对象的形式。在 ts4.0 中，有了具名元组的支持。

```
const arr5: [name: string, age: number, male: boolean] = ["zhangzf", 18, true];

// 可选修饰符
const arr6: [name: string, age: number, male?: boolean] = ["zhangzf", 18];
```

实际上除了显示地越界访问，还可能存在隐式地越界访问，如通过解构赋值的形式：

```
const arr7: string[] = [];
const [ele1, ele2, ...rest] = arr7;
```

对于数组，此时仍然无法检查出是否是否存在隐式访问，因为类型层面并不知道它到底有多少个元素。但对于数组，隐式的越界访问也能够被揪出来一个警告：

```
const arr8: [string, number, boolean] = ["111", 11, true];
const [name2, age2, male, other] = arr8; // 长度为 "3" 的元组类型 "[string, number,
boolean]" 在索引 "3" 处没有元素。
```

对象的类型标注

类似于数组类型，在 ts 中我们也需要特殊的类型标注来描述对象类型，即 interface，你可以理解为它代表了这个对象对外提供的接口结构。 首先我们使用 interface 声明了一个结构，然后使用这个结构作为一个对象的类型标注即可：

```
interface IDescription {
  name: string;
  age: number;
  male: boolean;
```

```
}
const obj1: IDescription = {
  name: "zhangzf",
  age: 18,
  male: true,
};
```

这里的"描述"指：

- 每一个属性的值必须**一一对应**到接口的属性类型
- 不能有多的属性，也不能有少的属性，包括直接在对象内部声明，或是 obj1.other = 'xxx' 这样属性访问赋值的形式。

除了声明属性以及属性的类型以外，我们还可以对属性进行修饰，常见的修饰包括**可选**与**只读**这两种。

修饰接口属性

```
interface IDescription1 {
  name: string;
  age: number;
  male?: boolean;
  func?: Function;
}
const obj2: IDescription1 = {
  name: "zhangzf",
  age: 18,
  male: true,
};
```

假设新增一个可选的函数类型属性，然后进行调用：obj2.func()，此时将会产生一个类型报错：不能调用可能是未定义的方法。但可选属性标记不会影响你对这个属性进行赋值，如：

```
boj2.male = false;
obj2.func = () => {};
```

即使你对可选属性进行了赋值，TypeScript 仍然会使用**接口的描述为准**进行类型检查，你可以使用类型断言、非空断言或可选链解决。 除了标记一个属性为可选以外，你还可以标记这个属性为只读：readonly。很多同学对这一关键字比较陌生，因为以往 js 中并没有这一概念，它的作用是**防止对象的属性被再次赋值**。

```
interface IDescription2 {
  readonly name: string;
  age: number;
}
const obj3: IDescription2 = {
  name: "zhangzf",
  age: 18,
```

```
};
obj3.name = "zhangzf99"; // 无法分配到 "name" ，因为它是只读属性。
```

其实在数组与元组层面也有着只读的修饰，但与对象类型有着两处不同。

- 你只能将整个数组/元组标记为只读，而不能像对象那样标记某个属性为只读。
- 一旦被标记为只读，那这个只读数组/元组的类型上，将不再具有 push、pop 等方法（即会修改原数组的方法），因此报错信息也将是**类型 xxx 上不存在属性“push”这种。**这一实现的本质是**只读数组与只读元组的类型实际上变成了 ReadyonlyArray，而不是 Array。**

type 与 interface

interface 可以用来描述对象、类的结构 type 类型别名可以用来**将一个函数签名、一组联合类型、一个工具类型等等抽离成一个完整独立的类型。**但大部分场景下接口都可以被类型别名所取代。

unique symbol

Symbol 在 js 中代表着一个唯一的值类型，类似于字符串类型，可以作为对象的属性名，并用于避免错误修改对象/Class 内部属性的情况。而在 ts 中，symbol 类型并不具有这一特性，一百个具有 symbol 类型的对象，它们的 symbol 类型指的都是 ts 中同一类型。为了实现“独一无二”的这个特性，ts 中支持了 unique symbol 这一类型声明，它是 symbol 类型的子类型，每一个 unique symbol 类型都是独一无二的。

```
const uniqueSymbolFoo: unique symbol = Symbol("zhangzf");
const uniqueSymbolBar: unique symbol = uniqueSymbolFoo; // 不能将类型“typeof
uniqueSymbolFoo”分配给类型“typeof uniqueSymbolBar”。
```

在 js 中，我们可以用 Symbol.for 方法来复用已创建的 Symbol，如 Symbol.for('zhangzf')会首先查找全局是否已经有使用 zhangzf 作为 key 的 Symbol 注册，如果有，则返回这个 Symbol，否则才会创建新的 Symbol。

掌握字面量类型与枚举，让你的类型再精确一些

```
interface IRes {
  code: number;
  status: string;
  data: any;
}
```

以上代码是一个接口结构，它描述了响应的消息结构，在大多数情况下，这里的 code 与 status 实际值会来自于一组确定值的集合，比如 code 可能是 10000 / 10001 / 50000，status 可能是 “success”/“failure”。而上面的类型只给出了一个宽泛的 number(string)，此时我们既不能在访问 code 时获得精确的提示，也失去了 ts 类型即文档的功能。

字面量类型与联合类型

```ts
interface Res {
  code: 10000 | 10001 | 50000;
  status: "success" | "failure";
  data: any;
}
```

字面量类型

“success”是一个值，为什么可以作为类型？在 ts 中，这叫做字面量类型，它代表着比原始类型更精确的类型，同是也是原始类型的子类型。 字面量类型主要包括**字符串字面量类型、数字字面量类型、布尔字面量类型、对象字面量类型**，他们可以直接作为类型标注：

```ts
const str: "zhangzf" = "zhangzf";
const num: 59 = 59;
const bool1: true = true;
```

```ts
// 报错！不能将类型“"linbudu599"”分配给类型“"linbudu"”。
const str1: "linbudu" = "linbudu599";

const str2: string = "linbudu";
const str3: string = "linbudu599";
```

上面的代码中，原始类型可以包括任意的同类型值，而字面量类型要求的是值级别的字面量一致。

单独使用字面量类型比较少见，因为单个字面量类型并没有什么实际意义。它通常和联合类型（即这里的 |）一起使用，表达一组字面量类型：

```ts
interface Tmp {
  bool: true | false;
  num: 1 | 2 | 3;
  str: "lin" | "bu" | "du";
}
```

联合类型

一组类型的可用集合，只要最终赋值的类型属于联合类型的成员之一，就可以认为符合这个联合类型。联合类型对其成员并没有任何限制，除了上面这样对同一类型字面量的联合，还可以将各种类型混合到一起：

```ts
interface Tmp {
  mixed: true | string | 599 | {} | (() => {}) | (1 | 2);
}
```

注意

- 对于联合类型中的函数类型，需要使用 （）包裹起来
- 函数类型并不存在字面量类型，因此这里的 (()=>{}) 就是一个合法的函数类型
- 你可以在联合类型中进一步嵌套联合类型，但这些嵌套的联合类型最终都会别展平到第一级中

联合类型的常用场景之一是通过多个对象类型的联合，来实现手动的互斥属性，即这一属性如果有字段 1，那就没有字段 2：

```
interface Tmp1 {
  user:
    | {
        vip: true;
        expires: string;
      }
    | {
        vip: false;
        promotion: string;
      };
}
declare var tmp: Tmp1;
if (tmp.user.vip) {
  console.log(tmp.user.expires);
}
```

在这个例子中，user 属性会满足普通用户与 vip 用户两种类型，这里 vip 属性的类型基于布尔字面量类型声明。我们在实际使用时可以通过判断此属性为 true，确保接下来的类型推导都会将其类型收窄到 vip 用户的类型（即联合类型的第一个分支）。这一能力的使用涉及类型守卫与类型控制流分析。

我们也可以通过类型别名来复用一组字面量联合类型：

```
type code = 10000 | 10001 | 50000;
type status = "success" | "failure";
```

对象字面量类型

类似的，对象字面量类型就是一个对象类型的值。当然，这也就意味着这个对象的值全都为字面量值：

```
interface Tmp2 {
  obj: {
    name: "zhangzf";
    age: 18;
  };
}
const tmp: Tmp2 = {
  obj: {
    name: "zhangzf",
    age: 18,
```

```
  },
};
```

如果要实现一个对象字面量类型，意味着完全的实现这个类型每一个属性的每一个值。对象字面量类型在实际开发中的使用较少，我们只需要了解。 总的来说，在需要更精确类型的情况下，我们可以使用字面量类型加上联合类型的方式，将类型从 string 这种宽泛的原始类型直接收窄到 “resolved" | "pending" | "rejected" 这种精确地字面量类型集合。 需要注意的是，无论是原始类型还是对象类型的字面量类型，它们的本质都是类型而不是值。它们在编译时同样会被擦除，同时也是被存储在内存中的类型空间而非值空间。 如果说字面量类型是对原始类型的进一步扩展（对象字面量类型使用较少），那么枚举在某些方面则可以理解为是对对象类型的扩展。

枚举

```
enum PageUrl {
  Home_Page_Url = "url1",
  Setting_Page_Url = "url2",
  Share_Page_Url = "url3",
}
const home = PageUrl.Home_Page_Url;
```

这些常量被真正地**约束在一个命名空间**下。如果没有声明枚举的值，它会默认使用数字枚举，并且从 0 开始，以 1 递增。

```
enum Items {
  Foo,
  Bar,
  Baz,
}
```

在这个例子中，Items.Foo，Items.Bar，Items.Baz 的值依次是 0，1，2。

如果你只为某一个成员指定了枚举值，那么之前未赋值成员仍然会使用从 0 递增的方式，之后的成员则会开始从枚举值递增。

```
enum Items1 {
  // 0
  Foo,
  Bar = 599,
  // 600
  Baz,
}
```

在数字枚举中，你可以使用延迟求值的枚举值，比如函数：

```
const returnNum = () => 100 + 499;
enum Items2 {
  Foo = returnNum(),
  Bar = 599,
  Baz,
}
```

需要注意的是，延迟求值的枚举值是有条件的。**如果你使用了延迟求值，那么没有使用延迟求值的枚举成员必须放在使用常量枚举值声明的成员之后，或者放在第一位**

```
enum Items3 {
  Baz,
  Foo = returnNum(),
  Bar = 599,
}
```

ts 中也可以同时使用字符串枚举值和数字枚举值：

```
enum Mixed {
  Num = 599,
  Str = "zhangzf",
}
```

枚举和对象的重要差异在于，**对象是单向映射的**，我们只能从键映射到键值。而**枚举是双向映射的**，即你可以从枚举成员映射到枚举值，也可以从枚举值映射到枚举成员：

```
enum Items4 {
  Foo,
  Bar,
  Baz,
}
const fooValue = Items4.Foo; // 0
const fooKey = Items[0]; // 'Foo'
```

常量枚举

常量枚举和枚举类似，只是其声明多了一个 const：

```
const enum Items5 {
  Foo,
  Bar,
  Baz,
}
const fooValue1 = Items.Foo; // 0
```

它和普通枚举的差异主要在访问性与编译产物。对于常量枚举，你只能通过枚举成员访问枚举值（而不能通过值访问成员）。同时，在编译产物中并不会存在一个额外的辅助对象，对枚举成员的访问会被**直接内联替换为枚举的值。**

函数与 Class 中的类型，详解函数重载与面向对象

函数

函数的类型签名

如果说变量的类型是描述了这个变量的值类型，那么函数的类型就是描述了**函数入参类型与函数返回值类型**，它们同样使用**🤫*的语法进行类型标注。

```
function foo(name: string): number {
  return name.length;
}
```

在函数类型中同样存在着类型推导。比如在这个例子中，可以不写返回值处的类型，它也能被正确推导为 number 类型。

在 js 中，我们称**function name(){}这一声明函数的方式为函数声明。**除了函数声明外，我们还可以通过**函数表达式**，即**const foo = function(){}**的形式声明一个函数。在表达式中进行类型声明的方式是这样的：

```
const foo1 = function (name: string): number {
  return name.length;
};
```

我们也可以像对变量进行类型标注那样，对 foo 这个变量进行类型声明：

```
const foo2: (name: string) => number = function (name) {
  return name.length;
};
```

这里的**(name: string) => number**看起来眼熟，它是 ES6 的重要特性之一：箭头函数。但在这里，它其实是 ts 中的**函数类型签名**。而实际的箭头函数，我们的类型标注也是类似的：

```
const foo3 = (name: string): number => {
  return name.length;
};

const foo4: (name: string) => number = (name) => {
  return name.length;
};
```

```
type FucnFoo = (name: string) => number;
const foo5: FucnFoo = (name) => {
  return name.length;
};
```

如果只是为了描述这个函数的类型结构，我们甚至可以使用 interface 来进行函数声明：

```
interface FuncFooStruct {
  (name: string): number;
}
```

void 类型

在 ts 中，一个没有返回值（即没有调用 return 语句）的函数，其返回类型应当被标记为 void 而不是 undefined，即使它实际的值是 undefined。

```
function foo6(): void {}
function bar(): void {
  return;
}
```

原因和我们在原始类型与对象类型一节中讲到的：在 ts 中，undefined 类型是一个实际的、有意义的类型值，而 void 才代表着空的、没有意义的类型值。因此在我们没有实际返回值时，使用 void 类型能更好地说明这个函数**没有进行返回操作**。但在上面的第二个例子中，其实更好的方式是使用 undefined：

```
function bar1(): undefined {
  return;
}
```

此时该函数表达的是，这个函数进行了返回操作，但没有返回实际的值。

可选参数与 rest 参数

在很多时候，我们会希望函数的参数可以更灵活，比如它不一定全部都必传，当你不传入参数时函数会使用此参数的默认值。正如在对象类型中我们使用**?**描述一个可选属性一样，在函数类型中我们也使用**?**描述一个可选参数：

```
// 在函数逻辑中注入可选参数默认值
function foo7(name: string, age?: number): number {
  const inputAge = age || 18;
  return name.length + inputAge;
}
// 直接为可选参数声明默认值
```

```
function foo8(name: string, age: number = 18): number {
  const inputAge = age;
  return name.length + inputAge;
}
```

需要注意的是，**可选参数必须位于必选参数之后。**毕竟在 js 中函数的入参是按照位置（形参），而不是按照参数名（名参）进行传递。当然，我们也可以直接将可选参数与默认值合并，但此时就不能够使用**?**了，因为既然都有默认值，那肯定是可选参数了。

对于 rest 参数的类型标注也比较简单，由于其实际上是一个数组，这里我们也应当使用数组类型进行标注：

```
function foo10(arg1: string, ...rest: any[]) {}

// 使用元组类型进行标注
function foo11(arg1: string, ...rest: [number, string]) {}
```

重载

在某些逻辑较复杂的情况下，函数可能有多组入参类型和返回值类型：

```
function func(foo: number, bar?: boolean): string | number {
  if (bar) {
    return String(foo);
  } else {
    return foo * 10;
  }
}
```

在这个实例中，函数的返回类型基于其入参 bar 的值，并且从其内部逻辑中我们知道，当 bar 为 true，返回值为 string 类型，否则为 number 类型。而这里的类型签名完全没有体现这一点，我们只知道它的返回值是这么个联合类型。

要想实现与入参关联的返回值类型，我们可以使用 ts 提供的**函数重载签名**，将以上的例子使用重载改写：

```
function func1(foo: number, bar: true): string;
function func1(foo: number, bar?: false): number;
function func1(foo: number, bar?: boolean): string | number {
  if (bar) {
    return String(foo);
  } else {
    return foo * 599;
  }
}
const res1 = func(599); // number
const res2 = func(599, true); // string
const res3 = func(599, false); // number
```

这里我们的三个 function func1 其实具有不同的意义：

- function func(foo: number, bar: true): string，重载签名一，传入 bar 的值为 true 时，函数返回值为 string 类型。
- function func(foo: number, bar?: false): number，重载签名二，不传入 bar，或传入 bar 的值为 false 时，函数返回值为 number 类型。
- function func(foo: number, bar?: boolean): string | number，函数的实现签名，会包含重载签名的所有可能情况。

基于重载签名，我们就实现了将入参类型和返回值类型的可能情况进行关联，获得了更精确的类型标注能力。

注意：拥有多个重载声明的函数再被调用时，是按照重载的什么顺序往下查找的。因此在第一个重载声明中，为了与逻辑中保持一致，即在 bar 为 true 时返回 string 类型，这里我们需要将第一个重载声明的 bar 声明为必选的字面量类型。

异步函数、Generator 函数等类型签名

```
async function asyncFunc(): Promise<void> {}
function* genFunc(): Iterable<void> {}
async function* asyncGenFunc(): AsyncIterable<void> {}
```

Class

```
class Foo {
  prop: string;
  constructor(inputProp: string) {
    this.prop = inputProp;
  }

  print(addon: string): void {
    console.log(`${this.prop} and ${addon}`);
  }

  get propA(): string {
    return `${this.prop}+A`;
  }

  set propA(value: string) {
    this.prop = `${value}+A`;
  }
}
```

注意：setter 方法不允许进行返回值的类型标注，可以理解为 setter 的返回值并不会被消费，它是一个只关注过程的函数。类的方法同样可以进行函数那样的重载，且语法基本一致。

就像函数可以通过**函数声明**与**函数表达式**创建一样，类也可以通过**类声明**和**类表达式**的方式创建。

```
const Foo1 = class {
  prop: string;
  constructor(inputProp: string) {
    this.prop = inputProp;
  }
  print(addon: string): void {
    console.log(`${this.prop} and ${addon}`);
  }
};
```

修饰符

在 ts 中我们能够为 Class 成员添加这些修饰符: **public / private / protected / readonly**。除了 readonly 以外，其他都是属于访问性修饰符，而 readonly 属于操作性修饰符 （就和 interface 中的 readonly 意义一致）。这些修饰符应用的位置在成员命名前：

```
class Foo {
  private prop: string;

  constructor(inputProp: string) {
    this.prop = inputProp;
  }

  protected print(addon: string): void {
    console.log(`${this.prop} and ${addon}`);
  }

  public get propA(): string {
    return `${this.prop}+A`;
  }

  public set propA(value: string) {
    this.propA = `${value}+A`;
  }
}
```

- public：此类成员在 **类、类的实例、子类** 中都能被访问。
- private：此类成员仅能在**类的内部**被访问。
- protected：此类成员仅能在**类与子类中**被访问，你可以将类和类的实例当做两种概念，即一旦实例化完毕，那就和类没有关系了，即**不允许再访问受保护的成员**。

当你不显示使用访问性修饰符，成员的访问性默认会被标记为 public。实际上，在上面的例子中，我们通过构造函数为类成员赋值的方式还是略显麻烦，需要声明类属性以及在构造函数中进行赋值。简单起见，我们可以**在构造函数中对参数应用访问性修饰符。**

```
class Foo5 {
  constructor(public arg1: string, private arg2: boolean) {}
```

```
}
new Foo5("zhangzf", true);
```

此时，参数会被直接作为类的成员（即实例的属性），免去后续的手动赋值。

静态成员

在 ts 中，可以使用 static 关键字来标识一个成员为静态成员：

```
class Foo6 {
  static staticHandle() {}
  public instanceHandle() {}
}
```

不同于实例成员，在类的内部静态成员无法通过 this 来访问，需要通过 Foo.staticHandle 这种形式进行访问。

```
var Foo = /** @class */ (function () {
  function Foo() {}
  Foo.staticHandler = function () {};
  Foo.prototype.instanceHandler = function () {};
  return Foo;
})();
```

从上述代码中可以看到，**静态成员直接被挂载在函数体上，而实例成员被挂载在原型上。**这就是二者最重要的差异：**静态成员不会被实例继承，它始终只属于当前定义的这个类（以及其子类）。**而原型对象上的实例成员则会**沿着原型链进行传递**，也就是能够被继承。

继承、实现、抽象类

与 js 一样，ts 中也使用 extends 关键字实现继承。

```
calss Base{}
class Derived extends Base{}
```

对于这两个类，比较严谨的称呼是**基类**与**派生类。** 基类中的哪些成员可以被派生类访问，完全是由其访问性修饰符决定的。派生类中可以访问到使用 public 或 protected 修饰符的基类成员。除了访问以外，基类中的方法也可以在派生类中被覆盖，但是可以通过 super 访问到基类中的方法。

```
class Base {
  print() {}
}
class Derived extends Base {
  print() {
    super.print();
```

```
  }
}
```

抽象类

抽象类是对类结构与方法的抽象，简单来说，**一个抽象类描述了一个类中应当有哪些成员（属性、方法等），一个抽象方法描述了这一方法在实际实现中的结构。**我们知道类的方法和函数非常相似，包括结构，因此抽象方法其实描述的就是这个方法的**入参类型**与**返回值类型。**

抽象类使用 abstract 关键字声明：

```
abstract class AbsFoo {
  abstract absProp: string;
  abstract get absGetter(): string;
  abstract absMethod(name: string): string;
}
```

注意，抽象类中的成员也需要使用 abstract 关键字才能被视为抽象类成员，如这里的抽象方法。

```
class Foo12 implements AbsFoo {
  absProp: string = "zhangzf";
  get absGetter() {
    return "zhangzf";
  }
  absMethod(name: string): string {
    return name;
  }
}
```

此时，我们必须完全实现这个抽象类的每一个抽象成员。需要注意的是，在 ts 中**无法声明静态的抽象成员。**

对于抽象类，它的本质就是描述类的结构。看到结构，你是否又想到了 interface？是的，interface 不仅可以声明函数结构，也可以声明类的结构：

```
interface FooStruct {
  absProp: string;
  get absGetter(): string;
  absMethod(input: string): string;
}
class Foo7 implements FooStruct {
  absProp: string = "linbudu";

  get absGetter() {
    return "linbudu";
  }

  absMethod(name: string) {
```

```
    return name;
  }
}
```

在这里，使用类去实现了一个接口。这里接口的作用和抽象类一样，都是**描述这个类的结构**。

探秘内置类型：any、unknown、never 与类型断言

any

使用 any 的场景

- 如果是类型不兼容报错导致使用 any ，考虑使用类型断言替代。
- 如果是类型太复杂导致使用 any ，考虑将这一处的类型去断言为你需要的最简类型。
- 如果想要表达一个未知类型，可以使用 unknown。

known

unknown 类型和 any 类型有点类似，一个 unknown 类型的变量可以再次赋值为任意其他类型，但只能赋值给 any 与 unknown 类型的变量。

```
let unknownVar: unknown = "linbudu";

unknownVar = false;
unknownVar = "linbudu";
unknownVar = {
  site: "juejin",
};

unknownVar = () => {};

const val1: string = unknownVar; // Error
const val2: number = unknownVar; // Error
const val3: () => {} = unknownVar; // Error
const val4: {} = unknownVar; // Error

const val5: any = unknownVar;
const val6: unknown = unknownVar;
```

unknown 和 any 的一个主要差异体现在赋值给别的变量时，any 放弃了所有的类型检查，而 unknown 没有。这一点也体现在对 unknown 类型的变量进行属性访问时：

```
let unknownVar: unknown;
unknownVar.foo(); // 报错：对象类型为 unknown
```

要对 unknown 类型进行属性访问，需要进行类型断言，即"虽然这是一个未知的类型，但是我跟你保证它在这里就是这个类型"。

```
let unknownVar: unknown;
(unknownVar as { foo: () => {} }).foo();
```

在类型未知的情况下，更推荐使用 unknown 标注。这相当于使用额外的心智负担保证了类型再各处的结构，后续重构为具体类型时也可以获得最初始的类型信息，同时还保证了类型检查的存在。

never

```
type UnionWithNever = "linbudu" | 599 | true | void | never;
```

鼠标悬浮在类型别名上，会发现显示的类型时"linbudu" | 599 | true | void。 never 类型被直接无视掉了，而 void 仍然存在。这是因为，void 作为类型表示一个空类型，就像没有返回值的函数使用 void 来作为返回值类型标注一样，void 类型就像 js 中的 null 一样代表“这里有类型，但是是个空类型”。

而 never 才是一个“什么都没有”的类型，它甚至不包括空的类型，严格来说，**never 类型不携带任何的类型信息**，因此会在联合类型中被直接移除，比如我们看 void 和 never 的类型兼容性：

```
declare let v1: never;
declare let v2: void;
v1 = v2; // X 类型void不能赋值给类型never
v2 = v1;
```

在编程语言的类型系统中，never 类型被称为 Bottom Type，是**整个类型系统层级中最底层的类型**。和 null 和 undefined 一样，它是所有类型的子类型，但只有 never 类型的变量能够赋值给另一个 never 类型变量。

通常我们不会显示的声明一个 never 类型，它主要被类型检查所使用。但在某些情况下使用 never 是符合逻辑的，比如一个只负责抛出错误的函数：

```
function justThrow(): never {
  throw new Error();
}
```

类型断言：警告编译器不准报错

类型断言能够显示告知类型检查程序当前这个变量的类型，可以进行类型分析的修正类型。它其实就是一个将变量的已有类型更改为新指定类型的操作，它的基本语法是 as NewType，你可以将 any/unknown 类型断言到一个具体的类型：

```
let unknownVar: unknown;
(unknownVar as { foo: () => {} }).foo();
```

还可以 as 到 any 来，跳过所有的类型检查：

```
const str: string = "zhangzf";
(str as any).func().foo().prop;
```

也可以在联合类型中断言一个具体的分支：

```
function foo(union: string | number) {
  if ((union as string).includes("linbudu")) {
  }

  if ((union as number).toFixed() === "599") {
  }
}
```

断言的正确使用方式，在 ts 类型分析不正确或不符合预期时，将其断言为此处的正确类型：

```
interface IFoo {
  name: string;
}
declare const obj: {
  foo: IFoo;
};
const { foo = {} as IFoo } = obj;
```

这里从 {} 字面量类型断言为了 IFOO 类型，即为结构赋值默认值进行了预期的类型断言。当然，更严谨的方式应该是定义为 Partial 类型，即 IFoo 的属性均为可选。 除了使用 as 语法以外，也可以使用 <> 语法。 需要注意的是，类型断言应当是在迫不得已的情况下使用的。虽然说我们可以用类型断言纠正不正确的类型分析，但类型分析在大部分场景下还是可以满足我们的需求的。

双重断言

如果在使用类型断言时，原类型与断言类型之间差异过大，ts 会给一个类型报错。

```
const str: string = "linbudu";

(str as unknown as { handler: () => {} }).handler();

// 使用尖括号断言
(<{ handler: () => {} }>(<unknown>str)).handler();
```

这是因为断言类型和原类型的差异太大，需要先断言到一个通用的类，即 any/unknown。这一通用类型包含了所有可能的类型，因此**断言到它**和**从它断言到另一个类型**差异不大。

非空断言

类型编程好帮手： ts 类型工具（上）

类型别名

```
type A = string;
```

通过 type 关键字声明了一个类型别名 A，同时它的类型等价于 string 类型。类型别名的作用主要是对一组类型或一个特定类型结构进行封装，以便于在其他地方进行复用。

比如抽离一组联合类型：

```

```