介绍

TypeScript的核心原则之一是对值所具有的*shape*进行类型检查。
它有时被称做“鸭式辨型法”或“结构性子类型化”。
在TypeScript里，接口的作用就是为这些类型命名和为你的代码或第三方代码定义契约。

接口初探

下面通过一个简单示例来观察接口是如何工作的：

```
function printLabel(labelledObj: { label: string }) {
  console.log(labelledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

类型检查器会查看`printLabel`的调用。
`printLabel`有一个参数，并要求这个对象参数有一个名为`label`类型为`string`的属性。
需要注意的是，我们传入的对象参数实际上会包含很多属性，但是编译器只会检查那些必需的属性是否存在，并且其类型是否匹配。
然而，有些时候TypeScript却并不会这么宽松，我们下面会稍做讲解。

下面我们重写上面的例子，这次使用接口来描述：必须包含一个`label`属性且类型为`string`：

```
interface LabelledValue {
  label: string;
}

function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

`LabelledValue`接口就好比一个名字，用来描述上面例子里的要求。
它代表了有一个`label`属性且类型为`string`的对象。
需要注意的是，我们在这里并不能像在其它语言里一样，说传给`printLabel`的对象实现了这个接口。我们只会去关注值的外形。
只要传入的对象满足上面提到的必要条件，那么它就是被允许的。

还有一点值得提的是，类型检查器不会去检查属性的顺序，只要相应的属性存在并且类型也是对的就可以。

可选属性

接口里的属性不全都是必需的。
有些是只在某些条件下存在，或者根本不存在。
可选属性在应用“option bags”模式时很常用，即给函数传入的参数对象中只有部分属性赋值了。

下面是应用了“option bags”的例子：

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): {color: string; area: number} {
  let newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

带有可选属性的接口与普通的接口定义差不多，只是在可选属性名字定义的后面加一个`?`符号。

可选属性的好处之一是可以对可能存在的属性进行预定义，好处之二是可以捕获引用了不存在的属性时的错误。
比如，我们故意将`createSquare`里的`color`属性名拼错，就会得到一个错误提示：

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = {color: "white", area: 100};
  if (config.color) {
    // Error: Property 'clor' does not exist on type 'SquareConfig'
    newSquare.color = config.clor;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

只读属性

一些对象属性只能在对象刚刚创建的时候修改其值。
你可以在属性名前用`readonly`来指定只读属性:

```
interface Point {
    readonly x: number;
    readonly y: number;
}
```

你可以通过赋值一个对象字面量来构造一个`Point`。
赋值后，`x`和`y`再也不能被改变了。

```
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
```

TypeScript具有`ReadonlyArray<T>`类型，它与`Array<T>`相似，只是把所有可变方法去掉了，因此可以确保数组创建后再也不能被修改：

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
ro[0] = 12; // error!
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

上面代码的最后一行，可以看到就算把整个`ReadonlyArray`赋值到一个普通数组也是不可以的。
但是你可以用类型断言重写：

```
a = ro as number[];
```

`readonly` vs `const`

最简单判断该用`readonly`还是`const`的方法是看要把它做为变量使用还是做为一个属性。
做为变量使用的话用`const`，若做为属性则使用`readonly`。

额外的属性检查

我们在第一个例子里使用了接口，TypeScript让我们传入`{ size: number; label: string; }`到仅期望得到`{ label: string; }`的函数里。
我们已经学过了可选属性，并且知道他们在“option bags”模式里很有用。

然而，天真地将这两者结合的话就会像在JavaScript里那样搬起石头砸自己的脚。
比如，拿`createSquare`例子来说：

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
    // ...
}

let mySquare = createSquare({ colour: "red", width: 100 });
```

注意传入`createSquare`的参数拼写为*`colour`*而不是`color`。

在JavaScript里，这会默默地失败。

你可能会争辩这个程序已经正确地类型化了，因为`width`属性是兼容的，不存在`color`属性，而且额外的`colour`属性是无意义的。

然而，TypeScript会认为这段代码可能存在bug。
对象字面量会被特殊对待而且会经过*额外属性检查*，当将它们赋值给变量或作为参数传递的时候。
如果一个对象字面量存在任何“目标类型”不包含的属性时，你会得到一个错误。

```
// error: 'colour' not expected in type 'SquareConfig'
let mySquare = createSquare({ colour: "red", width: 100 });
```

绕开这些检查非常简单。
最简便的方法是使用类型断言：

```
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

然而，最佳的方式是能够添加一个字符串索引签名，前提是你能够确定这个对象可能具有某些做为特殊用途使用的额外属性。
如果`SquareConfig`带有上面定义的类型的`color`和`width`属性，并且*还会*带有任意数量的其它属性，那么我们可以这样定义它：

```
interface SquareConfig {
    color?: string;
    width?: number;
    [propName: string]: any;
}
```

我们稍后会讲到索引签名，但在这我们要表示的是`SquareConfig`可以有任意数量的属性，并且只要它们不是`color`和`width`，那么就无所谓它们的类型是什么。

还有最后一种跳过这些检查的方式，这可能会让你感到惊讶，它就是将这个对象赋值给一个另一个变量：
因为`squareOptions`不会经过额外属性检查，所以编译器不会报错。

```
let squareOptions = { colour: "red", width: 100 };
let mySquare = createSquare(squareOptions);
```

要留意，在像上面一样的简单代码里，你可能不应该去绕开这些检查。
对于包含方法和内部状态的复杂对象字面量来讲，你可能需要使用这些技巧，但是大部额外属性检查错误是真正的bug。
就是说你遇到了额外类型检查出的错误，比如“option bags”，你应该去审查一下你的类型声明。
在这里，如果支持传入`color`或`colour`属性到`createSquare`，你应该修改`SquareConfig`定义来体现出这一点。

函数类型

接口能够描述JavaScript中对象拥有的各种各样的外形。
除了描述带有属性的普通对象外，接口也可以描述函数类型。

为了使用接口表示函数类型，我们需要给接口定义一个调用签名。
它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型。

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
```

这样定义后，我们可以像使用其它接口一样使用这个函数类型的接口。
下例展示了如何创建一个函数类型的变量，并将一个同类型的函数赋值给这个变量。

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
  let result = source.search(subString);
  return result > -1;
}
```

对于函数类型的类型检查来说，函数的参数名不需要与接口里定义的名字相匹配。
比如，我们使用下面的代码重写上面的例子：

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
  let result = src.search(sub);
  return result > -1;
}
```

函数的参数会逐个进行检查，要求对应位置上的参数类型是兼容的。
如果你不想指定类型，TypeScript的类型系统会推断出参数类型，因为函数直接赋值给了`SearchFunc`类型变量。
函数的返回值类型是通过其返回值推断出来的（此例是`false`和`true`）。
如果让这个函数返回数字或字符串，类型检查器会警告我们函数的返回值类型与`SearchFunc`接口中的定义不匹配。

```
let mySearch: SearchFunc;
mySearch = function(src, sub) {
    let result = src.search(sub);
    return result > -1;
}
```

可索引的类型

与使用接口描述函数类型差不多，我们也可以描述那些能够“通过索引得到”的类型，比如`a[10]`或`ageMap["daniel"]`。
可索引类型具有一个*索引签名*，它描述了对象索引的类型，还有相应的索引返回值类型。
让我们看一个例子：

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

上面例子里，我们定义了`StringArray`接口，它具有索引签名。
这个索引签名表示了当用`number`去索引`StringArray`时会得到`string`类型的返回值。

共有支持两种索引签名：字符串和数字。
可以同时使用两种类型的索引，但是数字索引的返回值必须是字符串索引返回值类型的子类型。
这是因为当使用`number`来索引时，JavaScript会将它转换成`string`然后再去索引对象。
也就是说用`100`（一个`number`）去索引等同于使用`"100"`（一个`string`）去索引，因此两者需要保持一致。

```
class Animal {
    name: string;
}
class Dog extends Animal {
    breed: string;
}

// Error: indexing with a 'string' will sometimes get you a Dog!
interface NotOkay {
    [x: number]: Animal;
    [x: string]: Dog;
}
```

字符串索引签名能够很好的描述`dictionary`模式，并且它们也会确保所有属性与其返回值类型相匹配。
因为字符串索引声明了`obj.property`和`obj["property"]`两种形式都可以。
下面的例子里，`name`的类型与字符串索引类型不匹配，所以类型检查器给出一个错误提示：

```
interface NumberDictionary {
  [index: string]: number;
  length: number;    // 可以，length是number类型
  name: string       // 错误，`name`的类型不是索引类型的子类型
}
```

最后，你可以将索引签名设置为只读，这样就防止了给索引赋值：

```
interface ReadonlyStringArray {
    readonly [index: number]: string;
}
let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!
```

你不能设置`myArray[2]`，因为索引签名是只读的。

类类型

实现接口

与C#或Java里接口的基本作用一样，TypeScript也能够用它来明确的强制一个类去符合某种契约。

```
interface ClockInterface {
```

```
    currentTime: Date;
}

class Clock implements ClockInterface {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

你也可以在接口中描述一个方法，在类里实现它，如同下面的`setTime`方法一样：

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

接口描述了类的公共部分，而不是公共和私有两部分。
它不会帮你检查类是否具有某些私有成员。

类静态部分与实例部分的区别

当你操作类和接口的时候，你要知道类是具有两个类型的：静态部分的类型和实例的类型。
你会注意到，当你用构造器签名去定义一个接口并试图定义一个类去实现这个接口时会得到一个错误：

```
interface ClockConstructor {
    new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

这里因为当一个类实现了一个接口时，只对其实例部分进行类型检查。
constructor存在于类的静态部分，所以不在检查的范围内。

因此，我们应该直接操作类的静态部分。
看下面的例子，我们定义了两个接口，`ClockConstructor`为构造函数所用和`ClockInterface`为实例方法所用。
为了方便我们定义一个构造函数`createClock`，它用传入的类型创建实例。

```
interface ClockConstructor {
    new (hour: number, minute: number): ClockInterface;
}
interface ClockInterface {
    tick();
}
```

```
function createClock(ctor: ClockConstructor, hour: number, minute: number): Clo
    return new ctor(hour, minute);
}

class DigitalClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("beep beep");
    }
}
class AnalogClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("tick tock");
    }
}

let digital = createClock(DigitalClock, 12, 17);
let analog = createClock(AnalogClock, 7, 32);
```

因为`createClock`的第一个参数是`ClockConstructor`类型，在`createClock(AnalogClock, 7, 32)`里，会检查`AnalogClock`是否符合构造函数签名。

扩展接口

和类一样，接口也可以相互扩展。
这让我们能够从一个接口里复制成员到另一个接口里，可以更灵活地将接口分割到可重用的模块里。

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

一个接口可以继承多个接口，创建出多个接口的合成接口。

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

let square = <Square>{};
```

```
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

混合类型

先前我们提过，接口能够描述JavaScript里丰富的类型。
因为JavaScript其动态灵活的特点，有时你会希望一个对象可以同时具有上面提到的多种类型。

一个例子就是，一个对象可以同时做为函数和对象使用，并带有额外的属性。

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) { };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

在使用JavaScript第三方库的时候，你可能需要像上面那样去完整地定义类型。

接口继承类

当接口继承了一个类类型时，它会继承类的成员但不包括其实现。
就好像接口声明了所有类中存在的成员，但并没有提供具体实现一样。
接口同样会继承到类的private和protected成员。
这意味着当你创建了一个接口继承了一个拥有私有或受保护的成员的类时，这个接口类型只能被这个类或其子类所实现（implement）。

当你有一个庞大的继承结构时这很有用，但要指出的是你的代码只在子类拥有特定属性时起作用。
这个子类除了继承至基类外与基类没有任何关系。
例：

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}
```

```
class Button extends Control {
    select() { }
}

class TextBox extends Control {
    select() { }
}

class Image {
    select() { }
}

class Location {
    select() { }
}
```

在上面的例子里，SelectableControl包含了Control的所有成员，包括私有成员state。因为state是私有成员，所以只能够是Control的子类们才能实现SelectableControl接口。
因为只有Control的子类才能够拥有一个声明于Control的私有成员state，这对私有成员的兼容性是必需的。

在Control类内部，是允许通过SelectableControl的实例来访问私有成员state的。
实际上，SelectableControl就像Control一样，并拥有一个select方法。
Button和TextBox类是SelectableControl的子类（因为它们都继承自Control并有select方法），但Image和Location类并不是这样的。