* **[Quick start](http://www.typescriptlang.org/docs/tutorial.html)**
* [Tutorials](http://www.typescriptlang.org/docs/tutorial.html#toc-tutorials)
* [What's New](http://www.typescriptlang.org/docs/tutorial.html#toc-whats-new)
* [Handbook](http://www.typescriptlang.org/docs/tutorial.html#toc-handbook)
* [Declaration Files](http://www.typescriptlang.org/docs/tutorial.html#toc-declaration-files)
* [Project Configuration](http://www.typescriptlang.org/docs/tutorial.html#toc-project-config)

Quick start

Let’s get started by building a simple web application with TypeScript.

**Installing TypeScript**

There are two main ways to get the TypeScript tools:

* Via npm (the Node.js package manager)
* By installing TypeScript’s Visual Studio plugins

Visual Studio 2015 and Visual Studio 2013 Update 2 include TypeScript by default. If you didn’t install TypeScript with Visual Studio, you can still [download it](http://www.typescriptlang.org/#download-links).

For NPM users:

npm install -g typescript

安装完可以执行命令验证是否安装成功：

tsc -v 查看typescript版本

**Building your first TypeScript file**

In your editor, type the following JavaScript code in greeter.ts:

**function** **greeter**(person) {

**return** "Hello, " + person;

}

**var** user = "Jane User";

document.body.innerHTML = greeter(user);

**Compiling your code**

We used a .ts extension, but this code is just JavaScript. You could have copy/pasted this straight out of an existing JavaScript app（你也可以直接拷贝js代码）.

At the command line, run the TypeScript compiler:

tsc greeter.ts

The result will be a file greeter.js which contains the same JavaScript that you fed in（生成一个新的文件，扩展名为js）. We’re up and running using TypeScript in our JavaScript app!

Now we can start taking advantage of some of the new tools TypeScript offers. Add a : string type annotation to the ‘person’ function argument as shown here:

**function** **greeter**(person: string) {

**return** "Hello, " + person;

}

**var** user = "Jane User";

document.body.innerHTML = greeter(user);

**Type annotations**

Type annotations（类型定义） in TypeScript are lightweight ways to record the intended contract of the function or variable. In this case, we intend the greeter function to be called with a single string parameter. We can try changing the call greeter to pass an array instead:

**function** **greeter**(person: string) {

**return** "Hello, " + person;

}

**var** user = [0, 1, 2];

document.body.innerHTML = greeter(user);

Re-compiling, you’ll now see an error:

greeter.ts(7,26): Supplied parameters do not match any signature of call target

Similarly, try removing all the arguments to the greeter call. TypeScript will let you know that you have called this function with an unexpected number of parameters. In both cases, TypeScript can offer static analysis（静态分析） based on both the structure of your code, and the type annotations you provide.

Notice that although there were errors, the greeter.js file is still created（尽管出错了，但是js文件还是被创建了）. You can use TypeScript even if there are errors in your code（当编译报错时生成的js文件可能仍然可以执行）. But in this case, TypeScript is warning that your code will likely not run as expected.

**Interfaces**

Let’s develop our sample further. Here we use an interface that describes objects that have a firstName and lastName field. In TypeScript, two types are compatible（兼容） if their internal structure is compatible. This allows us to implement an interface（实现一个接口） just by having the shape the interface requires, without an explicit implements clause（在typescript中，如果两种类型的内部结构兼容，那么他们就是兼容的。这使得我们只需要构造接口结构所需要的而不用显示的声明implements来实现一个接口）.

**interface** Person {

firstName: string;

lastName: string;

}

**function** **greeter**(person: Person) {

**return** "Hello, " + person.firstName + " " + person.lastName;

}

**var** user = { firstName: "Jane", lastName: "User" };

document.body.innerHTML = greeter(user);

**Classes**

对象与接口的内部结构兼容，因此这两种类型就兼容，从而此对象就可以作为参数

Finally, let’s extend the example one last time with classes. TypeScript supports new features in JavaScript, like support for class-based object-oriented programming（基于类的面向对象编程）.

Here we’re going to create a Student class with a constructor（构造器 ） and a few public fields. Notice that classes and interfaces play well together, letting the programmer decide on the right level of abstraction.

Also of note, the use of public on arguments to the constructor is a shorthand（简写） that allows us to automatically create properties with that name（使用了public就可以将该变量名做为类的属性了，也就不用像fullName显示声明）.

**class** Student {

fullName: string;

**constructor**(**public** firstName, **public** middleInitial, **public** lastName) {

**this**.fullName = firstName + " " + middleInitial + " " + lastName;

}

}

**interface** Person {

firstName: string;

lastName: string;

}

**function** **greeter**(person : Person) {

**return** "Hello, " + person.firstName + " " + person.lastName;

}

**var** user = **new** Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);

Re-run tsc greeter.ts and you’ll see the generated JavaScript is the same as the earlier code. Classes in TypeScript are just a shorthand（简写） for the same prototype-based OO（基于00的原型） that is frequently used in JavaScript（typescript中的class就是js中的原型的使用的简写）.

**Running your TypeScript web app**

Now type the following in greeter.html:

<!DOCTYPE html>

<**html**>

<**head**><**title**>TypeScript Greeter</**title**></**head**>

<**body**>

<**script** src="greeter.js"></**script**>

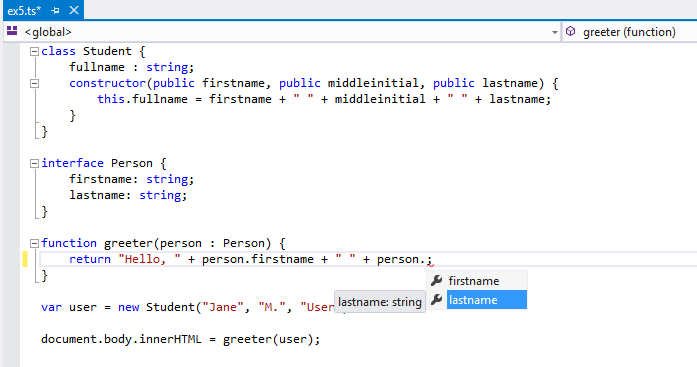
</**body**>

</**html**>

Open greeter.html in the browser to run your first simple TypeScript web application!

Optional: Open greeter.ts in Visual Studio, or copy the code into the TypeScript playground（也可以在vscode中打开）. You can hover over identifiers to see their types（鼠标悬停查看类型）. Notice that in some cases these types are inferred automatically for you. Re-type the last line, and see completion lists and parameter help based on the types of the DOM elements. Put your cursor on the reference to the greeter function, and hit F12 to go to its definition. Notice, too, that you can right-click on a symbol and use refactoring to rename it（反射）.

The type information provided works together with the tools to work with JavaScript at application scale. For more examples of what’s possible in TypeScript, see the Samples section of the website.



# HandBOOK

# Basic Types

# Introduction

For programs to be useful, we need to be able to work with some of the simplest units of data: numbers, strings, structures, boolean values, and the like. In TypeScript, we support much the same types as you would expect in JavaScript, with a convenient enumeration type（枚举类型） thrown in to help things along（加入了枚举类型）.

# Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

**let** isDone: boolean = false;

# Number

As in JavaScript, all numbers in TypeScript are floating point values（正如js，在typescript中的所有数字类型都是浮点值）. These floating point numbers get the type number. In addition to hexadecimal and decimal literals（除了16进制和十进制外）, TypeScript also supports binary and octal literals introduced in ECMAScript 2015（typescript也与es2015那样引入了二进制和8进制）.

**let** decimal: number = 6;

**let** hex: number = 0xf00d;

**let** binary: number = 0b1010;

**let** octal: number = 0o744;

# String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data（文本数据）. As in other languages, we use the type string to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data（单引号和双引号都可以修饰字符串）.

**let** color: string = "blue";

color = 'red';

You can also use template strings, which can span multiple lines and have embedded expressions（也可以使用字符串模板，模板中的字符串可以换行并嵌入表达式）. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }（嵌入的表达式的形式）.

**let** fullName: string = `Bob Bobbington`;

**let** age: number = 37;

**let** sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`

This is equivalent to declaring sentence like so:

**let** sentence: string = "Hello, my name is " + fullName + ".\n\n" +

"I'll be " + (age + 1) + " years old next month."

# Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

**let** list: number[] = [1, 2, 3];

The second way uses a generic array type, Array<elemType>:

**let** list: Array<number> = [1, 2, 3];

# Tuple

Tuple（元组） types allow you to express an array where the type of a fixed number of elements is known, but need not be the same（元组，但是元素类型可以不一致，与数组不同，数组的元素类型必须一致）. For example, you may want to represent a value as a pair of a string and a number:

// Declare a tuple type

**let** x: [string, number];

// Initialize it

x = ["hello", 10]; // OK

// Initialize it incorrectly

x = [10, "hello"]; // Error

When accessing an element with a known index, the correct type is retrieved:

console.log(x[0].substr(1)); // OK

console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'

When accessing an element outside the set of known indices, a union type is used instead（当调用的索引超过元组的下标时，类型只能是已定义的类型之一，不能是其他类型）:

x[3] = "world"; // OK, 'string' can be assigned to 'string | number'

console.log(x[5].toString()); // OK, 'string' and 'number' both have 'toString'

x[6] = true; // Error, 'boolean' isn't 'string | number'

Union types are an advanced topic that we’ll cover in a later chapter.

# Enum

A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

**enum** Color {Red, Green, Blue};

**let** c: Color = Color.Green;

By default, enums begin numbering their members starting at 0（默认索引起始值为0）. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

**enum** Color {Red = 1, Green, Blue};

**let** c: Color = Color.Green;

Or, even manually set all the values in the enum:

**enum** Color {Red = 1, Green = 2, Blue = 4};

**let** c: Color = Color.Green;

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren’t sure what that mapped to in the Color enum above, we could look up the corresponding name:

**enum** Color {Red = 1, Green, Blue};

**let** colorName: string = Color[2];

alert(colorName);

# Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks（在这些情况下，我们想有选择的选择类型并且让他们通过编译时的检查从而通过编译）. To do so, we label these with the any type:

**let** notSure: any = 4;

notSure = "maybe a string instead";

notSure = false; // okay, definitely a boolean

The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect Object to play a similar role, as it does in other languages. But variables of type Object only allow you to assign any value to them （但是object类型的变量只能被赋值为any类型的值）- you can’t call arbitrary methods（断言方法） on them, even ones that actually exist（any和object的不同）:

**let** notSure: any = 4;

notSure.ifItExists(); // okay, ifItExists might exist at runtime

notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

**let** prettySure: Object = 4;

prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.

The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

**let** list: any[] = [1, true, "free"];

list[1] = 100;

# Void

void is a little like the opposite（相反） of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

**function** **warnUser**(): **void** {

alert("This is my warning message");

}

Declaring variables of type void is not useful because you can only assign undefined or null to them:

**let** unusable: void = undefined;

# Null and Undefined

In TypeScript, both undefined and null actually have their own types named undefined and null respectively. Much like void, they’re not extremely useful on their own:

// Not much else we can assign to these variables!

**let** u: undefined = undefined;

**let** n: null = null;

By default null and undefined are subtypes of all other types（默认情况下null和undefined是其他类型的子类型，即其他类型变量都还可以赋值为undefined或者null）. That means you can assign null and undefinedto something like number.

However, when using the --strictNullChecks flag, null and undefined are only assignable to void and their respective types（严格代码审查下）. This helps avoid many common errors. In cases where you want to pass in either a string or null or undefined, you can use the union type string | null | undefined（联合类型）. Once again, more on union types later on.

As a note: we encourage the use of --strictNullChecks when possible, but for the purposes of this handbook, we will assume it is turned off.

# Never

The never type represents the type of values that never occur（不会发生）. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to,never (except never itself). Even any isn’t assignable to never.

Some examples of functions returning never:

// Function returning never must have unreachable end point

**function** **error**(message: string): **never** {

**throw** **new** Error(message);

}

// Inferred return type is never

**function** **fail**() {

**return** error("Something failed");

}

// Function returning never must have unreachable end point

**function** **infiniteLoop**(): **never** {

**while** (true) {

}

}

# Type assertions

Sometimes you’ll end up in a situation where you’ll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.

Type assertions （类型断言）are a way to tell the compiler “trust me, I know what I’m doing.” A type assertion is like a type cast（类型转换） in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler（不会影响运行时间，只会被编译器使用）. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms. One is the “angle-bracket” syntax（尖括号语法）:

**let** someValue: any = "this is a string";

**let** strLength: number = (<**string**>someValue).length;

And the other is the as-syntax:

**let** someValue: any = "this is a string";

**let** strLength: number = (someValue as string).length;

The two samples are equivalent. Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only as-style assertions are allowed.

# A note about let

You may’ve noticed that so far, we’ve been using the let keyword instead of JavaScript’s var keyword which you might be more familiar with. The let keyword is actually a newer JavaScript construct that TypeScript makes available. We’ll discuss the details later, but many common problems in JavaScript are alleviated by using let, so you should use it instead of var whenever possible.

Variable Declarations

Variable Declarations

let and const are two relatively new types of variable declarations in JavaScript. As we mentioned earlier, let is similar to var in some respects, but allows users to avoid some of the common “gotchas” that users run into in JavaScript. const is an augmentation of let （const是let的一个参数）in that it prevents re-assignment to a variable.

With TypeScript being a superset（超集） of JavaScript, the language naturally supports let and const. Here we’ll elaborate（阐明） more on these new declarations and why they’re preferable to var.

If you’ve used JavaScript offhandedly, the next section might be a good way to refresh your memory. If you’re intimately familiar with all the quirks（奇事） of var declarations in JavaScript, you might find it easier to skip ahead.

var declarations

Declaring a variable in JavaScript has always traditionally been done with the var keyword.

**var** a = 10;

As you might’ve figured out, we just declared a variable named a with the value 10.

We can also declare a variable inside of a function:

**function** **f**() {

**var** message = "Hello, world!";

**return** message;

}

and we can also access those same variables within other functions:

**function** **f**() {

**var** a = 10;

**return** **function** **g**() {

**var** b = a + 1;

**return** b;

}

}

**var** g = f();

g(); // returns '11'

In this above example, g captured the variable a declared in f. At any point that g gets called, the value of a will be tied to the value of a in f. Even if g is called once f is done running, it will be able to access and modify a.

**function** **f**() {

**var** a = 1;

a = 2;

**var** b = g();

a = 3;

**return** b;

**function** **g**() {

**return** a;

}

}

f(); // returns '2'

**Scoping rules（作用域规则）**

var declarations have some odd（奇怪的） scoping rules for those used to other languages. Take the following example:

**function** **f**(shouldInitialize: boolean) {

**if** (shouldInitialize) {

**var** x = 10;

}

**return** x;

}

f(true); // returns '10'

f(false); // returns 'undefined'

Some readers might do a double-take at this example. The variable x was declared *within the if block*, and yet we were able to access it from outside that block. That’s because var declarations are accessible anywhere within their containing function, module, namespace, or global scope - all which we’ll go over later on - regardless of the containing block. Some people call this *var-scoping* or *function-scoping*. Parameters are also function scoped.

These scoping rules can cause several types of mistakes. One problem they exacerbate is the fact that it is not an error to declare the same variable multiple times:

**function** **sumMatrix**(matrix: number[][]) {

**var** sum = 0;

**for** (**var** i = 0; i < matrix.length; i++) {

**var** currentRow = matrix[i];

**for** (**var** i = 0; i < currentRow.length; i++) {

sum += currentRow[i];

}

}

**return** sum;

}

Maybe it was easy to spot out for some, but the inner for-loop will accidentally overwrite the variable I because i refers to the same function-scoped variable. As experienced developers know by now, similar sorts of bugs slip through code reviews and can be an endless source of frustration.

**Variable capturing quirks**

Take a quick second to guess what the output of the following snippet is:

**for** (**var** i = 0; i < 10; i++) {

setTimeout(**function**() { console.log(i); }, 100 \* i);

}

For those unfamiliar, setTimeout will try to execute a function after a certain number of milliseconds (though waiting for anything else to stop running).

Ready? Take a look:

10

10

10

10

10

10

10

10

10

10

Many JavaScript developers are intimately familiar with this behavior, but if you’re surprised, you’re certainly not alone. Most people expect the output to be

0

1

2

3

4

5

6

7

8

9

Remember what we mentioned earlier about variable capturing? Every function expression we pass to setTimeout actually refers to the same i from the same scope.

Let’s take a minute to consider that means. setTimeout will run a function after some number of milliseconds, *but only* after the for loop has stopped executing;

By the time the for loop has stopped executing, the value of i is 10. So each time the given function gets called, it will print out 10!

A common work around is to use an IIFE - an Immediately Invoked Function Expression - to capture i at each iteration:

**for** (**var** i = 0; i < 10; i++) {

// capture the current state of 'i'

// by invoking a function with its current value

(**function**(i) {

setTimeout(**function**() { console.log(i); }, 100 \* i);

})(i);

}

This odd-looking pattern is actually pretty common. The i in the parameter list actually shadows the I declared in the for loop, but since we named them the same, we didn’t have to modify the loop body too much.

let declarations

By now you’ve figured out that var has some problems, which is precisely why let statements were introduced. Apart from the keyword used, let statements are written the same way var statements are（除了关键词的区别，let的语句与var语句一样）.

**let** hello = "Hello!";

The key difference is not in the syntax, but in the semantics（不同的不是语法，而是语境）, which we’ll now dive into.

**Block-scoping（块作用域）**

When a variable is declared using let, it uses what some call *lexical-scoping* or *block-scoping（块作用域）*. Unlike variables declared with var whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or for-loop.

**function** **f**(input: boolean) {

**let** a = 100;

**if** (input) {

// Still okay to reference 'a'

**let** b = a + 1;

**return** b;

}

// Error: 'b' doesn't exist here

**return** b;

}

Here, we have two local variables a and b. a’s scope is limited to the body of f while b’s scope is limited to the containing if statement’s block.

Variables declared in a catch clause also have similar scoping rules.

**try** {

**throw** "oh no!";

}

**catch** (e) {

console.log("Oh well.");

}

// Error: 'e' doesn't exist here

console.log(e);

Another property of block-scoped variables is that they can’t be read or written to before they’re actually declared. While these variables are “present” throughout their scope, all points up until their declaration are part of their *temporal dead zone（尽管变量的作用域是整个块，但是不能在定义他们之前使用）*. This is just a sophisticated way of saying you can’t access them before the let statement, and luckily TypeScript will let you know that.

a++; // illegal to use 'a' before it's declared;

**let** a;

Something to note is that you can still *capture* a block-scoped variable before it’s declared. The only catch is that it’s illegal to call that function before the declaration. If targeting ES2015, a modern runtime will throw an error; however, right now TypeScript is permissive and won’t report this as an error.

**function** **foo**() {

// okay to capture 'a'

**return** a;

}

// illegal call 'foo' before 'a' is declared

// runtimes should throw an error here

foo();

**let** a;

For more information on temporal dead zones, see relevant content on the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#Temporal_dead_zone_and_errors_with_let).

**Re-declarations and Shadowing**

With var declarations, we mentioned that it didn’t matter how many times you declared your variables; you just got one.

**function** **f**(x) {

**var** x;

**var** x;

**if** (true) {

**var** x;

}

}

In the above example, all declarations of x actually refer to the *same* x, and this is perfectly valid. This often ends up being a source of bugs. Thankfully, let declarations are not as forgiving.

**let** x = 10;

**let** x = 20; // error: can't re-declare 'x' in the same scope

The variables don’t necessarily need to both be block-scoped for TypeScript to tell us that there’s a problem.

**function** **f**(x) {

**let** x = 100; // error: interferes with parameter declaration

}

**function** **g**() {

**let** x = 100;

**var** x = 100; // error: can't have both declarations of 'x'

}

That’s not to say that block-scoped variable can never be declared with a function-scoped variable. The block-scoped variable just needs to be declared within a distinctly different block.

**function** **f**(condition, x) {

**if** (condition) {

**let** x = 100;

**return** x;

}

**return** x;

}

f(false, 0); // returns '0'

f(true, 0); // returns '100'

The act of introducing a new name in a more nested scope is called *shadowing*. It is a bit of a double-edged sword（双刃剑） in that it can introduce certain bugs on its own in the event of accidental shadowing, while also preventing certain bugs. For instance, imagine we had written our earlier sumMatrix function using let variables.

**function** **sumMatrix**(matrix: number[][]) {

**let** sum = 0;

**for** (**let** i = 0; i < matrix.length; i++) {

**var** currentRow = matrix[i];

**for** (**let** i = 0; i < currentRow.length; i++) {

sum += currentRow[i];

}

}

**return** sum;

}

This version of the loop will actually perform the summation correctly because the inner loop’s i shadows i from the outer loop.

Shadowing should *usually* be avoided in the interest of writing clearer code. While there are some scenarios（情景） where it may be fitting to take advantage of it, you should use your best judgement.

**Block-scoped variable capturing**

When we first touched on the idea of variable capturing（变量捕获） with var declaration, we briefly went into how variables act once captured. To give a better intuition of this, each time a scope is run, it creates an “environment” of variables（每一次运行，它都会创建一个变量的环境）. That environment and its captured variables can exist even after everything within its scope has finished executing（即使作用域中的代码都执行完，环境和它捕获的变量也会存在）.

**function** **theCityThatAlwaysSleeps**() {

**let** getCity;

**if** (true) {

**let** city = "Seattle";

getCity = **function**() {

**return** city;

}

}

**return** getCity();

}

Because we’ve captured city from within its environment, we’re still able to access it despite the fact that the if block finished executing.

Recall that with our earlier setTimeout example, we ended up needing to use an IIFE to capture the state of a variable for every iteration of the for loop. In effect, what we were doing was creating a new variable environment（环境作用域：function-scope） for our captured variables. That was a bit of a pain, but luckily, you’ll never have to do that again in TypeScript.

let declarations have drastically different behavior when declared as part of a loop. Rather than just introducing a new environment to the loop itself, these declarations sort of create a new scope *per iteration*. Since this is what we were doing anyway with our IIFE, we can change our old setTimeout example to just use a let declaration.

**for** (**let** i = 0; i < 10 ; i++) {

setTimeout(**function**() { console.log(i); }, 100 \* i);

}

and as expected, this will print out

0

1

2

3

4

5

6

7

8

9

const declarations

const declarations are another way of declaring variables.

**const** numLivesForCat = 9;

They are like let declarations but, as their name implies, their value cannot be changed once they are bound（它类似let，但是正如没名字所暗示的那样，定义后变量就绑定了值，而且不能改变）. In other words, they have the same scoping rules as let, but you can’t re-assign to them（它与let有同样的作用域规则而且不能改变值）.

This should not be confused with the idea that the values they refer to are *immutable（不可改变的）*.

**const** numLivesForCat = 9;

**const** kitty = {

name: "Aurora",

numLives: numLivesForCat,

}

// Error

kitty = {

name: "Danielle",

numLives: numLivesForCat

};

// all "okay"

kitty.name = "Rory";

kitty.name = "Kitty";

kitty.name = "Cat";

kitty.numLives--;

Unless you take specific measures to avoid it, the internal state of a const variable is still modifiable（const变量内部的仍然是可变的）. Fortunately, TypeScript allows you to specify that members of an object are readonly. The [chapter on Interfaces](http://www.typescriptlang.org/docs/handbook/interfaces.html) has the details.

let vs. const

Given that we have two types of declarations with similar scoping semantics（语义，词义）, it’s natural to find ourselves asking which one to use. Like most broad questions, the answer is: it depends.

Applying the [principle of least privilege](https://en.wikipedia.org/wiki/Principle_of_least_privilege), all declarations other than those you plan to modify should use const. The rationale is that if a variable didn’t need to get written to, others working on the same codebase shouldn’t automatically be able to write to the object, and will need to consider whether they really need to reassign to the variable. Using const also makes code more predictable when reasoning about flow of data.

On the other hand, let is not any longer to write out than var, and many users will prefer its brevity. The majority of this handbook uses let declarations in that interest.

Use your best judgement, and if applicable, consult the matter with the rest of your team.

Destructuring（解构）

Another ECMAScript 2015 feature that TypeScript has is destructuring. For a complete reference, see [the article on the Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment). In this section, we’ll give a short overview.

**Array destructuring（数组解构）**

The simplest form of destructuring is array destructuring assignment:

**let** input = [1, 2];

**let** [first, second] = input;

console.log(first); // outputs 1

console.log(second); // outputs 2

This creates two new variables named first and second. This is equivalent to using indexing, but is much more convenient:

first = input[0];

second = input[1];

Destructuring works with already-declared variables as well:

// swap variables

[first, second] = [second, first];

And with parameters to a function:

**function** **f**([first, second]: [number, number]) {

console.log(first);

console.log(second);

}

f(input);

You can create a variable for the remaining items in a list using the syntax ...:

**let** [first, ...rest] = [1, 2, 3, 4];

console.log(first); // outputs 1

console.log(rest); // outputs [ 2, 3, 4 ]

Of course, since this is JavaScript, you can just ignore trailing elements you don’t care about:

**let** [first] = [1, 2, 3, 4];

console.log(first); // outputs 1

Or other elements:

**let** [, second, , fourth] = [1, 2, 3, 4];

**Object destructuring（对象解构）**

You can also destructure objects:

**let** o = {

a: "foo",

b: 12,

c: "bar"

}

**let** { a, b } = o;

This creates new variables a and b from o.a and o.b. Notice that you can skip c if you don’t need it.

Like array destructuring, you can have assignment without declaration:

({ a, b } = { a: "baz", b: 101 });

Notice that we had to surround this statement with parentheses（圆括号）. JavaScript normally parses a { as the start of block.

You can create a variable for the remaining items in an object using the syntax ...:

**let** { a, ...passthrough } = o;

**let** total = passthrough.b + passthrough.c.length;

***Property renaming（属性保留）***

You can also give different names to properties:

**let** { a: newName1, b: newName2 } = o;

Here the syntax starts to get confusing. You can read a: newName1 as “a as newName1”. The direction is left-to-right, as if you had written:

**let** newName1 = o.a;

**let** newName2 = o.b;

Confusingly, the colon（冒号） here does *not* indicate the type. The type, if you specify it, still needs to be written after the entire destructuring:

**let** { a, b }: { a: string, b: number } = o;

***Default values***

Default values let you specify a default value in case a property is undefined:

**function** **keepWholeObject**(wholeObject: { a: string, b?: number }) {

**let** { a, b = 1001 } = wholeObject;

}

keepWholeObject now has a variable for wholeObject as well as the properties a and b, even if b is undefined.

**Function declarations**

Destructuring also works in function declarations. For simple cases this is straightforward:

**type** C = { a: string, b?: number }

**function** **f**({ a, b }: C): **void** {

// ...

}

But specifying defaults is more common for parameters, and getting defaults right with destructuring can be tricky. First of all, you need to remember to put the type before the default value.

**function** **f**({ a, b } = { a: "", b: 0 }): **void** {

// ...

}

f(); // ok, default to { a: "", b: 0 }

Then, you need to remember to give a default for optional properties on the destructured property instead of the main initializer. Remember that C was defined with b optional:

**function** **f**({ a, b = 0 } = { a: "" }): **void** {

// ...

}

f({ a: "yes" }) // ok, default b = 0

f() // ok, default to { a: "" }, which then defaults b = 0

f({}) // error, 'a' is required if you supply an argument

Use destructuring with care. As the previous example demonstrates, anything but the simplest destructuring expression is confusing. This is especially true with deeply nested destructuring, which gets *really* hard to understand even without piling on renaming, default values, and type annotations. Try to keep destructuring expressions small and simple. You can always write the assignments that destructuring would generate yourself.

**Spread（扩展）**

The spread operator is the opposite of destructuring. It allows you to spread an array into another array, or an object into another object. For example:

**let** first = [1, 2];

**let** second = [3, 4];

**let** bothPlus = [0, ...first, ...second, 5];

This gives bothPlus the value [0, 1, 2, 3, 4, 5]. Spreading creates a shallow copy（浅拷贝） of first and second. They are not changed by the spread.

You can also spread objects:

**let** defaults = { food: "spicy", price: "$$", ambiance: "noisy" };

**let** search = { ...defaults, food: "rich" };

Now search is { food: "rich", price: "$$", ambiance: "noisy" }. Object spreading is more complex than array spreading. Like array spreading, it proceeds from left-to-right, but the result is still an object. This means that properties that come later in the spread object overwrite properties that come earlier. So if we modify the previous example to spread at the end:

**let** defaults = { food: "spicy", price: "$$", ambiance: "noisy" };

**let** search = { food: "rich", ...defaults };

Then the food property in defaults overwrites food: "rich", which is not what we want in this case.

Object spread also has a couple of other surprising limits. First, it only includes own, enumerable properties（它只包含自己的属性，不包含方法）. Basically, that means you lose methods when you spread instances of an object:

**class** C {

p = 12;

m() {

}

}

**let** c = **new** C();

**let** clone = { ...c };

clone.p; // ok

clone.m(); // error!

Second, the Typescript compiler doesn’t allow spreads of type parameters from generic functions. That feature is expected in future versions of the language.

Interfaces（接口）

Introduction

One of TypeScript’s core principles is that type-checking focuses on the *shape（形状，结构）* that values have（即typescript做类型检查时只关心数据值的结构而不是其他，如具有同样的字段的json和object被看做是兼容的）. This is sometimes called “duck typing” or “structural subtyping”（结构上的子类型）. In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Our First Interface

The easiest way to see how interfaces work is to start with a simple example:

**function** **printLabel**(labelledObj: { label: string }) {

console.log(labelledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

The type-checker checks the call to printLabel. The printLabel function has a single parameter that requires that the object passed in has a property called label of type string. Notice that our object actually has more properties than this, but the compiler only checks that *at least* the ones required are present and match the types required. There are some cases where TypeScript isn’t as lenient, which we’ll cover in a bit.

We can write the same example again, this time using an interface to describe the requirement of having the label property that is a string:

**interface** LabelledValue {

label: string;

}

**function** **printLabel**(labelledObj: LabelledValue) {

console.log(labelledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

The interface LabelledValue is a name we can now use to describe the requirement in the previous example. It still represents having a single property called label that is of type string. Notice we didn’t have to explicitly say that the object we pass to printLabel implements this interface like we might have to in other languages. Here, it’s only the shape that matters. If the object we pass to the function meets the requirements listed, then it’s allowed.

It’s worth pointing out that the type-checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like “option bags” where you pass an object to a function that only has a couple of properties filled in.

Here’s an example of this pattern:

**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"});

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a ? at the end of the property name in the declaration.

The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface. For example, had we mistyped the name of the color property in createSquare, we would get an error message letting us know:

**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 'color' does not exist on type 'SquareConfig'

newSquare.color = config.color;

}

**if** (config.width) {

newSquare.area = config.width \* config.width;

}

**return** newSquare;

}

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

Readonly properties

Some properties should only be modifiable when an object is first created. You can specify this by puttingreadonly before the name of the property:

**interface** Point {

readonly x: number;

readonly y: number;

}

You can construct a Point by assigning an object literal. After the assignment, x and y can’t be changed.

**let** p1: Point = { x: 10, y: 20 };

p1.x = 5; // error!

TypeScript comes with a ReadonlyArray<T> type that is the same as Array<T> with all mutating methods removed, so you can make sure you don’t change your arrays after creation:

**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!

On the last line of the snippet you can see that even assigning the entire ReadonlyArray back to a normal array is illegal. You can still override it with a type assertion, though:

a = ro as number[];

**readonly vs const**

The easiest way to remember whether to use readonly or const is to ask whether you’re using it on a variable or a property. Variables use const whereas properties use readonly.

Excess Property Checks

In our first example using interfaces, TypeScript let us pass { size: number; label: string; } to something that only expected a { label: string; }. We also just learned about optional properties, and how they’re useful when describing so-called “option bags”.

However, combining the two naively would let you to shoot yourself in the foot the same way you might in JavaScript. For example, taking our last example using createSquare:

**interface** SquareConfig {

color?: string;

width?: number;

}

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

// ...

}

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

Notice the given argument to createSquare is spelled *colour* instead of color. In plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the width properties are compatible, there’s nocolor property present, and the extra colour property is insignificant.

However, TypeScript takes the stance that there’s probably a bug in this code. Object literals get special treatment and undergo *excess property checking* when assigning them to other variables, or passing them as arguments. If an object literal has any properties that the “target type” doesn’t have, you’ll get an error.

// error: 'colour' not expected in type 'SquareConfig'

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

Getting around these checks is actually really simple. The easiest method is to just use a type assertion:

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

However, a better approach might be to add a string index signature if you’re sure that the object can have some extra properties that are used in some special way. If SquareConfigs can have color and widthproperties with the above types, but could *also* have any number of other properties, then we could define it like so:

**interface** SquareConfig {

color?: string;

width?: number;

[propName: string]: any;

}

We’ll discuss index signatures in a bit, but here we’re saying a SquareConfig can have any number of properties, and as long as they aren’t color or width, their types don’t matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable: Since squareOptions won’t undergo excess property checks, the compiler won’t give you an error.

**let** squareOptions = { colour: "red", width: 100 };

**let** mySquare = createSquare(squareOptions);

Keep in mind that for simple code like above, you probably shouldn’t be trying to “get around” these checks. For more complex object literals that have methods and hold state, you might need to keep these techniques in mind, but a majority of excess property errors are actually bugs. That means if you’re running into excess property checking problems for something like option bags, you might need to revise some of your type declarations. In this instance, if it’s okay to pass an object with both a color or colourproperty to createSquare, you should fix up the definition of SquareConfig to reflect that.

Function Types

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. Each parameter in the parameter list requires both name and type.

**interface** SearchFunc {

(source: string, subString: string): boolean;

}

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

**let** mySearch: SearchFunc;

mySearch = **function**(source: string, subString: string) {

**let** result = source.search(subString);

**if** (result == -1) {

**return** false;

}

**else** {

**return** true;

}

}

For function types to correctly type-check, the names of the parameters do not need to match. We could have, for example, written the above example like this:

**let** mySearch: SearchFunc;

mySearch = **function**(src: string, sub: string): **boolean** {

**let** result = src.search(sub);

**if** (result == -1) {

**return** false;

}

**else** {

**return** true;

}

}

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. If you do not want to specify types at all, Typescript’s contextual typing can infer the argument types since the function value is assigned directly to a variable of type SearchFunc. Here, also, the return type of our function expression is implied by the values it returns (here false andtrue). Had the function expression returned numbers or strings, the type-checker would have warned us that return type doesn’t match the return type described in the SearchFunc interface.

**let** mySearch: SearchFunc;

mySearch = **function**(src, sub) {

**let** result = src.search(sub);

**if** (result == -1) {

**return** false;

}

**else** {

**return** true;

}

}

Indexable Types

Similarly to how we can use interfaces to describe function types, we can also describe types that we can “index into” like a[10], or ageMap["daniel"]. Indexable types have an *index signature* that describes the types we can use to index into the object, along with the corresponding return types when indexing. Let’s take an example:

**interface** StringArray {

[index: number]: string;

}

**let** myArray: StringArray;

myArray = ["Bob", "Fred"];

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

Above, we have a StringArray interface that has an index signature. This index signature states that when a StringArray is indexed with a number, it will return a string.

There are two types of supported index signatures: string and number. It is possible to support both types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is because when indexing with a number, JavaScript will actually convert that to astring before indexing into an object. That means that indexing with 100 (a number) is the same thing as indexing with "100" (a string), so the two need to be consistent.

**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;

}

While string index signatures are a powerful way to describe the “dictionary” pattern, they also enforce that all properties match their return type. This is because a string index declares that obj.property is also available as obj["property"]. In the following example, name’s type does not match the string index’s type, and the type-checker gives an error:

**interface** NumberDictionary {

[index: string]: number;

length: number; // ok, length is a number

name: string; // error, the type of 'name' is not a subtype of the indexer

}

Finally, you can make index signatures readonly in order to prevent assignment to their indices:

**interface** ReadonlyStringArray {

readonly [index: number]: string;

}

**let** myArray: ReadonlyStringArray = ["Alice", "Bob"];

myArray[2] = "Mallory"; // error!

You can’t set myArray[2] because the index signature is readonly.

Class Types

**Implementing an interface**

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

**interface** ClockInterface {

currentTime: Date;

}

**class** Clock **implements** ClockInterface {

currentTime: Date;

**constructor**(h: number, m: number) { }

}

You can also describe methods in an interface that are implemented in the class, as we do with setTime in the below example:

**interface** ClockInterface {

currentTime: Date;

setTime(d: Date);

}

**class** Clock **implements** ClockInterface {

currentTime: Date;

setTime(d: Date) {

**this**.currentTime = d;

}

**constructor**(h: number, m: number) { }

}

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance.

**Difference between the static and instance sides of classes**

When working with classes and interfaces, it helps to keep in mind that a class has *two* types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a construct signature and try to create a class that implements this interface you get an error:

**interface** ClockConstructor {

**new** (hour: number, minute: number);

}

**class** Clock **implements** ClockConstructor {

currentTime: Date;

**constructor**(h: number, m: number) { }

}

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, ClockConstructor for the constructor and ClockInterface for the instance methods. Then for convenience we define a constructor function createClock that creates instances of the type that is passed to it.

**interface** ClockConstructor {

**new** (hour: number, minute: number): ClockInterface;

}

**interface** ClockInterface {

tick();

}

**function** **createClock**(ctor: ClockConstructor, hour: number, minute: number): **ClockInterface** {

**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);

Because createClock’s first parameter is of type ClockConstructor, in createClock(AnalogClock, 7, 32), it checks that AnalogClock has the correct constructor signature.

Extending Interfaces

Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.

**interface** Shape {

color: string;

}

**interface** Square extends Shape {

sideLength: number;

}

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

square.color = "blue";

square.sideLength = 10;

An interface can extend multiple interfaces, creating a combination of all of the interfaces.

**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;

Hybrid Types

As we mentioned earlier, interfaces can describe the rich types present in real world JavaScript. Because of JavaScript’s dynamic and flexible nature, you may occasionally encounter an object that works as a combination of some of the types described above.

One such example is an object that acts as both a function and an object, with additional properties:

**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;

When interacting with 3rd-party JavaScript, you may need to use patterns like the above to fully describe the shape of the type.

Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it.

This is useful when you have a large inheritance hierarchy, but want to specify that your code works with only subclasses that have certain properties. The subclasses don’t have to be related besides inheriting from the base class. For example:

**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() { }

}

In the above example, SelectableControl contains all of the members of Control, including the private state property. Since state is a private member it is only possible for descendants of Controlto implement SelectableControl. This is because only descendants of Control will have a stateprivate member that originates in the same declaration, which is a requirement for private members to be compatible.

Within the Control class it is possible to access the state private member through an instance ofSelectableControl. Effectively, a SelectableControl acts like a Control that is known to have aselect method. The Button and TextBox classes are subtypes of SelectableControl (because they both inherit from Control and have a select method), but the Image and Location classes are not.