TypeScript不是存在于真空中。
它从JavaScript生态系统和大量现存的JavaScript而来。
将JavaScript代码转换成TypeScript虽乏味却不是难事。
接下来这篇教程将教你怎么做。
在开始转换TypeScript之前，我们假设你已经理解了足够多本手册里的内容。

设置目录

如果你在写纯JavaScript，你大概是想直接运行这些JavaScript文件，
这些文件存在于`src`，`lib`或`dist`目录里，它们可以按照预想运行。

若如此，那么你写的纯JavaScript文件将做为TypeScript的输入，你将要运行的是TypeScript的输出。
在从JS到TS的转换过程中，我们会分离输入文件以防TypeScript覆盖它们。
你也可以指定输出目录。

你可能还需要对JavaScript做一些中间处理，比如合并或经过Babel再次编译。
在这种情况下，你应该已经有了如下的目录结构。

那么现在，我们假设你已经设置了这样的目录结构：

```
projectRoot
├── src
│   ├── file1.js
│   └── file2.js
├── built
└── tsconfig.json
```

如果你在`src`目录外还有`tests`文件夹，那么在`src`里可以有一个`tsconfig.json`文件，
在`tests`里还可以有一个。

书写配置文件

TypeScript使用`tsconfig.json`文件管理工程配置，例如你想包含哪些文件和进行哪些检查。
让我们先创建一个简单的工程配置文件：

```
{
    "compilerOptions": {
        "outDir": "./built",
        "allowJs": true,
        "target": "es5"
    },
    "include": [
        "./src/**/*"
    ]
}
```

这里我们为TypeScript设置了一些东西:

1. 读取所有可识别的`src`目录下的文件（通过`include`）。
2. 接受JavaScript做为输入（通过`allowJs`）。
3. 生成的所有文件放在`built`目录下（通过`outDir`）。
4. 将JavaScript代码降级到低版本比如ECMAScript 5（通过`target`）。

现在，如果你在工程根目录下运行`tsc`，就可以在`built`目录下看到生成的文件。
`built`下的文件应该与`src`下的文件相同。
现在你的工程里的TypeScript已经可以工作了。

早期收益

现在你已经可以看到TypeScript带来的好处，它能帮助我们理解当前工程。
如果你打开像VS Code或Visual Studio这样的编译器，你就能使用像自动补全这样的工具。
你还可以配置如下的选项来帮助查找BUG：

- `noImplicitReturns` 会防止你忘记在函数末尾返回值。
- `noFallthroughCasesInSwitch` 会防止在`switch`代码块里的两个`case`之间忘记添加`break`语句。

TypeScript还能发现那些执行不到的代码和标签，你可以通过设置`allowUnreachableCode`和`allowUnusedLabels`选项来禁用。

与构建工具进行集成

在你的构建管道中可能包含多个步骤。
比如为每个文件添加一些内容。
每种工具的使用方法都是不同的，我们会尽可能的包涵主流的工具。

Gulp

如果你在使用时髦的Gulp，我们已经有一篇关于使用Gulp结合TypeScript并与常见构建工具Browserify，Babelify和Uglify进行集成的教程。
请阅读这篇教程。

Webpack

Webpack集成非常简单。
你可以使用`awesome-typescript-loader`，它是一个TypeScript的加载器，结合`source-map-loader`方便调试。
运行：

```
npm install awesome-typescript-loader source-map-loader
```

并将下面的选项合并到你的`webpack.config.js`文件里：

```
module.exports = {
    entry: "./src/index.ts",
    output: {
```

```
        filename: "./dist/bundle.js",
    },

    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'a
            { test: /\.tsx?$/, loader: "awesome-typescript-loader" }
        ],

        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by
            { test: /\.js$/, loader: "source-map-loader" }
        ]
    },

    // Other options...
};
```

要注意的是，`awesome-typescript-loader`必须在其它处理`.js`文件的加载器之前运行。

这与另一个TypeScript的Webpack加载器ts-loader是一样的。
你可以到这里了解两者之间的差别。

你可以在[React和Webpack教程](./React & Webpack.md)里找到使用Webpack的例子。

转换到TypeScript文件

到目前为止，你已经做好了使用TypeScript文件的准备。
第一步，将`.js`文件重命名为`.ts`文件。
如果你使用了JSX，则重命名为`.tsx`文件。

第一步达成？
太棒了！
你已经成功地将一个文件从JavaScript转换成了TypeScript!

当然了，你可能感觉哪里不对劲儿。
如果你在支持TypeScript的编辑器（或运行`tsc --pretty`）里打开了那个文件，你可能会看到有些行上有红色的波浪线。
你可以把它们当做在Microsoft Word里看到的红色波浪线一样。
但是TypeScript仍然会编译你的代码，就好比Word还是允许你打印你的文档一样。

如果对你来说这种行为太随便了，你可以让它变得严格些。
如果，你*不想*在发生错误的时候，TypeScript还会被编译成JavaScript，你可以使用`noEmitOnError`选项。
从某种意义上来讲，TypeScript具有一个调整它的严格性的刻度盘，你可以将指针拨动到

你想要的位置。

如果你计划使用可用的高度严格的设置，最好现在就启用它们（查看启用严格检查）。
比如，如果你不想让TypeScript将没有明确指定的类型默默地推断为`any`类型，可以在修改文件之前启用`noImplicitAny`。
你可能会觉得这有些过度严格，但是长期收益很快就能显现出来。

去除错误

我们提到过，若不出所料，在转换后将会看到错误信息。
重要的是我们要逐一的查看它们并决定如何处理。
通常这些都是真正的BUG，但有时必须要告诉TypeScript你要做的是什么。

由模块导入

首先你可能会看到一些类似`Cannot find name 'require'.`和`Cannot find name 'define'.`的错误。
遇到这种情况说明你在使用模块。
你仅需要告诉TypeScript它们是存在的：

```
// For Node/CommonJS
declare function require(path: string): any;
```

或

```
// For RequireJS/AMD
declare function define(...args: any[]): any;
```

最好是避免使用这些调用而改用TypeScript的导入语法。

首先，你要使用TypeScript的`module`标记来启用一些模块系统。
可用的选项有`commonjs`，`amd`，`system`，and `umd`。

如果代码里存在下面的Node/CommonJS代码：

```
var foo = require("foo");

foo.doStuff();
```

或者下面的RequireJS/AMD代码：

```
define(["foo"], function(foo) {
    foo.doStuff();
})
```

那么可以写做下面的TypeScript代码：

```
import foo = require("foo");

foo.doStuff();
```

获取声明文件

如果你开始做转换到TypeScript导入，你可能会遇到`Cannot find module 'foo'.`这样的错误。
问题出在没有*声明文件*来描述你的代码库。
幸运的是这非常简单。
如果TypeScript报怨像是没有`lodash`包，那你只需这样做

```
npm install -S @types/lodash
```

如果你没有使用`commonjs`模块模块选项，那么就需要将`moduleResolution`选项设置为`node`。

之后，你应该就可以导入`lodash`了，并且会获得精确的自动补全功能。

由模块导出

通常来讲，由模块导出涉及添加属性到`exports`或`module.exports`。
TypeScript允许你使用顶级的导出语句。
比如，你要导出下面的函数：

```
module.exports.feedPets = function(pets) {
    // ...
}
```

那么你可以这样写：

```
export function feedPets(pets) {
    // ...
}
```

有时你会完全重写导出对象。
这是一个常见模式，这会将模块变为可立即调用的模块：

```
var express = require("express");
var app = express();
```

之前你可以是这样写的：

```
function foo() {
    // ...
}
module.exports = foo;
```

在TypeScript里，你可以使用`export =`来代替。

```
function foo() {
    // ...
}
export = foo;
```

过多或过少的参数

有时你会发现你在调用一个具有过多或过少参数的函数。
通常，这是一个BUG，但在某些情况下，你可以声明一个使用`arguments`对象的函数而不需要写出所有参数:

```
function myCoolFunction() {
    if (arguments.length == 2 && !Array.isArray(arguments[1])) {
        var f = arguments[0];
        var arr = arguments[1];
        // ...
    }
    // ...
}

myCoolFunction(function(x) { console.log(x) }, [1, 2, 3, 4]);
myCoolFunction(function(x) { console.log(x) }, 1, 2, 3, 4);
```

这种情况下，我们需要利用TypeScript的函数重载来告诉调用者`myCoolFunction`函数的调用方式。

```
function myCoolFunction(f: (x: number) => void, nums: number[]): void;
function myCoolFunction(f: (x: number) => void, ...nums: number[]): void;
function myCoolFunction() {
    if (arguments.length == 2 && !Array.isArray(arguments[1])) {
        var f = arguments[0];
        var arr = arguments[1];
        // ...
    }
    // ...
}
```

我们为`myCoolFunction`函数添加了两个重载签名。
第一个检查`myCoolFunction`函数是否接收一个函数（它又接收一个`number`参数）和一个`number`数组。
第二个同样是接收了一个函数，并且使用剩余参数（`...nums`）来表示之后的其它所有参数必须是`number`类型。

连续添加属性

有些人可能会因为代码美观性而喜欢先创建一个对象然后立即添加属性：

```
var options = {};
options.color = "red";
options.volume = 11;
```

TypeScript会提示你不能给`color`和`volumn`赋值，因为先前指定`options`的类型为`{}`并不带有任何属性。
如果你将声明变成对象字面量的形式将不会产生错误：

```
let options = {
    color: "red",
    volume: 11
};
```

你还可以定义`options`的类型并且添加类型断言到对象字面量上。

```
interface Options { color: string; volume: number }

let options = {} as Options;
options.color = "red";
options.volume = 11;
```

或者，你可以将options指定成any类型，这是最简单的，但也是获益最少的。

any，Object，和{}

你可能会试图使用Object或{}来表示一个值可以具有任意属性，因为Object是最通用的类型。
然而在这种情况下**any是真正想要使用的类型**，因为它是最*灵活*的类型。

比如，有一个Object类型的东西，你将不能够在其上调用toLowerCase()。

越普通意味着更少的利用类型，但是any比较特殊，它是最普通的类型但是允许你在上面做任何事情。
也就是说你可以在上面调用，构造它，访问它的属性等等。
记住，当你使用any时，你会失去大多数TypeScript提供的错误检查和编译器支持。

如果你还是决定使用Object和{}，你应该选择{}。
虽说它们基本一样，但是从技术角度上来讲{}在一些深奥的情况里比Object更普通。

启用严格检查

TypeScript提供了一些检查来保证安全以及帮助分析你的程序。
当你将代码转换为了TypeScript后，你可以启用这些检查来帮助你获得高度安全性。

没有隐式的any

在某些情况下TypeScript没法确定某些值的类型。
那么TypeScript会使用any类型代替。
这对代码转换来讲是不错，但是使用any意味着失去了类型安全保障，并且你得不到工具的支持。
你可以使用noImplicitAny选项，让TypeScript标记出发生这种情况的地方，并给出一个错误。

严格的null与undefined检查

默认地，TypeScript把null和undefined当做属于任何类型。
这就是说，声明为number类型的值可以为null和undefined。
因为在JavaScript和TypeScript里，null和undefined经常会导致BUG的产生，所以TypeScript包含了strictNullChecks选项来帮助我们减少对这种情况的担忧。

当启用了strictNullChecks，null和undefined获得了它们自己各自的类型null和undefined。
当任何值*可能*为null，你可以使用联合类型。
比如，某值可能为number或null，你可以声明它的类型为number | null。

假设有一个值TypeScript认为可以为null或undefined，但是你更清楚它的类型，你可以使用!后缀。

```
declare var foo: string[] | null;
```

```
foo.length;  // error - 'foo' is possibly 'null'

foo!.length; // okay - 'foo!' just has type 'string[]'
```

要当心，当你使用strictNullChecks，你的依赖也需要相应地启用strictNullChecks。

this没有隐式的any

当你在类的外部使用this关键字时，它会默认获得any类型。
比如，假设有一个Point类，并且我们要添加一个函数做为它的方法：

```
class Point {
    constuctor(public x, public y) {}
    getDistance(p: Point) {
        let dx = p.x - this.x;
        let dy = p.y - this.y;
        return Math.sqrt(dx ** 2 + dy ** 2);
    }
}
// ...

// Reopen the interface.
interface Point {
    distanceFromOrigin(point: Point): number;
}
Point.prototype.distanceFromOrigin = function(point: Point) {
    return this.getDistance({ x: 0, y: 0});
}
```

这就产生了我们上面提到的错误 - 如果我们错误地拼写了getDistance并不会得到一个错误。
正因此，TypeScript有noImplicitThis选项。
当设置了它，TypeScript会产生一个错误当没有明确指定类型（或通过类型推断）的this被使用时。
解决的方法是在接口或函数上使用指定了类型的this参数：

```
Point.prototype.distanceFromOrigin = function(this: Point, point: Point) {
    return this.getDistance({ x: 0, y: 0});
}
```