

15	v3.7	68
15.1	可选链 (Optional Chaining)	68
15.2	空值合并 (Nullish Coalescing)	69
15.3	断言函数	70
15.4	更好地支持返回 never 的函数	72
15.5	(更加) 递归的类型别名	72
15.6	--declaration 和 --allowJs	74
15.7	useDefineForClassFields 编译选项和 declare 属性修饰符	76
15.8	利用项目引用实现无构建编辑	78
15.9	检查未调用的函数	79
15.10	TypeScript 文件中的 // @ts-nocheck	80
15.11	分号格式化选项	80
15.12	3.7 的破坏性变更	81
15.12.1	DOM 变更	81
15.12.2	类字段处理	81
15.12.3	函数真值检查	81
15.12.4	本地和导入的类型声明现在会产生冲突	81
15.12.5	3.7 API 变化	81
16	v3.8	82
16.1	类型导入和导出 (Type-Only Imports and Exports)	82
16.2	ECMAScript 私有变量 (ECMAScript Private Fields)	83
16.2.1	Which should I use?	86
16.3	export * as ns Syntax	87
16.4	顶层 await (Top-Level await)	87
16.5	es2020 for target and module	88
16.6	JSDoc 属性修饰词 (JSDoc Property Modifiers)	88
16.7	Better Directory Watching on Linux and watchOptions	89
16.8	"Fast and Loose" Incremental Checking	91
17	v3.9	92
17.1	改进类型推断和 Promise.all	92
17.1.1	awaited 类型	92
17.2	速度优化	92
17.3	// @ts-expect-error 注释	93
17.3.1	ts-ignore 还是 ts-expect-error?	93
17.4	在条件表达式中检查未被调用的函数	94
17.5	编辑器改进	94
17.5.1	在 JavaScript 中自动导入 CommonJS 模块	95
17.5.2	Code Actions 保留换行符	95
17.5.3	快速修复: 缺失的返回值表达式	97
17.6	支持"Solution Style"的 tsconfig.json 文件	98
18	v4.0	99
18.1	可变参数组类型	99
18.2	标签元组元素	102
18.3	从构造函数中推断类属性	103
18.4	断路赋值运算符	104
18.5	catch 语句中的 unknown 类型	106
18.6	自定义 JSX 工厂	106

ts_upgrade_from_2.2_to_5.3

Catalog

1	v2.3	1
1.1	ES5/ES3 的生成器和迭代支持	1
1.1.1	迭代器	1
1.1.2	生成器	1
1.1.3	新的--downlevelIteration 编译选项	1
1.2	异步迭代	1
1.2.1	异步迭代器	1
1.2.2	异步生成器	2
1.2.3	for-await-of 语句	2
1.2.4	注意事项	2
1.3	泛型参数默认类型	2
1.4	新的--strict 主要编译选项	3
1.5	改进的--init 输出	3
1.6	--checkJS 选项下 .js 文件中的错误	3
2	v2.4	4
2.1	动态导入表达式	4
2.2	字符串枚举	4
2.3	增强的泛型推断	4
2.3.1	返回类型作为推断目标	4
2.3.2	从上下文类型中推断类型参数	4
2.3.3	对泛型函数进行更严格的检查	5
2.4	回调参数的严格抗变	5
2.5	弱类型 (Weak Type) 探测	5
3	v2.5	7
3.1	可选的 catch 语句变量	7
3.2	checkJs/@ts-check 模式中的类型断言/转换语法	7
3.3	包去重和重定向	7
3.4	--preserveSymlinks (保留符号链接) 编译器选项	7
4	v2.6	8
4.1	严格函数类型	8
4.2	缓存模块中的标签模板对象	9
4.3	本地化的命令行诊断消息	10
4.4	通过 '// @ts-ignore' 注释隐藏 .ts 文件中的错误	11
4.5	更快的 tsc --watch	12
4.6	只写的引用现在会被标记未使用	12
5	v2.7	13
5.1	常量名属性	13
5.2	unique symbol 类型	13
5.3	更严格的类属性检查	14
5.4	显式赋值断言	15
5.5	固定长度元组	15
5.6	更优的对象字面量推断	16

ts_upgrade_from_2.2_to_5.3	
5.7	结构相同的类和 instanceof 表达式的处理方式改进..... 16
5.8	in 运算符实现类型保护..... 17
5.9	使用标记--esModuleInterop 引入非 ES 模块..... 17
5.9.1	数字分隔符..... 18
5.9.2	--watch 模式下具有更简洁的输出..... 19
5.9.3	更漂亮的--pretty 输出..... 19
6	v2.8..... 20
6.1	有条件类型..... 20
6.1.1	分布式有条件类型..... 20
6.1.2	有条件类型中的类型推断..... 22
6.1.3	预定义的有条件类型..... 23
6.2	改进对映射类型修饰符的控制..... 24
6.3	改进交叉类型上的 keyof..... 25
6.4	更好的处理.js 文件中的命名空间模式..... 25
6.4.1	立即执行的函数表达式做为命名空间..... 25
6.4.2	默认声明..... 25
6.4.3	原型赋值..... 25
6.4.4	嵌套与合并声明..... 26
6.5	各文件的 JSX 工厂..... 26
6.6	本地范围的 JSX 命名空间..... 26
6.7	新的--emitDeclarationsOnly..... 26
7	v2.9..... 27
7.1	keyof 和映射类型支持用 number 和 symbol 命名的属性..... 27
7.2	JSX 元素里的泛型参数..... 29
7.3	泛型标记模版里的泛型参数..... 29
7.4	import 类型..... 30
7.5	放开声明生成时可见性规则..... 30
7.6	支持 import.meta..... 31
7.7	新的--resolveJsonModule..... 31
7.8	默认--pretty 输出..... 31
7.9	新的--declarationMap..... 32
8	v3.0..... 33
8.1	工程引用..... 33
8.2	剩余参数和展开表达式里的元组..... 33
8.2.1	带元组类型的剩余参数..... 33
8.2.2	带有元组类型的展开表达式..... 33
8.2.3	泛型剩余参数..... 33
8.2.4	元组类型里的可选元素..... 34
8.2.5	元组类型里的剩余元素..... 34
8.3	新的 unknown 类型..... 34
8.4	在 JSX 里支持 defaultProps..... 38
8.4.1	说明..... 38
8.5	/// <reference lib="..." />指令..... 38
9	v3.1..... 40
9.1	元组和数组上的映射类型..... 40
9.2	函数上的属性声明..... 40
9.3	使用 typesVersions 选择版本..... 40

	ts_upgrade_from_2.2_to_5.3
9.3.1	匹配行为..... 41
9.3.2	多个字段..... 41
10	v3.2..... 43
10.1	strictBindCallApply..... 43
10.2	对象字面量的泛型展开表达式..... 43
10.3	泛型对象剩余变量和参数..... 44
10.4	BigInt..... 44
10.5	Non-unit types as union discriminants..... 45
10.6	tsconfig.json 可以通过 Node.js 包来继承..... 45
10.7	The new --showConfig flag..... 46
10.8	JavaScript 的 Object.defineProperty 声明..... 46
11	v3.3..... 47
11.1	改进调用联合类型时的行为..... 47
11.2	在复合工程中增量地检测文件的变化 --build --watch..... 48
12	v3.4..... 50
12.1	使用 --incremental 标志加快后续构建..... 50
12.1.1	复合项目..... 50
12.1.2	outFile..... 50
12.2	泛型函数的高阶类型推断..... 50
12.3	改进 ReadonlyArray 和 readonly 元祖..... 52
12.3.1	一个与 ReadonlyArray 相关的新语法..... 52
12.3.2	readonly 元祖..... 53
12.3.3	映射类型修饰语 readonly 和 readonly 数组..... 53
12.3.4	注意事项..... 54
12.4	const 断言..... 54
12.5	对 globalThis 的类型检查..... 56
13	v3.5..... 57
13.1	改进速度..... 57
13.1.1	类型检查速度提升..... 57
13.1.2	改进 --incremental..... 57
13.2	Omit 辅助类型..... 57
13.2.1	改进了联合类型中多余属性的检查..... 57
13.3	--allowUmdGlobalAccess 标志..... 58
13.4	更智能的联合类型检查..... 58
13.5	泛型构造函数的高阶类型推断..... 59
14	v3.6..... 61
14.1	更严格的生成器..... 61
14.2	更准确的数组展开..... 63
14.3	改进了 Promises 的 UX..... 63
14.4	标识符更好的支持 Unicode..... 64
14.5	支持在 SystemJS 中使用 import.meta..... 64
14.6	在环境上下文中允许 get 和 set 访问者..... 65
14.7	环境类和函数可以合并..... 65
14.8	APIs 支持 --build 和 --incremental..... 66
14.9	新的 TypeScript Playground..... 66
14.10	代码编辑的分号感知..... 66
14.11	更智能的自动导入..... 66

28.11	支持 JSDoc 中的 @satisfies	209
28.12	支持 JSDoc 中的 @overload	211
28.13	在 --build 模式下使用有关文件生成的选项	212
28.14	编辑器导入语句排序时不区分大小写	212
28.15	穷举式 switch/case 自动补全	214
28.16	速度, 内存以及代码包尺寸优化	214
29	v5.1	217
29.1	更易用的隐式返回 undefined 的函数	217
29.2	不相关的存取器类型	219
29.3	解耦 JSX 元素和 JSX 标签类型之间的类型检查	220
29.4	带有命名空间的 JSX 属性	221
29.5	模块解析时考虑 typeRoots	221
29.6	在 JSX 标签上链接光标	221
29.7	@param JSDoc 标记的代码片段自动补全	222
29.8	优化	222
29.8.1	避免非必要的类型初始化	222
29.8.2	联合字面量的反面情况检查	223
29.8.3	减少在解析 JSDoc 时的扫描函数调用	223
30	v5.2	224
30.1	using 声明与显式资源管理	224
30.2	Decorator Metadata	230
30.3	命名的和匿名的元组元素	234
30.4	更容易地使用联合数组上的方法	234
30.5	拷贝的数组方法	235
30.6	将 symbol 用于 WeakMap 和 WeakSet 的键	236
30.7	类型导入路径里使用 TypeScript 实现文件扩展名	236
30.8	对象成员的逗号补全	236
30.9	内联变量重构	236
30.10	可点击的内嵌参数提示	236
30.11	优化进行中的类型兼容性检查	237
31	v5.3	238
31.1	导入属性 (Import Attributes)	238
31.2	稳定支持 import type 上的 resolution-mode	238
31.3	在所有模块模式中支持 resolution-mode	239
31.4	switch (true) 类型细化	239
31.5	类型细化与布尔值的比较	239
31.6	利用 Symbol.hasInstance 来细化 instanceof	240
31.7	在实例字段上检查 super 属性访问	241
31.8	可以交互的类型内嵌提示	242
31.9	设置偏好 type 自动导入	243
31.10	优化: 略过 JSDoc 解析	243
31.11	通过比较非规范化的交叉类型进行优化	243
31.12	合并 tsserverlibrary.js 和 typescript.js	243

18.7	对启用了--noEmitOnError 的`build 模式进行速度优化	107
18.8	--incremental 和--noEmit	107
18.9	编辑器改进	108
18.9.1	转换为可选链	108
18.9.2	/** @deprecated */支持	108
18.9.3	启动时的局部语义模式	109
18.9.4	更智能的自动导入	109
18.10	我们的新网站	110
19	v4.1	111
19.1	模版字面量类型	111
19.2	在映射类型中更改映射的键	114
19.3	递归的有条件类型	115
19.4	索引访问类型检查 (--noUncheckedIndexedAccess)	116
19.5	不带 baseUrl 的 paths	118
19.6	checkJs 默认启用 allowJs	118
19.7	React 17 JSX 工厂	118
19.8	在编辑器中支持 JSDoc @see 标签	118
19.9	破坏性改动	119
19.9.1	lib.d.ts 更新	119
19.9.2	abstract 成员不能被标记为 async	119
19.9.3	any/unknown Are Propagated in Falsy Positions	119
19.9.4	Promise 的 resolve 的参数不再是可选的	119
19.9.5	有条件展开会创建可选属性	120
19.9.6	Unmatched parameters are no longer related	121
20	v4.2	122
20.1	更智能地保留类型别名	122
20.2	元组类型中前导的/中间的剩余元素	122
20.3	更严格的 in 运算符检查	124
20.4	--noPropertyAccessFromIndexSignature	124
20.5	abstract 构造签名	125
20.6	使用 --explainFiles 来理解工程的结构	127
20.7	改进逻辑表达式中的未被调用函数检查	128
20.8	解构出来的变量可以被明确地标记为未使用的	129
20.9	放宽了在可选属性和字符串索引签名间的限制	129
20.10	声明缺失的函数	130
21	v4.3	131
21.1	拆分属性的写入类型	131
21.2	override 和 --noImplicitOverride 标记	133
21.3	模版字符串类型改进	135
21.4	ECMAScript #private 的类成员	136
21.5	ConstructorParameters 可用于抽象类	137
21.6	按上下文细化泛型类型	138
21.7	检查总是为真的 Promise	140
21.8	static 索引签名	140
21.9	.tsbuildinfo 文件大小改善	141
21.10	在 --incremental 和 --watch 中进行惰性计算	141
21.11	导入语句的补全	142

21.12	编辑器对 @link 标签的支持	143
21.13	在非 JavaScript 文件上的跳转到定义	143
22 v4.4		145
22.1	针对条件表达式和判别式的别名引用进行控制流分析	145
22.2	Symbol 以及模版字符串索引签名	147
22.3	Defaulting to the unknown Type in Catch Variables (--useUnknownInCatchVariables)	149
22.4	异常捕获变量的类型默认为 unknown (--useUnknownInCatchVariables)	149
22.5	确切的可选属性类型 (--exactOptionalPropertyTypes)	150
22.6	类中的 static 语句块	151
22.7	tsc --help 更新与优化	152
22.8	性能优化	152
22.8.1	更快地生成声明文件	152
22.8.2	更快地标准化路径	153
22.8.3	更快地路径映射	153
22.8.4	更快地增量构建与 --strict	153
22.8.5	针对大型输出更快地生成 Source Map	153
22.8.6	更快的 --force 构建	153
22.9	JavaScript 中的拼写建议	153
22.10	内嵌提示 (Inlay Hints)	153
22.11	自动导入的补全列表里显示真正的路径	154
23 v4.5		156
23.1	支持从 node_modules 里读取 lib	156
23.2	改进 Awaited 类型和 Promise	156
23.3	模版字符串类型作为判别式属性	157
23.4	module es2022	157
23.5	在条件类型上消除尾递归	157
23.6	禁用导入省略	158
23.7	在导入名称前使用 type 修饰符	159
23.8	私有字段存在性检查	160
23.9	导入断言	161
23.10	使用 realPathSync.native 获得更快的加载速度	161
23.11	JSX Attributes 的代码片段自动补全	161
23.12	为未解决类型提供更好的编辑器支持	162
24 v4.6		164
24.1	允许在构造函数中的 super() 调用之前插入代码	164
24.2	基于控制流来分析解构的可辨识联合类型	164
24.3	改进的递归深度检查	165
24.4	索引访问类型推断改进	166
24.5	对因变参数的控制流分析	167
24.6	--target es2022	168
24.7	删除 react-jsx 中不必要的参数	168
24.8	JSDoc 命名建议	168
24.9	JavaScript 中更多的语法和绑定错误提示	169
24.10	TypeScript Trace 分析器	169
25 v4.7		171
25.1	Node.js 对 ECMAScript Module 的支持	171
25.1.1	package.json 里的 type 字段和新的文件扩展名	171

25.1.2	新的文件扩展名	172
25.1.3	CommonJS 互操作性	172
25.1.4	package.json 中的 exports, imports 以及自引用	173
25.2	设置模块检测策略	174
25.3	[] 语法元素访问的控制流分析	175
25.4	改进对象和方法里的函数类型推断	176
25.5	实例化表达式	177
25.6	infer 类型参数上的 extends 约束	178
25.7	可选的类型参数变型注释	178
25.8	使用 moduleSuffixes 自定义解析策略	180
25.9	resolution-mode	181
25.10	跳转到在源码中的定义	181
25.11	分组整理导入语句	182
26 v4.8		184
26.1	改进的交叉类型化简、联合类型兼容性以及类型细化	184
26.2	改进模版字符串类型中 infer 类型的类型推断	185
26.3	--build, --watch, 和 --incremental 的性能优化	185
26.4	比较对象和数组字面量时报错	186
26.5	改进从绑定模式中进行类型推断	186
26.6	修复文件监视 (尤其是在 git checkout 之间)	187
26.7	查找所有引用性能优化	187
26.8	从自动导入中排除指定文件	187
27 v4.9		188
27.1	satisfies 运算符	188
27.2	使用 in 运算符来细化并未列出其属性的对象类型	189
27.3	类中的自动存取器	191
27.4	在 NaN 上的相等性检查	192
27.5	监视文件功能使用文件系统事件	192
27.6	编辑器中的“删除未使用导入”和“排序导入”命令	193
27.7	在 return 关键字上使用跳转到定义	194
27.8	性能优化	194
28 v5.0		195
28.1	装饰器 Decorators	195
28.2	与旧的实验性的装饰器的差异	199
28.3	编写强类型的装饰器	200
28.4	const 类型参数	200
28.5	extends 支持多个配置文件	202
28.6	所有的 enum 均为联合 enum	203
28.7	--moduleResolution bundler	204
28.8	定制化解析的标记	205
28.8.1	allowImportingTsExtensions	205
28.8.2	resolvePackageJsonExports	205
28.8.3	resolvePackageJsonImports	205
28.8.4	allowArbitraryExtensions	205
28.8.5	customConditions	206
28.9	--verbatimModuleSyntax	206
28.10	支持 export type *	208

2 v2.4

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-2.4.html>

2.1 动态导入表达式

动态的 `import` 表达式是一个新特性，它属于 ECMAScript 的一部分，允许用户在程序的任何位置异步地请求某个模块。

这意味着你可以有条件地延迟加载其它模块和库。例如下面这个 `async` 函数，它仅在需要的时候才导入工具库：

```
async function getZipFile(name: string, files: File[]): Promise<File>
{
    const zipUtil = await import('./utils/create-zip-file');
    const zipContents = await zipUtil.getContentAsBlob(files);
    return new File(zipContents, name);
}
```

许多 bundlers 工具已经支持依照这些 `import` 表达式自动地分割输出，因此可以考虑使用这个新特性并把输出模块目标设置为 `esnext`。

2.2 字符串枚举

TypeScript 2.4 现在支持枚举成员变量包含字符串构造器。

```
enum Colors {
    Red = "RED",
    Green = "GREEN",
    Blue = "BLUE",
}
```

需要注意的是字符串枚举成员不能被反向映射到枚举成员的名字。换句话说，你不能使用 `Colors["RED"]` 来得到 `"Red"`。

2.3 增强的泛型推断

TypeScript 2.4 围绕着泛型的推断方式引入了一些很棒的变化。

2.3.1 返回类型作为推断目标

其一，TypeScript 能够推断调用的返回值类型。这可以优化你的体验和方便捕获错误。如下所示：

```
function arrayMap<T, U>(f: (x: T) => U): (a: T[]) => U[] {
    return a => a.map(f);
}
```

```
const lengths: (a: string[]) => number[] = arrayMap(s => s.length);
```

下面是一个你可能会见到的出错了的例子：

```
let x: Promise<string> = new Promise(resolve => {
    resolve(10);
    //      ~~ Error!
});
```

2.3.2 从上下文类型中推断类型参数

在 TypeScript 2.4 之前，在下面的例子里：

```
let f: <T>(x: T) => T = y => y;
```

`y` 将会具有 `any` 类型。这意味着虽然程序会检查类型，但是你却可以使用 `y` 做任何事情，比如：

```
let f: <T>(x: T) => T = y => y() + y.foo.bar;
```

这个例子实际上并不是类型安全的。

1 v2.3

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-2.3.html>

1.1 ES5/ES3 的生成器和迭代支持

首先是一些 ES2016 的术语：

1.1.1 迭代器

ES2015 引入了 [Iterator \(迭代器\)](#)，它表示提供了 `next`，`return`，以及 `throw` 三个方法的对象，具体满足以下接口：

```
interface Iterator<T> {
    next(value?: any): IteratorResult<T>;
    return?(value?: any): IteratorResult<T>;
    throw?(e?: any): IteratorResult<T>;
}
```

这种迭代器对于迭代可用的值时很有用，比如数组的元素或者 `Map` 的键。如果一个对象有一个返回 `Iterator` 对象的 `Symbol.iterator` 方法，那么我们说这个对象是“可迭代的”。

迭代器协议还定义了一些 ES2015 中的特性像 `for..of` 和展开运算符以及解构赋值中的数组的剩余运算的操作对象。

1.1.2 生成器

ES2015 也引入了“生成器”，生成器是可以通过 `Iterator` 接口和 `yield` 关键字被用来生成部分运算结果的函数。生成器也可以在内部通过 `yield*` 代理对与其他可迭代对象的调用。举例来说：

```
function* f() {
    yield 1;
    yield* [2, 3];
}
```

1.1.3 新的 `--downlevelIteration` 编译选项

之前迭代器只在编译目标为 ES6/ES2015 或者更新版本时可用。此外，设计迭代器协议的结构，比如 `for..of`，如果编译目标低于 ES6/ES2015，则只能在操作数组时被支持。

TypeScript 2.3 在 ES3 和 ES5 为编译目标时由 `--downlevelIteration` 编译选项增加了完整的对生成器和迭代器协议的支持。

通过 `--downlevelIteration` 编译选项，编译器会使用新的类型检查和输出行为，尝试调用被迭代对象的 `[Symbol.iterator]()` 方法（如果有），或者在对象上创建一个语义上的数组迭代器。

注意这需要非数组的值有原生的 `Symbol.iterator` 或者 `Symbol.iterator` 的运行时模拟实现。

使用 `--downlevelIteration` 时，在 ES5/ES3 中 `for..of` 语句、数组解构、数组中的元素展开、函数调用、`new` 表达式在支持 `Symbol.iterator` 时可用，但即便没有定义 `Symbol.iterator`，它们在运行时或开发时都可以被使用到数组上。

1.2 异步迭代

TypeScript 2.3 添加了对异步迭代器和生成器的支持，描述见当前的 [TC39 提案](#)。

1.2.1 异步迭代器

异步迭代引入了 `AsyncIterator`，它和 `Iterator` 相似。实际上的区别在于

`AsyncIterator` 的 `next`、`return` 和 `throw` 方法的返回的是迭代结果的 `Promise`，而不是结果本身。这允许 `AsyncIterator` 在生成值之前的时间点就加入异步通知。`AsyncIterator` 的接口如下：

```
interface AsyncIterator<T> {
```

```
next(value?: any): Promise<IteratorResult<T>>;
return?(value?: any): Promise<IteratorResult<T>>;
throw?(e?: any): Promise<IteratorResult<T>>;
}
```

一个支持异步迭代的对象如果有一个返回 AsyncIterator 对象的 Symbol.asyncIterator 方法，被称作是“可迭代的”。

1.2.2 异步生成器

[异步迭代提案](#)引入了“异步生成器”，也就是可以用来生成部分计算结果的异步函数。异步生成器也可以通过 yield* 代理对可迭代对象或异步可迭代对象的调用：

```
async function* g() {
  yield 1;
  await sleep(100);
  yield* [2, 3];
  yield* (async function* () {
    await sleep(100);
    yield 4;
  })();
}
```

和生成器一样，异步生成器只能是函数声明，函数表达式，或者类或对象字面量的方法。箭头函数不能作为异步生成器。异步生成器除了一个可用的 Symbol.asyncIterator 引用外（原生或三方实现），还需要一个可用的全局 Promise 实现（既可以是原生的，也可以是 ES2015 兼容的实现）。

1.2.3 for-await-of 语句

最后，ES2015 引入了 for...of 语句来迭代可迭代对象。相似的，异步迭代提案引入了 for...await...of 语句来迭代可异步迭代的对象。

```
async function f() {
  for await (const x of g()) {
    console.log(x);
  }
}
```

for...await...of 语句仅在异步函数或异步生成器中可用。

1.2.4 注意事项

- 始终记住我们对于异步迭代器的支持是建立在运行时有 Symbol.asyncIterator 支持的基础上的。你可能需要 Symbol.asyncIterator 的三方实现，虽然对于简单的目的可以仅仅是：(Symbol as any).asyncIterator = Symbol.asyncIterator || Symbol.for("Symbol.asyncIterator");
- 如果你没有声明 AsyncIterator，还需要在 --lib 选项中加入 esnext 来获取 AsyncIterator 声明。
- 最后，如果你的编译目标是 ES5 或 ES3，你还需要设置 --downlevelIterators 编译选项。

1.3 泛型参数默认类型

TypeScript 2.3 增加了对声明泛型参数默认类型的支持。

[示例](#)

考虑一个会创建新的 HTMLElement 的函数，调用时不加参数会生成一个 Div，你也可以选择性地传入子元素的列表。之前你必须这么去定义：

```
declare function create(): Container<HTMLDivElement,
HTMLDivElement[]>;
declare function create<T extends HTMLElement>(element: T):
Container<T, T[]>;
declare function create<T extends HTMLElement, U extends
HTMLElement>(element: T, children: U[]): Container<T, U[]>;
```

有了泛型参数默认类型，我们可以将定义化简为：

```
declare function create<T extends HTMLElement = HTMLDivElement, U =
T[]>(element?: T, children?: U): Container<T, U>;
```

泛型参数的默认类型遵循以下规则：

- 有默认类型的类型参数被认为是可选的。
- 必选的类型参数不能在可选的类型参数后。
- 如果类型参数有约束，类型参数的默认类型必须满足这个约束。
- 当指定类型实参时，你只需要指定必选类型参数的类型实参。未指定的类型参数会被解析为它们的默认类型。
- 如果指定了默认类型，且类型推断无法选择一个候选类型，那么将使用默认类型作为推断结果。
- 一个被现有类或接口合并的类或者接口的声明可以为现有类型参数引入默认类型。
- 一个被现有类或接口合并的类或者接口的声明可以引入新的类型参数，只要它指定了默认类型。

1.4 新的--strict 主要编译选项

TypeScript 加入的新检查项为了避免不兼容现有项目通常都是默认关闭的。虽然避免不兼容是好事，但这个策略的一个弊端则是使配置最高类型安全越来越复杂，这么做每次 TypeScript 版本发布时都需要显示地加入新选项。有了 --strict 编译选项，就可以选择最高级别的安全（了解随着更新版本的编译器增加了增强的类型检查特性可能会报新的错误）。

新的 --strict 编译器选项包含了一些建议配置的类型检查选项。具体来说，指定 --strict 相当于是指定了以下所有选项（未来还可能包括更多选项）：

- --strictNullChecks
- --noImplicitAny
- --noImplicitThis
- --alwaysStrict

确切地说，--strict 编译选项会为以上列出的编译器选项设置默认值。这意味着还可以单独控制这些选项。比如：

```
--strict --noImplicitThis false
```

这将是开启除 --noImplicitThis 编译选项以外的所有严格检查选项。使用这个方式可以表述除某些明确列出的项以外的所有严格检查项。换句话说，现在可以在默认最高级别的安全下排除部分检查。

从 TypeScript 2.3 开始，tsc --init 生成的默认 tsconfig.json 在 "compilerOptions" 中包含了 "strict: true" 设置。这样一来，用 tsc --init 创建的新项目默认会开启最高级别的安全。

1.5 改进的--init 输出

除了默认的 --strict 设置外，tsc --init 还改进了输出。tsc --init 默认生成的 tsconfig.json 文件现在包含了一些带描述的被注释掉的常用编译器选项，你可以去掉相关选项的注释来获得期望的结果。我们希望新的输出能简化新项目的配置并且随着项目成长保持配置文件的可读性。

1.6 --checkJS 选项下 .js 文件中的错误

即便使用了 --allowJs，TypeScript 编译器默认不会报 .js 文件中的任何错误。

TypeScript 2.3 中使用 --checkJs 选项，.js 文件中的类型检查错误也可以被报出。

你可以通过为它们添加 // @ts-nocheck 注释来跳过对某些文件的检查，反过来你也可以选择通过添加 // @ts-check 注释只检查一些 .js 文件而不需要设置 --checkJs 编译选项。你也可以通过添加 // @ts-ignore 到特定行的一行前来忽略这一行的错误。

.js 文件仍然会被检查确保只有标准的 ECMAScript 特性，类型标注仅在 .ts 文件中被允许，在 .js 中会被标记为错误。JSDoc 注释可以用来为你的 JavaScript 代码添加某些类型信息，更多关于支持的 JSDoc 结构的详情，请浏览 [JSDoc 支持文档](#)。

有关详细信息，请浏览 [类型检查 JavaScript 文件文档](#)。

4 v2.6

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-2.6.html>

4.1 严格函数类型

TypeScript 2.6 引入了新的类型检查选项，--strictFunctionTypes。--strictFunctionTypes 选项是--strict 系列选项之一，也就是说 --strict 模式下它默认是启用的。你可以通过在命令行或 tsconfig.json 中设置--strictFunctionTypes false 来单独禁用它。

--strictFunctionTypes 启用时，函数类型参数的检查是_抗变 (contravariantly) _而非_双变 (bivariantly) _的。关于变体 (variance) 对于函数类型意义的相关背景，请查看[协变 \(covariance\) 和抗变 \(contravariance\) 是什么?](#)。

这一更严格的检查应用于除方法或构造函数声明以外的所有函数类型。方法被专门排除在外是为了确保带泛型的类和接口 (如 Array<T>) 总体上仍然保持协变。

考虑下面这个 Animal 是 Dog 和 Cat 的父类型的例子：

```
declare let f1: (x: Animal) => void;
declare let f2: (x: Dog) => void;
declare let f3: (x: Cat) => void;
f1 = f2; // 启用 --strictFunctionTypes 时错误
f2 = f1; // 正确
f2 = f3; // 错误
```

第一个赋值语句在默认的类型检查模式中是允许的，但是在严格函数类型模式下会被标记错误。通俗地讲，默认模式允许这么赋值，因为它_可能是_合理的，而严格函数类型模式将它标记为错误，因为它不能_被证明_合理。任何一种模式中，第三个赋值都是错误的，因为它_永远不_合理。

用另一种方式来描述这个例子则是，默认类型检查模式中 T 在类型(x: T) => void 是_双变的_ (也即协变_或_抗变)，但在严格函数类型模式中 T 是_抗变_的。

例子

```
interface Comparer<T> {
  compare: (a: T, b: T) => number;
}
```

```
declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;
```

```
animalComparer = dogComparer; // 错误
dogComparer = animalComparer; // 正确
```

现在第一个赋值是错误的。更明确地说，Comparer<T>中的 T 因为仅在函数类型参数的位置被使用，是抗变的。

另外，注意尽管有的语言 (比如 C#和 Scala) 要求变体标注 (variance annotations) (out/in 或 +/-)，而由于 TypeScript 的结构化类型系统，它的变体是由泛型中的类型参数的实际使用自然得出的。

注意：

启用--strictFunctionTypes 时，如果 compare 被声明为方法，则第一个赋值依然是被允许的。更明确的说，Comparer<T>中的 T 因为仅在方法参数的位置被使用所以是双变的。

```
interface Comparer<T> {
  compare(a: T, b: T): number;
}
```

在 TypeScript 2.4 里，右手边的函数会隐式地获得类型参数，并且 y 的类型会被推断为那个类型参数的类型。

如果你使用 y 的方式是这个类型参数所不支持的，那么你会得到一个错误。在这个例子里，T 的约束是{} (隐式地)，所以在最后一个例子里会出错。

2.3.3 对泛型函数进行更严格的检查

TypeScript 在比较两个单一签名的类型时会尝试统一类型参数。因此，在涉及到两个泛型签名的时候会进行更严格的检查，这就可能发现一些 bugs。

```
type A = <T, U>(x: T, y: U) => [T, U];
type B = <S>(x: S, y: S) => [S, S];
```

```
function f(a: A, b: B) {
  a = b; // Error
  b = a; // Ok
}
```

2.4 回调参数的严格抗变

TypeScript 一直是以双变 (bivariant) 的方式来比较参数。这样做有很多原因，总体上来说这不会有什么大问题直到我们发现它应用在 Promise 和 Observable 上时有些副作用。

TypeScript 2.4 在处理两个回调类型时引入了收紧机制。例如：

```
interface Mappable<T> {
  map<U>(f: (x: T) => U): Mappable<U>;
}
```

```
declare let a: Mappable<number>;
declare let b: Mappable<string | number>;
```

```
a = b;
b = a;
```

在 TypeScript 2.4 之前，它会成功执行。当关联 map 的类型时，TypeScript 会双向地关联它们的类型 (例如 f 的类型)。当关联每个 f 的类型时，TypeScript 也会双向地关联那些参数的类型。

TS 2.4 里关联 map 的类型时，TypeScript 会检查是否每个参数都是回调类型，如果是的话，它会确保那些参数根据它所在的位置以抗变 (contravariant) 地方式进行检查。换句话说，TypeScript 现在可以捕获上面的 bug，这对某些用户来说可能是一个破坏性改动，但却是非常帮助的。

2.5 弱类型 (Weak Type) 探测

TypeScript 2.4 引入了“弱类型”的概念。任何只包含了可选属性的类型被当作是“weak”。比如，下面的 Options 类型是弱类型：

```
interface Options {
  data?: string,
  timeout?: number,
  maxRetries?: number,
}
```

在 TypeScript 2.4 里给弱类型赋值时，如果这个值的属性与弱类型的属性没有任何重叠属性时会得到一个错误。比如：

```
function sendMessage(options: Options) {
  // ...
}
```

```
ts_upgrade_from_2.2_to_5.3
const opts = {
  payload: "hello world!",
  retryOnFail: true,
}

// 错误!
sendMessage(opts);
// 'opts' 和 'Options' 没有重叠的属性
// 可能我们想要用'data'/'maxRetries'来代替'payload'/'retryOnFail'
因为这是一个破坏性改动，你可能想要知道一些解决方法：
1. 确定属性存在时再声明
2. 给弱类型增加索引签名（比如 [propName: string]: {}）
3. 使用类型断言（比如 opts as Options）
```

3 v2.5

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-2.5.html>

3.1 可选的 catch 语句变量

得益于@tinganho 所做的工作，TypeScript 2.5 实现了一个新的 ECMAScript 特性，允许用户省略 catch 语句中的变量。例如，当使用 JSON.parse 时，你可能需要将对应的函数调用放在 try / catch 中，但是最后可能并不会用到输入有误时会抛出的 SyntaxError（语法错误）。

```
let input = "...";
try {
  JSON.parse(input);
}
catch {
  // ^ 注意我们的 `catch` 语句并没有声明一个变量
  console.log("传入的 JSON 不合法\n\n" + input)
}
```

3.2 checkJs/@ts-check 模式中的类型断言/转换语法

TypeScript 2.5 引入了在使用纯 JavaScript 的项目中断言表达式类型的能力。对应的语法是 /** @type {...} */ 标注注释后加上被圆括号括起来，类型需要被重新演算的表达式。举例：

```
var x = /** @type {SomeType} */ (AnyParenthesizedExpression);
```

3.3 包去重和重定向

在 TypeScript 2.5 中使用 Node 模块解析策略进行导入时，编译器现在会检查文件是否来自“相同”的包。如果一个文件所在的包的 package.json 包含了与之前读取的包相同的 name 和 version，那么 TypeScript 会将它重定向到最顶层的包。这可以解决两个包可能会包含相同的类声明，但因为包含 private 成员导致他们在结构上不兼容的问题。

这也带来一个额外的好处，可以通过避免从重复的包中加载 .d.ts 文件减少内存使用和编译器及语言服务的运行时计算。

3.4 --preserveSymlinks（保留符号链接）编译器选项

TypeScript 2.5 带来了 preserveSymlinks 选项，它对应了 Node.js 中 --preserve-symlinks 选项的行为。这一选项也会带来和 Webpack 的 resolve.symlinks 选项相反的行为（也就是说，将 TypeScript 的 preserveSymlinks 选项设置为 true 对应了将 Webpack 的 resolve.symlinks 选项设为 false，反之亦然）。

在这一模式中，对于模块和包的引用（比如 import 语句和 /// <reference type=".." /> 指令）都会以相对符号链接文件的位置被解析，而不是相对于符号链接解析到的路径。更具体的例子，可以参考 [Node.js 网站的文档](#)。

4.5 更快的 `tsc --watch`

TypeScript 2.6 带来了更快的`--watch`实现。新版本优化了使用 ES 模块的代码的生成和检查。在一个模块文件中检测到的改变_只_会使改变的模块，以及依赖它的文件被重新生成，而不再是整个项目。有大量文件的项目应该从这一改变中获益最多。

这一新的实现也为 `tsserver` 中的监听带来了性能提升。监听逻辑被完全重写以更快响应改变事件。

4.6 只写的引用现在会被标记未使用

TypeScript 2.6 加入了修正的`--noUnusedLocals`和`--noUnusedParameters`[编译选项](#)实现。只被写但从没有被读的声明现在会被标记未使用。

例子

下面 `n` 和 `m` 都会被标记为未使用，因为它们的值从未被_读取_。之前 TypeScript 只会检查它们的值是否被_引用_。

```
function f(n: number) {
    n = 0;
}
```

```
class C {
    private m: number;
    constructor() {
        this.m = 0;
    }
}
```

另外仅被自己内部调用的函数也会被认为是未使用的。

例子

```
function f() {
    f(); // 错误: 'f' 被声明，但它的值从未被使用
}
```

```
declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;
```

```
animalComparer = dogComparer; // 正确, 因为双变
dogComparer = animalComparer; // 正确
```

TypeScript 2.6 还改进了与抗变位置相关的类型推导:

```
function combine<T>(...funcs: ((x: ) => void)[]): (x: T) => void {
    return x => {
        for (const f of funcs) f(x);
    }
}
```

```
function animalFunc(x: Animal) {}
function dogFunc(x: Dog) {}
```

```
let combined = combine(animalFunc, dogFunc); // (x: Dog) => void
```

这上面所有 `T` 的推断都来自抗变的位置，由此我们得出 `T` 的_最普遍子类型_。这与从协变位置推导出的结果恰恰相反，从协变位置我们得出的是_最普遍超类型_。

4.2 缓存模块中的标签模板对象

TypeScript 2.6 修复了标签字符串模板的输出，以更好地遵循 ECMAScript 标准。根据 [ECMAScript 标准](#)，每一次获取模板标签的值时，应该将_同一个_模板字符串数组对象（同一个 `TemplateStringArray`）作为第一个参数传递。在 TypeScript 2.6 之前，每一次生成的都是全新的模板对象。虽然字符串的内容是一样的，这样的输出会影响通过识别字符串来实现缓存失效的库，比如 [lit-html](#)。

例子

```
export function id(x: TemplateStringsArray) {
    return x;
}
```

```
export function templateObjectFactory() {
    return id`hello world`;
}
```

```
let result = templateObjectFactory() === templateObjectFactory(); //
TS 2.6 为 true
```

编译后的代码:

```
"use strict";
var __makeTemplateObject = (this && this.__makeTemplateObject) ||
function (cooked, raw) {
    if (Object.defineProperty) { Object.defineProperty(cooked, "raw",
{ value: raw }); } else { cooked.raw = raw; }
    return cooked;
};
```

```
function id(x) {
    return x;
}
```

```
var _a;  
function templateObjectFactory() {  
    return id(_a || (_a = __makeTemplateObject(["hello world"], ["hello world"])));  
}
```

var result = templateObjectFactory() === templateObjectFactory();
注意：这一改变引入了新的工具函数，__makeTemplateObject；如果你在搭配使用--importHelpers 和 tslib，需要更新到 1.8 或更高版本。

4.3 本地化的命令行诊断消息

TypeScript 2.6 npm 包加入了 13 种语言的诊断消息本地化版本。 命令行中本地化消息会在使用--locale 选项时显示。

例子

俄语显示的错误消息：

```
c:\ts>tsc --v  
Version 2.6.1  
  
c:\ts>tsc --locale ru --pretty c:\test\a.ts
```

../test/a.ts(1,5): error TS2322: Тип `""string""` не может быть назначен для типа `"number"`.

```
1 var x: number = "string";  
    ~
```

中文显示的帮助信息：

```
PS C:\ts> tsc --v  
Version 2.6.1  
  
PS C:\ts> tsc --locale zh-cn  
版本 2.6.1  
语法: tsc [选项] [文件 ...]
```

```
示例: tsc hello.ts  
      tsc --outFile file.js file.ts  
      tsc @args.txt
```

选项：
-h, --help 打印此消息。
--all 显示所有编译器选项。
-v, --version 打印编译器的版本。
--init 初始化 TypeScript 项目并创建 tsconfig.json 文件。
-p 文件或目录, --project 文件或目录 编译给定了其配置文件路径或带 "tsconfig.json" 的文件夹路径的项目。
--pretty 使用颜色和上下文风格化错误和消息(实验)。
-w, --watch 监视输入文件。

ts_upgrade_from_2.2_to_5.3
-t 版本, --target 版本 指定 ECMAScript 目标版本: "ES3"(默认)、"ES5"、"ES2015"、"ES2016"、"ES2017" 或 "ESNext"。
-m 种类, --module 种类 指定模块代码生成: "none"、"commonjs"、"amd"、"system"、"umd"、"es2015"或 "ESNext"。
--lib 指定要在编译中包括的库文件：
 'es5' 'es6' 'es2015' 'es7' 'es2016'
'es2017' 'esnext' 'dom' 'dom.iterable' 'webworker' 'scripthost'
'es2015.core' 'es2015.collection' 'es2015.generator' 'es2015.iterable'
'es2015.promise' 'es2015.proxy' 'es2015.reflect' 'es2015.symbol'
'es2015.symbol.wellknown' 'es2016.array.include' 'es2017.object'
'es2017.sharedmemory' 'es2017.string' 'es2017.intl'
'esnext.asynciterable'
--allowJs 允许编译 JavaScript 文件。
--jsx 种类 指定 JSX 代码生成: "preserve"、"react-native"或 "react"。 -d, --declaration 生成相应的 ".d.ts" 文件。
--sourceMap 生成相应的 ".map" 文件。
--outFile 文件 连接输出并将其发出到单个文件。
--outDir 目录 将输出结构重定向到目录。
--removeComments 请勿将注释发出到输出。
--noEmit 请勿发出输出。
--strict 启用所有严格类型检查选项。
--noImplicitAny 对具有隐式 "any" 类型的表达式和声明引发错误。
--strictNullChecks 启用严格的 NULL 检查。
--strictFunctionTypes 对函数类型启用严格检查。
--noImplicitThis 在带隐式 "any" 类型的 "this" 表达式上引发错误。
--alwaysStrict 以严格模式进行分析, 并为每个源文件发出 "use strict" 指令。
--noUnusedLocals 报告未使用的局部变量上的错误。
--noUnusedParameters 报告未使用的参数上的错误。
--noImplicitReturns 在函数中的所有代码路径并非都返回值时报告错误。
--noFallthroughCasesInSwitch 报告 switch 语句中遇到 fallthrough 情况的错误。
--types 要包含在编译中类型声明文件。
@<文件> 从文件插入命令行选项和文件。

4.4 通过 '// @ts-ignore' 注释隐藏 .ts 文件中的错误

TypeScript 2.6 支持在.ts 文件中通过在报错一行上方使用// @ts-ignore 来忽略错误。

例子

```
if (false) {  
    // @ts-ignore: 无法被执行的代码的错误  
    console.log("hello");  
}
```

// @ts-ignore 注释会忽略下一行中产生的所有错误。 建议实践中在@ts-ignore 之后添加相关提示, 解释忽略了什么错误。
请注意, 这个注释仅会隐藏报错, 并且我们建议你_极少_使用这一注释。

在 TypeScript 2.7 中，具有不同元数的元组不再允许相互赋值。感谢 [Tycho Grouwstra](#) 提交的 PR，元组类型现在会将它们的元数编码进它们对应的 length 属性的类型里。原理是利用数字字面量类型区分出不同长度的元组。

概念上讲，你可以把 [number, string] 类型等同于下面的 NumStrTuple 声明：

```
interface NumStrTuple extends Array<number | string> {
  0: number;
  1: string;
  length: 2; // 注意 length 的类型是字面量'2'，而不是'number'
}
```

请注意，这是一个破坏性改动。如果你想要和以前一样，让元组仅限制最小长度，那么你可以使用一个类似的声明但不显式指定 length 属性，这样 length 属性的类型就会回退为 number

```
interface MinimumNumStrTuple extends Array<number | string> {
  0: number;
  1: string;
}
```

注：这并不意味着元组是不可变长的数组，而仅仅是一个约定。

5.6 更优的对象字面量推断

TypeScript 2.7 改进了在同一上下文中的多对象字面量的类型推断。当多个对象字面量类型组成一个联合类型，TypeScript 现在会将它们_规范化_为一个对象类型，该对象类型包含联合类型中的每个对象的所有属性，以及属性对应的推断类型。

考虑这样的情形：

```
const obj = test ? { text: "hello" } : {}; // { text: string } | { text?: undefined }
const s = obj.text; // string | undefined
```

以前 obj 会被推断为 {}，第二行会报错因为 obj 没有属性。但这显然并不理想。

例子

```
// let obj: { a: number, b: number } | { a: string, b?: undefined } | { a?: undefined, b?: undefined }
let obj = [{ a: 1, b: 2 }, { a: "abc" }, {}][0];
obj.a; // string | number | undefined
obj.b; // number | undefined
多个对象字面量中的同一属性的所有推断类型，会合并成一个规范化的联合类型：
declare function f<T>(...items: T[]): T;
// let obj: { a: number, b: number } | { a: string, b?: undefined } | { a?: undefined, b?: undefined }
let obj = f({ a: 1, b: 2 }, { a: "abc" }, {});
obj.a; // string | number | undefined
obj.b; // number | undefined
```

5.7 结构相同的类和 instanceof 表达式的处理方式改进

TypeScript 2.7 对联合类型中结构相同的类和 instanceof 表达式的处理方式改进如下：

- 联合类型中，结构相同的不同类都会保留（而不是只保留一个）
- 联合类型中的子类型简化仅在一种情况下发生——若一个类继承自联合类型中另一个类，该子类会被简化。
- 用于类型检查的 instanceof 操作符基于继承关系来判断，而不是结构兼容来判断。

这意味着联合类型和 instanceof 能够区分结构相同的类。

5 v2.7

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-2.7.html>

5.1 常量名属性

TypeScript 2.7 新增了以常量（包括 ECMAScript symbols）作为类属性名的类型推断支持。

例子

```
// Lib
export const SERIALIZE = Symbol("serialize-method-key");
```

```
export interface Serializable {
  [SERIALIZE](obj: {}): string;
}
// consumer
import { SERIALIZE, Serializable } from "lib";
```

```
class JSONSerializableItem implements Serializable {
  [SERIALIZE](obj: {}) {
    return JSON.stringify(obj);
  }
}
```

这同样适用于数字和字符串的字面量

例子

```
const Foo = "Foo";
const Bar = "Bar";
```

```
let x = {
  [Foo]: 100,
  [Bar]: "hello",
};
```

let a = x[Foo]; // a 类型为 'number'; 在之前版本，类型为 'number | string'，现在可以追踪到类型

let b = x[Bar]; // b 类型为 'string';

5.2 unique symbol 类型

为了将 symbol 变量视作有唯一值的字面量，我们新增了类型 unique symbol。unique symbol 是 symbol 的子类型，仅由调用 Symbol() 或 Symbol.for() 或明确的类型注释生成。该类型只允许在 const 声明或者 readonly static 属性声明中使用。如果要引用某个特定的 unique symbol 变量，你必须使用 typeof 操作符。每个对 unique symbols 的引用都意味着一个完全唯一的声明身份，与被引用的变量声明绑定。

例子

```
// Works
declare const Foo: unique symbol;
```

```
// Error! 'Bar'不是 const 声明的
let Bar: unique symbol = Symbol();
```

// Works - 对变量 Foo 的引用，它的声明身份与 Foo 绑定

```
let Baz: typeof Foo = Foo;
```

// Also works.

```
class C {
    static readonly StaticSymbol: unique symbol = Symbol();
}
```

因为每个 unique symbols 都有个完全独立的身份，因此两个 unique symbols 类型之间不能赋值或比较。

[Example](#)

```
const Foo = Symbol();
const Bar = Symbol();
```

// Error: 不能比较两个 unique symbols.

```
if (Foo === Bar) {
    // ...
}
```

5.3 更严格的类属性检查

TypeScript 2.7 引入了一个新的控制严格性的标记--

strictPropertyInitialization。使用这个标记后，TypeScript 要求类的所有实例属性在构造函数里或属性初始化器中都得到初始化。比如：

```
class C {
    foo: number;
    bar = "hello";
    baz: boolean;
}
// ~~~
// Error! Property 'baz' has no initializer and is not assigned
// directly in the constructor.
constructor() {
    this.foo = 42;
}
```

上例中，baz 从未被赋值，因此 TypeScript 报错了。如果我们的本意就是让 baz 可以为 undefined，那么应该声明它的类型为 boolean | undefined。

在某些场景下，属性会被间接地初始化（使用辅助方法或依赖注入库）。这种情况下，你可以在属性上使用_显式赋值断言_（definite assignment assertion modifiers）来帮助类型系统识别类型（下面会讨论）

```
class C {
    foo!: number;
    // ^
    // Notice this exclamation point!
    // This is the "definite assignment assertion" modifier.
    constructor() {
        this.initialize();
    }

    initialize() {
        this.foo = 0;
    }
}
```

```
}
}
```

注意，--strictPropertyInitialization 会在其它--strict 模式标记下被启用，这可能会影响你的工程。你可以在 tsconfig.json 的 compilerOptions 里将 strictPropertyInitialization 设置为 false，或者在命令行上将--strictPropertyInitialization 设置为 false 来关闭检查。

5.4 显式赋值断言

显式赋值断言允许你在实例属性和变量声明之后加一个感叹号!，来告诉 TypeScript 这个变量确实已被赋值，即使 TypeScript 不能分析出这个结果。

[例子](#)

```
let x: number;
initialize();
console.log(x + x);
//      ~ ~
// Error! Variable 'x' is used before being assigned.
```

```
function initialize() {
    x = 10;
}
```

使用显式类型断言在 x 的声明后加上!，Typescript 可以认为变量 x 确实已被赋值

```
// Notice the '!'
let x!: number;
initialize();
```

```
// No error!
console.log(x + x);
```

```
function initialize() {
    x = 10;
}
```

在某种意义上，显式类型断言运算符是非空断言运算符（在表达式后缀的!）的对偶，就像下面这个例子

```
let x: number;
initialize();
```

```
// No error!
console.log(x! + x!);
```

```
function initialize() {
    x = 10;
}
```

在上面的例子中，我们知道 x 都会被初始化，因此使用显式类型断言比使用非空断言更合适。

5.5 固定长度元组

TypeScript 2.6 之前，[number, string, string] 被当作 [number, string] 的子类型。这对于 TypeScript 的结构性而言是合理的——[number, string, string] 的前两个元素各自是 [number, string] 里前两个元素的子类型。但是，我们注意到在在实践中的大多数情形下，这并不是开发者所希望的。

6 v2.8

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-2.8.html>

6.1 有条件类型

TypeScript 2.8 引入了_有条件类型_，它能够表示非统一的类型。 有条件的类型会以一个条件表达式进行类型关系检测，从而在两种类型中选择其一：

T extends U ? X : Y

上面的类型意思是，若 T 能够赋值给 U，那么类型是 X，否则为 Y。

有条件的类型 T extends U ? X : Y 或者_解析_为 X，或者_解析_为 Y，又或者_延迟_解析，因为它可能依赖一个或多个类型变量。 是否直接解析或推迟取决于：

- 首先，令 T' 和 U' 分别为 T 和 U 的实例，并将所有类型参数替换为 any，如果 T' 不能赋值给 U'，则将有条件的类型解析成 Y。直观上讲，如果最宽泛的 T 的实例不能赋值给最宽泛的 U 的实例，那么我们就可以断定不存在可以赋值的实例，因此可以解析为 Y。
- 其次，针对每个在 U 内由推断声明引入的类型变量，依据从 T 推断到 U 来收集一组候选类型（使用与泛型函数类型推断相同的推断算法）。对于给定的推断类型变量 V，如果有候选类型是从协变的位置上推断出来的，那么 V 的类型是那些候选类型的联合。反之，如果有候选类型是从逆变的位置上推断出来的，那么 V 的类型是那些候选类型的交叉类型。否则 V 的类型是 never。
- 然后，令 T'' 为 T 的一个实例，所有推断的类型变量用上一步的推断结果替换，如果 T''_明显可赋值_给 U，那么将有条件的类型解析为 X。除去不考虑类型变量的限制之外，_明显可赋值_的关系与正常的赋值关系一致。直观上，当一个类型明显可赋值给另一个类型，我们就能够知道它可以赋值给那些类型的_所有_实例。
- 否则，这个条件依赖于一个或多个类型变量，有条件的类型解析被推迟进行。

例子

```
type TypeName<T> =
  T extends string ? "string" :
  T extends number ? "number" :
  T extends boolean ? "boolean" :
  T extends undefined ? "undefined" :
  T extends Function ? "function" :
  "object";
```

```
type T0 = TypeName<string>; // "string"
type T1 = TypeName<"a">; // "string"
type T2 = TypeName<true>; // "boolean"
type T3 = TypeName<() => void>; // "function"
type T4 = TypeName<string[]>; // "object"
```

6.1.1 分布式有条件类型

如果有条件类型里待检查的类型是 naked type parameter，那么它也被称为“分布式有条件类型”。 分布式有条件类型在实例化时会自动分成联合类型。 例如，实例化 T extends U ? X : Y，T 的类型为 A | B | C，会被解析为 (A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)。

例子

```
type T10 = TypeName<string | (() => void)>; // "string" | "function"
type T12 = TypeName<string | string[] | undefined>; // "string" |
"object" | "undefined"
```

例子

```
class A {}
class B extends A {}
class C extends A {}
class D extends A { c: string }
class E extends D {}
```

```
let x1 = !true ? new A() : new B(); // A
let x2 = !true ? new B() : new C(); // B | C (previously B)
let x3 = !true ? new C() : new D(); // C | D (previously C)
```

```
let a1 = [new A(), new B(), new C(), new D(), new E()]; // A[]
let a2 = [new B(), new C(), new D(), new E()]; // (B | C | D)[]
(previously B[])
```

```
function f1(x: B | C | D) {
  if (x instanceof B) {
    x; // B (previously B | D)
  }
  else if (x instanceof C) {
    x; // C
  }
  else {
    x; // D (previously never)
  }
}
```

5.8 in 运算符实现类型保护

in 运算符现在会起到类型细化的作用。

对于一个 n in x 的表达式，当 n 是一个字符串字面量或者字符串字面量类型，并且 x 是一个联合类型： 在值为 "true" 的分支中，x 会有一个推断出来可选或被赋值的属性 n；在值为 "false" 的分支中，x 根据推断仅有可选的属性 n 或没有属性 n。

例子

```
interface A { a: number };
interface B { b: string };
```

```
function foo(x: A | B) {
  if ("a" in x) {
    return x.a;
  }
  return x.b; // 此时 x 的类型推断为 B，属性 a 不存在
}
```

5.9 使用标记--esModuleInterop 引入非 ES 模块

在 TypeScript 2.7 使用--esModuleInterop 标记后，为_CommonJS/AMD/UMD_模块生成基于__esModule 指示器的命名空间记录。这次更新使得 TypeScript 编译后的输出与 Babel 的输出更加接近。

之前版本中，TypeScript 处理_CommonJS/AMD/UMD_模块的方式与处理 ES6 模块一致，导致了一些问题，比如：

- TypeScript 之前处理 CommonJS/AMD/UMD 模块的命名空间导入（如 `import * as foo from "foo"`）时等同于 `const foo = require("foo")`。这样做很简单，但如果引入的主要对象（比如这里的 `foo`）是基本类型、类或者函数，就有问题。ECMAScript 标准规定了命名空间记录是一个纯粹的对象，并且引入的命名空间（比如前面的 `foo`）应该是不可调用的，然而在 TypeScript 中却可以。
- 同样地，一个 CommonJS/AMD/UMD 模块的默认导入（如 `import d from "foo"`）被处理成等同于 `const d = require("foo").default` 的形式。然而现在大多数可用的 CommonJS/AMD/UMD 模块并没有默认导出，导致这种引入语句在实践中不适用于非 ES 模块。比如 `import fs from "fs" or import express from "express"` 都不可用。

在使用标签 `--esModuleInterop` 后，这两个问题都得到了解决：

- 命名空间导入（如 `import * as foo from "foo"`）的对象现在被修正为不可调用的。调用会报错。
- 对 CommonJS/AMD/UMD 模块可以使用默认导入（如 `import d from "foo"`）且能正常工作了。

注：这个新特性有可能对现有的代码产生破坏，因此以标记的方式引入。但无论是新项目还是之前的项目，**我们都强烈建议使用它**。对于之前的项目，命名空间导入（`import * as express from "express"; express();`）需要被改写成默认引入（`import express from "express"; express();`）。

[例子](#)

使用 `--esModuleInterop` 后，会生成两个新的辅助

量 `__importStar` and `__importDefault`，分别对应导入 `*` 和导入 `default`，比如这样的输入：

```
import * as foo from "foo";
import b from "bar";

会生成：
"use strict";
var __importStar = (this && this.__importStar) || function (mod) {
    if (mod && mod.__esModule) return mod;
    var result = {};
    if (mod != null) for (var k in mod) if
(Object.hasOwnProperty.call(mod, k)) result[k] = mod[k];
    result["default"] = mod;
    return result;
}
var __importDefault = (this && this.__importDefault) || function (mod)
{
    return (mod && mod.__esModule) ? mod : { "default": mod };
}
exports.__esModule = true;
var foo = __importStar(require("foo"));
var bar_1 = __importDefault(require("bar"));
```

5.9.1 数字分隔符

TypeScript 2.7 支持 ECMAScript 的[数字分隔符提案](#)。这个特性允许用户在数字之间使用下划线 `_` 来对数字分组。

```
const million = 1_000_000;
const phone = 555_734_2231;
const bytes = 0xFF_0C_00_FF;
```

```
const word = 0b1100_0011_1101_0001;
```

5.9.2 --watch 模式下具有更简洁的输出

在 TypeScript 的 `--watch` 模式下进行重新编译后会清屏。这样就更方便阅读最近这次编译的输出信息。

5.9.3 更漂亮的 --pretty 输出

TypeScript 的 `--pretty` 标记可以让错误信息更易阅读和管理。我们对这个功能进行了两个主要的改进。首先，`--pretty` 对文件名，诊段代码和行数添加了颜色（感谢 Joshua Goldberg）。其次，格式化了文件名和位置，以便于在常用的终端里使用 `Ctrl+Click`, `Cmd+Click`, `Alt+Click` 等来跳转到编译器里的相应位置。


```
class C {
    x = 0;
    y = 0;
}

type T10 = ReturnType<() => string>; // string
type T11 = ReturnType<(s: string) => void>; // void
type T12 = ReturnType<(<T>() => T)>; // {}
type T13 = ReturnType<(<T extends U, U extends number[]>() => T)>; //
number[]
type T14 = ReturnType<typeof f1>; // { a: number, b: string }
type T15 = ReturnType<any>; // any
type T16 = ReturnType<never>; // any
type T17 = ReturnType<string>; // Error
type T18 = ReturnType<Function>; // Error
```

```
type T20 = InstanceType<typeof C>; // C
type T21 = InstanceType<any>; // any
type T22 = InstanceType<never>; // any
type T23 = InstanceType<string>; // Error
type T24 = InstanceType<Function>; // Error
注意: Exclude 类型是建议的Diff 类型的一种实现。我们使用 Exclude 这个名字是为了避免
破坏已经定义了 Diff 的代码, 并且我们感觉这个名字能更好地表达类型的语义。我们没有增加
Omit<T, K>类型, 因为它可以很容易的用 Pick<T, Exclude<keyof T, K>>来表示。
```

6.2 改进对映射类型修饰符的控制

映射类型支持在属性上添加 readonly 或?修饰符, 但是它们不支持_移除_修饰符。这对于[同态映射类型](#)有些影响, 因为同态映射类型默认保留底层类型的修饰符。TypeScript 2.8 为映射类型增加了增加或移除特定修饰符的能力。特别地, 映射类型里的 readonly 或?属性修饰符现在可以使用+或-前缀, 来表示修饰符是添加还是移除。

```
例子
type MutableRequired<T> = { -readonly [P in keyof T]-?: T[P] }; // 移
除 readonly 和?
type ReadonlyPartial<T> = { +readonly [P in keyof T]+?: T[P] }; // 添
加 readonly 和?
不带+或-前缀的修饰符与带+前缀的修饰符具有相同的作用。因此上面的
ReadonlyPartial<T>类型与下面的一致
type ReadonlyPartial<T> = { readonly [P in keyof T]?: T[P] }; // 添加
readonly 和?
利用这个特性, lib.d.ts 现在有了一个新的 Required<T>类型。它移除了T的所有属性的?
修饰符, 因此所有属性都是必需的。
```

```
例子
type Required<T> = { [P in keyof T]-?: T[P] };
注意在--strictNullChecks 模式下, 当同态映射类型移除了属性底层类型的?修饰符, 它同
时也移除了那个属性上的 undefined 类型:
```

```
例子
type Foo = { a?: string }; // 等同于 { a?: string | undefined }
type Bar = Required<Foo>; // 等同于 { a: string }
```

type T11 = TypeName<string[] | number[]>; // "object"
在T extends U ? X : Y 的实例化里, 对T的引用被解析为联合类型的一部分(比如, T指
向某一个部分, 在有条件类型分布到联合类型之后)。此外, 在X内对T的引用有一个附加
的类型参数约束U(例如, T被当成在X内可赋值给U)。

```
例子
type BoxedValue<T> = { value: T };
type BoxedArray<T> = { array: T[] };
type Boxed<T> = T extends any[] ? BoxedArray<T[number]> :
BoxedValue<T>;
```

```
type T20 = Boxed<string>; // BoxedValue<string>;
type T21 = Boxed<number[]>; // BoxedArray<number>;
type T22 = Boxed<string | number[]>; // BoxedValue<string> |
BoxedArray<number>;
注意在 Boxed<T>的 true 分支里, T 有个额外的约束 any[], 因此它适用于 T[number] 数组
元素类型。同时也注意一下有条件类型是如何分布成联合类型的。
有条件类型的分布式的属性可以方便地用来_过滤_联合类型:
type Diff<T, U> = T extends U ? never : T; // Remove types from T
that are assignable to U
type Filter<T, U> = T extends U ? T : never; // Remove types from T
that are not assignable to U
```

```
type T30 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T31 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" |
"c"
type T32 = Diff<string | number | (() => void), Function>; // string
| number
type T33 = Filter<string | number | (() => void), Function>; // () =>
void
```

```
type NonNullable<T> = Diff<T, null | undefined>; // Remove null and
undefined from T
```

```
type T34 = NonNullable<string | number | undefined>; // string |
number
type T35 = NonNullable<string | string[] | null | undefined>; //
string | string[]
```

```
function f1<T>(x: T, y: NonNullable<T>) {
    x = y; // Ok
    y = x; // Error
}
```

```
function f2<T extends string | undefined>(x: T, y: NonNullable<T>) {
    x = y; // Ok
    y = x; // Error
    let s1: string = x; // Error
    let s2: string = y; // Ok
}
```

有条件类型与映射类型结合时特别有用：

```
type FunctionPropertyNames<T> = { [K in keyof T]: T[K] extends
Function ? K : never }[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = { [K in keyof T]: T[K] extends
Function ? never : K }[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;
```

```
interface Part {
  id: number;
  name: string;
  subparts: Part[];
  updatePart(newName: string): void;
}
```

```
type T40 = FunctionPropertyNames<Part>; // "updatePart"
type T41 = NonFunctionPropertyNames<Part>; // "id" | "name" |
"subparts"
type T42 = FunctionProperties<Part>; // { updatePart(newName:
string): void }
type T43 = NonFunctionProperties<Part>; // { id: number, name:
string, subparts: Part[] }
```

与联合类型和交叉类型相似，有条件类型不允许递归地引用自己。比如下面的错误。

[例子](#)

```
type ElementType<T> = T extends any[] ? ElementType<T[number]> : T;
// Error
```

6.1.2 有条件类型中的类型推断

现在在有条件类型的 extends 子语句中，允许出现 infer 声明，它会引入一个待推断的类型变量。这个推断的类型变量可以在有条件类型的 true 分支中被引用。允许出现多个同类型变量的 infer。

例如，下面代码会提取函数类型的返回值类型：

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
有条件类型可以嵌套来构成一系列的匹配模式，按顺序进行求值：
type Unpacked<T> =
  T extends (infer U)[] ? U :
  T extends (...args: any[]) => infer U ? U :
  T extends Promise<infer U> ? U :
  T;
```

```
type T0 = Unpacked<string>; // string
type T1 = Unpacked<string[]>; // string
type T2 = Unpacked<() => string>; // string
type T3 = Unpacked<Promise<string>>; // string
type T4 = Unpacked<Promise<string>[]>; // Promise<string>
type T5 = Unpacked<Unpacked<Promise<string>[]>>; // string
下面的例子解释了在协变位置上，同一个类型变量的多个候选类型会被推断为联合类型：
type Foo<T> = T extends { a: infer U, b: infer U } ? U : never;
```

```
type T10 = Foo<{ a: string, b: string }>; // string
type T11 = Foo<{ a: string, b: number }>; // string | number
相似地，在抗变位置上，同一个类型变量的多个候选类型会被推断为交叉类型：
type Bar<T> = T extends { a: (x: infer U) => void, b: (x: infer U) =>
void } ? U : never;
type T20 = Bar<{ a: (x: string) => void, b: (x: string) => void }>;
// string
type T21 = Bar<{ a: (x: string) => void, b: (x: number) => void }>;
// string & number
当推断具有多个调用签名（例如函数重载类型）的类型时，用_最后_的签名（大概是最自由的包含所有情况的签名）进行推断。无法根据参数类型列表来解析重载。
declare function foo(x: string): number;
declare function foo(x: number): string;
declare function foo(x: string | number): string | number;
type T30 = ReturnType<typeof foo>; // string | number
无法在正常类型参数的约束子语句中使用 infer 声明：
type ReturnType<T extends (...args: any[]) => infer R> = R; // 错误，不支持
```

但是，可以这样达到同样的效果，在约束里删掉类型变量，用有条件类型替换：

```
type AnyFunction = (...args: any[]) => any;
type ReturnType<T extends AnyFunction> = T extends (...args: any[]) =>
infer R ? R : any;
```

6.1.3 预定义的有条件类型

TypeScript 2.8 在 lib.d.ts 里增加了一些预定义的有条件类型：

- Exclude<T, U> -- 从 T 中剔除可以赋值给 U 的类型。
- Extract<T, U> -- 提取 T 中可以赋值给 U 的类型。
- NonNullable<T> -- 从 T 中剔除 null 和 undefined。
- ReturnType<T> -- 获取函数返回值类型。
- InstanceType<T> -- 获取构造函数类型的实例类型。

[Example](#)

```
type T00 = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" |
"d"
type T01 = Extract<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" |
"c"

type T02 = Exclude<string | number | (() => void), Function>; //
string | number
type T03 = Extract<string | number | (() => void), Function>; // ()
=> void
```

```
type T04 = NonNullable<string | number | undefined>; // string |
number
type T05 = NonNullable<(() => string) | string[] | null | undefined>;
// (() => string) | string[]

function f1(s: string) {
  return { a: 1, b: s };
}
```

```
ts_upgrade_from_2.2_to_5.3
[E2.A]: string; // String-like name
}

type K1 = keyof Foo; // "a" | 5 | "c" | 10 | typeof e | E1.A | E2.A
type K2 = Extract<keyof Foo, string>; // "a" | "c" | E2.A
type K3 = Extract<keyof Foo, number>; // 5 | 10 | E1.A
type K4 = Extract<keyof Foo, symbol>; // typeof e
现在通过在键值类型里包含 number 类型, keyof 就能反映出数字索引签名的存在, 因此像
Partial<T>和 Readonly<T>的映射类型能够正确地处理带数字索引签名的对象类型:
type Arrayish<T> = {
    length: number;
    [x: number]: T;
}

type ReadonlyArrayish<T> = Readonly<Arrayish<T>>;

declare const map: ReadonlyArrayish<string>;
let n = map.length;
let x = map[123]; // Previously of type any (or an error with --noImplicitAny)
此外, 由于 keyof 支持用 number 和 symbol 命名的键值, 现在可以对对象的数字字面量 (如数字枚举类型) 和唯一的 symbol 属性的访问进行抽象。
const enum Enum { A, B, C }
```

```
type ReadonlyArrayish<T> = Readonly<Arrayish<T>>;
```

```
declare const map: ReadonlyArrayish<string>;
let n = map.length;
let x = map[123]; // Previously of type any (or an error with --noImplicitAny)
此外, 由于 keyof 支持用 number 和 symbol 命名的键值, 现在可以对对象的数字字面量 (如数字枚举类型) 和唯一的 symbol 属性的访问进行抽象。
const enum Enum { A, B, C }
```

```
const enumToStringMap = {
    [Enum.A]: "Name A",
    [Enum.B]: "Name B",
    [Enum.C]: "Name C"
}
```

```
const sym1 = Symbol();
const sym2 = Symbol();
const sym3 = Symbol();
```

```
const symbolToNumberMap = {
    [sym1]: 1,
    [sym2]: 2,
    [sym3]: 3
};
```

```
type KE = keyof typeof enumToStringMap; // Enum (i.e. Enum.A | Enum.B | Enum.C)
type KS = keyof typeof symbolToNumberMap; // typeof sym1 | typeof sym2 | typeof sym3
```

```
function getValue<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}
```

6.3 改进交叉类型上的 keyof

TypeScript 2.8 作用于交叉类型的 keyof 被转换成作用于交叉成员的 keyof 的联合。换句话说, keyof (A & B) 会被转换成 keyof A | keyof B。这个改动应该能够解决 keyof 表达式推断不一致的问题。

例子

```
type A = { a: string };
type B = { b: string };
```

```
type T1 = keyof (A & B); // "a" | "b"
type T2<T> = keyof (T & B); // keyof T | "b"
type T3<U> = keyof (A & U); // "a" | keyof U
type T4<T, U> = keyof (T & U); // keyof T | keyof U
type T5 = T2<A>; // "a" | "b"
type T6 = T3<B>; // "a" | "b"
type T7 = T4<A, B>; // "a" | "b"
```

6.4 更好的处理 .js 文件中的命名空间模式

TypeScript 2.8 加强了识别 .js 文件里的命名空间模式。JavaScript 顶层的空对象字面量声明, 就像函数和类, 会被识别成命名空间声明。

```
var ns = {}; // recognized as a declaration for a namespace `ns`
ns.constant = 1; // recognized as a declaration for var `constant`
顶层的赋值应该有一致的行为; 也就是说, var 或 const 声明不是必需的。
```

```
app = {}; // does NOT need to be `var app = {}`
```

```
app.C = class {
};
app.f = function() {
};
app.prop = 1;
```

6.4.1 立即执行的函数表达式做为命名空间

立即执行的函数表达式返回一个函数, 类或空的对象字面量, 也会被识别为命名空间:

```
var C = (function () {
    function C(n) {
        this.p = n;
    }
    return C;
})();
```

```
C.staticProperty = 1;
```

6.4.2 默认声明

“默认声明”允许引用了声明的名称的初始化器出现在逻辑或的左边:

```
my = window.my || {};
my.app = my.app || {};
```

6.4.3 原型赋值

你可以把一个对象字面量直接赋值给原型属性。独立的原型赋值也可以:

```
var C = function (p) {
    this.p = p;
};
C.prototype = {
    m() {
        console.log(this.p);
    }
}
```

```
ts_upgrade_from_2.2_to_5.3
  }
};
C.prototype.q = function(r) {
  return this.p === r;
};
```

6.4.4 嵌套与合并声明

现在嵌套的层次不受限制，并且多文件之间的声明合并也没有问题。以前不是这样的。

```
var app = window.app || {};
app.C = class { };
```

6.5 各文件的 JSX 工厂

TypeScript 2.8 增加了使用@jsx dom 指令为每个文件设置 JSX 工厂名。JSX 工厂也可以使用--jsxFactory 编译参数设置（默认值为 React.createElement）。TypeScript 2.8 你可以基于文件进行覆写。

例子

```
/** @jsx dom */
import { dom } from "./renderer"
<h></h>
生成:
var renderer_1 = require("./renderer");
renderer_1.dom("h", null);
```

6.6 本地范围的 JSX 命名空间

JSX 类型检查基于 JSX 命名空间里的定义，比如 JSX.Element 用于 JSX 元素的类型，JSX.IntrinsicElements 用于内置的元素。在 TypeScript 2.8 之前 JSX 命名空间被视为全局命名空间，并且一个工程只允许存在一个。TypeScript 2.8 开始，JSX 命名空间将在 jsxNamespace 下面查找（比如 React），允许在一次编译中存在多个 jsx 工厂。为了向后兼容，全局的 JSX 命名空间被当做回退选项。使用独立的@jsx 指令，每个文件可以有自己的 JSX 工厂。

6.7 新的--emitDeclarationsOnly

--emitDeclarationsOnly 允许_仅_生成声明文件；使用这个标记.js/.jsx 输出会被跳过。当使用其它的转换工具如 Babel 处理.js 输出的时候，可以使用这个标记。

7 v2.9

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-2.9.html>

7.1 keyof 和映射类型支持用 number 和 symbol 命名的属性

TypeScript 2.9 增加了在索引类型和映射类型上支持用 number 和 symbol 命名属性。在之前，keyof 操作符和映射类型只支持 string 命名的属性。

改动包括：

- 对某些类型 T，索引类型 keyof T 是 string | number | symbol 的子类型。
- 映射类型{ [P in K]: XXX }，其中 K 允许是可以赋值给 string | number | symbol 的任何值。
- 针对泛型 T 的对象的 for...in 语句，迭代变量推断类型之前为 keyof T，现在是 Extract<keyof T, string>。（换句话说，是 keyof T 的子集，它仅包含类字符串的值。）

对于对象类型 X，keyof X 将按以下方式解析：

- 如果 X 带有字符串索引签名，则 keyof X 为 string，number 和表示 symbol-like 属性的字面量类型的联合，否则
- 如果 X 带有数字索引签名，则 keyof X 为 number 和表示 string-like 和 symbol-like 属性的字面量类型的联合，否则
- keyof X 为表示 string-like，number-like 和 symbol-like 属性的字面量类型的联合。

在何处：

- 对象类型的 string-like 属性，是那些使用标识符，字符串字面量或计算后值为字符串字面量类型的属性名所声明的。
- 对象类型的 number-like 属性是那些使用数字字面量或计算后值为数字字面量类型的属性名所声明的。
- 对象类型的 symbol-like 属性是那些使用计算后值为 symbol 字面量类型的属性名所声明的。

对于映射类型{ [P in K]: XXX }，K 的每个字符串字面量类型都会引入一个名字为字符串的属性，K 的每个数字字面量类型都会引入一个名字为数字的属性，K 的每个 symbol 字面量类型都会引入一个名字为 symbol 的属性。并且，如果 K 包含 string 类型，那个同时也会引入字符串索引类型，如果 K 包含 number 类型，那个同时也会引入数字索引类型。

例子

```
const c = "c";
const d = 10;
const e = Symbol();
```

```
const enum E1 { A, B, C }
const enum E2 { A = "A", B = "B", C = "C" }
```

```
type Foo = {
  a: string;      // String-like name
  5: string;      // Number-like name
  [c]: string;    // String-like name
  [d]: string;    // Number-like name
  [e]: string;    // Symbol-like name
  [E1.A]: string; // Number-like name
```

7.9 新的`--declarationMap`

随着`--declaration`一起启用`--declarationMap`，编译器在生成`.d.ts`的同时还会生成`.d.ts.map`。语言服务现在也能够理解这些`map`文件，将声明文件映射到源码。换句话说，在启用了`--declarationMap`后生成的`.d.ts`文件里点击`go-to-definition`，将会导航到源文件里的位置（`.ts`），而不是导航到`.d.ts`文件里。

ts_upgrade_from_2.2_to_5.3

```
let x1 = getValue(enumToStringMap, Enum.C); // Returns "Name C"
let x2 = getValue(symbolToNumberMap, sym3); // Returns 3
```

这是一个破坏性改动；之前，`keyof`操作符和映射类型只支持`string`命名的属性。那些把总是把`keyof T`的类型当做`string`的代码现在会报错。

例子

```
function useKey<T, K extends keyof T>(o: T, k: K) {
  var name: string = k; // 错误: keyof T 不能赋值给字符串
}
```

推荐

- 如果函数只能处理字符串命名属性的键，在声明里使用`Extract<keyof T, string>`:
- `function useKey<T, K extends Extract<keyof T, string>>(o: T, k: K) {`
- `var name: string = k; // OK`
- `}`
- 如果函数能处理任何属性的键，那么可以在下游进行改动:
- `function useKey<T, K extends keyof T>(o: T, k: K) {`
- `var name: string | number | symbol = k;`
- `}`
- 否则，使用`--keyofStringsOnly`编译器选项来禁用新的行为。

7.2 JSX 元素里的泛型参数

JSX 元素现在允许传入类型参数到泛型组件里。

例子

```
class GenericComponent<P> extends React.Component<P> {
  internalProp: P;
}
```

```
type Props = { a: number; b: string; };
```

```
const x = <GenericComponent<Props> a={10} b="hi"/>; // OK
```

```
const y = <GenericComponent<Props> a={10} b={20} />; // Error
```

7.3 泛型标记模版里的泛型参数

标记模版是 ECMAScript 2015 引入的一种调用形式。类似调用表达式，可以在标记模版里使用泛型函数，TypeScript 会推断使用的类型参数。TypeScript 2.9 允许传入泛型参数到标记模版字符串。

例子

```
declare function styledComponent<Props>(strs: TemplateStringsArray):
Component<Props>;
```

```
interface MyProps {
  name: string;
  age: number;
}
```

```
styledComponent<MyProps> `
  font-size: 1.5em;
```



```
ts_upgrade_from_2.2_to_5.3
  text-align: center;
  color: palevioletred;
`;
```

```
declare function tag<T>(strs: TemplateStringsArray, ...args: T[]): T;
```

```
// inference fails because 'number' and 'string' are both candidates
that conflict
```

```
let a = tag<string | number> `${100} ${"hello"}`;
```

7.4 import 类型

模块可以导入在其它模块里声明的类型。但是非模块的全局脚本不能访问模块里声明的类型。这里，import 类型登场了。

在类型注释的位置使用 import("mod"), 就可以访问一个模块和它导出的声明，而不必导入它。

例子

在一个模块文件里，有一个 Pet 类的声明：

```
// module.d.ts
```

```
export declare class Pet {
  name: string;
}
```

它可以被用在非模块文件 global-script.ts:

```
// global-script.ts
```

```
function adopt(p: import("./module").Pet) {
  console.log(`Adopting ${p.name}...`);
}
```

它也可以被放在 .js 文件的 JSDoc 注释里，来引用模块里的类型：

```
// a.js
```

```
/**
 * @param p { import("./module").Pet }
 */
```

```
function walk(p) {
  console.log(`Walking ${p.name}...`);
}
```

7.5 放开声明生成时可见性规则

随着 import 类型的到来，许多在声明文件生成阶段报的可见性错误可以被编译器正确地处理，而不需要改变输入。

例如：

```
import { createHash } from "crypto";
```

```
export const hash = createHash("sha256");
```

```
//      ^^^^^
```

// Exported variable 'hash' has or is using name 'Hash' from external module 'crypto' but cannot be named.

TypeScript 2.9 不会报错，生成文件如下：

```
export declare const hash: import("crypto").Hash;
```

7.6 支持 import.meta

TypeScript 2.9 引入对 import.meta 的支持，它是当前 [TC39 建议](#) 里的一个元属性。

import.meta 类型是全局的 ImportMeta 类型，它在 lib.es5.d.ts 里定义。这个接口地使用十分有限。添加众所周知的 Node 和浏览器属性需要进行接口合并，还有可能需要根据上下文来增加全局空间。

例子

假设 __dirname 永远存在于 import.meta，那么可以通过重新开放 ImportMeta 接口来进行声明：

```
// node.d.ts
interface ImportMeta {
  __dirname: string;
}
```

用法如下：

```
import.meta.__dirname // Has type 'string'
```

import.meta 仅在输出目标为 ESNEXT 模块和 ECMAScript 时才生效。

7.7 新的 --resolveJsonModule

在 Node.js 应用里经常需要使用 .json。TypeScript 2.9 的 --resolveJsonModule 允许从 .json 文件里导入，获取类型。

例子

```
// settings.json
```

```
{
  "repo": "TypeScript",
  "dry": false,
  "debug": false
}
// a.ts
```

```
import settings from "./settings.json";
```

```
settings.debug === true; // OK
settings.dry === 2; // Error: Operator '===' cannot be applied
boolean and number
// tsconfig.json
```

```
{
  "compilerOptions": {
    "module": "commonjs",
    "resolveJsonModule": true,
    "esModuleInterop": true
  }
}
```

7.8 默认 --pretty 输出

从 TypeScript 2.9 开始，如果应用支持彩色文字，那么错误输出时会默认应用 --pretty。

TypeScript 会检查输出流是否设置了 isTTY 属性。

使用 --pretty false 命令行选项或 tsconfig.json 里设置 "pretty": false 来禁用 --pretty 输出。


```
ts_upgrade_from_2.2_to_5.3
x.foo; // Error
x[5]; // Error
x(); // Error
new x(); // Error
}
```

```
// typeof, instanceof, and user defined type predicates
```

```
declare function isFunction(x: unknown): x is Function;
```

```
function f20(x: unknown) {
  if (typeof x === "string" || typeof x === "number") {
    x; // string | number
  }
  if (x instanceof Error) {
    x; // Error
  }
  if (isFunction(x)) {
    x; // Function
  }
}
```

```
// Homomorphic mapped type over unknown
```

```
type T50<T> = { [P in keyof T]: number };
type T51 = T50<any>; // { [x: string]: number }
type T52 = T50<unknown>; // {}
```

```
// Anything is assignable to unknown
```

```
function f21<T>(pAny: any, pNever: never, pT: T) {
  let x: unknown;
  x = 123;
  x = "hello";
  x = [1, 2, 3];
  x = new Error();
  x = x;
  x = pAny;
  x = pNever;
  x = pT;
}
```

```
// unknown assignable only to itself and any
```

```
function f22(x: unknown) {
  let v1: any = x;
  let v2: unknown = x;
  let v3: object = x; // Error
  let v4: string = x; // Error
}
```

8 v3.0

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.0.html>

8.1 工程引用

TypeScript 3.0 引入了一个叫做工程引用的新概念。工程引用允许 TypeScript 工程依赖于其它 TypeScript 工程 - 特别要提的是允许 tsconfig.json 文件引用其它 tsconfig.json 文件。当指明了这些依赖后，就可以方便地将代码分割成单独的小工程，有助于 TypeScript（以及周边的工具）了解构建顺序和输出结构。

TypeScript 3.0 还引入了一种新的 tsc 模式，即--build 标记，它与工程引用同时运用可以加速构建 TypeScript。

相关详情请阅读[工程引用手册](#)。

8.2 剩余参数和展开表达式里的元组

TypeScript 3.0 增加了支持以元组类型与函数参数列表进行交互的能力。如下：

- [将带有元组类型的剩余参数扩展为离散参数](#)
- [将带有元组类型的展开表达式扩展为离散参数](#)
- [泛型剩余参数以及相应的元组类型推断](#)
- [元组类型里的可选元素](#)
- [元组类型里的剩余元素](#)

有了这些特性后，便有可能将转换函数和它们参数列表的高阶函数变为强类型的。

8.2.1 带元组类型的剩余参数

当剩余参数里有元组类型时，元组类型被扩展为离散参数序列。例如，如下两个声明是等价的：

```
declare function foo(...args: [number, string, boolean]): void;
declare function foo(args_0: number, args_1: string, args_2: boolean): void;
```

8.2.2 带有元组类型的展开表达式

在函数调用中，若最后一个参数是元组类型的展开表达式，那么这个展开表达式相当于元组元素类型的离散参数序列。

因此，下面的调用都是等价的：

```
const args: [number, string, boolean] = [42, "hello", true];
foo(42, "hello", true);
foo(args[0], args[1], args[2]);
foo(...args);
```

8.2.3 泛型剩余参数

剩余参数允许带有泛型类型，这个泛型类型被限制为是一个数组类型，类型推断系统能够推断这类泛型剩余参数里的元组类型。这样就可以进行高阶捕获和展开部分参数列表：

[例子](#)

```
declare function bind<T, U extends any[], V>(f: (x: T, ...args: U) => V, x: T): (...args: U) => V;
```

```
declare function f3(x: number, y: string, z: boolean): void;
```

```
const f2 = bind(f3, 42); // (y: string, z: boolean) => void
const f1 = bind(f2, "hello"); // (z: boolean) => void
const f0 = bind(f1, true); // () => void
```

```
ts_upgrade_from_2.2_to_5.3
f3(42, "hello", true);
f2("hello", true);
f1(true);
f0();
```

上例的 f2 声明，类型推断可以推断出 number, [string, boolean] 和 void 做为 T, U 和 V。

注意，如果元组类型是从参数序列中推断出来的，之后又扩展成参数列表，就像 U 那样，原来的参数名称会被用在扩展中（然而，这个名字没有语义上的意义且是察觉不到的）。

8.2.4 元组类型里的可选元素

元组类型现在允许在其元素类型上使用 ? 后缀，表示这个元素是可选的：

[例子](#)

```
let t: [number, string?, boolean?];
t = [42, "hello", true];
t = [42, "hello"];
t = [42];
```

在 --strictNullChecks 模式下，? 修饰符会自动地在元素类型中包含 undefined，类似于可选参数。

在元组类型的一个元素类型上使用 ? 后缀修饰符来把它标记为可忽略的元素，且它右侧所有元素也同时带有了 ? 修饰符。

当剩余参数推断为元组类型时，源码中的可选参数在推断出的类型里成为了可选元组元素。带有可选元素的元组类型的 length 属性是表示可能长度的数字字面量类型的联合类型。例如，[number, string?, boolean?] 元组类型的 length 属性的类型是 1 | 2 | 3。

8.2.5 元组类型里的剩余元素

元组类型里最后一个元素可以是剩余元素，形式为 ...X，这里 X 是数组类型。剩余元素代表元组类型是开放的，可以有零个或多个额外的元素。例如，[number, ...string[]] 表示带有一个 number 元素和任意数量 string 类型元素的元组类型。

[例子](#)

```
function tuple<T extends any[]>(...args: T): T {
    return args;
}
```

```
const numbers: number[] = getArrayOfNumbers();
const t1 = tuple("foo", 1, true); // [string, number, boolean]
const t2 = tuple("bar", ...numbers); // [string, ...number[]]
```

这个带有剩余元素的元组类型的 length 属性类型是 number。

8.3 新的 unknown 类型

TypeScript 3.0 引入了一个顶级的 unknown 类型。对照于 any，unknown 是类型安全的。任何值都可以赋给 unknown，但是当没有类型断言或基于控制流的类型细化时 unknown 不可以赋值给其它类型，除了它自己和 any 外。同样地，在 unknown 没有被断言或细化到一个确切类型之前，是不允许在其上进行任何操作的。

[例子](#)

```
// In an intersection everything absorbs unknown
```

```
type T00 = unknown & null; // null
type T01 = unknown & undefined; // undefined
type T02 = unknown & null & undefined; // null & undefined (which
becomes never)
type T03 = unknown & string; // string
```

```
type T04 = unknown & string[]; // string[]
type T05 = unknown & unknown; // unknown
type T06 = unknown & any; // any
```

```
// In a union an unknown absorbs everything
```

```
type T10 = unknown | null; // unknown
type T11 = unknown | undefined; // unknown
type T12 = unknown | null | undefined; // unknown
type T13 = unknown | string; // unknown
type T14 = unknown | string[]; // unknown
type T15 = unknown | unknown; // unknown
type T16 = unknown | any; // any
```

```
// Type variable and unknown in union and intersection
```

```
type T20<T> = T & {}; // T & {}
type T21<T> = T | {}; // T | {}
type T22<T> = T & unknown; // T
type T23<T> = T | unknown; // unknown
```

```
// unknown in conditional types
```

```
type T30<T> = unknown extends T ? true : false; // Deferred
type T31<T> = T extends unknown ? true : false; // Deferred (so it
distributes)
type T32<T> = never extends T ? true : false; // true
type T33<T> = T extends never ? true : false; // Deferred
```

```
// keyof unknown
```

```
type T40 = keyof any; // string | number | symbol
type T41 = keyof unknown; // never
```

```
// Only equality operators are allowed with unknown
```

```
function f10(x: unknown) {
    x == 5;
    x !== 10;
    x >= 0; // Error
    x + 1; // Error
    x * 2; // Error
    -x; // Error
    +x; // Error
}
```

```
// No property accesses, element accesses, or function calls
```

```
function f11(x: unknown) {
```

9 v3.1

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.1.html>

9.1 元组和数组上的映射类型

TypeScript 3.1, 在元组和数组上的映射对象类型现在会生成新的元组/数组, 而非创建一个新的类型并且这个类型上具有如 `push()`, `pop()` 和 `length` 这样的成员。例子:

```
type MapToPromise<T> = { [K in keyof T]: Promise<T[K]> };
```

```
type Coordinate = [number, number]
```

```
type PromiseCoordinate = MapToPromise<Coordinate>; //
[Promise<number>, Promise<number>]
```

`MapToPromise` 接收参数 `T`, 当它是个像 `Coordinate` 这样的元组时, 只有数值型属性会被转换。 `[number, number]` 具有两个数值型属性: `0` 和 `1`。 针对这样的数组, `MapToPromise` 会创建一个新的元组, `0` 和 `1` 属性是原类型的一个 `Promise`。 因此 `PromiseCoordinate` 的类型为 `[Promise<number>, Promise<number>]`。

9.2 函数上的属性声明

TypeScript 3.1 提供了在函数声明上定义属性的能力, 还支持 `const` 声明的函数。只需要在函数直接给属性赋值就可以了。 这样我们就可以规范 JavaScript 代码, 不必再借助于 `namespace`。 例子:

```
function readImage(path: string, callback: (err: any, image: Image) =>
void) {
    // ...
}
```

```
readImage.sync = (path: string) => {
    const contents = fs.readFileSync(path);
    return decodeImageSync(contents);
}
```

这里, `readImage` 函数异步地读取一张图片。 此外, 我们还在 `readImage` 上提供了一个便捷的函数 `readImage.sync`。

一般来说, 使用 ECMAScript 导出是个更好的方式, 但这个新功能支持此风格的代码能够在 TypeScript 里执行。 此外, 这种属性声明的方式允许我们表达一些常见的模式, 例如 React 函数组件 (之前叫做 SFC) 里的 `defaultProps` 和 `propTypes`。

```
export const FooComponent = ({ name }) => (
    <div>Hello! I am {name}</div>
);
```

```
FooComponent.defaultProps = {
    name: "(anonymous)",
};
```

[1] 更确切地说, 是上面那种同态映射类型。

9.3 使用 `typesVersions` 选择版本

由社区的反馈还有我们的经验得知, 利用最新的 TypeScript 功能的同时容纳旧版本的用户很困难。 TypeScript 引入了叫做 `typesVersions` 的新特性来解决这种情况。

```
let v5: string[] = x; // Error
let v6: {} = x; // Error
let v7: {} | null | undefined = x; // Error
}
```

```
// Type parameter 'T extends unknown' not related to object
```

```
function f23<T extends unknown>(x: T) {
    let y: object = x; // Error
}
```

```
// Anything but primitive assignable to { [x: string]: unknown }
```

```
function f24(x: { [x: string]: unknown }) {
    x = {};
    x = { a: 5 };
    x = [1, 2, 3];
    x = 123; // Error
}
```

```
// Locals of type unknown always considered initialized
```

```
function f25() {
    let x: unknown;
    let y = x;
}
```

```
// Spread of unknown causes result to be unknown
```

```
function f26(x: {}, y: unknown, z: any) {
    let o1 = { a: 42, ...x }; // { a: number }
    let o2 = { a: 42, ...x, ...y }; // unknown
    let o3 = { a: 42, ...x, ...y, ...z }; // any
}
```

```
// Functions with unknown return type don't need return expressions
```

```
function f27(): unknown {
}
```

```
// Rest type cannot be created from unknown
```

```
function f28(x: unknown) {
    let { ...a } = x; // Error
}
```

```
// Class properties of type unknown don't need definite assignment
```

```
class C1 {
```

```

a: string; // Error
b: unknown;
c: any;
}

```

8.4 在 JSX 里支持 defaultProps

TypeScript 2.9 和之前的版本不支持在 JSX 组件里使用 [React 的 defaultProps](#) 声明。用户通常不得不将属性声明为可选的，然后在 render 里使用非 null 的断言，或者在导出之前对组件的类型使用类型断言。

TypeScript 3.0 在 JSX 命名空间里支持一个新的类型别名

LibraryManagedAttributes。这个助手类型定义了检查 JSX 表达式之前在组件 Props 上的一个类型转换；因此我们可以进行定制：如何处理提供的 props 与推断 props 之间的冲突，推断如何映射，如何处理可选性以及不同位置的推断如何结合在一起。

我们可以利用它来处理 React 的 defaultProps 以及 propTypes。

```

export interface Props {
  name: string;
}

export class Greet extends React.Component<Props> {
  render() {
    const { name } = this.props;
    return <div>Hello {name.toUpperCase()}!</div>;
  }
  static defaultProps = { name: "world" };
}

```

// Type-checks! No type assertions needed!

```
let el = <Greet />
```

8.4.1 [说明](#)

8.4.1.1 [defaultProps 的确切类型](#)

默认类型是从 defaultProps 属性的类型推断而来。如果添加了显式的类型注释，比如

static defaultProps: Partial<Props>;，编译器无法识别哪个属性具有默认值（因为 defaultProps 类型包含了 Props 的所有属性）。

使用 static defaultProps: Pick<Props, "name">; 做为显式的类型注释，或者不添加类型注释。

对于函数组件（之前叫做 SFC），使用 ES2015 默认的初始化器：

```

function Greet({ name = "world" }: Props) {
  return <div>Hello {name.toUpperCase()}!</div>;
}

```

8.4.1.2 [@types/React 的改动](#)

仍需要在 @types/React 里 JSX 命名空间上添加 LibraryManagedAttributes 定义。

8.5 [/// <reference lib="..." />指令](#)

TypeScript 增加了一个新的三斜线指令（`/// <reference lib="name" />`），允许一个文件显式地包含一个已知的内置 lib 文件。

内置的 lib 文件的引用和 tsconfig.json 里的编译器选项 lib 相同（例如，使用

lib="es2015" 而不是 lib="lib.es2015.d.ts" 等）。

当你写的声明文件依赖于内置类型时，例如 DOM APIs 或内置的 JS 运行时构造函数如 Symbol 或 Iterable，推荐使用三斜线引用指令。之前，这个 .d.ts 文件不得不添加重覆的类型声明。

[例子](#)

在某个文件里使用 `/// <reference lib="es2017.string" />` 等同于指定 `--lib es2017.string` 编译选项。
`/// <reference lib="es2017.string" />`

```
"foo".padStart(4);
```

```
ts_upgrade_from_2.2_to_5.3
    return { ...t, ...u }; // T & U
}

declare let x: { a: string, b: number };
declare let y: { b: string, c: boolean };

let s1 = { ...x, ...y }; // { a: string, b: string, c: boolean }
let s2 = spread(x, y);    // { a: string, b: number } & { b: string, c:
boolean }
let b1 = s1.b; // string
let b2 = s2.b; // number & string
```

10.3 泛型对象剩余变量和参数

TypeScript 3.2 开始允许从泛型变量中解构剩余绑定。它是通过使用 lib.d.ts 里预定义的 Pick 和 Exclude 助手类型，并结合使用泛型类型和解构式里的其它绑定名实现的。

```
function excludeTag<T extends { tag: string }>(obj: T) {
    let { tag, ...rest } = obj;
    return rest; // Pick<T, Exclude<keyof T, "tag">>
}

const taggedPoint = { x: 10, y: 20, tag: "point" };
const point = excludeTag(taggedPoint); // { x: number, y: number }

10.4 BigInt
BigInt 里 ECMAScript 的一项提案，它在理论上允许我们建模任意大小的整数。
TypeScript 3.2 可以为 BigInt 进行类型检查，并支持在目标为 esnext 时输出 BigInt
字面量。
为支持 BigInt，TypeScript 引入了一个新的原始类型 bigint（全小写）。可以通过调用
BigInt() 函数或书写 BigInt 字面量（在整型数字字面量末尾添加 n）来获取 bigint。
let foo: bigint = BigInt(100); // the BigInt function
let bar: bigint = 100n;        // a BigInt literal

// *Slaps roof of fibonacci function*
// This bad boy returns ints that can get *so* big!
function fibonacci(n: bigint) {
    let result = 1n;
    for (let last = 0n, i = 0n; i < n; i++) {
        const current = result;
        result += last;
        last = current;
    }
    return result;
}
```

```
fibonacci(100000n)
尽管你可能会认为 number 和 bigint 能互换使用，但它们是不同的东西。
declare let foo: number;
declare let bar: bigint;

foo = bar; // error: Type 'bigint' is not assignable to type 'number'.
```

```
ts_upgrade_from_2.2_to_5.3
在 TypeScript 3.1 里使用 Node 模块解析时，TypeScript 会读取 package.json 文件，
找到它需要读取的文件，它首先会查看名字为 typesVersions 的字段。一个带有
typesVersions 字段的 package.json 文件：
{
    "name": "package-name",
    "version": "1.0",
    "types": "./index.d.ts",
    "typesVersions": {
        ">=3.1": { "*": ["ts3.1/*"] }
    }
}
```

package.json 告诉 TypeScript 去检查当前版本的 TypeScript 是否正在运行。如果是 3.1 或以上的版本，它会找出你导入的包的路径，然后读取这个包里面的 ts3.1 文件夹里的内容。这就是 { "*": ["ts3.1/*"] } 的意义 - 如果你对路径映射熟悉，它们的工作方式类似。

因此在上例中，如果我们正在从 "package-name" 中导入，并且正在运行的 TypeScript 版本为 3.1，我们会尝试从 [...] /node_modules/package-name/ts3.1/index.d.ts 开始解析。如果是从 package-name/foo 导入，由会查找 [...] /node_modules/package-name/ts3.1/foo.d.ts 和 [...] /node_modules/package-name/ts3.1/foo/index.d.ts。

那如果当前运行的 TypeScript 版本不是 3.1 呢？如果 typesVersions 里没有能匹配上的版本，TypeScript 将回退到查看 types 字段，因此 TypeScript 3.0 及之前的版本会重定向到 [...] /node_modules/package-name/index.d.ts。

9.3.1 匹配行为

TypeScript 使用 Node 的 [semver ranges](#) 去决定编译器和语言版本。

9.3.2 多个字段

typesVersions 支持多个字段，每个字段都指定了一个匹配范围。

```
{
    "name": "package-name",
    "version": "1.0",
    "types": "./index.d.ts",
    "typesVersions": {
        ">=3.2": { "*": ["ts3.2/*"] },
        ">=3.1": { "*": ["ts3.1/*"] }
    }
}

因为范围可能会重叠，因此指定的顺序是有意义的。在上例中，尽管 >=3.2 和 >=3.1 都匹配
TypeScript 3.2 及以上版本，反转它们的顺序将会有不同的结果，因此上例与下面的代码并
不等同。
{
    "name": "package-name",
    "version": "1.0",
    "types": "./index.d.ts",
    "typesVersions": {
        // 注意，这样写不生效
        ">=3.1": { "*": ["ts3.1/*"] },
        ">=3.2": { "*": ["ts3.2/*"] }
    }
}
```


10 v3.2

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.2.html>

10.1 strictBindCallApply

TypeScript 3.2 引入了一个新的`--strictBindCallApply` 编译选项（是`--strict` 选项家族之一）。在使用了此选项后，函数对象上的`bind`，`call` 和`apply` 方法将应用强类型并进行严格的类型检查。

```
function foo(a: number, b: string): string {
    return a + b;
}
```

```
let a = foo.apply(undefined, [10]);           // error: too few
argumnts
let b = foo.apply(undefined, [10, 20]);        // error: 2nd argument
is a number
let c = foo.apply(undefined, [10, "hello", 30]); // error: too many
arguments
let d = foo.apply(undefined, [10, "hello"]);    // okay! returns a
string
```

它的实现是通过引入了两种新类型来完成的，即`lib.d.ts`里的`CallableFunction`和`NewableFunction`。这些类型包含了针对常规函数和构造函数上`bind`、`call`和`apply`的泛型方法声明。这些声明使用了泛型剩余参数来捕获和反射参数列表，使之具有强类型。在`--strictBindCallApply`模式下，这些声明作用在`Function`类型声明出现的位置。

警告

由于更严格的检查可能暴露之前没发现的错误，因此这是`--strict`模式下的一个破坏性改动。此外，这个新功能还有另一个警告。由于有这些限制，`bind`、`call`和`apply`无法为重载的泛型函数或重载的函数进行完整地建模。当在泛型函数上使用这些方法时，类型参数会被替换为空对象类型（`{}`），并且若在有重载的函数上使用这些方法时，只有最后一个重载会被建模。

10.2 对象字面量的泛型展开表达式

TypeScript 3.2 开始，对象字面量允许泛型展开表达式，它产生交叉类型，和`Object.assign`函数或JSX字面量类似。例如：

```
function taggedObject<T, U extends string>(obj: T, tag: U) {
    return { ...obj, tag }; // T & { tag: U }
}
```

```
let x = taggedObject({ x: 10, y: 20 }, "point"); // { x: number, y:
number } & { tag: "point" }
```

属性赋值和非泛型展开表达式会最大程度地合并到泛型展开表达式的一侧。例如：

```
function foo1<T>(t: T, obj1: { a: string }, obj2: { b: string }) {
    return { ...obj1, x: 1, ...t, ...obj2, y: 2 }; // { a: string, x:
number } & T & { b: string, y: number }
}
```

非泛型展开表达式与之前的行为相同：函数调用签名和构造签名被移除，仅有非方法的属性被保留，针对同名属性则只有出现在最右侧的会被使用。它与交叉类型不同，交叉类型会连接调用签名和构造签名，保留所有的属性，合并同名属性的类型。因此，当展开使用泛型初始化的相同类型时可能会产生不同的结果：

```
function spread<T, U>(t: T, u: U) {
```


- 联合类型中最多有一个类型具有多个重载，
- 联合类型中最多有一个类型有泛型签名。

这意味着，像 `map` 这种操作 `number[] | string[]` 的方法，还是不能调用，因为 `map` 是泛型函数。

另一方面，像 `forEach` 就可以调用，因为它不是泛型函数，但在 `noImplicitAny` 模式可能有些问题。

```
interface Dog {
  kind: "dog";
  dogProp: any;
}
interface Cat {
  kind: "cat";
  catProp: any;
}
```

```
const catOrDogArray: Dog[] | Cat[] = [];
```

```
catOrDogArray.forEach(
  animal => {
    // ~~~~~~ error!
    // Parameter 'animal' implicitly has an 'any' type.
  });
```

添加显式的类型信息可以解决。

```
interface Dog {
  kind: "dog";
  dogProp: any;
}
interface Cat {
  kind: "cat";
  catProp: any;
}
```

```
const catOrDogArray: Dog[] | Cat[] = [];
catOrDogArray.forEach(
  (animal: Dog | Cat) => {
    if (animal.kind === "dog") {
      animal.dogProp;
      // ...
    } else if (animal.kind === "cat") {
      animal.catProp;
      // ...
    }
  });
```

11.2 在合复合工程中增量地检测文件的变化 --build --watch

TypeScript 3.0 引入了一个新特性来按结构进行构建，称做“复合工程”。目的是让用户能够把大型工程拆分成小的部分从而快速构建并保留项目结构。正是因为支持了复合工程，TypeScript 可以使用 `--build` 模式仅重新编译部分工程和依赖。可以把它当做工作内部构建的一种优化。

`bar = foo; // error: Type 'number' is not assignable to type 'bigint'.` ECMAScript 里规定，在算术运算符里混合使用 `number` 和 `bigint` 是一个错误。应该显式地将值转换为 `BigInt`。

```
console.log(3.141592 * 10000n); // error
console.log(3145 * 10n); // error
console.log(BigInt(3145) * 10n); // okay!
```

还有一点要注意的是，对 `bigint` 使用 `typeof` 操作符返回一个新的字符串：`"bigint"`。因此，TypeScript 能够正确地使用 `typeof` 细化类型。

```
function whatKindOfNumberIsIt(x: number | bigint) {
  if (typeof x === "bigint") {
    console.log("'x' is a bigint!");
  }
  else {
    console.log("'x' is a floating-point number");
  }
}
```

感谢 [Caleb Sander](#) 为实现此功能的付出。

警告

`BigInt` 仅在目标为 `esnext` 时才支持。可能不是很明显的一点是，因为 `BigInts` 针对算术运算符 `+`，`-`，`*` 等具有不同的行为，为老旧版（如 `es2017` 及以下）提供此功能时意味着重写出现它们的每一个操作。TypeScript 需根据类型和涉及到的每一处加法，字符串拼接，乘法等产生正确的行为。

因为这个原因，我们不会立即提供向下的支持。好的一面是，Node 11 和较新版本的 Chrome 已经支持了这个特性，因此你可以在目标为 `esnext` 时，使用 `BigInt`。

一些目标可能包含 `polyfill` 或类似 `BigInt` 的运行时对象。基于这些考虑，你可能会想要添加 `esnext.bigint` 到 `lib` 编译选项里。

10.5 Non-unit types as union discriminants

TypeScript 3.2 放宽了作为判别式属性的限制，来让类型细化变得容易。如果联合类型的共同属性包含了某些单体类型（如，字面符字面量，`null` 或 `undefined`）且不包含泛型，那么它就可以做为判别式。

因此，TypeScript 3.2 认为下例中的 `error` 属性可以做为判别式。这在之前是不可以的，因为 `Error` 并非是一个单体类型。那么，`unwrap` 函数体里的类型细化就可以正确地工作了。

```
type Result<T> =
  | { error: Error; data: null }
  | { error: null; data: T };
```

```
function unwrap<T>(result: Result<T>) {
  if (result.error) {
    // Here 'error' is non-null
    throw result.error;
  }
```

```
  // Now 'data' is non-null
  return result.data;
}
```

10.6 tsconfig.json 可以通过 Node.js 包来继承

TypeScript 3.2 现在可以从 `node_modules` 里解析 `tsconfig.json`。如果 `tsconfig.json` 文件里的 `"extends"` 设置为空，那么 TypeScript 会检测 `node_modules`

ts_upgrade_from_2.2_to_5.3

包。When using a bare path for the "extends" field in tsconfig.json, TypeScript will dive into node_modules packages for us.

```
{
  "extends": "@my-team/tsconfig-base",
  "include": ["./**/*"]
  "compilerOptions": {
    // Override certain options on a project-by-project basis.
    "strictBindCallApply": false,
  }
}
```

这里，TypeScript 会去 node_modules 目录里查找@my-team/tsconfig-base 包。针对每一个包，TypeScript 检查 package.json 里是否包含"tsconfig"字段，如果是，TypeScript 会尝试从那里加载配置文件。如果两者都不存在，TypeScript 尝试从根目录读取 tsconfig.json。这与 Nodejs 查找.js 文件或 TypeScript 查找.d.ts 文件的已有过程类似。

这个特性对于大型组织或具有很多分布的依赖的工程特别有帮助。

10.7 [The new --showConfig flag](#)

tsc, TypeScript 编译器，支持一个新的标记--showConfig。运行 tsc --showConfig 时，TypeScript 计算生效的 tsconfig.json 并打印（继承的配置也会计算在内）。这对于调试诊断配置问题很有帮助。

10.8 [JavaScript 的 Object.defineProperty 声明](#)

在编写 JavaScript 文件时（使用 allowJs），TypeScript 能识别出使用 Object.defineProperty 声明。也就是说会有更好的代码补全功能，和强类型检查，这需要在 JavaScript 文件里启用类型检查功能（打开 checkJs 选项或在文件顶端添加// @ts-check 注释）。

```
// @ts-check

let obj = {};
Object.defineProperty(obj, "x", { value: "hello", writable: false });

obj.x.toLowerCase();
// ~~~~~
// error:
//   Property 'toLowerCase' does not exist on type 'string'.
//   Did you mean 'toLowerCase'?

obj.x = "world";
// ~
// error:
//   Cannot assign to 'x' because it is a read-only property.
```

11 v3.3

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.3.html>

11.1 [改进调用联合类型时的行为](#)

在 TypeScript 之前的版本中，将可调类型联合后仅在它们具有相同的参数列表时才能被调用。

```
type Fruit = "apple" | "orange";
type Color = "red" | "orange";
```

```
type FruitEater = (fruit: Fruit) => number; // eats and ranks the fruit
```

```
type ColorConsumer = (color: Color) => string; // consumes and describes the colors
```

```
declare let f: FruitEater | ColorConsumer;
```

```
// Cannot invoke an expression whose type lacks a call signature.
//   Type 'FruitEater | ColorConsumer' has no compatible call signatures.ts(2349)
f("orange");
```

然而，上例中，FruitEater 和 ColorConsumer 应该都可以使用"orange"，并返回 number 或 string。

在 TypeScript 3.3 里，这个错误不存在了。

```
type Fruit = "apple" | "orange";
type Color = "red" | "orange";
```

```
type FruitEater = (fruit: Fruit) => number; // eats and ranks the fruit
```

```
type ColorConsumer = (color: Color) => string; // consumes and describes the colors
```

```
declare let f: FruitEater | ColorConsumer;
```

```
f("orange"); // It works! Returns a 'number | string'.
```

```
f("apple"); // error - Argument of type '"apple"' is not assignable to parameter of type '"orange"'.
```

```
f("red"); // error - Argument of type '"red"' is not assignable to parameter of type '"orange"'.
```

TypeScript 3.3，这些签名的参数被连结在一起构成了一个新的签名。

在上例中，fruit 和 color 连结在一起形成新的参数类型 Fruit & Color。Fruit & Color 和("apple" | "orange") & ("red" | "orange")是一样的，都相当于("apple" & "red") | ("apple" & "orange") | ("orange" & "red") | ("orange" & "orange")。那些不可能交叉的会规约成 never 类型，只剩下"orange" & "orange"，就是"orange"。

[警告](#)

这个新行为仅在满足如下情形时生效：

v3.3 => 改进调用联合类型时的行为

```
makeBoxedArray("hello!").value[0].toUpperCase();
// ~~~~~
// 错误: 类型 '{}' 没有 'toUpperCase' 属性
在旧版本中, 当从其他类型变量 (如 T 和 U) 推断时, TypeScript 会推断出空对象类型
({})。
在 TypeScript 3.4 中的类型参数推断时, 对于返回函数的泛型函数的调用,
TypeScript 将 (视情况而定) 把类型参数从泛型函数参数传递到生成的函数类型中。
换句话说, 而不是生成类型
(arg: {}) => Box<{}[]>
TypeScript 3.4 生成的类型
<T>(arg: T) => Box<T[]>
注意, T 已从 makeArray 传递到结果类型的类型参数列表中。 这意味着来自 compose 参
数的泛型已被保留, 我们的 makeBoxedArray 示例将正常运行!
```

```
interface Box<T> {
  value: T;
}

function makeArray<T>(x: T): T[] {
  return [x];
}

function makeBox<U>(value: U): Box<U> {
  return { value };
}

// 类型为 '<T>(arg: T) => Box<T[]>'
const makeBoxedArray = compose(
  makeArray,
  makeBox,
)

// 正常运行!
makeBoxedArray("hello!").value[0].toUpperCase();
更多细节, 你可以读到更多从这些原始的变动。
```

12.3 改进 [ReadonlyArray](#) 和 [readonly](#) 元祖

TypeScript 3.4 让使用只读的类似数组的类型更简单了。

12.3.1 一个与 [ReadonlyArray](#) 相关的新语法

[ReadonlyArray](#) 类型描述 [Array](#) 是只读的。任何带有 [ReadonlyArray](#) 引用的变量不能被添加、移除或者替换数组中的任何元素。

```
function foo(arr: ReadonlyArray<string>) {
  arr.slice(); // okay
  arr.push("hello!"); // error!
}
```

当期待数组不可变时使用 [ReadonlyArray](#) 替代 [Array](#) 是好实践, 考虑到数组有一个更棒的语法的情况下这通常有一点痛苦。尤其是, `number[]` 是一个省略版的 `Array<number>`, 就像 `Date[]` 是省略版的 `Array<Date>`。

ts_upgrade_from_2.2_to_5.3

TypeScript 2.7 还引入了 `--watch` 构建模式, 它使用了新的增量 `"builder"` API。背后的想法都是仅重新检查和生成改动过的文件或者是依赖项可能影响类型检查的文件。 可以把它们当成工程内部构建的优化。

在 3.3 之前, 使用 `--build --watch` 构建复合工程不会真正地使用增量文件检测机制。 在 `--build --watch` 模式下, 一个工程里的一处改动会导致整个工程重新构建, 而非仅检查那些真正受到影响的文件。

在 TypeScript 3.3 里, `--build` 模式的 `--watch` 标记也会使用增量文件检测。 因此 `--build --watch` 模式下构建非常快。 我们的测试结果显示, 这个功能会减少 50%到 75%的构建时间, 相比于原先的 `--build --watch`。 具体数字在这这个 [pull request](#) 里, 我们相信大多数复合工程用户会看到明显效果。

12 v3.4

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.4.html>

12.1 使用 `--incremental` 标志加快后续构建

TypeScript 3.4 引入了一个名为 `--incremental` 的新标志，它告诉 TypeScript 从上一次编译中保存有关项目图的信息。

下次使用 `--incremental` 调用 TypeScript 时，它将使用该信息来检测类型检查和生成对项目更改成本最低的方法。

// tsconfig.json

```
{
  "compilerOptions": {
    "incremental": true,
    "outDir": "./lib"
  },
  "include": ["./src"]
}
```

默认使用这些设置，当我们运行 `tsc` 时，TypeScript 将在输出目录 (`./lib`) 中查找名为 `.tsbuildinfo` 的文件。如果 `./lib/.tsbuildinfo` 不存在，它将被生成。但如果存在，`tsc` 将尝试使用该文件逐步进行类型检查并更新输出文件。

这些 `.tsbuildinfo` 文件可以安全地删除，并且在运行时对我们的代码没有任何影响——它们纯粹用于更快地编译。我们也可以将它们命名为我们想要的任何名字，并使用 `--`

`tsBuildInfoFile` 标志将它们放在我们想要的任何位置。

// front-end.tsconfig.json

```
{
  "compilerOptions": {
    "incremental": true,
    "tsBuildInfoFile": "./buildcache/front-end",
    "outDir": "./lib"
  },
  "include": ["./src"]
}
```

12.1.1 复合项目

复合项目的意图的一部分 (`tsconfig.json`s, `composite` 设置为 `true`) 是不同项目之间的引用可以增量构建。因此，复合项目将始终生成 `.tsbuildinfo` 文件。

12.1.2 outFile

当使用 `outFile` 时，构建信息文件的名称将基于输出文件的名称。例如，如果我们的输出 JavaScript 文件是 `./output / foo.js`，那么在 `--incremental` 标志下，TypeScript 将生成文件 `./output/foo.tsbuildinfo`。如上所述，这可以通过 `--tsBuildInfoFile` 标志来控制。

12.2 泛型函数的高阶类型推断

当来自其它泛型函数的推断产生用于推断的自由类型变量时，TypeScript 3.4 现在可以生成泛型函数类型。

这意味着在 3.4 中许多函数组合模式现在运行的更好了。

为了更具体，让我们建立一些动机并考虑以下 `compose` 函数：

```
function compose<A, B, C>(f: (arg: A) => B, g: (arg: B) => C): (arg: A) => C {
  return x => g(f(x));
}
```

```
}
```

`compose` 还有两个其他函数：

- `f` 它接受一些参数（类型为 `A`）并返回类型为 `B` 的值
- `g` 采用类型为 `B` 的参数（类型为 `f` 返回），并返回类型为 `C` 的值

`compose` 然后返回一个函数，它通过 `f` 然后 `g` 来提供它的参数。

调用此函数时，TypeScript 将尝试通过一个名为 `type argument inference` 的进程来计算 `A`, `B` 和 `C` 的类型。这个推断过程通常很有效：

```
interface Person {
  name: string;
  age: number;
}
```

```
function getDisplayName(p: Person) {
  return p.name.toLowerCase();
}
```

```
function getLength(s: string) {
  return s.length;
}
```

// 拥有类型 '(p: Person) => number'

```
const getDisplayNameLength = compose(
  getDisplayName,
  getLength,
);
```

// 有效并返回 `number` 类型

```
getDisplayNameLength({ name: "Person McPersonface", age: 42 });
```

推断过程在这里相当简单，因为 `getDisplayName` 和 `getLength` 使用的是可以轻松引用的类型。但是，在 TypeScript 3.3 及更早版本中，泛型函数如 `compose` 在传递其他泛型函数时效果不佳。

```
interface Box<T> {
  value: T;
}
```

```
function makeArray<T>(x: T): T[] {
  return [x];
}
```

```
function makeBox<U>(value: U): Box<U> {
  return { value };
}
```

// 类型为 '(arg: {}) => Box<{}[]>'

```
const makeBoxedArray = compose(
  makeArray,
  makeBox,
)
```

```
export default {
  red: "RED",
  blue: "BLUE",
  green: "GREEN",
} as const;
```

注意事项

需要注意的是，const 断言只能直接应用于简单的字面量表达式上。

// 错误! 'const' 断言只能用在 string, number, boolean, array, object literal。

```
let a = (Math.random() < 0.5 ? 0 : 1) as const;
```

// 有效!

```
let b = Math.random() < 0.5 ?
  0 as const :
  1 as const;
```

另一件得记住的事是 const 上下文不会直接将表达式转换为完全不可变的。

```
let arr = [1, 2, 3, 4];
```

```
let foo = {
  name: "foo",
  contents: arr,
} as const;
```

```
foo.name = "bar"; // 错误!
foo.contents = [];
```

```
foo.contents.push(5); // ...有效!
更多详情，你可以查看相应的 pull request。
```

12.5 [对 globalThis 的类型检查](#)

TypeScript 3.4 引入了对 ECMAScript 新 globalThis 全局变量的类型检查的支持，它指向的是全局作用域。与上述解决方案不同，globalThis 提供了一种访问全局作用域的标准方法，可以在不同环境中使用。

// 在一个全局文件里：

```
var abc = 100;
```

// 指向上面的 `abc`

```
globalThis.abc = 200;
```

注意，使用 let 和 const 声明的全局变量不会显示在 globalThis 上。

```
let answer = 42;
```

// 错误! 'typeof globalThis' 没有 'answer' 属性。

```
globalThis.answer = 333333;
```

同样重要的是要注意，在编译为老版本的 ECMAScript 时，TypeScript 不会转换引用到 globalThis 上。因此，除非您的目标是常青浏览器（已经支持 globalThis），否则您可能需要[使用 polyfill](#)。

更多详细信息，请参阅[该功能的 pull request](#)。

[参考：原文](#)

TypeScript 3.4 为 ReadonlyArray 引入了一个新的语法，就是在数组类型上使用了新的 readonly 修饰语。

```
function foo(arr: readonly string[]) {
  arr.slice(); // okay
  arr.push("hello!"); // 错误!
}
```

12.3.2 [readonly 元祖](#)

TypeScript 3.4 同样引入了对 readonly 元祖的支持。我们可以在任何元祖类型上加上前置 readonly 关键字用来表示它是 readonly 元祖，非常像我们现在可以对数组使用的省略版语法。就像你可能期待的，不像插槽可写的普通元祖，readonly 元祖只允许从那些位置读。

```
function foo(pair: readonly [string, string]) {
  console.log(pair[0]); // okay
  pair[1] = "hello!"; // 错误
}
```

普通的元祖是用相同的方式从 Array 继承的——一个元祖 T1, T2, ... Tn 继承自 Array< T1 | T2 | ... Tn > - readonly 元祖是继承自类型 ReadonlyArray。所以，一个 readonly 元祖 T1, T2, ... Tn 继承自 ReadonlyArray< T1 | T2 | ... Tn >。

12.3.3 [映射类型修饰语 readonly 和 readonly 数组](#)

在之前的 TypeScript 版本中，我们一般使用映射类型操作不同的类似数组的结构。

这意味着，一个映射类型像 Boxify 可以在数组上生效，元祖也是。

```
interface Box<T> { value: T }
```

```
type Boxify<T> = {
}
```

```
// { a: Box<string>, b: Box<number> }
type A = Boxify<{ a: string, b: number }>;
```

```
// Array<Box<number>>
type B = Boxify<number[]>;
```

```
// [Box<string>, Box<number>]
type C = Boxify<[string, boolean]>;
```

不幸的是，映射类型像 Readonly 实用类型在数组和元祖类型上实际上是无用的。

// lib.d.ts

```
type Readonly<T> = {
  readonly [K in keyof T]: T[K]
}
```

// 在 TypeScript 3.4 之前代码会如何执行

```
// { readonly a: string, readonly b: number }
type A = Readonly<{ a: string, b: number }>;
```

```
// number[]
type B = Readonly<number[]>;
```



```
// [string, boolean]
type C = Readonly<[string, boolean]>;
在 TypeScript 3.4, 在映射类型中的 readonly 修饰符将自动的转换类似数组结构到他们
相符合的 readonly 副本。
// 在 TypeScript 3.4 中代码会如何运行
```

```
// { readonly a: string, readonly b: number }
type A = Readonly<{ a: string, b: number }>;
```

```
// readonly number[]
type B = Readonly<number[]>;
```

```
// readonly [string, boolean]
type C = Readonly<[string, boolean]>;
类似地, 你可以编写一个类似 Writable 映射类型的实用程序类型来移除 readonly-
ness, 并将 readonly 数组容器转换回它们的可变等价物。
```

```
type Writable<T> = {
  -readonly [K in keyof T]: T[K]
}
```

```
// { a: string, b: number }
type A = Writable<{
  readonly a: string;
  readonly b: number
}>;
```

```
// number[]
type B = Writable<readonly number[]>;
```

```
// [string, boolean]
type C = Writable<readonly [string, boolean]>;
```

12.3.4 注意事项

它不是一个通用型操作, 尽管它看起来像。 readonly 类型修饰符只能用于数组类型和元组类型的语法。

```
let err1: readonly Set<number>; // 错误!
let err2: readonly Array<boolean>; // 错误!
```

```
let okay: readonly boolean[]; // 有效
你可以查看 pull request 了解更多详情。
```

12.4 const 断言

TypeScript 3.4 引入了一个叫 const 断言的字面量值的新构造。它的语法是用 const 代替类型名称的类型断言 (例如 123 as const)。当我们用 const 断言构造新的字面量表达式时, 我们可以用来表示:

- 该表达式中的字面量类型不应粗化 (例如, 不要从 'hello' 到 string)
- 对象字面量获得 readonly 属性
- 数组字面量成为 readonly 元组

```
// Type '"hello"'
```

```
let x = "hello" as const;
```

```
// Type 'readonly [10, 20]'
```

```
let y = [10, 20] as const;
```

```
// Type '{ readonly text: "hello" }'
```

```
let z = { text: "hello" } as const;
```

也可以使用尖括号断言语法, 除了 .tsx 文件之外。

```
// Type '"hello"'
let x = <const>"hello";
```

```
// Type 'readonly [10, 20]'
```

```
let y = <const>[10, 20];
```

```
// Type '{ readonly text: "hello" }'
```

```
let z = <const>{ text: "hello" };
```

此功能意味着通常可以省略掉仅用于将不可变性示意给编译器的类型。

// 不使用引用或声明的类型。

// 我们只需要一个 const 断言。

```
function getShapes() {
  let result = [
    { kind: "circle", radius: 100, },
    { kind: "square", sideLength: 50, },
  ] as const;

  return result;
}
```

```
for (const shape of getShapes()) {
  // 完美细化
  if (shape.kind === "circle") {
    console.log("Circle radius", shape.radius);
  }
  else {
    console.log("Square side length", shape.sideLength);
  }
}
```

请注意, 上面的例子不需要类型注释。const 断言允许 TypeScript 采用最具体的类型表达式。

如果你选择不使用 TypeScript 的 enum 结构, 这甚至可以用于在纯 JavaScript 代码中使用类似 enum 的模式。

```
export const Colors = {
  red: "RED",
  blue: "BLUE",
  green: "GREEN",
} as const;
```

```
// 或者使用 'export default'
```



```
ts_upgrade_from_2.2_to_5.3
kind: "bag";
value: U;
constructor(value: U) {
  this.value = value;
}
}
```

```
function composeCtor<T, U, V>(F: new (x: T) => U, G: new (y: U) => V):
(x: T) => V {
  return x => new G(new F(x))
}
```

```
let f = composeCtor(Box, Bag); // 拥有类型 '<T>(x: T) => Bag<Box<T>>''
let a = f(1024); // 拥有类型 'Bag<Box<number>>''
```

除了上面的组合模式之外，这种对泛型构造函数的新推断意味着在某些 UI 库（如 React）中对类组件进行操作的函数可以更正确地对泛型类组件进行操作。

```
type ComponentClass<P> = new (props: P) => Component<P>;
declare class Component<P> {
  props: P;
  constructor(props: P);
}
```

```
declare function myHoc<P>(C: ComponentClass<P>): ComponentClass<P>;
```

```
type NestedProps<T> = { foo: number, stuff: T };
```

```
declare class GenericComponent<T> extends Component<NestedProps<T>>
{ }
```

```
// 类型为 'new <T>(props: NestedProps<T>) => Component<NestedProps<T>>''
```

```
const GenericComponent2 = myHoc(GenericComponent);
```

想学习更多，在 [GitHub](#) 上查看原始的 [pull request](#)。

参考：原文

13 v3.5

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.5.html>

13.1 改进速度

TypeScript 3.5 为类型检查和增量构建采用了几个优化。

13.1.1 类型检查速度提升

TypeScript 3.5 包含对 TypeScript 3.4 的某些优化，可以更高效地进行类型检查。在代码补全列表等类型检查驱动的操作上，这些改进效果显著。

13.1.2 改进 --incremental

TypeScript 3.5 通过缓存计算状态的信息（编译器设置、寻找文件的原因、文件在哪里被找到等等），改进了在 3.4 中的 --incremental 构建模式。[我们发现重新构建花费的时间比 TypeScript 3.4 减少了 68%!](#)

有关更多信息，你可以查看这些 [pull requests](#)

- [缓存模块解析](#)
- [缓存 tsconfig.json 计算](#)

13.2 Omit 辅助类型

TypeScript 3.5 添加了新的 Omit 辅助类型，这个类型用来创建从原始类型中移除了某些属性的新类型。

```
type Person = {
  name: string;
  age: number;
  location: string;
};
```

```
type QuantumPerson = Omit<Person, "location">;
```

// 相当于

```
type QuantumPerson = {
  name: string;
  age: number;
};
```

使用 Omit 辅助，我们有能力复制 Person 中除了 location 之外的所有属性。

有关更多细节，在 [GitHub](#) 查看添加 Omit 的 [pull request](#)，以及[有关剩余对象使用 Omit 的更改](#)。

13.2.1 改进了联合类型中多余属性的检查

在 TypeScript 3.4 及之前的版本中，会出现确实不应该存在的多余属性却被允许存在的情况。例如，TypeScript 3.4 在对象字面量上允许不正确的 name 属性，甚至它的类型在 Point 和 Label 之中都不匹配。

```
type Point = {
  x: number;
  y: number;
};
```

```
type Label = {
  name: string;
};
```

```
const thing: Point | Label = {
  x: 0,
  y: 0,
  name: true // uh-oh!
};
```

以前，一个无区分的联合在它的成员上不会进行任何多余属性的检查，结果，类型错误的 name 属性溜了进来。

在 TypeScript 3.5 中，类型检查器至少会验证所有提供的属性属于某个联合类型的成员，且类型恰当，这意味着，上面的例子会正确的进行错误提示。

注意，只要属性类型有效，仍允许部分重叠。

```
const pl: Point | Label = {
  x: 0,
  y: 0,
  name: "origin" // okay
};
```

13.3 [--allowUmdGlobalAccess 标志](#)

在 TypeScript 3.5 中，使用新的 --allowUmdGlobalAccess 标志，你现在可以从任何位置引用全局的 UMD 申明——甚至模块。

```
export as namespace foo;
```

此模式增加了混合和匹配第三方库的灵活性，其中库声明的全局变量总是可以被使用，甚至可以从模块内部使用。

有关更多细节，[查看 GitHub 上的 pull request](#)。

13.4 [更智能的联合类型检查](#)

在 TypeScript 3.4 以及之前的版本中，下面的例子会无效：

```
type S = { done: boolean, value: number }
type T =
  | { done: false, value: number }
  | { done: true, value: number };
```

```
declare let source: S;
declare let target: T;
```

```
target = source;
```

这是因为 S 无法被分配给 { done: false, value: number } 或者 { done: true, value: number }。为啥？因为属性 done 在 S 不够具体——他是 boolean。而 T 的每个成员有一个明确的为 true 或者 false 属性 done。

这就是我们单独检查每个成员的意义：TypeScript 不只是将每个属性合并在一起，看看是否可以赋予 S。

如果这样做，一些糟糕的代码可能会像下面这样：

```
interface Foo {
  kind: "foo";
  value: string;
}
```

```
interface Bar {
  kind: "bar";
  value: number;
}
```

```
function doSomething(x: Foo | Bar) {
  if (x.kind === "foo") {
    x.value.toLowerCase();
  }
}
```

// uh-oh - 幸运的是，TypeScript 在这里会提示错误！

```
doSomething({
  kind: "foo",
  value: 123,
});
```

然而，对于原始的例子，这有点过于严格。如果你弄清除 S 的任何可能值的精确类型，你实际上可以看到它与 T 中的类型完全匹配。

在 TypeScript 3.5 中，当分配具有辨别属性的类型时，如 T，实际上将进一步将类似 S 的类型分解为每个可能的成员类型的并集。在这种情况下，由于 boolean 是 true 和 false 的联合，S 将被视为 {done: false, value: number} 和 {done: true, value: number}。

有关更多细节，你可以[在 GitHub 上查看原始的 pull request](#)。

13.5 [泛型构造函数的高阶类型推断](#)

在 TypeScript 3.4 中，我们改进了对返回函数的泛型函数的推断：

```
function compose<T, U, V>(f: (x: T) => U, g: (y: U) => V): (x: T) => V {
  return x => g(f(x))
}
```

将其他泛型函数作为参数，如下所示：

```
function arrayify<T>(x: T): T[] {
  return [x];
}
```

```
type Box<U> = { value: U }
function boxify<U>(y: U): Box<U> {
  return { value: y };
}
```

```
let newFn = compose(arrayify, boxify);
```

TypeScript 3.4 的推断允许 newFn 是泛型的。它的新类型是 <T> (x: T) => Box <T []>。而不是旧版本推断的，相对无用的类型，如 (x: {}) => Box <{} []>。

TypeScript 3.5 在处理构造函数的时候推广了这种行为。

```
class Box<T> {
  kind: "box";
  value: T;
  constructor(value: T) {
    this.value = value;
  }
}
```

```
class Bag<U> {
```

ts_upgrade_from_2.2_to_5.3

```
declare function displayUser(user: User): void;
```

```
async function f() {
    displayUser(userData());
    // ~~~~~
    // 'Promise <User>' 类型的参数不能分配给 'User' 类型的参数。
    // ...
    // 你忘记使用 'await' 吗?
}
```

在等待或 `.then()` - Promise 之前尝试访问方法也很常见。这是另一个例子，在许多其他方面，我们能够做得更好。

```
async function getCuteAnimals() {
    fetch("https://reddit.com/r/aww.json")
        .json()
    // ~~~~
    // 'Promise <Response>' 类型中不存在属性 'json'。
    // 你忘记使用 'await' 吗?
}
```

目的是即使用户不知道需要等待，至少，这些消息提供了更多关于从何处开始的上下文。

与可发现性相同，让您的生活更轻松 - 除了 Promises 上更好的错误消息之外，我们现在还在某些情况下提供快速修复。

```
async function f() {
    const response1: Promise<Response> = fetch(options);
    const response2 = fetch(options);
    processResponse(fetch(options));
    processResponse(response1);
    processResponse(response2);
}
```

有关更多详细信息，请参阅[原始问题](#)以及[链接回来的 pull request](#)。

14.4 标识符更好的支持 Unicode

当发射到 ES2015 及更高版本的目标时，TypeScript 3.6 在标识符中包含对 Unicode 字符的更好支持。

```
const hello = "world"; // previously disallowed, now allowed in '--target es2015'
// 以前不允许，现在在 '--target es2015' 中允许
```

14.5 支持在 SystemJS 中使用 import.meta

当模块目标设置为 system 时，TypeScript 3.6 支持将 `import.meta` 转换为 `context.meta`。

14 v3.6

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.6.html>

14.1 更严格的生成器

TypeScript 3.6 对迭代器和生成器函数引入了更严格的检查。在之前的版本中，用户无法区分一个值是生成的还是被返回的。

```
function* foo() {
    if (Math.random() < 0.5) yield 100;
    return "Finished!"
}
```

```
let iter = foo();
let curr = iter.next();
if (curr.done) {
    // TypeScript 3.5 以及之前的版本会认为 `value` 为 'string | number'。
    // 当 `done` 为 `true` 的时候，它应该知道 `value` 为 'string'!
    curr.value
}
```

另外，生成器只假定 `yield` 的类型为 `any`。

```
function* bar() {
    let x: { hello(): void } = yield;
    x.hello();
}
```

```
let iter = bar();
iter.next();
iter.next(123); // 不好! 运行时错误!
```

在 TypeScript 3.6 中，在我们第一个例子中检查器现在知道 `curr.value` 的正确类型应该是 `string`，并且，在最后一个例子中当我们调用 `next()` 时会准确的提示错误。这要感谢在 `Iterator` 和 `IteratorResult` 的类型定义包含了一些新的类型参数，并且一个被叫做 `Generator` 的新类型在 TypeScript 中用来表示生成器。

类型 `Iterator` 现在允许用户明确的定义生成的类型，返回的类型和 `next` 能够接收的类型。

```
interface Iterator<T, TReturn = any, TNext = undefined> {
    // 接受 0 或者 1 个参数 - 不接受 'undefined'
    next(...args: [] | [TNext]): IteratorResult<T, TReturn>;
    return?(value?: TReturn): IteratorResult<T, TReturn>;
    throw?(e?: any): IteratorResult<T, TReturn>;
}
```

以此为基础，新的 `Generator` 类型是一个迭代器，它总是有 `return` 和 `throw` 方法，并且也是可迭代的。

```
interface Generator<T = unknown, TReturn = any, TNext = unknown>
extends Iterator<T, TReturn, TNext> {
    next(...args: [] | [TNext]): IteratorResult<T, TReturn>;
    return(value: TReturn): IteratorResult<T, TReturn>;
    throw(e: any): IteratorResult<T, TReturn>;
    [Symbol.iterator]() : Generator<T, TReturn, TNext>;
}
```

v3.6 => 更严格的生成器

ts_upgrade_from_2.2_to_5.3

为了允许在返回值和生成值之间进行区分，TypeScript 3.6 转变 `IteratorResult` 类型为一个区别对待的联合类型：

```
type IteratorResult<T, TReturn = any> = IteratorYieldResult<T> |
IteratorReturnResult<TReturn>;
```

```
interface IteratorYieldResult<TYield> {
  done?: false;
  value: TYield;
}
```

```
interface IteratorReturnResult<TReturn> {
  done: true;
  value: TReturn;
}
```

简而言之，这意味当直接处理迭代器时，你将有能力细化值的类型。

为了正确的表示在调用生成器的 `next()` 方法的时候能被传入的类型，TypeScript 3.6 还可以在生成器函数内推断出 `yield` 的某些用法。

```
function* foo() {
  let x: string = yield;
  console.log(x.toUpperCase());
}
```

```
let x = foo();
x.next(); // 第一次调用 `next` 总是被忽略
x.next(42); // 错啦! 'number' 和 'string' 不匹配
```

如果你更喜欢显示的，你还可以使用显示的返回类型强制申明从生成表达式返回的、生成的和计算的值的类型。下面，`next()` 只能被 `booleans` 值调用，并且根据 `done` 的值，`value` 可以是 `string` 或者 `number`。

```
/**
 * - yields numbers
 * - returns strings
 * - can be passed in booleans
 */
function* counter(): Generator<number, string, boolean> {
  let i = 0;
  while (true) {
    if (yield i++) {
      break;
    }
  }
  return "done!";
}
```

```
var iter = counter();
var curr = iter.next()
while (!curr.done) {
  console.log(curr.value);
  curr = iter.next(curr.value === 5)
}
```

```
console.log(curr.value.toUpperCase());
```

```
// prints:
//
// 0
// 1
// 2
// 3
// 4
// 5
// DONE!
```

有关更多详细的改变，[查看 pull request](#)。

14.2 更准确的数组展开

在 ES2015 之前的目标中，对于像循环和数组展开之类的结构最忠实的生成可能有点繁重。因此，TypeScript 默认使用更简单的生成，它只支持数组类型，并支持使用 `--downlevelIteration` 标志迭代其它类型。在此标志下，发出的代码更准确，但更大。默认情况下 `--downlevelIteration` 默认关闭效果很好，因为大多数以 ES5 为目标的用户只计划使用带数组的迭代结构。但是，我们支持数组的生成在某些边缘情况下仍然存在一些可观察到的差异。

例如，以下示例：

```
[...Array(5)]
```

相当于以下数组：

```
[undefined, undefined, undefined, undefined, undefined]
```

但是，TypeScript 会将原始代码转换为代码：

```
Array(5).slice();
```

这略有不同。 `Array(5)` 生成一个长度为 5 的数组，但并没有在其中插入任何元素！

```
1 in [undefined, undefined, undefined] // true
```

```
1 in Array(3) // false
```

当 TypeScript 调用 `slice()` 时，它还会创建一个索引尚未设置的数组。

这可能看起来有点深奥，但事实证明许多用户遇到了这种令人不快的行为。TypeScript 3.6 不是使用 `slice()` 和内置函数，而是引入了一个新的 `__spreadArrays` 辅助程序，以准确地模拟 ECMAScript 2015 中在 `--downlevelIteration` 之外的旧目标中发生的事情。 `__spreadArrays` 也可以在 `tslib` 中使用（如果你正在寻找更小的包，那么值得一试）。

有关更多信息，请[参阅相关的 pull request](#)。

14.3 改进了 Promises 的 UX

Promise 是当今使用异步数据的常用方法之一。不幸的是，使用面向 Promise 的 API 通常会让用户感到困惑。TypeScript 3.6 引入了一些改进，以防止错误的处理 Promise。例如，在将它传递给另一个函数之前忘记 `.then()` 或等待 Promise 的完成通常是很常见的。TypeScript 的错误消息现在是专门的，并告知用户他们可能应该考虑使用 `await` 关键字。

```
interface User {
  name: string;
  age: number;
  location: string;
}
```

```
declare function getUserData(): Promise<User>;
```

15 v3.7

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.7.html>

15.1 可选链 (Optional Chaining)

[Playground](#)

在我们的 issue 列表上, 可选链是 [issue #16](#)。感受一下, 从那之后 TypeScript 的 issue 列表中新增了 23,000 条 issues。
可选链的核心是, 在我们编写代码中, 当遇到 null 或 undefined, TypeScript 可以立即停止解析一部分表达式。 可选链的关键点是一个为 *可选属性访问* 提供的新的运算符 `?.`。 比如我们可以这样写代码:

```
let x = foo?.bar.baz();
```

意思是, 当 foo 有定义时, 执行 `foo.bar.baz()` 的计算; 但是当 foo 是 null 或 undefined 时, 停止后续的解析, 直接返回 undefined。
更明确地说, 上面的代码和下面的代码等价。

```
let x = (foo === null || foo === undefined) ?
    undefined :
    foo.bar.baz();
```

注意, 当 bar 是 null 或 undefined, 我们的代码访问 baz 依然会报错。 同理, 当 baz 是 null 或 undefined, 在调用时也会报错。 `?.` 只检查它 左边 的值是不是 null 或 undefined, 不检查后续的属性。
你会发现自已可以使用 `?.` 来替换用了 `&&` 的大量空值检查代码。

```
// 以前
if (foo && foo.bar && foo.bar.baz) {
    // ...
}
```

```
// 以后
if (foo?.bar?.baz) {
    // ...
}
```

注意, `?.` 与 `&&` 的行为略有不同, 因为 `&&` 会作用在所有“假”值上 (例如, 空字符串、0、NaN 以及 false), 但 `?.` 是一个仅作用于结构上的特性。 它不会在有效数据 (比如 0 或空字符串) 上进行短路计算。
可选链还包括两个另外的用法。 首先是 *可选元素访问*, 表现类似于可选属性访问, 但是也允许我们访问非标识符属性 (例如: 任意字符串、数字和 symbol):

```
/**
 * 如果 arr 是一个数组, 返回第一个元素
 * 否则返回 undefined
 */
function tryGetFirstElement<T>(arr?: T[]) {
    return arr?.[0];
    // 等价于:
    // return (arr === null || arr === undefined) ?
    //     undefined :
    //     arr[0];
}
```

另一个是 *可选调用*, 判断条件是当该表达式不是 null 或 undefined, 我们就可以调用它。

```
// 此模块:
console.log(import.meta.url)
```

```
// 获得如下的转变:
System.register([], function (exports, context) {
    return {
        setters: [],
        execute: function () {
            console.log(context.meta.url);
        }
    };
});
```

14.6 在环境上下文中允许 get 和 set 访问者

在以前的 TypeScript 版本中, 该语言不允许在环境上下文中使用 get 和 set 访问器 (例如, 在 declare-d 类中, 或者在 .d.ts 文件中)。理由是, 就这些属性的写作和阅读而言, 访问者与属性没有区别, 但是, [因为 ECMAScript 的类字段提议可能与现有版本的 TypeScript 具有不同的行为](#), 我们意识到我们需要一种方法来传达这种不同的行为, 以便在子类中提供适当的错误。

因此, 用户可以在 TypeScript 3.6 中的环境上下文中编写 getter 和 setter。

```
declare class Foo {
    // 3.6+ 允许
    get x(): number;
    set x(val: number): void;
}
```

在 TypeScript 3.7 中, 编译器本身将利用此功能, 以便生成的 .d.ts 文件也将生成 get / set 访问器。

14.7 环境类和函数可以合并

在以前版本的 TypeScript 中, 在任何情况下合并类和函数都是错误的。现在, 环境类和函数 (具有 declare 修饰符的类/函数或 .d.ts 文件中) 可以合并。这意味着现在您可以编写以下内容:

```
export declare function Point2D(x: number, y: number): Point2D;
export declare class Point2D {
    x: number;
    y: number;
    constructor(x: number, y: number);
}
```

而不需要使用

```
export interface Point2D {
    x: number;
    y: number;
}
export declare var Point2D: {
    (x: number, y: number): Point2D;
    new (x: number, y: number): Point2D;
}
```

这样做的一个优点是可以很容易地表达可调用的构造函数模式, 同时还允许名称空间与这些声明合并 (因为 var 声明不能与名称空间合并)。

ts_upgrade_from_2.2_to_5.3

在 TypeScript 3.7 中，编译器将利用此功能，以便从 .js 文件生成的 .d.ts 文件可以适当地捕获类函数的可调性和可构造性。

有关更多详细信息，请参阅 [GitHub 上的原始 PR](#)。

14.8 [APIs 支持 --build 和 --incremental](#)

TypeScript 3.0 引入了对引用其他项目的支持，并使用 --build 标志以增量方式构建它们。此外，TypeScript 3.4 引入了 --incremental 标志，用于保存有关以前编译的信息，仅重建某些文件。这些标志对于更灵活地构建项目和加速构建非常有用。不幸的是，使用这些标志不适用于 Gulp 和 Webpack 等第三方构建工具。TypeScript 3.6 现在公开了两组 API 来操作项目引用和增量构建。

对于创建 --incremental 构建，用户可以利用

createIncrementalProgram 和 createIncrementalCompilerHost API。用户还可以使用新公开的 readBuilderProgram 函数从此 API 生成的 .tsbuildinfo 文件中重新保存旧程序实例，该函数仅用于创建新程序（即，您无法修改返回的实例 - 它意味着用于其他 create * Program 函数中的 oldProgram 参数）。

为了利用项目引用，公开了一个新的 createSolutionBuilder 函数，它返回一个新类型 SolutionBuilder 的实例。

有关这些 API 的更多详细信息，您可以[查看原始 pull request](#)。

14.9 [新的 TypeScript Playground](#)

TypeScript Playground 已经获得了急需的刷新功能，并提供了便利的新功能！

Playground 主要是 [Artem Tyurin](#) 的 [TypeScript Playground](#) 的一个分支，社区成员越来越多地使用它。我们非常感谢 Artem 在这里提供帮助！

新的 Playground 现在支持许多新的选项，包括：

- target 选项（允许用户切换输出 es5 到 es3、es2015、esnext 等）
- 所有的严格检查标记（包括 just strict）
- 支持纯 JavaScript 文件（使用 allowJs 和可选的 checkJs）

当分享 Playground 的链接时，这些选项也会保存下来，允许用户更可靠地分享示例，而无需告诉受众“哦，别忘了打开 noImplicitAny 选项！”。

在不久的将来，我们将更新 Playground 样本，添加 JSX 支持和改进自动类型获取，这意味着您将能够在 Playground 上体验到与编辑器中相同的体验。

随着我们改进 Playground 和网站，我们欢迎 GitHub 上的 [issue 和 pull request](#)！

14.10 [代码编辑的分号感知](#)

对于 Visual Studio 和 Visual Studio Code 编辑器可以自动的应用快速修复、重构和自动从其它模块导入值等其它的转换。这些转换都由 TypeScript 来驱动，老版本的 TypeScript 无条件的在语句的末尾添加分号，不幸的是，这和大多数用户的代码风格不相符，并且，很多用户对于编辑器自动输入分号很不爽。

TypeScript 现在在应用这些简短的编辑的时候，已经足够的智能去检测你的文件分号的使用情况。如果你的文件通常缺少分号，TypeScript 就不会添加分号。

更多细节，查看[这些 pull request](#)。

14.11 [更智能的自动导入](#)

JavaScript 有大量不同的模块语法或者约定：EMAScript standard、CommonJS、AMD、System.js 等等。在大多数的情况下，TypeScript 默认使用 ECMAScript standard 语法自动导入，这在具有不同编译器设置的某些 TypeScript 项目中通常是不合适的，或者在使用纯 JavaScript 和需要调用的 Node 项目中。

在决定如何自动导入模块之前，TypeScript 3.6 现在会更加智能的查看你的现有导入。你可以通过[这些 pull request](#)查看更多细节。

[接下来？](#)

要了解团队将要开展的工作，请[查看今年 7 月至 12 月的 6 个月路线图](#)。

与往常一样，我们希望这个版本的 TypeScript 能让编码体验更好，让您更快乐。如果您有任何建议或遇到任何问题，我们总是感兴趣，所以随时[在 GitHub 上提一个 issue](#)。

[参考](#)

- [Announcing TypeScript 3.6](#)

就像类型谓词签名一样，这些断言签名具有清晰的表现力。 我们可以用它们表达一些非常复杂的想法。

```
function assertIsDefined<T>(val: T): asserts val is NonNullable<T> {
    if (val === undefined || val === null) {
        throw new AssertionError(
            `Expected 'val' to be defined, but received ${val}`
        );
    }
}
```

想了解更多断言签名的细节，可以 [查看原始的 PR](#)。

15.4 [更好地支持返回 never 的函数](#)

作为断言签名实现的一部分，TypeScript 需要编码更多关于调用位置和调用函数的细节。 这给了我们机会扩展对另一类函数的支持——返回 never 的函数。

返回 never 的函数，即永远不会返回的函数。 它表明抛出了异常、触发了停止错误条件、或程序退出的情况。 例如，[@types/node 中的 process.exit\(...\)](#) 就被指定为返回 never。

为了确保函数永远不会潜在地返回 undefined、或者从所有代码路径中有效地返回，TypeScript 需要借助一些语法标志——函数结尾处的 return 或 throw。 这样，使用者就会发现自己的代码在“返回”一个停机函数。

```
function dispatch(x: string | number): SomeType {
    if (typeof x === "string") {
        return doThingWithString(x);
    }
    else if (typeof x === "number") {
        return doThingWithNumber(x);
    }
    return process.exit(1);
}
```

现在，这些返回 never 的函数被调用时，TypeScript 能识别出它们将影响代码执行流程，同时说明原因。

```
function dispatch(x: string | number): SomeType {
    if (typeof x === "string") {
        return doThingWithString(x);
    }
    else if (typeof x === "number") {
        return doThingWithNumber(x);
    }
    process.exit(1);
}
```

你可以和在断言函数的 [同一个 PR 中查看更多细节](#)。

15.5 [（更加）递归的类型别名](#)

[Playground](#)

类型别名在“递归”引用方面一直存在局限性。 原因是，类型别名必须能用它代表的东西来代替自己。 这在某些情况下是不可能的，因此编译器会拒绝某些递归别名，比如下面这个：

type Foo = Foo;
这是一个合理的限制，因为任何对 Foo 的使用都可以替换为 Foo，同时这个 Foo 能够替换为 Foo，而这个 Foo 应该.....（产生了无限循环）希望你理解到这个意思了！ 到最后，没有类型可以用来代替 Foo。

```
async function makeRequest(url: string, log?: (msg: string) => void) {
    log?.(`Request started at ${new Date().toISOString()}`);
    // 基本等价于：
    // if (log != null) {
    //     log(`Request started at ${new Date().toISOString()}`);
    // }

    const result = (await fetch(url)).json();

    log?.(`Request finished at at ${new Date().toISOString()}`);

    return result;
}
```

可选链的“短路计算”行为仅限于属性访问、调用、元素访问——它不会延伸到后续的表达式中。也就是说，

```
let result = foo?.bar / someComputation()
可选链不会阻止除法运算或 someComputation() 的进行。 上面这段代码实际上等价于：
let temp = (foo === null || foo === undefined) ?
    undefined :
    foo.bar;
```

```
let result = temp / someComputation();
当然，这可能会使得 undefined 参与了除法运算，导致在 strictNullChecks 编译选项下产生报错。
function barPercentage(foo?: { bar: number }) {
    return foo?.bar / 100;
    // ~~~~~
    // Error: Object is possibly undefined.
}
```

想了解更多细节，你可以 [检阅完整的草案](#) 以及 [查看原始的 PR](#)。

15.2 [空值合并（Nullish Coalescing）](#)

[Playground](#)

空值合并运算符 是另一个即将到来的 ECMAScript 特性（与可选链一起），我们的团队也参与了 TC39 的讨论工作。

你可以考虑使用 ?? 运算符来实现：当字段是 null 或 undefined 时，“回退”到默认值。比如我们可以这样写代码：

```
let x = foo ?? bar();
这种新方式的意思是，当 foo “存在”时 x 等于 foo； 但假如 foo 是 null 或 undefined ，x 等于 bar() 的计算结果。
同样的，上面的代码可以写出等价代码。
let x = (foo !== null && foo !== undefined) ?
    foo :
    bar();
```

当尝试使用默认值时，?? 运算符可以代替 || 的作用。 例如，下面的代码片段尝试获取上一次储存在 localStorage 中的 volume（如果它已保存）；但是因为使用了 || ，留下一个 bug。

```
function initializeAudio() {
    let volume = localStorage.volume || 0.5
```

```
    // ...
  }
}
如果 localStorage.volume 的值是 0，这段代码将会把 volume 的值设置为 0.5，这是一个意外情况。而 ?? 避免了将 0、NaN 和 "" 视为假值的意外情况。
我们非常感谢社区成员 Wenlu Wang 和 Titian Cernicova Dragomir 实现了这个特性！想了解更多细节，你可以 查看他们的 PR 和 空值合并草案的 Repo。
```

15.3 断言函数

Playground

有一类特定的函数，用于在出现非预期结果时抛出一个错误。这样的函数叫做“断言”函数（Assertion Function）。比方说，Node.js 中就有一个名为 assert 的断言函数。
assert(someValue === 42);
在上面的例子中，如果 someValue 不等于 42，那么 assert 就会抛出一个 AssertionError 错误。
在 JavaScript 中，断言经常被用于防止不正确传参。举个例子：

```
function multiply(x, y) {
  assert(typeof x === "number");
  assert(typeof y === "number");

  return x * y;
}
很遗憾，在 TypeScript 中，这些检查没办法正确编码。对于类型宽松的代码，意味着 TypeScript 检查得更少，而对于更加规范的代码，通常迫使使用者添加类型断言。
function yell(str) {
  assert(typeof str === "string");

  return str.toUpperCase();
  // 糟了！我们拼错了 'toUpperCase'。
  // 如果 TypeScript 依然能检查出来就太棒了！
}
```

有一个替代的写法，可以让 TypeScript 能够分析出问题，不过这样并不方便。

```
function yell(str) {
  if (typeof str !== "string") {
    throw new TypeError("str should have been a string.")
  }
  // 发现错误！
  return str.toUpperCase();
}
```

归根结底，TypeScript 的目标是以最小的改动为现存的 JavaScript 结构添加上类型声明。因此，TypeScript 3.7 引入了一个称为“断言签名”的新概念，用于模拟这些断言函数。

第一种断言签名模拟了 Node 中 assert 函数的功能。它确保在断言的范围内，无论什么判断条件都为必须真。

```
function assert(condition: any, msg?: string): asserts condition {
  if (!condition) {
    throw new AssertionError(msg)
  }
}
```

asserts condition 表示：如果 assert 函数成功返回，则传入的 condition 参数必须为真（否则它应该抛出一个 Error）。这意味着对于同作用域中的后续代码，条件必须为真。回到例子上，用这个断言函数意味着我们 能够 捕获之前 yell 示例中的错误。

```
function yell(str) {
  assert(typeof str === "string");

  return str.toUpperCase();
  //      ~~~~~
  // error: Property 'toUpperCase' does not exist on type 'string'.
  //      Did you mean 'toUpperCase'?
}
```

```
function assert(condition: any, msg?: string): asserts condition {
  if (!condition) {
    throw new AssertionError(msg)
  }
}
```

另一种类型的断言签名不通过检查条件语句实现，而是在 TypeScript 里显式指定某个变量或属性具有不同的类型。

```
function assertIsString(val: any): asserts val is string {
  if (typeof val !== "string") {
    throw new AssertionError("Not a string!");
  }
}
```

这里的 asserts val is string 保证了在 assertIsString 调用之后，传入的任何变量都有可以被视为是 string 类型的。

```
function yell(str: any) {
  assertIsString(str);

  // 现在 TypeScript 知道 'str' 是一个 'string'。

  return str.toUpperCase();
  //      ~~~~~
  // error: Property 'toUpperCase' does not exist on type 'string'.
  //      Did you mean 'toUpperCase'?
}
```

这些断言方法签名类似于类型谓词（type predicate）签名：

```
function isString(val: any): val is string {
  return typeof val === "string";
}
```

```
function yell(str: any) {
  if (isString(str)) {
    return str.toUpperCase();
  }
  throw "Oops!";
}
```

```
export class Worker {
  constructor(maxDepth?: number);
  started: boolean;
  depthLimit: number;
  /**
   * NOTE: queued jobs may add more items to queue
   * @type {Job[]}
   */
  queue: Job[];
  /**
   * Adds a work item to the queue
   * @param {Job} work
   */
  push(work: Job): void;
  /**
   * Starts the queue if it has not yet started
   */
  start(): boolean;
}
export type Job = () => void;
```

注意，当同时启用这两个选项时，TypeScript 不一定必须得编译成 .js 文件。如果只是简单的想让 TypeScript 创建 .d.ts 文件，你可以启用 --emitDeclarationOnly 编译选项。

想了解更多细节，你可以 [查看原始的 PR](#)。

15.7 useDefineForClassFields 编译选项和 declare 属性修饰符

当在 TypeScript 中写类公共字段时，我们尽力保证以下代码

```
class C {
  foo = 100;
  bar: string;
}
等价于构造函数中的相似语句
class C {
  constructor() {
    this.foo = 100;
  }
}
```

不幸的是，虽然这符合该提案早期的发展方向，但类公共字段极有可能以不同的方式进行标准化。所以取而代之的，原始代码示例可能需要进行脱糖处理，变成类似下面的代码：

```
class C {
  constructor() {
    Object.defineProperty(this, "foo", {
      enumerable: true,
      configurable: true,
      writable: true,
      value: 100
    });
    Object.defineProperty(this, "bar", {
      enumerable: true,
```

[其他语言也是这么处理类型别名的](#)，但是它确实会产生一些令人困惑的情形，影响类型别名的使用。例如，在 TypeScript 3.6 和更低的版本中，下面的代码会报错：

```
type ValueOrArray<T> = T | Array<ValueOrArray<T>>;
// ~~~~~
// error: Type alias 'ValueOrArray' circularly references itself.
```

这很令人困惑，因为使用者总是可以用接口来编写具有相同作用的代码，那么从技术上讲这没什么问题。

```
type ValueOrArray<T> = T | Array<ValueOrArray<T>>;

interface ArrayOfValueOrArray<T> extends Array<ValueOrArray<T>> {}
```

因为接口（以及其他对象 type）引入了一个间接的层级，并且它们的完整结构不需要立即建立，所以 TypeScript 可以处理这种结构。

但是，对于使用者而言，引入接口的方案并不直观。并且，用了 Array 的初始版本 ValueOrArray 没什么原则性问题。如果编译器多一点“惰性”，并且只按需计算 Array 的类型参数，那么 TypeScript 就可以正确地表示出这些了。

这正是 TypeScript 3.7 引入的。在类型别名的“顶层”，TypeScript 将推迟解析类型参数以便支持这些模式。

这意味着，用于表示 JSON 的以下代码.....

```
type Json =
  | string
  | number
  | boolean
  | null
  | JsonObject
  | JsonArray;

interface JsonObject {
  [property: string]: Json;
}

interface JsonArray extends Array<Json> {}
```

终于可以重写成不需要借助 interface 的形式。

```
type Json =
  | string
  | number
  | boolean
  | null
  | { [property: string]: Json }
  | Json[];
```

这个新的机制让我们在元组中，同样也可以递归地使用类型别名。下面的 TypeScript 代码在以前会报错，但现在是合法的：

```
type VirtualNode =
  | string
  | [string, { [key: string]: any }, ...VirtualNode[]];

const myNode: VirtualNode =
  ["div", { id: "parent" },
    ["div", { id: "first-child" }, "I'm the first child"],
```

```
["div", { id: "second-child" }, "I'm the second child"]
];
```

想了解更多细节, 你可以 [查看原始的 PR](#)。

15.6 --declaration 和 --allowJs

--declaration 选项允许我们从 TypeScript 源文件 (诸如 .ts 和 .tsx 文件) 生成 .d.ts 文件 (声明文件)。`.d.ts` 文件的重要性有几个方面:

首先, 它们使得 TypeScript 能够对外部项目进行类型检查, 同时避免重复检查其源代码。

另一方面, 它们使得 TypeScript 能够与现存的 JavaScript 库相互配合, 即使这些库构建时并未使用 TypeScript。最后, 还有一个通常被忽略的好处: 在使用支持 TypeScript 的编辑器时, TypeScript 和 JavaScript 使用者都可以从这些文件中受益, 例如更高级的自动完成。

不幸的是, --declaration 不能与 --allowJs 选项一起使用, --allowJs 选项允许混合使用 TypeScript 和 JavaScript 文件。这是一个令人沮丧的限制, 因为它意味着使用者在迁移代码库时无法使用 --declaration 选项, 即使代码包含了 JSDoc 注释。

TypeScript 3.7 对此进行了改进, 允许这两个选项一起使用!

这个功能最大的影响可能比较微妙: 在 TypeScript 3.7 中, 编写带有 JSDoc 注释的 JavaScript 库, 也能帮助 TypeScript 的使用者。

它的实现原理是, 在启用 allowJs 时, TypeScript 会尽可能地分析并理解常见的 JavaScript 模式; 然而, 用 JavaScript 表达的某些模式看起来不一定像它们在 TypeScript 中的等效形式。启用 declaration 选项后, TypeScript 会尽力识别 JSDoc 注释和 CommonJS 形式的模块输出, 并转换为有效的类型声明输出到 .d.ts 文件上。

比如下面这个代码片段

```
const assert = require("assert")
```

```
module.exports.blurImage = blurImage;
```

```
/**
 * Produces a blurred image from an input buffer.
 *
 * @param input {Uint8Array}
 * @param width {number}
 * @param height {number}
 */
function blurImage(input, width, height) {
  const numPixels = width * height * 4;
  assert(input.length === numPixels);
  const result = new Uint8Array(numPixels);

  // TODO

  return result;
}
```

将会生成如下 .d.ts 文件

```
/**
 * Produces a blurred image from an input buffer.
 *
 * @param input {Uint8Array}
```

```
* @param width {number}
* @param height {number}
*/
export function blurImage(input: Uint8Array, width: number, height:
number): Uint8Array;
除了基本的带有 @param 标记的函数, 也支持其他情形, 请看下面这个例子:
/**
 * @callback Job
 * @returns {void}
 */

/** Queues work */
export class Worker {
  constructor(maxDepth = 10) {
    this.started = false;
    this.depthLimit = maxDepth;
  }
  /**
   * NOTE: queued jobs may add more items to queue
   * @type {Job[]}
   */
  this.queue = [];
}
/**
 * Adds a work item to the queue
 * @param {Job} work
 */
push(work) {
  if (this.queue.length + 1 > this.depthLimit) throw new
Error("Queue full!");
  this.queue.push(work);
}
/**
 * Starts the queue if it has not yet started
 */
start() {
  if (this.started) return false;
  this.started = true;
  while (this.queue.length) {
    /** @type {Job} */(this.queue.shift())();
  }
  return true;
}
}
```

会生成如下 .d.ts 文件:

```
/**
 * @callback Job
 * @returns {void}
 */
/** Queues work */
```



```
ts_upgrade_from_2.2_to_5.3
    if (user.notify) {
        // OK, 调用了该函数
        user.notify();
    }
}
```

如果你打算对该函数进行测试但不调用它，你可以修改它的类型定义，让它可能是 `undefined/null`，或使用 `!!` 来编写类似 `if (!!user.isAdministrator)` 的代码，表示代码逻辑确实是这样的。

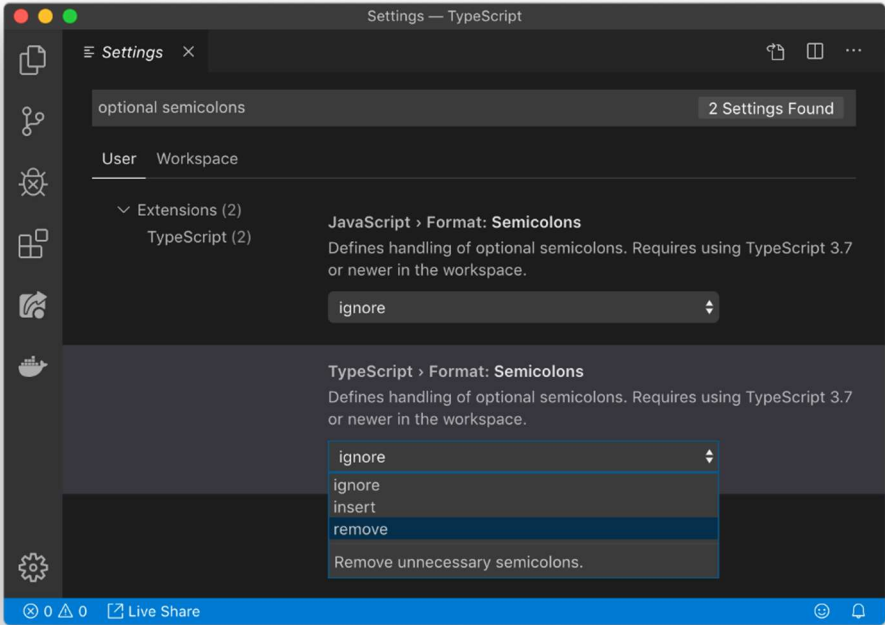
我们非常感谢社区成员 [@jwbay](#) 提出了 [这个问题的概念](#) 并持续跟进实现了 [这个需求的当前版本](#)。

15.10 [TypeScript 文件中的 // @ts-nocheck](#)

TypeScript 3.7 允许我们在 TypeScript 文件的顶部添加一行 `// @ts-nocheck` 注释来关闭语义检查。这个注释原本只在 `checkJs` 选项启用时的 JavaScript 源文件中有效，但我们扩展了它，让它能够支持 TypeScript 文件，这样所有使用者在迁移的时候会更方便。

15.11 [分号格式化选项](#)

JavaScript 有一个自动分号插入 (ASI, automatic semicolon insertion) 规则，TypeScript 内置的格式化程序现在能支持在可选的尾分号位置插入或删除分号。该设置现在在 [Visual Studio Code Insiders](#)，以及 Visual Studio 16.4 Preview 2 中的“工具选项”菜单中可用。



将值设定为“insert”或“remove”同时也会影响自动导入、类型提取、以及其他 TypeScript 服务提供的自动生成代码的格式。将设置保留为默认值“ignore”可以使生成代码的分号自动配置匹配当前文件的风格。

```
ts_upgrade_from_2.2_to_5.3
    configurable: true,
    writable: true,
    value: void 0
  });
}
```

当然，TypeScript 3.7 在默认情况下的编译结果与之前版本没有变化，我们增量地发布改动，以便帮助使用者减少未来潜在的破坏性变更。我们提供了一个新的编译选项 `useDefineForClassFields`，根据一些新的检查逻辑使用上面这种编译模式。最大的两个改变如下：

- 声明通过 `Object.defineProperty` 完成。
- 声明总是被初始化为 `undefined`，即使原有代码中没有显式的初始值。

对于现存的含有继承的代码，这可能会造成一些问题。首先，基类的 `set` 访问器不再被触发——它们将被完全覆写。

```
class Base {
  set data(value: string) {
    console.log("data changed to " + value);
  }
}
```

```
class Derived extends Base {
  // 当启用 'useDefineForClassFields' 时
  // 不再触发 'console.log'
  data = 10;
}
```

其次，基类中的属性设定也将不起作用。

```
interface Animal { animalStuff: any }
interface Dog extends Animal { dogStuff: any }
```

```
class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}
```

```
class DogHouse extends AnimalHouse {
  // 当启用 'useDefineForClassFields' 时
  // 调用 'super()' 后
  // 'resident' 只会被初始化成 'undefined'!
  resident: Dog;

  constructor(dog: Dog) {
    super(dog);
  }
}
```

这两个问题归结为，继承时混合覆写属性与访问器，以及属性不带初始值的重新声明。

ts_upgrade_from_2.2_to_5.3

为了检测这个访问器的问题，TypeScript 3.7 现在可以在 `.d.ts` 文件中编译出 `get/set`，这样 TypeScript 就能检查出访问器覆写的情况。

对于改变类字段的代码，将字段初始化写成构造函数内的语句，就可以解决此问题。

```
class Base {
  set data(value: string) {
    console.log("data changed to " + value);
  }
}
```

```
class Derived extends Base {
  constructor() {
    data = 10;
  }
}
```

而解决第二个问题，你可以显式地提供一个初始值，或添加一个 `declare` 修饰符来表示这个属性不要被编译。

```
interface Animal { animalStuff: any }
interface Dog extends Animal { dogStuff: any }
```

```
class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}
```

```
class DogHouse extends AnimalHouse {
  declare resident: Dog;
  // ^^^^^^^
  // 'resident' now has a 'declare' modifier,
  // and won't produce any output code.
```

```
  constructor(dog: Dog) {
    super(dog);
  }
}
```

目前，只有当编译目标是 ES5 及以上时 `useDefineForClassFields` 才可用，因为 ES3 中不支持 `Object.defineProperty`。要检查类似的问题，你可以创建一个分离的项目，设定编译目标为 ES5 并使用 `--noEmit` 来避免完全构建。

想了解更多细节，你可以 [去原始的 PR 查看这些改动](#)。

我们强烈建议使用者尝试 `useDefineForClassFields`，并在 `issues` 或下面的评论区域中提供反馈。应该碰到编译选项在使用难度上的反馈，这样我们就能够了解如何使迁移变得更容易。

15.8 [利用项目引用实现无构建编辑](#)

TypeScript 的项目引用功能，为我们提供了一种简单的方法来分解代码库，从而使编译速度更快。遗憾的是，当我们编辑一个依赖未曾构建（或者构建结果过时）的项目时，体验不好。在 TypeScript 3.7 中，当打开一个带有依赖的项目时，TypeScript 将自动切换为使用依赖中的 `.ts/.tsx` 源码文件。这意味着在带有外部引用的项目中，代码的修改会即时同步和

生效，编码体验会得到提升。你也可以适当地打开编译器选

项 `disableSourceOfProjectReferenceRedirect` 来禁用这个引用的功能，因为在超大型项目中这个功能可能会影响性能。

你可以 [阅读这个 PR 来了解这个改动的更多细节](#)。

15.9 [检查未调用的函数](#)

一个常见且危险的错误是：忘记调用一个函数，特别是当该函数不需要参数，或者它的命名容易被误认为是一个属性而不是函数时。

```
interface User {
  isAdministrator(): boolean;
  notify(): void;
  doNotDisturb?(): boolean;
}
```

// 之后...

// 有问题的代码，别用！

```
function doAdminThing(user: User) {
  // 糟了！
  if (user.isAdministrator) {
    sudo();
    editTheConfiguration();
  }
  else {
    throw new AccessDeniedError("User is not an admin");
  }
}
```

在这段代码中，我们忘了调用 `isAdministrator`，导致该代码错误地允许非管理员用户修改配置！

在 TypeScript 3.7 中，它会被识别成一个潜在的错误：

```
function doAdminThing(user: User) {
  if (user.isAdministrator) {
    // ~~~~~
    // error! This condition will always return true since the function
    is always defined.
    // Did you mean to call it instead?
```

这个检查功能是一个破坏性变更，基于这个因素，检查会非常保守。因此对这类错误的提示仅限于 `if` 条件语句中。当问题函数是可选属性、或未开启 `strictNullChecks` 选项、或该函数在 `if` 的代码块中有被调用，在这些情况下不会被视为错误：

```
interface User {
  isAdministrator(): boolean;
  notify(): void;
  doNotDisturb?(): boolean;
}
```

```
function issueNotification(user: User) {
  if (user.doNotDisturb) {
    // OK, 属性是可选的
  }
}
```

- TypeScript accessibility modifiers like public or private can't be used on private fields.
- Private fields can't be accessed or even detected outside of the containing class - even by JS users! Sometimes we call this *hard privacy*.

Apart from "hard" privacy, another benefit of private fields is that uniqueness we just mentioned. For example, regular property declarations are prone to being overwritten in subclasses.

```
class C {
    foo = 10;

    cHelper() {
        return this.foo;
    }
}
```

```
class D extends C {
    foo = 20;

    dHelper() {
        return this.foo;
    }
}
```

```
let instance = new D();
// 'this.foo' refers to the same property on each instance.
console.log(instance.cHelper()); // prints '20'
console.log(instance.dHelper()); // prints '20'
```

With private fields, you'll never have to worry about this, since each field name is unique to the containing class.

```
class C {
    #foo = 10;

    cHelper() {
        return this.#foo;
    }
}
```

```
class D extends C {
    #foo = 20;

    dHelper() {
        return this.#foo;
    }
}
```

```
let instance = new D();
// 'this.#foo' refers to a different field within each class.
console.log(instance.cHelper()); // prints '10'
```

15.12 [3.7 的破坏性变更](#)

15.12.1 [DOM 变更](#)

[lib.dom.d.ts 中的类型声明已更新](#)。这些变更大部分是与空值检查有关的检测准确性变更，最终的影响取决于你的代码库。

15.12.2 [类字段处理](#)

[正如上文提到的](#)，TypeScript 3.7 现在能够在 .d.ts 文件中编译出 get/set，这可能会对 3.5 和更低版本的 TypeScript 使用者来说是破坏性变更。TypeScript 3.6 的使用者不会受影响，因为该版本对这个功能已经进行了预兼容。

useDefineForClassFields 选项虽然自身没有破坏性变更，但不排除以下情形：

- 在派生类中用属性声明覆盖了基类的访问器
- 覆盖声明属性，但是没有初始值

要了解全部的影响，请查看 [上面关于 useDefineForClassFields 的章节](#)。

15.12.3 [函数真值检查](#)

正如上文提到的，现在当函数在 if 条件语句中未被调用时 TypeScript 会报错。当 if 条件语句中判断的是函数时将会报错，除非符合以下情形：

- 该函数是可选属性
- 未开启 strictNullChecks 选项
- 该函数在 if 的代码块中有被调用

15.12.4 [本地和导入的类型声明现在会产生冲突](#)

TypeScript 之前有一个 bug，导致允许以下代码结构：

```
// ./someOtherModule.ts
```

```
interface SomeType {
    y: string;
}
```

```
// ./myModule.ts
```

```
import { SomeType } from "./someOtherModule";
export interface SomeType {
    x: number;
}
```

```
function fn(arg: SomeType) {
    console.log(arg.x); // Error! 'x' doesn't exist on 'SomeType'
}
```

这里，SomeType 同时来源于 import 声明和本地 interface 声明。出人意料的是，在模块内部，SomeType 只会指向 import 的定义，而本地声明的 SomeType 仅在另一个文件的导入中起效。这很令人困惑，我们对类似的个案进行的调查表明，广大开发者通常理解的情况不一样。

在 TypeScript 3.7 中，[这个问题中的重复声明现在可以被正确地识别为一个错误](#)。合理的修复方案取决于开发者的原始意图，并应该逐案解决。通常，命名冲突不是故意的，最好的办法是重命名导入的那个类型。如果是要扩展导入的类型，则可以编写模块扩展 (module augmentation) 来代替。

15.12.5 [3.7 API 变化](#)

为了实现上文中提到的递归的类型别名模式，TypeReference 接口已经移除了 typeArguments 属性。开发者应该在 TypeChecker 实例上使用 getTypeArguments 函数来代替。

16 v3.8

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.8.html>

- [Type-Only Imports and Exports](#)
- [ECMAScript Private Fields](#)
- [export * as ns_Syntax](#)
- [Top-Level await](#)
- [JSDoc Property Modifiers](#)
- [Better Directory Watching on Linux and watchOptions](#)
- ["Fast and Loose" Incremental Checking](#)

16.1 类型导入和导出 (Type-Only Imports and Exports)

This feature is something most users may never have to think about; however, if you've hit issues under `--isolatedModules`, TypeScript's `transpileModule` API, or Babel, this feature might be relevant.

TypeScript 3.8 adds a new syntax for type-only imports and exports.

```
import type { Something } from "./some-module.js";
```

```
export type { Something };
```

`import type` only imports declarations to be used for type annotations and declarations. It *always* gets fully erased, so there's no remnant of it at runtime. Similarly, `export type` only provides an export that can be used for type contexts, and is also erased from TypeScript's output.

It's important to note that classes have a value at runtime and a type at design-time, and the use is context-sensitive. When using `import type` to import a class, you can't do things like `extend` from it.

```
import type { Component } from "react";
```

```
interface ButtonProps {
  // ...
}
```

```
class Button extends Component<ButtonProps> {
  // ~~~~~~
  // error! 'Component' only refers to a type, but is being used as a
  value here.
```

```
  // ...
}
```

If you've used Flow before, the syntax is fairly similar. One difference is that we've added a few restrictions to avoid code that might appear ambiguous.

```
// Is only 'Foo' a type? Or every declaration in the import?
// We just give an error because it's not clear.
```

```
import type Foo, { Bar, Baz } from "some-module";
```

```
// ~~~~~~
```

```
// error! A type-only import can specify a default import or named
bindings, but not both.
```

In conjunction with `import type`, TypeScript 3.8 also adds a new compiler flag to control what happens with imports that won't be utilized at runtime: `importsNotUsedAsValues`. This flag takes 3 different values:

- `remove`: this is today's behavior of dropping these imports. It's going to continue to be the default, and is a non-breaking change.
- `preserve`: this *preserves* all imports whose values are never used. This can cause imports/side-effects to be preserved.
- `error`: this preserves all imports (the same as the `preserve` option), but will error when a value import is only used as a type. This might be useful if you want to ensure no values are being accidentally imported, but still make side-effect imports explicit.

For more information about the feature, you can [take a look at the pull request](#), and [relevant changes](#) around broadening where imports from an `import type` declaration can be used.

16.2 ECMAScript 私有变量 (ECMAScript Private Fields)

TypeScript 3.8 brings support for ECMAScript's private fields, part of the [stage-3 class fields proposal](#).

```
class Person {
  #name: string

  constructor(name: string) {
    this.#name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.#name}!`);
  }
}
```

```
let jeremy = new Person("Jeremy Bearimy");
```

```
jeremy.#name
// ~~~~~
// Property '#name' is not accessible outside class 'Person'
// because it has a private identifier.
```

Unlike regular properties (even ones declared with the `private` modifier), private fields have a few rules to keep in mind. Some of them are:

- Private fields start with a `#` character. Sometimes we call these *private names*.
- Every private field name is uniquely scoped to its containing class.

ts_upgrade_from_2.2_to_5.3

body of an async function. However, with top-level await, we can use await at the top level of a module.

```
const response = await fetch("...");
const greeting = await response.text();
console.log(greeting);
```

```
// Make sure we're a module
```

```
export {};
```

Note there's a subtlety: top-level await only works at the top level of a *module*, and files are only considered modules when TypeScript finds an import or an export. In some basic cases, you might need to write out export {} as some boilerplate to make sure of this. Top level await may not work in all environments where you might expect at this point. Currently, you can only use top level await when the target compiler option is es2017 or above, and module is esnext or system. Support within several environments and bundlers may be limited or may require enabling experimental support.

For more information on our implementation, you can [check out the original pull request](#).

16.5 es2020 for target and module

TypeScript 3.8 supports es2020 as an option for module and target. This will preserve newer ECMAScript 2020 features like optional chaining, nullish coalescing, export * as ns, and dynamic import(...) syntax. It also means bigint literals now have a stable target below esnext.

16.6 JSDoc 属性修饰词(JSDoc Property Modifiers)

TypeScript 3.8 supports JavaScript files by turning on the allowJs flag, and also supports *type-checking* those JavaScript files via the checkJs option or by adding a // @ts-check comment to the top of your .js files.

Because JavaScript files don't have dedicated syntax for type-checking, TypeScript leverages JSDoc. TypeScript 3.8 understands a few new JSDoc tags for properties.

First are the accessibility modifiers: @public, @private, and @protected. These tags work exactly like public, private, and protected respectively work in TypeScript.

```
// @ts-check
```

```
class Foo {
  constructor() {
    /** @private */
    this.stuff = 100;
  }

  printStuff() {
    console.log(this.stuff);
  }
}
```

88

```
console.log(instance.dHelper()); // prints '20'
```

Another thing worth noting is that accessing a private field on any other type will result in a TypeError!

```
class Square {
  #sideLength: number;

  constructor(sideLength: number) {
    this.#sideLength = sideLength;
  }

  equals(other: any) {
    return this.#sideLength === other.#sideLength;
  }
}
```

```
const a = new Square(100);
const b = { sideLength: 100 };
```

```
// Boom!
```

```
// TypeError: attempted to get private field on non-instance
```

```
// This fails because 'b' is not an instance of 'Square'.
```

```
console.log(a.equals(b));
```

Finally, for any plain .js file users, private fields *always* have to be declared before they're assigned to.

```
class C {
  // No declaration for '#foo'
  // :(

  constructor(foo: number) {
    // SyntaxError!
    // '#foo' needs to be declared before writing to it.
    this.#foo = foo;
  }
}
```

JavaScript has always allowed users to access undeclared properties, whereas TypeScript has always required declarations for class properties. With private fields, declarations are always needed regardless of whether we're working in .js or .ts files.

```
class C {
  /** @type {number} */
  #foo;

  constructor(foo: number) {
    // This works.
    this.#foo = foo;
  }
}
```

For more information about the implementation, you can [check out the original pull request](#)

16.2.1 Which should I use?

We've already received many questions on which type of privates you should use as a TypeScript user: most commonly, "should I use the private keyword, or ECMAScript's hash/pound (#) private fields?" It depends!

When it comes to properties, TypeScript's private modifiers are fully erased - that means that at runtime, it acts entirely like a normal property and there's no way to tell that it was declared with a private modifier. When using the private` keyword, privacy is only enforced at compile-time/design-time, and for JavaScript consumers it's entirely intent-based.

```
class C {
  private foo = 10;
}

// This is an error at compile time,
// but when TypeScript outputs .js files,
// it'll run fine and print '10'.
console.log(new C().foo);    // prints '10'
//          ~~~
// error! Property 'foo' is private and only accessible within class
// 'C'.
```

TypeScript allows this at compile-time as a "work-around" to avoid the error.

```
console.log(new C()["foo"]); // prints '10'
```

The upside is that this sort of "soft privacy" can help your consumers temporarily work around not having access to some API, and also works in any runtime.

On the other hand, ECMAScript's # privates are completely inaccessible outside of the class.

```
class C {
  #foo = 10;
}

console.log(new C().#foo); // SyntaxError
//          ~~~~
// TypeScript reports an error *and*
// this won't work at runtime!
```

```
console.log(new C()["#foo"]); // prints undefined
//          ~~~~~~
// TypeScript reports an error under 'noImplicitAny',
// and this prints 'undefined'.
```

This hard privacy is really useful for strictly ensuring that nobody can take use of any of your internals. If you're a library author, removing or renaming a private field should never cause a breaking change.

As we mentioned, another benefit is that subclassing can be easier with ECMAScript's # privates because they *really* are private. When using ECMAScript # private fields, no subclass ever has to worry about collisions in field naming. When it comes to TypeScript's private property declarations, users still have to be careful not to trample over properties declared in superclasses. One more thing to think about is where you intend for your code to run. TypeScript currently can't support this feature unless targeting ECMAScript 2015 (ES6) targets or higher. This is because our downleveled implementation uses WeakMaps to enforce privacy, and WeakMaps can't be polyfilled in a way that doesn't cause memory leaks. In contrast, TypeScript's private-declared properties work with all targets - even ECMAScript 3!

A final consideration might be speed: private properties are no different from any other property, so accessing them is as fast as any other property access no matter which runtime you target. In contrast, because # private fields are downleveled using WeakMaps, they may be slower to use. While some runtimes might optimize their actual implementations of # private fields, and even have speedy WeakMap implementations, that might not be the case in all runtimes.

16.3 export * as ns Syntax

It's often common to have a single entry-point that exposes all the members of another module as a single member.

```
import * as utilities from "./utilities.js";
export { utilities };
```

This is so common that ECMAScript 2020 recently added a new syntax to support this pattern!

```
export * as utilities from "./utilities.js";
```

This is a nice quality-of-life improvement to JavaScript, and TypeScript 3.8 implements this syntax. When your module target is earlier than es2020, TypeScript will output something along the lines of the first code snippet.

16.4 顶层 await (Top-Level await)

TypeScript 3.8 provides support for a handy upcoming ECMAScript feature called "top-level await".

JavaScript users often introduce an async function in order to use await, and then immediately called the function after defining it.

```
async function main() {
  const response = await fetch("...");
  const greeting = await response.text();
  console.log(greeting);
}
```

```
main()
  .catch(e => console.error(e))
```

This is because previously in JavaScript (along with most other languages with a similar feature), await was only allowed within the

17 v3.9

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-3.9.html>

17.1 [改进类型推断](#)和 **Promise.all**

TypeScript 的最近几个版本（3.7 前后）更新了像 Promise.all 和 Promise.race 等的函数声明。 不巧的是，它引入了一些回归问题，尤其是在和 null 或 undefined 混合使用的场景中。

```
interface Lion {
  roar(): void;
}
```

```
interface Seal {
  singKissFromARose(): void;
}
```

```
async function visitZoo(
  lionExhibit: Promise<Lion>,
  sealExhibit: Promise<Seal | undefined>
) {
  let [lion, seal] = await Promise.all([lionExhibit, sealExhibit]);
  lion.roar();
  // ~~~~
  // 对象可能为 'undefined'
}
```

这是一种奇怪的行为！事实上，只有 sealExhibit 包含了 undefined 值，但是它却让 lion 也含有了 undefined 值。

得益于 [Jack Bates](#) 提交的 [PR](#)，这个问题已经被修复了，它改进了 TypeScript 3.9 中的类型推断流程。 上面的例子中已经不再产生错误。 如果你在旧版本的 TypeScript 中被 Promise 的这个问题所困扰，我们建议你尝试一下 3.9 版本！

17.1.1 [awaited 类型](#)

如果你一直关注 TypeScript，那么你可能会注意到[一个新的类型运算符 awaited](#)。 这个类型运算符的作用是准确地表达 JavaScript 中 Promise 的工作方式。

我们原计划在 TypeScript 3.9 中支持 awaited，但在现有的代码中测试过该特性后，我们发现还需要进行一些设计，以便让所有人能够顺利地使用它。 因此，我们从主分支中暂时移除了这个特性。 我们将继续试验这个特性，它不会被包含进本次发布。

17.2 [速度优化](#)

TypeScript 3.9 提供了多项速度优化。 TypeScript 在 material-ui 和 styled-components 代码包中拥有非常慢的编辑速度和编译速度。在发现了这点后，TypeScript 团队集中了精力解决性能问题。 TypeScript 优化了大型联合类型、交叉类型、有条件类型和映射类型。

- <https://github.com/microsoft/TypeScript/pull/36576>
- <https://github.com/microsoft/TypeScript/pull/36590>
- <https://github.com/microsoft/TypeScript/pull/36607>
- <https://github.com/microsoft/TypeScript/pull/36622>
- <https://github.com/microsoft/TypeScript/pull/36754>
- <https://github.com/microsoft/TypeScript/pull/36696>

```
new Foo().stuff;
// ~~~~~
// error! Property 'stuff' is private and only accessible within class 'Foo'.
```

- @public 是默认的，可以省略，它代表了一个属性可以从任何地方访问它
- @private 表示一个属性只能在包含的类中访问
- @protected 表示该属性只能在所包含的类及子类中访问，但不能在类的实例中访问

下一步，我们计划添加 @readonly 修饰符，来确保一个属性只能在初始化时被修改：

```
// @ts-check
```

```
class Foo {
  constructor() {
    /** @readonly */
    this.stuff = 100;
  }

  writeToStuff() {
    this.stuff = 200;
    // ~~~~~
    // Cannot assign to 'stuff' because it is a read-only property.
  }
}
```

```
new Foo().stuff++;
// ~~~~~
// Cannot assign to 'stuff' because it is a read-only property.
```

16.7 [Better Directory Watching on Linux and watchOptions](#)

TypeScript 3.8 ships a new strategy for watching directories, which is crucial for efficiently picking up changes to node_modules. For some context, on operating systems like Linux, TypeScript installs directory watchers (as opposed to file watchers) on node_modules and many of its subdirectories to detect changes in dependencies. This is because the number of available file watchers is often eclipsed by the of files in node_modules, whereas there are way fewer directories to track.

Older versions of TypeScript would *immediately* install directory watchers on folders, and at startup that would be fine; however, during an npm install, a lot of activity will take place within node_modules and that can overwhelm TypeScript, often slowing editor sessions to a crawl. To prevent this, TypeScript 3.8 waits slightly before installing directory watchers to give these highly volatile directories some time to stabilize.

Because every project might work better under different strategies, and this new approach might not work well for your workflows, TypeScript 3.8 introduces a new watchOptions field in tsconfig.json and jsconfig.json which allows users to tell the

compiler/language service which watching strategies should be used to keep track of files and directories.

```
{
  // Some typical compiler options
  "compilerOptions": {
    "target": "es2020",
    "moduleResolution": "node",
    // ...
  },

  // NEW: Options for file/directory watching
  "watchOptions": {
    // Use native file system events for files and directories
    "watchFile": "useFsEvents",
    "watchDirectory": "useFsEvents",

    // Poll files for updates more frequently
    // when they're updated a lot.
    "fallbackPolling": "dynamicPriority"
  }
}
```

watchOptions 包含四种新的选项：

- **watchFile**：监听单个文件的策略，它可以有以下值
 - **fixedPollingInterval**：以固定的时间间隔，检查文件的更改
 - **priorityPollingInterval**：以固定的时间间隔，检查文件的更改，但是使用「heuristics」检查某些类型的文件的频率比其他文件低（heuristics 怎么翻？）
 - **dynamicPriorityPolling**：使用动态队列，在该队列中，较少检查不经常修改的文件
 - **useFsEvents**（默认）：尝试使用操作系统/文件系统原生事件来监听文件更改
 - **useFsEventsOnParentDirectory**：尝试使用操作系统/文件系统原生事件来监听文件、目录的更改，这样可以使用较小的文件监听程序，但是准确性可能较低
- **watchDirectory**：在缺少递归文件监听功能的系统中，使用哪种策略监听整个目录树，它可以有以下值：
 - **fixedPollingInterval**：以固定的时间间隔，检查目录树的更改
 - **dynamicPriorityPolling**：使用动态队列，在该队列中，较少检查不经常修改的目录
 - **useFsEvents**（默认）：尝试使用操作系统/文件系统原生事件来监听目录更改
- **fallbackPolling**：当使用文件系统的事件，该选项用来指定使用特定策略，它可以有以下值
 - **fixedPollingInterval**：(同上)
 - **priorityPollingInterval**：(同上)
 - **dynamicPriorityPolling**：(同上)

- **synchronousWatchDirectory**：在目录上禁用延迟监听功能。在可能一次发生大量文件（如 node_modules）更改时，它非常有用，但是你可能需要一些不太常见的设置时，禁用它。

For more information on these changes, [head over to GitHub to see the pull request](#) to read more.

16.8 "Fast and Loose" Incremental Checking

TypeScript 3.8 introduces a new compiler option called `assumeChangesOnlyAffectDirectDependencies`. When this option is enabled, TypeScript will avoid rechecking/rebuilding all truly possibly-affected files, and only recheck/rebuild files that have changed as well as files that directly import them.

For example, consider a file `fileD.ts` that imports `fileC.ts` that imports `fileB.ts` that imports `fileA.ts` as follows:

```
fileA.ts <- fileB.ts <- fileC.ts <- fileD.ts
```

In `--watch` mode, a change in `fileA.ts` would typically mean that TypeScript would need to at least re-check `fileB.ts`, `fileC.ts`, and `fileD.ts`. Under `assumeChangesOnlyAffectDirectDependencies`, a change in `fileA.ts` means that only `fileA.ts` and `fileB.ts` need to be re-checked.

In a codebase like Visual Studio Code, this reduced rebuild times for changes in certain files from about 14 seconds to about 1 second.

While we don't necessarily recommend this option for all codebases, you might be interested if you have an extremely large codebase and are willing to defer full project errors until later (e.g. a dedicated build via a `tsconfig.fullbuild.json` or in CI).

For more details, you can [see the original pull request](#).

```
const maxValue = 100;

for (let i = 0; i <= maxValue; i++) {
    // First get the squared value.
    let square = i ** 2;

    // Now print the squared value.
    console.log(square);
}
```

这不是我们想要的 - 在 for 循环中，每条语句之间都有一个空行，但是重构后它们被移除了！TypeScript 3.9 调整后，它会保留我们编写的代码。

```
const maxValue = 100;

printSquares();

function printSquares() {
    for (let i = 0; i <= maxValue; i++) {
        // First get the squared value.
        let square = i ** 2;

        // Now print the squared value.
        console.log(square);
    }
}
```

上面列出的每一个 PR 都能够减少 5-10% 的编译时间（对于某些代码库）。对于 material-ui 库而言，现在能够节约大约 40% 的编译时间！

我们还调整了在编辑器中的文件重命名功能。从 Visual Studio Code 团队处得知，当重命名一个文件时，计算出需要更新的 import 语句要花费 5 到 10 秒的时间。TypeScript 3.9 通过[改变编译器和语言服务缓存文件查询的内部实现](#)解决了这个问题。

尽管仍有优化的空间，我们希望当前的改变能够为每个人带来更流畅的体验。

17.3 [// @ts-expect-error 注释](#)

设想一下，我们正在使用 TypeScript 编写一个代码库，它对外开放了一个公共函数 doStuff。该函数的类型声明了它接受两个 string 类型的参数，因此其它 TypeScript 的用户能够看到类型检查的结果，但该函数还进行了运行时的检查以便 JavaScript 用户能够看到一个有帮助的错误。

```
function doStuff(abc: string, xyz: string) {
    assert(typeof abc === "string");
    assert(typeof xyz === "string");
```

```
    // do some stuff
}
```

如果有人错误地使用了该函数，那么 TypeScript 用户能够看到红色的波浪线和错误提示，JavaScript 用户会看到断言错误。然后，我们想编写一条单元测试来测试该行为。

```
expect(() => {
    doStuff(123, 456);
}).toThrow();
```

不巧的是，如果你使用 TypeScript 来编译单元测试，TypeScript 会提示一个错误！

```
doStuff(123, 456);
// ~~~
```

// 错误：类型 'number' 不能够赋值给类型 'string'。

这就是 TypeScript 3.9 添加了 `// @ts-expect-error` 注释的原因。当一行代码带有 `// @ts-expect-error` 注释时，TypeScript 不会提示上例的错误；但如果该行代码没有错误，TypeScript 会提示没有必要使用 `// @ts-expect-error`。

示例，以下的代码是正确的：

```
// @ts-expect-error
console.log(47 * "octopus");
```

但是下面的代码：

```
// @ts-expect-error
console.log(1 + 1);
```

会产生错误：

未使用的 '`@ts-expect-error`' 指令。

非常感谢 [Josh Goldberg](#) 实现了这个功能。更多信息请参考 [the ts-expect-error pull request](#)。

17.3.1 [ts-ignore 还是 ts-expect-error?](#)

某些情况下，`// @ts-expect-error` 和 `// @ts-ignore` 是相似的，都能够阻止产生错误消息。两者的不同在于，如果下一行代码没有错误，那么 `// @ts-ignore` 不会做任何事。

你可能会想要抛弃 `// @ts-ignore` 注释转而去使用 `// @ts-expect-error`，并且想要知道哪一个更适用于以后的代码。实际上，这完全取决于你和你的团队，下面列举了一些具体情况。

如果满足以下条件，那么选择 `ts-expect-error`：

- 你在编写单元测试，并且想让类型系统提示错误

- 你知道此处有问题，并且很快会回来改正它，只是暂时地忽略该错误
- 你的团队成员都很积极，大家想要在代码回归正常后及时地删除忽略类型检查注释

如果满足以下条件，那么选择 ts-ignore:

- 项目规模较大，产生了一些错误但是找不到相应代码的负责人
- 正处于 TypeScript 版本升级的过程中，某些错误只在特定版本的 TypeScript 中存在，但是在其它版本中并不存在
- 你没有足够的时间考虑究竟应该使用 // @ts-ignore 还是 // @ts-expect-error

17.4 在条件表达式中检查未被调用的函数

在 TypeScript 3.7 中，我们引入了_未进行函数调用的检查_，当你忘记去调用某个函数时会产生错误。

```
function hasImportantPermissions(): boolean {  
    // ...  
}
```

```
// Oops!  
if (hasImportantPermissions) {  
    // ~~~~~  
    // 这个条件永远返回 true，因为函数已经被定义。  
    // 你是否想要调用该函数？  
    deleteAllTheImportantFiles();  
}  
然而，这个错误只会在 if 条件语句中才会提示。 多亏了 Alexander Tarasyuk 提交的 PR，  
现在这个特性也支持在三元表达式中使用，例如 cond ? trueExpr : falseExpr。  
declare function listFilesOfDirectory(dirPath: string): string[];  
declare function isDirectory(): boolean;
```

```
function getAllFiles(startFileName: string) {  
    const result: string[] = [];  
    traverse(startFileName);  
    return result;  
  
    function traverse(currentPath: string) {  
        return isDirectory  
            ? // ~~~~~  
              // 该条件永远返回 true  
              // 因为函数已经被定义。  
              // 你是否想要调用该函数？  
              listFilesOfDirectory(currentPath).forEach(traverse)  
            : result.push(currentPath);  
    }  
}
```

https://github.com/microsoft/TypeScript/issues/36048

17.5 编辑器改进

TypeScript 编译器不但支持在大部分编辑器中编写 TypeScript 代码，还支持着在 Visual Studio 系列的编辑器中编写 JavaScript 代码。 针对不同的编辑器，在使用 TypeScript/JavaScript 的新功能时可能会有所区别，但是

- Visual Studio Code 支持选择不同的 TypeScript 版本。或者，安装 JavaScript/TypeScript Nightly Extension 插件来使用最新的版本。
- Visual Studio 2017/2019 提供了 SDK 安装包，以及 MSBuild 安装包。
- Sublime Text 3 支持选择不同的 TypeScript 版本

17.5.1 在 JavaScript 中自动导入 CommonJS 模块

在使用了 CommonJS 模块的 JavaScript 文件中，我们对自动导入功能进行了一个非常棒的改进。

在旧的版本中，TypeScript 总是假设你想要使用 ECMAScript 模块风格的导入语句，并且无视你的文件类型。

```
import * as fs from "fs";  
然而，在编写 JavaScript 文件时，并不总是想要使用 ECMAScript 模块风格。 非常多的用户仍然在使用 CommonJS 模块，例如 require(...)。  
const fs = require("fs");
```

现在，TypeScript 会自动检测你正在使用的导入语句风格，并使用当前的导入语句风格。更新信息请参考 PR。

17.5.2 Code Actions 保留换行符

TypeScript 的重构工具和快速修复工具对换行符的处理不是非常好。 一个基本的示例如下。

```
const maxValue = 100;  
  
/*start*/  
for (let i = 0; i <= maxValue; i++) {  
    // First get the squared value.  
    let square = i ** 2;
```

```
    // Now print the squared value.  
    console.log(square);  
}
```

/*end*/
如果我们选中从/*start*/到/*end*/，然后进行“提取到函数”操作，我们会得到如下的代码。

```
const maxValue = 100;  
  
printSquares();  
  
function printSquares() {  
    for (let i = 0; i <= maxValue; i++) {  
        // First get the squared value.  
        let square = i ** 2;  
        // Now print the squared value.  
        console.log(square);  
    }  
}
```


ts upgrade from 2.2 to 5.3

第一个改动是展开元组类型的语法支持泛型。这就是说，我们能够表示在元组和数组上的高阶操作，尽管我们不清楚它们的具体类型。在实例化泛型展开时当在这类元组上进行泛型展开实例化（或者使用实际类型参数进行替换）时，它们能够产生另一组数组和元组类型。

例如，我们可以像下面这样给 tail 函数添加类型，避免了“重载的折磨”。

```
function tail<T extends any[]>(arr: readonly [any, ...T]) {
  const [_ignored, ...rest] = arr;
  return rest;
}
```

```
const myTuple = [1, 2, 3, 4] as const;
const myArray = ["hello", "world"];
```

```
const r1 = tail(myTuple);
//    [2, 3, 4]
```

```
const r2 = tail([...myTuple, ...myArray] as const);
//    [2, 3, 4, ...string[]]
```

第二个改动是，剩余元素可以出现在元组中的任意位置上 - 不只是末尾位置！

```
type Strings = [string, string];
type Numbers = [number, number];
```

```
type StrStrNumNumBool = [...Strings, ...Numbers, boolean];
//    [string, string, number, number, boolean]
```

在以前，TypeScript 会像下面这样产生一个错误：

剩余元素必须出现在元组类型的末尾。

但是在 TypeScript 4.0 中放开了这个限制。

注意，如果展开一个长度未知的类型，那么后面的所有元素都将被纳入到剩余元素类型。

```
type Strings = [string, string];
type Numbers = number[];
```

```
type Unbounded = [...Strings, ...Numbers, boolean];
//    [string, string, ...(number | boolean)[]]
```

结合使用这两种行为，我们能够为 concat 函数编写一个良好的类型签名：

```
type Arr = readonly any[];
```

```
function concat<T extends Arr, U extends Arr>(arr1: T, arr2: U):
[...T, ...U] {
  return [...arr1, ...arr2];
}
```

虽然这个签名仍有点长，但是我们不再需要像重载那样重复多次，并且对于任何数组或元组它都能够给出期望的类型。

该功能本身已经足够好了，但是它的强大更体现在一些复杂的场景中。例如，考虑有一个支持 [部分参数应用](#) 的函数 partialCall。partialCall 接受一个函数（例如叫作 f），以及函数 f 需要的一些初始参数。它返回一个新的函数，该函数接受 f 需要的额外参数，并最终以前参数和额外参数来调用 f。

```
function partialCall(f, ...headArgs) {
  return (...tailArgs) => f(...headArgs, ...tailArgs);
}
```

```
const maxValue = 100;

for (let i = 0; i <= maxValue; i++) {
  // First get the squared value.
  let square = i ** 2;

  // Now print the squared value.
  console.log(square);
}
```

更多信息请参考 [PR](#)

17.5.3 快速修复：缺失的返回值表达式

有时候，我们可能忘记在函数的最后添加返回值语句，尤其是在将简单箭头函数转换成还有花括号的箭头函数时。

// before

```
let f1 = () => 42;
```

// oops - not the same!

```
let f2 = () => {
  42;
};
```

感谢开源社区的 [Wenlu Wang](#) 的 [PR](#)，TypeScript 提供了快速修复功能来添加 return 语句，删除花括号，或者为箭头函数体添加小括号用以区分对象字面量。

```
interface Person {
  name: string;
  age: number;
}

function sortByAge(people: Person[]) {
  people.sort((a, b) => a.age - b.age);
}
```

17.6 支持"Solution Style"的 tsconfig.json 文件

编译器需要知道一个文件被哪个配置文件所管理，因此才能够应用适当的配置选项并且计算出当前“工程”包含了哪些文件。在默认情况下，编辑器使用 TypeScript 语言服务来向上遍历父级目录以查找 tsconfig.json 文件。

有一种特殊情况是 tsconfig.json 文件仅用于引用其它 tsconfig.json 文件。

```
// tsconfig.json
{
  files: [],
  references: [
    { path: "./tsconfig.shared.json" },
    { path: "./tsconfig.frontend.json" },
    { path: "./tsconfig.backend.json" },
  ],
}
```

这个文件除了用来管理其它项目的配置文件之外什么也没做，在某些环境中它被叫作“solution”。这里，任何一个 tsconfig.*.json 文件都不会被 TypeScript 语言服务所选用，但是我们希望语言服务能够分析出当前的 .ts 文件被上述 tsconfig.json 中引用的哪个配置文件所管理。

TypeScript 3.9 为这种类型的配置方式添加了编辑器的支持。更多信息请参考 [PR](#)。

18 v4.0

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.0.html>

18.1 可变参数组类型

在 JavaScript 中有一个函数 concat，它接受两个数组或元组并将它们连接在一起构成一个新数组。

```
function concat(arr1, arr2) {
  return [...arr1, ...arr2];
}
```

再假设有一个 tail 函数，它接受一个数组或元组并返回除首个元素外的所有元素。

```
function tail(arg) {
  const [, ...result] = arg;
  return result;
}
```

那么，我们如何在 TypeScript 中为这两个函数添加类型？

在旧版本的 TypeScript 中，对于 concat 函数我们能做的是编写一些函数重载签名。

```
function concat(arr1: [], arr2: []): [];
function concat<A>(arr1: [A], arr2: []): [A];
function concat<A, B>(arr1: [A, B], arr2: []): [A, B];
function concat<A, B, C>(arr1: [A, B, C], arr2: []): [A, B, C];
function concat<A, B, C, D>(arr1: [A, B, C, D], arr2: []): [A, B, C, D];
function concat<A, B, C, D, E>(arr1: [A, B, C, D, E], arr2: []): [A, B, C, D, E];
function concat<A, B, C, D, E, F>(arr1: [A, B, C, D, E, F], arr2: []): [A, B, C, D, E, F];
```

在保持第二个数组为空的情况下，我们已经编写了七个重载签名。接下来，让我们为 arr2 添加一个参数。

```
function concat<A2>(arr1: [], arr2: [A2]): [A2];
function concat<A1, A2>(arr1: [A1], arr2: [A2]): [A1, A2];
function concat<A1, B1, A2>(arr1: [A1, B1], arr2: [A2]): [A1, B1, A2];
function concat<A1, B1, C1, A2>(arr1: [A1, B1, C1], arr2: [A2]): [A1, B1, C1, A2];
function concat<A1, B1, C1, D1, A2>(arr1: [A1, B1, C1, D1], arr2: [A2]): [A1, B1, C1, D1, A2];
function concat<A1, B1, C1, D1, E1, A2>(arr1: [A1, B1, C1, D1, E1], arr2: [A2]): [A1, B1, C1, D1, E1, A2];
function concat<A1, B1, C1, D1, E1, F1, A2>(arr1: [A1, B1, C1, D1, E1, F1], arr2: [A2]): [A1, B1, C1, D1, E1, F1, A2];
```

这已经开始变得不合理了。不巧的是，在给 tail 函数添加类型时也会遇到同样的问题。

在受尽了“重载的折磨”后，它依然没有完全解决我们的问题。它只能针对已编写的重载给出正确的类型。如果我们想要处理所有情况，则还需要提供一个如下的重载：

```
function concat<T, U>(arr1: T[], arr2: U[]): Array<T | U>;
```

但是这个重载签名没有反映出输入的长度，以及元组元素的顺序。

TypeScript 4.0 带来了两项基础改动，还伴随着类型推断的改善，因此我们能够方便地添加类型。

```
sideLength!: number;
//          ^^^^^^^^^
//          类型注解

constructor(sideLength: number) {
    this.initialize(sideLength);
}

initialize(sideLength: number) {
    this.sideLength = sideLength;
}

get area() {
    return this.sideLength ** 2;
}
}
```

更多详情请参考 [PR](#)。

18.4 [断路赋值运算符](#)

JavaScript 以及其它很多编程语言支持一些_复合赋值_运算符。 复合赋值运算符作用于两个操作数，并将运算结果赋值给左操作数。 你从前可能见到过以下代码：

```
// 加
// a = a + b
a += b;

// 减
// a = a - b
a -= b;

// 乘
// a = a * b
a *= b;

// 除
// a = a / b
a /= b;

// 幂
// a = a ** b
a **= b;
```

```
// 左移位
// a = a << b
a <<= b;
```

JavaScript 中的许多运算符都具有一个对应的赋值运算符！ 目前为止，有三个值得注意的例外：逻辑_与_（&&），逻辑_或_（||）和逻辑_空值合并_（??）。 这就是为什么 TypeScript 4.0 支持了一个 ECMAScript 的新特性，增加了三个新的赋值运算符&&=，||=和??=。 这三个运算符可以用于替换以下代码：

TypeScript 4.0 改进了剩余参数和剩余元组元素的类型推断，因此我们可以为这种使用场景添加类型。

```
type Arr = readonly unknown[];

function partialCall<T extends Arr, U extends Arr, R>(
    f: (...args: [...T, ...U]) => R,
    ...headArgs: T
) {
    return (...tailArgs: U) => f(...headArgs, ...tailArgs);
}
```

此例中，partialCall 知道能够接受哪些初始参数，并返回一个函数，它能够正确地选择接受或拒绝额外的参数。

```
// @errors: 2345 2554 2554 2345
type Arr = readonly unknown[];

function partialCall<T extends Arr, U extends Arr, R>(
    f: (...args: [...T, ...U]) => R,
    ...headArgs: T
) {
    return (...tailArgs: U) => f(...headArgs, ...tailArgs);
}
// ---cut---
const foo = (x: string, y: number, z: boolean) => {};
```

```
const f1 = partialCall(foo, 100);
//          ~~~
// Argument of type 'number' is not assignable to parameter of type 'string'.
```

```
const f2 = partialCall(foo, "hello", 100, true, "oops");
//          ~~~~~
// Expected 4 arguments, but got 5.(2554)
```

```
// This works!
const f3 = partialCall(foo, "hello");
//    (y: number, z: boolean) => void
```

```
// What can we do with f3 now?
```

```
// Works!
f3(123, true);
```

```
f3();
```

f3(123, "hello"); 可变参数组类型支持了许多新的激动人心的模式，尤其是函数组合。 我们期望能够通过它来为 JavaScript 内置的 bind 函数进行更好的类型检查。 还有一些其它的类型推断改进以及模式引入进来，如果你想了解更多，请参考 [PR](#)。

18.2 标签元组元素

改进元组类型和参数列表使用体验的重要性在于它允许我们为 JavaScript 中惯用的方法添加强类型验证 - 例如对参数列表进行切片而后传递给其它函数。 这里至关重要的一点是我们可以使用元组类型作为剩余参数类型。

例如，下面的函数使用元组类型作为剩余参数：

```
function foo(...args: [string, number]): void {
  // ...
}
```

它与下面的函数基本没有区别：

```
function foo(arg0: string, arg1: number): void {
  // ...
}
```

对于 foo 函数的任意调用者：

```
function foo(arg0: string, arg1: number): void {
  // ...
}
```

```
foo("hello", 42);
```

```
foo("hello", 42, true); // Expected 2 arguments, but got 3.
```

```
foo("hello"); // Expected 2 arguments, but got 1.
```

但是，如果从代码可读性的角度来看，就能够看出两者之间的差别。 在第一个例子中，参数的第一个元素和第二个元素都没有参数名。 虽然这不影响类型检查，但是元组中元素位置上缺乏标签令它们难以使用 - 很难表达出代码的意图。

这就是为什么 TypeScript 4.0 中的元组可以提供标签。

```
type Range = [start: number, end: number];
```

为了加强参数列表和元组类型之间的联系，剩余元素和可选元素的语法采用了参数列表的语法。

```
type Foo = [first: number, second?: string, ...rest: any[]];
```

在使用标签元组时有一些规则要遵守。 其一是，如果一个元组元素使用了标签，那么所有元组元素必须都使用标签。

```
type Bar = [first: string, number];
```

// Tuple members must all have names or all not have names.(5084)

元组标签名不影响解构变量名，它们不必相同。 元组标签仅用于文档和工具目的。

```
function foo(x: [first: string, second: number]) {
  // ...
}
```

// 注意：不需要命名为'first'和'second'

```
const [a, b] = x;
```

a

// string

b

// number

}

总的来说，标签元组对于元组和参数列表模式以及实现类型安全的重载时是很便利的。 实际上，在代码编辑器中 TypeScript 会尽可能地将它们显示为重载。

```
type Name =
  | [first: string, last: string]
  | [first: string, middle: string, last: string]

function createPerson(...name: Name) {
  // ...
}

createPerson()
```

更多详情请参考 [PT](#)。

18.3 从构造函数中推断类属性

在 TypeScript 4.0 中，当启用了 noImplicitAny 时，编译器能够根据基于控制流的分析来确定类中属性的类型

```
class Square {
  // 在旧版本中，以下两个属性均为 any 类型
  area; // number
  sideLength; // number
}
```

```
constructor(sideLength: number) {
  this.sideLength = sideLength;
  this.area = sideLength ** 2;
}
```

}

如果没有在构造函数中的所有代码执行路径上为实例成员进行赋值，那么该属性会被认为可能为 undefined 类型。

```
class Square {
  sideLength; // number | undefined

  constructor(sideLength: number) {
    if (Math.random()) {
      this.sideLength = sideLength;
    }
  }
}
```

```
get area() {
  return this.sideLength ** 2;
  // ~~~~~
  // 对象可能为'undefined'
}
```

}

如果你清楚地知道属性类型（例如，类中存在类似于 initialize 的初始化方法），你仍需要明确地使用类型注解来指定类型，以及需要使用确切赋值断言 (!) 如果你启用了 strictPropertyInitialization 模式。

```
class Square {
  // 确切赋值断言
  // v
}
```

18.9 编辑器改进

TypeScript 编译器不但支持在大部分编辑器中编写 TypeScript 代码，还支持着在 Visual Studio 系列的编辑器中编写 JavaScript 代码。因此，我们主要工作之一是改善编辑器支持 - 这也是程序员花费了大量时间的地方。

针对不同的编辑器，在使用 TypeScript/JavaScript 的新功能时可能会有所区别，但是

- Visual Studio Code 支持[选择不同的 TypeScript 版本](#)。或者，安装 [JavaScript/TypeScript Nightly Extension](#) 插件来使用最新的版本。
- Visual Studio 2017/2019 提供了 SDK 安装包，以及 [MSBuild 安装包](#)。
- Sublime Text 3 支持[选择不同的 TypeScript 版本](#)

[这里是支持 TypeScript 的编辑器列表](#)，到这里查看你喜爱的编译器是否支持最新版本的 TypeScript。

18.9.1 转换为可选链

可选链是一个较新的大家喜爱的特性。TypeScript 4.0 带来了一个新的重构工具来转换常见的代码模式，以利用[可选链](#)和[空值合并](#)！

```
function tryCallF(a) {
  return a && a.b.c && a.b.c.d.e.f();
}
```

注意，虽然该项重构不能完美地捕获真实情况（由于 JavaScript 中较复杂的真值/假值关系），但是我们坚信它能够适用于大多数使用场景，尤其是在 TypeScript 清楚地知道代码类型信息的时候。

更多详情请参考 [PR](#)。

18.9.2 /** @deprecated */支持

TypeScript 现在能够识别代码中的 `/** @deprecated */` JSDoc 注释，并对编辑器提供支持。该信息会显示在自动补全列表中以及建议诊断信息，编辑器可以特殊处理它。在类似于 VS Code 的编辑器中，废弃的值会显示为删除线，例如 `~~like this~~`。

```
a = a && b;
a = a || b;
a = a ?? b;
或者相似的 if 语句
// could be 'a ||= b'
if (!a) {
  a = b;
}
```

还有以下的惰性初始化值的例子：

```
let values: string[];
(values ?? (values = [])).push("hello");
```

// After

```
(values ??= []).push("hello");
```

少数情况下当你使用带有副作用的存取器时，值得注意的是这些运算符只在必要时才执行赋值操作。也就是说，不仅是运算符右操作数会“短路”，整个赋值操作也会“短路”

```
obj.prop ||= foo();
```

// roughly equivalent to either of the following

```
obj.prop || (obj.prop = foo());
```

```
if (!obj.prop) {
  obj.prop = foo();
}
```

[尝试运行这个例子](#)来查看与 `_始终_` 执行赋值间的差别。

```
const obj = {
  get prop() {
    console.log("getter has run");

    // Replace me!
    return Math.random() < 0.5;
  },
  set prop(_val: boolean) {
    console.log("setter has run");
  }
};
```

```
function foo() {
  console.log("right side evaluated");
  return true;
}
```

```
console.log("This one always runs the setter");
obj.prop = obj.prop || foo();
```

```
console.log("This one *sometimes* runs the setter");
obj.prop ||= foo();
```

非常感谢社区成员 [Wenlu Wang](#) 为该功能的付出！

v4.0 => 短路赋值运算符

18.5 catch 语句中的 unknown 类型

在 TypeScript 的早期版本中，catch 语句中的捕获变量总为 any 类型。这意味着你可以在捕获变量上执行任意的操作。

```
try {
  // Do some work
} catch (x) {
  // x 类型为 'any'
  console.log(x.message);
  console.log(x.toUpperCase());
  x++;
  x.yadda.yadda.yadda();
}
```

上述代码可能导致错误处理语句中产生了_更多_的错误，因此该行为是不合理的。 因为捕获变量默认为 any 类型，所以它不是类型安全的，你可以在上面执行非法操作。

TypeScript 4.0 允许将 catch 语句中的捕获变量类型声明为 unknown 类型。 unknown 类型比 any 类型更加安全，因为它要求在使用之前必须进行类型检查。

```
try {
  // ...
} catch (e: unknown) {
  // Can't access values on unknowns
  console.log(e.toUpperCase());

  if (typeof e === "string") {
    // We've narrowed 'e' down to the type 'string'.
    console.log(e.toUpperCase());
  }
}
```

由于 catch 语句捕获变量的类型不会被默认地改变成 unknown 类型，因此我们考虑在未来添加一个新的--strict 标记来有选择性地引入该行为。 目前，我们可以通过使用代码静态检查工具来强制 catch 捕获变量使用了明确的类型注解：any 或：unknown。

更多详情请参考 [PR](#)。

18.6 自定义 JSX 工厂

在使用 JSX 时，*fragment* 类型的 JSX 元素允许返回多个子元素。 当 TypeScript 刚开始实现 fragments 时，我们不太清楚其它代码库该如何使用它们。 最近越来越多的库开始使用 JSX 并支持与 fragments 结构相似的 API。

在 TypeScript 4.0 中，用户可以使用 jsxFragmentFactory 选项来自定义 fragment 工厂。

例如，下例的 tsconfig.json 文件告诉 TypeScript 使用与 React 兼容的方式来转换 JSX，但使用 h 来代替 React.createElement 工厂，同时使用 Fragment 来代替

```
React.Fragment。
{
  compilerOptions: {
    target: "esnext",
    module: "commonjs",
    jsx: "react",
    jsxFactory: "h",
    jsxFragmentFactory: "Fragment",
  }
}
```

```
ts_upgrade_from_2.2_to_5.3
  },
}
如果针对每个文件具有不同的 JSX 工厂，你可以使用新的/** @jsxFrag */编译指令注释。
示例：
// 注意：这些编译指令注释必须使用 JSDoc 风格，否则不起作用
```

```
/** @jsx h */
/** @jsxFrag Fragment */
```

```
import { h, Fragment } from "preact";
```

```
export const Header = (
  <>
    <h1>Welcome</h1>
  </>
);
```

```
上述代码会转换为如下的 JavaScript
// 注意：这些编译指令注释必须使用 JSDoc 风格，否则不起作用
```

```
/** @jsx h */
/** @jsxFrag Fragment */
```

```
import { h, Fragment } from "preact";
```

```
export const Header = (
  h(
    Fragment,
    null,
    h("h1", null, "Welcome")
  )
);
```

非常感谢社区成员 [Noj Vek](#) 为该特性的付出。

更多详情请参考 [PR](#)

18.7 对启用了--noEmitOnError 的`build 模式进行速度优化

在以前，当启用了--noEmitOnError 编译选项时，如果在--incremental 构建模式下的前一次构建出错了，那么接下来的构建会很慢。 这是因为当启用了--noEmitOnError 时，前一次失败构建的信息不会被缓存到.tsbuildinfo 文件中。

TypeScript 4.0 对此做出了一些改变，极大地提升了这种情况下的编译速度，改善了应用--build 模式的场景（包含--incremental 和--noEmitOnError）。

更多详情请参考 [PR](#)。

18.8 --incremental 和--noEmit

TypeScript 4.0 允许同时使用--incremental 和--noEmit。 这在之前是不允许的，因为--incremental 需要生成.tsbuildinfo 文件； 然而，提供更快地增量构建对所有用户来讲都是十分重要的。

更多详情请参考 [PR](#)。

ts upgrade from 2.2 to 5.3

"top", "middle"和"bottom", 以及水平对齐的选项"left", "center"和"right"之间, 共有 9 种可能的字符串, 前者选项之一与后者选项之一之间使用短横线连接。

```
type VerticalAlignment = 'top' | 'middle' | 'bottom';
type HorizontalAlignment = 'left' | 'center' | 'right';
```

```
// Takes
//   | "top-left"    | "top-center"   | "top-right"
//   | "middle-left" | "middle-center" | "middle-right"
//   | "bottom-left" | "bottom-center" | "bottom-right"
```

```
declare function setAlignment(
  value: `${VerticalAlignment}-${HorizontalAlignment}`
): void;
```

```
setAlignment('top-left'); // works!
setAlignment('top-middel'); // error!
setAlignment('top-pot'); // error! but good doughnuts if you're ever
in Seattle
```

这样的例子还有很多, 但它仍只是小例子而已, 因为我们可以直接写出所有可能的值。实际上, 对于 9 个字符串来讲还算可以; 但是如果需要大量的字符串, 你就得考虑如何去自动生成 (或者简单地使用 string)。

有些值实际上是来自于动态创建的字符串字面量。例如, 假设 makeWatchedObject API 接收一个对象, 并生成一个几乎等价的对象, 但是带有一个新的 on 方法来检测属性的变化。

```
let person = makeWatchedObject({
  firstName: 'Homer',
  age: 42,
  location: 'Springfield',
});
```

```
person.on('firstNameChanged', () => {
  console.log(`firstName was changed!`);
});
```

注意, on 监听的是"firstNameChanged"事件, 而非仅仅是"firstName"。那么我们如何定义类型?

```
type PropEventSource<T> = {
  on(eventName: `${string & keyof T}Changed`, callback: () => void):
void;
};
```

```
/// Create a "watched object" with an 'on' method
/// so that you can watch for changes to properties.
declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;
这样做的话, 如果传入了错误的属性会产生一个错误!
```

```
type PropEventSource<T> = {
  on(eventName: `${string & keyof T}Changed`, callback: () => void):
void;
};
declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;
let person = makeWatchedObject({
```

112

v4.1 => 模版字面量类型

```
let obj = {
  /** @deprecated */
  someOperation() {
  }
}
obj.someOperation();

obj.|
  someOperation (method) someOperatio...

/** @deprecated */
class OldThing {
}

new OldThing();
```

感谢 [Wenlu Wang](#) 为该特性的付出。更多详情请参考 [PR](#)。

18.9.3 启动时的局部语义模式

我们从用户反馈得知在启动一个大的工程时需要很长的时间。罪魁祸首是一个叫作_程序构造_的处理过程。该处理是从一系列根文件开始解析并查找它们的依赖, 然后再解析依赖, 然后再解析依赖的依赖, 以此类推。你的工程越大, 你等待的时间就越长, 在这之前你不能使用编辑器的诸如“跳转到定义”等功能。

这就是为什么我们要提供一个新的编辑器模式, 在语言服务被完全加载之前提供局部编辑体验。这里的主要想法是, 编辑器可以运行一个轻量级的局部语言服务, 它只关注编辑器当前打开的文件。

很难准确地形容能够获得多大的提升, 但听说在 Visual Studio Code 项目中, 以前需要等待_20 秒到 1 分钟_的时间来完全加载语言服务。做为对比, 新的局部语义模式看起来能够将上述时间减少到几秒钟。示例, 从下面的视频中, 你可以看到左侧的 TypeScript 3.9 与右侧的 TypeScript 4.0 的对比。

当在编辑器中打开一个大型的代码仓库时, TypeScript 3.9 根本无法提供代码补全以及信息提示。反过来, 安装了 TypeScript 4.0 的编辑器能够在当前文件上_立即_提供丰富的编辑体验, 尽管后台仍然在加载整个工程。

目前, 唯一支持该模块的编辑器是 [Visual Studio Code](#), 并且在 [Visual Studio Code Insiders](#) 版本中还带来了一些体验上的优化。我们发现该特性在用户体验和功能性上仍有优化空间, 我们总结了一个[优化列表](#)。我们也期待你的使用反馈。

更多详情请参考[原始的提议](#), [功能实现的 PR](#), 以及[后续的跟踪帖](#)。

18.9.4 更智能的自动导入

自动导入是个特别好的功能, 它让编码更加容易; 然而, 每一次自动导入不好用的时候, 它就会导致一部分用户流失。一个特殊的问题是, 自动导入对于使用 TypeScript 编写的依赖不好用 - 也就是说, 用户必须在工程中的某处明确地编写一个导入语句。

那么为什么自动导入在@types 包上是好用的, 但是对于自己编写的代码却不好用? 这表明自动导入功能只适用于工程中已经引入的包。因为 TypeScript 会自动地将

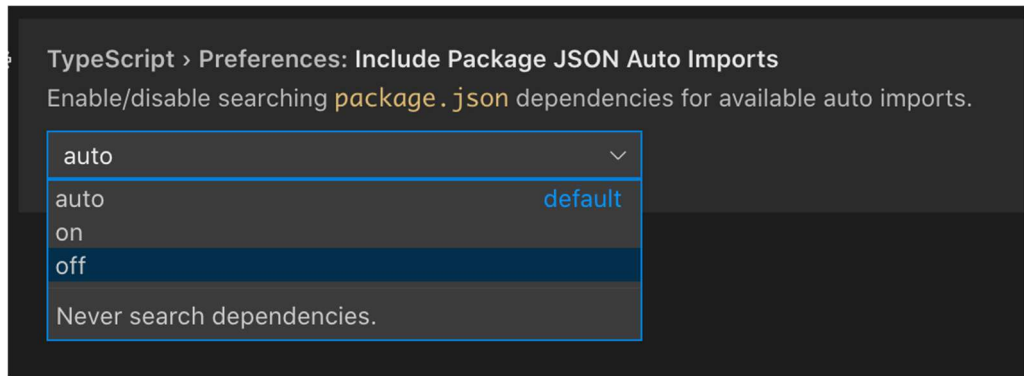
v4.0 => 编辑器改进

109

node_modules/@types 下面的包引入进工程，_那些_包才会被自动导入。 另一方面，其它的包会被排除，因为遍历 node_modules 下所有的包_相当_费时。

这就导致了在自动导入一个刚刚安装完但还没有开始使用的包时具有相当差的体验。

TypeScript 4.0 对编辑器环境进行了一点小改动，它会自动引入你的工程下的 package.json 文件中 dependencies（和 peerDependencies）字段里列出的包。 这些引入的包只用于改进自动导入功能，它们对类型检查等其它功能没有任何影响。 这使得自动导入功能对于项目中所有带有类型的依赖项都是可用的，同时不必遍历 node_modules。 少数情况下，若在 package.json 中列出了多于 10 个未导入的带有类型的依赖，那么该功能会被自动禁用以避免过慢的工程加载。 若想要强制启用该功能，或完全禁用该功能，则需要配置你的编辑器。 针对 Visual Studio Code，对应到“Include Package JSON Auto Imports”配置（或者 typescript.preferences.includePackageJsonAutoImports 配置）。



For more details, you can see the [proposal issue](#) along with [the implementing pull request](#).

18.10 我们的新网站

最近，我们重写了 [TypeScript 官网](#) 并且已经发布！

[我们在这里介绍了关于新网站的一些信息](#)；但仍期望用户给予更多的反馈！ 如果你有问题或建议，请到[这里提交 Issue](#)。

19 v4.1

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.1.html>

19.1 模版字符串类型

使用字符串字面量类型能够表示仅接受特定字符串参数的函数和 API。

```
function setVerticalAlignment(location: 'top' | 'middle' | 'bottom') {
    // ...
}
```

```
setVerticalAlignment('middel');
//                ^^^^^^^^^
// Argument of type '"middel"' is not assignable to parameter of type
// '"top" | "middle" | "bottom"'.
```

使用字符串字面量类型的好处是它能够对字符串进行拼写检查。

此外，字符串字面量还能用于映射类型中的属性名。 从这个意义上讲，它们可被当作构件使用。

```
type Options = {
    [K in
        | 'noImplicitAny'
        | 'strictNullChecks'
        | 'strictFunctionTypes']?: boolean;
};
```

```
// same as
// type Options = {
//     noImplicitAny?: boolean,
//     strictNullChecks?: boolean,
//     strictFunctionTypes?: boolean
// };
```

还有一处字符串字面量类型可被当作构件使用，那就是在构造其它字符串字面量类型时。

这也是 TypeScript 4.1 支持模版字符串类型的原因。 它的语法与 [JavaScript 中的模版字符串](#) 的语法是一致的，但是是用在表示类型的位置上。 当将其与具体类型结合使用时，它会将字符串拼接并产生一个新的字符串字面量类型。

```
type World = 'world';
```

```
type Greeting = `hello ${World}`;
//                ^^^^^^^^^
// "hello world"
```

如果在替换的位置上使用了联合类型会怎么样呢？ 它将生成由各个联合类型成员所表示的字符串字面量类型的联合。

```
type Color = 'red' | 'blue';
type Quantity = 'one' | 'two';
```

```
type SeussFish = `${Quantity | Color} fish`;
//                ^^^^^^^^^
// "one fish" | "two fish" | "red fish" | "blue fish"
```

除此之外，我们也可以在其它场景中应用它。 例如，有些 UI 组件库提供了指定垂直和水平对齐的 API，通常会使用类似于"bottom-right"的字符串来同时指定。 在垂直对齐的选项

```
// All of these return the type 'number[]':
deepFlatten([1, 2, 3]);
deepFlatten([[1], [2, 3]]);
deepFlatten([[1], [[2]], [[[3]]]]);
类似地, 在 TypeScript 4.1 中我们可以定义 Awaited 类型来拆解 Promise。
type Awaited<T> = T extends PromiseLike<infer U> ? Awaited<U> : T;
```

```
/// 类似于 `promise.then(...)`，但是类型更准确
declare function customThen<T, U>(
  p: Promise<T>,
  onFulfilled: (value: Awaited<T>) => U
): Promise<Awaited<U>>;
```

一定要注意, 虽然这些递归类型很强大, 但要有节制地使用它。

首先, 这些类型能做的更多, 但也会增加类型检查的耗时。 尝试为考拉兹猜想或斐波那契数列建模是一件有趣的事儿, 但请不要在 npm 上发布带有它们的.d.ts文件。

除了计算量大之外, 这些类型还可能会达到内置的递归深度限制。 如果到达了递归深度限制, 则会产生编译错误。 通常来讲, 最好不要去定义这样的类型。

更多详情, 请参考 [PR](#)。

19.4 索引访问类型检查 (--noUncheckedIndexedAccess)

TypeScript 支持一个叫做索引签名的功能。 索引签名用于告诉类型系统, 用户可以访问任意名称的属性。

```
interface Options {
  path: string;
  permissions: number;

  // 额外的属性可以被这个签名捕获
  [propName: string]: string | number;
}
```

```
function checkOptions(opts: Options) {
  opts.path; // string
  opts.permissions; // number

  // 以下都是允许的
  // 它们的类型为 'string | number'
  opts.yadda.toString();
  opts['foo bar baz'].toString();
  opts[Math.random()].toString();
}
```

上例中, Options 包含了索引签名, 它表示在访问未直接列出的属性时得到的类型为 string | number。 这是一种乐观的做法, 它假想我们非常清楚代码在做什么, 但实际上 JavaScript 中的大部分值并不支持任意的属性名。 例如, 大多数类型并不包含属性名为 Math.random()的值。 对许多用户来讲, 这不是期望的行为, 就好像没有利用到--strictNullChecks提供的严格类型检查。

这就是 TypeScript 4.1 提供了--noUncheckedIndexedAccess编译选项的原因。 在该新模式下, 任何属性访问 (例如 foo.bar) 或者索引访问 (例如 foo["bar"]) 都会被认为可能为 undefined。 例如在上例中, opts.yadda 的类型为 string | number |

```
firstName: 'Homer',
age: 42,
location: 'Springfield',
});

// error!
person.on('firstName', () => {});
```

```
// error!
person.on('frstNameChanged', () => {});
我们还可以在模版字面上做一些其它的事情: 可以从替换的位置来推断类型。 我们将上面的例子改写成泛型, 由 eventName 字符串来推断关联的属性名。
```

```
type PropEventSource<T> = {
  on<K extends string & keyof T>(
    eventName: `${K}Changed`,
    callback: (newValue: T[K]) => void
  ): void;
};
```

```
declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;
```

```
let person = makeWatchedObject({
  firstName: 'Homer',
  age: 42,
  location: 'Springfield',
});

// works! 'newName' is typed as 'string'
person.on('firstNameChanged', (newName) => {
  // 'newName' has the type of 'firstName'
  console.log(`new name is ${newName.toUpperCase()}`);
});
```

```
// works! 'newAge' is typed as 'number'
person.on('ageChanged', (newAge) => {
  if (newAge < 0) {
    console.log('warning! negative age');
  }
});
```

这里我们将 on 定义为泛型方法。 当用户使用"firstNameChanged"来调用该方法, TypeScript 会尝试去推断出 K 所表示的类型。 为此, 它尝试将 K 与"Changed"之前的内容进行匹配并推断出"firstName"。 一旦 TypeScript 得到了结果, on 方法就能够从原对象上获取 firstName 的类型, 此例中是 string。 类似地, 当使用"ageChanged"调用时, 它会找到属性 age 的类型为 number。

类型推断可以用不同的方式组合, 常见的是解构字符串, 再使用其它方式重新构造它们。 实际上, 为了便于修改字符串字面量类型, 我们引入了一些新的工具类型来修改字符大小写。

```
type EnthusiasticGreeting<T extends string> = `${Uppercase<T>}`;
```

```
type HELLO = EnthusiasticGreeting<'hello'>;
```

v4.1 => 模版字面量类型


```
//      ^^^^^
//  "HELLO"

新的类型别名为 Uppercase, Lowercase, Capitalize 和 Uncapitalize。 前两个会转换字符串中的所有字符，而后面两个只转换字符串的首字母。
更多详情, 查看原 PR 以及正在进行的切换类型别名助手的 PR。
```

19.2 [在映射类型中更改映射的键](#)

让我们先回顾一下，映射类型可以使用任意的键来创建新的对象类型。

```
type Options = {
    [K in
        | 'noImplicitAny'
        | 'strictNullChecks'
        | 'strictFunctionTypes']?: boolean;
};

// same as
// type Options = {
//     noImplicitAny?: boolean,
//     strictNullChecks?: boolean,
//     strictFunctionTypes?: boolean
// };

或者，基于任意的对象类型来创建新的对象类型。
///

```
 'Partial<T>' 等同于 'T'，只是把每个属性标记为可选的。
type Partial<T> = {
 [K in keyof T]?: T[K];
};

到目前为止，映射类型只能使用提供给它的键来创建新的对象类型；然而，很多时候我们想要创建新的键，或者过滤掉某些键。
这就是 TypeScript 4.1 允许更改映射类型中的键的原因。它使用了新的 as 语句。
type MappedTypeWithNewKeys<T> = {
 [K in keyof T as NewKeyType]: T[K];
 // ^^^^^^^^^^^^^^^^^
 // 这里是新的语法！
};

通过 as 语句，你可以利用例如模版字面量类型，并基于原属性名来轻松地创建新属性名。
type Getters<T> = {
 [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K];
};
```


```

```
interface Person {
    name: string;
    age: number;
    location: string;
}

type LazyPerson = Getters<Person>;
// type LazyPerson = {
//     getName: () => string;
//     getAge: () => number;
//     getLocation: () => string;
// }
```

```
// }

此外，你可以巧用 never 类型来过滤掉某些键。 也就是说，在某些情况下你不必使用 Omit 工具类型。
// 删除 'kind' 属性
type RemoveKindField<T> = {
    [K in keyof T as Exclude<K, 'kind'>]: T[K];
};

interface Circle {
    kind: 'circle';
    radius: number;
}

type KindlessCircle = RemoveKindField<Circle>;

type RemoveKindField<T> = {
    [K in keyof T as Exclude<K, 'kind'>]: T[K];
};

interface Circle {
    kind: 'circle';
    radius: number;
}

type KindlessCircle = RemoveKindField<Circle>;
// type KindlessCircle = {
//     radius: number;
// }

更多详情，请参考 PR。
```

19.3 [递归的有条件类型](#)

在 JavaScript 中较为常见的是，一个函数能够以任意的层级来展平（flatten）并构建容器类型。 例如，可以拿 Promise 实例对象上的 .then() 方法来举例。 .then(...) 方法能够拆解每一个 Promise，直到它找到一个非 Promise 的值，然后将该值传递给回调函数。 Array 上也存在一个相对较新的 flat 方法，它接收一个表示深度的参数，并以此来决定展平操作的层数。 在过去，我们无法使用 TypeScript 类型系统来表达上述例子。 虽然也存在一些 hack，但基本上都不切合实际。 TypeScript 4.1 取消了对有条件类型的一些限制 - 因此它现在可以表达上述类型。 在 TypeScript 4.1 中，允许在有条件类型的分支中立即引用该有条件类型自身，这就使得编写递归的类型别名变得更加容易。 例如，我们想定义一个类型来获取嵌套数组中的元素类型，可以定义如下的 deepFlatten 类型。
type ElementType<T> = T extends ReadonlyArray<infer U> ?
ElementType<U> : T;

```
function deepFlatten<T extends readonly unknown[]>(x: T):
ElementType<T>[] {
    throw 'not implemented';
}
```



```
ts_upgrade_from_2.2_to_5.3
//      ^^^^^^^^
doSomethingAsync((value) => {
    doSomething();
    resolve(value);
    //      ^^^^^
});
});
```

然而，有时 `resolve()` 确实需要不带参数来调用 在这种情况下，我们可以给 `Promise` 传入明确的 `void` 泛型类型参数（例如，`Promise<void>`）。它利用了 `TypeScript 4.1` 中的一个新功能，一个潜在的 `void` 类型的末尾参数会变成可选参数。

```
new Promise<void>((resolve) => {
    //      ^^^^^^^
    doSomethingAsync(() => {
        doSomething();
        resolve();
    });
});
```

`TypeScript 4.1` 提供了快速修复选项来解决该问题。

19.9.5 有条件展开会创建可选属性

在 `JavaScript` 中，对象展开（例如，`{ ...foo }`）不会操作假值。因此，在 `{ ...foo }` 代码中，如果 `foo` 的值为 `null` 或 `undefined`，则它会被略过。

很多人利用该性质来可选地展开属性。

```
interface Person {
    name: string;
    age: number;
    location: string;
}
```

```
interface Animal {
    name: string;
    owner: Person;
}
```

```
function copyOwner(pet?: Animal) {
    return {
        ...(pet && pet.owner),
        otherStuff: 123,
    };
}
```

// We could also use optional chaining here:

```
function copyOwner(pet?: Animal) {
    return {
        ...pet?.owner,
        otherStuff: 123,
    };
}
```

`undefined`，而不是 `string | number`。如果需要访问那个属性，你可以先检查属性是否存在或者使用非空断言运算符（`!` 后缀字符）。

```
// @noUncheckedIndexedAccess
interface Options {
    path: string;
    permissions: number;

    // 额外的属性可以被这个签名捕获
    [propName: string]: string | number;
}
// ---cut---
```

```
function checkOptions(opts: Options) {
    opts.path; // string
    opts.permissions; // number
```

```
// 在 noUncheckedIndexedAccess 下，以下操作不允许
opts.yadda.toString();
opts['foo bar baz'].toString();
opts[Math.random()].toString();
```

```
// 首先检查是否存在
if (opts.yadda) {
    console.log(opts.yadda.toString());
}
```

```
// 使用 ! 非空断言，“我知道在做什么”
opts.yadda!.toString();
}
```

使用 `--noUncheckedIndexedAccess` 的一个结果是，通过索引访问数组元素时也会进行严格类型检查，就算是在遍历检查过边界的数组时。

```
// @noUncheckedIndexedAccess
function screamLines(strs: string[]) {
    // 下面会有问题
    for (let i = 0; i < strs.length; i++) {
        console.log(strs[i].toUpperCase());
    }
}
```

如果你不需要使用索引，那么可以使用 `for-of` 循环或 `forEach` 来遍历。

```
// @noUncheckedIndexedAccess
function screamLines(strs: string[]) {
    // 可以正常工作
    for (const str of strs) {
        console.log(str.toUpperCase());
    }
}
```

```
// 可以正常工作
strs.forEach((str) => {
    console.log(str.toUpperCase());
});
```

```
});
}
```

这个选项虽可以用来捕获访问越界的错误，但对大多数代码来讲有些烦，因此它不会被`--strict`选项自动启用；然而，如果你对此选项感兴趣，可以尝试一下，看它是否适用于你的代码。

更多详情，请参考 [PR](#)。

19.5 [不带 baseUrl 的 paths](#)

路径映射的使用很常见 - 通常它用于优化导入语句，以及模拟在单一代码仓库中进行链接的行为。

不幸的是，在使用 `paths` 时必须指定 `baseUrl`，它允许裸路径描述符基于 `baseUrl` 进行解析。 它会导致在自动导入时会使用较差的路径。

在 TypeScript 4.1 中，`paths` 不必与 `baseUrl` 一起使用。 它会一定程度上帮助解决上述的问题。

19.6 [checkJs 默认启用 allowJs](#)

从前，如果你想要对 JavaScript 工程执行类型检查，你需要同时启用 `allowJs` 和 `checkJs`。 这样的体验让人讨厌，因此现在 `checkJs` 会默认启用 `allowJs`。

更多详情，请参考 [PR](#)。

19.7 [React 17 JSX 工厂](#)

TypeScript 4.1 通过以下两个编译选项来支持 React 17 中的 `jsx` 和 `jsx`s 工厂函数：

- `react-jsx`
- `react-jsxdev`

这两个编译选项分别用于生产环境和开发环境中。 通常，编译选项之间可以继承。 例如，用于生产环境的 `tsconfig.json` 如下：

```
// ./src/tsconfig.json
{
  "compilerOptions": {
    "module": "esnext",
    "target": "es2015",
    "jsx": "react-jsx",
    "strict": true
  },
  "include": ["./**/*"]
}
```

另外一个用于开发环境的 `tsconfig.json` 如下：

```
// ./src/tsconfig.dev.json
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "jsx": "react-jsxdev"
  }
}
```

更多详情，请参考 [PR](#)。

19.8 [在编辑器中支持 JSDoc @see 标签](#)

编辑器对 TypeScript 和 JavaScript 代码中的 JSDoc 标签 `@see` 有了更好的支持。 它允许你使用像“跳转到定义”这样的功能。 例如，在下例中的 JSDoc 里可以使用跳转到定义到 `first` 或 `C`。

```
// @filename: first.ts
```

```
export class C {}

// @filename: main.ts
import * as first from './first';

/**
 * @see first.C
 */
function related() {}
感谢贡献者 Wenlu Wang 实现了这个功能!
```

19.9 [破坏性改动](#)

19.9.1 [lib.d.ts 更新](#)

`lib.d.ts` 包含一些 API 变动，在某种程度上是因为 DOM 类型是自动生成的。 一个具体的变动是 `Reflect.enumerate` 被删除了，因为它在 ES2016 中被删除了。

19.9.2 [abstract 成员不能被标记为 async](#)

`abstract` 成员不再可以被标记为 `async`。 这可以通过删除 `async` 关键字来修复，因为调用者只关注返回值类型。

19.9.3 [any/unknown Are Propagated in Falsy Positions](#)

从前，对于表达式 `foo && somethingElse`，若 `foo` 的类型为 `any` 或 `unknown`，那么整个表达式的类型为 `somethingElse`。

例如，在以前此处的 `x` 的类型为 `{ someProp: string }`。

```
declare let foo: unknown;
declare let somethingElse: { someProp: string };
```

```
let x = foo && somethingElse;
```

然而，在 TypeScript 4.1 中，会更谨慎地确定该类型。 由于不清楚 `&&` 左侧的类型，我们会传递 `any` 和 `unknown` 类型，而不是 `&&` 右侧的类型。

常见的模式是检查与 `boolean` 的兼容性，尤其是在谓词函数中。

```
function isThing(x: any): boolean {
  return x && typeof x === 'object' && x.blah === 'foo';
}
```

一种合适的修改是使用 `!!foo && someExpression` 来代替 `foo && someExpression`。

19.9.4 [Promise 的 resolve 的参数不再是可选的](#)

在编写如下的代码时

```
new Promise((resolve) => {
  doSomethingAsync(() => {
    doSomething();
    resolve();
  });
});
```

你可能会得到如下的错误：

```
resolve()
~~~~~
```

`error TS2554: Expected 1 arguments, but got 0.`

`An argument for 'value' was not provided.`

这是因为 `resolve` 不再有可选参数，因此默认情况下，必须给它传值。 它通常能够捕获 `Promise` 的 bug。 典型的修复方法是传入正确的参数，以及添加明确的类型参数。

```
new Promise<number>((resolve) => {
  // ...
})
```

v4.1 => 破坏性改动

```
let StringsAndMaybeBoolean: [...string[], boolean?];
// ~~~~~ 错误
这些不在结尾的剩余元素能够用来描述，可接收任意数量的前导参数加上固定数量的结尾参数的函数。
declare function doStuff(
  ...args: [...names: string[], shouldCapitalize: boolean]
): void;
```

```
doStuff(/*shouldCapitalize:*/ false);
doStuff('fee', 'fi', 'fo', 'fum', /*shouldCapitalize:*/ true);
尽管 JavaScript 中没有声明前导剩余参数的语法，但我们仍可以将 doStuff 函数的参数声明为带有前导剩余元素 ...args 的元组类型。使用这种方式可以帮助我们描述许多的 JavaScript 代码！
更多详情，请参考 PR。
```

20.3 更严格的 in 运算符检查

在 JavaScript 中，如果 in 运算符的右操作数是非对象类型，那么会产生运行时错误。TypeScript 4.2 确保了该错误能够在编译时被捕获。

```
'foo' in 42;
// The right-hand side of an 'in' expression must not be a primitive.
这个检查在大多数情况下是相当保守的，如果你看到提示了这个错误，那么代码中很可能真的有问题。
非常感谢外部贡献者 Jonas Hübötter 的 PR！
```

20.4 --noPropertyAccessFromIndexSignature

在 TypeScript 刚开始支持索引签名时，它只允许使用方括号语法来访问索引签名中定义的元素，例如 person["name"]。

```
interface SomeType {
  /** 这是索引签名 */
  [propName: string]: any;
}

function doStuff(value: SomeType) {
  let x = value['someProperty'];
}

这就导致了在处理带有任意属性的对象时变得烦锁。例如，假设有一个容易出现拼写错误的 API，容易出现在属性名的末尾位置多写一个字母 s 的错误。

interface Options {
  /** 要排除的文件模式。 */
  exclude?: string[];

  /**
   * 这会将其余所有未声明的属性定义为 'any' 类型。
   */
  [x: string]: any;
}

function processOptions(opts: Options) {
  // 注意，我们想要访问 `excludes` 而不是 `exclude`
  if (opts.excludes) {
```

此处，如果 pet 定义了，那么 pet.owner 的属性会被展开 - 否则，不会有属性被展开到目标对象中。

在之前，copyOwner 的返回值类型为基于每个展开运算结果的联合类型：The return type of copyOwner was previously a union type based on each spread:

```
{ x: number } | { x: number, name: string, age: number, location: string }
```

它精确地展示了操作是如何进行的：如果 pet 定义了，那么 Person 中的所有属性都存在；否则，在结果中不存在 Person 中的任何属性。它是一种要么全有要么全无的操作。

然而，我们发现这个模式被过度地使用了，在单一对象中存在数以百计的展开运算，每一个展开操作可能会添加成百上千的操作。结果就是这项操作可能非常耗时，并且用处不大。

在 TypeScript 4.1 中，返回值类型有时会使用全部的可选类型。

```
{
  x: number;
  name?: string;
  age?: number;
  location?: string;
}
```

这样的结果是有更好的性能以及更佳地展示。

更多详情，请参考 PR。目前，该行为还不完全一致，我们期待在未来会有所改进。

19.9.6 Unmatched parameters are no longer related

从前 TypeScript 在关联参数时，如果参数之间没有联系，则会将其关联为 any 类型。由于 TypeScript 4.1 的改动，TypeScript 会完全跳过这个过程。这意味着一些可赋值性检查会失败，同时也意味着重载解析可能会失败。例如，在解析 Node.js 中 util.promisify 函数的重载时可能会选择不同的重载签名，这可能会导致产生新的错误。

做为一个变通方法，你可能需要使用类型断言来消除错误。

20 v4.2

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.2.html>

20.1 [更智能地保留类型别名](#)

在 TypeScript 中，使用类型别名能够给某个类型起个新名字。倘若你定义了一些函数，并且它们全都使用了 string | number | boolean 类型，那么你就可以定义一个类型别名来避免重复。

```
type BasicPrimitive = number | string | boolean;
TypeScript 使用了一系列规则来推测是否该在显示类型时使用类型别名。 例如，有如下的代码。
```

```
export type BasicPrimitive = number | string | boolean;
```

```
export function doStuff(value: BasicPrimitive) {
    let x = value;
    return x;
}
```

如果在 Visual Studio, Visual Studio Code 或者 [TypeScript 演练场](#) 编辑器中把鼠标光标放在 x 上，我们就会看到信息面板中显示出了 BasicPrimitive 类型。同样地，如果我们查看由该文件生成的声明文件（.d.ts），那么 TypeScript 会显示出 doStuff 的返回值类型为 BasicPrimitive 类型。

那么你猜一猜，如果返回值类型为 BasicPrimitive 或 undefined 时会发生什么？

```
export type BasicPrimitive = number | string | boolean;
```

```
export function doStuff(value: BasicPrimitive) {
    if (Math.random() < 0.5) {
        return undefined;
    }
}
```

```
    return value;
}
```

可以在 [TypeScript 4.1 演练场](#) 中查看结果。虽然我们希望 TypeScript 将 doStuff 的返回值类型显示为 BasicPrimitive | undefined，但是它却显示成了 string | number | boolean | undefined 类型！这是怎么回事？

这与 TypeScript 内部的类型表示方式有关。当基于一个联合类型来创建另一个联合类型时，TypeScript 会将类型标准化，也就是把类型展开为一个新的联合类型 - 但这么做也可能丢失信息。类型检查器不得不根据 string | number | boolean | undefined 类型来尝试每一种可能的组合并查看使用了哪些类型别名，即便这样也可能会有多个类型别名指向 string | number | boolean 类型。

TypeScript 4.2 的内部实现更加智能了。我们会记录类型是如何被构造的，会记录它们原本的编写方式和之后的构造方式。我们同样会记录和区分不同的类型别名！

有能力根据类型使用的方式来回显这个类型就意味着，对于 TypeScript 用户来讲能够避免显示很长的类型；同时也意味着会生成更友好的 .d.ts 声明文件、错误消息和编辑器内显示的类型及签名帮助信息。这会让 TypeScript 对于初学者来讲更友好一些。

更多详情，请参考 [PR: 改进保留类型别名的联合](#)，以及 [PR: 保留间接的类型别名](#)。

20.2 [元组类型中前导的/中间的剩余元素](#)

在 TypeScript 中，元组类型用于表示固定长度和元素类型的数组。

```
// 存储了一对数字的元组
```

```
let a: [number, number] = [1, 2];
```

```
// 存储了一个 string，一个 number 和一个 boolean 的元组
```

```
let b: [string, number, boolean] = ['hello', 42, true];
```

随着时间的推移，TypeScript 中的元组类型变得越来越复杂，因为它们也被用来表示像 JavaScript 中的参数列表类型。结果就是，它可能包含可选元素和剩余元素，以及用于工具和提高可读性的标签。

```
// 包含一个或两个元素的元组。
```

```
let c: [string, string?] = ['hello'];
```

```
c = ['hello', 'world'];
```

```
// 包含一个或两个元素的标签元组。
```

```
let d: [first: string, second?: string] = ['hello'];
```

```
d = ['hello', 'world'];
```

```
// 包含剩余元素的元组 - 至少前两个元素是字符串，
```

```
// 以及后面的任意数量的布尔元素。
```

```
let e: [string, string, ...boolean[]];
```

```
e = ['hello', 'world'];
```

```
e = ['hello', 'world', false];
```

```
e = ['hello', 'world', true, false, true];
```

在 TypeScript 4.2 中，剩余元素会按它们的使用方式进行展开。在之前的版本中，TypeScript 只允许 ...rest 元素位于元组的末尾。

但现在，剩余元素可以出现在元组中的任意位置 - 但有一点限制。

```
let foo: [...string[], number];
```

```
foo = [123];
```

```
foo = ['hello', 123];
```

```
foo = ['hello!', 'hello!', 'hello!', 123];
```

```
let bar: [boolean, ...string[], boolean];
```

```
bar = [true, false];
```

```
bar = [true, 'some text', false];
```

```
bar = [true, 'some', 'separated', 'text', false];
```

唯一的限制是，剩余元素之后不能出现可选元素或其它剩余元素。换句话说，一个元组中只允许有一个剩余元素，并且剩余元素之后不能有可选元素。

```
interface Clown {
    /*...*/
}
```

```
interface Joker {
    /*...*/
}
```

```
let StealersWheel: [...Clown[], 'me', ...Joker[]];
```

```
// ~~~~~ 错误
```


在使用了该选项时，TypeScript 编译器会输出非常详细的信息来说明某个文件被包含进工程的原因。 为了更易理解，我们可以把输出结果存到文件里，或者通过管道使用其它命令来查看它。

```
# 将输出保存到文件
tsc --explainFiles > explanation.txt
```

```
# 将输出传递给工具程序 `less`，或编辑器 VS Code
tsc --explainFiles | less
```

```
tsc --explainFiles | code -
通常，输出结果首先会给出列出包含 lib.d.ts 文件的原因，然后是本地文件，再然后是 node_modules 文件。
```

```
TS_Compiler_Directory/4.2.2/lib/lib.es5.d.ts
  Library referenced via 'es5' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2015.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2015.d.ts
  Library referenced via 'es2015' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2016.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2016.d.ts
  Library referenced via 'es2016' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2017.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2017.d.ts
  Library referenced via 'es2017' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2018.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2018.d.ts
  Library referenced via 'es2018' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2019.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2019.d.ts
  Library referenced via 'es2019' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2020.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2020.d.ts
  Library referenced via 'es2020' from file
'TS_Compiler_Directory/4.2.2/lib/lib.esnext.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.esnext.d.ts
  Library 'lib.esnext.d.ts' specified in compilerOptions
```

... More Library References...

```
foo.ts
  Matched by include pattern '**/*' in 'tsconfig.json'
目前，TypeScript 不保证输出文件的格式 - 它在将来可能会改变。 关于这一点，我们也打算改进输出文件格式，请给出你的建议！
更多详情，请参考 PR！
```

20.7 [改进逻辑表达式中的未被调用函数检查](#)

感谢 [Alex Tarasyuk](#) 提供的持续改进，TypeScript 中的未调用函数检查现在也作用于 `&&` 和 `||` 表达式。

在 `--strictNullChecks` 模式下，下面的代码会产生错误。

```
function shouldDisplayElement(element: Element) {
```

```
    console.error(
      'The option `excludes` is not valid. Did you mean
`exclude`?'
    );
  }
}
```

为了便于处理以上情况，在从前的时候，TypeScript 允许使用点语法来访问通过字符串索引签名定义的属性。 这会让从 JavaScript 代码到 TypeScript 代码的迁移工作变得容易。 然而，放宽限制同样意味着更容易出现属性名拼写错误。

```
interface Options {
  /** 要排除的文件模式。 */
  exclude?: string[];

  /**
   * 这会将其余所有未声明的属性定义为 'any' 类型。
   */
  [x: string]: any;
}
// ---cut---
function processOptions(opts: Options) {
  // ...

  // 注意，我们不小心访问了错误的 `excludes`。
  // 但是！这是合法的！
  for (const excludePattern of opts.excludes) {
    // ...
  }
}
```

在某些情况下，用户会想要选择使用索引签名 - 在使用点号语法进行属性访问时，如果访问了没有明确定义的属性，就得到一个错误。

这就是为什么 TypeScript 引入了一个新的 --

`noPropertyAccessFromIndexSignature` 编译选项。 在该模式下，你可以有选择的启用 TypeScript 之前的行为，即在上述使用场景中产生错误。 该编译选项不属于 `strict` 编译选项集合的一员，因为我们知道该功能只适用于部分用户。

更多详情，请参考 [PR](#)。 我们同时要感谢 [Wenlu Wang](#) 为该功能的付出！

20.5 [abstract 构造签名](#)

TypeScript 允许将一个类标记为 `abstract`。 这相当于告诉 TypeScript 这个类只是用于继承，并且有些成员需要在子类中实现，以便能够真正地创建出实例。

```
abstract class Shape {
  abstract getArea(): number;
}
```

```
// 不能创建抽象类的实例
new Shape();
```

```
class Square extends Shape {
  #sideLength: number;
```



```
ts_upgrade_from_2.2_to_5.3
    constructor(sideLength: number) {
        super();
        this.#sideLength = sideLength;
    }

    getArea() {
        return this.#sideLength ** 2;
    }
}
```

// 没问题

```
new Square(42);
```

为了能够确保一贯的对 new 一个 abstract 类进行限制，不允许将 abstract 类赋值给接收构造签名的值。

```
abstract class Shape {
    abstract getArea(): number;
}
```

```
interface HasArea {
    getArea(): number;
}
```

// 不能将抽象构造函数类型赋值给非抽象构造函数类型。

```
let Ctor: new () => HasArea = Shape;
```

如果有代码调用了 new Ctor，那么上述的行为是正确的，但若想要编写 Ctor 的子类，就会出现过度限制的情况。

```
abstract class Shape {
    abstract getArea(): number;
}
```

```
interface HasArea {
    getArea(): number;
}
```

```
function makeSubclassWithArea(Ctor: new () => HasArea) {
    return class extends Ctor {
        getArea() {
            return 42;
        }
    };
}
```

// 不能将抽象构造函数类型赋值给非抽象构造函数类型。

```
let MyShape = makeSubclassWithArea(Shape);
```

对于内置的工具类型 InstanceType 来讲，它也不是工作得很好。

// 错误！

// 不能将抽象构造函数类型赋值给非抽象构造函数类型。

```
type MyInstance = InstanceType<typeof Shape>;
```

这就是为什么 TypeScript 4.2 允许在构造签名上指定 abstract 修饰符。

```
abstract class Shape {
    abstract getArea(): number;
}
// ---cut---
interface HasArea {
    getArea(): number;
}
```

// Works!

```
let Ctor: abstract new () => HasArea = Shape;
```

在构造签名上添加 abstract 修饰符表示可以传入一个 abstract 构造函数。它不会阻止你传入其它具体的类/构造函数 - 它只是想表达不会直接调用这个构造函数，因此可以安全地传入任意一种类型。

这个特性允许我们编写支持抽象类的混入工厂函数。例如，在下例中，我们可以同时使用混入函数 withStyles 和 abstract 类 SuperClass。

```
abstract class SuperClass {
    abstract someMethod(): void;
    badda() {}
}
```

```
type AbstractConstructor<T> = abstract new (...args: any[]) => T
```

```
function withStyles<T extends AbstractConstructor<object>>>(Ctor: T) {
    abstract class StyledClass extends Ctor {
        getStyles() {
            // ...
        }
    }
    return StyledClass;
}
```

```
class SubClass extends withStyles(SuperClass) {
    someMethod() {
        this.someMethod()
    }
}
```

注意，withStyles 展示了一个特殊的规则，若一个类（StyledClass）继承了被抽象构造函数所约束的泛型值，那么这个类也需要被声明为 abstract。由于无法知道传入的类是否拥有更多的抽象成员，因此也无法知道子类是否实现了所有的抽象成员。

更多详情，请参考 [PR](#)。

20.6 使用 --explainFiles 来理解工程的结构

TypeScript 用户时常会问“为什么 TypeScript 包含了这个文件？”。推断程序中所包含的文件是个很复杂的过程，比如有很多原因会导致使用了 lib.d.ts 文件的组合，会导致 node_modules 中的文件被包含进来，会导致有些已经 exclude 的文件被包含进来。这就是 TypeScript 提供 --explainFiles 的原因。

```
tsc --explainFiles
```

```
// Don't allow NaN and stuff.
if (!Number.isFinite(num)) {
    this.#size = 0;
    return;
}

this.#size = num;
}
```

上例中，set 存取器使用了更广泛的类型种类（string、boolean 和 number），但 get 存取器保证它的值为 number。现在，我们再给这类属性赋予其它类型的值就不会报错了！

```
class Thing {
    #size = 0;

    get size(): number {
        return this.#size;
    }

    set size(value: string | number | boolean) {
        let num = Number(value);

        // Don't allow NaN and stuff.
        if (!Number.isFinite(num)) {
            this.#size = 0;
            return;
        }

        this.#size = num;
    }
}
// ---cut---
let thing = new Thing();

// 可以给 `thing.size` 赋予其它类型的值！
thing.size = 'hello';
thing.size = true;
thing.size = 42;
```

// 读取 `thing.size` 总是返回数字！

```
let mySize: number = thing.size;
```

当需要判定两个同名属性间的关系时，TypeScript 将只考虑“读取的”类型（比如，get 存取器上的类型）。而“写入”类型只在直接写入属性值时才会考虑。

注意，这个模式不仅作用于类。你也可以在对象字面量中为 getter 和 setter 指定不同的类型。

```
function makeThing(): Thing {
    let size = 0;
    return {
        get size(): number {
```

```
// ...
return true;
}
```

```
function getVisibleItems(elements: Element[]) {
    return elements.filter((e) => shouldDisplayElement &&
e.children.length);
    // ~~~~~
    // 该条件表达式永远返回 true，因为函数永远是定义了的。
    // 你是否想要调用它？
}
更多详情，请参考 PR。
```

20.8 解构出来的变量可以被明确地标记为未使用的

感谢 [Alex Tarasyuk](#) 提供的另一个 PR，你可以使用下划线（_ 字符）将解构变量标记为未使用的。

```
let [_first, second] = getValues();
```

在之前，如果 _first 未被使用，那么在启用了 noUnusedLocals 时 TypeScript 会产生一个错误。现在，TypeScript 会识别出使用了下划线的 _first 变量是有意的未使用的变量。

更多详情，请参考 PR。

20.9 放宽了在可选属性和字符串索引签名间的限制

字符串索引签名可用于为类似于字典的对象添加类型，它表示允许使用任意的键来访问对象：

```
const movieWatchCount: { [key: string]: number } = {};
```

```
function watchMovie(title: string) {
    movieWatchCount[title] = (movieWatchCount[title] ?? 0) + 1;
}
```

当然了，对于不在字典中的电影名而言 movieWatchCount[title] 的值为 undefined。

（TypeScript 4.1 增加了 --noUncheckedIndexedAccess 选项，在访问索引签名时会增加 undefined 值。）即便一定会有 movieWatchCount 中不存在的属性，但在之前的版本中，由于 undefined 值的存在，TypeScript 会将可选对象属性视为不可以赋值给兼容的索引签名。

```
type WesAndersonWatchCount = {
    'Fantastic Mr. Fox'?: number;
    'The Royal Tenenbaums'?: number;
    'Moonrise Kingdom'?: number;
    'The Grand Budapest Hotel'?: number;
};
```

```
declare const wesAndersonWatchCount: WesAndersonWatchCount;
```

```
const movieWatchCount: { [key: string]: number } =
wesAndersonWatchCount;
```

// ~~~~~ 错误！

// 类型 'WesAndersonWatchCount' 不允许赋值给类型 '{ [key: string]: number; }'。

// 属性 '"Fantastic Mr. Fox"' 与索引签名不兼容。

// 类型 'number | undefined' 不允许赋值给类型 'number'。

// 类型 'undefined' 不允许赋值给类型 'number'。（2322）

v4.2 => 解构出来的变量可以被明确地标记为未使用的

ts_upgrade_from_2.2_to_5.3

TypeScript 4.2 允许这样赋值。 但是不允许使用带有 `undefined` 类型的非可选属性进行赋值，也不允许将 `undefined` 值直接赋值给某个属性：

```
type BatmanWatchCount = {
  'Batman Begins': number | undefined;
  'The Dark Knight': number | undefined;
  'The Dark Knight Rises': number | undefined;
};
```

```
declare const batmanWatchCount: BatmanWatchCount;
```

// 在 TypeScript 4.2 中仍是错误。

```
const movieWatchCount: { [key: string]: number } = batmanWatchCount;
```

// 在 TypeScript 4.2 中仍是错误。

// 索引签名不允许显式地赋值 `undefined`。

```
movieWatchCount["It's the Great Pumpkin, Charlie Brown"] = undefined;
```

这条新规则不适用于数字索引签名，因为它们被当成是类数组的并且是稠密的：

```
declare let sortOfArrayish: { [key: number]: string };
```

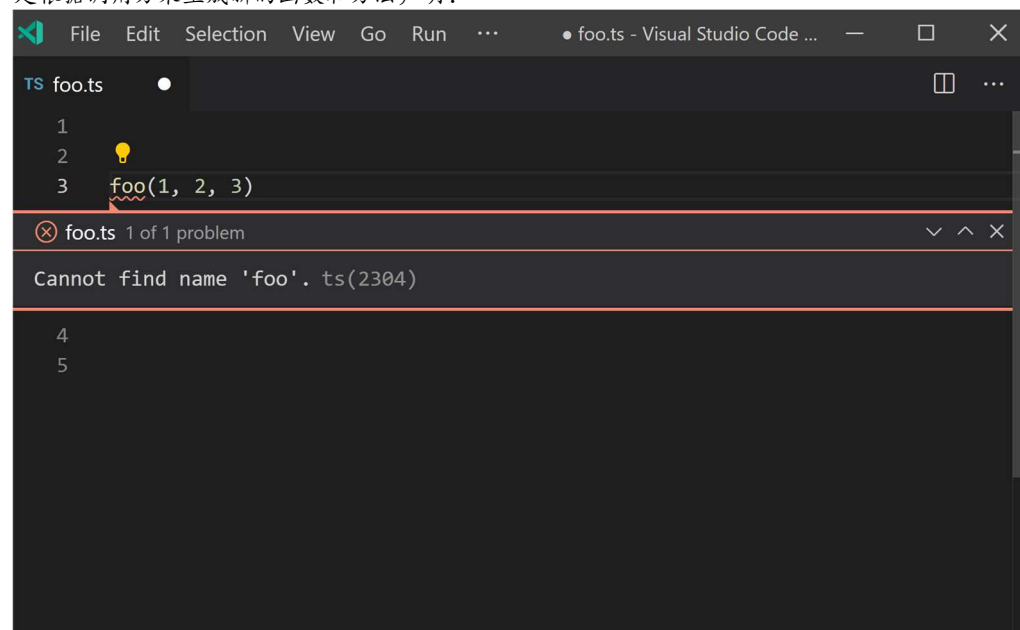
```
declare let numberKeys: { 42?: string };
```

```
sortOfArrayish = numberKeys;
```

更多详情，请参考 [PR](#)。

20.10 [声明缺失的函数](#)

感谢 [Alexander Tarasyuk](#) 提交的 [PR](#)，TypeScript 支持了一个新的快速修复功能，那就是根据调用方来生成新的函数和方法声明！



21 v4.3

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.3.html>

21.1 [拆分属性的写入类型](#)

在 JavaScript 中，API 经常需要对传入的值进行转换，然后再保存。这种情况在 `getter` 和 `setter` 中也常出现。例如，在某个类中的一个 `setter` 总是需要将传入的值转换成 `number`，然后再保存到私有字段中。

```
class Thing {
  #size = 0;

  get size() {
    return this.#size;
  }

  set size(value) {
    let num = Number(value);

    // Don't allow NaN and stuff.
    if (!Number.isFinite(num)) {
      this.#size = 0;
      return;
    }

    this.#size = num;
  }
}
```

我们该如何将这段 JavaScript 代码改写为 TypeScript 呢？从技术上讲，我们不必进行任何特殊处理 - TypeScript 能够识别出 `size` 是一个数字。

但问题在于 `size` 不仅仅是允许将 `number` 赋值给它。我们可以通过将 `size` 声明为 `unknown` 或 `any` 来解决这个问题：

```
class Thing {
  // ...
  get size(): unknown {
    return this.#size;
  }
}
```

但这不太友好 - `unknown` 类型会强制在读取 `size` 值时进行类型断言，同时 `any` 类型也不会去捕获错误。如果我们真想要为转换值的 API 进行建模，那么之前版本的 TypeScript 会强制我们在准确性（读取容易，写入难）和自由度（写入方便，读取难）两者之间进行选择。

这就是 TypeScript 4.3 允许分别为读取和写入属性值添加类型的原因。

```
class Thing {
  #size = 0;

  get size(): number {
    return this.#size;
  }

  set size(value: string | number | boolean) {
    let num = Number(value);
```

```
ts_upgrade_from_2.2_to_5.3
declare let s1: `${number}-${number}-${number}`;
declare let s2: `1-2-3`;
declare let s3: `${number}-2-3`;
```

```
s1 = s2;
s1 = s3;
```

在检查字符串字面量类型时，例如 `s2`，TypeScript 可以匹配字符串的内容并计算出在第一个赋值语句中 `s2` 与 `s1` 兼容。然而，当再次遇到模版字符串类型时，则会直接放弃进行匹配。

结果就是，像 `s3` 到 `s1` 的赋值语句会出错。

现在，TypeScript 会去判断是否模版字符串的每一部分都能够成功匹配。你现在可以混合并使用不同的替换字符串来匹配模版字符串，TypeScript 能够更好地计算出它们是否兼容。

```
declare let s1: `${number}-${number}-${number}`;
declare let s2: `1-2-3`;
declare let s3: `${number}-2-3`;
declare let s4: `1-${number}-3`;
declare let s5: `1-2-${number}`;
declare let s6: `${number}-2-${number}`;
```

// 下列均无问题

```
s1 = s2;
s1 = s3;
s1 = s4;
s1 = s5;
s1 = s6;
```

在这项改进之后，TypeScript 提供了更好的推断能力。示例如下：

```
declare function foo<V extends string>(arg: `${V}*`): V;
```

```
function test<T extends string>(s: string, n: number, b: boolean, t:
T) {
  let x1 = foo('*hello*'); // "hello"
  let x2 = foo('**hello**'); // "**hello*"
  let x3 = foo(`${s}*` as const); // string
  let x4 = foo(`${n}*` as const); // `${number}`
  let x5 = foo(`${b}*` as const); // "true" | "false"
  let x6 = foo(`${t}*` as const); // `${T}`
  let x7 = foo(`${s}*` as const); // `${string}*`
}
```

更多详情，请参考 [PR: 利用按上下文归类](#)，以及 [PR: 改进模版字符串类型的类型推断和检查](#)。

21.4 ECMAScript #private 的类成员

TypeScript 4.3 扩大了在类中可被声明为 `#private` `#names` 的成员的的范围，使得它们在运行时成为真正的私有的。除属性外，方法和存取器也可进行私有命名。

```
class Foo {
  #someMethod() {
    //...
  }

  get #someValue() {
    return 100;
  }
}
```

```
return size;
},
set size(value: string | number | boolean) {
  let num = Number(value);

  // Don't allow NaN and stuff.
  if (!Number.isFinite(num)) {
    size = 0;
    return;
  }

  size = num;
},
};
}
```

事实上，我们在接口/对象类型上支持了为属性的读和写指定不同的类型。

// Now valid!

```
interface Thing {
  get size(): number;
  set size(value: number | string | boolean);
}
```

此处的一个限制是属性的读取类型必须能够赋值给属性的写入类型。换句话说，getter 的类型必须能够赋值给 setter。这在一定程度上确保了一致性，一个属性应该总是能够赋值给它自身。

更多详情，请参考 [PR](#)。

21.2 override 和 --noImplicitOverride 标记

当在 JavaScript 中去继承一个类时，覆写方法十分容易 - 但不幸的是可能会犯一些错误。其中一个就是会导致丢失重命名。例如：

```
class SomeComponent {
  show() {
    // ...
  }
  hide() {
    // ...
  }
}
```

```
class SpecializedComponent extends SomeComponent {
  show() {
    // ...
  }
  hide() {
    // ...
  }
}
```

`SpecializedComponent` 是 `SomeComponent` 的子类，并且覆写了 `show` 和 `hide` 方法。猜一猜，如果有人想要将 `show` 和 `hide` 方法删除并用单个方法代替会发生什么？

```
class SomeComponent {
```

ts_upgrade_from_2.2_to_5.3

```

-   show() {
-       // ...
-   }
-   hide() {
-       // ...
-   }
+   setVisible(value: boolean) {
+       // ...
+   }
}
class SpecializedComponent extends SomeComponent {
    show() {
        // ...
    }
    hide() {
        // ...
    }
}

```

哦，不！SpecializedComponent 中的方法没有被更新。而是变为添加了两个没用的 show 和 hide 方法，它们可能都没有被调用。

此处的部分问题在于我们不清楚这里是想添加新的方法，还是想覆写已有的方法。因此，TypeScript 4.3 增加了 override 关键字。

```

class SpecializedComponent extends SomeComponent {
    override show() {
        // ...
    }
    override hide() {
        // ...
    }
}

```

当一个方法被标记为 override，TypeScript 会确保在基类中存在同名的方法。

```

class SomeComponent {
    setVisible(value: boolean) {
        // ...
    }
}
class SpecializedComponent extends SomeComponent {
    override show() {
        // ~~~~
        // 错误
    }
}

```

这是一项重大改进，但如果忘记在方法前添加 override 则不会起作用 - 这也是人们常犯的错误。

例如，可能会不小心覆写了基类中的方法，并且还没有意识到。

```

class Base {
    someHelperMethod() {
        // ...
    }
}

```

}

```

class Derived extends Base {
    // 不是真正想覆写基类中的方法，
    // 只是想编写一个本地的帮助方法
    someHelperMethod() {
        // ...
    }
}

```

因此，TypeScript 4.3 中还增加了一个 --noImplicitOverride 选项。当启用了该选项，如果覆写了父类中的方法但没有添加 override 关键字，则会产生错误。在上例中，如果启用了 --noImplicitOverride，则 TypeScript 会报错，并提示我们需要重命名 Derived 中的方法。

感谢开发者社区的贡献。该功能是在[这个 PR](#)中由 [Wenlu Wang](#) 实现，一个更早的 override 实现是由 [Paul Cody Johnston](#) 完成。

21.3 模版字符串类型改进

在近期的版本中，TypeScript 引入了一种新类型，即：模版字符串类型。它可以通过连接操作来构造类字符串类型：

```

type Color = 'red' | 'blue';
type Quantity = 'one' | 'two';

```

```

type SeussFish = `${Quantity | Color} fish`;
// 等同于
//   type SeussFish = "one fish" | "two fish"
//                       | "red fish" | "blue fish";
// 或者与其它类字符串类型进行模式匹配。
declare let s1: `${number}-${number}-${number}`;
declare let s2: `1-2-3`;

```

// 正确

s1 = s2;

我们做的首个改动是 TypeScript 应该在何时去推断模版字符串类型。当一个模版字符串的类型是由类字符串字面量类型进行的按上下文归类（比如，TypeScript 识别出将模版字符串传递给字面量类型时），它会得到模版字符串类型。

```

function bar(s: string): `hello ${string}` {
    // 之前会产生错误，但现在没有问题
    return `hello ${s}`;
}

```

在类型推断和 extends string 的类型参数上也会起作用。

```

declare let s: string;
declare function f<T extends string>(x: T): T;

```

// 以前: string

// 现在: `hello-\${string}`

```
let x2 = f(`hello ${s}`);
```

另一个主要的改动是 TypeScript 会更好地进行类型关联，并在不同的模版字符串之间进行推断。

示例如下：

v4.3 => 模版字符串类型改进


```
// ...

return collection;
// ~~~~~
// 错误: 类型 'T[]' 不能赋值给类型 'C'。
// 'T[]' 可以赋值给 'C' 的类型约束, 但是
// 'C' 可能使用 'Set<T> | T[]' 的不同子类型进行实例化。
}
```

TypeScript 4.3 是怎么做的? 在一些关键的位置, 类型系统会去查看类型的约束。例如, 在遇到 `collection.length` 时, TypeScript 不去关心 `collection` 的类型为 `C`, 而是会去查看可访问的属性, 而这些是由 `T[] | Set<T>` 泛型约束决定的。在类似的地方, TypeScript 会获取由泛型约束细化出的类型, 因为它包含了用户关心的信息; 而在其它的一些地方, TypeScript 会去细化初始的泛型类型 (但结果通常也是该泛型类型)。换句话说, 根据泛型值的使用方式, TypeScript 的处理方式会稍有不同。最终结果就是, 上例中的代码不会产生编译错误。更多详情, 请参考 [PR](#)。

21.7 [检查总是为真的 Promise](#)

在 `strictNullChecks` 模式下, 在条件语句中检查 `Promise` 是否真时会产生错误。

```
async function foo(): Promise<boolean> {
  return false;
}

async function bar(): Promise<string> {
  if (foo()) {
    // ~~~~~
    // Error!
    // This condition will always return true since
    // this 'Promise<boolean>' appears to always be defined.
    // Did you forget to use 'await'?
    return 'true';
  }
  return 'false';
}
```

[这项改动](#)是由 [Jack Works](#) 实现。

21.8 [static 索引签名](#)

与明确的类型声明相比, 索引签名允许我们在一个值上设置更多的属性。

```
class Foo {
  hello = 'hello';
  world = 1234;

  // 索引签名:
  [propName: string]: string | number | undefined;
}
```

```
let instance = new Foo();
```

```
}

publicMethod() {
  // 可以使用
  // 可以在类内部访问私有命名成员。
  this.#someMethod();
  return this.#someValue;
}
}
```

```
new Foo().#someMethod();
// ~~~~~
// 错误!
// 属性 '#someMethod' 无法在类 'Foo' 外访问, 因为它是私有的。
```

```
new Foo().#someValue;
// ~~~~~
// 错误!
// 属性 '#someValue' 无法在类 'Foo' 外访问, 因为它是私有的。
更为广泛地, 静态成员也可以有私有命名。
```

```
class Foo {
  static #someMethod() {
    // ...
  }
}
```

```
Foo.#someMethod();
// ~~~~~
// 错误!
// 属性 '#someMethod' 无法在类 'Foo' 外访问, 因为它是私有的。
```

该功能是由 Bloomberg 的朋友开发的: [PR](#) - 由 [Titian Cernicova-Dragomir](#) 和 [Kubilay Kahveci](#) 开发, 并得到了 [Joey Watts](#), [Rob Palmer](#) 和 [Tim McClure](#) 的帮助支持。感谢他们!

21.5 [ConstructorParameters 可用于抽象类](#)

在 TypeScript 4.3 中, `ConstructorParameters` 工具类型可以用在 `abstract` 类上。

```
abstract class C {
  constructor(a: string, b: number) {
    // ...
  }
}
```

```
// 类型为 '[a: string, b: number]'
```

```
type CParams = ConstructorParameters<typeof C>;
这多亏了 TypeScript 4.2 支持了声明抽象的构造签名:
type MyConstructorOf<T> = {
  new (...args: any[]): T;
};
```

// 或使用简写形式:

```
type MyConstructorOf<T> = abstract new (...args: any[]) => T;
```

更多详情, 请参考 [PR](#)。

21.6 按上下文细化泛型类型

TypeScript 4.3 能够更智能地对泛型进行类型细化。这让 TypeScript 能够支持更多模式, 甚至有时还能够发现错误。

设想有这样的场景, 我们想要编写一个 `makeUnique` 函数。它接受一个 `Set` 或 `Array`, 如果接收的是 `Array`, 则对数组进行排序并去除重复的元素。最后返回初始的集合。

```
function makeUnique<T>(  
  collection: Set<T> | T[],  
  comparer: (x: T, y: T) => number  
) : Set<T> | T[] {  
  // 假设元素已经是唯一的  
  if (collection instanceof Set) {  
    return collection;  
  }  
  
  // 排序, 然后去重  
  collection.sort(comparer);  
  for (let i = 0; i < collection.length; i++) {  
    let j = i;  
    while (  
      j < collection.length &&  
      comparer(collection[i], collection[j + 1]) === 0  
    ) {  
      j++;  
    }  
    collection.splice(i + 1, j - i);  
  }  
  return collection;  
}
```

暂且不谈该函数的具体实现, 假设它就是某应用中的一个需求。我们可能会注意到, 函数签名没能捕获到 `collection` 的初始类型。我们可以定义一个类型参数 `C`, 并用它代替 `Set<T> | T[]`。

```
- function makeUnique<T>(collection: Set<T> | T[], comparer: (x: T, y:  
T) => number): Set<T> | T[]
```

```
+ function makeUnique<T, C extends Set<T> | T[]>(collection: C,  
comparer: (x: T, y: T) => number): C
```

在 TypeScript 4.2 以及之前的版本中, 如果这样做的话会产生很多错误。

```
function makeUnique<T, C extends Set<T> | T[]>(  
  collection: C,  
  comparer: (x: T, y: T) => number  
) : C {  
  // 假设元素已经是唯一的  
  if (collection instanceof Set) {  
    return collection;  
  }  
}
```

```
// 排序, 然后去重  
collection.sort(comparer);  
// ~~~~~  
// 错误: 属性 'sort' 不存在于类型 'C' 上。  
for (let i = 0; i < collection.length; i++) {  
  // ~~~~~  
  // 错误: 属性 'length' 不存在于类型 'C' 上。  
  let j = i;  
  while (  
    j < collection.length &&  
    comparer(collection[i], collection[j + 1]) === 0  
  ) {  
    // ~~~~~  
    // 错误: 属性 'length' 不存在于类型 'C' 上。  
    // ~~~~~  
    // 错误: 元素具有隐式的 'any' 类型, 因为 'number' 类型的表达式不能用来索引  
    'Set<T> | T[]' 类型。  
    j++;  
  }  
  collection.splice(i + 1, j - i);  
  // ~~~~~  
  // 错误: 属性 'splice' 不存在于类型 'C' 上。  
}  
return collection;  
}
```

全是错误! 为何 TypeScript 要对我们如此刻薄?

问题在于进行 `collection instanceof Set` 检查时, 我们期望它能够成为类型守卫, 并根据条件将 `Set<T> | T[]` 类型细化为 `Set<T>` 和 `T[]` 类型; 然而, 实际上 TypeScript 没有对 `Set<T> | T[]` 进行处理, 而是去细化泛型值 `collection`, 其类型为 `C`。

虽是细微的差别, 但结果却不同。TypeScript 不会去读取 `C` 的泛型约束 (即 `Set<T> | T[]`) 并细化它。如果能让 TypeScript 由 `Set<T> | T[]` 进行类型细化, 它就会忘记在每个分支中 `collection` 的类型为 `C`, 因为没有比较好的办法去保留这些信息。假设 TypeScript 真这样做了, 那么上例也会有其它的错误。在函数返回的位置期望得到一个 `C` 类型的值, 但从每个分支中得到的却是 `Set<T>` 和 `T[]`, 因此 TypeScript 会拒绝编译。

```
function makeUnique<T>(  
  collection: Set<T> | T[],  
  comparer: (x: T, y: T) => number  
) : Set<T> | T[] {  
  // 假设元素已经是唯一的  
  if (collection instanceof Set) {  
    return collection;  
  }  
  // ~~~~~  
  // 错误: 类型 'Set<T>' 不能赋值给类型 'C'。  
  // 'Set<T>' 可以赋值给 'C' 的类型约束, 但是  
  // 'C' 可能使用 'Set<T> | T[]' 的不同子类型进行实例化。  
}
```

目前为止，TypeScript 的编辑器功能不会去尝试读取这些文件，因此“跳转到定义”会失败。在最好的情况下，“跳转到定义”会跳转到类似 `declare module "*.css"` 这样的声明语句上，如果它能够找到的话。

现在，在执行“跳转到定义”命令时，TypeScript 的语言服务会尝试跳转到正确的文件，即使它们不是 JavaScript 或 TypeScript 文件！在 CSS, SVGs, PNGs, 字体文件, Vue 文件等的导入语句上尝试一下吧。

更多详情，请参考 [PR](#)。

// 没问题

```
instance['whatever'] = 42;
```

// 类型为 'string | number | undefined'

```
let x = instance['something'];
```

目前为止，索引签名只允许在类的实例类型上进行设置。感谢 [Wenlu Wang](#) 的 [PR](#)，现在索引签名也可以声明为 `static`。

```
class Foo {
```

```
    static hello = 'hello';
```

```
    static world = 1234;
```

```
    static [propName: string]: string | number | undefined;
```

```
}
```

// 没问题

```
Foo['whatever'] = 42;
```

// 类型为 'string | number | undefined'

```
let x = Foo['something'];
```

类静态类型上的索引签名检查规则与类实例类型上的索引签名的检查规则是相同的，即每个静态属性必须与静态索引签名类型兼容。

```
class Foo {
```

```
    static prop = true;
```

```
    // ~~~~
```

```
    // 错误! 'boolean' 类型的属性 'prop' 不能赋值给字符串索引类型
```

```
    // 'string | number | undefined'.
```

```
    static [propName: string]: string | number | undefined;
```

```
}
```

21.9 [.tsbuildinfo 文件大小改善](#)

TypeScript 4.3 中，作为 `--incremental` 构建组分的 `.tsbuildinfo` 文件会变得非常小。这得益于一些内部格式的优化，使用以数值标识的查找表来替代重复多次的完整路径以及类似的信息。这项工作的灵感源自于 [Tobias Koppers](#) 的 [PR](#)，而后在 [PR](#) 中实现，并在 [PR](#) 中进行优化。

我们观察到了 `.tsbuildinfo` 文件有如下的变化：

- 1MB 到 411 KB
- 14.9MB 到 1MB
- 1345MB 到 467MB

不用说，缩小文件的尺寸会稍微加快构建速度。

21.10 [在 --incremental 和 --watch 中进行惰性计算](#)

`--incremental` 和 `--watch` 模式的一个问题是虽然它会加快后续的编译速度，但是首次编译很慢 - 有时会非常地慢。这是因为在该模式下需要保存和计算当前工程的一些信息，有时还需要将这些信息写入 `.tsbuildinfo` 文件，以备后续之用。

因此，TypeScript 4.3 也对 `--incremental` 和 `--watch` 进行了首次构建时的优化，让它可以和普通构建一样快。为了达到目的，大部分信息会进行按需计算，而不是和往常一样全部一次性计算。虽然这会加重后续构建的负担，但是 TypeScript 的 `--incremental` 和 `--watch` 功能会智能地处理一小部分文件，并保存住会对后续构建有用的

ts_upgrade_from_2.2_to_5.3

信息。这就好比，`--incremental` 和 `--watch` 构建会进行“预热”，并能够在多次修改文件后加速构建。

在一个包含了 3000 个文件的仓库中，**这能节约大概三分之一的构建时间！**

[这项改进](#) 是由 [Tobias Koppers](#) 开启，并在 [PR](#) 里完成。感谢他们！

21.11 导入语句的补全

在 JavaScript 中，关于导入导出语句的一大痛点是其排序问题 - 尤其是导入语句的写法如下：

```
import { func } from './module.js';
```

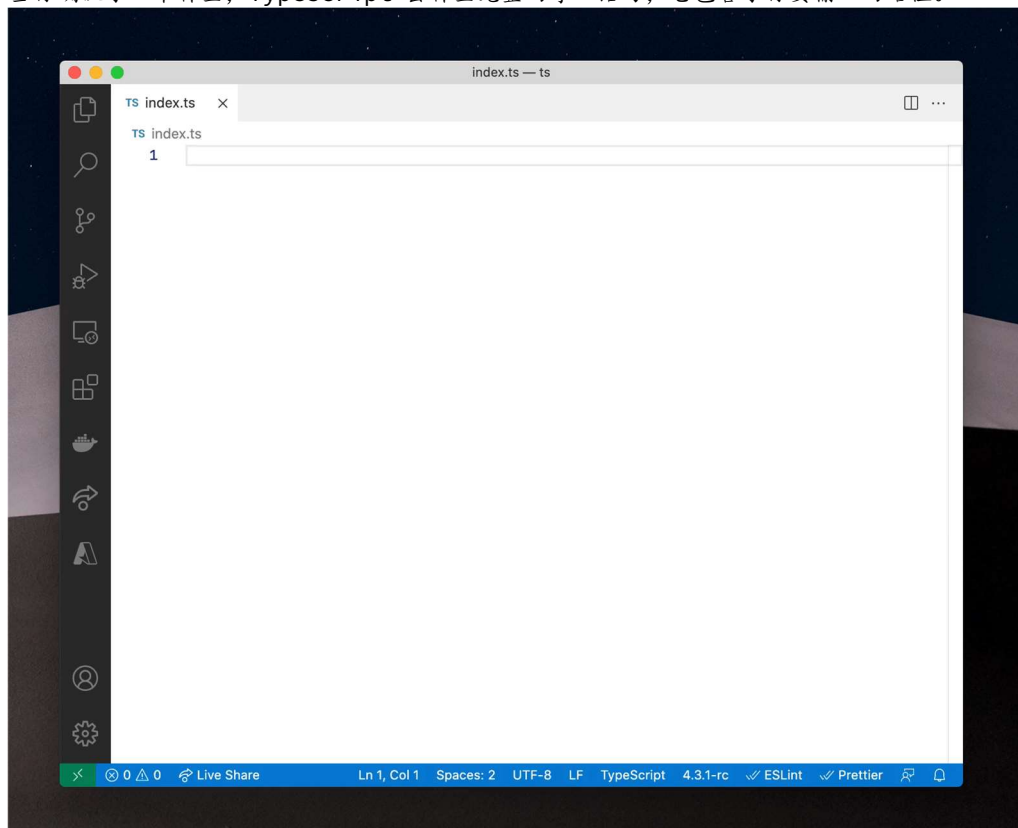
而非

```
from './module.js' import { func };
```

这导致了在书写完整的导入语句时很难受，因为自动补全无法工作。例如，你输入了 `import {`，TypeScript 不知道你要从哪个模块里导入，因此它不能提供补全信息。

为缓解该问题，我们可以利用自动导入功能！自动导入能够提供每个可能导出并在文件顶端插入一条导入语句。

因此当你输入 `import` 语句并没提供一个路径时，TypeScript 会提供一个可能的导入列表。当你确认了一个补全，TypeScript 会补全完整的导入语句，它包含了你要输入的路径。



该功能需要编辑器的支持。你可以在 [Insiders 版本的 Visual Studio Code](#) 中进行尝试。

更多详情，请参考 [PR](#)！

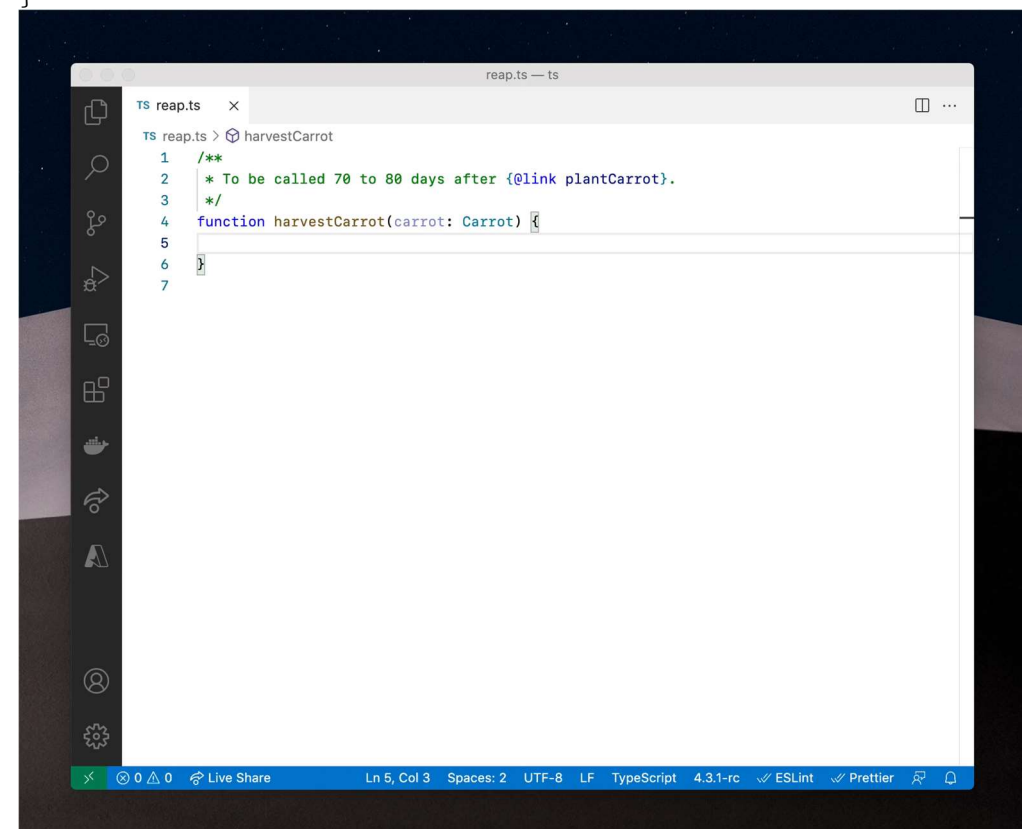
21.12 编辑器对 @link 标签的支持

TypeScript 现在能够理解 `@link` 标签，并会解析它指向的声明。也就是说，你将鼠标悬停在 `@link` 标签上会得到一个快速提示，或者使用“跳转到定义”或“查找全部引用”命令。

例如，在支持 TypeScript 的编辑器中你可以在 `@link bar` 中的 `bar` 上使用跳转到定义，它会跳转到 `bar` 的函数声明。

```
/**
 * To be called 70 to 80 days after {@link plantCarrot}.
 */
function harvestCarrot(carrot: Carrot) {}
```

```
/**
 * Call early in spring for best results. Added in v2.1.0.
 * @param seed Make sure it's a carrot seed!
 */
function plantCarrot(seed: Seed) {
  // TODO: some gardening
}
```



更多详情，请参考 [PR](#)！

21.13 在非 JavaScript 文件上的跳转到定义

许多加载器允许用户在 JavaScript 的导入语句中导入资源文件。例如典型的 `import './styles.css'` 语句。

v4.3 => 编辑器对 `@link` 标签的支持

ts_upgrade_from_2.2_to_5.3

来索引对象。TypeScript 也无法表示由一部分 string 类型的键组成的索引签名 - 例如, 对象属性名是以 data- 字符串开头的索引签名。

TypeScript 4.4 解决了这个问题, 允许 symbol 索引签名以及模版字符串。

例如, TypeScript 允许声明一个接受任意 symbol 值作为键的对象类型。

```
interface Colors {
    [sym: symbol]: number;
}

const red = Symbol('red');
const green = Symbol('green');
const blue = Symbol('blue');
```

```
let colors: Colors = {};
```

```
// 没问题
colors[red] = 255;
let redVal = colors[red];
// ^ number
```

```
colors[blue] = 'da ba dee';
// 错误: 'string' 不能赋值给 'number'
相似地, 可以定义带有模版字符串的索引签名。一个场景是用来免除对以 data- 开头的属性名执行的 TypeScript 额外属性检查。当传递一个对象字面量给目标类型时, TypeScript 会检查是否存在相比于目标类型的额外属性。
```

```
interface Options {
    width?: number;
    height?: number;
}
```

```
let a: Options = {
    width: 100,
    height: 100,

    'data-blah': true,
};
```

```
interface OptionsWithDataProps extends Options {
    // 允许以 'data-' 开头的属性
    [optName: `data-${string}`]: unknown;
}
```

```
let b: OptionsWithDataProps = {
    width: 100,
    height: 100,
    'data-blah': true,

    // 使用未知属性会报错, 不包括以 'data-' 开始的属性
    'unknown-property': true,
```

22 v4.4

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.4.html>

22.1 针对条件表达式和判别式的别名引用进行控制流分析

在 JavaScript 中, 总会用多种方式对某个值进行检查, 然后根据不同类型的值执行不同的操作。TypeScript 能够理解这些检查, 并将它们称作为类型守卫。我们不需要在变量的每一个使用位置上都指明类型, TypeScript 的类型检查器能够利用基于控制流的分析技术来检查是否在前面使用了类型守卫。

例如, 可以这样写

```
function foo(arg: unknown) {
    if (typeof arg === 'string') {
        console.log(arg.toUpperCase());
        //      ^?
    }
}
```

这个例子中, 我们检查 arg 是否为 string 类型。TypeScript 识别出了 typeof arg === "string" 检查, 它被当作是一个类型守卫, 并且知道在 if 分支内 arg 的类型为 string。这样就可以正常地访问 string 类型上的方法, 例如 toUpperCase()。但如果我们将条件表达式提取到一个名为 argIsString 的常量会发生什么?

// 在 TS 4.3 及以下版本

```
function foo(arg: unknown) {
    const argIsString = typeof arg === 'string';
    if (argIsString) {
        console.log(arg.toUpperCase());
        //      ~~~~~
        // 错误! 'unknown' 类型上不存在 'toUpperCase' 属性。
    }
}
```

在之前版本的 TypeScript 中, 这样做会产生错误 - 就算 argIsString 的值为类型守卫, TypeScript 也会丢掉这个信息。这不是想要的结果, 因为我们可能想要在不同的地方重用这个检查。为了绕过这个问题, 通常需要重复多次代码或使用类型断言。

在 TypeScript 4.4 中, 情况有所改变。上面的例子不再产生错误! 当 TypeScript 看到我们在检查一个常量时, 会额外检查它是否包含类型守卫。如果那个类型守卫操作的是 const 常量, 某个 readonly 属性或某个未修改的参数, 那么 TypeScript 能够对该值进行类型细化。

不同种类的类型守卫都支持, 不只是 typeof 类型守卫。例如, 对于可辨识联合类型同样适用。

```
type Shape =
    | { kind: 'circle'; radius: number }
    | { kind: 'square'; sideLength: number };
```

```
function area(shape: Shape): number {
    const isCircle = shape.kind === 'circle';
    if (isCircle) {
        // 知道此处为 circle
        return Math.PI * shape.radius ** 2;
```



```
ts_upgrade_from_2.2_to_5.3
} else {
    // 知道此处为 square
    return shape.sideLength ** 2;
}
}
```

在 TypeScript 4.4 版本中对判别式的分析又进了一层 - 现在可以提取出判别式然后细化原来的对象类型。

```
type Shape =
  | { kind: 'circle'; radius: number }
  | { kind: 'square'; sideLength: number };
```

```
function area(shape: Shape): number {
    // Extract out the 'kind' field first.
    const { kind } = shape;

    if (kind === 'circle') {
        // We know we have a circle here!
        return Math.PI * shape.radius ** 2;
    } else {
        // We know we're left with a square here!
        return shape.sideLength ** 2;
    }
}
```

另一个例子，该函数会检查它的两个参数是否有内容。

```
function doSomeChecks(
    inputA: string | undefined,
    inputB: string | undefined,
    shouldDoExtraWork: boolean
) {
    const mustDoWork = inputA && inputB && shouldDoExtraWork;
    if (mustDoWork) {
        // We can access 'string' properties on both 'inputA' and
        'inputB'!
        const upperA = inputA.toUpperCase();
        const upperB = inputB.toUpperCase();
        // ...
    }
}
```

TypeScript 知道如果 mustDoWork 为 true 那么 inputA 和 inputB 都存在。也就是说不需要编写像 inputA! 这样的非空断言的代码来告诉 TypeScript inputA 不为 undefined。

一个好的性质是该分析同时具有可传递性。TypeScript 可以通过这些常量来理解在它们背后执行的检查。

```
function f(x: string | number | boolean) {
    const isString = typeof x === 'string';
    const isNumber = typeof x === 'number';
    const isStringOrNumber = isString || isNumber;
    if (isStringOrNumber) {
        x;
```

```
// ^?
} else {
    x;
    // ^?
}
```

```
}
```

注意这里会有一个截点 - TypeScript 并不是毫无限制地去追溯检查这些条件表达式，但对于大多数使用场景而言已经足够了。

这个功能能让很多直观的 JavaScript 代码在 TypeScript 里也好用，而不会妨碍我们。

更多详情请参考 [PR!](#)

22.2 [Symbol](#) 以及模版字符串索引签名

TypeScript 支持使用索引/签名来为对象的每个属性定义类型。这样我们就可以将对象当作字典类型来使用，把字符串放在方括号里来进行索引。

例如，可以编写由 string 类型的键映射到 boolean 值的类型。如果我们给它赋予 boolean 类型以外的值会报错。

```
interface BooleanDictionary {
    [key: string]: boolean;
}
```

```
declare let myDict: BooleanDictionary;
```

```
// 允许赋予 boolean 类型的值
```

```
myDict['foo'] = true;
myDict['bar'] = false;
```

```
// 错误
```

```
myDict['baz'] = 'oops';
```

虽说在这里 [Map 可能是更适合的数据结构](#)（具体的说是 Map<string, boolean>），但 JavaScript 对象通常更方便或者正是我们要操作的目标。

相似地，Array<T> 已经定义了 number 索引签名，我们可以插入和获取 T 类型的值。

// 这是 TypeScript 内置的部分 Array 类型

```
interface Array<T> {
    [index: number]: T;
```

```
// ...
```

```
}
```

```
let arr = new Array<string>();
```

```
// 没问题
```

```
arr[0] = 'hello!';
```

```
// 错误，期待一个 'string' 值
```

```
arr[1] = 123;
```

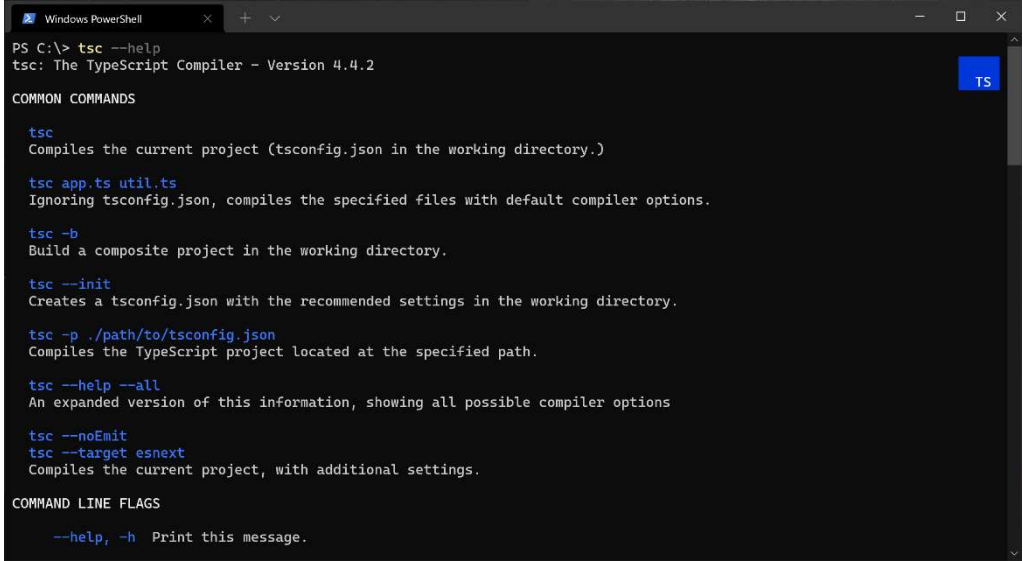
索引签名是一种非常有用的表达方式。然而，直到现在它们只能使用 string 和 number 类型的键（string 索引签名存在一个有意为之的怪异行为，它们可以接受 number 类型的键，因为 number 会被转换为字符串）。这意味着 TypeScript 不允许使用 symbol 类型的键

```
// Prints:
// 1
// 2
// 3
class Foo {
  static prop = 1
  static {
    console.log(Foo.prop++);
  }
  static {
    console.log(Foo.prop++);
  }
  static {
    console.log(Foo.prop++);
  }
}
```

感谢 [Wenlu Wang](#) 为 TypeScript 添加了该支持。 更多详情请参考 [PR](#)。

22.7 [tsc --help 更新与优化](#)

TypeScript 的 --help 选项完全更新了！ 感谢 [Song Gao](#)，我们[更新了编译选项的描述](#)和 [--help 菜单的配色样式](#)。



更多详情请参考 [Issue](#)。

22.8 [性能优化](#)

22.8.1 [更快地生成声明文件](#)

TypeScript 现在会缓存下内部符号是否可以在不同上下文中被访问，以及如何显示指定的类型。 这些改变能够改进 TypeScript 处理复杂类型时的性能，尤其是在使用了 --declaration 标记来生成 .d.ts 文件的时候。

更多详情请参考 [PR](#)。

};
最后，索引签名现在支持联合类型，只要它们是无限域原始类型的联合 - 尤其是：

- string
- number
- symbol
- 模版字符串（例如 `hello-\${string}`）

带有以上类型的联合的索引签名会展开为不同的索引签名。

```
interface Data {
  [optName: string | symbol]: any;
}
```

// 等同于

```
interface Data {
  [optName: string]: any;
  [optName: symbol]: any;
}
```

更多详情请参考 [PR](#)。

22.3 [Defaulting to the unknown Type in Catch Variables \(--useUnknownInCatchVariables\)](#)

22.4 [异常捕获变量的类型默认为 unknown \(--useUnknownInCatchVariables\)](#)

在 JavaScript 中，允许使用 throw 语句抛出任意类型的值，并在 catch 语句中捕获它。 因此，TypeScript 从前会将异常捕获变量的类型设置为 any 类型，并且不允许指定其它的类型注解：

```
try {
  // 谁知道它会抛出什么东西
  executeSomeThirdPartyCode();
} catch (err) {
  // err: any
  console.error(err.message); // 可以，因为类型为 'any'
  err.thisWillProbablyFail(); // 可以，因为类型为 'any' :(
}
```

当 TypeScript 引入了 unknown 类型后，对于追求高度准确性和类型安全的用户来讲在 catch 语句的捕获变量处使用 unknown 成为了比 any 类型更好的选择，因为它强制我们去检测要使用的值。 后来，TypeScript 4.0 允许用户在 catch 语句中明确地指定 unknown（或 any）类型，这样就可以根据情况有选择一使用更严格的类型检查； 然而，在每一处 catch 语句里手动指定 : unknown 是一件繁琐的事情。 因此，TypeScript 4.4 引入了一个新的标记 --useUnknownInCatchVariables。 它将 catch 语句捕获变量的默认类型由 any 改为 unknown。

```
declare function executeSomeThirdPartyCode(): void;
```

```
try {
  executeSomeThirdPartyCode();
} catch (err) {
  // err: unknown

  // Error! Property 'message' does not exist on type 'unknown'.
  console.error(err.message);
}
```

```
// Works! We can narrow 'err' from 'unknown' to 'Error'.
if (err instanceof Error) {
    console.error(err.message);
}
}
```

这个标记属性于 `--strict` 标记家族的一员。也就是说如果你启用了 `--strict`, 那么该标记也自动启用了。在 TypeScript 4.4 中, 你可能会看到如下的错误:

Property 'message' does not exist on type 'unknown'.

Property 'name' does not exist on type 'unknown'.

Property 'stack' does not exist on type 'unknown'.

如果我们不想处理 `catch` 语句中 `unknown` 类型的捕获变量, 那么可以明确使用 `: any` 类型注解, 这样就会关闭严格类型检查。

`declare function executeSomeThirdPartyCode(): void;`

```
try {
    executeSomeThirdPartyCode();
} catch (err: any) {
    console.error(err.message); // Works again!
}
```

更多详情请参考 [PR](#)。

22.5 [确切的可选属性类型 \(--exactOptionalPropertyTypes\)](#)

在 JavaScript 中, 读取对象上某个不存在的属性会得到 `undefined` 值。与此同时, 某个已有属性的值也允许为 `undefined` 值。有许多 JavaScript 代码都会对这些情况一视同仁, 因此最初 TypeScript 将可选属性视为添加了 `undefined` 类型。例如,

```
interface Person {
    name: string;
    age?: number;
}

// 等同于:
interface Person {
    name: string;
    age?: number | undefined;
}
```

这意味着用户可以给 `age` 明确地指定 `undefined` 值。

```
const p: Person = {
    name: 'Daniel',
    age: undefined, // This is okay by default.
};
```

因此默认情况下, TypeScript 不区分带有 `undefined` 类型的属性和不存在的属性。虽说这在大部分情况下是没问题的, 但并非所有的 JavaScript 代码都如此。像是 `Object.assign`, `Object.keys`, 对象展开 (`{ ...obj }`) 和 `for-in` 循环这样的函数和运算符会区别对待属性是否存在于对象之上。在 `Person` 例子中, 如果 `age` 属性的存在与否是至关重要的, 那么就可能会导致运行时错误。

在 TypeScript 4.4 中, 新的 `--exactOptionalPropertyTypes` 标记指明了可选属性的确切表示方式, 即不自动添加 `| undefined` 类型:

```
interface Person {
    name: string;
```

```
    age?: number;
}
```

```
// 启用 'exactOptionalPropertyTypes'
const p: Person = {
    name: 'Daniel',
    age: undefined, // 错误! undefined 不是一个成员
};
```

该标记**不是** `--strict` 标记家族的一员, 需要显式地开启。该标记要求同时启用 `--strictNullChecks` 标记。我们已经更新了 `DefinitelyTyped` 以及其它的声明定义来帮助进行平稳地过渡, 但你仍可能遇到一些问题, 这取决于代码的结构。更多详情请参考 [PR](#)。

22.6 [类中的 static 语句块](#)

TypeScript 4.4 支持了 [类中的 static 语句块](#), 一个即将到来的 ECMAScript 特性, 它能够帮助编写复杂的静态成员初始化代码。

```
declare function someCondition(): boolean
```

```
class Foo {
    static count = 0;

    // 静态语句块:
    static {
        if (someCondition()) {
            Foo.count++;
        }
    }
}
```

在静态语句块中允许编写一系列语句, 它们可以访问类中的私有字段。也就是说在初始化代码中能够编写语句, 不会暴露变量, 并且可以完全访问类的内部信息。

```
declare function loadLastInstances(): any[]
```

```
class Foo {
    static #count = 0;

    get count() {
        return Foo.#count;
    }

    static {
        try {
            const lastInstances = loadLastInstances();
            Foo.#count += lastInstances.length;
        }
        catch {}
    }
}
```

若不使用 `static` 语句块也能够编写上述代码, 只不过需要使用一些折中的 `hack` 手段。一个类可以有多个 `static` 语句块, 它们的运行顺序与编写顺序一致。

23 v4.5

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.5.html>

23.1 [支持从 node_modules 里读取 lib](#)

为确保对 TypeScript 和 JavaScript 的支持可以开箱即用，TypeScript 内置了一些声明文件（.d.ts）。这些声明文件描述了 JavaScript 语言中可用的 API，以及标准的浏览器 DOM API。虽说 TypeScript 会根据工程中 target 的设置来提供默认值，但你仍然可以通过在 tsconfig.json 文件中设置 lib 来指定包含哪些声明文件。

TypeScript 包含的声明文件偶尔也会成为缺点：

- 在升级 TypeScript 时，你必须处理 TypeScript 内置声明文件的升级带来的改变，这可能成为一项挑战，因为 DOM API 的变动十分频繁。
- 难以根据你的需求以及工程依赖的需求去定制声明文件（例如，工程依赖声明了需要使用 DOM API，那么你可能也必须要使用 DOM API）。

TypeScript 4.5 引入了覆盖特定内置 lib 的方式，它与 @types/ 的工作方式类似。在决定应包含哪些 lib 文件时，TypeScript 会先去检查 node_modules 下面的 @typescript/lib-* 包。例如，若将 dom 作为 lib 中的一项，那么 TypeScript 会尝试使用 node_modules/@typescript/lib-dom。然后，你就可以使用包管理器去安装特定的包作为 lib 中的某一项。例如，现在 TypeScript 会将 DOM API 发布到 @types/web。如果你想要给工程指定一个固定版本的 DOM API，你可以在 package.json 文件中添加如下代码：

```
{
  "dependencies": {
    "@typescript/lib-dom": "npm:@types/web"
  }
}
```

从 4.5 版本开始，你可以更新 TypeScript 和依赖管理工具生成的锁文件来确保使用固定版本的 DOM API。你可以根据自己情况来逐步更新类型声明。十分感谢 [saschanaz](#) 提供的帮助。更多详情，请参考 [PR](#)。

23.2 [改进 Awaited 类型和 Promise](#)

TypeScript 4.5 引入了一个新的 Awaited 类型。该类型用于描述 async 函数中的 await 操作，或者 Promise 上的 .then() 方法 - 尤其是递归地解开 Promise 的行为。

```
// A = string
type A = Awaited<Promise<string>>;
```

```
// B = number
type B = Awaited<Promise<Promise<number>>>>;
```

`// C = boolean | number`
`type C = Awaited<boolean | Promise<number>>>;`
Awaited 有助于描述现有 API，比如 JavaScript 内置的 Promise.all，Promise.race 等等。实际上，正是涉及 Promise.all 的类型推断问题促进了 Awaited 类型的产生。例如，下例中的代码在 TypeScript 4.4 及之前的版本中会失败。

```
declare function MaybePromise<T>(value: T): T | Promise<T> | PromiseLike<T>;
```

22.8.2 [更快地标准化路径](#)

TypeScript 经常需要对文件路径进行“标准化”操作来得到统一的格式，以便编译器能够随处使用它。它包括将反斜线替换成正斜线，或者删除路径中间的 ./ 和 ../ 片段。当 TypeScript 需要处理成千上万的路径时，这个操作就会很慢。在 TypeScript 4.4 里会先对路径进行快速检查，判断它们是否需要标准化。这些改进能够减少 5-10% 的工程加载时间，对于大型工程来讲效果会更加明显。更多详情请参考 [PR](#) 以及 [PR](#)。

22.8.3 [更快地路径映射](#)

TypeScript 现在会缓存构造的路径映射（通过 tsconfig.json 里的 paths）。对于拥有数百个路径映射的工程来讲效果十分明显。更多详情请参考 [PR](#)。

22.8.4 [更快地增量构建与 --strict](#)

这曾是一个缺陷，在 --incremental 模式下，如果启用了 --strict 则 TypeScript 会重新进行类型检查。这导致了不管是否开启了 --incremental 构建速度都挺慢。TypeScript 4.4 修复了这个问题，该修复也应用到了 TypeScript 4.3 里。更多详情请参考 [PR](#)。

22.8.5 [针对大型输出更快地生成 Source Map](#)

TypeScript 4.4 优化了为超大输出文件生成 source map 的速度。在构建旧版本的 TypeScript 编译器时，结果显示节省了 8% 的生成时间。感谢 [David Michon](#) 提供了这项[简洁的优化](#)。

22.8.6 [更快的 --force 构建](#)

当在工程引用上使用了 --build 模式时，TypeScript 必须执行“是否更新检查”来确定是否需要重新构建。在进行 --force 构建时，该检查是无关的，因为每个工程依赖都要被重新构建。在 TypeScript 4.4 里，--force 会避免执行无用的步骤并进行完整的构建。更多详情请参考 [PR](#)。

22.9 [JavaScript 中的拼写建议](#)

TypeScript 为在 Visual Studio 和 Visual Studio Code 等编辑器中的 JavaScript 编写体验赋能。大多数情况下，在处理 JavaScript 文件时，TypeScript 会置身事外；然而，TypeScript 经常能够提供有理有据的建议且不过分地侵入其中。这就是为什么 TypeScript 会为 JavaScript 文件提供拼写建议 - 不带有 // @ts-check 的文件或者关闭了 checkJs 选项的工程。即，TypeScript 文件中已有的 "Did you mean...?" 建议，现在它们也作用于 JavaScript 文件。这些拼写建议也暗示了代码中可能存在错误。我们在测试该特性时已经发现了已有代码中的一些错误！更多详情请参考 [PR](#)！

22.10 [内嵌提示 \(Inlay Hints\)](#)

TypeScript 4.4 支持了内嵌提示特性，它能帮助显示参数名和返回值类型等信息。可将其视为一种友好的“ghost text”。


```
export function getFavoriteColor(dayOfWeek: string) : Color | undefined {
  if (dayOfWeek === "Tuesday") {
    return new Color(
      kind: "rgb",
      red: 0x64,
      green: 0x95,
      blue: 0xED
    );
  }
}
```

该特性由 [Wenlu Wang](#) 的 [PR](#) 所实现。
他也在 [Visual Studio Code 里进行了集成](#) 并在 [July 2021 \(1.59\) 发布](#)。若你想尝试该特性，需确保安装了[稳定版](#)或 [insiders](#) 版本的编辑器。你也可以在 Visual Studio Code 的设置里修改何时何地显示内嵌提示。

22.11 [自动导入的补全列表里显示真正的路径](#)

当 Visual Studio Code 显示补全列表时，包含自动导入在内的补全列表里会显示指向模块的路径；然而，该路径通常不是 TypeScript 最终替换进来的模块描述符。该路径通常是相对于 *workspace* 的，如果你导入了 *moment* 包，你大概会看到 `node_modules/moment` 这样的路径。

```
cla
  classNames node_modules/classnames/index
  ClassificationTy... node_modules/typescrip...
  ClassificationTypeNa... node_modules/type...
  calendarFormat node_modules/moment/mome...
  callbackify
  callbackify
  CallTracker
```

这些路径很难处理且容易产生误导，尤其是插入的路径同时需要考虑 Node.js 的 `node_modules` 解析，路径映射，符号链接以及重新导出等。
这就是为什么 TypeScript 4.4 中的补全列表会显示真正的导入模块路径。

```
cla
  classNames classnames
  ClassificationType typescript
  ClassificationTypeNames typescript
  calendarFormat moment
  callbackify util
  CallTracker assert/strict
  Calendar antd
```

由于该计算可能很昂贵，当补全列表包含许多条目时最终的模块描述符会在你输入更多的字符时显示出来。你仍可能看到基于 *workspace* 的相对路径；然而，当编辑器“预热”后，再多输入几个字符它们会被替换为真正的路径。

当同时使用了这些选项时，需要有一种方式来表示导入语句是否可以被合法地丢弃。

TypeScript 已经有类似的功能，即 `import type`：

```
import type { BaseType } from "./some-module.js";
import { someFunc } from "./some-module.js";
```

```
export class Thing implements BaseType {
  // ...
}
```

这是有效的，但还可以提供更好的方式来避免使用两条导入语句从相同的模块中导入。因此，TypeScript 4.5 允许在每个命名导入前使用 `type` 修饰符，你可以按需混合使用它们。

```
import { someFunc, type BaseType } from "./some-module.js";
```

```
export class Thing implements BaseType {
  someMethod() {
    someFunc();
  }
}
```

上例中，在 `preserveValueImports` 模式下，能够确定 `BaseType` 可以被删除，同时 `someFunc` 应该被保留，于是就会生成如下代码：

```
import { someFunc } from "./some-module.js";
```

```
export class Thing {
  someMethod() {
    someFunc();
  }
}
```

更多详情，请参考 [PR](#)。

23.8 [私有字段存在性检查](#)

TypeScript 4.5 支持了检查对象上是否存在某私有字段的 ECMAScript Proposal。现在，你可以编写带有 `#private` 字段成员的类，然后使用 `in` 运算符检查另一个对象是否包含相同的字段。

```
class Person {
  #name: string;
  constructor(name: string) {
    this.#name = name;
  }

  equals(other: unknown) {
    return other &&
      typeof other === "object" &&
      #name in other && // <- this is new!
      this.#name === other.#name;
  }
}
```

该功能一个有趣的地方是，`#name in other` 隐含了 `other` 必须是使用 `Person` 构造的，因为只有在这种情况下才可能存在该字段。这是该提议中关键的功能之一，同时也是为什么这项提议叫作“`ergonomic brand checks`”的原因 - 因为私有字段通常作为一种“商标”来区

```
async function doSomething(): Promise<[number, number]> {
  const result = await Promise.all([MaybePromise(100),
  MaybePromise(200)]);
```

```
  // 错误！
  //
  //   [number | Promise<100>, number | Promise<200>]
  //
  // 不能赋值给类型
  //
  //   [number, number]
  return result;
}
```

现在，`Promise.all` 结合并利用 `Awaited` 来提供更好的类型推断结果，同时上例中的代码也不再会有错误。

更多详情，请参考 [PR](#)。

23.3 [模版字符串类型作为判别式属性](#)

TypeScript 4.5 可以对模版字符串类型的值进行细化，同时可以识别模版字符串类型的判别式属性。

例如，下面的代码在以前会出错，但在 TypeScript 4.5 里没有错误。

```
export interface Success {
  type: `${string}Success`;
  body: string;
}
```

```
export interface Error {
  type: `${string}Error`;
  message: string;
}
```

```
export function handler(r: Success | Error) {
  if (r.type === "HttpSuccess") {
    // 'r' 的类型为 'Success'
    let token = r.body;
  }
}
```

更多详情，请参考 [PR](#)。

23.4 [module es2022](#)

感谢 [Kagami S. Rosylight](#)，TypeScript 现在支持了一个新的 `module` 设置：`es2022`。`module es2022` 的主要功能是支持顶层的 `await`，即可以在 `async` 函数外部使用 `await`。该功能在 `--module esnext` 里已经被支持了（现在又增加了 `--module nodenext`），但 `es2022` 是支持该功能的首个稳定版本。

更多详情，请参考 [PR](#)。

23.5 [在条件类型上消除尾递归](#)

当 TypeScript 检测到了以下情况时通常需要优雅地失败，比如无限递归、极其耗时以至影响编辑器使用体验的类型展开操作。因此，TypeScript 会使用试探式的方法来确保它在试图拆分一个无限层级的类型时或操作将生成大量中间结果的类型时不会偏离轨道。

v4.5 => 模版字符串类型作为判别式属性

```
type InfiniteBox<T> = { item: InfiniteBox<T> };
```

```
type Unpack<T> = T extends { item: infer U } ? Unpack<U> : T;
```

```
// error: Type instantiation is excessively deep and possibly infinite.
```

```
type Test = Unpack<InfiniteBox<number>>;
```

上例是有意写成简单且没用的类型，但是存在大量有用的类型恰巧会触发试探。 作为示例，下面的 TrimLeft 类型会从字符串类型的开头删除空白。 若给定一个在开头位置有一个空格的字符串类型，它会直接将空格后面的字符串再传入 TrimLeft。

```
type TrimLeft<T extends string> = T extends ` ${infer Rest}`
  ? TrimLeft<Rest>
  : T;
```

```
// Test = "hello" | "world"
```

```
type Test = TrimLeft<" hello" | " world">;
```

这个类型也许有用，但如果字符串起始位置有 50 个空格，就会产生错误。

```
type TrimLeft<T extends string> = T extends ` ${infer Rest}`
  ? TrimLeft<Rest>
  : T;
```

```
// error: Type instantiation is excessively deep and possibly infinite.
```

```
type Test = TrimLeft<"                                oops">;
```

这很讨厌，因为这种类型在表示字符串操作时很有用 - 例如，URL 路由解析器。 更差的是，越有用的类型越会创建更多的实例化类型，结果就是对输入参数会有限制。

但也有一个可取之处：TrimLeft 在一个分支中使用了尾递归的方式编写。 当它再次调用自己时，是直接返回了结果并且不存在后续操作。 由于这些类型不需要创建中间结果，因此可以被更快地实现并且可以避免触发 TypeScript 内置的类型递归试探。

这就是 TypeScript 4.5 在条件类型上删除尾递归的原因。 只要是条件类型的某个分支为另一个条件类型，TypeScript 就不会去生成中间类型。 虽说仍然会进行一些试探来确保类型没有偏离方向，但已无伤大雅。

注意，下面的类型不会被优化，因为它使用了包含条件类型的联合类型。

```
type GetChars<S> = S extends `${infer Char}${infer Rest}`
  ? Char | GetChars<Rest>
  : never;
```

如果你想将它改成尾递归，可以引入帮助类型来接收一个累加类型的参数，就如同尾递归函数一样。

```
type GetChars<S> = GetCharsHelper<S, never>;
type GetCharsHelper<S, Acc> = S extends `${infer Char}${infer Rest}`
  ? GetCharsHelper<Rest, Char | Acc>
  : Acc;
```

更多详情，请参考 [PR](#)。

23.6 禁用导入省略

在某些情况下，TypeScript 无法检测导入是否被使用。 例如，考虑下面的代码：

```
import { Animal } from "./animal.js";
```

```
eval("console.log(new Animal().isDangerous())");
```

默认情况下，TypeScript 会删除上面的导入语句，因为它看上去没有被使用。 在

TypeScript 4.5 里，你可以启用新的标记 `preserveValueImports` 来阻止

TypeScript 从生成的 JavaScript 代码里删除导入的值。 虽说应该使用 `eval` 的理由不多，但在 Svelte 框架里有类似的情况：

```
<!-- A .svelte File -->
<script>
  import { someFunc } from "./some-module.js";
</script>
```

```
<button on:click="{someFunc}">Click me!</button>
```

同样在 Vue.js 中，使用 `<script setup>` 功能：

```
<!-- A .vue File -->
<script setup>
  import { someFunc } from "./some-module.js";
</script>
```

```
<button @click="someFunc">Click me!</button>
```

这些框架会根据 `<script>` 标签外的标记来生成代码，但 TypeScript 仅仅会考虑 `<script>` 标签内的代码。 也就是说 TypeScript 会自动删除对 `someFunc` 的导入，因此上面的代码无法运行！ 使用 TypeScript 4.5，你可以通过 `preserveValueImports` 来避免发生这种情况。

当该标记和 `--isolatedModules`` 一起使用时有个额外要求：导入的类型必须被标记为 `type-only`，因为编译器一次处理一个文件，无法知道是否导入了未被使用的值，或是导入了必须要被删除的类型以防运行时崩溃。

```
// Which of these is a value that should be preserved? tsc knows, but
`ts.transpileModule`,
// ts-loader, esbuild, etc. don't, so `isolatedModules` gives an
error.
```

```
import { someFunc, BaseType } from "./some-module.js";
//                                ^^^^^^^^^
```

```
// Error: 'BaseType' is a type and must be imported using a type-only
import
```

```
// when 'preserveValueImports' and 'isolatedModules' are both enabled.
这催生了另一个 TypeScript 4.5 的功能，导入语句中的 type 修饰符，它尤其重要。
```

更多详情，请参考 [PR](#)。

23.7 在导入名称前使用 type 修饰符

上面提到，`preserveValueImports` 和 `isolatedModules` 结合使用时有额外的要求，这是为了让构建工具能够明确知道是否可以省略导入语句。

```
// Which of these is a value that should be preserved? tsc knows, but
`ts.transpileModule`,
// ts-loader, esbuild, etc. don't, so `isolatedModules` issues an
error.
```

```
import { someFunc, BaseType } from "./some-module.js";
//                                ^^^^^^^^^
```

```
// Error: 'BaseType' is a type and must be imported using a type-only
import
```

```
// when 'preserveValueImports' and 'isolatedModules' are both enabled.
```

24 v4.6

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.6.html>

24.1 允许在构造函数中的 super() 调用之前插入代码

在 JavaScript 的类中，在引用 this 之前必须先调用 super()。在 TypeScript 中同样有这个限制，只不过在检查时过于严格。在之前版本的 TypeScript 中，如果类中存在属性初始化器，那么在构造函数里，在 super() 调用之前不允许出现任何其它代码。

```
class Base {
  // ...
}

class Derived extends Base {
  someProperty = true;

  constructor() {
    // 错误!
    // 必须先调用 'super()' 因为需要初始化 'someProperty'。
    doSomeStuff();
    super();
  }
}
```

这样做是因为程序实现起来容易，但这样做也会拒绝很多合法的代码。TypeScript 4.6 放宽了限制，它允许在 super() 之前出现其它代码，与此同时仍然会检查在引用 this 之前顶层的 super() 已经被调用。

感谢 [Joshua Goldberg](#) 的 PR。

24.2 基于控制流来分析解构的可辨识联合类型

TypeScript 可以根据判别式属性来细化类型。例如，在下面的代码中，TypeScript 能够在检查 kind 的类型后细化 action 的类型。

```
type Action =
  | { kind: "NumberContents", payload: number }
  | { kind: "StringContents", payload: string };

function processAction(action: Action) {
  if (action.kind === "NumberContents") {
    // `action.payload` is a number here.
    let num = action.payload * 2
    // ...
  }
  else if (action.kind === "StringContents") {
    // `action.payload` is a string here.
    const str = action.payload.trim();
    // ...
  }
}
```

这样就可以使用持有不同数据的对象，但通过共同的字段来区分它们。这在 TypeScript 是很常见的；然而，根据个人的喜好，你可能想对上例中的 kind 和 payload 进行解构。就像下面这样：

分不同类的实例。因此，TypeScript 能够在每次检查中细化 other 类型，直到细化为 Person 类型。

感谢来自 Bloomberg 的朋友提交的 PR: [Ashley Claymore](#), [Titian Cernicova-Dragomir](#), [Kubilay Kahveci](#), 和 [Rob Palmer](#)!

23.9 导入断言

TypeScript 4.5 支持了 ECMAScript Proposal 中的 导入断言。该语法会被运行时所使用来检查导入是否为期望的格式。

```
import obj from "./something.json" assert { type: "json" };
TypeScript 不会检查这些断言，因为它们依赖于宿主环境。TypeScript 会保留原样，稍后让浏览器或者运行时来处理它们（也可能会出错）。
// TypeScript 允许
// 但浏览器可能不允许
import obj from "./something.json" assert {
  type: "fluffy bunny"
};
动态的 import() 调用可以通过第二个参数来使用导入断言。
const obj = await import("./something.json", {
  assert: { type: "json" },
});
第二个参数的类型为 ImportCallOptions，并且目前它只接受一个 assert 属性。
感谢 Wenlu Wang 实现了 这个功能!
```

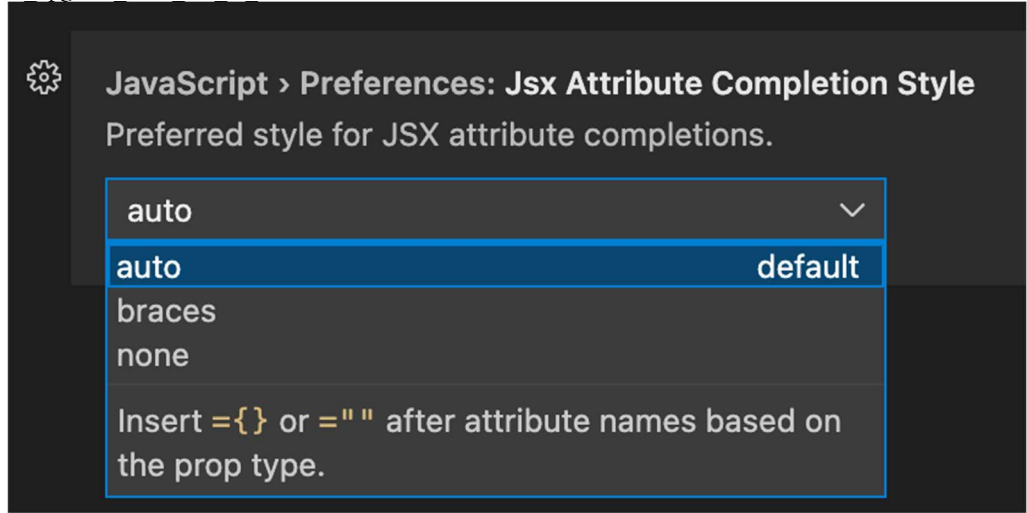
23.10 使用 realPathSync.native 获得更快的加载速度

TypeScript 在所有操作系统上使用了 Node.js realPathSync 函数的系统原生实现。以前，这个函数只在 Linux 上使用了，但在 TypeScript 4.5 中，在大小写不敏感的操作系统上，如 Windows 和 MacOS，也被采用了。对于一些代码库来讲这个改动会提升 5 ~ 13% 的加载速度（和操作系统有关）。更多详情请参考 PR。

23.11 JSX Attributes 的代码片段自动补全

TypeScript 4.5 为 JSX 属性提供了代码片段自动补全功能。当在 JSX 标签上输入属性时，TypeScript 已经能够提供提供建议；但对于代码片段自动补全来讲，它们会删除部分已经输入的字符来添加一个初始化器并将光标放到正确的位置。

TypeScript 通常会使用属性的类型来判断插入哪种初始化器，但你可以在 Visual Studio Code 中自定义该行为。

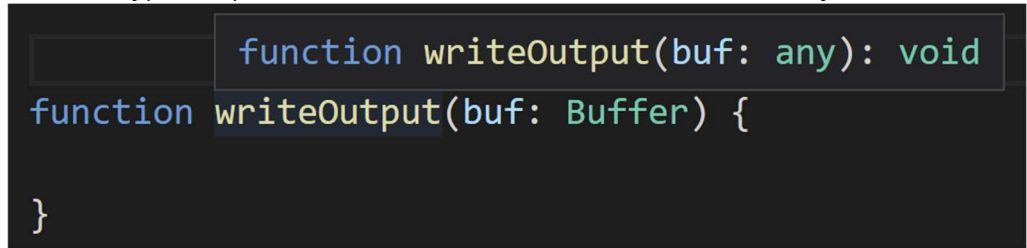


注意，该功能只在新版本的 Visual Studio Code 中支持，因此你可能需要使用 Insiders 版本。更多详情，请参考 [PR](#)。

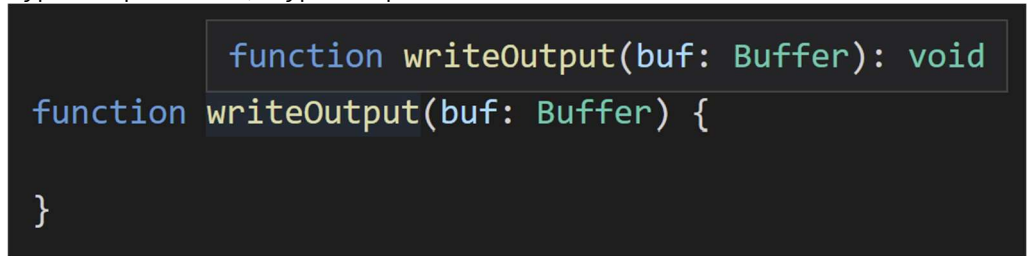
23.12 为未解决类型提供更好的编辑器支持

在某些情况下，编辑器会使用一个轻量级的“部分”语义模式 - 比如编辑器正在等待加载完整的工程，又或者是 [GitHub 的基于 web 的编辑器](#)。

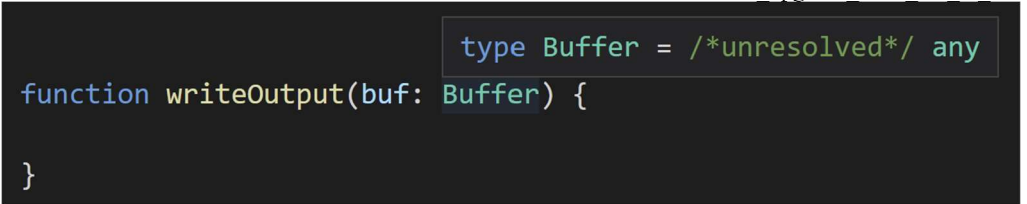
在旧版本 TypeScript 中，如果语言服务无法找到一个类型，它会输出 `any`。



上例中，没有找到 `Buffer`，因此 TypeScript 在 *quick info* 里显示了 `any`。在 TypeScript 4.5 中，TypeScript 会尽可能保留你编写的代码。



然而，当你将鼠标停在 `Buffer` 上时，你会看到 TypeScript 无法找到 `Buffer` 的提示。



总之，在 TypeScript 还没有读取整个工程的时候，它提供了更加平滑的体验。注意，在它正常情况下，当无法找到某个类型时总会产生错误。更多详情，请参考 [PR](#)。

24.6 [--target es2022](#)

TypeScript 的 `--target` 编译选项现在支持使用 `es2022`。这意味着像类字段这样的特性能够稳定地在输出结果中保留。这也意味着像 [Arrays 的上 `at\(\)` 和 `Object.hasOwn` 方法](#) 或者 [new Error 时的 `cause` 选项](#) 可以通过设置新的 `--target` 或者 `--lib` `es2022` 来使用。

感谢 [Kagami Sascha Rosylight \(saschanaz\)](#) 的[实现](#)。

24.7 [删除 `react-jsx` 中不必要的参数](#)

在以前, 当使用 `--jsx react-jsx` 来编译如下的代码时

```
export const el = <div>foo</div>;
```

TypeScript 会生成如下的 JavaScript 代码:

```
import { jsx as _jsx } from "react/jsx-runtime";
export const el = _jsx("div", { children: "foo" }, void 0);
```

末尾的 `void 0` 参数是没用的, 删掉它会减小打包的体积。

感谢 <https://github.com/a-tarasyuk> 的 [PR](#), TypeScript 4.6 会删除 `void 0` 参数。

24.8 [JSDoc 命名建议](#)

在 JSDoc 里, 你可以用 `@param` 标签来文档化参数。

```
/**
 * @param x The first operand
 * @param y The second operand
 */
```

```
function add(x, y) {
  return x + y;
}
```

但是, 如果这些注释已经过时了会发生什么? 就比如, 我们将 `x` 和 `y` 重命名为 `a` 和 `b`?

```
/**
 * @param x {number} The first operand
 * @param y {number} The second operand
 */
function add(a, b) {
  return a + b;
}
```

在之前 TypeScript 仅会在对 JavaScript 文件执行类型检查时报告这个问题 - 通过使用 `checkJs` 选项, 或者在文件顶端添加 `// @ts-check` 注释。

现在, 你能够在编译器中的 TypeScript 文件上看到类似的提示! TypeScript 现在会给出建议, 如果函数签名中的参数名与 JSDoc 中的参数名不一致。

```
type Action =
  | { kind: "NumberContents", payload: number }
  | { kind: "StringContents", payload: string };
```

```
function processAction(action: Action) {
  const { kind, payload } = action;
  if (kind === "NumberContents") {
    let num = payload * 2
    // ...
  }
  else if (kind === "StringContents") {
    const str = payload.trim();
    // ...
  }
}
```

此前, TypeScript 会报错 - 当 `kind` 和 `payload` 是由同一个对象解构为变量时, 它们会被独立对待。

在 TypeScript 4.6 中可以正常工作!

当解构独立的属性为 `const` 声明, 或当解构参数到变量且没有重新赋值时, TypeScript 会检查被解构的类型是否为可辨识联合。如果是的话, TypeScript 就能够根据类型检查来细化变量的类型。因此上例中, 通过检查 `kind` 的类型可以细化 `payload` 的类型。

更多详情请查看 [PR](#)。

24.3 [改进的递归深度检查](#)

TypeScript 要面对一些有趣的挑战, 因为它是构建在结构化类型系统之上, 同时又支持了泛型。

在结构化类型系统中, 对象类型的兼容性是由对象包含的成员决定的。

```
interface Source {
  prop: string;
}
```

```
interface Target {
  prop: number;
}
```

```
function check(source: Source, target: Target) {
  target = source;
  // error!
  // Type 'Source' is not assignable to type 'Target'.
  //   Types of property 'prop' are incompatible.
  //     Type 'string' is not assignable to type 'number'.
}
```

`Source` 与 `Target` 的兼容性取决于它们的属性是否可以执行赋值操作。此例中是指 `prop` 属性。

当引入了泛型后, 有一些难题需要解决。例如, 下例中的 `Source<string>` 是否可以赋值给 `Target<number>`?

```
interface Source<T> {
  prop: Source<Source<T>>;
}
```



```
interface Target<T> {
  prop: Target<Target<T>>;
}

function check(source: Source<string>, target: Target<number>) {
  target = source;
}

// 要想回答这个问题，TypeScript 需要检查 prop 的类型是否兼容。这又要回答另一个问题：Source<Source<string>> 是否能够赋值给 Target<Target<number>>？要想回答这个问题，TypeScript 需要检查 prop 的类型是否与那些类型兼容，结果就是还要检查 Source<Source<Source<string>>> 是否能够赋值给 Target<Target<Target<number>>>？继续发展下去，就会注意到类型会进行无限展开。
// TypeScript 使用了启发式的算法 - 当一个类型达到特定的检查深度时，它表现出了将会进行无限展开，那么就认为它可能是兼容的。通常情况下这是没问题的，但是也可能出现漏报的情况。

interface Foo<T> {
  prop: T;
}

declare let x: Foo<Foo<Foo<Foo<Foo<Foo<string>>>>>>>>;
declare let y: Foo<Foo<Foo<Foo<Foo<string>>>>>>>>;
```

x = y;

通过人眼观察我们知道上例中的 x 和 y 是不兼容的。虽然类型的嵌套层次很深，但人家就是这样声明的。启发式算法要处理的是在探测类型过程中生成的深层次嵌套类型，而非程序员明确手写出的类型。

TypeScript 4.6 现在能够区分出这类情况，并且对上例进行正确的错误提示。此外，由于不再担心会对明确书写的类型进行误报，TypeScript 能够更容易地判断类型的无限展开，并且降低了类型兼容性检查的成本。因此，像 DefinitelyTyped 上的 redux-immutable、react-lazylog 和 yup 代码库，对它们的类型检查时间降低了 50%。你可能已经体验过这个改动了，因为它被挑选合并到了 TypeScript 4.5.3 中，但它仍然是 TypeScript 4.6 中值得关注的一个特性。更多详情请阅读 [PR](#)。

24.4 [索引访问类型推断改进](#)

TypeScript 现在能够正确地推断通过索引访问到另一个映射对象类型的类型。

```
interface TypeMap {
  "number": number;
  "string": string;
  "boolean": boolean;
}

type UnionRecord<P extends keyof TypeMap> = { [K in P]:
  {
    kind: K;
    v: TypeMap[K];
    f: (p: TypeMap[K]) => void;
  }
}
```

```
}[P];

function processRecord<K extends keyof TypeMap>(record: UnionRecord<K>) {
  record.f(record.v);
}

// 这个调用之前是有问题的，但现在没有问题
processRecord({
  kind: "string",
  v: "hello!",

  // 'val' 之前会隐式地获得类型 'string | number | boolean',
  // 但现在会正确地推断为类型 'string'。
  f: val => {
    console.log(val.toUpperCase());
  }
})

// 该模式已经被支持了并允许 TypeScript 判断 record.f(record.v) 调用是合理的，但是在以前，processRecord 调用中对 val 的类型推断并不好。
// TypeScript 4.6 改进了这个情况，因此在启用 processRecord 时不再需要使用类型断言。
// 更多详情请阅读 PR。

24.5 对因变参数的控制流分析
// 函数签名可以声明为剩余参数且其类型可以为可辨识联合元组类型。
function func(...args: ["str", string] | ["num", number]) {
  // ...
}

// 这意味着 func 的实际参数完全依赖于第一个实际参数。若第一个参数为字符串 "str" 时，则第二个参数为 string 类型。若第一个参数为字符串 "num" 时，则第二个参数为 number 类型。
// 像这样 TypeScript 是由签名来推断函数类型时，TypeScript 能够根据依赖的参数来细化类型。
type Func = (...args: ["a", number] | ["b", string]) => void;

const f1: Func = (kind, payload) => {
  if (kind === "a") {
    payload.toFixed(); // 'payload' narrowed to 'number'
  }
  if (kind === "b") {
    payload.toUpperCase(); // 'payload' narrowed to 'string'
  }
};

f1("a", 42);
f1("b", "hello");
// 更多详情请阅读 PR。
```

ts upgrade from 2.2 to 5.3

当一个 .ts 文件被编译为 ESM 时, ECMAScript import / export 语句在生成的 .js 文件中原样输出; 当一个 .ts 文件被编译为 CommonJS 模块时, 则会产生与使用了 --module commonjs 选项一致的输出结果。

这也意味着 ESM 和 CJS 模块中的 .ts 文件路径解析是不同的。例如, 现在有如下的代码:

```
// ./foo.ts
export function helper() {
  // ...
}

// ./bar.ts
import { helper } from "./foo"; // only works in CJS
```

helper();

这段代码在 CommonJS 模块里没问题, 但在 ESM 里会出错, 因为相对导入需要使用完整的扩展名。因此, 我们不得不重写代码并使用 foo.ts 输出文件的扩展名, bar.ts 必须从 ./foo.js 导入。

```
// ./bar.ts
import { helper } from "./foo.js"; // works in ESM & CJS
```

helper();

初看可能感觉很繁琐, 但 TypeScript 的自动导入工具以及路径补全工具会有所帮助。此外还需要注意的是该行为同样适用于 .d.ts 文件。当 TypeScript 在一个 package 里找到了 .d.ts 文件, 它会基于这个 package 来解析 .d.ts 文件。

25.1.2 新的文件扩展名

package.json 文件里的 type 字段让我们可以继续使用 .ts 和 .js 文件扩展名; 但你可能偶尔需要编写与 type 设置不符的文件, 或者更喜欢明确地表达意图。

为此, Node.js 支持了两个文件扩展名: .mjs 和 .cjs。 .mjs 文件总是使用 ESM, 而 .cjs 则总是使用 CommonJS 模块, 它们分别会生成 .mjs 和 .cjs 文件。正因此, TypeScript 也支持了两个新的文件扩展名: .mts 和 .cts。当 TypeScript 生成 JavaScript 文件时, 将生成 .mjs 和 .cjs。

TypeScript 还支持了两个新的声明文件扩展名: .d.mts 和 .d.cts。当 TypeScript 为 .mts 和 .cts 生成声明文件时, 对应的扩展名为 .d.mts 和 .d.cts。这些扩展名的使用完全是可选的, 但通常是有帮助的, 不论它们是不是你工作流中的一部分。

25.1.3 CommonJS 互操作性

Node.js 允许 ESM 导入 CommonJS 模块, 就如同它们是带有默认导出的 ESM。

```
// ./foo.cts
export function helper() {
  console.log("hello world!");
}
```

```
// ./bar.mts
import foo from "./foo.cjs";
```

```
// prints "hello world!"
```

foo.helper();

在某些情况下, Node.js 会综合和合成 CommonJS 模块里的命名导出, 这提供了便利。此时, ESM 既可以使用“命名空间风格”的导入 (例如, import * as foo from "..."), 也可以使用命名导入 (例如, import { helper } from "...")。

```
/**
 * Returns an object from a path string - the opposite of format().
 *
 * any
 *
 * JSDoc '@param' tag has name 'pathString', but there is no
 * parameter with that name. ts(8024)
 *
 * Quick Fix... (Ctrl+.)
 *
 * @param pathString path to evaluate.
 */
parse(p: string): ParsedPath;
/**
 * Returns a path string from an object - the opposite of parse().
 *
 * @param pathString path to evaluate.
 */
format(pP: FormatInputPathObject): string;
```

该改动是由 [Alexander Tarasyuk](#) 提供的!

24.9 JavaScript 中更多的语法和绑定错误提示

TypeScript 将更多的语法和绑定错误检查应用到了 JavaScript 文件上。如果你在 Visual Studio 或 Visual Studio Code 这样的编辑器中打开 JavaScript 文件时就会看到这些新的错误提示, 或者当你使用 TypeScript 编译器来处理 JavaScript 文件时 - 即便你没有打开 checkJs 或者添加 // @ts-check 注释。

做为例子, 如果在 JavaScript 文件中的同一个作用域中有两个同名的 const 声明, 那么 TypeScript 会报告一个错误。

```
const foo = 1234;
// ~~~
// error: Cannot redeclare block-scoped variable 'foo'.

// ...
```

```
const foo = 5678;
// ~~~
// error: Cannot redeclare block-scoped variable 'foo'.
```

另外一个例子, TypeScript 会报告修饰符是否被正确地使用了。

```
function container() {
  export function foo() {
    // ~~~~~
  }
}
```

这些检查可以通过在文件顶端添加 // @ts-nocheck 注释来禁用, 但是我们很想听听在大家的 JavaScript 工作流中使用该特性的反馈。你可以在 Visual Studio Code 安装 [TypeScript 和 JavaScript Nightly 扩展](#) 来提前体验, 并阅读 [PR1](#) 和 [PR1](#)。

24.10 TypeScript Trace 分析器

有人偶尔会遇到创建和比较类型时很耗时的情况。TypeScript 提供了一个 --generateTrace 选项来帮助识别耗时的类型, 或者帮助诊断 TypeScript 编译器中的问

题。虽说由 `--generateTrace` 生成的信息是非常有帮助的（尤其是在 TypeScript 4.6 的改进后），但是阅读这些 `trace` 信息是比较难的。

近期，我们发布了 [@typescript/analyze-trace](https://github.com/microsoft/TypeScript/blob/master/src/compiler/trace.ts) 工具来帮助阅读这些信息。虽说我们不认为每个人都需要使用 `analyze-trace`，但是我们认为它会为遇到了 [TypeScript 构建性能](#) 问题的团队提供帮助。

更多详情请查看 [repo](#)。

25 v4.7

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.7.html>

25.1 Node.js 对 ECMAScript Module 的支持

在过去的几年中，Node.js 为支持 ECMAScript 模块（ESM）而做了一些工作。这是一项有难度的工作，因为 Node.js 生态圈是基于 CommonJS（CJS）模块系统构建的，而非 ESM。支持两者之间的互操作带来了巨大挑战，有大量的特性需要考虑；然而，在 Node.js 12 及以上版本中，已经提供了对 ESM 的大部分支持。在 TypeScript 4.5 期间的一个 `nightly` 版本中支持了在 Node.js 里使用 ESM 以获得用户反馈，同时让代码库作者们有时间为此提前作准备。

TypeScript 4.7 正式地支持了该功能，它添加了两个新的 `module` 选项：`node16` 和 `nodenext`。

```
{
  "compilerOptions": {
    "module": "node16",
  }
}
```

这些新模式带来了一些高级特征，下面将一一介绍。

25.1.1 package.json 里的 `type` 字段和新的文件扩展名

Node.js 在 [package.json](#) 中支持了一个新的设置，叫做 `type`。`"type"` 可以被设置为 `"module"` 或者 `"commonjs"`。

```
{
  "name": "my-package",
  "type": "module",

  "//": "...",
  "dependencies": {
  }
}
```

这些设置会控制 `.js` 文件是作为 ESM 进行解析还是作为 CommonJS 模块进行解析，若没有设置，则默认值为 CommonJS。当一个文件被当做 ESM 模块进行解析时，会使用如下与 CommonJS 模块不同的规则：

- 允许使用 `import / export` 语句
- 允许使用顶层的 `await`
- 相对路径导入必须提供完整的扩展名（需要使用 `import './foo.js'` 而非 `import './foo'`）
- 解析 `node_modules` 里的依赖可能不同
- 不允许直接使用像 `require` 和 `module` 这样的全局值
- 需要使用特殊的规则来导入 CommonJS 模块

我们回头会介绍其中一部分。

为了让 TypeScript 融入该系统，`.ts` 和 `.tsx` 文件现在也以同样的方式工作。当 TypeScript 遇到 `.ts`、`.tsx`、`.js` 或 `.jsx` 文件时，它会向上查找 `package.json` 来确定该文件是否使用了 ESM，然后再以此决定：

- 如何查找该文件所导入的其它模块
- 当需要产生输出的时，如何转换该文件

ts_upgrade_from_2.2_to_5.3

```
class C {
  [key]: string;

  constructor(str: string) {
    // oops, forgot to set 'this[key]'
  }

  screamString() {
    return this[key].toUpperCase();
  }
}
```

在 TypeScript 4.7 里, `--strictPropertyInitialization` 会提示错误说 `[key]` 属性在构造函数里没有被赋值。

感谢 [Oleksandr Tarasiuk](#) 提交的[代码](#)。

25.4 改进对象和方法里的函数类型推断

TypeScript 4.7 可以对数组和对象里的函数进行更精细的类型推断。它们可以像普通参数那样将类型从左向右进行传递。

```
declare function f<T>(arg: {
  produce: (n: string) => T,
  consume: (x: T) => void }
): void;
```

```
// Works
f({
  produce: () => "hello",
  consume: x => x.toLowerCase()
});
```

```
// Works
f({
  produce: (n: string) => n,
  consume: x => x.toLowerCase(),
});
```

```
// Was an error, now works.
f({
  produce: n => n,
  consume: x => x.toLowerCase(),
});
```

```
// Was an error, now works.
f({
  produce: function () { return "hello"; },
  consume: x => x.toLowerCase(),
});
```

```
// Was an error, now works.
f({
```

```
// ./foo.cts
export function helper() {
  console.log("hello world!");
}
```

```
// ./bar.mts
import { helper } from "./foo.cjs";
```

```
// prints "hello world!"
helper();
```

有时候 TypeScript 不知道命名导入是否会被综合合并, 但如果 TypeScript 能够通过确定地 CommonJS 模块导入了解到该信息, 那么就会提示错误。

关于互操作性, TypeScript 特有的注意点是如下的语法:

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

在 CommonJS 模块中, 它可以归结为 `require()` 调用, 在 ESM 里, 它会导入 [createRequire](#) 来完成同样的事情。对于像浏览器这样的平台 (不支持 `require()`) 这段代码的可移植性较差, 但对互操作性是有帮助的。你可以这样改写:

```
// ./foo.cts
export function helper() {
  console.log("hello world!");
}
```

```
// ./bar.mts
import foo = require("./foo.cjs");
```

```
foo.helper()
```

最后值得注意的是在 CommonJS 模块里导入 ESM 的唯一方法是使用动态 `import()` 调用。这也许是一个挑战, 但也是目前 Node.js 的行为。

更多详情, 请阅读[这里](#)。

25.1.4 package.json 中的 `exports`, `imports` 以及自引用

Node.js 在 `package.json` 支持了一个新的字段 `exports` 来定义入口位置。它比在 `package.json` 里定义 `"main"` 更强大, 它能控制将包里的哪些部分公开给使用者。

下例的 `package.json` 支持对 CommonJS 和 ESM 使用不同的入口位置:

```
// package.json
{
  "name": "my-package",
  "type": "module",
  "exports": {
    ".": {
      // Entry-point for `import "my-package"` in ESM
      "import": "./esm/index.js",

      // Entry-point for `require("my-package")` in CJS
      "require": "./commonjs/index.cjs",
    },
  },
}
```

```
// CJS fall-back for older versions of Node.js
```



```
"main": "./commonjs/index.cjs",
}
```

关于该特性的更多详情请阅读[这里](#)。下面我们主要关注 TypeScript 是如何支持它的。在以前 TypeScript 会先查找 "main" 字段，然后再查找其对应的声明文件。例如，如果 "main" 指向了 ./lib/index.js，TypeScript 会查找名为 ./lib/index.d.ts 的文件。代码包作者可以使用 "types" 字段来控制该行为（例如，"types": "./types/index.d.ts"）。新实现的工作方式与[导入条件](#)相似。默认地，TypeScript 使用与[导入条件](#)相同的规则 - 对于 ESM 里的 import 语句，它会查找 import 字段；对于 CommonJS 模块里的 import 语句，它会查找 require 字段。如果找到了文件，则去查找相应的声明文件。如果你想将声明文件指向其它位置，则可以添加一个 "types" 导入条件。

```
// package.json
{
  "name": "my-package",
  "type": "module",
  "exports": {
    ".": {
      // Entry-point for `import "my-package"` in ESM
      "import": {
        // Where TypeScript will look.
        "types": "./types/esm/index.d.ts",

        // Where Node.js will look.
        "default": "./esm/index.js"
      },
      // Entry-point for `require("my-package")` in CJS
      "require": {
        // Where TypeScript will look.
        "types": "./types/commonjs/index.d.cts",

        // Where Node.js will look.
        "default": "./commonjs/index.cjs"
      },
    },
  },

  // Fall-back for older versions of TypeScript
  "types": "./types/index.d.ts",

  // CJS fall-back for older versions of Node.js
  "main": "./commonjs/index.cjs"
}
```

注意，"types" 条件在 "exports" 中需要被放在开始的位置。TypeScript 也支持 package.json 里的 "imports" 字段，它与查找声明文件的工作方式类似。此外，还支持[一个包引用它自己](#)。这些特性通常不特殊设置，但是是支持的。

25.2 [设置模块检测策略](#)

在 JavaScript 中引入模块带来的一个问题是让“Script”代码和新的模块代码之间的界限变得模糊。（译者注：对于任意一段 JavaScript 代码，它的类型只能为“Script”或

“Module”两者之一，它们是 ECMAScript 语言规范中定义的术语。）模块中的 JavaScript 存在些许不同的执行方式和作用域规则，因此工具们需要确定每个文件的执行方式。例如，Node.js 要求模块入口脚本是一个 .mjs 文件，或者它有一个邻近的 package.json 文件且带有 "type": "module"。TypeScript 的规则则是如果一个文件里存在 import 或 export 语句，那么它是模块文件；反之会把 .ts 和 .js 文件当作是“Script”文件，它们存在于[全局作用域](#)。这与 Node.js 中对 package.json 的处理行为不同，因为 package.json 可以改变文件的类型；又或者是在 --jsx react-jsx 模式下一个 JSX 文件显式地导入了 JSX 工厂函数。它也与当下的期望不符，因为大多数的 TypeScript 代码是基于模块来编写的。以上就是 TypeScript 4.7 引入了 moduleDetection. moduleDetection 选项的原因。它接受三个值：

- 1. "auto", 默认值
- 2. "legacy", 行为与 TypeScript 4.6 和以前的版本相同
- 3. "force"

在 "auto" 模式下，TypeScript 不但会检测 import 和 export 语句，它还会检测：

- 若启用了 --module nodenext / --module node16，那么 package.json 里的 "type" 字段是否为 "module"，以及
- 若启用了 --jsx react-jsx，那么当前文件是否为 JSX 文件。

在这些情况下，我们想将每个文件都当作模块文件。"force" 选项能够保证每个非声明文件都被当成模块文件，不论 module，moduleResolution 和 jsx 是如何设置的。与此同时，使用 "legacy" 选项会回退到以前的行为，仅通过检测 import 和 export 语句来决定是否为模块文件。

更多详情请阅读 [PR](#)。

25.3 [\[\] 语法元素访问的控制流分析](#)

在 TypeScript 4.7 里，当索引键值是字面量类型和 unique symbol 类型时会细化访问元素的类型。例如，有如下代码：

```
const key = Symbol();

const numberOrString = Math.random() < 0.5 ? 42 : "hello";

const obj = {
  [key]: numberOrString,
};

if (typeof obj[key] === "string") {
  let str = obj[key].toUpperCase();
}

在之前，TypeScript 不会处理涉及 obj[key] 的类型守卫，也就不知道 obj[key] 的类型是 string。它会将 obj[key] 当作 string | number 类型，因此调用 toUpperCase() 会产生错误。TypeScript 4.7 能够知道 obj[key] 的类型为 string。这意味着在 --strictPropertyInitialization 模式下，TypeScript 能够正确地检查计算属性是否被初始化。
// 'key' has type 'unique symbol'
const key = Symbol();
```


另一个原因则有关精度和速度。TypeScript 已经在尝试推断类型参数的变型并做为一项优化。这样做可以快速对大型的结构化类型进行类型检查。提前计算变型省去了深入结构内部进行兼容性检查的步骤，仅比较类型参数相比于一次又一次地比较完整的类型结构会快得多。但经常也会出现这个计算十分耗时，并且在计算时产生了环，从而无法得到准确的变型关系。

```
type Foo<T> = {
  x: T;
  f: Bar<T>;
}

type Bar<U> = (x: Baz<U[]>) => void;
```

```
type Baz<V> = {
  value: Foo<V[]>;
}

declare let foo1: Foo<unknown>;
declare let foo2: Foo<string>;
```

```
foo1 = foo2; // Should be an error but isn't ✕
foo2 = foo1; // Error - correct ✓
```

提供明确的类型注解能够加快对环状类型的解析速度，有利于提高准确度。例如，将上例的 T 设置为逆变可以帮助阻止有问题的赋值运算。

```
- type Foo<T> = {
+ type Foo<in out T> = {
  x: T;
  f: Bar<T>;
}
```

我们并不推荐为所有的类型参数都添加变型注解：例如，我们是能够（但不推荐）将变型设置为更严格的关系（即便实际上不需要），因此 TypeScript 不会阻止你将类型参数设置为不变，就算它们实际上是协变的、逆变的或者是分离的。因此，如果你选择添加明确的变型标记，我们推荐要经过深思熟虑后准确地使用它们。但如果你操作的是深层次的递归类型，尤其是作为代码库作者，那么你可能会对使用这些注解来让用户获利感兴趣。这些注解能够帮助提高准确性和类型检查速度，甚至可以增强代码编辑的体验。可以通过实验来确定变型计算是否为类型检查时间的瓶颈，例如使用像 [analyze-trace](#) 这样的工具。更多详情请阅读[这里](#)。

25.8 使用 [moduleSuffixes](#) 自定义解析策略

TypeScript 4.7 支持了 moduleSuffixes 选项来自定义模块说明符的查找方式。

```
{
  "compilerOptions": {
    "moduleSuffixes": [".ios", ".native", ""]
  }
}
```

对于上述配置，如果有如下的导入语句：

```
import * as foo from "./foo";
```

它会尝试查找文件 ./foo.ios.ts, ./foo.native.ts 最后是 ./foo.ts。

```
produce() { return "hello" },
consume: x => x.toLowerCase(),
});
```

之所以有些类型推断之前会失败是因为，若要知道 produce 函数的类型则需要在找到合适的类型 T 之前间接地获得 arg 的类型。（译者注：这些之前失败的情况均是需要进行按上下文文件归类的场景，即需要先知道 arg 的类型，才能确定 produce 的类型；如果不需要执行按上下文归类就能确定 produce 的类型则没有问题。）TypeScript 现在会收集与泛型参数 T 的类型推断相关的函数，然后进行惰性地类型推断。更多详情请阅读[这里](#)。

25.5 实例化表达式

我们偶尔可能会觉得某个函数过于通用了。例如有一个 makeBox 函数。

```
interface Box<T> {
  value: T;
}
```

```
function makeBox<T>(value: T) {
  return { value };
}
```

假如我们想要定义一组更具体的可以收纳扳子和锤子的 Box 函数。为此，我们将 makeBox 函数包装进另一个函数，或者明确地定义一个 makeBox 的类型别名。

```
function makeHammerBox(hammer: Hammer) {
  return makeBox(hammer);
}
```

// 或者

```
const makeWrenchBox: (wrench: Wrench) => Box<Wrench> = makeBox;
这样可以工作，但有些浪费且笨重。理想情况下，我们可以在替换泛型参数的时候直接声明 makeBox 的别名。
TypeScript 4.7 支持了该特性！我们现在可以直接为函数和构造函数传入类型参数。
const makeHammerBox = makeBox<Hammer>;
const makeWrenchBox = makeBox<Wrench>;
这样我们可以让 makeBox 只接受更具体的类型并拒绝其它类型。
const makeStringBox = makeBox<string>;
```

```
// TypeScript 会提示错误
makeStringBox(42);
这对构造函数也生效，例如 Array, Map 和 Set。
// 类型为 `new () => Map<string, Error>`
const ErrorMap = Map<string, Error>;
```

```
// 类型为 `Map<string, Error>`
const errorMap = new ErrorMap();
当函数或构造函数接收了一个类型参数，它会生成一个新的类型并保持所有签名使用了兼容的类型参数列表，将形式类型参数替换成给定的实际类型参数。其它种类的签名会被丢弃，因为 TypeScript 认为它们不会被使用到。
更多详情请阅读这里。
```

25.6 [infer 类型参数上的 extends 约束](#)

有条件类型有点儿像一个进阶功能。它允许我们匹配并依据类型结构进行推断，然后作出某种决定。例如，编写一个有条件类型，它返回元组类型的第一个元素如果它类似 `string` 类型的话。

```
type FirstIfString<T> =
  T extends [infer S, ...unknown[]]
    ? S extends string ? S : never
    : never;

// string
type A = FirstIfString<[string, number, number]>;

// "hello"
type B = FirstIfString<["hello", number, number]>;

// "hello" | "world"
type C = FirstIfString<["hello" | "world", boolean]>;

// never
type D = FirstIfString<[boolean, number, string]>;
FirstIfString 匹配至少有一个元素的元组类型，将元组第一个元素的类型提取到 S。然后检查 S 与 string 是否兼容，如果是就返回它。
可以注意到我们必须使用两个有条件类型来实现它。我们也可以这样定义 FirstIfString：
type FirstIfString<T> =
  T extends [string, ...unknown[]]
    // Grab the first type out of `T`
    ? T[0]
    : never;
```

它可以工作但要更多的“手动”操作且不够形象。我们不是进行类型模式匹配并给首个元素命名，而是使用 `T[0]` 来提取 `T` 的第 0 个元素。如果我们处理的是比元组类型复杂得多的类型就会变得棘手，因此 `infer` 可以让事情变得简单。使用嵌套的条件来推断类型再去匹配推断出的类型是很常见的。为了省去那一层嵌套，TypeScript 4.7 允许在 `infer` 上应用约束。

```
type FirstIfString<T> =
  T extends [infer S extends string, ...unknown[]]
    ? S
    : never;
```

通过这种方式，在 TypeScript 去匹配 `S` 时，它也会保证 `S` 是 `string` 类型。如果 `S` 不是 `string` 就是进入到 `false` 分支，此例中为 `never`。更多详情请阅读[这里](#)。

25.7 [可选的类型参数变型注释](#)

先看一下如下的类型。

```
interface Animal {
  animalStuff: any;
}

interface Dog extends Animal {
  dogStuff: any;
```

```
}

// ...

type Getter<T> = () => T;

type Setter<T> = (value: T) => void;
假设有两个不同的 Getter 实例。要想知道这两个 Getter 实例是否可以相互替换完全依赖于类型 T。例如要知道 Getter<Dog> → Getter<Animal> 是否允许，则需要检查 Dog → Animal 是否允许。因为对 T 与 Getter<T> 的判断是相同“方向”的，我们称 Getter 是协变的。相反的，判断 Setter<Dog> → Setter<Animal> 是否允许，需要检查 Animal → Dog 是否允许。这种在方向上的“翻转”有点像数学里判断 $-x < -y$ 等同于判断 $y < x$。当我们需要像这样翻转方向来比较 T 时，我们称 Setter 对于 T 是逆变的。在 TypeScript 4.7 里，我们可以明确地声明类型参数上的变型关系。因此，现在如果想在 Getter 上明确地声明对于 T 的协变关系则可以使用 out 修饰符。
type Getter<out T> = () => T;
相似的，如果想要明确地声明 Setter 对于 T 是逆变关系则可以指定 in 修饰符。
type Setter<in T> = (value: T) => void;
使用 out 和 in 的原因是类型参数的变型关系依赖于它们被用在输出的位置还是输入的位置。若不思考变型关系，你也可以只关注 T 是被用在输出还是输入位置上。当然也有同时使用 out 和 in 的时候。
interface State<in out T> {
  get: () => T;
  set: (value: T) => void;
}
当 T 被同时用在输入和输出的位置上时就成了不变关系。两个不同的 State<T> 不允许互换使用，除非两者的 T 是相同的。换句话说，State<Dog> 和 State<Animal> 不能互换使用。
从技术上讲，在纯粹的结构化类型系统里，类型参数和它们的变型关系不太重要 - 我们只需要将类型参数替换为实际类型，然后再比较相匹配的类型成员之间是否兼容。那么如果 TypeScript 使用结构化类型系统为什么我们要在意类型参数的变型呢？还有为什么我们会想要为它们添加类型注释呢？
其中一个原因是可以让读者能够明确地知道类型参数是如何被使用的。对于十分复杂的类型来讲，可能很难确定一个类型参数是用于输入或者输出又或者两者兼有。如果我们忘了说明类型参数是如何被使用的，TypeScript 也会提示我们。举个例子，如果忘了在 State 上添加 in 和 out 就会产生错误。
interface State<out T> {
  // ~~~~~
  // error!
  // Type 'State<sub-T>' is not assignable to type 'State<super-T>'
  // as implied by variance annotation.
  // Types of property 'set' are incompatible.
  // Type '(value: sub-T) => void' is not assignable to type
  '(value: super-T) => void'.
  // Types of parameters 'value' and 'value' are incompatible.
  // Type 'super-T' is not assignable to type 'sub-T'.
  get: () => T;
  set: (value: T) => void;
```

26 v4.8

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.8.html>

26.1 改进的交叉类型化简、联合类型兼容性以及类型细化

TypeScript 4.8 为 `--strictNullChecks` 带来了一系列修正和改进。这些变化会影响交叉类型和联合类型的工作方式，也作用于 TypeScript 的类型细化。

例如，`unknown` 与 `{}` | `null` | `undefined` 类型神似，因为它接受 `null`，`undefined` 以及任何其它类型。TypeScript 现在能够识别出这种情况，允许将 `unknown` 赋值给 `{}` | `null` | `undefined`。

译者注：除 `null` 和 `undefined` 类型外，其它任何类型都可以赋值给 `{}` 类型。

```
function f(x: unknown, y: {} | null | undefined) {
```

```
    x = y; // 可以工作
```

```
    y = x; // 以前会报错，现在可以工作
```

```
}
```

另一个变化是 `{}` 与任何其它对象类型交叉会得到那个对象类型。因此，我们可以重

写 `NonNullable` 类型为与 `{}` 的交叉类型，因为 `{}` & `null` 和 `{}` & `undefined` 会被消掉。

```
- type NonNullable<T> = T extends null | undefined ? never : T;
```

```
+ type NonNullable<T> = T & {};
```

之所以称其为一项改进，是因为交叉类型可以被化简和赋值了，而有条件类型目前是不支持的。

因此，`NonNullable<NonNullable<T>>` 至少可以简化为 `NonNullable<T>`，在以前这是不行的。

```
function foo<T>(x: NonNullable<T>, y: NonNullable<NonNullable<T>> ) {
```

```
    x = y; // 一直没问题
```

```
    y = x; // 以前会报错，现在没问题
```

```
}
```

这些变化还为我们带来了更合理的控制流分析和类型细化。比如，`unknown` 在条件为“真”的分支中被细化为 `{}` | `null` | `undefined`。

```
function narrowUnknownishUnion(x: {} | null | undefined) {
```

```
    if (x) {
```

```
        x; // {}
```

```
    }
```

```
    else {
```

```
        x; // {} | null | undefined
```

```
    }
```

```
}
```

```
function narrowUnknown(x: unknown) {
```

```
    if (x) {
```

```
        x; // 以前是 'unknown', 现在是 '{}'
```

```
    }
```

```
    else {
```

```
        x; // unknown
```

```
    }
```

```
}
```

注意 `moduleSuffixes` 末尾的空字符串 `""` 是必须的，只有这样 TypeScript 才会去查

找 `./foo.ts`。也就是说，`moduleSuffixes` 的默认值是 `[""]`。

这个功能对于 React Native 工程是很有用的，因为对于不同的目标平台会有不同

的 `tsconfig.json` 和 `moduleSuffixes`。

这个功能是由 [Adam Foxman](#) 贡献的！

25.9 resolution-mode

Node.js 的 ECMAScript 解析规则是根据当前文件所属的模式以及使用的语法来决定如何解

析导入；然而，在 ECMAScript 模块里引用 CommonJS 模块也是很常用的，或者反过来。

TypeScript 允许使用 `/// <reference types="..." />` 指令。

```
/// <reference types="pkg" resolution-mode="require" />
```

```
// or
```

```
/// <reference types="pkg" resolution-mode="import" />
```

此外，在 Nightly 版本的 TypeScript 里，`import type` 可以指定导入断言来达到同样的目的。

```
// Resolve `pkg` as if we were importing with a `require()``
```

```
import type { TypeFromRequire } from "pkg" assert {
```

```
    "resolution-mode": "require"
```

```
};
```

```
// Resolve `pkg` as if we were importing with an `import``
```

```
import type { TypeFromImport } from "pkg" assert {
```

```
    "resolution-mode": "import"
```

```
};
```

```
export interface MergedType extends TypeFromRequire, TypeFromImport {}
```

这些断言也可以用在 `import()` 类型上。

```
export type TypeFromRequire =
```

```
    import("pkg", { assert: { "resolution-mode":
```

```
"require" } }).TypeFromRequire;
```

```
export type TypeFromImport =
```

```
    import("pkg", { assert: { "resolution-mode":
```

```
"import" } }).TypeFromImport;
```

```
export interface MergedType extends TypeFromRequire, TypeFromImport {}
```

`import type` 和 `import()` 语法仅在 [Nightly 版本](#) 里支持 `resolution-mode`。你可能会看到如下的错误：

Resolution mode assertions are unstable.

Use nightly TypeScript to silence this error.

Try updating with 'npm install -D typescript@next'.

如果你在 TypeScript 的 Nightly 版本中使用了该功能，别忘了可以[提供反馈](#)。

更多详情请查看 [PR：引用指令](#) 和 [PR：类型导入断言](#)。

25.10 跳转到在源码中的定义

TypeScript 4.7 支持了一个实验性的编辑器功能叫作 *Go To Source Definition*（跳转到在源码中的定义）。它和 *Go To Definition*（跳转到定义）相似，但不是跳转到声明

ts_upgrade_from_2.2_to_5.3

文件中。而是查找相应的实现文件（比如 .js 或 .ts 文件），并且在那里查找定义 - 即便这些文件总是会被声明文件 .d.ts 所遮蔽。

当你想查看导入的三方库的函数实现而不是 .d.ts 声明文件时是很便利的。

你可以在最新版本的 Visual Studio Code 里试用该功能。但该功能还是预览版，存在一些已知的限制。在某些情况下 TypeScript 使用启发式的方法来猜测函数定义的代码在哪个 .js 文件中，因此结果可能不太精确。Visual Studio Code 也不会提示哪些结果是通过猜测得到的，但我们正在实现它。

更多详情请参考 [PR](#)。

25.11 [分组整理导入语句](#)

TypeScript 为 JavaScript 和 TypeScript 提供了叫做“Organize Imports”（整理导入语句）编辑器功能。可是，它的行为有点简单粗暴，它直接排序所有的导入语句。

例如，在如下的代码上使用“Organize Imports”：

```
// local code
import * as bbb from "./bbb";
import * as ccc from "./ccc";
import * as aaa from "./aaa";

// built-ins
import * as path from "path";
import * as child_process from "child_process"
import * as fs from "fs";
```

// some code...

你会得到：

```
// local code
import * as child_process from "child_process";
import * as fs from "fs";
// built-ins
import * as path from "path";
import * as aaa from "./aaa";
import * as bbb from "./bbb";
import * as ccc from "./ccc";
```

// some code...

这不是我们想要的。尽管导入语句已经按它们的路径排序了，并且注释和折行被保留了，但仍不是我们期望的。

TypeScript 4.7 在“Organize Imports”时会考虑分组。再次在上例代码上执行“Organize Imports”会得到期望的结果：

```
// local code
import * as aaa from "./aaa";
import * as bbb from "./bbb";
import * as ccc from "./ccc";

// built-ins
import * as child_process from "child_process";
import * as fs from "fs";
import * as path from "path";
```

// some code...

感谢 [Minh Quy](#) 的 [PR](#)。

27 v4.9

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-4.9.html>

27.1 satisfies 运算符

TypeScript 开发者有时会感到进退两难：既想要确保表达式能够匹配某种类型，也想要表达式获得最确切的类型用作类型推断。

例如：

// 每个属性可能是 string 或 RGB 元组。

```
const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255]
```

// ^^^^ 拼写错误

```
};
```

// 我们想要在 'red' 上调用数组的方法

```
const redComponent = palette.red.at(0);
```

// 或者在 'green' 上调用字符串的方法

```
const greenNormalized = palette.green.toUpperCase();
```

注意，这里写成了 bleu，但我们想写的是 blue。通过给 palette 添加类型注释就能够捕获 bleu 拼写错误，但同时我们也失去了属性各自的信息。

```
type Colors = "red" | "green" | "blue";
```

```
type RGB = [red: number, green: number, blue: number];
```

```
const palette: Record<Colors, string | RGB> = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255]
```

// ~~~~ 能够检测到拼写错误

```
};
```

// 意想不到的错误 - 'palette.red' 可能为 string

```
const redComponent = palette.red.at(0);
```

新的 satisfies 运算符让我们可以验证表达式是否匹配某种类型，同时不改变表达式自身的类型。例如，可以使用 satisfies 来检验 palette 的所有属性与 string | number[] 是否兼容：

```
type Colors = "red" | "green" | "blue";
```

```
type RGB = [red: number, green: number, blue: number];
```

```
const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255]
```

// ~~~~ 捕获拼写错误

```
} satisfies Record<Colors, string | RGB>;
```

泛型也会进行类似的细化。当检查一个值不为 null 或 undefined 时，TypeScript 会将其与 {} 进行交叉 - 等同于使用 NonNullable。把所有变化放在一起，我们就可以在不使用类型断言的情况下定义下列函数。

```
function throwIfNullable<T>(value: T): NonNullable<T> {
  if (value === undefined || value === null) {
    throw Error("Nullable value!");
  }
}
```

// 以前会报错，因为 'T' 不能赋值给 'NonNullable<T>'。

// 现在会细化为 'T & {}' 并且不报错，因为它等同于 'NonNullable<T>'。

```
return value;
```

```
}
```

value 细化为了 T & {}, 此时它与 NonNullable<T> 等同 - 因此在函数体中不再需要使用 TypeScript 的特定语法。

就该改进本身而言可能是一个很小的变化 - 但它却实实在在地修复了在过去几年中报告的大量问题。

更多详情，请参考[这里](#)。

26.2 改进模版字符串类型中 infer 类型的类型推断

近期，TypeScript 支持了在有条件类型中的 infer 类型变量上添加 extends 约束。

// 提取元组类型中的第一个元素，若其能够赋值给 'number'，

// 返回 'never' 若无这样的元素。

```
type TryGetNumberIfFirst<T> =
  T extends [infer U extends number, ...unknown[]] ? U : never;
```

若 infer 类型出现在模版字符串类型中且被原始类型所约束，则 TypeScript 会尝试将其解析为字面量类型。

// SomeNum 以前是 'number'; 现在是 '100'。

```
type SomeNum = "100" extends `${infer U extends number}` ? U : never;
```

// SomeBigInt 以前是 'bigint'; 现在是 '100n'。

```
type SomeBigInt = "100" extends `${infer U extends bigint}` ? U :
never;
```

// SomeBool 以前是 'boolean'; 现在是 'true'。

```
type SomeBool = "true" extends `${infer U extends boolean}` ? U :
never;
```

现在它能更好地表达代码库在运行时的行为，提供更准确的类型。

要注意的一点是当 TypeScript 解析这些字面量类型时会使用贪心策略，尽可能多地提取原始类型；然后再回头检查解析出的原始类型是否匹配字符串的内容。也就是说，TypeScript 检查从字符串到原始类型再到字符串是否匹配。如果发现字符串前后对不上了，那么回退到基本的原始类型。

// JustNumber 为 `number` 因为 TypeScript 解析出 `"1.0"`，但

`String(Number("1.0"))` 为 `"1"` 不匹配。

```
type JustNumber = "1.0" extends `${infer T extends number}` ? T :
never;
```

更多详情请参考[这里](#)。

26.3 --build, --watch, 和 --incremental 的性能优化

TypeScript 4.8 优化了使用 --watch 和 --incremental 时的速度，以及使用 --build 构建工程引用时的速度。例如，现在在 --watch 模式下 TypeScript 不会去更新 v4.8 => 改进模版字符串类型中 infer 类型的类型推断

未改动文件的时间戳，这使得重新构建更快，避免与其它监视 TypeScript 输出文件的构建工具之间产生干扰。此外，TypeScript 也能够重用 --build, --watch 和 --incremental 之间的信息。这些优化有多大效果？在一个相当大的代码库上，对于简单常用的操作有 10%-25% 的改进，对于无改动操作的场景节省了 40% 的时间。在 TypeScript 代码库中我们也看到了相似的结果。更多详情请参考[这里](#)。

26.4 [比较对象和数组字面量时报错](#)

在许多语言中，== 操作符在对象上比较的是“值”。例如，在 Python 语言中想检查列表是否为空时可以使用 == 检查该值是否与空列表相等。

```
if people_at_home == []:
    print("that's where she lies, broken inside. </3")
```

在 JavaScript 里却不是这样，使用 == 和 === 比较对象和数组时比较的是引用。我们确信这会让 JavaScript 程序员搬起石头砸自己脚，且最坏的情况是在生产环境中存在 bug。因此，TypeScript 现在不允许如下的代码：

```
let peopleAtHome = [];
```

```
if (peopleAtHome === []) {
// ~~~~~
// This condition will always return 'false' since JavaScript compares
objects by reference, not value.
    console.log("that's where she lies, broken inside. </3")
}
```

非常感谢 [Jack Works](#) 的贡献。更多详情请参考[这里](#)。

26.5 [改进从绑定模式中进行类型推断](#)

在某些情况下，TypeScript 会从绑定模式中获取类型来帮助类型推断。

```
declare function chooseRandomly<T>(x: T, y: T): T;
```

```
let [a, b, c] = chooseRandomly([42, true, "hi!"], [0, false, "bye!"]);
//   ^   ^   ^
//   |   |   |
//   |   | string
//   |   |
//   |   | boolean
//   |   |
//   |   | number
```

当 chooseRandomly 需要确定 T 的类型时，它主要检查 [42, true, "hi!"] 和 [0, false, "bye!"]; 但 TypeScript 还需要确定这两个类型是 Array<number | boolean | string> 还是 [number, boolean, string]。为此，它会检查当前类型推断候选列表中是否存在元组类型。当 TypeScript 看到了绑定模式 [a, b, c]，它创建了类型 [any, any, any]，该类型会被加入到 T 的候选列表（作为推断 [42, true, "hi!"] 和 [0, false, "bye!"] 的参考）但优先级较低。这对 chooseRandomly 函数来讲不错，但在有些情况下不合适。例如：

```
declare function f<T>(x?: T): T;
```

```
let [x, y, z] = f();
```

绑定模式 [x, y, z] 提示 f 应该输出 [any, any, any] 元组；但是 f 不应该根据绑定模式来改变类型参数的类型。它不应该像变戏法一样根据被赋的值突然变成一个类数组的

值，因此绑定模式过多地影响到了生成的类型。由于绑定模式中均为 any 类型，因此我们也就让 x, y 和 z 为 any 类型。在 TypeScript 4.8 里，绑定模式不会成为类型参数的候选类型。它们仅在参数需要更确切的类型时提供参考，例如 chooseRandomly 的情况。如果你想回到之前的行为，可以提供明确的类型参数。更多详情请参考[这里](#)。

26.6 [修复文件监视（尤其是在 git checkout 之间）](#)

长久以来 TypeScript 中存在一个 bug，它对在编辑器中使用 --watch 模式监视文件改动处理的不好。它有时表现为错误提示不准确，需要重启 tsc 或 VS Code 才行。这在 Unix 系统上常发生，例如用 vim 保存了一个文件或切换了 git 的分支。这是因为错误地假设了 Node.js 在不同文件系统下处理文件重命名的方式。Linux 和 macOS 使用 [inodes](#)，[Node.js 监视的是 inodes 的变化而非文件路径](#)。因此，当 Node.js 返回了 [watcher 对象](#)，根据平台和文件系统的不同，它即可能监视文件路径也可能是 inode。为了高效，TypeScript 尝试重用 watcher 对象，如果它检测到文件路径仍存在于磁盘上。这里就产生了问题，因为即使给定路径上的文件仍然存在，但它可能是全新创建的文件，inode 已经发生了变化。TypeScript 重用了 watcher 对象而非重新创建一个 watcher 对象，因此可能监视了一个完全不相关的文件。TypeScript 4.8 能够在 inode 系统上处理这些情况，新建 watcher 对象。非常感谢 [Marc Celani](#) 和他的团队的贡献。更多详情请参考[这里](#)。

26.7 [查找所有引用性能优化](#)

在编辑器中执行“查找所有引用”时，TypeScript 现在能够更智能地聚合引用。在 TypeScript 自己的代码库中去搜索一个广泛使用的标识符时能够减少 20% 时间。更多详情请参考[这里](#)。

26.8 [从自动导入中排除指定文件](#)

TypeScript 4.8 增加了一个编辑器首选项从自动导入中排除指定文件。在 Visual Studio Code 里，可以将文件名和 globs 添加到 Settings UI 的“Auto Import File Exclude Patterns”下，或者 .vscode/settings.json 文件中：

```
{
  // Note that `javascript.preferences.autoImportFileExcludePatterns`
  can be specified for JavaScript too.
  "typescript.preferences.autoImportFileExcludePatterns": [
    "**/node_modules/@types/node"
  ]
}
```

如果你想避免导入某些模块或代码库，它个功能就派上用场了。有些模块可能有过多的导出以致于影响到了自动导入功能，让我们难以选择一条自动导入。更多详情请参考[这里](#)。

```

    this.#__name = name;
}

constructor(name: string) {
    this.name = name;
}
}

```

更多详情请参考 [PR](#)。

27.4 在 NaN 上的相等性检查

在 JavaScript 中，你无法使用内置的相等运算符去检查某个值是否等于 NaN。

由于一些原因，NaN 是个特殊的数值，它代表 不是一个数字。 没有值等于 NaN，包括 NaN 自己！

```

console.log(NaN == 0) // false
console.log(NaN === 0) // false

```

```

console.log(NaN == NaN) // false
console.log(NaN === NaN) // false

```

换句话说，任何值都不等于 NaN。

```

console.log(NaN != 0) // true
console.log(NaN !== 0) // true

```

```

console.log(NaN != NaN) // true
console.log(NaN !== NaN) // true

```

从技术上讲，这不是 JavaScript 独有的问题，任何使用 IEEE-754 浮点数的语言都有一样的问题： 但是 JavaScript 中主要的数值类型为浮点数，并且解析数值时经常会得到 NaN。因此，检查 NaN 是很常见的操作，正确的方法是使用 Number.isNaN 函数 - 但像上文提到的，很多人可能不小心地使用了 someValue === NaN 来进行检查。

现在，如果 TypeScript 发现直接比较 NaN 会报错，并提示使用 Number.isNaN。

```

function validate(someValue: number) {
    return someValue !== NaN;
    // ~~~~~
    // error: This condition will always return 'true'.
    // Did you mean '!Number.isNaN(someValue)'?
}

```

我们确信这个改动会帮助捕获初级的错误，就如同 TypeScript 也会检查比较对象字面量和数组字面量一样。

感谢 [Oleksandr Tarasiuk](#) 提交的 [PR](#)。

27.5 监视文件功能使用文件系统事件

在先前的版本中，TypeScript 主要依靠轮询来监视每个文件。 使用轮询的策略意味着定期检查文件是否有更新。 在 Node.js 中，fs.watchFile 是内置的使用轮询来检查文件变动的方法。 虽说轮询在跨操作系统和文件系统的情况下更稳妥，但是它也意味着 CPU 会定期地被中断，转而去检查是否有文件更新即便在没有任何改动的情况下。 这在只有少数文件的时候问题不大，但如果工程包含了大量文件 - 或 node_modules 里有大量的文件 - 就会变得非常吃资源。

通常来讲，更好的做法是使用文件系统事件。 做为轮询的替换，我们声明对某些文件的变动感兴趣并提供回调函数用于处理有改动的文件。 大多数现代的平台提供了

如 CreateIoCompletionPort、kqueue、epoll 和 inotify API。 Node.js 对这些 API 进行了抽象，提供了 fs.watch API。 文件系统事件通常可以很好地工作，但是也存在

// 依然可以访问这些方法

```

const redComponent = palette.red.at(0);
const greenNormalized = palette.green.toUpperCase();
satisfies 可以用来捕获许多错误。 例如，检查一个对象是否包含了某个类型要求的所有的键，并且没有多余的：
type Colors = "red" | "green" | "blue";

```

// 确保仅包含 'Colors' 中定义的键

```

const favoriteColors = {
    "red": "yes",
    "green": false,
    "blue": "kinda",
    "platypus": false
}
// ~~~~~ 错误 - "platypus" 不在 'Colors' 中
} satisfies Record<Colors, unknown>;

```

// 'red', 'green', and 'blue' 的类型信息保留下来

```

const g: boolean = favoriteColors.green;
有可能我们不太在乎属性名，在乎的是属性值的类型。 在这种情况下，我们也能够确保对象属性值的类型是匹配的。
type RGB = [red: number, green: number, blue: number];

```

```

const palette = {
    red: [255, 0, 0],
    green: "#00ff00",
    blue: [0, 0]
    // ~~~~~ 错误！
} satisfies Record<string, string | RGB>;

```

// 类型信息保留下来

```

const redComponent = palette.red.at(0);
const greenNormalized = palette.green.toUpperCase();
更多示例请查看这里和这里。 感谢 Oleksandr Tarasiuk 对该属性的贡献。

```

27.2 使用 in 运算符来细化并未列出其属性的对象类型

开发者经常需要处理在运行时不完全已知的值。 事实上，我们常常不能确定对象的某个属性是否存在，是否从服务端得到了响应或者读取到了某个配置文件。 JavaScript 的 in 运算符能够检查对象上是否存在某个属性。

从前，TypeScript 能够根据没有明确列出的属性来细化类型。

```

interface RGB {
    red: number;
    green: number;
    blue: number;
}

```

```

interface HSV {
    hue: number;
    saturation: number;
}

```

ts_upgrade_from_2.2_to_5.3

```
    value: number;
}

function setColor(color: RGB | HSV) {
    if ("hue" in color) {
        // 'color' 类型为 HSV
    }
    // ...
}
```

此处，RGB 类型上没有列出 hue 属性，因此被细化掉了，剩下了 HSV 类型。

那如果每个类型上都没有列出这个属性呢？在这种情况下，语言无法提供太多的帮助。看下面的 JavaScript 示例：

```
function tryGetPackageName(context) {
    const packageJSON = context.packageJSON;
    // Check to see if we have an object.
    if (packageJSON && typeof packageJSON === "object") {
        // Check to see if it has a string name property.
        if ("name" in packageJSON && typeof packageJSON.name ===
"string") {
            return packageJSON.name;
        }
    }

    return undefined;
}
```

将上面的代码改写为合适的 TypeScript，我们会给 context 定义一个类型；然而，在旧版本的 TypeScript 中如果声明 packageJSON 属性的类型为安全的 unknown 类型会有问题。

```
interface Context {
    packageJSON: unknown;
}
```

```
function tryGetPackageName(context: Context) {
    const packageJSON = context.packageJSON;
    // Check to see if we have an object.
    if (packageJSON && typeof packageJSON === "object") {
        // Check to see if it has a string name property.
        if ("name" in packageJSON && typeof packageJSON.name ===
"string") {
            // ~~~~
            // error! Property 'name' does not exist on type 'object'.
            return packageJSON.name;
            // ~~~~
            // error! Property 'name' does not exist on type 'object'.
        }
    }

    return undefined;
}
```

v190

v4.9 => 使用 in 运算符来细化并未列出其属性的对象类型

ts_upgrade_from_2.2_to_5.3

这是因为当 packageJSON 的类型从 unknown 细化为 object 类型后，in 运算符会严格地将类型细化为包含了所检查属性的某个类型。因此，packageJSON 的类型仍为 object。

TypeScript 4.9 增强了 in 运算符的类型细化功能，它能够更好地处理没有列出属性的类型。现在 TypeScript 不是什么也不做，而是将其类型与 Record<"property-key-being-checked", unknown> 进行类型交叉运算。

因此在上例中，packageJSON 的类型将从 unknown 细化为 object 再细化为 object & Record<"name", unknown>。这样就允许我们访问并细化类型 packageJSON.name。

```
interface Context {
    packageJSON: unknown;
}
```

```
function tryGetPackageName(context: Context): string | undefined {
    const packageJSON = context.packageJSON;
    // Check to see if we have an object.
    if (packageJSON && typeof packageJSON === "object") {
        // Check to see if it has a string name property.
        if ("name" in packageJSON && typeof packageJSON.name ===
"string") {
            // Just works!
            return packageJSON.name;
        }
    }

    return undefined;
}
```

TypeScript 4.9 还会严格限制 in 运算符的使用，以确保左侧的操作数能够赋值给 string | number | symbol，右侧的操作数能够赋值给 object。它有助于检查是否使用了合法的属性名，以及避免在原始类型上进行检查。
更多详情请查看 [PR](#)。

27.3 类中的自动存取器

TypeScript 4.9 支持了 ECMAScript 即将引入的“自动存取器”功能。自动存取器的声明如同定义一个类的属性，只不过是需要使用 accessor 关键字。

```
class Person {
    accessor name: string;

    constructor(name: string) {
        this.name = name;
    }
}

class Person {
    #__name: string;

    get name() {
        return this.#__name;
    }

    set name(value: string) {
```

v4.9 => 类中的自动存取器

191

"这些 any 是怎么回事？都啥啊？"

先别急 - 这里我们是想简化一下问题，将注意力集中在函数的功能上。注意一

下 `loggedMethod` 接收原方法 (`originalMethod`) 作为参数并返回一个函数：

1. 打印 "Entering..." 消息
2. 将 `this` 值以及所有的参数传递给原方法
3. 打印 "Exiting..." 消息，并且
4. 返回原方法的返回值。

现在可以使用 `loggedMethod` 来装饰 `greet` 方法：

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }

  @loggedMethod
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}
```

```
const p = new Person('Ron');
p.greet();
```

```
// 输出:
//
// LOG: Entering method.
// Hello, my name is Ron.
// LOG: Exiting method.
```

我们刚刚在 `greet` 上使用了 `loggedMethod` 装饰器 - 注意一下写法 `@loggedMethod`。这样做之后，`loggedMethod` 被调用时会传入被装饰的目标 `target` 以及一个上下文对象 `context object` 作为参数。因为 `loggedMethod` 返回了一个新函数，因此这个新函数会替换掉 `greet` 的原始定义。

在 `loggedMethod` 的定义中带有第二个参数。它就是上下文对象 `context object`，包含了一些有关于装饰器声明细节的有用信息 - 例如是否为 `#private` 成员，或者 `static`，或者方法的名称。让我们重写 `loggedMethod` 来使用这些信息，并且打印出被装饰的方法的名字。

```
function loggedMethod(
  originalMethod: any,
  context: ClassMethodDecoratorContext
) {
  const methodName = String(context.name);

  function replacementMethod(this: any, ...args: any[]) {
    console.log(`LOG: Entering method '${methodName}'.`);
    const result = originalMethod.call(this, ...args);
    console.log(`LOG: Exiting method '${methodName}'.`);
    return result;
  }
}
```

一些注意事项。一个 `watcher` 需要考虑 [inode watching](#) 的问题、[在一些文件系统中不可用](#) 的问题（比如：网络文件系统）、嵌套的文件监控是否可用、重命名目录是否触发事件以及可用 `file watcher` 耗尽的问题！换句话说，这件事不是那么容易做的，特别是我们还需要跨平台。

因此，过去我们的默认选择是普遍好用的方式：轮询。虽不总是，但大部分时候是这样的。后来，我们提供了[选择文件监视策略的方法](#)。这让我们收到了很多使用反馈并改善跨平台的问题。由于 `TypeScript` 必须要能够处理大规模的代码并且也已经有了改进，因此我们觉得切换到使用文件系统事件是件值得做的事情。

在 `TypeScript` 4.9 中，文件监视已经默认使用文件系统事件的方式，仅当无法初始化事件监视时才回退到轮询。对大部分开发者来讲，在使用 `--watch` 模式或在 `Visual Studio`、`VS Code` 里使用 `TypeScript` 时会极大降低资源的占用。

[文件监视方式仍然是可以配置的](#)，可以使用环境变量和 `watchOptions` - 像 [VS Code 这样的编辑器还支持单独配置](#)。如果你的代码使用的是网络文件系统（如 `NFS` 和 `SMB`）就需要回退到旧的行为；但如果服务器有强大的处理能力，最好是启用 `SSH` 并且通过远程运行 `TypeScript`，这样就可以使用本地文件访问。`VS Code` 支持了很多[远程开发](#)的工具。

27.6 编辑器中的“删除未使用导入”和“排序导入”命令

以前，`TypeScript` 仅支持两个管理导入语句的编辑器命令。拿下面的代码举例：

```
import { Zebra, Moose, HoneyBadger } from "./zoo";
import { foo, bar } from "./helper";
```

```
let x: Moose | HoneyBadger = foo();
```

第一个命令是“组织导入语句”，它会删除未使用的导入并对剩下的条目排序。因此会将上面的代码重写为：

```
import { foo } from "./helper";
import { HoneyBadger, Moose } from "./zoo";
```

```
let x: Moose | HoneyBadger = foo();
```

在 `TypeScript` 4.3 中，引入了“排序导入语句”命令，它仅排序导入语句但不进行删除，因此会将上例代码重写为：

```
import { bar, foo } from "./helper";
import { HoneyBadger, Moose, Zebra } from "./zoo";
```

```
let x: Moose | HoneyBadger = foo();
```

使用“排序导入语句”的注意事项是，在 `VS Code` 中该命令只能在保存文件时触发，而非能够手动执行的命令。

`TypeScript` 4.9 添加了另一半功能，提供了“移除未使用的导入”功能。`TypeScript` 会移除未使用的导入命名和语句，但是不能改变当前的排序。

```
import { Moose, HoneyBadger } from "./zoo";
import { foo } from "./helper";
```

```
let x: Moose | HoneyBadger = foo();
```

该功能对任何编译器都是可用的；但要注意的是，`VS Code` (1.73+) 会内置这个功能并且可以使用 `Command Palette` 来执行。如果用户想要使用更细的“移除未使用的导入”或“排序导入”命令，那么可以将“组织导入”的快捷键绑定到这些命令上。

更多详情请参考[这里](#)。

27.7 在 return 关键字上使用跳转到定义

在编辑器中，当在 return 关键字上使用跳转到定义功能时，TypeScript 会跳转到函数的顶端。这会帮助理解 return 语句是属于哪个函数的。

我们期待这个功能扩展到更多的关键字上，例如 await 和 yield 或者 switch、case 和 default。感谢 [Oleksandr Tarasiuk](#) 的实现。

27.8 性能优化

TypeScript 进行了一些较小的但是能觉察到的性能优化。

首先，重写了 TypeScript 的 forEachChild 函数使用函数查找表代替 switch 语句。forEachChild 是编译器在遍历语法节点时会反复调用的函数，和部分语言服务一起大量地被使用在编译绑定阶段。对 forEachChild 函数的重构减少了绑定阶段和语言服务操作的 20% 时间消耗。

当我们看到了 forEachChild 的效果后也在 visitEachChild（在编译器和语言服务中用来变换节点的函数）上进行了类似的优化。同样的重构减少了 3% 生成工程输出的时间消耗。

对于 forEachChild 的优化最初是受到了 [Artemis Everfree](#) 文章的启发。虽说我们认为速度提升的根本原因是由于函数体积和复杂度的降低而非这篇文章里提到的问题，但我们非常感谢能够从中获得经验并快速地进行重构让 TypeScript 运行得更快。

最后，TypeScript 还优化了在条件类型的 true 分支中保留类型信息。例如：

```
interface Zoo<T extends Animal> {  
    // ...  
}
```



```
type MakeZoo<A> = A extends Animal ? Zoo<A> : never;  
TypeScript 在检查 Zoo<A>时需要记住 A 是 Animal。TypeScript 通过新建 A 和 Animal 的交叉类型来保留该信息；然而，TypeScript 之前采用的是即时求值的方式，即便有时是不需要的。而且类型检查器中的一些问题代码使得这些类型无法被简化。TypeScript 现在会推迟类型交叉操作直到真的有需要的时候。对于大量地使用了有条件类型的代码来说，你会觉察到大幅的提速，但从我们的性能测试结果来看却只看到了 3% 的类型检查性能提升。
```

28 v5.0

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-5.0.html>

28.1 装饰器 Decorators

装饰器是即将到来的 ECMAScript 特性，它允许我们定制可重用的类以及类成员。

考虑如下的代码：

```
class Person {  
    name: string;  
    constructor(name: string) {  
        this.name = name;  
    }  
  
    greet() {  
        console.log(`Hello, my name is ${this.name}.`);  
    }  
}
```



```
const p = new Person('Ron');  
p.greet();
```

这里的 greet 很简单，但我们假设它很复杂 - 例如包含异步的逻辑，是递归的，具有副作用等。不管你把它想像成多么混乱复杂，现在我们想插入一些 console.log 语句来调试 greet。

```
class Person {  
    name: string;  
    constructor(name: string) {  
        this.name = name;  
    }  
  
    greet() {  
        console.log('LOG: Entering method.');  
        console.log(`Hello, my name is ${this.name}.`);  
  
        console.log('LOG: Exiting method.');    }  
}
```

这个做法太常见了。如果有种办法能给每一个类方法都添加打印功能就太好了！这就是装饰器的用武之地。让我们编写一个函数 loggedMethod：

```
function loggedMethod(originalMethod: any, _context: any) {  
    function replacementMethod(this: any, ...args: any[]) {  
        console.log('LOG: Entering method.');        const result = originalMethod.call(this, ...args);  
        console.log('LOG: Exiting method.');        return result;  
    }  
  
    return replacementMethod;  
}
```



```
@register
class Bar {
    // ...
}
```

// error - before *and* after is not allowed

```
@before
@after
export class Bar {
    // ...
}
```

28.3 编写强类型的装饰器

上面的例子 `loggedMethod` 和 `bound` 是故意写的简单并且忽略了大量和类型有关的细节。

为装饰器添加类型可能会很复杂。例如，强类型的 `loggedMethod` 可能像下面这样：

```
function loggedMethod<This, Args extends any[], Return>(
    target: (this: This, ...args: Args) => Return,
    context: ClassMethodDecoratorContext<
        This,
        (this: This, ...args: Args) => Return
    >
) {
    const methodName = String(context.name);

    function replacementMethod(this: This, ...args: Args): Return {
        console.log(`LOG: Entering method '${methodName}'.`);
        const result = target.call(this, ...args);
        console.log(`LOG: Exiting method '${methodName}'.`);
        return result;
    }

    return replacementMethod;
}
```

我们必须分别给原方法的 `this`、形式参数和返回值添加类型，上面使用了类型参数 `This`、`Args` 以及 `Return`。装饰器函数到底有多复杂取决于你要确保什么。但要记住，装饰器被使用的次数远多于被编写的次数，因此强类型的版本是通常希望得到的 - 但我们需要在可读性之间做出取舍，因此要尽量保持简洁。

未来会有更多关于如何编写装饰器的文档 - 但是[这篇文章](#)详细介绍了装饰器的工作方式。

28.4 `const` 类型参数

在推断对象类型时，TypeScript 通常会选择一个通用类型。例如，下例中 `names` 的推断类型为 `string[]`：

```
type HasNames = { readonly names: string[] };
function getNamesExactly<T extends HasNames>(arg: T): T['names'] {
    return arg.names;
}
```

// Inferred type: string[]

```
const names = getNamesExactly({ names: ['Alice', 'Bob', 'Eve'] });
```

这样做的目的通常是为了允许后面可以进行修改。

```
return replacementMethod;
```

```
}
```

我们使用了上下文参数。TypeScript 提供了名为 `ClassMethodDecoratorContext` 的类型用于描述装饰器方法接收的上下文对象。

除了元数据外，上下文对象中还提供了一个有用的函数 `addInitializer`。它提供了一种方式来 `hook` 到构造函数的起始位置。

例如在 JavaScript 中，下面的情形很常见：

```
class Person {
    name: string;
    constructor(name: string) {
        this.name = name;

        this.greet = this.greet.bind(this);
    }

    greet() {
        console.log(`Hello, my name is ${this.name}.`);
    }
}
```

或者，`greet` 可以被声明为使用箭头函数初始化的属性。

```
class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }

    greet = () => {
        console.log(`Hello, my name is ${this.name}.`);
    };
}
```

这类代码的目的是确保 `this` 值不会被重新绑定，当 `greet` 被独立地调用或者在用作回调函数时。

```
const greet = new Person('Ron').greet;
```

// 我们不希望下面的调用失败

```
greet();
```

我们可以定义一个装饰器来利用 `addInitializer` 在构造函数里调用 `bind`。

```
function bound(originalMethod: any, context:
    ClassMethodDecoratorContext) {
    const methodName = context.name;
    if (context.private) {
        throw new Error(
            `'bound' cannot decorate private properties like ${methodName as
            string}.`
        );
    }
    context.addInitializer(function () {
        this[methodName] = this[methodName].bind(this);
    });
}
```

```
ts_upgrade_from_2.2_to_5.3
});
}
bound 没有返回值 - 因此当它装饰一个方法时, 不会影响原先的方法。 但是, 它会在字段被初始化前添加一些逻辑。
```

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }

  @bound
  @loggedMethod
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}
```

```
const p = new Person('Ron');
const greet = p.greet;
```

```
// Works!
greet();
```

我们将两个装饰器叠在了一起 - @bound 和 @loggedMethod。 这些装饰器以“相反的”顺序执行。 也就是说, @loggedMethod 装饰原始方法 greet, @bound 装饰的是 @loggedMethod 的结果。 此例中, 这不太重要 - 但如果你的装饰器带有副作用或者期望特定的顺序, 那就不一样了。

值得注意的是: 如果你在乎代码样式, 也可以将装饰器放在同一行上。

```
@bound @loggedMethod greet() {
  console.log(`Hello, my name is ${this.name}.`);
}
```

可能不太明显的一点是, 你甚至可以定义一个返回装饰器函数的函数。 这样我们可以在一定程度上定制最终的装饰器。 我们可以让 loggedMethod 返回一个装饰器并且定制如何打印消息。

```
function loggedMethod(headMessage = 'LOG:') {
  return function actualDecorator(
    originalMethod: any,
    context: ClassMethodDecoratorContext
  ) {
    const methodName = String(context.name);

    function replacementMethod(this: any, ...args: any[]) {
      console.log(`${headMessage} Entering method '${methodName}'.`);
      const result = originalMethod.call(this, ...args);
      console.log(`${headMessage} Exiting method '${methodName}'.`);
      return result;
    }

    return replacementMethod;
  };
}
```

```
}
这样做之后, 在使用 loggedMethod 装饰器之前需要先调用它。 接下来就可以传入任意字符串作为打印消息的前缀。
```

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }

  @loggedMethod('')
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}
```

```
const p = new Person('Ron');
p.greet();
```

```
// Output:
//
//   Entering method 'greet'.
//   Hello, my name is Ron.
//   Exiting method 'greet'.
```

装饰器不仅可以用在方法上! 它们也可以被用在属性/字段, 存取器 (getter/setter) 以及自动存取器。 甚至, 类本身也可以被装饰, 用于处理子类化和注册。

想深入了解装饰器, 可以阅读 Axel Rauschmayer 的[文章](#)。

更多详情请参考 [PR](#)。

28.2 与旧的实验性的装饰器的差异

如果你有一定的 TypeScript 经验, 你会发现 TypeScript 多年前就已经支持了“实验性的”装饰器特性。 虽然实验性的装饰器非常地好用, 但是它实现的是旧版本的装饰器规范, 并且总是需要启用 --experimentalDecorators 编译器选项。 若没有启用它并且使用了装饰器, TypeScript 会报错。

在未来的一段时间内, --experimentalDecorators 依然会存在; 然而, 如果不使用该标记, 在新代码中装饰器语法也是合法的。 在 --experimentalDecorators 之外, 它们的类型检查和代码生成方式也不同。 类型检查和代码生成规则存在巨大差异, 以至于虽然装饰器可以被定义为同时支持新、旧装饰器的行为, 但任何现有的装饰器函数都不太可能这样做。

新的装饰器提案与 --emitDecoratorMetadata 的实现不兼容, 并且不支持在参数上使用装饰器。 未来的 ECMAScript 提案可能会弥补这个差距。

最后要注意的是: 除了可以在 export 关键字之前使用装饰器, 还可以在 export 或者 export default 之后使用。 但是不允许混合使用两种风格。

```
// allowed
@register
export default class Foo {
  // ...
}
```

```
// also allowed
export default
```

```
function isPrimaryColor(c: Color): c is PrimaryColor {
    // Narrowing literal types can catch bugs.
    // TypeScript will error here because
    // we'll end up comparing 'Color.Red' to 'Color.Green'.
    // We meant to use ||, but accidentally wrote &&.
    return c === Color.Red && c === Color.Green && c === Color.Blue;
}
```

为每个枚举成员提供其自己的类型的一个问题是，这些类型在某种程度上与成员的实际值相关联。在某些情况下，无法计算该值 - 例如，枚举成员可能由函数调用初始化。

```
enum E {
    Blah = Math.random(),
}
```

每当 TypeScript 遇到这些问题时，它会悄悄地退而使用旧的枚举策略。这意味着放弃所有联合类型和字面量类型的优势。

TypeScript 5.0 通过为每个计算成员创建唯一类型，成功将所有枚举转换为联合枚举。这意味着现在所有枚举都可以被细化，并且每个枚举成员都有其自己的类型。

更多详情请参考 [PR](#)

28.7 --moduleResolution bundler

TypeScript 4.7 支持将 --module 和 --moduleResolution 选项设置为 node16 和 nodenext。这些选项的目的是更好地模拟 Node.js 中 ECMAScript 模块的精确查找规则；然而，这种模式存在许多其他工具实际上并不强制执行的限制。

例如，在 Node.js 的 ECMAScript 模块中，任何相对导入都需要包含文件扩展名。

```
// entry.mjs
import * as utils from './utils'; // wrong - we need to include the
file extension.
```

```
import * as utils from './utils.mjs'; // works
```

对于 Node.js 和浏览器来说，这样做有一些原因 - 它可以加快文件查找速度，并且对于简单的文件服务器效果更好。但是对于许多使用打包工具的开发人员来说，node16 / nodenext 设置很麻烦，因为打包工具中没有这么多限制。在某些方面，node 解析模式对于任何使用打包工具的人来说是更好的。

但在某些方面，原始的 node 解析模式已经过时了。大多数现代打包工具在 Node.js 中使用 ECMAScript 模块和 CommonJS 查找规则的融合。例如，像在 CommonJS 中一样，无扩展名的导入也可以正常工作，但是在查找包的导出条件时，它们将首选像在 ECMAScript 文件中一样的 import 条件。

为了模拟打包工具的工作方式，TypeScript 现在引入了一种新策略：--moduleResolution bundler。

```
{
  "compilerOptions": {
    "target": "esnext",
    "moduleResolution": "bundler"
  }
}
```

如果你使用如 Vite, esbuild, swc, Webpack, parcel 等现代打包工具，它们实现了混合的查找策略，新的 bundler 选项是更好的选择。

另一方面，如果您正在编写一个要发布到 npm 的代码库，那么使用 bundler 选项可能会隐藏影响未使用打包工具用户的兼容性问题。因此，在这些情况下，使用 node16 或 nodenext 解析选项可能是更好的选择。

然而，根据 getNamesExactly 的具体功能和预期使用方式，通常情况下需要更加具体的类型。

直到现在，API 作者们通常不得不在一些位置添加 as const 来达到预期的类型推断目的：

```
// The type we wanted:
//   readonly ["Alice", "Bob", "Eve"]
// The type we got:
//   string[]
const names1 = getNamesExactly({ names: ['Alice', 'Bob', 'Eve'] });
```

```
// Correctly gets what we wanted:
//   readonly ["Alice", "Bob", "Eve"]
const names2 = getNamesExactly({ names: ['Alice', 'Bob', 'Eve'] } as
const);
```

这样做既繁琐又容易忘。在 TypeScript 5.0 里，你可以为类型参数声明添加 const 修饰符，这使得 const 形式的类型推断成为默认行为：

```
type HasNames = { names: readonly string[] };
function getNamesExactly<const T extends HasNames>(arg: T): T['names']
{
    //           ^^^^^^
    return arg.names;
}
```

```
// Inferred type: readonly ["Alice", "Bob", "Eve"]
// Note: Didn't need to write 'as const' here
const names = getNamesExactly({ names: ['Alice', 'Bob', 'Eve'] });
注意，const 修饰符不会拒绝可修改的值，并且不需要不可变约束。使用可变类型约束可能会产生令人惊讶的结果。
```

```
declare function fnBad<const T extends string[]>(args: T): void;
```

```
// 'T' is still 'string[]' since 'readonly ["a", "b", "c"]' is not
assignable to 'string[]'
fnBad(['a', 'b', 'c']);
这里，T 的候选推断类型为 readonly ["a", "b", "c"]，但是 readonly 只读数组不能用在需要可变数组的地方。这种情况下，类型推断会回退到类型约束，将数组视为 string[] 类型，因此函数调用仍然会成功。
```

```
这个函数更好的定义是使用 readonly string[]:
declare function fnGood<const T extends readonly string[]>(args: T):
void;
```

```
// T is readonly ["a", "b", "c"]
fnGood(['a', 'b', 'c']);
要注意 const 修饰符只影响在函数调用中直接写出的对象、数组和基本表达式的类型推断，因此，那些无法（或不会）使用 as const 进行修饰的参数在行为上不会有任何变化：
declare function fnGood<const T extends readonly string[]>(args: T):
void;
const arr = ['a', 'b', 'c'];
```

```
// 'T' is still 'string[]'-- the 'const' modifier has no effect here
```

```
ts_upgrade_from_2.2_to_5.3
```

```
fnGood(arr);
```

更多详情请参考 [PR](#), [PR](#) 和 [PR](#)。

28.5 [extends](#) 支持多个配置文件

在管理多个项目时，拥有一个“基础”配置文件，其他 `tsconfig.json` 文件可以继承它，这会非常有帮助。这就是为什么 TypeScript 支持使用 `extends` 字段来

从 `compilerOptions` 中复制字段的原因。

```
// packages/front-end/src/tsconfig.json
```

```
{
  "extends": "../.././tsconfig.base.json",
  "compilerOptions": {
    "outDir": "../lib"
    // ...
  }
}
```

然而，有时您可能想要从多个配置文件中进行继承。例如，假设您正在使用一个[在 npm 上发布的 TypeScript 基础配置文件](#)。如果您希望自己所有的项目也使用 npm 上的 `@tsconfig/strictest` 包中的选项，那么有一个简单的解决方案：

让 `tsconfig.base.json` 从 `@tsconfig/strictest` 进行扩展：

```
// tsconfig.base.json
```

```
{
  "extends": "@tsconfig/strictest/tsconfig.json",
  "compilerOptions": {
    // ...
  }
}
```

这在某种程度上是有效的。如果您的某些工程不想使用 `@tsconfig/strictest`，那么必须手动禁用这些选项，或者创建一个不继承

于 `@tsconfig/strictest` 的 `tsconfig.base.json`。

为了提高灵活性，TypeScript 5.0 允许 `extends` 字段指定多个值。例如，有如下的配置文件：

```
{
  "extends": ["a", "b", "c"],
  "compilerOptions": {
    // ...
  }
}
```

这样写就如同是直接继承 `c`，而 `c` 继承于 `b`，`b` 继承于 `a`。如果出现冲突，后来者会被采纳。

在下面的例子中，在最终

的 `tsconfig.json` 中 `strictNullChecks` 和 `noImplicitAny` 会被启用。

```
// tsconfig1.json
```

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

```
// tsconfig2.json
```

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

```
// tsconfig.json
```

```
{
  "extends": ["/tsconfig1.json", "/tsconfig2.json"],
  "files": ["/index.ts"]
}
```

另一个例子，我们可以这样改写最初的示例：

```
// packages/front-end/src/tsconfig.json
```

```
{
  "extends": [
    "@tsconfig/strictest/tsconfig.json",
    "../.././tsconfig.base.json"
  ],
  "compilerOptions": {
    "outDir": "../lib"
    // ...
  }
}
```

更多详情请参考：[PR](#)。

28.6 所有的 `enum` 均为联合 `enum`

在最初 TypeScript 引入枚举类型时，它们只不过是一组同类型的数值常量。

```
enum E {
  Foo = 10,
  Bar = 20,
}
```

`E.Foo` 和 `E.Bar` 唯一特殊的地方在于它们可以赋值给任何期望类型为 `E` 的地方。除此之外，它们基本上等同于 `number` 类型。

```
function takeValue(e: E) {}
```

```
takeValue(E.Foo); // works
```

```
takeValue(123); // error!
```

直到 TypeScript 2.0 引入了枚举字面量类型，枚举才变得更为特殊。枚举字面量类型为每个枚举成员提供了其自己的类型，并将枚举本身转换为每个成员类型的联合类型。它们还允许我们仅引用枚举中的一部分类型，并细化掉那些类型。

```
// Color is like a union of Red | Orange | Yellow | Green | Blue | Violet
```

```
enum Color {
  Red, Orange, Yellow, Green, Blue, /* Indigo */, Violet
}
```

```
// Each enum member has its own type that we can refer to!
```

```
type PrimaryColor = Color.Red | Color.Green | Color.Blue;
```



```
export = {
  foo,
  bar,
  baz,
};

// ==>

function foo() {}
function bar() {}
function baz() {}

module.exports = {
  foo,
  bar,
  baz,
};
```

虽然这是一种限制，但它确实有助于使一些问题更加明显。 例如，在 `--module node16` 下很容易忘记在 `package.json` 中设置 `type` 字段。 结果是开发人员会开始编写 CommonJS 模块而不是 ES 模块，但却没有意识到这一点，从而导致查找规则和 JavaScript 输出出现意外的结果。 这个新的标志确保您有意识地使用文件类型，因为语法是刻意不同的。 因为 `--verbatimModuleSyntax` 相比于 `--importsNotUsedAsValues` 和 `--preserveValueImports` 提供了更加一致的行为，推荐使用前者，后两个标记将被弃用。

更多详情请参考 [PR](#) 和 [issue](#)。

28.10 [支持 export type *](#)

在 TypeScript 3.8 引入类型导入时，该语法不支持在 `export * from "module"` 或 `export * as ns from "module"` 重新导出上使用。 TypeScript 5.0 添加了对两者的支持：

```
// models/vehicles.ts
export class Spaceship {
  // ...
}

// models/index.ts
export type * as vehicles from './vehicles';

// main.ts
import { vehicles } from './models';

function takeASpaceship(s: vehicles.Spaceship) {
  // ok - `vehicles` only used in a type position
}

function makeASpaceship() {
  return new vehicles.Spaceship();
  //      ^^^^^^^^^
  // 'vehicles' cannot be used as a value because it was exported
  using 'export type'.
```

更多详情请参考 [PR](#)

28.8 [定制化解析的标记](#)

JavaScript 工具现在可以模拟“混合”解析规则，就像我们上面描述的 `bundler` 模式一样。 由于工具的支持可能有所不同，因此 TypeScript 5.0 提供了启用或禁用一些功能的方法，这些功能可能无法与您的配置一起使用。

28.8.1 [allowImportingTsExtensions](#)

`--allowImportingTsExtensions` 允许 TypeScript 文件导入使用了 TypeScript 特定扩展名的文件，例如 `.ts`、`.mts`、`.tsx`。 此标记仅在启用了 `--noEmit` 或 `--emitDeclarationOnly` 时允许使用， 因为这些导入路径无法在运行时的 JavaScript 输出文件中被解析。 这里的期望是，您的解析器（例如打包工具、运行时或其他工具）将保证这些在 `.ts` 文件之间的导入可以工作。

28.8.2 [resolvePackageJsonExports](#)

`--resolvePackageJsonExports` 强制 TypeScript 使用 [package.json](#) 里的 [exports](#) 字段，如果它尝试读取 `node_modules` 里的某个包。 当 `--moduleResolution` 为 `node16`、`nodenext` 和 `bundler` 时，该选项的默认值为 `true`。

28.8.3 [resolvePackageJsonImports](#)

`--resolvePackageJsonImports` 强制 TypeScript 使用 [package.json](#) 里的 [imports](#) 字段，当它查找以 `#` 开头的文件时，且该文件的父目录中包含 `package.json` 文件。 当 `--moduleResolution` 为 `node16`、`nodenext` 和 `bundler` 时，该选项的默认值为 `true`。

28.8.4 [allowArbitraryExtensions](#)

在 TypeScript 5.0 中，当导入路径不是以已知的 JavaScript 或 TypeScript 文件扩展名结尾时，编译器将查找该路径的声明文件，形式为 `{文件基础名称}.d.{扩展名}.ts`。 例如，如果您在打包项目中使用 CSS 加载器，您可能需要编写（或生成）如下的声明文件：

```
/* app.css */
.cookie-banner {
  display: none;
}
// app.d.css.ts
declare const css: {
  cookieBanner: string;
};
export default css;
// App.tsx
import styles from './app.css';
```

`styles.cookieBanner`; // string

默认情况下，该导入将引发错误，告诉您 TypeScript 不支持此文件类型，您的运行时可能不支持导入它。 但是，如果您已经配置了运行时或打包工具来处理它，您可以使用新的 `--allowArbitraryExtensions` 编译器选项来抑制错误。

需要注意的是，历史上通常可以通过添加名为 `app.css.d.ts` 而不是 `app.d.css.ts` 的声明文件来实现类似的效果 - 但是，这只在 Node.js 中 CommonJS 的 `require` 解析规则下可以工作。 严格来说，前者被解析为名为 `app.css.js` 的 JavaScript 文件的声明文件。 由于 Node 中的 ESM 需要使用包含扩展名的相对文件导入，因此在 `--`

moduleResolution 为 node16 或 nodenext 时, TypeScript 会在示例的 ESM 文件中报错。

更多详情请参考 [PR](#) [PR](#)。

28.8.5 customConditions

--customConditions 接受额外的[条件](#)列表, 当 TypeScript 从 package.json 的 [exports](#) 或 [imports](#) 字段解析时, 这些条件应该成功。 这些条件会被添加到解析器默认使用的任何现有条件中。

例如, 有如下的配置:

```
{
  "compilerOptions": {
    "target": "es2022",
    "moduleResolution": "bundler",
    "customConditions": ["my-condition"]
  }
}
```

每当 package.json 里引用了 exports 或 imports 字段时, TypeScript 都会考虑名为 my-condition 的条件。

所以当从具有如下 package.json 的包中导入时:

```
{
  // ...
  "exports": {
    ".": {
      "my-condition": "./foo.mjs",
      "node": "./bar.mjs",
      "import": "./baz.mjs",
      "require": "./biz.mjs"
    }
  }
}
```

TypeScript 会尝试查找 foo.mjs 文件。

该字段仅在 --moduleResolution 为 node16, nodenext 和 bundler 时有效。

28.9 --verbatimModuleSyntax

在默认情况下, TypeScript 会执行[导入省略](#)。 大体上来讲, 如果有如下代码:

```
import { Car } from './car';

export function drive(car: Car) {
  // ...
}
```

TypeScript 能够检测到导入语句仅用于导入类型, 因此会删除导入语句。 最终生成的 JavaScript 代码如下:

```
export function drive(car) {
  // ...
}
```

大多数情况下这是没问题的, 因为如果 Car 不是从 ./car 导出的值, 我们将会得到一个运行时错误。

但在一些特殊情况下, 它增加了一层复杂性。 例如, 不存在像 import "./car"; 这样的语句 - 这个导入语句会被完全删除。 这对于有副作用的模块来讲是有区别的。

TypeScript 的 JavaScript 代码生成策略还有其它一些复杂性 - 导入省略不仅只是由导入语句的使用方式决定 - 它还取决于值的声明方式。 因此, 如下的代码的处理方式不总是那么明显:

```
export { Car } from './car';
```

这段代码是应该保留还是删除? 如果 Car 是使用 class 声明的, 那么在生成的 JavaScript 代码中会被保留。 但是如果 Car 是使用类型别名或 interface 声明的, 那么在生成的 JavaScript 代码中会被省略。

尽管 TypeScript 可以根据多个文件来综合判断如何生成代码, 但不是所有的编译器都能够做到。

导入和导出语句中的 type 修饰符能够起到一点作用。 我们可以使用 type 修饰符明确声明导入和导出是否仅用于类型分析, 并且可以在生成的 JavaScript 文件中完全删除。

```
// This statement can be dropped entirely in JS output
import type * as car from './car';
```

// The named import/export 'Car' can be dropped in JS output

```
import { type Car } from './car';
export { type Car } from './car';
```

type 修饰符本身并不是特别管用 - 默认情况下, 导入省略仍会删除导入语句, 并且不强制要求您区分类型导入和普通导入以及导出。 因此, TypeScript 提供了 --importsNotUsedAsValues 来确保您使用类型修饰符, --preserveValueImports 来防止某些模块消除行为, 以及 --isolatedModules 来确保您的 TypeScript 代码在不同编译器中都能正常运行。 不幸的是, 理解这三个标志的细节很困难, 并且仍然存在一些意外行为的边缘情况。

TypeScript 5.0 提供了一个新的 --verbatimModuleSyntax 来简化这个情况。 规则很简单 - 所有不带 type 修饰符的导入导出语句会被保留。 任何带有 type 修饰符的导入导出语句会被删除。

```
// Erased away entirely.
import type { A } from 'a';
```

```
// Rewritten to 'import { b } from "bcd";'
import { b, type c, type d } from 'bcd';
```

```
// Rewritten to 'import {} from "xyz";'
import { type xyz } from 'xyz';
```

使用这个新的选项, 实现了所见即所得。

但是, 这在涉及模块互操作性时会有一些影响。 在这个标志下, 当您的设置或文件扩展名暗示了不同的模块系统时, ECMAScript 的导入和导出不会被重写为 require 调用。 相反, 您会收到一个错误。 如果您需要生成使用 require 和 module.exports 的代码, 您需要使用早于 ES2015 的 TypeScript 的模块语法:

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

// ==>

```
const foo = require('foo');
function foo() {}
function bar() {}
function baz() {}
```

```
ts_upgrade_from_2.2_to_5.3
    value = formatter.format(value);
}
```

```
    console.log(value);
}
```

现在不论是编写 TypeScript 文件还是 JavaScript 文件，TypeScript 都能够提示函数调用是否正确。

```
// all allowed
printValue('hello!');
printValue(123.45);
printValue(123.45, 2);
```

```
printValue('hello!', 123); // error!
```

更多详情请参考 [PR](#)，感谢 [Tomasz Lenarcik](#)。

28.13 在 `--build` 模式下使用有关文件生成的选项

TypeScript 现在允许在 `--build` 模式下使用如下选项：

- `--declaration`
- `--emitDeclarationOnly`
- `--declarationMap`
- `--sourceMap`
- `--inlineSourceMap`

这使得在构建过程中定制某些部分变得更加容易，特别是在你可能会有不同的开发和生产构建时。

例如，一个库的开发构建可能不需要生成声明文件，但是生产构建则需要。一个项目可以将生成声明文件配置为默认关闭，并使用如下方式构建：

```
tsc --build -p ./my-project-dir
```

开发完毕后，在“生产环境”构建时使用 `--declaration` 选项：

```
tsc --build -p ./my-project-dir --declaration
```

更多详情请参考 [PR](#)。

28.14 编辑器导入语句排序时不区分大小写

在 Visual Studio 和 VS Code 等编辑器中，TypeScript 可以帮助组织和排序导入和导出语句。不过，通常情况下，对于何时将列表“排序”，可能会有不同的解释。

例如，下面的导入列表是否已排序？

```
import { Toggle, freeze, toBoolean } from './utils';
```

令人惊讶的是，答案可能是“这要看情况”。如果我们不考虑大小写敏感性，那么这个列表显然是没有排序的。字母 `f` 排在 `t` 和 `T` 之前。

但在大多数编程语言中，排序默认是比较字符串的字节值。JavaScript 比较字符串的方式意味着“Toggle”总是排在“freeze”之前，因为根据 [ASCII 字符编码](#)，大写字母排在小写字母之前。所以从这个角度来看，导入列表是已排序的。

以前，TypeScript 认为导入列表已排序，因为它进行了基本的大小写敏感排序。这可能让开发人员感到沮丧，因为他们更喜欢不区分大小写的排序方式，或者使用像 ESLint 这样的工具默认需要不区分大小写的排序方式。

现在，TypeScript 默认会检测大小写敏感性。这意味着 TypeScript 和类似 ESLint 的工具通常不会因为如何最好地排序导入而“互相冲突”。

我们的团队还在尝试更多的排序策略，你可以在[这里了解更多](#)。这些选项可能最终可以由编辑器进行配置。目前，它们仍然不稳定和实验性的，你可以通过在 JSON 选项中使用

```
}
```

更多详情请参考 [PR](#)。

28.11 支持 JSDoc 中的 `@satisfies`

TypeScript 4.9 支持 `satisfies` 运算符。它确保了表达式的类型是兼容的，且不影响类型自身。例如，有如下代码：

```
interface CompilerOptions {
    strict?: boolean;
    outDir?: string;
    // ...
}
```

```
interface ConfigSettings {
    compilerOptions?: CompilerOptions;
    extends?: string | string[];
    // ...
}
```

```
let myConfigSettings = {
    compilerOptions: {
        strict: true,
        outDir: '../lib',
        // ...
    },
```

```
    extends: ['@tsconfig/strictest/tsconfig.json',
        '../.../tsconfig.base.json'],
} satisfies ConfigSettings;
```

这里，TypeScript 知道 `myConfigSettings.extends` 声明为数组 - 因为 `satisfies` 会验证对象的类型。因此，如果我们想在 `extends` 上进行映射操作，那是可以的。

```
declare function resolveConfig(configPath: string): CompilerOptions;
```

```
let inheritedConfigs = myConfigSettings.extends.map(resolveConfig);
```

这对 TypeScript 用户来讲是有用的，但是许多人使用 TypeScript 来对带有 JSDoc 的 JavaScript 代码进行类型检查。因此，TypeScript 5.0 支持了新的 JSDoc 标签 `@satisfies` 来做相同的事。

```
/** @satisfies */ 能够检查出类型不匹配：
```

```
// @ts-check
```

```
/**
 * @typedef CompilerOptions
 * @prop {boolean} [strict]
 * @prop {string} [outDir]
 */
```

```
/**
 * @satisfies {CompilerOptions}
 */
```

v5.0 => 支持 JSDoc 中的 `@satisfies`

```
ts_upgrade_from_2.2_to_5.3
let myCompilerOptions = {
  outdir: '../lib',
  // ~~~~~~ oops! we meant outDir
};
但它会保留表达式的原始类型，允许我们稍后使用值的更详细的类型。
// @ts-check
```

```
/**
 * @typedef CompilerOptions
 * @prop {boolean} [strict]
 * @prop {string} [outDir]
 */

/**
 * @typedef ConfigSettings
 * @prop {CompilerOptions} [compilerOptions]
 * @prop {string | string[]} [extends]
 */

/**
 * @satisfies {ConfigSettings}
 */
let myConfigSettings = {
  compilerOptions: {
    strict: true,
    outDir: '../lib',
  },
  extends: ['@tsconfig/strictest/tsconfig.json',
    '../.../tsconfig.base.json'],
};

let inheritedConfigs = myConfigSettings.extends.map(resolveConfig);
/** @satisfies */ 也可以在行内的括号表达式上使用。 可以像下面这样定义 myConfigSettings:
let myConfigSettings = /** @satisfies {ConfigSettings} */ {
  compilerOptions: {
    strict: true,
    outDir: '../lib',
  },
  extends: ['@tsconfig/strictest/tsconfig.json',
    '../.../tsconfig.base.json'],
};
为什么？当你更深入地研究其他代码时，比如函数调用，它通常更有意义。
compileCode(
  /** @satisfies {ConfigSettings} */ {
    // ...
  }
);
更多详情请参考 PR。 感谢作者 Oleksandr Tarasiuk。
```

28.12 支持 JSDoc 中的 @overload

在 TypeScript 中，你可以为一个函数指定多个重载。使用重载能够描述一个函数可以使用不同的参数进行调用，也可能返回不同的结果。它们可以限制调用方如何调用函数，并细化他们将得到的结果。

```
// Our overloads:
function printValue(str: string): void;
function printValue(num: number, maxFractionDigits?: number): void;
```

```
// Our implementation:
function printValue(value: string | number, maximumFractionDigits?: number) {
  if (typeof value === 'number') {
    const formatter = Intl.NumberFormat('en-US', {
      maximumFractionDigits,
    });
    value = formatter.format(value);
  }
```

```
    console.log(value);
}
```

这里表示 printValue 的第一个参数可以为 string 或 number 类型。如果接收的是 number 类型，那么它还接收第二个参数决定打印的小数位数。

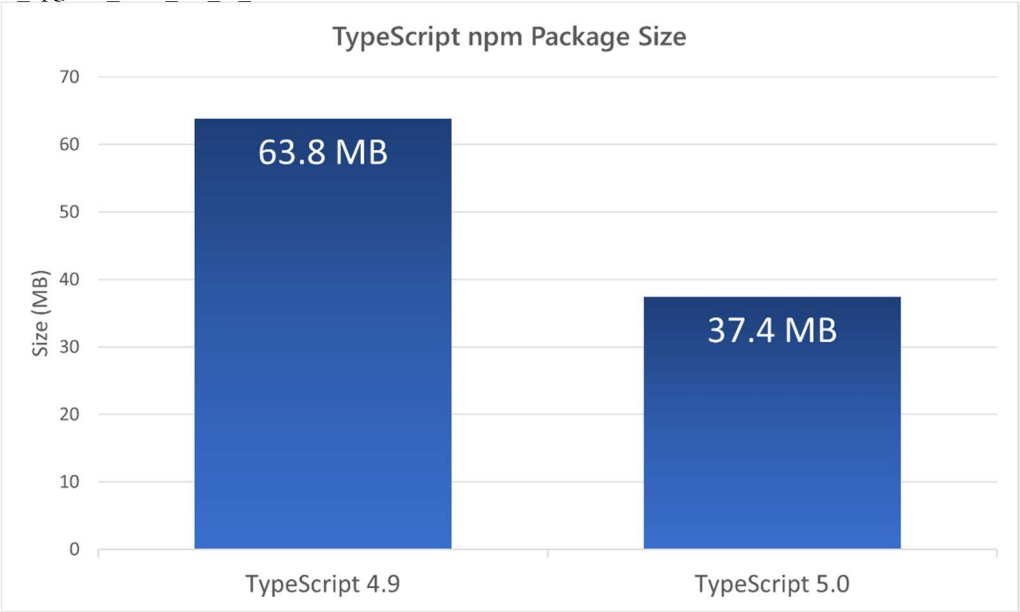
TypeScript 5.0 支持在 JSDoc 里使用 @overload 来声明重载。每一个 JSDoc @overload 标记都表示一个不同的函数重载。

```
// @ts-check
```

```
/**
 * @overload
 * @param {string} value
 * @return {void}
 */
```

```
/**
 * @overload
 * @param {number} value
 * @param {number} [maximumFractionDigits]
 * @return {void}
 */
```

```
/**
 * @param {string | number} value
 * @param {number} [maximumFractionDigits]
 */
function printValue(value, maximumFractionDigits) {
  if (typeof value === 'number') {
    const formatter = Intl.NumberFormat('en-US', {
      maximumFractionDigits,
    });
```



怎么做到的呢？我们将在未来的博客文章中详细介绍一些值得注意的改进。但我们不会让你等到那篇博客文章。

首先，我们最近将 TypeScript 从命名空间迁移到了模块，这使我们能够利用现代构建工具来执行像作用域提升这样的优化。使用这些工具，重新审视我们的打包策略，并删除一些已过时的代码，使 TypeScript 4.9 的 63.8 MB 包大小减少了约 26.4 MB。这也通过直接函数调用为我们带来了显著的加速。我们在这里撰写了关于我们迁移到模块的[详细介绍](#)。

TypeScript 还在编译器内部对象类型上增加了更多的一致性，并且也减少了一些这些对象类型上存储的数据。这减少了多态操作，同时平衡了由于使我们的对象结构更加一致而带来的内存使用增加。

我们还在将信息序列化为字符串时执行了一些缓存。类型显示，它可能在错误报告、声明生成、代码补全等情况下使用，是非常昂贵的操作。TypeScript 现在对一些常用的机制进行缓存，以便在这些操作之间重复使用。

我们进行了一个值得注意的改变，改善了我们的解析器，即在某些情况下，利用 var 来避免在闭包中使用 let 和 const 的成本。这提高了一些解析性能。

总的来说，我们预计大多数代码库应该会从 TypeScript 5.0 中看到速度的提升，并且一直能够保持 10% 到 20% 之间的优势。当然，这将取决于硬件和代码库的特性，但我们鼓励你今天就在你的代码库上尝试它！

更多详情：

- [Migrate to Modules](#)
- [Node Monomorphization](#)
- [Symbol Monomorphization](#)
- [Identifier Size Reduction](#)
- [Printer Caching](#)
- [Limited Usage of var](#)

typescript.unstable 条目来选择它们。下面是你可以尝试的所有选项（设置为它们的默认值）：

```
{
  "typescript.unstable": {
    // Should sorting be case-sensitive? Can be:
    // - true
    // - false
    // - "auto" (auto-detect)
    "organizeImportsIgnoreCase": "auto",

    // Should sorting be "ordinal" and use code points or consider
    // Unicode rules? Can be:
    // - "ordinal"
    // - "unicode"
    "organizeImportsCollation": "ordinal",

    // Under `organizeImportsCollation: "unicode`,
    // what is the current locale? Can be:
    // - [any other locale code]
    // - "auto" (use the editor's locale)
    "organizeImportsLocale": "en",

    // Under `organizeImportsCollation: "unicode`,
    // should upper-case letters or lower-case letters come first? Can
    // be:
    // - false (locale-specific)
    // - "upper"
    // - "lower"
    "organizeImportsCaseFirst": false,

    // Under `organizeImportsCollation: "unicode`,
    // do runs of numbers get compared numerically (i.e. "a1" < "a2" <
    // "a100")? Can be:
    // - true
    // - false
    "organizeImportsNumericCollation": true,

    // Under `organizeImportsCollation: "unicode`,
    // do letters with accent marks/diacritics get sorted distinctly
    // from their "base" letter (i.e. is é different from e)? Can be
    // - true
    // - false
    "organizeImportsAccentCollation": true
  },
  "javascript.unstable": {
    // same options valid here...
  }
}
```

更多详情请参考 [PR](#) 和 [PR](#)。

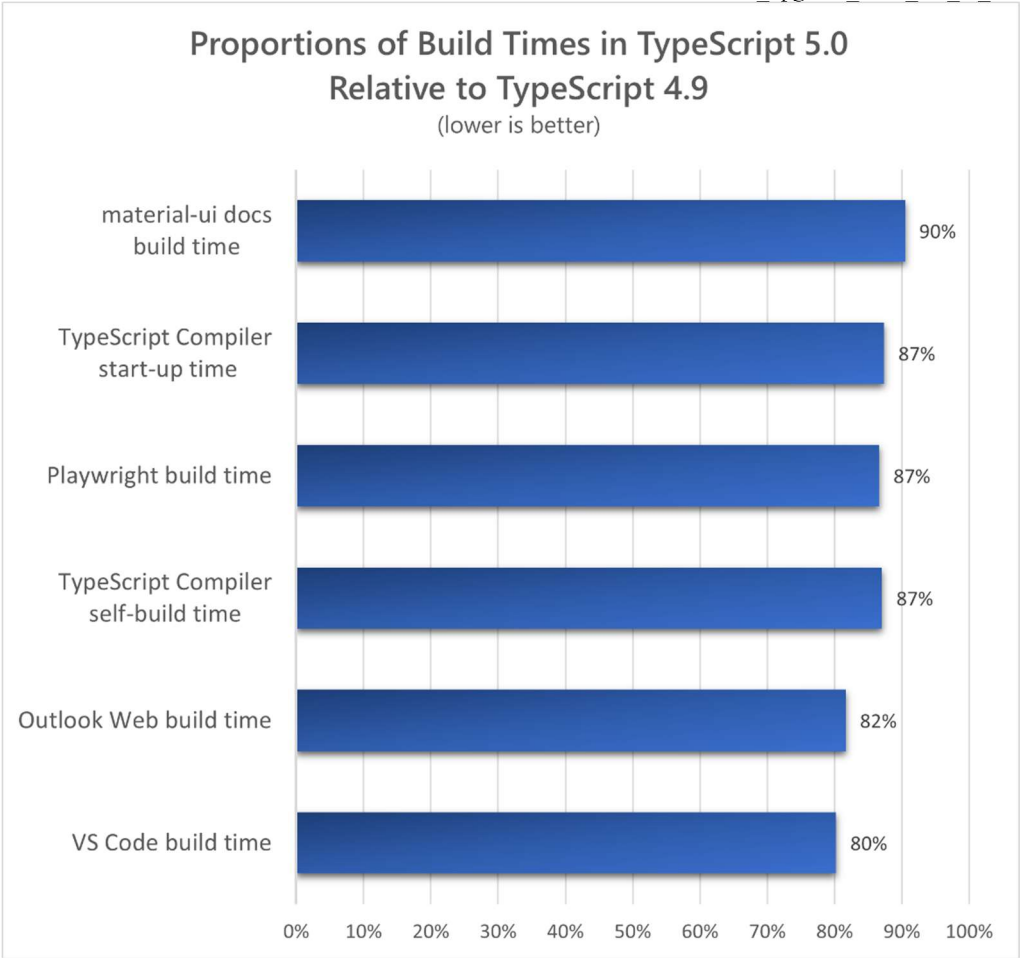
28.15 穷举式 switch/case 自动补全

在编写 switch 语句时，TypeScript 现在会检测被检查的值是否具有字面量类型。 如果是，它将提供一个补全选项，可以为每个未覆盖的情况构建骨架代码。
更多详情请参考 [PR](#)。

28.16 速度，内存以及代码包尺寸优化

TypeScript 5.0 在我们的代码结构、数据结构和算法实现方面进行了许多强大的变化。 这些变化的意义在于，整个体验都应该更快 — 不仅仅是运行 TypeScript，甚至包括安装 TypeScript。
以下是我们相对于 TypeScript 4.9 能够获得的一些有趣的速度和大小优势。

Scenario	Time or Size Relative to TS 4.9
material-ui build time	90%
TypeScript Compiler startup time	89%
Playwright build time	88%
TypeScript Compiler self-build time	87%
Outlook Web build time	82%
VS Code build time	80%
typescript npm Package Size	59%



ts_upgrade_from_2.2_to_5.3

```
class SafeBox {
  #value: string | undefined;

  // Only accepts strings!
  set value(newValue: string) {}

  // Must check for 'undefined'!
  get value(): string | undefined {
    return this.#value;
  }
}
```

实际上，这与在 `--exactOptionalProperties` 选项下可选属性的检查方式类似。
更多详情请参考 [PR](#)。

29.3 解耦 JSX 元素和 JSX 标签类型之间的类型检查

TypeScript 在 JSX 方面的一个痛点是对每个 JSX 元素标签的类型要求。这个 TypeScript 版本使得 JSX 库更准确地描述了 JSX 组件可以返回的内容。对于许多人来说，这具体意味着可以在 React 中使用 [异步服务器组件](#)。

做为背景知识，JSX 元素是下列其一：

```
// A self-closing JSX tag
<Foo />
```

```
// A regular element with an opening/closing tag
<Bar></Bar>
```

在类型检查 `<Foo />` 或 `<Bar></Bar>` 时，TypeScript 总是查找名为 JSX 的命名空间，并且获取名为 `Element` 的类型。换句话说，它查找 `JSX.Element`。

但是为了检查 `Foo` 或 `Bar` 是否是有效的标签名，TypeScript 大致上只需获取

由 `Foo` 或 `Bar` 返回或构造的类型，并检查其与 `JSX.Element`（或另一种叫

做 `JSX.ElementClass` 的类型，如果该类型可构造）的兼容性。

这里的限制意味着如果组件返回或“render”比 `JSX.Element` 更宽泛的类型，则无法使用组件。例如，一个 JSX 库可能会允许组件返回 `strings` 或 `Promises`。

作为一个更具体的例子，[未来版本](#)的 React 已经提出了对返回 `Promise` 的组件的有限支持，但是现有版本的 TypeScript 无法表达这一点，除非有人大幅放宽 `JSX.Element` 类型。

```
import * as React from 'react';
```

```
async function Foo() {
  return <div></div>;
}
```

```
let element = <Foo />;
//      ~~~
// 'Foo' cannot be used as a JSX component.
// Its return type 'Promise<Element>' is not a valid JSX element.
为了给 library 提供一种表达这种情况的方法，TypeScript 5.1 现在查找一个名为 JSX.ElementType 的类型。ElementType 精确地指定了在 JSX 元素中作为标签使用的内容。因此现在可以像如下这样定义：
```

```
namespace JSX {
  export type ElementType =
```

29 v5.1

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-5.1.html>

29.1 更易用的隐式返回 undefined 的函数

在 JavaScript 中，如果一个函数运行结束时没有遇到 `return` 语句，它会返回 `undefined` 值。

```
function foo() {
  // no return
}
```

```
// x = undefined
```

```
let x = foo();
```

然而，在之前版本的 TypeScript 中，只有返回值类型为 `void` 和 `any` 的函数可以不带 `return` 语句。这意味着，就算明知函数返回 `undefined`，你也必须包含 `return` 语句。

```
// fine - we inferred that 'f1' returns 'void'
function f1() {
  // no returns
}
```

```
// fine - 'void' doesn't need a return statement
function f2(): void {
  // no returns
}
```

```
// fine - 'any' doesn't need a return statement
function f3(): any {
  // no returns
}
```

```
// error!
// A function whose declared type is neither 'void' nor 'any' must
return a value.
function f4(): undefined {
  // no returns
}
```

如果某些 API 期望函数返回 `undefined` 值，这可能会让人感到痛苦 —— 你需要至少有一个显式的返回 `undefined` 语句，或者一个带有显式注释的 `return` 语句。

```
declare function takesFunction(f: () => undefined): undefined;
```

```
// error!
// Argument of type '() => void' is not assignable to parameter of
type '() => undefined'.
takesFunction(() => {
  // no returns
});
```

```
// error!
```

// A function whose declared type is neither 'void' nor 'any' must return a value.

```
takesFunction(): undefined => {
  // no returns
};
```

// error!
// Argument of type '() => void' is not assignable to parameter of type '() => undefined'.

```
takesFunction() => {
  return;
};
```

// works

```
takesFunction() => {
  return undefined;
};
```

// works

```
takesFunction(): undefined => {
  return;
};
```

这种行为非常令人沮丧和困惑，尤其是在调用自己无法控制的函数时。理解推断 `void` 与 `undefined` 之间的相互作用，以及一个返回 `undefined` 的函数是否需要 `return` 语句等等，似乎会分散注意力。

首先，TypeScript 5.1 允许返回 `undefined` 的函数不包含返回语句。

```
// Works in TypeScript 5.1!
function f4(): undefined {
  // no returns
}
```

// Works in TypeScript 5.1!

```
takesFunction(): undefined => {
  // no returns
};
```

其次，如果一个函数没有返回表达式，并且被传递给期望返回 `undefined` 值的函数的地方，TypeScript 会推断该函数的返回类型为 `undefined`。

```
// Works in TypeScript 5.1!
takesFunction(function f() {
  // ^ return type is undefined
  // no returns
});
```

// Works in TypeScript 5.1!

```
takesFunction(function f() {
  // ^ return type is undefined

  return;
});
```

为了解决另一个类似的痛点，在 TypeScript 的 `--noImplicitReturns` 选项下，只返回 `undefined` 的函数现在有了类似于 `void` 的例外情况，在这种情况下，并不是每个代码路径都必须以显式的返回语句结束。

```
// Works in TypeScript 5.1 under '--noImplicitReturns'!
function f(): undefined {
  if (Math.random()) {
    // do some stuff...
    return;
  }
}
```

更多详情请参考 [Issue](#), [PR](#)

29.2 不相关的存取器类型

TypeScript 4.3 支持将成对的 `get` 和 `set` 定义为不同的类型。

```
interface Serializer {
  set value(v: string | number | boolean);
  get value(): string;
}
```

```
declare let box: Serializer;
```

```
// Allows writing a 'boolean'
box.value = true;
```

```
// Comes out as a 'string'
console.log(box.value.toUpperCase());
```

最初，我们要求 `get` 的类型是 `set` 类型的子类型。这意味着：
`box.value = box.value;`
永远是合法的。

然而，大量现存的和提议的 API 带有毫无关联的 `get` 和 `set` 类型。例如，考虑一个常见的情况 - DOM 中的 `style` 属性和 `CSSStyleRule` API。每条样式规则都有一个 [style 属性](#)，它是一个 `CSSStyleDeclaration`；然而，如果你尝试给该属性写值，它仅支持字符串。

TypeScript 5.1 现在允许为 `get` 和 `set` 访问器属性指定完全不相关的类型，前提是它们具有显式的类型注解。虽然这个版本的 TypeScript 还没有改变这些内置接口的类型，但 `CSSStyleRule` 现在可以按以下方式定义：

```
interface CSSStyleRule {
  // ...

  /** Always reads as a `CSSStyleDeclaration` */
  get style(): CSSStyleDeclaration;

  /** Can only write a `string` here. */
  set style(newValue: string);

  // ...
}
```

这也允许其他模式，比如要求 `set` 访问器只接受“有效”的数据，但指定 `get` 访问器可以返回 `undefined`，如果某些基础状态还没有被初始化。

30 v5.2

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-5.2.html>

30.1 using 声明与显式资源管理

TypeScript 5.2 支持了 ECMAScript 即将引入的新功能 [显式资源管理](#)。让我们探索一下引入该功能的一些动机，并理解这个功能给我们带来了什么。

在创建对象之后需要进行某种形式的“清理”是很常见的。例如，您可能需要关闭网络连接，删除临时文件，或者只是释放一些内存。让我们来想象一个函数，它创建一个临时文件，对它进行多种操作的读写，然后关闭并删除它。

```
import * as fs from 'fs';

export function doSomeWork() {
  const path = '.some_temp_file';
  const file = fs.openSync(path, 'w+');

  // use file...

  // Close the file and delete it.
  fs.closeSync(file);
  fs.unlinkSync(path);
}

// 这看起来不错，但如果需要提前退出会发生什么？
export function doSomeWork() {
  const path = '.some_temp_file';
  const file = fs.openSync(path, 'w+');

  // use file...
  if (someCondition()) {
    // do some more work...

    // Close the file and delete it.
    fs.closeSync(file);
    fs.unlinkSync(path);
    return;
  }
}
```

```
// Close the file and delete it.
fs.closeSync(file);
fs.unlinkSync(path);
}
```

我们可以看到存在重复的容易忘记的清理代码。同时无法保证在代码抛出异常时，关闭和删除文件会被执行。解决办法是用 try/finally 语句包裹整段代码。

```
export function doSomeWork() {
  const path = '.some_temp_file';
  const file = fs.openSync(path, 'w+');

  try {
    // use file...
```

```
// All the valid lowercase tags
keyof IntrinsicAttributes
// Function components
(props: any) => Element
// Class components
new (props: any) => ElementClass;
```

```
export interface IntrinsicAttributes extends /*...*/ {}
export type Element = /*...*/;
export type ClassElement = /*...*/;
```

```
}
```

感谢 [Sebastian Silberman](#) 的 [PR](#)。

29.4 带有命名空间的 JSX 属性

TypeScript 支持在 JSX 里使用带有命名空间的属性。

```
import * as React from "react";
```

```
// Both of these are equivalent:
const x = <Foo a:b="hello" />;
const y = <Foo a : b="hello" />;
```

```
interface FooProps {
  "a:b": string;
}
```

```
function Foo(props: FooProps) {
  return <div>{props["a:b"]}</div>;
}
```

当名字的第一段是小写名称时，在 JSX.IntrinsicAttributes 上查找带命名空间的标记名是类似的。

// In some library's code or in an augmentation of that library:

```
namespace JSX {
  interface IntrinsicElements {
    ['a:b']: { prop: string };
  }
}
```

// In our code:

```
let x = <a:b prop="hello!" />;
```

感谢 [Oleksandr Tarasiuk](#) 的 [PR](#)。

29.5 模块解析时考虑 typeRoots

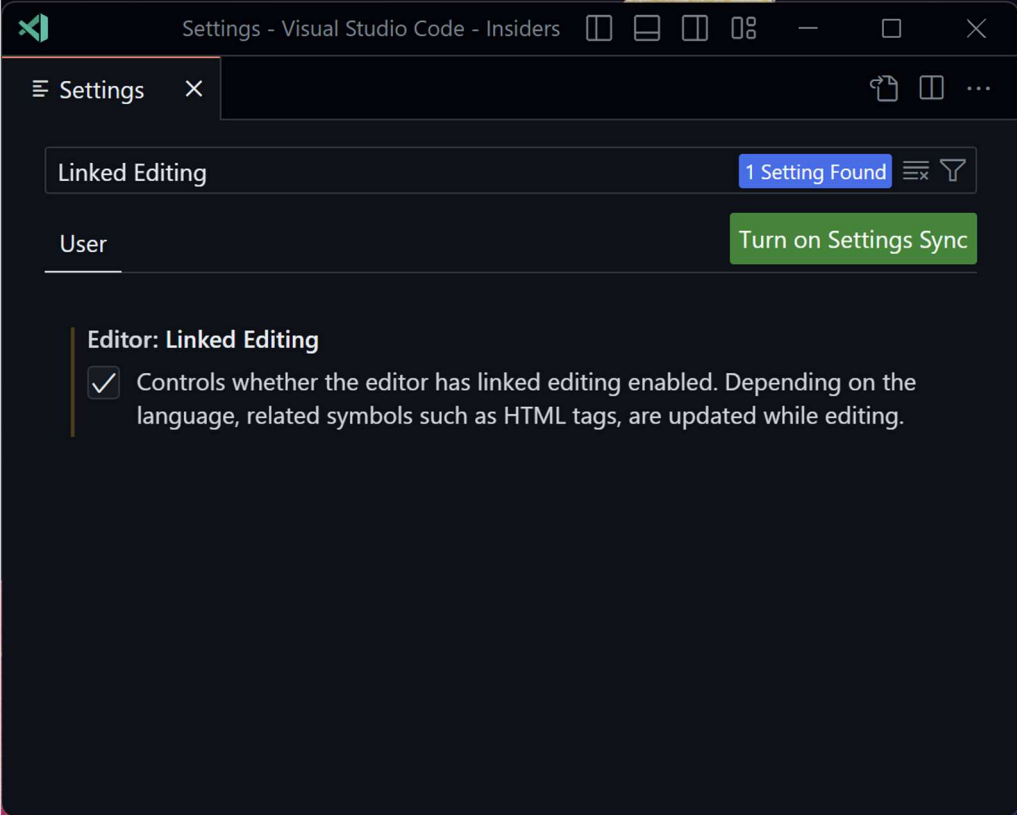
当 TypeScript 的模块解析策略无法解析一个路径时，它现在会相对于 typeRoots 继续解析。

更多详情请参考 [PR](#)。

29.6 在 JSX 标签上链接光标

TypeScript 现在支持 [链接编辑](#) JSX 标签名。链接编辑（有时称作“光标镜像”）允许编辑器同时自动编辑多个位置。

这个新特性在 TypeScript 和 JavaScript 里都可用，并且可以在 Visual Studio Code Insiders 版本中启用。在 Visual Studio Code 里，你既可以用设置界面的 Editor: Linked Editing 配置：



也可以用 JSON 配置文件中的 editor.linkedEditing:

```
{
  // ...
  "editor.linkedEditing": true
}
```

这个功能也将在 Visual Studio 17.7 Preview 1 中得到支持。

29.7 [@param JSDoc 标记的代码片段自动补全](#)

现在，在 TypeScript 和 JavaScript 文件中输入 @param 标签时，TypeScript 提供代码片段自动补全。这可以帮助在为代码编写文档和添加 JSDoc 类型时，减少打字和文本跳转次数。

更多详情请参考 [PR](#)。

29.8 [优化](#)

29.8.1 [避免非必要的类型初始化](#)

TypeScript 5.1 现在避免在已知不包含对外部类型参数的引用的对象类型中执行类型实例化。这有可能减少许多不必要的计算，并将 `material-ui` 的文档目录的类型检查时间缩短了 50% 以上。

更多详情请参考 [PR](#)。

29.8.2 [联合字面量的反面情况检查](#)

当检查源类型是否是联合类型的一部分时，TypeScript 首先使用该源类型的内部类型标识符进行快速查找。如果查找失败，则 TypeScript 会检查与联合类型中的每个类型的兼容性。当将字面量类型与纯字面量类型的联合类型进行关联时，TypeScript 现在可以避免针对联合中的每个其他类型进行完整遍历。这个假设是安全的，因为 TypeScript 总是将字面量类型内部化/缓存 — 虽然有一些与“全新”字面量类型相关的边缘情况需要处理。

[这个优化](#)可以减少[问题代码](#)的类型检查时间从 45 秒到 0.4 秒。

29.8.3 [减少在解析 JSDoc 时的扫描函数调用](#)

在旧版本的 TypeScript 中解析 JSDoc 注释时，它们会使用扫描器/标记化程序将注释分解为细粒度的标记，然后将内容拼回到一起。这对于规范化注释文本可能是有帮助的，使多个空格只折叠成一个；但这样做会极大地增加“对话”量，意味着解析器和扫描器会非常频繁地来回跳跃，从而增加了 JSDoc 解析的开销。

TypeScript 5.1 已经移动了更多的逻辑来分解 JSDoc 注释到扫描器/标记化程序中。现在，扫描器直接将更大的内容块返回给解析器，以便根据需要进行处理。

[这些更改](#)将几个大约 10Mb 的大部分为散文评论的 JavaScript 文件的解析时间减少了约一半。对于一个更现实的例子，我们的性能套件对 [xstate](#) 的快照减少了约 300 毫秒的解析时间，使其更快地加载和分析。

ts_upgrade_from_2.2_to_5.3

你可能已经注意到了，在这些例子中使用的都是同步方法。然而，很多资源释放的场景涉及到异步操作，我们需要等待它们完成才能进行后续的操作。

这就是为什么现在还有一个新的 `Symbol.asyncDispose`，它带来了另一个亮点 - `await using` 声明。它与 `using` 声明相似，但关键是它查找需要 `await` 的资源。它使用名为 `Symbol.asyncDispose` 的方法，尽管它们也可以操作在任何具有 `Symbol.dispose` 的对象上操作。为了方便，TypeScript 引入了全局类型 `AsyncDisposable` 用来表示拥有异步 `dispose` 方法的对象。

```
async function doWork() {
    // Do fake work for half a second.
    await new Promise(resolve => setTimeout(resolve, 500));
}
```

```
function loggy(id: string): AsyncDisposable {
    console.log(`Constructing ${id}`);
    return {
        async [Symbol.asyncDispose]() {
            console.log(`Disposing (async) ${id}`);
            await doWork();
        },
    }
}
```

```
async function func() {
    await using a = loggy("a");
    await using b = loggy("b");
    {
        await using c = loggy("c");
        await using d = loggy("d");
    }
    await using e = loggy("e");
    return;

    // Unreachable.
    // Never created, never disposed.
    await using f = loggy("f");
}
```

```
func();
// Constructing a
// Constructing b
// Constructing c
// Constructing d
// Disposing (async) d
// Disposing (async) c
// Constructing e
// Disposing (async) e
// Disposing (async) b
// Disposing (async) a
```

```
if (someCondition()) {
    // do some more work...
    return;
} finally {
    // Close the file and delete it.
    fs.closeSync(file);
    fs.unlinkSync(path);
}
```

虽说这样写更加健壮，但是也为我们的代码增加了一些“噪音”。如果我们在 `finally` 块中开始添加更多的清理逻辑，还可能遇到其他的自食其果的问题。例如，异常可能会阻止其他资源的释放。这些就是[显式资源管理](#)想要解决的问题。该提案的关键思想是将资源释放（我们试图处理的清理工作）作为 JavaScript 中的一等概念来支持。

首先，增加了一个新的 `symbol` 名字为 `Symbol.dispose`，然后可以定义包含 `Symbol.dispose` 方法的对象。为了方便，TypeScript 为此定义了一个新的全局类型 `Disposable`。

```
class TempFile implements Disposable {
    #path: string;
    #handle: number;

    constructor(path: string) {
        this.#path = path;
        this.#handle = fs.openSync(path, 'w+');
    }

    // other methods

    [Symbol.dispose]() {
        // Close the file and delete it.
        fs.closeSync(this.#handle);
        fs.unlinkSync(this.#path);
    }
}
```

之后可以调用这些方法

```
export function doSomeWork() {
    const file = new TempFile('.some_temp_file');

    try {
        // ...
    } finally {
        file[Symbol.dispose]();
    }
}
```

将清理逻辑移动到 `TempFile` 本身没有带来多大的价值；仅仅是将清理的代码从 `finally` 提取到方法而已，你总是可以这样做。但如果该方法有一个众所周知的名字那么 JavaScript 就可以基于此构造其它功能。

ts_upgrade_from_2.2_to_5.3

这将引出该功能的第一个亮点: `using` 声明! `using` 是一个新的关键字, 支持声明新的不可变绑定, 像 `const` 一样。不同点是 `using` 声明的变量在即将离开其作用域时, 它的 `Symbol.dispose` 方法会被调用! 因此, 我们可以这样编写代码:

```
export function doSomeWork() {
    using file = new TempFile(".some_temp_file");

    // use file...

    if (someCondition()) {
        // do some more work...
        return;
    }
}
```

看一下 - 没有 `try / finally` 代码块! 至少, 我们没有见到。从功能上讲, 这些正是 `using` 声明要帮我们做的事, 但我们不必自己处理它。

你可能熟悉 C# 中的 `using`, Python 中的 `with`, Java 中的 `try-with-resource` 声明。这些与 JavaScript 中的 `using` 关键字是相似的, 都提供了一种明确的方式来“清理”对象, 在它们即将离开作用域时。

`using` 声明在其所在的作用域的最后才执行清理工作, 或在“提前返回”(如 `return` 语句或 `throw` 错误)之前执行清理工作。释放的顺序是先入后出, 像栈一样。

```
function loggy(id: string): Disposable {
    console.log(`Creating ${id}`);

    return {
        [Symbol.dispose]() {
            console.log(`Disposing ${id}`);
        }
    }
}
```

```
function func() {
    using a = loggy("a");
    using b = loggy("b");
    {
        using c = loggy("c");
        using d = loggy("d");
    }
    using e = loggy("e");
    return;

    // Unreachable.
    // Never created, never disposed.
    using f = loggy("f");
}
```

```
func();
// Creating a
```

```
// Creating b
// Creating c
// Creating d
// Disposing d
// Disposing c
// Creating e
// Disposing e
// Disposing b
// Disposing a
```

`using` 声明对异常具有适应性; 如果抛出了一个错误, 那么在资源释放后会重新抛出错误。另一方面, 一个函数体可能正常执行, 但是 `Symbol.dispose` 可能抛出异常。这种情况下, 异常会被重新抛出。

但如果释放之前的逻辑以及释放时的逻辑都抛出了异常会发生什么? 为处理这类情况引入了一个新的类型 `SuppressedError`, 它是 `Error` 类型的子类型。 `SuppressedError` 类型的 `suppressed` 属性保存了上一个错误, 同时 `error` 属性保存了最后抛出的错误。

```
class ErrorA extends Error {
    name = "ErrorA";
}
class ErrorB extends Error {
    name = "ErrorB";
}
```

```
function throwy(id: string) {
    return {
        [Symbol.dispose]() {
            throw new ErrorA(`Error from ${id}`);
        }
    };
}
```

```
function func() {
    using a = throwy("a");
    throw new ErrorB("oops!")
}
```

```
try {
    func();
}
catch (e: any) {
    console.log(e.name); // SuppressedError
    console.log(e.message); // An error was suppressed during disposal.

    console.log(e.error.name); // ErrorA
    console.log(e.error.message); // Error from a

    console.log(e.suppressed.name); // ErrorB
    console.log(e.suppressed.message); // oops!
}
```

```

    constructor(firstName: string, lastName: string, age: number) {
        // ...
    }
}

```

此处的意图是，只有 `age` 和 `fullName` 可以被序列化，因为它们应用了 `@serialize` 装饰器。我们定义了 `toJSON` 方法来做这件事，但它只是调用了 `jsonfy`，它会使用 `@serialize` 创建的 `metadata`。

下面是 `./serialize.ts` 可能的定义：

```

const serializables = Symbol();

type Context =
    | ClassAccessorDecoratorContext
    | ClassGetterDecoratorContext
    | ClassFieldDecoratorContext;

export function serialize(_target: any, context: Context): void {
    if (context.static || context.private) {
        throw new Error('Can only serialize public instance members.');
```

```

    }
    if (typeof context.name === 'symbol') {
        throw new Error('Cannot serialize symbol-named properties.');
```

```

    }

    const propNames = ((context.metadata[serializables] as
        | string[]
        | undefined) ??= []);
    propNames.push(context.name);
}

export function jsonify(instance: object): string {
    const metadata = instance.constructor[Symbol.metadata];
    const propNames = metadata?.[serializables] as string[] | undefined;
    if (!propNames) {
        throw new Error('No members marked with @serialize.');
```

```

    }

    const pairStrings = propNames.map(key => {
        const strKey = JSON.stringify(key);
        const strValue = JSON.stringify((instance as any)[key]);
        return `${strKey}: ${strValue}`;
    });

    return `{ ${pairStrings.join(', ')} }`;
}

```

该方法有一个局部 `symbol` 名字为 `serializables` 用于保存和获取使用 `@serializable` 标记的属性。当每次调用 `@serializable` 时，它都会在 `metadata` 上保存这些属性名。当 `jsonfy` 被调用时，从 `metadata` 上获取属性列表，之后从实例上获取实际值，最后序列化名和值。

如果你期望其他人能够一致地执行清理逻辑，通过使

用 `Disposable` 和 `AsyncDisposable` 来定义类型可以使你的代码更易于使用。实际上，存在许多现有的类型，它们拥有 `dispose()` 或 `close()` 方法。例如，Visual Studio Code APIs 定义了 [自己的 Disposable 接口](#)。在浏览器和诸如 Node.js、Deno 和 Bun 等运行时中，API 也可以选择对已经具有清理方法（如文件句柄、连接等）的对象使

用 `Symbol.dispose` 和 `Symbol.asyncDispose`。

现在也许对于库来说这听起来很不错，但对于你的场景来说可能有些过于复杂。如果你需要进行大量的临时清理，创建一个新类型可能会引入过度抽象和关于最佳实践的问题。例如，再次以我们的 `TempFile` 示例为例。

```

class TempFile implements Disposable {
    #path: string;
    #handle: number;

    constructor(path: string) {
        this.#path = path;
        this.#handle = fs.openSync(path, "w+");
    }

    // other methods

    [Symbol.dispose]() {
        // Close the file and delete it.
        fs.closeSync(this.#handle);
        fs.unlinkSync(this.#path);
    }
}

export function doSomeWork() {
    using file = new TempFile(".some_temp_file");

    // use file...

    if (someCondition()) {
        // do some more work...
        return;
    }
}

```

我们只是想记住调用两个函数，但这是最好的写法吗？我们应该在构造函数中调

用 `openSync`，创建一个 `open()` 方法，还是自己传递句柄？我们是否应该为每个需要执行的操作公开一个方法，还是只将属性公开？

这就引出了这个特性的最后亮点：`DisposableStack` 和 `AsyncDisposableStack`。这些对象非常适用于一次性的清理工作，以及任意数量的清理工作。`DisposableStack` 是一个对象，它具有多个方法用于跟踪 `Disposable` 对象，并且可以接受函数来执行任意的清理工作。我们还可以将它们分配给 `using` 变量，因为它们也是 `Disposable`！所以下面是我们可以编写原始示例的方式。

```

function doSomeWork() {
    const path = ".some_temp_file";
    const file = fs.openSync(path, "w+");
}

```

```

using cleanup = new DisposableStack();
cleanup.defer(() => {
    fs.closeSync(file);
    fs.unlinkSync(path);
});

// use file...

if (someCondition()) {
    // do some more work...
    return;
}

// ...

```

在这里，`defer()` 方法只需要一个回调函数，该回调函数将在 `cleanup` 释放后运行。通常，在创建资源后应立即调用 `defer`（以及其他 `DisposableStack` 方法，如 `use` 和 `adopt`）。顾名思义，`DisposableStack` 以类似堆栈的方式处理它所跟踪的所有内容，按照先进后出的顺序进行处理，因此在创建值后立即进行 `defer` 处理有助于避免奇怪的依赖问题。`AsyncDisposableStack` 的工作原理类似，但可以跟踪异步函数和 `AsyncDisposable`，并且本身也是 `AsyncDisposable`。

在许多方面，`defer` 方法与 Go、Swift、Zig、Odin 等语言中的 `defer` 关键字类似，因此其使用约定应该相似。

由于这个特性非常新，大多数运行时环境不会原生支持它。要使用它，您需要为以下内容提供运行时的 `polyfills`：

- `Symbol.dispose`
- `Symbol.asyncDispose`
- `DisposableStack`
- `AsyncDisposableStack`
- `SuppressedError`

然而，如果您只对使用 `using` 和 `await using` 感兴趣，您只需要为内置的 `symbol` 提供 `polyfill`，通常以下简单的方法可适用于大多数情况：

```

Symbol.dispose ??= Symbol('Symbol.dispose');
Symbol.asyncDispose ??= Symbol('Symbol.asyncDispose');

```

您还需要将编译 `target` 设置为 `es2022` 或以下，配置 `lib` 为 `"esnext"` 或 `"esnext.disposable"`。

```

{
    "compilerOptions": {
        "target": "es2022",
        "lib": ["es2022", "esnext.disposable", "dom"]
    }
}

```

更多详情请参考 [PR](#)。

30.2 Decorator Metadata

TypeScript 5.2 实现了 ECMAScript 即将引入的新功能 [Decorator Metadata](#)。这个功能的关键思想是使装饰器能够轻松地在它们所使用或嵌套的任何类上创建和使用元数据。

在任意的装饰器函数上，现在可以访问上下文对象的 `metadata` 属性。`metadata` 属性是一个普通的对象。由于 JavaScript 允许我们对其任意添加属性，它可以被用作可由每个装饰器更新的字典。或者，由于每个 `metadata` 对象对于每个被装饰的部分来讲是等同的，它可以被用作 `Map` 的键。当类的装饰器运行时，这个对象可以通过 `Symbol.metadata` 访问。

```

interface Context {
    name: string;
    metadata: Record;
}

```

```

function setMetadata(_target: any, context: Context) {
    context.metadata[context.name] = true;
}

```

```

class SomeClass {
    @setMetadata
    foo = 123;

    @setMetadata
    accessor bar = 'hello!';

    @setMetadata
    baz() {}
}

```

```

const ourMetadata = SomeClass[Symbol.metadata];

```

```

console.log(JSON.stringify(ourMetadata));
// { "bar": true, "baz": true, "foo": true }

```

它可以被应用在不同的场景中。`Metadata` 信息可以附加在调试、序列化或者依赖注入的场景中。由于每个被装饰的类都会生成 `metadata` 对象，框架可以选择用它们做为 `key` 来访问 `Map` 或 `WeakMap`，或者跟踪它的属性。

例如，我们想通过装饰器来跟踪哪些属性和存取器是可以通过 `Json.stringify` 序列化的：

```

import { serialize, jsonify } from './serializer';

```

```

class Person {
    firstName: string;
    lastName: string;

    @serialize
    age: number;

    @serialize
    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    }

    toJSON() {
        return jsonify(this);
    }
}


```

30.6 将 `symbol` 用于 `WeakMap` 和 `WeakSet` 的键

现在可以将 `symbol` 用于 `WeakMap` 和 `WeakSet` 的键，它也是 ECMAScript 的[新功能](#)。

```
const myWeakMap = new WeakMap();
```

```
const key = Symbol();
const someObject = { /*...*/ };
```

```
// Works! 
myWeakMap.set(key, someObject);
myWeakMap.has(key);
```

[这个更新](#)是由 [Leo Elmecker-Plakolm](#) 代表 Bloomberg 提供的。我们想向他们表示感谢！

30.7 类型导入路径里使用 `TypeScript` 实现文件扩展名

`TypeScript` 支持在类型导入路径里使用声明文件扩展名和实现文件扩展名，不论是否启用了 `allowImportingTsExtensions`。也意味着你现在可以编写 `import type` 语句并使用 `.ts`、`.mts`、`.cts` 以及 `.tsx` 文件扩展。

```
import type { JustAType } from "./justTypes.ts";
```

```
export function f(param: JustAType) {
  // ...
}
```

这也意味着，`import()` 类型（用在 `TypeScript` 和 `JavaScript` 的 `JSDoc` 中）也可以使用这些扩展名。

```
/**
 * @param {import("./justTypes.ts").JustAType} param
 */
export function f(param) {
  // ...
}
```

更多详情请查看 [PR](#)。

30.8 对象成员的逗号补全

在给对象添加新属性时很容易忘记添加逗号。在之前，如果你忘了写逗号并且请求自动补全，`TypeScript` 会给出差的不相关的补全结果。`TypeScript 5.2` 现在在您缺少逗号时会优雅地提供对象成员的自动补全。但为了避免语法错误的出现，它还会自动插入缺失的逗号。

更多详情请查看 [PR](#)。

30.9 内联变量重构

`TypeScript 5.2` 现在具有一种重构方法，可以将变量的内容内联到所有使用位置。使用“内联变量”重构将消除变量并将所有变量的使用替换为其初始化值。请注意，这可能会导致初始化程序的副作用在不同的时间运行，并且运行的次数与变量的使用次数相同。

更多详情请查看 [PR](#)。

30.10 可点击的内嵌参数提示

内嵌提示可以让我们一目了然地获取信息，即使它在我们的代码中不存在——比如参数名称、推断类型等等。在 `TypeScript 5.2` 中，我们开始使得与内嵌提示进行交互成为可能。例如，在 `Visual Studio Code Insiders` 中，您现在可以点击内联提示以跳转到参数的定义处。

使用 `symbol` 意味着该数据可以被他人访问。另一选择是使用 `WeakMap` 并用该 `metadata` 对象做为键。这样可以保持数据的私密性，并且在这种情况下使用更少的类型断言，但其他方面类似。

```
const serializables = new WeakMap();
```

```
type Context =
  | ClassAccessorDecoratorContext
  | ClassGetterDecoratorContext
  | ClassFieldDecoratorContext;
```

```
export function serialize(_target: any, context: Context): void {
  if (context.static || context.private) {
    throw new Error('Can only serialize public instance members.');
```

```
  }
  if (typeof context.name !== 'string') {
    throw new Error('Can only serialize string properties.');
```

```
  }

  let propNames = serializables.get(context.metadata);
  if (propNames === undefined) {
    serializables.set(context.metadata, (propNames = []));
  }
  propNames.push(context.name);

  export function jsonify(instance: object): string {
    const metadata = instance.constructor[Symbol.metadata];
    const propNames = metadata && serializables.get(metadata);
    if (!propNames) {
      throw new Error('No members marked with @serialize.');
```

```
    }

    const pairStrings = propNames.map(key => {
      const strKey = JSON.stringify(key);
      const strValue = JSON.stringify((instance as any)[key]);
      return `${strKey}: ${strValue}`;
    });

    return `{ ${pairStrings.join(', ')} }`;
```

```
  }
  注意，这里的实现没有考虑子类 and 继承。留给读者作为练习。
  由于该功能比较新，大多数运行时都没实现它。如果想要使用，则需要使用 Symbol.metadata 的 polyfill。例如像下面这样就可以适用大部分场景：
  Symbol.metadata ??= Symbol('Symbol.metadata');
  你还需要将编译 target 设为 es2022 或以下，配置 lib 为 "esnext" 或者 "esnext.decorators"。
```

```
{
  "compilerOptions": {
    "target": "es2022",
    "lib": ["es2022", "esnext.decorators", "dom"]
```



```
}
}
}
感谢 Oleksandr Tarasiuk 的贡献。
```

30.3 命名的和匿名的元组元素

元组类型已经支持了为每个元素定义可选的标签和命名。

```
type Pair = [first: T, second: T];
```

这些标签不改变功能 - 它们只是用于增强可读性和工具支持。

然而, TypeScript 之前有个限制是不允许混用有标签和无标签的元素。 换句话说, 要么所有元素都没有标签, 要么所有元素都有标签。

```
// ☒ fine - no labels
type Pair1 = [T, T];

// ☒ fine - all fully labeled
type Pair2 = [first: T, second: T];

// ☒ previously an error
type Pair3 = [first: T, T];
// ~~~~~
// Tuple members must all have names
// or all not have names.
如果是剩余元素就比较烦人了, 我们必须添加标签 rest 或者 tail.
// ☒ previously an error
type TwoOrMore_A = [first: T, second: T, ...T[]];
// ~~~~~
// Tuple members must all have names
// or all not have names.
```

```
// ☒
type TwoOrMore_B = [first: T, second: T, rest: ...T[]];
这也意味着这个限制必须在类型系统内部进行强制执行, 这意味着 TypeScript 将失去标签。
type HasLabels = [a: string, b: string];
type HasNoLabels = [number, number];
type Merged = [...HasNoLabels, ...HasLabels];
// ^ [number, number, string, string]
// 'a' and 'b' were lost in 'Merged'
在 TypeScript 5.2 中, 对元组标签的全有或全无限制已经被取消。 而且现在可以在展开的元组中保留标签。
```

感谢 [Josh Goldberg](#) 和 [Mateusz Burzyński](#) 的贡献。

30.4 更容易地使用联合数组上的方法

在之前版本的 TypeScript 中, 在联合数组上调用方法可能很痛苦。

```
declare let array: string[] | number[];
```

```
array.filter(x => !!x);
// ~~~~~ error!
// This expression is not callable.
// Each member of the union type '...' has signatures,
// but none of those signatures are compatible
```

```
// with each other.
此例中, TypeScript 会检查是否每个版本的 filter 都与 string[] 和 number[] 兼容。 在没有一个连贯的策略的情况下, TypeScript 会束手无策地说: “我无法使其工作”。
在 TypeScript 5.2 里, 在放弃之前, 联合数组会被特殊对待。 使用每个元素类型构造一个新数组, 然后在其上调用方法。
对于上例来说, string[] | number[] 被转换为 (string | number)[] (或者是 Array<string | number>), 然后在该类型上调用 filter。 有一个注意事项, filter 会产生 Array<string | number> 而不是 string[] | number[]; 但对于新生成的值, 出现“出错”的风险较小。
这意味着在以前不能使用的情况下, 许多方法如 filter、find、some、every 和 reduce 都可以在数组的联合类型上调用。
更多详情请参考 PR。
```

30.5 拷贝的数组方法

TypeScript 5.2 支持了 ECMAScript 提案 [Change Array by Copy](#)。JavaScript 中的数组有很多有用的方法如 sort(), splice(), 以及 reverse(), 这些方法在数组中原地修改元素。 通常, 我们想创建一个新数组, 还想影响原来的数组。 为达到此目的, 你可以使用 slice() 或者展开数组 (例如 [...myArray]) 获取一份拷贝, 然后再执行操作。 例如, 你可以用 myArray.slice().reverse() 来获取反转的数组的拷贝。 还有一个典型的例子 - 创建一份拷贝, 但是修改其中的一个元素。 有许多方法可以实现这一点, 但最明显的方法要么是由多个语句组成的...

```
const copy = myArray.slice();
copy[someIndex] = updatedValue;
doSomething(copy);
要么意图不明显...
doSomething(
  myArray.map((value, index) => (index === someIndex ? updatedValue : value))
);
```

所有这些对于如此常见的操作来说都很繁琐。 这就是为什么 JavaScript 现在有了 4 个新的方法, 执行相同的操作, 但不影响原始数据: toSorted、toSpliced、toReversed 和 with。 前三个方法执行与它们的变异版本相同的操作, 但返回一个新的数组。 with 也返回一个新的数组, 但其中一个元素被更新 (如上所述)。

修改	拷贝
myArray.reverse()	myArray.toReversed()
myArray.sort((a, b) => ...)	myArray.toSorted((a, b) => ...)
myArray.splice(start, deleteCount, ...items)	myArray.toSpliced(start, deleteCount, ...items)
myArray[index] = updatedValue	myArray.with(index, updatedValue)

请注意, 复制方法始终创建一个新的数组, 而修改操作则不一致。 这些方法不仅存在于普通数组上 - 它们还存在于 TypedArray 上, 例如 Int32Array, Uint8Array, 等。

感谢 [Carter Snook](#) 的 [PR](#)。


```
ts_upgrade_from_2.2_to_5.3
}
```

```
interface B {
  b: string;
}

type MyType = A | B;

function isA(x: MyType): x is A {
  return 'a' in x;
}

function someFn(x: MyType) {
  if (isA(x) === true) {
    console.log(x.a); // works!
  }
}
```

感谢 Mateusz Burzyński 的 [PR](#)。

31.6 利用 `Symbol.hasInstance` 来细化 `instanceof`

JavaScript 的一个稍微晦涩的特性是可以覆盖 `instanceof` 运算符的行为。为此，`instanceof` 运算符右侧的值需要具有一个名为 `Symbol.hasInstance` 的特定方法。

```
class Weirdo {
  static [Symbol.hasInstance](testedValue) {
    // wait, what?
    return testedValue === undefined;
  }
}

// false
console.log(new Thing() instanceof Weirdo);

// true
console.log(undefined instanceof Weirdo);
```

为了更好地支持 `instanceof` 的行为，TypeScript 现在会检查是否存在 `[Symbol.hasInstance]` 方法且被定义为类型判定函数。如果有的话，`instanceof` 运算符左侧的值会按照类型判定进行细化。

```
interface PointLike {
  x: number;
  y: number;
}

class Point implements PointLike {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}
```

更多详情请查看 [PR](#)。

30.11 优化进行中的类型兼容性检查

由于 TypeScript 采用的是结构化的类型系统，通常需要比较类型成员；然而，递归类型会造成一些问题。例如：

```
interface A {
  value: A;
  other: string;
}
```

```
interface B {
  value: B;
  other: number;
}
```

在检查 A 是否与 B 类型兼容时，TypeScript 会检查 A 和 B 中 `value` 的类型是否兼容。此时，类型系统需要停止进一步检查并继续检查其他成员。为此，类型系统必须跟踪两个类型是否已经相关联。

此前，TypeScript 已经保存了配对类型的栈，并迭代检查类型是否已经关联。当这个堆栈很浅时，这不是一个问题；但当堆栈不是很浅时，那就是个[问题](#)了。

在 TypeScript 5.2 中，一个简单的 Set 就能跟踪这些信息。在使用了 `drizzle` 库的测试报告中，这项改动减少了超过 33% 的时间花费！

```
Benchmark 1: old
  Time (mean ± σ):      3.115 s ± 0.067 s    [User: 4.403 s, System: 0.124 s]
  Range (min ... max):  3.018 s ... 3.196 s    10 runs
```

```
Benchmark 2: new
  Time (mean ± σ):      2.072 s ± 0.050 s    [User: 3.355 s, System: 0.135 s]
  Range (min ... max):  1.985 s ... 2.150 s    10 runs
```

```
Summary
  'new' ran
  1.50 ± 0.05 times faster than 'old'
```

更多详情请查看 [PR](#)。

31 v5.3

<http://www.patrickzhong.com/TypeScript/zh/release-notes/typescript-5.3.html>

31.1 导入属性 (Import Attributes)

TypeScript 5.3 支持了最新的 [import attributes](#) 提案。

该特性的一个用例是为运行时提供期望的模块格式信息。

```
// We only want this to be interpreted as JSON,
// not a runnable/malicious JavaScript file with a `.json` extension.
import obj from "./something.json" with { type: "json" };
TypeScript 不会检查属性内容，因为它们是宿主环境相关的。TypeScript 会原样保留它们，浏览器和运行时都会处理它们。
```

```
// TypeScript is fine with this.
// But your browser? Probably not.
import * as foo from "./foo.js" with { type: "fluffy bunny" };
动态的 import() 调用也可以在第二个参数里使用该特性。
const obj = await import('./something.json', {
  with: { type: 'json' },
});
```

第二个参数的期望类型为 ImportCallOptions，默认只支持一个名为 with 的属性。请注意，导入属性是之前提案“[导入断言](#)”的演进，该提案已在 TypeScript 4.5 中实现。最明显的区别是使用 with 关键字而不是 assert 关键字。但不太明显的区别是，现在运行时可以自由地使用属性来指导导入路径的解析和解释，而导入断言只能在加载模块后断言某些特性。随着时间的推移，TypeScript 将逐渐弃用旧的导入断言语法，转而采用导入属性的提议语法。现有的使用 assert 的代码应该迁移到 with 关键字。而需要导入属性的新代码应该完全使用 with 关键字。

感谢 Oleksandr Tarasiuk 实现了这个功能！也感谢 Wenlu Wang 实现了 import assertions！

31.2 稳定支持 import type 上的 resolution-mode

TypeScript 4.7 在 `/// <reference types="..." />` 里支持了 resolution-mode 属性，它用来控制一个描述符是使用 import 还是 require 语义来解析。

```
/// <reference types="pkg" resolution-mode="require" />
```

```
// or

/// <reference types="pkg" resolution-mode="import" />
在 type-only 导入上，导入断言也引入了相应的字段：然而，它仅在 TypeScript 的夜间版本中得到支持 其原因是在精神上，导入断言并不打算指导模块解析。因此，这个特性以实验性的方式仅在夜间版本中发布，以获得更多的反馈。
```

但是，导入属性 (Import Attributes) 可以指导解析，并且我们也已经看到了有意义的用例，TypeScript 5.3 在 import type 上支持了 resolution-mode。

```
// Resolve `pkg` as if we were importing with a `require()`
import type { TypeFromRequire } from "pkg" with {
  "resolution-mode": "require"
};
```

```
// Resolve `pkg` as if we were importing with an `import`
import type { TypeFromImport } from "pkg" with {
```

```
  "resolution-mode": "import"
};
```

```
export interface MergedType extends TypeFromRequire, TypeFromImport {}
这些导入属性也可以用在 import() 类型上。
export type TypeFromRequire =
  import("pkg", { with: { "resolution-mode":
"require" } }).TypeFromRequire;
```

```
export type TypeFromImport =
  import("pkg", { with: { "resolution-mode":
"import" } }).TypeFromImport;
```

```
export interface MergedType extends TypeFromRequire, TypeFromImport {}
更多详情，请参考 PR。
```

31.3 在所有模块模式的支持下支持 resolution-mode

此前，仅在 moduleResolution 为 node16 和 nodenext 时支持 resolution-mode。为了使查找模块更容易，尤其针对类型，resolution-mode 现在可以在所有其它的 moduleResolution 选项下工作，例如 bundler、node10，甚至在 classic 下也不报错。更多详情，请参考 [PR](#)。

31.4 switch (true) 类型细化

TypeScript 5.3 会针对 switch (true) 里的每一个 case 条件进行类型细化。

```
function f(x: unknown) {
  switch (true) {
    case typeof x === 'string':
      // 'x' is a 'string' here
      console.log(x.toUpperCase());
      // falls through...

    case Array.isArray(x):
      // 'x' is a 'string | any[]' here.
      console.log(x.length);
      // falls through...

    default:
      // 'x' is 'unknown' here.
      // ...
  }
}
```

感谢 Mateusz Burzyński 的[贡献](#)。

31.5 类型细化与布尔值的比较

有时，您可能会发现自己在条件语句中直接与 true 或 false 进行比较。通常情况下，这些比较是不必要的，但您可能出于风格上的考虑或为了避免 JavaScript 中真值相关的某些问题而偏好这样做。不过，之前 TypeScript 在进行类型细化时并不识别这样的形式。

TypeScript 5.3 在类型细化时可以理解这类表达式。

```
interface A {
  a: string;
```

能是具有挑战性的。 意外加载两个模块太容易了，而且代码可能在 API 的不同实例上无法正常工作。 即使它可以工作，加载第二个捆绑包会增加资源使用量。

基于此，我们决定合并这两个文件。 typescript.js 现在包含了以前在 tsserverlibrary.js 中的内容，而 tsserverlibrary.js 现在只是重新导出 typescript.js。 在这个合并前后，我们看到了以下包大小的减小：

	Before	After	Diff	Diff (percent)
Packed	6.90 MiB	5.48 MiB	-1.42 MiB	-20.61%
Unpacked	38.74 MiB	30.41 MiB	-8.33 MiB	-21.50%

	Before	After	Diff	Diff (percent)
lib/tsserverlibrary.d.ts	570.95 KiB	865.00 B	-570.10 KiB	-99.85%
lib/tsserverlibrary.js	8.57 MiB	1012.00 B	-8.57 MiB	-99.99%
lib/typescript.d.ts	396.27 KiB	570.95 KiB	+174.68 KiB	+44.08%
lib/typescript.js	7.95 MiB	8.57 MiB	+637.53 KiB	+7.84%

换句话说，这意味着包大小减小了超过 20.5%。
更多详情请参考 [PR](#)。

```
}

distanceFromOrigin() {
  return Math.sqrt(this.x ** 2 + this.y ** 2);
}

static [Symbol.hasInstance](val: unknown): val is PointLike {
  return (
    !!val &&
    typeof val === 'object' &&
    'x' in val &&
    'y' in val &&
    typeof val.x === 'number' &&
    typeof val.y === 'number'
  );
}
}

function f(value: unknown) {
  if (value instanceof Point) {
    // Can access both of these - correct!
    value.x;
    value.y;

    // Can't access this - we have a 'PointLike',
    // but we don't *actually* have a 'Point'.
    value.distanceFromOrigin();
  }
}
```

能够看到例子中，Point 定义了自己的 [Symbol.hasInstance] 方法。 它实际上充当了对称为 PointLike 的单独类型的自定义类型保护。 在函数 f 中，我们能够使用 instanceof 将 value 细化为 PointLike，但不能细化到 Point。 这意味着我们可以访问属性 x 和 y，但无法访问 distanceFromOrigin 方法。
更多详情请参考 [PR](#)。

31.7 [在实例字段上检查 super 属性访问](#)

在 JavaScript 中，能够使用 super 关键字来访问基类中的声明。

```
class Base {
  someMethod() {
    console.log('Base method called!');
  }
}

class Derived extends Base {
  someMethod() {
    console.log('Derived method called!');
    super.someMethod();
  }
}
```

```
new Derived().someMethod();
// Prints:
//   Derived method called!
//   Base method called!
```

这与 `this.someMethod()` 是不同的，因为它可能调用的是重写的方法。 这是一个微妙的区别，而且通常情况下，如果一个声明从未被覆盖，这两者可以互换，使得区别更加微妙。

```
class Base {
  someMethod() {
    console.log('someMethod called!');
  }
}
```

```
class Derived extends Base {
  someOtherMethod() {
    // These act identically.
    this.someMethod();
    super.someMethod();
  }
}
```

```
new Derived().someOtherMethod();
// Prints:
//   someMethod called!
//   someMethod called!
```

将它们互换使用的问题在于，`super` 关键字仅适用于在原型上声明的成员，而不适用于实例属性。 这意味着，如果您编写了 `super.someMethod()`，但 `someMethod` 被定义为一个字段，那么您将会得到一个运行时错误！

```
class Base {
  someMethod = () => {
    console.log('someMethod called!');
  };
}
```

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

```
new Derived().someOtherMethod();
//
// Doesn't work because 'super.someMethod' is 'undefined'.
```

TypeScript 5.3 现在更仔细地检查 `super` 属性访问/方法调用，以确定它们是否对应于类字段。 如果是这样，我们现在将会得到一个类型检查错误。

[这个检查](#)是由 Jack Works 开发！

31.8 [可以交互的类型内嵌提示](#)

TypeScript 的内嵌提示支持跳转到类型定义！ 这便利在代码间跳转变得简单。

更多详情请参考 [PR](#)。

31.9 [设置偏好 type 自动导入](#)

之前，当 TypeScript 为类型自动生成导入语句时，它会根据配置添加 `type` 修饰符。 例如，当为 `Person` 生成自动导入语句时：

```
export let p: Person;
TypeScript 通常会这样生成 Person 导入：
import { Person } from './types';
```

```
export let p: Person;
如果设置了 verbatimModuleSyntax，它会添加 type 修饰符：
import { type Person } from './types';
```

```
export let p: Person;
```

然而，也许你的编辑器不支持这些选项；或者你偏好显式地使用 `type` 导入。[最近的一项改动](#)，TypeScript 把它变成了针对编辑器的配置项。 在 Visual Studio Code 中，你可以在 "TypeScript > Preferences: Prefer Type Only Auto Imports" 启用该功能，或者在 JSON 配置文件中

的 `typescript.preferences.preferTypeOnlyAutoImports` 设置。

31.10 [优化：略过 JSDoc 解析](#)

当通过 `tsc` 运行 TypeScript 时，编译器现在将避免解析 JSDoc。 这不仅减少了解析时间，还减少了存储注释以及垃圾回收所花费的内存使用量。 总体而言，您应该会看到编译速度稍微更快，并在 `--watch` 模式下获得更快的反馈。

[具体改动在这](#)。

由于并非每个使用 TypeScript 的工具都需要存储 JSDoc（例如 `typescript-eslint` 和 `Prettier`），因此这种解析策略已作为 API 的一部分公开。 这使得这些工具能够获得与 TypeScript 编译器相同的内存和速度改进。 注释解析策略的新选项在 `JSDocParsingMode` 中进行了描述。 关于此拉取请求的更多信息，请参阅 [PR](#)。

31.11 [通过比较非规范化的交叉类型进行优化](#)

在 TypeScript 中，联合类型和交叉类型始终遵循特定的形式，其中交叉类型不能包含联合类型。 这意味着当我们在一个联合类型上创建一个交叉类型，例如 `A & (B | C)`，该交叉类型将被规范化为 `(A & B) | (A & C)`。 然而，在某些情况下，类型系统会保留原始形式以供显示目的使用。

事实证明，原始形式可以用于一些巧妙的快速路径类型比较。

例如，假设我们有 `SomeType & (Type1 | Type2 | ... | Type99999NINE)`，我们想要确定它是否可以赋值给 `SomeType`。 回想一下，我们实际上没有一个交叉类型作为源类型，而是一个联合类型，看起来像是 `(SomeType & Type1) | (SomeType & Type2) | ... | (SomeType & Type99999NINE)`。 当检查一个联合类型是否可以赋值给目标类型时，我们必须检查联合类型的每个成员是否可以赋值给目标类型，这可能非常慢。

在 TypeScript 5.3 中，我们查看了我们能够隐藏的原始交叉类型形式。 当我们比较这些类型时，我们会快速检查目标类型是否存在于源交叉类型的任何组成部分中。

更多详情请参考 [PR](#)。

31.12 [合并 tsserverlibrary.js 和 typescript.js](#)

TypeScript 本身包含两个库文件：`tsserverlibrary.js` 和 `typescript.js`。 在 `tsserverlibrary.js` 中有一些仅在其中可用的 API（如 `ProjectService` API），对某些导入者可能很有用。 尽管如此，这两个是不同的捆绑包，有很多重叠的部分，在包中重复了一些代码。 更重要的是，由于自动导入或肌肉记忆的原因，要始终一致地使用其中一个可