《TypeScript编程》

编程 TypeScript

作者 [Boris] [Cherny]

在美国印刷。

由 [O’Reilly Media, Inc.] 出版，地址：1005 Gravenstein Highway North, Sebastopol, CA 95472。

O’Reilly 书籍可用于教育、商业或销售推广用途。大多数书名也有在线版本 (http://oreilly.com)。如需更多信息，请联系我们的企业/机构销售部门：800-998-9938 或 [*corporate@oreilly.com*]。

[开发编辑:] Angela Rufino [索引编制:] Margaret	Troutman
[采编编辑:] Jennifer Pollock [内部设计:] David Fut	ato
[制作编辑:] Katherine Tozer [封面设计:] Karen M	ontgomery
[文字编辑:] Rachel Head [插图:] Rebe	cca Demarest
[校对:] Charles Roumeliotis	

- [2019年5月:] 第一版

第一版修订历史

- [2019-04-18:] 首次发布

发布详情请见 http://oreilly.com/catalog/errata.csp?isbn=9781492037651。

978-1-492-03765-1

[LSI]

献词

献给 Sasha 和 Michael，他们也许有一天会爱上类型。

前言

这是一本面向各行各业程序员的书：专业 JavaScript 工程师、C# 程序员、Java 支持者、Python 爱好者、Ruby 爱好者、Haskell 极客。无论你用什么语言编程，只要你有一些编程经验并了解函数、变量、类和错误的基础知识，这本书就适合你。一些 JavaScript 经验，包括对文档对象模型(DOM)和网络的基本了解，将对你有所帮助——虽然我们不会深入探讨这些概念，但它们是优秀示例的源泉，如果你不熟悉它们，示例可能不会那么有意义。

无论你过去使用过什么编程语言，我们所有人共同的经验是追踪异常，逐行跟踪代码以找出错误所在以及如何修复它。这正是 TypeScript 通过自动检查您的代码并指出您可能遗漏的错误来帮助预防的体验。

如果您以前没有使用过静态类型语言也没关系。我将教您有关类型以及如何有效使用它们来减少程序崩溃、更好地记录代码，并在更多用户、工程师和服务器之间扩展应用程序。我会尽量避免使用复杂的词汇，并以直观、易记和实用的方式解释思想，沿途使用大量示例来保持具体性。

这就是 TypeScript 的特点：与许多其他类型语言不同，TypeScript 非常实用。它发明了全新的概念，使您能够更简洁、更精确地表达，让您以有趣、现代和安全的方式编写应用程序。

本书的组织结构

本书有两个目标：让您深入理解 TypeScript 语言的工作原理（理论）并提供大量关于如何编写生产 TypeScript 代码的实用建议（实践）。

由于 TypeScript 是一种如此实用的语言，理论很快就会转向实践，本书的大部分内容最终都是两者的结合，前几章几乎完全是理论，最后几章几乎完全是实践。

我将从编译器、类型检查器和类型的基础知识开始。然后我将广泛概述 TypeScript 中不同的类型和类型运算符，它们的用途以及如何使用它们。利用我们学到的知识，我将涵盖一些高级主题，如 TypeScript 最复杂的类型系统特性、错误处理和异步编程。最后，我将总结如何将 TypeScript 与您最喜欢的框架（前端和后端）一起使用，将现有的 JavaScript 项目迁移到 TypeScript，以及在生产环境中运行 TypeScript 应用程序。

大多数章节末尾都有一套练习题。尝试自己完成这些练习——它们会给你比仅仅阅读更深入的直觉来理解我们所涵盖的内容。章节练习的答案可在线获取，网址为 *https://github.com/bcherny/programming-typescript-answers*。

风格

代码风格和约定

在本书中，我尽量坚持使用单一的代码风格。这种风格的某些方面是非常个人化的——例如：

- 我只在必要时使用分号。
- 我使用两个空格进行缩进。
- 当程序是一个快速片段，或者程序的结构比细节更重要时，我使用像 `a`、`f` 或 `_` 这样的短变量名。

然而，代码风格的某些方面是我认为你也应该遵循的。其中一些是：

- 你应该使用最新的 JavaScript 语法和特性（最新的 JavaScript 版本通常就叫做”esnext”）。这将使你的代码与最新标准保持一致，提高互操作性和可搜索性，并且可以帮助减少新员工的学习时间。它还让你能够利用强大的现代 JavaScript 特性，如箭头函数、promises 和生成器。
- 你应该在大部分时候使用展开运算符（`...`）保持数据结构不可变。
- 你应该确保一切都有类型，在可能的情况下进行推断。小心不要滥用显式类型；这将有助于保持代码清晰简洁，并通过暴露不正确的类型而不是掩盖它们来提高安全性。
- 你应该保持代码的可重用性和通用性。多态性（参见”多态性”）是你最好的朋友。

当然，这些想法并不新鲜。但是当你坚持这些原则时，TypeScript 工作得特别好。TypeScript 的内置降级编译器、对只读类型的支持、强大的类型推断、对多态性的深度支持，以及完全结构化的类型系统都鼓励良好的编码风格，同时语言保持了令人难以置信的表达力，并且忠于底层的 JavaScript。

在我们开始之前，还有几个注意事项。

JavaScript 不暴露指针和引用；相反，它有值类型和引用类型。值是不可变的，包括字符串、数字和布尔值等，而引用指向通常可变的数据结构，如数组、对象和函数。当我在本书中使用”值”这个词时，我通常是宽泛地指代 JavaScript 值或引用。

最后，当与 JavaScript、错误类型的第三方库、遗留代码互操作时，或者如果你赶时间，你可能会发现自己在实际项目中编写不太理想的 TypeScript 代码。本书主要展示你*应该*如何编写 TypeScript，并论证为什么你应该努力不妥协。但在实践中，你的代码有多正确取决于你和你的团队。

本书使用的约定

本书使用以下排版约定：

斜体
表示新术语、URL、电子邮件地址、文件名和文件扩展名。

`等宽字体`
用于程序清单，以及在段落中引用程序元素，如变量或函数名、数据类型、环境变量、语句和关键字。

`等宽斜体`
显示应该被用户提供的值或由上下文确定的值替换的文本。

提示

此元素表示提示或建议。

注意

此元素表示一般说明。

警告

此元素表示警告或注意事项。

使用代码示例

补充材料（代码示例、练习等）可在 *https://github.com/bcherny/programming-typescript-answers* 下载。

本书旨在帮助你完成工作。一般来说，如果本书提供了示例代码，你可以在你的程序和文档中使用它。除非你要复制代码的重要部分，否则你不需要联系我们获得许可。例如，编写一个使用本书中几个代码块的程序不需要许可。销售或分发包含 O’Reilly 图书示例的 CD-ROM 需要许可。通过引用本书并引用示例代码来回答问题不需要许可。将本书中大量示例代码合并到你的产品文档中需要许可。

我们感谢但不要求署名。署名通常包括标题、作者、出版商和 ISBN。例如：“*Programming TypeScript* by Boris Cherny (O’Reilly). Copyright 2019 Boris Cherny, 978-1-492-03765-1.”

如果你觉得你对代码示例的使用超出了合理使用或上述许可，请随时通过 *permissions@oreilly.com* 联系我们。

O’Reilly Online Learning

注意

近 40 年来，*O’Reilly Media* 一直为公司提供技术和商业培训、知识和见解，帮助公司取得成功。

我们独特的专家和创新者网络通过图书、文章、会议和我们的在线学习平台分享他们的知识和专业技能。O’Reilly 的在线学习平台让你按需访问现场培训课程、深度学习路径、交互式编码环境，以及来自 O’Reilly 和其他 200 多个出版商的大量文本和视频资源。更多信息，请访问 *http://oreilly.com*。

如何联系我们

请就本书的评论和问题联系出版社：

- O’Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (美国或加拿大)
- 707-829-0515 (国际或本地)
- 707-829-0104 (传真)

我们为本书建立了一个网页，其中列出了勘误表、示例以及任何其他信息。您可以通过 *https://oreil.ly/programming-typescript* 访问此页面。

如需对本书发表评论或询问技术问题，请发送电子邮件至 *bookquestions@oreilly.com*。

有关我们的书籍、课程、会议和新闻的更多信息，请访问我们的网站 *http://www.oreilly.com*。

在Facebook上关注我们：*http://facebook.com/oreilly*

在Twitter上关注我们：*http://twitter.com/oreillymedia*

在YouTube上观看我们：*http://www.youtube.com/oreillymedia*

致谢

本书是多年来片段和涂鸦的产物，随后是一整年早晨、夜晚、周末和节假日的写作时光。

感谢O’Rielly给我写这本书的机会，感谢我的编辑Angela Rufino在整个过程中的支持。感谢Nick Nance在[“类型安全API”]中的贡献，感谢Shyam Seshadri在[“Angular 6/7”]中的贡献。感谢我的技术编辑：TypeScript团队的Daniel Rosenwasser，他花费了大量时间阅读这份手稿并指导我了解TypeScript类型系统的细节，以及Jonathan Creamer、Yakov Fain、Paul Buying和Rachel Head的技术编辑和反馈。感谢我的家人——Liza和Ilya、Vadim、Roza和Alik、Faina和Yosif——鼓励我进行这个项目。

最重要的是，感谢我的伴侣Sara Gilford，她在整个写作过程中支持我，即使这意味着取消周末计划、深夜写作和编程，以及太多关于类型系统细节的不经意谈话。没有你我无法完成这本书，我永远感激你的支持。

[1] 如果你不是从JavaScript转来的，这里有一个例子：如果你有一个对象`o`，想要向其添加一个属性`k`，值为`3`，你可以直接修改`o`——`o.k = 3`——或者你可以对`o`应用更改，创建一个*新*对象作为结果——`let p = {...o, k: 3}`。

第1章 介绍

所以，你决定买一本关于TypeScript的书。为什么？

也许是因为你厌倦了那些奇怪的`cannot read property blah of [undefined]` JavaScript错误。或者你听说TypeScript可以帮助你的代码更好地扩展，想看看这是怎么回事。或者你是C#开发者，一直在考虑尝试JavaScript。或者你是函数式编程开发者，决定是时候将技能提升到下一个水平了。或者你的老板对你的代码导致生产问题感到如此厌烦，以至于他们把这本书作为圣诞礼物送给了你（如果我说中了请告诉我）。

无论你的原因是什么，你听到的都是真的。TypeScript是将为下一代Web应用、移动应用、NodeJS项目和物联网(IoT)设备提供支持的语言。它将通过检查常见错误使你的程序更安全，作为你自己和未来工程师的文档，使重构变得轻松，并且使你一半的单元测试变得不必要（“什么单元测试？”）。TypeScript将使你作为程序员的生产力翻倍，它还会帮你和街对面那个可爱的咖啡师约会。

但在你匆忙跑到街对面之前，让我们稍微详细解释一下，从这个问题开始：当我说”更安全”时到底是什么意思？我谈论的当然是*类型安全*。

类型安全

使用类型来防止程序执行无效操作。[1]

以下是一些无效操作的例子：

- 将数字和列表相乘
- 当函数实际需要对象列表时，用字符串列表调用函数
- 在对象上调用该对象实际不存在的方法
- 导入最近被移动的模块

有些编程语言试图最大化利用这样的错误。它们试图弄清楚当你做了无效操作时你真正的意图，因为嘿，你能做什么就做什么，对吧？以JavaScript为例：

```
3 + []              // 计算结果为字符串"3"

let obj = {}
obj.foo             // 计算结果为undefined

function a(b) {
  return b/2
}
a("z")              // 计算结果为NaN
```

请注意，当你试图做明显无效的事情时，JavaScript不会抛出异常，而是试图尽力而为并尽可能避免异常。JavaScript是在帮助你吗？当然是。这是否让你更容易快速捕获错误？可能不是。

现在想象一下，如果JavaScript抛出更多异常而不是静静地尽力处理我们给它的内容。我们可能会得到这样的反馈：

```
3 + []              // 错误：你真的想要将数字和数组相加吗?

let obj = {}
```

obj.foo // 错误：你忘记在obj上定义属性”foo”。

function a(b) { return b/2 } a(“z”) // 错误：函数”a”期望一个数字，// 但你给了它一个字符串。

不要误解我的意思：试图为我们修复错误是编程语言的一个很好的特性（如果它不仅仅适用于程序就更好了！）。但对于JavaScript来说，这个特性在你在代码中犯错的时间和你*发现*错误的时间之间造成了断层。通常，这意味着你第一次听到你的错误将是来自其他人。

所以这里有一个问题：JavaScript到底什么时候告诉你犯了错误?

对的：当你真正*运行*你的程序时。你的程序可能在你在浏览器中测试时运行，或者当用户访问你的网站时运行，或者当你运行单元测试时运行。如果你很有纪律并编写大量的单元测试和端到端测试，在推送前对代码进行冒烟测试，并在交付给用户之前内部测试一段时间，你有希望在用

户之前发现你的错误。但如果你不这样做呢?

这就是TypeScript的用武之地。比TypeScript给你有用的错误消息更酷的是它*何时*给你这些消息：TypeScript*在你的文本编辑器中，当你输入时*给你错误消息。这意味着你不必依赖单元测试或冒烟测试或同事来捕获这些问题：TypeScript会为你捕获它们，并在你编写程序时警告你。让我们看看TypeScript对我们之前的示例说了什么：

3 + [] // 错误 TS2365: 操作符’+‘不能应用于类型’3’ // 和’never[]’。

let obj = {} obj.foo // 错误 TS2339: 类型’{}‘上不存在属性’foo’。

function a(b: number) { return b / 2 } a(“z”) // 错误 TS2345: 类型’“z”’的参数不能赋值给 // 类型’number’的参数。

除了消除整类与类型相关的bug之外，这实际上会改变你编写代码的方式。你会发现自己在类型级别上勾勒出程序，然后在值级别上填充它；^[2]^ 你会在设计程序时思考边缘情况，而不是事后想到；你会设计出更简单、更快、更容易理解和更容易维护的程序。

你准备好开始这段旅程了吗？我们开始吧！

^[1]^ 根据你使用的静态类型语言，"无效"可能意味着很多事情，从运行时会崩溃的程序到不会崩溃但显然无意义的事情。

^[2]^ 如果你不确定这里的"类型级别"是什么意思，不要担心。我们会在后面的章节中深入讨论。

第2章：TypeScript：万米高空视角

在接下来的几章中，我将介绍TypeScript语言，给你概述TypeScript编译器(TSC)如何工作，并带你浏览TypeScript的特性以及你可以用它们开发的模式。我们将从编译器开始。

编译器

根据你过去使用的编程语言（也就是在你决定买这本书并致力于类型安全生活之前），你会对程序如何工作有不同的理解。TypeScript的工作方式与JavaScript或Java等其他主流语言相比是不同寻常的，所以在我们继续之前，我们在同一个页面上是很重要的。

让我们从宽泛开始：程序是包含由你这个程序员编写的一堆文本的文件。该文本由一个称为*编译器(compiler)*的特殊程序解析，它将其转换为*抽象语法树(AST)*，这是一个忽略空白、

注释以及你在制表符与空格辩论中立场等内容的数据结构。然后编译器将该AST转换为称为*字节码(bytecode)*的较低级表示。你可以将该字节码馈送到另一个称为*运行时(runtime)*的程序中来评估它并获得结果。所以当你运行程序时，你真正做的是告诉运行时评估由编译器从你的源代码解析的AST生成的字节码。细节各不相同，但对于大多数语言来说，这是一个准确的高级视图。

再一次，步骤是：

1. 程序被解析为AST。

2. AST被编译为字节码。

3. 字节码由运行时评估。

TypeScript的特殊之处在于，它不是直接编译成字节码，而是编译成...JavaScript代码！然后你像平常一样运行那些JavaScript代码——在浏览器中、使用NodeJS、或者手工用纸笔执行（为了任何在机器崛起后阅读这本书的人）。

此时你可能在想："等等！在上一章中你说TypeScript让我的代码更安全！这是什么时候发生的？"

很好的问题。我其实跳过了一个关键步骤：在TypeScript编译器为你的程序生成AST之后——但在它生成代码之前——它会对你的代码进行*类型检查*。

类型检查器(Typechecker) {#typechecker .calibre29}

一个验证你的代码是类型安全的特殊程序。

这种类型检查就是TypeScript背后的魔法。这就是TypeScript如何确保你的程序按预期工作，没有明显的错误，以及街对面那个可爱的咖啡师真的会在他们说的时候给你回电话。（别担心，他们可能只是很忙。）

所以如果我们包括类型检查和JavaScript生成，编译TypeScript的过程现在大致如图2-1所示：

```
<figure class="calibre33">
<div id="calibre_link-16" class="figure">
<h6 id="figure-2-1.-compiling-and-running-typescript"
class="calibre34"><span class="calibre">图2-1. </span>编译和运行
TypeScript</h6>
</div>
</figure>
```

步骤1-3由TSC完成，步骤4-6由运行在你的浏览器、NodeJS或任何你使用的JavaScript引擎中的JavaScript运行时完成。

注意 {#note-2 .calibre22}

JavaScript编译器和运行时往往被合并成一个叫做*引擎*的单一程序；作为程序员，这是你通常会交互的对象。这就是V8（为NodeJS、Chrome和Opera提供动力的引擎）、SpiderMonkey（Firefox）、JSCore（Safari）和Chakra（Edge）的工作方式，这也是JavaScript看起来像*解释型*语言的原因。

在这个过程中，步骤1-2使用你程序的类型；步骤3不使用。这点值得重申：*当TSC将你的代码从TypeScript编译到JavaScript时，它不会查看你的类型*。这意味着你程序的类型永远不会影响你程序的生成输出，只用于类型检查。这个特性让你可以毫无风险地试验、更新和改进你程序的类型，而不用担心破坏你的应用程序。

类型系统 {#the-type-system .calibre13}

现代语言都有各种不同的*类型系统*。

类型系统(Type system) {#type-system .calibre29}

类型检查器用来为你的程序分配类型的一套规则。

通常有两种类型系统：你必须用显式语法告诉编译器每个东西是什么类型的类型系统，以及自动为你推断事物类型的类型系统。两种方法都有权衡。

TypeScript受到两种类型系统的启发：你可以显式注解你的类型，或者你可以让TypeScript为你推断大部分类型。

要显式地向TypeScript表明你的类型是什么，使用注解。注解采用*值：类型*的形式，告诉类型检查器："嘿！你看到这里的这个*值*了吗？它的类型是*类型*。"让我们看几个例子（每行后面的注释是TypeScript推断的实际类型）：

let a: number = 1 // a是number类型 let b: string = ‘hello’ // b是string类型 let c: boolean[] = [true, false] // c是boolean数组类型

如果你希望TypeScript为你推断类型，只需省略它们，让TypeScript开始工作：

let a = 1 // a是number类型 let b = ‘hello’ // b是string类型 let c = [true, false] // c是boolean数组类型

你会立即注意到TypeScript在为你推断类型方面有多出色。如果你省略注解，类型是相同的！在本书中，我们只在必要时使用注解，并尽可能让TypeScript为我们施展推断魔法。

注意 {#note-3 .calibre22}

一般来说，让TypeScript为你推断尽可能多的类型是好的风格，将显式类型的代码保持在最少。

TypeScript与JavaScript对比 {#typescript-versus-javascript .calibre17}

让我们更深入地了解TypeScript的类型系统，以及它与JavaScript类型系统的比较。[表2-1]提供了一个概览。良好地理解两者之间的差异是构建TypeScript工作原理心理模型的关键。

类型系统特性	JavaScript	TypeScript
类型是如何绑定的？	动态	静态
类型是否自动转换？	是	否（大多数情况下）
何时检查类型？	运行时	编译时
何时暴露错误？	运行时（大多数情况下）	编译时（大多数情况下）

: [表2-1.]JavaScript和TypeScript类型系统比较

类型是如何绑定的？

动态类型绑定意味着JavaScript需要实际运行你的程序才能知道其中事物的类型。JavaScript在运行程序之前不知道你的类型。

TypeScript是一种*渐进式类型*语言。这意味着TypeScript在编译时知道程序中所有事物的类型时工作得最好，但它不必知道每个类型就能编译你的程序。即使在无类型的程序中，TypeScript也能为你推断一些类型并捕获一些错误，但如果不知道所有事物的类型，它会让很多错误漏到你的用户那里。

这种渐进式类型对于将传统代码库从无类型JavaScript迁移到有类型TypeScript非常有用

（更多内容见["从JavaScript逐步迁移到TypeScript"]），但除非你正在迁移代码库，否则你应该追求100%的类型覆盖。除非明确说明，否则本书采用这种方法。

类型是否自动转换?

JavaScript是弱类型的，意味着如果你做了无效的操作，比如将数字和数组相加（就像我们在[第1章]中做的），它会应用一系列规则来弄清楚你真正的意思，以便用你给它的东西尽力完成任务。让我们详细了解JavaScript如何计算`3 + [1]`的具体示例：

1. JavaScript注意到`3`是一个数字，`[1]`是一个数组。

2. 因为我们使用`+`，它假设我们想要连接两者。

3. 它隐式地将`3`转换为字符串，产生`"3"`。

4. 它隐式地将`[1]`转换为字符串，产生`"1"`。

5. 它连接结果，产生`"31"`。

我们也可以更明确地做这件事（这样JavaScript就避免了步骤1、3和4）：

```
3 + [1]; // 求值为 “31”

(3).toString() + [1].toString() // 求值为 “31”
```

虽然JavaScript试图通过为你做巧妙的类型转换来提供帮助，但TypeScript一旦你做了无效的事情就会抱怨。当你通过TSC运行相同的JavaScript代码时，你会得到一个错误：

```
3 + [1]; // Error TS2365: 操作符 ‘+’ 不能应用于 // 类型 ‘3’ 和 ‘number[]’。

(3).toString() + [1].toString() // 求值为 “31”
```

如果你做了看起来不对的事情，TypeScript会抱怨，如果你明确表达你的意图，TypeScript就不会妨碍你。这种行为是有意义的：有哪个头脑正常的人会试图将数字和数组相加，期望结果是字符串（当然，除了在你的初创公司地下室里在烛光下编程的JavaScript女巫Bavmorda）？

JavaScript进行的这种隐式转换可能是非常难以追踪的错误源，是许多JavaScript程序员的噩梦。它使个别工程师难以完成工作，并且使跨大型团队扩展代码变得更加困难，因为每个工程师都需要理解你的代码所做的隐式假设。

简而言之，如果你必须转换类型，请明确地进行。

何时检查类型?

在大多数地方，JavaScript不关心你给它什么类型，而是试图尽力将你给它的东西转换为它期望的东西。

另一方面，TypeScript在编译时对你的代码进行类型检查（记住本章开头列表中的步骤2？），所以你不需要实际运行代码就能看到前面示例中的`Error`。TypeScript*静态分析*你的代码寻找这样的错误，并在你运行代码之前向你显示它们。如果你的代码不能编译，这是一个非常好的信号，表明你犯了错误，应该在尝试运行代码之前修复它。

[图2-2]显示了当我在VSCode（我选择的代码编辑器）中输入最后一个代码示例时发生的情况。

<figure class="calibre33">

<h6 id="figure-2-2.-typeerror-reported-by-vscode" class="calibre34">
图2-2. VSCode报告的TypeError</h6>

</figure>

通过为您首选的代码编辑器安装良好的TypeScript扩展，错误会在您*输入代码时*以红色波浪线的形式出现在您的代码下方。这大大加快了编写代码、意识到错误以及更新代码修复错误之间的反馈循环。

错误何时浮现?

当JavaScript抛出异常或执行隐式类型转换时，它是在运行时发生的。这意味着您必须实际运行程序才能获得有用的信号，表明您做了一些无效的操作。在最好的情况下，这意味着作为单元测试的一部分；在最坏的情况下，这意味着来自用户的愤怒邮件。

TypeScript在编译时抛出语法相关错误和类型相关错误。在实践中，这意味着这些类型的错误会在您的代码编辑器中显示，就在您输入时——如果您以前从未使用过增量编译的静态类型语言，这是一种令人惊叹的体验。

话虽如此，有很多错误TypeScript无法在编译时为您捕获——比如堆栈溢出、网络连接中断和格式错误的用户输入——这些仍会导致运行时异常。TypeScript所做的是将大多数在纯JavaScript环境中原本会成为运行时错误的错误转变为编译时错误。

代码编辑器设置

现在您对TypeScript编译器和类型系统的工作原理有了一些直观理解，让我们设置您的代码编辑器，这样我们就可以开始深入研究一些真正的代码了。

首先下载一个代码编辑器来编写您的代码。我喜欢VSCode，因为它提供了特别好的TypeScript编辑体验，但您也可以使用Sublime Text、Atom、Vim、WebStorm或您喜欢的任何编辑器。工程师们对IDE往往非常挑剔，所以我留给您来决定。如果您确实想使用VSCode，请按照[网站](https://code.visualstudio.com/)上的说明进行设置。

TSC本身是一个用TypeScript编写的命令行应用程序，这意味着您需要NodeJS来运行它。按照官方NodeJS [网站](https://nodejs.org)上的说明在您的机器上安装并运行NodeJS。

NodeJS附带NPM，这是一个包管理器，您将使用它来管理项目的依赖关系并协调构建过程。我们将首先使用它来安装TSC和TSLint（TypeScript的代码检查工具）。首先打开您的终端并创建一个新文件夹，然后在其中初始化一个新的NPM项目：

创建一个新文件夹

mkdir chapter-2 cd chapter-2

初始化一个新的NPM项目（按照提示操作）

npm init

安装TSC、TSLint和NodeJS的类型声明

npm install – save-dev typescript tslint @types/node

```
## tsconfig.json

每个TypeScript项目都应该在其根目录中包含一个名为*tsconfig.json*的文件。这个*tsconfig.json*是TypeScript项目定义诸如应该编译哪些文件、将它们编译到哪个目录以及发出哪个版本的JavaScript等内容的地方。

在您的根文件夹中创建一个名为*tsconfig.json*的新文件（`touch tsconfig.json`），然后在您的代码编辑器中打开它并给它以下内容：
```

{ “compilerOptions”: { “lib”: [“es2015”], “module”: “commonjs”, “outDir”: “dist”, “sourceMap”: true,

“strict”: true, “target”: “es2015” }, “include”: [“src”] }

```
让我们简要介绍一些选项以及它们的含义（表2-2）：

| 选项 | 描述 |
|------|------|
| `include` | TSC应该在哪些文件夹中查找您的TypeScript文件？ |
| `lib` | TSC应该假设在您运行代码的环境中存在哪些API？这包括ES5的`Function.prototype.bind`、ES2015的`Object.assign`以及DOM的`document.querySelector`等。 |
| `module` | TSC应该将您的代码编译为哪个模块系统（CommonJS、SystemJS、ES2015等）？ |
| `outDir` | TSC应该将生成的JavaScript代码放在哪个文件夹中？ |
| `strict` | 在检查无效代码时尽可能严格。此选项强制您的所有代码都正确类型化。我们将在书中的所有示例中使用它，您也应该在TypeScript项目中使用它。 |
| `target` | TSC应该将您的代码编译为哪个JavaScript版本（ES3、ES5、ES2015、ES2016等）？ |

表2-2. *tsconfig.json*选项
```

这些只是可用选项中的一部分——*tsconfig.json*支持数十个选项，并且一直在添加新选项。在实践中，您不会经常更改这些选项，除了在切换到新的模块打包器时调整`module`和`target`设置，为浏览器编写TypeScript时向`lib`添加`"dom"`（您将在第12章中了解更多），或者在将现有JavaScript代码迁移到TypeScript时调整`strict`级别（参见"从JavaScript逐步迁移到TypeScript"）。要获取完整的最新支持选项列表，请访问TypeScript网站上的官方文档。

请注意，虽然使用*tsconfig.json*文件配置TSC很方便，因为它让我们可以将配置检入源代码控制，但您也可以从命令行设置TSC的大多数选项。运行`./node_modules/.bin/tsc --help`可获取可用命令行选项列表。

tslint.json

您的项目还应该有一个*tslint.json*文件，其中包含您的TSLint配置，编纂您希望代码遵循的任何风格约定（制表符与空格等）。

注意

使用TSLint是可选的，但强烈建议所有TypeScript项目使用它来强制执行一致的编码风格。最重要的是，它将为您节省在代码审查期间与同事争论代码风格的时间。

以下命令将生成一个带有默认TSLint配置的*tslint.json*文件：

./node_modules/.bin/tslint –init

然后，您可以添加覆盖项以符合您自己的编码风格。例如，我的*tslint.json*看起来像这样：

{ “defaultSeverity”: “error”, “extends”: [“tslint:recommended”], “rules”: { “semicolon”: false, “trailing-comma”: false } }

要获取完整的可用规则列表，请访问TSLint文档。您也可以添加自定义规则，或安装额外的预设（如ReactJS的预设）。

index.ts

现在您已经设置了*tsconfig.json*和*tslint.json*，创建一个包含您的第一个TypeScript文件的*src*文件夹：

mkdir src touch src/index.ts

您的项目文件夹结构现在应该如下所示：

chapter-2/ ├──node_modules/ ├──src/ │ └──index.ts ├──package.json ├──tsconfig.json └──tslint.json

在代码编辑器中打开*src/index.ts*，并输入以下TypeScript代码：

console.log（‘Hello TypeScript!’）

然后，编译并运行您的TypeScript代码：

使用TSC编译您的TypeScript

./node_modules/.bin/tsc

使用NodeJS运行您的代码

node ./dist/index.js

如果您按照这里的所有步骤操作，您的代码应该运行，您应该在控制台中看到一条日志：

Hello TypeScript!

就是这样——您刚刚从头开始设置并运行了您的第一个TypeScript项目。做得好！

提示

由于这可能是您第一次从头开始设置TypeScript项目，我想逐步介绍每个步骤，以便您了解所有的组成部分。下次您可以采取一些快捷方式来更快地完成这项工作：

- 安装[`ts-node`](https://npmjs.org/package/ts-node)，并使用它通过单个命令编译和运行您的TypeScript。

- 使用脚手架工具如 [`typescript-node-starter`](https://github.com/Microsoft/TypeScript-Node-Starter) 来快速生成文件夹结构。

练习

现在环境已经设置好了，在代码编辑器中打开 *src/index.ts*。输入以下代码：

let a = 1 + 2 let b = a + 3 let c = { apple: a, banana: b } let d = c.apple * 4

现在将鼠标悬停在 `a`、`b`、`c` 和 `d` 上，注意 TypeScript 如何为你推断所有变量的类型：`a` 是 `number` 类型，`b` 是 `number` 类型，`c` 是具有特定形状的对象，`d` 也是 `number` 类型（[图2-3]）。

<figure class="calibre33">

<h6 id="figure-2-3.-typescript-inferring-types-for-you"
class="calibre34">图2-3. TypeScript 为你推断类型</h6>

</figure>

稍微调试一下你的代码。看看你能否：

- 让 TypeScript 在你做无效操作时显示红色波浪线（我们称之为"抛出 `TypeError`"）。

- 阅读 `TypeError`，并尝试理解其含义。

- 修复 `TypeError` 并看到红色波浪线消失。

如果你有雄心，尝试编写一段 TypeScript 无法推断类型的代码。

^[1]^ 这个光谱上有各种语言：JavaScript、Python 和 Ruby 在运行时推断类型；Haskell 和 OCaml 在编译时推断和检查缺失的类型；Scala 和 TypeScript 需要一些显式类型并在编译时推断和检查其余部分；Java 和 C 几乎所有东西都需要显式注解，它们在编译时检查这些注解。

^[2]^ 确实，JavaScript 在解析程序后但运行前会暴露语法错误和一些选定的错误（如同一作用域中多个同名的 `const` 声明）。如果你在构建过程中解析 JavaScript（例如使用 Babel），你可以在构建时暴露这些错误。

^[3]^ 增量编译语言在你做小改动时可以快速重新编译，而不必重新编译整个程序（包括你没有触及的部分）。

^[4]^ 这使 TSC 归属于称为*自托管编译器(self-hosting compilers)*的神秘编译器类别，即编译自身的编译器。

^[5]^ 在这个练习中，我们手动创建了 *tsconfig.json*。当你将来设置 TypeScript 项目时，可以使用 TSC 内置的初始化命令为你生成一个：`./node_modules/.bin/tsc --init`。

第3章. 关于类型的一切

在上一章中，我介绍了类型系统的概念，但我从未定义类型系统中的*类型*真正意味着什么。

类型

一组值以及你可以对它们执行的操作。

如果这听起来令人困惑，让我给出几个熟悉的例子：

- `boolean` 类型是所有布尔值的集合（只有两个：`true` 和 `false`）以及你可以对它们执行的操作（如 `||`、`&&` 和 `!`）。

- `number` 类型是所有数字的集合以及你可以对它们执行的操作（如 `+`、`-`、`*`、`/`、`%`、`||`、`&&` 和 `?`），包括你可以调用的方法如 `.toFixed`、`.toPrecision`、`.toString` 等等。

- `string` 类型是所有字符串的集合以及你可以对它们执行的操作（如 `+`、`||` 和 `&&`），包括你可以调用的方法如 `.concat` 和 `.toUpperCase`。

当你看到某个东西是类型 `T` 时，你不仅知道它是一个 `T`，而且你也确切知道*你可以对那个 `T` 做什么*（以及你不能做什么）。记住，重点是使用类型检查器阻止你做无效的事情。类型检查器知道什么是有效的、什么是无效的方法是通过查看你正在使用的类型以及你如何使用它们。

在本章中，我们将浏览 TypeScript 中可用的类型，并介绍你可以对每种类型执行的基本操作。[图3-1] 给出了概述。

<figure class="calibre33">

<h6 id="figure-3-1.-typescripts-type-hierarchy" class="calibre34">
图3-1. TypeScript 的类型层次结构</h6>

</figure>

谈论类型

当程序员谈论类型时，他们共享一个精确、通用的词汇来描述他们的意思。我们将在整本书中使用这个词汇。

假设你有一个函数，它接受某个值并返回该值乘以自身：

function squareOf(n) { return n * n } squareOf(2) // 计算结果为 4 squareOf(‘z’) // 计算结果为 NaN

> 显然，这个函数只对数字有效——如果你向 `squareOf` 传递除数字以外的任何东西，结果都是无效的。所以我们要做的是显式*注解*参数的类型：

function squareOf(n: number) { return n * n } squareOf(2) // 计算结果为 4 squareOf(‘z’) // Error TS2345: Argument of type ‘“z”’ is not assignable to // parameter of type ‘number’.

如果现在我们用除了数字之外的任何类型调用 `squareOf`，TypeScript会立即报错。这是一个简单的例子（我们将在下一章详细讨论函数），但足以介绍讨论TypeScript中类型的几个关键概念。我们可以对最后的代码示例说出以下几点：

1. `squareOf` 的参数 `n` 被*约束为* `number` 类型。
2. 值 `2` 的类型*可分配给*（等价于：*兼容*） `number` 类型。

如果没有类型注解，`squareOf` 的参数是不受约束的，你可以向它传递任何类型的参数。一旦我们对其进行约束，TypeScript就会为我们验证调用函数的每个地方都使用了兼容的参数。在这个例子中，`2` 的类型是 `number`，它可以分配给 `squareOf` 的注解 `number`，所以TypeScript接受我们的代码；但是 `'z'` 是 `string` 类型，它不能分配给 `number`，所以TypeScript会报错。

你也可以从*界限*的角度来思考：我们告诉TypeScript `n` 的*上界*是 `number`，所以我们传递给 `squareOf` 的任何值都必须最多是一个 `number`。如果它超出了 `number`（比如，如果它是一个可能是 `number` 或可能是 `string` 的值），那么它就不能分配给 `n`。

我将在第6章中更正式地定义可分配性、界限和约束。现在，你只需要知道这是我们用来讨论一个类型是否可以在需要特定类型的地方使用的语言。

类型基础

让我们来了解一下TypeScript支持的类型、它们包含的值以及你可以用它们做什么。我们还将介绍一些用于处理类型的基本语言特性：类型别名、联合类型和交集类型。

any

`any` 是类型中的教父。它可以为了代价做任何事情，但除非你完全没有其他选择，否则你不会想要请 `any` 帮忙。在TypeScript中，所有东西在编译时都需要有一个类型，而当你（程序员）和TypeScript（类型检查器）无法确定某个东西是什么类型时，`any` 就是默认类型。它是最后的手段类型，你应该尽可能避免使用它。

为什么应该避免它？还记得类型是什么吗？（它是一组值以及你可以用它们做的事情。）`any` 是*所有*值的集合，你可以用 `any` 做*任何事情*。这意味着如果你有一个 `any` 类型的值，你可以对它进行加法运算、乘法运算、调用 `.pizza()` 方法——任何事情。

`any` 使你的值表现得就像在常规JavaScript中一样，并完全阻止了类型检查器发挥其魔力。当你允许 `any` 进入你的代码时，你就是在盲飞。像避火一样避开 `any`，只在非常、非常不得已的情况下使用它。

在确实需要使用它的极少数情况下，你可以这样做：

```
let a: any = 666            // any
let b: any = ['danger']     // any
let c = a + b               // any
```

注意第三个类型应该报告错误（为什么你要尝试将数字和数组相加？），但没有报告，因为你告诉TypeScript你在添加两个 `any`。如果你想使用 `any`，你必须明确地使用它。当TypeScript推断某个值是 `any` 类型时（例如，如果你忘记注解函数的参数，或者如果你导入了一个未类型化的JavaScript模块），它将抛出编译时异常并在编辑器中给你一个红色波浪线。通过明确地用 `any` 类型注解 `a` 和 `b`（`: any`），你避免了异常——这是你告诉TypeScript你知道自己在做什么的方式。

TSC标志： noImplicitAny

默认情况下，TypeScript是宽松的，不会对它推断为`any`的值进行投诉。要让TypeScript对隐式`any`进行投诉，请确保在你的*tsconfig.json*中启用`noImplicitAny`标志。

`noImplicitAny`是TSC `strict`系列标志的一部分，所以如果你已经在*tsconfig.json*中启用了`strict`（正如我们在”tsconfig.json”中所做的），你就可以开始了。

unknown

如果 any 是教父，那么 unknown 就是《惊爆点》中卧底FBI特工Johnny Utah的基努·里维斯：悠闲自在，与坏人打成一片，但内心深处尊重法律，站在好人一边。对于少数你确实不知道值的类型的情况，不要使用 any，而应该使用 unknown。像 any 一样，它代表任何值，但TypeScript不会让你使用 unknown 类型，直到你通过检查它是什么来细化它（参见”细化”部分）。

unknown 支持哪些操作？你可以比较 unknown 值（使用 ==、===、||、&& 和 ?），对它们取反（使用 !），并通过 JavaScript 的 typeof 和 instanceof 操作符来细化它们（就像你可以对任何其他类型做的那样）。使用 unknown 的方式如下：

```
let a: unknown = 30          // unknown
let b = a === 123            // boolean
let c = a + 10               // Error TS2571: Object is of type
'unknown'.
if (typeof a === 'number') {
  let d = a + 10             // number
}
```

这个例子应该能让你大致了解如何使用 unknown：

1. TypeScript 永远不会推断某些内容为 unknown —你必须显式地注释它（ a ）。
2. 你可以将值与类型为 unknown 的值进行比较（ b ）。
3. 但是，你不能做那些假设 unknown 值是特定类型的操作（ c ）；你必须首先向 TypeScript 证明该值确实是那种类型（ d ）。

boolean

`boolean` 类型有两个值：`true` 和 `false`。你可以比较它们（使用 `==`、`===`、`||`、`&&` 和 `?`），对它们取反（使用 `!`），除此之外没有太多其他操作。使用 `boolean` 的方式如下：

```
let a = true                    // boolean
var b = false                   // boolean
const c = true                  // true
let d: boolean = true           // boolean
let e: true = true              // true
let f: true = false             // Error TS2322: Type 'false' is not
assignable
                                // to type 'true'.
```

这个例子展示了几种告诉 TypeScript 某些内容是 `boolean` 的方法：

1. 你可以让 TypeScript 推断你的值是 `boolean`（`a` 和 `b`）。
2. 你可以让 TypeScript 推断你的值是特定的 `boolean`（`c`）。
3. 你可以明确告诉 TypeScript 你的值是 `boolean`（`d`）。
4. 你可以明确告诉 TypeScript 你的值是特定的 `boolean`（`e` 和 `f`）。

一般来说，你会在程序中使用第一种或第二种方法。很少情况下，你会使用第四种方法—只有当它为你带来额外的类型安全时才会使用（我会在本书中展示这样的例子）。你几乎永远不会使用第三种方法。

第二种和第四种情况特别有趣，因为虽然它们做的事情很直观，但只有极少数编程语言支持这种特性，所以对你来说可能是新的。在那个例子中我做的事情是说：“嘿 TypeScript！看到这里的变量 `e` 吗？`e` 不只是任何普通的 `boolean`—它是特定的 `boolean` 值 `true`。”通过使用值作为类型，我实际上将 `e` 和 `f` 的可能值从所有 `booleans` 限制为各自的一个特定 `boolean`。这个特性被称为*类型字面量(type literals)*。

类型字面量

表示单个值且仅此一个值的类型。

在第四种情况中，我显式地用类型字面量注释了变量，而在第二种情况中，TypeScript 为我推断了字面量类型，因为我使用了 `const` 而不是 `let` 或 `var`。因为 TypeScript 知道一旦原始值用 `const` 赋值，它的值永远不会改变，所以它会为该变量推断出尽可能窄的类型。这就是为什么在第二种情况中 TypeScript 将 `c` 的类型推断为 `true` 而不是 `boolean`。要了解更多关于 TypeScript 为什么对 `let` 和 `const` 推断不同类型的信息，请跳转到[“类型扩展”]。

我们将在本书中重新访问类型字面量。它们是一个强大的语言特性，让你在各种地方都能获得额外的安全性。类型字面量使 TypeScript 在语言世界中独树一帜，是你应该向你的 Java 朋友炫耀的东西。

number

`number` 是所有数字的集合：整数、浮点数、正数、负数、`Infinity`、`NaN` 等等。数字可以做，嗯，数字相关的事情，比如加法（`+`）、减法（`-`）、取模（`%`）和比较（`<`）。让我们看几个例子：

```
let a = 1234                  // number
var b = Infinity * 0.10       // number
const c = 5678                // 5678
let d = a < b                 // boolean
let e: number = 100           // number
let f: 26.218 = 26.218        // 26.218
let g: 26.218 = 10            // Error TS2322: Type '10' is not
assignable
                              // to type '26.218'.
```

就像在 `boolean` 例子中一样，有四种将某些内容类型化为 `number` 的方法：

1. 你可以让 TypeScript 推断你的值是 `number`（`a` 和 `b`）。
2. 你可以使用 `const`，这样 TypeScript 推断你的值是特定的 `number`（`c`）。
3. 你可以明确告诉 TypeScript 你的值是 `number`（`e`）。
4. 你可以明确告诉 TypeScript 你的值是特定的 `number`（`f` 和 `g`）。

与 `boolean` 一样，通常情况下你会让 TypeScript 为你推断类型(第一种方式)。偶尔你会做一些巧妙的编程，需要将 number 类型限制为特定值(第二种或第四种方式)。没有充分理由将某个东西明确类型化为 `number`(第三种方式)。

提示

处理长数字时，使用数字分隔符使这些数字更易读。你可以在类型和值位置都使用数字分隔符：

```
let oneMillion = 1_000_000 // 等同于 1000000
let twoMillion: 2_000_000 = 2_000_000
```

bigint

`bigint` 是 JavaScript 和 TypeScript 的新成员：它让你可以处理大整数而不会遇到舍入错误。虽然 `number` 类型只能表示高达 2^{53} 的整数，但 `bigint` 也可以表示比这更大的整数。`bigint` 类型是所有 BigInt 的集合，支持加法(`+`)、减法(`-`)、乘法(*)、除法(`/`)和比较(`<`)等操作。这样使用：

```
let a = 1234n                 // bigint
const b = 5678n               // 5678n
var c = a + b                 // bigint
let d = a < 1235              // boolean
let e = 88.5n                 // 错误 TS1353: bigint 字面量必须是整数。
let f: bigint = 100n          // bigint
let g: 100n = 100n            // 100n
let h: bigint = 100           // 错误 TS2322: 类型 '100' 不能赋值
                              // 给类型 'bigint'。
```

与 `boolean` 和 `number` 一样，有四种声明 bigint 的方式。尽可能让 TypeScript 推断你的 bigint 类型。

警告

在撰写本文时，`bigint` 尚未得到所有 JavaScript 引擎的原生支持。如果你的应用依赖于 `bigint`，请务必检查它是否被你的目标平台支持。

string

`string` 是所有字符串的集合，以及你可以对它们执行的操作，如连接(`+`)、切片(`.slice`)等。让我们看一些例子:

```
let a = 'hello'             // string
var b = 'billy'             // string
const c = '!'               // '!'
let d = a + ' ' + b + c     // string
let e: string = 'zoom'      // string
let f: 'john' = 'john'      // 'john'
let g: 'john' = 'zoe'       // 错误 TS2322: 类型 "zoe" 不能赋值
                            // 给类型 "john"。
```

与 `boolean` 和 `number` 一样，有四种声明 `string` 类型的方式，你应该尽可能让 TypeScript 为你推断类型。

symbol

`symbol` 是一个相对较新的语言特性，随着最新的主要 JavaScript 修订版本(ES2015)而到来。Symbol 在实践中不经常出现；它们用作对象和映射中字符串键的替代品，在你想要确保人们使用正确的已知键并且不会意外设置键的地方——比如为你的对象设置默认迭代器（`Symbol.iterator`），或在运行时覆盖你的对象是否是某个实例（`Symbol.hasInstance`）。Symbol 具有类型 `symbol`，你可以对它们做的事情并不多：

```
let a = Symbol('a')         // symbol
let b: symbol = Symbol('b') // symbol
var c = a === b             // boolean
let d = a + 'x'             // 错误 TS2469: '+' 运算符不能应用于
                            // 类型 'symbol'。
```

JavaScript 中 `Symbol('a')` 的工作方式是创建一个具有给定名称的新 `symbol`；该 `symbol` 是唯一的，不会等于(用 `==` 或 `===` 比较时)任何其他 `symbol`(即使你创建了一个具有完全相同名称的第二个 symbol！)。类似于当你用 `let` 声明值 `27` 时被推断为 `number`，但当你用 `const` 声明时为特定数字 `27`，symbol 被推断为 `symbol` 类型，但可以明确类型化为 `unique symbol`：

```
const e = Symbol('e')                 // typeof e
const f: unique symbol = Symbol('f') // typeof f
let g: unique symbol = Symbol('f')   // 错误 TS1332: 类型为 'unique
symbol'
                                      // 类型的变量必须是 'const'。
let h = e === e             // boolean
let i = e === f             // 错误 TS2367: 此条件将始终返回
```

```
                        // 'false' 因为类型 'unique symbol' 和
                        // 'unique symbol' 没有重叠。
```

这个例子展示了创建唯一符号的几种方法：

1. 当你声明一个新的 `symbol` 并将其赋值给 `const` 变量（不是 `let` 或 `var` 变量）时，TypeScript 会推断其类型为 `unique symbol`。它会在你的代码编辑器中显示为 `typeof` *你的变量名*，而不是 `unique symbol`。

2. 你可以显式地将 `const` 变量的类型注解为 `unique symbol`。

3. `unique symbol` 总是等于它自身。

4. TypeScript 在编译时知道一个 `unique symbol` 永远不会等于任何其他 `unique symbol`。

把 `unique symbols` 想象成其他字面量类型，如 `1`、`true` 或 `"literal"`。它们是创建代表 `symbol` 特定实例的类型的一种方式。

对象

TypeScript 的对象类型指定对象的形状。值得注意的是，它们无法区分简单对象（如你用 `{}` 创建的那种）和更复杂的对象（用 `new Blah` 创建的那种）。这是有意设计的：JavaScript 通常是*结构化类型*的，所以 TypeScript 更偏向这种编程风格而不是*命名类型*风格。

结构化类型

一种编程风格，你只关心对象具有某些属性，而不关心它的名称是什么（命名类型）。在某些语言中也称为*鸭子类型*（或者，不以貌取人）。

在 TypeScript 中有几种方法使用类型来描述对象。第一种是将值声明为 `object`：

let a: object = { b: ‘x’ }

当你访问 `b` 时会发生什么？

a.b // Error TS2339: Property ‘b’ does not exist on type ‘object’.

等等，这没什么用！如果你不能对它做任何事情，那么将某个东西类型化为 `object` 有什么意义？

嗯，这是个很好的观点，有抱负的 TypeScript 开发者！实际上，`object` 比 `any` 稍微窄一点，但差别不大。`object` 对它描述的值告诉你的信息不多，只是说这个值是一个 JavaScript 对象（并且它不是 `null`）。

如果我们省略显式注解，让 TypeScript 自己处理会怎样？

let a = { b: 'x' } // {b: string} a.b // string

let b = { c: { d: 'f' } } // {c: {d: string}}

好的！你刚刚发现了类型化对象的第二种方式：对象字面量语法（不要与类型字面量混淆）。你可以让 TypeScript 为你推断对象的形状，或者在花括号(`{}`)内显式描述它：

let a: {b: number} = { b: 12 } // {b: number}

使用 const 声明对象时的类型推断

如果我们使用 `const` 来声明对象会发生什么？

const a: {b: number} = { b: 12 } // 仍然是 {b: number}

你可能会惊讶 TypeScript 将 `b` 推断为 `number`，而不是字面量 `12`。毕竟，我们学过当声明 `number` 或 `string` 时，我们对 `const` 或 `let` 的选择会影响 TypeScript 如何推断我们的类型。

与我们到目前为止看过的基本类型---`boolean`、`number`、`bigint`、`string` 和 `symbol`---不同，用 `const` 声明对象不会提示 TypeScript 更窄地推断其类型。这是因为 JavaScript 对象是可变的，据 TypeScript 所知，你可能在创建它们后更新它们的字

段。

我们在"类型扩展"中更深入地探讨这个想法---包括如何选择更窄的推断。

对象字面量语法说："这是一个具有这种形状的东西。"这个东西可能是对象字面量，也可能是类：

let c: { firstName: string lastName: string } = { firstName: ‘john’, lastName: ‘barrowman’ }

class Person { constructor(public firstName: string, // public 是 // this.firstName = firstName 的简写 public lastName: string) {} } c = new Person(‘matt’, ‘smith’) // OK

`{firstName: string, lastName: string}` 描述了对象的*形状*，上例中的对象字面量和类实例都满足该形状，所以 TypeScript 允许我们将 `Person` 赋值给 `c`。

让我们探索一下当我们添加额外属性或遗漏必需属性时会发生什么：

let a: {b: number}

a = {} // Error TS2741: Property ‘b’ is missing in type ‘{}’ // but required in type ‘{b: number}’.

a = { b: 1,

c: 2 // 错误 TS2322: Type ‘{b: number; c: number}’ is not assignable } // to type ‘{b: number}’. Object literal may only specify known // properties, and ‘c’ does not exist in type ‘{b: number}’.

明确赋值 {#definite-assignment .calibre29}

这是我们看到的第一个例子，其中我们首先声明一个变量（`a`），然后用值初始化它（`{}` 和 `{b: 1, c: 2}`）。这是一个常见的 JavaScript 模式，TypeScript 也支持它。

当你在一个地方声明变量并稍后初始化时，TypeScript 会确保你的变量在使用之前已经*明确赋值*：

let i: number let j = i * 3 // 错误 TS2454: Variable ‘i’ is used // before being assigned.

别担心，即使你省略显式类型注解，TypeScript 也会为你强制执行这一点：

let i let j = i * 3 // 错误 TS2532: Object is possibly // ‘undefined’.

默认情况下，TypeScript 对对象属性非常严格——如果你说对象应该有一个名为 `b` 的属性，它是 `number` 类型，TypeScript 期望有且只有 `b`。如果缺少 `b`，或者有额外的属性，TypeScript 会报错。

你能告诉 TypeScript 某些东西是可选的，或者可能有比你计划的更多属性吗？当然可以：

let a: { b: number c?: string [key: number]: boolean }

[![1](images/000000.png)]{#calibre_link-41 .calibre4}

: `a` 有一个属性 `b`，它是 `number` 类型。

[![2](images/000001.png)]{#calibre_link-42 .calibre4}

: `a` 可能有一个属性 `c`，它是 `string` 类型。如果设置了 `c`，它可能是 `undefined`。

[![3](images/000002.png)]{#calibre_link-43 .calibre4}

: `a` 可能有任意数量的数值属性，它们是 `boolean` 类型。

让我们看看可以将什么类型的对象赋给 `a`：

a = {b: 1} a = {b: 1, c: undefined} a = {b: 1, c: ‘d’ } a = {b: 1, 10: true} a = {b: 1, 10: true, 20: false} a = {10: true} // 错误 TS2741: Property ‘b’ is missing in type // ‘{10: true}’. a = {b: 1, 33: ‘red’ } // 错误 TS2741: Type ‘string’ is not assignable // to type ‘boolean’.

索引签名 {#index-signatures .calibre29}

`[key: T]: U` 语法称为*索引签名*(index signature)，这是告诉 TypeScript 给定对象可能包含更多键的方式。读法是："对于这个对象，所有类型为 `T` 的键必须具有类型为 `U` 的值。"索引签名让你可以安全地向对象添加更多键，除了你明确声明的任何键之外。

索引签名需要记住一个规则：索引签名键的类型（`T`）必须可赋值给 `number` 或 `string`。

还要注意，你可以为索引签名键的名称使用任何词——它不必是 `key`：

let airplaneSeatingAssignments: { [seatNumber: string]: string } = { ‘34D’: ‘Boris Cherny’, ‘34E’: ‘Bill Gates’ }

可选（`?`）并不是声明对象类型时唯一可以使用的修饰符。你还可以用 `readonly` 修饰符将字段标记为只读（也就是说，你可以声明一个字段在被赋予初始值后不能被修改——有点像对象属性的 `const`）：

let user: { readonly firstName: string } = { firstName: ‘abby’ }

user.firstName // string user.firstName = ‘abbey with an e’ // 错误 TS2540: Cannot assign to ‘firstName’ because it // is a read-only property.

对象字面量记法有一个特殊情况：空对象类型（`{}`）。除了 `null` 和 `undefined` 之外，每种类型都可以赋值给空对象类型，这可能使其难以使用。尽可能避免使用空对象类型：

let danger: {} danger = {} danger = {x: 1} danger = [] danger = 2

作为对象的最后说明，值得一提的是将某些东西类型化为对象的另一种方式：`Object`。这与使用 `{}` 几乎相同，最好避免使用。

总结一下，在 TypeScript 中有四种声明对象的方式：

1. 对象字面量记法（如 `{a: string}`），也称为*形状*(shape)。当你知道对象可能具有哪些字段，或者当对象的所有值都具有相同类型时使用这种方式。

2. 空对象字面量记法（`{}`）。尽量避免使用。

3. `object` 类型。当你只想要一个对象，而不关心它有哪些字段时使用这种方式。

4. `Object` 类型。请尽量避免使用它。

在你的 TypeScript 程序中，你应该几乎总是坚持使用第一种和第三种方式。小心避免第二种和第四种方式——使用代码检查工具来警告它们，在代码审查中抱怨它们，打印海报——使用你的团队首选的工具让它们远离你的代码库。

[表 3-1] 是前面列表中选项 2-4 的便利参考。

值	`{}`	`object`	`Object`
`{}`	是	是	是
`['a']`	是	是	是
`function () {}`	是	是	是
`new String('a')`	是	是	是
`'a'`	是	**否**	是
`1`	是	**否**	是
`Symbol('a')`	是	**否**	是
`null`	**否**	**否**	**否**
`undefined`	**否**	**否**	**否**

: [表 3-1.] 该值是有效的对象吗？

插曲：类型别名、联合类型和交集类型

你正在迅速成为一名经验丰富的 TypeScript 程序员。你已经见过几种类型以及它们如何工作，现在对类型系统、类型和安全性的概念很熟悉了。是时候深入了解了。

如你所知，如果你有一个值，你可以对其执行某些操作，这取决于其类型允许什么。例如，你可

以使用 `+` 来添加两个数字，或者使用 `.toUpperCase` 来将字符串转换为大写。

如果你有一个*类型*，你也可以对其执行一些操作。我将在这里介绍一些类型级别的操作--本书后面还会有更多内容，但这些操作非常常见，我想尽早介绍它们。

类型别名

就像你可以使用变量声明（`let`、`const` 和 `var`）来声明一个别名值的变量一样，你可以声明一个指向类型的类型别名。它看起来像这样：

```
type Age = number

type Person = { name: string age: Age }
```

`Age` 只是一个 `number`。它也有助于使 `Person` 形状的定义更容易理解。别名从不被 TypeScript 推断，所以你必须明确地类型化它们：

```
let age: Age = 55

let driver: Person = { name: 'James May' age: age }
```

因为 `Age` 只是 `number` 的别名，这意味着它也可以分配给 `number`，所以我们可以将其重写为：

```
let age = 55

let driver: Person = { name: 'James May' age: age }
```

无论你在哪里看到使用类型别名，你都可以替换为它别名的类型，而不会改变程序的含义。

像 JavaScript 变量声明（`let`、`const` 和 `var`）一样，你不能声明两次类型：

type Color = ‘red’ type Color = ‘blue’ // Error TS2300: Duplicate identifier ‘Color’.

和 `let` 和 `const` 一样，类型别名是块作用域的。每个块和每个函数都有自己的作用域，内部类型别名声明会遮蔽外部声明：

type Color = ‘red’

let x = Math.random() < .5

if (x) { type Color = ‘blue’ // This shadows the Color declared above. let b: Color = ‘blue’ } else { let c: Color = ‘red’ }

类型别名对于 DRY 原则（Don't Repeat Yourself）处理重复的复杂类型很有用，并且可以清楚地表明变量的用途（有些人更喜欢描述性的类型名称而不是描述性的变量名称！）。在决定是否为类型设置别名时，请使用与决定是否将值提取到自己的变量中相同的判断。

联合类型和交集类型

如果你有两个东西 `A` 和 `B`，那些东西的*联合*是它们的和（`A` 或 `B` 或两者中的一切），*交集*是它们的共同点（`A` 和 `B` 中都有的一切）。思考这个最简单的方法是使用集合。在[图 3-2]中，我将集合表示为圆圈。左边是两个集合的联合或*和*；右边是它们的交集或*乘积*。

<figure class="calibre33">

图3-2. 并集(|)和交集(&)

TypeScript提供了特殊的类型运算符来描述类型的并集和交集：`|`表示并集，`&`表示交集。由于类型很像集合，我们可以用同样的方式来思考它们：

type Cat = {name: string, purrs: boolean} type Dog = {name: string, barks: boolean, wags: boolean} type CatOrDogOrBoth = Cat | Dog type CatAndDog = Cat & Dog

如果某个东西是`CatOrDogOrBoth`，你对它了解什么？你知道它有一个字符串类型的`name`属性，除此之外就不多了。另一方面，你可以将什么赋值给`CatOrDogOrBoth`？嗯，可以是`Cat`、`Dog`，或者两者都是：

// Cat let a: CatOrDogOrBoth = { name: ‘Bonkers’, purrs: true }

// Dog a = { name: ‘Domino’, barks: true, wags: true }

// Both a = { name: ‘Donkers’, barks: true, purrs: true, wags: true }

这点值得重申：具有并集类型(`|`)的值不一定是并集中的某个特定成员；实际上，它可以同时是两个成员！

另一方面，你对`CatAndDog`了解什么？你的犬猫混合超级宠物不仅有`name`，还可以打呼噜、吠叫和摇尾巴：

let b: CatAndDog = { name: ‘Domino’, barks: true, purrs: true, wags: true }

并集比交集出现得更频繁。以这个函数为例：

function trueOrNull(isTrue: boolean) { if (isTrue) { return ‘true’ } return null }

这个函数返回值的类型是什么？嗯，它可能是`string`，也可能是`null`。我们可以将其返回类型表达为：

type Returns = string | null

那这个呢?

function(a: string, b: number) { return a || b }

如果`a`为真，则返回类型是`string`，否则是`number`：换句话说，是`string | number`。

并集自然出现的最后一个地方是数组（特别是异构类型的数组），我们接下来会讨论。

数组

与JavaScript一样，TypeScript数组是特殊的对象类型，支持连接、推入、搜索和切片等操作。看例子：

let a = [1, 2, 3] // number[] var b = [‘a’, ‘b’] // string[] let c: string[] = [‘a’] // string[] let d = [1, ‘a’] // (string | number)[] const e = [2, ‘b’] // (string | number)[]

let f = [‘red’] f.push(‘blue’) f.push(true) // Error TS2345: Argument of type ‘true’ is not // assignable to parameter of type ‘string’.

let g = [] // any[] g.push(1) // number[] g.push(‘red’) // (string | number)[]

let h: number[] = [] // number[] h.push(1) // number[] h.push(‘red’) // Error TS2345: Argument of type ‘“red”’ is not // assignable to parameter of type ‘number’.

注意

TypeScript支持两种数组语法：`T[]`和`Array<T>`。它们在含义和性能上都是相同的。本书使用`T[]`语法是为了简洁，但你可以为自己的代码选择任何喜欢的风格。

当你阅读这些例子时，注意除了`c`和`h`之外的所有内容都是隐式类型。你还会注意到TypeScript对可以在数组中放入什么有规则。

经验法则是保持数组*同构*。也就是说，不要在单个数组中混合苹果、橘子和数字——尽量设计你的程序，使数组的每个元素都具有相同的类型。原因是否则你将不得不做更多工作来向TypeScript证明你所做的是安全的。

要了解为什么当数组是同构的时候事情会更容易，看看例子`f`。我用字符串`'red'`初始化了一个数组（在声明数组时它只包含字符串，所以TypeScript推断它必须是字符串数组）。然后我将`'blue'`推入其中；`'blue'`是字符串，所以TypeScript让它通过了。然后我试图将`true`推入数组，但失败了！为什么？因为`f`是字符串数组，而`true`不是字符串。

另一方面，当我初始化`d`时，我给了它一个`number`和一个`string`，所以TypeScript推断它必须是`number | string`类型的数组。因为每个元素可能是数字或字符串，你必须在使用之前检查它是哪种类型。例如，假设你想映射该数组，将每个字母转换为大写，将每个数字乘以三：

let d = [1, ‘a’]

d.map(_ => { if (typeof _ === ‘number’) { return _ * 3 } return _.toUpperCase() })

你需要使用 `typeof` 查询每个项目的类型，检查它是 `number` 还是 `string`，然后才能对其进行任何操作。

与对象一样，使用 `const` 创建数组不会提示 TypeScript 更窄地推断其类型。这就是为什么 TypeScript 推断 `d` 和 `e` 都是 `number | string` 数组的原因。

`g` 是特殊情况：当你初始化一个空数组时，TypeScript 不知道数组元素应该是什么类型，所以它给你好处并将它们设为 `any`。当你操作数组并向其添加元素时，TypeScript 开始拼凑你的数组类型。一旦你的数组离开定义它的作用域（例如，如果你在函数中声明它，然后返回它），TypeScript 将为其分配一个不能再扩展的最终类型：

function buildArray() { let a = [] // any[] a.push(1) // number[] a.push(‘x’) // (string | number)[] return a }

let myArray = buildArray() // (string | number)[] myArray.push(true) // Error 2345: Argument of type ‘true’ is not // assignable to parameter of type ‘string | number’.

因此，就 `any` 的使用而言，这个不应该让你太担心。

元组

元组是 `array` 的子类型。它们是一种特殊的数组类型方式，具有固定长度，其中每个索引的值具有特定的已知类型。与大多数其他类型不同，元组在声明时必须显式类型化。这是因为 JavaScript 语法对于元组和数组是相同的（都使用方括号），而 TypeScript 已经有从方括号推断数组类型的规则：

let a: number = [1]

// A tuple of [first name, last name, birth year] let b: [string, string, number] = [‘malcolm’, ‘gladwell’, 1963]

b = [‘queen’, ‘elizabeth’, ‘ii’, 1926] // Error TS2322: Type ‘string’ is not // assignable to type ‘number’.

元组也支持可选元素。就像在对象类型中一样，`?` 表示"可选"：

// An array of train fares, which sometimes vary depending on direction let trainFares: [number, number?][] = [[3.75], [8.25, 7.70], [10.50]]

// Equivalently: let moreTrainFares: (number | [number, number])[] = [// …]

元组也支持剩余元素，你可以使用它来为具有最小长度的元组进行类型化：

// A list of strings with at least 1 element let friends: [string, …string[]] = [‘Sara’, ‘Tali’, ‘Chloe’, ‘Claire’]

// A heterogeneous list let list: [number, boolean, …string[]] = [1, false, ‘a’, ‘b’, ‘c’]

元组类型不仅安全地编码异构列表，而且还捕获它们类型化的列表的长度。这些特性为你提供了比普通数组更多的安全性——经常使用它们。

只读数组和元组

虽然常规数组是可变的（意味着你可以对它们进行 `.push`、`.splice` 并就地更新它们），这可能是你大部分时间想要的，但有时你想要一个不可变数组——一个你可以更新以产生新数组，保持原数组不变的数组。

TypeScript 自带一个 `readonly` 数组类型，你可以使用它来创建不可变数组。只读数组就像常规数组一样，但你不能就地更新它们。要创建只读数组，使用显式类型注解；要更新只读数组，使用非变异方法如 `.concat` 和 `.slice`，而不是变异方法如 `.push` 和 `.splice`：

let as: readonly number[] = [1, 2, 3] // readonly number[] let bs: readonly number[] = as.concat(4) // readonly number[] let three = bs[2] // number as[4] = 5 // Error TS2542: Index signature in type // ‘readonly number[]’ only permits reading. as.push(6) // Error TS2339: Property ‘push’ does not // exist on type ‘readonly number[]’.

与 `Array` 类似，TypeScript 提供了几种更冗长的方式来声明只读数组和元组：

type A = readonly string[] // readonly string[] type B = ReadonlyArray // readonly string[] type C = Readonly<string[]> // readonly string[]

type D = readonly [number, string] // readonly [number, string] type E = Readonly<[number, string]> // readonly [number, string]

使用哪种语法——简洁的 `readonly` 修饰符，还是更冗长的 `Readonly` 或 `ReadonlyArray` 工具类型——完全取决于个人喜好。

请注意，虽然只读数组在某些情况下可以通过避免可变性来让代码更容易推理，但它们底层仍然是常规的 JavaScript 数组。这意味着即使是对数组的小幅更新也需要先复制原始数组，如果不小心处理，这可能会影响应用程序的运行时性能。对于小数组，这种开销很少引起注意，但对于较大的数组，开销可能变得很显著。

提示

如果你计划大量使用不可变数组，请考虑使用更高效的实现，比如 Lee Byron 出色的 [`immutable`](https://www.npmjs.com/package/immutable)。

null、undefined、void 和 never

JavaScript 有两个值来表示某些东西的缺失：`null` 和 `undefined`。TypeScript 支持这两个值，并且也有对应的类型——你能猜到它们叫什么吗？没错，这些类型也分别叫做 `null` 和 `undefined`。

它们都是特殊类型，因为在 TypeScript 中，`undefined` 类型的唯一值就是 `undefined`，`null` 类型的唯一值就是 `null`。

JavaScript 程序员通常可以互换使用这两个值，不过有一个值得一提的微妙语义区别：`undefined` 意味着某些东西还没有被定义，而 `null` 意味着值的缺失（比如当你试图计算一个值，但在过程中遇到了错误）。这些只是约定，TypeScript 不会强制你遵循它们，但这可能是一个有用的区别。

除了 `null` 和 `undefined`，TypeScript 还有 `void` 和 `never`。这些是非常具体的、专用的类型，在不存在的不同类型之间划出了更细的界线：`void` 是不显式返回任何内容的函数的返回类型（例如 `console.log`），而 `never` 是根本不会返回的函数的类型（比如抛出异常的函数，或永远运行的函数）：

```
// (a) 返回数字或 null 的函数 function a(x: number) { if (x < 10) { return x } return null }

// (b) 返回 undefined 的函数 function b() { return undefined }

// (c) 返回 void 的函数 function c() { let a = 2 + 2 let b = a * a }

// (d) 返回 never 的函数 function d() { throw TypeError('I always error') }

// (e) 另一个返回 never 的函数 function e() { while (true) { doSomething() } }
```

(a) 和 (b) 分别显式返回 `null` 和 `undefined`。(c) 返回 `undefined`，但它不是通过显式的 `return` 语句来实现的，所以我们说它返回 `void`。(d) 抛出异常，(e) 永远运行——两者都不会返回，所以我们说它们的返回类型是 `never`。

如果 `unknown` 是所有其他类型的超类型，那么 `never` 就是所有其他类型的子类型。我们称之为*底类型*。这意味着它可以赋值给所有其他类型，`never` 类型的值可以在任何地方安全使用。这主要具有理论意义，但当你与其他语言专家讨论 TypeScript 时会遇到这个概念。

表 3-2 总结了四种缺失类型的用法。

```
| 类型           | 含义                                  |
|---------------|--------------------------------------|
| `null`        | 值的缺失                               |
| `undefined`   | 尚未被赋值的变量                        |
| `void`        | 没有 `return` 语句的函数                |
| `never`       | 永远不会返回的函数                      |

`never`       永不返回的函数
```

: [表3-2.]表示缺失某些东西的类型 {#calibre_link-54}

严格空值检查 {#strict-null-checking .calibre29}

在较旧版本的TypeScript中（或将TSC的`strictNullChecks`选项设置为`false`时），`null`的行为略有不同：它是除`never`之外所有类型的子类型。这意味着每个类型都是可空的，如果不首先检查是否为`null`，你永远无法真正信任任何变量的类型。例如，如果有人将变量`pizza`传递给你的函数，并且你想在其上调用`.addAnchovies`方法，你必须首先检查你的`pizza`是否为`null`，然后才能为其添加美味的小鱼。在实践中，对每个变量都这样做真的很繁琐，所以人们经常忘记实际检查。然后，当某些东西真的为`null`时，你会在运行时遇到可怕的空指针异常：

function addDeliciousFish(pizza: Pizza) { return pizza.addAnchovies() // Uncaught TypeError: Cannot read } // property ‘addAnchovies’ of null

// TypeScript lets this fly with strictNullChecks = false addDeliciousFish(null)

`null`被在1960年代引入它的人称为["十亿美元的错误"](http://bit.ly/2WEdZNO)。`null`的问题在于它是大多数语言的类型系统无法表达且不会检查的东西；所以当程序员试图对他们认为已定义的变量做某事，但在运行时实际上是`null`时，代码会抛出运行时异常！

为什么？别问我，我只是写这本书的人。但是语言正在逐步将`null`编码到它们的类型系统中，TypeScript就是如何正确实现这一点的绝佳例子。如果目标是在用户遇到错误之前在编译时捕获尽可能多的bug，那么能够在类型系统中检查`null`是不可或缺的。

枚举 {#enums .calibre17}

枚举是一种*枚举*类型可能值的方式。它们是将键映射到值的无序数据结构。可以将它们想象为键在编译时固定的对象，因此TypeScript可以在你访问时检查给定的键是否确实存在。

有两种枚举：从字符串映射到字符串的枚举，以及从字符串映射到数字的枚举。它们看起来像这样：

enum Language { English, Spanish, Russian }

注意 {#note-6 .calibre22}

按照约定，枚举名称采用大写和单数形式。它们的键也是大写的。

TypeScript将自动为枚举的每个成员推断一个数字作为值，但你也可以显式设置值。让我们明确说明TypeScript在上一个例子中推断的内容：

enum Language { English = 0, Spanish = 1, Russian = 2 }

要从枚举中检索值，你可以使用点记号或方括号记号访问它——就像你从常规对象获取值一样：

let myFirstLanguage = Language.Russian // Language let mySecondLanguage = Language[‘English’] // Language

你可以将枚举拆分到多个声明中，TypeScript会自动为你合并它们（要了解更多信息，请跳到["声明合并"]）。请注意，当你拆分枚举时，TypeScript只能为其中一个声明推断值，因此为每个枚举成员显式赋值是一个好习惯：

enum Language { English = 0, Spanish = 1 }

enum Language { Russian = 2 }

你可以使用计算值，并且不必定义所有值（TypeScript会尽力推断缺失的部分）：

enum Language { English = 100, Spanish = 200 + 300, Russian // TypeScript infers 501 (the next number after 500) }

你也可以为枚举使用字符串值，甚至混合字符串和数字值：

enum Color { Red = ‘#c10000’, Blue = ‘#007ac1’, Pink = 0xc10050, // A hexadecimal literal White = 255 // A decimal literal }

let red = Color.Red // Color let pink = Color.Pink // Color

为了方便起见，TypeScript允许你通过值和键访问枚举，但这很快就会变得不安全：

let a = Color.Red // Color let b = Color.Green // Error TS2339: Property ‘Green’ does not exist // on type ‘typeof Color’. let c = Color[0] // string let d = Color[6] // string (!!!)

你不应该能够获取 `Color[6]`，但 TypeScript 没有阻止你！我们可以通过选择使用 `const enum` 而不是普通枚举来让 TypeScript 防止这种不安全的访问，从而使用更安全的枚举行为子集。让我们重写之前的 `Language` 枚举：

const enum Language { English, Spanish, Russian }

// 访问有效的枚举键 let a = Language.English // Language

// 访问无效的枚举键 let b = Language.Tagalog // Error TS2339: Property ‘Tagalog’ does not exist // on type ‘typeof Language’.

// 访问有效的枚举值 let c = Language[0] // Error TS2476: A const enum member can only be // accessed using a string literal.

// 访问无效的枚举值 let d = Language[6] // Error TS2476: A const enum member can only be // accessed using a string literal.

`const enum` 不允许你进行反向查找，因此行为更像普通的 JavaScript 对象。它默认情况下也不会生成任何 JavaScript 代码，而是在使用枚举成员的地方内联其值（例如，TypeScript 会将每次出现的 `Language.Spanish` 替换为其值 `1`）。

TSC 标志：preserveConstEnums

`const enum` 内联可能会在你从其他人的 TypeScript 代码中导入 `const enum` 时导致安全问题：如果枚举作者在你编译了 TypeScript 代码后更新了他们的 `const enum`，那么你的枚举版本和他们的版本可能在运行时指向不同的值，而 TypeScript 对此一无所知。

如果你使用 `const enum`，请小心避免内联它们，并且只在你控制的 TypeScript 程序中使用它们：避免在你计划发布到 NPM 或作为库供他人使用的程序中使用它们。

要为 `const enum` 启用运行时代码生成，请在你的 *tsconfig.json* 中将 `preserveConstEnums` TSC 设置切换为 `true`：

{ “compilerOptions”: { “preserveConstEnums”: true } }

让我们看看如何使用 `const enum`：

const enum Flippable { Burger, Chair, Cup, Skateboard, Table }

function flip(f: Flippable) { return ‘flipped it’ }

flip(Flippable.Chair) // ‘flipped it’ flip(Flippable.Cup) // ‘flipped it’ flip(12) // ‘flipped it’ (!!!)

一切看起来都很好——`Chairs` 和 `Cups` 完全按预期工作……直到你意识到所有数字也都可以赋值给枚举！这种行为是 TypeScript 可赋值性规则的不幸后果，要修复它，你必须特别小心，只使用字符串值的枚举：

const enum Flippable { Burger = ‘Burger’, Chair = ‘Chair’, Cup = ‘Cup’, Skateboard = ‘Skateboard’, Table = ‘Table’ }

function flip(f: Flippable) { return ‘flipped it’ }

flip(Flippable.Chair) // ‘flipped it’ flip(Flippable.Cup) // ‘flipped it’ flip(12) // Error TS2345: Argument of type ‘12’ is not // assignable to parameter of type ‘Flippable’. flip(‘Hat’) // Error TS2345: Argument of type ‘“Hat”’ is not // assignable to parameter of type ‘Flippable’.

只需要枚举中有一个讨厌的数值就能让整个枚举变得不安全。

警告

由于安全使用枚举会带来所有这些陷阱，我建议你远离它们——在 TypeScript 中有很多更好的方式来表达自己。

如果同事坚持使用枚举而你无法改变他们的想法，请确保在他们外出时悄悄合并一些 TSLint 规则来警告数值和非 `const` 枚举。

总结

简而言之，TypeScript 带有一堆内置类型。你可以让 TypeScript 从你的值推断类型，或者你可以显式地为你的值指定类型。`const` 会推断更具体的类型，`let` 和 `var` 推断更通用的类型。大多数类型都有通用和更具体的对应类型，后者是前者的子类型（见 [表 3-3]）。

类型	子类型
`boolean`	布尔字面量
`bigint`	BigInt字面量
`number`	数字字面量
`string`	字符串字面量
`symbol`	`unique symbol`
`object`	对象字面量
Array	元组
`enum`	`const enum`

表3-3. 类型及其更具体的子类型

练习

1. 对于以下每个值，TypeScript会推断出什么类型?

   1. `let a = 1042`
   2. `let b = 'apples and oranges'`
   3. `const c = 'pineapples'`
   4. `let d = [true, true, false]`
   5. `let e = {type: 'ficus'}`
   6. `let f = [1, false]`
   7. `const g = [3]`
   8. `let h = null` (在你的代码编辑器中尝试这个，如果结果让你惊讶，请跳到"类型拓宽"部分！)

2. 为什么以下每个代码会抛出它所示的错误?

   1. ```
      let i: 3 = 3
      i = 4 // Error TS2322: Type '4' is not assignable to type '3'.
      ```

   2. ```
      let j = [1, 2, 3]
      j.push(4)
      j.push('5') // Error TS2345: Argument of type '"5"' is not
                  // assignable to parameter of type 'number'.
      ```

   3. ```
      let k: never = 4 // Error TSTS2322: Type '4' is not assignable
                       // to type 'never'.
      ```

   4. ```
      let l: unknown = 4
      let m = l * 2 // Error TS2571: Object is of type 'unknown'.
      ```

^[1]^ 几乎是这样。当`unknown`是联合类型的一部分时，联合的结果将是`unknown`。你将在"联合类型和交集类型"中了解更多关于联合类型的内容。

^[2]^ JavaScript中的对象使用字符串作为键；数组是使用数值键的特殊对象类型。

^[3]^ 有一个细微的技术差异：`{}`允许你为`Object`原型上的内置方法定义任何你想要的

类型，如`.toString`和`.hasOwnProperty`（访问[MDN](https://mzl.la/2VSuDJz)了解更多关于原型的内容），而`Object`强制要求你声明的类型可分配给`Object`原型上的那些类型。例如，这段代码通过类型检查：`let a: {} = {toString() { return 3 }}`。但如果你将类型注解改为`Object`，TypeScript会报错：`let b: Object = {toString() { return 3 }}`会导致`Error TS2322: Type 'number' is not assignable to type 'string'`。

^[4]^ DRY这个缩写代表"Don't Repeat Yourself"（不要重复自己）——代码不应该重复的理念。这个概念由Andrew Hunt和David Thomas在他们的书《程序员修炼之道：从小工到专家》(The Pragmatic Programmer: From Journeyman to Master)（Addison-Wesley）中提出。

^[5]^ 跳到"可辨识联合类型"来学习如何提示TypeScript你的联合是不相交的，该联合类型的值必须是其中一个，而不是两个都是。

^[6]^ 底部类型的理解方式是作为没有值的类型。底部类型对应于总是为假的数学命题。

第4章 函数

在上一章中，我们介绍了TypeScript类型系统的基础：原始类型、对象、数组、元组和枚举，以及TypeScript类型推断的基础和类型可分配性的工作原理。现在你已经准备好学习TypeScript的杰作（或者如果你是函数式程序员的话，可以说是存在的理由）：函数。本章将涵盖的一些主题包括：

- 在TypeScript中声明和调用函数的不同方式

- 签名重载

- 多态函数

- 多态类型别名

声明和调用函数

在JavaScript中，函数是一等对象。这意味着你可以完全像使用任何其他对象一样使用它们：将它们赋值给变量，将它们传递给其他函数，从函数中返回它们，将它们赋值给对象和原型，向它们写入属性，读取这些属性，等等。在JavaScript中你可以对函数做很多事情，TypeScript用其丰富的类型系统对所有这些事情进行建模。

以下是TypeScript中函数的样子（这从上一章应该看起来很熟悉）：

function add(a: number, b: number) { return a + b }

你通常会显式注释函数参数（在这个例子中是`a`和`b`）——TypeScript总是会在你的函数体中推断类型，但在大多数情况下它不会推断你的参数类型，除了一些它可以从上下文推断类型的特殊情况（更多内容见"上下文类型"）。返回类型是被推断的，但如果你愿意，你也可以显式注释它：

function add(a: number, b: number): number { return a + b }

注意

在本书中，我会在有助于你这个读者理解函数作用的地方显式注释返回类型。否则我会省略注释，因为TypeScript已经为我们推断了它们，而且我们为什么要重复工作呢?

最后一个例子使用了*命名函数语法*来声明函数，但JavaScript和TypeScript支持至少五种方式来实现：

// 命名函数 function greet(name: string) { return ‘hello’ + name }

// 函数表达式 let greet2 = function(name: string) { return ‘hello’ + name }

// 箭头函数表达式 let greet3 = (name: string) => { return ‘hello’ + name }

// 简写箭头函数表达式 let greet4 = (name: string) => ‘hello’ + name

// 函数构造器 let greet5 = new Function(‘name’, ‘return “hello” + name’)

除了函数构造器(function constructor)（除非你被蜜蜂追赶否则不应该使用，因为它们完全不安全），所有这些语法都被TypeScript以类型安全的方式支持，并且它们都遵循相同的规则：参数通常需要强制类型注解，返回类型的注解是可选的。

注意

术语的快速复习：

- 参数(parameter)是函数运行所需的数据片段，在函数声明中声明。也称为*形式参数*。

- 参数(argument)是您在调用函数时传递给函数的数据片段。也称为*实际参数*。

当您在TypeScript中调用函数时，不需要提供任何额外的类型信息--只需传入一些参数，TypeScript就会检查您的参数是否与函数参数的类型兼容：

add(1, 2) // 求值为 3 greet（‘Crystal’) // 求值为 ‘hello Crystal’

当然，如果您忘记了一个参数，或者传递了错误类型的参数，TypeScript会很快指出：

add(1) // Error TS2554: Expected 2 arguments, but got 1. add(1, ‘a’) // Error TS2345: Argument of type ‘“a”’ is not assignable // to parameter of type ‘number’.

可选参数和默认参数

就像在对象和元组类型中一样，您可以使用`?`将参数标记为可选。在声明函数参数时，必需参数必须放在前面，后跟可选参数：

function log(message: string, userId?: string) { let time = new Date().toLocaleTimeString() console.log(time, message, userId || ‘Not signed in’) }

log(‘Page loaded’) // 记录 “12:38:31 PM Page loaded Not signed in” log(‘User signed in’, ‘da763be’) // 记录 “12:38:31 PM User signed in da763be”

就像在JavaScript中一样，您可以为可选参数提供默认值。语义上它类似于使参数可选，因为调用者不再需要传入它（不同之处在于默认参数不必在参数列表的末尾，而可选参数必须）。

例如，我们可以将`log`重写为：

function log(message: string, userId = ‘Not signed in’) { let time = new Date().toISOString() console.log(time, message, userId) }

log(‘User clicked on a button’, ‘da763be’) log(‘User signed out’)

注意当我们为`userId`提供默认值时，我们移除了它的可选注解`?`。我们也不需要再为它指定类型。TypeScript足够智能，可以从默认值推断参数的类型，保持我们的代码简洁易读。

当然，您也可以为默认参数添加显式类型注解，就像对没有默认值的参数一样：

type Context = { appId?: string userId?: string }

function log(message: string, context: Context = {}) { let time = new Date().toISOString() console.log(time, message, context.userId) }

您会发现自己经常使用默认参数而不是可选参数。

剩余参数

如果一个函数接受一个参数列表，你当然可以简单地将列表作为数组传入：

```ts
function sum(numbers: number[]): number {
  return numbers.reduce((total, n) => total + n, 0)
}

sum([1, 2, 3]) // 计算结果为 6
```

有时候，你可能会选择*可变参数*函数API——接受可变数量参数的函数——而不是*固定参数*API，后者接受固定数量的参数。传统上，这需要使用JavaScript的魔法`arguments`对象。

`arguments`之所以是”魔法”，是因为你的JavaScript运行时会自动在函数中定义它，并将你传递给函数的参数列表分配给它。因为`arguments`只是类数组，而不是真正的数组，所以你必须先将其转换为数组，然后才能调用内置的`.reduce`方法：

```
function sumVariadic(): number {
  return Array
    .from(arguments)
    .reduce((total, n) => total + n, 0)
}

sumVariadic(1, 2, 3) // 计算结果为 6
```

但是使用 `arguments` 有一个大问题：它完全不安全！如果你在文本编辑器中将鼠标悬停在 `total` 或 `n` 上，你会看到类似图4-1所示的输出。

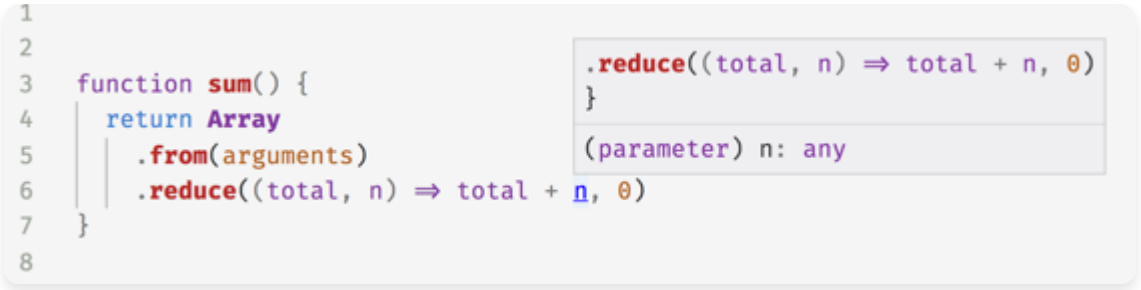


图4-1. arguments是不安全的

这意味着TypeScript推断出 `n` 和 `total` 都是 `any` 类型，并静默地让其通过——直到你尝试使用 `sumVariadic`：

```
sumVariadic(1, 2, 3) // Error TS2554: Expected 0 arguments, but got 3.
```

因为我们没有声明 `sumVariadic` 接受参数，从TypeScript的角度来看它不接受任何参数，所以当我们尝试使用它时会得到 `TypeError`。

那么，我们如何安全地为可变参数函数添加类型？

剩余参数来拯救！我们可以使用剩余参数(rest parameters)来安全地让我们的 `sum` 函数接受任意数量的参数，而不是求助于不安全的 `arguments` 魔法变量：

```
function sumVariadicSafe(...numbers: number[]): number {
  return numbers.reduce((total, n) => total + n, 0)
}
```

```
sumVariadicSafe(1, 2, 3) // 计算结果为 6
```

就是这样！注意这个可变参数`sum`与我们原始的单参数`sum`函数之间唯一的变化就是参数列表中额外的`...`——其他什么都不需要改变，而且它是完全类型安全的。

一个函数最多只能有一个剩余参数，并且该参数必须是函数参数列表中的最后一个。例如，看看TypeScript内置的`console.log`声明（如果你不知道什么是`interface`，不要担心——我们将在第5章中介绍）。`console.log`接受一个可选的`message`，以及任意数量的要记录的附加参数：

```
interface Console {
  log(message?: any, ...optionalParams: any[]): void
}
```

call、apply和bind

除了使用括号`()`调用函数外，JavaScript还支持至少两种其他调用函数的方式。以本章前面的`add`为例：

```
function add(a: number, b: number): number {
  return a + b
}

add(10, 20)                // 计算结果为 30
add.apply(null, [10, 20])  // 计算结果为 30
add.call(null, 10, 20)     // 计算结果为 30
add.bind(null, 10, 20)()   // 计算结果为 30
```

`apply`在你的函数中将一个值绑定到`this`（在这个例子中，我们将`this`绑定到`null`），并将其第二个参数展开到你函数的参数上。`call`做同样的事情，但是按顺序应用其参数而不是展开。

`bind()`是类似的，它将一个`this`参数和参数列表*绑定*到你的函数。区别是`bind`不调用你的函数；相反，它返回一个新函数，你可以用`()`、`.call`或`.apply`调用它，如果你想的话，可以传入更多参数来绑定到迄今为止未绑定的参数。

TSC标志： strictBindCallApply

为了在代码中安全地使用 `.call`、`.apply` 和 `.bind`，请确保在你的 *tsconfig.json* 中启用 `strictBindCallApply` 选项（如果你已经启用了 `strict` 模式，它会自动启用）。

为 this 添加类型

如果你不是从 JavaScript 转过来的，你可能会惊讶地发现在 JavaScript 中，`this` 变量是为每个函数定义的，而不仅仅是那些作为类方法存在的函数。`this` 的值根据你如何调用函数而不同，这使得它出了名的脆弱且难以理解。

提示

出于这个原因，许多团队禁止在除类方法之外的任何地方使用 `this`——要在你的代码库中也这样做，请启用 `no-invalid-this` TSLint 规则。

`this` 脆弱的原因与它的赋值方式有关。一般规则是，当调用方法时，`this` 会取点左边那个东西的值。例如：

```
let x = {
  a() {
    return this
  }
}
x.a() // 在 a() 的主体中，this 是对象 x
```

但是如果你在某个时候在调用之前重新赋值 `a`，结果就会改变！

```
let a = x.a
a() // 现在，在 a() 的主体中，this 是 undefined
```

假设你有一个用于格式化日期的工具函数，看起来像这样：

```
function fancyDate() {
  return ${this.getDate()}/${this.getMonth()}/${this.getFullYear()}
}
```

你在早期作为程序员的日子里设计了这个 API（在你学习函数参数之前）。要使用 `fancyDate`，你必须将一个 `Date` 绑定到 `this` 上来调用它：

```
fancyDate.call(new Date) // 计算结果为 "4/14/2005"
```

如果你忘记将 `Date` 绑定到 `this`，你会得到一个运行时异常！

```
fancyDate() // Uncaught TypeError: this.getDate is not a function
```

虽然探索 `this` 的所有语义超出了本书的范围，但这种行为——`this` 取决于你调用函数的方式，而不是你声明它的方式——至少可以说是令人惊讶的。

幸运的是，TypeScript 支持你。如果你的函数使用 `this`，请确保将你期望的 `this` 类型声明为函数的第一个参数（在任何附加参数之前），TypeScript 会在每个调用点强制 `this` 确实是你所说的那样。`this` 不像其他参数那样处理——当用作函数签名的一部分时，它是一个保留字：

```
function fancyDate(this: Date) {
  return ${this.getDate()}/${this.getMonth()}/${this.getFullYear()}
}
```

现在看看当我们调用 `fancyDate` 时会发生什么：

```
fancyDate.call(new Date) // 计算结果为 "6/13/2008"

fancyDate() // Error TS2684: The 'this' context of type 'void' is
            // not assignable to method's 'this' of type 'Date'.
```

我们将一个运行时错误转换为编译时错误，给了 TypeScript 足够的信息来在编译时而不是运行时警告错误。

TSC 标志：noImplicitThis

要强制在函数中始终显式注释 `this` 类型，请在你的 *tsconfig.json* 中启用 `noImplicitThis` 设置。`strict` 模式包含 `noImplicitThis`，所以如果你已经启用了它，你就可以了。

请注意，`noImplicitThis` 不会为类或对象上的函数强制执行 `this` 注释。

生成器函数

生成器函数（简称*生成器*）是一种便捷的方式来*生成*一堆值。它们让生成器的消费者能够精细控制值的产生速度。因为它们是惰性的——也就是说，它们只在消费者请求时才计算下一个值——它们可以做一些否则难以做到的事情，比如生成无限列表。

它们的工作方式是这样的：

```
function* createFibonacciGenerator() {
  let a = 0
  let b = 1
  while (true) {
    yield a;
    [a, b] = [b, a + b]
  }
}

let fibonacciGenerator = createFibonacciGenerator() //
IterableIterator<number>
fibonacciGenerator.next()   // 计算结果为 {value: 0, done: false}
fibonacciGenerator.next()   // 计算结果为 {value: 1, done: false}
fibonacciGenerator.next()   // 计算结果为 {value: 1, done: false}
fibonacciGenerator.next()   // 计算结果为 {value: 2, done: false}
fibonacciGenerator.next()   // 计算结果为 {value: 3, done: false}
fibonacciGenerator.next()   // 计算结果为 {value: 5, done: false}
```

❶

函数名前的星号(`*`)使该函数成为生成器(generator)。调用生成器返回一个可迭代的迭代器(iterable iterator)。

❷

我们的生成器可以永远生成值。

❸

生成器使用`yield`关键字来*产出*值。当消费者请求生成器的下一个值时(例如，通过调用`next`)，`yield`将结果发送回消费者并暂停执行，直到消费者请求下一个值。这样`while(true)`循环不会立即导致程序永远运行并崩溃。

❹

为了计算下一个斐波那契数，我们在单个步骤中将`a`重新赋值为`b`，将`b`重新赋值为`a + b`。
我们调用了`createFibonacciGenerator`，它返回了一个`IterableIterator`。每次我们调用`next`时，迭代器计算下一个斐波那契数并将其`yield`回给我们。注意TypeScript如何能够从我们`yield`的值的类型推断出迭代器的类型。

你也可以显式注解生成器，将它产出的类型包装在`IterableIterator`中：

```
function* createNumbers(): IterableIterator<number> {
  let n = 0
  while (1) {
    yield n++
  }
}

let numbers = createNumbers()
numbers.next()              // 计算结果为 {value: 0, done: false}
numbers.next()              // 计算结果为 {value: 1, done: false}
numbers.next()              // 计算结果为 {value: 2, done: false}
```

我们不会在本书中深入探讨生成器——它们是一个很大的主题，由于本书是关于TypeScript的，我不想被JavaScript特性分散注意力。简而言之，它们是TypeScript也支持的一个超级酷的JavaScript语言特性。要了解更多关于生成器的信息，请访问MDN上的页面。

迭代器

迭代器是生成器的另一面：生成器是产生值流的方式，而迭代器是消费这些值的方式。术语可能会变得相当混乱，所以让我们从几个定义开始。

可迭代对象(Iterable)

任何包含名为`Symbol.iterator`属性的对象，其值是返回迭代器的函数。

迭代器(Iterator)

任何定义名为`next`方法的对象，该方法返回具有`value`和`done`属性的对象。

当你创建生成器时(比如，通过调用`createFibonacciGenerator`)，你会得到一个*既是*可迭代对象*又是*迭代器的值——一个*可迭代迭代器*——因为它既定义了`Symbol.iterator`属性又定义了`next`方法。

你可以通过创建实现`Symbol.iterator`或`next`的对象(或类)来手动定义迭代器或可迭代对象。例如，让我们定义一个返回数字1到10的迭代器：

```
let numbers = {
  *[Symbol.iterator]() {
    for (let n = 1; n <= 10; n++) {
      yield n
    }
  }
}
```

如果你在代码编辑器中输入该迭代器并将鼠标悬停在其上，你会看到TypeScript推断出的类型([图4-2])。

图4-2. 手动定义迭代器

换句话说，`numbers`是一个迭代器，调用生成器函数`numbers[Symbol.iterator]()`返回一个可迭代迭代器。

你不仅可以定义自己的迭代器，还可以使用JavaScript内置的迭代器来处理常见的集合类型——`Array`、`Map`、`Set`、`String`等——来做如下事情：

```
// 使用for-of遍历迭代器
for (let a of numbers) {
  // 1, 2, 3, 等等
}

// 展开迭代器
let allNumbers = [...numbers] // number[]

// 解构迭代器
let [one, two, ...rest] = numbers // [number, number, number[]]
```

同样，我们不会在本书中更深入地讨论迭代器。你可以在MDN上阅读更多关于迭代器和异步迭代器的信息。

TSC标志： downlevelIteration

如果你要将TypeScript编译为早于`ES2015`的JavaScript版本，可以在*tsconfig.json*中使用[`downlevelIteration`]标志启用自定义迭代器。

如果您的应用程序对包大小特别敏感，您可能希望保持`downlevelIteration`禁用状态：在旧环境中使自定义迭代器工作需要大量代码。例如，前面的`numbers`示例生成了接近1 KB的压缩代码。

调用签名

到目前为止，我们已经学会了为函数的参数和返回类型进行类型标注。现在，让我们转换思路，讨论如何表达函数本身的完整类型。

让我们重新审视本章开头的 `sum` 函数。提醒一下，它看起来是这样的：

```
function sum(a: number, b: number): number {
  return a + b
}
```

`sum` 的类型是什么？好吧，`sum` 是一个函数，所以它的类型是：

```
Function
```

正如您可能猜到的那样，`Function` 类型并不是您在大多数时候想要使用的。就像 `object` 描述所有对象一样，`Function` 是所有函数的通用类型，并不能告诉您它所标注的特定函数的任何信息。

我们还能如何为 `sum` 标注类型？`sum` 是一个接受两个 `number` 类型参数并返回一个 `number` 的函数。在 TypeScript 中，我们可以将其类型表达为：

```
(a: number, b: number) => number
```

这是 TypeScript 表示函数类型或*调用签名*（也称为*类型签名*）的语法。您会注意到它与箭头函数非常相似——这是有意为之！当您将函数作为参数传递或从其他函数返回时，这就是您将使用的语法来为它们标注类型。

注意

参数名称 `a` 和 `b` 只是作为文档，不会影响具有该类型的函数的可分配性。

函数调用签名只包含*类型级别*代码——也就是说，只有类型，没有值。这意味着函数调用签名可以表达参数类型、`this` 类型（参见[“typing this”]）、返回类型、剩余类型和可选类型，但它们不能表达默认值（因为默认值是一个值，而不是类型）。而且由于它们没有供 TypeScript 推断的函数体，调用签名需要显式的返回类型标注。

类型级别和值级别代码

人们在谈论静态类型编程时经常使用”类型级别”和”值级别”这些术语，有一个共同的词汇表会很有帮助。

在本书中，当我使用*类型级别代码*这个术语时，我指的是专门由类型和类型操作符组成的代码。与之对比的是*值级别代码*，即其他所有内容。一个经验法则是：如果它是有效的 JavaScript 代码，那么它就是值级别的；如果它是有效的 TypeScript 但不是有效的 JavaScript，那么它就是类型级别的。

为了确保我们理解一致，让我们看一个例子——这里的粗体术语是类型级别的，其他所有内容都是值级别的：

```
function area(radius: **number**): **number | null** {
  if (radius < 0) {
    return null
  }
  return Math.PI * (radius ** 2)
}

let r: **number** = 3
let a = area(r)
if (a !== null) {
  console.info('result:', a)
}
```

粗体的类型级别术语是类型标注和联合类型操作符 `|`；其他所有内容都是值级别术语。

让我们回顾本章中到目前为止见过的几个函数示例，并将它们的类型提取为绑定到类型别名的独立调用签名：

```
// function greet(name: string)
type Greet = (name: string) => string

// function log(message: string, userId?: string)
type Log = (message: string, userId?: string) => void

// function sumVariadicSafe(...numbers: number[]): number
type SumVariadicSafe = (...numbers: number[]) => number
```

理解了吗？函数的调用签名看起来与它们的实现非常相似。这是有意为之的，是一个语言设计选择，使调用签名更容易推理。

让我们使调用签名与其实现之间的关系更加具体。如果您有一个调用签名，如何声明实现该签名的函数？您只需将调用签名与实现它的函数表达式结合起来。例如，让我们重写 `Log` 以使用其闪亮的新签名：

```
type Log = (message: string, userId?: string) => void

let log: Log = (
  message,
  userId = 'Not signed in'
) => {
  let time = new Date().toISOString()
  console.log(time, message, userId)
}
```

❶

我们声明一个函数表达式 `log`，并显式地将其类型标注为 `Log` 类型。

❷

我们不需要两次标注参数。由于 `message` 作为 `Log` 定义的一部分已经被标注为 `string`，我们不需要在这里再次为其标注类型。相反，我们让 TypeScript 从 `Log` 为我们推断它。

❸

我们为 `userId` 添加一个默认值，因为我们在 `Log` 的签名中捕获了 `userId` 的类型，但我们无法将默认值作为 `Log` 的一部分捕获，因为 `Log` 是一个类型，不能包含值。

❹

我们不需要再次标注返回类型，因为我们已经在 `Log` 类型中将其声明为 `void`。

上下文类型

请注意，最后一个示例是我们见到的第一个不需要显式标注函数参数类型的示例。因为我们已经声明了 `log` 是 `Log` 类型，TypeScript 能够从上下文推断出 `message` 必须是 `string` 类型。这是 TypeScript 类型推断的一个强大功能，称为*上下文类型*。

在本章前面，我们提到了上下文类型出现的另一个地方：回调函数。[5]

让我们声明一个函数 `times`，它调用其回调 `f` 若干次 `n`，每次将当前索引传递给 `f`：

```
function times(
  f: (index: number) => void,
  n: number
) {
  for (let i = 0; i < n; i++) {
    f(i)
  }
}
```

当你调用 `times` 时，如果你内联声明传递给 `times` 的函数，就不必显式标注该函数：

```
times(n => console.log(n), 4)
```

TypeScript 从上下文推断出 `n` 是一个 `number`—我们在 `times` 的签名中声明了 `f` 的参数 `index` 是 `number`，TypeScript 足够聪明地推断出 `n` 就是那个参数，所以它必须是 `number`。

请注意，如果我们没有内联声明 `f`，TypeScript 就无法推断其类型：

```
function f(n) { // Error TS7006: Parameter 'n' implicitly has an 'any'
type.
  console.log(n)
}
```

```
times(f, 4)
```

重载函数类型

我们在上一节中使用的函数类型语法—`type Fn = (...) => ...`—是一个*简写调用签名*。我们可以更明确地完整写出来。再次以 `Log` 为例：

```
// 简写调用签名
type Log = (message: string, userId?: string) => void

// 完整调用签名
type Log = {
  (message: string, userId?: string): void
}
```

两者在各个方面完全等价，仅在语法上有所不同。

你是否会想要使用完整调用签名而不是简写？对于像我们的 `Log` 函数这样的简单情况，你应该优先使用简写；但对于更复杂的函数，完整签名有几个不错的使用案例。

其中第一个就是*重载*函数类型。但首先，重载函数到底意味着什么？

重载函数

具有多个调用签名的函数。

在大多数编程语言中，一旦你声明了一个接受某些参数并产生某种返回类型的函数，你就只能用确切的那些参数调用该函数，并且总是得到相同的返回类型。JavaScript 中并非如此。因为 JavaScript 是如此动态的语言，一个给定函数有多种调用方式是常见的模式；不仅如此，有时输出类型实际上会依赖于参数的输入类型！

TypeScript 用其静态类型系统来建模这种动态性—重载函数声明，以及函数的输出类型依赖于其输入类型。我们可能认为这个语言特性是理所当然的，但对于类型系统来说，这确实是一个非常高级的特性！

你可以使用重载函数签名来设计真正富有表现力的 API。例如，让我们设计一个预订假期的 API—我们称之为 `Reserve`。让我们先勾勒出其类型（这次使用完整类型签名）：

```
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
}
```

然后让我们为 `Reserve` 编写一个实现的存根：

```
let reserve: Reserve = (from, to, destination) => {
  // ...
}
```

所以想要预订巴厘岛旅行的用户必须使用 `from` 日期、`to` 日期和 `"Bali"` 作为目的地来调用我们的 `reserve` API。

我们可能会重新调整我们的 API 来支持单程旅行：

```
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
  (from: Date, destination: string): Reservation
}
```

你会注意到，当你尝试运行这段代码时，TypeScript 会在你实现 `Reserve` 的地方给出错误（参见 [图 4-3]）。

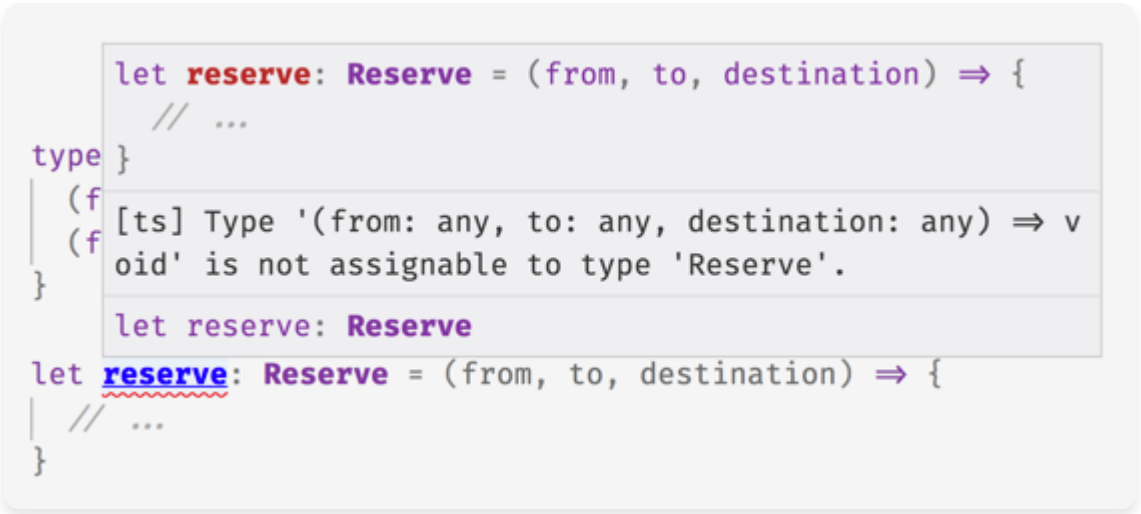


图 4-3. 缺少组合重载签名时的 TypeError

这是因为 TypeScript 中调用签名重载的工作方式。如果你为函数 `f` 声明了一组重载签名，从调用者的角度来看，`f` 的类型是这些重载签名的联合。但从 `f` 的*实现*角度来看，需要有一个可以实际实现的单一组合类型。你需要在实现 `f` 时手动声明这个组合调用签名——它不会为你自动推断。对于我们的 `Reserve` 示例，我们可以这样更新 `reserve` 函数：

```
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
  (from: Date, destination: string): Reservation
}

let reserve: Reserve = (
  from: Date,
  toOrDestination: Date | string,
  destination?: string
) => {
  // ...
}
```

❶

我们声明两个重载函数签名。

❷

实现的签名是我们手动组合两个重载签名的结果（换句话说，我们手动计算了 `Signature1 | Signature2`）。注意，组合签名对调用 `reserve` 的函数不可见；从消费者的角度来看，`Reserve` 的签名是：

```
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
  (from: Date, destination: string): Reservation
}
```

值得注意的是，这不包括我们创建的组合签名：

```
// 错误!
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
  (from: Date, destination: string): Reservation
  (from: Date, toOrDestination: Date | string,
    destination?: string): Reservation
}
```

由于 `reserve` 可能以两种方式被调用，当你实现 `reserve` 时，你必须向 TypeScript 证明你检查了它是如何被调用的：[6]

```
let reserve: Reserve = (
  from: Date,
  toOrDestination: Date | string,
  destination?: string
) => {
  if (toOrDestination instanceof Date && destination !== undefined) {
    // 预订单程旅行
  } else if (typeof toOrDestination === 'string') {
    // 预订往返旅行
  }
}
```

保持重载签名具体化

一般来说，在声明重载函数类型时，每个重载签名（例如 `Reserve`）都必须可赋值给实现的签名（例如 `reserve`）。这意味着在声明实现签名时你可以过于宽泛，只要你的所有重载都可以赋值给它。例如，这样是可行的：

```
let reserve: Reserve = (
  from: any,
  toOrDestination: any,
  destination?: any
) => {
  // ...
}
```

使用重载时，尽量保持实现签名尽可能具体，以便更容易实现函数。这意味着在我们的示例中，更偏向使用 `Date` 而不是 `any`，以及使用 `Date | string` 的联合类型而不是 `any`。

为什么保持类型窄化使实现给定签名的函数更容易？如果你将参数类型设为 `any` 并想将其用作 `Date`，你必须向 TypeScript 证明它实际上是一个日期：

```
function getMonth(date: any): number | undefined {
  if (date instanceof Date) {
    return date.getMonth()
  }
}
```

但如果你预先将参数类型设为 `Date`，则无需在实现中做额外工作：

```
function getMonth(date: Date): number {
  return date.getMonth()
}
```

重载在浏览器 DOM API 中自然出现。`createElement` DOM API 例如用于创建新的 HTML 元素。它接受一个对应 HTML 标签的字符串，并返回该标签类型的新 HTML 元素。TypeScript 为每个 HTML 元素提供内置类型。这些包括：

- `HTMLAnchorElement` 用于 `<a>` 元素
- `HTMLCanvasElement` 用于 `<canvas>` 元素
- `HTMLTableElement` 用于 `<table>` 元素

重载调用签名是建模 `createElement` 工作方式的自然方法。想想你如何为 `createElement` 设定类型（在继续阅读之前尝试自己回答！）。

答案：

```
type CreateElement = {
  (tag: 'a'): HTMLAnchorElement
```

```
  (tag: 'canvas'): HTMLCanvasElement
  (tag: 'table'): HTMLTableElement
  (tag: string): HTMLElement
}

let createElement: CreateElement = (tag: string): HTMLElement => {
  // ...
}
```

❶

我们对参数类型进行重载，使用字符串字面量类型进行匹配。

❷

我们添加一个通用情况：如果用户传递自定义标签名，或尚未纳入 TypeScript 内置类型声明的前沿实验性标签名，我们返回通用的 `HTMLElement`。由于 TypeScript 按声明顺序解析重载，[7] 当你调用 `createElement` 时使用没有特定重载定义的字符串（例如 `createElement('foo')`），TypeScript 将回退到 `HTMLElement`。

❸

要输入实现的参数类型，我们将该参数在 `createElement` 重载签名中可能具有的所有类型组合起来，结果是 `'a' | 'canvas' | 'table' | string`。由于这三个字符串字面量类型都是 `string` 的子类型，所以类型简化为只是 `string`。

注意

在本节的所有示例中，我们都重载了函数表达式。但是如果我们想重载函数声明呢？TypeScript 一如既往地为你考虑周全，为函数声明提供了等效的语法。让我们重写 `createElement` 重载：

```
function createElement(tag: 'a'): HTMLAnchorElement
function createElement(tag: 'canvas'): HTMLCanvasElement
function createElement(tag: 'table'): HTMLTableElement
function createElement(tag: string): HTMLElement {
```

```
  // ...
}
```

使用哪种语法取决于你，并且取决于你正在重载的函数类型（函数表达式或函数声明）。

完整类型签名不仅限于重载如何调用函数。你还可以使用它们来对函数上的属性进行建模。由于 JavaScript 函数只是可调用对象，你可以为它们分配属性来执行以下操作：

```
function warnUser(warning) {
  if (warnUser.wasCalled) {
    return
  }
  warnUser.wasCalled = true
  alert(warning)
}
warnUser.wasCalled = false
```

也就是说，我们向用户显示警告，并且我们不会多次显示警告。让我们使用 TypeScript 来输入 `warnUser` 的完整签名：

```
type WarnUser = {
  (warning: string): void
  wasCalled: boolean
}
```

然后我们可以将 `warnUser` 重写为实现该签名的函数表达式：

```
let warnUser: WarnUser = (warning: string) => {
  if (warnUser.wasCalled) {
    return
  }
  warnUser.wasCalled = true
  alert(warning)
}
warnUser.wasCalled = false
```

注意 TypeScript 足够智能，能够意识到虽然我们在声明 `warnUser` 函数时没有将 [`wasCalled`] 分配给 `warnUser`，但我们在之后立即将 [`wasCalled`] 分配给了它。

多态性

到目前为止，在本书中，我们一直在讨论具体类型以及使用具体类型的函数的方法和原因。什么是*具体类型*？事实证明，到目前为止我们看到的每种类型都是具体的：

- `boolean`
- `string`
- `Date[]`
- `{a: number} | {b: string}`
- `(numbers: number[]) => number`

当你确切知道期望什么类型，并且想要验证实际传递了该类型时，具体类型很有用。但有时，你事先不知道期望什么类型，并且你不想将函数的行为限制为特定类型！

作为我的意思的示例，让我们实现 `filter`。你使用 `filter` 来迭代数组并对其进行筛选；在 JavaScript 中，它可能如下所示：

```
function filter(array, f) {
  let result = []
  for (let i = 0; i < array.length; i++) {
    let item = array[i]
    if (f(item)) {
      result.push(item)
    }
  }
  return result
}

filter([1, 2, 3, 4], _ => _ < 3) // 计算结果为 [1, 2]
```

让我们首先提取 `filter` 的完整类型签名，并为类型添加一些占位符 `unknown`：

```
type Filter = {
  (array: unknown, f: unknown) => unknown[]
}
```

现在，让我们尝试用 `number` 来填充类型：

```
type Filter = {
  (array: number[], f: (item: number) => boolean): number[]
}
```

将数组元素类型化为 `number` 在此示例中效果很好，但 `filter` 旨在成为一个通用函数——你可以过滤数字数组、字符串数组、对象数组、其他数组，任何东西。我们编写的签名适用于数字数组，但它不适用于其他类型元素的数组。让我们尝试使用重载来扩展它以也适用于字符串数组：

```
type Filter = {
  (array: number[], f: (item: number) => boolean): number[]
  (array: string[], f: (item: string) => boolean): string[]
}
```

到目前为止一切都很好（尽管为每种类型写出重载可能会变得混乱）。对象数组呢？

```
type Filter = {
  (array: number[], f: (item: number) => boolean): number[]
  (array: string[], f: (item: string) => boolean): string[]
  (array: object[], f: (item: object) => boolean): object[]
}
```

这在初看时可能看起来不错，但让我们尝试使用它来看看它在哪里出现问题。如果你使用该签名实现一个 `filter` 函数（即 `filter: Filter`），并尝试使用它，你会得到：

```
let names = [
  {firstName: 'beth'},
```

```
  {firstName: 'caitlyn'},
  {firstName: 'xin'}
]

let result = filter(

names,
  _ => _.firstName.startsWith('b')
) // Error TS2339: Property 'firstName' does not exist on type
'object'.

result[0].firstName // Error TS2339: Property 'firstName' does not
exist
                    // on type 'object'.
```

此时，TypeScript 抛出这个错误就很容易理解了。我们告诉 TypeScript 我们可能会向 `filter` 传递一个数字、字符串或对象的数组。我们传递了一个对象数组，但请记住 `object` 并不会告诉你对象的具体形状。所以每次我们尝试访问数组中对象的属性时，TypeScript 都会抛出错误，因为我们没有告诉它对象具有什么特定的形状。

该怎么办呢?

如果你来自支持泛型类型的语言，那么现在你肯定翻白眼并大声喊道：“这就是泛型的用途！”好消息是，你说得很对（坏消息是，你刚刚把邻居家的孩子给喊醒了）。

如果你之前没有使用过泛型类型，我将首先定义它们，然后用我们的 `filter` 函数给出一个例子。

泛型类型参数(Generic type parameter)

一个占位符类型，用于在多个地方强制执行类型级约束。也被称为*多态类型参数(polymorphic type parameter)*。

回到我们的 `filter` 例子，当我们用泛型类型参数 `T` 重写它时，其类型如下所示：

```
type Filter = {
  <T>(array: T[], f: (item: T) => boolean): T[]
```

```
}
```

我们在这里所做的是说：“这个函数 `filter` 使用一个泛型类型参数 `T`；我们不知道这个类型提前会是什么，所以 TypeScript 如果你能在我们每次调用 `filter` 时推断出它是什么，那就太好了。” TypeScript 从我们为 `array` 传递的类型推断出 `T`。一旦 TypeScript 推断出 `T` 对于给定的 `filter` 调用是什么，它就会在看到的每个 `T` 的地方替换该类型。`T` 就像一个占位符类型，由类型检查器根据上下文填充；它*参数化* `Filter` 的类型，这就是为什么我们称它为泛型类型*参数*。

注意

因为每次说”泛型类型参数”都很啰嗦，人们通常简化为”泛型类型”或简单地称为”泛型(generic)“。在本书中我会交替使用这些术语。

看起来很有趣的角括号 `<>` 是你声明泛型类型参数的方式（把它们想象成 `type` 关键字，但用于泛型类型）；你放置角括号的位置决定了泛型的作用域（只有几个地方可以放置它们），TypeScript 确保在它们的作用域内，泛型类型参数的所有实例最终都绑定到相同的具体类型。由于角括号在这个例子中的位置，TypeScript 会在我们调用 `filter` 时将具体类型绑定到我们的泛型 `T`。它会根据我们调用 `filter` 时传递的内容来决定将哪个具体类型绑定到 `T`。你可以在一对角括号之间声明任意多个逗号分隔的泛型类型参数。

注意

`T` 只是一个类型名称，我们可以使用任何其他名称：`A`、`Zebra` 或 `l33t`。按照惯例，人们使用从字母 `T` 开始的大写单字母名称，然后继续到 `U`、`V`、`W` 等，具体取决于他们需要多少个泛型。

如果你连续声明很多泛型或以复杂的方式使用它们，考虑偏离这个惯例，使用更具描述性的名称，如 `Value` 或 `WidgetType`。

有些人喜欢从 `A` 而不是 `T` 开始。不同的编程语言社区更喜欢其中一种，这取决于他们的传统：函数式语言用户更喜欢 `A`、`B`、`C` 等，因为它们类似于你在数学证明中可能找到的希腊字母 α、β 和 γ；面向对象语言用户倾向于使用 `T` 表示”Type”。TypeScript 虽然支持两种编程风格，但使用后一种惯例。

就像函数的参数在每次调用该函数时重新绑定一样，每次调用 `filter` 都会为 `T` 获得自己的绑定：

```
type Filter = {
  <T>(array: T[], f: (item: T) => boolean): T[]
}

let filter: Filter = (array, f) => // ...

// (a) T 绑定到 number
filter([1, 2, 3], _ => _ > 2)

// (b) T 绑定到 string
filter(['a', 'b'], _ => _ !== 'b')

// (c) T 绑定到 {firstName: string}
let names = [
  {firstName: 'beth'},
  {firstName: 'caitlyn'},
  {firstName: 'xin'}
]
filter(names, _ => _.firstName.startsWith('b'))
```

TypeScript 从我们传递的参数类型推断这些泛型绑定。让我们逐步了解 TypeScript 如何为 (a) 绑定 `T`：

1. 从 `filter` 的类型签名，TypeScript 知道 `array` 是某种类型 `T` 的元素数组。
2. TypeScript 注意到我们传入了数组 `[1, 2, 3]`，所以 `T` 必须是 `number`。
3. 每当 TypeScript 看到 `T` 时，它会将 `number` 类型替换进去。所以参数 `f: (item: T) => boolean` 变成 `f: (item: number) => boolean`，返回类型 `T[]` 变成 `number[]`。
4. TypeScript 检查所有类型是否满足可赋值性，以及我们作为 `f` 传入的函数是否可赋值给其新推断的签名。

泛型(Generics)是一种强大的方式，可以用比具体类型更通用的方式来表达你的函数做什么。理解泛型的方式就是将它们视为*约束*。就像将函数参数注解为 `n: number` 约束参数 `n` 的

*值*为 `number` 类型一样，使用泛型 `T` 约束你绑定到 `T` 的任何*类型*在 `T` 出现的任何地方都是相同的类型。

提示

泛型类型也可以用在类型别名、类和接口中——我们将在本书中大量使用它们。我会在涵盖更多主题时在上下文中介绍它们。

尽可能使用泛型。它们将帮助保持你的代码通用、可重用和简洁。

泛型何时绑定?

你声明泛型类型的位置不仅决定了类型的作用域，还决定了 TypeScript 何时将具体类型绑定到你的泛型上。从上一个[示例]:

```
type Filter = {
  <T>(array: T[], f: (item: T) => boolean): T[]
}

let filter: Filter = (array, f) =>
  // ...
```

因为我们将 `<T>` 声明为调用签名的一部分（就在签名的开始括号 `()` 之前），TypeScript 会在我们实际调用 `Filter` 类型的函数时将具体类型绑定到 `T`。

如果我们将 `T` 的作用域设为类型别名 `Filter`，TypeScript 会要求我们在使用 `Filter` 时显式绑定一个类型:

```
type Filter<T> = {
  (array: T[], f: (item: T) => boolean): T[]
}

let filter: Filter = (array, f) => // Error TS2314: Generic type
'Filter'
  // ...                           // requires 1 type argument(s).

type OtherFilter = Filter          // Error TS2314: Generic type
'Filter'
                                   // requires 1 type argument(s).

let filter: Filter<number> = (array, f) =>
  // ...

type StringFilter = Filter<string>
```

```
let stringFilter: StringFilter = (array, f) =>
  // ...
```

通常，TypeScript 会在你使用泛型时将具体类型绑定到你的泛型：对于函数，是在你调用它们时；对于类，是在你实例化它们时（在[“多态性”]中会详细介绍）；对于类型别名和接口（参见[“接口”]），是在你使用或实现它们时。

你可以在哪里声明泛型?

对于 TypeScript 声明调用签名的每种方式，都有一种向其添加泛型类型的方法：

```
type Filter = {
  <T>(array: T[], f: (item: T) => boolean): T[]
}
let filter: Filter = // ...

type Filter<T> = {
  (array: T[], f: (item: T) => boolean): T[]
}
let filter: Filter<number> = // ...

type Filter = <T>(array: T[], f: (item: T) => boolean) => T[]
let filter: Filter = // ...

type Filter<T> = (array: T[], f: (item: T) => boolean) => T[]
let filter: Filter<string> = // ...

function filter<T>(array: T[], f: (item: T) => boolean): T[] {
  // ...
}
```

❶

完整的调用签名，`T` 作用域限定为单个签名。因为 `T` 的作用域限定为单个签名，TypeScript 会在你调用 `filter` 类型的函数时将此签名中的 `T` 绑定到具体类型。每次对 `filter` 的调用都会获得自己的 `T` 绑定。

❷

完整的调用签名，`T` 作用域限定为*所有*签名。因为 `T` 声明为 `Filter` 类型的一部分（而不是特定签名类型的一部分），TypeScript 会在你声明 `Filter` 类型的函数时绑定 `T`。

❸

类似于 [

❶

]，但使用简写调用签名而非完整签名。

❹

类似于 [

❷

]，但使用简写调用签名而非完整签名。

❺

命名函数调用签名，`T` 作用域限定为该签名。TypeScript 会在你调用 `filter` 时将具体类型绑定到 `T`，每次对 `filter` 的调用都会获得自己的 `T` 绑定。
作为第二个示例，让我们编写一个 `map` 函数。`map` 与 `filter` 非常相似，但不是从数组中移除项目，而是使用映射函数转换每个项目。我们先从勾画实现开始：

```typescript
function map(array: unknown[], f: (item: unknown) => unknown):
unknown[] {
  let result = []
  for (let i = 0; i < array.length; i++) {
    result[i] = f(array[i])
  }
  return result
}
```

在继续之前，请尝试思考如何让 `map` 变为泛型(generic)，将每个 `unknown` 替换为某种类型。你需要多少个泛型？如何声明泛型，并将它们作用域限定在 `map` 函数中？`array`、`f` 和返回值的类型应该是什么？

准备好了吗？如果你没有先自己尝试做一遍，我鼓励你尝试一下。你可以做到的。真的！

好的，不再唠叨了。以下是答案：

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {
  let result = []
  for (let i = 0; i < array.length; i++) {
    result[i] = f(array[i])
  }
  return result
}
```

我们恰好需要两个泛型类型：`T`表示输入数组成员的类型，`U`表示输出数组成员的类型。我们传入一个`T`数组，和一个映射函数，该函数接受一个`T`并将其映射为`U`。最后，我们返回一个`U`数组。

标准库中的filter和map

我们对`filter`和`map`的定义与TypeScript内置的非常相似：

```
interface Array<T> {
  filter(
    callbackfn: (value: T, index: number, array: T[]) => any,
    thisArg?: any
  ): T[]
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U,
    thisArg?: any
  ): U[]
}
```

我们还没有涉及接口(interface)，但这个定义说明`filter`和`map`是类型为`T`的数组上的函数。它们都接受一个函数`callbackfn`，以及函数内部`this`的类型。

`filter`使用作用域为整个`Array`接口的泛型`T`。`map`也使用`T`，并添加了第二个泛型`U`，该泛型仅作用域限定在`map`函数中。这意味着当你创建数组时，TypeScript会将具体类型绑定到`T`，该数组上的每次`filter`和`map`调用都会共享该具体类型。每次调用`map`时，该调用将获得自己的`U`绑定，同时还可以访问已绑定的`T`。

JavaScript标准库中的许多函数都是泛型的，特别是`Array`原型上的那些函数。数组可以包含任何类型的值，所以我们将该类型称为`T`，可以说诸如”`.push`接受类型为`T`的参数”，或”`.map`从`T`数组映射到`U`数组”。

泛型类型推断

在大多数情况下，TypeScript在推断泛型类型方面做得很好。当你调用我们之前编写的`map`函数时，TypeScript推断出`T`是`string`，`U`是`boolean`：

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {
  // ...
}

map(
  ['a', 'b', 'c'],  // T的数组
  _ => _ === 'a'    // 返回U的函数
)
```

但是，你也可以显式注解泛型。泛型的显式注解是全有或全无的；要么注解每个必需的泛型类型，要么一个也不注解：

```
map <string, boolean>(
  ['a', 'b', 'c'],
  _ => _ === 'a'
)

map <string>( // Error TS2558: Expected 2 type arguments, but got 1.
  ['a', 'b', 'c'],
  _ => _ === 'a'
)
```

TypeScript会检查每个推断的泛型类型是否可分配给其对应的显式绑定泛型；如果不可分配，你将收到错误：

```
// 正确，因为boolean可分配给boolean | string
map<string, boolean | string>(
  ['a', 'b', 'c'],
  _ => _ === 'a'
```

```
)

map<string, number>(
  ['a', 'b', 'c'],
  _ => _ === 'a'  // Error TS2322: Type 'boolean' is not assignable
)                 // to type 'number'.
```

由于TypeScript从你传入泛型函数的参数中推断泛型的具体类型，有时你会遇到这样的情况：

```
let promise = new Promise(resolve =>
  resolve(45)
)
promise.then(result => // 推断为{}
  result * 4 // Error TS2362: The left-hand side of an arithmetic
operation must
)            // be of type 'any', 'number', 'bigint', or an enum type.
```

怎么回事？为什么TypeScript推断`result`为`{}`？因为我们没有给它足够的信息来工作——由于TypeScript只使用泛型函数参数的类型来推断泛型的类型，它将`T`默认为`{}`！

要修复这个问题，我们必须显式注解`Promise`的泛型类型参数：

```
let promise = new Promise<number>(resolve =>
  resolve(45)
)
promise.then(result => // number
  result * 4
)
```

泛型类型别名

我们在本章前面的 `Filter` 示例中已经接触过泛型类型别名。如果你还记得上一章中的 `Array` 和 `ReadonlyArray` 类型（见[“只读数组和元组”]），它们也是泛型类型别名！让我们通过一个简短的示例深入探讨在类型别名中使用泛型。

让我们定义一个 `MyEvent` 类型来描述 DOM 事件，如 `click` 或 `mousedown`：

```
type MyEvent<T> = {
  target: T
  type: string
}
```

注意这是在类型别名中声明泛型类型的唯一有效位置：紧跟在类型别名名称之后，在赋值操作符（`=`）之前。

`MyEvent` 的 `target` 属性指向事件发生的元素：`<button />`、`<div />` 等等。例如，你可以这样描述一个按钮事件：

```
type ButtonEvent = MyEvent<HTMLButtonElement>
```

当你使用像 `MyEvent` 这样的泛型类型时，必须在使用该类型时显式绑定其类型参数；它们不会为你自动推断：

```
let myEvent: Event<HTMLButtonElement | null> = {
  target: document.querySelector('#myButton'),
  type: 'click'
}
```

你可以使用 `MyEvent` 来构建另一个类型——比如 `TimedEvent`。当 `TimedEvent` 中的泛型 `T` 被绑定时，TypeScript 也会将其绑定到 `MyEvent`：

```
type TimedEvent<T> = {
  event: MyEvent<T>
  from: Date
  to: Date
}
```

你也可以在函数签名中使用泛型类型别名。当 TypeScript 将类型绑定到 `T` 时，它也会为你将其绑定到 `MyEvent`：

```
function triggerEvent<T>(event: MyEvent<T>): void {
  // ...
}

triggerEvent({ // T 是 Element | null
  target: document.querySelector('#myButton'),
  type: 'mouseover'
})
```

让我们一步步分析这里发生的事情：

1. 我们用一个对象调用 `triggerEvent`。
2. TypeScript 看到根据我们函数的签名，我们传递的参数必须具有 `MyEvent<T>` 类型。它还注意到我们将 `MyEvent<T>` 定义为 `{target: T, type: string}`。
3. TypeScript 注意到我们传递的对象的 `target` 字段是 `document.querySelector('#myButton')`。这意味着 `T` 必须是 `document.querySelector('#myButton')` 的任何类型：`Element | null`。所以 `T` 现在被绑定到 `Element | null`。
4. TypeScript 遍历并将每个 `T` 的出现替换为 `Element | null`。
5. TypeScript 检查我们所有的类型是否满足可赋值性。它们满足，所以我们的代码通过类型检查。

有界多态性

注意

在本节中，我将使用二叉树作为示例。如果你以前没有使用过二叉树，不用担心。对于我们的目的，基础知识是：

- 二叉树是一种数据结构。
- 二叉树由节点组成。
- 节点保存一个值，可以指向最多两个子节点。
- 节点可以是两种类型之一：*叶节点*（leaf node，意味着它没有子节点）或*内部节点*（inner node，意味着它至少有一个子节点）。

有时，仅仅说”这个东西是某种泛型类型 `T`，那个东西必须具有相同的类型 `T`“是不够的。有时你还想说”类型 `U` 应该*至少是* `T`”。我们称之为对 `U` 设置*上界*。

为什么我们想要这样做？假设我们正在实现一个二叉树，并有三种类型的节点：

1. 常规的 `TreeNode`
2. `LeafNode`，它们是没有子节点的 `TreeNode`
3. `InnerNode`，它们是有子节点的 `TreeNode`

让我们首先为我们的节点声明类型：

```
type TreeNode = {
  value: string
}
type LeafNode = TreeNode & {
  isLeaf: true
}
type InnerNode = TreeNode & {
```

```
  children: [TreeNode] | [TreeNode, TreeNode]
}
```

我们所说的是：`TreeNode` 是一个具有单个属性 `value` 的对象。`LeafNode` 类型具有 `TreeNode` 的所有属性，加上一个始终为 `true` 的 `isLeaf` 属性。`InnerNode` 也具有 `TreeNode` 的所有属性，加上一个指向一个或两个子节点的 `children` 属性。

接下来，让我们编写一个 `mapNode` 函数，它接受一个 `TreeNode` 并映射其值，返回一个新的 `TreeNode`。我们想要提出一个可以这样使用的 `mapNode` 函数：

```
let a: TreeNode = {value: 'a'}
let b: LeafNode = {value: 'b', isLeaf: true}
let c: InnerNode = {value: 'c', children: [b]}

let a1 = mapNode(a, _ => _.toUpperCase()) // TreeNode
let b1 = mapNode(b, _ => _.toUpperCase()) // LeafNode
let c1 = mapNode(c, _ => _.toUpperCase()) // InnerNode
```

现在暂停一下，*考虑一下如何编写一个* `mapNode` *函数，它接受* `TreeNode` *的子类型并返回相同的子类型*。传入 `LeafNode` 应该返回 `LeafNode`，`InnerNode` 应该返回 `InnerNode`，`TreeNode` 应该返回 `TreeNode`。在继续之前考虑一下你会如何实现这个功能。这可能吗？

答案如下：

```
function mapNode<T extends TreeNode>(
  node: T,
  f: (value: string) => string
): T {
  return {
    ...node,
    value: f(node.value)
  }
}
```

❶

`mapNode` 是一个定义了单个泛型类型参数 `T` 的函数。 `T` 有一个 `TreeNode` 的上界。也就是说， `T` 可以是 `TreeNode` ，或者是 `TreeNode` 的子类型。

❷

`mapNode` 接受两个参数，第一个是类型为 `T` 的 `node` 。因为在[

❶

]中我们说 `node extends TreeNode` ，如果我们传入不是[`TreeNode`]的东西——比如空对象 `{}` 、 `null` 或 `TreeNode` 数组——那会立即出现红色波浪线。 `node` 必须是 `TreeNode` 或 `TreeNode` 的子类型。

❸

`mapNode` 返回类型为 `T` 的值。记住 `T` 可能是 `TreeNode` ，或者是 `TreeNode` 的任何子类型。
为什么我们必须这样声明 `T` ？

- 如果我们将 `T` 类型化为仅仅是 `T` （省略 `extends TreeNode` ），那么 `mapNode` 会抛出编译时错误，因为你无法安全地在无界的类型 `T` 的 `node` 上读取 `node.value` （如果用户传入一个数字会怎样？）。
- 如果我们完全省略 `T` 并将 `mapNode` 声明为 `(node: TreeNode, f: (value: string) => string) => TreeNode` ，那么在映射节点后我们会丢失信息： `a1` 、 `b1` 和 `c1` 都将只是 `TreeNode` 。

通过说 `T extends TreeNode` ，我们能够保留输入节点的特定类型（ `TreeNode` 、 `LeafNode` 或 `InnerNode` ），即使在映射它之后。

具有多个约束的有界多态性

在上一个例子中，我们对 `T` 施加了单个类型约束： `T` 必须至少是一个[`TreeNode`]。但是如果你想要多个类型约束怎么办？

只需扩展这些约束的交集（ `&` ）：

```
type HasSides = {numberOfSides: number}
type SidesHaveLength = {sideLength: number}

function logPerimeter<
  Shape extends HasSides & SidesHaveLength
>(s: Shape): Shape {
  console.log(s.numberOfSides * s.sideLength)
  return s
}

type Square = HasSides & SidesHaveLength
let square: Square = {numberOfSides: 4, sideLength: 3}
logPerimeter(square) // Square, 记录"12"
```

❶

`logPerimeter`是一个函数，接受类型为`Shape`的单个参数`s`。

❷

`Shape`是一个泛型类型，它扩展了`HasSides`类型和[`SidesHaveLength`]类型。换句话说，`Shape`必须至少有具有长度的边。

❸

`logPerimeter`返回与你给它的完全相同类型的值。

使用有界多态性来建模参数个数

另一个你会发现自己使用有界多态性的地方是建模可变参数函数（接受任意数量参数的函数）。例如，让我们实现JavaScript内置`call`函数的自己版本（提醒一下，`call`是一个接受函数和可变数量参数，并将这些参数应用于函数的函数）。[8]我们将这样定义和使用它，对稍后填入的类型使用`unknown`：

```
function call(
  f: (...args: unknown[]) => unknown,
```

```
  ...args: unknown[]
): unknown {
  return f(...args)
}

function fill(length: number, value: string): string[] {
  return Array.from({length}, () => value)
}

call(fill, 10, 'a') // 求值为10个'a'的数组
```

现在让我们填入`unknown`。我们想要表达的约束是：

- `f`应该是一个接受某组参数`T`并返回某种类型`R`的函数。我们事先不知道它会有多少参数。
- `call`接受`f`，以及`f`本身接受的相同参数集合`T`。同样，我们事先不知道确切期望多少参数。
- `call`返回与`f`返回的相同类型`R`。

我们需要两个类型参数：`T`，它是一个参数数组，和`R`，它是任意的返回值。让我们填入类型：

```
function call<T extends unknown[], R>(
  f: (...args: T) => R,
  ...args: T
): R {
  return f(...args)
}
```

这究竟是如何工作的？让我们逐步浏览：

❶

`call`是一个可变参数函数（提醒一下，可变参数函数是接受任意数量参数的函数），有两个类型参数：`T`和`R`。`T`是`unknown[]`的子类型；也就是说，`T`是任意类型的数组或元组。

❷

`call`的第一个参数是函数`f`。`f`也是可变参数的，其参数与`args`共享类型：无论`args`是什么类型，`f`的参数都具有完全相同的类型。

❸

除了函数`f`之外，`call`还有可变数量的附加参数`...args`。`args`是一个剩余参数——即描述可变数量参数的参数。`args`的类型是`T`，而`T`必须是数组类型（实际上，如果我们忘记声明`T`扩展数组类型，TypeScript会向我们抛出波浪线错误），因此TypeScript将根据我们为`args`传入的具体参数推断出`T`的*元组类型*。

❹

`call`返回类型为`R`的值（`R`绑定到`f`返回的任何类型）。
现在当我们调用`call`时，TypeScript将准确知道返回类型是什么，当我们传递错误数量的参数时它会报错：

```
let a = call(fill, 10, 'a')      // string[]
let b = call(fill, 10)           // Error TS2554: Expected 3
arguments; got 2.
let c = call(fill, 10, 'a', 'z') // Error TS2554: Expected 3
arguments; got 4.
```

我们使用类似的技术来利用TypeScript为剩余参数推断元组类型的方式，以改进元组的类型推断，详见[“改进元组的类型推断”]。

泛型类型默认值

就像您可以为函数参数提供默认值一样，您也可以为泛型类型参数提供默认类型。例如，让我们重新访问[“泛型类型别名”]中的`MyEvent`类型。提醒一下，我们使用该类型来建模DOM事件，它看起来像这样：

```
type MyEvent<T> = {
  target: T
  type: string
}
```

要创建新事件，我们必须显式地将泛型类型绑定到`MyEvent`，表示事件被分发到的HTML元素类型：

```
let buttonEvent: MyEvent<HTMLButtonElement> = {
  target: myButton,
  type: string
}
```

作为当我们事先不知道`MyEvent`将绑定到的具体元素类型时的便利，我们可以为`MyEvent`的泛型添加默认值：

```
type MyEvent<T = HTMLElement> = {
  target: T
  type: string
}
```

我们也可以利用这个机会应用我们在前几节学到的内容，为`T`添加边界，确保`T`是HTML元素：

```
type MyEvent<T extends HTMLElement = HTMLElement> = {
  target: T
  type: string
}
```

现在，我们可以轻松创建不特定于特定HTML元素类型的事件，在创建事件时不必手动将`MyEvent`的`T`绑定到`HTMLElement`：

```
let myEvent: MyEvent = {
  target: myElement,
  type: string
}
```

请注意，就像函数中的可选参数一样，具有默认值的泛型类型必须出现在没有默认值的泛型类型之后：

```
// 好的
type MyEvent2<
  Type extends string,
  Target extends HTMLElement = HTMLElement,
> = {
  target: Target
  type: Type
}

// 不好的
type MyEvent3<
  Target extends HTMLElement = HTMLElement,
  Type extends string  // Error TS2706: Required type parameters may
> = {                  // not follow optional type parameters.
  target: Target
  type: Type
}
```

类型驱动开发

强大的类型系统带来了强大的能力。当您使用TypeScript编写代码时，您会经常发现自己”以类型为导向”。当然，这指的是*类型驱动开发*(type-driven development)。

类型驱动开发

一种编程风格，您首先勾勒出类型签名，然后再填入值。

静态类型系统的要点是约束表达式可以保存的值的类型。类型系统越有表现力，它就越能告诉您该表达式中包含的值。当您将有表现力的类型系统应用于函数时，函数的类型签名最终可能会告诉您关于该函数所需了解的大部分信息。

让我们看看本章前面`map`函数的类型签名：

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {
  // ...
}
```

仅仅看这个签名——即使你之前从未见过`map`——你应该对`map`的作用有一些直觉：它接受一个`T`类型的数组和一个从`T`映射到`U`的函数，返回一个`U`类型的数组。注意，你无需查看函数的实现就能知道这一点！[9]

当你编写 TypeScript 程序时，首先定义函数的类型签名——换句话说，*以类型为先*——稍后再填充实现。通过首先在类型层面勾勒程序，你可以确保在深入实现之前，一切在高层次上都是合理的。

你会注意到，到目前为止，我们一直在做相反的事情：先实现，然后推导类型。现在你已经掌握了在 TypeScript 中编写和类型化函数的技能，我们将切换模式，首先勾勒类型，然后填充细节。

总结

在本章中，我们讨论了如何声明和调用函数、如何为参数设置类型，以及如何在 TypeScript 中表达常见的 JavaScript 函数特性，如默认参数、剩余参数、生成器函数和迭代器。我们讨论了函数的调用签名和实现之间的区别、上下文类型化以及重载函数的不同方式。最后，我们深入介绍了函数的多态性(polymorphism)和类型别名：为什么有用、如何以及在何处声明泛型类型、TypeScript 如何推断泛型类型，以及如何声明和为泛型添加边界和默认值。我们以类型驱动开发的简短说明结束：它是什么，以及如何使用你新获得的函数类型知识来实践它。

练习

1. TypeScript 推断函数类型签名的哪些部分：参数、返回类型，还是两者？

2. JavaScript 的 `arguments` 对象是类型安全的吗？如果不是，你可以用什么来代替？

3. 你希望能够预订立即开始的假期。更新本章前面的重载 `reserve` 函数（[“重载函数类型”]），添加第三个调用签名，该签名只接受目的地，不需要明确的开始日期。更新 `reserve` 的实现以支持这个新的重载签名。

4. [困难] 更新我们本章前面的 `call` 实现（[“使用有界多态性建模参数数量”]），使其仅适用于第二个参数是 `string` 的函数。对于所有其他函数，你的实现应该在编译时失败。

5. 实现一个小型类型安全断言库 `is`。首先勾勒你的类型。完成后，你应该能够这样使用它：

```
// 比较字符串和字符串
is('string', 'otherstring') // false

// 比较布尔值和布尔值
is(true, false) // false

// 比较数字和数字
is(42, 42) // true

// 比较两个不同类型应该给出编译时错误
is(10, 'foo') // Error TS2345: Argument of type '"foo"' is not
assignable
              // to parameter of type 'number'.

// [困难] 我应该能够传递任意数量的参数
is([1], [1, 2], [1, 2, 3]) // false
```

[1] 为什么它们不安全？如果你在代码编辑器中输入最后一个例子，你会看到它的类型是 `Function`。这个 `Function` 类型是什么？它是一个可调用的对象（你知道，在它后面加上 `()`），并且拥有来自 `Function.prototype` 的所有原型方法。但它的参数和返回类型

是无类型的，所以你可以用任何参数调用这个函数，TypeScript 会袖手旁观，看着你做一些在你居住的任何城镇都应该是非法的事情。

[2] 要深入了解 `this`，请查看 Kyle Simpson 的 O'Reilly You Don't Know JS 系列。

[3] 值得注意的是，`Object` 和 `Number` 不是迭代器。

[4] 这个经验法则的例外是枚举和命名空间。枚举既生成类型又生成值，而命名空间只存在于值层面。完整参考请参见[附录 C]。

[5] 如果你以前没有听过"回调"这个术语，它只是你作为参数传递给另一个函数的函数。

[6] 要了解更多，请跳到["细化"]。

[7] 大部分情况下——TypeScript 在按顺序解析重载之前，会将字面量重载提升到非字面量重载之上。不过，你可能不想依赖这个特性，因为对于不熟悉这种行为的其他工程师来说，这会使你的重载难以理解。

[8] 为了简化我们的实现，我们将设计我们的 `call` 函数不考虑 `this`。

[9] 有一些编程语言（如类似 Haskell 的语言 Idris）具有内置的约束求解器，能够从你编写的签名*自动*为你实现函数体！

[第5章.]类和接口

如果你像大多数来自面向对象编程语言的程序员一样，类是你的基础工具。类是你组织和思考代码的方式，它们是你主要的封装单元。你会很高兴地了解到，TypeScript 类大量借鉴了 C#，支持可见性修饰符、属性初始化器、多态性、装饰器和接口等特性。但由于 TypeScript 类编译为常规的 JavaScript 类，你也可以以类型安全的方式表达 JavaScript 习惯用法，如混入(mixins)。

TypeScript 的某些类特性，如属性初始化器和装饰器，也被 JavaScript 类支持[1]，因此会生成运行时代码。其他特性，如可见性修饰符、接口和泛型，是 TypeScript 独有的特性，仅存在于编译时，当您将应用程序编译为 JavaScript 时不会生成任何代码。

在本章中，我将通过一个扩展示例来指导您了解如何在 TypeScript 中使用类，以便您不仅能够获得对 TypeScript 面向对象语言特性的直觉理解，还能了解我们如何以及为什么使用它们。尽量跟着做，在我们进行的过程中在您的代码编辑器中输入代码。

类和继承

我们将构建一个国际象棋引擎。 我们的引擎将模拟一场国际象棋游戏，并为两个玩家轮流移动棋子提供 API。

我们先从勾勒类型开始：

```
// 表示一场国际象棋游戏
class Game {}

// 一个国际象棋棋子
class Piece {}

// 棋子的坐标集合
class Position {}
```

有六种类型的棋子：

```
// ...
class King extends Piece {}
class Queen extends Piece {}
class Bishop extends Piece {}
class Knight extends Piece {}
class Rook extends Piece {}
class Pawn extends Piece {}
```

每个棋子都有颜色和当前位置。在国际象棋中，位置被建模为（字母，数字）坐标对；字母沿 x 轴从左到右运行，数字沿 y 轴从下到上运行（[图 5-1]）。

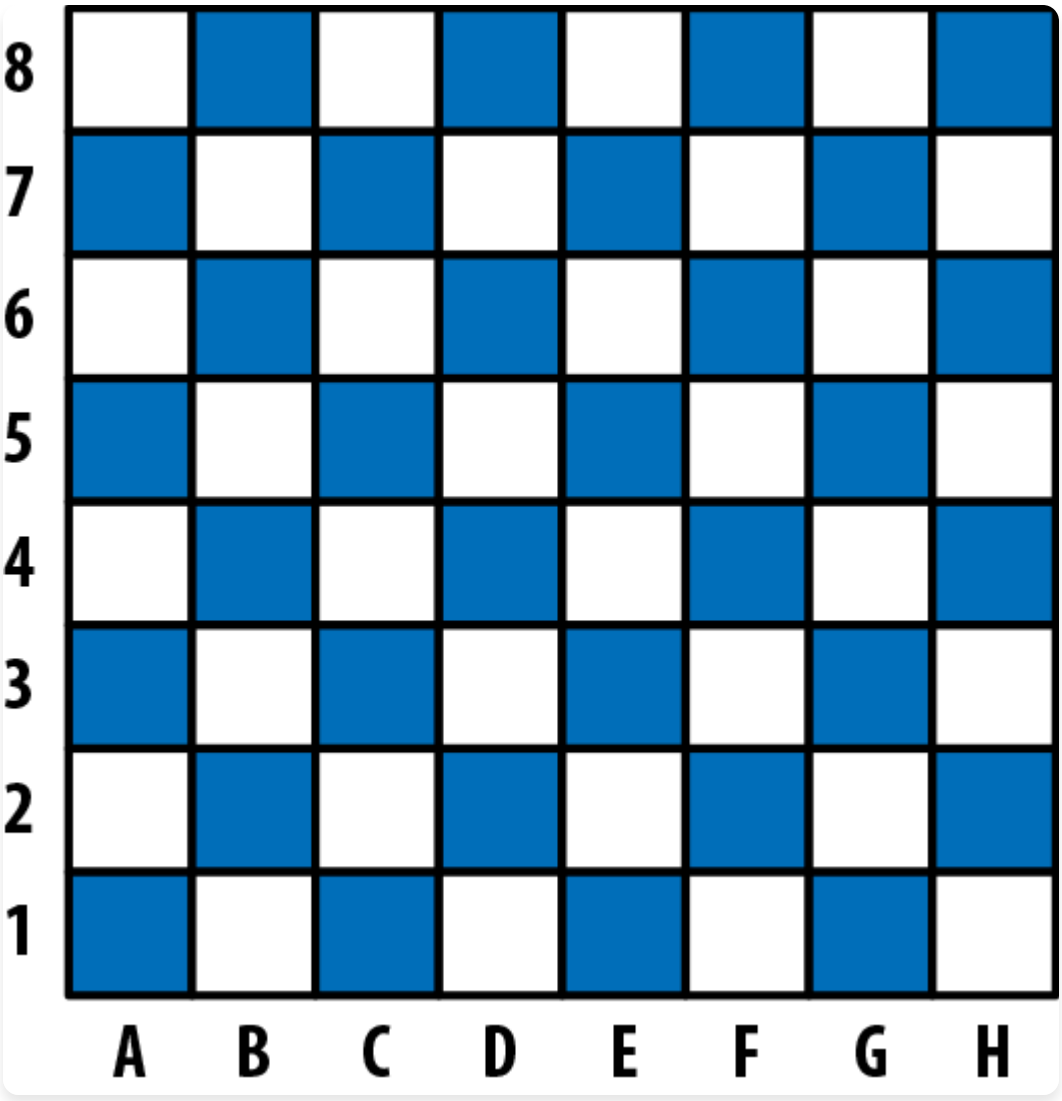


图 5-1. 国际象棋中的标准代数记号法：A-H（x 轴）被称为”列”，1-8（倒置的 y 轴）被称为”行”

让我们将颜色和位置添加到我们的 `Piece` 类：

```
type Color = 'Black' | 'White'
type File = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H'
type Rank = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

class Position {
  constructor(
    private file: File,
    private rank: Rank
```

```
  ) {}
}

class Piece {
  protected position: Position
  constructor(
    private readonly color: Color,
    file: File,
    rank: Rank
  ) {
    this.position = new Position(file, rank)
  }
}
```

❶

由于颜色、行和列相对较少，我们可以手动枚举它们的可能值作为类型字面量。这将让我们通过将这些类型的域从所有字符串和所有数字约束到少数几个非常特定的字符串和数字来挤出一些额外的安全性。

❷

构造函数中的 `private` *访问修饰符*会自动将参数分配给 `this`（`this.file` 等），并将其可见性设置为私有，这意味着 `Piece` 实例内的代码可以读写它，但 `Piece` 实例外的代码不能。`Piece` 的不同实例可以访问彼此的私有成员；任何其他类的实例——甚至是 `Piece` 的子类——都不能。

❸

我们将实例变量 `position` 声明为 `protected`。与 `private` 一样，`protected` 将属性分配给 `this`，但与 `private` 不同，`protected` 使属性对 `Piece` 的实例和 `Piece` 的任何子类的实例都可见。我们在声明时没有给 `position` 赋值，所以我们必须在 `Piece` 的构造函数中给它赋值。如果我们没有在构造函数中赋值，TypeScript 会告诉我们该变量没有*明确赋值*，即，我们说它是 `T` 类型，但实际上它是 `T | undefined`，因为它在属性初始化器或构造函数中没有被赋值——所以我们需要更新它的签名以指示它不一定是 `Position`，也可能是 `undefined`。

❹

`new Piece` 接受三个参数：`color`、`file` 和 `rank`。我们为 `color` 添加了两个修饰符：`private`，意思是将其分配给 `this` 并确保它只能从 `Piece` 的实例访问，以及 `readonly`，意思是在这次初始分配后它只能读取而不能再写入。

TSC 标志： strictNullChecks 和 strictPropertyInitialization

要选择加入类实例变量的明确赋值检查，请在您的 *tsconfig.json* 中启用 `strictNullChecks` 和 `strictPropertyInitialization` 标志。 如果您已经在使用 `strict` 标志，您就可以开始了。

TypeScript 支持三个访问修饰符用于类上的属性和方法：

`public`
可以从任何地方访问。这是默认的访问级别。

`protected`
从此类及其子类的实例中可访问。

`private`
仅从此类的实例中可访问。

使用访问修饰符，你可以设计不暴露太多实现细节的类，而是暴露定义良好的API供他人使用。

我们定义了一个 `Piece` 类，但我们不希望用户直接实例化一个新的 `Piece` ——我们希望他们扩展它来创建 `Queen`、`Bishop` 等，然后实例化*那个*。我们可以使用类型系统通过 `abstract` 关键字来为我们强制执行这一点：

```
// ...
abstract class Piece {
  constructor(
    // ...
```

现在如果你尝试直接实例化一个 `Piece`， TypeScript 会报错：

```
new Piece('White', 'E', 1)  // 错误 TS2511: 无法创建抽象类的实例。
```

`abstract` 关键字意味着你不能直接实例化该类，但这并不意味着你不能在其上定义一些方法：

```
// ...
abstract class Piece {
  // ...
  moveTo(position: Position) {
    this.position = position
  }
  abstract canMoveTo(position: Position): boolean
}
```

我们的 `Piece` 类现在：

- 告诉其子类它们必须实现一个名为 `canMoveTo` 的方法，该方法与给定签名兼容。如果一个类扩展了 `Piece` 但忘记实现抽象的 `canMoveTo` 方法，那在编译时就是一个类型错误：当你实现一个抽象类时，你也必须实现其抽象方法。
- 为 `moveTo` 提供了默认实现（其子类可以根据需要重写）。我们没有在 `moveTo` 上放置访问修饰符，所以它默认是 `public` 的，意味着它可以从任何其他代码中读取和写入。

让我们更新 `King` 来实现 `canMoveTo`，以满足这个新要求。我们还将为方便起见实现一个 `distanceFrom` 函数，这样我们可以轻松计算两个棋子之间的距离：

```
// ...
class Position {
  // ...
  distanceFrom(position: Position) {
    return {
      rank: Math.abs(position.rank - this.rank),
      file: Math.abs(position.file.charCodeAt(0) -
this.file.charCodeAt(0))
    }
  }
}

class King extends Piece {
```

```
  canMoveTo(position: Position) {
    let distance = this.position.distanceFrom(position)
    return distance.rank < 2 && distance.file < 2
  }
}
```

当我们创建新游戏时，我们会自动创建棋盘和一些棋子：

```
// ...
class Game {
  private pieces = Game.makePieces()

  private static makePieces() {
    return [

      // 国王
      new King('White', 'E', 1),
      new King('Black', 'E', 8),

      // 皇后
      new Queen('White', 'D', 1),
      new Queen('Black', 'D', 8),

      // 主教
      new Bishop('White', 'C', 1),
      new Bishop('White', 'F', 1),
      new Bishop('Black', 'C', 8),
      new Bishop('Black', 'F', 8),

      // ...
    ]
  }
}
```

由于我们严格地类型化了 `Rank` 和 `File`，如果我们输入了另一个字母（如 `'J'`）或超出范围的数字（如 `12`），TypeScript 会给我们一个编译时错误（[图5-2]）。

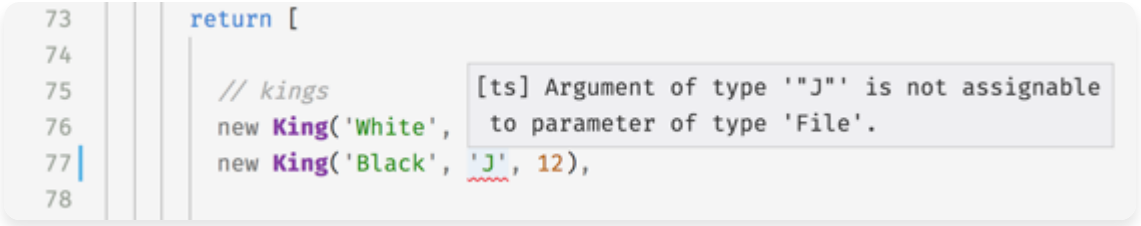


图5-2. TypeScript帮助我们坚持使用有效的行和列

这足以展示TypeScript类是如何工作的——我将避免深入细节，比如如何知道骑士何时可以吃掉一个棋子、主教如何移动等等。如果你有雄心，看看你能否使用我们到目前为止所做的作为起点，自己实现游戏的其余部分。

总结：

- 使用 `class` 关键字声明类。使用 `extends` 关键字扩展它们。
- 类可以是具体的或 `abstract` 的。抽象类可以有 `abstract` 方法和 `abstract` 属性。
- 方法可以是 `private`、`protected` 或默认的 `public`。它们可以是实例方法或静态方法。
- 类可以有实例属性，这些属性也可以是 `private`、`protected` 或默认的 `public`。你可以在构造函数参数中或作为属性初始化器声明它们。
- 在声明实例属性时，你可以将它们标记为 `readonly`。

super调用

像JavaScript一样，TypeScript支持 `super` 调用。 如果你的子类重写了在其父类上定义的方法（比如说，如果 `Queen` 和 `Piece` 都实现了 `take` 方法），子实例可以进行 `super` 调用来调用其父类版本的方法（例如， `super.take` ）。有两种 `super` 调用：

- 方法调用，如 `super.take` 。
- 构造函数调用，具有特殊形式 `super()` 并且只能从构造函数中调用。如果你的子类有构造函数，你必须在子类的构造函数中调用 `super()` 来正确连接类（别担心，如果你忘记了 TypeScript 会警告你；它在这方面像一个酷炫的未来机器人大象）。

注意你只能通过 `super` 访问父类的方法，而不是它的属性。

使用 this 作为返回类型

就像你可以将 `this` 用作值一样，你也可以将它用作类型（就像我们在”为 this 添加类型”中做的那样）。在处理类时，`this` 类型对于注释方法的返回类型很有用。

例如，让我们构建一个简化版的 ES6 `Set` 数据结构，支持两个操作：向集合添加数字，以及检查给定数字是否在集合中。你这样使用它：

```
let set = new Set
set.add(1).add(2).add(3)
set.has(2) // true
set.has(4) // false
```

让我们定义 `Set` 类，从 `has` 方法开始：

```
class Set {
  has(value: number): boolean {
    // ...
  }
}
```

那么 `add` 呢？当你调用 `add` 时，你得到一个 `Set` 的实例。我们可以这样定义类型：

```
class Set {
  has(value: number): boolean {
    // ...
  }
  add(value: number): Set {
    // ...
  }
}
```

到目前为止，一切都很好。当我们尝试继承 `Set` 时会发生什么？

```
class MutableSet extends Set {
  delete(value: number): boolean {
    // ...
  }
}
```

当然，`Set` 的 `add` 方法仍然返回一个 `Set`，我们需要在子类中用 `MutableSet` 覆盖它：

```
class MutableSet extends Set {
  delete(value: number): boolean {
    // ...
  }
  add(value: number): MutableSet {
    // ...
  }
}
```

在处理扩展其他类的类时，这可能会变得有点繁琐——你必须为每个返回 `this` 的方法覆盖签名。如果你最终不得不覆盖每个方法来满足类型检查器，那么从基类继承还有什么意义呢？

相反，你可以使用 `this` 作为返回类型注释，让 TypeScript 为你完成工作：

```
class Set {
  has(value: number): boolean {
    // ...
  }
  add(value: number): this {
    // ...
  }
}
```

现在，你可以从 `MutableSet` 中移除 `add` 覆盖，因为 `Set` 中的 `this` 指向 `Set` 实例，而 `MutableSet` 中的 `this` 指向 `MutableSet` 实例：

```
class MutableSet extends Set {
  delete(value: number): boolean {
    // ...
  }
}
```

这对于处理链式 API 来说是一个非常方便的功能，就像我们在”建造者模式”中做的那样。

接口

当你使用类时，你会经常发现自己将它们与接口一起使用。

像类型别名一样，接口是命名类型的一种方式，这样你就不必内联定义它。类型别名和接口大多是同一事物的两种语法（像函数表达式和函数声明），但有一些小差异。让我们从它们的共同点开始。考虑以下类型别名：

```
type Sushi = {
  calories: number
  salty: boolean
  tasty: boolean
}
```

将它重写为接口很容易：

```
interface Sushi {
  calories: number
  salty: boolean
  tasty: boolean
}
```

在你使用 `Sushi` 类型别名的任何地方，你也可以使用你的 `Sushi` 接口。两个声明都定义了形状，这些形状可以相互赋值（实际上，它们是相同的！）。

当你开始组合类型时，事情变得更有趣。让我们除了 `Sushi` 之外再建模另一种食物：

```
type Cake = {
  calories: number
  sweet: boolean
  tasty: boolean
}
```

很多食物都有卡路里并且美味——不只是 `Sushi` 和 `Cake`。让我们将 `Food` 提取到它自己的类型中，并根据它重新定义我们的食物：

```
type Food = {
  calories: number
  tasty: boolean
}
type Sushi = Food & {
  salty: boolean
}
type Cake = Food & {
  sweet: boolean
}
```

几乎等效地，你也可以用接口来做：

```
interface Food {
  calories: number
  tasty: boolean
}
interface Sushi extends Food {
  salty: boolean
}
interface Cake extends Food {
  sweet: boolean
}
```

注意

接口不必扩展其他接口。实际上，接口可以扩展任何形状：对象 `type`、`class` 或另一个 `interface`。

类型和接口之间的区别是什么？有三个，而且它们很微妙。

第一个是类型别名更通用，因为它们的右侧可以是任何类型，包括类型表达式（一个类型，以及可能一些类型运算符如 `&` 或 `|`）；对于接口，右侧必须是一个形状。例如，没有办法将以下类型别名重写为接口：

```
type A = number
type B = A | string
```

第二个区别是当你扩展一个接口时，TypeScript会确保你正在扩展的接口可以赋值给你的扩展。例如：

```
interface A {
  good(x: number): string
  bad(x: number): string
}

interface B extends A {
  good(x: string | number): string
  bad(x: string): string  // 错误 TS2430: 接口'B'错误地扩展了
}                         // 接口'A'。类型'number'不能赋值
                          // 给类型'string'。
```

当你使用交集类型(intersection types)时情况并非如此：如果你将上个示例中的接口转换为类型别名，并将`extends`转换为交集(`&`)，TypeScript会尽力将你的扩展与它扩展的类型结合起来，为`bad`产生重载签名而不是编译时错误（在你的代码编辑器中试试！）。

当你为对象类型建模继承时，TypeScript对接口进行的可赋值性检查可以是捕获错误的有用工具。

第三个区别是同一作用域内具有相同名称的多个接口会自动合并；同一作用域内具有相同名称的多个*类型别名*会抛出编译时错误。这是一个称为*声明合并*的特性。

声明合并

声明合并是TypeScript自动组合共享相同名称的多个声明的方式。当我们介绍枚举时提到过它（[“Enums”]），在使用其他特性如`namespace`声明时也会遇到它（见[“Namespaces”]）。在本节中，我们将在接口的上下文中简要介绍声明合并。要进行更深入的了解，请转到[“Declaration Merging”]。

例如，如果你声明两个名称相同的`User`接口，那么TypeScript会自动将它们组合成一个接口：

```
// User有一个字段，name
interface User {
  name: string
}

// User现在有两个字段，name和age
interface User {
  age: number
}

let a: User = {
  name: 'Ashley',
  age: 30
}
```

这是如果你用类型别名重复该示例会发生的情况：

```
type User = {  // 错误 TS2300: 重复的标识符'User'。
  name: string
}

type User = {  // 错误 TS2300: 重复的标识符'User'。
  age: number
}
```

注意两个接口不能冲突；如果一个将`property`类型化为`T`，另一个将其类型化为`U`，且`T`和`U`不相同，那么你会得到一个错误：

```
interface User {
  age: string
}

interface User {
  age: number  // 错误 TS2717: 后续属性声明必须具有
}              // 相同类型。属性'age'必须是'string'类型，
               // 但这里是'number'类型。
```

如果你的接口声明了泛型（跳转到[“Polymorphism”]了解更多），这些泛型必须以完全相同的方式声明才能使两个接口可以合并——甚至包括泛型的名称！

```
interface User<Age extends number> {  // 错误 TS2428: 'User'的所有声明
  age: Age                            // 必须具有相同的类型参数。
}

interface User<Age extends string> {
  age: Age
}
```

有趣的是，这是TypeScript不仅检查两个类型是否可赋值，而且检查是否*相同*的罕见地方。

实现

当你声明一个类时，你可以使用`implements`关键字来说明它满足特定的接口。就像其他显式类型注解一样，这是一种方便的方法，可以添加类型级约束，确保你的类在尽可能接近实现本身的地方正确实现，这样不正确实现产生的错误就不会出现在下游不太清楚为什么抛出错误的地方。这也是实现适配器、工厂和策略等常见设计模式的熟悉方式（在本章末尾查看一些示例）。

这是它的样子：

```
interface Animal {
  eat(food: string): void
  sleep(hours: number): void
}

class Cat implements Animal {
  eat(food: string) {
    console.info('Ate some', food, '. Mmm!')
  }
  sleep(hours: number) {
    console.info('Slept for', hours, 'hours')
  }
}
```

`Cat`必须实现`Animal`声明的每个方法，如果需要，可以在此基础上实现更多方法和属性。

接口可以声明实例属性，但它们不能声明可见性修饰符（`private`、`protected`和`public`），也不能使用`static`关键字。你也可以将实例属性标记为`readonly`，就像我们在Objects（在[第3章]）中为对象类型所做的那样：

```
interface Animal {
  readonly name: string
  eat(food: string): void
```

```
  sleep(hours: number): void
}
```

你不仅限于实现一个接口——你可以实现任意多个：

```
interface Animal {
  readonly name: string
  eat(food: string): void
  sleep(hours: number): void
}

interface Feline {
  meow(): void
}

class Cat implements Animal, Feline {
  name = 'Whiskers'
  eat(food: string) {
    console.info('Ate some', food, '. Mmm!')
  }
  sleep(hours: number) {
    console.info('Slept for', hours, 'hours')
  }
  meow() {
    console.info('Meow')
  }
}
```

所有这些功能都是完全类型安全的。如果您忘记实现某个方法或属性，或者实现不正确，TypeScript 将会提供帮助（见[图 5-3]）。

图 5-3. TypeScript 在您忘记实现必需方法时抛出错误

实现接口与扩展抽象类

实现接口与扩展抽象类非常相似。区别在于接口更通用和轻量，而抽象类更专用和功能丰富。

接口是一种对形状进行建模的方式。在值层面上，这意味着对象、数组、函数、类或类实例。接口不生成 JavaScript 代码，只在编译时存在。

抽象类只能对类进行建模。它生成的运行时代码是 JavaScript 类。抽象类可以有构造函数，提供默认实现，并为属性和方法设置访问修饰符。接口无法做到这些。

使用哪个取决于您的用例。当多个类之间共享实现时，使用抽象类。当您需要一种轻量的方式来表示”这个类是 `T` “时，使用接口。

类是结构化类型的

就像 TypeScript 中的每种其他类型一样，TypeScript 通过结构而不是名称来比较类。类与任何共享其形状的其他类型兼容，包括定义与类相同属性或方法的普通对象。对于来自 C#、Java、Scala 和大多数其他使用名义类型的语言的开发者来说，这点很重要。这意味着如果您有一个接受 `Zebra` 的函数，而您给它一个 `Poodle`，TypeScript 可能不会介意：

```
class Zebra {
  trot() {
    // ...
  }
}

class Poodle {
  trot() {
    // ...
  }
}

function ambleAround(animal: Zebra) {
  animal.trot()
}

let zebra = new Zebra
let poodle = new Poodle

ambleAround(zebra)   // OK
ambleAround(poodle)  // OK
```

正如您当中的系统发育学家所知道的，斑马不是贵宾犬——但 TypeScript 不介意！只要 `Poodle` 可以赋值给 `Zebra`，TypeScript 就没问题，因为从我们函数的角度来看，两者是可互换的；重要的只是它们实现了 `.trot`。如果您使用几乎任何其他对类进行名义类型化的语言，这段代码会引发错误；但 TypeScript 彻底采用结构化类型，所以这段代码是完全可以接受的。

这个规则的例外是具有 `private` 或 `protected` 字段的类：当检查形状是否可以赋值给类时，如果类具有任何 `private` 或 `protected` 字段，并且形状不是该类的实例或该类的子

类的实例，那么形状就不能赋值给该类：

```
class A {
  private x = 1
}
class B extends A {}
function f(a: A) {}

f(new A)   // OK
f(new B)   // OK

f({x: 1})  // Error TS2345: Argument of type '{x: number}' is not
           // assignable to parameter of type 'A'. Property 'x' is
           // private in type 'A' but not in type '{x: number}'.
```

类同时声明值和类型

TypeScript 中大多数可以表达的东西要么是值要么是类型：

```
// values
let a = 1999
function b() {}

// types
type a = number
interface b {
  (): void
}
```

在 TypeScript 中，类型和值是分别命名空间的。根据您如何使用术语（在此例中为 `a` 或 `b`），TypeScript 知道是否将其解析为类型或值：

```
// ...
if (a + 1 > 3) //... // TypeScript 从上下文推断您指的是值 a

let x: a = 3         // TypeScript 从上下文推断你意思是类型 a
```

这种*上下文项解析*真的很棒，让我们能够实现一些酷炫的东西，比如实现伴生类型(companion types)（参见[“伴生对象模式(Companion Object Pattern)”]）。

类和枚举是特殊的。它们之所以独特，是因为它们既在类型命名空间中生成类型，也在值命名空间中生成值：

```
class C {}
let c: C
  = new C

enum E {F, G}
```

```
let e: E
  = E.F
```

❶

在这个上下文中，`C` 指的是我们 `C` 类的实例类型。

❷

在这个上下文中，`C` 指的是 `C` 这个值。

❸

在这个上下文中，`E` 指的是我们 `E` 枚举的类型。

❹

在这个上下文中，`E` 指的是 `E` 这个值。

当我们使用类时，我们需要一种方式来表达”这个变量应该是这个类的实例”，枚举也是如此（“这个变量应该是这个枚举的成员”）。因为类和枚举在类型层面生成类型，所以我们能够轻松地表达这种”is-a”关系。[2]

我们还需要一种方式在运行时表示类，以便我们可以使用 `new` 实例化它，调用静态方法，进行元编程，并用 `instanceof` 操作它——所以类也需要生成一个值。

在前面的例子中，`C` 指的是类 `C` 的*实例*。如何表示 `C` 类本身？我们使用 `typeof` 关键字（TypeScript 提供的类型操作符，类似于 JavaScript 的值级别 `typeof`，但用于类型）。

让我们创建一个 `StringDatabase` 类——世界上最简单的数据库：

```
type State = {
  [key: string]: string
}

class StringDatabase {
  state: State = {}
  get(key: string): string | null {
```

```
    return key in this.state ? this.state[key] : null
  }
  set(key: string, value: string): void {
    this.state[key] = value
  }
  static from(state: State) {
    let db = new StringDatabase
    for (let key in state) {
      db.set(key, state[key])
    }
    return db
  }
}
```

这个类声明生成了什么类型？实例类型 `StringDatabase`：

```
interface StringDatabase {
  state: State
  get(key: string): string | null
  set(key: string, value: string): void
}
```

以及构造函数类型 `typeof StringDatabase`：

```
interface StringDatabaseConstructor {
  new(): StringDatabase
  from(state: State): StringDatabase
}
```

也就是说，`StringDatabaseConstructor` 有一个方法 `.from`，使用 `new` 构造函数会得到一个 `StringDatabase` 实例。结合起来，这两个接口建模了类的构造函数和实例两个方面。

那个 `new()` 部分被称为*构造函数签名*，这是 TypeScript 表示给定类型可以使用 `new` 操作符实例化的方式。因为 TypeScript 是结构化类型的，这是我们描述类的最佳方式：类是任何可以被 `new` 的东西。

在这种情况下，构造函数不接受任何参数，但你也可以用它来声明接受参数的构造函数。例如，假设我们更新 `StringDatabase` 以接受可选的初始状态：

```
class StringDatabase {
  constructor(public state: State = {}) {}
  // ...
}
```

我们然后可以将 `StringDatabase` 的构造函数签名类型化为：

```
interface StringDatabaseConstructor {
  new(state?: State): StringDatabase
  from(state: State): StringDatabase
}
```

所以，类声明不仅在值和类型层面生成项，而且在类型层面生成两个项：一个表示类的实例；一个表示类构造函数本身（可通过 `typeof` 类型操作符访问）。

多态性

像函数和类型一样，类和接口对泛型类型参数有丰富的支持，包括默认值和边界。 你可以将泛型作用域限定为整个类或接口，或者限定为特定方法：

```
class MyMap<K, V> {
  constructor(initialKey: K, initialValue: V) {
    // ...
  }
  get(key: K): V {
    // ...
  }
  set(key: K, value: V): void {
    // ...
  }
  merge<K1, V1>(map: MyMap<K1, V1>): MyMap<K | K1, V | V1> {
    // ...
  }
  static of<K, V>(k: K, v: V): MyMap<K, V> {
    // ...
  }
}
```

❶

在声明 `class` 时绑定类作用域的泛型类型。这里， `K` 和 `V` 对 `MyMap` 上的每个实例方法和实例属性都可用。

❷

注意你不能在 `constructor` 中声明泛型类型。相反，将声明移到你的 `class` 声明中。

❸

在类内部的任何地方使用类作用域的泛型类型。

❹

实例方法可以访问类级别的泛型，也可以在此基础上声明自己的泛型。`.merge` 使用了类级别的泛型 `K` 和 `V`，同时还声明了两个自己的泛型 `K1` 和 `V1`。

❺

静态方法无法访问其类的泛型，就像在值层面它们无法访问其类的实例变量一样。`of` 方法无法访问在 [

❶

] 中声明的 `K` 和 `V`；相反，它声明了自己的 `K` 和 `V` 泛型。
你也可以将泛型绑定到接口上：

```
interface MyMap<K, V> {
  get(key: K): V
  set(key: K, value: V): void
}
```

和函数一样，你可以显式地将具体类型绑定到泛型，或者让 TypeScript 为你推断类型：

```
let a = new MyMap<string, number>('k', 1) // MyMap<string, number>
let b = new MyMap('k', true) // MyMap<string, boolean>

a.get('k')
b.set('k', false)
```

Mixins

JavaScript 和 TypeScript 没有 `trait` 或 `mixin` 关键字，但我们可以直接实现它们。这两种都是模拟*多重继承*（继承多个其他类的类）和进行*面向角色编程*的方法，这是一种编程风格，你不会说”这个东西是一个 `Shape` “，而是描述一个东西的属性，比如”它可以被测量”或”它有四条边”。不是”is-a”关系，而是描述”can”和”has-a”关系。

让我们构建一个 mixin 实现。

Mixins 是一种模式，允许我们将行为和属性*混合到*类中。按照约定，mixins：

- 可以有状态（即实例属性）
- 只能提供具体方法（不是抽象方法）
- 可以有构造函数，按照它们的类被混入的相同顺序调用

TypeScript 没有内置的 mixins 概念，但我们很容易自己实现它们。例如，让我们为 TypeScript 类设计一个调试库。我们称它为 `EZDebug`。该库通过让你记录使用该库的任何类的信息来工作，以便你可以在运行时检查它们。我们这样使用它：

```
class User {
  // ...
}

User.debug() // 评估为 'User({"id": 3, "name": "Emma Gluzman"})'
```

有了标准的 `.debug` 接口，我们的用户将能够调试任何东西！让我们构建它。我们将用一个 mixin 建模，称为 `withEZDebug`。mixin 只是一个接受类构造函数并返回类构造函数的函数，所以我们的 mixin 可能看起来像这样：

```
type ClassConstructor = new(...args: any[]) => {}

function withEZDebug<C extends ClassConstructor>(Class: C) {
  return class extends Class {
    constructor(...args: any[]) {
```

```
        super(...args)
      }
    }
  }
```

[

]

我们首先声明一个类型 `ClassConstructor`，它表示任何构造函数。由于 TypeScript 完全是结构化类型的，我们说构造函数是任何可以被 `new` 的东西。我们不知道构造函数可能有什么类型的参数，所以我们说它接受任意数量的任意类型参数。

[

]

我们声明带有单个类型参数 `C` 的 `withEZDebug` mixin。`C` 必须至少是一个类构造函数，我们用 `extends` 子句强制执行这一点。我们让 TypeScript 推断 `withEZDebug` 的返回类型，这是 `C` 和我们新匿名类的交集。

[

]

由于 mixin 是一个接受构造函数并返回构造函数的函数，我们返回一个匿名类构造函数。

[

]

类构造函数必须接受*至少*你传入的类可能接受的参数。但记住，由于我们事先不知道你可能传入什么类，我必须保持尽可能通用，这意味着任意数量的任意类型参数——就像

`ClassConstructor`。
[

]
最后，由于这个匿名类扩展了另一个类，为了正确连接一切，我们需要记住也调用 `Class` 的构造函数。
就像普通的 JavaScript 类一样，如果你在 `constructor` 中没有任何更多逻辑，你可以省略 [

] 和 [

] 行。我们不打算在这个 `withEZDebug` 示例的构造函数中放入任何逻辑，所以我们可以省略它们。

现在我们已经设置了样板代码，是时候进行一些调试魔法了。当我们调用 `.debug` 时，我们想要记录类的构造函数名称和实例的值：

```
type ClassConstructor = new(...args: any[]) => {}

function withEZDebug<C extends ClassConstructor>(Class: C) {
  return class extends Class {
    debug() {
      let Name = Class.constructor.name
      let value = this.getDebugValue()
      return Name + '(' + JSON.stringify(value) + ')'
    }
  }
}
```

但是等等！我们如何确保类实现了 `.getDebugValue` 方法，以便我们可以调用它？在继续之前先思考一下——你能想出来吗？

答案是，我们不是接受任何旧类，而是使用泛型类型来确保传递给 `withEZDebug` 的类定义了 `.getDebugValue` 方法：

```
type ClassConstructor<T> = new(...args: any[]) => T

function withEZDebug<C extends ClassConstructor<{
  getDebugValue(): object
}>>(Class: C) {
  // ...
}
```

❶

我们向 `ClassConstructor` 添加泛型类型参数。

❷

我们将形状类型绑定到 `ClassConstructor`，`C`，强制要求传递给 [`withEZDebug`] 的构造函数至少定义 `.getDebugValue` 方法。

就是这样！那么，你如何使用这个不可思议的调试工具呢？像这样：

```
class HardToDebugUser {
  constructor(
    private id: number,
    private firstName: string,
    private lastName: string
  ) {}
  getDebugValue() {
    return {
      id: this.id,
      name: this.firstName + ' ' + this.lastName
    }
  }
}

let User = withEZDebug(HardToDebugUser)
```

```
let user = new User(3, 'Emma', 'Gluzman')
user.debug() // 计算结果为 'User({"id": 3, "name": "Emma Gluzman"})'
```

很酷，对吧？你可以将任意数量的混入(mixin)应用到一个类上，产生具有越来越丰富行为的类，所有这些都以类型安全的方式进行。混入帮助封装行为，并且是指定可重用行为的表达性方式。[4]

装饰器(Decorators)

装饰器(Decorators)是一个实验性的 TypeScript 特性，为我们提供了一种干净的语法来对类、类方法、属性和方法参数进行元编程。它们只是在你装饰的对象上调用函数的语法。

TSC 标志： experimentalDecorators

因为它们仍然是实验性的——这意味着它们可能会以向后不兼容的方式更改，甚至可能在未来的 TypeScript 版本中完全移除——装饰器隐藏在 TSC 标志后面。如果你接受这一点，并希望尝试这个特性，在你的 *tsconfig.json* 中设置 `"experimentalDecorators": true` 并继续阅读。

为了了解装饰器是如何工作的，让我们从一个例子开始：

```
@serializable
class APIPayload {
  getValue(): Payload {
    // ...
  }
}
```

`@serializable` 类装饰器包装我们的 `APIPayload` 类，并可选地返回一个替换它的新类。没有装饰器，你可能会用以下方式实现同样的功能：

```
let APIPayload = serializable(class APIPayload {
  getValue(): Payload {
    // ...
  }
})
```

对于每种类型的装饰器，TypeScript 要求你在作用域中有一个具有给定名称和该类型装饰器所需签名的函数（参见[表 5-1]）。

[表 5-1.]不同类型装饰器函数的预期类型签名 {#calibre_link-207}

你正在装饰的内容	预期类型签名
类	`(Constructor: {new(...any[]) => any}) => any`

你正在装饰的内容	预期类型签名
方法	`(classPrototype: {}, methodName: string, descriptor: [PropertyDescriptor]) => any`
静态方法	`(Constructor: {new(...any[]) => any}, methodName: string, descriptor: PropertyDescriptor) => any`
方法参数	`(classPrototype: {}, paramName: string, index: number) => void`
静态方法参数	`(Constructor: {new(...any[]) => any}, paramName: string, index: number) => void`
属性	`(classPrototype: {}, propertyName: string) => any`
静态属性	`(Constructor: {new(...any[]) => any}, propertyName: string) => any`
属性获取器/设置器	`(classPrototype: {}, propertyName: string, descriptor: [PropertyDescriptor]) => any`
静态属性获取器/设置器	`(Constructor: {new(...any[]) => any}, propertyName: string, descriptor: PropertyDescriptor) => any`

TypeScript 不提供任何内置的装饰器：无论你使用什么装饰器，你都必须自己实现（或从 NPM 安装）。每种装饰器的实现——对于类、方法、属性和函数参数——都是满足特定签名的常规函数，具体取决于它装饰的内容。例如，我们的 `@serializable` 装饰器可能看起来像这样：

```
type ClassConstructor<T> = new(...args: any[]) => T

function serializable<
  T extends ClassConstructor<{
    getValue(): Payload
```

```
  }>
>(Constructor: T) {
  return class extends Constructor {
    serialize() {
      return this.getValue().toString()
    }
  }
}
```

❶

记住，`new()` 是我们在TypeScript中结构化类型化类构造函数的方式。对于可以被扩展（使用 `extends`）的类构造函数，TypeScript要求我们使用 `any` 扩展来类型化其参数：`new(...any[])`。

❷

`@serializable` 可以装饰任何实例实现了 `.getValue` 方法的类，该方法返回一个 `Payload`。

❸

类装饰器(class decorators)是接受单个参数（类）的函数。如果装饰器函数返回一个类（如示例中所示），它将在运行时替换它所装饰的类；否则，它将返回原始类。

❹

为了装饰类，我们返回一个扩展它的类，并在过程中添加一个 `.serialize` 方法。

当我们尝试调用 `.serialize` 时会发生什么？

```
let payload = new APIPayload
let serialized = payload.serialize() // Error TS2339: Property
'serialize' does
                                     // not exist on type
'APIPayload'.
```

TypeScript假设装饰器不会改变它所装饰对象的形状——这意味着你没有添加或删除方法和属性。它在编译时检查你返回的类是否可以分配给你传入的类，但在撰写本文时，TypeScript不会跟踪你在装饰器中所做的扩展。

在TypeScript中的装饰器(decorators)成为更成熟的功能之前，我建议你避免使用它们，而坚持使用常规函数：

```
let DecoratedAPIPayload = serializable(APIPayload)
let payload = new DecoratedAPIPayload
payload.serialize()                  // string
```

我们在本书中不会更深入地讨论装饰器。更多信息，请前往官方文档。

模拟final类

虽然TypeScript不支持类或方法的`final`关键字，但很容易为类模拟它。如果你之前没有使用过面向对象语言，`final`是一些语言用来标记类为不可扩展，或方法为不可重写的关键字。

要在TypeScript中模拟`final`类，我们可以利用私有构造函数：

```
class MessageQueue {
  private constructor(private messages: string[]) {}
}
```

当`constructor`被标记为`private`时，你不能`new`该类或扩展它：

```
class BadQueue extends MessageQueue {}  // Error TS2675: Cannot extend
a class
                                        // 'MessageQueue'. Class
constructor is
                                        // marked as private.

new MessageQueue([])                    // Error TS2673: Constructor
of class
                                        // 'MessageQueue' is private
and only
                                        // accessible within the class
                                        // declaration.
```

除了防止你扩展类——这是我们想要的——私有构造函数也防止你直接实例化它。但对于`final`类，我们确实希望能够实例化类，只是不要扩展它。我们如何保留第一个限制但摆脱第二个限制呢？很简单：

```
class MessageQueue {
  private constructor(private messages: string[]) {}
  static create(messages: string[]) {
    return new MessageQueue(messages)
```

```
  }
}
```

这稍微改变了 `MessageQueue` 的API，但它在编译时很好地防止了扩展：

```
class BadQueue extends MessageQueue {}  // Error TS2675: Cannot extend
a class
                                        // 'MessageQueue'. Class
constructor is
                                        // marked as private.

MessageQueue.create([]) // MessageQueue
```

设计模式

如果我们不在TypeScript中实现一两个设计模式，这就不会是一个关于面向对象编程的章节，对吧?

工厂模式

*工厂模式(factory pattern)*是一种创建某种类型对象的方式，将决定创建哪个具体对象的决策留给创建该对象的特定工厂。

让我们建立一个鞋子工厂。我们首先定义一个`Shoe`类型和几种类型的鞋子：

```
type Shoe = {
  purpose: string
}

class BalletFlat implements Shoe {
  purpose = 'dancing'
}

class Boot implements Shoe {
  purpose = 'woodcutting'
}

class Sneaker implements Shoe {
  purpose = 'walking'
}
```

注意这个示例使用了一个`type`，但我们同样可以使用`interface`来代替。

现在，让我们创建一个鞋子工厂：

let Shoe = { create(type: 'balletFlat' | 'boot' | 'sneaker'): Shoe { switch (type) { case 'balletFlat': return new BalletFlat case 'boot': return new Boot case 'sneaker': return new Sneaker } } }

```
[![1](images/000000.png)]{#calibre_link-216 .calibre4}

:   为`type`使用联合类型有助于使`.create`尽可能类型安全，防止消费者在编译时传入无效的`type`。
```

[![2](images/000001.png)]{#calibre_link-217 .calibre4}

: 在`type`上使用switch语句使TypeScript能够轻松强制我们已经处理了每种类型的`Shoe`。

在这个例子中，我们使用伴生对象模式（参见["Companion Object Pattern"]）来声明一个类型`Shoe`和一个同名的值`Shoe`（记住TypeScript为值和类型有单独的命名空间），作为一种方式来表明该值提供了操作该类型的方法。要使用工厂，我们只需调用`.create`：

Shoe.create(‘boot’) // Shoe

太棒了！我们有了一个工厂模式。注意我们可以更进一步，在`Shoe.create`的类型签名中指明传入`'boot'`会得到一个`Boot`，`'sneaker'`会得到一个`Sneaker`，等等，但这会破坏工厂模式给我们的抽象（即消费者不应该知道他们会得到什么具体的类，只要知道该类满足特定的接口即可）。

建造者模式 {#builder-pattern .calibre17}

*建造者模式*是一种将对象的构造与对象实际实现方式分离的方法。[]{#calibre_link-840 .calibre4 primary="design patterns" secondary="builder pattern" data-type="indexterm"}[]{#calibre_link-742 .calibre4 primary="classes" secondary="design patterns" tertiary="builder pattern" data-type="indexterm"}[]{#calibre_link-709 .calibre4 primary="builder pattern" data-type="indexterm"} 如果你使用过JQuery，或ES6数据结构如`Map`和`Set`，这种API风格应该看起来很熟悉。它看起来是这样的：

new RequestBuilder() .setURL(‘/users’) .setMethod(‘get’) .setData({firstName: ‘Anna’}) .send()

我们如何实现`RequestBuilder`？很简单——我们从一个空类开始：

class RequestBuilder {}

首先我们添加`.setURL`方法：

class RequestBuilder {

private url: string | null = null

setURL(url: string): this { this.url = url return this } }

[]{#calibre_link-220 .calibre4}

：　我们在私有实例变量`url`中跟踪用户设置的URL，我们将其初始化为`null`。

[]{#calibre_link-1857 .calibre4}

：　`setURL`的返回类型是`this`（参见["Using this as a Return Type"]），即用户调用`setURL`的`RequestBuilder`的特定实例。

现在让我们从示例中添加其他方法：

class RequestBuilder {

private data: object | null = null private method: ‘get’ | ‘post’ | null = null private url: string | null = null

setMethod(method: ‘get’ | ‘post’): this { this.method = method return this } setData(data: object): this { this.data = data return this } setURL(url: string): this { this.url = url return this }

send() { // … } }

就是这样。

注意 {#note-15 .calibre22}

这种传统的建造者设计并不完全安全：我们可以在设置方法、URL或数据之前调用`.send`，导

致运行时异常（记住，这是坏的异常类型）。有关如何改进此设计的一些想法，请参见练习4。[]{#calibre_link-741 .calibre4 primary="classes" secondary="design patterns" startref="ix_classdes" data-type="indexterm"}[]{#calibre_link-839 .calibre4 primary="design patterns" startref="ix_despatt" data-type="indexterm"}

总结 {#summary-2 .calibre13}

我们现在已经从各个角度探索了TypeScript类：如何声明类；如何从类继承并实现接口；如何将类标记为`abstract`使其无法实例化；如何使用`static`在类上放置字段或方法，不使用它在实例上放置；如何使用`private`、`protected`和`public`可见性修饰符控制对字段或方法的访问；以及如何使用`readonly`修饰符将字段标记为不可写。我们涵盖了如何安全地使用`this`和`super`，探索了类同时作为值和类型意味着什么，并讨论了类型别名和接口之间的区别、声明合并的基础知识，以及在类中使用泛型类型。最后，我们涵盖了一些使用类的更高级模式：mixins、装饰器和模拟`final`类。为了结束这一章，我们详细推导了一些使用类的常见模式。

练习 {#exercises-3 .calibre13}

1. 类和接口之间有什么区别？

2. 当你将类的构造函数标记为`private`时，这意味着你无法实例化或扩展该类。当你将其标记为`protected`时会发生什么？在你的代码编辑器中尝试一下，看看你能否弄明白。

3. 扩展我们在["Factory Pattern"]中开发的实现，使其更安全，但代价是稍微破坏抽象。更新实现，使消费者在编译时知道调用`Shoe.create('boot')`返回一个`Boot`，调用`Shoe.create('balletFlat')`返回一个`BalletFlat`（而不是都返回一个`Shoe`）。提示：回想一下["Overloaded Function Types"]。

4. [难度较高] 作为练习，思考一下你可能如何设计一个类型安全的建造者模式。扩展["Builder Pattern"]中的建造者模式：

1. 在编译时保证不能在至少设置了URL和方法之前调用`.send`。如果您还强制用户按特定顺序调用方法，会更容易做出这种保证吗？（提示：您可以返回什么而不是`this`？）

2. [更难] 如果您想做出这种保证，但仍然让人们可以按任意顺序调用方法，您会如何改变您的设计？（提示：您可以使用什么TypeScript功能来使每个方法的返回类型在每次方法调用后"添加"到`this`类型？）

^[1]^ 或者即将被JavaScript类支持。

^[2]^ 因为TypeScript是结构化类型的，当然，对于类的关系更像是"看起来像"——任何实现

与您的类相同形状的对象都可以分配给您的类的类型。

^[3]^ 请注意，TypeScript在这里很挑剔：构造函数类型的参数类型必须是`any[]`（不是`void`、`unknown[]`等），以便我们能够扩展它。

^[4]^ 少数几种语言——Scala、PHP、Kotlin和Rust等——实现了混入(mixins)的简化版本，称为*特质(traits)*。特质类似于混入，但没有构造函数，不支持实例属性。这使得连接它们变得更容易，并防止多个特质之间访问它们与基类共享状态时发生冲突。

第6章 高级类型 {#chapter-6.-advanced-types .calibre12}

TypeScript拥有世界级的类型系统，支持强大的类型级编程功能，这些功能甚至可能让最挑剔的Haskell程序员感到嫉妒。正如您现在所知，该类型系统不仅表达能力强，而且易于使用，使得声明类型约束和关系变得简单、简洁，并且大多数时候是推断出来的。

我们需要如此富有表达力和独特的类型系统，因为JavaScript如此动态。建模原型、动态绑定`this`、函数重载和不断变化的对象等特性需要丰富的类型系统和类型操作符工具带，这会让蝙蝠侠都要重新审视。

我将从深入研究TypeScript中的子类型化、可分配性、变性和类型拓宽开始本章，为您在前几章中发展的直觉提供更多定义。然后我将更详细地介绍TypeScript基于控制流的类型检查功能，包括细化和完整性，并继续介绍一些高级类型级编程功能：键入和映射对象类型、使用条件类型、定义自己的类型保护，以及类型断言和明确赋值断言等逃生舱。最后，我将介绍从您的类型中挤出更多安全性的高级模式：伴侣对象模式、改进元组类型的推断、模拟名义类型，以及安全扩展原型。

类型之间的关系 {#relationships-between-types .calibre13}

让我们首先更仔细地看看TypeScript中的类型关系。

子类型和超类型 {#subtypes-and-supertypes .calibre17}

我们在["谈论类型"]中稍微讨论了可分配性。现在您已经看到了TypeScript提供的大部分类型，我们可以更深入地研究，从顶部开始：什么是子类型？

子类型 {#subtype .calibre29}

如果您有两个类型`A`和`B`，并且`B`是`A`的子类型，那么您可以在任何需要`A`的地方安全地使用`B`（[图6-1]）。

<figure class="calibre33">

<h6 id="figure-6-1.-b-is-a-subtype-of-a" class="calibre34"><span
class="calibre">图6-1. B是A的子类型</h6>

</figure>

如果您回头看第3章最开始的[图3-1]，您会看到TypeScript内置的子类型关系。例如：

- Array是Object的子类型。

- Tuple是Array的子类型。

- 一切都是`any`的子类型。

- `never`是一切的子类型。

- 如果您有一个扩展`Animal`的类`Bird`，那么`Bird`是`Animal`的子类型。

从我刚才给出的子类型定义，这意味着：

- 在任何需要Object的地方，您也可以使用Array。

- 在任何需要Array的地方，您也可以使用Tuple。

- 在任何需要`any`的地方，您也可以使用Object。

- 您可以在任何地方使用`never`。

- 在任何需要`Animal`的地方，您也可以使用`Bird`。

正如您可能已经猜到的，超类型是子类型的反面。

超类型 {#supertype .calibre29}

如果您有两个类型`A`和`B`，并且`B`是`A`的超类型，那么您可以在任何需要`B`的地方安全地使用`A`（[图6-2]）。

<figure class="calibre33">

<h6 id="figure-6-2.-b-is-a-supertype-of-a" class="calibre34"><span
class="calibre">图6-2. B是A的超类型</h6>

</figure>

再次从[图3-1]中的流程图：

- Array是Tuple的超类型。

- Object是Array的超类型。

- Any是一切的超类型。

- Never不是任何事物的超类型。

- `Animal`是`Bird`的超类型。

这只是子类型工作方式的相反，没有更多。

变性 {#variance .calibre17}

对于大多数类型，直观地判断某个类型 `A` 是否是另一个类型 `B` 的子类型是相当容易的。对于像 `number`、`string` 等简单类型，您可以直接查看 [图3-1] 中的流程图，或者通过推理得出（"`number` 包含在联合类型 `number | string` 中，所以它必须是它的子类型"）。

但对于参数化（泛型）类型和其他更复杂的类型，情况变得更加复杂。考虑以下情况：

- 什么时候 `Array<A>` 是 `Array<B>` 的子类型?

- 什么时候形状 `A` 是另一个形状 `B` 的子类型?

- 什么时候函数 `(a: A) => B` 是另一个函数 `(c: C) => D` 的子类型?

包含其他类型的类型（即具有类型参数的类型如 `Array<A>`、具有字段的形状如 `{a: number}`，或函数如 `(a: A) => B`）的子类型规则更难推理，答案也不那么明确。事实上，这些复杂类型的子类型规则是编程语言之间的重大分歧点——几乎没有两种语言是相同的！

为了使以下规则更易于阅读，我将介绍一些语法，让我们能够更精确和简洁地讨论类型。这种语法不是有效的 TypeScript；它只是您和我在讨论类型时共享通用语言的一种方式。别担心，我发誓这种语法不是数学：

- `A <: B` 表示"`A` 是类型 `B` 的子类型或与之相同"。

- `A >: B` 表示"`A` 是类型 `B` 的超类型或与之相同"。

形状和数组协变性

为了对为什么语言在复杂类型的子类型规则上存在分歧有一些直觉，让我通过一个复杂类型的例子：形状来带您了解。假设您有一个描述应用程序中用户的形状。您可以用一对类似这样的类型来表示它：

// 从服务器获取的现有用户 type ExistingUser = { id: number name: string }

// 尚未保存到服务器的新用户 type NewUser = { name: string }

现在假设您公司的一名实习生被分配编写一些删除用户的代码。他们这样开始：

function deleteUser(user: {id?: number, name: string}) { delete user.id }

let existingUser: ExistingUser = { id: 123456, name: ‘Ima User’ }

deleteUser(existingUser)

`deleteUser` 接受类型为 `{id?: number, name: string}` 的对象，并传入类型为 `{id: number, name: string}` 的 `existingUser`。注意 `id` 属性的类型（`number`）是期望类型（`number | undefined`）的*子类型*。因此，整个对象 `{id: number, name: string}` 是 `{id?: number, name: string}` 的子类型，所以 TypeScript 允许这样做。

您看到这里的安全问题了吗？这是一个微妙的问题：将 `ExistingUser` 传递给 `deleteUser` 后，TypeScript 不知道用户的 `id` 已被删除，所以如果我们在用 `deleteUser(existingUser)` 删除它后读取 `existingUser.id`，TypeScript 仍然认为 `existingUser.id` 是 `number` 类型！

显然，在期望其超类型的地方使用对象类型可能是不安全的。那么为什么 TypeScript 允许这样做？一般来说，TypeScript 并不是设计为完全安全的；相反，它的类型系统试图在捕获真正的错误和易于使用之间取得平衡，而不需要您获得编程语言理论学位来理解为什么某些东西是错误的。这种特定的不安全情况是实际的：由于破坏性更新（如删除属性）在实践中相对罕见，TypeScript 是宽松的，允许您将对象分配给期望其超类型的地方。

那么相反的方向如何--您可以将对象分配给期望其子类型的地方吗?

让我们为遗留用户添加一个新类型，然后删除该类型的用户（假设您正在为同事在开始使用 TypeScript 之前编写的代码添加类型）：

type LegacyUser = { id?: number | string name: string }

let legacyUser: LegacyUser = { id: '793331', name: 'Xin Yang' }

deleteUser(legacyUser) // Error TS2345: Argument of type 'LegacyUser' is not // assignable to parameter of type '{id?: number | // undefined, name: string}'. Type 'string' is not // assignable to type 'number | undefined'.

当我们传递一个具有类型为期望类型的超类型的属性的形状时，TypeScript 会抱怨。这是因为 `id` 是 `string | number | undefined`，而 `deleteUser` 只处理 `id` 为 `number | undefined` 的情况。

TypeScript的行为如下：如果你期望一个形状，你也可以传递一个属性类型为其期望类型的`<:`的类型，但你不能传递一个属性类型为其期望类型的超类型的形状。当谈论类型时，我们说TypeScript形状（对象和类）在其属性类型中是*协变的*。也就是说，要使对象`A`可分配给对象`B`，其每个属性必须是`<:`其在`B`中对应属性。

更一般地说，协变只是四种方差中的一种：

不变

: 你想要确切的`T`。

协变

: 你想要`<:T`。

逆变

: 你想要`>:T`。

双变

: 你可以接受`<:T`或`>:T`。

在TypeScript中，每个复杂类型在其成员中都是协变的——对象、类、数组和函数返回类型——有一个例外：函数参数类型是*逆变的*。

注意

并非所有语言都做出同样的设计决定。在某些语言中，对象在其属性类型中是*不变的*，因为正如我们所看到的，协变属性类型可能导致不安全的行为。一些语言对可变和不可变对象有不同的规则（试着自己推理一下！）。一些语言——如Scala、Kotlin和Flow——甚至有明确的语法供程序员为其自己的数据类型指定方差。

在设计TypeScript时，其作者选择了易用性和安全性之间的平衡。当你使对象在其属性类型中不变时，尽管它更安全，但可能会使类型系统变得繁琐，因为你最终会禁止在实践中安全的东西（例如，如果我们没有在`deleteUser`中`delete` `id`，那么传入一个是期望类型超类型的对象就是完全安全的）。

函数方差

让我们从几个例子开始。

函数`A`是函数`B`的子类型，如果`A`具有与`B`相同或更少的元数(参数数量)，并且：

1. `A`的`this`类型要么未指定，要么是`>: B`的`this`类型。

2. `A`的每个参数都是`>:`其在`B`中对应参数。

3. `A`的返回类型是`<: B`的返回类型。

多读几遍，确保你理解每条规则的含义。你可能已经注意到，要使函数`A`成为函数`B`的子类型，我们说它的`this`类型和参数必须是`>:`它们在`B`中的对应物，而它的返回类型必须是`<:`！为什么方向像这样翻转？为什么不像对象、数组、联合类型等那样，每个组件（`this`类型、参数类型和返回类型）都简单地是`<:`？

为了回答这个问题，让我们自己推导出来。我们将从定义三种类型开始（为了清楚起见，我们将使用`class`，但这适用于任何`A <: B <: C`的类型选择）：

class Animal {} class Bird extends Animal { chirp() {} } class Crow extends Bird { caw() {} }

在这个例子中，`Crow`是`Bird`的子类型，`Bird`是`Animal`的子类型。也就是说，`Crow <: Bird <: Animal`。

现在，让我们定义一个接受`Bird`并让它chirp的函数：

function chirp(bird: Bird): Bird { bird.chirp() return bird }

到目前为止，一切都很好。TypeScript让你向`chirp`传递什么样的东西？

chirp(new Animal) // Error TS2345: Argument of type ‘Animal’ is not assignable chirp(new Bird) // to parameter of type ‘Bird’. chirp(new Crow)

你可以传递`Bird`的实例（因为那是`chirp`的参数`bird`的类型）或`Crow`的实例（因为它是`Bird`的子类型）。很好：传入子类型按预期工作。

让我们创建一个新函数。这次，它的参数将是一个*函数*：

function clone(f: (b: Bird) => Bird): void { // … }

`clone`需要一个函数`f`，它接受`Bird`并返回`Bird`。你可以安全地为`f`传递什么类型的函数？显然你可以传递一个接受`Bird`并返回`Bird`的函数：

function birdToBird(b: Bird): Bird { // … } clone(birdToBird) // OK

那么一个接受`Bird`并返回`Crow`或`Animal`的函数呢？

function birdToCrow(d: Bird): Crow { // … } clone(birdToCrow) // OK

function birdToAnimal(d: Bird): Animal { // … } clone(birdToAnimal) // Error TS2345: Argument of type ‘(d: Bird) => Animal’ is // not assignable to parameter of type ‘(b: Bird) => Bird’. // Type ‘Animal’ is not assignable to type ‘Bird’.

`birdToCrow`按预期工作，但`birdToAnimal`给我们一个错误。为什么？想象`clone`的实现看起来像这样：

function clone(f: (b: Bird) => Bird): void { let parent = new Bird let babyBird = f(parent) babyBird.chirp() }

如果我们向 `clone` 函数传递一个返回 `Animal` 的 `f`，那么我们就无法对其调用 `.chirp` 方法！因此 TypeScript 必须在编译时确保我们传入的函数返回*至少*一个 `Bird`。

我们称函数在其返回类型上是*协变的(covariant)*，这是一种高级说法，意思是对于一个函数成为另一个函数的子类型，其返回类型必须是 `<:` 另一个函数的返回类型。

好的，那么参数类型呢？

function animalToBird(a: Animal): Bird { // … } clone(animalToBird) // 正确

function crowToBird(c: Crow): Bird { // … } clone(crowToBird) // 错误 TS2345: 类型 ‘(c: Crow) => Bird’ 的参数不能 // 分配给类型 ‘(b: Bird) => Bird’ 的参数。

对于一个函数可分配给另一个函数，其参数类型（包括 `this`）都必须是 `>:` 另一个函数中对应的参数类型。要理解为什么，请想一下用户在将 `crowToBird` 传递给 `clone` 之前可能是如何实现它的。如果他们这样做会怎么样？

function crowToBird(c: Crow): Bird { c.caw() return new Bird }

现在如果 `clone` 用 `new Bird` 调用 `crowToBird`，我们会得到一个异常，因为 `.caw` 只在 `Crow` 上定义，而不是在所有 `Bird` 上。

这意味着函数在其参数和 `this` 类型上是*逆变的(contravariant)*。也就是说，对于一个函数成为另一个函数的子类型，它的每个参数及其 `this` 类型都必须是 `>:` 另一个函数中的对应参数。

谢天谢地，您不必记忆和背诵这些规则。当代码编辑器在您传递类型错误的函数时给您红色波浪线时，只需将这些规则放在脑海中，这样您就知道为什么 TypeScript 会给您这样的错误了。

TSC 标志：strictFunctionTypes

由于历史原因，TypeScript 中的函数默认情况下在其参数和 `this` 类型上实际上是协变的。要选择使用我们刚才探索的更安全的逆变行为，请确保在您的 *tsconfig.json* 中启用 `{"strictFunctionTypes": true}` 标志。

`strict` 模式包含 `strictFunctionTypes`，所以如果您已经使用 `{"strict": true}`，那就没问题了。

可分配性

子类型和超类型关系是任何静态类型语言的核心概念。它们对于理解*可分配性(assignability)*的工作原理也很重要（作为提醒，可分配性是指 TypeScript 关于是否可以在需要另一种类型 `B` 的地方使用类型 `A` 的规则）。

当 TypeScript 想要回答"类型 `A` 是否可分配给类型 `B`？"这个问题时，它遵循一些简单的规则。对于*非枚举类型*——如数组、布尔值、数字、对象、函数、类、类实例和字符串，包括字面量类型——如果以下任一条件为真，`A` 可分配给 `B`：

1. `A <: B`。

2. `A` 是 `any`。

规则1 就是子类型的定义：如果 `A` 是 `B` 的子类型，那么在需要 `B` 的地方也可以使用 `A`。

规则2 是规则1 的例外，是与 JavaScript 代码互操作的便利。

对于用 `enum` 或 `const enum` 关键字创建的*枚举类型*，如果以下任一条件为真，类型 `A` 可分配给枚举 `B`：

1. `A` 是枚举 `B` 的成员。

2. `B` 至少有一个 `number` 类型的成员，且 `A` 是 `number`。

规则1 与简单类型完全相同（如果 `A` 是枚举 `B` 的成员，那么 `A` 的类型就是 `B`，所以我们所说的就是 `B <: B`）。

规则2 是使用枚举的便利。正如我们在["枚举"]中谈到的，规则2 是 TypeScript 中不安全性的一个重要来源，这也是我建议抛弃婴儿和洗澡水，完全避免枚举的原因之一。

类型拓宽

类型拓宽 是理解 TypeScript 类型推断工作原理的关键。一般来说，TypeScript[]{#calibre_link-1643 .calibre4 primary="types" secondary="relationships between" tertiary="type widening" data-type="indexterm"}[]{#calibre_link-1587 .calibre4 primary="type widening" data-type="indexterm"} 在推断类型时会比较宽松，并且倾向于推断出更一般的类型，而不是最具体的类型。这让程序员的工作变得更轻松，意味着花费更少的时间来平息类型检查器的抱怨。[]{#calibre_link-1865 .calibre4 primary="widening types" see="type widening" data-type="indexterm"}

在第3章中，你已经看到了几个类型拓宽的实际例子。让我们看看更多的例子。

当你以允许后续修改的方式声明变量时（例如，使用 `let` 或 `var`），它的类型[]{#calibre_link-1741 .calibre4 primary="variables" secondary="mutable, type widening" data-type="indexterm"} 会从字面值拓宽到该字面值所属的基础类型：

let a = ‘x’ // string let b = 3 // number var c = true // boolean const d = {x: 3} // {x: number}

enum E {X, Y, Z} let e = E.X // E

不可变声明就不是这样了[]{#calibre_link-1740 .calibre4 primary="variables" secondary="immutable, no type widening" data-type="indexterm"}：

const a = ‘x’ // ‘x’ const b = 3 // 3 const c = true // true

enum E {X, Y, Z} const e = E.X // E.X

你可以使用显式类型注解来防止类型被拓宽：

let a: ‘x’ = ‘x’ // ‘x’ let b: 3 = 3 // 3 var c: true = true // true const d: {x: 3} = {x: 3} // {x: 3}

当你使用 `let` 或 `var` 重新赋值一个未被拓宽的类型时，TypeScript 会为你拓宽它。要告诉 TypeScript 保持窄化，在你的原始声明中添加显式类型注解：

const a = ‘x’ // ‘x’ let b = a // string

const c: ‘x’ = ‘x’ // ‘x’ let d = c // ‘x’

初始化为 `null` 或 `undefined` 的变量会被拓宽为 `any`：

let a = null // any a = 3 // any a = ‘b’ // any

但是当一个变量[]{#calibre_link-1296 .calibre4 primary="null type" secondary="variables initialized to, type widening" data-type="indexterm"}[]{#calibre_link-1715 .calibre4 primary="undefined type" secondary="variables initialized to, type widening" data-type="indexterm"}[]{#calibre_link-632 .calibre4 primary="any type" secondary="widening of variables initialized as null or undefined to" data-type="indexterm"}初始化为 `null` 或 `undefined` 离开其声明的作用域时，TypeScript 会为它分配一个确定的类型：

function x() { let a = null // any a = 3 // any a = ‘b’ // any return a }

x() // string

const 类型 {#the-const-type .calibre39}

TypeScript 提供了一个特殊的 `const` 类型，你可以使用它来在声明时退出类型拓宽。[]{#calibre_link-795 .calibre4 primary="const" secondary="using to opt out of type widening at declaration time" data-type="indexterm"}[]{#calibre_link-1590 .calibre4 primary="type widening" secondary="preventing with const type" data-type="indexterm"}将其用作类型断言（请阅读"类型断言"部分）：

```
let a = {x: 3} // {x: number} let b: {x: 3} // {x: 3} let c = {x: 3} as const // {readonly x: 3}
```

`const` 让你的类型退出拓宽并递归地将其成员标记为 `readonly`，即使对于深度嵌套的数据结构也是如此：

```
let d = [1, {x: 2}] // (number | {x: number})[] let e = [1, {x: 2}] as const // readonly [1, {readonly x: 2}]
```

当你想要 TypeScript 尽可能窄地推断你的类型时，使用 `as const`。[]{#calibre_link-655 .calibre4 primary="as const" data-type="indexterm"}[]{#calibre_link-1588 .calibre4 primary="type widening" startref="ix_typwide" data-type="indexterm"}[]{#calibre_link-1644 .calibre4 primary="types" secondary="relationships between" startref="ix_typerelwide" tertiary="type widening" data-type="indexterm"}

多余属性检查 {#excess-property-checking .calibre39}

当 TypeScript 检查一个对象类型是否可分配给另一个对象类型时，类型拓宽也会发挥作用。[]{#calibre_link-1386 .calibre4 primary="properties" secondary="excess property checking" data-type="indexterm"}[]{#calibre_link-1589 .calibre4 primary="type widening" secondary="excess property checking" data-type="indexterm"}[]{#calibre_link-988 .calibre4 primary="excess property checking" data-type="indexterm"}

回忆一下"形状和数组变性"部分，对象类型在其成员方面是协变的。但如果 TypeScript 严格遵循这个规则而不进行任何额外检查，可能会导致问题。

例如，考虑一个 `Options` 对象，你可能会将其传递给一个类来配置它：

type Options = { baseURL: string cacheSize?: number tier?: ‘prod’ | ‘dev’ }

class API { constructor(private options: Options) {} }

new API({ baseURL: ‘https://api.mysite.com’, tier: ‘prod’ })

现在，如果你拼错了一个选项会发生什么？

new API({ baseURL: ‘https://api.mysite.com’, tierr: ‘prod’ // Error TS2345: Argument of type ‘{tierr: string}’ }) // is not assignable to parameter of type ‘Options’. // Object literal may only specify known properties, // but ‘tierr’ does not exist in type ‘Options’. // Did you mean to write ‘tier’?

这是处理 JavaScript 时的常见错误，所以 TypeScript 帮助我们捕获它非常有用。[]{#calibre_link-926 .calibre4 primary="errors" secondary="TypeScript errors" tertiary="TS2345" data-type="indexterm"} 但是如果对象类型在其成员中是协变的，TypeScript 是如何捕获这个错误的呢？

也就是说：

- 我们期望类型 `{baseURL: string, cacheSize?: number, tier?: 'prod' | 'dev'}`。

- 我们传入了类型 `{baseURL: string, tierr: string}`。

- 我们传入的类型是我们期望类型的子类型，但不知何故，TypeScript 知道要报告错误。

TypeScript 能够捕获这个错误是由于其*多余属性检查*，其工作原理如下：当你尝试将一个新鲜对象字面量类型 `T` 分配给另一个类型 `U`，并且 `T` 具有 `U` 中不存在的属性时，TypeScript 会报告错误。[]{#calibre_link-1301 .calibre4 primary="object literals" secondary="fresh object literal type" data-type="indexterm"}

[]{#calibre_link-1021 .calibre4 primary="fresh object literal type" data-type="indexterm"}

*新鲜对象字面量类型*是 TypeScript 从对象字面量推断出的类型。如果该对象字面量使用了类型断言（参见["类型断言"]）或被分配给变量，那么新鲜对象字面量类型会被*拓宽*为常规对象类型，其新鲜性消失。

这个定义很密集，所以让我们再次浏览我们的示例，这次尝试更多的变体：

type Options = { baseURL: string cacheSize?: number tier?: ‘prod’ | ‘dev’ }

class API { constructor(private options: Options) {} }

new API({ baseURL: ‘https://api.mysite.com’, tier: ‘prod’ })

new API({ baseURL: ‘https://api.mysite.com’, badTier: ‘prod’ // Error TS2345: Argument of type ‘{baseURL: string; badTier: }) // string}’ is not assignable to parameter of type ‘Options’.

new API({ baseURL: ‘https://api.mysite.com’, badTier: ‘prod’ } as Options)

let badOptions = { baseURL: ‘https://api.mysite.com’, badTier: ‘prod’ } new API(badOptions)

let options: Options = { baseURL: ‘https://api.mysite.com’, badTier: ‘prod’ // Error TS2322: Type ‘{baseURL: string; badTier: string}’ } // is not assignable to type ‘Options’. new API(options)

[![1](images/000000.png)]{#calibre_link-235 .calibre4}

: 我们使用 `baseURL` 和两个可选属性之一 `tier` 实例化 `API`。这按预期工作。[]{#calibre_link-927 .calibre4 primary="errors" secondary="TypeScript errors" tertiary="TS2345" data-type="indexterm"}[]{#calibre_link-907 .calibre4 primary="errors" secondary="TypeScript errors" tertiary="TS2322" data-type="indexterm"}

[![2](images/000001.png)]{#calibre_link-236 .calibre4}

: 这里，我们将 `tier` 拼写错误为 `badTier`。我们传递给 `new API` 的选项对象

是新鲜的（因为它的类型是推断出来的，它没有被分配给变量，并且我们没有对其类型进行类型断言），所以 TypeScript 对其运行多余属性检查，揭示了多余的 `badTier` 属性（在我们的选项对象中定义但不在 `Options` 类型上）。

[![3](images/000002.png)]{#calibre_link-237 .calibre4}

: 我们断言我们的无效选项对象是 `Options` 类型。TypeScript 不再认为它是新鲜的，并退出多余属性检查：没有错误。如果你不熟悉 `as T` 语法，请阅读["类型断言"]。

[![4](images/000003.png)]{#calibre_link-238 .calibre4}

: 我们将选项对象分配给变量 `badOptions`。TypeScript 不再认为它是新鲜的，并退出多余属性检查：没有错误。

[![5](images/000004.png)]{#calibre_link-239 .calibre4}

: 当我们明确将 `options` 类型化为 `Options` 时，我们分配给 `options` 的对象是新鲜的，所以 TypeScript 执行多余属性检查，捕获我们的错误。请注意，在这种情况下，多余属性检查不是在我们将 `options` 传递给 `new API` 时发生的；而是在我们尝试将选项对象分配给变量 `options` 时发生的。

不要担心——你不需要记住这些规则。它们是 TypeScript 的内部启发式方法，用于以实用的方式捕获尽可能多的错误，以免成为你这个程序员的负担。只需在你想知道 TypeScript 是如何知道抱怨那个甚至连 Ivan（你公司代码库久经沙场的守门人和代码审查大师）都没有注意到的错误时记住它们。

类型细化 {#refinement .calibre17}

TypeScript[]{#calibre_link-1640 .calibre4 primary="types" secondary="relationships between" tertiary="refinement" data-type="indexterm"}[]{#calibre_link-1413 .calibre4 primary="refinement (types)" data-type="indexterm"}[]{#calibre_link-1006 .calibre4 primary="flow-based type inference" data-type="indexterm"} 执行基于流的类型推断，这是一种符号执行，其中类型检查器使用控制流语句如 `if`、`?`、`||` 和 `switch`，以及类型查询如 `typeof`、`instanceof` 和 `in`，在进行过程中*细化*类型，就像程序员阅读代码一样。^[1]{#calibre_link-336 .calibre16 data-type="noteref"}^ 这对类型检查器来说是一个非常方便的功能，但是另一个非常少语言支持的功能。^[2]{#calibre_link-337 .calibre16 data-type="noteref"}^

让我们通过一个例子来了解。假设我们构建了一个在 TypeScript 中定义 CSS 规则的 API，一个同事想要使用它来设置 HTML 元素的 `width`。他们传入宽度，然后我们想要解析和验证它。

我们首先实现一个函数来将 CSS 字符串解析为值和单位：

// 我们使用字符串字面量联合来描述 // CSS 单位可能具有的值 type Unit = ‘cm’ | ‘px’ | ‘%’

// 枚举单位 let units: Unit[] = [‘cm’, ‘px’, ‘%’]

// 检查每个单位，如果没有匹配则返回null function parseUnit(value: string): Unit | null {

```
for (let i = 0; i < units.length; i++) {
  if (value.endsWith(units[i])) {
    return units[i]
  }
}
return null
```

然后我们可以使用 `parseUnit` 来解析用户传递给我们的 width 值。`width` 可能是一个数字（我们假设它是像素单位），或者是一个带有单位的字符串，或者它可能是 `null` 或 `undefined`。

我们在这个例子中多次利用了类型细化(type refinement)：

```
type Width = {
  unit: Unit,
  value: number
}

function parseWidth(width: number | string | null | undefined): Width
| null {
  // 如果 width 是 null 或 undefined，提前返回
  if (width == null) {
    return null
  }

  // 如果 width 是数字，默认为像素
  if (typeof width === 'number') {
    return {unit: 'px', value: width}
```

```
  }

  // 尝试从 width 中解析单位
  let unit = parseUnit(width)
  if (unit) {
    return {unit, value: parseFloat(width)}
  }

  // 否则，返回 null
  return null
}
```

❶

TypeScript 足够智能，知道对 `null` 进行宽松相等性检查在 JavaScript 中对 `null` 和 `undefined` 都会返回 `true`。它知道如果这个检查通过，那么我们将返回；如果我们没有返回，那意味着检查没有通过，所以从那时起 `width` 的类型就是 `number | string`（它不能再是 `null` 或 `undefined` 了）。我们说该类型从 `number | string | null | undefined` 细化为 `number | string`。

❷

`typeof` 检查在运行时查询一个值以查看其类型是什么。TypeScript 在编译时也利用了 `typeof`：在检查通过的 `if` 分支中，TypeScript 知道 `width` 是一个 `number`；否则（因为那个分支有 `return`）`width` 必须是一个 `string`——这是剩下的唯一类型。

❸

因为调用 `parseUnit` 可能返回 `null`，我们通过测试其结果是否为真值来检查它是否返回了 `null`。[3] TypeScript 知道如果 `unit` 为真值，那么它在 `if` 分支中必须是 `Unit` 类型——否则，`unit` 必须为假值，意味着它必须是 `null` 类型（从 `Unit | null` 细化而来）。

❹

最后，我们返回 `null`。这只有在用户为 `width` 传递了一个 `string`，但该字符串包含了我们不支持的单位时才会发生。

我已经详细说明了 TypeScript 在这里执行的每个类型细化所思考的内容，但我希望对于你这个阅读代码的程序员来说，这已经是直观和显而易见的了。TypeScript 在捕捉你阅读和编写代码时的思路方面做得非常出色，并以类型检查和推断规则的形式将其具体化。

判别联合类型

正如我们刚刚学到的，TypeScript 对 JavaScript 的工作原理有着深入的理解，能够跟随你细化类型，就像你在脑中跟踪程序时所做的那样。

例如，假设我们正在为应用程序构建自定义事件系统。我们首先定义几种事件类型，以及处理传入事件的函数。想象一下，`UserTextEvent` 模拟键盘事件（例如，用户在文本 `<input />` 中输入了某些内容），`UserMouseEvent` 模拟鼠标事件（例如，用户将鼠标移动到坐标 `[100, 200]`）：

```
type UserTextEvent = {value: string}
type UserMouseEvent = {value: [number, number]}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
  if (typeof event.value === 'string') {
    event.value  // string
    // ...
    return
  }
  event.value    // [number, number]
}
```

在 `if` 块内，TypeScript 知道 `event.value` 必须是 `string`（因为 `typeof` 检查），这意味着在 `if` 块之后，`event.value` 必须是 `[number, number]` 元组（因为 `if` 块中的 `return`）。

如果我们让这变得更复杂一点会怎样？让我们为事件类型添加更多信息，看看当我们细化类型时 TypeScript 表现如何：

```
type UserTextEvent = {value: string, target: HTMLInputElement}
type UserMouseEvent = {value: [number, number], target: HTMLElement}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
  if (typeof event.value === 'string') {
    event.value  // string
    event.target // HTMLInputElement | HTMLElement (!!!)
    // ...
    return
  }
  event.value    // [number, number]
  event.target   // HTMLInputElement | HTMLElement (!!!)
}
```

虽然细化对 `event.value` 有效，但它没有延伸到 `event.target`。为什么？当 `handle` 接受 `UserEvent` 类型的参数时，这并不意味着我们必须传递 `UserTextEvent` 或 `UserMouseEvent`——实际上，我们可以传递 `UserMouseEvent | UserTextEvent` 类型的参数。由于联合类型的成员可能重叠，TypeScript 需要更可靠的方式来知道我们何时处于联合类型的一种情况而不是另一种情况。

做到这一点的方法是使用字面量类型来*标记*联合类型的每种情况。好的标记具有以下特点：

- 在联合类型的每种情况中位于同一位置。这意味着如果是对象类型的联合，则是同一对象字段；如果是元组类型的联合，则是同一索引。在实践中，标记联合通常使用对象类型。
- 作为字面量类型（字面量字符串、数字、布尔值等）进行类型标注。你可以混合匹配不同类型的字面量，但通常最好坚持使用单一类型；典型做法是使用字符串字面量类型。
- 非泛型。标签不应该接受任何泛型参数。
- 互斥（即在联合类型中唯一）。

基于这些原则，让我们再次更新事件类型：

```
type UserTextEvent = {type: 'TextEvent', value: string, target:
HTMLInputElement}
type UserMouseEvent = {type: 'MouseEvent', value: [number, number],
                       target: HTMLElement}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
  if (event.type === 'TextEvent') {
    event.value  // string
    event.target // HTMLInputElement
    // ...
    return
  }
  event.value    // [number, number]
  event.target   // HTMLElement
}
```

现在当我们基于标签字段的值（`event.type`）细化`event`时，TypeScript知道在`if`分支中`event`必须是`UserTextEvent`，在`if`分支之后它必须是`UserMouseEvent`。由于标签在联合类型中是唯一的，TypeScript知道这两个类型是互斥的。

在编写需要处理联合类型不同情况的函数时使用标记联合。例如，在处理Flux动作、Redux reducer或React的`useReducer`时，它们是极其宝贵的工具。

完整性

> *一个程序员睡前在床头柜上放了两个杯子：一个装满水的，以防口渴；一个空杯子，以防不渴。*
>
> *匿名*

完整性，也称为*穷尽性检查*(exhaustiveness checking)，是允许类型检查器确保你覆盖了所有情况的机制。它来源于Haskell、OCaml以及其他基于模式匹配的语言。

TypeScript会在各种情况下检查完整性，并在你遗漏情况时给出有用的警告。这是防止真实bug的极其有用的特性。例如：

```
type Weekday = 'Mon' | 'Tue'| 'Wed' | 'Thu' | 'Fri'
type Day = Weekday | 'Sat' | 'Sun'

function getNextDay(w: Weekday): Day {
  switch (w) {
    case 'Mon': return 'Tue'
  }
}
```

我们明显遗漏了几天（这周太漫长了）。TypeScript前来救援：

```
Error TS2366: Function lacks ending return statement and
return type does not include 'undefined'.
```

TSC标志： noImplicitReturns

要让TypeScript检查你所有函数的代码路径都返回一个值（如果遗漏了位置就抛出前面的警告），在你的*tsconfig.json*中启用`noImplicitReturns`标志。是否启用这个标志由你决定：有些人更喜欢较少显式`return`的代码风格，有些人则乐于为了更好的类型安全和被类型检查器捕获更多bug而接受一些额外的`return`。

这个错误消息告诉我们，要么我们遗漏了一些情况，应该在末尾用返回类似`'Sat'`的兜底`return`语句来覆盖它们（那就太好了，对吧），要么我们应该将`getNextDay`的返回类型调整为`Day | undefined`。在我们为每个`Day`添加了`case`之后，错误就消失了（试试看！）。因为我们注解了`getNextDay`的返回类型，而不是所有分支都保证返回该类型的值，所以TypeScript警告我们。

这个例子中的实现细节并不重要：无论你使用什么样的控制结构——`switch`、`if`、`throw`等等——TypeScript都会保护你，确保你覆盖了每一种情况。

这里是另一个例子：

```
function isBig(n: number) {
  if (n >= 100) {
    return true
  }
}
```

也许客户关于错过截止日期的持续语音邮件让你紧张不安，你忘记在业务关键的`isBig`函数中处理小于`100`的数字。再次，不要害怕——TypeScript在保护你：

```
Error TS7030: Not all code paths return a value.
```

或者也许周末给了你清理思路的机会，你意识到应该重写之前的`getNextDay`例子以提高效率。与其使用`switch`，为什么不在对象中进行常数时间查找呢？

```
let nextDay = {
  Mon: 'Tue'
}

nextDay.Mon // 'Tue'
```

随着你的比熊犬在另一个房间里狂吠（是因为邻居的狗？），你心不在焉地忘记在提交代码并继续做其他事情之前在新的 `nextDay` 对象中填入其他日期。

虽然下次你尝试访问 `nextDay.Tue` 时 TypeScript 会给你一个错误，但你可以在声明 `nextDay` 时更加主动地处理这个问题。有两种方法可以做到这一点，你将在[“Record 类型”]和[“映射类型”]中学到；但在我们到达那里之前，让我们稍微绕个弯，了解一下对象类型的类型运算符。

高级对象类型

对象是 JavaScript 的核心，TypeScript 为你提供了一整套安全表达和操作对象的方法。

对象类型的类型运算符

还记得我在[“联合和交集类型”]中介绍的两个类型运算符联合（`|`）和交集（`&`）吗？事实证明它们并不是 TypeScript 为你提供的唯一类型运算符！让我们来介绍更多在处理形状时很有用的类型运算符。

键入运算符

假设你有一个复杂的嵌套类型来建模从你选择的社交媒体 API 返回的 GraphQL API 响应：

```
type APIResponse = {
  user: {
    userId: string
    friendList: {
      count: number
      friends: {
        firstName: string
        lastName: string
      }[]
    }
  }
}
```

你可能从 API 获取该响应，然后渲染它：

```
function getAPIResponse(): Promise<APIResponse> {
  // ...
}

function renderFriendList(friendList: unknown) {
  // ...
}
```

```
let response = await getAPIResponse()
renderFriendList(response.user.friendList)
```

`friendList` 的类型应该是什么？（现在它被临时设为 `unknown`。）你可以把它写出来并用它来重新实现你的顶级 `APIResponse` 类型：

```
type FriendList = {
  count: number
  friends: {
    firstName: string
    lastName: string
  }[]
}

type APIResponse = {
  user: {
    userId: string
    friendList: FriendList
  }
}

function renderFriendList(friendList: FriendList) {
  // ...
}
```

但是这样你就必须为每个顶级类型想出名字，而你并不总是想要这样做（例如，如果你使用构建工具从 GraphQL 模式生成 TypeScript 类型）。相反，你可以*键入*到你的类型中：

```
type APIResponse = {
  user: {
    userId: string
    friendList: {
      count: number
      friends: {
        firstName: string
        lastName: string
      }[]
    }
```

```
  }
}

type FriendList = APIResponse['user']['friendList']

function renderFriendList(friendList: FriendList) {
  // ...
}
```

你可以键入任何形状（对象、类构造函数或类实例）和任何数组。例如，要获取单个朋友的类型：

```
type Friend = FriendList['friends'][number]
```

`number` 是键入数组类型的一种方式；对于元组，使用 `0`、`1` 或其他数字字面量类型来表示你想键入的索引。

键入的语法故意类似于你在常规 JavaScript 对象中查找字段的方式——就像你可能在对象中查找值一样，你也可以在形状中查找类型。注意，在键入时查找属性类型必须使用方括号表示法，而不是点表示法。

keyof 运算符

使用 `keyof` 来获取对象的所有键作为字符串字面量类型的联合。使用前面的 `APIResponse` 示例：

```
type ResponseKeys = keyof APIResponse // 'user'
type UserKeys = keyof APIResponse['user'] // 'userId' | 'friendList'
type FriendListKeys =
  keyof APIResponse['user']['friendList'] // 'count' | 'friends'
```

结合键入和 `keyof` 运算符，你可以实现一个类型安全的 getter 函数，它在对象中根据给定的键查找值：

```
function get<
  O extends object,
  K extends keyof O
>(
  o: O,
  k: K
): O[K] {
  return o[k]
}
```

❶

`get` 是一个接受对象 `o` 和键 `k` 的函数。

❷

`keyof O` 是一个字符串字面量类型的联合，代表 `o` 的所有键。泛型类型 `K` 扩展——并且是该联合的子类型。例如，如果 `o` 的类型是 `{a: number, b: string, c: boolean}`，那么 `keyof o` 就是类型 `'a' | 'b' | 'c'`，而 `K`（扩展自 `keyof o`）可以是类型 `'a'`、`'b'`、`'a' | 'c'` 或 `keyof o` 的任何其他子类型。

❸

`O[K]` 是当你在 `O` 中查找 `K` 时得到的类型。延续 [

❷

] 的例子，如果 `K` 是 `'a'`，那么我们在编译时就知道 `get` 返回一个 `number`。或者，如果 `K` 是 `'b' | 'c'`，那么我们知道 `get` 返回 `string | boolean`。

这些类型操作符的酷炫之处在于它们如何精确且安全地让你描述形状类型：

```
type ActivityLog = {
  lastEvent: Date
  events: {
    id: string
    timestamp: Date
```

```
    type: 'Read' | 'Write'
  }[]
}

let activityLog: ActivityLog = // ...
let lastEvent = get(activityLog, 'lastEvent') // Date
```

TypeScript 为你工作，*在编译时*验证 `lastEvent` 的类型是 `Date`。当然，你也可以扩展它以便更深入地索引对象。让我们重载 `get` 来接受最多三个键：

```
type Get = {
  <
    O extends object,
    K1 extends keyof O
  >(o: O, k1: K1): O[K1]
  <
    O extends object,
    K1 extends keyof O,
    K2 extends keyof O[K1]
  >(o: O, k1: K1, k2: K2): O[K1][K2]
  <
    O extends object,
    K1 extends keyof O,
    K2 extends keyof O[K1],
    K3 extends keyof O[K1][K2]
  >(o: O, k1: K1, k2: K2, k3: K3): O[K1][K2][K3]
}

let get: Get = (object: any, ...keys: string[]) => {
  let result = object
  keys.forEach(k => result = result[k])
  return result
}

get(activityLog, 'events', 0, 'type') // 'Read' | 'Write'

get(activityLog, 'bad') // Error TS2345: Argument of type '"bad"'
                        // is not assignable to parameter of type
                        // '"lastEvent" | "events"'.
```

❶

我们为 `get` 声明了一个重载函数签名，包含三种情况：当我们用一个键、两个键和三个键调用 `get` 时。

❷

这个单键情况与上一个例子相同：`O` 是 `object` 的子类型，`K1` 是该对象键的子类型，返回类型是当你用 `K1` 索引 `O` 时得到的特定类型。

❸

双键情况类似于单键情况，但我们声明了另一个泛型类型 `K2`，来建模嵌套对象上可能的键，这个嵌套对象是用 `K1` 索引 `O` 的结果。

❹

我们基于 [

❷

] 进行两次索引——我们首先得到 `O[K1]` 的类型，然后在结果上得到 `[K2]` 的类型。

❺

在这个例子中我们处理最多三个嵌套键；如果你正在编写一个实际的库，你可能会想要处理更多的情况。

很酷，对吧？如果你有时间，把这个例子展示给你的 Java 朋友们，确保在你向他们解释的时候得意地炫耀一下。

TSC 标志：keyofStringsOnly

在 JavaScript 中，对象和数组可以同时拥有字符串和符号键。按照惯例，我们通常为数组使用数字键，这些键在运行时被强制转换为字符串。

因此，TypeScript 中的 `keyof` 默认返回 `number | string | symbol` 类型的值（不过如果你在一个更具体的形状上调用它，TypeScript 可以推断出该联合类型的更具体的子类型）。

这种行为是正确的，但可能使 `keyof` 的使用变得冗长，因为你可能必须向 TypeScript 证明你正在操作的特定键是 `string`，而不是 `number` 或 `symbol`。

要选择使用 TypeScript 的传统行为——键必须是字符串——请在 *tsconfig.json* 中启用 `keyofStringsOnly` 标志。

Record 类型

TypeScript 内置的 `Record` 类型是一种将对象描述为从某物到某物的映射的方式。

回想一下[“完整性”]中 `Weekday` 例子，有两种方法来强制对象定义一组特定的键。`Record` 类型是第一种。

让我们使用 `Record` 构建一个从一周中每一天到下一天的映射。使用 `Record`，你可以对 `nextDay` 中的键和值施加一些约束：

```
type Weekday = 'Mon' | 'Tue'| 'Wed' | 'Thu' | 'Fri'
type Day = Weekday | 'Sat' | 'Sun'

let nextDay: Record<Weekday, Day> = {
  Mon: 'Tue'
}
```

现在，你立即得到一个友好、有用的错误消息：

```
Error TS2739: Type '{Mon: "Tue"}' is missing the following properties
from type 'Record<Weekday, Day>': Tue, Wed, Thu, Fri.
```

当然，向你的对象添加缺失的 `Weekday` 会让错误消失。

`Record` 与常规的对象索引签名相比，为你提供了一个额外的自由度：使用常规索引签名，你可以约束对象值的类型，但键只能是常规的 `string`、`number` 或 `symbol`；而使用 `Record`，你还可以将对象键的类型约束为 `string` 和 `number` 的子类型。

映射类型

TypeScript 为我们提供了第二种更强大的方式来声明更安全的 `nextDay` 类型：映射类型 (mapped types)。让我们使用映射类型来表示 `nextDay` 是一个对象，它为每个 `Weekday` 都有一个键，其值是 `Day`：

```
let nextDay: {[K in Weekday]: Day} = {
  Mon: 'Tue'
}
```

这是另一种获得如何修复遗漏内容有用提示的方法：

```
Error TS2739: Type '{Mon: "Tue"}' is missing the following properties
from type '{Mon: Weekday; Tue: Weekday; Wed: Weekday; Thu: Weekday;
Fri: Weekday}': Tue, Wed, Thu, Fri.
```

映射类型是 TypeScript 独有的语言特性。像字面量类型一样，它们是一个实用特性，对于静态类型化 JavaScript 的挑战来说非常合理。

如你所见，映射类型有自己特殊的语法。像索引签名一样，每个对象最多只能有一个映射类型：

```
type MyMappedType = {
  [Key in UnionType]: ValueType
}
```

顾名思义，这是一种映射对象键和值类型的方法。实际上，TypeScript 使用映射类型来实现我们之前使用的内置 `Record` 类型：

```
type Record<K extends keyof any, T> = {
  [P in K]: T
```

```
}
```

映射类型比单纯的 `Record` 给你更多的力量，因为除了让你给对象的键和值赋予类型之外，当你将它们与键入类型(keyed-in types)结合使用时，它们还允许你对哪个值类型对应哪个键名施加约束。

让我们快速浏览一下你可以用映射类型做的一些事情。

```
type Account = {
  id: number
  isEmployee: boolean
  notes: string[]
}

// 使所有字段可选
type OptionalAccount = {
  [K in keyof Account]?: Account[K]
}

// 使所有字段可空
type NullableAccount = {
  [K in keyof Account]: Account[K] | null
}

// 使所有字段只读
type ReadonlyAccount = {
  readonly [K in keyof Account]: Account[K]
}

// 再次使所有字段可写（等同于 Account）
type Account2 = {
  -readonly [K in keyof ReadonlyAccount]: Account[K]
}

// 再次使所有字段必需（等同于 Account）
type Account3 = {
  [K in keyof OptionalAccount]-?: Account[K]
}
```

❶

我们通过映射 `Account` 创建一个新的对象类型 `OptionalAccount`，在此过程中将每个字段标记为可选。

❷

我们通过映射 `Account` 创建一个新的对象类型 `NullableAccount`，在此过程中为每个字段添加 `null` 作为可能的值。

❸

我们通过获取 `Account` 并使其每个字段都只读（即可读但不可写）来创建一个新的对象类型 `ReadonlyAccount`。

❹

我们可以将字段标记为可选（`?`）或 `readonly`，我们也可以取消标记它们。使用减号（`-`）操作符——一个只在映射类型中可用的特殊类型操作符——我们可以撤销 `?` 和 `readonly`，分别使字段再次必需和可写。这里我们通过映射 `ReadonlyAccount` 并使用减号（`-`）操作符移除 `readonly` 修饰符来创建一个新的对象类型 `Account2`，等同于我们的 `Account` 类型。

❺

我们通过映射 `OptionalAccount` 并使用减号（`-`）操作符移除可选（`?`）操作符来创建一个新的对象类型 `Account3`，等同于我们原来的 `Account` 类型。

注意

减号（`-`）有一个对应的加号（`+`）类型操作符。你可能永远不会直接使用这个操作符，因为它是隐含的：在映射类型中，`readonly` 等同于 `+readonly`，`?` 等同于 `+?`。`+` 只是为了完整性而存在。

内置映射类型

我们在上一节中派生的映射类型非常有用，以至于 TypeScript 内置了许多这样的类型：

`Record<Keys, Values>`
一个具有 `Keys` 类型的键和 `Values` 类型的值的对象

`Partial<Object>`
将 `Object` 中的每个字段标记为可选

`Required<Object>`
将 `Object` 中的每个字段标记为非可选

`Readonly<Object>`
将 `Object` 中的每个字段标记为只读

`Pick<Object, Keys>`
返回 `Object` 的子类型，只包含给定的 `Keys`

伴生对象模式(Companion Object Pattern)

伴侣对象模式来自 Scala，是一种将共享相同名称的对象和类配对在一起的方法。在 TypeScript 中，有一个类似且同样有用的模式——我们也称之为伴侣对象模式——我们可以用它来配对类型和对象。

它看起来像这样：

```
type Currency = {
  unit: 'EUR' | 'GBP' | 'JPY' | 'USD'
  value: number
}

let Currency = {
  DEFAULT: 'USD',
  from(value: number, unit = Currency.DEFAULT): Currency {
    return {unit, value}
  }
}
```

请记住，在 TypeScript 中，类型和值存在于不同的命名空间中；你将在[“声明合并”]中进一步了解这一点。这意味着在同一作用域中，你可以将相同的名称（在这个例子中是 `Currency`）同时绑定到类型和值。通过伴侣对象模式，我们利用这种分离的命名空间来声明一个名称两次：首先作为类型，然后作为值。

这种模式有一些很好的特性。它让你将在语义上属于单个名称（如 `Currency`）的类型和值信息组合在一起。它还让使用者可以同时导入两者：

```
import {Currency} from './Currency'

let amountDue: Currency = {
  unit: 'JPY',
  value: 83733.10
}
```

```
let otherAmountDue = Currency.from(330, 'EUR')
```

❶

将 `Currency` 用作类型

❷

将 `Currency` 用作值

当类型和对象在语义上相关，且对象提供操作该类型的实用方法时，使用伴侣对象模式。

高级函数类型

让我们看看一些经常与函数类型一起使用的更高级技术。

改进元组的类型推断

当你在 TypeScript 中声明元组时，TypeScript 在推断该元组的类型时会比较宽松。它将基于你提供的内容推断最通用的可能类型，忽略元组的长度以及哪个位置保存哪种类型：

```
let a = [1, true] // (number | boolean)[]
```

但有时你想要更严格的推断，将 `a` 视为固定长度的元组而不是数组。当然，你可以使用类型断言将元组转换为元组类型（更多内容见[“类型断言”]）。或者，你可以使用 `as const` 断言（[“const 类型”]）来尽可能窄地推断元组的类型，将其标记为只读。

如果你想将元组类型化为元组，但避免类型断言，并避免 `as const` 给你的窄推断和只读修饰符怎么办？为此，你可以利用 TypeScript 推断剩余参数类型的方式（跳回[“使用有界多态性来建模元数”]了解更多内容）：

```
function tuple<
  T extends unknown[]
>(
  ...ts: T
): T {
  return ts
}

let a = tuple(1, true) // [number, boolean]
```

❶

我们声明一个 `tuple` 函数，用于构造元组类型（而不是使用内置的 `[]` 语法）。

❷

我们声明一个单一类型参数 `T`，它是 `unknown[]` 的子类型（意味着 `T` 是任何类型的数组）。

❸

`tuple` 接受可变数量的参数 `ts`。由于 `T` 描述了一个剩余参数，TypeScript 将为其推断元组类型。

❹

`tuple` 返回与推断 `ts` 相同的元组类型的值。

❺

我们的函数返回与传递给它的参数相同的参数。魔法全在类型中。
当你的代码使用大量元组类型时，利用这种技术来避免类型断言。

用户定义的类型守卫

对于某些返回 `boolean` 的函数类型，仅仅说你的函数返回一个 `boolean` 可能是不够的。例如，让我们编写一个函数来告诉你是否传递了一个 `string`：

```
function isString(a: unknown): boolean {
  return typeof a === 'string'
}

isString('a') // 计算结果为 true
isString([7]) // 计算结果为 false
```

到目前为止一切正常。如果你尝试在一些真实代码中使用 `isString` 会发生什么？

```
function parseInput(input: string | number) {
  let formattedInput: string
  if (isString(input)) {
    formattedInput = input.toUpperCase() // 错误 TS2339: 属性
'toUpperCase'
  }                                      // 在类型 'number' 上不存在。
}
```

这是怎么回事？如果 `typeof` 对常规类型细化有效（参见”细化”一节），为什么在这里不起作用？

类型细化的问题在于它只在你所在的作用域内强大到足以细化变量的类型。一旦你离开那个作用域，细化就不会延续到你进入的新作用域中。在我们的 `isString` 实现中，我们使用 `typeof` 将输入参数的类型细化为 `string`，但因为类型细化不会延续到新作用域，它丢失了——TypeScript 只知道 `isString` 返回了一个 `boolean`。

我们可以做的是告诉类型检查器，`isString` 不仅返回一个 `boolean`，而且每当该 `boolean` 为 `true` 时，我们传递给 `isString` 的参数就是一个 `string`。为了做到这一点，我们使用一种叫做*用户定义类型守卫*的东西：

```
function isString(a: unknown): a is string {
  return typeof a === 'string'
}
```

类型守卫是 TypeScript 的内置功能，它让你可以使用 `typeof` 和 `instanceof` 来细化类型。但有时，你需要能够自己声明类型守卫——这就是 `is` 操作符的作用。当你有一个细化其参数类型并返回 `boolean` 的函数时，你可以使用用户定义的类型守卫来确保每当你使用该函数时，该细化都会流转。

用户定义的类型守卫仅限于单个参数，但它们不限于简单类型：

```
type LegacyDialog = // ...
type Dialog = // ...

function isLegacyDialog(
  dialog: LegacyDialog | Dialog
): dialog is LegacyDialog {
  // ...
}
```

你不会经常使用用户定义的类型守卫，但当你使用时，它们对于编写清晰、可重用的代码非常棒。如果没有它们，你就必须内联所有的 `typeof` 和 `instanceof` 类型守卫，而不是构建像 `isLegacyDialog` 和 `isString` 这样的函数来以更好封装、更可读的方式执行相同的检查。

条件类型

条件类型可能是整个 TypeScript 中最独特的功能。在高层次上，条件类型让你可以说，“声明一个依赖于类型 `U` 和 `V` 的类型 `T`；如果 `U <: V`，那么将 `T` 分配给 `A`，否则将 `T` 分配给 `B`。”

在代码中它可能看起来像这样：

```
type IsString<T> = T extends string
  ? true
  : false

type A = IsString<string> // true
type B = IsString<number> // false
```

让我们逐行分析。

❶

我们声明一个新的条件类型 `IsString`，它接受一个泛型类型 `T`。这个条件类型的”条件”部分是 `T extends string`；也就是说，“`T` 是 `string` 的子类型吗？”

❷

如果 `T` 是 `string` 的子类型，我们解析为类型 `true`。

❸

否则，我们解析为类型 `false`。

注意语法看起来就像一个常规的值级三元表达式，但在类型级别。就像常规三元表达式一样，你也可以嵌套它们。

条件类型不限于类型别名。你可以在几乎任何可以使用类型的地方使用它们：在类型别名、接口、类、参数类型以及函数和方法中的泛型默认值。

分布式条件

虽然你可以在TypeScript中用多种方式来表达简单的条件——通过条件类型、重载函数签名和映射类型——条件类型让你能做更多事情。原因在于它们遵循*分配律*(distributive law)（还记得代数课吗？）。这意味着如果你有一个条件类型，那么右边的表达式与表1中左边的表达式是等价的。

表6-1. 条件类型的分配

这样写...	等价于
`string extends T ? A : B` `string exte	nds T ? A : B`
`(string | number) extends T ? A : B` `(string ext	ends T ? A : B) | (number extends T ? A : B)`
`(string | number | boolean) extends T ? A : B` `(string ext	ends T ? A : B) | (number extends T ? A : B) | (boolean extends T ? A : B)`

我知道，我知道，你买这本书不是来学数学的——你是来学类型的。那我们来看更具体的例子。假设我们有一个函数，它接受某个类型为`T`的变量，并将其提升为类型`T[]`的数组。如果我们为`T`传入一个联合类型会发生什么？

```
type ToArray<T> = T[]
type A = ToArray<number>          // number[]
type B = ToArray<number | string> // (number | string)[]
```

很直观。现在如果我们添加一个条件类型会发生什么？（注意这里的条件实际上没有做任何事情，因为它的两个分支都解析为相同的类型`T[]`；它只是告诉TypeScript要*分配*`T`到元组类型上。）看看这个：

```
type ToArray2<T> = T extends unknown ? T[] : T[]
type A = ToArray2<number> // number[]
```

```
type B = ToArray2<number | string> // number[] | string[]
```

你注意到了吗？当你使用条件类型时，TypeScript会将联合类型分配到条件的分支上。这就像获取条件类型并将其映射（呃，*分配*）到联合中的每个元素上。

为什么这很重要？它让你能安全地表达一系列常见操作。

例如，TypeScript带有`&`用于计算两个类型的共同点，`|`用于取两个类型的联合。让我们构建`Without<T, U>`，它计算在`T`中但不在`U`中的类型。

```
type Without<T, U> = T extends U ? never : T
```

你这样使用`Without`：

```
type A = Without<
  boolean | number | string,
  boolean
> // number | string
```

让我们逐步看看TypeScript如何计算这个类型：

1. 从输入开始：

   ```
   type A = Without<boolean | number | string, boolean>
   ```

2. 将条件分配到联合上：

   ```
   type A = Without<boolean, boolean>
          | Without<number, boolean>
          | Without<string, boolean>
   ```

3. 代入`Without`的定义并应用`T`和`U`：

```
type A = (boolean extends boolean ? never : boolean)
       | (number extends boolean ? never : number)
       | (string extends boolean ? never : string)
```

4. 评估条件：

```
type A = never
       | number
       | string
```

5. 简化：

```
type A = number | string
```

如果不是因为条件类型的分配性质，我们最终会得到`never`（如果你不确定为什么，自己演算一遍看看！）。

infer关键字

条件类型的最后一个特性是能够将泛型类型声明为条件的一部分。作为复习，到目前为止我们只看到了一种声明泛型类型参数的方式：使用尖括号（`<T>`）。条件类型有自己的语法来内联声明泛型类型：`infer`关键字。

让我们声明一个条件类型`ElementType`，它获取数组元素的类型：

```
type ElementType<T> = T extends unknown[] ? T[number] : T
type A = ElementType<number[]> // number
```

现在，让我们使用`infer`重写它：

```
type ElementType2<T> = T extends (infer U)[] ? U : T
type B = ElementType2<number[]> // number
```

在这个简单的例子中，`ElementType`等价于`ElementType2`。注意`infer`子句如何声明一个新的类型变量`U`——TypeScript会根据上下文推断`U`的类型，基于你传递给`ElementType2`的`T`。

还要注意为什么我们内联声明`U`而不是预先声明它，与`T`一起。如果我们确实预先声明它会发生什么？

```
type ElementUgly<T, U> = T extends U[] ? U : T
type C = ElementUgly<number[]> // Error TS2314: Generic type
'ElementUgly'
                               // requires 2 type argument(s).
```

糟糕。因为`ElementUgly`定义了两个泛型类型`T`和`U`，当实例化`ElementUgly`时，我们必须传入这两个泛型参数。但如果我们这样做，这就违背了拥有`ElementUgly`类型的初衷；它把计算`U`的负担推给了调用者，而我们原本希望`ElementUgly`能自己计算这个类型。

老实说，这是一个有些愚蠢的例子，因为我们已经有了键入操作符（`[]`）来查找数组元素的类型。那么来看一个更复杂的例子?

```
type SecondArg<F> = F extends (a: any, b: infer B) => any ? B : never

// 获取 Array.slice 的类型
type F = typeof Array['prototype']['slice']

type A = SecondArg<F> // number | undefined
```

所以，`[].slice` 的第二个参数是 `number | undefined`。而且我们在编译时就知道这一点——试试在 Java 中做到这一点。

内置条件类型

条件类型(conditional types)让你能在类型层面表达一些非常强大的操作。这就是为什么TypeScript内置了一些全局可用的条件类型：

`Exclude<T, U>`
就像我们之前的 `Without` 类型一样，计算 `T` 中不在 `U` 中的类型：

```
type A = number | string
type B = string
type C = Exclude<A, B>  // number
```

`Extract<T, U>`
计算 `T` 中可以赋值给 `U` 的类型：

```
type A = number | string
type B = string
type C = Extract<A, B>  // string
```

`NonNullable<T>`
计算排除 `null` 和 `undefined` 的 `T` 版本：

```
type A = {a?: number | null}
type B = NonNullable<A['a']>  // number
```

`ReturnType<F>`
计算函数的返回类型（注意这对泛型和重载函数不会按预期工作）：

```
type F = (a: number) => string
type R = ReturnType<F>  // string
```

`InstanceType<C>`

计算类构造函数的实例类型：

```
type A = {new(): B}
type B = {b: number}
type I = InstanceType<A>  // {b: number}
```

逃逸舱口(Escape Hatches)

有时你没有时间把某些东西完美地类型化，你只是希望 TypeScript 相信你正在做的事情是安全的。也许你正在使用的第三方模块的类型声明是错误的，你想在将修复贡献回 DefinitelyTyped 之前测试你的代码，或者也许你正在从 API 获取数据，但还没有用 Apollo 重新生成类型声明。

幸运的是，TypeScript 知道我们只是人类，当我们只是想做某件事而没有时间向 TypeScript 证明它是安全的时候，它给了我们一些逃逸舱口。

注意

如果还不明显的话，你应该尽可能少地使用以下 TypeScript 特性。如果你发现自己依赖于它们，你可能做错了什么。

类型断言(Type Assertions)

如果你有一个类型 `B` 且 `A <: B <: C`，那么你可以向类型检查器断言 `B` 实际上是一个 `A` 或一个 `C`。值得注意的是，你只能断言一个类型是其自身的超类型或子类型——例如，你不能断言一个 `number` 是一个 `string`，因为这些类型没有关系。

TypeScript 为类型断言提供了两种语法：

```
function formatInput(input: string) {
  // ...
}

function getUserInput(): string | number {
  // ...
}

let input = getUserInput()

// 断言 input 是一个 string
formatInput(input as string)

// 这等价于
formatInput(<string>input)
```

❶

我们使用类型断言（`as`）告诉 TypeScript `input` 是一个 `string`，而不是类型所显示的 `string | number`。例如，如果你想快速测试你的 `formatInput` 函数，并且你确定对于你的测试，`getUserInput` 返回一个 `string`，你可能会这样做。

❷

类型断言的传统语法使用尖括号。这两种语法在功能上是等价的。

注意

优先使用 `as` 语法进行类型断言，而不是尖括号（`<>`）语法。前者是明确的，但后者可能与 TSX 语法冲突（参见[“TSX = JSX + TypeScript”]）。使用 TSLint 的 `no-angle-bracket-type-assertion` 规则来自动为您的代码库强制执行这一点。

有时，两种类型可能关联性不够充分，所以您不能断言一种是另一种。要解决这个问题，只需断言为 `any`（记住从[“可赋值性”]中学到的，`any` 可赋值给任何东西），然后在角落里花几分钟思考您所做的事情：

```
function addToList(list: string[], item: string) {
  // ...
}

addToList('this is really,' as any, 'really unsafe')
```

显然，类型断言是不安全的，您应该尽可能避免使用它们。

非空断言

对于可空类型的特殊情况——即类型为 `T | null` 或 `T | null | undefined`——TypeScript 有特殊的语法来断言该类型的值是 `T`，而不是 `null` 或 `undefined`。这在几个地方会出现。

例如，假设我们编写了一个在 Web 应用程序中显示和隐藏对话框的框架。每个对话框都有一个唯一的 ID，我们使用它来获取对话框 DOM 节点的引用。一旦对话框从 DOM 中移除，我们就删除其 ID，表示它不再在 DOM 中活跃：

```
type Dialog = {
  id?: string
}

function closeDialog(dialog: Dialog) {
  if (!dialog.id) {
    return
  }
  setTimeout(() =>
    removeFromDOM(
      dialog,
      document.getElementById(dialog.id) // Error TS2345: Argument of
type
                                         // 'string | undefined' is
not assignable
                                         // to parameter of type
'string'.
    )
  )
}

function removeFromDOM(dialog: Dialog, element: Element) {
  element.parentNode.removeChild(element) // Error TS2531: Object is
possibly
                                          //'null'.
  delete dialog.id
}
```

❶

如果对话框已经被删除（所以它没有 `id`），我们提前返回。

❷

我们在事件循环的下一轮从 DOM 中移除对话框，这样依赖于 `dialog` 的任何其他代码都有机会完成运行。

❸

因为我们在箭头函数内部，现在我们处于一个新的作用域中。TypeScript 不知道是否有代码在 [

❶

] 和 [

❸

] 之间修改了 `dialog`，所以它使我们在 [

❶

] 中所做的类型缩窄失效。除此之外，虽然我们知道如果 `dialog.id` 被定义，那么具有该 ID 的元素肯定存在于 DOM 中（因为我们这样设计了我们的框架），但 TypeScript 所知道的只是调用 `document.getElementById` 返回 `HTMLElement | null`。我们知道它总是非空的 `HTMLElement`，但 TypeScript 不知道——它只知道我们给它的类型。

❹

同样，虽然我们知道对话框肯定在 DOM 中，并且它肯定有一个父 DOM 节点，但 TypeScript 所知道的只是 `element.parentNode` 的类型是 `Node | null`。

解决这个问题的一种方法是到处添加一堆 `if (_ === null)` 检查。虽然如果您不确定某个东西是否为 `null`，这是正确的做法，但 TypeScript 为您确定它不是 `null | undefined` 的情况提供了特殊的语法：

```
type Dialog = {
  id?: string
}

function closeDialog(dialog: Dialog) {
  if (!dialog.id) {
    return
  }
  setTimeout(() =>
    removeFromDOM(
      dialog,
      document.getElementById(dialog.id!)!
    )
  )
}

function removeFromDOM(dialog: Dialog, element: Element) {
  element.parentNode!.removeChild(element)
  delete dialog.id
}
```

注意点缀的非空断言操作符（`!`），它告诉 TypeScript 我们确定 `dialog.id`、我们的 `document.getElementById` 调用的结果和 `element.parentNode` 都被定义了。当非空断言跟随一个可能为 `null` 或 `undefined` 的类型时，TypeScript 将假设该类型已定义：`T | null | undefined` 变成 `T`，`number | string | null` 变成 `number | string`，等等。

当你发现自己大量使用非null断言时，这通常表明你应该重构代码。例如，我们可以通过将 `Dialog` 拆分为两种类型的联合来摆脱断言：

```
type VisibleDialog = {id: string}
type DestroyedDialog = {}
type Dialog = VisibleDialog | DestroyedDialog
```

然后我们可以更新 `closeDialog` 来利用联合类型：

```
function closeDialog(dialog: Dialog) {
  if (!('id' in dialog)) {
    return
  }
  setTimeout(() =>
    removeFromDOM(
      dialog,
      document.getElementById(dialog.id)!
    )
  )
}

function removeFromDOM(dialog: VisibleDialog, element: Element) {
  element.parentNode!.removeChild(element)
  delete dialog.id
}
```

在我们检查 `dialog` 是否定义了 `id` 属性之后——这意味着它是一个 `Visible[Dialog]`——即使在箭头函数内部，TypeScript 也知道对 `dialog` 的引用没有改变：箭头函数内部的 `dialog` 与函数外部的 `dialog` 是同一个，所以类型收窄会延续，而不会像上个例子那样被失效。

明确赋值断言

TypeScript 为非null断言的特殊情况提供了特殊语法，用于明确赋值检查（提醒一下，明确赋值检查是 TypeScript 确保在使用变量时该变量已被赋值的方式）。例如：

```
let userId: string

userId.toUpperCase() // Error TS2454: Variable 'userId' is used
                     // before being assigned.
```

显然，TypeScript 通过捕获这个错误为我们提供了很大的帮助。我们声明了变量 `userId`，但在尝试将其转换为大写之前忘记为其赋值。如果 TypeScript 没有注意到这一点，这将是一个运行时错误！

但是，如果我们的代码看起来更像这样呢？

```
let userId: string
fetchUser()

userId.toUpperCase() // Error TS2454: Variable 'userId' is used
                     // before being assigned.

function fetchUser() {
  userId = globalCache.get('userId')
}
```

我们恰好拥有世界上最好的缓存，当我们查询这个缓存时，100% 的时间都能命中缓存。所以在调用 `fetchUser` 之后，`userId` 保证会被定义。但是 TypeScript 无法静态检测到这一点，所以它仍然抛出与之前相同的错误。我们可以使用明确赋值断言来告诉 TypeScript，在读取时 `userId` 肯定会被赋值（注意感叹号）：

```
let userId!: string
fetchUser()
```

```
userId.toUpperCase() // OK

function fetchUser() {
  userId = globalCache.get('userId')
}
```

与类型断言和非null断言一样，如果你发现自己经常使用明确赋值断言，那么你可能做错了什么。

模拟名义类型

在这本书的这个阶段，如果我在凌晨三点把你摇醒并大喊”TYPESCRIPT 的类型系统是结构化的还是名义的？！“你会大声回答”当然是结构化的！现在离开我的房子，否则我就报警！“这对我闯入进行早晨类型系统问题来说是一个公平的回应。

撇开法律不谈，现实是有时名义类型确实很有用。例如，假设你的应用程序中有几种 `ID` 类型，代表在系统中寻址不同类型对象的唯一方式：

```
type CompanyID = string
type OrderID = string
type UserID = string
type ID = CompanyID | OrderID | UserID
```

`UserID` 类型的值可能是一个看起来像 `"d21b1dbf"` 的简单哈希。虽然你可能将其别名为 `UserID`，但在底层它当然只是一个普通的 `string`。接受 `UserID` 的函数可能看起来像这样：

```
function queryForUser(id: UserID) {
  // ...
}
```

这是很好的文档，它帮助团队中的其他工程师确切知道应该传入哪种类型的 `ID`。但由于 `UserID` 只是 `string` 的别名，这种方法在预防错误方面作用有限。工程师可能意外传入错误类型的 `ID`，而类型系统将毫不知情！

```
let id: CompanyID = 'b4843361'
queryForUser(id) // OK (!!!)
```

这就是*名义类型*派上用场的地方。虽然 TypeScript 不原生支持名义类型，但我们可以用一种称为*类型标记*的技术来模拟它们。类型标记需要一些设置工作，在 TypeScript 中使用它的体

验不如在内置支持名义类型别名的语言中那么流畅。话虽如此，品牌类型可以让你的程序显著更安全。

注意

根据您的应用程序和工程团队的规模（团队越大，这种技术越有可能在防止错误方面发挥作用），您可能不需要这样做。

首先为每个名义类型创建一个合成的*类型品牌*：

```
type CompanyID = string & {readonly brand: unique symbol}
type OrderID = string & {readonly brand: unique symbol}
type UserID = string & {readonly brand: unique symbol}
type ID = CompanyID | OrderID | UserID
```

`string` 和 `{readonly brand: unique symbol}` 的交集当然是无意义的。我选择它是因为无法自然地构造该类型，创建该类型值的唯一方法是使用断言。这是品牌类型的关键特性：它们使得意外使用错误类型变得困难。我使用 `unique symbol` 作为”品牌”，因为它是 TypeScript 中真正名义类型的两种之一（另一种是 `enum`）；我将该品牌与 `string` 取交集，以便我们可以断言给定的 `string` 是给定的品牌类型。

现在我们需要一种方法来创建 `CompanyID`、`OrderID` 和 `UserID` 类型的值。为此，我们将使用伴生对象模式（在[“伴生对象模式”]中介绍）。我们将为每种品牌类型创建一个构造函数，使用类型断言来构造我们无意义类型的值：

```
function CompanyID(id: string) {
  return id as CompanyID
}

function OrderID(id: string) {
  return id as OrderID
}

function UserID(id: string) {
  return id as UserID
}
```

最后，让我们看看使用这些类型的感觉如何：

```
function queryForUser(id: UserID) {
  // ...
}

let companyId = CompanyID('8a6076cf')
let orderId = OrderID('9994acc1')
let userId = UserID('d21b1dbf')

queryForUser(userId)    // OK
queryForUser(companyId) // Error TS2345: Argument of type 'CompanyID'
is not
                        // assignable to parameter of type 'UserID'.
```

这种方法的好处是运行时开销很少：每个 `ID` 构造只需要一个函数调用，无论如何，您的 JavaScript 虚拟机可能会内联它。在运行时，每个 `ID` 只是一个 `string`——品牌纯粹是编译时构造。

同样，对于大多数应用程序，这种方法是过度的。但对于大型应用程序，以及在处理容易混淆的类型（如不同种类的 ID）时，品牌类型可能是一个杀手级的安全功能。

安全地扩展原型

在构建 JavaScript 应用程序时，传统认为扩展内置类型的原型是不安全的。这个经验法则可以追溯到 jQuery 之前的时代，当时明智的 JavaScript 法师构建了像 MooTools 这样的库，直接扩展和覆盖内置原型方法。但当太多法师同时增强原型时，就会产生冲突。没有静态类型系统，您只能在运行时从愤怒的用户那里发现这些冲突。

如果您不是来自 JavaScript，您可能会惊讶地了解到，在 JavaScript 中，您可以在运行时修改任何内置方法（如 `[].push`、`'abc'.toUpperCase` 或 `Object.assign`）。因为它是如此动态的语言，JavaScript 为您提供了对每个内置对象原型的直接访问——`Array.prototype`、`Function.prototype`、`Object.prototype` 等等。

虽然在过去扩展这些原型是不安全的，但如果您的代码由像 TypeScript 这样的静态类型系统覆盖，那么现在您可以安全地这样做。

例如，我们将向 `Array` 原型添加一个 `zip` 方法。安全扩展原型需要两件事。首先，在一个 *.ts* 文件（比如，*zip.ts*）中，我们扩展 `Array` 原型的类型；然后，我们用新的 `zip` 方法增强原型：

```
// 告诉 TypeScript 关于 .zip
interface Array<T> {
  zip<U>(list: U[]): [T, U][]
}

// 实现 .zip
Array.prototype.zip = function<T, U>(
  this: T[],
  list: U[]
): [T, U][] {
  return this.map((v, k) =>
    tuple(v, list[k])
  )
}
```

❶

我们首先告诉 TypeScript 我们正在向 `Array` 添加 `zip`。我们利用接口合并（[“声明合并”]）来增强全局 `Array<T>` 接口，将我们自己的 `zip` 方法添加到已经全局定义的接口中。

由于我们的文件没有任何显式的导入或导出——这意味着它处于脚本模式，如[“模块模式与脚本模式”]中所述——我们能够通过声明一个与现有 `Array<T>` 接口完全相同名称的接口来直接增强全局 `Array` 接口，让 TypeScript 为我们处理两者的合并。如果我们的文件处于模块模式（例如，如果我们需要为我们的 `zip` 实现 `import` 某些内容，可能就会出现这种情况），我们必须将全局扩展包装在 `declare global` 类型声明中（参见[“类型声明”]）：

```
declare global {
  interface Array<T> {
    zip<U>(list: U[]): [T, U][]
  }
}
```

`global` 是一个特殊的命名空间，包含所有全局定义的值（任何您可以在模块模式文件中使用而无需先 `import` 的内容；请参见第10章），它允许您从模块模式文件中增强全局作用域中的名称。

❷

然后我们在 `Array` 的原型上实现 `zip` 方法。我们使用 `this` 类型，以便 TypeScript 正确推断我们调用 `.zip` 的数组的 `T` 类型。

❸

因为 TypeScript 推断映射函数的返回类型为 `(T | U)[]`（TypeScript 不够智能，无法意识到它实际上始终是一个元组，第0个索引中包含 `T`，第1个索引中包含 `U`），我们使用我们的 `tuple` 实用程序（来自[“改进元组的类型推断”]）来创建元组类型，而无需诉诸类型断言。

注意，当我们声明 `interface Array<T>` 时，我们为整个 TypeScript 项目增强了全局 `Array` 命名空间——这意味着即使我们不从文件中导入 *zip.ts*，TypeScript 也会认为 `[].zip` 是可用的。但是为了增强 `Array.prototype`，我们必须确保任何使用 `zip` 的文

件首先加载 *zip.ts*，以便在 `Array.prototype` 上安装 `zip` 方法。我们如何确保任何使用 `zip` 的文件首先加载 *zip.ts*?

很简单：我们更新 *tsconfig.json*，明确从项目中排除 *zip.ts*，以便使用者必须明确先 `import` 它：

```
{
  *exclude*: [
    "./zip.ts"
  ]
}
```

现在我们可以完全安全地使用 `zip`：

```
import './zip'

[1, 2, 3]
  .map(n => n * 2)        // number[]
  .zip(['a', 'b', 'c'])   // [number, string][]
```

运行这个代码会给我们先映射然后压缩数组的结果：

```
[
  [2, 'a'],
  [4, 'b'],
  [6, 'c']
]
```

总结

在本章中，我们涵盖了 TypeScript 类型系统最高级的特性：从方差(variance)的来龙去脉到基于流的类型推断、细化(refinement)、类型拓宽(type widening)、完整性(totality)以及映射和条件类型。然后我们推导出了一些处理类型的高级模式：使用类型品牌来模拟名义类型，利用条件类型的分配属性在类型级别操作类型，以及安全地扩展原型。

如果您没有理解或不记得所有内容，那也没关系——稍后再回到本章，当您在努力表达某些内容更安全时，将其用作参考。

练习

1. 对于以下每对类型，判断第一个类型是否可以赋值给第二个类型，以及为什么可以或不可以。从子类型和方差的角度考虑这些问题，如果不确定，请参考本章开头的规则（如果仍然不确定，只需将其输入代码编辑器中检查！）：

    1. `1` 和 `number`
    2. `number` 和 `1`
    3. `string` 和 `number | string`
    4. `boolean` 和 `number`
    5. `number[]` 和 `(number | string)[]`
    6. `(number | string)[]` 和 `number[]`
    7. `{a: true}` 和 `{a: boolean}`
    8. `{a: {b: [string]}}` 和 `{a: {b: [number | string]}}`
    9. `(a: number) => string` 和 `(b: number) => string`
    10. `(a: number) => string` 和 `(a: string) => string`
    11. `(a: number | string) => string` 和 `(a: string) => string`
    12. `E.X`（定义在枚举 `enum E {X = 'X'}` 中）和 `F.X`（定义在枚举 `enum F {X = 'X'}` 中）

2. 如果您有对象类型 `type O = {a: {b: {c: string}}}`，那么 `keyof O` 的类型是什么？`O['a']['b']` 呢？

3. 编写一个 `Exclusive<T, U>` 类型，计算在 `T` 或 `U` 中但不在两者中都存在的类型。例如，`Exclusive<1 | 2 | 3, 2 | 3 | 4>` 应该解析为 `1 | 4`。逐步写出类型检查器如何计算 `Exclusive<1 | 2, 2 | 4>`。

4. 重写示例（来自[“确定赋值断言”]），以避免确定赋值断言。

[1] 符号执行(symbolic execution)是程序分析的一种形式，您使用称为符号求值器的特殊程序以与运行时相同的方式运行程序，但不为变量分配确定值；相反，每个变量都被建模为一个符号，其值在程序运行时受到约束。符号执行让您可以说诸如”这个变量从未使用过”、“这个函数从不返回”或”在第102行的 `if` 语句的正分支中，变量 `x` 保证不为 `null` “之类的话。

[2] 基于流的类型推断(Flow-based type inference)被少数几种语言支持，包括TypeScript、Flow、Kotlin和Ceylon。它是一种在代码块内细化类型的方法，是C/Java风格显式类型注解和Haskell/OCaml/Scala风格模式匹配的替代方案。其思想是采用符号执行引擎并将其直接嵌入类型检查器中，以便向类型检查器提供反馈，并以更接近人类程序员的方式推理程序。

[3] JavaScript有七个假值： `null` 、 `undefined` 、 `NaN` 、 `0` 、 `-0` 、 `""` ，当然还有 `false` 。其他一切都是真值。

[4] DefinitelyTyped是第三方JavaScript类型声明的开源仓库。要了解更多信息，请跳转到[“JavaScript That Has Type Declarations on DefinitelyTyped”]。

[5] 在某些语言中，这些也称为*不透明类型*。

[6] 您可能想要避免扩展原型还有其他原因，比如代码可移植性、使依赖关系图更明确，或通过仅加载实际使用的方法来提高性能。然而，安全性不再是其中的原因之一。

第7章 处理错误

> *一个物理学家、一个结构工程师和一个程序员开车经过一个陡峭的高山隘口时刹车失灵了。汽车越来越快，他们努力转过弯道，有一两次脆弱的防撞栏杆救了他们，免于从山边滚下去。他们确信所有人都会死，突然他们发现了一个逃生车道。他们开进逃生车道，安全停下。*
>
> *物理学家说：“我们需要建模刹车片的摩擦力和由此产生的温度上升，看看能否弄清楚它们为什么失效。”*
>
> *结构工程师说：“我想后面有几把扳手。我去看看能否弄清楚出了什么问题。”*
>
> *程序员说：“我们为什么不看看这个问题能否复现？”*
>
> *佚名*

TypeScript竭尽所能将运行时异常转移到编译时：从它提供的丰富类型系统到它执行的强大静态和符号分析，它努力工作，这样您就不必花费周五晚上调试拼写错误的变量和空指针异常(null pointer exceptions)(这样您的待命同事也不会因此而错过他们姑奶奶的生日聚会)。

不幸的是，无论您用什么语言编写，有时运行时异常确实会溜过去。TypeScript在防止它们方面确实很擅长，但即使它也无法防止网络和文件系统故障、解析用户输入错误、堆栈溢出和内存不足错误等问题。它所做的——得益于其丰富的类型系统——是为您提供许多方法来处理最终通过的运行时错误。

在本章中，我将带您了解在TypeScript中表示和处理错误的最常见模式：

- 返回`null`
- 抛出异常
- 返回异常
- `Option`类型

您使用哪种机制取决于您自己和您的应用程序。当我介绍每种错误处理机制时，我会讨论它的优缺点，这样您就可以为自己做出正确的选择。

返回null

我们将编写一个程序，询问用户的生日，然后将其解析为`Date`对象：

```
function ask() {
  return prompt('When is your birthday?')
}

function parse(birthday: string): Date {
  return new Date(birthday)
}

let date = parse(ask())
console.info('Date is', date.toISOString())
```

我们也许应该验证用户输入的日期——毕竟，这只是一个文本提示：

```
// ...
function parse(birthday: string): Date | null {
  let date = new Date(birthday)
  if (!isValid(date)) {
    return null
  }
  return date
}

// 检查给定日期是否有效
function isValid(date: Date) {
  return Object.prototype.toString.call(date) === '[object Date]'
      && !Number.isNaN(date.getTime())
}
```

当我们使用这个函数时，我们被迫在使用结果之前首先检查结果是否为`null`：

```
// ...
let date = parse(ask())
if (date) {
  console.info('Date is', date.toISOString())
} else {
  console.error('Error parsing date for some reason')
}
```

返回`null`是以类型安全方式处理错误的最轻量级方法。有效的用户输入产生`Date`，无效的用户输入产生`null`，类型系统为我们检查是否处理了两种情况。

然而，这样做我们会丢失一些信息——`parse`没有告诉我们操作确切为什么失败，这对必须梳理我们的日志来调试此问题的工程师来说很糟糕，对用户来说也很糟糕，他们得到的弹窗说”由于某种原因解析日期出错”，而不是具体的、可操作的错误，如”以YYYY/MM/DD的形式输入日期”。

返回`null`也很难组合：当您开始嵌套和链式操作时，必须在每次操作后检查`null`会变得冗长。

抛出异常

让我们抛出异常而不是返回 `null`，这样我们可以处理特定的失败模式，并获得一些关于失败的元数据，以便更容易地调试它。

```
// ...
function parse(birthday: string): Date {
  let date = new Date(birthday)
  if (!isValid(date)) {
    throw new RangeError('Enter a date in the form YYYY/MM/DD')
  }
  return date
}
```

现在当我们使用这个代码时，我们需要小心地捕获异常，这样我们可以优雅地处理它而不会崩溃整个应用程序：

```
// ...
try {
  let date = parse(ask())
  console.info('Date is', date.toISOString())
} catch (e) {
  console.error(e.message)
}
```

我们可能想要小心地重新抛出其他异常，这样我们就不会静默地吞掉每一个可能的错误：

```
// ...
try {
  let date = parse(ask())
  console.info('Date is', date.toISOString())
} catch (e) {
  if (e instanceof RangeError) {
    console.error(e.message)
```

```
  } else {
    throw e
  }
}
```

我们可能想要为更特定的情况子类化错误，这样当另一个工程师修改 `parse` 或 `ask` 抛出其他 `RangeError` 时，我们可以区分我们的错误和他们添加的错误：

```
// ...

// 自定义错误类型
class InvalidDateFormatError extends RangeError {}
class DateIsInTheFutureError extends RangeError {}

function parse(birthday: string): Date {
  let date = new Date(birthday)
  if (!isValid(date)) {
    throw new InvalidDateFormatError('Enter a date in the form
YYYY/MM/DD')
  }
  if (date.getTime() > Date.now()) {
    throw new DateIsInTheFutureError('Are you a timelord?')
  }
  return date
}

try {
  let date = parse(ask())
  console.info('Date is', date.toISOString())
} catch (e) {
  if (e instanceof InvalidDateFormatError) {
    console.error(e.message)
  } else if (e instanceof DateIsInTheFutureError) {
    console.info(e.message)
  } else {
    throw e
  }
}
```

看起来不错。我们现在可以做的不仅仅是标示出什么东西失败了：我们可以使用自定义错误来指示它*为什么*失败。这些错误在梳理我们的服务器日志来调试问题时可能会很方便，或者我们可以将它们映射到特定的错误对话框，为我们的用户提供关于他们做错了什么以及如何修复的可操作反馈。我们还可以通过将任意数量的操作包装在单个 `try` / `catch` 中来有效地链接和嵌套操作（我们不必检查每个操作的失败，就像我们返回 `null` 时所做的那样）。

使用这个代码感觉如何？假设大的 `try` / `catch` 在一个文件中，其余的代码在从其他地方导入的库中。工程师如何知道要捕获那些特定类型的错误（`InvalidDateFormatError` 和 `DateIsInTheFutureError`），或者甚至只是检查常规的旧 `RangeError`？（请记住 TypeScript 不会将异常编码为函数签名的一部分。）我们可以在函数名称中指示它（`parseThrows`），或将其包含在文档块中：

```
/**
 * @throws {InvalidDateFormatError} The user entered their birthday
incorrectly.
 * @throws {DateIsInTheFutureError} The user entered a birthday in the
future.
 */
function parse(birthday: string): Date {
  // ...
```

但在实践中，工程师可能根本不会将此代码包装在 `try` / `catch` 中并检查异常，因为工程师很懒（至少，我是），而类型系统没有告诉他们错过了一个情况并且应该处理它。然而，有时——比如在这个例子中——错误是如此预期，以至于下游代码真的应该处理它们，以免它们导致程序崩溃。

我们还可以如何向消费者表明他们应该处理成功和错误情况？

返回异常

TypeScript 不是 Java，不支持 `throws` 子句。[1] 但我们可以通过联合类型实现类似的功能：

```
// ...
function parse(
  birthday: string
): Date | InvalidDateFormatError | DateIsInTheFutureError {
  let date = new Date(birthday)
  if (!isValid(date)) {
    return new InvalidDateFormatError('Enter a date in the form
YYYY/MM/DD')
  }
  if (date.getTime() > Date.now()) {
    return new DateIsInTheFutureError('Are you a timelord?')
  }
  return date
}
```

现在消费者被迫处理所有三种情况——`InvalidDateFormatError`、`DateIsInTheFutureError` 和成功解析——否则他们会在编译时得到一个 `TypeError`：

```
// ...
let result = parse(ask()) // 要么是日期要么是错误
if (result instanceof InvalidDateFormatError) {
  console.error(result.message)
} else if (result instanceof DateIsInTheFutureError) {
  console.info(result.message)
} else {
  console.info('Date is', result.toISOString())
}
```

在这里，我们成功利用了 TypeScript 的类型系统来：

- 在 `parse` 的签名中编码可能的异常。

- 向消费者传达可能抛出哪些特定异常。
- 强制消费者处理（或重新抛出）每个异常。

懒惰的消费者可以避免单独处理每个错误。但他们必须明确地这样做：

```
// ...
let result = parse(ask()) // 要么是日期要么是错误
```

if (result instanceof Error) { console.error(result.message) } else { console.info(‘Date is’, result.toISOString()) }

当然，您的程序仍可能因内存不足错误或堆栈溢出异常而崩溃，但对于这些情况我们无法做太多恢复处理。

这种方法轻量且不需要复杂的数据结构，但也足够信息化，让使用者能够了解错误代表的失败类型以及搜索更多信息的方向。

缺点是链式调用和嵌套错误操作很快就会变得冗长。如果一个函数返回 `T | Error1`，那么任何使用该函数的函数有两个选择：

1. 显式处理 `Error1`。

2. 处理 `T`（成功情况）并将 `Error1` 传递给其使用者处理。如果您经常这样做，使用者必须处理的可能错误列表会快速增长：

    ```
    function x(): T | Error1 {
      // ...
    }
    function y(): U | Error1 | Error2 {
      let a = x()
      if (a instanceof Error) {
        return a
      }
      // Do something with a
    }
    function z(): U | Error1 | Error2 | Error3 {
    ```

```
  let a = y()
  if (a instanceof Error) {
    return a
  }
  // Do something with a
}
```

这种方法虽然冗长，但给了我们极好的安全性。

Option类型

您也可以使用专用的数据类型来描述异常。这种方法与返回值和错误的联合类型相比有一些缺点（特别是与不使用这些数据类型的代码的互操作性），但它确实让您能够链式操作可能出错的计算。三种最流行的选择是 `Try`、`Option` 和 `Either` 类型。在本章中，我们只会介绍 `Option` 类型；其他两种在精神上是相似的。

注意

请注意，`Try`、`Option` 和 `Either` 数据类型不像 `Array`、`Error`、`Map` 或 `Promise` 那样内置于JavaScript环境中。如果您想使用这些类型，您需要在NPM上找到实现，或者自己编写它们。

`Option` 类型来自Haskell、OCaml、Scala和Rust等语言。其思想是不返回值，而是返回一个可能包含也可能不包含值的*容器*。容器上定义了一些方法，让您可以链式操作，即使里面实际上可能没有值。容器可以是几乎任何数据结构，只要它能容纳一个值。例如，您可以使用数组作为容器：

// … function parse(birthday: string): Date[] { let date = new Date(birthday) if (!isValid(date)) { return [] } return [date] }

let date = parse(ask()) date .map(_ => *.toISOString()) .forEach(* => console.info(‘Date is’, _))

注意

如您可能注意到的，`Option` 的一个缺点是，就像我们最初返回 `null` 的方法一样，它不会告诉使用者错误发生的原因；它只是表明出了问题。

当您需要连续执行多个操作，每个操作都可能失败时，`Option` 真正发挥作用。

例如，之前我们假设 `prompt` 总是成功的，而 `parse` 可能会失败。但如果 `prompt` 也可能失败怎么办？如果用户取消了生日提示，这可能会发生——这是一个错误，我们不应该继续计算。我们可以用...另一个 `Option` 来建模！

function ask() { let result = prompt(‘When is your birthday?’) if (result === null) { return [] } return [result] } // … ask() .map(parse) .map(date => date.toISOString()) // Error TS2339: Property ‘toISOString’ does not exist on type ‘Date[]’. .forEach(date => console.info(‘Date is’, date))

糟糕——这不起作用。因为我们将 `Date` 数组（`Date[]`）映射为 `Date` 数组的数组（`Date[][]`），我们需要在继续之前将其展平为 `Date` 数组：

flatten(ask() .map(parse)) .map(date => date.toISOString()) .forEach(date => console.info(‘Date is’, date))

// 将数组的数组展平为数组 function flatten(array: T[][]): T[] { return Array.prototype.concat.apply([], array) }

这一切都变得有点笨拙了。因为类型没有告诉您太多（所有东西都是常规数组），所以很难一眼看出代码在做什么。为了解决这个问题，让我们将我们正在做的事情——将值放入容器中，暴露对该值进行操作的方法，以及暴露从容器中获取结果的方法——包装在一个特殊的数据类型中，该类型有助于记录我们的方法。完成实现后，您将能够像这样使用该数据类型：

ask() .flatMap(parse) .flatMap(date => new Some(date.toISOString())) .flatMap(date => new Some(‘Date is’ + date)) .getOrElse(‘Error parsing date for some reason’)

我们将这样定义我们的 `Option` 类型：

- `Option` 是一个由两个类实现的接口：`Some<T>` 和 `None`（见[图7-1]）。它们是 `Option` 的两种类型。`Some<T>` 是一个包含类型 `T` 值的 `Option`，而 `None` 是一个不包含值的 `Option`，表示失败。

- `Option` 既是一个类型也是一个函数。它的类型是一个接口，简单地作为 `Some` 和 `None` 的超类型。它的函数是创建 `Option` 类型新值的方式。

让我们从勾勒类型开始：

interface Option {} class Some implements Option { constructor(private value: T) {} } class None implements Option {}

1. `Option<T>` 是一个我们在 `Some<T>` 和 `None` 之间共享的接口。

2. `Some<T>` 表示一个成功的操作，产生了一个值。就像我们之前使用的数组一样，`Some<T>` 是该值的容器。

3. `None` 表示一个失败的操作，不包含值。

这些类型等价于我们基于数组的 `Option` 实现中的以下内容：

- `Option<T>` 是 `[T] | []`。

- `Some<T>` 是 `[T]`。

- `None` 是 `[]`。

你能用 `Option` 做什么？对于我们的基础实现，我们将只定义两个操作：

`flatMap`
:   一种在可能为空的 `Option` 上链式操作的方式

`getOrElse`
:   一种从 `Option` 中检索值的方式

我们将从在 `Option` 接口上定义这些操作开始，这意味着 `Some<T>` 和 `None` 需要为它们提供具体的实现：

interface Option { flatMap(f: (value: T) => Option): Option getOrElse(value: T): T } class Some extends Option { constructor(private value: T) {} } class None extends Option {}

也就是说：

- `flatMap` 接受一个函数 `f`，该函数接受类型 `T` 的值（`Option` 包含的值的类型）并返回一个 `U` 的 `Option`。`flatMap` 用 `Option` 的值调用 `f`，并返回一个新的 `Option<U>`。

- `getOrElse` 接受一个与 `Option` 包含的值相同类型 `T` 的默认值，并返回该默认值（如果 `Option` 是空的 `None`）或 `Option` 的值（如果 `Option` 是 `Some<T>`）。

在类型的指导下，让我们为 `Some<T>` 和 `None` 实现这些方法：

interface Option { flatMap(f: (value: T) => Option): Option getOrElse(value: T): T } class Some implements Option { constructor(private value: T) {} flatMap(f: (value: T) => Option): Option { return f(this.value) } getOrElse(): T { return this.value } } class None implements Option { flatMap(): Option { return this } getOrElse(value: U): U { return value } }

1. 当我们在 `Some<T>` 上调用 `flatMap` 时，我们传入一个函数 `f`，`flatMap` 用 `Some<T>` 的值调用它来产生一个新类型的新 `Option`。

2. 在 `Some<T>` 上调用 `getOrElse` 只是返回 `Some<T>` 的值。

3. 由于 `None` 表示失败的计算，在它上面调用 `flatMap` 总是返回一个 `None`：一旦计算失败，我们无法从该失败中恢复（至少在我们的特定 `Option` 实现中不能）。

4. 在 `None` 上调用 `getOrElse` 总是返回我们传递给 `getOrElse` 的值。

我们实际上可以超越这个简单的实现，更好地指定我们的类型。如果你只知道你有一个 `Option` 和一个从 `T` 到 `Option<U>` 的函数，那么 `Option<T>` 总是 `flatMap` 到 `Option<U>`。但是当你知道你有一个 `Some<T>` 或一个 `None` 时，你可以更加具体。

[表7-1]显示了在两种类型的 `Option` 上调用 `flatMap` 时我们想要的结果类型。

	来自 `Some<T>`	来自 `None`
到 `Some<U>`	`Some<U>`	`None`
到 `None`	`None`	`None`

表7-1. 在Some<T>和None上调用.flatMap(f)的结果

也就是说，我们知道在 `None` 上映射总是导致 `None`，在 `Some<T>` 上映射导致 `Some<T>` 或 `None`，这取决于调用 `f` 返回什么。我们将利用这一点并使用重载签名来给 `flatMap` 更具体的类型：

interface Option { flatMap(f: (value: T) => None): None flatMap(f: (value: T) => Option): Option getOrElse(value: T): T } class Some implements Option { constructor(private value: T) {} flatMap(f: (value: T) => None): None flatMap(f: (value: T) => Some): Some flatMap(f: (value: T) => Option): Option { return f(this.value) }
getOrElse(): T { return this.value } } class None implements Option { flatMap(): None { return this } getOrElse(value: U): U { return value } }

我们快完成了。剩下要做的就是实现 `Option` 函数，我们将用它来创建新的 `Option`。我们已经实现了 `Option` *类型*作为接口；现在我们要实现一个同名函数（记住 TypeScript 对类型和值有两个单独的命名空间）作为创建新 `Option` 的方式，类似于我们在["伴侣对象模式"]中所做的。如果用户传入 `null` 或 `undefined`，我们将返回一个 `None`；否则，我们将返回一个 `Some`。再次，我们将重载签名来实现这一点：

function Option(value: null | undefined): None function Option(value: T): Some function Option(value: T): Option { if (value == null) { return new None } return new Some(value) }

[![1](images/000000.png)]{#calibre_link-363 .calibre4}

: 如果消费者用 `null` 或 `undefined` 调用 `Option`，我们返回一个 `None`。

[![2](images/000001.png)]{#calibre_link-364 .calibre4}

: 否则，我们返回一个 `Some<T>`，其中 `T` 是用户传入的值的类型。

[![3](images/000002.png)]{#calibre_link-365 .calibre4}

: 最后，我们手动计算两个重载签名的上界。`null | undefined` 和 `T` 的上界是 `T | null | undefined`，简化为 `T`。`None` 和 `Some<T>` 的上界是 `None | Some<T>`，我们已经有了一个名字：`Option<T>`。

就是这样。我们已经派生出一个完全工作的、最小的 `Option` 类型，让我们可以安全地对可能为 `null` 的值执行操作。我们这样使用它：

let result = Option(6) // Some .flatMap(n => Option(n * 3)) // Some .flatMap(n => new None) // None .getOrElse(7) // 7

回到我们的生日提示例子，我们的代码现在按预期工作：

ask() // Option .flatMap(parse) // Option .flatMap(date => new Some(date.toISOString())) // Option .flatMap(date => new Some(‘Date is’ + date)) // Option .getOrElse(‘Error parsing date for some reason’) // string

`Option` 是处理一系列可能成功也可能失败的操作的强大方式。它们为你提供了出色的类型安全，并通过类型系统向消费者表明给定操作可能失败。

然而，`Option` 并非没有缺点。它们用 `None` 表示失败，所以你得不到关于什么失败了以及为什么失败的更多详细信息。它们也不与不使用 `Option` 的代码互操作（你必须显式包装那些 API 以返回 `Option`）。

尽管如此，你在那里所做的还是相当巧妙的。你添加的重载让你能够做一些在大多数语言中无法表达的事情，即使是那些依赖 `Option` 类型来处理可空值的语言。通过尽可能通过重载调用签名将 `Option` 限制为只是 `Some` 或 `None`，你使你的代码变得更加安全，让很多 Haskell 程序员非常嫉妒。现在去给自己来杯冷饮吧——你值得拥有。

总结

在这一章中，我们涵盖了在 TypeScript 中表示和从错误中恢复的不同方式：返回 `null`、抛出异常、返回异常和 `Option` 类型。你现在拥有了一套安全处理可能失败的事物的方法。你选择哪种方法取决于你，并取决于：

- 你是否想要简单地表示某些事情失败了（`null`、`Option`），或者给出关于为什么失败的更多信息（抛出和返回异常）。

- 你是否想要强制消费者显式处理每个可能的异常（返回异常），或者编写更少的错误处理样板代码（抛出异常）。

- 你是否需要一种组合错误的方式（`Option`），或者只是在它们出现时处理它们（`null`、异常）。

练习

1. 为以下 API 设计一种处理错误的方式，使用本章中的一种模式。在这个 API 中，每个操作都可能失败——请随时更新 API 的方法签名以允许失败（或者不要，如果你愿意的话）。想想你如何在处理出现的错误的同时执行一系列操作（例如，获取登录用户的 ID，然后获取他们的朋友列表，然后获取每个朋友的姓名）：

    ```
    class API {
      getLoggedInUserID(): UserID
      getFriendIDs(userID: UserID): UserID[]
      getUserName(userID: UserID): string
    }
    ```

^[1]^ 如果你以前没有使用过 Java，`throws` 子句表示方法可能抛出的运行时异常类型，所以消费者必须处理这些异常。

^[2]^ 也称为 `Maybe` 类型。

^[3]^ 搜索"try type"或"either type"以获取关于这些类型的更多信息。

第8章 异步编程、并发和并行

到目前为止，本书主要讨论的是同步程序——接受输入、执行某些操作，然后在单次运行中完成的程序。但真正有趣的程序——构成现实世界应用程序的构建块，它们发起网络请求、与数据库和文件系统交互、响应用户交互、将CPU密集型工作转移到单独的线程——都使用异步API，如回调、Promise和流。

这些异步任务是JavaScript真正闪耀并区别于Java和C++等其他主流多线程语言的地方。流行的JavaScript引擎如V8和SpiderMonkey用一个线程完成传统上需要多个线程的工作，它们采用聪明的方式在其他任务空闲时将任务多路复用到单个线程上。这个*事件循环*是JavaScript

引擎的标准线程模型，也是我们将假定您使用的模型。从最终用户的角度来看，您的引擎使用事件循环模型还是多线程模型通常并不重要，但这确实会影响我对事物工作原理的解释以及我们设计事物的方式。

这种事件循环并发模型是JavaScript如何避免多线程编程中常见的所有问题，以及同步数据类型、互斥锁、信号量和所有其他多线程术语的开销。当您确实在多个线程上运行JavaScript时，很少使用共享内存；典型的模式是使用消息传递，并在线程间发送数据时序列化数据。这种设计让人想起Erlang、Actor系统和其他纯函数式并发模型，这就是使JavaScript中的多线程编程万无一失的原因。

话虽如此，异步编程确实使程序更难理解，因为您不能再逐行地在心中跟踪程序；您必须知道何时暂停并将执行转移到其他地方，以及何时再次恢复。

TypeScript为我们提供了理解异步程序的工具：类型让我们能够跟踪异步工作，对`async`/`await`的内置支持让我们将熟悉的同步思维应用到异步程序中。我们还可以使用TypeScript为多线程程序指定严格的消息传递协议(protocol)（这比听起来简单得多）。如果所有其他方法都失败了，当您同事的异步代码变得太复杂而您必须熬夜调试时，TypeScript可以给您一个背部按摩（当然，在编译器标志后面）。

但在我们开始处理异步程序之前，让我们更多地谈论现代JavaScript引擎中异步性实际是如何工作的——我们如何能够在看似单线程上暂停和恢复执行？

JavaScript的事件循环

让我们从一个例子开始。我们设置几个定时器，一个在一毫秒后触发，另一个在两毫秒后触发：

setTimeout(() => console.info('A'), 1) setTimeout(() => console.info('B'), 2) console.info('C')

现在，控制台会记录什么呢？是`A`、`B`、`C`吗？

如果您是JavaScript程序员，您直觉知道答案是否定的——实际的触发顺序是`C`、`A`，然后是`B`。如果您之前没有使用过JavaScript或TypeScript，这种行为可能看起来神秘且反直觉。实际上，这非常简单；它只是不遵循与C中的`sleep`或Java中在另一个线程中调度工作相同的并发模型。

在高层次上，JavaScript虚拟机像这样模拟并发（参见图8-1）：

- 主JavaScript线程调用原生异步API，如[`XMLHTTPRequest`]（用于AJAX请求）、

`setTimeout`（用于休眠）、`readFile`（用于从磁盘读取文件）等。这些API由JavaScript平台提供——您无法自己创建它们。

- 一旦您调用原生异步API，控制权返回到主线程，执行继续进行，就好像从未调用过API一样。

- 异步操作完成后，平台在其*事件队列*中放置一个*任务*。每个线程都有自己的队列，用于将异步操作的结果中继回主线程。任务包括关于调用的一些元信息，以及对来自主线程的回调函数的引用。

- 每当主线程的调用栈被清空时，平台将检查其事件队列是否有待处理的任务。如果有等待的任务，平台运行它；这触发一个函数调用，控制权返回到该主线程函数。当由该函数调用产生的调用栈再次为空时，平台再次检查事件队列是否有准备好的任务。这个循环重复，直到调用栈和事件队列都为空，并且所有异步原生API调用都已完成。

![图8-1：JavaScript的事件循环：调用异步API时发生的事情](images/000022.png)

有了这些信息，是时候回到我们的 `setTimeout` 示例了。以下是发生的过程：

1. 我们调用 `setTimeout`，它调用原生的超时 API，传入我们传递的回调函数引用和参数 `1`。

2. 我们再次调用 `setTimeout`，它再次调用原生超时 API，传入我们传递的第二个回调函数引用和参数 `2`。

3. 我们将 `C` 记录到控制台。

4. 在后台，至少一毫秒后，我们的 JavaScript 平台向其事件队列添加一个任务，表明第一个 `setTimeout` 的超时已经过去，其回调现在准备被调用。

5. 再过一毫秒后，平台为第二个 `setTimeout` 的回调向事件队列添加第二个任务。

6. 由于调用栈是空的，步骤 3 完成后，平台查看其事件队列是否有任何任务。如果步骤 4 和/或 5 完成了，那么它会找到一些任务。对于每个任务，它会调用相应的回调函数。

7. 一旦两个定时器都已过期，事件队列和调用栈都为空，程序退出。

这就是为什么我们记录了 `C`、`A`、`B`，而不是 `A`、`B`、`C`。有了这个基线，我们可以开始讨论如何安全地为异步代码添加类型。

使用回调

异步 JavaScript 程序的基本单元是*回调*。回调是一个普通的函数，你将其作为参数传递给另一个函数。就像在同步程序中一样，另一个函数在完成其工作（发出网络请求等）后调用你的函数。异步代码调用的回调只是函数，在它们的类型签名中没有任何迹象表明它们是异步调用的。

对于 NodeJS 原生 API，如 `fs.readFile`（用于异步从磁盘读取文件内容）和 `dns.resolveCname`（用于异步解析 `CNAME` 记录），回调的约定是第一个参数是错误或 `null`，第二个参数是结果或 `null`。

以下是 `readFile` 的类型签名：

function readFile(path: string, options: {encoding: string, flag?: string}, callback: (err: Error | null, data: string | null) => void): void

注意 `readFile` 的类型和 `callback` 的类型都没有什么特别之处：两者都是常规的 JavaScript 函数。从签名来看，没有迹象表明 `readFile` 是异步的，控制权会在调用 `readFile` 后立即传递到下一行（不等待其结果）。

注意

要自己运行以下示例，请务必首先安装 NodeJS 的类型声明：

npm install @types/node – save-dev

要了解更多关于第三方类型声明的信息，请跳转到["JavaScript That Has Type Declarations on DefinitelyTyped"]。

例如，让我们编写一个 NodeJS 程序来读取和写入你的 Apache 访问日志：

import * as fs from ‘fs’

// 从 Apache 服务器的访问日志读取数据 fs.readFile(‘/var/log/apache2/access_log’, {encoding: ‘utf8’}, (error, data) => { if (error) { console.error(‘error reading!’, error) return } console.info(‘success reading!’, data) })

// 同时，向同一访问日志写入数据 fs.appendFile(‘/var/log/apache2/access_log’, ‘New access log entry’, error => { if (error) { console.error(‘error writing!’, error) } })

除非你是 TypeScript 或 JavaScript 工程师，熟悉 NodeJS 内置 API 的工作原理，知道它们是异步的，并且不能依赖代码中 API 调用出现的顺序来决定文件系统操作实际发生的顺序，否则你不会知道我们刚刚引入了一个微妙的错误，即第一个 `readFile` 调用可能会也可能不会返回附加了我们新行的访问日志，这取决于此代码运行时文件系统的繁忙程度。

你可能从经验中知道 `readFile` 是异步的，或者因为你在 NodeJS 的文档中看到了它，或者因为你知道 NodeJS 通常坚持这样的约定：如果函数的最后一个参数是一个接受两个参数的函数--按顺序是 `Error | null` 和 `T | null`--那么该函数通常是异步的，或者因为你跑到走廊对面向邻居借一杯糖，最终待了一会儿聊天，然后不知怎么地聊到了 NodeJS 中的异步编程话题，他们告诉你几个月前遇到的类似问题以及如何解决的。

无论是什么原因，类型肯定没有帮你理解这一点。

除了你无法使用类型来帮助指导你对函数同步性的直觉外，回调也很难排序--这可能导致一些人所说的"回调金字塔"：

async1((err1, res1) => { if (res1) { async2(res1, (err2, res2) => { if (res2) { async3(res2, (err3, res3) => { // …

```
})
      }
    })
  }
})
```

在对操作进行排序时，通常希望在操作成功时继续沿着链条执行，一旦遇到错误就立即终止。使用回调函数时，你必须手动处理这种情况；当你还需要考虑同步错误时（例如，NodeJS的约定是当你给它一个错误类型的参数时抛出异常，而不是用Error对象调用你提供的回调函数），正确排序回调函数可能会变得容易出错。

排序只是你可能想要在异步任务上运行的一种操作——你可能还想并行运行函数以知道它们何时全部完成，让它们竞争以获得第一个完成的结果，等等。

这是普通回调函数的限制。如果没有更复杂的抽象来操作异步任务，处理以某种方式相互依赖的多个回调函数可能会很快变得混乱。

总结一下：

- 使用回调函数来执行简单的异步任务。
- 虽然回调函数非常适合建模简单任务，但当你试图处理大量异步任务时，它们很快就会变得复杂。

用Promise重获理智

幸运的是，我们不是第一个遇到这些限制的程序员。在本节中，我们将开发*promise*的概念，这是一种对异步工作进行抽象的方法，使我们能够组合它、排序它等等。即使你以前使用过promise或future，这也将是理解它们如何工作的有用练习。

注意

大多数现代JavaScript平台都内置支持promise。在本节中，我们将开发自己的部分`Promise`实现作为练习，但在实践中，你应该使用内置的或现成的实现。检查你喜欢的平台是否支持promise 这里，或者跳到[“lib”]了解更多关于在不原生支持promise的平台上进行polyfill的信息。

我们将从一个例子开始，展示我们如何使用`Promise`首先向文件追加内容，然后读取结果：

```
function appendAndReadPromise(path: string, data: string):
Promise<string> {
  return appendPromise(path, data)
    .then(() => readPromise(path))
    .catch(error => console.error(error))
}
```

注意这里没有回调函数金字塔——我们有效地将想要做的事情线性化为一个单一的、易于理解的异步任务链。当一个成功时，下一个就会运行；如果失败，我们跳到`catch`子句。使用基于回调的API，这可能看起来更像：

```
function appendAndRead(
  path: string,
  data: string
  cb: (error: Error | null, result: string | null) => void
) {
  appendFile(path, data, error => {
    if (error) {
      return cb(error, null)
```

```
    }
    readFile(path, (error, result) => {
      if (error) {
        return cb(error, null)
      }
      cb(null, result)
    })
  })
}
```

让我们设计一个允许我们这样做的`Promise` API。

`Promise`从简单的开始：

```
class Promise {
}
```

一个新的`Promise`接受一个我们称为*executor*的函数，`Promise`实现会用两个参数调用它，一个`resolve`函数和一个`reject`函数：

```
type Executor = (
  resolve: Function,
  reject: Function
) => void

class Promise {
  constructor(f: Executor) {}
}
```

`resolve`和`reject`是如何工作的？让我们通过思考如何手动将基于回调的NodeJS API（如`fs.readFile`）包装在基于`Promise`的API中来演示。我们像这样使用NodeJS的内置`fs.readFile` API：

```
import {readFile} from 'fs'

readFile(path, (error, result) => {
```

```
  // ...
})
```

在我们的`Promise`实现中包装该API，现在看起来像这样：

```
import {readFile} from 'fs'

function readFilePromise(path: string): Promise<string> {
  return new Promise((resolve, reject) => {
    readFile(path, (error, result) => {
      if (error) {
        reject(error)
      } else {
        resolve(result)
      }
    })
  })
}
```

因此，`resolve`参数的类型取决于我们使用的特定API（在这种情况下，它的参数类型将是`result`的类型），而`reject`参数的类型总是某种类型的`Error`。回到我们的实现，让我们通过用更具体的类型替换不安全的`Function`类型来更新我们的代码：

```
type Executor<T, E extends Error> = (
  resolve: (result: T) => void,
  reject: (error: E) => void
) => void
// ...
```

因为我们希望能够通过查看`Promise`来了解它将解析为什么类型（例如，`Promise<number>`表示一个异步任务，结果为`number`），我们将使`Promise`成为泛型，并在其构造函数中将其类型参数传递给`Executor`类型：

```
// ...
class Promise<T, E extends Error> {
```

```
  constructor(f: Executor<T, E>) {}
}
```

到目前为止，一切都很好。我们定义了 Promise 的构造函数API并了解了其中涉及的类型。现在，让我们考虑链式调用——我们想要公开哪些操作来运行一系列 Promise 、传播它们的结果并捕获它们的异常？如果你回看本节开头的初始代码示例，这就是 then 和 catch 的作用。让我们将它们添加到我们的 Promise 类型中：

```
// ...
class Promise<T, E extends Error> {
  constructor(f: Executor<T, E>) {}
  then<U, F extends Error>(g: (result: T) => Promise<U, F>):
Promise<U, F>
  catch<U, F extends Error>(g: (error: E) => Promise<U, F>):
Promise<U, F>
}
```

then 和 catch 是连接 Promise 的两种方式： then 将 Promise 的成功结果映射到一个新的 Promise ，[2]而 catch 通过将错误映射到新的 Promise 来从拒绝中恢复。

使用 then 看起来是这样的：

```
let a: () => Promise<string, TypeError> = // ...
let b: (s: string) => Promise<number, never> = // ...
let c: () => Promise<boolean, RangeError> = // ...

a()
  .then(b)
  .catch(e => c()) // b不会出错，所以这是在a出错时执行
  .then(result => console.info('Done', result))
  .catch(e => console.error('Error', e))
```

因为 b 的第二个类型参数的类型是 never （意味着 b 永远不会抛出错误），第一个 catch 子句只会在 a 出错时被调用。但注意，当我们使用 Promise 时，我们不必关心 a 可能抛出错误但 b 不会这一事实——如果 a 成功，那么我们将 Promise 映射到 b ，否则我们跳转到第一个 catch 子句并将 Promise 映射到 c 。如果 c 成功，那么我们记录 Done ，如果它被

拒绝，那么我们再次`catch`。这模仿了常规的`try`/`catch`语句的工作方式，并且为异步任务做了`try`/`catch`为同步任务所做的事情（见图8-2）。

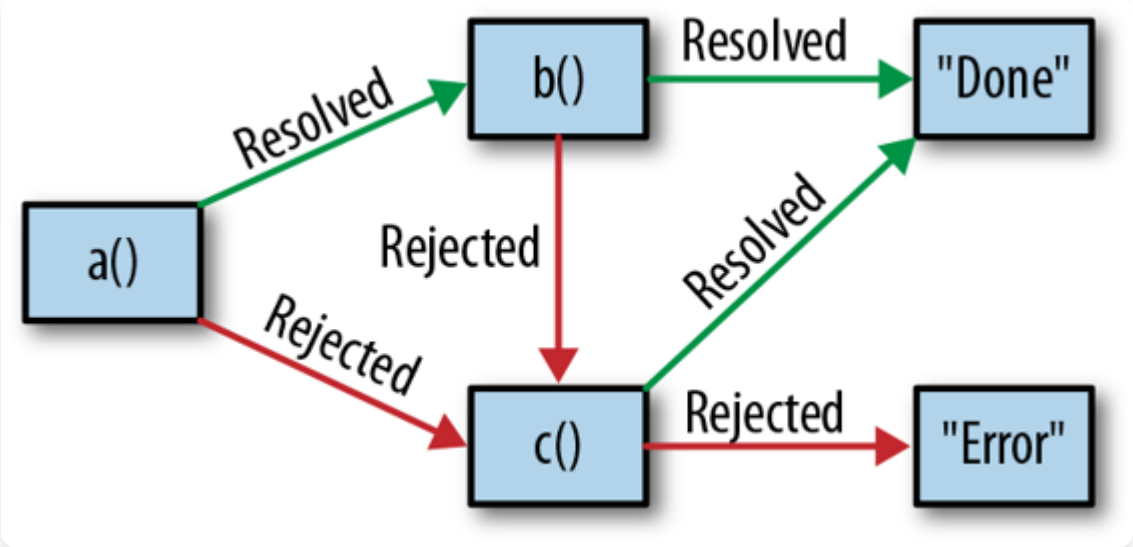


图8-2.Promise状态机

我们还必须处理抛出实际异常的`Promise`情况（即`throw Error('foo')`）。当我们实现`then`和`catch`时，我们将通过用`try`/`catch`包装代码并在`catch`子句中拒绝来做到这一点。不过，这确实有一些影响。这意味着：

1. 每个`Promise`都有可能被拒绝，我们无法静态检查这一点（因为TypeScript不支持在函数签名中指示函数可能抛出哪些异常）。

2. `Promise`并不总是会被`Error`拒绝。因为TypeScript别无选择，只能继承JavaScript的行为，而在JavaScript中当你`throw`时可以抛出任何东西——字符串、函数、数组、`Promise`，不一定是`Error`——我们不能假设拒绝将是`Error`的子类型。这很不幸，但这是我们为了不必强迫消费者对每个promise链进行`try/catch`（可能分布在多个文件或模块中！）而做出的牺牲。

考虑到这一点，让我们通过不类型化错误来放宽我们的`Promise`类型：

```
type Executor<T> = (
  resolve: (result: T) => void,
  reject: (error: unknown) => void
) => void
```

```
class Promise<T> {
  constructor(f: Executor<T>) {}
  then<U>(g: (result: T) => Promise<U>): Promise<U> {
    // ...
  }
  catch<U>(g: (error: unknown) => Promise<U>): Promise<U> {
    // ...
  }
}
```

我们现在有了一个完全成熟的`Promise`接口。

我将把它与`then`和`catch`的实现连接起来作为练习留给你。`Promise`的实现以编写正确而著称难度很高——如果你有雄心并且有几个小时的空闲时间，请前往ES2015规范了解`Promise`的状态机在底层应该如何工作的详细介绍。

async和await

Promise 是一个非常强大的抽象概念，用于处理异步代码。它们非常受欢迎，甚至在 JavaScript（因此也在 TypeScript 中）中拥有自己的语法：`async` 和 `await`。这种语法让你能够像处理同步操作一样与异步操作进行交互。

提示

将 `await` 视为 `.then` 的语言级语法糖。当你 `await` 一个 `Promise` 时，必须在 `async` 块中执行。而且不用 `.catch`，你可以将 `await` 包装在常规的 `try` / `catch` 块中。

假设你有以下 promise（我们在上一节中没有涵盖 `finally`，但它的行为正如你所想的那样，在 `then` 和 `catch` 都有机会触发后执行）：

```
function getUser() {
  getUserID(18)
    .then(user => getLocation(user))
    .then(location => console.info('got location', location))
    .catch(error => console.error(error))
    .finally(() => console.info('done getting location'))
}
```

要将此代码转换为 `async` 和 `await`，首先将其放在 `async` 函数中，然后 `await` promise 的结果：

```
async function getUser() {
  try {
    let user = await getUserID(18)
    let location = await getLocation(user)
    console.info('got location', user)
  } catch(error) {
    console.error(error)
  } finally {
    console.info('done getting location')
```

```
  }
}
```

由于 `async` 和 `await` 是 JavaScript 特性，我们不会在这里深入探讨——只需说 TypeScript 完全支持它们，并且它们是完全类型安全的。每当使用 promises 时都要使用它们，以便更容易推理链式操作并避免大量的 `then`。要了解更多关于 `async` 和 `await` 的信息，请访问它们在 MDN 上的文档。

异步流(Async Streams)

虽然 promises 非常适合建模、排序和组合未来值，但如果你有多个值，这些值将在未来的多个时间点变得可用，该怎么办呢？这听起来并不那么奇特——想想从文件系统读取的文件位、通过互联网从 Netflix 服务器流式传输到你笔记本电脑的视频像素、你填写表单时的一系列击键、一些朋友来到你家参加晚宴，或者在超级星期二期间投入投票箱的选票。虽然这些事情在表面上可能听起来很不同，但你可以将它们都视为异步流；它们都是事物的列表，其中每个事物在未来的某个时间点出现。

有几种方法可以对此进行建模，最常见的是使用事件发射器(event emitter)（如 NodeJS 的 `EventEmitter`）或响应式编程库(reactive programming library)，如 RxJS。两者之间的区别就像回调和 promises 之间的区别：事件快速且轻量级，而响应式编程库更强大，为你提供了组合和排序事件流的能力。

我们将在下一节中讨论事件发射器。要了解更多关于响应式编程的信息，请查看你喜欢的响应式编程库的文档——例如 RxJS、MostJS 或 xstream。

事件发射器(Event Emitters)

在高层次上，事件发射器提供了支持在通道上发射事件并监听该通道上事件的 API：

```
interface Emitter {

  // 发送事件
  emit(channel: string, value: unknown): void

  // 当事件被发送时执行某些操作
  on(channel: string, f: (value: unknown) => void): void

}
```

事件发射器是 JavaScript 中的一种流行设计模式。在使用 DOM 事件、JQuery 事件或 NodeJS 的 `EventEmitter` 模块时，你可能已经遇到过它们。

在大多数语言中，像这样的事件发射器是不安全的。这是因为 `value` 的类型取决于特定的 `channel`，在大多数语言中，你不能使用类型来表示这种关系。除非你的语言既支持重载函数签名又支持字面量类型，否则你很难说”这是在此通道上发射的事件类型”。生成方法来发射事件并监听每个通道的宏是解决这个问题的常见解决方法，但在 TypeScript 中，你可以使用类型系统自然而安全地表达这一点。

例如，假设我们正在使用 NodeRedis 客户端，这是一个用于流行的 Redis 内存数据存储的 Node API。它的工作原理如下：

```
import Redis from 'redis'

// 创建一个新的 Redis 客户端实例
let client = redis.createClient()

// 监听客户端发出的几个事件
client.on('ready', () => console.info('Client is ready'))
client.on('error', e => console.error('An error occurred!', e))
```

```
client.on('reconnecting', params => console.info('Reconnecting...',
params))
```

作为使用 Redis 库的程序员，我们希望知道在使用 `on` API 时回调函数中期望的参数类型。但是由于每个参数的类型取决于 Redis 发出的通道，单一类型是不够的。如果我们是这个库的作者，实现安全性的最简单方法是使用重载类型：

```
type RedisClient = {
  on(event: 'ready', f: () => void): void
  on(event: 'error', f: (e: Error) => void): void
  on(event: 'reconnecting',
     f: (params: {attempt: number, delay: number}) => void): void
}
```

这种方法效果不错，但有点冗长。让我们用映射类型来表达它（参见[“映射类型”]），将事件定义提取到它们自己的类型 `Events` 中：

```
type Events = {
  ready: void
  error: Error
  reconnecting: {attempt: number, delay: number}
}

type RedisClient = {
  on<E extends keyof Events>(
    event: E,
    f: (arg: Events[E]) => void
  ): void
}
```

❶

我们首先定义一个单一对象类型，枚举 Redis 客户端可能发出的每个事件，以及该事件的参数。

❷

我们遍历我们的 `Events` 类型，告诉 TypeScript `on` 可以用我们定义的任何事件来调用。然后我们可以使用这个类型让 Node-Redis 库更安全，通过尽可能安全地为它的两个方法—`emit` 和 `on`—进行类型定义：

```
// ...
type RedisClient = {
  on<E extends keyof Events>(
    event: E,
    f: (arg: Events[E]) => void
  ): void
  emit<E extends keyof Events>(
    event: E,
    arg: Events[E]
  ): void
}
```

这种将事件名称和参数提取到一个形状中并遍历该形状以生成监听器和发射器的模式在现实世界的 TypeScript 代码中很常见。它也很简洁且非常安全。当发射器以这种方式类型化时，你不能拼错键、错误类型化参数或忘记传入参数。它还为使用你的代码的工程师提供文档，因为他们的代码编辑器会向他们建议可能监听的事件和这些事件回调中的参数类型。

现实中的发射器

使用映射类型构建类型安全事件发射器是一种流行的模式。例如，这就是 DOM 事件在 TypeScript 标准库中的类型化方式。`WindowEventMap` 是从事件名称到事件类型的映射，`.addEventListener` 和 [`.removeEventListener`] API 遍历它以产生比默认 `Event` 类型更好、更具体的事件类型：

```
// lib.dom.ts
interface WindowEventMap extends GlobalEventHandlersEventMap {
  // ...
  contextmenu: PointerEvent
  dblclick: MouseEvent
  devicelight: DeviceLightEvent
  devicemotion: DeviceMotionEvent
  deviceorientation: DeviceOrientationEvent
  drag: DragEvent
```

```
  // ...
}

interface Window extends EventTarget, WindowTimers,
WindowSessionStorage,
  WindowLocalStorage, WindowConsole, GlobalEventHandlers,
IDBEnvironment,
  WindowBase64, GlobalFetch {
  // ...
  addEventListener<K extends keyof WindowEventMap>(
    type: K,
    listener: (this: Window, ev: WindowEventMap[K]) => any,
    options?: boolean | AddEventListenerOptions
  ): void
  removeEventListener<K extends keyof WindowEventMap>(
    type: K,
    listener: (this: Window, ev: WindowEventMap[K]) => any,
    options?: boolean | EventListenerOptions
  ): void
}
```

类型安全多线程

到目前为止，我们一直在讨论可能在单个 CPU 线程上运行的异步程序，这是你将编写的大多数 JavaScript 和 TypeScript 程序可能属于的程序类别。但有时，在执行 CPU 密集型任务时，你可能会选择真正的并行性：将工作分配到多个线程的能力，以便更快地完成或保持主线程空闲和响应。在本节中，我们将探索在浏览器和服务器中编写安全并行程序的几种模式。

在浏览器中：使用 Web Workers

Web Workers 是在浏览器中进行多线程处理的一种广泛支持的方式。您可以从主 JavaScript 线程启动一些 workers——特殊的受限后台线程——并使用它们来处理那些原本会阻塞主线程并使 UI 无响应的任务（即 CPU 密集型任务）。Web Workers 是在浏览器中真正并行运行代码的一种方式；虽然 `Promise` 和 `setTimeout` 等异步 API 能并发运行代码，但 Workers 让您能够在另一个 CPU 线程上并行运行代码。Web Workers 可以发送网络请求、写入文件系统等，但有一些小的限制。

因为 Web Workers 是浏览器提供的 API，其设计者非常重视安全性——不是我们熟知和喜爱的类型安全，而是*内存安全*。任何编写过 C、C++、Objective C 或多线程 Java 或 Scala 的人都知道并发操作共享内存的陷阱。当您有多个线程从同一块内存中读取和写入时，很容易遇到各种并发问题，如不确定性、死锁等。

因为浏览器代码必须特别安全，并最小化崩溃浏览器和造成糟糕用户体验的可能性，主线程和 Web Workers 之间以及 Web Workers 和其他 Web Workers 之间通信的主要方式是*消息传递*。

注意

要跟上本节中的示例，请确保通过在您的 *tsconfig.json* 中启用 `dom` 库来告诉 TSC 您计划在浏览器中运行此代码：

```
{
  "compilerOptions": {
    "lib": ["dom", "es2015"]
  }
}
```

对于您在 Web Worker 中运行的代码，请使用 `webworker` 库：

```
{
  "compilerOptions": {
    "lib": ["webworker", "es2015"]
```

```
  }
}
```

如果您对 Web Worker 脚本和主线程都使用单个 *tsconfig.json*，请同时启用两者。

消息传递 API 的工作原理如下。您首先从一个线程生成一个 web worker：

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')
```

然后，您向该 worker 传递消息：

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')

worker.postMessage('some data')
```

您可以使用 `postMessage` API 向另一个线程传递几乎任何类型的数据。

主线程会在将数据传递给 worker 线程之前克隆您传递的数据。在 Web Worker 端，您使用全局可用的 `onmessage` API 监听传入事件：

```
// WorkerScript.ts
onmessage = e => {
  console.log(e.data) // 输出 'some data'
}
```

要进行相反方向的通信——从 worker 回到主线程——您使用全局可用的 `postMessage` 向主线程发送消息，并在主线程中使用 `.onmessage` 方法监听传入消息。将所有内容整合在一起：

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')
worker.onmessage = e => {
```

```
  console.log(e.data) // 输出 'Ack: "some data"'
}
worker.postMessage('some data')

// WorkerScript.ts
onmessage = e => {
  console.log(e.data) // 输出 'some data'
  postMessage(`Ack: "${e.data}"`)
}
```

这个 API 很像我们在[“事件发射器”]中看到的事件发射器 API。这是一种传递消息的简单方式，但没有类型，我们不知道是否正确处理了可能发送的所有可能类型的消息。

由于这个 API 实际上就是一个事件发射器，我们可以应用与常规事件发射器相同的技术来为其添加类型。例如，让我们为聊天客户端构建一个简单的消息层，我们将在 worker 线程中运行它。消息层将向主线程推送更新，我们不会担心错误处理、权限等问题。我们首先定义一些传入和传出消息类型（主线程向 worker 线程发送 `Commands`，worker 线程向主线程发回 `Events`）：

```
// MainThread.ts
type Message = string
type ThreadID = number
type UserID = number
type Participants = UserID[]

type Commands = {
  sendMessageToThread: [ThreadID, Message]
  createThread: [Participants]
  addUserToThread: [ThreadID, UserID]
  removeUserFromThread: [ThreadID, UserID]
```

}

type Events = { receivedMessage: [ThreadID, UserID, Message] createdThread: [ThreadID, Participants] addedUserToThread: [ThreadID, UserID] removedUserFromThread: [ThreadID, UserID] }

我们如何将这些类型应用到 Web Worker 消息传递 API 中？最简单的方式可能是定义一个包含所有可能消息类型的联合类型，然后根据 `Message` 类型进行切换。但这可能会变得相当乏味。对于我们的 `Command` 类型，它可能看起来像这样：

// WorkerScript.ts type Command = | {type: ‘sendMessageToThread’, data: [ThreadID, Message]} | {type: ‘createThread’, data: [Participants]} | {type: ‘addUserToThread’, data: [ThreadID, UserID]} | {type: ‘removeUserFromThread’, data: [ThreadID, UserID]}

onmessage = e => processCommandFromMainThread(e.data)

function processCommandFromMainThread(command: Command) { switch (command.type) { case ‘sendMessageToThread’: let [threadID, message] = command.data console.log(message) // … } }

[![1](images/000000.png)]{#calibre_link-386 .calibre4}

: 我们定义了一个联合类型，包含主线程可能发送给工作线程的所有可能命令，以及每个命令的参数。

[![2](images/000001.png)]{#calibre_link-387 .calibre4}

: 这只是一个常规的联合类型。当定义长的联合类型时，使用前导管道符（`|`）可以使这些类型更易读。

[![3](images/000002.png)]{#calibre_link-388 .calibre4}

: 我们接收通过无类型的 `onmessage` API 发送的消息，并将处理它们的任务委托给我们的类型化 `processCommandFromMainThread` API。

[![4](images/000003.png)]{#calibre_link-389 .calibre4}

: `processCommandFromMainThread` 负责处理来自主线程的所有传入消息。它是无类型 `onmessage` API 的安全、类型化包装器。

[![5](images/000004.png)]{#calibre_link-390 .calibre4}

: 由于 `Command` 类型是一个可辨识联合类型（参见 \[\[discriminated unions\]\]），我们使用 `switch` 来详尽地处理主线程可能发送给我们的每种可能的消息

类型。

让我们将 Web Workers 的特殊 API 抽象到熟悉的基于 `EventEmitter` 的 API 之后。这样我们就可以减少传入和传出消息类型的冗长性。

我们将从构建 NodeJS 的 `EventEmitter` API 的类型安全包装器开始（它在浏览器中可通过 NPM 上的 [`events` 包](https://www.npmjs.com/package/events) 获得）：

import EventEmitter from ‘events’

class SafeEmitter< Events extends Record<PropertyKey, unknown[]>
> { private emitter = new EventEmitter
emit(
channel: K, …data: EventsK) { return this.emitter.emit(channel, …data) } on(
channel: K, listener: (…data: EventsK) => void) { return this.emitter.on(channel, listener) } }

[![1](images/000000.png)]{#calibre_link-396 .calibre4}

: `SafeEmitter` 声明了一个泛型类型 `Events`，这是一个从 `PropertyKey`（TypeScript 的内置类型，表示有效的对象键：`string`、`number` 或 `Symbol`）到参数列表的 `Record` 映射。

[![2](images/000001.png)]{#calibre_link-397 .calibre4}

: 我们将 `emitter` 声明为 `SafeEmitter` 上的私有成员。我们这样做而不是继承 `SafeEmitter`，因为我们的 `emit` 和 `on` 签名比它们在 `EventEmitter` 中的重载对应项更加严格，由于函数在其参数上是逆变的（记住，要使函数 `a` 可赋值给另一个函数 `b`，其参数必须是 `b` 中对应参数的超类型），TypeScript 不会让我们声明这些重载。

[![3](images/000002.png)]{#calibre_link-398 .calibre4}

: `emit` 接受一个 `channel` 加上与我们在 `Events` 类型中定义的参数列表对应的参数。

[![4](images/000003.png)]{#calibre_link-399 .calibre4}

: 类似地，`on` 接受一个 `channel` 和一个 `listener`。`listener` 接受可变数

量的参数，对应我们在 `Events` 类型中定义的参数列表。

我们可以使用 `SafeEmitter` 大幅减少安全实现监听层所需的样板代码。在工作线程端，我们将所有 `onmessage` 调用委托给我们的发射器，并向消费者公开一个便捷且安全的监听器 API：

// WorkerScript.ts type Commands = { sendMessageToThread: [ThreadID, Message] createThread: [Participants] addUserToThread: [ThreadID, UserID] removeUserFromThread: [ThreadID, UserID] }

type Events = { receivedMessage: [ThreadID, UserID, Message] createdThread: [ThreadID, Participants] addedUserToThread: [ThreadID, UserID] removedUserFromThread: [ThreadID, UserID] }

// 监听来自主线程的事件 let commandEmitter = new SafeEmitter ()

// 向主线程发射事件 let eventEmitter = new SafeEmitter ()

// 使用我们的类型安全事件发射器 // 包装来自主线程的传入命令 onmessage = command => commandEmitter.emit(command.data.type, …command.data.data)

// 监听工作线程发出的事件，并将它们发送到主线程 eventEmitter.on(‘receivedMessage’, data => postMessage({type: ‘receivedMessage’, data})) eventEmitter.on(‘createdThread’, data => postMessage({type: ‘createdThread’, data})) // 等等

// 响应来自主线程的 sendMessageToThread 命令 commandEmitter.on(‘sendMessageToThread’, (threadID, message) =>

console.log(好的，我将向线程ID ${threadID}发送消息))

// 向主线程发送一个事件 eventEmitter.emit(‘createdThread’, 123, [456, 789])

反过来，我们也可以使用基于`EventEmitter`的API从主线程向工作线程发送命令。请注意，如果您在自己的代码中使用这种模式，您可能会考虑使用功能更全面的发射器（如Paolo Fragomeni优秀的[`EventEmitter2`](https://www.npmjs.com/package/eventemitter2)），它支持通配符监听器，这样您就不必为每种类型的事件手动添加监听器：

// MainThread.ts type Commands = { sendMessageToThread: [ThreadID, Message] createThread: [Participants] addUserToThread: [ThreadID, UserID] removeUserFromThread: [ThreadID, UserID] }

type Events = { receivedMessage: [ThreadID, UserID, Message] createdThread: [ThreadID, Participants] addedUserToThread: [ThreadID, UserID] removedUserFromThread: [ThreadID, UserID] }

let commandEmitter = new SafeEmitter () let eventEmitter = new SafeEmitter ()

let worker = new Worker(‘WorkerScript.js’)

// 监听来自工作线程的事件，// 并使用我们的类型安全事件发射器重新发射它们 worker.onmessage = event => eventEmitter.emit(event.data.type, …event.data.data)

// 监听此线程发出的命令，并将其发送到我们的工作线程 commandEmitter.on(‘sendMessageToThread’, data => worker.postMessage({type: ‘sendMessageToThread’, data})) commandEmitter.on(‘createThread’, data => worker.postMessage({type: ‘createThread’, data})) // 等等。

// 当工作线程告诉我们创建了新线程时执行某些操作 eventEmitter.on(‘createdThread’, (threadID, participants) => console.log(‘创建了一个新的聊天线程！’, threadID, participants))

// 向我们的工作线程发送命令 commandEmitter.emit(‘createThread’, [123, 456])

就是这样！我们创建了一个简单的类型安全包装器，用于熟悉的事件发射器抽象，可以在各种设置中使用，从浏览器中的光标事件到跨线程通信，使线程间传递消息变得安全。这是TypeScript中的常见模式：即使某些东西不安全，您通常也可以将其包装在类型安全的API中。

类型安全协议 {#typesafe-protocols .calibre39}

到目前为止，我们已经研究了在两个线程之间来回传递消息。要扩展这种技术以表示特定命令总是接收特定事件作为响应，需要做什么？

让我们构建一个简单的调用-响应协议，可以用来跨线程移动函数评估。我们无法轻松地在线程间传递函数，但可以在工作线程中定义函数并向其发送参数，然后发送结果回来。例如，假设我们正在构建一个支持三种操作的矩阵数学引擎：查找矩阵的行列式、计算两个矩阵的点积以及反转矩阵。

您知道惯例---让我们首先为这三种操作勾勒出类型：

type Matrix = number[][]

type MatrixProtocol = { determinant: { in: [Matrix] out: number } ‘dot-product’: { in: [Matrix, Matrix] out: Matrix } invert: { in: [Matrix] out: Matrix } }

我们在主线程中定义矩阵，并在工作线程中运行所有计算。再次，这个想法是用安全的操作包装不安全的操作（从工作线程发送和接收非类型化消息），为消费者公开一个定义良好的类型化API。在这个简单的实现中，我们首先定义一个简单的请求-响应协议`Protocol`，它列出了工作线程可以执行的操作及其预期的输入和输出类型。然后我们定义一个通用的[`createProtocol`]函数，它接受一个`Protocol`和一个Worker的文件路径，并返回一个函数，该函数接受该协议中的`command`并返回一个最终函数，我们可以调用它来实际评估特定参数集的`command`。好的，开始吧：

type Protocol = { [command: string]: { in: unknown[] out: unknown } }
function createProtocol

(script: string) { return (command: K) => (…args: P[K][‘in’]) => new Promise<P[K][‘out’]> ((resolve, reject) => { let worker = new Worker(script) worker.onerror = reject worker.onmessage = event => resolve(event.data.data) worker.postMessage({command, args}) }) }

[![1](images/000000.png)]{#calibre_link-405 .calibre4}

: 我们首先定义一个通用的`Protocol`类型，它不特定于我们的`MatrixProtocol`。

[![2](images/000001.png)]{#calibre_link-406 .calibre4}

: 当我们调用`createProtocol`时，我们传入工作线程`script`的文件路径，以及一个特定的`Protocol`。

[![3](images/000002.png)]{#calibre_link-407 .calibre4}

: `createProtocol`返回一个匿名函数，我们可以用`command`调用它，`command`是

我们在[![2](images/000001.png)]中绑定的`Protocol`中的一个键。

[![4](images/000003.png)]{#calibre_link-408 .calibre4}

: 然后我们使用在[![3](images/000002.png)]中传递的特定命令的`in`类型调用该函数。

[![5](images/000004.png)]{#calibre_link-409 .calibre4}

: 这为我们提供了该命令的特定 `out` 类型的 `Promise`，如我们特定协议中定义的那样。注意我们必须显式地将类型参数绑定到 `Promise`，否则它默认为 `{}`。

现在让我们将 `MatrixProtocol` 类型以及 Web Worker 脚本的路径应用到 `createProtocol`（我们不会深入讨论如何计算行列式的细节，我假设你已经在 *MatrixWorkerScript.ts* 中实现了它）。我们将得到一个可以用来在该协议中运行特定命令的函数：

let runWithMatrixProtocol = createProtocol('MatrixWorkerScript.js') let parallelDeterminant = runWithMatrixProtocol('determinant')

parallelDeterminant([[1, 2], [3, 4]]) .then(determinant => console.log(determinant) // -2)

很酷，对吧？我们将完全不安全的东西——线程间无类型消息传递——抽象为完全类型安全的请求-响应协议。你可以使用该协议运行的所有命令都在一个地方（`MatrixProtocol`），而我们的核心逻辑（`createProtocol`）与具体的协议实现（`runWithMatrixProtocol`）是分离的。

每当你需要在两个进程之间进行通信——无论是在同一台机器上还是在网络上的多台计算机之间——类型安全的协议(protocol)都是使该通信安全的绝佳工具。虽然本节有助于培养对协议解决哪些问题的直觉，但对于实际应用，你可能会想要使用现有工具，如 Swagger、gRPC、Thrift 或 GraphQL——有关概述，请转到["类型安全的 API"]。

在 NodeJS 中：使用子进程(Child Processes)

注意

要跟随本节中的示例，请务必从 NPM 安装 NodeJS 的类型声明：

npm install @types/node – save-dev

要了解更多关于使用类型声明的信息，请跳到["在 DefinitelyTyped 上有类型声明的 JavaScript"]。

NodeJS 中的类型安全并行处理与浏览器中 Web Worker 线程的工作方式相同（见["类型安全协议"]）。虽然消息传递层本身是不安全的，但在其之上构建类型安全的 API 很容易。NodeJS 的子进程 API 看起来像这样：

// MainThread.ts import {fork} from ‘child_process’

let child = fork(‘./ChildThread.js’)

child.on(‘message’, data => console.info(‘Child process sent a message’, data))

child.send({type: ‘syn’, data: [3]})

[![1](images/000000.png)]{#calibre_link-416 .calibre4}

：我们使用 NodeJS 的 `fork` API 来生成一个新的子进程。

[![2](images/000001.png)]{#calibre_link-417 .calibre4}

：我们使用 `on` API 监听来自子进程的传入消息。NodeJS 子进程可能向其父进程发送几种消息；这里，我们只关心 ``'message'`` 消息。

[![3](images/000002.png)]{#calibre_link-418 .calibre4}

：我们使用 `send` API 向子进程发送消息。

在我们的子线程中，我们使用 `process.on` API 监听来自主线程的消息，并使用 `process.send` 发送回消息：

// ChildThread.ts process.on(‘message’, data => console.info(‘Parent process sent a message’, data))

process.send({type: ‘ack’, data: [3]})

[![1](images/000000.png)]{#calibre_link-422 .calibre4}

: 我们在全局定义的 `process` 上使用 `on` API 来监听来自父线程的传入消息。

[![2](images/000001.png)]{#calibre_link-423 .calibre4}

: 我们在 `process` 上使用 `send` API 向父进程发送消息。

因为机制与 Web Workers 如此相似，我将把在 NodeJS 中实现类型安全协议来抽象进程间通信作为练习留给你。

总结

在本章中，我们从 JavaScript 事件循环(event loop)的基础知识开始，然后讨论了 JavaScript 中异步代码的构建模块以及如何在 TypeScript 中安全地表达它们：回调(callbacks)、Promise、`async`/`await` 和事件发射器(event emitters)。然后我们涵盖了多线程，探讨了在线程间传递消息（在浏览器和服务器上）以及构建完整的线程间通信协议。

就像[第7章]一样，使用哪种技术取决于你：

- 对于简单的异步任务，回调是最直接的。

- 对于需要排序和并行化的更复杂任务，Promise 和 `async`/`await` 是你的朋友。

- 当 Promise 不够用时（例如，如果你要多次触发事件），请使用事件发射器或像 RxJS 这样的响应式流库。

- 要将这些技术扩展到多线程，请使用事件发射器、类型安全协议或类型安全 API（见["类型安全 API"]）。

练习

1. 实现一个通用的 `promisify` 函数，它可以接受任何只需一个参数和一个回调函数的函数，并将其包装为返回 promise 的函数。完成后，你应该能够像这样使用 `promisify`（首先安装 NodeJS 的类型声明，使用 `npm install @types/node --save-dev`）：

    ```
    ```

```
    import {readFile} from 'fs'

    let readFilePromise = promisify(readFile)
    readFilePromise('./myfile.ts')
      .then(result => console.log('success reading file',
result.toString()))
      .catch(error => console.error('error reading file', error))
    ```
```

2. 在["类型安全协议"]部分，我们推导了类型安全矩阵数学协议的一半。给定在主线程中运行的协议的这一半，请实现在 Web Worker 线程中运行的另一半。

3. 使用映射类型（如["在浏览器中：使用 Web Workers"]所示）为 NodeJS 的 `child_process` 实现类型安全的消息传递协议。

^[1]^ 嗯，如果你分叉你的浏览器平台，或者构建一个 C++ NodeJS 扩展的话，你可以这样做。

^[2]^ 眼尖的读者会注意到这个 API 与我们在["选项类型"]中开发的 `flatMap` API 是多么相似。这种相似性并非偶然！`Promise` 和 `Option` 都受到了函数式编程语言 Haskell 普及的单子(Monad)设计模式的启发。

^[3]^ `Observables` 是响应式编程处理随时间变化的值的方法的基本构建块。目前有一个正在进行中的提案要将 `Observables` 标准化，即 [`Observable` 提案](https://tc39.github.io/proposal-observable/)。期待在这本书的未来版本中深入了解 `Observables`，一旦该提案被 JavaScript 引擎更广泛地采用。

^[4]^ 除了函数、错误、DOM 节点、属性描述符、getter 和 setter，以及原型方法和属性。有关更多信息，请访问 [HTML5 规范](http://w3c.github.io/html/infrastructure.html#safe-passing-of-structured-data)。

^[5]^ 你也可以使用 `Transferable` API 通过引用在线程之间传递某些类型的数据（如 `ArrayBuffer`）。在本节中，我们不会使用 `Transferable` 来显式跨线程转移对象所有权，但这是一个实现细节。如果你在用例中使用 `Transferable`，从类型安全的角度来看，方法是相同的。

^[6]^ 这种实现是简单的，因为每次我们发出命令时它都会生成一个新的 worker；在现实世界中，你可能希望有一个池化机制来维护一个温暖的 worker 池，并回收已释放的 worker。

# 第9章. 前端和后端框架 {#chapter-9.-frontend-and-backend-frameworks .calibre12}
```

虽然你可以从头开始构建应用程序的每一部分——服务器上的网络和数据库层、前端的用户界面框架和状态管理解决方案——但你可能不应该这样做。很难把细节做对，幸运的是，前端和后端的许多这些难题已经被其他工程师解决了。通过利用现有的工具、库和框架来构建前端和后端的东西，我们可以快速迭代，并在构建自己的应用程序时站在稳定的基础上。

在本章中，我们将介绍一些解决客户端和服务器端常见问题的最流行的工具和框架。我们将讨论你可能会使用每个框架做什么，以及如何将其安全地集成到你的 TypeScript 应用程序中。

前端框架 {#frontend-frameworks .calibre13}

TypeScript 天然适合前端应用程序的世界。凭借其对 JSX 的丰富支持和安全建模可变性的能力，TypeScript 为你的应用程序提供了结构和安全性，使在前端开发这个快节奏环境中编写正确、可维护的代码变得更容易。

当然，所有内置的 DOM API 都是类型安全的。要在 TypeScript 中使用它们，只需在项目的 *tsconfig.json* 中包含它们的类型声明：

{ “compilerOptions”: { “lib”: [“dom”, “es2015”] } }

这将告诉 TypeScript 在类型检查你的代码时包含 *lib.dom.d.ts*——它的内置浏览器和 DOM 类型声明。

注意 {#note-27 .calibre22}

`lib` *tsconfig.json* 选项只是告诉 TypeScript 在处理项目中的代码时包含一组特定的类型声明；它不会发出任何额外的代码，或生成任何在运行时存在的 JavaScript。例如，它不会让 DOM 在 NodeJS 环境中神奇地工作（你的代码会编译，但在运行时会失败）——你需要确保你的类型声明与你的 JavaScript 环境在运行时实际支持的内容相匹配。跳到["构建你的 TypeScript 项目"]了解更多。

启用 DOM 类型声明后，你将能够安全地使用 DOM 和浏览器 API 来执行以下操作：

// 从全局 window 对象读取属性 let model = { url: window.location.href }

// 创建一个 [redacted] 元素 let input = document.createElement(‘input’)

// 给它添加一些 CSS 类 input.classList.add(‘Input’, ‘URLInput’)

// 当用户输入时，更新模型 input.addEventListener(‘change’, () => model.url = input.value.toUpperCase())

// 将 [redacted] 注入到 DOM 中 document.body.appendChild(input)

当然，所有这些代码都经过类型检查，并且带有常见的功能特性，如编辑器中的自动补全。例如，考虑这样的情况：

document.querySelector(‘.Element’).value // Error TS2339: Property ‘value’ does // not exist on type ‘Element’.

TypeScript 会抛出错误，因为 `querySelector` 的返回类型是可空的。

虽然对于简单的前端应用程序，这些底层的 DOM API 已经足够，能够为浏览器提供安全、类型引导的编程所需的功能，但大多数真实世界的前端应用程序使用框架来抽象 DOM 渲染和重新渲染、数据绑定以及事件的工作方式。以下部分将提供一些关于如何在几个最流行的浏览器框架中有效使用 TypeScript 的指导。

React

React 是目前最流行的前端框架之一，在类型安全方面是一个很好的选择。

React 如此安全的原因是 React 组件——React 应用程序的基本构建块——都是用 TypeScript 定义和使用的。这个特性在前端框架中很难找到，这意味着组件定义和使用者都经过类型检查。你可以使用类型来表达诸如"这个组件接受用户 ID 和颜色"或"这个组件只能将列表项作为子元素"之类的内容。然后 TypeScript 会强制执行这些约束，验证你的组件确实按照它们声明的方式工作。

围绕组件定义和使用者——前端应用程序的*视图层*——的这种安全性是杀手级特性。视图传统上是拼写错误、遗漏属性、参数类型错误和元素嵌套不当导致程序员集体花费数千小时抓狂和愤怒刷新浏览器的地方。当你开始使用 TypeScript 和 React 为视图添加类型时，就是你和你的团队在前端生产力翻倍的那一天。

JSX 入门

使用 React 时，你使用一种称为 *JavaScript XML*（JSX）的特殊 DSL 来定义视图，你将其直接嵌入到 JavaScript 代码中。它在你的 JavaScript 中看起来有点像 HTML。然后

你通过 JSX 编译器运行你的 JavaScript，将那种奇特的 JSX 语法重写为常规的 JavaScript 函数调用。

这个过程看起来像这样。假设你正在为朋友的餐厅构建菜单应用程序，你使用以下 JSX 列出早午餐菜单上的几个项目：

- Homemade granola with yogurt
- Fantastic french toast with fruit
- Tortilla Española with salad

通过 JSX 编译器（如 Babel 的 [`transform-react-jsx` 插件](http://bit.ly/2uENY4M)）运行该代码后，你将得到以下输出：

React.createElement(‘ul’, { ‘class’: ‘list’ }, React.createElement(‘li’, null, ‘Homemade granola with yogurt’), React.createElement(‘li’, null, ‘Fantastic French toast with fruit’), React.createElement(‘li’, null, ‘Tortilla Española with salad’));

TSC 标志：esModuleInterop

因为 JSX 编译为对 `React.createElement` 的调用，所以请确保将 React 库导入到使用 JSX 的每个文件中，以便在作用域中有一个名为 `React` 的变量：

import React from ‘react’

别担心——如果你忘记了，TypeScript 会警告你：

// Error TS2304: Cannot find name ‘React’.

还要注意，我在 *tsconfig.json* 中设置了 `{"esModuleInterop": true}` 以支持

不使用通配符（`*`）导入来导入 `React`。如果你在跟进，要么在你自己的 *tsconfig.json* 中启用 `esModuleInterop`，要么使用通配符导入：

import * as React from ‘react’

JSX 的好处是你可以编写看起来很像普通 HTML 的代码，然后自动编译成 JavaScript 引擎友好的格式。作为工程师，你只需使用熟悉的、高级的、声明式的 DSL，不必处理实现细节。

你不需要JSX来使用React（你可以直接编写编译后的代码，它会正常工作），你也可以在不使用React的情况下使用JSX（JSX标签编译成的特定函数调用——在前面的例子中是 `React.createElement`——是可配置的），但React与JSX的结合是神奇的，使编写视图变得非常有趣，而且非常安全。

TSX = JSX + TypeScript

包含JSX的文件使用*.jsx*文件扩展名。包含JSX的TypeScript文件使用*.tsx*扩展名。TSX之于JSX就像TypeScript之于JavaScript——一个编译时安全和辅助层，帮助你更高效地工作并产生更少错误的代码。要为你的项目启用TSX支持，将以下行添加到你的 *tsconfig.json*中：

{ “compilerOptions”: { “jsx”: “react” } }

在撰写本文时，`jsx`指令有三种模式：

`react`

: 将JSX编译为*.js*文件，使用JSX pragma(默认为`React.createElement`)。

`react-native`

: 保留JSX而不编译它，但生成*.js*扩展名的文件。

`preserve`

: 对JSX进行类型检查但不编译它，生成*.jsx*扩展名的文件。

在底层，TypeScript以可插拔的方式暴露了一些用于输入TSX的钩子(hooks)。这些是`global.JSX`命名空间上的特殊类型，TypeScript将其视为整个程序中TSX类型的真实来源。如果你只是使用React，则不需要深入到那么底层；但如果你正在构建自己的使用TSX（且不使用React）的TypeScript库——或者如果你好奇React类型声明是如何做到这一点的——请转到[附录G]。

在React中使用TSX

React让我们声明两种类型的组件：函数组件和类组件。两种类型的组件都接受一些属性并渲染一些TSX。从使用者的角度来看，它们是相同的。

声明和渲染函数组件如下所示：

import React from ‘react’

type Props = { isDisabled?: boolean size: ‘Big’ | ‘Small’ text: string onClick(event: React.MouseEvent): void }

export function FancyButton(props: Props) { const [toggled, setToggled] = React.useState(false) return <button className={‘Size-’ + props.size} disabled={props.isDisabled || false} onClick={event => { setToggled(!toggled) props.onClick(event) }} >{props.text} }

let button = <FancyButton size=‘Big’ text=‘Sign Up Now’ onClick={() => console.log(‘Clicked!’)} />

[![1](images/000000.png)]{#calibre_link-438 .calibre4}

: 我们必须将`React`变量引入当前作用域，以便在React中使用TSX。由于TSX编译为`React.createElement`函数调用，这意味着我们需要导入`React`以便在运行时定义它。

[![2](images/000001.png)]{#calibre_link-439 .calibre4}

: 我们首先声明可以传递给`FancyButton`组件的特定属性集。`Props`始终是对象类型，按约定命名为`Props`。对于我们的`FancyButton`组件，`isDisabled`是可选的，而其余属性是必需的。

[![3](images/000002.png)]{#calibre_link-440 .calibre4}

: React有自己的DOM事件包装器类型集。使用React事件时，请确保使用React的事件类型而不是常规DOM事件类型。

[![4](images/000003.png)]{#calibre_link-441 .calibre4}

: 函数组件只是一个常规函数，最多有一个参数（`props`对象）并返回React可渲染类型。React是宽容的，可以渲染广泛的类型范围：TSX、字符串、数字、布尔值、`null`和`undefined`。

[![5](images/000004.png)]{#calibre_link-442 .calibre4}

: 我们使用React的`useState`钩子为函数组件声明本地状态。`useState`是React中可用的少数钩子之一，你可以组合使用它们来创建自己的自定义钩子。注意，因为我们将初始值`false`传递给`useState`，TypeScript能够推断这个状态片段是`boolean`；如果我们使用了TypeScript无法推断的类型——例如数组——我们会明确绑定类型（例如，使用useState<number[]>([])）。

[![6](images/000005.png)]{#calibre_link-443 .calibre4}

: 我们使用TSX语法创建`FancyButton`的实例。`<FancyButton />`语法几乎与调用`FancyButton`相同，但它让React为我们管理`FancyButton`的生命周期。

就是这样。TypeScript强制执行：

- JSX格式良好。标签被关闭并正确嵌套，标签名称没有拼写错误。

- 当我们实例化一个 `<FancyButton />` 时，我们将所有必需的——以及任何可选的——属性传递给 `FancyButton`（`size`、`text` 和 `onClick`），并且这些属性都被正确地类型化了。

- 我们不向 `FancyButton` 传递任何无关的属性，只传递那些必需的。

类组件也类似：

import React from ‘react’ import {FancyButton} from ‘./FancyButton’

type Props = { firstName: string userId: string }

type State = { isLoading: boolean }
class SignupForm extends React.Component<Props, State> { state = { isLoading: false } render() {

return <>

Sign up for a 7-day supply of our tasty toothpaste now, {this.props.firstName}.

```
  <FancyButton
    isDisabled={this.state.isLoading}
    size='Big'
    text='Sign Up Now'
    onClick={this.signUp}
  />
</>
```

} private signUp = async () => { this.setState({isLoading: true}) try { await fetch(‘/api/signup?userId=’ + this.props.userId) } finally { this.setState({isLoading: false}) } } }

let form =

```
[![1](images/000000.png)]{#calibre_link-450 .calibre4}

:   和之前一样，我们导入 `React` 来将其引入作用域。

[![2](images/000001.png)]{#calibre_link-451 .calibre4}

:   和之前一样，我们声明一个 `Props` 类型来定义在创建 `<SignupForm />` 实例时需要传递的数据。

[![3](images/000002.png)]{#calibre_link-452 .calibre4}

:   我们声明一个 `State` 类型来建模组件的本地状态。

[![4](images/000003.png)]{#calibre_link-453 .calibre4}

:   要声明一个类组件，我们需要扩展 `React.Component` 基类。
```

[![5](images/000004.png)]{#calibre_link-454 .calibre4}

: 我们使用属性初始化器来声明本地状态的默认值。

[![6](images/000005.png)]{#calibre_link-455 .calibre4}

: 和函数组件一样，类组件的 `render` 方法返回一些可被 React 渲染的东西：TSX、字符串、数字、布尔值、`null` 或 [undefined]。

[![7](images/000006.png)]{#calibre_link-456 .calibre4}

: TSX 支持使用特殊的 `<>...</>` 语法创建片段。片段是一个无名的 TSX 元素，用来包装其他 TSX，是在需要返回单个 TSX 元素的地方避免渲染额外 DOM 元素的方法。例如，React 组件的 `render` 方法需要返回单个 TSX 元素；为了做到这一点，我们可以用 `<div>` 或任何其他元素包装我们的代码，但那会在渲染期间产生不必要的开销。

[![8](images/000007.png)]{#calibre_link-457 .calibre4}

: 我们使用箭头函数定义 `signUp`，以确保函数中的 `this` 不会被重新绑定。

[![9](images/000008.png)]{#calibre_link-458 .calibre4}

: 最后，我们实例化我们的 `SignupForm`。就像实例化函数组件时一样，我们也可以直接用 `new SignupForm({firstName: 'Albert', userId: '13ab9g3'})` 来 `new` 它，但那意味着 React 无法为我们管理 `SignupForm` 实例的生命周期。

注意我们如何在这个例子中混合搭配基于值的（`FancyButton`、`SignupForm`）和内置的（`section`、`h2`）组件。我们让 TypeScript 工作来验证以下事项：

- 所有必需的状态字段都在 `state` 初始化器中或在构造函数中被定义了

- 我们在 `props` 和 `state` 上访问的任何内容实际存在，并且是我们认为的类型

- 我们不直接写入 `this.state`，因为在 React 中，状态更新必须通过 `setState` API 进行

- 调用 `render` 确实返回一些 JSX

使用 TypeScript，你可以让你的 React 代码更安全，并因此成为一个更好、更快乐的人。

注意 {#note-28 .calibre22}

我们没有使用 React 的 `PropTypes` 功能，这是一种在运行时声明和检查属性类型的方法。由于 TypeScript 已经在编译时为我们检查类型，我们不需要再做一遍。

Angular 6/7 {#angular-67 .calibre17}

由 Shyam Seshadri 贡献

Angular 是一个功能更全面的前端框架，相比 React 而言，它不仅支持渲染视图，还支持发送和管理网络请求、路由和依赖注入。它从头开始构建以与 TypeScript 配合工作（实际上，框架本身就是用 TypeScript 编写的！）。

Angular 工作方式的核心是内置在 Angular CLI（Angular 的命令行实用程序）中的预先编译(Ahead-of-Time, AoT)编译器，它获取你通过 TypeScript 注解提供的类型信息，并使用该信息将你的代码编译为常规的 JavaScript。Angular 不是直接调用 TypeScript，而是在最终委托给 TypeScript 并将其编译为 JavaScript 之前，对你的代码应用一系列优化和转换。

让我们看看 Angular 如何使用 TypeScript 及其 AoT 编译器来使编写前端应用程序变得安全。

脚手架工具 {#scaffolding .calibre39}

要初始化一个新的Angular项目，首先需要使用NPM全局安装Angular CLI：

```
npm install @angular/cli -g
```

然后，使用Angular CLI初始化一个新的Angular应用：

```
ng new my-angular-app
```

按照提示操作，Angular CLI将为您设置一个基础的Angular应用程序。

在本书中，我们不会深入探讨Angular应用程序的结构，或如何配置和运行它。如需详细信息，请访问[Angular官方文档](https://angular.io/docs)。

组件 {#components .calibre39}

让我们构建一个Angular组件。Angular组件类似于React组件，包含描述组件DOM结构、样式和控制器的方式。在Angular中，您使用Angular CLI生成组件样板代码，然后手动填写详细信息。一个Angular组件由几个不同的文件组成：

- 一个模板，描述组件渲染的DOM

- 一组CSS样式

- 一个组件类，这是一个TypeScript类，用来控制组件的业务逻辑

让我们从组件类开始：

import {Component, OnInit} from '@angular/core'

@Component({ selector: 'simple-message', styleUrls: ['./simple-message.component.css'], templateUrl: './simple-message.component.html' }) export class SimpleMessageComponent implements OnInit { message: string ngOnInit() { this.message = 'No messages, yet' } }

在很大程度上，这是一个相当标准的TypeScript类，只有一些不同之处体现了Angular如何利用TypeScript。主要体现在：

- Angular的生命周期钩子(lifecycle hooks)作为TypeScript接口提供——只需声明您要`implement`的接口（`ngOnChanges`、`ngOnInit`等）。然后TypeScript会强制您实现符合所需生命周期钩子的方法。在这个示例中，我们实现了`OnInit`接口，这要求我们实现`ngOnInit`方法。

- Angular大量使用TypeScript装饰器(decorators)（参见["装饰器"]）来声明与您的Angular组件、服务和模块相关的元数据。在这个示例中，我们使用`selector`来声明人们如何使用我们的组件，并使用`templateUrls`和`styleUrl`将HTML模板和CSS样式表链接到我们的组件。

TSC标志：fullTemplateTypeCheck {#tsc-flag-fulltemplatetypecheck .calibre42}

要为您的Angular模板启用类型检查(typechecking)（您应该这样做！），请确保在您的*tsconfig.json*中启用`fullTemplateTypeCheck`：

{ “angularCompilerOptions”: { “fullTemplateTypeCheck”: true } }

请注意，`angularCompilerOptions`并不是为TSC指定选项。相反，它定义了特定于Angular的AoT编译器的编译器标志。

服务 {#services .calibre39}

Angular内置了依赖注入器(dependency injector, DI)，这是框架负责实例化服务并将它们作为参数传递给依赖它们的组件和服务的一种方式。这可以使实例化和测试服务及组件变得更容易。

让我们更新`SimpleMessageComponent`来注入一个依赖项`MessageService`，负责从服务器获取消息：

import {Component, OnInit} from ‘@angular/core’ import {MessageService} from ‘../services/message.service’

@Component({ selector: ‘simple-message’, templateUrl: ‘./simple-message.component.html’, styleUrls: [‘./simple-message.component.css’] }) export class SimpleMessageComponent implements OnInit { message: string constructor(private messageService: MessageService) {} ngOnInit() { this.messageService.getMessage().subscribe(response => this.message = response.message) } }

Angular的AoT编译器查看组件`constructor`接受的参数，提取出它们的类型（例如，`MessageService`），然后在相关的依赖注入器的依赖映射中搜索该特定类型的依赖项。然后它实例化该依赖项（`new`它）如果还没有被实例化，并将其传递给`SimpleMessageComponent`实例的构造函数。所有这些DI(Dependency Injection)内容都相当复杂，但当你的应用程序增长并且有多个依赖项时，这会很方便，你可能根据应用程序的配置方式（例如，`ProductionAPIService`与`DevelopmentAPIService`）或测试时（`MockAPIService`）使用这些依赖项。

现在让我们快速看一下如何定义一个服务：

import {Injectable} from ‘@angular/core’ import {HttpClient} from ‘@angular/common/http’

@Injectable({ providedIn: ‘root’ }) export class MessageService { constructor(private http: HttpClient) {} getMessage() { return this.http.get(‘/api/message’) } }

每当我们在Angular中创建一个服务时，我们再次使用TypeScript装饰器将其注册为可以`Injectable`的东西，并定义它是在应用程序的根级别还是在子模块中提供。在这里，我们注册了服务`MessageService`，允许我们在应用程序的任何地方注入它。在任何组件或服务的构造函数中，我们只需要请求一个`MessageService`，Angular会神奇地处理将其传递进来。

在了解了如何安全使用这两个流行的前端框架之后，让我们继续讨论前端和后端之间的接口类型化。

类型安全的API

由Nick Nance贡献

无论你决定使用哪些前端和后端框架，你都需要一种在机器之间安全通信的方式——从客户端到服务器、服务器到客户端、服务器到服务器以及客户端到客户端。

在这个领域有一些竞争的工具和标准。但在我们探索它们是什么以及如何工作之前，让我们思考一下我们如何构建自己的解决方案，以及它可能有什么好处和缺点（毕竟我们是工程师）。

我们想要解决的问题是：虽然我们的客户端和服务器可能是100%类型安全的——安全的堡垒——但在某个时候它们需要通过HTTP、TCP或其他基于套接字的协议等无类型网络协议相互通信。我们如何使这种通信类型安全？

一个好的起点可能是我们在"类型安全协议"中开发的类型安全协议。它可能看起来像这样：

type Request = | {entity: ‘user’, data: User} | {entity: ‘location’, data: Location}

// client.ts async function get(entity: R[‘entity’]): Promise<R[‘data’]> { let res = await fetch(/api/${entity}) let json = await res.json() if (!json) { throw ReferenceError(‘Empty response’) } return json }

// app.ts async function startApp() { let user = await get(‘user’) // User }

你可以构建相应的`post`和`put`函数来写回你的REST API，并为你的服务器支持的每个实体添加一个类型。在后端，你然后会为每种实体类型实现相应的一组处理器，从你的数据库读取以返回客户端请求的任何实体。

但是如果你的服务器不是用TypeScript编写的，或者你无法在客户端和服务器之间共享你的`Request`类型（导致两者随着时间推移而不同步），或者你不使用REST（也许你使用GraphQL代替）会怎么样？或者如果你有其他客户端要支持，比如iOS上的Swift客户端或Android上的Java客户端？

这就是类型化、代码生成的API发挥作用的地方。它们有很多种类，每种都有许多语言（包括TypeScript）的可用库——例如：

- 用于RESTful API的[Swagger](https://github.com/swagger-api/swagger-codegen)

- 用于GraphQL的[Apollo](https://www.npmjs.com/package/apollo)和[Relay](https://facebook.github.io/relay/)

- 用于RPC的[gRPC](https://grpc.io/)和[Apache Thrift](https://thrift.apache.org/)

这些工具依赖于服务器和客户端的共同真实来源——Swagger的数据模型、Apollo的GraphQL模式(schemas)、gRPC的Protocol Buffers——然后编译成你可能使用的任何语言的特定语言绑定（在我们的情况下，那就是TypeScript）。

这种代码生成是防止你的客户端和服务器（或多个客户端）彼此不同步的方法；由于每个平台都共享一个通用模式(schema)，你不会遇到这样的情况：你更新了你的iOS应用以支持一个字段，但忘记在你的拉取请求上按合并以添加服务器对它的支持。

深入了解每个框架的详细信息超出了本书的范围。为你的项目选择一个，然后前往其文档了解更多。

后端框架

当你构建一个与数据库交互的应用程序时，你可能会从原始SQL或API调用开始，这些本质上是无类型的：

// PostgreSQL，使用 node-postgres let client = new Client let res = await client.query(‘SELECT name FROM users where id = $1’, [739311]) // any

// MongoDB，使用 node-mongodb-native db.collection（‘users’）.find({id: 739311}) .toArray((err, user) => // user 是 any）

通过一些手动类型标注，你可以使这些API更安全，并摆脱大部分的`any`：

db.collection（‘users’）.find({id: 739311}) .toArray((err, user: User) => // user 是 any）

然而，原始SQL API仍然相当底层，很容易使用错误的类型，或者忘记类型而意外地得到`any`。

这就是*对象关系映射器*(ORMs)发挥作用的地方。ORM从你的数据库模式生成代码，为你提供高级API来表达查询、更新、删除等操作。在静态类型语言中，这些API是类型安全的，所以你不必担心正确地输入类型和手动绑定泛型类型参数。

当从TypeScript访问你的数据库时，请考虑使用ORM。在撰写本文时，Umed Khudoiberdiev的优秀的[TypeORM](https://www.npmjs.com/package/typeorm)是TypeScript最完整的ORM，支持MySQL、PostgreSQL、Microsoft SQL Server、Oracle，甚至MongoDB。使用TypeORM，你获取用户姓名的查询可能看起来像这样：

let user = await UserRepository .findOne({id: 739311}) // User | undefined

注意这个高级API，它既安全（防止SQL注入攻击等）又默认类型安全（我们知道`findOne`返回什么类型而无需手动注释）。在处理数据库时总是使用ORM——它更方便，会让你免于凌晨四点被电话吵醒，因为`saleAmount`字段是`null`，因为你昨晚把它更新为[`orderAmount`]，而你的同事在你外出时决定为你运行数据库迁移，预期你的拉取请求会落地，但是在午夜时分你的拉取请求失败了，即使迁移成功了，你在纽约的销售团队醒来发现所有客户的订单都是正好`null`美元（这发生在...一个朋友身上）。

总结

在本章中我们涵盖了很多内容：直接操作DOM；使用React和Angular；使用Swagger、gRPC和GraphQL等工具为你的API添加类型安全；以及使用TypeORM安全地与数据库交互。

JavaScript框架变化迅速，到你阅读这本书时，这里描述的特定API和框架可能正在成为博物

馆展品。使用你对*类型安全框架解决什么问题*的新直觉，来识别可以利用他人工作使你的代码更安全、更抽象、更模块化的地方。从本章中要带走的重要思想不是2019年使用的最佳框架是什么，而是什么样的问题可以用框架更好地解决。

通过类型安全的UI代码、类型化API层和类型安全后端的组合，你可以从应用程序中消除整类错误，并因此睡得更安稳。

第10章. 命名空间.模块

当你编写程序时，你可以在几个层面表达封装。在最低层面，函数封装行为，像对象和列表这样的数据结构封装数据。然后你可能将函数和数据分组到类中，或者将它们作为命名空间实用工具分开，为你的数据使用单独的数据库或存储。每个文件一个类或一组实用工具是典型的。更进一步，你可能将几个类或实用工具分组到一个包中，发布到NPM。

当我们谈论模块时，重要的是要区分编译器（TSC）如何解析模块、构建系统（Webpack、Gulp等）如何解析模块，以及模块如何在运行时实际加载到应用程序中（`<script />` 标签、SystemJS等）。在JavaScript世界中，通常有不同的程序来完成这些工作，这可能使模块难以理解。CommonJS和ES2015模块标准使三个程序之间的互操作变得更容易，而像Webpack这样强大的打包器有助于抽象化底层发生的三种解析。

在本章中，我们将专注于这三种程序中的第一种：TypeScript如何解析和编译模块。我们将在[第12章]中讨论构建系统和运行时加载器如何与模块协作，这里我们将讨论：

- 命名空间和模块化代码的不同方式

- 导入和导出代码的不同方式

- 随着代码库增长扩展这些方法

- 模块模式与脚本模式

- 什么是声明合并(declaration merging)，以及你可以用它做什么

但首先，让我们了解一些背景知识。

JavaScript模块简史 {#a-brief-history-of-javascript-modules .calibre13}

由于TypeScript编译为JavaScript并与之互操作，它必须支持JavaScript程序员使用的各种模块标准。

在最初（1995年），JavaScript不支持任何类型的模块系统。没有模块，一切都在全局命名空间中声明，这使得构建和扩展应用程序变得非常困难。你可能很快就会用完变量名，并遇到变量

名冲突；没有为每个模块暴露显式的API(Application Programming Interface)，很难知道你应该使用模块的哪些部分，哪些部分是私有实现细节。

为了帮助解决这些问题，人们使用对象或*立即调用函数表达式*(IIFEs)来模拟模块，将它们分配给全局`window`，使它们可供应用程序中的其他模块（以及托管在同一网页上的其他应用程序）使用。它看起来像这样：

window.emailListModule = { renderList() {} // … }

window.emailComposerModule = { renderComposer() {} // … }

window.appModule = { renderApp() { window.emailListModule.renderList() window.emailComposerModule.renderComposer() } }

由于加载和运行JavaScript会阻塞浏览器的UI，随着Web应用程序的增长并包含越来越多的代码行，用户的浏览器会变得越来越慢。出于这个原因，聪明的程序员开始在页面加载后动态加载JavaScript，而不是一次性全部加载。在JavaScript首次发布近10年后，Dojo（Alex Russell，2004）、YUI（Thomas Sha，2005）和LABjs（Kyle Simpson，2009）发布了模块加载器——在初始页面加载完成后延迟（通常是异步）加载JavaScript代码的方法。延迟和异步模块加载意味着三件事：

1. 模块需要良好封装。否则，在依赖项流入时页面可能会损坏。

2. 模块之间的依赖关系需要明确。否则，我们不知道需要加载哪些模块以及按什么顺序加载。

3. 每个模块在应用程序中都需要一个唯一标识符。否则，就没有可靠的方法来指定需要加载哪些模块。

使用LABjs加载模块看起来像这样：

$LAB .script('/emailBaseModule.js').wait() .script('/emailListModule.js') .script('/emailComposerModule.js')

大约在同一时间，NodeJS（Ryan Dahl，2009）正在开发中，其创建者从JavaScript的成长痛苦和其他语言中吸取了教训，决定将模块系统直接构建到平台中。像任何好的模块系统一样，

它需要满足与LABjs和YUI加载器相同的三个标准。NodeJS通过CommonJS模块标准做到了这一点，它看起来像这样：

// emailBaseModule.js var emailList = require(‘emailListModule’) var emailComposer = require(‘emailComposerModule’)

module.exports.renderBase = function() { // … }

与此同时，在Web上，AMD模块标准（James Burke，2008）——由Dojo和RequireJS推动——正在兴起。它支持一组等效的功能，并带有自己的构建系统来打包JavaScript代码：

define(‘emailBaseModule’, [‘require’, ‘exports’, ‘emailListModule’, ‘emailComposerModule’], function(require, exports, emailListModule, emailComposerModule) { exports.renderBase = function() { // … } })

几年之后，Browserify(Browserify) 发布了（James Halliday，2011年），为前端工程师提供了在前端也使用 CommonJS 的能力。CommonJS 成为模块打包和导入/导出语法的事实标准。

CommonJS 的处理方式存在一些问题。其中，`require` 调用必须是同步的，并且 CommonJS 模块解析算法并不适合在 Web 上使用。除此之外，在某些情况下使用它的代码无法进行静态分析（作为 TypeScript 程序员，这应该引起你的注意），因为 `module.exports` 可以出现在任何地方（甚至在永远不会实际到达的死代码分支中），而 `require` 调用可以出现在任何地方并包含任意字符串和表达式，这使得静态链接 JavaScript 程序变得不可能，也无法验证所有引用的文件是否真的存在并导出它们声称导出的内容。

在这种背景下，ES2015——ECMAScript 语言的第六版——引入了一个新的导入和导出标准，它具有简洁的语法并且可以进行静态分析。它看起来是这样的：

// emailBaseModule.js import emailList from ‘emailListModule’ import emailComposer from ‘emailComposerModule’

export function renderBase() { // … }

这是我们今天在 JavaScript 和 TypeScript 代码中使用的标准。然而，在撰写本文时，该标准尚未在每个 JavaScript 运行时中得到原生支持，因此我们必须将其编译为受支持的格式（NodeJS 环境使用 CommonJS，浏览器环境使用全局变量或模块可加载格式）。

TypeScript 为我们提供了几种在模块中使用和导出代码的方式：使用全局声明、使用标准 ES2015 的 `import` 和 `export`，以及使用与 CommonJS 模块向后兼容的 `import`。除此之外，TSC 的构建系统让我们可以为各种环境编译模块：全局变量、ES2015、CommonJS、AMD、SystemJS 或 UMD（CommonJS、AMD 和全局变量的混合——无论消费者环境中恰好可用的是什么）。

import, export

除非你被狼追赶，否则你应该在 TypeScript 代码中使用 ES2015 的 `import` 和 `export`，而不是使用 CommonJS、全局或命名空间模块。它们看起来是这样的——与普通的 JavaScript 相同：

// a.ts export function foo() {} export function bar() {}

// b.ts import {foo, bar} from ‘./a’ foo() export let result = bar()

ES2015 模块标准支持默认导出：

// c.ts export default function meow(loudness: number) {}

// d.ts import meow from ‘./c’ // 注意缺少{花括号} meow(11)

它还支持使用通配符导入（`*`）从模块导入所有内容：

// e.ts import * as a from ‘./a’ a.foo() a.bar()

以及重新导出模块的部分（或全部）导出：

// f.ts export * from ‘./a’ export {result} from ‘./b’ export meow from ‘./c’

因为我们编写的是 TypeScript 而不是 JavaScript，所以我们当然可以导出类型和接口以及值。而且因为类型和值存在于不同的命名空间中，所以导出两个共享相同名称的东西——一个在值级别，一个在类型级别——是完全可以的。就像任何其他代码一样，当你实际使用它时，TypeScript 会推断你指的是类型还是值：

// g.ts export let X = 3 export type X = {y: string}

// h.ts import {X} from ‘./g’

let a = X + 1 // X 指的是值 X let b: X = {y: ‘z’} // X 指的是类型 X

模块路径是文件系统上的文件名。这将模块与它们在文件系统中的布局方式耦合起来，但对于需要了解该布局以便将模块名称解析为文件的模块加载器来说，这是一个重要特性。

动态导入

随着应用程序变得越来越大，初始渲染的时间也会变得越来越糟。这对于网络可能成为瓶颈的前端应用程序来说尤其是一个问题，但它也适用于后端应用程序，因为当你在顶层导入更多代码时，启动时间会变长——这些代码需要从文件系统加载、解析、编译和执行，这一切都会阻塞其他代码的运行。

在前端，解决这个问题的一种方法（除了写更少的代码！）是使用*代码分割*：将你的代码分块成一堆生成的 JavaScript 文件，而不是将所有内容打包在一个大文件中。通过分割，你可以获得并行加载多个块的好处，这减轻了大型网络请求的负担（参见图10-1）。

<figure class="calibre33">

<h6 id="figure-10-1.-network-waterfall-for-javascript-loaded-from-facebook.com" class="calibre34">图 10-1. 从 facebook.com 加载的 JavaScript 网络瀑布图</h6>

</figure>

进一步的优化是在真正需要时才懒加载代码块。真正大型的前端应用程序——比如Facebook和Google的那些——作为常规做法使用这种优化。没有它，客户端可能在初始页面加载时就要加载数GB的JavaScript代码，这可能需要几分钟或几小时（更不用说一旦人们收到手机账单后可能会停止使用这些服务）。

懒加载对于其他原因也很有用。例如，流行的[Moment.js](https://momentjs.com)日期操作库(date manipulation library)附带支持世界各地使用的每种日期格式的包，按语言环境分割。每个包大约重3KB。为每个用户加载所有这些语言环境可能是不可接受的性能和带宽消耗；相反，你可能想要检测用户的语言环境，然后只加载相关的日期包。

LABjs及其同类产品引入了在实际需要时才懒加载代码的概念，这个概念在*动态导入(dynamic imports)*中被正式化。它看起来像这样：

let locale = await import(‘locale_us-en’)

你可以将`import`用作语句来静态拉取代码（正如我们到目前为止一直使用的），或者用作返回模块`Promise`的函数（正如我们在这个例子中所做的）。

虽然你可以向`import`传递一个评估为字符串的任意表达式，但这样做会失去类型安全性。要安全地使用动态导入，请确保：

1. 直接向`import`传递字符串字面量，而不是先将字符串赋值给变量。

2. 向`import`传递表达式并手动标注模块的签名。

如果使用第二个选项，一个常见的模式是静态导入模块，但只在类型位置使用它，这样TypeScript就会编译掉静态导入（要了解更多，请参见["The types Directive"]）。例如：

import {locale} from ‘./locales/locale-us’

async function main() { let userLocale = await getUserLocale() let path = ./locales/locale-${userLocale} let localeUS: typeof locale = await import(path) }

我们从*./locales/locale-us*导入了`locale`，但我们只将其用于类型，我们通过`typeof locale`获取了这个类型。我们需要这样做是因为TypeScript无法静态查找`import(path)`的类型，因为`path`是一个计算变量而不是静态字符串。因为我们从未将`locale`用作值，而只是搜刮它的类型，所以TypeScript编译掉了静态导入（在这个例子中，TypeScript根本不生成任何顶层导出），为我们提供了出色的类型安全性和动态计算的导入。

TSC 设置: module

TypeScript仅在`esnext`模块模式中支持动态导入。要使用动态导入，在你的*tsconfig.json*的`compilerOptions`中设置`{"module": "esnext"}`。跳转到["Running TypeScript on the Server"]和["Running TypeScript in the Browser"]了解更多。

使用 CommonJS 和 AMD 代码

当消费使用 CommonJS 或 AMD 标准的 JavaScript 模块时，你可以简单地从中导入名称，就像对 ES2015 模块一样：

import {something} from ‘./a/legacy/commonjs/module’

默认情况下，CommonJS 默认导出与 ES2015 默认导入不兼容；要使用默认导出，你必须使用通配符导入：

import * as fs from ‘fs’ fs.readFile(‘some/file.txt’)

为了更顺畅地互操作，在你的 *tsconfig.json* 的 `compilerOptions` 中设置`{"esModuleInterop": true}`。现在，你可以省略通配符：

import fs from ‘fs’ fs.readFile(‘some/file.txt’)

注意

正如我在本章开头提到的，即使这段代码能够编译，这并不意味着它在运行时就能工作。无论你使用哪种模块标准——`import`/`export`、CommonJS、AMD、UMD 或浏览器全局变量——你的模块打包器和模块加载器都必须了解该格式，以便它们能够在编译时正确地打包和拆分你的代码，并在运行时正确地加载你的代码。前往第 12 章了解更多信息。

模块模式与脚本模式

TypeScript 使用两种模式之一来解析你的每个 TypeScript 文件：*模块模式*或*脚本模式*。它基于一个简单的启发式规则来决定使用哪种模式：你的文件是否包含任何 `import` 或 `export`？如果有，则使用模块模式；否则，使用脚本模式。

模块模式是我们到目前为止一直使用的，也是你大部分时间会使用的。在模块模式中，你使用 `import` 和 `import()` 来从其他文件中引入代码，使用 `export` 来让其他文件可以访问代码。如果你使用任何第三方 UMD 模块（提醒一下，UMD 模块试图使用 CommonJS、RequireJS 或浏览器全局变量，取决于环境支持哪种），你必须首先 `import` 它们，不能直接使用它们的全局导出。

在脚本模式中，你声明的任何顶级变量都可以在项目中的其他文件中使用，而无需显式导入，并且你可以安全地使用来自第三方 UMD 模块的全局导出，而无需首先显式导入它们。脚本模式的几个用例包括：

- 快速原型化浏览器代码，你计划编译为完全没有模块系统（在你的 *tsconfig.json* 中设置 `{"module": "none"}`）并作为原始 `<script />` 标签包含在你的 HTML 文件中。

- 创建类型声明（参见"类型声明"）

你几乎总是想要坚持使用模块模式，当你编写真实世界的代码时，TypeScript 会自动为你选择模块模式，这些代码会 `import` 其他代码并 `export` 内容供其他文件使用。

命名空间

TypeScript 给我们另一种封装代码的方式：`namespace` 关键字。命名空间对于很多 Java、C#、C++、PHP 和 Python 程序员来说会很熟悉。

提示

如果你来自具有命名空间的语言，请注意虽然 TypeScript 支持命名空间，但它们不是封装代

码的首选方式；如果你不确定是使用命名空间还是模块，请选择模块。

命名空间抽象了文件在文件系统中如何布局的具体细节；你不必知道你的 `.mine` 函数位于 `schemes/scams/bitcoin/apps` 文件夹中，相反你可以使用一个简短、方便的命名空间如 `Schemes.Scams.Bitcoin.Apps.mine` 来访问它。

假设我们有两个文件——一个用于发出 HTTP GET 请求的模块，和一个使用该模块发出请求的消费者：

// Get.ts namespace Network { export function get(url: string): Promise { // … } }

// App.ts namespace App { Network.get(‘https://api.github.com/repos/Microsoft/typescript’) }

命名空间必须有一个名称（如 `Network`），它可以导出函数、变量、类型、接口或其他命名空间。`namespace` 块中任何没有显式导出的代码都是该块的私有代码。由于命名空间可以导出命名空间，您可以轻松地建模嵌套命名空间。假设我们的 `Network` 模块变得很大，我们想将它拆分成几个子模块。我们可以使用命名空间来做到这一点：

namespace Network { export namespace HTTP { export function get (url: string): Promise { // … } } export namespace TCP { listenOn(port: number): Connection { //… } // … } export namespace UDP { // … } export namespace IP { // … } }

现在，我们所有与网络相关的实用程序都在 `Network` 下的子命名空间中。例如，我们现在可以从任何文件中调用 `Network.HTTP.get` 和 `Network.TCP.listenOn`。与接口一样，命名空间可以被增强，这使得在文件间拆分它们变得很方便。TypeScript 会为我们递归合并同名的命名空间：

// HTTP.ts namespace Network { export namespace HTTP { export function get(url: string): Promise { // … } } }

// UDP.ts namespace Network { export namespace UDP { export function send(url: string, packets: Buffer): Promise { // … } } }

// MyApp.ts Network.HTTP.get<Dog[]>(‘http://url.com/dogs’) Network.UDP.send(‘http://url.com/cats’, new Buffer(123))

如果您最终有很长的命名空间层级，您可以使用*别名*来缩短它们以方便使用。注意，尽管语法相似，但别名不支持解构（就像导入 ES2015 模块时所做的那样）：

// A.ts namespace A { export namespace B { export namespace C { export let d = 3 } } }

// MyApp.ts import d = A.B.C.d

let e = d * 3

冲突

不允许同名导出之间发生冲突：

// HTTP.ts namespace Network { export function request(url: string): T { // … } }

// HTTP2.ts namespace Network { // Error TS2393: Duplicate function implementation. export function request(url: string): T { // … } }

无冲突规则的例外是重载的环境函数声明(overloaded ambient function declarations)，您可以使用它来细化函数类型：

// HTTP.ts namespace Network { export function request(url: string): T }

// HTTP2.ts namespace Network { export function request(url: string, priority: number): T }

// HTTPS.ts namespace Network { export function request(url: string, algo: ‘SHA1’ | ‘SHA256’): T }

编译输出

与导入和导出不同，命名空间不遵循您的 *tsconfig.json* 的 `module` 设置，并且总是编译为全局变量。让我们揭开面纱，看看生成的输出是什么样子的。假设我们有以下模块：

// Flowers.ts namespace Flowers { export function give(count: number) { return count + ' flowers' } }

通过 `TSC` 运行它，生成的 JavaScript 输出如下所示：

let Flowers (function (Flowers) { function give(count) { return count + ' flowers' } Flowers.give = give })(Flowers || (Flowers = {}))

[![1](images/000000.png)]{#calibre_link-475 .calibre4}

: `Flowers` 在 IIFE（立即调用函数表达式）中声明——一个立即调用自身的函数——以创建闭包并防止未显式导出的变量从 `Flowers` 模块中泄漏出去。

[![2](images/000001.png)]{#calibre_link-476 .calibre4}

: TypeScript 将我们导出的 `give` 函数分配给 `Flowers` 命名空间。

[![3](images/000002.png)]{#calibre_link-477 .calibre4}

: 如果 `Flowers` 命名空间已经全局定义，那么 TypeScript 会增强它（`Flowers`）；否则，TypeScript 创建并增强新创建的命名空间（`Flowers = {}`）。

尽可能优先使用模块而非命名空间

优先使用常规模块（`import` 和 `export` 类型）而非命名空间，这样可以更紧密地遵循 JavaScript 标准并使您的依赖关系更加明确。

显式依赖对于可读性、强制模块隔离（因为命名空间会自动合并，但模块不会）和静态分析有很多好处，这对于大型前端项目非常重要，因为在这些项目中，剔除死代码并将编译后的代码拆分

为多个文件对于性能至关重要。

在NodeJS环境中运行TypeScript程序时，模块也是明确的选择，因为NodeJS对CommonJS有内置支持。在浏览器环境中，一些程序员为了简单而偏好命名空间，但对于中大型项目，尽量坚持使用模块而不是命名空间。

声明合并

到目前为止，我们已经接触了TypeScript为我们进行的三种类型的合并：

- 合并值和类型，使得同一个名称可以引用值或类型，取决于我们如何使用它（参见"伴生对象模式"）

- 将多个命名空间合并为一个

- 将多个接口合并为一个（参见"声明合并"）

正如您可能已经直觉到的，这些是更通用的TypeScript行为的三个特例。TypeScript有一套丰富的行为来合并不同种类的名称，解锁了各种其他方式难以表达的模式（参见表10-1）。

从 \ 到	值	类	枚举	函数	类型别名	接口	命名空间	模块
值	否	否	否	否	是	是	否	---
类	---	否	否	否	否	是	是	---
枚举	---	---	是	否	否	否	是	---
函数	---	---	---	否	是	是	是	---
类型别名	---	---	---	---	否	否	是	---
接口	---	---	---	---	---	是	是	---
命名空间	---	---	---	---	---	---	是	---
模块	---	---	---	---	---	---	---	是

表10-1. 声明可以合并吗?

这意味着，例如，如果您在同一作用域中声明一个值和一个类型别名，TypeScript将允许这样做，并根据您在值位置还是类型位置使用该名称来推断您的意思——类型还是值。这就是让我们能够实现"伴生对象模式"中描述的模式的原因。这也意味着您可以使用接口和命名空间来实现伴生对象——您不仅限于值和类型别名。或者您可以利用模块合并来增强第三方模块声明（更多内容请参见"扩展模块"）。或者您可以通过将枚举与命名空间合并来向枚举添加静态方法（试试看！）。

moduleResolution标志

眼尖的读者可能会注意到在*tsconfig.json*中可用的`moduleResolution`标志。该标志控制TypeScript用于解析应用程序中模块名称的算法。该标志支持两种模式：

- `node`：始终使用此模式。它使用与NodeJS相同的算法来解析模块。以`.`、`/`或`~`为前缀的模块（如`./my/file`）从本地文件系统解析，要么相对于当前文件，要么使用绝对路径（相对于您的*/*目录，或您的*tsconfig.json*的`baseUrl`设置），具体取决于您使用的前缀。TypeScript从您的*node_modules*文件夹加载没有前缀的模块路径，与NodeJS相同。TypeScript在两个方面基于NodeJS的解析策略：

  1. 除了NodeJS查看的包的*package.json*中的`main`字段来找到目录中的默认可导入文件外，TypeScript还查看TypeScript特定的`types`属性（更多内容请参见"JavaScript的类型查找"）。

  2. 当导入未指定扩展名的文件时，TypeScript首先查找具有该名称和*.ts*扩展名的文件，然后是*.tsx*、*.d.ts*，最后是*.js*。

- `classic`：永远不要使用这种模式。在此模式下，相对路径的解析方式与 `node` 模式相同，但对于无前缀的名称，TypeScript 会在当前文件夹中查找具有给定名称的文件，然后逐个文件夹向上遍历目录树，直到找到匹配的文件。对于来自 NodeJS 或 JavaScript 世界的任何人来说，这种行为都非常令人意外，并且与其他构建工具的互操作性很差。

总结

在本章中，我们涵盖了 TypeScript 的模块系统，从 JavaScript 模块系统的简史开始，包括 ES2015 模块和使用动态导入安全地延迟加载代码，与 CommonJS 和 AMD 模块的互操作，以及模块模式与脚本模式。然后我们涵盖了命名空间、命名空间合并，以及 TypeScript 的声明合并是如何工作的。

在使用 TypeScript 开发应用程序时，请尽量坚持使用 ES2015 模块。TypeScript 不关心你使用哪种模块系统，但它会使与构建工具的集成变得更容易（参见第 12 章了解更多）。

练习

1. 尝试声明合并，以便：

   1. 使用命名空间和接口（而不是值和类型）重新实现伴生对象（来自["伴生对象模式"]）。

   2. 为枚举添加静态方法。

^[1]^ 我真的希望这个玩笑能经得起时间考验，并且不会后悔没有投资比特币。

第 11 章：与 JavaScript 互操作

我们生活的世界并不完美。你的咖啡可能太热，喝的时候会稍微烫伤嘴巴，你的父母可能会打电话给你，留言频率有点太高，无论你给市政府打多少次电话，你家车道旁的那个坑洞仍然在那里，你的代码可能没有完全被静态类型覆盖。

我们大多数人都处于这种情况：尽管偶尔你可能有机会在 TypeScript 中启动一个全新项目，但大多数时候它会作为一个小的安全岛屿开始，嵌入在一个更大、不太安全的代码库中。也许你有一个隔离良好的组件，想要在上面尝试 TypeScript，即使你的公司在其他地方都使用常规的 ES6 JavaScript，或者你厌倦了在早上 6 点被唤醒，因为你重构了一些代码却忘记更新调用点（现在是早上 7 点，你正在同事醒来之前匆忙将 TSC 合并到代码库中）。无论哪种方式，你可能会从无类型海洋中的 TypeScript 岛屿开始。

到目前为止，在本书中我一直在教你以正确的方式编写 TypeScript。本章是关于以实用的方式编写 TypeScript，在正在从无类型语言迁移的真实代码库中，使用第三方 JavaScript 库，有时为了快速热修补以解决生产问题而牺牲类型安全性。本章致力于与 JavaScript 一起工作。我们将探索：

- 使用类型声明

- 从 JavaScript 逐步迁移到 TypeScript

- 使用第三方 JavaScript 和 TypeScript

类型声明

*类型声明*是扩展名为 *.d.ts* 的文件。与 JSDoc 注释（参见["步骤 2b：添加 JSDoc 注释（可选）"]）一起，它是为原本无类型的 JavaScript 代码附加 TypeScript 类型的一种方式。

类型声明的语法与常规 TypeScript 相似，但有一些差异：

- 类型声明只能包含类型，不能包含值。这意味着没有函数、类、对象或变量实现，也没有参数的默认值。

- 虽然类型声明不能定义值，但它们可以声明在你的 JavaScript 中某处*存在*一个已定义的值。我们为此使用特殊的 `declare` 关键字。

- 类型声明只为对使用者可见的内容声明类型。我们不包括未导出的类型或函数体内局部变量的类型等内容。

让我们进入一个例子，看看一段 TypeScript (*.ts*) 代码及其等效的类型声明 (*.d.ts*)。这个例子是来自流行的 RxJS 库的一段相当复杂的代码；可以忽略它具体做什么的细节，而是注意它使用的语言特性（导入、类、接口、类字段、函数重载等等）：

import {Subscriber} from ‘./Subscriber’ import {Subscription} from ‘./Subscription’ import {PartialObserver, Subscribable, TeardownLogic} from ‘./types’

export class Observable implements Subscribable { public _isScalar: boolean = false constructor(subscribe?: (this: Observable, subscriber: Subscriber) => TeardownLogic) { if (subscribe) { this._subscribe = subscribe } }

static create(subscribe?: (subscriber: Subscriber) => TeardownLogic) { return new Observable(subscribe) } subscribe(observer?: PartialObserver): Subscription subscribe(next?: (value: T) => void, error?: (error: any) => void, complete?: () => void): Subscription subscribe(observerOrNext?: PartialObserver | ((value: T) => void), error?: (error: any) => void, complete?: () => void): Subscription { // … } }

通过启用 `declarations` 标志 (`tsc -d Observable.ts`) 运行这段代码的TSC编译器，会生成以下 *Observable.d.ts* 类型声明：

import {Subscriber} from ‘./Subscriber’ import {Subscription} from ‘./Subscription’ import {PartialObserver, Subscribable, TeardownLogic} from ‘./types’

export declare class Observable implements Subscribable { _isScalar: boolean constructor(subscribe?: (this: Observable, subscriber: Subscriber) => TeardownLogic); static create(subscribe?: (subscriber: Subscriber) => TeardownLogic): Observable subscribe(observer?: PartialObserver): Subscription subscribe(next?: (value: T) => void, error?: (error: any) => void, complete?: () => void): Subscription }

[![1](images/000000.png)]{#calibre_link-487 .calibre4}

: 注意 `class` 前的 `declare` 关键字。我们实际上不能在类型声明中定义一个类，但我们可以*声明*我们在 *.d.ts* 文件对应的JavaScript文件中定义了一个类。将 `declare` 想象成一个确认："我保证我的JavaScript导出了这种类型的类。"

[![2](images/000001.png)]{#calibre_link-488 .calibre4}

: 因为类型声明不包含实现，我们只保留 `subscribe` 的两个重载，而不保留其实现的签名。

注意 *Observable.d.ts* 就是去掉实现的 *Observable.ts*。换句话说，它只是 *Observable.ts* 中的类型。

这个类型声明对于使用 *Observable.ts* 的RxJS库中的其他文件并不有用，因为它们可以直接访问 *Observable.ts* 源TypeScript文件并直接使用它。但是，如果你在TypeScript应用程序中使用RxJS，它就很有用了。

想想看：如果RxJS的作者想要在NPM上为他们的TypeScript用户打包类型信息（RxJS既可以在TypeScript应用程序中使用，也可以在JavaScript应用程序中使用），他们有两个选择：打包源TypeScript文件（给TypeScript用户）和编译后的JavaScript文件（给JavaScript用户），或者发布编译后的JavaScript文件和给TypeScript用户的类型声明。后者减少了文件大小，并明确了要使用的正确导入。它还有助于保持应用程序的快速编译时间，因为你的TSC实例不必在每次编译自己的应用程序时重新编译RxJS（实际上，这就是我们在"项目引用"中介绍的优化策略有效的原因！）。

类型声明文件有几个用途：

1. 当其他人在他们的TypeScript应用程序中使用你编译后的TypeScript时，他们的TSC实例将查找与你生成的JavaScript文件对应的 *.d.ts* 文件。这告诉TypeScript你的项目的类型是什么。

2. 具有TypeScript支持的代码编辑器（如VSCode）将读取这些 *.d.ts* 文件，在用户键入时为他们提供有用的类型提示，即使他们不使用TypeScript。

3. 它们通过避免不必要地重新编译你的TypeScript代码来显著加快编译时间。

类型声明是告诉TypeScript的一种方式："存在这个在JavaScript中定义的东西，我将向你描述它。"当我们谈论类型声明时，我们经常称它们为*环境的*，以便将它们与包含值的常规声明区分开来；例如，*环境变量声明*使用 `declare` 关键字来声明变量在JavaScript中的某处定义，而常规的非环境变量声明是不使用 `declare` 关键字声明变量的普通 `let` 或 `const` 声明。

你可以使用类型声明来做几件事：

- 告诉TypeScript关于在JavaScript某处定义的全局变量。例如，如果你polyfilled了 `Promise` 全局变量或在浏览器环境中定义了 `process.env`，你可能使用*环境变量声明

*来给TypeScript一个提示。

- 定义在你的项目中全局可用的类型，所以使用它时你不必先导入它（我们称之为环境类型声明）。

- 告诉TypeScript关于你用NPM安装的第三方模块（*环境模块声明*）。

类型声明，无论你用它来做什么，都必须存在于脚本模式的 *.ts* 或 *.d.ts* 文件中（回顾我们之前在["模块模式与脚本模式"]中的讨论）。按照惯例，如果文件有对应的 *.js* 文件，我们给文件一个 *.d.ts* 扩展名；否则，我们使用 *.ts* 扩展名。文件名无关紧要---例如，我喜欢坚持使用单个顶级 *types.ts* 文件，直到它变得难以管理---一个类型声明文件可以包含任意数量的类型声明。

最后，虽然类型声明文件中的顶级值需要 `declare` 关键字（`declare let`、`declare function`、`declare class` 等），但顶级类型和接口不需要。

了解了这些基本规则后，让我们简要看看每种类型声明的一些示例。

环境变量声明

环境变量声明是一种告诉 TypeScript 有关全局变量的方法，该变量可以在项目中的任何 *.ts* 或 *.d.ts* 文件中使用，而无需先显式导入它。

假设你在浏览器中运行一个 NodeJS 程序，程序在某个时候检查 `process.env.NODE_ENV`（它是 `"development"` 或 `"production"`）。当你运行程序时，你得到一个难看的运行时错误：

Uncaught ReferenceError: process is not defined.

你在 Stack Overflow 上搜索了一下，意识到让程序运行的最快方法是自己填充 `process.env.NODE_ENV` 并将其硬编码。所以你创建了一个新文件 *polyfills.ts*，并定义了一个全局 `process.env`：

process = { env: { NODE_ENV: ‘production’ } }

当然，TypeScript 然后出手相救，给你抛出一个红色波浪线，试图拯救你免于你显然正在犯

的错误：增强 `window` 全局对象：

Error TS2304: Cannot find name ‘process’.

但在这种情况下，TypeScript 过度保护了。你真的想要增强 `window`，并且你想要安全地做到这一点。

那么你该怎么办？你在 Vim 中打开 *polyfills.ts*（你知道这要走向何处）并键入：

declare let process: { env: { NODE_ENV: ‘development’ | ‘production’ } }

process = { env: { NODE_ENV: ‘production’ } }

你向 TypeScript 声明有一个全局对象 `process`，它有一个单一的属性 `env`，该属性有一个属性 `NODE_ENV`。一旦你告诉 TypeScript 这些，红色波浪线就会消失，你就可以安全地定义你的 `process` 全局对象。

TSC 设置：lib

TypeScript 带有一套类型声明，用于描述 JavaScript 标准库，包括内置的 JavaScript 类型，如 `Array` 和 `Promise`，以及内置类型上的方法，如 `''.toUpperCase`。它还包括全局对象，如 `window` 和 `document`（在浏览器环境中），以及 `onmessage`（在 Web Worker 环境中）。

你可以使用 *tsconfig.json* 的 `lib` 字段引入 TypeScript 的内置类型声明。跳转到["lib"]，深入了解如何调整项目的 `lib` 设置。

环境类型声明

环境类型声明遵循与环境变量声明相同的规则：声明必须存在于脚本模式的 *.ts* 或 *.d.ts* 文件中，并且它将在你的项目中的其他文件中全局可用，无需显式导入。例如，让我们声明一个全局实用类型 `ToArray<T>`，如果 `T` 还不是数组，则将其提升为数组。我们可以在项目中的任何脚本模式文件中定义这种类型---对于这个例子，让我们在顶级 *types.ts* 文件中定义它：

type ToArray = T extends unknown[] ? T : T[]

现在我们可以从任何项目文件中使用这种类型，无需显式导入：

function toArray(a: T): ToArray { // … }

考虑使用环境类型声明来建模在整个应用程序中使用的数据类型。例如，你可能使用它们来使我们在["模拟标称类型(Nominal Types)"]中开发的 `UserID` 类型全局可用：

type UserID = string & {readonly brand: unique symbol}

现在，你可以在应用程序的任何地方使用 `UserID`，而无需先显式导入它。

环境模块声明

当你使用一个 JavaScript 模块并想要快速为其声明一些类型以便安全使用时——无需先将类型声明贡献回 JavaScript 模块的 GitHub 仓库或 DefinitelyTyped——ambient 模块声明就是你需要的工具。

Ambient 模块声明是一个常规的类型声明，被特殊的 `declare module` 语法包围：

declare module ‘module-name’ { export type MyType = number export type MyDefaultType = {a: string} export let myExport: MyType let myDefaultExport: MyDefaultType export default myDefaultExport }

模块名（本例中的 `'module-name'`）对应一个确切的 `import` 路径。当你导入该路径时，你的 ambient 模块声明告诉 TypeScript 有什么可用：

import ModuleName from ‘module-name’ ModuleName.a // string

如果你有嵌套模块，确保在声明中包含完整的 `import` 路径：

declare module ‘@most/core’ { // 类型声明 }

如果你只想快速告诉 TypeScript "我正在导入这个模块——稍后再为其添加类型，现在就假设它是 `any`"，保留头部但省略实际声明：

// 声明一个可以被导入的模块，其每个导入都是 any declare module ‘unsafe-module-name’

现在如果你使用这个模块，它就不那么安全了：

import {x} from ‘unsafe-module-name’ x // any

模块声明支持通配符导入，因此你可以为匹配给定模式的任何 `import` 路径提供类型。使用通配符（`*`）来匹配 `import` 路径：

// 为使用 Webpack 的 json-loader 导入的 JSON 文件添加类型 declare module ’json!*’ { let value: object export default value }

// 为使用 Webpack 的 style-loader 导入的 CSS 文件添加类型 declare module ’*.css’ { let css: CSSRuleList export default css }

现在，你可以加载 JSON 和 CSS 文件：

import a from ‘json!myFile’ a // object

import b from ‘./widget.css’ b // CSSRuleList

注意

要让最后两个示例工作，你需要配置你的构建系统来加载 *.json* 和 *.css* 文件。你可以向 TypeScript 声明这些路径模式是安全导入的，但 TypeScript 无法自己构建它们。

跳转到 ["JavaScript That Doesn't Have Type Declarations on DefinitelyTyped"] 查看如何使用 ambient 模块声明为无类型的第三方 JavaScript 声明类型的示例。

从 JavaScript 逐步迁移到 TypeScript

TypeScript 在设计时就考虑了 JavaScript 互操作性，而不是事后的想法。因此，虽然不是完全无痛的，但迁移到 TypeScript 是一个很好的体验，让你可以一次转换一个文件，在迁移过程中选择更严格的安全级别，向你的老板和同事展示静态类型化代码的影响力，一次提交一次。

从高层次来看，这是你想要达到的目标：你的代码库应该完全用 TypeScript 编写，具有严格的类型覆盖，你依赖的第三方 JavaScript 库应该自带高质量、严格的类型。任何可以在编译时捕获的错误都会被捕获，TypeScript 丰富的自动完成功能将编写每行代码所需的时间减半。你可能需要采取一些小步骤才能达到那里：

- 将 TSC 添加到你的项目中。

- 开始对现有的 JavaScript 代码进行类型检查。

- 将你的 JavaScript 代码迁移到 TypeScript，一次一个文件。

- 为你的依赖项安装类型声明，要么为没有类型的依赖项添加存根类型，要么为无类型依赖项编写类型声明并将它们贡献回 DefinitelyTyped。

- 为你的代码库开启 `strict` 模式。

这个过程可能需要一段时间，但你会立即看到安全性和生产力的提升，并在继续进行时发现更多收益。让我们逐步完成这些步骤。

步骤 1：添加 TSC

在处理结合了 TypeScript 和 JavaScript 的代码库时，首先让 TSC 编译 JavaScript 文件以及您的 TypeScript 文件。在您的 *tsconfig.json* 中：

{ “compilerOptions”: { “allowJs”: true }

通过这一个改变，您现在可以使用 TSC 来编译 JavaScript。只需将 TSC 添加到构建过程中，然后要么通过 TSC 运行每个现有的 JavaScript 文件，要么继续通过现有构建过程运行遗留的 JavaScript 文件，并通过 TSC 运行新的 TypeScript 文件。

当 `allowJs` 设置为 `true` 时，TypeScript 不会对您现有的 JavaScript 代码进行类型检查，但它会使用您要求的模块系统（在 *tsconfig.json* 的 `module` 字段中）将其转译（到 ES3、ES5 或 *tsconfig.json* 中 `target` 设置的任何版本）。第一步，完成。提交它，给自己鼓掌一下——您的代码库现在使用 TypeScript 了！

步骤 2a：为 JavaScript 启用类型检查（可选）

现在 TSC 正在处理您的 JavaScript，为什么不也对其进行类型检查呢？虽然您的 JavaScript 中可能没有显式的类型注释，但请记住 TypeScript 在为您推断类型方面有多么出色；它可以像在 TypeScript 代码中一样推断 JavaScript 中的类型。在您的 *tsconfig.json* 中启用此功能：

{ “compilerOptions”: { “allowJs”: true, “checkJs”: true }

现在，每当 TypeScript 编译 JavaScript 文件时，它都会尽力推断类型并在进行过程中进行类型检查，就像它对常规 TypeScript 代码所做的那样。

如果您的代码库很大，打开 `checkJs` 会一次报告太多类型错误，请将其关闭，然后通过在文件顶部添加 `// @ts-check` 指令（常规注释）来逐个文件启用检查。或者，如果几个大文件产生了大部分错误，而您暂时不想修复它们，请保持 `checkJs` 开启，并仅对这些文件添加 `// @ts-nocheck` 指令。

注意

因为 TypeScript 无法推断所有内容（例如，函数参数类型），它会将 JavaScript 代码中的许多类型推断为 `any`。如果您在 *tsconfig.json* 中启用了 `strict` 模式（您应该启用！），在迁移时您可能希望暂时允许隐式 `any`。在您的 *tsconfig.json* 中，添加：

{ “compilerOptions”: { “allowJs”: true, “checkJs”: true, “noImplicitAny”: false }

当您将大部分代码迁移到 TypeScript 时，不要忘记再次打开 `noImplicitAny`！它可能会揭示许多您遗漏的真实错误（当然，除非您是 Xenithar，JavaScript 女巫 Bathmorda 的门徒，能够在脑海中进行类型检查，只需一大锅艾草的帮助）。

当 TypeScript 处理 JavaScript 代码时，它使用比处理 TypeScript 代码更宽松的推断算法。具体来说：

- 所有函数参数都是可选的。

- 函数和类的属性类型是从使用中推断出来的（而不必预先声明）：

class A { x = 0 // number | string | string[], 从使用中推断 method() { this.x = 'foo' } otherMethod() { this.x = ['array', 'of', 'strings'] } }

- 在声明对象、类或函数后，您可以为其分配额外的属性。在底层，TypeScript 通过为每个类和函数声明生成相应的命名空间，并自动为每个对象字面量添加索引签名来实现这一点。

步骤 2b：添加 JSDoc 注释（可选）

也许你正在赶时间，只需要为一个添加到旧JavaScript文件中的新函数添加一个类型注解。[]{#calibre_link-1691 .calibre4 primary="TypeScript" secondary="gradually migrating JavaScript code to" tertiary="adding JSDoc annotations" data-type="indexterm"}[]{#calibre_link-1169 .calibre4 primary="JSDoc annotations" data-type="indexterm"} 在你有机会将该文件转换为TypeScript之前，你可以使用JSDoc注解来为你的新函数添加类型。

你可能以前见过JSDoc；它是JavaScript和TypeScript代码上方那些看起来奇怪的注释，带有`@`注解，如`@param`、`@returns`等等。TypeScript理解JSDoc，并将其作为类型检查器的输入，就像它使用TypeScript代码中的显式类型注解一样。

假设你有一个3000行的工具文件（是的，我知道是你的"朋友"写的）。你向其中添加了一个新的工具函数：

export function toPascalCase(word) { return word.replace(/+/g, ([a, ...b]) => a.toUpperCase() + b.join('').toLowerCase()) }

在不将*utils.js*完全转换为TypeScript的情况下——这可能会发现一堆你必须修复的bug——你可以只为你的`toPascalCase`函数添加注解，在无类型JavaScript的海洋中开辟出一个安全的小岛：

/** * @param word {string} An input string to convert * @returns {string} The string in PascalCase */ export function toPascalCase(word) { return word.replace(/+/g, ([a, …b]) => a.toUpperCase() + b.join(’ ’).toLowerCase()) }

没有JSDoc注解，TypeScript会将`toPascalCase`的类型推断为`(word: any) => string`。现在，当TypeScript编译你的代码时，它知道`toPascalCase`的类型是`(word: string) => string`。而且你还得到了一些很好的文档！

前往[TypeScript Wiki](http://bit.ly/2YCTWBf)了解更多支持的JSDoc注解。

步骤3：将文件重命名为.ts {#step-3-rename-your-files-to-.ts .calibre17}

一旦你将TSC添加到构建过程中，并可选地开始对JavaScript进行类型检查和注解，就该开始切换到TypeScript了。[]{#calibre_link-1694 .calibre4 primary="TypeScript" secondary="gradually migrating JavaScript code to" tertiary="renaming files to .ts" data-type="indexterm"}

一次一个文件，将文件扩展名从*.js*（或*.coffee*、*.es6*等）更新为*.ts*。一旦你在代码编辑器中重命名文件，你就会看到你的朋友们——红色波浪线出现（是TypeError，不是儿童电视节目），揭示类型错误、遗漏的情况、忘记的`null`检查和拼写错误的变量名。修复这些错误有两种策略：

1. 做对它。花时间正确地为形状、字段和函数添加类型，这样你就可以在所有使用它们的文件中捕获错误。如果你启用了`checkJs`，在你的*tsconfig.json*中打开`noImplicitAny`来发现`any`并为其添加类型，然后将其关闭以减少对剩余JavaScript文件进行类型检查时的噪音输出。

2. 做得快。批量将JavaScript文件重命名为*.ts*扩展名，并保持*tsconfig.json*设置宽松（意味着`strict`设置为`false`），在重命名后尽可能少地抛出类型错误。将复杂类型标记为`any`以满足类型检查器。修复剩余的类型错误，然后提交。完成后，逐一开启`strict`模式标志（`noImplicitAny`、`noImplicitThis`、`strictNullChecks`等），并修复出现的错误。（参见[附录F]获取这些标志的完整列表。）

提示 {#tip-8 .calibre22}

如果你选择走快速粗糙的路线，一个有用的技巧是定义一个环境类型声明`TODO`作为`any`的类型别名，并使用它而不是`any`，这样你可以更容易地找到和跟踪缺失的类型。[]{#calibre_link-613 .calibre4 primary="ambient type declarations" secondary="defining TODO as type alias for any" data-type="indexterm"}[]{#calibre_link-630 .calibre4 primary="any type" secondary="TODO ambient type declaration as type alias for" data-type="indexterm"} 你也可以称它为更具体的名称，这样在项目范围的代码搜索中更容易找到：

// globals.ts type TODO_FROM_JS_TO_TS_MIGRATION = any

// MyMigratedUtil.ts export function mergeWidgets(widget1: TODO_FROM_JS_TO_TS_MIGRATION, widget2: TODO_FROM_JS_TO_TS_MIGRATION): number { // … }

这两种方法都是公平的，你想选择哪种取决于你。因为TypeScript是一种渐进类型语言，它从头开始构建，以尽可能安全地与无类型JavaScript代码互操作。无论你是在严格类型的TypeScript与无类型JavaScript之间互操作，还是在严格类型的TypeScript与松散类型的TypeScript之间互操作，TypeScript都会尽力确保你尽可能安全地进行，并且在你精心构建的严格类型岛上，一切都尽可能安全。

步骤4：使其严格 {#step-4-make-it-strict .calibre17}

一旦你将大量JavaScript迁移到TypeScript，你会希望通过逐一选择TSC更严格的标志[]{#calibre_link-1531 .calibre4 primary="TSC compiler" secondary="using more stringent flags on migrated JavaScript code" data-type="indexterm"}[]{#calibre_link-1695 .calibre4 primary="TypeScript" secondary="gradually migrating JavaScript code to" tertiary="using strict TSC flags" data-type="indexterm"}来使代码尽可能安全（参见[附录F]获取标志的完整列表）。

最后，你可以禁用TSC的JavaScript互操作性标志，强制所有代码都用严格类型的TypeScript编写：

{ “compilerOptions”: { “allowJs”: false, “checkJs”: false }

这将展现最后几轮类型相关的错误。修复这些错误，你将得到一个最纯净、安全的代码库，即使是最hardcore的OCaml工程师也会拍拍你的后背表示赞许（如果你礼貌地询问的话）。

按照这些步骤进行，在为你控制的JavaScript添加类型时会走得很远，但对于你不控制的JavaScript，比如从NPM安装的代码，该怎么办呢？为了到达那里，我们首先需要绕一个小弯路...

JavaScript的类型查找

当你从TypeScript文件中导入JavaScript文件时，TypeScript遵循这样一个算法来查找JavaScript代码的类型声明（记住，当我们谈论TypeScript时，"文件"和"模块"是可以互换的）：

1. 查找与你的*.js*文件同名的兄弟*.d.ts*文件。如果存在，将其用作*.js*文件的类型声明。

    例如，假设你有以下文件夹结构：

my-app/ ├——src/ | ├——index.ts | └——legacy/ | ├——old-file.js | └——old-file.d.ts “、

然后你从*index.ts*中导入*old-file*：

```
// index.ts
import './legacy/old-file'
```

TypeScript将使用*src/legacy/old-file.d.ts*作为*[./legacy/old-file]*的类型声明来源。

2. 否则，如果`allowJs`和`checkJs`为true，推断*.js*文件的类型（根据*.js*文件中的任何JSDoc注解）。

3. 否则，将整个模块视为`any`。

当导入第三方JavaScript模块——也就是你安装到*node_modules*的NPM包时——TypeScript使用略有不同的算法：

1. 查找模块的本地类型声明。如果存在，使用它。

   例如，假设你的应用程序文件夹结构如下：

   ```
   my-app/
   ├──node_modules/
   |  └──foo/
   ├──src/
   |  ├──index.ts
   |  └──types.d.ts
   ```

   *types.d.ts*看起来像这样：

   ```
   // types.d.ts
   declare module 'foo' {
     let bar: {}
     export default bar
   }
   ```

   如果你然后导入`foo`，TypeScript将使用*types.d.ts*中的环境模块声明作为`foo`的类型来源：

   ```
   // index.ts
   import bar from 'foo'
   ```

2. 否则，查看模块的*package.json*。如果它定义了一个名为`types`或`typings`的字段，使用该字段指向的*.d.ts*文件作为模块类型声明的来源。

3. 否则，逐个目录向上遍历，查找具有该模块类型声明的*node_modules/@types*目录。

   例如，假设你安装了React：

   ```
   npm install react --save
   npm install @types/react --save-dev
   ```

```
my-app/
├──node_modules/
| ├──@types/
| | └──react/
| └──react/
├──src/
| └──index.ts
```

当你导入React时，TypeScript将找到*@types/react*文件夹并将其用作React类型声明的来源：

```
// index.ts
import * as React from 'react'
```

4. 否则，继续执行本地类型查找算法的第1-3步。

这是很多步骤，但一旦掌握窍门，它就非常直观了。

TSC设置： types和typeRoots

默认情况下，TypeScript在你项目文件夹和包含文件夹（*../node_modules/@types*等）的*node_modules/@types*中查找第三方类型声明。大多数时候，你想保持这种行为不变。

要覆盖全局类型声明的默认行为，请在*tsconfig.json*中配置`typeRoots`，提供一个要查找类型声明的文件夹数组。例如，你可以告诉TypeScript在*typings*文件夹以及*node_modules/@types*中查找类型声明：

```
{
  "compilerOptions": {
    "typeRoots" : ["./typings", "./node_modules/@types"]
  }
}
```

为了更精细的控制，请在*tsconfig.json*中使用`types`选项来指定你希望TypeScript为哪些包查找类型。例如，以下配置忽略所有第三方类型声明，除了React的声明：

```
{
  "compilerOptions": {
    "types" : ["react"]
  }
}
```

使用第三方JavaScript

注意

我假设你正在使用像 NPM 或 Yarn 这样的包管理器来安装第三方 JavaScript。如果你是那种喜欢手动复制粘贴代码的人——羞耻啊。

当你 `npm install` 第三方 JavaScript 代码到你的项目中时，有三种可能的情况：

1. 你安装的代码自带类型声明。
2. 你安装的代码不带类型声明，但在 DefinitelyTyped 上有可用的声明。
3. 你安装的代码不带类型声明，且在 DefinitelyTyped 上也没有可用的声明。

让我们深入了解这三种情况。

自带类型声明的 JavaScript

你可以知道一个包是否自带类型声明，只需在 `{"noImplicitAny": true}` 的情况下 `import` 它，如果 TypeScript 没有给你红色波浪线提示，那就说明有。

如果你要安装的代码是从 TypeScript 编译而来，或者其作者足够友善地在其 NPM 包中包含了类型声明，那么你很幸运。只需安装代码并开始使用它，就能获得完整的类型支持。

一些自带类型声明的 NPM 包示例：

```
npm install rxjs
npm install ava
npm install @angular/cli
```

警告

除非你安装的代码确实是从 TypeScript 编译而来，否则你始终面临着一个风险：自带的类型声明可能与这些声明所描述的代码不匹配。当类型声明与源代码打包在一起时，发生这种情况的风险相当低（特别是对于流行的包），但这是需要注意的。

在 DefinitelyTyped 上有类型声明的 JavaScript

即使你导入的第三方代码没有自带类型声明，它的声明也可能在 DefinitelyTyped 上可用，这是 TypeScript 社区维护的、用于开源项目的环境模块声明的集中存储库。

要检查你安装的包是否在 DefinitelyTyped 上有可用的类型声明，可以在 TypeSearch 上搜索，或者直接尝试安装声明。所有 DefinitelyTyped 类型声明都在 `@types` 作用域下发布到 NPM，所以你可以直接从该作用域 `npm install`：

```
npm install lodash --save            # 安装 Lodash
npm install @types/lodash --save-dev # 安装 Lodash 的类型声明
```

大多数时候，你会想要使用 `npm install` 的 `--save-dev` 标志将安装的类型声明添加到你的 *package.json* 的 `devDependencies` 字段中。

注意

由于 DefinitelyTyped 上的类型声明是社区维护的，它们可能存在不完整、不准确或过时的风险。虽然大多数流行包都有维护良好的类型声明，但如果你发现正在使用的声明可以改进，请花时间改进它们并将它们贡献回 DefinitelyTyped，这样其他 TypeScript 用户就能受益于你的辛勤工作。

在 DefinitelyTyped 上没有类型声明的 JavaScript

这是三种情况中最不常见的。你有几个选择，从最便宜最不安全的到最耗时最安全的：

1. *将特定导入加入白名单*，在你的无类型导入上方添加 `// @ts-ignore` 指令。TypeScript 会让你使用无类型模块，但该模块及其所有内容都将被类型化为 `any`：

```
// @ts-ignore
import Unsafe from 'untyped-module'

Unsafe  // any
```

2. *将此模块的所有用法加入白名单*，通过创建一个空的类型声明文件并存根该模块。例如，如果你安装了很少使用的包 `nearby-ferret-alerter`，你可以创建一个新的类型声明（例如 *types.d.ts*）并向其添加环境类型声明：

```
// types.d.ts
declare module 'nearby-ferret-alerter'
```

这告诉 TypeScript 存在一个你可以导入的模块（`import alert from 'nearby-ferret-alerter'`），但它没有告诉 TypeScript 该模块中包含的类型的任何信息。这种方法比第一种稍好一些，因为现在有一个中心的 *types.d.ts* 文件枚举了应用程序中所有无类型的模块，但它同样不安全，因为 `nearby-ferret-alerter` 及其所有导出仍然会被类型化为 `any`。

3. *创建环境模块声明(ambient module declaration)*。就像之前的方法一样，创建一个名为 *types.d.ts* 的文件并添加一个空声明（`declare module 'nearby-ferret-alerter'`）。现在，填入类型声明。例如，结果可能如下所示：

```
// types.d.ts
declare module 'nearby-ferret-alerter' {
  export default function alert(loudness: 'soft' | 'loud'):
Promise<void>
```

```
  export function getFerretCount(): Promise<number>
}
```

现在当你 `import alert from 'nearby-ferret-alerter'` 时，TypeScript 将确切知道 `alert` 的类型是什么。它不再是 `any`，而是 `(loudness: 'quiet' | 'loud') => Promise<void>`。

4. *创建类型声明并将其贡献回 NPM*。如果你已经完成了第三个选项，现在有了模块的本地类型声明，请考虑将其贡献回 NPM，这样下一个需要出色的 `nearby-ferret-alerter` 包的类型声明的人也可以使用它。为此，你可以向 `nearby-ferret-alerter` Git 仓库提交拉取请求并直接贡献类型声明，或者，如果该仓库的维护者不想负责维护 TypeScript 类型声明，则将你的声明贡献给 DefinitelyTyped。

为第三方 JavaScript 编写类型声明很简单，但具体如何做取决于你要键入的模块类型。在键入不同类型的 JavaScript 模块时会出现一些常见模式（从 NodeJS 模块到 jQuery 增强和 Lodash 混入到 React 和 Angular 组件）。请前往[附录 D]查看键入第三方 JavaScript 模块的配方列表。

注意

为无类型 JavaScript 自动生成类型声明是一个活跃的研究领域。查看 `dts-gen` 以获得为任何第三方 JavaScript 模块自动生成类型声明脚手架的方法。

总结

有几种方法可以从 TypeScript 使用 JavaScript。[表 11-1] 总结了这些选项。

[表 11-1.] 从 TypeScript 使用 JavaScript 的方法

方法	tsconfig.json 标志	类型安全性
导入无类型 JavaScript	`{"allowJs": true}`	差
导入并检查 JavaScript	`{"allowJs": true, "checkJs": true}`	一般
导入并检查 JSDoc 注解的 JavaScript	`{"allowJs": true, "checkJs": true, "strict": true}`	优秀
使用类型声明导入 JavaScript	`{"allowJs": false, "strict": true}`	优秀
导入 TypeScript	`{"allowJs": false, "strict": true}`	优秀

在本章中，我们涵盖了将 JavaScript 和 TypeScript 一起使用的各个方面，从不同类型的类型声明以及如何使用它们，到将现有 JavaScript 项目逐步迁移到 TypeScript，再到安全（和不安全）地使用第三方 JavaScript。与 JavaScript 互操作可能是 TypeScript 最棘手的方面之一；有了你可以使用的所有工具，你现在有能力在自己的项目中做到这一点。

[1] 使用 `*` 的通配符匹配遵循与常规 glob 模式匹配 相同的规则。

[2] DefinitelyTyped 是 JavaScript 类型声明的开源仓库。请继续阅读以了解更多信息。

[3] 对于真正大型的项目，通过 TSC 运行每个单独文件可能会很慢。有关改善大型项目性能的方法，请参阅[“项目引用”]。

[4] 严格来说，这对模块模式文件是正确的，但对脚本模式文件不是。在[“模块模式与脚本模式”]中阅读更多信息。

第 12 章. 构建和运行 TypeScript

如果您已经在生产环境中部署和运行过JavaScript应用程序，那么您也知道如何运行TypeScript应用程序——一旦将其编译为JavaScript，两者并没有太大区别。本章节讨论TypeScript应用程序的生产化和构建，但这里大部分内容对TypeScript应用来说并不独有—它们大多也适用于JavaScript应用程序。我们将把它分为四个部分，涵盖：

- 构建任何TypeScript应用程序必须要做的事情
- 在服务器上构建和运行TypeScript应用程序
- 在浏览器中构建和运行TypeScript应用程序
- 为您的TypeScript应用程序构建并发布到NPM

构建您的TypeScript项目

构建TypeScript项目很简单。在本节中，我们将涵盖您需要理解的核心概念，以便为您计划运行应用程序的任何环境构建项目。

项目布局

我建议将您的源TypeScript代码保存在顶级*src/*文件夹中，并将其编译到顶级*dist/*文件夹中。这种文件夹结构是一种流行的约定，将源代码和生成的代码分隔到两个顶级文件夹中可以让您在后续与其他工具集成时更加轻松。它也使得从源代码控制中排除生成的构件变得更容易。

在可能的情况下，尽量坚持这个约定：

```
my-app/
├──dist/
| ├──index.d.ts
| ├──index.js
| └──services/
|   ├──foo.d.ts
|   ├──foo.js
|   ├──bar.d.ts
|   └──bar.js
├──src/
| ├──index.ts
| └──services/
|   ├──foo.ts
|   └──bar.ts
```

构件

当您将TypeScript程序编译为JavaScript时，TSC可以为您生成几种不同的构件（表12-1）。

类型	文件扩展名	tsconfig.json标志	默认生成?
JavaScript	*.js*	`{"emitDeclarationOnly": false}` 是	
源映射 *.js.	map* `{ “so	urceMap”: true}` 否	
类型声明 *.d.ts*	`{ “declar	ation”: true}` 否	
声明映射 *.d.ts.m	ap* `{ “decla	rationMap”: true}` 否	

表12-1. TSC可以为您生成的构件

第一种构件类型—JavaScript文件—应该很熟悉。TSC将您的TypeScript代码编译为JavaScript，然后您可以使用JavaScript平台（如NodeJS或Chrome）运行它。如果您运行`tsc yourfile.ts`，TSC将对*yourfile.ts*进行类型检查并将其编译为JavaScript。

第二种构件类型—源映射(source maps)—是特殊文件，它将生成的JavaScript的每个片段链接回生成它的TypeScript文件的特定行和列。这对于调试您的代码很有帮助（Chrome DevTools将显示您的TypeScript代码，而不是生成的JavaScript），以及将JavaScript异常堆栈跟踪中的行和列映射回TypeScript（如”错误监控”中提到的工具会在您提供源映射时自动执行此查找）。

第三种构件—类型声明—让其他TypeScript项目可以利用您生成的类型。

最后，声明映射(declaration maps)用于加快TypeScript项目的编译时间。您将在”项目引用”中了解更多相关信息。本章的其余部分我们将讨论如何以及为什么要生成这些构件。

确定您的编译目标

JavaScript可能是一种不寻常的语言：它不仅具有快速发展的规范和年度发布周期，而且作为程序员，您并不总是能够控制运行程序的平台实现的JavaScript版本。除此之外，许多JavaScript程序是*同构的(isomorphic)*，意味着您可以在服务器或客户端上运行它们。例如：

- 如果您在自己控制的服务器上运行后端JavaScript程序，那么您可以精确控制它将运行的JavaScript版本。
- 如果你将后端 JavaScript 程序作为开源项目发布，你不知道消费者的 JavaScript 平台将支持哪个 JavaScript 版本。在 NodeJS 环境中，你能做的最好的事情就是声明支持的 NodeJS 版本范围，但在浏览器环境中你就束手无策了。
- 如果你在浏览器中运行 JavaScript，你不知道人们会使用哪种浏览器来运行它——支持大多数现代 JavaScript 特性的最新 Chrome、Firefox 或 Edge，这些浏览器中缺少一些前沿功能的稍微过时的版本，像 Internet Explorer 8 这样的过时浏览器，或者像你车库里 PlayStation 4 上运行的嵌入式浏览器。你能做的最好的事情就是定义人们的浏览器需要支持的最小特性集来运行你的应用程序，为尽可能多的这些特性提供 polyfill，并尝试检测用户何时使用你的应用程序无法运行的真正旧浏览器，并向他们显示需要升级的消息。
- 如果你发布一个同构 JavaScript 库（例如，在浏览器和服务器上都运行的日志库），那么你必须同时支持最低 NodeJS 版本和一系列浏览器 JavaScript 引擎和版本。

并非每个 JavaScript 环境都能开箱即用地支持每个 JavaScript 特性，但你仍应尝试用最新的语言版本编写代码。有两种方法可以做到这一点：

1. *转译*（即自动转换）应用程序，从最新版本的 JavaScript 转换为你的目标平台支持的最旧 JavaScript 版本。我们对语言特性如 `for..of` 循环和 `async` / `await` 执行此操作，它们可以分别自动转换为 `for` 循环和 `.then` 调用。
2. *Polyfill*（即为其提供实现）你运行的 JavaScript 运行时中缺少的任何现代特性。我们对 JavaScript 标准库提供的特性（如 `Promise`、`Map` 和 `Set`）和原型方法（如 `Array.prototype.includes` 和 `Function.prototype.bind`）执行此操作。

TSC 内置支持将你的代码转译为较旧的 JavaScript 版本，但它不会自动对你的代码进行 polyfill。这值得重申：TSC 将为较旧的环境转译大多数 JavaScript 特性，但它不会为缺失的特

性提供实现。

TSC 提供了三个设置来精确定位你想要目标的环境：

- `target` 设置你想要转译到的 JavaScript 版本：`es5`、`es2015` 等。
- `module` 设置你想要目标的模块系统：`es2015` 模块、`commonjs` 模块、`systemjs` 模块等。
- `lib` 告诉 TypeScript 在你的目标环境中哪些 JavaScript 特性可用：`es5` 特性、`es2015` 特性、`dom` 等。它实际上不实现这些特性——那是 polyfill 的作用——但它确实告诉 TypeScript 这些特性是可用的（要么原生支持，要么通过 polyfill）。

你计划运行应用程序的环境决定了你应该用 `target` 转译到哪个 JavaScript 版本以及将 `lib` 设置为什么。如果你不确定，`es5` 通常是两者的安全默认值。你将 `module` 设置为什么取决于你是否目标 NodeJS 或浏览器环境，以及如果是后者你使用什么模块加载器。

提示

如果你需要支持一组不寻常的平台，请在 Juriy Zaytsev（也称为 Kangax）的兼容性表中查看你的目标平台原生支持哪些 JavaScript 特性。

让我们更深入地了解 `target` 和 `lib`；我们将把 `module` 留给[“在服务器上运行 TypeScript”]和[“在浏览器中运行 TypeScript”]章节。

target

TSC 的内置转译器支持将大多数 JavaScript 特性转换为较旧的 JavaScript 版本，这意味着你可以用最新的 TypeScript 版本编写代码并将其转译为你需要支持的任何 JavaScript 版本。由于 TypeScript 支持最新的 JavaScript 特性（如 `async` / `await`，在撰写本文时尚未得到所有主要 JavaScript 平台的支持），你几乎总是会发现自己利用这个内置转译器将你的代码转换为 NodeJS 和浏览器今天理解的东西。

让我们看看 TSC 为较旧的 JavaScript 版本转译和不转译哪些特定的 JavaScript 特性（[表 12-2] 和 [表 12-3]）。

注意

过去，JavaScript 语言每隔几年发布一个新版本，语言版本号递增（ES1、ES3、ES5、ES6）。从 2015 年开始，JavaScript 语言现在采用年度发布周期，每个语言版本以发布年份命名（ES2015、ES2016 等等）。然而，一些 JavaScript 特性在实际规划到特定 JavaScript 版本之前就获得了 TypeScript 支持；我们称这些特性为”ESNext”（即下一个版本）。

版本	特性
ES2015	`const`、`let`、`for..of` 循环、数组/对象展开（`...`）、标记模板字符串、类、生成器、箭头函数、函数默认参数、函数剩余参数、解构声明/赋值/参数
ES2016	指数运算符（`**`）
ES2017	`async` 函数、`await` Promise
ES2018	`async` 迭代器
ES2019	catch 子句中的可选参数
ESNext	数字分隔符（`123_456`）

表 12-2. TSC 会转换的特性

版本	特性
ES5	对象 getter/setter
ES2015	正则表达式 `y` 和 `u` 标志
ES2018	正则表达式 `s` 标志
ESNext	BigInt（`123n`）

表 12-3. TSC 不会转换的特性

要设置转换目标，打开您的 *tsconfig.json* 并将 `target` 字段设置为：

- `es3` 用于 ECMAScript 3
- `es5` 用于 ECMAScript 5（如果不确定使用什么，这是一个不错的默认选择）
- `es6` 或 `es2015` 用于 ECMAScript 2015
- `es2016` 用于 ECMAScript 2016
- `es2017` 用于 ECMAScript 2017
- `es2018` 用于 ECMAScript 2018
- `esnext` 用于最新的 ECMAScript 版本

例如，要编译到 ES5：

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

lib

如我所提到的，将代码转换到较旧的 JavaScript 版本有一个问题：虽然大多数语言特性可以安全地转换（`let` 转为 `var`，`class` 转为 `function`），但如果您的目标环境不支持较新的库特性，您仍然需要自己*填充*(polyfill)功能。一些例子包括像 `Promise` 和 `Reflect` 这样的工具，以及像 `Map`、`Set` 和 `Symbol` 这样的数据结构。当目标是最新的 Chrome、Firefox 或 Edge 等前沿环境时，您通常不需要任何填充；但如果您的目标是几个版本之前的浏览器——或大多数 NodeJS 环境——您将需要填充缺失的特性。

值得庆幸的是，您不需要自己编写填充。相反，您可以从流行的填充库如 `core-js` 安装它们，或者通过使用 `@babel/polyfill` 运行您的类型检查过的 TypeScript 代码通过 Babel 来自动添加填充到您的代码中。

提示

如果您计划在浏览器中运行您的应用程序，请小心不要通过包含每一个填充来膨胀您的 JavaScript 包的大小，无论运行代码的浏览器是否真正需要它——您的目标平台可能已经支持您正在填充的一些特性。相反，使用像 Polyfill.io 这样的服务来只加载用户浏览器需要的那些填充。

一旦您将填充添加到您的代码中，是时候告诉 TSC 您的环境保证支持您填充的特性了——输入您的 *tsconfig.json* 的 `lib` 字段。例如，如果您已经填充了所有 ES2015 特性加上 ES2016 的 `Array.prototype.includes`，您可以使用这个配置：

```
{
  "compilerOptions": {
    "lib": ["es2015", "es2016.array.includes"]
  }
}
```

如果您在浏览器中运行您的代码，还要启用 DOM 类型声明，用于像 `window`、`document` 和在浏览器中运行 JavaScript 时获得的所有其他 API：

```
{
  "compilerOptions": {
    "lib": ["es2015", "es2016.array.include", "dom"]
  }
}
```

要获取支持的 lib 的完整列表，运行 `tsc --help`。

启用 Source Maps

Source maps 是一种将转译后的代码链接回生成它的源代码的方法。大多数开发者工具（如 Chrome DevTools）、错误报告和日志框架，以及构建工具都了解 source maps。由于典型的构建管道可能会产生与您开始时的代码非常不同的代码（例如，您的管道可能将 TypeScript 编译为 ES5 JavaScript，使用 Rollup 进行 tree-shake，使用 Prepack 进行预评估，然后使用 Uglify 进行压缩），在整个构建管道中使用 source maps 可以使调试生成的 JavaScript 变得容易得多。

通常在开发中使用 source maps 是一个好主意，并在浏览器和服务器环境中将 source maps 部署到生产环境。不过有一个注意事项：如果您在浏览器代码中依赖于某种程度的安全性，不要在生产环境中将 source maps 发送到浏览器。

项目引用

随着应用程序的增长，TSC 对代码进行类型检查和编译的时间会越来越长。这个时间大致与代码库的大小呈线性增长。在本地开发时，缓慢的增量编译时间会严重拖慢您的开发速度，并使使用 TypeScript 变得痛苦。

为了解决这个问题，TSC 提供了一个叫做*项目引用*的功能，它可以显著加快编译时间，包括增量编译时间。对于任何有几百个文件或更多的项目，项目引用都是必不可少的。

使用方法如下：

1. 将您的 TypeScript 项目拆分成多个项目。一个项目只是一个包含 *tsconfig.json* 和一些 TypeScript 代码的文件夹。尽量以这样的方式拆分您的代码：倾向于一起更新的代码位于同一个文件夹中。

2. 在每个项目文件夹中，创建一个至少包含以下内容的 *tsconfig.json*：

```
{
  "compilerOptions": {
    "composite": true,
    "declaration": true,
    "declarationMap": true,
    "rootDir": "."
  },
  "include": [
    "./**/*.ts"
  ],
  "references": [
    {
      "path": "../myReferencedProject",
      "prepend": true
    }
  ],
}
```

这里的关键配置项是：

- `composite`，告诉 TSC 这个文件夹是更大 TypeScript 项目的子项目。
- `declaration`，告诉 TSC 为这个项目生成 *.d.ts* 声明文件。项目引用的工作方式是，项目可以访问彼此的声明文件和生成的 JavaScript，但不能访问它们的源 TypeScript 文件。这创建了一个边界，TSC 不会尝试超越这个边界重新类型检查或重新编译您的代码：如果您在子项目 *A* 中更新一行代码，TSC 不必重新类型检查您的其他子项目 *B*；TSC 检查类型错误所需的只是 *B* 的类型声明。这是使项目引用在重建大型项目时如此高效的核心行为。
- `declarationMap`，告诉 TSC 为生成的类型声明构建 source maps。
- `references`，这是您的子项目依赖的子项目数组。每个引用的 `path` 应该指向包含 *tsconfig.json* 的文件夹，或者直接指向 TSC 配置文件（如果您的配置文件不叫 *tsconfig.json*）。`prepend` 会将您引用的子项目生成的 JavaScript 和 source maps 与您的子项目生成的 JavaScript 和 source maps 连接起来。注意 `prepend` 只有在您使用 `outFile` 时才有用—如果您不使用 `outFile`，可以舍弃 `prepend`。
- `rootDir`，明确指定这个子项目应该相对于根项目（`.`）进行编译。或者，您可以指定一个 `outDir`，它是根项目 `outDir` 的子文件夹。

3. 创建一个根 *tsconfig.json*，引用任何尚未被其他子项目引用的子项目：

```
{
  "files": [],
  "references": [
    {"path": "./myProject"},
    {"path": "./mySecondProject"}
  ]
}
```

4. 现在当您使用 TSC 编译项目时，使用 `build` 标志告诉 TSC 考虑项目引用：

```
tsc --build # 或者，简写为 tsc -b
```

警告

在撰写本文时，项目引用是 TypeScript 的一个新功能，有一些粗糙的边缘。在使用它们时，请注意：

- 在克隆或重新获取项目后重建整个项目（使用 `tsc -b`），以便重新生成任何丢失或过时的 *.d.ts* 文件。或者，检入您生成的 *d.ts* 文件。
- 不要在项目引用中使用 `noEmitOnError: false`—TSC 总是会将选项硬编码为 `true`。
- 手动确保给定的子项目不会被超过一个其他子项目前置。否则，被双重前置的子项目会在编译输出中出现两次。注意如果你只是引用而不是前置，那就没问题。

使用extends减少tsconfig.json样板代码

因为你可能希望所有子项目共享相同的编译器选项，所以在根目录中创建一个”基础”*tsconfig.json*很方便，子项目的*tsconfig.json*可以扩展它：

```
{
  "compilerOptions": {
    "composite": true,
    "declaration": true,
    "declarationMap": true,
    "lib": ["es2015", "es2016.array.include"],
    "rootDir": ".",
    "sourceMap": true,
    "strict": true,
    "target": "es5",
  }
}
```

然后，使用*tsconfig.json*的 `extends` 选项更新你的子项目来扩展它：

```
{
  "extends": "../tsconfig.base",
  "include": [
    "./**/*.ts"
  ],
```

```
  "references": [
    {
      "path": "../myReferencedProject",
      "prepend": true
    }
  ],
}
```

错误监控

TypeScript会在编译时警告你错误，但你还需要一种方法来发现用户在运行时遇到的异常，这样你就可以尝试在编译时防止它们（或至少修复导致运行时错误的bug）。使用Sentry或Bugsnag等错误监控工具来报告和整理你的运行时异常。

在服务器上运行TypeScript

要在NodeJS环境中运行你的TypeScript代码，只需将你的代码编译为ES2015 JavaScript（如果你的目标是旧版NodeJS版本则编译为ES5），并将*tsconfig.json*的模块标志设置为`commonjs`：

```
{
  "compilerOptions": {
    "target": "es2015",
    "module": "commonjs"
  }
}
```

这将把你的ES2015 `import`和`export`调用分别编译为`require`和`module.exports`，这样你的代码就可以在NodeJS上运行，无需进一步打包。

如果你正在使用源映射（你应该使用！），你需要将源映射输入到你的NodeJS进程中。只需从NPM获取`source-map-support`包，并按照包的设置说明操作。大多数进程监控、日志记录和错误报告工具，如PM2、Winston和Sentry，都内置了对源映射的支持。

在浏览器中运行TypeScript

编译TypeScript在浏览器中运行比在服务器上运行TypeScript需要更多工作。

首先，选择一个要编译到的模块系统。一个好的经验法则是在发布供他人使用的库时（例如在NPM上）坚持使用`umd`，以最大化与人们可能在项目中使用的各种模块打包器的兼容性。

如果你只是打算自己使用代码而不将其发布到NPM，你编译到的格式取决于你使用的模块打包器。查看你的打包器文档——例如，Webpack和Rollup最适合ES2015模块，而Browserify需要CommonJS模块。以下是一些指导原则：

- 如果你使用SystemJS模块加载器，将`module`设置为`systemjs`。
- 如果你通过ES2015感知的模块打包器（如Webpack或Rollup）运行代码，将`module`设置为`es2015`或更高版本。
- 如果你使用ES2015感知的模块打包器且代码使用动态导入（参见”动态导入”），将`module`设置为`esnext`。
- 如果你正在构建供其他项目使用的库，并且在`tsc`之后不通过任何额外的构建步骤运行代码，通过将`module`设置为`umd`来最大化与人们使用的不同加载器的兼容性。
- 如果你使用CommonJS打包器（如Browserify）打包模块，将`module`设置为`commonjs`。
- 如果你计划使用RequireJS或其他AMD模块加载器加载代码，将`module`设置为`amd`。
- 如果你希望你的顶级导出在`window`对象上全局可用（就像你是墨索里尼的侄孙一样），将`module`设置为`none`。注意TSC会通过编译为`commonjs`来抑制你给其他软件工程师造成痛苦的热情，如果你的代码处于模块模式（参见[“模块模式与脚本模式”]）。

接下来，配置你的构建管道将所有TypeScript编译为单个JavaScript文件（通常称为”bundle”）或一组JavaScript文件。虽然TSC可以通过`outFile`TSC标志为小型项目做到这一点，但该标志仅限于生成SystemJS和AMD bundles。由于TSC不像Webpack这样的专用构建工具那样支持构建插件和智能代码拆分，你很快就会发现自己需要一个更强大的打包器。

这就是为什么对于前端项目，你应该从一开始就使用更强大的构建工具。无论你使用什么构建工具，都有TypeScript插件，例如：

- Webpack的`ts-loader`
- Browserify的`tsify`
- Babel的`@babel/preset-typescript`
- Gulp的`gulp-typescript`
- Grunt的`grunt-ts`

虽然关于优化JavaScript bundle以实现快速加载的完整讨论超出了本书的范围，但一些简要的建议——不特定于TypeScript——是：

- 保持代码模块化，避免代码中的隐式依赖（当你将东西分配给`window`全局对象或其他全局对象时可能发生这种情况），这样你的构建工具可以更准确地分析项目的依赖关系图。
- 使用动态导入来懒加载初始页面加载不需要的代码，这样你就不会不必要地阻塞页面渲染。
- 利用构建工具的自动代码拆分功能，这样你可以避免加载过多的JavaScript并不必要地减慢页面加载速度。
- 制定测量页面加载时间的策略，无论是综合的还是理想情况下使用真实用户数据。随着你的应用增长，初始加载时间可能会变得越来越慢；只有当你有方法测量它时，你才能优化那个加载时间。像New Relic和Datadog这样的工具在这里是无价的。
- 保持你的生产构建与开发构建尽可能相似。两者分歧越大，你遇到的只在生产环境中出现的难以修复的bug就越多。
- 最后，当发布TypeScript在浏览器中运行时，制定一个填充缺失浏览器功能的策略。这可能是作为每个bundle一部分发布的标准polyfills集，或者是基于用户浏览器支持功能的动态polyfills集。

将你的TypeScript代码发布到NPM

编译你的TypeScript代码以便其他TypeScript和JavaScript项目可以使用它是很容易的。在编译为JavaScript供外部使用时，有几个最佳实践需要记住：

- 生成source maps，这样你可以调试自己的代码。
- 编译到ES5，这样其他人可以轻松构建和运行你的代码。
- 注意你编译到的模块格式（UMD、CommonJS、ES2015等）。
- 生成类型声明，这样其他TypeScript用户可以获得你代码的类型。

首先用`tsc`将你的TypeScript编译为JavaScript，并生成相应的类型声明。确保配置你的*tsconfig.json*以最大化与流行JavaScript环境和构建系统的兼容性（更多内容参见[“构建你的TypeScript项目”]）：

```
{
"compilerOptions": {
  "declaration": true,
  "module": "umd",
  "sourceMaps": true,
  "target": "es5"
  }
}
```

接下来，在你的*.npmignore*中将TypeScript源代码列入黑名单，避免其发布到NPM从而增大包的大小。在你的*.gitignore*中，从Git仓库中排除生成的构件以避免污染它：

```
# .npmignore

*.ts # 忽略 .ts 文件
!*.d.ts # 允许 .d.ts 文件
```

```
# .gitignore

*.d.ts # 忽略 .d.ts 文件
*.js # 忽略 .js 文件
```

注意

如果你坚持推荐的项目布局并将源文件保存在*src/*中，将生成的文件保存在*dist/*中，你的*.ignore*文件会更简单：

```
# .npmignore

src/ # 忽略源文件
```

```
# .gitignore

dist/ # 忽略生成的文件
```

最后，在项目的*package.json*中添加一个`"types"`字段来表明它带有类型声明（注意这不是强制性的，但对使用TypeScript的消费者来说是TSC的一个有用提示），并添加一个脚本在发布前构建你的包，以确保包的JavaScript、类型声明和source maps始终保持最新并与编译它们的TypeScript同步：

```
{
  "name": "my-awesome-typescript-project",
  "version": "1.0.0",
  "main": "dist/index.js",
```

```
  "types": "dist/index.d.ts",
  "scripts": {
    "prepublishOnly": "tsc -d"
```

```
  }
}
```

就是这样！现在当你使用 `npm publish` 将包发布到 NPM 时，NPM 会自动将你的 TypeScript 编译为既能被使用 TypeScript 的人使用（具有完整的类型安全），也能被使用 JavaScript 的人使用（如果他们的代码编辑器支持，也会有一定的类型安全）的格式。

三斜杠指令(Triple-Slash Directives)

TypeScript 包含一个鲜为人知、很少使用且大多过时的功能，称为*三斜杠指令*。这些指令是特殊格式的 TypeScript 注释，作为对 TSC 的指示。

它们有几种类型，在本节中，我们将只介绍其中两种：用于省略仅类型完整模块导入的 `types`，以及用于命名生成的 AMD 模块的 `amd-module`。完整参考请参见[附录 E]。

types 指令

当你从模块中导入某些内容时，根据你导入的内容，TypeScript 在将代码编译为 JavaScript 时不一定需要生成 `import` 或 `require` 调用。如果你有一个 `import` 语句，其导出仅在模块的类型位置使用（即，你只是从模块中导入了一个类型），TypeScript 不会为该 `import` 生成任何 JavaScript 代码——可以将其视为仅存在于类型级别。这个功能称为 *导入省略 (import elision)*。

规则的例外是用于副作用的导入：如果你导入整个模块（而不是从该模块导入特定导出或通配符），该导入在编译 TypeScript 时会生成 JavaScript 代码。例如，如果你想确保脚本模式模块中定义的环境类型在程序中可用（就像我们在[“安全地扩展原型”]中所做的那样），你可能会这样做。例如：

```
// global.ts
type MyGlobal = number

// app.ts
import './global'
```

使用 `tsc app.ts` 将 *app.ts* 编译为 JavaScript 后，你会注意到 [`./global`] 导入没有被省略：

```
// app.js
import './global'
```

如果你发现自己在编写这样的导入，你可能想首先确保你的导入确实需要使用副作用，并且没有其他方法重写代码来更明确地表示你正在导入哪个值或类型（例如，`import {MyType} from './global'` ——TypeScript 会为你省略这个——而不是 `import './global'` ）。或者，看看是否可以在 *tsconfig.json* 的 `types` 、 `files` 或 `include` 字段中包含你的环境类型，完全避免导入。

如果这些都不适用于你的用例，你想继续使用完整模块导入但避免为该导入生成 JavaScript `import` 或 `require` 调用，请使用 `types` 三斜杠指令。三斜杠指令是三个斜杠 `///` 后

跟几个可能的 XML 标签之一，每个标签都有自己的必需属性集。对于 `types` 指令，它看起来像这样：

- 声明对环境类型声明的依赖：

```
/// <reference types="./global" />
```

- 声明对 *@types/jasmine/index.d.ts* 的依赖：

```
/// <reference types="jasmine" />
```

你可能不会经常使用这个指令。如果你确实使用了，你可能需要重新思考如何在项目中使用类型，并考虑是否有办法减少对环境类型的依赖。

amd-module 指令

当将 TypeScript 代码编译为 AMD 模块格式时（在 *tsconfig.json* 中用 `{"module": "amd"}` 表示），TypeScript 默认会生成匿名 AMD 模块。你可以使用 AMD 三斜杠指令为发出的模块命名。

假设你有以下代码：

```
export let LogService = {
  log() {
    // ...
  }
}
```

编译为 `amd` 模块格式，TSC 生成以下 JavaScript 代码：

```
define(['require', 'exports'], function(require, exports) {
  exports.__esModule = true
  exports.LogService = {
    log() {
      // ...
    }
  }
})
```

如果你熟悉 AMD 模块格式，你可能已经注意到这是一个匿名 AMD 模块。要为 AMD 模块命名，请在代码中使用 `amd-module` 三斜杠指令：

```
/// <amd-module name="LogService" />
export let LogService = {
  log() {
    // ...
```

```
  }
}
```

我们使用 `amd-module` 指令，并在其上设置 `name` 属性。
我们的其余代码保持不变。

使用 TSC 重新编译为 AMD 模块格式，我们现在得到以下 JavaScript：

```
/// <amd-module name='LogService' />
define('LogService', ['require', 'exports'], function(require,
exports) {
  exports.__esModule = true
  exports.LogService = {
    log() {
      // ...
    }
  }
})
```

当编译为 AMD 模块时，使用 `amd-module` 指令来使您的代码更容易打包和调试（或者，如果可以的话，切换到更现代的模块格式，如 ES2015 模块）。

总结

在本章中，我们涵盖了在生产环境中构建和运行 TypeScript 应用程序所需了解的一切，无论是在浏览器中还是在服务器上。我们讨论了如何选择要编译的 JavaScript 版本，如何标记在您的环境中可用的库（以及当库缺失时如何进行 polyfill），以及如何构建和发布源映射与您的应用程序，以便在生产环境中更容易调试并在本地开发。然后我们探讨了如何模块化您的 TypeScript 项目以保持编译时间快速。最后，我们以如何在服务器和浏览器中运行您的 TypeScript 应用程序、如何将您的 TypeScript 代码发布到 NPM 供他人使用、导入消除(import elision)如何工作，以及对于 AMD 用户如何使用三斜线指令(triple-slash directives)来命名您的模块结束。

[1] 如果您使用了 TSC 不会转译且您的目标环境也不支持的语言特性，您通常可以找到 Babel 插件来为您转译它。要找到最新的插件，请在您喜爱的搜索引擎中搜索”babel plugin <特性名称>“。

第13章 结论

我们的共同旅程即将结束。

我们已经涵盖了什么是类型以及它们为何有用；TSC 如何工作；TypeScript 支持什么类型；TypeScript 的类型系统如何处理推断、可赋值性、细化、扩展和完整性；上下文类型化的规则；协变如何工作；以及如何使用类型操作符。我们涵盖了函数和类和接口、迭代器和可迭代对象和生成器、重载、多态类型、混入、装饰器，以及您可以偶尔使用的各种逃生舱，以牺牲安全性在截止日期前完成代码。我们探讨了安全处理异常的不同方式及其权衡，以及如何使用类型使并发、并行和异步程序安全。我们深入研究了将 TypeScript 与 Angular 和 React 等流行框架一起使用，以及命名空间和模块如何工作。我们研究了在前端和后端使用、构建和部署 TypeScript，并讨论了如何逐步将代码迁移到 TypeScript、如何使用类型声明、如何将您的代码发布到 NPM 以供他人使用、如何安全使用第三方代码，以及如何构建您的 TypeScript 项目。

我希望我已经用静态类型的福音感染了您。我希望您现在有时会发现自己在实现程序之前先用类型勾勒程序，我希望您对如何使用类型使应用程序更安全有了深度直觉的理解。我希望我已经改变了您对世界的看法，至少一点点，并且您现在在编写代码时会以类型的方式思考。

您现在有能力向他人传授 TypeScript。倡导安全性，帮助使您的公司和朋友的代码变得更好、更有趣。

最后，继续探索。TypeScript 可能不是您的第一门语言，它也可能不会是您的最后一门。继续学习编程的新方法、思考类型的新方法，以及思考安全性和易用性之间权衡的新方法。也许您会创造出 TypeScript 之后的下一个重大突破，也许有一天我会是那个写相关内容的人…

附录A 类型操作符

TypeScript 支持丰富的类型操作符集合来处理类型。使用表 A-1 作为方便的参考，当您想要了解更多关于某个操作符的信息时。

类型操作符	语法	用于	了解更多
类型查询	`typeof`, `instanceof`	任何类型	[“细化”],[“类声明值和类型”]
键	`keyof`	对象类型	[“keyof 操作符”]
属性查找	`O[K]`	对象类型	[“keying-in 操作符”]
映射类型	`[K in O]`	对象类型	[“映射类型”]

修饰符添加 `+` 对象类型 [“映射类型”] 修饰符减法 `-` 对象类型 [“映射类型”] 只读修饰符 `readonly` 对象类型、数组类型、元组类型 [“对象”], [“类和继承”], [“只读数组和元组”] 可选修饰符 `?` 对象类型、元组类型、函数参数类型 [“对象”], [“元组”], [“可选参数和默认参数”] 条件类型 `?` 泛型类型、类型别名、函数参数类型 [“条件类型”] 非 `null` 断言 `!` 可空类型 [“非空断言”], [“明确赋值断言”] 泛型类型参数默认值 `=` 泛型类型 [“泛型类型默认值”] 类型断言 `as`、`<>` 任何类型 [“类型断言”], [“const类型”] 类型守卫 `is` 函数返回类型 [“用户定义的类型守卫”]

: [表A-1。]类型操作符 {#calibre_link-521}

[附录B。]类型工具

TypeScript的类型工具被打包到其标准库中。[表B-1]枚举了编写时所有可用的工具。

参见*es5.d.ts*获取最新参考。

类型工具 使用	对象 描述	
`ConstructorParameters` 类构造函数类	型 类构造函数参数类型的元	组
`Exclude` 联合类型	从另一个类型中排	除一个类型
`Extract` 联合类型	选择可赋值给另一	个类型的子类型
`InstanceType` 类构造函数类	型 通过 `new` 实例化类构造	函数得到的实例类型
`NonNullable` 可空类型	从类型中排除`nul	l 和 undefined`
`Parameters` 函数类型	函数参数类型的元	组
`Partial` 对象类型	使对象中的所有属	性变为可选
`Pick` 对象类型	对象类型的子类型	，包含其键的子集
`Readonly` 数组、对象和	元组类型 使对象中的所有属性变为	只读，或使数组或元组变为只读
`ReadonlyArray` 任何类型	创建给定类型的不	可变数组
`Record` 对象类型	从键类型到值类型	的映射
`Required` 对象类型	使对象中的所有属	性变为必需

类型工具 使用	对象 描述
`ReturnType` 函数类型	函数的返回类型

：[表B-1。]类型工具 {#calibre_link-534}

[附录C。]作用域声明

TypeScript声明具有丰富的行为集合，用于建模类型和值，就像在JavaScript中一样，它们可以以多种方式重载。本附录涵盖了其中两种行为，总结了哪些声明生成类型(以及哪些生成值)，以及哪些声明可以合并。

它是否生成类型?

一些TypeScript声明创建类型，一些创建值，一些两者都创建。参见[表C-1]快速参考。

关键字 生	成类型? 生成值?
`class` 是	是
`const`、`let`、`var` 否 是	
`enum` 是	是
`function` 否	是
`interface` 是	否
`namespace` 否	是
`type` 是	否

: [表C-1。]声明是否生成类型? {#calibre_link-536}

它是否合并?

声明合并是TypeScript的核心行为。利用它来创建更丰富的API，更好地模块化你的代码，并让你的代码更安全。

[表C-2]转载自[“声明合并”]；它是TypeScript为你合并哪种声明的便捷参考。

[表 C-2.]声明能否合并？ {#calibre_link-537}

	到									
值	类	枚举值	枚举	函数	类	型别名	接口	命名空间	模块	—
类	否	否	否	—	—	否	否	否	否	—
枚举值	否	否	否	—	—	否	否	是	否	—
枚举	否	否	否	—	—	否	否	否	否	—
函数	否	否	否	—	—	否	—	—	—	—
类	否	否	否	—	—	否	—	—	—	—
型别名	否	否	否	—	—	否	—	—	—	—
接口	否	否	否	—	—	否	否	否	否	—
命名空间	否	否	否	—	—	否	否	是	是	—
模块	否	否	否	—	—	否	否	是	是	—
—	是	—	—	—	—	—	—	—	—	—

[附录 D.]为第三方JavaScript模块编写声明文件的方法

本附录涵盖了在为第三方模块进行类型声明时反复出现的几个关键构建块和模式。有关为第三方代码进行类型声明的更深入讨论，请参阅[“DefinitelyTyped上没有类型声明的JavaScript”]。

由于模块声明文件必须位于 *.d.ts* 文件中，因此不能包含值，当你声明模块类型时，需要使用 `declare` 关键字来确认给定类型的值确实由你的模块导出。[表 D-1] 提供了常规声明及其类型声明等效项的简短摘要。

[表 D-1.]TypeScript及其仅类型等效项 {#calibre_link-539}

.ts	*.d.ts*
`var a = 1` `declare var	a: number`
`let a = 1` `declare let	a: number`
`const a = 1` `declare con	st a: 1`
`function a(b) { return b.toFixed() }` `declare fun	ction a(b: number): string`
`class A { b() { return 3 } }` `declare cla	ss A { b(): number }`
`namespace A {}` `declare nam	espace A {}`
`type A = number` `type A = nu	mber`
`interface A { b?: string }` `interface A	{ b?: string }`

导出类型

你的模块是使用全局、ES2015还是CommonJS导出将影响你编写声明文件的方式。

全局导出

如果你的模块只向全局命名空间分配值而实际上不导出任何内容，你可以直接创建一个脚本模式文件（参见[“模块模式与脚本模式”]）并在变量、函数和类声明前添加 `declare` 前缀（其他类型的声明—`enum`、`type` 等—保持不变）：

```
// 全局变量
declare let someGlobal: GlobalType

// 全局类
declare class GlobalClass {}

// 全局函数
declare function globalFunction(): string

// 全局枚举
enum GlobalEnum {A, B, C}

// 全局命名空间
namespace GlobalNamespace {}

// 全局类型别名
type GlobalType = number

// 全局接口
interface GlobalInterface {}
```

这些声明中的每一个都将在你的项目的每个文件中全局可用，而不需要显式导入。在这里，你可以在项目的任何文件中使用 `someGlobal` 而无需先导入它，但在运行时，`someGlobal` 需要被分配到全局命名空间（浏览器中的 `window` 或 NodeJS 中的 `global`）。

注意避免在声明文件中使用 `import` 和 `export`，以保持文件处于脚本模式。

ES2015导出

如果你的模块使用ES2015导出—即 `export` 关键字—只需将 `declare` （确认定义了全局变量）替换为 `export` （确认导出了ES2015绑定）：

```
// 默认导出
declare let defaultExport: SomeType
export default defaultExport

// 命名导出
export class SomeExport {
  a: SomeOtherType
}

// 类导出
export class ExportedClass {}

// 函数导出
export function exportedFunction(): string

// 枚举导出
enum ExportedEnum {A, B, C}

// 命名空间导出
export namespace SomeNamespace {
  let someNamespacedExport: number
}

// 类型导出
export type SomeType = {
  a: number
}

// 接口导出
export interface SomeOtherType {
  b: string
}
```

CommonJS导出

CommonJS在ES2015之前是事实上的模块标准，在撰写本文时仍然是NodeJS的标准。它也使用 `export` 关键字，但语法略有不同：

```
declare let defaultExport: SomeType
export = defaultExport
```

注意我们是如何将导出分配给 `export`，而不是将 `export` 用作修饰符（就像我们对ES2015导出所做的那样）。

第三方CommonJS模块的类型声明只能包含一个导出。要导出多个内容，我们利用声明合并（参见[附录 C]）。

例如，要为多个导出进行类型声明而没有默认导出，我们导出单个[`namespace`]：

```
declare namespace MyNamedExports {
  export let someExport: SomeType
  export type SomeType = number
  export class OtherExport {
    otherType: string
  }
}
export = MyNamedExports
```

那么对于既有默认导出又有命名导出的 CommonJS 模块呢？我们可以利用声明合并：

```
declare namespace MyExports {
  export let someExport: SomeType
  export type SomeType = number
}
declare function MyExports(a: number): string
export = MyExports
```

UMD 导出

为 UMD 模块添加类型几乎与为 ES2015 模块添加类型相同。唯一的区别是，如果你想让你的模块对脚本模式文件全局可用（参见[“模块模式与脚本模式”]），你需要使用特殊的 `export as namespace` 语法。例如：

```
// 默认导出
declare let defaultExport: SomeType
export default defaultExport

// 命名导出
export class SomeExport {
  a: SomeType
}

// 类型导出
export type SomeType = {
  a: number
}

export as namespace MyModule
```

注意最后一行——如果你的项目中有脚本模式文件，现在可以在全局 `MyModule` 命名空间上直接使用该模块（无需先导入）：

```
let a = new MyModule.SomeExport
```

扩展模块

扩展模块的类型声明比为模块添加类型要少见，但如果你编写 JQuery 插件或 Lodash mixin 时可能会遇到。尽可能避免这样做；相反，考虑使用独立的模块。也就是说，不使用 Lodash mixin 而使用常规函数，不使用 JQuery 插件——等等，你为什么还在使用 JQuery?

全局变量

如果你想扩展另一个模块的全局命名空间或接口，只需创建一个脚本模式文件（参见[“模块模式与脚本模式”]），然后增强它。注意这只对接口和命名空间有效，因为 TypeScript 会为你处理合并。

例如，让我们为 JQuery 添加一个很棒的新 `marquee` 方法。我们先安装 `jquery` 本身：

```
npm install jquery --save
npm install @types/jquery --save-dev
```

然后在项目中创建一个新文件——比如 *jquery-extensions.d.ts*——并将 `marquee` 添加到 JQuery 的全局 `JQuery` 接口（我通过查看其类型声明发现 JQuery 在 `JQuery` 接口上定义其方法）：

```
interface JQuery {
  marquee(speed: number): JQuery<HTMLElement>
}
```

现在，在任何使用 JQuery 的文件中，我们都可以使用 `marquee`（当然，我们也要为 `marquee` 添加运行时实现）：

```
import $ from 'jquery'
$(myElement).marquee(3)
```

注意这与我们在[“安全地扩展原型”]中用来扩展内置全局变量的技术相同。

模块

扩展模块导出更加复杂，并且有更多陷阱：你需要正确地为扩展添加类型，在运行时以正确的顺序加载模块，并确保当你要扩展的模块的类型声明结构发生变化时更新你的扩展类型。

作为示例，让我们为 React 添加一个新的导出类型。我们先安装 React 及其类型声明：

```
npm install react --save
npm install @types/react --save-dev
```

然后我们利用模块合并（参见[“声明合并”]）并简单地声明一个与我们的 React 模块同名的模块：

```
import {ReactNode} from 'react'

declare module 'react' {
  export function inspect(element: ReactNode): void
}
```

注意与扩展全局变量的示例不同，我们的扩展文件是在模块模式还是脚本模式并不重要。

那么扩展模块中的特定导出呢？受 ReasonReact 启发，假设我们想为 React 组件添加一个内置的 reducer（reducer 是为 React 组件声明一组明确状态转换的方法）。在撰写本文时，React 的类型声明将 `React.Component` 类型声明为接口和类，它们被合并为单个 UMD 导出：

```
export = React
export as namespace React

declare namespace React {
  interface Component<P = {}, S = {}, SS = any>
    extends ComponentLifecycle<P, S, SS> {}
  class Component<P, S> {
    constructor(props: Readonly<P>)
    // ...
```

```
  }
  // ...
}
```

让我们用 `reducer` 方法扩展 `Component`。我们可以在项目根目录的 *react-extensions.d.ts* 文件中输入以下内容来实现：

```
import 'react'

declare module 'react' {
  interface Component<P, S> {
    reducer(action: object, state: S): S
  }
}
```

❶

我们导入 `'react'`，将扩展文件切换到脚本模式，这是消费 React 模块所需的。注意还有其他方法可以切换到脚本模式，比如导入其他东西、导出某些东西或导出空对象（`export {}`）——我们不一定要专门导入 `'react'`。

❷

我们声明 `'react'` 模块，向 TypeScript 表明我们想为该特定 `import` 路径声明类型。因为我们已经安装了 `@types/react`（它为完全相同的 `'react'` 路径定义了导出），TypeScript 会将此模块声明与 `@types/react` 提供的声明合并。

❸

我们通过声明自己的 `Component` 接口来增强 React 提供的 `Component` 接口。遵循接口合并规则（[“声明合并”]），我们必须在声明中使用与 `@types/react` 中完全相同的签名。

❹

最后，我们声明 `reducer` 方法。

在声明这些类型后（并假设我们已经在某处实现了支持此更新的运行时行为），现在可以以类型安全的方式声明带有内置 `reducers` 的 React 组件：

```
import * as React from 'react'

type Props = {
  // ...
}

type State = {
  count: number
  item: string
}
```

```
type Action =
  | {type: 'SET_ITEM', value: string}
  | {type: 'INCREMENT_COUNT'}
  | {type: 'DECREMENT_COUNT'}

class ShoppingBasket extends React.Component<Props, State> {
  reducer(action: Action, state: State): State {
    switch (action.type) {
      case 'SET_ITEM':
        return {...state, item: action.value}
      case 'INCREMENT_COUNT':
        return {...state, count: state.count + 1}
      case 'DECREMENT_COUNT':
        return {...state, count: state.count - 1}
    }
  }
}
```

如本节开始所述，尽可能避免使用这种模式是一个好的做法（尽管它很酷），因为它会使你的模块变得脆弱并依赖于加载顺序。相反，尝试使用组合，使你的模块扩展消费它们正在扩展的模块，并导出一个包装器而不是修改该模块。

附录E. 三斜杠指令

三斜杠指令(triple-slash directives)只是常规的JavaScript注释，TypeScript查找它们来执行诸如为特定文件调整编译器设置，或指示你的文件依赖于另一个文件等操作。将你的指令放在文件顶部，在任何代码之前。三斜杠指令如下所示（每个指令是一个三斜杠，`///`，后跟一个XML标签）：

```
/// <directive attr="value" />
```

TypeScript支持一些三斜杠指令。表E-1列出了你最可能使用的指令：

`amd-module`
前往[“amd-module指令”]了解更多。

`lib`
`lib`指令是向TypeScript指示你的模块依赖于TypeScript的哪些`lib`库的方式，如果你的项目没有*tsconfig.json*，你可能想要这样做。在你的*tsconfig.json*中声明你依赖的`lib`库几乎总是更好的选择。

`path`
当使用TSC的`outFile`选项时，使用`path`指令来声明对另一个文件的依赖，以便其他文件在你编译的输出中比依赖文件出现得更早。如果你的项目使用`import`和`export`，你可能永远不会使用这个指令。

`type`
前往[“types指令”]了解更多关于`type`指令的信息。

指令	语法	用途
`amd-module`	`<amd-module name="MyComponent" />`	在编译到AMD模块时声明导出名称
`lib`	`<reference lib="dom" />`	声明你的类型声明依赖于TypeScript内置的哪些`lib`

指令	语法	用途
		库
`path`	`<reference path="./path.ts" />`	声明你的模块依赖于哪些TypeScript文件
`type`	`<reference types="./path.d.ts" />`	声明你的模块依赖于哪些类型声明文件

表E-1. 三斜杠指令

内部指令

你可能永远不会在自己的代码中使用`no-default-lib`指令（表E-2）。

指令	语法	用途
`no-default-lib`	`<reference no-default-lib="true" />`	告诉TypeScript对此文件完全不使用任何`lib`库

表E-2. 内部三斜杠指令

已弃用的指令

你永远不应该使用`amd-dependency`指令（表E-3），而应该坚持使用常规的`import`。

指令	语法	替代使用
`amd-dependency`	`<amd-dependency path="./a.ts" name="MyComponent" />`	`import`

表E-3. 已弃用的三斜杠指令

附录F. TSC编译器安全标志

提示

有关可用编译器标志的完整列表，请访问TypeScript手册网站。

每个TypeScript发布版本都会引入新的检查，你可以启用这些检查从你的代码中获得更多安全性。其中一些标志（以`strict`为前缀）作为`strict`标志的一部分包含在内；或者，你可以一次选择加入一个`strict`标志。表F-1列出了在撰写时可用的与安全性相关的编译器标志。

标志	描述
`alwaysStrict`	发出`'use strict'`。
`noEmitOnError`	当代码有类型错误时不发出JavaScript。
`noFallthroughCasesInSwitch`	确保每个`switch` case要么返回值要么break。
`noImplicitAny`	当变量类型被推断为`any`时报错。
`noImplicitReturns`	确保每个函数中的每个代码路径都明确返回。参见[“完整性”]。
`noImplicitThis`	当在函数中使用`this`而没有明确注释`this`类型时报错。参见[“为this添加类型”]。
`noUnusedLocals`	警告未使用的局部变量。
`noUnusedParameters`	警告未使用的函数参数。在参数名前加`_`前缀可忽略此错误。
`strictBindCallApply`	为`bind`、`call`和`apply`强制类型安全。参见[“call、apply和bind”]。

标志	描述
`strictFunctionTypes`	强制函数在其参数和 `this` 类型上是逆变的。参见[“函数变性”]。
`strictNullChecks`	将 `null` 提升为类型。参见[“null、undefined、void和never”]。
`strictPropertyInitialization`	强制类属性要么是可空的要么被初始化。参见[第5章]。

表F-1. TSC安全标志

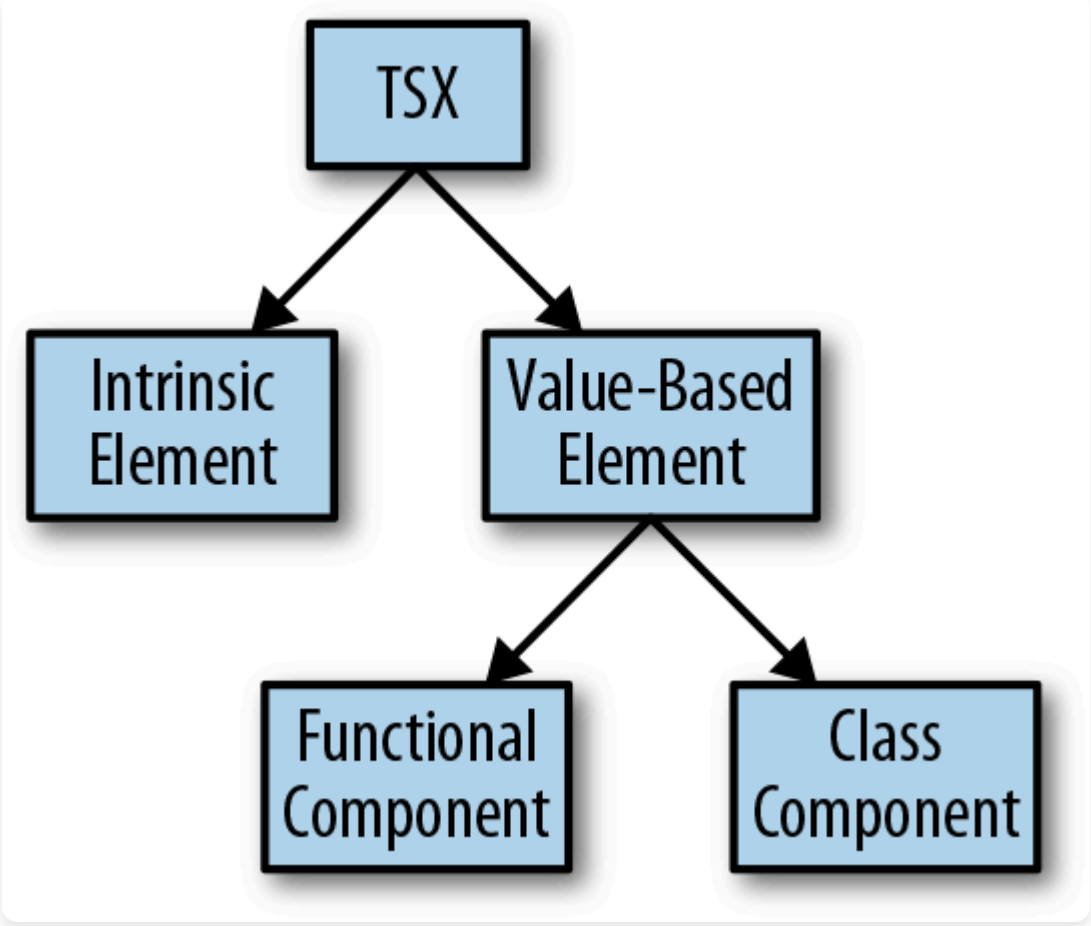
附录G. TSX

在底层，TypeScript暴露了一些钩子，以可插拔的方式为TSX添加类型。这些是`global.JSX`命名空间上的特殊类型，TypeScript将其视为整个程序中TSX类型的权威来源。

注意

如果您只是使用React，您不需要了解这些底层钩子，但如果您正在编写一个不使用React而使用TSX的TypeScript库，本附录为您可以使用的钩子提供了有用的参考。

TSX支持两种元素：内置元素（*内在元素(intrinsic elements)*）和用户定义元素（*基于值的元素(value-based elements)*）。内在元素总是具有小写名称，指的是内置元素，如`<li>`、`<h1>`和`<div>`。基于值的元素具有帕斯卡命名的名称，指的是您使用React（或任何您正在与TSX一起使用的前端框架）创建的元素；它们可以定义为函数或类。参见图G-1。



图G-1. TSX元素的种类

以React的类型声明为例，我们将介绍TypeScript用于安全地为TSX添加类型的钩子。以下是React如何钩入TSX来安全地为JSX添加类型：

```
declare global {
  namespace JSX {
    interface Element extends React.ReactElement<any> {}
    interface ElementClass extends React.Component<any> {
      render(): React.ReactNode
    }
    interface ElementAttributesProperty {
      props: {}
    }
    interface ElementChildrenAttribute {
      children: {}
    }
```

```
    type LibraryManagedAttributes<C, P> = // ...

    interface IntrinsicAttributes extends React.Attributes {}
    interface IntrinsicClassAttributes<T> extends
React.ClassAttributes<T> {}

    interface IntrinsicElements {
      a: React.DetailedHTMLProps<
        React.AnchorHTMLAttributes<HTMLAnchorElement>,
        HTMLAnchorElement
      >
      abbr: React.DetailedHTMLProps<
        React.HTMLAttributes<HTMLElement>,
        HTMLElement
      >
      address: React.DetailedHTMLProps<
        React.HTMLAttributes<HTMLElement>,
        HTMLElement
      >
      // ...
    }
  }
}
```

❶

`JSX.Element` 是基于值的TSX元素的类型。

❷

`JSX.ElementClass` 是基于值的类组件实例的类型。每当您声明一个计划使用TSX的 `<MyComponent />` 语法实例化的类组件时，其类必须满足此接口。

❸

`JSX.ElementAttributesProperty` 是TypeScript查看以确定组件支持哪些属性的属性名称。对于React，这意味着 `props` 属性。TypeScript在类实例上查找此值。

❹

`JSX.ElementChildrenAttribute` 是TypeScript查看以确定组件支持哪些类型的子元素的属性名称。对于React，这意味着 `children` 属性。

❺

`JSX.IntrinsicAttributes` 是所有内在元素支持的属性集合。对于React，这意味着 `key` 属性。

❻

`JSX.IntrinsicClassAttributes` 是所有类组件（内在的和基于值的）支持的属性集合。对于React，这意味着 `ref` 。

❼

[翻译内容]

- [可赋值性(assignability)], [谈论类型], [可赋值性]
  - [对于枚举], [枚举]
  - [推断的泛型类型到显式绑定的泛型], 泛型类型推断
- [赋值，this变量], [类型化this]
- [AST（抽象语法树）], [编译器]
- [异步和等待语法], [异步编程、并发和并行], async和await
- [异步编程], [异步编程、并发和并行]-类型安全多线程
  - async和await, async和await
  - [异步流], [异步流]-类型安全多线程
    - [事件发射器], [事件发射器]-类型安全多线程
  - [回调函数], [使用回调函数]-[使用回调函数]
  - [JavaScript的事件循环], [JavaScript的事件循环]-[JavaScript的事件循环]
  - [Promise], [使用Promise重获理智]-[使用Promise重获理智]
- [异步加载模块], [JavaScript模块简史]

B

C

    - [装饰器], [装饰器]-[装饰器]
    - 设计模式, 设计模式-[建造者模式]
        - [建造者模式], [建造者模式]
        - 工厂模式, 工厂模式
    - [实现接口], 实现
        - [与扩展抽象类的区别], 实现接口与扩展抽象类
    - [混入(mixins)], [混入]-[装饰器]
    - 多态性, 多态性
    - 模拟final类, 模拟final类
    - [结构化类型], 类是结构化类型的
    - super调用, [super]
    - [使用this作为返回类型], [使用this作为返回类型]
- [经典模式(moduleResolution标志)], 声明合并
- [Angular中的CLI（命令行界面）], [Angular 6/7]
    - [安装], [脚手架]
- [客户端/服务器通信], [类型安全API]-[类型安全API]
- [代码编辑器], [何时检查类型？]
    - [为TypeScript设置代码编辑器], [代码编辑器设置]-[tslint.json]
- [本书的代码示例], 使用代码示例
- [代码分割], [动态导入]
- [代码风格]
    - [为TSLint配置], [tslint.json]
    - [在本书中], 风格
- [CommonJS模块设置(TSC)], [调整编译目标]
- [CommonJS模块标准], [JavaScript模块简史]
    - [使用CommonJS的JavaScript模块], [使用CommonJS和AMD代码]
    - [CommonJS的问题], [JavaScript模块简史]
- [伙伴对象模式], [伙伴对象模式], 声明合并

- [TypeScript项目的编译目标], [调整编译目标]-lib
  - [target设置], target
- [编译器]
  - [angularCompilerOptions], [组件]
  - [AoT Angular编译器], [服务]
  - [TypeScript编译器], [编译器]-[编译器]
- [组件]
  - [Angular组件], [组件]
  - [React组件], [React]
    - [函数和类组件], [在React中使用TSX]
- [composite（编译器选项）], 项目引用
- [具体类型], 多态性
  - [绑定到泛型类型参数], 多态性
- 条件类型, 条件类型-[逃生出口]

和

- [TS2365], [介绍], [类型是否会自动转换？]
- [TS2366], 完整性
- [TS2367], [symbol(符号)]
- [TS2393], [冲突]
- [TS2428], 声明合并
- [TS2430], 接口
- [TS2454], [确定赋值断言]
- [TS2469], [symbol(符号)]
- [TS2476], [枚举]
- [TS2511], 类和继承
- [TS2531], 非空断言
- [TS2532], [对象]

- [TS2540], [对象]
- [TS2542], [只读数组和元组]
- [TS2554], [声明和调用函数], [剩余参数], [使用有界多态性建模参数数量]
- [TS2558], 泛型类型推断
- [TS2571], [unknown(未知)]
- [TS2684], [类型化this]
- [TS2706], 泛型类型默认值
- [TS2717], 声明合并
- [TS2739], [Record类型]
- [TS2741], [对象], [对象]
- [TS7006], [上下文类型化]
- [TS7030], 完整性

E

- [experimentalDecorators TSC标志], [装饰器]
- [exports(导出)]
  - [CommonJS], [JavaScript模块简史]
  - [ES2015标准], [JavaScript模块简史]-[import, export]
  - [模块模式与脚本模式], [模块模式与脚本模式]
  - [module.exports], [JavaScript模块简史]
  - [namespace(命名空间)], [命名空间]
  - [使用CommonJS和AMD代码], [使用CommonJS和AMD代码]
- [extends关键字], 类和继承
- [extends选项(TSC)], 项目引用

F

- [factory pattern(工厂模式)], 工厂模式
- [filesystem(文件系统)，作为文件名的模块路径], [import, export]
- [final classes(最终类)，模拟], [模拟最终类]
- [finally子句], async和await
- [fixed-arity functions(固定参数数量函数)], [剩余参数]
- [flow-based type inference(基于流的类型推断)], [细化]
- [folder structure(文件夹结构)，TypeScript项目的], [index.ts], 项目布局
- [formal parameters(形式参数)] （[见]参数）
- [frameworks(框架)], [前端和后端框架]-总结
  - [backend(后端)], [后端框架]
  - [frontend(前端)], [前端和后端框架]-[服务]
    - [Angular 6/7], [Angular 6/7]-[服务]
    - [在tsconfig.json中启用DOM API], [前端框架]
    - [React], [React]-[在React中使用TSX]
  - [typesafe APIs(类型安全API)], [类型安全API]-[类型安全API]

- [fresh object literal type(新鲜对象字面量类型)], [多余属性检查]
- [fullTemplateTypeCheck TSC标志], [组件]
- [Function类型], 调用签名
- [functions(函数)], [函数]-练习, 高级函数类型-[用户定义类型保护]
    - [绑定泛型], 泛型何时绑定?
    - [声明和调用], [声明和调用函数]-重载函数类型
        - [call signatures(调用签名)], 调用签名-调用签名
        - [contextual typing(上下文类型化)], [上下文类型化]
        - [使用命名函数语法声明], [声明和调用函数]
        - [generator functions(生成器函数)], 生成器函数
        - [调用函数], [声明和调用函数]
        - [使用call、apply和bind调用], call、apply和bind
        - [iterators(迭代器)], 迭代器-迭代器
        - [optional和default参数], [可选和默认参数]
        - [overloaded function types(重载函数类型)], 重载函数类型-重载函数类型
        - [rest parameters(剩余参数)], [剩余参数]
        - [类型化this变量], [类型化this]
    - [decorator(装饰器)], [装饰器]
    - [改进元组类型推断], [改进元组类型推断]
    - [null、undefined、void和never返回类型], [null、undefined、void和never]
    - [重载环境函数声明], [冲突]
    - [parameter types(参数类型)], [讨论类型]
    - [polymorphism(多态性)], 多态性-类型驱动开发
        - [bounded(有界)], 有界多态性-[使用有界多态性建模参数数量]
        - [generic type defaults(泛型类型默认值)], 泛型类型默认值
    - [type-driven development(类型驱动开发)], 类型驱动开发
    - [user-defined type guards(用户定义类型保护)], [用户定义类型保护]
    - [variance(变型)], [函数变型]

        - [covariant return types(协变返回类型)], [函数变型]

G

- [generator functions(生成器函数)], 生成器函数
- [generic type parameters(泛型类型参数)], 多态性
    - （[另见]泛型类型；泛型）
    - [作为约束], 多态性
    - [尖括号内的逗号分隔列表], 多态性
    - [命名约定], 多态性
- [generic types(泛型类型)], 多态性
    - [Promise], [使用Promise重获理智]
- [generics(泛型)], 多态性
    - [绑定，何时发生], 泛型何时绑定？-[在哪里可以声明泛型？]
    - 有界多态性, 有界多态性-[使用有界多态性建模参数数量]
    - [类和接口对其的支持], 多态性-[混入]
- [声明为条件的一部分], [条件类型(Conditional Types)], [infer关键字(The infer Keyword)]
    - [接口声明(interface declaring)], [声明合并(Declaration Merging)]
    - [子类型(subtyping)], [变异性(Variance)]
    - [类型别名(type aliases)], [泛型类型别名(Generic Type Aliases)]
    - [类型默认值(type defaults)], [泛型类型默认值(Generic Type Defaults)]
    - [类型推断(type inference)], [泛型类型推断(Generic Type Inference)]-[泛型类型别名(Generic Type Aliases)]
- [全局命名空间(global namespace)], [安全扩展原型(Safely Extending the Prototype)], [编译输出(Compiled Output)]
- [全局JSX命名空间(global.JSX namespace)], [TSX = JSX + TypeScript]
- [全局变量(浏览器)(globals (browser))], [JavaScript模块简史(A Brief History of JavaScript Modules)], [使用CommonJS和AMD代码(Using CommonJS and AMD Code)]
- [渐进式类型语言(gradually typed languages)], [类型如何绑定？(How are types bound?)]

H

- [同构数组(homogeneous arrays)], [数组(Arrays)]
- [HTML模板(Angular)(HTML templates (Angular))], [组件(Components)]

I

- [立即调用函数表达式(IIFEs)(immediately invoked function expressions (IIFEs))], [JavaScript模块简史(A Brief History of JavaScript Modules)]
- [不可变数组(immutable arrays)], [只读数组和元组(Read-only arrays and tuples)]
- [implements关键字(implements keyword)], [实现(Implementations)]
- [import函数(import function)], [动态导入(Dynamic Imports)]
- [import语句(import statements)], [安全扩展原型(Safely Extending the Prototype)], [动态导入(Dynamic Imports)]
- [导入(imports)]
  - [CommonJS], [JavaScript模块简史(A Brief History of JavaScript Modules)]
  - [动态(dynamic)], [动态导入(Dynamic Imports)]
  - [ES2015标准(ES2015 standard for)], [JavaScript模块简史(A Brief History of JavaScript Modules)]-[import, export]
  - [导入省略(import elision)], [types指令(The types Directive)]
  - [模块模式与脚本模式(module mode versus script mode)], [模块模式与脚本模式(Module Mode Versus Script Mode)]
  - [精确导入路径的模块名(module name for exact import path)], [环境模块声明(Ambient Module Declarations)]
  - [使用CommonJS和AMD代码(using CommonJS and AMD code)], [使用CommonJS和AMD代码(Using CommonJS and AMD Code)]
  - [将无类型导入列入白名单(whitelisting an untyped import)], [DefinitelyTyped上没有类型声明的JavaScript(JavaScript That Doesn’t Have Type Declarations on DefinitelyTyped)]
- [索引签名(index signatures)], [对象(Objects)]
  - [Record对象(Record object)], [Record类型(The Record Type)]

- [index.ts], [index.ts]
- [infer关键字(infer keyword)], [infer关键字(The infer Keyword)]
- [推断类型(inferring types)], [类型系统(The Type System)]
- [继承(inheritance)], [类和继承(Classes and Inheritance)]
    - [模拟多重继承(simulating multiple inheritance)], [混入(Mixins)]
- [内部节点(inner nodes)], [有界多态性(Bounded Polymorphism)]
- [实例方法(instance methods)], [类和继承(Classes and Inheritance)]
- [instanceof操作符(instanceof operator)], [类同时声明值和类型(Classes Declare Both Values and Types)]
- [接口(interfaces)], [接口(Interfaces)]-[实现接口与扩展抽象类(Implementing Interfaces Versus Extending Abstract Classes)]
    - [Angular生命周期钩子(Angular lifecycle hooks)], [组件(Components)]
    - [泛型绑定(binding of generics)], [泛型何时绑定？(When Are Generics Bound?)]
    - [与类型别名的比较(comparison with type aliases)], [接口(Interfaces)]
    - [声明合并(declaration merging)], [声明合并(Declaration Merging)]
    - [扩展对象、类或其他接口(extending objects, classes, or other interfaces)], [接口(Interfaces)]
    - [实现(implementation)], [实现(Implementations)]
    - [实现与扩展抽象类(implementing versus extending abstract classes)], [实现接口与扩展抽象类(Implementing Interfaces Versus Extending Abstract Classes)]
    - [合并(merging)], [声明合并(Declaration Merging)]
    - [多态性(polymorphism)], [多态性(Polymorphism)]
    - [类型声明文件中的顶级接口(top-level, in type declaration files)], [类型声明(Type Declarations)]
- [内部三斜杠指令(internal triple-slash directives)], [内部指令(Internal Directives)]
- [解释型语言(interpreted languages)], [编译器(The Compiler)]
- [交集类型(intersection types)], [联合类型和交集类型(Union and intersection types)], [接口(Interfaces)]
    - [数组(array)], [数组(Arrays)]

- [内在元素(TSX)(intrinsic elements (TSX))], [TSX]
- [无效操作(invalid actions)], [介绍(Introduction)]
- [不变性(invariance)], [形状和数组变异性(Shape and array variance)]
- [同构程序(isomorphic programs)], [调整编译目标(Dialing In Your Compile Target)]
- [可迭代迭代器(iterable iterators)], [迭代器(Iterators)]
- [IterableIterator类型(IterableIterator type)], [生成器函数(Generator Functions)]
- [可迭代对象(iterables)], [迭代器(Iterators)]
- [迭代器(iterators)], [迭代器(Iterators)]-[迭代器(Iterators)]
  - [常见集合类型的内置迭代器(built-in, for common collection types)], [迭代器(Iterators)]
  - [定义(defined)], [迭代器(Iterators)]

J

- [JavaScript], [前言(Preface)]
  - [集合类型的内置迭代器(built-in iterators for collection types)], [迭代器(Iterators)]
  - [TypeScript项目的编译目标(compile target for TypeScript projects)], [调整编译目标(Dialing In Your Compile Target)]-lib
    - [语言版本和发布(language versions and releases)], target
  - [配置构建管道将TypeScript代码编译为JavaScript(configuring build pipeline to compile TypeScript code to)], [在浏览器中运行TypeScript(Running TypeScript in the Browser)]
  - [引擎(engine)], [编译器(The Compiler)]
  - [事件循环(event loop)], [JavaScript的事件循环(JavaScript’s Event Loop)]-[JavaScript的事件循环(JavaScript’s Event Loop)]
  - [与JavaScript互操作(interoperating with)], [与JavaScript互操作(Interoperating with JavaScript)]-[总结(Summary)]
    - [逐步将代码迁移到TypeScript(gradually migrating code to TypeScript)], [从JavaScript逐步迁移到TypeScript(Gradually Migrating from JavaScript to TypeScript)]-[步骤4：使其严格(Step 4: Make It strict)]
    - [JavaScript的类型查找(type lookup for JavaScript)], [JavaScript的类型查找(Type Lookup for JavaScript)]-[使用第三方JavaScript(Using Third-Party JavaScript)]

        - [使用第三方JavaScript(using third-party JavaScript)], [使用第三方JavaScript(Using Third-Party JavaScript)]-[DefinitelyTyped上没有类型声明的JavaScript(JavaScript That Doesn’t Have Type Declarations on DefinitelyTyped)]
        - [使用类型声明(using type declarations)], [类型声明(Type Declarations)]-[环境模块声明(Ambient Module Declarations)]
        - [从TypeScript使用JavaScript的方式(ways to use JavaScript from TypeScript)], [总结(Summary)]
    - [最新语法和特性(latest syntax and features)], [风格(Style)]
    - [模块，简史(modules, brief history of)], [JavaScript模块简史(A Brief History of JavaScript Modules)]-[JavaScript模块简史(A Brief History of JavaScript Modules)]
    - [优化包以实现快速加载(optimizing bundles for fast loading)], [在浏览器中运行TypeScript(Running TypeScript in the Browser)]
    - [类型系统，与TypeScript的比较(type system, comparison to TypeScript)], [TypeScript与JavaScript(TypeScript Versus JavaScript)]-[错误何时浮出水面？(When are errors surfaced?)]
    - [编译为JavaScript的TypeScript程序(TypeScript program compiled to)], [产物(Artifacts)]
- [JSDoc注解(JSDoc annotations)], [步骤2b：添加JSDoc注解（可选）(Step 2b: Add JSDoc Annotations (Optional))]
- [JSON（JavaScript对象表示法）(JSON (JavaScript Object Notation))]
    - [加载.json文件(loading .json files)], [环境模块声明(Ambient Module Declarations)]
- [JSX（JavaScript XML）(JSX (JavaScript XML))], [JSX入门(A JSX primer)]
    - [React钩入TSX以正确类型化JSX(React hooking into TSX to type JSX correctly)], [TSX]
    - [TSX和(TSX and)], [TSX = JSX + TypeScript]
    - [将TSX与React一起使用(using TSX with React)], [将TSX与React一起使用(Using TSX with React)]-[将TSX与React一起使用(Using TSX with React)]
- [jsx指令(jsx directive)], [TSX = JSX + TypeScript]

K

- [键入操作符(keying-in operator)], [键入操作符(The keying-in operator)], [infer关键字(The infer Keyword)]

- [keyof操作符(keyof operator)], [keyof操作符(The keyof operator)]
- [keyofStringsOnly TSC标志(keyofStringsOnly TSC flag)], [keyof操作符(The keyof operator)]

L

- [LABjs，用于加载模块(LABjs, loading module with)], [JavaScript模块简史(A Brief History of JavaScript Modules)]
  - [懒加载代码(lazy-loading code)], [动态导入(Dynamic Imports)]
- [懒惰和异步加载模块(lazy and asynchronous loading modules)], [JavaScript模块简史(A Brief History of JavaScript Modules)]
  - [懒加载代码块(lazy loading chunks of code)], [动态导入(Dynamic Imports)]
- [以类型为先导(leading with the types)], [类型驱动开发(Type-Driven Development)]
- [叶节点(leaf nodes)], [有界多态性(Bounded Polymorphism)]
- [let]
  - [使用let的类型声明(type declarations with)], [总结(Summary)]
  - [使用const而不是let(using const instead of)], [布尔值(boolean)]
  - [使用let的变量声明(variable declarations with)], [类型别名(Type aliases)]
- [lib指令(lib directive)], 三斜杠指令(Triple-Slash Directives)
- [lib设置（TSC）(lib setting (TSC))], [环境变量声明(Ambient Variable Declarations)], [调整编译目标(Dialing In Your Compile Target)], lib
- [lib tsconfig.json选项(lib tsconfig.json option)], [前端框架(Frontend Frameworks)]
- [生命周期钩子（Angular）(lifecycle hooks (Angular))], [组件(Components)]
- [listeners, creating], [Event Emitters]
  - [Web Workers中的onmessage API], [在浏览器中：使用Web Workers]

M

- [映射类型(mapped types)], 映射类型-[伴生对象模式]
  - [内置], 内置映射类型
  - [用于构建类型安全事件发射器], [事件发射器]
- [MatrixProtocol], [类型安全协议]
- [内存安全(memory safety)], [在浏览器中：使用Web Workers]
- [消息传递（Web Workers）], [在浏览器中：使用Web Workers]
  - [消息传递API], [在浏览器中：使用Web Workers]
  - [onmessage API], [在浏览器中：使用Web Workers]
- [方法], 类和继承
  - [抽象], 类和继承
  - [访问修饰符], 类和继承
- [混入(mixins)], [混入]-[装饰器]
- [模块模式 vs 脚本模式], [模块模式 vs 脚本模式]
- [模块设置(TSC)], [调整编译目标]
- [模块设置(tsconfig.json)], [编译输出]
- [module.exports], [JavaScript模块简史]
- [moduleResolution TSC 标志], 声明合并
- 模块, [命名空间.模块]-[模块模式 vs 脚本模式]
  - [环境模块声明(ambient module declarations)], [环境模块声明]
  - [amd-module指令], [amd-module指令]
  - [JavaScript模块简史], [JavaScript模块简史]-[JavaScript模块简史]
  - [import, export], [import, export]-[import, export]
    - [动态导入(dynamic imports)], [动态导入]
    - [模块模式或脚本模式], [模块模式 vs 脚本模式]
    - [使用CommonJS和AMD代码], [使用CommonJS和AMD代码]
  - [从JavaScript导入的类型查找], [JavaScript类型查找]-使用第三方JavaScript
  - [tsconfig.json中的模块字段], [步骤1：添加TSC]
  - [选择将TypeScript编译为的模块系统], 在浏览器中运行TypeScript

    - [优先使用模块而非命名空间], [编译输出]
    - [告知TypeScript关于通过NPM安装的第三方模块], [类型声明]
    - [从第三方JavaScript导入无类型模块], [在DefinitelyTyped上没有类型声明的JavaScript]
    - [为第三方JavaScript模块编写声明文件], [为第三方JavaScript模块编写声明文件的方法]-模块
- [Moment.js日期操作库], [动态导入]
- [多线程], 类型安全多线程-[在NodeJS中：使用子进程]
    - [在浏览器中使用Web Workers], [在浏览器中：使用Web Workers]-[类型安全协议]
    - [在NodeJS中使用子进程的类型安全多线程], [在NodeJS中：使用子进程]-[在NodeJS中：使用子进程]

N

- [命名函数语法], [声明和调用函数]
- [namespace关键字], [命名空间]
- [命名空间], [命名空间]-[编译输出]
    - [冲突], [冲突]
    - [编译输出], [编译输出]
    - [导出], [命名空间]
    - [长命名空间，使用别名缩短], [命名空间]
    - [合并], 声明合并
    - [优先使用模块而非命名空间], [编译输出]
    - [子命名空间], [命名空间]
- [never类型], [null、undefined、void和never], [分布式条件类型]
    - [使用总结], [null、undefined、void和never]
- [new操作符], 类同时声明值和类型
    - [扩展类构造器的new(…any[])语法], [装饰器]
- [next方法], 迭代器
- [no-invalid-this TSLint规则], [this类型]

- [Redis的Node API], [事件发射器]
- [node模式（moduleResolution标志）], 声明合并
- [NodeJS]
  - [基于回调的fs.readFile API], [使用回调]-[使用回调]
  - [EventEmitter], [异步流], [在浏览器中：使用Web Workers]
  - [安装和运行], [代码编辑器设置]
  - 模块, [JavaScript模块简史]
  - [优先使用模块而非命名空间], [编译输出]
  - [在NodeJS中运行TypeScript代码], 在服务器上运行TypeScript
  - [使用子进程的类型安全多线程], [在NodeJS中：使用子进程]-[在NodeJS中：使用子进程]
  - [将基于回调的fs.readFile API包装为基于Promise的API], [使用Promise恢复理性]
- [节点（二叉树）], 有界多态性
- [tsconfig.json中的noImplicitAny标志], any
- [tsconfig.sh中的noImplicitAny标志], [步骤2a：为JavaScript启用类型检查（可选）]
- [noImplicitReturns TSC标志], 完整性
- [noImplicitThis TSC选项], [this类型]
- [名义类型，模拟], 模拟名义类型-模拟名义类型
- [名义类型系统(nominal typing)], [对象]
- [非空断言(nonnull assertions)], 非空断言
- [NPM包管理器], [代码编辑器设置]
  - [安装第三方JavaScript], 使用第三方JavaScript
  - [将TypeScript代码发布到NPM], [将TypeScript代码发布到NPM]-[将TypeScript代码发布到NPM]
  - [发送类型声明到NPM], [在DefinitelyTyped上没有类型声明的JavaScript]
- [null类型], [null、undefined、void和never]
  - [使用示例], [null、undefined、void和never]
  - [返回null的函数], 返回null

    - 非空断言, 非空断言
    - [编程语言中的问题], [null、undefined、void和never]
    - [严格null检查], [null、undefined、void和never]
    - [使用总结], [null、undefined、void和never]
    - [初始化为null的变量，类型拓宽], [类型拓宽]
- [number类型], [类型全览], number
- [数字，数字分隔符], number

O

- [对象字面量], [对象]
    - [新鲜对象字面量类型(fresh object literal type)], [多余属性检查]
- [对象关系映射器(ORMs)], [后端框架]
- [对象], [对象]-[插曲：类型别名、联合与交叉], 高级对象类型-[伴生对象模式]
    - [伴生对象模式], [伴生对象模式]
    - [协变成员], [形状和数组变异性]
    - [在TypeScript中声明], [对象]
    - [空对象], [对象]
- [过滤数组], 多态性
- [索引签名], [对象]
- [某些语言中的不变属性类型], [形状和数组变异性]
- 映射类型, 映射类型-[伴生对象模式]
- [属性], [对象]
- [Record类型], [Record类型]
- [形状], [对象]
- 对象类型的类型运算符, 对象类型的类型运算符-[keyof运算符]
    - 键入运算符, 键入运算符
    - [keyof运算符], [keyof运算符]

- [可观察对象(Observables)], [异步流]
- [选项类型(Option type)], [选项类型]-[选项类型]
  - [flatMap和getOrElse方法], [选项类型]
  - [在Some<T>和None类中的实现], [选项类型]
  - [在Some<T>和None上实现flatMap和getOrElse], [选项类型]
  - [实现Option函数], [选项类型]
  - [使用重载签名为flatMap提供更具体的类型], [选项类型]
- [ORM(对象关系映射器)], [后端框架]
- [outFile TSC标志], 在浏览器中运行TypeScript
- [重载环境函数声明], [冲突]
- 重载函数类型, 重载函数类型-重载函数类型
  - [过滤函数], 多态性
  - [保持重载签名的具体性], 重载函数类型

P

- [并行性], [类型安全的多线程]
  - ([另见] 多线程)
- [参数]
  - [参数与参数的兼容性], [声明和调用函数]
  - [函数参数的逆变性], [函数变异性]
  - [定义], [声明和调用函数]
  - [泛型类型参数], 多态性
  - [可选和默认参数], [可选和默认参数]
  - [JavaScript类型推断中的可选函数参数], [步骤2a：为JavaScript启用类型检查（可选）]
  - [剩余参数], [剩余参数]
  - [类型参数], [讨论类型]
- [路径指令], [三斜线指令]

- [路径（模块）], [导入、导出], [动态导入], 声明合并
- [polyfill], [调整编译目标]
  - [process.env.NODE_ENV的polyfill], [环境变量声明]
  - [目标环境不支持较新库功能的polyfill], lib
- 多态性, 多态性-类型驱动开发
  - [泛型的绑定], 泛型何时绑定？-[在哪里可以声明泛型？]
  - [有界], 有界多态性-[使用有界多态性建模元数]
    - [用于建模元数], [使用有界多态性建模元数]
    - [具有多个约束], 具有多个约束的有界多态性
  - [类和接口], 多态性-[混入]
  - 泛型类型别名, 泛型类型别名
  - 泛型类型默认值, 泛型类型默认值
  - 泛型类型推断, 泛型类型推断-泛型类型别名
  - [泛型类型参数], 多态性
- [preserveConstEnums TSC标志], [枚举]
- [private（访问修饰符）], 类和继承
  - [类构造函数], [模拟最终类]
  - [具有私有字段的类], 类是结构化类型的
- 项目引用, 项目引用-错误监控
- [项目，为TypeScript设置], [index.ts]
- [promise], [用Promise恢复理智]-[用Promise恢复理智]
  - [Promise实现], [用Promise恢复理智]
    - [泛型类型], [用Promise恢复理智]
    - [用错误拒绝Promise], [用Promise恢复理智]
    - [用then和catch序列化Promise], [用Promise恢复理智]
  - [由import函数返回], [动态导入]
  - [与async和await一起使用], async和await
  - [将基于回调的NodeJS API fs.readFile包装为基于Promise的API], [用Promise恢复理智]

- [属性]
  - [访问修饰符], 类和继承
  - [多余属性检查], [多余属性检查]
  - [函数属性，用完整调用签名建模], 重载函数类型
  - [对象和类属性，协变性], [形状和数组变异性]
  - [JavaScript代码的类型推断], [步骤2a：为JavaScript启用类型检查（可选）]
- [protected（访问修饰符）], 类和继承
  - [具有受保护字段的类], 类是结构化类型的
- [协议]
  - [协议上的客户端/服务器通信], [类型安全的API]
  - [类型安全协议], [类型安全协议], [类型安全的API]
- [原型，安全扩展], [安全扩展原型]-总结
- [public（访问修饰符）], 类和继承
  - [方法], 类和继承

R

- [React], [React]-[与React一起使用TSX]
  - [TSX中的hooks安全类型化JSX], [TSX]
  - [JSX入门], [JSX入门]
  - [TypeScript中的TSX或JSX支持], [TSX = JSX + TypeScript]
  - [与React一起使用TSX], [与React一起使用TSX]
- [响应式编程库], [异步流]
- [readonly修饰符], [对象]
  - [与数组和元组一起使用], [只读数组和元组]
- [ReadonlyArray类型], [只读数组和元组]
- [Record类型], [Record类型]
- [Redis库，使用], [事件发射器]

S

    - [在Promise中], [使用Promise恢复理智]
- [服务(Angular)], [服务]
- [形状]
    - [子类型和超类型规则], [形状和数组协变]
    - [子类型化], [形状和数组协变]
- [简写调用签名], 重载函数类型
- [源码映射], [构建产物]
    - [启用], [启用源码映射]
    - [注入到NodeJS进程], 在服务器上运行TypeScript
- [SQL调用], [后端框架]
- [src/index.ts], [index.ts]
- [TSC中的严格标志], any, [TSC编译器安全标志]
- [严格模式(TSC)], [步骤2a：为JavaScript启用类型检查(可选)]
- [strictBindCallApply选项], call、apply和bind
- [strictFunctionTypes TSC标志], [函数协变]
- [strictNullChecks和strictPropertyInitialization TSC标志], 类和继承
- [strictNullChecks选项], [null、undefined、void和never]
- [string类型], [关于类型的一切], string
- [字符串]
    - [JavaScript中的自动类型转换], [类型会自动转换吗？]
    - [过滤字符串数组], [多态]
- [结构化类型], [对象]
    - [TypeScript中的类], 类是结构化类型的

- [子类型], [子类型和超类型]
  - [可赋值性与], [可赋值性]
  - [更复杂类型的子类型规则], [协变]
    - [形状和数组], [形状和数组协变]
- super调用, [super]
- [超类型], [子类型和超类型]
  - [可赋值性与], [可赋值性]
  - [在期望超类型的地方使用对象类型], [形状和数组协变]
- [symbol类型], symbol
  - [unique symbol的使用示例], symbol
- [Symbol.iterator], 迭代器

T

- [联合类型的标签], [区分联合类型]
- [target设置(TSC)], [调整编译目标]
- [任务], [JavaScript的事件循环]
- [测试，依赖测试来发现错误], [介绍]
- [then子句], [使用Promise恢复理智]
  - [使用async和await替代], async和await
- [第三方JavaScript，使用], 使用第三方JavaScript-[在DefinitelyTyped上没有类型声明的JavaScript]
- [this变量], call、apply和bind
  - [函数this类型中的逆变], [函数协变]
  - [类型], [为this添加类型]

    - [用作方法返回类型], [使用this作为返回类型]
- 完整性, 完整性-完整性
- [特征], [混入]
    - (另见混入)
- [转译], [调整编译目标]
    - [TSC为JavaScript功能转译的支持情况], target
    - [TSC对转译代码到旧版JavaScript的支持], [调整编译目标]
- [三斜线指令], [三斜线指令]-[amd-module指令], [三斜线指令]-[废弃的指令]
    - [amd-module], [amd-module指令]
    - [types], [types指令]
- [true和false值(boolean类型)], boolean
- [Try类型], [Option类型]
- [try/catch语句], 抛出异常, [使用Promise恢复理智]
- [TSC编译器], [编译器], [代码编辑器设置]
    - [生成的构建产物], [构建产物]
    - [checkJs标志], [步骤2a：为JavaScript启用类型检查(可选)]
    - [编译JavaScript代码], [步骤1：添加TSC]
    - [使用tsconfig.json配置], [tsconfig.json]
    - [declarations标志], [类型声明]
    - [downlevelIteration标志], 迭代器
    - [启用DOM APIs], [前端框架]
    - [为dom库启用], [在浏览器中：使用Web Workers]
    - [启用noImplicitAny标志], [步骤2a：为JavaScript启用类型检查(可选)]
    - [为webworker库启用], [在浏览器中：使用Web Workers]
    - [esModuleInterop标志], [JSX入门], [使用CommonJS和AMD代码]

    - [experimentalDecorators标志], [装饰器]
    - [fullTemplateTypeCheck标志], [组件]
    - [生成JavaScript包], 在浏览器中运行TypeScript
    - [使用npm安装], [代码编辑器设置]
    - [jsx标志], [TSX = JSX + TypeScript]
    - [keyofStringsOnly标志], [keyof操作符]
    - [lib设置], [环境变量声明]
    - [moduleResolution标志], 声明合并
    - [namespace输出], [编译输出]
    - [noImplicitReturns标志], 完整性
    - [noImplicitThis选项], [为this添加类型]
    - [preserveConstEnums选项], [枚举]
    - 项目引用, 项目引用-错误监控
    - [安全标志], [TSC编译器安全标志]
    - [设置以调整目标环境], [调整编译目标]
    - [strictBindCallApply选项], call、apply和bind
    - [strictFunctionTypes标志], [函数协变]
    - [strictNullChecks和strictPropertyInitialization标志], 类和继承
    - [strictNullChecks选项], [null、undefined、void和never]
    - [types和typeRoots设置], [JavaScript的类型查找]
    - [在迁移的JavaScript代码上使用更严格的标志], [步骤4：使其严格]
- [tsconfig.json], [tsconfig.json]
    - [启用noImplicitAny标志], any
    - [lib选项], [前端框架]
    - [module:esnext], [动态导入]
    - [选项], [tsconfig.json]
- [TSLint]

    - [使用tslint.json配置], [tslint.json]
    - [安装], [代码编辑器设置]
    - [no-angle-bracket-type-assertion规则], [类型断言]
    - [no-invalid-this规则], [为this添加类型]
- [tslint.json], [tslint.json]
- [TSX]
    - [参考], [TSX]-[TSX]
    - [与React一起使用], [与React一起使用TSX]-[与React一起使用TSX]
- [元组], [元组]
    - 改进元组的类型推断, 改进元组的类型推断
    - [可选元素], [元组]
    - [只读], [只读数组和元组]
    - [剩余元素], [元组]
- [类型别名], [类型别名]
- [泛型绑定], [何时绑定泛型？]
- [与接口比较], 接口
- [用于命名空间], [命名空间]
- [泛型], 泛型类型别名
- [类型标记], 模拟名义类型
- [类型声明], [类型声明]-[环境模块声明], [构建产物]
    - [环境模块声明], [环境模块声明]
    - [环境类型声明], [环境类型声明]
    - [环境变量声明], [环境变量声明]
    - [为第三方JavaScript创建类型声明并发布到NPM], [DefinitelyTyped上没有类型声明的JavaScript]

    - [与常规TypeScript语法的区别], [类型声明]
    - [DOM类型声明，启用], lib
    - [与等效TypeScript代码的示例], [类型声明]
    - [DefinitelyTyped上的JavaScript类型声明], [DefinitelyTyped上有类型声明的JavaScript]
    - [DefinitelyTyped上没有类型声明的JavaScript代码], [DefinitelyTyped上没有类型声明的JavaScript]
    - [类型声明文件的用途], [类型声明]
- [类型守卫], [用户定义类型守卫]
- [类型推断]
    - 改进元组的类型推断, 改进元组的类型推断
    - [JavaScript代码中的类型], [步骤2a：为JavaScript启用类型检查（可选）]
- 类型字面量, boolean
- [类型操作符], [类型操作符]
    - [用于对象类型], [对象类型的类型操作符]-[keyof操作符]
        - [索引操作符], [索引操作符]
        - [keyof], [keyof操作符]
    - [- (减号) 和 + (加号)], 映射类型
- [类型细化] ([参见] 细化)
- 类型安全, [介绍]
- [类型签名] ([参见] 调用签名)
- [类型系统], [类型系统]-[何时出现错误？]
    - [JavaScript和TypeScript的比较], [TypeScript与JavaScript]-[何时出现错误？]
    - [TypeScript类型系统处理的内容], [结论]
- [类型工具], [类型工具]
- [类型扩宽], [类型扩宽]-[const类型]

    - [过度属性检查], [过度属性检查]
    - [使用const类型防止扩宽], [const类型]
- 类型驱动开发, 类型驱动开发
- [类型级代码], 调用签名
- [类型检查器], [编译器], [关于类型的一切]
- [类型检查], [编译器]
    - [Angular模板的类型检查], [组件]
    - [JavaScript代码类型检查], [步骤2a：为JavaScript启用类型检查（可选）]
    - [JavaScript与TypeScript的类型检查], [何时检查类型？]
- [typeof操作符], 类同时声明值和类型, [用户定义类型守卫]
    - [用于数组项], [数组]
- [类型]
    - 高级对象类型, 高级对象类型-[伴随对象模式]
        - [伴随对象模式], [伴随对象模式]
        - 映射类型, 映射类型-[伴随对象模式]
        - [Record], [Record类型]
        - [类型操作符], [对象类型的类型操作符]-[keyof操作符]
    - [绑定，JavaScript与TypeScript], [类型如何绑定？]
    - 类同时声明值和类型, 类同时声明值和类型
    - 条件类型, 条件类型-[逃生舱]
        - 内置条件类型, 内置条件类型
        - [使用infer关键字内联声明泛型], infer关键字
        - [分布式条件类型], [分布式条件类型]
    - [转换，JavaScript与TypeScript], [类型是否自动转换？]
    - [声明生成类型], [是否生成类型？]
    - [声明this变量的类型], [为this添加类型]

- [定义], [关于类型的一切]
- [DOM类型声明，启用], [前端框架]
- [逃生舱], [逃生舱]-明确赋值断言
  - 明确赋值断言, 明确赋值断言
  - 非空断言, 非空断言
  - 模拟名义类型, 模拟名义类型-模拟名义类型
  - [类型断言], [类型断言]-[类型断言]
- [安全扩展原型], [安全扩展原型]-总结
- [函数类型], 高级函数类型-[用户定义类型守卫]
  - 改进元组的类型推断, 改进元组的类型推断
  - [用户定义类型守卫], [用户定义类型守卫]
- [合并类型和值], 声明合并
- [类型之间的关系], [类型之间的关系]-判别联合类型
  - [可赋值性], [可赋值性]
  - [细化], [细化]-判别联合类型
  - [子类型和超类型], [子类型和超类型]
  - [类型扩宽], [类型扩宽]-[const类型]
  - [变异性], [变异性]-[函数变异性]
- [在类型中勾勒程序], [结论]
- [TypeScript支持的类型], 类型基础-总结
  - any, any
  - [数组], [数组]-[只读数组和元组]
  - bigint, bigint
  - boolean, boolean
  - [null、undefined、void和never], [null、undefined、void和never]-[null、undefined、void和never]
  - number, number
  - [对象], [对象]-[间奏：类型别名、联合和交集]

        - string, string
        - symbol, symbol
        - [元组], [元组]
        - [类型别名], [类型别名]
        - [类型及其子类型], 总结
        - [联合和交集类型], [联合和交集类型]
        - unknown, unknown
    - [类型的术语和词汇], [讨论类型]
    - [类型声明文件中的顶级类型], [类型声明]
    - 完整性, 完整性-完整性
    - [JavaScript的类型查找], [JavaScript的类型查找]-使用第三方JavaScript
- [types和typeRoots设置 (TSC)], [JavaScript的类型查找]
- [types指令], [types指令]
- [TypeScript]
    - [TypeScript的优势], [介绍]
    - [构建TypeScript项目], [构建你的TypeScript项目]-错误监控
        - [TSC生成的构建产物], [构建产物]
        - [调整编译目标], [调整编译目标]-lib
- [启用源映射], [启用源映射(Enabling Source Maps)]
- 错误监控, [错误监控(Error Monitoring)]
- 项目布局, [项目布局(Project Layout)]
- 项目引用, [项目引用(Project References)]-[错误监控(Error Monitoring)]
- [编译器], [TypeScript: 一万英尺视角(TypeScript: A 10_000 Foot View)]-[编译器(The Compiler)]
- [错误处理], [介绍(Introduction)]

- [逐步将 JavaScript 代码迁移到], [从 JavaScript 逐步迁移到 TypeScript(Gradually Migrating from JavaScript to TypeScript)]-[步骤 4: 启用严格模式(Step 4: Make It strict)]
    - [添加 JSDoc 注释], [步骤 2b: 添加 JSDoc 注释（可选）(Step 2b: Add JSDoc Annotations (Optional))]
    - [向 JavaScript 项目添加 TSC], [步骤 1: 添加 TSC(Step 1: Add TSC)]
    - [为 JavaScript 启用类型检查], [步骤 2a: 为 JavaScript 启用类型检查（可选）(Step 2a: Enable Typechecking for JavaScript (Optional))]
    - [将文件重命名为 .ts], [步骤 3: 将文件重命名为 .ts(Step 3: Rename Your Files to .ts)]
    - [使用严格的 TSC 标志], [步骤 4: 启用严格模式(Step 4: Make It strict)]
- [模块，使用和导出], [JavaScript 模块简史(A Brief History of JavaScript Modules)]
- [将代码发布到 NPM], [将 TypeScript 代码发布到 NPM(Publishing Your TypeScript Code to NPM)]-[将 TypeScript 代码发布到 NPM(Publishing Your TypeScript Code to NPM)]
- [在浏览器中运行], [在浏览器中运行 TypeScript(Running TypeScript in the Browser)]
- [在服务器上运行], [在服务器上运行 TypeScript(Running TypeScript on the Server)]
- [设置代码编辑器], [代码编辑器设置(Code Editor Setup)]-[tslint.json]
- [设置项目，index.ts 文件], [index.ts]
- [三斜杠指令], 三斜杠指令(Triple-Slash Directives)-[amd-module 指令(The amd-module Directive)]
- [类型系统], [类型系统(The Type System)]-[何时显示错误？(When are errors surfaced?)]
    - [与 JavaScript 的比较], [TypeScript 对比 JavaScript(TypeScript Versus JavaScript)]-[何时显示错误？(When are errors surfaced?)]
- [TypeSearch], [在 DefinitelyTyped 上具有类型声明的 JavaScript(JavaScript That Has Type Declarations on DefinitelyTyped)]

U

- [undefined 类型], [null、undefined、void 和 never]
  - [使用示例], [null、undefined、void 和 never]
  - 非空断言, [非空断言(Nonnull Assertions)]
  - [使用总结], [null、undefined、void 和 never]
  - [初始化为 undefined 的变量，类型扩展], [类型扩展(Type Widening)]
- [联合类型], [联合类型和交叉类型(Union and intersection types)]
  - [添加条件类型], [分布式条件类型(Distributive Conditionals)]
  - [为线程间通信定义], [在浏览器中：使用 Web Workers(In the Browser: With Web Workers)]
  - [可区分的], [可区分联合类型(Discriminated union types)]
- [唯一符号], [符号(symbol)], [模拟名义类型(Simulating Nominal Types)]
- [unknown 类型], unknown
- 用户定义的类型守卫, [用户定义的类型守卫(User-Defined Type Guards)]

V

- [基于值的元素 (TSX)], [TSX]
- [值级代码], [调用签名(Call Signatures)]
- [值: 类型注释], [类型系统(The Type System)]
- [值], [样式(Style)]
  - [声明值和类型的类], [类声明值和类型(Classes Declare Both Values and Types)]
  - [生成值的声明], [它是否生成类型？ (Does It Generate a Type?)]
  - [合并值和类型], [声明合并(Declaration Merging)]
  - [null 和 undefined], [null、undefined、void 和 never]
  - [类型声明文件中的顶级], [类型声明(Type Declarations)]
  - [类型声明和], [类型声明(Type Declarations)]
- [var]
  - [使用 var 的类型声明], [总结(Summary)]
  - [使用 const 代替], [布尔类型(boolean)]
  - [使用 var 的变量声明], [类型别名(Type aliases)]
- [变量]
  - [环境变量声明], [环境变量声明(Ambient Variable Declarations)]
  - [环境声明与常规声明], [类型声明(Type Declarations)]
  - [声明变量，然后用值初始化], [对象(Objects)]
  - [不可变变量，无类型扩展], [类型扩展(Type Widening)]
  - [可变变量，类型扩展], [类型扩展(Type Widening)]
  - [具有显式注释类型的], [类型系统(The Type System)]
  - [具有隐式推断类型的], [类型系统(The Type System)]
- [可变参数函数], [剩余参数(Rest Parameters)]
- [型变], [型变(Variance)]-[函数型变(Function variance)]

    - [函数], [函数型变(Function variance)]
    - [型变的种类], [形状和数组型变(Shape and array variance)]
    - [形状和数组], [形状和数组型变(Shape and array variance)]
- [前端应用程序的视图层 (React)], [React]
- [void 类型], [null、undefined、void 和 never]
    - [使用总结], [null、undefined、void 和 never]
- [VSCode], [代码编辑器设置(Code Editor Setup)]

W

X

- [XML]
  - [JSX (JavaScript XML)], [JSX 入门(A JSX primer)]
  - [三斜杠指令中的标签], 三斜杠指令(Triple-Slash Directives)
  - [TSX (JSX 和 TypeScript)], [TSX = JSX + TypeScript]
  - [在 React 中使用 TSX], [在 React 中使用 TSX(Using TSX with React)]-[在 React 中使用 TSX(Using TSX with React)]

Y

- [yield 关键字], [生成器函数(Generator Functions)]

关于作者

Boris Cherny 是 Facebook 的工程和产品负责人。此前，他在风投、广告技术和许多初创公司工作过，其中大部分已经不存在了。他对编程语言、代码合成和静态分析，以及构建人们喜爱的用户体验感兴趣。在业余时间，他运营旧金山 TypeScript 聚会，并在他的个人博客 *performancejs.com* 上写作。可以在 GitHub 上找到他：*https://github.com/bcherny*。

版权页

《Programming TypeScript》封面上的动物是原驼(*Lama guanicoe*)。原驼是骆驼的野生祖先，也与骆驼有亲缘关系。在绵羊被引入南美大陆之前，原驼遍布南美洲大部分地区。它们生活在干燥、陡峭的山区，海拔高达 13,000 英尺。如今大约 600,000 只原驼中的大部分生活在阿根廷，特别是巴塔哥尼亚地区。

为了在崎岖不平的斜坡地形中生存，原驼(guanaco)的每只脚上都有两个带垫的脚趾，并且重心较低。原驼肩部的平均高度不到四英尺。它们厚实的羊毛外衣呈浅到中等的红棕色，底部接近白色。长长的睫毛保护它们的眼睛免受强风侵袭，而大而尖的耳朵帮助它们感知威胁。

原驼成群出行，群体由几只雌性、一岁以下的幼崽和一只繁殖雄性组成。当它们感到威胁时，原驼会发出高音调的叫声来警告群体逃跑，有人说这听起来像吠叫般的笑声。美洲狮和狐狸捕食原驼，所以它们轮流站在山丘上尖叫，在需要逃到安全地带时警告群体的其他成员。原驼通常能以每小时35英里的速度奔跑。如果捕食者追赶群体，雄性会跑到后面进行防御。

原驼的怀孕期接近整整一年。一旦出生，被称为*chulengo*的幼崽只需要五分钟就能开始行走。在与父母生活一年后，幼崽必须找到自己的群体。这种突然的驱逐可能导致了原驼幼崽的低存活率——只有30%能达到成年。成年原驼可以活15到20年。

在原驼栖息地的崎岖山坡和灌木丛中，草类和植被坚韧厚实，是重要的水分来源。原驼有三室胃，帮助它们消化植物并更长时间地保存液体。它们的上唇分成两半——这种适应性让抓取食物变得更容易。

O’Reilly封面上的许多动物都濒临灭绝；它们对世界都很重要。

封面插图由Karen Montgomery绘制，基于《动物自然图像博物馆》中的黑白版画。封面字体是Gilroy Semibold和Guardian Sans。正文字体是Adobe Minion Pro；标题字体是Adobe Myriad Condensed；代码字体是Dalton Maag的Ubuntu Mono。