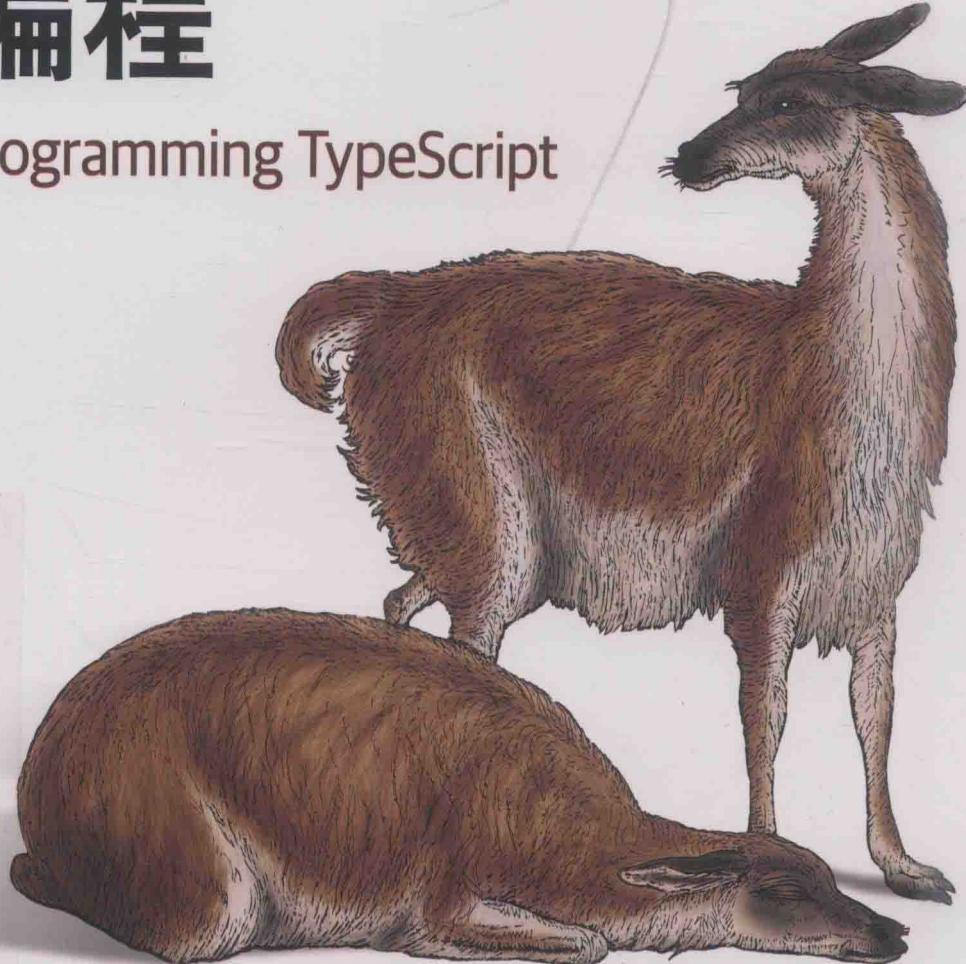


O'REILLY®

TypeScript

编程

Programming TypeScript



Boris Cherny 著

安道 译

TypeScript 编程

作者：Boris Cherny

我们编写本书的目的是帮助你学习 TypeScript。本书从基础到进阶，循序渐进地介绍了 TypeScript 的核心概念、语义和最佳实践。通过大量的示例代码，让你能够快速上手并掌握 TypeScript 的强大功能。

本书适合所有对编程感兴趣的读者，特别是那些希望提升自己的 JavaScript 技能或学习新的编程语言的开发者。无论你是初学者还是有一定经验的开发者，都能从本书中受益匪浅。

Boris Cherny 著

安道 译



Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

ia, Inc. 授权中国电力出版社出版



中国电力出版社
CHINA ELECTRIC POWER PRESS

Copyright © 2019 Boris Cherny. All rights reserved.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2020.
Authorized translation of the English edition, 2019 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2019。

简体中文版由中国电力出版社出版 2020。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

图书在版编目 (CIP) 数据

TypeScript 编程 / (美) 鲍里斯·切尔尼 (Boris Cherny) 著；安道译. — 北京：中国电力出版社，2020.6

书名原文：Programming TypeScript

ISBN 978-7-5198-4596-4

I. ①T… II. ①鲍… ②安… III. ①超文本标记语言－程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2020)第065507号

北京市版权局著作权合同登记 图字：01-2020-1281号

出版发行：中国电力出版社

地 址：北京市东城区北京站西街 19 号（邮政编码 100005）

网 址：<http://www.cepp.sgcc.com.cn>

责任编辑：刘炽 (liuchi1030@163.com)

责任校对：王小鹏

装帧设计：Karen Montgomery, 张健

责任印制：杨晓东

印 刷：北京天宇星印刷厂

版 次：2020 年 6 月第一版

印 次：2020 年 6 月北京第一次印刷

开 本：750 毫米 × 980 毫米 16 开本

印 张：22

字 数：417 千字

印 数：0001—3000 册

定 价：88.00 元

版 权 专 有 侵 权 必 究

本书如有印装质量问题，我社营销中心负责退换

O'Reilly Media, Inc.介绍

O'Reilly以“分享创新知识、改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来O'Reilly图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

献给 Sasha 和 Michael，希望有一天你们也会爱上类型。

目录

前言	1
第 1 章 导言	9
第 2 章 TypeScript 概述	13
2.1 编译器	13
2.2 类型系统	15
TypeScript VS. JavaScript	16
2.3 代码编辑器设置	20
2.3.1 tsconfig.json	20
2.3.2 tslint.json	22
2.4 index.ts	23
2.5 练习题	24
第 3 章 类型全解	26
3.1 类型术语	27
3.2 类型浅谈	28
3.2.1 any	28
3.2.2 unknown	30
3.2.3 boolean	30
3.2.4 number	32

3.2.5 bigint	33
3.2.6 string.....	34
3.2.7 symbol	34
3.2.8 对象	35
3.2.9 中场休息：类型别名、并集和交集.....	42
3.2.10 数组	46
3.2.11 元组	48
3.2.12 null、undefined、void 和 never.....	51
3.2.13 枚举	53
3.3 小结	58
3.4 练习题	58

第 4 章 函数 60

4.1 声明和调用函数	60
4.1.1 可选和默认的参数.....	62
4.1.2 剩余参数.....	64
4.1.3 call、apply 和 bind	65
4.1.4 注解 this 的类型	66
4.1.5 生成器函数.....	68
4.1.6 迭代器	70
4.1.7 调用签名.....	72
4.1.8 上下文类型推导	75
4.1.9 函数类型重载	76
4.2 多态	83
4.2.1 什么时候绑定泛型.....	88
4.2.2 可以在什么地方声明泛型	89
4.2.3 泛型推导.....	91
4.2.4 泛型别名.....	93
4.2.5 受限的多态	95
4.2.6 泛型默认类型	100
4.3 类型驱动开发	101

4.4 小结	102
4.5 练习题	103
第 5 章 类和接口.....	104
5.1 类和继承.....	104
5.2 super	110
5.3 以 this 为返回类型.....	111
5.4 接口	113
5.4.1 声明合并.....	115
5.4.2 实现	117
5.4.3 实现接口还是扩展抽象类	119
5.5 类是结构化类型	120
5.6 类既声明值也声明类型	121
5.7 多态	124
5.8 混入	125
5.9 装饰器	129
5.10 模拟 final 类.....	132
5.11 设计模式.....	133
5.11.1 工厂模式.....	133
5.11.2 建造者模式	134
5.12 小结	136
5.13 练习题	136
第 6 章 类型进阶.....	138
6.1 类型之间的关系	139
6.1.1 子类型和超类型	139
6.1.2 型变	141
6.1.3 可赋值性.....	148
6.1.4 类型拓宽.....	149
6.1.5 细化	154
6.2 全面性检查	159

6.3 对象类型进阶	161
6.3.1 对象类型的类型运算符	161
6.3.2 Record 类型	166
6.3.3 映射类型	167
6.3.4 伴生对象模式	170
6.4 函数类型进阶	171
6.4.1 改善元组的类型推导	171
6.4.2 用户定义的类型防护措施	172
6.5 条件类型	174
6.5.1 条件分配	175
6.5.2 infer 关键字	177
6.5.3 内置的条件类型	178
6.6 解决办法	179
6.6.1 类型断言	179
6.6.2 非空断言	180
6.6.3 明确赋值断言	183
6.7 模拟名义类型	184
6.8 安全地扩展原型	187
6.9 小结	189
6.10 练习题	190
第 7 章 处理错误	192
7.1 返回 null	193
7.2 抛出异常	194
7.3 返回异常	197
7.4 Option 类型	199
7.5 小结	206
7.6 练习题	207
第 8 章 异步编程、并发和并行	208
8.1 JavaScript 的事件循环	209
8.2 处理回调	211
8.3 promise：让一切回到正轨	214

8.4 async 和 await	219
8.5 异步流	220
事件发射器.....	221
8.6 多线程类型安全	224
8.6.1 在浏览器中：使用 Web 职程.....	224
8.6.2 在 NodeJS 中：使用子进程.....	234
8.7 小结	235
8.8 练习题	236
第 9 章 前后端框架	237
9.1 前端框架	237
9.1.1 React.....	239
9.1.2 Angular 6/7	246
9.2 类型安全的 API	250
9.3 后端框架.....	252
9.4 小结	253
第 10 章 命名空间和模块.....	254
10.1 JavaScript 模块简史	255
10.2 import、export	258
10.2.1 动态导入.....	259
10.2.2 使用 CommonJS 和 AMD 模块.....	262
10.2.3 模块模式与脚本模式	262
10.3 命名空间	263
10.3.1 冲突	265
10.3.2 编译输出.....	266
10.4 声明合并	268
10.5 小结	269
10.6 练习题	270
第 11 章 与 JavaScript 互操作	271
11.1 类型声明	272
11.1.1 外参变量声明	275

11.1.2 外参类型声明	277
11.1.3 外参模块声明	278
11.2 逐步从 JavaScript 迁移到 TypeScript	280
11.2.1 第一步：添加 TSC	280
11.2.2 第二步（上）：对 JavaScript 代码做类型检查（可选）	281
11.2.3 第二步（下）：添加 JSDoc 注解（可选）	283
11.2.4 第三步：把文件重命名为 .ts	284
11.2.5 第四步：严格要求	285
11.3 寻找 JavaScript 代码的类型信息	286
11.4 使用第三方 JavaScript	289
11.4.1 自带类型声明的 JavaScript	289
11.4.2 DefinitelyTyped 中有类型声明的 JavaScript	290
11.4.3 DefinitelyTyped 中没有类型声明的 JavaScript	290
11.5 小结	292
第 12 章 构建和运行 TypeScript	293
12.1 构建 TypeScript 项目	293
12.1.1 项目结构	293
12.1.2 构建产物	294
12.1.3 设置编译目标	295
12.1.4 生成源码映射	300
12.1.5 项目引用	300
12.1.6 监控错误	303
12.2 在服务器中运行 TypeScript	304
12.3 在浏览器中运行 TypeScript	304
12.4 把 TypeScript 代码发布到 NPM 中	307
12.5 三斜线指令	308
12.5.1 types 指令	309
12.5.2 amd-module 指令	310
12.6 小结	311
第 13 章 总结	313

附录 A 类型运算符	315
附录 B 实用类型	317
附录 C 限定作用范围的声明	319
附录 D 为第三方 JavaScript 模块编写声明文件的技巧	321
附录 E 三斜线指令	329
附录 F 安全相关的 TSC 编译器标志	331
附录 G TSX	333

随着 Python 的蓬勃发展，最近几年来有越来越多的开发者开始使用 Python 编程语言。尽管这并不是一个全新的趋势，但 Python 在一些领域（如数据科学、机器学习和人工智能）的应用确实已经非常广泛。本书将帮助你掌握 Python 的基础知识，让你能够更高效地完成工作，从而更好地应对未来的挑战。

无论你是刚刚接触 Python 的初学者，还是有一定经验的开发者，看了这本书后，相信你会对 Python 有更深的理解。书中众多的例子和详细的注释，可以帮助你更好地理解 Python 的语法和语义，从而提高你的编程水平。

如果你是已经有了一定编程经验的开发者，或者希望进一步提升自己的技能，那么这本书也是个不错的选择。书中对各种高级主题的深入探讨，以及对常见错误的详细分析，可以帮助你更好地理解 Python 的内部机制，从而写出更高效的代码。

与其他类型的书籍相比，这本书的特点在于它非常实用。通过具体的案例和

前言

本书适合各类程序员阅读。有一部分建议你采用，还有另一部分

本书适合各类程序员阅读，例如 JavaScript 专业工程师、C# 从业者、Java 拥护者、Python 爱好者、Ruby 偏爱者和 Haskell 支持者。不管你使用什么编程语言，只要有一定的编程经验，了解函数、变量、类和错误等基础知识，就可以阅读这本书。如果你使用过 JavaScript，有文档对象模型（Document Object Model, DOM）和网络编程经验，那就更好了。本书虽然没有深入探讨这些概念，但是将从这些方面举例。如果你不熟悉这些概念，可能无法深入领会示例的意图。

无论使用哪一门编程语言，我们都有共同的经历。为了追查异常，我们一行一行分析代码，找出问题所在，各个击破。而 TypeScript 能助我们一臂之力，它会自动检查代码，指出那些逃过我们眼睛的错误。

如果你没使用过静态类型语言也没关系。笔者将教你类型的知识，告诉你如何使用类型减少程序崩溃的可能、提升代码的语义，便于多位工程师共同维护，也让应用能惠及更多的用户，能在多台服务器上弹性伸缩。在行文上，笔者将力求浅显易懂，以直观、易记的方式讲解相关概念，从实用角度出发，通过大量示例把抽象的问题讲清楚。

与其他类型语言相比，TypeScript 的特点是非常注重实用。TypeScript 发明了

一套全新的概念，保证代码简洁、准确，使编写应用的过程充满乐趣，更符合现代标准，也更安全。

内容结构

本书有两个目的：一是深入讲解 TypeScript 语言的原理（理论层面）；二是给出大量实用的建议，助你写出更好的 TypeScript 代码（应用层面）。

前面说过，TypeScript 是一门注重实用的语言，理论与应用往往是联系在一起的。本书多数篇幅将穿插讲解这两方面，不过前几章基本只讲理论，后几章则几乎只说具体应用。

本书首先介绍编译器、类型检查器和类型的基础知识。然后，分别说明 TypeScript 中不同类型和类型运算符的作用和用法。掌握这些基础之后，可以深入探讨一些高级话题，比如 TypeScript 最为复杂的类型系统特性、错误处理和异步编程。最后，说明怎样结合你最喜欢的框架（前端和后端）使用 TypeScript，如何把现有的 JavaScript 项目迁移到 TypeScript，以及如何在生产环境下运行 TypeScript 应用。

本书每章的末尾都有练习题，请你尝试自己解答，这样才能更深入地领会所讲的内容。练习题的参考答案在网上，地址为 <https://github.com/bcherny/programming-typescript-answers>。

代码风格

本书尽量一以贯之，使用同一种代码风格。笔者采用的代码风格有一部分带有强烈的个人风格，例如：

- 只在必要时使用分号。
- 使用两个空格缩进。

- 在简单的代码片段中，或者程序的结构比细节重要时，使用简短的名称命名变量，例如 `a`、`f` 或 `_`。

本书使用的编程风格，有一部分也建议你采用。比如说：

- 应该使用最新的 JavaScript 句法和特性（最新版 JavaScript 通常称为“esnext”）。这样能让代码符合最新的标准，提升代码的互操作性，便于搜索，也能减少新员工的前期投入时间。此外，还可以充分利用 JavaScript 的新特性，例如箭头函数、promise 和生成器。
- 尽量使用展开运算符（...），保持数据结构不可变。^{注1}
- 所有值都要有类型，不过尽量推导而出。切记不要滥用显式类型，让类型错误暴露出来，从而保证代码简洁、增加安全。
- 保证代码的可用性和普适性。多态（见 4.2 节）是个有力的工具。

当然，这些思想都是全新的。不过，沿用这些风格对 TypeScript 的正常运作有至关重要的作用。TypeScript 内置的下层编译器支持只读类型，有强大的类型推导功能，深置对多态的支持，而且具有完整的结构化类型系统，这些都促使我们使用良好的编程风格。与底层的 JavaScript 相比，TypeScript 在语言层面上仍不失表现力和真实性。

在进入正文之前，还有几点要说明。

JavaScript 没有指针和引用的概念，有的只是值和引用类型。值是不可变的，包括字符串、数字和布尔值；而引用通常指向可变的数据结构，例如数组、对象和函数。本书中出现的“值”，一般不使用它的严格定义，而是指 JavaScript 值或引用。

联系我们



注 1：以防你没接触过 JavaScript，举个例子：假如有个对象 `o`，我们想为该对象添加一个值为 3 的属性 `k`；为此，可以直接修改 `o`，使用 `o.k = 3` 句法，也可以使用 `let p = {...o, k: 3}` 句法，新建一个对象。

最后，在与 JavaScript 互操作时，使用未严格遵守类型规定的第三方库时，维护旧代码或匆匆上手时，很容易写出不太理想的 TypeScript 代码。本书的主要目的是教你如何正确编写 TypeScript，告诫你一定要遵守规则。但在实际中，怎么编写代码才算正确要看你自己或你的团队。

本书约定

本书采用下述排版约定。

斜体 (*Italic*)

表示新术语、URL、电子邮件地址、文件名和扩展名。

等宽字体 (Constant Width)

表示程序清单，在段落中出现则表示程序元素，例如变量、函数名、数据类型、环境变量、语句和关键字。

斜体等宽字体 (*Constant Width Italic*)

表示应该替换成用户提供的值，或者由上下文决定的值。



表示提示或建议。



表示一般性说明。



表示警告。

使用代码示例

本书的补充材料（代码示例、练习题等）可到 <https://github.com/bcherny/programming-typescript-answers> 下载。

本书是帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如：“Programming TypeScript by Boris Cherny (O'Reilly). Copyright 2019 Boris Cherny, 978-1-492-03765-1.”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，请通过 permissions@oreilly.com 与我们联系。

O'Reilly Online Learning

O'REILLY[®] 40 年间，O'Reilly Media 为众多公司提供技术和商业培训，提升知识储备和洞察力，为企业的成功助力。

我们有一群独家专家和创新者，他们通过图书、文章、会议和在线学习平台分享知识和技术。O'Reilly 的在线学习平台提供按需访问的直播培训课程、详细的学习路径、交互式编程环境，以及由 O'Reilly 和其他 200 多家出版社出版的书籍和视频。详情请访问 <http://oreilly.com>。

联系我们

任何有关本书的意见或疑问，请按照以下地址联系出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

勘误、示例和其他信息可到 <https://oreil.ly/programming-typescript> 上获取。

对本书的评论或技术疑问，可以发电子邮件到 bookquestions@oreilly.com。

欲了解本社图书、课程、会议和新闻等更多信息，请访问我们的网站 <http://www.oreilly.com>。

我们的 Facebook：<http://facebook.com/oreilly>。

我们的 Twitter：<http://twitter.com/oreillymedia>。

我们的 YouTube：<http://www.youtube.com/oreillymedia>。

致谢

几年间断断续续的写作和涂涂画画，加上一年的早起晚睡，以及周末和节假日的奋笔疾书，这本书才得以出版。

感谢 O'Reilly 策划这样一本书，感谢编辑 Angela Rufino 在整个过程中给予我的支持。感谢 Nick Nance 对 9.2 节的贡献，感谢 Shyam Seshadri 对 9.1.2 节的贡献。感谢技术编辑 Daniel Rosenwasser，他是 TypeScript 团队的一员，用了很多时间阅读草稿，指导我弄清了 TypeScript 类型系统的方方面面。感谢 Jonathan Creamer、Yakov Fain、Paul Buying 和 Rachel Head 对本书做技术编辑、

提供反馈。感谢我的家人 Liza、Ilya、Vadim、Roza、Alik、Faina 和 Yosif，感谢他们鼓励才能使我坚持完成这个项目。

感谢我的伴侣 Sara Gilford，在撰写本书的过程中她始终支持我，即使取消了周末计划、写作和编程到深夜，她也没有怨言，而且主动与我讨论类型系统的复杂细节。如果没有你，我不可能写完这本书，你的支持让我铭记终生。

导言

不要闹别扭了，应该说 TypeScript 是程序员们面对的另一个全新的世界。基础的东西也有了本质的提升（向上）。但是对 TypeScript 而言，最大的一个转折点就是书中好几处的产生与发展的章节了。这些章节证明这是些很人性化的书。

奇怪，为什么要读一本讲 TypeScript 的书呢？

也许是你受够了奇怪的 cannot read property blah of undefined JavaScript 错误。也许是听说 TypeScript 能让代码更好地弹性伸缩，想一探究竟。也许是 C# 从业人员，想尝试一下 JavaScript。也许你喜欢函数式编程，决定向前迈进一步。也许是你的老板受够了错误百出的生产代码，把本书作为圣诞节礼物送给你（很感动吧）。

不管出于什么原因，你没有听错。TypeScript 将成为下一代 Web 应用、移动应用、NodeJS 项目和物联网（Internet of Things, IoT）设备的主力开发语言。使用 TypeScript 开发的程序更安全，常见的错误都能检查出来，写出的代码还可以作为文档，供自己和以后的工程师使用。重构也不再是痛苦的过程，而且还能省去一半的单元测试（“什么单元测试？”）。TypeScript 能让程序员事半功倍，节省的时间说不定还能与街对面那个可爱的咖啡师来场约会呢。

但是，先别急着跑到街对面，首先我们要明确一个问题：“更安全”到底是什么意思？这里的安全当然是指“类型安全”。

类型安全

借助类型避免程序做无效的事情。^{注1}

下面举几个无效事情的例子：

- 一个数字乘以一个列表。
- 调用接受一组对象的函数却传入一组字符串。
- 在对象上调用不存在的方法。
- 导入已经被移除的模块。

这些问题在一些编程语言中时有发生。编程语言发现你在做无效的事情时会尝试判断你的真正意图，以 JavaScript 为例：

```
3 + []           // 求值结果为字符串 "3"  
  
let obj = {}  
obj.foo         // 求值结果为 undefined  
  
function a(b) {  
    return b/2  
}  
a("z")          // 求值结果为 NaN
```

注意，遇到显而易见的无效操作时，JavaScript 没有抛出异常，而是尽自己所能，避免抛出异常。JavaScript 这么做有意义吗？当然有。但是这样做便于快速捕获问题吗？或许不能。

现在试想，如果 JavaScript 不在背后静悄悄地做这么多工作，而是直接抛出异常，情况又如何呢？我们可能会得到如下的反馈：

注1：在不同的静态类型语言中，“无效”的所指有所不同，可以指运行时程序崩溃，也可以指未崩溃，但是做的事情却毫无意义。

```
3 + []          // Error: Did you really mean to add a number and an array?  
  
let obj = {}  
obj.foo        // Error: You forgot to define the property "foo" on obj.  
  
function a(b) {  
    return b/2  
}  
a("z")         // Error: The function "a" expects a number,  
               // but you gave it a string.
```

不要理解错了，尝试修正错误是编程语言一个很好的特性（要是程序之外的其他东西也有这个特性就好了！）。但是对 JavaScript 来说，这个特性让代码中错误的产生与发现脱节了。这往往导致错误是由他人转告给你的。

那么问题来了，JavaScript 到底什么时候才会告诉你代码有问题呢？

答案是在你真正运行程序时，可能是在浏览器中试运行时或者是用户访问网站时，也可能是运行单元测试时。对一个训练有素的程序员来说，可能编写了大量单元测试和端对端测试，在推送之前做了冒烟测试，在最终发布之前还在内部试运行了一段时间，那么可能会先于用户发现错误。但是，实际往往不是如此。

这时 TypeScript 的作用就突显出来了。更好的一点是 TypeScript 给出错误消息的时间点：在输入代码的过程中，文本编辑器会给出错误消息。如此一来，我们便不用依靠单元测试或冒烟测试，抑或同事来捕获这类问题，在编写程序的过程中 TypeScript 就能捕获这些问题，然后提醒你。下面来看 TypeScript 对前述示例的反馈：

```
3 + []          // Error TS2365: Operator '+' cannot be applied to types '3'  
               // and 'never[]'.  
  
let obj = {}  
obj.foo        // Error TS2339: Property 'foo' does not exist on type '{}'.  
  
function a(b: number) {  
    return b / 2  
}
```

```
a("z") // Error TS2345: Argument of type '"z"' is not assignable to  
// parameter of type 'number'.
```

除了消除与类型有关的一整类问题之外，TypeScript 还彻底改变了编写代码的方式。现在，我们先在类型层面规划整个程序，然后再深入到值层面；^{注2} 在设计程序的过程中就考虑边缘情况，而不等到事后再补救。这样设计出来的程序更简单，运行速度更快，而且更容易理解和维护。

准备好开始本次旅程了吗？出发！

（这部电影是关于美国公司智商 Iqlevel，它来帮助那些经常犯错误、做不出决策、无法完成任务的员工。如果员工觉得正在做无趣的事情时会自动地从工作上移开或对老板说不清楚是什么，他们就会感到真得感谢管理者并感谢自己，因为自己能够接收到一个快照。快照显示单击或滑动一下，就能马上得到答案。）

function iq() {
 // 可以使用 keyof T 作为参数类型。T 是由函数参数中的 iqlevel 所推断出的。
 // 例如，如果将上面的签名写成本文一开始提到的提升函数形式，类型推断将
 // 为 void，但通过使用 keyof T，TypeScript 将推断出参数类型为
 // keyof iqlevel，即字符串。这样，如果调用 iq('z')，TypeScript 将会报错。
}

注 2：如果你不理解“类型层面”的意思，不用担心，后续章节将深入探讨。

TypeScript 概述

接下来的几章介绍 TypeScript 语言，概述 TypeScript 编译器（TSC）的工作原理，介绍开发中能用到的 TypeScript 特性和模式。先从编译器讲起。

2.1 编译器

你对程序运行原理的理解，取决于你过去（购买本书准备进入类型安全世界之前）使用的是哪一门编程语言。与 JavaScript 或 Java 等主流语言相比，TypeScript 的运作方式显得与众不同。因此，在深入探讨之前，有必要弄清这一点。

概括地说，程序由一些文件构成，文件中是你（即程序员）编写的文本。这些文本由一个特殊的程序（称为编译器）解析，转换成抽象句法树（abstract syntax tree, AST）。AST 是去掉了空白、注释和缩进用的制表符或空格之后的数据结构。编译器把 AST 转换成一种称为字节码（bytecode）的低层表示。字节码再传给运行时程序计算，得到最终结果。也就是说，运行程序就是让运行时计算由编译器从源码解析得来的 AST 生成的字节码。不同语言的具体细节有所不同，但是整体大致如此。

综上，步骤如下：

1. 把程序解析为 AST。

2. 把 AST 编译成字节码。

3. 运行时计算字节码。

TypeScript 的特殊之处在于，它不直接编译成字节码，而是编译成 JavaScript 代码。然后再像往常一样，在浏览器中，或者使用 NodeJS，抑或使用纸和笔（机器起义之后的岁月）运行得到的 JavaScript 代码。

现在你可能思索着，“且慢，前一章说 TypeScript 能让代码变得更安全，它是怎么做到的呢？”

好问题。其实，笔者跳过了一个至关重要的步骤：TypeScript 编译器生成 AST 之后，真正运行代码之前，TypeScript 会对代码做类型检查。

类型检查器

检查代码是否符合类型安全要求的特殊程序。

类型检查是 TypeScript 的魔力所在。有了这一步，TypeScript 才能保证程序能按预期正常运行，没有显而易见的错误，才能保证街对面那位可爱的咖啡师会信守承诺，再给你打电话（别心急，说不定她们现在正忙）。

那么，如果加入类型检查和运行 JavaScript，TypeScript 编译的过程大致变成图 2-1 中所示的那样。

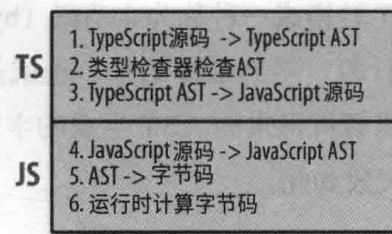


图 2-1：编译和运行 TypeScript

第 1~3 步由 TSC 操作，第 4~6 步由浏览器、NodeJS 或其他 JavaScript 引擎中的 JavaScript 运行时操作。



JavaScript 编译器和运行时通常聚在一个称为引擎的程序中。程序员一般就是与引擎交互的。V8（驱动 NodeJS、Chrome 和 Opera 的引擎）、SpiderMonkey（Firefox）、JSCore（Safari）和 Chakra（Edge）都是如此。正是因为这样，JavaScript 看起来才像是一门解释型语言。

在这个过程中，第 1~2 步使用程序的类型，第 3 步不使用。重申一下，TSC 把 TypeScript 编译成 JavaScript 时，不会考虑类型。这意味着，程序中的类型对程序生成的输出没有任何影响，类型只在类型检查这一步使用。这个特性确保我们可以随意改动、更新和改进程序中的类型，而无需担心会破坏应用的功能。

2.2 类型系统

现代语言采用的类型系统不尽相同。

类型系统

类型检查器为程序分配类型时使用的一系列规则。

一般来说，类型系统有两种：一种通过显式句法告诉编译器所有值的类型，另一种自动推导值的类型。这两种类型系统各有利弊。^{注 1}

注 1：不同语言采用不同的类型系统，JavaScript、Python 和 Ruby 在运行时推导类型；Haskell 和 OCaml 在编译时推导和检查类型；Scala 和 TypeScript 要求显式声明部分类型，然后在编译时推导和检查余下的部分；Java 和 C 几乎需要显式注解所有类型，然后在编译时检查。

TypeScript 身兼两种类型系统，可以显式注解类型，也可以让 TypeScript 推导多数类型。

为了显式告知 TypeScript 你使用的是什么类型，需要使用注解。注解的形式为 *value: type*，就像是告诉类型检查器，“嘿，看到这个 *value* 了吗？它的类型为 *type*。”来看几个示例（每一行后面的注释是 TypeScript 推导出的类型）：

```
let a: number = 1           // a 是一个数字
let b: string = 'hello'     // b 是一个字符串
let c: boolean[] = [true, false] // c 是一个布尔值数组
```

如果想让 TypeScript 推导类型，那就去掉注解，让 TypeScript 自动推导：

```
let a = 1                   // a 是一个数字
let b = 'hello'             // b 是一个字符串
let c = [true, false]        // c 是一个布尔值数组
```

TypeScript 推导类型的好处立刻显现。去掉注解后，类型并没有变。本书只在必要时使用注解，多数情况下都把工作交给 TypeScript 的推导功能。



一般来说，最好让 TypeScript 推导类型，少数情况下才显式注解类型。

TypeScript VS. JavaScript

本节深入探讨 TypeScript 的类型系统，并与 JavaScript 的类型系统比较。两门语言之间的差别概括见表 2-1。掌握二者的差别才能更好地理解 TypeScript 的运作方式。

表 2-1：比较 JavaScript 和 TypeScript 的类型系统

类型系统特性	JavaScript	TypeScript
类型是如何绑定的？	动态	静态
是否自动转换类型？	是	否（多数时候）

表 2-1：比较 JavaScript 和 TypeScript 的类型系统（续）

类型系统特性	JavaScript	TypeScript
何时检查类型？	运行时	编译时
何时报告错误？	运行时（多数时候）	编译时（多数时候）

类型是如何绑定的？

JavaScript 动态绑定类型，因此必须运行程序才能知道类型。在运行程序之前，JavaScript 对类型一无所知。

TypeScript 是渐进式类型语言。这意味着，在编译时知道所有类型能让 TypeScript 充分发挥作用，但在编译程序之前，并不需要知道全部类型。即便是没有类型的程序，TypeScript 也能推导出部分类型，捕获部分错误，但这并不全面，大量错误可能会暴露给用户。

渐进式类型特别便于把没有类型的 JavaScript 代码基迁移到具有类型的 TypeScript（详见 11.2 节）。然而，除非你处在迁移代码基的过程之中，否则应该实现 100% 的类型覆盖率。如无特殊说明，本书都采用这种策略。

是否自动转换类型？

JavaScript 是弱类型语言，如果执行无效的操作，例如计算一个数与一个数组的和（像第 1 章做过的那样），JavaScript 将根据一系列规则判断你的真正意图。下面以 JavaScript 表达式 `3 + [1]` 为例说明具体步骤：

1. JavaScript 发现 `3` 是一个数字，`[1]` 是一个数组。
2. 由于使用的是 `+`，JavaScript 假定你是想拼接二者。
3. JavaScript 把 `3` 隐式转换为字符串，得到 `"3"`。
4. JavaScript 把 `[1]` 隐式转换为字符串，得到 `"1"`。
5. 把两个字符串拼接在一起，得到 `"31"`。

我们可以更明确地表明意图（让 JavaScript 跳过第 1 步、第 3 步和第 4 步）：

```
3 + [1]; // 求值结果为 "31"
```

```
(3).toString() + [1].toString() // 求值结果为 "31"
```

JavaScript 会自作聪明，自动转换类型，而 TypeScript 发现无效的操作时则报错。使用 TSC 运行这段 JavaScript 代码，将得到如下错误：

```
3 + [1]; // Error TS2365: Operator '+' cannot be applied to  
// types '3' and 'number[]'.
```

```
(3).toString() + [1].toString() // 求值结果为 "31"
```

如果执行可能不正确的操作，TypeScript 会报错。然而，如果明确表明意图，TypeScript 则不会做出阻拦。其实这个行为是符合常理的，试想，谁会去计算一个数字与一个数组的和，而且预期结果为一个字符串（除非你雇用一个 JavaScript 女巫，在创业公司的地下室点着蜡烛编程）？

JavaScript 这种隐式转换可能导致难以追踪的错误，令很多 JavaScript 程序员深恶痛绝。这会导致工程师个人无法顺利完成工作，也会导致大型团队的合作混乱，毕竟让每个工程师都了解代码中的隐式约定是一件很难的事。

简言之，如果必须转换类型，请明确表明你的意图。

何时检查类型？

多数情况下，JavaScript 不在乎你使用的是什么类型，它会尽自己所能把你提供的值转换成预期的类型。

而 TypeScript 会在编译时对代码做类型检查（还记得本章开头的第 2 步吗？），因此不用运行代码就能看到前例中的 Error。TypeScript 会对代码做静态分析，找出这类错误，在运行之前反馈给你。如果代码不能编译，很有可能就表明代码中有错误，在运行之前要修正。

笔者在 VSCode（笔者选择使用的代码编辑器）中输入上述示例看到的错误如图 2-2 所示。

```
1 3 + [1]
2 [ts] Operator '+' cannot be applied to types
3 '3' and 'number[]'. [2365]
4
```

图 2-2：VSCode 报告的 TypeError

为代码编辑器安装恰当的 TypeScript 扩展，在输入代码的过程中如果有错误，出错位置的下方将出现一条红色波浪线。这极大地缩短了反馈循环，在编写代码的过程中便能发现错误，然后立即修正。

何时报告错误？

JavaScript 在运行时抛出异常或执行隐式类型转换。^{注2} 这意味着，必须真正运行程序才能知道有些操作是无效的。最好的情况是，单元测试发现了错误；最坏的情况是，用户会给你发一封气冲冲的电子邮件。

TypeScript 在编译时报告句法和类型相关的错误。实际上，这些错误会在代码编辑器中显示，输入代码后立即就有反馈。对没有使用过渐进编译式静态类型语言的人来说，这是一个非常棒的体验。^{注3}

尽管如此，还有大量错误是 TypeScript 在编译时无法捕获的，例如堆栈溢出、网络断连和恶意的用户输入，这些属于运行时异常。TypeScript 所能做的是把纯 JavaScript 代码中那些运行时错误提前到编译时报告。

注 2： 其实，JavaScript 在解析程序之后、运行程序之前报告句法错误和部分缺陷（例如在同一个作用域中多次使用 `const` 声明同名常量）。如果是在构建过程中解析 JavaScript（例如使用 Babel），错误则在构建时报告。

注 3： 渐进编译式语言在小修小改后可以快速重新编译，而不用重新编译整个程序（包括那些没有改动的部分）。

2.3 代码编辑器设置

现在，我们对 TypeScript 编译器和类型系统有了一定的认识。接下来要设置代码编辑器，开始分析具体的代码。

首先下载一个代码编辑器，用于编写代码。笔者喜欢 VSCode，因为它提供的 TypeScript 编辑体验特别好。你也可以使用 Sublime Text、Atom、Vim、WebStorm 或其他你喜欢的编辑器。工程师对 IDE 特别挑剔，所以决定权留给你自己。如果你想使用 VSCode，请按照官网上的说明设置 (<https://code.visualstudio.com>)。

TSC 是一个使用 TypeScript 编写的命令行应用，^{注4} 需要通过 NodeJS 运行。在设备中安装 NodeJS 的说明参见官网 (<https://nodejs.org>)。

NodeJS 附带包管理器 NPM，用于管理项目的依赖和编排构建。我们将使用 NPM 安装 TSC 和 TSLint（一个 TypeScript linter）。打开终端，新建一个文件夹，在该文件夹中初始化一个新的 NPM 项目：

```
# 新建一个文件夹  
mkdir chapter-2  
cd chapter-2  
  
# 初始化一个新的 NPM 项目（根据提示操作）  
npm init  
  
# 安装 TSC、TSLint 和 NodeJS 的类型声明  
npm install --save-dev typescript tslint @types/node
```

2.3.1 tsconfig.json

每个 TypeScript 项目都应该在根目录中放一个名为 *tsconfig.json* 的文件，在该文件中定义要编译哪些文件、把文件编译到哪个目录中，以及使用哪个版本的 JavaScript 运行。

注4：如此得来的 TSC 显得特别神秘，我们称之为自托管编译器或自编译型编译器。

在根目录中新建一个名为 *tsconfig.json* 的文件（touch *tsconfig.json*），^{注5} 然后在代码编辑器中打开，写入下述内容：

```
{  
  "compilerOptions": {  
    "lib": ["es2015"],  
    "module": "commonjs",  
    "outDir": "dist",  
    "sourceMap": true,  
    "strict": true,  
    "target": "es2015"  
},  
  "include": [  
    "src"  
]  
}
```

下面简要介绍一下部分选项的作用（见表 2-2）。

表 2-2: *tsconfig.json* 选项

选项	说明
include	TSC 在哪个文件夹中寻找 TypeScript 文件
lib	TSC 假定运行代码的环境中有哪些 API？包括 ES5 的 Function.prototype.bind、ES2015 的 Object.assign 和 DOM 的 document.querySelector
module	TSC 把代码编译成哪个模块系统（CommonJS、SystemJS、ES2015 等）
outDir	TSC 把生成的 JavaScript 代码放在哪个文件夹中
strict	检查无效代码时尽量严格。该选项强制所有代码都正确声明了类型。本书中的所有示例都使用这个选项，你的 TypeScript 项目也应该这么做
target	TSC 把代码编译成哪个 JavaScript 版本（ES3、ES5、ES2015、ES2016 等）

这只是其中部分选项，*tsconfig.json* 支持的选项还有很多，而且一直有新的选项出现。实际上，这些选项很少改动，偶尔需要改动的是，切换模块打包工具时修改 *module* 和 *target* 设置，编写在浏览器中运行的 TypeScript 时在

注5： 这里我们是自己动手创建 *tsconfig.json* 文件的。以后设置 TypeScript 项目时，可以使用 TSC 内置的初始化命令生成该文件：`./node_modules/.bin/tsc --init`。

`lib` 中添加 “`dom`”（第 12 章详细说明），或者把现有 JavaScript 代码迁移到 TypeScript 时调整严格（strict）级别（见 11.2 节）。完整且最新的选项列表参阅 TypeScript 网站中的官方文档 (<http://bit.ly/2JWfsgY>)。

注意，使用 `tsconfig.json` 文件配置 TSC 是很方便的，因为我们可以把配置纳入源码控制。除此之外，TSC 的很多选项还可以通过命令行设置。可用的命令行选项请运行 `./node_modules/.bin/tsc --help` 命令查看。

2.3.2 tslint.json

项目中还应该有个 `tslint.json` 文件，保存 TSLint 配置，为代码制订风格上的约定（使用制表符还是空格等）。



TSLint 不是必用的工具，不过强烈建议所有 TypeScript 项目都使用，从而保证编程风格一致。最重要的是，在审阅代码时，无须再浪费口舌与同事争论哪种代码风格更好。

下述命令将生成一个含有 TSLint 默认配置的 `tslint.json` 文件：

```
./node_modules/.bin/tslint --init
```

然后，可以覆盖某些选项，满足自己对编程风格的要求。例如，我的 `tslint.json` 文件是这样的：

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

完整的可用规则列表参见 TSLint 文档 (<https://palantir.github.io/tslint/rules/>)。此外，还可以添加自定义的规则，或者安装额外的预设（例如针对 ReactJS 的预设，<https://www.npmjs.com/package/tslint-react>）。

2.4 index.ts

设置好 `tsconfig.json` 和 `tslint.json` 之后，新建 `src` 文件夹，我们的第一个 TypeScript 文件就保存在这里：

```
mkdir src  
touch src/index.ts
```

现在，项目的文件夹结构如下所示：

```
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`，输入下述 TypeScript 代码：

```
1 console.log('Hello TypeScript!')
```

然后，编译并运行 TypeScript 代码：

```
# 使用 TSC 编译 TypeScript  
./node_modules/.bin/tsc
```

```
# 使用 NodeJS 运行代码  
node ./dist/index.js
```

如果你严格按照前述步骤操作，代码能顺利运行，在控制台中将看到一条日志：

```
Hello TypeScript!
```

恭喜，你从零开始设置并运行了第一个 TypeScript 项目。做得不错！



这是你第一次从零开始创建 TypeScript 项目，笔者详细说明了每一步，以便你对项目整体有个全面的认识。下一次创建 TypeScript 项目时，可以采取一些捷径：

- 安装 `ts-node` (<https://npmjs.org/package/ts-node>)，这样只需一个命令便能编译和运行 TypeScript 代码。
- 使用 `typescript-node-starter` (<https://github.com/Microsoft/TypeScript-Node-Starter>) 之类的脚手架工具，快速生成文件夹结构。

2.5 练习题

环境已经搭建好，在代码编辑器中打开 `src/index.ts`，输入下述代码：

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

把鼠标悬停在 `a`、`b`、`c` 和 `d` 上，注意 TypeScript 推导出的变量类型：`a` 是一个数字，`b` 是一个数字，`c` 是一个具有特定结构的对象，`d` 也是一个数字（见图 2-3）。

```
6 let a = 1 + 2
7 let b = a + 3
8 let c = {
9   apple: a,
10  banana: b
11 } let d: number
12 let d = c.apple * 4
```

图 2-3：TypeScript 自动推导类型

自己试验一下，看你能不能做到：

- 在执行无效的操作时让 TypeScript 显示一条红色的波浪线（这叫“抛出 `TypeError`”）。
 - 阅读 `TypeError`，尝试理解其内容。
 - 修正 `TypeError`，让红色波浪线消失。

如果你想挑战一下自己，可以试着编写一些 TypeScript 无法推导出类型的代码。

类型全解

前一章介绍了类型系统，但是并未定义类型系统中的“类型”到底是什么意思。

类型

一系列值及可以对其执行的操作。

如果你不明其意，下面举几个你熟悉的例子：

- `boolean` 类型包含所有布尔值（只有两个：`true` 和 `false`），以及可以对布尔值执行的操作（例如`||`、`&&` 和`!`）。
- `number` 类型包括所有数字，以及可以对数字执行的操作（例如`+`、`-`、`*`、`/`、`%`、`||`、`&&` 和`?`），还有可以在数字上调用的方法，例如`.toFixed`、`.toPrecision`、`.toString` 等。
- `string` 类型包括所有字符串，以及可以对字符串执行的操作（例如`+`、`||` 和`&&`），还有可以在字符串上调用的方法，例如`.concat` 和`.toUpperCase`。

对`T`类型的值来说，我们不仅知道值的类型是`T`，还知道可以（及不可以）对该值做什么操作。记住，最终是由类型检查器去阻止你做无效的操作。而类型检查器则通过使用的类型和具体用法判断操作是否有效。

本章概览 TypeScript 中可用的类型，介绍每一种类型的基本用法。TypeScript 中类型的结构如图 3-1 所示。

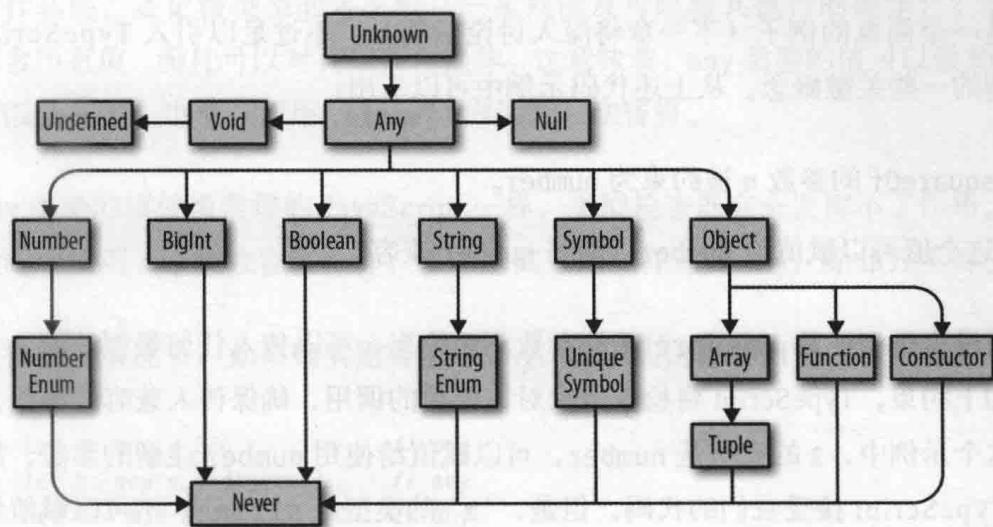


图 3-1：TypeScript 的类型层次结构

3.1 类型术语

程序员讨论类型时，有一套精确而通用的术语。本书贯穿始终都采用这套术语。

假设有个函数，接受某个值，返回该值与自己的乘积：

```
function squareOf(n) {
    return n * n
}
squareOf(2)      // 求值结果为 4
squareOf('z')    // 求值结果为 NaN
```

显然，这个函数只能操作数字。如果把数字之外的值传给 `squareOf` 函数，结果是无效的。下面显式注解参数的类型：

```
function squareOf(n: number) {
    return n * n
}
squareOf(2)      // 求值结果为 4
```

```
squareOf('z') // Error TS2345: Argument of type '"z"' is not assignable to  
// parameter of type 'number'.
```

现在，调用 `squareOf` 时如果传入数字之外的值，TypeScript 将立即报错。这只是一个简单的例子（下一章将深入讨论函数），不过足以引入 TypeScript 类型的一些关键概念。从上述代码示例中可以看出：

1. `squareOf` 的参数 `n` 被约束为 `number`。
2. 这个值可以赋值给 `number`（即与 `number` 兼容）。

如果没有类型注解，`squareOf` 的参数不受约束，可以传入任何类型的值。一旦加上约束，TypeScript 将检查每次对该函数的调用，确保传入兼容的参数。在这个示例中，`2` 的类型是 `number`，可以赋值给使用 `number` 注解的参数，因此 TypeScript 接受我们的代码。但是，'`z`' 的类型是 `string`，不可以赋值给 `number` 类型的参数，所以 TypeScript 报错。

另外，也可以把类型注解理解为某种界限。在这个示例中，我们告诉 TypeScript，`n` 的上限是 `number`，因此传给 `squareOf` 的值至多是一个数字。如果传入超过数字的值（比如说数组或字符串），那就不能赋值给 `n`。

可赋值性、界限和约束在第 6 章正式定义。现在你只须知道，讨论是否可以在某个地方使用某个类型时，我们将使用这一套术语。

3.2 类型浅谈

本节逐一介绍 TypeScript 支持的类型、各类型包含的值，以及可以对类型执行的操作。此外，本节还涵盖一些与类型有关的基本语言特性：类型别名、并集类型和交集类型。

3.2.1 any

`any` 是类型的教父。为达目的，它不惜一切代价，但是不要轻易请它出面，除非迫不得已。在 TypeScript 中，编译时一切都是有类型的，如果你（程序员）

和 TypeScript（类型检查器）无法确定类型是什么，默认为 `any`。这是兜底类型，应该尽量避免使用。

为什么呢？还记得类型的定义吗（一系列值及可以对其执行的操作）？`any` 包含所有值，而且可以对其做任何操作。这意味着，`any` 类型的值可以做加法、可以做乘法，也可以调用 `.pizza()` 方法，一切皆可。

`any` 类型的值就像常规的 JavaScript 一样，类型检查器完全发挥不了作用。在代码中使用 `any` 就像盲飞一样。要像远离火源一样远离 `any`，除非万不得已。

在极少数情况下，如果确实需要使用 `any`，像下面这样使用：

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

注意，正常情况下第三个语句将报错（谁会计算一个数字和一个数组的和呢？），但是却没有，因为我们告诉 TypeScript，相加的两个值都是 `any` 类型。如果想使用 `any`，一定要显式注解。倘若 TypeScript 推导出值的类型为 `any`（例如忘记注解函数的参数，或者引入没有类型的 JavaScript 模块），将抛出运行时异常，在编辑器中显示一条红色波浪线。显式把 `a` 和 `b` 的类型注解为 `any` (`: any`)，TypeScript 便不会抛出异常，因为这样做就是告诉 TypeScript，你知道自己在做什么。



TSC 标志：noImplicitAny

默认情况下，TypeScript 很宽容，在推导出类型为 `any` 时不会报错。如果想让 TypeScript 在遇到隐式 `any` 类型时报错，请在 `tsconfig.json` 中启用 `noImplicitAny` 标志。

`noImplicitAny` 隶属于 TSC 的 `strict` 标志家族，如果已经在 `tsconfig.json` 中启用了 `strict`（2.3.1 节就是这么做的），那就不用专门设置该标志了。

3.2.2 unknown

如果 `any` 是教父，那么 `unknown` 就是“惊爆点”电影中基努·里维斯饰演的 FBI 卧底 Johnny Utah，他吊儿郎当，与坏人同流合污，但是内心却尊重法律，站在好人这一边。少数情况下，如果你确实无法预知一个值的类型，不要使用 `any`，应该使用 `unknown`。与 `any` 类似，`unknown` 也表示任何值，但是 TypeScript 会要求你再做检查，细化类型（见 6.1.5 节）。

`unknown` 类型支持哪些操作呢？`unknown` 类型的值可以比较（使用 `==`、`===`、`||`、`&&` 和 `?`）、可以否定（使用 `!`）、（与其他任何类型一样）可以使用 JavaScript 的 `typeof` 和 `instanceof` 运算符细化。`unknown` 的用法如下：

```
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
}
```

通过这个示例，我们大致可以了解 `unknown` 的用法：

1. TypeScript 不会把任何值推导为 `unknown` 类型，必须显式注解（a）。^{注1}
2. `unknown` 类型的值可以比较（b）。
3. 但是执行操作时不能假定 `unknown` 类型的值为某种特定类型（c），必须先向 TypeScript 证明一个值确实是某个类型（d）。

3.2.3 boolean

`boolean` 类型有两个值：`true` 和 `false`。该类型的值可以比较（使用 `==`、`===`、`||`、`&&` 和 `?`）、可以否定（使用 `!`），此外则没有多少操作。`boolean` 类型的用法如下：

注1：差不多是这样。如果 `unknown` 是并集类型的一部分，联合的结果是 `unknown`。并集类型的详细说明见“并集类型和交集类型”一节。

```
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'.
```

这个示例表明我们可以通过多种方式告诉 TypeScript 一个值的类型为 `boolean`：

1. 可以让 TypeScript 推导出值的类型为 `boolean` (`a` 和 `b`)。
2. 可以让 TypeScript 推导出值为某个具体的布尔值 (`c`)。
3. 可以明确告诉 TypeScript，值的类型为 `boolean` (`d`)。
4. 可以明确告诉 TypeScript，值为某个具体的布尔值 (`e` 和 `f`)。

一般来说，我们在程序中采用第一种或第二种方式。极少数情况下使用第四种方式，仅当需要额外提升类型安全时（后文将对此举例）。第三种方式几乎从不使用。

第二种和第四种情况比较特殊，虽然也算直观，但是很少有其他编程语言支持，因此你可能并不熟悉。那个例子的意思是，“嗨！TypeScript，看到变量 `e` 了吗？`e` 可不是普通的 `boolean` 类型，而是只为 `true` 的 `boolean` 类型。”把类型设为某个值，就限制了 `e` 和 `f` 在所有布尔值中只能取指定的那个值。这个特性称为类型字面量（type literal）。

类型字面量

仅表示一个值的类型。

第四种情况使用类型字面量显式注解了变量，而第二种情况则由 TypeScript 推导出一个字面量类型，因为这里使用的是 `const`，而不是 `let` 或 `var`。使用

`const` 声明的基本类型的值，赋值之后便无法修改，因此 TypeScript 推导出的是范围最窄的类型。所以，在第二种情况中，TypeScript 推导出的 `c` 的类型为 `true`，而不是 `boolean`。TypeScript 推导出的类型为什么会根据使用 `let` 和 `const` 而有所不同呢？请阅读 6.1.4 节。

后文还将多次讲到类型字面量。这是一个非常强大的语言特性，能让安全性更上一层楼。类型字面量使 TypeScript 在编程语言的世界中鹤立鸡群，是 Java 程序员可望不可及的。

3.2.4 number

`number` 包括所有数字：整数、浮点数、正数、负数、`Infinity`、`Nan` 等。显然，数字可以做算术运算，例如加（+）、减（-）、求模（%）和比较（<）。来看几个示例：

```
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'.
```

与讲解 `boolean` 类型时所举的示例类似，把值声明为 `number` 类型也有四种方式：

1. 可以让 TypeScript 推导出值的类型为 `number` (`a` 和 `b`)。
2. 可以使用 `const`，让 TypeScript 推导出值为某个具体的数字 (`c`)。
3. 可以明确告诉 TypeScript，值的类型为 `number` (`e`)。
4. 可以明确告诉 TypeScript，值为某个具体的数字 (`f` 和 `g`)。

与 `boolean` 类型相同的一点是，我们通常让 TypeScript 自己推导类型（第一种方式）。偶尔，我们可能会做些巧妙的编程设计，要求数字的类型限制为

特定的值（第二种方式或第四种方式）。如果没有特殊原因，不要把值的类型显式注解为 `number`（第三种方式）。



处理较长的数字时，为了便于辨识数字，建议使用数字分隔符。在类型和值所在的位置上都可以使用数字分隔符。

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

3.2.5 bigint

`bigint` 是 JavaScript 和 TypeScript 新引入的类型，在处理较大的整数时，不用再担心舍入误差。`number` 类型表示的整数最大为 2^{53} ，`bigint` 能表示的数比这大得多。`bigint` 类型包含所有 `BigInt` 数，支持加 (+)、减 (-)、乘 (*)、除 (/) 和比较 (<)。用法如下：

```
let a = 1234n          // bigint
const b = 5678n       // 5678n
var c = a + b         // bigint
let d = a < 1235      // boolean
let e = 88.5n         // Error TS1353: A bigint literal must be an
                     // integer.
let f: bigint = 100n   // bigint
let g: 100n = 100n    // 100n
let h: bigint = 100    // Error TS2322: Type '100' is not assignable
                     // to type 'bigint'.
```

与 `boolean` 和 `number` 一样，声明为 `bigint` 类型也有四种方式。尽量让 TypeScript 推导 `bigint` 类型。



写作本书时，不是所有 JavaScript 引擎都支持 `bigint`。如果你的应用依赖于 `bigint`，一定要检查目标平台是否支持。

3.2.6 string

`string` 包含所有字符串，以及可以对字符串执行的操作，例如拼接（`+`）、切片（`.slice`）等。来看几个示例：

```
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'    // Error TS2322: Type "zoe" is not assignable
                        // to type "john".
```

与 `boolean` 和 `number` 一样，声明为 `string` 类型也有四种方式。尽量让 TypeScript 推导 `string` 类型。

3.2.7 symbol

`symbol` 是一个相对较新的语言特性，由最新的 JavaScript 主版本（ES2015）引入。其实，符号不太常用，不过经常用于代替对象和映射的字符串键，确保使用正确的已知键，以防键被意外设置，例如设置对象的默认迭代器（`Symbol.iterator`），或者在运行时覆盖不管对象是什么的实例（`Symbol.hasInstance`）。符号的类型为 `symbol`，可对符号执行的操作没有多少。

```
let a = Symbol('a')        // symbol
let b: symbol = Symbol('b') // symbol
var c = a === b           // boolean
let d = a + 'x'            // Error TS2469: The '+' operator cannot be applied
                          // to type 'symbol'.
```

在 JavaScript 中，`Symbol('a')` 使用指定的名称新建一个符号，这个符号是唯一的，不与其他任何符号相等（使用 `==` 或 `===` 比较），即便再使用相同的名称创建一个符号也是如此。我们知道，使用 `let` 声明的值 `27` 将推导为 `number` 类型，而使用 `const` 声明时则为具体的数字 `27`。类似地，符号经推导得到的类型是 `symbol`，此外也可以显式声明为 `unique symbol` 类型：

```
const e = Symbol('e')          // typeof e
const f: unique symbol = Symbol('f') // typeof f
let g: unique symbol = Symbol('f') // Error TS1332: A variable whose type is a
                                  // 'unique symbol' type must be 'const'.
let h = e === e              // boolean
let i = e === f              // Error TS2367: This condition will always return
                            // 'false' since the types 'unique symbol' and
                            // 'unique symbol' have no overlap.
```

这个示例展示了几种创建特定符号的方式：

1. 使用 `const`（而不是 `let` 或 `var`）声明的符号，TypeScript 推导为 `unique symbol` 类型。在代码编辑器中显示为 `typeof yourVariableName`，而不是 `unique symbol`。
2. 可以显式注解 `const` 变量的类型为 `unique symbol`。
3. `unique symbol` 类型的值始终与自身相等。
4. TypeScript 在编译时知道一个 `unique symbol` 类型的值绝不会与另一个 `unique symbol` 类型的值相等。

`unique symbol` 与其他字面量类型其实是一样的，比如 `1`、`true` 或 `"literal"`，是创建表示特定符号的类型的方式。

3.2.8 对象

TypeScript 的对象类型表示对象的结构。注意，通过对象类型无法区分不同的简单对象（使用 `{}` 创建）或复杂的对象（使用 `new Blah` 创建）。这是一种设计选择，JavaScript 一般采用结构化类型，TypeScript 直接沿用，而没有采用名义化类型。

结构化类型

一种编程设计风格，只关心对象有哪些属性，而不管属性使用什么名称（名义化类型）。在某些语言中也叫鸭子类型（即不以貌取人）。

在 TypeScript 中使用类型描述对象有好几种方式。第一种，把一个值声明为 `object` 类型：

```
let a: object = {  
  b: 'x'  
}
```

访问 `b` 时会发生什么呢？

```
a.b // Error TS2339: Property 'b' does not exist on type 'object'.
```

慢着，这不合理啊！把一个值声明为 `object` 类型，却做不了任何操作，这有什么意义呢？

能提出这样的问题，说明你对 TypeScript 有一定的了解。其实，`object` 仅比 `any` 的范围窄一些，但也窄不了多少。`object` 对值知之甚少，只能表示该值是一个 JavaScript 对象（而且不是 `null`）。

如果不显式注解，让 TypeScript 接手呢？

```
let a = {  
  b: 'x'  
}           // {b: string}  
a.b         // string  
  
let b = {  
  c: {  
    d: 'f'  
  }  
}           // {c: {d: string}}
```

恭喜，你刚刚发现了声明对象类型的第二种方式，称为对象字面量句法（不要与类型字面量搞混了）。你可以让 TypeScript 推导对象的结构，也可以在花括号 (`{}`) 内明确描述：

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

使用 const 声明对象时的类型推导

使用 `const` 声明对象情况又如何呢？

```
const a: {b: number} = {  
    b: 12  
} // 仍是 {b: number}
```

让人惊讶的是，TypeScript 推导出的 `b` 是一个数字，而不是字面量 `12`。前面介绍 `number` 和 `string` 类型时讲过，`const` 和 `let` 对 TypeScript 的推导结果是有影响的。

与目前所讲的基本类型 (`boolean`、`number`、`bigint`、`string` 和 `symbol`) 不同，使用 `const` 声明对象不会导致 TypeScript 把推导的类型缩窄。这是因为 JavaScript 对象是可变的，所以在 TypeScript 看来，创建对象之后你可能会更新对象的字段。

我们将在 6.1.4 节深入讨论这个话题，并说明如何缩窄推导的类型。

对象字面量句法的意思是，“这个东西的结构是这样的。”这个“东西”可能是一个对象字面量，也可能是一个类：

```
let c: {  
    firstName: string  
    lastName: string  
} = {  
    firstName: 'john',  
    lastName: 'barrowman'  
}  
  
class Person {  
    constructor(  
        public firstName: string, // public 是 this.firstName = firstName  
                                // 的简写形式  
        public lastName: string  
    ) {}  
}  
c = new Person('matt', 'smith') // OK
```

{firstName: string, lastName: string} 描述的是一个对象的结构，上述示例中的对象字面量和类实例都满足该结构，因此 TypeScript 允许我们把一个 Person 实例赋值给 c。

下面来看添加额外的属性或者缺少必要的属性时会发生什么：

```
let a: {b: number}

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

a = {
  b: 1,
  c: 2 // Error 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

这是我们第一次遇到先声明变量 (a) 再使用值 ({} 和 {b: 1, c: 2}) 初始化的情况。这是一种常见的 JavaScript 模式，TypeScript 也支持。

如果先声明变量，然后再初始化，TypeScript 将确保在使用该变量时已经明确为其赋值了：

```
let i: number
let j = i * 3 // Error TS2454: Variable 'i' is used
              // before being assigned.
```

别担心，即便没有显式注解类型，TypeScript 也会强制检查：

```
let i
let j = i * 3 // Error TS2532: Object is possibly
              // 'undefined'.
```

默认情况下，TypeScript 对对象的属性要求十分严格。如果声明对象有个类型为 `number` 的属性 `b`，TypeScript 将预期对象有这么一个属性，而且只有这个属性。如果缺少 `b` 属性，或者多了其他属性，TypeScript 将报错。

那么，有没有什么方法可以告诉 TypeScript 某个属性是可选的，或者实际的属性可能比计划的多呢？当然有：

```
let a: {  
    b: number ①  
    c?: string ②  
    [key: number]: boolean ③  
}
```

- ① `a` 有个类型为 `number` 的属性 `b`。
- ② `a` 可能有个类型为 `string` 的属性 `c`。如果有属性 `c`，其值可以为 `undefined`。
- ③ `a` 可能有任意多个数字属性，其值为布尔值。

下面看一下可以把哪些类型的对象赋值给 `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}          // Error TS2741: Property 'b' is missing in type  
                      // '{10: true}'.  
a = {b: 1, 33: 'red'} // Error TS2741: Type 'string' is not assignable  
                      // to type 'boolean'.
```

索引签名

[key: T]: U 句法称为索引签名，我们通过这种方式告诉 TypeScript，指定的对象可能有更多的键。这种句法的意思是，“在这个对象中，类型为 T 的键对应的值为 U 类型。”借助索引签名，除显式声明的键之外，可以放心添加更多的键。

索引签名还有一条规则要记住：键的类型 (T) 必须可赋值给 `number` 或 `string`。^{注 2}

另外注意，索引签名中键的名称可以是任何词，不一定非得用 `key`：

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

声明对象类型时，可选符号 (?) 不是唯一可用的修饰符。此外，还可以使用 `readonly` 修饰符把字段标记为只读（即指明为字段赋予初始值之后无法再修改，类似于使用 `const` 声明对象的属性）：

```
let user: {
  readonly firstName: string
} = {
  firstName: 'abby'
}

user.firstName // string
user.firstName =
  'abbey with an e' // Error TS2540: Cannot assign to 'firstName' because it
                    // is a read-only property.
```

对象字面量表示法有一个特例：空对象类型 (`{}`)。除 `null` 和 `undefined` 之

注 2：JavaScript 对象的键为字符串；数组是特殊的对象，键为数字。

外的任何类型都可以赋值给空对象类型，使用起来比较复杂。请尽量避免使用空对象类型。

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

最后，还要讲一种声明对象类型的方式：`Object`。这与`{}`的作用基本一样，最好也避免使用。^{注3}

综上所述，在 TypeScript 中声明对象类型有四种方式：

1. 对象字面量表示法（例如`{a: string}`），也称对象的结构。如果知道对象有哪些字段，或者对象的值都为相同的类型，使用这种方式。
2. 空对象字面量表示法（`{}`）。尽量避免使用这种方式。
3. `object`类型。如果需要一个对象，但对对象的字段没有要求，使用这种方式。
4. `Object`类型。尽量避免使用这种方式。

在 TypeScript 程序中，应该坚持使用第一种和第三种方式。第二种和第四种方式应尽量避免使用。你可以使用 linter 提醒、在代码审查阶段提出异议、打印一张海报时刻提醒自己，或者使用团队在用的工具让代码基远离这两种方式。

上述列表中第 2~4 种方式的快速参考见表 3-1。

注 3： 技术细节上有点细微差别：使用`{}`时，可以把`Object`原型（参阅 MDN，<https://mzl.la/2VSuDJz>）内置的方法（例如`.toString`和`.hasOwnProperty`）定义为任何类型，而`Object`要求声明的类型必须可赋值给`Object`原型内置的类型。例如，`let a: {} = {toString() { return 3 }}`能通过类型检查。但是，如果把类型注解改成`Object`，变为`let b: Object = {toString() { return 3 }}`，TypeScript 将报错：`Error TS2322: Type 'number' is not assignable to type 'string'.`

表 3-1：这个值是否是有效的对象

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

3.2.9 中场休息：类型别名、并集和交集

你正在快速成长为 TypeScript 专家级程序员。我们学习了几个类型，知道它们的运作方式，也熟悉了类型系统、类型、安全性等概念。是时候更进一步了。

我们知道，对一个值，在类型允许的情况下，我们可以对其执行特定的操作。例如，可以使用 `+` 计算两个数字的和，可以调用 `.toUpperCase` 方法把字符串变成大写。

其实，在类型自身上也可以执行一些操作。本节介绍几个类型层面的操作，这样的操作还有很多，后文再做说明，只是这几个操作十分常用，所以笔者觉得有必要尽早介绍。

类型别名

我们可以使用变量声明（`let`、`const` 和 `var`）为值声明别名，类似地，还可以为类型声明别名。具体做法如下：

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

```
    age: Age
}
}
```

Age 就是一个 number。这样可以让 Person 的结构定义更易于理解。

TypeScript 无法推导类型别名，因此必须显式注解：

```
let age: Age = 55

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

由于 Age 是 number 的别名，因此可以赋值给 number，所以上述代码也可以写成：

```
let age = 55

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

使用类型别名的地方都可以替换成源类型，程序的语义不受影响。

与 JavaScript 中的变量声明 (let、const 和 var) 一样，同一类型不能声明两次：

```
type Color = 'red'
type Color = 'blue' // Error TS2300: Duplicate identifier 'Color'.
```

同样与 let 和 const 一样的是，类型别名采用块级作用域。每一块代码和每一个函数都有自己的作用域，内部的类型别名将遮盖外部的类型别名：

```
type Color = 'red'

let x = Math.random() < .5

if (x) {
  type Color = 'blue' // 遮盖上面声明的 Color
  let b: Color = 'blue'
} else {
```

```
let c: Color = 'red'  
}
```

类型别名有助于减少重复输入复杂的类型（DRY），^{注4}还能更清楚地表明变量的作用（较之具有描述性的变量名称，有些人更喜欢使用具有描述性的类型名称）。判断是否为类型创建别名的方法，与判断是否把值声明为变量的方法一样。

并集类型和交集类型

给定两个事物 A 和 B，它们的并集是二者的内容之和（包括在 A、B 中，以及同时在二者中的内容），它们的交集是二者共有的内容（既在 A 中也在 B 中）。借助集合最好理解这两个概念。在图 3-2 中，圆表示集合，左边是两个集合的并集（即二者之和），右边是两个集合的交集（即二者之积）。

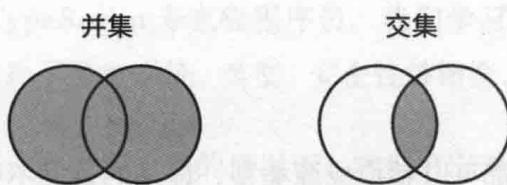


图 3-2：并集 (l) 和交集 (&)

为了处理类型的并集和交集，TypeScript 提供了特殊的类型运算符：并集使用 |，交集使用 &。类型与集合有很多相似之处，两者之间基本可以画上等号：

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

如果某个值是 CatOrDogOrBoth 类型，那么我们将得知什么信息呢？我们知道该值有个字符串类型的 name 属性，此外没有什么了。换个角度，哪些值可以

注 4：DRY 是“Don't Repeat Yourself”的首字母缩写，意思是不要重复编写代码。这个思想由 Andrew Hunt 和 David Thomas 在《程序员修炼之道：从小工到专家》一书中提出。

赋值给 `CatOrDogOrBoth` 类型的变量呢？可以是 `Cat` 类型的值，也可以是 `Dog` 类型的值，还可以二者兼具。

```
// Cat
let a: CatOrDogOrBoth = {
    name: 'Bonkers',
    purrs: true
}
```

```
// Dog
a = {
    name: 'Domino',
    barks: true,
    wags: true
}
```

```
// 二者兼具
a = {
    name: 'Donkers',
    barks: true,
    purrs: true,
    wags: true
}
```

重申一下，一个并集类型（`|`）的值不一定属于并集中的某一个成员，还可以同时属于每个成员。^{注5}

另一方面，从 `CatAndDog` 类型中我们能得知什么信息呢？你的犬猫混种超级宠物不仅有名字（`name`），还能喵喵叫（`purr`）、汪汪叫（`bark`）及摇尾巴（`wag`）。

```
let b: CatAndDog = {
    name: 'Domino',
    barks: true,
    purrs: true,
    wags: true
}
```

注5：如果并集不相交，那么值只能属于并集类型中的某个成员，不能同时属于每个成员。让 TypeScript 知道并集不相交的方法参见“辨别并集类型”一节。

并集通常比交集更符合常理。以下面的函数为例：

```
function trueOrNull(isTrue: boolean) {  
    if (isTrue) {  
        return 'true'  
    }  
    return null  
}
```

这个函数的返回值是什么类型呢？可能是一个字符串，也能是 `null`。这样的类型可以表示为：

```
type Returns = string | null
```

那么下面这个函数呢？

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

如果 `a` 是真值，返回值的类型为 `string`，否则为 `number`，即 `string | number`。

另一个更适合使用并集的是数组（尤其是混合类数组），下面具体讨论。

3.2.10 数组

与在 JavaScript 中一样，TypeScript 中的数组也是特殊类型的对象，支持拼接、推入、搜索和切片等操作。举几个例子：

```
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'.
```



TypeScript 支持两种注解数组类型的句法：`T[]` 和 `Array<T>`。二者的作用和性能无异。本书采用 `T[]` 句法，因为写法更简洁。你在编写代码时，可以选择任何一种自己喜欢的方式。

阅读上述示例可以看出，除了 `c` 和 `h` 之外，其他变量都没有显式注解类型。而且，TypeScript 对什么样的值可以放入数组也有规定。

一般情况下，数组应该保持同质。也就是说，不要在同一个数组中存储苹果、橙子和数字。设计程序时要规划好，保证数组中的每个元素都具有相同的类型。如若不然，我们要做些额外的工作，让 TypeScript 相信我们执行的操作是安全的。

若想知道为什么要使用同质数组，请看变量 `f`。这个数组使用字符串 `'red'` 初始化（声明该数组时使用的元素是字符串，经推导，TypeScript 认定这必是一个字符串数组），然后推入 `'blue'`。`'blue'` 是字符串，因此 TypeScript 允许执行该操作。接着尝试把 `true` 推入数组，但却失败了。为什么呢？因为 `f` 是字符串数组，而 `true` 不是字符串。

另一边，初始化 `d` 时传入的元素是一个数字和一个字符串，经推导，TypeScript 认定这个数组的类型必是 `number | string`。由于这个数组中的元素可以是数字也可以是字符串，因此使用之前必须检查。比如说，我们想映射这个数组，把字母变成大写，把数字乘以 3：

```
let d = [1, 'a']
```

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

为此，必须使用 `typeof` 检查每个元素的类型，判断元素是数字还是字符串，然后再做相应的操作。

与对象一样，使用 `const` 声明数组不会导致 TypeScript 推导出范围更窄的类型。鉴于此，TypeScript 推导出的 `d` 和 `e` 的类型均为 `number | string`。

`g` 比较特殊。初始化空数组时，TypeScript 不知道数组中元素的类型，推导出的类型为 `any`。向数组中添加元素后，TypeScript 开始拼凑数组的类型。当数组离开定义时所在的作用域后，TypeScript 将最终确定一个类型，不再扩张：

```
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'.
```

假如一直使用 `any`，这个示例也不会让你急出一身汗。

3.2.11 元组

元组是 `array` 的子类型，是定义数组的一种特殊方式，长度固定，各索引位上的值具有固定的已知类型。与其他多数类型不同，声明元组时必须显式注解类型。这是因为，创建元组使用的句法与数组相同（都使用方括号），而 TypeScript 遇到方括号，推导出来的是数组的类型。

```
let a: [number] = [1]
```

```
// [名, 姓, 出生年份] 形式的元组
let b: [string, string, number] = ['malcolm', 'gladwell', 1963]

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

元组也支持可选的元素。与在对象的类型中一样，? 表示“可选”：

```
// 火车票价数组, 不同的方向价格有时不同
let trainFares: [number, number?][] = [
  [3.75],
  [8.25, 7.70],
  [10.50]
]

// 等价于
let moreTrainFares: ([number] | [number, number])[] = [
  // ...
]
```

元组也支持剩余元素，即为元组定义最小长度：

```
// 字符串列表, 至少有一个元素
let friends: [string, ...string[]} = ['Sara', 'Tali', 'Chloe', 'Claire']

// 元素类型不同的列表
let list: [number, boolean, ...string[]} = [1, false, 'a', 'b', 'c']
```

元组类型能正确定义元素类型不同的列表，还能知晓该种列表的长度。这些特性使得元组比数组安全得多，应该经常使用。

只读数组和元组

常规的数组是可变的（可以使用 `.push` 方法把元素推入数组、使用 `.splice` 编接数组，还可以就地更新数组），这也是多数时候我们想要的行为，不过有时我们希望数组不可变，修改之后得到新的数组，而原数组没有变化。

TypeScript 原生支持只读数组类型，用于创建不可变的数组。只读数组与常规的数组没有多大差别，只是不能就地更改。若想创建只读数组，要显式注解

类型；若想更改只读数组，使用非变型方法，例如 `.concat` 和 `.slice`，不能使用可变型方法，例如 `.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[]'.
```

我们知道，注解数组类型还可以使用 `Array`。类似地，声明只读数组和元组，也可以使用长格式句法：

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

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

使用简洁的 `readonly` 修饰符，还是使用较长的 `Readonly` 或 `ReadonlyArray` 句法，全凭个人喜好。

注意，只读数组不可变的特性能让代码更易于理解，不过其背后提供支持的仍是常规的 JavaScript 数组。这意味着，即便只对数组做小小的改动，也要先复制整个原数组，如有不慎，会影响应用的运行性能。对小型数组来说，影响微乎其微，但是对大型数组可能会造成极大的影响。



如果打算大量使用不可变数组，请找一种更为高效的实现，Lee Byron 开发的 `immutable` 包 (<https://www.npmjs.com/package/immutable>) 就很不错。

3.2.12 null、undefined、void 和 never

在 JavaScript 中，有两个值表示缺少什么：`null` 和 `undefined`。TypeScript 也支持这两个值，而且各自都有类型。你可以猜一下类型的名称是什么？没错，类型名称也是 `null` 和 `undefined`。

这两个类型比较特殊，在 TypeScript 中，`undefined` 类型只有 `undefined` 一个值，`null` 类型只有 `null` 一个值。

JavaScript 程序员往往不区分二者，但是它们在语义上有细微的差别：`undefined` 的意思是尚未定义，而 `null` 表示缺少值（例如在计算一个值的过程中遇到了错误）。这只是一个约定，TypeScript 并不强迫你遵守这样的规则，不过知道它们的区别也没坏处。

除了 `null` 和 `undefined` 之外，TypeScript 还有 `void` 和 `never` 类型。这两个类型有明确的特殊作用，进一步划分不同情况下的“不存在”：`void` 是函数没有显式返回任何值（例如 `console.log`）时的返回类型，而 `never` 是函数根本不返回（例如函数抛出异常，或者永远运行下去）时使用的类型。

```
// (a) 一个返回数字或 null 的函数
function a(x: number) {
    if (x < 10) {
        return x
    }
    return null
}

// (b) 一个返回 undefined 的函数
function b() {
    return undefined
}

// (c) 一个返回 void 的函数
function c() {
    let a = 2 + 2
    let b = a * a
}
```

```
// (d) 一个返回 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`，但是没有使用 `return` 语句明确指定，于是我们说该函数返回 `void`。 (d) 抛出异常， (e) 一直运行，因此二者都不返回，所以我们说这两个函数的返回类型为 `never`。

如果说 `unknown` 是其他每个类型的父类型，那么 `never` 就是其他每个类型的子类型。我们可以把 `never` 理解为“兜底类型”。这意味着，`never` 类型可赋值给其他任何类型，在任何地方都能放心使用 `never` 类型的值。这一点基本上只具有理论意义，不过在与其他语言熟手讨论 TypeScript 时可以提及这一点，炫耀自己的知识储备。

这四个表示缺失的类型总结见表 3-2。

表 3-2：表示缺少什么的类型

类型	含义
<code>null</code>	缺少值
<code>undefined</code>	尚未赋值的变量
<code>void</code>	没有 <code>return</code> 语句的函数
<code>never</code>	永不返回的函数

严格检查 null

在旧版 TypeScript 中（或者把 TSC 的 `strictNullChecks` 选项设为 `false`），`null` 的行为稍有不同：`null` 是除 `never` 之外所有类型的子类型。这意味着，每个类型的值都可能为 `null`，因此必须首先检查是否为 `null` 值，否则不能信任任何类型的值。例如，我们想在传入函数的变量 `pizza` 上调用 `.addAnchovies` 方法，在披萨上撒一些美味的小鱼之前必须检查 `pizza` 的值是否为 `null`。在实际使用中，如果对每个变量都做这样的检查，显得十分烦琐，多数人都会忘记这一步。可是，倘若真的遇到 `null` 值，在运行时将出现后果严重的空指针异常：

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

// 如果 strictNullChecks = false, TypeScript 将放行
addDeliciousFish(null)
```

20 世纪 60 年代引入 `null` 的人称之为“价值十亿的错误” (<http://bit.ly/2WEdZNO>)。`null` 的问题是多数语言的类型系统无法表示这个值，因此也不做检查。那么，程序员在执行操作时如果认为变量已经定义，但在运行时却发现实际的值为 `null`，代码将抛出运行时异常。

为什么呢？别问我，我只负责写这本书。不过编程语言已经意识到了这个问题，在类型系统中实现了 `null`，TypeScript 就是很好的例子。如果想在编译时尽量捕获更多的缺陷，不要等到用户发现才回过头修正，那么类型系统就有必要检查 `null`。

3.2.13 枚举

枚举的作用是列举类型中包含的各个值。这是一种无序数据结构，把键映射到值上。枚举可以理解为编译时键固定的对象，访问键时，TypeScript 将检查指定的键是否存在。

枚举分为两种：字符串到字符串之间的映射和字符串到数字之间的映射。如下所示：

```
enum Language {  
    English,  
    Spanish,  
    Russian  
}
```



按约定，枚举名称为大写的单数形式。枚举中的键也为大写。

TypeScript 可以自动为枚举中的各个成员推导对应的数字，你也可以自己手动设置。下面给出经 TypeScript 推导后上述示例得到的结果：

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

枚举中的值使用点号或方括号表示法访问，就像访问常规对象中的值一样：

```
let myFirstLanguage = Language.Russian      // Language  
let mySecondLanguage = Language['English']  // Language
```

一个枚举可以分成几次声明，TypeScript 将自动把各部分合并在一起（详情参见 10.4 节）。注意，如果分开声明枚举，TypeScript 只能推导出其中一部分的值，因此最好为枚举中的每个成员显式赋值。

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

成员的值可以经计算得出，而且不必为所有成员都赋值（TypeScript 将尽自己所能推导缺少的值）：

```
enum Language {  
    English = 100,  
    Spanish = 200 + 300,  
    Russian  
} // TypeScript 推导出的值为 501(即 500 后的下一个数)
```

枚举的值也可以为字符串，甚至混用字符串和数字：

```
enum Color {  
    Red = '#c10000',  
    Blue = '#007ac1',  
    Pink = 0xc10050, // 十六进制字面量  
    White = 255 // 十进制字面量  
}  
  
let red = Color.Red // Color  
let pink = Color.Pink // Color
```

TypeScript 比较灵活，既允许通过值访问枚举，也允许通过键访问，不过这样极易导致问题：

```
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 (!!!)
```

其实 `Color[6]` 不存在，但是 TypeScript 并不阻止你这么做。为了避免这种不安全的访问操作，可以通过 `const enum` 指定使用枚举的安全子集。下面使用该方法重写前面的 `Language` 枚举：

```
const enum Language {  
    English,  
    Spanish,  
    Russian  
}
```

```
// 访问一个有效的枚举键
let a = Language.English // Language

// 访问一个无效的枚举键
let b = Language.Tagalog // Error TS2339: Property 'Tagalog' does not exist
                          // on type 'typeof Language'.

// 访问一个有效的枚举值
let c = Language[0]      // Error TS2476: A const enum member can only be
                          // accessed using a string literal.

// 访问一个无效的枚举值
let d = Language[6]      // Error TS2476: A const enum member can only be
                          // accessed using a string literal.
```

`const enum` 不允许反向查找，行为与常规的 JavaScript 对象很像。另外，默认也不生成任何 JavaScript 代码，而是在用到枚举成员的地方内插对应的值（例如，TypeScript 将把 `Language.Spanish` 替换成对应的值，即 1）。



TSC 标志: `preserveConstEnums`

`const enum` 内插值的行为在从其他人编写的 TypeScript 代码中导入 `const enum` 时可能导致安全问题：假如原作者在你编译 TypeScript 代码之后更新了 `const enum`，那么在运行时你使用的枚举与原作者的枚举指向的值可能不同，而 TypeScript 没有这么智能，无法得知这一变化。

使用 `const enum` 时请尽量避免内插，而且只在受自己控制的 TypeScript 程序中使用。不要在计划发布到 NPM 中的程序，或者开放给其他人使用的库中使用。

如果想为 `const enum` 生成运行时代码，在 `tsconfig.json` 中把 TSC 选项 `preserveConstEnums` 设为 `true`:

```
{
  "compilerOptions": {
    "preserveConstEnums": true
  }
}
```

const enum 的用法如下：

```
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' (!!!)
```

一切都很正常，Chair 和 Cup 都能按预期翻转……直到你发现数字也可赋值给枚举。这个行为是 TypeScript 的赋值规则导致的不良后果，为了修正这个问题，我们要额外小心，只在枚举中使用字符串值：

```
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'.
```

枚举中一个讨厌的数值就能置整个枚举于不安全的境地。



由于使用枚举极易导致安全问题，因此笔者建议远离枚举。同样的意图，在 TypeScript 中有大量更好的方式表达。

如果你的同事坚持使用枚举，怎么劝都不听，那就一定要制订几条 TSLint 规则，发现数值和非 `const` 枚举时发出提醒。

3.3 小结

可以看到，TypeScript 内置了大量类型。我们可以让 TypeScript 根据值推导类型，也可以自己显式注解类型。使用 `const` 时推导出的类型更具体，而 `let` 和 `var` 更一般化。多数类型都分一般和具体两种形式，后者是前者的子类型（见表 3-3）。

表 3-3：类型及更具体的子类型

类型	子类型
<code>boolean</code>	<code>Boolean</code> 字面量
<code>bignumber</code>	<code>BigNumber</code> 字面量
<code>number</code>	<code>Number</code> 字面量
<code>string</code>	<code>String</code> 字面量
<code>symbol</code>	<code>unique symbol</code>
<code>object</code>	<code>Object</code> 字面量
数组	元组
<code>enum</code>	<code>const enum</code>

3.4 练习题

1. 下列各值，TypeScript 推导出的类型是什么？

- `let a = 1042`
- `let b = 'apples and oranges'`
- `const c = 'pineapples'`
- `let d = [true, true, false]`

- e. `let e = {type: 'ficus'}`
 - f. `let f = [1, false]`
 - g. `const g = [3]`
 - h. `let h = null` (先在代码编辑器中试一下, 如果觉得结果让人惊讶,请翻到 6.1.4 节)
2. 为什么下述代码会抛出错误?
- a.
`let i: 3 = 3
i = 4 // Error TS2322: Type '4' is not assignable to type '3'.`
 - b.
`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'.`
 - c.
`let k: never = 4 // Error TSTS2322: Type '4' is not assignable
// to type 'never'.`
 - d.
`let l: unknown = 4
let m = l * 2 // Error TS2571: Object is of type 'unknown'.`

第 4 章

函数

3.3 小结

前一章介绍了 TypeScript 的类型系统，涵盖基本类型、对象、数组、元组和枚举，还说明了 TypeScript 类型推导的基本方法，以及类型可赋值性的概念。现在该端上 TypeScript 的主菜了（对函数式程序员来说是立命之本），即函数。本章涵盖以下话题：

- 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。

4.1 声明和调用函数

在 JavaScript 中，函数是一等对象。这意味着，我们可以像对象那样使用函数：可以赋值给变量；可以作为参数传给其他函数；可以作为函数的返回值；可以赋值给对象和原型；可以赋予属性；可以读取属性等。在 JavaScript 中对函数可执行的操作有很多，TypeScript 通过丰富的类型系统延续了这一传统。

TypeScript 函数是下面这样的（读过前一章之后，你应该不会感到陌生）：

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

通常，我们会显式注解函数的参数（上例中的 a 和 b）。TypeScript 能推导出函数体中的类型，但是多数情况下无法推导出参数的类型，只在少数特殊情况下能根据上下文推导出参数的类型（详见 4.1.8 节）。返回类型能推导出来，不过也可以显式注解：

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



基于有利读者理解函数的作用，本书会显式注解返回类型。除此之外，都将省略注解。既然 TypeScript 自己能推导，为什么还要重复劳动呢？

上述示例声明函数使用的是具名函数句法，不过 JavaScript 和 TypeScript 至少还支持五种声明函数的方式：

```
// 具名函数  
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  
}  
  
// 箭头函数表达式简写形式  
let greet4 = (name: string) =>  
    'hello ' + name
```

```
// 函数构造方法  
let greet5 = new Function('name', 'return "hello " + name')
```

除了函数构造方法,^{注1} 其他几种句法在 TypeScript 中都可以放心使用，能确保类型安全，而且遵守的注解规则也是一样：通常需要注解参数的类型，而返回类型不要求必须注解。



简单回顾一下相关术语：

- 形参 (parameter, 也称 *formal parameter*) 是声明函数时指定的运行函数所需的数据。
- 实参 (argument, 也称 *actual parameter*) 是调用函数时传给函数的数据。

在 TypeScript 中调用函数时，无需提供任何额外的类型信息，直接传入实参即可，TypeScript 将检查实参是否与函数形参的类型兼容。

```
add(1, 2)          // 求值结果为 3  
greet('Crystal') // 求值结果为 'hello Crystal'
```

当然，如果忘记传入某个参数，或者传入的参数类型有误，TypeScript 将指出问题：

```
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'.
```

4.1.1 可选和默认的参数

与对象和元组类型一样，可以使用? 把参数标记为可选的。声明函数的参数时，必要的参数放在前面，随后才是可选的参数：

注 1：为什么函数构造方法不安全呢？在代码编辑器中输入最后一个例子，你会发现其类型为 Function。这是什么类型呢？这是一种可调用的对象（即可以在后面加上()），而且具有 Function.prototype 的所有原型方法。但是这里没有体现参数和返回值的类型，因此可以使用任何参数调用函数，TypeScript 会袖手旁观，眼看着你做一些在任何地方都不合法的事情。

```
function log(message: string, userId?: string) {  
    let time = new Date().toLocaleTimeString()  
    console.log(time, message, userId || 'Not signed in')  
}  
  
log('Page loaded') // Logs "12:38:31 PM Page loaded Not signed in"  
log('User signed in', 'da763be') // Logs "12:38:31 PM User signed in da763be"
```

与在 JavaScript 中一样，可以为可选参数提供默认值。这样做在语义上与把参数标记为可选的一样，即在调用时无需传入参数的值（区别是，带默认值的参数不要求放在参数列表的末尾，而可选参数必须放在末尾）。

例如，上述 log 函数可以改写为：

```
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，我们为 userId 提供了默认值，而且删除了可选参数注解 ?。这个参数在调用时可以不提供值。TypeScript 足够智能，能根据默认值推导出参数的类型，从而保证代码简洁、易于理解。

当然，如果愿意也可以显式注解默认参数的类型，就像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)  
}
```

较之可选参数，更常使用默认参数。

4.1.2 剩余参数

如果一个函数接受一组参数，简单起见，可以通过一个数组传入这些参数：

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

sum([1, 2, 3]) // 求值结果为 6
```

不过，有时我们需要的是可变参数函数，即参数的数量不定，而不想让参数的数量固定。以前，这个需求通过 JavaScript 神奇的 `arguments` 对象实现。

说 `arguments` 神奇，是因为 JavaScript 在运行时自动在函数内定义该对象，并把传给函数的参数列表赋予该对象。`arguments` 是个类似数组的对象，在调用内置的 `.reduce` 方法之前要把它转换成数组：

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

sumVariadic(1, 2, 3) // 求值结果为 6
```

可是，使用 `arguments` 有个比较大的问题：根本不安全！如果你在代码编辑器中把鼠标悬停在 `total` 或 `n` 上，将看到类似图 4-1 中的输出。

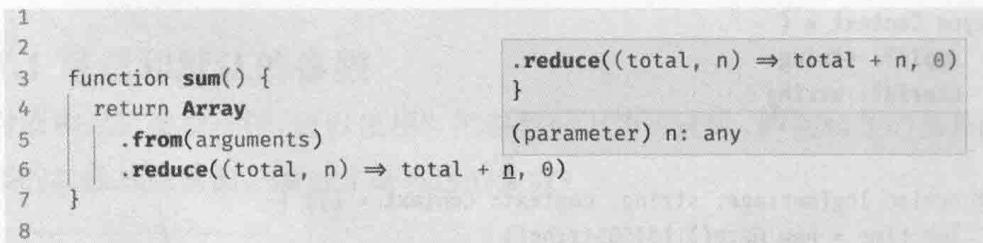


图 4-1: `arguments` 不安全

可以看到，经 TypeScript 推导，`n` 和 `total` 的类型都是 `any`。这样问题就漏过去了，到使用时才会报错：

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

由于声明 `sumVariadic` 函数时没有指定参数，因此在 TypeScript 看来，该函数不接受任何参数，所以使用时会得到 `TypeError`。

那么，怎样确保可变参数函数的类型安全呢？

这就要使用剩余参数（rest parameter）了。`arguments` 对象虽然神奇，却不安全。为了确保 `sum` 函数可以安全接受任意个参数，应该使用剩余参数。

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

```
sumVariadicSafe(1, 2, 3) // 求值结果为 6
```

就这么简单！注意，可变参数版本的 `sum` 与原来只接受一个参数的 `sum` 函数之间唯有一处不同，即参数列表中多了 `...`，其他都没变，但是却保证了安全。

一个函数最多只能有一个剩余参数，而且必须位于参数列表的最后。以 TypeScript 内置函数 `console.log` 的声明为例（如果你不知道 `interface` 是什么意思，别担心，第 5 章将做说明），该方法接受一个可选的参数 `message`，以及任意个要输出的额外参数：

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

4.1.3 call、apply 和 bind

调用函数，除了使用圆括号 `()` 之外，JavaScript 至少还支持两种其他方式。以本章前面的 `add` 函数为例：

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

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

`apply` 为函数内部的 `this` 绑定一个值，然后展开第二个参数，作为参数传给要调用的函数。`call` 的用法类似，不过是按顺序应用参数的，而不做展开。

`bind()` 差不多，也为函数的 `this` 和参数绑定值。不过，`bind` 并不调用函数，而是返回一个新函数，让你通过 `()`、`.call` 或 `.apply` 调用，而且可以再传入参数，绑定到尚未绑定值的参数上。



TSC 标志: strictBindCallApply

为了安全使用 `.call`、`.apply` 和 `.bind`，要在 `tsconfig.json` 中启用 `strictBindCallApply` 选项（启用 `strict` 模式后自动启用该选项）。

4.1.4 注解 `this` 的类型

如果你以前没用过 JavaScript，有一件事可能会令你惊讶：JavaScript 中的每个函数都有 `this` 变量，而不局限于类中的方法。以不同的方式调用函数，`this` 的值也不同，这极易导致代码脆弱、难以理解。



鉴于此，很多团队禁止在类方法以外使用 `this`。如果你也想这么做，启用 TSLint 的 `no-invalid-this` 规则。

`this` 之所以脆弱，与它的赋值方式有关。一般来说，`this` 的值为调用方法时位于点号左侧的对象。例如：

```
let x = {
  a() {
    return this
  }
}
```

```
}

x.a() // 在 a() 的定义体中, this 的值为 x 对象
```

但是, 倘若在调用 a 之前重新赋值了, 结果将发生变化。

```
let a = x.a
a() // 现在, 在 a() 的定义体中, this 的值为 undefined
```

假设有个格式化日期的实用函数, 如下所示:

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

这是你作为新手程序员时(还未学习函数的参数)设计的 API。调用 fancyDate 时, 要为 this 绑定一个 Date 对象:

```
fancyDate.call(new Date) // 求值结果为 "4/14/2005"
```

如果忘记为 this 绑定 Date 对象, 将出现运行时错误:

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

篇幅有限, 本书不对 this 做全面探讨。^{注2} this 的值取决于调用函数的方式, 而不受声明方式的干扰, 这一点可能会让你感到惊讶。

幸好, 我们有 TypeScript 做后盾。如果函数使用 this, 请在函数的第一个参数中声明 this 的类型(放在其他参数之前), 这样每次调用函数时, TypeScript 将确保 this 的确是你预期的类型。this 不是常规的参数, 而是保留字, 是函数签名的一部分:

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

注2: 如果想深入了解 this, 请阅读 Kyle Simpson 写的“You Don’t Know JS”系列图书(O’Reilly 出版, <http://shop.oreilly.com/product/0636920033738.do>)。

现在，调用 fancyDate 函数将得到如下结果：

```
fancyDate.call(new Date) // 求值结果为 "6/13/2008"  
fancyDate() // Error TS2684: The 'this' context of type 'void' is  
// not assignable to method's 'this' of type 'Date'.
```

运行时错误不见了，TypeScript 收集了足够的信息，可以在编译时提醒此类错误。



TSC 标志：noImplicitThis

如果想强制显式注解函数中 this 的类型，在 `tsconfig.json` 中启用 `noImplicitThis` 设置。`strict` 模式包括 `noImplicitThis`，如果已经启用该模式，就不用设置 `noImplicitThis` 了。

注意，`noImplicitThis` 不强制要求为类或对象的函数注解 this。

4.1.5 生成器函数

生成器函数（简称生成器）是生成一系列值的便利方式。生成器的使用方可以精确控制生成什么值。生成器是惰性的，只在使用方要求时才计算下一个值。鉴于此，可以利用生成器实现一些其他方式难以实现的操作，例如生成无穷列表。

生成器的用法如下：

```
function* createFibonacciGenerator() { ❶  
  let a = 0  
  let b = 1  
  while (true) { ❷  
    yield a; ❸  
    [a, b] = [b, a + b] ❹  
  }  
}  
  
let fibonacciGenerator = createFibonacciGenerator() // IterableIterator<number>  
fibonacciGenerator.next() // 求值结果为 {value: 0, done: false}
```

```
fibonacciGenerator.next() // 求值结果为 {value: 1, done: false}
fibonacciGenerator.next() // 求值结果为 {value: 1, done: false}
fibonacciGenerator.next() // 求值结果为 {value: 2, done: false}
fibonacciGenerator.next() // 求值结果为 {value: 3, done: false}
fibonacciGenerator.next() // 求值结果为 {value: 5, done: false}
```

- ① 函数名称前面的星号 (*) 表明这是一个生成器函数。调用生成器返回一个可迭代的迭代器。
- ② 这个生成器可一直生成值。
- ③ 生成器使用 `yield` 关键字产出值。使用方让生成器提供下一个值时（例如，调用 `next`），`yield` 把结果发给使用方，然后停止执行，直到使用方要求提供下一个值为止。因此，这里的 `while(true)` 循环不会一直运行下去，程序不会崩溃。
- ④ 为了计算下一个斐波纳契数，我们在一步中把 `b` 赋值给 `a`、把 `a + b` 赋值给 `b`。

调用 `createFibonacciGenerator` 得到的是一个 `IterableIterator`。每次调用 `next`，迭代器计算下一个斐波纳契数，然后通过 `yield` 产出。注意，TypeScript 能通过产出值的类型推导出迭代器的类型。

此外，也可以显式注解生成器，把产出值的类型放在 `IterableIterator` 中：

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

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

本书不深入探讨生成器，这是一个很大的话题，毕竟这是一本讲 TypeScript 的书，笔者不想分心去讨论 JavaScript 特性。你只需知道，生成器是

JavaScript 语言一个很酷的特性，而且 TypeScript 也支持。如果想进一步学习生成器，请访问 MDN 上的相关页面 (<https://mzl.la/2UiIk4>)。

4.1.6 迭代器

迭代器是生成器的相对面：生成器是生成一系列值的方式，而迭代器是使用这些值的方式。你可能被术语搞晕了，那我们就先从几个定义入手。

可迭代对象

有 `Symbol.iterator` 属性的对象，而且该属性的值为一个函数，返回一个迭代器。

迭代器

定义有 `next` 方法的对象，该方法返回一个具有 `value` 和 `done` 属性的对象。

创建生成器（例如，调用 `createFibonacciGenerator`），得到的值既是可迭代对象，也是迭代器，称为可迭代的迭代器，因为该值既有 `Symbol.iterator` 属性，也有 `next` 方法。

此外，我们可以自己动手定义迭代器或者可迭代对象，为此只需分别创建实现 `Symbol.iterator` 属性和 `next` 方法的对象（或类）。例如，下面的代码定义一个返回数字 1~10 的迭代器：

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

在代码编辑器中输入上述代码，把鼠标悬停在迭代器上，你将看到 TypeScript 推导出的类型（见图 4-2）。

```
1      let numbers: {
2          [Symbol.iterator](): IterableIterator<number>;
3      }
4
5      let numbers = {
6          *[Symbol.iterator]() {
7              for (let n = 1; n <= 10; n++) {
8                  yield n
9              }
10         }
11     }
```

图 4-2：自己定义一个迭代器

也就是说，`numbers` 是一个迭代器，调用生成器函数 `numbers[Symbol.iterator]()` 返回一个可迭代的迭代器。

除了可以自己定义迭代器之外，还可以使用 JavaScript 内置的常用集合类型（`Array`、`Map`、`Set`、`String` 等^{注3}）迭代器，执行如下的操作：

```
// 使用 for-of 迭代一个迭代器
for (let a of numbers) {
    // 1, 2, 3, etc.
}

// 展开一个迭代器
let allNumbers = [...numbers] // number[]

// 析构一个迭代器
let [one, two, ...rest] = numbers // [number, number, number[]]
```

注 3：注意，`Object` 和 `Number` 不是迭代器。

同样，本书也不深入讨论迭代器。如果想进一步学习迭代器和异步迭代器，请访问 MDN (<https://mzl.la/2OAoyIo>)。



TSC 标志: downlevelIteration

如果把 TypeScript 编译成早于 ES2015 的 JavaScript 版本，可在 `tsconfig.json` 中设置 `downlevelIteration` 标志，启用自定义的迭代器。

如果特别在意打包体积，应该禁用 `downlevelIteration`，因为在陈旧的环境中，生成自定义迭代器的代码非常多。例如，上述 `numbers` 示例生成的代码将近 1 KB (gzip 压缩后)。

4.1.7 调用签名

目前，我们学习了如何注解函数的参数和返回值的类型。下面调转方向，说一说函数自身的完整类型。

再看一下本章前面定义的 `sum` 函数。以防你忘了，下面再次给出：

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

`sum` 的类型是什么呢？没错，由于 `sum` 是一个函数，那么它的类型是：

Function

你可能发现了，多数时候 `Function` 类型并不是我们想要的最终结果。我们知道，`object` 能描述所有对象，类似地，`Function` 也可以表示所有函数，但是并不能体现函数的具体类型。

那么，`sum` 的类型还能怎么表示呢？`sum` 是一个接受两个 `number` 参数、返回一个 `number` 的函数。在 TypeScript 中，可以像下面这样表示该函数的类型：

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

这是 TypeScript 表示函数类型的句法，也称调用签名（或叫类型签名）。注意，调用签名的句法与箭头函数十分相似，这是有意为之的。如果把函数做为参数传给另一个函数，或者作为其他函数的返回值，就要使用这样的句法注解类型。



a 和 b 这两个参数名称只是一种表意手段，不影响该类型函数的可赋值性。

函数的调用签名只包含类型层面的代码，即只有类型，没有值。因此，函数的调用签名可以表示参数的类型、this 的类型（见 4.1.4 节）、返回值的类型、剩余参数的类型和可选参数的类型，但是无法表示默认值（因为默认值是值，不是类型）。调用签名没有函数的定义体，无法推导出返回类型，所以必须显式注解。

类型层面和值层面代码

讨论静态类型语言时，人们常使用“类型层面”和“值层面”两个术语，所以我们有必要弄清具体含义。

在本书中，“类型层面代码”指只有类型和类型运算符的代码。而其他的都是“值层面代码”。一个简单的判断标准是，如果是有效的 JavaScript 代码，就是值层面代码；如果是有效的 TypeScript 代码，但不是有效的 JavaScript 代码，那就是类型层面代码。^{注4}

防止你未理解下面举个例子。该例中的加粗部分是类型层面代码，其余的则是值层面代码。

```
function area(radius: number): number | null {  
    if (radius < 0) {  
        return null  
    }  
}
```

注 4：这个判断标准不适用于枚举和命名空间。枚举既生成类型也生成值，而命名空间只存在于值层面。详见附录 C。

```
    return Math.PI * (radius ** 2)
}

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

加粗的类型层面代码是类型注解和并集类型运算符 |，其余的都是值层面代码。

目前，本章举了一些函数示例，下面从中挑选几个，使用单独的调用签名表示函数的类型，然后绑定给类型别名：

```
// 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
```

看懂了吗？函数的调用签名与具体实现十分相似。这是有意为之的，TypeScript 的设计人员这么做是为了让调用签名更易于理解。

下面我们来分析一下调用签名与其实现之间的关系。假如给你一个调用签名，如何据此声明函数呢？只需把调用签名和实现签名的函数表达式合在一起。下面举个例子，使用新学的签名重写 Log：

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

let log: Log = (①
  message, ②
  userId = 'Not signed in' ③
) => { ④
  let time = new Date().toISOString()
  console.log(time, message, userId)
}
```

- ❶ 声明一个函数表达式 `log`, 显式注解其类型为 `Log`。
- ❷ 不必再次注解参数的类型, 因为在定义 `Log` 类型时已经注解了 `message` 的类型为 `string`。这里不用再次注解, TypeScript 能从 `Log` 中推导出来。
- ❸ 为 `userId` 设置一个默认值。`userId` 的类型可以从 `Log` 的签名中获取, 但是默认值却不得而知, 因为 `Log` 是类型, 不包含值。
- ❹ 无需再次注解返回类型, 因为在 `Log` 类型中已经声明为 `void`。

4.1.8 上下文类型推导

注意, 前一个示例是我们见过的第一次不用显式注解函数参数类型的情况。由于我们已经把 `log` 的类型声明为 `Log`, 所以 TypeScript 能从上下文中推导出 `message` 的类型为 `string`。这是 TypeScript 类型推导的一个强大特性, 称为上下文类型推导 (contextual typing)。

本章前面出现过一处使用上下文类型推导的情形: 回调函数。^{注5}

下面声明一个函数 `times`, 它调用 `n` 次回调 `f`, 每次把当前索引传给 `f`:

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

调用 `times` 时, 传给 `times` 的函数如果是在行内声明的, 无需显式注解函数的类型:

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

注5: 你可能没听说过“回调”这个术语, 其实它就是一个函数, 只是作为参数传给另一个函数。

TypeScript 能从上下文中推导出 `n` 是一个数字，因为在 `times` 的签名中，我们声明 `f` 的参数 `index` 是一个数字。TypeScript 足够聪明，能推导出 `n` 就是那个参数，那么该参数的类型必然是 `number`。

注意，如果 `f` 不是在行内声明的，TypeScript 则无法推导出它的类型：

```
function f(n) { // Error TS7006: Parameter 'n' implicitly has an 'any' type.  
    console.log(n)  
}  
  
times(f, 4)
```

4.1.9 函数类型重载

前一节使用的函数类型句法，即 `type Fn = (...): void`，其实是简写型调用签名。如果愿意，可以使用完整形式。仍以 `Log` 为例：

```
// 简写型调用签名  
type Log = (message: string, userId?: string) => void  
  
// 完整型调用签名  
type Log = {  
    (message: string, userId?: string): void  
}
```

这两种写法完全等效，只是使用的句法不同。

那么何时使用完整型调用签名，何时使用简写型调用签名呢？简单的情况下，例如 `Log` 函数，首选简写形式，但是较为复杂的函数，使用完整签名有更多好处。

首先是重载函数类型的情况。不过在此之前我们要明确“重载”到底是什么意思。

重载函数

有多个调用签名的函数。

在多数编程语言中，声明函数时一旦指定了特定的参数和返回类型，就只能使用相应的参数调用函数，而且返回值的类型始终如一。但是 JavaScript 的情况却非如此。因为 JavaScript 是一门动态语言，势必需要以多种方式调用一个函数的方法。不仅如此，而且有时输出的类型取决于输入的参数类型。

TypeScript 也支持动态重载函数声明，而且函数的输出类型取决于输入类型，这一切都得益于 TypeScript 的静态类型系统。你可能以为这个语言特性是理所当然的，但其实只有先进的类型系统才有。

我们可以使用重载的函数签名设计十分具有表现力的 API。假如我们要设计一个预定假期的 API，名为 `Reserve`。先从类型规划入手（这一次使用完整类型签名）：

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

然后实现 `Reserve`：

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

假如有个用户想预定到巴厘岛的旅行，调用 `reserve` API 时要传入开始日期 (`from`)、结束日期 (`to`) 和目的地 "Bali"。

我们可以修改一下，让这个 API 支持单程旅行：

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

```
(from: Date, destination: string): Reservation  
}
```

运行这段代码，你会发现在实现 `Reserve` 的地方 TypeScript 将报错（见图 4-3）：

```
let reserve: Reserve = (from, to, destination) => {  
    // ...  
}  
type Reserve = (...)  
(f: (...))  
(f: (...))  
[ts] Type '(from: any, to: any, destination: any) => void' is not assignable to type 'Reserve'.  
let reserve: Reserve  
let reserve: Reserve = (from, to, destination) => {  
    // ...  
}
```

图 4-3：缺少重载的另一个签名时提示 TypeError

这是由 TypeScript 的调用签名重载机制造成的。如果为函数 `f` 声明多个重载的签名，在调用方看来，`f` 的类型是各签名的并集。但是在实现 `f` 时，必须一次实现整个类型组合。实现 `f` 时，我们要自己设法声明组合后的调用签名，TypeScript 无法自动推导。对 `Reserve` 示例来说，我们可以像下面这样更新 `reserve` 函数：

```
type Reserve = {  
    (from: Date, to: Date, destination: string): Reservation  
    (from: Date, destination: string): Reservation  
} ①  
  
let reserve: Reserve = (  
    from: Date,  
    toOrDestination: Date | string,  
    destination?: string  
) => { ②  
    // ...  
}
```

- ① 声明两个重载的函数签名。
- ② 自己动手组合两个签名（即自行计算 Signature1 | Signature2 的结果），实现声明的签名。注意，组合后的签名对调用 reserve 的函数是不可见的。在使用方看来，Reserve 的签名是：

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

注意，类型声明中没有组合后的签名：

```
// 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 可以通过两种方式调用，因此实现 reserve 时要向 TypeScript 证明你检查过了调用方式：^{注 6}

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

注 6：详见 6.1.5 节。

让重载的签名具体一些

一般来说，声明重载的函数类型时，每个重载的签名（例如 `Reserve`）都必须可以赋值给实现的签名（例如 `reserve`）。但是，声明实现的签名时一般可能会更宽泛一些，保证所有重载的签名都可以赋值给实现的签名。例如，下述代码是可行的：

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

重载时，应该尽量让实现的签名具体一些，这样更容易实现函数。因此，在这个示例中，使用 `Date` 好过 `any`，使用 `Date | string` 并集好过 `any`。

为什么对特定的签名来说让类型的范围窄一些更易于实现函数呢？如果把参数的类型注解为 `any`，但是想传入 `Date` 类型的值，那么必须向 TypeScript 证明传入的值的确是日期：

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

然而，如果我们事先标明参数的类型为 `Date`，实现函数时就不用再做额外的检查：

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

浏览器 DOM API 有大量重载。例如，DOM API 中的 `createElement` 用于新建 HTML 元素，其参数为表示 HTML 标签的字符串，返回值为对应类型的 HTML 元素。TypeScript 内置了每个 HTML 元素的类型。例如：



- 表示 `<a>` 元素的 `HTMLAnchorElement`。
- 表示 `<canvas>` 元素的 `HTMLCanvasElement`。
- 表示 `<table>` 元素的 `HTMLTableElement`。

通过重载调用签名实现 `createElement` 最合适不过了。请思考一下你会怎样声明 `createElement` 的类型（先别往下读，自己尝试解答一下）。

答案揭晓：

```
type CreateElement = {
  (tag: 'a'): HTMLAnchorElement ①
  (tag: 'canvas'): HTMLCanvasElement
  (tag: 'table'): HTMLTableElement
  (tag: string): HTMLElement ②
}

let createElement: CreateElement = (tag: string): HTMLElement => { ③
  // ...
}
```

- ➊ 重载参数的类型，与字符串字面量类型匹配。
- ➋ 添加一个兜底情况：如果用户传入自定义的标签名，或者前沿实验性的标签名，TypeScript 还未内置对应的类型声明，返回一般性的 `HTMLElement`。TypeScript 按照声明的顺序解析重载，^{注7} 调用 `createElement` 时如果传入没有相应重载定义的字符串（例如 `createElement('foo')`），TypeScript 将回落到 `HTMLElement`。
- ➌ 注解实现的参数时，要把该参数在 `createElement` 重载签名中的所有类型组合在一起，得到 `'a' | 'canvas' | 'table' | string`。由于前三个字符串字面量类型是 `string` 的子类型，所以可以把最终结果简化为 `string`。

注 7： 多数情况下如此。TypeScript 先把字面量重载提到非字面量重载前面，然后才按顺序解析。但是，不要依赖这个特性，这样容易导致不了解这个特性的工程师不理解你写的重载。



本节所有的示例重载的都是函数表达式。但是，如果我们想重载函数声明呢？同样，TypeScript 也支持这么做，句法与函数声明相同。下面重写 createElement 重载：

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

具体使用哪种句法由你自己决定，而且取决于你重载的是何种函数（函数表达式还是函数声明）。

完整的类型签名并不只限于用来重载调用函数的方式，还可以描述函数的属性。由于 JavaScript 函数是可调用的对象，因此我们可以为函数赋予属性，例如：

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

这个函数给用户显示一个提醒，但是只显示一次。下面使用 TypeScript 描述 warnUser 的完整类型签名：

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

下面把 warnUser 改写成函数表达式，实现该签名：

```
let warnUser: WarnUser = (warning: string) => {  
    if (warnUser.wasCalled) {
```

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

注意，TypeScript 足够聪明，它能发现，尽管声明 `warnUser` 函数时没有给 `wasCalled` 赋值，但是随后就赋值了。

4.2 多态

目前，本书都在讲具体类型的用法和用途，以及使用具体类型的函数。什么是具体类型呢？巧的是，目前见到的每个类型都是具体类型：

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

使用具体类型的前提是明确知道需要什么类型，并且想确认传入的确实是那个类型。但是，有时事先并不知道需要什么类型，不想限制函数只能接受某个类型。

为了说明这种情况，下面举个例子：实现 `filter` 函数。这个函数迭代数组，筛选符合条件的元素。在 JavaScript 中，可以这样实现 `filter` 函数：

```
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) // 求值结果为 [1, 2]
```

从中可以提取出 filter 函数的完整类型签名，类型先用 unknown 代替：

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

下面我们尝试填入具体的类型，比如说 number：

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

这里，数组元素的类型可以为 number，不过 filter 函数的作用本应更一般，可以筛选数字数组、字符串数组、对象数组等。我们编写的签名可以处理数字数组，但是不能处理元素为其他类型的数组。下面通过重载，让 filter 函数也能处理字符串数组：

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

目前一切顺利。那么对数组呢？

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

乍一看这么做没什么问题，可是用着就会遇到麻烦。不信，请实现具有该签名（即 filter: Filter）的 filter 函数试试，你会得到如下错误：

```
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'.
```

现在，你应该能明白 TypeScript 为什么会抛出这个错误。我们告诉 TypeScript，传给 `filter` 的可能是数字数组、字符串数组或对象数组。这里传入的是对象数组，可是你还记得吗？`object` 无法描述对象的结构。因此，尝试访问数组中某个对象的属性时，TypeScript 抛出错误，毕竟我们没有指明该对象的具体结构。

这个问题要怎么解决呢？

如果以前用过支持泛型 (generic type) 的语言，你会突然眼前一亮，大声叫道：“该用泛型啊！”好消息是，你说的完全没错（坏消息是，一声大叫惊醒了邻居家的孩子）。

以防你没用过泛型，下面先给出定义，然后再以 `filter` 函数为例说明用法。

泛型参数

在类型层面施加约束的占位类型，也称多态
类型参数。

仍以 `filter` 函数为例，使用泛型参数 `T` 重写后得到的声明如下：

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

这样做的意思是，“filter 函数使用一个泛型参数 T，可是我们事先不知道具体是什么类型；TypeScript，调用 filter 函数时，如果你能推导出该参数的类型，那最好。” TypeScript 从传入的 array 中推导 T 的类型。调用 filter 时，TypeScript 推导出 T 的具体类型后，将把 T 出现的每一处替换为推导出的类型。T 就像是一个占位类型，类型检查器将根据上下文填充具体的类型。T 把 Filter 的类型参数化了，因此才称其为泛型参数。



每次都说“泛型参数”有点浪费口舌，因此人们经常简称“泛型”。这两种说法本书都有使用。

泛型参数使用奇怪的尖括号 `<>` 声明（你可以把尖括号理解为 type 关键字，只不过声明的是泛型）。尖括号的位置限定泛型的作用域（只有少数几个地方可以使用尖括号），TypeScript 将确保当前作用域中相同的泛型参数最终都绑定同一个具体类型。鉴于这个示例中尖括号的位置，TypeScript 将在调用 filter 函数时为泛型 T 绑定具体类型。而为 T 绑定哪一个具体类型，取决于调用 filter 函数时传入的参数。在一对尖括号中可以声明任意个以逗号分隔的泛型参数。



T 就是一个类型名称，如果愿意，可以使用任何其他名称，例如 A、Zebra 或 133t。按照惯例，人们经常使用单个大写字母，从 T 开始，依次使用 U、V、W 等。如果一行中声明的泛型较多，或者使用方式较为复杂，可以不遵守这个惯例，转而使用更具描述性的名称，例如 Value 或 WidgetType。

有些人不喜欢从 T 开始，而是从 A 开始。不同的语言社区有不同的习惯，比如函数式语言喜欢使用 A、B、C 等，因为这类用户喜欢数学证明中常见的希腊字母 α 、 β 和 γ ；面向对象语言倾向于使用表示“Type”的 T。这两种编程风格 TypeScript 都支持，但是最好使用后一种惯例。

我们知道，每次调用函数时都要重新绑定函数的参数。类似地，每次调用 `filter` 都会重新绑定 `T`：

```
type Filter = {  
    <T>(array: T[], f: (item: T) => boolean): T[]  
}  
  
let filter: Filter = (array, f) => // ...  
  
// (a) T 绑定为 number  
filter([1, 2, 3], _ => _ > 2)  
  
// (b) T 绑定为 string  
filter(['a', 'b'], _ => _ !== 'b')  
  
// (c) T 绑定为 {firstName: string}  
let names = [  
    {firstName: 'beth'},  
    {firstName: 'caitlyn'},  
    {firstName: 'xin'}  
]  
filter(names, _ => _.firstName.startsWith('b'))
```

TypeScript 根据传入的参数的类型推导泛型绑定的类型。下面分析一下 TypeScript 如何绑定 (a) 情况下 `T` 的类型：

1. 根据 `filter` 的类型签名，TypeScript 知道 `array` 中的元素为某种类型 `T`。
2. TypeScript 知道传入的数组是 `[1, 2, 3]`，因此 `T` 必定是 `number`。
3. TypeScript 遇到 `T`，便把它替换成 `number`。因此，参数 `f: (item: T) => boolean` 将变成 `f: (item: number) => boolean`，返回类型 `T[]` 将变成 `number[]`。
4. 经检查，TypeScript 确认这些类型都满足可赋值性，而且传入的 `f` 函数可赋值给刚推导出的签名。

泛型让函数的功能更具一般性，比接受具体类型的函数更强大。泛型可以理解为一种约束。我们知道，把函数的参数注解为 `n: number`，参数 `n` 的值就被约束为 `number` 类型。同样，泛型 `T` 把 `T` 所在位置的类型约束为 `T` 绑定的类型。



泛型也可在类型别名、类和接口中使用。本书将大量使用泛型，在介绍相关话题时再详细说明。

只要可能就应该使用泛型，这样写出的代码更具一般性，可重复使用，并且简单扼要。

4.2.1 什么时候绑定泛型

声明泛型的位置不仅限定泛型的作用域，还决定 TypeScript 什么时候为泛型绑定具体的类型。以之前的一个示例为例：

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

<T> 在调用签名中声明（位于签名的开始圆括号前面），TypeScript 将在调用 Filter 类型的函数时为 T 绑定具体类型。

而如果把 T 的作用域限定在类型别名 Filter 中，TypeScript 则要求在使用 Filter 时显式绑定类型：

```
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) =>  
    // ...
```

一般来说，TypeScript 在使用泛型时为泛型绑定具体类型：对函数来说，在调用函数时；对类来说，在实例化类时（见 5.7 节）；对类型别名和接口（见 5.4 节）来说，在使用别名和实现接口时。

4.2.2 可以在什么地方声明泛型

只要是 TypeScript 支持的声明调用签名的方式，都有办法在签名中加入泛型：

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

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

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

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

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

- ❶ 一个完整的调用签名，`T` 的作用域在单个签名中。鉴于此，TypeScript 将在调用 `filter` 类型的函数时为签名中的 `T` 绑定具体类型。每次调用 `filter` 将为 `T` 绑定独立的类型。
- ❷ 一个完整的调用签名，`T` 的作用域涵盖全部签名。由于 `T` 是 `Filter` 类型的一部分（而不属于某个具体的签名），因此 TypeScript 将在声明 `Filter` 类型的函数时绑定 `T`。
- ❸ 与❶类似，不过声明的不是完整调用签名，而是简写形式。
- ❹ 与❷类似，不过声明的不是完整调用签名，而是简写形式。

- ⑤ 一个具名函数调用签名，`T` 的作用域在签名中。TypeScript 将在调用 `filter` 时为 `T` 绑定具体类型，而且每次调用 `filter` 将为 `T` 绑定独立的类型。

再举个例子：编写 `map` 函数。`map` 的作用与 `filter` 基本一样，但是不从数组中删除元素，而是通过映射函数转换各个元素。先写出框架：

```
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  
}
```

放慢脚步，仔细想一想怎样让 `map` 函数更具一般性。我们知道，要把每一个 `unknown` 替换成某个类型，那么需要多少个泛型呢？如何声明这些泛型，又如何把作用域限定在 `map` 函数中呢？`array`、`f` 和返回值各是什么类型呢？

想出来了吗？想不出来没关系，但一定要试一试。你能想出来的，真的！

好吧，不啰嗦了。答案如下：

```
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  
}
```

我们需要两个泛型：表示输入数组中元素类型的 `T`，以及表述输出数组中元素类型的 `U`。这个函数接受的参数为一个 `T` 类型的数组和一个把 `T` 映射为 `U` 的函数。最后，返回一个 `U` 类型的数组。

标准库中的 filter 和 map 函数

我们定义的 filter 和 map 函数与 TypeScript 自带的版本几乎没有什么不同：

```
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[]
}
```

我们还未介绍接口，不过上述定义也不难理解，可以看出 filter 和 map 函数处理的是 T 类型的数组。这两个函数都接受一个函数 callbackfn，以及该函数中 this 的类型。

filter 使用的泛型 T，其作用域在整个 Array 接口中。map 也使用 T，而且还有一个作用域只在 map 函数中的泛型 U。这意味着，TypeScript 在创建函数时为 T 绑定具体类型，而每次在数组上调用 filter 和 map 时都共用这个具体类型。而每次调用 map，将为 U 绑定独立的具体类型，此外还能访问已经绑定类型的 T。

JavaScript 标准库中的很多函数都使用泛型，尤其是 Array 原型的方法。数组中元素的值可以为任何类型，我们可以把元素的类型命名为 T，然后表达一些操作，例如 “.push 接受的参数类型为 T” “.map 把 T 类型的数组映射到 U 类型的数组”。

4.2.3 泛型推导

多数情况下，TypeScript 能自动推导出泛型。例如调用前面编写的 map 函数，经 TypeScript 推导，T 的类型是 string，U 的类型是 boolean：

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {  
    // ...  
}  
  
map(  
    ['a', 'b', 'c'], // T 类型的数组  
    _ => _ === 'a' // 返回类型为 U 的函数  
)
```

不过，也可以显式注解泛型。显式注解泛型时，要么所有必须的泛型都注解，要么都不注解：

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

TypeScript 将检查推导出来的每个泛型是否可赋值给显式绑定的泛型，如果不可赋值，将报错：

```
// 没问题，因为 boolean 可赋值给 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'.  
)
```

TypeScript 根据传入泛型函数的参数推导泛型的具体类型，有时你会遇到这样的情况：

```
let promise = new Promise(resolve =>  
    resolve(45)  
)
```

```
promise.then(result => // 推导结果为 {}
  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 推导出来的 `result` 的类型为 `{}`？因为我们没有提供足够的信息，毕竟 TypeScript 只通过泛型函数的参数类型推导泛型的类型，所以 `T` 默认为 `{}`。

这个问题的修正方法是显式注解 `Promise` 的泛型参数：

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

4.2.4 泛型别名

本章前面所举的 `Filter` 示例已经涉及泛型别名。另外，你可能还记得前一章介绍的 `Array` 和 `ReadonlyArray` 类型（见“只读数组和元组”一节），这两个也是泛型别名。下面举个简单的例子，深入讨论如何在类型别名中使用泛型。

我们来定义一个 `MyEvent` 类型，描述 DOM 事件，例如 `click` 或 `mousedown`：

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

注意，在类型别名中只有这一个地方可以声明泛型，即紧随类型别名的名称之后、赋值运算符 (=) 之前。

`MyEvent` 的 `target` 属性指向触发事件的元素，比如一个 `<button />`、一个 `<div />` 等。例如，可以像下面这样描述一个按钮事件：

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

使用 `MyEvent` 这样的泛型时，必须显式绑定类型参数，TypeScript 无法自行推导：

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

我们可以使用 `MyEvent` 构建其他类型，比如说 `TimedEvent`。绑定 `TimedEvent` 中的泛型 `T` 时，TypeScript 同时还会把它绑定给 `MyEvent`：

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

泛型别名也可以在函数的签名中使用。TypeScript 为 `T` 绑定类型时，还会自动为 `MyEvent` 绑定：

```
function triggerEvent<T>(event: MyEvent<T>): void {  
  // ...  
}  
  
triggerEvent({ // T 是 Element | null  
  target: document.querySelector('#myButton'),  
  type: 'mouseover'  
})
```

下面逐步说明这里涉及的操作：

1. 调用 `triggerEvent` 时传入一个对象。
2. 根据函数的签名，TypeScript 认定传入的参数类型必为 `MyEvent<T>`。TypeScript 还发现定义 `MyEvent<T>` 时声明的类型为 `{target: T, type: string}`。
3. TypeScript 发现传给该对象 `target` 字段的值为 `document.querySelector('#myButton')`。这意味着，`T` 必定为 `document.`

- querySelector('#myButton') 的类型，即 Element | null。因此，T 现在绑定为 Element | null。
- TypeScript 检查全部代码，把 T 出现的每一处替换为 Element | null。
 - TypeScript 确认所有类型都满足可赋值性，确保代码是类型安全的。

4.2.5 受限的多态



本节以二叉树为例说明。如果你没用过二叉树，也不用担心。本节只涉及一些基本知识：

- 二叉树是一种数据结构。
- 二叉树由节点构成。
- 一个节点有一个值，最多可以指向两个子节点。
- 节点有两种类型：叶节点（没有子节点）和内节点（至少有一个子节点）。

有时，说“这个是泛型 T，那个也是泛型 T”还不够，我们还想表达“类型 U 至少应为 T”，即为 U 设一个上限。

为什么要这么做呢？假设我们在实现一个二叉树，把节点分为三类：

- 常规的 TreeNode。
- LeafNode，即没有子节点的 TreeNode。
- InnerNode，即有子节点的 TreeNode。

首先声明表示各种节点的类型：

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

```
type InnerNode = TreeNode & {  
    children: [TreeNode] | [TreeNode, TreeNode]  
}
```

这段代码的意思是，`TreeNode` 是一个对象，只有一个属性 `value`；`LeafNode` 类型具有 `TreeNode` 的所有属性，外加一个 `isLeaf` 属性，其值始终为 `true`；`InnerNode` 也具有 `TreeNode` 的全部属性，另外还有一个 `children` 属性，指向一个或两个子节点。

接下来，编写 `mapNode` 函数，映射传入的 `TreeNode` 的值，返回一个新的 `TreeNode`。我们希望像下面这样使用 `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
```

现在请停下来思考一下，如果想把 `TreeNode` 的子类型传给 `mapNode` 函数，而且仍返回该子类型，那么应该怎么做呢？传入 `LeafNode`，返回 `LeafNode`；传入 `InnerNode`，返回 `InnerNode`；传入 `TreeNode`，返回 `TreeNode`。继续阅读之前，请考虑一下你会怎么做。这个要求能实现吗？

答案如下：

```
function mapNode<T extends TreeNode>(  
    node: T, ②  
    f: (value: string) => string  
) : T { ③  
    return {  
        ...node,  
        value: f(node.value)  
    }  
}
```

- ① `mapNode` 函数定义了一个泛型参数 `T`。`T` 的上限为 `TreeNode`，即 `T` 可以是 `TreeNode`，也可以是 `TreeNode` 的子类型。
- ② `mapNode` 接受两个参数，第一个是类型为 `T` 的 `node`。由于在①中指明了 `node extends TreeNode`，如果传入 `TreeNode` 之外的类型，例如空对象 {}、`null` 或 `TreeNode` 数组，立即就能看到一条红色波浪线。`node` 要么是 `TreeNode` 类型，要么是 `TreeNode` 的子类型。
- ③ `mapNode` 的返回值类型为 `T`。注意，`T` 要么是 `TreeNode` 类型，要么是 `TreeNode` 的子类型。

为什么这样声明 `T` 呢？

- 如果只输入 `T`（没有 `extends TreeNode`），那么 `mapNode` 会抛出编译时错误，因为这样不能从 `T` 类型的 `node` 中安全读取 `node.value`（试想传入一个数字的情况）。
- 如果根本不用 `T`，把 `mapNode` 声明为 `(node: TreeNode, f: (value: string) => string) => TreeNode`，那么映射节点后将丢失信息：`a1`、`b1` 和 `c1` 都只是 `TreeNode`。

声明 `Textends TreeNode`，输入节点的类型（`TreeNode`、`LeafNode` 或 `InnerNode`）将得到保留，映射后类型也不变。

有多个约束的受限多态

在上面的示例中，我们只为 `T` 施加了一个类型约束，即 `T` 至少为 `TreeNode`。那么，如果需要多个类型约束呢？

方法是扩展多个约束的交集（`&`）：

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

function logPerimeter<①
  Shape extends HasSides & SidesHaveLength ②>
```

```
>(s: Shape): Shape { ❸
  console.log(s.numberOfSides * s.sideLength)
  return s
}

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

- ❶ logPerimeter 函数只接受一个参数，类型为 Shape。
- ❷ Shape 是一个泛型，同时扩展 HasSides 和 SidesHaveLength 类型。也就是说，Shape 至少要有一定长度的边。
- ❸ logPerimeter 返回值的类型与输入类型一样。

使用受限的多态模拟变长参数

借助受限的多态还可以模拟变长参数函数（可接受任意个参数的函数）。下面我们可以实现一版 JavaScript 内置的 call 函数（回顾一下，call 函数接受一个函数和不定量的参数，这些参数将传给第一个参数指定的函数），以此为例进行说明。^{注 8} 我们将像下面这样定义和使用 call 函数，unknown 类型稍后再替换：

```
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') // 求值结果为由 10 个 'a' 构成的数组
```

下面来替换 unknown。我们想表达的约束是：

注 8：简单起见，我们实现 call 函数时不考虑 this。

- `f` 函数接受一系列 `T` 类型的参数，返回某种类型 `R`。我们事先不知道 `f` 有多少个参数。
- `call` 的参数为 `f`，以及 `f` 接受的那些 `T` 类型的参数。同样，我们事先也不知道具体有多少个参数。
- `call` 返回的类型与 `f` 一样，也是 `R`。

因此，需要两个类型参数：`T`，即参数数组；`R`，任意类型的返回值。下面把类型填进去：

```
function call<T extends unknown[], R>( ①
  f: (...args: T) => R, ②
  ...args: T ③
): R { ④
  return f(...args)
}
```

为什么要这么做呢？下面一步步分析：

- ❶ `call` 是一个变长参数函数（提醒一下，变长参数函数是可接受任意个参数的函数），有两个类型参数：`T` 和 `R`。`T` 是 `unknown[]` 的子类型，即 `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` 也是变长参数函数，参数的类型与 `args` 一样，`args` 是什么类型，`f` 的参数就是什么类型。
- ❸ 除了函数 `f` 之外，`call` 还接受数量不定的额外参数 `...args`。`args` 是剩余参数，数量不定。`args` 的类型是 `T`，`T` 必须为某一种数组类型（假如忘记指定 `T` 扩展数组类型，TypeScript 将显示一条波浪线），因此 TypeScript 将根据具体传入的 `args` 把 `T` 推导为一种元组类型。
- ❹ `call` 返回类型为 `R` 的值（`R` 受 `f` 返回类型的限制）。

这样，调用 `call` 时，TypeScript 知道返回值具体为什么类型，而且如果传入的参数数量有误，TypeScript 将报错。

```
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.
```

TypeScript 在为剩余参数推导类型时利用了一项改进推导结果的技术，详见 6.4.1 节。

4.2.6 泛型默认类型

函数的参数可以指定默认值，类似地，泛型参数也可以指定默认类型。下面以 4.2.4 节的 `MyEvent` 类型为例。你可能还记得，这个类型用于描述 DOM 事件，具体声明如下：

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

新建事件时，我们要为 `MyEvent` 显式绑定一个泛型，表示触发事件的 HTML 元素的类型：

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

为了给事先不知道 `MyEvent` 将绑定何种元素的情况提供便利，我们可以为 `MyEvent` 的泛型参数指定一个默认类型：

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

此外，我们还可以利用前几节所学的知识，为 `T` 设置限制，确保 `T` 是一个 HTML 元素：

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

这样，我们可以轻易创建一个不限定为特定 HTML 元素类型的事件，而且在创建事件时不用自己动手把 MyEvent 的 T 泛型绑定为 HTMLElement：

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

注意，与函数的可选参数一样，有默认类型的泛型要放在没有默认类型的泛型后面：

```
// 正确  
type MyEvent2<  
    Type extends string,  
    Target extends HTMLElement = HTMLElement,  
> = {  
    target: Target  
    type: Type  
}  
  
// 错误  
type MyEvent3<  
    Target extends HTMLElement = HTMLElement,  
    Type extends string // Error TS2706: Required type parameters may  
> = { // not follow optional type parameters.  
    target: Target  
    type: Type  
}
```

4.3 类型驱动开发

强大的类型系统自有强大的功能。编写 TypeScript 时，往往你会发现自己“受类型的指引”。理所当然，我们称之为类型驱动开发（type-driven development）。

类型驱动开发

先草拟类型签名，然后填充值的编程风格。

静态类型系统的要义是约束表达式的值可以为什么类型。类型系统的表现力越强，提供的关于表达式中值的信息越多。使用表现力强的类型系统注解函数，通过函数的类型签名就能知晓关于函数的多数信息。

下面来看一下本章前面声明的 `map` 函数的类型签名：

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

即便以前没见过 `map` 函数，仅通过签名也能直观地看出 `map` 的作用：接受一个 `T` 类型的数组和一个把 `T` 映射为 `U` 的函数，返回一个 `U` 类型的数组。注意，我们甚至不用看函数的实现就能知晓这些信息。^{注9}

编写 TypeScript 程序时，先定义函数的类型签名，即“受类型的指引”，然后再具体实现。先在类型层面规划程序可以确保在着手实现之前程序的整体合理性。

你应该发现了，目前为止，我们都是反着做的，即受实现的指引，然后再推演类型。现在我们知道在 TypeScript 中如何编写函数和注解函数类型了，那么就可以换种模式，先规划类型，然后再填充细节。

4.4 小结

本章讨论了如何声明和调用函数、如何注解参数的类型，如何表达常用的 JavaScript 函数特性，包括默认参数、剩余参数、生成器函数，以及 TypeScript 中的迭代器。我们探讨了函数的调用签名与具体实现之间的区别，

注 9：有些编程语言（比如 Haskell 类语言 Idris）内置了约束求解程序，能根据你编写的签名自动实现函数定义体。

以及上下文类型推导和重载函数的几种方式。最后，深入说明了函数多态和类型别名，讲解了多态的用途、如何及在何处声明泛型、TypeScript 如何推导泛型，以及如何为泛型添加限制和默认值。本章末尾简单介绍了类型驱动开发，说明了如何利用刚学的函数类型知识使用这种编程风格。

4.5 练习题

1. TypeScript 能从函数的类型签名中推导出哪些部分的类型，参数、返回值，还是二者都可以？
2. JavaScript 的 `arguments` 对象是类型安全的吗？如果不是，我们可以采取什么措施？
3. 假如你想预定立即开始的旅行。更新本章前面重载的 `reserve` 函数（见 4.1.9 节），添加第三个调用签名。这个签名只有目的地，没有开始日期。更新 `reserve` 的实现，支持这个新增的签名。
4. （有难度）更新本章前面实现的 `call` 函数（使用受限的多态模拟变长参数），让它只支持第二个参数为字符串的函数。如果传入除此以外的函数，在编译时报错。
5. 实现一个类型安全的小型断言库 `is`。先草拟类型。实现之后，可以像下面这样使用：

```
// 字符串与字符串比较
is('string', 'otherstring') // false

// 布尔值与布尔值比较
is(true, false) // false

// 数字与数字比较
is(42, 42) // true

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

// (有难度) 可以传入任意个参数
is([1], [1, 2], [1, 2, 3]) // false
```

第 5 章

类和接口

用过面向对象编程语言的程序员对类肯定不陌生，这是必备技能。类是组织和规划代码的方式，是封装的基本单位。你将惊喜地发现，TypeScript 类大量借用了 C# 的相关理论，支持可见性修饰符、属性初始化语句、多态、装饰器和接口。不过，由于 TypeScript 类将编译成常规的 JavaScript 类，所以我们也能使用一些 JavaScript 惯用法，例如兼顾类型安全的混入（mixin）。

TypeScript 的某些类特性，例如属性初始化语句和装饰器，JavaScript 类也支持，^{注1} 因此能生成运行时代码。其他特性（例如可见性修饰符、接口和泛型）是 TypeScript 专属的特性，只存在于编译时，把应用编译为 JavaScript 后不生成任何代码。

本章带领你通过实例学习 TypeScript 类的用法，让你直观地了解 TypeScript 的面向对象语言特性，掌握类的使用方法和缘由。在阅读的过程中，请自己动手在代码编辑器中输入书中给出的代码。

5.1 类和继承

我们将制作一个国际象棋引擎。这个引擎为国际象棋游戏建模，提供一个 API 供两个玩家交替走棋。

注 1：或者已经提上日程，即将支持。

首先草拟类型：

```
// 表示一次国际象棋游戏  
class Game {}
```

```
// 表示一个国际象棋棋子  
class Piece {}
```

```
// 一个棋子的一组坐标  
class Position {}
```

棋子分为六类：

```
// ...  
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，位置使用（字母，数字）坐标对表示；字母沿 x 轴从左至右展开，数字沿 y 轴从下到上展开（见图 5-1）。

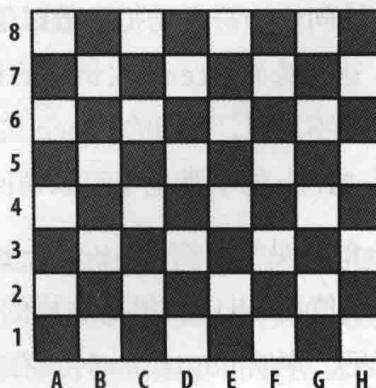


图 5-1：国际象棋中标准的代数标记法：A~H（沿 x 轴），称为“竖线”；1~8（沿 y 轴倒排），称为“横线”

下面为 Piece 类添加颜色和位置：

```

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

class Position {
    constructor(
        private file: File, ②
        private rank: Rank
    ) {}
}

class Piece {
    protected position: Position ③
    constructor(
        private readonly color: Color, ④
        file: File,
        rank: Rank
    ) {
        this.position = new Position(file, rank)
    }
}

```

- ① 由于颜色、横线和竖线相对较少，因此我们可以自己动手把可能的值使用类型字面量列举出来。这样做限定了类型的范围，只能是全部字符串和全部数字中的某些特定字符串和数字，进一步提升了安全性。
- ② 构造方法中的 `private` 访问修饰符自动把参数赋值给 `this` (`this.file` 等)，并把可见性设为私有，这意味着 `Piece` 实例中的代码可以读取和写入，但是 `Piece` 实例外部的代码不可以。不同的 `Piece` 实例访问各自的私有成员；其他类的实例，即便是 `Piece` 的子类也不可以访问私有成员。
- ③ 把实例变量 `position` 的可见性声明为 `protected`。与 `private` 类似，`protected` 也把属性赋值给 `this`，但是这样的属性对 `Piece` 的实例和 `Piece` 子类的实例都可见。声明 `position` 时没有为其赋值，因此在 `Piece` 的构造方法中要赋值。如果未在构造方法中赋值，TypeScript 将提醒我们变量没有明确赋值。也就是说，我们声明的类型是 `T`，但事实上是 `T | undefined`，这是因为我们没有在属性初始化语句或构造方法中为其赋值。如此一来，我们要更新属性的签名，指明其值不一定是一个 `Position` 实例，还有可能是 `undefined`。

- ④ new Piece 接受三个参数：color、file 和 rank。我们为 color 指定了两个修饰符：一个是 private，把它赋值给 this，并确保只能由 Piece 的实例访问；另一个是 readonly，指明在初始赋值后，这个属性只能读取，不能再赋其他值。



TSC 标志：strictNullChecks 和 strictPropertyInitialization

如果想检查类的实例变量有没有明确赋值，在 `tsconfig.json` 中启用 `strictNullChecks` 和 `strictPropertyInitialization`。如果已经启用 `strict` 标志，那就不用再设置这两个标志了。

TypeScript 类中的属性和方法支持三个访问修饰符：

`public`

任何地方都可访问。这是默认的访问级别。

`protected`

可由当前类及其子类的实例访问。

`private`

只可由当前类的实例访问。

访问修饰符的作用是不让类暴露过多实现细节，而是只开放规范的 API，供外部使用。

我们定义了一个 Piece 类，但是并不希望用户直接实例化 Piece，而是在此基础上扩展，定义 Queen、Bishop 等类，实例化这些子类。为此，我们可以通过类型系统做出限制，具体方法是使用 `abstract` 关键字：

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

现在，如果尝试直接实例化 Piece，TypeScript 将报错：

```
new Piece('White', 'E', 1) // Error TS2511: Cannot create an instance
                           // of an abstract class.
```

abstract 关键字表明，我们不能直接初始化该类，但是并不阻止我们在类中定义方法：

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

现在，Piece 类包含以下信息：

- 告诉子类，子类必须实现一个名为 canMoveTo 的方法，而且要兼容指定的签名。如果扩展 Piece 的类忘记实现抽象的 canMoveTo 方法，编译时将报类型错误。注意，实现抽象类时也要实现抽象方法。
- Piece 类为 moveTo 方法提供了默认实现（如果子类愿意，也可以覆盖默认实现）。我们没有为 moveTo 方法设置访问修饰符，因此默认为 public，所以其他代码可读也可写。

下面更新 King 类的定义，实现 canMoveTo 方法，满足最新的需求。为了便于计算两个棋子之间的距离，我们还将实现 distanceFrom 方法。

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

开始新游戏时，我们想自动创建一个棋盘和一些棋子：

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

  private static makePieces() {
    return [
      // 王
      new King('White', 'E', 1),
      new King('Black', 'E', 8),
      // 后
      new Queen('White', 'D', 1),
      new Queen('Black', 'D', 8),
      // 象
      new Bishop('White', 'C', 1),
      new Bishop('White', 'F', 1),
      new Bishop('Black', 'C', 8),
      new Bishop('Black', 'F', 8),
      // ...
    ]
  }
}
```

由于我们对 Rank 和 File 类型做出了严格的限制，如果输入别的字母（如 'J'）或范围外的数字（如 12），TypeScript 将报编译时错误（见图 5-2）。

```
73     return [
74
75     // kings
76     new King('White', 'J', 12),
77     new King('Black', 'J', 12),
78 ]
```

图 5-2：TypeScript 协助我们保证横线和竖线的有效性

以上内容足够让你了解 TypeScript 类的用法了，笔者没有深入一些细节，例如马如何吃掉其他棋子、象如何移动等。如果你想挑战自己，可以根据目前所学的知识，尝试实现国际象棋游戏余下的功能。

总结一下：

- 类使用 `class` 关键字声明。扩展类时使用 `extends` 关键字。
- 类可以是具体的，也可以是抽象的（`abstract`）。抽象类可以有抽象方法和抽象属性。
- 方法的可见性可以是 `private`、`protected` 或 `public`（默认）。方法分实例方法和静态方法两种。
- 类可以有实例属性，可见性也可以是 `private`、`protected` 或 `public`（默认）。实例属性可在构造方法的参数中声明，也可通过属性初始化语句声明。
- 声明实例属性时可以使用 `readonly` 把属性标记为只读。

5.2 super

与 JavaScript一样，TypeScript 也支持 `super` 调用。如果子类覆盖父类中定义的方法（假如 `Queen` 和 `Piece` 都实现了 `take` 方法），在子类中可以使用 `super` 调用父类中的同名方法（例如 `super.take`）。`super` 有两种调用方式：

- 方法调用，例如 `super.take`。
- 构造方法调用。此时使用特殊的形式 `super()`，而且只能在构造方法中调用。如果子类有构造方法，在子类的构造方法中必须调用 `super()`，把父子关

系连接起来（别担心，如果你忘记了，TypeScript会提醒你，这就像一个超现代的机器大象）。

注意，使用 `super` 只能访问父类的方法，不能访问父类的属性。

5.3 以 `this` 为返回类型

`this` 可以用作值，此外还能用作类型（像 4.1.4 节那样）。对类来说，`this` 类型还可用于注解方法的返回类型。

举个例子：实现 ES6 中 `Set` 数据结构的简化版。我们将为这个数据结构实现两个操作：把数字添加到集合中，以及检查指定的数字是否在集合中。`Set` 结构的用法如下：

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

下面定义 `Set` 类，先从 `has` 方法开始：

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

那么 `add` 方法呢？调用 `add` 得到的是一个 `Set` 实例。该方法的返回类型可以这样注解：

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

这样做是可以，但是如果我们要想定义 Set 的子类呢？

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

当然，Set 类中的 add 方法依然返回一个 Set 实例，但是在子类中，要替换成 MutableSet：

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

扩展其他类时，要把返回 this 的每个方法的签名覆盖掉，就显得十分麻烦。如果只是为了让类型检查器满意，这样做就失去了继承基类的意义。

正确的做法是使用 this 注解返回类型，把相关工作交给 TypeScript：

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

如此一来，我们可以把 MutableSet 中覆盖的 add 方法删除，因为在 Set 中，this 指向一个 Set 实例，而在 MutableSet 中，this 指向一个 MutableSet 实例。

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

```
    }
}
```

对链式 API（见 5.11.2 节）来说，这是一个特别便利的特性。

5.4 接口

类经常当做接口使用。

与类型别名相似，接口是一种命名类型的方式，这样就不用在行内定义了。类型别名和接口算是同一概念的两种句法（就像函数表达式和函数声明之间的关系），不过二者之间还是有一些细微差别。先看二者的共同点。以下述类型别名为例：

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

这个类型别名可以重写为下述接口：

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

在使用 `Sushi` 类型别名的地方都能使用 `Sushi` 接口。两个声明都定义结构，而且二者可以相互赋值（其实，二者完全一样）。

把类型组合在一起时，更为有趣。下面为 `Sushi` 之外的另一个食物建模：

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

除 Sushi 和 Cake 以外，很多食物都含热量，而且美味可口。下面单独声明一个 Food 类型，然后在此基础上重新定义两种食物：

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

同样，还可以使用接口定义，而且结果几乎一样：

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

 接口不一定扩展其他接口。其实，接口可以扩展任何结构：对象类型、类或其他接口。

类型和接口之间有什么区别呢？有三个细微的差别。

第一，类型别名更为通用，右边可以是任何类型，包括类型表达式（类型，外加 & 或 | 等类型运算符）；而在接口声明中，右边必须为结构。例如，下述类型别名不能使用接口重写：

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

第二个区别是，扩展接口时，TypeScript 将检查扩展的接口是否可赋值给被扩展的接口。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 // Error TS2430: Interface 'B' incorrectly extends  
} // interface 'A'. Type 'number' is not assignable  
// to type 'string'.
```

而使用交集类型时则不会出现这种问题。如果把前例中的接口换成类型别名，把 `extends` 换成交集运算符（`&`），TypeScript 将尽自己所能，把扩展和被扩展的类型组合在一起，最终的结果是重载 `bad` 的签名，而不会抛出编译时错误（不信你可以自己在代码编辑器中试试）。

建模对象类型的继承时，TypeScript 对接口所做的可赋值性检查是捕获错误的有力工具。

第三个区别是，同一作用域中的多个同名接口将自动合并；同一作用域中的多个同名类型别名将导致编译时错误。这个特性称为声明合并。

5.4.1 声明合并

声明合并指的是 TypeScript 自动把多个同名声明组合在一起。介绍枚举时讲过这个特性（见 3.2.13 节），讨论命名空间声明时还会再讲（见 10.3 节）。本节简单介绍接口的声明合并。如果想深入了解，请翻到 10.4 节。

例如，倘若你声明了两个名为 `User` 的接口，TypeScript 将自动把二者组合成一个接口：

```
// User 只有一个字段，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
}
```

而使用类型别名重写的话，将得到下述错误：

```
type User = { // Error TS2300: Duplicate identifier 'User'.
    name: string
}

type User = { // Error TS2300: Duplicate identifier 'User'.
    age: number
}
```

注意，两个接口不能有冲突。如果在一个接口中某个属性的类型为 T，而在另一个接口中该属性的类型为 U，由于 T 和 U 不是同一种类型，TypeScript 将报错：

```
interface User {
    age: string
}

interface User {
    age: number // Error TS2717: Subsequent property declarations must have
} // the same type. Property 'age' must be of type 'string',
   // but here has type 'number'.
```

如果接口中声明了泛型（见 5.7 节），那么两个接口中要使用完全相同的方式声明泛型（名称一样还不行），这样才能合并接口。

```
interface User<Age extends number> { // Error TS2428: All declarations of 'User'
    age: Age // must have identical type parameters.
}
```

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

有趣的是，TypeScript 很少这么做，但是在这里，TypeScript 不仅检查两个类型满不满足可赋值性，还会确认二者是否完全一致。

5.4.2 实现

声明类时，可以使用 `implements` 关键字指明该类满足某个接口。与其他显式类型注解一样，这是为类添加类型层面约束的一种便利方式。这么做能尽量保证类在实现上的正确性，防止错误出现在下游，不知具体原因。这也是实现常用的设计模式（例如适配器、工厂和策略）的一种常见方法（本章末尾有一些示例）。

下面举个例子：

```
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，在此基础上还可以实现其他方法和属性。

接口可以声明实例属性，但是不能带有可见性修饰符（`private`、`protected` 和 `public`），也不能使用 `static` 关键字。另外，像对象类型一样（见第 3 章），可以使用 `readonly` 把实例属性标记为只读：

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

一个类不限于只能实现一个接口，而是想实现多少都可以：

```
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')  
    }  
}
```

这些特性在类型上都是彻底安全的。如果忘记实现某个方法或属性，或者实现方式有误，TypeScript 会提醒你（见图 5-3）。

```
10 [ts]
11 Class 'Cat' incorrectly implements interface
12 'Feline'.
13     Property 'meow' is missing in type 'Cat' bu
14 t required in type 'Feline'. [2420]
15 • index.tsx(8, 3): 'meow' is declared here.
16
17 class Cat
18 class Cat implements Animal, Feline {
19     name = 'Whiskers'
20     eat(food: string) {
21         console.info('Ate some', food, '. Mmm!')
22     }
23     sleep(hours: number) {
24         console.info('Slept for', hours, 'hours')
25     }
26 }
```

图 5-3：忘记实现一个必要的方法，TypeScript 抛出错误

5.4.3 实现接口还是扩展抽象类

实现接口其实与扩展抽象类差不多。区别是，接口更通用、更轻量，而抽象类的作用更具体、功能更丰富。

接口是对结构建模的方式。在值层面可表示对象、数组、函数、类或类的实例。接口不生成 JavaScript 代码，只存在于编译时。

抽象类只能对类建模，而且生成运行时代码，即 JavaScript 类。抽象类可以有构造方法，可以提供默认实现，还能为属性和方法设置访问修饰符。这些在接口中都做不到。

具体使用哪个，取决于实际用途。如果多个类共用同一个实现，使用抽象类。如果需要一种轻量的方式表示“这个类是 T 型”，使用接口。

5.5 类是结构化类型

与 TypeScript 中的其他类型一样，TypeScript 根据结构比较类，与类的名称无关。类与其他类型是否兼容，要看结构；如果常规的对象定义了同样的属性或方法，也与类兼容。从 C#、Java、Scala 和其他多数名义类型编程语言转过来的程序员，一定要记住这一点。这意味着，如果一个函数接受 Zebra 实例，而我们传入一个 Poodle 实例，TypeScript 可能并不介意：

```
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
```

你认识的系统发育学家会说，斑马肯定不是贵妇犬，但是 TypeScript 并不介意。只要 Poodle 可赋值给 Zebra，TypeScript 就不报错，因为在该函数看来，二者是可互用的，毕竟两个类都实现了 `.trot` 方法。如果你使用的是名义类型语言，上述代码将报错，但是 TypeScript 是彻底的结构化类型语言，因此这段代码完全有效。

然而，如果类中有使用 `private` 或 `protected` 修饰的字段，情况就不一样了。检查一个结构是否可赋值给一个类时，如果类中有 `private` 或 `protected` 字段，而且结构不是类或其子类的实例，那么结构就不可赋值给类：

```
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}'.
```

5.6 类既声明值也声明类型

在 TypeScript 中，多数时候，表达的要么是值要么是类型：

```
// 值  
let a = 1999  
function b() {}  
  
// 类型  
type a = number  
interface b {  
    (): void  
}
```

在 TypeScript 中，类型和值位于不同的命名空间中。根据场合，TypeScript 知道你要使用的是类型还是值（上例中的 a 或 b）：

```
// ...  
if (a + 1 > 3) //... // TypeScript 从上下文中推导出你指的是值 a  
let x: a = 3      // TypeScript 从上下文中推导出你指的是类型 a
```

这种根据上下文进行解析的特性十分有用，可以做一些很酷的事情，例如实现伴生类型（companion type，见 6.3.4 节）。

类和枚举比较特殊，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 ①  
= new C ②  
  
enum E {F, G}  
let e: E ③  
= E.F ④
```

① 这里，C 指 C 类的实例类型。

② 这里，C 指值 C。

③ 这里，E 指 E 枚举的类型。

④ 这里，E 指值 E。

使用类时，我们需要一种方式表达“这个变量应是这个类的实例”，枚举同样如此（“这个变量应是这个枚举的一个成员”）。由于类和枚举在类型层面生成类型，所以我们可以轻易表达这种“是什么”关系。^{注2}

此外，我们还需要一种在运行时表示类的方式，这样才能使用 new 实例化类、在类上调用静态方法、做元编程、使用 instanceof 操作，因此类还需要生成值。

在上述示例中，C 指 C 类的一个实例。那要怎么表示 C 类自身的类型呢？使用 typeof 关键字（TypeScript 提供的类型运算符，作用类似于 JavaScript 中值层面的 typeof，不过操作的是类型）。

下面声明一个 StringDatabase 类，实现世界上最简单的数据库：

```
type State = {  
    [key: string]: string  
}  
  
class StringDatabase {  
    state: State = {}  
    get(key: string): string | null {
```

注 2：当然，因为 TypeScript 采用的是结构化类型，所以对类来说，更准确地关系应该是“看起来像”，即与一个类结构相同的类型便可赋值给该类。

```
    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
  }
}
```

这个类声明生成的类型是什么呢？是实例类型 `StringDatabase`：

```
interface StringDatabase {
  state: State
  get(key: string): string | null
  set(key: string, value: string): void
}
```

以及构造方法类型 `typeof StringDatabase`：

```
interface StringDatabaseConstructor {
  new(): StringDatabase
  from(state: State): StringDatabase
}
```

即，`StringDatabaseConstructor` 只有一个方法 `.from`，使用 `new` 运算符操作这个构造方法得到一个 `StringDatabase` 实例。这两个接口组合在一起对类的构造方法和实例进行建模。

`new()` 那一行称为构造方法签名，TypeScript 通过这种方式表示指定的类型可以使用 `new` 运算符实例化。鉴于 TypeScript 采用的是结构化类型，这是描述类的最佳方式，即可以通过 `new` 运算符实例化的是类。

这个示例中的构造方法不接受任何参数，不过也可以声明接受参数的构造方法。例如，我们可以更新 `StringDatabase`，让它接受一个可选的初始化状态：

```
class StringDatabase {  
    constructor(public state: State = {}) {}  
    // ...  
}
```

现在，`StringDatabase` 构造方法的签名如下：

```
interface StringDatabaseConstructor {  
    new(state?: State): StringDatabase  
    from(state: State): StringDatabase  
}
```

综上，类声明不仅在值层面和类型层面生成相关内容，而且在类型层面生成两部分内容：一部分表示类的实例，另一部分表示类的构造方法（通过类型运算符 `typeof` 获取）。

5.7 多态

与函数和类型一样，类和接口对泛型参数也有深层支持，包括默认类型和限制。泛型的作用域可以放在整个类或接口中，也可放在特定的方法中：

```
class MyMap<K, V> { ①  
    constructor(initialKey: K, initialValue: V) { ②  
        // ...  
    }  
    get(key: K): V { ③  
        // ...  
    }  
    set(key: K, value: V): void {  
        // ...  
    }  
    merge<K1, V1>(map: MyMap<K1, V1>): MyMap<K | K1, V | V1> { ④  
        // ...  
    }  
    static of<K, V>(k: K, v: V): MyMap<K, V> { ⑤  
        // ...  
    }  
}
```

- ① 声明类时绑定作用域为整个类的泛型。K 和 V 在 MyMap 的每个实例方法和实例属性中都可用。
- ② 注意，在构造方法中不能声明泛型。应该在类声明中声明泛型。
- ③ 在类内部，任何地方都能使用作用域为整个类的泛型。
- ④ 实例方法可以访问类一级的泛型，而且自己也可以声明泛型。`.merge` 方法使用了类一级的泛型 K 和 V，而且自己还声明了两个泛型：K1 和 V1。
- ⑤ 静态方法不能访问类的泛型，这就像在值层面不能访问类的实例变量一样。`of` 不能访问①中声明的 K 和 V，不过该方法自己声明了泛型 K 和 V。

接口也可以绑定泛型：

```
interface MyMap<K, V> {  
    get(key: K): V  
    set(key: K, value: V): void  
}
```

与函数一样，我们可以显式为泛型绑定具体类型，也可以让 TypeScript 自动推导：

```
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)
```

5.8 混入

JavaScript 和 TypeScript 都没有 trait 或 mixin 关键字，不过自己实现起来也不难。这两个特性都用于模拟多重继承（一个类扩展两个以上的类），可做面向角色编程。这是一种编程风格，在这种风格中，我们不表述“这是一个 Shape”，而是描述事物的属性，表述“这个东西可以度量”或者“这个东西有四条边”；我们不再关心“是什么”关系，转而描述“能做什么”和“有什么”关系。

下面我们自己动手实现混入。

混入这种模式把行为和属性混合到类中。按照惯例，混入有以下特性：

- 可以有状态（即实例属性）。
- 只能提供具体方法（与抽象方法相反）。
- 可以有构造方法，调用的顺序与混入类的顺序一致。

TypeScript 没有内置混入的概念，不过我们可以自己动手轻易实现。下面我们将设计一个调试 TypeScript 类的库，以此为例进行说明。我们把这个库命名为 `EZDebug`，它的作用是输出关于类的一些信息，方便在运行时审查类。`EZDebug` 的用法如下：

```
class User {  
    // ...  
}  
  
User.debug() // 求值结果为 'User({ "id": 3, "name": "Emma Gluzman" })'
```

通过这个标准的 `.debug` 接口，用户便可以调试任何类。下面开始实现。我们将通过一个混入实现这个接口，将其命名为 `withEZDebug`。混入其实就是一个函数，只不过这个函数接受一个类构造方法，而且返回一个类构造方法。这个混入的声明如下：

```
type ClassConstructor = new(...args: any[]) => {} ①  
  
function withEZDebug<C extends ClassConstructor>(Class: C) { ②  
    return class extends Class { ③  
        constructor(...args: any[]) { ④  
            super(...args) ⑤  
        }  
    }  
}
```

① 先声明类型 `ClassConstructor`，表示任意构造方法。由于 TypeScript 完全采用结构化类型，因此可以使用 `new` 运算符操作的就是构造方法。我们

不知道这个构造方法接受什么类型的参数，所以指明它可以接受任意个任意类型的参数。^{注3}

- ❷ 声明 withEZDebug 混入，只接受一个类型参数，C。C 至少是类构造方法，使用 extends 子句表示这一要求。我们让 TypeScript 推导 withEZDebug 的返回类型，结果是 C 与该匿名类的交集。
- ❸ 由于混入是接受一个构造方法并返回一个构造方法的函数，所以这里返回一个匿名类构造方法。
- ❹ 类构造方法至少要接受传入的类接受的参数。但是请注意，由于我们事先不知道将传入什么类，所以要尽量放宽要求，允许传入任意个任意类型的参数，跟 ClassConstructor 一样。
- ❺ 最后，因为这个匿名类扩展自其他类，为了正确建立父子关系，别忘了调用 Class 的构造方法。

与常规的 JavaScript 类一样，如果构造方法中没有什么逻辑，可以省略第❻行和第❼行。在这个 withEZDebug 示例中，我们不打算在构造方法中放任何逻辑，因此可以省略那两行。

准备工作做好后，下面该实现调试功能了。调用 .debug 时，我们想输出类的构造方法名称和实例的值：

```
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) + ')'
        }
    }
}
```

注3：注意，在这方面 TypeScript 要求比较严格：构造方法类型的参数的类型必须为 any[]（不能是 void、unknown[] 等），这样才能扩展。

慢着！这里要调用 `.getDebugValue` 方法，可是我们怎样确保类实现了这个方法呢？继续往下阅读之前请思考一下。你能想出办法吗？

答案是，不接受常规的类，而是使用泛型确保传给 `withEZDebug` 的类定义了 `.getDebugValue` 方法：

```
type ClassConstructor<T> = new(...args: any[]) => T ①

function withEZDebug<C extends ClassConstructor<{
  getDebugValue(): object ②
}>>(Class: C) {
  // ...
}
```

- ① 为 `ClassConstructor` 添加一个泛型参数。
- ② 为 `ClassConstructor` 绑定一个结构类型，`C`，规定传给 `withEZDebug` 的构造方法至少定义了 `.getDebugValue` 方法。

这样就可以了！那么，怎样使用这个功能强大的调试工具呢？如下所示：

```
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() // 求值结果为 'User({id: 3, name: "Emma Gluzman"})'
```

不错吧？我们可以把任意多个混入混合到类中，为类增添更丰富的行为，而

且这一切在类型上都是安全的。混入有助于封装行为，是描述可重用行为的一种重要方式。^{注4}

5.9 装饰器

装饰器是 TypeScript 的一个实验特性，为类、类方法、属性和方法参数的元编程提供简洁的句法。其实，装饰器就是在装饰目标上调用函数的一种句法。



TSC 标志: experimentalDecorators

装饰器是一个实验特性，以后的 TypeScript 版本可能会做出不向后兼容的改动，甚至彻底删除。这个特性隐藏在一个 TSC 标志背后。如果你不在意，想使用这个特性，在 `tsconfig.json` 中设置 `"experimentalDecorators": true`，然后继续阅读本节。

为了让你对装饰器有个感性认识，下面举个例子：

```
@serializable
class APIPayload {
    getValue(): Payload {
        // ...
    }
}
```

`@serializable` 类装饰器对 `APIPayload` 类进行包装，可以返回一个新类，替代原类。不使用装饰器，可以像下面这样实现这种操作：

```
let APIPayload = serializable(class APIPayload {
    getValue(): Payload {
        // ...
    }
})
```

注 4：很多语言，比如 Scala、PHP、Kotlin 和 Rust，实现了精简版混入，称为性状（trait）。性状与混入类似，但是没有构造方法，也不支持实例属性。因此性状更易于使用，而且不会在多个性状访问性状与基类共用的状态时产生冲突。

对不同种类的装饰器, TypeScript 要求作用域中有那种装饰器指定名称的函数,而且该函数还要具有相应的签名(见表 5-1)。

表 5-1: 不同种类装饰器函数要具有的类型签名

装饰目标	具有的类型签名
类	(Constructor: {new(...any[]) => any}) => any
方法	(classPrototype: {}, methodName: string, descriptor: PropertyDescriptor) => any
静态方法	(Constructor: {new(...any[]) => any}, methodName: string, descriptor: PropertyDescriptor) => any
方法的参数	(classPrototype: {}, paramName: string, index: number) => void
静态方法的参数	(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
属性设值方法 / 读值方法	(classPrototype: {}, propertyName: string, descriptor: PropertyDescriptor) => any
静态属性设值方法 / 读值方法	(Constructor: {new(...any[]) => any}, propertyName: string, descriptor: PropertyDescriptor) => any

TypeScript 没有内置任何装饰器, 如果你想使用, 只能自己实现(或者从 NPM 中安装)。不同种类的装饰器(包括类装饰器、方法装饰器、属性装饰器和函数参数装饰器)都是常规的函数, 只不过要满足相应的特定签名。例如, 前面使用的 @serializable 装饰器可以像下面这样实现:

```
type ClassConstructor<T> = new(...args: any[]) => T ①

function serializable<
  T extends ClassConstructor<{
    getValue(): Payload ②
  }>
```

```
>(Constructor: T) { ③
  return class extends Constructor { ④
    serialize() {
      return this.getValue().toString()
    }
  }
}
```

- ① 记住，在 TypeScript 中，类的构造方法使用 `new()` 表示结构化类型。如果类的构造方法可被扩展（使用 `extends`），TypeScript 要求参数的类型为可展开的 `any`，即 `new(...any[])`。
- ② `@serializable` 可以装饰任何实现了 `.getValue` 方法，而且返回一个 `Payload` 的类的实例。
- ③ 类装饰器是一个接受单个参数（即目标类）的函数。如果装饰器函数返回一个类（上例就是这样），在运行时这个类将替代被装饰的类；否则，要返回原类。
- ④ 为了装饰目标类，我们返回一个扩展原类的类，增加 `.serialize` 方法。

调用 `.serialize` 有什么效果呢？

```
let payload = new APIPayload
let serialized = payload.serialize() // Error TS2339: Property 'serialize' does
                                    // not exist on type 'APIPayload'.
```

TypeScript 假定装饰器不改变装饰目标的结构，意即不增加或删除方法和属性。TypeScript 在编译时检查返回的类是否可赋值给传入的类，但是写作本书时，TypeScript 不记录你在装饰器中做了什么扩展。

在 TypeScript 的装饰器成为稳定特性之前，笔者建议不要使用，而是继续使用常规的函数：

```
let DecoratedAPIPayload = serializable(APIPayload)
let payload = new DecoratedAPIPayload
payload.serialize()           // string
```

本书不对装饰器做深入讨论。更多信息，参阅官方文档 (<http://bit.ly/2IDQdIU>)。

5.10 模拟 final 类

虽然 TypeScript 的类和方法不支持 `final` 关键字，但是我们可以轻易模拟。以防你没有多少面向对象语言的使用经验，说一下 `final` 关键字的作用：某些语言使用这个关键字把类标记为不可扩展，或者把方法标记为不可覆盖。

在 TypeScript 中，可以通过私有的构造方法模拟 `final` 类：

```
class MessageQueue {  
    private constructor(private messages: string[]) {}  
}
```

把 `constructor` 标记为 `private` 后，不能使用 `new` 运算符实例化类，也不能扩展类：

```
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.
```

除了禁止扩展类以外（这是我们的目的），私有的构造方法还禁止直接实例化类。但是，我们希望 `final` 类能实例化，禁止扩展就行了。那么，怎样保留第一个限制，而避免第二个限制呢？简单：

```
class MessageQueue {  
    private constructor(private messages: string[]) {}  
    static create(messages: string[]): MessageQueue {  
        return new MessageQueue(messages)  
    }  
}
```

这样稍微修改 MessageQueue 的 API 之后，编译时便会禁止扩展：

```
class BadQueue extends MessageQueue {} // Error TS2675: Cannot extend a class
                                         // 'MessageQueue'. Class constructor is
                                         // marked as private.

MessageQueue.create([]) // MessageQueue
```

5.11 设计模式

不自己动手实现一两个设计模式怎么算得上是讨论面向对象编程的一章呢？

5.11.1 工厂模式

工厂模式（factory pattern）是创建某种类型的对象的一种方式，这种方式把创建哪种具体对象留给创建该对象的工厂决定。

下面我们来构建一个造鞋工厂。首先定义 Shoe 类型，以及几种鞋：

```
type Shoe = {
  purpose: string
}

class BalletFlat implements Shoe {
  purpose = 'dancing'
}

class Boot implements Shoe {
  purpose = 'woodcutting'
}

class Sneaker implements Shoe {
  purpose = 'walking'
}
```

注意，这个示例使用的是 type，此外也可以使用 interface。

下面建造制鞋工厂：

```
let Shoe = {  
  create(type: 'balletFlat' | 'boot' | 'sneaker'): Shoe { ❶  
    switch (type) { ❷  
      case 'balletFlat': return new BalletFlat  
      case 'boot': return new Boot  
      case 'sneaker': return new Sneaker  
    }  
  }  
}
```

- ❶ 把 `type` 的值指定为一个并集类型有助于提升 `.create` 的类型安全，以免使用方在编译时传入一个无效的 `type`。
- ❷ 使用 `switch` 检查 `type` 的值，确保处理每一种 `Shoe`。

这个示例使用伴生对象模式（见 6.3.4 节）声明类型 `Shoe` 和同名的值 `Shoe`（还记得吗，TypeScript 把值和类型放在不同的命名空间中），以此表明值提供了操作类型的方法。若想使用这个工厂，只需调用 `.create`:

```
Shoe.create('boot') // Shoe
```

简单吧，这就是工厂模式。注意，我们可以更明确一些，在 `Shoe.create` 的类型签名中指明，传入 '`boot`' 得到一个 `Boot` 实例，传入 '`sneaker`' 得到一个 `Sneaker` 实例等，但是这样做破坏了工厂模式所做的抽象（即使用方不知道将得到什么具体的类，知道的只是该类满足特定的接口）。

5.11.2 建造者模式

建造者模式（builder pattern）把对象的建造方式与具体的实现方式区分开。如果你用过 jQuery，或者 ES6 的 `Map` 和 `Set` 等数据结构，对这种 API 风格应该不会陌生。下面举个例子：

```
new RequestBuilder()  
  .setURL('/users')  
  .setMethod('get')  
  .setData({firstName: 'Anna'})  
  .send()
```

`RequestBuilder` 要怎么实现呢？简单，先从一个空类开始：

```
class RequestBuilder {}
```

然后添加 `.setURL` 方法：

```
class RequestBuilder {  
    private url: string | null = null ①  
  
    setURL(url: string): this { ②  
        this.url = url  
        return this  
    }  
}
```

- ① 把用户设置的 URL 保存在私有实例变量 `url` 中，其初始值为 `null`。
- ② `setURL` 的返回类型是 `this`（见 5.3 节），即用户调用 `setURL` 的那个 `RequestBuilder` 实例。

接下来再添加其他几个方法：

```
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() {  
    // ...  
}  
}
```

整个 API 构建完毕。



传统的建造者模式不彻底安全：有可能在设定方法、URL 和数据之前调用 `.send`，导致运行时异常（这可是一种不好的异常）。这种模式的改进思路参见第 4 道练习题。

5.12 小结

至此，我们全面讨论了 TypeScript 类，讲了如何声明类、如何继承类和实现接口；如果把类标记为 `abstract`，禁止实例化；如何在类中声明静态字段和方法，如何在类中声明实例字段和方法；如何使用 `private`、`protected` 和 `public` 等可见性修饰符控制对字段和方法的访问；如何使用 `readonly` 修饰符把字段标记为不可写的。本章还介绍了如何安全使用 `this` 和 `super`，探讨了类同时具有值和类型的意义，讨论了类型别名与接口之间的区别、声明合并的基本概念和泛型在类中的使用。接下来，介绍了几种与类有关的高级模式：混入、装饰器，以及通过模拟手段实现的 `final` 类。本章最后使用类实现了两个常用的设计模式。

5.13 练习题

1. 类和接口之间有什么区别？
2. 把类的构造方法标记为 `private` 后，不能实例化或扩展类。那么把构造方法标记为 `protected` 呢？请你在代码编辑器中试一试，看能不能找出答案。
3. 在 5.11.1 节实现的基础上，牺牲一点抽象性，设法提升安全性。更新实现，让使用方在编译时知道，调用 `Shoe.create('boot')` 返回一个 `Boot` 实例，

调用 `Shoe.create('balletFlat')` 返回一个 `BalletFlat` 实例（而不是都返回一个 `Shoe` 实例）。提示：回想一下 4.1.9 节的内容。

4. (有难度) 思考如何设计对类型安全的建造者模式。在 5.11.2 节实现的基础上进行扩展，做到以下两点：

 - a. 在编译时确保不能在设置 URL 和请求方法之前调用 `.send`。如果要求用户以特定顺序调用各方法，是不是更容易满足（提示：不返回 `this`，应该返回什么）？
 - b. (更有难度) 在保证前一点的基础上，如果想让用户以任意顺序调用其他方法，应该如何修改设计呢（提示：使用哪个 TypeScript 特性可以在调用方法之后把方法的返回类型“添加”到 `this` 类型上）？

类型进阶

TypeScript 一流的类型系统支持强大的类型层面编程特性，就算资深的 Haskell 程序员对此也可能艳羡不已。现在我们知道，TypeScript 的类型系统不仅具有极强的表现力，易于使用，而且可通过简洁明了的方式声明类型约束和关系，并且多数时候能自动为我们推导。

我们需要这样一个具有表现力且不同寻常的类型系统，因为 JavaScript 十分动态。原型、动态绑定 `this`、函数重载和变化无常的对象要求有一个特性丰富的类型系统，而且要绑上一条串有各色类型运算符的腰带，让蝙蝠侠都自愧不如。

本章首先探讨 TypeScript 中的子类型、可赋值性、型变和类型拓宽，加深你对前几章内容的感性认识。然后，深入说明 TypeScript 基于控制流的类型检查特性，包括类型细化和全面性检查。接下来讨论类型层面的一些高级编程特性：“键入”和映射对象类型，使用条件类型，自定义类型防护措施，以及类型断言和明确赋值断言等救命稻草。最后，介绍一些高级模式，尽量提升类型的安全性：伴生对象模式，改善元组的类型推导，模拟名义类型，以及安全扩展原型的方式。

6.1 类型之间的关系

首先，深入讨论 TypeScript 中的类型关系。

6.1.1 子类型和超类型

3.1 节简单讲过可赋值性。我们已经了解了 TypeScript 提供的多数类型，现在可以深挖一些细节了。首当其冲，我们要弄清一个问题：子类型是什么？

子类型

给定两个类型 A 和 B，假设 B 是 A 的子类型，那么在需要 A 的地方都可以放心使用 B（见图 6-1）。

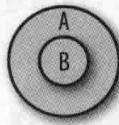


图 6-1：B 是 A 的子类型

再看一下第 3 章开头的图 3-1，从那幅图中可以看出 TypeScript 各个类型之间的关系。例如：

- Array 是 Object 的子类型。
- Tuple 是 Array 的子类型。
- 所有类型都是 any 的子类型。
- never 是所有类型的子类型。
- 如果 Bird 类扩展自 Animal 类，那么 Bird 是 Animal 的子类型。

根据前面给出的子类型定义，这意味着：

- 需要 Object 的地方都可以使用 Array。
- 需要 Array 的地方都可以使用 Tuple。
- 需要 any 的地方都可以使用 Object。
- never 可在任何地方使用。
- 需要 Animal 的地方都可以使用 Bird。

你可能猜到了，超类型正好与子类型相反。

超类型

给定两个类型 A 和 B，假设 B 是 A 的超类型，那么在需要 B 的地方都可以放心使用 A（见图 6-2）。

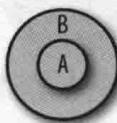


图 6-2：B 是 A 的超类型

同样，从图 3-1 中的流程图可以得到以下信息：

- Array 是 Tuple 的超类型。
- Object 是 Array 的超类型。
- any 是所有类型的超类型。
- never 不是任何类型的超类型。
- Animal 是 Bird 的超类型。

这正好与子类型相反，其他没什么好说的了。

6.1.2 型变

多数时候，很容易判断 A 类型是不是 B 类型的子类型。例如，对 `number`、`string` 等类型来说，只需对照图 3-1 中的流程图，或者自己分析（`number` 包含在并集类型 `number | string` 中，那么 `number` 必定是它的子类型）。

但是对参数化类型（泛型）和其他较为复杂的类型来说，情况不那么明晰。请看下面几种情况：

- 什么情况下 `Array<A>` 是 `Array` 的子类型？
- 什么情况下结构 A 是结构 B 的子类型？
- 什么情况下函数 `(a: A) => B` 是函数 `(c: C) => D` 的子类型？

如果一个类型中包含其他类型（即带有类型参数的类型，如 `Array<A>`；带有字段的结构，如 `{a: number}`；或者函数，如 `(a: A) => B`），使用上述规则很难判断谁是子类型。很多编程语言在判断复杂类型的子类型规则上有差异，几乎没有两门语言是一样的。

为了让下述规则便于理解，笔者将引入一套句法，以便使用简洁且准确的语言讨论类型。这套句法不是有效的 TypeScript 代码，只是为了让你我使用同一套语言讨论类型。别担心，这套句法比数学简单多了：

- `A <: B` 指“`A` 类型是 `B` 类型的子类型，或者为同种类型”。
- `A >: B` 指“`A` 类型是 `B` 类型的超类型，或者为同种类型”。

结构和数组型变

为了让你直观地理解为什么不同的编程语言在判断复杂类型的子类型规则上有差异，下面分析一个复杂类型的例子：结构。假如你的应用中有个描述用户的结构，你可能会使用两个类型表示用户：

```
// 从服务器中获取的现有用户
type ExistingUser = {
  id: number
```

```
    name: string
}
// 还未保存到服务器中的新用户
type NewUser = {
    name: string
}
```

假如我们给公司的一个实习生分配了一项任务：编写删除用户的代码。这个实习生是这样写的：

```
function deleteUser(user: {id?: number, name: string}) {
    delete user.id
}

let existingUser: ExistingUser = {
    id: 123456,
    name: 'Ima User'
}

deleteUser(existingUser)
```

`deleteUser` 接受一个对象，类型为 `{id?: number, name: string}`，我们传入的 `existingUser` 对象是 `{id: number, name: string}` 类型。注意，`id` 属性的类型 (`number`) 是预期类型 (`number | undefined`) 的子类型。因此，`{id: number, name: string}` 作为一个整体是 `{id?: number, name: string}` 的子类型，所以 TypeScript 不会报错。

你发现这里的安全问题了吗？不仔细还真看不出来：把 `ExistingUser` 类型的值传给 `deleteUser` 函数之后，TypeScript 不知道用户的 `id` 已经被删除，所以调用 `deleteUser(existingUser)` 把该属性删除之后再读取 `existingUser.id`，TypeScript 仍认为 `existingUser.id` 是 `number` 类型。

显然，在预期某个类型的超类型的地方使用该类型是不安全的。可是，为什么 TypeScript 不阻止我们呢？一般来说，TypeScript 在设计上不是只顾安全性，TypeScript 的类型系统尽量在捕获问题和易于使用上做到了平衡，让我们无需深入研究编程语言理论就能理解出错的原因。这个例子是特殊情况，由于破

坏性更新（例如删除一个属性）在实际中很少见，所以 TypeScript 放宽了要求，允许我们在预期某类型的超类型的地方使用那个类型。

那么反过来呢？能不能在预期某类型的子类型的地方使用那个类型呢？

下面我们来添加一个表示旧用户的类型，然后删除该类型的用户（假设你把类型添加到还未开始使用 TypeScript 之前同事编写的代码中）：

```
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'.
```

我们传入的结构中有一个属性的类型是预期类型的超类型，TypeScript 报错了。这是因为 `id` 的类型是 `string | number | undefined`，而 `deleteUser` 函数只处理了 `id` 为 `number | undefined` 类型的情况。

TypeScript 的行为是这样的：对预期的结构，还可以使用属性的类型 `<:` 预期类型的结构，但是不能传入属性的类型是预期类型的超类型的结构。在类型上，我们说 TypeScript 对结构（对象和类）的属性类型进行了协变（covariant）。也就是说，如果想保证 A 对象可赋值给 B 对象，那么 A 对象的每个属性都必须 `<: B` 对象的对应属性。

其实，协变只是型变的四种方式之一：

不变

只能是 T。

协变

可以是 `<:T`。

逆变

可以是 `>:T`。

双变

可以是 `<:T` 或 `>:T`。

在 TypeScript 中，每个复杂类型的成员都会进行协变，包括对象、类、数组和函数的返回类型。不过有个例外：函数的参数类型进行逆变。



不是所有编程语言都采用这种设计方式。在某些语言中，对象的属性类型“不变”，原因正如前面说过的，协变属性的类型可能导致不安全的行为。有些语言对可变对象和不可变对象有不同的规则（请你自己想一想原因）。有些语言，例如 Scala、Kotlin 和 Flow，甚至有显式指定对数据类型进行型变的句法。

设计 TypeScript 时，设计人员在易用性和安全性上做出了权衡。不允许型变对象的属性类型，安全性是更高了，但是会导致类型系统用起来很烦琐，而且会禁止实际上安全的操作（假如 `deleteUser` 没有删除 `id`，完全可以传入预期类型的超类型）。

函数型变

先看几个例子。

如果函数 A 的参数数量小于或等于函数 B 的参数数量，而且满足下述条件，那么函数 A 是函数 B 的子类型：

1. 函数 A 的 `this` 类型未指定，或者 `>: 函数 B 的 this 类型`。
2. 函数 A 的各个参数的类型 `>: 函数 B 的相应参数`。
3. 函数 A 的返回类型 `<: 函数 B 的返回类型`。

请多读几遍，确保自己理解了每一条的意思。你可能注意到了，如果函数 A 是函数 B 的子类型，那么函数 A 的 this 类型和参数的类型必定 >：函数 B 的 this 类型和参数的类型，而函数 A 的返回类型必定 <：函数 B 的返回类型。为什么方向会变呢？为什么不像对象、数组、并集类型等那样，每一部分（this 类型、参数的类型和返回类型）都是 <：呢？

为了回答这个问题，我们自己推导一下。首先，定义三个类型（清楚起见，我们将使用类，不过使用满足 $A <: B <: C$ 的其他类型也可以）：

```
class Animal {}  
class Bird extends Animal {  
    chirp() {}  
}  
class Crow extends Bird {  
    caw() {}  
}
```

在这个示例中，Crow 是 Bird 的子类型，Bird 又是 Animal 的子类型。即， $Crow <: Bird <: Animal$ 。

下面定义一个参数为 Bird 对象的函数，让这只鸟叫一声：

```
function chirp(bird: Bird): Bird {  
    bird.chirp()  
    return bird  
}
```

目前没什么问题。TypeScript 允许我们把什么传给 chirp 函数呢？

```
chirp(new Animal) // Error TS2345: Argument of type 'Animal' is not assignable  
                    // to parameter of type 'Bird'.  
chirp(new Bird)  
chirp(new Crow)
```

可以传入一个 Bird 实例（这就是 chirp 的参数 bird 的类型），或者一个 Crow 实例（因为 Crow 是 Bird 的子类型）。很好，传入子类型能按预期正常运作。

下面再定义一个函数，这个函数的参数是一个函数：

```
function clone(f: (b: Bird) => Bird): void {  
    // ...  
}
```

clone 的参数是一个函数，该函数的参数是一个 Bird，返回值也是一个 Bird。什么类型的函数可以作为 f 传入呢？显然，可以传入一个接受 Bird 并返回 Bird 的函数：

```
function birdToBird(b: Bird): Bird {  
    // ...  
}  
clone(birdToBird) // OK
```

如果是接受 Bird，但返回 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 可按预期调用，但是 birdToAnimal 报错了。为什么呢？假设 clone 是下面这样实现的：

```
function clone(f: (b: Bird) => Bird): void {  
    let parent = new Bird  
    let babyBird = f(parent)  
    babyBird.chirp()  
}
```

如果传给 clone 函数的 f 返回一个 Animal，那就不能调用 .chirp。在编译时，TypeScript 会确保传入的函数至少返回一个 Bird。

函数返回类型的协变指一个函数是另一个函数的子类型，即一个函数的返回类型 $<$: 另一个函数的返回类型。

好，那么参数的类型呢？

```
function animalToBird(a: Animal): Bird {  
    // ...  
}  
clone(animalToBird) // OK  
  
function crowToBird(c: Crow): Bird {  
    // ...  
}  
clone(crowToBird) // Error TS2345: Argument of type '(c: Crow) => Bird' is not  
                  // assignable to parameter of type '(b: Bird) => Bird'.
```

为了保证一个函数可赋值给另一个函数，该函数的参数类型（包括 `this`）都要 $>$: 另一个函数相应参数的类型。为了理解这其中的原因，请想一下在传给 `clone` 之前，用户会怎样实现 `crowToBird`。如果是下面这样实现的呢？

```
function crowToBird(c: Crow): Bird {  
    c.caw()  
    return new Bird  
}
```

现在，把 `crowToBird` 传给 `clone`，如果 `crowToBird` 的参数是 `Bird` 实例，TypeScript 将抛出异常，这是因为 `.caw` 只在 `Crow` 中定义了，不是所有 `Bird` 都定义了。

这表明，函数不对参数和 `this` 的类型做型变。也就是说，一个函数是另一个函数的子类型，必须保证该函数的参数和 `this` 的类型 $>$: 另一个函数相应参数的类型。

欣喜的是，我们不用记诵这些规则。如果代码编辑器在传递类型有误的函数时显示一条红色波浪线，你能联想到这些规则，知道 TypeScript 抛出错误的原因就可以了。



TSC 标志: strictFunctionTypes

考虑历史遗留问题，TypeScript 中的函数其实默认会对参数和 this 的类型做协变。如果想更安全一些，像前文所讲的那样不做型变，请在 `tsconfig.json` 中启用 `{"strictFunctionTypes": true}` 标志。

`strict` 模式包含 `strictFunctionTypes`，如果已经设置了 `{"strict": true}`，那就不用再启用 `strictFunctionTypes` 标志了。

6.1.3 可赋值性

子类型和超类型关系是静态类型语言的核心概念，对理解可赋值性也十分重要（提醒一下，可赋值性指在判断需要 B 类型的地方可否使用 A 类型时 TypeScript 采用的规则）。

TypeScript 在回答“`A` 类型是否可赋值给 `B` 类型？”这个问题时，将遵循几个简单的规则。对非枚举类型来说，例如数组、布尔值、数字、对象、函数、类、类的实例和字符串，以及字面量类型，在满足下述任一条件时，`A` 类型可赋值给 `B` 类型：

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，那么需要 `B` 的地方也可以使用 `A`。

规则 2 是规则 1 的例外，是为了方便与 JavaScript 代码互操作。

对于使用 `enum` 或 `const enum` 关键字创建的枚举类型，满足以下任一条件时，`A` 类型可赋值给枚举类型 `B`：

1. `A` 是枚举 `B` 的成员。
2. `B` 至少有一个成员是 `number` 类型，而且 `A` 是数字。

规则1与简单类型的规则完全一样（如果A是枚举B的成员，那么A的类型是B，也就是 $B \leq B$ ）。

规则2是为了方便使用枚举。正如3.2.13节所说，规则2是安全性的一大隐患。无需顾虑，根本不要使用枚举。

6.1.4 类型拓宽

类型拓宽（type widening）是理解TypeScript类型推导机制的关键。一般来说，TypeScript在推导类型时会放宽要求，故意推导出一个更宽泛的类型，而不限定为某个具体的类型。这样做对程序员是有好处的，大大减少了消除类型检查器报错的时间。

在第3章中，我们已经见过类型拓宽的实际应用。下面再看几个例子。

声明变量时如果允许以后修改变量的值（例如使用let或var声明），变量的类型将拓宽，从字面值放大到包含该字面量的基类型：

```
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
```

然而，声明不可变的变量时，情况则不同：

```
const a = 'x'        // 'x'
const b = 3            // 3
const c = true         // true

enum E {X, Y, Z}
const e = E.X          // E.X
```

我们可以显式注解类型，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}
```

如果使用 `let` 或 `var` 重新为非拓宽类型赋值，TypeScript 将自动拓宽。倘若不想让 TypeScript 拓宽，一开始声明时要显式注解类型：

```
const a = 'x'            // 'x'  
let b = a                // string  
  
const c: 'x' = 'x'        // 'x'  
let d = c                // 'x'
```

初始化为 `null` 或 `undefined` 的变量将拓宽为 `any`：

```
let a = null             // any  
a = 3                   // any  
a = 'b'                 // any
```

但是，当初始化为 `null` 或 `undefined` 的变量离开声明时所在的作用域后，TypeScript 将为其分配一个具体类型：

```
function x() {  
    let a = null           // any  
    a = 3                 // any  
    a = 'b'               // any  
    return a  
}  
  
x()                      // string
```

const 类型

TypeScript 中有个特殊的 `const` 类型，我们可以在单个声明中使用它禁止类型拓宽。这个类型用作类型断言（见 6.6.1 节）：

```
let a = {x: 3}            // {x: number}  
let b: {x: 3}             // {x: 3}  
let c = {x: 3} as const   // {readonly x: 3}
```

`const`不仅能阻止拓宽类型，还将递归把成员设为 `readonly`，不管数据结构的嵌套层级有多深：

```
let d = [1, {x: 2}]           // (number | {x: number})[]
let e = [1, {x: 2}] as const // readonly [1, {readonly x: 2}]
```

如果想让 TypeScript 推导的类型尽量窄一些，请使用 `as const`。

多余属性检查

TypeScript 检查一个对象是否可赋值给另一个对象类型时，也涉及到类型拓宽。

“结构和数组型变”一节讲过，对象类型的成员会做协变。但是，如果 TypeScript 严守这个规则，而不做额外的检查，将导致一个问题。

假如有个 `Options` 对象，我们把它传给类做些配置：

```
type Options = {
  baseURL: string
  cacheSize?: number
  tier?: 'prod' | 'dev'
}

class API {
  constructor(private options: Options) {}
}

new API({
  baseURL: 'https://api.mysite.com',
  tier: 'prod'
})
```

试想，如果有选项拼写错误会发生什么？

```
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'?
```

编写 JavaScript 代码经常遇到这样的问题，TypeScript 能捕获这种问题真是帮了一个大忙。可是，既然对象类型的成员会做协变，TypeScript 是如何捕获这种问题的呢？

过程是这样的：

- 预期的类型是 `{baseUrl: string, cacheSize?: number, tier?: 'prod' | 'dev'}`。
- 传入的类型是 `{baseUrl: string, tier: string}`。
- 传入的类型是预期类型的子类型，可是不知为何，TypeScript 知道要报告错误。

TypeScript 之所以能捕获这样的问题，是因为它会做多余属性检查，具体过程是：尝试把一个新鲜对象字面量类型（fresh object literal type）`T` 赋值给另一个类型 `U` 时，如果 `T` 有不在 `U` 中的属性，TypeScript 将报错。

新鲜对象字面量类型指的是 TypeScript 从对象字面量中推导出来的类型。如果对象字面量有类型断言（见 6.6.1 节），或者把对象字面量赋值给变量，那么新鲜对象字面量类型将拓宽为常规的对象类型，也就不能称其为新鲜对象字面量类型。

不太容易理解吧？下面再分析一下这个示例，这一次以不同的方式调用 API：

```
type Options = {
  baseUrl: string
  cacheSize?: number
  tier?: 'prod' | 'dev'
}

class API {
  constructor(private options: Options) {}
}
```

```

new API({ ❶
  baseURL: 'https://api.mysite.com',
  tier: 'prod'
})

new API({ ❷
  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({ ❸
  baseURL: 'https://api.mysite.com',
  badTier: 'prod'
} as Options)

let badOptions = { ❹
  baseURL: 'https://api.mysite.com',
  badTier: 'prod'
}
new API(badOptions)

let options: Options = { ❺
  baseURL: 'https://api.mysite.com',
  badTier: 'prod' //Error TS2322: Type '{baseURL: string; badTier: string}' is not assignable to type 'Options'.
}
new API(options)

```

- ❶ 使用 `baseURL` 和两个可选属性中的 `tier` 实例化 API。这一次能按预期运行。
- ❷ 这里，把 `tier` 错误拼写为 `badTier`。我们传给 `new API` 的是新鲜的选项对象（因为其类型是推导出来的，没有赋值给变量，也没有对类型下断言），因此 TypeScript 将做多余属性检查，发现 `badTier` 属性是多余的（选项对象中有，但是 `Options` 类型中没有）。
- ❸ 断言传入的无效选项对象是 `Options` 类型。TypeScript 不再把它视作新鲜对象，因此不做多余属性检查，所以不报错。如果你不熟悉 `as T` 句法，请翻到 6.6.1 节。
- ❹ 把选项对象赋值给变量 `badOptions`。TypeScript 不再把它视作新鲜对象，因此不做多余属性检查，所以不报错。

- ⑤ 显式注解 `options` 的类型为 `Options`, 赋值给 `options` 的是一个新鲜对象, 所以 TypeScript 执行多余属性检查, 捕获存在的问题。注意, 这种情况下, TypeScript 不在把 `options` 传给 `new API` 时做多余属性检查, 而是在把选项对象赋值给变量 `options` 时检查。

别担心, 你无需记住这些规则。这些规则供 TypeScript 内部使用, 力求以一种务实的方法捕获尽可能多的问题, 把重担从程序员的肩上卸下来。如果你发现 TypeScript 对这样的问题报错了, 而公司中久经沙场的代码基守护者、代码主要的审核人 Ivan 都没有察觉, 你知道是这方面的原因就行了。

6.1.5 细化

TypeScript 采用的是基于流的类型推导, 这是一种符号执行, 类型检查器在检查代码的过程中利用流程语句 (如 `if`、`?、||` 和 `switch`) 和类型查询 (如 `typeof`、`instanceof` 和 `in`) 细化类型, 就像程序员阅读代码的流程一样。^{注1} 这是一个极其便利的特性, 但是很少有语言支持。^{注2}

下面分析一个示例。假设我们想使用 TypeScript 构建一个定义 CSS 规则的 API, 有个同事想用该 API 为一个 HTML 元素设置 `width`。这个 API 要对传入的宽度做解析和验证。

首先, 实现一个函数, 把 CSS 字符串中的值和单位解析出来:

注 1: 符号执行是一种分析程序的方式, 这种方式使用一个特殊的程序 (称为符号求值程序) 运行程序, 过程与运行时运行程序时一样, 只是不为变量赋予具体的值, 而使用符号建模变量, 在程序运行的过程中约束变量的值。符号执行可以表达“这个变量从未使用”“这个函数永不返回”, 或者“在 `if` 语句的肯定分支中, 第 102 行的变量 `x` 肯定不是 `null`”。

注 2: 有一些语言支持基于流的类型推导, 例如 TypeScript、Flow、Kotlin 和 Ceylon。这是细化代码块中类型的一种方式, 类似于 C/Java 采用的显式类型注解, 以及 Haskell/OCaml/Scala 采用的模式匹配。基于流的类型推导在类型检查器中内嵌符号执行引擎, 为类型检查器提供反馈, 以接近人类程序员的方式分析程序。

```
// 使用字符串字面量的并集描述 CSS 单位的可能取值
type Unit = 'cm' | 'px' | '%'

// 列举单位
let units: Unit[] = ['cm', 'px', '%']

// 检查各个单位，如果没有匹配，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
}
```

我们可以使用 `parseUnit` 函数解析用户传入的宽度值。`width` 的值可能是一个数字（此时假定单位为像素），可能是一个带单位的字符串，也可能是 `null` 或 `undefined`。

下述代码对类型做了多次细化：

```
type Width = {
  unit: Unit,
  value: number
}

function parseWidth(width: number | string | null | undefined): Width | null {
  // 如果 width 是 null 或 undefined，尽早返回
  if (width == null) { ①
    return null
  }

  // 如果 width 是一个数字，默认单位为像素
  if (typeof width === 'number') { ②
    return {unit: 'px', value: width}
  }

  // 尝试从 width 中解析出单位
  let unit = parseUnit(width)
  if (unit) { ③
    return {unit, value: parseFloat(width)}
  }
}
```

```
// 否则，返回 null
return null ④
}
```

- ❶ TypeScript 足够智能，与 `null` 做不严格的等值检查便能在遇到 JavaScript 值 `null` 和 `undefined` 时返回 `true`。TypeScript 知道，这项检查通过后函数将返回，而未返回则表明检查不通过，`width` 的类型将变成 `number | string`（不可能为 `null` 或 `undefined`）。这一步把 `number | string | null | undefined` 类型细化为 `number | string`。
- ❷ `typeof` 运算符在运行时查询值的类型。TypeScript 在编译时也会利用 `typeof`：检查通过后，在 `if` 分支中，TypeScript 知道 `width` 的类型为 `number`；否则（因为 `if` 分支返回了），`width` 的类型必为 `string`，毕竟只剩下这一个类型了。
- ❸ 由于调用 `parseUnit` 函数可能返回 `null`，所以我们要检查结果是否为真值。^{注3} TypeScript 知道，如果 `unit` 为真值，那么在 `if` 分支中，其类型必为 `Unit`；否则，`unit` 必为假值，因此类型必为 `null`（细化 `Unit | null` 得来）。
- ❹ 最后，返回 `null`。只有用户给 `width` 传入一个字符串，而且字符串中包含不受支持的单位时才会出现这种情况。

这个示例把 TypeScript 的类型细化讲得清清楚楚，希望阅读这段代码的你已经完全理解了。TypeScript 对你在阅读和编写代码时脑中的想法了如指掌，通过类型检查和推导规则把你的想法具体表达出来。

辨别并集类型

我们知道，TypeScript 对 JavaScript 的运作了如指掌，而且能跟随你的步调细化类型，就像能读懂你在分析程序时脑中所想的逻辑一样。

注 3：JavaScript 有七个假值：`null`、`undefined`、`Nan`、`0`、`-0`、`""` 和 `false`。除此之外均为真值。

举个例子，假如我们在为一个应用构建自定义的事件系统。首先，定义几个事件类型，然后声明一个函数，处理收到的事件。假设 `UserTextEvent` 描述键盘事件（比如用户在 `<input />` 中输入一些文本），`UserMouseEvent` 描述鼠标事件（比如用户把鼠标移到 `[100, 200]` 坐标处）。

```
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]
}
```

TypeScript 知道，在 `if` 块中 `event.value` 肯定是一个字符串（因为使用 `typeof` 做了检查）。这意味着，在 `if` 块后面，`event.value` 肯定是 `[number, number]` 形式的元组（因为 `if` 块中有 `return`）。

如果情况变复杂了呢？下面为事件类型增加一些信息，看看 TypeScript 细化类型的结果如何：

```
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 (!!!)
}
```

`event.value` 的类型可以顺利细化，但是 `event.target` 却不可以。为什么呢？`handle` 函数的参数是 `UserEvent` 类型，但这并不意味着一定传入 `UserTextEvent` 或 `UserMouseEvent` 类型的值，还有可能传入 `UserMouseEvent` | `UserTextEvent` 类型的值。由于并集类型的成员有可能重复，因此 TypeScript 需要一种更可靠的方式，明确并集类型的具体情况。

为此，要使用一个字面量类型标记并集类型的各种情况。一个好的标记要满足下述条件：

- 在并集类型各组成部分的相同位置上。如果是对象类型的并集，使用相同的字段；如果是元组类型的并集，使用相同的索引。实际使用中，带标记的并集类型通常为对象类型。
- 使用字面量类型（字符串、数字、布尔值等字面量）。可以混用不同的字面量类型，不过最好使用同一种类型。通常，使用字符串字面量类型。
- 不要使用泛型。标记不应该有任何泛型参数。
- 要互斥（即在并集类型中是独一无二的）。

了解这些之后，下面来更新事件类型：

```
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，根据标记字段 (`event.type`) 的值细化 `event`，TypeScript 知道，在 `if` 分支中 `event` 必为 `UserTextEvent` 类型，而在 `if` 分支后面，`event` 必为 `UserMouseEvent` 类型。由于标记在并集类型的各个组成部分中是独一无二的，所以 TypeScript 知道二者是互斥的。

如果函数要处理并集类型的不同情况，应该使用标记。例如，在处理 Flux 动作、Redux 规约器或 React 的 `useReducer` 时，其作用十分巨大。

6.2 全面性检查

程序员睡觉之前会在床头柜上放两个杯子，一个杯子装满水，防止口渴，另一个杯子空着，防止不渴。

——佚名

全面性检查（也称穷尽性检查）是类型检查器所做的一项检查，为的是确保所有情况都被覆盖了。这个概念源自基于模式匹配的语言，例如 Haskell、OCaml 等。

TypeScript 在很多情况下都会做全面性检查，如果发现缺少某种情况会提醒你。这个特性对我们帮助极大，能避免很多常见的问题。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'
    }
}
```

显然，这里遗漏了好几天（一周为何这么长）。TypeScript 能捕获这个问题：

```
Error TS2366: Function lacks ending return statement and  
return type does not include 'undefined'.
```



TSC 标志：noImplicitReturns

如果想让 TypeScript 检查函数的所有代码路径都有返回值（在缺少返回值时抛出上述错误），在 `tsconfig.json` 中启用 `noImplicitReturns` 标志。是否启用这个标志由你自己决定，有些人喜欢代码中的 `return` 少一些，有些人喜欢多一些，以便提升类型安全，让类型检查器捕获更多的错误。

上述错误消息指出，可能是我们遗漏了某些情况，应该在最后加上一个兜底 `return` 语句，返回 'Sat' 等值（如果这样还真不错啊），也可能预示着我们要调整 `getNextDay` 的返回类型，改成 `Day | undefined`。为每一天都加上 `case` 语句之后，这个错误将消失（不信你试一试）。由于我们注解了 `getNextDay` 的返回类型，而没有分支能保证返回该类型的值，所以 TypeScript 才发出提醒。

这个示例的实现细节无关紧要，不管使用哪种控制结构（`switch`、`if`、`throw` 等），TypeScript 都能在未涵盖所有情况时做出提醒。

下面再看一个例子：

```
function isBig(n: number) {  
    if (n >= 100) {  
        return true  
    }  
}
```

可能是客户不断的催促让你大意了，忘记在对业务逻辑至关重要的 `isBig` 函数中处理小于 100 的数。同样，不要惧怕，TypeScript 在替你监视着呢：

```
Error TS7030: Not all code paths return a value.
```

周末闲暇时，你可能灵光一闪，想重写前面给出的 `getNextDay` 函数，把 `switch` 语句换掉，提高一些效率。为什么不在对象中做常量时间查找呢？

```
let nextDay = {  
    Mon: 'Tue'  
}  
  
nextDay.Mon // 'Tue'
```

家中的比雄犬在另一个房间狂叫（领居家的狗出了什么事），你的心已经飞出去，还没在 `nextDay` 对象中编写其他几天就提交代码，然后着手做其他事了。

虽然 TypeScript 会在下次访问 `nextDay.Tue` 时报错，但是当初声明 `nextDay` 时我们可以做好预防措施。预防措施有两个，将在 6.3.2 节和 6.3.3 节讲解。现在，我们先叉个话题，介绍一下对象类型的类型运算符。

6.3 对象类型进阶

对象是 JavaScript 的核心，为了以安全的方式描述和处理对象，TypeScript 提供了一系列方式。

6.3.1 对象类型的类型运算符

还记得“并集类型和交集类型”一节介绍的并集(`|`)和交集(`&`)类型运算符吗？TypeScript 提供的类型运算符不止这两个。下面再介绍几个处理对象结构的类型运算符。

“键入”运算符

假设有个复杂的嵌套类型，描述从社交媒体 API 中得到的 GraphQL API 响应：

```
type APIResponse = {  
    user: {  
        userId: string  
        friendList: {  
            count: number  
            friends: {  
                firstName: string  
                lastName: string  
            }[]  
        }  
    }  
}
```

```
}
```

从 API 中获取响应之后要渲染：

```
function getAPIResponse(): Promise<APIResponse> {
  // ...
}

function renderFriendList(friendList: unknown) {
  // ...
}

let response = await getAPIResponse()
renderFriendList(response.user.friendList)
```

`friendList` 是什么类型呢（暂且使用 `unknown`）？我们可以单独声明 `friendList` 的类型，然后重新实现顶层的 `APIResponse` 类型：

```
type FriendList = {
  count: number
  friends: {
    firstName: string
    lastName: string
  }[]
}

type APIResponse = {
  user: {
    userId: string
    friendList: FriendList
  }
}

function renderFriendList(friendList: FriendList) {
  // ...
}
```

但是这样做要为每个顶层类型想一个名称，我们可不想一直做这样的事（例如，使用构建工具为 GraphQL 模式生成 TypeScript 类型）。除此之外，我们可以“键入”类型：

```
type APIResponse = {
  user: {
    userId: string
    friendList: {
      count: number
      friends: {
        firstName: string
        lastName: string
      }[]
    }
  }
}

type FriendList = APIResponse['user']['friendList']

function renderFriendList(friendList: FriendList) {
  // ...
}
```

任何结构（对象、类构造方法或类的实例）和数组都可以“键入”。例如，单个好友的类型可以这样声明：

```
type Friend = FriendList['friends'][number]
```

`number` 是“键入”数组类型的方式。若是元组，使用`0`、`1`或其他数字字面量类型表示想“键入”的索引。

“键入”的句法与在 JavaScript 对象中查找字段的句法类似，这是故意为之的：既然可以在对象中查找值，那么也能在结构中查找类型。但是要注意，通过“键入”查找属性的类型时，只能使用方括号表示法，不能使用点号表示法。

keyof 运算符

`keyof` 运算符获取对象所有键的类型，合并为一个字符串字面量类型。以前面的 `APIResponse` 为例：

```
type ResponseKeys = keyof APIResponse // 'user'
type UserKeys = keyof APIResponse['user'] // 'userId' | 'friendList'
type FriendListKeys =
  keyof APIResponse['user']['friendList'] // 'count' | 'friends'
```

把“键入”和`keyof`运算符结合起来，可以实现对类型安全的读值函数，读取对象中指定键的值：

```
function get<①
  O extends object,
  K extends keyof O ②
>(
  o: O,
  k: K
): O[K] { ③
  return o[k]
}
```

- ① `key` 函数的参数为一个对象 `o` 和一个键 `k`。
- ② `keyof O` 是一个字符串字面量类型并集，表示 `o` 对象的所有键。`K` 类型扩展这个并集（是该并集的子类型）。假如 `o` 的类型为 `{a: number, b: string, c: boolean}`，那么 `keyof o` 的类型为 `'a' | 'b' | 'c'`，而 `K`（扩展自 `keyof o`）可以是类型 `'a'`、`'b'`、`'a' | 'c'`，或者 `keyof o` 的其他子类型。
- ③ `O[K]` 的类型为在 `O` 中查找 `K` 得到的具体类型。接着②说，如果 `K` 是 `'a'`，那么在编译时 `get` 返回一个数字；如果 `K` 是 `'b' | 'c'`，那么 `get` 返回 `string | boolean`。

这两个类型运算符强大的地方在于，可以准确、安全地描述结构类型：

```
type ActivityLog = {
  lastEvent: Date
  events: {
    id: string
    timestamp: Date
    type: 'Read' | 'Write'
  }[]
}

let activityLog: ActivityLog = // ...
let lastEvent = get(activityLog, 'lastEvent') // Date
```

TypeScript 将为我们代劳，在编译时确认 `lastEvent` 的类型是否为 `Date`。当然，

我们可以更进一步，更深层次地“键入”对象。下面重载 get 函数，让它最多接受三个键：

```
type Get = { ①
  <
    O extends object,
    K1 extends keyof O
  >(o: O, k1: K1): O[K1] ②
  <
    O extends object,
    K1 extends keyof O,
    K2 extends keyof O[K1] ③
  >(o: O, k1: K1, k2: K2): O[K1][K2] ④
  <
    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] ⑤
}

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"'.
```

- ① 为 get 声明重载函数签名，分别对应以一个键、两个键和三个键调用 get 函数的情况。
- ② 接受一个键的情况与前例相同：O 是 object 的子类型，K1 是那个对象的键的子类型，而返回类型是通过 K1 “键入” O 之后得到的具体类型。
- ③ 接受两个键的情况与接受一个键的情况类似，不过多声明了一个泛型 K2，用于描述通过 K1 “键入” O 后得到的嵌套对象的键。

- ④ 根据②，这里要“键入”两次，首先获得 `0[K1]` 的类型，然后在此结果上获得 `[K2]` 的类型。
- ⑤ 这个示例最多只处理三个键。实际编写相关的库时，可能还要处理更多的键。

很棒吧！如果有时间，请把这个示例给你使用 Java 的朋友看一下，在你介绍的过程中，一定要露出洋洋得意的表情。



TSC 标志: `keyofStringsOnly`

在 JavaScript 中，对象和数组的键可以使用字符串，也可以使用符号。按照惯例，数组通常使用数字键，但在运行时将被强制转换为字符串。

鉴于此，TypeScript 中的 `keyof` 运算符默认返回 `number | string | symbol` 类型（处理具体结构时，TypeScript 将推导出该并集类型的子类型）。

这是正确的行为，但容易导致使用 `keyof` 时啰里啰嗦，因为我们要向 TypeScript 证明某个键是 `string`，而不是 `number` 或 `symbol`。

如果想使用 TypeScript 以前的行为，即要求键必须为字符串，在 `tsconfig.json` 中启用 `keyofStringsOnly` 标志。

6.3.2 Record 类型

TypeScript 内置的 `Record` 类型用于描述有映射关系的对象。

请回顾一下 6.2 节中的 `Weekday` 示例，强制要求对象定义一组具体的键有两种方式。第一种就是使用 `Record` 类型。

下面使用 `Record` 构建一个映射，从一周中的每一日映射到次日。使用 `Record` 可以为 `nextDay` 的键和值添加一些约束：

```
type Weekday = 'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri'  
type Day = Weekday | 'Sat' | 'Sun'
```

```
let nextDay: Record<Weekday, Day> = {  
    Mon: 'Tue'  
}
```

现在，立即就能看到一个对我们有帮助的错误消息：

```
Error TS2739: Type '{Mon: "Tue"}' is missing the following properties  
from type 'Record<Weekday, Day>': Tue, Wed, Thu, Fri.
```

显然，把缺少的日子加上之后，这个错误就会消失。

与常规的对象索引签名相比，`Record` 提供了更多的便利：使用常规的索引签名可以约束对象中值的类型，不过键只能用 `string`、`number` 或 `symbol` 类型；使用 `Record` 还可以约束对象的键为 `string` 和 `number` 的子类型。

6.3.3 映射类型

TypeScript 还提供了一种更强大的方式，即映射类型（mapped type），使用这种方式声明 `nextDay` 的类型更安全。下面使用映射类型表示 `nextDay` 对象中有对应 `Weekday` 的各个键，而且其值为 `Day` 类型：

```
let nextDay: {[K in Weekday]: Day} = {  
    Mon: 'Tue'  
}
```

TypeScript 也会提供适当的信息，帮助我们修正缺少的日子：

```
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。与字面量类型一样，这是一项实用特性，为 JavaScript 弥补了静态类型。

可以看到，映射类型使用独特的句法。与索引签名相同，一个对象最多有一个映射类型：

```
type MyMappedType = {
  [Key in UnionType]: ValueType
}
```

从名称可以看出，这是一种在对象的键和值的类型之间建立映射的方式。其实，TypeScript 内置的 `Record` 类型也是使用映射类型实现的：

```
type Record<K extends keyof any, T> = {
  [P in K]: T
}
```

映射类型的功能比 `Record` 强大，在指定对象的键和值的类型以外，如果结合“键入”类型，还能约束特定键的值是什么类型。

下面举几个例子说明使用映射类型可以做哪些事：

```
type Account = {
  id: number
  isEmployee: boolean
  notes: string[]
}
```

// 所有字段都是可选的

```
type OptionalAccount = {
  [K in keyof Account]?: Account[K] ①
}
```

// 所有字段都可为 null

```
type NullableAccount = {
  [K in keyof Account]: Account[K] | null ②
}
```

// 所有字段都是只读的

```
type ReadonlyAccount = {
  readonly [K in keyof Account]: Account[K] ③
}
```

// 所有字段都是可写的（等同于 Account）

```
type Account2 = {
  -readonly [K in keyof ReadonlyAccount]: Account[K] ④
}
```

```
// 所有字段都是必须的（等同于 Account）
type Account3 = {
  [K in keyof OptionalAccount]-?: Account[K] ⑤
}
```

- ① 新建对象类型 `OptionalAccount`, 与 `Account` 建立映射, 在此过程中把各个字段标记为可选的。
- ② 新建对象类型 `NullableAccount`, 与 `Account` 建立映射, 在此过程中为每个字段增加可选值 `null`。
- ③ 新建对象类型 `ReadonlyAccount`, 与 `Account` 建立映射, 把各字段标记为只读（即可读不可写）。
- ④ 字段可以标记为可选的 (?) 或只读的 (readonly), 也可以把这个约束去掉。使用减号 (-) 运算符（一个特殊的类型运算符, 只对映射类型可用）可以把 ? 和 `readonly` 撤销, 分别把字段还原为必须的和可写的。这里新建一个对象类型 `Account2`, 与 `ReadonlyAccount` 建立映射, 使用减号 (-) 运算符把 `readonly` 修饰符去掉, 最终得到的类型等同于 `Account`。
- ⑤ 新建对象类型 `Account3`, 与 `OptionalAccount` 建立映射, 使用减号 (-) 运算符把可选 (?) 运算符去掉, 最终得到的类型等同于 `Account`。



减号 (-) 运算符有个对应的加号 (+) 运算符。一般不直接使用加号运算符, 因为它通常蕴含在其他运算符中。在映射类型中, `readonly` 等效于 `+readonly`, `?` 等效于 `+?`。`+` 的存在只是为了确保整体协调。

内置的映射类型

前一节讨论的映射类型非常有用, TypeScript 甚至内置了一些:

`Record<Keys, Values>`

键的类型为 `Keys`、值的类型为 `Values` 的对象。

`Partial<Object>`

把 `Object` 中的每个字段都标记为可选的。

Required<Object>

把 Object 中的每个字段都标记为必须的。

Readonly<Object>

把 Object 中的每个字段都标记为只读的。

Pick<Object, Keys>

返回 Object 的子类型，只含指定的 Keys。

6.3.4 伴生对象模式

伴生对象模式源自 Scala (<http://bit.ly/2I9Nqg2>)，目的是把同名的对象和类配对在一起。在 TypeScript 中也有类似的模式，而且作用相似，即把类型和对象配对在一起，我们也称之为伴生对象模式。

伴生对象模式是下面这样的：

```
type Currency = {  
    unit: 'EUR' | 'GBP' | 'JPY' | 'USD'  
    value: number  
}  
  
let Currency = {  
    DEFAULT: 'USD',  
    from(value: number, unit = Currency.DEFAULT): Currency {  
        return {unit, value}  
    }  
}
```

还记得吗，TypeScript 中的类型和值分别在不同的命名空间中。10.4 节将对这一思想做深入说明。这意味着，在同一个作用域中，可以有同名（这里的 Currency）的类型和值。伴生对象模式在彼此独立的命名空间中两次声明相同的名称，一个是类型，另一个是值。

这种模式有几个不错的性质。首先，可以把语义上归属同一名称的类型和值放在一起。其次，使用方可以一次性导入二者。

```
import {Currency} from './Currency'

let amountDue: Currency = { ❶
  unit: 'JPY',
  value: 83733.10
}

let otherAmountDue = Currency.from(330, 'EUR') ❷
```

❶ 使用的是类型 `Currency`。

❷ 使用的是值 `Currency`。

如果一个类型和一个对象在语义上有关联，就可以使用伴生对象模式，由对象提供操作类型的实用方法。

6.4 函数类型进阶

本节讲解函数类型常用的几种高级技术。

6.4.1 改善元组的类型推导

TypeScript 在推导元组的类型时会放宽要求，推导出的结果尽量宽泛，不在乎元组的长度和各位置的类型。

```
let a = [1, true] // (number | boolean)[]
```

然而，有时我们希望推导的结果更严格一些，把上例中的 `a` 视作固定长度的元组，而不是数组。当然，我们可以使用类型断言把元组转换成元组类型（详见 6.6.1 节），也可以使用 `as const` 断言（`const` 类型）把元组标记为只读的，尽量收窄推导出的元组类型。

可是，如果我们希望元组的推导结果为元组类型，但不想使用类型断言，也不想使用 `as const` 收窄推导结果，并把元组标记为只读的呢？为此，我们可以利用 TypeScript 推导剩余参数的类型的方式（详见“使用受限的多态模拟变长参数”一节）：

```
function tuple<①>(②)
  T extends unknown[] ②
  >(
    ...ts: T ③
  ): T { ④
    return ts ⑤
  }

let a = tuple(1, true) // [number, boolean]
```

- ① 声明 `tuple` 函数，用于构建元组类型（代替内置的 `[]` 句法）。
- ② 声明一个类型参数 `T`，它是 `unknown[]` 的子类型（表明 `T` 是任意类型的数组）。
- ③ `tuple` 函数接受不定数量的参数 `ts`。由于 `T` 描述的是剩余参数，因此 TypeScript 推导出的是一个元组类型。
- ④ `tuple` 函数的返回类型与 `ts` 的推导结果相同。
- ⑤ 这个函数返回传入的参数。神奇之处全在类型中。

如果代码中大量使用元组类型，而又不想使用类型断言，可以利用这个技术。

6.4.2 用户定义的类型防护措施

在某些处理真假值的情况下，比如函数的返回值，只说函数返回 `boolean` 还不够。举个例子，我们来编写一个判断传入的参数是否为字符串的函数：

```
function isString(a: unknown): boolean {
  return typeof a === 'string'
}

isString('a') // 求值结果为 true
isString([7]) // 求值结果为 false
```

目前看来没什么问题。那么，在实际使用中，`isString` 函数的效果如何呢？

```
function parseInput(input: string | number) {
  let formattedInput: string
  if (isString(input)) {
    formattedInput = input.toUpperCase() // Error TS2339: Property 'toUpperCase'
```

```
    } // does not exist on type 'number'.
}
```

怎么回事？细化类型时可以使用的 `typeof` 运算符（见 6.1.5 节），在这里怎么不起作用了？

类型细化的能力有限，只能细化当前作用域中变量的类型，一旦离开这个作用域，类型细化能力不会随之转移到新作用域中。在 `isString` 的实现中，我们使用 `typeof` 把输入参数的类型细化为 `string`，可是由于类型细化不转移到新作用域中，细化能力将消失，TypeScript 只知道 `isString` 函数返回一个布尔值。

`isString` 函数是返回一个布尔值，但是我们要让类型检查器知道，当返回的布尔值是 `true` 时，表明我们传给 `isString` 函数的是一个字符串。为此，我们要使用用户定义的类型防护措施：

```
function isString(a: unknown): a is string {
    return typeof a === 'string'
}
```

类型防护措施是 TypeScript 内置的特性，是 `typeof` 和 `instanceof` 细化类型的背后机制。可是，有时我们需要自定义类型防护措施的能力，`is` 运算符就起这个作用。如果函数细化了参数的类型，而且返回一个布尔值，我们可以使用用户定义的类型防护措施确保类型的细化能在作用域之间转移，在使用该函数的任何地方都起作用。

用户定义的类型防护措施只限于一个参数，但是不限于简单的类型：

```
type LegacyDialog = ...
type Dialog = ...

function isLegacyDialog(
    dialog: LegacyDialog | Dialog
): dialog is LegacyDialog {
    // ...
}
```

用户定义的类型防护措施不太常用，不过使用这个特性可以简化代码，提升代码的可重用性。如果没有这个特性，要在行内使用 `typeof` 和 `instanceof` 类型防护措施，而构建 `isLegacyDialog` 和 `isString` 之类的函数做同样的检查可实现更好的封装，提升代码的可读性。

6.5 条件类型

在 TypeScript 的众多特性中，条件类型算是最独特的。概括地说，条件类型表达的是，“声明一个依赖类型 U 和 V 的类型 T，如果 $U <: V$ ，把 T 赋值给 A；否则，把 T 赋值给 B”。

用代码写出来，可能是下面这样的：

```
type IsString<T> = T extends string ①
  ? true ②
  : false ③

type A = IsString<string> // true
type B = IsString<number> // false
```

下面逐行分析一下。

- ① 声明一个条件类型 `IsString`，它有个泛型参数 `T`。这个条件类型中的“条件”是 `T extends string`，即“`T`是不是 `string` 的子类型？”
- ② 如果 `T` 是 `string` 的子类型，得到的类型为 `true`。
- ③ 否则，得到的类型为 `false`。

注意，这里使用的句法与值层面的三元表达式差不多，只是现在位于类型层面。与常规三元表达式相似的另一点是，条件类型句法可以嵌套。

条件类型不限于只能在类型别名中使用，可以使用类型的地方几乎都能使用条件类型，包括类型别名、接口、类、参数的类型，以及函数和方法的泛型默认类型。

6.5.1 条件分配

通过前面的示例我们知道，TypeScript 可以通过多种不同的方式表达简单的条件，例如条件类型、重载的函数签名和映射类型。不过，条件类型的作用更强大，原因在于条件类型遵守分配律（还记得代数课上老师教的分配律吗）。因此，表 6-1 中左边的条件类型与右边的表达式是等效的。

表 6-1：分配条件类型

这个类型……	等价于
<code>string extends T ? A : B</code>	<code>string extends T ? A : B</code>
<code>(string number) extends T ? A : B</code>	<code>(string extends T ? A : B) (number extends T ? A : B)</code>
<code>(string number boolean) extends T ? A : B</code>	<code>(string extends T ? A : B) (number extends T ? A : B) (boolean extends T ? A : B)</code>

笔者知道你买这本书不是学数学的，你为的是类型。那就具体一点。假如有个函数接受 `T` 类型的参数，把该参数提升为数组类型 `T[]`。那么，如果传入一个并集类型呢？

```
type ToArray<T> = T[]
type A = ToArray<number>           // number[]
type B = ToArray<number | string> // (number | string)[]
```

很好理解。如果是条件类型呢（注意，其实条件在这里什么作用也不起，两个分支最终得到的都是 `T[]` 类型；这里我们是想让 TypeScript 把 `T` 分配给元组类型）？请看下面的代码：

```
type ToArray2<T> = T extends unknown ? T[] : T[]
type A = ToArray2<number> // number[]
type B = ToArray2<number | string> // number[] | string[]
```

明白了吗？使用条件类型时，TypeScript 将把并集类型分配到各个条件分支中。这就好像把条件类型映射（分配）到并集类型的各个组成部分上。

这样做有什么用呢？答案是可以安全地表达一些常见的操作。

我们知道，TypeScript 内置了计算两个类型交集的 & 运算符，还内置了计算两个类型并集的 | 运算符。下面我们来构建 Without<T, U>，计算在 T 中而不在 U 中的类型。

```
type Without<T, U> = T extends U ? never : T
```

Without 的用法如下：

```
type A = Without<
  boolean | number | string,
  boolean
> // number | string
```

下面具体分析 TypeScript 是如何计算这个类型的：

1. 先分析输入的类型：

```
type A = Without<boolean | number | string, boolean>
```

2. 把条件分配到并集中：

```
type A = Without<boolean, boolean>
| Without<number, boolean>
| Without<string, boolean>
```

3. 代入 Without 的定义，替换 T 和 U：

```
type A = (boolean extends boolean ? never : boolean)
| (number extends boolean ? never : number)
| (string extends boolean ? never : string)
```

4. 计算条件：

```
type A = never
| number
| string
```

5. 化简：

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

如果条件类型没有分配性质，最终的结果将是 never（请自己分析一下原因）。

6.5.2 infer 关键字

条件类型的最后一个特性是可以在条件中声明泛型。回顾一下，目前我们只见过一种声明泛型参数的方式，即使用尖括号（`<T>`）。在条件类型中声明泛型不使用这个句法，而使用 `infer` 关键字。

下面声明一个条件类型 `ElementType`，获取数组中元素的类型：

```
type ElementType<T> = T extends unknown[] ? T[number] : T  
type A = ElementType<number[]> // number
```

使用 `infer` 关键字可以重写为：

```
type ElementType2<T> = T extends (infer U)[] ? U : T  
type B = ElementType2<number[]> // number
```

这里，`ElementType` 和 `ElementType2` 是等价的。注意，`infer` 子句声明了一个新的类型变量 `U`，TypeScript 将根据传给 `ElementType2` 的 `T` 推导 `U` 的类型。

另外注意，`U` 是在行内声明的，而没有和 `T` 一起在前面声明。倘若在前面声明，结果如何？

```
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！由于 `ElementUgly` 定义了两个泛型（`T` 和 `U`），因此在实例化 `ElementUgly` 时要传入两个参数。可是，如果这么做就违背了声明 `ElementUgly` 类型的初衷，计算 `U` 的重担落在了调用方身上，而我们希望由 `ElementUgly` 自己计算类型。

当然，这个示例不太恰当，毕竟我们已经有能获取数组元素类型的“键入”运算符（`[]`）了。不妨再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
```

可见，`[].slice` 的第二个参数是 `number | undefined` 类型。而且在编译时便可知晓这一点。Java 能做到吗？

6.5.3 内置的条件类型

利用条件类型可以在类型层面表达一些强大的操作。鉴于此，TypeScript 自带了一些全局可用的条件类型。

Exclude<T, U>

与前面的 `Without` 类型相似，计算在 `T` 中而不在 `U` 中的类型：

```
type A = number | string  
type B = string  
type C = Exclude<A, B> // number
```

Extract<T, U>

计算 `T` 中可赋值给 `U` 的类型：

```
type A = number | string  
type B = string  
type C = Extract<A, B> // string
```

NonNullable<T>

从 `T` 中排除 `null` 和 `undefined`：

```
type A = {a?: number | null}  
type B = NonNullable<A['a']> // number
```

ReturnType<F>

计算函数的返回类型（注意，不适用于泛型和重载的函数）：

```
type F = (a: number) => string  
type R = ReturnType<F> // string
```

计算类构造方法的实例类型：

```
type A = {new(): B}
type B = {b: number}
type I = InstanceType<A> // {b: number}
```

6.6 解决办法

有时，我们没有足够的时间把所有类型都规划好，这时我们希望 TypeScript 能相信我们，即便如此也是安全的。说不定我们使用的第三方模块中声明的类型是错误的，但是我们想先测试自己的代码，闲下来之后再为 DefinitelyTyped^{注4} 贡献修正后的版本，还有可能我们是从 API 中获取数据，而暂时还没有使用 Apollo 重新生成类型声明。

可喜的是，TypeScript 知道我们是人类，为我们提供了一些救命稻草，在我们没有时间向 TypeScript 证明安全性的时候，让我们放心做一些操作。



TypeScript 虽然提供了这些救命稻草，但是要尽量少用。如果大量依赖这些特性，可能说明你的做法不当。

6.6.1 类型断言

给定类型 B，如果 A <: B <: C，那么我们可以向类型检查器断定，B 其实是 A 或 C。注意，我们只能断定一个类型是自身的超类型或子类型，不能断定 number 是 string，因为这两个类型毫无关系。

TypeScript 为类型断言提供了两种句法：

```
function formatInput(input: string) {
    // ...
}
```

注 4： DefinitelyTyped (<https://github.com/DefinitelyTyped/DefinitelyTyped>) 是一个开源项目，为第三方 JavaScript 添加类型声明。详见 11.4.2 节。

```
function getUserInput(): string | number {  
    // ...  
}  
  
let input = getUserInput()  
  
// 断定 input 是字符串  
formatInput(input as string) ①  
  
// 等效于  
formatInput(<string>input) ②
```

- ❶ 使用类型断言 (as) 告诉 TypeScript，input 是字符串，而不是 `string | number` 类型。如果想先测试 `formatInput` 函数，而且肯定 `getUserInput` 函数返回一个字符串，就可以这样做。
- ❷ 类型断言的旧句法使用尖括号。这两种句法的作用是相同的。



类型断言的 `as` 句法和尖括号 (`<>`) 句法，优先使用前者。`as` 句法意思明确，而尖括号句法可能与 TSX 句法冲突（见“`TSX = JSX + TypeScript`”一节）。可以使用 `TSLint` 的 `no-angle-bracket-type-assertion` 规则 (<http://bit.ly/2WEGGKe>) 强制要求代码基使用 `as` 句法。

有时，两个类型之间关系不明，无法断定具体类型。此时，直接断定为 `any` (6.1.3 节讲过，`any` 可赋值给任何类型)，然后找个角落，思考一下代码有没有问题：

```
function addToList(list: string[], item: string) {  
    // ...  
}  
  
addToList('this is really,' as any, 'really unsafe')
```

显然，类型断言不安全，应尽量避免使用。

6.6.2 非空断言

可为空的类型，即 `T | null` 或 `T | null | undefined` 类型，比较特殊，

TypeScript 为此提供了专门的句法，用于断定类型为 `T`，而不是 `null` 或 `undefined`。这种情况并不少见。

假如我们编写了一个框架，用于显示和隐藏 Web 应用中的对话框。每个对话框都有唯一的 ID，通过 ID 可以获取对话框的 DOM 节点的引用。对话框从 DOM 中移除后，把对应的 ID 删除，表明该对话框已经不在 DOM 中。

```
type Dialog = {  
    id?: string  
}  
  
function closeDialog(dialog: Dialog) {  
    if (!dialog.id) { ❶  
        return  
    }  
    setTimeout(() => ❷  
        removeFromDOM(  
            dialog,  
            document.getElementById(dialog.id) // Error TS2345: Argument of type  
                // 'string | undefined' is not assignable  
                // to parameter of type 'string'. ❸  
        )  
    )  
}  
  
function removeFromDOM(dialog: Dialog, element: Element) {  
    element.parentNode.removeChild(element) // Error TS2531: Object is possibly  
        // 'null'. ❹  
    delete dialog.id  
}
```

- ❶ 如果对话框已经被删除（因此没有 `id`），那就尽早返回。
- ❷ 在下一次事件轮询时，把对话框从 DOM 中删除，给依赖 `dialog` 的其他代码留出时间运行完毕。
- ❸ 身处一个箭头函数中，开始一个新作用域。TypeScript 不知道❶和❹之间的代码修改了 `dialog`，因此❶中所做的类型细化不起作用。鉴于此，虽然 `dialog.id` 存在绝对可以确定 DOM 中有该 ID 对应的元素（毕竟框架就是这么设计的），但是在 TypeScript 看来，调用 `document.getElementById`

返回的类型将是 `HTMLElement | null`。我们知道结果是不为空的 `HTMLElement`，但是 TypeScript 不知道，TypeScript 只知道我们指定的类型。

- ④ 类似地，尽管我们知道对话框一定在 DOM 中，而且绝对有父级 DOM 节点，然而 TypeScript 只知道 `element.parentNode` 的类型是 `Node | null`。

这个问题的一种修正方法是大量使用 `if (_ === null)` 做检查。在不确定是否为 `null` 时，确实要这么做，但是如果确定不可能是 `null | undefined`，可以使用 TypeScript 提供的一种特殊句法：

```
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
}
```

注意深藏其中的非空断言运算符 `(!)`。这个运算符告诉 TypeScript，我们确定 `dialog.id`、`document.getElementById` 调用和 `element.parentNode` 得到的结果已定义。在可能为 `null` 或 `undefined` 类型的值后面加上非空断言运算符，TypeScript 将假定该类型已定义：`T | null | undefined` 变成 `T, number | string | null` 变成 `number | string` 等。

如果频繁使用非空断言，这就表明你的代码需要重构。比如说，我们可以把 `Dialog` 拆分为两个类型的并集，这样就不用使用非空断言了：

```
type VisibleDialog = {id: string}
type DestroyedDialog = {}
type Dialog = VisibleDialog | DestroyedDialog
```

然后更新 `closeDialog` 函数，改用并集类型：

```
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
}
```

确认 `dialog` 有 `id` 属性之后（表明是 `VisibleDialog` 类型），即使在箭头函数中 TypeScript 也知道 `dialog` 引用没有变化：箭头函数内的 `dialog` 与外面的 `dialog` 相同，因此细化结果随之转移，不会像前例那样不起作用。

6.6.3 明确赋值断言

TypeScript 为非空断言提供了专门的句法，用于检查有没有明确赋值（提醒一下，TypeScript 通过明确赋值检查确保使用变量时已经为其赋值）。例如：

```
let userId: string

userId.toUpperCase() // Error TS2454: Variable 'userId' is used
                    // before being assigned.
```

显然，TypeScript 十分周到，为我们捕获了这个错误。我们声明了变量 `userId`，但是还未赋值就想把它转换成大写形式。倘若 TypeScript 没有注意到这个问题，将在运行时抛出错误。

但是，如果是下面这样的代码呢？

```
let userId: string
fetchUser()

userId.toUpperCase() // Error TS2454: Variable 'userId' is used
                     // before being assigned.

function fetchUser() {
  userId = globalCache.get('userId')
}
```

我们使用的是世界上最好的缓存，每次查询都能保证一定得到结果。因此，调用 `fetchUser` 之后，`userId` 肯定已经有值了。但是，TypeScript 无法通过静态方式知晓这一点，所以依然抛出相同的错误。我们可以使用明确赋值断言告诉 TypeScript，在读取 `userId` 时，肯定已经为它赋值了（留意感叹号）：

```
let userId!: string
fetchUser()

userId.toUpperCase() // OK

function fetchUser() {
  userId = globalCache.get('userId')
}
```

与类型断言和非空断言一样，如果经常使用明确赋值断言，可能表明你的代码有问题。

6.7 模拟名义类型

读到这里，如果我在半夜三点把你摇醒，大声问你“TypeScript 是结构化类型系统还是名义类型系统？”，你肯定会大叫一声，“不明摆着是结构化类型吗！滚出我的房间，要不我报警了！”如果有人深更半夜问我这样的问题，我也会这么回答。

说是这么说，但是有时名义类型确有用武之地。举个例子，比如说你的应用中有几个 ID 类型，以不同的方式确定系统中各种对象的类型：

```
type CompanyID = string
type OrderID = string
type UserID = string
type ID = CompanyID | OrderID | UserID
```

`UserID` 类型的值可能是一个简单的哈希值，比如 "d21b1dbf"。尽管你可能会声明一个别名 `UserID`，但是说到底这就是常规的字符串。应用中可能有个函数，其参数为一个 `UserID` 值：

```
function queryForUser(id: UserID) {
    // ...
}
```

这是很好的文档，有助于其他工程师了解该传入哪个类型的 ID。可是，`UserID` 毕竟只是 `string` 的别名，有些问题还是无法避免。万一工程师不小心传入错误的 ID 类型，类型系统就无能为力了。

```
let id: CompanyID = 'b4843361'
queryForUser(id) // OK (!!?)
```

这时就体现名义类型的作用了。^{注5} 虽然 TypeScript 原生不支持名义类型，但是我们可以使用类型烙印（type branding）技术模拟实现。使用类型烙印技术之前要稍微设置一下，而且在 TypeScript 中使用这个技术不像在原生支持名义类型别名的语言中那么平顺。可是，带烙印的类型可以极大地提升程序的安全性。



是否使用类型烙印技术取决于具体应用和工程师团队的规模（对大型团队来说，这项技术能更好地发挥避免错误的作用）。

首先，为各个名义类型合成类型烙印：

^{注5} 在某些语言中，名义类型也叫“隐含类型”（opaque type）。

```
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
```

显然，使用 `string` 和 `{readonly brand: unique symbol}` 的交集显得有点乱，但是只能这么做，如果想创建这个类型的值，别无他法，只能使用断言。这就是带烙印的类型的一个重要性质：不太可能意外使用错误的类型。笔者选择的“烙印”是 `unique symbol`，因为这是 TypeScript 中两个真正意义上是名义类型的类型之一（另一个是 `enum`）。之所以取这个烙印与 `string` 的交集，是为了通过断言指明给定的字符串属于这个带烙印的类型。

现在，我们要找到一种方式创建 `CompanyID`、`OrderID` 和 `UserID` 类型的值。为此，我们将使用伴生对象模式（见 6.3.4 节）。我们将为每个带烙印的类型声明一个构造函数，使用类型断言构建各合成类型的值：

```
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，看一下如何使用这些类型：

```
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'.
```

这种方式的优点是，降低了运行时的开销，每构建一个 ID 只需调用一个函数，而且 JavaScript VM 还有可能把函数放在内存内。在运行时，一个 ID 就是一个字符串，烙印纯粹是一种编译时结构。

同样，多数应用没必要费这么大事。不过，对大型应用来说，或者处理容易混淆的类型时（比如不同种类的 ID），带烙印的类型能极大地提升安全性。

6.8 安全地扩展原型

构建 JavaScript 应用时，传统的观点是扩展内置类型的原型不安全。这个观点可追溯到 jQuery 出现之前的日子，那时 JavaScript 高手在构建 MooTools (<https://mootools.net>) 等库时都是直接扩展和覆盖内置的原型方法。但是，各路高手都在设法增强原型的功能，冲突随之而来。没有静态类型系统的管护，这些冲突只能在运行时暴露出来，由怒冲冲的用户反馈回来。

没有用过 JavaScript 的人可能会感到惊讶，JavaScript 居然允许在运行时修改内置的方法（例如 `[].push`、`'abc'.toUpperCase` 或 `Object.assign`）。JavaScript 是一门十分动态的语言，每一个内置对象的原型都可以直接访问，包括 `Array.prototype`、`Function.prototype`、`Object.prototype` 等。

虽然过去认为扩展原型不安全，但是有了 TypeScript 提供的静态类型系统，现在可以放心扩展原型了。^{注6}

举个例子，为 `Array` 原型添加 `zip` 方法。为了安全扩展原型，我们要做两件事。首先，在一个 `.ts` 文件（比如 `zip.ts`）中扩展 `Array` 原型的类型；然后，新增 `zip` 方法，增强原型的功能。

注 6： 不建议扩展原型还有其他原因，例如代码可移植性、保持依赖图清晰明了，以及只加载真正使用的函数，提升性能。然而，安全性已经不在其列。

```
// 让 TypeScript 知道 .zip 方法的存在
interface Array<T> { ❶
    zip<U>(list: U[]): [T, U][]
}

// 实现 .zip 方法
Array.prototype.zip = function<T, U>(
    this: T[], ❷
    list: U[]
): [T, U][] {
    return this.map((v, k) =>
        tuple(v, list[k]) ❸
    )
}
```

- ❶ 首先让 TypeScript 知道我们要为 `Array` 添加 `zip` 方法。我们利用接口合并特性（见 5.4.1 节）增强全局接口 `Array<T>`，为这个全局定义的接口添加 `zip` 方法。

这个文件没有显式导入或导出（意味着在脚本模式中，见 10.2.3 节），因此可以直接增强全局接口 `Array`。我们声明一个接口，与现有的 `Array<T>` 同名，TypeScript 将负责合并二者。如果文件在模块模式中（如果实现 `zip` 需要导入其他代码，便是这种情况），就要把全局扩展放在 `declare global` 类型声明中（见 11.1 节）：

```
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（在模块模式中无需导入就能使用任何值，见第 10 章），可以增强模块模式文件中全局作用域内的名称。

- ❷ 然后在 `Array` 的原型上实现 `zip` 方法。这里使用 `this` 类型，以便让 TypeScript 正确推导出调用 `.zip` 方法的数组的类型 `T`。
- ❸ 由于 TypeScript 推导出的映射函数的返回类型是 `(T | U)[]`（TypeScript 没那么智能，意识不到这个元组的 0 索引始终是 `T`、1 索引始终是 `U`），

所以我们使用 `tuple` 函数（见 6.4.1 节）创建一个元组类型，而不使用类型断言。

注意，我们声明的 `interface Array<T>` 是对全局命名空间 `Array` 的增强，影响整个 TypeScript 项目，即便没有导入 `zip.ts` 文件，在 TypeScript 看来，`[].zip` 方法也可用。但是，为了增强 `Array.prototype`，我们要确保用到 `zip` 方法的文件都已经加载了 `zip.ts` 文件，这样才能让 `Array.prototype` 上的 `zip` 方法生效。那么，如何才能保证使用 `zip` 方法的文件已经加载了 `zip.ts` 文件呢？

简单，编辑 `tsconfig.json` 文件，把 `zip.ts` 排除在项目之外，这样使用方就必须先使用 `import` 语句将其导入：

```
{
  *exclude*: [
    "./zip.ts"
  ]
}
```

现在可以随心使用 `zip` 方法了，而且完全安全：

```
import './zip'

[1, 2, 3]
  .map(n => n * 2)      // number[]
  .zip(['a', 'b', 'c'])  // [number, string][]
```

运行上述代码，首先映射，然后拼凑，得到的结果如下：

```
[
  [2, 'a'],
  [4, 'b'],
  [6, 'c']
]
```

6.9 小结

本章涵盖了 TypeScript 最为高级的特性，从型变、基于流的类型推导、类型

细化、类型拓宽、全面性检查，到映射类型和条件类型，不一而足。然后讨论了处理类型的几个高级模式，包括模拟名义类型的类型烙印、利用条件类型的分配性质在类型层面操作类型，以及安全地扩展原型。

如果你没有理解或记住全部内容，不用担心。在你苦思冥想，不知如何以安全的方式表达时，请把本章当做一份参考手册，经常翻一翻。

6.10 练习题

1. 判断下述各组类型中第一个类型是否可赋值给第二个类型，并说出原因。
如果不确定，请回想一下子类型和型变，对照本章开头介绍的相关规则（倘若还是不确定，输入到代码编辑器中检查）：
 - a. 1 和 number
 - b. number 和 1
 - c. string 和 number | string
 - d. boolean 和 number
 - e. number[] 和 (number | string)[]
 - f. (number | string)[] 和 number[]
 - g. {a: true} 和 {a: boolean}
 - h. {a: {b: [string]}} 和 {a: {b: [number | string]}}
 - i. (a: number) => string 和 (b: number) => string
 - j. (a: number) => string 和 (a: string) => string
 - k. (a: number | string) => string 和 (a: string) => string
 - l. E.X(在枚举中定义 enum E {X = 'X'}) 和 F.X(在枚举中定义 enum F {X = 'X'})
2. 给定对象类型 type O = {a: {b: {c: string}}}, keyof O 是什么类型？
O['a']['b'] 呢？

3. 编写一个 `Exclusive<T, U>` 类型，计算在 T 或 U 中，但不同时在二者中的类型。例如，`Exclusive<1 | 2 | 3, 2 | 3 | 4>` 的结果应为 `1 | 4`。写出类型检查器求值 `Exclusive<1 | 2, 2 | 4>` 的每一步。
 4. 重写 6.6.3 节中的示例，不要使用明确赋值断言。

处理错误

一辆车疾驶在陡峭的高山隘口中，车上坐着一位物理学家、一位结构工程师和一位程序员。突然，刹车失灵，汽车行驶得越来越快。前遇弯路，幸好路边有防撞护栏，磕磕碰碰，这才使车没有跌落山崖。本以为就要命丧此地，可天无绝人之路，一条逃生通道映入眼帘。急转之下，他们拐入逃生通道，汽车平稳地停了下来。

物理学家说：“我们要模拟刹车片的摩擦力，分析由此产生的温升，尝试找出刹车失效的原因。”

结构工程师说：“后备厢里好像有几个扳手，我下车看看到底出了什么问题。”

程序员说：“我们为什么不试试是否能复现这次事故呢？”

— 佚名

TypeScript 竭尽所能，把运行时异常转移到编译时。TypeScript 是功能丰富的类型系统，加上强大的静态和符号分析能力，包揽了大量辛苦的工作，周五晚上我们不用再拖着疲惫的身体调试拼写错误的变量和空指针异常（值班的同事也不会因此而未能准时参加姑妈的生日晚宴）。

然而，不管使用什么语言，总是有一些异常会在运行时暴露出来。TypeScript 虽能尽职尽责，但有些问题是无法避免的，例如网络和文件系统异常、解析用户输入时出现的错误、堆栈溢出及内存不足。不过，TypeScript 的类型系统足够强大，提供了很多处理运行时错误的方式，不会眼看程序崩溃。

本章讨论 TypeScript 为表示和处理错误而提供的一些最为常用的模式：

- 返回 null。
- 抛出异常。
- 返回异常。
- Option 类型。

具体使用哪种机制由你自己决定，当然也要看应用的需求。在介绍每种错误处理机制时，笔者还会指出各自的优缺点，为你的选择提供依据。

7.1 返回 null

我们来编写一个程序，把用户输入的生日解析为 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())
```

用户输入的日期要做验证，毕竟我们只要求用户输入文本：

```
// ...
function parse(birthday: string): Date | null {
    let date = new Date(birthday)
    if (!isValid(date)) {
        return null
    }
    return date
}

// 检查指定的日期是否有效
function isValid(date: Date) {
```

```
return Object.prototype.toString.call(date) === '[object Date]'  
    && !Number.isNaN(date.getTime())  
}
```

我们强制要求先检查结果是否为 `null`, 然后再使用:

```
// ...  
let date = parse(ask())  
if (date) {  
    console.info('Date is', date.toISOString())  
} else {  
    console.error('Error parsing date for some reason')  
}
```

考虑到类型安全, 返回 `null` 是处理错误最为轻量的方式。有效的用户输入得到一个 `Date` 对象, 无效的用户输入得到一个 `null`, 类型系统会自动检查我们有没有涵盖这两种情况。

然而, 这么做丢失了一些信息, `parse` 函数没有指出操作到底为什么失败, 负责调试的工程师要梳理日志才能找出原因, 用户看到弹出窗口中显示的是“解析日期出错, 原因未知”, 而不是提示如何改正的消息, 例如“请输入 YYYY/MM/DD 格式的日期”。

返回 `null` 也不利于程序的编写, 每次操作都检查结果是否为 `null` 太烦琐, 不利于嵌套和串联操作。

7.2 抛出异常

把返回 `null` 改成抛出异常方便处理具体的问题, 这样能获取关于异常的元数据, 便于调试。

```
// ...  
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，使用上述代码时要捕获异常，以优雅的方式处理，免得整个应用崩溃：

```
// ...
try {
    let da
    consol
} catch
    consol
}
```

此外，也可以再次抛出其他异常，不放过任何一个错误：

```
// ...
try {
    let date =
        console.i
} catch (e)
if (e ins
    console
} else {
    throw e
}
}
```

我们可以使用错误的子类，更具体地表示问题的种类。如果其他工程师修改了 `parse` 或 `ask` 函数，抛出其他 `RangeError`，我们可以更好地区分：

11 / 11

```
// 自定义错误类型  
class InvalidDateFormatError 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
    }
}

```

不错，我们不仅表明有什么地方出错了，还通过一个自定义的错误指出了失败的原因。抛出这些错误之后，加上服务器日志的辅助，我们可以更好地调试问题，还可以显示恰当的对话框，告诉用户什么地方做错了、该采取什么措施修正。另外，我们可以在一个 try/catch 结构中串联和嵌套一系列操作（不用像返回 null 时那样检查每个操作是否失败）。

这样的代码又该如何使用呢？假如我们把一大段 try/catch 结构放在一个文件中，余下的代码在一个库中，从别的地方导入。那么，工程师怎么知道要捕获什么类型的错误呢（`InvalidDateFormatError` 和 `DateIsInTheFutureError`），或者直接检查常规的 `RangeError` 错误（注意，TypeScript 的函数签名中不含异常信息）？我们可以在函数的名称中指明（`parseThrows`），也可以在 docblock 中注明：

```

/**
 * @throws {InvalidDateFormatError} The user entered their birthday incorrectly.
 * @throws {DateIsInTheFutureError} The user entered a birthday in the future.
 */
function parse(birthday: string): Date {
    // ...
}

```

可是实际上，工程师可能不会把代码放在 try/catch 结构中，根本不检查异常，

毕竟工程师都是懒的（至少笔者是），而且类型系统不会指出缺少了什么情况要做处理。然而，有时（例如本例）下游代码希望我们处理异常，以免导致程序崩溃。

如果想告诉使用方，成功和出错两种情况都要处理，我们还能做什么呢？

7.3 返回异常

TypeScript 与 Java 不同，不支持 `throws` 子句。^{注1} 不过我们可以使用并集类型近似实现这个特性：

```
// ...
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，使用方被迫处理所有情况，即 `InvalidDateFormatError`、`DateIsInTheFutureError` 和成功解析，否则在编译时将报错 `TypeError`：

```
// ...
let result = parse(ask()) // 返回一个日期或错误
if (result instanceof InvalidDateFormatError) {
  console.error(result.message)
} else if (result instanceof DateIsInTheFutureError) {
  console.info(result.message)
} else {
  console.info('Date is', result.toISOString())
}
```

注 1：以防你没用过 Java，笔者告诉你，`throws` 子句的作用是指出一个方法可能抛出什么运行时异常，让使用方知道该处理哪些异常。

这里，我们利用 TypeScript 的类型系统实现了以下三点：

- 在 `parse` 函数的签名中加入可能出现的异常。
- 告诉使用方可能抛出哪些异常。
- 迫使使用方处理（或再次抛出）每一个异常。

使用方太懒的话，可以不用逐一处理各个异常，但是要明确写出来：

```
// ...
let result = parse(ask()) // 一个日期或错误
if (result instanceof Error) {
  console.error(result.message)
} else {
  console.info('Date is', result.toISOString())
}
```

当然，程序可能还是会在内存不足或堆栈溢出时崩溃，但是对此我们却无能为力。

这种方式比较轻量，不需要什么精巧的数据结构，但又提供了足够的信息，让使用方知道错误表示的是什么问题，在需要进一步了解情况时也知道该搜索什么。

这种方式的缺点是串联和嵌套可能出错的操作时容易让人觉得烦琐。如果一个函数返回 `T | Error1`，那么使用该函数的函数有两个选择：

1. 显式处理 `Error1`。
2. 处理 `T`（成功的情况），把 `Error1` 传给使用方处理。然而这样传来传去，使用方要处理的错误将越来越多：

```
function x(): T | Error1 {
  // ...
}
function y(): U | Error1 | Error2 {
  let a = x()
  if (a instanceof Error) {
```

```
    return a
  }
  // 对 a 执行一定的操作
}
function z(): U | Error1 | Error2 | Error3 {
  let a = y()
  if (a instanceof Error) {
    return a
  }
  // 对 a 执行一定的操作
}
```

这种方式确实烦琐，但却极大地保证了安全。

7.4 Option 类型

除此之外，还可以使用专门的数据类型描述异常。这种方式与返回值和错误的并集相比是有缺点的（尤其是与不使用这些数据类型的代码互操作时），但是却便于串联可能出错的操作。在这方面，常用的三个选项是 `Try`、`Option`^{注2} 和 `Either` 类型。本章只介绍 `Option` 类型，其他两个类型本质上基本相同。^{注3}



注意，`Try`、`Option` 和 `Either` 与 `Array`、`Error`、`Map` 或 `Promise` 不同，不是 JavaScript 环境内置的数据类型。如果想使用，要在 NPM 中寻找实现，或者自己编写。

`Option` 类型源自 Haskell、OCaml、Scala 和 Rust 等语言，隐含的意思是，不返回一个值，而是返回一个容器，该容器中可能有一个值，也可能没有。这个容器有一些方法，即使没有值也能串联操作。容器几乎可以是任何数据结构，只要能在里面存放值。例如，可以使用数组作为容器：

注 2：也叫 `Maybe` 类型。

注 3：如果想进一步了解另外两种类型，请搜索“try type”和“either type”。

```
// ...
function parse(birthday: string): Date[] {
  let date = new Date(birthday)
  if (!isValid(date)) {
    return []
  }
  return [date]
}

let date = parse(prompt())
date
  .map(_ => _.toISOString())
  .forEach(_ => console.info('Date is', _))
```



你可能注意到了，Option 的缺点与返回 null 一样，只告诉使用方什么地方出错了，而未说明出错的原因。

Option 真正发挥作用是在一次执行多个操作，而每个操作都有可能出错。

例如，之前我们一直假定 prompt 始终成功，而 parse 可能失败。但是，如果 prompt 也有可能失败呢？比如说，用户撤销了生日输入提示。这也是错误，我们不能放之不理，继续计算。此时，仍然可以使用 Option。

```
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，出错了。这是因为我们把一个 Date 数组（Date[]）映射到了一个 Date 数组构成的数组上（Date[][]），解决方法是在操作之前整平数组：

```

flatten(asn())
  .map(parse)
  .map(date => date.toISOString())
  .foreach(date => console.info('Date is', date))

// 把数组构成的数组整平为常规数组
function flatten<T>(array: T[][]): T[] {
  return Array.prototype.concat.apply([], array)
}

```

这样有点不便。`Option` 类型没有告诉我们什么信息（一切都是常规的数组），很难一眼看出这段代码在做什么。为了改变这种局面，我们可以把整个过程（把一个值放入容器，提供操作那个值的方式，提供从容器中取回结果的方式）包装成一个特殊的数据结构，让一切一目了然。实现好以后，我们可以像下面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')

```

我们将采用下述方式定义 `Option` 类型：

- `Option` 是一个接口，实现两个类：`Some<T>` 和 `None`（见图 7-1）。这是两种 `Option`。`Some<T>` 是包含一个 `T` 类型值的 `Option`，`None` 是没有值的 `Option`，表示失败。
- `Option` 既是类型也是函数。作为类型，它是一个接口，表示 `Some` 和 `None` 的超类型。作为函数，它是创建 `Option` 类型值的方式。

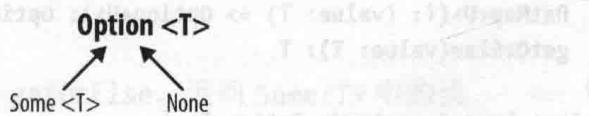


图 7-1: `Option<T>` 一分为二：`Some<T>` 和 `None`

先拟出类型：

```
interface Option<T> {} ①
class Some<T> implements Option<T> { ②
    constructor(private value: T) {}
}
class None implements Option<never> {} ③
```

- ① Option<T> 是一个接口，Some<T> 和 None 都可以实现该接口。
- ② Some<T> 表示操作成功，得到一个值。与前面使用的数组一样，Some<T> 是该值的容器。
- ③ None 表示操作失败，不包含值。

这几个类型等效于下述通过数组实现的 Option：

- Option<T> 是 [T] | []。
- Some<T> 是 [T]。
- None 是 []。

我们能对 Option 做些什么呢？就目前的实现，我们只定义两个操作：

flatMap

串联操作可能为空的 Option。

getOrElse

从 Option 中取回值。

首先在 Option 接口中定义这两个操作，然后在 Some<T> 和 None 中具体实现：

```
interface Option<T> {
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
}
class Some<T> extends Option<T> {
    constructor(private value: T) {}
}
class None extends Option<never> {}
```

可以看出：

- flatMap 接受一个函数 f，该函数接受一个 T 类型的值（Option 内的值的类型），返回一个内含 U 类型值的 Option。
- getOrElse 接受一个默认值，其类型与 Option 中值的类型相同，也是 T，返回的值为默认值（Option 为不含值的 None）或 Option 中的值（Option 为 Some<T>）。

在类型的指引之下，下面分别在 Some<T> 和 None 中实现这两个方法：

```
interface Option<T> {  
    flatMap<U>(f: (value: T) => Option<U>): Option<U>  
    getOrElse(value: T): T  
}  
  
class Some<T> implements Option<T> {  
    constructor(private value: T) {}  
    flatMap<U>(f: (value: T) => Option<U>): Option<U> { ❶  
        return f(this.value)  
    }  
    getOrElse(): T { ❷  
        return this.value  
    }  
}  
  
class None implements Option<never> {  
    flatMap<U>(): Option<U> { ❸  
        return this  
    }  
    getOrElse<U>(value: U): U { ❹  
        return value  
    }  
}
```

- ❶ 在 Some<T> 类型的值上调用 flatMap 时，传入函数 f，使用 Some<T> 中的值产出一个新类型的 Option。
- ❷ 在 Some<T> 类型的值上调用 getOrElse，返回 Some<T> 中的值。
- ❸ 由于 None 表示操作失败，因此在 None 上调用 flatMap 始终返回一个 None 值，毕竟操作失败后无法再恢复（至少对我们实现的 Option 来说是这样）。
- ❹ 在 None 类型的值上调用 getOrElse 始终返回传给 getOrElse 的值。

其实，上述实现有点简单，我们可以更进一步更好地区分两种类型。给定一个 Option 类型的值和一个把 T 变成 Option<U> 的函数，flatMap 始终把 Option<T> 值映射为 Option<U> 值。然而，如果我们事先知道是 Some<T> 值或 None 值，那就可以更具体一些。

在两种 Option 上调用 flatMap 预期得到的结果类型见表 7-1。

表 7-1：从 Some<T> 和 None 上调用 .flatMap(f) 的结果

	从 Some<T>	从 None
到 Some<U>	Some<U>	None
到 None	None	None

也就是说，映射 None 值始终得到 None 值，而映射 Some<T> 值得到 Some<T> 值或 None 值，具体取决于 f 返回什么。鉴于此，下面通过重载的签名名为 flatMap 指定更具体的类型：

```
interface Option<T> {
    flatMap<U>(f: (value: T) => None): None
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
}

class Some<T> implements Option<T> {
    constructor(private value: T) {}
    flatMap<U>(f: (value: T) => None): None
    flatMap<U>(f: (value: T) => Some<U>): Some<U>
    flatMap<U>(f: (value: T) => Option<U>): Option<U> {
        return f(this.value)
    }
    getOrElse(): T {
        return this.value
    }
}

class None implements Option<never> {
    flatMap(): None {
        return this
    }
    getOrElse<U>(value: U): U {
        return value
    }
}
```

```
    }
}
```

马上完成了，现在只剩下用于新建 Option 值的 Option 函数未实现了。我们已经通过接口实现了 Option 类型，接下来要实现一个同名函数（还记得吗，TypeScript 中的类型和值位于独立的命名空间里），用于创建 Option 值，具体做法与 6.3.4 节一样。如果用户传入 null 或 undefined，返回 None 值；否则，返回 Some 值。同样，我们依然通过重载签名实现这一点：

```
function Option<T>(value: null | undefined): None ①
function Option<T>(value: T): Some<T> ②
function Option<T>(value: T): Option<T> { ③
    if (value == null) {
        return new None
    }
    return new Some(value)
}
```

- ① 调用 Option 时，如果使用方传入 null 或 undefined，返回一个 None 值。
- ② 否则，返回一个 Some<T> 值，其中 T 是用户传入值的类型。
- ③ 最后，我们手动为这两个重载的签名计算上限。null | undefined 和 T 的上限是 T | null | undefined，简化为 T。None 和 Some<T> 的上限是 None | Some<T>，即我们定义的 Option<T>。

一切就绪。我们得到了一个完全可用的精简版 Option 类型，现在可以安全地操作可能为 null 的值了。用法如下：

```
let result = Option(6) // Some<number>
    .flatMap(n => Option(n * 3)) // Some<number>
    .flatMap(n => new None) // None
    .getOrElse(7) // 7
```

回到要求用户输入生日的示例，现在代码能像预想中那样运行了：

```
ask()
    .flatMap(parse) // Option<string>
                    // Option<Date>
```

```
.flatMap(date => new Some(date.toISOString())) // Option<string>
.flatMap(date => new Some('Date is ' + date)) // Option<string>
.getOrElse('Error parsing date for some reason') // string
```

对一系列可能成功也可能失败的操作，`Option` 是一种强有力地执行方式，不仅保证了类型安全性，还通过类型系统向使用方指出了某个操作可能失败。

然而，`Option` 也不是没有缺点。`Option` 通过一个 `None` 值表示失败，没有关于失败的详细信息，也不知道失败的原因。另外，与不使用 `Option` 的代码无法互操作（要自己动手包装这些 API，让它们返回 `Option`）。

尽管如此，本节实现的特性还是很棒。我们所做的重载，在多数语言中是无法实现的，即便是依靠 `Option` 类型处理空值的语言也不行。我们通过重载的调用签名限制 `Option` 只能为 `Some` 或 `None`，这样写出的代码安全得多，让 Haskell 程序员十分羡慕。现在，去开一瓶冰啤酒吧，这是你应得的。

7.5 小结

本章介绍了在 TypeScript 中表示和处理错误的几种不同方式，包括返回 `null`、抛出异常、返回异常和 `Option` 类型。现在，遇到失败的操作时我们有很多方案可选，而且都兼顾安全。具体选择哪种方案取决于你自己，但是要考虑以下事项：

- 你只想表示有操作失败了 (`null`、`Option`)，还是想进一步指出失败的原因（抛出异常和返回异常）。
- 你想强制要求使用方显式处理每一个可能出现的异常（返回异常），还是尽量少编写处理错误的样板代码（抛出异常）。
- 你想深入分析错误 (`Option`)，还是在遇到错误时做简单的处理 (`null`、异常)。

7.6 练习题

使用本章介绍的某种模式为下述 API 设计一种处理错误的方式。这个 API 中的每个操作都可能失败，请根据需要更新 API 中的方法签名，允许操作失败（或者不允许）。试想在一系列操作中该如何处理可能遇到的错误（例如，先获取已登录用户的 ID，然后列出用户的好友，最后获取各好友的名字）。

```
class API {  
    getLoggedInUserID(): UserID  
    getFriendIDs(userID: UserID): UserID[]  
    getUserName(userID: UserID): string  
}
```

异步编程、并发和并行

目前，本书讨论的几乎都是同步程序，接收输入，执行操作，然后结束。然而现实可不这么简单，网络请求、数据库和文件系统交互、用户操作的响应，以及在单独的线程中执行 CPU 密集型运算，都要用到异步 API，比如说回调、promise 和流。

这些异步任务才能真正突显 JavaScript 的优势，才是与其他主流多线程语言（如 Java 和 C++）与众不同的特色。V8 和 SpiderMonkey 等流行的 JavaScript 引擎使用一个线程完成传统上需要多个线程执行的任务，这些引擎十分聪明，在一个线程中多路复用任务，而其他任务则处于空闲状态。这种事件循环(event loop)是 JavaScript 引擎的标准线程模型，本书也假定你使用的是这样的引擎。对终端用户来说，引擎采用事件循环模型还是多线程模型往往无关紧要，但是对下文要讲的运作原理和设计动机却有一定的影响。

JavaScript 采用事件循环式并发模型，规避了多线程编程令人深恶痛绝的顽疾，也不用再去辨别同步式数据类型、互斥对象、计数信号量等多线程术语了。即便使用多线程运行 JavaScript，也很少使用共享内存，常见的做法是通过消息在线程之间传递和序列化数据。这种设计容易让人想到 Erlang、actor 系统，以及其他纯函数式并发模型，也正是因为这样的设计，JavaScript 的多线程编程才十分简单易用。

尽管如此，异步编程还是会让程序难以理解。我们不能一行行分析程序，而要知道代码在何处暂停，移到其他地方执行，又在何时返回原处。

TypeScript 为理解异步程序提供了便利的工具，通过类型可以追踪异步操作，借助内置的 `async/await` 可以把熟悉的同步思想应用到异步程序上。使用 TypeScript 还可以为多线程程序指定严格的消息传递协议（听着复杂，其实很简单）。最坏的情况下，如果同事编写的异步代码太过复杂，让你调试到很晚，TypeScript 也能给你一些宽慰（当然，要通过一个编译器标志开启）。

在动手编写异步程序之前，我们要讨论一下异步在现代的 JavaScript 引擎中具体是如何运作的，弄清楚为什么在看似单个的线程中可以暂停和恢复执行。

8.1 JavaScript 的事件循环

先看一个例子。我们设置两个计数器，一个在 1 毫秒后触发，另一个在 2 毫秒后触发：

```
setTimeout(() => console.info('A'), 1)
setTimeout(() => console.info('B'), 2)
console.info('C')
```

那么，控制台中将输出什么内容呢？是 A、B、C 吗？

JavaScript 程序员一眼就能看出，真正的触发顺序是 C、A 和 B。没用过 JavaScript 或 TypeScript 的人对这个行为可能会感到奇怪，觉得不符合直觉。其实，这段代码十分容易理解，只是与 C 语言中 `sleep` 采用的并发模型，以及 Java 把作业调度到另一个线程中的方式有所不同。

概括地说，JavaScript VM 采用下述方式模拟并发（见图 8-1）：

- JavaScript 主线程调用 `XMLHttpRequest`（处理 Ajax 请求）、`setTimeout`（休眠一段时间）、`readFile`（从磁盘中读取文件）等异步 API。这些 API 由 JavaScript 平台提供，我们自己不能创建。^{注 1}

注 1：也不是彻底不行，你可以复刻浏览器平台，或者使用 C++ 构建 NodeJS 扩展。

- 调用原生的异步 API 后，控制权返回主线程，继续向下执行，就像从未调用异步 API 一样。
- 异步操作执行完毕后，平台在事件队列中添加一个任务。每个线程都有自己的队列，异步操作的结果就通过队列发回主线程。任务中有关于调用的元信息，还有主线程中回调函数的引用。
- 主线程的调用堆栈清空后，平台将检查事件队列中有没有待处理的任务。如果有等待处理的任务，平台着手处理，触发一个函数调用，把控制权返还给主线程中的那个函数。调用那个函数之后，如果调用堆栈又变空了，平台再次检查事件队列中有没有可以处理的任务了。这个循环一直运转下去，直到调用堆栈和事件队列都为空，而且所有原生的异步 API 调用都已结束。

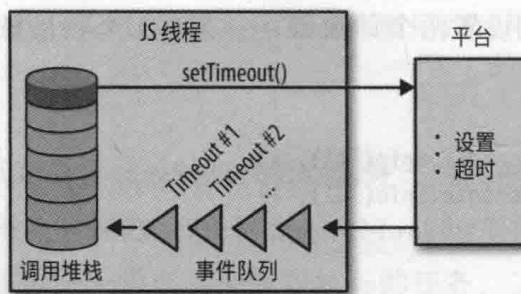


图 8-1：JavaScript 的事件循环：调用异步 API 的过程

了解这些信息之后，我们回到前面的 `setTimeout` 示例。那段代码的执行过程是这样的：

1. 调用 `setTimeout` 函数，调用原生的超时 API，传入一个回调引用和参数 1。
2. 再次调用 `setTimeout` 函数，再次调用原生的超时 API，传入另一个回调引用和参数 2。
3. 在控制台中输出 C。
4. 在背后，至少 1 毫秒之后，JavaScript 平台在事件队列中添加一个任务，指明第一个 `setTimeout` 调用的超时时间到了，现在可以调用指定的回调了。

5. 又过了 1 毫秒，平台又在事件队列中添加一个任务，调用第二个 `setTimeout` 的回调。
6. 由于调用堆栈为空，执行完第 3 步之后，平台查看事件队列中还有没有任务。如果执行到第 4 步和（或）第 5 步，队列中有任务，平台将调用相应的回调函数。
7. 等两个计数器都到点后，事件循环和调用堆栈均为空，程序退出。

正是这样，那段代码才输出 C、A、B，而不是 A、B、C。掌握这些知识之后，我们可以开始讨论如何以类型安全的方式编写异步代码了。

8.2 处理回调

JavaScript 异步程序的核心基础是回调。回调其实就是常规的函数，只是作为参数传给另一个函数。就像在同步程序中一样，另一个函数在做完操作（处理网络请求等）之后调用回调函数。异步代码调用的回调也是函数，而且类型签名中没有标明函数是异步调用的。

NodeJS 的原生 API，例如 `fs.readFile`（采用异步方式从磁盘中读取文件的内容）和 `dns.resolveCname`（采用异步方式解析 CNAME 记录），按照约定，回调的第一个参数是错误或 `null`，第二个参数是结果或 `null`。

`readFile` 的类型签名如下所示：

```
function readFile(  
  path: string,  
  options: {encoding: string, flag?: string},  
  callback: (err: Error | null, data: string | null) => void  
): void
```

注意，`readFile` 和 `callback` 的类型没有什么特别之处，都是常规的 JavaScript 函数。签名中没有特别标明 `readFile` 是异步的，也没有指出在调用 `readFile` 之后控制权会立即传给下一行代码（不等待调用 `readFile` 的结果）。



如果你想自己运行下述示例，请先安装 NodeJS 的类型声明：

```
npm install @types/node --save-dev
```

关于第三方类型声明的说明，见 11.4.2 节。

举个例子，下面我们编写一个 NodeJS 程序，读写 Apache 访问日志：

```
import * as fs from 'fs'

// 从 Apache 服务器的访问日志中读取数据
fs.readFile(
  '/var/log/apache2/access_log',
  {encoding: 'utf8'},
  (error, data) => {
    if (error) {
      console.error('error reading!', error)
      return
    }
    console.info('success reading!', data)
  }
)

// 采用并发方式把数据写入该访问日志
fs.appendFile(
  '/var/log/apache2/access_log',
  'New access log entry',
  error => {
    if (error) {
      console.error('error writing!', error)
    }
  }
)
```

如果你不是 TypeScript 或 JavaScript 工程师，对 NodeJS 的内置 API 不太了解，不知道这些 API 是异步处理的，不能把 API 在代码中调用的顺序理解为执行文件系统的操作顺序，那就肯定发现不了这里有个小问题：`readFile` 虽然在前面调用，但是读取出来的访问日志可能没有后面新增的那行日志，具体有没有要看运行这段代码时文件系统有多繁忙。

你可能知道 `readFile` 是异步的，也许是根据自己的经验判断的；也许是你看

过 NodeJS 文档；也许是你知道，按照 NodeJS 的一般约定，如果函数的最后一个参数是一个接受两个参数的函数，而且顺序为 `Error | null` 和 `T | null` 类型，那么这个函数通常是异步的；也许是你在大厅遇到邻居，想借一块糖，闲聊几句后谈到了 NodeJS 异步编程，邻居说他几个月前遇到过类似地问题，还告诉了你他是如何解决的。

无论你是如何知道的，从类型上肯定是看不出来。

不光是从类型上看不出函数是否异步执行，回调也难以理清，这往往会导致人们口中所说的“回调金字塔”：

```
async1((err1, res1) => {
  if (res1) {
    async2(res1, (err2, res2) => {
      if (res2) {
        async3(res2, (err3, res3) => {
          // ...
        })
      }
    })
  }
})
```

按顺序执行的操作通常是一环扣一环的，前一步成功才执行下一步，除非遇到错误。使用回调时，我们要自己动手维护执行的顺序。如果还想像同步操作那样处理错误（例如，NodeJS 的习惯做法是在参数类型有误时抛出异常，而不把 `Error` 对象传给回调），按一定顺序执行的回调很容易出错。

有序操作只是我们想借助异步任务执行的一种操作，此外我们可能还想并行运行几个函数，获知全部函数何时运行完毕，或者让几个函数竞争，只获取第一个结束的函数返回的结果等。

这是传统回调的局限所在。如果不对异步任务的执行做更复杂的抽象，使用相互之间有依赖关系的多个回调很容易让人身陷囹圄。

综上得出，

- 使用回调可执行简单的异步任务。
- 虽然回调适合处理简单的任务，但是如果异步任务变多，很容易变成一团乱麻。

8.3 promise：让一切回到正轨

幸好我们不是遇到这些限制的第一批程序员。本节说明 promise，这个概念对异步任务进行抽象，方便编排任务、排列任务等。即使你以前用过 promise 或 future，通过本节也能更好地理解它们的原理。



多数现代的 JavaScript 平台都内建对 promise 的支持。本节，我们将手动实现 Promise（部分功能）；不过在实际开发中，你应该使用内建的版本或别人实现好的版本。如果想知道你使用的平台是否支持 promise，请访问 <http://bit.ly/2uMxkk5>。在原生不支持 promise 的平台中，可以借助腻子脚本变相使用，详见“lib”一节。

下面举个例子，指出我们想如何使用 Promise：先向文件中添加一些内容，然后再把文件中的内容读取出来。

```
function appendAndReadPromise(path: string, data: string): Promise<string> {
  return appendPromise(path, data)
    .then(() => readPromise(path))
    .catch(error => console.error(error))
}
```

注意，这里没有回调金字塔，我们把想执行的一系列异步任务变成易于理解的线性链：前一个任务成功后才能执行下一个任务，倘若失败，则跳到 catch 子句。假如是基于回调的 API，那么写出的代码可能是下面这样：

```
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)
  })
})
}
```

针对这个设想，下面我们动手实现 Promise API。

开头很简单：

```
class Promise {
```

```
}
```

`new Promise` 接受一个函数，我们称之为执行器（executor）。在 `Promise` 的实现中，执行器接受两个参数，一个是 `resolve` 函数，另一个是 `reject` 函数。

```
type Executor = (
  resolve: Function,
  reject: Function
) => void
```

```
class Promise {
  constructor(f: Executor) {}
```

```
}
```

那么，`resolve` 和 `reject` 是如何运作的呢？下面通过示例说明一下。假设我们想把 NodeJS 中一个回调的 API（例如 `fs.readFile`）改造成基于 `Promise` 的 API。NodeJS 内置的 `fs.readFile` API 是这样使用的：

```
import {readFile} from 'fs'

readFile(path, (error, result) => {
  // ...
})
```

改用 `Promise` 实现之后，变成这样了：

```
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)
      }
    })
  })
}
```

可见，`resolve` 的参数是什么类型取决于具体使用的 API（这里，其参数的类型就是 `result` 的类型），而 `reject` 的参数始终是 `Error` 类型。因此，我们要更新实现，把不安全的 `Function` 类型替换为更具体的类型：

```
type Executor<T, E extends Error> = (
  resolve: (result: T) => void,
  reject: (error: E) => void
) => void
// ...
```

由于我们想一打眼就看出 `Promise` 的最终类型（比如 `Promise<number>` 表示结果为一个 `number` 的异步任务），所以要使用泛型，并在构造方法中把类型参数传给 `Executor`：

```
// ...
class Promise<T, E extends Error> {
  constructor(f: Executor<T, E>) {}
}
```

目前来看还不错，我们定义了 `Promise` 的构造方法的 API，还能看出具体的类型。接下来考虑串联：若想按顺序运行一系列 `Promise`，要开放哪些操作才能在 `Promise` 之间传递结果，并且捕获异常呢？回过头再看一下本节开头的代码，你会发现，我们需要的是 `then` 和 `catch`。下面把它们添加到 `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 以不同的方式排列 Promise：then 把成功从一个 Promise 获得的结果映射到一个新 Promise 上，^{注2}catch 则把错误映射到一个新 Promise 上，从被拒状态中走出来。

then 的使用方法如下：

```
let a: () => Promise<string, TypeError> = // ...
let b: (s: string) => Promise<number, never> = // ...
let c: () => Promise<boolean, RangeError> = // ...

a()
  .then(b)
  .catch(e => c()) // b 绝不抛出错误，因此这个子句只在 a 出错时运行
  .then(result => console.info('Done', result))
  .catch(e => console.error('Error', e))
```

b 的第二个参数是 never 类型（表示 b 永不抛出错误），第一个 catch 子句只在 a 出错时才调用。不过请注意，使用 Promise 时我们并不在意 a 可能抛出错误、b 绝不抛出错误，如果 a 成功，就把 Promise 映射到 b 上，否则跳到第一个 catch 子句，把 Promise 映射到 c 上。如果 c 成功，那就输出 Done；倘若失败，就再次交给 catch 子句。这个行为与 try/catch 语句很像，只不过 try/catch 处理的是同步任务，而这里处理的是异步任务（见图 8-2）。

注 2：心明眼亮的读者可能发现了，这个 API 与 7.4 节开发的 flatMap API 十分相似。这不是巧合！Promise 和 Option 都受函数式编程语言 Haskell 中的 Monad 设计模式影响。

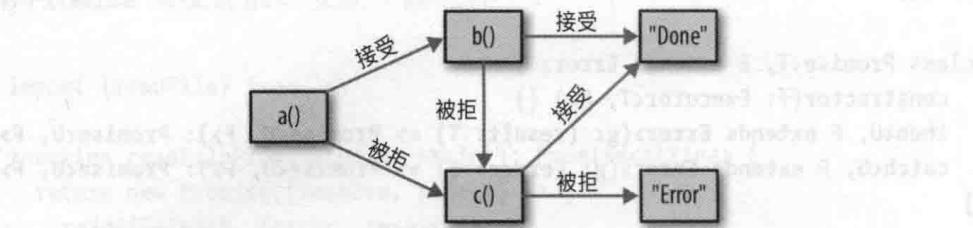


图 8-2: Promise 状态机

此外，我们还要处理 `Promise` 真正抛出异常的情况（例如 `throw Error('foo')`）。为此，在实现 `then` 和 `catch` 时要把代码放在 `try/catch` 结构中，在 `catch` 分支中处理被拒的情况。然而事情并没有这么简单，这里还涉及一些其他问题：

- `Promise` 对象都有可能被拒，而通过静态检查发现不了这一点（因为 TypeScript 不支持在函数的签名中指定可能抛出什么异常）。
- `Promise` 对象被拒不一定是因为有错误。TypeScript 别无选择，只能继承 JavaScript 的行为，而在 JavaScript 中，通过 `throw` 可以抛出一切，可以是一个字符串、一个函数、一个数组、一个 `Promise` 对象，不一定总是 `Error` 对象。我们不能认定被拒后返回的就是 `Error` 的子类型。我们无能为力，只能做点牺牲，免得使用方把每个 `promise` 链（可能分散在多个文件或模块中）都放在 `try/catch` 结构中。

考虑到这两点，我们要放宽对 `Promise` 类型的要求，不指定错误类型：

```

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> {
    // ...
  }
}
  
```

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

至此，Promise 接口完全设计好了。

then 和 catch 的具体实现留作练习。Promise 的实现需要一定的技巧，很容易出错。有追求的读者，如果有几小时空闲，可以看一下 ES2015 规范 (<http://bit.ly/2JT3KUh>)，了解 Promise 状态机的内部机理。

8.4 async 和 await

promise 对异步代码所做的抽象十分强大，而且广受欢迎，JavaScript 自身（当然包括 TypeScript）也有相应的句法：async 和 await。使用这种句法，可以像同步操作那样处理异步操作。



await 可以视作 .then 在语言层上的句法糖。使用 await 处理 Promise 对象时，要把相关的代码放在 async 块中。这种情况下不再使用 .catch，而是把 await 放在常规的 try/catch 块中。

以下述 promise 为例（前一节没有介绍 finally，其实它的行为与你想的一样，即在触发 then 和 catch 之后运行）：

```
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'))
}
```

使用 async 和 await 改写，要放在一个 async 函数中，然后再使用 await 获得 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')
}
}
```

`async` 和 `await` 是 JavaScript 特性，这里就不深入探讨了。你只需要知道，TypeScript 也支持这两个句法，而且对类型完全安全。处理 `promise` 对象时，如果想理清串联的操作，又不想写大量 `then`，就可以使用这两个句法。如果想深入了解 `async` 和 `await`，请访问 MDN 中的文档 (<https://mzl.la/2TJLFYt>)。

8.5 异步流

`promise` 对象是便于建模、排列和编排未来的值，但是如果多个值在未来的不同时刻产出呢？这种情况并不少见，比如从文件系统中读取文件、Netflix 服务器通过互联网向你的笔记本电脑中传送的视频流、填写表单时敲击的一系列键盘、一群好友陆续到你家参加晚宴，以及在总统竞选初选日投入票箱的选票。表面上看，这些事情没什么共同点，不过却都可以视作异步流，都是在未来某个时刻发生的一系列事件。

这样的操作有多种不同的建模方式，最为常见的是事件发射器（例如 NodeJS 的 `EventEmitter`）或响应式编程库（例如 RxJS，<https://www.npmjs.com/package/@reactivex/rxjs>）。^{注3} 这两种方式之间的区别就像回调和 `promise` 对象一样：事件简单、轻量，而响应式编程库更强大，可以编排和排列事件流。

事件发射器在下一节讨论。如果想进一步了解响应式编程，请阅读各响应

注3：响应式编程采用 `Observables` 在一段时间内处理多个值。目前有一个提案正在讨论标准化 `Observables` (<https://tc39.github.io/proposal-observable/>)。这个提案未来如果被 JavaScript 引擎广泛采纳，本书以后的版本将深入探讨 `Observables`。

式编程库的文档，例如 RxJS (<https://www.npmjs.com/package/@reactive/rxjs>)、MostJS (<https://github.com/mostjs/core>) 或 xstream (<https://www.npmjs.com/package/xstream>)。

事件发射器

概括来说，事件发射器提供的 API 用于在通道中发射事件，并监听该通道中的事件：

```
interface Emitter {  
    // 发送事件  
    emit(channel: string, value: unknown): void  
  
    // 发送事件后做些事情  
    on(channel: string, f: (value: unknown) => void): void  
}
```

事件发射器是 JavaScript 中一种常见的设计模式。在使用 DOM 事件、jQuery 事件或 NodeJS EventEmitter 模块的过程中，你可能遇到过。

在多数语言中，像这样的事件发射器是不安全的。原因在于，`value` 的类型由具体的 `channel` 而定，而在多数语言中，无法使用类型表示二者之间的关系。如果你使用的语言不同时支持重载函数签名和字面量类型，那就没法表达“这是在该通道中发射的某类型”。为了解决这个问题，经常采用宏，让宏生成发射事件的方法和监听各个通道的方法。然而，在 TypeScript 中，我们可以使用类型系统直接表述 `value` 的类型与 `channel` 之间的关系。

假设我们在使用 NodeRedis 客户端 (https://github.com/NodeRedis/node_redis)，这是内存数据存储器 Redis 的 Node API。具体方法如下：

```
import Redis from 'redis'  
  
// 创建一个 Redis 客户端实例  
let client = redis.createClient()
```

```
// 监听该客户端发射的几个事件
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))
```

作为使用这个 Redis 库的程序员，我们希望知道 on API 传给回调的参数是什么类型。可是，各参数的类型取决于 Redis 在哪个通道中发射事件，而不是具体某一个类型。假如我们是这个库的作者，为保安全，最简单的方式是使用重载类型：

```
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)
}
```

这样做没问题，但是有点啰嗦。下面使用映射类型（见 6.3.3 节）改写，把事件定义提出来，单声明一个类型 Events：

```
type Events = { ①
  ready: void
  error: Error
  reconnecting: {attempt: number, delay: number}
}

type RedisClient = { ②
  on<E extends keyof Events>(
    event: E,
    f: (arg: Events[E]) => void
  ): void
}
```

- ① 首先定义一个单独的对象类型，列举 Redis 客户端可能发射的每个事件，以及各事件的参数。
- ② 映射 Events 类型，告诉 TypeScript，调用 on 时可以传入前面定义的任何事件。

现在，我们可以使用这个类型提升 Node-Redis 库的安全性，尽量增加 `emit` 和 `on` 两个方法的安全：

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

把事件名称和参数提取到结构中，然后映射该结构，生成监听器和发射器，这是 TypeScript 代码常见的模式。这样写出的代码简洁，而且非常安全。像这样指定发射器的类型，你肯定不会拼错一个键、错误使用一个参数的类型，或者忘记传入参数。此外，使用代码的其他工程师还可以把这种方式当做文档，代码编辑器会给出建议的事件以及传入事件回调的参数类型。

现实中的发射器

使用映射类型构建对类型安全的事件发射器是很常见的做法。例如，在 TypeScript 标准库中，DOM 事件的类型就是这样声明的。`WindowEventMap` 是事件名称到事件类型的映射，`.addEventListener` 和 `.removeEventListener` API 都通过映射生成更为具体的事件类型，而不是默认的 `Event` 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
}
```

8.6 多线程类型安全

目前，我们讨论的异步程序基本上运行在一个 CPU 线程中，你编写的 JavaScript 和 TypeScript 程序大都如此。不过，一些 CPU 密集型任务可能需要并行，把一项任务分到多个线程中。这么做可能是为了提升速度，也可能是想让主线程空闲出来，继续响应后续操作。本节讨论编写安全的并行程序的几种模式，涵盖浏览器和服务器。

8.6.1 在浏览器中：使用 Web 职程

浏览器大都支持使用 Web 职程（worker）处理多线程。为免某些操作（例如 CPU 密集型任务）阻塞主线程，导致 UI 无响应，我们可以在 JavaScript 主线程中创建一些职程（严格受限的后台线程），把相关操作交给它们。Web 职程真正在浏览器中采用并行方式运行代码，而 `Promise` 和 `setTimeout` 等异步 API 只是并发运行代码。职程在另一个 CPU 线程中并行运行代码。Web 职程可以处理网络请求、文件系统写入等操作，不过有一些小限制。

Web 职程是浏览器提供的 API，设计人员对安全性提出了更高的要求。这里

说的“安全性”并不是我们一直追求的类型安全，而是内存安全。编写 C、C++、Objective-C 代码，或者在 Java 或 Scala 中做多线程编程的程序员肯定知道并发操作共享内存是多么困难。如果多个线程读写同一块内存，很容易遇到各种并发问题，例如不确定性、死锁等。

在浏览器中，代码必须安全，而且要降低崩溃的风险，以免导致糟糕的用户体验。鉴于此，主线程和 Web 职程之间，以及两个 Web 职程之间的主要通信手段是消息传递。



如果你想跟着本节的示例一起操作，需要告诉 TSC 你打算在浏览器中运行代码。方法是在 *tsconfig.json* 中加入 `dom` 库：

```
{  
  "compilerOptions": {  
    "lib": ["dom", "es2015"]  
  }  
}
```

在 Web 职程中运行的代码要使用 `webworker` 库：

```
{  
  "compilerOptions": {  
    "lib": ["webworker", "es2015"]  
  }  
}
```

如果 Web 职程脚本和主线程使用同一个 *tsconfig.json*，那就同时加入上述两个库。

消息传递 API 是这样使用的。首先，在主线程中派生一个 Web 职程：

```
// MainThread.ts  
let worker = new Worker('WorkerScript.js')
```

然后，把消息传给该进程：

```
// MainThread.ts  
let worker = new Worker('WorkerScript.js')
```

```
worker.postMessage('some data')
```

通过 `postMessage` API 几乎可以把任何数据传给另一个线程。^{注4}

主线程在把数据交给职程之前会克隆数据。^{注5} 在 Web 职程这一端，使用全局可用的 `onmessage` API 监听入站事件：

```
// WorkerScript.ts
onmessage = e => {
  console.log(e.data) // 输出 'some data'
}
```

如果想反过来通讯，由职程发消息给主线程，使用全局可用的 `postMessage` 把消息发给主线程，在主线程中使用 `.onmessage` 方法监听入站消息。两个方向的通信综合如下：

```
// 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 与 8.5.1 节讨论的事件发射器 API 很像。这样传递消息是简单，但是没有类型，无法确保正确处理了可能发送的所有消息类型。

注 4： 函数、错误、DOM 节点、属性描述符、读值方法、设值方法、原型方法和属性除外。详见 HTML5 规范 (<http://w3c.github.io/html/infrastructure.html#safe-passing-of-structured-data>)。

注 5： 还可以使用 Transferable API 在线程之间通过引用传递特定类型的数据（例如 `ArrayBuffer`）。本节不使用 Transferable 在线程之间显式转移对象的所有权，但这只是实现细节的区别。如果使用 Transferable，在类型安全层面是没有差别的。

其实，这个 API 就是事件发射器，因此我们可以采用前面的方式加入类型。举个例子，下面为一个聊天客户端构建一个简单的消息层，运行在职程中。这个消息层把更新推送给主线程，我们暂且不管错误处理、权限等。首先，定义一些入站和出站消息类型（主线程把 Commands 发给职程，职程把 Events 发给主线程）：

```
// 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]
}
```

那么如何把这些类型应用到 Web 职程的消息 API 中呢？最简单的方法之一是，定义一个并集类型，囊括所有消息类型，然后分别处理不同的消息类型。不过这样做十分麻烦。以 Command 类型为例，写出的代码如下所示：

```
// WorkerScript.ts
type Command = ①
| {type: 'sendMessageToThread', data: [ThreadID, Message]} ②
| {type: 'createThread', data: [Participants]}
| {type: 'addUserToThread', data: [ThreadID, UserID]}
| {type: 'removeUserFromThread', data: [ThreadID, UserID]}

onmessage = e => ③
processCommandFromMainThread(e.data)

function processCommandFromMainThread(④
    command: Command
```

```
) {  
  switch (command.type) { ⑤  
    case 'sendMessageToThread':  
      let [threadID, message] = command.data  
      console.log(message)  
      // ...  
  }  
}
```

- ❶ 为主线程可能发给进程的所有命令定义一个并集类型，并且指定各命令的参数。
- ❷ 这就是一个常规的并集类型。在定义较长的并集类型时，以管道符号 (|) 开头利于理解。
- ❸ onmessage API 没有类型信息，那就把消息委托给有类型信息的 processCommandFromMainThread API 处理。
- ❹ processCommandFromMainThread 对无类型信息的 onmessage API 进行包装，负责处理从主线程接收到的所有消息。
- ❺ Command 是可辨别的并集类型（见“辨别并集类型”一节），可以使用 switch 穷尽主线程发来的每一种消息。

下面，我们要对 Web 进程不那么直白的 API 进行抽象，使用熟悉的 EventEmitter API 包装，简化入站消息和出站消息类型。

首先，对 NodeJS 的 EventEmitter API（在浏览器中可通过 NPM 中的 events 包使用，<https://www.npmjs.com/package/events>）做一层类型安全包装：

```
import EventEmitter from 'events'  
  
class SafeEmitter<  
  Events extends Record<PropertyKey, unknown[]> ①  
 > {  
   private emitter = new EventEmitter ②  
   emit<K extends keyof Events>( ③  
     channel: K,  
     ...data: Events[K]  
   ) {
```

```
    return this.emitter.emit(channel, ...data)
}

on<K extends keyof Events>(④
  channel: K,
  listener: (...data: Events[K]) => void
) {
  return this.emitter.on(channel, listener)
}
}
```

- ❶ SafeEmitter 声明一个泛型 Events，以及一个从 PropertyKey (TypeScript 内置的类型，表示对象的有效键：string、number 或 Symbol) 到参数列表的 Record 映射。
- ❷ 把 emitter 声明为 SafeEmitter 的私有成员。我们没有扩展 SafeEmitter，因为 emit 和 on 的签名比在 EventEmitter 中更严格，而且作为参数的函数会逆变（记住，如果函数 a 可赋值给函数 b，那么 a 的参数必为 b 中相应参数的超集），TypeScript 不允许重载。
- ❸ emit 接受一个 channel，以及为 Events 类型定义的一组参数。
- ❹ 类似地，on 接受一个 channel 和一个 listener。listener 接受为 Events 类型定义的一组参数。

使用 SafeEmitter 可以极大地减少安全实现监听层所需的样板代码量。在前端端，我们把所有 onmessage 调用委托给发射器，向外开放一个便利且安全的监听器 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]
}

// 监听主线程发来的事件
let commandEmitter = new SafeEmitter <Commands>()

// 把事件发射回主线程
let eventEmitter = new SafeEmitter <Events>()

// 使用类型安全的事件发射器包装主线程发来的命令
onmessage = command =>
  commandEmitter.emit(
    command.data.type,
    ...command.data.data
  )

// 监听职程触发的事件，发给主线程
eventEmitter.on('receivedMessage', data =>
  postMessage({type: 'receivedMessage', data})
)
eventEmitter.on('createdThread', data =>
  postMessage({type: 'createdThread', data})
)
// 等等

// 回应主线程发来的 sendMessageToThread 命令
commandEmitter.on('sendMessageToThread', (threadID, message) =>
  console.log(OK, I will send a message to threadID ${threadID})
)

// 把事件发回主线程
eventEmitter.emit('createdThread', 123, [456, 789])

```

反过来，在主线程中也可以使用基于 `EventEmitter` 的 API 把命令发给职程。注意，如果你想在自己的代码中采用这种模式，应该使用功能更完善的发射器（例如 Paolo Fragomeni 开发的 `EventEmitter2`，<https://www.npmjs.com/package/eventemitter2>），最好支持通配监听器，这样就无需自己动手为每种事件添加监听器了。

```

// 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 <Commands>()
let eventEmitter = new SafeEmitter <Events>()

let worker = new Worker('WorkerScript.js')

// 监听职程发来的事件,
// 使用类型安全的事件发射器重新发射
worker.onmessage = event =>
    eventEmitter.emit(
        event.data.type,
        ...event.data.data
    )

// 监听该线程发出的命令, 把命令发给职程
commandEmitter.on('sendMessageToThread', data =>
    worker.postMessage({type: 'sendMessageToThread', data})
)
commandEmitter.on('createThread', data =>
    worker.postMessage({type: 'createThread', data})
)
// 等等

// 职程报告创建了新线程时做些事情
eventEmitter.on('createdThread', (threadID, participants) =>
    console.log('Created a new chat thread!', threadID, participants)
)

// 把命令发给职程
commandEmitter.emit('createThread', [123, 456])

```

对 Web 职程的讨论到此结束！本节对我们熟悉的事件发射器进行了类型安全包装，这样包装后可在多个场合下使用，例如处理浏览器中的光标事件及在

线程之间安全地通信。如果发现某个行为不安全，可以做一层包装，提供类型安全的 API，这是 TypeScript 编程常见的模式。

类型安全的协议

现在，我们知道如何在两个线程之间传递消息了。那么，若想指明一个命令始终收到特定事件的响应应该怎么做呢？

下面构建一个简单的调用 - 响应协议，在线程之间传递函数的求值结果。在线程之间传递函数不太容易，不过我们可以选择在进程中定义函数，把参数发给该函数，再把结果发送回来。假如我们想构建一个矩阵计算引擎，让它支持三种运算：求矩阵的行列式、计算两个矩阵的点积和求逆矩阵。

计算方法我们都知道，下面先为这三种运算声明类型：

```
type Matrix = number[][]

type MatrixProtocol = {
  determinant: {
    in: [Matrix]
    out: number
  }
  'dot-product': {
    in: [Matrix, Matrix]
    out: Matrix
  }
  invert: {
    in: [Matrix]
    out: Matrix
  }
}
```

我们将在主线程中定义矩阵，运算则交给进程。总体思路与之前一样，我们对不安全的操作（进程发送和接收不带类型的消息）进行包装，把带类型的 API 开放给使用方。理想情况下，我们首先定义一个简单的请求响应协议 **Protocol**，列出进程可执行的操作，并为预期的输入和输出声明类型。^{注6} 然后，

注 6：这样实现针对的是理想情况，每执行一个命令都派生一个进程。实际使用中，或许会考虑维护一个进程池，让一些进程保持活跃，并回收空闲的进程。

定义一个泛型函数 `createProtocol`，其参数为一个 `Protocol` 和一个职程文件的路径；返回结果为一个函数，其参数为传入的协议中的 `command`；该函数最终也返回一个函数，我们调用这个函数求值 `command` 处理参数的结果。代码如下：

```
type Protocol = { ❶
  [command: string]: {
    in: unknown[]
    out: unknown
  }
}

function createProtocol<P extends Protocol>(script: string) { ❷
  return <K extends keyof P>(command: K) => ❸
    (...args: P[K]['in']) => ❹
      new Promise<P[K]['out']>((resolve, reject) => { ❺
        let worker = new Worker(script)
        worker.onerror = reject
        worker.onmessage = event => resolve(event.data.data)
        worker.postMessage({command, args})
      })
}
}
```

- ❶ 首先定义一个多用途的 `Protocol` 类型，而不限于只能处理 `MatrixProtocol`。
- ❷ 调用 `createProtocol` 时，传入职程文件的路径 `script`，以及特定的 `Protocol`。
- ❸ `createProtocol` 返回一个匿名函数，其参数为一个 `command`，值为❷中绑定的 `Protocol` 的键。
- ❹ 然后，使用❸中通过 `in` 传入的命令类型调用该函数。
- ❺ 得到的结果是一个 `Promise`，其类型为具体的协议定义的 `out` 类型。注意，我们要显式为 `Promise` 绑定一个类型参数，否则默认为 {}。

下面把 `MatrixProtocol` 类型和 Web 职程脚本的路径传给 `createProtocol`（不具体说明如何计算行列式，笔者假定你已经在 `MatrixWorkerScript.ts` 中实现了），得到一个函数，可在该协议中运行特定的命令：

```
let runWithMatrixProtocol = createProtocol<MatrixProtocol>('MatrixWorkerScript.js')
)
let parallelDeterminant = runWithMatrixProtocol('determinant')

parallelDeterminant([[1, 2], [3, 4]])
.then(determinant =>
  console.log(determinant) // -2
)
```

不错吧？我们把完全不安全的操作（在线程之间传递无类型的消息）抽象成了对类型绝对安全的请求 - 响应协议。协议支持的所有命令集中在一个地方 (`MatrixProtocol`)，而且核心逻辑 (`createProtocol`) 与协议的具体实现 (`runWithMatrixProtocol`) 是区分开的。

在两个进程之间通信，无论是在同一台设备中，还是在网络中的不同设备上，类型安全的协议都可以保障通信的安全。本节清楚地说明了协议可以解决哪些问题，不过在实际使用中，建议直接使用现有的工具，例如 Swagger、gRPC、Thrift 或 GraphQL，9.2 节将简要介绍这几个工具。

8.6.2 在 NodeJS 中：使用子进程



如果你想跟着本节的示例一起操作，请使用 NPM 安装 NodeJS 的类型声明：

```
npm install @types/node --save-dev
```

类型声明的进一步使用说明，参见 11.4.2 节。

在 NodeJS 中并行执行类型安全的操作，方式与在浏览器中使用 Web 职程一样（见“类型安全的协议”一节）。虽然消息传递层不安全，但是我们可以在此基础上自己动手构建类型安全的 API。NodeJS 的子进程 API 使用方法如下：

```
// MainThread.ts
import {fork} from 'child_process'
```

```
let child = fork('./ChildThread.js') ①

child.on('message', data => ②
  console.info('Child process sent a message', data)
)

child.send({type: 'syn', data: [3]}) ③
```

- ① 使用 NodeJS 的 `fork` API 派生子进程。
- ② 使用 `on` API 监听子进程发来的消息。NodeJS 的子进程发给父进程的消息有好几种，这里只监听 '`message`' 消息。
- ③ 使用 `send` API 把消息发给子进程。

在子线程中，使用 `process.on` API 监听主线程发来的消息，使用 `process.send` 把消息发给主线程：

```
// ChildThread.ts
process.on('message', data => ①
  console.info('Parent process sent a message', data)
)

process.send({type: 'ack', data: [3]}) ②
```

- ① 在全局定义的 `process` 上调用 `on` API 监听父线程发来的消息。
- ② 在 `process` 上调用 `send` API 把消息发给父进程。

可以看出，消息传递机制与 Web 职程差不多。笔者不再说明在 NodeJS 中如何实现类型安全的协议，抽象进程间的通信。实现细节留作练习。

8.7 小结

本章首先介绍了 JavaScript 事件循环，接着讨论了在 JavaScript 中编写异步代码的方式，以及在 TypeScript 中如何安全表达，包括回调、`promise`、`async/await` 和事件发射器。然后，又介绍了多线程，探讨了在线程之间传递消息的方式（涵盖浏览器和服务器），还为线程之间的通信构建了完整的协议。

与第 7 章一样，具体使用哪一项技术由你自己决定：

- 对简单的异步任务来说，使用回调更容易理解。
- 如果任务较复杂，需要排序和并行，那就使用 `promise` 和 `async/await`。
- 如果觉得 `promise` 不够用了（例如多次触发一个事件），可以转用事件发射器或响应式流处理库（如 RxJS）。
- 牵涉到多个线程时，使用事件发射器、类型安全的协议或类型安全的 API（见 9.2 节）。

8.8 练习题

1. 实现通用的 `promisify` 函数，其参数为一个函数，该函数接受一个参数和一个回调，对这个函数做包装，返回一个 `promise` 对象。实现之后，可以像下面这样使用 `promisify` 函数（在此之前要通过 `npm install @types/node --save-dev` 安装 NodeJS 的类型声明）：

```
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. 类型安全的协议只给出了类型安全的矩阵计算协议的一部分。根据在主线程中运行的这一半，实现在 Web 职程中运行的另一半。
3. 使用映射类型（见 8.6.1 节）为 NodeJS 的 `child_process` 实现类型安全的消息传递协议。

前后端框架

应用的每个组成部分都可以由我们自己动手从头编写，比如服务器端的网络和数据库层、前端的用户界面和状态管理方案，但是我们无需自费力气，毕竟有些细节我们不一定能处理到位。幸好，前后端涉及的多数棘手问题已经被其他工程师解决了。利用现有的工具、库和框架构建前后端，便于快速迭代，而且能为我们的应用奠定坚实的基础。

本章介绍一些最为流行的工具和框架，涵盖客户端和服务器端的常见问题。在这个过程中，我们将讨论各框架的可能用途，以及如何安全地把框架集成到 TypeScript 应用中。

9.1 前端框架

TypeScript 特别适合用于开发前端应用。TypeScript 对 JSX 有很好的支持，而且能安全地建模不可变性，从而提升应用的结构和安全性，写出的代码正确性高、便于维护，这在快速发展的前端开发领域是难能可贵的闪光点。

当然，所有内置的 DOM API 在类型上都是安全的。如果想在 TypeScript 中使用，只需在项目的 `tsconfig.json` 中引入相应的类型声明：

```
{  
  "compilerOptions": {  
    "lib": ["dom", "es2015"]  
  }  
}
```

```
    }  
}
```

这样设置的目的是让 TypeScript 在做类型检查时引入 *lib.dom.d.ts*，即内置的浏览器和 DOM 类型声明。



tsconfig.json 中的 `lib` 选项仅仅是让 TypeScript 在处理项目的代码时引入一组指定的类型声明，不产生额外的代码，也不在运行时生成任何 JavaScript 代码。比如说，代码不会像变魔术一样自动可在 NodeJS 环境中使用（代码能编译，但是运行时将出错），我们自己要确保使用与 JavaScript 环境相匹配的类型声明。详情参见 12.1 节。

引入 DOM 类型声明之后，我们便可以放心使用 DOM 和浏览器 API，例如：

```
// 从全局对象 window 上读取属性  
let model = {  
  url: window.location.href  
}  
  
// 创建一个 <input /> 元素  
let input = document.createElement('input')  
  
// 为该元素设定几个 CSS 类  
input.classList.add('Input', 'URLInput')  
  
// 在用户输入时更新模型  
input.addEventListener('change', () =>  
  model.url = input.value.toUpperCase()  
)  
  
// 把这个 <input /> 元素插入 DOM 中  
document.body.appendChild(input)
```

当然，这些代码都将经过类型检查，而且在编辑器中可以自动补全。举个例子，请看下面的代码：

```
document.querySelector('.Element').value // Error TS2339: Property 'value' does  
                                         // not exist on type 'Element'.
```

TypeScript 会抛出错误，因为 `querySelector` 的返回类型可以为空。

对简单的前端应用来说，这些低层的 DOM API 可能就够用了，在类型的指引下，我们可以放心针对浏览器编程。然而，实际上多数前端应用会使用某个框架，对 DOM 渲染和重新渲染、数据绑定及事件处理等做一层抽象。下面几节简要说明如何在 TypeScript 中正确使用最为流行的几个浏览器框架。

9.1.1 React

React 是现今最流行的前端框架之一，在考量类型安全上，也是个不错的选择。

说 React 安全，是因为 React 组件（React 应用的基本组成）本身是使用 TypeScript 编写的，而且这些组件也要通过 TypeScript 使用。在众多前端框架中，很难发现还有哪一个框架做到了这一点。这意味着，组件的定义和使用都要经过类型检查。通过类型，我们可以表述“该组件接受一个用户 ID 和一个颜色”，或者“该组件只能提供一些列表项目”。这些约束在 TypeScript 的监管之下，可以确保组件做了该做的事。

组件的定义和使用（在前端应用的视图层使用）都是安全的，这是 React 脱颖而出的关键。视图层往往潜藏各种错误，比如打错字、遗漏属性、错拼参数和错误嵌套元素，令程序员抓耳挠腮，疯狂刷新浏览器。不过，一旦开始使用 TypeScript 和 React 编写视图，你自己和你所在的团队在前端开发上必将事半功倍。

JSX 简介

在 React 中，视图使用特殊的 DSL 定义，叫做 JavaScript XML（JSX）。JSX 直接嵌套在 JavaScript 代码中，有点儿像 JavaScript 中的 HTML。JavaScript 代码经过 JSX 编译器处理之后，独特的 JSX 句法将变成常规的 JavaScript 函数调用。

下面举个例子说明这个过程。假如你在为一位朋友的餐馆开发一个菜单应用，你使用下述 JSX 列出早午餐菜单上的几道菜：

```
<ul class='list'>
<li>Homemade granola with yogurt</li>
```

```
<li>Fantastic french toast with fruit</li>
<li>Tortilla Espanola with salad</li>
</ul>
```

使用 JSX 编译器处理之后，比如 Babel 的 `transform-react-jsx` 插件 (<http://bit.ly/2uENY4M>)，得到的输出如下：

```
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 Espanola with salad'
  )
);
```



TSC 标志：esModuleInterop

JSX 经编译后变成 `React.createElement` 调用，因此必须在使用 JSX 的每个文件中导入 `React` 库，这样在作用域中才有名为 `React` 的变量：

```
import React from 'react'

<ul /> // Error TS2304: Cannot find name 'React'.
```

不过请注意，笔者在 `tsconfig.json` 中设置了 `{"esModuleInterop": true}`，无需使用通配符 (*) 就能导入 `React`。如果你想跟着本节的内容操作，可以在 `tsconfig.json` 中启用 `esModuleInterop`，也可以使用通配符导入：

```
import * as React from 'react'
```

JSX 的优势在于，我们可以像常规的 HTML 那样编写视图，然后编译成对 JavaScript 引擎友好的格式。作为工程师，我们使用熟悉的高级声明式 DSL 即可，无需分心处理实现细节。

使用 React 不一定要使用 JSX（完全可以直接编写编译后得到的代码），JSX 也不一定必须在 React 中使用（编译 JSX 标签得到的函数调用，即前例中的 `React.createElement` 是可以配置的），不过 React 和 JSX 组合在一起使用效果更好，这样编写视图充满乐趣，而且十分安全。

TSX = JSX + TypeScript

JSX 文件的扩展名为 `.jsx`。如果 TypeScript 文件中包含 JSX，扩展名则为 `.tsx`。TSX 与 JSX 之间的关系就像 TypeScript 与 JavaScript 一样，TSX 具有编译时安全性，而且能协助我们提高效率，写出的代码错误更少。如果想在项目中增加 TSX 支持，在 `tsconfig.json` 中加入下面这一行：

```
{
  "compilerOptions": {
    "jsx": "react"
  }
}
```

目前，`jsx` 指令有三种模式：

react

- 使用 JSX 编译指示（pragma），把 JSX 编译为 `.js` 文件。

react-native

- 保留 JSX，不编译，但是会生成一个 `.js` 文件。

preserve

- 对 JSX 做类型检查，但是不编译，生成一个 `.jsx` 文件。

TypeScript 在底层开放了几个钩子，允许我们替换注解 TSX 类型的方式。这些类型在特殊的 `global.JSX` 命名空间中，TypeScript 使用该命名空间检查程

序中的 TSX 类型。如果你只是使用 React，无需深入到这一层级，但是，假如你想自己开发使用 TSX 的 TypeScript 库（而且不使用 React），或者好奇 React 的类型声明是如何处理的，请翻到附录 G。

在 React 中使用 TSX

在 React 中，我们可以声明两种组件：函数组件和类组件。这两种组件都接受一些属性，渲染一些 TSX。在使用者看来，两者之间没有区别。

声明及渲染函数组件的方式如下所示：

```
import React from 'react' ①

type Props = { ②
  isDisabled?: boolean
  size: 'Big' | 'Small'
  text: string
  onClick(event: React.MouseEvent<HTMLButtonElement>): void ③
}

export function FancyButton(props: Props) { ④
  const [toggled, setToggled] = React.useState(false) ⑤
  return <button
    className={'Size-' + props.size}
    disabled={props.isDisabled || false}
    onClick={event => {
      setToggled(!toggled)
      props.onClick(event)
    }}
    >{props.text}</button>
}

let button = <FancyButton ⑥
  size='Big'
  text='Sign Up Now'
  onClick={() => console.log('Clicked!')}
/>
```

- ① 若想在 React 中使用 TSX，要把 React 变量导入当前作用域。TSX 编译成 React.createElement 函数调用，因此我们要导入 React，这样在运行时才有该变量的定义。

- ② 首先声明传给 `FancyButton` 组件的一组属性。`Props` 始终为对象类型，而且习惯命名为 `Props`。在 `FancyButton` 组件中，`isDisabled` 是可选的，其他属性都是必需的。
- ③ React 有自己的一套 DOM 事件包装类型。处理 React 事件时，必须使用 React 的事件类型，不能使用常规的 DOM 事件类型。
- ④ 函数组件其实就是普通的函数，该函数最多接受一个参数（`props` 对象），返回一个 React 可渲染的类型。React 的要求并不严格，可渲染的类型有很多，包括：`TSX`、字符串、数字、布尔值、`null` 和 `undefined`。
- ⑤ 使用 React 的 `useState` 钩子为函数组件声明局部状态。`useState` 是 React 提供的众多钩子中的一个，你也可以结合不同的钩子自定义钩子。注意，我们传给 `useState` 的初始值是 `false`，TypeScript 据此可以推导出这个状态是 `boolean` 类型。倘若使用 TypeScript 无法推导的类型，例如数组，那就需要显式绑定类型（例如 `useState<number[]>([]);`）。
- ⑥ 使用 `TSX` 句法创建一个 `FancyButton` 实例。`<FancyButton />` 句法的作用基本上与调用 `FancyButton` 函数一样，不过这样做可以把 `FancyButton` 的生命周期交给 React 管理。

就这么简单。在 TypeScript 的监管之下，可以保证：

- JSX 的格式良好。标签要关闭、要正确嵌套，而且标签名称不能有拼写错误。
- 实例化 `<FancyButton />` 时要把所有必需的属性（`size`、`text` 和 `onClick`；外加可选的属性）传给 `FancyButton`，而且各属性都要具有正确的类型。
- 没有把多余的属性传给 `FancyButton`，而只传入必需的属性。

类组件差不多：

```
import React from 'react' ①  
import {FancyButton} from './FancyButton'
```

```

type Props = { ❷
  firstName: string
  userId: string
}

type State = { ❸
  isLoading: boolean
}

class SignupForm extends React.Component<Props, State> { ❹
  state = { ❺
    isLoading: false
  }
  render() { ❻
    return <> ❼
      <h2>Sign up for a 7-day supply of our tasty
        toothpaste now, {this.props.firstName}.</h2>
      <FancyButton
        isDisabled={this.state.isLoading}
        size='Big'
        text='Sign Up Now'
        onClick={this.signUp}
      />
    </>
  }
  private signUp = async () => { ❽
    this.setState({isLoading: true})
    try {
      await fetch('/api/signup?userId=' + this.props.userId)
    } finally {
      this.setState({isLoading: false})
    }
  }
}

let form = <SignupForm firstName='Albert' userId='13ab9g3' /> ❾

```

- ❶ 与之前一样，把 React 导入当前作用域。
- ❷ 与之前一样，声明 Props 类型，定义创建 `<SignupForm />` 实例时要传入什么数据。
- ❸ 声明 State 类型，用于建模该组件的局部状态。
- ❹ 声明类组件时要扩展 `React.Component` 基类。

- ⑤ 使用属性初始化语句声明局部状态的默认值。
- ⑥ 与函数组件类似，类组件的 `render` 方法返回一些可被 React 渲染的内容，可以是 TSX、字符串、数字、布尔值、`null` 或 `undefined`。
- ⑦ TSX 支持使用特殊的 `<>...</>` 句法返回片段。片段是放在一个没有名称的 TSX 元素中的一段 TSX。如果只想返回一个 TSX 元素，而又不想多渲染额外的 DOM 元素，就可以使用片段。假如有一个 React 组件的 `render` 方法只需要返回一个 TSX 元素，我们可以把该元素放在 `<div>` 或其他元素中，但是这样做在渲染时有额外的消耗。
- ⑧ 使用箭头句法定义 `signUp` 函数，确保在该函数中 `this` 不会重新绑定。
- ⑨ 最后，实例化 `SignupForm`。与实例化函数组建一样，我们也可以直接使用 `new`，比如 `new SignupForm({firstName: 'Albert', userId: '13ab9g3'})`，但是这样做就无法让 React 代为管理 `SignupForm` 实例的生命周期了。

注意，这个示例混用了基于值的组件（`FancyButton`、`SignupForm`）和原生组件（`section`、`h2`）。我们利用 TypeScript 验证了：

- 所有必需的状态字段都在 `state` 初始化语句或构造方法中定义了。
- 通过 `props` 和 `state` 访问的值都存在，而且是预期的类型。
- 没有通过 `this.state` 直接设定状态，因为在 React 中，状态要通过 `setState` API 更新。
- 调用 `render` 确实返回了一些 JSX。

借助 TypeScript 可以确保 React 代码更安全，进而促使我们个人的生活更加美好、舒畅。



我们没有使用 React 的 `PropTypes` 特性。这个特性能在运行时声明并检查属性的类型。由于 TypeScript 已经在编译时检查了，没必要再检查一次。

9.1.2 Angular 6/7

Shyam Seshadri 撰写

前端框架 Angular 比 React 的功能更完善，不仅支持渲染视图，还能发送和管理网络请求、分发路由及注入依赖。Angular 完全支持 TypeScript（其实该框架本身就是使用 TypeScript 编写的）。

Angular 的核心是内置在 Angular 命令行工具 Angular CLI 中的预编译器（Ahead-of-Time，AoT）。AoT 编译器从 TypeScript 注解中获取类型信息，利用这些信息把你编写的代码编译成常规的 JavaScript 代码。Angular 不把相关工作直接交给 TypeScript，而是先对代码进行一系列优化和转换，然后再委托 TypeScript 把代码编译成 JavaScript。

下面说明 Angular 是如何利用 TypeScript 和 AoT 编译器提升前端应用的安全性的。

基本结构

新建 Angular 项目之前，首先要使用 NPM 全局安装 Angular CLI：

```
npm install @angular/cli --global
```

然后，使用 Angular CLI 初始化一个 Angular 应用：

```
ng new my-angular-app
```

按照提示操作，Angular CLI 将为你创建一个 Angular 应用骨架。

本书不深入说明 Angular 应用的结构，以及如何配置和运行 Angular 应用。详情参见 Angular 官方文档 (<https://angular.io/docs>)。

组件

下面我们来构建一个 Angular 组件。Angular 组件与 React 组件类似，可以指明组件的 DOM 结构、样式和控制器。Angular 组件的样板代码可以使用

Angular CLI 生成，以此为基础再补充细节。一个 Angular 组件由几个不同的文件构成：

- 一个模板，指明组件渲染的 DOM。
- 一组 CSS 样式。
- 一个组件类，这是一个 TypeScript 类，描述组件的业务逻辑。

先声明组件类：

```
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'
  }
}
```

这个 TypeScript 类的多数内容不难理解，只有几处涉及 Angular 对 TypeScript 的利用，你可能不明白，说明如下：

- Angular 的生命周期钩子是 TypeScript 接口，只定义了应该实现哪些方法（`ngOnChanges`、`ngOnInit` 等）。TypeScript 负责确保我们实现了符合生命周期钩子要求的方法。这里我们实现的是 `OnInit` 接口，该接口要求实现 `ngOnInit` 方法。
- Angular 大量使用 TypeScript 装饰器（见 5.9 节），为 Angular 组件、服务和模块声明相关的元数据。这个示例通过 `selector` 声明他人可以通过怎么的方式使用该组件，使用 `templateUrls` 和 `styleUrl` 链接该组件的 HTML 模板和 CSS 样式表。



TSC 标志: fullTemplateTypeCheck

如果想对 Angular 模板做类型检查, 请在 `tsconfig.json` 中启用 `fullTemplateTypeCheck`:

```
{  
  "angularCompilerOptions": {  
    "fullTemplateTypeCheck": true  
  }  
}
```

注意, `angularCompilerOptions` 不是 TSC 特有的选项, 而是 Angular 的 AoT 编译器使用的编译器标志。

服务

Angular 内置依赖注入器 (dependency injector, DI), 该框架通过 DI 管理服务的实例化, 并负责把服务作为参数传给依赖某一服务的组件和服务。这样做便于实例化及测试服务和组件。

下面更新 `SimpleMessageComponent`, 注入 `MessageService` 依赖。该服务的作用是从服务器中获取消息。

```
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 编译器分析组件的 `constructor` 接受哪些参数，提取出参数的类型（例如 `MessageService`），在相关依赖注入器的依赖图中搜索该类型的依赖。然后，使用 `new` 关键字实例化那个依赖（如果尚未实例化的话），传给 `SimpleMessageComponent` 实例的构造方法。DI 相关的操作十分复杂，不过当你的应用体量变大以后，实际使用的依赖可能取决于应用的配置（例如使用 `ProductionAPIService` 还是 `DevelopmentAPIService`）或者是否在做测试（`MockAPIService`），这时候你就会发现，使用 DI 更方便。

下面简单看一下如何定义服务：

```
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')
  }
}
```

在 Angular 中创建服务也要使用 TypeScript 装饰器把服务注册为 `Injectable`，定义在应用的根层还是在子模块中提供服务。这里，我们注册了 `MessageService` 服务，那么该服务就可以在应用的任何地方注入了。在组件或服务的构造方法中，我们只需指明需要提供一个 `MessageService`，这样 Angular 便会自动传入该服务。

讲完如何安全地使用两个流行的前端框架之后，下面换个话题，讨论如何为介于前端和后端之间的接口声明类型。

9.2 类型安全的 API

Nick Nance 撰写

不管你打算使用哪个前端和后端框架，你肯定需要一种安全在设备之间（客户端到服务器、服务器到客户端、服务器到服务器和客户端到客户端）通信的方式。

这一领域有很多工具和标准。不过，在讨论这些工具和标准的用法和原理之前，请思考一下如何自己动手实现，以及自己实现有什么优缺点（毕竟我们是工程师）。

我们要解决的问题是这样的：虽然客户端和服务器自身可能是 100% 安全的（安全第一），但是有时二者之间可能会通过无类型信息的协议通信，比如 HTTP、TCP 或其他基于套接字的协议。那么，如何让这样的通信对类型安全呢？

最先想到的可能是使用对类型安全的协议，例如“类型安全的协议”一节开发的那个。如下所示：

```
type Request =
  | {entity: 'user', data: User}
  | {entity: 'location', data: Location}

// client.ts
async function get<R extends Request>(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
}
```

我们可以分别定义 `post` 和 `put` 函数，通过 REST API 写入数据，并为服务器支持的各个实体添加类型。在后端，我们可以实现一些处理函数，分别对应各种实体，从数据库中读取数据，把请求的实体发回客户端。

可是，如果服务器端的代码不是用 TypeScript 编写的呢？如果不能在客户端和服务器之间共享 `Request` 类型呢（最终导致两端脱节）？如果不使用 REST 架构呢（有可能使用 GraphQL）？如果还要支持其他客户端呢，比如 Swift 客户端、iOS 客户端或 Android 中的 Java 客户端？

这时应该使用由代码生成的带类型信息的 API。具体做法有很多种，而且一些语言（包括 TypeScript）有相应的库，例如：

- 针对 REST 式 API 的 Swagger (<https://github.com/swagger-api/swagger-codegen>)。
- 针对 GraphQL 的 Apollo (<https://www.npmjs.com/package/apollo>) 和 Relay (<https://facebook.github.io/relay/>)。
- 针对 RPC 的 gRPC (<https://grpc.io/>) 和 Apache Thrift (<https://thrift.apache.org/>)。

这些工具都要求服务器和客户端使用相同的数据源，Swagger 使用数据模型、Apollo 使用 GraphQL 模式（schema）、gRPC 使用 Protocol Buffers。从数据源获取的信息将被编译成针对特定语言（我们关注的是 TypeScript）的绑定。

通过代码生成可以避免客户端与服务器（或多个客户端）之间脱节。各平台共用同一个模式（schema），不会出现在 iOS 应用中新增字段后忘记合并拉取请求把新字段也添加到服务器中的情况。

篇幅有限，本书不深入探讨每个框架。请根据需要为自己的项目选择一个框架，通过框架的文档进一步学习。

9.3 后端框架

构建与数据库交互的应用，一开始可能会直接使用原始 SQL 或 API 调用，而这两种方式本身都不含类型信息：

```
// 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 is any
  )
```

我们可以自己动手添加类型，提升 API 的安全性，尽量减少出现 any 的情况：

```
db.collection('users')
  .find({id: 739311})
  .toArray((err, user: User) =>
    // user is any
  )
```

然而，原始 SQL API 还是相当低层，而且容易使用错误的类型，或者忘记添加类型，导致得到的还是 any。

对象关系器（Object-relational Mapper, ORM）应运而生。ORM 根据数据库模式生成代码，提供的是高层 API，可以执行查询、更新、删除等操作。在静态类型语言中，这些 API 对类型是安全的，无须担心有没有使用正确的类型，也不用自己动手绑定泛型参数。

通过 TypeScript 访问数据库，建议使用 ORM。目前，Umed Khudoiberdiev 开发的 TypeORM (<https://www.npmjs.com/package/typeorm>) 是在 TypeScript 可用的 ORM 中最完善的一个，支持 MySQL、PostgreSQL、Microsoft SQL

Server、Oracle，甚至 MongoDB。使用 TypeORM，可以通过类似下面的代码查询用户的名字：

```
let user = await UserRepository
  .findOne({id: 739311}) // User | undefined
```

使用这样的高层 API，默认情况下就兼顾安全性（可以避免 SQL 注入等攻击）和类型安全（不自己动手注解就知道 findOne 的返回类型）。与数据库交互一定要使用 ORM，ORM 用起来方便，而且你也不会在凌晨 4 点被人叫起来，紧急处理 saleAmount 字段为 null 的问题。这种情况不是不可能发生，比如说你在前一天晚上把这个字段改成了 orderAmount，同事知道你出差在外，晚些时候才会发拉取请求，于是决定先运行数据库迁移；尽管这次迁移成功了，但是午夜时分拉取请求却失败了；纽约的销售团队醒来后发现所有客户订单上的金额都是 null。

9.4 小结

本章讲了很多内容：直接操纵 DOM，使用 React 和 Angular，通过 Swagger、gRPC 和 GraphQL 等工具提升 API 的类型安全性，以及使用 TypeORM 安全地与数据库交互。

JavaScript 框架日新月异，你读到这一章时，这里讨论的 API 和框架可能已经过时。然而，你已经知道类型安全的框架能解决怎样的问题，请根据这一知识发掘他人的劳动成果中值得自己借鉴的地方，提升代码的安全性，让代码更抽象、更模块化。本章的目的不是告诉你现在最好的框架是哪一个，而是让你知道使用框架可以解决什么样的问题。

借助类型安全的 UI 代码、带类型信息的 API 层和类型安全的后端，应用中的很多 bug 将无所遁形，夜晚我们也将睡得更加踏实。

命名空间和模块

在一个程序中，我们可以在不同的层级上进行封装。在最底层，函数封装行为，对象和列表等数据结构封装数据。函数和数据还可以放在类中，还可以把数据放在单独的数据库或数据存储器中，把函数和数据放入独立的命名空间里。通常，一个类或一系列实用函数放在一个文件中。再向上一层，我们可能会把多个类或多组实用函数组织在一起，构建成包（package），发布到 NPM 中。

模块（module）是一个重要的概念，我们要知道编译器（TSC）是如何解析模块的，要知道构建系统（Webpack、Gulp 等）是如何解析模块的，还要知道模块在运行时是如何加载到应用中的（使用 `<script />` 标签、SystemJS 等）。对 JavaScript 来说，这些操作往往都由单独的程序执行，有点儿混乱。CommonJS 和 ES2015 模块标准简化了这三个程序之间的互操作，Webpack 等强大的打包程序进一步抽象了这三个解析过程背后涉及的操作。

本章着重讨论这三种工具中的第一种，即 TypeScript 解析和编译模块所用的工具，第 12 章再探讨构建系统和运行时加载程序。本章涵盖以下内容：

- 实现命名空间和模块的不同方式。
- 导入和导出代码的不同方式。
- 在代码基增长的过程中弹性伸缩这些方法。

- 模块模式与脚本模式的比较。

- 声明合并的概念及其作用。

不过，首先要了解一些基础知识。

10.1 JavaScript 模块简史

由于 TypeScript 最终编译成 JavaScript，并与 JavaScript 互操作，因此 TypeScript 要支持 JavaScript 程序使用的各种模块标准。

起初（1995 年），JavaScript 不支持任何模块系统。由于没有模块，一切都在全局命名空间中声明，导致应用非常难以构建和弹性伸缩。很快，变量名称将用尽，致使冲突出现。而且，我们无法显式指明各个模块对外开放哪些 API，导致难以分清模块的哪些部分是供外部使用的，哪些部分是不便开放的实现细节。

为了解决这样的问题，人们通过对对象或立即调用的函数表达式（Immediately Invoked Function Expression，IIFE）模拟模块，把对象或 IIFE 附加到全局对象 `window` 上，供当前应用（或同一服务器中的其他应用）中的其他模块使用。具体方式如下所示：

```
window.emailListModule = {
  renderList() {}
  // ...
}

window.emailComposerModule = {
  renderComposer() {}
  // ...
}

window.appModule = {
  renderApp() {
    window.emailListModule.renderList()
    window.emailComposerModule.renderComposer()
  }
}
```

可是加载和运行 JavaScript 会阻塞浏览器 UI，随着应用的增长，代码行数越来越多，用户的浏览器也变得越来越慢。鉴于此，一些聪明的程序员发明了一种新方法：在页面加载完成后动态加载 JavaScript，而不同步加载。JavaScript 首次发布近 10 年后出现的 Dojo (Alex Russell, 2004 年)、YUI (Thomas Sha, 2005 年) 和 LABjs (Kyle Simpson, 2009 年) 等库内置了模块加载程序，在页面加载完毕后惰性（通常是异步）加载 JavaScript 代码。为了实现异步惰性加载模块，要做到以下三点：

1. 模块要适当地封装。否则，不断流入的依赖将导致页面崩溃。
2. 模块之间的依赖要显式声明。否则，不知道需要加载什么模块，以及以什么顺序加载。
3. 每个模块在应用中都要有一个唯一标识符。否则，无法指定需要加载哪个模块。

使用 LABjs 加载模块的方法如下：

```
$LAB
  .script('/emailBaseModule.js').wait()
  .script('/emailListModule.js')
  .script('/emailComposerModule.js')
```

在差不多同样的时间点，NodeJS (Ryan Dahl, 2009 年) 也在开发中，开发人员深谙 JavaScript 在发展过程中面临的问题，同时参考其他语言，决定在该平台中内置一个模块系统。与其他优秀的模块系统一样，NodeJS 的模块系统也要满足 LABjs 和 YUI 的加载程序那样的三个重要条件。NodeJS 采用的是 CommonJS 模块标准，使用方法如下所示：

```
// emailBaseModule.js
var emailList = require('emailListModule')
var emailComposer = require('emailComposerModule')

module.exports.renderBase = function() {
  // ...
}
```

与此同时，由 Dojo 和 RequireJS 主推的 AMD 模块标准 (James Burke, 2008 年) 横空出世。这个标准支持的功能不变，而且内置了一个构建系统，用于打包 JavaScript 代码。

```
define('emailBaseModule',
  ['require', 'exports', 'emailListModule', 'emailComposerModule'],
  function(require, exports, emailListModule, emailComposerModule) {
    exports.renderBase = function() {
      // ...
    }
  }
)
```

几年之后，Browserify (James Halliday, 2011 年) 问世，让前端工程师也可以在前端使用 CommonJS。CommonJS 变成了模块化打包及导入和导出句法的事实标准。

可是，CommonJS 处理事情的方式有几个问题，包括：`require` 必须同步调用，CommonJS 模块解析算法不适合在 Web 中使用。另外，使用 CommonJS 的代码在某些情况下不会做静态分析（这一定能引起 TypeScript 程序员的注意），这是因为 `module.exports` 可能出现在任何位置（甚至可以在永远不会执行的代码分支中），`require` 调用也可以出现在任何位置，而且可能包含任意字符串和表达式，这就导致无法静态链接 JavaScript 程序，不能确认被引用的文件真实存在，也无法确保被引用的文件真的导出了声称导出的代码。

在这个背景之下，ECMAScript 语言第 6 版，即 ES2015 为导入和导出引入了一个新标准，句法更简洁，而且支持静态分析。这个标准的用法如下所示：

```
// emailBaseModule.js
import emailList from 'emailListModule'
import emailComposer from 'emailComposerModule'

export function renderBase() {
  // ...
}
```

如今，我们在 JavaScript 和 TypeScript 中使用的就是这个标准。然而，写作本书之时，不是每个 JavaScript 引擎都原生支持这个标准，所以我们要把代码编译成被支持的格式（在 NodeJS 环境中使用，编译成 CommonJS 格式；在浏览器环境中使用，编译成全局或单个模块可加载的格式）。

在使用其他模块及从模块中导出代码上，TypeScript 提供了好几种方式：可以使用全局声明，可以使用 ES2015 中标准的 `import` 和 `export`，也可以使用 CommonJS 模块那种向后兼容的 `import`。除此之外，TSC 的构建系统还支持针对不同的目标环境编译模块，包括：全局、ES2015、CommonJS、AMD、SystemJS 和 UMD（CommonJS、AMD 和全局的混合体，就看使用方的环境中有哪个标准）。

10.2 import、export

除非迫不得已，在 TypeScript 代码中应该使用 ES2015 的 `import` 和 `export` 句法，不要使用 CommonJS、全局或命名空间下的模块。在 TypeScript 中，ES2015 的导入和导出句法就像常规的 JavaScript 代码一样，如下所示：

```
// 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，没有花括号
meow(11)
```

此外，也支持使用通配符 (*) 从一个模块中导入一切：

```
// e.ts
import * as a from './a'
a.foo()
a.bar()
```

以及从一个模块中重新导出部分（或全部）：

```
// f.ts
export * from './a'
export {result} from './b'
export meow from './c'
```

由于我们编写的是 TypeScript，而不是 JavaScript 代码，理所当然，除了值以外，还可以导出类型和接口。又因为类型和值位于不同的命名空间中，所以完全可以导出两个同名的内容，一个在值层面，另一个在类型层面。与其他代码一样，TypeScript 将推导出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 指代值 X
let b: X = {y: 'z'}    // X 指代类型 X
```

模块路径是文件系统中的文件名。这让模块与模块在文件系统中的目录结构耦合起来了，不过对模块加载程序来说，却是个重要的特性：加载程序知道目录结构才能根据模块名称找到文件。

10.2.1 动态导入

应用体量增大之后，初次渲染的时间将不断增加。这对前端应用来说是不容小觑的问题，毕竟网络是一个瓶颈。后端应用同样不能忽视这个问题，如果顶层要导入很多代码，从文件系统中导入、解析、编译和求值，这一系列操作耗费的时间很多，而且会阻塞其他代码的执行。

在前端，解决这个问题的方法之一（除了减少代码量外）是代码拆分（code splitting）：把代码拆分成小块，生成一系列 JavaScript 文件，而不把所有代码都放在一个大型文件中。拆分之后，我们可以并行加载多块代码，降低网络请求的负担（见图 10-1）。

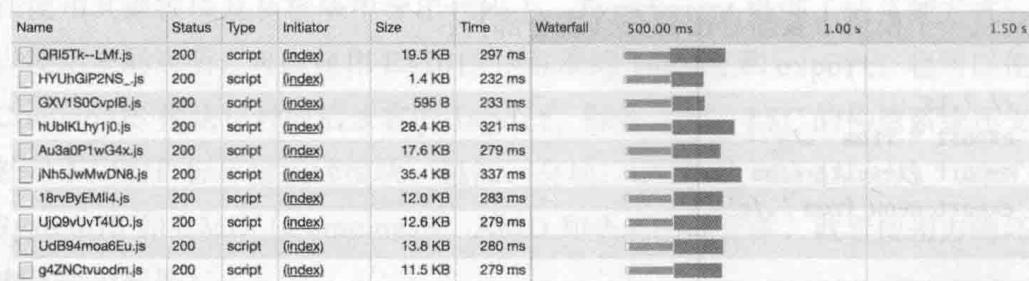


图 10-1：facebook.com 加载 JavaScript 时看到的网络请求瀑布流

进一步的优化措施是惰性加载分块，即在真正用到时才加载。对大型前端应用，比如 Facebook 和 Google 的一些应用，势必要采用这种优化方式。如若不然，首次加载页面时客户端可能要加载数千兆字节的 JavaScript 代码，这将耗时几分钟或几小时（更糟的是，用户收到移动消费账单后说不定再也不会使用这些服务了）。

惰性加载还有其他用处。以流行的 Moment.js 库 (<https://momentjs.com>) 为例，这个库的作用是处理日期，内含支持世界各地所用日期格式的包，按区域划分。每个包的大小约为 3 KB。如果每个用户都加载所有包，在性能上肯定是无法接受的，而且带宽也不允许。正确的做法是检测用户的区域设置，只加载针对目标区域的包。

LABjs 及其衍生库引入了在真正需要时才惰性加载的概念，这个概念正式的名称是“动态导入”（dynamic imports），用法如下：

```
let locale = await import('locale_us-en')
```

import 可以作为一个语句使用，作用是静态获取代码（在此之前都是这样用

的)；另外，也可以作为一个函数调用，此时返回结果是一个 `Promise` (这个例子是这样用的)。

尽管传给 `import` 的参数可以是任何求值结果为字符串的表达式，不过这样做丧失了类型安全性。出于安全考虑，动态导入应该使用下面两种做法中的其中一种：

1. 直接把字符串字面量传给 `import`，不要事先赋值给变量。
2. 把表达式传给 `import`，但要手动注解模块的签名。

如果选择第二种方式，常见的做法是静态导入模块，不过只做为类型使用，TypeScript 在编译时将把静态导入忽略（见 12.5.1 节）。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` 中导入 `locale`，不过仅用作类型，通过 `typeof locale` 获取。之所以要这么做，是因为 TypeScript 通过静态方式无法获知 `import(path)` 的类型，而这其中的原因在于 `path` 是需要通过计算才能得到结果的变量，而不是静态的字符串。在这段代码中，我们不把 `locale` 当作值使用，而是只通过它获得它的类型，TypeScript 在编译时将把这个静态导入语句忽略（即 TypeScript 不生成任何顶层导入代码）。这样写出的代码不仅具有十足的类型安全性，而且还能动态计算导入哪个包。



TSC 设置：module

TypeScript 仅在 `esnext` 模块模式下支持动态导入。若想使用动态导入，在 `tsconfig.json` 中的 `compilerOptions` 选项里设置 `{"module": "esnext"}`。详情参见 12.2 节和 12.3 节。

10.2.2 使用 CommonJS 和 AMD 模块

使用采用 CommonJS 或 AMD 标准的 JavaScript 模块时，可以直接从模块中导入名称，就像使用 ES2015 模块一样：

```
import {something} from './a/legacy/commonjs/module'
```

默认情况下，CommonJS 的默认导出不能与 ES2015 的默认导出互操作；若想使用默认导出，要借助通配符导入：

```
import * as fs from 'fs'  
fs.readFile('some/file.txt')
```

如果想更直接一些，要在 `tsconfig.json` 中的 `compilerOptions` 选项中设置 `{"esModuleInterop": true}`。这样就不需要使用通配符了：

```
import fs from 'fs'  
fs.readFile('some/file.txt')
```



本章开头说过，这段代码虽然能编译通过，但并不意味着可在运行时正常运行。不管使用什么模块标准，是 `import/export`、CommonJS、AMD、UMD，还是浏览器全局模块，模块打包程序和加载程序都要理解所用的格式，这样在编译时才能打包和分拆代码，并在运行时正确加载代码。详情参见第 12 章。

10.2.3 模块模式与脚本模式

TypeScript 采用两种模式编译 TypeScript 文件：模块模式和脚本模式。具体编译为哪个模式，通过一项检测确定：文件中有没有 `import` 或 `export` 语句？如果有，使用模块模式；否则，使用脚本模式。

目前为止，我们使用的都是模块模式；多数时候，我们使用的都是这个模式。在模块模式下，我们使用 `import` 和 `import()` 从其他文件中引入代码，使用 `export` 把代码开放给其他文件使用。如果想使用第三方 UMD 模块（注意，UMD 模块尝试使用 CommonJS、RequireJS 或浏览器全局模块，具体取决于

环境支持哪个），必须先使用 `import` 将其导入，而且不能直接使用全局导出的代码。

在脚本模式下，声明的顶层变量在项目中的任何文件中都可以使用，无需导入；而且，可以放心使用第三方 UMD 模块中的全局导出，不用事先导入。下述情况使用脚本模式：

- 快速验证不打算编译成任何模块系统的浏览器代码（在 `tsconfig.json` 中设置 `{"module": "none"}`），在 HTML 文件中直接使用 `<script />` 标签引入。
- 创建类型声明（见 11.1 节）。

其实，在使用 TypeScript 编写代码时，基本上应该始终使用模块模式，通过 `import` 导入代码，通过 `export` 导出代码，供其他文件使用。

10.3 命名空间

TypeScript 还提供了另一种封装代码的方式：`namespace` 关键字。很多 Java、C#、C++、PHP 和 Python 程序员对命名空间肯定不陌生。



如果你以前使用的语言支持命名空间，要注意一点：虽然 TypeScript 支持命名空间，但这并不是封装代码的首选方式。如果你不确定该使用命名空间还是模块，选择模块准没错。

命名空间所做的抽象摒除了文件在文件系统中的目录结构，我们无须知道 `.mine` 函数保存在 `schemes/scams/bitcoin/apps` 文件夹中，若想使用这个函数，使用简洁的命名空间 `Schemes.Scams.Bitcoin.Apps.mine` 即可。^{注 1}

假设有两个文件，一个文件是发起 HTTP GET 请求的模块，另一个文件使用该模块发起请求：

注 1： 真希望这个笑话能一直流传下去，但我一点也不后悔没有投资比特币。

```
// Get.ts
namespace Network {
    export function get<T>(url: string): Promise<T> {
        // ...
    }
}

// App.ts
namespace App {
    Network.get<GitRepo>('https://api.github.com/repos/Microsoft/typescript')
}
```

命名空间必须有名称。命名空间可以导出函数、变量、类型、接口或其他命名空间。`namespace` 块中没有显式导出的代码为所在块的私有代码。由于命名空间可以导出命名空间，因此命名空间可以嵌套。假设 `Network` 模块的功能增多了，需要分成几个子模块。此时，可以使用命名空间：

```
namespace Network {
    export namespace HTTP {
        export function get <T>(url: string): Promise <T> {
            // ...
        }
    }

    export namespace TCP {
        listenOn(port: number): Connection {
            //...
        }
        // ...
    }

    export namespace UDP {
        // ...
    }

    export namespace IP {
        // ...
    }
}
```

现在，网络相关的功能都放在 `Network` 名下的子命名空间中。例如，我们可以在任何文件中调用 `Network.HTTP.get` 和 `Network.TCP.listenOn`。与接口一样，命名空间也可以分散在多个文件中。TypeScript 将递归合并名称相同的命名空间：

```

// HTTP.ts
namespace Network {
    export namespace HTTP {
        export function get<T>(url: string): Promise<T> {
            // ...
        }
    }
}

// UDP.ts
namespace Network {
    export namespace UDP {
        export function send(url: string, packets: Buffer): Promise<void> {
            // ...
        }
    }
}

// MyApp.ts
Network.HTTP.get<Dog[]>('http://url.com/dogs')
Network.UDP.send('http://url.com/cats', new Buffer(123))

```

如果命名空间的层次结构太深，为了使用上的方便，可以创建别名。注意，尽管句法相似，但是解构导入（比如导入 ES2015 模块）不支持别名。

```

// 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

```

10.3.1 冲突

导出相同名称的代码会导致冲突：

```
// HTTP.ts
namespace Network {
    export function request<T>(url: string): T {
        // ...
    }
}

// HTTP2.ts
namespace Network {
    // Error TS2393: Duplicate function implementation.
    export function request<T>(url: string): T {
        // ...
    }
}
```

不过，这条规则有个例外：改进函数类型时对外参（ambient）函数声明的重载不导致冲突。

```
// HTTP.ts
namespace Network {
    export function request<T>(url: string): T
}

// HTTP2.ts
namespace Network {
    export function request<T>(url: string, priority: number): T
}

// HTTPS.ts
namespace Network {
    export function request<T>(url: string, algo: 'SHA1' | 'SHA256'): T
}
```

10.3.2 编译输出

与导入和导出不一样，命名空间不遵守 `tsconfig.json` 中的 `module` 设置，始终编译为全局变量。下面通过实际输出的代码来理解这一句话到底是什么意思。以下述模块为例：

```
// Flowers.ts
namespace Flowers {
    export function give(count: number) {
```

```
        return count + ' flowers'  
    }  
}
```

经 TSC 编译后输出的代码如下：

```
let Flowers  
(function (Flowers) { ①  
    function give(count) {  
        return count + ' flowers'  
    }  
    Flowers.give = give ②  
})(Flowers || (Flowers = {})) ③
```

- ① `Flowers` 在 IIFE（立即调用的函数）中声明，创建一个闭包，防止没有显式导出的变量跳出 `Flowers` 模块。
- ② TypeScript 把导出的 `give` 函数分配给 `Flowers` 命名空间。
- ③ 如果全局作用域中已经定义了 `Flowers` 命名空间，TypeScript 将在此基础上扩充 (`Flowers`)；否则，TypeScript 创建 `Flowers` 命名空间，并进行扩充 (`Flowers = {}`)。

模块优于命名空间

为了更符合 JavaScript 标准，也为了更明确地指明依赖，较之命名空间，应该更多地使用常规的模块（可以 `import` 和 `export` 那种）。

明确指明依赖好处很多，可以提升代码可读性，可以强制模块隔离（命名空间自动合并，而模块不会），还可以做静态分析。不要小看这一点，在大型前端项目中，剔除无用代码、把编译得到的代码分拆到多个文件中，对性能的提升是至关重要的。

如果在 NodeJS 环境中运行 TypeScript 程序，模块也是更明智的选择，因为 NodeJS 内置对 CommonJS 的支持。在浏览器环境中，有些程序员为了方便而喜欢使用命名空间，但是在中大型项目中，建议全部使用模块，不要用命名空间。

10.4 声明合并

目前，我们接触到了 TypeScript 所做的三种合并：

- 合并值和类型，根据使用情况区分同一个名称引用的是值还是类型（见 6.3.4 节）。
- 把多个命名空间合并成一个。
- 把多个接口合并成一个（见 5.4.1 节）。

你可能猜到了，这是 TypeScript 中一个常规行为的三个特例。TypeScript 支持合并很多类型的名称，这让一些难以表达的模式变为可能（见表 10-1）。

表 10-1：声明可以合并吗？

		到							
		值	类	枚举	函数	类型别名	接口	命名空间	模块
从	值	否	否	否	否	是	是	否	—
	类	—	否	否	否	否	是	是	—
枚举	—	—	是	否	否	否	否	是	—
函数	—	—	—	否	是	是	是	是	—
类型别名	—	—	—	—	—	否	否	是	—
接口	—	—	—	—	—	—	是	是	—
命名空间	—	—	—	—	—	—	—	是	—
模块	—	—	—	—	—	—	—	—	是

可以看出，TypeScript 允许在同一个作用域中声明具有相同名称的值和类型别名，TypeScript 将根据上下文推导出具体指代的是类型还是值。正是由于这个特性，我们才能实现 6.3.4 节所讲的模式。另外，除了值和类型别名以外，还可以使用接口和命名空间实现伴生对象。不仅如此，我们还可以利用模块合并特性增强第三方模块声明（见 D.2 节），也可以借助合并特性通过命名空间为枚举添加静态方法（请你试试）。

moduleResolution 标志

眼尖的读者可能发现了，`tsconfig.json` 中有个 `moduleResolution` 标志。TypeScript 使用这个标志指定的算法解析应用中模块的名称。这个标志支持两种模式：

- `node`: 应该始终使用这个模式。在这个模式下，TypeScript 使用 NodeJS 所用的算法解析模块。以`.`、`/`或`~`开头的模块名称（例如`./my/file`）相对于本地文件系统解析，要么相对于当前文件，要么使用绝对路径（相对`/`目录或`tsconfig.json`中`baseUrl`设定的目录），具体要看以什么开头。如果模块路径不包含前述几个前缀，与 NodeJS 一样，TypeScript 从`node_modules`文件夹中加载模块。在 NodeJS 的解析策略之上，TypeScript 又增加了两个步骤：
 - a. NodeJS 在`package.json`中`main`字段指定的目录中寻找默认可导入的文件，除此之外，TypeScript 还会在`types`属性指定的包中搜寻类型声明（见 11.3 节）。
 - b. 如果导入的文件没有扩展名，TypeScript 首先寻找扩展名为`.ts`的同名文件，随后依次寻找扩展名为`.tsx`、`.d.ts`和`.js`的文件。
- `classic`: 不应该使用这个模式。在这个模式下，相对路径的解析方式与`node`模式一样，但是如果路径没有前缀，TypeScript 将在当前文件夹中寻找指定名称的文件，倘若找不到，沿着目录树向上一层，直至找到为止。对于从 NodeJS 或 JavaScript 转过来的人来说，这个行为十分怪异，而且与其他构建工具也无法很好地配合。

10.5 小结

本章介绍了 TypeScript 的模块系统，首先简要回顾了 JavaScript 模块系统的发展历史，说明了 ES2015 模块出现的时机，讨论了如何使用动态导入安全地加载代码、如何与 CommonJS 和 AMD 模块互操作，还对比了模块模式和脚

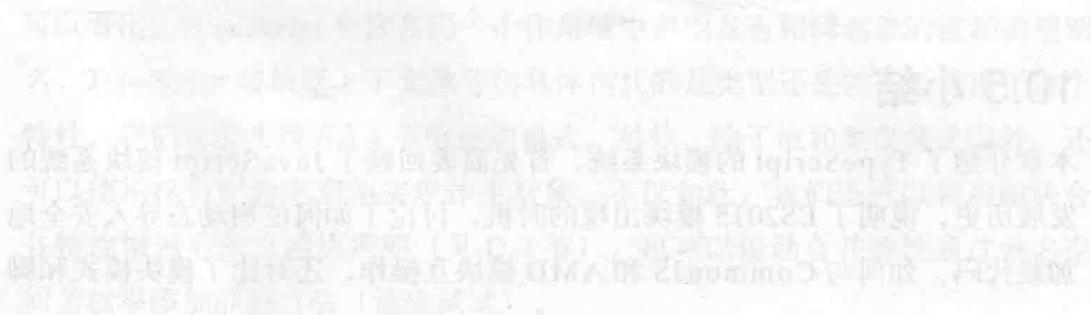
本模式。随后，讲解了命名空间、命名空间合并，讨论了 TypeScript 声明合并的运作方式。

如今，使用 TypeScript 开发应用，应该尽量使用 ES2015 模块。其实，TypeScript 并不介意你使用什么模块系统，但是使用 ES2015 模块更方便与构建工具集成（详见第 12 章）。

10.6 练习题

1. 借助声明合并完成下面两题：

- 把前面通过值和类型实现的伴生对象重新使用命名空间和接口实现。
- 为枚举添加静态方法。



与 JavaScript 互操作

我们生活的世界并不完美。咖啡可能太热，烫到嘴；经常可能没接到父母打来的电话，留下语音留言；多次向市政厅报告，车道上的坑洼可能还是没有填平。同样，代码也不一定都有静态类型。

很多人都遇到过这样的情况：业余时间决定使用 TypeScript 开发一个项目，虽然项目中用到的各个组件是安全的，但是我们要设法把组件嵌入到不那么安全的大型代码基之中。可能你的公司选择使用常规的 ES6 JavaScript，而你自己想通过 TypeScript 使用某个独立的组件；有些人受够了早上 6 点被叫起来，修改重构后忘记在调用的地方换用新代码的问题（现在已经上午 7 点，但是你决定在同事们上班之前把 TSC 合并到代码基中）。无论如何，刚开始使用 TypeScript 时，类型或许并不那么完善。

目前，本书一直在教你怎样正确地编写 TypeScript。而本章则讨论在实际开发中如何编写 TypeScript，毕竟你可能正在把代码基从无类型的语言向 TypeScript 迁移的过程之中，也有可能要使用第三方 JavaScript 库，或者为了紧急修补一个漏洞而牺牲一点类型安全性。本章专门探讨如何与 JavaScript 互操作，涵盖以下三个话题：

- 使用类型声明。

- 逐步从 JavaScript 迁移到 TypeScript。
- 使用第三方 JavaScript 和 TypeScript 库。

11.1 类型声明

类型声明文件的扩展名为 `.d.ts`。类型声明配合 JSDoc 注解（见 11.2.3 节），是为无类型的 JavaScript 代码附加 TypeScript 类型的一种方式。

类型声明的句法与常规的 TypeScript 代码类似，不过也有几点区别：

- 类型声明只能包含类型，不能有值。这意味着，类型声明不能实现函数、类、对象或变量，参数也不能有默认值。
- 类型声明虽然不能定义值，但是可以声明 JavaScript 代码中定义了某个值。此时，使用特殊的 `declare` 关键字。
- 类型声明只声明使用方可见的类型。如果某些代码不导出，或者是函数体内的局部变量，则不为其声明类型。

下面以一段 TypeScript 代码 (`.ts`) 为例，说明对应的类型声明 (`.d.ts`)。这个示例中的代码摘自流行的 RxJS 库，请不要在意细节，把注意力放在语言特性上（导入、类、接口、类字段、函数重载等）。

```
import {Subscriber} from './Subscriber'
import {Subscription} from './Subscription'
import {PartialObserver, Subscribable, TeardownLogic} from './types'

export class Observable<T> implements Subscribable<T> {
  public _isScalar: boolean = false
  constructor(
    subscribe?: (
      this: Observable<T>,
      subscriber: Subscriber<T>
    ) => TeardownLogic
  ) {
    if (subscribe) {
      this._subscribe = subscribe
    }
  }
}
```

```

    }
}

static create<T>(subscribe?: (subscriber: Subscriber<T>) => TeardownLogic) {
    return new Observable<T>(subscribe)
}

subscribe(observer?: PartialObserver<T>): Subscription
subscribe(
    next?: (value: T) => void,
    error?: (error: any) => void,
    complete?: () => void
): Subscription
subscribe(
    observerOrNext?: PartialObserver<T> | ((value: T) => void),
    error?: (error: any) => void,
    complete?: () => void
): Subscription {
    // ...
}
}

```

使用 TSC 编译，加上 declarations 标志（tsc -d Observable.ts），得到的类型声明 Observable.d.ts 如下所示：

```

import {Subscriber} from './Subscriber'
import {Subscription} from './Subscription'
import {PartialObserver, Subscribable, TeardownLogic} from './types'

export declare class Observable<T> implements Subscribable<T> { ①
    _isScalar: boolean
    constructor(
        subscribe?: (
            this: Observable<T>,
            subscriber: Subscriber<T>
        ) => TeardownLogic
    );
    static create<T>(
        subscribe?: (subscriber: Subscriber<T>) => TeardownLogic
    ): Observable<T>
    subscribe(observer?: PartialObserver<T>): Subscription
    subscribe(
        next?: (value: T) => void,
        error?: (error: any) => void,
        complete?: () => void
    ): Subscription ②
}

```

- ① 注意，`class` 前面有个 `declare` 关键字。在类型声明中虽然不可以真正定义类，但是却可以声明 `.d.ts` 文件对应的 JavaScript 文件中定义了指定的类。`declare` 可以理解为一种断言：“我发誓，我写的 JavaScript 代码导出了这个类型的类。”
- ② 由于类型声明中不能有具体实现，所以这里只保留了 `subscribe` 的两个重载，而没有各自实现的签名。

注意，`Observable.d.ts` 文件中是 `Observable.ts` 文件去掉实现后剩下的内容。也就是说，`Observable.d.ts` 文件中只有 `Observable.ts` 文件里的类型。

这个类型声明对 RxJS 库中使用 `Observable.ts` 的其他文件而言没有什么用，因为其他文件可以直接访问 `Observable.ts`，而该文件自身就是 TypeScript 源文件。然而，如果你在自己的 TypeScript 应用中使用 RxJS，类型声明就有用了。

可以想见，如果 RxJS 的开发人员想打包信息，发布到 NPM 中供 TypeScript 用户使用（RxJS 在 TypeScript 和 JavaScript 应用中都可以使用），有两种选择：第一，打包 TypeScript 源文件（供 TypeScript 用户使用）和编译得到的 JavaScript 文件（供 JavaScript 用户使用）；第二，提供编译得到的 JavaScript 文件和供 TypeScript 用户使用的类型声明。后一种方式所占的文件体积较小，而且十分明确该导入什么。另外，后一种方式还能减少编译应用所用的时间，因为编译应用时，TSC 不用重新编译 RxJS（12.1.5 节引入的优化策略正是基于这个原因）。

类型声明文件有以下几个作用：

1. 其他人在他们的 TypeScript 应用中使用你提供的编译好的 TypeScript 代码时，TSC 会寻找与生成的 JavaScript 文件对应的 `.d.ts` 文件，让 TypeScript 知道项目中涉及哪些类型。
2. 支持 TypeScript 的代码编辑器（例如 VSCode）会读取 `.d.ts` 文件，在输入代码的过程中显示有用的类型提示（即使用户不使用 TypeScript）。
3. 由于无须重新编译 TypeScript 代码，能极大地减少编译时间。

类型声明的作用是告诉 TypeScript，“JavaScript 文件中定义了这个，我来告诉你它的信息。”类型声明描述的是外参环境，与包含值的常规声明要区分开。例如，外参变量声明使用 `declare` 关键字声明 JavaScript 文件中定义了某个变量，而常规的变量声明使用 `let` 或 `const` 关键字声明变量。

借助类型声明可以做到以下几件事：

- 告诉 TypeScript，JavaScript 文件中定义了某个全局变量。假如你在浏览器环境中通过腻子脚本全局定义了 `Promise` 或 `process.env`，或许应该使用外参变量声明让 TypeScript 提前知道这一点。
- 定义在项目中任何地方都可以使用的全局类型，无须导入就能使用（这叫外参类型声明）。
- 描述通过 NPM 安装的第三方模块（外参模块声明）。

类型声明，不管作何用途，始终放在脚本模式下的 `.ts` 或 `.d.ts` 文件中（请回顾一下 10.2.3 节对脚本模式的讨论）。按约定，如果有对应的 `.js` 文件，类型声明文件使用 `.d.ts` 扩展名；否则，使用 `.ts` 扩展名。类型声明文件的名称没有具体要求，例如，笔者一般使用存储在顶层目录中的 `types.ts`，除非内容多到不可控，而且，一个类型声明文件中可以有任意多个类型声明。

最后要注意一点：在类型声明文件中，顶层值要使用 `declare` 关键字（`declare let`、`declare function`、`declare class` 等），而顶层类型和接口则不需要。

了解这些基础知识之后，下面通过一些例子简要说明每种类型声明。

11.1.1 外参变量声明

外参变量声明让 TypeScript 知道全局变量的存在，无须显式导入即可在项目中的任何 `.ts` 或 `.d.ts` 文件中使用。

假设你在浏览器中运行一个 NodeJS 程序，某个时刻，该程序会检查 `process.`

`env.NODE_ENV` 的值（为 "development" 或 "production"）。运行这个程序，你会看到如下的运行时错误：

```
Uncaught ReferenceError: process is not defined.
```

在 Stack Overflow 中查阅一番之后，你发现，让该程序正常运行最简单的方法是自己定义 `process.env.NODE_ENV`。为此，你新建一个文件，名为 `polyfills.ts`，在该文件中定义一个全局对象 `process.env`：

```
process = {
  env: {
    NODE_ENV: 'production'
  }
}
```

当然，TypeScript 会显示一条红色波浪线，告诉你不应该增强全局对象 `window`：

```
Error TS2304: Cannot find name 'process'.
```

不过，在这里，TypeScript 防备过当了。你确实想增强 `window` 对象，而且希望以一种安全的方式增强。

那么应该怎么做呢？你立即在 Vim 中打开 `polyfills.ts`（你发现了问题所在），输入：

```
declare let process: {
  env: {
    NODE_ENV: 'development' | 'production'
  }
}

process = {
  env: {
    NODE_ENV: 'production'
  }
}
```

你向 TypeScript 声明，有个全局对象名为 `process`，而且该对象有个属性，名为 `env`，该属性名下也有一个属性，名为 `NODE_ENV`。告诉 TypeScript 这些信息之后，那条红色波浪线就会消失，现在便可以安全地定义全局对象 `process` 了。



TSC 设置：lib

TypeScript 自带一些类型声明，用于描述 JavaScript 标准库，包括 JavaScript 内置的类型，例如 `Array` 和 `Promise`；内置类型的方法，例如 `'''.toUpperCase();` 以及一些全局对象，例如 `window` 和 `document`（浏览器环境）及 `onmessage`（Web 职程环境）。

如果需要，可以通过 `tsconfig.json` 文件中的 `lib` 字段引入这些内置的类型声明。`lib` 设置的详细说明参见“lib”一节。

11.1.2 外参类型声明

外参类型声明的规则与外参变量声明一样：外参类型声明保存在脚本模式下的 `.ts` 或 `.d.ts` 文件中，无须显式导入即可在项目中的其他文件里全局使用。举个例子，声明一个全局可用的 `ToArray<T>` 类型，把 `T` 变为数组（如果还不是数组的话）。这个类型可以在项目中任何一个使用脚本模式的文件里定义，比如说在项目顶层目录中的 `types.ts` 文件里定义：

```
type ToArray<T> = T extends unknown[] ? T : T[]
```

现在，在项目中的任何一个文件里都可以使用该类型，无须显式导入：

```
function toArray<T>(a: T): ToArray<T> {
  // ...
}
```

可以借助外参类型声明为应用中常用的数据类型建模。例如，可以让 6.7 节开发的 `UserID` 类型全局可用：

```
type UserID = string & {readonly brand: unique symbol}
```

现在，在应用中可以自由使用 UserID，无须先显式导入。

11.1.3 外参模块声明

使用 JavaScript 模块时，为了安全，你可能想随手声明一些类型。如果先给 JavaScript 模块在 GitHub 中的仓库或 DefinitelyTyped 提交类型声明，显然要麻烦一些。这时，便可以使用外参模块声明。

外参模块声明就是把常规的类型声明放在特殊的句法 `declare module` 中：

```
declare module 'module-name' {
    export type MyType = number
    export type MyDefaultType = {a: string}
    export let myExport: MyType
    let myDefaultExport: MyDefaultType
    export default myDefaultExport
}
```

模块名称（本例中的 '`module-name`'）是 `import` 导入的路径。导入这个路径后，TypeScript 便获知了外参模块声明提供的信息：

```
import ModuleName from 'module-name'
ModuleName.a // string
```

如果是嵌套模块，声明时要使用完整的导入路径：

```
declare module '@most/core' {
    // Type declaration
}
```

如果只想告诉 TypeScript，“我要导入这个模块，具体类型稍后确定，现在先假设为 `any`”，那么只保留头部，省略声明即可：

```
// 声明一个可被导入的模块，但是导入的类型为 any
declare module 'unsafe-module-name'
```

现在使用这个模块，安全性有点损失：

```
import {x} from 'unsafe-module-name'  
x // any
```

模块声明支持通配符导入，借此可以为匹配指定模式的任何导入路径声明类型。导入路径使用通配符 (*) 匹配：^{注1}

```
// 为 Webpack 的 json-loader 导入的 JSON 文件声明类型  
declare module 'json!*' {  
    let value: object  
    export default value  
}
```

```
// 为 Webpack 的 style-loader 导入的 CSS 文件声明类型  
declare module '*.css' {  
    let css: CSSRuleList  
    export default css  
}
```

现在，可以加载 JSON 和 CSS 文件了：

```
import a from 'json!myFile'  
a // object  
  
import b from './widget.css'  
b // CSSRuleList
```



为了让上述两例正常运行，要配置构建系统，允许加载.json 和.css 文件。虽然可以向 TypeScript 声明这些路径模式是可以安全导入的，但是 TypeScript 不会自作主张构建这些文件。

11.4.3 节将举例说明如何使用外参模块声明为没有类型信息的第三方 JavaScript 代码声明类型。

注 1：通配符的匹配规则与在常规的 glob 模式匹配 (https://en.wikipedia.org/wiki/Glob_programming) 中一样。

11.2 逐步从 JavaScript 迁移到 TypeScript

TypeScript 在设计之初就考虑到了与 JavaScript 的互操作问题，而不是后来增补的。虽然迁移到 TypeScript 不是一键就能完成的操作，但整个过程的体验还是不错的，我们可以一次转换一个文件，在迁移的过程中逐步提升代码基的安全等级，让老板和同事认识到静态类型对代码有多么大的影响。

总体上，最终我们希望整个代码基都使用 TypeScript 编写，达到类型全覆盖，而且依赖的第三方 JavaScript 库也带有高质量的严格类型信息。bug 应该尽量在编译时捕获，TypeScript 丰富的自动补全功能还能减少一半的编写代码时间。为了达到这样的状态，我们可以像蹒跚学步的幼儿一样，一次向前迈一小步：

- 在项目中添加 TSC。
- 对现有的 JavaScript 代码做类型检查。
- 把 JavaScript 代码改写成 TypeScript，一次改一个文件。
- 为依赖安装类型声明，没用到类型的依赖就算了，否则要自己动手为没有类型信息的依赖编写类型声明，然后提交给 DefinitelyTyped。^{注 2}
- 为代码基开启 strict 模式。

这个过程不是一蹴而就的，不过你立刻就能看到安全和效率上的提升，而且将不断发现更多的好处。下面分别讨论每一步。

11.2.1 第一步：添加 TSC

如果代码基中既有 TypeScript 也有 JavaScript，那就设置一下，也让 TSC 编译 JavaScript 文件。在 *tsconfig.json* 中这样设置：

```
{  
  "compilerOptions": {  
    "allowJs": true
```

注 2： DefinitelyTyped 是一个开源仓库，收集 JavaScript 库的类型声明。详情参见下文。

```
    }
}
```

这样设置之后，TSC 就能编译 JavaScript 文件了。把 TSC 添加到构建过程中，可以使用 TSC 分别编译现有的各个 JavaScript 文件，^{注3} 也可以继续使用原有构建过程运行 JavaScript 文件，同时使用 TSC 编译新编写的 TypeScript 文件。

把 `allowJs` 设为 `true`，TypeScript 不会对现有的 JavaScript 代码做类型检查，但是会使用指定的模块系统（参照 `tsconfig.json` 中的 `module` 字段）转译（`transpile`）JavaScript 文件（转义为 ES3、ES5 等，由 `tsconfig.json` 中的 `target` 字段设定）。第一步结束，可以提交了。给自己点个赞吧，代码基现在使用 TypeScript 了！

11.2.2 第二步（上）：对 JavaScript 代码做类型检查（可选）

既然让 TSC 处理 JavaScript 代码了，为什么不更进一步，对代码做类型检查呢？尽管 JavaScript 代码中没有显式类型注解，但是别忘了，TypeScript 能自动推导类型，就像在 TypeScript 代码中一样。在 `tsconfig.json` 中启用这个功能：

```
{
  "compilerOptions": {
    "allowJs": true,
    "checkJs": true
  }
}
```

现在，TypeScript 编译 JavaScript 文件时会尽量推导类型，并做类型检查，这一点与常规的 TypeScript 代码一样。

如果代码基较大，启用 `checkJs` 后一次可能报告一大堆类型错误。这时，请禁用该功能，分别在各个 JavaScript 文件中加入 `// @ts-check` 指令（一个普通的注释，放在文件顶部），一次只检查一个 JavaScript 文件。或者，如

注 3： 如果项目特别大，TSC 一个个编译 JavaScript 文件可能耗时较久。针对大型项目的性能优化措施，请阅读 12.1.5 节。

果一个文件的内容较多，抛出的错误很多，而我们暂时不想修正，可以保留 `checkJs` 设置，在这样的文件中加入 `// @ts-nocheck` 指令。



我们知道，TypeScript 不能推导出全部类型信息（例如函数的参数类型就推导不出来），因此 JavaScript 代码中将出现很多 `any` 类型。如果在 `tsconfig.json` 中启用了 `strict` 模式（应该启用），在迁移的过程中可以临时允许隐式 `any`。在 `tsconfig.json` 中加入：

```
{  
    "compilerOptions": {  
        "allowJs": true,  
        "checkJs": true,  
        "noImplicitAny": false  
    }  
}
```

大部分代码都迁移到 TypeScript 之后，别忘了启用 `noImplicitAny`，以免错过某些真正的错误（当然，除非你是 JavaScript 女巫 Bathmorda 的弟子 Xenithar，无须借助任何帮助，一锅艾蒿就能增强法力，扫一眼便能在大脑中做类型检查）。

TypeScript 处理 JavaScript 代码时，使用的推导算法比 TypeScript 代码宽容很多。具体规则如下：

- 所有函数的参数都是可选的。
- 函数和类的属性根据使用场景推导类型（无须事先声明）：

```
class A {  
    x = 0 // number | string | string[], 根据使用场景推导而出  
    method() {  
        this.x = 'foo'  
    }  
    otherMethod() {  
        this.x = ['array', 'of', 'strings']  
    }  
}
```

- 声明对象、类或函数之后，可以再附加额外的属性。在背后，TypeScript

为各个类和函数声明生成一个命名空间，为每个对象字面量自动添加一个索引签名。

11.2.3 第二步（下）：添加 JSDoc 注解（可选）

有时，你可能很匆忙，只想在现有的 JavaScript 文件中为一个新定义的函数添加类型注解。在腾出时间把 JavaScript 文件改写成 TypeScript 之前，你可以使用 JSDoc 为新增的函数添加类型注解。

或许你以前见过 JSDoc，就是那些看着有点奇怪的注释，放在 JavaScript 和 TypeScript 代码上面，以 @ 开头，例如 `@param`、`@returns` 等。TypeScript 能读懂 JSDoc，会把 JSDoc 当成类型检查器的输入，功效与显式注解 TypeScript 代码的类型一样。

假设有一个 3000 行代码的文件（是的，是你的“朋友”写的），你在其中新增了一个实用函数：

```
export function toPascalCase(word) {
    return word.replace(
        /\w+/g,
        ([a, ...b]) => a.toUpperCase() + b.join('').toLowerCase()
    )
}
```

在未把 `utils.js` 完全转换成 TypeScript 之前（转换之后说不定会捕获一堆需要修正的 bug），可以只注解 `toPascalCase` 函数，在无类型的 JavaScript 海洋中隔出一座安全小岛：

```
/**
 * @param word {string} An input string to convert
 * @returns {string} The string in PascalCase
 */
export function toPascalCase(word) {
    return word.replace(
        /\w+/g,
        ([a, ...b]) => a.toUpperCase() + b.join('').toLowerCase()
    )
}
```

如果没有 JSDoc 注解，TypeScript 将把 `toPascalCase` 的类型推导为 `(word: any) => string`。而现在，编译这段代码时，TypeScript 知道 `toPascalCase` 的类型是 `(word: string) => string`。此外，我们还得到了不错的文档。

TypeScript 支持的 JSDoc 注解参阅 TypeScript Wiki (<http://bit.ly/2YCTWBf>)。

11.2.4 第三步：把文件重命名为 .ts

把 TSC 添加到构建过程中以后，你可能开始做类型检查及注解 JavaScript 代码了。在这之后，该切换成 TypeScript 了。

一次一个文件，把文件的扩展名由 `.js`（或者 `.coffee`、`.es6` 等）改成 `.ts`。在代码编辑器中重命名之后，你会看到熟悉的红色波浪线（指的是 `TypeError`，不是电视上放的儿童剧），指出类型错误、大小写错误、遗忘的 `null` 检查，以及拼错的变量名。这些错误有两种修正策略：

1. 根治法：花点时间为结构、字段和函数添加正确的类型信息，捕获使用方可能出现的错误。如果启用了 `checkJs`，在 `tsconfig.json` 中启用 `noImplicitAny`，把 `any` 暴露出来，逐个修正，然后再禁用，以免对剩下的 JavaScript 文件做类型检查时输出太多错误。
2. 快速法：把 JavaScript 文件批量重命名为 `.ts` 扩展名，但是让 `tsconfig.json` 中的设置保持宽松一些（即把 `strict` 设为 `false`），尽量少抛出一些类型错误。为了让类型检查器喘口气，把复杂的类型声明为 `any`。修正余下的类型错误，然后提交。接下来，逐个开启严格模式相关的标志（`noImplicitAny`、`noImplicitThis`、`strictNullChecks` 等），修正出现的错误（这些标志的完整列表见附录 F）。



如果你选择快速法，建议定义一个外参类型声明 `TODO`，做为 `any` 的类型别名。添加类型信息时用 `TODO` 代替 `any`，这样便于找到并跟踪缺失的类型。当然，这种外参类型声明也可以使用更为具体的名称，这样便于在项目中做全局搜索：

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

上述两种方式差不多，具体使用哪一种由你自己决定。TypeScript 是一门渐进式类型语言，由底而上都考虑到了与无类型信息的 JavaScript 代码的互操作。不管是有严格类型信息的 TypeScript 与无类型信息的 JavaScript 代码互操作，还是有严格类型信息的 TypeScript 与类型信息不完善的 TypeScript 代码互操作，TypeScript 都将尽自己所能确保操作的安全性，确保在我们构建的严格类型海岛上，一切都是安全的。

11.2.5 第四步：严格要求

把大多数 JavaScript 代码迁移到 TypeScript 之后，为了保障代码的安全性，应该逐个启用更为严格的 TSC 标志（完整的标志列表见附录 F）。

最后，可以禁用与 JavaScript 互操作的 TSC 标志，强制要求所有代码都使用严格类型的 TypeScript 编写：

```
{
  "compilerOptions": {
    "allowJs": false,
    "checkJs": false
  }
}
```

这样修改之后，最后一批与类型相关的错误将浮出水面。修正这些错误，你将得到一个崭新且安全的代码基，多数资深的 OCaml 工程师会对此艳羡不已（前提是你不能太张狂）。

经过以上几步，我们为自己能控制的 JavaScript 代码添加了类型，可是那些不受我们控制的 JavaScript 代码怎么办，例如使用 NPM 安装的包？为达最终目的，我们要稍微绕点路……

11.3 寻找 JavaScript 代码的类型信息

在 TypeScript 文件中导入 JavaScript 文件（注意，在 TypeScript 中，“文件”和“模块”可以互换使用）^{注4}，TypeScript 按照下述算法查找 JavaScript 代码的类型声明：

1. 在同一级目录中寻找与 `.js` 文件同名的 `.d.ts` 文件。如果存在这样的文件，把该文件用作 `.js` 文件的类型声明。

假如文件夹结构如下所示：

```
my-app/
  └── src/
    |   └── index.ts
    |   └── legacy/
    |       |   └── old-file.js
    |       |   └── old-file.d.ts
```

我们在 `index.ts` 中导入了 `old-file`：

```
// index.ts
import './legacy/old-file'
```

TypeScript 将把 `src/legacy/old-file.d.ts` 用作 `./legacy/old-file` 的类型声明。

2. 如果不存在这样的文件，而且 `allowJs` 和 `checkJs` 的值为 `true`，推导 `.js` 文件的类型信息（由 `.js` 文件中的 JSDoc 注解得出）。
3. 如果无法推导，把整个模块视作 `any`。

然而，导入第三方 JavaScript 模块（即安装到 `node_modules` 中的 NPM 包）时，TypeScript 使用的算法稍有不同：

1. 在本地寻找模块的类型声明，找到就用。

假如应用的文件夹结构如下所示：

^{注4}: 严格来说，只对模块模式下的文件成立，对脚本模式下的文件不成立。详见 10.2.3 节。

```
my-app/
├── node_modules/
|   └── foo/
|       ├── index.ts
|       └── types.d.ts
```

types.d.ts 的内容如下所示：

```
// types.d.ts
declare module 'foo' {
    let bar: {}
    export default bar
}
```

如果导入 *foo*, TypeScript 将使用 *types.d.ts* 中的外参模块声明做为 *foo* 的类型信息源：

```
// index.ts
import bar from 'foo'
```

2. 如果在本地找不到，再分析模块的 *package.json*。如果该文件中定义了名为 *types* 或 *typings* 的字段，使用字段设置的 *.d.ts* 文件做为模块的类型声明源。
3. 如果没有上述字段，沿着目录结构逐级向上，寻找 *node_modules/@types* 文件夹，看看其中有没有模块的类型声明。

假如我们安装了 React：

```
npm install react --save
npm install @types/react --save-dev
```

```
my-app/
├── node_modules/
|   └── @types/
|       └── react/
|           └── react/
└── src/
    └── index.ts
```

导入 React 时，TypeScript 会找到 `@types/react` 文件夹，把它用作 React 的类型声明源：

```
// index.ts
import * as React from 'react'
```

4. 如果依然找不到类型声明，按照前面针对本地算法的 1~3 步查找。

步骤有点多，不过理解之后你会发现整个过程还是符合直观逻辑的。



TSC 设置：types 和 typeRoots

默认情况下，TypeScript 在项目根目录下的 `node_modules/@types` 文件夹及上级目录下的相应文件夹（`../node_modules/@types` 等）中寻找第三方类型声明。多数时候，这就是我们需要的行为。

若想修改寻找全局类型声明的默认行为，在 `tsconfig.json` 中配置 `typeRoots`，把值设为一个文件夹路径组成的数组，让 TypeScript 在这些文件夹中寻找类型声明。例如，可以让 TypeScript 在 `typings` 和 `node_modules/@types` 文件夹中寻找类型声明：

```
{
  "compilerOptions": {
    "typeRoots" : ["./typings", "./node modules/@types"]
  }
}
```

如果想更为细致地控制，在 `tsconfig.json` 中使用 `types` 选项指定希望 TypeScript 寻找哪个包的类型。例如，下述配置忽略所有第三方类型声明，唯有 React 除外：

```
{
  "compilerOptions": {
    "types" : ["react"]
  }
}
```

11.4 使用第三方 JavaScript



笔者假设你使用 NPM 或 Yarn 等包管理器安装第三方 JavaScript。自己动手复制粘贴代码可不是个好习惯。

使用 NPM 在项目中安装的第三方 JavaScript 大致可以分为三种情况：

1. 安装的代码自带类型声明。
2. 安装的代码没有自带类型声明，不过 DefinitelyTyped 中有相应的类型声明。
3. 安装的代码没有自带类型声明，而且 DefinitelyTyped 中也没有相应的类型声明。

下面分别讨论这三种情况。

11.4.1 自带类型声明的 JavaScript

如果一个包自带类型声明，而且项目中设置了 `{"noImplicitAny": true}`，那么导入时 TypeScript 不会显示红色的波浪线。

如果你安装的代码是由 TypeScript 编译而来的，或者作者很负责任，在 NPM 包中放入了类型声明，那就比较幸运，安装后可以直接使用。

比如说，以下几个 NPM 包自带了类型声明：

```
npm install rxjs
npm install ava
npm install @angular/cli
```



如果你安装的代码不是由 TypeScript 编译而来的，肯定有自带的类型声明与声明所描述的代码不匹配的风险。类型声明与源码打包在一起的话，这个风险比较低（尤其是流行的包），但也不是彻底无风险。

11.4.2 DefinitelyTyped 中有类型声明的 JavaScript

就算你导入的第三方代码没有自带类型声明，DefinitelyTyped (<https://github.com/DefinitelyTyped/DefinitelyTyped>) 中也可能有相应的类型声明。

DefinitelyTyped 是由社区维护的中心化仓库，为众多开源项目提供外参模块声明。

若想检查你安装的包在 DefinitelyTyped 中有没有类型声明，在 TypeSearch ([https://microsoft.github.io/TypeSearch/](https://microsoft.github.io>TypeSearch/)) 中搜索，或者直接尝试安装类型声明。 DefinitelyTyped 中的所有类型声明都已发布到 NPM 中，放在 @types 作用域下。因此，可以直接从该作用域下安装：

```
npm install lodash --save          # 安装 Lodash  
npm install @types/lodash --save-dev # 安装 Lodash 的类型声明
```

多数时候，执行 `npm install` 命令时应该加上 `--save-dev` 标志，把安装的类型声明添加到 `package.json` 文件里的 `devDependencies` 字段中。



由于 DefinitelyTyped 中的类型声明是由社区维护的，因此存在不完整、不准确或过时等风险。多数流行的包维护得比较好，不过如果你发现自己使用的类型声明可以改进，请花点时间改进，然后提交给 DefinitelyTyped (<http://bit.ly/2U7QYWP>)，让其他 TypeScript 用户能从你的劳动中获益。

11.4.3 DefinitelyTyped 中没有类型声明的 JavaScript

这是三种情况中最少见的。遇到这种情况，有好几种选择，从用时最少、最不安全的方案到用时最多、最安全的方案都有：

1. 在导入的文件中加上 `// @ts-ignore` 指令，把该文件加入白名单。TypeScript 允许使用无类型信息的模块，但这样的模块及其中的全部内容都是 `any` 类型：

```
// @ts-ignore
import Unsafe from 'untyped-module'

Unsafe // any
```

2. 创建一个空的类型声明文件，假装已为模块提供类型信息，把对该模块的使用加入白名单。假如你安装了少有人使用的 `nearby-ferret-alerter` 包，那就新建一个类型声明（例如 `types.d.ts`），写入下述外参类型声明：

```
// types.d.ts
declare module 'nearby-ferret-alerter'
```

这么做的目的是告诉 TypeScript，有这样一个可导入的模块 (`import alert from 'nearby-ferret-alerter'`)，但是 TypeScript 对该模块中的类型一无所知。这种方案比第一种稍好一些，毕竟在 `types.d.ts` 文件中集中列举了应用中无类型的模块。可是这样做依然不安全，因为 `nearby-ferret-alerter` 及其导出的全部代码仍是 `any` 类型。

3. 自己编写外参模块声明。与前一种方案一样，创建一个名为 `types.d.ts` 的文件，写入空声明 (`declare module 'nearby-ferret-alerter'`)。不过这一次还要填充类型声明。例如，该文件中的内容可能是这样的：

```
// types.d.ts
declare module 'nearby-ferret-alerter' {
    export default function alert(loudness: 'soft' | 'loud'): Promise<void>
    export function getFerretCount(): Promise<number>
}
```

现在，`import alert from 'nearby-ferret-alerter'` 之后，TypeScript 知道 `alert` 的类型是什么：不再是 `any` 了，而是 `(loudness: 'quiet' | 'loud') => Promise<void>`。

4. 自己编写类型声明，并且发布到 NPM 中。如果你选择了第三种方案，本地已经有类型声明了，那么建议你把自己的劳动成果发布到 NPM 中，让使用 `nearby-ferret-alerter` 包的其他人受益。为此，你可以向 `nearby-ferret-alerter` 包的 Git 仓库提交一个拉取请求，直接贡献自己编写的类

型声明；如果仓库的维护者不愿意投入精力维护 TypeScript 类型声明，你还可以把自己的类型声明贡献给 DefinitelyTyped。

为第三方 JavaScript 编写类型声明不难，但是具体过程要看模块的种类。为不同种类的 JavaScript 模块（NodeJS 模块、jQuery 功能增强、Lodash 混入、React 和 Angular 组件等）添加类型信息有不同的惯用方式。附录 D 给出了为第三方 JavaScript 模块添加类型信息的一些窍门。



为无类型的 JavaScript 自动生成类型声明是一个活跃的研究领域。例如，`dts-gen` (<https://www.npmjs.com/package/dts-gen>) 能自动为第三方 JavaScript 模块生成类型声明骨架。

11.5 小结

在 TypeScript 中使用 JavaScript 有好几种方式。各种方式概述见表 11-1。

表 11-1：在 TypeScript 中使用 JavaScript 的不同方式

方式	tsconfig.json 标志	类型安全性
导入无类型的 JavaScript	<code>{"allowJs": true}</code>	较差
导入并检查 JavaScript	<code>{"allowJs": true, "checkJs": true}</code>	尚可
导入并检查有 JSDoc 注解的 JavaScript	<code>{"allowJs": true, "checkJs": true, "strict": true}</code>	极好
导入带类型声明的 JavaScript	<code>{"allowJs": false, "strict": true}</code>	极好
导入 TypeScript	<code>{"allowJs": false, "strict": true}</code>	极好

本章讨论了把 JavaScript 和 TypeScript 放在一起使用的几个话题，先介绍了不同种类的类型声明及其用法，接着说明了如何把现有的 JavaScript 项目逐步迁移到 TypeScript，最后讨论了如何安全地使用第三方 JavaScript 代码（以及可能导致不安全的因素）。与 JavaScript 互操作是 TypeScript 最棘手的方面之一，经过本章的学习，你应该可以在自己的项目中运用自如了。

构建和运行 TypeScript

如果你曾在生产环境中部署并运行过 JavaScript 应用，也就知道如何运行 TypeScript 应用了，毕竟把 TypeScript 编译成 JavaScript 之后，两者没有什么区别。本章讨论如何构建 TypeScript 应用并部署到生产环境，不过 TypeScript 应用没有太多特别之处，基本上与 JavaScript 应用相当。本章分为四节，分别讨论：

- 构建 TypeScript 应用的过程。
- 构建 TypeScript 应用并在服务器中运行的方式。
- 构建 TypeScript 应用并在浏览器中运行的方式。
- 构建 TypeScript 应用并发布到 NPM 中的方式。

12.1 构建 TypeScript 项目

构建 TypeScript 应用很简单。本章讨论构建项目相关的核心理论，涵盖运行项目的所有环境。

12.1.1 项目结构

笔者建议把 TypeScript 源码放在顶级文件夹 `src/` 中，把编译结果放在顶级文件夹 `dist/` 中。这种文件夹结构是一种约定，比较流行。把源码和生成的代码

放在两个顶级文件夹中，在需要与其他工具集成时，你会从中受益的。另外，我们还能轻易地把生成的构建产物（artifact）排除在版本控制之外。

请尽量遵守这个约定：

```
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
```

12.1.2 构建产物

把 TypeScript 程序编译成 JavaScript 的过程中，TSC 会生成一些构建产物（见表 12-1）。

表 12-1：TSC 生成的构建产物

文件种类	扩展名	tsconfig.json 标志	默认生成？
JavaScript	.js	{"emitDeclarationOnly": false}	是
源码映射	.js.map	{"sourceMap": true}	否
类型声明	.d.ts	{"declaration": true}	否
声明映射	.d.ts.map	{"declarationMap": true}	否

第一种构建产物，即 JavaScript 文件，你应该不陌生。TSC 把 TypeScript 代码编译成 JavaScript 代码，以便在 JavaScript 平台（NodeJS 或 Chrome）中运行。执行 `tsc yourfile.ts` 命令时，TSC 先对 `yourfile.ts` 做类型检查，然后把它编译成 JavaScript。

第二种构建产物，即源码映射，是一种特殊的文件，它把生成的 JavaScript 文件中的每一部分链接回到对应 TypeScript 文件中的具体行和列。这种文件对调试代码有所帮助（Chrome DevTools 会显示 TypeScript 代码，而不是生成的 JavaScript 代码），还能把 JavaScript 异常的堆栈跟踪映射到 TypeScript 源码中的某行某列（为 12.1.6 节提到的工具提供源码映射，那些工具便会自动查找异常是 TypeScript 源码中的哪一行哪一列导致的）。

第三种构建产物，即类型声明，把生成的类型提供给其他 TypeScript 项目使用。

最后一种构建产物，即声明映射，能提升编译 TypeScript 项目的速度。详细说明见 12.1.5 节。本章余下的内容讨论如何以及为何要生成这些构建产物。

12.1.3 设置编译目标

JavaScript 是一门不太寻常的语言，规范演进的速度很快，一年一发布，而且作为程序员的我们并不总是能控制运行程序的平台使用的是哪个 JavaScript 版本。另外，很多 JavaScript 程序是同构的，即可以在服务器中运行，也可以在客户端中运行。这就导致了一系列问题：

- 在受我们自己控制的服务器中运行 JavaScript 后端程序，具体使用哪个 JavaScript 版本由我们自己决定。
- 可是，如果开源自己编写的 JavaScript 后端程序，使用方的平台中是哪个 JavaScript 版本就无从得知了。如果是 NodeJS 环境，我们还可以声明支持哪些 NodeJS 版本，但是浏览器环境就没那么幸运了。
- 如果在浏览器中运行 JavaScript，我们不可能知道用户使用的是什么浏览器，最新的 Chrome、Firefox 和 Edge 支持最新的 JavaScript 特性，这三个浏览器较旧的版本不支持最前沿的功能，陈旧的浏览器（例如 Internet Explorer 8）或嵌入式浏览器（例如 PlayStation 4 中运行的浏览器）更是拖后腿。我们只能精简功能，为其他功能提供腻子脚本，确保应用基本能用，并且尝试检测用户使用的浏览器是否过于陈旧，导致应用无法正常运行，如果是，显示一个消息，提醒用户升级。

- 如果发布同构的 JavaScript 库（例如既可在浏览器中也可以在服务器中运行的日志库），要指定最低支持 NodeJS 哪一版，以及支持哪些浏览器 JavaScript 引擎和版本。

不是所有 JavaScript 环境都原生支持所有 JavaScript 特性，但是我们依然应该尽量使用最新的 JavaScript 版本编写代码。对此，可以采用两种策略：

1. 把使用最新版 JavaScript 编写的应用转译成目标平台支持的旧版 JavaScript。在这个过程中，`for..of` 循环和 `async/await` 等语言特性将被分别转换成 `for` 循环和 `.then` 调用。
2. 为目标 JavaScript 运行时提供腻子脚本，支持较新的特性。通过这种方式可以提供 JavaScript 标准库缺少的功能（例如 `Promise`、`Map` 和 `Set`）和原型方法（例如 `Array.prototype.includes` 和 `Function.prototype.bind`）。

TSC 原生支持把代码转译成旧版 JavaScript，不过无法自动添加腻子脚本。一定要记住这一点：TSC 能把多数 JavaScript 特性转译成旧环境支持的版本，但是没有为缺少的特性提供实现。

为了指明想接入的目标平台，TSC 提供了三个设置：

- `target` 设定把代码转译成哪个 JavaScript 版本：`es5`、`es2015` 等。
- `module` 设定想使用的模块系统：`es2015` 模块、`commonjs` 模块、`systemjs` 模块等。
- `lib` 告诉 TypeScript，目标环境支持哪些 JavaScript 特性：`es5` 特性、`es2015` 特性、`dom`，等等。TypeScript 没有为这些特性提供具体的实现，这是腻子脚本的功用。`lib` 设置只是告诉 TypeScript，指定的特性可用（原生提供或通过腻子脚本实现）。

你要根据想运行应用的环境设置 `target` 和 `lib`，指明把代码转译成哪个 JavaScript 版本。如不确定，这两个设置都使用默认值 `es5`，通常是没有问题的。

把 `module` 设定为什么，取决于目标环境是 NodeJS 还是浏览器，如果是后者，还取决于你想使用什么模块加载器。



如果需要支持不太寻常的平台，请在 Juriy Zaytsev（又名 Kangax）制作的兼容表 (<http://kangax.github.io/compat-table/es5/>) 中查看目标平台原生支持哪些 JavaScript 特性。

下面深入讨论 `target` 和 `lib`, `module` 在 12.2 节和 12.3 节探讨。

target

TSC 内置的转译器支持把多数 JavaScript 特性转换成较旧的 JavaScript 版本，这意味着，使用最新的 TypeScript 版本编写的代码可以转译成想支持的任何 JavaScript 版本。TypeScript 支持最新的 JavaScript 特性（例如 `async/await`，但是截至目前，不是所有主流 JavaScript 平台都支持），为了把代码转换成 NodeJS 和浏览器当前能理解的版本，几乎离不开内置的转译器。

分别看一下 TSC 能把哪些 JavaScript 特性转译成较旧的 JavaScript 版本（见表 12-2），而哪些特性不能转译（见表 12-3）。^{注 1}



过去，每隔几年会发布新一版 JavaScript 语言，版本号逐渐递增（ES1、ES3、ES5、ES6）。不过从 2015 年开始，每一年都会发布新版本 JavaScript，而且版本号也变成了所在的年份（ES2015、ES2016 等）。然而，有些 JavaScript 特性先由 TypeScript 支持，而后才在某个 JavaScript 版本中定案，我们把这样的特性称为“ESNext”（即下一个版本）。

注 1：如果某个语言特性不能被 TSC 转译，而且目标环境也不支持，通常可以找到支持转译的 Babel 插件。为了找到最新的插件，请在你喜欢使用的搜索引擎中搜索“babel plugin <feature name>”。

表 12-2: TSC 支持转译的特性

版本	特性
ES2015	<code>const</code> 、 <code>let</code> 、 <code>for..of</code> 循环、数组 / 对象展开 (...)、带标签的模板字符串、类、生成器、箭头函数、函数默认参数、函数剩余参数、析构声明 / 赋值 / 参数
ES2016	幂运算符 (**)
ES2017	<code>async</code> 函数、 <code>await</code> <code>promise</code>
ES2018	<code>async</code> 迭代器
ES2019	<code>catch</code> 子句的可选参数
ESNext	数字分隔符 (123_456)

表 12-3: TSC 不支持转译的特性

版本	特性
ES5	对象读值方法和设值方法
ES2015	正则表达式的 <code>y</code> 和 <code>u</code> 标志
ES2018	正则表达式的 <code>s</code> 标志
ESNext	<code>BigInt</code> (123n)

设定转译目标的方法是，打开 `tsconfig.json`，把 `target` 字段设为：

- `es3`: ECMAScript 3。
- `es5`: ECMAScript 5（如不确定该使用哪个版本，使用这个默认值）。
- `es6` 或 `es2015`: ECMAScript 2015。
- `es2016`: ECMAScript 2016。
- `es2017`: ECMAScript 2017。
- `es2018`: ECMAScript 2018。
- `esnext`: 最新的 ECMAScript 版本。

例如，转译成 ES5：

```
{  
  "compilerOptions": {  
    "target": "es5"  
  }  
}
```

lib

前文说过，把代码转译成旧版 JavaScript 有一个问题：虽然多数语言特性可以安全转译（`let` 转译为 `var`, `class` 转译为 `function`），但是如果目标环境不支持较新的标准库特性，还要借助腻子脚本提供具体实现。比如说，`Promise` 和 `Reflect`，以及 `Map`、`Set` 和 `Symbol` 等数据结构。如果目标环境是最新版 Chrome、Firefox 或 Edge，通常无须使用腻子脚本，但是，如果目标环境是早前几版浏览器，或者多数 NodeJS 环境，就要借助腻子脚本提供缺少的特性。

幸好，我们无须自己动手编写腻子脚本。我们可以从流行的腻子脚本库中安装，例如 `core-js` (<https://www.npmjs.com/package/core-js>)，或者使用 Babel 运行通过类型检查的 TypeScript 代码，让 `@babel/polyfill` (<https://babeljs.io/docs/en/babel-polyfill>) 自动添加腻子脚本。



如果打算在浏览器中运行应用，为了减小 JavaScript 代码的体积，不要打包全部腻子脚本，要判断运行应用的浏览器到底是否需要某个腻子脚本，有可能目标平台已经支持某些通过腻子脚本实现的特性。建议使用 Polyfill.io (<https://polyfill.io/v2/docs/>) 这样的服务，只加载用户的浏览器需要的腻子脚本。

把腻子脚本添加到代码中之后，要告诉 TSC，目标环境一定支持相应的特性。具体方法是在 `tsconfig.json` 文件里的 `lib` 字段中设置。假如我们为所有 ES2015 特性和 ES2016 的 `Array.prototype.includes` 添加了腻子脚本，那么可以使用下述配置：

```
{  
  "compilerOptions": {  
    "lib": ["es2015", "es2016.array.includes"]  
  }  
}
```

```
    }
}
```

如果想在浏览器中运行代码，还要加上 `window`、`document`，以及在浏览器中运行代码所需的其他 API 的 DOM 类型声明：

```
{
  "compilerOptions": {
    "lib": ["es2015", "es2016.array.include", "dom"]
  }
}
```

若想查看所有支持的库，执行 `tsc --help` 命令。

12.1.4 生成源码映射

源码映射把转译得到的代码与对应的源码链接起来。多数开发者工具（如 Chrome DevTools）、错误报告库、日志框架和构建工具都支持源码映射。由于构建流程产出的代码通常与最初编写的代码有很大的差异（比如说，在构建流程中可能会把 TypeScript 编译成 ES5 JavaScript，再使用 Rollup 筛除无用的代码，接着使用 Prepack 预先求值，最后使用 Uglify 精简代码），因此在构建流程中使用源码映射对调试得到的 JavaScript 代码有很大的帮助。

一般来说，建议在开发环境中使用源码映射，也建议在浏览器和服务器等生产环境中提供源码映射。不过有个例外：如果你想通过混淆提高代码在浏览器中运行的安全性，那就不要在生产环境中提供源码映射。

12.1.5 项目引用

随着应用的增长，TSC 对代码做类型检查和编译所用的时间将越来越长。花费在类型检查和编译上的时间几乎与代码基的体量呈线性关系。在本地开发过程中，逐渐增多的编译时间将严重拖慢开发效率，让人不再愿意使用 TypeScript。

为了解决这个问题，TSC 内置了一个称为“项目引用”（project references）

的功能，能显著减少编译耗时。如果项目中有上百个文件，那就一定要使用项目引用。

项目引用的用法如下：

1. 把一个 TypeScript 项目分成多个子项目。一个子项目放在一个文件夹中，该文件夹中有一个 `tsconfig.json` 文件和一些 TypeScript 文件。拆分项目时，应该把可能一起更新的代码放在同一个文件夹中。
2. 在各个子项目的文件夹中创建一个 `tsconfig.json` 文件，至少写入下述内容：

```
{  
  "compilerOptions": {  
    "composite": true,  
    "declaration": true,  
    "declarationMap": true,  
    "rootDir": ".."  
  },  
  "include": [  
    "./**/*.ts"  
  ],  
  "references": [  
    {  
      "path": "../myReferencedProject",  
      "prepend": true  
    }  
  ]  
}
```

这里关键的设置是：

- `composite`: 告诉 TSC，这个文件夹是一个大型 TypeScript 项目的子项目。
- `declaration`: 告诉 TSC，为这个子项目生成 `.d.ts` 声明文件。在项目引用中，子项目可以访问各子项目的声明文件和生成的 JavaScript，但是不能访问 TypeScript 源文件。这就划定了一条界限，TSC 不会再重新检查或编译代码：如果更新了子项目 A 中的一行代码，TSC 不会对子项目 B 重新做类型检查和编译；TSC 所要做的只是检查 B 的类型声明中有没有类型错误。这个行为是项目引用在重新构建大型项目时用时较少的关键。

- **declarationMap**: 告诉 TSC，为生成的类型声明构建源码映射。
 - **references**: 在一个数组中列出该子项目依赖的其他子项目。每个引用的 path 字段要指向一个内有 *tsconfig.json* 文件的文件夹，或者直接指向一个 TSC 配置文件（如果配置文件的名称不是 *tsconfig.json*）。prepend 字段指明把引用的子项目生成的 JavaScript 和源码映射与当前子项目生成的 JavaScript 和源码映射拼接在一起。注意，**prepend** 只在使用 **outFile** 时有用；如未使用 **outFile**，可以不设置 **prepend**。
 - **rootDir**: 明确指明该子项目应该相对根项目（`.`）编译。另一种做法是设置 **outDir**，指向根项目的 **outDir** 中的一个子文件夹。
3. 在根文件夹中创建一个 *tsconfig.json* 文件，引用没有被其他子项目引用的子项目：
- ```
{
 "files": [],
 "references": [
 {"path": "./myProject"},
 {"path": "./mySecondProject"}
]
}
```

4. 现在，使用 TSC 编译项目时，通过 **build** 标志让 TSC 把项目引用考虑进来：
- ```
tsc --build # 或者简写为 tsc -b
```



写作本书时，项目引用是 TypeScript 的一个新功能，还有些瑕疵。使用项目引用时要注意以下几点：

- 克隆或重新获取之后要重新构建整个项目（使用 `tsc -b` 命令），重新生成缺少或过时的 `.d.ts` 文件。如果不这样做，那就把生成的 `.d.ts` 文件也检入仓库。
- 使用项目引用时，不要设置 `noEmitOnError: false`，TSC 始终把该设置设为 `true`。
- 自己动手确认一个子项目没有与多个子项目拼接在一起。否则，多次拼接的子项目将在编译输出中出现多次。注意，只引用而不拼接，那就没问题。

使用 `extends` 减少 `tsconfig.json` 中的样板代码量

有时，所有子项目的编译器选项是一样的。这种情况下，最好在根目录中创建一个基本的 `tsconfig.json` 文件，让子项目的 `tsconfig.json` 文件在此基础上扩展：

```
{  
  "compilerOptions": {  
    "composite": true,  
    "declaration": true,  
    "declarationMap": true,  
    "lib": ["es2015", "es2016.array.include"],  
    "rootDir": ".",
    "sourceMap": true,  
    "strict": true,  
    "target": "es5",  
  }  
}
```

然后，更新子项目的 `tsconfig.json` 文件，使用 `extends` 选项扩展：

```
{  
  "extends": "../tsconfig.base",  
  "include": [  
    "./**/*.ts"  
  ],  
  "references": [  
    {  
      "path": "../myReferencedProject",  
      "prepend": true  
    }  
  ],  
}
```

12.1.6 监控错误

TypeScript 在编译时会报告错误，不过我们还是需要在运行时监控用户可能遇到的异常，然后设法在编译时避免（至少也要修正导致运行时错误的缺陷）。若想报告和整理运行时异常，可以使用 Sentry (<https://sentry.io>) 和 Bugsnag (<https://bugsnag.com>) 等错误监控工具。

12.2 在服务器中运行 TypeScript

若想在 NodeJS 环境中运行 TypeScript 代码，把代码编译成 ES2015 JavaScript（如果目标环境使用旧版 NodeJS，编译为 ES5），并在 `tsconfig.json` 文件中把 `module` 字段设为 `commonjs`：

```
{  
  "compilerOptions": {  
    "target": "es2015",  
    "module": "commonjs"  
  }  
}
```

这样设置之后，ES2015 中的 `import` 和 `export` 调用将分别被编译成 `require` 和 `module.exports`，无须进一步打包就能在 NodeJS 中运行。

如果想使用源码映射（应该使用），要想办法把源码映射提供给 NodeJS 进程。为此，从 NPM 中安装 `source-map-support` 包 (<https://www.npmjs.com/package/source-map-support>)，然后按照说明设置即可。多数进程监控、日志和错误报告工具，例如 PM2 (<https://www.npmjs.com/package/pm2>)、Winston (<https://www.npmjs.com/package/winston>) 和 Sentry (<https://sentry.io>) 都内置对源码映射的支持。

12.3 在浏览器中运行 TypeScript

若想在浏览器中运行 TypeScript，编译过程比在服务器中运行要复杂一些。

首先，要为编译选择一个目标模块系统。经验表明，如果是发布一个库，供他人使用（例如发布到 NPM 上），应该始终使用 `umd`，以便最大程度上兼容各人在自己的项目中使用的不同的模块打包工具。

如果代码只供自己使用，不发布到 NPM 上，那么编译为哪种格式取决于你使用的模块打包工具。具体应该使用什么模块系统请参阅打包工具的文档，

例如 Webpack 和 Rollup 对 ES2015 模块支持最好，而 Browserify 要求使用 CommonJS 模块。下面给出一些参考：

- 如果使用 SystemJS (<https://github.com/systemjs/systemjs>) 模块加载器，把 `module` 设为 `systemjs`。
- 如果使用兼容 ES2015 的模块打包工具，例如 Webpack (<https://webpack.js.org>) 或 Rollup (<https://github.com/rollup/rollup>)，把 `module` 设为 `es2015` 或更高的版本。
- 如果使用兼容 ES2015 的模块打包工具，而且代码中用到了动态导入（见 10.2.1 节），把 `module` 设为 `esnext`。
- 如果构建供其他项目使用的库，而且经过 `tsc` 处理之后没有再通过额外的构建步骤处理，为了最大程度上兼容各种加载器，把 `module` 设为 `umd`。
- 如果使用 CommonJS 打包工具打包模块，例如 Browserify (<https://github.com/browserify/browserify>)，把 `module` 设为 `commonjs`。
- 如果打算使用 RequireJS (<https://requirejs.org>) 或其他 AMD 模块加载器加载代码，把 `module` 设为 `amd`。
- 如果希望导出的顶级代码可在 `window` 对象上全局使用（拥有特权的人就可以这样做），把 `module` 设为 `none`。注意，如果代码在模块模式下（见 10.2.3 节），为了减少其他软件工程师的痛苦，TSC 会强制把代码编译成 `commonjs` 模块。

接下来，配置构建流程，把所有 TypeScript 代码编译成一个 JavaScript 文件或一系列 JavaScript 文件。虽然设置 `outFile` 标志后，TSC 会把小型项目编译为一个 JavaScript 文件，但是只能打包成 SystemJS 和 AMD 模块。另外，与专门的构建工具（如 Webpack）不同，TSC 不支持构建插件和智能代码分拆，在某个阶段，你会发现需要使用功能更强大的打包工具。

鉴于此，对前端项目来说，一开始就建议使用功能更强大的构建工具。各种构建工具基本上都有 TypeScript 插件，例如：

- Webpack (<https://webpack.js.org>) 的 `ts-loader` (<http://bit.ly/2Gw3uH2>)。
- Browserify (<http://bit.ly/2IDpfGe>) 的 `tsify` (<http://bit.ly/2KOaZgw>)。
- Babel (<https://babeljs.io>) 的 `@babel/preset-typescript` (<http://bit.ly/2vc2Sjy>)。
- Gulp (<https://gulpjs.com>) 的 `gulp-typescript` (<http://bit.ly/2vanubN>)。
- Grunt (<https://gruntjs.com>) 的 `grunt-ts` (<http://bit.ly/2PgUXuq>)。

本书不详细讨论加快 JavaScript 包加载速度的优化措施，下面给出几点基本建议（不专门针对 TypeScript）：

- 保持代码模块化，避免隐式依赖（比如说修改 `window` 或其他全局对象），以便让构建工具更为准确地分析项目的依赖图。
- 使用动态导入惰性加载初次加载页面时不需要的代码，以免阻塞页面渲染。
- 合理利用构建工具提供的自动代码拆分功能，以免加载过多 JavaScript 代码，拖慢页面的加载速度。
- 制定衡量页面加载时间的策略，可以用合成的数据，当然最好用真实的用户数据。随着应用的增长，初次加载时间肯定会越来越慢，有一定的衡量方法才能想办法优化。在这方面，New Relic (<https://newrelic.com>) 和 Datadog (<https://www.datadoghq.com>) 是不错的工具。
- 针对生产环境的构建过程要尽量与开发过程中使用的构建过程一样。两者之间的差异越大，在生产环境中暴露出来的难以修复的 bug 越多。
- 最后，若是在浏览器中运行 TypeScript，要制定一定的策略，为浏览器缺少的特性添加腻子脚本。可以制作一个标准化的腻子脚本库，每次构建都打包，也可以根据用户浏览器的支持情况，动态打包腻子脚本。

12.4 把 TypeScript 代码发布到 NPM 中

编译 TypeScript 代码供其他 TypeScript 和 JavaScript 项目使用其实很简单。如果是编译成供他人使用的 JavaScript，有几点最佳实践要牢记于心：

- 生成源码映射，以便调试自己的代码。
- 编译为 ES5，让其他人能轻易构建并运行你的代码。
- 审慎选择编译为哪种模块格式（UMD、CommonJS、ES2015 等）。
- 生成类型声明，让其他 TypeScript 用户知晓你的代码中有哪些类型。

首先，使用 `tsc` 把 TypeScript 编译成 JavaScript，并且生成相应的类型声明。切记要配置 `tsconfig.json`，最大程度上兼容各种流行的 JavaScript 环境和构建系统（见 12.1 节）：

```
{  
  "compilerOptions": {  
    "declaration": true,  
    "module": "umd",  
    "sourceMaps": true,  
    "target": "es5"  
  }  
}
```

接下来，在 `.npmignore` 中列出不想发布到 NPM 中的 TypeScript 源码，尽量减少包的体积；在 `.gitignore` 中列出生成的构建产物，不纳入 Git 仓库：

```
# .npmignore  
  
*.ts # 忽略 .ts 文件  
!*.d.ts # 保留 .d.ts 文件  
  
# .gitignore  
  
*.d.ts # 忽略 .d.ts 文件  
*.js # 忽略 .js 文件
```



如果你沿用了前文推荐的项目结构，把源码放在 `src/` 文件夹中、把生成的文件放在 `dist/` 文件夹中，那么两个 `.ignore` 文件的内容更简单：

```
# .npmignore  
src/ # 忽略源文件  
  
# .gitignore  
dist/ # 忽略生成的文件
```

最后，在项目的 `package.json` 文件中添加 `"types"` 字段，指明该项目自带类型声明（注意，这一步不是必须的，不过加上 `"types"` 字段后使用 TypeScript 的用户将从中受益）。再添加一个脚本，在发布前构建包，确保包中的 JavaScript、类型声明和源码映射与编译源 TypeScript 是同步的。

```
{  
  "name": "my-awesome-typescript-project",  
  "version": "1.0.0",  
  "main": "dist/index.js",  
  "types": "dist/index.d.ts",  
  "scripts": {  
    "prepublishOnly": "tsc -d"  
  }  
}
```

整个过程结束！现在，执行 `npm publish` 命令把包推送到 NPM 中之后，NPM 将自动把 TypeScript 编译成既可被 TypeScript 用户使用（具有全面的类型安全性）也可由 JavaScript 用户使用的格式（如果编辑器支持的话，具有一定的类型安全性）。

12.5 三斜线指令

TypeScript 有个鲜为人知的特性，这个特性很少使用，也有点过时了，即三斜线指令。这种指令是格式特殊的 TypeScript 注释，作用是为 TSC 下达命令。

三斜线指令有好多种，本节只介绍其中两种：`types`，消隐只含类型的整个模块导入语句；`amd-module`，为生成的 AMD 模块命名。完整的指令清单，参阅附录 E。

12.5.1 types 指令

从一个模块中导入代码，在编译成 JavaScript 时，TypeScript 不一定会生成 `import` 或 `require` 调用，具体要看导入的是什么。如果 `import` 语句导入的代码只在模块的类型层面使用（即只从模块中导入类型），TypeScript 不会为这样的 `import` 语句生成任何 JavaScript 代码，毕竟导入的代码只位于类型层面。这个特性叫做“导入消隐”（`import elision`）。

然而，如果想利用导入的副作用，这个规则不再适用：若是导入整个模块（而不是只导入一部分导出的代码或通过通配符导入），TypeScript 会生成 JavaScript 代码。比如说，为了确保脚本模式下的模块中定义的外参类型可在程序中使用，就会这么做（6.8 节这样做过）。例如：

```
// global.ts
type MyGlobal = number

// app.ts
import './global'
```

执行 `tsc app.ts` 命令，把 `app.ts` 编译成 JavaScript，你会发现导入 `./global` 的语句没有消隐：

```
// app.js
import './global'
```

出现这样的导入语句，建议你先确定是否真的需要利用导入的副作用，而且没有办法改写代码，明确指定要导入哪个值或类型（例如，把 `import './global'` 改成 `import {MyType} from './global'`，让 TypeScript 消隐导入语句）。另外，还可以看一下能否在 `tsconfig.json` 文件里的 `types`、`files` 或 `include` 字段中设置外参类型，完全避免导入。

如果上述方法都不可行，而且仍想导入整个模块，但是却不希望生成 JavaScript `import` 或 `require` 调用，那就使用三斜线指令 `types`。三斜线指令以三条斜线 `///` 开头，后跟一个可用的 XML 标签；各 XML 标签有一些必须设置的属性。`types` 指令的用法如下：

- 声明依赖一个外参类型声明：

```
/// <reference types=".global" />
```

- 声明依赖 `@types/jasmine/index.d.ts`：

```
/// <reference types="jasmine" />
```

这个指令不常用。如果使用了，应该重新思考你在项目中使用类型的方式，找到一种少依赖外参类型的方式。

12.5.2 amd-module 指令

把 TypeScript 代码编译成 AMD 模块格式时（在 `tsconfig.json` 中设置了 `{"module": "amd"}`），TypeScript 默认生成匿名 AMD 模块。这种情况下，可以使用 AMD 三斜线指令为生成的模块命名。

假如有以下代码：

```
export let LogService = {
  log() {
    // ...
  }
}
```

编译为 `amd` 模块格式时，TSC 将生成如下的 JavaScript 代码：

```
define(['require', 'exports'], function(require, exports) {
  exports.__esModule = true
  exports.LogService = {
    log() {
      // ...
    }
})
```

```
    })  
})
```

对 AMD 模块格式熟悉的人应该发现了，这是一个匿名 AMD 模块。若想为 AMD 模块起个名称，在代码中使用三斜线指令 `amd-module`：

```
/// <amd-module name="LogService" /> ①  
export let LogService = { ②  
    log() {  
        // ...  
    }  
}
```

① 使用 `amd-module` 指令，设置 `name` 属性。

② 余下的代码保持不变。

再次使用 TSC 编译成 AMD 模块格式，生成的 JavaScript 变成了：

```
/// <amd-module name='LogService' />  
define('LogService', ['require', 'exports'], function(require, exports) {  
    exports.__esModule = true  
    exports.LogService = {  
        log() {  
            // ...  
        }  
    }  
})
```

编译成 AMD 模块时，可以使用 `amd-module` 指令让代码更易于打包和调试（如果可以，建议换用更现代化的模块格式，例如 ES2015 模块）。

12.6 小结

本章说明了构建及在生产环境中运行 TypeScript 应用的方方面面，涵盖浏览器和服务器环境。我们讨论了如何选择编译的 JavaScript 版本、把哪些库标记为在目标环境中可用（以及如何为缺少的特性添加腻子脚本库），以及如何构建并随应用一起分发源码映射，方便在生产环境中调试和本地开发。接着，

探讨了如何模块化 TypeScript 项目，提升编译速度。最后，介绍了如何在服务器和浏览器中运行 TypeScript 应用、如何把 TypeScript 代码发布到 NPM 中供他人使用、导入消隐背后的原理，以及如何使用三斜线指令为 AMD 模块命名。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

从本章的“对模块化声明”部分可知，通过 `module` 和 `export` 语句，可以在同一个文件中实现模块化。如果希望将模块化拆分到多个文件中，可以使用 `import` 和 `export` 语句。

总结

不进此书，你将对 TypeScript 一无所知。但本章将为你介绍一些关于 TypeScript 的基础知识，帮助你开始学习。本章将介绍 TypeScript 的基本概念、语法规则、类型推断、可赋值性、类型细化、类型拓宽和全面性检查。通过阅读本章，你将能够更好地理解 TypeScript 的核心思想，并在未来的开发工作中应用这些知识。

这段旅程即将告一段落。

我们一起学习了很多知识：什么是类型及类型的作用；TSC 的运作方式；TypeScript 支持哪些类型；TypeScript 的类型系统如何处理推导、可赋值性、类型细化、类型拓宽和全面性检查；根据上下文确定类型的规则；型变的方式；类型运算符的用法；函数、类、接口、迭代器、可迭代对象、生成器、重载、多态类型、混入、装饰器；在截止时间之前，为了写出代码而牺牲安全性可以采用的折中方案。我们还讨论了安全处理异常的不同方式，以及各种方式的优缺点；讨论了如何借助类型安全地进行并发、并行和异步编程。随后，介绍了如何在 Angular 和 React 等流行的框架中使用 TypeScript，以及如何使用命名空间和模块。最后，说明了在前后端开发中如何使用、构建和部署 TypeScript，讨论了逐步把代码迁移到 TypeScript 的方式、如何使用类型声明、如何把代码发布到 NPM 中供他人使用、如何安全地使用第三方代码，以及如何构建 TypeScript 项目。

希望你发现了静态类型的优势。希望你现在偶尔会在着手实现程序之前先规划类型，希望你对类型能让应用更安全有了更为深刻的认识。希望你看待事物的态度有了改变，哪怕只有一点点；希望你现在编写代码时会用类型思维思考问题。

目前，你已经掌握了足够的知识，可以教其他人使用 TypeScript 了。请向身边的人传播安全的重要性，为公司和朋友能写出更好的代码贡献一份力量，让编程充满更多乐趣。

最后，请继续探索未知。TypeScript 可能不是你接触的第一门语言，应该也不是最后一门。请继续学习编程的新方式、类型的新思想，发掘在安全性和易用性之间权衡的新方式。或许，继 TypeScript 之后，你会创造一种新思潮，说不定有一天笔者会为此而写一本书……

附录一 TypeScript 完美入门

本书着重于 TypeScript 的基础语法，从概念到实践，从浅入深地讲解了 TypeScript 的各个方面，帮助你快速掌握 TypeScript 的基本概念，理解 TypeScript 的语法规则，从而能够更高效地编写出高质量的 TypeScript 代码。本书还介绍了如何使用 TypeScript 进行单元测试，以及如何通过 TypeScript 的泛型功能来提高代码的可读性和可维护性。通过本书的学习，相信你能够掌握 TypeScript 的核心语法，并能够将其应用到实际项目中去。希望本书能够帮助你成为一名优秀的 TypeScript 开发者。

由于书中不可避免地会涉及一些高级话题，部分章节会稍显复杂，建议读者根据自己的需求选择阅读。同时，书中也提供了大量的示例代码，以便读者能够更好地理解 TypeScript 的语法和特性。希望本书能够成为你学习 TypeScript 的良师益友。

类型运算符

TypeScript 支持的类型运算符十分丰富，附表 A 给出了简要说明，如果想深入了解，请参照相应的章节。

附表 A：类型运算符

类型运算符	句法	适用于	详细说明
类型查询	<code>typeof</code> 、 <code>instanceof</code>	任何类型	6.1.5 节、5.6 节
键	<code>keyof</code>	对象类型	“keyof 运算符”一节
属性查找	<code>O[K]</code>	对象类型	键入“运算符”一节
映射类型	<code>[K in O]</code>	对象类型	6.3.3 节
加修饰符	<code>+</code>	对象类型	6.3.3 节
减修饰符	<code>-</code>	对象类型	6.3.3 节
只读修饰符	<code>readonly</code>	对象类型、数组 类型、元组类型	3.2.8 节、5.1 节、“只读数组和元 组”一节
可选修饰符	<code>?</code>	对象类型、元组类型、 函数的参数类型	3.2.8 节、3.2.11 节、4.1.1 节
条件类型	<code>?</code>	泛型、类型别名、 函数的参数类型	6.5 节
非空断言	<code>!</code>	可能为空的类型	6.6.2 节、6.6.3 节

附表 A：类型运算符（续）

类型运算符	句法	适用于	详细说明
泛型参数的默认值	=	泛型	4.2.6 节
类型断言	as、<>	任何类型	6.6.1 节、“const 类型”一节
类型防护	is	函数的返回类型	6.4.2 节

如果要向 TypeScript 提交一个新功能，首先需要向其他人提出。TypeScript 项目中没有一个正式的流程，但通常会为此编写一个脚本。

如果要向 TypeScript 提交一个新功能，首先需要向其他人提出。TypeScript 项目中没有一个正式的流程，但通常会为此编写一个脚本。

功能	提案问题	投票	状态
“子类或 keyof”	面向求值	Yes	完成
“非写入”大括号	投票	[100]	正在讨论
“只读”大括号	投票	[0]	待处理
“只读”属性	投票	[0]	待审查
“只读”方法	投票	[0]	待审查
“只读”构造函数	投票	[0]	待审查
“只读”类	投票	[0]	待审查
“只读”接口	投票	[0]	待审查
“只读”联合类型	投票	[0]	待审查
“只读”元组	投票	[0]	待审查
“只读”字面量对象	投票	[0]	待审查
“只读”枚举	投票	[0]	待审查
“只读”类成员	投票	[0]	待审查
“只读”方法成员	投票	[0]	待审查
“只读”属性成员	投票	[0]	待审查
“只读”接口成员	投票	[0]	待审查
“只读”元组成员	投票	[0]	待审查
“只读”字面量对象成员	投票	[0]	待审查
“只读”枚举成员	投票	[0]	待审查

实用类型

TypeScript 的标准库自带了一些实用的类型。附表 B 列出了目前可用的实用类型。

最新的实用类型列表参见 `es5.d.ts` (<http://bit.ly/2I0Ve2U>)。

附表 B：实用类型

实用类型	适用于	说明
<code>ConstructorParameters</code>	类构造方法类型	类构造方法的参数类型组成的元组
<code>Exclude</code>	并集类型	从一个类型中排除另一个类型
<code>Extract</code>	并集类型	选择可赋值给某个类型的子类型
<code>InstanceType</code>	类构造方法类型	使用 <code>new</code> 实例化类得到的实例的类型
<code>NonNullable</code>	可为空的类型	从类型中排除 <code>null</code> 和 <code>undefined</code>
<code>Parameters</code>	函数类型	函数的参数类型组成的元组
<code>Partial</code>	对象类型	把对象的所有属性设为可选的
<code>Pick</code>	对象类型	一个对象类型的子类型，只含对象的部分键
<code>Readonly</code>	数组类型、对象类型和元组类型	把对象的所有属性设为只读的，或者把数组或元组设为只读的
<code>ReadonlyArray</code>	任何类型	把指定类型变成不可变的数组
<code>Record</code>	对象类型	键的类型到值的类型的映射

附表 B：实用类型（续）

实用类型	适用于	说明
Required	对象类型	把对象的所有属性设为必需的
ReturnType	函数类型	函数的返回类型

限定作用范围的声明

为了建模类型和值，TypeScript 声明提供了丰富的行为。而且与在 JavaScript 中一样，这些行为可以通过多种方式重载。本篇附录介绍其中两个行为，概述哪些声明生成类型（哪些声明生成值），以及哪些声明可以合并。

C.1 是否生成类型？

有些 TypeScript 声明创建类型，有些创建值，还有些既创建类型也创建值。附表 C-1 给出了简单的参考。

附表 C-1：声明生成类型吗？

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

C.2 是否合并?

声明合并是 TypeScript 的一个核心行为。利用这个行为可以创建更生动的 API、更好地模块化代码，提升代码的安全性。

附表 C-2 与 10.4 节给出的表格一样，从中可以快速查阅 TypeScript 合并哪些声明。

附表 C-2：声明可以合并吗？

		到							
		值	类	枚举	函数	类型别名	接口	命名空间	模块
从	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

为第三方 JavaScript 模块 编写声明文件的技巧

本篇附录介绍为第三方模块提供类型信息时常用的方法和模式。深入讨论见 11.4.3 节。

由于模块声明文件保存在 `.d.ts` 文件中，而且文件中不能有值，因此声明模块类型时要使用 `declare` 关键字申明模块确实导出了指定类型的值。附表 D 简单列出了常规的声明及相应的类型声明。

附表 D: TypeScript 声明及相应的类型声明

.ts	.d.ts
<code>var a = 1</code>	<code>declare var a: number</code>
<code>let a = 1</code>	<code>declare let a: number</code>
<code>const a = 1</code>	<code>declare const a: 1</code>
<code>function a(b) { return b.toFixed() }</code>	<code>declare function a(b: number): string</code>
<code>class A { b() { return 3 } }</code>	<code>declare class A { b(): number }</code>
<code>namespace A {}</code>	<code>declare namespace A {}</code>
<code>type A = number</code>	<code>type A = number</code>
<code>interface A { b?: string }</code>	<code>interface A { b?: string }</code>

D.1 导出的类型

导出的方式不同，例如全局导出、ES2015 导出或 CommonJS 导出，编写声明文件的方式也不同。

D.1.1 全局导出

如果在一个模块中只为全局命名空间赋值，而没有导出任何代码，只需创建一个脚本模式的文件（见 10.2.3 节），在变量、函数和类声明前面加上 `declare`（其他声明，例如 `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 {}

// 全局类型声明
type GlobalType = number

// 全局接口
interface GlobalInterface {}
```

这些声明在项目中的任何文件里都可以使用，无须显式导入。比如说，在项目中的任何文件里都可以使用 `someGlobal`，无须事先导入，不过在运行时，`someGlobal` 要赋值给全局命名空间（浏览器中的 `window`，NodeJS 中的 `global`）。

注意，为了保证文件在脚本模式下，在声明文件中不能使用 `import` 和 `export`。

D.1.2 ES2015 导出

如果模块使用 ES2015 导出方式，即使用 `export` 关键字，把 `declare`（申明定义了全局变量）替换为 `export`（申明导出了 ES2015 绑定）即可：

```
// 默认导出
declare let defaultExport: SomeType
export default defaultExport

// 具名导出
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
}
```

D.1.3 CommonJS 导出

在 ES2015 之前，CommonJS 是模块的事实标准，而且，截至写作本书时，仍是 NodeJS 的模块标准。CommonJS 模块也使用 `export` 关键字导出代码，但是句法稍有不同：

```
declare let defaultExport: SomeType  
export = defaultExport
```

注意，我们是把要导出的代码赋值给 `export`，而不是把 `export` 用作修饰符（ES2015 模块采用这种方式）。

第三方 CommonJS 模块的类型声明中只能有一个导出语句。若想导出多个东西，要利用声明合并行为（见附录 C）。

例如，若想为多个导出声明类型，而且没有默认导出，可以导出一个命名空间：

```
declare namespace MyNamedExports {  
    export let someExport: SomeType  
    export type SomeType = number  
    export class OtherExport {  
        otherType: string  
    }  
}  
export = MyNamedExports
```

可是，如果 CommonJS 模块中既有默认导出也有具名导出呢？这时就要利用声明合并行为了：

```
declare namespace MyExports {  
    export let someExport: SomeType  
    export type SomeType = number  
}  
declare function MyExports(a: number): string  
export = MyExports
```

D.1.4 UMD 导出

为 UMD 模块提供类型信息的方式基本上与 ES2015 模块一样。唯一的区别是，若想让模块在脚本模式（见 10.2.3 节）下的文件中全局可用，要使用特殊的 `export as namespace` 句法。例如：

```
// 默认导出  
declare let defaultExport: SomeType  
export default defaultExport
```

```
// 具名导出
export class SomeExport {
    a: SomeType
}

// 类型导出
export type SomeType = {
    a: number
}

export as namespace MyModule
```

注意最后一行。有了这一行，项目中处于脚本模式下的文件可以在全局命名空间 `MyModule` 上直接使用该模块（无须事先导入）：

```
let a = new MyModule.SomeExport
```

D.2 扩展模块

扩展模块的类型声明不像为模块编写类型声明那么常见，不过在编写 jQuery 插件或 Lodash 混入时偶尔用得到。请尽量避免这样做，正确的做法是使用另一个模块。不要使用 Lodash 混入，建议使用常规的函数；不要使用 jQuery 插件……慢着，你居然还在使用 jQuery？

D.2.1 扩展全局代码

如果想扩展一个模块中的全局命名空间或接口，只需创建一个脚本模式文件（见 10.2.3 节），在该文件中增强功能。注意，这种方式只适用于接口和命名空间，因为 TypeScript 将自动处理合并。

举个例子：为 jQuery 增加 `marquee` 方法。首先，安装 `jquery`：

```
npm install jquery --save
npm install @types/jquery --save-dev
```

然后，在项目中新建一个文件，比如说命名为 `jquery-extensions.d.ts`。在该文

件中为 jQuery 的全局接口 JQuery 添加 marquee 方法（在 jQuery 的类型声明中查找一番之后，笔者发现 jQuery 在 JQuery 接口上定义方法）：

```
interface JQuery {  
    marquee(speed: number): JQuery<HTMLElement>  
}
```

现在，在使用 jQuery 的任何文件中都可以使用 marquee 方法（当然，还要为 marquee 添加运行时实现）：

```
import $ from 'jquery'  
$(myElement).marquee(3)
```

注意，6.8 节也是利用这种技术扩展内置全局接口的。

D.2.2 扩展模块

扩展模块导出的代码有点麻烦，而且陷阱更多：要为扩展正确声明类型，在运行时要以正确的顺序加载模块，而且在被扩展的模块的类型声明发生结构变化后要更新扩展的类型。

举个例子：为 React 新导出的函数声明类型。首先安装 React 及其类型声明：

```
npm install react --save  
npm install @types/react --save-dev
```

然后，利用模块合并行为（见 10.4 节），声明一个与 React 同名的模块：

```
import {ReactNode} from 'react'  
  
declare module 'react' {  
    export function inspect(element: ReactNode): void  
}
```

注意，与扩展全局代码的示例不同，这里对扩展文件的模式没有要求，模块模式或脚本模式都可以。

那么，怎样扩展某个模块导出的代码呢？受 ReasonReact (<https://reasonml.github.io/reason-react>) 启发，我们打算为 React 组件添加一个内置 reducer（为 React 组件声明一组可转换的状态）。写作本书时，React 的类型声明把 React.Component 类型声明为一个接口和一个类，使用 UMD 标准导出时，两者将合而为一：

```
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>)
        // ...
    }
    // ...
}
```

下面扩展 Component，增加 reducer 方法。在项目的根目录中创建 `react-extensions.d.ts` 文件，写入下述代码：

```
import 'react' ①

declare module 'react' { ②
    interface Component<P, S> { ③
        reducer(action: object, state: S): S ④
    }
}
```

- ① 导入 'react'，把该扩展文件切换到脚本模式。若想使用 React 模块，必须这么做。注意，切换到脚本模式还有其他方法，例如导入代码、导出代码，或者导出一个空对象 (`export {}`)，不一定非要导入 'react'。
- ② 声明 'react' 模块，告诉 TypeScript 我们想为 `import` 指定的路径声明类型。由于我们安装了 `@types/react`（这个包为同名的 'react' 路径定义了一个导出），因此 TypeScript 将把这个模块声明与 `@types/react` 提供的声明合并。

- ③ 声明 Component 接口，增强 React 提供的 Component 接口。根据接口合并规则（5.4.1 节），这里使用的签名要与 @types/react 中的完全一样。
- ④ 最后，声明 reducer 方法。

声明这些类型之后（假设在别处提供了运行时实现），能使用内置的 reducers 以类型安全的方式声明 React 组件了：

```
import * as React from 'react'

type Props = {
  // ...
}

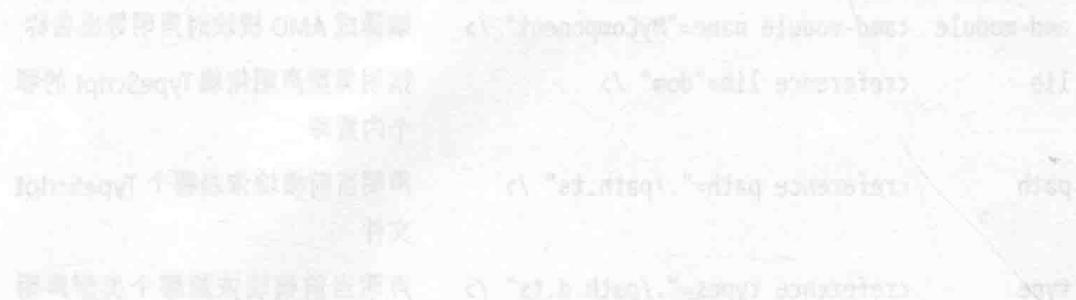
type State = {
  count: number
  item: string
}

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}
    }
  }
}
```

本节开头说过，应该尽量避免这样做（虽然功能很强大），因为这会把模块变得脆弱，而且对加载顺序有要求。相反，应该尝试使用组合模式，让模块扩展使用被扩展的模块。另外，不要修改模块，应该导出一个包装容器。

三斜线指令



三斜线指令其实就是常规的 JavaScript 注释，只不过 TypeScript 会据此调整所在文件的编译器设置，或者指明所在文件依赖了另一个文件。三斜线指令放在文件的顶部，位于所有代码前面。三斜线指令的用法如下（以三条斜线 `///` 开头，后跟一个 XML 标签）：

```
/// <directive attr="value" />
```

TypeScript 支持很多三斜线指令，最常用的一部分见附表 E-1。

amd-module

详见 12.5.2 节。

lib

`lib` 指令告诉 TypeScript 当前模块依赖 TypeScript 的哪个库，如果项目中没有 `tsconfig.json` 文件，就可以使用该指令。不过，在 `tsconfig.json` 文件中声明依赖的库是更好的选择。

path

如果为 TSC 指定了 `outFile` 选项，可以使用 `path` 指令声明依赖了另一个文件，这样在编译输出中被依赖的文件就会出现在当前文件的前面。如果在项目中使用 `import` 和 `export`，基本上用不到这个指令。

type

详见 12.5.1 节。

附表 E-1：三斜线指令

指令	句法	作用
amd-module	<amd-module name="MyComponent" />	编译成 AMD 模块时声明导出名称
lib	<reference lib="dom" />	指明类型声明依赖 TypeScript 的哪个内置库
path	<reference path="./path.ts" />	声明当前模块依赖哪个 TypeScript 文件
type	<reference types="./path.d.ts" />	声明当前模块依赖哪个类型声明文件

E.1 内部指令

在我们自己编写的代码中，基本上用不到 no-default-lib 指令（见附表 E-2）。

附表 E-2：内部三斜线指令

指令	句法	作用
no-default-lib	<reference no-default-lib="true" />	告诉 TypeScript，当前文件不使用任何内置库

E.2 弃用的指令

永远不要使用 amd-dependency 指令（见附表 E-3），应该始终使用常规的 import。

附表 E-3：弃用的三斜线指令

指令	句法	正确做法
amd-dependency	<amd-dependency path="./a.ts" name="MyComponent" />	import

安全相关的 TSC 编译器标志



完整的编译器标志列表参阅 TypeScript Handbook 网站 (<http://bit.ly/2JWfsgY>)。

每次发布 TypeScript 新版都会引入新的规则，力求进一步提升代码的安全性。以 `strict` 开头的标志囊括在 `strict` 标志之内，也可以逐个开启。附表 F 列出了写作本书时与安全有关的编译器标志。

附表 F：TSC 安全标志

标志	说明
<code>alwaysStrict</code>	生成 <code>'use strict'</code>
<code>noEmitOnError</code>	代码中存在类型错误时不生成 JavaScript
<code>noFallthroughCasesInSwitch</code>	确保 <code>switch</code> 语句的每个分支都返回值或跳出
<code>noImplicitAny</code>	如果变量的类型被推导为 <code>any</code> ，报错
<code>noImplicitReturns</code>	确保每个函数中的每个代码分枝都有显式返回。见 6.2 节
<code>noImplicitThis</code>	在函数中如果用到了 <code>this</code> ，但是没有显式注解 <code>this</code> 的类型，报错。见 4.1.4 节
<code>noUnusedLocals</code>	发现未用到的局部变量时发出提醒

附表 F: TSC 安全标志 (续)

标志	说明
noUnusedParameters	发现未用到的函数参数时发出提醒。在参数名称前加上 <code>_</code> , 可以忽略这样的错误
strictBindCallApply	为 <code>bind</code> 、 <code>call</code> 和 <code>apply</code> 强施类型安全。见 4.1.3 节
strictFunctionTypes	强制要求函数的参数和 <code>this</code> 可以逆变。见函数型变
strictNullChecks	把 <code>null</code> 提升为一个类型。见 3.2.12 节
strictPropertyInitialization	强制要求类的属性要么为空要么已经初始化。见第 5 章

内部指令

在你自己编写的代码中，若干工具不时地会使用一些指令，是用不上来的。这些指令的用途一目了然：顾名思义的先驱强推 `ignorant`（奇父先教）就是从 `Object` 的构造函数派生出来的，从名字就可以知道，它能通过 `Object` 的方法来操作对象。而 `getters` 和 `deleters` 则是与 `Object` 的属性相关的。

全局指令

全局指令是直接对整个语言的运行时环境起作用的。它们可以修改全局对象，也可以修改全局的语义。

如果 `useDefineForClass` 是真的，那么

全局变量 `Object` 就会变成一个类，从而可以使用类的语法。

如果 `useCallForConstruct` 是真的，那么

全局变量 `Object` 就会变成一个构造函数。

如果 `useDeleteForAssignment` 是真的，那么 `Object` 就会变成一个可赋值的全局变量。

如果 `useDeleteForAssignment` 是真的，那么 `Object` 就会变成一个可赋值的全局变量。

如果 `useDeleteForAssignment` 是真的，那么 `Object` 就会变成一个可赋值的全局变量。

TSX

什么是这个奇怪的 TSX 元素的类型？

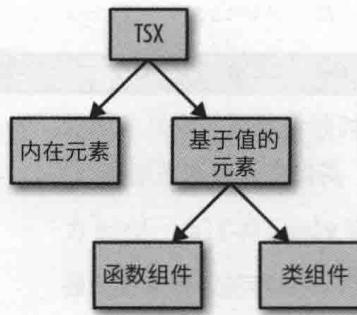
从技术上讲，这是一个严重的类型注解失败的类型。只要阅读一下 TypeScript 的代码库，你很快就会明白（<https://github.com/Microsoft/TypeScript/blob/v3.0.0/lib/tsc.ts#L100>），类型注解失败的原因是由于缺少对某些类型的处理。如果缺少对某些类型的处理，那么类型注解失败。

TypeScript 在底层开放了几个钩子，允许我们替换注解 TSX 类型的方式。这些类型在特殊的 `global.JSX` 命名空间中，TypeScript 使用该命名空间检查程序中的 TSX 类型。



如果你只是使用 React，无须知晓这些低层钩子。但是，假如你想自己开发使用 TSX 的 TypeScript 库（而不使用 React），可以通过本篇附录了解有哪些钩子可用。

TSX 支持两种元素：内置元素（内在元素）和用户定义的元素（基于值的元素）。内在元素的名称全为小写形式，表示原生元素，例如 ``、`<h1>` 和 `<div>`。基于值的元素使用 PascalCased 形式命名，表示我们使用 React（或者使用 TSX 的其他前端框架）创建的元素，可以通过函数或类定义。各种元素之间的关系如附图 G 所示。



附图 G: TSX 元素的种类

下面以 React 的类型声明 (<http://bit.ly/2CNzeW2>) 为例说明 TypeScript 如何通过钩子安全地注解 JSX 类型的:

```

declare global {
  namespace JSX {
    interface Element extends React.ReactElement<any> {} ①
    interface ElementClass extends React.Component<any> { ②
      render(): React.ReactNode
    }
    interface ElementAttributesProperty { ③
      props: {}
    }
    interface ElementChildrenAttribute { ④
      children: {}
    }
  }
  type LibraryManagedAttributes<C, P> = // ... ⑤

  interface IntrinsicAttributes extends React.Attributes {} ⑥
  interface IntrinsicClassAttributes<T> extends React.ClassAttributes<T> {} ⑦

  interface IntrinsicElements { ⑧
    a: React.DetailedHTMLProps<
      React.AnchorHTMLAttributes<HTMLAnchorElement>,
      HTMLAnchorElement
    >
    abbr: React.DetailedHTMLProps<
      React.HTMLAttributes<HTMLElement>,
      HTMLElement
    >
  }
}

```

```
address: React.DetailedHTMLProps<
  React.HTMLAttributes<HTMLElement>,
  HTMLElement
>
// ...
}
}
}
```

- ① JSX.Element 是一个基于值的 TSX 元素的类型。
- ② JSX.ElementClass 是一个基于值的类组件的实例的类型。只要声明打算使用 TSX 的 `<MyComponent />` 句法实例化的类组件，其类必须满足这个接口。
- ③ JSX.ElementAttributesProperty 指明 TypeScript 通过哪个属性找出一个组件支持哪些属性。对 React 来说，是 `props` 属性。TypeScript 在类的实例中寻找这个值。
- ④ JSX.ElementChildrenAttribute 指明 TypeScript 通过哪个属性找出一个组件支持哪些子元素。对 React 来说，是 `children` 属性。
- ⑤ JSX.LibraryManagedAttributes 指明可以声明和初始化属性类型的其他位置。对 React 来说，`propTypes` 是可以声明属性类型的其他位置，`defaultProps` 为属性声明默认值。
- ⑥ JSX.IntrinsicAttributes 是所有内在元素都支持的属性集合。对 React 来说，是 `key` 属性。
- ⑦ JSX.IntrinsicClassAttributes 是所有类组件（包括内在元素和基于值的元素）都支持的属性集合。对 React 来说，则是 `ref` 属性。
- ⑧ JSX.IntrinsicElements 列举出可以在 TSX 中使用的每一种 HTML 元素，把元素的标签名映射到属性和子元素的类型上。由于 JSX 不是 HTML，因此 React 的类型声明要告诉 TypeScript 在 TSX 表达式中具体可以使用什么元素，而且，每个标准的 HTML 元素都可以在 TSX 中使用，所以必须逐个列出各元素及其属性的类型（以 `<a>` 标签为例，有效的属性包括 `href: string` 和 `rel: string`，但是 `value` 无效），以及子元素的类型。

在全局命名空间 JSX 中声明这些类型，TypeScript 便会对 JSX 做类型检查，并按照指定的方式定制检查行为。倘若你不是编写使用 JSX 的库（而且不使用 React），一辈子可能都接触不到这些钩子。

TypeScript 编程

你肯定听使用动态类型语言的程序员讲过，随着代码行数和工程师数量的增多，弹性伸缩将越来越难。正是出于这方面的考虑，Facebook、Google 和 Microsoft 为 JavaScript 和 Python 套了一层渐进式静态类型。本书展示其中一种类型套层，即 TypeScript 的独特之处。借助 TypeScript 强大的静态类型系统，编程将变成一件充满乐趣的事。

本书针对 JavaScript 中级程序员，通过 Boris Cherny 的讲解，你将精通 TypeScript 语言，学会使用 TypeScript 摒除代码中的 bug，在工程人员增多后仍能保证代码可弹性伸缩。

内容要点：

- 学习基础知识：学习 TypeScript 的不同类型和类型运算符，了解其作用和用法。
- 探讨高级话题：理解 TypeScript 复杂的类型系统，学习如何安全地处理错误和构建异步程序。
- 联系实际应用：在你钟爱的前后端框架中使用 TypeScript，把现有的 JavaScript 项目迁移到 TypeScript，以及在生产环境中运行 TypeScript 应用。

Boris Cherny 就职于 Facebook，是工程和产品部门主管。他曾在风投公司、广告技术公司和一些初创公司工作。他喜欢研究编程语言、代码合成和静态分析，乐于构建让人钟爱的用户体验。

“这本书是深入学习 TypeScript 的不二之选。本书揭露了在 JavaScript 的基础上使用类型系统的好处，详解了精通 TypeScript 语言的要领。”

—Minko Gechev
Google Angular 团队的工程师

“通过这本书，我快速而有效地掌握了 TypeScript 的工具和生态系统。以前我在实际使用中遇到的各种问题，在这本书中都有详细的实例讲解。‘类型进阶’一章详细解释了我过去不懂的术语，而且展示了如何利用 TypeScript 提升代码的安全性，同时保障代码易于使用。”

—Sean Grove
OneGraph 联合创始人

PROGRAMMING / JAVASCRIPT

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-5198-4596-4



9 787519 845964 >

定价：88.00元