Краткое содержание

Отзывы 10

Пролог 12

Глава 1. Вступление 18

Глава 2. TypeScript с высоты птичьего полета 21

Глава 3. Подробно о типах 33

Глава 4. Функции 67

Глава 5. Классы и интерфейсы 111

Глава 6. Продвинутые типы 145

Глава 7. Обработка ошибок 198

Глава 8. Асинхронное программирование, конкурентность и параллельная обработка 215

Глава 9. Фронтенд- и бэкенд-фреймворки 247

Глава 10. Пространства имен и модули 265

Глава 11. Взаимодействие с JavaScript 283

Глава 12. Создание и запуск TypeScript 306

Глава 13. Итоги 327

Приложение А. Операторы типов 329

Приложение Б. Утилиты типов 330

Приложение В. Область действия деклараций 331

Приложение Г. Правила написания файлов деклараций для сторонних модулей JavaScript 333

Приложение Д. Директивы с тремя слешами 342

Приложение Е. Флаги безопасности компилятора TSC 344

Приложение Ж. TSX 346

Об авторе 349

Об обложке 350

Оглавление

Отзывы **10**

Пролог **12**

Структура книги 12
Стиль 13
Использование примеров кода 15
Благодарности 16
От издательства 17

Глава 1. Вступление 18

Глава 2. TypeScript с высоты птичьего полета 21

Компилятор 21
Система типов 23
Настройка редактора кода 27
index.ts 30
Упражнения к главе 2 32

Глава 3. Подробно о типах 33

О типах 34
Типы от а до я 35
Итоги 65
Упражнения к главе 3 66

Глава 4. Функции 67

Объявление и вызов функций 67
Полиморфизм 90
Разработка на основе типов 108
Итоги 109
Упражнения к главе 4 110

Глава 5. Классы и интерфейсы ... 111
Классы и наследование ... 111
super ... 117
Использование this в качестве возвращаемого типа ... 117
Интерфейсы ... 119
Классы структурно типизированы ... 126
Классы объявляют и значения, и типы ... 127
Полиморфизм ... 131
Примеси ... 132
Декораторы ... 135
Имитация финальных классов ... 138
Паттерны проектирования ... 139
Итоги ... 142
Упражнения к главе 5 ... 144

Глава 6. Продвинутые типы ... 145
Связи между типами ... 145
Тотальность ... 165
Продвинутые типы объектов ... 167
Продвинутые функциональные типы ... 177
Условные типы ... 180
Запасные решения ... 185
Имитация номинальных типов ... 191
Безопасное расширение прототипа ... 193
Итоги ... 196
Упражнения к главе 6 ... 197

Глава 7. Обработка ошибок ... 198
Возврат null ... 199
Выбрасывание исключений ... 200
Возврат исключений ... 203

Тип Option 205
Итоги 213
Упражнение к главе 7 214

Глава 8. Асинхронное программирование, конкурентность и параллельная обработка 215
Цикл событий 216
Работа с обратными вызовами 218
Промисы как здоровая альтернатива 221
async и await 227
Async-потоки 228
Типобезопасная многопоточность 231
Итоги 245
Упражнения к главе 8 246

Глава 9. Фронтенд- и бэкенд-фреймворки 247
Фронтенд-фреймворки 247
Типобезопасные API 260
Бэкенд-фреймворки 262
Итоги 264

Глава 10. Пространства имен и модули 265
Краткая история модулей JavaScript 266
import, export 269
Пространства имен 274
Слияние деклараций 279
Итоги 281
Упражнение к главе 10 282

Глава 11. Взаимодействие с JavaScript 283
Декларации типов 283
Поэтапная миграция из JavaScript в TypeScript 292
Поиск типов для JavaScript 298

Использование стороннего кода JavaScript 301
Итоги 305

Глава 12. Создание и запуск TypeScript 306
Создание проекта в TypeScript 306
Запуск TypeScript на сервере 317
Запуск TypeScript в браузере 318
Публикация TypeScript-кода на NPM 321
Директивы с тремя слешами 322
Итоги 326

Глава 13. Итоги 327

Приложение А. Операторы типов 329

Приложение Б. Утилиты типов 330

Приложение В. Область действия деклараций 331
Генерирует ли декларация тип 331
Допускает ли декларация слияние 331

Приложение Г. Правила написания файлов деклараций для сторонних модулей JavaScript 333
Типы экспорта 334
Расширение модуля 337

Приложение Д. Директивы с тремя слешами 342
Внутренние директивы 343
Нежелательные директивы 343

Приложение Е. Флаги безопасности компилятора TSC 344

Приложение Ж. TSX 346

Об авторе 349

Об обложке 350

Отзывы

Отличная книга для углубленного изучения TypeScript. Она демонстрирует все преимущества использования системы типов и помогает обрести уверенность при работе с JavaScript.

Минко Гечев,
инженер команды Angular в Google

Книга «Профессиональный TypeScript. Разработка масштабируемых JavaScript-приложений» помогла мне быстро освоить инструменты и внутреннее устройство этого языка. Она дала ответы на все мои вопросы с помощью реальных примеров. Глава «Продвинутые типы» сломала терминологические барьеры и показала, как TypeScript позволяет создать безопасный и удобный код.

Шон Гров,
сооснователь OneGraph

Борис создал обширное руководство по TypeScript. Прочтите его вдоль и поперек. А затем еще разочек.

Блейк Эмбри, инженер в Opendoor,
автор TypeScript Node and Typings
(«Типизации и Node в TypeScript»)

Посвящается Саше и Михаилу.
Возможно, и они однажды полюбят типы.

Пролог

Эта книга предназначена для программистов всех направлений: JavaScript-инженеров, C#-разработчиков, сторонников Java, любителей Python, Ruby и Haskell. На каком бы языке вы ни писали, если вам известны функции, переменные, классы и связанные с ними ошибки, то эта книга для вас. Наличие некоторого опыта в JavaScript, включая базовые знания об объектной модели документа (DOM) и обмене информацией по сети, поможет вам освоить представленный материал. Хоть мы и не сильно углубляемся в эти понятия, на них основаны приведенные примеры.

Работая с любым языком программирования, мы отслеживаем исключения и вычитываем код строку за строкой в поиске неисправности и способа ее устранения. TypeScript позволяет автоматизировать эту неприятную часть процесса разработки.

Если раньше вы не встречались со статической типизацией, то узнаете о типах из этой книги. Я расскажу, как с их помощью снизить риск программного сбоя, улучшить документирование кода и масштабировать его для большего числа пользователей, инженеров и серверов. Без сложных терминов я объясню материал, подключая вашу интуицию и память, и в этом мне помогут примеры.

TypeScript, в отличие от множества других типизированных языков, преимущественно практический. Он вводит новые концепции, позволяющие выражать идеи более кратко и точно и играючи создавать современные приложения безопасным способом.

Структура книги

Я постарался передать вам теоретическое понимание работы TypeScript и достаточное количество практических советов по написанию кода.

TypeScript — практический язык, поэтому в книге теория и практика в основном дополняют друг друга, но в первых двух главах преимущественно освещается теория, а ближе к концу представлена только практика.

Мы рассмотрим такие основы, как компилятор, модуль проверки типов и сами типы. Далее обсудим их разновидности и операторы, после чего перейдем к углубленным темам, таким как особенности системы типов, обработка ошибок и асинхронное программирование. В завершение я расскажу, как использовать TypeScript с вашими любимыми фреймворками (фронтенд и бэкенд), производить миграцию существующего JavaScript-проекта в TypeScript и запускать TypeScript-приложение в продакшене.

Большинство глав завершаются набором упражнений. Попробуйте выполнить их самостоятельно, чтобы лучше усвоить материал. Ответы к ним доступны по адресу https://github.com/bcherny/programming-typescript-answers.

Стиль

В коде я старался придерживаться единого стиля, поэтому отмечу, какие мои личные предпочтения он содержит.

- Точка с запятой только при необходимости.
- Отступы двумя пробелами.
- Короткие имена переменных вроде `a`, `f` или `_` там, где программа является фрагментом кода или ее структура важнее деталей.

Я считаю, что некоторых аспектов этого стиля стоит придерживаться и вам. К ним относятся:

- Использование синтаксиса JavaScript и новейших возможностей этого языка (последняя версия JavaScript обычно называется esnext). Так ваш код будет соответствовать последним стандартам и иметь хорошую функциональную совместимость с поисковыми системами. Это поможет снизить временные затраты на набор новых сотрудников и позволит оценить все преимущества таких мощных инструментов, как стрелочные функции, промисы и генераторы.
- Использование оператора расширения `spread` для длительного сохранения неизменности структур данных[1].

[1] Если у вас нет опыта работы в JavaScript, то вот пример: чтобы объекту `o` добавить свойство `k` со значением `3`, можно либо изменить `o` напрямую — `o.k = 3`, либо применить это изменение к `o`, создав тем самым *новый* объект — `let p = {...o, k: 3}`.

- Привычка убеждаться, что у любого элемента есть тип, по возможности выведенный. Старайтесь не злоупотреблять явными типами. Так вы сделаете код лаконичным и чистым, а также повысите его безопасность, выявляя неверные типы, а не используя временные решения.
- Стремление сохранить код переиспользуемым и обобщенным (см. раздел «Полиморфизм» на с. 90).

Конечно, все это не ново, но для TypeScript имеет особую актуальность. Его встроенный низкоуровневый компилятор, поддержка типов `readonly` (только для чтения), мощный интерфейс типов, глубокая поддержка полиморфизма и полностью структурная система типов позволяют добиться хорошего стиля, в то время как сам язык остается выразительным и верным для лежащего в его основе JavaScript.

Еще пара заметок до перехода к основному материалу.

JavaScript не предоставляет указатели и ссылки. Вместо них используются значения и ссылочные типы. Значения являются неизменными и включают такие элементы, как *strings (*строки), *numbers* (числа) и *booleans* (логические значения), в то время как ссылочные типы часто указывают на изменяемые структуры данных вроде массивов, объектов и функций. Когда я использую слово «значение», то обычно имею в виду либо значение JavaScript, либо ссылку.

И последнее. Написание идеального TypeScript-кода затрудняется взаимодействием с JavaScript, некорректно типизированными сторонними библиотеками и наследованным кодом. Также ему мешает спешка. В книге я буду часто советовать вам избегать компромиссов. В реальности достаточную корректность кода будете определять только вы и ваша команда.

Типографские соглашения

Ниже приведен список используемых обозначений.

Курсив

Используется для обозначения новых терминов.

`Моноширинный`

Применяется для оформления листингов программ и программных элементов в обычном тексте, таких как имена переменных и функций, базы данных, типы данных, переменные окружения, инструкции и ключевые слова.

`Моноширинный жирный`

Обозначает команды или другой текст, который должен вводиться пользователем. Иногда используется в листингах для привлечения внимания.

Так обозначаются примечания общего характера.

Так выделяются советы и предложения.

Так обозначаются предупреждения и предостережения.

Использование примеров кода

Вспомогательный материал (примеры кода, упражнения и пр.) доступен для загрузки по адресу: https://github.com/bcherny/programming-typescript-answers.

В общем случае все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабаты-

ваете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам следует получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Написание этой книги заняло год и потребовало множества бессонных ночей, ранних подъемов и потерянных выходных. В ее основе лежат многолетние наработки — сниппеты и заметки, собранные воедино, доработанные и обогащенные теорией.

Хочу поблагодарить O'Reilly за предоставленную возможность работы над книгой и лично редактора Анжелу Руфино за поддержку на протяжении всего процесса. Благодарю Ника Нэнса за помощь в создании раздела «Типобезопасные API» и Шьяма Шешадри за помощь с подразделом «Angular» (см. с. 256). Спасибо моим научным редакторам: Даниэлю Розенвассеру из команды TypeScript, который провел огромное количество времени, вычитывая рукопись и знакомя меня с нюансами системы типов, а также Джонатану Кримеру, Якову Файну, Полу Байингу и Рэйчел Хэд за технические правки и обратную связь. Спасибо моей семье — Лизе и Илье, Вадиму, Розе и Алику, Фаине и Иосифу — за вдохновение на реализацию этого проекта.

Самую большую благодарность выражаю Саре Гилфорд, которая поддерживала меня на протяжении всего процесса написания, даже когда отменялись планы на вечер и выходные или я внезапно начинал рассуждать на тему различных деталей системы типов. Я бесконечно благодарен ей за поддержку, без которой точно не справился бы.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Вступление

Итак, вы купили книгу о TypeScript. Зачем она нужна?

Вероятно, затем, что вам изрядно надоели странные ошибки в JavaScript вроде *cannot read property... of undefined* («невозможно прочесть свойство... принадлежащее *undefined*»). Или вы слышали, что TypeScript помогает масштабировать код, и решили изучить этот вопрос. Или вы работаете в C# и подумываете о переходе на JavaScript. А может, занимаетесь функциональным программированием и настало время повысить свой уровень. Или эту книгу вы получили в подарок на Новый год от босса, который устал от проблем, вызванных вашим кодом (если я перегибаю, остановите меня).

TypeScript — это язык будущих веб- и мобильных приложений, проектов NodeJS и IoT (систем интернет-управления устройствами). Он позволяет создавать более безопасные программы, обеспечивать их документацией, полезной и вам, и будущим инженерам, поддерживает безболезненный рефакторинг, а также избавляет от необходимости проводить половину модульных, или юнит-тестов. (Каких еще модульных тестов?) TypeScript может удвоить вашу продуктивность и даже устроить свидание с той милой бариста из кафе напротив.

Но прежде, чем спешить к ней, разберемся с упомянутыми преимуществами. Что конкретно я имею в виду, когда говорю «более безопасные»? Конечно, речь идет о *безопасности типов*.

БЕЗОПАСНОСТЬ ТИПОВ

Использование типов для предотвращения неверного поведения программ[1].

[1] В каждом отдельном статически типизированном языке «неверное поведение» может означать разное, начиная от программ, дающих сбой при запуске, и заканчивая рабочими, но бессмысленными приемами.

Вот несколько примеров неверного поведения кода:

- Перемножение числа и списка.
- Вызов функции со списком строк, когда требуется список объектов.
- Вызов метода для объекта, когда фактически данный метод не существует в этом объекте.
- Импортирование модуля, который недавно был перемещен.

Некоторые языки программирования стремятся извлечь пользу из подобных ошибок — стараются понять, что вы имели в виду. Возьмем, к примеру, такой JavaScript-код:

```
3 + []            // Вычисляется как строка "3"

let obj = {}
obj.foo           // Вычисляется как undefined

function a(b) {
  return b/2
}
a("z")            // Вычисляется как NaN
```

JavaScript делает все возможное, чтобы избежать исключения. Оказывается ли он полезен? Определенно да. Но можете ли вы при этом быстро и точно обнаруживать баги? Скорее всего, нет.

А теперь представьте, что JavaScript выдает больше исключений вместо попытки извлечь максимум из того, что мы ему дали. Тогда получим подобную реакцию:

```
3 + []            // Ошибка: вы точно хотели добавить число и массив?

let obj = {}
obj.foo           // Ошибка: вы забыли определить свойство "foo" в obj.

function a(b) {
  return b/2
}
a("z")            // Ошибка: функция "a" ожидает число,
                  // но вы передали ей строку.
```

Не поймите меня превратно: попытка исправить за нас наши же ошибки — это приятная особенность языка программирования (ох, если бы она работала не только для программ). Но в JavaScript она создает разрыв между моментом, когда вы допускаете ошибку, и временем ее *обнаружения*. Доходит до того, что вы узнаете об ошибке от кого-то другого.

Когда именно JavaScript сообщает об ошибке?

Или при запуске программы для тестирования в браузере, или при посещении сайта пользователем, или в начале модульного тестирования. Если вы пишете множество модульных и полных тестов, проверяя код на работоспособность, то можно рассчитывать на то, что вы увидите ошибки раньше пользователей. Но что, если вы этого не делаете?

Вот здесь-то и появляется TypeScript. И самое крутое в том, что он выдает сообщения об ошибках в момент их появления (во время типизации) в вашем редакторе. Посмотрим, что он скажет по предыдущему примеру:

```
3 + []          // Ошибка TS2365: оператор '+' не может быть применен
                // для типов '3' и 'never[]'.

let obj = {}
obj.foo       // Ошибка TS2339: свойство 'foo' не существует в типе '{}'.

function a(b: number) {
  return b / 2
}
a("z")          // Ошибка TS2345: аргумент типа '"z"' не может быть
                // присвоен параметру типа 'number'.
```

Он не только устраняет целый класс багов, связанных с типами, но и изменяет подход к написанию кода. Вы начнете делать наброски программы на уровне типов еще до начала ее наполнения на уровне значений[1], обдумывать пограничные случаи уже во время ее проектирования. В итоге вы станете проектировать более простые, быстрые, понятные и легко обслуживаемые программы.

Если вы готовы начать путешествие, тогда приступим!

---

[1] Если вы не совсем понимаете, что в данном случае значит «уровень значений», не переживайте. В следующих главах мы раскроем это понятие.

ГЛАВА 2

TypeScript с высоты птичьего полета

В нескольких следующих главах мы рассмотрим TypeScript, работу его компилятора (TSC), а также функции и паттерны, которые вы можете разрабатывать.

Компилятор

Ваше представление о программах во многом зависит от того, с какими языками программирования вы работали ранее. TypeScript отличается от большинства основных языков.

Программы — это файлы, содержащие прописанный вами текст. Специальная программа — компилятор — считывает и преобразует ваш текст в абстрактное синтаксическое дерево (АСД). Оно представляет собой структуру данных, игнорирующую пустые области, комментарии и ваше ценное мнение о пробелах или табуляции. Затем компилятор преобразует АСД в низкоуровневую форму — байт-код, который можно запустить в среде выполнения и получить результат. Итак, когда вы запускаете программу, фактически вы просите среду выполнения считать байт-код, сгенерированный компилятором на основе АСД, полученного из исходного кода. Детали этого процесса могут отличаться, но для большинства языков он выглядит так:

1. Программа преобразуется в АСД.
2. АСД компилируется в байт-код.
3. Байт-код считывается средой выполнения.

Особенность TypeScript в том, что вместо компиляции прямо в байт-код он компилирует в код JavaScript. Затем вы просто запускаете его в браузере,

с NodeJS или вручную, с помощью бумаги и ручки (вариант для тех, кто читает книгу во время восстания машин).

«Причем здесь безопасность кода?» — спросите вы.

Отличный вопрос. Я пропустил важный этап: после создания АСД компилятор проверяет типы.

МОДУЛЬ ПРОВЕРКИ ТИПОВ

Специальная программа, определяющая типобезопасность кода.

В проверке типов заключена магия TypeScript. С ее помощью он убеждается, что программа работает так, как вы ожидаете, и что приятная бариста из кафе напротив перезвонит вам (на самом деле она сейчас просто занята).

Итак, если мы добавим проверку типов и преобразование в JavaScript, то процесс компиляции TypeScript будет выглядеть примерно так (рис. 2.1).

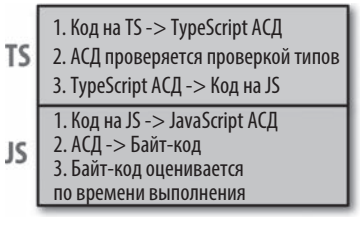

Рис. 2.1. Компиляция и запуск TypeScript

Шаги 1–3 производятся компилятором, а шаги 4–6 — средой выполнения JavaScript, находящейся в вашем браузере, или NodeJS, или любым другим JavaScript-движком.

JavaScript-компиляторы и среды выполнения, как правило, представляют собой единую программу, называемую движком. Будучи программистом, с ним вы и будете взаимодействовать. Так работают V8 (движок, лежащий в основе NodeJS, Chrome и Opera), SpiderMonkey (Firefox), JSCore (Safari) и Chakra (Edge). Именно поэтому JavaScript и называют *интерпретируемым* языком.

В течение всего процесса шаги 1–2 используют типы программы, а шаг 3 уже этого не делает. Стоит еще раз повториться: когда TSC компилирует код в JavaScript, он не будет смотреть на типы. Это означает, что типы никогда не смогут повлиять на сгенерированный вывод и будут использованы только для проверки типов. Эта особенность позволяет безопасно с ними экспериментировать — обновлять и улучшать их без риска сломать приложение.

Система типов

В современных языках реализованы различные системы типов.

> **СИСТЕМА ТИПОВ**
>
> Набор правил, используемых модулем проверки типов для присвоения типов программе.

Главным образом системы типов делятся на два вида: в одних вы должны сообщать компилятору тип каждого элемента посредством явного синтаксиса, другие выводят типы автоматически. Оба вида имеют как плюсы, так и минусы[1].

TypeScript создан на стыке этих двух видов: вы можете явно аннотировать типы либо позволить TypeScript делать их вывод за вас.

Для явного объявления типов используются аннотации, которые сообщают TypeScript, что такое-то значение имеет такой-то тип. Давайте взглянем на несколько примеров (комментарии в соответствующих строках указывают типы, выведенные TypeScript):

```
let a: number = 1                    // a является number
let b: string = 'hello'              // b является string
let c: boolean[] = [true, false]     // c является массивом booleans
```

[1] JavaScript, Python и Ruby выводят типы в среде выполнения. Haskell и OCaml делают вывод типов и проверяют недостающие типы в процессе компиляции. Scala и TypeScript иногда требуют явного указания типов, вывод же и проверку остальных они производят при компиляции. Java и C нуждаются в явных аннотациях практически для всего, что проверяют в среде выполнения.

Если вы хотите, чтобы TypeScript вывел за вас типы, то просто не прописывайте их:

```
let a = 1                          // a является number
let b = 'hello'                    // b является string
let c = [true, false]              // c является массивом booleans
```

Вы сразу убедитесь, насколько хорошо он справляется с этой задачей. Убрав аннотации, вы увидите, что типы остались прежними. На протяжении всей книги мы будем использовать аннотирование только по необходимости и позволим TypeScript демонстрировать свои волшебные способности.

В большинстве случаев лучше позволять TypeScript выводить типы по мере его возможностей и снижать тем самым объем явно аннотированного кода до минимума.

TypeScript vs JavaScript

Давайте углубимся в систему типов TypeScript и произведем сравнение с ее аналогом в JavaScript (табл. 2.1). Правильное понимание разницы является ключом для построения образной модели функционирования TypeScript.

Таблица 2.1. Сравнение систем типов JavaScript и TypeScript

Система типов	JavaScript	TypeScript
Как связываются типы	Динамически	Статически
Конвертируются ли типы автоматически	Да	Нет (в основном)
Когда проверяются типы	Во время выполнения	Во время компиляции
Когда вскрываются ошибки	Во время выполнения (в основном)	Во время компиляции (в основном)

Как связываются типы

Динамическое связывание типов подразумевает, что JavaScript знакомится с типами в программе только после ее запуска.

TypeScript является языком *с постепенной типизацией* — он работает лучше, если знает все типы программы во время компиляции. Но даже