Краткое содержание

Оглавление

Отзывы

«“Эффективный TypeScript” рассматривает наиболее распространенные проблемы, с которыми мы сталкиваемся при работе с TypeScript, и дает практические, ориентированные на результаты советы. Книга будет полезна и опытным, и начинающим разработчикам».

Райан Кавано, ведущий инженер по TypeScript в Microsoft

«“Эффективный TypeScript” содержит практические рецепты и должна быть под рукой у каждого профессионального разработчика. Даже если вы думаете, что знаете TypeScript, купите эту книгу — не пожалеете».

Яков Файн, Java-чемпион

«TypeScript захватывает мир разработки... Глубокое понимание TypeScript поможет разработчикам проявить себя, используя мощные функции языка».

Джейсон Киллиан, соучредитель TypeScript NYC
и бывший специалист по обслуживанию TSLint

«Эта книга не только о том, что может делать TypeScript, но и о том, почему полезна отдельно взятая функция и где применять шаблоны для достижения наибольшего эффекта. Книга фокусируется на практических советах, которые будут полезны в повседневной работе. При этом в ней достаточно теории, чтобы читатель на глубинном уровне понял, как все работает. Я считаю себя опытным пользователем TypeScript, но узнал много нового из этой книги».

Джесси Халлетт, старший инженер-программист, Originate, Inc.

Посвящается Алекс.
Ты — мой тип.

Введение

Весной 2016 года я навестил своего бывшего коллегу Эвана Мартина, работавшего в офисе Google в Сан-Франциско, и поинтересовался, чем он сейчас занимается. Этот вопрос я задавал ему неоднократно в разные годы и постоянно получал новые и весьма интересные ответы. Он рассказывал об инструментах построения C++, аудиодрайверах Linux, плагинах для Emacs и онлайн-кроссвордах. На этот раз Эван занялся TypeScript и Visual Studio Code.

Он меня удивил. Я слышал о TypeScript, но знал, что это разработка Microsoft, ошибочно думая, что она работает с .NET. Казалось шуткой, что Эван, который всю жизнь пользовался Linux, вдруг оценил Microsoft.

Затем он продемонстрировал мне VS Code и TypeScript в деле: все работало очень быстро, а интеллект кода легко выстраивал модель системы типов. Я много лет аннотировал типы в комментариях JSDoc для Closure Compiler и воспринял TS как реально работающий типизированный JavaScript. Неужели Microsoft разработала кроссплатформенный текстовый редактор на платформе Chromium? Возможно, этот язык с его набором инструментов действительно достоин изучения.

Я не так давно присоединился к Sidewalk Labs и приступил к написанию первого JavaScript-проекта. Основа кода все еще была достаточно мала, и мы с Эваном смогли конвертировать его в TypeScript всего за несколько дней.

С тех пор TypeScript меня зацепил. Он представляет собой не просто систему типов, а целый набор служб языка, удобных в использовании. Поэтому он не только повышает безопасность разработки в JavaScript, но и делает работу увлекательнее.

Кому адресована эта книга

Обычно книги, в названии которых есть слово «эффективный», рассматриваются в качестве второй основной книги по теме. Поэтому «Эффективный TypeScript» окажется максимально полезен тем, кто уже имеет опыт работы с JavaScript и TypeScript. Цель этой книги — не обучать читателей пользоваться инструментами, а помочь им повысить свой профессиональный уровень. Прочитав ее, вы сформируете лучшее представление о работе компонентов TypeScript, сможете избежать многих ловушек и ошибок и развить свои навыки. В то время как справочное руководство покажет пять разных путей применения языка для реализации одной задачи, «эффективная» книга объяснит, какой из этих путей лучше и почему.

В течение последних лет TypeScript развивался очень быстро, но я надеюсь, что сейчас он достаточно стабилен и моя книга еще долго будет актуальной. Все основное внимание в ней сконцентрировано на самом языке, а не на различных фреймворках и прочих инструментах. Вы не встретите здесь примеров использования React или Angular, равно как и пояснений возможного конфигурирования TypeScript для работы с webpack, Babel или rollup, зато обнаружите много универсальных советов.

Почему я написал эту книгу

Когда я начинал работать в Google, мне вручили экземпляр третьего издания «Эффективный C++», и книга показалась мне своеобразной. Ее нельзя было назвать доступной для начинающих или полноценным руководством по языку. Вместо пояснения назначения тех или иных инструментов C++ она обучала эффективно использовать их и включала множество коротких специфичных правил с примерами.

Эффект от ее чтения вместе с ежедневным использованием языка оказался ощутим. Я уже работал с C++, но только после знакомства с книгой стал уверенно пользоваться его компонентами. Позднее у меня были похожие опыты с чтением «Эффективный Java» и «Эффективный JavaScript».

Если вам комфортно работать с несколькими языками, разберитесь, чем именно они отличаются, и примите вызов от TypeScript. В процессе написания книги я сам узнал много нового, чего искренне желаю и вам.

Структура книги

Книга представляет собой сборник кратких эссе (правил). Правила объединены в тематические разделы (главы), к которым можно обращаться автономно в зависимости от интересующего вопроса.

Заголовок каждого правила содержит совет, поэтому ознакомьтесь с оглавлением. Если, например, вы пишете документацию и сомневаетесь, надо ли писать информацию типов, обратитесь к оглавлению и правилу 30 («Не повторяйте информацию типа в документации»).

Практически все выводы в книге продемонстрированы на примерах кода. Думаю, вы, как и я, склонны читать технические книги, глядя в примеры и лишь вскользь просматривая текстовую часть. Конечно, я надеюсь, что вы внимательно прочитаете объяснения, но основные моменты я отразил в примерах.

Прочитав каждый совет, вы сможете понять, как именно и почему он поможет вам использовать TypeScript более эффективно. Вы также поймете, если он окажется непригодным в каком-то случае. Мне запомнился пример, приведенный Скоттом Майерсом, автором книги «Эффективный C++»: разработчики ПО для ракет могли пренебречь советом о предупреждении утечки ресурсов, потому что их программы уничтожались при попадании ракеты в цель. Мне неизвестно о существовании ракет с системой управления, написанной на JavaScript, но такое ПО есть на телескопе James Webb. Поэтому будьте осторожны.

Каждое правило заканчивается блоком «Следует запомнить». Бегло просмотрев его, вы сможете составить общее представление о материале и выделить главное. Но я настоятельно рекомендую читать правило полностью.

Условные обозначения в примерах кода

Большинство примеров кода относятся к TypeScript, кроме случаев, где контекст прямо указывает на пример с JSOB, GraphQL и т. д. Большая часть работы с TypeScript подразумевает взаимодействие с редактором и вывод, поэтому я использовал некоторые сокращения.

Большинство редакторов выделяют ошибки волнистым подчеркиванием. Чтобы увидеть полное сообщение об ошибке, нужно навести курсор на подчеркнутый текст. Для отображения ошибок в примерах кода

я делал пометки в виде волнистой линии в комментариях под строкой с ошибкой:

```
let str = 'not a number';
let num: number = str;
// ~~~ Тип 'string' не может быть назначен для типа 'number'.
```

Иногда я редактировал сообщение об ошибке для придания ему ясности и краткости, но никогда не удалял саму ошибку. Если вы скопируете любой из образцов кода в свой редактор, то получите такую же ошибку.

Для привлечения внимания к отсутствию ошибки я использовал такое обозначение: `// ok`:

```
let str = 'not a number';
let num: number = str as any; // ok
```

Наведя курсор на отдельный символ в редакторе, вы должны увидеть, к какому типу он принадлежит. Дополнительно я использовал комментарий, начинающийся со слова «тип».

```
let v = {str: 'hello', num: 42}; // тип { str: string; num: number; }
```

Тип указан для первого символа строки (в данном случае `v`) или для результата вызова функции:

```
'four score'.split(' '); // строковый тип[]
```

Вы увидите такие же типы в своем редакторе. В случае с вызовом функции может потребоваться назначить временную переменную для отображения типа. В некоторых случаях я буду использовать фиктивные значения для отображения типа переменной в отдельной ветви кода:

```
function foo(x: string|string[]) {
  if (Array.isArray(x)) {
    x; // строковый тип []
  } else {
    x; // строковый тип
  }
}
```

Для `x;` слеши добавлены лишь в целях указания типа в каждой части условного выражения. Вам не нужно включать подобные выражения в код. Если из контекста не следует иное, то примеры кода должны проверяться флагом `-strict`. Все примеры были получены в TypeScript 3.5.3.

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, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный жирный`

Обозначает команды или другой текст, который должен вводиться пользователем. Также 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/oreillymedia/Effective_TypeScript.

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 разрабатываете 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

Эта книга появилась благодаря усилиям многих людей. Спасибо Эвану Мартину за подробное знакомство с TypeScript. Доуи Осинге — за установку связи с O'Reilly и поддержку в проекте. Бретту Слаткину — за советы по структуре и подсказку, что среди моих знакомых есть автор «эффективной» книги. Скотту Мейерсу — за внедрение этого формата и за его полезный пост «Эффективные книги» в блоге. Моим научным редакторам: Рику Баттлайну, Райану Кавано, Борису Черному, Якову Файну, Джейсону Киллиану и Джесси Халлету. Спасибо Джэкобу Баскину и Кэт Буш за замечания по отдельным правилам, а также всем моим коллегам, посвятившим годы работе с TypeScript. Команде TypeScript NYC: Джейсону, Орте и Кириллу, а также всем спикерам. Многие правила появились благодаря обсуждениям на семинаре, а именно правило 37 (Джейсон Киллиан) и правило 51 (Стив Фолкнер). Было еще большое количество постов и обсуждений, многие из которых собраны в ветке r/typescript на Reddit. Отдельную благодарность хочу выразить разработчикам, предоставившим образцы кода, которые оказались очень полезны для понимания базовых особенностей TypeScript: Андерсу, Дэниэлу, Райану и всей команде TypeScript в Microsoft — спасибо за общение и отзывчивость в решении возникавших проблем. Даже в случае простых недоразумений было очень приятно сообщить баг и видеть, как сам Андерс Хейлсберг моментально его исправляет. В завершение хочу сказать спасибо Алекс за постоянную поддержку в течение всего проекта и понимание моей потребности в работе по утрам, вечерам и выходным.

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

Знакомство с TypeScript

Эта глава поможет составить общее впечатление о TypeScript, прежде чем мы перейдем к деталям. Что это такое и как его воспринимать? Каким образом он связан с JavaScript? Как он работает с нулевыми типами, функциями `any` и утиной типизацией?

TypeScript — необычный язык. Он не использует интерпретатор (как Ruby или Python) и не проводит компиляцию кода в низкоуровневый язык (как Java или C). Вместо этого он компилирует программу в другой высокоуровневый язык — JavaScript. Приложение запускает именно JavaScript, а не TypeScript. Поэтому их взаимосвязь весьма существенна, но может вызвать путаницу.

Система типов в TypeScript имеет свои особенности, которые будут подробно разобраны. В настоящей главе мы предупредим об основных сюрпризах.

ПРАВИЛО 1. TypeScript и JavaScript взаимосвязаны

Наверняка вы встречали выражение: «TypeScript — это надмножество JavaScript» или «TypeScript — это типизированное расширение JavaScript». Но какая на самом деле между ними взаимосвязь?

TypeScript действительно является надмножеством JavaScript в синтаксическом смысле. Любая программа JavaScript, не имеющая синтаксических ошибок, является и программой TypeScript. Модуль контроля типов в TypeScript отмечает некоторые проблемные места в коде, но все равно его обрабатывает и выдает в виде JavaScript. Эту часть отношений между ними мы ближе рассмотрим в правиле 3.

Файлы TypeScript имеют расширения .ts или .tsx вместо расширений .js и .jsx, характерных для JavaScript. Однако переименование файла main.js в main.ts не изменит код.

Это очень удобно, ведь можно просто продолжить работать с кодом, пользуясь дополнительными преимуществами TypeScript. Это бы не сработало, если бы вы решили переписать код на язык вроде Java. Плавный процесс переноса кода является одной из отличительных особенностей TypeScript (глава 8).

Все программы JS совместимы с TS, но не всегда наоборот. Существуют программы TS, не подходящие к JS. Это обусловлено тем, что TypeScript отличается дополнительным синтаксисом в спецификации типов (правило 53).

К примеру, вот рабочая программа TypeScript:

```
function greet(who: string) {
  console.log('Hello', who);
}
```

Если вы ее запустите через программу вроде Node, которая работает с JavaScript, то получите ошибку:

```
function greet(who: string) {
                   ^

SyntaxError: Unexpected token :
```

Содержимое `: string` является аннотацией типа, характерной для TypeScript. Использовав ее хоть раз, вы выходите за рамки JavaScript (рис. 1.1).

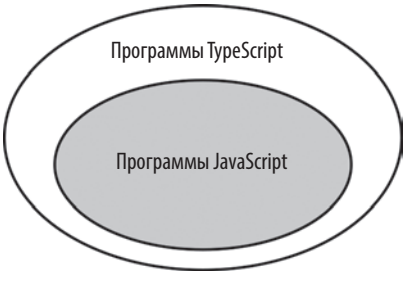

Рис. 1.1. Все программы JavaScript поддерживаются в TypeScript, но не наоборот (диаграмма Венна)

Это не говорит о том, что TS бывает вреден для JS-программ. Рассмотрим следующий код JS:

```
let city = 'new york city';
console.log(city.toUppercase());
```

Программа выдаст ошибку при запуске:

```
TypeError: city.toUppercase is not a function
```

В ней нет аннотаций типов, но модуль проверки типов TS все равно обнаруживает проблему:

```
let city = 'new york city';
console.log(city.toUppercase());
              // ~~~~~~~~~~~ Свойство 'toUppercase' не существует в типе
              //             'string'. Может, вы имели в виду 'toUpperCase'?
```

В этой ситуации не пришлось сообщать TypeScript, что тип `city` был `string`. Он вывел его, опираясь на исходное значение. Вывод типов является основным занятием TS (глава 3).

Одна из целей системы типов в TypeScript — это поиск кода, который выдаст исключение при выполнении, без его запуска. Поэтому TS называют «статичной» системой типов. Тем не менее модуль проверки типов не всегда способен обнаружить код, приводящий к выводу исключений.

Даже если код не выдает исключение, возможно, он все еще делает не то, что вам нужно. TypeScript старается уловить подобные проблемы.

Например, следующий код JavaScript:

```
const states = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska', capital: 'Juneau'},
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capitol);
}
```

выведет:

```
undefined
undefined
undefined
```

Упс! Что здесь пошло не так? Это рабочая JavaScript-программа (а значит, и TypeScript), и она запустилась без сообщений об ошибках, но выполнила совсем не то, что от нее ожидали. Даже без добавления аннотаций типов модуль их проверки способен обнаружить место ошибки (а также предложить ее решение):

```
for (const state of states) {
  console.log(state.capitol);
                // ~~~~~~~ Свойство 'capitol' не существует в типе
                //         '{ name: string; capital: string; }'.
                //         Вы имели в виду 'capital'?
}
```

Мало того что TypeScript способен отлавливать ошибки даже при отсутствии аннотирования типов, он делает это еще эффективнее, если вы аннотирование проведете. Все потому, что прописанные типы конкретно сообщают TS о ваших *намерениях*, что позволяет ему легче обнаруживать места кода, им не соответствующие. Например, обратная ситуация:

```
const states = [
  {name: 'Alabama', capitol: 'Montgomery'},
  {name: 'Alaska', capitol: 'Juneau'},
  {name: 'Arizona', capitol: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capital);
                // ~~~~~~~ Свойство 'capital' не существует в типе
                //        '{ name: string; capitol: string; }'.
                //        Вы имели в виду 'capitol'?
}
```

Та подсказка, которая оказалась столь полезна в первом случае, теперь совершенно неверна. Проблема в том, что вы указали одно свойство в двух разных вариантах, и TypeScript не знает, какой из них верный. Он может угадывать, но не всегда правильно. Решением станет прояснение ваших намерений однозначным объявлением типов `states`:

```
interface State {
  name: string;
  capital: string;
}
const states: State[] = [
  {name: 'Alabama', capitol: 'Montgomery'},
                  // ~~~~~~~~~~~~~~~~~~~~
```

```
  {name: 'Alaska', capitol: 'Juneau'},
                  // ~~~~~~~~~~~~~~~~~~
  {name: 'Arizona', capitol: 'Phoenix'},
                  // ~~~~~~~~~~~~~~~~~~~Объектный литерал может
                  //         определять только известные свойства,
                  // но `capitol` не существует в типе `State`.
                  // Возможно, вы хотели написать `capital`?
  // ...
];
for (const state of states) {
  console.log(state.capital);
}
```

Теперь ошибка соответствует проблеме, а предложенное решение корректно. Прописывая свое намерение, вы помогаете TypeScript обнаружить и другие потенциальные проблемы. Например, если написать `capitol` только в одной части массива, то в первом примере ошибки бы не возникло. Но уже при аннотированном типе она обнаружится:

```
const states: State[] = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska', capitol: 'Juneau'},
                 // ~~~~~~~~~~~~~~~~~ Вы имели в виду 'capital'?
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
```

Таким образом, мы можем добавить на диаграмму Венна еще одну группу: программы TypeScript, прошедшие проверку типов (рис. 1.2).

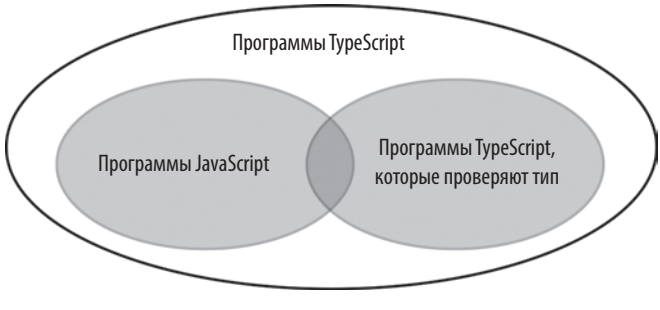

Рис. 1.2. Все программы JavaScript являются TypeScript-программами, но лишь некоторые программы JavaScript (и TypeScript) проходят проверку типов

Да, TypeScript — это надмножество JavaScript, и конечно, вы захотите, чтобы код прошел все проверки типов.

Система типов TypeScript *моделирует* процесс выполнения JavaScript. Вы удивитесь такому моделированию, если привыкли к языку с более строгими проверками выполнения. Например:

```
const x = 2 + '3'; // ok, строковый тип
const y = '2' + 3; // ok, строковый тип
```

Оба этих выражения пройдут проверку типов, даже несмотря на то что они сомнительны и вызвали бы ошибки выполнения во многих других языках. Но в данном случае мы просто определяем выполнение JavaScript и на выводе получим результат обоих выражений в виде строки `"23"`.

Тем не менее TypeScript имеет границы дозволенного. Модуль проверки типов указывает на проблемы в приведенных выражениях, даже несмотря на то что они не выдают исключения во время выполнения:

```
const a = null + 7; // вычисляется как 7 в JS
       // ~~~~~~~~ Оператор '+' не может быть применен к типам ...
const b = [] + 12; // вычисляется как '12' в JS
       // ~~~~~~~ Оператор '+' не может быть применен к типам ...
alert('Hello', 'TypeScript'); // alerts "Hello".
            // ~~~~~~~~~~~~ Ожидается аргумент 0-1, но получен 2.
```

Основная задача системы типов в TypeScript заключается в моделировании поведения JavaScript при выполнении. Причем TypeScript склонен воспринимать непонятные элементы как ошибки, а не как замысел разработчика, то есть идет дальше простого моделирования выполнения. В случае с `capitol` программа не выдавала ошибку (возвращая `undefined`), но модуль проверки типов все же ее отмечал.

Как же TypeScript принимает решения? Доверьтесь команде его разработчиков. Вам нравится складывать `null` и `7` или `[]` и `12`? А может быть, вызывать функции с лишними аргументами? TypeScript это вряд ли поддержит.

Может ли быть, что программа, прошедшая проверку типов, все равно выдаст ошибку при выполнении? Да, и вот пример:

```
const names = ['Alice', 'Bob'];
console.log(names[2].toUpperCase());
```

При запуске она выдаст:

```
TypeError: Cannot read property 'toUpperCase' of undefined
```

Доступ к массиву оказался за пределами, ожидаемыми TypeScript. Результатом стало исключение.

Ошибки также ускользают от обнаружения, когда вы используете тип `any`, который мы подробно рассмотрим в правиле 5 и еще более углубленно изучим в главе 5.

Основная причина исключений кроется в том, что понятный TypeScript тип значения расходится с реальным. Система типов TypeScript не гарантирует точности своих статичных типов. Если таковая точность для вас важна, обратите внимание на другие языки вроде Reason или Elm, но за их повышенный уровень безопасного выполнения придется заплатить повышенной сложностью перехода с JavaScript.

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ TypeScript — это надмножество JavaScript. Проще говоря, все программы JavaScript по умолчанию являются программами TypeScript. Однако в TS имеется некоторый специфичный синтаксис, не позволяющий некоторым его программам быть совместимыми с JavaScript.
- ✓ TypeScript имеет систему типов, моделирующую выполнение JavaScript, а также обнаруживающую код, который при выполнении выдаст исключение. Гарантий полного представления исключений нет: возможны случаи, когда код проходит проверку типов, но при выполнении выдает ошибку.
- ✓ Несмотря на то что TypeScript моделирует поведение JavaScript, существуют конструкции, которые JS допускает, а TS нет. К их числу относятся вызовы функций с неверными числами или аргументами. Их использование, в большинстве случаев, — дело вкуса.