
Оглавление

ОТЗЫВЫ	10
ВВЕДЕНИЕ	12
От издательства.....	17
ГЛАВА 1. ЗНАКОМСТВО С TYPESCRIPT.....	18
Правило 1. TypeScript и JavaScript взаимосвязаны.....	18
Правило 2. Выбирайте нужные опции в TypeScript	25
Правило 3. Генерация кода не зависит от типов	28
Правило 4. Привыкайте к структурной типизации.....	35
Правило 5. Ограничьте применение типов any	40
ГЛАВА 2. СИСТЕМА ТИПОВ В TYPESCRIPT	44
Правило 6. Используйте редактор для работы с системой типов	44
Правило 7. Воспринимайте типы как наборы значений.....	48
Правило 8. Правильно выражайте отношение символа к пространству типов или пространству значений	56
Правило 9. Объявление типа лучше его утверждения	62
Правило 10. Избегайте оберточных типов (String, Number, Boolean, Symbol, BigInt).....	66
Правило 11. Проверяйте пределы исключительных свойств типа	69
Правило 12. По возможности применяйте типы ко всему выражению функции	73

Правило 13. Знайте разницу между <code>type</code> и <code>interface</code>	76
Правило 14. Операции типов и обобщения сокращают повторы.....	81
Правило 15. Используйте сигнатуры индексов для динамических данных.....	90
Правило 16. В качестве сигнатур индексов используйте массивы, кортежи и <code>ArrayLike</code> , но не <code>number</code>	94
Правило 17. Используйте <code>readonly</code> против ошибок, связанных с изменемостью	98
Правило 18. Используйте отображенные типы для синхронизации значений.....	105
ГЛАВА 3. ВЫВОД ТИПОВ	109
Правило 19. Не засоряйте код ненужными аннотациями типов	109
Правило 20. Для разных типов — разные переменные.....	117
Правило 21. Контролируйте расширение типов.....	119
Правило 22. Старайтесь сужать типы.....	123
Правило 23. Создавайте объекты целиком.....	127
Правило 24. Применяйте псевдонимы согласованно	130
Правило 25. Для асинхронного кода используйте функции <code>async</code> вместо обратных вызовов.....	134
Правило 26. Используйте контекст при выводе типов	139
Правило 27. Используйте функциональные конструкции и библиотеки для содействия движению типов	144
ГЛАВА 4. ПРОЕКТИРОВАНИЕ ТИПОВ	149
Правило 28. Используйте типы, имеющие допустимые состояния.....	149
Правило 29. Будьте либеральны в том, что берете, но консервативны в том, что даете.....	155
Правило 30. Не повторяйте информацию типа в документации.....	159
Правило 31. Смещайте нулевые значения на периферию типов.....	161
Правило 32. Предпочитайте объединения интерфейсов интерфейсам объединений.....	165

Правило 33. Используйте более точные альтернативы типов <code>string</code>	169
Правило 34. Лучше сделать тип незавершенным, чем ошибочным	174
Правило 35. Генерируйте типы на основе API и спецификаций, а не данных.....	178
Правило 36. Именуйте типы согласно области их применения.....	183
Правило 37. Рассмотрите использование маркировок для номинального типизирования	186
ГЛАВА 5. ЭФФЕКТИВНОЕ ПРИМЕНЕНИЕ ANY	190
Правило 38. Используйте максимально узкий диапазон для типов <code>any</code>	190
Правило 39. Используйте более точные варианты <code>any</code>	193
Правило 40. Скрывайте небезопасные утверждения типов в грамотно типизированных функциях	195
Правило 41. Распознавайте изменяющиеся <code>any</code>	197
Правило 42. Используйте <code>unknown</code> вместо <code>any</code> для значений с неизвестным типом	201
Правило 43. Используйте типобезопасные подходы вместо обезьяньего патча.....	205
Правило 44. Отслеживайте зону охвата типов для сохранения типобезопасности	207
ГЛАВА 6. ДЕКЛАРАЦИИ ТИПОВ И @TYPES.....	211
Правило 45. Размещайте TypeScript и <code>@types</code> в <code>devDependencies</code>	211
Правило 46. Проверяйте совместимость трех версий, задействованных в декларациях типов	213
Правило 47. Экспортируйте все типы, появляющиеся в публичных API.....	218
Правило 48. Используйте TSDoc для комментариев в API	219
Правило 49. Определяйте тип <code>this</code> в обратных вызовах	222
Правило 50. Лучше условные типы, чем перегруженные декларации	226
Правило 51. Зеркалируйте типы для разрыва зависимостей.....	229
Правило 52. Тестируйте типы с осторожностью.....	231

ГЛАВА 7. НАПИСАНИЕ И ЗАПУСК КОДА	237
Правило 53. Используйте возможности ECMAScript, а не TypeScript.....	237
Правило 54. Проводите итерацию по объектам.....	243
Правило 55. Иерархия DOM — это важно.....	246
Правило 56. Не полагайтесь на private при скрытии информации.....	251
Правило 57. Используйте карты кода для отладки.....	254
ГЛАВА 8. ПЕРЕНОС ДАННЫХ В TYPESCRIPT	259
Правило 58. Пишите современный JavaScript.....	260
Правило 59. Используйте // @ts-check и JSDoc для экспериментов в TypeScript.....	269
Правило 60. Используйте allowJs для совмещения TypeScript и JavaScript.....	274
Правило 61. Конвертируйте модуль за модулем, восходя по графу зависимостей.....	276
Правило 62. Не считайте миграцию завершенной, пока не включите noImplicitAny.....	281
ОБ АВТОРЕ	284
ОБ ОБЛОЖКЕ	285

ПРАВИЛО 15. Используйте сигнатуры индексов для динамических данных

JavaScript обладает удобным синтаксисом для создания объектов:

```
const rocket = {  
  name: 'Falcon 9',  
  variant: 'Block 5',  
  thrust: '7,607 kN',  
};
```

Объекты в JavaScript отображают строковые ключи в значения любого типа. TypeScript позволяет производить подобное гибкое отображение при помощи назначения для типа *сигнатуры индекса*:

```
type Rocket = {[property: string]: string};  
const rocket: Rocket = {  
  name: 'Falcon 9',  
  variant: 'v1.0',  
  thrust: '4,940 kN',  
}; // ok
```

`{[property: string]: string}` является сигнатурой индекса и определяет три параметра:

Имя ключа

Необходимо только для документации и не применяется для модуля проверки типов.

Тип ключа

Должен быть некоторой комбинацией `string`, `number` или `symbol`, но в основном вам понадобится использовать `string` (правило 16).

Тип значений

Может быть чем угодно.

Поскольку сигнатура индекса будет проходить проверку типов, нужно учитывать некоторые минусы:

- Она позволяет использовать любые ключи, включая неверные. Даже если вы напишете `Name` вместо `name`, тип `Rocket` будет по-прежнему считаться рабочим.

- Она не требует наличия каких-либо специфичных ключей. Даже при {} тип Rocket будет действующим.
- Она не может иметь различные типы для различных ключей. Thrust, вероятно, должен быть number, но не string.
- Языковые службы TypeScript не помогут вам прописывать подобные типы. В то время как вы печатаете name:, вам не будет предложена автоподстановка, так как ключ может оказаться любым.

Коротко говоря, сигнатуры индексов не отличаются точностью. Почти всегда можно найти лучшую альтернативу. В этом случае Rocket однозначно должен быть прописан как interface:

```
interface Rocket {
  name: string;
  variant: string;
  thrust_kN: number;
}
const falconHeavy: Rocket = {
  name: 'Falcon Heavy',
  variant: 'v1',
  thrust_kN: 15_200
};
```

thrust_kN является number и TypeScript проведет проверку наличия всех необходимых полей. При этом будут доступны все полезные языковые сервисы TypeScript: автоподстановка, переход к определению и переименование.

Так зачем же использовать сигнатуры индексов? Для динамических данных. Они могут применяться в CSV-файлах, где имеется строковый заголовок, и нужно представить строки данных в виде объектов, отображающих имена столбцов в значения:

```
function parseCSV(input: string): {[columnName: string]: string}[] {
  const lines = input.split('\n');
  const [header, ...rows] = lines;
  return rows.map(rowStr => {
    const row: {[columnName: string]: string} = {};
    rowStr.split(',').forEach((cell, i) => {
      row[header[i]] = cell;
    });
    return row;
  });
}
```

Если вы не знаете наперед, какие у столбцов будут имена, а пользователи будут лучше понимать значения колонок в конкретном контексте, сигнатура индекса будет уместна:

```
interface ProductRow {
  productId: string;
  name: string;
  price: string;
}
```

```
declare let csvData: string;
const products = parseCSV(csvData) as unknown as ProductRow[];
```

Чтобы столбцы точно оправдали ваши ожидания, можете добавить `undefined` к значению типа.

```
function safeParseCSV(
  input: string
): {[columnName: string]: string | undefined}[] {
  return parseCSV(input);
}
```

Но тогда каждое обращение потребует проверки:

```
const rows = parseCSV(csvData);
const prices: {[product: string]: number} = {};
for (const row of rows) {
  prices[row.productId] = Number(row.price);
}
const safeRows = safeParseCSV(csvData);
for (const row of safeRows) {
  prices[row.productId] = Number(row.price);
  // ~~~~~ Тип 'undefined' не может быть использован
  // в качестве типа индекса.
}
```

Конечно, это усложнит работу с типом. Поэтому выбор `undefined` остается за вами.

Если тип имеет ограниченный набор доступных полей, то не следует моделировать его с помощью сигнатуры индекса. Например, если известно, что данные будут иметь ключи вроде A, B, C, D, но при этом вы не знаете, сколько именно их будет, то смоделируйте тип с помощью либо опциональных полей, либо объединения:

```
interface Row1 { [column: string]: number } // слишком обширно
interface Row2 { a: number; b?: number; c?: number; d?: number } // лучше
type Row3 =
  | { a: number; }
  | { a: number; b: number; }
  | { a: number; b: number; c: number; }
  | { a: number; b: number; c: number; d: number };
```

Последняя форма является наиболее точной, но менее удобной в работе.

Если сложность в использовании сигнатуры индекса кроется в чрезмерной обширности `string`, то можно обратиться к ряду альтернатив.

Одна из них — использование `Record`. Это обобщенный тип, дающий типу ключа гибкость. В частности, он позволяет передавать подмножества `string`:

```
type Vec3D = Record<'x' | 'y' | 'z', number>;
// тип Vec3D = {
//   x: number;
//   y: number;
//   z: number;
// }
```

Еще один способ — это использование отображенного типа, с помощью которого можно применять разные типы для разных ключей.

```
type Vec3D = {[k in 'x' | 'y' | 'z']: number};
// так же, как и выше
type ABC = {[k in 'a' | 'b' | 'c']: k extends 'b' ? string : number};
// тип ABC = {
//   a: number;
//   b: string;
//   c: number;
// }
```

СЛЕДУЕТ ЗАПОМНИТЬ

- ✓ Используйте сигнатуры индексов, когда свойства объекта не могут быть известны до момента выполнения программы. Например, если вы загружаете их из CSV-файла.
- ✓ Рассмотрите вариант добавления `undefined` к типу значения сигнатуры индекса для более безопасного обращения.
- ✓ По возможности старайтесь использовать не сигнатуры индексов, а более точные типы: `interface`, `Record` или отображенные типы.