Оглавление

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: {[produt: 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.