Что нового в TypeScript 5.0: Декораторы, тип Const, улучшение Enums, скорость и многое другое!

Описание

TypeScript 5.0 был официально выпущен 16 марта 2023 года и теперь доступен для использования всем желающим. В этом выпуске представлено множество новых возможностей, цель которых – сделать TypeScript меньше, проще и быстрее. В новом выпуске модернизированы декораторы для настройки классов, что позволяет настраивать классы и их члены многократно используемым способом. Разработчики теперь могут добавлять модификатор const к объявлению параметра типа, что позволяет использовать const-подобные выводы по умолчанию. Кроме того, в новом выпуске все перечисления стали объединенными перечислениями, что упрощает структуру кода и ускоряет работу с TypeScript. В этой статье вы изучите изменения, внесенные в TypeScript 5.0, и подробно рассмотрите его новые функции и возможности.

Начало работы с TypeScript 5.0

TypeScript – это официальный компилятор, который вы можете установить в свой проект с помощью npm. Если вы хотите начать использовать TypeScript 5.0 в своем проекте, вы можете выполнить следующую команду в каталоге проекта:

```
npm install -D typescript
```

Это установит компилятор в каталог node_modules, который теперь можно запустить с помощью команды npx tsc. Вы также можете найти инструкции по

использованию новой версии TypeScript в Visual Studio Code в этой документации.

Что нового в TypeScript 5.0?

В этой статье мы рассмотрим 5 основных обновлений, внесенных в TypeScript. К ним относятся:

- Модернизированные декораторы
- Представление параметров типа const
- Улучшения для перечислений
- Улучшения производительности TypeScript 5.0
- Разрешение Bundler для лучшего разрешения модулей

Модернизированные декораторы

Декораторы уже некоторое время существуют в TypeScript под экспериментальным флагом, но в новом выпуске они приведены в соответствие с предложением ECMAScript, которое сейчас находится на стадии 3, что означает, что оно находится на стадии добавления в TypeScript. Декораторы – это способ настройки поведения классов и их членов с возможностью многократного использования. Например, если у вас есть класс, у которого есть два метода, greet и getAge:

```
class Person {
    name: string;
    age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hello, my name is ${this.name}.`);
    }

    getAge() {
        console.log(`I am ${this.age} years old.`);
    }
}

const p = new Person('Ron', 30);
p.greet();

p.getAge();
```

В реальном мире этот класс должен иметь более сложные методы, которые

обрабатывают некоторую асинхронную логику, имеют побочные эффекты и т.д., где вы захотите бросить несколько вызовов **console.log**, чтобы помочь отладить методы.

```
class Person {
    name: string;
    age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log('LOG: Method Execution Starts.');
        console.log(`Hello, my name is ${this.name}.`);
        console.log('LOG: Method Execution Ends.');
    }

    getAge() {
        console.log('LOG: Method Execution Starts.');
        console.log(`I am ${this.age} years old.`);
        console.log('Method Execution Ends.');
    }
}

const p = new Person('Ron', 30);
p.greet();

p.getAge();
```

Это часто встречающийся паттерн, и было бы удобно иметь решение, применимое к каждому методу. Здесь на помощь приходят декораторы. Мы можем определить функцию с именем **debugMethod**, которая будет выглядеть следующим образом:

```
function debugMethod(originalMethod: any, context: any) {
    function replacementMethod(this: any, ...args: any[]) {
        console.log('Method Execution Starts.');
        const result = originalMethod.call(this, ...args);
        console.log('Method Execution Ends.');
        return result;
    }
    return replacementMethod;

}
```

В приведенном выше коде debugMethod принимает исходный метод (originalMethod) и возвращает функцию, которая делает следующее:

1. Заносит в журнал сообщение “Выполнение метода начинается.”.
2. Передает исходный метод и все его аргументы (включая this).

3. Выдает сообщение “Выполнение метода завершено”.
4. Возвращает то, что вернул исходный метод.

Используя декораторы, вы можете применить debugMethod к своим методам, как показано в приведенном ниже коде:

```
class Person {
    name: string;
    age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
    @debugMethod
    greet() {
        console.log(`Hello, my name is ${this.name}.`);
    }
    @debugMethod
    getAge() {
        console.log(`I am ${this.age} years old.`);
    }
}
const p = new Person('Ron', 30);
p.greet();

p.getAge();
```

В результате будет выведено следующее:

```
LOG: Entering method.
Hello, my name is Ron.
LOG: Exiting method.
LOG: Entering method.
I am 30 years old.

LOG: Exiting method.
```

При определении функции-декоратора (**debugMethod**) передается второй параметр под названием **context** (это объект контекста – содержит полезную информацию о том, как был объявлен декорированный метод, а также имя метода). Вы можете обновить свой **debugMethod**, чтобы получить имя метода из объекта контекста:

```
function debugMethod(
    originalMethod: any,
    context: ClassMethodDecoratorContext
) {
    const methodName = String(context.name);
    function replacementMethod(this: any, ...args: any[]) {
        console.log(`'${methodName}' Execution Starts.`);
        const result = originalMethod.call(this, ...args);
```

```
        console.log(`'${methodName}' Execution Ends.`);
        return result;
    }
    return replacementMethod;

}
```

Когда вы запустите свой код, вывод теперь будет содержать имя каждого метода, украшенного декоратором **debugMethod**:

```
'greet' Execution Starts.
Hello, my name is Ron.
'greet' Execution Ends.
'getAge' Execution Starts.
I am 30 years old.

'getAge' Execution Ends.
```

С помощью декораторов можно сделать гораздо больше. Не стесняйтесь проверять исходный запрос на вытягивание для получения дополнительной информации о том, как использовать декораторы в TypeScript.

Внедрение параметров типа const

Это еще один большой релиз, который дает вам новый инструмент с дженериками, чтобы улучшить вывод, который вы получаете при вызове функций. По умолчанию, когда вы объявляете значения с const, TypeScript определяет тип, а не его буквальные значения:

```
// Inferred type: string[]

const names = ['John', 'Jake', 'Jack'];
```

До сих пор, чтобы добиться желаемого вывода, вам приходилось использовать утверждение const, добавляя “as const”:

```
// Inferred type: readonly ["John", "Jake", "Jack"]

const names = ['John', 'Jake', 'Jack'] as const;
```

Когда вы вызываете функции, все происходит аналогично. В приведенном ниже коде предполагаемый тип стран – **string[]**:

```
type HasCountries = { countries: readonly string[] };
function getCountriesExactly(arg: T): T['countries'] {
    return arg.countries;
}

// Inferred type: string[]
```

```
const countries = getCountriesExactly({ countries: ['USA', 'Canada', 'India']
```

Вы можете захотеть более конкретный тип, одним из способов исправления которого до сих пор было добавление утверждения **as const**:

```
// Inferred type: readonly ["USA", "Canada", "India"]

const names = getNamesExactly({ countries: ['USA', 'Canada', 'India'] } as con
```

Это может быть трудно запомнить и реализовать. Однако в TypeScript 5.0 появилась новая функция, позволяющая добавить модификатор const к объявлению параметра типа, который будет автоматически применять const-подобный вывод по умолчанию.

```
type HasCountries = { countries: readonly string[] };
function getNamesExactly(arg: T): T['countries'] {
    return arg.countries;
}

// Inferred type: readonly ["USA", "Canada", "India"]

const names = getNamesExactly({ countries: ['USA', 'Canada', 'India'] });
```

Использование параметров типа const позволяет разработчикам более четко выразить намерение в коде. Если переменная должна быть постоянной и никогда не изменяться, использование параметра типа const гарантирует, что она никогда не будет случайно изменена.

Улучшения для перечислений

Перечисления в TypeScript – это мощная конструкция, позволяющая разработчикам определять набор именованных констант. В TypeScript 5.0 были внесены улучшения в перечисления, чтобы сделать их еще более гибкими и полезными.

Например, если в функцию передается следующее перечисление:

```
enum Color {
    Red,
    Green,
    Blue,
}

function getColorName(colorLevel: Color) {
    return colorLevel;
}

console.log(getColorName(1));
```

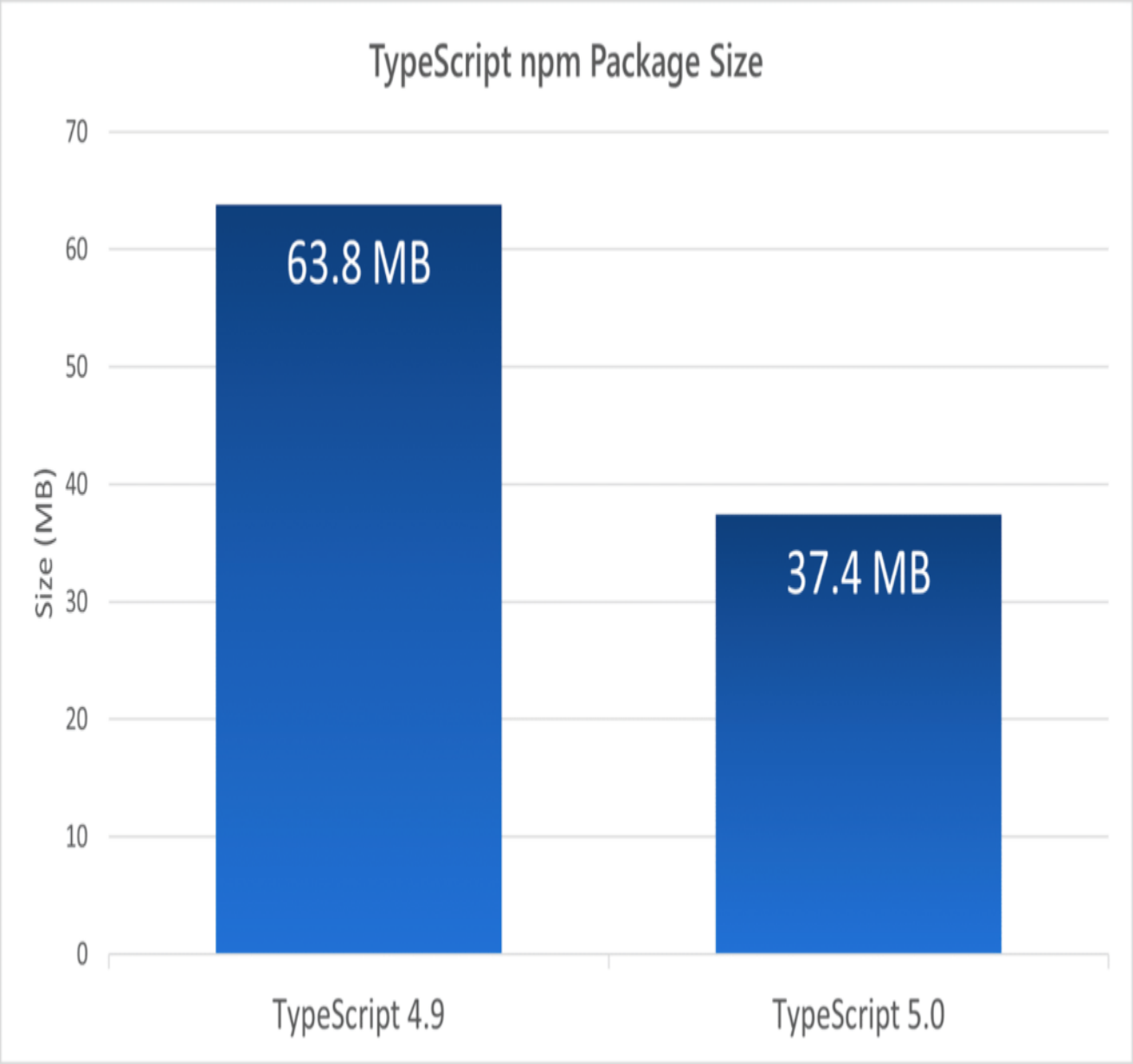
До появления TypeScript 5.0 можно было передать неверный номер уровня, и ошибка не возникала. Но после введения TypeScript 5.0 он будет немедленно выдавать ошибку. Кроме того, в новом выпуске все перечисления превращаются в объединения перечислений путем создания уникального типа для каждого вычисляемого члена. Это усовершенствование позволяет сузить все перечисления и ссылаться на их члены как на типы:

```
enum Color {
    Red,
    Purple,
    Orange,
    Green,
    Blue,
    Black,
    White,
}

type PrimaryColor = Color.Red | Color.Green | Color.Blue;

function isPrimaryColor(c: Color): c is PrimaryColor {
    return c === Color.Red || c === Color.Green || c === Color.Blue;
}

console.log(isPrimaryColor(Color.White)); // Outputs: false

console.log(isPrimaryColor(Color.Red)); // Outputs: true
```

Улучшения производительности TypeScript 5.0

TypeScript 5.0 включает множество значительных изменений в структуре кода, структурах данных и алгоритмических расширениях. Это позволило улучшить всю работу с TypeScript, от установки до выполнения, сделав ее более быстрой и эффективной. Например, разница между размером пакета TypeScript 5.0 и 4.9 весьма впечатляет. Недавно TypeScript был переведен из пространств имен в модули, что позволяет ему использовать современные инструменты сборки,

которые могут выполнять такие оптимизации, как поднятие области видимости. Кроме того, удаление некоторого устаревшего кода позволило сэкономить около 26,4 МБ от размера пакета TypeScript 4.9, составлявшего 63,8 МБ.



Вот еще несколько интересных выигрышей в скорости и размере между TypeScript 5.0 и 4.9:

Сценарий	**Время или размер относительно TS 4.9**
Время создания Material-ui	90%
Время запуска компилятора	89%

Время сборки Playwright	88%
Время самостоятельной сборки TypeScript Compiler	87%
Время сборки Outlook Web	82%
Время сборки VS Code	80%
Typescript npm Размер пакета	59%

Разрешение Bundler для лучшего разрешения модулей

Когда вы пишете оператор импорта в TypeScript, компилятор должен знать, к чему относится этот импорт. Он делает это с помощью разрешения модулей. Например, когда вы пишете **import { a } from “moduleA”**, компилятору необходимо знать определение **a** в **moduleA**, чтобы проверить его использование. В TypeScript 4.7 для параметров —**module** и **moduleResolution** были добавлены две новые опции: **node16** и **nodenext**. Цель этих опций – более точно представить точные правила поиска модулей ECMAScript в Node.js. Однако этот режим имеет несколько ограничений, которые не соблюдаются другими инструментами.

Например, в модуле ECMAScript в Node.js любой относительный импорт должен включать расширение файла, чтобы он работал правильно:

```
import * as utils from "./utils"; // Wrong
```

```
import * as utils from "./utils.mjs"; // Correct
```

В TypeScript появилась новая стратегия под названием “moduleResolution bundler”. Эта стратегия может быть реализована путем добавления следующего кода в раздел “compilerOptions” вашего конфигурационного файла TypeScript:

```
{
    "compilerOptions": {
        "target": "esnext",
        "moduleResolution": "bundler"
    }

}
```

Эта новая стратегия подходит для тех, кто использует современные бандлеры, такие как Vite, esbuild, swc, Webpack, Parcel и другие, использующие гибридную стратегию поиска.

Износы

TypeScript 5.0 поставляется с некоторыми изменениями, включая требования к времени выполнения, изменения в lib.d.ts и изменения, нарушающие API.

- **Требования к времени выполнения:** TypeScript теперь ориентирован на ECMAScript 2018, и пакет устанавливает минимальное ожидание движка 12.20. Поэтому пользователи Node.js должны иметь минимальную версию 12.20 или более позднюю для использования TypeScript 5.0.
- **Изменения lib.d.ts:** В то, как генерируются типы для DOM, были внесены некоторые изменения, которые могут повлиять на существующий код. В частности, некоторые свойства были преобразованы из числовых в числовые литеральные типы, а свойства и методы для обработки событий вырезания, копирования и вставки были перенесены между интерфейсами.
- **Разрушающие изменения API:** Были удалены некоторые ненужные интерфейсы, а также внесены некоторые улучшения в корректность. TypeScript 5.0 также перешел на модули.

В TypeScript 5.0 устарели некоторые параметры и соответствующие им значения, включая target: ES3, out, noImplicitUseStrict, keyofStringsOnly, suppressExcessPropertyErrors, suppressImplicitAnyIndexErrors, noStrictGenericChecks, charset, importsNotUsedAsValues и preserveValueImports, а также prepend в ссылках проекта. Хотя эти конфигурации останутся действительными до TypeScript 5.5, будет выпущено предупреждение, чтобы предупредить пользователей, все еще использующих их.

Заключение

В этой статье вы узнали о некоторых основных возможностях и улучшениях TypeScript 5.0, таких как улучшения перечислений, разрешение бандлера и параметры типа const, а также улучшения скорости и размера.

Дата Создания
28.04.2023