A Quick Guide to TypeScript 5.0 Decorators

Expand Your TypeScript 5.0 Toolkit with Decorators

Zack · Follow

5 min read · Apr 12, 2023

Photo by vadim kaipov on Unsplash

The TypeScript 5.0 release, officially supports the Phase 3 decorator proposal. The proposal has four stages, which means it stabilizes quickly without major changes to the API. In this article, we will explore based on these stable APIs.

What are Decorators

Decorators are a powerful feature in TypeScript that allows developers to modify or extend the behavior of classes, methods, accessors, and properties. They offer an elegant way to add functionality or modify the behavior of existing constructs without altering their original implementation.

The History of Decorators

Decorators have a rich history in the TypeScript and JavaScript ecosystems. The concept of decorators was inspired by Python and other programming languages that use similar constructs to modify or extend the behavior of classes, methods, and properties. The initial decorator's proposal for JavaScript was introduced in 2014, and since then, several versions of the proposal have been developed, with the current one being at stage 3 of the ECMAScript standardization process.

The Syntax of Decorators

Decorators are functions that are prefixed with the '@' symbol and placed immediately before the construct they are meant to modify:

```
@decorator
class MyClass {
  @decorator1
  method() {
    // ...
  }
}

// Same: You can put these decorators on the same line
@decorator class MyClass {
  @decorator2 @decorator1 method() {
    // ...
  }
}
```

Decorator Functions and Their Capabilities

A decorator is a function that takes the construct being decorated as its argument and may return a modified version of the construct or a new construct altogether. Decorators can be used to:

- Modify the behavior of a class, method, accessor, or property
- Add new functionality to a class or method
- Provide metadata for a construct
- Enforce coding standards or best practices

Class Decorators

Class decorators are applied to class constructors and can be used to modify or extend the behavior of a class. Some common use cases for class decorators include:

- Collecting instances of a class
- Freezing instances of a class
- Making classes function-callable

Example: Collecting instances of a class

```
type Constructor<T = {}> = new (...args: any[]) => T;

class InstanceCollector {
  instances = new Set();

  install = <Class extends Constructor>(
    Value: Class,
    context: ClassDecoratorContext<Class>
  ) => {
    const _this = this;
    return class extends Value {
      constructor(...args: any[]) {
        super(...args);
        _this.instances.add(this);
      }
    };
```

```
@collector.install
class Calculator {
  add(a: number, b: number): number {
    return a + b;
  }
}

const calculator1 = new Calculator();
const calculator2 = new Calculator();

console.log('instances: ', collector.instances);
```

Method Decorators

Method decorators are applied to class methods and can be used to modify or extend the behavior of a method. Some common use cases for method decorators include:

- Tracing method invocations
- Binding methods to instances
- Applying functions to methods

Example: Tracing method invocations

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Args) => Return
  >
) {
  const methodName = String(context.name);
```

```
  function replacementMethod(this: This, ...args: Args): Return {
    console.log(`LOG: Entering method '${methodName}'.`);
    const result = target.call(this, ...args);
    console.log(`LOG: Exiting method '${methodName}'.`);
    return result;
  }

  return replacementMethod;
}

class Calculator {
  @log
  add(a: number, b: number): number {
    return a + b;
  }
}

const calculator = new Calculator();
console.log(calculator.add(2, 3));
```

Getter and Setter Decorators

Getter and setter decorators are applied to class accessors, allowing developers to modify or extend their behavior. Common use cases for getter and setter decorators include:

- Compute values lazily and cache
- Implementing read-only properties
- Validating property assignments

Example: Compute values lazily and cache

```
function lazy<This, Return>(
  target: (this: This) => Return,
  context: ClassGetterDecoratorContext<This, Return>
```

```
) {
  return function (this: This): Return {
    const value = target.call(this);
    Object.defineProperty(this, context.name, { value, enumerable: true });
    return value;
  };
}

class MyClass {
  private _expensiveValue: number | null = null;

  @lazy
  get expensiveValue(): number {
    this._expensiveValue ??= computeExpensiveValue();
    return this._expensiveValue;
  }
}

function computeExpensiveValue(): number {
  // Expensive computation here…
  console.log('computing...'); // Only call once

  return 42;
}

const obj = new MyClass();

console.log(obj.expensiveValue);
console.log(obj.expensiveValue);
console.log(obj.expensiveValue);
```

Field Decorators

Field decorators are applied to class fields and can be used to modify or extend the behavior of a field. Common use cases for field decorators include:

- Changing initialization values of fields
- Implementing read-only fields
- Dependency injection

- Emulating enums

Example: Changing initialization values of fields

```
function addOne<T>(
  target: undefined,
  context: ClassFieldDecoratorContext<T, number>
) {
  return function (this: T, value: number) {
    console.log('addOne: ', value); // 3
    return value + 1;
  };
}

function addTwo<T>(
  target: undefined,
  context: ClassFieldDecoratorContext<T, number>
) {
  return function (this: T, value: number) {
    console.log('addTwo: ', value); // 1
    return value + 2;
  };
}


class MyClass {
  @addOne
  @addTwo
  x = 1;
}

console.log(new MyClass().x); // 4
```

There is an additional knowledge point here, when you stack multiple decorators, they will run in “reverse order”. In this example, you can see that `1` is printed first in `addTwo`, and then `addOne`.

Auto-Accessor Decorators

Auto-accessors are a new language feature that simplifies the creation of getter and setter pairs:

```
class C {
  accessor x = 1;
}


// Same
class C {
  #x = 1;

  get x() {
    return this.#x;
  }

  set x(val) {
    this.#x = val;
  }
}
```

Not only is this a convenient way of expressing simple accessor pairs, but it helps avoid problems that occur when decorator authors try to replace instance fields with accessors on the prototype, because ECMAScript instance fields shadow accessors when they are mounted on the instance.

It can also use decorators, such as the following read-only automatic accessor:

```
function readOnly<This, Return>(
  target: ClassAccessorDecoratorTarget<This, Return>,
  context: ClassAccessorDecoratorContext<This, Return>
) {
  const result: ClassAccessorDecoratorResult<This, Return> = {
    get(this: This) {
```

```
        return target.get.call(this);
      },
      set() {
        throw new Error(
          `Cannot assign to read-only property '${String(context.name)}'.`
        );
      },
    };

    return result;
}

class MyClass {
  @readOnly accessor myValue = 123;
}

const obj = new MyClass();

console.log(obj.myValue);
obj.myValue = 456; // Error: Cannot assign to read-only property 'myValue'.
console.log(obj.myValue);
```

Conclusion

Decorators are a powerful feature in TypeScript that provide an elegant way to modify or extend the behavior of classes, methods, accessors, and properties. As the decorator's proposal continues to evolve and mature, you can gradually choose to apply it to your project.

References

[1] https://github.com/microsoft/TypeScript/pull/50820

Thanks for reading. If you like such stories and want to support me, please consider becoming a Medium member. It costs $5 per month and gives unlimited access to Medium content. I'll get a little commission if you sign up via my link.

Typescript JavaScript Programming Web Development

Front End Development

Written by Zack

1.4K Followers

Programmer #JavaScript #Rust

Follow

More from Zack

Zack

Unlocking the Magic of Infer in TypeScript

Infer Your Way to TypeScript Mastery: Unlocking Dynamic and Expressive Types

Zack in Level Up Coding

How to Commit Multiline Messages in git commit

There are some easy ways to do this, it all depends on your personal preference

5 min read · Apr 1, 2023

103 1

3 min read · Sep 25, 2022

321 2

Zack

Simplify Syntax Highlighting with highlight.js

A Quick Start Tutorial

5 min read · Apr 9, 2023

18 2

Zack in Level Up Coding

How To Read a File Line by Line in JavaScript

Which is the better way?

3 min read · Aug 21, 2022

76 1

See all from Zack

Recommended from Medium

Faez in Dev Genius

TypeScript Decorators

Decorators in TypeScript

3 min read · Feb 6, 2024

7

Tari Ibaba in JavaScript in Plain English

Why You Need To Stop Using Nested If Statements

There's something much cleaner and readable.

· 4 min read · 6 days ago

142 1

Lists

General Coding Knowledge

20 stories · 1093 saves

Coding & Development

11 stories · 549 saves

Stories to Help You Grow as a Software Developer

19 stories · 964 saves

ChatGPT

21 stories · 559 saves

Davide Passafaro

Angular HostAttributeToken: the new way to inject attributes

Angular new HostAttributeToken API allows you to inject host attributes using the inject...

3 min read · Mar 26, 2024

178 2

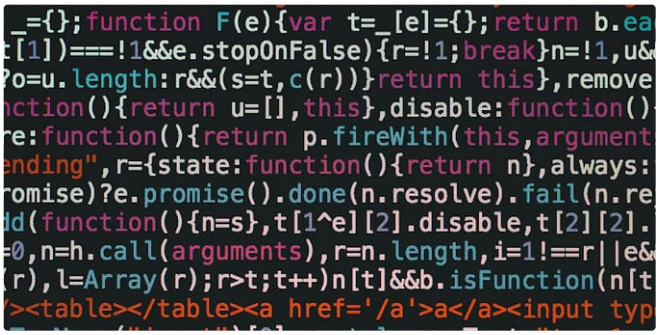
Cihan in Interesting Coding

Ambient Modules in TypeScript — with a Special Example

In TypeScript, we have a feature known as ambient modules. These allow us to import...

· 6 min read · Nov 8, 2023

51



Chandan Kumar

10 Angular Hacks to Supercharge Your Development Workflow

Photo by Lautaro Andreani on Unsplash

3 min read · Mar 5, 2024

86 1

Rainer Hahnekamp in ngconf

How do I test code using inject()?

This article discusses testing Angular code, which uses the inject function for dependen...

6 min read · Mar 5, 2024

26

See more recommendations