JUNE 9, 2015

Decorators & metadata reflection in TypeScript: From Novice to Expert (Part III)

An in-depth look to the TypeScript implementation of decorators and how they make possible new exciting JavaScript features like reflection or dependency injection.

This article is the third part of a series:

- **PART I: Method decorators**
- **PART II: Property decorators & Class decorators**
- **PART III: Parameter decorators & Decorator factory**
- **PART IV: Types serialization & The metadata reflection API**

In the previous post in this series we learned what are decorators and how they are implemented in TypeScript. We know how to work with class, method and property decorators.

In this post we will learn about:

- The remaining type of decorator: the **parameter decorator**.
- How to implement a **decorator factory**.
- How to implement **configurable decorators**.

We are going to use the following class to showcase these concepts.

```
class Person {

  public name: string;
  public surname: string;

  constructor(name : string, surname : string) {
    this.name = name;
```

```
    this.surname = surname;
  }

  public saySomething(something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
  }
}
```

Let's get started.

1. Parameter decorators

As we already know, the signature of a `ParameterDecorator` looks as follows.

```
declare type ParameterDecorator = (target: Object, propertyKey:
string | symbol, parameterIndex: number) => void;
```

We can use a parameter decorator named `logParameter` as follows:

```
class Person {

  public name: string;
  public surname: string;

  constructor(name : string, surname : string) {
    this.name = name;
    this.surname = surname;
  }

  public saySomething(@logParameter something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
  }
}
```

When compiled into JavaScript the `__decorate` method (which we explained in PART I) is invoked here.

```
    Object.defineProperty(Person.prototype, "saySomething",
        __decorate([
            __param(0, logParameter)
        ], Person.prototype, "saySomething",
Object.getOwnPropertyDescriptor(Person.prototype,
"saySomething")));
    return Person;
```

If we compare it with the previous decorators we could assume that because the `Object.defineProperty()` is invoked, the method

`saySomething` will be replaced by the value returned by the `__decorated` function (like in the method decorator). This assumption is wrong.

If we examine the code snippet above, we will notice that there is a new function named `__param`.

The `__param` function is generated by the TypeScript compiler and looks as follows:

```
var __param = this.__param || function (index, decorator) {

    // return a decorator function (wrapper)
    return function (target, key) {

        // apply decorator (return is ignored)
        decorator(target, key, index);
    }
};
```

The `__param` function returns a decorator that wraps the parameter decorator (refereed as `decorator`).

As we can see when the parameter decorator is invoked, its return is ignored. This means that when the `__decorate` function is invoked, its return will not be used to override the `saySomething` method .

This is the reason why **parameter decorators don't return**.

The decorator wrapper in `__param` is used to store the index of the parameter in a closure. The index is just the position in the list of arguments.

```
class foo {
  // foo index === 0
  public foo(@logParameter foo: string) : string {
    return "bar";
  }
  // bar index === 1
  public foobar(foo: string, @logParameter bar: string) : string {
    return "foobar";
  }
}
```

Now we know that **a parameter decorator takes 3 parameters**:

- The **prototype** of the class being decorated.
- The **name** of the method that contains the parameter being decorated.

- The **index** of that parameter being decorated.

Let's implement the `logProperty` decorator.

```
function logParameter(target: any, key : string, index : number) {
  var metadataKey = `log_${key}_parameters`;
  if (Array.isArray(target[metadataKey])) {
    target[metadataKey].push(index);
  }
  else {
    target[metadataKey] = [index];
  }
}
```

The parameter decorator above adds a new property (`metadataKey`) to the class prototype. The new property is an array and contains the indices of the parameters being decorated. We can consider this new property as **metadata**.

A parameter decorator is not supposed to modify the behavior of a constructor, method or property. **A parameter decorator should only be used to generate some sort of metadata**.

Once the metadata has been created we can use another decorator to read it. For example, in the example bellow we can find and updated version of the method decorator that we created in PART II.

The original version logged in console the function name and all its arguments when it was invoked.

The new version **reads the metadata** to log in console only the parameters that have been decorated using the parameter decorator.

```
class Person {

  public name: string;
  public surname: string;

  constructor(name : string, surname : string) {
    this.name = name;
    this.surname = surname;
  }

  @logMethod
  public saySomething(@logParameter something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
  }
```

```
}

function logMethod(target: Function, key: string, descriptor: any)
{
  var originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {

    var metadataKey = `__log_${key}_parameters`;
    var indices = target[metadataKey];

    if (Array.isArray(indices)) {

      for (var i = 0; i < args.length; i++) {

        if (indices.indexOf(i) !== -1) {

          var arg = args[i];
          var argStr = JSON.stringify(arg) || arg.toString();
          console.log(`${key} arg[${i}]: ${argStr}`);
        }
      }
      var result = originalMethod.apply(this, args);
      return result;
    }
    else {

      var a = args.map(a => (JSON.stringify(a) ||
a.toString())).join();
      var result = originalMethod.apply(this, args);
      var r = JSON.stringify(result);
      console.log(`Call: ${key}(${a}) => ${r}`);
      return result;
    }
  }
  return descriptor;
}
```

In the PART IV of this series we will learn a better way to work with metadata: **The metadata reflection API**. The bellow is just a sneak peek of what we will learn.

```
function logParameter(target: any, key: string, index: number) {
  var indices = Reflect.getMetadata(`log_${key}_parameters`,
target, key) || [];
  indices.push(index);
  Reflect.defineMetadata(`log_${key}_parameters`, indices, target,
key);
}
```

2. Decorator factory

The official TypeScript decorators proposal defines a decorator factory as follows:

> **A decorator factory is a function that can accept any number of arguments, and must return one of the types of decorator.**

We have learned how to implement and consume all the available types of decorator (class, method, property and parameter) but there is something that we can improve. Let's consider the following code snippet:

```
@logClass
class Person {

  @logProperty
  public name: string;

  public surname: string;

  constructor(name : string, surname : string) {
    this.name = name;
    this.surname = surname;
  }

  @logMethod
  public saySomething(@logParameter something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
  }
}
```

The above works but it would be better if we could just **consume a decorator everywhere without having to worry about its type** as follows:

```
@log
class Person {

  @log
  public name: string;

  public surname: string;

  constructor(name : string, surname : string) {
    this.name = name;
    this.surname = surname;
  }

  @log
  public saySomething(@log something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
  }
}
```

We can achieve this by wrapping all the decorators with a decorator factory. The decorator factory is able to identify what type of decorator is required by checking the arguments passed to the decorator:

```
function log(...args : any[]) {
  switch(args.length) {
    case 1:
      return logClass.apply(this, args);
    case 2:
      return logProperty.apply(this, args);
    case 3:
      if(typeof args[2] === "number") {
        return logParameter.apply(this, args);
      }
      return logMethod.apply(this, args);
    default:
      throw new Error("Decorators are not valid here!");
  }
}
```

3. Configurable decorators

The last thing that we will learn in this post is how to **allow developers to pass arguments to a decorator when it is consumed**.

```
@logClassWithArgs({ when : { name : "remo"} })
class Person {
  public name: string;

  // ...
}
```

We can use a decorator factory to create configurable decorators.

```
function logClassWithArgs(filter: Object) {
    return (target: Object) => {
        // implement class decorator here, the class decorator
        // will have access to the decorator arguments (filter)
        // because they are  stored in a closure
    }
}
```

We can apply the same idea to the other decorator types (method, property and parameter) to make them configurable.

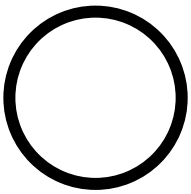
Conclusion

We now understand in-depth 4 out of the 4 available types of decorators, how to create a decorator factory and how to use a decorator factory to implement configurable decorators.

In the next post we will learn how to use **the metadata reflection API**.

Don't forget to subscribe if you don't want to miss it out more JavaScript and TypeScript content.

Please feel free to talk about this article with us via @OweR_ReLoaDeD and @WolkSoftwareLtd



180 KUDOS

NOW READ THIS

Writing JavaScript APIs

This article covers the most important things that you will need to consider before and while writing your own utilities and libraries. We'll focus on how to make your code accessible to other developers. A couple of topics will be... Continue →