SEPTEMBER 9, 2015

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

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 fourth 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**

I will assume that you have already read all the previous posts of these series.

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, property and parameter decorators, how to create a decorator factory and how to use a decorator factory to implement configurable decorators.

In this post we will learn about:

- **1. Why we need reflection in JavaScript?**
- **2. The metadata reflection API**
- **3. Basic type serialization**
- **4. Complex type serialization**

Let’s start by learning why we need reflection in JavaScript.

1. Why we need reflection in JavaScript?

The name reflection is used to describe code which is able to inspect other code in the same system (or itself).

Reflection is useful for a number of use cases (Composition/Dependency Injection, Run-time Type Assertions, Testing).

Our JavaScript applications are getting bigger and bigger and we are starting to need some tools (like inversion of control containers) and features like (run-time type assertions) to manage this increasing complexity. The problem is that because there is not reflection in JavaScript some of this tools and features cannot be implemented, or at least they can not be implemented to be as powerful as they are in programming languages like C# or Java.

A powerful reflection API should allow us to examine an unknown object at run-time and find out everything about it. We should be able to find things like:

- The name of the entity.
- The type of the entity.
- Which interfaces are implemented by the entity.
- The name and types of the properties of the entity.
- The name and types of the constructor arguments of the entity.

In JavaScript we can use functions like Object.getOwnPropertyDescriptor() or Object.keys() to find some information about an entity but we need reflection to implement more powerful development tools.

However, things are about to change because TypeScript is starting to support some Reflection features. Let's take a look to this features:

2. The metadata reflection API

The native JavaScript support for a metadata reflection API is in an early stage of development. There is a proposal to add Decorators to ES7, along with a prototype for an ES7 Reflection API for Decorator Metadata, which is available online.

Some of the guys from the TypeScript team have been working on a Polyfill for the prototype of the ES7 Reflection API and the TypeScript compiler is now able to emit some serialized design-time type metadata for decorators.

We can use this metadata reflection API polyfill by using the reflect-metadata package:

```
npm install reflect-metadata;
```

We must use it with TypeScript 1.5 and the compiler flag `emitDecoratorMetadata` set to true. We also need to including a reference to `reflect-metadata.d.ts`. and load the `Reflect.js` file.

We then need to implement our own decorators and use one of the available **reflect metadata design keys**. For the moment there are only three available:

- **Type metadata** uses the metadata key `"design:type"`.
- **Parameter type metadata** uses the metadata key `"design:paramtypes"`.
- **Return type metadata** uses the metadata key `"design:returntype"`.

Let's see a couple of examples.

A) Obtaining type metadata using the reflect metadata API

Let's declare the following property decorator:

```
function logType(target : any, key : string) {
  var t = Reflect.getMetadata("design:type", target, key);
  console.log(`${key} type: ${t.name}`);
}
```

We can then apply it to one of the properties of a class to obtain its type:

```
class Demo{
  @logType // apply property decorator
  public attr1 : string;
}
```

The example above logs the following in console:

```
attr1 type: String
```

B) Obtaining Parameter type metadata using the reflect metadata API

Let’s declare the following parameter decorator:

```
    function logParamTypes(target : any, key : string) {
      var types = Reflect.getMetadata("design:paramtypes", target,
key);
      var s = types.map(a => a.name).join();
      console.log(`${key} param types: ${s}`);
    }
```

We can then apply it to one of the method of a class to obtain information about the types of its arguments:

```
    class Foo {}
    interface IFoo {}

    class Demo{
      @logParameters // apply parameter decorator
      doSomething(
        param1 : string,
        param2 : number,
        param3 : Foo,
        param4 : { test : string },
        param5 : IFoo,
        param6 : Function,
        param7 : (a : number) => void,
      ) : number {
          return 1
      }
    }
```

The example above logs the following in console:

```
    doSomething param types: String, Number, Foo, Object, Object,
Function, Function
```

C) Obtaining return type metadata using the reflect metadata API

We can also get information about the return type of a method using the `"design:returntype"` metadata key:

```
Reflect.getMetadata("design:returntype", target, key);
```

3. Basic type serialization

Let's take a look to the `"design:paramtypes"` example above again. Notice the that interfaces `IFoo` and object literal `{ test : string}` are serialized as `Object`. This is because TypeScript only supports basic type serialization. The basic type serialization rules are:

- `number` serialized as `Number`
- `string` serialized as `String`
- `boolean` serialized as `Boolean`
- `any` serialized as `Object`
- `void` serializes as `undefined`
- `Array` serialized as `Array`
- If a `Tuple`, serialized as `Array`
- If a `class` serialized it as the class constructor
- If an `Enum` serialized it as `Number`
- If has at least one call signature, serialized as `Function`
- Otherwise serialized as `Object` (Including interfaces)

Interfaces and object literals may be serialize in the future via **complex type serialization** but this feature is not available at this time.

4. Complex types serialization

The TypeScript team is working on a proposal that will allow us to generate metadata for complex types.

They proposal describes how some complex types will be serialized. The serialization rules above will still be used for basic type but a different serialization logic will be used for complex types. In the proposal there is a base type that is used to describe all the possible types:

```
/**
  * Basic shape for a type.
  */
interface _Type {
  /**
    * Describes the specific shape of the type.
    * @remarks
    * One of: "typeparameter", "typereference", "interface",
"tuple", "union",
    * or "function".
    */
  kind: string;
}
```

We can also find the classes that will be used to describe each of the possible types. For example, we can find the class proposed to be used to serialize genetic interfaces `interface foo<bar> { /* ... */}`:

```
/**
  * Describes a generic interface.
  */
interface InterfaceType extends _Type {
  kind: string; // "interface"

  /**
    * Generic type parameters for the type. May be undefined.
    */
  typeParameters?: TypeParameter[];

  /**
    * Implemented interfaces.
    */
  implements?: Type[];

  /**
    * Members for the type. May be undefined.
    * @remarks Contains property, accessor, and method
declarations.
    */
  members?: { [key: string | symbol | number]: Type; };

  /**
    * Call signatures for the type. May be undefined.
    */
  call?: Signature[];

  /**
    * Construct signatures for the type. May be undefined.
    */
  construct?: Signature[];

  /**
    * Index signatures for the type. May be undefined.
    */
  index?: Signature[];
}
```

As we can see above there will be an attribute which indicates the implemented interfaces:

```
  /**
    * Implemented interfaces.
    */
  implements?: Type[];
```

That information could be used to do things like validate if an entity implements certain interface at run-time, which could be really useful for an IoC container.

We don’t know when complex type serialization support will be added to TypeScript but we cannot wait because we have plans to use it to add some cool features to our awesome IoC container for JavaScript: InversifyJS.

5. Conclusion

In this series we have learned 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.

We also know how to work with the metadata reflection API.

We will keep this blog updated and write more about the metadata reflection API in the future. Don’t forget to subscribe if you don’t want to miss it out!

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

NOW READ THIS

Introducing InversifyJS

A lightweight inversion of control container for TypeScript & JavaScript apps A few months ago I started to think about the possibility of creating a requireJS TypeScript fork in order to add dependency inversion (DI) support. After... **Continue →**