[TS] 6. Decorators

이번 포스팅에서는 현재 JavaScript에서도 ts39/proposal stage-2에 올라와있는 `Decorator`에 대해 알아보겠습니다.

Table of contents

Setup

자바스크립트 babel환경에서 데코레이터를 테스트해보기 위해서는 babel 플러그인이 추가적으로 필요합니다.

```
$ npm install babel-core babel-plugin-transform-decorator
```

`babel-core` 를 기본으로 하며, babel-plugin을 추가적으로 설치해줍니다.

```
{
  //...
  "plugins": ["transform-decorators-legacy"]
}
```

해당 프로젝트의 babel설정을 담고 있는 `.babelrc` 파일에 설치한 플러그인을 추가해줍니다. 보다 구체적인 해당 개발환경은 여기를 참고해주세요.

TyeScript에서는 `tsconfig.json` 의 `compilerOption` 을 다음과 같이 변경해줍니다.

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

Intro

TypeScript(JavaScript)에서 `@` 이라는 character로 사용하는 문법을 `Decorator(데코레이터)` 라고 합니다. 자바를 경험해보신 분이라면 `Annotation` 인가? 라고 생각하기 쉬운데요, 조금 다릅니다. 데코레이터는 **함수** 라고 할 수 있습니다. 데코레이터는 말 그대로 코드 조각을 장식해주는 역할을 하며 타입스크립트에서는 그 기능을 함수로 구현할 수 있습니다.

Decorator는 클래스 선언, 메서드, 접근 제어자, 속성 또는 매개 변수에 첨부할 수 있는 특별한 종류의 선언입니다. 데코레이터는 `@expression` 형식을 사용하는데, expression은 데코레이팅 된 선언에 대한 정보와 함께 존재하며 이는 **런타임에** 호출됩니다.

참조(reference)

데코레이터는 `@decorator`과 같이 사용할 수 있으며 `@[name]`의 형식일 때 `name`에 해당하는 이름의 함수를 참조하게 됩니다.

실행 시점(execute time)

이렇게 데코레이터로 정의된 함수는 데코레이터가 적용된 메소드가 실행되거나 클래스가 `new` 라는 키워드를 통해 인스턴스화 될 때가 아닌 런타임 때 실행됩니다. 즉, 매번 실행되지 않습니다.

그럼 데코레이터가 메소드에 적용되는 경우, 클래스에 적용되는 경우, 프로퍼티에 적용되는 경우 이렇게 세 가지로 나누어 코드를 살펴보겠습니다.

Decorator to Method

> 메소드에 적용되는 경우

우선 데코레이터로 사용할 `chaining` 이라는 함수를 정의해줍니다.

```
function chaining(target: any, key: string, descriptor: P
  console.log(target); // {bark: f, constructor: f}
  console.log(key); // bark
  console.log(descriptor); // {value: f, writable: true, 
}
```

위 함수는 추후 메소드에 `@chaining` 형식으로 사용될 함수입니다. `@` 과 함께 함수가 호출되는 경우 받게 되는 파라미터는 다음과 같습니다.

> - target : 속성을 정의하고자 하는 객체
> - name : 속성의 이름
> - descriptor : 새로 정의하고자 하는 속성에 대한 설명

이는 `Object.defineProperty()` 를 통해 이를 정의하고 있기 때문입니다.

`target` 은 해당 메소드가 속해있는 클래스 프로토타입을 가리키게 되며 `Pet` 의 프로토타입에는 `constructor` 와 `bark` 메소드가 있는 것을 확인할 수 있습니다. `name` 은 데코레이터가 적용된 메소드의 이름이 됩니다. `descriptor` 는 `defineProperty` 에서 정의할 수 있는 각각의 속성값들이 됩니다.

Object의 `defineProperty`에 해당하는 보다 자세한 내용은 여기에서 살펴보실 수 있습니다. 그럼 각각을 활용해서 `chaining` 기능을 구현해보겠습니다.

```
// Decorator to method
function chaining(target: any, key: string, descriptor: P
  const fn: Function = descriptor.value;

  descriptor.value = function(...args: any[]) {
    fn.apply(target, args);
    return target;
  }
}
```

descriptor의 `value`가 데코레이터가 적용된 함수, 즉 실행 대상이라고 할 수 있습니다. `descriptor.value`를 재정의(override)하기 전에 `fn`이라는 변수로 caching해둔 다음, 호출한 후의 일을 정의하기 위해 위와 같이 재정의 해줍니다. 재정의 하기 전 caching 해둔 함수를 호출하기 위해서 `apply` 함수를 사용했습니다. 어떠한 변수가 얼만큼 전달될지 모르니 rest parameter를 통해 `fn`을 호출해주는 코드입니다.

위와 같이 `descriptor.value`가 재정의 되면 `chaining`이 적용된 메소드는 재정의된대로 호출되게 됩니다.

> `apply` 함수에 대한 내용은 여기를 참고해주세요.

```
class Pet {
  @chaining
  bark() {
  }
}
```

위와 같이 적용해보겠습니다.

```
const pet = new Pet();
```

`Pet` 클래스에서 `bark` 라는 메소드는 `Pet.prototype.bark` 로 됩니다. class syntax 내부에서 위 코드에서는 `bark` 라는 메소드가 `Pet` 의 prototype의 프로퍼티로 추가되기 전에 `decorate` 함수가 실행되어 본래 `bark` 라는 메소드에서 정의된 것에 추가적인 '**장식**' 을 더해 prototype에 추가되도록 합니다.

> 만약 compile target이 ES5보다 낮다면 `PropertyDescriptor` 값으로 `undefined` 이 전달됩니다.

```
pet.bark().bark();
```

위 데코레이터의 효과로 `return this;` 를 해주지 않아도 chaining 기능을 사용하여 메소드를 호출할 수 있습니다.

Decorator to Class

하지만 데코레이터가 class에 적용되었을 때는 그 signature가 조금 달라집니다. 클래스 데코레이터는 클래스 선언 바로 전에 선언됩니다. 클래스 데코레이터는 클래스 생성자에 적용되며 클래스 정의를 관찰, 수정 또는 대체하는 데 사용할 수 있습니다.

클래스 데코레이터가 값을 반환하면 클래스 선언을 제공된 생성자 함수로 바꿉니다.

```
function component(target, name, descriptor) {
  console.log(target); // ...
  console.log(name); // undefined
  console.log(descriptor); //undefined
}
```

메소드에 데코레이터를 적용하듯이 데코레이터 함수를 선언하면 올바른 선언을 할 수 없습니다. 클래스에 적용되는 데코레이터 함수에 전달되는 인자는 `constructor` 하나입니다. 제대로 된 데코레이터 선언은 다음과 같습니다.

```
function classDecorator<T extends {new(...args:any[]):{}}>
  return class extends constructor {
    newProperty = "new property";
  }
}
```

클래스에 적용되는 데코레이터 함수 내에서 새로운 생성자 함수를 반환하면 원래 프로토타입을 유지해야 합니다. 런타임에 데코레이터를 적용하는 로직은 이를 수행하지 않기 때문입니다. 위 코드에서는 기존의 프로토타입을 유지하기 위해 적용되는 클래스의 `constructor` 를 `extends` 합니다.

```
@classDecorator
class Pet {
  constructor(name: string) {
    this.name = name;
  }
}

const pet = new Pet("async");
console.log(pet.newProperty); // new Property
```

`classDecorator` 데코레이터가 적용된 `Pet` 클래스의 인스턴스에는 `newProperty` 가 존재하지 않지만 데코레이터 함수에서 해당 클래스의 constructor를 재정의했기 대문에 `newProperty` 에 접근할 수 있습니다.

Decorator with parameter

> 파라미터를 받는 데코레이터

데코레이터 함수에 인자를 넘겨줄 수 있습니다. 이 인자는 무엇이든 될 수 있습니다. 예제 코드로 descriptor의 `enumerable` 속성을 변경하는 데코레이터를 만들어보겠습니다.

```
function enumerableToFalse(target: any, propertyKey: stri
  descriptor.enumerable = false;
};
```

이렇게 정의하면 `enumerableToFalse` 이 적용된 메소드의 enumerable 속성은 false가 됩니다. 위 `enumerableToFalse` 함수를 한 번 감싸서 반환하는 함수를 만들면 다음과 같습니다.

```
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, desc
    descriptor.enumerable = value;
  };
}
```

이제 이 함수를 데코레이터 함수로 사용할 수 있습니다. `value`에 해당하는 값으로 데코레이터를 적용하는 메소드의 `enumerable` 속성을 제어할 수 있습니다.

```
class Pet {
  @enumerable(false)
  bark() {
  //...
  }
}
```

위와 같이 `false` 라는 인자를 받는 데코레이터를 정의했습니다. 저 인자에는 함수도 들어갈 수 있으며 데코레이터도 들어갈 수 있습니다.

마무리

target을 ES5로 지정해야 제대로 된 데코레이터를 사용할 수 있어서 아직 한계가 있는 Decorator지만 React에서는 HOC(High-Order-Component)에 많이 사용하고 있는 Decorator 였습니다!

감사합니다.

6. Decorator in TypeScript end

Reference

- TypeScript Official Document - Generics
- https://github.com/wycats/javascript-decorators
- https://medium.com/google-developers/exploring-es7-decorators-76ecb65fb841
- https://www.sitepoint.com/javascript-decorators-what-they-are/
- https://cabbageapps.com/fell-love-js-decorators/
- https://javarouka.github.io/blog/2016/09/30/decorator-exploring/#class-il-gyeongu
- https://github.com/jayphelps/core-decorators