介绍

随着TypeScript和ES6里引入了类，在一些场景下我们需要额外的特性来支持标注或修改类及其成员。
装饰器（Decorators）为我们在类的声明及成员上通过元编程语法添加标注提供了一种方式。
Javascript里的装饰器目前处在建议征集的第二阶段，但在TypeScript里已做为一项实验性特性予以支持。

> 注意　装饰器是一项实验性特性，在未来的版本中可能会发生改变。

若要启用实验性的装饰器特性，你必须在命令行或`tsconfig.json`里启用`experimentalDecorators`编译器选项：

命令行:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

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

装饰器

*装饰器*是一种特殊类型的声明，它能够被附加到类声明，方法，访问符，属性或参数上。
装饰器使用`@expression`这种形式，`expression`求值后必须为一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入。

例如，有一个`@sealed`装饰器，我们会这样定义`sealed`函数：

```
function sealed(target) {
    // do something with "target" ...
}
```

> 注意　后面类装饰器小节里有一个更加详细的例子。

装饰器工厂

如果我们要定制一个修饰器如何应用到一个声明上，我们得写一个装饰器工厂函数。
*装饰器工厂*就是一个简单的函数，它返回一个表达式，以供装饰器在运行时调用。

我们可以通过下面的方式来写一个装饰器工厂函数：

```
function color(value: string) { // 这是一个装饰器工厂
```

```
    return function (target) { //  这是装饰器
        // do something with "target" and "value"...
    }
}
```

注意　下面方法装饰器小节里有一个更加详细的例子。

装饰器组合

多个装饰器可以同时应用到一个声明上，就像下面的示例：

- 书写在同一行上：

```
@f @g x
```

- 书写在多行上：

```
@f
@g
x
```

当多个装饰器应用于一个声明上，它们求值方式与复合函数相似。在这个模型下，当复合f和g时，复合的结果$(f \circ g)(x)$等同于$f(g(x))$。

同样的，在TypeScript里，当多个装饰器应用在一个声明上时会进行如下步骤的操作：

1. 由上至下依次对装饰器表达式求值。
2. 求值的结果会被当作函数，由下至上依次调用。

如果我们使用装饰器工厂的话，可以通过下面的例子来观察它们求值的顺序：

```
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescripto
        console.log("f(): called");
    }
}

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescripto
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

在控制台里会打印出如下结果：

```
f(): evaluated
g(): evaluated
g(): called
```

```
f(): called
```

装饰器求值

类中不同声明上的装饰器将按以下规定的顺序应用：

1. *参数装饰器*，然后依次是*方法装饰器*，*访问符装饰器*，或*属性装饰器*应用到每个实例成员。
2. *参数装饰器*，然后依次是*方法装饰器*，*访问符装饰器*，或*属性装饰器*应用到每个静态成员。
3. *参数装饰器*应用到构造函数。
4. *类装饰器*应用到类。

类装饰器

*类装饰器*在类声明之前被声明（紧靠着类声明）。
类装饰器应用于类构造函数，可以用来监视，修改或替换类定义。
类装饰器不能用在声明文件中(`.d.ts`)，也不能用在任何外部上下文中（比如`declare`的类）。

类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数。

如果类装饰器返回一个值，它会使用提供的构造函数来替换类的声明。

> 注意 如果你要返回一个新的构造函数，你必须注意处理好原来的原型链。
> 在运行时的装饰器调用逻辑中*不会*为你做这些。

下面是使用类装饰器(`@sealed`)的例子，应用在`Greeter`类：

```
@sealed
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

我们可以这样定义`@sealed`装饰器：

```
function sealed(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}
```

当`@sealed`被执行的时候，它将密封此类的构造函数和原型。(注：参见Object.seal)

下面是一个重载构造函数的例子。

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

```
        newProperty = "new property";
        hello = "override";
    }
}

@classDecorator
class Greeter {
    property = "property";
    hello: string;
    constructor(m: string) {
        this.hello = m;
    }
}

console.log(new Greeter("world"));
```

方法装饰器

*方法装饰器*声明在一个方法的声明之前（紧靠着方法声明）。
它会被应用到方法的*属性描述符*上，可以用来监视，修改或者替换方法定义。
方法装饰器不能用在声明文件(`.d.ts`)，重载或者任何外部上下文（比如`declare`的类）中。

方法装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 成员的*属性描述符*。

    注意　如果代码输出目标版本小于`ES5`，*属性描述符*将会是`undefined`。

如果方法装饰器返回一个值，它会被用作方法的*属性描述符*。

    注意　如果代码输出目标版本小于`ES5`返回值会被忽略。

下面是一个方法装饰器（`@enumerable`）的例子，应用于`Greeter`类的方法上：

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }

    @enumerable(false)
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

我们可以用下面的函数声明来定义`@enumerable`装饰器：

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

这里的`@enumerable(false)`是一个装饰器工厂。
当装饰器`@enumerable(false)`被调用时，它会修改属性描述符的`enumerable`属性。

访问器装饰器

*访问器装饰器*声明在一个访问器的声明之前（紧靠着访问器声明）。
访问器装饰器应用于访问器的*属性描述符*并且可以用来监视，修改或替换一个访问器的定义。
访问器装饰器不能用在声明文件中（.d.ts），或者任何外部上下文（比如`declare`的类）里。

> 注意　TypeScript不允许同时装饰一个成员的`get`和`set`访问器。取而代之的是，一个成员的所有装饰的必须应用在文档顺序的第一个访问器上。这是因为，在装饰器应用于一个*属性描述符*时，它联合了`get`和`set`访问器，而不是分开声明的。

访问器装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 成员的*属性描述符*。

> 注意　如果代码输出目标版本小于`ES5`，*Property Descriptor*将会是`undefined`。

如果访问器装饰器返回一个值，它会被用作方法的*属性描述符*。

> 注意　如果代码输出目标版本小于`ES5`返回值会被忽略。

下面是使用了访问器装饰器（`@configurable`）的例子，应用于`Point`类的成员上：

```
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }

    @configurable(false)
    get x() { return this._x; }

    @configurable(false)
    get y() { return this._y; }
}
```

我们可以通过如下函数声明来定义`@configurable`装饰器：

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

属性装饰器

*属性装饰器*声明在一个属性声明之前（紧靠着属性声明）。
属性装饰器不能用在声明文件中（.d.ts），或者任何外部上下文（比如`declare`的类）里。

属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。

> 注意　*属性描述符*不会做为参数传入属性装饰器，这与TypeScript是如何初始化属性装饰器的有关。
> 因为目前没有办法在定义一个原型对象的成员时描述一个实例属性，并且没办法监视或修改一个属性的初始化方法。
> 因此，属性描述符只能用来监视类中是否声明了某个名字的属性。

如果属性装饰器返回一个值，它会被用作方法的*属性描述符*。

> 注意　如果代码输出目标版本小于`ES5`，返回值会被忽略。

如果访问符装饰器返回一个值，它会被用作方法的*属性描述符*。

我们可以用它来记录这个属性的元数据，如下例所示：

```
class Greeter {
    @format("Hello, %s")
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        let formatString = getFormat(this, "greeting");
        return formatString.replace("%s", this.greeting);
    }
}
```

然后定义`@format`装饰器和`getFormat`函数：

```
import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {
    return Reflect.metadata(formatMetadataKey, formatString);
}

function getFormat(target: any, propertyKey: string) {
    return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}
```

这个`@format("Hello, %s")`装饰器是个 装饰器工厂。
当`@format("Hello, %s")`被调用时，它添加一条这个属性的元数据，通过`reflect-`

metadata库里的Reflect.metadata函数。
当getFormat被调用时，它读取格式的元数据。

> 注意　这个例子需要使用reflect-metadata库。
> 查看元数据了解reflect-metadata库更详细的信息。

参数装饰器

*参数装饰器*声明在一个参数声明之前（紧靠着参数声明）。
参数装饰器应用于类构造函数或方法声明。
参数装饰器不能用在声明文件（.d.ts），重载或其它外部上下文（比如declare的类）里。

参数装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 参数在函数参数列表中的索引。

> 注意　参数装饰器只能用来监视一个方法的参数是否被传入。

参数装饰器的返回值会被忽略。

下例定义了参数装饰器（@required）并应用于Greeter类方法的一个参数：

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    @validate
    greet(@required name: string) {
        return "Hello " + name + ", " + this.greeting;
    }
}
```

然后我们使用下面的函数定义 @required 和 @validate 装饰器：

```
import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol, parameterIndex:
    let existingRequiredParameters: number[] = Reflect.getOwnMetadata(requiredM
    existingRequiredParameters.push(parameterIndex);
    Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, tar
}

function validate(target: any, propertyName: string, descriptor: TypedPropertyD
    let method = descriptor.value;
    descriptor.value = function () {
        let requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetad
        if (requiredParameters) {
```

```
            for (let parameterIndex of requiredParameters) {
                if (parameterIndex >= arguments.length || arguments[parameterIn
                    throw new Error("Missing required argument.");
                }
            }
        }

        return method.apply(this, arguments);
    }
}
```

`@required`装饰器添加了元数据实体把参数标记为必需的。
`@validate`装饰器把`greet`方法包裹在一个函数里在调用原先的函数前验证函数参数。

> 注意　这个例子使用了`reflect-metadata`库。
> 查看元数据了解`reflect-metadata`库的更多信息。

元数据

一些例子使用了`reflect-metadata`库来支持实验性的metadata API。
这个库还不是ECMAScript (JavaScript)标准的一部分。
然而，当装饰器被ECMAScript官方标准采纳后，这些扩展也将被推荐给ECMAScript以采纳。

你可以通过npm安装这个库：

```
npm i reflect-metadata --save
```

TypeScript支持为带有装饰器的声明生成元数据。
你需要在命令行或`tsconfig.json`里启用`emitDecoratorMetadata`编译器选项。

Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

tsconfig.json:

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

当启用后，只要`reflect-metadata`库被引入了，设计阶段添加的类型信息可以在运行时使用。

如下例所示：

```
import "reflect-metadata";

class Point {
    x: number;
    y: number;
```

```
}

class Line {
    private _p0: Point;
    private _p1: Point;

    @validate
    set p0(value: Point) { this._p0 = value; }
    get p0() { return this._p0; }

    @validate
    set p1(value: Point) { this._p1 = value; }
    get p1() { return this._p1; }
}

function validate<T>(target: any, propertyKey: string, descriptor: TypedPropert
    let set = descriptor.set;
    descriptor.set = function (value: T) {
        let type = Reflect.getMetadata("design:type", target, propertyKey);
        if (!(value instanceof type)) {
            throw new TypeError("Invalid type.");
        }
        set(value);
    }
}
```

TypeScript编译器可以通过`@Reflect.metadata`装饰器注入设计阶段的类型信息。
你可以认为它相当于下面的TypeScript：

```
class Line {
    private _p0: Point;
    private _p1: Point;

    @validate
    @Reflect.metadata("design:type", Point)
    set p0(value: Point) { this._p0 = value; }
    get p0() { return this._p0; }

    @validate
    @Reflect.metadata("design:type", Point)
    set p1(value: Point) { this._p1 = value; }
    get p1() { return this._p1; }
}
```

> **注意**　装饰器元数据是个实验性的特性并且可能在以后的版本中发生破坏性的改变（breaking changes）。