使用 2022-03 装饰器 API 进行 JavaScript 元编程

中文 English

JavaScript 装饰器终于进入了 第 3 阶段！最新版本已经被 Babel 支持，很快也会被 TypeScript 支持。

本文介绍了 ECMAScript 提案 “Decorators”（2022-03 版本，第 3 阶段），作者为 Daniel Ehrenberg 和 Chris Garrett。

装饰器是以 `@` 符号开头的关键字，可以用于类和类成员（如方法）前。例如，`@trace` 就是一个装饰器：

```
class C {
  @trace
  toString() {
    return 'C';
  }
}
```

装饰器会改变被装饰结构的行为。在这个例子中，每次调用 `.toString()` 都会被“追踪”（参数和结果会被输出到控制台）。稍后我们会看到 `@trace` 的实现。

装饰器主要是面向对象的特性，在 Ember、Angular、Vue、Web 组件框架和 MobX 等 OOP 框架和库中非常流行。

关于装饰器，有两个相关角色：

- 库作者需要了解装饰器 API，以便实现它们。
- 库使用者只需知道如何应用装饰器。

本文主要面向库作者：我们将学习装饰器的工作原理，并用这些知识实现几个装饰器。

---

1 装饰器的历史（可选章节）
    1.1 装饰器的历史
    1.1 Babel 装饰器实现的历史
2 什么是装饰器?
    2.1 装饰器函数的结构
    2.1 装饰器能做什么?
    2.1 总结表格
3 装饰器语法与语义的更多信息（可选章节）
    3.1 装饰器表达式的语法
    3.1 装饰器如何执行?
    3.1 装饰器初始化器何时运行?

4 装饰器暴露数据的技巧
4.1 在外部作用域存储暴露的数据
4.1 通过工厂函数管理暴露的数据
4.1 通过类管理暴露的数据
5 类装饰器
5.1 示例：收集实例
5.1 确保 instanceof 正常工作
5.1 示例：冻结实例
5.1 示例：让类可函数调用
6 类方法装饰器
6.1 示例：追踪方法调用
6.1 示例：将方法绑定到实例
6.1 示例：对方法应用函数
7 类 getter 装饰器、setter 装饰器
7.1 示例：惰性计算值
8 类字段装饰器
8.1 示例：改变字段初始化值
8.1 示例：只读字段（实例公有字段）
8.1 示例：依赖注入（实例公有字段）
8.1 示例："友元"可见性（实例私有字段）
8.1 示例：枚举（静态公有字段）
9 自动访问器：类定义的新成员
9.1 为什么需要自动访问器?
10 类自动访问器装饰器
10.1 示例：只读自动访问器
11 常见问题
11.1 为什么函数不能被装饰?
12 更多与装饰器相关的提案
13 资源
13.1 实现
13.1 带装饰器的库
14 致谢
15 延伸阅读

---

1 装饰器的历史（可选章节）

（本章节为可选内容。如果您跳过此章节，仍然可以理解后续内容。）

让我们从装饰器的历史开始。主要回答以下两个问题：

- 为什么这个提案花了这么长时间?
- 为什么感觉 JavaScript 似乎已经有装饰器很多年了?

1.1 装饰器的历史

以下历史描述了：

- 各个团体如何在各自的项目中工作，并在 TC39 提案上进行协作。
- TC39 提案如何通过 TC39 过程 的各个阶段（从 0 开始，到 4 结束，提案准备添加到 ECMAScript 中）。在此过程中，提案经历了多次变更。

以下是相关事件的时间顺序：

- 2014-04-10：Yehuda Katz 向 TC39 提出了装饰器提案。该提案晋升为第 0 阶段。
    - Katz 的提案是与 Ron Buckton 合作创建的。关于该提案的讨论可以追溯到 2013 年 7 月。
- 2014-10-22（ngEurope 大会，巴黎）：Angular 团队宣布 Angular 2.0 正在使用 AtScript 编写，并编译为 JavaScript（通过 Traceur）和 Dart。计划包括基于 TypeScript 的 AtScript，同时添加：
    - 三种类型的 *注解*：
        - *类型注解*
        - *字段注解* 显式声明字段。
        - *元数据注解* 语法与装饰器相同，但仅添加元数据，不改变被注解结构的工作方式。
    - 运行时类型检查
    - 类型反射
- 2015-01-28：Yehuda Katz 和 Jonathan Turner 报告说 Katz 和 TypeScript 团队正在交流想法。
- 2015-03-05（ng-conf，盐湖城）：Angular 团队和 TypeScript 团队宣布 Angular 将从 AtScript 切换到 TypeScript，并且 TypeScript 将采用 AtScript 的一些特性（特别是装饰器）。
- 2015-03-24：装饰器提案进入第 1 阶段。彼时，他们在 GitHub 上有 一个仓库（由 Yehuda Katz 创建），后来迁移到 当前所在位置。
- 2015-07-20：TypeScript 1.5 发布，支持通过 第 1 阶段装饰器，需开启标志 `--experimentalDecorators`。

  多个 JavaScript 项目（如 Angular 和 MobX）使用了这个 TypeScript 特性，这使得 JavaScript 看起来已经有装饰器。

  到目前为止，TypeScript 尚未支持更新版本的装饰器 API。Ron Buckton 的一个拉取请求提供了对第 3 阶段装饰器的支持，可能会在 v4.9 之后的版本中发布。
- 2016-07-28：在 Yehuda Katz 和 Brian Terlson 的介绍后，提案进入第 2 阶段。

- 2017-07-27：Daniel Ehrenberg 在几个月前加入提案后首次进行了装饰器的介绍。他推动了提案的演变，持续了好几年。
- 随后，Chris Garrett 加入了提案，并帮助其进入第 3 阶段，该阶段发生在 2022-03-28。装饰器元数据被移至 一个单独的提案，该提案从第 2 阶段开始。

之所以花了很长时间才进入第 3 阶段，是因为很难让所有利益相关者就 API 达成一致。人们关心的包括与其他特性（如类成员和私有状态）的相互作用以及性能等问题。

1.2 Babel 装饰器实现的历史

Babel 紧密跟踪了装饰器提案的演变，这要感谢 Logan Smyth、Nicolò Ribaudo 和其他人的努力：

- 2015-03-31：Babel 5.0.0 支持第 1 阶段装饰器。
- 2015-11-29 Logan Smyth 的一个外部插件将第 1 阶段装饰器带到了 Babel 6。
- 2018-08-27 Babel 7.0.0 通过官方的 `@babel/plugin-proposal-decorators` 支持第 2 阶段装饰器。
- 官方插件目前支持 以下版本：
  - `"legacy"`: 第 1 阶段装饰器
  - `"2018-09"`: 第 2 阶段装饰器
  - `"2021-12"`: 第 2 阶段装饰器的更新版本
  - `"2022-03"`: 第 3 阶段装饰器

2 什么是装饰器?

装饰器让我们可以改变 JavaScript 结构（如类和方法）的工作方式。让我们回顾一下使用装饰器 `@trace`：

```
class C {
  @trace
  toString() {
    return 'C';
  }
}
```

要实现 `@trace`，我们只需编写一个函数 （确切的实现稍后会展示）：

```
function trace(decoratedMethod) {
  // 返回一个替换 `decoratedMethod` 的函数。
}
```

带有装饰器方法的类大致等同于以下代码：

```
class C {
  toString() {
    return 'C';
  }
}
C.prototype.toString = trace(C.prototype.toString);
```

换句话说：装饰器是一个可以应用于语言结构的函数。我们通过在它们前面加上 `@` 和它的名字来实现这一点。

编写和使用装饰器是*元编程*：

- 我们不直接编写处理用户数据的代码（*编程*）。
- 我们编写一种生成代码的东西，生成的代码可以用来处理用户数据（*元编程*）。

有关元编程的更多信息，请参见“深度 JavaScript”中的“编程与元编程”部分。

2.1 The shape of decorator functions

在我们探索装饰器函数的示例之前，我想先看看它们的 TypeScript 类型签名：

```
type Decorator = (
  value: DecoratedValue, // 字段不同
  context: {
    kind: string;
    name: string | symbol;
    addInitializer(initializer: () => void): void;

    // Don't always exist:
    static: boolean;
    private: boolean;
    access: {get: () => unknown, set: (value: unknown) => void};
  }
) => void | ReplacementValue; // 仅字段不同
```

也就是说，装饰器就是一个函数。它的参数是：

- 装饰器应用于的 `value`。
- 包含 `value` 附加信息的对象 `context`，具有：
  - 关于 `value` 的附加信息（`.static`、`.private`）
  - 一个小型 API（`.access`、`.addInitializer`），具有元编程功能

属性 `.kind` 告诉装饰器它被应用于哪种类型的 JavaScript 结构。我们可以对多种结构使用相同的函数。

目前，装饰器可以应用于类、方法、getter、setter、字段和 *自动访问器*（稍后将解释的一种新类成员）。`.kind` 的值反映了这一点：

- `'class'`

- `'method'`
- `'getter'`
- `'setter'`
- `'accessor'`
- `'field'`

以下是 `Decorator` 的确切类型：

```
type Decorator =
  | ClassDecorator
  | ClassMethodDecorator
  | ClassGetterDecorator
  | ClassSetterDecorator
  | ClassAutoAccessorDecorator
  | ClassFieldDecorator
;
```

我们很快会遇到这些装饰器及其类型签名 – 只有这些部分会有所不同：

- `value` 的类型
- `context` 的某些属性
- 返回类型

2.2 装饰器能做什么？

每个装饰器最多有四种能力：

- 通过改变参数 `value` 来改变被装饰的实体。
- 通过返回兼容的值来替换被装饰的实体：
  - “兼容”意味着返回的值必须与被装饰值具有相同的类型 – 例如，类装饰器必须返回可调用的值。
  - 如果装饰器不想替换被装饰的值，它可以返回 `undefined` – 可以是显式返回，也可以是通过不返回任何内容隐式返回。
- 向其他人暴露对被装饰实体的访问。`context.access` 使其能够通过其方法 `.get()` 和 `.set()` 来实现。
- 在被装饰实体及其容器（如果有的话）存在后处理它们：该功能由 `context.addInitializer` 提供。它允许装饰器注册一个 *初始化器* – 在一切准备就绪时调用的回调（更多细节在稍后解释）。

接下来的小节将演示这些能力。起初我们不会使用 `context.kind` 来检查装饰器应用于哪种类型的构造。稍后我们会这样做。

2.2.1 能力：替换被装饰实体

在下面的示例中，装饰器 `@replaceMethod` 用它返回的函数替换了方法 `.hello()`（B 行）。

```
function replaceMethod() {
  return function () { // (A)
    return `How are you, ${this.name}?`;
  }
}

class Person {
  constructor(name) {
    this.name = name;
  }
  @replaceMethod
  hello() { // (B)
    return `Hi ${this.name}!`;
  }
}

const robin = new Person('Robin');
assert.equal(
  robin.hello(), 'How are you, Robin?'
);
```

2.2.2 能力：向其他人暴露对被装饰实体的访问

在下一个示例中，装饰器 `@exposeAccess` 将一个对象存储在变量 `acc` 中，使我们能够访问 `Color` 实例的 `.green` 属性。

```
let acc;
function exposeAccess(_value, {access}) {
  acc = access;
}

class Color {
  @exposeAccess
  name = 'green'
}

const green = new Color();
assert.equal(
  green.name, 'green'
);
// 使用 `acc` 获取和设置 `green.name`
assert.equal(
  acc.get.call(green), 'green'
);
acc.set.call(green, 'red');
assert.equal(
```

```
  green.name, 'red'
);
```

2.2.3 能力：处理被装饰实体及其容器

在以下代码中，我们使用装饰器 `@collect` 将被装饰方法的键存储在实例属性 `.collectedMethodKeys` 中：

```
function collect(_value, {name, addInitializer}) {
  addInitializer(function () { // (A)
    if (!this.collectedMethodKeys) {
      this.collectedMethodKeys = new Set();
    }
    this.collectedMethodKeys.add(name);
  });
}

class C {
  @collect
  toString() {}
  @collect
  [Symbol.iterator]() {}
}
const inst = new C();
assert.deepEqual(
  inst.collectedMethodKeys,
  new Set(['toString', Symbol.iterator])
);
```

在 A 行添加的初始化函数必须是普通函数，因为需要访问隐式参数 `this`。箭头函数无法提供此访问权限 – 它们的 `this` 是静态作用域的（像任何普通变量一样）。

2.3 总结表格

类型签名：

装饰器种类	`(input) => output`	`.access`
Class	`(func) => func2`	–
Method	`(func) => func2`	`{get}`
Getter	`(func) => func2`	`{get}`
Setter	`(func) => func2`	`{set}`
Auto-accessor	`({get,set}) => {get,set,init}`	`{get,set}`
Field	`() => (initValue)=>initValue2`	`{get,set}`

函数中 `this` 的值：

this 是 →	undefined	Class	Instance
Decorator function	✓		
Static initializer		✓	
Non-static initializer			✓
Static field decorator result		✓	
Non-static field decorator result			✓

3 装饰器语法与语义的更多信息（可选章节）

(本章节为可选内容。如果您跳过此章节，仍然可以理解后续内容。)

3.1 装饰器表达式的语法

- 装饰器表达式以一个或多个标识符的链开始，标识符之间用点分隔。除第一个标识符外，每个标识符都可以是私有的（前缀为 `#`）。不允许使用方括号 `[]`。
- 可选：末尾可以加上括号中的函数调用参数。下一小节将解释这意味着什么。
- 如果我们把它放在括号中，我们可以使用任何表达式：

```
@(«expr»)
```

无论装饰器允许在哪里，我们都可以使用多个装饰器。以下代码演示了装饰器语法

```
// 五个 MyClass 的装饰器

@myFunc
@myFuncFactory('arg1', 'arg2')

@libraryModule.prop
@someObj.method(123)

@(wrap(dict['prop'])) // 任意表达式

class MyClass {}
```

3.2 装饰器如何执行?

- 评估：`@` 符号后面的表达式在类定义的执行过程中被评估，与计算属性键和静态字段一起（见代码）。结果必须是函数。它们被存储在临时位置（可以认为是局部变量），以便稍后调用。

- 调用：装饰器函数在类定义的执行过程中稍后被调用，在方法被评估后，但在构造函数和原型被组装之前。结果再次存储在临时位置。

- 应用：在所有装饰器函数被调用后，使用它们的结果，这可能会影响构造函数和原型。类装饰器在所有方法和字段装饰器之后应用。

以下代码说明了装饰器表达式、计算属性键和字段初始化器的评估顺序：

```
function decorate(str) {
  console.log(`EVALUATE @decorate(): ${str}`);
  return () => console.log(`APPLY @decorate(): ${str}`); // (A)
}
function log(str) {
  console.log(str);
  return str;
}

@decorate('class')
class TheClass {

  @decorate('static field')
  static staticField = log('static field value');

  @decorate('prototype method')
  [log('computed key')]() {}

  @decorate('instance field')
  instanceField = log('instance field value');
    // 仅在我们实例化类时，此初始化器才会运行
}

// Output:
// EVALUATE @decorate(): class
// EVALUATE @decorate(): static field
// EVALUATE @decorate(): prototype method
// computed key
// EVALUATE @decorate(): instance field
// APPLY @decorate(): prototype method
// APPLY @decorate(): static field
// APPLY @decorate(): instance field
// APPLY @decorate(): class
// static field value
```

函数 `decorate` 在每次评估 `decorate()` 表达式时被调用。在 A行，它返回实际的装饰器函数，该函数稍后会被应用。

3.3 装饰器初始化器何时运行?

装饰器初始化器的运行时间取决于装饰器的种类：

- 类装饰器初始化器在类完全定义并且所有静态字段初始化后运行。
- 非静态类元素装饰器的初始化器在实例化期间运行，在实例字段初始化之前。

- 静态类元素装饰器的初始化器在类定义期间运行，在静态字段定义之前，但在所有其他类元素定义之后。

为什么会这样？对于非静态初始化器，我们有五个选项 – 它们可以在以下时机运行：

1. 在 `super` 之前
2. 在 `super` 之后，字段初始化之前
3. 按定义顺序交错在字段之间
4. 在字段初始化之后，子类实例化之前
5. 在子类实例化之后

为什么选择了第 2 种方案？

- 第 1 种方案被拒绝，因为装饰器初始化器必须能够访问 `this`，而在 `super` 运行之前无法做到这一点。
- 第 3 种方案被拒绝，因为同时运行所有装饰器初始化器比确保它们正确交错要简单。
- 第 4 种方案被拒绝，因为在字段之前运行装饰器初始化器可以确保字段在被装饰方法完全初始化之前不会被访问。例如，如果有 `@bind` 装饰器，则字段初始化器可以依赖于被装饰的方法已被绑定。
- 第 5 种方案被拒绝，因为它允许超类干扰子类，这会破坏超类不应意识到其子类的规则。

以下代码演示了 Babel 当前如何调用装饰器初始化器。请注意，Babel 尚不支持类字段装饰器的初始化器（这是对装饰器 API 的最近更改）。

```
// 我们在记录步骤时等待实例化后，
// 这样我们就可以将`this`的值与实例进行比较。
const steps = [];
function push(msg, _this) {
  steps.push({msg, _this});
}
function pushStr(str) {
  steps.push(str);
}

function init(_value, {name, addInitializer}) {
  pushStr(`@init ${name}`);
  if (addInitializer) {
    addInitializer(function () {
      push(`DECORATOR INITIALIZER ${name}`, this);
    });
  }
}

@init class TheClass {
```

```
  //--- Static ---

  static {
    pushStr('static block');
  }

  @init static staticMethod() {}
  @init static accessor staticAcc = pushStr('staticAcc');
  @init static staticField = pushStr('staticField');

  //--- Non-static ---

  @init prototypeMethod() {}
  @init accessor instanceAcc = pushStr('instanceAcc');
  @init instanceField = pushStr('instanceField');

  constructor() {
    pushStr('constructor');
  }
}

pushStr('===== 实例化 =====');
const inst = new TheClass();

for (const step of steps) {
  if (typeof step === 'string') {
    console.log(step);
    continue;
  }
  let thisDesc = '???';
  if (step._this === TheClass) {
    thisDesc = TheClass.name;
  } else if (step._this === inst) {
    thisDesc = 'inst';
  } else if (step._this === undefined) {
    thisDesc = 'undefined';
  }
  console.log(`${step.msg} (this===${thisDesc})`);
}

// Output:
// @init staticMethod
// @init staticAcc
// @init prototypeMethod
// @init instanceAcc
// @init staticField
// @init instanceField
// @init TheClass
// DECORATOR INITIALIZER staticMethod (this===TheClass)
// DECORATOR INITIALIZER staticAcc (this===TheClass)
// static block
```

```
// staticAcc
// staticField
// DECORATOR INITIALIZER TheClass (this===TheClass)
// ===== Instantiation =====
// DECORATOR INITIALIZER prototypeMethod (this===inst)
// DECORATOR INITIALIZER instanceAcc (this===inst)
// instanceAcc
// instanceField
// constructor
```

4 装饰器暴露数据的技巧

有时装饰器会收集数据。让我们探索它们如何将这些数据提供给其他方。

4.1 在外部作用域存储暴露的数据

最简单的解决方案是将数据存储在外部作用域的某个位置。例如，装饰器 `@collect` 收集类并将它们存储在 Set `classes` 中（A 行）：

```
const classes = new Set(); // (A)

function collect(value, {kind, addInitializer}) {
  if (kind === 'class') {
    classes.add(value);
  }
}

@collect
class A {}
@collect
class B {}
@collect
class C {}

assert.deepEqual(
  classes, new Set([A, B, C])
);
```

此方法的缺点是，如果装饰器来自另一个模块，则此方法无效。

4.2 通过工厂函数管理暴露的数据

一种更复杂的方法是使用工厂函数 `createClassCollector()`，该函数返回：

- 一个类装饰器 `collect`
- 一个 Set `classes`，装饰器将把收集到的类添加到其中

```
function createClassCollector() {
  const classes = new Set();
```

```
  function collect(value, {kind, addInitializer}) {
    if (kind === 'class') {
      classes.add(value);
    }
  }
  return {
    classes,
    collect,
  };
}

const {classes, collect} = createClassCollector();

@collect
class A {}
@collect
class B {}
@collect
class C {}

assert.deepEqual(
  classes, new Set([A, B, C])
);
```

4.3 通过类管理暴露的数据

我们也可以使用类来代替工厂函数。它有两个成员：

- `.classes`，一个包含收集到的类的 Set
- `.install`，一个类装饰器

```
class ClassCollector {
  classes = new Set();
  install = (value, {kind}) => { // (A)
    if (kind === 'class') {
      this.classes.add(value); // (B)
    }
  };
}

const collector = new ClassCollector();

@collector.install
class A {}
@collector.install
class B {}
@collector.install
class C {}

assert.deepEqual(
```

```
    collector.classes, new Set([A, B, C])
  );
```

我们通过将一个箭头函数分配给公共实例字段来实现 `.install`（A 行）。实例字段初始化器在 `this` 指向当前实例的作用域中运行。这也是箭头函数的外部作用域，解释了为什么 `this` 在 B 行具有该值。

我们也可以通过 getter 来实现 `.install`，但那样的话，每次读取 `.install` 时都必须返回一个新函数。

5 类装饰器

类装饰器具有以下类型签名：

```
type ClassDecorator = (
  value: Function,
  context: {
    kind: 'class';
    name: string | undefined;
    addInitializer(initializer: () => void): void;
  }
) => Function | void;
```

类装饰器的能力：

- 它可以通过改变 `value` 来改变被装饰的类。
- 它可以通过返回可调用的值来替换被装饰的类。
- 它可以注册初始化器，初始化器在被装饰的类完全设置后调用。
- 由于类不是其他语言结构的成员（而方法是类的成员），因此它不获取 `context.access`。

5.1 示例：收集实例

在下面的示例中，我们使用装饰器收集所有被装饰类的实例：

```
class InstanceCollector {
  instances = new Set();
  install = (value, {kind}) => {
    if (kind === 'class') {
      const _this = this;
      return function (...args) { // (A)
        const inst = new value(...args); // (B)
        _this.instances.add(inst);
        return inst;
      };
    }
  };
}
```

```
const collector = new InstanceCollector();

@collector.install
class MyClass {}

const inst1 = new MyClass();
const inst2 = new MyClass();
const inst3 = new MyClass();

assert.deepEqual(
  collector.instances, new Set([inst1, inst2, inst3])
);
```

我们通过将一个函数包装在 `.install` 中来实现收集实例。该函数在实例化时被调用，并将新创建的实例添加到收集器的实例集合中。

请注意，我们不能在 A 行返回一个箭头函数，因为箭头函数不能被 new 调用。

此方法的一个缺点是，它破坏了 `instanceof`：

```
assert.equal(
  inst1 instanceof MyClass,
  false
);
```

下一小节将解释我们如何修复这个问题。

5.2 确保 `instanceof` 正常工作

在本节中，我们将使用简单的装饰器 `@countInstances` 来演示如何使被包装的类支持 `instanceof`。

5.2.1 通过 `.prototype` 启用 `instanceof`

启用 `instanceof` 的一种方法是将包装函数的 `.prototype` 设置为被包装的 `value` 的 `.prototype`（A 行）：

```
function countInstances(value) {
  const _this = this;
  let instanceCount = 0;
  // 包装器必须是可通过 new 调用的
  const wrapper = function (...args) {
    instanceCount++;
    const instance = new value(...args);
    // 修改实例
    instance.count = instanceCount;
    return instance;
  };
  wrapper.prototype = value.prototype; // (A)
```

```
  return wrapper;
}

@countInstances
class MyClass {}

const inst1 = new MyClass();
assert.ok(inst1 instanceof MyClass);
assert.equal(inst1.count, 1);

const inst2 = new MyClass();
assert.ok(inst2 instanceof MyClass);
assert.equal(inst2.count, 2);
```

这是有效的，原因如下：

```
inst instanceof C
C.prototype.isPrototypeOf(inst)
```

有关 `instanceof` 的更多信息，请参见《探索 JavaScript》。

5.2.2 通过 `Symbol.hasInstance` 启用 `instanceof`

启用 `instanceof` 的另一种选择是给包装函数一个键为 `Symbol.hasInstance` 的方法（A 行）：

```
function countInstances(value) {
  const _this = this;
  let instanceCount = 0;
  // 包装器必须是可通过 new 调用的
  const wrapper = function (...args) {
    instanceCount++;
    const instance = new value(...args);
    // 修改实例
    instance.count = instanceCount;
    return instance;
  };
  // 属性是只读的，因此我们不能使用赋值
  Object.defineProperty( // (A)
    wrapper, Symbol.hasInstance,
    {
      value: function (x) {
        return x instanceof value;
      }
    }
  );
  return wrapper;
}

@countInstances
class MyClass {}
```

```
const inst1 = new MyClass();
assert.ok(inst1 instanceof MyClass);
assert.equal(inst1.count, 1);

const inst2 = new MyClass();
assert.ok(inst2 instanceof MyClass);
assert.equal(inst2.count, 2);
```

《探索 JavaScript》 中有关于 `Symbol.hasInstance` 的更多信息。

5.2.3 通过子类化启用 `instanceof`

我们还可以通过返回 `value` 的子类来启用 `instanceof`（A 行）：

```
function countInstances(value) {
  const _this = this;
  let instanceCount = 0;
  // 包装器必须是可通过 new 调用的
  return class extends value { // (A)
    constructor(...args) {
      super(...args);
      instanceCount++;
      // 修改实例
      this.count = instanceCount;
    }
  };
}

@countInstances
class MyClass {}

const inst1 = new MyClass();
assert.ok(inst1 instanceof MyClass);
assert.equal(inst1.count, 1);

const inst2 = new MyClass();
assert.ok(inst2 instanceof MyClass);
assert.equal(inst2.count, 2);
```

5.3 示例：冻结实例

装饰器类 `@freeze` 通过冻结它所装饰的类生成的所有实例来实现：

```
function freeze (value, {kind}) {
  if (kind === 'class') {
    return function (...args) {
      const inst = new value(...args);
      return Object.freeze(inst);
    }
```

```
  }
}

@freeze
class Color {
  constructor(name) {
    this.name = name;
  }
}

const red = new Color('red');
assert.throws(
  () => red.name = 'green',
  /^TypeError: Cannot assign to read only property 'name'/
);
```

此装饰器有以下缺点：

- 它破坏了 `instanceof`。我们已经看到如何修复这个问题。
- 对装饰类的子类化效果不佳：
  - 构造函数的连接方式并不理想，因为混合了包装构造函数。这可以通过返回被装饰的 `value` 的子类来部分修复。
  - 子类无法设置属性，因为它们的 `this` 是不可变的。无法避免此缺点。

最后一个缺点可以通过在所有构造函数执行后让类装饰器访问装饰类的实例来避免。

这将改变继承的工作方式，因为超类现在可以更改由子类添加的属性。因此，未来是否会有这样的机制尚不确定。

5.4 示例：让类可函数调用

被 `@functionCallable` 装饰的类可以通过函数调用而不是 `new` 操作符来调用：

```
function functionCallable(value, {kind}) {
  if (kind === 'class') {
    return function (...args) {
      if (new.target !== undefined) {
        throw new TypeError('This function can’t be new-invoked');
      }
      return new value(...args);
    }
  }
}

@functionCallable
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```
const robin = Person('Robin');
assert.equal(
  robin.name, 'Robin'
);
```

6 类方法装饰器

类方法装饰器具有以下类型签名：

```
type ClassMethodDecorator = (
  value: Function,
  context: {
    kind: 'method';
    name: string | symbol;
    static: boolean;
    private: boolean;
    access: { get: () => unknown };
    addInitializer(initializer: () => void): void;
  }
) => Function | void;
```

方法装饰器的能力：

- 它可以通过改变 `value` 来改变被装饰的方法。
- 它可以通过返回一个函数来替换被装饰的方法。
- 它可以注册初始化器。
- `context.access` 仅支持获取其属性的值，而不支持设置。

构造函数不能被装饰：它们看起来像方法，但实际上并不是真正的方法。

6.1 示例：追踪方法调用

装饰器 `@trace` 包装方法，以便将它们的调用及结果记录到控制台：

```
function trace(value, {kind, name}) {
  if (kind === 'method') {
    return function (...args) {
      console.log(`CALL ${name}: ${JSON.stringify(args)}`);
      const result = value.apply(this, args);
      console.log('=> ' + JSON.stringify(result));
      return result;
    };
  }
}

class StringBuilder {
  #str = '';
  @trace
  add(str) {
```

```
    this.#str += str;
  }
  @trace
  toString() {
    return this.#str;
  }
}

const sb = new StringBuilder();
sb.add('Home');
sb.add('page');
assert.equal(
  sb.toString(), 'Homepage'
);

// Output:
// CALL add: ["Home"]
// => undefined
// CALL add: ["page"]
// => undefined
// CALL toString: []
// => "Homepage"
```

6.2 示例：将方法绑定到实例

通常，提取方法（A 行）意味着我们无法函数调用它们，因为这会将 `this` 设置为 `undefined`：

```
class Color1 {
  #name;
  constructor(name) {
    this.#name = name;
  }
  toString() {
    return `Color(${this.#name})`;
  }
}

const green1 = new Color1('green');
const toString1 = green1.toString; // (A)
assert.throws(
  () => toString1(),
  /^TypeError: Cannot read properties of undefined/
);
```

我们可以通过装饰器 `@bind` 修复这个问题：

```
function bind(value, {kind, name, addInitializer}) {
  if (kind === 'method') {
    addInitializer(function () { // (B)
      this[name] = value.bind(this); // (C)
```

```
    });
  }
}

class Color2 {
  #name;
  constructor(name) {
    this.#name = name;
  }
  @bind
  toString() {
    return `Color(${this.#name})`;
  }
}

const green2 = new Color2('green');
const toString2 = green2.toString;
assert.equal(
  toString2(), 'Color(green)'
);

// 自有属性 green2.toString 与
// Color2.prototype.toString 不同
assert.ok(Object.hasOwn(green2, 'toString'));
assert.notEqual(
  green2.toString,
  Color2.prototype.toString
);
```

通过将 A 行的初始化器注册为普通函数，我们可以在实例化时访问 `this`。这也解释了为什么 B 行的 `this` 具有该值。(line C)

6.3 示例：对方法应用函数

库 `core-decorators` 提供了一个装饰器，允许我们对方法应用函数。这使我们能够使用像 Lodash 的 `memoize()` 这样的辅助函数。以下代码展示了装饰器 `@applyFunction` 的实现：

```
import { memoize } from 'lodash-es';

function applyFunction(functionFactory) {
  return (value, {kind}) => { // 装饰器函数
    if (kind === 'method') {
      return functionFactory(value);
    }
  };
}

let invocationCount = 0;
```

```
class Task {
  @applyFunction(memoize)
  expensiveOperation(str) {
    invocationCount++;
    // str` 的昂贵开销 😀
    return str + str;
  }
}

const task = new Task();
assert.equal(
  task.expensiveOperation('abc'),
  'abcabc'
);
assert.equal(
  task.expensiveOperation('abc'),
  'abcabc'
);
assert.equal(
  invocationCount, 1
);
```

7 类 getter 装饰器、setter 装饰器

这些是 getter 装饰器和 setter 装饰器的类型签名：

```
type ClassGetterDecorator = (
  value: Function,
  context: {
    kind: 'getter';
    name: string | symbol;
    static: boolean;
    private: boolean;
    access: { get: () => unknown };
    addInitializer(initializer: () => void): void;
  }
) => Function | void;

type ClassSetterDecorator = (
  value: Function,
  context: {
    kind: 'setter';
    name: string | symbol;
    static: boolean;
    private: boolean;
    access: { set: (value: unknown) => void };
    addInitializer(initializer: () => void): void;
  }
) => Function | void;
```

getter 装饰器和 setter 装饰器的能力类似于方法装饰器。

7.1 示例：惰性计算值

惰性计算属性值的实现需要两种技术：

- 我们通过 getter 实现该属性。这样，计算其值的代码仅在读取属性时执行。
- 装饰器 `@lazy` 包装原始 getter：当包装器第一次被调用时，它调用 getter 并创建一个自身数据属性，其值为计算结果。从此以后，无论何时有人读取该属性，自身属性都会覆盖继承的 getter。

```
class C {
  @lazy
  get value() {
    console.log('COMPUTING');
    return 'Result of computation';
  }
}

function lazy(value, {kind, name, addInitializer}) {
  if (kind === 'getter') {
    return function () {
      const result = value.call(this);
      Object.defineProperty( // (A)
        this, name,
        {
          value: result,
          writable: false,
        }
      );
      return result;
    };
  }
}

console.log('1 new C()');
const inst = new C();
console.log('2 inst.value');
assert.equal(inst.value, 'Result of computation');
console.log('3 inst.value');
assert.equal(inst.value, 'Result of computation');
console.log('4 end');

// Output:
// 1 new C()
// 2 inst.value
// COMPUTING
// 3 inst.value
// 4 end
```

请注意，属性 `.[name]` 是不可变的（因为只有一个 getter），这就是为什么我们必须在 A 行定义该属性，而不能使用赋值的原因。

8 类字段装饰器

类字段装饰器具有以下类型签名：

```
type ClassFieldDecorator = (
  value: undefined,
  context: {
    kind: 'field';
    name: string | symbol;
    static: boolean;
    private: boolean;
    access: { get: () => unknown, set: (value: unknown) => void };
    addInitializer(initializer: () => void): void;
  }
) => (initialValue: unknown) => unknown | void;
```

字段装饰器的能力：

- 它不能改变或替换它的字段。如果我们需要该功能，我们必须使用 *自动访问器*（稍后将解释的内容）。
- 通过返回一个接收原始初始化值并返回新初始化值的函数来改变字段的初始化值。
    - 在该函数内部，`this` 指向当前实例。
- 注册初始化器。这是装饰器 API 的一个最近更改（2022-03 之后），之前是不可行的。
- 它可以通过 `context.access` 暴露对其字段的访问（即使是私有字段）。

8.1 示例：改变字段初始化值

装饰器 `@twice` 通过返回一个函数来改变字段的原始初始化值，该函数接收原始值并返回新值：

```
function twice() {
  return initialValue => initialValue * 2;
}

class C {
  @twice
  field = 3;
}

const inst = new C();
```

```
assert.equal(
  inst.field, 6
);
```

8.2 示例：只读字段（实例公有字段）

装饰器 `@readOnly` 通过在字段完全设置后（通过赋值或构造函数）使其不可变来实现。

```
const readOnlyFieldKeys = Symbol('readOnlyFieldKeys');

@readOnly
class Color {
  @readOnly
  name;
  constructor(name) {
    this.name = name;
  }
}

const blue = new Color('blue');
assert.equal(blue.name, 'blue');
assert.throws(
  () => blue.name = 'brown',
  /^TypeError: Cannot assign to read only property 'name'/
);

function readOnly(value, {kind, name}) {
  if (kind === 'field') { // (A)
    return function () {
      if (!this[readOnlyFieldKeys]) {
        this[readOnlyFieldKeys] = [];
      }
      this[readOnlyFieldKeys].push(name);
    };
  }
  if (kind === 'class') { // (B)
    return function (...args) {
      const inst = new value(...args);
      for (const key of inst[readOnlyFieldKeys]) {
        Object.defineProperty(inst, key, {writable: false});
      }
      return inst;
    }
  }
}
```

我们需要两步来让 `@readOnly` 生效(这也是类也被装饰了的原因):

- 我们首先收集所有要变成只读的字段 (line A).

- 等实例已经创建完成，我们再把收集的字段设为不可写(line B)。我们需要把类包起来因为装饰器初始化器执行得太早了。

和不可变类似，这个装饰器也会破坏`instanceof`的行为。 变通方法也是一样的。

我们一会会看到， `@readOnly`能够和auto-accessor互动而非和字段互动的版本。而且使用过程不需要装饰类。

8.3 示例：依赖注入（实例public字段）

依赖注入（Dependency Injection） 是由以下观察推动的：如果我们提供类的构造函数其依赖项（而不是构造函数自己设置它们），那么就更容易适应不同的环境，包括测试。

这是一种*控制反转（Inversion of Control）*：构造函数本身不执行初始化操作，而是由我们代为完成。以下是实现依赖注入的几种方法：

1. 手动创建依赖项并传递给构造函数
2. 通过前端框架（如React）中的"contexts"实现
3. 通过装饰器和*依赖注入注册表*（*依赖注入容器*的一种变体）实现

以下代码是方法 #3 的一个简单实现：

```
const {registry, inject} = createRegistry();

class Logger {
  log(str) {
    console.log(str);
  }
}
class Main {
  @inject logger;
  run() {
    this.logger.log('Hello!');
  }
}

registry.register('logger', Logger);
new Main().run();

// Output:
// Hello!
```

这就是 `createRegistry()` 的实装方法:

```
function createRegistry() {
  const nameToClass = new Map();
  const nameToInstance = new Map();
  const registry = {
    register(name, componentClass) {
```

```
      nameToClass.set(name, componentClass);
    },
    getInstance(name) {
      if (nameToInstance.has(name)) {
        return nameToInstance.get(name);
      }
      const componentClass = nameToClass.get(name);
      if (componentClass === undefined) {
        throw new Error('Unknown component name: ' + name);
      }
      const inst = new componentClass();
      nameToInstance.set(name, inst);
      return inst;
    },
  };
  function inject (_value, {kind, name}) {
    if (kind === 'field') {
      return () => registry.getInstance(name);
    }
  }
  return {registry, inject};
}
```

8.4 示例：“友元”可见性（实例私有字段）

我们可以通过将某些类成员设为私有来改变其可见性，从而防止它们被公开访问。不过，还有更多有用的可见性类型。例如，*友元可见性*允许一组*友元*（函数、其他类等）访问该成员。

有多种方式可以指定友元。在以下示例中，所有能访问`friendName`的对象都是`classWithSecret.#name`的友元。其核心思想是：一个模块中包含相互协作的类和函数，而某些实例数据应该仅对这些协作者可见。

```
const friendName = new Friend();

class ClassWithSecret {
  @friendName.install #name = 'Rumpelstiltskin';
  getName() {
    return this.#name;
  }
}

// 能访问 `secret`的，也能访问 inst.#name
const inst = new ClassWithSecret();
assert.equal(
  friendName.get(inst), 'Rumpelstiltskin'
);
friendName.set(inst, 'Joe');
assert.equal(
```

```
  inst.getName(), 'Joe'
);
```

这是Friend类实装的过程：

```
class Friend {
  #access = undefined;
  #getAccessOrThrow() {
    if (this.#access === undefined) {
      throw new Error('The friend decorator wasn't used yet');
    }
    return this.#access;
  }
  // 一个`this`已经被绑定好了的类属性
  // （绑定到了这个实例上）
  install = (_value, {kind, access}) => {
    if (kind === 'field') {
      if (this.#access) {
        throw new Error('This decorator can only be used once');
      }
      this.#access = access;
    }
  }
  get(inst) {
    return this.#getAccessOrThrow().get.call(inst);
  }
  set(inst, value) {
    return this.#getAccessOrThrow().set.call(inst, value);
  }
}
```

8.5 示例：枚举（静态公有字段）

实现枚举的方式有很多种。一种面向对象风格的实现方式是使用类和静态属性（关于此方法的更多信息）：

```
class Color {
  static red = new Color('red');
  static green = new Color('green');
  static blue = new Color('blue');
  constructor(enumKey) {
    this.enumKey = enumKey;
  }
  toString() {
    return `Color(${this.enumKey})`;
  }
}
assert.equal(
  Color.green.toString(),
  'Color(green)'
);
```

我们可以使用装饰器自动实现以下功能：

- 创建一个从"枚举键"（字段名称）到枚举值的映射(Map)
- 将枚举键添加到枚举值中 - 无需通过构造函数传递

实现方式如下所示：

```
function enumEntry(value, {kind, name}) {
  if (kind === 'field') {
    return function (initialValue) {
      if (!Object.hasOwn(this, 'enumFields')) {
        this.enumFields = new Map();
      }
      this.enumFields.set(name, initialValue);
      initialValue.enumKey = name;
      return initialValue;
    };
  }
}

class Color {
  @enumEntry static red = new Color();
  @enumEntry static green = new Color();
  @enumEntry static blue = new Color();
  toString() {
    return `Color(${this.enumKey})`;
  }
}
assert.equal(
  Color.green.toString(),
  'Color(green)'
);
assert.deepEqual(
  Color.enumFields,
  new Map([
    ['red', Color.red],
    ['green', Color.green],
    ['blue', Color.blue],
  ])
);
```

9 自动访问器：类定义的新成员

装饰器提案引入了一项新的语言特性：*自动访问器(auto-accessors)*。通过在类字段前添加`accessor`关键字即可创建自动访问器。它使用起来像普通字段，但在运行时采用不同的实现方式。这将有助于装饰器的使用，我们稍后会看到。以下是自动访问器的示例：

```
class C {
  static accessor myField1;
```

```
  static accessor #myField2;
  accessor myField3;
  accessor #myField4;
}
```

字段和自动访问器有何区别?

- 字段会创建以下两种结构之一:
  - 属性（静态或实例）
  - 私有槽位（静态或实例）
- 自动访问器会为数据创建一个私有槽位（静态或实例），并同时创建:
  - 公共的getter-setter对（静态或原型）
  - 私有的getter-setter对（静态或实例）
    - 私有槽位不可继承，因此永远不会存在于原型中

请看以下类示例:

```
class C {
  accessor str = 'abc';
}
const inst = new C();
assert.equal(
  inst.str, 'abc'
);
inst.str = 'def';
assert.equal(
  inst.str, 'def'
);
```

其内部实现如下所示:

```
class C {
  #str = 'abc';
  get str() {
    return this.#str;
  }
  set str(value) {
    this.#str = value;
  }
}
```

以下代码展示了自动访问器的getter和setter所在的位置:

```
class C {
  static accessor myField1;
  static accessor #myField2;
  accessor myField3;
  accessor #myField4;

  static {
    // Static getter and setter
```

```
    assert.ok(
      Object.hasOwn(C, 'myField1'), 'myField1'
    );
    // Static getter and setter
    assert.ok(
      #myField2 in C, '#myField2'
    );

    // Prototype getter and setter
    assert.ok(
      Object.hasOwn(C.prototype, 'myField3'), 'myField3'
    );
    // Private getter and setter
    // （保存在实例上，但在各个实例之间共享）
    assert.ok(
      #myField4 in new C(), '#myField4'
    );
  }
}
```

有关私有getter、私有setter和私有方法的槽位为何存储在实例中的更多信息，请参阅《探索JavaScript》中的"私有方法和访问器"章节。

9.1 为什么需要自动访问器?

自动访问器是装饰器所必需的:

- 装饰器只能影响字段的初始值
- 但可以完全替换自动访问器

因此，当装饰器需要比字段更多的控制权时，我们必须使用自动访问器而非字段。

10 类自动访问器装饰器（Class auto-accessor）

类自动访问器装饰器具有以下类型签名:

```
type ClassAutoAccessorDecorator = (
  value: {
    get: () => unknown;
    set: (value: unknown) => void;
  },
  context: {
    kind: 'accessor';
    name: string | symbol;
    static: boolean;
    private: boolean;
    access: { get: () => unknown, set: (value: unknown) => void };
    addInitializer(initializer: () => void): void;
  }
) => {
```

```
  get?: () => unknown;
  set?: (value: unknown) => void;
  init?: (initialValue: unknown) => unknown;
} | void;
```

自动访问器装饰器的能力：

- 通过参数`value`接收自动访问器的getter和setter
  - `context.access`提供相同的功能
- 可以通过返回包含`.get()`和/或`.set()`方法的对象来替换被装饰的自动访问器
- 可以通过返回包含`.init()`方法的对象来影响自动访问器的初始值
- 可以注册初始化器

10.1 示例：只读自动访问器

我们已经实现了字段的装饰器`@readOnly`。现在为自动访问器实现同样的功能：

```
const UNINITIALIZED = Symbol('UNINITIALIZED');
function readOnly({get,set}, {name, kind}) {
  if (kind === 'accessor') {
    return {
      init() {
        return UNINITIALIZED;
      },
      get() {
        const value = get.call(this);
        if (value === UNINITIALIZED) {
          throw new TypeError(
            `Accessor ${name} hasn't been initialized yet`
          );
        }
        return value;
      },
      set(newValue) {
        const oldValue = get.call(this);
        if (oldValue !== UNINITIALIZED) {
          throw new TypeError(
            `Accessor ${name} can only be set once`
          );
        }
        set.call(this, newValue);
      },
    };
  }
}

class Color {
  @readOnly
  accessor name;
  constructor(name) {
```

```
    this.name = name;
  }
}

const blue = new Color('blue');
assert.equal(blue.name, 'blue');
assert.throws(
  () => blue.name = 'yellow',
  /^TypeError: Accessor name can only be set once$/
);

const orange = new Color('orange');
assert.equal(orange.name, 'orange');
```

与字段版本相比，此装饰器有一个显著的优点：它不需要包装类即可确保被装饰的结构变为只读。

11 常见问题

11.1 为什么函数不能被装饰?

当前提案是以“类”为出发点的。函数表达式的装饰器提案已经提出。 然而，自那以后进展不大，并且尚未有函数 *声明* 的提案。

另一方面，装饰函数相对简单：

```
const decoratedFunc = decorator((x, y) => {});
```

甚至 管道运算符提案看起来更好：

```
const decoratedFunc = (x, y) => {} |> decorator(%);
```

12 More decorator-related proposals

以下这些ECMAScript提案，提供了更多装饰器相关的特性：

- Stage 2: “Decorator Metadata” 作者 Chris Garrett (最近更新：2022-04-11)
  - “本提案旨在扩展装饰器提案，增加装饰器与被装饰的值的元数据的互动。”
- Stage 0: “Function Expression Decorators” 作者 Igor Minar (最近更新：2016-01-25)
- Stage 0: “Method Parameter Decorators” 作者 Igor Minar (最近更新：2016-01-25)

13 资源

13.1 实现

- Babel 目前通过 `@babel/plugin-proposal-decorators` 对第 3 阶段装饰器提供了最佳支持。
  - 确保选择 最新的装饰器版本。
  - 本文中所有代码均通过 Babel 开发。
- TypeScript 目前在一个标志后面支持第 1 阶段装饰器。
  - 有 Ron Buckton 的一个 PR 支持第 3 阶段装饰器，并可能在 TypeScript 4.9 之后的版本中发布。

13.2 带装饰器的库

这些是带装饰器的库。目前它们仅支持第 1 阶段装饰器，但可以作为可能性的灵感：

- core-decorators.js by Jay Phelps (targets Babel)
- “Helpful Decorators For TypeScript Projects” by Netanel Basal

14 致谢

- 感谢 Chris Garrett 解答我关于装饰器的问题。

15 延伸阅读

- 章节：Callables [普通函数、箭头函数、类、方法] 在《探索 JavaScript》中
- 章节：Classes 在《探索 JavaScript》中