Decorators

装饰器

JS新特性产生流程

首先看一下目前标准流程是 5 个阶段,Stage0 ~ Stage 4

- Stage0: 稻草人(Strawpersion),由TC39成员发起,通常是提出新想法或是对未纳入正式的提案进行修改。
- Stage1:提案(Proposal),提出一些具体的问题和解决方案。
- Stage2:草稿(Draft),用ES语法尽可能精确地描述提案的语法、语义和API,并提供实验性的实现。意味着提案会有很大概率出现在正式版本的中。
- Stage3:候选人(Candidate),到了该阶段,提案基本已经定型,仅根据外部反馈针对关键问题进行更改。
- Stage4:完成(Finish),该提案会出现在正式的规范文档中,并在下一个版本的ES中正式支持。

Introduction

Decorators 是对类、类元素或其他 JavaScript 语法形式在定义期间调用的函数。

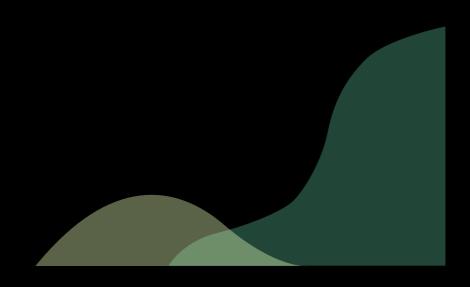
```
@defineElement("my-class")
class C extends HTMLElement {
    @reactive accessor clicked = false;
    @callOnRender fetchDate(){
        // todo
    }
}
```

装饰器的三种能力

■ 替换:将所修饰的元素替换成其他值(用其他方法替换所修饰的方法,用其他属性替换所修饰的属性等等);

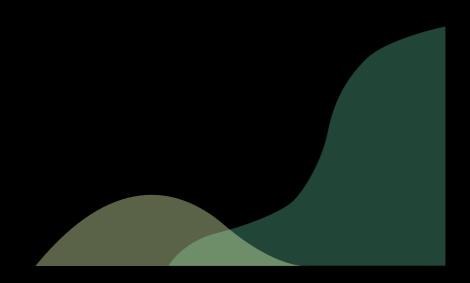
■ 访问:通过访问器来访问所修饰元素的能力;

■ 初始化:初始化所修饰的元素。



装饰器的四种类型

- Classes
- Class fields (public, private, and static)
- Class methods (public, private, and static)
- Class accessors (public, private, and static)



详细设计

- 1. Decorator 标识符 (在@后面的名字) 被放在被修饰的类、field、method 之前
- 2. 装饰器在类的定义过程中被调用(作为函数),在方法被声明之后,但在构造函数和原型被组合起来之前。
- 3. 类装饰器在其他所有装饰器被调用后调用



调用顺序

Evaluating decorators

同类的装饰器从上到下,从左到右开始调用,自定义访问装饰器和get、set以及方法装饰器一起排序,然后是属性装饰器,然后是类装饰器。

同一个属性的装饰器从下到上调用。

```
@step('5')
@step('4')
name:string='Step'
```

类似于

```
step('5')(step('4')(name))
```

调用装饰器

当装饰器被调用时,它们接收两个参数。

- 1. 被装饰的值,或者在类字段的情况下是 undefined。
- 2. 一个包含被装饰的值的上下文对象

ts 类型定义如下

```
type Decorator = (value: Input, context
  kind: string;
  name: string | symbol;
  access: {
    get?(): unknown;
    set?(value: unknown): void;
  };
  private?: boolean;
  static?: boolean;
  addInitializer?(initializer: () => vo
}) => Output | void;
```

Input和Output代表了传递给特定装饰器和从其返回的 值。所有的装饰器都可以选择不返回任何东西,默认使

ts 类型定义如下

```
type Decorator = (value: Input, context
 kind: string;
 name: string | symbol;
 access: {
   get?(): unknown;
    set?(value: unknown): void;
 };
 private?: boolean;
 static?: boolean;
 addInitializer?(initializer: () => vc ■ addInitializer: 允许用户添加额外的初始化逻辑
}) => Output | void;
```

Input和Output代表了传递给特定装饰器和从其返回的 值。所有的装饰器都可以选择不返回任何东西, 默认使 上下文对象也根据被装饰的值而变化:

- kind:被装饰的值的种类,包含这些值:"class"
- name:值的名称,如果是私有元素,则是对它的描 述(例如,可读的名称)。
- access:一个包含访问该值的方法的对象。
- static:是否为静态类元素。只适用于类元素
- private:是否是一个私有的类元素。只适用于类 元素。

类定义装饰器

定义

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

类装饰器接收被装饰的类作为第一个参数,并可以选择 返回一个新的可调用值(一个类、函数或代理)来替代 它。如果返回的是一个不可调用的值,那么就会抛出一 个错误。

```
function logged(value, { kind, name })
  if (kind === "class") {
    return class extends value {
      constructor(...args) {
        super(...args);
        console.log(`constructing an in
```

类方法装饰器

定义

```
type ClassMethodDecorator = (value: Funkind: "method";
  name: string | symbol;
  access: { get(): unknown };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => voi
}) => Function | void;
```

类方法装饰器接收被装饰的方法作为第一个值,并可以选择返回一个新的方法来替换它。如果一个新的方法被返回,它将取代原型上的原方法(如果是静态方法,则取代类本身)。如果返回任何其他类型的值,迄全产生

```
function logged(value, { kind, name })
  if (kind === "method") {
    return function (...args) {
      console.log(`starting ${name} wit
      const ret = value.call(this, ...a
      console.log(`ending ${name}`);
      return ret;
class C {
  01 aaaaa
```

类访问器装饰器

定义

```
type ClassGetterDecorator = (value: Fun
  kind: "getter";
 name: string | symbol;
  access: { get(): unknown };
 static: boolean;
  private: boolean;
  addInitializer(initializer: () => voi
}) => Function | void;
type ClassSetterDecorator = (value: Fun
  kind: "setter";
 name: string | symbol;
       a. [ aat/walua. unknoum). watd l
```

访问器装饰器接收原始的底层getter/setter函数作为第一个值,并且可以选择返回一个新的getter/setter函数来替代它。像方法装饰器一样,这个新的函数被放置在原型上,以取代原来的函数(或者对于静态访问器来说,被放置在类上),如果返回任何其他类型的值,将被抛出一个错误。

```
class C {
    @foo
    get x() {
        // ...
    }
    set x(val) {
```

类属性装饰器

定义

```
type ClassFieldDecorator = (value: unde
   kind: "field";
   name: string | symbol;
   access: { get(): unknown, set(value:
   static: boolean;
   private: boolean;
}) => (initialValue: unknown) => unknown
```

与方法和访问器不同,类字段在被装饰时没有一个直接的输入值。相反,用户可以选择返回一个初始化函数,该函数在字段被分配时运行,接收字段的初始值并返回一个新的初始值。如果除了函数之外的任何其他类型的值被返回。将抛出一个错误

```
function logged(value, { kind, name })
  if (kind === "field") {
   return function (initialValue) {
      console.log(`initializing ${name}
     return initialValue;
   };
 // ...
class C {
  algebra
```

类自动访问器装饰器

类的自动访问器是一种新的结构,通过在类的字段前添加访问器关键字来定义。

```
class C {
  accessor x = 1;
}
```

类方法装饰器接收被装饰的方法作为第一个值,并可以选择返回一个新的方法来替换它。如果一个新的方法被返回,它将取代原型上的原方法(如果是静态方法,则取代类本身)。如果返回任何其他类型的值,将会产生一个错误。

与常规字段不同,自动访问器在类原型上定义 getter 和 setter。这个 getter 和 setter 默认用于获取和设置私有字段上的值。

```
class C {
    #x = 1;

    get x() {
        return this.#x;
    }

    set x(val) {
        this.#x = val;
    }
}
```

类自动访问器装饰器

定义

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

与字段装饰器不同,自动访问器装饰器接收一个值,该值是一个对象,包含在类的原型(或者在静态自动访问器的情况下类本身)上定义的 get 和 set 访问器。然后装饰器可以包装这些属性,并返回一个新的 get 和/或 set,允许装饰器截获对属性的访问。这是一种在字段中不可能实现的功能,但在自动访问器中可以实现。此外,自动访问器可以返回一个 init 函数,该函数可用于更改私有字段中备份值的初始值,类似于字段修饰符。如果返回了一个对象,但没有了任何值,默认使用原始行为。如果返回包含这些属性的对象之外的其他类型的值,将引发错误。

类自动访问器装饰器 示例

```
function logged(value, { kind, name }) {
  let { get, set } = value;
  return {
    get() {
      console.log(`getting ${name}`);
    }
}
```