JS面向对象

Status	Not started
Project	💥 JS
Tags	

构造函数：new创建和直接创建

在JavaScript中，构造函数通常是普通函数，用于通过 `new` 关键字创建新对象时初始化对象的属性。使用 `new` 关键字时，构造函数的行为会有一些特殊的改变：

1. **创建新对象**：JavaScript引擎会创建一个新的空对象。
2. **设置原型链**：新创建的对象的原型（`__proto__`）会被设置为构造函数的 `prototype` 属性。
3. **构造函数内的 `this` 指向新对象**：在构造函数内部，`this` 关键字会指向新创建的对象，允许属性和方法被添加到这个对象上。
4. **自动返回新对象**：除非构造函数显式返回一个非原始值（如另一个对象），否则新创建的对象会在构造函数执行完毕后自动返回。

使用 `new` 调用构造函数

当你使用 `new` 关键字调用一个函数时，该函数就作为构造函数来执行，如上所述。例如：

```
function Person(name) {
    this.name = name;
    this.greet = function() {
        console.log("Hello, my name is " + this.name);
    };
}

var person = new Person("Alice");
person.greet();  // 输出: Hello, my name is Alice
```

不使用 `new` 调用构造函数

如果你调用一个设计为构造函数的函数，但没有使用 `new` 关键字，它就像一个普通函数那样执行。这意味着：

- `this` 通常会指向全局对象（在非严格模式下）或者是 `undefined`（在严格模式下），而不是一个新的空对象。
- 函数可能不会按预期工作，因为它可能依赖于 `this` 指向一个新的对象来设置属性。

例如：

```
function Person(name) {
    this.name = name;
}

var person = Person("Alice");  // 没有使用new
console.log(person);           // 输出: undefined
console.log(name);             // 如果在非严格模式下，可能输出 "Al
ice"，因为name被添加到了全局对象
```

在这个例子中，因为没有使用 `new`，`this.name = name` 实际上将 `name` 添加到了全局对象上，而 `person` 变量是 `undefined`，因为普通函数没有返回值。

结论

虽然构造函数在技术上是普通函数，可以不使用 `new` 调用，但这样做通常会导致错误或不符合预期的行为。为了安全和遵循良好的编程实践，应当总是使用 `new` 关键字来调用构造函数，以确保正确地创建和初始化新对象。

JS的constructor

在JavaScript中，`constructor` 是一个特殊的方法，主要用于在类语法中定义一个类的构造函数。从ES6开始，JavaScript引入了类（class）语法，使得基于原型的对象创建和继承更加直观和易于管理。`constructor` 方法是类内部定义对象初始化的地方，它会在通过 `new` 关键字创建类的新实例时自动调用。

JavaScript中的 `constructor` 使用示例

下面是一个使用 `constructor` 的简单示例：

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hello, my name is ${this.name} and I am
${this.age} years old.`);
    }
}

const person = new Person('Alice', 30);
person.greet();  // 输出: Hello, my name is Alice and I am 30
years old.
```

在这个例子中，`Person` 类有一个 `constructor` 方法，它接受两个参数并将它们设置为实例的属性。`new Person('Alice', 30)` 创建了一个新的 `Person` 实例，并调用 `constructor` 来初始化属性。

为什么 `constructor` 在JavaScript中不那么常用

尽管 `constructor` 在类定义中扮演着核心角色，但在实际的JavaScript编程实践中，其使用频率可能不如其他语言中构造函数的使用频率高，原因包括：

1. **历史原因和习惯**：在ES6之前，JavaScript没有类语法，对象通常通过函数构造器或对象字面量创建。许多JavaScript开发者习惯了这种方式，即使在类语法被引入后，这种习惯也影响了其编码风格。

2. **简单对象的创建**：对于许多简单的用途，使用对象字面量或工厂函数可能更简单和直接。这些方法不需要定义一个类，因此也就不需要 `constructor`。
3. **灵活性和动态性**：JavaScript允许在运行时动态添加或更改对象的属性和方法。在某些情况下，开发者可能选择在对象创建后动态地配置对象，而不是在构造函数中静态定义所有内容。
4. **函数式编程风格的流行**：JavaScript的函数式编程风格支持使用高阶函数和不可变数据，这种风格不鼓励使用传统的基于类的面向对象方法。
5. **框架和库的抽象**：许多现代JavaScript框架和库（如React）提供了自己的抽象来创建组件和对象，这些抽象可能隐藏了直接使用 `constructor` 的需要。

结论

虽然 `constructor` 在定义JavaScript类时是必要的，但由于历史、习惯、编程风格和技术栈的多样性，它在日常JavaScript开发中的直接使用可能没有其他一些语言中构造函数的使用频繁。开发者可能更倾向于使用其他模式和抽象来构造和管理对象，特别是在使用现代JavaScript框架和库的环境中。

JavaScript(JS)和C++在对象构造方面的区别

以下是详细解释JavaScript(JS)和C++在对象构造方面的区别。

1. 语言类型差异:
   - C++是一种静态类型语言,在编译时需要明确指定变量的数据类型。
   - JS是一种动态类型语言,变量的数据类型在运行时确定,无需预先声明。
2. 类和对象的定义方式:
   - 在C++中,使用class关键字定义类,通过构造函数初始化对象。类定义了对象的属性和方法。
   - 在JS中,对象可以直接使用对象字面量({})或构造函数来创建。JS的对象是基于原型(prototype)的。
3. 构造函数的区别:
   - C++的构造函数是一种特殊的成员函数,与类同名,用于初始化对象的成员变量。可以有多个构造函数(构造函数重载)。

- JS的构造函数是一个普通函数,通过new关键字调用,用于创建和初始化对象。JS没有构造函数重载的概念。

4. 对象属性和方法的定义:
   - 在C++中,对象的属性和方法在类中预先定义。通过访问修饰符(如public、private)控制访问权限。
   - 在JS中,对象的属性和方法可以在构造函数内部或外部动态添加。对象的属性可以直接访问和修改。
5. 继承方式:
   - C++支持类的单继承和多继承,通过派生类继承基类的属性和方法。
   - JS通过原型链实现继承。每个对象都有一个原型对象,通过原型链可以访问和继承属性和方法。
6. 内存管理:
   - C++需要手动管理内存,通过new和delete操作符分配和释放内存。
   - JS采用自动内存管理,通过垃圾回收机制自动回收不再使用的内存。

示例:

C++中的类和对象:

```
class Person {
public:
    Person(string name, int age) {
        this->name = name;
        this->age = age;
    }
    void introduce() {
        cout << "My name is " << name << ", I am " << age <<
" years old." << endl;
    }
private:
    string name;
    int age;
};
```

```
int main() {
    Person p("Alice", 20);
    p.introduce();
    return 0;
}
```

JS中的对象:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.introduce = function() {
        console.log("My name is " + this.name + ", I am " + t
his.age + " years old.");
    }
}

var p = new Person("Alice", 20);
p.introduce();
```

这只是JS和C++在对象构造方面的一些主要区别。JS作为一种动态、灵活的语言,与C++相比有独特的对象创建和管理方式。理解它们之间的差异对于前端开发非常重要。

JS为什么要设计原型这个概念

JavaScript(JS)中引入原型(prototype)这个概念有几个重要的原因:

1. 实现对象之间的继承:
   在JS中,原型是实现对象之间继承的主要方式。每个对象都有一个原型对象,通过原型链,对象可以访问和继承其原型对象的属性和方法。这种基于原型的继承模型提供了一种灵活和动态的继承机制,与基于类的继承不同。
2. 共享属性和方法:
   通过将属性和方法定义在原型对象上,可以实现多个对象之间的属性和方法共享。当多个对象共享同一个原型时,它们可以访问原型上定义的属性和方法,而不需要在每个对象上重复定义。这种共享机制可以节省内存空间,提高代码的重用性。

3. 动态扩展对象:
   原型允许在运行时动态地为对象添加或修改属性和方法。通过修改对象的原型,可以为该对象及其所有继承自该原型的对象添加新的属性和方法。这种动态扩展的能力使得JS具有很大的灵活性,可以在不修改原有代码的情况下扩展对象的功能。

4. 简化对象创建:
   使用原型可以简化对象的创建过程。通过将共享的属性和方法定义在原型上,然后通过构造函数或对象字面量创建对象,可以快速创建具有相同属性和方法的多个对象实例。这种方式避免了在每个对象上重复定义相同的属性和方法,提高了代码的可读性和可维护性。

5. 支持函数式编程:
   原型是JS函数式编程的重要基础。函数在JS中也是对象,它们也有原型。通过在函数的原型上添加方法,可以实现函数的扩展和组合。这种函数式编程的特性使得JS可以更好地支持高阶函数、闭包等概念,提供了更多的编程范式和抽象能力。

总之,原型是JS语言的一个核心概念,它提供了一种灵活、动态的对象继承和扩展机制。通过原型,JS可以实现对象之间的属性和方法共享,动态扩展对象,简化对象创建,并支持函数式编程。这些特性使得JS成为一种强大而灵活的编程语言,特别适用于前端开发领域。

JS中的原型和原型链

JavaScript 中的原型（Prototype）和原型链（Prototype Chain）是实现对象继承和属性共享的重要机制。下面我将详细解释原型和原型链的概念，并提供相应的代码示例。

1. 原型（Prototype）：
   - 在 JavaScript 中，每个函数都有一个 prototype 属性，它指向一个对象，称为原型对象。
   - 原型对象包含了该函数所有实例共享的属性和方法。
   - 当使用函数作为构造函数创建新对象时，新对象会继承原型对象的属性和方法。

示例代码：

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

```
Person.prototype.sayHello = function() {
  console.log("Hello, my name is " + this.name);
};

var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);

person1.sayHello(); // 输出: Hello, my name is Alice
person2.sayHello(); // 输出: Hello, my name is Bob
```

在上述示例中，Person 函数是一个构造函数，它的 prototype 属性指向一个原型对象。通过在原型对象上定义 sayHello 方法，所有 Person 的实例（person1 和 person2）都继承了该方法，可以直接调用。

2. 原型链（Prototype Chain）：
    - 原型链是 JavaScript 实现继承的方式。
    - 每个对象都有一个内部属性 [[Prototype]]（也称为 __proto__），它指向该对象的原型对象。
    - 当访问一个对象的属性或方法时，如果该对象本身没有定义该属性或方法，JavaScript 引擎会沿着原型链向上查找，直到找到匹配的属性或方法，或者到达原型链的末尾（null）。

示例代码：

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.eat = function() {
  console.log(this.name + " is eating.");
};

function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
```

```
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {
  console.log("Woof!");
};

var dog = new Dog("Buddy", "Labrador");
dog.eat();  // 输出: Buddy is eating.
dog.bark(); // 输出: Woof!
```

在上述示例中，Dog 构造函数通过 Object.create(Animal.prototype) 创建了一个新的对象，并将其赋值给 Dog.prototype，建立了 Dog 和 Animal 之间的原型继承关系。这样，Dog 的实例不仅继承了 Dog.prototype 的属性和方法，还可以通过原型链访问 Animal.prototype 的属性和方法。

当调用 dog.eat() 时，由于 dog 对象本身没有 eat 方法，JavaScript 引擎会沿着原型链向上查找，首先在 Dog.prototype 中查找，如果没有找到，则继续在 Animal.prototype 中查找，最终找到了 eat 方法并执行。

```
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
```

这两行代码的作用是建立 Dog 构造函数与 Animal 构造函数之间的原型继承关系，并修复 Dog.prototype.constructor 的指向。

1. Dog.prototype = Object.create(Animal.prototype);
    - 这行代码使用 Object.create() 方法创建了一个新对象，该对象的原型对象是 Animal.prototype。
    - 将新创建的对象赋值给 Dog.prototype，使得 Dog 的原型对象成为 Animal 原型对象的一个实例。

- 通过这种方式，Dog 的实例可以通过原型链访问 Animal 原型对象上的属性和方法。
- 这行代码建立了 Dog 和 Animal 之间的原型继承关系。

示例代码：

```
console.log(Dog.prototype.__proto__ === Animal.prototype); // 输出: true
```

上述代码验证了 Dog.prototype 的原型对象确实是 Animal.prototype。

1. Dog.prototype.constructor = Dog;
   - 在上一行代码中，我们将 Dog.prototype 设置为一个新对象，这会导致 Dog.prototype.constructor 指向 Animal 构造函数，而不是 Dog 构造函数。
   - 为了修复这个问题，我们需要手动将 Dog.prototype.constructor 重新指向 Dog 构造函数。
   - 这行代码确保了 Dog 的实例的 constructor 属性指向正确的构造函数。

示例代码：

```
var dog = new Dog("Buddy", "Labrador");
console.log(dog.constructor === Dog); // 输出: true
```

上述代码验证了 dog 对象的 constructor 属性确实指向 Dog 构造函数。

总结：

- Dog.prototype = Object.create(Animal.prototype); 建立了 Dog 和 Animal 之间的原型继承关系，使 Dog 的实例可以访问 Animal 原型对象上的属性和方法。
- Dog.prototype.constructor = Dog; 修复了 Dog.prototype.constructor 的指向，确保 Dog 的实例的 constructor 属性指向 Dog 构造函数。

这两行代码是在使用原型继承时常见的模式，用于正确设置原型链和维护 constructor 属性的指向。它们确保了 Dog 构造函数与 Animal 构造函数之间的继承关系，并保持了正确的构造函数引用。

3. 原型链的特点和注意事项：

- 原型链的末尾是 Object.prototype，它的 [[Prototype]] 属性指向 null。
- 通过原型链实现继承时，子类的原型对象应该是父类的实例，而不是直接引用父类的原型对象。
- 在原型链上进行属性查找时，如果原型链较长，可能会影响性能。
- 如果在原型链的不同层级上定义了同名属性或方法，则在查找时会优先访问更近的定义。

示例代码：

```
console.log(dog instanceof Dog);      // 输出: true
console.log(dog instanceof Animal);   // 输出: true
console.log(dog instanceof Object);   // 输出: true

console.log(dog.hasOwnProperty("name"));  // 输出: true
console.log(dog.hasOwnProperty("eat"));   // 输出: false

dog.eat = function() {
  console.log("Overridden eat method");
};
dog.eat(); // 输出: Overridden eat method
```

在上述示例中，通过 instanceof 运算符可以验证对象的原型链关系。dog 对象既是 Dog 的实例，也是 Animal 和 Object 的实例，因为它们在原型链上。

使用 hasOwnProperty 方法可以检查属性是否直接存在于对象本身，而不是继承自原型链。

如果在对象上直接定义与原型链上同名的属性或方法，会覆盖原型链上的定义，优先访问对象自身的定义。

原型和原型链是 JavaScript 中实现对象继承和属性共享的基础机制。理解原型和原型链的工作原理对于编写和理解 JavaScript 代码非常重要。通过合理利用原型和原型链，可以实现代码的复用、继承和扩展，提高开发效率和代码的可维护性。

从编译原理的角度理解原型和原型链

从编译原理的角度来看，JavaScript 的原型和原型链机制与面向对象编程语言中的继承和多态有一些相似之处。下面我们从编译原理的角度深入探讨原型和原型链。

1. 对象的内部表示：
    - 在 JavaScript 的实现中，每个对象都有一个内部的属性列表，用于存储对象的属性和方法。
    - 对象的属性列表通常采用哈希表或类似的数据结构来实现，以便快速查找和访问属性。
    - 除了自身的属性列表外，每个对象还有一个指向其原型对象的内部指针（通常称为 [[Prototype]] 或 __proto__）。
2. 原型对象的表示：
    - 原型对象也是一个普通的 JavaScript 对象，它具有自己的属性列表。
    - 原型对象的属性列表中包含了所有实例对象共享的属性和方法。
    - 当创建一个新对象时，该对象的原型指针会指向其构造函数的 prototype 属性所引用的原型对象。
3. 属性查找和原型链：
    - 当访问一个对象的属性或方法时，JavaScript 引擎首先在对象自身的属性列表中查找该属性。
    - 如果在对象自身的属性列表中找到了该属性，则直接返回属性值或执行相应的方法。
    - 如果在对象自身的属性列表中没有找到该属性，JavaScript 引擎会沿着对象的原型链向上查找。
    - 原型链是由对象的原型指针连接而成的一条链表，它表示了对象的继承关系。
    - JavaScript 引擎会依次访问原型链上的每个对象，在其属性列表中查找目标属性，直到找到该属性或者达到原型链的末尾（null）。
4. 原型链的解析和优化：
    - 在 JavaScript 的编译和解释过程中，原型链的解析和查找是一个频繁发生的操作。
    - 为了提高属性查找的效率，JavaScript 引擎会对原型链进行优化和缓存。

- 常见的优化技术包括内联缓存（Inline Caching）和隐藏类（Hidden Class）。
- 内联缓存会将属性的查找结果缓存起来，当下次访问相同的属性时，可以直接使用缓存的结果，避免重复查找。
- 隐藏类用于跟踪对象的属性布局，通过共享相同的隐藏类，可以加速属性的访问和优化内存布局。

5. 原型继承的实现：
   - 在 JavaScript 中，原型继承是通过将一个对象的原型指针指向另一个对象来实现的。
   - 当创建一个新对象时，可以将其原型指针设置为另一个对象，从而建立继承关系。
   - 常见的继承模式包括原型链继承、构造函数继承和组合继承等。
   - 这些继承模式通过操作原型指针和构造函数的 prototype 属性来建立和维护对象之间的继承关系。
6. 原型和原型链的性能考量：
   - 原型和原型链的设计虽然提供了灵活的继承和属性共享机制，但也引入了一些性能上的考量。
   - 过长的原型链可能会导致属性查找的性能下降，因为需要遍历更多的对象才能找到目标属性。
   - 频繁地修改原型对象上的属性可能会影响所有继承自该原型的对象，导致性能下降和意外的行为。
   - 因此，在设计和使用原型和原型链时，需要权衡灵活性和性能，并采取适当的优化措施。

从编译原理的角度来看，JavaScript 的原型和原型链机制与传统的面向对象编程语言有一些区别。JavaScript 通过对象的属性列表和原型指针来实现继承和属性共享，而不是直接使用类和继承关系。这种基于原型的设计提供了更大的灵活性和动态性，但也引入了一些性能和优化方面的挑战。

JavaScript 引擎在编译和解释过程中，需要进行原型链的解析和优化，以提高属性查找和访问的效率。通过使用内联缓存、隐藏类等技术，可以加速原型链的查找和优化内存布局。

总之，原型和原型链是 JavaScript 中实现对象继承和属性共享的核心机制。从编译原理的角度来看，它们涉及到对象的内部表示、属性查找、原型链解析以及相关的优化技术。理解原型和原型链的工作原理和性能特点，有助于编写高效和可维护的 JavaScript 代码。