DEV Create account

Shingai Zivuku
Posted on Oct 30, 2023

5

Detailed Explanation of JavaScript Prototype Chain

#javascript #tutorial #programming #webdev

Every object in JavaScript has a prototype property, which references another object. The referenced object also has its own prototype, and so on, forming a prototype chain. The top of the prototype chain is `Object.prototype`, which is the root prototype of all objects.

When accessing a property of an object, JavaScript will search the prototype chain for a matching property, starting with the object itself and moving up the chain until it reaches the end of the chain. If no matching property is found, the property lookup fails.

How the JavaScript prototype chain works

The working principle of the JavaScript prototype chain is very simple. Here's a very simple example of how the prototype chain works:

```
// Define a human constructor
function Person(name, age) {
    this.name = name;
    this.age = age;
}

// Add a method for greeting
Person.prototype.greet = function() {
    console.log(`Hi, my name is ${this.name} and I'm ${this.age} years old.`
};

// Create a human instance
const person = new Person('John', 30);

// Access the person's name attribute and output "John"
console.log(person.name);
```

```
person.greet();
```

In this example, we create a Person constructor and add a greet method to its prototype object. When we create a person instance using the new keyword, it inherits the greet method from the Person constructor's prototype object.

Because the person object itself does not have a greet method, JavaScript will look up the prototype chain upwards until it finds a matching method.

Multiple Ways to Create Objects

In JavaScript, we have many ways to create objects. Depending on how it is created, the prototype chain of a JavaScript object will be different.

Creating Objects Using Literals

The prototype of objects created using literals is `Object.prototype`, that is, all objects created by literals are instances of Object.

```
// Create an empty object
const emptyObj = {};
// Create an object with properties and methods
const obj = {
    name: 'John',
    age: 30,
    greet() {
        console.log(`Hi, my name is ${this.name} and I'm ${this.age} years ol
    }
};
```

Creating Objects Using Constructor Methods

The prototype of an object created using a constructor is the prototype of the constructor function.

```
// Define a Person constructor
function Person(name, age) {
    this.name = name;
    this.age = age;
}
// Add a greet method
Person.prototype.greet = function () {
    console.log(`Hi, my name is ${this.name} and I'm ${this.age} years old.`
};
```

In this example, we create a Person object using the constructor method. When we use the `new` keyword to create a person instance, it inherits the `greet` method from the Person constructor's prototype.

Class-based Object Creation

Introduced in ES6, the class keyword makes classes and inheritance in JavaScript more intuitive and straightforward. While still underpinned by the prototype chain mechanism, this syntactic sugar streamlines the object creation and inheritance process.

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

    greet() {
      console.log(`Hi, my name is ${this.name} and I'm ${this.age} years old.
    }
  }

  const person = new Person('John', 30);
```

Class-based objects are identical to constructor-based objects. Their prototype is the object referenced by the Class constructor's prototype property.

Modifying Prototype and Inheritance

Modifying an object's prototype can change its inheritance relationship. When an object's prototype changes, so does its prototype chain, allowing objects to inherit properties and methods from their prototype chain. This enables code reuse and extension.

```
// Define an Animal constructor function
class Animal {
  constructor(legs) {
    this.legs = legs;
  }

  move() {
    console.log('Moving...');
```

```
// Define a Bird constructor function
class Bird extends Animal {
  constructor(name, legs) {
    super(legs);
    this.name = name;
  }

  fly() {
    console.log('Flying...');
  }
}

// Create a new bird object
const bird = new Bird('Pigeon', 2);

// Print the bird's name and legs
console.log(bird.name); // Pigeon
console.log(bird.legs); // 2

// Call the move and fly methods
bird.move(); // Moving...
bird.fly(); // Flying...
```

Here we define an Animal constructor with a move method. We then define a Bird constructor that inherits the prototype object of the Animal constructor using the `Object.create()` method, thus enabling the Bird constructor to inherit the move method. Finally, we add a fly method to the Bird constructor and create a bird object.

Performance and Optimization of JavaScript Prototype Chain

The prototype chain in JavaScript can introduce a performance overhead, as accessing a property requires searching the chain step-by-step until the property is found or the end of the chain is reached. A deep prototype chain can therefore lead to performance degradation. To optimize performance, design object prototype chains carefully to avoid overly large and complex structures.

Advanced JavaScript Prototype Chain

- Deep dive into the implementation details of the prototype chain, including `[[Prototype]]` attributes, `__proto__` properties, and the `Object.getPrototypeOf()` method.

- Examine the relationship between the prototype chain and the scope chain, and explain why global variables can be accessed in the prototype object.
- Discuss the use cases of the `Object.create()` and `Object.setPrototypeOf()` methods in prototype chain operations.
- Recommend excellent resources and articles, such as MDN documentation and the JavaScript authoritative guide, for further study.

Conclusion

By deeply understanding the concept, working principle, and applications of the JavaScript prototype chain, we can better understand the underpinnings of JavaScript object-oriented programming. The prototype chain is not only the foundation for inheritance and property lookup in JavaScript, but it also helps us to gain a deeper understanding of the design philosophy of the JavaScript language. By skillfully applying the prototype chain, we can write efficient and scalable front-end code.

DEV

Create account

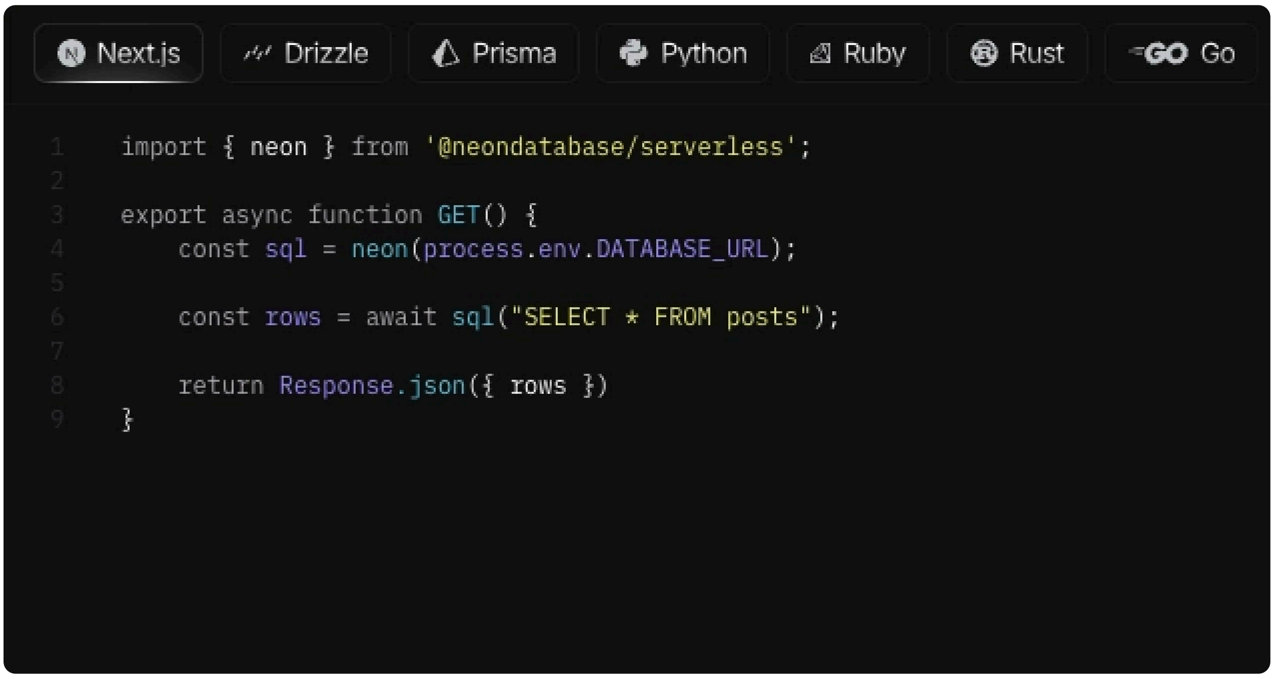
Read More

Top comments (0)

Code of Conduct • Report abuse

Neon PROMOTED

•••



```
import { neon } from '@neondatabase/serverless';

export async function GET() {
    const sql = neon(process.env.DATABASE_URL);

    const rows = await sql("SELECT * FROM posts");

    return Response.json({ rows })
}
```

Next.js applications: Set up a Neon project in seconds

If you're starting a new project, Neon has got your databases covered. No credit cards. No trials. No getting in your way.

Get started →

Shingai Zivuku

My skillset spans DevSecOps, Cyber Security, and Technical Writing. I'm passionate about using technology to secure and optimize workflows. I'm also open to collaboration.

Create account

More from Shingai Zivuku

Frontend Refresh Project - An Electronic Spider
#javascript #programming #tutorial #webdev

Scopes, Scope Chains, and Closures Explained
#javascript #programming #beginners #tutorial

Optimizing File Processing in React with Multipart Uploads and Downloads
#javascript #react #frontend #productivity