17. 手写 bind 实现

1. 核心概念 (Core Concept)

`Function.prototype.bind()` 是一个内置的方法，它返回一个**新的函数**。这个新函数在被调用时，`this` 关键词会被指定为传递给 `bind()` 的第一个参数，并且随后的参数将作为提供给绑定的函数的参数序列的前面部分。

2. 为什么需要它？ (The "Why")

- **固定 `this` 指向:** 在 JavaScript 中，函数的 `this` 值取决于函数的调用方式。`bind` 提供了在函数被调用前，强制指定其 `this` 值的能力，这在面向对象编程、处理事件监听器或回调函数时非常有用，避免了 `this` 指向丢失的问题。
- **偏函数应用 (Partial Function Application):** `bind` 方法的第二个参数及其后的参数可以预设（固定）函数的参数。这意味着你可以创建一个新函数，它已经带有一部分参数，等待后续的参数补全。

3. API 与用法 (API & Usage)

虽然是手写实现，但理解原生 `bind` 的 API 是基础。原生 `bind` 的签名大致如下：

```
function.bind(thisArg, arg1, arg2, ...);
```

- `thisArg`：当绑定函数被调用时，`this` 关键字的值。可以设置为任何值，例如对象实例或 `null`。
- `arg1, arg2, ...`：当绑定函数被调用时，这些参数将作为目标函数调用时前置的参数。

手写实现的核心思路:

手写实现需要模拟原生 `bind` 的行为：

1. 返回一个新的函数。
2. 新函数被调用时，确保目标函数的 `this` 指向 `bind` 的第一个参数。
3. 新函数能够接收自己的参数，并将其与 `bind` 传入的预设参数合并后，一并传递给目标函数。
4. 考虑到绑定函数作为构造函数被调用时的特殊行为（此时 `this` 应该指向新创建的实例，而不是 `bind` 时指定的 `thisArg`）。

经典代码示例 (模拟实现):

```
// 模拟 Function.prototype.bind
Function.prototype.myBind = function(thisArg, ...args) {
  const originalFunc = this; // 这里的 'this' 指向调用 myBind 的函数本身
```

```
  // 检查调用者是否是函数
  if (typeof originalFunc !== 'function') {
      throw new TypeError('Function.prototype.myBind - what is trying to
be bound is not callable');
  }

  // 返回一个新的函数
  const boundFunc = function( ... innerArgs) {
    // 合并 bind 时传入的预设参数和调用 boundFunc 时传入的实际参数
    const totalArgs = args.concat(innerArgs);

    // 判断是否作为构造函数使用（通过 new 关键字调用）
    // 如果是，this 指向新创建的实例，否则指向 bind 传入的 thisArg
    // instanceof 检测可以识别通过 new boundFunc(...) 创建的对象
    if (this instanceof boundFunc) {
       // 当绑定的函数作为构造函数调用时
       const instance = new originalFunc( ... totalArgs);
       // originalFunc.apply(instance, totalArgs); // 另一种方式但更复杂处理原
型链
       return instance;

    } else {
       // 当绑定的函数作为普通函数调用时
       return originalFunc.apply(thisArg, totalArgs);
    }
  };

  // 确保绑定后的函数具有原函数的原型链，便于 instanceof 判断
  // 这是为了模拟原生 bind 作为构造函数时的行为
  // 注意：直接设置 boundFunc.prototype = originalFunc.prototype; 可能会导致问
题
  // MDN 建议创建一个中间函数进行原型链继承（如果需要完全模拟构造函数行为）
  // 一个简化的，能通过 instanceof boundFunc 的实现（但非完全模拟构造器行为）：
  // boundFunc.prototype = Object.create(originalFunc.prototype); // 这句可
忽略或根据需要添加更复杂逻辑

  return boundFunc;
};

// --- 示例用法 ---
const obj = { name: 'Alice' };

function greet(greeting, punctuation) {
  console.log(greeting + ', I am ' + this.name + punctuation);
}

// 普通函数调用
const boundGreet = greet.myBind(obj, 'Hello');
boundGreet('!'); // 输出: Hello, I am Alice!
```

```
// 偏函数应用 + 普通调用
const boundGreetFormal = greet.myBind(obj, 'Good morning', ',');
boundGreetFormal('!'); // 输出: Good morning, I am Alice,!

// 尝试作为构造函数（在模拟实现中需要特殊处理的部分）
function Person(name) {
  this.name = name;
}
const BoundPerson = Person.myBind({ name: 'Ignored' }); // 这里的 { name:
'Ignored' } 在 new 调用时会被忽略
const personInstance = new BoundPerson('Bob');
console.log(personInstance.name); // 输出: Bob（期望 behavior）
console.log(personInstance instanceof Person); // 输出: true（期望
behavior）
console.log(personInstance instanceof BoundPerson); // 输出: true
```

4. 关键注意事项 (Key Considerations)

- **返回值是新函数:** `bind` 不会改变原函数，而是返回一个**全新的函数**。每次调用 `bind` 都会产生一个不同的新函数实例。
- **`thisArg` 的处理:**
  - 如果 `thisArg` 是 `null` 或 `undefined`，在非严格模式下，`this` 会被自动替换为全局对象（浏览器中是 `window`）。在严格模式下，`this` 保持 `null` 或 `undefined`。
  - 如果 `thisArg` 是原始值（如字符串、数字、布尔值），它会被封装成对应的包装对象。
- **参数合并:** `bind` 传入的参数会固定在新函数被调用时的参数列表的前面。
- **作为构造函数使用:** 当使用 `new` 关键字调用 `bind` 返回的新函数时，`bind` 时指定的 `thisArg` 会被忽略，新函数内部的 `this` 将指向新创建的实例。同时，`bind` 时传入的预设参数和 `new` 调用时传入的参数会一并传递给原函数（作为构造函数）。手写实现需要特别处理这种情况。

5. 参考资料 (References)

- **MDN Web Docs:**
  - `Function.prototype.bind()` (核心参考)
- **ECMAScript Language Specification:**
  - ECMAScript 2024 Language Specification - 19.2.3.2 Function.prototype.bind(thisArg, ...args) (规范层面的详细定义，用于最高准确性验证)
- **业界公认技术博客:** (通常用于理解实现细节和常见陷阱，例如讲解 `new` 处理的部分)
  - (此处不列出具体博客链接以保持通用性，但手写实现过程中会参考如 You Don't Know JS 或其他高质量技术文章对 `bind` 实现细节的解析)