37. 实现一个简化版 EventEmitter

1. 核心概念 (Core Concept)

EventEmitter 是 Node.js 和许多 JavaScript 环境中实现事件发布/订阅（Publish/Subscribe）模式的核心模块或模式。它允许对象注册监听特定事件，并在事件发生时触发相应的回调函数。一个简化版的 EventEmitter 旨在模拟这一行为，提供 `on`（或 `addListener`）用于注册监听器，`emit`（或 `trigger`）用于触发事件，以及可选的 `off`（或 `removeListener`）用于移除监听器。

2. 为什么需要它？ (The "Why")

1. **解耦 (Decoupling):** 发布者和订阅者之间无需直接引用，通过事件作为中介进行通信，降低模块间的耦合度。
2. **灵活性 (Flexibility):** 一个事件可以有多个监听器，一个对象可以监听多个事件。支持一对多和多对多的通信模式。
3. **响应式 (Reactive):** 允许对象对异步发生的事件做出响应，是构建事件驱动架构的基础。

3. API 与用法 (API & Usage)

一个简化版的 EventEmitter 通常会实现以下核心 API:

- **`on(eventName, listener)`**: 注册一个监听指定 `eventName` 事件的 `listener` 回调函数。
- **`emit(eventName, [...args])`**: 触发指定 `eventName` 事件，并将 `...args` 参数传递给该事件的所有监听器。
- **`off(eventName, listener)` (可选)**: 注销指定 `eventName` 事件的某个 `listener`。

以下是一个基于常见实现思路的简化版 EventEmitter 代码示例：

```
class SimpleEventEmitter {
    /**
     * 存储事件名到监听器列表的映射
     * @private
     * @type {Map<string, Function[]>}
     */
    _events = new Map();

    /**
     * 注册一个事件监听器。
     * @param {string} eventName 事件的名称。
     * @param {Function} listener 事件发生时应调用的函数。
     * @returns {this} 返回 EventEmitter 实例，方便链式调用。
     */
```

```
    on(eventName, listener) {
        if (typeof listener !== 'function') {
            throw new TypeError('Listener must be a function');
        }

        if (!this._events.has(eventName)) {
            this._events.set(eventName, []);
        }

        // 避免重复添加相同的监听器，简化版可以不考虑，但实际通常会考虑
        // if (!this._events.get(eventName).includes(listener)) {
        this._events.get(eventName).push(listener);
        // }

        return this;
    }

    /**
     * 触发一个事件，调用所有注册到该事件的监听器。
     * @param {string} eventName 要触发的事件的名称。
     * @param {...any} args 传递给监听器的参数。
     * @returns {boolean} 如果事件有监听器被调用，则返回 true；否则返回 false。
     */
    emit(eventName, ...args) {
        if (!this._events.has(eventName)) {
            return false; // 没有注册该事件的监听器
        }

        const listeners = this._events.get(eventName).slice(); // 创建副本
以防止在遍历过程中修改数组

        for (const listener of listeners) {
            try {
                 // 在这里，监听器中的 `this` 通常会丢失，如果是类方法，需要绑定或
使用箭头函数
                 // `listener.call(this, ...args);` 如果需要绑定触发者的上下文
                listener(...args);
            } catch (error) {
                // 实际 EventEmitter 通常会有一个 'error' 事件来处理监听器中的错
误
                console.error(`Error in event listener for
"${eventName}":`, error);
            }
        }

        return listeners.length > 0;
    }

    /**
     * 移除指定的事件监听器。
```

```
     * @param {string} eventName 事件的名称。
     * @param {Function} listener 要移除的监听器函数。
     * @returns {this} 返回 EventEmitter 实例，方便链式调用。
     */
    off(eventName, listener) {
         if (typeof listener !== 'function') {
            throw new TypeError('Listener must be a function');
        }

        if (!this._events.has(eventName)) {
            return this; // 没有注册该事件的监听器
        }

        const listeners = this._events.get(eventName);
        const index = listeners.indexOf(listener);

        if (index !== -1) {
            listeners.splice(index, 1);
        }

        // 如果该事件没有剩余的监听器，可以考虑从 Map 中移除该键，以节省内存
        if (listeners.length === 0) {
             this._events.delete(eventName);
        }

        return this;
    }

     /**
      * 移除所有事件的所有监听器，或指定事件的所有监听器。
      * @param {string} [eventName] 如果指定，则只移除该事件的监听器；否则移除所有事件的监听器。
      * @returns {this} 返回 EventEmitter 实例，方便链式调用。
      */
    removeAllListeners(eventName) {
        if (eventName) {
            this._events.delete(eventName);
        } else {
            this._events.clear();
        }
        return this;
    }
}

// --- 示例用法 ---
const myEmitter = new SimpleEventEmitter();

function handler1(arg1, arg2) {
    console.log('Handler 1 received:', arg1, arg2);
}
```

```
function handler2(arg) {
    console.log('Handler 2 received:', arg);
}

myEmitter.on('data', handler1);
myEmitter.on('data', handler2);
myEmitter.on('status', (status) => {
    console.log('Status changed:', status);
});

console.log('--- Emitting "data" ---');
myEmitter.emit('data', 'Hello', 'World'); // Output: Handler 1 received:
Hello World, Handler 2 received: Hello World

console.log('--- Emitting "status" ---');
myEmitter.emit('status', 'active'); // Output: Status changed: active

console.log('--- Emitting non-existent event ---');
myEmitter.emit('nonexistent'); // Output: false (returned by emit)

console.log('--- Removing handler2 from "data" ---');
myEmitter.off('data', handler2);

console.log('--- Emitting "data" again ---');
myEmitter.emit('data', 'Goodbye'); // Output: Handler 1 received: Goodbye

console.log('--- Removing all listeners for "data" ---');
myEmitter.removeAllListeners('data');

console.log('--- Emitting "data" one last time ---');
myEmitter.emit('data', 'Final'); // No output
```

4. 关键注意事项 (Key Considerations)

1. **内存管理:** 当一个事件不再被使用时，如果没有移除其监听器，可能会导致内存泄漏。实现 `off` / `removeListener` 和 `removeAllListeners` 是关键。
2. **错误处理:** 监听器函数内部的错误不会自动向上冒泡。实际的 EventEmitter 通常会触发一个特殊的 `error` 事件来处理监听器异常，以避免程序崩溃。
3. **`this` 上下文:** 在 `emit` 触发时，如果不显式绑定，监听器函数中的 `this` 默认可能指向 `undefined` (严格模式下) 或全局对象 (非严格模式下)。 Node.js 的 EventEmitter 默认会将 `this` 强制绑定到 EventEmitter 实例，实现时需注意这一点。
4. **同步执行:** 默认的 `emit` 方法会**同步**调用所有监听器。如果某个监听器执行耗时，可能会阻塞后续代码。对于需要异步执行的场景，监听器内部需要手动处理异步逻辑（如使用 `setTimeout`, `Promise` 等）。
5. **`once` 实现:** 常见的 EventEmitter 还包括 `once` 方法，用于注册一个只执行一次的监听器。这可以通过在内部包装原始监听器，并在首次执行后立即移除自身来实现。

5. 参考资料 (References)

- **Node.js EventEmitter 文档:** https://nodejs.org/dist/latest/docs/api/events.html (这是最权威的 EventEmitter 规范，尽管实现可能有所不同，但其 API 和行为理念是核心参考)
- **MDN EventTarget 文档:** https://developer.mozilla.org/en-US/docs/Web/API/EventTarget (Web 标准中的事件模型，概念相似，但实现和 API 有差异)
- **JavaScript 事件循环与事件驱动 (概念层面):** https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/EventLoop (理解 EventEmitter 是事件驱动架构的一部分)
