Tema: Asincronismo en JavaScript

Nombre: Carlos Ortiz
Docente: Christian Muñoz
Fecha: Domingo 17 de nov. de 2024

Consulte sobre el asincronismo en JavaScript y responda las siguientes preguntas de manera crítica. Además, suba un ejemplo de cada uno al repositorio de GitHub.

¿Qué es el asincronismo en JavaScript?

El asincronismo en JavaScript permite ejecutar tareas que llevan tiempo (como leer archivos, hacer solicitudes HTTP, o consultar una base de datos) sin detener la ejecución del resto del código. Esto es esencial porque JavaScript es un lenguaje **single-threaded**: solo tiene un hilo de ejecución principal (MDN, 2024).

JavaScript usa un mecanismo llamado **event loop** para manejar tareas asincrónicas, dividiéndolas en:
- **Call Stack**: Donde se ejecutan las funciones.
- **Task Queue**: Cola de tareas pendientes para ser procesadas.
- **Microtask Queue**: Cola de microtareas, como las promesas resueltas, que tienen prioridad.

1. ¿Cuál es el funcionamiento de los callbacks?

Un **callback** es una función que se pasa como argumento a otra función para que sea ejecutada más tarde, generalmente cuando una operación asincrónica se completa (MDN, 2024).

Por ejemplo:
Al realizar un pedido a domicilio:
- Llamas al restaurante y haces un pedido (función principal).
- Dejas tu número para que te llamen cuando el pedido esté listo (callback).
- Mientras tanto, sigues con tus actividades (ejecución continua).
- Cuando el restaurante termina tu pedido, te llaman para notificarte (ejecutan el callback).

Ejemplo técnico:

```
function descargarArchivo(url, callback) {
   console.log(`Descargando archivo de ${url}`);
   setTimeout(() => {
      console.log("Archivo descargado");
      callback(); // Llama al callback después de la descarga
   }, 3000); // Simula 3 segundos de espera
}

descargarArchivo("https://example.com/archivo", () => {
```

```
    console.log("Archivo procesado después de la descarga");
});
```

2. ¿Cómo mejora el uso de promesas?

Una promesa es un objeto que representa el eventual resultado (éxito o error) de una operación asincrónica. Tiene tres estados:
1. **Pending**La operación aún no se ha completado.
2. **Fulfilled**: La operación se completó con éxito.
3. **Rejected**: La operación falló.

Ejemplo:
Piensa en un contrato legal:
- **Pendiente**: Ambos están negociando.
- **Cumplido**: Las condiciones se cumplen.
- **Rechazado**: Algo sale mal y se rompe el contrato.

Ejemplo técnico:

```
function descargarArchivo(url) {
    return new Promise((resolve, reject) => {
        console.log(`Descargando archivo de ${url}`);
        setTimeout(() => {
            const exito = true; // Simula si la operación tiene éxito
            if (exito) {
                resolve("Archivo descargado correctamente");
            } else {
                reject("Error al descargar el archivo");
            }
        }, 3000);
    });
}

descargarArchivo("https://example.com/archivo")
    .then((mensaje) => {
        console.log(mensaje); // Se ejecuta si la promesa se resuelve
    })
    .catch((error) => {
        console.error(error); // Se ejecuta si la promesa se rechaza
    });
```

Ventajas de las promesas:
1. Mejor legibilidad.
2. No más anidamientos excesivos.
3. Manejo integrado de errores con `.catch()`.

3. ¿Cómo se revolucionó el manejo asincrónico con async y await?

Async/Await: Una revolución en el asincronismo

El **async/await** es una sintaxis más legible para trabajar con promesas. Introducido en ES2017, permite escribir código asincrónico que parece síncrono (MDN, 2024).

- `**async**`: Marca una función como asincrónica. Siempre devuelve una promesa.
- `**await**`: Pausa la ejecución de la función asincrónica hasta que una promesa se resuelva o rechace.

Analogía:
Piensa en una lista de tareas (promesas). Con async/await, es como si pudieras hacer "pausa" en una tarea hasta que termine, y luego pasar a la siguiente.

Ejemplo técnico:

```
async function procesarArchivo(url) {
    try {
        console.log("Iniciando descarga...");
        const mensaje = await descargarArchivo(url); // Pausa hasta que la promesa se
resuelva
        console.log(mensaje);
        console.log("Procesamiento del archivo completado");
    } catch (error) {
        console.error("Error:", error);
    }
}

procesarArchivo("https://example.com/archivo");
```

Ventajas de async/await:
1. **Legibilidad**: Parece código síncrono, lo que facilita su comprensión.
2. **Manejo de errores más limpio:** Usa `try...catch`.
3. **Ejecución secuencial y ordenada:** Ideal para tareas que dependen unas de otras.

6. Comparativa técnica entre callbacks, promesas y async/await

Característica	Callbacks	Promesas	Async/Await
Legibilidad	Baja (anidamientos excesivos)	Mejor	Alta
Manejo de errores	Manual (try/catch o lógica)	Integrado con .catch()	Uso nativo de try/catch
Código secuencial	Difícil	Más fácil	Muy sencillo
Complejidad	Crece con la cantidad de tareas	Más manejable	Simplificado

Conclusión:
- **Callbacks** son la base del asincronismo, pero se vuelven difíciles de manejar en escenarios complejos.
- **Promesas** resuelven muchos problemas de los callbacks, como el "callback hell".
- **Async/await** simplifica aún más el manejo de tareas asincrónicas, haciéndolo más legible y fácil de mantener.

Bibliografia:

MDN. (2024). *Introducción a JavaScript asíncrono*. Introducción a JavaScript Asíncrono.