Реализация Sleep()

При первом изучении JavaScript многие разработчики используют конструкцию, аналогичную `Thread.sleep()` в Java, пытаясь ввести неблокирующую задержку в программу. Раньше это был краеугольный педагогический способ представления, как `setTimeout` вписывается в поведение JavaScript во время выполнения.

С асинхронными функциями это уже не так! Построить утилиту, которая позволяет функции `sleep()` — «заснуть» — на миллисекунды, очень просто:

```
async function sleep(delay) {
    return new Promise((resolve) => setTimeout(resolve, delay));
}

async function foo() {
    const t0 = Date.now();
    await sleep(1500);    // sleep for ~1500ms
    console.log(Date.now() - t0);
}
foo();
// 1502
```

Максимизация распараллеливания

Если ключевое слово `await` не используется с осторожностью, программа может упустить возможные ускорения распараллеливания. Рассмотрим следующий пример, который ожидает пять случайных тайм-аутов последовательно:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
 const t0 = Date.now();
 await randomDelay(0);
 await randomDelay(1);
 await randomDelay(2);
 await randomDelay(3);
 await randomDelay(4);
 console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 0 finished
// 1 finished
// 2 finished
// 3 finished
// 4 finished
// 2219ms elapsed
```

После свертывания в цикл `for` получаем следующее:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
    const t0 = Date.now();
    for (let i = 0; i < 5; ++i) {
        await randomDelay(i);
    }
    console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 0 finished
// 1 finished
// 2 finished
// 3 finished
// 4 finished
// 2219ms elapsed
```

Даже если между промисами нет взаимозависимости, эта асинхронная функция будет приостанавливать работу и ждать завершения каждого из них, прежде чем запускать следующий. Это гарантирует сохранение порядка, но за счет общего времени выполнения.

Если сохранение порядка не требуется, лучше инициализировать промисы сразу и ожидать результатов по мере их появления. Это можно сделать следующим образом:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        setTimeout(console.log, 0, `${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
    const t0 = Date.now();

    const p0 = randomDelay(0);
    const p1 = randomDelay(1);
    const p2 = randomDelay(2);
    const p3 = randomDelay(3);
    const p4 = randomDelay(4);

    await p0;
```

```
    await p1;
    await p2;
    await p3;
    await p4;

    setTimeout(console.log, 0, `${Date.now() - t0}ms elapsed`);
}
foo();

// 1 finished
// 4 finished
// 3 finished
// 0 finished
// 2 finished
// 2219ms elapsed
```

После свертывания в массив и цикла `for` получаем следующее:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
    const t0 = Date.now();

    const promises = Array(5).fill(null).map((_, i) => randomDelay(i));

    for (const p of promises) {
        await p;
    }

    console.log(`${Date.now() - t0}ms elapsed`);
}

foo();

// 4 finished
// 2 finished
// 1 finished
// 0 finished
// 3 finished
// 877ms elapsed
```

Обратите внимание, что, хотя выполнение промисов нарушило порядок, операторы промисов предоставляются разрешенным значениям *в* порядке:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
```

```
        resolve(id);
    }, delay));
}

async function foo() {
    const t0 = Date.now();

    const promises = Array(5).fill(null).map((_, i) => randomDelay(i));

    for (const p of promises) {
        console.log(`awaited ${await p}`);
    }
    console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 1 finished
// 2 finished
// 4 finished
// 3 finished
// 0 finished
// awaited 0
// awaited 1
// awaited 2
// awaited 3
// awaited 4
// 645ms elapsed
```

Серийное выполнение промисов

В разделе «Промисы» этой главы обсуждается, как составлять промисы, которые выполняются последовательно и передают значения последующему промису. С async/await цепочка промисов становится очень простой:

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

async function addTen(x) {
    for (const fn of [addTwo, addThree, addFive]) {
        x = await fn(x);
    }
    return x;
}

addTen(9).then(console.log);    // 19
```

Здесь `await` напрямую передает возвращаемое значение каждой функции и результат выводится итеративно. Предыдущий пример не имеет дела с промисами, но его можно перенастроить на использование асинхронных функций — и, следовательно, промисов — вместо этого:

```
async function addTwo(x) {return x + 2;}
async function addThree(x) {return x + 3;}
```

```
async function addFive(x) {return x + 5;}

async function addTen(x) {
    for (const fn of [addTwo, addThree, addFive]) {
        x = await fn(x);
    }
    return x;
}

addTen(9).then(console.log);    // 19
```

Трассировка стека и управление памятью

Промисы и асинхронные функции имеют значительную степень совпадения с точки зрения функциональности, которую они предоставляют, но они значительно расходятся, когда дело доходит до того, как они представлены в памяти. Рассмотрим следующий пример, который показывает чтение трассировки стека для отклоненного промиса:

```
function fooPromiseExecutor(resolve, reject) {
    setTimeout(reject, 1000, 'bar');
}

function foo() {
    new Promise(fooPromiseExecutor);
}

foo();
// Uncaught (in promise) bar
// setTimeout
// setTimeout (async)
// fooPromiseExecutor
// foo
```

Если вы основываетесь на полученном ранее понимании промисов, это чтение трассировки стека должно вас озадачить. Трассировка стека должна в буквальном смысле, представлять вложенную природу вызовов функций, которая существует в настоящее время в стеке памяти движка JavaScript. Когда обработчик тайм-аута выполняет и отклоняет промис, показанное сообщение об ошибке идентифицирует вложенные функции, которые были вызваны для первоначального создания экземпляра промиса. Однако известно, что эти функции *уже возвращены* и, следовательно, не будут найдены в трассировке стека.

Ответ прост: движок JavaScript выполняет дополнительную работу по сохранению стека вызовов, в то время как это возможно при создании структуры промиса. Когда выдается ошибка, этот стек вызовов извлекается логикой обработки ошибок среды выполнения и поэтому доступен в трассировке стека. Это, конечно, означает, что он должен сохранять трассировку стека в памяти и потребуются затраты на вычисления и хранение.

Рассмотрим предыдущий пример, как если он был переработан с помощью асинхронных функций:

```
function fooPromiseExecutor(resolve, reject) {
    setTimeout(reject, 1000, 'bar');
}

async function foo() {
    await new Promise(fooPromiseExecutor);
}
foo();

// Uncaught (in promise) bar
// foo
// async function (async)
// foo
```

С этой структурой трассировка стека точно представляет текущий стек вызовов, потому что `fooPromiseExecutor` вернулась и больше не находится в стеке, но `foo` приостановлена и еще не завершена. Среда выполнения JavaScript может просто хранить указатель от вложенной функции на ее контейнерную функцию, как это было бы с синхронным стеком вызовов функций. Этот указатель эффективно сохраняется в памяти и может использоваться для генерации трассировки стека в случае ошибки. Такая стратегия не влечет за собой дополнительных затрат, как в случае с предыдущим примером, и поэтому должна быть предпочтительна, если производительность имеет решающее значение для вашего приложения.

ИТОГИ

Освоение асинхронного поведения в однопоточной среде выполнения JavaScript долгое время было сложной задачей. С введением промисов в ES6 и async/await в ES7 асинхронные конструкции в ECMAScript были значительно улучшены. Промисы и async/await не только сделали доступными паттерны, которые ранее было трудно или невозможно реализовать, но и породили совершенно новый способ написания JavaScript, который стал чище, короче и проще для понимания и отладки.

Промисы созданы, чтобы предложить чистую абстракцию вокруг асинхронного кода. Они могут представлять асинхронно исполняемый блок кода, но также могут представлять асинхронно вычисленное значение. Они особенно полезны в ситуации, когда возникает необходимость сериализации блоков асинхронного кода. Промисы — это восхитительно податливая конструкция: они могут быть сериализованы, объединены в цепочку, составлены, расширены и объединены.

Асинхронные функции являются результатом применения парадигмы промисов к функциям JavaScript. Они вводят возможность приостановить выполнение функции, не блокируя основной поток выполнения. Они чрезвычайно полезны как при написании читаемого кода, ориентированного на промисы, так и при управлении сериализацией и распараллеливанием асинхронного кода. Они являются одной из наиболее важных вещей в современном JavaScript-инструментарии.