Laboratorium 9 - Przetwarzanie asynchroniczne (wstęp do Node.js)

Piotr Karamon

09.12.2024r.

Treści zadań

Zadanie 1

- 1a: Zaimplementuj funkcję loop, wg instrukcji w pliku z Rozwiązaniem 3.
- 1b: wykorzystaj funkcję waterfall biblioteki async.

Zadanie 2

Proszę napisać program obliczający liczbę linii we wszystkich plikach tekstowych z danego drzewa katalogów. Do testów proszę wykorzystać zbiór danych Traceroute Data. Program powinien wypisywać liczbę linii w każdym pliku, a na końcu ich globalną sumę. Proszę zmierzyć czas wykonania dwóch wersji programu: z synchronicznym (jeden po drugim) przetwarzaniem plików, z asynchronicznym (jednoczesnym) przetwarzaniem plików. Przydatne moduły:

- `walkdir` – trawersacja drzewa katalogów
- `fs` – operacje na systemie plików (moduł wbudowany)

Do obliczania liczby linii w pliku tekstowym proszę wykorzystać następujący fragment kodu:

```
fs.createReadStream(file).on('data', function(chunk) {
                count += chunk.toString('utf8')
                .split(/\r\n|[\n\r\u0085\u2028\u2029]/g)
                .length-1;
            }).on('end', function() {
                console.log(file, count);
            }).on('error', function(err) {
                console.error(err);
            });
```

Zadanie 1

Mamy daną sekwencję operacji:

```
function printAsync(s, cb) {
  var delay = Math.floor(Math.random() * 1000 + 500);
  setTimeout(function () {
    console.log(s);
    if (cb) cb();
  }, delay);
}

function task(n) {
  return new Promise((resolve, reject) => {
    printAsync(n, function () {
```

```
      resolve(n);
    });
  });
}

task(1)
  .then((n) => {
    console.log("task", n, "done");
    return task(2);
  })
  .then((n) => {
    console.log("task", n, "done");
    return task(3);
  })
  .then((n) => {
    console.log("task", n, "done");
    console.log("done");
  });
```

Funkcja `loop(m)`

Wykorzystamy fakt, że taki ciąg operacji zwraca `Promise`, a zatem możemy wywołać na nim funkcję `then` w pętli `for`, by ”doklejać” kolejne zadania.

Tworzymy funkcję `task123`.

```
function task123() {
  return task(1)
    .then((n) => {
      console.log("task", n, "done");
      return task(2);
    })
    .then((n) => {
      console.log("task", n, "done");
      return task(3);
    })
    .then((n) => {
      console.log("task", n, "done");
      console.log("done");
    });
}
```

Funkcja `loop`:

```
function loop(m) {
  let promise = task123();
  for (i = 1; i < m; i++) {
    promise = promise.then(task123);
  }
  return promise;
}

loop(4);
```

Wynik programu(pierwotny ciąg operacji został wykomentowany):

```
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
```

Funkcja `loopWithWaterfall(m)`

Do rozwiązania naszego problemu możemy użyć funkcji `waterfall` z biblioteki `async`. Jak sama nazwa wskazuje, funkcja ta bierze ciąg asynchronicznych funkcji i wywołuje je w kolejności, przekazując wynik jednej do drugiej.

```
const { waterfall } = require("async");
function loopWithWaterfall(m) {
  const tasks = Array.from({ length: m }, () => async () => task123());
  waterfall(tasks);
}

loopWithWaterfall(4);
```

Wynik programu nie uległ zmianie:

```
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
```

Wnioski

- W obu funkcjach udało się uzyskać efekt sekwencyjnego wykonania.
- Dzięki wykorzystaniu funkcji zwracających `Promise` możemy budować sekwencje operacji w sposób czytelny i łatwy do zarządzania.
- Choć często sekwencje wywołań `then` są tworzone statycznie jedna po drugiej, to jak pokazuje funkcja `loop` można to robić w sposób dynamiczny, w celu np. wymuszenia kolejności pomiędzy zadaniami.
- Funkcja `waterfall` jest bardziej ograniczająca niż podejście przedstawione w funkcji `loop`, bo musimy na starcie podać tablicę naszych zadań do wykonania.

Zadanie 2

Najpierw tworzymy funkcję która zlicza ilość linii w pliku, według podanego kodu w poleceniu:

```
const SPLIT_REGEX = /\r\n|[\n\r\u0085\u2028\u2029]/g;

function countLinesInFile(path) {
  let count = 0;
  return new Promise((resolve, reject) => {
    fs.createReadStream(path)
      .on("data", function (chunk) {
        count += chunk.toString("utf8").split(SPLIT_REGEX).length - 1;
      })
      .on("end", function () {
        console.log(path, count);
        resolve(count);
      })
      .on("error", function (err) {
        console.error(err);
        reject(err);
      });
  });
}
```

Funkcja `countLinesInFile` zwraca obietnicę, która jest spełniona gdy zakończymy przetwarzanie pliku, a odrzucona gdy wystąpi błąd.

Podejście synchroniczne

W tym podejściu tworzymy sekwencję operacji w podobnym sposób jak w funkcji `loop`. Wartość `value` to obecna suma linii w plikach.

```
const walker = walkdir("./PAM08");
let promise = new Promise((resolve, _) => resolve(0));
walker.on("file", (path, stat) => {
  if (stat.isFile()) {
    promise = promise.then(
      async (value) => value + (await countLinesInFile(path))
    );
  }
});

walker.on("end", () => {
  promise
    .then((value) => {
      console.log("Total lines:", value);
    })
    .catch((err) => {
      console.error(err);
    });
});
```

```
C:\Users\piotr\Documents\tw-labjs\PAM08\WisconsinA\WisconsinA_www.yahoo.co.jp.html 25
C:\Users\piotr\Documents\tw-labjs\PAM08\WisconsinA\WisconsinA_www.yahoo.com.html 18
C:\Users\piotr\Documents\tw-labjs\PAM08\WisconsinA\WisconsinA_www.youtube.com.html 17
Total lines: 61823
```

Aby zmierzyć czas wykoania programu tworzymy funkcję `measure` i `calculateAverageTime`, w celu pomiaru czasu wykonania została zakomentowana linia.

```
console.log(path, count);
```

```
function measure() {
  return new Promise((resolve, reject) => {
    const start = performance.now();
```

```
    let promise = new Promise((resolve, _) => resolve(0));
    const walker = walkdir("./PAM08");

    walker.on("file", (path, stat) => {
      if (stat.isFile()) {
        promise = promise.then(
          async (value) => value + (await countLinesInFile(path))
        );
      }
    });

    walker.on("end", () => {
      promise
        .then((value) => {
          console.log("Total lines:", value);
          const end = performance.now();
          resolve(end - start);
        })
        .catch((err) => {
          console.error(err);
          reject(err);
        });
    });
  });
}

async function calculateAverageTime() {
  const N = 10;
  let sum = 0;
  for (let i = 0; i < N; i++) {
    sum += await measure();
  }
  console.log(`Average time: ${sum / N}ms`);
}

calculateAverageTime();
```

Wynik:

```
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Average time: 369.36642ms
```

Podejście asynchroniczne

Większość kodu jest identyczna z podejściem synchronicznym. Kluczowe zmiany znajdują się w callbackach do `walker.on("file")` i `walker.on(ęnd")`. W tym podejściu tworzymy tablicę obietnic/zadań, a następnie używając funkcji `Promise.all` czekamy, aż wszystkie się zakończą. Wtedy dostajemy tablice wartości zwróconych z naszych obietnic i je sumujemy.

```
const tasks = [];
const walker = walkdir("./PAM08");
walker.on("file", (path, stat) => {
  tasks.push(countLinesInFile(path));
});

walker.on("end", () => {
  Promise.all(tasks)
    .then((values) => {
      console.log(
        "Total lines:",
        values.reduce((a, b) => a + b, 0)
      );
    })
    .catch((err) => {
      console.error(err);
    });
});
```

Wynik:

```
...
C:\Users\piotr\Documents\tw-labjs\PAM08\WashingtonDCC\WashingtonDCC_www.wikipedia.org.html 33
C:\Users\piotr\Documents\tw-labjs\PAM08\WashingtonDCC\WashingtonDCC_www.yahoo.com.html 57
C:\Users\piotr\Documents\tw-labjs\PAM08\WashingtonDCC\WashingtonDCC_www.youtube.com.html 37
Total lines: 61823
```

W identyczny sposób jak w podejściu synchroniczmym tworzymy funkcję `measure`. Funkcja `calculateAverageTime` pozostaje bez zmian.

Wynik:

```
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Total lines: 61823
Average time: 144.57031ms
```

Wnioski

- Widzimy, że w celu rozwiązania naszego problemu użyto jednocześnie:
  - mechanizmu obietnic
  - `async` / `await`
  - callbacki

  Wszystkie te mechanizmy dobrze ze sobą współpracują, co jest szczególnie ważne gdy dana biblioteka wspiera tylko jeden z nich. `walkdir` wspiera jedynie callbacki, ale i tak mogliśmy łatwo stworzyć funkcję, która z niej korzysta i jednocześnie zwraca obietnicę. Choć takie "mieszanie" da się zrobić, to jednak tracimy na czytelności kodu, więc lepiej robić to tylko w ramach ostateczności.

- Wyniki obu podejść są identyczne.

- Wersja asynchroniczna jest wedle oczekiwań dużo szybsza od synchronicznej(ponad dwukrotnie).
- Programowanie asynchroniczne w języku JavaScript silnie różni się od programowania asynchroniczengo np. w Javie. W programowaniu asynchronicznym w JavaScript większy nacisk kładzie się na mechanizmy oparte na zdarzeniach oraz zarządzanie obietnicami. W Javie natomiast asynchroniczność często opiera się na wielowątkowości i zarządzaniu wątkami (np. przy użyciu `ExecutorService`).

Bibliografia

- Walkdir - npm package
- Node.js File System (fs) API
- Async.js Documentation