Teoria współbieżności

Laboratorium 9

Przetwarzanie asynchroniczne (wstęp do Node.js)

Albert Gierlach

02.12.2020

1. Zadanie 1

a) Zaimplementuj funkcję loop, wg instrukcji w pliku z Rozwiązaniem 3.

b) wykorzystaj funkcję waterfall biblioteki async.

2. Koncept rozwiązania

W podpunkcie a) do zapewnienia sekwencyjności wykorzystam rekurencję oraz mechanizm obietnic (*Promise*). Kod pojedynczej sekwencji zostanie zamknięty w funkcję, która zwróci obietnicę. Dzięki temu będzie wiadomo kiedy skończyła się sekwencja i kiedy wystartować nową.

Drugi podpunkt będzie korzystał z funkcji *waterfall*, który zapewnia wywołanie sekwencyjne, ale wywoływane funkcje muszą przyjmować parametr, który będzie *callbackiem* oraz muszą go wywołać, gdy zakończą operacje.

3. Implementacja oraz wyniki

Wariant pierwszy:

```
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);
        });
    });
}

function task_sequence() {
    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');
    });
}

function loop(m) {
    if (m === 0) {
        return;
    }

    task_sequence().then(() => {
        console.log("next sequence")
        loop(m - 1)
    })
}


loop(4);
```

Wykonanie programu dało następujące wyniki:

```
1
task 1 done
2
task 2 done
3
task 3 done
done
next sequence
1
task 1 done
2
task 2 done
3
task 3 done
done
next sequence
1
task 1 done
```

```
2
task 2 done
3
task 3 done
done
next sequence
1
task 1 done
2
task 2 done
3
task 3 done
done
next sequence
```

Wariant drugi (pominięto funkcje, które były zdefiniowane w poprzednim wariancie):

```
let async = require("async");

function task_sequence(cb) {
    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');
        console.log("next sequence")
        cb()
    });
}

function loop(m) {
    let task_list = Array.from({length: m}, () => task_sequence);
    async.waterfall(task_list);
}

loop(4);
```

Wykonanie programu dało następujące wyniki:

```
1
task 1 done
2
task 2 done
3
task 3 done
done
next sequence
1
task 1 done
2
task 2 done
3
task 3 done
done
next sequence
1
task 1 done
2
task 2 done
3
task 3 done
done
next sequence
1
task 1 done
2
task 2 done
3
task 3 done
done
next sequence
```

4. Wnioski

Obserwując listingi przedstawiące wyjście programów, można zauważyć, że sekwencje powtarzają się cztery razy, czyli tak jakie było założenie. Zastosowanie mechanizmu obietnic pozwala zagwarantować sekwencyjność operacji, ale większą jego zaletą jest to, że możemy zlecić jakąś operację i kontynuować inne operacje, podczas gdy w tle np. czekamy na pobranie danych

z zewnętrznego API. Dodatkowym plusem jest to, że możemy pisać kod, który nie bazuje na mechanizmie *callbacków*, przez co unikniemy tzw. *callback hell* oraz kod będzie bardziej czytelny oraz znacznie się uprości. Przykład poniżej - wersja z *callbackami*:

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       }else{
           dataBase.getRoles(username, (error, roles) => {
               if (error){
                   callback(error)
               }else {
                   dataBase.logAccess(username, (error) => {
                       if (error){
                           callback(error);
                       }else{
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

Kod z użyciem *Promises*

```
const verifyUser = function(username, password) {
   database.verifyUser(username, password)
       .then(userInfo => dataBase.getRoles(userInfo))
       .then(rolesInfo => dataBase.logAccess(rolesInfo))
       .then(finalResult => {
           //do whatever the 'callback' would do
       })
       .catch((err) => {
           //do whatever the error handler needs
       });
};
```

Kod z użyciem *async/await*

```
const verifyUser = async function(username, password){
   try {
       const userInfo = await dataBase.verifyUser(username, password);
       const rolesInfo = await dataBase.getRoles(userInfo);
       const logStatus = await dataBase.logAccess(userInfo);
       return userInfo;
   }catch (e){
       //handle errors as needed
   }
};
```

5. 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.

6. Koncept rozwiązania

Do zebrania ścieżek do plików wykorzystany zostanie moduł *walkdir*. Później zostaną stworzone dwa warianty funkcji zliczających linie w plikach - jedna synchronicza, a druga asynchroniczna. Wersja synchroniczna wykorzysta funckję *waterfall* w celu zapewnienia sekwencyjności, a wersja asynchroniczna skorzysta z metody *Promise.all*, która zagreguje wszystkie obiekty *Promise* i pozwoli wykonać operację na zwróconych wynikach.

7. Implementacja oraz wyniki

```
const walk = require('walkdir');
const fs = require('fs');
const async = require('async');
const performance = require('perf_hooks').performance;
```

```
function countLines(path) {
    return new Promise(((resolve, reject) => {
        let cnt = 0;
        fs.createReadStream(path).on('data', function (chunk) {
            cnt += chunk.toString('utf8')
                .split(/\r\n|[\n\r\u0085\u2028\u2029]/g)
                .length - 1;
        }).on('end', function () {
            // console.log(path, cnt);
            resolve(cnt);
        }).on('error', function (err) {
            // console.error(err);
            reject(err);
        });
    }));
}


function syncCount(paths) {
    let totalLines = 0;

    const tasks_arr = paths.map((p) => (cb) => {
        countLines(p).then(l => {
            totalLines += l;
            cb();
        });
    });

    return new Promise(((resolve) => {
        async.waterfall(tasks_arr)
            .then(() => {
                resolve(totalLines)
            })
    }));
}

const PATH = './PAM08'
const paths = walk.sync(PATH).filter(p => {
    return fs.lstatSync(p).isFile()
})

function measureSync(){
    const start = performance.now()
```

```
    syncCount(paths).then((totalLines) => {
        const timeElapsed = performance.now() - start;
        console.log("Synchronously: " + Math.round(timeElapsed) + "ms")
        console.log(totalLines + " lines")

        //now call async version
        measureAsync();
    })
}

function measureAsync(){
    const start = performance.now()
    Promise.all(
        paths.map(p => countLines(p))
    ).then((lines) => {
        const totalLines = lines.reduce((acc, val) => acc + val)

        const timeElapsed = performance.now() - start;
        console.log("Asynchronously: " + Math.round(timeElapsed) + "ms")
        console.log(totalLines + " lines")
    })
}

measureSync()
// measureAsync();
```

Wykonanie programu dało następujące efekty:

```
Synchronously: 229ms
61823 lines
Asynchronously: 99ms
61823 lines
```

8. Wnioski

Patrząc na wyniki można stwierdzić, że wykonanie wariant asynchroniczny okazał się ponad dwa razy szybszy, dzięki zrównolegleniu operacji. Asynchroniczność to bardzo ważny aspekt języka JavaScript, ponieważ wykorzystuje się go głównie w aplikacjach internetowych, a tam jest potrzeba dużej responsywności, więc model wykonania jaki oferuje JavaScript jest atrakcyjny pod tym względem. Dodatkowo nowa wersja języka posiada słowa kluczowe *async*/*await*, które pozwalają uniknąć tzw. *.then() chains* oraz tworzyć kod bardziej czytelnym.

9. Bibliografia

- https://www.npmjs.com/package/walkdir
- https://nodejs.org/api/fs.html
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await
- https://caolan.github.io/async/v3/docs.html
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise