Blockchain for Industrial Engineers: Decentralized Application Development

บล็อกเชนสำหรับวิศวกรอุตสาหการ: การพัฒนาแอปพลิเคชันแบบกระจายศูนย์

Asynchronous Programming

Synchronous programming

- Normally, a given program's code runs straight along, with only one thing happening at once.
- If a function relies on the result of another function, it has to wait for the other function to finish and return.
    - And until that happens, the entire program is essentially stopped from the perspective of the user.
    - Also referred to as *blocking*.

Blocking code

```
function blocking() {
  for (let i = 0; i < 5e8; i++) {}
  console.log("Finish blocking calculation");
}

console.log("Start");
blocking();
console.log("Done");
```

Issue

- Synchronicity can lead to frustrating experience for a user.
- This is the basis of **asynchronous programming**.

Where do we find asynchronousity?

- It is up to the programming environment to provide you with APIs that allow you to run such tasks asynchronously.
- For all blockchain operations, most of the APIs are asynchronous.
    - *(Unfortunately…)*

Asynchronous javascript

`setTimeout` API

```
setTimeout(() => console.log("I waited for 3 seconds."), 3000);
// Or
setTimeout(() => {
  console.log("I waited for 3 seconds.");
}, 3000);
```

```javascript
function notblocking() {
  setTimeout(() => {
    console.log("Finish non-blocking calculation");
  }, 3000);
}

console.log("Start");
notblocking();
console.log("Done");
```

Asynchronous programming with Promises

`fetch`

- The Fetch API is a modern interface that allows you to make HTTP requests to servers from web browsers.
- The `fetch()` method is available in the global scope that instructs the web browsers to send a request to a URL.
- Let's `fetch` information from Star Wars API

`fetch`

```
const result = fetch("https://swapi.dev/api/people/1");

console.log(result); // Promise { <state>: "pending" }
```

- The `fetch()` method returns a `Promise`.
- But what is a **Promise**?

Promise (def. 1)

- A `Promise` allows you to defer further actions until after a previous action has completed, or respond to its failure.
- This is useful for setting up a sequence of async operations to work correctly.

Promise (def. 2)

- A `Promise` is an object that represents an intermediate state of an operation — in effect, a promise that a result of some kind will be returned at some point in the future.
- There is no guarantee of exactly when the operation will complete and the result will be returned but there is a guarantee that when the **result is available**, or **the promise fails**.
- You can then write the code that will be executed in order to do something else with a successful result, or to gracefully handle a failure case.

Promise (def. 3)

A `Promise` is in one of these states:

- `pending`: initial state, neither fulfilled nor rejected.
- `fulfilled`: meaning that the operation was completed successfully.
- `rejected`: meaning that the operation failed.

Exploring the states of a Promise

(Use firefox console)

```
new Promise(() => {}); //Promise { <state>: "pending" }s

Promise.resolve(); // Promise { <state>: "fulfilled", <value>: undefined }

Promise.resolve("I waited."); // Promise { <state>: "fulfilled", <value>: "I waited." }

Promise.reject(); // Promise { <state>: "rejected", <reason>: undefined }

Promise.reject("Wrong"); // Promise { <state>: "rejected", <reason>: "Wrong" }
```

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("foo"); //or reject('foo')
  }, 5000);
});
```

- You can keep typing `myPromise` in the Firefox console to see the state changed.

Creating `fake_fetch`

- Use `promise` and `setTimeout`

```
function fake_fetch(tag, isSuccess = true, wait = 2000) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (isSuccess) {
        resolve(`Success: ${tag}`);
      } else {
        reject(`Error: ${tag}`);
      }
    }, wait);
  });
}
```

Using `fake_fetch`

```
console.log("Start");
fake_fetch("R1").then((tag) => {
  console.log(tag);
});
```

Chaining multiple `fake_fetch`

```
console.log("Start");
fake_fetch("R1")
  .then((tag) => {
    console.log(tag);
    return fake_fetch("R2");
  })
  .then((tag) => {
    console.log(tag);
  });
```

Compared with

```
console.log("Start");
fake_fetch("R1").then((tag) => console.log(tag));
fake_fetch("R2").then((tag) => console.log(tag));
```

With error handling

```
console.log("Start");
fake_fetch("R1")
  .then((tag) => {
    console.log(tag);
    return fake_fetch("R2", false);
  })
  .then((tag) => {
    console.log(tag);
  })
  .catch((tag) => {
    console.log(tag);
  });
```

`async` and `await`

- An `async` function is a function declared with the `async` keyword, and the `await` keyword is permitted.
- The `async` and `await` keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

```
async function call_fetch() {
  const tag1 = await fake_fetch("R1");
  console.log(tag1);
  const tag2 = await fake_fetch("R2");
  console.log(tag2);
}

console.log("Start");
call_fetch();
```

With error handling

```
async function call_fetch() {
  try {
    const tag1 = await fake_fetch("R1");
    console.log(tag1);
    const tag2 = await fake_fetch("R2", false);
    console.log(tag2);
  } catch (err) {
    console.log(err);
  }
}

console.log("Start");
call_fetch();
```