Apprendre la syntaxe JavaScript : Promesses

États d'une promesse JavaScript

Un objet JavaScript `Promise` peut être dans l'un des trois états suivants : `pending`, `resolved` ou `rejected`. Tant que la valeur n'est pas encore disponible, le `Promise` reste dans l'`pending` état. Ensuite, il passe à l'un des deux états : `resolved` ou `rejected`.
Une `resolved` promesse est synonyme d'un accomplissement réussi. Suite à des erreurs, la promesse peut aller en l'`rejected` état.
Dans l'exemple de bloc de code, si l'état `Promise` est activé `resolved`, le premier paramètre contenant une fonction de rappel de la `.then()` méthode imprimera la valeur résolue. Sinon, une alerte s'affichera.

```
const promesse = nouvelle promesse ( (
résoudre , rejeter ) => {
  const res = vrai ;
  // Une opération asynchrone.
  si ( res ) {
    résoudre ( 'Résolu !' ) ;
  }
  sinon {
    rejeter ( Erreur ( 'Erreur' ) ) ;
  }
} ) ;

promesse . then ( ( res ) => console .
log ( res ) , ( err ) => alert ( err ) )
;
```

La `.catch()`méthode de traitement des rejets

La fonction passée comme deuxième argument à une `.then()` méthode d'un objet promesse est utilisée lorsque la promesse est rejetée. Une alternative à cette approche consiste à utiliser la `.catch()` méthode JavaScript de l'objet promise. Les informations pour le rejet sont disponibles pour le gestionnaire fourni dans la `.catch()` méthode.

```
const promesse = nouvelle promesse ( (
résoudre , rejeter ) => {
  setTimeout ( ( ) => {
    rejeter ( Erreur ( 'Promesse rejetée
sans condition.' ) ) ;
  } , 1000 ) ;
} ) ;

promesse . alors ( ( res ) => {
  consoler . log ( valeur ) ;
} ) ;

promesse . attraper ( ( erreur ) => {
  alerte ( erreur ) ;
} ) ;
```

Promesse JavaScript.all()

The JavaScript `Promise.all()` method can be used to execute multiple promises in parallel. The function accepts an array of promises as an argument. If all of the promises in the argument are resolved, the promise returned from `Promise.all()` will resolve to an array containing the resolved values of all the promises in the order of the initial array. Any rejection from the list of promises will cause the greater promise to be rejected.
In the code block, `3` and `2` will be printed respectively even though `promise1` will be resolved after `promise2`.

```
const promise1 = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve(3);
  }, 300);
});
const promise2 = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve(2);
  }, 200);
});

Promise.all([promise1,
promise2]).then((res) => {
  console.log(res[0]);
  console.log(res[1]);
});
```

Executor function of JavaScript Promise object

A JavaScript promise's executor function takes two functions as its arguments. The first parameter represents the function that should be called to resolve the promise and the other one is used when the promise should be rejected. A `Promise` object may use any one or both of them inside its executor function.
In the given example, the promise is always resolved unconditionally by the `resolve` function. The `reject` function could be used for a rejection.

```
const executorFn = (resolve, reject) => {
  resolve('Resolved!');
};

const promise = new Promise(executorFn);
```

.then() method of a JavaScript Promise object

The `.then()` method of a JavaScript `Promise` object can be used to get the eventual result (or error) of the asynchronous operation.
`.then()` accepts two function arguments. The first handler supplied to it will be called if the promise is resolved. The second one will be called if the promise is rejected.

```
const promise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve('Result');
  }, 200);
});

promise.then((res) => {
  console.log(res);
}, (err) => {
```

```
  alert(err);
});
```

setTimeout()

`setTimeout()` is an asynchronous JavaScript function that executes a code block or evaluates an expression through a callback function after a delay set in milliseconds.

```
const loginAlert = () =>{
  alert('Login');
};

setTimeout(loginAlert, 6000);
```

Avoiding nested `Promise` and `.then()`

In JavaScript, when performing multiple asynchronous operations in a sequence, promises should be composed by chaining multiple `.then()` methods. This is better practice than nesting.
Chaining helps streamline the development process because it makes the code more readable and easier to debug.

```
const promise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve('*');
  }, 1000);
});

const twoStars = (star) => {
  return (star + star);
};

const oneDot = (star) => {
  return (star + '.');
};

const print = (val) => {
  console.log(val);
};

// Chaining them all together
promise.then(twoStars).then(oneDot).then(
print);
```

Creating a Javascript Promise object

An instance of a JavaScript `Promise` object is created using the `new` keyword.
The constructor of the `Promise` object takes a function, known as the *executor function*, as the argument. This function is responsible for resolving or rejecting the promise.

```
const executorFn = (resolve, reject) => {
  console.log('The executor function of
the promise!');
};

const promise = new Promise(executorFn);
```

The `Promise` Object

A `Promise` is an object that can be used to get the outcome of an asynchronous operation when that result is not instantly available.
Since JavaScript code runs in a non-blocking manner, promises become essential when we have to wait for some asynchronous operation without holding back the execution of the rest of the code.

Chaining multiple .then() methods

The `.then()` method returns a Promise, even if one or both of the handler functions are absent. Because of this, multiple `.then()` methods can be chained together. This is known as composition.
In the code block, a couple of `.then()` methods are chained together. Each method deals with the resolved value of their respective promises.

```
const promise = new Promise(resolve =>
setTimeout(() => resolve('dAlan'), 100));

promise.then(res => {
  return res === 'Alan' ?
Promise.resolve('Hey Alan!') :
Promise.reject('Who are you?')
}).then((res) => {
  console.log(res)
}, (err) => {
  alert(err)
});
```