Promise Chaining

Summary: in this tutorial, you will learn about the JavaScript promise chaining pattern that chains the promises to execute asynchronous operations in sequence.

Introduction to the JavaScript promise chaining

Sometimes, you want to execute two or more related asynchronous operations, where the next operation starts with the result from the previous one. For example:

First, create a new promise that resolves to the number 10 after 3 seconds:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});
```

> Note that the `setTimeout()` function simulates an asynchronous operation.

Then, invoke the `then()` method of the promise:

```
p.then((result) => {
    console.log(result);
    return result * 2;
});
```

The callback passed to the `then()` method executes once the promise is resolved. In the callback, we show the result of the promise and return a new value multiplied by two (`result*2`).

Because the `then()` method returns a new `Promise` with a value resolved to a value, you can call the `then()` method on the return `Promise` like this:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});

p.then((result) => {
    console.log(result);
    return result * 2;
}).then((result) => {
    console.log(result);
    return result * 3;
});
```

Output:

```
10
20
```

In this example, the return value in the first `then()` method is passed to the second `then()` method. You can keep calling the `then()` method successively as follows:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});

p.then((result) => {
    console.log(result); // 10
    return result * 2;
}).then((result) => {
    console.log(result); // 20
    return result * 3;
```

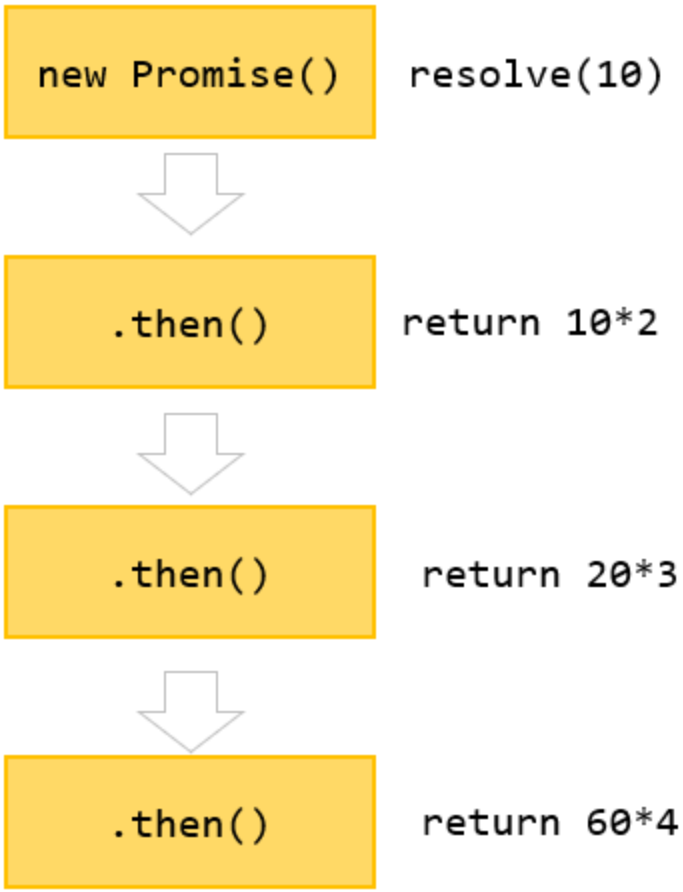
```
}).then((result) => {
    console.log(result); // 60
    return result * 4;
});
```

Output:

```
10
20
60
```

The way we call the `then()` method like this is often referred to as a **promise chain**.

The following picture illustrates the promise chain:



Multiple handlers for a promise

When you call the `then()` method multiple times on a promise, it is not the promise chaining. For example:

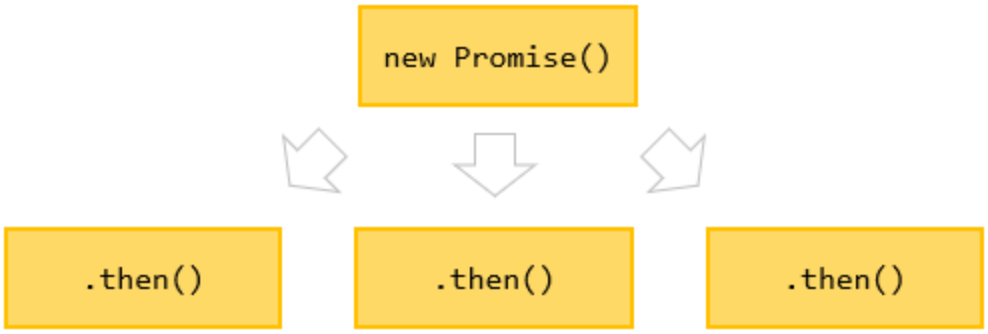
```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});

p.then((result) => {
    console.log(result); // 10
    return result * 2;
})

p.then((result) => {
    console.log(result); // 10
    return result * 3;
})

p.then((result) => {
    console.log(result); // 10
    return result * 4;
});
```

Output:

```
10
10
10
```

In this example, we have multiple handlers for one promise. These handlers have no relationships. Also, they execute independently and don't pass the result from one to another like the promise chain above.

The following picture illustrates a promise that has multiple handlers:



In practice, you will rarely use multiple handlers for one promise.

Returning a Promise

When you return a value in the `then()` method, the `then()` method returns a new `Promise` that immediately resolves to the return value.

Also, you can return a new promise in the `then()` method, like this:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});

p.then((result) => {
    console.log(result);
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(result * 2);
        }, 3 * 1000);
    });
}).then((result) => {
    console.log(result);
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(result * 3);
        }, 3 * 1000);
```

```
    });
}).then(result => console.log(result));
```

Output:

```
10
20
60
```

This example shows 10, 20, and 60 after every 3 seconds. This code pattern allows you to execute some tasks in sequence.

The following modified the above example:

```
function generateNumber(num) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(num);
    }, 3 * 1000);
  });
}

generateNumber(10)
  .then((result) => {
    console.log(result);
    return generateNumber(result * 2);
  })
  .then((result) => {
    console.log(result);
    return generateNumber(result * 3);
  })
  .then((result) => console.log(result));
```

Promise chaining syntax

Sometimes, you have multiple asynchronous tasks you want to execute in sequence. In addition, you need to pass the result of the previous step to the next one. In this case, you can use the following

syntax:

```
step1()
    .then(result => step2(result))
    .then(result => step3(result))
    ...
```

If you need to pass the result of the previous task to the next one without passing the result, you use this syntax:

```
step1()
    .then(step2)
    .then(step3)
    ...
```

Suppose that you want to perform the following asynchronous operations in sequence:

- First, get the user from the database.
- Second, get the services of the selected user.
- Third, calculate the service cost from the user's services.

The following functions illustrate the three asynchronous operations:

```
function getUser(userId) {
    return new Promise((resolve, reject) => {
        console.log('Get the user from the database.');
        setTimeout(() => {
            resolve({
                userId: userId,
                username: 'admin'
            });
        }, 1000);
    })
}

function getServices(user) {
```

```
    return new Promise((resolve, reject) => {
        console.log(`Get the services of ${user.username} from the API.`);
        setTimeout(() => {
            resolve(['Email', 'VPN', 'CDN']);
        }, 3 * 1000);
    });
}

function getServiceCost(services) {
    return new Promise((resolve, reject) => {
        console.log(`Calculate the service cost of ${services}.`);
        setTimeout(() => {
            resolve(services.length * 100);
        }, 2 * 1000);
    });
}
```

The following uses the promises to serialize the sequences:

```
getUser(100)
    .then(getServices)
    .then(getServiceCost)
    .then(console.log);
```

Output

```
Get the user from the database.
Get the services of admin from the API.
Calculate the service cost of Email,VPN,CDN.
300
```

> Note that ES2017 introduced the `async` / `await` that helps you write code that is cleaner than using the promise chaining technique.

In this tutorial, you have learned about the promise chain that executes multiple asynchronous tasks in sequence.

Quiz