async function

Tags	Tools
Created time	@April 23, 2023 10:58 PM
Last edited time	@April 23, 2023 11:11 PM

```
const myPromise = new Promise((resolve, reject) => {
  // Do some asynchronous operation, such as fetching data from an API
  // If the operation is successful, call resolve with the result
  // If the operation fails, call reject with an error message or object
});
```

```
async function wait() {
  console.log('Waiting...');
  await new Promise(resolve => setTimeout(resolve, 1000));
  console.log('Done!');
}

wait();

// In this example, the wait function logs a message to the console,
// then waits for 1 second using await and a Promise that resolves
// after 1 second, and finally logs another message to the console.

// Overall, async and await make it easier to write and understand
// asynchronous code in JavaScript.
```

```
async function getData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  return data;
}
```

If there are no `await` keywords in the `async` function, the function will behave like a regular synchronous function. The function will execute from start to finish without any pausing or waiting for promises to resolve.

Asynchronous JavaScript is a programming paradigm that allows JavaScript code to execute non-blocking operations, which means that code can continue to run while waiting for certain operations to complete.

In other words, asynchronous code does not wait for one operation to finish before moving on to the next one. Instead, it registers a callback function to be executed when the operation is complete.

Here's an example to illustrate this concept:

```
javascriptCopy code
console.log('Before');
setTimeout(() => {
  console.log('After');
}, 2000);
console.log('During');
```

In this example, we use the `setTimeout()` function to simulate a time-consuming operation. The first `console.log()` statement will be executed immediately, printing 'Before' to the console.

Then, the `setTimeout()` function is called with a callback function that will be executed after a delay of 2000 milliseconds (2 seconds).

However, the `setTimeout()` function is non-blocking, so the program execution continues to the next line of code without waiting for the 2-second delay to expire.

The second `console.log()` statement is then executed, printing 'During' to the console.

After the 2-second delay has passed, the callback function passed to `setTimeout()` is executed, printing 'After' to the console.

The output of the above code will be:
Before
During
After

As you can see, the 'After' message is printed to the console after the 'During' message, even though it was registered to be executed after a delay. This is the power of asynchronous programming in JavaScript.

```
console.log('Before');
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
console.log('During');
```

In this example, we use the `fetch()` function to make an HTTP request to a JSON API.

Like the `setTimeout()` function, `fetch()` is also non-blocking, which means that the program execution continues to the next line of code without waiting for the HTTP request to complete.

After calling `fetch()`, we chain a series of `.then()` methods to handle the response data once it's available.

The first `.then()` method converts the response into a JSON object, and the second `.then()` method logs the JSON object to the console.

We also chain a `.catch()` method to handle any errors that may occur during the HTTP request.

The output of the above code will be:

```
yamlCopy code
Before
During
{ userId: 1, id: 1, title: 'delectus aut autem', completed: false }
```

As you can see, the 'During' message is printed to the console before the JSON object is logged, even though the JSON object is only available after the HTTP request completes. This is because the `fetch()` function is asynchronous, and the program execution continues while the HTTP request is in progress.

```
function delay(milliseconds) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve();
    }, milliseconds);
  });
}

console.log('Before');
delay(2000)
  .then(() => console.log('After'))
  .catch(error => console.error(error));
console.log('During');
```

In this example, we define a `delay()` function that returns a Promise that resolves after a certain number of milliseconds.

Inside the `delay()` function, we use the `setTimeout()` function to delay the resolution of the Promise by the specified number of milliseconds.

In the main program, we call the `delay()` function with a delay of 2000 milliseconds (2 seconds).

After calling `delay()`, we chain a `.then()` method to log 'After' to the console once the Promise is resolved.

We also chain a `.catch()` method to handle any errors that may occur during the Promise resolution.

The output of the above code will be:

```
mathematicaCopy code
Before
During
After
```

As you can see, the 'After' message is printed to the console after the 'During' message, even though the `delay()` function takes 2 seconds to resolve the Promise. This is because the `delay()` function returns a Promise, which allows us to use the `.then()` method to register a callback function to be executed when the Promise is resolved.