



Факультет компьютерных наук

Департамент программной
инженерии

Москва
2025

Асинхронный JS HTTP-запросы

Однопоточность
Стек вызовов

Очередь задач
Event Loop

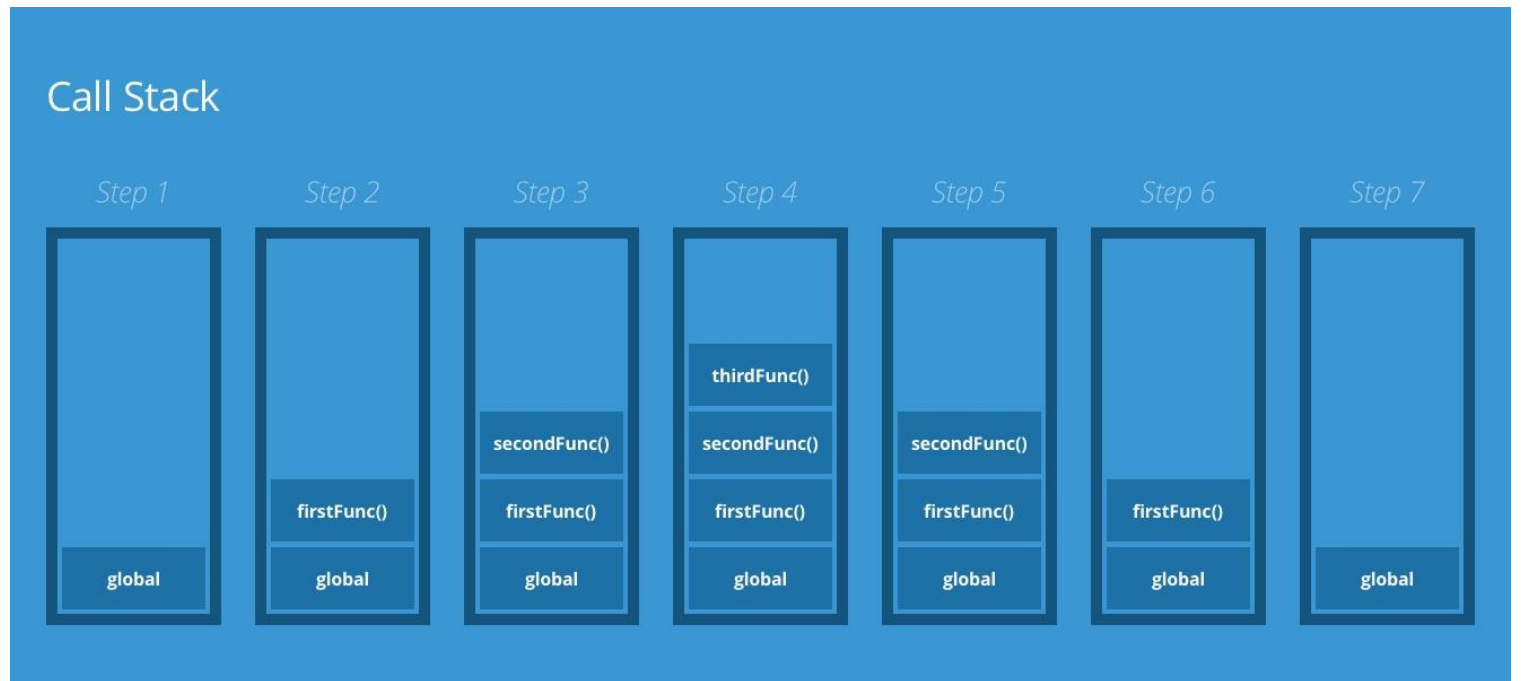


Стек вызовов



Стек вызовов

Стек вызовов (Call Stack) – механизм интерпретатора языка программирования, необходимый для отслеживания текущего своего местонахождения в скрипте во время его исполнения. С помощью стека вызовов интерпретатор понимает, где в коде он находится сейчас, какая функция или метод были или будут вызываться внутри какой области видимости.





```
1  function multiply(a, b) {  
2      return a * b;  
3  }  
4  
5  function square(x) {  
6      return multiply(x, x);  
7  }  
8  
9  function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack



```
1 function multiply(a, b) {  
2     return a * b;  
3 }  
4  
5 function square(x) {  
6     return multiply(x, x);  
7 }  
8  
9 function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack



```
1  function multiply(a, b) {  
2      return a * b;  
3  }  
4  
5  function square(x) {  
6      return multiply(x, x);  
7  }  
8  
9  function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack



```
1  function multiply(a, b) {  
2      return a * b;  
3  }  
4  
5  function square(x) {  
6      return multiply(x, x);  
7  }  
8  
9  function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack




```
1  function multiply(a, b) {  
2      return a * b;  
3  }  
4  
5  function square(x) {  
6      return multiply(x, x);  
7  }  
8  
9  function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack



```
1  function multiply(a, b) {  
2      return a * b;  
3  }  
4  
5  function square(x) {  
6      return multiply(x, x);  
7  }  
8  
9  function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack


printSquared(4);



```

1  function multiply(a, b) {
2      return a * b;
3  }
4
5  function square(x) {
6      return multiply(x, x);
7  }
8
9  function printSquared(x) {
10     const squared = square(x);
11     console.log(squared);
12 }
13
14 printSquared(4);

```

Call Stack

square(x);

printSquared(4);



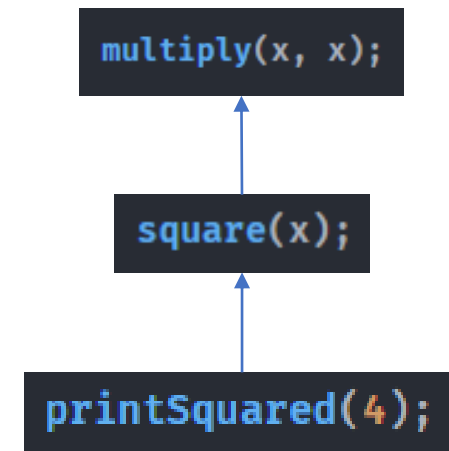


```

1  function multiply(a, b) {
2      return a * b;
3  }
4
5  function square(x) {
6      return multiply(x, x);
7  }
8
9  function printSquared(x) {
10     const squared = square(x);
11     console.log(squared);
12 }
13
14 printSquared(4);

```

Call Stack

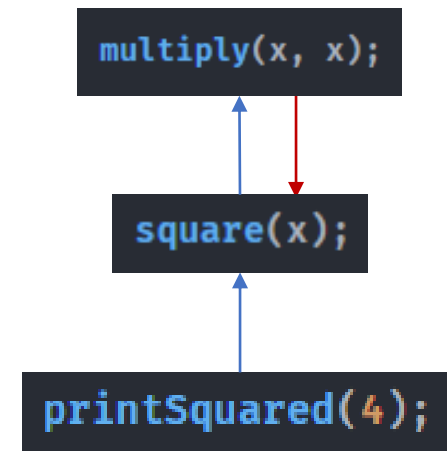


```

1  function multiply(a, b) {
2      return a * b;
3  }
4
5  function square(x) {
6      return multiply(x, x);
7  }
8
9  function printSquared(x) {
10     const squared = square(x);
11     console.log(squared);
12 }
13
14 printSquared(4);

```

Call Stack



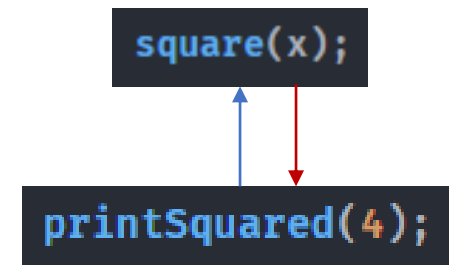


```

1  function multiply(a, b) {
2      return a * b;
3  }
4
5  function square(x) {
6      return multiply(x, x);
7  }
8
9  function printSquared(x) {
10     const squared = square(x);
11     console.log(squared);
12 }
13
14 printSquared(4);

```

Call Stack





```

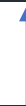
1  function multiply(a, b) {
2      return a * b;
3  }
4
5  function square(x) {
6      return multiply(x, x);
7  }
8
9  function printSquared(x) {
10     const squared = square(x);
11     console.log(squared);
12 }
13
14 printSquared(4);

```

Call Stack

`console.log(squared);`

`printSquared(4);`





```
1  function multiply(a, b) {  
2      return a * b;  
3  }  
4  
5  function square(x) {  
6      return multiply(x, x);  
7  }  
8  
9  function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack

```
printSquared(4);
```



```
1  function multiply(a, b) {  
2      return a * b;  
3  }  
4  
5  function square(x) {  
6      return multiply(x, x);  
7  }  
8  
9  function printSquared(x) {  
10     const squared = square(x);  
11     console.log(squared);  
12 }  
13  
14 printSquared(4);
```

Call Stack



```
1 console.log('foo');  
2  
3 setTimeout(  
4     function () {  
5         console.log('bar');  
6     },  
7     1000  
8 );  
9  
10 console.log('baz');
```

Call Stack



```
1 console.log('foo');
2
3 setTimeout(
4     function () {
5         console.log('bar');
6     },
7     1000
8 );
9
10 console.log('baz');
```

Call Stack

Output



```
1 console.log('foo');  
2  
3 setTimeout(  
4     function () {  
5         console.log('bar');  
6     },  
7     1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

```
console.log('foo');
```

Output



```
1 console.log('foo');  
2  
3 setTimeout(  
4     function () {  
5         console.log('bar');  
6     },  
7     1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

Output

foo



```
1 console.log('foo');  
2  
3 setTimeout(  
4     function () {  
5         console.log('bar');  
6     },  
7     1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

```
setTimeout(fn, 1000)
```

Output

foo



```
1 console.log('foo');  
2  
3 setTimeout(  
4   function () {  
5     console.log('bar');  
6   },  
7   1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

```
setTimeout(fn, 1000)
```

Output

foo

Task Queue

```
anonymous();
```





```
1 console.log('foo');
2
3 setTimeout(
4     function () {
5         console.log('bar');
6     },
7     1000
8 );
9
10 console.log('baz');
```

Call Stack

Output

foo

Task Queue

anonymous();





```
1 console.log('foo');  
2  
3 setTimeout(  
4     function () {  
5         console.log('bar');  
6     },  
7     1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

```
console.log('baz');
```

Output

foo

Task Queue

```
anonymous();
```





```
1 console.log('foo');  
2  
3 setTimeout(  
4     function () {  
5         console.log('bar');  
6     },  
7     1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

Output

foo
baz

Task Queue

anonymous();





```
1 console.log('foo');  
2  
3 setTimeout(  
4   function () {  
5     console.log('bar');  
6   },  
7   1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

anonymous();

Output

foo
baz

Task Queue



```
1 console.log('foo');  
2  
3 setTimeout(  
4     function () {  
5         console.log('bar');  
6     },  
7     1000  
8 );  
9  
10 console.log('baz');
```

Call Stack

Output

foo
baz
bar

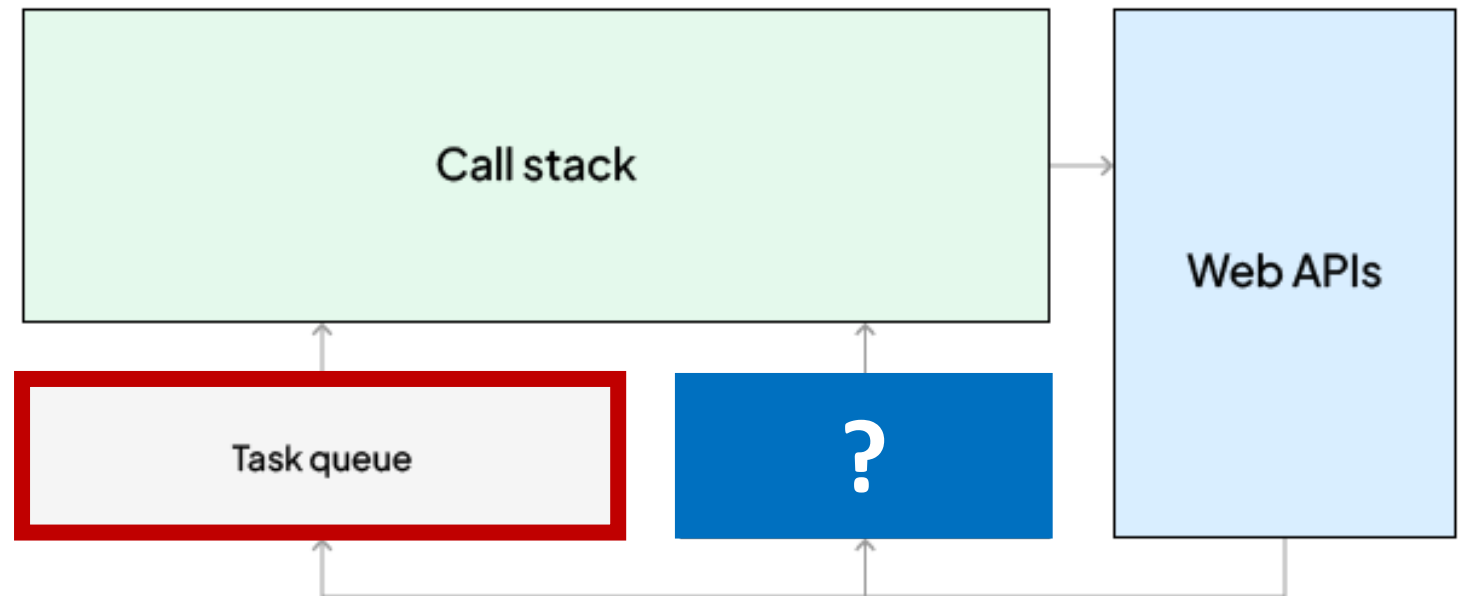
Task Queue

Очередь задач

Очередь задач (Task Queue) – механизм интерпретатора языка программирования, необходимый для хранения функций, вызов которых отложен во времени.

Функции, помещенные в очередь задач, ждут истечения своего таймера, а потом попадают в стек вызовов как только он становится пустым (т. е. когда основной поток выполнения скрипта завершается).

Функции, помещенные в очередь задач, называются **макрозадачами**.





Способы создать макрозадачу

- `setTimeout(fn [, delay, arg1, arg2, ...])`
- `setInterval(fn [, delay, arg1, arg2, ...])`
- `setImmediate(fn)`

Способы удалить макрозадачу

- `clearTimeout(timerId)`
- `clearInterval(timerId)`
- `clearImmediate(timerId)`



Способы создать макрозадачу

- `setTimeout(fn [, delay, arg1, arg2, ...])`
- `setInterval(fn [, delay, arg1, arg2, ...])`
- `setImmediate(fn)`
- `HTMLElement.addEventListener(event, fn)`

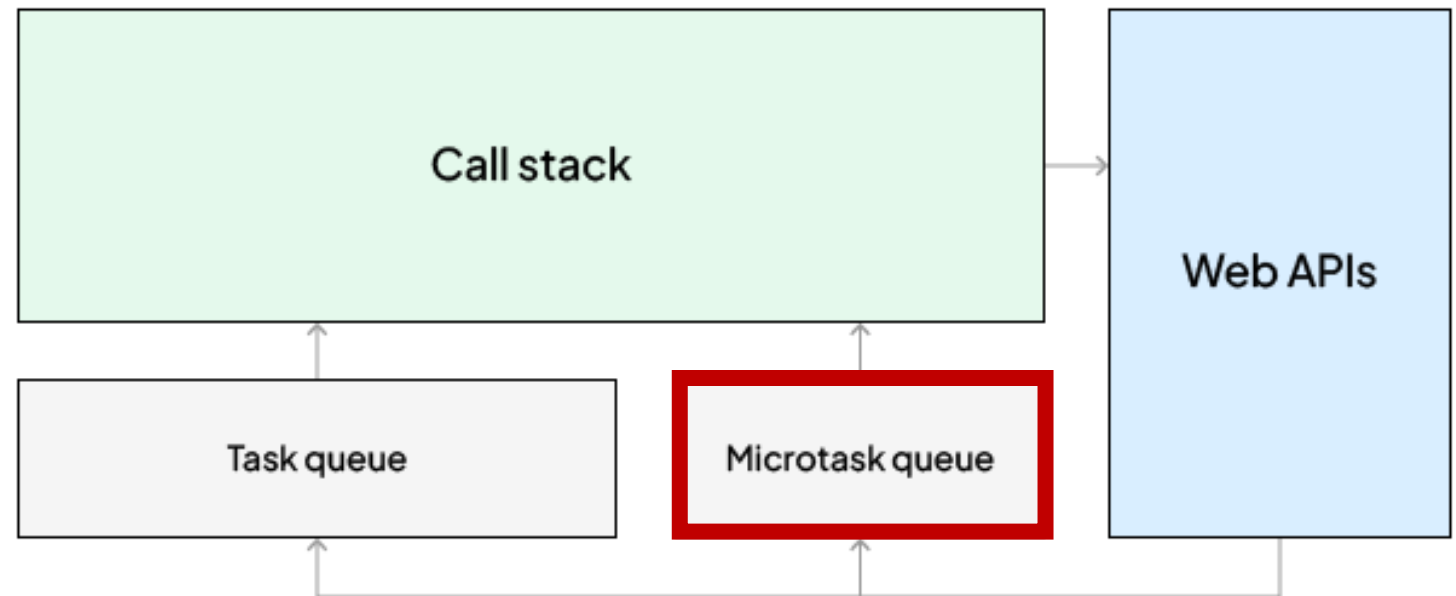
Способы удалить макрозадачу

- `clearTimeout(timerId)`
- `clearInterval(timerId)`
- `clearImmediate(timerId)`

Микрозадачи

Микрозадача – функция, которая также, как и макрозадача, отложена в исполнении по времени, но имеет более высокий приоритет исполнения, а также момент и логика ее выполнения зависит от результата работы кода, породившего микрозадачу.

Микрозадачи используются для выполнения запросов на сервер, при этом не тормозя основной поток, что обеспечивает отзывчивость пользовательского интерфейса при этом запросе, создавая «иллюзию» многопоточности.



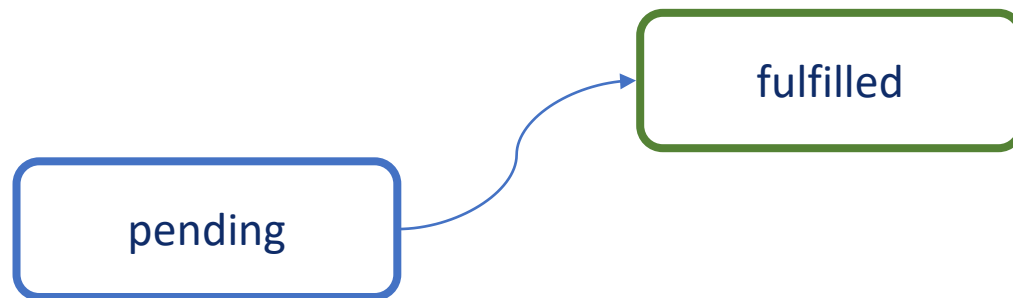


Промисы

Промис (Promise) – объект JavaScript, представляющий собой обертку для значения, неизвестного на момент создания промиса. Такой объект позволяет обрабатывать результат асинхронной операции таким образом, будто эта операция синхронная: вместо результата работы асинхронной операции возвращается «обещание» получить результат в будущем.



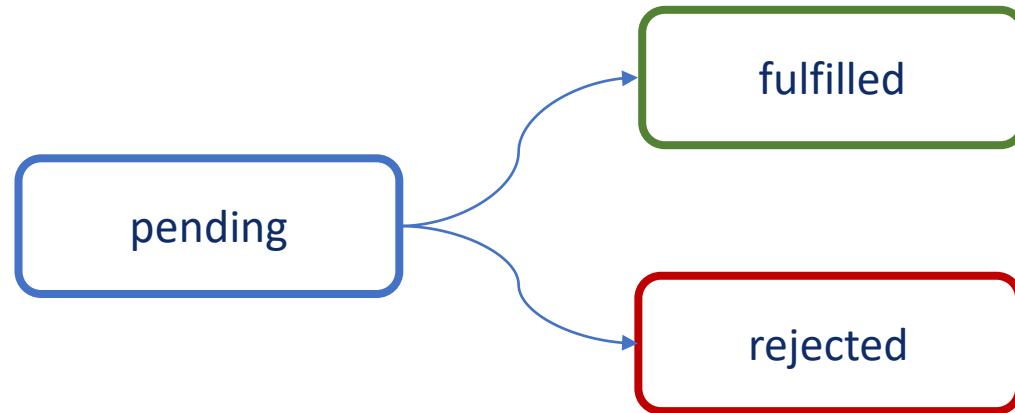
```
1 new Promise((resolve, reject) => {  
2     // some executions here  
3 });
```

```
1 new Promise((resolve, reject) => {  
2   resolve('Good result')  
3 });
```

```
> Promise.resolve('Good result')
```

```
< ▼ Promise {<fulfilled>: 'Good result'} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
    [[PromiseResult]]: "Good result"
```



```
1 new Promise((resolve, reject) => {  
2   reject('Bad result')  
3 });
```

```
> Promise.reject('Bad result')
```

```
< ▼ Promise {<rejected>: 'Bad result'} ⓘ
```

```
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "rejected"  
    [[PromiseResult]]: "Bad result"
```

```
✖ ► Uncaught (in promise) Bad result
```



```
1  // Promise.prototype.then();  
2  
3  Promise.resolve(10)  
4      .then(x => x * 2);  
5  // Promise { <fulfilled>: 20 }
```



```
1  Promise.resolve(10)
2    .then(x => x * 2)
3    .then(x => x / 5)
4    .then(x => x.toString())
5  // Promise { <fulfilled>: '4' }
```



```
1 Promise.resolve('foo')
2   .then(
3     x => {
4       throw 'error!';
5     }
6   )
7 // Promise { <rejected>: 'error!' }
```



```
1 Promise.reject('bar')
2   .then(
3     null,
4     err => {
5       throw 'error!';
6     }
7   )
8 // Promise { <rejected>: 'error!' }
```




```
1  Promise.reject('bar')  
2    .then(() => 'Надеюсь, всё будет хорошо')  
3  // Promise { <rejected>: 'bar' }
```

Если в then() попадает rejected-промис, а в then() только один аргумент, то then() игнорируется



```
1  Promise.resolve('foo')
2    .finally(() => doSomething())
3  // Promise { <fulfilled>: 'foo' }
4
5  //Promise.prototype.finally()
6  Promise.reject('bar')
7    .finally(() => doSomething())
8  // Promise { <rejected>: 'bar' }
```

```
1  Promise.all([
2    asyncProcessA(),
3    asyncProcessB()
4  ])
5    .then(([resultA, resultB]) => {
6      // сделать что-нибудь с
7      // resultA и resultB (только если все они fullfiled)
8    })
9
10 Promise.allSettled([
11   asyncProcessA(),
12   asyncProcessB()
13 ])
14   .then(([resultA, resultB]) => {
15     // сделать что-нибудь с
16     // resultA и resultB (вне зависимости от их состояния)
17   })
```

```
1  Promise.race([
2      asyncProcessA(),
3      asyncProcessB()
4  ])
5      .then(result => {
6          // result - результат
7          // самого быстрого из промисов
8      })
```



Промисы

Промис (Promise) – объект JavaScript, представляющий собой обертку для значения, неизвестного на момент создания промиса. Такой объект позволяет обрабатывать результат асинхронной операции таким образом, будто эта операция синхронная: вместо результата работы асинхронной операции возвращается «обещание» получить результат в будущем.

Код, помещенный внутрь конструктора, выполняется в основном потоке.

Все, что лежит в `then()`, `catch()` или `finally()`, является микрозадачей и выполняется раньше, чем макрозадача.



```
1 new Promise((resolve, reject) => {  
2   // some executions here  
3 });
```



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

Output

Task Queue

Microtask Queue



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

`console.log(1);`

Output

1

Task Queue

Microtask Queue



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

```
setTimeout(() =>
  console.log(2));
```

Output

1

Task Queue

Microtask Queue



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

Output

1

Task Queue

`console.log(2)`

Microtask Queue



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

Promise.resolve(3)

Output

1

Task Queue

console.log(2)

Microtask Queue



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

Output

1

Task Queue

`console.log(2)`

Microtask Queue

`.then(x => {
 console.log(x);
 return x;
})``.then(x => console.log(x));`



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

```
console.log(4);
```

Output

```
1
4
```

Task Queue

```
console.log(2)
```

Microtask Queue

```
.then(x => {
  console.log(x);
  return x;
})
```

```
.then(x => console.log(x));
```



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

Output

1
4

Task Queue

`console.log(2)`

Microtask Queue

`.then(x => {
 console.log(x);
 return x;
})``.then(x => console.log(x));`



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

```
setTimeout(() =>
  console.log(5),
  0
);
```

Output

1
4

Task Queue

```
console.log(2)
```

Microtask Queue

```
.then(x => {
  console.log(x);
  return x;
})
```

```
.then(x => console.log(x));
```



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

Output

1
4

Task Queue

`console.log(2)``console.log(5),`

Microtask Queue

`.then(x => {
 console.log(x);
 return x;
})``.then(x => console.log(x));`



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

```
console.log(6);
```

Output

```
1
4
6
```

Task Queue

```
console.log(2)
```

```
console.log(5),
```

Microtask Queue

```
.then(x => {
  console.log(x);
  return x;
})
```

```
.then(x => console.log(x));
```



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

```
.then(x => {
  console.log(x);
  return x;
})
```

Output

1 3
4
6

Task Queue

```
console.log(2)
```

```
console.log(5),
```

Microtask Queue

```
.then(x => console.log(x));
```



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

```
.then(x => console.log(x));
```

Output

1	3
4	3
6	

Task Queue

```
console.log(2)
```

```
console.log(5),
```

Microtask Queue



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

`console.log(2)`

Output

1	3
4	3
6	2

Task Queue

`console.log(5),`

Microtask Queue



```
1 console.log(1);
2
3 setTimeout(() =>
4   console.log(2));
5
6 Promise.resolve(3)
7   .then(x => {
8     console.log(x);
9     return x;
10  })
11   .then(x => console.log(x));
12
13 console.log(4);
14
15 setTimeout(() =>
16   console.log(5),
17   0
18 );
19
20 console.log(6);
```

Call Stack

`console.log(5),`

Output

1	3
4	3
6	2
	5

Task Queue

Microtask Queue



```
1  const f = () => {  
2      return Promise.resolve(1);  
3  }  
4  f().then(console.log); // 1  
5
```



```
1  const f = async () => {  
2      return 1;  
3  }  
4  f().then(console.log); // 1
```



```

1  const f = async () => {
2      const result1 = await someRequest();
3      const result2 = await someRequest(result1);
4
5      return someRequest(result2);
6  }
7
8  // Все равно, что
9
10 const f = () => {
11     return someRequest() // это промис
12         .then(result1 => someRequest(result1)) // и это промис
13         .then(result2 => someRequest(result2)); // и это
14 }

```



HTTP-запросы



Первый способ выполнить запрос: XHR

XMLHttpRequest – API, который предоставляет клиенту функциональность для обмена данными между клиентом и сервером. Данный API предоставляет простой способ получения данных по ссылке без перезагрузки страницы. Это позволяет обновлять только часть веб-страницы не прерывая пользователя. XMLHttpRequest используется в AJAX запросах и особенно в single-page приложениях.

Пример использования



```
1  const requestObj = new XMLHttpRequest();
```



Второй способ выполнить запрос...





Задание на семинар: создание страницы со списком задач

Необходимо загрузить от 1 до 50 задач с сайта <https://jsonplaceholder.typicode.com/todos> и отобразить полученный список на странице.

Требования:

1. Реализовать **синхронную** функцию `fetchTodos(count)`, которая будет возвращать промис с результатом работы запроса. Внутри промиса нужно объявить `XMLHttpRequest` и выполнить GET-запрос с параметром `_limit`, равным `count`. Если запрос выполнен успешно (код 200), то нужно резолвить результат запроса, иначе реджектить ошибку
2. Реализовать **асинхронную** функцию `loadTodos()`, которая будет отображать на странице результат запроса. Функция умеет обрабатывать ошибку и показывать на странице, что с запросом что-то не так. `loadTodos` вызывает `fetchTodos`
3. Реализовать на странице поле для ввода количества запрашиваемых элементов (от 1 до 50) и кнопку, нажатие которой выполняет запрос.
4. Реализовать отображение полученного списка на странице: текст выполненных задач окрашен в зеленый цвет, невыполненных – в красный.



Спасибо за семинар!

Вы прекрасны!

Пройдя [эту](#) форму, вы можете получить дополнительный балл к семинару.

[Сюда](#) вы можете загрузить свои 5ые домашние задания



