Раздел
Промисы, async/await

Навигация по уроку
Колбэк в колбэке
Перехват ошибок
Адская пирамида вызовов

Задачи (1)
Комментарии

Поделиться

Редактировать на GitHub

Введение: колбэки

Многие действия в JavaScript *асинхронные*.

Например, рассмотрим функцию `loadScript(src)`:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Эта функция загружает на страницу новый скрипт. Когда в тело документа добавится конструкция `<script src="…">`, браузер загрузит скрипт и выполнит его.

Вот пример использования этой функции:

```
// загрузит и выполнит скрипт
loadScript('/my/script.js');
```

Такие функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.

Если после вызова `loadScript(…)` есть какой-то код, то он не будет ждать, пока скрипт загрузится.

```
loadScript('/my/script.js');
// код, написанный после вызова функции loadScript,
// не будет дожидаться полной загрузки скрипта
// ...
```

Мы хотели бы использовать новый скрипт, как только он будет загружен. Скажем, он объявляет новую функцию, которую мы хотим выполнить.

Но если мы просто вызовем эту функцию после `loadScript(…)`, у нас ничего не выйдет:

```
loadScript('/my/script.js'); // в скрипте есть "functio

newFunction(); // такой функции не существует!
```

Действительно, ведь у браузера не было времени загрузить скрипт. Сейчас функция `loadScript` никак не позволяет отследить момент загрузки. Скрипт загружается, а потом выполняется. Но нам нужно точно знать, когда это произойдёт, чтобы использовать функции и переменные из этого скрипта.

Давайте передадим функцию `callback` вторым аргументом в `loadScript`, чтобы вызвать её, когда скрипт загрузится:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);


```

```
  document.head.append(script);
}
```

Раздел

Промисы, async/await

Навигация по уроку

Колбэк в колбэке

Перехват ошибок

Адская пирамида вызовов

Задачи (1)

Комментарии

Теперь, если мы хотим вызвать функцию из скрипта, нужно делать это в колбэке:

```
loadScript('/my/script.js', function() {
  // эта функция вызовется после того, когда загрузится
  newFunction(); // теперь всё работает
  ...
});
```

Смысл такой: вторым аргументом передаётся функция (обычно анонимная), которая выполняется по завершении действия.

Возьмём для примера реальный скрипт с библиотекой функций:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/loda
  alert(`Здорово, скрипт ${script.src} загрузился`);
  alert( _ ); // функция, объявленная в загруженном скр
});
```

Такое написание называют асинхронным программированием с использованием колбэков. В функции, которые выполняют какие-либо асинхронные операции, передаётся аргумент `callback` — функция, которая будет вызвана по завершению асинхронного действия.

Мы поступили похожим образом в `loadScript`, но это, конечно, распространённый подход.

Колбэк в колбэке

Как нам загрузить два скрипта один за другим: сначала первый, а за ним второй?

Первое, что приходит в голову, вызвать `loadScript` ещё раз уже внутри колбэка, вот так:

```
loadScript('/my/script.js', function(script) {

  alert(`Здорово, скрипт ${script.src} загрузился, загр

  loadScript('/my/script2.js', function(script) {
    alert(`Здорово, второй скрипт загрузился`);
  });

});
```

Когда внешняя функция `loadScript` выполнится, вызовется та, что внутри колбэка.

А что если нам нужно загрузить ещё один скрипт?..

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {


```

Раздел
Промисы, async/await
Навигация по уроку
Колбэк в колбэке
Перехват ошибок
Адская пирамида вызовов
Задачи (1)
Комментарии
Поделиться
Редактировать на GitHub

```
    loadScript('/my/script3.js', function(script) {
      // ...и так далее, пока все скрипты не будут загр
    });

  })

});
```

Каждое новое действие мы вынуждены вызывать внутри колбэка. Этот вариант подойдёт, когда у нас одно-два действия, но для большего количества уже не удобно. Альтернативные подходы мы скоро разберём.

Перехват ошибок

В примерах выше мы не думали об ошибках. А что если загрузить скрипт не удалось? Колбэк должен уметь реагировать на возможные проблемы.

Ниже улучшенная версия `loadScript`, которая умеет отслеживать ошибки загрузки:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Не удалось

  document.head.append(script);
}
```

Мы вызываем `callback(null, script)` в случае успешной загрузки и `callback(error)`, если загрузить скрипт не удалось.

Живой пример:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // обрабатываем ошибку
  } else {
    // скрипт успешно загружен
  }
});
```

Опять же, подход, который мы использовали в `loadScript`, также распространён и называется «колбэк с первым аргументом-ошибкой» («error-first callback»).

Правила таковы:

1. Первый аргумент функции `callback` зарезервирован для ошибки. В этом случае вызов выглядит вот так: `callback(err)`.
2. Второй и последующие аргументы — для результатов выполнения. В этом случае вызов выглядит вот так: `callback(null, result1, result2…)`.

Одна и та же функция `callback` используется и для информирования об ошибке, и для передачи результатов.

Адская пирамида вызовов

На первый взгляд это рабочий способ написания асинхронного кода. Так и есть. Для одного или двух вложенных вызовов всё выглядит нормально.

Но для нескольких асинхронных действий, которые нужно выполнить друг за другом, код выглядит вот так:

```
loadScript('1.js', function(error, script) {

```

Раздел

Промисы, async/await

Навигация по уроку

Колбэк в колбэке

Перехват ошибок

Адская пирамида вызовов

Задачи (1)

Комментарии

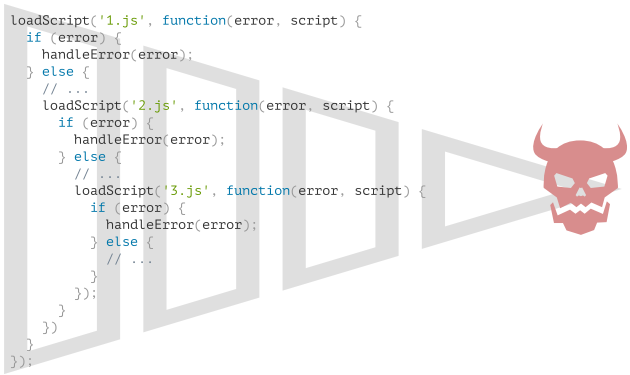

```
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...и так далее, пока все скрипты не буду
          }
        });

      }
    })
  }
});
```

В примере выше:

1. Мы загружаем `1.js`. Продолжаем, если нет ошибок.
2. Мы загружаем `2.js`. Продолжаем, если нет ошибок.
3. Мы загружаем `3.js`. Продолжаем, если нет ошибок. И так далее `(*)`.

Чем больше вложенных вызовов, тем наш код будет иметь всё большую вложенность, которую сложно поддерживать, особенно если вместо `...` у нас код, содержащий другие цепочки вызовов, условия и т.д.

Иногда это называют «адом колбэков» или «адской пирамидой колбэков».



Пирамида вложенных вызовов растёт вправо с каждым асинхронным действием. В итоге вы сами будете путаться, где что есть.

Такой подход к написанию кода не приветствуется.

Мы можем попытаться решить эту проблему, изолируя каждое действие в отдельную функцию, вот так:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
```

Раздел
Промисы, async/await
Навигация по уроку
Колбэк в колбэке
Перехват ошибок
Адская пирамида вызовов
Задачи (1)
Комментарии
Поделиться
Редактировать на GitHub

```
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...и так далее, пока все скрипты не будут загруж
  }
};
```

Заметили? Этот код делает всё то же самое, но вложенность отсутствует, потому что все действия вынесены в отдельные функции.

Код абсолютно рабочий, но кажется разорванным на куски. Его трудно читать, вы наверняка заметили это. Приходится прыгать глазами между кусками кода, когда пытаешься его прочесть. Это неудобно, особенно, если читатель не знаком с кодом и не знает, что за чем следует.

Кроме того, все функции `step*` одноразовые, и созданы лишь только, чтобы избавиться от «адской пирамиды вызовов». Никто не будет их переиспользовать где-либо ещё. Таким образом, мы, кроме всего прочего, засоряем пространство имён.

Нужно найти способ получше.

К счастью, такие способы существуют. Один из лучших — использовать промисы, о которых рассказано в следующей главе.

Задачи

Анимация круга с помощью колбэка

В задаче Анимированный круг находится код для анимации появления круга.

Давайте представим, что теперь нам нужен не просто круг, а круг с сообщением внутри. И сообщение должно появляться *после* анимации (когда круг достигнет своих размеров), иначе это будет некрасиво.

В том решении функция `showCircle(cx, cy, radius)` рисовала круг, но способа узнать, что всё нарисовано, не было.

Поэтому добавим в параметры колбэк: `showCircle(cx, cy, radius, callback)`, который выполним, когда анимация будет завершена. Функция `callback` будет добавлять в наш круг `<div>` элемент с сообщением.

Посмотрите пример:

```
showCircle(150, 150, 100, div => {
  div.classList.add('message-ball');
  div.append("Hello, world!");
});
```

Демо:

Click me

Возьмите за основу решение задачи Анимированный круг.

решение

Раздел

Промисы, async/await

Навигация по уроку

Колбэк в колбэке

Перехват ошибок

Адская пирамида вызовов

Задачи (1)

Комментарии

Поделиться

Редактировать на GitHub

Комментарии

перед тем как писать...