

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение


Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Продвинутая работа с функциями](#) 19-го августа 2019

Привязка контекста к функции

При передаче методов объекта в качестве колбэков, например для `setTimeout`, возникает известная проблема – потеря `this`.

В этой главе мы посмотрим, как её можно решить.

Потеря «this»

Мы уже видели примеры потери `this`. Как только метод передаётся отдельно от объекта – `this` теряется.

Вот как это может произойти в случае с `setTimeout`:

```
1 let user = {
2   firstName: "Вася",
3   sayHi() {
4     alert(`Привет, ${this.firstName}!`);
5   }
6 };
7
8 setTimeout(user.sayHi, 1000); // Привет, undefined!
```

При запуске этого кода мы видим, что вызов `this.firstName` возвращает не «Вася», а `undefined`!

Это произошло потому, что `setTimeout` получил функцию `sayHi` отдельно от объекта `user` (именно здесь функция и потеряла контекст). То есть последняя строка может быть переписана как:

```
1 let f = user.sayHi;
2 setTimeout(f, 1000); // контекст user потеряли
```

Метод `setTimeout` в браузере имеет особенность: он устанавливает `this=window` для вызова функции (в Node.js `this` становится объектом таймера, но здесь это не имеет значения). Таким образом, для `this.firstName` он пытается получить `window.firstName`, которого не существует. В других подобных случаях `this` обычно просто становится `undefined`.

Задача довольно типичная – мы хотим передать метод объекта куда-то ещё (в этом конкретном случае – в планировщик), где он будет вызван. Как бы сделать так, чтобы он вызывался в правильном контексте?

Решение 1: сделать функцию-обёртку

Самый простой вариант решения – это обернуть вызов в анонимную функцию, создав замыкание:

```
1 let user = {
2   firstName: "Вася",
3   sayHi() {
4     alert(`Привет, ${this.firstName}!`);
5   }
6 };
7
8 setTimeout(function() {
9   user.sayHi(); // Привет, Вася!
10 }, 1000);
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение

Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Теперь код работает корректно, так как объект `user` достаётся из замыкания, а затем вызывается его метод `sayHi`.

То же самое, только короче:

```
1 setTimeout(() => user.sayHi(), 1000); // Привет, Вася!
```

Выглядит хорошо, но теперь в нашем коде появилась небольшая уязвимость.

Что произойдёт, если до момента срабатывания `setTimeout` (ведь задержка составляет целую секунду!) в переменную `user` будет записано другое значение? Тогда вызов неожиданно будет совсем не тот!

```
1 let user = {
2   firstName: "Вася",
3   sayHi() {
4     alert(`Привет, ${this.firstName}!`);
5   }
6 };
7
8 setTimeout(() => user.sayHi(), 1000);
9
10 // ...в течение 1 секунды
11 user = { sayHi() { alert("Другой пользователь в 'setTim
12
13 // Другой пользователь в 'setTimeout'!
```

Следующее решение гарантирует, что такого не случится.

Решение 2: привязать контекст с помощью bind

В современном JavaScript у функций есть встроенный метод `bind`, который позволяет зафиксировать `this`.

Базовый синтаксис `bind`:

```
1 // полный синтаксис будет представлен немного позже
2 let boundFunc = func.bind(context);
```

Результатом вызова `func.bind(context)` является особый «экзотический объект» (термин взят из спецификации), который вызывается как функция и прозрачно передаёт вызов в `func`, при этом устанавливая `this=context`.

Другими словами, вызов `boundFunc` подобен вызову `func` с фиксированным `this`.

Например, здесь `funcUser` передаёт вызов в `func`, фиксируя `this=user`:

```
1 let user = {
2   firstName: "Вася"
3 };
4
5 function func() {
6   alert(this.firstName);
7 }
8
9 let funcUser = func.bind(user);
10 funcUser(); // Вася
```

Здесь `func.bind(user)` – это «связанный вариант» `func`, с фиксированным `this=user`.

Все аргументы передаются исходному методу `func` как есть, например:



Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение

Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
1 let user = {
2   firstName: "Вася"
3 };
4
5 function func(phrase) {
6   alert(phrase + ', ' + this.firstName);
7 }
8
9 // привязка this к user
10 let funcUser = func.bind(user);
11
12 funcUser("Привет"); // Привет, Вася (аргумент "Привет")
```

Теперь давайте попробуем с методом объекта:

```
1 let user = {
2   firstName: "Вася",
3   sayHi() {
4     alert(`Привет, ${this.firstName}!`);
5   }
6 };
7
8 let sayHi = user.sayHi.bind(user); // (*)
9
10 sayHi(); // Привет, Вася!
11
12 setTimeout(sayHi, 1000); // Привет, Вася!
```

В строке (*) мы берём метод `user.sayHi` и привязываем его к `user`. Теперь `sayHi` – это «связанная» функция, которая может быть вызвана отдельно или передана в `setTimeout` (контекст всегда будет правильным).

Здесь мы можем увидеть, что `bind` исправляет только `this`, а аргументы передаются как есть:

```
1 let user = {
2   firstName: "Вася",
3   say(phrase) {
4     alert(`${phrase}, ${this.firstName}!`);
5   }
6 };
7
8 let say = user.say.bind(user);
9
10 say("Привет"); // Привет, Вася (аргумент "Привет" перед
11 say("Пока"); // Пока, Вася (аргумент "Пока" передан в ф
```

i Удобный метод: `bindAll`

Если у объекта много методов и мы планируем их активно передавать, то можно привязать контекст для них всех в цикле:

```
1 for (let key in user) {
2   if (typeof user[key] == 'function') {
3     user[key] = user[key].bind(user);
4   }
5 }
```

Некоторые JS-библиотеки предоставляют встроенные функции для удобной массовой привязки контекста, например `_.bindAll(obj)` в `lodash`.

Частичное применение

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение

Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



До сих пор мы говорили только о привязывании `this`. Давайте шагнём дальше.

Мы можем привязать не только `this`, но и аргументы. Это делается редко, но иногда может быть полезно.

Полный синтаксис `bind`:

```
1 let bound = func.bind(context, [arg1], [arg2], ...);
```

Это позволяет привязать контекст `this` и начальные аргументы функции.

Например, у нас есть функция умножения `mul(a, b)`:

```
1 function mul(a, b) {
2   return a * b;
3 }
```

Давайте воспользуемся `bind`, чтобы создать функцию `double` на её основе:

```
1 function mul(a, b) {
2   return a * b;
3 }
4
5 let double = mul.bind(null, 2);
6
7 alert( double(3) ); // = mul(2, 3) = 6
8 alert( double(4) ); // = mul(2, 4) = 8
9 alert( double(5) ); // = mul(2, 5) = 10
```



Вызов `mul.bind(null, 2)` создаёт новую функцию `double`, которая передаёт вызов `mul`, фиксируя `null` как контекст, и `2` – как первый аргумент. Следующие аргументы передаются как есть.



Это называется **частичное применение** – мы создаём новую функцию, фиксируя некоторые из существующих параметров.

Обратите внимание, что в данном случае мы на самом деле не используем `this`. Но для `bind` это обязательный параметр, так что мы должны передать туда что-нибудь вроде `null`.

В следующем коде функция `triple` умножает значение на три:

```
1 function mul(a, b) {
2   return a * b;
3 }
4
5 let triple = mul.bind(null, 3);
6
7 alert( triple(3) ); // = mul(3, 3) = 9
8 alert( triple(4) ); // = mul(3, 4) = 12
9 alert( triple(5) ); // = mul(3, 5) = 15
```



Для чего мы обычно создаём частично применённую функцию?

Польза от этого в том, что возможно создать независимую функцию с понятным названием (`double`, `triple`). Мы можем использовать её и не передавать каждый раз первый аргумент, т.к. он зафиксирован с помощью `bind`.

В других случаях частичное применение полезно, когда у нас есть очень общая функция и для удобства мы хотим создать её более специализированный вариант.

Например, у нас есть функция `send(from, to, text)`. Потом внутри объекта `user` мы можем захотеть использовать её частный вариант: `sendTo(to, text)`, который отправляет текст от имени текущего пользователя.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение

Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Частичное применение без контекста

Что если мы хотим зафиксировать некоторые аргументы, но не контекст `this`? Например, для метода объекта.

Встроенный `bind` не позволяет этого. Мы не можем просто опустить контекст и перейти к аргументам.

К счастью, легко создать вспомогательную функцию `partial`, которая привязывает только аргументы.

Вот так:

```
1 function partial(func, ...argsBound) {
2   return function(...args) { // (*)
3     return func.call(this, ...argsBound, ...args);
4   }
5 }
6
7 // использование:
8 let user = {
9   firstName: "John",
10  say(time, phrase) {
11    alert(`[${time}] ${this.firstName}: ${phrase}!`);
12  }
13 };
14
15 // добавляем частично применённый метод с фиксированным
16 user.sayNow = partial(user.say, new Date().getHours() +
17
18 user.sayNow("Hello");
19 // Что-то вроде этого:
20 // [10:00] John: Hello!
```

Результатом вызова `partial(func[, arg1, arg2...])` будет обёртка `(*)`, которая вызывает `func` с:

- Тем же `this`, который она получает (для вызова `user.sayNow` – это будет `user`)
- Затем передаёт ей `...argsBound` – аргументы из вызова `partial` ("10:00")
- Затем передаёт ей `...args` – аргументы, полученные обёрткой ("Hello")

Благодаря оператору расширения `...` реализовать это очень легко, не правда ли?

Также есть готовый вариант `_partial` из библиотеки `lodash`.

Итого

Метод `bind` возвращает «привязанный вариант» функции `func`, фиксируя контекст `this` и первые аргументы `arg1, arg2 ...`, если они заданы.

Обычно `bind` применяется для фиксации `this` в методе объекта, чтобы передать его в качестве колбэка. Например, для `setTimeout`.

Когда мы привязываем аргументы, такая функция называется «частично применённой» или «частичной».

Частичное применение удобно, когда мы не хотим повторять один и тот же аргумент много раз. Например, если у нас есть функция `send(from, to)` и `from` всё время будет одинаков для нашей задачи, то мы можем создать частично применённую функцию и дальше работать с ней.

✓ Задачи

Связанная функция как метод

важность: 5

Что выведет функция?

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение

Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
1 function f() {
2   alert( this ); // ?
3 }
4
5 let user = {
6   g: f.bind(null)
7 };
8
9 user.g();
```

решение

Повторный bind

важность: 5

Можем ли мы изменить this дополнительным связыванием?

Что выведет этот код?

```
1 function f() {
2   alert(this.name);
3 }
4
5 f = f.bind( {name: "Вася"} ).bind( {name: "Петя"} );
6
7 f();
```

решение

Свойство функции после bind

важность: 5

В свойство функции записано значение. Изменится ли оно после применения bind? Обоснуйте ответ.

```
1 function sayHi() {
2   alert( this.name );
3 }
4 sayHi.test = 5;
5
6 let bound = sayHi.bind({
7   name: "Вася"
8 });
9
10 alert( bound.test ); // что выведет? почему?
```

решение

Исправьте функцию, теряющую "this"

важность: 5

Вызов askPassword() в приведённом ниже коде должен проверить пароль и затем вызвать user.loginOk/loginFail в зависимости от ответа.

Однако, его вызов приводит к ошибке. Почему?

Исправьте выделенную строку, чтобы всё работало (других строк изменять не надо).

```
1 function askPassword(ok, fail) {
2   let password = prompt("Password?", '');
3   if (password == "rockstar") ok();
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение

Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться



Редактировать на GitHub



```
4   else fail();
5 }
6
7 let user = {
8   name: 'Вася',
9
10  loginOk() {
11    alert(`${this.name} logged in`);
12  },
13
14  loginFail() {
15    alert(`${this.name} failed to log in`);
16  },
17
18 };
19
20 askPassword(user.loginOk, user.loginFail);
```

решение

Использование частично применённой функции для логина

важность: 5

Это задание является немного усложнённым вариантом одного из предыдущих – [Исправьте функцию, теряющую "this"](#).

Объект `user` был изменён. Теперь вместо двух функций `loginOk/loginFail` у него есть только одна – `user.login(true/false)`.

Что нужно передать в вызов функции `askPassword` в коде ниже, чтобы она могла вызывать функцию `user.login(true)` как `ok` и функцию `user.login(false)` как `fail`?

```
1 function askPassword(ok, fail) {
2   let password = prompt("Password?", '');
3   if (password == "rockstar") ok();
4   else fail();
5 }
6
7 let user = {
8   name: 'John',
9
10  login(result) {
11    alert( this.name + (result ? ' logged in' : ' faile
12  }
13 };
14
15 askPassword(?, ?); // ?
```

Ваши изменения должны затрагивать только выделенный фрагмент кода.

решение

Проводим [курсы по JavaScript и фреймворкам](#).



Комментарии

перед тем как писать...

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Потеря «this»

Решение 1: сделать функцию-обёртку

Решение 2: привязать контекст с помощью bind

Частичное применение

Частичное применение без контекста

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)

