

Раздел

[Разное](#)

Навигация по уроку

Каррирование? Зачем?

Продвинутая реализация
каррирования

Итого

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠](#) → [Язык программирования JavaScript](#) → [Разное](#) 📅 19-го декабря 2019

Каррирование

Каррирование – продвинутая техника для работы с функциями. Она используется не только в JavaScript, но и в других языках.

Каррирование – это трансформация функций таким образом, чтобы они принимали аргументы не как `f(a, b, c)`, а как `f(a)(b)(c)`.

Каррирование не вызывает функцию. Оно просто трансформирует её.

Давайте сначала посмотрим на пример, чтобы лучше понять, о чём речь, а потом на практическое применение каррирования.

Создадим вспомогательную функцию `curry(f)`, которая выполняет каррирование функции `f` с двумя аргументами. Другими словами, `curry(f)` для функции `f(a, b)` трансформирует её в `f(a)(b)`.

```
1 function curry(f) { // curry(f) выполняет каррирование
2   return function(a) {
3     return function(b) {
4       return f(a, b);
5     };
6   };
7 }
8
9 // использование
10 function sum(a, b) {
11   return a + b;
12 }
13
14 let carriedSum = curry(sum);
15
16 alert( carriedSum(1)(2) ); // 3
```

Как вы видите, реализация довольно проста: это две обёртки.

- Результат `curry(func)` – обёртка `function(a)`.
- Когда она вызывается как `sum(1)`, аргумент сохраняется в лексическом окружении и возвращается новая обёртка `function(b)`.
- Далее уже эта обёртка вызывается с аргументом `2` и передаёт вызов к оригинальной функции `sum`.

Более продвинутые реализации каррирования, как например `_.curry` из библиотеки `lodash`, возвращают обёртку, которая позволяет запустить функцию как обычным образом, так и частично.

```
1 function sum(a, b) {
2   return a + b;
3 }
4
5 let carriedSum = _.curry(sum); // используем _.curry из
6
7 alert( carriedSum(1, 2) ); // 3, можно вызывать как обы
8 alert( carriedSum(1)(2) ); // 3, а можно частично
```

Каррирование? Зачем?

Чтобы понять пользу от каррирования, нам определённо нужен пример из реальной жизни.

Например, у нас есть функция логирования `log(date, importance, message)`, которая форматирует и выводит информацию. В реальных

Раздел

[Разное](#)

Навигация по уроку

Каррирование? Зачем?

Продвинутая реализация
каррирования

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



проектах у таких функций есть много полезных возможностей, например, посылать логи по сети, здесь для простоты используем `alert`:

```
1 function log(date, importance, message) {
2   alert(`[${date.getHours()}:${date.getMinutes()}] [${i
3 }
```

А теперь давайте применим к ней каррирование!

```
1 log = _.curry(log);
```

После этого `log` продолжает работать нормально:

```
1 log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

...Но также работает вариант с каррированием:

```
1 log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Давайте сделаем удобную функцию для логов с текущим временем:

```
1 // logNow будет частичным применением функции log с фик
2 let logNow = log(new Date());
3
4 // используем её
5 logNow("INFO", "message"); // [HH:mm] INFO message
```



Теперь `logNow` – это `log` с фиксированным первым аргументом, иначе говоря, «частично применённая» или «частичная» функция.



Мы можем пойти дальше и сделать удобную функцию для именно отладочных логов с текущим временем:

```
1 let debugNow = logNow("DEBUG");
2
3 debugNow("message"); // [HH:mm] DEBUG message
```

Итак:

1. Мы ничего не потеряли после каррирования: `log` всё так же можно вызывать нормально.
2. Мы можем легко создавать частично применённые функции, как сделали для логов с текущим временем.

Продвинутая реализация каррирования

В случае, если вам интересны детали, вот «продвинутая» реализация каррирования для функций с множеством аргументов, которую мы могли бы использовать выше.

Она очень короткая:

```
1 function curry(func) {
2
3   return function curried(...args) {
4     if (args.length >= func.length) {
5       return func.apply(this, args);
6     } else {
7       return function(...args2) {
8         return curried.apply(this, args.concat(args2));
9       }
10    }
11  }
```

Раздел

Разное

Навигация по уроку

Каррирование? Зачем?

Продвинутая реализация каррирования

Итого

Комментарии

Поделиться



Редактировать на GitHub



```
11 };  
12  
13 }
```

Примеры использования:

```
1 function sum(a, b, c) {  
2   return a + b + c;  
3 }  
4  
5 let curriedSum = curry(sum);  
6  
7 alert( curriedSum(1, 2, 3) ); // 6, всё ещё можно вызывать  
8 alert( curriedSum(1)(2,3) ); // 6, каррирование первого  
9 alert( curriedSum(1)(2)(3) ); // 6, каррирование всех а
```

Новое `curry` выглядит сложновато, но на самом деле его легко понять.

Результат вызова `curry(func)` – это обёртка `curried`, которая выглядит так:

```
1 // func -- функция, которую мы трансформируем  
2 function curried(...args) {  
3   if (args.length >= func.length) { // (1)  
4     return func.apply(this, args);  
5   } else {  
6     return function pass(...args2) { // (2)  
7       return curried.apply(this, args.concat(args2));  
8     }  
9   }  
10 };
```



Когда мы запускаем её, есть две ветви выполнения `if`:

1. Вызвать сейчас: если количество переданных аргументов `args` совпадает с количеством аргументов при объявлении функции (`func.length`) или больше, тогда вызов просто переходит к ней.
2. Частичное применение: в противном случае `func` не вызывается сразу. Вместо этого, возвращается другая обёртка `pass`, которая снова применит `curried`, передав предыдущие аргументы вместе с новыми. Затем при новом вызове мы опять получим либо новое частичное применение (если аргументов недостаточно) либо, наконец, результат.

Например, давайте посмотрим, что произойдёт в случае `sum(a, b, c)`. У неё три аргумента, так что `sum.length = 3`.

Для вызова `curried(1)(2)(3)`:

1. Первый вызов `curried(1)` запоминает 1 в своём лексическом окружении и возвращает обёртку `pass`.
2. Обёртка `pass` вызывается с (2): она берёт предыдущие аргументы (1), объединяет их с тем, что получила сама (2) и вызывает `curried(1, 2)` со всеми ними. Так как число аргументов всё ещё меньше 3-х, `curry` возвращает `pass`.
3. Обёртка `pass` вызывается снова с (3). Для следующего вызова `pass(3)` берёт предыдущие аргументы (1, 2) и добавляет к ним 3, делая вызов `curried(1, 2, 3)` – наконец 3 аргумента, и они передаются оригинальной функции.

Если всё ещё не понятно, просто распишите последовательность вызовов на бумаге.



Раздел

[Разное](#)

Навигация по уроку

Каррирование? Зачем?

Продвинутая реализация
каррирования

Итого

Комментарии

Поделиться



Редактировать на GitHub



Только функции с фиксированным количеством аргументов

Для каррирования необходима функция с фиксированным количеством аргументов.

Функцию, которая использует остаточные параметры, типа `f(...args)`, так каррировать не получится.

Немного больше, чем каррирование

По определению, каррирование должно превращать `sum(a, b, c)` в `sum(a)(b)(c)`.

Но, как было описано, большинство реализаций каррирования в JavaScript более продвинуты: они также оставляют вариант вызова функции с несколькими аргументами.

Итого

Каррирование – это трансформация, которая превращает вызов `f(a, b, c)` в `f(a)(b)(c)`. В JavaScript реализация обычно позволяет вызывать функцию обоими вариантами: либо нормально, либо возвращает частично применённую функцию, если количество аргументов недостаточно.

Каррирование позволяет легко получать частичные функции. Как мы видели в примерах с логгами: универсальная функция `log(date, importance, message)` после каррирования возвращает нам частично применённую функцию, когда вызывается с одним аргументом, как `log(date)` или двумя аргументами, как `log(date, importance)`.



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



Комментарии

перед тем как писать...