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)`.

```
function curry(f) { // curry(f) выполняет каррирование
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// использование
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3
```

Как вы видите, реализация довольна 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.

```
function sum(a, b) {
  return a + b;
}

let curriedSum = _.curry(sum); // используем _.curry из lodash

alert( curriedSum(1, 2) ); // 3, можно вызывать как обычно
alert( curriedSum(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 проектах у таких функций есть много полезных возможностей, например, посылать логи по сети, здесь для простоты используем `alert`:

```
function log(date, importance, message) {
  alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);
}
```

А теперь давайте применим к ней каррирование!

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

После этого `log` продолжает работать нормально:

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

...Но также работает вариант с каррированием:

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

Давайте сделаем удобную функцию для логов с текущим временем:

```
// logNow будет частичным применением функции log с фиксированным первым аргументом
let logNow = log(new Date());

// используем её
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:

```
let debugNow = logNow("DEBUG");

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:

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  };

}
```

Примеры использования:

```
function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, всё ещё можно вызывать нормально
alert( curriedSum(1)(2,3) ); // 6, каррирование первого аргумента
alert( curriedSum(1)(2)(3) ); // 6, каррирование всех аргументов
```

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

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

```
// func -- функция, которую мы трансформируем
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function pass(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};
```

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.

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 логами: 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)`.

Загрузить комментарии