5 ноября 2022 г.

Объект функции, NFE

Как мы уже знаем, в JavaScript функция – это значение.

Каждое значение в JavaScript имеет свой тип. А функция – это какой тип?

В JavaScript функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но и использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

Свойство «name»

Объект функции содержит несколько полезных свойств.

Например, имя функции нам доступно как свойство «name»:

```
1  function sayHi() {
2    alert("Hi");
3  }
4
5  alert(sayHi.name); // sayHi
```

Что довольно забавно, логика назначения `name` весьма умная. Она присваивает корректное имя даже в случае, когда функция создаётся без имени и тут же присваивается, вот так:

```
1  let sayHi = function() {
2    alert("Hi");
3  };
4
5  alert(sayHi.name); // sayHi (есть имя!)
```

Это работает даже в случае присваивания значения по умолчанию:

```
1  function f(sayHi = function() {}) {
2    alert(sayHi.name); // sayHi (работает!)
3  }
4
5  f();
```

В спецификации это называется «контекстное имя»: если функция не имеет name, то JavaScript пытается определить его из контекста.

Также имена имеют и методы объекта:

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }

}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

В этом нет никакой магии. Бывает, что корректное имя определить невозможно. В таких случаях свойство name имеет пустое значение. Например:

```
// функция объявлена внутри массива
let arr = [function() {}];

alert( arr[0].name ); // <пустая строка>
// здесь отсутствует возможность определить имя, поэтому его нет
```

Впрочем, на практике такое бывает редко, обычно функции имеют `name`.

Свойство «length»

Ещё одно встроенное свойство «length» содержит количество параметров функции в её объявлении. Например:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

Как мы видим, троеточие, обозначающее «остаточные параметры», здесь как бы «не считается»

Свойство `length` иногда используется для интроспекций в функциях, которые работают с другими функциями.

Например, в коде ниже функция `ask` принимает в качестве параметров вопрос `question` и произвольное количество функций-обработчиков ответа `handler`.

Когда пользователь отвечает на вопрос, функция вызывает обработчики. Мы можем передать два типа обработчиков:

- Функцию без аргументов, которая будет вызываться только в случае положительного ответа.
- Функцию с аргументами, которая будет вызываться в обоих случаях и возвращать ответ.

Чтобы вызвать обработчик `handler` правильно, будем проверять свойство `handler.length`.

Идея состоит в том, чтобы иметь простой синтаксис обработчика без аргументов для положительных ответов (наиболее распространённый случай), но также и возможность передавать универсальные обработчики:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }

}

// для положительных ответов вызываются оба типа обработчиков
// для отрицательных - только второго типа
ask("Вопрос?", () => alert('Вы ответили да'), result => alert(result));
```

Это частный случай так называемого Ad-hoc-полиморфизма – обработка аргументов в зависимости от их типа или, как в нашем случае – от значения `length`. Эта идея имеет применение в библиотеках JavaScript.

Пользовательские свойства

Мы также можем добавить свои собственные свойства.

Давайте добавим свойство `counter` для отслеживания общего количества вызовов:

```
function sayHi() {
  alert("Hi");

  // давайте посчитаем, сколько вызовов мы сделали
  sayHi.counter++;
}
sayHi.counter = 0; // начальное значение

sayHi(); // Hi
sayHi(); // Hi

alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2 раза
```

Свойство не есть переменная

Свойство функции, назначенное как `sayHi.counter = 0`, не *объявляет* локальную переменную `counter` внутри неё. Другими словами, свойство `counter` и переменная `let counter` – это две независимые вещи.

Мы можем использовать функцию как объект, хранить в ней свойства, но они никак не влияют на её выполнение. Переменные – это не свойства функции и наоборот. Это два параллельных мира.

Иногда свойства функции могут использоваться вместо замыканий. Например, мы можем переписать функцию-счётчик из главы Область видимости переменных, замыкание, используя её свойство:

```
function makeCounter() {
  // вместо
  // let count = 0

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

Свойство `count` теперь хранится прямо в функции, а не в её внешнем лексическом окружении.

Это хуже или лучше, чем использовать замыкание?

Основное отличие в том, что если значение `count` живёт во внешней переменной, то оно не доступно для внешнего кода. Изменить его могут только вложенные функции. А если оно присвоено как свойство функции, то мы можем его получить:

```
function makeCounter() {

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

Поэтому выбор реализации зависит от наших целей.

Named Function Expression

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя.

Например, давайте объявим Function Expression:

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
```

И присвоим ему имя:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

Чего мы здесь достигли? Какова цель этого дополнительного имени `func`?

Для начала заметим, что функция всё ещё задана как Function Expression. Добавление `"func"` после `function` не превращает объявление в Function Declaration, потому что оно все ещё является частью выражения присваивания.

Добавление такого имени ничего не ломает.

Функция все ещё доступна как `sayHi()`:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};

sayHi("John"); // Hello, John
```

Есть две важные особенности имени `func`, ради которого оно даётся:

1. Оно позволяет функции ссылаться на себя же.
2. Оно не доступно за пределами функции.

Например, ниже функция `sayHi` вызывает себя с `"Guest"`, если не передан параметр `who`:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // использует func, чтобы снова вызвать себя же
  }
};

sayHi(); // Hello, Guest

// А вот так - не сработает:
func(); // Ошибка, func не определена (недоступна вне функции)
```

Почему мы используем `func`? Почему просто не использовать `sayHi` для вложенного вызова?

Вообще, обычно мы можем так поступить:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};
```

Однако, у этого кода есть проблема, которая заключается в том, что значение `sayHi` может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнёт выдавать ошибки:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Ошибка: sayHi не является функцией
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Ошибка, вложенный вызов sayHi больше не работает!
```

Так происходит, потому что функция берёт `sayHi` из внешнего лексического окружения. Так как локальная переменная `sayHi` отсутствует, используется внешняя. И на момент вызова эта внешняя `sayHi` равна `null`.

Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы.

Давайте используем его, чтобы исправить наш код:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // Теперь всё в порядке
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (вложенный вызов работает)
```

Теперь всё работает, потому что имя `"func"` локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию.

Внешний код все ещё содержит переменные `sayHi` и `welcome`, но теперь `func` – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Это не работает с Function Declaration

Трюк с «внутренним» именем, описанный выше, работает только для Function Expression и *не* работает для Function Declaration. Для Function Declaration синтаксис не предусматривает возможность объявить дополнительное «внутреннее» имя.

Зачастую, когда нам нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

Итого

Функции – это объекты.

Их свойства:

- `name` – имя функции. Обычно берётся из объявления функции, но если там нет – JavaScript пытается понять его из контекста.
- `length` – количество аргументов в объявлении функции. Троеточие («остаточные параметры») не считается.

Если функция объявлена как Function Expression (вне основного потока кода) и имеет имя, тогда это называется Named Function Expression (Именованным Функциональным Выражением). Это имя может быть использовано для ссылки на себя же, для рекурсивных вызовов и т.п.

Также функции могут содержать дополнительные свойства. Многие известные JavaScript-библиотеки искусно используют эту возможность.

Они создают «основную» функцию и добавляют множество «вспомогательных» функций внутрь первой. Например, библиотека jQuery создаёт функцию с именем `$`. Библиотека lodash создаёт функцию `_`, а потом добавляет в неё `_.clone`, `_.keyBy` и другие свойства (чтобы узнать о ней побольше см. документацию). Они делают это, чтобы уменьшить засорение глобального пространства имён посредством того, что одна библиотека предоставляет только одну глобальную переменную, уменьшая вероятность конфликта имён.

Таким образом, функция может не только делать что-то сама по себе, но также и предоставлять полезную функциональность через свои свойства.

Задачи

Установка и уменьшение значения счётчика

важность: 5

Измените код `makeCounter()` так, чтобы счётчик мог уменьшать и устанавливать значение:

- `counter()` должен возвращать следующее значение (как и раньше).
- `counter.set(value)` должен устанавливать счётчику значение `value`.
- `counter.decrease()` должен уменьшать значение счётчика на 1.

Посмотрите код из песочницы с полным примером использования.

P.S. Для того, чтобы сохранить текущее значение счётчика, можно воспользоваться как замыканием, так и свойством функции. Или сделать два варианта решения: и так, и так.

Открыть песочницу с тестами для задачи.

решение

В решении использована локальная переменная `count`, а методы сложения записаны прямо в `counter`. Они разделяют одно и то же лексическое окружение и также имеют доступ к текущей переменной `count`.

```
function makeCounter() {
  let count = 0;

  function counter() {
    return count++;
  }

  counter.set = value => count = value;

  counter.decrease = () => count--;

  return counter;
}
```

Открыть решение с тестами в песочнице.

Сумма с произвольным количеством скобок

важность: 2

Напишите функцию `sum`, которая бы работала следующим образом:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

P.S. Подсказка: возможно вам стоит сделать особый метод преобразования в примитив для функции.

решение

1. В общем, чтобы это *хоть как-нибудь* заработало, результат, возвращаемый `sum`, должен быть функцией.
2. Между вызовами эта функция должна удерживать в памяти текущее значение счётчика.
3. Согласно заданию, функция должна преобразовываться в число, когда она используется с оператором `==`. Функции – объекты, так что преобразование происходит, как описано в главе Преобразование объектов в примитивы, поэтому можно создать наш собственный метод, возвращающий число.

Код:

```
function sum(a) {

  let currentSum = a;

  function f(b) {
    currentSum += b;
    return f;
  }

  f.toString = function() {
    return currentSum;
  };

  return f;
}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1)(2) ); // 6
alert( sum(6)(-1)(-2)(-3) ); // 0
alert( sum(0)(1)(2)(3)(4)(5) ); // 15
```

Пожалуйста, обратите внимание на то, что функция `sum` выполняется лишь однажды и просто возвращает функцию `f`.

Далее, при каждом последующем вызове, `f` суммирует свой аргумент со значением `currentSum` и возвращает себя же.

В последней строке `f` нет никакой рекурсии.

Вот как выглядит рекурсия:

```
function f(b) {
  currentSum += b;
  return f(); // <-- рекурсивный вызов
}
```

В нашем случае мы просто возвращаем функцию, не вызывая её:

```
function f(b) {
  currentSum += b;
  return f; // <-- не вызывает себя. Просто возвращает
}
```

Функция `f` будет использоваться в последующем вызове и снова возвращать себя столько раз, сколько будет необходимо. Затем, при использовании в качестве числа или строки, метод `toString` возвращает `currentSum` – число. Также здесь мы можем использовать `Symbol.toPrimitive` или `valueOf` для преобразования.

Предыдущий урок Следующий урок

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье — пишите, что именно и с какого места.

Юрий

14 дней назад

⚡ **Хочешь понять зачем нужен объект функции?**
Тогда гоу в мой блог ТГ-блог «Джун на фронте»!

👨‍💻 Автор - системный администратор, который с декабря 2021 года освоил HTML, CSS, JS, Vue, Nuxt, React Native, MongoDB и Node.js.

Изучи мой путь в мире разработки: от новичка до создателя 🤖 **Телеграм-бота для автоматической отправки** откликов!

💡 Тут ты найдешь полезные ответы на насущные вопросы, анализ рынка вакансий и советы от человека, прошедшего этот путь.

Вбивайте «**Джун на фронте**» и присоединяйтесь.

0 0 Ответить

Роман

19 дней назад

И в чем я не прав?

Код:

```
"use sctrict"

function sum(val) {

let currentSum = val;

function sum(val) {

currentSum += val;
return sum;
}

sum[Symbol.toPrimitive] = function(hint) {
```

0 0 Ответить

Nachang Роман

18 дней назад edited

Здравствуйте, Роман.
Ваше решение абсолютно верное, не считая опечатки в "use sCtrict".

Суть в том, что метод `[[toPrimitive]]` сработает, когда нужно будет привести объект к примитивному значению. Вывод в консоль не требует этого, поэтому объект выводится «как есть».

Функция `alert(message)` ожидает в качестве аргумента примитивное значение. Если передать объект, он будет неявно приведён к примитивному значению. То же самое относится и к оператору «==», который приводит свои операнды к примитивным значениям перед сравнением.

```
// Неявно приведём к строке:
console.log(sum(1)(2) + ''); // Вывод: '3'
// Явно приведём к строке:
console.log(String(sum(1)(2))); // Вывод: '3'

// Неявно приведём к числу:
console.log(+sum(1)(2)(3)); // Вывод: 6
// Явно приведём к числу:
console.log(Number(sum(1)(2)(3))); // Вывод: 6

// Явно вызовем метод [[toPrimitive]]
console.log(sum(6)(-1)(-2)(-3)[Symbol.toPrimitive]()); // Вывод: 0
```

показать больше

1 0 Ответить

Р **Роман** Nachang

18 дней назад

Благодарю!

0 0 Ответить

К **Константин Гришин**

месяц назад

Моё субъективное мнение, вот такой код проще проиллюстрирует информацию приведенную в статье:

```
function sum(first_n) {
let sum = first_n;
let f = function(next_n) {
sum += next_n;
f.result = () => {
return sum
}
return f;
}
return f;
}
```

Вызов: `sum(5)(6)(8)(15).result()` вернёт 34

А то во втором задании с == например: `sum(1)(2)(3) == 6; // 1 + 2 + 3` не очевидно, что результатом

сравнения должно быть true. И я сначала подумал, что сравнение вернёт сумму чисел.

0 0 Ответить

Aleksey Pavlov
2 месяца назад

```
function sum(initial) {
let counter = initial

function f(n) {
counter += n

return f
}

f[Symbol.toPrimitive] = function () {
return counter
}

return f
}
```

0 0 Ответить

Влад
3 месяца назад

Вторая задача сложная - я сам дошел до идеи с рекурсией но не додумался зделать для рекурсивной функции свойство toString , довольно красивое решение
Теперь буду держать в голове что можно влиять на поведение обьектов и функций в зависимости от того к какому типу их нужно привести

0 0 Ответить

Aleksey Pavlov → Влад
2 месяца назад

лучше для это использовать Symbol.toPrimitive

0 0 Ответить

Odelschwank
3 месяца назад

```
1-st: function makeCounter() {

// ... ваш код ...
function counter() {
return counter.count++;
}
counter.set = function(newCount) {
return counter.count = newCount;
}

counter.decrease = function() {
counter.count--;
}
```

```
}
counter.count = 0;

return counter;
}
```

0 0 Ответить

A **Albert M**
6 месяцев назад

Вторая задача, получилось неуклюже конечно, но зато сам

```
function sum(a) {
sum.sum += a;
return sum;
}
sum.sum = 0;
sum.valueOf = function() {
let sum = this.sum;
this.sum = 0;
return sum;
}
```

0 0 Ответить

V **Vladimir** → Albert M
4 месяца назад

ты когда решения придумываешь, ты хоть проверяешь, как они работают?

0 0 Ответить

P **Рамазан Тавбулаев**
9 месяцев назад

Люди, я понимаю второе задание сложное, но оно, судя по важности, чисто для энтузиастов.
А вот первое задание как вы можете НЕ решить ? Оно абсолютно понятное, если вы действительно читали, а не пробегали глазами прошлые темы.

2 2 Ответить

 **Golden Boy**
10 месяцев назад

С недавних пор пришел к осознанию, что иногда для меня есть смысл пропускать решение задач, так как на это уходит очень много времени... 😑

2 0 Ответить

 **Dedal** → Golden Boy
8 месяцев назад

За себя могу сказать, что поначалу на некоторые задачи уходило часа по 2, а то и 3 времени. Брал паузу и снова возвращался к решению, иногда приходилось заново перечитывать главу и вновь разбираться в работе тех или иных методов и поведении структур данных. Считаю оно того стоит, сейчас уже заметил, что в целом начал решать задачи гораздо быстрее, чувствуется прогресс при таком подходе. Мне кажется задачи лучше не пропускать ведь именно в их решении и получается первичный опыт программирования, который даст задел для решения тестовых заданий при собесе и

дальнейшей реализации программных проектов.

3 0 Ответить

A **Anatolii Tarasov (Tarasov.Fron** → Golden Boy
9 месяцев назад

Пока есть идеи как решить - нужно пытаться решить. Если идей нет - нужно подумать пять минут и если решения не пришло, можно посмотреть ответ. И двигаться дальше.

Я вернулся перечитать весь учебник через три месяца после его первичного прочтения/прорешивания и многие вещи встают на свои места, причем сходу.

В том числе потому что я уже понимаю как и куда их реально можно применять. Изучил реакт, редакс, немного ноду и понимаю, как это дальше работает и зачем.

В общем, начитанность решает. И ресурсов мозга на осознание концепций уходит гораздо меньше. Но без первичного вникания и взрыва мозга никуда не продвинуться, так что не отступай. Я, бывало, перечитывал одну главу раз по двадцать, прежде чем хотя бы что-то вставало в мозгу на свое место.

7 0 Ответить

V **Vladimir** → Anatolii Tarasov (Tarasov.Fron
4 месяца назад

как успехи? нашел работу?

0 0 Ответить

O **Omak** → Anatolii Tarasov (Tarasov.Fron
9 месяцев назад

Отличный совет!

1 0 Ответить

S **Svytoslav Matsera**
10 месяцев назад

```
sum = function func(x){
if (func.res){
func.res += x;
}else{
func.res = x;
}
func[Symbol.toPrimitive]=function(hint){
return func.res
}
return func
}
```

0 0 Ответить

E **exponenta**
год назад edited

Решение 1-ой задачи

```
function makeCounter() {
function counter() {
return ++counter.count;
}
counter.count = 0;
counter.set = function(value) {
return this.count = value;
}
counter.decrease = function() {
return this.count && --this.count;
}
return counter;
}
```

1 0 Ответить

V **Virtual NETSCPE**
год назад edited

мой вариант решения последней задачи, я думаю у меня даже лучше вышло, пожалуйста оцените

0 0 Ответить

Загрузить ещё комментарии