Глава 3. Использование функций

Функции — это рабочие лошадки JavaScript, являющиеся одновременно и основным доступным программисту средством абстракции, и механизмом реализации. Функции сами по себе играют те роли, которые в других языках исполняются самыми разными средствами: процедурами, методами, конструкторами и даже классами и модулями. После освоения всех тонкостей использования функций вы овладеете существенной частью JavaScript. Обратной стороной медали можно считать то, что на изучение всех тонкостей эффективного использования функций в различных обстоятельствах нужно много времени.

18 Разберитесь в различиях между вызовами функций, методов и конструкторов

Если вы знакомы с объектно-ориентированным программированием, то, наверное, привыкли думать о функциях, методах и конструкторах классов как о трех разных вещах. В JavaScript это лишь три модели одной и той же конструкции — функции.

Простейшей моделью является вызов функции:

```
function hello(username) {
    return "hello, " + username;
}
hello("Keyser Söze"); // "hello, Keyser Söze"
```

Этот код выполняет именно то, что в нем показано: вызывает функцию `hello` и связывает параметр `username` с переданным этому вызову аргументом.

Методы в JavaScript являются нечем иным, как свойствами объекта, которым повезло стать функциями:

```
var obj = {
    hello: function() {
        return "hello, " + this.username;
    },
    username: "Hans Gruber"
};
obj.hello(); // "hello, Hans Gruber"
```

Обратите внимание на то, как `hello` для обращения к свойствам `obj` ссылается на `this`. Вы могли бы предположить, что `this` связывается с `obj`, поскольку метод `hello` был определен для `obj`. Но мы можем скопировать ссылку на ту же самую функцию в другом объекте и получить другой ответ:

```
var obj2 = {
    hello: obj.hello,
    username: "Boo Radley"
};
obj2.hello(); // "hello, Boo Radley"
```

На самом деле в вызове метода вариант связывания `this`, также известный, как *получатель* вызова, определяет само выражение вызова. Выражение `obj.hello()` ищет свойство `hello` объекта `obj` и вызывает его с получателем `obj`. Выражение `obj2.hello()` ищет свойство `hello` объекта `obj2` (которое оказывается той же функцией, что и в вызове `obj.hello`), но вызывает его с получателем `obj2`.

В общем, вызов метода в отношении объекта приводит к поиску метода, а затем к использованию объекта в качестве получателя метода.

Поскольку методы являются не чем иным, как функциями, вызванными для конкретного объекта, ничто не мешает ссылаться на `this` и обычной функции:

```
function hello() {
    return "hello, " + this.username;
}
```

Это может пригодиться для предопределения функции для ее совместного использования несколькими объектами:

```
var obj1 = {
    hello: hello,
    username: "Gordon Gekko"
};
obj1.hello(); // "hello, Gordon Gekko"
var obj2 = {
    hello: hello,
    username: "Biff Tannen"
};
obj2.hello(); // "hello, Biff Tannen"
```

Однако функции, использующие `this`, в качестве функций практически бесполезны (в отличие от применения в качестве методов):

```
hello(); // "hello, undefined"
```

Вызов функции, не оформленный как вызов метода, предоставляет в качестве получателя глобальный объект, который в данном случае не имеет свойства по имени `username` и поэтому выдает значение `undefined`. Вызов метода в качестве функции редко делает что-либо полезное, если метод зависит от `this`, поскольку нет никаких причин надеется, что глобальный объект будет соответствовать тем ожиданиям, которые метод выстраивал в отношении вызываемого объекта. Фактически, предлагаемое по умол-

чанию связывание с глобальным объектом довольно проблематично, поскольку строгий режим ES5 по умолчанию связывает `this` с `undefined`:

```
function hello() {
    "use strict";
    return "hello, " + this.username;
}
hello(); // ошибка: невозможно прочитать свойство
         // "username",
         // относящееся к неопределенному (undefined)
         // объекту
```

Это помогает выявлять ситуации ошибочного использования методов в качестве простых функций за счет того, что они приводят к сбою намного быстрее: при попытке обратиться к свойствам объекта `undefined` тут же выдается сообщение об ошибке.

Третьей разновидностью функций являются конструкторы. Точно так же, как методы и простые функции, конструкторы определяются с помощью ключевого слова `function`:

```
function User(name, passwordHash) {
    this.name = name;
    this.passwordHash = passwordHash;
}
```

Вызов `User` с помощью оператора `new` позволяет считать эту функцию конструктором:

```
var u = new User("sfalken",
"0ef33ae791068ec64b502d6cb0191387");
u.name; // "sfalken"
```

В отличие от вызовов функций и методов, вызов конструктора передает в качестве значения `this` совершенно новый объект и неявно возвращает в качестве результата новый объект. Основная роль функции-конструктора заключается в инициализации объекта.

УЗЕЛКИ НА ПАМЯТЬ

- Вызовы методов предоставляют в качестве получателя объект, в котором ведется поиск свойства метода.
- Вызовы функций предоставляют в качестве своих получателей глобальный объект (или `undefined` для функций, выполняемых в строгом режиме). Вызов методов с использованием синтаксиса вызова функции вряд ли может принести какую-либо пользу.
- Конструкторы вызываются с помощью оператора `new` и получают в качестве своих получателей совершенно новый объект.

19 НАУЧИТЕСЬ ПОЛЬЗОВАТЬСЯ ФУНКЦИЯМИ ВЫСШЕГО ПОРЯДКА

Понятие *функций высшего порядка* раньше часто использовалось приверженцами функционального программирования; это был термин для посвященных, то, что представлялось как передовая методика программирования. На самом деле ничего подобного. Элегантная лаконичность функций зачастую позволяет получить более простой и компактный код. С годами языки создания сценариев переняли эту методику, сняв покровы таинственности с некоторых идиом функционального программирования.

Функции высшего порядка — это не что иное, как функции, получающие в качестве аргументов другие функции или возвращающие функции в качестве своих результатов. Получение функции в качестве аргумента (которая часто называется *функцией обратного вызова*, поскольку она «ответно вызывается» функцией высшего порядка) является довольно эффективной и выразительной идиомой, которая интенсивно используется в программах, написанных на JavaScript.

Рассмотрим стандартный метод `sort` для работы с массивами. Чтобы работать со всевозможными массивами,

метод `sort` с помощью вызывающего кода определяет, как сравнивать два элемента массива:

```
function compareNumbers(x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
}
[3, 1, 4, 1, 5, 9].sort(compareNumbers); // [1, 1, 3,
                                         //  4, 5, 9]
```

Для передачи вызывающему коду объекта с методом сравнения может понадобиться стандартная библиотека, но поскольку требуется только один метод, проще и компактнее воспользоваться функцией непосредственным образом. В сущности, если взять безымянную функцию, показанный пример можно упростить еще больше:

```
[3, 1, 4, 1, 5, 9].sort(function(x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
}); // [1, 1, 3, 4, 5, 9]
```

Приобретение навыков использования функций высшего порядка зачастую может помочь упростить ваш код и избавиться от неприятных стереотипов в программировании. Многие широко распространенные операции с массивами имеют довольно привлекательные абстракции высшего порядка, с которыми стоит ознакомиться. Рассмотрим простое преобразование строкового массива. Используя цикл, можно было бы написать:

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = [];
for (var i = 0, n = names.length; i < n; i++) {
    upper[i] = names[i].toUpperCase();
}
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Однако существует весьма удобный метод `map`, ориентированный на работу с массивами (и появившийся в ES5). Он позволяет полностью исключить элементы цикла, реализовав поэлементное преобразование с помощью локальной функции:

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = names.map(function(name) {
    return name.toUpperCase();
});
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Как только вы научитесь пользоваться функциями высшего порядка, можно будет приступать к написанию собственных. Верным признаком такой возможности является наличие дублированного или похожего кода. Представим, к примеру, что одна из частей программы создает строку из букв алфавита:

```
var aIndex = "a".charCodeAt(0); // 97
var alphabet = "";
for (var i = 0; i < 26; i++) {
    alphabet += String.fromCharCode(aIndex + i);
}
alphabet; // "abcdefghijklmnopqrstuvwxyz"
```

А другая часть программы создает строку, содержащую цифры:

```
var digits = "";
for (var i = 0; i < 10; i++) {
    digits += i;
}
digits; // "0123456789"
```

И еще в какой-то части программы создается строка произвольных символов:

```
var random = "";
for (var i = 0; i < 8; i++) {
    random +=
    String.fromCharCode(Math.floor(Math.random() * 26)
                                        + aIndex);
}
random; // "bdwvfrtp" (каждый раз будет другой
        //  результат)
```

В каждом примере создается строка, не похожая на другие, но во всех примерах используется общая логика. В каждом цикле строка генерируется путем объединения результатов некоторого вычисления для создания каждого отдельного сегмента. Мы можем выделить общие части и переместить их в одну вспомогательную функцию:

```
function buildString(n, callback) {
    var result = "";
    for (var i = 0; i < n; i++) {
        result += callback(i);
    }
    return result;
}
```

Обратите внимание на то, что в реализации `buildString` содержатся все общие части каждого цикла, а на изменяемом месте этих частей используется параметр: количество проходов цикла становится переменной `n`, а конструирование каждого строкового сегмента обеспечивает вызов функции обратного вызова. Теперь с помощью `buildString` можно упростить все три примера:

```
var alphabet = buildString(26, function(i) {
    return String.fromCharCode(aIndex + i);
});
alphabet; // "abcdefghijklmnopqrstuvwxyz"
var digits = buildString(10, function(i) { return i; });
digits; // "0123456789"
```

```
var random = buildString(8, function() {
    return
     String.fromCharCode(Math.floor(Math.random() * 26)
                                + aIndex);
});
random; // "ltvisfjr" (каждый раз будет другой
        //  результат)
```

От создания абстракций высшего порядка можно добиться многих преимуществ. Если есть сложные части реализации, например, получение правильных условий границ цикла, они локализируются в реализацию, представляющую собой функцию высшего порядка. Это позволяет исправлять любые ошибки в логике только один раз, а не «охотиться» на них в каждом экземпляре шаблонного кода, разбросанного по всей программе. Если потребуется оптимизировать производительность операции, у вас опять же будет только одно место, где потребуется что-то изменить. И наконец, если дать абстракции простое и понятное имя, например `buildString`, то тем, кто будет читать ваш код, проще будет разобраться с тем, что он делает, не разбираясь с деталями реализации.

Привычка применять функции высшего порядка, когда обнаруживается повторение одного и того же шаблонного кода, приводит к созданию более компактного кода, обладающего более высокой производительностью и удобством чтения. Умение выявлять общие шаблонные фрагменты кода и перемещать их во вспомогательные функции высшего порядка является для разработчика весьма важным навыком.

УЗЕЛКИ НА ПАМЯТЬ

- Функции высшего порядка — это функции, которые получают другие функции в качестве аргументов или возвращают функции в качестве результата.
- Изучите функции высшего порядка из существующих библиотек.
- Научитесь находить общие шаблонные фрагменты кода, которые можно переместить в функции высшего порядка.