**Внутреннее устройство поисковых методов**

Эта глава не обязательна при первом чтении учебника.

Если вы хотите действительно глубоко понимать, что происходит при поиске, то посмотрите эту главу. Если нет – её можно пропустить.

Несмотря на схожесть в синтаксисе, поисковые методы get\* и querySelector\* внутри устроены очень по-разному.

**[document.getElementById(id)](http://learn.javascript.ru/searching-elements-internals" \l "document-getelementbyid-id)**

Браузер поддерживает у себя внутреннее соответствие id -> элемент. Поэтому нужный элемент возвращается сразу. Это очень быстро.

**[elem.querySelector(query), elem.querySelectorAll(query)](http://learn.javascript.ru/searching-elements-internals" \l "elem-queryselector-query-elem-queryselectorall-query)**

Чтобы найти элементы, удовлетворяющие поисковому запросу, браузер не использует никаких сложных структур данных.

Он просто перебирает все подэлементы внутри элемента elem(или по всему документу, если вызов в контексте документа) и проверяет каждый элемент на соответствие запросу query.

Вызов querySelector прекращает перебор после первого же найденного элемента, а querySelectorAll собирает найденные элементы в «псевдомассив»: внутреннюю структуру данных, по сути аналогичную массиву JavaScript.

Этот перебор происходит очень быстро, так как осуществляется непосредственно движком браузера, а не JavaScript-кодом.

Оптимизации:

* В случае поиска по ID: elem.querySelector('#id'), большинство браузеров оптимизируют поиск, используя вызов getElementById.
* Последние результаты поиска сохраняются в кеше. Но это до тех пор, пока документ как-нибудь не изменится.

**[elem.getElementsBy\*(…)](http://learn.javascript.ru/searching-elements-internals" \l "elem-getelementsby)**

Результаты поиска getElementsBy\* – живые! При изменении документа – изменяется и результат запроса.

Например, найдём все div при помощи querySelectorAll и getElementsByTagName, а потом изменим документ:

<div></div>

<script>

var resultGet = document.getElementsByTagName('div');

var resultQuery = document.querySelectorAll('div');

alert( resultQuery.length + ', ' + resultGet.length ); // 1, 1

document.body.innerHTML = ''; // удалить всё содержимое BODY

alert( resultQuery.length + ', ' + resultGet.length ); // 1, 0

</script>

Как видно, длина коллекции, найденной через querySelectorAll, осталась прежней. А длина коллекции, возвращённой getElementsByTagName, изменилась.

Дело в том, что результат запросов getElementsBy\* – это не массив, а специальный объект, имеющий тип [NodeList](http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html" \l "ID-536297177)или [HTMLCollection](http://www.w3.org/TR/DOM-Level-2-HTML/html.html" \l "ID-75708506). Он похож на массив, так как имеет нумерованные элементы и длину, но внутри это не готовая коллекция, а «живой поисковой запрос».

Собственно поиск выполняется только при обращении к элементам коллекции или к её длине.

**[Алгоритмы getElementsBy\*](http://learn.javascript.ru/searching-elements-internals" \l "algoritmy-getelementsby)**

Поиск getElementsBy\* наиболее сложно сделать эффективно, так как его результат – «живая» коллекция, она должна быть всегда актуальной для текущего состояния документа.

var elems = document.getElementsByTagName('div');

alert( elems[0] );

// изменили документ

alert( elems[0] ); // результат может быть уже другой

Можно искать заново при каждой попытке получить элемент из elems. Тогда результат будет всегда актуален, но поиск будет работать уж слишком медленно. Да и зачем? Ведь, скорее всего, документ не поменялся.

**Чтобы производительность getElementsBy\* была достаточно хорошей, активно используется кеширование результатов поиска.**

Для этого есть два основных способа: назовём их условно «Способ Firefox» (Firefox, IE) и «Способ WebKit» (Chrome, Safari, Opera).

Для примера, рассмотрим поиск в произвольном документе, в котором есть 1000 элементов div.

Посмотрим, как будут работать браузеры, если нужно выполнить следующий код:

// вместо document может быть любой элемент

var elems = document.getElementsByTagName('div');

alert( elems[0] );

alert( elems[995] );

alert( elems[500] );

alert( elems.length );

**Способ Firefox**

Перебрать подэлементы document.body в порядке их появления в поддереве. Запоминать *все найденные элементы*во внутренней структуре данных, чтобы при повторном обращении обойтись без поиска.

Разбор действий браузера при выполнении кода выше:

1. Браузер создаёт пустую «живую коллекцию» elems. Пока ничего не ищет.
2. Перебирает элементы, пока не найдёт первый div. Запоминает его и возвращает.
3. Перебирает элементы дальше, пока не найдёт элемент с индексом 995. Запоминает все найденные.
4. Возвращает ранее запомненный элемент с индексом 500, без дополнительного поиска!
5. Продолжает обход поддерева с элемента, на котором остановился (995) и до конца. Запоминает найденные элементы и возвращает их количество.

**Способ WebKit**

Перебирать подэлементы document.body. Запоминать только один, *последний найденный*, элемент, а также, по окончании перебора – длину коллекции.

Здесь кеширование используется меньше.

Разбор действий браузера по строкам:

1. Браузер создаёт пустую «живую коллекцию» elems. Пока ничего не ищет.
2. Перебирает элементы, пока не найдёт первый div. Запоминает его и возвращает.
3. Перебирает элементы дальше, пока не найдёт элемент с индексом 995. Запоминает его и возвращает.
4. Браузер запоминает только последний найденный, поэтому не помнит об элементе 500. Нужно найти его перебором поддерева. Этот перебор можно начать либо с начала – вперед по поддереву, 500-й по счету) либо с элемента 995 – назад по поддереву, 495-й по счету. Так как назад разница в индексах меньше, то браузер выбирает второй путь и идёт от 995-го назад 495 раз. Запоминает теперь уже 500-й элемент и возвращает его.
5. Продолжает обход поддерева с 500-го (не 995-го!) элемента и до конца. Запоминает число найденных элементов и возвращает его.

Основное различие – в том, что Firefox запоминает все найденные, а Webkit – только последний. Таким образом, «метод Firefox» требует больше памяти, но гораздо эффективнее при повторном доступе к предыдущим элементам.

А «метод Webkit» ест меньше памяти и при этом работает не хуже в самом важном и частом случае – последовательном переборе коллекции, без возврата к ранее выбранным.

**Запомненные элементы сбрасываются при изменениях DOM.**

Документ может меняться. При этом, если изменение может повлиять на результаты поиска, то запомненные элементы необходимо сбросить. Например, добавление нового узла div сбросит запомненные элементы коллекции elem.getElementsByTagName('div').

Сбрасывание запомненных элементов при изменении документа выполняется интеллектуально.

1. Во-первых, при добавлении элемента будут сброшены только те коллекции, которые могли быть затронуты обновлением. Например, если в документе есть два независимых раздела <section>, и поисковая коллекция привязана к первому из них, то при добавлении во второй – она сброшена не будет.

Если точнее – будут сброшены все коллекции, привязанные к элементам вверх по иерархии от непосредственного родителя нового div и выше, то есть такие, которые потенциально могли измениться. И только они.

1. Во-вторых, если добавлен только div, то не будут сброшены запомненные элементы для поиска по другим тегам, например elem.getElementsByTagName('a').
2. …И, конечно же, не любые изменения DOM приводят к сбросу кешей, а только те, которые могут повлиять на коллекцию. Если где-то добавлен новый атрибут элементу – с кешем для getElementsByTagName ничего не произойдёт, так как атрибут никак не может повлиять на результат поиска по тегу.

Прочие поисковые методы, такие как getElementsByClassName тоже сбрасывают кеш при изменениях интеллектуально.

Разницу в алгоритмах поиска легко «пощупать». Посмотрите сами:

<script>

for (var i = 0; i < 10000; i++) document.write('<span> </span>');

var elements = document.body.getElementsByTagName('span');

var len = elements.length;

var d = new Date;

for (var i = 0; i < len; i++) elements[i];

alert( "Последовательно: " + (new Date - d) + "мс" ); // (1)

var d = new Date;

for (var i = 0; i < len; i += 2) elements[i], elements[len - i - 1];

alert( "Вразнобой: " + (new Date - d) + "мс" ); // (2)

</script>

В примере выше первый цикл проходит элементы последовательно. А второй – идет по шагам: один с начала, потом один с конца, потом ещё один с начала, ещё один – с конца, и так далее.

Количество обращений к элементам одинаково.

* В браузерах, которые запоминают все найденные (Firefox, IE) – скорость будет одинаковой.
* В браузерах, которые запоминают только последний (Webkit) – разница будет порядка 100 и более раз, так как браузер вынужден бегать по дереву при каждом запросе заново.