# **<h1>Intersection Observer API. Примеры использования одного из самых универсальных и полезных инструментов во фронтенде.</h1>**

Спецификация описывает данный API как инструмент, позволяющий отслеживать видимость и позицию DOM элементов (“targets”) относительно их родителя или видимой части экрана браузера(“root”). API является асинхронным, что существенно упрощает взаимодействие с пользователем, кроме того, со стороны разработки мы получаем универсальный способ, который отлично подходит для решения многих проблем, которые ранее приходилось реализовывать сначала из разу в раз.

Возьмем к примеру данный фрагмент кода, который мы использовали в прошлом для отслеживания позиции элемента:

let targets = document.querySelectorAll('.target');

let windowHeight = window.innerHeight;

window.addEventListener("scroll", function () {

targets.forEach(function(target) {

let targetPositionTop = target.getBoundingClientRect().top;

if (targetPositionTop - windowHeight <= 0) {

//Какие-то действия

}

});

});

Данный подход является медленным, поскольку использование обработчиков событий и циклов, на каждой итерации которых вызывается [Element.getBoundingClientRect()](https://developer.mozilla.org/ru/docs/Web/API/Element/getBoundingClientRect), заставляет браузер делать [re-layout](https://gist.github.com/paulirish/5d52fb081b3570c81e3a) всей страницы, что вносит значительный ущерб сайту. Кроме того, подобный код работает в основном потоке, что провоцирует проблемы с производительностью.

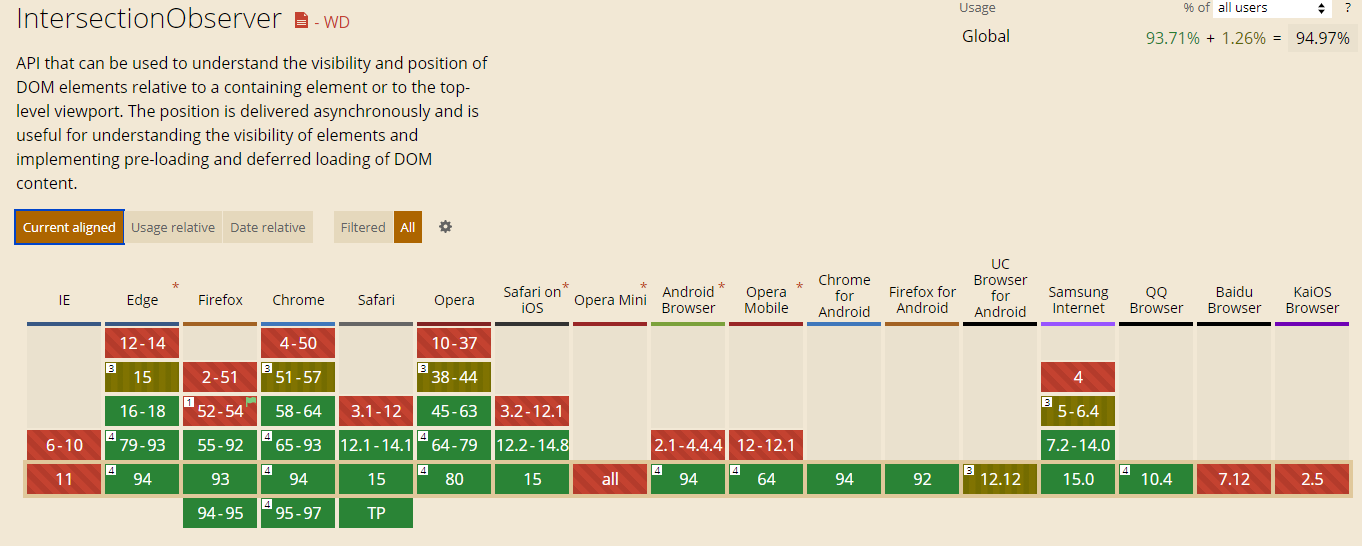
В свою очередь IntersectionObserver производит отслеживание видимости целевого элемента намного эффективнее. Асинхронность данного API устраняет необходимость в дорогостоящих запросах к DOM и стилям. Таким образом, больше нет необходимости вычислять пересечение элементов в основном потоке, и браузер может оптимизировать эти процессы на своё усмотрение, существенно снижая нагрузку на CPU и GPU.

# 

# **<h2>Инициализация и настройка Intersection Observer.</h2>**

Целью данной статьи является представление примеров использования API, но несмотря на то, что руководств по настройке в сети более чем [достаточно](https://developer.mozilla.org/ru/docs/Web/API/Intersection_Observer_API), со своей стороны все же считаю необходимым описать ключевые особенности работы с Intersection Observer.

1. Первое, с чего хотелось бы начать - с проверки поддержки IOA в браузере.



*1. Поддержка IOA к октябрю 2021 года на caniuse к на ок*

Для того, чтобы удостовериться в поддержке IOA в вашем браузере, достаточно проверить наличие переменной “IntersectionObserver” в глобальном объекте “window”:

if ('IntersectionObserver' in window) {

console.log("IntersectionObserver поддерживается");

}

Intersection Observer API имеет довольно хорошую поддержку в современных браузерах, но, несмотря на это, для старых версий есть [полифилл](https://github.com/w3c/IntersectionObserver/tree/main/polyfill).

1. С помощью функции-конструктора “IntersectionObserver” нужно создать объект-наблюдатель, указать для него 2 аргумента - функцию для вызова и настройки отслеживания:

let observer = new IntersectionObserver(callback, options);

Функция “callback”, которую мы создадим позже, будет вызвана всякий раз для элемента (“target”) при пересечении его с областью видимости документа или заданным родительским элементом (“root”) с теми настройками( объект “options”), которые мы передадим.

let options = {

root: document.querySelector(“box”),

rootMargin: '0px',

threshold: 1.0

}

**root** - элемент, который используется в качестве области просмотра для проверки видимости целевого элемента. В том случае, если данный параметр не задан, областью просмотра будет viewport..

**rootMargin** - определяет отступы вокруг корневого элемента для расширения или сжатия кадра захвата, когда размеры корневого элемента не обеспечивают достаточной гибкости.

**threshold** - принимает значениям от 0 до 1 и представляет процент пересечения целевого элемента с областью просмотра. Значение 0 будет означать, что пересечение составляет 0%, значение 1 - 100%. Если данный параметр не установлен, используется значение по умолчанию 0. В тех случаях, когда вам нужно следить за одним и тем же элементом на разных процентных соотношениях, необходимо установить несколько пороговых значений в виде массива в объекте параметров, например:

let options= {

root:null,

rootMargin:'0px',

threshold:[0, 0.25, 0.5, 0.75, 1]

};

let callback = function(entries, observer) {

entries.forEach(function(entry) {

// Действия, которые необходимо применить к каждому элементу, позицию которого мы отслеживаем:

entry.time;

entry.rootBounds;

entry.boundingClientRect;

entry.intersectionRect;

entry.intersectionRatio;

entry.target;

entry.isIntersecting;

})

}

Функция обратного вызова(функция “callback”) принимает аргумент “entries” - массив наблюдаемых нами элементов и объект созданного наблюдателя. Внутри функции находится цикл, который применяет логику, необходимую для каждого entry. IntersectionObserver может принимать только один элемент для наблюдения, именно поэтому используем метод [Array.prototype.forEach()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach), чтобы наблюдать за каждым отдельно.

Рассмотрим, какие из свойств объекта entry нам доступны:

entry.time - время, за которое произошло изменение

entry.rootBounds - результат вызова getBoundingClientRect() для области просмотра

entry.boundingClientRect - результат вызова getBoundingClientRect() для наблюдаемого элемента

entry.intersectionRect - пересечение двух прямоугольников(рассчитаных выше) и эффективно сообщает вам, какая часть наблюдаемого элемента видна

entry.intersectionRatio - сообщает, какая часть элемента видна

entry.target - информация об элементе, который пересекся с областью просмотра

entry.isIntersecting - булевое значение, отображающее, пересекается ли наблюдаемый элемент с областью просмотра

1. Создаем переменную “target”, в которую кладет тот элемент или элементы, за которыми будем наблюдать.

let target = document.querySelector('.item');

1. Последнее, что нам необходимо - запустить отслеживание изменений пересечения целевого элемента с областью просмотра в любом направлении, для чего используем функцию “observe” объекта “observer” с аргументом, которым является переменная “target”

observer.observe(target);

Так выглядит код в целом:

if ('IntersectionObserver' in window) {

let options = {

root: document.querySelector(“box”),

rootMargin: '0px',

threshold: 1.0

}

let callback = function(entries, observer) {

entries.forEach(function(entry) {

// Действия, которые необходимо применить к каждому элементу, позицию которого мы отслеживаем:

})

}

let observer = new IntersectionObserver(callback, options);

let target = document.querySelector('.item');

observer.observe(target);

}

# 

# **<h2>Примеры использования</h2>**

# **<h3>Ленивая загрузка изображений</h3>**

Один из самых очевидных и популярных примеров, который в прошлом мы реализовывали при помощи подключения сторонних инструментов.

Сейчас же, при наличии Intersection Observer API, ленивую загрузку реализовать крайне просто:

//html

<img data-src="https://images.pexels.com/photos/4406626/pexels-photo-4406626.jpeg" src="" alt="" class="lazy">

<img data-src="https://images.pexels.com/photos/1557651/pexels-photo-1557651.jpeg" src="" alt="" class="lazy">

<img data-src="https://images.pexels.com/photos/4590789/pexels-photo-4590789.jpeg" src="" alt="" class="lazy">

//css

img {

width: 100%;

height: 700px;

object-fit: cover;

object-position: center;

}

//js

<script>

if ('IntersectionObserver' in window) {

let observer = new IntersectionObserver(

function(entries, observer) {

entries.forEach(function(entry) {

if(entry.isIntersecting) {

entry.target.src = entry.target.dataset.src;

observer.unobserve(entry.target);

}

});

},

{

rootMargin: '0px',

threshold: 0

});

let targets = document.querySelectorAll('.lazy');

targets.forEach(function(target) {

observer.observe(target);

});

}

</script>

1. Создаем на странице 3 изображения с классом “lazy” и data-атрибутом “src”, в котором находится URL изображения.
2. Создаем и конфигурируем объект Intersection Observer.

* В функции обратного вызова перебираем в цикле все вхождения и применяем функцию isIntersecting, которая сообщает, пересекается ли изображение с областью просмотра и если да, берем его значение data-атрибута “data-src” и назначаем его атрибуту “src”. Как только “src” атрибут был заполнен, выполняем функцию “unobserve” объекта “observer” для того, чтобы пересечение целевого объекта с областью просмотра больше не отслеживалось.
* Конфигурируем объект опций, где по умолчанию областью просмотра будет viewport, отступы вокруг элемента опции “rootMargin” будут равны 0, а процентный порог пересечения “threshold” будет равен 0%

1. Находим все элементы с классом “lazy” и запускаем функцию “observe” объекта “observer” для отслеживания пересечения целевого элемента с областью просмотра.



Отмечу, что в переходном периоде уже находится свойство “[loading](https://developer.mozilla.org/ru/docs/Web/Performance/Lazy_loading)”, которое нам даст возможность еще больше упростить ленивую загрузку ресурсов и отказаться даже от Intersection Observer API.

**<h3>Переходы и анимация</h3>**

Мы часто встречаем библиотеки для переходов и анимации по событию “скролла”, например [wow.js](https://wowjs.uk/), [aos.js](https://michalsnik.github.io/aos/) и многие другие.

Функционал Intersection Observer API позволяет нам используя все тот же код, сконфигурировать переходы для элементов, когда они попадают в область просмотра.

//html

<div>

<div class="item"></div>

<div class="item"></div>

<div class="item"></div>

</div>

//css

.item {

width: 300px;

height: 300px;

margin: 700px 0;

transition: transform 1s ease;

}

.item:first-child {

background-color: #ff0000;

}

.item:nth-child(2) {

background-color: #00ff00;

}

.item:last-child {

background-color: #8a2be2;

}

.item.active {

transform: translateX(100%);

}

//js

<script>

if ('IntersectionObserver' in window) {

let observer = new IntersectionObserver(

function(entries, observer) {

entries.forEach(function(entry, index) {

if(entry.isIntersecting) {

setTimeout(function() {

entry.target.classList.add("active");

observer.unobserve(entry.target);

}, index \* 1000)

}

});

},

{

rootMargin: '0px',

threshold: 1.0

});

let targets = document.querySelectorAll(".item");

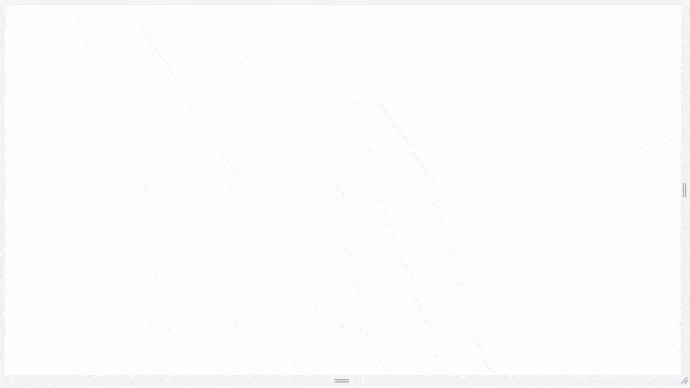
targets.forEach(function(target) {

observer.observe(target);

});

}

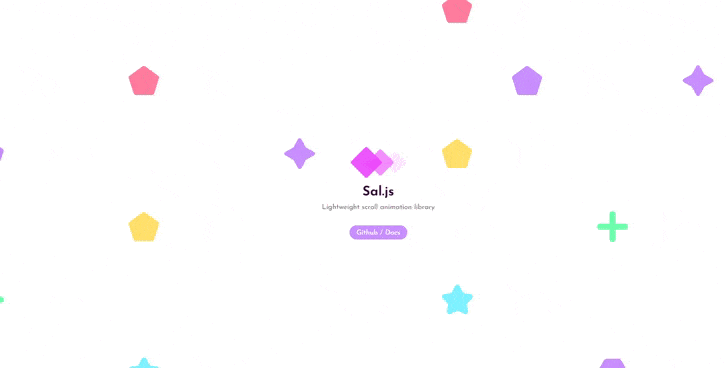
</script>



Мы наблюдаем за 3-мя элементами с классом “item” и как только из них каждый попадает в область просмотра 50% своей высоты, сразу же добавляется класс “active”, добавляющий другое стилистическое поведение элементу.

А что на счет своей библиотеки анимаций? Нет проблем!

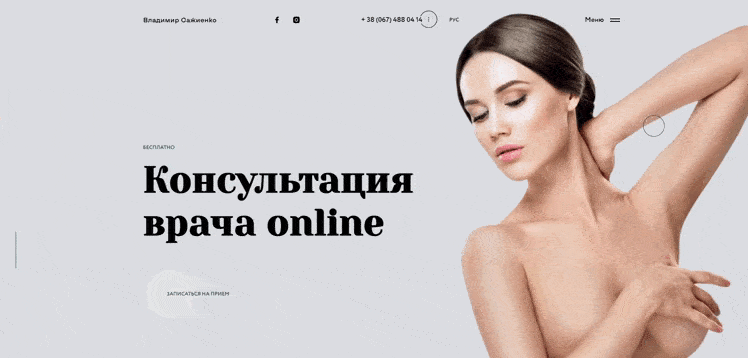
Вы можете создать библиотеку анимаций, подобную [sal.js](https://mciastek.github.io/sal/), построенную на Intersection Observer API.



Кроме того, на рынке существует большое количество библиотек, которые требуют дополнительных решений для запуска анимаций по событию “скролла”.

Одна из них - [anime.js](https://animejs.com/), для которой триггером, который отлавливает позицию элемента в области просмотра является библиотека [Waypoints](http://imakewebthings.com/waypoints/).

Не так давно я создал мини-инструмент, который позволяет триггерить анимацию библиотеки anime.js - [anime-in-viewport](https://github.com/dmitriyakkerman/anime-in-viewport), преимуществом которого является маленький вес и простота в использовании.



**<h3>Бесконечная прокрутка</h3>**

Бесконечная прокрутка - способ организации контента на странице. Когда юзер доходит до нижней части страницы, дополнительный контент подгружается автоматически. Данный способ помогает существенно ускорить начало взаимодействия пользователя с сайтом за счет того, что на первом этапе мы загружаем лишь необходимую информацию.

Прием бесконечной прокрутки используют такие известные ресурсы как Facebook, Instagram, Twitter, Youtube и многие другие.

//html

<div class="root">

<div id="container" class="container"></div>

<div id="loading" class="loading">Загрузка...</div>

</div>

//css

img {

width: 700px;

height: 700px;

object-fit: cover;

object-position: center;

}

.root {

display: flex;

flex-direction: column;

justify-content: center;

width: 65%;

height: 500px;

margin: 0 auto;

overflow-y: scroll;

}

.loading {

padding: 10px;

margin: 25px 0;

text-align: center;

border-top: solid 1px;

border-bottom: solid 1px;

}

//js

<script>

if ('IntersectionObserver' in window) {

let container = document.getElementById("container");

let loader = document.getElementById("loading");

let images = [ 'https://images.pexels.com/photos/4406626/pexels-photo-4406626.jpeg?auto=compress&cs=tinysrgb&dpr=2&h=750&w=1260', 'https://images.pexels.com/photos/1557651/pexels-photo-1557651.jpeg?auto=compress&cs=tinysrgb&dpr=2&h=750&w=1260', 'https://images.pexels.com/photos/4590789/pexels-photo-4590789.jpeg?auto=compress&cs=tinysrgb&dpr=2&h=750&w=1260'

];

let imagesLoadedCount = 0;

function loadMore() {

let image = document.createElement("img");

image.src = images[imagesLoadedCount];

container.appendChild(image);

imagesLoadedCount++;

}

let observer = new IntersectionObserver(function(entries, observer) {

entries.forEach((entry) => {

if (entry.isIntersecting) {

if(imagesLoadedCount < images.length) {

setTimeout(function() {

loadMore();

}, 1000)

}

else {

loader.remove();

}

}

});

},

{

rootMargin: "0px",

threshold: 0

}

);

observer.observe(loader);

}

</script>

Допустим, у нас имеется 3 изображения, которые хотим подгружать частично по 1. Для этого создаем объект наблюдателя с двумя аргументами:

* функцией, которая будет отслеживать позицию элемента “loader” на странице и при пересечении его с областью просмотра по условию запускать другую функцию “loadMore”, которая будет добавлять в контейнер новые изображения
* объектом опций

Соответственно, когда пользователь проскроллит до элемента “loader”, он увидит текст “Загрузка...”, после чего через некоторое время получит 1 изображение и так далее, пока количество загруженных изображений не станет эквивалентным общему количеству.



**<h3>Навигация по странице</h3>**

Навигация по странице - удобный инструмент, который не только помогает перемещаться по документу, но и отображать, на каком ее этапе мы находимся. С Intersection Observer API подобную навигацию реализовать крайне просто и быстро:

//html

<div class="page-wrapper">

<nav class="header-nav">

<ul class="header-nav\_\_wrapper">

<li class="header-nav\_\_item header-nav\_\_item--active" data-section-number="1"><a href="#section-1">Section 1</a></li>

<li class="header-nav\_\_item" data-section-number="2"><a href="#section-2">Section 2</a></li>

<li class="header-nav\_\_item" data-section-number="3"><a href="#section-3">Section 3</a></li>

<li class="header-nav\_\_item" data-section-number="4"><a href="#section-4">Section 4</a></li>

<li class="header-nav\_\_item" data-section-number="5"><a href="#section-5">Section 5</a></li>

</ul>

</nav>

<div class="section\_\_wrapper" id="section-1" data-section-number="1"><span class="section\_\_label" >Section 1</span></div>

<div class="section\_\_wrapper" id="section-2" data-section-number="2"><span class="section\_\_label">Section 2</span></div>

<div class="section\_\_wrapper" id="section-3" data-section-number="3"><span class="section\_\_label">Section 3</span></div>

<div class="section\_\_wrapper" id="section-4" data-section-number="4"><span class="section\_\_label">Section 4</span></div>

<div class="section\_\_wrapper" id="section-5" data-section-number="5"><span class="section\_\_label">Section 5</span></div>

</div>

//css

html {

height: 100vh;

overflow-y: hidden;

}

.page-wrapper {

width: 100%;

height: 100vh;

overflow-y: scroll;

scroll-snap-type: proximity;

scroll-snap-points-y: repeat(100vh);

scroll-snap-type: y proximity;

scroll-behavior: smooth;

background-color: #E7E5DE;

}

.header-nav {

width: 100%;

position: -webkit-sticky;

position: sticky;

top: 0;

height: 70px;

margin-bottom: -70px;

}

.header-nav\_\_wrapper {

width: 100%;

display: flex;

flex-direction: row;

}

.header-nav\_\_item {

font-family: 'Lato', sans-serif;

flex-grow: 1;

display: flex;

justify-content: center;

}

.header-nav a {

text-decoration: none;

font-weight: bold;

text-transform: uppercase;

padding: 20px 50px 18px 50px;

border-bottom: solid 1.5px transparent;

color: #000000;

}

.header-nav > a:hover,

.header-nav\_\_item--active > a {

border-bottom-color: #000000;

}

.section\_\_wrapper {

height: 100vh;

width: 100%;

min-height: 500px;

display: flex;

justify-content: center;

align-items: center;

scroll-snap-align: start;

}

.section span {

font-family: 'Lato', sans-serif;

font-weight: 100;

text-transform: lowercase;

font-size: 100px;

}

.section\_\_wrapper:nth-child(odd) {

background-color: #bfbfbf;

color: #F5F5F6;

}

.section\_\_wrapper:nth-child(even) {

background-color: #F5F5F6;

color: #607d8b;

}

//js

<script>

if ('IntersectionObserver' in window) {

let navItems = document.querySelectorAll('.header-nav\_\_item');

let observer = new IntersectionObserver(function (entries) {

entries.forEach(function (entry) {

if (entry.isIntersecting) {

let sectionNumber = entry.target.getAttribute('data-section-number');

navItems.forEach(function (item) {

item.classList.remove('header-nav\_\_item--active');

})

document.querySelector('[data-section-number="' + sectionNumber + '"]').classList.add('header-nav\_\_item--active')

}

});

}, {

threshold: 0.5

});

let sections = document.querySelectorAll('.section\_\_wrapper');

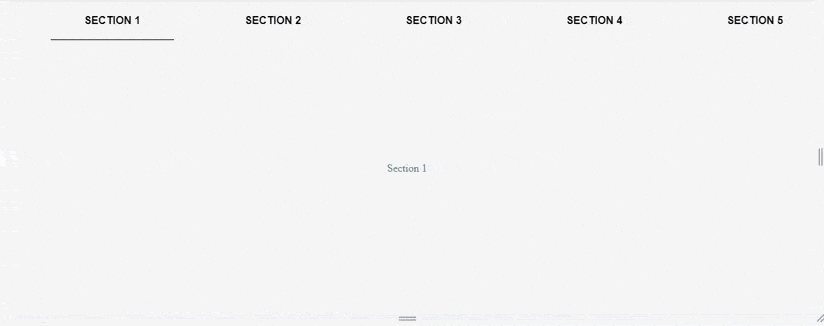
sections.forEach(function (section) {

observer.observe(section);

});

}

</script>



**<h2>Вывод</h2>**

Intersection Observer API - довольно простая, но очень полезная технология, которая позволяет нам делать всевозможные вещи, связанные с обнаружением элементов в области просмотра, одновременно помогая добиться действительно хорошего прироста производительности.