> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

Strict mode

Режим *strict* (*строгий* режим), введённый в ECMAScript 5 , позволяет использовать более строгий вариант JavaScript. Это не просто подмножество языка: в нем сознательно используется семантика, отличающаяся от обычно принятой. Не поддерживающие строгий режим браузеры будут по-другому выполнять код, написанный для строгого режима, поэтому не полагайтесь на строгий режим без тестирования поддержки используемых особенностей этого режима. Строгий и обычный режим могут сосуществовать одновременно, а скрипт может переключаться в строгий режим по мере надобности.

Строгий режим принёс ряд изменений в обычную семантику JavaScript. Во-первых, строгий режим заменяет исключениями некоторые ошибки, которые интерпретатор JavaScript ранее молча пропускал. Во-вторых, строгий режим исправляет ошибки, которые мешали движкам JavaScript выполнять оптимизацию — в некоторых случаях код в строгом режиме может быть оптимизирован для более быстрого выполнения, чем код в обычном режиме. В-третьих, строгий режим запрещает использовать некоторые элементы синтаксиса, которые, вероятно, в следующих версиях ECMAScript получат особый смысл.

Если вы хотите изменить свой код так, чтобы он работал в строгой версии JavaScript, посмотрите статью Переход к строгому режиму .

Активизация строгого режима

Строгий режим применяется ко *всему скрипту* или к *отдельным функциям*. Он не может быть применён к блокам операторов, заключённых в фигурные скобки — попытка использовать его в подобном контексте будет проигнорирована. Код в `eval`, `Function`, в атрибутах обработчиков событий, в строках, переданных в `setTimeout`, и т.п. рассматривается как законченный скрипт, и активизация строгого режима в нём выполняется ожидаемым образом.

Строгий режим для скриптов

Чтобы активизировать строгий режим для всего скрипта, нужно поместить оператор `"use strict";` или `'use strict';` перед всеми остальными операторами скрипта (выдержать приведённый синтаксис буквально).

```
JS
```

```
var v = "Привет! Я скрипт в строгом режиме!";
```

В этой синтаксической конструкции кроется ловушка, в которую уже угодили даже самые известные сайты : нельзя бездумно объединять скрипты с разными режимами. Объединение скрипта в строгом режиме со скриптом в обычном выглядит как скрипт в строгом режиме! Справедливо и обратное: объединение обычного скрипта со строгим выглядит как нестрогий скрипт. Объединение только строгих или только обычных скриптов проходит без последствий, проблему вызывает совместное использование скриптов со строгим и обычным режимом. Поэтому рекомендуется включать строгий режим только на уровне функций (хотя бы в течение переходного периода).

Вы также можете использовать подход "обёртывания" всего содержимого скрипта в функцию, для которой включён строгий режим. Это уменьшит возможность возникновения проблем при объединении скриптов, но одновременно потребует явно экспортировать из контекста функции все глобальные переменные.

Строгий режим для функций

Аналогично, чтобы включить строгий режим для функции, поместите оператор `"use strict";` (или `'use strict';`) в тело функции перед любыми другими операторами.

JS

```
function strict() {
  // Строгий режим на уровне функции
  "use strict";
  function nested() {
    return "И я тоже!";
  }
  return "Привет! Я функция в строгом режиме! " + nested();
}
function notStrict() {
  return "Я не strict.";
}
```

Строгий режим для модулей

ECMAScript 2015 представил модули JavaScript и, следовательно, 3-й способ войти в строгий режим. Все содержимое модулей JavaScript автоматически находится в строгом режиме, и для его запуска не требуется никаких инструкций.

```
function strict() {
    // Потому что это модуль, я strict по умолчанию
}
export default strict;
```

Изменения в строгом режиме

Строгий режим изменяет синтаксис и поведение среды исполнения. Изменения главным образом попадают в следующие категории: преобразование ошибок в исключения; изменения, упрощающие вычисление переменной в определённых случаях использования её имени; изменения, упрощающие `eval` и `arguments`; изменения, упрощающие написание "безопасного" JavaScript, и изменения, предвосхищающие дальнейшее развитие ECMAScript.

Преобразование ошибок в исключения

Строгий режим превращает некоторые прощавшиеся ранее ошибки в исключения. JavaScript был разработан с расчётом на низкий порог вхождения, и временами он придаёт заведомо ошибочным операциям семантику нормального кода. Иногда это помогает срочно решить проблему, а иногда это создаёт худшие проблемы в

будущем. Строгий режим расценивает такие ошибки как ошибки времени выполнения, для того чтобы они могли быть обнаружены и исправлены в обязательном порядке.

Во-первых, строгий режим делает невозможным случайное создание глобальных переменных. В обычном JavaScript опечатка в имени переменной во время присваивания приводит к созданию нового свойства глобального объекта, и выполнение продолжается (хотя в современном JavaScript оно, вероятно, аварийно завершится в дальнейшем). Присваивания, которые могут случайно создать глобальную переменную, в строгом режиме выбрасывают исключение:

JS

```
"use strict";
// Предполагая, что не существует глобальной переменной
mistypeVaraible = 17; // mistypedVaraible, эта строка выбросит ReferenceError
// из-за опечатки в имени переменной
```

Во-вторых, строгий режим заставляет присваивания, которые всё равно завершились бы неудачей, выбрасывать исключения. Например, `NaN` — глобальная переменная, защищённая от записи. В обычном режиме присваивание `NaN` значения ничего не делает; разработчик не получает никакого сообщения об ошибке. В строгом режиме присваивание `NaN` значения выбрасывает исключение. Любое присваивание, которое в обычном режиме завершается неудачей (присваивание значения свойству, защищённому от записи; присваивание значения свойству, доступному только на чтение; присваивание нового свойства нерасширяемому объекту), в строгом режиме выбросит исключение:

JS

```
"use strict";

// Присваивание значения глобальной переменной, защищённой от записи
var undefined = 5; // выдаст TypeError
var Infinity = 5; // выдаст TypeError

// Присваивание значения свойству, защищённому от записи
var obj1 = {};
Object.defineProperty(obj1, "x", { value: 42, writable: false });
obj1.x = 9; // выдаст TypeError

// Присваивание значения свойству, доступному только для чтения
```

```
var obj2 = {
  get x() {
    return 17;
  },
};
obj2.x = 5; // выдаст TypeError

// Задание нового свойства нерасширяемому объекту
var fixed = {};
Object.preventExtensions(fixed);
fixed.newProp = "ohai"; // выдаст TypeError
```

В-третьих, в строгом режиме попытки удалить неудаляемые свойства будут вызывать исключения (в то время как прежде такая попытка просто не имела бы эффекта):

JS

```
"use strict";
delete Object.prototype; // выдаст TypeError
```

В-четвёртых, строгий режим требует, чтобы все свойства, перечисленные в сериализованном объекте, встречались только один раз. В обычном коде имена свойств могут дублироваться, а значение свойства определяется последним объявлением. Но, в таком случае, дублирование — просто почва для багов, если код редактируется с тем, чтобы поменять значение свойства как-то по-другому, кроме изменения последнего объявления. Дублирование имён свойств в строгом режиме является синтаксической ошибкой:

> Примечание: Это уже не является проблемой в ECMAScript 2015 (Firefox bug 1041128).

JS

```
"use strict";
var o = { p: 1, p: 2 }; // !!! синтаксическая ошибка
```

В-пятых, строгий режим требует, чтобы имена аргументов в объявлении функций встречались только один раз. В обычном коде последний повторённый аргумент

скрывает предыдущие аргументы с таким же именем. Эти предыдущие аргументы всё ещё доступны через `arguments[i]`, так что они не полностью потеряны. Тем не менее, такое сокрытие несёт в себе мало смысла и, скорее всего, не имеет под собой цели (например, может скрывать опечатку), поэтому в строгом режиме дублирование имён аргументов является синтаксической ошибкой:

JS

```
function sum(a, a, c) {
  // !!! синтаксическая ошибка
  "use strict";
  return a + a + c; // ошибка, если код был запущен
}
```

В-шестых, строгий режим запрещает синтаксис восьмеричной системы счисления. Восьмеричный синтаксис не является частью ECMAScript, но поддерживается во всех браузерах с помощью дописывания нуля спереди к восьмеричному числу: `0644 === 420` и `"\045" === "%"`. В ECMAScript 2015 восьмеричное число поддерживается также с помощью дописывания перед числом "`0o`". Т.е.

```
var a = 0o10; // ES2015: Восмеричное
```

Иногда начинающие разработчики думают, что ведущий ноль не имеет семантического значения, и используют его для выравнивания — но это меняет значение числа! Восьмеричный синтаксис редко бывает полезен и может быть неправильно использован, поэтому строгий режим считает восьмеричные числа синтаксической ошибкой:

JS

```
"use strict";
var sum =
  015 + // !!! синтаксическая ошибка
  197 +
  142;

var sumWithOctal = 0o10 + 8;
console.log(sumWithOctal); // 16
```

В-седьмых, строгий режим в ECMAScript 2015 запрещает установку свойств primitive значениям. Без строгого режима, установка свойств просто игнорируется (no-op), со строгим режимом, однако, выдаёт TypeError .

```
(function() {
'use strict';

false.true = '';         // TypeError
(14).sailing = 'home';   // TypeError
'with'.you = 'far away'; // TypeError

})();
```

Упрощение работы с переменными

Строгий режим упрощает сопоставление имени переменной с местом её определения в коде. Многие оптимизации времени компиляции полагаются на возможность считать, что переменная *X* хранится в *этом конкретном* месте исходного кода. Иногда, при компиляции JavaScript простое сопоставление имени переменной с местом её определения в коде не возможно, без выполнения того самого кода. Строгий же режим исключает большинство таких случаев, благодаря чему оптимизации компилятора работают эффективнее.

Во-первых, строгий режим запрещает использование with . Проблема с with в том, что во время выполнения любое имя внутри блока может ссылаться как на свойство обрабатываемого объекта, так и на переменную в окружающем (или даже в глобальном) контексте — невозможно знать об этом заранее. Строгий режим считает with синтаксической ошибкой, поэтому не остаётся шанса использовать имя переменной внутри with для ссылки на неизвестное место во время выполнения:

JS

```
"use strict";
var x = 17;
with (obj) {
  // !!! синтаксическая ошибка
  // Если код не в строгом режиме, то будет ли x ссылаться на переменную var x, или
  // на свойство obj.x? Предугадать без запуска кода невозможно,
  // следовательно такой код не может быть эффективно оптимизирован.
```

```
  x;
}
```

Простая альтернатива `with` уже существует — присваивание объекта переменной с коротким именем и затем доступ к нужному свойству как свойству этой переменной.

Во-вторых, [`eval` в строгом режиме не добавляет новых переменных в окружающий контекст]. В обычном режиме, при вызове `eval("var x;")` переменная `x` добавится в область видимости окружающей функции либо в глобальный контекст. В общем случае, это означает, что в каждой функции, в которой присутствует вызов `eval`, имена переменных которые не ссылаются на аргумент или локальную переменную, должны сопоставляться с местом их определения в коде только во время выполнения (потому что `eval` мог ввести новую переменную, которая может перекрыть внешнюю переменную). В строгом режиме `eval` создаёт переменные только в контексте выполняемого кода, так что `eval` не может повлиять на то, ссылается ли имя на локальную или на внешнюю переменную:

JS

```
var x = 17;
var evalX = eval("'use strict'; var x = 42; x");
console.assert(x === 17);
console.assert(evalX === 42);
```

Соответственно, если функция `eval` вызвана непосредственно в форме выражения `eval(...)`, то внутри кода в строгом режиме, передаваемый в неё код будет выполнен в строгом режиме. Передаваемый код может содержать в себе включение строгого режима, но в этом нет необходимости.

JS

```
function strict1(str) {
  "use strict";
  return eval(str); // str будет выполнен как код строгого режима
}
function strict2(f, str) {
  "use strict";
  return f(str); // не eval(...): str выполнится в строгом режиме только в том
  // случае, если в нем содержится вызов строгого режима
}
function nonstrict(str) {
```

```
  return eval(str); // str выполнится в строгом режиме только в том
  // случае, если в нем содержится вызов строгого режима
}
strict1("'Строгий режим!'");
strict1("'use strict'; 'Строгий режим!'");
strict2(eval, "'Не строгий режим.'");
strict2(eval, "'use strict'; 'Строгий режим!'");
nonstrict("'Не строгий режим.'");
nonstrict("'use strict'; 'Строгий режим!'");
```

Таким образом, имена в строгом коде, передаваемом в `eval`, ведут себя так же, как имена в нестрогом коде, передаваемом в `eval` внутри строгого режима.

В-третьих, строгий режим запрещает удаление простых имён. `delete name` в строгом режиме является синтаксической ошибкой:

JS

```
"use strict";

var x;
delete x; // !!! синтаксическая ошибка

eval("var y; delete y;"); // !!! синтаксическая ошибка
```

Упрощение `eval` и `arguments`

В строгом режиме снижается количество странностей в поведении `arguments` и `eval`, оба из которых примешивают определённое количество магии в обычный код. Так `eval` добавляет или удаляет переменные и меняет их значения, а переменная `arguments` может удивить своими проиндексированными свойствами, которые являются ссылками (синонимами) для поименованных аргументов функции. Строгий режим делает большой шаг в прояснении этих двух ключевых слов, но полное их обуздание произойдёт лишь в следующей редакции ECMAScript.

Во-первых, ключевые слова `eval` и `arguments` не могут быть переопределены или изменены. Все подобные попытки это сделать являются синтаксическими ошибками:

JS

```
"use strict";
eval = 17;
arguments++;
++eval;
var obj = { set p(arguments) {} };
var eval;
try {
} catch (arguments) {}
function x(eval) {}
function arguments() {}
var y = function eval() {};
var f = new Function("arguments", "'use strict'; return 17;");
```

Во-вторых, в строгом режиме поля объекта `arguments` не связаны с проименованными аргументами функции, а являются их продублированными копиями значений. В обычном коде внутри функции, первым аргументом которой является `arg`, изменение значения переменной `arg` также меняет значение и у поля `arguments[0]`, и наоборот (кроме случаев, когда аргумент в функцию не передан, или `arguments[0]` удалён). В строгом режиме `arguments` хранит копии значений аргументов переданных при вызове функции. `arguments[i]` не отслеживает изменений соответствующего именованного аргумента, и именованный аргумент не отслеживает значение соответствующего arguments[i].

JS

```
function f(a) {
  "use strict";
  a = 42;
  return [a, arguments[0]];
}
var pair = f(17);
console.assert(pair[0] === 42);
console.assert(pair[1] === 17);
```

В-третьих, свойство `arguments.callee` больше не поддерживается. В обычном коде свойство `arguments.callee` ссылается на саму функцию для вызова которой и был создан объект `arguments`. Малоприменимое свойство, так как функция заранее известна, и к ней можно обратиться и по её имени непосредственно. Более того, `arguments.callee` значительно затрудняет такую оптимизацию, как инлайнинг, потому как должна быть сохранена возможность обратиться к незаинлайненой функции на

случай, если присутствует обращение к arguments.callee. В строгом режиме arguments.callee превращается в неудаляемое свойство, которое выбрасывает предостерегающее исключение при любой попытке обращения к нему:

JS

```
"use strict";
var f = function () {
  return arguments.callee;
};
f(); // выдаст TypeError
```

"Безопасный" JavaScript

Строгий режим упрощает написание "безопасного" JavaScript-кода. Сейчас некоторые веб-сайты предоставляют пользователям возможность писать JavaScript, который будет выполняться на сайте *от имени других пользователей.* В браузерах, JavaScript может иметь доступ к приватной информации пользователя, поэтому, в целях ограничения доступа к запретной функциональности, такой JavaScript перед выполнением должен быть частично преобразован. Гибкость JavaScript делает это практически невозможным без многочисленных проверок во время исполнения. Функционал, исполняемый языком иногда столь массивен, что выполнение любых дополнительных проверок во время исполнения скрипта приведёт к значительной потере производительности. Однако, некоторые особенности строгого режима, плюс обязательное требование того, чтобы JavaScript, загруженный пользователем, имел строгий режим и вызывался определённым способом, существенно снижают потребность в таких проверках.

Во-первых, значение, передаваемое в функцию как `this`, в строгом режиме не приводится к объекту (не "упаковывается"). В обычной функции `this` всегда представляет собой объект: либо это непосредственно объект, в случае вызова с `this`, представляющим объект-значение; либо значение, упакованное в объект, в случае вызова с `this` типа Boolean, string, или number; либо глобальный объект, если тип `this` это `undefined` или `null`. (Для точного определения конкретного `this` используйте `call`, `apply`, или `bind`.) Автоматическая упаковка не только снижает производительность, но и выставляет на показ глобальный объект, что в браузерах является угрозой безопасности, потому что глобальный объект предоставляет доступ к функциональности, которая должна быть ограничена в среде "безопасного"

JavaScript. Таким образом, для функции в строгом режиме точно определённый `this` не упаковывается в объект, а если не определён точно, `this` является `undefined`:

JS

```
"use strict";
function fun() {
  return this;
}
console.assert(fun() === undefined);
console.assert(fun.call(2) === 2);
console.assert(fun.apply(null) === null);
console.assert(fun.call(undefined) === undefined);
console.assert(fun.bind(true)() === true);
```

Во-вторых, в строгом режиме больше не представляется возможным осуществлять "прогонку" стека JavaScript посредством базовых расширений ECMAScript. В обычном коде, использующем эти расширения, когда функция `fun` находится в процессе своего вызова, `fun.caller` представляет собой функцию, вызвавшую `fun`, а `fun.arguments` это `аргументы` для данного вызова `fun`. Оба расширения являются проблемными для "безопасного" JavaScript, так как они позволяют "безопасному" коду получить доступ к "привилегированным" функциям и их (потенциально небезопасным) аргументам. Если `fun` находится в строгом режиме, то `fun.caller`, так же как и `fun.arguments`, представляют собой неудаляемые свойства, которые приведут к вызову исключения при попытке их чтения или записи:

JS

```
function restricted() {
  "use strict";

  restricted.caller; // выдаст TypeError
  restricted.arguments; // выдаст TypeError
}
function privilegedInvoker() {
  return restricted();
}
privilegedInvoker();
```

В-третьих, в функциях строгого режима свойство `arguments` больше не предоставляет доступ к переменным, созданным внутри функции. В некоторых

предыдущих реализациях ECMAScript `arguments.caller` представлял собой объект, свойства которого являлись ссылками на переменные, созданные внутри функции при её вызове. Это представляет собой угрозу безопасности , так как нарушает возможность скрывать приватные данные внутри функций (замыканий). Также это делает невозможными большинство оптимизаций. Исходя из этих причин, ни один из современных браузеров не реализует этого поведения. Но всё же, ввиду своей исторической функциональности, `arguments.caller` для функций в строгом режиме всё ещё является неудаляемым свойством, которое вызывает исключение при попытке его чтения или записи:

JS

```
"use strict";
function fun(a, b) {
  "use strict";

  var v = 12;
  return arguments.caller; // выдаст TypeError
}
fun(1, 2); // не выводит v (или a, или b)
```

Подготовка почвы для будущих версий ECMAScript

В будущих версиях ECMAScript с высокой вероятностью появится новый синтаксис, и для упрощения перехода на новые версии, в строгом режиме ECMAScript 5 введено несколько ограничений. Всегда проще вносить изменения в стандарт, если заранее подготовить для них основу в строгом режиме.

Во-первых, в строгом режиме зарезервирован для использования следующий список ключевых слов: `implements` , `interface` , `let` , `package` , `private` , `protected` , `public` , `static` и `yield` . В строгом режиме, следовательно, вы не можете задействовать эти слова для именования или обращения к переменным или аргументам.

JS

```
function package(protected) {
  // !!!
  "use strict";
  var implements; // !!!

  // !!!
```

```
  interface: while (true) {
    break interface; // !!!
  }

  function private() {} // !!!
}
function fun(static) {
  "use strict";
} // !!!
```

Два замечания, специфичных для Mozilla: Первое, если ваш код создан на JavaScript 1.7 или выше (например, chrome code, или тег `<script type="">` заполнен правильно), и применён строгий режим, то `let` и `yield` имеют ту же функциональность, которая у них была изначально, когда они только появились. Однако в веб, в строгом коде загруженном через `<script src="">` или `<script>...</script>`, нельзя будет использовать `let` / `yield` в качестве идентификаторов. Второе, в то время как ES5 зарезервировал слова `class`, `enum`, `export`, `extends`, `import` и `super` для любого режима, в Firefox 5 Mozilla они были зарезервированы намного раньше и лишь для строгого режима.

Во-вторых, в строгом режиме запрещается объявление функций глубже самого верхнего уровня скрипта или функции . В обычном коде в браузерах, объявление функций позволено "везде", *что не является частью ES5 (или даже ES3!)* Это расширение различных браузеров, не имеющее общего совместимого подхода. Есть надежда, что в последующих редакциях ECMAScript будет определена новая семантика для объявления функций вне верхнего уровня скрипта или функции. Запрет на объявление таких функций в строгом режиме производит "зачистку" для спецификации в будущем релизе ECMAScript:

JS

```
"use strict";
if (true) {
  function f() {} // !!! синтаксическая ошибка
  f();
}
for (var i = 0; i < 5; i++) {
  function f2() {} // !!! синтаксическая ошибка
  f2();
}
```

```
function baz() {
  // верно
  function eit() {} // тоже верно
}
```

Данный запрет не является особенностью строгого режима, потому что такое объявление функций является одним из расширений основного ES5. Но это рекомендация комитета ECMAScript, и браузеры реализуют её.

Строгий режим в браузерах

В большинстве браузеров в настоящее время строгий режим реализован. Однако не стоит впадать в слепую зависимость от него, потому что существует множество Версий браузеров, поддерживающих строгий режим лишь частично или вовсе не поддерживающих оный (например, Internet Explorer ниже версии 10!). *Строгий режим изменяет семантику.* Надежда на эти изменения приведёт к ошибкам и погрешностям в браузерах, в которых строгий режим не реализован. Проявляйте осторожность при использовании строгого режима, и подкрепляйте надёжность строгого режима тестами особенностей, которые проверяют, насколько верно реализованы его фрагменты. Наконец, старайтесь *тестировать свой код в браузерах, как поддерживающих, так и не поддерживающих строгий режим*. Если вы проводите тестирование только в тех браузерах, которые не поддерживают строгий режим, то вполне вероятно у вас появятся проблемы в браузерах, его поддерживающих, и наоборот.

Смотрите также

- Where's Walden? » New ES5 strict mode support: now with poison pills!
- Where's Walden? » New ES5 strict mode requirement: function statements not at top level of a program or function are prohibited
- Where's Walden? » New ES5 strict mode support: new vars created by strict mode eval code are local to that code only
- John Resig - ECMAScript 5 Strict Mode, JSON, and More
- ECMA-262-5 in detail. Chapter 2. Strict Mode.
- Strict mode compatibility table

This page was last modified on 28 нояб. 2023 г. by MDN contributors.