Приложения

A

ES2018 и ES2019

Начиная с ECMAScript 2015, комитет TC-39 начал выпускать новую спецификацию ECMA каждый год. Это позволяет собрать все отдельные предложения, которые находятся на достаточно продвинутой стадии, и упаковать их в единый пакет. Однако эта упаковка имеет ограниченное значение, поскольку производители браузеров, как правило, принимают предложения по частям. Когда предложение достигнет стадии 4, его поведение не изменится, только, скорее всего, оно будет включено в следующую версию ECMAScript и браузеры начнут применять функции предложения по своему усмотрению.

Предложение ECMAScript 2018 было завершено в январе 2018 года и содержит улучшения для асинхронной итерации, операторов остатка и распространения, регулярных выражений и промисов. TC-39 поддерживает GitHub-репозиторий (https://github.com/tc39/ecma262), который можно использовать для отслеживания состояния различных предложений:

TODO FIX ASYNC ITERATION

TODO ES2019 http://exploringjs.com/es2018-es2019/toc.html

> **ПРИМЕЧАНИЕ** Поскольку функции, описанные в этой главе, новые, они будут поддерживаться браузером в ограниченном виде (если вообще будут). Обратитесь к https://caniuse.com/, чтобы определить, поддерживает ли версия браузера определенную функцию.

АСИНХРОННАЯ ИТЕРАЦИЯ

Асинхронное выполнение и протокол итератора — две чрезвычайно распространенные темы в новых функциях ECMAScript последних выпусков. Асинхронное выполнение включает в себя высвобождение контроля над потоком выполнения, чтобы позволить

медленным операциям завершаться до восстановления управления, а протокол итератора включает определение канонического порядка для произвольных объектов. Асинхронная итерация — это просто логическое усвоение этих двух понятий.

Синхронный итератор предоставляет пару `{value, done}` каждый раз при вызове `next()`. Конечно, это требует, чтобы вычисления и извлечения ресурсов, необходимые для определения содержимого этой пары, были завершены к моменту выхода из вызова `next()`, иначе эти значения не будут определены. При использовании *синхронного* итератора для перебора значений, определенных *асинхронно*, основной поток выполнения будет заблокирован в ожидании завершения асинхронной операции.

С асинхронными итераторами эта проблема полностью решена. Асинхронный итератор предоставляет промис, который разрешается в пару `{value, done}` каждый раз при вызове `next()`. Таким образом, поток выполнения может быть освобожден и выполнит работу в другом месте, пока разрешается текущая итерация цикла.

Создание и использование асинхронного итератора

Асинхронные итераторы проще всего понять при сравнении с традиционными синхронными итераторами. Ниже приведен простой класс `Emitter`, который содержит функцию синхронного генератора, создающего синхронный итератор, который будет считать от 0 до 4:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.syncIdx = 0;
    }

    *[Symbol.iterator]() {
        while(this.syncIdx < this.max) {
            yield this.syncIdx++;
        }
    }
}

const emitter = new Emitter(5);

function syncCount() {
    const syncCounter = emitter[Symbol.iterator]();

    for (const x of syncCounter) {
        console.log(x);
    }
}

syncCount();
// 0
// 1
// 2
// 3
// 4
```

Предыдущий пример работает только потому, что на каждой итерации следующее значение может быть сразу же получено. Если вместо этого вы не хотите блокировать основной поток выполнения при определении следующего значения для выдачи, вы также можете определить функцию асинхронного генератора, которая будет выдавать значения, обернутые в промис.

Это может быть выполнено с использованием асинхронных версий итераторов и генераторов. ECMAScript 2018 определяет `Symbol.asyncIterator`, который позволяет определять и вызывать функции генерации Promise. В спецификации также представлен асинхронный итератор цикла `for`, цикл `for-await-of`, предназначенный для использования этого асинхронного итератора.

С их использованием предыдущий пример может быть расширен для поддержки синхронной и асинхронной итерации:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.syncIdx = 0;
        this.asyncIdx = 0;
    }

    *[Symbol.iterator]() {
        while(this.syncIdx < this.max) {
            yield this.syncIdx++;
        }
    }
    async *[Symbol.asyncIterator]() {
        *[Symbol.asyncIterator]() {
            while(this.asyncIdx < this.max) {
                yield new Promise((resolve) => resolve(this.asyncIdx++;
            }
        }
    }

    const emitter = new Emitter(5);

    function syncCount() {
        const syncCounter = emitter[Symbol.iterator]();

        for (const x of syncCounter) {
            console.log(x);
        }
    }

    async function asyncCount() {
        const asyncCounter = emitter[Symbol.asyncIterator]();

        for await(const x of asyncCounter) {
            console.log(x);
        }
    }

syncCount();
```

```
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

Для дальнейшего понимания поменяйте приведенный выше пример так, чтобы синхронный генератор был передан в цикл `for-await-of`:

```
const emitter = new Emitter(5);

async function asyncIteratorSyncCount() {
    const syncCounter = emitter[Symbol.iterator]();

    for await(const x of syncCounter) {
        console.log(x);
    }
}

asyncIteratorSyncCount();
// 0
// 1
// AsyncIteratorExample01.js
2
// 3
// 4
```

Даже несмотря на то что синхронный счетчик перебирает примитивные значения, цикл `for-await-of` будет обрабатывать значения, как если бы они были возвращены, завернутые в промисы. Это демонстрирует мощь цикла ожидания, который позволяет ему свободно обрабатывать как синхронные, так и асинхронные итерации. Это неверно для обычного цикла `for`, который не может обрабатывать асинхронный итератор:

```
function syncIteratorAsyncCount() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for (const x of asyncCounter) {
        console.log(x);
    }
}

syncIteratorAsyncCount();
// TypeError: asyncCounter is not iterable
```

Одна из наиболее важных концепций асинхронных итераторов состоит в том, что обозначение `Symbol.asyncIterator` не изменяет поведение функции генератора или

способ его использования. Обратите внимание, что функция генератора определяется как асинхронная функция и обозначается как генератор с использованием звездочки. `Symbol.asyncIterator` просто обещает внешней конструкции, такой как цикл ожидания, что связанный итератор возвратит объекты-промисы последовательности.

Понимание очереди асинхронного итератора

Конечно, предыдущий пример является довольно надуманным, так как промисы, возвращаемые итератором, мгновенно разрешаются, и поэтому он представляет собой не что иное, как тонко завернутый синхронный итератор. Предположим вместо этого, что полученные промисы разрешаются через неопределенный период времени; более того, предположим, что они возвращаются вне очереди. Асинхронный итератор должен эмулировать синхронный итератор всеми возможными способами, включая выполнение по порядку кода, связанного с каждой итерацией. Для решения этой проблемы асинхронные итераторы поддерживают очередь обратных вызовов, чтобы гарантировать, что обработчик итератора для более раннего значения всегда завершит выполнение, прежде чем перейти к более позднему значению, даже если более позднее значение разрешается до более раннего значения.

Чтобы доказать это, асинхронный итератор в следующем примере возвращает промисы, которые разрешаются через произвольный период времени. Асинхронная очередь итераций гарантирует, что порядок разрешения промисов не влияет на порядок итераций. В результате целые числа будут напечатаны в нужном порядке (через случайные интервалы):

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.syncIdx = 0;
        this.asyncIdx = 0;
    }

    *[Symbol.iterator]() {
        while(this.syncIdx < this.max) {
            yield this.syncIdx++;
        }
    }
    async *[Symbol.asyncIterator]() {
        while(this.asyncIdx < this.max) {
            yield new Promise((resolve) => {
                setTimeout(() => {
                    resolve(this.asyncIdx++);
                }, Math.floor(Math.random() * 1000));
            });
        }
    }
}

const emitter = new Emitter(5);
```

```
function syncCount() {
    const syncCounter = emitter[Symbol.iterator]();

    for (const x of syncCounter) {
        console.log(x);
    }
}

async function asyncCount() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for await(const x of asyncCounter) {
        console.log(x);
    }
}

syncCount();
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
AsyncIteratorExample02.js
```

Обработка reject() в асинхронном итераторе

Поскольку композиция асинхронных итераторов состоит из промисов, необходимо учитывать возможность отклонения одного из промисов, созданных итератором. Поскольку проект асинхронной итерации требует завершения по порядку, нет смысла проходить через отклоненный промис в цикле; следовательно, отклоненный промис заставит итератор завершиться:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.asyncIdx = 0;
    }

    async *[Symbol.asyncIterator]() {
        while (this.asyncIdx < this.max) {
            if (this.asyncIdx < 3) {
                yield this.asyncIdx++;
            } else {
                throw 'Exited loop';
            }
        }
```

```
    }
}

const emitter = new Emitter(5);

async function asyncCount() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for await (const x of asyncCounter) {
        console.log(x);
    }
}

asyncCount();
// 0
// 1
// 2
// Uncaught (in promise) Exited loop
```

Ручное управление асинхронным итератором с использованием next()

Цикл `for-await-of` предлагает две полезные функции: он использует очередь асинхронных итераторов для обеспечения порядка выполнения и скрывает структуру промисов асинхронных итераторов. Однако использование такого цикла скрывает большую часть базового поведения.

Поскольку асинхронный итератор по-прежнему следует протоколу итератора, можно также легко пройти асинхронную итерацию, используя `next()`. Как описано ранее, `next()` содержит промис, который преобразуется в `{value, done}`. Это означает, что нужно использовать Promise API для извлечения методов, но это также означает, что не обязательно использовать очередь асинхронных итераторов.

```
const emitter = new Emitter(5);

const asyncCounter = emitter[Symbol.asyncIterator]();

console.log(asyncCounter.next());
// { value: Promise, done: false }
```

Асинхронные циклы верхнего уровня

Как правило, асинхронное поведение, включая циклы `for-await-of`, не может существовать вне асинхронной функции. Тем не менее может возникнуть необходимость использовать асинхронное поведение в таком контексте. Это может быть достигнуто путем создания асинхронного IIFE:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.asyncIdx = 0;
```

```
    }

    async *[Symbol.asyncIterator]() {
        while(this.asyncIdx < this.max) {
            yield new Promise((resolve) => resolve(this.asyncIdx++));
        }
    }
}

const emitter = new Emitter(5);

(async function() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for await(const x of asyncCounter) {
        console.log(x);
    }
})();

// 0
// 1
// 2
// 3
// 4
```

Реализация наблюдаемых объектов

Поскольку асинхронные итераторы будут терпеливо ждать следующей итерации, не неся при этом вычислительных затрат, открывается совершенно новый путь для реализации наблюдаемого интерфейса. На высоком уровне она примет форму захвата событий, оборачивания их в промисы, а затем передачи этих событий через итератор, чтобы позволить наблюдателю подключиться к асинхронному итератору. Когда событие запускается, следующий промис в асинхронном итераторе разрешается с этим событием.

> **ПРИМЕЧАНИЕ** Тема наблюдаемых объектов выходит за рамки этой книги, потому что они в основном реализованы в сторонних библиотеках. Для дальнейшего изучения загляните в чрезвычайно популярную библиотеку RxJS (http://reactivex.io/rxjs/).

Упрощенным примером этого будет захват видимого потока событий браузера. Для этого требуется очередь промисов, каждый из которых соответствует отдельному событию. Очередь также сохранит порядок, в котором генерируются события, что является желательным для такого рода проблем.

```
class Observable {
    constructor() {
        this.promiseQueue = [];

        // Содержит разрешатель для следующего промиса в очереди
        this.resolve = null;
```

```
        // Выдвигает начальный промис в очереди, который будет
        // разрешен с первым наблюдаемым событием
        this.enqueue();
    }

    // Создание нового промиса, сохранение его метода resolve и
    // хранение его в очереди
    enqueue() {
        this.promiseQueue.push(
            new Promise((resolve) => this.resolve = resolve));
    }

    // Удаление промиса из начала очереди и
    // его возврат
    dequeue() {
        return this.promiseQueue.shift();
    }
}
```

Чтобы использовать этот вывод из очереди промисов, определите метод асинхронного генератора для класса. Этот генератор должен работать для любого типа события:

```
class Observable {
    constructor() {
        this.promiseQueue = [];

        // Содержит разрешатель для следующего промиса в очереди
        this.resolve = null;

        // Выдвигает начальный промис в очереди, который будет
        // разрешен с первым наблюдаемым событием
        this.enqueue();
    }

    // Создание нового промиса, сохранение его метода resolve и
    // хранение его в очереди
    enqueue() {
        this.promiseQueue.push(
            new Promise((resolve) => this.resolve = resolve));
    }

    // Удаление промиса из начала очереди и
    // его возврат
    dequeue() {
        return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
        // Всякий раз, когда генерируется событие, промис в начале очереди
        // разрешается с помощью объекта события и
        // в очередь помещается другой промис.
        element.addEventListener(eventType, (event) => {
            this.resolve(event);
            this.enqueue();
        });
```

```
        // Каждый разрешенный промис в начале очереди будет
        // передавать объект события асинхронному итератору.
        while (1) {
            yield await this.dequeue();
        }
    }
}
```

С этим полностью определенным классом теперь можно легко определить наблюдаемый объект на элементах DOM. Предположим, что на странице есть кнопка `<button>`; можно записать поток событий `click` на этой кнопке и вывести каждое из них в консоль следующим образом:

```
class Observable {
    constructor() {
        this.promiseQueue = [];

        // Содержит разрешатель для следующего промиса в очереди
        this.resolve = null;

        // Выдвигает начальный промис в очереди, который будет
        // разрешен с первым наблюдаемым событием
        this.enqueue();
    }

    // Создание нового промиса, сохранение его метода resolve и
    // хранение его в очереди
    enqueue() {
        this.promiseQueue.push(
            new Promise((resolve) => this.resolve = resolve));
    }

    // Удаление промиса из начала очереди и
    // его возврат
    dequeue() {
        return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
        // Всякий раз, когда генерируется событие, промис в начале очереди
        // разрешается с помощью объекта события и
        // в очередь помещается другой промис.
        element.addEventListener(eventType, (event) => {
            this.resolve(event);
            this.enqueue();
        });

        // Каждый разрешенный промис в начале очереди будет
        // передавать объект события асинхронному итератору.
        while (1) {
            yield await this.dequeue();
        }
    }
}
```

```
(async function() {
    const observable = new Observable();

    const button = document.querySelector('button');
    const mouseClickIterator = observable.fromEvent(button, 'click');

    for await (const clickEvent of mouseClickIterator) {
        console.log(clickEvent);
    }
})();
```

ОПЕРАТОРЫ ОСТАТКА И РАСПРОСТРАНЕНИЯ ДЛЯ ЛИТЕРАЛОВ ОБЪЕКТОВ

В спецификации ECMAScript 2018 вся элегантность операторов остатка и распространения в массивах теперь также доступна внутри литералов объектов. Это позволяет объединять объекты или собирать свойства в новые объекты.

Оператор остатка

Оператор остатка позволяет использовать один оператор при деструктурировании объекта, чтобы собрать все оставшиеся неуказанные перечислимые свойства в один объект. Это можно сделать следующим образом:

```
const person = { name: 'Matt', age: 27, job: 'Engineer' };
const { name, ...remainingData } = person;

console.log(name);              // Matt
console.log(remainingData);     // { age: 27, job: 'Engineer' }
```

Оператор остатка может использоваться не более одного раза для каждого литерала объекта и должен быть указан последним. Поскольку для каждого литерала объекта может существовать только один оператор остатка, можно использовать вложенные операторы остатка. При вложении из-за отсутствия двусмысленности относительно того, какие элементы поддерева свойств выделяются любому заданному оператору остатка, результирующие объекты никогда не будут перекрываться по своему содержанию:

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { name, job: { title, ...remainingJobData }, ...remainingPersonData }
    = person;

console.log(name);                  // Matt
console.log(title);                 // Engineer
console.log(remainingPersonData);   // { age: 27 }
console.log(remainingJobData);      // { level: 10 }

const { ...a, job } = person;
// SyntaxError: Rest element must be last element
```

Оператор остатка выполняет поверхностное копирование между объектами, поэтому ссылки на объекты будут копироваться вместо создания полных копий объектов:

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { ...remainingData } = person;

console.log(person === remainingData);            // false
console.log(person.job === remainingData.job);    // true
```

Оператор остатка скопирует все перечисляемые собственные свойства, включая символы:

```
const s = Symbol();
const foo = { a: 1, [s]: 2, b: 3 }

const {a, ...remainingData} = foo;

console.log(remainingData);
// { b: 3, Symbol(): 2 }
```

Оператор распространения

Оператор распространения позволяет объединять два объекта таким же образом, как и конкатенация массивов. Оператор распространения, примененный к внутреннему объекту, выполнит поверхностное копирование всех перечисляемых собственных свойств, включая символы, во внешний объект:

```
const s = Symbol();
const foo = { a: 1 };
const bar = { [s]: 2 };

const foobar = {...foo, c: 3, ...bar};

console.log(foobar);
// { a: 1, c: 3 Symbol(): 2 }
```

Порядок, в котором распределяются объекты, имеет значение по двум причинам:

1. Объекты отслеживают порядок вставки. Свойства, скопированные из распространяемых объектов, будут выполняться в том порядке, в котором они перечислены внутри литерала объекта.
2. Объекты будут перезаписывать свойства при обнаружении дубликатов. Последнее найденное свойство будет тем, значение которого будет записано.

Эти соглашения о порядке показаны здесь:

```
const foo = { a: 1 };
const bar = { b: 2 };
const foobar = {c: 3, ...bar, ...foo};

console.log(foobar);
// { c: 3, b: 2, a: 1}
```

```
const baz = { c: 4 };

const foobarbaz = {...foo, ...bar, c: 3, ...baz };

console.log(foobarbaz);
// { a: 1, b: 2, c: 4 }
```

Как и в случае с оператором остатка, все выполняемые копии являются поверхностными:

```
const foo = { a: 1 };
const bar = { b: 2, c: { d: 3 } };

const foobar = {...foo, ...bar};

console.log(foobar.c === bar.c); // true
```

ОПИСАНИЕ FINALLY() В ПРОМИСАХ

Раньше существовали только неэлегантные способы определения поведения, которое имело бы место после того, как промис выходит из состояния «ожидания» независимо от результата. Обычно оно принимает форму утилизации обработчика:

```
let resolveA, rejectB;

function finalHandler() {
    console.log('finished');
}

function resolveHandler(val) {
    console.log('resolved');
    finalHandler();
}

function rejectHandler(err) {
    console.log('rejected');
    finalHandler();
}

new Promise((resolve, reject) => {
    resolveA = resolve;
})
.then(resolveHandler, rejectHandler);

new Promise((resolve, reject) => {
    rejectB = reject;
})
.then(resolveHandler, rejectHandler);

resolveA();
rejectB();
// resolved
```

```
// finished
// rejected
// finished
```

С `Promise.prototype.finally()` можно объединить все в один общий обработчик. Обработчику `finally()` не передаются никакие аргументы и он не знает, обрабатывает он разрешенный или отклоненный промис. Предыдущий пример может быть изменен следующим образом:

```
let resolveA, rejectB;

function finalHandler() {
    console.log('finished');
}

function resolveHandler(val) {
    console.log('resolved');
}

function rejectHandler(err) {
    console.log('rejected');
}

new Promise((resolve, reject) => {
    resolveA = resolve;
})
.then(resolveHandler, rejectHandler);
.finally(finalHandler);

new Promise((resolve, reject) => {
    rejectB = reject;
})
.then(resolveHandler, rejectHandler);
.finally(finalHandler);

resolveA();
rejectB();
// resolved
// rejected
// finished
// finished
```

Здесь вы заметите, что порядок ведения журнала изменился. Каждый `finally()` создает новый экземпляр промиса, и эти новые промисы добавляются в очередь микрозадач браузера и разрешаются только после разрешения предыдущих промисов обработчика.

РАСШИРЕНИЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

ECMAScript 2018 содержит несколько новых наворотов для регулярных выражений.

Флаг dotAll

Одна досадная особенность регулярных выражений заключалась в том, что токен совпадения с одним символом (точка «.») не соответствовал символам конца строки, таким как \n и \r, или не-BMP-символам, таким как эмодзи.

```
const text = `
foo
bar
`;

const re = /foo.bar/;

console.log(re.test(text));    // false
```

В этом предложении вводится флаг «s» (обозначает одиночную линию), который исправляет это поведение:

```
const text = `
foo
bar
`;

const re = /foo.bar/s;

console.log(re.test(text));    // true
```

Ретроспективные проверки

Регулярные выражения поддерживают как положительные, так и отрицательные опережающие проверки, которые позволяют декларировать ожидания после сопоставления сегментов:

```
const text = 'foobar';

// Положительная опережающая проверка
// Утверждение, что значение следует за указанным, но не захватывается
const rePositiveMatch = /foo(?=bar)/;
const rePositiveNoMatch = /foo(?=baz)/;

console.log(rePositiveMatch.exec(text));
// ["foo"]

console.log(rePositiveNoMatch.exec(text));
// null

// Отрицательная опережающая проверка
// Утверждение, что значение не следует за указанным, но не захватывается
const reNegativeNoMatch = /foo(?!bar)/;
const reNegativeMatch = /foo(?!baz)/;

console.log(reNegativeNoMatch.exec(text));
// null

console.log(reNegativeMatch.exec(text));
// ["foo"]
```

Новое предложение вводит зеркальное отражение этих проверок, положительные и отрицательные ретроспективные проверки. Они работают идентично опережающим проверкам, за исключением того, что они работают для проверки содержимого, предшествующего сопоставленным сегментам:

```
const text = 'foobar';

// Положительная ретроспективная проверка
// Утверждение, что значение предшествует указанному, но не захватывается
const rePositiveMatch = /(?<=foo)bar/;
const rePositiveNoMatch = /(?<=baz)bar/;

console.log(rePositiveMatch.exec(text));
// ["bar"]

console.log(rePositiveNoMatch.exec(text));
// null

// Отриательная ретроспективная проверка
// Утверждение, что значение не предшествует указанному, но не захватывается
const reNegativeNoMatch = /(?<!foo)bar/;
const reNegativeMatch = /(?<!baz)bar/;

console.log(reNegativeNoMatch.exec(text));
// null

console.log(reNegativeMatch.exec(text));
// ["bar"]
```

Именованные группы захвата

Как правило, в нескольких группах захвата обращение выполнялось по индексу, что оказалось ужасным разочарованием, потому что индексы не дают контекста относительно того, что они на самом деле содержат:

```
const text = '2018-03-14';

const re = /(\d+)-(\d+)-(\d+)/;

console.log(re.exec(text));
//["2018-03-14", "2018", "03", "14"]
```

Предложение позволяет связать действительный идентификатор JavaScript с группой захвата, которую затем можно извлечь из свойства `groups` результата:

```
const text = '2018-03-14';

const re = /(?<year>\d+)-(?<month>\d+)-(?<day>\d+)/;

console.log(re.exec(text).groups);
// { year: "2018", month: "03", day: "14" }
```

Экранирование свойств Unicode

Стандарт Unicode определяет свойства для каждого символа. Свойства символов, такие как имя символа, категории, обозначение пробелов, а также сценарий или язык, внутри которого определяется символ, доступны как свойства символов. Экранирование свойств Unicode позволяют использовать эти свойства внутри регулярных выражений.

Некоторые свойства являются двоичными, что означает, что они могут применяться автономно. Примерами этого являются `Uppercase` и `White_Space`. Другие свойства ведут себя как пары ключ/значение, где свойство будет соответствовать значению свойства. Примером этого является `Script_Extensions=Greek`.

Список свойств Unicode можно найти по адресу http://unicode.org/Public/UNIDATA/PropertyAliases.txt.

Список значений свойств Unicode можно найти по адресу http://unicode.org/Public/UNIDATA/PropertyValueAliases.txt.

Экранирование свойств Unicode в регулярных выражениях может использовать `\p` для выбора соответствия или `\P` для выбора несоответствия:

```
const pi = String.fromCharCode(0x03C0);
const linereturn = `
`;

const reWhiteSpace = /\p{White_Space}/u;
const reGreek = /\p{Script_Extensions=Greek}/u;
const reNotWhiteSpace = /\P{White_Space}/u;
const reNotGreek = /\P{Script_Extensions=Greek}/u;

console.log(reWhiteSpace.test(pi));               // false
console.log(reWhiteSpace.test(linereturn));       // true
console.log(reNotWhiteSpace.test(pi));            // true
console.log(reNotWhiteSpace.test(linereturn));    // false

console.log(reGreek.test(pi));               // true
console.log(reGreek.test(linereturn));       // false
console.log(reNotGreek.test(pi));            // false
console.log(reNotGreek.test(linereturn));    // true
```

МЕТОДЫ СГЛАЖИВАНИЯ МАССИВОВ

ECMAScript 2019 добавил два метода в прототип `Array`, `flat()` и `flatMap()`, которые значительно упрощают операции сглаживания массива. Без этих методов сглаживание — это неприятное дело, которое требует итеративного или рекурсивного решения.

> **ПРИМЕЧАНИЕ** `flat()` и `flatMap()` строго ограничены сглаживанием вложенных массивов. Вложенные итерируемые объекты, такие как `Map` и `Set`, не будут сглажены.

Array.prototype.flatten()

Ниже приведен пример того, как может выглядеть простая рекурсивная реализация без использования этих новых методов:

```
function flatten(sourceArray, flattenedArray = []) {
    for (const element of sourceArray) {
        if (Array.isArray(element)) {
            flatten(element, flattenedArray);
        } else {
            flattenedArray.push(element);
        }
    }

    return flattenedArray;
}

const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(flatten(arr))
// [0, 1, 2, 3, 4, 5, 6]
```

Во многих отношениях этот пример напоминает древовидную структуру данных; каждый элемент в массиве ведет себя как дочерний узел, а элементы, не являющиеся массивами, являются листьями. Следовательно, в этом примере входной массив представляет собой дерево высотой 2 с 7 листьями. Сглаживание этого массива по сути является упорядоченным обходом листьев.

Иногда полезно иметь возможность указать, сколько уровней вложенности массива должно быть сглажено. Рассмотрим следующий пример, который изменяет исходную реализацию и позволяет указать глубину выравнивания:

```
function flatten(sourceArray, depth, flattenedArray = []) {
    for (const element of sourceArray) {
        if (Array.isArray(element) && depth > 0) {
            flatten(element, depth - 1, flattenedArray);
        } else {
            flattenedArray.push(element);
        }
    }

    return flattenedArray;
}

const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(flatten(arr, 1))
// [0, 1, 2, 3, [4, 5], 6]
```

Для решения этих случаев использования был добавлен метод `Array.prototype.flat()`. Он принимает аргумент `depth` (по умолчанию с глубиной 1) и возвращает поверхностную копию экземпляра `Array`, сглаженного до указанной глубины. Это показано здесь:

```
const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(arr.flat(2));
// [0, 1, 2, 3, 4, 5, 6]

console.log(arr.flat());
// [0, 1, 2, 3, [4, 5], 6]
```

Поскольку выполняется поверхностное копирование, массивы с циклами будут копировать значения из исходного массива при сглаживании:

```
const arr = [[0], 1, 2, [3, [4, 5]], 6];

arr.push(arr);

console.log(arr.flat());
// [0, 1, 2, 3, 4, 5, 6, [0], 1, 2, [3, [4, 5]], 6]
```

Array.prototype.flatMap()

Метод `Array.prototype.flatMap()` позволяет выполнить операцию отображения перед сглаживанием массива. `arr.flatMap(f)` функционально эквивалентен `arr.map(f).flat()`, но `arr.flatMap()` более эффективен, поскольку браузер должен выполнять только один обход.

Сигнатура функции `flatMap()` идентична `map()`. Простой пример выглядит так:

```
const arr = [[1], [3], [5]];

console.log(arr.map(([x]) => [x, x + 1]));
// [[1, 2], [3, 4], [5, 6]]

console.log(arr.flatMap(([x]) => [x, x + 1]));
// [1, 2, 3, 4, 5, 6]
```

`flatMap()` особенно полезен в ситуациях, когда метод объекта, не являющегося массивом, возвращает массив — к примеру, `split()`. Рассмотрим следующий пример, где набор входных строк разбивается на слова и объединяется в массив из одного слова:

```
const arr = ['Lorem ipsum dolor sit amet,', 'consectetur adipiscing elit.'];

console.log(arr.flatMap((x) => x.split(/[\W+]/)));
// ["Lorem", "ipsum", "dolor", "sit", "amet", "", "consectetur", "adipiscing",
"elit", ""]
```

Удобный трюк (хотя он может привести к снижению производительности) — использовать пустой массив для фильтрации результатов после `map()`. В следующем примере расширяется приведенный выше пример для удаления пустых строк:

```
const arr = ['Lorem ipsum dolor sit amet,', 'consectetur adipiscing elit.'];

console.log(arr.flatMap((x) => x.split(/[\W+]/)).flatMap((x) => x || []));
// ["Lorem", "ipsum", "dolor", "sit", "amet", consectetur", "adipiscing", "elit"]
```

Здесь каждая пустая строка в результатах сначала отображается в пустой массив. При сглаживании они эффективно пропускаются в возвращаемом массиве.

> **ПРИМЕЧАНИЕ** Использование этой стратегии не рекомендуется, поскольку каждое фильтруемое значение требует создания нового экземпляра `Array`, который немедленно отбрасывается.

OBJECT. FROMENTRIES()

ECMAScript 2019 добавил статический метод `fromEntries()` в класс `Object`, который создает объект из набора пар массивов ключ—значение. Этот метод выполняет операцию, противоположную `Object.entries()`, и демонстрируется здесь:

```
const obj = {
    foo: 'bar',
    baz: 'qux'
};

const objEntries = Object.entries(obj);

console.log(objEntries);
// [["foo", "bar"], ["baz", "qux"]]

console.log(Object.fromEntries(objEntries));
// { foo: "bar", baz: "qux" }
```

Статический метод ожидает итерируемый объект, содержащий любое количество итерируемых объектов размером 2. Это особенно полезно в тех случаях, когда нужно преобразовать экземпляр `Map` в экземпляр `Object`, поскольку выходные данные итератора `Map` точно соответствуют сигнатуре, которую получает `fromEntries()`:

```
const map = new Map().set('foo', 'bar');

console.log(Object.fromEntries(map));
// { foo: "bar" }
```

МЕТОДЫ ОБРЕЗКИ СТРОК

ECMAScript 2019 добавил два метода к прототипу `String`, `trimStart()` и `trimEnd()`, которые позволяют выполнять целевое удаление пробелов. Эти методы предназначены для замены `trimLeft()` и `trimRight()`, которые имеют неоднозначное значение в контексте языков, читаемых справа налево, таких как арабский и иврит.

Эти два метода фактически противоположны `padStart()` и `padEnd()` с одним пробелом. Следующий пример добавляет пробел к строке, а затем удаляет ее с обеих сторон:

```
let s = ' foo ';

console.log(s.trimStart());    // "foo "
console.log(s.trimEnd());      // " foo"
```

SYMBOL.PROTOTYPE.DESCRIPTION

В ECMAScript 2019 добавлена возможность проверки необязательного описания `Symbol` через свойство `description`. Ранее это было доступно только тогда, когда символ был приведен к строке:

```
const s = Symbol('foo');

console.log(s.toString());
// Symbol(foo)
```

С выходом ES2019 каждый объект `Symbol` получил свойство `description` только для чтения, которое предоставляет описание. Если описания нет, по умолчанию используется значение `undefined`.

```
const s = Symbol('foo');

console.log(s.description);
// foo
```

НЕОБЯЗАТЕЛЬНАЯ ПРИВЯЗКА CATCH

До ES2019 структура блока `try/catch` была довольно жесткой. Даже если не нужно использовать перехваченный объект ошибки, парсер все равно потребует присвоения имени переменной объекту ошибки внутри условия `catch`:

```
try {
    throw 'foo';
} catch (e) {
    // Произойдет ошибка, но сам объект ошибки не имеет значения
}
```

В ES2019 можно опустить присвоение объекта ошибки и просто полностью игнорировать ошибку:

```
try {
     throw 'foo';
} catch {
    // Произойдет ошибка, но сам объект ошибки не имеет значения
}
```

ЕЩЕ НЕСКОЛЬКО УЛУЧШЕНИЙ

ES2019 также добавляет несколько настроек в существующий инструментарий:

- `Array.prototype.sort()` *стабилен*, что означает, что эквивалентные объекты не будут переупорядочены в выходных данных.
- Одиночные суррогатные символы UTF-16 не могут быть закодированы в UTF-8, что вызывает проблемы с `JSON.stringify()`. Вместо того чтобы возвращать непарные суррогатные кодовые точки в виде отдельных кодовых единиц UTF16, теперь они будут представлены экранированными последовательностями JSON.
- Ранее и U+2028 LINE SEPARATOR, и U+2029 PARAGRAPH SEPARATOR были действительны в строках JSON, но недопустимы в строках ECMAScript. ES2019 представляет совместимость между строками ECMAScript и JSON.
- Раньше у поставщиков браузеров было множество возможностей указать, что было возвращено из `Function.prototype.toString()`. ES2019 требует, чтобы этот метод возвращал исходный код функции всякий раз, когда это возможно, в противном случае значение `{ [native code] }`.

Б

Строгий режим

Строгий режим (strict mode) был представлен в ECMAScript 5. Он позволяет реализовать более строгую проверку ошибок глобально или локально (в одной функции). Преимущество строго режима в том, что он позволяет раньше получать уведомления об ошибках, и вы можете оперативно устранять их.

Важно понимать правила строгого режима, так как в будущих версиях ECMAScript он будет использоваться по умолчанию. Строгий режим поддерживается во всех основных браузерах.

ВКЛЮЧЕНИЕ СТРОГОГО РЕЖИМА

Включить строгий режим можно с помощью следующей *директивы* (pragma) — строки, которая не назначается никакой переменной:

```
"use strict";
```

Этот синтаксис допустим даже в ECMAScript 3. Если интерпретатор JavaScript не поддерживает строгий режим, эта директива просто игнорируется как строковый литерал, не назначенный никакой переменной.

Если эта директива применяется глобально, то есть вне функции, строгий режим включается для всего сценария. Это означает, что добавление директивы в сценарий, который объединяется с другими сценариями в одном файле, включает строгий режим для всего кода в файле.

Строгий режим можно также включить только внутри отдельной функции, например:

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

    // другие операции
}
```

Если у вас нет полного контроля над всеми сценариями на странице, рекомендуется включать строгий режим только внутри конкретных функций, которые были протестированы в нем.

ПЕРЕМЕННЫЕ

Способ и время создания переменных в строгом и обычном режимах различаются. Так, в строгом режиме нельзя создать глобальную переменную случайно. Например, следующий код в нестрогом режиме создает глобальную переменную:

```
// Переменная не объявлена
// Нестрогий режим: создается глобальная переменная
// Строгий режим: генерируется ошибка ReferenceError

message = "Hello world!";
```

Хотя переменной `message` не предшествует ключевое слово `let` и она не определена как свойство глобального объекта, она все же автоматически создается как глобальная. В строгом режиме присваивание значения необъявленной переменной приводит при запуске кода к ошибке `ReferenceError`.

Кроме того, в строгом режиме нельзя вызвать для переменной оператор `delete`. В нестрогом режиме это возможно, при этом интерпретатор просто возвращает `false`. В строгом режиме попытка удалить переменную приводит к ошибке:

```
// Удаление переменной
// Нестрогий режим: ошибка игнорируется без каких-либо действий
// Строгий режим: генерируется ошибка ReferenceError

let color = "red";
delete color;
```

Строгий режим также налагает ограничения на имена переменных. В нем запрещено использовать переменные с именами `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` и `yield`. Они теперь являются зарезервированными словами, которые, возможно, будут задействованы в будущих редакциях ECMAScript. Попытка использовать их в качестве имен переменных в строгом режиме приведет к синтаксической ошибке.

ОБЪЕКТЫ

В строгом режиме операции с объектами чаще возвращают ошибки, потому что в нестрогом режиме многие ошибки просто игнорируются. Благодаря этому строгий режим способствует раннему устранению ошибок.

Вот некоторые ситуации, когда при доступе к свойству объекта возникает ошибка:

- присваивание значения свойству, доступному только для чтения, приводит к ошибке `TypeError`;

- применение оператора `delete` к неконфигурируемому свойству приводит к ошибке `TypeError`;
- попытка добавить свойство к нерасширяемому объекту приводит к ошибке `TypeError`.

Другое ограничение объектов имеет место, когда вы объявляете их с помощью литералов. Если используется литерал объекта, имена свойств должны быть уникальными, например:

```
// Два свойства с одним именем
// Нестрогий режим: ошибки нет, приоритет отдается второму свойству
// Строгий режим: генерируется синтаксическая ошибка

let person = {
                name: "Nicholas",
                name: "Greg"
            };
```

Этот литерал объекта `person` содержит два свойства с именем `name`. В нестрогом режиме к объекту будет добавлено второе свойство, а в строгом возникнет синтаксическая ошибка.

> **ПРИМЕЧАНИЕ** Ограничение на дубликаты имен свойств было снято в ECMAScript 6. Ключи литеральных свойств дублирующихся объектов не выдают ошибку в строгом режиме.

ФУНКЦИИ

Прежде всего, строгий режим требует, чтобы именованные аргументы функций были уникальными, например:

```
// Повторяющиеся именованные аргументы
// Нестрогий режим: ошибки нет, действителен только второй аргумент
// Строгий режим: генерируется ошибка SyntaxError

function sum (num, num) {
    // какие-то действия
}
```

В нестрогом режиме это объявление функции не приводит к ошибке. Вы можете получить доступ ко второму аргументу `num` по имени, тогда как первый доступен только через объект `arguments`.

В строгом режиме также слегка меняется поведение объекта `arguments`. В нестрогом режиме изменения именованного аргумента отражаются в этом объекте, а в строгом — нет, например:

```
// Изменение значения именованного аргумента
// Нестрогий режим: изменение отражается на объекте arguments
```

```
// Строгий режим: изменение не отражается на объекте arguments

function showValue(value) {
    value = "Foo";
    alert(value);           // "Foo"
    alert(arguments[0]);    // Нестрогий режим: "Foo"
                            // Строгий режим: "Hi"
}

showValue("Hi");
```

Эта функция `showValue()` принимает единственный именованный аргумент `value`. Мы вызываем ее с аргументом `"Hi"`, который назначается переменной `value`, но внутри функции значение `value` изменяется на `"Foo"`. В нестрогом режиме при этом также изменяется значение `arguments[0]`, но в строгом режиме это разные сущности.

В строгом режиме также недоступны свойства `arguments.callee` и `arguments.caller`. В нестрогом режиме они представляют текущую функцию и вызвавшую ее функцию соответственно, а в строгом попытка доступа к любому из этих свойств приводит к ошибке `TypeError`, например:

```
// Попытка доступа к свойству arguments.callee
// Нестрогий режим: код выполняется обычным образом
// Строгий режим: генерируется ошибка TypeError

function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num-1)
    }
}

let result = factorial(5);
```

При попытке прочитать или записать свойство `caller` или `arguments` функции также генерируется ошибка `TypeError`. В приведенном примере эта ошибка возникла бы при доступе к свойствам `factorial.caller` и `factorial.callee`.

Как и в случае переменных, в строгом режиме нельзя назначать функциям имена `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` и `yield`.

Наконец, в строгом режиме можно объявлять функции только на верхнем уровне сценария или функции. Это означает, например, что объявление функции в инструкции `if` является синтаксической ошибкой:

```
// Объявление функции в инструкции if
// Нестрогий режим: функция поднимается за пределы инструкции if
// Строгий режим: генерируется синтаксическая ошибка

if (true) {
    function doSomething() {
        // ...
    }
}
```

Этот синтаксис допустим во всех браузерах в нестрогом режиме, но приводит к синтаксической ошибке в строгом.

Параметры функций

ES6 представил оператор остатка, деструктурированные параметры и параметры по умолчанию — обширный набор новых возможностей для функций для организации, структурирования и определения их параметров. Небольшое изменение, введенное в ECMAScript 7, диктует, что функции, использующие любой из этих расширенных параметров, не могут использовать строгий режим в своем теле без выдачи ошибки. Использование глобального строгого режима все еще разрешено.

```
// допустимо
function foo(a, b, c) {
    "use strict";
}

// недопустимо
function bar(a, b, c='d') {
    "use strict";
}

// недопустимо
function baz({a, b, c}) {
    "use strict";
}

// недопустимо
function qux(a, b, ...c) {
    "use strict";
}
```

Новые функции, представленные в ES6, предполагают, что параметры будут анализироваться в том же режиме, что и тело функции. Поскольку прагма `"use strict"` встречается внутри тела функции, синтаксический анализатор JavaScript должен был бы проверить прагму внутри тела функции, прежде чем принимать параметры, что вносит много путаницы. Поэтому спецификация ES7 ввела это соглашение для того, чтобы синтаксический анализатор мог точно знать, в каком режиме он работает перед анализом функции.

Функция eval()

Печально известная функция `eval()` в строгом режиме стала работать иначе. Основное изменение состоит в том, что она больше не создает переменные или функции в контексте-контейнере, например:

```
// Использование функции eval() для создания переменной
// Нестрогий режим: в оповещении выводится число 10
// Строгий режим: при вызове alert(x) генерируется ошибка ReferenceError
```

```
function doSomething() {
    eval("let x=10");
    alert(x);
}
```

В нестрогом режиме этот код создает локальную переменную `x` в функции `doSomething()`, а затем выводит ее значение в оповещении. В строгом режиме вызов `eval()` не создает переменную `x` внутри функции `doSomething()`, и при вызове `alert()` возникает ошибка `ReferenceError`, потому что переменная `x` не объявлена.

Переменные и функции можно объявлять в функции `eval()`, но они остаются ограниченными специальной областью видимости, которая доступна в течение выполнения кода, а по его завершении уничтожается. Например, следующий код работает без ошибок:

```
"use strict";
let result = eval("let x=10, y=11; x+y");
alert(result);    // 21
```

Здесь мы объявляем переменные `x` и `y` внутри функции `eval()`, а затем вычисляем их сумму. Переменная `result` содержит результат сложения, хотя сами переменные `x` и `y` в момент вызова `alert()` больше не существуют.

ИДЕНТИФИКАТОРЫ EVAL И ARGUMENTS

В строгом режиме теперь явно запрещено использовать имена `eval` и `arguments` в качестве идентификаторов и изменять значения этих переменных:

```
// Переопределение eval и arguments
// Нестрогий режим: все в порядке
// Строгий режим: синтаксическая ошибка

let eval = 10;
let arguments = "Hello world!";
```

В нестрогом режиме вы можете перезаписать значения `eval` и `arguments`, но в строгом это приводит к синтаксической ошибке. Ошибка возникнет во всех перечисленных далее случаях использования этих имен:

- объявление переменной с помощью ключевого слова `let`;
- присваивание переменной другого значения;
- попытка изменить значение переменной, например с помощью оператора `++`;
- использование в качестве имени функции;
- использование в качестве именованного аргумента функции;
- использование в качестве имени исключения в инструкции `try-catch`.

ПРЕОБРАЗОВАНИЕ ЗНАЧЕНИЯ THIS

Одной из серьезнейших проблем с безопасностью и одним из самых непонятных аспектов применения JS-кода является преобразование значения `this` в ряде ситуаций. При вызове метода `apply()` или `call()` для функции значение `null` или `undefined` приводится в нестрогом режиме к глобальному объекту. В строгом режиме значение `this` функции всегда используется так, как указано, например:

```
// Доступ к свойству
// Нестрогий режим: доступ к глобальному свойству
// Строгий режим: ошибка, потому что this имеет значение null

let color = "red";

function displayColor() {
    alert(this.color);
}

displayColor.call(null);
```

Этот код передает в метод `displayColor.call()` значение `null`, так что в нестрогом режиме значением `this` функции является глобальный объект, в результате выводится оповещение со строкой `"red"`. В строгом режиме значение `this` функции равно `null`, и при доступе к свойству объекта `null` генерируется ошибка.

Как правило, функции преобразуют свое значение `this` в тип объекта, что в разговорной речи называется «упаковкой». Примитивы будут принудительно использоваться в своих эквивалентах оболочек объектов.

```
function foo() {
    console.log(this);
}

foo.call();     // Window {}
foo.call(2);    // Number {2}
```

При выполнении в строгом режиме значение `this` больше не будет упаковываться:

```
function foo() {
    "use strict";
    console.log(this);
}

foo.call();     // undefined
foo.call(2);    // 2
```

КЛАССЫ И МОДУЛИ

Классы и модули являются двумя носителями контейнеров кода, введенных в ECMAScript 6. Для любого из них не существует концепции предшественника ECMAScript, и поэтому нет необходимости поддерживать синтаксическую совместимость с унаследованными версиями ECMAScript. Поэтому TC-39 решил, что

весь код, определенный внутри классов и модулей ES6, по умолчанию находится в строгом режиме.

Для классов это включает как объявления классов, так и выражения; конструктор, методы экземпляра, статические методы, методы получения и установки находятся в строгом режиме. Для модулей весь код, определенный внутри них, будет находиться в строгом режиме.

ДРУГИЕ ИЗМЕНЕНИЯ

Следует также упомянуть несколько других отличий строгого режима. Так, в нем недоступна инструкция `with`. Она изменяет способ разрешения идентификаторов и была удалена из строгого режима ради упрощения языка. Попытка использовать `with` в строгом режиме приведет к синтаксической ошибке.

```
// Использование инструкции with
// Нестрогий режим: все в порядке
// Строгий режим: синтаксическая ошибка

with(location) {
    alert(href);
}
```

Также в строгом режиме недоступны *восьмеричные литералы*. Они начинаются с нуля и традиционно были причиной многих ошибок. Теперь использование восьмеричного литерала в строгом режиме считается синтаксической ошибкой.

```
// Использование восьмеричного литерала
// Нестрогий режим: value имеет значение 8
// Строгий режим: синтаксическая ошибка

let value = 010;
```

Как уже говорилось, в ECMAScript 5 метод `parseInt()` был изменен; в нестрогом режиме он обрабатывает восьмеричные литералы так же, как десятичные литералы с начальным нулем, например:

```
// Использование восьмеричного литерала в функции parseInt()
// Нестрогий режим: value имеет значение 8
// Строгий режим: value имеет значение 10

let value = parseInt("010");
```

B

JavaScript-библиотеки и фреймворки

Библиотеки (libraries) в JavaScript помогают компенсировать различия браузеров и упрощают доступ к их сложным функциям. Различают *библиотеки общего назначения* (general libraries) и *специальные библиотеки* (specialty libraries). JavaScript-библиотеки общего назначения предоставляют доступ к наиболее востребованному функционалу браузеров и могут использоваться для реализации базовых возможностей веб-сайтов и веб-приложений. Специальные библиотеки предназначены для решения специфичных задач и используются только в отдельных частях веб-сайтов и веб-приложений. В этом приложении представлен обзор нескольких библиотек и даны ссылки на дополнительные ресурсы.

ФРЕЙМВОРКИ

Обозначение «фреймворк» охватывает спектр различных шаблонов, но в некоторой форме все они обеспечивают продуманную организационную структуру, в рамках которой могут формироваться сложные приложения. Использование фреймворка позволяет приложениям поддерживать согласованные условные обозначения кода при элегантном масштабировании по размеру и сложности. Они предлагают надежные механизмы для общих задач, таких как определение и повторное использование компонентов, управление потоком данных, маршрутизация и многие другие.

Все чаще JavaScript-фреймворки принимают форму *одностраничного приложения* (single page application, SPA). Одностраничные приложения используют API истории браузера HTML5 для предоставления всего пользовательского интерфейса приложения с маршрутизацией URL-адресов и только одной начальной

загрузкой страницы. Фреймворк управляет состоянием приложения, а также всеми компонентами пользовательского интерфейса во время выполнения приложения. У большинства популярных платформ SPA есть сильные сообщества разработчиков, а также множество сторонних расширений.

React

Созданный в Facebook фреймворк *React* охватывает компонент «представление» в модели Model-View-Controller. Его ограниченная сфера применения означает, что его можно использовать вместе с другими платформами или расширениями React для достижения полного охвата MVC. React использует однонаправленный поток данных, является декларативным и основанным на компонентах, использует виртуальную DOM для эффективной перерисовки страницы и предлагает синтаксис JSX, который позволяет писать разметку внутри JavaScript. Facebook также поддерживает дополнительную структуру React, которая называется «Flux».

Лицензия: MIT License.

Веб-сайт: https://reactjs.org/

Angular

Проект *Angular*, впервые выпущенный компанией Google в 2010 году, представляет собой полнофункциональный фреймворк для веб-приложений с архитектурой Model-View-Viewmodel. В 2016 году проект разделился на две ветви: Angular 1.x, который является продолжением первоначального проекта AngularJS, и Angular 2, который является полностью переработанным фреймворком, построенным на конструкциях ES6 и TypeScript. Последние версии обеих выпусков — директивные и основанные на компонентах реализации, и оба проекта используются надежными сообществами разработчиков и сторонними надстройками.

Лицензия: MIT License.

Веб-сайт: https://angularjs.org/ и https://angular.io/

VUE

Vue — это полнофункциональный фреймворк для веб-приложений, менее своеобразный, чем такие фреймворки, как Angular. Со времени выпуска в 2014 году сообщество его разработчиков сильно выросло, и многие разработчики перешли на Vue из-за его производительности и организационных выгод и в то же время менее — из-за строгого соблюдения принципов.

Лицензия: MIT License.

Веб-сайт: https://vuejs.org/

Ember

Ember похож на Angular тем, что также является архитектурой Model-View-Viewmodel и использует предпочтительные соглашения для заполнения веб-приложения. В версии 2.0 в 2015 году было представлено много поведенческих характеристик, используемых в фреймворке React.

Лицензия: MIT License.

Веб-сайт: https://emberjs.com/

Meteor

Meteor совершенно не похож на другие фреймворки в этом списке, потому что это изоморфный фреймворк JavaScript, то есть клиент и сервер имеют общую кодовую базу. Он также использует протокол обновления данных в режиме реального времени, который постоянно передает свежие изменения данных из БД клиенту. Meteor — чрезвычайно самоуверенный фреймворк; однако преимущество этого заключается в том, что функции приложения можно быстро разрабатывать с помощью надежного готового набора инструментов.

Лицензия: MIT License.

Веб-сайт: http://meteor.com/

Backbone.js

Минимальная библиотека с открытым исходным кодом Model-View-Controller (MVC), созданная поверх Underscore.js, *Backbone.js* оптимизирована для одностраничных приложений, что позволяет легко обновлять части страницы по мере изменения состояния приложения.

Лицензия: MIT License.

Веб-сайт: http://backbonejs.org/

БИБЛИОТЕКИ ОБЩЕГО НАЗНАЧЕНИЯ

JavaScript-библиотеки общего назначения предоставляют функционал, охватывающий несколько областей. Все библиотеки общего назначения уравнивают различия в интерфейсе и реализациях браузеров, заключая общий функционал в новые API. Некоторые из этих API напоминают встроенный функционал, тогда как другие выглядят совершенно иначе. Библиотеки общего назначения обычно поддерживают взаимодействие с DOM, технологию Ajax и вспомогательные методы, помогающие решать типичные задачи.

jQuery

jQuery — это библиотека с открытым исходным кодом, которая предоставляет функциональный интерфейс программирования для JavaScript. В основе jQuery лежит использование CSS-селекторов для работы с DOM-элементами. Благодаря цепочке вызовов jQuery-код больше похож не на JS-сценарий, а на описание того, что должно произойти. Такой стиль написания кода особо популярен среди дизайнеров и разработчиков прототипов.

Лицензия: MIT License или General Public License (GPL).

Веб-сайт: http://jquery.com/

Библиотека Google Closure

Библиотека Google Closure — это универсальный инструментарий JavaScript, во многом похожий на jQuery. Он состоит из чрезвычайно широкого набора модулей, охватывающих как низкоуровневые операции, так и высокоуровневые компоненты и виджеты. Библиотека Closure разработана так, что модули могут быть включены по мере необходимости, а библиотека создана для работы вместе с компилятором Google Closure (рассматривается в Приложении Г «JavaScript-инструменты»).

Лицензия: Apache 2.0.

Веб-сайт: https://developers.google.com/closure/library/

Underscore.js

Underscore.js, хотя и не является общей библиотекой в строгом смысле, предоставляет некоторые дополнительные функциональные возможности для функционального программирования на JavaScript. В документации говорится о Underscore.js как о дополнении к jQuery, предоставляющем дополнительные низкоуровневые функциональные возможности для работы с объектами, массивами, функциями и другими типами данных JavaScript.

Лицензия: MIT License.

Веб-сайт: http://documentcloud.github.com/underscore/

Lodash

В той же категории, что и Underscore.js, *Lodash* также является утилитарной библиотекой, которая предоставляет дополнительный инструментарий JavaScript. В нем представлены улучшенные методы для собственных типов, таких как массивы, объекты, функции и примитивы.

Лицензия: MIT License.

Веб-сайт: https://lodash.com/

Prototype

Prototype — это библиотека с открытым исходным кодом, предоставляющая простые API для решения типичных задач веб-программирования. Первоначально разработанная для Ruby on Rails, она содержит определения полезных классов и обеспечивает возможность наследования в JavaScript. Для этого в Prototype реализованы классы, инкапсулирующие часто используемый и сложный функционал в простые API-вызовы. Библиотека Prototype содержится в одном файле, который можно легко подключить к любой странице.

Лицензия: MIT License и Creative Commons Attribution-Share Alike 3.0 Unported.

Веб-сайт: www.prototypejs.org/

Dojo Toolkit

В *Dojo Toolkit*, библиотеке с открытым исходным кодом, группы функциональных возможностей организованы в пакеты, которые можно загружать по требованию. Dojo поддерживает множество параметров и конфигураций, охватывая почти все задачи, которые приходится решать средствами JavaScript.

Лицензия: «новая» BSD License или Academic Free License 2.1.

Веб-сайт: www.dojotoolkit.org/

MooTools

MooTools — это компактная оптимизированная библиотека с открытым исходным кодом, которая добавляет методы к встроенным JavaScript-объектам для расширения их функционала и предоставляет ряд новых объектов. К ее достоинствам можно отнести небольшой размер и простой API.

Лицензия: MIT License.

Веб-сайт: www.mootools.net/

qooxdoo

qooxdoo — это библиотека с открытым исходным кодом, цель которой — помочь во всем цикле разработки веб-приложений. qooxdoo реализует свои собственные версии классов и интерфейсов для создания модели программирования, аналогичной традиционным объектно-ориентированным (OO) языкам. Библиотека включает полный набор инструментов GUI и компиляторы для упрощения процесса сборки интерфейса. qooxdoo начинался как внутренняя библиотека для веб-хостинговой компании 1&1 (www.1and1.com), а затем был выпущен под лицензией с открытым исходным кодом.

Лицензия: GNU Lesser General Public License (LGPL) or Eclipse Public License (EPL).

Веб-сайт: www.qooxdoo.org/

БИБЛИОТЕКИ ДЛЯ АНИМАЦИИ И ЭФФЕКТОВ

Анимация и другие визуальные эффекты стали важными составляющими веб-контента. Реализовать плавные анимации на веб-страницах непросто, но, к счастью, некоторые разработчики уже позаботились об этом, создав специальные библиотеки. Многие из упомянутых ранее JavaScript-библиотек общего назначения также поддерживают анимации.

D3

Самая популярная библиотека анимации, *D3* (для «документов, управляемых данными»), сегодня является наиболее надежным и мощным инструментом визуализации данных JavaScript. Он имеет очень обширный набор функций, охватывающий холст, SVG, CSS и HTML5 визуализации. Библиотека дает возможность контролировать окончательную отрисовку с предельной точностью.

Лицензия: BSD.

Веб-сайт: http://www.d3js.org/

three.js

three.js — одна из самых популярных доступных библиотек WebGL. Она предлагает простой API, который позволяет выполнять сложные 3D-отрисовки и анимации.

Лицензия: MIT License.

Веб-сайт: https://threejs.org/

moo.fx

Библиотека *moo.fx* с открытым исходным кодом работает поверх Prototype или MooTools. Она очень компактна (последняя версия занимает всего 3 Кбайт) и предназначена для создания анимаций при минимуме кодирования. По умолчанию moo.fx входит в MooTools, но ее можно также загрузить отдельно для использования с Prototype.

Лицензия: MIT License.

Веб-сайт: http://moofx.mad4milk.net/

Lightbox

Библиотека *Lightbox* предназначена для создания простых графических наложений на страницах. Для работы ей требуются библиотеки Prototype и script.aculo.us. Идея в том, чтобы пользователи могли просматривать изображения или последовательности изображений, не покидая текущую страницу. Для Lightbox-наложений можно настраивать вид и переходы.

Лицензия: Creative Commons Attribution 2.5 License.

Веб-сайт: www.huddletogether.com/projects/lightbox2/

Г

JavaScript-инструменты

Написание JS-кода во многом напоминает программирование на любом другом языке, при этом, как и в других языках, вы можете использовать вспомогательные инструменты. Количество таких инструментов для JavaScript постоянно растет, благодаря чему искать проблемы, оптимизировать и развертывать приложения становится проще. Некоторые из этих инструментов запускаются из JS-кода, другие можно запускать извне браузера. В этом приложении представлен обзор нескольких полезных инструментов и ссылки на ресурсы с дополнительными сведениями.

ПРИМЕЧАНИЕ Вы заметите, что некоторые инструменты включены в несколько разделов этого приложения. Многие инструменты JavaScript, доступные сегодня, должны быть инструментами управления проектами по принципу «все в одном» и, таким образом, они относятся к нескольким сферам.

СРЕДСТВА УПРАВЛЕНИЯ ПАКЕТАМИ

Проекты JavaScript обычно должны использовать сторонние библиотеки и ресурсы, чтобы избежать дублирования кода и ускорения разработки. Сторонние библиотеки, называемые «пакетами», размещаются в общедоступных репозиториях. Пакеты могут принимать форму ресурсов, которые будут доставляться в браузер, библиотек JavaScript, которые будут скомпилированы в рамках проекта, или даже инструментов для конвейера разработки проекта. Эти пакеты почти всегда активно разрабатываются и пересматриваются, в дополнение к различным версиям выпусков. *Средства управления пакетами* в JavaScript позволяют управлять тем, от каких пакетов зависят проекты, как обращаться к ним и устанавливать их, а также какие версии устанавливать.

Средства управления пакетами предлагают интерфейс командной строки для установки или удаления зависимостей проекта. Конфигурация проекта обычно хранится в локальном файле манифеста проекта.

npm

npm, который расшифровывается как «диспетчер пакетов Node (Node Package Manager)», является менеджером пакетов по умолчанию для среды выполнения NodeJS. Сторонние пакеты, предоставляемые в реестре npm, могут быть указаны как зависимости проекта и установлены локально через командную строку. Репозиторий npm содержит как серверные, так и клиентские пакеты JavaScript.

npm предназначен для использования на сервере, где размер графа зависимостей менее критичен. При установке пакетов npm использует вложенное дерево зависимостей для разрешения всех зависимостей проекта; каждая зависимость проекта будет устанавливать свою собственную версию любых других пакетов, от которых она зависит. Это означает, что если ваш проект имеет три пакетных зависимости — A, B и C — и каждая из них зависит от отдельной версии пакета D, npm установит три отдельные версии пакета D, по одной для каждой зависимости.

Веб-сайт: https://www.npmjs.com/

Bower

Bower поддерживает многие из тех же ритмов, что и npm, включая CLI для установки и управления пакетами, но основное внимание уделяется управлению пакетами, которые будут использоваться для клиента. Одно из основных различий между Bower и npm состоит в том, что Bower использует плоскую структуру зависимостей. Это означает, что зависимости проекта будут совместно использовать пакеты, от которых они зависят, и задача пользователя состоит в том, чтобы разрешить эти зависимости. Например, если ваш проект имеет три зависимости пакета — A, B и C — и каждая из них зависит от другой версии пакета D, нужно будет найти одну версию пакета D, которая удовлетворяет требованиям всех этих пакетов, потому что плоская структура зависимостей диктует, что вы устанавливаете одну любую версию данного пакета.

Веб-сайт: https://bower.io/

JSPM

JSPM — это средство управления пакетами, построенное на использовании SystemJS для динамической загрузки модулей. Сам он похож на npm, но не зависит от реестра; пакеты из реестра npm, Github или даже пользовательских реестров могут быть установлены вместе с помощью его CLI. Вместо того чтобы связывать и предварительно компилировать ресурсы на сервере, JSPM позволяет динамически предоставлять пакеты клиенту по мере необходимости через SystemJS. Как и Bower, он использует плоскую структуру зависимостей.

Веб-сайт: https://jspm.io/

Yarn

Разработанное в Facebook, *Yarn* — это специальное средство управления пакетами, которое во многом является усовершенствованной версией npm. Он может получить доступ ко всем тем же пакетам npm через реестр Yarn, и он устанавливает их так же, как и npm. Основное различие между Yarn и npm заключается в том, что он предлагает функции, позволяющие ускорить установку, кэширование пакетов, файлы блокировки и улучшенные функции безопасности пакетов.

Веб-сайт: https://yarnpkg.com/

СРЕДСТВА ЗАГРУЗКИ МОДУЛЕЙ

Средства загрузки модулей позволяют запрашивать модули с сервера по требованию, а не загружать все модули JS или один связанный файл JS одновременно. Спецификация модуля ECMAScript 6 описывает возможную цель для браузеров, чтобы изначально поддерживать динамическую загрузку модулей. В настоящее время многие браузеры по-прежнему не поддерживают загрузку модулей ES6, и поэтому средства загрузки модулей служат своего рода заменой, которая позволяет пакетам динамически загружать модули из клиента.

SystemJS

Средство загрузки модулей *SystemJS* предназначено для использования на сервере или клиенте. SystemJS поддерживает все форматы модулей, включая AMD, CommonJS, UMD и ES6. Он также поддерживает транспиляцию в браузере (не рекомендуется для крупных проектов из-за проблем с производительностью).

Веб-сайт: https://github.com/systemjs/

RequireJS

RequireJS построен на основе спецификации модулей AMD и предлагает исключительную поддержку устаревших браузеров. Хотя RequireJS проверен временем, сообщество JavaScript в целом в значительной степени отбрасывает формат модулей AMD, поэтому начинать крупный проект с RequireJS не рекомендуется.

Веб-сайт: http://requirejs.org/

СБОРЩИКИ МОДУЛЕЙ

Сборщики модулей позволяют объединять произвольное количество модулей в различных форматах в пакеты, которые можно загрузить на клиент. Сборщик будет отслеживать граф зависимостей приложения и упорядочивать модули по мере необходимости. Часто приложение может обслуживаться как один пакет, но также возможны настройки для нескольких пакетов. Сборщик модулей также

часто поддерживает объединение необработанных или скомпилированных ресурсов CSS. Сборщики могут принимать форму самореализующейся связки или могут оставаться в виде объединенных активов модуля, которые не будут выполняться до тех пор, пока это не потребуется.

Webpack

Обладая широким набором функций и превосходной расширяемостью, *Webpack* в подавляющем большинстве является самым популярным на сегодняшний день сборщиком приложений. Он может объединять различные типы модулей, поддерживает широкий спектр плагинов и полностью совместим с большинством библиотек шаблонов и транспиляции.

Веб-сайт: https://webpack.js.org/

JSPM

JSPM — это средство управления пакетами, созданное на основе SystemJS и загрузчика модулей ES6. Один из рекомендованных рабочих процессов JSPM состоит в объединении всех модулей в один файл, который затем будет загружен с помощью SystemJS. Эта возможность доступна через JSPM CLI.

Веб-сайт: https://jspm.io/

Browserify

Browserify — это несколько более старый, но широко используемый и проверенный в бою пакет модулей, который поддерживает синтаксис зависимостей `require()` CommonJS в стиле NodeJS.

Веб-сайт: http://browserify.org/

Rollup

Rollup во многих отношениях похож на Browserify по своим возможностям связывания модулей, но также предоставляет встроеную возможность встряхивания дерева. Rollup может анализировать граф зависимостей приложения и удалять любые модули, которые фактически не используются.

Веб-сайт: https://rollupjs.org/

СРЕДСТВА КОМПИЛЯЦИИ/ТРАНСПИЛЯЦИИ И СИСТЕМЫ СТАТИЧЕСКОГО ТИПА

Код веб-приложения, написанный в редакторе, почти никогда не является точной копией кода, который будет передан клиенту. Разработчики часто хотят использовать новые функции спецификации ECMAScript, которые еще не получили

универсального одобрения браузером. Кроме того, разработчики также часто хотят расширить или усовершенствовать свою кодовую базу с помощью статической системы типов или функций, выходящих за пределы спецификации ECMA. Существует целый ряд инструментов для решения различных аспектов этих потребностей.

Babel

Babel — один из самых популярных инструментов, используемых для компиляции новейших функций спецификации ECMAScript до версии ECMA, удобной для браузера. Он также поддерживает JSX от React и широкий спектр плагинов и совместим со всеми основными инструментами сборки.

Веб-сайт: https://babeljs.io/

Google Closure Compiler

Google Closure Compiler — это мощный JS-компилятор, способный варьировать уровни оптимизации компиляции, а также надежная система проверки статического типа. Аннотации типа выполняются в комментариях в стиле JSDoc.

Веб-сайт: https://developers.google.com/closure/

CoffeeScript

CoffeeScript — это расширенный синтаксис ECMAScript, который компилируется в обычный JavaScript; почти все в CoffeeScript является выражением. Его улучшения синтаксиса вдохновлены Ruby, Python и Haskell.

Сайт: http://coffeescript.org/

TypeScript

TypeScript от Microsoft — это типизированный расширенный набор JavaScript, который включает в себя надежную проверку статических типов и основные улучшения синтаксиса. Поскольку это строгий расширенный набор JavaScript, обычные JS-программы имеют допустимый синтаксис TypeScript. TypeScript также может использовать файлы определения типа для указания информации о типе для существующих библиотек JavaScript.

Веб-сайт: http://typescriptlang.org/

Flow

Flow от Facebook — это система аннотаций простого типа для JavaScript. Его синтаксис типа очень похож на TypeScript, но он не добавляет дополнительных возможностей языка, кроме объявлений типов.

Веб-сайт: https://flow.org/

ВЫСОКОЭФФЕКТИВНЫЕ ИНСТРУМЕНТЫ СЦЕНАРИЕВ

Распространенная критика JavaScript заключается в том, что он невероятно медленный и неуместен для вычислений, требующих гибкости. Независимо от того, сколько опасности таится в обозначении «медленный», это не меняет того факта, что язык никогда не создавался для поддержки гибких вычислений. Для решения этой проблемы существует ряд проектов, которые пытаются расширить парадигмы выполнения кода в браузере, чтобы позволить программе работать с почти родной скоростью и использовать аппаратную оптимизацию.

WebAssembly

Проект *WebAssembly* (или *wasm*) работает над реализацией языка, который может выполняться во многих местах (переносимый) и который существует как двоичный язык, компилируемый из нескольких языков низкого уровня, таких как C++ и Rust. Код WebAssembly работает на совершенно отдельной виртуальной машине и на JavaScript в браузере, и его способность взаимодействовать с различными API браузера крайне ограничена. С JavaScript и DOM можно взаимодействовать косвенным и ограниченным образом, но главной целью WebAssembly является создание чрезвычайно быстрого языка, который может работать в веб-браузерах (и в других местах) и предлагать практически собственную производительность и аппаратное ускорение. Спецификация WebAssembly все еще дорабатывается и уточняется, но это одна из самых многообещающих областей в технологии браузеров.

Веб-сайт: http://webassembly.org/

asm.js

asm.js основан на идее, что скомпилированный JavaScript может выполняться намного быстрее, чем рукописный JavaScript. asm.js — это подмножество JavaScript, с которым может быть скомпилирован и выполнен код языка низкого уровня в обычном браузере или движке Node. Современные механизмы JavaScript выводят типы во время выполнения, а код asm.js делает эти выводы типов (и, следовательно, связанные с ними операции) гораздо менее затратными в вычислительном отношении, выполняя принудительное приведение типов с помощью лексических подсказок. Он также широко использует TypedArrays, которые предлагают огромный скачок производительности по сравнению с традиционным ассоциативным массивом JS. asm.js не так быстр, как WebAssembly, но все же обеспечивает значительный прирост производительности за счет компиляции.

Веб-сайт: http://asmjs.org/

Emscripten и LLVM

Хотя он никогда не выполняется в браузере, *Emscripten* является важным набором инструментов для компиляции низкоуровневого кода в WebAssembly и asm.js.

Emscripten использует компилятор *LLVM* для компиляции таких языков, как C, C ++ и Rust в код, который может выполняться либо непосредственно в браузере (asm.js), либо на виртуальной машине (WebAssembly).

Веб-сайт Emscripten: http://kripken.github.io/emscripten-site/

Веб-сайт LLVM: http://llvm.org/

РЕДАКТОРЫ

VIM, Emacs и им подобные являются отличными текстовыми редакторами, но по мере того, как ваша среда сборки и приложения масштабируются по сложности, становится все более полезным автоматизация типичных задач в редакторах, таких как автозаполнение, форматирование файлов, кодирование и организация каталогов проекта. Доступен широкий выбор *редакторов* и IDE, предлагающих эти функции, как бесплатных, так и платных.

Sublime Text

Sublime Text — популярный текстовый редактор с закрытым исходным кодом. Его можно использовать для разработки на всех языках, но он предлагает чрезвычайно обширную библиотеку дополнений, поддерживаемую сообществом. Производительность редактора является одной из лучших в своем классе.

Веб-сайт: https://sublimetext.com/

Стоимость: бесплатная пробная версия; одноразовая лицензия за 80 долларов.

Atom

Atom от Github — это текстовый редактор с открытым исходным кодом, обладающий многими теми же функциями, что и Sublime Text, включая процветающее сообщество и сторонние дополнения к пакетам. Редактор иногда имеет проблемы с производительностью, но он постоянно делает успехи в этой области.

Сайт: https://atom.io/

Стоимость: бесплатно.

Brackets

Brackets от Adobe похож на Atom в том, что они оба распространяются с открытым исходным кодом, но редактор разработан специально для веб-разработчиков и предлагает ряд очень впечатляющих и уникальных функций, предназначенных для внешнего интерфейса. Сопровождает все это обширная библиотека дополнений.

Сайт: http://brackets.io/

Стоимость: бесплатно.

Visual Studio Code

Visual Studio Code от Microsoft — это редактор с открытым исходным кодом, основанный на платформе Electron. Подобно другим основным редакторам, он легко расширяется благодаря своей библиотеке пакетов.

Веб-сайт: https://jetbrains.com/webstorm/

Стоимость: бесплатно.

WebStorm

WebStorm от JetBrains — это неприступно высокооктановая среда разработки, предназначенная для использования в качестве окончательного инструментария для разработки проектов, с надежной интеграцией с ведущими средами внешнего интерфейса. Он также интегрируется с большинством инструментов сборки и систем контроля версий.

Веб-сайт: https://jetbrains.com/webstorm/

Стоимость: бесплатная пробная версия; ежемесячные/ежегодные лицензионные сборы.

СИСТЕМЫ СБОРКИ И АВТОМАТИЗАЦИИ, СРЕДСТВА ЗАПУСКА ЗАДАЧ

Процесс преобразования локального каталога проекта в приложение, обслуживаемое в конечной версии продукта, обычно объединяется в ряд задач. Каждая из этих задач часто состоит из конвейера из множества подзадач — например, создание и развертывание приложения будет включать в себя объединение модулей, компиляцию, минимизацию и передачу статических ресурсов, среди многих других задач. Выполнение модульного или интеграционного тестирования может включать в себя инициализацию тестовых наборов и ускорение работы браузеров. Чтобы упростить управление этими задачами и их использование, существует ряд инструментов, позволяющих более эффективно составлять и организовывать задачи приложения.

Grunt

Grunt — это средство запуска задач NodeJS, которое использует объекты конфигурации для декларативного определения того, как задачи должны выполняться. Он имеет обширное сообщество и большой выбор дополнений, с помощью которых можно организовать работу над задачами проекта.

Веб-сайт: https://gruntjs.com/

Gulp

Подобно Grunt, *Gulp* также выполняет задачи NodeJS, но вместо этого использует конвейерный подход в стиле Unix для определения задач, где отдельные задачи определяются как функции JavaScript. Gulp также имеет динамичное сообщество и библиотеку пакетов.

Веб-сайт: https://gulpjs.com/

Brunch

Brunch — это еще один инструмент сборки NodeJS, который разработан для упрощения и удобства использования, а не для максимальной конфигурируемости. Он новее, чем Gulp или Grunt, но уже имеет огромный выбор дополнений.

Сайт: http://brunch.io/

npm

Несмотря на то что *npm* не является инструментом сборки, он предлагает функцию сценариев, которая, по мнению многих разработчиков, прекрасно работает как средство запуска задач без дополнительных излишеств. Сценарии определяются непосредственно внутри пакета NodeJS.

Веб-сайт: https://docs.npmjs.com/misc/scripts/

СРЕДСТВА АНАЛИЗА И ФОРМАТИРОВАНИЯ КОДА

Отчасти проблема с отладкой JavaScript заключается в том, что многие IDE не указывают автоматически на синтаксические ошибки при вводе. Большинство разработчиков пишут какой-либо код, а затем загружают его в браузер для поиска ошибок. Можно значительно уменьшить количество таких ошибок, проверяя код JavaScript перед развертыванием. *Средства анализа* проверяют основной синтаксис и выдают предупреждения о стиле.

Средства форматирования — это инструменты, которые неявно понимают синтаксические правила языка и используют набор отступов, пробелов, переносов строк и других стратегий, чтобы предоставить возможность автоматически аккуратно организовать содержимое файла. Средства форматирования никогда не нарушат или не изменят код или его семантическое значение, поскольку они осведомлены о видах изменений, которые могут повлиять на выполнение.

ESLint

ESLint — это JavaScript-утилита с открытым исходным кодом, изначально разработанная не кем иным, как автором предыдущих изданий этой книги Николасом Закасом. Он спроектирован так, чтобы быть полностью «подключаемым»: он поставляется из коробки с общим набором правил по умолчанию, но правила полностью

настраиваемы, и имеется большая библиотека изменяемых и переключаемых правил, которые можно использовать для настройки поведения анализатора.

Веб-сайт: www.eslint.org/

Google Closure Compiler

В *Closure Compiler* встроена служебная программа, которая может быть активирована с помощью флагов командной строки. Этот анализатор работает с абстрактным синтаксическим деревом кода, поэтому он не выполняет проверки пробелов, отступов или других вопросов организации кода, которые не влияют на выполнение.

Веб-сайт: https://developers.google.com/closure/

JSLint

JSLint — это средство проверки JavaScript, написанное Дугласом Крокфордом. Он проверяет синтаксические ошибки на уровне ядра, используя наименьший общий знаменатель для кросс-браузерных проблем. (Следуйте самым строгим правилам, чтобы ваш код работал везде.) Вы можете включить предупреждения Крокфорда о стиле написания кода, включая формат, использование необъявленных глобальных переменных и многое другое. Хотя JSLint написан на JavaScript, его можно запустить из командной строки через интерпретатор Rhino на основе Java, а также через WScript и другие интерпретаторы JavaScript. Веб-сайт предоставляет собственные версии для каждого интерпретатора командной строки.

Веб-сайт: www.jslint.com/

JSHint

JSHint — это ответвление JSLint, который предоставляет больше возможностей для настройки применяемых правил. Подобно JSLint, он сначала проверяет синтаксические ошибки, а затем ищет проблемные паттерны кодирования. Каждая проверка JSLint также присутствует в JSHint, но разработчики лучше контролируют, какие правила нужно применять. Также как и JSLint, JSHint можно запустить из командной строки с помощью Rhino.

Веб-сайт: www.jshint.com/

Clang Format

ClangFormat — это набор инструментов форматирования, созданный на основе библиотеки LibFormat на основе проекта Clang. Он использует правила форматирования Clang для автоматической пространственной реорганизации кода (без фактического изменения его семантической структуры). Он работает как самостоятельный инструмент, который можно использовать из командной строки, но также имеет несколько интеграций с редакторами.

Веб-сайт: https://clang.llvm.org/docs/ClangFormat.html

СРЕДСТВА УМЕНЬШЕНИЯ КОЛИЧЕСТВА КОДА

Важной частью процесса сборки JavaScript является обработка вывода для удаления лишних символов. Это гарантирует, что только минимальное количество байтов передается в браузер для анализа и, в конечном итоге, ускоряет работу пользователя. Существует несколько таких *средств уменьшения количества кода* (*минимизаторов*) с различными степенями сжатия.

Uglify

Uglify, в настоящее время его третий выпуск, представляет собой набор инструментов, который позволяет минимизировать, украсить и сжать JS-код. Его можно запустить из командной строки, и он предлагает чрезвычайно широкий диапазон параметров сжатия, которые можно настроить для настройки минимизации.

Веб-сайт: https://github.com/mishoo/UglifyJS2/

Google Closure Compiler

Несмотря на то что *Closure* не является минимизатором в строгом смысле, он предлагает несколько уровней оптимизации, которые в процессе проведения процедур оптимизации в том числе минимизируют код.

Веб-сайт: https://developers.google.com/closure/

JSMin

JSMin — это основанный на C инструмент, написанный Дугласом Крокфордом, который выполняет базовое сжатие JavaScript. В первую очередь он удаляет пробелы и комментарии, чтобы гарантировать, что полученный код все еще может быть выполнен без проблем. JSMin доступен как исполняемый файл Windows с исходным кодом на C и многих других языках.

Веб-сайт: www.crockford.com/javascript/jsmin.html

Dojo ShrinkSafe

Те же люди, ответственные за Dojo Toolkit, имеют инструмент под названием *ShrinkSafe*, который использует интерпретатор Rhino в JavaScript, чтобы сначала анализировать код JavaScript в потоке токенов, а затем использовать его для безопасного кодирования. Как и в случае с JSMin, ShrinkSafe удаляет лишние пробелы (но не разрывы строк) и комментарии, но также делает еще один шаг вперед и заменяет имена локальных переменных двухсимвольными именами. Результатом является улучшенная производительность по сравнению с JSMin, без риска появления синтаксических ошибок.

Веб-сайт: http://shrinksafe.dojotoolkit.org/

СРЕДСТВА МОДУЛЬНОГО ТЕСТИРОВАНИЯ

Большинство библиотек JavaScript используют некоторую форму модульного тестирования для своего собственного кода, а некоторые публикуют среду модульного тестирования для использования другими. Разработка через тестирование (Test-driven development , TDD) — это процесс разработки программного обеспечения, основанный на использовании модульного тестирования.

Mocha

Mocha предлагает отличную конфигурируемость и расширяемость при разработке модульных тестов. Тесты достаточно гибкие, а их последовательное выполнение означает точную отчетность и более простую отладку.

Веб-сайт: https://mochajs.org/

Jasmine

Несмотря на то что он находится на более старой стороне спектра модульных тестов, *Jasmine* по-прежнему чрезвычайно популярен. Он поставляется со всем необходимым, без каких-либо внешних зависимостей, а его синтаксис прост и удобен для чтения.

Веб-сайт: https://jasmine.github.io/

qUnit

qUnit — это среда модульного тестирования, разработанная для использования с jQuery. Действительно, сам jQuery использует qUnit для всего своего тестирования. Несмотря на это, qUnit не зависит от jQuery и может использоваться для тестирования любого JS-кода. qUnit известен как очень простой фреймворк для модульного тестирования, позволяющий легко приступить к работе.

Веб-сайт: https://github.com/jquery/qunit

JsUnit

Исходная библиотека модульного тестирования JavaScript не привязана к какой-либо конкретной библиотеке. *JsUnit* — это порт популярной среды тестирования JUnit для Java. Тесты выполняются на странице и могут быть настроены для автоматического тестирования и отправки результатов на сервер. Сайт содержит примеры и основную документацию.

Веб-сайт: www.jsunit.net/

Dojo Object Harness

Dojo Object Harness (DOH) начинался как инструмент внутреннего юнит-тестирования для Dojo, а затем был выпущен для всеобщего использования. Как и в других средах, модульные тесты запускаются внутри браузера.

Веб-сайт: www.dojotoolkit.org/

СРЕДСТВА ГЕНЕРАЦИИ ДОКУМЕНТАЦИИ

Большинство IDE добавляют *генераторы документации* для основного языка. Поскольку в JavaScript нет официальной IDE, документация традиционно выполняется вручную или с помощью перепрофилирования генераторов документации для других языков. Однако существует ряд генераторов документации, специально предназначенных для JavaScript.

ESDoc

ESDoc способен генерировать очень продвинутые страницы документации для кода, включая возможность размещать ссылки на исходный код со страницы самой документации. Он также имеет библиотеку дополнений для расширения своих возможностей. Однако ESDoc требует, чтобы кодовая база состояла исключительно из модулей ES6.

Веб-сайт: https://esdoc.org/

documentation.js

documentation.js обрабатывает комментарии JSDoc внутри кода для автоматической генерации документации в HTML, Markdown или JSON. Он совместим с последними версиями ECMAScript и всеми основными инструментами сборки, а также может работать с аннотациями Flow.

Веб-сайт: https://documentation.js.org/

Docco

По данным веб-сайта, *Docco* является «быстрым и грязным» генератором документации. Мотивация этого инструмента заключается в том, что это очень простой способ создания HTML-страниц, которые описывают кодовую базу. Docco хрупок в некоторых отношениях, но он предлагает наименьшее количество барьеров для создания документации напрямую из кода.

Веб-сайт: http://ashkenas.com/docco/

JsDoc Toolkit

JsDoc Toolkit был одним из первых генераторов документации JavaScript. Требуется ввести Javadoc-подобные комментарии в исходный код, которые затем будут обработаны и выведены в виде HTML-файлов. Можно настроить формат HTML с помощью одного из готовых шаблонов JsDoc или создать свой собственный. Набор инструментов JsDoc доступен в виде пакета Java.

Веб-сайт: https://github.com/jsdoc3/jsdoc/

YUI Doc

YUI Doc — генератор документации YUI. Генератор написан на Python, поэтому ему нужна среда исполнения Python для установки. YUI Doc выводит файлы HTML со встроенными поисками свойств и методов, реализованными с использованием виджета автозаполнения YUI. Как в случае с JsDoc, YUI Doc требует, чтобы Javadoc-подобные комментарии были вставлены в исходный код. HTML по умолчанию можно изменить, изменив файл шаблона HTML по умолчанию и связанную таблицу стилей.

Веб-сайт: www.yuilibrary.com/projects/yuidoc/

AjaxDoc

Цель *AjaxDoc* немного отличается от целей предыдущих генераторов. Вместо создания HTML-файлов для документации JavaScript, он создает XML-файлы в том же формате, что и файлы, созданные для языков .NET, таких как C # и Visual Basic .NET. Это позволяет стандартным генераторам документации .NET создавать документацию в виде файлов HTML. AjaxDoc требует формат комментариев к документации, аналогичный комментариям к документации для всех языков .NET. AjaxDoc был создан для использования с решениями ASP.NET Ajax, но его можно использовать и в отдельных проектах.

Веб-сайт: www.codeplex.com/ajaxdoc/