Прототипы

Курс: JavaScript, часть 2: прототипы и асинхронность

20 февраля 2018 г.

Оглавление

Глава 1

Прототипы

1.1. Прототипы

Сформулируем задачу: пусть у нас есть некоторый объект, который олицетворяет студента. Он записан в литеральной нотации и описывает характеристики студента, а также полезные для него действия. Этот объект содержит поле name, которое хранит имя студента и метод `getName`, который возвращает имя нашего студента.

```
var student = {
    name: 'Billy',
    type: 'human',
    getName: function () {
        return this.name;
    },
    sleep: function () {
        console.info('zzzZZZ...');
    }
};

student.getName();
// Billy
```

Объект студента сложно рассматривать в отрыве от объекта преподавателя. У преподавателя также есть ряд полей и методов, например поле `name`, которое хранит имя. Если мы посмотрим на два этих объекта внимательнее, мы уви-

дим, что в них очень много похожего: у каждого из них есть метод `getName`, который выполняет одинаковую работу. Таким образом, мы дублируем реализацию одного и того же метода в двух разных объектах, и это проблема.
К счастью, решение очень простое. Мы можем выделить общие части в отдельную конструкцию.
Назовем объединяющий объект person/личность. В итоге мы получим три несвязанных объекта: студента, преподавателя и личность.

```
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

Так как мы забрали у наших объектов студента и преподавателя полезный метод `getName`, нам необходимо после нашего рефакторинга решить следующую задачу: научить студента пользоваться общим кодом, который мы вынесли в другой объект. Для решения этой задачи мы можем воспользоваться методом заимствования. Для этого мы можем позаимствовать метод `getName` у объекта `person` и вызвать его при помощи метода `call`, передав первым аргументом объект студента.

```
var student = {
    name: 'Billy',
};
var person = {
    getName: function () {
        return this.name;
    }
};
person.getName.call(student);
```

Нам хотелось бы вызывать метод `getName`, как и раньше, от лица студента. Можем ли мы связать два наших объекта студента и `person` таким образом, чтобы это было возможным?
Необходимо лишь в специальное внутреннее поле `[[Prototype]]` одного объекта записать ссылку на другой. Так, мы можем записать в это поле у объекта

`student` ссылку на объект `person` и получить желаемое поведение. Обратиться напрямую ко внутреннему полю, конечно, нельзя, но существует ряд способов, которые позволяют записать в него новое значение. Один из них — геттер и сеттер `_proto_`.

```
var student = {
    name: 'Billy',
    sleep: function () {},
    [[Prototype]]: <link to person>,
};
student['[[Prototype]]'] = person; //так не работает!
```

> Объект, на который указывает ссылка во внутреннем поле `[[Prototype]]`, называется **прототипом**.

1.2. Цепочки прототипов

Что происходит, когда мы пытаемся вызвать метод, которого нет у объекта, но он есть в прототипе? В этом случае интерпретатор переходит по ссылке, которая хранится во внутреннем поле `Prototype` и пробует найти этот метод в прототипе. В нашем случае мы вызываем метод `getName` у объекта student, но этого метода у этого объекта нет. Интерпретатор смотрит значения внутреннего поля `Prototype`, видит там ссылку на прототип — `person`, и переходит по этой ссылке, пробуя найти этот метод уже в прототипе. Там он этот метод находит и вызывает. Важно заметить, что `this` при исполнении этого метода будет ссылаться на объект `student`, так как мы этот метод вызываем от лица студента.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

Но что произойдет, если мы попытаемся вызвать метод, которого нет не только у объекта, но и в прототипе? Можно заметить, что у прототипа также есть внутреннее поле `Prototype`.

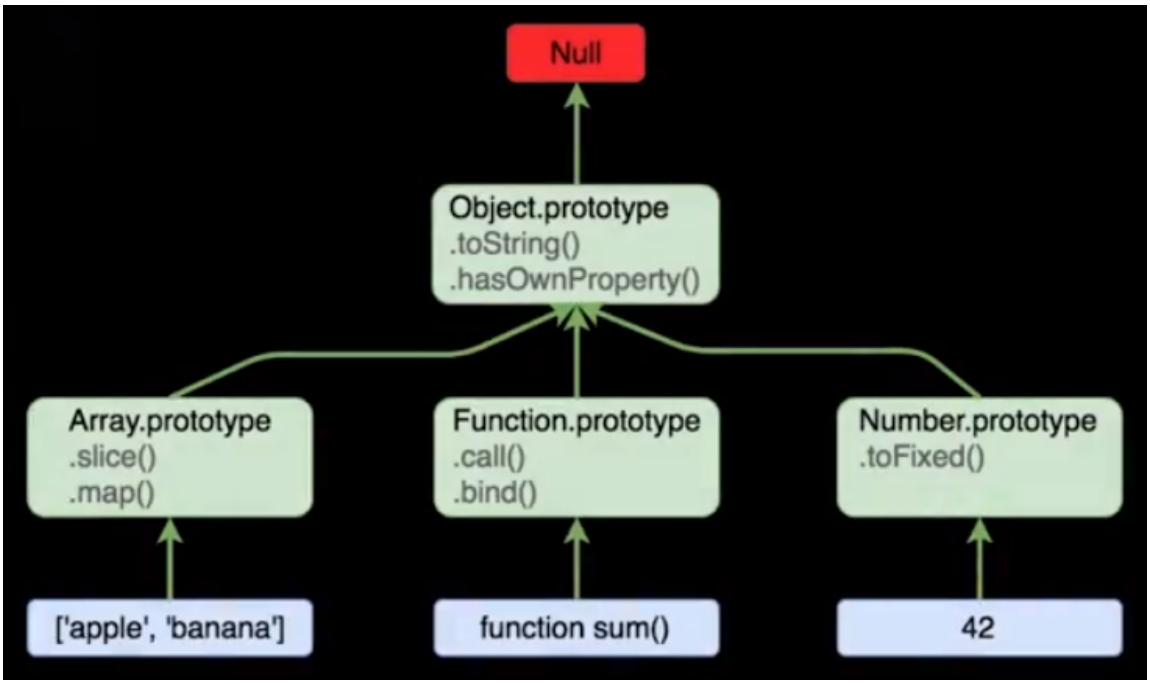
> Интерпретатор идет по выстроенной цепочке прототипов в поисках поля или метода до тех пор, пока не встретит значение `null` в специальном внутреннем поле `Prototype`. Если он прошел весь путь по цепочке, но так и не нашел искомого метода или поля, в этом случае он вернет `undefined`.

В нашем случае может показаться, что поиск остановится уже на объекте `person`, ведь мы специально не записывали никакую ссылку во внутреннее поле `Prototype`. Но любой объект уже имеет в качестве прототипа некоторый глобальный прототип — глобальный прототип для всех объектов. Он расположен в специальном поле `Prototype` функции `Object` и хранит в себе методы, полезные для всех объектов.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    [[Prototype]]: <Object.prototype>
};
```

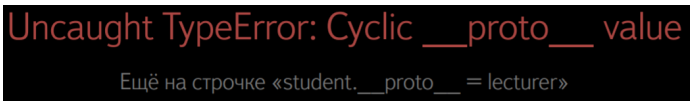
Мы выяснили, что в нашем случае поиск метода не остановится на объекте `person` и мы проследуем по цепочке прототипов дальше в глобальный прототип

для всех объектов. И уже там наш поиск остановится, так как внутреннее поле `Prototype` глобального прототипа для всех объектов имеет значение `null`. Помимо общего, глобального прототипа для всех объектов, существуют в языке более частные глобальные прототипы: для массивов, функций, и т.д. Каждый из этих прототипов по умолчанию в качестве прототипа имеет глобальный прототип для всех объектов. И для того, чтобы поиск метода или поля по цепочке прототипа всегда заканчивался, в этом глобальном прототипе в поле `Prototype`, во внутреннем поле, хранится значение `null`.



Давайте попробуем обмануть интерпретатор и в качестве прототипа для преподавателя выбрать студента, а в качестве прототипа для студента выбрать преподавателя. И попробуем вызвать заведомо несуществующий метод или поле. В данном случае может показаться, что поиск будет идти бесконечно, интерпретатор будет вечно ходить по созданному нами циклу в цепочке прототипов. Но интерпретатор позаботился о таком поведении и выбросит ошибку уже на этапе попытки создания такой цепочки.

```
1 var lecturer = { name:
  ↪  'Sergey'}
2 var student = { name:
  ↪  'Billy' }
3
4 lecturer.__proto__ =
  ↪  student;
5 student.__proto__ =
  ↪  lecturer;
6
7 console.info(lecturer.abrakadabra);
```



1.3. Способы установки прототипов

Есть три способа установки прототипа.

1.3.1. `__proto__`

Первый — это сеттер, геттер `__proto__`. Не идеальный метод:

- не является частью ECMAScript 5;
- он долгое время не являлся частью спецификации языка, и более того, поддерживался далеко не всеми платформами;
- появился он благодаря разработчикам браузеров, которые потихонечку внедряли эту возможность в свои продукты.

1.3.2. Метод `create`

Следующий способ установки прототипов — использование специального метода `create`, который в качестве параметра принимает в себя объект, который мы хотим видеть в качестве прототипа для нового объекта, который этот метод возвращает.

```
1 var student = Object.create(person)
```

Особенности способа:

- уже является частью ECMAScript 5;
- делает больше работы, чем простое присваивание ссылки;
- создаёт новые объекты и не может менять прототип существующих.

1.3.3. `setPrototypeOf`

Последний способ установки прототипа – специальный метод `setPrototypeOf`. Этот метод принимает уже два параметра: первый — исходный объект, а второй — объект, который мы хотим видеть в качестве прототипа для исходного.

```
var student = {
    name: 'Billy',
    sleep: function () {}
};

var person = {
    type: 'human',
    getName: function () {}
};

Object.setPrototypeOf(student, person);

student.getName();
// Billy
```

Особенности способа:

- появился только в ECMAScript 6
- близок к __proto__, но имеет особенность:
  - если мы попробуем присвоить через сеттер, геттер `__proto__` в качестве прототипа не объект, а число, то интерпретатор неявно проигнорирует это поведение; попробовав проделать тот же самый фокус с методом `setPrototypeOf`, интерпретатор поведет себя более явно и выбросит ошибку.

У метода `setPrototypeOf` есть парный метод `getPrototypeOf`. Этот метод возвращает ссылку на прототип. В отличие от `setPrototypeOf`, этот метод появился сравнительно давно в языке и позволяет нам проследовать по всей цепочке прототипов.

```
Object.getPrototypeOf(student) === person;
// true
Object.getPrototypeOf(person) === Object.prototype;
// true
Object.getPrototypeOf(Object.prototype) === null;
// true
```

1.4. Эффект затенения

Чтобы поменять значение какого-либо поля у объекта, нам достаточно выполнить простое присваивание. Но что, если мы попытаемся изменить поле, которого нет у объекта, но есть в его прототипе? Например, поле `type`.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}

var person = {
    type: 'human',
    getName: function () {}
}

console.info(student.type); // human

student.type = 'robot';

console.info(student.type); // robot

console.info(person.type); // ???
```

Выполнив простое присваивание, мы добьемся желаемого. Может показаться,

что интерпретатор, не найдя это поле у студента, перейдет в прототип и поменяет значение уже там, но он оставит это поле в прототипе неприкосновенным. Вместо этого он создаст копию на стороне объекта, но уже с новым значением. Такой эффект называется **эффектом затенения свойства**.

```
console.info(person.type); // 'human'
```

Благодаря эффекту затенения мы можем переопределить методы, находящиеся в глобальном прототипе. Например, метод `toString`, который вызывается при приведении объекта к строке.

```
Object.prototype = {
    toString: function () {}
};
student.toString();
// [object Object]
console.info('Hello,' + student);
// Hello, [object Object]
```

1.5. Поля только для чтения в прототипах

Мы можем не просто установить поле, а задать ему некоторые характеристики, например, пометить это поле как изменяемое или неизменяемое.
Допустим, у нас есть объект студента с полем `name`, которое хранит его имя. Давайте добавим еще одно поле для этого объекта. Воспользуемся методом `defineProperty` и в качестве первого параметра передадим туда сам объект, в качестве второго параметра передадим название поля, а в качестве третьего — набор характеристик. Укажем значение поля, а также укажем, что это поле неизменяемое: зададим атрибуту `writable` значение `false`.

```
var student = { name: 'Billy' };
Object.defineProperty(student,'gender', {
    writable: false,
    value: 'male',
});
```

Если мы попытаемся перезаписать это начальное значение, то интерпретатор не даст нам этого сделать и сохранит исходное, причем сделает это неявно. Чтобы сделать поведение интерпретатора явным, нам необходимо переключиться в строгий режим интерпретации. Для этого понадобится добавить дополнительную директиву `use strict` в начало нашей программы. В этом случае при попытке перезаписать поле только для чтения интерпретатор бросит ошибку, в которой сообщит нам, что поле у объекта неизменяемое.

```
'use strict';
var student = { name: 'Billy' };
Object.defineProperty(student,
'gender'
, {
writable: false,
value: 'male'
});
```

Таким же образом работают неизменяемые поля в прототипах. Создадим новое поле в нашем прототипе person. Пусть это будет поле, которое хранит текущую планету, зададим ей начальное значение и атрибут `writable: false`. Мы увидим, что при попытке перезаписать это поле от лица исходного объекта, от лица студента, интерпретатор в строгом режиме также среагирует ошибкой, не даст нам этого сделать и скажет, что поле `planet` у объекта — неизменяемое.

```
Object.defineProperty(person,
'planet', {
    writable: false,
    value: 'Earth'
});

console.info(student.planet); // Earth

student.planet = 'Mars'; // TypeError: Cannot assign to read only
 ↪  property 'planet' of object
```

1.6. Сеттеры и геттеры в прототипах

Пусть у нас есть объект `student` с уже готовым полем, которое хранит имя студента. Мы хотим добавить для студента еще одно поле, которое будет хранить его возраст, такое, чтобы с ним было удобно работать.
Например, мы хотим передавать в это поле возраст в виде некоторой строки, но преобразовывать его внутри к числу. Для этого мы определяем сеттер и при помощи функции `parseInt` передаваемую строку, в которой содержится возраст, преобразуем к числу и сохраняем во внутреннее поле.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};

Object.defineProperty(student,'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});

student.age = '20 лет';

console.info(student.age); // 20;
```

Далее мы определяем геттер, который позволяет получать уже готовое значение из этого внутреннего поля. Имеет смысл добавить поле возраста не конкретно к студенту, а в его прототип, в объект `person`. Для этого мы воспользуемся тем же самым методом `defineProperty`, но в качестве первого параметра передадим уже не студента, а прототип. Далее попробуем указать возраст для студента в виде строки и увидим, что все работает как надо.

```
var student = {
    [[Prototype]]: <person>
};
var person = {
    type: 'human'
};
Object.defineProperty(person,'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});

student.age = '20 лет';

console.info(student.age); // 20;

student.hasOwnProperty(age); // false;
```

Здесь мы вспоминаем про эффект затенения: если мы попытаемся установить некоторое поле, которого нет у объекта, но есть в прототипе, поле в прототипе не будет изменено. Вместо этого интерпретатор создаст копию этого поля на объекте с новым значением. Но если поле в прототипе определено при помощи сеттера/геттера, данный эффект работать не будет, копия поля у объекта `student` не появится.

> Если поле в прототипе определено как геттер или сеттер, то эффект затенения **не** работает.

1.7. Неперечисляемые поля в прототипах

Мы можем пометить некоторые поля у объекта как перечисляемые (значение по умолчанию) или неперечисляемые. Если поля перечисляемые, то при помощи оператора `for...in` мы можем получить весь список полей объекта.
Более того, оператор `for...in` перечисляет не только поля самого объекта, но и поля связанного с ним прототипа. Допустим, если у нашего объекта есть прототип и в нём есть поля/методы, оператор `for...in` перечислит их наряду с полями объекта.
Это может быть нежелательным поведением, и, возможно, мы хотим перечислить именно собственные поля объекта, не затрагивая при этом поля в

прототипе. Для этого нам понадобится специальный метод `hasOwnProperty`, в качестве аргумента который принимает название поля. Данный метод просто отвечает на вопрос: принадлежит ли это поле объекту или нет. Добавив это условие в оператор `for...in`, мы можем вывести только собственные поля объекта.

```
var student = {
    name: 'Billy',
    age: 20,
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    getName: function () {}
};

for (var key in student)
    if (student.hasOwnProperty(key)) console.info(key);

// 'age', 'name'
```

Аналогично этой технике мы можем воспользоваться другим методом — методом `keys`, который хранится в функции `Object`. Для этого в этот метод мы передаём объект, а на выходе получаем массив из ключей полей объекта.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}
var person = {
    type: 'human',
    getName: function () {}
}

var keys = Object.keys(student); // Получаем массив ключей

console.info(keys);

// ['name']
```

Чтобы добавить в объект неперечисляемое поле, воспользуемся методом `defineProperty`. Для этого передадим в него первым параметром сам объект, вторым параметром — название нового поля, а третьим параметром — характеристики. Укажем значение этого объекта. И с помощью специального атрибута укажем, что это поле неперечисляемое.

```
var student = { name: 'Billy' };

Object.defineProperty(student,'age', {
    enumerable: false,
    value: '20'
});

for (var key in student) console.info(key);
// 'name'

Object.keys(student);
// ['name']
```

В результате данное поле не будет участвовать в перечислениях, организуемых оператором `for...in` или методом `keys`.
Таким же образом мы можем задать неперечисляемое поле в прототипе, также воспользовавшись методом `defineProperty`. И данное поле также не будет участвовать в перечислениях.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};

var person = {
    type: 'human'
};

Object.defineProperty(person,'age', {
    enumerable: false
});

for (var key in student) console.info(key);
// 'name', 'type'
```

Важно заметить, что у глобальных прототипов для объектов или для массивов поля, обозначенные там, неперечисляемые. Мы не увидим их в попытке перечислить все поля конкретного объекта, несмотря на то, что в его прототипе может лежать глобальный прототип.

```
Object.prototype = {
    toString: function () {},
    [[Prototype]]: null
};
var person = {
    type: 'human',
    [[Prototype]]: <Object.prototype>
};

for (var key in person) console.info(key);

// 'type'
```

Конструкторы

Курс: JavaScript, часть 2: прототипы и асинхронность

20 февраля 2018 г.

Оглавление

Глава 1

Конструкторы

1.1. Конструкторы

Обычно в программе мы работаем не с одним конкретным объектом, а с целой коллекцией однотипных объектов. И нам необходимо уметь создавать объекты того же типа. Создание нового объекта такого же типа — достаточно громоздкая операция.
Можно вынести процесс создания новых объектов в конструктор.
Пусть это будет обычная функция — назовем ее `createStudent`, которая на вход принимает в качестве параметра имя студента, а на выходе отдает нам новый объект с заполненными полями и необходимыми методами.

```
1  function createStudent(name) {
2      return {
3          name: name,
4          sleep: function () {
5              console.info('zzzZZ ...');
6          }
7      };
8  }
9  var billy = createStudent('Billy');
10 var willy = createStudent('Willy');
```

Данное решение простое, но каждый раз при вызове конструктора мы будем создавать новую функцию, которая будет реализовывать метод `sleep`. Можно вынести его в прототип. Для этого создадим новый объект `studentProto`,

который будет являться прототипом для всех вновь создаваемых студентов, и перенесем туда наш метод `sleep`. После этого нам необходимо добавить туда вызов метода `setPrototypeOf`, который будет привязывать новых студентов к уже созданному нами прототипу. Каждый студент будет иметь доступ к методам, которые хранятся в прототипе для него.

```
var studentProto = {
    sleep: function () {
        console.info('zzzZZ ...');
    }
};
function createStudent(name) {
    var student = {
        name: name
    };
    Object.setPrototypeOf(student, studentProto);
    return student;
}
```

Мы можем воспользоваться уже готовым механизмом для создания конструкторов. Любая функция может быть конструктором, если мы вызовем ее при помощи специального оператора `new`. Функция `createStudent` может стать конструктором сама, но нам необходимо переписать ее реализацию и оставить только передаваемый аргумент, который будет хранить имя вновь создаваемого студента; всю остальную работу за нас сделает интерпретатор. Перед тем как исполнить код нашего конструктора, он создаст новый объект и присвоит его в переменную `this`. Далее мы заполним этот объект полями, и в конце интерпретатор за нас вернет объект, хранящийся по ссылке `this`. Можно сказать, что при вызове функции как конструктора с оператором `new` `this` внутри этой функции при исполнении будет указывать на вновь создаваемый объект.

```
var billy = new createStudent('Billy');

function createStudent(name) {
    // var this = {};
    this.name = name;
    // return this;
}
```

Такой код уже будет работать, но читается он не очень хорошо; переименуем нашу функцию просто в функцию `Student`.

```
function Student(name) {
    this.name = name;
}
var billy = new Student('Billy');
```

Функции-конструкторы принято именовать с заглавной буквы. Почему это важно?
Что произойдет, если мы попытаемся вызвать случайно, например, нашу функцию-конструктор как обычную функцию без оператора `new`? В этом случае переданное значение имени студента не будет записано в новый объект студента, а будет записано в глобальный объект в поле `name`, так как, вызывая функцию, `this` по умолчанию будет ссылаться на глобальный объект. Мы можем следовать соглашению и дополнительно включить строгий режим интерпретации, который защитит нас от такого поведения. В этом случае `this` будет иметь значение `undefined`, и мы не сможем присвоить в него никакие поля.

```
function Student(name) {
    this.name = name;
}
var billy = Student('Billy'); // Поле появится в глобальном объекте!

window.name === 'Billy'; // true

'use strict'; // TypeError: Cannot set property 'name' of undefined
```

Давайте попробуем вмешаться в работу конструктора, в работу интерпретатора и как-то поменять поведение конструктора. Например, мы захотим сами возвращать какой-то сконструированный объект. Можем ли мы это сделать? В данном случае интерпретатор нам полностью доверяет и вернет тот объект, который мы возвращаем при помощи оператора `return`.

```
function Student(name) {
    this.name = name;
    return {
        name: 'Muahahahahaha!'
    };
}
var billy = new Student('Billy');
console.info(billy.name); // Muahahahahaha
```

Но если мы попытаемся вернуть из конструктора какое-то примитивное значение — число, строку или `null`, в этом случае интерпретатор просто проигнорирует эту строку и будет работать как раньше: он будет возвращать вновь создаваемый объект.

```
function Student(name) {
this.name = name;
return null; // Evil mode on!
}
var billy = new Student('Billy');
console.info(billy.name);
// Billy
```

1.2. Конструкторы и прототипы

Допустим, у нас есть конструктор студентов и нам хотелось бы добавить в него несколько методов. Логично было бы хранить их в прототипе. Для того чтобы автоматически привязывать этот прототип для всех вновь создаваемых студентов, нам необходимо поместить его в хранилище. Оно есть у каждой функции конструктора в специальном поле prototype. Конструктор в момент исполнения выполняет дополнительный шаг: привязывает тот объект, который мы поместили в хранилище, в качестве прототипа для всех вновь создаваемых объектов; создавая новых студентов при помощи нашего конструктора, мы увидим, что у каждого из них внутреннее поле `prototype` будет ссылаться на тот объект, который мы ранее поместили в хранилище.

```
Student.prototype = {
    sleep: function () {}
};
function Student(name) {
    // var this = {};
    this.name = name;
    // Object.setPrototypeOf(this, Student.prototype);
    // return this;
}
```

Данное хранилище может вам напомнить другое хранилище, а именно то, которое расположено в специальном поле `prototype` функции `Object`. И более того, автоматически каждый создаваемый объект в JavaScript имеет в качестве прототипа объект этого хранилища. Изначально нам было не очень очевидно, каким же образом осуществлялась данная привязка, ведь мы создавали объекты при помощи литеральной конструкции, а не конструктора и оператора `new`. Но под капотом интерпретатор вызывает тот же самый конструктор `Object` оператором `new`.
Давайте подробнее поговорим про специальное поле `.prototype`:

- есть у каждой функции;
- хранит объект;
- имеет смысл только при вызове функции как конструктора;
- имеет вложенное поле `.constructor` (неперечисляемое, хранит ссылку на саму функцию).

Обращаясь к полю `.constructor`, мы можем получить доступ к конструктору, т.е можем, например, выяснить имя конструктора конкретного объекта. Например, если мы сконструировали нового студента `Billy` на основе конструктора Student, мы можем обратиться к этому полю у `Billy`. У `Billy` этого поля нет, но оно есть в прототипе, который, как мы знаем, изначально хранится в хранилище Student.`prototype`. И так как данное поле хранит ссылку на функцию, мы можем посмотреть имя этой функции, обратившись к полю name.

```
function Student(name) {
    this.name = name;
}

Student.prototype.constructor === Student; // true
var billy = new Student('Billy');
console.info(billy.constructor.name); // Student
```

Важно помнить о поле `.constructor`, так как мы очень легко можем его перезаписать. В нашем случае мы это и сделали. Мы просто поместили в поле .prototype, в это хранилище, новый объект, тем самым уничтожив поле `.constructor`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = {
    sleep: function () {}
};
```

Чтобы этого не произошло, достаточно не перезаписывать тот объект, который хранится изначально в этом хранилище, а дополнять его новыми методами. Таким образом, мы получим на основе этого конструктора новые объекты. Они будут иметь доступ как к методам из прототипа, так и к специальному полю `.constructor`.

```
function Student(name) {
    this.name = name;
}

Student.prototype.sleep = function () {
    console.info('zzzZZ ...');
}

var Billy = new Student('Billy');
billy.sleep(); // zzzZZ ...
billy.constructor === Student; // true
```

```
function Student(name) {
    this.name = name;
}
Student.prototype.sleep = function () {};
```

```
function Person() {
    this.type = 'human';
}
Person.prototype.getName = function () {
    return this.name;
}
```

1.3. Конструкторы и цепочки прототипов

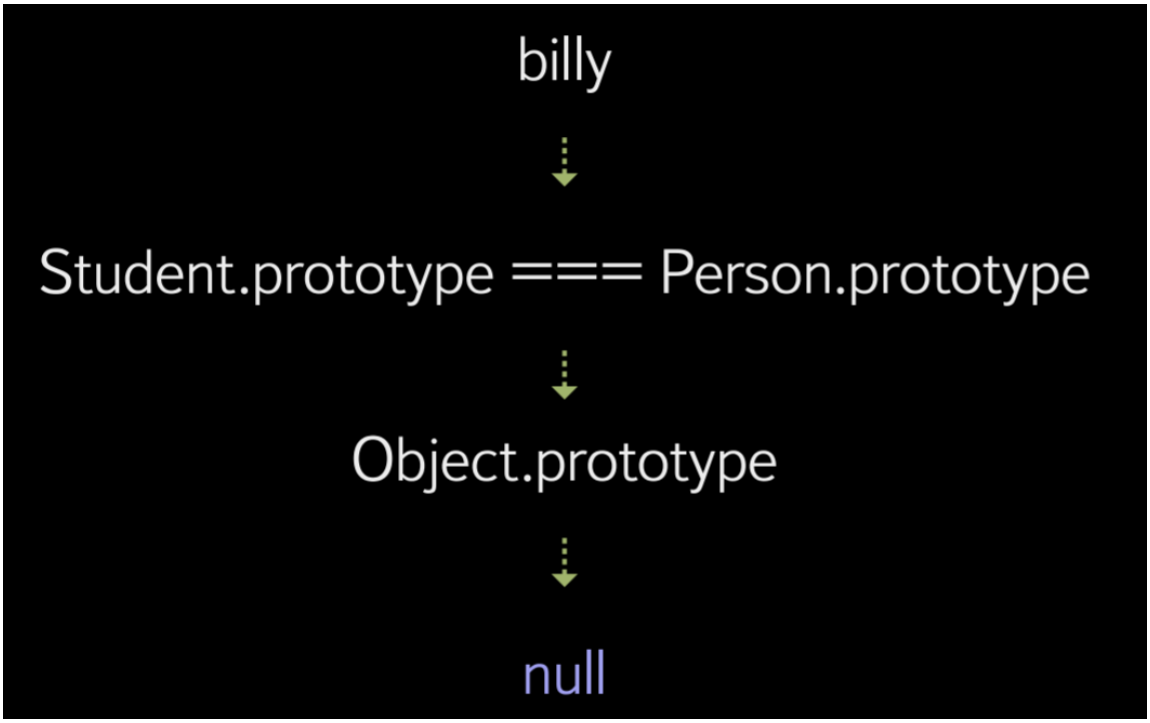
У нас есть некоторый конструктор студентов, который принимает на вход имя студента и записывает его в поле name. А также хранилище, в поле .prototype нашего конструктора мы поместили объект, который хотим видеть в качестве прототипа для всех вновь создаваемых студентов. Там сейчас один метод – sleep.
Допустим, у нас есть более абстрактный конструктор, который также имеет хранилище и в котором расположен другой метод – getName.
В данном случае нам бы не хотелось дублировать реализацию этого метода в хранилище конструктора студентов. Для этого мы можем пойти самым простым путем и хранилище конструктора студентов сделать на основе хранилища конструктора Person: присваиваем в специальное поле prototype ссылку на хранилище конструктора Person. Далее, мы можем расширить наше хранилище для студентов более специфичным для них методом – методом sleep. Итак, мы можем создавать теперь новых студентов, которые будут иметь доступ к методу sleep из своего хранилища и к методу getName, которое изна-

```
function Student(name) {
this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};
```

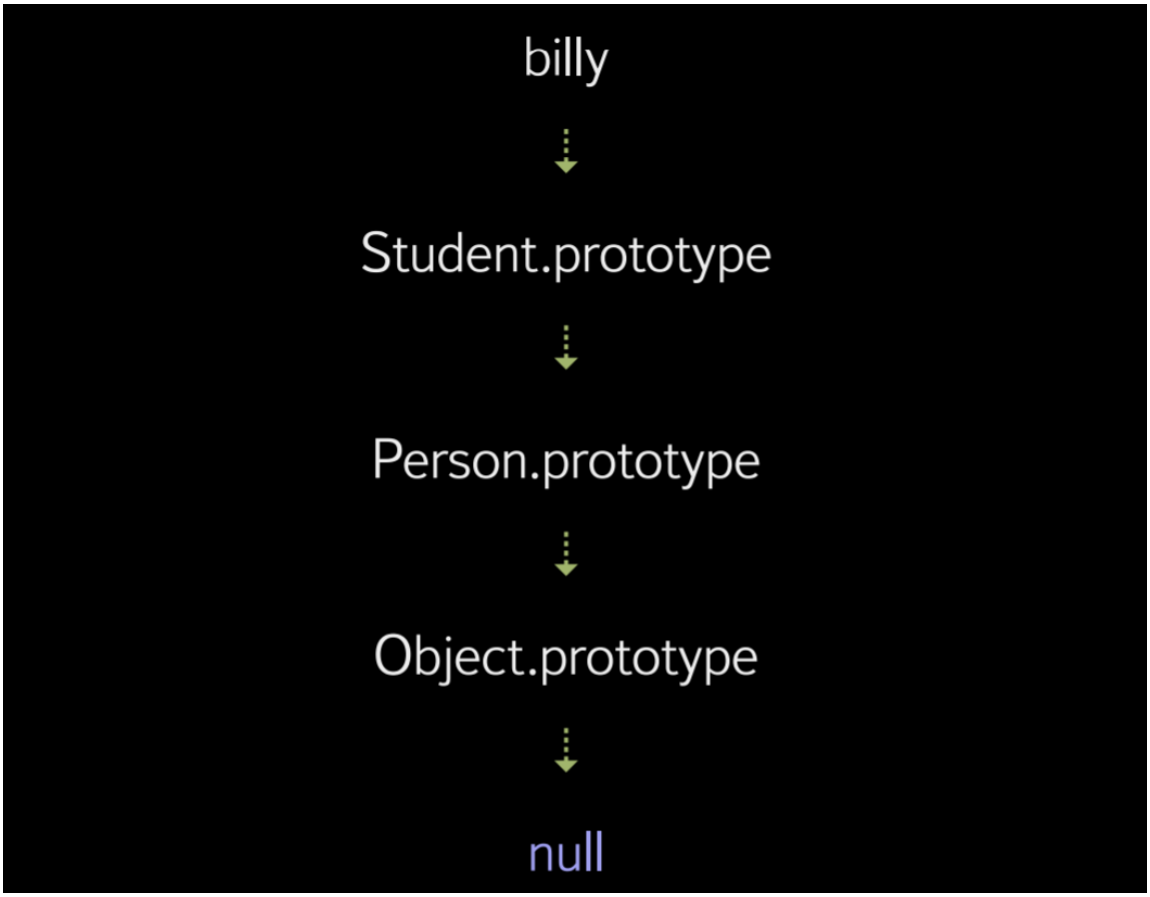
чально было в хранилище конструктора `Person`.
Данный способ имеет подводный камень. Если мы в своей программе попробуем использовать объекты другого типа, создадим для них конструктор и в хранилище этого конструктора запишем ссылку на то же самое хранилище, которое связано с конструктором `Person`, безусловно, наш новый объект – преподаватель – получит доступ к методам из этого хранилища `Person`. Но мы с удивлением обнаружим, что преподаватель получит доступ и к методам, которые мы ранее определили только для студентов, например к методу `sleep`.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Person.prototype;
Student.prototype.sleep = function () {};

function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Person.prototype;

var sergey = new Lecturer('Sergey');

sergey.sleep(); // zzzZZ ...
```

Это происходит потому, что все три этих хранилища хранят сейчас ссылки на один и тот же объект. Наша цепочка прототипов выглядит примерно так.



Это нежелательное поведение и нам бы хотелось, чтобы цепочка выглядела примерно так.



1.3.1. Метод `create`

Решить эту проблему нам поможет специальный метод `create`. Вместо обычного присваивания мы в хранилище для конструктора студентов будем записывать объект, который нам этот метод возвращает. Такой же трюк мы можем проделать и с конструктором преподавателя. Попробуем теперь создать нового преподавателя и вызвать у него метод, который мы определили только для студентов. Благодаря методу `create`, мы увидим, что преподаватели не будут иметь доступ к методам, которые мы определили только для студентов.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};


function Lecturer(name) {
    this.name = name;
}
Lecturer.prototype = Object.create(Person.prototype);

var sergey = new Lecturer('Sergey');
sergey.sleep(); //TypeError: sergey.sleep is not a function
```

Попробуем разобраться, как работает метод `create`. Метод создает пустой объект, прототипом которого становится объект, переданный первым аргументом. Есть прототип для фруктов, которые хранит единственное поле. Оно говорит нам о том, что все фрукты полезные. И на основе этого прототипа фруктов мы будем создавать новые фрукты при помощи метода `create`, передавая в качестве аргумента наш прототип. Так мы можем создать яблоко и проверить, что оно действительно полезное, несмотря на то, что этого поля у самого яблока нет. Оно есть у его прототипа.

```
var fruitProto = {
isUsefull: true
}
var apple = Object.create(fruitProto);
apple.isUsefull; // true
```

Внутри метод `create` устроен достаточно просто. Он создает простейшие конструкторы из возможных, а именно: пустую функцию. Далее, в хранилище этого конструктора он записывает ссылку на тот объект, который мы передаем в качестве первого аргумента, тот объект, который хотим видеть в качестве прототипа для всех создаваемых объектов. Далее, он при помощи этого конструктора создает новый объект и возвращает его. Таким образом, все вновь

создаваемые объекты будут иметь в качестве прототипа тот объект, который мы передаем первым аргументом.

```
var apple = Object.create(fruitProto);

Object.create = function(prototype) {
    // Простейший конструктор пустых объектов
    function EmptyFunction() {};
    EmptyFunction.prototype = prototype;
    return new EmptyFunction();
};
```

В метод `create` мы можем передавать не только объекты, но и, например, значение `null`. В этом случае мы создадим объект, в качестве прототипа которого не будет выступать ни один из объектов, даже глобальный прототип для всех объектов. И мы не получим доступ к методам из этого глобального прототипа. Метод `create` помогает нам связать два хранилища разных конструкторов так, чтобы они не ссылались на один и тот же объект. И хранилище для конструктора студентов будет представлять из себя отдельный объект, но во внутреннем поле `prototype` которого лежит ссылка на другое хранилище – хранилище конструктора `Person`. Далее мы можем расширить хранилище для студентов специфичными для них методами.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};
```

И здесь мы допустили ту же самую ошибку, что и ранее. Мы полностью перезаписали хранилище студентов и забыли о поле конструктора.
Давайте его вернем. Достаточно просто присвоить в него ссылку на функцию.

```
function Student(name) {
    this.name = name;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;
```

Итоговое решение нашей задачи выглядит примерно так.

```
function Person() {
    this.type = 'human';
}

Person.prototype.getName = function () {
    return this.name;
};

function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');
```

1.4. Инспектирование связей между объектами, конструкторами и прототипами

1.4.1. `getPrototypeOf`

Первый способ — это метод `getPrototypeOf`. На вход он принимает в себя объект, а на выходе дает ссылку на прототип для этого объекта. В данном случае

у нас есть объект студента, в качестве прототипа для которого выступает объект `person`. Вызвав метод `getPrototypeOf` и передав туда ссылку на объект студента, мы на выходе получим на объект `person`.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}

var person = {
    type: 'human',
    getName: function () {}
}

Object.getPrototypeOf(student) === person; // true
```

1.4.2. `isPrototypeOf`

Следующий способ инспектирования связей между объектами и прототипом предлагает нам метод `isPrototypeOf`. Допустим, у нас есть некий конструктор студентов, который на вход принимает имя студента и сохраняет его в поле `name` для каждого студента; хотим к каждому студенту привязать некоторый прототип. Мы помещаем этот прототип в поле `prototype`, в хранилище, и делаем его объектом на основе другого хранилища — конструктора `person` при помощи метода `create`. Далее, мы в наше хранилище добавляем специфичный для студентов метод `sleep` и восстанавливаем конструктор. Попробуем теперь создать нашего студента и воспользоваться для проверки связи между созданным студентом и его прототипом методом is`prototype`Of. Данный метод отвечает на вопрос: «Является ли объект прототипом для того объекта, который мы передаем в качестве аргументов?»

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

Student.prototype.isPrototypeOf(billy); // true
```

Более того, данный метод позволяет инспектировать не только прямую связь, но и связь конечного объекта с одним из прототипов цепочки. Таким образом, он даст утвердительный ответ и на вопросы: «Является ли объект, который лежит в хранилище конструктора `person`, прототипом для `Billy`?» и «Является ли глобальный прототип для всех объектов прототипом для `Billy`?»

```
Person.prototype.isPrototypeOf(billy); // true

Object.prototype.isPrototypeOf(billy); // true
```

1.4.3. `instanceof`

Следующий способ инспектирования связей между объектами и конструкторами — специальный оператор `instanceof`. Он позволяет ответить на вопрос: «Является ли объект объектом определенного конструктора?» В данном случае мы создали нового студента `Billy` на основе конструктора Student и проверяем, является ли `Billy` объектом этого конструктора.

```
function Student(name) {
    this.name = name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.sleep = function () {};

Student.prototype.constructor = Student;

var billy = new Student('Billy');

billy instanceof Student; // true
```

Но этот оператор работает несколько сложнее, и он ответит, что `Billy` является и объектом конструктора `Person`, хотя напрямую это не так.

```
billy instanceof Person; // true

billy instanceof Object; // true
```

Но можно заметить, что хранилище, связанное с конструктором `Person`, лежит в цепочке прототипов до `Billy`. Если мы немножечко переформулируем то, как работает этот оператор, все встанет на свои места. Можно сказать, что он отвечает на вопрос: «Является ли `Billy` студентом?» или: «Является ли `Billy` личностью?» Более того, `Billy`, конечно же, является объектом — тоже правда.

Давайте разберем, как работает оператор `instanceof`. Например, как он проверяет, связаны ли между собой объект `Billy` с конструктором `Person`. Вначале он проверяет, является ли прототипом для `Billy` объект, который хранится в хранилище этого конструктора. И это не так. Тогда он проверяет следующую гипотезу: «А, может быть, прототипа у `Billy` совсем нет, и там хранится значение `null`?» Эта гипотеза также не подтверждается, и тогда он идет по цепочке прототипов. Он проверяет, является ли прототип прототипа `Billy` объектом, который хранится в хранилище конструктора `Person`. На этот раз гипотеза подтверждается, и в этом случае данный оператор отвечает нам `true`.

```
billy instanceof Person;
billy.__proto__ === Person.prototype;
// false -> Может, там null?
billy.__proto__ === null;
// false -> Идём дальше по цепочке
billy.__proto__.__proto__ === Person.prototype;
// true -> Возвращаем true
```

Если мы попробуем проинспектировать объект с самой короткой цепочкой прототипов, оператор `instanceof` возвращает нам `false`. Мы создаем объект и проверяем, что этот объект — это действительно объект, но оператор `instanceof` возвращает нам `false`. Если мы еще раз посмотрим, как работает этот оператор, все встанет на свои места. Итак, в первую очередь, он проверяет, является ли прототипом для нашего одинокого объекта глобальный прототип для всех объектов. Это не так. Тогда он проверяет: «Возможно, у нашего одинокого объекта совсем нет прототипа, и во внутреннем поле `prototype` лежит значение `null`?» В этом случае оператор `instanceof` при положительном подтверждении такой гипотезы возвращает нам `false`.

```
var foreverAlone = Object.create(null);

foreverAlone instanceof Object; // false

Object.create(null).__proto__ === Object.prototype;
// false -> Может, там null?

Object.create(null).__proto__ === null;
// true -> Так и есть, возвращаем false!
```

1.5. Решение проблемы дублирования кода в конструкторах

У нас есть конструктор студентов, конструктор преподавателей и конструктор личностей. Если мы посмотрим на конструкторы преподавателей и студентов, мы увидим, что они похожи и выполняют одинаковую работу: принимают в

качестве аргумента имя студента или преподавателя и сохраняют его в поле `name`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Нам бы хотелось избежать этого дублирования. Проще всего — вынести этот общий код в отдельный конструктор `Person`. Перенесем туда строчку, которая сохраняет имя студента или преподавателя в конструктор `Person`.

```
function Student(name) {
    this.name = name;
}

function Lecturer(name) {
    this.name = name;
}

function Person() {
    this.type = 'human';
}
```

Если мы сейчас попытаемся создать нового студента и передадим туда имя, мы не получим желаемого результата, так как мы забрали этот код из конструктора для студентов и переместили его в конструктор для личностей. Поэтому нам необходимо немножечко изменить реализацию конструктора студентов и добавить туда вызов конструктора личностей. Мы можем это сделать при помощи метода `call`. Вызываем наш конструктор личностей при помощи этого

метода и в качестве контекста передаем туда `this`, а в качестве второго аргумента передаем имя создаваемого нами студента.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

function Student(name) {
    // this ссылается на новый объект студента
    Person.call(this, name);
}

var billy = new Student('Billy');

console.info(billy.name); // Billy
```

Благодаря тому, что в конструкторе `this` будет ссылаться на новый создаваемый объект, в нашем случае — на нового студента, мы вызовем конструктор `Person` с контекстом в виде этого студента, и таким образом передаваемое имя мы положим в поле name именно для студента. Все будет работать.

1.6. Вызов затеняемого метода в затеняющем

Допустим, у нас есть некоторый конструктор `Person`, конструктор личностей, который на вход принимает имя этих личностей и кладет в поле name. Также мы определяем прототип для всех личностей с методом `getName`, который возвращает это имя. Положим этот прототип в хранилище конструктора `Person`.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}
```

Далее мы захотим использовать в нашей программе объекты другого типа и создадим конструктор для них. Прототипом для них мы сделаем объект на основе прототипа для личностей, но захотим изменить метод `getName`. Нам не хочется дублировать тот код, который есть в методе `getName` прототипа личностей, а лишь дополнить и добавлять к нему некую строку.
Вначале нам может показаться, что мы можем вызвать просто метод `getName` из прототипа `Person` внутри метода `getName` прототипа для студентов. Но в этом случае произойдет рекурсивный вызов того же самого метода. И интерпретатор нам скажет, что количество вызовов заполнило весь стек.

```
1  Student.prototype = Object.create(Person.prototype);
2
3  Student.prototype.getName = function () {
4      return 'Student ' + this.getName();
5  };
6
7  var billy = new Student('Billy');
8
9  billy.getName(); // RangeError: Maximum call stack size exceeded
```

Когда мы вызываем метод `getName` у созданного нами студента, например, `Billy`, внутри этого метода `this` будет ссылаться, собственно, на этого самого студента. Так как метода `getName` у самого `Billy` нет, он пойдет искать его в прототипе и найдет метод в прототипе, который хранится в хранилище конструктора `Student` в поле `Student.prototype`. Фактически метод будет вызывать сам себя.
Каким образом мы можем решить эту проблему? Самое простое — использовать другое название метода вместо эффекта затенения. В этом случае все будет работать. Мы будем вызывать метод с другим названием, например, getStudentName, заходить внутрь этого метода, `this` внутри него будет по-прежнему ссылаться на объект `Student`, то есть на `Billy`, попробуем найти у `Billy` метод `getName`, не найдем его там, пройдем по цепочке прототипов в объект, который хранится в хранилище конструктора студентов Student.`prototype`. И там мы этого метода не найдем, проследуем дальше по цепочке прототипов и уже перейдем в хранилище, которое хранится в конструкторе `Person`. Там мы этот метод находим, спокойно его вызываем, возвращаем имя студента, добавляем к нему наш необходимый префикс, и все работает, как нужно.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getStudentName = function () {
    return 'Student ' + this.getName();
};

var billy = new Student('Billy');

billy.getStudentName();
```

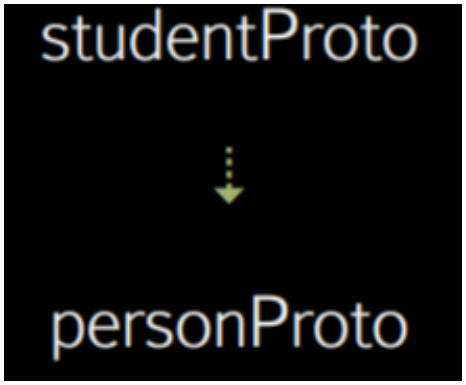
Более элегантным способом будет использование метода `call`. Мы можем напрямую в затеняющем методе вызывать затеняемый при помощи этого метода `call`, но передавать туда текущий контекст. А текущий контекст будет ссылаться на создаваемого объекта конструктора студентов, а именно на студента. Таким образом мы вызываем затеняемый метод от лица этого самого студента, получаем имя этого студента и добавляем к нему префикс `student`.

```
function Person(name) {
    this.type = 'human';
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.getName = function () {
    return 'Student ' + Person.prototype.getName.call(this);
};
```

1.7. Сравнение трёх подходов к конструированию объектов: функции-конструкторы, метод `create`, «Классы»

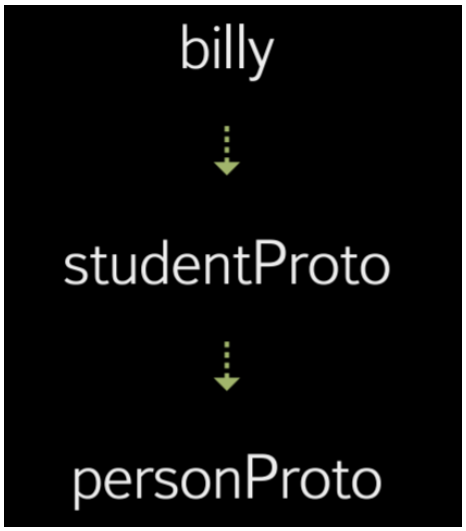
Для конструирования объектов и построения связей между ними достаточно лишь использовать только метод `create`.
Давайте снова попробуем создать конструктор для студентов. Для этого нам вновь понадобится ряд прототипов, выстроенных в цепочку. На этот раз мы воспользуемся только методом `create` и обычными объектами, так как любой прототип по факту — это обычный объект. Итак, для начала нам понадобится прототип личности. Создадим простой объект `personProto` и добавим туда метод `getName`, который будет возвращать имя нашей личности. На основе этого прототипа создадим более специфичный прототип уже для студентов. При помощи метода `create` мы свяжем два этих прототипа в цепочку. Далее мы можем расширить наш прототип для студентов уже специфичными для студентов методами, например, методом `sleep`.

```
var personProto = {
    getName: function () {
        return this.name;
    }
};
var studentProto =
 ↪  Object.create(personProto);

studentProto.sleep = function () {};
```



Далее на основе прототипа для студентов мы уже будем создавать студентов. Для этого снова воспользуемся методом `create`. Таким образом, благодаря этому методу мы встроим нашего студента в цепочку прототипов. И далее всё, что нам останется, это только дополнить нашего студента полями, которые необходимы каждому конкретному студенту с конкретными значениями. В данном случае мы присваиваем нашему студенту имя `Billy`.

```
var billy =
 ↪  Object.create(studentProto);

billy.name = 'Billy';
```



Благодаря такому подходу нам понадобилось значительно меньше строчек кода, чем в классическом, чтобы научить нашу программу создавать новые объекты студентов. Но и здесь мы можем произвести улучшение. Для этого нам понадобится ещё одна возможность метода `create`, а именно — данный метод принимает не один аргумент, а два. В качестве второго аргумента вы можете

передать необходимые поля с их начальными значениями и характеристиками, которые вы хотите видеть при создании объекта в итоговом объекте.

```
var apple = Object.create(fruit, {
    shape: { value: 'round', writable: false },
    color: { value: 'Green' },
    amount: { writable: true }
});

apple.amount = 'half';
```

Посмотрим, каким образом мы можем использовать данную возможность. Чтобы создавать наших студентов в одну строку, нам понадобится дополнительная функция-помощник, назовём её `create` и положим в наш прототип для студентов. Иногда такие функции называют фабриками — фабриками объектов. Данная функция будет на вход принимать в себя имя студента, а внутри себя вызывать метод `Object.create`, в качестве контекста передавать `this` и в качестве второго аргумента передавать набор полей, которые мы хотим видеть у студента, а именно: мы хотим у него видеть поле `name`, и мы сразу заполняем его тем, что нам приходит в фабрику. Таким образом мы можем создавать новых студентов в одну строчку, как и при классическом подходе при помощи оператора `new`. В данном случае мы просто вызываем наш метод `create`, который лежит в прототипе студента, передаём туда имя и получаем новый объект, нового студента с этим именем.

```
var personProto = {};

personProto.getName = function () { return this.name; }

var studentProto = Object.create(personProto);

studentProto.sleep = function () {};

studentProto.create = function (name) {
    return Object.create(this, {
        name: { value: name }
    });
}

var billy = studentProto.create('Billy');
```

Единого мнения, какой из этих подходов лучше, у разработчиков нет.
Также есть новая версия спецификаций, которая вводит новую синтаксическую конструкцию, а именно «классы». И вместо функций-конструкторов мы можем определить класс и на основе этого класса создавать новые объекты.

```
function Student(name) {
    this.name = name;
}

Student.prototype.getName = function () {
    return this.name
};

class Student {
    constructor(name) {
        this.name = name;
    }

    getName() {
        return this.name;
    }
}
```

Классы - это не более, чем обычные функции-конструкторы. У них также есть специальное поле `prototype` — хранилище для прототипов, для всех вновь создаваемых этим классом объектов. Также если мы посмотрим тип классов, то мы увидим, что это обычная функция.

```
class Student {
// ...
}

var billy = new Student('Billy');

billy.getName(); // Billy

Student.prototype.isPrototypeOf(billy); // true

typeof Student; // function
```

Таким образом, дискуссия трансформировалась в другую: что выбрать, классы или метод `Object.create`? Наиболее полно её раскрыл в статье разработчик Eric Elliott, и я рекомендую вам всем ознакомиться с ней.

Асинхронность

Курс: JavaScript, часть 2: прототипы и асинхронность

4 апреля 2018 г.

Оглавление

Глава 3

Асинхронный код

3.1. Асинхронный код

Все примеры, которые мы вам показывали в наших лекциях ранее, были написаны в синхронном стиле: если у нас было описано несколько функций, и мы вызывали их одну за другой.
Чтобы понять, как работает асинхронный код, давайте рассмотрим две новые структуры: стек вызовов и очередь событий.

3.1.1. Стек вызовов

Стек вызовов — это структура данных, которой оперирует интерпретатор. Как только мы начинаем исполнять наш код, интерпретатор складывает в стек вызовов анонимную функцию. Как только выполнение нашего кода заканчивается, анонимная функция выталкивается из стека. Если при выполнении нашего кода встречается вызов другой функции, то интерпретатор складывает эту функцию в стек вызовов.
В нашем случае при выполнении анонимной функции мы встретили вызов функции `toCheerUp`: мы положили ее в стек вызовов и начали ее интерпретировать.

```
function prepareCoffee()
 ↪  {
    toStove();
}
function toCheerUp() {
    prepareCoffee();
}
toCheerUp();
```

В функции `toCheerUp` мы встречаем вызов другой функции — `prepareCoffee`: мы складываем эту функцию в стек вызовов и идем ее интерпретировать.

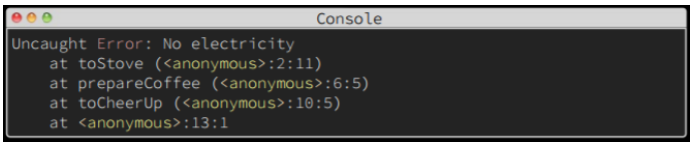
```
function prepareCoffee()
 ↪  {
    toStove();
}
function toCheerUp() {
    prepareCoffee();
}
toCheerUp();
```

В функции `prepareCoffee` мы вызываем функцию `toStove`. Она выводит на консоль некоторую строку. Как только интерпретация этой функции заканчивается, функция пропадает из стека, и мы возвращаемся к предыдущей функции. А она тоже достается из стека. Аналогичным образом мы поступаем с двумя оставшимися функциями. Они пропадают из стека, и выполнение нашего кода заканчивается.

Давайте рассмотрим следующий пример. Перепишем функцию `toStove` таким образом, чтобы она выбрасывала исключение при помощи метода `throw`. Если мы дойдем по стеку вызовов до функции `toStove` и она выбросит исключение,

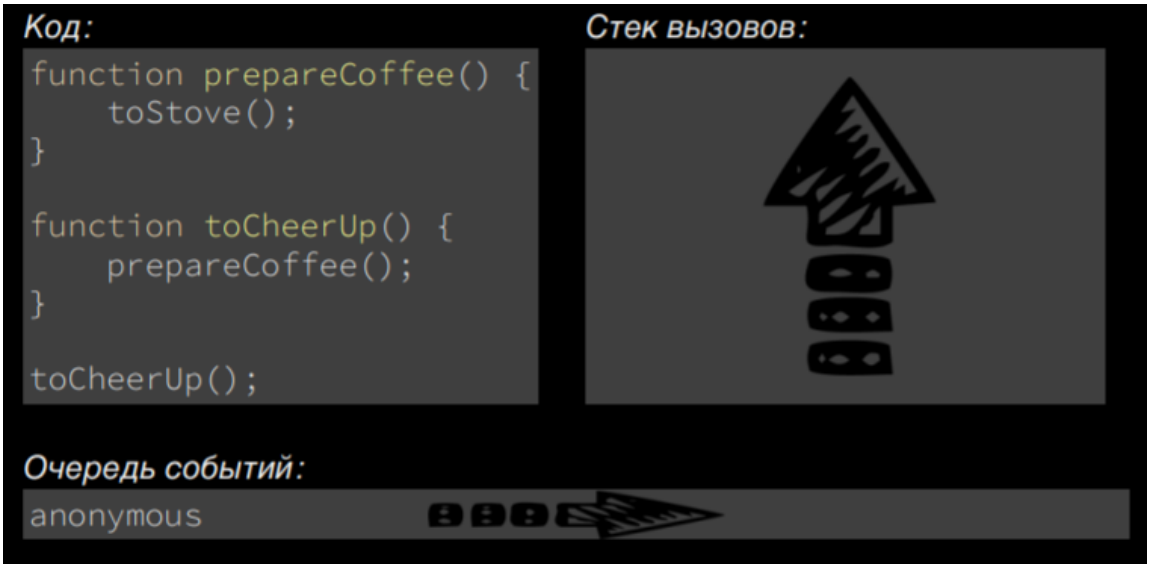
то в консоли мы увидим следующую ошибку: мы не можем поставить кофе на плиту, потому что нет электричества, и увидим стек вызовов, где ошибка произошла.

```
1  function toStove {
2      throw new Error('No
       ↪  electricity');
3  }
4
5  function prepareCoffee()
   ↪  {
6      toStove();
7  }
8  function toCheerUp() {
9      prepareCoffee();
10 }
11 toCheerUp();
```



3.1.2. Очередь событий

Как только интерпретатор начинает исполнять наш код, он складывает анонимную функцию не сразу в стек вызовов, а для начала помещает ее в очередь событий.



Далее работа очереди и стека согласуются следующим образом: как только стек вызовов пустеет, он достает первую функцию из очереди событий. В нашем случае это анонимная функция.
Далее мы начинаем интерпретировать код, который написан в анонимной функции. Встречаем вызов функции `toCheerUp`, она вызывает функцию `prepareCoffee`, она вызывает функцию `toStove` и т.д. Мы складываем функции в стек вызовов, как только функции завершаются, мы достаем их из стека вызовов. Как только стек опустел, наша программа снова обращается к очереди событий, и если там есть новая функция, то она перекладывает ее в стек вызовов и начинает ее исполнять. В нашем случае в очереди событий ничего нет, значит, наша программа завершится. Таким образом работает цикл событий.

3.2. Системные таймеры

3.2.1. `setTimeout`

Первым аргументом `setTimeout` принимает функцию, которая будет вызвана через `delay` миллисекунд, которые указываются вторым параметром. Третий параметр и все остальные — это аргументы, с которыми будет вызвана функция.

```
1 setTimeout(func[, delay, param1, param2, ...]);
```

В качестве первого аргумента `func` можно передать также строчку. Эта строка будет проинтерпретирована, однако не рекомендуется использовать этот вариант, поскольку он устарел и остается для обратной совместимости.
Объявим две функции `fromStove` и `toStove`, которые ставят и снимают кофе с плиты соответственно. Заводим системный таймер на 5000 миллисекунд, который вызовет функцию `fromStove`.

```
function toStove {
return 'Поставить на плиту';
}

function fromStove {
return 'Снять с плиты';
}

toStove();
setTimeout(fromStove, 5000);
```

Когда интерпретатор начинает выполнять наш код, он помещает анонимную функцию в очередь событий, а затем в стек, т.к. он пуст. Мы начинаем ее выполнять и вызываем `toStove`. Затем следующим шагом мы вызываем функцию `setTimeout` и передаем туда два аргумента: `callback fromStove` и время, через которое нужно позвать этот `callback`. Интерпретатор запоминает, что нужно завести системный таймер и запускает отсчет.
Через 5 секунд интерпретатор поймет, что нужно вызвать системный таймер, и кладет `callback fromStove` в очередь событий. При этом знание о том, что нужно завести системный таймер, из интерпретатора пропадает. Если стек событий при этом пустой, то функция `fromStove` пропадает из очереди событий и попадает в стек, выполняется и возвращает строчку 'Снять с плиты'.

3.2.2. `setInterval`

Так же, как и `setTimeout`, интервал принимает следующие аргументы: функцию `callback`, которую мы будем вызывать через `delay` миллисекунд, которая указывается во втором параметре, и набор аргументов, с которыми нужно вызвать `callback`. Но, в отличие от `setTimeout`, `setInterval` будет вызывать функцию не один раз, а до тех пор, пока его не остановят.

```
setInterval(func[, delay, param1, param2, ...]);
```

Объявим две функции `toStove`, которая ставит кофе на плиту, и `toStir`, которая помешивает кофе. Чтобы не забыть помешивать кофе, мы заводим системный таймер — `setInterval`. Передаем туда два аргумента: `callback`, который нужно вызвать — `toStir`, и время.

```
function toStove {
return 'Поставить на плиту';
}

function toStir {
return 'Помешивать';
}

toStove();
setInterval(toStir, 1000);
```

Интерпретатор начинает выполнять наш код, кладет анонимную функцию в очередь, которая помещается в стек вызовов, выполняется и вызывает функцию `toStove`. Возвращает нам строку 'Поставить на плиту' и передает выполнение дальше.
Далее мы вызываем функцию `setInterval`, которая заводит системный таймер. Системный таймер будет срабатывать каждую секунду, поэтому, после того как пройдет одна секунда, в очереди событий окажется `callback` — `toStir`. Если стек вызовов при этом пустой, то мы сразу выполним эту функцию и получим результат. Через две секунды также выполнится системный таймер и снова положит функцию `toStir` в очередь событий. Аналогичные действия произойдут через 3, 4 и т.д секунды — пока наш системный таймер не остановит.

Чтобы остановить системный таймер, мы воспользуемся парным к нему методом — в данном случае методом `clearInterval`. Этот метод принимает на вход некоторый идентификатор — это результат вызова метода `setInterval`. В большинстве случаев id — это число, однако в документации нигде явно

этого не сказано. После вызова функции `clearInterval` из памяти интерпретатора пропадет знание о том, что нужно вызывать функцию `toStir`. Таким образом, системный таймер очистится.

```
var id = setInterval(toStir, 1000);
clearInterval(id);
```

3.3. Работа с файлами

Еще один способ положить функцию в очередь событий — это выполнить асинхронную операцию, в результате которой будет выполнен `callback` — функция обратного вызова. В качестве асинхронной операции в данном примере мы рассмотрим операции работы с файловой системой.
В следующем примере мы подключаем библиотеку `fs` с помощью функции `require`. Чтобы прочитать файл `data.json`, который лежит в той же директории, что и файл с нашим исходным кодом, мы воспользуемся переменной `__dirname`. Далее мы вызываем метод `readFileSync`, которому передаем два аргумента: первый аргумент — это имя файла, который мы хотим прочитать, второй аргумент — это кодировка, в которой нужно прочитать файл. Если не указать кодировку, то в переменной `data` окажется буфер с данными. Если же кодировка указана, то в переменной `data` мы увидим строку.

```
var fs = require('fs');
var fileName = __dirname + '/data.json';
var data = fs.readFileSync(fileName,
'utf-8');
console.log(data); // readFileSync: 3ms
```

В нашем случае в файле `data.json` лежит некая строка в формате `JSON`.
В нашем случае метод `readFileSync` является синхронным, на это указывает суффикс `Sync` в его названии. Давайте измерим время выполнения операции `readFileSync`. Сделаем мы это при помощи вызова метода `console.time`, передавая туда первым аргументом некоторый идентификатор. В нашем случае мы передадим туда строку с названием метода. После этого начинается отсчет времени. Мы выполняем функцию `readFileSync` и останавливаем отсчет времени при помощи вызова функции `console.timeEnd`, передавая туда тот

же самый идентификатор. В результате на консоли мы увидим время чтения файла: три миллисекунды.
Если мы возьмем файл побольше, например, файл `bigData.mov` и выполним те же самые замеры, мы увидим, что время файла заняло аж целых три с половиной секунды.

```
var fs = require('fs');
var fileName = __dirname + '/bigData.mov';
console.time('readFileSync');
var data = fs.readFileSync(fileName,
'utf-8');
console.timeEnd('readFileSync');
console.log(data); // readFileSync: 3567ms
```

Это огромное время, на протяжении которого интерпретатор ничего не делал. Для того, чтобы избавиться от этого негативного эффекта, мы можем заменить синхронную функцию `readFileSync` на ее асинхронный аналог `readFile`. Однако если это мы сделаем в лоб, то в переменной `data` мы увидим `undefined`.

```
var fs = require('fs');
var fileName = __dirname + '/data.json';
var data = fs.readFile(fileName,
'utf-8');
console.log(data); // undefined
```

В самом деле, запуская `readFile`, мы всего лишь отдаем инструкцию операционной системе на чтение файла. Когда чтение файла завершится, будет вызвана функция `callback`, которая передается третьим аргументом в функцию `readFile`.

3.4. Функция обратного вызова (`callback`)

`Callback` или функция обратного вызова в `Node.js` выглядит следующим образом. Первым аргументом она принимает ошибку, которая возникла в результате асинхронной операции. Если при выполнении асинхронной операции не возникло ошибок, то в качестве первого аргумента принято передавать `null`.

Второй аргумент содержит данные, с которыми завершилась асинхронная операция. В нашем случае — чтение файла — в переменной `data` будет лежать содержимое этого файла.

```
function cb(err, data) {
    if (err) {
        console.error(err.stack);
    } else {
        console.log(data)
    }
}
```

Достоинства `callback`:

- оптимальная производительность;
- не нужно подключать дополнительные библиотеки.

Недостатки

Уровень вложенности нашего кода растет вместе с ростом его сложности. Читаем файл `data.json` и обрабатываем результат, то есть выводим на консоль в предпоследней строке. Эта строка находится на втором уровне вложенности. Однако если мы захотим прочитать еще один файл, например, файл `ext.json` в случае, если файл `data.json` был прочитан удачно, то обработчик положительного результата будет находиться уже на четвертом уровне вложенности, и ситуация будет ухудшаться с увеличением сложности нашего кода.

```
var fs = require('fs');

fs.readFile('data.json', function (err, data) {
    if (err) {
        console.error(err.stack);
    } else {
        fs.readFile('ext.json', function (e, ext) {
            if(e) {
                console.error(e.stack);
            } else {
                console.log(data + ext);
            }
        });
    }
});
```

Обработчик ошибок и данных, разных по своей природе, находится в одном месте кода.
Все наши `callback`-и начинались с `if`. Если в переменной `error` находится какая-то ошибка, то мы идем по одной ветке кода, иначе — идем по другой ветке кода. Это увеличивает сложность нашего кода.
Мы можем пропустить некоторые исключения, когда пишем функцию, которая принимает `callback`.
Мы хотим прочитать два файла параллельно. Для этого реализуем функцию `readTwoFiles`, которая принимает один единственный аргумент, `callback`. Первой строчкой кода мы создадим переменную `tmp`, которая в себе будет содержать результат чтения первого файла. Далее мы запускаем чтение файлов. Файл, который будет прочитан первым, окажется в переменной `tmp`. Файл, который будет прочитан второй по счету, вызовет функцию `callback`. При этом в качестве данных он передаст конкатинацию своего содержимого и переменной `tmp`.

```
var fs = require('fs');

function readTwoFiles(cb) {
    var tmp;

    fs.readFile('data.json', function (err, data) {
        if (tmp) {cb(err, data + tmp);}
        else { throw Error('Mu-ha-ha!'); }
    });

    fs.readFile('ext.json', function (err, data) {
        if (tmp) {cb(err, data + tmp);}
        else { tmp = data; }
    });
```

Однако если мы при написании нашего кода допустим неконтролируемое исключение, например, мы можем это симулировать при помощи вызова метода `throw`, то вызывающая сторона, то есть код, который позовет нашу функцию `readTwoFiles`, никогда не получит `callback` с ошибкой.
Мы пропустили эту ошибку и сделали наш код ненадежным. Перменная `tmp` выглядит очень неуместно, и так получается, когда мы пишем сложный код с использованием `callback`-ов.
Однако использование `callback`-ов для работы с асинхронным кодом — широко используемый подход, и я рекомендую вам его применять в двух случаях:

- если вам нужно написать высокопроизводительный код,
- пишете код какой-то внешней библиотеки.

`callback`-и являются стандартом де-факто работы с асинхронным кодом.

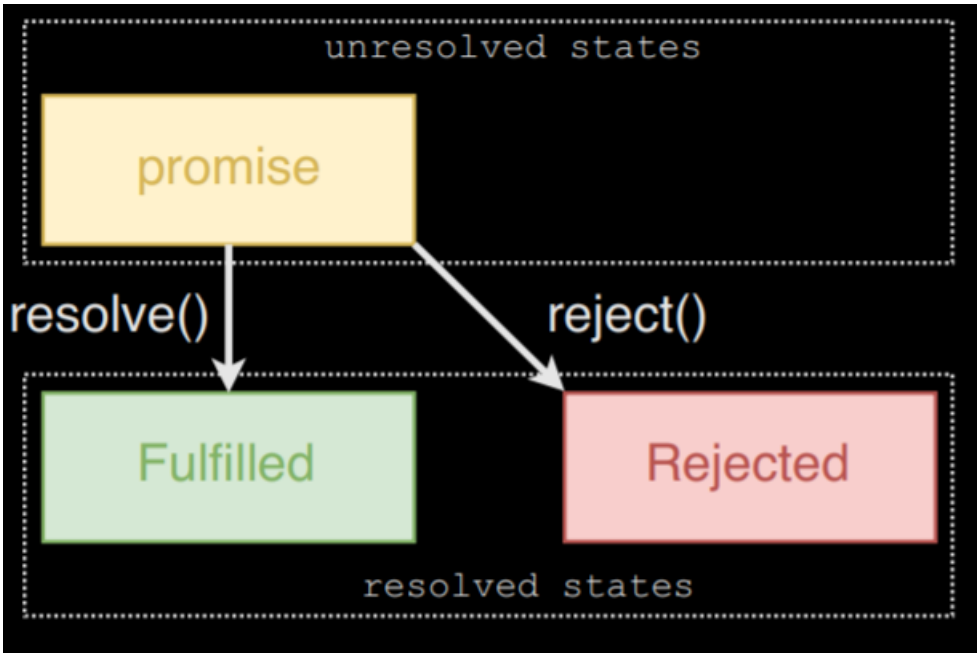
3.5. Промисы

В этом видео мы поговорим о еще одном способе работы с асинхронным кодом, который называется `Promises`. Чтобы создать promise, нам нужно вызвать конструктор `promise` и передать первым аргументом функцию. Внутри этой функции будет содержаться работа с асинхронным кодом. Если чтение файла завершилось с ошибкой, то мы вызываем функцию `reject`, куда передаем информацию об ошибке. Если чтение файла завершилось удачно, то мы

вызываем функцию `resolve`. В качестве параметров в функцию `resolve` мы передаем содержимое файла.

```
var promise = new Promise(function (resolve, reject) {
    fs.readFile('data.json', function (err, data) {
        if (err) {
            reject(err);
        } else {
            resolve(data);
        }
    });
});
```

Как только мы создали `promise`, он находится в состоянии pending — неопределенное состояние. Если асинхронная операция завершилась хорошо, то есть позвали метод `resolve`, то `promise` переходит в состояние fulfilled. Если во время выполнения асинхронной операции произошла ошибка или позвали метод `reject`, то `promise` переходит в состояние rejected. Оба эти состояния являются конечными.



Чтобы получить результат работы `promise`, нам необходимо навесить обработчики: метод `then`. Первым аргументов `then` получает функцию, которая будет вызвана в случае, если `promise` завершился успешно. Если во время выполнения `promise` произошла ошибка, то вызовется функция, которая передается вторым аргументом.

```
promise.then(function (data) {
        console.log(data)
    }, function (err) {
        console.error(err);
    });
```

Недостатки:

- нам приходится писать больше кода, поскольку появляются функции-обработчики;
- производительность `promise` немного ниже, чем производительность `callback`.

Преимущества

Самое главное преимущество заключается в том, что мы отловим неконтролируемые исключения.
Если во время чтения файла произошла неконтролируемая ошибка, мы это можем проиллюстрировать, вызвав ее самостоятельно при помощи метода `throw`, то эту ошибку мы поймаем в обработчике, который передается вторым параметром в метод `then`. И мы увидим на консоли сообщение об ошибке.

```
var promise = new Promise(function (resolve, reject) {
    fs.readFile('data.json', function (err, data) {
        if (err) {
            reject(err);
        } else {
            throw new Error('Mu-ha-ha!');
        }
    });
});
```

```
promise.then(console.log, console.error); //[Error: Mu-ha-ha!]
```

Еще одно преимущество заключается в том, что мы можем навесить несколько обработчиков. Это называется цепочка `promise`.

3.6. Цепочки промисов

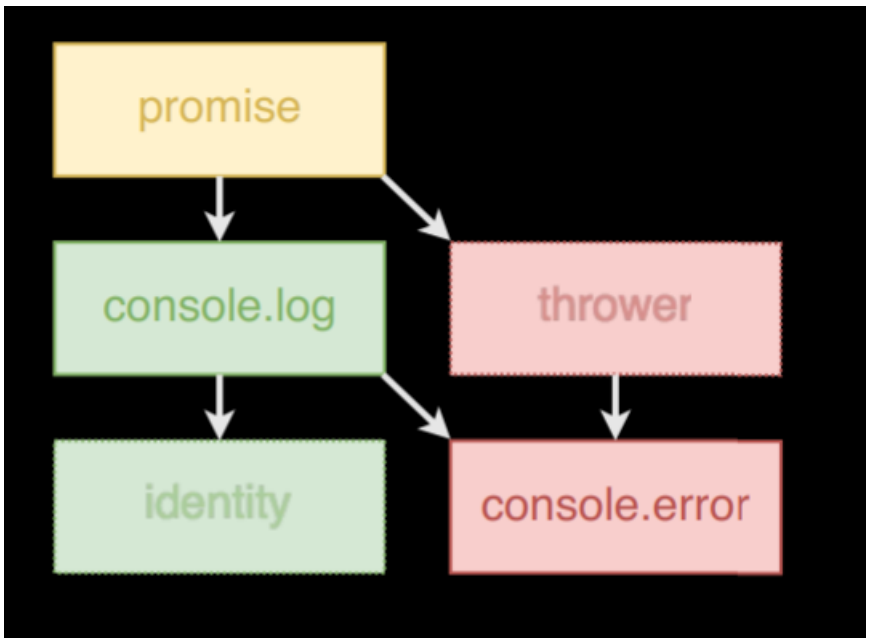
Чтобы лучше понять, как работает цепочка промисов, давайте рассмотрим две следующие функции. Первая функция — `identity` возвращает результат, который мы передаем ей без изменений. Вторая функция — `thrower`. Она выбрасывает исключения с теми данными, которые мы передаем ей первым аргументом.

```
function identity(data) {
    return data;
}

function thrower(err) {
    throw err;
}
```

Мы можем переписать наш пример следующим образом. Вызываем метод `then` у нашего промиса, который читает файл, и первым аргументом передаем `console.log`: выводим содержимое файла на консоль. В качестве второго аргумента мы указываем функцию `thrower`: если при чтении файла произошла ошибка, функция `thrower` пробросит ее дальше по цепочке промисов. После того как мы вызвали метод `then` первый раз, создается новый промис, у которого мы также можем позвать метод `then`. В качестве первого аргумента мы указываем метод `identity`. В качестве второго аргумента мы указываем `console.log`.

```
promise
    .then(console.log, thrower)
    .then(identity, console.error);
```

Полученную цепочку промисов мы можем изобразить на схеме следующим образом.



Из нее становится понятно, что на консоли ошибка окажется не только в случае неудачного чтения файла, но и в случае неконтролируемых исключений обработчика.
В промисе мы будем асинхронно читать некоторые файлы.

```
promise
.then(JSON.parse, thrower)
.then(identity, getDefault)
.then(getAvatar, thrower)
.then(identity, console.error);
```

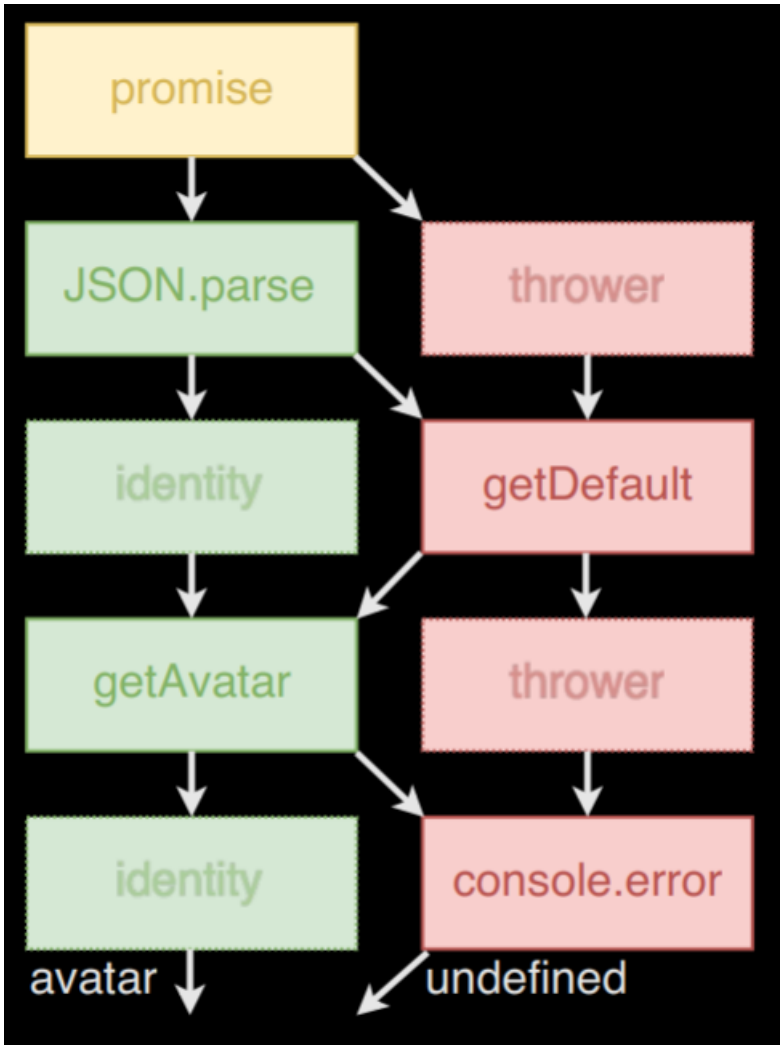
Если чтение файла завершилось хорошо, мы вызовем функцию `JSON.parse`, Эта функция преобразует содержимое файла в JSON. Если чтение завершилось с ошибкой, мы вызываем функцию `thrower`. Эта функция опрокидывает ошибку далее по цепочке промисов. Если парсинг завершился хорошо, то мы попадаем в метод `identity`, то есть прокидываем полученные данные дальше по цепочке промисов как есть. А если при чтении файла или при парсинге данных произошла ошибка, мы вызываем метод `getDefault`.

```
function getDefault() {
    return { name: 'Sergey' };
}
```

Это метод возвращает некоторый JSON, который позволит нам дальше работать с данными.
Если в цепочке промисов встречается обработчик, который ответственен за ошибку, и мы в него попадаем, то цепочка промисов переходит из `rejected` в состояние `fulfilled`. То есть далее мы попадем в метод `getAvatar`.

```
function getAvatar (data) {
    var name = data.name;
    return request('https://my.avatar/' + name);
}
```

Этот `getAvatar` делает асинхронный запрос на удаленный сервер за аватаром пользователя. В качестве второго обработчика ошибки, мы передаем метод `thrower`.
Замыкает цепочку промисов следующая пара. Первым аргументом мы передаем метод `identity` — он возвращает результат, который получили с удаленного сервера. А в качестве второго колбека мы указываем функцию `console error`: выводит на консоль ошибку, которая получилась при запросе на удаленный сервер.
Схематично эту цепочку промисов мы можем изобразить следующим образом.



Из нее: если какая-то асинхронная операция завершилась с ошибкой, то мы, обработав эту ошибку, можем создать новый промис, который уже находится в состоянии `fulfilled`.
Правила, по которым осуществляются переходы по цепочке промисов:

- если очередной обработчик, который мы навесили в методе `then`, возвращает данные, то мы передаем эти данные дальше по цепочке промисов как есть;
- если в `then` возвращается промис, то мы дожидаемся, пока выполнится этот промис, и далее передаем уже результат его работы;

- если в обработчике происходят неконтролируемые исключения, то промис переходит в состояние `rejected`.

Полученный код мы можем записать еще короче. Мы можем убрать все функции `thrower` и `identity` по следующим правилам. Функции `thrower` мы просто убираем, а все методы `then`, которые принимают первым аргументом функцию `identity`, мы заменяем на метод `catch`. Таким образом, наш код выглядит вот так.

```
promise
    .then(JSON.parse)
    .catch(getDefault)
    .then(getAvatar)
    .catch(console.error)
```

3.6.1. Параллельное выполнение промисов.

Реализуем функцию `readFile`, которая на вход принимает путь до файла и возвращает новый промис, который читает файл. Для того чтобы запустить чтение двух файлов параллельно, нам нужно вызвать метод `promise.All` с массивом промисов. Если оба файла прочитаются успешно, то мы вызываем обработчик, который навесили при помощи метода `then`, где получим массив из двух элементов, которые содержат первый и второй файлы соответственно.

```
function readFile(name) {
return new Promise(function (resolve, reject) {
    fs.readFile(name, function (err, data) {
        err ? reject(err) : resolve(data);
        });
    });
}

Promise
    .all([
        readFile('data.json'),
        readFile('ext.json')
    ])
    .then(function (data) {
        console.log(data[0] + data[1])
    });
```

Для того чтобы создать промисы без совершения асинхронных операций, мы можем позвать метод `promise.resolve`. Этот метод создаст нам новый промис, который переходит в состояние `fulfiled` с теми данными, которые мы передали в качестве аргумента в метод `resolve`.

```
Promise
    .resolve('{"name": "Sergey"}')
    .then(console.log); // '{"name": "Sergey"}'
```

Парный к нему метод `reject` создает промис, который находится в состоянии `rejected` с той ошибкой, которую мы передали в метод `reject`.

```
Promise
    .reject(new Error('Mu-ha-ha!'))
    .catch(console.error);
```

Node.js

Курс: JavaScript, часть 2: прототипы и асинхронность

4 апреля 2018 г.

Оглавление

Глава 4

Node.js

4.1. Блокирующий ввод/вывод

Любая программа потребляет те или иные виды ресурсов в том или ином объеме. Например, если ваша программа выполняет какие-то сложные вычисления, она в значительной степени потребляет время центрального процессора. Возможно, она хочет хранить какие-то промежуточные вычисления в оперативной памяти, таким образом потребляя и этот ресурс. А, может быть, для вычислений необходимо считать какой-то массив данных с диска или, например, с удаленного хранилища. В этом случае она обращается к системам ввода-вывода для чтения с локального диска или для обращения по сети.
Например, вычисления числа Фибоначчи. Эта операция, в основном, потребляет время центрального процессора. Мы можем сказать, что она ограничена производительностью этого процессора, и если мы поставим процессор мощнее, мы сможем вычислить это число быстрее.
Второй пример: подсчет количества строк в файле. Здесь уже не так много вычислений, нужно лишь увеличивать счетчик. И большую часть времени данная операция будет потреблять ресурсы систем ввода-вывода, ведь файл необходимо вначале прочитать, а потом подсчитать количество строк в нем.
Посмотрим на основные операции, из которых состоит работа веб-сервера:

- прочитать запрос от пользователя: операция, которая потребляет ресурсы системы ввода-вывода;
- получив запрос, мы должны его разобрать: вычисления, потребляем время центрального процессора;

- для подготовки ответа для пользователя нам необходимо сходить в базу данных или сделать запрос какому-то удаленному API: ресурсы системы ввода-вывода;
- на основе этих данных генерируем HTML: вычисления;
- отправляем этот полученный HTML пользователю: ввод-вывод.

Наш веб-сервер запускается операционной системой в потоке. В рамках одного потока может выполняться одновременно только одна операция, независимо от того, вычислительная она, или, например, мы читаем файл с диска. Для обращения к системам ввода-вывода приложение активно взаимодействует с операционной системой. И по умолчанию данное взаимодействие — **блокирующее**.
Например, мы выполняем вычисления, и для ответа нам понадобилось прочитать какие-то данные из локального файла. Мы делаем запрос в операционную систему и начинаем ждать данные из этого файла. Так как ввод-вывод у нас блокирующий, на это время наше приложение замирает и не может больше выполнять никаких операций. В этот самый момент к нам может прийти второй пользователь, и мы не сможем его обслужить. Таким образом, мы для каждой операции ввода-вывода блокируем поток выполнения нашей программы.
Блокирующий ввод-вывод может накладывать существенные ограничения на производительность нашего веб-сервера. Допустим, мы читаем 1 Кб данных с твердотельного накопителя. Это занимает 0,0014 миллисекунды. За это время стандартное ядро двухгигагерцового процессора позволяет выполнить 28 тысяч циклов, то есть примерно 28 тысяч операци. Но пока мы ждем этот килобайт данных, наш поток приложения блокируется и мы не можем выполнить эти 28 тысяч циклов.

4.2. Многопоточность и неблокирующий ввод/-вывод

4.2.1. Многопоточность

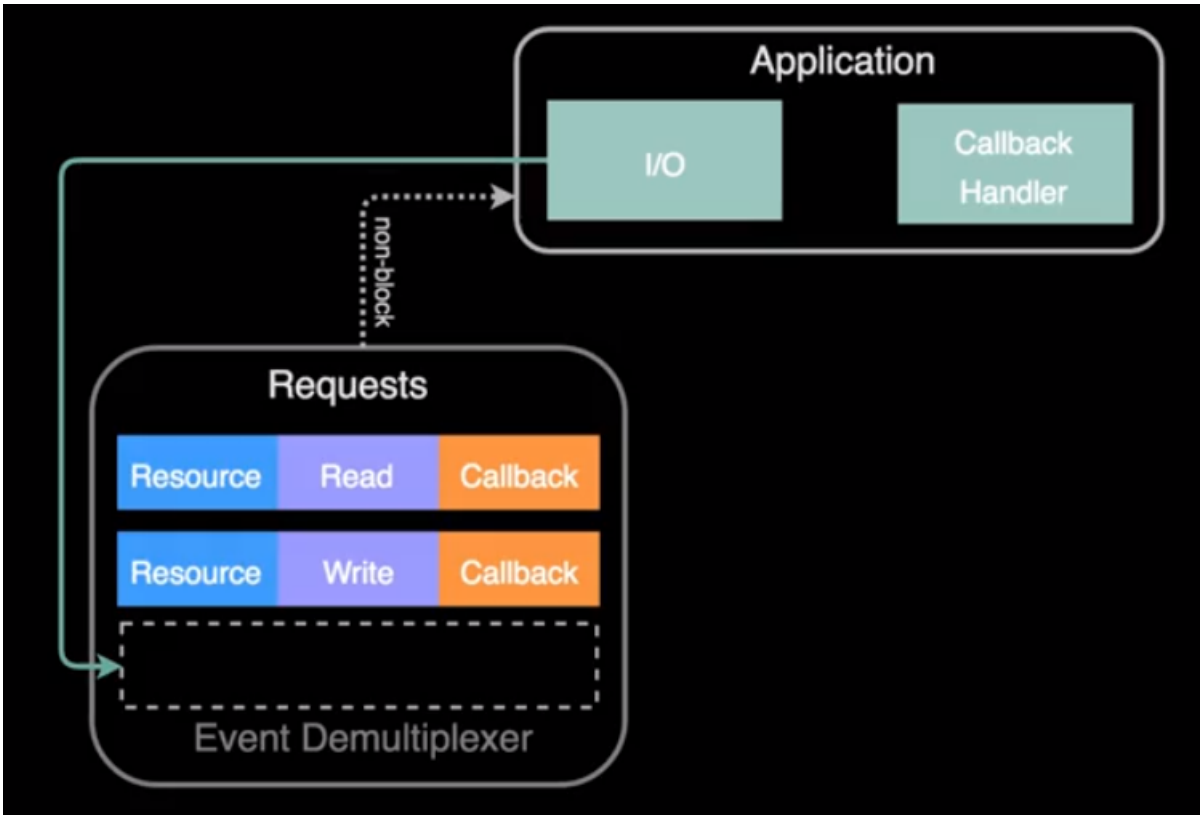
Мы поднимаем еще один экземпляр нашего приложения в отдельном потоке. И пока первый поток занят ожиданием операции ввода/вывода, второй поток в этот момент может без ограничений обслужить следующего пользователя.
Ограничения метода:

- поднятие потока — это не бесплатная операция, но это несущественно, т.к. можем использовать пул подготовленных потоков;
- есть определенный лимит на количество поднимаемых потоков в операционной системе, и если количество пользователей, которое нам необходимо обслужить, будет заметно превышать этот лимит, мы снова столкнемся с проблемами с производительностью;
- каждый поток требует дополнительной памяти для своей работы.

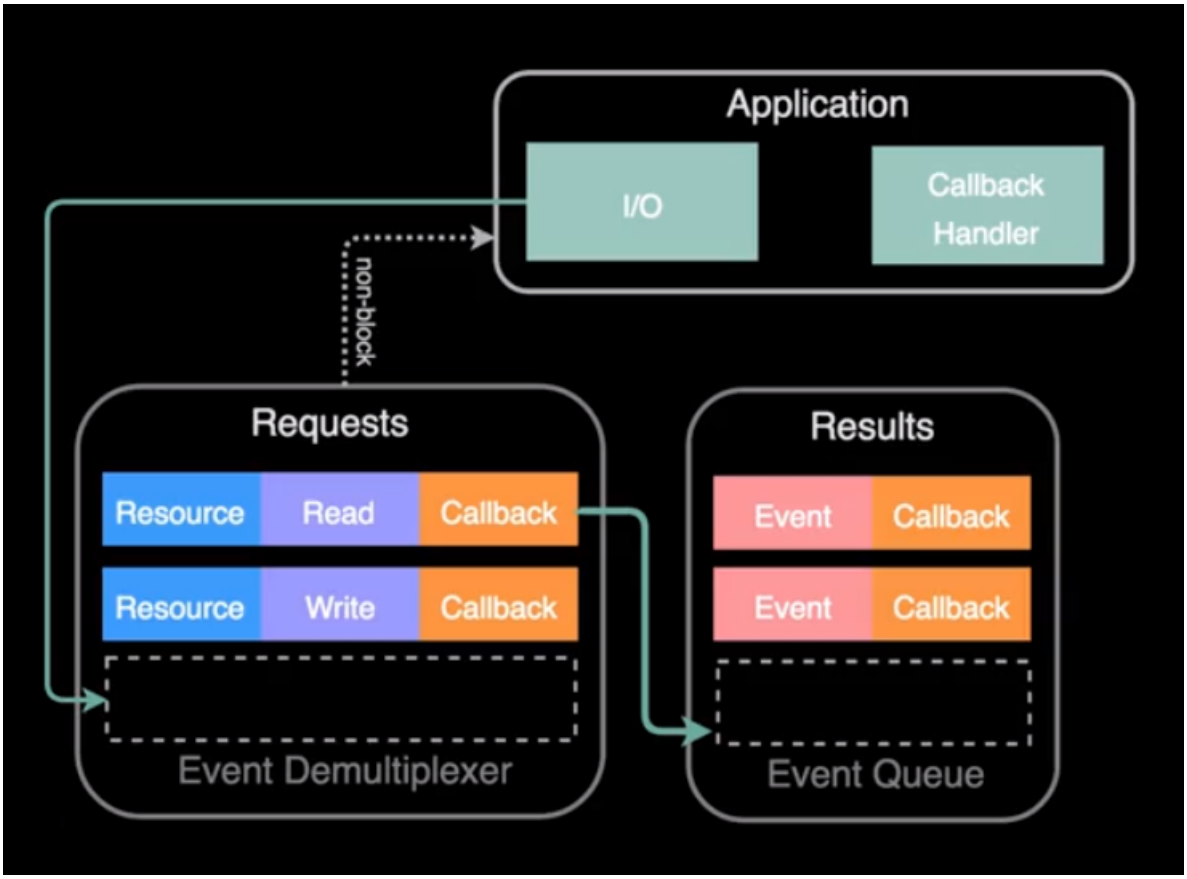
.

4.2.2. Паттерн `reactor`

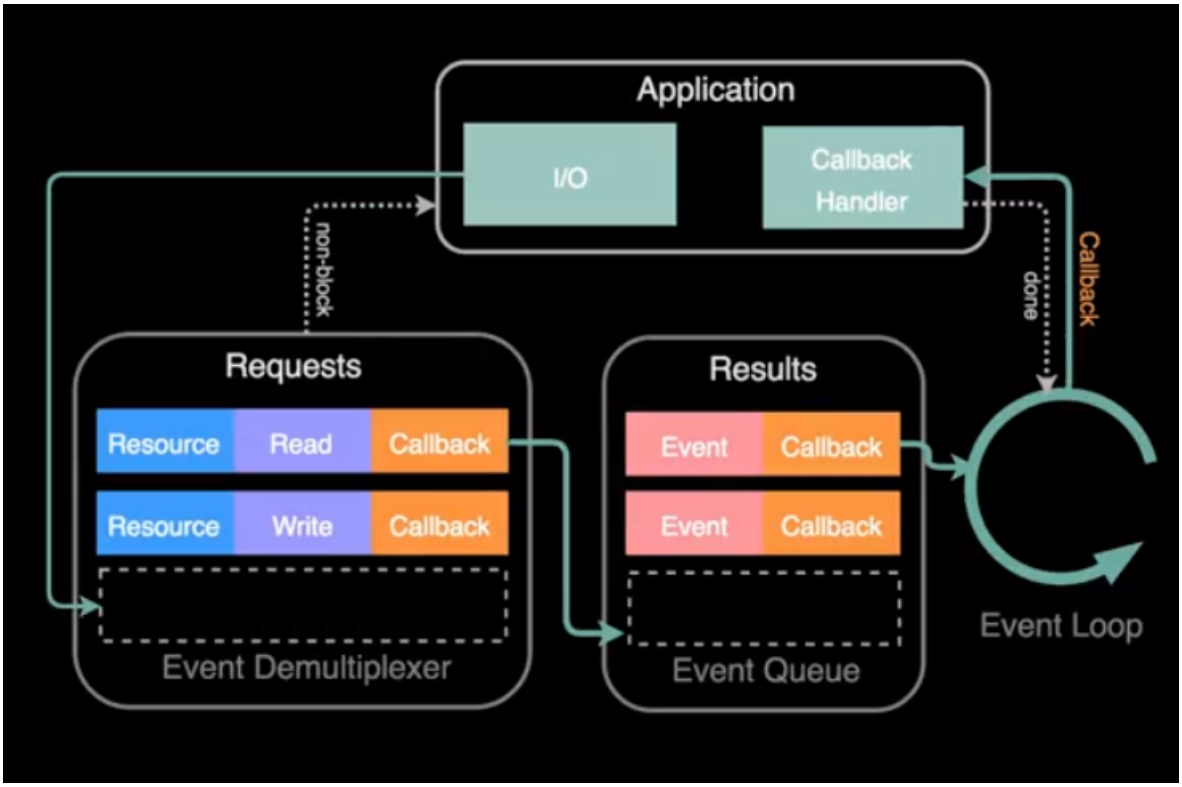
Как только мы запрашиваем какой-то ресурс, например, локальный файл или пытаемся сделать запрос по сети, мы делаем это со специальным флагом. Как только мы делаем запрос, операционная система его регистрирует и в тот же самый момент возвращает управление в поток нашего приложения. И наше приложение может заниматься другой полезной работой. Как только операционная система подготовит ресурс для обработки, она уведомит нас об этом, и наше приложение получит это уведомление и может с этим файлом что-то сделать.
Давайте подробнее рассмотрим, как наше приложение работает с неблокирующей системой ввода/вывода. Например, для продолжения работы нашего приложения ей необходимо прочитать некоторый файл. Для этого она регистрирует необходимый ресурс и операцию над ним в специальном механизме ОС — демультиплексере событий.



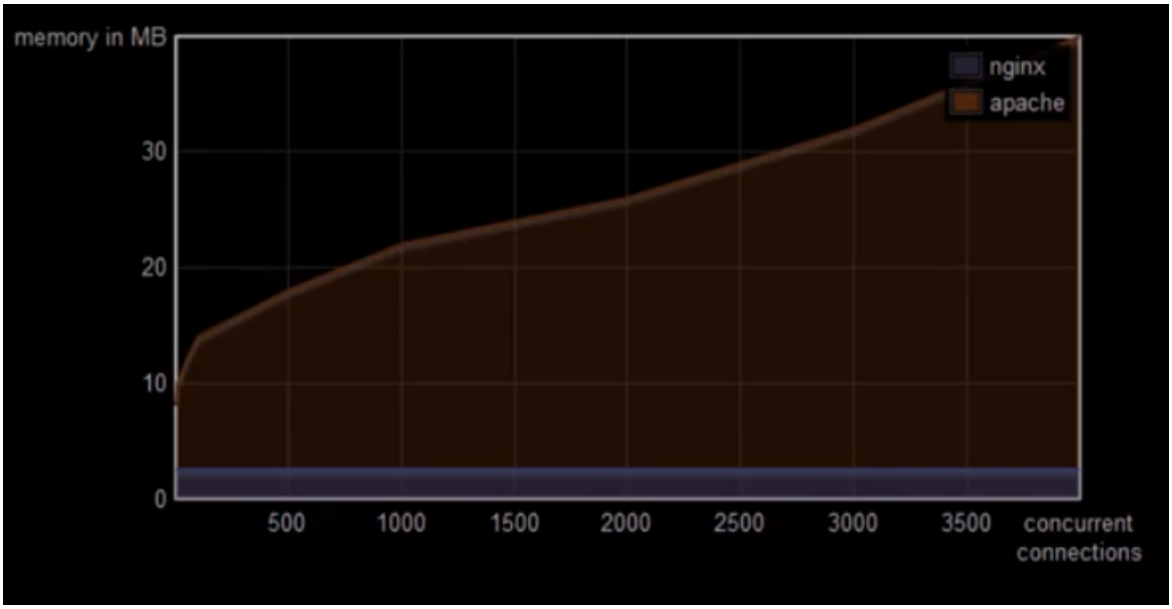
Помимо этого, она в запросе указывает ссылку на обработчик/`callback`, который необходимо выполнить, как только ресурс будет готов. После регистрации запроса на ресурс управление передается обратно в приложение, и наше приложение не блокируется и может продолжать заниматься полезными делами. Как только ресурс будет готов, демультиплексер событий положит в очередь событий сообщение о том, что он готов. С каждым событием будет связан обработчик, который необходимо вызвать.

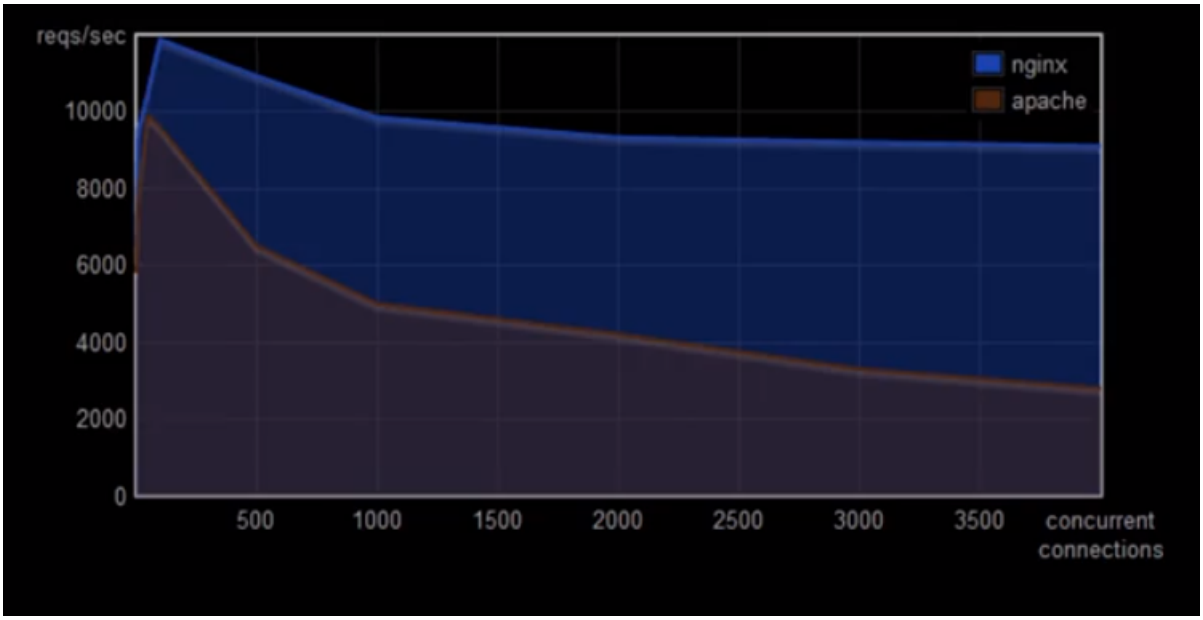


Очередь событий шаг за шагом разбирает в бесконечном цикле обработчик событий. Он вызывает обработчики, они выполняются на стороне приложения, и управление обратно возвращается обработчику событий, он берет следующее событие, вызывает следующий обработчик. В момент выполнения обработчика на стороне приложения может возникнуть потребность запросить еще ресурсов, и тогда мы снова пойдем по кругу.



Весь этот набор механизмов и взаимодействий описан в паттерне `reactor`. Мы рассмотрели два способа решения проблемы блокирующего ввода/вывода. Сравнение двух этих подходов наиболее ярко прослеживается в противостоянии двух известных серверов: Apache и Nginx. Изначально Apache использовал первый подход, а именно многопоточность: на каждый запрос он поднимал отдельный поток. В то же самое время Nginx использовал уже паттерн `reactor` для обработки входящих запросов.

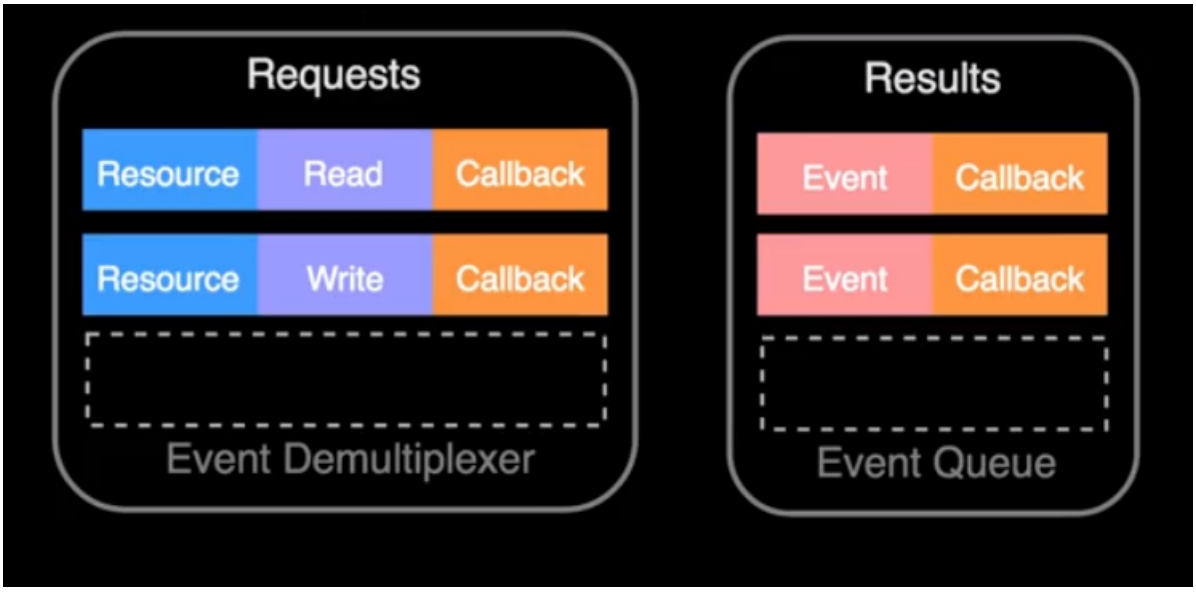




В то время, как для обработки все большего количества одновременных запросов серверу Apache понадобилось все больше и больше памяти, потребление памяти сервером Nginx с увеличением количества одновременных запросов оставалось на одном уровне.

4.3. Архитектура Node.js

Преимущество Nginx над Apache вдохновило разработчика Райана Дала на создание платформы Node.js.
В первую очередь, Райану Далу необходимо было подружить свою программную платформу с демультиплексером событий, но в разных операционных системах данный механизм реализован по-разному. И Райан Дал написал небольшую обертку libuv над тремя реализациями этих систем.
Здесь стоит сделать паузу и подробнее рассмотреть особенности этой библиотеки. Несмотря на то, что все современные операционные системы предоставляют нам неблокирующий интерфейс для работы с вводом-выводом, это не всегда так. Например, в Linux-системах, операции над локальными файлами до сих пор всегда блокирующие, в отличие от тех же сетевых операций. Поэтому для эмуляции неблокирующего поведения libuv под капотом использует первый подход многопоточности для решения этой проблемы.
По умолчанию, эта библиотека поднимает пул из четырех потоков, но этот параметр мы можем поменять. Это означает, что четыре долгих операции над локальными файлами заблокируют нам всё приложение. И об этом важно помнить.
Если вам захочется узнать подробнее об этой библиотеке я рекомендую вам начать с видео Берта Белдера. Далее, по ссылке1 и ссылке2, вы можете познакомиться с другими нюансами в работе с этой библиотекой.
Итак, для того чтобы ваша программа могла взаимодействовать с механизмами неблокирующего ввода-вывода при регистрации запроса на какой-либо ресурс, она должна указывать еще и ссылку на обработчик результата `Callback`, который будет вызван, как только ресурс будет готов.



Для этого нашу программу в синхронном стиле мы должны написать в асинхронном: вызовы синхронных методов заменить на асинхронные, и для обработки результатов передавать дополнительно в качестве последнего аргумента функцию «Обработчик». Этот паттерн называется паттерном `callback` и он следует двум соглашениям:

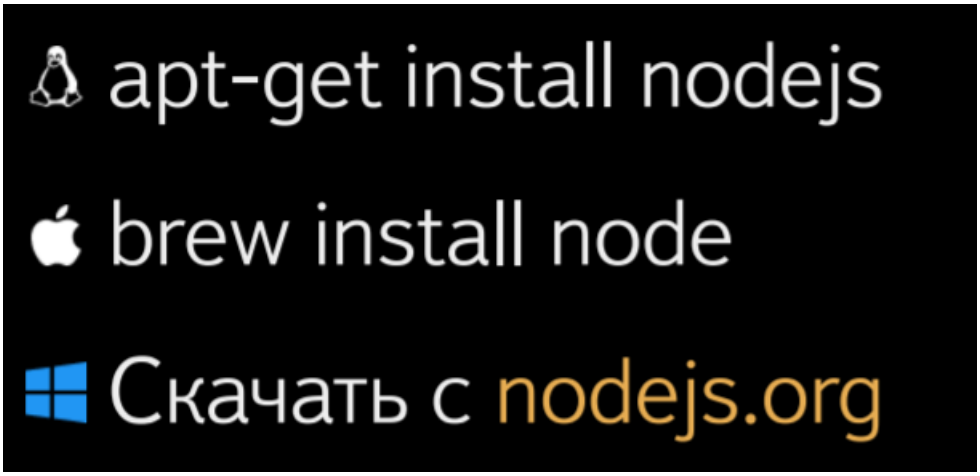
- они всегда идут последним аргументом;
- если мы не смогли получить какой-то ресурс, ошибка об этом должна приходить в обработчик в качестве первого аргумента.

```
var data = request('https://api.github.com/');
var result = writeFile(file, data);
console.info(result);

request('https://api.github.com/', function (err, data) {
    writeFile(file, data, function (err, result) {
        console.info(result);
    });
}
```

Для написания таких программ как нельзя лучше подходит язык программирования JavaScript:

- данный язык из коробки предлагает нам функции первого класса и замыкание, а это значит, что мы можем использовать функции в качестве аргументов;
- язык уже готов к работе с EventLoop — обработчиком событий, благодаря тому, что он прекрасно взаимодействовал с этим механизмом в браузерах;
- язык имеет большое комьюнити.

В качестве интерпретатора был выбран движок V8 от компании Google как наиболее быстрый на тот момент.
Существует API для работы с файловой системой, для осуществления запросов по http и для логирования. Всё это называется Core JavaScript API. Но JS — это open-source разработка, соответственно, весь исходный код располагается на github.
Всё, что осталось сделать — это объединить библиотеку LIBUV с интерпретатором V8 и с удобным JavaScript API для разработчиков. И это стало возможным благодаря написанию bindings, которые позволили связать код, написанный на C++ и C с кодом, написанным на JavaScript. Все эти библиотеки и механизмы были связаны в единую платформу, которая получила название Node.js.
Я рекомендую вам ознакомиться с оригинальной презентацией этой платформы от Райана Дала, а также по второй и третьей ссылке вы можете изучить более подробно архитектуру этой платформы, познакомиться с тем, как она работает с обработчиком событий.
Начать работу с этой платформой очень просто: достаточно её установить.



На текущий момент разработчики node.js поддерживают две ветки версии: чётные версии — это версии с длительной поддержкой. Вы можете использовать приложение, написанное на платформе этой версии в бою без каких-либо опасений. Нечётные версии — это нестабильные версии, но они включают, как правило, все самые новые возможности.

4.4. Модули

> **Модуль — это текстовый файл, в котором написан код на языке JavaScript. Все Node.js-приложения состоят из набора таких модулей. Если какой-либо модуль реиспользуется между разными приложениями, такие модули называют пакетами.**

В основе синтаксиса этих модулей лежит спецификация, разработанная проектом CommonJS, и по ссылке вы с ней сможете подробней ознакомиться.
Попробуем создать простой модуль для вычисления гипотенузы. В нем есть основная функция и дополнительная для вычисления квадрата числа.

```
1 // hypotenuse.js
2 function square(n) {
3     return n * n;
4 }
5 function calculateHypo(a, b) {
6     return Math.sqrt(square(a) + square(b));
7 }
```

Чтобы наш модуль стал полезным, нам необходимо экспортировать из него основную функцию. Для этого Node.js, перед тем как интерпретировать код нашего модуля, добавляет в него специальный объект с метаинформацией. Помимо прочего в нем содержится, например, полный путь до файла, где этот модуль описан, а также специальное поле exports, которое представляет собой объект. Если мы в этот объект поместим нашу функцию для вычисления гипотенузы, она будет доступна для других модулей.

```
// hypotenuse.js
// module = {
// filename: '/absolute/path/to/hypotenuse.js'
,
// exports: {}
// }
function square(n) {
    return n * n;
}
module.exports.calculate = function (a, b) {
    return Math.sqrt(square(a) + square(b));
}
// return module.exports;
```

Написав простой модуль и экспортировав из него функцию вычисления гипотенузы, мы можем импортировать этот модуль в другом файле при помощи специальной функции `require`.

```
// index.js
var hypotenuse = require('./hypotenuse.js')
hypotenuse.calculate(3, 4); // 5
```

Мы можем экспортировать не только объекты, но и функции. Так, в нашем модуле для вычисления гипотенузы у нас есть одна основная функция, и мы можем сразу экспортировать ее, поместив в специальное поле `exports`.

```
// hypotenuse.js
function square(n) {
    return n * n;
}
module.exports = function (a, b) {
    return Math.sqrt(square(a) + square(b));
}
// index.js
var hypotenuse = require('./hypotenuse.js');
hypotenuse(3, 4); // 5
```

Мы не ограничены экспортом только функции или объектов, и можем экспортировать другие типы данных, например число или конструктор, или экспортировать объект, созданный на основе этого конструктора.

```
module.exports = 42; // Число

function Student(name) {
    this.name = name;
}
Student.prototype.getName = function() {
    return this.name;
};
module.exports = Student; // Конструктор

module.exports = new Student('Billy'); // Объект
```

Существует более лаконичная форма записи экспорта. Так, наряду со специальным объектом модуль, в котором есть поле `exports`, Node.js создает специальную переменную, которая хранит ссылку на это поле, и мы можем записать наш экспорт чуть лаконичнее.

```
// hypotenuse.js
function square(n) {
    return n * n;
}

exports.calculate = function (a, b) {
    return Math.sqrt(square(a) + square(b));
}
```

Но с этой возможностью стоит обращаться немного с осторожностью, так как если мы попробуем записать в эту переменную функцию, то с удивлением обнаружим, что никакую функцию мы на самом деле не экспортировали, так как Node.js возвращает всегда то, что находится в поле exports объекта module. А в данном случае мы просто перезаписали нашу переменную exports, и связь между этой переменной и полем exports просто потерялась.

```
// hypotenuse.js
function square(n) {}
exports = function (a, b) {
return Math.sqrt(square(a) + square(b));
}
// index.js
var hypotenuse = require('./hypotenuse.js');
hypotenuse(3, 4); // hypotenuse is not a function
```

Помимо собственных модулей мы можем импортировать встроенные в Node.js, и так их достаточно много. Например, мы можем импортировать модуль url. Этот модуль предоставляет ряд функций для работы с адресами. Так, мы можем разобрать адресно составные части при помощи функции parse и получить объект с полями, каждое из которых хранит какую-либо из частей адреса.

```
var url = require('url');
url.parse('https://yandex.ru/');
{

    protocol: 'https:',
    host: 'yandex.ru',
    port: null,
    path: '/'

}
```

Если встроенных в Node.js модулей будет недостаточно, мы можем импортировать модули других разработчиков. Для этого мы воспользуемся той же самой функцией `require`, но передадим туда уже идентификатор модуля, который для нее обозначил сторонний разработчик. Так мы можем импортировать самый популярный на данный момент модуль `lodash`. Данный модуль предоставляет несколько полезных методов для работы, например с объектами, или массивами.

```
var lodash = require('lodash');

lodash.shuffle([1, 2, 3, 4]);
// [4, 1, 3, 2]

lodash.uniq([2, 1, 2]);
// [2, 1]
```

Давайте рассмотрим важную особенность импорта модулей. Итак, допустим, у нас есть модуль, в котором есть счетчик (`counter`), и его начальное значение 1. Мы экспортируем из этого модуля функцию, которая ничего не делает кроме того, как увеличивает этот счетчик на 1.

```
var counter = 1;
module.exports = function() {
    return counter++;
};
```

Далее мы попробуем использовать этот модуль, но подключим его дважды, создадим два счетчика: `counter` и `anotherCounter`. Попробуем вызвать вначале первый счетчик. Вызвав первый счетчик, мы получим его начальное значение, и затем оно увеличивается на 1. Вызвав его второй раз, мы получим увеличенное на 1 значение — 2.

```
var counter = require('./counter');
var anotherCounter = require('./counter');
console.info(counter()); // 1
console.info(counter()); // 2
```

Что произойдет, если мы попробуем вызвать другой счетчик? Отсчет продолжится. Почему это так? На самом деле модули импортируются один раз, и после того как функция `require` отработала, экспорт кешируется. Кешируется он в специальном свойстве этой функции — `cache`. Выглядит она примерно так.

```
{
'/absolute/path/to/filename.js': {
filename: '...'
,
exports: {},
}
}
```

Это простой объект, в качестве ключей которых абсолютный путь до файла, где расположен модуль. А далее объект, описывающий этот модуль, включая специальное поле `exports`, где хранятся все экспортированные функции.
Каким же образом Node.js понимает, какой из типов модулей необходимо импортировать прямо сейчас?

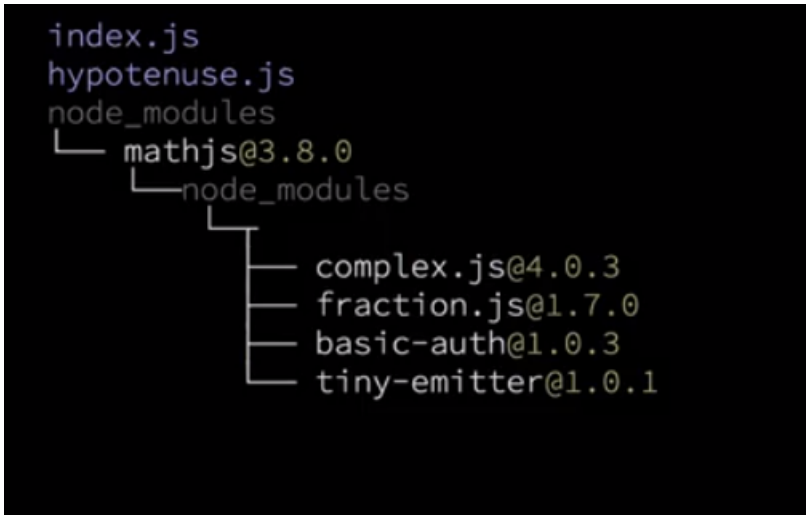
- вначале она смотрит, есть ли встроенный модуль с тем именем, который мы передали в функцию `require`: если есть, импортирует его;
- если в функцию `require` мы передали путь до модуля, то импортируется модуль по указанному пути;
- в противном случае модуль или пакет ищется в специальной директории `node_modules`, начиная с текущей директории, где находится файл с нашим исходным кодом.

Полный алгоритм несколько сложнее, и вы сможете ознакомиться с ним по ссылке.

4.5. Пакетный менеджер NPM

Чтобы находить и устанавливать модули, вместе с Node.js устанавливается специальный инструмент командной строки. Именно он и позволяет находить и устанавливать модули. Этот инструмент взаимодействует с глобальным хранилищем модулей. Вы можете посмотреть информацию об этом хранилище через веб-интерфейс здесь.
Первая команда, которая вам понадобится — команда `init`. После ее запуска она задаст вам несколько простых вопросов: как будет называться ваш модуль, кто ее автор и т.д. После этого она создаст специальный файл-манифест с названием `package` и с расширением json, который будет описывать ваш

модуль согласно вашим ответам. Можно сказать, что модуль плюс этот файл-манифест и есть используемый пакет.
Допустим, нашему приложению необходим пакет с математическими функциями. Чтобы найти такой пакет, мы можем воспользоваться следующей командой — командой `search`. Она ищет пакет в хранилище по имени и находит все подходящие и выводит в виде списка.
Альтернативный способ поиска пакетов — использование веб-интерфейса. Вы можете воспользоваться официальным сайтом этого пакетного менеджера npmjs.com или его альтернативной версией npms.io. Выбрав подходящий пакет, вы можете посмотреть информацию о нем при помощи команды `show`.
Если вас пакет устраивает, можете установить его при помощи команды `install`, передав ей имя пакета. Данная команда устанавливает пакет в отдельную директорию в качестве зависимости и помещает ее в директорию node_modules на уровне ваших файлов. Если у этой зависимости есть подзависимости, они установятся в ту же самую директорию, но уже у зависимости. Это легко представить в виде дерева файлов.



Если мы посмотрим на результат выполнения этой команды, то увидим, что на уровне наших файлов создалась директория node_modules, внутри которой есть директория с нашей зависимостью. А внутри директории с зависимостью создалась еще одна папка node_modules, внутри которой есть подзависимости для нашей зависимости.
Если ваше приложение совместимо с определенной версией зависимости, вы можете указать ее дополнительно в команде install. Разработчикам пакетов

рекомендуется следовать специальному соглашению о версионировании своих пакетов. Это соглашение предлагает следующий формат — разбивать версию пакета на три: мажорную (новые возможности без обратной совместимости с предыдущими версиями), минорную (новые возможности + обратная совместимость) и патчевую (исправление ошибок или рефакторинг), и разделять эти три числа точками.
Мы можем зафиксировать нашу зависимость в файле-манифесте package.json, передав в команду `install` специальный флаг `save`.
Первая группа зависимостей — та, что требуется для работы нашего приложения напрямую, а вторая группа зависимостей — которая необходима для тестирования нашего пакета, сборки и другого типа обслуживания. Чтобы установить пакет и записать его во вторую группу, необходимо передать другой флаг в команду `install`, а именно `save-dev`. Зависимости фиксируются в специальный раздел файла-манифеста "dependencies". Мы можем зафиксировать там весь набор зависимостей и, более того, заполнить этот раздел вручную. И тогда установка всего набора будет проходить очень просто: `install` без каких-либо параметров, и она установит за нас все зависимости.
Зафиксировать зависимость мы можем с разной версией, указывая ее в разном формате. Самый простой формат — просто указать версию. Если мы хотим, чтобы при установке нашей зависимости `npm` всегда брал самую свежую версию из определенного диапазона, мы можем указать их в виде диапазона так или таким образом.

```
{

"dependencies": {
"express": "1.2.3",
"express": ">1.2.3",
"express": ">=1.2.3",
}
```

Если мы готовы автоматически устанавливать только свежие патч-версии, но не готовы автоматически устанавливать свежие минор-версии нашей зависимости, тогда перед версией мы добавляем значок «тильда».
Если же мы готовы устанавливать и свежие, и минорные версии — добавляем другой значок. Вместо одной из версий мы можем указать «звездочку» или x, что означает «любое число». Более того, вместо версии мы можем указывать

определенные теги. Например, тег `latest` означает, что мы хотим устанавливать всегда самую последнюю версию этой зависимости.

```
{
"dependencies": {
"express": "~1.2.3", // >=1.2.3 <1.3.0
"express": "^1.2.3", // >=1.2.3 <2.0.0
"express": "1.2.*",
"express": "latest",
}
```

Более того, `npm` умеет устанавливать зависимости не только из глобального хранилища, но и например с github. Для этого мы в качестве зависимости указываем специальный git URL. Также мы можем указать тег или даже ветку, а может быть и конкретный коммит.

```
{
"dependencies": {
"express": "git://github.com/expressjs/express.git",
"express": "git://github.com/expressjs/express.git#4.13.4",
"express": "git://github.com/expressjs/express.git#master",
"express": "git://github.com/expressjs/express.git#f3d99a4",
}
```

Это далеко не полный набор вариантов, как мы можем фиксировать зависимости. С полным набором вы можете ознакомиться вот по этой ссылке.
Вы можете управлять поведением данной командной строки, записав в файл npmrc ряд опций. Рассмотрим самые полезные из них. Так, вы можете указать опцию `save` со значением `true`, и тогда команда `install` будет всегда фиксировать зависимость в файл-манифест, независимо от того, передали ли вы флаг save или нет. Более того, вы всегда можете фиксировать зависимость строго.

```
save=true // Всегда фиксировать зависимость
save-exact=true // Строго фиксировать версию
```

Я очень рекомендую этот параметр, так как не все разработчики пакетов следуют соглашению о версионировании: даже новая патч-версия вашей зависимости может нарушить работу вашего приложения.
Подробнее прочитать про файл-манифест можно по следующей ссылке, а узнать некоторые трюки и советы по работе с данной командной утилитой — из презентации.

4.6. http-клиент и http-сервер на Node.js

Начнем мы с решения задачи организации веб-сервера: приложения, которое будет принимать запросы от пользователей и отвечать на них. Для решения этой задачи нам понадобится модуль http. Давайте подключим его и попробуем создать объект нашего веб-сервера. Для этого воспользуемся конструктором Server и создадим такой объект. Далее нам необходимо при помощи метода `on` подписать наш сервер на событие `request`, которое будет вызываться каждый раз, когда к нам будет приходить новый запрос от пользователя. Далее нам необходимо запустить наш веб-сервер. Мы сделаем это при помощи вызова метода `listen` и передадим туда номер порта, который будет слушать наш веб-сервер.

```
1  var http = require('http');
2
3  var server = new http.Server();
4
5  server.on('request', function (req, res) {
6      res.end('Hello, User!');
7  });
8
9  server.listen(8080);
```

Как только в наш веб-сервер придет новый запрос от пользователя, произойдет событие `request`, и будет вызван обработчик, связанный с этим событием. В качестве первого аргумента в обработчик придет объект, экземпляр конструктора `IncomingMessage`. Данный объект поможет нам получить информацию о запросе: http-метод, с которым был сделан этот запрос, а также заголовки или путь до ресурса, который мы запрашиваем у веб-сервера.

```
server.on('request', function (req, res) {
    console.info(req.method); // GET

    console.log(req.headers); // {'accept-encoding': 'gzip'}

    console.log(req.url); // /favicon.ico

});
```

В качестве второго аргумента в наш обработчик придет экземпляр другого конструктора, `ServerResponse`. Здесь можем посмотреть текущий статус ответа или поменять его. Также мы можем поменять заголовки ответа и, наконец, отправить в ответе какое-то сообщение при помощи метода `write`. Мы можем вызывать этот метод несколько раз и отправить несколько пачек ответа. Как только мы поймем, что все, что мы хотели отправить пользователю, отправили, мы завершаем наш ответ методом `end`.

```
server.on('request', function (req, res) {
    console.info(res.statusCode); // 200

    res.setHeader('content-type', 'text/html');
    res.write('<strong>Hello!</strong>');
    res.end();

});
```

Данный модуль позволяет легко сделать запрос к нашему веб-серверу и прочитать ответ от него. Для этого мы вновь подключаем этот модуль, но в этот раз конструируем объект запроса при помощи другого метода, метода request и передаем в него параметры запроса. Мы хотим обратиться к веб-серверу, который у нас запущен локально, по тому самому порту, который этот веб-сервер обслуживает.

```
var http = require('http');

var req = http.request({
    hostname: 'localhost',
    port: 8080
});
```

Далее мы подписываем наш объект запроса на событие «ответ»: происходит каждый раз, как только к нам приходит ответ от веб-сервера. Мы назначаем для этого события обработчик и внутри события начинаем собирать ответ по частям, так как веб-сервер нам его отправляет тоже по частям. Каждый раз, когда веб-сервер будет отправлять нам часть ответа при помощи метода `write`, будет вызвано событие `data`, и мы получаем частичку ответа в отдельную переменную `body`. Как только веб-сервер нам даст сигнал о том, что он закончил с ответом, мы получим событие `end`, и в обработчике этого события мы уже можем вывести ответ. Как только мы навесим наш обработчик на объект запроса, мы можем этот запрос выполнить при помощи метода `end`.

```
req.on('response', function (response) {
    var body = '';

    response.on('data', function (chunk) {
        body += chunk; // res.write(chunk);
    });

    response.on('end', function () {
        console.info(body); // res.end();
    });
});
req.end();
```

Мы работаем с нашим веб-сервером в асинхронном стиле, благодаря этому не блокируем поток выполнения всего приложения. Мы подписываем объект запроса на специальное событие `response` — ответ от сервера. Это становится возможным благодаря тому, что объект запроса в своей основе имеет механизм генератора событий или механизм `EventEmitter`. Данный механизм лежит в основе многих модулей платформы Node.js, и мы сможем познакомиться с

ним подробнее, если создадим экземпляр конструктора `EventEmitter`, который лежит в модуле `events`.
Данный механизм позволяет подписывать объекты на события и назначать для этих событий обработчики. В данном случае мы подписываем наш объект на событие `log` и вызываем обработчик, который выводит на экран все, что придет в качестве данных этого события. Далее `EventEmitter` может помочь нам сгенерировать это событие и передать данные, связанные с ним, которые попадут в обработчик.

```
1 var EventEmitter = require('events').EventEmitter;
2 var emitter = new EventEmitter();
3
4 emitter.on('log', console.info);
5 emitter.emit('log','Hello!'); // Hello!
6
7 emitter.emit('unknown event'); // Do nothing
8 emitter.emit('error'); // Uncaught, unspecified "error" event.
```

Если мы попробуем сгенерировать событие, с которым не связан ни один обработчик, Node.js просто проигнорирует. Но из этого правила есть одно исключение. Если мы попробуем сгенерировать событие ошибки, в этом случае Node.js не проигнорирует, а наоборот бросит нам ошибку и скажет, что с этим событием не связан ни один обработчик, как бы подталкивая нас всегда назначать обработчики в случае ошибок.
Следующие два модуля помогают нам наладить работу нашего веб-сервера.
Первый из них — это модуль `url`. Этот модуль позволяет нам работать с url-адресами: например, методом `parse`, который позволяет разобрать url-адрес на составляющие. У него есть парный метод, метод `format`, который наоборот позволяет собрать адрес из составляющих в строку.

```
url.parse('https://yandex.ru/');
// {
//      protocol: 'https:',
//      host: 'yandex.ru',
//      path: '/',
// ...
// }

url.format({
    protocol: 'https:',
    host: 'yandex.ru'
});

https://yandex.ru
```

И последний модуль `querystring`. Данный модуль позволяет разобрать строку, содержащую GET-параметры, в объект с этими параметрами для дальнейшей обработки. Также есть в нем и парный метод, который позволяет этот объект собрать обратно в строку.

```
querystring.parse('foo=bar&arr=a&arr=b');
// {
// foo: 'bar',
// arr: ['a','b']
// }
querystring.stringify({
    foo: 'bar',
    arr: ['a','b']
});
// foo=bar&arr=a&arr=b
```

4.7. Работа с локальной файловой системой

Модуль для работы с локальными файлами носит название fs и, в частности, предлагает нам метод для асинхронного чтения содержимого файлов, не блокируя при этом поток выполнения приложения.

При помощи метода `readFile` мы можем прочитать этот файл, передав в качестве первого аргумента абсолютный или относительный путь до файла. А в качестве второго аргумента — обработчик, который будет вызван, как только операционная система подготовит для нас этот файл и уведомит нас об этом. В данном случае мы попытаемся прочитать текстовый файл с текущим исходным кодом. Для этого передадим в качестве первого аргумента переменную `fileName`, которая хранит абсолютный путь до текущего файла.

```
1  var fs = require('fs');
2
3  fs.readFile(__filename, function (err, content) {
4      console.info(content);
5  });
```

Как только файл будет прочитан, будет вызван обработчик. Если произойдет какая-то ошибка, в качестве первого аргумента мы сможем получить доступ к ней. Если ошибка не возникнет, и файл будет прочитан успешно, то его содержимое придет нам в качестве второго аргумента: объект конструктора `Buffer`, который представляет из себя на экране набор чисел в 16-ричной системе счисления.
Этот объект предназначен для работы с бинарными данными. Его можно рассматривать как массив чисел от 0 до 255, где каждое число представляет собой байт.
Создадим для примера другой объект при помощи конструктора `Buffer` из одноименного модуля. В качестве параметра передадим туда массив с кодом символа — буквой B. Для того чтобы преобразовать полученный буфер к строке, достаточно просто вызвать метод `toString`. Более того, мы можем передать в качестве первого аргумента в метод toString кодировку этого символа.

```
1  var letterB = new Buffer([98]);
2
3  console.info(letterB.toString()); // b
4
5  console.info(letterB.toString('utf-8')); // b
```

Допустим, у нас есть некий буфер, и мы не знаем, в какой кодировке написана эта строка. Мы пробуем просто преобразовать его к строке, по умолчанию бу-

дет использована кодировка utf8, и мы на выходе получим не совсем желаемый результат.

```
var msg = new Buffer([0x2f, 0x04, 0x3d, 0x04, 0x34, 0x04, 0x35, 0x04,
 ↪  0x3a, 0x04, 0x41, 0x04]);

msg.toString(); // Default: utf8

// \u0004=\u00044\u00045\u0004:\u0004A\u0004

msg.toString('ucs2'); // 'Яндекс'
```

Преобразовать буфер в строку можно другим способом, передав параметр, указывающий кодировку методу readFile.

```
fs.readFile(__filename, function (err, data) {
    console.info(data.toString('utf-8'));
});

fs.readFile(__filename, 'utf-8', function (err, data) {
    console.info(data);
});
```

Помимо метода чтения файла, этот модуль, конечно, предлагает и много других возможностей. Мы можем добавлять содержимое файла, перезаписывать его, удалять файлы, и даже работать с директориями.

```
fs.appendFile();

fs.writeFile();

fs.unlink();

fs.mkdir();

fs.stat(__filename, function (stats) {
    console.info(stat.isDirectory()); // false
});
```

Метод `watch` позволяет следить за содержимым файла. Мы можем подписаться на изменения файла и получать уведомления каждый раз, когда этот файл будет изменен. В обработчик будет приходить первым аргументом событие: изменен этот файл или переименован. Таким же образом мы можем следить не только за файлами, но и за директориями.

```
fs.watch();

fs.watch(__filename, function (event, filename) {
    console.info(event); // change or rename
});

fs.watch(__dirname, function (event, filename) {
    console.info(event); // change or rename
});
```

Такой метод часто используется в программах, отвечающих за сборку файлов из какого-то исходного формата в какой-то конечный в автоматическом режиме.
Модуль fs также предлагает для всех методов синхронные способы их вызова. Использовать их следует с осторожностью, так как они будут блокировать поток выполнения вашей программы.

```
fs.readFileSync(__filename);

fs.writeFileSync(__filename, data);

fs.mkdirSync('/games/diablo3');
```

DOM

Курс: JavaScript, часть 2: прототипы и асинхронность

19 февраля 2018 г.

Оглавление

Глава 5

DOM

5.1. Поиск элементов

> **DOM (Document Object Model) – это программный интерфейс для работы с XML- и HTML-документами. Это дерево, узлами которого могут быть атрибуты, элементы, свойства и многие другие.**

Простейшее DOM-дерево представляет собой вот такую обычную HTML страницу.

```
<!DOCTYPE html>
<html>
<head>
 <title>DOM</title>
</head>
<body>
<form class="auth form" id="auth" data-form-value="123"
 ↪  action="/login/">
 <input type="text" value="" name="login">
 <input type="password" value="" name="password">
 <button>Войти</button>
</form>
</body>
</html>
```

DOM позволяет искать элементы, позволяет производить обход дерева, позволяет работать с атрибутами и свойствами узлов, а также добавлять, удалять их и обрабатывать на них события.
Работа с DOM-деревом всегда начинается с поиска элементов.
Метод `getElementById` возвращает нам элемент по его идентификатору. В данном случае мы ищем по идентификатору form и находим наш элемент формы. Объект, который возвращается нам, имеет класс HTMLFormElement.

```
1  document.getElementById('auth') // [object HTMLFormElement]
```

Метод `getElementById` всегда возвращает ровно один элемент, так как по спецификации элемент не может быть повторен с одинаковым идентификатором на странице. И в DOM-дереве всегда должен быть ровно один элемент с одним идентификатором. Как следствие, этот метод самый быстрый.
Следующий метод, который мы рассмотрим — это `getElementByTagName`. Он возвращает элементы с тем тегом, который указан в качестве параметра. В данном случае мы находим все элементы input. И этот метод всегда возвращает объект класса HTMLCollection.

```
1  document.getElementsByTagName('input') // [object HTMLCollection]
```

Этот метод достаточно устаревший и не очень удобный для применения, так как вам редко нужно находить все теги какого-то вида на странице.
На смену устаревшим методам для поиска были добавлены более современные, и прежде всего это метод `querySelector`, который позволяет искать внутри DOM-дерева элементы в зависимости от CSS-селектора.
В данном случае мы ищем нашу форму по ее идентификатору.

```
1  document.querySelector('#auth') // [object HTMLFormElement]
```

Здесь же мы ищем эту же самую форму, но уже по ее классу.

```
1  document.querySelector('.auth') // [object HTMLFormElement]
```

Механизм поиска элементов по CSS-селектору очень гибкий, так как с помощью различных CSS правил и их комбинаций вы можете задавать очень

сложные правила для поиска. Например, в данном случае мы находим все элементы, у которых есть атрибут `name`, значение которого — `login`.

```
document.querySelector('[name="login"]') // [object HTMLInputElement]
```

`querySelector` всегда возвращает первый элемент из всех возможных, найденных внутри DOM-дерева. Для поиска множества элементов разработан метод `querySelectorAll`, который принимает точно также на вход CSS-селектор, но возвращает уже все DOM-элементы, которые этому селектору соответствуют. В данном случае мы ищем все `input` и все `button` на странице и получаем объект класса `NodeList`.

```
document.querySelectorAll('input,button') // [object NodeList]
```

`HTMLCollection`, про которую мы говорили в связи с методом `getElementByTagName`, и `NodeList`, про который мы говорили только что вместе с методом `querySelectorAll`, не являются массивами. Как следствие, у них нет методов для итерирования, таких как forEach, map, reduce или любых других методов работы с массивами. Итак, что же мы можем сделать для итерирования? Самый простой способ для итерирования по коллекциям, и `HTMLCollection`, и `NodeList` — это цикл `for`. И тот и другой являются подобиями массива. И у того и у другого есть свойство `len`, которое говорит о количестве элементов внутри коллекции. И к любому элементу внутри коллекции можно обратиться по индексу.

```
var collection = document.querySelectorAll('input,button');

for (var i = 0, len = collection.length; i < len; i++) {
    var elem = collection[i];
}
```

Следующий вариант — это с помощью метода `Array.prototype.slice` преобразовать нашу коллекцию к настоящему массиву и уже воспользоваться всеми методами, которые есть у массива.

```
var elems = document.querySelectorAll('input,button');

var elemsList = Array.prototype.slice.call(elems);

elemsList.forEach(function(elem) {
    ...
});
```

И последний вариант, который предоставляют нам современные браузеры — это метод `Array.from`, который на вход получает что-то, подобное массиву, и преобразует это в полноценный, настоящий массив. В нашем случае мы получаем массив с элементами и уже можем применить любые методы для работы с массивами к ним.

```
var elemsList =
 ↪  Array.from(document.querySelectorAll('input,button'));

elemsList.forEach(function(elem) {
    ...
});
```

5.2. Атрибуты и свойства

Все элементы в DOM-дереве могут обладать атрибутами или свойствами, для работы со всеми ними разработаны методы, которые позволяют получить к ним доступ.

5.2.1. Star attribute

Мы получаем наш элемент формы с помощью `getElementById`, и впоследствии, когда вы будете видеть переменную `form` — это всегда наш элемент формы. И вызываем у нее метод `getAttribute`. В качестве параметра он получает имя атрибута, который мы хотим, к которому мы хотим обратиться, а в качестве возвращаемого значения в виде строки — содержимое этого атрибута. И таким образом мы получаем значение атрибута `action`.

```
var form = document.getElementById('auth');

form.getAttribute('action'); // '/login/'
```

Также есть метод `hasAttribute`, который проверяет наличие атрибута, и метод `setAttribute`, который устанавливает значение атрибута. При повторном вызове метод has уже вернет нам true, так как новый атрибут уже был добавлен в DOM-дерево. Есть также метод `remove` для удаления атрибута по его имени.

```
form.hasAttribute('method'); // false
form.setAttribute('method', 'POST');
form.hasAttribute('method'); // true
form.getAttribute('method'); // 'POST'
form.removeAttribute('method');
form.hasAttribute('method'); // false
```

5.2.2. data-атрибуты

> **data-атрибуты – атрибуты, которые начинаются с префикса data-, а далее следует название атрибута в виде латинских букв, цифр и знаков дефиса. И с помощью них можно записывать совершенно произвольные значения под совершенно произвольным именем.**

```
form.getAttribute('data-form-value'); // '123'

form.dataset.formValue; // '123'
form.dataset.hasOwnProperty('formValue'); // true

form.dataset.fooBarBazBaf = 'boo'; // data-foo-bar-baz-baf="boo"
form.hasAttribute('data-foo-bar-baz-baf'); // true
```

Существует специальный объект `dataset`, который есть внутри любого DOM-элемента, который содержит значения всех data-атрибутов, которые есть внутри этого элемента. Он преобразовывает имя атрибута, убирает от него префикс

data-, а все дефисы и последующие буквы преобразуют к CamelCase. Таким образом имя атрибута становится валидным идентификатором для переменной в JavaScript, и мы можем обращаться к нему через так называемую дотнотацию. И мы можем с помощью этого как обратиться к атрибуту на чтение, так и установить новое значение атрибута, которые будет прописано в DOM-дерево уже в развернутом data- виде.

5.2.3. Свойства

У всех элементов есть как атрибуты, так и свойства. И порой это совершенно разные вещи.

```
form.getAttribute('action'); // "/"
form.getAttribute('method'); // null
form.getAttribute('id'); // "auth"


form.action; // "https://yandex.ru/login/"
form.method; // "get"
form.id; // "auth"
```

Порой атрибуты и свойства ведут себя очень непредсказуемо, и значения у них могут сильно отличаться. Поэтому лучше использовать атрибуты в большинстве случаев вместо свойств, а если вы хотите пользоваться свойствами, то использовать самые простые и общие свойства: `id` и `className`.

5.2.4. Классы

Атрибут `class` — это, пожалуй, один из самых широко используемых атрибутов в html. С помощью него вы можете помечать элементы для того, чтобы наложить на них css-свойства и настроить их визуальное отображение, для того чтобы выбрать их с помощью селектора, и вообще объединить в какие-то логически связанные группы.
Для работы с атрибутом `class` есть свойство `className`, которое в строковом представлении содержит значение атрибута `class` в html и представляет весь набор классов, который указан у этого html-элемента в виде строки.

```
form.className; // 'auth form'
form.className += ' login-form'; // 'auth form login-form'
```

Но пользоваться им не очень удобно. Для того чтобы отредактировать набор классов: убрать класс, добавить класс, вам необходимо устанавливать это свойство и заменять строчку уже существующую на новую с помощью конкатенации, с помощью регулярных выражений или с помощью замены внутри строки. Это усложняет и поиск, и выборку, и проверку наличия класса.
Поэтому разработчики создали новый, достаточно современный метод `classList`, который содержит набор методов для работы с классами:

- `add`, который добавляет класс к списку уже существующих;
- `item`, который получает класс по его индексу и по порядковому расположению его в html-элементе;
- `contains`, который позволяет вам по имени класса проверить, есть ли он внутри этого элемента;
- `remove`, который позволяет вам удалить класс из html-элемента.

```
form.classList.add('login-form');
form.classList.item(1); // 'form'
form.classList.item(2); // 'login-form'
form.classList.contains('login-form'); // true
form.classList.remove('login-form');
```

5.3. Создание элементов

Давайте теперь поговорим о создании элемента. Страницы становятся все более интерактивными, и все больше элементов на них появляется динамически в зависимости от каких-то действий пользователя: мы получили ответ от сервера, отрисовали элемент, наполнили его текстом и таким образом сообщили пользователю о том, что произошло.
Для этого существует несколько методов. Прежде всего, это метод `createElement`, который, собственно говоря, отвечает за создание элемента, и метод `appendChild`, который добавляет элемент внутрь нашей страницы.

Сначала мы создаем элемент `span` с помощью `createElement` и конфигурируем наш элемент.

```
var elem = document.createElement('span');

elem.className = 'error';
elem.setAttribute('id', 'auth-error');
elem.setAttribute('status', 'auth-error');
elem.textContent = 'Введен неверный логин или пароль';
```

Важно понимать: созданный элемент находится в памяти и не находится внутри тела страницы (не связан с DOM-деревом).
С помощью `appendChild` мы можем добавить наш элемент к любому другому, как вложенный новый дочерний элемент. В нашем случае мы добавляем созданный и отконфигурированный `span` последним элементом к тегу `body`, то есть добавляем в самый конец страницы.

```
document.body.appendChild(elem);
```

При создании нового DOM-элемента нам не всегда нужно создавать его с нуля: можно взять за основу уже существующий, который есть на странице, скопировать его и поменять в нем какие-то свойства.
Для этого существует метод `cloneNode`, который принимает в качестве необязательного параметра true или false. False — это значение по умолчанию, а если мы передадим true, то все дочерние элементы внутри нашего элемента, который мы хотим склонировать, будут скопированы. Все это будет помещено во вновь созданный склонированный объект. И после этого мы можем так же, как у любого другого элемента, у этой копии поменять атрибуты и свойства, это никоим образом не повлияет на родительский элемент, с которого мы сняли снимок, и с помощью метода `appendChild` также добавить его на страницу.

```
var clone = elem.cloneNode(true);
clone.id = 'mail-error';
clone.textContent = 'Не удалось отправить письмо';

document.body.appendChild(clone);
```

Более подробно вы можете ознакомиться с методами по ссылкам на документацию.

5.4. События в DOM

События — это то, ради чего мы работаем с DOM-элементами вообще: на любых современных страницах множество событий, которые обрабатываются.
Так или иначе практически все DOM-элементы могут реагировать на те или иные события: загрузки, на которое реагирует вся страница, событие загрузки какого-то элемента, событие загрузки вашего JavaScript.
Некоторые DOM-элементы могут реагировать на какие-то события, которых нет у других. Например, событие `submit` есть только у формы. С полным списком событий вы можете ознакомиться в документации.
Самый простой способ добавить обработчик события на какой-то элемент — это явно прописать его в качестве атрибута у этого элемента. Для каждого события есть атрибут с префиксом on, который будет ему соответствовать: onsubmit у формы, onclick, например, у баттона. А далее вы указываете в качестве значения атрибута js-функцию, которая будет вызвана в качестве обработчика этого события.

```
1  <form class="auth form" id="auth" data-form-value="123"
   ↪  action="/login/" onsubmit="submitHandler()">
2
3  <input type="text" value="" name="login">
4
5  <input type="password" value="" name="password">
6
7  <button onclick="clickHandler()">Войти</button>
8
9  </form>
```

Это рабочий способ, и он будет работать даже в самых старых браузерах, но я бы настоятельно не рекомендовал вам им пользоваться:

- вы не можете навешать более одного обработчика на какое-то конкретное событие;
- JS код и ваш HTML код становятся очень тесно связаны;

- меняя HTML код или меняя JavaScript код, вам очень важно всегда помнить, что вы не можете переименовать значение/тип атрибута, иначе это будет другое событие;
- вы не можете навешивать эти обработчики динамически, должны изначально присутствовать в теле HTML-документа.

Для решения всех этих проблем разработчики стандартов W3C создали методaddEventListener, который добавляет слушателя на какое-то событие, которое может произойти на DOM-элементе. В нашем случае мы добавляем для формы на событие `submit`, которое указывается в качестве первого параметра, слушателя, который будет являться js-функцией.

```
var form = document.getElementById('auth');
form.addEventListener('submit', submitHandler);
```

Но обработчиков может быть значительно больше, и вы можете добавить их целый пул, таким образом создать стек обработки событий и разбить вашу обработку на несколько отдельных функций. В каждую из этих функций в качестве первого параметра будет передан объект `event`.

```
var button = document.querySelector('button');

button.addEventListener('click', function(event) {
    console.log(this, event);
});

button.addEventListener('click', clickHandler);

button.addEventListener('click', yetAnotherClickHandler);
```

Все обработчики всегда вызываются в том порядке, в котором они добавляются в качестве слушателей.
Контекстом любой функции обработки события всегда будет являться DOM-элемент, на котором это событие произошло, и на котором был вызван `addEventListener` для того, чтобы его послушать. Как я уже говорил, в любую функцию обработчика событий в качестве первого параметра передается объект события. Обычно эту переменную называют `event` для того, чтобы не путать со всеми

остальными. И у нее есть несколько свойств, которые для нас особенно важны. Прежде всего это свойство `target`, которое указывает на фактический DOM-узел — самый последний листок DOM-дерева, на котором произошло событие. Свойства `altKey`, `ctrlKey`, `shiftKey`, которые говорят о том, была ли нажата соответствующая клавиша во время события и значение type, которое указывает строкой тип события, который фактически сейчас произошел.
Существует два различных, противоположных способа обработки событий: Bubbling и Capturing.

Всплытие события или Bubbling

Это значение, которое принимается по умолчанию в браузерах, и тот вариант обработки событий, который наиболее распространен. При нем событие как бы поднимается вверх, от самого глубокого DOM-узла — к самому верхнему и заканчивается на объекте `document`, который описывает весь наш HTML документ. Для того, чтобы событие всплывало, в качестве последнего параметра в `addEventListener` нужно передать `false`, либо не передавать ничего, так как это значение по умолчанию.

Перехват событий или Capturing

При этом обработчике событий событие погружается от самого верхнего элемента к самому глубокому, самому нижнему, на котором оно могло бы произойти, и сначала срабатывает на объекте `document` и постепенно погружается вниз.
Для того, чтобы событие перехватывалось, в качестве последнего параметра в `addEventListener` вам нужно передать `true`.
Иногда вы хотите остановить событие на каком-то конкретном участке и не хотите, чтобы оно всплывало или погружалось далее. Для того, чтобы остановить всплытие, существует метод `stopPropagation`, который вызывается у объекта «событие». То же самое можно сделать с Capturing вызовом того же метода: событие не будет погружаться дальше.

```
1 var button = document.querySelector('button');
2 button.addEventListener('click', function(event) {
3     event.stopPropagation();
4 }, true); // Capturing
```

Множество событий могут быть назначены на одном элементе, и тогда эти обработчики выстроятся в стек и будут вызываться постепенно, друг за другом, в порядке их добавления. Но если мы этого не хотим, если мы хотим остановить все обработчики в рамках этого элемента, а также не давать им всплывать или погружаться, то мы можем вызвать метод `stopImmediatePropagation` на требуемом элементе. Тогда стек будет опустошен, и событие перестанет всплывать/погружаться. Но пользоваться этим методом нужно очень осторожно, так как вы можете удалить обработчик, который был поставлен другим человеком.
У многих DOM-элементов есть так называемые действия по умолчанию. Например, при клике на ссылки, мы переходим по ссылке, при клике по кнопке мы отправляем форму.Для того чтобы отменить действие по умолчанию, существует метод `preventDefault`, который вызывается у объекта-события.

```
1 var form = document.getElementById('auth');
2
3 form.addEventListener('submit', function(event) {
4     console.log(event.target);
5     event.preventDefault();
6 });
```

5.4.1. Делегировние событий

> **"Делегировние событий" – метод обработки события на родительском элементе относительно того, на котором оно произошло фактически.**

Например, у нас есть список, состоящий из множества ссылок, которые появляются динамически, в зависимости от каких-то действий пользователя.

```
<ul>
<li><a href="#maps">Карты</a></li>
<li><a href="#market">Маркет</a></li>
<li><a href="#news">Новости</a></li>
<li><a href="#translate">Переводчик</a></li>
<li><a href="#images">Картинки</a></li>
<li><a href="#video">Видео</a></li>
<li><a href="#music">Музыка</a></li>
<li><a href="#more" class="more">ещё</a></li>
</ul>
```

Мы бы могли получить все ссылки, пойти по ним в цикле и на каждую назначить свой обработчик событий. Но при добавлении любой новой нам бы приходилось каждый раз делать это заново, а при удалении какого-то элемента нам нужно было бы удалять за ним и события.
Мы можем поступить иначе: назначить обработчик на весь наш список и ловить конкретные события на конкретных элементах. В нашем случае это делегирование можно было бы оформить следующим образом.

```
document.addEventListener('click', function(event) {
    event.preventDefault();

    if (event.target.tagName === 'A') {
        if (event.target.classList.contains('more')) {
            openMorePopup();
        } else {
            console.log(event.target.href);
        }
    }
});
```

Мы начинаем слушать событие клика на объекте `document`, для которого всплывают все события. После этого мы отменяем действия по умолчанию (переход по ссылке). Далее мы обращаемся к свойству `event.target` и получаем фактически элемент, на котором произошло событие, то есть то, куда мы в итоге кликнули, и проверяем, является ли он ссылкой. А далее, с помощью метода `classList` проверяем, есть ли у этой ссылки тот или иной класс. И в зависимости от этого, либо открываем наш попап, либо переходим по ссылке.

Делегирование — достаточно мощный инструмент, который позволяет навешивать вам всего один обработчик на один тип события и получать их с множества различных узлов. Эти узлы могут создаваться динамически, могут удаляться, но не надо переживать за то, добавили ли вы обработку событий на него или нет — любое событие всплывет и может быть обработано на объекте `document` как самой высшей инстанции для обработки события. Единственное, что вам стоит при этом учитывать — это то, что у объекта события нет методов для проверки того, на каком фактически элементе оно произошло и есть ли у этого элемента какой-то класс/ CSS-селектор. И из-за этого приходится городить конструкции в виде нескольких if'ов, проверять явно. Рекомендую поступать следующим образом: получать `event.target`. Так как это может быть глубокий вложенный элемент, всегда помнить об этом, подниматься на определенное количество уровней выше до какого-то элемента с каким-то классом, который вас гарантированно интересует. Например, в нашем случае это мог бы быть тег `le` с элементом списка. И уже на нем проверять события и навешивать обработчики.