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.

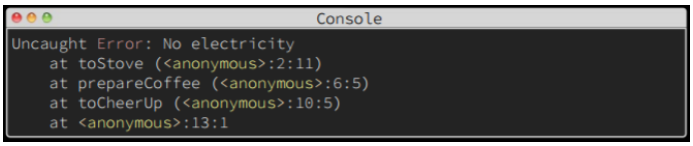
```
function prepareCoffee()
 ↪  {
    toStove();
}
function toCheerUp() {
    prepareCoffee();
}
toCheerUp();
```

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,

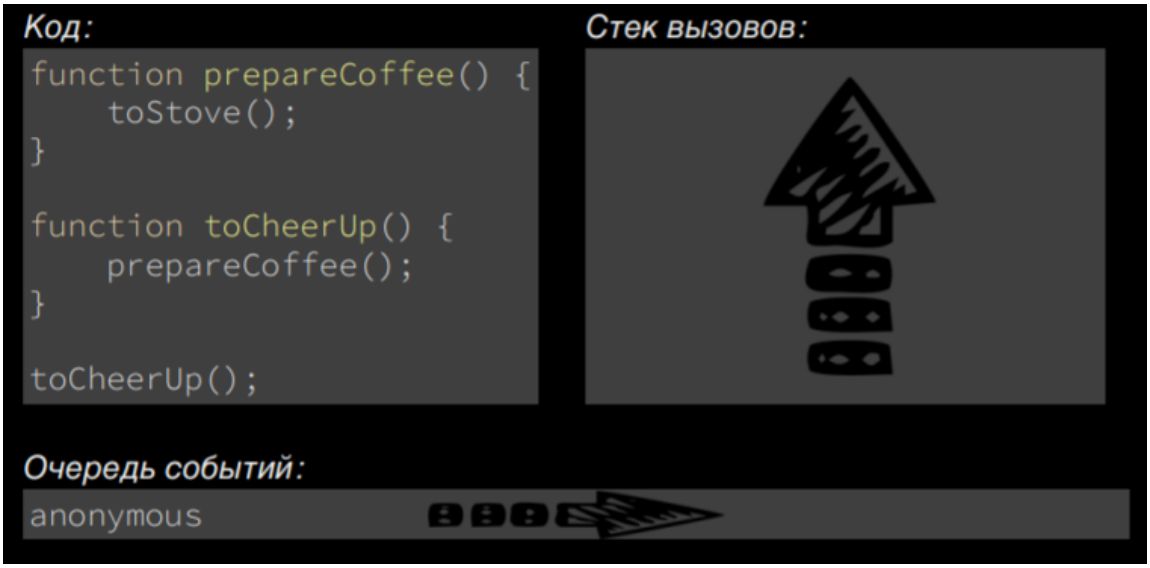
то в консоли мы увидим следующую ошибку: мы не можем поставить кофе на плиту, потому что нет электричества, и увидим стек вызовов, где ошибка произошла.

```
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.

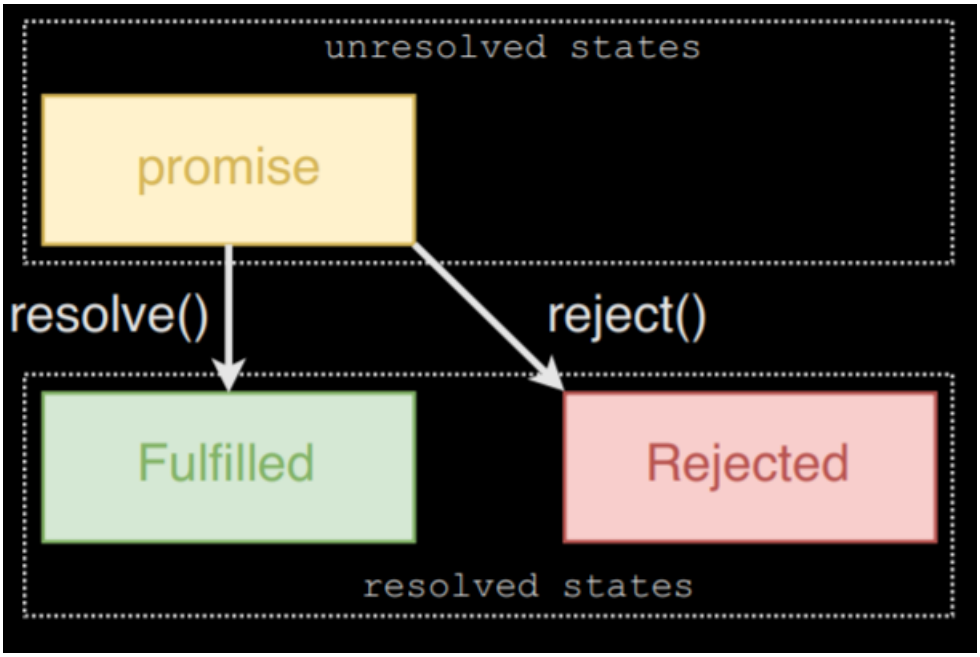
3.5. Промисы

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.

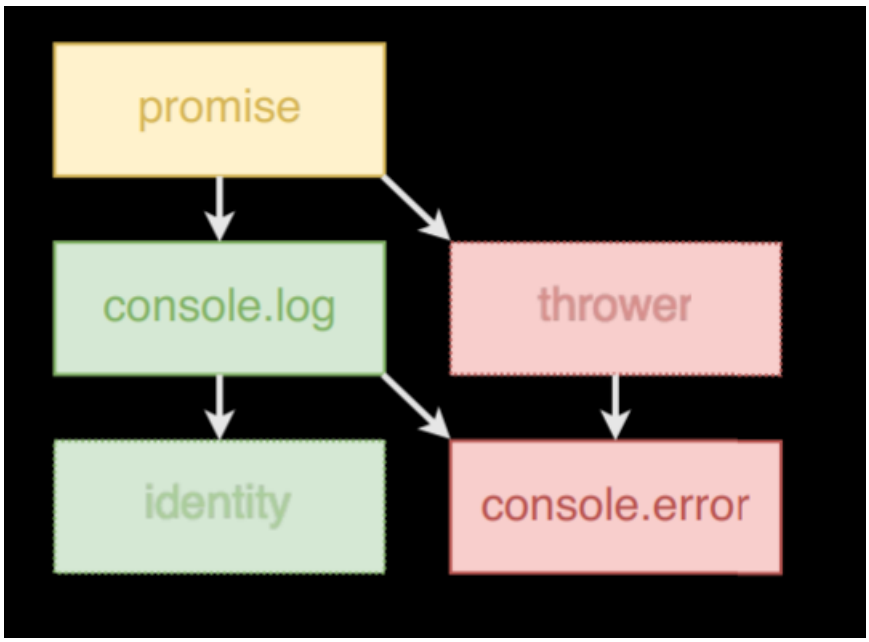
```
function identity(data) {
    return data;
}

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);
```

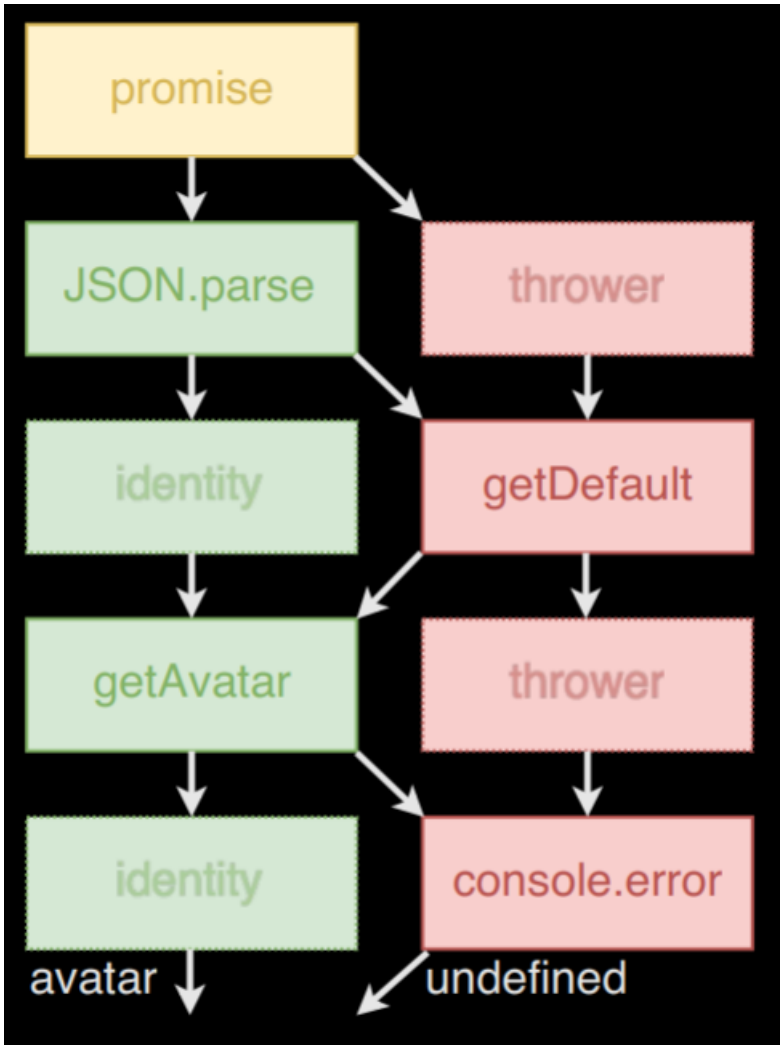
Если чтение файла завершилось хорошо, мы вызовем функцию `JSON.parse`, Эта функция преобразует содержимое файла в JSON. Если чтение завершилось с ошибкой, мы вызываем функцию `thrower`. Эта функция опрокидывает ошибку далее по цепочке промисов. Если парсинг завершился хорошо, то мы попадаем в метод `identity`, 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);
```