|  |  |
| --- | --- |
| **Spread syntax** (...) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected. | Синтаксис распространения (...) позволяет расширить итерацию, такую ​​как выражение массива или строка, в местах, где ожидается ноль или более аргументов (для вызовов функций) или элементов (для литералов массива), или выражение объекта, которое должно быть развернуто в места, где ожидается ноль или более пар ключ-значение (для объектных литералов). |
| Spread syntax can be used when all elements from an object or array need to be included in a list of some kind. | Синтаксис распространения можно использовать, когда все элементы объекта или массива необходимо включить в какой-либо список. |
| In the above example, the defined function takes x, y, and z as arguments and returns the sum of these values. An array value is also defined. | В приведенном выше примере определенная функция принимает x, y и z в качестве аргументов и возвращает сумму этих значений. Также определяется значение массива. |
| When we invoke the function, we pass it all the values in the array using the spread syntax and the array name — ...numbers. | Когда мы вызываем функцию, мы передаем ей все значения в массиве, используя синтаксис распространения и имя массива - ... числа. |
| If the array contained more than three numbers, e.g. [1, 2, 3, 4], then it would still work fine, except that all four would be passed, but only the first three would be used unless you added more arguments to the function, e.g.: | Если в массиве более трех чисел, например [1, 2, 3, 4], тогда он все равно будет работать нормально, за исключением того, что все четыре будут переданы, но будут использоваться только первые три, если вы не добавите дополнительные аргументы в функцию, например: |
| The above example is somewhat rigid; the real value in spread syntax is that it works with the same value, no matter how many elements are contained in the object, array, etc. | Приведенный выше пример несколько жесток; реальное значение синтаксиса распространения состоит в том, что он работает с одним и тем же значением, независимо от того, сколько элементов содержится в объекте, массиве и т. д. |
| It is commonly used when you want to add a new item to a local data store, or display all stored items plus a new addition. A very simple version of this kind of action could look like so: | Он обычно используется, когда вы хотите добавить новый элемент в локальное хранилище данных или отобразить все сохраненные элементы плюс новое добавление. Очень простой вариант такого действия может выглядеть так: |
| In the above example you can rerun the last line as many times as you like, to keep adding an additional 12 to the end of the array. | В приведенном выше примере вы можете повторно запускать последнюю строку сколько угодно раз, чтобы продолжать добавлять еще 12 в конец массива. |
| Rest syntax looks exactly like spread syntax. In a way, rest syntax is the opposite of spread syntax. Spread syntax "expands" an array into its elements, while rest syntax collects multiple elements and "condenses" them into a single element. See [rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters). | Рест-синтаксис выглядит точно так же, как синтаксис распространения. В каком-то смысле синтаксис rest противоположен синтаксису распространения. Синтаксис Spread «расширяет» массив на его элементы, в то время как синтаксис rest собирает несколько элементов и «уплотняет» их в один элемент. См. Остальные параметры. |
| If you need to use any of the special characters literally (actually searching for a "\*", for instance), you must escape it by putting a backslash in front of it. For instance, to search for "a" followed by "\*" followed by "b", you'd use /a\\*b/ — the backslash "escapes" the "\*", making it literal instead of special. | Если вам нужно использовать какой-либо из специальных символов буквально (например, при поиске "\*"), вы должны избежать его, поставив перед ним обратную косую черту. Например, для поиска «a», за которым следует «\*», за которым следует «b», вы должны использовать / a \ \* b / - обратная косая черта «экранирует» «\*», делая его буквальным, а не специальным. |
| Similarly, if you're writing a regular expression literal and need to match a slash ("/"), you need to escape that (otherwise, it terminates the pattern). For instance, to search for the string "/example/" followed by one or more alphabetic characters, you'd use /\/example\/[a-z]+/i—the backslashes before each slash make them literal. | Точно так же, если вы пишете литерал регулярного выражения и вам нужно сопоставить косую черту ("/"), вам нужно убрать его (в противном случае он завершает шаблон). Например, для поиска строки «/ example /», за которой следует один или несколько буквенных символов, вы должны использовать / \ / example \ / [a-z] + / i - обратная косая черта перед каждой косой чертой делает их буквальными. |
| To match a literal backslash, you need to escape the backslash. For instance, to match the string "C:\" where "C" can be any letter, you'd use /[A-Z]:\\/ — the first backslash escapes the one after it, so the expression searches for a single literal backslash. | Чтобы соответствовать буквальной обратной косой черте, вам нужно избегать обратной косой черты. Например, чтобы соответствовать строке «C: \», где «C» может быть любой буквой, вы должны использовать / [AZ]: \\ / - первая обратная косая черта экранирует следующую после нее, поэтому выражение ищет одну буквальная обратная косая черта. |
| If using the RegExp constructor with a string literal, remember that the backslash is an escape in string literals, so to use it in the regular expression, you need to escape it at the string literal level. /a\\*b/ and new RegExp("a\\\*b") create the same expression, which searches for "a" followed by a literal "\*" followed by "b". | Если вы используете конструктор RegExp со строковым литералом, помните, что обратная косая черта является escape-символом в строковых литералах, поэтому, чтобы использовать его в регулярном выражении, вам нужно экранировать его на уровне строкового литерала. / a \ \* b / и new RegExp («a \\ \* b») создают одно и то же выражение, которое ищет «a», за которым следует буквальный «\*», за которым следует «b». |
| Если вы хотите узнать, найден ли шаблон в строке, используйте методы test () или search (); для получения дополнительной информации (но более медленного выполнения) используйте методы exec () или match (). Если вы используете exec () или match () и в случае успешного совпадения, эти методы возвращают массив и обновляют свойства связанного объекта регулярного выражения, а также предопределенного объекта регулярного выражения RegExp. Если совпадение не удается, метод exec () возвращает значение null (которое приводит к значению false). | When you want to know whether a pattern is found in a string, use the test() or search() methods; for more information (but slower execution) use the exec() or match() methods. If you use exec() or match() and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, RegExp. If the match fails, the exec() method returns null (which coerces to false). |
| A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) is an object representing the eventual completion or failure of an asynchronous operation. Since most people are consumers of already-created promises, this guide will explain consumption of returned promises before explaining how to create them. | Обещание - это объект, представляющий возможное завершение или сбой асинхронной операции. Поскольку большинство людей являются потребителями уже созданных обещаний, это руководство объяснит потребление возвращенных обещаний, прежде чем объяснять, как их создавать. |
| Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function. | По сути, обещание - это возвращаемый объект, к которому вы прикрепляете обратные вызовы, вместо того, чтобы передавать обратные вызовы в функцию. |
| Imagine a function, createAudioFileAsync(), which asynchronously generates a sound file given a configuration record and two callback functions, one called if the audio file is successfully created, and the other called if an error occurs. | Представьте себе функцию createAudioFileAsync (), которая асинхронно генерирует звуковой файл с учетом записи конфигурации и двух функций обратного вызова, одна из которых вызывается в случае успешного создания аудиофайла, а другая - в случае возникновения ошибки. |
| This convention has several advantages. We will explore each one. | У этого соглашения есть несколько преимуществ. Мы исследуем каждую из них. |
| Unlike old-fashioned passed-in callbacks, a promise comes with some guarantees: | В отличие от устаревших переданных обратных вызовов, обещание имеет некоторые гарантии: |
| Callbacks added with [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) will never be invoked before the [completion of the current run](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#run-to-completion) of the JavaScript event loop. | Обратные вызовы, добавленные с помощью then (), никогда не будут вызываться до завершения текущего запуска цикла событий JavaScript. |
| These callbacks will be invoked even if they were added *after* the success or failure of the asynchronous operation that the promise represents. | Эти обратные вызовы будут вызываться, даже если они были добавлены после успеха или неудачи асинхронной операции, которую представляет обещание. |
| Multiple callbacks may be added by calling [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) several times. They will be invoked one after another, in the order in which they were inserted. | Можно добавить несколько обратных вызовов, вызвав then () несколько раз. Они будут вызываться один за другим в том порядке, в котором они были вставлены. |
| One of the great things about using promises is **chaining**. | Одна из замечательных особенностей использования обещаний - это цепочка. |
| A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. We accomplish this by creating a **promise chain**. | Распространенной потребностью является выполнение двух или более асинхронных операций подряд, при этом каждая последующая операция начинается, когда предыдущая операция завершается успешно, с результатом с предыдущего шага. Мы достигаем этого, создавая цепочку обещаний. |
| This second promise (promise2) represents the completion not just of doSomething(), but also of the successCallback or failureCallback you passed in, which can be other asynchronous functions returning a promise. When that's the case, any callbacks added to promise2 get queued behind the promise returned by either successCallback or failureCallback. | Это второе обещание (обещание2) представляет собой завершение не только doSomething (), но также переданных вами successCallback или failureCallback, которые могут быть другими асинхронными функциями, возвращающими обещание. В этом случае любые обратные вызовы, добавленные к обещанию2, помещаются в очередь после обещания, возвращаемого либо successCallback, либо failureCallback. |
| Basically, each promise represents the completion of another asynchronous step in the chain. | По сути, каждое обещание представляет собой завершение другого асинхронного шага в цепочке. |
| In the old days, doing several asynchronous operations in a row would lead to the classic callback pyramid of doom: | Раньше выполнение нескольких асинхронных операций подряд приводило к классической пирамиде обратного вызова гибели: |
| With modern functions, we attach our callbacks to the returned promises instead, forming a promise chain: | В современных функциях мы вместо этого присоединяем обратные вызовы к возвращаемым обещаниям, формируя цепочку обещаний: |
| It's possible to chain after a failure, i.e. a catch, which is useful to accomplish new actions even after an action failed in the chain. Read the following example: | Возможно создание цепочки после сбоя, то есть улова, который полезен для выполнения новых действий даже после сбоя действия в цепочке. Прочтите следующий пример: |
| A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future. | Обещание - это прокси для значения, не обязательно известного на момент создания обещания. Он позволяет связать обработчики с конечным значением успеха или причиной сбоя асинхронного действия. Это позволяет асинхронным методам возвращать значения, как синхронные методы: вместо того, чтобы немедленно возвращать окончательное значение, асинхронный метод возвращает обещание предоставить значение в какой-то момент в будущем. |
| A Promise is in one of these states:   * *pending*: initial state, neither fulfilled nor rejected. * *fulfilled*: meaning that the operation was completed successfully. * *rejected*: meaning that the operation failed. | Обещание находится в одном из следующих состояний:  в ожидании: исходное состояние, ни выполнено, ни отклонено.  выполнено: означает, что операция была успешно завершена.  отклонено: это означает, что операция не удалась. |
| A pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached | Отложенное обещание может быть выполнено со значением или отклонено по причине (ошибка). Когда происходит любой из этих вариантов, вызываются связанные обработчики, поставленные в очередь методом then обещания. Если обещание уже выполнено или отклонено при присоединении соответствующего обработчика, обработчик будет вызван, поэтому между завершением асинхронной операции и присоединением ее обработчиков не будет состояния гонки. |
| Several other languages have mechanisms for lazy evaluation and deferring a computation, which they also call "promises", e.g. Scheme. Promises in JavaScript represent processes that are already happening, which can be chained with callback functions. If you are looking to lazily evaluate an expression, consider the [arrow function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) with no arguments: f = () => expression to create the lazily-evaluated expression, and f() to evaluate. | В нескольких других языках есть механизмы для отложенного вычисления и отложенного вычисления, которые они также называют «обещаниями», например Схема. Обещания в JavaScript представляют собой процессы, которые уже происходят, которые могут быть связаны с функциями обратного вызова. Если вы хотите лениво вычислить выражение, рассмотрите стрелочную функцию без аргументов: f = () =&gt; expression, чтобы создать выражение с отложенным вычислением, и f (), чтобы оценить. |
| The .then() method takes up to two arguments; the first argument is a callback function for the resolved case of the promise, and the second argument is a callback function for the rejected case. Each .then() returns a newly generated promise object, which can optionally be used for chaining; for example: | Метод .then () принимает до двух аргументов; первый аргумент - это функция обратного вызова для разрешенного случая обещания, а второй аргумент - функция обратного вызова для отклоненного случая. Каждый .then () возвращает новый сгенерированный объект обещания, который при желании можно использовать для связывания; Например: |
| Handling a rejected promise in each .then() has consequences further down the promise chain. Sometimes there is no choice, because an error must be handled immediately. In such cases we must throw an error of some type to maintain error state down the chain. On the other hand, in the absence of an immediate need, it is simpler to leave out error handling until a final .catch() statement. A .catch() is really just a .then() without a slot for a callback function for the case when the promise is resolved. | Обработка отклоненного обещания в каждом .then () имеет последствия в цепочке обещаний. Иногда выбора нет, потому что ошибку нужно обрабатывать немедленно. В таких случаях мы должны выдать ошибку определенного типа, чтобы поддерживать состояние ошибки в цепочке. С другой стороны, при отсутствии немедленной необходимости проще оставить обработку ошибок до финального оператора .catch (). На самом деле .catch () - это просто .then () без слота для функции обратного вызова на тот случай, когда обещание выполнено. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |