**Promise -edit**

1. Cú pháp:

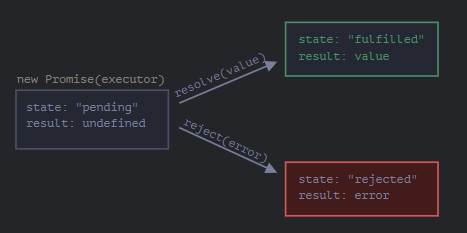
let promise = new Promise(function(resolve, reject) {

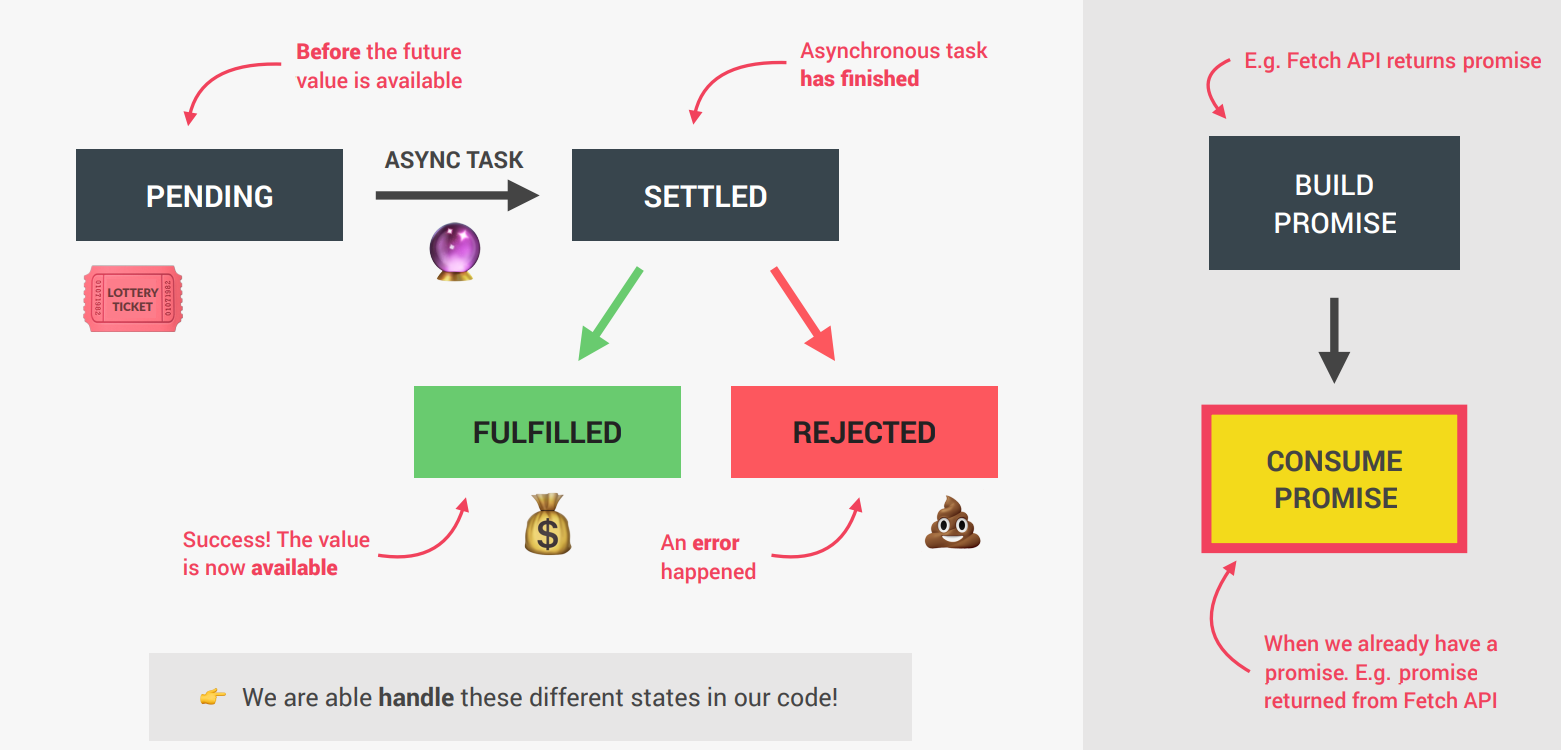
// executor (the producing code, "singer")

});

|  |  |
| --- | --- |
| New Promise | * Được gọi là excutor (thực thi) * *Khi new Promise đc tạo, thì excutor sẽ tự động chạy* |
| Resole, reject   * *Resole(value) nếu công việc thực thi thành công trả về kết quả value* * *Reject(error) nếu xảy ra lỗi, error là error object* | * Các đối số (arguments) * Là callback do chính JS cung cấp |

* Promise lifecycle: Promise object được trả về bởi cấu trúc new Promise có các thuộc tính nội bộ gồm:
* State – ban đầu là “pending” (đang chờ xử lý) sau đó thay đổi thành “fulfilled (hoàn thành)” khi được giải quyết hoặc “rejected (từ chối)” khi bị từ chối.
* Result – ban đầu là “undefined” sau khi đổi giá trị khi resolve(value) được gọi hoặc error khi reject(error) được gọi.





* *Fulfilled: là 1 promise trả thành công về 1 giá trị mà ta mong muốn. (Giá trị bây giờ là có sẵn)*
* *1 Promise được resolve (giải quyết) hoặc reject (bị từ chối) được gọi là “settled (đã giải quyết)”, trái ngược với 1 promise “pending” ban đầu.*

1. **VD1: Thực thi thành công “Fulfiled promise”**

let promise = new Promise(function(resolve, reject) {

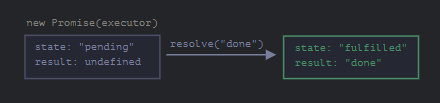
// the function is executed automatically when the promise is constructed

// after 1 second signal that the job is done with the result "done"

setTimeout(() => resolve("done"), 1000);

});

* Executor đã được gọi tự động và ngay lập tức (bởi new Promise)
* Executor nhận được 2 đối số là resolve và reject. Các function này được định nghĩa bởi JS vì vậy chúng ta k cần phải định nghĩa chúng. Chúng ta chỉ cần gọi chúng khi sẵn sàng.
  + Sau 1 giây “xử lý” executor gọi resolve(“done”) để tạo kết quả. Điều này thay đổi trạng thái của promise object.



1. **VD2: Reject (error)**

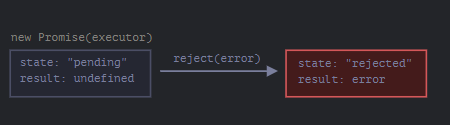
let promise = new Promise(function(resolve, reject) {

// after 1 second signal that the job is finished with an error

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

Lệnh gọi reject (…) di chuyển promise object sang trạng thái “rejected (bị từ chối)”



1. NOTE1:

Executor nên thực hiện một công việc ( thường là việc gì đó mất thời gian) và sau đó gọi resolve hoặc reject để thay đổi trạng thái của promise object tương ứng.

1. NOTE2: PROMISES là

* 1 Object được sử dụng như 1 người giữ chỗ cho kết quả tương lai của 1 hành động không đồng bộ
* Giống như 1 thùng chứa cho 1 giá trị được phân phối không đồng bộ
* Là 1 thùng chứa cho 1 giá trị trong tương lai (VD phản hồi từ 1 cuộc call AJAX response
* Chúng ta k cần phải dự vào event và các callback function để xử lý các kết quả không đồng bộ
* Thay vì callback lồng nhau, chúng ta có thể sử dụng chain promise cho 1 trình tự các hoạt động không đồng bộ (thoát khỏi callback hell)

1. NOTE3: Có thể chỉ có 1 kết quả duy nhất hoặc 1 error

Executor nên chỉ gọi 1 lần resolve hoặc 1 reject. Bất kỳ thay đổi trạng thái thái là cuối cùng

Tất cả các cuộc gọi resolve và reject đều bị bỏ qua

let promise = new Promise(function(resolve, reject) {

resolve("done");

reject(new Error("…")); // ignored

setTimeout(() => resolve("…")); // ignored

});

* Ý tưởng là 1 công vc được làm bởi executor có thể có 1 kết quả hoặc 1 error. Ngoài ra resolve/ reject mong đợi chỉ 1 đối số ( hoặc k có) và sẽ bỏ qua các đối số bổ sung

1. Reject với error objects

Trong trường hợp xảy ra sự cố, executor nên gọi reject. Điều này có thể được thực hiện với bất kỳ loại đối số nào giống như resolve. Nhưng nên sử dụng các Error object hoặc các error object kế thừa.

1. Gọi ngay lập tức resolve / reject

Trong thực tế, 1 executor luôn thực hiện điều gì đó không đồng bộ và gọi resolve/ reject sau 1 thời gian nhưng không nhất thiết phải làm vậy. Chúng ta có thể gọi resolve/ reject ngay lập tức ntn:

let promise = new Promise(function(resolve, reject) {

// not taking our time to do the job

resolve(123); // immediately give the result: 123

});

Chằng hạn như điều này có thể xảy ra khi chúng ta bắt đầu làm 1 công vc nhưng sau đó thấy rằng mọi thứ đã được hoàn thành và được lưu vào bộ nhớ cached

1. State và result là nội bộ

Thuộc tính state và result của promise object là nội bộ. Chúng ta k thể truy cập trực tiếp vào chúng. Chúng ta có thể sử dụng method .then / .catch/ .finally cho vc đó.

**Then, catch**

1. Then

promise.then(

function(result) { /\* handle a successful result \*/ },

function(error) { /\* handle an error \*/ }

);

* Đối số đầu tiên của .then là 1 function cái mà chạy khi promise được (resolve) giải quyết và nhận (result) kết quả
* Đối số thứ 2 của .then là 1 function cái mà chạy khi promise được (reject) từ chối và nhận được error

VD1. A sussessfully resolved promise

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

// resolve runs the first function in .then

promise.then(

result => alert(result), // shows "done!" after 1 second

error => alert(error) // doesn't run

);

Function đầu tiên được thực thi (executed)

VD2. A rejection

let promise = new Promise(function(resolve, reject) {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// reject runs the second function in .then

promise.then(

result => alert(result), // doesn't run

error => alert(error) // shows "Error: Whoops!" after 1 second

);

* Nếu chúng ta chỉ quan tâm đến việc hoàn thành thành công thì chúng ta chỉ có thể cung cấp 1 đối số chức năng cho .then

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // shows "done!" after 1 second

1. **Catch**

Nếu chúng ta chỉ cần quan tâm đến error thì chỉ cần sử dụng null như 1 đối số đầu tiên*: .then(null, errorHandlingFunction)*. Hoặc chúng ta có thể sử dụng .*catch(errorHandlingFunction)*

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// .catch(f) is the same as promise.then(null, f)

promise.catch(alert); // shows "Error: Whoops!" after 1 second

* .catch(f) tương tự như .then(null, f)

1. **Finally “cuối cùng” (trình xử lý cuối cùng)**

Finally như 1 mệnh đề cuối trong try { … } catch {…}, nó là finally trong promise

* .finally(f) tương tự như .then(f, f) trong khi f luôn chạy, khi promise được giải quyết có thể là resolve hoặc reject
* Ý tưởng của finally là thiết lập 1 trình xử lý để thực hiện cleanup/ finalizing sau khi các kích hoạt trc đó đc hoàn tất.

new Promise((resolve, reject) => {

/\* do something that takes time, and then call resolve or maybe reject \*/

})

// runs when the promise is settled, doesn't matter successfully or not

.finally(() => stop loading indicator)

// so the loading indicator is always stopped before we go on

.then(result => show result, err => show error)

NOTE: finally(f) k chính là bí danh của then(f, f)

* 1 finally xử lý k có đối số. Trong finally chúng ta k bt liệu promise có thành công hay không.
* A finally xử lý “passes through” kết quả hoặc error cho trình xử lý phù hợp tiếp theo (vd result đc chuyền từ finally đến then)

new Promise((resolve, reject) => {

setTimeout(() => resolve("value"), 2000);

})

.finally(() => alert("Promise ready")) // triggers first

.then(result => alert(result)); // <-- .then shows "value"

* Giá trị (value) đc trả về bởi promise đầu tiên được chuyển đến finally sau đó sang tiếp .then. Điều này rất thuận tiện bởi finally k có nghĩa là xử lý 1 kết quả promise (nó là nơi dọn dẹp “cleanup” bất kể kết quả là gì)

new Promise((resolve, reject) => {

throw new Error("error");

})

.finally(() => alert("Promise ready")) // triggers first

.catch(err => alert(err)); // <-- .catch shows the error

* Một trình xử lý cuối cùng cũng không nên trả lại bất cứ thứ gì. Nếu đúng như vậy, giá trị được trả về sẽ bị bỏ qua một cách âm thầm.
* Ngoại lệ duy nhất cho quy tắc này là khi trình xử lý cuối cùng đưa ra lỗi. Sau đó, lỗi này chuyển đến trình xử lý tiếp theo, thay vì bất kỳ kết quả nào trước đó.
* **Để tóm tắt:**
* Trình xử lý cuối cùng không nhận được kết quả của trình xử lý trước đó (nó không có đối số). Thay vào đó, kết quả này được chuyển qua trình xử lý phù hợp tiếp theo.
* Nếu trình xử lý cuối cùng trả về một cái gì đó, nó sẽ bị bỏ qua.
* Khi cuối cùng đưa ra một lỗi, thì việc thực thi sẽ chuyển đến trình xử lý lỗi gần nhất.

1. Chúng tôi có thể đính kèm các trình xử lý vào các lời hứa đã giải quyết

Nếu một lời hứa đang chờ xử lý, các trình xử lý .then/catch/finally sẽ đợi kết quả của nó.

Đôi khi, có thể một lời hứa đã được giải quyết khi chúng ta thêm một trình xử lý vào nó.

// the promise becomes resolved immediately upon creation

let promise = new Promise(resolve => resolve("done!"));

promise.then(alert); // done! (shows up right now)

NOTE:

Test1: output?

let promise = new Promise(function(resolve, reject) {

resolve(1);

setTimeout(() => resolve(2), 1000);

});

promise.then(alert);

* Đầu ra là: 1.
* Lệnh gọi giải quyết thứ hai bị bỏ qua vì chỉ lệnh gọi từ chối/giải quyết đầu tiên được tính đến. Các cuộc gọi tiếp theo sẽ bị bỏ qua

