February 20, 2022

Promise API

There are 6 static methods in the `Promise` class. We'll quickly cover their use cases here.

Promise.all

Let's say we want many promises to execute in parallel and wait until all of them are ready.

For instance, download several URLs in parallel and process the content once they are all done.

That's what `Promise.all` is for.

The syntax is:

```
let promise = Promise.all(iterable);
```

`Promise.all` takes an iterable (usually, an array of promises) and returns a new promise.

The new promise resolves when all listed promises are resolved, and the array of their results becomes its result.

For instance, the `Promise.all` below settles after 3 seconds, and then its result is an array `[1, 2, 3]`:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 when promises are ready: each promise contributes an array
```

Please note that the order of the resulting array members is the same as in its source promises. Even though the first promise takes the longest time to resolve, it's still first in the array of results.

A common trick is to map an array of job data into an array of promises, and then wrap that into `Promise.all`.

For instance, if we have an array of URLs, we can fetch them all like this:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// map every url to the promise of the fetch
let requests = urls.map(url => fetch(url));

// Promise.all waits until all jobs are resolved
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

A bigger example with fetching user information for an array of GitHub users by their names (we could fetch an array of goods by their ids, the logic is identical):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // all responses are resolved successfully
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // shows 200 for every url
    }

    return responses;
  })
  // map array of responses into an array of response.json() to read their content
  .then(responses => Promise.all(responses.map(r => r.json())))
  // all JSON answers are parsed: "users" is the array of them
  .then(users => users.forEach(user => alert(user.name)));
```

If any of the promises is rejected, the promise returned by `Promise.all` immediately rejects with that error.

For instance:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Here the second promise rejects in two seconds. That leads to an immediate rejection of `Promise.all`, so `.catch` executes: the rejection error becomes the outcome of the entire `Promise.all`.

In case of an error, other promises are ignored

If one promise rejects, `Promise.all` immediately rejects, completely forgetting about the other ones in the list. Their results are ignored.

For example, if there are multiple `fetch` calls, like in the example above, and one fails, the others will still continue to execute, but `Promise.all` won't watch them anymore. They will probably settle, but their results will be ignored.

`Promise.all` does nothing to cancel them, as there's no concept of "cancellation" in promises. In another chapter we'll cover `AbortController` that can help with that, but it's not a part of the Promise API.

`Promise.all(iterable)` allows non-promise "regular" values in `iterable`

Normally, `Promise.all(...)` accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's passed to the resulting array "as is".

For instance, here the results are `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

So we are able to pass ready values to `Promise.all` where convenient.

Promise.allSettled

A recent addition

This is a recent addition to the language. Old browsers may need polyfills.

`Promise.all` rejects as a whole if any promise rejects. That's good for "all or nothing" cases, when we need *all* results successful to proceed:

```
Promise.all([
  fetch('/template.html'),
```

```
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // render method needs results of all fetches
```

`Promise.allSettled` just waits for all promises to settle, regardless of the result. The resulting array has:

- `{status:"fulfilled", value:result}` for successful responses,
- `{status:"rejected", reason:error}` for errors.

For example, we'd like to fetch the information about multiple users. Even if one request fails, we're still interested in the others.

Let's use `Promise.allSettled`:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

The `results` in the line `(*)` above will be:

```
[
  {status: 'fulfilled', value: ...response...},
  {status: 'fulfilled', value: ...response...},
  {status: 'rejected', reason: ...error object...}
]
```

So for each promise we get its status and `value/error`.

Polyfill

If the browser doesn't support `Promise.allSettled`, it's easy to polyfill:

```
if (!Promise.allSettled) {
  const rejectHandler = reason => ({ status: 'rejected', reason });

  const resolveHandler = value => ({ status: 'fulfilled', value });

  Promise.allSettled = function (promises) {
    const convertedPromises = promises.map(p => Promise.resolve(p).then(resolveHanc
    return Promise.all(convertedPromises);
  };
}
```

In this code, `promises.map` takes input values, turns them into promises (just in case a non-promise was passed) with `p => Promise.resolve(p)`, and then adds `.then` handler to every one.

That handler turns a successful result `value` into `{status:'fulfilled', value}`, and an error `reason` into `{status:'rejected', reason}`. That's exactly the format of `Promise.allSettled`.

Now we can use `Promise.allSettled` to get the results of *all* given promises, even if some of them reject.

Promise.race

Similar to `Promise.all`, but waits only for the first settled promise and gets its result (or error).

The syntax is:

```
let promise = Promise.race(iterable);
```

For instance, here the result will be `1`:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, so it became the result. After the first settled promise "wins the race", all further results/errors are ignored.

Promise.any

Similar to `Promise.race`, but waits only for the first fulfilled promise and gets its result. If all of the given promises are rejected, then the returned promise is rejected with `AggregateError` – a special error object that stores all promise errors in its `errors` property.

The syntax is:

```
let promise = Promise.any(iterable);
```

For instance, here the result will be `1`:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 1
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, but it was rejected, so the second promise became the result. After the first fulfilled promise "wins the race", all further results are ignored.

Here's an example when all promises fail:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 100
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 20
]).catch(error => {
  console.log(error.constructor.name); // AggregateError
  console.log(error.errors[0]); // Error: Ouch!
  console.log(error.errors[1]); // Error: Error!
});
```

As you can see, error objects for failed promises are available in the `errors` property of the `AggregateError` object.

Promise.resolve/reject

Methods `Promise.resolve` and `Promise.reject` are rarely needed in modern code, because `async/await` syntax (we'll cover it a bit later) makes them somewhat obsolete.

We cover them here for completeness and for those who can't use `async/await` for some reason.

Promise.resolve

`Promise.resolve(value)` creates a resolved promise with the result `value`.

Same as:

```
let promise = new Promise(resolve => resolve(value));
```

The method is used for compatibility, when a function is expected to return a promise.

For example, the `loadCached` function below fetches a URL and remembers (caches) its content. For future calls with the same URL it immediately gets the previous content from cache, but uses `Promise.resolve` to make a promise of it, so the

returned value is always a promise:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

We can write `loadCached(url).then(…)`, because the function is guaranteed to return a promise. We can always use `.then` after `loadCached`. That's the purpose of `Promise.resolve` in the line `(*)`.

Promise.reject

`Promise.reject(error)` creates a rejected promise with `error`.

Same as:

```
let promise = new Promise((resolve, reject) => reject(error));
```

In practice, this method is almost never used.

Summary

There are 6 static methods of `Promise` class:

1. `Promise.all(promises)` – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, it becomes the error of `Promise.all`, and all other results are ignored.
2. `Promise.allSettled(promises)` (recently added method) – waits for all promises to settle and returns their results as an array of objects with:
   - `status`: `"fulfilled"` or `"rejected"`
   - `value` (if fulfilled) or `reason` (if rejected).
3. `Promise.race(promises)` – waits for the first promise to settle, and its result/error becomes the outcome.
4. `Promise.any(promises)` (recently added method) – waits for the first promise to fulfill, and its result becomes the outcome. If all of the given promises are rejected, `AggregateError` becomes the error of `Promise.any`.
5. `Promise.resolve(value)` – makes a resolved promise with the given value.
6. `Promise.reject(error)` – makes a rejected promise with the given error.

Of all these, `Promise.all` is probably the most common in practice.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)