Promise API

There are 4 static methods in the `Promise` class. We'll quickly cover their use cases here.

Promise.resolve

The syntax:

```
let promise = Promise.resolve(value);
```

Returns a resolved promise with the given `value`.

Same as:

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

The method is used when we already have a value, but would like to have it "wrapped" into a promise.

For instance, the `loadCached` function below fetches the `url` and remembers the result, so that future calls on the same URL return it immediately:

```
function loadCached(url) {
  let cache = loadCached.cache || (loadCached.cache = new Map());

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

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

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

Promise.reject

The syntax:

```
let promise = Promise.reject(error);
```

Create a rejected promise with the `error`.

Same as:

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

We cover it here for completeness, rarely used in real code.

Promise.all

The method to run many promises in parallel and wait till all of them are ready.

The syntax is:

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

It takes an `iterable` object with promises, technically it can be any iterable, but usually it's an array, and returns a new promise. The new promise resolves with when all of them are settled and has an array of their results.

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

Please note that the relative order is the same. Even though the first promise takes the longest time to resolve, it is 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 fetch(github url)
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 more real-life example with fetching user information for an array of github users by their names (or we could fetch an array of goods by their ids, the logic is same):

```
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 ready, we can show HTTP status codes
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // shows 200 for every url
    }

    return responses;
  })
  // map array of responses into 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, `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!")), 200
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

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

The important detail is that promises provide no way to "cancel" or "abort" their execution. So other promises continue to execute, and the eventually settle, but all their results are ignored.

There are ways to avoid this: we can either write additional code to `clearTimeout` (or otherwise cancel) the promises in case of an error, or we can make errors show up as members in the resulting array (see the task below this chapter about it).

`Promise.all(iterable)` allows non-promise items in `iterable`

Normally, `Promise.all(iterable)` accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's wrapped in `Promise.resolve`.

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

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

So we are able to pass non-promise values to `Promise.all` where convenient.

Promise.race

Similar to `Promise.all` takes an iterable of promises, but instead of waiting for all of them to finish – waits for the first result (or error), and goes on with it.

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!")), 200
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

So, the first result/error becomes the result of the whole `Promise.race`. After the first settled promise "wins the race", all further results/errors are ignored.

Summary

There are 4 static methods of `Promise` class:

1. `Promise.resolve(value)` – makes a resolved promise with the given value,
2. `Promise.reject(error)` – makes a rejected promise with the given error,
3. `Promise.all(promises)` – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, then it becomes the error of `Promise.all`, and all other results are ignored.
4. `Promise.race(promises)` – waits for the first promise to settle, and its result/error becomes the outcome.

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

Tasks

Fault-tolerant Promise.all

We'd like to fetch multiple URLs in parallel.

Here's the code to do that:

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

Promise.all(urls.map(url => fetch(url)))
  // for each response show its status
  .then(responses => { // (*)
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`);
    }
  ));
```

The problem is that if any of requests fails, then `Promise.all` rejects with the error, and we loose results of all the other requests.

That's not good.

Modify the code so that the array `responses` in the line `(*)` would include the response objects for successful fetches and error objects for failed ones.

For instance, if one of URLs is bad, then it should be like:

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

Promise.all(...) // your code to fetch URLs...
  // ...and pass fetch errors as members of the resulting array...
  .then(responses => {
    // 3 urls => 3 array members
    alert(responses[0].status); // 200
    alert(responses[1].status); // 200
    alert(responses[2]); // TypeError: failed to fetch (text may vary)
  });
```

P.S. In this task you don't have to load the full response using `response.text()` or `response.json()`. Just handle fetch errors the right way.

Open the sandbox for the task.

solution

Fault-tolerant fetch with JSON

Improve the solution of the previous task Fault-tolerant Promise.all. Now we need not just to call `fetch`, but to load the JSON objects from given URLs.

Here's the example code to do that:

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

// make fetch requests
Promise.all(urls.map(url => fetch(url)))
  // map each response to response.json()
  .then(responses => Promise.all(
    responses.map(r => r.json())
  ))
  // show name of each user
  .then(users => {  // (*)
    for(let user of users) {
      alert(user.name);
    }
  });
```

The problem is that if any of requests fails, then `Promise.all` rejects with the error, and we loose results of all the other requests. So the code above is not fault-tolerant, just like the one in the previous task.

Modify the code so that the array in the line `(*)` would include parsed JSON for successful requests and error for errored ones.

Please note that the error may occur both in `fetch` (if the network request fails) and in `response.json()` (if the response is invalid JSON). In both cases the error should become a member of the results object.

The sandbox has both of these cases.

Open the sandbox for the task.

solution

Comments

- You're welcome to post additions, questions to the articles and answers to them.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen...)
- If you can't understand something in the article – please elaborate.