Asynch JS: The Power Of $.Deferred

(/profiles/#jeremychone)
Published **Aug. 29, 2012**

SUPPORTED BROWSERS:

Table of Contents

Localizations

This article is additionally available in the following languages:

日本語 (/ja/tutorials/async/deferred/)
한국어 (/ko/tutorials/async/deferred/)

One of the most important aspects about building smooth and responsive HTML5 applications is the synchronization between all the different parts of the application such as data fetching, processing, animations, and user interface elements.

The main difference with a desktop or a native environment is that browsers do not give access to the threading model and provide a single thread for everything accessing the user interface (i.e. the DOM). This means that all the application logic accessing and modifying the user interface elements is always in the same thread, hence the importance of keeping all the application work units as small and efficient as possible and taking advantage of any asynchronous capabilities the browser offers as much as possible.

Browser Asynchronous APIs

Luckily, Browsers provide a number of asynchronous APIs such as the commonly used XHR (XMLHttpRequest or 'AJAX') APIs, as well as IndexedDB, SQLite, HTML5 Web workers, and the HTML5 GeoLocation APIs to name a few. Even some DOM related actions are exposed asynchronously, such CSS3 animation via the transitionEnd events.

The way browsers expose asynchronous programming to the application logic is via events or callbacks. In event-based asynchronous APIs, developers register an event handler for a given object (e.g. HTML Element or other DOM objects) and then call the action. The browser will perform the action usually in a different thread, and trigger the event in the main thread when appropriate.

For example, code using the XHR API, an event based asynchronous API, would look like this:

```
// Create the XHR object to do GET to
/data resource
var xhr = new XMLHttpRequest();
xhr.open("GET","data",true);

// register the event handler
xhr.addEventListener('load',function(){
  if(xhr.status === 200){
      alert("We got data: " +
xhr.response);
  }
},false)

// perform the work
xhr.send();
```

The CSS3 transitionEnd event is another example of a event based asynchronous API.

```
// get the html element with id
'flyingCar'
var flyingCarElem =
document.getElementById("flyingCar");

// register an event handler
// ('transitionEnd' for FireFox,
'webkitTransitionEnd' for webkit)
flyingCarElem.addEventListener("transit
ionEnd",function(){
  // will be called when the transition
has finished.
  alert("The car arrived");
});

// add the CSS3 class that will trigger
the animation
// Note: some browers delegate some
transitions to the GPU , but
//       developer does not and should
not have to care about it.
flyingCarElemen.classList.add('makeItFl
y')
```

Other browser APIs, such as SQLite and HTML5 Geolocation, are callback based, meaning that the developer passes a function as argument that will get called back by the underlying implementation with the corresponding resolution.

For example, for HTML5 Geolocation, the code looks like this:

```
// call and pass the function to
callback when done.
navigator.geolocation.getCurrentPositio
n(function(position){
            alert('Lat: ' +
position.coords.latitude + ' ' +
                  'Lon: ' +
position.coords.longitude);
});
```

In this case, we just call a method and pass a function that will get called back with the requested result. This allows the browser to implement this functionality synchronously or asynchronously and give a single API to the developer regardless of the implementation details.

Making Applications Asynchronous-Ready

Beyond the browser's built-in asynchronous APIs, well architected applications should expose their low level APIs in an asynchronous fashion as well, especially when they do any sort of I/O or computational heavy processing. For example, APIs to get data should be asynchronous, and should NOT look like this:

```
// WRONG: this will make the UI freeze
when getting the data
var data = getData();
alert("We got data: " + data);
```

This API design requires the **getData()** to be blocking, which will freeze the user interface until the data is fetched. If the data is local in the JavaScript context then this might not be an issue, however if the data needs to be fetched from the network or even locally in a SQLite or index store this could have dramatic impact on the user experience.

The right design is to proactively make all application API that could take some time to process, asynchronous from the beginning as retrofitting synchronous application code to be asynchronous can be a daunting task.

For example the simplistic getData() API would become something like:

```
getData(function(data){
   alert("We got data: " + data);
});
```

The nice thing about this approach is that this forces the application UI code to be asynchronous-centric from the beginning and allows the underlying APIs to decide if they need to be asynchronous or not in a later stage.

Note that not all the application API need or should be asynchronous. The rule of thumb is that any API that does any type of I/O or heavy processing (anything that can take longer than 15ms) should be exposed asynchronously from the start even if the first implementation is synchronous.

Handling Failures

One catch of asynchronous programing is that the traditional try/catch way to handle failures does not really work anymore, as errors usually happen in another thread. Consequently, the callee needs to have a structured way to notify the caller when something goes wrong during the processing.

In an event-based asynchronous API this is often accomplished by the application code querying the event or object when receiving the event. For callback based asynchronous APIs, the best practice is to have a second argument that takes a function that would be called in case of a failure with the appropriate error information as argument.

Our getData call would look like this:

```
// getData(successFunc,failFunc);
getData(function(data){
  alert("We got data: " + data);
}, function(ex){
  alert("oops, some problem occured: "
+ ex);
});
```

Putting it together with $.Deferred

One limitation of the above callback approach is that it can become really cumbersome to write even moderately advanced synchronization logic.

For example, if you need to wait for two asynchronous API to be done before doing a third one, code complexity can rise quickly.

```
// first do the get data.
getData(function(data){
  // then get the location
  getLocation(function(location){
    alert("we got data: " + data + "
and location: " + location);
  },function(ex){
    alert("getLocation failed: "  +
ex);
  });
},function(ex){
  alert("getData failed: " + ex);
});
```

Things can even get more complex when the application needs to make the same call from multiple parts of the application, as every call will have to perform these multi step calls, or the application will have to implement its own caching mechanism.

Luckily, there is a relatively old pattern, called Promises (kind of similar to Future (http://docs.oracle.com/javase/1.5.0/docs/api/java/util in Java) and a robust and modern implementation in jQuery core called $.Deferred (http://api.jquery.com/category/deferred-object/) that provides a simple and powerful solution to asynchronous programing.

To make it simple, the Promises pattern defines that the asynchronous API returns a Promise object which is kind of a “Promise that the result will be resolved with the corresponding data." To get the resolution, the caller gets the Promise object and calls a *done(successFunc(data))* which will tell the Promise object to call this *successFunc* when the "data" is resolved.

So, the getData call example above becomes like this:

```
// get the promise object for this API
var dataPromise = getData();

// register a function to get called
when the data is resolved
dataPromise.done(function(data){
  alert("We got data: " + data);
});

// register the failure function
dataPromise.fail(function(ex){
  alert("oops, some problem occured: "
+ ex);
});

// Note: we can have as many
dataPromise.done(...) as we want.
dataPromise.done(function(data){
  alert("We asked it twice, we get it
twice: " + data);
});
```

Here, we get the *dataPromise* object first and then call the *.done* method to register a function we want to get called back when the data gets resolved. We can also call the *.fail* method to handle the eventual failure. Note that we can have as many *.done* or *.fail* calls as we need since the underlying Promise implementation (jQuery code) will handle the registration and callbacks.

With this pattern, it is relatively easy to implement more advanced synchronization code, and jQuery already provides the most common one such a $.when (http://api.jquery.com/jQuery.when/) .

For example, the nested *getData/getLocation* callback above would become something like:

```
// assuming both getData and
getLocation return their respective
Promise
var combinedPromise = $.when(getData(),
getLocation())

// function will be called when both
getData and getLocation resolve
combinePromise.done(function(data,locat
ion){
  alert("We got data: " + dataResult +
" and location: " + location);
});
```

And the beauty of it all is that jQuery.Deferred makes it very easy for developers to implement the asynchronous function. For example, the getData could look something like this:

```
function getData(){
  // 1) create the jQuery Deferred
object that will be used
  var deferred = $.Deferred();

  // ---- AJAX Call ---- //
  XMLHttpRequest xhr = new
XMLHttpRequest();
  xhr.open("GET","data",true);

  // register the event handler

xhr.addEventListener('load',function(){
    if(xhr.status === 200){
      // 3.1) RESOLVE the DEFERRED
(this will trigger all the done()...)
      deferred.resolve(xhr.response);
    }else{
      // 3.2) REJECT the DEFERRED (this
will trigger all the fail()...)
      deferred.reject("HTTP error: " +
xhr.status);
    }
  },false)

  // perform the work
  xhr.send();
  // Note: could and should have used
jQuery.ajax.
  // Note: jQuery.ajax return Promise,
but it is always a good idea to wrap it
  //       with application semantic in
another Deferred/Promise
  // ---- /AJAX Call ---- //

  // 2) return the promise of this
deferred
  return deferred.promise();
}
```

So, when the getData() is called, it first creates a new jQuery.Deferred object (1) and then returns its Promise (2) so that the caller can register its done and fail functions. Then, when the XHR call returns, it either resolves the deferred (3.1) or reject it (3.2). Doing the deferred.resolve will trigger all the done(...) functions and other promise functions (e.g., then and pipe) and calling deferred.reject will call all the fail() functions.

Use Cases

Here are some good use cases where Deferred can be very useful:

Data Access: Exposing data access APIs as $.Deferred is often the right design. This is obvious for remote data, as synchronous remote calls would completely ruin the user experience, but is also true for local data as often the lower level APIs (e.g., SQLite and IndexedDB) are asynchronous themselves. The Deferred API''s $.when and .pipe are extremely powerful to synchronize and chain asynchronous sub-queries.

UI Animations: Orchestrating one or more animations with transitionEnd events can be quite tedious, especially when the animations are a mixed of CSS3 animation and JavaScript (as it is often the case). Wrapping the animation functions as Deferred can significantly reduce the code complexity and improve flexibility. Even a simple generic wrapper function like cssAnimation(className) that will return the Promise object that gets resolved on transitionEnd could be of a great help.

UI Component Display: This is a little bit more advanced, but advanced HTML Component frameworks should use the Deferred as well. Without going too much into the details (this will be the subject of another post), when an application needs to display different parts of the user interface, having the lifecycle of those components encapsulated in Deferred allows greater control of the timing.

Any browser asynchronous API: For normalization purpose, it is often a good idea to wrap the browser API calls as Deferred. This takes literally 4 to 5 lines of code each, but will greatly simplify any application code. As show in the above getData/getLocation pseudo code, this allows the applications code to have one asynchronous model across all types of API (browsers, application specifics, and compound).

Caching: This is kind of a side benefit, but can be very useful in some occasion. Because the Promise APIs (e.g., .done(..) and .fail(..)) can be called before or after the asynchronous call is performed, the Deferred object can be used as a caching handle for an asynchronous call. For example, a CacheManager could just keep track of Deferred for given requests, and return the Promise of the matching Deferred if it has not be invalidated. The beauty is that the caller does not have to know if the call has already been resolved or is in the process of being resolved, its callback function will get called exactly the same way.

Conclusion

While the $.Deferred concept is simple, it can take time to get a good handle on it. However, given the nature of the browser environment, mastering asynchronous programing in JavaScript is a must for any serious HTML5 application developer and the Promise pattern (and the jQuery implementation) are tremendous tools to make asynchronous programming reliable and powerful.

Avatar

Markus • a year ago

is there any means how to know in a .fail() listener of a combined promise, which promise actually failed:

e.g. in:
var combinedPromise = $.when(getData(), getLocation())
combinedPromise.fail(function(...)) {
// how to know which of the 2 promises failed?
}

3 ^ | v • Reply • Share ›

jeremychone → Markus • a year ago

Good question. $.when fail on the first .reject, and consequently, I think (I have not tested it) that the first argument you get on the combinedPromise.fail callback function would be the reject argument of the rejected deferred. So, if you can tell the fail arguments apart, then you know.

The alternative is to keep reference of each individual Promise and call isRejected().

^ | v • Reply • Share ›

Avatar

asabaylus • a year ago

Very clear and well explained thanks!

2 ^ | v • Reply • Share ›

Avatar

Andrew M. Andrews III • a year ago

Am I missing something, or is there a risk here that resolve() or reject() could run before a handler has been attached using done() or fail()?

2 ^ | v • Reply • Share ›

jeremychone → Andrew M. Andrews III • a year ago

No, this is part of the $.Deferred implementation. When calling, .done(), .fail(), .then(), .pipe(), you just register a callback function. If the Deferred was already resolved (meaning .resolve() or .reject() was called) the callback functions will be called immediately, otherwise, the callbacks will occur on the .resolve() or .reject().

make sense?

3 ^ | ⌄ • Reply • Share ›

Andrew M. Andrews III
➜ jeremychone
• a year ago

Makes sense, and it's certainly desirable behavior, although not apparent from the code :)

3 ^ | ⌄ • Reply • Share ›

Avatar anonymous • 8 months ago

where is scrollbar???

1 ^ | ⌄ • Reply • Share ›

Avatar Jonathan • 9 months ago

Instead of:

"... is that browsers do not give access to the threading model and provide a single thread for everything ..."

Use:

"... is that browsers do not give access to the threading model, but instead provide a single thread for everything ..."

1 ^ | ⌄ • Reply • Share ›

Avatar Liviu Bundă • 9 months ago

You have a small mistake in the "combinedPromise" example. The callback registered by done has a parameter called "data", yet in the function it is used as "dataResult".

1 ^ | ⌄ • Reply • Share ›

Avatar spacemonkey82 • a year ago

if i understand correctly, the deferred objects in $.when get executed asynchronously in parallel. what would i need to do to make them execute serially?

1 • Reply • Share ›

Ori spacemonkey82 • a year ago

Concurrently, not in parallel.

• Reply • Share ›

Jared Roberts spacemonkey82 • a year ago

use Deferred.pipe()

• Reply • Share ›

Avatar Ludomsix Ludomsix • 3 months ago

tst

• Reply • Share ›

Avatar Daniel Stern • 4 months ago

It looks like my comprehension of this topic has just been deferred. :)

• Reply • Share ›

Avatar Carlos Roberto Gomes Junior • 5 months ago

Good tutorial, thanks!

• Reply • Share ›

Avatar Finne • 7 months ago

Excellent article! I was confused about $.Deferred but this helped me understand it :)

• Reply • Share ›

Avatar Thibault Lenclos • a year ago

Very good explanation, thanks.

• Reply • Share ›

Avatar gosukiwi • a year ago