ClojureScript + JavaScript

2011 November 11

Kevin J. Lynagh — Keming Labs

Slides: keminglabs.com/talks/
Email: kevin@keminglabs.com
Office: +1 888 502 1042

This handout accompanies *Extending JavaScript Libraries from ClojureScript*, a talk at Clojure Conj 2011. A recording of the talk, slides, and this handout are available online. Contact Keming Labs if you need to visualize data, typeset mathematics, or require assistance finding beer in Portland, Oregon.

JavaScript doesn't have built-in namespace support; best practice is to define new functions within a closure to isolates them from rest of the page and minimize the chance of collisions. The following is an example of a jQuery plugin (taken from the official jQuery documentation):

```
(function( $ ){

  var methods = {
    init: function( options ) {},
    show: function( ) {},
    hide: function( ) {},
    update: function( content ) {}
  };

  $.fn.tooltip = function( method ) {
    if ( methods[method] ) {
      return methods[method].apply(
        this, Array.prototype.slice.call( arguments, 1 ));
    } else if ( typeof method === 'object' || ! method ) {
      return methods.init.apply( this, arguments );
    } else {
      $.error( 'Method ' +  method + ' does not exist!' );
    }
  };

})( jQuery );
```

Wrap everything in a closure to prevent collisions.

Inline all of your functions in a single object.

Add *one* function to jQuery, which dispatches to the methods defined above via a string argument (or calls `init` if an object of parameters is given).

The return value of this immediately-executed function is undefined. Instead, the function mutates the jQuery object (passed in as the argument `$`) to include the newly defined plugin.

Use the plugin by invoking its single method repeatedly:

```
$('div').tooltip({foo: 17})
  .tooltip('update', 'New tooltip content!');
```

Compare this to Clojure, where you can simply partition functions into groups via the namespace macro, `(ns)`:

```
(ns tooltip) ;;in src/cljs/tooltip.cljs
(defn init [$sel & {:keys [foo bar]
                    :or {foo 1, bar 2}}])
(defn show [$sel])
(defn hide [$sel])
(defn update [$sel new-content])

;;in src/cljs/your_namespace.cljs
(ns your-namespace
  (:require [tooltip :as tt]))

(let [$ js/jQuery]
  (tt/init ($ "div") :foo 17)
  (tt/update ($ "div") "New tooltip content!"))
```

Use destructuring to define keyword arguments with defaults on `init`. To maintain referential transparency, pass the selection, `$sel`, into each function. (contrast to jQuery, which relies on the `this` dynamic binding.)

You can use compile-time macros from ClojureScript. Just define them in a Clojure file on the classpath:

```
(ns demo.macros) ;;src/clj/demo/macros

(defmacro shim [name]
  "Define façade to host functions with arities 0-3"
  `(defn ~name
     ([sel#] (. sel# (~name)))
     ([sel# a1#] (. sel# ~name a1#))
     ([sel# a1# a2#] (. sel# ~name a1# a2#))
     ([sel# a1# a2# a3#] (. sel# ~name a1# a2# a3#))))
```

Why not use `apply` instead of writing out each arity? Many JavaScript libraries are written in an object-oriented style and rely on the `this` dynamic binding. When using `apply` to execute a function, the ClojureScript compiler generates code that changes `this`, leading to opaque bugs.

and then use `use-macros` or `require-macros` in the namespace form to bring them into your file:

```
(ns demo ;;in src/cljs/demo.cljs
  (:use-macros [demo.macros :only [shim]])
  (:require [clojure.set :as set]))

(defn p [x]
  "Print to the browser console."
  (.log js/console x))

(p (js* "this")) ;;=> DOMWindow object

(let [a (into #{} (range 10))
      b #{2 3 5 7 11}]
  (set/difference a b)) ;;=> #{0 1 4 6 8 9}
```

Interop as in other Clojures: the dot special form calls methods; `js/` refers to the JavaScript namespace. The `js*` function executes a string as JavaScript; use only in emergencies.

ClojureScript includes sets.

```
;;This atom will always point to an even number.
(def N (atom 0 :validator #(= 0 (mod % 2))))

;;Print to the console whenever this atom is updated.
(add-watch N :the-watcher
           (fn [key n old-val new-val]
             (p (str "N changed from " old-val " to " new-val))))

(reset! N 4) ;;This will work,
(try
  (reset! N 5) ;;but this won't.
  (catch js/Error e ;;No typed exceptions, just js/Error.
    (p (str "Can't do that; leaving atom as: " @N)))
  (finally (p "Keep going...")))
```

ClojureScript includes the atom reference type to handle mutable state. You can restrict the values that an atom references by assigning a validator function. Watcher functions on atoms are a great way to write event-driven programs *à la* Backbone.js.

```
(let [o (js-obj)]
  (aset o "x" 1)
  (set! (.y o) 2)
  (.stringify js/JSON o)) ;;=> "x":1,"y":2;

(apply array [1 2 3]) ;;=> [1,2,3];
(cljs.core.Vector/fromArray (js* "[1, 2, 3]")) ;;=> [1 2 3]
```

If you need to build a JavaScript object for interop, `(js-obj)` builds one and you can use `aset` or `set!` mutate it.

Likewise, `(array)` constructs a JavaScript object.

Further reading: Chris Granger's *Pinot* includes DOM helpers and a system for communicating with *Noir* web servers using Clojure datatypes (no JSON!). *Cljs-d3* wraps and extends the D3 data visualization library. Luke VanderHart explains how to use Google's Closure compiler with external JavaScript libraries and ClojureScript.

`github.com/ibdknox/pinot`
`github.com/ibdknox/noir`
`github.com/lynaghk/cljs-d3`
`lukevanderhart.com`