Client-Side Caterwaul

Spencer Tipping

November 26, 2010

Contents

Chapter 1

Introduction

Most JavaScript users are writing client-side applications, though node.js may change that. Caterwaul can be useful for this and plays well with jQuery and other common client-side libraries. This guide goes through some enhancements that you can make to your code when you use Lisp-style macros in conjunction with jQuery and jQuery UI.

1.1 Setup

Using Caterwaul is really easy. Most applications have a setup like this:

```
<script src='.../jquery.js'></script>
<script>
  $(function () {
    // application code
  });
</script>
```

If you wanted to use Caterwaul in the application code, you'd do this:

```
<script src='.../jquery.js'></script>
<script src='.../caterwaul.js'></script>
<script>
  // Yes, this looks weird, but it's how it works :)
  $(caterwaul.clone('std')(function () {
    // application code can now use macros
  }));
</script>
```

The latest development version of Caterwaul is always available at http://spencertipping.com/caterwaul/caterwaul.js, though if you have a production application you'll probably be better off deploying a snapshot with

your code.[1] You can also get stable versions from http://spencertipping.com/caterwaul/stable.

1.2 Running the examples

This guide contains lots of code examples that use Caterwaul. I recommend using the online Caterwaul compiler, up at http://spencertipping.com/caterwaul/compiler, to see the generated code.

1.3 Further reading

After using Caterwaul for some web development, I've taken some common macros and factored them into their own library. You can download this library at http://github.com/spencertipping/montenegro – the macros defined there are probably more complete than the ad-hoc macros that you're likely to define for a project.

I recommend becoming acquainted with Caterwaul's `seq` and `continuation` modules, as these make it easy to handle some common cases that come up in client-side development. (At some point I'll discuss how to use those modules in this guide.)

[1] Both to protect against breaking changes, which sometimes happen, and to protect against downtime from my webhost, which is a definite possibility.

Chapter 2

The `std` Library

Caterwaul comes with a bunch of useful macros accessible via the `std` configuration. I highly recommend importing them for use in your projects. In this chapter I'll go over some examples demonstrating how to use the `std` library in your code.

2.1 `fn` and `fb`

The most common thing you're likely to use in `std` is the `fn` macro, along with its companion macros `fb`, `fn_`, and `fb_`. These are abbreviations for common function patterns. For example, here's a common jQuery pattern:

```
$('input.validated').blur(function () {
  var t = $(this);
  $.getJSON('/validate', {value: t.val()}, function (result) {
    if (result.valid)
      t.removeClass('invalid');
    else
      t.addClass('invalid');
  });
});
```

The first thing to do is collapse the inner function. We can use `fn[]` to do it, sort of like this:

```
// Doesn't work, but it's the right idea:
$.getJSON('/validate', {value: t.val()}, fn[result][
  if (result.valid)
    t.removeClass('invalid');
  else
    t.addClass('invalid');
]);
```

The problem here is that *if* is a statement-mode construct; that is, you can't say things like `3 + if (foo) bar;`. We need to convert the body of the function into an expression, which is most easily achieved using `?:`.

```
$('input.validated').blur(function () {
  var t = $(this);
  $.getJSON('/validate', {value: t.val()}, fn[result][
    result.valid ? t.removeClass('invalid') : t.addClass('invalid')]);
});
```

Cool, so that's some code out of the way. But this is just the beginning. Notice that the AJAX closure isn't bound, so we have to define `t`. The `fb` macro was designed to get rid of this problem:

```
$('input.validated').blur(function () {
  $.getJSON('/validate', {value: $(this).val()}, fb[result][
    result.valid ? $(this).removeClass('invalid') : $(this).addClass('invalid')]);
});
```

`fb` is just like `fn`, except that it binds to the outer `this`. This means that `this` inside the body of an `fb` function will be the same value that it was outside the function, regardless of how it is called.

Now let's tackle the outer function. It's a perfect candidate for `fn[]` because it has just one statement and that statement can be interpreted as an expression. It's tempting to do this:

```
// Doesn't work; syntax error
$('input.validated').blur(fn[][
  $.getJSON(...)
]);
```

Unfortunately you can't follow an expression with empty braces or parens. Rather than defining a useless argument, Caterwaul gives you the `fn_` macro (and `fb_` when you want to preserve `this`):

```
$('input.validated').blur(fn_[
  $.getJSON('/validate', {value: $(this).val()}, fb[result][
    result.valid ? $(this).removeClass('invalid') : $(this).addClass('invalid')])]);
```

2.1.1 Defining constructors: `fc`

I found out not too long ago that it's problematic for constructor functions to return values. It's possible to write a non-returning function with `fn` by having it return `undefined`, but that's a lot of typing for something that's ultimately a simple requirement. Since version 0.4, the macro `fc[][]` (and correspondingly `fc_[]`) is available for writing non-returning functions:

```
var problem = fn[x][this.x = x];        // A bad constructor
var fixed   = fc[x][this.x = x];        // This one works
```

2.2 `let` and `where`

Continuing the previous example, let's suppose you are concerned about minimizing jQuery allocations and don't want to repeat the expression `$(this)` more than necessary. The easiest way to eliminate this repetition is to allocate a temporary variable, but `var` is statement-mode and won't work inside `fn` or `fn_`. There are a couple of macros in `std` to work around this, `let` and `where`. First, here's the code with a temporary variable:

```
// Doesn't work; var is statement-mode
$('input.validated').blur(fn_[
  var t = $(this);
  $.getJSON(...);
]);
```

Here's how to use `let` to achieve the same effect:

```
$('input.validated').blur(fn_[
  let[t = $(this)] in
  $.getJSON('/validate', {value: t.val()}, fn[result][
    result.valid ? t.removeClass('invalid') : t.addClass('invalid')])]);
```

`where` does the same thing, but with a different syntax:

```
$('input.validated').blur(fn_[
  $.getJSON('/validate', {value: t.val()}, fn[result][
    result.valid ? t.removeClass('invalid') : t.addClass('invalid')]),
  where[t = $(this)]]);
```

2.2.1 Caveats of `let` and `where`

`let` and `where` are not semantically equivalent to `var`. In particular, here are some cases where they behave differently:

```
var x1 = 4,
    y1 = x1;
y1              // -> 4

let[x2 = 4,
    y2 = x2] in
y2              // -> ReferenceError: x2 is undefined

y3, where[x3 = 4, y3 = x3]
                // -> ReferenceError: x3 is undefined
```

The reason has to do with how they're expanded. The above `let` and `where` expand into this:

```
// The let definition:
(function (x2, y2) {
  return y2
}).call(this, 4, x2)

// The where definition:
(function (x3, y3) {
  return y3
}).call(this, 4, x3)
```

Now the problem should be obvious; the values you're assigning are in the outer scope, but the variables don't come into existence until the inner scope.

2.2.2 The solution: `let*` and `where*`

These macros expand into `var` expansions, like this:

```
// let*[x = 5, y = x] in ... or
// ..., where*[x = 5, y = x]:
(function () {
  var x = 5, y = x;
  return ...;
}).call(this);
```

Because the expansions are placed in a `var` definition, you can also define recursive functions:[1]

```
let*[factorial = fn[n][n > 1 ? n * factorial(n - 1) : 1]] in factorial(5);
```

An alternative syntax is provided for both `let` and `let*`:

```
let[x = 5][x + 1]       // is the same as:
let[x = 5] in x + 1
```

This syntax has the advantage that you don't have to worry about the precedence of `in` relative to the right-hand side of the `let`.

2.2.3 Function shorthands

The factorial definition above can be shortened using a syntax similar to OCaml and Haskell (though without guards):

```
let*[factorial(n) = n > 1 ? n * factorial(n - 1) : 1] in factorial(5);
```

[1] You could do this before using the applicative variant of the Y combinator, but nobody really does this. Caterwaul also doesn't provide a fixed-point function, though in the future I imagine I'll write one for it.

This is called an lvalue-macro, since it modifies the behavior of assignments based on properties of the left-hand side. The builtin `std.lvalue` library provides function assignment, as used above. (By extension, curried assignments are allowed; for example, `let*[f(x)(y) = x + y] in ...`.) Javascript's grammar isn't wonderfully accepting of arbitrary values placed on the left-hand side of an assignment, but due to an old IE bug function calls are syntactically (though not semantically in normal Javascript environments) lvalues.[2]

Note that `std.lvalue` applies not only inside `let` and `where`, but across your code in general. For example:

```
var f;
f(x) = x + 1;
f(5)    // -> 6

f(x)(y) = x + y;
f(1)(2) // -> 3
```

Unfortunately it isn't possible to write `var f(x) = 10`, but the lvalue semantics would also apply there if it were.

A common use of function shorthands is to extend jQuery or to bind a callback, for example:

```
$.fn.how_many() = this.find('*').length;
var f;
f(x, y)() = console.log(x + y);
$('...').click(f(3, 4));
```

In this example, the line `f(x, y)() = ...` assigns a curried function to `f`. Caterwaul generates this for the assignment:

```
f = function (x, y) {
  return function () {
    return console.log(x + y);
  };
};
```

Note that saying something like `$.fn.foo()() = ...` won't do what you want. The reason is that the inner function loses its `this` binding, so you won't be able to refer to the jQuery object! I don't have a solution in mind for this yet, but it's high on my list of things to fix.

2.3 `/se` and `/re`

As of version 0.4, Caterwaul's standard library supports some nice ways of introducing side-effects. You would use these when you have an expression

[2] This is the one good thing IE did for the world.

that you want to reference more than once, but you don't want to introduce a new `let`-binding for it. The most common case is probably creating an object whose key is variable:

```
// In plain Javascript:
var object = function (k, v) {
  var result = {};
  result[k] = v;
  return result;
};

// Using a let:
var object = fn[k, v][let[result = {}][result[k] = v, result]];

// Using /se:
var object = fn[k, v][{} /se[_[k] = v]];
```

The expression `x /se[y]` expands into `let[_ = x][y, x]`. In other words, it introduces `y` as a side-effect and returns `x`. There's a corresponding right-handed side effect macro, `x /re[y]`, that expands into `let[_ = x][y]` – that is, it returns `y` instead of `x`. Side-effects are often more readable than allocating temporary variables, especially when writing prototyped functions:

```
var my_class = fc[x, y][this.x = x, this.y = y]
/se[_.prototype.toString() = '<#{this.x}, #{this.y}>'];
```

2.4 `std.string`

It's common to write code like this:

```
$('button').click(fn_[
  alert('You clicked on ' + $(this).text() + '!')]);
```

Part of the `std` library in Caterwaul is a macro that performs string interpolation, just like in Ruby. You can use it like this:

```
$('button').click(fn_[
  alert('You clicked on #{$(this).text()}!')]);
```

You can even embed strings and use regular expressions without escaping backslashes. The only things that don't work are:

1. Using the same style of quotation mark that was used to quote the string; for example, `'foo#{'bar'}'` won't parse correctly (it will cause a syntax error).

2. Using a close-brace in the interpolated expression; e.g. `'foo#{let[x = {foo: "bar"}] in x.foo}'`. In this case, the close-brace will terminate the string interpolation and you'll get a syntax error in the expanded code. At some point string interpolation may use a full lex/parse to avoid this problem, but that will be in a future major release.

2.5 `when` and `unless`

I like to use short-circuit logic for conditionals, but there are some times when you want Perl-style postfix conditional logic. A common case is when the condition slightly obscures the meaning of the code when it's placed first:

```
var f = fn[x][
  // Notice that there isn't a return statement.
  // The value of the expression is automatically returned.
  x !== null && x !== undefined && x.toLowerCase && x.toLowerCase()];
```

Much clearer is to say it this way:

```
var f = fn[x][
  x.toLowerCase(), when[x !== null && x !== undefined && x.toLowerCase]];
```

This lets you put the function's meaning first, leaving the exceptional cases to an aside. `unless` is also provided, which does the opposite:

```
var f = fn[x][
  x.toLowerCase(), unless[x === null || x === undefined || ! x.toLowerCase]];
```

Because the comma operator associates left, you can stack `when` and `unless`. However, they will be evaluated outside-in:

```
var f = fn[x][
  // The conditions below need to be in this order, since x being null or
  // undefined causes the toLowerCase check to fail.
  x.toLowerCase(), when[x.toLowerCase], unless[x === null || x === undefined]];
```

Chapter 3

Defining Macros

Caterwaul's standard macros can be useful, but the real point of having a macro-oriented compiler is being able to write your own. Caterwaul comes with several macro-defining macros[1] that make it easier to extend.

Before I get into the details of defining macros, here's an example where a macro definition is useful:

```
// Without macros:
$('.add').live('click', fn_[$(this).parents('.list').eq(0).append('foo')]);
$('.remove').live('click', fn_[$(this).parents('.removable').eq(0).remove()]);
```

Let's define a macro that will simplify this code (details in section 3.1):

```
defmacro[_ >c> _][fn[selector, handler][
  qs[$(_selector).live('click', fn_[_handler, where[t = $(this)]])].
    replace({_selector: selector, _handler: handler}),
  when[selector.is_string()]]];
```

(If you're using the latest build of Caterwaul, you can write this instead, with the caveat that non-strings will also be matched:)

```
defsubst[_selector >c> _handler]
        [$(_selector).live('click', fn_[_handler, where[t = $(this)]])];
```

Now we can write this:

```
'.add' >c> t.parents('.list').eq(0).append('foo');
'.remove' >c> t.parents('.removable').eq(0).remove();
```

[1] If this doesn't make sense, just pretend I didn't say it.

3.1 Patterns

There are two parts to a macro definition. The first is the pattern; this is just an expression that is used to match against syntax elsewhere. Underscores in an expression are treated as wildcards and the trees they match will be passed into your macroexpansion function. For the >c> macro above, for example, the pattern was `_ >c> _`. This locates syntax trees that look like `x > c > y`, where `x` and `y` are arbitrary expressions. The nice thing about structural macros is that the tree matching is precedence-aware; so, for instance, `3 + 4 >c> 5, 10` would result in `3 + 4` for the left match, and `5` for the right; `10` is not a part of that tree, since comma takes lower precedence than >.[2]

Here are the patterns for some builtin macros:

```
fn[_][_]          // _ can match a comma-delimited list of things, too.
fn_[_]            // ... or arbitrary expressions
fb[_][_]
fb_[_]

let[_] in _       // 'in' has the same precedence as '<' or '>'
_, where[_]
_, when[_]
_, unless[_]

defmacro[_][_]    // defmacro is a macro too!
```

Just a _ is used for string interpolation. The reason is that strings are atoms, and there isn't a way to specify properties about the tree when you're writing a pattern for it. So the string interpolation macro ends up visiting every node in the tree, even though it only modifies strings that contain `#{}` blocks.

3.2 Expander functions

Once you have the pattern in place, you need a function to receive the matched fragments. The function is called whenever a matching tree is found in your source code, and it returns a new syntax tree just like a Lisp macro would.[3]

3.2.1 `qs[]`

`qs[]` is the simplest way to quote an expression. A quoted expression doesn't get evaluated; instead, it gets returned as a syntax tree. Syntax trees define some

[2] Combining operators around an identifier like I did here is safe as long as each operator symbol has equal precedence. For my own sanity, I like to use the same operator on each side. Associativity isn't an issue, by the way – binary operators with equal precedence always have the same assocativity (otherwise the grammar would be ambiguous).

[3] Conveniently, your expander function can decline to match by returning a falsy value. This can be handy when you want to test for conditions that a pattern won't detect.

useful operations, perhaps most importantly the `replace()` method, which lets you replace pieces of a tree. Let's write a pathological macro that replaces every addition with a multiplication:

```
defmacro[_ + _][fn[left, right][qs[l * r].replace({l: left, r: right})]];
```

And that's it. Each instance of "l" in the quoted form is replaced by the `left` that we got from the pattern match, and "r" with `right`.

3.2.2 Using `when` and `unless` in expanders

`when` and `unless` are totally compatible with macros. For example:

```
defmacro[_ + _][fn[left, right][qs[l * r].replace({l: left, r: right}),
                                unless[left.is_string() || right.is_string()]]];
```

If you return something falsy from a macro function, it will leave the syntax alone. Both `when` and `unless` return falsy values when their conditions fail.

I highly recommend reading the Caterwaul annotated source, available at http://spencertipping.com/caterwaul/caterwaul.html, for an in-depth look at how Caterwaul works and how to define more sophisticated macros (this section is just the tip of the iceberg).

3.3 Binding variables (dynamic scoping)

Asking programming language enthusiasts about dynamic scoping is like asking them about computed `GOTO`. You'll often get a very polarized response, and with good reason – many otherwise good languages have been held back by their reliance on such features. However, as with most things, there are times when it's useful to be able to define variables outside of the lexical scope chain, and I'll go over how to do it using macros.

The basic idea is simple. In Lisp you can write a (non-hygienic) macro like this:

```
(defmacro let-x-be-5-in (expression)
  `(let ((x 5)) ,expression))

(let-x-be-5-in (+ x 6))   ; -> 11
```

You can imagine how easily this can produce unreadable code if used ubiquitously, but if your variable name is very predictable it's nice to eliminate the duplication involved in creating it. A common case is a jQuery event handler, where the two relevant variables are `e` and `this` (well, really `$(this)`). Let's take some jQuery code and write macros to make it shorter:

```
$('a.foo').click(fn[e][$('p').append('foo')]);
$('a.bar').click(fn[e][$(this).parents('p').eq(0).find('a.bar').remove()]);

$('input').keyup(fn[e][send_to_server($(this).val()),
                       when[e.keyCode === 13]]);
```

An obvious bit of common code is the `fn[e]` that we keep writing. Another is `$(this)`. But stepping back, there's a high-level pattern:

```
$(<selector>).<event>(fn[e][<body>]);
```

If we assume that we're going to need `$(this)` in the `<body>` above, then we really have something closer to this:

```
$(<selector>).<event>(fn[e][<body>, where[t = $(this)]]);
```

What should the event look like? I'm partial to the `x >letters> y` syntax, and I think it suits jQuery code well. Let's implement it this way first, and then I'll go over some alternatives:

```
defmacro[_ >_> _][fn[selector, event, body][
  qs[$(_selector)._event_name(fn[e][_body, where[t = $(this)]])].
    replace({_selector: selector, _event_name: event_name, _body: body}),
  when[event_name && selector.is_string()],
  where[event_name = event.data === 'c' ? 'click' :
                     event.data === 'k' ? 'keyup' : null]]];
```

With this macro, the code from earlier becomes:

```
'a.foo' >c> $('p').append('foo');
'a.bar' >c> t.parents('p').eq(0).find('a.bar').remove();

// Parens required here, since comma is lower precedence than >
'input' >k> (send_to_server(t.val()), when[e.keyCode === 13]);
```

This last line is a good reason to reconsider how we're defining this macro. Generally you want to use an infix macro when both macro arguments are comprised entirely of high-precedence operators, but conditionals use comma or short-circuit logic and have very low precedence.[4] We can reuse the macro expander from above, but change the pattern to be grouped instead of infix:

```
defmacro[_._[_]][fn[selector, event, body][...]];
```

Now our code is this:

```
'a.foo'.c[$('p').append('foo')];
'a.bar'.c[t.parents('p').eq(0).find('a.bar').remove()];

'input'.k[send_to_server(t.val()), when[e.keyCode === 13]];
```

[4] If you're going to be writing many infix macros, I recommend becoming very familiar with Javascript's operator precedence and associativity.