Client-Side Caterwaul

Spencer Tipping

November 1, 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. This shows you the compiled functions that Caterwaul produces from your code.

[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')])]);
```

2.2 `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.

2.3 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.

2.4 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[$(_).live('click', fn_[_, where[t = $(this)]])].s('_', [selector, handler]),
  when[selector.is_string()]]];
```

Now we can write this:

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

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

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

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.

[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).