The tiny table sorter – or – you can write LINQ in JavaScript

devblogs.microsoft.com/oldnewthing/20130819-00

August 19, 2013

Raymond Chen

I had a little side project that displayed status information in a table, and I figured, hey, let me add sorting. And it was a lot easier than I thought. I just put the header row in the `THEAD` and the table contents in the `TBODY`, then I could use this code to sort the table:

```
function sortByColumn(table, sortCol, direction) {
 direction = direction || 1; // default sort ascending
 var tBody = table.tBodies[0];
 Array.prototype.map.call(tBody.rows, function (row) {
   var cell = row.cells[sortCol];
   return { row: row, key: cell.sortKey || cell.innerText };
 }).sort(function (a, b) {
   if (a.key < b.key) return -direction;
   if (a.key > b.key) return direction;
   return 0;
 }).forEach(function (o) {
   tBody.appendChild(o.row);
 });
}
```

Each cell can have an optional `sortKey` custom attribute which specifies how the item should sort. If there is no `sortKey`, then I just use the cell's `innerText`. (My table was constructed at runtime from an `XmlHttpRequest`, so adding the `sortKey` to the date fields was not difficult.)

One handy thing about the functions in the `Array` prototype is that as a rule, they do not actually require that the `this` object be an array. As long as it has a `length` property and integer subscripts, you can use it as if it were an array. The `map` function is okay with read-only access; some other function like `sort` require read-write access. To call a function with a custom `this` parameter, you use the `call` method on the function object itself, passing the artificial `this` as the first parameter, with the remaining parameters following.

First, the `sortByColumn` function takes the rows of the table body and `map` s each one to a record consisting of the sort key and the original row. The sort key is the `sortKey` property, if true-ish, we will use it; otherwise, we use the text of the cell.

I took a few shortcuts here. Depending on your browser, you may need to use `textContent` instead of `innerText`, and you may need to use `getAttribute` instead of property notation. And my function doesn't handle the case where the sort key is defined but is false-ish. Here's a more general version:

```
var textProperty = table.innerText ? "innerText" : "textContent";
...
   return { row: row,
            key: cell.hasAttribute("sortKey") ?
                 cell.getAttribute(sortKey") :
                 cell[textProperty] };
...
```

Anyway, after we map the rows to an array of sort records, we sort the records by comparing the `key`, either by string or by number. The code assumes that every column is either all-strings or all-numbers; it doesn't try to handle the mixed case. This is easy to enforce in the code that generates the table because the only way to get a non-string as a sort key is to set it explicitly as the `sortKey` attribute.

Finally, we take the sorted records and insert the sorted rows back into the table.

This is a common programming pattern: Decorate, operate, undecorate.[1] We started with a bunch of rows, and we wanted to sort them. We can't sort rows directly, so instead we converted the rows into something we *can* sort, but remembered the row that each converted item came from. We then perform the sort operation, and then recover the original rows from the decoration, now in sorted order, which we can then use for whatever operation we really wanted. I sort of combined the last two step into one. More formally, it would look like this:

```
function sortByColumn(table, sortCol, direction) {
 direction = direction || 1; // default sort ascending
 var tBody = table.tBodies[0];
 // decorate: convert the row into a record
 Array.prototype.map.call(tBody.rows, function (row) {
   var cell = row.cells[sortCol];
   return { row: row, key: cell.sortKey || cell.innerText };
 })
 // operate on the record
 .sort(function (a, b) {
   if (a.key < b.key) return -direction;
   if (a.key > b.key) return direction;
   return 0;
 })
 // undecorate: convert the record back into a row
 .map(function (o) {
   return o.row;
 })
 // operate on the sorted rows
 .forEach(function (r) {
   tBody.appendChild(r);
 });
}
```

Category theorists I'm sure have some fancy names they can use to describe this concept, like *natural transformation* and *functor category* and *splitting*.

LINQ also has a fancy name for this: *let*, which is a special case of *select* where LINQ generates the record for you.

LINQ let query	`from d in data let y = f(d.xValue)`
LINQ query	`from d in data select new { d = d, y = f(d.xValue) }`
LINQ fluent	`data.Select(d => new { d = d, y = f(d.xValue) })`
LINQ fluent old delegate syntax	`data.Select(delegate(Data d) { return new { d = d, y = f(d.xValue) }; })`
JavaScript	`data.map(function (d) { return { d: d, y: f(d.xValue) }; })`

JavaScript's `map` is the same as LINQ's *Select*, just with different decorative bits.

```
data.Select(delegate(Data d) { return new { d = d, y = f(d.xValue) }; })
```

```
data.map   (function(     d) { return     { d:  d, y:  f(d.xValue) }; })
```

Similarly, JavaScript's `filter` is the same as LINQ's *Where*, JavaScript's `some` is the same as LINQ's *Any*, JavaScript's `every` is the same as LINQ's *All*, and JavaScript's `reduce` is the same as LINQ's *Aggregate*. JavaScript's `sort` is sort of like LINQ's *Sort*, except that it modifies the array in place rather than generating a new result.

Bonus chatter: In theory, I could've just sorted the table directly by doing the sort key extraction inside the comparator:

```
function sortByColumn(table, sortCol, direction) {
 direction = direction || 1; // default sort ascending
 var tBody = table.tBodies[0];
 Array.prototype.map.call(tBody.rows, function (r) {
   return r;
 }).sort(function (a, b) {
   var keyA = a.cells[sortCol].sortKey || a.cells[sortCol].innerText;
   var keyB = b.cells[sortCol].sortKey || b.cells[sortCol].innerText;
   if (keyA < keyB) return -direction;
   if (keyA > keyB) return direction;
   return 0;
 }).forEach(function (r) {
   tBody.appendChild(r);
 });
}
```

but since I had to convert the rows into an array anyway (since you cannot modify the `rows` property by subscript assignment), I figured I'd do the extracting while I was there.

I guess I could've added a LINQy sort method:

```
function defaultComparator(a, b) {
  if (a < b) return -1;
  if (a > b) return 1;
  return 0;
}
Array.prototype.orderBy =
function Array_orderBy(extractKey, comparator, direction) {
  direction = direction || 1;
  comparator = comparator || defaultComparator;
  return Array.prototype.map.call(this, function (d) {
    return { key: extractKey.call(d), original: d };
  }).sort(function (a, b) {
    return direction * comparator(a.key, b.key);
  }).map(function (r) {
    return r.original;
  });
};
```

Then my `sortByColumn` function would just be

```
function sortByColumn(table, sortCol, direction) {
 direction = direction || 1; // default sort ascending
 var tBody = table.tBodies[0];
 Array.prototype.orderBy.call(tBody.rows, function (r) {
   var cell = r.cells[sortCol];
   return { key: cell.sortKey || cell.innerText, row: r };
 }, direction).forEach(function (r) {
   tBody.appendChild(r);
 });
}
```

But if I had done that, I wouldn't have had a cute one-function table sorter!

[1] In perl, this pattern is known as the Schwartzian transform. I prefer to think of it as completing the commutative diagram:

$$
\begin{array}{ccc}
 & g & \\
\hline
B & \rightarrow & B \\
\hline
f \downarrow & & \downarrow f \\
\hline
A & \dashrightarrow & A \\
\hline
 & f \circ g \circ f^{-1} &
\end{array}
$$

Mathematicians get all excited when they see something of the form $f \circ g \circ f^{-1}$: That's the form of a conjugation operation. Which makes sense, because conjugation is a way of looking at an algebraic group through different-colored glasses. In our case, the magic glasses make every row look like its sort key.

Bonus chatter: $linq is a Javascript LINQ library.

Raymond Chen

Follow