Programming(https://www.sitepoint.com/programming/) **Article**

Implementing Memoization in JavaScript

By Colin Ihrig (https://www.sitepoint.com/author/cjihrig/) August 23, 2012

Programs often waste time calling functions which recalculate the same results over and over again. This is particularly true with recursive and mathematical functions. A perfect example of this is the Fibonacci number (http://en.wikipedia.org/wiki/Fibonacci_number) generator. The Fibonacci sequence is a series of integers, beginning with zero and one, in which each value is the sum of the previous two numbers in the series. Based on this definition, the first ten Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. From a programming perspective, the *n*th Fibonacci number is typically computed recursively using the following function.

```
function fibonacci(n) {
  if (n === 0 || n === 1)
    return n;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

This function performs well for small values of “n”. However, performance quickly degrades as “n” increases. This is because the two recursive calls repeat the same work. For example, to compute the 50th Fibonacci number, the recursive function must be called over 40 billion times (40,730,022,147 times to be specific)! To make matters worse, computing the 51st number requires this work to be duplicated nearly two full times. This problem of repeating work could be mitigated if the function remembered what it had previously computed.

Memoization Basics

Memoization (http://en.wikipedia.org/wiki/Memoization) is a programming technique which attempts to increase a function’s performance by caching its previously computed results. Because JavaScript objects behave like associative arrays, they are ideal candidates to act as caches. Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned, without executing the entire function. However, if the data is not cached, then the function is executed, and the result is added to the cache.

In the following example, the original Fibonacci function is rewritten to include memoization. In the example, a self-executing anonymous function returns an inner function, f(), which is used as the Fibonacci function. When f() is returned, its closure allows it to continue to access the “memo” object, which stores all of its previous results. Each time f() is executed, it first checks to see if a result exists for the current value of “n”. If it does, then the cached value is returned. Otherwise, the original Fibonacci code is executed. Note that “memo” is defined outside of f() so that it can retain its value over multiple function calls. Recall that the original recursive function was called over 40 billion times to compute the 50th Fibonacci number. By implementing memoization, this number drops to 99.

```
var fibonacci = (function() {
  var memo = {};

  function f(n) {
    var value;

    if (n in memo) {
      value = memo[n];
    } else {
      if (n === 0 || n === 1)
        value = n;
      else
        value = f(n - 1) + f(n - 2);

      memo[n] = value;
    }

    return value;
  }

  return f;
})();
```

Handling Multiple Arguments

In the previous example, the function accepted a single argument. This made implementing the cache fairly trivial. Unfortunately, most functions require multiple arguments, which complicates the indexing of the cache. To memoize a function with multiple arguments, either the cache must become multi-dimensional, or all of the arguments must be combined to form a single index.

In a multi-dimensional approach, the cache becomes a hierarchy of objects instead of a single object. Each dimension is then indexed by a single parameter. The following example implements a multi-dimensional cache for the Fibonacci function. In this example, the function accepts an additional argument, “x”, which does nothing. Each time the function is invoked, the code checks that the “x” dimension exists, and initializes it if it does not exist. From that point forward, the “x” dimension is used to cache the “n” values. The result is that the function calls fibonacci(“foo”, 3) and fibonacci(“bar”, 3) are not treated as the same result.

```
var fibonacci = (function() {
  var memo = {};

  function f(x, n) {
    var value;

    memo[x] = memo[x] || {};

    if (x in memo && n in memo[x]) {
      value = memo[x][n];
    } else {
      if (n === 0 || n === 1)
        value = n;
      else
        value = f(x, n - 1) + f(x, n - 2);

      memo[x][n] = value;
    }

    return value;
  }

  return f;
})();
```

The alternative to a multi-dimensional cache is a single cache object which is indexed by a combination of all of the function’s arguments. Under this approach, the arguments are transformed into an array and then used to index the cache. Each function has a built in object named “arguments (https://developer.mozilla.org/en/JavaScript/Reference/Functions_and_function_scope/arguments)” which contains the arguments which were passed in. “arguments” is a type of object known as an Array-like object (http://nfriedly.com/techblog/2009/06/advanced-javascript-objects-arrays-and-array-like-objects/). It is similar to an array, but cannot be used to index the cache. Therefore, it must first be transformed into an actual array. This can be done using the array slice() method. The array representation can then be used to index the cache as shown before. The following example shows how this is accomplished. Note that an additional variable, “slice”, is defined as a reference to the array slice() method. By storing this reference, the overhead of repeatedly computing Array.prototype.slice() can be avoided. The call() method is then used to apply slice() to “arguments”.

```
var fibonacci = (function() {
  var memo = {};
  var slice = Array.prototype.slice;

  function f(x, n) {
    var args = slice.call(arguments);
    var value;

    if (args in memo) {
      value = memo[args];
    } else {
      if (n === 0 || n === 1)
        value = n;
      else
        value = f(x, n - 1) + f(x, n - 2);

      memo[arguments] = value;
    }

    return value;
  }

  return f;
})();
```

Caching Object Arguments

The memoization scheme presented here does not handle object arguments well. When objects are used as an index, they are first converted to a string representation such as "[object Object]". This causes multiple objects to incorrectly map to the same cache location. This behavior can be corrected by performing stringification (http://cjihrig.com/blog/native-browser-support-for-json-part-2-of-2/) on object arguments prior to indexing. Unfortunately, this also slows down the memoization process. The following example creates a generic memoized function which takes an object as a parameter. Note that the object argument is stringified using JSON.stringify() (https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/JSON/stringify) in order to create an index into the cache.

```
var foo = (function() {
  var memo = {};

  function f(obj) {
    var index = JSON.stringify(obj);

    if (index in memo) {
      return memo[index];
    } else {
      // memoized function contents
      return (memo[index] = function_value);
    }

  }

  return f;
})();
```

Automatic Memoization

In all of the previous examples, the functions were explicitly modified to add memoization. It is also possible to implement a memoization infrastructure without modifying the functions at all. This is useful because it allows the function logic to be implemented separately from the memoization logic. This is done by creating a utility function which takes a function as input and applies memoization to it. The following memoize() function takes a function, "func", as input. memoize() returns a new function which wraps a caching mechanism around "func". Note that this function does not handle object arguments. In order to handle objects, a loop is required which would inspect each argument individually and stringify as needed.

```
function memoize(func) {
  var memo = {};
  var slice = Array.prototype.slice;

  return function() {
    var args = slice.call(arguments);

    if (args in memo)
      return memo[args];
    else
      return (memo[args] = func.apply(this, args));

  }
}
```

Limitations

There are several things which must be kept in mind when implementing memoization. First, by storing old results, memoized functions consume additional memory. In the Fibonacci example, the additional memory consumption is unbounded. If memory usage is a concern, then a fixed size cache should be used. The overhead associated with memoization can also make it impractical for functions with execute quickly or that are executed infrequently.

The biggest limitation of memoization is that it can only be automated with functions that are *referentially transparent (http://en.wikipedia.org/wiki/Referential_transparency_(computer_science))*. A function is considered referentially transparent if its output depends only on its inputs, and it does not cause any side effects. A call to a referentially transparent function can be replaced by its return value without changing the semantics of the program. The Fibonacci function is referentially transparent because it depends solely on the value of "n". In the following example, the function foo() is not referentially transparent because it uses a global variable, "bar". Since "bar" can be modified outside of foo(), there is no guarantee that the return value will remain the same for each input value. In this example, the two calls to foo() return the values two and three, even though the same arguments are passed to both calls.

```
var bar = 1;

function foo(baz) {
  return baz + bar;
}

foo(1);
bar++;
foo(1);
```

Things to Remember

Memoization can potentially increase performance by caching the results of previous function calls.

Memoized functions store a cache which is indexed by their input arguments. If the arguments exist in the cache, then the cached value is returned. Otherwise, the function is executed and the newly computed value is added to the cache.

Object arguments should be stringified before using as an index.

Memoization can be automatically applied to referentially transparent functions.

Memoization may not be ideal for infrequently called or fast executing functions.

Was this helpful?

More: arrays (https://www.sitepoint.com/tag/arrays/), caching (https://www.sitepoint.com/tag/caching-ruby/), closures (https://www.sitepoint.com/tag/closures/), Fibonacci number (https://www.sitepoint.com/tag/fibonacci-number/), memoization (https://www.sitepoint.com/tag/memoization/)