Writing Custom Iterators

Steve Weston

November 26, 2018

1 Introduction

An *iterator* is a special type of object that supplies data on demand, one element[1] at a time. This is a nice abstraction that can help simplify many programs. Iterators are particularly useful in parallel computing, since they facilitate splitting a problem into smaller pieces that can then be executed in parallel.

Iterators can also be used to reduce the total memory that is needed at any one time. For example, if you want to process the lines of text in a file, it is common to write a loop that reads the file one line at a time, rather than reading the entire file in order to avoid running out of memory on huge files. That's the basic idea of iterators. Iterators provide a standard method for getting the next element, which allows us to write functions that take an iterator as an argument to provide a source of data. The function doesn't need to know what kind of iterator it is. It just needs to know how to get another piece of data. The data could be coming from a file, a database, a vector, or it could be dynamically generated.

There are a number of iterators that come in the `iterators` package. The `iapply` function allows you to iterate over arrays, in much the same way as the standard `apply` function. `apply` has fixed rules on how the results are returned, which may require you to reshape the results, which can be inefficient, as well as inconvenient. But since `iapply` doesn't process any data or combine the results, it is more flexible. You can use `iapply` with the `foreach` package to perform a parallel `apply` operation, and combine the results any way you want via the `.combine` argument to `foreach`.

Another iterator that comes in the `iterators` package is the `isplit` function, which works much like the standard `split` function. `split` returns a list containing all of the data divided into groups. `isplit` only generates one group at a time, as they are needed, which can reduce the amount memory that is needed.

But of course, there will be times when you need an iterator that isn't provided by the `iterators`

[1] An "element" in this case can be basically any object. I don't mean to suggest that the data is necessarily returned as scalar values, for example.

package. That is when you need to write your own custom iterator. Fortunately, that is fairly easy to do.

2 What methods are needed for an iterator?

Basically, an iterator is an S3 object whose base class is `iter`, and has `iter` and `nextElem` methods. The purpose of the `iter` method is to return an iterator for the specified object. For iterators, that usually just means returning itself, which seems odd at first. But the `iter` method can be defined for other objects that don't define a `nextElem` method. We call those objects *iterables*, meaning that you can iterate over them. The `iterators` package defines `iter` methods for vectors, lists, matrices, and data frames, making those objects iterables. By defining an `iter` method for iterators, they can be used in the same context as an iterable, which can be convenient. For example, the `foreach` function takes iterables as arguments. It calls the `iter` method on those arguments in order to create iterators for them. By defining the `iter` method for all iterators, we can pass iterators to `foreach` that we created using any method we choose. Thus, we can pass vectors, lists, or iterators to `foreach`, and they are all processed by `foreach` in exactly the same way.

The `iterators` package comes with an `iter` method defined for the `iter` class that simply returns itself. That is usually all that is needed for an iterator. However, if you want to create an iterator for some existing class, you can do that by writing an `iter` method that returns an appropriate iterator. That will allow you to pass an instance of your class to `foreach`, which will automatically convert it into an iterator. The alternative is to write your own function that takes arbitrary arguments, and returns an iterator. You can choose whichever method is most natural.

The most important method required for iterators is `nextElem`. This simply returns the next value, or throws an error. Calling the `stop` function with the string `'StopIteration'` indicates that there are no more values available in the iterator.

Now before we write our own iterator, let's try calling the `iter` and `nextElem` methods on an existing one. Since a list is an iterable, we can create an iterator for that list by calling `iter` on it:

```
> it <- iter(list(1:2, 3:4))
```

We can now call `nextElem` on the resulting iterator to get the values from the list:

```
> nextElem(it)
```

```
[1] 1 2
```

```
> nextElem(it)
```

```
[1] 3 4
```

```
> tryCatch(nextElem(it), error=function(e) e)
```

```
<simpleError: StopIteration>
```

As you can see, it is possible to call these methods manually, but it's somewhat awkward, since you have to handle the `'StopIteration'` error. Later on, we'll see one solution to this difficulty, although, in general, you don't call these method explicitly.

3 A simple iterator

It's time to show the implementation of a very simple iterator. Although I've made it sound like you have to write your own `iter` and `nextElem` methods, you can inherit them. In fact, that's what all of the following examples do. I do that by inheriting from the `abstractiter` class. The `abstractiter` class uses the standard `iter` method which returns itself, and defines a `nextElem` method that calls the `nextElem` element of the object. Let's take a look at the implementation of these two methods:

```
> iterators:::iter.iter
```

```
function (obj, ...)
{
    obj
}
<bytecode: 0x56406d042f58>
<environment: namespace:iterators>
```

```
> iterators:::nextElem.abstractiter
```

```
function (obj, ...)
{
    obj$nextElem()
}
<bytecode: 0x56406eaa0fb0>
<environment: namespace:iterators>
```

Now here's a function that creates a very simple iterator that uses these two methods:

```
> iforever <- function(x) {
+   nextEl <- function() x
+   obj <- list(nextElem=nextEl)
+   class(obj) <- c('iforever', 'abstractiter', 'iter')
+   obj
+ }
```

Note that I called the internal function `nextEl` rather than `nextElem`. I do that by convention to avoid masking the standard `nextElem` generic function. That causes problems when you want your iterator to call the `nextElem` method of another iterator, which can be quite useful, as we'll see in a later example.

We create an instance of this iterator by calling the `iforever` function, and then use it by calling the `nextElem` method on the resulting object:

```
> it <- iforever(42)
> nextElem(it)
```

```
[1] 42
```

```
> nextElem(it)
```

```
[1] 42
```

You can also get values from an iterator using `as.list`. But since this is an infinite iterator, you need to use the `n` argument to avoid using up a lot of memory and time:

```
> unlist(as.list(it, n=6))
```

```
[1] 42 42 42 42 42 42
```

Notice that it doesn't make sense to implement this iterator by defining a new `iter` method, since there is no natural iterable on which to dispatch. The only argument that we need is the object for the iterator to return, which can be of any type. Instead, we implement this iterator by defining a normal function that returns the iterator.

This iterator is quite simple to implement, and possibly even useful.[2] The iterator returned by `iforever` is a list that has a single element named `nextElem`, whose value is a function that returns

[2]Be careful how you use this iterator! If you pass it to `foreach`, it will result in an infinite loop unless you pair it with a non-infinite iterator. Also, *never* pass this to the `as.list` function without the `n` argument.

the value of `x`. Because we are subclassing `abstractiter`, we inherit a `nextElem` method that will call this function, and because we are subclassing `iter`, we inherit an `iter` method that will return itself.

Of course, the reason this iterator is so simple is because it doesn't contain any state. Most iterators need to contain some state, or it will be difficult to make it return different values and eventually stop. Managing the state is usually the real trick to writing iterators.

4 A stateful iterator

Let's modify the previous iterator to put a limit on the number of values that it returns. I'll call the new function `irep`, and give it another argument called `times`:

```
> irep <- function(x, times) {
+   nextEl <- function() {
+     if (times > 0)
+       times <<- times - 1
+     else
+       stop('StopIteration')
+
+     x
+   }
+
+   obj <- list(nextElem=nextEl)
+   class(obj) <- c('irep', 'abstractiter', 'iter')
+   obj
+ }
```

Now let's try it out:

```
> it <- irep(7, 6)
> unlist(as.list(it))
```

```
[1] 7 7 7 7 7 7
```

The real difference between `iforever` and `irep` is in the function that gets called by the `nextElem` method. This function not only accesses the values of the variables `x` and `times`, but it also modifies the value of `times`. This is accomplished by means of the "`<<-`" [3] operator, and the magic of lexical

[3] It's commonly believed that "<< −" is only used to set variables in the global environment, but that isn't true. I think of it as an *inheriting* assignment operator.

scoping. Technically, this kind of function is called a *closure*, and is a somewhat advanced feature of R. The important thing to remember is that nextEl is able to get the value of variables that were passed as arguments to irep, and it can modify those values using the "<<-" operator. These are *not* global variables: they are defined in the enclosing environment of the nextEl function. You can create as many iterators as you want using the irep function, and they will all work as expected without conflicts.

Note that this iterator only uses the arguments to irep to store its state. If any other state variables are needed, they can be defined anywhere inside the irep function.

5 Using an iterator inside an iterator

The previous section described a general way of writing custom iterators. Almost any iterator can be written using those basic techniques. At times, it may be simpler to make use of an existing iterator to implement a new iterator. Let's say that you need an iterator that splits a vector into subvectors. That can allow you to process the vector in parallel, but still use vector operations, which is essential to getting good sequential performance in R. The following function returns just such an iterator:

```
> ivector <- function(x, ...) {
+  i <- 1
+  it <- idiv(length(x), ...)
+
+  nextEl <- function() {
+    n <- nextElem(it)
+    ix <- seq(i, length=n)
+    i <<- i + n
+    x[ix]
+  }
+
+  obj <- list(nextElem=nextEl)
+  class(obj) <- c('ivector', 'abstractiter', 'iter')
+  obj
+ }
```

ivector uses ... to pass options on to idiv. idiv supports the chunks argument to split its argument into a specified number of pieces, and the chunkSize argument to split it into pieces of a specified maximum size.

Let's create an ivector iterator to split a vector into three pieces using the chunks argument:

```
> it <- ivector(1:25, chunks=3)
> as.list(it)


[[1]]
[1] 1 2 3 4 5 6 7 8 9

[[2]]
[1] 10 11 12 13 14 15 16 17

[[3]]
[1] 18 19 20 21 22 23 24 25
```

Note that the `nextEl` function doesn't seem to throw a `StopIteration` exception. It is actually throwing it indirectly, by calling `nextElem` on the iterator created via the `idiv` function. This function is fairly simple, because most of the tricky stuff is handled by `idiv`. `ivector` focuses on operating on the vector.

It should be clear that only minor modification need to be made to this function to create an iterator over the blocks of rows or columns of a matrix or data frame. But I'll leave that as an exercise for the reader.

6 Adding a `hasNext` method to an iterator

At times it would be nice to write a loop that explicitly gets the values of an iterator. Although that is certainly possible with a standard iterator, it requires some rather awkward error handling. One solution to this problem is to add a method that indicates whether there is another value available in the iterator. Then you can write a simple while loop that stops when there are no more values.

One way to do that would be to define a new S3 method called `hasNext`. Here's the definition of a `hasNext` generic function:

```
> hasNext <- function(obj, ...) {
+   UseMethod('hasNext')
+ }
```

We also need to define `hasNext` method for a iterator class that we'll call `ihasNext`:

```
> hasNext.ihasNext <- function(obj, ...) {
+   obj$hasNext()
+ }
```

As you can see, an `ihasNext` object must be a list with a `hasNext` element that is a function. That's the same technique that the `abstractiter` class uses to implement the `nextElem` method.

Now we'll define a function, called `ihasNext`, that takes an arbitrary iterator and returns returns an `ihasNext` iterator that wraps the specified iterator. That allows us to turn any iterator into an `ihasNext` iterator, thus providing it with a `hasNext` method:[4]

```
> ihasNext <- function(it) {
+   if (!is.null(it$hasNext)) return(it)
+   cache <- NULL
+   has_next <- NA
+
+   nextEl <- function() {
+     if (!hasNx())
+       stop('StopIteration', call.=FALSE)
+     has_next <<- NA
+     cache
+   }
+
+   hasNx <- function() {
+     if (!is.na(has_next)) return(has_next)
+     tryCatch({
+       cache <<- nextElem(it)
+       has_next <<- TRUE
+     },
+     error=function(e) {
+       if (identical(conditionMessage(e), 'StopIteration')) {
+         has_next <<- FALSE
+       } else {
+         stop(e)
+       }
+     })
+     has_next
+   }
+
+   obj <- list(nextElem=nextEl, hasNext=hasNx)
+   class(obj) <- c('ihasNext', 'abstractiter', 'iter')
+   obj
+ }
```

When the `hasNext` method is called, it calls the `nextElem` method on the underlying iterator,

[4]Thanks to Hadley Wickham for contributing this function, which I only hacked up a little. You can also find this function, along with `hasNext` and `hasNext.ihasNext` in the examples directory of the iterators packages.

and the resulting value is saved. That value is then passed to the user when `nextElem` is called. Of course, it also does the right thing if you don't call `hasNext`, or if you call it multiple times before calling `nextElem`.

So now we can easily create an `icount` iterator, and get its values in a while loop, without having to do any messy error handling:

```
> it <- ihasNext(icount(3))
> while (hasNext(it)) {
+   print(nextElem(it))
+ }
```

```
[1] 1
[1] 2
[1] 3
```

7 A recycling iterator

The `ihasNext` function from the previous section is an interesting example of a function that takes an iterator and returns an iterator that wraps the specified iterator. In that case, we wanted to add another method to the iterator. In this example, we'll return an iterator that recycles the values of the wrapped iterator:[5]

```
> irecycle <- function(it) {
+   values <- as.list(iter(it))
+   i <- length(values)
+
+   nextEl <- function() {
+     i <<- i + 1
+     if (i > length(values)) i <<- 1
+     values[[i]]
+   }
+
+   obj <- list(nextElem=nextEl)
+   class(obj) <- c('irecycle', 'abstractiter', 'iter')
+   obj
+ }
```

[5] Actually, some of the standard `iter` methods support a `recycle` argument. But this is a nice example, and a more general solution, since it works on any iterator.

This is fairly nice, but note that this is another one of those infinite iterators that we need to be careful about. Also, make sure that you don't pass an infinite iterator to `irecycle`. That would be pointless of course, since there's no reason to recycle an iterator that never ends. It would be possible to write this to avoid that problem by not grabbing all of the values right up front, but you would still end up saving values that will never be recycled, so I've opted to keep this simple.

Let's try it out:

```
> it <- irecycle(icount(3))
> unlist(as.list(it, n=9))
```

```
[1] 1 2 3 1 2 3 1 2 3
```

8 Limiting infinite iterators

I was tempted to add an argument to the `irecycle` function to limit the number of values that it returns, because sometimes you want to recycle for awhile, but not forever. I didn't do that, because rather than make `irecycle` more complicated, I decided to write yet another function that takes an iterator and returns a modified iterator to handle that task:

```
> ilimit <- function(it, times) {
+   it <- iter(it)
+
+   nextEl <- function() {
+     if (times > 0)
+       times <<- times - 1
+     else
+       stop('StopIteration')
+
+     nextElem(it)
+   }
+
+   obj <- list(nextElem=nextEl)
+   class(obj) <- c('ilimit', 'abstractiter', 'iter')
+   obj
+ }
```

Note that this looks an awful lot like the `irep` function that we implemented previously. In fact, using `ilimit`, we can implement `irep` using `iforever` much more simply, and without duplication of code:

```
> irep2 <- function(x, times)
+   ilimit(iforever(x), times)
```

To demonstrate `irep2`, I'll use `ihasNext` and a while loop:

```
> it <- ihasNext(irep2('foo', 3))
> while (hasNext(it)) {
+   print(nextElem(it))
+ }

[1] "foo"
[1] "foo"
[1] "foo"
```

Here's one last example. Let's recycle a vector three times using `ilimit`, and convert it back into a vector using `as.list` and `unlist`:

```
> iterable <- 1:3
> n <- 3
> it <- ilimit(irecycle(iterable), n * length(iterable))
> unlist(as.list(it))

[1] 1 2 3 1 2 3 1 2 3
```

Sort of a complicated version of:

```
> rep(iterable, n)

[1] 1 2 3 1 2 3 1 2 3
```

Aren't iterators fun?

9 Conclusion

Writing your own iterators can be quite simple, and yet is very useful and powerful. It provides a very effective way to extend the capabilities of other packages that use iterators, such as the `foreach` package. By writing iterators that wrap other iterators, it is possible to put together a powerful and flexible set of tools that work well together, and that can solve many of the complex problems that come up in parallel computing.