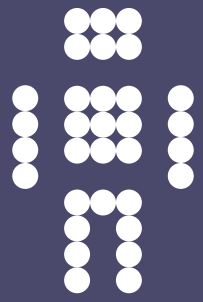
CFEngine



Iteration, Explored and Explained

A CFEngine Special Topics Handbook

CFEngine AS

Iteration is about repeating operations in a list.

In CFEngine it is the process by which a list variuable is expanded into its component parts – a powerful and much-used idiom in CFEngine. It enables a level of abstraction that can save the system maintainer the job of repetitive specification of similar promises with slight differences, and can save time and ease readability of configuration files.

What is iteration?

Iteration is about repeating operations in a list. In CFEngine, iteration is used to make a number of related promises, that fall into a pattern based on elements of a list. This is what would correspond to something like this pseudo-code in an imperative language:

foreach item in list
make promise
end

In CFEngine, we do not write loops; rather, they are implicit. Suppose '`@(list)`' is a list variable (the '`@`' means list). If we refer to this identifier using a scalar reference '`$(list)`', then CFEngine understands this to mean, take each scalar item in the list and repeat the current promise, replacing the instance with elements of the list in turn.

Iterated promises

Consider the following set of promises to report on the values of four separate monitor values:

```
bundle agent no_iteration
{
reports:
  cfengine_3::
    "mon.value_rootprocs is $(mon.value_rootprocs)";
    "mon.value_otherprocs is $(mon.value_otherprocs)";
    "mon.value_diskfree is $(mon.value_diskfree)";
    "mon.value_loadavg is $(mon.value_loadavg)";
}
```

What we did was create four distinct reports, where each report announces which monitor variable it will be reporting, and the follows with the actual value of that monitor variable. For simple reports, this is perfectly adequate and straightforward, but it lacks abstraction and repeatability. Suppose we wanted to add a variable to report, we'd need a new report promise. If we wanted to change the wording of the reports, we'd possibly have to edit four promises, and this can be time consuming and error-prone.

Consider instead the following example, which generates exactly the same reports:

```
bundle agent iteration1
{
vars:
    "monvars" slist => {
                       "rootprocs",
                       "otherprocs",
                       "diskfree",
                       "loadavg"
                       };

reports:

  cfengine_3::
```

```
    "mon.value_$(monvars) is $(mon.value_$(monvars))";
}
```

What we have done is to first specify a list variable `monvars`, and then iterate over the values contained in that list by referencing the list variable *as a scalar*. In CFEngine, simply referring to a list variable as a scalar automatically iterates over that variable.

Note that in terms of raw "lines of code", neither example shows an advantage (and in fact, the reports that are created by the iteration in this second example are *identical* to the reports in the first example).

However, we already have a gain in maintainer efficiency. By changing the single report format, we automatically change all the reports. And we have separated the semantics of the reports from the list of monitoring variables.

Admittedly, this is a simple example, but if you understand this one, we can continue with more compelling examples.

Iterating across multiple lists

Although iteration is a powerful concept in and of itself, CFEngine can iterate across multiple lists simultaneously. In the previous example, we looked at the current values of four monitor variables, but since CFEngine also gives us access to the averaged values and the standard deviation, how would we create a series of reports that listed all three statistical components of each variable? The answer is simply to do another iteration:

```
bundle agent iteration2
{
vars:
    "stats"   slist => { "value", "av", "dev" };

    "monvars" slist => {
                       "rootprocs",
                       "otherprocs",
                       "diskfree",
                       "loadavg"
                       };
reports:

  cfengine_3::
    "mon.$(stats)_$(monvars) is $(mon.$(stats)_$(monvars))";
}
```

Through the addition of a new list called `stats`, we can now iterate over both it and the `monvars` list in the same promise. The reports that we thus generate will report on `value_rootprocs`, `av_rootprocs`, and `dev_rootprocs`, followed next by `value_otherprocs`, `av_otherprocs`, etc, ending finally with `dev_loadavg`. The leftward lists are iterated over completely before going to the next value in the rightward lists.

Iterating over nested lists

Recall that CFEngine iterates over complete promise units, not small parts of a promise. Let's look at an example that could show a common misunderstanding.

If you look at the monitor variables that are described in the CFEngine Reference Guide, you'll notice that some variables reference the number of packets *in* and *out* of a host. So you might be tempted to do the following, which might not do what you expect.

```
bundle agent iteration3a
{
vars:
    "stats" slist => { "value", "av", "dev" };
    "inout" slist => { "in", "out" };

    "monvars" slist => {
                "rootprocs",    "otherprocs",
                "diskfree",
                "loadavg",
                "smtp_$(inout)",  #
                "www_$(inout)",   # look here
                "wwws_$(inout)"   #
                };

reports:
  cfengine_3::
    "mon.$(stats)_$(monvars) is $(mon.$(stats)_$(monvars))";
}
```

What this says is, for each value in '`$(inout)`', define '`monvars`' to be a variable. There are thus two attempts to defined the single name '`monvars`' as a list with two different right-hand-sides (one for 'in' and one for 'out'). This will result in the error:

```
!! Redefinition of variable "monvars" (embedded list in RHS) in context "iteration3a"
!! Redefinition of variable "monvars" (embedded list in RHS) in context "iteration3a"
```

Whenever a promise contains an iteration (that is, when the promise string or any of its attributes contain a scalar reference to a list variable), that promise is automatically re-stated with successive values from the list. So the example above is exactly the same as if we had said the following:

```
bundle agent iteration3b
{
vars:
    "stats" slist => { "value", "av", "dev" };

    "monvars" slist => {
                "rootprocs",    "otherprocs",
                "diskfree",
                "loadavg",
                "smtp_in",
                "www_in",       "wwws_in"
```

```
                };

    "monvars" slist => {
                "rootprocs",    "otherprocs",
                "diskfree",
                "loadavg",
                "smtp_out",
                "www_out",      "wwws_out"
                };

reports:
  cfengine_3::
    "mon.$(stats)_$(monvars) is $(mon.$(stats)_$(monvars))";
}
```

Notice that the promise is repeated twice, but the only thing that is different is the *right hand side* of the promise – the contents of the list, expanded using iteration over the `inout` list variable. Not only will this not do what we want, it will generate an error, because the second promise on the variable `monvars` will overwrite the value promised in the first promise! All that we will see in the reports are the *second* definition of the `monvars` list.

Fixing Iterating across nested lists

```
bundle agent iteration3c
{
vars:
    "stats" slist => { "value", "av", "dev" };
    "inout" slist => { "in", "out" };

    "monvars_$(inout)" slist => {
                                "smtp_$(inout)",  #
                                "www_$(inout)",   # look here
                                "wwws_$(inout)"   #
                                };

reports:
  cfengine_3::
    "mon.$(stats)_$(monvars_in)  is $(mon.$(stats)_$(monvars_in))";
    "mon.$(stats)_$(monvars_out) is $(mon.$(stats)_$(monvars_out))";
}
```

CFEngine does not allow an unlimited level of nesting, for reasons of efficiency and readability, and adding further levels of nesting will start to work against you. Note that we had to explicitly refer to the two variables that we created: `$(monvars_in)` and `$(monvars_out)`, and specifying more will get very messy very quickly. However, the next sections show an easier-to-read workaround.

Iterating across multiple lists, revisted

When a list variable is referenced as a scalar variable (that is, when the list variable is referenced as `$(list)`) instead of as a list (using `@(list)`), CFEngine assumes that it should substitute each scalar from the list in turn, and thus iterate over the list elements using a loop.

If more than one list variable is referenced in this manner in a single promise, each list variable is iterated over, so that every possible combination of scalar components is represented. Consider the following example.

In this example, note that the `letters` list is referenced in both the left-hand and right-hand side of the promise, the `digits` list is referenced only in the left-hand side, and the `symbols` list is only referenced in the left-hand side:

```
bundle agent iteration4a
{
vars:
    "letters" slist => { "a", "b" };
    "digits"  slist => { "1", "2" };
    "symbols" slist => { "@", "#" };

commands:
    "/bin/echo ${letters}, ${digits}+${digits}, "
        args => "${letters} and ${symbols}'";
}
```

Like a backwards-reading odometer, the left-most variable cycles the fastest and the right-most list cycles the slowest. Most importantly, no matter how many times or places a list variable is referenced as a scalar in a single promise, each combination of values is visited *only once*, regardless of whether the iteration variable is in the lefthand side or the righthand side of a promise or both.

The example above is exactly equivalent to this (much more) verbose set of promises. As you can see, there are 2*2*2 = 8 promises generated, which contains every possible comination of elements from the lists `letters`, `digits`, and `symbols`:

```
bundle agent iteration4b
{
commands:
    "/bin/echo a, 1+1, "
        args => "a and @";
    "/bin/echo b, 1+1, "
        args => "b and @";
    "/bin/echo a, 2+2, "
        args => "a and @";
    "/bin/echo b, 2+2, "
        args => "b and @";
    "/bin/echo a, 1+1, "
        args => "a and #";
    "/bin/echo b, 1+1, "
        args => "b and #";
    "/bin/echo a, 2+2, "
```

```
        args => "a and #";
    "/bin/echo b, 2+2, "
        args => "b and #";
}
```

Nesting promises workaround

Recall the problem of nesting iterations, we can now see how to repair our error. The key is to ensure that there is a distinct and unique promise created for every combination of iterated variables that we want to use. Here is how to solve the problem of listing the input and output packet counts:

```
bundle agent iteration5a
{
vars:
    "stats" slist => { "value", "av", "dev" };
    "inout" slist => { "in", "out" };
    "io_names" slist => { "smtp", "www", "wwws" };
    "io_vars[$(io_names)_$(inout)]" int => "0";
    "monvars" slist => {
                "rootprocs",    "otherprocs",
                "diskfree",
                "loadavg",
                getindices("io_vars")
                };

reports:
  cfengine_3::
    "mon.$(stats)_$(monvars) is $(mon.$(stats)_$(monvars))";
}
```

The output of this is

```
R: mon.value_rootprocs is $(mon.value_rootprocs)
R: mon.av_rootprocs is $(mon.av_rootprocs)
R: mon.dev_rootprocs is $(mon.dev_rootprocs)
R: mon.value_otherprocs is $(mon.value_otherprocs)
R: mon.av_otherprocs is $(mon.av_otherprocs)
R: mon.dev_otherprocs is $(mon.dev_otherprocs)
R: mon.value_diskfree is $(mon.value_diskfree)
R: mon.av_diskfree is $(mon.av_diskfree)
R: mon.dev_diskfree is $(mon.dev_diskfree)
R: mon.value_loadavg is $(mon.value_loadavg)
R: mon.av_loadavg is $(mon.av_loadavg)
R: mon.dev_loadavg is $(mon.dev_loadavg)
R: mon.value_wwws_in is $(mon.value_wwws_in)
R: mon.av_wwws_in is $(mon.av_wwws_in)
R: mon.dev_wwws_in is $(mon.dev_wwws_in)
R: mon.value_www_out is $(mon.value_www_out)
R: mon.av_www_out is $(mon.av_www_out)
R: mon.dev_www_out is $(mon.dev_www_out)
R: mon.value_www_in is $(mon.value_www_in)
R: mon.av_www_in is $(mon.av_www_in)
R: mon.dev_www_in is $(mon.dev_www_in)
```

```
R: mon.value_smtp_in is $(mon.value_smtp_in)
R: mon.av_smtp_in is $(mon.av_smtp_in)
R: mon.dev_smtp_in is $(mon.dev_smtp_in)
R: mon.value_wwws_out is $(mon.value_wwws_out)
R: mon.av_wwws_out is $(mon.av_wwws_out)
R: mon.dev_wwws_out is $(mon.dev_wwws_out)
R: mon.value_smtp_out is $(mon.value_smtp_out)
R: mon.av_smtp_out is $(mon.av_smtp_out)
R: mon.dev_smtp_out is $(mon.dev_smtp_out)
```

In this case, all we are doing is creating an array called `io_vars`. Note that the indices of the elements of the array are iterated from *two* lists, so in this case we'll have 2*3 = 6 elements in the array, covering all the combinations of the two lists `inout` and `inout-names`.

The values of the array elements can be whatever we like. In this case, we're making all the values `0`, because we don't care what the actual values are – we only care about the *keys* of the array. We add the list of the keys to the `monvars` list by using the return value from `getindices("io_vars")`.

Looking at the example above, you might just as easily be tempted to do the following:

```
bundle agent iteration5b
{
vars:
    "stats" slist => { "value", "av", "dev" };
    "inout" slist => { "in", "out" };
    "io_names" slist => { "smtp", "www", "wwws" };
    "io_vars[$(io_names)_$(inout)]" string => "$(io_names)_$(inout)";
    "monvars" slist => {
                "rootprocs",    "otherprocs",
                "diskfree",
                "loadavg",
                @(io_vars)
                };

reports:
  cfengine_3::
    "mon.$(stats)_$(monvars) is $(mon.$(stats)_$(monvars))";
}
```

However, this is wrong. We cannot use `@(io_vars)`, because `io_vars` is not a *list*, it is an *array*! You can only use the `@` dereferencing sigil on lists.

The power of iteration in CFEngine

Iteration and abstraction are power tools in CFEngine. In closing, consider the following simple and straightforward example, where we report on all of the monitoring variables available to us in CFEngine:

```
bundle agent iteration6
{
vars:
    "stats" slist => {"value", "av", "dev"};
```

```
    "inout" slist => {"in", "out"};
    "io_names" slist => {
                "netbiosns", "netbiosdgm", "netbiosssn",
                "irc",
                "cfengine",
                "nfsd",
                "smtp",
                "www",          "wwws",
                "ftp",
                "ssh",
                "dns",
                "icmp",         "udp",
                "tcpsyn",       "tcpack",       "tcpfin",       "tcpmisc"
                };
    "io_vars[$(io_names)_$(inout)]" string => "$(io_names)_$(inout)";

    "n" slist => {"0", "1", "2", "3"};
    "n_names" slist => {
                "temp",
                "cpu"
                };
    "n_vars[$(n_names)$(n)]" string => "$(n_names)$(n)";

    "monvars" slist => {
                "rootprocs",    "otherprocs",
                "diskfree",
                "loadavg",
                "webaccess",    "weberrors",
                "syslog",
                "messages",
                getindices("io_vars"),
                getindices("n_vars")
                };

reports:
  cfengine_3::
    "mon.$(stats)_$(monvars) is $(mon.$(stats)_$(monvars))";
}
```

In this example, we create a two arrays (`io_vars` and `n_vars`), and a number of lists (but the most important ones are `stats` and `monvars`). We have but a single report promise, but it iterates over these latter two lists. With only a single reports promise and intelligent use of lists and arrays, we are able to report on every one of the `3*(8+2*18+4*2)==156` monitor variables. And to change the format of every report, we will only have a single statement to change.

Summary of iteration

Used judiciously and intelligently, iterators are a powerful way of expressing patterns. They enable you to abstract out the concepts from the nitty-gritty details, and to specify, in very few lines, complex combinations of elements. Perhaps more importantly, they ease the burden of maintainability, by making short work of repetitive problems.