The Go Programming Language ▽

The Go Blog

Arrays, slices (and strings): The mechanics of 'append'

26 September 2013

Introduction

One of the most common features of procedural programming languages is the concept of an array. Arrays seem like simple things but there are many questions that must be answered when adding them to a language, such as:

- fixed-size or variable-size?
- is the size part of the type?
- what do multidimensional arrays look like?
- does the empty array have meaning?

The answers to these questions affect whether arrays are just a feature of the language or a core part of its design.

Next article

Strings, bytes, runes and characters in Go

Previous article

The first Go program

Links

golang.org
Install Go
A Tour of Go
Go Documentation
Go Mailing List
Go+ Community
Go on Twitter

Blog index

In the early development of Go, it took about a year to decide the answers to these questions before the design felt right. The key step was the introduction of slices, which built on fixed-size arrays to give a flexible, extensible data structure. To this day, however, programmers new to Go often stumble over the way slices work, perhaps because experience from other languages has colored their thinking.

In this post we'll attempt to clear up the confusion. We'll do so by building up the pieces to explain how the `append` built-in function works, and why it works the way it does.

Arrays

Arrays are an important building block in Go, but like the foundation of a building they are often hidden below more visible components. We must talk about them briefly before we move on to the more interesting, powerful, and prominent idea of slices.

Arrays are not often seen in Go programs because the size of an array is part of its type, which limits its expressive power.

The declaration

```
var buffer [256]byte
```

declares the variable `buffer`, which holds 256 bytes. The type of `buffer` includes its size, `[256]byte`. An array with 512 bytes would be of the distinct type `[512]byte`.

The data associated with an array is just that: an array of elements. Schematically, our buffer looks like this in memory,

```
buffer: byte byte byte ... 256 times ... byte byte byte
```

That is, the variable holds 256 bytes of data and nothing else. We can access its elements with the familiar indexing syntax, `buffer[0]`, `buffer[1]`, and so on through `buffer[255]`. (The index range 0 through 255 covers 256 elements.) Attempting to index `buffer` with a value outside this range will crash the program.

There is a built-in function called `len` that returns the number of elements of an array or slice and also of a few other data types. For arrays, it's obvious what `len` returns. In our example, `len(buffer)` returns the fixed value 256.

Arrays have their place—they are a good representation of a transformation matrix for instance—but their most common purpose in Go is to hold storage for a slice.

Slices: The slice header

Slices are where the action is, but to use them well one must understand exactly what they are and what they do.

A slice is a data structure describing a contiguous section of an array stored separately from the slice variable itself. A slice is not an array. A slice describes a piece of an array.

Given our `buffer` array variable from the previous section, we could create a slice that describes elements 100 through 150 (to be precise, 100 through 149, inclusive) by slicing the array:

```
var slice []byte = buffer[100:150]
```

In that snippet we used the full variable declaration to be explicit. The variable `slice` has type `[]byte`, pronounced "slice of bytes", and is initialized from the array, called `buffer`, by slicing elements 100 (inclusive) through 150 (exclusive). The more idiomatic syntax would drop the type, which is set by the initializing expression:

```
var slice = buffer[100:150]
```

Inside a function we could use the short declaration form,

```
slice := buffer[100:150]
```

What exactly is this slice variable? It's not quite the full story, but for now think of a slice as a little data structure with two elements: a length and a pointer to an element of an array. You can think of it as being built like this behind the scenes:

```
type sliceHeader struct {
    Length        int
    ZerothElement *byte
}

slice := sliceHeader{
    Length:        50,
    ZerothElement: &buffer[100],
}
```

Of course, this is just an illustration. Despite what this snippet says that `sliceHeader` struct is not visible to the programmer, and the type of the element pointer depends on the type of the elements, but this gives the general idea of the mechanics.

So far we've used a slice operation on an array, but we can also slice a slice, like this:

```
slice2 := slice[5:10]
```

Just as before, this operation creates a new slice, in this case with elements 5 through 9 (inclusive) of the original slice, which means elements 105 through 109 of the original array. The underlying `sliceHeader` struct for the `slice2` variable looks like this:

```
slice2 := sliceHeader{
    Length:        5,
    ZerothElement: &buffer[105],
}
```

Notice that this header still points to the same underlying array, stored in the `buffer` variable.

We can also reslice, which is to say slice a slice and store the result back in the original slice structure. After

```
slice = slice[5:10]
```

the `sliceHeader` structure for the `slice` variable looks just like it did for the `slice2` variable. You'll see reslicing used often, for example to truncate a slice. This statement drops the first and last elements of our slice:

```
slice = slice[1:len(slice)-1]
```

[Exercise: Write out what the `sliceHeader` struct looks like after this assignment.]

You'll often hear experienced Go programmers talk about the "slice header" because that really is what's stored in a slice variable. For instance, when you call a function that takes a slice as an argument, such as bytes.IndexRune, that header is what gets passed to the function. In this call,

```
slashPos := bytes.IndexRune(slice, '/')
```

the `slice` argument that is passed to the `IndexRune` function is, in fact, a "slice header".

There's one more data item in the slice header, which we talk about below, but first let's see what the existence of the slice header means when you program with slices.

Passing slices to functions

It's important to understand that even though a slice contains a pointer, it is itself a value. Under the covers, it is a struct value holding a pointer and a length. It is not a pointer to a struct.

This matters.

When we called `IndexRune` in the previous example, it was passed a copy of the slice header. That behavior has important ramifications.

Consider this simple function:

```
func AddOneToEachElement(slice []byte) {
    for i := range slice {
        slice[i]++
    }
}
```

It does just what its name implies, iterating over the indices of a slice (using a `for range` loop), incrementing its elements.

Try it:

```
func main() {
    slice := buffer[10:20]
    for i := 0; i < len(slice); i++ {
        slice[i] = byte(i)
    }
    fmt.Println("before", slice)
    AddOneToEachElement(slice)
    fmt.Println("after", slice)
}
```

Run

(You can edit and re-execute these runnable snippets if you want to explore.)

Even though the slice header is passed by value, the header includes a pointer to elements of an array, so both the original slice header and the copy of the header passed to the function describe the same array. Therefore, when the function returns, the modified elements can be seen through the original slice variable.

The argument to the function really is a copy, as this example shows:

```
func SubtractOneFromLength(slice []byte) []byte {
    slice = slice[0 : len(slice)-1]
    return slice
}

func main() {
    fmt.Println("Before: len(slice) =", len(slice))
    newSlice := SubtractOneFromLength(slice)
    fmt.Println("After:  len(slice) =", len(slice))
    fmt.Println("After:  len(newSlice) =", len(newSlice))
}
```

Run

Here we see that the contents of a slice argument can be modified by a function, but its header cannot. The length stored in the `slice` variable is not modified by the call to the function, since the function is passed a copy of the slice header, not the original. Thus if we want to write a function that modifies the header, we must return it as a result parameter, just as we have done here. The `slice` variable is unchanged but the returned value has the new length, which is then stored in `newSlice`,

Pointers to slices: Method receivers

Another way to have a function modify the slice header is to pass a pointer to it. Here's a variant of our previous example that does this:

```
func PtrSubtractOneFromLength(slicePtr *[]byte) {
    slice := *slicePtr
    *slicePtr = slice[0 : len(slice)-1]
}

func main() {
    fmt.Println("Before: len(slice) =", len(slice))
    PtrSubtractOneFromLength(&slice)
    fmt.Println("After:  len(slice) =", len(slice))
    fmt.Println("After:  len(slice) =", len(slice))
}
```

Run

It seems clumsy in that example, especially dealing with the extra level of indirection (a temporary variable helps), but there is one common case where you see pointers to slices. It is idiomatic to use a pointer receiver for a method that modifies a slice.

Let's say we wanted to have a method on a slice that truncates it at the final slash. We could write it like this:

```
type path []byte

func (p *path) TruncateAtFinalSlash() {
    i := bytes.LastIndex(*p, []byte("/"))
    if i >= 0 {
        *p = (*p)[0:i]
    }
}

func main() {
    pathName := path("/usr/bin/tso") // Conversion from string to path.
    pathName.TruncateAtFinalSlash()
    fmt.Printf("%s\n", pathName)
}
```

Run

If you run this example you'll see that it works properly, updating the slice in the caller.

[Exercise: Change the type of the receiver to be a value rather than a pointer and run it again. Explain what happens.]

On the other hand, if we wanted to write a method for `path` that upper-cases the ASCII letters in the path (parochially ignoring non-English names), the method could be a value because the value receiver will still point to the same underlying array.

```
type path []byte

func (p path) ToUpper() {
    for i, b := range p {
        if 'a' <= b && b <= 'z' {
            p[i] = b + 'A' - 'a'
        }
    }
}

func main() {
    pathName := path("/usr/bin/tso")
    pathName.ToUpper()
    fmt.Printf("%s\n", pathName)
}
```

Run

Here the `ToUpper` method uses two variables in the `for range` construct to capture the index and slice element. This form of loop avoids writing `p[i]` multiple times in the body.

[Exercise: Convert the `ToUpper` method to use a pointer receiver and see if its behavior changes.]

[Advanced exercise: Convert the `ToUpper` method to handle Unicode letters, not just ASCII.]

Capacity

Look at the following function that extends its argument slice of `ints` by one element:

```
func Extend(slice []int, element int) []int {
    n := len(slice)
    slice = slice[0 : n+1]
    slice[n] = element
    return slice
}
```

(Why does it need to return the modified slice?) Now run it:

```
func main() {
    var iBuffer [10]int
    slice := iBuffer[0:0]
    for i := 0; i < 20; i++ {
        slice = Extend(slice, i)
        fmt.Println(slice)
    }
}
```

Run

See how the slice grows until... it doesn't.

It's time to talk about the third component of the slice header: its capacity. Besides the array pointer and length, the slice header also stores its capacity:

```
type sliceHeader struct {
    Length        int
    Capacity      int
    ZerothElement *byte
}
```

The `Capacity` field records how much space the underlying array actually has; it is the maximum value the `Length` can reach. Trying to grow the slice beyond its capacity will step beyond the limits of the array and will trigger a panic.

After our example slice is created by

```
slice := iBuffer[0:0]
```

its header looks like this:

```
slice := sliceHeader{
    Length:        0,
    Capacity:      10,
    ZerothElement: &iBuffer[0],
}
```

The `Capacity` field is equal to the length of the underlying array, minus the index in the array of the first element of the slice (zero in this case). If you want to inquire what the capacity is for a slice, use the built-in function `cap`:

```
if cap(slice) == len(slice) {
    fmt.Println("slice is full!")
    }
```

Make

What if we want to grow the slice beyond its capacity? You can't! By definition, the capacity is the limit to growth. But you can achieve an equivalent result by allocating a new array, copying the data over, and modifying the slice to describe the new array.

Let's start with allocation. We could use the `new` built-in function to allocate a bigger array and then slice the result, but it is simpler to use the `make` built-in function instead. It allocates a new array and creates a slice header to describe it, all at once. The `make` function takes three arguments: the type of the slice, its initial length, and its capacity, which is the length of the array that `make` allocates to hold the slice data. This call creates a slice of length 10 with room for 5 more (15-10), as you can see by running it:

```
    slice := make([]int, 10, 15)
    fmt.Printf("len: %d, cap: %d\n", len(slice), cap(slice))
```

Run

This snippet doubles the capacity of our `int` slice but keeps its length the same:

```
    slice := make([]int, 10, 15)
    fmt.Printf("len: %d, cap: %d\n", len(slice), cap(slice))
    newSlice := make([]int, len(slice), 2*cap(slice))
    for i := range slice {
        newSlice[i] = slice[i]
    }
    slice = newSlice
    fmt.Printf("len: %d, cap: %d\n", len(slice), cap(slice))
```

Run

After running this code the slice has much more room to grow before needing another reallocation.

When creating slices, it's often true that the length and capacity will be same. The `make` built-in has a shorthand for this common case. The length argument defaults to the capacity, so you can leave it out to set them both to the same value. After

```
gophers := make([]Gopher, 10)
```

the `gophers` slice has both its length and capacity set to 10.

Copy

When we doubled the capacity of our slice in the previous section, we wrote a loop to copy the old data to the new slice. Go has a built-in function, `copy`, to make this easier. Its arguments are two slices, and it copies the data from the right-hand argument to the left-hand argument. Here's our example rewritten to use `copy`:

```
    newSlice := make([]int, len(slice), 2*cap(slice))
    copy(newSlice, slice)
```

Run

The `copy` function is smart. It only copies what it can, paying attention to the lengths of both arguments. In other words, the number of elements it copies is the minimum of the lengths of the two slices. This can save a little bookkeeping. Also, `copy` returns an integer value, the number of elements it copied, although it's not always worth checking.

The `copy` function also gets things right when source and destination overlap, which means it can be used to shift items around in a single slice. Here's how to use `copy` to insert a value into the middle of a slice.

```
// Insert inserts the value into the slice at the specified index,
// which must be in range.
// The slice must have room for the new element.
func Insert(slice []int, index, value int) []int {
    // Grow the slice by one element.
    slice = slice[0 : len(slice)+1]
    // Use copy to move the upper part of the slice out of the way and
open a hole.
    copy(slice[index+1:], slice[index:])
    // Store the new value.
    slice[index] = value
    // Return the result.
    return slice
}
```

There are a couple of things to notice in this function. First, of course, it must return the updated slice because its length has changed. Second, it uses a convenient shorthand. The expression

```
slice[i:]
```

means exactly the same as

```
slice[i:len(slice)]
```

Also, although we haven't used the trick yet, we can leave out the first element of a slice expression too; it defaults to zero. Thus

```
slice[:]
```

just means the slice itself, which is useful when slicing an array. This expression is the shortest way to say "a slice describing all the elements of the array":

```
array[:]
```

Now that's out of the way, let's run our `Insert` function.

```
    slice := make([]int, 10, 20) // Note capacity > length: room to add
element.
    for i := range slice {
        slice[i] = i
    }
    fmt.Println(slice)
    slice = Insert(slice, 5, 99)
    fmt.Println(slice)
```

Run

Append: An example

A few sections back, we wrote an Extend function that extends a slice by one element. It was buggy, though, because if the slice's capacity was too small, the function would crash. (Our Insert example has the same problem.) Now we have the pieces in place to fix that, so let's write a robust implementation of Extend for integer slices.

```
func Extend(slice []int, element int) []int {
    n := len(slice)
    if n == cap(slice) {
        // Slice is full; must grow.
        // We double its size and add 1, so if the size is zero we still
grow.
        newSlice := make([]int, len(slice), 2*len(slice)+1)
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0 : n+1]
    slice[n] = element
    return slice
}
```

In this case it's especially important to return the slice, since when it reallocates the resulting slice describes a completely different array. Here's a little snippet to demonstrate what happens as the slice fills up:

```
    slice := make([]int, 0, 5)
    for i := 0; i < 10; i++ {
        slice = Extend(slice, i)
        fmt.Printf("len=%d cap=%d slice=%v\n", len(slice), cap(slice),
slice)
        fmt.Println("address of 0th element:", &slice[0])
    }
```

Run

Notice the reallocation when the initial array of size 5 is filled up. Both the capacity and the address of the zeroth element change when the new array is allocated.

With the robust Extend function as a guide we can write an even nicer function that lets us extend the slice by multiple elements. To do this, we use Go's ability to turn a list of function arguments into a slice when the function is called. That is, we use Go's variadic function facility.

Let's call the function Append. For the first version, we can just call Extend repeatedly so the mechanism of the variadic function is clear. The signature of Append is this:

```
func Append(slice []int, items ...int) []int
```

What that says is that Append takes one argument, a slice, followed by zero or more int arguments. Those arguments are exactly a slice of int as far as the implementation of Append is concerned, as you can see:

```
// Append appends the items to the slice.
// First version: just loop calling Extend.
func Append(slice []int, items ...int) []int {
    for _, item := range items {
        slice = Extend(slice, item)
    }
    return slice
}
```

Notice the for range loop iterating over the elements of the items argument, which has implied type []int. Also notice the use of the blank identifier _ to discard the index in the loop, which we don't need in this case.

Try it:

```
    slice := []int{0, 1, 2, 3, 4}
    fmt.Println(slice)
    slice = Append(slice, 5, 6, 7, 8)
    fmt.Println(slice)
```

Run

Another new technique in this example is that we initialize the slice by writing a composite literal, which consists of the type of the slice followed by its elements in braces:

```
    slice := []int{0, 1, 2, 3, 4}
```

The Append function is interesting for another reason. Not only can we append elements, we can append a whole second slice by "exploding" the slice into arguments using the **...** notation at the call site:

```
    slice1 := []int{0, 1, 2, 3, 4}
    slice2 := []int{55, 66, 77}
    fmt.Println(slice1)
    slice1 = Append(slice1, slice2...) // The '...' is essential!
    fmt.Println(slice1)
```

Run

Of course, we can make Append more efficient by allocating no more than once, building on the innards of Extend:

```
// Append appends the elements to the slice.
// Efficient version.
func Append(slice []int, elements ...int) []int {
    n := len(slice)
    total := len(slice) + len(elements)
    if total > cap(slice) {
        // Reallocate. Grow to 1.5 times the new size, so we can still
grow.
        newSize := total*3/2 + 1
        newSlice := make([]int, total, newSize)
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[:total]
    copy(slice[n:], elements)
    return slice
}
```

Here, notice how we use copy twice, once to move the slice data to the newly allocated memory, and then to copy the appending items to the end of the old data.

Try it; the behavior is the same as before:

```
slice1 := []int{0, 1, 2, 3, 4}
slice2 := []int{55, 66, 77}
fmt.Println(slice1)
slice1 = Append(slice1, slice2...) // The '...' is essential!
fmt.Println(slice1)
```

Run

Append: The built-in function

And so we arrive at the motivation for the design of the `append` built-in function. It does exactly what our `Append` example does, with equivalent efficiency, but it works for any slice type.

A weakness of Go is that any generic-type operations must be provided by the run-time. Some day that may change, but for now, to make working with slices easier, Go provides a built-in generic `append` function. It works the same as our `int` slice version, but for any slice type.

Remember, since the slice header is always updated by a call to `append`, you need to save the returned slice after the call. In fact, the compiler won't let you call append without saving the result.

Here are some one-liners intermingled with print statements. Try them, edit them and explore:

```
// Create a couple of starter slices.
slice := []int{1, 2, 3}
slice2 := []int{55, 66, 77}
fmt.Println("Start slice: ", slice)
fmt.Println("Start slice2:", slice2)

// Add an item to a slice.
slice = append(slice, 4)
fmt.Println("Add one item:", slice)

// Add one slice to another.
slice = append(slice, slice2...)
fmt.Println("Add one slice:", slice)

// Make a copy of a slice (of int).
slice3 := append([]int(nil), slice...)
fmt.Println("Copy a slice:", slice3)

// Copy a slice to the end of itself.
fmt.Println("Before append to self:", slice)
slice = append(slice, slice...)
fmt.Println("After append to self:", slice)
```

Run

It's worth taking a moment to think about the final one-liner of that example in detail to understand how the design of slices makes it possible for this simple call to work correctly.

There are lots more examples of `append`, `copy`, and other ways to use slices on the community-built "Slice Tricks" Wiki page.

Nil

As an aside, with our newfound knowledge we can see what the representation of a `nil` slice is. Naturally, it is the zero value of the slice header:

```
sliceHeader{
    Length:        0,
    Capacity:      0,
    ZerothElement: nil,
}
```

or just

```
sliceHeader{}
```

The key detail is that the element pointer is `nil` too. The slice created by

```
array[0:0]
```

has length zero (and maybe even capacity zero) but its pointer is not `nil`, so it is not a nil slice.

As should be clear, an empty slice can grow (assuming it has non-zero capacity), but a `nil` slice has no array to put values in and can never grow to hold even one element.

That said, a `nil` slice is functionally equivalent to a zero-length slice, even though it points to nothing. It has length zero and can be appended to, with allocation. As an example, look at the one-liner above that copies a slice by appending to a `nil` slice.

Strings

Now a brief section about strings in Go in the context of slices.

Strings are actually very simple: they are just read-only slices of bytes with a bit of extra syntactic support from the language.

Because they are read-only, there is no need for a capacity (you can't grow them), but otherwise for most purposes you can treat them just like read-only slices of bytes.

For starters, we can index them to access individual bytes:

```
slash := "/usr/ken"[0] // yields the byte value '/'.
```

We can slice a string to grab a substring:

```
usr := "/usr/ken"[0:4] // yields the string "/usr"
```

It should be obvious now what's going on behind the scenes when we slice a string.

We can also take a normal slice of bytes and create a string from it with the simple conversion:

```
str := string(slice)
```

and go in the reverse direction as well:

```
slice := []byte(usr)
```

The array underlying a string is hidden from view; there is no way to access its contents except through the string. That means that when we do either of these conversions, a copy of the array must be made. Go takes care of this, of course, so you don't have to. After either of these conversions, modifications to the array underlying the byte slice don't affect the corresponding string.

An important consequence of this slice-like design for strings is that creating a substring is very efficient. All that needs to happen is the creation of a two-word string header. Since the string is read-only, the original string and the string resulting from the slice operation can share the same array safely.

A historical note: The earliest implementation of strings always allocated, but when slices were added to the language, they provided a model for efficient string handling. Some of the benchmarks saw huge speedups as a result.

There's much more to strings, of course, and a separate blog post covers them in greater depth.

Conclusion

To understand how slices work, it helps to understand how they are implemented. There is a little data structure, the slice header, that is the item associated with the slice variable, and that header describes a section of a separately allocated array. When we pass slice values around, the header gets copied but the array it points to is always shared.

Once you appreciate how they work, slices become not only easy to use, but powerful and expressive, especially with the help of the `copy` and `append` built-in functions.

More reading

There's lots to find around the intertubes about slices in Go. As mentioned earlier, the "Slice Tricks" Wiki page has many examples. The Go Slices blog post describes the memory layout details with clear diagrams. Russ Cox's Go Data Structures article includes a discussion of slices along with some of Go's other internal data structures.

There is much more material available, but the best way to learn about slices is to use them.

By Rob Pike

Related articles

- Go Slices: usage and internals

Terms of Service | Privacy Policy | View the source code