Go 1.2 Runtime Symbol Information

Go 1.2 Runtime Symbol Information

Russ Cox
July 2013

Abstract

This is the new runtime symbol support that will be in Go 1.2. The changes are internal only, not user visible, but they remove some flaky code, reduce memory footprint, and make garbage collection implementation easier.

Background

To get accurate crash-time stack traces years ago, I moved the Plan 9 symbol and pc->line number (pcln) tables into memory that would be mapped at run time, and then I put code to parse them in the runtime package. In addition to stack traces, the stack walking ability is now used for profiling and garbage collection. This was a significant improvement on what was there before, but it has a significant drawback: the first thing a Go program does is spend time and memory allocations decoding those tables into a form that supports the kinds of queries needed during the stack walks. There has been a comment in the runtime for a long time (on struct Func) that says "Eventually, the loaded symbol table should be closer to this form." Doing that would make it possible to read the tables directly, eliminating the start-up time and memory costs of decoding.

There are other problems with the approach too: it's a bit awkward to get new metadata from the compiler through to the runtime, because it has to be sent to the linker in one form, the linker has to encode it in the symbol table in a second form, and the runtime has to decode the new symbol table extensions. Carl has been running into this with garbage collection additions. Keith also pointed out that this was a mess a month or so ago.

PC-Value Tables

Before we look at the details of the plan, we need to introduce the concept of a pc-value table, a generalization of the current pcln table. A pc-value table maps a code location denoted by program counter to an int32 value. The specific encoding is not terribly important, except to say that it can be made fairly compact and easy to decode on the fly. The appendix gives a worked example.

There are many interesting uses of pc-value tables. The most obvious is mapping program counter to line number, but another use is to generate a pcsp table recording the stack frame size at each program counter, so that stack unwinding can always determine where the current frame ends and the next one begins. (In fact, some of the early Plan 9 linkers emitted such a table, but 5l, 6l, and 8l never have.) Another use would be for the compiler to record stack variable liveness information at each call site, so that the garbage collector can ignore dead stack slots. Another would be for the compiler to record information about which registers contain values that need to be flushed to memory, so that a panic could update local or global variables appropriately.

Proposed Plan

The plan is to replace the current symbol and pcln tables with an in-memory form explicitly designed to be used by the Go runtime without any preprocessing. New pseudo-instructions will make it possible to define custom per-function metadata and custom per-function pc-value tables, without linker changes. Only the compilers and runtime would need to change to add new data, and both of those are unavoidable (the compiler must generate the data and the runtime must consume it).

Specifically, the new function symbol table is a program counter lookup table of the form

    N pc0 func0 pc1 func1 pc2 func2 ... pc(N-1) func(N-1) pcN

This table is a count N followed by a list of alternating pc, function metadata pointer values. To find the function for a given program counter, the runtime does binary search on the pc values. The final pcN value is the address just beyond func(N-1), so that the binary search can distinguish between a pc inside func(N-1) and a pc outside the text segment.

Each of the funcN values is a uintptr offset from the beginning of the function symbol table to the following structure:

```
struct        Func
{
        uintptr        entry;  // start pc
        int32 name;         // name (offset to C string)
        int32 args;         // size of arguments passed to function
        int32 frame;        // size of function frame, including saved caller PC
        int32       pcsp;                // pcsp table (offset to pcvalue table)
        int32       pcfile;          // pcfile table (offset to pcvalue table)
        int32       pcln;                  // pcln table (offset to pcvalue table)
        int32       nfuncdata;          // number of entries in funcdata list
        int32       npcdata;          // number of entries in pcdata list
};
```

The fixed per-function metadata holds the function's entry PC and pointers (in the form of offsets from the beginning of the pclntab) to the name and pc-value tables giving current stack pointer offset, file number, and line number. To find the file:line of a given program counter, you use the pcfile and pcln tables to compute file number and line number, use the file number to index into a table of all source files used in the program (more details below), and use the line number directly. Without getting bogged down in the details, let's just say that this is vastly simpler than the current file:line derivation.

In addition to this fixed per-function metadata, there is a list of opaque per-function metadata pointers and a list of opaque per-function pc-value tables. The pseudo-instruction

```
FUNCDATA $2, $sym(SB)
```

declares that index 2 of the funcdata list should be a pointer to "sym". The index's only purpose is to allow definition of multiple kinds of funcdata without colliding. There will need to be a central registry of indexes, probably maintained in src/pkg/runtime/funcdata.h.

Similarly, the pseudo-instruction

```
PCDATA $3, $45
```

declares that the value with index 3 associated with the current program counter is 45. Each of the pcdata indexes (PCDATA $1, PCDATA $2, and so on) encodes to a separate pc-value table. Just as with funcdata, the index allows the definition of multiple kinds of pcdata for a given function, and there will be a registry for those indexes too.

At run time, the runtime can, from a Func, retrieve the funcdata with a given index or the pcdata with a given index at a given program counter. A pcdata stream may produce an int32 interpreted by reference to corresponding data retrieved as a funcdata pointer.

The struct Func is followed immediately in memory by npcdata int32s giving the offsets to the pcdata tables; if nfuncdata > 0, the npcdata int32s are followed immediately by a possible int32 gap for alignment and then nfuncdata uintptrs giving the funcdata values. If pcsp, pcfile, pcln, or any of the pcdata offsets is zero, that table is considered missing, and all PCs take value -1.

The pcfile pc-value table produces indexes into a table of source file names. That table has the format:

N+1 name1 name2 name3 … nameN

where N+1 is an int32 and each nameX is an int32 offset to a C string. The offset is relative to the beginning of the table. There is no “name 0”.

The memory for the function symbol table, file name table, Func structures, and the data referred to by offset from those are all contiguous in memory and recorded as a single symbol, “pclntab”. The end address is recorded as “epclntab”. To make it easy to find, in most file formats the symbol has an object file section to itself. The first word of pclntab is a magic number uint32 0xfffffffb followed by two zero bytes followed by one byte giving the instruction size quantum (1 for x86, 4 for ARM), one byte giving the size of a uintptr in bytes. So for an amd64 system, the pclntab symbol begins:

[4] 0xfffffffb

```
[2] 0x00 0x00
[1] 0x01
[1] 0x08
    [8] N (size of function symbol table)
    [8] pc0
    [8] func0 offset
    [8] pc1
    [8] func1 offset
    …
    [8] pcN
    [4] int32 offset from start to source file table
    … and then data referred to by offset, in an unspecified order …
```

The [n] notation indicates an n-byte host-endian value. On a 32-bit system, the 8-byte fields would be 4-byte fields.

Placing all the memory contiguously and assigning it to a single symbol makes it possible for debuggers to load and interpret the table by copying just that one symbol out of the read-only data segment.

In a program composed of distinct modules or shared libraries, each module must define its own pclntab made available to the runtime. In such a program, looking up a function by program counter would first identify the corresponding module and then proceed as above.

To simplify the loading of programs composed of separate modules, the PC values, the Func.entry field, and the funcdata table values are all defined as pointers relative to the base address of the module containing the pclntab. A main executable has base address zero, so the pointers are usable directly.

Storage Overhead

The new table, being a C struct, is not as compact as the current Plan 9 symtab encoding. The list of full source file paths is also much less compact (the Plan 9 symtab encodes paths as element index sequences, so that shared prefixes are only stored once, but doing so here would require allocation to prepare the full string).

Even so, it appears that the new table is smaller than the old tables, because the Plan 9 symbol table contains significant amounts of information that a debugger might use but that the runtime has no use for, such as the stack location and type of every local variable in a function.

In godoc built at tip, the Plan 9 symtab+pcln tables occupy 2.1 MB in the file and mapped into read-only memory during execution, and then the decoded form is likely another 1 MB or so. Under this proposal, the new tables, including all the associated string data, occupy just under 1 MB, and they do not require a separate decoded form. So the executing image at run time is *smaller* by 2 MB, 1 MB less read-only data and 1 MB less decoded data. But the executable file is 1 MB larger, because the symtab and pcln are still stored (now elsewhere) in the file as debugging information for use by go tool nm and go tool addr2line (used by pprof and sometimes directly by Go developers).

Appendix: PC-Value Table Encoding

A pc-value table is a sequence of (value, pc) pairs declaring the given value to hold up to but not including the given program counter. The table encodes deltas from the previous entry, assuming an initial value of -1 and pc set to the function entry. PC deltas are always positive, but they are scaled by the instruction alignment for the architecture: on x86 2 means 2 bytes, but on ARM 2 means 2 words, so 8 bytes. Value deltas can be negative. The value delta is zig-zag encoded, so the numbers 0, 1, 2, 3, 4, 5, 6, 7 are the encoding for 0, -1, 1, -2, 2, -3, 3, and so on. Both the scaled PC deltas and the zig-zag encoded value deltas are written as variable-length base-128 integers in little-endian order, using the 0x80 bit to indicate that the number continues in the next byte. This matches the Plan 9 pcln table encoding in sprit but not in detail.

Here is the program counter, stack pointer offset, and disassembly for a simple 'func main':

```
2000        (8)         TEXT        main.main+0(SB),$32
2000      0 (8)         MOVQ        2208(GS),CX
2009        (8)         CMPQ        SP,(CX)
200c        (8)         JHI        ,2015
200e        (8)         CALL        ,19ee0+runtime.morestack00
2013        (8)         JMP        ,2000
```

```
  2015        (8)         SUBQ        $32,SP
  2019     32 (9)         LEAQ        go.string."hello world"+0(SB),BX
  2021        (9)         LEAQ        (SP),BP
  2025        (9)         MOVQ        BP,DI
  2028        (9)         MOVQ        BX,SI
  202b        (9)         MOVSQ        ,
  202d        (9)         MOVSQ        ,
  202f        (9)         CALL        ,f660+runtime.printstring
  2034        (9)         CALL        ,f710+runtime.printnl
  2039        (10)        LEAQ        go.string."%S\n"+0(SB),BX
  2041        (10)        LEAQ        (SP),BP
  2045        (10)        MOVQ        BP,DI
  2048        (10)        MOVQ        BX,SI
  204b        (10)        MOVSQ        ,
  204d        (10)        MOVSQ        ,
  204f        (10)        LEAQ        go.string."hi"+0(SB),BX
  2057        (10)        LEAQ        16(SP),BP
  205c        (10)        MOVQ        BP,DI
  205f        (10)        MOVQ        BX,SI
  2062        (10)        MOVSQ        ,
  2064        (10)        MOVSQ        ,
  2066        (10)        PUSHQ        $runtime.goprintf·f+0(SB),
  206b     40 (10)        PUSHQ        $32,
  206d     48 (10)        CALL        ,db80+runtime.deferproc
  2072        (10)        POPQ        ,CX
  2073     40 (10)        POPQ        ,CX
  2074     32 (10)        TESTQ        AX,AX
  2077        (10)        JNE        $0,2092
  2079        (11)        CALL        ,15820+runtime.SYSR1
  207e        (12)        MOVQ        $0,AX
  2081        (13)        MOVQ        (AX),BP
  2084        (13)        MOVQ        BP,(SP)
  2088        (13)        CALL        ,f520+runtime.printint
  208d        (13)        CALL        ,f710+runtime.printnl
  2092        (14)        CALL        ,dbf0+runtime.deferreturn
  2097        (14)        ADDQ        $32,SP
  209b      0 (14)        RET        ,
  209c done
```

The corresponding table of (value, pc) pairs is:

```
 0     0x2019
32     0x206b
40     0x206d
48     0x2073
40     0x2074
32     0x209b
 0     0x209c
```

The deltas to be encoded are:

```
  1     0x19
 32     0x52
  8     0x02
  8     0x06
 -8     0x01
 -8     0x27
-32     0x01
```

Applying the zig-zag encoding to the value deltas:

```
  2     0x19
 64     0x52
 16     0x02
 16     0x06
 15     0x01
 15     0x27
 63     0x01
```

and since none of the values are bigger than 128, those bytes give the encoded table (in hexadecimal):

```
02 19 40 52 10 02 10 06 0f 01 0f 27 3f 01
```

As another example, this table gives file number instead of stack pointer offset. The file number is constant for the whole program:

```
2000        (8)         TEXT        main.main+0(SB),$32
2000      2 (8)         MOVQ        2208(GS),CX
2009        (8)         CMPQ        SP,(CX)
200c        (8)         JHI        ,2015
200e        (8)         CALL        ,19ee0+runtime.morestack00
2013        (8)         JMP        ,2000
2015        (8)         SUBQ        $32,SP
2019        (9)         LEAQ        go.string."hello world"+0(SB),BX
2021        (9)         LEAQ        (SP),BP
2025        (9)         MOVQ        BP,DI
2028        (9)         MOVQ        BX,SI
202b        (9)         MOVSQ        ,
202d        (9)         MOVSQ        ,
202f        (9)         CALL        ,f660+runtime.printstring
2034        (9)         CALL        ,f710+runtime.printnl
2039        (10)        LEAQ        go.string."%S\n"+0(SB),BX
2041        (10)        LEAQ        (SP),BP
2045        (10)        MOVQ        BP,DI
2048        (10)        MOVQ        BX,SI
204b        (10)        MOVSQ        ,
204d        (10)        MOVSQ        ,
204f        (10)        LEAQ        go.string."hi"+0(SB),BX
2057        (10)        LEAQ        16(SP),BP
205c        (10)        MOVQ        BP,DI
205f        (10)        MOVQ        BX,SI
2062        (10)        MOVSQ        ,
2064        (10)        MOVSQ        ,
2066        (10)        PUSHQ        $runtime.goprintf·f+0(SB),
206b        (10)        PUSHQ        $32,
206d        (10)        CALL        ,db80+runtime.deferproc
2072        (10)        POPQ        ,CX
2073        (10)        POPQ        ,CX
2074        (10)        TESTQ        AX,AX
2077        (10)        JNE        $0,2092
2079        (11)        CALL        ,15820+runtime.SYSR1
207e        (12)        MOVQ        $0,AX
2081        (13)        MOVQ        (AX),BP
2084        (13)        MOVQ        BP,(SP)
2088        (13)        CALL        ,f520+runtime.printint
208d        (13)        CALL        ,f710+runtime.printnl
2092        (14)        CALL        ,dbf0+runtime.deferreturn
2097        (14)        ADDQ        $32,SP
```

```
  209b        (14)        RET        ,
  209c  done
```

The table of (value, pc) pairs is:

```
 2      0x209c
```

The deltas to be encoded are:

```
 3      0x9c
```

Applying zig-zag to the value deltas:

```
 6      0x9c
```

And encoding the deltas as varints:

```
 06 9c 01
```

The varint encoding of 0x9c, which splits in base 128 into 01 1C, is the byte 0x80 | 0x1C followed by 0x01. The 0x80 indicates that another byte follows.

Appendix: Zig-Zag Encoding

To convert signed integers into zig-zag encoding, shift left one, and then if the high bit shifted out was a 1, bitwise invert the value. To decode, reverse.

```
package main

import "fmt"

func encode(v int32) uint32 {
        return uint32(v<<1) ^ uint32(v>>31)
}

func decode(u uint32) int32 {
        return int32(u>>1) ^ int32(u<<31)>>31
}

func main() {
        for i := int32(-10); i <= 10; i++ {
                fmt.Printf("%4d %4d %4d\n",
                  i, encode(i),
                  decode(encode(i)))
        }
}
```

[http://play.golang.org/p/JX_vtu_9YM]