48 | 程序性能分析基础（上）

2018-11-30 郝林

作为拾遗的部分，今天我们来讲讲与Go程序性能分析有关的基础知识。

Go语言为程序开发者们提供了丰富的性能分析API，和非常好用的标准工具。这些API主要存在于：

1. `runtime/pprof`；
2. `net/http/pprof`；
3. `runtime/trace`；

这三个代码包中。

另外，`runtime`代码包中还包含了一些更底层的API。它们可以被用来收集或输出Go程序运行过程中的一些关键指标，并帮助我们生成相应的概要文件以供后续分析时使用。

至于标准工具，主要有`go tool pprof`和`go tool trace`这两个。它们可以解析概要文件中的信息，并以人类易读的方式把这些信息展示出来。

此外，`go test`命令也可以在程序测试完成后生成概要文件。如此一来，我们就可以很方便地使用前面那两个工具读取概要文件，并对被测程序的性能加以分析。这无疑会让程序性能测试的一手资料更加丰富，结果更加精确和可信。

在Go语言中，用于分析程序性能的概要文件有三种，分别是：CPU概要文件（CPU Profile）、内存概要文件（Mem Profile）和阻塞概要文件（Block Profile）。

这些概要文件中包含的都是：在某一段时间内，对Go程序的相关指标进行多次采样后得到的概要信息。

对于CPU概要文件来说，其中的每一段独立的概要信息都记录着，在进行某一次采样的那个时刻，CPU上正在执行的Go代码。

而对于内存概要文件，其中的每一段概要信息都记载着，在某个采样时刻，正在执行的Go代码以及堆内存的使用情况，这里包含已分配和已释放的字节数量和对象数量。至于阻塞概要文件，其中的每一段概要信息，都代表着Go程序中的一个goroutine阻塞事件。

注意，在默认情况下，这些概要文件中的信息并不是普通的文本，它们都是以二进制的形式展现的。如果你使用一个常规的文本编辑器查看它们的话，那么肯定会看到一堆“乱码”。

这时就可以显现出`go tool pprof`这个工具的作用了。我们可以通过它进入一个基于命令行的交互式界面，并对指定的概要文件进行查阅。就像下面这样：

```
$ go tool pprof cpuprofile.out
Type: cpu
Time: Nov 9, 2018 at 4:31pm (CST)
Duration: 7.96s, Total samples = 6.88s (86.38%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

关于这个工具的具体用法，我就不在这里赘述了。在进入这个工具的交互式界面之后，我们只要输入指令`help`并按下回车键，就可以看到很详细的帮助文档。

我们现在来说说怎样生成概要文件。

你可能会问，既然在概要文件中的信息不是普通的文本，那么它们到底是什么格式的呢？一个对广大的程序开发者而言，并不那么重要的事实是，它们是通过protocol buffers生成的二进制数据流，或者说字节流。

概括来讲，protocol buffers是一种数据序列化协议，同时也是一个序列化工具。它可以把一个值，比如一个结构体或者一个字典，转换成一段字节流。

也可以反过来，把经过它生成的字节流反向转换为程序中的一个值。前者就被叫做序列化，而后者则被称为反序列化。

换句话说，protocol buffers定义和实现了一种“可以让数据在结构形态和扁平形态之间互相转换”的方式。

Protocol buffers的优势有不少。比如，它可以在序列化数据的同时对数据进行压缩，所以它生成的字节流，通常都要比相同数据的其他格式（例如XML和JSON）占用的空间明显小很多。

又比如，它既能让我们自己去定义数据序列化和结构化的格式，也允许我们在保证向后兼容的前提下去更新这种格式。

正因为这些优势，Go语言从1.8版本开始，把所有profile相关的信息生成工作都交给protocol buffers来做了。这也是我们在上述概要文件中，看不到普通文本的根本原因了。

Protocol buffers的用途非常广泛，并且在诸如数据存储、数据传输等任务中有着很高的使用率。不过，关于它，我暂时就介绍到这里。你目前知道这些也就足够了。你并不用关心`runtime/pprof`包以及`runtime`包中的程序是如何序列化这些概要信息的。

继续回到怎样生成概要文件的话题，我们依然通过具体的问题来讲述。

我们今天的问题是：怎样让程序对CPU概要信息进行采样？

这道题的典型回答是这样的。

这需要用到`runtime/pprof`包中的API。更具体地说，在我们想让程序开始对CPU概要信息进行采样的时候，需要调用这个代码包中的`StartCPUProfile`函数，而在停止采样的时候

则需要调用该包中的StopCPUProfile函数。

问题解析

runtime/pprof.StartCPUProfile函数（以下简称StartCPUProfile函数）在被调用的时候，先会去设定CPU概要信息的采样频率，并会在单独的goroutine中进行CPU概要信息的收集和输出。

注意，StartCPUProfile函数设定的采样频率总是固定的，即：100赫兹。也就是说，每秒采样100次，或者说每10毫秒采样一次。

赫兹，也称Hz，是从英文单词“Hertz”（一个英文姓氏）音译过来的一个中文词。它是CPU主频的基本单位。

CPU的主频指的是，CPU内核工作的时钟频率，也常被称为CPU clock speed。这个时钟频率的倒数即为时钟周期（clock cycle），也就是一个CPU内核执行一条运算指令所需的时间，单位是秒。

例如，主频为1000Hz的CPU，它的单个内核执行一条运算指令所需的时间为0.001秒，即1毫秒。又例如，我们现在常用的3.2GHz的多核CPU，其单个内核在1个纳秒的时间里就可以至少执行三条运算指令。

StartCPUProfile函数设定的CPU概要信息采样频率，相对于现代的CPU主频来说是非常低的。这主要有两个方面的原因。

一方面，过高的采样频率会对Go程序的运行效率造成很明显的负面影响。因此，runtime包中SetCPUProfileRate函数在被调用的时候，会保证采样频率不超过1MHz（兆赫），也就是说，它只允许每1微秒最多采样一次。StartCPUProfile函数正是通过调用这个函数来设定CPU概要信息的采样频率的。

另一方面，经过大量的实验，Go语言团队发现100Hz是一个比较合适的设定。因为这样做既可以得到足够多、足够有用的概要信息，又不至于让程序的运行出现停滞。另外，操作系统对高频采样的处理能力也是有限的，一般情况下，超过500Hz就很可能得不到及时的响应了。

在StartCPUProfile函数执行之后，一个新启用的goroutine将会负责执行CPU概要信息的收集和输出，直到runtime/pprof包中的StopCPUProfile函数被成功调用。

StopCPUProfile函数也会调用runtime.SetCPUProfileRate函数，并把参数值（也就是采样频率）设为0。这会让针对CPU概要信息的采样工作停止。

同时，它也会给负责收集CPU概要信息的代码一个“信号”，以告知收集工作也需要停止了。

在接到这样的“信号”之后，那部分程序将会把这段时间内收集到的所有CPU概要信息，全部写入到我们在调用StartCPUProfile函数的时候指定的写入器中。只有在上述操作全部完成之后，StopCPUProfile函数才会返回。

好了，经过这一番解释，你应该已经对CPU概要信息的采样工作有一定的认识了。你可以去看看demo96.go文件中的代码，并运行几次试试。这样会有助于你加深对这个问题的理解。

总结

我们这两篇内容讲的是Go程序的性能分析，这其中的内容都是你从事这项任务必备的一些知识和技巧。

首先，我们需要知道，与程序性能分析有关的API主要存在于runtime、runtime/pprof和net/http/pprof这几个代码包中。它们可以帮助我们收集相应的性能概要信息，并把这些信息输出到我们指定的地方。

Go语言的运行时系统会根据要求对程序的相关指标进行多次采样，并对采样的结果进行组织和整理，最后形成一份完整的性能分析报告。这份报告就是我们一直在说的概要信息的汇总。

一般情况下，我们会把概要信息输出到文件。根据概要信息的不同，概要文件的种类主要有三个，分别是：CPU概要文件（CPU Profile）、内存概要文件（Mem Profile）和阻塞概要文件（Block Profile）。

在本文中，我提出了一道与上述几种概要信息有关的问题。在下一篇文章中，我们会继续对这部分问题的探究。

你对今天的内容有什么样的思考与疑惑，可以给我留言，感谢你的收听，我们下次再见。

戳此查看Go语言专栏文章配套详细代码。

上一篇　47 | 基于HTTP协议的网络服务

下一篇　49 | 程序性能分析基础（下）

精选留言 2

吴超
1543559721

您这个和我在程序中直接 import _ "net/http/pprof" ,然后开一个HTTP服务查看pprof有什么区别啊 。
下面是http://127.0.0.1:6060/debug/pprof/HTTP服务返回的结果

/debug/pprof/

Types of profiles available:
Count Profile
2 allocs
0 block
0 cmdline
1029 goroutine
2 heap
0 mutex
0 profile
16 threadcreate
0 trace
full goroutine stack dump
Profile Descriptions:

allocs: A sampling of all past memory allocations
block: Stack traces that led to blocking on synchronization primitives
cmdline: The command line invocation of the current program
goroutine: Stack traces of all current goroutines

heap: A sampling of memory allocations of live objects. You can specify the gc GET parameter to run GC before taking the heap sample.
mutex: Stack traces of holders of contended mutexes
profile: CPU profile. You can specify the duration in the seconds GET parameter. After you get the profile file, use the go tool pprof command to investigate the profile.
threadcreate: Stack traces that led to the creation of new OS threads
trace: A trace of execution of the current program. You can specify the duration in the seconds GET parameter. After you get the trace file, use the go tool trace command to investigate the trace.

虢國技醬
1566628021

打卡