Gperftools性能分析工具使用教程

[Leave a reply](http://bitjoy.net/2017/02/07/gperftools-tutorial/#respond)

[上一篇博客简单介绍了Linux平台下不同性能分析工具的特点](http://bitjoy.net/2017/02/07/introduction-to-performance-analysis-tools-in-linux/)，最后是Google出品的gperftools胜出。今天我们将举一个用gperftools对链接库进行性能分析的例子。

假设我们创建了一个TestGperftools项目，其中包括一个动态链接库模块complex和主程序main，build文件夹用于存放动态链接库和可执行程序，文件结构如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | czl@ubuntu:~/czl/TestGperftools$ tree  .  ├── build  ├── complex  │   ├── complex.cpp  │   ├── complex.h  │   └── Makefile  ├── main  │   ├── main.cpp  │   └── Makefile  └── Makefile    3 directories, 6 files |

其中complex借用了[这篇博客](http://www.cppfans.com/articles/basecalc/c_pi_10000.asp)的第三个公式计算圆周率π，主要用来模拟一个很耗时的操作。该模块会生成一个libcomplex.so动态链接库供外部调用。其complex.cpp函数如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #include "complex.h"    Complex::Complex() {  }    void Complex::recordPI(double x) {      pis.push\_back(x);      if (pis.size() > 10000) pis.clear();  }    double Complex::calculatePI(int n) {      double x = 2, z = 2;      int a = 1, b = 3;      while (n--) {          z = z\*a / b;          x += z;          a++;          b += 2;          recordPI(x);      }      return x;  }    Complex::~Complex() {  } |

calculatePI为计算π的函数，传入参数n表示循环计算的次数，也就是第三个公式中累加的次数，当然次数越多，精度越高，时间越长。recordPI是刻意加上去的一个函数，作为calculatePI的被调函数，同时包含STL的一些操作。该模块的Makefile如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | SRC = complex.cpp  OBJS = $(SRC:.cpp=.o)  BUILD\_DIR = ../build  TARGET = libcomplex.so    all: $(TARGET)    clean:      -rm -f $(OBJS) $(TARGET)    $(OBJS): $(SRC)      $(CXX) -fPIC -c $(SRC)    $(TARGET): $(OBJS)      $(CXX) -shared -o $(TARGET) $(OBJS)      cp $(TARGET) $(BUILD\_DIR) |

注意生成动态链接库的Makefile格式，编译时需要-fPIC生成位置无关目标文件，链接时需要-shared选项。由Makefile文件可知，该模块会生成libcomplex.so动态链接库。

下面我们来看一下main.cpp文件：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include"../complex/complex.h"  #include<iostream>  #include<iomanip>  using namespace std;    vector<long long> vll;    void recordSum(long long s) {      vll.push\_back(s);      if (vll.size() > 10000) vll.clear();  }    long long calculateSum(long long n) {      long long s = 0;      for (int i = 1; i <= n; ++i) {          s += i;          recordSum(s);      }      return s;  }    int main() {      long long n = 1000000000;      Complex c;      double p = c.calculatePI(n);      long long s = calculateSum(n);      cout << "pi=" << fixed << setprecision(13) << p << endl;      cout << "1+2+...+" << n << "=" << s << endl;      return 0;  } |

main函数除了调用Complex类计算π之外，还会调用自身的calculateSum计算1+2+...+n的和，并且也会调用一个刻意的recordSum函数。该模块的Makefile如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | SRC = main.cpp  OBJS = $(SRC:.cpp=.o)  BUILD\_DIR = ../build  LIBS = -lcomplex  TARGET = main    all: $(TARGET)    clean:      -rm -f $(OBJS) $(TARGET)    $(OBJS): $(SRC)      $(CXX) -c $(SRC)    $(TARGET): $(OBJS)      $(CXX) -L$(BUILD\_DIR) -o $(TARGET) $(OBJS) $(LIBS)      cp $(TARGET) $(BUILD\_DIR) |

可以看到该模块链接了-lcomplex动态链接库，注意第16行链接的顺序不能乱，-L要在-o前面，$(OBJS)要在$(LIBS)的前面。

编译的方法为：直接在TestGperftools目录中执行make，主make会分别进入complex和main文件夹执行各自的子make，最后动态链接库libcomplex.so和可执行程序main都复制到了build文件夹中。具体代码请看根目录中的Makefile文件，这里不再列出。

接下来我们要安装gperftools性能分析工具，在Ubuntu下安装非常简单，直接执行以下命令，该命令安装了两个软件，google-perftools和graphviz，前者就是gperftools，后者是用于可视化性能分析结果的。

|  |  |
| --- | --- |
| 1 | sudo apt-get install google-perftools graphviz |

下面我们开始对main进行性能分析，我们预期libcomplex.so中的calculatePI会是主要的性能瓶颈，看看gperftools给出的结果是否符合预期。

1. 进入到build文件夹，执行以下命令：

|  |  |
| --- | --- |
| 1 | $ export LD\_LIBRARY\_PATH=. |

1. 该命令表示查找动态链接库的首选位置为当前路径，如果没有设置，在执行main程序时，会找不到libcomplex.so而报错：./main: error while loading shared libraries: libcomplex.so: cannot open shared object file: No such file or directory
2. 执行以下命令：

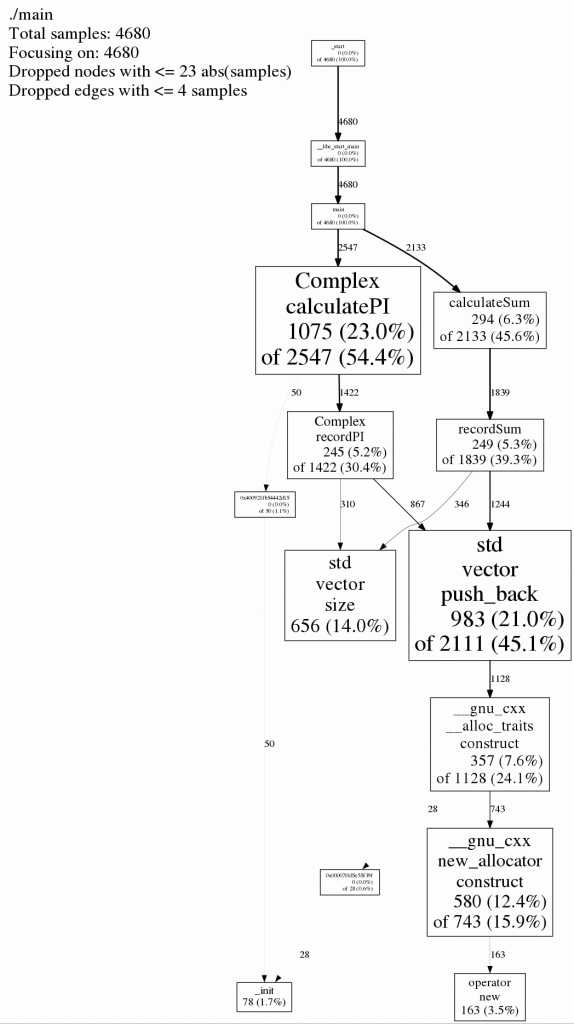
|  |  |
| --- | --- |
| 1 | $ LD\_PRELOAD=/usr/lib/libprofiler.so.0 CPUPROFILE=./main.prof ./main |

1. LD\_PRELOAD用于预加载gperftools的动态链接库，有些发行版可能是/usr/lib/libprofiler.so，我的是后面有.0，也有可能是其他数字；CPUPPROFILE用于指定生成的性能分析结果文件路径，我设置的是在当前文件夹下生成main.prof；最后接上需要分析的可执行程序./main，如果你的可执行程序由命令行参数，也可以一并跟在./main后面。
2. 程序执行结束之后，输出如下：

|  |  |
| --- | --- |
| 1  2  3 | pi=3.1415926535898  1+2+...+1000000000=500000000500000000  PROFILE: interrupts/evictions/bytes = 4680/1585/112344 |

1. 前两行是程序输出，最后一行是gperftools输出。
2. 最后进行性能分析，执行以下命令：

|  |  |
| --- | --- |
| 1 | $ google-pprof --web ./main ./main.prof(废弃)  pprof --text ./main ./main.prof > a.text  pprof --pdf ./main ./main.prof > a.pdf |

1. --web表示性能分析结果以网页的形式展示，后面依次跟可执行程序和上一步生成的\*.prof文件。程序自动打开浏览器，性能分析结果如下图所示。  
   [](http://bitjoy.net/wp-content/uploads/2017/02/profiler-output.gif)详情页中，左上角给出了汇总结果，总共采样了4680次，扔掉了采样次数小于23和4次的节点和边。gperftools 性能分析图中，每个节点代表一个函数，节点内包含三个信息，从上往下依次为：函数名、该函数自身被采样到的次数（及占总采样数的比例）、该函数和被调函数被采样到的次数（及占总采样数的比例），节点的大小正比于该函数自身被采样到的次数；每条边代表调用者到被调用者被采样到的次数。简而言之，图中节点越大，该函数自身耗时占比越大，越可能是性能瓶颈。  
   由图可知，Complex::calculatePI确实是性能瓶颈，占用了23.0%的时间，而calculateSum只占用了6.3%的时间。另外，我们刻意增加的recordPI和recordSum也占用了不少的时间，因为需要不断的push\_back以及重新分配内存，所以占用的时间也很多。
2. 总的来说，这样的分析结果是符合预期的，我们能够根据这样一个热点分布图，找到性能瓶颈，然后有针对的进行性能优化。比如针对上面的例子，我们可以优化calculatePI，如果对π的精度要求不是很高，可以减少while循环的次数；对于vector的push\_back，因为我们是大于10000时会clear一下，为了避免多次的clear和reallocate，可以预先设置vector的大小为10000，然后重复对0~9999下标进行填充，这样就只需一次分配内存了。

本博客完整的TestGperftools项目可以查看我的Github：[TestGperftools](https://github.com/01joy/TestGperftools)。

gperftools套装中还包括a memory leak detgector和a heap profiler，用于检查程序中是否有内存泄漏的风险以及对内存堆栈的分析，具体介绍可以查看[其Github上的WIKI](https://github.com/gperftools/gperftools/wiki)。

一些有用的链接：

1. 3步学会使用gperftools的超简易教程：<https://wiki.geany.org/howtos/profiling/gperftools>
2. gperftools官方文档：<https://gperftools.github.io/gperftools/cpuprofile.html>

This entry was posted in [0和1](http://bitjoy.net/category/0%e5%92%8c1/) and tagged [gperftools](http://bitjoy.net/tag/gperftools/), [性能分析](http://bitjoy.net/tag/%e6%80%a7%e8%83%bd%e5%88%86%e6%9e%90/) on [February 7, 2017](http://bitjoy.net/2017/02/07/gperftools-tutorial/).