系统性能的评估维度可能很多，包括应用的吞吐量、响应时间、任务完成时间和资源利用率等。但是这些指标（metrics）仅仅是表象，一旦发现异常，如何从代码级别定位性能问题才是解决问题的关键，本文介绍了一种使用火焰图（Flame Graph）来做性能分析的方法，在实战中具备很高的可操作性和快速pinpoint问题的能力。

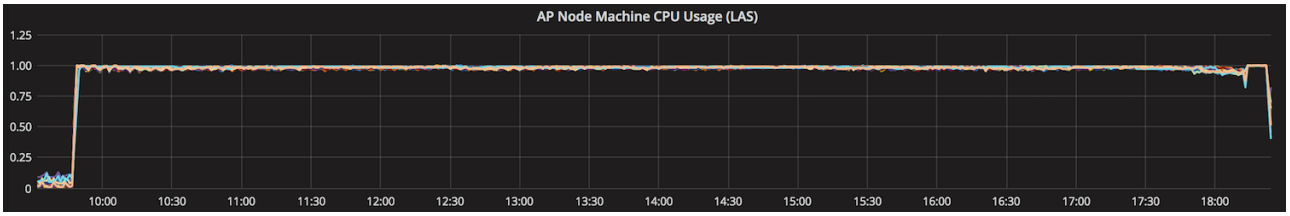
下面按照1. 发现问题，2. 分析问题，3. 解决问题三个章节展开，最后是4. Lesson Learned。

# 1. 发现问题

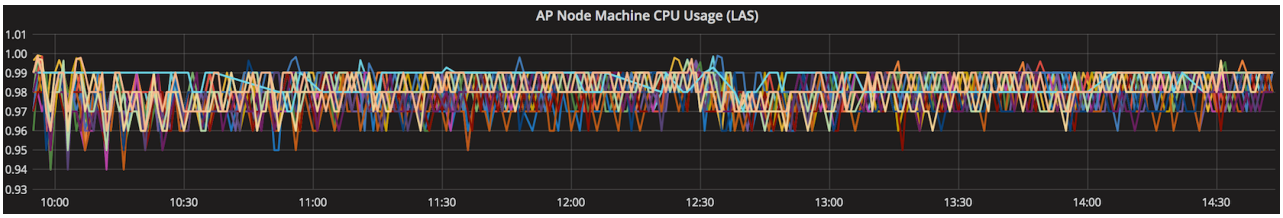
最近公司完成大数据集群的迁移，应用大多是Spark开发的，但是仍然存在一个老的每日运行的Hadoop任务突然发现指标异常，主要体现在

1. 任务完成时间超长。超过一天仍未完成。

2. CPU占用率几乎到达100%。由于Hadoop集群label化了，所以只在一个子集的node节点上会运行该程序，暂时和其他应用隔离开来，使用Zabbix+Grafana收集的CPU占用率指标如下图。




放大中间部分，可以看出每台机器的CPU都基本吃满了。




# 2. 分析问题

由于该Hadoop程序已稳定运行N年，代码一直没动，所以排除人为bug导致。另外的变量就是集群环境的变化，由于新集群的机器配置都比较高，48核CPU+256G内存，而程序设置的mapreduce.map.memory.mb和mapreduce.map.cpu.vcores参数都保持不变，导致一台物理机器上可运行的实例数比之前多了很多，CPU资源不足，所以打满了CPU。

那么被动的去调整资源，不如深入去主动的寻找程序中的优化点，以提高资源利用率，毕竟资源都是要烧钱的。

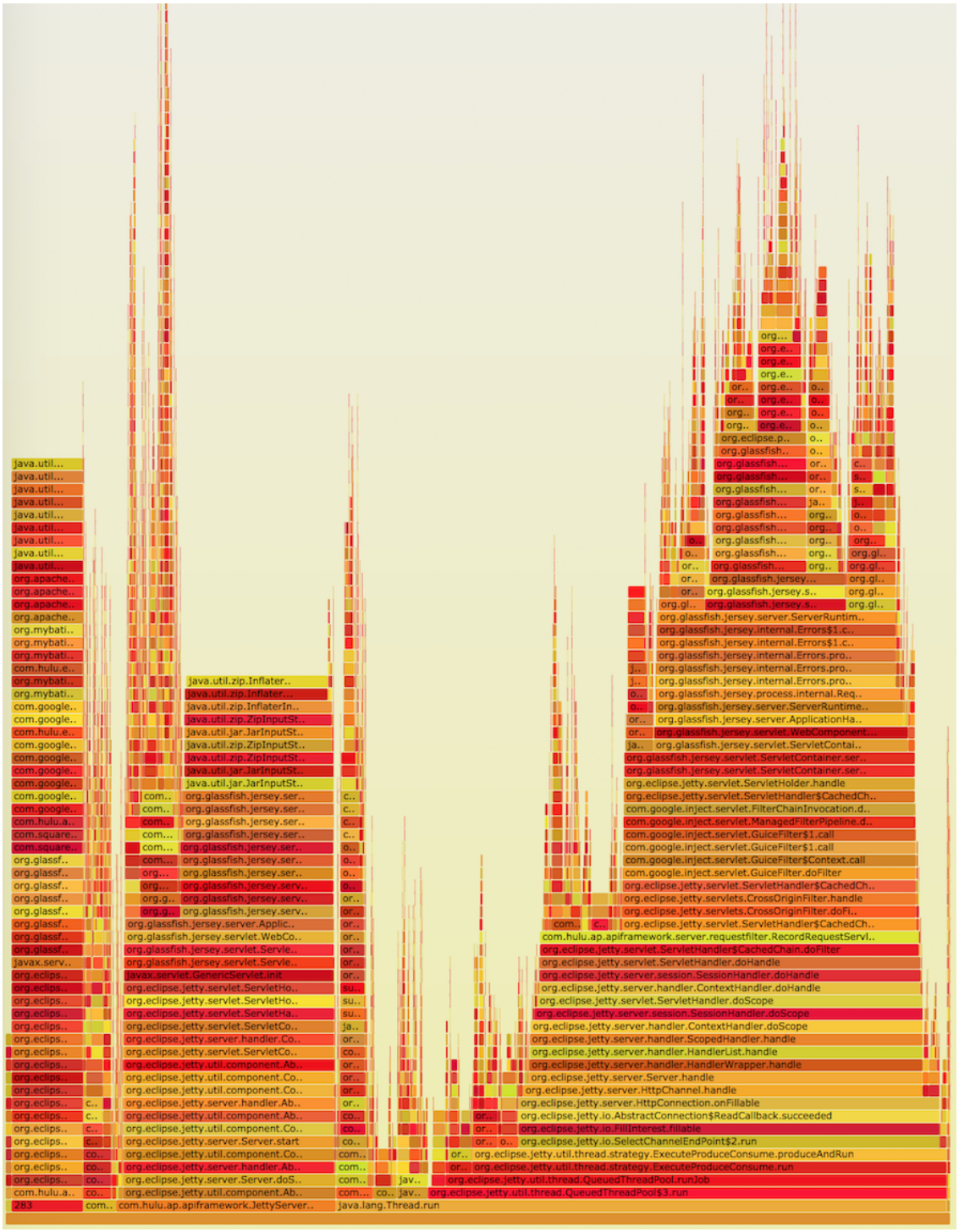
老的Hadoop程序业务逻辑很复杂，真去深究恐怕收获不大，所以要考虑一种“一击致命”的方案，pinpoint症结即可。

此时火焰图登场，火焰图，英文叫做Flame Graph，是Brendan Gregg发明的，详细看大师的博客。

这里简短介绍，火焰图可以提供可视化的性能分析能力，可以迅速定位最热的code-path，以SVG的格式的图片展示，可缩放查看局部。火焰图支持不同的分析类型，包括CPU、Off-CPU、内存等等，本文遇到的问题由于CPU是瓶颈，故主要采用CPU这种类型做性能分析。火焰图使用一些系统工具来做采样收集数据，例如Linux上用perf, SystemTap，Windows用Xperf.exe，然后对收集的数据进行处理、转化、最终渲染成一张SVG图片。

本文遇到的问题是生产环境直接CPU满负荷，如果在实际测试系统中可以采用ab等工具压测模拟，一般瓶颈不是CPU就是I/O，如果压力很大CPU仍然有富余，那么就需要看下Off-CPU类型的分析，看I/O卡在哪。

下图就是一张我压测一个Jetty HTTP Server的CPU类型的火焰图，火焰图展示了在采样周期内，code-path被执行的时间占比，纵轴是code-path，从下至上一般就是调用栈，相同的采样调用栈会被合并，栈顶元素就是采样的时候CPU运行的stack，横轴是某个stack的占用时间，跨度越大说明其占CPU比重越高，也就是最耗费CPU的，stack在横轴上是按照字母序排列的，颜色深浅仅仅是为了区分，并没特殊意义。




上图可以看出很多“尖峰”，这算是比较正常的案例。需要特别关注火焰图中的“平顶山”，往往代表着某段代码的调用占用CPU过多。

那么如何收集一个Hadoop或者Spark作业的火焰图呢？我的思路是首先在某台机器上安装SystemTap，然后使用Jeremy Manson提供了一个工具Lightweight Asynchronous Sampling Profiler生成一个so，在Hadoop或者Spark作业的JAVA_OPTS上加上agent参数，对class字节码进行instrumentation，从而监控JVM程序的运行，输出采样数据，也就是stack trace采样，最后使用Brendan Gregg的Flame Graph工具生成SVG图片。

## Step1. 安装SystemTap

我的环境是ubuntu，安装步骤参考链接，只需要执行Systemtap Installation和Where to get debug symbols for kernel X?这两部分的步骤即可。

然后验证下是否安装成功，执行如下打印hello world即可。

```
sudo stap -e 'probe kernel.function("sys_open") {log('
```

## Step2. 生成agent文件

Jeremy Manson提供了一个工具Lightweight Asynchronous Sampling Profiler。目前只支持hotspot JVM，可以从git clone https://github.com/dcapwell/lightweight-java-profiler，然后执行make all。注意可以在src/globasl.h中设定参数。

```
// 每秒采样点数
static const int kNumInterrupts = 100;

// 最大采样的stack数量
static const int kMaxStackTraces = 3000;

// 一个采样stack的最大深度
static const int kMaxFramesToCapture = 128;
```

## Step3. 修改Hadoop或者Spark作业参数

在JAVA_OPTS中加入如下参数。其中liblagent.so就是Step2. make all之后，在build-64目录下生成的文件。

```
 -agentpath:path/to/liblagent.so
```

如果只在某些机器上安装了so，而分布式作业，往往是成百上千的Task并行执行，那么可以调整max failure percentage，即使某些task fail fast，那么也不影响采样，例如Hadoop的参数是mapreduce.map.failures.maxpercent，可以设置成99。Spark 2.2.0中可以尝试spark.task.maxFailures参数。

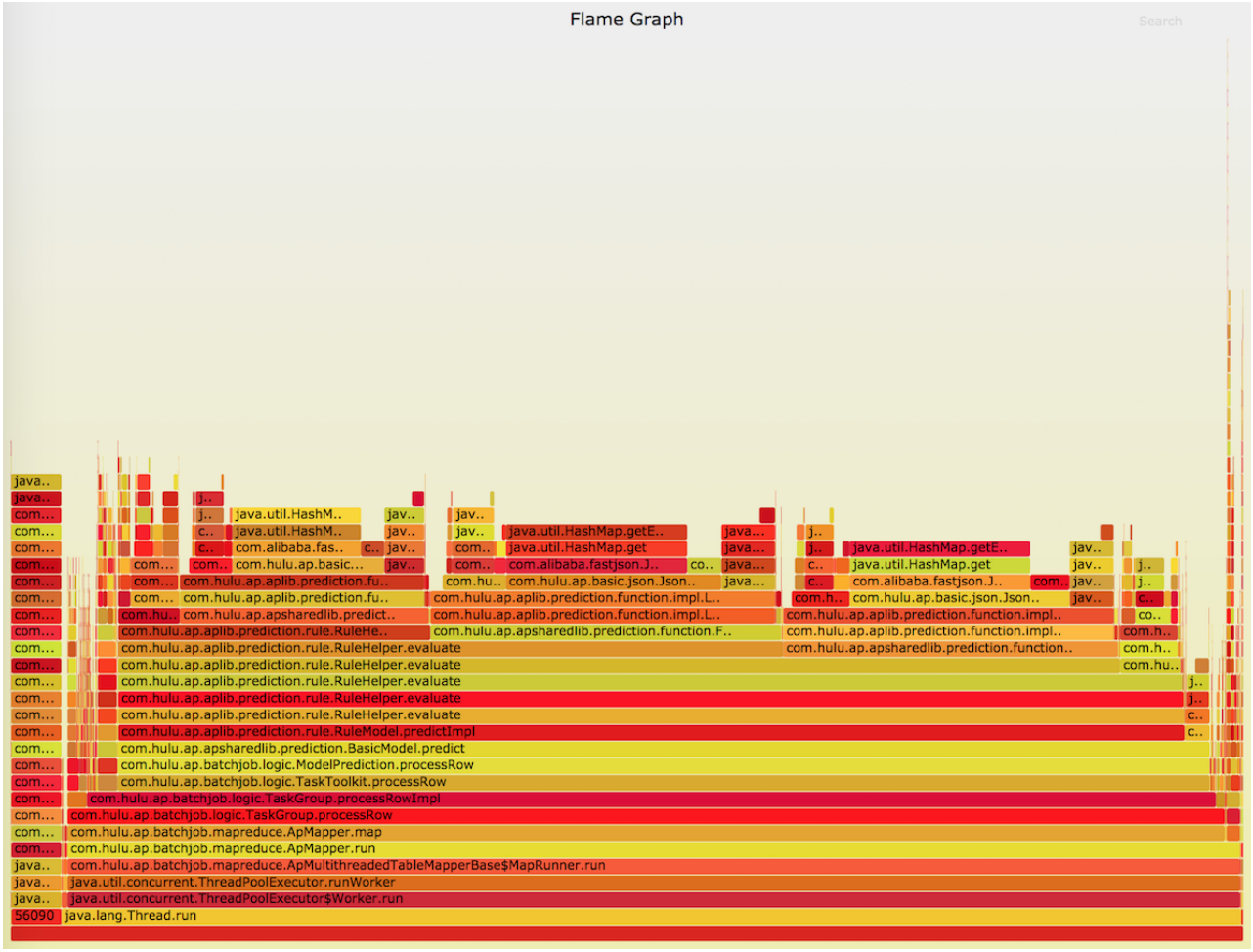
注意启动之后，会在启动目录下生成空的traces.txt文件，只有程序执行完毕才会把采样数据写入文件，所以在程序运行期间做一个链接，使用ln命令即可，这样即使task执行完毕被YARN clean up，也不至于丢掉采样数据。

## Step4. 生成火焰图

使用Brendan Gregg的Flame Graph工具生成SVG图片。

```
git clone http://github.com/brendangregg/FlameGraph
cd FlameGraph
./stackcollapse-ljp.awk < ../traces.txt | ./flamegraph
```

我司内有问题的Hadoop作业的火焰图如下，正如猜测，“平顶山”清晰可见，几乎都是用户自己的代码类，最右边很细条的是Hadoop类的调用栈，基本不占用CPU，可以通过stack清晰的看到究竟是什么类的、什么方法疯狂的占用了CPU，从而有的放矢的进行优化。




由于解决的思路比较trivial，并且涉及公司内的应用，所以暂且简单说下，我们的程序会使用一个DSL来做数据的过滤，每天过滤的row高达8000亿条记录，而DSL的嵌套又会加剧调用次数，数据无schema，在运行时做Json解析，所以大量的CPU都在做Json解析，可以看到栈顶都是util.HashMap#get(..)方法。另外，不合理的使用Java的集合框架，也是加剧了性能的恶化，例如会看到util.ArrayList#add(..)扩容上浪费了很多CPU时间。

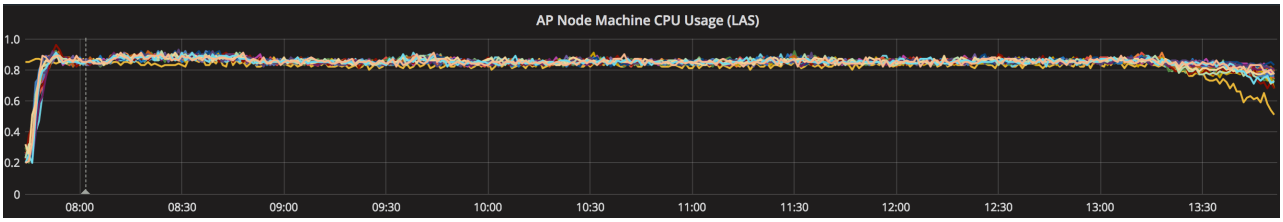
# 3. 解决问题

这个章节不过多展开，因为一旦定位了问题，总会有办法解决的，关键就是用火焰图帮助我们大海捞针式的一击致命找到症结。

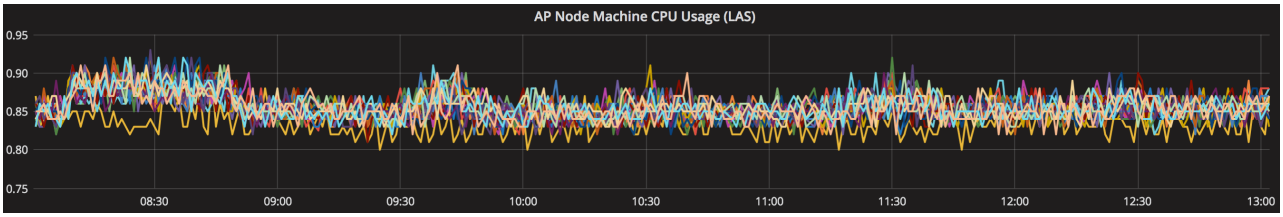
代码层面的解决方案如下：
1）数据序列化方式改进。使用Protocol Buffer代替Json，避免运行时做低效的反序列化。
2）DSL的过滤分为逻辑执行计划，和物理执行计划两部分，在逻辑执行计划内做优化，flatten化多个同样的过滤函数，最优化CPU pipeline，减少调用次数。
3）List的新建加入initial capacity，防止膨胀拷贝。

资源调整方面的解决方案如下：
1）合理设置mapreduce.map.memory.mb和mapreduce.map.cpu.vcores参数，让资源充分利用，并且不至于压垮机器。

优化后，从Grafana上的截图可以看出，整个作业的运行时间不仅CPU占用降低了，而且整体的运行时间也缩短了。




放大局部，占用率保持在85%左右，最大化资源利用率。




# 4. Lesson Learned

解决问题的能力非常重要，问题虽然千变万化，但是发现他们的手段无非就几种，各种profiling，debugging和tuning工具都可以对应用进行全面的“体检”，包括CPU、内存、网络I/O等，通过这些具体system/application level的指标监控，可以发现问题的症结。本文中没有采用JVM的一些常用的内存和进程分析工具，转而使用火焰图，所以一下就可以找到问题。解决问题的思路，第一，优化应用本身，可以通过代码级别，或者像Hadoop/Spark作业，算子的合理使用也可以大大提升性能，第二，优化资源调参，这需要建立在对building blocks和自己开发的应用程序有一定的认识和理解上。

总之，发现问题不可怕，可怕的是找不到根源，希望本文可以给读者一种思路，通过一个特定的性能分析工具集，可以帮助我们pinpoint问题，从而不断的缩小搜索范围，最终定位问题，做有针对性的优化。

# 参考资料

http://www.brendangregg.com/flamegraphs.html
https://www.slideshare.net/brendangregg/blazing-performance-with-flame-graphs
https://huoding.com/2016/08/18/531
http://tacy.github.io/blog/2014/07/16/FlameGraph/
http://dmdgeeker.com/post/flamegraph/