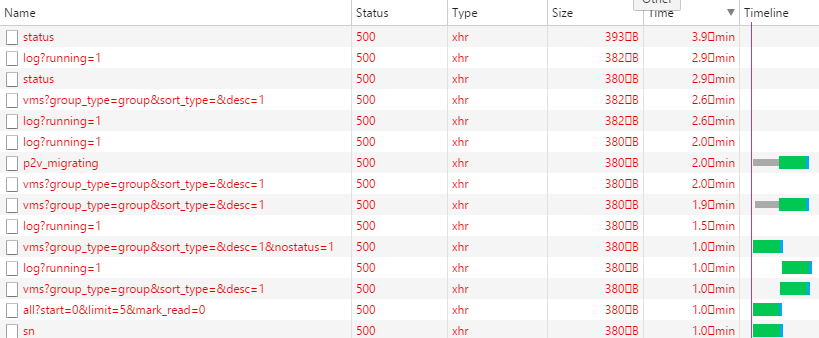
# aSV WEBUI响应速度优化

[mnstory.net](http://mnstory.net)

## 现象

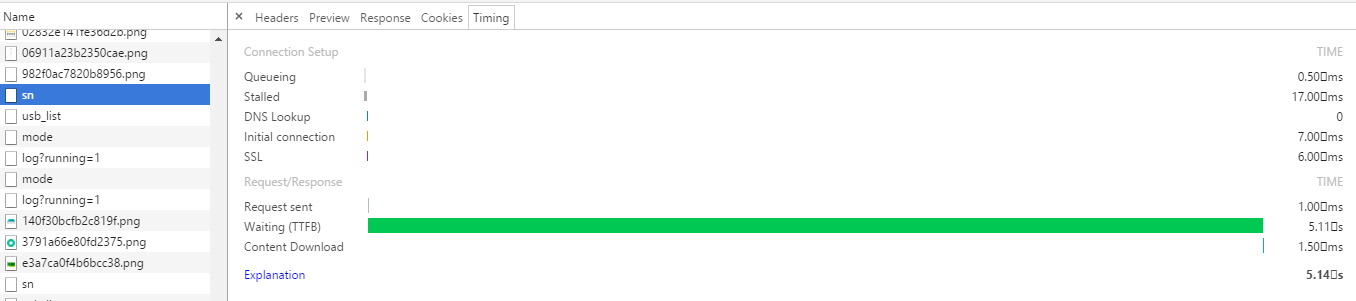
前端UI卡，卡到什么程度，基本上，在线用户一多，会10s以上没有反应，说10s是比较客气的，毕竟也是我们自己部门做的软件，不能自责太多，如果看我们的公网演示环境，现象可能是直接打不开，我优化性能的环境是测试中心真实设备，该设备一共6台主机组成集群，有315台虚拟机，其中运行的虚拟机为81台。

我们用CHROME看看，在没有优化的情况下，主要请求所耗时间，容许我贴一张48台主机的VMP集群，在开了10个页面后然后关闭其他9个的效果（看Time列）：



我们先看下获取序列号的API /cluster/sn，这个API稳定，不会因为CPU，虚拟机数量等有太大的变化，但是和主机数量有关。

用CHROME看一下数据请求，发现TTFB(time to first byte)耗时5.11s：



这时我们登陆后端，用命令行工具请求API：

time vtpsh get /cluster/sn

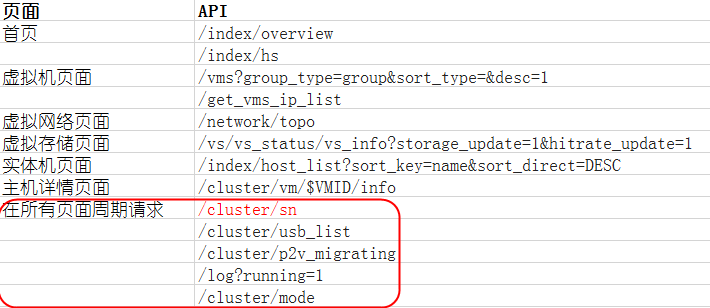
结果是耗时2s。

这说明：

1. 前端请求比后端耗时，说明vtpsh与前端的差异路径上有瓶颈。
2. 后端耗时2s，说明这个API真的有点慢。

## 影响API

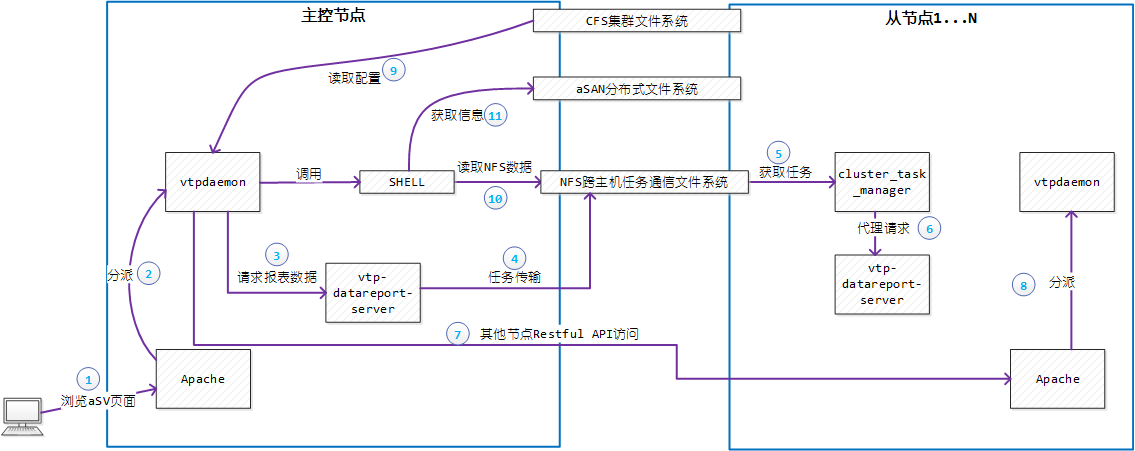
用CHROME分析，影响页面显示慢请求都有哪些：



不详细介绍这些API影响哪个视图，总之，这些API响应慢，导致了前端久久无DOM更新。

## aSV请求响应原理

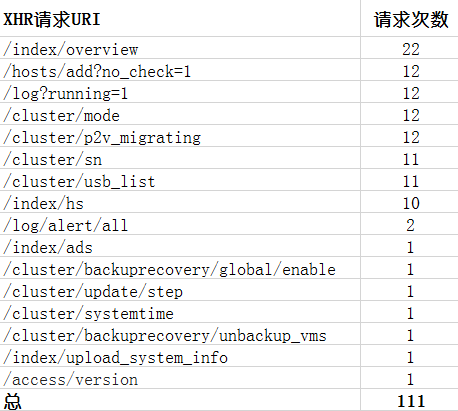
经过几日优化，我理了一下主要的请求响应流程，我将图上的步骤用序号显示出来，如下：



我们来看一下，第1、2点，从前端请求apache，apache又将PERL任务分配给vtpdaemon进程，如果只有一个用户打开一个页面，会是什么现象，我们打开主页，等待一分钟，发现主页请求XMLHttpRequest 111条：



再细看下这一分钟主页里面XHR都请求了哪些URI：



从上表可以看出在12次附近的请求颇多，掐指一算，应该是5s一次的请求周期 60/5 = 12，如果是定时的5s一次，那应该不会出现11次10次的请求，那表示5s一次是请求间隙，不是固定请求周期。

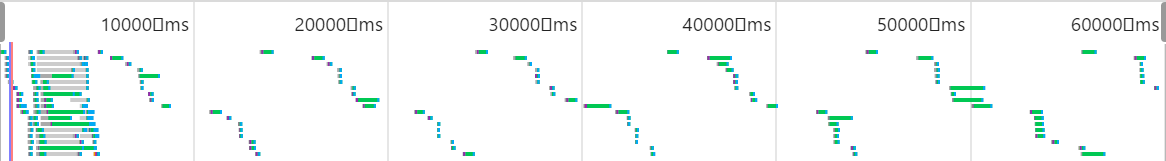
请求间隙表示发出请求、等待返回、处理内容后再次请求的间隙，从上表看/index/hs这个家伙很耗时，因为都是5秒的间隙，别人都请求了12次，你才10次！

当然，细心的观众应该会觉得，既然5s的间隙1分钟能请求12次，表示每次TTFB(time to first byte)不到一秒，还需要优化？你还能怎么优化？

你是对的，如果真是这样，也不用操刀，但是，上门的数据是我优化后的，不是优化前的，优化前的数据只看了看没有记录下来！

事实上，优化前的数据是很糟糕的，随便抓一个网页看下，5s一次TTFB算是不错的，有30s未返回的。

我们再看看，这一分钟的请求都是怎么发的：



除了前面有点崩溃的并发外，后面的周期都和大姨妈一样精准。

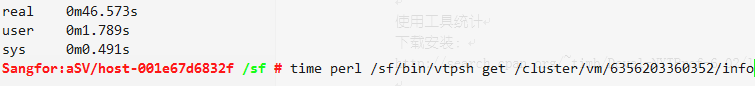
你说，简单的一个主页，你用得着这样子吗？1分钟请求111次XHR，是什么样的并发需求，这是前端的强烈宣言吗？

你以为这样够了？没有，你看看前端是怎么虐后端的，当然后端也太不争气，100多条请求就扛不住了。



我很辛苦的统计了上面一个表格（是的，很辛苦，不要问我是怎么统计的，全靠两只手）。上面表格是每个页面1分钟内请求次数汇总，我们只看XHR，为什么只看XHR？因为非XHR请求apache自己就能够搞定，这点数量，不够apache塞牙缝，XHR需要后端PERL代码响应，你要是看过那个PERL代码，你会明白的……如果你不明白，一会我还要逐个分析。

我们模拟一个普通用户的常规操作，假设用户和我一样小家子气，只开了一个“虚拟机页面”，一个“虚拟机网络页面”，一个“主机详情页面”，二个“虚拟机详情页面”，二个“VNC控制台页面”，如果他只开了10分钟，我们算算，这10分钟一共有4290次XHR请求，其实不算什么，但是，你要知道，我们有忒多忒多的请求耗时不是1秒，不是5秒，不是10秒，而是：



恩，获取虚拟机详情，只花了46秒。

假设测试中心同时有3个用户在使用，那么他们骂娘的概率应该会很大，假设公司公网演示环境同时30个用户在使用，那么他们不会骂了，自己死了算了。

好了，危言耸听这么久了，开始优化！

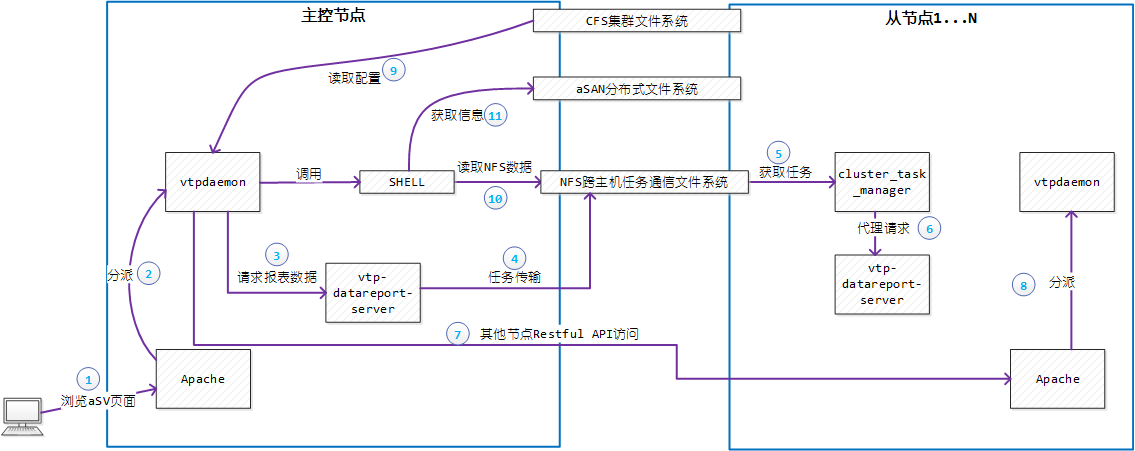
我们主要用NYTProf来辅助定位性能，运行方法就是，在你的perl参数里面加入-d:NYTProf，当然，你得安装这个包：

time perl -d:NYTProf /sf/bin/vtpsh get /cluster/vms --group\_type=group --sort\_type --desc=1

nytprofhtml

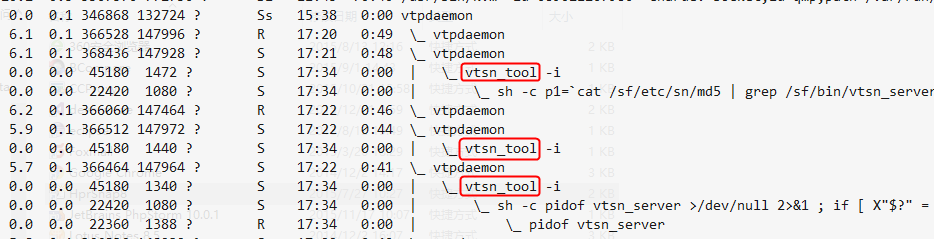
## 优化1：放大vtpdaemon处理进程数量

我们再把上门那副图贴一贴（为什么又要贴？因为画的漂亮！）



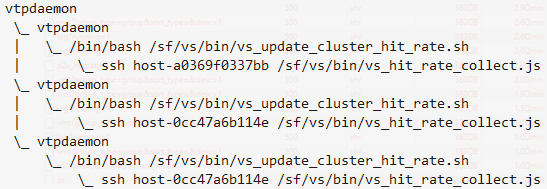
再来看看，第1、2点，从前端请求apache，apache又将PERL任务分配给vtpdaemon进程，在vtpdaemon的监控进程里，定义的vtpdaemon最多只能有3个子进程，3个进程阻塞处理这么多请求，正如厕所门口那些排队等待尿尿的美女，不论是坑和人都很急，wh之前放开到10个进程，但是，如果没人访问岂不是忒浪费，当然，能根据实际请求数来开启进程是最好的，但是，这样写代码是不是太麻烦了些，我折中下，认为放开到5个子进程是挺合适的。

来看看aSV版本6台主机，5个子进程的情况：



麻麻，这么多坑都被一个叫vtsn\_tool的家伙占着，难怪厕所外面排了这么长的队。

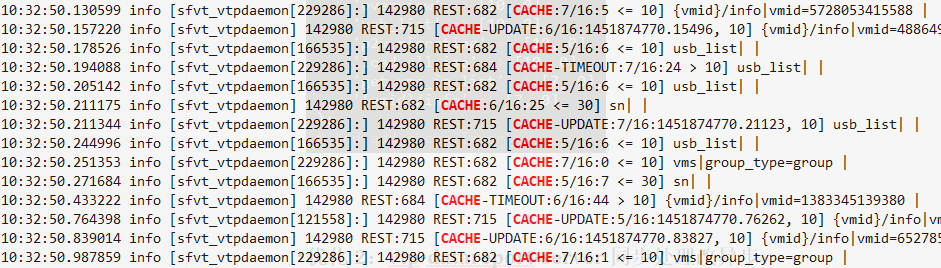
来看看VMP版本48台主机，3个进程的情况：



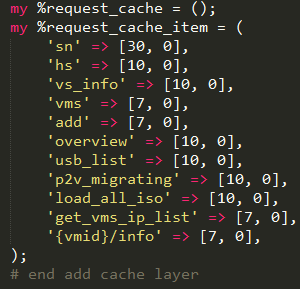
谁有便秘，了然于胸。

## 优化2：vtpdaemon加入后端cache

前端请求量比较大，这样的话每个都去做一次请求，导致消耗大量资源并且影响整体响应，下图是一秒钟后端CACHE的日志，修改后8次CACHE命中，6次未命中，总的看来，命中占了一半（因为vtpdaemon是多进程模式，所以每个进程有自己的CACHE，导致命中下降）：



实现上，我将下列请求做了CACHE，单进程两次请求的timeout是array的第一个数，第二个设置为1可以支持多用户的进入不同的CACHE：



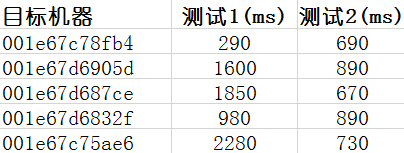
## 优化3：vtp-datareport-server同步处理改异步

我们打开“虚拟机详情页面”，看到右上角那个资源状态图，他可能来自“从节点”也可能来自“主控节点”。

来自主控节点的比较好办，看看图中3这条路径，vtpdaemon直接向vtp-datareport-server请求数据，vtp-datareport-server已经准备好，直接回复即可；如果这台虚拟机不是运行在主控，那可能就在某从节点上，这时vtp-datareport-server需要向从节点请求数据，就要走4、5、6这条路径。

看看4这条路径，是将任务写入NFS文件系统，是的，我们是用的NFS来同步个主机间的数据的，简单！粗暴！

得益于NFS的稳定性，目前小规模集群，例如10台还是扛得住的，看看NFS任务分发到返回需要的时间：



简单说，5台主机，差一些的需要2.2秒，好的，需要290ms。

用NFS作为传输层，目前已经显示出瓶颈，但是我不愿意修改，因为……没有时间改啊！！！

当NFS将数据同步到目标主机后，目标主机有个叫cluster\_task\_manager的死轮程序，会不停去自己主机的目录里面查找是否有新的task文件，如果有，将文件内容读出来，通过nc链接本机的vtp-datareport-server，等待vtp-datareport-server返回数据，然后再将结果写入NFS对应目录。

这里，也就是图上的位置6，在测试机器上有不少错误日志，我看了下代码， 这条数据处理逻辑是非常经典的，就是 select -> accept -> recv all -> process -> send all -> goto select。

而由于位置5获取任务的脚本，是死轮的，就算死轮也有周期，通过日志分析，这脚本一次性可以批量提交上百条请求，好在vtp-datareport-server的process流程不涉及阻塞操作，不过等recv all和send all的延迟下来，如果你的任务很不幸，排到了最后一位，那么等前面一百个处理完，可能天色已经不早了，这也不难理解为啥上面的获取虚拟机详情会出现46秒的情况。

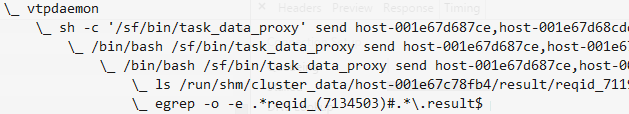
由于时间余额不足，NFS传输层不便动刀，只好修改vtp-datareport-server的网络处理逻辑，将刚才的经典模式修改为libevent的epoll模式。

效果就是，修改后基本会耗时在阻塞接受和发送上，可消除耗时峰值。

## 优化4：vtp-datareport-server接收任务正则优化

这个优化是wh做的，他的任务是，解决vtp-datareport-server占用CPU高问题，接着上面的描述，我们可以看到，当数据从NFS返回来后，就是位置4的反向操作，这里怎么从NFS获取数据？

目前的实现是，vtp-datareport-server批量将任务发送出去:



然后利用inotify监控NFS目录是否有变化，如果变化，轮训里面的task response，如果和自己发送出去的id匹配，则表示这条response是自己的。

某主机task数是600条，请求任务假设是50条，要多次匹配50x600=30000的正则表达式，自然很耗CPU，wh将其修改为查hash表的方式，有一定效果。

## 优化5：get\_vms\_ip\_list每个节点发出get请求优化

/cluster/get\_vms\_ip\_list 这个API在“虚拟机页面”用来显示虚拟机网卡IP，这些IP是用户在虚拟机上安装了aTool工具后，aTool上报给vtp-datareport-guest，然后由vtp-datareport-guest写入每个节点的tmp目录，每个节点的vtpdaemon提供一个叫/cluster/get\_vms\_ip\_info的API，这个API会去读取tmp目录下每个有aTool透传的信息，并将网口IP提取出来。

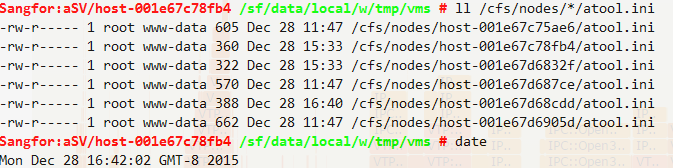
如果在主控上，要获取所有节点上安装有aTool的虚拟机的IP，怎么做？不管你怎么做，反正get\_vms\_ip\_list是向每个节点一个一个发送http get请求，访问其提供的get\_vms\_ip\_info API，然后将结果合并在一起，正如路径7、8所示。

测试环境总共6台主机，从主控向其他5台请求，优化前，耗时5.69秒：



刚好一台主机请求耗时1秒，可以推测，按当前流程，支持50台主机，这个API需要50秒才返回，前提是不要遇到connect timeout。

优化过程比较繁杂，首先，从数据源收集端vtp-datareport-guest处动刀，将网络数据单独提取出来，写入到每个节点的cfs配置目录，考虑到cfs文件系统同步成本，能不做写操作尽量不写，事实证明，没人会蛋疼老是修改IP玩，如图，在16:42查看，最早的一次改动文件在11:47：



然后我们从cfs目录读取读取文件，同样的6台主机，之前耗时5.69秒，现在耗时551毫秒：



## 优化6：overview读取cfs文件优化

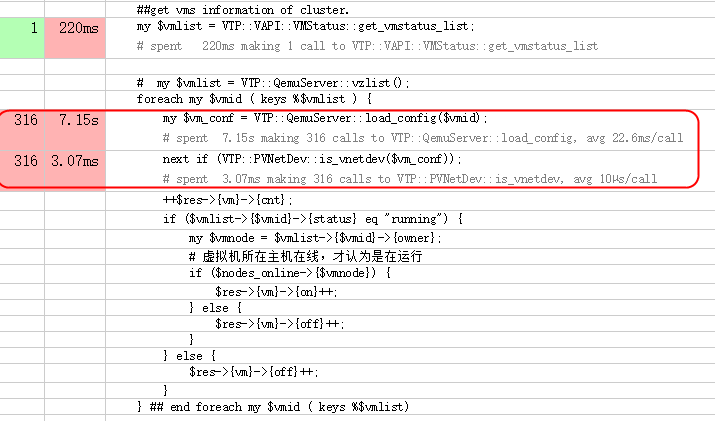
/index/overview 这个API是用来在“主页”显示虚拟机数量，主机数量等信息的，上面一个优化，我们将内容放入cfs目录，带来了飞速的处理，而overview也是从cfs目录读取文件，见图上9位置，鉴于历史上东施效颦之说，并不是所有的女人用同样的姿势都会引得男人垂涎，可以想见，也不是所有的读cfs目录都能带来飞速。

overview最大的问题是，他连续从cfs目录读了315次，为什么是315这个打假的好日子，不要想太多，因为有315台虚拟机：



好代码共赏（下面读的是316次，耗时7.67秒）：





细看一下，这两行代码其实没做啥事情，就是看看这个设备是不是网络设备，如果是，将其排除。

作为一个非专业的PERL CGI开发，我找找代码，发现在cfs目录下有一个实时更新的.vmlist文件，此文件包含了所有虚拟机信息，当然，也包含这里需要的是否是网络设备信息，替换后，315台主机遍历时间从7.67秒变为487毫秒。



## 优化7：检测nfs目录优化

vtpdaemon里面充斥了大量的命令行调用，如图上10位置流程，本来命令行调用是比较慢的，而vtpdaemon里面的run\_command函数更是发扬了这个慢，比比较慢还慢。

如果用run\_command函数去运行ls nfs dir，如你所知，nfs目录是网络目录，你ls一下人家nfs程序得多累啊，又要遍历目录又要刷CACHE又要去网络上同步数据，而我们是不停在ls，还设了个专职，叫做check\_nfs\_status的函数来刷，就一次获取虚拟机详细信息的API中，就检测了7次NFS目录是否存在，这个API耗时3.85秒，见图中11位置：

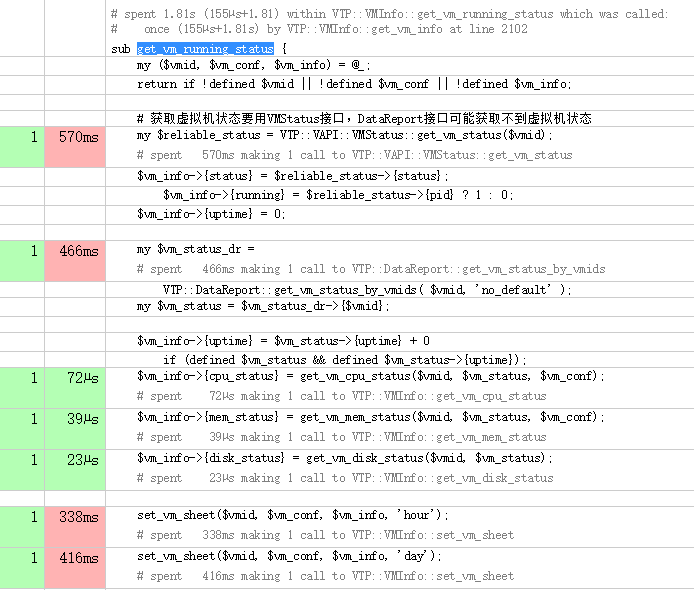


我的改法简单，用内置的-d检测nfs目录是否命令，如果有文件一定存在，用-f检测nfs目录里的文件更好。优化后，原来需要3.85秒，现在需要946毫秒。

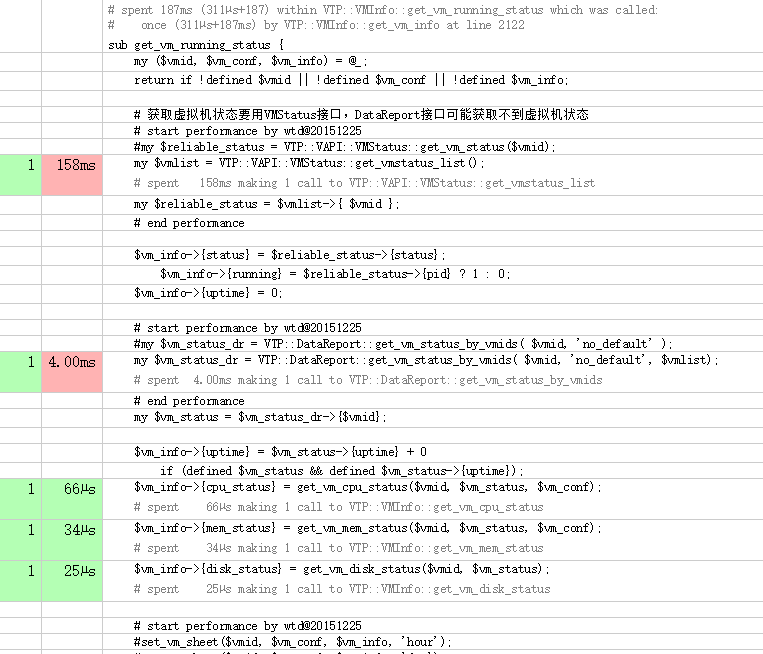


## 优化8：vmid/info参数CACHE优化

用来获取单VM的报表，涉及到好几处嵌套API调用，最耗时的是get\_vmstatus\_list这个函数，在不同的子函数里面被调用多次，看下这段代码，消耗毫秒级别的地方：



这个优化过程很简单，分析了代码后，其实是子函数获取vmlist导致耗时，将获取的内容CACHE下来，通过参数传入子函数，耗时从1.81秒降到187毫秒。



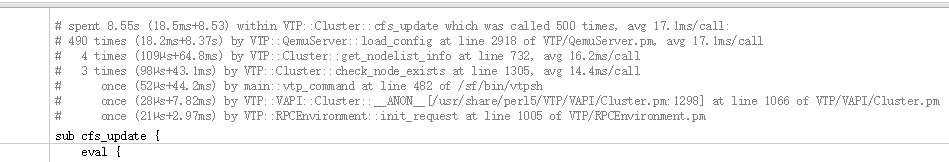
## 优化9：vminfo\_list的cfs\_update优化

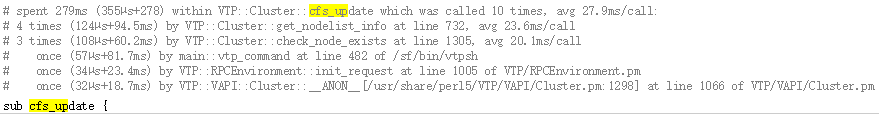
这个函数，调用了500次cfs\_update，我大胆的将490次这个update去掉了，说实话，这不是个很好的解决办法，但是是当前最有效的办法，会有问题吗？

我已收到过恐吓，说是一定要调用cfs\_update。

从原理上来说，一个API刷一次和刷500次的差别，无非是为了保证时间间隔少点，是的，是间隔，但是你就算刷10000次，也可能在你刚update后cfs目录被更新，你读到脏数据啊，老数据啊，反正都有可能是坏数据，你真的很在乎这是1s还是200ms内发生？

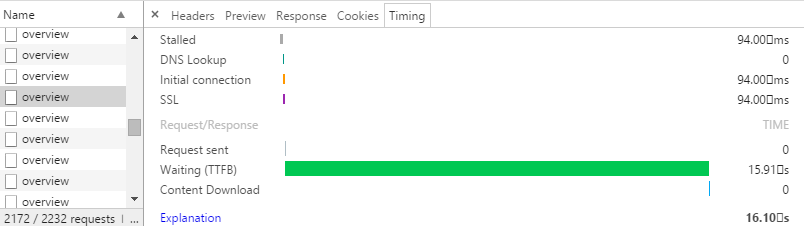
如果真有问题，可能唯一的原因是，请求来的时候一次都不调用cfs\_update，或是别人的路径里面忘记调用，但是函数总入口 init\_request这个地方调用过，如果有问题请找我，不要吓我：

修改后，耗时从8550毫秒降到279毫秒：

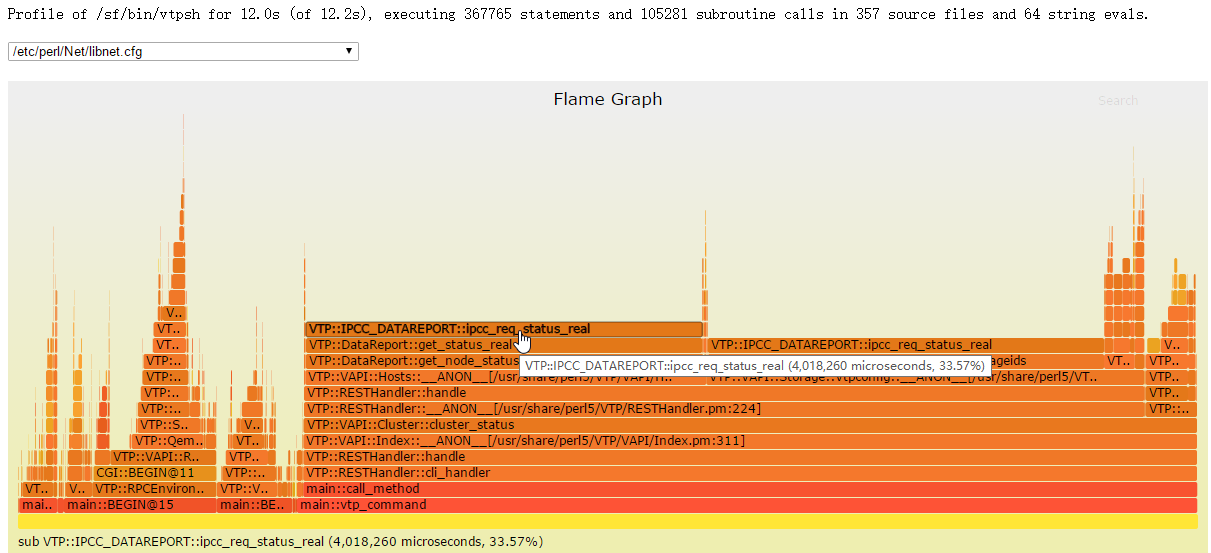


## 待优化1：NFS传输层时延

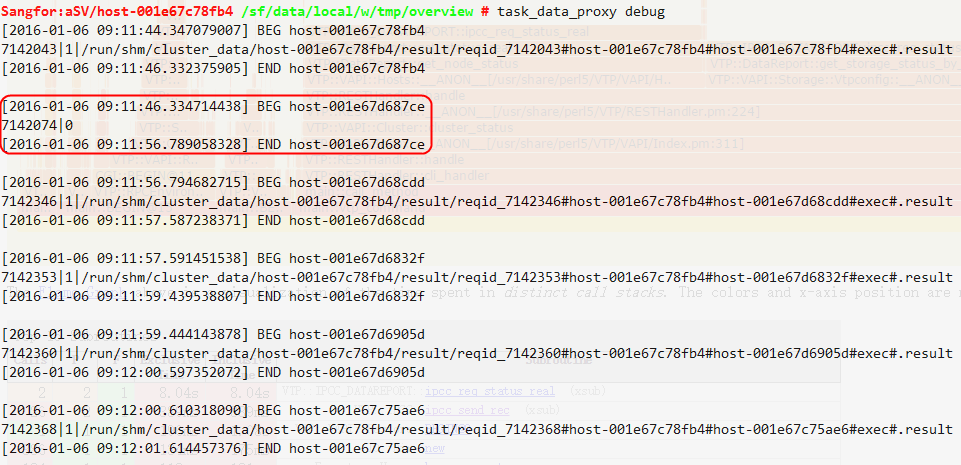
如果从前端看，一个API非常耗时：



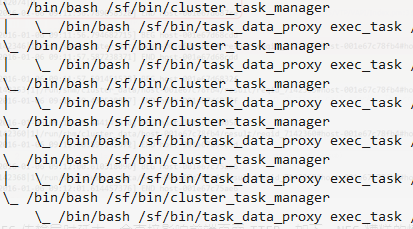
然后从后端看，是卡在了发送消息给datareport



你若细细看过我上面画的流程图，看路径3、4、5、6，事实上并不是datareport自己很忙，而是NFS很忙：



Cluster\_task\_manager很忙：



看看日志，说不定对端的datareport-server也很忙!

不知道是先有任务过多还是先卡，反正卡的时候就任务多，任务多的时候就卡，看看约4387个任务 ：

find /sf/share/nfs\_cluster\_data/ | wc -l



那一定卡出翔。

如果你想自己NFS执行一个最基本任务耗时多少，附上命令一条：

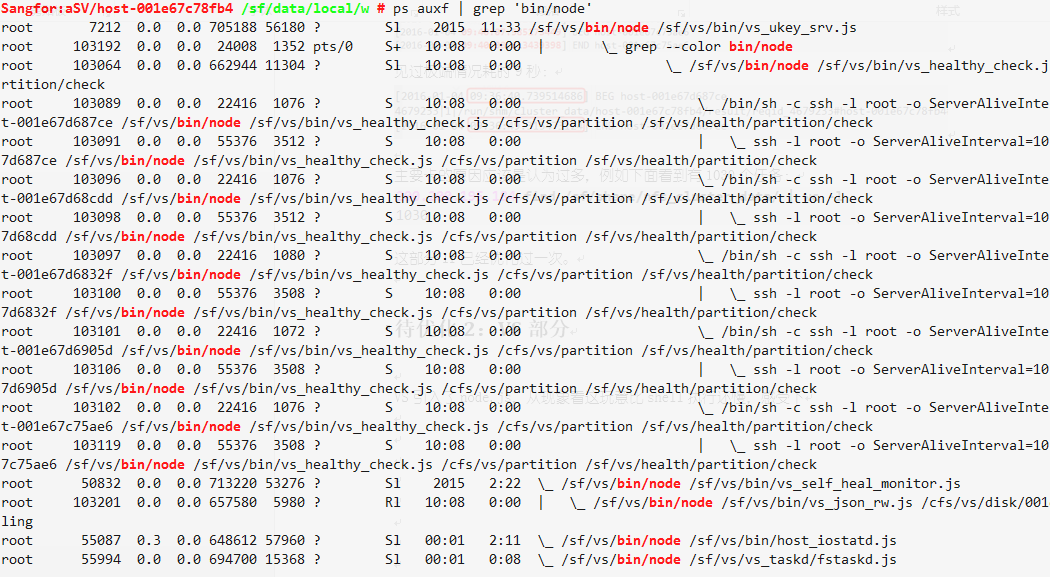
task\_data\_proxy debug | grep -P '\d+:\d+:\d+.\d+' > /tmp/nfstime

awk 'function t2ms(strtime){split(strtime, ta, /[:.]/); return (3600\*ta[1]+60\*ta[2]+ta[3])\*1000 + substr(ta[4], 0, 4)} function extra(str){return t2ms(substr(str,0,length(str)))} { t=extra($2) } {getline; print $4, extra($2)-t}' /tmp/nfstime | sort -n -k2

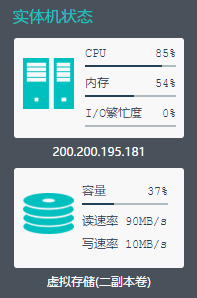
当然，之前说了，这部分我暂时不准备动它，但是它却是瓶颈之一。所以，就算优化了这么多，还是很卡，不要伤心，不要难过，你只要回滚我的优化，发现简直动不了，一对比，就会很开心。

## 待优化2：VS部分

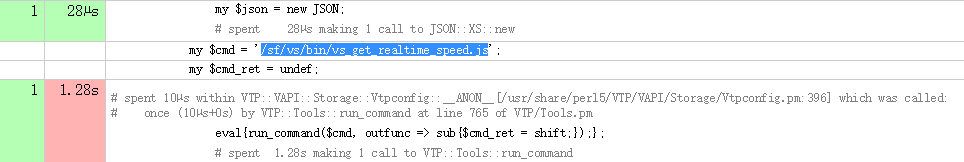
VS引入了node.js，还是紧跟流行节奏的：



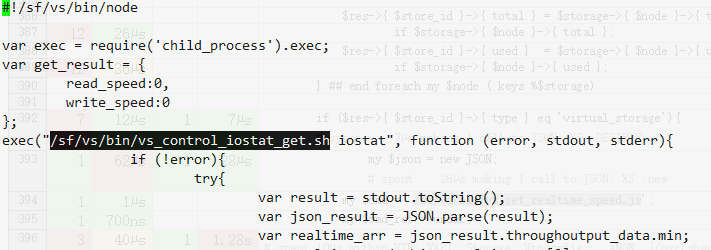
后台执行node.js应该不错，要是前端调用，就成为了瓶颈，看下主页面进入的时候，要获取aSAN的速率，是如何实现的，如下图：



平均一次耗时2s左右，导致主页刷新缓慢，再看下这个图的速率后端，其实现过程为，先调用node.js解析vs\_get\_realtime\_speed.js脚本：

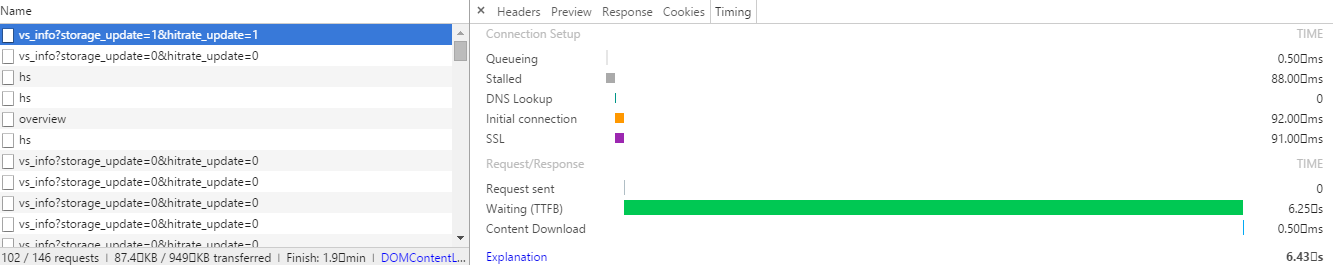


js里面又调用vs\_control\_iostat\_get.sh获取io速率:



这样看，2秒的耗时算是快的。

2秒的问题不大，需要修改的是vs\_info这个API，在没有任务排队的情况下，耗时6秒，导致虚拟存储页面加载慢：

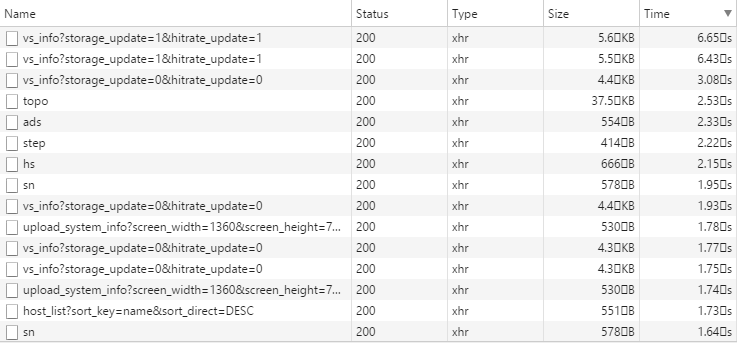


部分地方在多主机的情况下，实现方式让人老眼昏花，不用问有多少请求，48台的集群，实在是，截不完图：

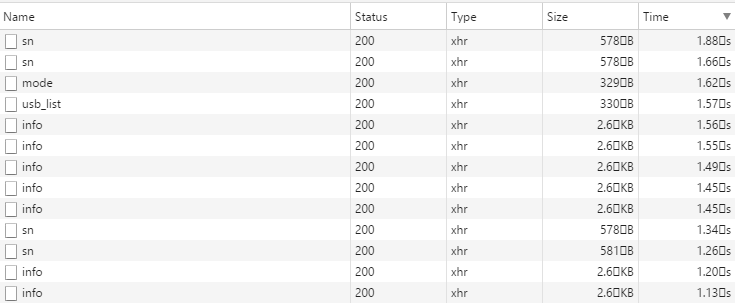


## 优化后

测试中心6台主机情况，一级页面：



二级页面（虚拟机详情页面）：



可能你还是不太明白上面图的意思，其实，耗时最多的vs\_info，sn之类的本来就没有优化，所以按时间排序，出现在了前面，而优化了的overview，get\_vms\_ip\_list等你都看不到，info之所以在1s多，因为cache机制会有timeout，info我是设置的7s一次timeout，timeout后最慢的是1s多，相对于之前的6s是有极大进步，最快的你也看不到。差不多就这个意思，看不到的，表示优化好了。

其实不是程序复杂，而是写的简单，千万不要做for循环程序员，想着想着“一个for循环就搞定了嘛！”

2016/1/4