# 复杂网络大师赛第四名技术分享（篇二：工程技巧的胜利、附代码）

2017-12-29 宾狗 学术 复杂网络

## 0x00 背景

（代码见文末）

在上一篇文章《复杂网络大师赛第四名技术分享（篇一：思路与分析）》中，我们大概介绍了一下这次复杂网络大师赛的基本情况，并简要的分析了官方的评价算法和我自己的思路。其中很重要的一个环节就是利用官方的评价算法来优化节点重要性的序列，但是这个方法有一个硬伤，就是对评价算法的效率要求非常高。而官方提供的`groovy`版本仅仅用来计算鲁棒值还好，如果想在它的基础上进行改造，用于寻找更优的序列就有点不切实际了。

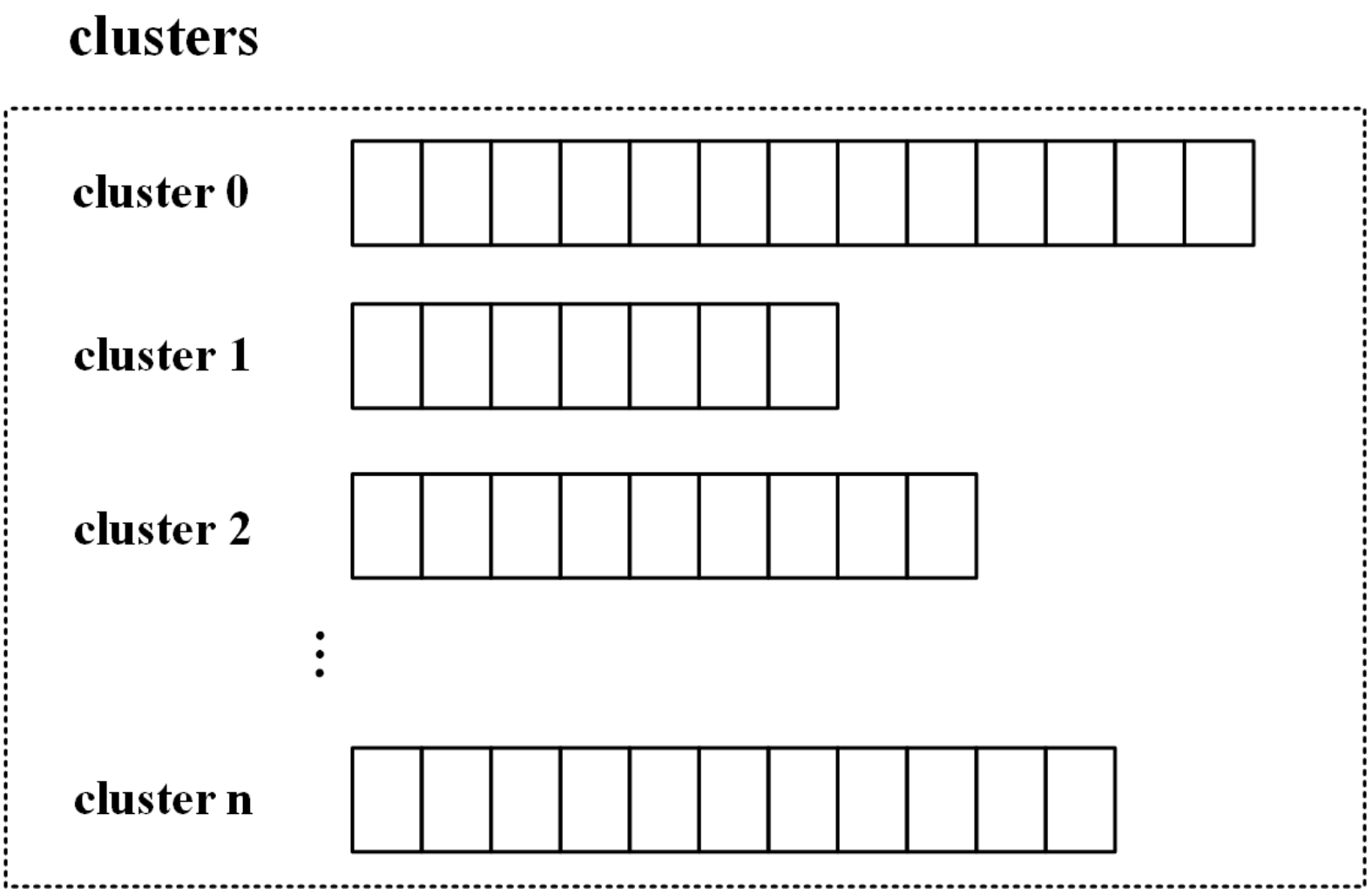
提到高效，相信很多人第一个想到的就是C/C++，的确，单从效率上来看，众多编程语言无出其右者。但是运行效率和开发效率似乎是一对天生的冤家，运行效率上去了，必然会导致在开发和程序优化上耗费大量的时间。C语言虽然是很多人的最先学习的编程语言，却很少人敢拍着胸脯说自己能很好的驾驭它，尤其涉及到一些内存和程序优化相关的问题，如果在不熟悉的情况下硬上，可能最后的运行效率还比不上Java或者Python，“辱没”了C语言的美名。

本文就来讲讲我在用C语言实现官方的评价算法时踩过的坑，以及我是如何思考并一个一个的解决这些问题的。正是依靠这个高效的工程实现，将我的排名从二十多一路刷到第四，所以我厚颜无耻的称之为“工程技巧的胜利”……当然，如果你是资深的C/C++攻城狮，就权当看看故事会吧。

## 0x01 STL的坑？

一开始由于我对纯C还是抱有一丝敬畏之心(其实就是怕麻烦)，而且想当然的认为C++的效率也差不到哪去，所以决定先用C++写一版。

回顾一下上一篇文章对评价算法的描述，我们需要用到clusters来存储所有的集群和集群中的成员。先脑补一下变量clusters理论上的存储结构，如下图所示




注意，图中每一个小的cluster应该都是可动态变化的数组，能够添加指定个数的节点(在cluster合并时需要加入多个节点)，看看官方的`groovy`版本是如何实现的，截取其中的一个代码片段，如下所示

NetMaster.groovy

集团ID

集团成员

节点ID

```groovy
def sumList = new HashMap<Integer,HashSet<Integer>>()//存储所有的节点
def vMap = new HashMap<Integer,Integer>()//存储所有节点的位置
Double sum = 0D //当前最大的连通节点数量
Double sumNum = 0D //所有连通节点的总数量
println("开始建图")

int index_v = 0
produceList.reverseEach {//反向循环建立边集合点
    index_v++
    def temp = map.get(it) as ArrayList<Integer>//得到这个节点所有的边
    if (temp){//占有边

        def vSet = null as HashSet
        def key = null
        boolean flag = false
```

其中，变量`sumList`对应我们算法描述中的clusters，变量`vMap`对应算法描述中的node_clusterid，从变量定义中可以看出来，这里是用了`HashMap`数据结构存储所有集团及其集团成员。其中`Integer`对应了集团ID号，而`HashSet<Integer>`则用于存储该集团中的成员(也就是节点ID号)。

当然，C++里是没有`HashMap`或者`HashSet`这种数据结构的，不过好在C++的STL中倒是有诸如`unordered_map`和`vector`这样的容器。于是乎，我依靠C++的文档，照着官方的`groovy`代码将评价算法实现了一遍，期间踩坑无数。当我好不容易排除所有bug，满怀激动的编译执行时，现实却狠狠的给了我一记闷棍。即使是计算规模最小的40万节点的网络鲁棒值，程序跑到一半竟然就慢如蜗牛，过了5分钟都没有出结果，用C++写的程序效率竟然比不过`Java`？无奈之下，只得强行结束进程查找原因。

经过一番搜索，我大概找出了导致程序运行缓慢的原因，问题集中在`vector`的使用上：

1. 第一个问题是我初始化`vector`的形式，考虑到集团是动态变化的，所以我没有在初始化的时候指定它的大小。而网上很多资料都提到，应该尽量在一开始就给`vector`分配足够的空间，用`vector.reserve()`可以实现。
2. 第二个问题其实与第一个问题相关，就是向`vector`中添加元素的方式，我使用的是`vector.push_back(vi)`，而网上的资料说的非常明确，`push_back`的效率非常低下，原因在于`push_back`会先做一步越界检查，即使`vector`的空间足够。而高效的做法是在确保不越界的情况下，用类似`vector[i]=vi`的方式实现。

也就是说，为了提高效率，我们应该尽量提前给`vector`申请好足够的存储空间，并利用指针索引的方式插入元素。可是问题来了，集团大小都是动态变化的，我们如何提前预知某个集团应该预留多少空间呢？

那么，索性给每个集团都预留足够的空间：将每个`vector`的大小都设置为该网络中所有节点的数量，这样就可以放心的使用指针索引的方式插入和读取元素，而不用担心越界的问题了。

## 0x02 内存不足？

按照刚才的思路，对上一版程序稍作修改之后，兴奋的编译执行，却再次被现实打脸，程序提示内存不足。为什么会这样？我们来做一个简单的计算，以规模为40万节点的网络为例，假设用`int`表示节点ID号，存储一个40万节点大小的集团所需要的空间大约为`4*4*100000`字节，约`1.53MB`左右，如果网络中有1万个度为1的节点(这已经是非常保守的估计了，实际还要比这多)，也就意味着在算法的初期总共要开辟`1.53*100000 MB`的空间，约`14.94GB`，这还仅仅是规模最小的网络的保守估计，对其他200万节点规模的网络，所需的内存量无疑是天文数字。因此，这种粗放暴力的内存申请方式显然是要不得的。

可是这样就陷入了两难的处境，提前分配足够的`vector`空间会导致内存不足。若不提前给`vector`分配好内存空间，就要动态的扩展`vector`大小而严重影响效率。

权衡之下，我决定放弃使用C++的`vector`存储集团成员，转而采用C语言中数组存储，并用`malloc`方式动态申请内存。

## 0x03 诡异的Bug

有的同学可能会问，用`malloc`申请的数组长度不是固定的么，能够满足动态扩展的需求吗？当然是可以的，有两种实现方式：第一种是用`realloc`函数，另一种是重新`malloc`一块更大的内存，然后把原来的数组拷贝过来，再将原数组释放掉即可。

这一次没有了STL的便利，我又费尽周折的用纯C实现了一版评价算法，但是编译执行后竟然报了一个系统级的错误。这另我百思不得其解，因为我在另一个几千节点规模的小型网络中对程序进行了测试，能够得到正确结果，

至少说明这一版程序在逻辑上是没有问题的。在后续的调试过程中，我发现当程序运行到某一阶段时，错误会出现在`malloc`申请内存这一行(如果使用`realloc`同样会有类似的情况)，而函数中的参数均正常。在我的理解中，如果`malloc`申请内存成功，则会返回对应的指针，若申请内存失败，则返回`null`，而现在的情况是直接报系统异常，确实让我无法理解。

考虑到只有在网络规模比较大时才会出现这种情况，我猜测这可能是与短时间内高频次、密集的申请释放内存有关，也许是底层哪里出了bug吧。

经历了N次失败，是时候进行一下反思了，我把目前所遇到的困难和限制进行了梳理：

- 内存限制，这要求我们不能采用粗暴的内存申请方式
- 效率问题，这要求我们在能用C的情况下尽量用C(用C++的`vector`也得用指针索引的方式，还不知直接用C)
- 未知底层bug的困扰，这要求我们尽量少在程序运行过程中频繁申请和释放内存

这看起来完全无解！然而，巧合的时，在某天值班的时候，我突然想起了以前学漏洞利用时，看了很长一段时间的Linux堆内存管理的机制，瞬间找到了突破口。

## 0x04 堆内存管理的启发

Linux的堆内存管理可谓博大精深，过于复杂的理论和实现细节就不涉及了，这里我们只要重点关注其中的fast bin即可。首先需要解释几个概念，在堆内存管理中，我们把内存块称为chunk，已分配给用户的称为allocated chunk，空闲的称为free chunk。所谓堆内存管理，最基本同时也是最重要的工作就是实现高效的分配和回收chunk。内存中大量free chunk该如何记录和索引呢？这就需要用到bin了，bin是一种记录free chunk的链表数据结构。Linux系统针对不同大小的free chunk，将bin分为了4类：

- fast bin
- unsorted bin
- small bin
- large bin

那到底什么是fast bin？看下面这张示意图




Fast Bin Snapshot

其中`main_arena.fastbinsY`对应的就是fast bin，而fast bin下方用链表串起来的块就是free chunk，为了和其他chunk区分开来，我们称之为fast chunk。乍一看，fast bin不就是个指针数组(链表)吗？数组中每一个元素都是一个指针，分别指向不同大小的fast chunk链表的头节点。没错，从本质上看fast bin就是一个简单的指针数组，但可别小看了fast bin，它是所有bin中操作速度最快的，这与它的用途和巧妙设计是分不开的。我认为fast bin的高效源于两个方面：

- 按大小管理fast chunk：从图中可以看出，每个fast bin所管理的fast chunk大小是不同的，第一个为16 bytes，后面依次为32 bytes和64bytes，分别满足不同大小的内存申请需求，也避免了空间的浪费。

- 分配与回收fast chunk采用了LIFO(后进先出)算法：fast chunk虽然是用单链表连接起来的，但其操作方式却更像栈，因为所有操作都是在链表尾进行的。用户释放内存(free)时，空闲出来的fast chunk根据其大小添加到对应的链表尾上；用户申请内存(malloc)时，满足要求的fast chunk从对应的链表尾上卸下，分配给用户使用。这是一个典型的LIFO(后入先出)算法，这样做有什么好处呢？大家以前学习操作系统的时候应该知道内存中的页面调度算法有LRU和LFU等，无论哪种都是希望最近使用过的内存页尽量驻留在物理内存中，以减少缺页中断的触发。而LIFO算法恰到好处的配合了这个特性，用户最近释放的内存(表明最近被使用过，有很大可能直接就在物理内存中)下一次就是最先被分配出去的，因此保证了fast bin在时间效率上也极高。

## 0x05 评价算法的最终实现

前面花了大量篇幅讲解堆内存管理的原理，到底与我们的问题有什么关系呢？我们再回顾一下前面遇到的难题，即要节约空间，又要减少运行时频繁的申请和释放内存，结合上一节说的堆内存管理，我们的问题似乎都可以依靠堆内存管理来解决。是不是可以自己模拟一个算法层面的堆内存管理器呢？姑且称之为SHMM(Simulated Heap Memory Manager)好了。

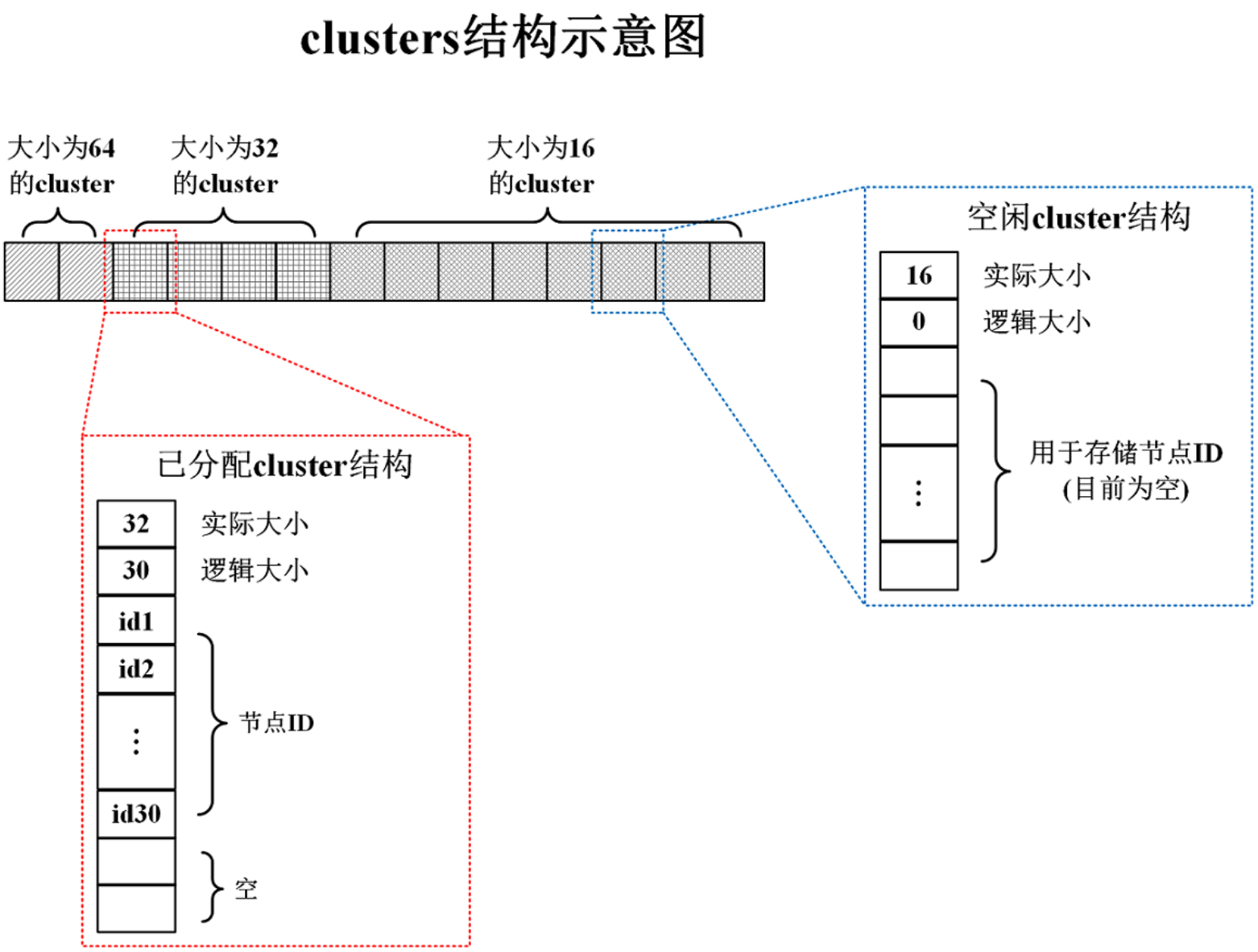
试想一下，在程序开始阶段，我们提前把所需要的内存申请完毕，并按照其大小分门别类管理起来。在程序的运行过程中，所有的内存申请和释放过程由SHMM“接管”。申请内存就不是使用`malloc`了，而是由SHMM从提前申请好的内存中找出一个大小相匹配的空闲块，交由程序使用；释放内存也不再是`free`了，而是由SHMM重新标记为空闲块并管理起来。

还有一个问题，我们怎么知道一开始申请多少内存够呢？这就涉及到合理的设计空闲块的大小和数量了，回顾一下评价算法(以40万节点为例)，假设算法进行到了最后一步，此时仅剩最后两个集团，其中一个集团规模为30万，另一个集团规模为10万，合并以后得到规模为40万集团。这个过程仅需要一个40万大小的空闲块(开始实际存30万个节点，合并后存40万节点)和10万大小的空闲块即可。进一步反推和扩展，不难得出下面这个方案：

- 40万大小空闲块*1≈1.5MB
- 20万大小空闲块*2≈1.5MB
- 10万大小空闲块*4≈1.5MB
- ……
- 98大小空闲块*4096≈1.5MB
- 49大小空闲块*8192≈1.5MB
- ……
- 4大小空闲块*131072≈1.9MB

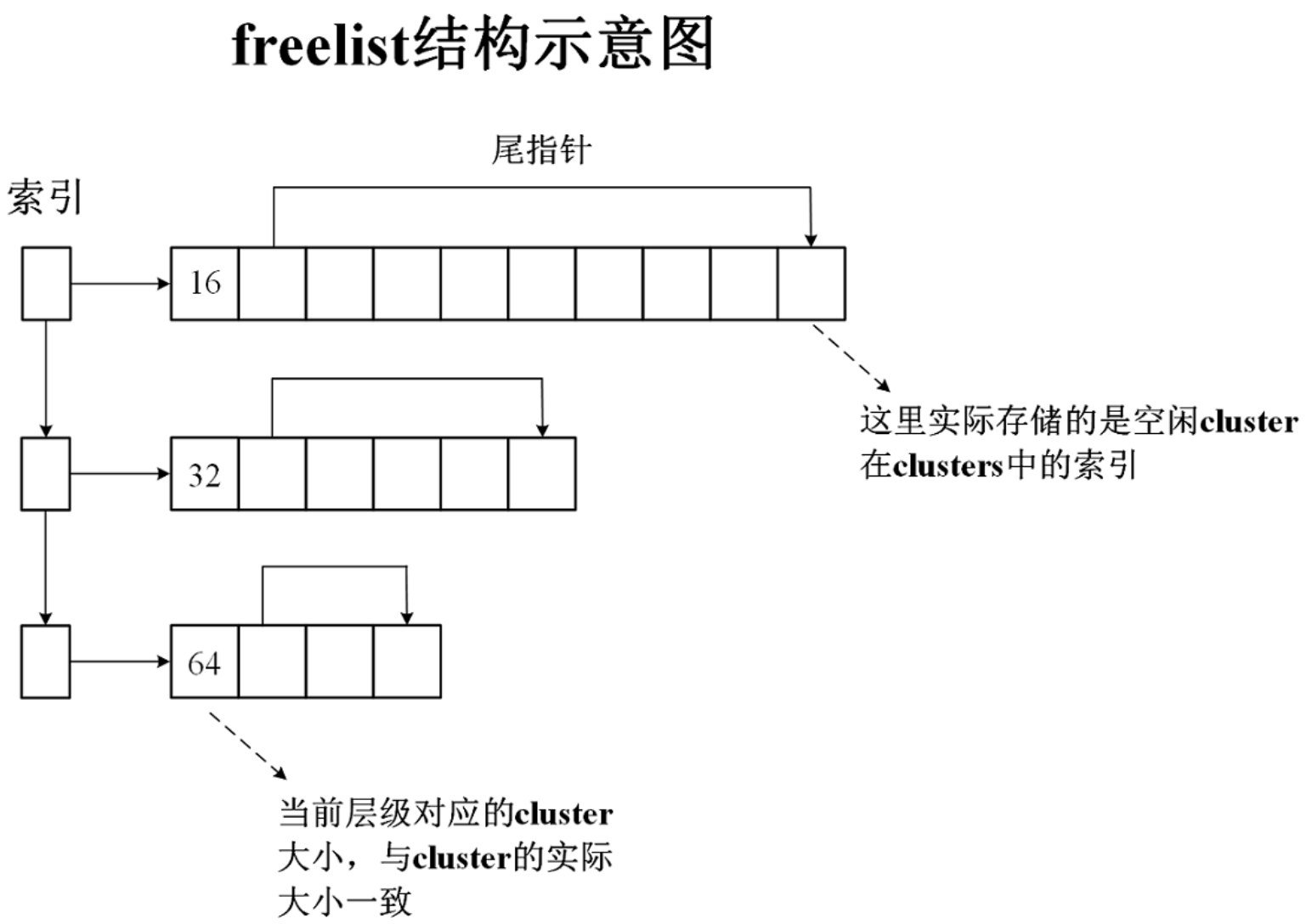
可以看出，以上方案基本上可以满足任意cluster添加节点或是合并的需求，且所需内存空间不超过30MB！

那么具体该怎么实现？直接看我画的示意图：



由于是示意图，我只画了三种大小的空闲块，大家领会意思即可。clusters在这里是作为索引所有cluster(无论是空闲的还是已经分配的cluster)的指针数组，在程序的开始阶段，就需要根据当前网络的规模，按上面的模式一次性申请完所有空闲块内存。其中，每个cluster的结构我也画在了图中，一目了然。

下面就是最重要的部分：模拟实现堆内存管理。这里我模仿fast bin设计了一个freelist，如下图所示




其原理不再赘述，就是把fast bin照搬过来。不过需要指出的是，freelist中实际存储的并非cluster或指向cluster的指针，而是空闲的cluster在clusters上的索引。基于freelist结构，很容易实现cluster的“申请”、“释放”与“合并”函数。

利用算法模拟出来的堆内存管理，完美的避开了毫无头绪的诡异bug，在所有代码全部完成之后，再次编译执行，成功！更为重要的是，在时间效率上有了显著提升，执行速度吊打官方的`groovy`版本。

## 0x06 最后一公里&并行化提速

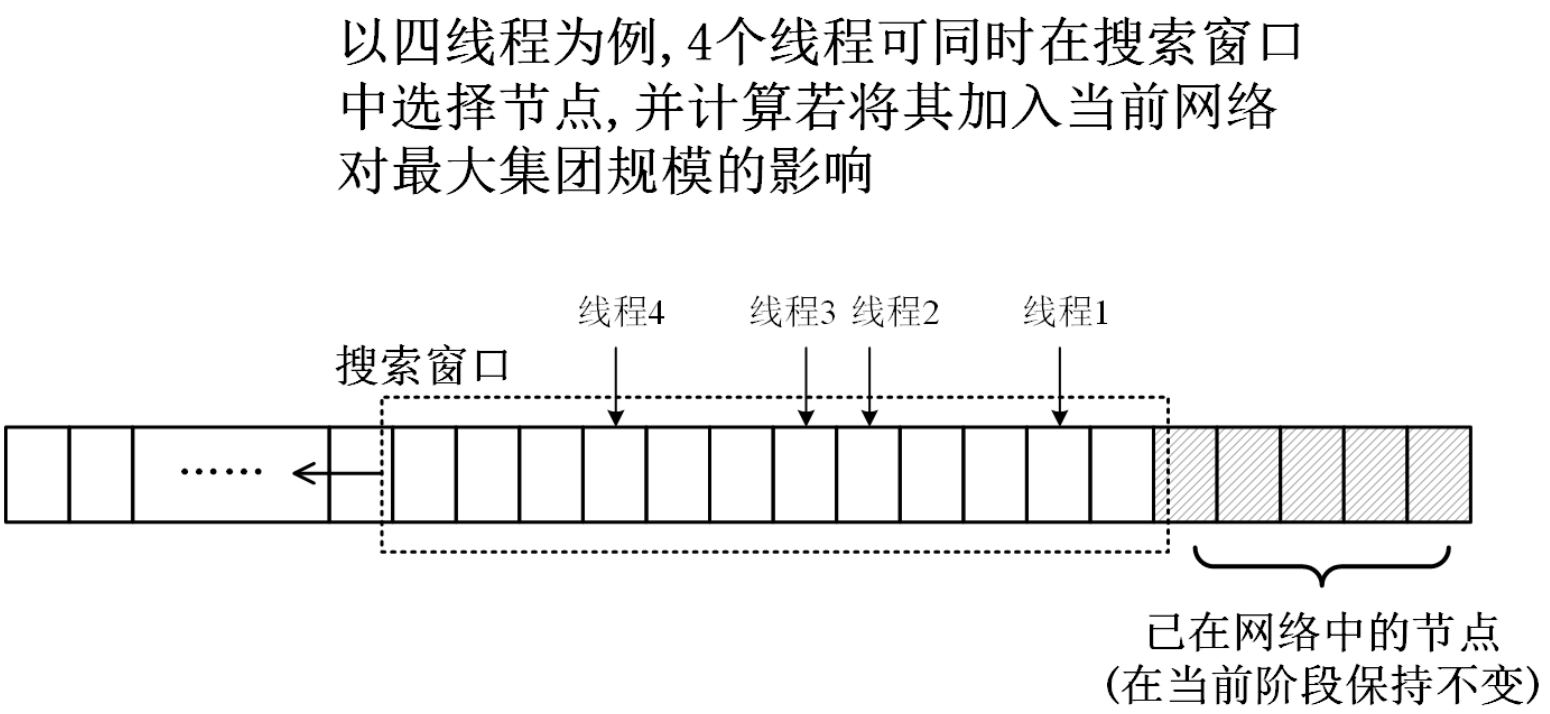
现在我们手里有了C语言版的评价算法，已经迈过了最艰难的一道坎，下面就是考虑如何完成的终极目标了。

在上一篇文章中，我已经把思路讲的很详细了：第一步先用带贪心策略的`PageRank`算法得到一个初始序列，第二部在评价算法的基础上优化这个序列。

在实现贪心策略的`PageRank`算法时，我用到了`igraph`这个库，实现起来也比较简单，这里直接略过。在实现第二步的过程中，需要注意和评价算法的区别：在评价算法中没有搜索窗口，将节点添加到当前网络以及更新当前网络的最大集团规模的操作是同步进行的；而在有搜索窗口的情况下，需要先寻找使当前网络最大集团规模最小的节点，然后再执行添加节点的操作。

至此，我们后续的过程就是不断调整$k$的取值刷结果了，$k$值在几百到几千的范围内时，运行速度非常块。但是随着$k$值的不断增大，运行速度急剧下降。如果打开系统性能监视器可以看到，在我们的多核的机器上，满负荷运行的只有一个核，这是对系统性能的极大浪费！想要进一步提速必须将原来的程序并行化，采用多线程的方式执行，但是并非所有程序都能够简单的并行化。以我们这个问题为例，添加节点的过程有着非常强的前后依赖性，只有在前一个节点加入到网络之后，相关的集团合并等更新操作完成之后才能够添加下一个节点。对于有前后依赖性的程序片段，我们是无法进行并行化改造的。

但是，在搜索窗口中的寻找最优节点的过程就没有这个限制，如下图所示




我们前面说过，寻找最优节点的过程并不修改当前网络结构，而是在找到最优节点后再将其添加到网络中。由于当前网络结构不变，因此每一次寻找并计算的过程都是互相独立的，这就很适合并行化改造。当然，这里的并行化改造并不要求大家有丰富的并行编程经验，也不需要使用`CUDA`之类的牛刀。我们可用简单方便的`OpenMP`库，只需要添加几行编译器指令就能够轻松实现并行化。关于`OpenMP`的使用，大家可以参考网上的诸多教程。在具体的使用过程中，要尽量避免数据依赖和竞争，设置好临界区。

再次编译执行可以发现，此时所有的核都处于满载运行状态，运行速度有了显著提升。

## 0x07 实现代码&小结

代码已上传到我的github，欢迎大家拍砖~

剩下的就是找最优的$k$值了，这个过程就是手工尝试了……依靠着诸多高效的改进和坚持不懈的努力，我最终刷到了第四名。当然还有继续上升的空间，不过限于时间没有做更多尝试了。

自主提交阶段 ∨　　综合排行榜 ∨

| 排名 | 排名变化 | 团队logo | 队名 | 最高得分 | 提交次数 | 最后提交时间 |
|---|---|---|---|---|---|---|
| 1 | - | | wwa8756d313 | 0.94082 | 5 | 2017-10-04 15:11 |
| 2 | - | | wwa8757a05a1 | 0.94445 | 2 | 2017-09-29 14:52 |
| 3 | - | | 格式不正确 | 0.98778 | 31 | 2017-10-09 20:45 |
| 4 | ↑ 11 | | 月亮代表我的心 | 0.98808 | 6 | 2017-10-15 23:59 |
| 5 | ↑ 4 | | cauc_qsm | 0.98828 | 10 | 2017-10-15 23:26 |
| 6 | ↓ 2 | | 技术ob | 0.99286 | 8 | 2017-09-15 10:11 |
| 7 | ↓ 2 | | zhfkt | 0.99286 | 19 | 2017-09-08 15:55 |
| 8 | ↓ 2 | | DETA | 0.99827 | 3 | 2017-07-21 11:15 |
| 9 | ↓ 2 | | 瓜皮和他的小伙伴 | 1.00716 | 2 | 2017-09-14 10:01 |
| 10 | ↓ 2 | | 我爱蒋孟灵 | 1.00898 | 7 | 2017-10-15 23:48 |

仔细思考起来，在很多细节上还有可以进一步优化的地方，例如：

- 设置一个搜索窗口起点：在构建网络的初期，几乎所有节点的度都是1，都是自成一个集团，这个时候是没有必要在搜索窗口中找最优点的。
- 搜索过程的提前终止：当搜索窗口找到一个节点，使当前集团规模无变化时(加入了其他小集团)，可以提前结束搜索过程。

以上可以在真正的实践中慢慢优化了。

从工程化实现的过程来看，我们经历了一段非常崎岖的路程，事实上任何大型项目的实现过程都无法避开高效、可拓展等坑，这也是一段必由之路。

那么如果你对我的参赛感悟和一些其他闲扯感兴趣的话，欢迎关注本系列的第三篇文章~

## 参考资料

### C++的效率相关

- Why push_back is slower than operator[] for an previous allocated vector
- Do not waste time with STL vectors
- Fast iteration over STL vector elements
- 6 Tips to supercharge C++11 vector performance

### Linux堆内存管理相关

- Understanding glibc malloc
- Linux堆内存管理深入分析(上半部)
- Linux堆内存管理深入分析(下半部)

---