**谨慎使用STL**

[](http://www.jianshu.com/u/62c6f75ec9d5)

作者 [lxfeng](http://www.jianshu.com/u/62c6f75ec9d5) 关注

2016.08.15 21:56\* 字数 1651 阅读 110评论 0喜欢 1

**谨慎使用STL**

最近解决了一个因为大量使用STL造成的严重内存泄漏问题，再次记录下。

上周上线了基于用户tag的推荐，没天服务器会自动生成新的tag数据，然后scp到指定目录下，推荐服务中，对文件做了监控，如果改变就会重新加载解析。

函数代码如下：

void ReposManager::ReLoadUidTagData(const std::string& file\_name) {

std::string file\_content;

bool ret = file::ReadFileToString(file\_name, &file\_content);

std::unordered\_map<int, std::unordered\_map<std::string, double> > uid\_tags\_map;

if (ret == false) return;

std::vector<std::string> content;

SplitString(file\_content, '\n', &content);

ifstream fin(file\_name);

for (auto line: content)

std::vector<std::string> strs;

std::vector<std::string> vstr;

SplitString(line, '=', &strs);

if (strs.size() < 2) continue;

double weight = StringToDouble(strs[1]);

SplitString(strs[0], ':', &vstr);

if (vstr.size() < 2) continue;

int uid = StringToInt(vstr[0]);

std::string tag = vstr[1];

std::unordered\_map<std::string, double>& tags\_map = uid\_tags\_map[uid];

tags\_map[tag] = weight;

}

for (auto it = uid\_tags\_map.begin(); it != uid\_tags\_map.end(); ++it) {

int uid = it->first;

std::unordered\_map<std::string, double>& uid\_tags = uid\_tags\_map[uid];

std::vector<std::string> tags;

for (auto it = uid\_tags.begin(); it != uid\_tags.end(); ++it) {

tags.push\_back(it->first);

}

auto cmp = [&](const std::string& a, const std::string& b) {

return uid\_tags[a] > uid\_tags[b];

};

std::sort(tags.begin(), tags.end(), cmp);

uid\_tags\_[uid] = tags;

}

}

文件存储格式为uid:tag=weight 有很多行，共220M左右，函数的功能是对每行进行分割，然后存到成员变量uid\_tags\_中。函数初看之下没什么问题，但上线后第二天内存涨到3.6G，线上服务器内存很大也不能这么浪费，我记得刚启动加载完毕才2.0G，怎么涨了这么多，考虑原因，估计是凌晨的文件变更再次解析导致的，重启改动文件测试，果然是这个。

文件220M，全部加载进内存，解析过程中算上额外的数据结构的确会引起内存增长，但这些内存消耗只是暂时的，局部变量函数结束后，内存就释放了，显然这里的一些局部变量没有释放掉，这个情况还的确没遇到过。

google了解到STL的内存分配器，分多级，对于申请交大的内存，他会在堆上去申请。我们理解的局部变量出了作用域就会释放掉，这是因为大部分的局部变量都在函数栈里，函数执行完了，整个栈都会释放掉。周末在家试了下，尝试手动通过vector的swap以及map的clear，手动释放，在本机有一定的效果，内存没有继续增长，以为解决了。

周一到公司，放到测试服务器上，发现还是没有解决了，内存还是会出现较大增长，同样的二进制文件，为何会有差异，考虑再三，估计是虚拟机和测试服的配置不同，虚拟机内存不大，所以内存释放得会快点，测试服内存十几个G，内存不是很紧张，所以长时间得不到释放。(看来保持测试环境和线上环境的统一，这个还是很有必要的，保证了运行环境的一致，便于排查问题)。

在知乎看到有人回答malloc\_trim（0），google了到使用malloc\_trim(0),后来发现无效，然后看C语言的malloc只是提供了接口，具体的实现在各个平台不一样，linux下是glibc的里的malloc，了解到可以用google的tcmalloc替换掉linux默认的malloc，替换后内存从原来高达3.6G到2.6G，看来效果还是杠杆的! 回头自己对代码做了下检查，觉得存储的文件可以再修改下，

uid:tag1=weight

uid:tag2=weight

...

改为：

uid:tag1=weight,tag2=weight,tag3=weight....

将每个uid的所属tag整合到一起，原来220M的文件变味了150M左右，然后修改文件解析函数：

void ReposManager::ReLoadUidTagData(const std::string& file\_name) {

std::string file\_content;

bool ret = file::ReadFileToString(file\_name, &file\_content);

if (ret == false) return;

std::vector<std::pair<std::string, double>> tag\_list;

using namespace std;

auto cmp = [](const pair<string, double>& a, const pair<string, double>& b) {

return a.second > b.second;

};

std::vector<std::string> content;

SplitString(file\_content, '\n', &content);

for (auto line: content) {

if (line.length() == 0) continue;

std::vector<std::string> strs;

SplitString(line, ':', &strs);

if (strs.size() < 2) continue;

int uid = StringToInt(strs[0]);

std::vector<std::string> tags;

SplitString(strs[1], ',', &tags);

tag\_list.clear();

for (int i = 0; i < tags.size(); i++) {

std::vector<std::string> tmp;

SplitString(tags[i], '=', &tmp);

if (tmp.size() < 2) continue;

std::string tag = tmp[0];

double weight = StringToDouble(tmp[1]);

tag\_list.push\_back(std::pair<string, double>(tag, weight));

}

std::sort(tag\_list.begin(), tag\_list.end(), cmp);

uid\_tags\_[uid].clear();

for (int i = 0; i < tag\_list.size(); ++i) {

uid\_tags\_[uid].push\_back(tag\_list[i].first);

}

}

}

相比于修改前的，减少了一个 STL局部变量。

std::unordered\_map<int, std::unordered\_map<std::string, double> > uid\_tags\_map;

修改完毕，放到测试服务器测试，内存占用稳定到1.5G左右，多次加载不会出现增长，问题搞定，上线部署。

代码都会写，如何写出高性能，高质量的代码，这就是个技术活。

总结：

1. malloc()/free()作为C标准，ANSI C并没有指定它们具体应该如何实现。各个平台上（windows, mac, linux等等），调用这两个函数时，实现不一样。
2. 在linux下，malloc()/free()的实现是由**glibc库**负责的。STL的内存释放，有时候并没有直接返还给os，只是返还给了分配器。
3. 针对大量数据，谨慎大量使用STL局部变量，虽然栈上分配的，但它维护的队列是分配在heap上的，它操作的内存不一定能够即使释放，可能产生碎片。