开源工具链

目录

- 背景
  - 静态检测
  - 动态检测
- 常用内存泄露检测工具总结
  - 一、valgrind
  - 二、mtrace
  - 三、splint
  - 四、Gperftools heap-profiler
  - 五、Visual Leak Detector
  - 六、Address Sanitizer（ASan）
  - 七、CRT:_CrtCheckMemory
  - 八、BoundsChecker
- 不同内存检测工具对比情况
  - 二、mtrace
  - 三、splint
  - 四、Gperftools heap-profiler
  - 五、Visual Leak Detector
  - 六、Address Sanitizer（ASan）
  - 七、CRT:_CrtCheckMemory
  - 八、BoundsChecker
- 不同内存检测工具对比情况

背景

C、C ++ 因其灵活性、高效性等特点一直以来都是主流程序设计语言之一。它们与Java 等高级语言相比，在编程中程序员需要自己管理内存，并对程序中所涉及的内存操作有很清晰的认识。内存泄露最容易发生的地方：即两个部分的接口部分，一个函数申请内存，一个函数释放内存。并且这些函数由不同的人开发、使用，这样造成内存泄露的可能性就比较大了。内存问题很难让人察觉，特别是内存泄漏，它不同于其他内存错误，如多次释放、野指针、或者是数组越界等容易暴露出来，内存泄漏错误一般隐藏得比较深，在数万行的代码之中寻找内存泄漏无异于大海捞针。因此借助内存分析工具来检验内存错误是非常具有实用价值的，不仅能够提高软件的质量，还能缩短软件的开发周期。

目前国内外已经有一些内存检测工具，如PC-Lint、Polyspace 属于静态检测工具，memwatch、dmalloc、Valgrind、Fense、DDMEM 属于动态检测工具，但它们都有其相应的缺点，比如memwatch、dmalloc、DDMEM、Fense 需要修改源程序，Valgrind上的工具Memcheck 只能运行在Linux 平台，同时开销也比较高，影响程序的运行效率。

静态检测

所谓静态检测，就是不运行程序，在程序的编译阶段进行检测，主要原理就是对 new 与 delete， malloc 与 free 进行匹配检测，基本上能检测出 大部分 coding 中因为粗心导致的问题。

常用的静态检测的工具有 splint, PC-LINT, BEAM，Cppcheck 等，每种检测各有千秋，具体使用方法可以直接去看官方说明文档。但是静态检测不能判定跨线程的内存申请与释放。这时候就需要动态检测出场了。

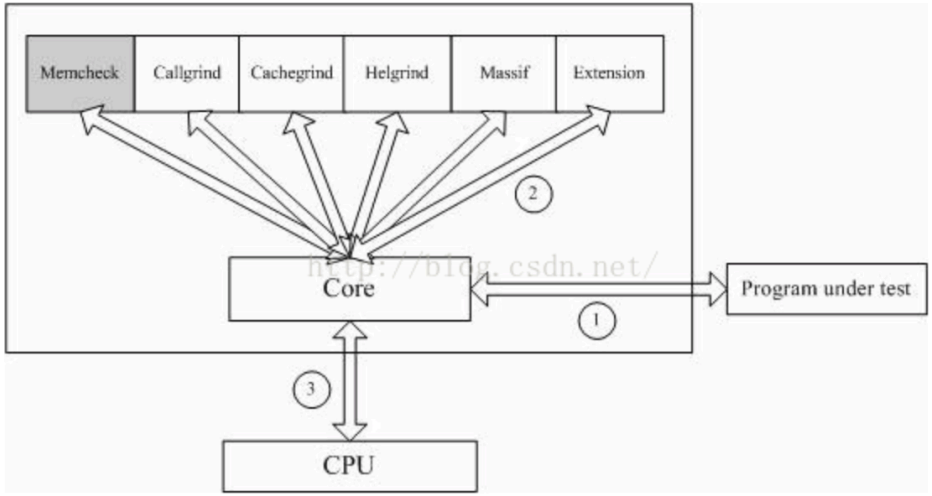
动态检测

所谓动态检测，就是运行程序的过程中，对程序的内存分配情况进行记录并判定。常用的工具有 valgrind, Rational purify 等，每种工具有各自的特点，需要看情况自行选用。对于动态检测来说，最大的弊端就是会加重程序的负担，对于一些大型工程，涉及到多个动态库，带来的负担太重，这时候就需要自己根据需求写一套了。

常用内存泄露检测工具总结

一、valgrind

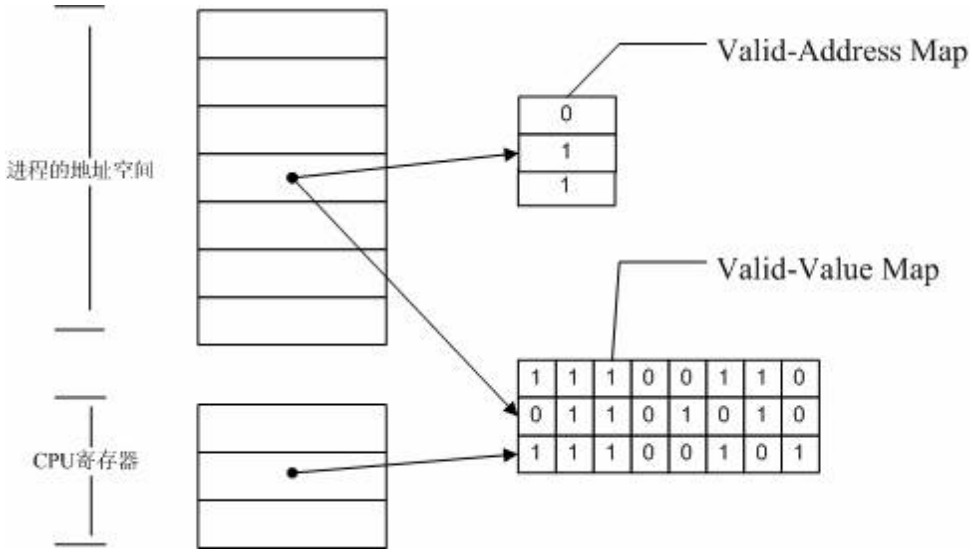
Valgrind是一套Linux下，开放源代码（GPL V2）的仿真调试工具的集合。Valgrind由内核（core）以及基于内核的其他调试工具组成。内核类似于一个框架（framework），它模拟了一个CPU环境，并提供服务给其他工具；而其他工具则类似于插件 (plug-in)，利用内核提供的服务完成各种特定的内存调试任务。Valgrind的体系结构如下图所示：



Memcheck。这是valgrind应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。可以解决：

- 使用未初始化的内存 (Use of uninitialised memory)
- 使用已经释放了的内存 (Reading/writing memory after it has been free’d)
- 使用超过 malloc分配的内存空间(Reading/writing off the end of malloc’d blocks)
- 对堆栈的非法访问 (Reading/writing inappropriate areas on the stack)
- 申请的空间是否有释放 (Memory leaks – where pointers to malloc’d blocks are lost forever)
- malloc/free/new/delete申请和释放内存的匹配(Mismatched use of malloc/new/new [] vs free/delete/delete [])
- src和dst的重叠(Overlapping src and dst pointers in memcpy() and related functions)

Memcheck 能够检测出内存问题，关键在于其建立了两个全局表。



Valid-Value 表： 对于进程的整个地址空间中的每一个字节(byte)，都有与之对应的8 个bits；对于CPU 的每个寄存器，也有一个与之对应的bit 向量。这些 bits 负责记录该字节或者寄存器值是否具有有效的、已初始化的值。 Valid-Address 表： 对于进程整个地址空间中的每一个字节(byte)，还有与之对应的1个bit，负责记录该地址是否能够被读写。

检测原理：

当要读写内存中某个字节时，首先检查这个字节对应的A bit。如果该A bit显示该位置是无效位置，memcheck 则报告读写错误。 内核（core）类似于一个虚拟的CPU 环境，这样当内存中的某个字节被加载到真实的CPU 中时，该字节对应的V bit 也被加载到虚拟的CPU 环境中。一旦寄存器中的值，被用来产生内存地址，或者该值能够影响程序输出，则memcheck 会检查对应的V bits，如果该值尚未初始化，则会报告使用未初始化内存错误。

二、mtrace

mtrace 工具的主要思路是在调用内存分配和释放的函数中装载 “钩子（hook）” 函数，通过 “钩子（hook）” 函数打印的日志来帮助我们分析对内存的使用是否存在问题。对该工具的使用包括两部分内容，一个是要修改源码，装载 hook 函数，另一个是通过运行修改后的程序，生成特殊的 log 文件，然后利用 mtrace 工具分析日志，判断是否存在内存泄露以及定位可能发生内存泄露的代码位置。

首先需要改动一下我们的源码。添加以下两个辅助函数：

```
</>                                                                    Java
```

```
#include <mcheck.h>

void mtrace(void);
void muntrace(void);
```

其中 `mtrace()` 用于开启内存分配跟踪， `muntrace()` 用于取消内存分配跟踪。具体的做法是 `mtrace()` 函数中会为那些和动态内存分配有关的函数（譬如 malloc()、realloc()、memalign() 以及 free()）安装 “钩子（hook）” 函数，这些 hook 函数会为我们记录所有有关内存分配和释放的跟踪信息，而 `muntrace()` 则会卸载相应的 hook 函数。基于这些 hook 函数生成的调试跟踪信息，就可以分析是否存在 “内存泄露” 这类问题了。

三、splint

是一个GNU免费授权的 Lint程序，是一个检查C/C++程序安全弱点和编写错误的程序。静态程序分析往往作为一个多人参与的项目中代码审查过程的一个阶段，因编写完一部分代码之后就可以进行静态分析，分析过程不需要执行整个程序，这有助于在项目早期发现以下问题：Splint会进行多种常规检查，包括未使用的变量，类型不一致，使用未定义变量，无法执行的代码，忽略返回值，执行路径未返回，无限循环等错误。

静态分析工具相比编译器，对代码进行了更加严格的检查，像数组越界访问、内存泄漏、使用不当的类型转换等问题，都可以通过静态分析工具检查出来，我们甚至可以在分析工具的分析标准里定义代码的编写规范，在检测到不符合编写规范的代码时抛出告警，这些功能都是编译器没有的。但是静态检测只是比编译器更加严谨的一点，并不能作为判断内存泄露的依据。所以至今，静态内存泄露检测只是作为一种辅助工具，只能发现一些容易被忽略的问题，并不能解决根本问题。所以还是推荐使用动态内存检测。

四、Gperftools heap-profiler

gperftools 是 google 开源的一个工具集，包含了 tcmalloc，heap profiler，heap checker，cpu profiler 等等。

gperf tools需要替换libc的malloc库，替换为tcmalloc：thread cache malloc，通过在tcmalloc加打桩，即可定位函数级别的内存的累积量

Release 版本的 C++ 程序可执行文件在编译时全部都链接了 gperftools。在 gperftools 的 heap profiler 中，提供了 HeapProfilerStart 和 HeapProfilerStop 的接口，使得我们可以在运行时启动和停止 heap profiler。同时，每个程序都暴露了 RPC 接口，用于接收控制命令和调试命令。在调试命令中，我们就增加了调用 HeapProfilerStart 和 HeapProfilerStop 的命令。由于链接了 tcmalloc，所以 tcmalloc 可以获取所有内存分配和回收的信息。当 heap profiler 启动后，就会定期的将程序内存分配和回收的行为 dump 到一个临时文件中。

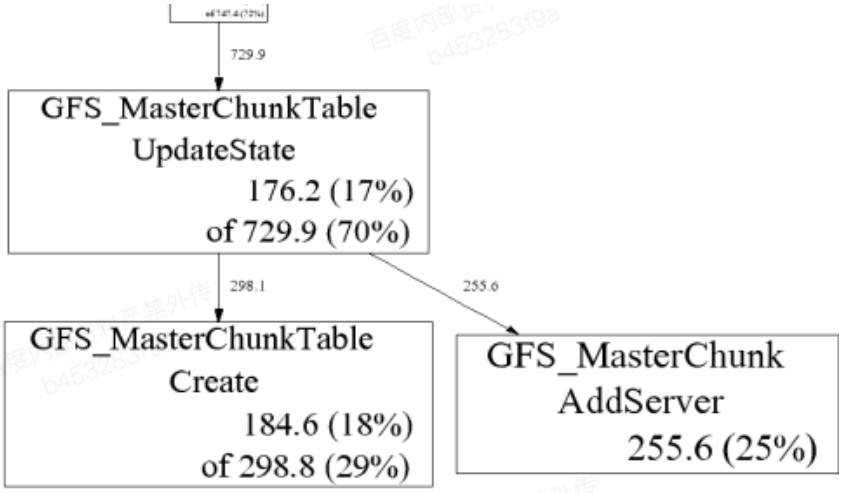
当程序运行一段时间后，将得到一组 heap profile 文件

```Java
profile.0001.heap
  profile.0002.heap
  ...
  profile.0100.heap
```

每个 profile 文件中都包含了一段时间内，程序中内存分配和回收的记录。如果想要找到内存泄露的线索，可以通过使用

```Java
pprof --base=profile.0001.heap /usr/bin/xxx profile.0100.heap --text
```

来进行查看，也可以生成 pdf 文件，会更直观一些。这样一来，就可以很方便的对线上程序的内存泄露进行 Debug 了。



五、Visual Leak Detector

Visual Leak Detector是一款免费的、健全的、开源的Visual C++内存泄露检测系统。相比Visual C++自带的内存检测机制，Visual Leak Detector可以显示导致内存泄露的完整内存分配调用堆栈。 主页地址： http://vld.codeplex.com/ 旧版地址： http://www.codeproject.com/Articles/9815/Visual-Leak-Detector-Enhanced-Memory-Leak-Detectio 下载Visual Leak Detector，当前版本2.2.3，在Visual C++ IDE的"工具"→"选项"→"项目和解决方案"→"VC++ 目录"，"包含文件"增加VLD的"\include"路径，"库文件"增加VLD的"\lib\Win32"路径，另外动态库"\bin\Win32"路径在安装时已经添加到环境变量里面了，若是未添加，则需要手动拷贝"\bin\Win32"下的文件到工程Debug目录。下面记录下使用方法： 1.新建一个Win32控制台项目； 2.添加代码如下所示：

```java
#include "stdafx.h"
#include "vld.h"

int _tmain(int argc, _TCHAR* argv[])
{
    char *pBuf = new char[200];
    return 0;
}
```

3.在Debug模式下的“输出”窗口，将有如下信息：

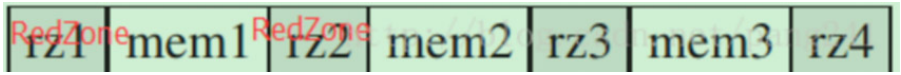
图片上传失败，请重新上传

报告列出了内存泄露是在第几块，所在的地址，泄露的字节，调用的堆栈，内存内容。双击调用堆栈可以跳转到所在行。 4.在Release模式下，不会链接Visual Leak Detector。 5.Visual Leak Detector有一些配置项，可以设置内存泄露报告的保存地（文件、调试器），拷贝"\Visual Leak Detector"路径下的vld.ini文件到工程的Debug目录下（在IDE运行的话，则需要拷贝到工程目录下），修改以下项： ReportFile = .\memory_leak_report.txt ReportTo = both

六、Address Sanitizer（ASan）

AddressSanitizer（ASan）是一个快速的内存错误检测工具。它非常快，只拖慢程序两倍左右。它包括一个编译器instrumentation模块和一个提供malloc()/free()替代项的运行时库。从gcc 4.8开始，AddressSanitizer成为gcc的一部分。

AddressSanitizer主要包括两部分：插桩(Instrumentation)和动态运行库(Run-time library)。插桩主要是针对在llvm编译器级别对访问内存的操作(store，load，alloca等)，将它们进行处理。动态运行库主要提供一些运行时的复杂的功能(比如poison/unpoison shadow memory)以及将malloc,free等系统调用函数hook住。该算法的思路是：如果想防住Buffer Overflow漏洞，只需要在每块内存区域右端（或两端，能防overflow和underflow）加一块区域（RedZone），使RedZone的区域的影子内存（Shadow Memory)设置为不可写即可。



内存映射

AddressSanitizer保护的主要原理是对程序中的虚拟内存提供粗粒度的影子内存(每8个字节的内存对应一个字节的影子内存)，为了减少overhead，采用了直接内存映射策略，所采用的具体策略如下：Shadow=(Mem >> 3) + offset。每8个字节的内存对应一个字节的影子内存，影子内存中每个字节存取一个数字k,如果k=0，则表示该影子内存对应的8个字节的内存都能访问，如果0<k<7,表示前k个字节可以访问，如果k为负数，不同的数字表示不同的错误（e.g. Stack buffer overflow, Heap buffer overflow)。

插桩

为了防止buffer overflow，需要将原来分配的内存两边分配额外的内存Redzone，并将这两边的内存加锁，设为不能访问状态，这样可以有效的防止buffer overflow(但不能杜绝buffer overflow)。

插桩后的代码：

在动态运行库中将malloc/free函数进行了替换。在malloc函数中额外的分配了Redzone区域的内存，将与Redzone区域对应的影子内存加锁，主要的内存区域对应的影子内存不加锁。 free函数将所有分配的内存区域加锁，并放到了隔离区域的队列中(保证在一定的时间内不会再被malloc函数分配)，可检测Use after free类的问题。

（不是特别懂。。）

七、CRT:_CrtCheckMemory

在使用Debug版的malloc分配内存时，malloc会在内存块的头中记录分配该内存的文件名及行号。当程序退出时CRT会在main()函数返回之后做一些清理工作，这个时候来检查调试堆内存，如果仍然有内存没有被释放，则一定是存在内存泄漏。从这些没有被释放的内存块的头中，就可以获得文件名及行号。

八、BoundsChecker

NuMega是一个动态测试工具，主要应用于白盒测试。该工具的特点是学习简单、使用方便、功能有效。NuMega共有三个独立的子功能——BoundsChecker、TrueCoverage、TrueTime。BoundsChecker为代码检错工具，TrueCoverage为测试覆盖率统计工具，TrueTime为程序运行性能测试工具。

BoundsChecker采用一种被称为 Code Injection的技术，来截获对分配内存和释放内存的函数的调用。简单地说，当你的程序开始运行时，BoundsChecker的DLL被自动载入进 程的地址空间（这可以通过system-level的Hook实现），然后它会修改进程中对内存分配和释放的函数调用，让这些调用首先转入它的代码，然后 再执行原来的代码。BoundsChecker在做这些动作的时，无须修改被调试程序的源代码或工程配置文件，这使得使用它非常的简便、直接。

不同内存检测工具对比情况

检测工具	分类	描述	性能影响	是否会修改源代码	能否指出具体位置和原因	应用场景	使用	备注
valgrind	动态检测	一个强大开源的程序检测工具	极大	否	能	适用Linux、Mac(Mac10.14后就不支持了)	valgrind --tool=memcheck ./a.out	安装下载：https://www.valgrinc 详细使用：https://blog.csdn.ne 原理介绍：https://blog.csdn.ne
mtrace	动态检测	GNU扩展, 用来跟踪malloc, mtrace为内存分配函数（malloc, realloc, memalign, free）安装hook函数	大	是	能	适用Linux	无需安装，使用时包含头文件mcheck.h，程序中调用mtrace和muntrace方法即可。 mtrace 用于开启内存使用记录，muntrace用于取消内存使用记录。内存使用情况记录	mtrace这个工具本身 详细使用：https://www.jianshu. 原理介绍：https://zhuanlan.zhi

							到一个文件，值由环境变量：MALLOC_TRACE决定。	
3	splint	静态检测	一个针对C语言的开源程序静态分析工具	无	否	能	Linux、Windows	需要下载源码并安装程序，通过标志和注释来获取想要的错误信息 splint *.c（Linux）	安装下载： http://splint.org/dow 详细使用： https://www.cnblogs .html
4	Gperftools heap-profiler	动态检测	是 google 开源的一个工具集	有消耗，较低	是	不能，需要自己分析输出的内存变化情况	适用Linux	需要替换libc的malloc库，替换为tcmalloc：thread cache malloc，通过在tcmalloc加打桩，即可定位函数级别的内存的累积量，需要自行分析内存变化情况找到内存泄露位置。	安装与详细使用： https://www.cnblogs
5	Visual Leak Detector	动态检测	免费的、开源的、强大的内存泄露检测系统，可以安装当作VS的一个插件。相比Visual C++自带的内存检测机制，Visual Leak Detector可以显示导致内存泄露的完整内存分配调用堆栈。	无	否	能指出位置，不能给原因	Visual C++ IDE上的工具	在安装完成后，在工程中指定其include和lib，然后添加头文件#include <vld.h>这样就可以使用了。	安装下载： https://blog.csdn.ne
6	Address Sanitizer (ASan)	动态检测	AddressSanitizer（内存错误检测器）最初由google研发，简称asan，用于运行时检测C/C++程序中的内存访问错误，相比较传统工具如valgind，运行速度快，检测到错误之后，输出信息非常详细，可以通过add2line符合输出，从而	有消耗，较低	否	能	Linux i386/x86_64 、OS X 10.7 - 10.11 (i386/x86_64)、iOS Simulator、Android ARM、FreeBSD i386/x86_64	gcc -fsanitize=address - ggdb -o test test.c 编译后运行	详细使用： https://blog.csdn.ne 原理介绍： https://github.com/g hm

			直接定位到代码行，方便快速的定位问题					
	CRT					Visual C++ IDE	在程序退出前的最后一个地方调用_CrtDumpMemoryLeaks（）	
7	BoundsChecker					Visual C++ IDE		
	dmalloc		用于检查C/C++内存泄露(leak)的工具，即检查是否存在直到程序运行结束还没有释放的内存,以一个运行库的方式发布		是			
8 9	memwatch		和dmalloc一样，它能检测未释放的内存、同一段内存被释放多次、位址存取错误及不当使用未分配之内存区域		是		在需要检测的.c文件里面包含memwatch.h文件，编译的时候加上几个参数即可	
10	mpatrol		一个跨平台的 C++ 内存泄漏检测器					
	dbgmem							
	Electric Fence							
11	Performance Monitor							
	Tencent tMem Monitor							

12 本文中提到的内存泄露检测工具：

13 MFC宏定义、_CrtDumpMemoryLeaks、VLD、Tencent tMem Monitor

14 CRT:_CrtCheckMemory、bounds checker、Visual Leak Detector（详细的工作原理）

CRT、BoundsChecker、Performance Monitor

其他：

基于MTuner软件进行qt的mingw编译程序的内存泄漏检测

二、mtrace

mtrace 工具的主要思路是在调用内存分配和释放的函数中装载 “钩子（hook）” 函数，通过 “钩子（hook）” 函数打印的日志来帮助我们分析对内存的使用是否存在问题。对该工具的使用包括两部分内容，一个是要修改源码，装载 hook 函数，另一个是通过运行修改后的程序，生成特殊的 log 文件，然后利用 mtrace 工具分析日志，判断是否存在内存泄露以及定位可能发生内存泄露的代码位置。

首先需要改动一下我们的源码。添加以下两个辅助函数：

```java
#include <mcheck.h>

void mtrace(void);
void muntrace(void);
```

其中 `mtrace()` 用于开启内存分配跟踪，`muntrace()` 用于取消内存分配跟踪。具体的做法是 `mtrace()` 函数中会为那些和动态内存分配有关的函数（譬如 malloc()、realloc()、memalign() 以及 free()）安装 “钩子（hook）” 函数，这些 hook 函数会为我们记录所有有关内存分配和释放的跟踪信息，而 `muntrace()` 则会卸载相应的 hook 函数。基于这些 hook 函数生成的调试跟踪信息，就可以分析是否存在 “内存泄露” 这类问题了。

三、splint

是一个GNU免费授权的 Lint程序，是一个检查C/C++程序安全弱点和编写错误的程序。静态程序分析往往作为一个多人参与的项目中代码审查过程的一个阶段，因编写完一部分代码之后就可以进行静态分析，分析过程不需要执行整个程序，这有助于在项目早期发现以下问题：Splint会进行多种常规检查，包括未使用的变量，类型不一致，使用未定义变量，无法执行的代码，忽略返回值，执行路径未返回，无限循环等错误。

静态分析工具相比编译器，对代码进行了更加严格的检查，像数组越界访问、内存泄漏、使用不当的类型转换等问题，都可以通过静态分析工具检查出来，我们甚至可以在分析工具的分析标准里定义代码的编写规范，在检测到不符合编写规范的代码时抛出告警，这些功能都是编译器没有的。但是静态检测只是比编译器更加严谨的一点，并不能作为判断内存泄露的依据。所以至今，静态内存泄露检测只是作为一种辅助工具，只能发现一些容易被忽略的问题，并不能解决根本问题。所以还是推荐使用动态内存检测。

四、Gperftools heap-profiler

gperftools 是 google 开源的一个工具集，包含了 tcmalloc，heap profiler，heap checker，cpu profiler 等等。

gperf tools需要替换libc的malloc库，替换为tcmalloc：thread cache malloc，通过在tcmalloc加打桩，即可定位函数级别的内存的累积量

Release 版本的 C++ 程序可执行文件在编译时全部都链接了 gperftools。在 gperftools 的 heap profiler 中，提供了 HeapProfilerStart 和 HeapProfilerStop 的接口，使得我们可以在运行时启动和停止 heap profiler。同时，每个程序都暴露了 RPC 接口，用于接收控制命令和调试命令。在调试命令中，我们就增加了调用 HeapProfilerStart 和 HeapProfilerStop 的命令。由于链接了 tcmalloc，所以 tcmalloc 可以获取所有内存分配和回收的信息。当 heap profiler 启动后，就会定期的将程序内存分配和回收的行为 dump 到一个临时文件中。

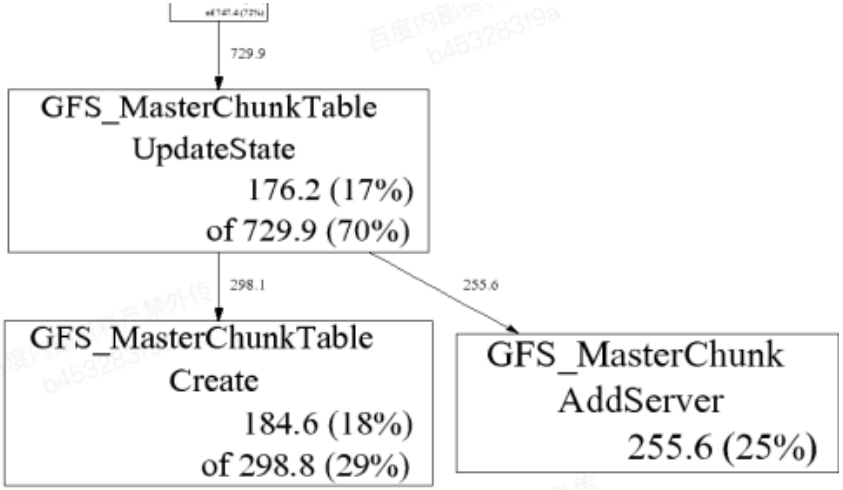
当程序运行一段时间后，将得到一组 heap profile 文件

```Java
profile.0001.heap
  profile.0002.heap
  ...
  profile.0100.heap
```

每个 profile 文件中都包含了一段时间内，程序中内存分配和回收的记录。如果想要找到内存泄露的线索，可以通过使用

```Java
pprof --base=profile.0001.heap /usr/bin/xxx profile.0100.heap --text
```

来进行查看，也可以生成 pdf 文件，会更直观一些。这样一来，就可以很方便的对线上程序的内存泄露进行 Debug 了。



五、Visual Leak Detector

Visual Leak Detector是一款免费的、健全的、开源的Visual C++内存泄露检测系统。相比Visual C++自带的内存检测机制，Visual Leak Detector可以显示导致内存泄露的完整内存分配调用堆栈。 主页地址： http://vld.codeplex.com/ 旧版地址： http://www.codeproject.com/Articles/9815/Visual-Leak-Detector-Enhanced-Memory-Leak-Detectio 下载Visual Leak Detector，当前版本2.2.3，在Visual C++ IDE的"工具"→"选项"→"项目和解决方案"→"VC++ 目录"，"包含文件"增加VLD的"\include"路径，"库文件"增加VLD的"\lib\Win32"路径，另外动态库"\bin\Win32"路径在安装时已经添加到环境变量里面了，若是未添加，则需要手动拷贝"\bin\Win32"下的文件到工程Debug目录。下面记录下使用方法： 1.新建一个Win32控制台项目； 2.添加代码如下所示：

```java
#include "stdafx.h"
#include "vld.h"

int _tmain(int argc, _TCHAR* argv[])
{
    char *pBuf = new char[200];
    return 0;
}
```

3.在Debug模式下的“输出”窗口，将有如下信息：

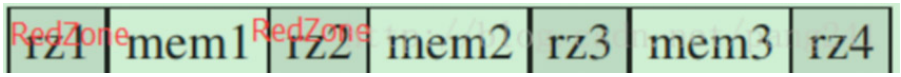
图片上传失败，请重新上传

报告列出了内存泄露是在第几块，所在的地址，泄露的字节，调用的堆栈，内存内容。双击调用堆栈可以跳转到所在行。 4.在Release模式下，不会链接Visual Leak Detector。 5.Visual Leak Detector有一些配置项，可以设置内存泄露报告的保存地（文件、调试器），拷贝"\Visual Leak Detector"路径下的vld.ini文件到工程的Debug目录下（在IDE运行的话，则需要拷贝到工程目录下），修改以下项： ReportFile = .\memory_leak_report.txt ReportTo = both

六、Address Sanitizer（ASan）

AddressSanitizer（ASan）是一个快速的内存错误检测工具。它非常快，只拖慢程序两倍左右。它包括一个编译器instrumentation模块和一个提供malloc()/free()替代项的运行时库。从gcc 4.8开始，AddressSanitizer成为gcc的一部分。

AddressSanitizer主要包括两部分：插桩(Instrumentation)和动态运行库(Run-time library)。插桩主要是针对在llvm编译器级别对访问内存的操作(store，load，alloca等)，将它们进行处理。动态运行库主要提供一些运行时的复杂的功能(比如poison/unpoison shadow memory)以及将malloc,free等系统调用函数hook住。该算法的思路是：如果想防住Buffer Overflow漏洞，只需要在每块内存区域右端（或两端，能防overflow和underflow）加一块区域（RedZone），使RedZone的区域的影子内存（Shadow Memory)设置为不可写即可。



内存映射

AddressSanitizer保护的主要原理是对程序中的虚拟内存提供粗粒度的影子内存(每8个字节的内存对应一个字节的影子内存)，为了减少overhead，采用了直接内存映射策略，所采用的具体策略如下：Shadow=(Mem >> 3) + offset。每8个字节的内存对应一个字节的影子内存，影子内存中每个字节存取一个数字k,如果k=0，则表示该影子内存对应的8个字节的内存都能访问，如果0<k<7,表示前k个字节可以访问，如果k为负数，不同的数字表示不同的错误（e.g. Stack buffer overflow, Heap buffer overflow)。

插桩

为了防止buffer overflow，需要将原来分配的内存两边分配额外的内存Redzone，并将这两边的内存加锁，设为不能访问状态，这样可以有效的防止buffer overflow(但不能杜绝buffer overflow)。

插桩后的代码：

在动态运行库中将malloc/free函数进行了替换。在malloc函数中额外的分配了Redzone区域的内存，将与Redzone区域对应的影子内存加锁，主要的内存区域对应的影子内存不加锁。 free函数将所有分配的内存区域加锁，并放到了隔离区域的队列中(保证在一定的时间内不会再被malloc函数分配)，可检测Use after free类的问题。

（不是特别懂。。）

七、CRT:_CrtCheckMemory

在使用Debug版的malloc分配内存时，malloc会在内存块的头中记录分配该内存的文件名及行号。当程序退出时CRT会在main()函数返回之后做一些清理工作，这个时候来检查调试堆内存，如果仍然有内存没有被释放，则一定是存在内存泄漏。从这些没有被释放的内存块的头中，就可以获得文件名及行号。

八、BoundsChecker

NuMega是一个动态测试工具，主要应用于白盒测试。该工具的特点是学习简单、使用方便、功能有效。NuMega共有三个独立的子功能——BoundsChecker、TrueCoverage、TrueTime。BoundsChecker为代码检错工具，TrueCoverage为测试覆盖率统计工具，TrueTime为程序运行性能测试工具。

BoundsChecker采用一种被称为 Code Injection的技术，来截获对分配内存和释放内存的函数的调用。简单地说，当你的程序开始运行时，BoundsChecker的DLL被自动载入进 程的地址空间（这可以通过system-level的Hook实现），然后它会修改进程中对内存分配和释放的函数调用，让这些调用首先转入它的代码，然后 再执行原来的代码。BoundsChecker在做这些动作的时，无须修改被调试程序的源代码或工程配置文件，这使得使用它非常的简便、直接。

不同内存检测工具对比情况

检测工具	分类	描述	性能影响	是否会修改源代码	能否指出具体位置和原因	应用场景	使用	备注
valgrind	动态检测	一个强大开源的程序检测工具	极大	否	能	适用Linux、Mac(Mac10.14后就不支持了)	valgrind --tool=memcheck ./a.out	安装下载：https://www.valgrinc 详细使用：https://blog.csdn.ne 原理介绍：https://blog.csdn.ne
mtrace	动态检测	GNU扩展, 用来跟踪malloc, mtrace为内存分配函数（malloc, realloc, memalign, free）安装hook函数	大	是	能	适用Linux	无需安装，使用时包含头文件mcheck.h，程序中调用mtrace和muntrace方法即可。 mtrace 用于开启内存使用记录，muntrace用于取消内存使用记录。内存使用情况记录到一个文件，值由环境变量：MALLOC_TRACE决定。	mtrace这个工具本身 详细使用：https://www.jianshu. 原理介绍：https://zhuanlan.zhi
splint	静态检测	一个针对C语言的开源程序静态分析工具	无	否	能	Linux、Windows	需要下载源码并安装程序，通过标志和注释来获取想要的错误信息 splint *.c（Linux）	安装下载：http://splint.org/dow 详细使用：https://www.cnblogs .html
Gperftools heap-profiler	动态检测	是 google 开源的一个工具集	有消耗，较低	是	不能，需要自己分析输出的内存变化情况	适用Linux	需要替换libc的malloc库，替换为tcmalloc：thread cache malloc，通过在tcmalloc加打桩，即可定位函数级别的内存	安装与详细使用：https://www.cnblogs

							的累积量，需要自行分析内存变化情况找到内存泄露位置。	
5	Visual Leak Detector	动态检测	免费的、开源的、强大的内存泄露检测系统，可以安装当作VS的一个插件。相比Visual C++自带的内存检测机制，Visual Leak Detector可以显示导致内存泄露的完整内存分配调用堆栈。	无	否	能指出位置，不能给原因	Visual C++ IDE上的工具	在安装完成后，在工程中指定其include和lib，然后添加头文件#include <vld.h>这样就可以使用了。	安装下载： https://blog.csdn.ne
6	Address Sanitizer (ASan)	动态检测	AddressSanitizer（内存错误检测器）最初由google研发，简称asan，用于运行时检测C/C++程序中的内存访问错误，相比较传统工具如valgind，运行速度快，检测到错误之后，输出信息非常详细，可以通过add2line符合输出，从而直接定位到代码行，方便快速的定位问题	有消耗，较低	否	能	Linux i386/x86_64 、OS X 10.7 - 10.11 (i386/x86_64)、iOS Simulator、Android ARM、FreeBSD i386/x86_64	gcc -fsanitize=address -ggdb -o test test.c 编译后运行	详细使用： https://blog.csdn.ne 原理介绍： https://github.com/ghm
7	CRT						Visual C++ IDE	在程序退出前的最后一个地方调用_CrtDumpMemoryLeaks（）	
	BoundsChecker						Visual C++ IDE		
8	dmalloc		用于检查C/C++内存泄露(leak)的工具，即检查是否存在直到程序运行结束还没有释放的内存,以一个运行库的方式发布		是				
9									

10	memwatch		和dmalloc一样，它能检测未释放的内存、同一段内存被释放多次、位址存取错误及不当使用未分配之内存区域		是			在需要检测的.c文件里面包含memwatch.h文件，编译的时候加上几个参数即可	
	mpatrol		一个跨平台的 C++ 内存泄漏检测器						
11	dbgmem								
	Electric Fence								
12	Performance Monitor								
13	Tencent tMem Monitor								
14									

本文中提到的内存泄露检测工具：

15 MFC宏定义、_CrtDumpMemoryLeaks、VLD、Tencent tMem Monitor

16 CRT:_CrtCheckMemory、bounds checker、Visual Leak Detector（详细的工作原理）

CRT、BoundsChecker、Performance Monitor

其他：

基于MTuner软件进行qt的mingw编译程序的内存泄漏检测