菜鸟也能搞定 C++内存泄漏

www.hacker.com.cn 黑客防线

背景

C++内存分配与释放均由用户代码自行控制，灵活的机制有如潘多拉之盒，即让程序员有了更广的发挥空间，也产生了代代相传的内存泄漏问题。对于新手来说，最常犯的错误就是 new 出一个对象而忘记释放，对于一般小应用程序来说，一点内存空间不算什么。但是当内存泄漏问题出现在需要 24 小时运行的平台类程序上的时候，将会使系统可用内存飞速减少，最后耗尽系统资源，导致系统崩溃。

所以学会如何防止并检查内存泄漏，是一个合格的 c++程序员必须具备的能力。但是由于内存泄漏是程序运行并满足一定条件时才会发生，直接从代码中查出泄漏原因的难度较大，而且一旦内存泄漏发生在多线程程序中，从大量的代码中要靠人工找出泄漏原因，无论对新人还是老手都是一场噩梦。

本文介绍一种在 vs2003 中检查内存泄漏的方法，供各位新人老手参考，在 vc6 中实现需要做一些变动，详情可自行参照相关资料。

检查策略分析

首先，假定我们需要检测一个 24 小时运行的平台程序的内存泄漏情况，我们无法确定具体的内存泄漏速度，但是我们可以确定该程序在一定时间内（如 10 分钟）泄漏的内存量是接近的，设为 L(leak)。

考虑在 10 分钟的运行时间内程序新申请到的内存 A(alloc)，这部分内存其实包含了程序运行正常申请，并会在后续运行中进行释放的普通内存块 N(normal)和泄漏的内存 L，即：

A = N + L

在后续的运行中，由于 N 部分不断的申请和释放，所以这部分的总量基本上是不变的，而 L 部分由于只申请而不释放，占用的内存总量将会越来越大。

将这个结果放到运行时间轴上，现在我们观察程序运行中的 20 分钟，我们假定内存泄漏速度为 dL/10 分钟，时间轴如下：

----------|----------|----------|----------
Tn-2 Tn-1 Tn

三点间隔均为 10 分钟，则我们有如下结论：

Tn 点总的内存分配量 An = N + dL * n，N 为正常分配内存，dL*n 为内存泄漏量的总和，而 Tn-1 点的内存总量则为 An-1 = N + dL*(n-1)。注意，我们这里不考虑释放的内存量，仅考虑增加的内存量。因此很明显单位时间内的内存泄漏量 dL = An - An-1。

生成内存 Dump 文件的代码实现

要完成如上的策略，我们首先需要能跟踪内存块的分配与释放情况，并且在运行时将分配情况保存到文件中，以便进行比较分析，所幸 m$已经为我们提供了一整套手段，可以方便地进行内存追踪。具体实现步骤如下：

包含内存追踪所需库

在 StdAfx.h 中添加如下代码，注意必须定义宏_CRTDBG_MAP_ALLOC，否则后续 dump 文件将缺少内存块的代码位置。

```
#ifdef _DEBUG
//for memory leak check
#define _CRTDBG_MAP_ALLOC //使生成的内存 dump 包含内存块分配的具体代码为止
#include<stdlib.h>
#include<crtdbg.h>
#endif
```

启动内存追踪

上述步骤完成后，则可以在应用程序启动处添加如下代码，启动内存追踪，启动后程序将自动检测内存的分配与释放情况，并允许将结果输出。

```
//enable leak check
_CrtSetDbgFlag( _CRTDBG_REPORT_FLAG);
```

将结果输出指向 dump 文件

由于默认情况下，内存泄漏的 dump 内容是输出到 vs 的 debug 输出窗口，但是对于服务类程序肯定没法开着 vs 的 debug 模式来追踪内存泄漏，所以必须将 dump 内容的输出转到 dump 文件中。在程序中添加如下部分：

```
HANDLE hLogFile;//声明日志文件句柄
hLogFile = CreateFile("./log/memleak.log", GENERIC_WRITE,
FILE_SHARE_WRITE|FILE_SHARE_READ,
NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);//创建日志文件
_CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);//将 warn 级别的内容都输出到文件（注意 dump 的报告级别即为 warning）
_CrtSetReportFile(_CRT_WARN, hLogFile);//将日志文件设置为警告的输出文件
```

保存内存 Dump

完成了以上的设置，我们就可以在程序中添加如下代码，输出内存 dump 到指定的 dump 文件中：

```
_CrtMemState s1, s2, s3;//定义 3 个临时内存状态
.....
_CrtDumpMemoryLeaks();//Dump 从程序开始运行到该时刻点，已分配而未释放的内
```

```
存，即前述 An
//以下部分非必要，仅为方便后续分析增加信息
_CrtMemCheckpoint( &s2 );
if ( _CrtMemDifference( &s3, &s1, &s2) )
{
_CrtMemDumpStatistics( &s3 );//dump 相邻时间点间的内存块变化
//for next compare
_CrtMemCheckpoint( &s1 );
}
time_t now = time(0);
struct tm *nowTime = localtime(&now);
_RPT4(_CRT_WARN,"%02d %02d:%02d:%02d snapshot dump.\n",
nowTime->tm_mday, nowTime->tm_hour,nowTime->tm_min,nowTime->tm_sec);//输出该次 dump 时间
```

以上代码最好放在一个函数中由定时器定期触发，或者手动 snapshot 生成相等时间段的内存 dump。

dump 文件内容示例如下：

```
Detected memory leaks!
Dumping objects ->
{20575884} normal block at 0x05C4C490, 87 bytes long.
Data: < > 02 00 1D 90 84 9F A6 89 00 00 00 00 00 00 00 00
...
d:\xxxxx\xxxworker.cpp(903) : {20575705} normal block at 0x05D3EF90, 256 bytes long.
Data: < > 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
Object dump complete.
0 bytes in 0 Free Blocks.
215968 bytes in 876 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 220044 bytes.
Total allocations: 7838322 bytes.
10 16:29:14 snapshot dump.
```

上面红色部分即为用户代码中分配而未释放的内存块位置。

解析 Dump 文件

前面我们已经通过 dump 文件获取到各时刻点的内存 dump，根据前面的分析策略，我们只需要将第 n 次 dump 的内存块分配情况 An，与第 n-1 次 dump 内存块分配情况 An-1 作比较，即可定位到发生内存泄漏的位置。由于 dump 文件一般容量巨大，靠人工进行对比几乎不可能，所以仅介绍比较的思路，各位需要自行制作小工具进行处理。

1、提取两个相邻时间点的 dump 文件 D1 和 D2，设 D1 是 D2 之前的 dump

2、各自提取 dump 文件中用户代码分配的内存块（即有明确代码位置，而且为 normal block 的内存块），分别根据内存块 ID（如“d:\xxxxx\xxxworker.cpp(903) : {20575705}”红色部分）保存在列表 L1 和 L2

3、遍历列表 L2，记录内存块 ID 没有在 L1 中出现过的内存块，这些内存块即为可能泄漏的内存

4、根据 3 的结果，按照内存的分配代码位置，统计各处代码泄漏的内存块个数，降序排列，分配次数越多的代码，内存泄漏可能性越大。

系统状态报告