内存泄漏的检测、定位和解决经验总结

【摘要】

结合局端 MCU 项目中 CSS、NMS 模块内存泄漏检测、修正的过程，简要介绍了内存泄漏检测的工具，提出了内存泄漏检测的一些方法(怎样对程序结构进行改造，怎样对程序进行隔离以易于进行内存泄漏检测)。总结了内存泄漏检测过程中成功和失败的体会，希望能对后来者有所启发。

【关键词】

内存泄漏

一、故障或失误概况

局端 MCU 项目中 CSS(Conference Schedule System)、NMS（NetWork Management System）模块自 2.03 版本起就有内存泄露的问题，开发 NGN 版本时也花过大量的精力来争取解决这个问题，虽然也修正了一些内存泄漏，但最终检测工具表面现象显示剩下的内存泄露都是所使用的开发库的代码产生的，于是也就大意的认为是所使用的 ACE/TAO 库本身有内存泄漏，于是无果而终，使这两个模块的内存泄漏问题一直延续到 2.03.20x 版本。

由于后续测试部和开发部进行测试时引入了 Robot 来进行自动测试，使业务操作量巨升，此时 CSS 模块的内存泄漏问题就更明显了，从程序启动时的 13M 内存，经过一两个月后可以飙升到 200 M 左右，正因为此局端 MCU 项目好几个程序都采用了看门狗的方式来定时检测程序的状态防止程序当掉。

由于问题比较严重，于是再次进行内存泄漏问题的攻关，测试时 NMS 模块业务操作量小内存泄漏不明显，于是此次攻关重点是查 CSS 的内存泄漏问题。

此次再次进行 CSS 行内存泄露问题的研究，力争解决 CSS 的内存泄露问题，实在找不到解决方案(例如为所使用的开发库的原因)也定位出具体原因供项目组参考。

二、诊断过程

2.1 工具介绍

目前 Windows 平台上流行的内存泄露检测工具有 Rational Purify、BoundsChecker、insure++，由于 CSS 较复杂又是多线程采用 Rational Purify 工具程序就启动不起来，无法进行检测；parasoft 公司的 insure+

+工具传说中比较好用，但由于我们公司没有相应的 licence 无法使用；最后选择采用限制版的 BoundsChecker 来进行检测。

还有就是要一个好的实时检测程序使用内存状况并能实时记录下每个时刻内存使用情况的工具，我们这里使用的是本部传输产品测试部自己开发的一个小工具 MemSample。(见附录 1)

对于工具方面这也是个问题，若每个公司都有个统一可用工具列表，然后附上相应的使用说明就可以省了许多精力和时间。现在好多工具都是自己去浩如烟海的网上找，有些还需要 licence 等，这是个很大的精力浪费。

2.2 诊断概况

CSS 模块为后台服务程序通过访问 Web 页面的形式来对外提供服务，运行了该模块的程序过一段时间后机器性能明显下降，经检测为内存严重消耗。测试部使用内存监视工具 MemSample 对各模块的内存进行监测，发现在一个月内 CSS 程序占用的内存从初始的 13M 飙升到 200 多 M。

由现象可以判断肯定是存在内存泄漏问题，由于 CSS 对外提供服务接口繁多一时不知道那个接口导致的内存泄漏，因此确定如下测试过程:

测试方法 1:通过单元测试的方法来调用部分接口进行测试。

- 采用 CPPUNIT 测试框架辅助以桩程序的方式来模拟 Web 页面手工操作的方式来调用相应接口进行测试。
- 程序测试过程中采用内存泄漏代码级检测工具 BoundsChecker 全程跟踪程序中内存分配和释放情况。
- 采用内存监测工具 MemSample 对程序占用的整体内存进行监测并采集各时间段内存占用数据。

经过第一种测试方法发现了程序中存在的一些内存泄漏，并解决之，但解决之后发现还是存在内存泄漏，通过分析发现模拟测试步骤和真实环境下的操作唯一的不同点就是：

模拟测试是在程序内部直接调用接口，而真实环境下是通过访问 Web 页面，Web 页面再通过 CORBA 协议来调用程序接口。即模拟测试少了 CORBA 调用那部分的操作，为此重新制定第二种测试方法：

测试方法 2:真实操作环境下进行测试

- 由于接口繁多，重点挑一两个使用频繁的接口进行测试。
- 人工访问调用所测接口的那个 Web 页面，不断频繁刷新，保证和真实操作环境完全一致。
- 测试过程中启动内存监测工具 MemSample 对 CSS 占用内存进行监测并采集各时间段内存占用数据。

测试后发现所测接口均存在内存泄漏，每次接口调用都会有几十 K 的内存泄漏，此时扩展到其它接口发现所有使用 CORBA 协议通过 Web 页面调用的接口均存在内存泄漏。比较两种测试方法可以初步确定是在调用 CORBA 协议部分的代码出现了内存泄漏，下面进行整个检测过程详细描述。

2.3 诊断过程

采用 BoundsChecker 检测和 Rational Purify 工具不一样，Rational Purify 可以在程序运行中随时给出即时的内存泄漏报告，而使用 BoundsChecker 则只有在程序自行结束后才会触发它去收集数据给出内存泄漏报告(注意：必须是程序自己运行完毕退出，而不能人为的去关闭该程序否则得不到内存泄漏报告)。

2.3.1 对程序的改造

由于要让程序自动结束，所以要对程序结构进行改造，前段时间部门推行的单元测试为此次内存泄漏检测提供了很好的基础。

CSS 模块为单元测试专门创建了一个新的 VC Project，共用 CSS 中的所有代码，但自己工程中的单元测试代码又不会影响 CSS 原来的代码。此次进行内存泄漏的检测就首先打算运行单元测试的代码然后检测是否有内存泄漏。

单元测试程序改造如下：

1)程序启动时开启两个线程，一个 CSS 线程负责正常启动 CSS，另一个测试线程负责运行测试用例对已经正常启动的 CSS 进行测试。

2)测试线程运行测试用例完毕后要让 CSS 线程退出，并结束自己，只有这样 BoundsChecker 才能拿到内存泄漏数据。

2.3.2 内存泄漏检测

第一次内存泄漏检测，CSS 测试程序运行一遍测试用例正常退出后，得到 BoundsChecker 的检测报告显示：

- 所有采取单态模式的类中的相应的指针指向的动态分配的对象都没有被释放。
- 线程被异常终止时，该线程中动态分配的内存没有释放掉。
- 动态分配的全局变量(对象)没有被释放掉。
- 绝大部分的内存泄漏集中在 ACE/TAO(ACE 的 CORBA 库)提供的一些函数里。

其中前三种情况就是 1.问题描述中提到的在第一次内存泄漏检测就解决的，最后一个是造成内存泄漏问题严重的罪魁祸首，在第二次内存泄漏检测才彻底解决，这里列下相应的解决方法和思路。

2.3.3 内存泄漏修正方法

BoundsChecker Data Collection 设置中将跟踪的 call stack 深度加大即可以晰的看到每个内存泄漏是从哪个函数哪条语句产生的。BoundsChecker 出的报告中内存泄漏数据和资源泄漏数据对于查找内存泄漏问题都是非常有帮助的。[见附录 2、3]

根据报告进行内存泄漏定位发现泄漏分为两种：

1)在程序关闭时候的内存泄漏，只有程序关闭时才发生的内存泄漏，如一些动态分配的全局对象，线程没有正常退出而是直接被 Terminate 掉等。

2)程序运行中的内存泄漏，即运行中会不断发生泄漏，如一段代码动态分配了内存忘释放了，这样这段代码每次运行都会产生内存泄漏。

2.3.3.1 程序关闭时内存泄漏检测和修正方法

虽然程序关闭时才发生的内存泄漏不会导致程序运行中内存使用的增长，但由于 BoundsChecker 给出的报告中只给出程序运行期间发生的所有内存泄漏，所有关闭时内存泄漏和运行时内存泄漏都混杂在一起不好定位，为此首先要解决关闭时的内存泄漏问题，由于关闭时内存泄漏一般都是动态分配的全局变量没释放、线程不是正常退出等情况产生的所以比较好定位。解决方法如下：

1)所有动态分配的全局变量在程序退出时都 Delete 掉。

2)让所有线程都正常运行结束退出，而不是被异常 Terminate 掉。

3)所有使用单态模式的类都必须有专门的内存清理函数在程序结束前调用。

经过以上方法的处理后，CSS 程序剩下的内存泄漏全部集中在 ACE/TAO(ACE 的 CORBA 库)提供的 dll 上。

2.3.3.2 程序运行时内存泄漏检测和修正方法

第二次内存泄漏检测运行 CSS 测试程序得到的检测报告发现的内存泄露全部集中在 ACE/TAO(ACE 的 CORBA 库)提供的 dll 上，原因可能有如下两种：

1) 程序中使用 ACE 和 TAO 的函数不当没有正确释放内存。

2) ACE 和 TAO 库确实是存在内存泄露问题。

由于此时显示内存泄漏都是在 TAO 的函数中发生的，一下子没了思路不知道怎么查了，只好一步一步的去查程序代码了。

一开始测试程序只是将所有测试用例执行一遍然后就退出，为了测试是否会有运行时内存泄漏，让测试程序将所有测试用例运行多遍才退出，理论上若有运行时内存泄漏的话，运行 n 遍测试用例产生的内存泄漏应该是运行一遍产生的内存泄漏的 n 倍。照着这个思路往下走，发现实际上并不是这么回事，基本上内存泄漏的字节数不变，由此可见测试用例没有覆盖到真正产生运行时内存泄漏的代码，目前报告中发现的内存泄漏都只是程序关闭时产生的内存泄漏。

怎样定位哪段代码会产生运行时内存泄漏呢？程序运行时内存泄漏问题比较难定位，特别是程序有多个线程时，CSS 内存泄漏检测首先采用的方法为进行线程隔离。

1) 线程隔离法

将程序中主要的可能产生内存泄漏的线程一个个进行隔离，每次只运行其中的一个或关闭其中的一个，然后看 BoundsChekcer 产生的内存泄漏字节数是否有变化，以此来判断该线程是否会产生内存泄漏。若该线程会产生内存泄漏则对该线程的代码进行仔细分析以找出原因。

但 CSS 程序进行线程隔离后也发现这些线程都不产生内存泄漏。

2) 功能检查法

此时只好从 CSS 的功能上来定位了，CSS 主要是和 Web 进行交互，接受 Web 发过来的指令然后进行开关会等操作。分析由于测试程序只是测试 CSS 的内部流程，而没有涉及到 Web 和 CSS 的接口那块，而 Web 和 CSS 之间是使用 CORBA 机制通讯的，恰好内存泄漏报告也显示内存泄漏基本集中在 TAO(CORBA 开发库)那部分。

于是选择一个 Web 页面(获取 mcu 列表)，不断的刷新该页面，发现此时 CSS 使用内存果然在不断上涨。也就是所获取 mcu 列表这个功能对应的代码会产生内存泄漏。

由于这个小功能对应的代码是有限的就几十行，仔细的检查也没发现什么问题，此时想到测试部认为 NMS 模块没有内存泄漏，只要对比 NMS 模块的类似功能的代码就能知道内存泄漏的原因了。但检查发现两边代码是一样的，然后去测 NMS 的相关页面发现 NMS 也有内存泄漏，只是因为 NMS 业务量比较小不太容易暴露该问题而已。

此时更相信可能是 TAO 开发库自己产生的内存泄漏了，正要放弃之际，慕然出现了转机，我们科室的李喜欣也使用了 TAO 库，此时彭峰和他也参与进内存泄漏的攻关，经检测发现喜欣的程序没有内存泄漏。

一开始认为是使用的 TAO 库的版本不一样，于是就将我们的代码放入李喜欣 TAO 库所在环境中测试，发现在他哪里还是也有内存泄漏，也就是所和 TAO 库的版本没有关系。可能确实是代码中函数使用方法有误。

于是针对那部分功能代码开始一条条**语句屏蔽**以期找出产生内存泄漏的语句，功夫不负有心人当我们将一条 memset()语句屏蔽掉后终于没有内存泄漏了。

分析原因是由于 memset 将一个 CORBA 对象所占内存清零了，虽然 IDL 中定义的为一个结构，但经过 IDL 文件的编译，实际上在 CORBA 已将 IDL 中的结构转换为相应的 CORBA 对象了，该对象中含有智能指针，这些指针在对象创建时都已经指向了一些动态分配的空间，由于 memset 操作，导致这些指针全部清零，于是那些指针原来指向动态分配的空间就永远都得不到释放进而产生了内存泄漏。

修正 memset 问题后就彻底的解决了内存泄漏问题。

三、推广建议

本文简要介绍了进行内存泄漏检测的一些工具，并提出了一些进行内存泄漏检测的有效方法和经验，对于要进行内存泄漏检测的项目具有一定的借鉴意义。

若需要文中提到的一些内存检测工具可以联系作者。

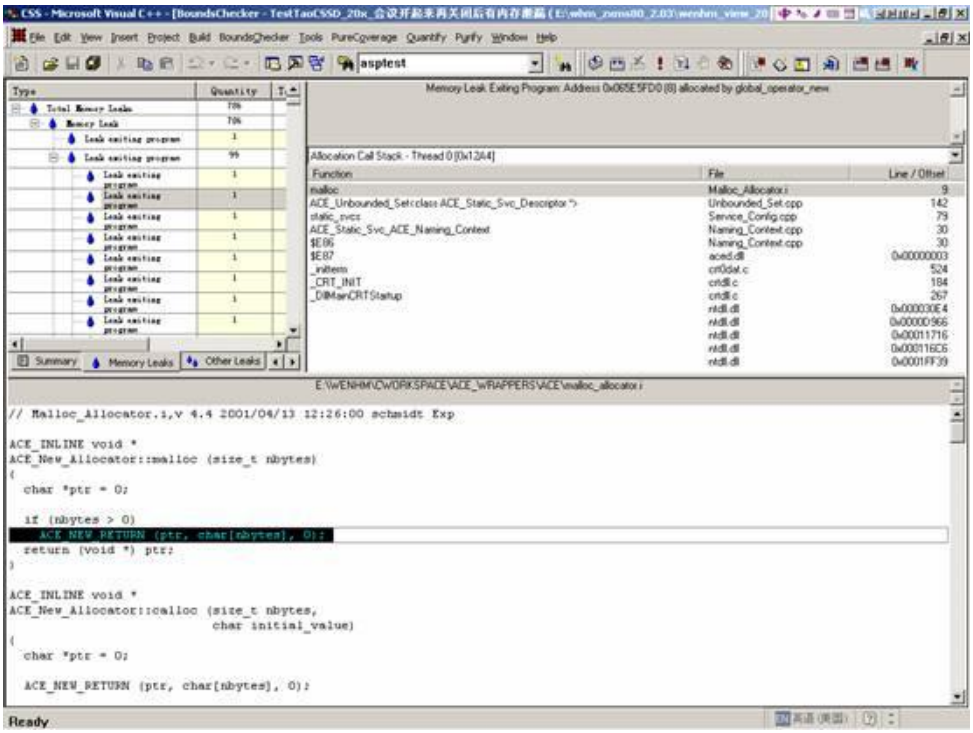
四、后记

经过漫长的内存泄漏检测终于将内存泄漏问题解决了，修正 CSS 和 NMS 的内存一直都稳定在 13M 左右，在查找内存泄漏的问题中有以下几点体会：

1. 不要随便怀疑别人首先要怀疑自己，第一次内存泄漏检测没有彻底找出原因就是因为认为是开发库自己的内存泄漏，第二次也差点就打算收工了。最后发现还是自己使用不当导致的。
2. 要有耐心，不能烦躁，当一个问题你找不到一丝丝线索时多和别人交流是很有帮助的，这次能彻底解决内存泄漏就是由于彭峰和李喜欣的加入，三个丑皮匠顶个诸葛亮真是至理名言。
3. 工具很重要，使用好的工具能事半功倍。

附录 1.监视程序内存使用的工具：MemSample

附录 2:BoundsChecker 内存泄漏数据截图



附录 3:BoundsChecker 资源泄漏数据截图