1. **专业工具**

**2.1版本控制**

版本控制系统(version control system)允许多位开发者在同一组文件上工作。由于版本控制系统记录了每个文件的历史,所以它可以追踪文件中的每个改动,需要时还可以把改动还原。版本控制系统允许多位用户同时修改文件,甚至修改同一个文件,并避免互相破坏成果。因为版本控制系统主要供程序员管理源代码使用,所以有时候又称其为源代码控制(source control)。版本控制系统也可以用来管理其他类型的文件,一般以文本为佳,下文将探讨其中的原因。许多游戏工作室使用单一版本控制系统,同时管理文本类型的源代码,以及以二进制文件为主的游戏资产,如纹理、三维网格、动画、音频文件等。

**为何使用版本控制**：多位工程师组成团队合作开发软件时,版本控制至关重要。版本控制系统有以下功能。

1. 提供中央版本库(repository)，工程师们可以共享其中的代码。
2. 保留每个源文件的所有更改记录。
3. 提供为某些版本加上标签的机制,供以后提取已加标签的版本。
4. 允许代码从主生产线上建立分支(branch)。这一功能经常用来制作示范程序,或是为较旧的软件版本制作补丁(patch)。

源代码控制系统甚至在单人开发的项目里也有所应用。单人开发的项目虽然用不上多人开发的功能,但是其版本控制功能,如维护历史修改记录、为版本添加标签、建立示范程序/补丁的分支、追踪缺陷等,仍然是非常有用的。

**常见的版本控制系统**：常见的版本控制系统如下：

1. SCCS和RCS:源代码控制系统(Source Code Control System, SCCS)和版本控制系统(Re-vision Control System，RCS)是两个古老的版本控制系统。两者皆使用命令行界面,主要用于UNIX 上。
2. CVS:并发版本管理系统(Concurrent Version System,CVS)是高强度、专业级、基于命令行接口的版本控制系统,原本建立在RCS 之上(但CVS 现在已成为独立工具)。CVS在UNIX上应用较多,但在其他开发平台如微软Windows也能使用。CVS是开源的,并按GPL授权。CVSNT (也称为 WinCVS)是一个原生的 Windows实现,基于CVS 并和CVS兼容。
3. Subversion: Subversion (简称SVN)是一个开源版本控制系统,其目标是取代并改进CVS。因为Subversion开源且免费,是个人项目、学生项目和小工作室的首选。
4. Git: Git是开源版本控制系统,用于许多受人敬佩的项目,包括Linux内核。在Git 开发模型里,程序员把文件的变更提交到一个分支上。之后，该程序员可以轻易地把其修改合并到任何一个分支上,因为Git“知道”如何回溯文件的区别(diff)，并把区别重新应用在新的基修订版(base revision)上,这个过程Git 将之称为衍合(rebasing)。此开发模型使Git在处理多个代码分支时非常高效和快捷。有关Git的更多信息可参考官网。
5. Perforce:Perforce是专业级的源代码控制系统,同时支持基于文本和GUI的接口。Perforce成名之处在于其变更列表(changelist)的概念。变更列表,指被视为同一个逻辑单元而进行修改的源文件集合。变更列表会以原子方式(atomically)签入(check-in)版本库,即,要么整个变更列表成功提交,要么没有东西提交进去。许多游戏公司使用Perforce,包括顽皮狗和艺电。
6. NzN Alienbrain: Alienbrain是针对游戏产业而特别设计的强大版本控制系统,具有丰富的功能。其最著名的特点是支持包含文本及二进制游戏资产的海量数据库,并配合可定制的用户界面,以针对特定的人员,如美术设计师、制作人及程序员等。
7. ClearCase: ClearCase是专业级的源代码控制系统,是为超大规模的软件项目而开发的。ClearCase功能强大,并且提供独特的用户接口，以扩展Windows资源管理器的功能。笔者未曾见过游戏业界内使用ClearCase，可能是因为其价格较为昂贵。
8. 微软的 Visual SourceSafe: SourceSafe是轻量级的源代码控制软件包,已成功地应用于一些游戏项目中。

**2.2-IDE**

编译式语言，如C++。需要使用编译器和链接器，将源代码转换成可执行程序。

**源文件、头文件及翻译单元**：用C+＋编写的程序由源文件(source file)所组成。常见的C++源文件的扩展名为.c、.cc、.cxx和.cpp,这些文件中包含程序的大量源代码。因为编译器每次只翻译一个C++源文件至机器码,所以在技术上,源文件被称为翻译单元(translation unit)。

有一种特殊的源文件被称为头文件(header file)。头文件通常用于在多个翻译单元之间分享信息,例如类型声明及函数原型。C++编译器并不“了解”头文件,实际情况是,C+＋预处理器(preprocessor)预先把每个#include语句替换为相对应的头文件内容,然后再把翻译单元送交编译器。这是头文件和源文件之间一个细微但非常重要的区别。从程序员角度来看,头文件是独立的文件,但多亏有预处理器把头文件展开,编译器接收到的才都是翻译单元。

**程序库、可执行文件及动态链接库**：编译翻译单元后,输出的机器码会存储在对象文件(在Windows下采用.obj扩展名,在基于UNIX的系统里则是.o)中。对象文件中的机器码具有如下特征。

1. 可重定位的：未决定代码的内存地址
2. 未链接的：未解决的外部函数参考，以及翻译单元外定义的全局数据

对象文件可以集合成程序库(library)。程序库只是一个简单的存档(archive),像ZIP或tar文件一样,包含零到多个对象文件。程序库只是为方便而设置,允许把大量的对象文件集合成单个易用的文件。

链接器把对象文件和程序库链接成可执行文件(executable)。可执行文件包含完全解析的机器码,操作系统可载入及执行这些机器码。链接器的工作包括:

1. 计算全部机器码的最终相对地址，即当程序执行时机器码在内存中的分布
2. 确保正确地解析每个翻译单元（对象文件）中地所有外部函数参考和全局数据

可执行文件里的机器码仍然是浮动的，即文件中的所有指令和数据地址相对于一个任意的基址,而非绝对地址。直至程序载入内存,在执行之前,程序的最终绝对基址才会决定下来。

动态链接库(dynamic linked library, DLL)是一种特殊的库,其行为像正常的静态链接库和可执行文件的混合体。DLL的行为像库,因为它包含函数,供其他多个不同的可执行文件调用。然而, DLL的行为也像可执行文件,因为操作系统能独立地载入DLL,而且DLL可包含启动及终止代码,其执行方式和C++可执行文件的main ()函数相似。

使用了DLL的可执行文件含有未完全链接(partially linked)的机器代码。在最后的可执行文件中,已解析大多数函数及数据参考,但是存于DLL的函数和数据参考则维持未链接的状态。当运行可执行文件时,操作系统需要解析所有未链接的函数。在此过程中,操作系统会找出合适的DLL文件,若该DLL文件不在内存中则要载入,之后需要修正一些内存地址。载人动态链接库是操作系统非常重要的功能,因为这样就可以只更新个别 DLL,而不需要更新使用到这些DLL的可执行文件。

**生成配置**：C/C++的预处理器、编译器和链接器都提供了大量选项,用来控制代码生成的方式。这些选项通常由执行编译器的命令行设定。例如,使用微软编译器,生成一个翻译单元的典型命令行如下:

C :\> cl /c foo.cpp /Fo foo.obj /wall /od /zi

此命令行告诉编译器和链接器编译但不链接(/c)foo.cpp翻译单元,将结果输出到foo.obj对象文件(/Fo foo.obj),打开所有警告(/wall),关掉所有优化(/od)并产生调试信息(/zi)。

现代的编译器提供了大量选项,每次生成代码时都重新指定这些选项,这既不现实又易犯错,因此生成配置(build configuration)应运而生。生成配置是解决方案内个别项目的预处理器、编译器和链接器的选项集合。程序员可设置任意数量的生成配置,可任意命名,并在每个生成配置中设定不同的预处理器、编译器和链接器选项。默认把同一组选项应用到项目中的每个翻译单元,但也可以在个别翻译单元上做特殊设置,以替代项目的全局设置。(笔者建议,如非必要,避免使用此设置方式,因为很难分辨哪些.cpp有自定义设置,哪些没有。)

多数项目都至少有两个生成配置,通常名为“调试(Debug)”和“发布(Release)”。发布生成做最终软件出版之用,而调试生成则做开发之用。调试生成比发布生成运行得慢,但调试生成向程序员提供了宝贵的开发及调试信息。

**常用生成选项**：列举一些游戏引擎项目生成设置中最常见的选项

1. 预处理器设：C++预处理器处理#include文件的展开,以及处理#define宏(macro)的定义和替换。所有现代的C++预处理器皆有一个极强大的功能,就是可以通过命令行定义预处理宏(因而也能通过生成配置定义)。用这种方式定义宏,和在代码中编写#define 指令等效。多数编译器提供此功能的命令行选项为-D或/D,此选项可出现多次。

此功能让生成选项和代码沟通，而不需要修改代码本身。举一个常见的例子，在调试生成中必然会定义\_DEBUG 符号,而在发布生成中会定义NDEBUG符号进行替代。源代码可以检查这些符号,去“了解”目前是生成调试模式还是发布模式。这称为条件编译(conditional compilation)。

编译器也可以基于其编译环境和目标平台的信息，自由地将“魔法”预处理宏加入代码中。例如,当编译一个C++文件时,大多数编译器会定义\_\_cplusplus宏,从而能编写代码自动地适应C或C++编译。

又例如,每个编辑器都会通过一个“魔法”宏,让代码识别编译器。当用微软的编译器编译代码时,编译器会定义\_MSC\_vER宏;当使用GNU编译器(gcc)时,则会定义\_GNUC\_宏,其他编译器也如是。与此相似,执行代码的目标平台也是用宏来定义的。例如,生成32位Windows机器的执行代码时,就会定义\_WIN32符号。可以利用这些关键功能去编写跨平台代码,因为这些宏使代码“了解”目前被哪个编译器编译,并需要编译至哪个目标平台。

1. 编译器设置：控制编译器产生的对象文件是否包含调试信息(debugging information)是最常见的编译选项之一。调试器使用此信息去逐步执行代码、显示变量的值等。调试信息会增大磁盘上的可执行文件大小,也会方便黑客做反向工程。因此,最终发布的可执行文件必会去除这些调试信息。然而,在开发期间,调试信息是无价之宝,应该经常包含在生成的代码中。

另外,也可以控制编译器是否展开内联函数(inline function)。如关掉内联函数展开,每个内联函数在内存中只有一份,有唯一的内存地址。这样设置,使用调试器追踪代码时就容易得多,但其明显的代价是放弃了正常内联函数执行速度的提升。

内联函数展开是被称为优化(optimization)的泛代码转换例子之一。可以使用编译器选项控制编译器去尝试优化代码的进取性(aggressiveness),以及使用哪些优化方法。优化可能会打乱代码里的语句次序,完全去除一些变量,把变量移到不同地方,或在函数里将CPU寄存器(register)作为新用途重复使用。经优化的代码常会迷惑大多数调试器,令调试器以不同方式对用户“说谎”，并难以观察真实的执行情况。因此,在调试生成中,通常会关上优化选项。这样一来,每个变量、每行代码都会和原来编写的保持一致。但是,未经优化的代码,执行时较完全优化的代码会慢许多。

1. 链接器设置链接器也提供了多个选项,例如,控制输出文件的类型(如可执行文件或DLL)指定将哪些外部库链接至可执行文件,以及指定搜索哪些程序库的路径。惯例之一,调试时,可执行文件链接调试用的库,发布版本则链接优化的库。

链接器选项也可控制堆栈大小、程序载入内存时的首选基址、代码在哪些平台上执行(以做平台相关的优化),以及许多其他细节选项,不在此展开叙述。

**典型生成配置**：通常,游戏项目不止有两种配置。以下是游戏开发中的一些常见配置。

1. 调试(Debug):调试生成版本是非常慢的程序版本。此版本关闭了各种优化,禁用了所有函数内联,并且包含完整的调试信息。此生成版本用来测试新代码,以及调试在开发过程中出现的几乎所有最不平凡 (nontrivial)的问题。
2. 发布(Release):发布生成版本是较快的程序版本,但仍然保留调试信息并开启断言(assertion)。游戏能表现接近最终产品的运行速度,并留有机会去调试问题。
3. 制作(Production):制作生成配置是为生成最终发行给消费者的游戏版本而设置的。此配置有时也被称作“最终(Final)”或“光盘(Disk)”配置。制作生成配置与发布生成的差别在于前者去除了所有调试信息,通常关闭了所有断言,并完全启动优化。调试制作生成版本非常棘手,但制作生成版本是最快及最精干的生成类型。
4. 工具(Tools):有些游戏工作室的工具和游戏本身会共用代码库。在此方案中，加入“工具”生成配置很合理,用于为工具条件编译共用代码。工具生成配置一般会定义一个预处理宏(如TOOLS\_BUILD)，以告之代码当前是在生成工具用的版本。例如，某个工具可能需要一些C++类提供编辑用函数，而这些函数在游戏中并不需要,那么就可以用#ifdef TOOLS \_BUILD... #endif指令包围这些函数。由于工具通常也要分调试和发布版本,所以开发者会建立两个工具生成,如命名为“ToolsDebug”及“ToolsRelease”。

混合生成版本：混合生成版本(hybrid build)是指在其配置中,大部分翻译单元是发布模式，只有一小部分翻译单元为调试模式。使用这种配置,容易调试当前要监视的代码,而其余的代码能继续以全速运行。

基于文本的生成工具，如 make，能很容易地设置混合生成。用户能以翻译单元为单位把某些翻译单元设置为调试模式。大致做法是:定义一个make变量,如 $HYBRID\_SOURCES,列举所有要设置为调试模式的翻译单元(.cpp文件);设置生成规则,编译所有翻译单元的调试及发布两个版本,并将每个对象文件(.obj/.o)按其版本分别输出到两个文件夹;设定最终的链接规则,链接$HYBRID\_SOURCES列举的对象文件调试版本,以及其他对象文件的发布版本。若设置正确,make的依赖规则能处理余下的工作。

生成配置和可测试性：项目支持的生成配置越多,也就越难测试。虽然配置之间可能相差无几,但一些bug仍有可能只出现在某个配置中，而不出现在其他配置中。因此,每个配置都必须彻底测试。多数游戏工作室并不正式测试调试生成版本，因为调试配置主要用于开发新功能时,以及在其他配置遇到问题时做调试之用。然而,若测试人员花大部分时间测试发布配置,那么并不能在制作母片(gold master)2的前夜,直接制作出游戏的制作生成版本,并期望它的 bug状况与发布生成版本一模一样。实际上,在alpha到beta测试阶段,测试团队应同样彻底地测试发布及制作生成版本,保证制作生成版本不会暗藏任何烦扰人的意外情况。保持最少数量的生成配置,最有利于测试。事实上,有些工作室为此而不加入制作生成配置,彻底地测试发布生成后,就直接发行发布生成(但这去除了调试信息)。

**调试代码**：

1. 断点：断点(breakpoint)是代码调试的基本所需。每个断点都可以使得调试器在程序中的某行停下来,以便观察程序当时的运行状况。
2. 单步执行代码：当断点被命中时,可单步执行代码。
3. 调用堆栈：命中断点时(或手动暂停程序),双击调用堆栈窗口中的条目,就可在调用堆栈里上下移动。此操作非常有用,能检查从main()开始调用至目前代码行的一连串函数调用。例如,有时候在深层的嵌套函数调用中,此方法能往上追查父代函数,从而找出bug的源头。
4. 监视窗口：当单步执行代码并在调用堆栈里上下移动时，程序员需要检查程序中变量的值。监视窗(watch window)就是为此而设置的
5. 数据断点：常规的断点触发条件是CPU的程序计数器命中某个机器指令或代码行。然而,现在的调试器提供另一个极有用的功能,就是能够设立另一种断点,其触发条件是数据写入(即改变)某指定地址,所以这种断点称为数据断点(data breakpoint)。由于它是通过CPU的一个特殊功能而实现的,该功能可以在指定地址被写入时引发一个中断(interrupt)，所以这种断点又称为硬件断点(hardware breakpoint)。

以下介绍数据断点的典型用法。例如,在追查一个bug 时，发现某个对象的成员变量m\_angle 的值为零(0.of)，而此变量的值应该永不为零。程序员可能不知道哪个函数可能把零写入了此变量,但是程序员知道该变量的地址。要找出“肇事者”,可在变量的地址中设立数据断点,之后让程序继续执行。当该变量的值被改动时，调试器就会自动停下来。那时便可以检查调用堆栈,肇事的函数就能被捉个正着。

1. 条件断点：在“断点”窗口中,任何种类的断点(数据断点或常规的代码行断点)都可以设置条件(condition)和命中次数(hit count)。

条件断点(conditional breakpoint)使调试器在每次断点被命中时,都对输入的C/C++表达式进行评估。若表达式为真(true)，调试器会暂停程序,让用户检查程序运行情况。若表达式为假(false),调试器会忽略断点,继续运行程序。这个功能可以用来建立一种断点,当一个函数被某个实例调用时才触发。例如,假设一个游戏关卡中有20辆坦克,程序员希望第3辆坦克被调用时暂停程序。如果第3辆坦克的内存地址为0x12345678，便可以设置断点条件为“(uintptr\_t)this == 0x12345678”，以限制断点只命中第3辆坦克实体。

另外，设置命中次数之后,调试器每次命中时就会令该计数器减1,直至计数器变为零才触发暂停程序。这种设置对位于循环中的断点很有用。例如,要检查循环中第376个迭代的情况(例如数组中的第376个元素),不可能人工慢慢地去按F5键375次!但你可以设置命中次数,让 Visual Studio自动完成任务。

有一点要小心,如果设定了条件断点,那么每次命中时调试器都要对表达式取值,因而会减慢调试器和游戏的运行速度。

**调试已优化的生成**：之前提及,使用发布版本调试问题是非常棘手的,这主要缘于编译器优化代码的方式。在理想情况下,程序员都宁愿选用调试生成进行调试。然而,这经常不可行。有时候,一个bug重现率很低，当bug出现时就要好好把握调试机会，甚至有时候某个bug只出现在别人机器的发布生成中。另外,一些bug只出现在发布生成中,在调试生成中又会神奇地消失。这些只出现在发布生成的可怕bug,有时候是由未初始化变量造成的。因为在调试模式中变量和动态分配的内存通常会被设为零，而在发布模式中这些内存没有初始化,内容为随机值。bug只出现在发布生成版本的其他常见原因包括,在发布生成中意外地略去一些代码(例如,一些重要代码被错误地放进断言语句),数据结构的大小或数据成员打包方式在调试和发布模式间有差异,内联或编译器引进的优化触发bug, (罕见情况下)编译器的优化器本身的bug,也会导致在发布生成中产生错误的代码。

显然,每位程序员都必须有调试发布生成的能力,即使这看上去并不是一件轻松的事情。减轻调试优化代码之痛,最佳办法是多练习,并且在有机会时扩展这方面的技能。以下是一些窍门。

1. 学习在调试器中阅读及单步执行反汇编(disassembly):在发布生成中,调试器经常不能正确地显示目前正在执行的代码行。“感谢”指令乱序,在源代码检示模式中,经常会看见程序计数器不规律地在函数内游走。然而,若使用反汇编检示模式,则一切都变得正常(即能逐条执行汇编语言指令)。每位C/C+＋程序员都应该稍稍了解目标平台的架构和汇编语言。如能这样,即使调试器受迷惑了,程序员也不会被迷惑。
2. 运用寄存器去推理变量的值或地址:有时候,调试器不能在发布生成中正确显示变量的值或对象的内容。但是,如果程序计数器距离变量的初次使用不远,那么有很大机会该变量的值或地址仍然存于其中一个CPU寄存器里。若可以向前追踪反汇编,找到变量第一次载入寄存器的位置,就可以不断检查该寄存器以得知变量的值或地址。可使用寄存器窗口,或在监视窗口键入寄存器名字,检查寄存器内容。
3. 使用地址去检查变量及对象内容:知道变量或数据结构的地址,就可以在监视窗口中转换为适当的类别去检查其内容。例如,一个Foo类的实体位于0x1378AOCo,就可在监视窗口键入“(Foo\*)0x1378A0co”,那么调试器就会诠释该地址为指向一个Foo对象的指针。
4. 利用静态和全局变量:就算经过优化的生成版本,调试器通常也能够检查静态和全局变量。若不能推算出一个变量或对象的地址,则可以看看可能直接或间接地存有该地址的静态和全局变量。例如,若要找一个物理系统内部变量的地址,可能会发现它是存于PhysicsWorld全局变量中的一个成员变量。
5. 修改代码:若想相对简单地重现一个只出现在发布生成版本中的 bug,可考虑修改源代码以协助调试问题。增加打印语句去显示情况,引入全局变量使在调试器里检查问题变量或对象更容易,加入代码以检测问题状态,加入代码去孤立一个类的某个实例。

**2.3-剖析工具**

游戏通常是高性能的实时系统。因此，游戏程序员经常要寻求加速代码执行的方法。有一个尽管不太科学但很有用的经验法则,称为帕累托法则(Pareto principle)'。此法则指出,在很多情况下，一些事件 80%的后果只取决于20%的原因,所以它又被称为80-20规则(80-20rule)。计算机科学常使用此法则的变种,称为90-10规则(90-10 rule),指任何程序的90%挂钟时间(wall clock time)2消耗在运行仅10%的代码上。换句话说,优化那10%的代码,带来的总体运行速度提升可达完全优化的90%。

使用剖析器可以找到那需要着重优化的代码。剖析器能度量代码的执行时间，并能告知每个函数所花的时间。这些数据可以引导程序员去优化占“狮子份额”执行时间的函数。

一些剖析器也能告之每个函数的调用次数,这也是一个必须了解的重要数据。某个函数耗用大量时间,原因有二: (a)运行函数本身需很长时间, (b)函数被频繁调用。例如,一个函数运行A\*算法2搜寻游戏世界里的最优路径,每帧可能执行的数次，而函数本身就需要花较多的时间。另一方面,一个计算点积(dot product)的函数,运行可能只需几个时钟周期,但每帧此函数可能会被调用数十万次,从而拖慢游戏的帧率(frame rate)。

若使用恰当的剖析器,则能进一步获取更多信息。一些剖析工具能报告调用图(call graph),可告之某函数被哪些函数调用(称为父函数, parent function),以及该函数调用了哪些函数(称为子函数, child function,或后代, descendant)。甚至可以知道函数的运行时间花在后代的百分比,以及每个函数占整体运行时间的百分比。

剖析器大致可分为两类。

1. 统计式剖析器(statistical profiler):此类剖析器是不唐突的(unobtrusive),意指启动剖析器后,目标代码的执行速度差不多和没使用剖析器时相同。这些剖析器的原理是,周期性地为CPU的程序计数器寄存器采样,并以此获得正在执行的函数。由每个函数的采样数目,可计算出该函数占整体执行时间的近似百分比。对于运行于Pentium机器上的Windows平台,Intel的 VTune软件是统计式剖析器中的不二之选,现在还提供了Linux版本。
2. 测控式剖析器(instrumental profiler):此类剖析器能提供最精确、最详尽的计时数据,但是却要以不能实时运行程序为代价——当启动剖析器后,目标程序慢如蜗牛。此类剖析器必须预处理可执行文件,为其中每个函数安插特殊的初构代码(prologue code)和终解代码(epilogue code)。初构代码和终解代码会调用剖析器的库,调查程序的堆栈并记录所有细节,包括调用该函数的父函数、父函数调用子函数的次数。此类剖析器甚至可以设定监察每―行源代码,告之执行每行代码所花的时间。这些剖析结果极精准和详细,可是启动剖析器会令游戏慢得几乎无法玩。IBM的 Rational Quantify软件4(Rational Purify Plus工具套装之一)是一个优秀的测控式剖析器。

微软也发行了混合这两种类型的剖析器,名为LOP,代表低开销剖析器(Low OverheadProfiler)。它使用统计方法,周期性地对处理器的状态采样，对程序速度的影响很小。但是,每个采样都会剖析调用堆栈,从而找出每个采样的一连串父函数。因此, LOP提供一般统计式剖析器无法提供的数据,例如被父函数调用的分布。

**2.4-内存泄漏和损坏检测**

困扰C/C++程序员的另外两个问题是内存泄漏(memory leak)和内存损坏(memory corruption)。如果一块内存在分配后永不释放,就会产生内存泄漏。泄漏会浪费内存,最终造成致命性的内存不足(out of memory)。内存损坏则指,程序不慎把数据写进内存的错误位置,覆盖了该位置原来的重要数据,同时也未能把数据写到应该写的位置。两个问题皆可毫不含糊地归咎于同一个语言特征——指针(pointer)。

指针是强大的工具,用得其所固然有益,但也很易化益为弊。若指针指向已释放的内存,或者指针被意外地赋值为非零整数或浮点数,那么这些指针就化为内存损坏的危险工具,因为数据最终会被写入几乎任何地方。同样,若用指针来跟踪已分配的内存,也极容易在用完后忘记释放内存,导致内存泄漏。

避免因指针问题造成内存泄漏的方法之一,是养成良好的编程习惯。这种习惯无疑可以确保编写理论上永不发生内存损坏和内存泄漏的可靠代码。当然,帮助检测内存泄漏和内存损坏的工具也必不可少。幸运的是,现在有许多这类现成的工具。

**2.5-其他工具**

1. 区别工具(difference/diff tool):区别工具用来比较一个文本文档的两个版本,找出版本之间的差异(关于区别工具的详细讨论可参见维基1)。区别工具通常以行为基准计算区别,但一些新的区别工具也能显示一行之中改动了的字符。多数版本控制系统都附带一个区别工具。有些程序员如对区别工具有偏好,可在版本控制系统里自行设置。流行的区别工具有ExamDiff2 、AraxisMerge3、WinDif(能在 Windows 的一些Option Pack 及一些独立网站找到)、GNU区别工具包4。

2. 三路合并工具(three-way merge tool):当两人修改同一个文件时,就会产生两组区别。能把两组区别合并成含二人改动的最终文件的工具,称为三路合并工具。“三路”是指合并事实上使用的3个版本―—原版本、用户A的版本、用户B的版本。(维基有关于二路和三路合并技术的讨论)。很多合并工具附有连带的区别工具。流行的合并工具有AraxisMerge 、WinMerge6。Perforce也附有杰出的三路合并工具。

3. 十六进制编辑器(hex editor):十六进制编辑器用于查看及修改二进制文件的内容。数据通常以十六进制整数显示,因而得名。大部分好用的十六进制编辑器,能把数据以整数(1~16字节)、浮点数(32位或64位)、文本(ASCII)方式显示。十六进制编辑器在追查二进制文件格式问题,或对未知的二进制格式文件做反向工程时特别有用。而这两种工作在游戏引擎开发圈里比较常见。市面上有数不尽的十六进制编辑器,笔者使用Expert CommercialSoftware公司的HexEdit8，读者的选择也许不同。