Kittie.Concept 手册

滇狐
edyfox@sohu.com
version: 0.02.8088

February 19, 2006

摘要

本文介绍了一个灵活而可扩展的工程文件转换工具 Kittie.Concept，给出该软件的详细使用说明，简要叙述了该软件的设计原理与架构，并描述了该软件文件输入输出插件的开发方法。另外，本文通过 Kittie.Concept 阐述了一个支持二次开发，可以通过插件扩展功能的软件系统的设计方案和注意事项，为二次开发接口的定义提供了参考。

1 概述

任何一个软件项目，除了初学编程的人编写的小程序外，都由多个源代码组成的。在生成目标程序的时候，需要将源代码逐一编译、链接。编译、链接的过程需要调用多个程序，提供复杂的命令行，这些操作让程序员逐一手工完成是不现实的。因此任何一个集成开发环境（IDE）里，都会提供一个工程管理工具。

然而，目前每个集成开发环境里都有自己的工程管理工具，而每个工程管理工具都生成自己特有的工程文件，如 VC6 的工程文件是 .dsp 格式，VC7 的工程文件是 .vcproj 格式，而 MinGW Developer Studio 的工程文件是 .msp 格式。当将已有工程从一个集成环境迁移到另外一个集成环境下的时候，已有的工程文件往往不能被新的集成环境所识别，需要重新创建工程文件，并重新手工添加文件。对于一个拥有很多文件的大工程来说，这一过程是相当烦琐的。

Kittie.Concept 可以读入多种类型的工程文件，生成多种类型的工程文件，因此可用来在各种不同类型的工程文件之间进行转换，为项目的迁移带来了方便。Kittie.Concept 使用插件的形式读入和写出工程文件，可以通过安装新的插件的形式，使该软件能够识别更多的文件格式，不断扩展软件的功能。

2 工作原理与局限

现实世界中存在着多种多样的工程文件，如果我们为每一种工程文件到另外一种工程文件都编写一个转换程序的话，工作量将大到无法估量的地步。为了减少工作量，我们首先定义一种工程文件的中间表现形式。读入工程文件时，将工程文件翻译为中间表示形式，然后再将中间表示形式翻译为输出文件。

如图 1 所示，当有 4 种文件类型需要互相转换时，如果直接两两进行转换，需要用 12 种不同的转换程序才能完成。而如果引入一种中间表示法的话，只需要 4 种输入程序，4 种输出程序，一共 8 种程序就可以实现目标了。当文件格式的数量很大的时候，引入中间表示形式后可以节省大量的工作量。

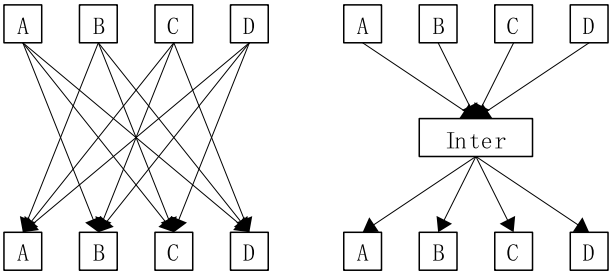

图 1: 文件格式转换

然而，引入了中间表现形式后，也带来了一个比较严重的问题：不同的编译器，不同的集成开发环境下，工程文件所支持的选项是大不一样的。定义一个包揽一切可能选项的工程文件中间表示形式，几乎是一件不可能的事情。

在 Kittie.Concept 中，工程文件的中间表示形式采用的是与 .kmk 格式[1]对等的形式[2]，而 .kmk 是针对 gcc 编译器设计的，因此对于使用 gcc 作为编译器的集成环境，Kittie.Concept 都能获得较好的兼容性。然而，对于其它编译器，如果用到了太多复杂的特性的话，就可能会出问题。例如，Kittie.Concept 在读入 Visual C++ 工程的时候，就无法无法保留“使用预编译头”的信息，因为 gcc 中并没有与之对应的命令行选项。因此，有时候用户可能需要对 Kittie.Concept 输出的文件再进行一些手工调整。幸运的是，一般情况下，普通用户都不会用到太多太古怪的编译器参数，因此需要手工做的调整都不会太大。另外，文件列表、自定义编译选项等都能够正确地转换，已经能够为程序员节约大量的工作量了。

3 安装说明

由于 Kittie.Concept 原本就是为 gcc 设计的，因此在安装 Kittie.Concept 之前，最好先安装一份 gcc。如果在 Windows 下使用的话，推荐安装 MinGW。

3.1 编译生成安装包

从 1.00.8109 版开始，Kittie.Concept 不再发布可执行文件，均以源代码包的方式发布，安装之前必须先编译。Kittie.Concept 是一个面向开发人员的工具，因此假设 Kittie.Concept 的用户都知道编译程序的基本步骤。

目前 Kittie.Concept 仅支持 Win32、x86 Linux、FreeBSD 和 SunOS。在上述未列出的平台下，作者不保证 Kittie.Concept 能够正常编译并运行，而且目前有部分插件明确知道无法在 MacOS 下正常执行。如果你在自己的平台下编译 Kittie.Concept 出现问题，请与作者联系。

首先，进入 `kittiedist` 文件夹，选择一个与你的平台最接近的平台，运行“`./build.<platform> CFG=Release`”编译脚本。注意：Win32 下的编译脚本是 `build.bat`，而不是 `build.win32`，这是由Windows 的特点决定的。例如，如果你希望在非 x86 体系的 Linux 系统下编译 Kittie.Concept，你可以运行 x86 Linux 的编译脚本 `./build.x86_linux`。如果编译失败或有什么其它问题，请与作者联系。

[1] 关于 .kmk 格式的具体信息，请参看 in_kmk 插件的文档。

[2] 目前有一处例外：.kmk 格式在描述自定义编译选项时，仅支持单个输出文件名，但 Kittie.Concept 的内部表示形式支持多个输出文件名。

Win32 下按以上步骤编译 Kittie.Concept 需要 MinGW 的支持，如果你没有安装 MinGW，你也可以使用 Visual C++ 7.1 编译它（Visual Studio 2003）。`kittiedist/makefile.win32` 下有为 Visual C++ 7.1 生成的工程文件，双击打开 kittie.sln 并编译即可。注意：要使用下文所说的安装程序生成脚本生成安装包，必须生成工程的 Release 版本。

编译成功后，到 `kittiedist` 文件夹下运行相应平台的安装包生成脚本 `./mkdist.<platform>`。注意：Win32 下的安装包生成脚本是 `mkdist.bat`，而不是 `mkdist.win32`。生成该平台下的安装程序后，参照第 3.2 节的说明安装即可。

3.2 Kittie.Concept 的安装

Kittie.Concept 的 Windows 版本无需安装，直接把 `kittiedist/win32` 文件夹下的所有文件复制到某个文件夹下，并设置一下环境变量 `%PATH%`，让 cmd.exe 能够直接找到 gcc 和 kittie 等程序即可。为了能够更方便地使用 kittie，使用 Windows 的朋友还可以直接安装 MinGW Kittie 套装。

`kittiedist` 下的 `x86_linux` 文件夹、`freebsd` 和 `sunos` 文件夹下分别含有 Linux、FreeBSD 和 SunOS 下的安装程序。`install.sh` 和 `uninstall.sh` 是为具有 root 权限的用户编写的安装和卸载程序，该安装程序会将 Kittie.Concept 安装到 /usr/local 下。不具有 root 权限的用户可以使用 none root 版本的安装卸载程序：`install_none_root.sh` 和 `uninstall_none_root.sh`，该安装程序会将 Kittie.Concept 安装到用户的个人目录下（`$HOME`）。

3.3 手工安装

如果只希望使用 Kittie.Concept 而不打算参与到 Kittie.Concept 的开发中的话，可以跳过这一章直接阅读下一章。但是如果希望参与开发 Kittie.Concept 的话，则需要认真阅读本章，了解该软件中每个文件所处的路径和手工安装的方法。

在 Linux 或 FreeBSD 下，如果你具有 root 权限的话，请参看图 2 描述的路径，复制编译得到的各个文件到指定地点。其中，`/usr/local/kittie/doc/` 里存放的是软件与插件的文档，它来自各个插件的文件夹下，它不是运行软件所必须的部分。关于图中的 `kmake`、`buildtags` 和 `kittiemake`，请参看第 4.2 节的详细解释。

如果你对软件默认插件、语言包的搜索路径（`/usr/local/share/kittie`）不满意，想把软件安装到其它地方的话，需要对软件的工程文件进行相应修改并重新编译安装。

如果你不具有 root 权限，请参看图 3 描述的路径，复制编译得到的各个文件到指定地点。

kittie 加载插件和语言包时，个人主目录会被优先搜索。如果在主目录下找到了所需文件，它将不会再去搜索公共目录下的文件。

在 Windows 下安装，可以任意创建一个目录作为 kittie 的安装目录，然后参看图 3 描述的路径，复制编译得到的各个文件到指定地点。装完毕后的目录结构应该如图 4 所示。安装完毕后还需要设置一下环境变量 `%PATH%`，让 cmd.exe 能够直接找到 gcc 和 kittie 等程序。

当然，如果愿意的话，在 Windows 下也可以把 kittie 的插件和语言包安装到个人主目录中，安装方法和 Linux 或 FreeBSD 相同。默认情况下，主目录的路径是 Windows 安装分区下的“`\Documents and Settings\<用户名>\`”。

关于图中的 `kmake.exe`、`buildtags.bat` 和 `kittiemake.bat`，请参看第 4.2 节的详细解释。

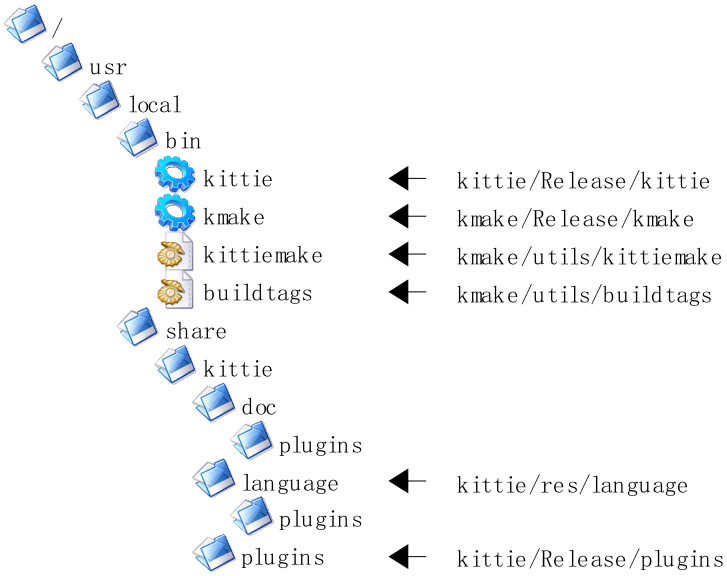

图 2: Linux 与 FreeBSD 下安装路径示意图

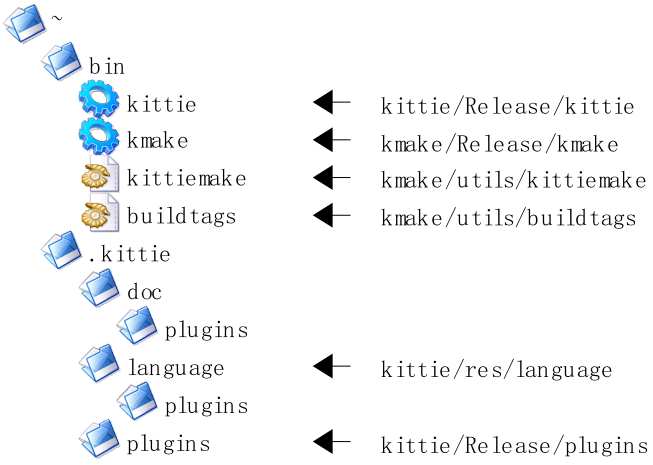

图 3: Linux 与 FreeBSD 下非 root 用户安装路径示意图

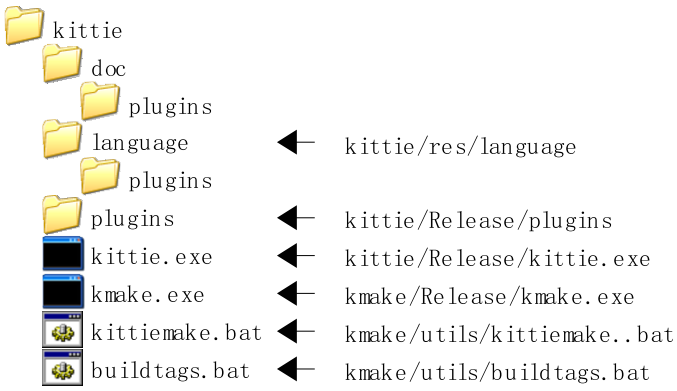

图 4: Windows 下安装路径示意图

3.4 开发者模式安装

在 Linux 或 FreeBSD 下，作为一个 Kittie.Concept 开发人员，如果每次编译完毕插件后都需要重新重复一遍上述安装步骤，未免显得过于烦琐。你可以通过编写脚本来简化这一安装过程，但使用符号链接，则可以进一步减少工作量。在 `/usr/local/bin` 下创建符号链接`kittie`、`kmake`、`kittiemake` 和 `buildtags`，分别指向 `kittie/Release` 下的 `kittie`、`kmake/Release` 下的 `kmake`、`kmake/utils` 下的 `kittiemake` 和 `buildtags`；在 `/usr/local/share` 下创建目录 `kittie`，在 `kittie` 目录下创建符号链接 `plugins` 和 `language`，分别指向目录 `kittie/Release/plugins` 和 `kittie/res/language`。这样，每次修改语言包或重新编译程序后，就不必重新复制安装相应文件了。

Windows 平台下虽然不支持符号链接，但由于 kittie 的 Windows 版本使用相对与 `kittie.exe` 的路径查找插件与语言包，因此允许同一台机器中多份不同版本的 kittie 共存，调试起来也没有太大问题。

4 使用说明

Kittie.Concept 可以解析多种类型的工程文件，生成多种类型的文件。各种文件格式插件支持的选项都各不相同，具体插件的命令行参数请参看插件附带的文档。

4.1 命令行参数说明

使用 kittie 的命令行如下：

`kittie [附加选项] [--|-i] <文件名> [[-o] <文件名>]`

kittie 将“`--`”或“`-i`”之后出现的参数看做输入文件名，将“`-o`”之后出现的参数看做输出文件名。如果用户没有指定“`--`”、“`-i`”、“`-o`”选项的话，kittie 使用出现的第一个文件名作为输入文件名，将第二个文件名作为输出文件名。

默认情况下，kittie 使用文件名的扩展名作为插件名，加载相应插件处理文件。如果想要采用与扩展名不同的插件名处理文件的话，可以使用“`-I <输入插件名>`”或“`-O <输出文件名>`”指定插件名。

如果指定了插件名，但没有指定文件名，kittie 会向插件询问插件支持的默认文件名。一般输入插件的默认文件名是标准输入，输出插件的默认文件名

是标准输出[3]。

如果插件名和文件名均没有指定，则 kittie 使用 kmk 作为默认输入插件，读取标准输入，使用 mak 作为默认输出插件，写至 `Makefile`。

命令行中所有“`-p`”打头的项，都会作为输入插件参数，传递给输入插件，而所有“`-g`”打头的项，都会作为输出插件参数，传递给输出插件[4]。例如，如果命令行中包含“`-pplatform=win32 -grough`”，则输入插件会收到“`platform=win32`”，输出插件会收到“`rough`”。每个插件支持的命令行选项都不一样，可以查阅具体插件附带的文档获取更多信息。

另外，可以使用以下命令行格式查询指定插件的帮助信息或版本信息：

```
kittie [-I 输入插件] [-O 输出插件] <--help|--version>
```

4.2 与外部工具的整合

虽然 Kittie.Concept 可以处理多种类型的工程文件，但实际使用中，它的主要用途还是用来生成 Makefile 并调用 gcc 编译工程。kmake、kittiemake 可以简化使用 kittie 编译工程的工作。

kmake 的命令格式如下：

```
kmake [传递给 kittie 的参数] <文件名> [传递给 make 的参数]
```

kmake 的作用是，首先将工作目录切换到被编译的文件所处的目录，然后调用 kittie，将指定的文件编译为 .mak 文件，再调用 make[5] 编译生成目标文件，最后切换回原路径。

例如，当在命令行调用“`kmake -I kmk ftplite.vc CFG=Release`”的时候，kmake 会首先调用“`kittie -I kmk ftplite.vc ftplite.mak`”，将输入文件转换为 Makefile，再调用“`make -f ftplite.mak CFG=Release`”，编译该工程。kmake 为在浏览器中直接将 kittie 和 make 整合到 .kmk 右键菜单中提供了可能。

kmake 并不要求输入文件格式必须是 .kmk，在 Windows 平台下，使用 MinGW 作为编译器的时候，我们可以使用 kmake 直接编译 VC 的工程文件：“`kmake slue.vcproj`”。

buildtags 的作用是，如果一个目录下存在 `ORDER` 文件，则将 `ORDER` 文件中指定的所有工程文件内包含的文件列表导出到 `filelist.txt` 中；如果不存在 `ORDER` 文件，则将当前目录下的所有 .kmk 文件中包含的文件列表导出到 `filelist.txt` 中。获取到文件列表后，调用 ctags 生成 tags 文件。得到 tags 文件后，在支持 tags 的文本编辑器如 VIM 里就能方便地浏览代码了。

kittiemake 的作用是，如果一个目录下存在 `ORDER` 文件，则对 `ORDER` 文件中指定的每个工程文件调用 kmake 进行编译；如果不存在 `ORDER` 文件，则对当前目录下的所有 .kmk 文件调用 kmake 进行编译。编译完毕后，为编译过的所有工程创建 tags 文件。当一个目录下存在多个工程文件间存在依赖关系，必须按照一定次序编译时，或者只想编译当前目录下的部分工程文件时，使用 `ORDER` 文件可以获得很大的方便。

Kittie.Concept 的前身 kittie 0 原本就是为 VIM 设计的，因此它与 VIM 的配合非常好。只要设置编译器为 kittiemake，就可以直接享受到VIM 的全套便利：

```
:set makeprg=kittiemake
```

这样，需要编译工程的时候，只需要直接输入：

```
:make
```

或者：

[3] .mak 格式的默认输出文件名是 Makefile，为了保证与 kittie 0 的兼容性。

[4] “p”代表“parser”，“g”代表“generator”。

[5] 在 FreeBSD 下，kmake 调用 gmake。

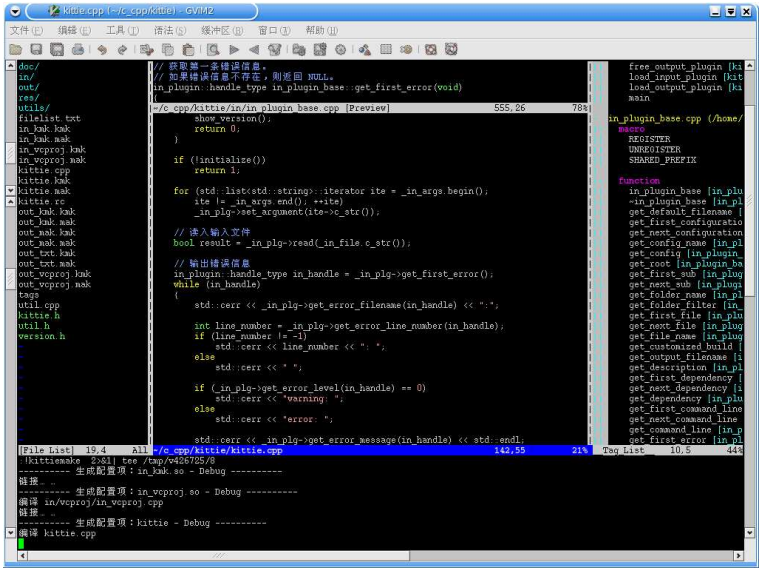

图 5: Kittie.Concept 在 VIM 下工作

```
:make CFG=Release
```

等命令，然后回车就可以了。VIM 能够识别 kittie 输出的出错信息，当 .kmk 文件中存在语法错误时，VIM 能够自动定位到出错行。至于 gcc 输出的错误信息，VIM 一向都支持得很好的，任何地方存在编译错误，都能够很快地定位到出错位置。

在 Windows 下使用VIM 编译程序时，make 的过程被导入到临时文件中，不在屏幕上显示，这带来了极大的不便。为了在 make 的过程中能够看清输出信息，不至于等得太心急，最好再安装一个 tee.exe。将 tee.exe 放到 Windows 目录，或某个 %PATH% 能够找到的地方，然后修改 _vimrc，增加一句：

```
:set shellpipe=2>&1\|\ tee
```

这样编译的时候就舒服多了。

在 VIM 中安装了足够的插件后，可以得到一个非常强大而方便的开发环境。有兴趣的朋友可以参看 VIM 相关网站。在 VIM 中调用 kittie 编译文件的效果如图 5 所示。

5 插件开发指南

Kittie.Concept 是一个可以通过插件开发扩展功能的软件，插件的设计方式，使得插件的开发可以做到与具体编译器无关，甚至可以做到语言无关。

5.1 插件接口设计

无论是 Windows、Linux 还是 FreeBSD，都支持动态链接库。动态链接库可以在软件运行的时候，通过文件名动态加载到内存中，然后运行动态链接库中提供的函数，为插件的设计提供了可能。

对于一个只需导出两三个函数的小型插件而言，直接将插件中用到的函数写成动态库导出函数，便可以很方便地实现一个插件。但是，如果需要导出数十个，甚至上百个函数的话，直接写成动态库导出函数的话，在使用插件的

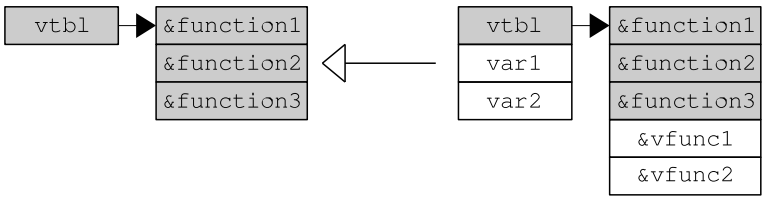

图 6: 接口类示意图

时候一一获取导出函数的地址，将会成为程序员的一场噩梦。为了节省工作量，提高插件的可用性，我们仅在插件中导出一个函数，在该函数中创建一个 C++ 类，并把类的指针返回给用户，用户通过这个指针调用类里提供的方法。这样就可以仅导出一个函数，就为用户提供数十个方法。

对于同一个编译器，使用同一套运行库，直接导出 C++ 类既简单又方便。然而，如果插件的开发者与软件的开发者使用的编译器版本不同，或者使用的运行库不同，亦或两者都不同，导出普通的 C++ 类很可能导致软件不能识别，会出现很严重的问题。为了避免这些问题的出现，我们在导出 C++ 类的时候，只导出继承自接口类的实现。

接口类是指不含有任何数据成员，所有成员函数均为纯虚函数的类。对接口类的所有访问，都必须通过接口类的方法进行。如图 6 所示，当继承自接口类时，所有数据成员都被存放到 vtable 之后，而且不会被调用者直接访问到，因此不同编译器的不同实现不会导致数据成员无法识别。同理，为了避免不同编译器之间运行库的不同，接口类的成员方法不应该接受或返回任何非基本类型，如 `std::string` 等都是不允许的。

由于不同编译器对析构函数的处理有细节上的不同，因此接口类对象使用完毕后，还需要通过动态库导出的另一个函数释放它占有的内存。

Kittie.Concept 的文件输入插件导出以下两个采取 C 链接方式[6] 的函数：

`in_plugin *DECORATOR create_entity(void)`：创建输入插件的对象实例。

`void DECORATOR release_entity(in_plugin *plg)`：释放该输入插件的对象实例。

Kittie.Concept 的文件输出插件导出以下两个采取 C 链接方式的函数：

`out_plugin *DECORATOR create_entity(void)`：创建输出插件的对象实例。

`void DECORATOR release_entity(out_plugin *plg)`：释放该输出插件的对象实例。

5.2 插件的编写

文件输入插件需要导出上文所述的两个函数，并实现 `in_plugin` 类，需要导出的函数列表与功能请参看附录 A。

为了保证移植性，`in_plugin` 类的接口显得多而复杂。为了降低插件编写难度，减小工作量，如果采用 C++ 编译器编写插件的话，可以让插件继承自 `in_plugin_base`。该类的详细介绍请参看附录 C。

文件输入插件需要导出上文所述的两个函数，并实现 `out_plugin` 类，需要导出的函数列表与功能请参看附录 B。

[6] 即把函数声明为 `extern "C"`，以阻止 C++ 为该函数创建修饰名。如果在 Windows 下，还需要用 .def 文件去掉 `__stdall` 在函数名尾部带来的“`@0`”、“`@4`”标记。

为了降低插件编写难度，减小工作量，如果采用 C++ 编译器编写插件的话，可以让插件继承自 `out_plugin_base`。该类的详细介绍请参看附录 D。

6 综述

作为一个概念项目，Kittie.Concept 的主要目的是探索插件与二次开发接口的设计方法。然而，Kittie.Concept 作为一个独立的项目，在跨平台软件开发领域，也具有自己独立的用途与应用前景。

参考文献

[1] MSDN Library, http://msdn.microsoft.com

[2] Linux man pages

[3] Dale Rogerson. COM 技术内幕：微软组件对象模型, *Inside COM: Microsoft's Component Object Model*, 杨秀章. 清华大学出版社, 1999

[4] Stan Lippman. 深度探索 C++ 对象模型, *Inside The C++ Object Model*, 侯捷. 华中科技大学出版社, 2001

[5] Martin D. Carroll, Margaret A. Ellis. C++ 代码设计与重用, *Designing and Coding Reusable C++*, 陈伟柱. 人民邮电出版社, 2002

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 设计模式：可复用面向对象软件的基础, *Design Patterns: Elements of Reusable Object-Oriented software*, 李英军, 马晓星, 蔡敏, 刘建中. 机械工业出版社, 2000

A in_plugin 接口列表

该插件中所有 const char *const 类型的参数，都由调用者（用户）负责释放空间，插件不必、也不能释放该空间。

该插件中所有返回值为 const char *const 的函数，返回的指针指向内存空间都由插件负责释放，用户不必、也不能释放该空间。一般的做法是，当用户下一次调用返回值类型为 const char *const 的函数的时候，插件释放前一次返回时分配的空间，前一次返回的指针变为无效。当释放插件的时候，释放最后一次为返回值分配的空间。

用户应该即时将返回值复制到自己的内存块中，以防指针失效造成未定义的结果，带来难以调试的错误。

A.1 类型定义

handle_type：输入插件使用的句柄类型。

A.2 插件工作接口

bool DECORATOR read(const char *const filename)：读取并分析指定文件。如果文件分析成功，则该函数返回 true，否则返回 false。

void DECORATOR set_argument(const char *const arg)：为插件设置命令行参数。kittie 从命令行取得参数之后，将所有以“-p”开头的参数记录下来，并调用输入插件的这个函数将命令行参数传递给插件，传递给插件的参数中不包含“-p”前缀。例如，当用户命令行中包含“-pplatform=win32”时，输入插件将会收到“platform=win32”参数。

A.3 插件信息读取接口

const char *const DECORATOR get_default_filename(void)：返回该插件读取的默认输入文件名。如果用户在 kittie 的命令行中未提供输入文件名，则 kittie 调用该函数获取默认输入文件名。

const char *const DECORATOR get_help_message(void)：获取插件的帮助信息，用于屏幕显示。

long DECORATOR get_version(void)：获取插件的版本号。版本号为4 字节数据，第一字节为主版本号，第二字节为次版本号。后两字节为修正号，按 Big Endian 方式排列。

A.4 配置读取接口

handle_type DECORATOR get_first_configuration(void)：获取第一个配置的句柄。如果配置不存在，则该函数返回 NULL。

handle_type DECORATOR get_next_configuration(handle_type ch)：获取下一个配置项的句柄。如果下一个配置项不存在，则该函数返回 NULL。

const char *const DECORATOR get_config_name(handle_type ch)：获取配置名称。

const char *const DECORATOR get_config(handle_type ch, const char *const name)：获取配置中配置项的取值。关于配置项的详细信息，请参看附录 C.2。

A.5 文件树信息读取接口

handle_type DECORATOR get_root(void)：获取文件树树根的句柄。

handle_type DECORATOR get_first_sub(handle_type th, handle_type * info)：获取指定句柄下的第一棵子树，返回用于继续枚举子树的句柄。如果子树不存在，则返回 NULL。

handle_type DECORATOR get_next_sub(handle_type th, handle_type ih, handle_type *info)：获取指定句柄下的下一棵子树，返回用于继续枚举子树的句柄。如果子树不存在，则返回 NULL。

const char *const DECORATOR get_folder_name(handle_type info)：获取指定节点的名称。

const char *const DECORATOR get_folder_filter(handle_type info)：获取指定节点的扩展名筛选项。

handle_type DECORATOR get_first_file(handle_type th, handle_type * info)：获取指定句柄下的第一片叶子，返回用于继续枚举叶子的句柄。如果叶子不存在，则返回 NULL。

handle_type DECORATOR get_next_file(handle_type th, handle_type ih, handle_type *info)：获取指定句柄下的下一棵叶子，返回用于继续枚举叶子的句柄。如果叶子不存在，则返回 NULL。

const char *const DECORATOR get_file_name(handle_type info)：获取叶子节点的名称。

handle_type DECORATOR get_customized_build(handle_type info)：获取自定义编译指令的句柄。如果不存在自定义编译指令，则该函数返回 NULL。

handle_type DECORATOR get_first_output(handle_type dh)：获取自定义编译指令中第一个输出文件的句柄。

handle_type DECORATOR get_next_output(handle_type ch, handle_type dh)：获取自定义编译指令中下一个输出文件的句柄。

const char *const DECORATOR get_output_filename(handle_type cbh)：获取自定义编译指令的输出文件名。

const char *const DECORATOR get_description(handle_type cbh)：获取自定义编译指令的说明文字。

handle_type DECORATOR get_first_dependency(handle_type dh)：获取自定义编译指令中第一个依赖项的句柄。

handle_type DECORATOR get_next_dependency(handle_type ch, handle_type dh)：获取自定义编译指令中下一个依赖项的句柄。

const char *const DECORATOR get_dependency(handle_type dh)：获取自定义编译指令中依赖项。

handle_type DECORATOR get_first_command_line(handle_type dh)：获取第一个命令行的句柄。

handle_type DECORATOR get_next_command_line(handle_type ch, handle_type dh)：获取下一个命令行的句柄。

const char *const DECORATOR get_command_line(handle_type dh)：获取命令行。

A.6 错误信息读取接口

`handle_type DECORATOR get_first_error(void)`：获取第一条错误信息的句柄，如果错误信息不存在，则返回 `NULL`。

`handle_type DECORATOR get_next_error(handle_type eh)`：获取下一条错误信息的句柄，如果下一条错误信息不存在，则返回 `NULL`。

`const char *const DECORATOR get_error_filename(handle_type eh)`：获取错误信息对应的文件名。

`int DECORATOR get_error_line_number(handle_type eh)`：获取错误信息所处的行号。

`int DECORATOR get_error_level(handle_type eh)`：获取错误信息的级别。0 表示警告信息；1 表示错误信息。

`const char *const DECORATOR get_error_message(handle_type eh)`：获取字符串描述的错误信息。

A.7 其它

`void DECORATOR close_handle(handle_type handle)`：关闭句柄，释放相应空间。

B out_plugin 接口列表

输出插件对于 const char *const 的处理方式与输入插件相同，请参看附录 A 的详细解释。

B.1 类型定义

handle_type：输出插件使用的句柄类型。

B.2 插件工作接口

bool DECORATOR write(const char *const filename, in_plugin *plg)：输出文件到 filename，输出插件从输入插件中获取分析得到的信息。如果文件输出成功，则该函数返回 true，否则返回 false。

void DECORATOR set_argument(const char *const arg)：为插件设置命令行参数。kittie 从命令行取得参数之后，将所有以“-g”开头的参数记录下来，并调用输出插件的这个函数将命令行参数传递给插件，传递给插件的参数中不包含“-g”前缀。例如，当用户命令行中包含“-gplatform=win32”时，输入插件将会收到“platform=win32”参数。

B.3 插件信息读取接口

const char *const DECORATOR get_default_filename(void)：返回该插件读取的默认输出文件名。如果用户在 kittie 的命令行中未提供输出文件名，则 kittie 调用该函数获取默认输出文件名。

const char *const DECORATOR get_help_message(void)：获取插件的帮助信息，用于屏幕显示。

long DECORATOR get_version(void)：获取插件的版本号。版本号为4 字节数据，第一字节为主版本号，第二字节为次版本号。后两字节为修正号，按 Big Endian 方式排列。

B.4 错误信息读取接口

handle_type DECORATOR get_first_error(void)：获取第一条错误信息的句柄，如果错误信息不存在，则返回 NULL。

handle_type DECORATOR get_next_error(handle_type eh)：获取下一条错误信息的句柄，如果下一条错误信息不存在，则返回 NULL。

const char *const DECORATOR get_error_filename(handle_type eh)：获取错误信息对应的文件名。

int DECORATOR get_error_level(handle_type eh)：获取错误信息的级别。0 表示警告信息；1 表示错误信息。

const char *const DECORATOR get_error_message(handle_type eh)：获取字符串描述的错误信息。

B.5 其它

void DECORATOR close_handle(handle_type handle)：关闭句柄，释放相应空间。

C in_plugin_base 与相关类

为了保证跨编译器，in_plugin 成员函数繁多，实现难度也较大。为了降低开发成本，如果使用 C++ 编译器开发输入插件，可以继承自 in_plugin_base 类。

C.1 in_plugin_base 类

in_plugin_base 实现了配置项、树和错误信息的遍历等涉及到的烦琐的句柄操作，用户不再需要操心内存和资源的分配、释放的问题，还为自类提供了一个语言包读取接口：

```
std::string get_string(std::string key);
std::string get_string(const char *const key);
```

in_plugin_base 类会试图到软件安装路径[7] 中 language 目录下的 plugins 目录中查找名为“in_<扩展名>”的语言包。关于语言包的格式，请参看附录 C.5。

子类需要在构造函数中调用父类的构造函数，传递插件对应的文件扩展名作为参数（不包含点号），实现 read 方法并在 read 方法中填充 config、files 和 errors 变量，实现get_help_message、get_version、set_argument 方法。如果插件不打算使用标准输入作为默认的输入文件名的话，还可以重载父类的 get_default_filename 方法，返回其它文件名。

config 是一个存放 configuration 类型的双链表（std::list）；files 是一个树枝为 folder 类型，树叶为 file_item 类型的树（tree）；errors 是一个存放 error_message 类型的双链表（std::list）。下述章节将会详细介绍这些类型的使用方法。

C.2 configuration 类

configuration 类提供了以下方法：

configuration(std::string name)：构造一个名称为 name 的配置。

configuration(const configuration &other)：拷贝构造函数。

~configuration(void)：析构函数。

std::string get_name(void) const：获取配置名称。

std::string get_value(std::string name) const：获取配置项的值。如果指定的配置项不存在，则返回空字符串。

bool set_value(std::string name, std::string value)：设置配置项。如果指定的配置项不存在，则返回 false，否则返回 true。

目前 Kittie.Concept 支持的配置项有以下几种：

OBJ_DIR：编译中间文件（如 .o 文件）存放的路径。

OUTPUT_DIR：编译得到的目标文件输出路径。

TARGET：编译得到的目标文件名。

C_INCLUDE_DIRS：编译器附加包含目录。该部分的各项参数使用空格分隔，每项必须以“-I”开头，例如：“-I../euc -I/usr/local/include”。

[7] 关于语言包的搜索路径的详细解释，请参看第 3.3 节。

C_PREPROC：编译器附加宏定义。该部分的各项参数使用空格分隔，每项必须以“-D”开头，例如：“-DWIN32 -D_DEBUG”。

CFLAGS：编译器附加参数，参数间使用空格分隔，例如：“-Wall -frtti”。

RC_INCLUDE_DIRS：资源编译器附加包含目录。该部分的各项参数使用空格分隔，每项必须以“-I”开头。

RC_PREPROC：资源编译器附加宏定义。该部分的各项参数使用空格分隔，每项必须以“-D”开头。

RCFLAGS：资源编译器附加参数。

LIB_DIRS：链接时使用的附加库目录。该部分的各项参数使用空格分隔，每项必须以“-L”开头，例如：“-Llib -l/usr/local/lib”。

LIBS：链接时使用的附加库。该部分的各项参数使用空格分隔，每项必须以“-l”开头，例如：“-lpthread -ldl”。

LDFLAGS：链接器附加参数，参数间空格分隔，如：“-enable-stdcall-fixup -shared”。

C.3 tree 与相关类型

tree 类：

in_plugin_base 类中使用的 tree，是 tree<folder, file_item> 的模板特化。

folder 类：

file_item 类：

file_item(std::string name, customized_build *cb = NULL)：该类析构时会删除 cb 指针，因此用户不要删除构造函数传入的 cb 指针。

file_item(const file_item &other)：拷贝构造函数。该类的拷贝构造使用引用计数，与被拷贝对象共用 customized_build 数据。

~file_item(void)：析构函数。

std::string get_name(void) const：获取文件名。

customized_build *get_customized_build(void)：获取自定义编译选项。如果没有定义自定义编译选项的话，该方法返回 NULL。

const file_item &operator=(const file_item &other)：赋值操作符。

customized_build 类：

customized_build(std::string description = std::string())：构造函数。该类不允许拷贝构造。

~customized_build(void)：析构函数。

std::string get_description(void) const：获取自定义编译指令的用途描述。

void set_description(std::string desc)：设置自定义编译指令的用途描述。

void add_output(std::string output)：添加输出文件。

const std::list<std::string> &get_output(void) const：获取输出文件列表。

void add_command_line(std::string cmd)：添加命令行。

const std::list<std::string> &get_command_line(void) const：获取命令行列表。

void add_dependency(std::string dep)：添加附加依赖项。

const std::list<std::string> &get_dependency(void) const：获取附加依赖项。

C.4 in_plugin_base::error_message 类

in_plugin_base::error_message 提供以下方法：

error_message(void)：构造函数。

error_message(const error_message &other)：拷贝构造函数。

提供以下属性：

std::string filename：出错文件名。

int line_number：出错行号。

int error_level：错误级别。0 代表警告信息，1 代表错误信息。

std::string message：错误信息文本。

C.5 euc::language 类

euc::language 是用于读取语言包的类。语言包是一个文本文件，格式为：

<语言串名>=<语言串值>

语言包的文件名形式为：

<语言包名>[_语言名].lan

文件名中不包含“语言名”的为默认语言包，例如“in_vcproj.lan”表示名为“in_vcproj 的默认语言包”，“in_kmk_zh_CN.lan”表示名为“in_kmk 的简体中文语言包”。当加载当前区域的语言包失败的时候，就会加载默认语言包。

euc::language 类提供以下方法：

language(std::string name, const std::list<std::string> &folders, std::string def_token = std::string("<ERROR>"))：从 folders 指定的文件夹下加载前缀为 name 的语言包。如果语言包中不存在指定的字符串，则返回 def_token 指定的默认字符串。

~language(void)：析构函数。

std::string get_string(std::string name)：获取指定语言串名对应的语言串值。

std::string get_string(const char *const name)：获取指定语言串名对应的语言串值。

D out_plugin_base 与相关类

为了降低开发成本，如果使用 C++ 编译器开发输出插件，可以继承自 out_plugin_base 类。

D.1 out_plugin_base 类

out_plugin_base 实现了错误信息的遍历涉及到的句柄操作，用户不再需要操心内存和资源的分配、释放的问题，还为自类提供了一个语言包读取接口：

```
std::string get_string(std::string key);
std::string get_string(const char *const key);
```

out_plugin_base 类会试图到软件安装路径中 language 目录下的 plugins 目录中查找名为“out_<扩展名>”的语言包。关于语言包的详细资料，请参看附录 C。

子类需要在构造函数中调用父类的构造函数，传递插件对应的文件扩展名作为参数（不包含点号），实现 write 方法并在 write 方法中填充 errors 变量，实现get_help_message、get_version、set_argument 方法。如果插件不打算使用标准输出作为默认的输出文件名的话，可重载 get_default_filename 方法，返回其它文件名。

errors 是一个存放 error_message 类型的双链表（std::list）。下一节将会详细介绍该类型的使用方法。

D.2 out_plugin_base::error_message 类

out_plugin_base::error_message 提供以下方法：

error_message(void)：构造函数。

error_message(const error_message &other)：拷贝构造函数。

提供以下属性：

std::string filename：出错文件名。

int error_level：错误级别。0 代表警告信息，1 代表错误信息。

std::string message：错误信息文本。

目录