5 SWIG基础

- 运行SWIG
  - 输入格式
  - SWIG输出
  - 评论
  - C预处理器
  - SWIG指令
  - 解析器限制
- 包装简单的C声明
  - 基本类型处理
  - 全局变量
  - 常数
  - 关于const的简短说明
  - 一个关于char的警示故事*
- 指针和复杂对象
  - 简单的指针
  - 运行时指针类型检查
  - 派生类型，结构和类
  - 未定义的数据类型
  - Typedef
- 其他实用性
  - 通过价值传递结构
  - 按价值回报
  - 链接到结构变量
  - 链接到char *
  - 数组
  - 创建只读变量
  - 重命名和忽略声明
    - 简单重命名特定标识符
    - 先进的重命名支持
    - 限制全局重命名规则
    - 忽略所有内容，然后包装一些选定的符号
  - 默认/可选参数
  - 指向函数和回调的指针
- 结构和联合
  - Typedef和结构
  - 字符串和结构
  - 数组成员
  - 结构数据成员
  - C构造函数和析构函数
  - 将成员函数添加到C结构
  - 嵌套结构
  - 有关结构包装的其他注意事项
- 代码插入
  - SWIG的输出
  - 代码插入块
  - 内联代码块
  - 初始化块
- 界面构建策略
  - 为SWIG准备C程序
  - SWIG接口文件
  - 为什么要使用单独的界面文件？
  - 获取正确的头文件
  - 与main（）做什么

本章描述SWIG的基本操作，其输入文件的结构以及如何处理标准ISO C声明。下一章将介绍C ++支持。但是，C ++程序员仍应阅读本章以了解基础知识。有关每种目标语言的特定详细信息将在后面的章节中介绍。

5.1运行SWIG

要运行SWIG，请使用带有选项和以下文件名的swig命令：

```
swig [options]文件名
```

其中filename是SWIG接口文件或C / C ++头文件。通过运行swig -help可以看到完整的帮助。以下是可以使用的常见选项集。还为每种目标语言定义了其他选项。可以通过针对特定于语言*<lang>*的选项运行swig- *<lang>* -help获得完整列表，例如，针对Ruby的swig -ruby -help。

```
支持的目标语言选项
     -csharp-生成C＃包装
     器
     -d-
```

```
      生成D包装器-go- 生成Go包装器-guile-生成Guile包装器
      -java-生成Java包装器
      -javascript-生成Javascript包装器
      -lua-生成Lua包装器
      -octave-生成八度包装
      -perl5-生成Perl 5包装
      -php7-生成PHP 7包装
      -python-生成Python包装
      -r-生成R（aka GNU S）包装
      -ruby-生成Ruby包装
      -scilab-生成Scilab包装器
      -tcl8-生成Tcl 8包装器-xml-
      生成XML包装器

实验目标语言选项
      -mzscheme-生成MzScheme / Racket包装器
      -ocaml-生成OCaml包装器

常规选项
      -addextern-添加额外的外部声明
      -c ++-启用C ++处理
      -co <文件>-从SWIG库中
      检出<文件> -copyctor-尽可能自动生成副本构造函数
      -cpperraswarn-将预处理器#error语句视为#warning（默认值）
      -cppext <ext>-将生成的C ++文件的文件扩展名更改为<ext>
                          （默认为cxx）
      -版权-显示版权声明
      -debug- classes- 显示有关在接口中找到的类的信息
      -debug-module <n>-显示模块在第1-4阶段的分析树，<n>是阶段的csv列表
      -debug-symtabs-显示符号表信息
      -debug-symbols-
      在符号表中显示目标语言符号-debug-csymbols-在以下位置显示C符号符号表
      -debug-lsymbols-显示目标语言层符号-debug-tags-
      显示有关在界面中找到的标签的信息
      -debug-template-显示调试模板的信息
      -debug-top <n>-在第1-4阶段显示整个分析树，<n>是csv的阶段列表
      -debug-typedef-显示有关类型和typedef的信息接口
      -debug-typemap-显示typemap调试信息
      -debug-tmsearch-显示typemap搜索调试信息
      -debug-tmused-显示使用过的typemap 调试信息
      -director-打开所有类的Director模式，主要用于测试
      -dirprot-Turn关于导演类的受保护成员的包装（默认）
      -D <symbol>-定义符号<symbol>（用于条件编译）
      -E -预处理只，不产生包装代码
      -external运行时[文件] -导出SWIG运行栈
      -fakeversion <V> -使SWIG假的程序版本编号，以<V>
      -fcompact -编译在紧凑模式
      -features <list>-设置全局功能，其中<list>是功能的逗号分隔列表
                          ，例如-features Directors，autodoc = 1
                          如果未为该功能提供明确的值，则使用默认值1
      -fastdispatch-启用快速分派模式以产生更快的过载调度程序代码
      -Fmicrosoft-以Microsoft格式显示错误/警告消息
      -Fstandard-以常用格式显示错误/警告消息
      -fvirtual-以虚拟消除模式编译-help-
      显示帮助-I--
      不搜索当前目录
      -I <dir>-在目录<dir中查找SWIG文件>
      -忽略忽略-忽略缺少的包含文件
      -importall-将所有#include语句作为导入
      -includeall-将所有#include语句作为
      -l <ifile>-包括SWIG库文件<ifile>
      -macroerrors-报告宏内的错误
      -makedefault-创建默认的构造函数/析构函数（默认）
      -M-列出所有依赖项
      -MD-与`-M -MF <file>'等效，但不暗示`
      -E'-MF <file>-将依赖生成到<file>中并继续生成包装器
      -MM-列出依赖关系，但忽略文件SWIG库
      -MMD-与`-MD'类似，但是忽略SWIG库中的文件
      -module <名称>-将模块名称设置为<name>
      -MP-为所有依赖项生成伪造目标
      -MT <target>-设置由依赖项生成发出的规则的目标
      -nocontract-关闭合同检查
      -nocpperraswarn-不要将预处理器#error语句视为#warning
      -nodefault-不生成默认构造函数或默认析构函数
      -nodefaultctor-不生成隐式默认构造函数
```

```
     -nodefaultdtor-不生成隐式默认析构函数
     -nodirprot-不包装导演保护的成员
     -noexcept-不包装异常说明符
     -nofastdispatch-禁用快速分派模式（默认）
     -nopreprocess-跳过预处理程序步骤
     -notemplatereduce-禁用模板中的typedef缩减
     -O-启用优化选项：
                         -fastdispatch -fvirtual
     -o <outfile>-将C / C ++输出文件的名称设置为<outfile>
     -oh <headfile>-设置名称导演的C ++输出头文件到<headfile>
     -outcurrentdir-将默认输出dir设置为当前dir而不是输入文件的路径
     -outdir <dir>-将特定于语言的文件输出目录设置为<dir>
     -pcreversion-显示PCRE版本信息-small-
     在虚拟消除和紧凑模式下编译
     -swiglib -报告SWIG库的位置并退出
     -templatereduce-减少模板中的所有typedef -v-
     以详细模式运行
     -version-显示SWIG版本号
     -Wall-删除所有警告抑制，还意味着-Wextra
     -Wallkw-启用关键字警告所有受支持的语言
     -Werror-将警告视为错误
     -Wextra-添加以下附加警告：202,309,403,405,512,321,322
     -w <列表>-禁止/添加警告消息，例如-w401，+
     321- 请参见Warnings.html -xmlout <文件>-将解析树的XML版本写入<文件>正常处理后
```

参数也可以在命令行选项文件（也称为响应文件）中传递，如果它们超过了系统命令行的长度限制，则很有用。为此，请将参数放在文件中，然后提供带有@前缀的文件名，如下所示：

```
file @ 文件
```

从文件读取的选项将代替文件选项插入。如果该文件不存在或无法读取，则该选项将按字面意义处理且不会被删除。

文件中的选项用空格分隔。通过将整个选项括在单引号或双引号中，可以在选项中包括空格字符。通过在要包含的字符前面加上反斜杠，可以包含任何字符（包括反斜杠）。该文件本身可能包含其他@file选项；任何此类选项将被递归处理。

5.1.1输入格式

作为输入，SWIG需要一个包含ISO C / C ++声明和特殊SWIG指令的文件。通常，这是一个特殊的SWIG接口文件，通常以特殊的.i或.swg后缀表示。在某些情况下，SWIG可以直接用于原始头文件或源文件。但是，这不是最典型的情况，有多个原因可能导致您不希望这样做（稍后说明）。

SWIG接口的最常见格式如下：

```
%module mymodule
%{
#include" myheader.h"
%}
//现在列出ISO C / C ++声明
int foo;
int bar（int x）;
...
```

使用特殊的%module 指令提供模块名称。在“ 模块介绍”部分中进一步描述了模块。

在一切%{...%}块被简单地逐字复制到由SWIG创建成品包装纸文件。该部分几乎总是用于包含头文件和其他使编译的包装器代码编译所需的声明。需要强调的是，只是因为您在一大口输入文件中的声明，该声明也很重要，*不会*自动出现在生成的包装代码---因此，你需要确保你包括在适当的头文件%{.. 。%}部分。应当注意，SWIG不会解析或解释%{...%}中包含的文本。在%{...%} SWIG中的语法和语义类似于解析器生成工具（例如yacc或bison）的输入文件中使用的声明部分。

5.1.2 SWIG输出

SWIG的输出是一个C / C ++文件，其中包含构建扩展模块所需的所有包装器代码。SWIG可能会根据目标语言生成一些其他文件。默认情况下，名称为file.i的输入文件将转换为file_wrap.c或file_wrap.cxx文件 （取决于是否使用了-c ++选项）。可以使用-o选项更改输出C / C ++文件的名称。在某些情况下，编译器使用文件后缀来确定源语言（C，C ++等）。因此，如果您想要不同于默认值的内容，则必须使用 -o选项更改SWIG生成的包装文件的后缀。例如：

```
$ swig -c ++ -python -o example_wrap.cpp example.i
```

SWIG创建的C / C ++输出文件通常包含为目标脚本语言构建扩展模块所需的所有内容。SWIG不是存根编译器，通常也不需要编辑输出文件（如果您查看输出，则可能不想这样做）。为了构建最终的扩展模块，将编译SWIG输出文件并将其与C / C ++程序的其余部分链接以创建共享库。

对于许多目标语言，SWIG还将以目标语言生成代理类文件。这些特定于语言的文件的默认输出目录与生成的C / C ++文件的目录相同。可以使用-outdir选项修改。例如：

```
$ swig -c ++ -python -outdir pyfiles -o cppfiles / example_wrap.cpp example.i
```

如果目录cppfiles和pyfiles存在，将生成以下内容：

```
cppfiles / example_wrap.cpp
pyfiles / example.py
```

如果使用-outcurrentdir选项（不带-o），则SWIG的行为类似于典型的C / C ++编译器，则默认输出目录为当前目录。如果没有此选项，则默认输出目录是输入文件的路径。如果同时使用-o和 -outcurrentdir，则-outcurrentdir将被有效忽略，因为如果不使用-outdir覆盖，则语言文件的输出目录与生成的C / C ++文件位于同一目录。

5.1.3注释

C和C ++样式注释可能出现在界面文件中的任何位置。在SWIG的早期版本中，注释用于生成文档文件。但是，此功能目前正在修复中，并将在以后的SWIG版本中重新出现。

5.1.4 C预处理程序

像C一样，SWIG通过C预处理程序的增强版对所有输入文件进行预处理。支持所有标准预处理器功能，包括文件包含，条件编译和宏。但是，除非提供了-includeall命令行选项，否则将忽略#include语句 。禁用的原因包括：SWIG有时用于处理原始C头文件。在这种情况下，您通常只希望扩展模块在提供的头文件中包含函数，而不是该头文件可能包含的所有内容（即系统头，C库函数等）。

还应注意，SWIG预处理程序会跳过%{...%}块内包含的所有文本。此外，预处理器还包括许多宏处理增强功能，这些功能使其比普通的C预处理器更强大。这些扩展在“ 预处理器 ”一章中进行介绍。

5.1.5 SWIG指令

SWIG的大多数操作由特殊指令控制，这些指令始终以“ % ” 开头，以将其与常规C声明区分开。这些指令用于提供SWIG提示或以某种方式更改SWIG的解析行为。

由于SWIG指令不是合法的C语法，因此通常无法将它们包含在头文件中。但是，可以使用如下条件编译将SWIG指令包含在C头文件中：

```
/ * header.h ---一些头文件* /

/ * SWIG指令-仅在SWIG正在运行时可见* /
#ifdef SWIG
%module foo
#endif
```

SWIG是SWIG在解析输入文件时定义的特殊预处理符号。

5.1.6解析器限制

尽管SWIG可以解析大多数C / C ++声明，但是它没有提供完整的C / C ++解析器实现。这些限制中的大多数都与非常复杂的类型声明和某些高级C ++功能有关。具体来说，当前不支持以下功能：

- 非常规类型声明。例如，SWIG不支持以下声明（即使这是合法的C）：

  ```
  / *存储说明符（extern）的非常规放置* /
  const int extern编号；

  / *额外的声明符分组* /
  矩阵（foo）； //全局变量

  / *参数中的额外声明符分组* /
  void bar（Spam（Grok）（Doh））；
  ```

  实际上，很少（如果有的话）C程序员实际编写这样的代码，因为这种风格从未出现在编程书籍中。但是，如果您感到特别困惑，则可以确定破坏SWIG（尽管为什么要这么做？）。

- 不建议在C ++源文件（.C，.cpp或.cxx文件中的代码）上运行SWIG。通常的方法是提供SWIG头文件以解析C ++定义和声明。主要原因是，如果SWIG解析作用域定义或声明（对于C ++源文件是正常的），则将其忽略，除非先前已解析符号的声明。例如

  ```
  / * bar不包装，除非已定义foo且
     foo中的bar声明已被解析* /
  int foo :: bar（int）{
  ```

```
  ...任何...
}
```

- 尚未完全支持C ++的某些高级功能，例如嵌套类。请参阅“ C ++ 嵌套类”部分以获取更多信息。

如果发生解析错误，则可以使用条件编译来跳过有问题的代码。例如：

```
#ifndef SWIG
...一些错误的声明...
#endif
```

或者，您可以仅从接口文件中删除有问题的代码。

SWIG不提供完整的C ++解析器实现的原因之一是，它设计为可处理不完整的规范，并且在处理C / C ++数据类型时允许的程度很高（例如，即使缺少SWIG，SWIG仍可以生成接口）类声明或不透明的数据类型）。不幸的是，这种方法使实现C / C ++解析器的某些部分变得极为困难，因为大多数编译器使用类型信息来帮助解析更复杂的声明（对于真正好奇的实现而言，主要的复杂之处在于SWIG解析器不使用K&R第234页中所述的单独的*typedef-name*终端符号）。

5.2包装简单的C声明

SWIG通过创建一个与C程序中使用声明的方式紧密匹配的接口来包装简单的C声明。例如，考虑以下接口文件：

```
%module示例

%inline%{
extern double sin（double x）;
extern int strcmp（const char *, const char *）;
extern int Foo;
%}
#define状态50
#define版本“ 1.1”
```

在此文件中，有两个函数sin（）和strcmp（），一个全局变量Foo，以及两个常量STATUS和 VERSION。当SWIG创建扩展模块时，这些声明可以分别作为脚本语言函数，变量和常量来访问。例如，在Tcl中：

```
%sin 3
5.2335956
%strcmp Dave Mike
-1
%放置$ Foo
42
%放置$ STATUS
50
%放置$ VERSION
1.1
```

或在Python中：

```
>>> example.sin（3）
5.2335956
>>> example.strcmp（ '戴夫'， '麦克'）
-1
>>>打印example.cvar.Foo
42
>>>打印example.STATUS
50
>>>打印例版本
1.1
```

SWIG会尽可能创建一个与基础C / C ++代码紧密匹配的接口。但是，由于语言，运行时环境和语义之间的细微差异，因此并非总是可能做到这一点。接下来的几节描述了此映射的各个方面。

5.2.1基本类型处理

为了构建接口，SWIG必须将C / C ++数据类型转换为目标语言中的等效类型。通常，脚本语言提供的原始类型集比C语言少得多。因此，此转换过程涉及一定数量的类型强制。

大多数脚本语言提供使用C中的int或long数据类型实现的单个整数类型。以下列表显示了SWIG将在目标语言中与整数进行相互转换的所有C数据类型：

```
int
短
长
无符号
签名
无符号短短
无符号长
无符号char有
符号char
bool
```

从C转换整数值时，将使用强制转换将其转换为目标语言的表示形式。因此，C中的16位短可以被提升为32位整数。当整数朝另一个方向转换时，该值将转换回原始的C类型。如果该值太大而无法容纳，它将被无声地截断。

unsigned char和signed char是特殊情况，它们被当作小的8位整数处理。通常，char 数据类型映射为一个单字符ASCII字符串。

的布尔数据类型强制转换并从0和1的整数值，除非目标语言提供了一种特殊的布尔类型。

使用大整数值时需要格外小心。大多数脚本语言使用32位整数，因此映射64位长整数可能会导致截断错误。32位无符号整数（可能显示为较大的负数）可能会出现类似的问题。根据经验，可以安全使用int数据类型以及char和short数据类型的所有变体。对于无符号的int和long数据类型，在用SWIG包装程序之后，需要仔细检查程序的正确操作。

尽管SWIG解析器支持long long数据类型，但并非所有语言模块都支持它。这是因为long long 通常超过目标语言中可用的整数精度。在某些模块（例如Tcl和Perl5）中，长整型整数被编码为字符串。这样就可以代表这些数字的全部范围。但是，它不允许 在算术表达式中使用long long值。还应注意，尽管long long是ISO C99标准的一部分，但并非所有C编译器都普遍支持它。在尝试将这种类型与SWIG一起使用之前，请确保使用支持很长时间的编译器。

SWIG识别以下浮点类型：

```
浮动
双
```

浮点数与目标语言中浮点数的自然表示方式相互映射。这几乎总是C double。 SWIG不支持很少使用的long double数据类型。

该炭的数据类型被映射到一个以NULL终止的ASCII字符串，单个字符。当用于脚本语言时，它显示为包含字符值的细小字符串。将值转换回C时，SWIG从脚本语言中提取一个字符串，并去除第一个字符作为char值。因此，如果将值“ foo”分配给 char数据类型，则它将获得值“ f”。

该字符*数据类型为NULL结尾的ASCII字符串处理。SWIG将其映射为目标脚本语言的8位字符串。SWIG将目标语言中的字符串转换为NULL终止的字符串，然后再将其传递给C / C ++。这些字符串的默认处理不允许它们嵌入NULL字节。因此，char *数据类型通常不适合传递二进制数据。但是，可以通过定义SWIG类型图来更改此行为。有关详细信息，请参见“类型映射 ”一章。

目前，SWIG对Unicode和宽字符字符串（C wchar_t类型）提供了有限的支持。某些语言为wchar_t提供了类型映射，但请记住，它们可能无法在不同的操作系统之间移植。这是一个微妙的话题，许多程序员对此知之甚少，而且跨语言的实现方式不一致。对于那些提供Unicode支持的脚本语言，Unicode字符串通常以8位表示形式提供，例如UTF-8，可以将其映射到char *类型（在这种情况下，SWIG接口可能会起作用）。如果要包装的程序使用Unicode，则不能保证目标语言中的Unicode字符将使用相同的内部表示形式（例如，UCS-2与UCS-4）。您可能需要编写一些特殊的转换函数。

5.2.2全局变量

SWIG尽可能将C / C ++全局变量映射为脚本语言变量。例如，

```
%模块示例
double foo;
```

产生如下脚本语言变量：

```
＃Tcl的
设置foo的[3.5];＃设置FOO至3.5
使$ foo的;＃打印foo的值

＃的Python
cvar.foo = 3.5＃设置FOO∽3.5
的foo打印cvar.foo＃打印值

＃Perl的
$富= 3.5 ; ＃将foo设置为3.5
print $ foo，“ \ n”; FOO的＃打印值

＃红宝石
```

```
Module.foo = 3.5#集富至3.5
打印Module.foo， “\ n” #打印FOO的值
```

无论何时使用脚本语言变量，都将访问基础的C全局变量。尽管SWIG尽一切努力使全局变量像脚本语言变量一样工作，但并非总是可以这样做。例如，在Python中，必须通过称为cvar的特殊变量对象（如上所示）访问所有全局变量。在Ruby中，变量作为模块的属性被访问。其他语言可能会将变量转换为一对访问器函数。例如，Java模块生成一对用于操纵值的函数double get_foo（） 和set_foo（double val）。

最后，如果全局变量已声明为const，则仅支持只读访问。注意：此行为是SWIG-1.3的新增功能。SWIG的早期版本错误地处理了const并创建了常量。

5.2.3常数

可以使用#define，枚举或特殊的%constant指令创建常量。以下接口文件显示了一些有效的常量声明：

```
#define I_CONST 5 //整数常量
#define PI 3.14159 //浮点常量
#define S_CONST“ hello world” //字符串常量
#define NEWLINE'\ n'//字符常量

enum boolean {NO = 0，是= 1};
列举月份{1月，2月，3月，4月，5月，6月，7月，8月，9月
             ，10月，11月，12月};
%constant double BLAH = 42.37;
#define PI_4 PI / 4
#define FLAGS 0x04 | 0x08 | 0x40
```

在#define声明中，常量的类型由语法推断。例如，假定带小数点的数字为浮点数。此外，SWIG必须能够完全解析#define中使用的所有符号，以便实际创建常量。此限制是必要的，因为 #define也用于定义预处理器宏，这些宏绝对不打算成为脚本语言接口的一部分。例如：

```
#define EXTERN extern

EXTERN void foo（）;
```

在这种情况下，您可能不想创建一个名为EXTERN的常量 （值是多少？）。通常，SWIG不会为宏创建常量，除非该值可以由预处理器完全确定。例如，在上面的示例中，声明

```
#定义PI_4 PI / 4
```

定义一个常量，因为PI已被定义为常量并且该值是已知的。但是，出于相同的保守原因，即使使用简单转换的常量也将被忽略，例如

```
#define F_CONST（double）5 //带强制转换的浮点常量
```

允许使用常量表达式，但SWIG不会对其进行求值。而是将它们传递到输出文件，并让C编译器执行最终评估（但是，SWIG会执行有限形式的类型检查）。

对于枚举，至关重要的是将原始枚举定义包含在接口文件中的某个位置（在头文件中或在%{%}块中）。SWIG仅将枚举转换为将常量添加到脚本语言所需的代码。它需要原始的枚举声明，以便获得C编译器分配的正确的枚举值。

所述%恒定指令用于更精确地创建对应于不同的C数据类型常数。尽管对于简单值通常不需要它，但是在处理指针和其他更复杂的数据类型时它会更加有用。通常， 仅当您要向脚本语言界面中添加原始头文件中未定义的常量时，才使用%constant。

5.2.4关于const的简短介绍

C编程的一个常见困惑是声明中const限定词的语义含义 -尤其是当它与指针和其他类型修饰符混合使用时。实际上，SWIG的早期版本处理const的方式不正确-SWIG-1.3.7和较新的版本已修复这种情况。

从SWIG-1.3开始，所有变量声明（无论是否使用const）都被包装为全局变量。如果声明恰好被声明为const，则将其包装为只读变量。要确定变量是否为const，您需要查看const 限定符（出现在变量名称之前）的最右端。如果最右边的 const出现在所有其他类型修饰符（例如指针）之后，则该变量为const。否则，事实并非如此。

以下是const声明的一些示例。

```
const char a; //一个常量字符
char const b; //一个常量字符（相同）
char * const c; //指向字符
const char * const d的常量指针; //指向常量字符的常量指针
```

这是一个非const声明的示例：

```
const char * e; //指向常量字符的指针。指针
                          //可以修改。
```

在这种情况下，指针e可以更改-只是指向的值是只读的。

请注意，对于函数中使用的const参数或返回类型，SWIG几乎忽略了它们是const的事实， 有关更多信息，请参见const正确性部分。

兼容性说明：更改SWIG以将const声明作为只读变量处理的原因之一 是，在许多情况下const变量的值可能会更改。例如，库可能会在其公共API中将符号导出为 const以阻止修改，但仍允许通过其他内部机制来更改值。此外，程序员常常忽略了这样的事实，即使用char * const这样的常量声明，可以修改所指向的基础数据-只是指针本身是常量。在嵌入式系统中，const声明可能指向只读内存地址，例如内存映射的I / O设备端口的位置（该值会更改，但硬件不支持对该端口进行写操作）。而不是试图建立一堆特殊情况下进入const的预选赛上，全新演绎常量为“只读”，是简单而准确的实际语义匹配常量在C / C ++。如果您真的想像在较早版本的SWIG中那样创建一个常量，请改用%constant指令。例如：

```
%constant double PI = 3.14159;
```

要么

```
#ifdef SWIG
#define const%constant
#endif
const double foo = 3.4;
const double bar = 23.4;
const int spam = 42;
#ifdef SWIG
#undef const
#endif
...
```

5.2.5 char的警示故事*

在进一步介绍之前，现在必须提到涉及char *的一些注意事项 。当字符串从脚本语言传递给C char *时，指针通常指向存储在解释器中的字符串数据。修改此数据几乎总是一个非常糟糕的主意。此外，某些语言可能明确禁止这样做。例如，在Python中，字符串应该是不可变的。如果您违反了此规定，那么在将模块释放到世界上时，您可能会遭受巨大的愤怒。

问题的主要来源是可能会就地修改字符串数据的函数。一个经典的例子就是这样的函数：

```
char * strcat（char * s，const char * t）
```

尽管SWIG一定会为此生成包装，但是其行为是不确定的。实际上，这可能会导致您的应用程序因段故障或其他与内存相关的问题而崩溃。这是因为s指的是目标语言中的一些内部数据，即您不应接触的数据。

底线：除了只读输入值外，不要依赖char *。但是，必须注意，您可以使用typemaps更改SWIG的行为。

5.3指针和复杂对象

大多数C程序都会操纵数组，结构和其他类型的对象。本节讨论这些数据类型的处理。

5.3.1简单指针

指向原始C数据类型的指针，例如

```
整数*
双***
字符**
```

SWIG完全支持。SWIG并没有尝试将指向的数据转换为脚本表示，而是将指针本身编码为一个包含指针实际值和类型标记的表示。因此，上述指针的SWIG表示（在Tcl中）可能看起来像这样：

```
_10081012_p_int
_1008e124_ppp_double
_f8ac_pp_char
```

空指针由字符串“ NULL”或用类型信息编码的值0表示。

SWIG将所有指针视为不透明对象。因此，指针可以由函数返回并根据需要传递给其他C函数。出于所有实际目的，脚本语言接口的工作方式与在C程序中使用指针的方式完全相同。唯一的区别是没有取消引用指针的机制，因为这将要求目标语言理解底层对象的内存布局。

指针值的脚本语言表示形式绝对不能直接操作。即使显示的值看起来像十六进制地址，但使用的数字可能与实际的机器地址有所不同（例如，在小字节序机器上，数字可能以相反的顺序出现）。此外，SWIG通常不会将指针映射到高级对象（如关联数组或列表）中（例如，将 int *转换为整数列表）。SWIG不这样做的原因有几个：

- C声明中没有足够的信息来正确地将指针映射到更高级别的构造中。例如，一个int * 可能确实是一个整数数组，但是如果它包含一千万个元素，则将其转换为列表对象可能不是一个好主意。
- SWIG不知道与指针关联的基础语义。例如，一个int *可能根本不是一个数组-也许它是一个输出值！
- 通过以一致的方式处理所有指针，SWIG的实现大大简化并且不容易出错。

5.3.2运行时指针类型检查

通过允许从脚本语言操纵指针，扩展模块有效地绕过了C / C ++编译器中的编译时类型检查。为了防止错误，将类型签名编码到所有指针值中，并用于执行运行时类型检查。这种类型检查过程是SWIG不可或缺的一部分，如果不使用类型图（在后面的章节中介绍），则不能禁用或修改它。

像C一样，void *匹配任何类型的指针。此外， 可以将NULL指针传递给希望接收指针的任何函数。尽管这有可能导致崩溃，但是NULL指针有时也用作哨兵值或表示缺失/空值。因此，SWIG将NULL指针留给应用程序检查。

5.3.3派生类型，结构和类

对于其他所有内容（结构，类，数组等），SWIG都应用了一个非常简单的规则：

其他一切都是指针

换句话说，SWIG通过引用来操纵其他所有内容。该模型之所以有意义，是因为大多数C / C ++程序大量使用了指针，而SWIG可以使用已经存在的类型检查指针机制来处理指向基本数据类型的指针。

尽管这听起来很复杂，但实际上非常简单。假设您有一个像这样的接口文件：

```
%module fileio
FILE * fopen（char *，char *）；
int fclose（FILE *）；
unsigned fread（void * ptr，unsigned size，unsigned nobj，FILE *）；
无符号fwrite（void * ptr，无符号大小，无符号nobj，FILE *）；
无效* malloc（int nbytes）；
void free（void *）；
```

在此文件中，SWIG不知道FILE是什么，但是由于它被用作指针，所以它到底是什么并不重要。如果将此模块包装到Python中，则可以像您期望的那样使用这些函数：

```
＃复制文件
def filecopy（source，target）：
    f1 = fopen（source，“ r”）
    f2 = fopen（target，“ w”）
    buffer = malloc（8192）
    nbytes = fread（buffer，8192，1，f1）
    while（nbytes> 0）：
        fwrite（buffer，8192，1，f2）
            nbytes = fread（buffer，8192，1，f1）
    free（buffer）
```

在这种情况下，f1，f2和buffer都是包含C指针的不透明对象。不管它们包含什么值，只要没有这些知识，我们的程序就可以正常工作。

5.3.4未定义的数据类型

当SWIG遇到未声明的数据类型时，它将自动假定它是结构或类。例如，假设以下功能出现在SWIG输入文件中：

```
void matrix_multiply（矩阵* a，矩阵* b，矩阵* c）；
```

SWIG不知道什么是“ 矩阵 ”。但是，它显然是指向某物的指针，因此SWIG使用其通用指针处理代码生成包装器。

与C或C ++不同，SWIG实际上并不关心是否 在接口文件中预先定义了Matrix。这允许SWIG仅根据部分或有限的信息生成接口。在某些情况下，只要您可以在脚本语言界面中将不透明引用传递给Matrix，您就可能不在乎Matrix到底是什么。

要提到的一个重要细节是，当存在未指定的类型名称时，SWIG会很高兴为接口生成包装器。但是，**所有未指定的类型都在内部作为指向结构或类的指针处理！**例如，考虑以下声明：

```
无效foo（size_t num）；
```

如果未声明size_t，则SWIG会生成希望接收到size_t *类型的包装器（此映射将在稍后描述）。结果，脚本接口的行为可能会很奇怪。例如：

```
foo（40）；
TypeError：预期为_p_size_t。
```

解决此问题的唯一方法是确保使用typedef正确声明类型名称 。

5.3.5 Typedef

与C一样，typedef可用于在SWIG中定义新的类型名称。例如：

```
typedef unsigned int size_t;
```

SWIG接口中出现的typedef定义不会传播到生成的包装器代码。因此，需要在包含的头文件中定义它们，或将它们放在声明部分中，如下所示：

```
%{
/ *包含在生成的包装文件中* /
typedef unsigned int size_t;
%}
/ *告诉SWIG * /
typedef unsigned int size_t;
```

要么

```
%inline%{
typedef unsigned int size_t;
%}
```

在某些情况下，您可能可以包括其他头文件来收集类型信息。例如：

```
%模块示例
%import“ sys / types.h”
```

在这种情况下，您可以按以下方式运行SWIG：

```
$ swig -I / usr / include -includeall example.i
```

应该注意的是，这里的里程会相差很大。众所周知，系统头很复杂，并且可能依赖于各种非标准的C编码扩展（例如，针对GCC的特殊指令）。除非您确切地指定正确的包含目录和预处理器符号，否则这可能无法正常工作（您必须进行实验）。

SWIG跟踪typedef声明，并将此信息用于运行时类型检查。例如，如果您使用上面的typedef并具有以下函数声明：

```
无效foo（unsigned int * ptr）；
```

相应的包装函数将接受unsigned int *或size_t *类型的参数。

5.4其他实用性

到目前为止，本章已经介绍了将SWIG用于简单接口所需的几乎所有知识。但是，某些C程序使用的惯用法在映射到脚本语言接口时会更加困难。本节介绍其中一些问题。

5.4.1按价值传递结构

有时，C函数采用按值传递的结构参数。例如，考虑以下功能：

```
double dot_product（Vector a，Vector b）；
```

为了解决这个问题，SWIG通过创建等效于以下内容的包装器将函数转换为使用指针：

```
double wrap_dot_product（Vector * a，Vector * b）{
  Vector x = * a;
  向量y = * b;
  返回dot_product（x，y）;
}
```

在目标语言中，dot_product（）函数现在接受指向Vector的指针，而不是Vector。在大多数情况下，此转换是透明的，因此您可能不会注意到。

5.4.2按值返回

按值返回结构或类数据类型的C函数更难处理。考虑以下功能：

```
向量cross_product（向量v1，向量v2）;
```

该函数希望返回Vector，但SWIG仅真正支持指针。结果，SWIG创建了这样的包装器：

```
向量* wrap_cross_product（向量* v1，向量* v2）{
  向量x = * v1;
  向量y = * v2;
  向量*结果;
  结果=（向量*）malloc（sizeof（向量））;
  *（结果）=交叉（x，y）;
  返回结果;
}
```

或如果SWIG与-c ++选项一起运行：

```
向量* wrap_cross（向量* v1，向量* v2）{
  向量x = * v1;
  向量y = * v2;
  向量*结果=新向量（cross（x，y））; //使用默认的复制构造函数
  返回结果;
}
```

在这两种情况下，SWIG都会分配一个新对象并返回对该对象的引用。由用户决定何时不再使用返回的对象。显然，如果您不知道隐式内存分配，并且不采取步骤释放结果，则会泄漏内存。也就是说，应该注意的是，某些语言模块现在可以自动跟踪新创建的对象并为您回收内存。有关更多详细信息，请查阅每种语言模块的文档。

还应注意，在C ++中按值进行传递/返回的处理有一些特殊情况。例如，如果Vector没有定义默认构造函数，则以上代码片段将无法正常工作。关于SWIG和C ++的部分提供了有关这种情况的更多信息。

5.4.3链接到结构变量

当遇到涉及结构的全局变量或类成员时，SWIG会将其作为指针处理。例如，像这样的全局变量

```
向量unit_i;
```

被映射到下面的一对set / get函数：

```
向量* unit_i_get（）{
  return&unit_i;
}
void unit_i_set（Vector * value）{
  unit_i = * value;
}
```

再次需要注意一些。以这种方式创建的全局变量将显示为目标脚本语言中的指针。释放或销毁此类指针将是一个非常糟糕的主意。同样，C ++类必须提供正确定义的副本构造函数，以使分配正常工作。

5.4.4链接到char *

当出现char *类型的全局变量时，SWIG使用malloc（）或 new来为新值分配内存。具体来说，如果您有这样的变量

```
char * foo;
```

SWIG生成以下代码：

```
/ * C模式* /
void foo_set（char * value）{
  if（foo）free（foo）;
  foo =（字符*）malloc（strlen（value）+1）;
  strcpy（foo，value）;
}

/ * C ++模式。使用-c ++选项时* /
void foo_set（char * value）{
  if（foo）delete [] foo;
  foo =新的char [strlen（value）+1];
  strcpy（foo，value）;
}
```

如果这不是您想要的行为，请考虑使用%immutable指令将变量设置为只读 。或者，您可以编写一个简短的辅助函数以完全根据需要设置该值。例如：

```
%inline%{
  void set_foo（char * value）{
    strncpy（foo，value，50）;
  }
%}
```

注意：如果编写这样的辅助函数，则必须从目标脚本语言将其作为函数调用（它不能像变量一样工作）。例如，在Python中，您将必须编写：

```
>>> set_foo（“ Hello World”）
```

char *变量的 一个常见错误是链接到这样声明的变量：

```
char * VERSION =“ 1.0”;
```

在这种情况下，该变量将是可读的，但是任何尝试更改该值的操作都会导致分段或常规保护错误。这是由于以下事实：当未使用malloc（）或new分配当前分配给变量的字符串文字值时，SWIG尝试使用free或delete释放旧值 。要解决此问题，您可以将变量标记为只读，编写类型映射（如第6章所述）或编写特殊的set函数，如图所示。另一种选择是将变量声明为数组：

```
char VERSION [64] =“ 1.0”;
```

声明const char * 类型的变量时，SWIG仍会生成用于设置和获取值的函数。但是，默认行为并*没有*释放以前的内容（从而可能导致内存泄漏）。实际上，包装此类变量时，您可能会收到诸如以下的警告消息：

```
范例：i.20。类型图警告。设置const char *变量可能会泄漏内存
```

出现这种现象的原因是，经常使用const char *变量来指向字符串文字。例如：

```
const char * foo =“ Hello World \ n”;
```

因此，在这样的指针上调用free（）是一个非常糟糕的主意。另一方面，将指针更改为指向其他某个值*是*合法的。设置这种类型的变量时，SWIG会分配一个新字符串（使用malloc或new）并更改指针以指向新值。但是，由于不释放旧值，因此重复修改该值将导致内存泄漏。

5.4.5数组

SWIG完全支持数组，但是它们始终作为指针处理，而不是将它们映射到目标语言中的特殊数组对象或列表。因此，以下声明：

```
int foobar（int a [40]）;
void grok（char * argv []）;
void transpose（double a [20] [20]）;
```

就像真的这样声明它们一样进行处理：

```
int foobar(int * a);
void grok(char ** argv);
void transpose(double(* a)[20]);
```

像C一样，SWIG不执行数组边界检查。用户应确保指针指向适当分配的内存区域。

多维数组被转换为指向较小一维数组的指针。例如：

```
int [10]; //映射为int *
int [10] [20]; //映射到int(*)[20]
int [10] [20] [30]; //映射到int(*)[20] [30]
```

需要注意的是在C型系统是非常重要的，多维数组一个[] []是NOT等价于单个指针 *一个或一个双指针如**一个。而是使用指向数组的指针（如上所示），其中指针的实际值是数组的起始存储位置。强烈建议读者在使用SWIG之前，先清理一下C书，并重新阅读数组中的内容。

支持数组变量，但默认情况下为只读。例如：

```
int a [100] [200];
```

在这种情况下，读取变量'a'会返回类型为int(*)[200]的指针，该指针指向 数组&a [0] [0]的第一个元素。尝试修改“ a”会导致错误。这是因为SWIG不知道如何将数据从目标语言复制到阵列中。要解决此限制，您可能需要编写一些简单的辅助函数，如下所示：

```
%inline%{
void a_set(int i, int j, int val) {
  a [i] [j] = val;
}
int a_get(int i, int j) {
  return a [i] [j];
}
%}
```

为了动态创建各种大小和形状的数组，在界面中编写一些辅助函数可能很有用。例如：

```
//一些数组助手
%inline%{
  / *创建任何类型的[size]数组* /
  int * int_array(int size) {
    return(int *) malloc(size * sizeof(int));
  }
  / *创建一个二维数组[size] [10] * /
  int(* int_array_10(int size))[10] {
    return(int(*) [10]) malloc(size * 10 * sizeof(int) );
  }
%}
```

char 数组由SWIG作为特殊情况处理。在这种情况下，可以将目标语言的字符串存储在数组中。例如，如果您有这样的声明，

```
字符路径名[256];
```

SWIG生成用于获取和设置值的函数，它们等效于以下代码：

```
char * pathname_get() {
  返回路径名;
}
void pathname_set(char * value) {
  strncpy(pathname, value, 256);
}
```

在目标语言中，可以像普通变量一样设置该值。

5.4.6创建只读变量

可以使用%immutable 伪指令创建一个只读变量，如下所示：

```
// File: interface.i

int a; //可以读/写
%immutable;
int b, c, d; //只读变量
%mutable;
x, y的两倍 //读/写
```

在%不变指令启动只读模式，直到它使用的是明确禁用%可变指令。作为关闭和打开只读模式的替代方法，也可以将各个声明标记为不可变的。例如：

```
%不可变x; //将x设为只读
...
double x; //只读（来自先前的%immutable指令）
double y; //读写
...
```

在%可变和%不变的指令实际上 %的功能指令这样定义：

```
#define%immutable%feature（“ immutable”）#
define%mutable%feature（“ immutable”，“”）
```

如果要使所有包装的变量为只读，但不限制一两个，则采用这种方法可能会更容易：

```
%不可变的; //将所有变量设为只读
%feature（“ immutable”，“ 0”）x; //除了，使x读/写
...
double x;
y
双z;
...
```

当声明声明为const时，也会创建只读变量。例如：

```
const int foo; / *只读变量* /
char * const version =“ 1.0”; / *只读变量* /
```

兼容性说明：只读访问曾经由一对指令 %readonly和%readwrite控制。尽管这些指令仍然有效，但它们会生成警告消息。只需将指令更改为%immutable;和 %mutable; 取消警告。不要忘记多余的分号！

5.4.7重命名和忽略声明

5.4.7.1简单重命名特定标识符

通常，将C声明的名称包装到目标语言时使用该名称。但是，这可能会与关键字或脚本语言中已经存在的功能产生冲突。要解决名称冲突，可以使用%rename 指令，如下所示：

```
// interface.i

%rename（my_print）打印;
extern void print（const char *）;

%rename（foo）a_really_long_and_annoying_name;
extern int a_really_long_and_annoying_name;
```

SWIG仍然调用正确的C函数，但是在这种情况下，函数print（）会在目标语言中真正被称为“ my_print（） ”。

%rename指令 的位置是任意的，只要它出现在要重命名的声明之前即可。一种常见的技术是编写用于包装头文件的代码，如下所示：

```
// interface.i

%rename（my_print）打印;
%rename（foo）a_really_long_and_annoying_name;

%include“ header.h”
```

%rename将重命名操作应用于将来出现的所有名称。重命名适用于函数，变量，类和结构名称，成员函数以及成员数据。例如，如果您有两个C ++类，所有这些类都具有名为“ print”的成员函数（在Python中是关键字），则可以通过指定以下内容将它们全部重命名为“ output”：

```
%rename（输出）打印； //将所有`print'函数重命名为`output'
```

SWIG通常不会执行任何检查以查看其包装的功能是否已在目标脚本语言中定义。但是，如果您对名称空间和模块的使用非常谨慎，则通常可以避免这些问题。

包装C代码时，通常只需使用带有%rename的标识符/符号即可。在包装C ++代码时，在使用C ++功能（例如函数重载，默认参数，名称空间，模板特殊化等）时，简单地使用带有%rename的简单标识符/符号可能会太过局限。如果使用%rename指令和C ++，请确保您可以阅读 SWIG和C ++章节，尤其是有关 方法重载和默认参数的重命名和歧义解决的章节 。

与%rename紧密相关的是%ignore指令。 %ignore指示SWIG忽略与给定标识符匹配的声明。例如：

```
%ignore打印；//忽略所有名为print
%ignore MYMACRO的声明；//忽略宏
...
#define MYMACRO 123
void print（const char *）；
...
```

与%ignore匹配的任何函数，变量等都不会被包装，因此无法从目标语言中获取。%ignore的常见用法是有选择地从头文件中删除某些声明，而不必向头添加条件编译。但是，应该强调的是，这仅适用于简单的声明。如果您需要删除整段有问题的代码，则应改用SWIG预处理程序。

兼容性说明：较早版本的SWIG提供了用于重命名声明的特殊%name指令。例如：

```
%name（输出）extern void print（const char *）；
```

该指令仍受支持，但已弃用，应该避免使用。在%重命名 指令是原始头文件信息功能更强大，更好的支持包装。

5.4.7.2高级重命名支持

尽管为特定的声明编写%rename很简单，但有时需要将相同的重命名规则应用于SWIG输入中的许多（也许是全部）标识符。例如，可能有必要对目标语言中的所有名称进行某种转换，以更好地遵循其命名约定，例如为所有包装的函数添加特定的前缀。为每个函数单独进行操作是不切实际的，因此，如果未指定要重命名的标识符的名称，SWIG支持对所有声明应用重命名规则：

```
%rename（“ myprefix_%s”）“”； //打印-> myprefix_print
```

这也表明%rename的参数不必是文字字符串，而可以是类似printf（）的格式字符串。如上所示，以最简单的形式将“%s”替换为原始声明的名称。但是，这并不总是足够的，SWIG提供了对常规格式字符串语法的扩展，以允许对参数应用（SWIG定义的）函数。例如，要将所有C函数do_something_long（）包装 为更像Java的doSomethingLong（），可以使用“ lowercamelcase”扩展格式说明符，如下所示：

```
%rename（“%（lowercamelcase）s”）“”； // foo_bar-> fooBar; FooBar-> fooBar
```

某些函数可以参数化，例如“ strip”可以从其参数中删除提供的前缀。将前缀指定为格式字符串的一部分，在函数名称后加冒号：

```
%rename（“%（strip：[wx]）s”）“”； // wxHello-> Hello; FooBar-> FooBar
```

下表总结了所有当前定义的功能，并举例说明了每个功能。请注意，其中一些具有两个名称，一个较短的名称和一个更具描述性的名称，但其他两个功能在其他方面是等效的：

功能	退货	示例（输入/输出）	
大写或大写	字符串的大写版本。	打印	打印
小写或小写	字符串的小写版本。	打印	打印
标题	字符串，首字母大写，其余小写。	打印	打印
第一大写	字符串，首字母大写，其余字母不变。	打印	打印它
小写	字符串，首字母小写，其余字母不变。	打印它	打印
驼峰或ctitle	字符串，首字母大写，下划线后的任何字母（在处理中都将被删除），其余均小写。	print_it	打印它

小写字母大写或小写	字符串，下划线后的每个字母（在过程中将被删除）均大写并休息，包括小写的第一个字母。	print_it	打印
底壳或utitle	小写字符串，在原始字符串的每个大写字母之前插入下划线，并在字符串末尾插入任何数字。从逻辑上讲，这是骆驼背的反面。	打印它	print_it
策划	带有所有下划线的字符串用破折号代替，从而使Lispers / Schemers更令人愉悦。	print_it	打印它
删除：[前缀]	没有给定前缀的字符串，或者如果字符串不是以此前缀开头的则为原始字符串。请注意，方括号应按原样使用，例如%rename（" strip：[wx]"）	wxPrint	打印
rstrip：[后缀]	不带给定后缀的字符串，或者原始字符串（如果不以此后缀结尾）。请注意，应在字面上使用方括号，例如%rename（" rstrip：[Cls]"）	印刷品	打印
正则表达式：/ pattern / subst /	（类似于Perl的）正则表达式替换操作之后的字符串。此函数允许将任意正则表达式应用于标识符名称。所述 *图案*部分是在Perl的语法规则的表达（通过所支持的Perl兼容正则表达式（PCRE））库和*SUBST*字符串可以包含以下形式的反向引用\ n其中Ñ是一个数字从0至9，或一个以下转义序列中的一个：\ l，\ L，\ u，\ U或\ E。向后引用被相应捕获组的内容替换，而转义序列在替换字符串中执行大小写转换：\ l和\ L转换为小写，而\ u和\ U转换为大写。每对元素之间的区别在于\ l和\ u仅 更改下一个字符的大小写，而\ L和\ U则对所有其余字符进行更改，直到遇到\ E。最后请注意，反斜杠需要在C字符串中转义，因此在实践中" \\"必须在所有这些转义序列中使用。例如，要删除下划线前的任何字母前缀并大写其余部分，可以使用以下指令： %rename（" regex：/（\\ w +）_（。*）/ \\ u \\ 2 /"）	prefix_print	打印
命令：cmd	外部命令cmd的输出，并将字符串作为输入传递给它。请注意，与所有其他函数相比，此函数非常慢，因为它涉及生成单独的进程，因此不建议将其用于许多声明。该*cmd*不在方括号中，而是必须以三重'<'结束，例如%rename（" command：tr -d aeiou <<<"） （删除所有元音的荒谬示例）	打印	prnt

以上所有功能中最通用的功能（不包括计数 命令，其原理上甚至更强大，但出于性能考虑通常应避免使用）是 正则表达式。以下是其用法的更多示例：

```
//从所有标识符中删除wx前缀，但以wxEVT开头的标识符除外
%rename（"%（regex：/ wx（？！EVT）（。*）/ \\ 1 /）s"）""; // wxSomeWidget-
                                                > SomeWidget // wxEVT_PAINT-> wxEVT_PAINT

//应用重命名枚举元素的规则，以避免公共前缀
//在C#/ Java中是多余的
%rename（"%（regex：/ ^（[AZ] [az] +）+ _（。*）/ \\ 2 /）s", %$ isenumitem）""; // Colour_Red-> Red

//删除所有"设置/获取"前缀。
%rename（"%（regex：/ ^（Set | Get）（。*）/ \\ 2/2 / ss"）""; // SetValue->值
                                                // GetValue->值
```

和以前一样，上面有关%rename的所有说明也适用于 %ignore。实际上，后者只是前者的一种特殊情况，忽略标识符与将其重命名为特殊的 " $ ignore"值相同。所以以下片段

```
%ignore打印；
```

和

```
%rename（" $ ignore"）打印；
```

完全等效，并且%rename可以使用前面描述的匹配可能性来有选择地忽略多个声明。

5.4.7.3限制全局重命名规则

如前几节所述，可以重命名单个声明或一次将重命名规则应用于所有声明。但实际上，后者很少适用，因为总则总是有一些例外。为了处理它们，可以使用后续的匹配参数来限制未命名的%rename的范围 。可以将其应用于SWIG关联的任何属性，并在其输入中出现声

明。例如：

```
%rename（“ foo”，match $ name =“ bar”）“”;
```

可以用来达到与简单的效果相同的效果

```
%rename（“ foo”）栏;
```

因此，就其本身而言，这并不是很有趣。但是match也可以应用于声明类型，例如match =“ class”仅将匹配限制为类声明（在C ++中），而match =“ enumitem”将其限制为枚举元素。SWIG还为此类匹配表达式提供了便利宏，例如

```
%rename（“%（title）s”，%$ isenumitem）“”;
```

将所有枚举元素的名称大写，但不会更改其他声明的大小写。同样，可以使用%$ isclass，%$ isfunction， %$ isconstructor，%$ isunion，%$ istemplate和%$ isvariable。可能还有许多其他检查，并且本文档并不详尽，请参阅swig.swg中的“%rename谓词”部分以 获取支持的匹配表达式的完整列表。

除了从字面上将某些字符串与match匹配之外，您还可以使用regexmatch或notregexmatch将字符串与正则表达式进行匹配。例如，要忽略所有后缀为“ Old”的函数，可以使用

```
%rename（“ $ ignore”，regexmatch $ name =“ Old $”）“”;
```

对于像这样的简单情况，最好为声明名称直接指定正则表达式，也可以使用regextarget来完成 ：

```
%rename（“ $ ignore”，regextarget = 1）“ Old $”;
```

注意，仅对声明本身的名称进行检查，如果您需要匹配C ++声明的全名，则必须使用fullname属性：

```
%rename（“ $ ignore”，regextarget = 1，fullname = 1）“ NameSpace :: ClassName ::。* Old $”;
```

至于notregexmatch，它将匹配仅限制为与指定的正则表达式不匹配的字符串。因此，要将所有声明重命名为小写，仅包含大写字母的声明除外：

```
%rename（“ $（lower）s”，notregexmatch $ name =“ ^ [AZ] + $”）“”;
```

最后，%rename和%ignore指令的变体可以用于帮助包装C ++重载函数和方法或使用默认参数的C ++方法。C ++章节的“ 重命名和歧义解决”部分对此进行了描述 。

5.4.7.4忽略所有内容，然后包装一些选定的符号

使用上述技术，可以忽略标头中的所有内容，然后有选择地包装一些选定的方法或类。例如，考虑一个标头myheader.h ，其中包含许多类，而在该标头中只需要一个名为Star的类，则可以采用以下方法：

```
%忽视 “”; //忽略一切//忽略

选择的类'Star'
%rename（“%s”）Star;

//由于忽略所有内容，因此在类中将包括构造函数，析构函数，方法等
//，因此也必须明确忽略它们：
%rename（“%s”）Star :: Star;
%rename（“%s”）Star ::〜Star;
%rename（“%s”）Star :: shine; //命名方法

%include“ myheader.h”
```

另一种方法可能更合适，因为它不需要命名所选类中的所有方法，而只是从忽略类开始。这不会对类的任何成员添加显式忽略，因此，当所选类被忽略时，其所有方法都将被包装。

```
%rename（$ ignore，%$ isclass）“”; //只忽略所有类
%rename（“%s”）Star; //忽略'Star'
%include“ myheader.h”
```

5.4.8默认/可选参数

SWIG支持C和C ++代码中的默认参数。例如：

```
int plot（double x，double y，int color = WHITE）；
```

在这种情况下，SWIG会生成包装器代码，其中默认参数在目标语言中是可选的。例如，可以在Tcl中按以下方式使用此函数：

```
%plot -3.4 7.5＃使用默认值
%plot -3.4 7.5 10＃设置颜色为10
```

尽管ISO C标准不允许使用默认参数，但SWIG接口中指定的默认参数可同时用于C和C ++。

注意：关于默认参数的使用和SWIG生成的包装器代码，存在一个细微的语义问题。在C代码中使用默认参数时，默认值会发送到包装器中，并使用一组完整的参数来调用该函数。这与包装C ++时不同，在C ++中，为每个默认参数生成一个重载的包装器方法。 有关更多详细信息，请参考C ++章节中有关默认参数的部分。

5.4.9指向函数和回调的指针

有时，C库可能包含期望接收指向函数的指针的函数-可能用作回调。当回调函数是用C而不是目标语言定义的时，SWIG为函数指针提供完全支持。例如，考虑如下函数：

```
int binary_op（int a，int b，int（* op）（int，int））；
```

当您第一次将类似这样的内容包装到扩展模块中时，您可能会发现该功能无法使用。例如，在Python中：

```
>>> def add（x，y）：
...返回x + y
...
>>> binary_op（3，4，add）追溯（
最近一次调用）：
  文件“ <stdin>”，第1行，在？
TypeError：类型错误。预期的_p_f_int_int__int
>>>
```

发生此错误的原因是，SWIG不知道如何将脚本语言函数映射到C回调中。但是，现有的C函数可以用作参数，只要将它们安装为常量即可。一种方法是使用%constant指令，如下所示：

```
/ *具有回调的函数* /
int binary_op（int a，int b，int（* op）（int，int））；

/ *一些回调函数* /
%constant int add（int，int）；
%constant int sub（int，int）；
%constant int mul（int，int）；
```

在这种情况下，add，sub和mul成为目标脚本语言中的函数指针常量。这使您可以如下使用它们：

```
>>> binary_op（3，4，添加）
7
>>> binary_op（3，4，mul）
12
>>>
```

不幸的是，通过将回调函数声明为常量，它们不再可以作为函数访问。例如：

```
>>> add（3，4）
追溯（最近一次呼叫过去）：
  文件“ <stdin>”，第1行，在？中？
TypeError：无法调用对象：'_ff020efc_p_f_int_int__int'
>>>
```

如果要使函数既可以用作回调函数又可以用作函数，则可以使用%callback和%nocallback伪指令，如下所示：

```
/ *具有回调的函数* /
int binary_op（int a，int b，int（* op）（int，int））；

/ *一些回调函数* /
%callback（“%s_cb”）；
int add（int，int）；
int sub（int，int）；
int mul（int，int）；
%nocallback;
```

%callback 的参数是一个printf样式的格式字符串，该字符串指定回调常量的命名约定（%s被替换为函数名）。回调模式将一直有效，直到使用%nocallback明确禁用它为止。当您执行此操作时，该界面现在如下所示：

```
>>> binary_op（3，4，add_cb）
7
>>> binary_op（3，4，mul_cb）
12
>>> add（3，4）
7
>>> mul（3，4）
12
```

请注意，将函数用作回调时，将使用特殊名称（例如add_cb）。要正常调用该函数，只需使用原始函数名称，例如add（）即可。

SWIG提供了许多对标准C printf格式的扩展，在这种情况下可能有用。例如，以下变体将回调安装为所有大写常量，例如 ADD，SUB和MUL：

```
/ *一些回调函数* /
%callback（“%（uppercase）s”）；
int add（int，int）；
int sub（int，int）；
int mul（int，int）；
%nocallback;
```

格式字符串“%（lowercase）s”将所有字符转换为小写。字符串“%（title）s”将大写第一个字符，并将其余字符转换为小写。

现在，有关功能指针支持的最后说明。尽管SWIG通常不允许使用目标语言编写回调函数，但是可以使用类型映射和其他高级SWIG功能来实现。有关类型映射的更多信息，请参见类型映射章节；有关回调的更多信息，请参见各个目标语言章节。“导演”功能可用于将C / C ++ 回调到目标语言，请参阅回调到目标语言。

5.5结构和联合

本节描述了处理ISO C结构和联合声明时SWIG的行为。下一部分将介绍处理C ++的扩展。

如果SWIG遇到结构或联合的定义，它将创建一组访问器函数。尽管SWIG不需要结构定义来构建接口，但是提供定义可以访问结构成员。SWIG生成的访问器函数仅获取指向对象的指针，并允许访问单个成员。例如，声明：

```
结构向量{
  double x，y，z;
}
```

被转换为以下访问器函数集：

```
double Vector_x_get（struct Vector * obj）{
  return obj-> x;
}
double Vector_y_get（struct Vector * obj）{
  return obj-> y;
}
double Vector_z_get（struct Vector * obj）{
  return obj-> z;
}
void Vector_x_set（struct Vector * obj，double value）{
  obj-> x = value;
}
void Vector_y_set（struct Vector * obj，double value）{
  obj-> y = value;
}
void Vector_z_set（struct Vector * obj，double value）{
```

```
  obj-> z = value;
}
```

另外，如果在接口中未定义默认的构造函数和析构函数，SWIG将创建默认构造函数和析构函数。例如：

```
struct Vector * new_Vector () {
  return (Vector *) calloc (1, sizeof (struct Vector) ) ;
}
void delete_Vector (struct Vector * obj) {
  free (obj) ;
}
```

使用这些低级访问器函数，可以使用以下代码从目标语言中对对象进行最少的操作：

```
v = new_Vector ()
Vector_x_set (v, 2)
Vector_y_set (v, 10)
Vector_z_set (v, -5)
...
delete_Vector (v)
```

但是，大多数SWIG的语言模块也提供了更方便的高级界面。继续阅读。

5.5.1 Typedef和结构

SWIG支持以下在C程序中很常见的构造：

```
typedef struct {
  double x, y, z;
向量;
```

遇到此问题时，SWIG会假定该对象的名称为“ Vector”，并像以前一样创建访问器函数。唯一的区别是，使用typedef允许SWIG 在其生成的代码上删除 struct关键字。例如：

```
double Vector_x_get (Vector * obj) {
  return obj-> x;
}
```

如果像这样使用两个不同的名称：

```
typedef struct vector_struct {
  double x, y, z;
向量;
```

使用名称Vector代替vector_struct，因为这是更典型的C编程风格。如果稍后在接口中定义的声明使用类型struct vector_struct，则SWIG知道这与Vector相同， 并且它将生成适当的类型检查代码。

5.5.2字符串和结构

涉及字符串的结构需要格外小心。SWIG假定使用malloc（）动态分配了char *类型的所有成员，并且它们都是NULL终止的ASCII字符串。修改此类成员后，将释放先前的内容，并分配新的内容。例如 ：

```
%module mymodule
...
struct Foo {
  char * name;
  ...
}
```

这将导致以下访问器功能：

```
char * Foo_name_get (Foo * obj) {
  return Foo-> name;
```

```
}

char * Foo_name_set（Foo * obj，char * c）{
  if（obj-> name）
    free（obj-> name）;
  obj->名称=（char *）malloc（strlen（c）+1）;
  strcpy（obj-> name，c）;
  返回obj-> name;
}
```

如果此行为与应用程序中的行为不同，则可以使用SWIG“ memberin”类型映射进行更改。有关更多详细信息，请参见“类型映射”一章。

注意：如果使用-c ++选项，则new和delete用于执行内存分配。

5.5.3数组成员

数组可能显示为结构的成员，但它们将是只读的。SWIG将编写一个访问器函数，该函数将指针返回数组的第一个元素，但不会编写更改数组本身内容的函数。当检测到这种情况时，SWIG可能会生成如下警告消息：

```
接口：i.116。警告。数组成员将为只读
```

为了消除警告消息，可以使用类型映射，但这将在后面的章节中进行讨论。在许多情况下，警告消息是无害的。

5.5.4结构数据成员

有时，结构将包含本身就是结构的数据成员。例如：

```
typedef struct Foo {
  int x;
} Foo;

typedef struct Bar {
  int y;
  Foo f; / *结构成员* /
} Bar;
```

包装结构成员时，除非将%naturalvar指令用于更像C ++引用的结构（请参阅C ++成员数据），否则将其作为指针处理。成员变量作为指针的访问器有效包装如下：

```
Foo * Bar_f_get（Bar * b）{
  return&b-> f;
}
void Bar_f_set（Bar * b，Foo * value）{
  b-> f = * value;
}
```

这样做的原因有些微妙，但与修改和访问数据成员内部的数据有关。例如，假设你想修改的值FX 一个的酒吧这样的对象：

```
酒吧* b;
b-> fx = 37;
```

将此分配转换为函数调用（将在脚本语言界面中使用）将导致以下代码：

```
酒吧* b;
Foo_x_set（Bar_f_get（b），37）;
```

在此代码中，如果Bar_f_get（）函数将返回Foo而不是 Foo *，则结果修改将应用于f的*副本*，而不是数据成员f本身。显然，这不是您想要的！

应该注意的是，只有在SWIG知道数据成员是结构或类时，才会发生对指针的转换。例如，如果您具有这样的结构，

```
struct Foo {
  WORD w;
};
```

并且对WORD一无所知，那么SWIG将生成如下更普通的访问器函数：

```
字Foo_w_get（Foo * f）{
  return f-> w;
}
void Foo_w_set（FOO * f，WORD value）{
  f-> w = value;
}
```

兼容性说明： SWIG-1.3.11和更早版本将所有非原始成员数据类型转换为指针。从SWIG-1.3.12开始，*仅*当已知数据类型为结构，类或联合时，才进行此转换。这不太可能破坏现有代码。但是，如果您需要告诉SWIG未声明的数据类型实际上是一个结构，则只需使用正向结构声明即可，例如“ struct Foo;”。。

5.5.5 C构造函数和析构函数

在包装结构时，通常有一种用于创建和销毁对象的机制。如果您什么都不做，SWIG将使用malloc（）和free（）自动生成用于创建和销毁对象的函数。注意： 仅当在C代码上使用SWIG时（即，命令行中*未*提供-c ++选项时），才适用 malloc（）。C ++的处理方式不同。

如果您不希望SWIG为您的接口生成默认构造函数，则可以使用%nodefaultctor指令或 -nodefaultctor命令行选项。例如：

```
swig -nodefaultctor example.i
```

要么

```
%module foo
...
%nodefaultctor; //不要创建默认的构造函数
...声明...
%clearnodefaultctor; //重新启用默认构造函数
```

如果需要更精确的控制，%nodefaultctor可以有选择地针对单个结构定义。例如：

```
%nodefaultctor Foo; //没有Foo的默认构造函数
...
struct Foo {//没有生成默认的构造函数。
};

struct Bar {//生成默认构造函数。
};
```

由于大多数时候忽略隐式或默认析构函数会导致内存泄漏，因此SWIG始终会尝试生成它们。但是，如果需要，可以使用%nodefaultdtor有选择地禁用默认/隐式析构函数的生成。

```
%nodefaultdtor Foo; //没有Foo的默认/隐式析构函数
...
struct Foo {//没有生成默认的析构函数。
};

struct Bar {//生成默认析构函数。
};
```

兼容性说明：在SWIG-1.3.7之前，SWIG不会生成默认构造函数或析构函数，除非您使用-make_default明确将其打开。但是，似乎大多数用户都希望具有构造函数和析构函数，因此现在已将其启用为默认行为。

注意：还有-nodefault选项和 %nodefault指令，它们会禁用默认或隐式析构函数的生成。这可能会导致目标语言之间的内存泄漏，强烈建议您不要使用它们。

5.5.6将成员函数添加到C结构

大多数语言都提供了一种用于创建类并支持面向对象编程的机制。从C的角度来看，面向对象的编程实际上可以归结为将函数附加到结构的过程。这些功能通常在结构（或对象）的实例上运行。尽管C ++可以自然地映射到这种方案，但尚无直接的机制将其与C代码一起使用。但是，SWIG提供了一个特殊的 %extend指令，它可以将方法附加到C结构，以构建面向对象的接口。假设您有一个带有以下声明的C头文件：

```
/ * file：vector.h * /
...
typedef struct Vector {
  double x，y，z;
向量；
```

通过编写如下的SWIG接口， 可以使Vector看起来很像类：

```
// file: vector.i
%module mymodule
%{
#include " vector.h"
%}

%include " vector.h" // //抓取原始的C头文件
%extend Vector {//将这些函数附加到struct Vector
  Vector ( double x, double y, double z) {
    向量* v;
    v =（向量*）malloc（sizeof（向量））;
    v-> x = x;
    v-> y = y;
    v-> z = z;
    返回v;
  }
  ~Vector () {
    free（$ self）;
  }
  doubleitude () {
    return sqrt（$ self-> x * $ self-> x + $ self-> y * $ self-> y + $ self-> z * $ self-> z）;
  }
  void print () {
    printf（"向量[%g, %g, %g] \ n", $ self-> x, $ self-> y, $ self-> z）;
  }
};
```

注意$ self特殊变量的用法。它的用法与C ++“ this”指针相同，并且在需要访问struct实例时应使用它。还要注意，即使对于C代码，也已使用C ++构造函数和析构函数语法来模拟构造函数和析构函数。尽管与正常的C ++构造函数实现有一个微妙的区别，那就是尽管构造函数声明是按照普通的C ++构造函数声明的，但新构造的对象必须像构造函数声明具有返回值一样返回，在这种情况下为Vector *。

现在，当与Python中的代理类一起使用时，您可以执行以下操作：

```
>>> v = Vector（
3, 4, 0 ）＃创建一个新矢量>>> print v.magnitude（）＃打印幅度
5.0
>>> v.print（）＃打印出来
[ 3, 4, 0 ]
>>> del v＃销毁它
```

所述%延伸指令还可以在载体结构的定义内使用。例如：

```
// file: vector.i
%module mymodule
%{
#include " vector.h"
%}

typedef struct Vector {
  double x, y, z;
  %extend {
    Vector（double（double x, double y, double z））{...}
    ~Vector（）{...}
    ...
  }
}向量;
```

注意，%extend可以用来访问外部编写的函数，只要它们遵循本示例中使用的命名约定即可：

```
/ *文件: vector.c * /
/ *矢量方法* /
#include " vector.h"
矢量* new_Vector（double x, double y, double z）{
  Vector * v;
  v =（向量*）malloc（sizeof（向量））;
  v-> x = x;
  v-> y = y;
  v-> z = z;
```

```
  返回v;
}
void delete_Vector（Vector * v）{
  free（v）;
}

double Vector_magnitude（Vector * v）{
  return sqrt（v-> x * v-> x + v-> y * v-> y + v-> z * v-> z）;
}

// File：vector.i
//接口文件
%module mymodule
%{
#include“ vector.h”
%}

typedef struct Vector {
  double x，y，z;
  %extend {
    Vector（int，int，int）; //这会调用new_Vector（）〜Vector
     （）; //这会调用delete_Vector（）doubleitude
     （）; //这将调用Vector_magnitude（）
    ...
  }
} Vector;
```

用于%extend的名称应该是该结构的名称，而不是该结构的任何typedef的名称。例如：

```
typedef struct Integer {
  int value;
} Int;
%extend Integer {...} / *正确的名称* /
%extend Int {...} / *错误的名称* /

struct Float {
  float value;
};
typedef struct Float FloatValue;
%extend Float {...} / *正确的名称* /
%extend FloatValue {...} / *错误的名称* /
```

该规则有一个例外，那就是匿名命名结构时，例如：

```
typedef struct {
  double value;
} Double;
%extend Double {...} / *好的* /
```

%extend指令的 一个鲜为人知的功能是它还可用于添加综合属性或修改现有数据属性的行为。例如，假设您要使幅度成为Vector的只读属性， 而不是方法。为此，您可以编写如下代码：

```
//将新属性添加到Vector
%extend Vector {
  const doubleitude;
}
//现在提供Vector_magnitude_get函数的实现
%{
const double Vector_magnitude_get（Vector * v）{
  return（const double）sqrt（v-> x * v-> x + v-> y * v-> y + v -> z * v-> z）;
}
%}
```

现在，出于所有实际目的，幅度将看起来像对象的属性。

也可以使用类似的技术来处理要处理的数据成员。例如，考虑以下接口：

```
typedef struct Person {
  char name [50];
  ...
}人；
```

假设您想确保名称始终为大写，则可以按照以下方式重写接口，以确保在读取或写入名称时都会发生这种情况：

```
typedef struct Person {{
  extend {
    char name [50];
  }
  ...
}人；

%{
#include <string.h>
#include <ctype.h>

void make_upper（char * name）{
  char * c;
  对于（c =名称; * c; ++ c）
    * c =（字符）toupper（（int）* c）;
}

/ *强制使用大小写的set / get函数的特定实现* /

char * Person_name_get（Person * p）{
  make_upper（p-> name）;
  返回p-> name;
}

void Person_name_set（Person * p，char * val）{
  strncpy（p-> name，val，50）;
  make_upper（p->名称）;
}
%}
```

最后，应该强调的是，即使 可以使用%extend添加新的数据成员，这些新成员也不需要在对象中分配额外的存储（例如，它们的值必须完全根据结构的现有属性来合成，或者在其他地方获得）。

兼容性注意： 在%扩展指令是在一个新的名字%addmethods指令。由于%addmethods不仅可以用于扩展方法，还可以用于扩展结构，因此已选择了更合适的指令名称。

5.5.7嵌套结构

有时，C程序将包含以下结构：

```
typedef struct Object {
  int objtype;
  联合{
    int ivalue;
    双值
    char * strvalue;
    无效* ptrvalue;
  } intRep;
}对象；
```

当SWIG遇到此问题时，它将执行结构拆分操作，将声明转换为以下内容：

```
typedef union {
  int ivalue;
  双值
  char * strvalue;
  无效* ptrvalue;
} Object_intRep;

typedef struct Object {
  int objType;
  Object_intRep intRep;
}对象；
```

然后，SWIG将创建一个Object_intRep结构，以在接口文件中使用。将为这两种结构创建访问器功能。在这种情况下，将创建如下函数：

```
Object_intRep * Object_intRep_get(Object * o) {
  return(Object_intRep *) &o-> intRep;
}
int Object_intRep_ivalue_get(Object_intRep * o) {
  return o-> ivalue;
}
int Object_intRep_ivalue_set(Object_intRep * o, int value) {
  return(o-> ivalue = value);
}
double Object_intRep_dvalue_get(Object_intRep * o) {
  return o-> dvalue;
}
等...
```

尽管此过程有些麻烦，但它的工作方式与您在目标脚本语言中所期望的一样，尤其是在使用代理类时。例如，在Perl中：

```
#Perl5脚本，用于访问嵌套成员
$ o = CreateObject(); #以某种方式创建对象
$ o-> {intRep}-> {ivalue} = 7#更改o.intRep.ivalue的值
```

如果您有很多嵌套结构声明，建议在运行SWIG之后再次检查它们。尽管它们很有可能会起作用，但在某些情况下您可能必须修改接口文件。

最后，请注意，嵌套在C ++模式下的处理方式有所不同，请参见嵌套类。

5.5.8关于结构包装的其他注意事项

SWIG不在乎.i文件中结构的声明是否与基础C代码中使用的结构声明完全匹配（嵌套结构的情况除外）。因此，省略有问题的成员或完全省略结构定义都没有问题。如果您乐于传递指针，则无需为SWIG提供结构定义即可完成此操作。

从SWIG1.3开始，对SWIG的代码生成器进行了许多改进。具体地说，即使已经按照诸如此类的高级访问器功能描述了结构访问，

```
double Vector_x_get(Vector * v) {
  return v-> x;
}
```

实际上，大多数生成的代码都直接内联到包装函数中。因此，在生成的包装文件中实际上没有函数Vector_x_get()。例如，当创建一个Tcl模块时，将生成以下函数：

```
静态int
_wrap_Vector_x_get(ClientData clientData, Tcl_Interp * interp,
                   int objc, Tcl_Obj * CONST objv []) {
  结构向量* arg1;
  双重结果;

  如果(SWIG_GetArgs(interp, objc, objv, " p: Vector_x_get self", &arg0,
                   SWIGTYPE_p_Vector) == TCL_ERROR)
    返回TCL_ERROR;
  结果=(双精度)(arg1-> x);
  Tcl_SetObjResult(interp, Tcl_NewDoubleObj((double) result));
  返回TCL_OK;
}
```

该规则的唯一例外是用%extend定义的方法。在这种情况下，添加的代码包含在单独的函数中。

最后，需要注意的是，大多数语言模块都可以选择构建更高级的界面。尽管您可能永远不会使用这里描述的低级接口，但是大多数SWIG的语言模块都以某种方式使用它。

5.6代码插入

有时有必要在SWIG生成的结果包装文件中插入特殊代码。例如，您可能想包括其他C代码以执行初始化或其他操作。有四种常见的插入代码的方法，但是了解SWIG输出的结构首先是很有用的。

5.6.1 SWIG的输出

SWIG创建其输出C / C ++文件时，将其分为五个部分，分别对应于运行时代码，标头，包装函数和模块初始化代码（按此顺序）。

- 开始部分。
  一个占位符，供用户将代码放在C / C ++包装器文件的开头。这最常用于定义在后面的部分中使用的预处理器宏。

- 运行时代码。
  该代码是SWIG的内部代码，用于包括模块其余部分使用的类型检查和其他支持功能。
- 标头部分。
  这是%{...%}指令中包含的用户定义支持代码。通常，它由头文件和其他帮助函数组成。
- 包装代码。
  这些是SWIG自动生成的包装。
- 模块初始化。
  SWIG生成的函数，用于在加载时初始化模块。

5.6.2代码插入块

使用%insert指令可以将代码块插入生成的代码的给定部分。可以通过以下两种方式之一使用它：

```
%insert（“ section”）“文件名”
%insert（“ section”）%{...%}
```

第一种会将给定文件名中文件的内容转储到命名部分。第二个代码将花括号之间的代码插入到命名部分中。例如，以下代码将代码添加到运行时部分：

```
%insert（“运行时”）%{
  ...运行时部分中的代码...
%}
```

有5个部分，但是，某些目标语言会添加其他部分，其中一些会导致代码生成到目标语言文件中，而不是C / C ++包装器文件中。在目标语言章节中提供了这些文档。以代码节命名的宏可用作附加指令，这些宏指令通常代替%insert使用。例如，使用%runtime代替%insert（“ runtime”）。生成的C / C ++包装器文件中的有效节和节的顺序如下所示：

```
%begin%{
  ...开始部分中的代码...
%}

%runtime%{
  ...运行时部分中的代码...
%}

%header%{
  ...标头部分中的代码...
%}

%wrapper %{
  ...包装部分中的代码...
%}

%init%{
  ... init部分中的代码...
%}
```

裸%{...%}指令是与%header%{...%}相同的快捷方式 。

在%开始部分是有效空的，因为它只是默认情况下包含痛饮旗帜。提供此部分是为了让用户在生成任何其他代码之前将代码插入包装文件的顶部。代码插入块中的所有内容均原样复制到输出文件中，而SWIG不会对其进行解析。大多数SWIG输入文件都有至少一个这样的块，以包含头文件并支持C代码。可以根据需要将其他代码块放置在SWIG文件中的任何位置。

```
%module mymodule
%{
#include“ my_header.h”
%}
...在此处声明函数
%{

void some_extra_function（）{
  ...
}
%}
```

代码块的常见用法是编写“辅助”功能。这些是专门用于构建接口的功能，但通常对于普通C程序而言不可见。例如 ：

```
%{
/ *创建一个新向量* /
静态向量* new_Vector（）{
```

```
  return (向量*) malloc (sizeof (Vector));
}

%}
//现在将其包装
Vector * new_Vector ();
```

5.6.3内联代码块

由于编写辅助函数的过程相当普遍，因此有一种特殊的内联代码块形式，其用法如下：

```
%inline%{
/ *创建一个新向量* /
向量* new_Vector () {
  return (向量*) malloc (sizeof (Vector));
}
%}
```

这与写作相同：

```
%{
/ *创建一个新向量* /
向量* new_Vector () {
  return (向量*) malloc (sizeof (Vector));
}
%}

/ *创建一个新向量* /
向量* new_Vector () {
  return (向量*) malloc (sizeof (Vector));
}
```

换句话说，%inline指令将逐字后面的所有代码插入接口文件的头部分。然后，SWIG预处理程序和解析器都会对代码进行解析。因此，以上示例仅使用一个声明创建了一个新命令new_Vector。由于%inline%{...%}块中的代码同时提供给C编译器和SWIG，因此在%{...%}块中包含任何SWIG指令都是非法的。

注意：当SWIG解析此代码时，通常的SWIG C预处理程序规则适用于%apply块中的代码。例如，如前所述，默认情况下，SWIG的C预处理程序不遵循#include指令。

5.6.4初始化块

如果%init节中包含代码，则将其直接复制到模块初始化函数中。例如，如果您需要在模块加载时执行一些额外的初始化，则可以编写以下代码：

```
%init%{
  init_variables ();
%}
```

请注意，某些语言后端（例如C＃或Java）没有任何初始化功能，因此您应该定义一个全局对象，对其执行必要的初始化：

```
%init%{
  静态结构MyInit {MyInit () {init_variables (); }} myInit;
%}
```

5.7接口构建策略

本节介绍了使用SWIG构建接口的一般方法。与特定脚本语言有关的细节可以在后面的章节中找到。

5.7.1为SWIG准备C程序

SWIG不需要修改您的C代码，但是如果您向它提供原始C头文件或源代码的集合，则结果可能不是您期望的-实际上，它们可能很糟糕。您可以按照以下步骤为C程序创建接口：

- 确定要包装的功能。可能没有必要访问C程序的每个功能-因此，稍加周全就可以大大简化最终的脚本语言界面。C头文件是查找要包装的东西的特别好资源。
- 创建一个新的界面文件来描述程序的脚本语言界面。
- 将适当的声明复制到接口文件中，或使用SWIG的%include指令处理整个C源/头文件。
- 确保接口文件中的所有内容均使用ISO C / C ++语法。

- 确保接口文件中所有必需的` typedef ’声明和类型信息均可用。特别是，请确保按照C / C ++编译器的要求以正确的顺序指定类型信息。最重要的是，在使用类型之前先定义它！如果需要，AC编译器会告诉您是否不提供完整类型信息，而SWIG通常不会发出警告或错误，因为它设计为在没有完整类型信息的情况下可以工作。但是，如果未正确指定类型信息，则包装器可能不是最佳选择，甚至会导致无法编译的C / C ++代码。
- 如果您的程序具有main（）函数，则可能需要重命名它（继续阅读）。
- 运行SWIG并进行编译。

尽管这听起来可能很复杂，但是一旦掌握了这一过程，该过程将变得相当容易。

在构建接口的过程中，SWIG可能会遇到语法错误或其他问题。解决此问题的最佳方法是简单地将有问题的代码复制到单独的接口文件中并进行编辑。但是，SWIG开发人员一直在努力改进SWIG解析器-您应将解析错误报告给swig-devel邮件列表或 SWIG Bug跟踪器。

5.7.2 SWIG接口文件

使用SWIG的首选方法是生成单独的接口文件。假设您具有以下C头文件：

```
/ *文件：header.h * /

#include <stdio.h>
#include <math.h>

extern int foo（double）;
extern双杠（int，int）;
extern void dump（FILE * f）;
```

该头文件的典型SWIG接口文件如下所示：

```
/ *文件：interface.i * /
%module mymodule
%{
#include“ header.h”
%}
extern int foo（double）;
extern双杠（int，int）;
extern void dump（FILE * f）;
```

当然，在这种情况下，我们的头文件非常简单，因此我们可以使用更简单的方法并使用如下接口文件：

```
/ *文件：interface.i * /
%module mymodule
%{
#include“ header.h”
%}
%include“ header.h”
```

这种方法的主要优点是在将来头文件发生更改时对接口文件的维护最少。在更复杂的项目中，由于维护成本较低，因此包含大量%include和#include语句的接口文件是最常见的接口文件设计方法之一。

5.7.3为什么要使用单独的接口文件？

尽管SWIG可以解析许多头文件，但更常见的是编写一个特殊的.i文件来定义包的接口。您可能要这样做的原因有几个：

- 几乎没有必要在大型程序包中访问每个功能。在脚本环境中，许多C函数可能几乎没有用。因此，为什么要包裹它们？
- 单独的接口文件为提供有关如何构造接口的更精确规则的机会。
- 接口文件可以提供更多的结构和组织。
- SWIG无法解析出现在头文件中的某些定义。使用单独的文件可以消除或解决这些问题。
- 接口文件提供了接口的更精确定义。想要扩展系统的用户可以转到界面文件，立即查看可用的文件，而不必从头文件中挖掘出来。

5.7.4获取正确的头文件

有时，为了使SWIG生成的代码正确编译，有必要使用某些头文件。确保使用%{%}块包含某些头文件，如下所示：

```
%module graphics
%{
#include <GL / gl.h>
#include <GL / glu.h>
%}

//将其余的声明放在此处
...
```

5.7.5如何处理main（）

如果您的程序定义了main（）函数，则可能需要摆脱它或重命名它才能使用脚本语言。大多数脚本语言定义了自己的main（）过程，而该过程被调用。main（）在进行动态加载时也没有任何意义。有几种方法可以解决main（）冲突：

- 完全摆脱main（）。
- 将main（）重命名为其他名称。您可以通过使用-Dmain = oldmain之类的选项编译C程序来实现。
- 不使用脚本语言时，使用条件编译仅包括main（）。

摆脱main（）可能会导致程序潜在的初始化问题。要解决此问题，您可以考虑编写一个名为program_init（）的特殊函数，该函数在启动时会初始化您的程序。然后，可以从脚本语言中调用此函数作为第一个操作，或者在加载SWIG生成的模块时调用此函数。

作为一般说明，许多C程序仅使用main（） 函数来解析命令行选项和设置参数。但是，通过使用脚本语言，您可能正在尝试创建更具交互性的程序。在许多情况下，旧的 main（）程序可以完全由Perl，Python或Tcl脚本替换。

注意：在某些情况下，您可能倾向于为main（）创建脚本语言包装。如果这样做，编译可能会起作用，并且您的模块甚至可能正确加载。唯一的麻烦是，当您调用 main（）包装器时，您会发现它实际上调用了脚本语言解释器本身的 main（）！此行为是动态链接器中使用的符号绑定机制的副作用。最重要的是：不要这样做。