第21章

编写大型程序：包和符号

在第4章里，我讨论了Lisp读取器是如何将文本名字转化成用来传递给求值器的对象的，它们是一种称为符号的对象。实践表明，拥有专门用来表示各种名字的内置数据类型，对于多种编程任务都是有用的。[1]不过，这并不是本章的主题。在本章里，我将讨论一个处理名字方面的问题：如何避免彼此无关的人开发的代码出现名字冲突。

举个例子，假设你正在编写一个程序并决定在其中使用一个第三方库。你不想为了避免这些名字与你自己的程序中所使用的名字相冲突，而要知道那个库中的每一个函数、变量、类和宏的名字。你更希望这个库中的大多数名字和你程序中的名字是彼此无关的，哪怕是它们碰巧使用了相同的文本表示。同时，你希望这个库所定义的特定名字就是被用来访问的——这些名字构成了它的公共API，你会在自己的程序中用到它们。

在Common Lisp中，这一名字空间问题最后归结到，如何控制读取器将文本名称转化成符号这个问题上：如果你希望相同的名字可以被求值器同等看待，那么需要确保读取器使用相同的符号来表示每个名字。同样地，如果你希望两个名字被视作是不同的，甚至在它们刚好具有相同的文本名字时，那么就需要让读取器分别创建不同的符号来表示它们。

21.1 读取器是如何使用包的

在第4章里，我简单讨论了Lisp读取器是如何将名字转化成符号的，但我刻意忽略了大部分细节。现在是时候从更近的角度来看看这个过程中究竟发生什么了。

我先来描述读取器所理解的名字的语法，以及该语法和包之间的关系。目前，你可以将包想像成一个字符串和符号的映射表。如同你在下一节里将会看到的，实际的映射过程比一个简单的查找表稍微灵活一些，但这对读者来说通常意义不大。每个包也都有一个名字，该名字可用来通过函数`FIND-PACKAGE`找到对应的包。

读取器用来访问包中的名字-符号映射的两个关键函数是`FIND-SYMBOL`和`INTERN`。这两个函数都接受一个字符串和一个可选的包。当包未指定时，包参数默认为全局变量`*PACKAGE*`的值，也称为当前包。

---

① 依赖于符号数据类型的编程，被恰如其名地称为符号计算。它和基于数值的编程正好相反。一个所有程序员都应当熟悉的、主要的符号计算程序的例子是编译器，它将一个程序的文本视为符号数据并将其转化成一种新的形式。

FIND-SYMBOL在包中查找名为给定字符串的符号并将其返回，如果没有找到任何符号则返回**NIL**。**INTERN**也会返回一个已有的符号，否则它会创建一个以该字符串命名的新符号并将其添加到包里。

你所使用的大多数名字都是非限定的（unqualified），就是说名字里不带冒号。当读取器读到这样的名字时，它先将名字中所有未转义的字母转换成大写形式，然后将得来的字符串传给**INTERN**，从而将该名字转化成一个符号。这样，每当读取器读到相同的包里面相同的名字时，它都将得到相同的符号对象。这是很重要的，因为求值器使用符号的对象标识来决定一个给定函数所指向的函数、变量或其他程序元素。因此，诸如`(hello-world)`这样的表达式得以调用一个特定的`hello-world`函数的原因就在于，读取器在读取对该函数的调用和定义该函数的**DEFUN**形式时返回了相同的符号。

含有单冒号或双冒号的名字是包限定的名字。当读取器读取包限定的名字时，它将名字在冒号处拆开，前一部分作为包的名字，后一部分作为符号的名字。读取器查找适当的包并用它来将符号名转化成一个符号对象。

只含单冒号的名字必须指向一个外部符号（external symbol）——一个被包导出（export）作为公共接口来使用的符号。如果命名的包不含有一个给定名字的符号，或是含有该符号但并未导出，那么读取器会产生一个错误。含双冒号的名字可以指向命名包中的任何符号，尽管这通常不是个好主意——导出符号的集合定义了一个包的公共接口，而如果你不遵守包作者关于哪些名字是公开的而哪些名字是私有的约定，那么在使用时肯定会遇到麻烦。另一方面，有时一个包的作者可能忽略了导出一个确实应当开放给公众的符号。在这种情况下，含双冒号的名字可以让你无需等待该包的下一个版本的发布即可完成手头的工作。

读取器理解的符号语法的另外两点分别是关键字符号和未进入（uninterned）包的符号。关键字符号在书写上以一个冒号开始。这类符号在名为**KEYWORD**的包中创建并自动导出。更进一步，当读取器在**KEYWORD**包中创建一个符号时，它也会定义一个以该符号作为其名字和值的常量。这就是你可以在参数列表中使用关键字而无需引用它们的原因——当它们出现在一个值的位置上时，它们求值到自身。这样：

```
(eql ':foo :foo) → T
```

和所有符号一样，关键字符号的名字在它们被创建之前就被读取器全部转换成大写形式了。名字中不包含前导冒号。

```
(symbol-name :foo) → "FOO"
```

未进入的符号在写法上以“`#:`”开始。这些名字（在去掉“`#:`”后）被正常地转换成大写形式并被转化成符号，但这些符号还没有进入任何包，每当读取器读到一个带有“`#:`”的名字时，它都会创建一个新的符号。这样：

```
(eql '#:foo '#:foo) → NIL
```

一般不需要自行书写这种语法，但有时当你打印一个含有由**GENSYM**所返回的符号的S-表达

式时就会看到它。

```
(gensym) → #:G3128
```

21.2　包和符号相关的术语

如同我先前提到的，包所实现的从名字到符号的映射比简单的查找表更加灵活。在核心层面上，每一个包都含有一个从名字到符号的查找表，但还有其他几种方式通过一个给定包中的非限定名字可以访问一个符号。为了更有意义地谈论这些方法，你需要了解一些词汇表。

首先，所有可在一个给定包中通过**FIND-SYMBOL**找到的符号被称为在该包中是**可访问的**(accessible)。换句话说，一个包中可访问的符号是，该包为当前包时非限定名字指向的符号。

可以通过两种方式访问一个符号。前一种方式要求包的名字-符号表中含有该符号的项，这时我们称该符号**存在**（present）于该包中。当读取器让一个新符号进入一个包时，该符号会添加到包的名字-符号表中。该符号首先停留的包称作该符号的**主包**（home package）。

另一种方式是当某个包继承一个符号时，该符号在该包中就是可访问的。一个包通过**使用**(use)其他包来继承这些包中的符号。在被使用的包中，只有**外部**（external）符号才能被继承。可以通过在包中**导出**（export）一个符号来使其成为外部符号。导出操作除了可以使包的其他使用者继承符号以外，还可以使其能够通过带有单冒号的限定名称来引用，如同你在上一节里所看到的那样。

为了保证从名字到符号的映射的确定性，包系统只允许每个名字在给定的包中指向单一符号。这就是说，一个包不能同时有一个本身定义的符号和一个同名的继承得来的符号，或是同时从不同的包中继承两个具有相同名字的不同符号。不过，你可以通过使其中一个可访问的符号成为**隐蔽**（shadow）符号来解决冲突，这可以使其他同名的符号变得不可访问。每一个包都在它们的名字-符号列表之外维护了一个隐蔽符号的列表。

一个已有的符号可以通过将其添加到另一个包的名字-符号表中，来**导入**（import）到这个包。这样，同一个符号可以同时存在于多个包中。有时，导入符号只是因为希望它们在被导入的包中可以直接访问而无需使用它们的主包。其他时候，导入符号则是因为只有存在的符号才可以被导出或是成为隐蔽符号。举个例子，如果一个包需要使用两个带有同名外部符号的包，那么其中一个符号必须导入该包，以便可添加到该包的隐蔽符号列表并使另一个符号成为不可访问的。

最后，一个已有的符号可以从一个包中**退出**（unintern），这会导致它被清除出名字-符号表，并且如果它是一个隐蔽符号，也会被清除出隐蔽符号列表。你可能想让一个符号从一个包中退出，从而消除该符号和一个来自你想使用的包中的外部符号之间的冲突。一个不存在于任何包中的符号称为**未进入**（uninterned）的符号，它不能被读取器读取，并且将采用`#:foo`语法进行打印。

21.3　三个标准包

在下一节里，我将向你展示如何定义你自己的包，包括如何让一个包使用另一个包，如何导出、隐蔽和导入符号。不过首先让我们看一些你已经在使用的包。最初启动Lisp时，***PACKAGE***

的值通常是COMMON-LISP-USER包，有时也叫做CL-USER。[1]CL-USER使用了包COMMON-LISP，后者导出了语言标准定义的所有名字。因此，当在REPL中键入一个表达式时，所有诸如标准函数、宏、变量之类的名字都将转化成从COMMON-LISP包中导出的符号，而其他名字进入到COMMON-LISP-USER包中。例如，名字***PACKAGE***是从COMMON-LISP包中导出的，如果你想要查看***PACKAGE***的值，可以输入：

```
CL-USER> *package*
#<The COMMON-LISP-USER package>
```

这是因为COMMON-LISP-USER使用了COMMON-LISP，或者也可以使用一个包限定的名字：

```
CL-USER> common-lisp:*package*
#<The COMMON-LISP-USER package>
```

甚至可以使用COMMON-LISP的昵称CL：

```
CL-USER> cl:*package*
#<The COMMON-LISP-USER package>
```

但是*X*不是COMMON-LISP中的符号，因此如果你输入：

```
CL-USER> (defvar *x* 10)
*X*
```

那么读取器将把**DEFVAR**作为COMMON-LISP中的符号来读取，同时把*X*作为COMMON-LISP-USER中的符号来读取。

REPL不能在COMMON-LISP包中启动，因为你不能在这个包中添加新符号。COMMON-LISP-USER是作为一个“模板”包来提供的，在其中你可以创建自己的名字，同时还能轻松地访问COMMON-LISP的所有符号。[2]通常情况下，你定义的所有包都将使用COMMON-LISP，因此不需要写出类似下面的代码：

```
(cl:defun (x) (cl:+ x 2))
```

第三个标准包是**KEYWORD**包，这个包被Lisp读取器用来创建以冒号开始的名字。这样，你也可以使用显式的包限定方式来引用任何关键字符号：

---

① 每一个包都有一个正式名字以及零个或多个昵称（nickname），昵称可用在任何需要用到包名的地方，例如带有包限定的名字，或是在一个**DEFPACKAGE**或**IN-PACKAGE**形式中引用那个包。

② COMMON-LISP-USER也允许提供对由从其他语言实现所定义的包导出的符号的访问。尽管这在本意上是为了给用户提供方便——它使得实现相关的功能易于访问，但它也给Lisp初学者带来许多疑惑：Lisp会抱怨重定义某些语言标准并未涉及的符号名。要想看到在一个特定的实现中COMMON-LISP-USER都从哪些包中继承了符号，可以在REPL中求值下列表达式：

```
(mapcar #'package-name (package-use-list :cl-user))
```

而要想查出一个符号最初来源于哪个包，可以求值下列表达式：

```
(package-name (symbol-package 'some-symbol))
```

同时把some-symbol替换成你想要的符号。例如：

```
(package-name (symbol-package 'car)) → "COMMON-LISP"
(package-name (symbol-package 'foo)) → "COMMON-LISP-USER"
```

继承自实现定义的包的符号将返回一些其他的值。

```
CL-USER> :a
:A
CL-USER> keyword:a
:A
CL-USER> (eql :a keyword:a)
T
```

21.4　定义你自己的包

使用COMMON-LISP-USER包对于在REPL中进行尝试是好的，不过一旦你开始编写实际的程序就会需要定义新的包，这样不同的程序可以加载到同一个Lisp环境中，而不会破坏彼此的名字。而当你编写可能用于不同环境中的库时，你会想要定义分开的包并导出那些构成了库的公共API的符号。

尽管如此，在开始定义包之前，理解包无法做到的一件事是很重要的。包无法提供对于谁可以调用什么函数或访问什么变量的直接控制。它们只提供对于名字空间的基本控制，这是通过控制读取器将文本名字转换成符号对象来完成的。但在后面求值器中，当符号被解释成一个函数、变量或其他任何东西的名字时，包机制就无能为力了。因此，谈论把一个函数或变量从一个包中导出是没有意义的。你可以导出符号以便特定的名字更加易于访问，但包系统并不允许你限制这些名字如何被使用。[①]

记住上述这件事，你可以开始学习如何定义包并将它们捆绑在一起了。你可以通过宏**DEFPACKAGE**来定义新的包，在创建包的同时还能指定它使用哪些包，导出哪些包，以及它从其他包里导入什么符号，还可以创建隐蔽符号来解决冲突。[②]

假设你正在使用包来编写一个将E-mail消息组织进一个可搜索数据库的程序，我将在此背景下描述所有有关的选项。这个程序是完全假想出来的，包括我将引用的其他库在内。要点在于，查看如何组织用于这样一个程序的包。

所需的第一个包是提供了整个应用程序命名空间的包，你需要命名你的函数、变量，等等，而无需担心和不相关的代码产生命名冲突。因此你最好用**DEFPACKAGE**定义一个新的包。

如果应用程序写得足够简单，没有用到超出语言本身所能提供的库，那么你可以定义一个如下的简单包：

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp))
```

这段代码定义了一个包，名为COM.GIGAMONKEYS.EMAIL-DB，其继承了由COMMON-LISP包

① 这与Java的包系统不同，Java的包系统在提供类的名字空间的同时还引入了Java的访问控制机制。非Lisp语言中包系统最接近Common Lisp的语言是Perl。

② **DEFPACKAGE**进行的所有处理也都可以通过管理包对象的函数来完成。尽管如此，由于一个包在通常情况下都要在使用前被完全定义，所以这些函数很少用到。另外，**DEFPACKAGE**可以采用正确的顺序来完成所有的包管理操作。例如，**DEFPACKAGE**可以在使用那些用到的包之前将有关符号添加到隐蔽符号列表中。

导出的所有符号。[1]

事实上，有几种选择用于表示包的名字，如同你将看到的，**DEFPACKAGE**中符号的名字也有几种表示方法。包和符号都是用字符串来命名的。不过，在**DEFPACKAGE**形式中，可以使用字符串描述符（string designator）来指定包和符号的名字。字符串描述符要么是一个字符串，代表其自身；要么是一个符号，代表其名字；要么是一个字符，代表一个含有该字符的单字符串。像上面的**DEFPACKAGE**那样使用关键字符号，是一种允许把名字书写成小写字母的常用风格，读取器将把名字转换成大写形式。也可以用字符串来书写**DEFPACKAGE**，若这样就需要全部使用大写形式。事实上，由于读取器的大小写转换约定，多数符号和包真正的名字都是大写的。[2]

```
(defpackage "COM.GIGAMONKEYS.EMAIL-DB"
  (:use "COMMON-LISP"))
```


你也可以使用非关键字符号——**DEFPACKAGE**中的名字不会被求值，但是随后读取**DEFPACKAGE**形式的操作将使这些符号进入到当前的包中，这在某种程度上泄露了名字空间，并可能也会在以后你试图使用该包时带来问题。[3]

为了读取这个包中的代码，你需要使用**IN-PACKAGE**宏来使其成为当前包：

```
(in-package :com.gigamonkeys.email-db)
```

如果你在REPL中输入这个表达式，它将改变***PACKAGE***的值，从而影响REPL读取后续表达式的方式，直到你通过另一个**IN-PACKAGE**调用来改变它。类似地，如果你在用**LOAD**加载的文件或**COMPILE-FILE**编译的文件中包含了一个**IN-PACKAGE**，那么它将改变当时的包，从而影响文件中后续表达式的读取方式。[4]

通过将当前包设置为COM.GIGAMONKEYS.EMAIL-DB包，除了那些继承自COMMON-LISP的名字以外，你几乎可以使用任何你想要使用的名字来实现任何目的。这样，你可以定义一个新的、与先前定义在**COMMON-LISP-USER**中的函数共存的hello-world函数。这是已有函数的行为：

```
CL-USER> (hello-world)
hello, world
NIL
```

① 在许多Lisp实现里，如果你只是使用COMMON-LISP包，那么:use子句是可选的。如果省略了它，包将自动从一个由具体实现所定义的包列表中继承名字，而这个包列表通常都包括COMMON-LISP。尽管如此，如果总是显式地指定你想要使用的包列表，那么代码将会变得更加具有可移植性。那些不愿意打字的人可以使用包的昵称，从而将其写成(:use :cl)。

② 使用关键字来代替字符串还有另一个优点：Allegro支持一种“现代模式”的Lisp，其中读取器并不做大小写转换，并且在带有大写名称的COMMON-LISP包以外，它还提供了一个使用小写字母的common-lisp包。严格来讲，这种Lisp并不符合Common Lisp标准，因为所有标准中定义的名字都是被定义成大写的。但如果你使用关键字符号来编写**DEFPACKAGE**形式，那么它将可以同时工作在Common Lisp和与之相近的另一种模式下。

③ 一些人使用“#:”语法的未进入的符号来代替关键字符号。这通过在关键字包中未进入任何符号来节省一小部分内存，在**DEFPACKAGE**通过它实现（或代码展开）之后，符号会变成垃圾。然而，差异是如此微小，以至于这被归结为美学范畴。

④ 使用**IN-PACKAGE**而不是仅仅用**SETF**来修改***PACKAGE***的原因在于，**IN-PACKAGE**可以展开成在文件被**COMPILE-FILE**编译时和文件加载时都会执行的代码，从而在编译期就可以改变读取器读取文件其余部分的方式。

现在你可以用**IN-PACKAGE**切换到新的包上。[①]注意提示符的改变——具体的形式取决于开发环境，但在SLIME中默认提示符由包名的简化版本构成。

```
CL-USER> (in-package :com.gigamonkeys.email-db)
#<The COM.GIGAMONKEYS.EMAIL-DB package>
EMAIL-DB>
```

你可以在这个包里定义一个新的hello-world：

```
EMAIL-DB> (defun hello-world () (format t "hello from EMAIL-DB package~%"))
HELLO-WORLD
```

然后用如下方式测试它：

```
EMAIL-DB> (hello-world)
hello from EMAIL-DB package
NIL
```

现在切换回CL-USER：

```
EMAIL-DB> (in-package :cl-user)
#<The COMMON-LISP-USER package>
CL-USER>
```

旧的函数行为没有被干扰：

```
CL-USER> (hello-world)
hello, world
NIL
```

21.5　打包可重用的库

尽管工作在E-mail数据库上，但你可能需要编写几个与存取文本相关而与E-mail本身无关的函数。你可能意识到这些函数会对其他程序有用，并且决定将它们重新打包成一个库。你应当定义一个新的包，这次将导出一些特定的名字，从而使其对于其他包可见。

```
(defpackage :com.gigamonkeys.text-db
  (:use :common-lisp)
  (:export :open-db
           :save
           :store))
```

你再次使用了COMMON-LISP包，因为你需要在COM.GIGAMONKEYS.TEXT-DB中访问标准函数。:export子句指定了COM.GIGAMONKEYS.TEXT-DB外部的名字，这些名字可以被所有使用它的包直接访问。因此，在定义了这个包以后，你可以将主应用程序包的定义作如下改变：

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db))
```

① 在SLIME的REPL缓冲区里，你也可以使用REPL快捷键来改变当前包。键入一个逗号，然后在Command:提示下输入change-package。

现在COM.GIGAMONKEYS.EMAIL-DB中编写的代码可以同时使用非限定名字来引用COMMON-LISP和COM.GIGAMONKEYS.TEXT-DB中导出的符号。所有其他的名字都将继续直接进入COM.GIGAMONKEYS.EMAIL-DB包中。

21.6 导入单独的名字

现在假设你找到了一个可以处理E-mail消息的第三方函数库。该库的API中所使用的名字是从COM.ACME.EMAIL包中导出的，因此你可以通过使用这个包来轻松获得对这些名字的访问权限。但假设你只需要用到这个库的一个函数，且其他的导出符号跟你已经使用（或计划使用）的符号有冲突。[1]在这种情况下，你可以使用**DEFPACKAGE**中的:import-from子句只导入所需要的那一个符号。例如，如果想要使用的函数的名字为parse-email-address，那么可以把**DEFPACAKGE**作如下改写：


```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db)
  (:import-from :com.acme.email :parse-email-address))
```

现在COM.GIGAMONKEYS.EMAIL-DB包中任何代码里出现的parse-email-address都将被读取为COM.ACME.EMAIL中的同名符号。如果需要从单个包中导入多个符号，那只需在单个:import-from子句中的包名之后包含多个名字即可。一个**DEFPACKAGE**也可以含有多个:import-from子句，以便从不同的包中分别导入符号。

有时你可能会遇到相反的情况——一个包可能导出了一大堆你想要使用的名字，但还有少数是你不需要的。不用将所有你需要的符号都列在一个:import-from子句中，你可以直接使用这个包，同时把不需要继承的符号放在一个:shadow子句里。例如，假设COM.ACME.TEXT包导出了许多用于文本处理的函数和类的名字，更进一步假设多数这些函数和类都是你想要用在代码中的，但其中一个名字build-index跟你已经使用的名字冲突了。你可以通过隐蔽这个符号将来自COM.ACME.TEXT的build-index符号设置为不可见的。

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index))
```

这个:shadow子句导致创建了一个新的名为BUILD-INDEX的符号，且这个符号直接添加到COM.GIGAMONKEYS.EMAIL-DB的名字-符号映射表中。如果读取器读到了名字BUILD-INDEX，将把它转化成COM.GIGAMONKEYS.EMAIL-DB表中的符号，而不是继承自COM.ACME.TEXT的那个符号了。这个新的符号也会添加到作为COM.GIGAMONKEYS.EMAIL-DB包的一部分的隐蔽符号列

① 在开发过程中，如果你试图使用一个包，它导出了一些与你当前所在包的已有符号同名的符号，那么Lisp将会立即报错并通常会提供给你一个再启动，把所用到的包中的那个符号去掉。关于这个过程的更多细节，请参见21.8节。

表中。所以，如果你以后使用了另一个同样导出了BUILD-INDEX符号的包，包系统将会知道这里没有冲突，也就是说你总是希望使用来自COM.GIGAMONKEYS.EMAIL-DB的符号，而不是从其他包里继承的同名符号。

当你想要使用两个导出了同样名字的包时，一个类似的情形出现了在这种情况下，读取器在读到文本名字时不可能知道你究竟想要使用哪一个继承的符号，你必须通过隐蔽有冲突的名字来消除歧义。如果你根本就不需要使用这个名字，那可以通过:shadow子句将该名字屏蔽掉，从而在包中创建出一个同名的新符号来。但如果你确实想要使用一个继承来的符号，那么你需要通过:shadowing-import-from子句来消除歧义。和:import-from子句一样，:shadowing-import-from子句由一个包名及紧接着需要从那个包中导入的名字所构成。举个例子，如果COM.ACME.TEXT导出了一个名字SAVE，它与从COM.GIGAMONKEYS.TEXT-DB中导出的名字相冲突，那么可以使用如下的**DEFPACKAGE**形式来消除歧义：

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index)
  (:shadowing-import-from :com.gigamonkeys.text-db :save))
```

21.7　打包技巧

前面已经讨论了一些常见情形下用包来管理名字空间的方法，而关于如何使用包的另一层面内容也是值得讨论的——关于如何组织代码来使用不同的包的基本技巧。在本节里，我将讨论一些关于如何组织代码的概括性规则，也即相对于那些通过**IN-PACKAGE**来使用包的代码来说，应该在哪里保存**EFPACKAGE**形式呢？

因为包要被读取器使用，因此一个包必须在加载或编译一个含有切换到该包的**IN-PACKAGE**表达式的文件之前就被定义。包也必须定义在可能用到它的其他**DEFPACKAGE**形式之前。举个例子，如果你打算在COM.GIGAMONKEYS.EMAIL-DB中使用COM.GIGAMONKEYS.TEXT-DB包，那么COM.GIGAMONKEYS.TEXT-DB的**DEFPACKAGE**必须在COM.GIGAMONKEYS.EMAIL-DB的**DEFPACKAGE**之前被求值。

确保包在它们被用到之前总是存在的，最佳方法是把所有的**DEFPACKAGE**定义放在与需要在这些包里读取的源代码分开的文件里。一些人喜欢针对每个单独的包都创建一个形如foo-package.lisp的文件，而另一些人则创建单个packages.lisp来包含一组相关的包的所有**DEFPACKAGE**形式。两种思路都是合理的，尽管每个包一个文件的方法也对组织并根据包之间的依赖关系以正确的顺序加载单独的文件提出了要求。

不论用哪种方式，一旦所有的**DEFPACKAGE**形式都从那些用到它们的代码中分离出来了，你就可以调整加载文件的顺序，让那些含有**DEFPACKAGE**的文件在编译或加载任何其他文件之前进

行加载。对于简单的程序，可以手工完成这件事：简单地加载那些含有DEFPACKAGE形式的文件，其中有可能需要先用COMPILE-FILE编译。然后加载使用这些包的文件，可再次预先选用COMPILE-FILE编译它们。不过，需要注意的是，直到你用LOAD加载那些以源代码的形式或是COMPILE-FILE输出的文件形式存在的包定义之前，包都是不存在的。因此，如果你正在编译所有的文件，仍然必须先加载所有的包定义，然后再编译那些需要从这些包里读取符号的文件。

完全手工来做这些事很快就会令人厌烦。对于简单的程序，你可以通过编写一个load.lisp文件自动完成这些步骤，该文件里含有以正确顺序排列的适当的LOAD和COMPILE-FILE调用，然后你只需加载那个文件就好了。对于更复杂的程序，你会希望使用一种系统定义功能（system definition facility）以正确的顺序来加载和编译文件。[①]

另一个关键的概括性规则是每个文件里应该只含有一个IN-PACKAGE形式，并且它应当是该文件中除注释以外的第一个形式。含有DEFPACKAGE形式的文件应当以(in-package "COMMON-LISP-USER")开始，而所有其他的文件都应当含有一个属于某个包的IN-PACKAGE形式。

如果你违反了这个规则，在文件中间切换当前包，那么就会迷惑那些没有注意到第二个IN-PACKAGE的读者。另外，许多Lisp开发环境，尤其是诸如SLIME这种基于Emacs的环境，是通过查看IN-PACKAGE来决定与Common Lisp通信时所使用的包的。如果每个文件里有多个IN-PACKAGE，也可能会干扰这些工具。

另一方面，多个文件以相同的包来读取，每个文件都使用相同的IN-PACKAGE形式，这是没有问题的。问题只是你想要怎样组织代码。

关于打包技巧的最后一点是如何给包命名。所有包名都存在于扁平的名字空间里——包名只是字符串，而不同的包必须带有文本上可区分的名字。这样，你就得考虑包名字冲突的可能性。如果你只使用自己开发的包，那么可以随意地使用短名称。但如果你正在计划使用第三方库或是发布你的代码给其他程序员使用，那么需要遵守一个可以令不同包之间名字冲突最小的命名约定。最近，许多程序员都采用了Java风格的命名方式，如同本章里使用的那些包名一样，它由一个逆向的Internet域名后跟一个点和一个描述性的字符串组成。

21.8 包的各种疑难杂症

一旦你熟悉了包，就不会再花许多时间来思考它们了。其实本来也没什么可思考的。不过，一些困扰初级Lisp程序员的疑难杂症，使得包系统看起来比它实际的情况更加复杂和不友好了。

排名第一的疑难杂症通常出现在使用REPL的时候。当你正在寻找一些定义了特别感兴趣的函数的库时，你试图像下面这样调用它们中的一个：

```
CL-USER> (foo)
```

然后，伴随以下错误信息，程序会进入调试器：

① 所有可通过本书的Web站点获得的那些来自“实践”章节的代码，都使用了ASDF系统定义库。我将在第32章里讨论ASDF。

```
attempt to call 'FOO' which is an undefined function.
   [Condition of type UNDEFINED-FUNCTION]
Restarts:
  0: [TRY-AGAIN] Try calling FOO again.
  1: [RETURN-VALUE] Return a value instead of calling FOO.
  2: [USE-VALUE] Try calling a function other than FOO.
  3: [STORE-VALUE] Setf the symbol-function of FOO and call it again.
  4: [ABORT] Abort handling SLIME request.
  5: [ABORT] Abort entirely from this (lisp) process.
```

啊！原来你忘了使用那个库的包。于是你退出调试器并试图使用该库的包来得到对名字FOO的访问，然后再调用该函数。

```
CL-USER> (use-package :foolib)
```

但这次出现了以下错误信息，并再次进入了调试器：

```
Using package 'FOOLIB' results in name conflicts for these symbols: FOO
   [Condition of type PACKAGE-ERROR]
Restarts:
  0: [CONTINUE] Unintern the conflicting symbols from the 6➥
'COMMON-LISP-USER' package.
  1: [ABORT] Abort handling SLIME request.
  2: [ABORT] Abort entirely from this (lisp) process.
```

啊？问题在于第一次调用foo的时候，读取器读取名字foo，在求值器接手并发现这个新创建的符号不是一个函数的名字之前就使其进入了CL-USER包。这个新的符号随后又和FOOLIB包中导出的同名符号相冲突了。如果你在试图调用foo之前记得**USE-PACKAGE** FOOLIB，那么读取器就可以将foo读取成一个继承而来的符号，从而就不会在CL-USER中创建一个新符号了。

不过，现在还为时不晚，因为调试器给出的第一个再启动将会以正确的方式来修复：它将使foo符号从COMMON-LISP-USER中退出，把CL-USER包恢复到调用foo之前的状态，从而使**USE-PACKAGE**得以进行并使继承来的foo在CL-USER中可用。

这类问题也会发生在加载和编译文件时。举个例子，如果你定义了一个包MY-APP，其中的代码打算使用来自包FOOLIB的函数名字，但是当你在(in-package :my-app)下编译文件时却忘记了:use FOOLIB，这样，读取器会将这些原本打算从FOOLIB中读取的符号改为在MY-APP中创建新符号。当你试图运行编译后的代码时，就会得到函数未定义的错误。如果你随后试图重定义MY-APP包，加上:use FOOLIB，那么就会得到符号冲突的错误。解决方案是一样的：选择再启动来从MY-APP中消除有冲突的符号。然后，你需要重新编译MY-APP包中的代码，这样它们就可以指向那些继承的名字了。

下一个疑难杂症本质上是第一个的相反形式。在这种情况下，你已经定义了一个用到了诸如FOOLIB的包，再次假设它是MY-APP。现在你开始编写MY-APP中的代码。尽管你使用FOOLIB是为了引用foo函数，但FOOLIB同时还导出了其他一些符号。如果你把其中一个导出的诸如bar的符号用作了自己代码里的某个函数的名字，那么Lisp就不会报错。相反，你的函数的名字将会是FOOLIB导出的那个符号，这会破坏FOOLIB中bar的定义。

这个问题的危害更大，因为它不会明显地报错。从求值器的观点来看，这只是要求将一个新

函数关联到一个旧的名字上，有时这是完全合法的。唯一的可疑之处只在于，做这个重定义的代码是在一个与函数名符号所在的包名不同的***PACKAGE***下读取的，但求值器不必关心这点。不过在多数Lisp环境下，你会得到一个关于“redefining BAR, originally defined in ...”的警告。应当留心那些警告。如果你破坏了一个库中的定义，那么可以通过使用**LOAD**重新加载库的代码来恢复它。[1]

最后一个包相关的疑难杂症相对来说比较简单，但它给多数Lisp程序员至少带来了几次麻烦：你定义了一个使用COMMON-LISP同时还可能有其他一些库的包。然后，在REPL中你切换到那个包里去做一些事。紧接着你决定完全退出Lisp环境并试图调用(quit)。不过，quit并不是来自COMMON-LISP包的名字，它被具体实现定义在了某个实现相关的包里了，后者往往是COMMON- LISP-USER包所使用的。解决方案很简单：切换回CL-USER包然后再退出，或者使用SLIME的REPL快捷键quit，它可以使你免去记忆特定Common Lisp实现的退出函数究竟是exit还是quit的烦恼。

至此，你基本完成了对Common Lisp的了解。在下一章里，我将讨论扩展形式的**LOOP**宏的细节。在那之后，本书的其余部分都将面向“实践”：一个垃圾过滤器、一个用来解析二进制文件的库以及一个带有Web接口的流式MP3服务器的各部分。

① 某些Common Lisp实现，包括Allegro和SBCL，提供了一种“锁定”特定包中符号的机制，这一机制可以确保只有当诸如**DEFUN**、**DEFVAR**和**DEFCLASS**这类定义形式所在的主包是当前包时才能顺利通过。