台 Linux 机器上编译和链接下面的源文件：

```
1  void foo(void);
2
3  int main() {
4      foo();
5      return 0;
6  }
```

那么编译器会没有障碍地运行，但是当链接器无法解析对 foo 的引用时，就会终止：

```
linux> gcc -Wall -Og -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function 'main':
/tmp/ccSz5uti.o(.text+0x7): undefined reference to 'foo'
```

对全局符号的符号解析很棘手，还因为多个目标文件可能会定义相同名字的全局符号。在这种情况中，链接器必须要么标志一个错误，要么以某种方法选出一个定义并抛弃其他定义。Linux 系统采纳的方法涉及编译器、汇编器和链接器之间的协作，这样也可能给不警觉的程序员带来一些麻烦。

**旁注 对 C++ 和 Java 中链接器符号的重整**

C++ 和 Java 都允许重载方法，这些方法在源代码中有相同的名字，却有不同的参数列表。那么链接器是如何区别这些不同的重载函数之间的差异呢？C++ 和 Java 中能使用重载函数，是因为编译器将每个唯一的方法和参数列表组合编码成一个对链接器来说唯一的名字。这种编码过程叫做重整(mangling)，而相反的过程叫做恢复(demangling)。

幸运的是，C++ 和 Java 使用兼容的重整策略。一个被重整的类名字是由名字中字符的整数数量，后面跟原始名字组成的。比如，类 Foo 被编码成 3Foo。方法被编码为原始方法名，后面加上__，加上被重整的类名，再加上每个参数的单字母编码。比如，Foo::bar(int, long)被编码为 bar__3Fooil。重整全局变量和模板名字的策略是相似的。

### 7.6.1 链接器如何解析多重定义的全局符号

链接器的输入是一组可重定位目标模块。每个模块定义一组符号，有些是局部的(只对定义该符号的模块可见)，有些是全局的(对其他模块也可见)。如果多个模块定义同名的全局符号，会发生什么呢？下面是 Linux 编译系统采用的方法。

在编译时，编译器向汇编器输出每个全局符号，或者是强(strong)或者是弱(weak)，而汇编器把这个信息隐含地编码在可重定位目标文件的符号表里。函数和已初始化的全局变量是强符号，未初始化的全局变量是弱符号。

根据强弱符号的定义，Linux 链接器使用下面的规则来处理多重定义的符号名：

- 规则 1：不允许有多个同名的强符号。
- 规则 2：如果有一个强符号和多个弱符号同名，那么选择强符号。
- 规则 3：如果有多个弱符号同名，那么从这些弱符号中任意选择一个。

比如，假设我们试图编译和链接下面两个 C 模块：

```
1  /* foo1.c */
2  int main()
3  {
4      return 0;
5  }
```