Unix I/O 模型是在操作系统内核中实现的。应用程序可以通过诸如 open、close、lseek、read、write 和 stat 这样的函数来访问 Unix I/O。较高级别的 RIO 和标准I/O 函数都是基于(使用)Unix I/O 函数来实现的。RIO 函数是专为本书开发的 read 和 write 的健壮的包装函数。它们自动处理不足值，并且为读文本行提供一种高效的带缓冲的方法。标准 I/O 函数提供了 Unix I/O 函数的一个更加完整的带缓冲的替代品，包括格式化的 I/O 例程，如 printf 和 scanf。

那么，在你的程序中该使用这些函数中的哪一个呢？下面是一些基本的指导原则：

- G1：只要有可能就使用标准 I/O。对磁盘和终端设备 I/O 来说，标准 I/O 函数是首选方法。大多数 C 程序员在其整个职业生涯中只使用标准 I/O，从不受较低级的 Unix I/O 函数的困扰(可能 stat 除外，因为在标准 I/O 库中没有与它对应的函数)。只要可能，我们建议你也这样做。
- G2：不要使用 scanf 或 rio_readlineb 来读二进制文件。像 scanf 或 rio_readlineb 这样的函数是专门设计来读取文本文件的。学生通常会犯的一个错误就是用这些函数来读取二进制文件，这就使得他们的程序出现了诡异莫测的失败。比如，二进制文件可能散布着很多 0xa 字节，而这些字节又与终止文本行无关。
- G3：对网络套接字的 I/O 使用 RIO 函数。不幸的是，当我们试着将标准I/O 用于网络的输入输出时，出现了一些令人讨厌的问题。如同我们将在 11.4 节所见，Linux 对网络的抽象是一种称为*套接字*的文件类型。就像所有的 Linux 文件一样，套接字由文件描述符来引用，在这种情况下称为套接字描述符。应用程序进程通过读写套接字描述符来与运行在其他计算机的进程实现通信。

标准 I/O 流，从某种意义上而言是*全双工*的，因为程序能够在同一个流上执行输入和输出。然而，对流的限制和对套接字的限制，有时候会互相冲突，而又极少有文档描述这些现象：

- *限制一：跟在输出函数之后的输入函数。*如果中间没有插入对 fflush、fseek、fsetpos 或者 rewind 的调用，一个输入函数不能跟随在一个输出函数之后。fflush 函数清空与流相关的缓冲区。后三个函数使用 Unix I/O lseek 函数来重置当前的文件位置。
- *限制二：跟在输入函数之后的输出函数。*如果中间没有插入对 fseek、fsetpos 或者 rewind 的调用，一个输出函数不能跟随在一个输入函数之后，除非该输入函数遇到了一个文件结束。

这些限制给网络应用带来了一个问题，因为对套接字使用 lseek 函数是非法的。对流 I/O 的第一个限制能够通过采用在每个输入操作前刷新缓冲区这样的规则来满足。然而，要满足第二个限制的唯一办法是，对同一个打开的套接字描述符打开两个流，一个用来读，一个用来写：

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

但是这种方法也有问题，因为它要求应用程序在两个流上都要调用 fclose，这样才能释放与每个流相关联的内存资源，避免内存泄漏：

```
fclose(fpin);
fclose(fpout);
```