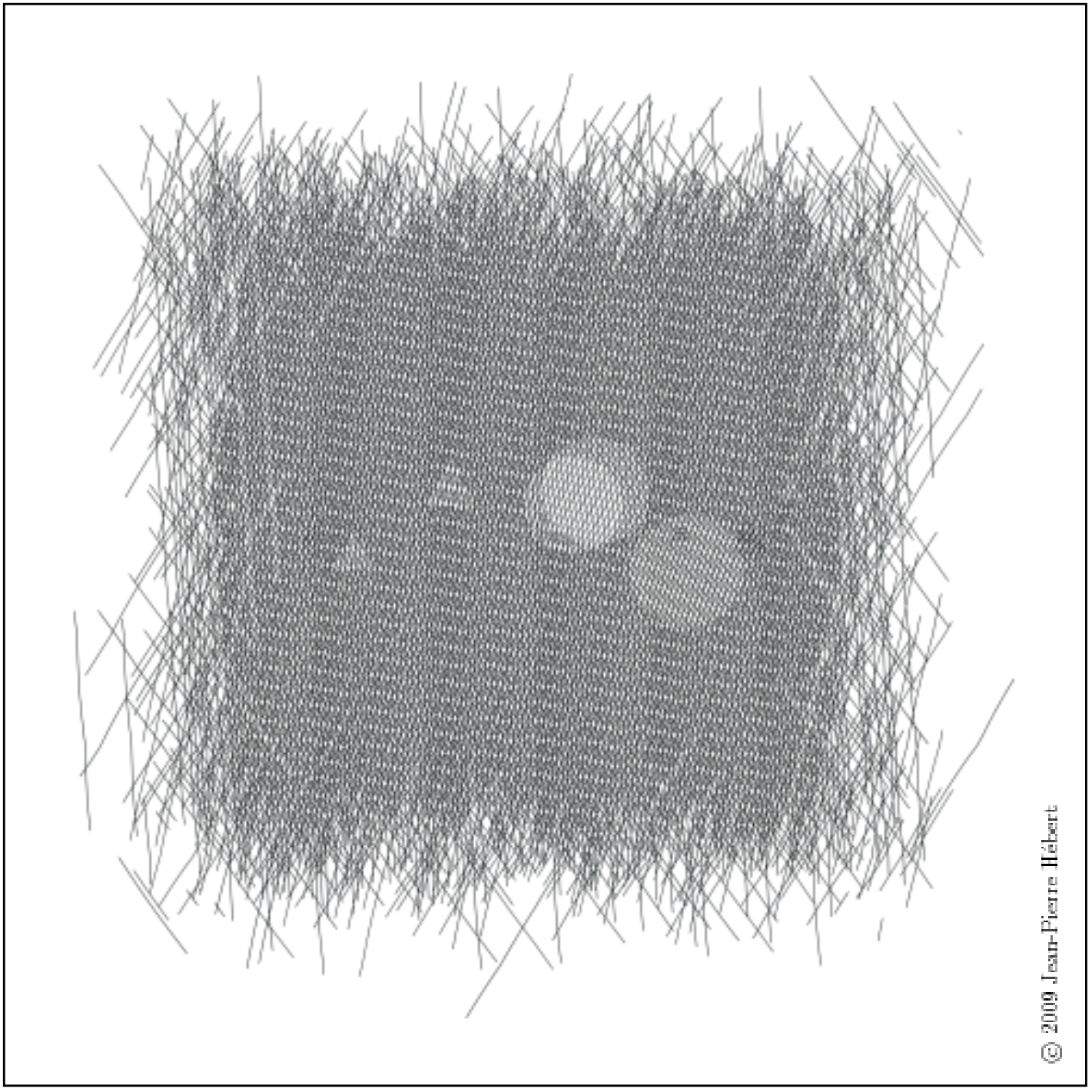

第 7 章。输入和输出

所有输入和输出操作都通过端口执行。端口是进入（可能是无限的）数据流（通常是文件）的指针，程序可以通过该开口从流中绘制字节或字符，或者将字节或字符放入流中。端口可以是输入端口和/或输出端口。

端口是一等对象，就像 Scheme 中的任何其他对象一样。与过程一样，端口不像字符串和数字那样具有打印表示形式。最初有三个端口：当前输入端口、当前输出端口和当前错误端口，它们是连接到进程的标准输入、标准输出和标准错误流的文本端口。提供了几种打开新端口的方法。

输入端口通常指向有限的流，例如，存储在磁盘上的输入文件。如果要求其中一个输入操作（例如 get-u8、get-char 或 get-datum）从已到达有限流末尾的端口读取数据，则它将返回一个特殊的 eof（文件末尾）对象。谓词 eof-object? 可用于确定从输入操作返回的值是否为 eof 对象。

端口可以是二进制的，也可以是文本的。二进制端口允许程序从基础流读取或写入 8 位无符号字节或“八位字节”。文本端口允许程序读取或写入字符。

在许多情况下，基础流被组织为一个字节序列，但这些字节应被视为字符的编码。在这种情况下，可以使用转码器创建文本端口，以将字节解码为字符（用于输入）或将字符编码为字节（用于输出）。转码器封装了一个编解码器，该编解码器确定字符如何表示为字节。提供了三种标准编解码器：拉丁语-1 编解码器、Unicode utf-8 编解码器和 Unicode utf-16 编解码器。对于 latin-1 编码，每个字符正好由一个字节表示。对于 utf-8，每个字符由 1 到 4 个字节表示，对于 utf-16，每个字符由 2 个或 4 个字节表示。

转码器还封装了一种 eol 样式，该样式确定是否以及如何识别行尾。如果 eol 样式为 none，则无法识别行尾。其他六种标准 eol 样式如下：

lf: 换行符
cr: 回车符
nel: Unicode 下一行字符
ls: Unicode 行分隔符
crlf: 回车符后跟换行符，以及
crnel: 回车符后跟下一行

eol 样式对输入和输出操作的影响不同。对于输入，除无 eol 样式外的任何 eol 样式都会导致每个行结束字符或双字符序列转换为单个换行符。对于输出，除无 eol 样式外的任何 eol 样式都会导致换行符转换为与 eol 样式关联的特定单字符或双字符序列。在输入方向上，除 none 之外的所有 eol 样式都是等效的，而在输出方向上，eol 样式 none 和 lf 是等效的。

除了编解码器和 eol 样式之外，转码器还封装了另一条信息：一种错误处理模式，该模式确定在发生解码或编码错误时会发生什么情况，即，如果一个字节序列无法转换为在输入方向上封装了编解码器的字符，或者一个字符不能转换为在输出方向上封装编解码器的字节序列。错误处理模式为忽略、引发或替换。如果忽略错误处理模式，则会忽略有问题的字节序列或字符。如果引发错误处理模式，则会引发条件类型为 i/o 解码或 i/o 编码的异常;在输入方向上，端口位于字节序列之外。如果错误处理模式被替换，则会产生替换字符或字符编码：在输入方向上，替换字符为U + FFFD，而在输出方向上，替换是utf-8和utf-16编解码器的U + FFFD编码或拉丁-1编解码器的问号字符（？）的编码。

可以对端口进行缓冲以提高效率，以消除每个字节或字符对操作系统的调用开销。支持三种标准缓冲区模式：

块、行和无。使用块缓冲时，从流中提取输入，并以某种与实现相关的大小的块的形式将输出发送到流。使用行缓冲时，缓冲是逐行执行的，或者是在其他依赖于实现的基础上执行的。行缓冲通常与仅针对文本输出端口的块缓冲区分开来。二进制端口中没有线路划分，输入很可能是从可用的流中提取的。使用无缓冲模式时，不执行缓冲，因此输出会立即发送到流，并且仅在需要时绘制输入。

本章的其余部分介绍对转码器、文件端口、标准端口、字符串和字节传量端口、自定义端口、常规端口操作、输入操作、输出操作、便利 I/O、文件系统操作以及字节向量和字符串之间的转换的操作。

第 7.1 节. 转码器

如上所述，转码器封装了三个值：编解码器、eol 样式和错误处理模式。本节介绍创建或操作转码器的过程以及转码器封装的值。

过程：（生成转码器编解码器）
过程：（生成转码器编解码器 eol 样式）
过程：（生成转码器编解码器 eol 样式错误处理模式）
返回：封装编解码器、eol 样式和error-handling-mode
libraries: （rnrs io ports）, （rnrs）

eol 样式必须是有效的 eol 样式符号（lf、cr、nel、ls、crlf、crnel 或 none）;它默认为平台的本机 eol 样式。错误处理模式必须是有效的错误处理模式符号（忽略、引发或替换），并且默认要替换。

过程：（转码器-编解码器转码器）
返回：封装在转码器
过程中的编解码器：（转码器-eol-style 转码器）返回：封装在转码器
过程中的 eol 样式符号：（转码器-错误-处理-模式转码器）
）
返回：封装在转码器
库中的错误处理模式符号：（rnrs io ports）、（rnrs）

过程：（本机转码器）
返回：本机转码器
库：（rnrs io ports），（rnrs）

本机转码器依赖于实现，可能因平台或区域设置而异。

过程：（拉丁语-1-编解码器）
返回：ISO 8859-1（拉丁语 1）字符编码
过程的编解码器：（utf-8-编解码器）
返回：Unicode UTF-8 字符编码
过程的编解码器：（utf-16-编解码器）
返回：Unicode UTF-16 字符编码库的编解码器：（rnrs io ports），（rnrs）

语法：（eol 样式的符号）
返回：符号
库：（rnrs io ports），（rnrs）

符号必须是符号 lf、cr、nel、ls、crlf、crnel 或 none 之一。表达式（eol 样式符号）等效于表达式（引号符号），但前者在展开时检查符号是 eol 样式符号之一。eol 样式的语法也提供了有用的文档。

(eol-style crlf) ⇒ crlf
(eol-style lfcr) ⇒ 语法冲突

过程： (native-eol-style)
返回： the native eol style
libraries: (rnrs io ports), (rnrs)

本机 eol 样式依赖于实现，并且可能因平台或区域设置而异。

语法： (错误处理模式符号)
返回： 符号
库： (rnrs io ports), (rnrs)

符号必须是忽略、升高或替换的符号之一。表达式(错误处理模式符号)等效于表达式(引号符号)，只是前者在展开时检查该符号是错误处理模式符号之一。错误处理模式语法也提供了有用的文档。

(错误处理模式替换)⇒替换
(错误处理模式 relpace) ⇒ 语法冲突

第 7.2 节. 打开文件

本节中的过程用于打开文件端口。打开其他类型的端口（例如，字符串端口或自定义端口）的过程将在后续部分中介绍。

每个文件打开操作都接受一个路径参数，该参数命名要打开的文件。它必须是命名文件的字符串或某个其他与实现相关的值。

某些文件打开过程接受可选选项、b 模式和 ？转码器参数。options 必须是在构成以下文件选项条目中描述的有效文件选项的符号上设置的枚举，并且它默认为 （file-options） 的值。b 模式必须是下面缓冲区模式条目中描述的有效缓冲区模式，并且默认为阻止。？转码器必须是转码器或#f;如果是转码器，则打开操作返回基础二进制文件的转码端口，而如果#f（默认值），则打开操作返回二进制端口。

本节中的过程创建的二进制端口支持端口位置和设置端口位置！ 操作。本节中的过程创建的文本端口是否支持这些操作取决于实现。

语法： （文件选项符号 ...）
返回： 文件选项枚举集
库： （rnrs io ports）， （rnrs）

文件选项枚举集可以传递给文件打开操作，以控制打开操作的各个方面。有三个标准文件选项：不创建、不失败和无截断，它们仅影响创建输出（包括输入/输出）端口的文件打开操作。

使用默认文件选项（即 （file-options） 的值），当程序尝试打开文件进行输出时，如果文件已存在，则会引发条件类型 i/o-file-already-exists 的异常;如果文件尚不存在，则会创建该文件。如果包含 no-fail 选项，则在文件已存在的情况下不会引发异常。相反，该文件被打开并截断为零长度。如果包含 no-create 选项，则如果文件不存在，则不会创建该文件。相反，会引发条件类型为 i/o-file-not-not-exist 的异常。“不创建”选项表示“不失败”选项。仅当包含或隐含了无故障选项时，不截断选项才相关，在这种情况下，如果打开现有文

件，则不会截断该文件，但端口的位置仍设置为文件的开头。

也许更容易想象默认的文件选项是虚构的选项符号创建，如果存在失败和截断；“不创建”将删除“创建”，“无失败”将删除“如果存在”，“无截断”将删除截断。

实现可能支持其他文件选项符号。例如，Chez Scheme支持控制文件是否被压缩或应该被压缩，是否被锁定以进行独占访问，以及在创建文件时给予文件什么权限的选项[9]。

语法：（缓冲区模式符号）
返回：符号
库：（rnrs io ports），（rnrs）

符号必须是块状符号、行符号或无符号之一。表达式（缓冲区模式符号）等效于表达式（引号符号），只是前者在扩展时检查该符号是缓冲区模式符号之一。缓冲区模式语法还提供了有用的文档。

```
（缓冲模式块）⇒块
（缓冲区模式缓冲）⇒ 语法冲突
```

语法：（缓冲区模式？obj）
返回：如果 obj 是有效的缓冲区模式，#t，#f否则
库：（rnrs io ports）、（rnrs）

```
（缓冲模式？’块）⇒ #t
（缓冲模式？’line）⇒ #t
（buffer-mode？’none）⇒ #t
（buffer-mode？“别的东西）⇒ #f
```

过程： （打开文件输入端口路径）
过程： （打开文件输入端口路径选项）
过程： （打开文件输入端口路径选项 b 模式）
过程： （打开文件输入端口路径选项 b 模式 ？ 转码器）
返回：命名文件
库的新输入端口：（rnrs io ports）、 （rnrs）

如果 存在 ？转码器并且未#f，则它必须是转码器，并且此过程返回一个文本输入端口，其转码器为 ？转码器。否则，此过程将返回二进制输入端口。有关其他参数的约束和影响的说明，请参阅本节的引出部分。

过程： （打开文件输出端口路径）
过程： （打开文件输出端口路径选项）
过程： （打开文件输出端口路径选项 b 模式）
过程： （打开文件输出端口路径选项 b 模式 ？ 转码器）
返回：命名文件
库的新输出端口：（rnrs io ports）、 （rnrs）

如果 存在 ？转码器并且未#f，则它必须是转码器，并且此过程返回一个文本输出端口，其转码器为 ？转码器。否则，此过程将返回二进制输出端口。有关其他参数的约束和影响的说明，请参阅本节的引出部分。

过程： （打开文件输入/输出端口路径）
过程： （打开文件输入/输出端口路径选项）
过程： （打开文件输入/输出端口路径选项 b 模式）过程：（打开文件输入/输出端口路径选项 b 模式）
？ transcoder）
返回：命名文件
库的新输入/输出端口：（rnrs io ports）、 （rnrs）

如果 存在 ？转码器并且未#f，则它必须是转码器，并且此过程返回一个文本输入/输出端口，其转码器为 ？转码器。否则，此过程将返回二进制输入/输出端口。有关其他参数的约束和影响的说明，请参阅本节的引出部分。

第 7.3 节. 标准端口

本节中描述的过程返回附加到进程的标准输入、标准输出和标准错误流的端口。第一组返回“现成的”文本端口，其中包含依赖于实现的转码器（如果有）和缓冲区模式。第二组创建新的二进制端口，可用于二进制输入/输出，或者在转码端口的帮助下，使用程序提供的转码器和缓冲区模式进行文本输入/输出。

过程：（当前输入端口）
返回：当前输入端口
过程：（当前输出端口）
返回：当前输出端口
过程：（当前错误端口）
返回：当前错误端口
库：（rnrs io ports），（rnrs io simple）、（rnrs）

电流输入、电流输出和电流误差端口返回最初与进程的标准输入、标准输出和标准误差流关联的预构建文本端口。

电流输入端口和电流输出端口返回的值可以通过方便的 I/O 过程（包括从文件输入和输出到文件）临时更改（第 7.9 节）。

过程：（标准输入端口）
返回：连接到标准输入流

过程的全新二进制输入端口：（标准输出端口）
返回：连接到标准输出流
过程的全新二进制输出端口：（标准错误端口）
返回：连接到标准错误流
库的新二进制输出端口：（rnrs io ports）、（rnrs）

由于端口可能被缓冲，因此，如果附加到进程标准流之一的多个端口上的操作是交错的，则会导致混淆。因此，仅当程序不再需要使用附加到标准流的任何现有端口时，这些过程通常才适用。

第 7.4 节. 字符串和字节向量端口

本节中的过程允许将字节向量和字符串用作输入或输出流。

本节中的过程创建的二进制端口支持端口位置和设置端口位置！操作。本节中的过程创建的文本端口是否支持这些操作取决于实现。

过程：（open-bytevector-input-port bytevector）
过程：（open-bytevector-input-port bytevector ? transcoder）
返回：一个新的输入端口，它从字节向量
库中提取输入：（rnrs io ports），（rnrs）

如果 存在 ？转码器并且未#f，则它必须是转码器，并且此过程返回一个文本输入端口，其转码器为 ？转码器。否则，此过程将返回二进制输入端口。

调用此过程后修改字节向量的效果未指定。

```
(let ([ip (open-bytevector-input-port #vu8(1 2))])
  (let* ([x1 (get-u8 ip)] [x2 (get-u8 ip)] [x3 (get-u8 ip)])
    (list x1 x2 (eof-object? x3))))  ⇒  (1 2 #t)
```

无需关闭字节向量端口;与任何其他对象一样，当不再需要它时，它的存储将自动回收，并且打开的字节向量端口不会占用任何操作系统资源。

过程：(open-string-input-port string)
返回：一个新的文本输入端口，它从字符串
库中提取输入：(rnrs io ports)， (rnrs)

调用此过程后修改字符串的效果未指定。新端口可能有也可能没有转码器，如果有，转码器取决于实现。虽然不是必需的，但鼓励实现支持字符串端口的端口位置和设置端口位置!

```
(get-line (open-string-input-port "hi.\n what's up?\n"))  ⇒  "嗨。"
```

无需关闭字符串端口;当不再需要它时，它的存储将被自动回收，就像任何其他对象一样，并且打开的字符串端口不会占用任何操作系统资源。

过程：(打开字节向量输出端口)
过程：(打开字节向量输出端口？转码器)
返回：两个值，一个新的输出端口和一个提取过程
库：(rnrs io ports)， (rnrs)

如果 存在 ？转码器并且未#f，则它必须是转码器，并且端口值是其转码器为 ？转码器的文本输出端口。否则，端口值为二进制输出端口。

提取过程是一个过程，当在没有参数的情况下调用时，将创建一个字节，其中包含端口中累积的字节，清除端口的累积字节，将其位置重置为零，并返回字节向量。累积的字节包括写入当前位置末尾以外的任何字节（如果该位置已从其最大范围设置后退）。

```
(let-values ([(op g) (open-bytevector-output-port)])
(put-u8 op 15)
(put-u8 op 73)
(put-u8 op 115)
(set-port-position! op 2)
(let ([bv1 (g)])
(put-u8 op 27)
(list bv1 (g))))⇒ (#vu8(15 73 115) #vu8(27))
```

无需关闭字节向量端口;与任何其他对象一样，当不再需要它时，它的存储将自动回收，并且打开的字节向量端口不会占用任何操作系统资源。

过程：（打开字符串输出端口）
返回：两个值，一个新的文本输出端口和一个提取过程
库：（rnrs io ports），（rnrs）

提取过程是一个过程，当在没有参数的情况下调用时，将创建一个包含端口中累积字符的字符串，清除端口中累积的字符，将其位置重置为零，然后返回该字符串。累积的字符包括写在当前位置末尾之外的任何字符（如果该位置已从其最大范围后退）。虽然不是必需的，但鼓励实现支持字符串端口的端口位置和设置端口位置！

```
(let-values ([(op g) (open-string-output-port)])
(put-string op “some data”)
(let ([str1 (g)])
```

```
(put-string op "new stuff")
(list str1 (g))))⇒ ("一些数据" "新东西")
```

无需关闭字符串端口;当不再需要它时，它的存储将被自动回收，就像任何其他对象一样，并且打开的字符串端口不会占用任何操作系统资源。

过程：（使用字节向量输出端口调用过程）
过程：（使用字节向量输出端口过程？转码器）
返回：包含累积字节
库的字节向量：（rnrs io ports），（rnrs）

如果 存在 ？转码器且未#f，则它必须是转码器，并且使用转码器为 转码器的文本字节向量输出端口调用过程。否则，使用二进制字节向量输出端口调用过程。如果过程返回，则创建一个字节，其中包含端口中累积的字节，从端口中清除累积的字节，将端口的位置重置为零，并从调用字节输出端口返回字节。如果过程由于调用在过程处于活动状态时创建的延续而多次返回，则每次过程返回时都会发生这些操作。

```
(let ([tx (make-transcoder (latin-1-codec) (eol
style lf)
(error-handling-mode replace)))
(call-with-bytevector-output-port
(lambda (p) (put-string p "abc"))
tx))⇒ #vu8(97 98 99)
```

过程：（使用字符串输出端口调用过程）
返回：包含累积字符
库的字符串：（rnrs io ports），（rnrs）

过程是用一个参数（一个字符串输出端口）调用的。如果过程返回，则创建一个包含端口中累积的字符的字符串，

从端口中清除累积的字符，将端口的位置重置为零，并从“使用字符串调用输出端口”返回该字符串。如果过程由于调用在过程处于活动状态时创建的延续而多次返回，则每次过程返回时都会发生这些操作。

可与字符串输出端口调用与放置基准一起使用，以定义一个过程，即对象>字符串，该过程返回包含对象的打印表示的字符串。

```
(define (object->string x)
(call-with-string-output-port
(lambda (p) (put-datum p x))))

(object->string (cons 'a '(b c))) ⇒ "(a b c)"
```

第 7.5 节.打开自定义端口

过程： (make-custom-binary-input-port id r! gp sp! close)
返回： 一个新的自定义二进制输入端口
过程： (make-custom-binary-output-port id w! gp sp! close)
返回： 一个新的自定义二进制输出端口
过程： (make-custom-binary-input/output-port id r! 哇! gp sp! close)
返回： 一个新的自定义二进制输入/输出端口
库： (rnrs io ports)， (rnrs)

这些过程允许程序从任意字节流创建端口。id 必须是命名新端口的字符串;该名称仅用于提供信息，并且实现可以选择将其包含在自定义端口的打印语法（如果有）中。r! 和 w! 必须是过程，而 gp、sp! 和 close 必须分别是过程或#f。下面将介绍这些参数。

r!

调用以从自定义端口获取输入，例如，支持 get-u8 或 get-bytevector-n。它使用三个参数调用：bytevector、start 和 n。start 将是一个非负的精确整数，n 将是一个正的精确整数，start 和 n 的总和不会超过字节向量的长度。如果字节流位于文件末尾，则 r! 应返回确切的 0。否则，它应该从流中读取至少一个字节，最多n个字节，将这些字节存储在从开始开始的字节向量的连续位置，并以确切的正整数返回实际读取的字节数。

w!

调用以将输出发送到端口，例如，支持 put-u8 或 put-bytevector。它使用三个参数调用：bytevector、start 和 n。start 和 n 将是非负精确整数，start 和 n 的总和不会超过字节向量的长度。哇！应从字节向量写入最多 n 个连续字节，从开始并返回，作为精确的非负整数，实际写入的字节数。

全科医生

调用以查询端口的位置。如果#f，则端口将不支持端口位置。如果未#f，则将传递零个参数，并应将当前位置作为字节流开头的位移返回为精确的非负整数，以字节为单位。

啪！

调用以设置端口的位置。如果#f，则端口将不支持设置端口位置！如果未#f，则将传递一个参数，一个精确的非负整数，将新位置表示为字节流开头的以字节为单位的位移，并且它应该将位置设置为此值。

关闭

调用以关闭字节流。如果#f，则在关闭新端口时，不会执行任何操作来关闭字节流。如果未#f，则将传递零参数，并应采取任何必要的操作来关闭字节流。

如果新端口是输入/输出端口，并且不提供 gp 或 sp! 过程，则在输入操作之后发生输出操作时，实现可能无法正确定位端口，因为必须执行输入缓冲以支持 lookahead-u8，并且通常为了提高效率而执行。出于同样的原因，如果未提供 sp! 过程，则在输入操作之后对端口位置的调用可能不会返回准确的位置。因此，创建自定义二进制输入/输出端口的程序通常应同时提供 gp 和 sp! 过程。

过程：（make-custom-textual-input-port id r! gp sp! close）
返回：一个新的自定义文本输入端口
过程：（make-custom-textual-output-port id w! gp sp! close）
返回：一个新的自定义文本输出端口
过程：（make-custom-textual-input/output-port id r! 哇! gp sp! close）
返回：一个新的自定义文本输入/输出端口
库：（rnrs io ports），（rnrs）

这些过程允许程序从任意字符流创建端口。id 必须是命名新端口的字符串;该名称仅用于提供信息，并且实现可以选择将其包含在自定义端口的打印语法（如果有）中。r! 和 w! 必须是过程，而 gp、sp! 和 close 必须分别是过程或#f。下面将介绍这些参数。

r!

被调用以从端口获取输入，例如，支持 get-char 或 get-string-n。它使用三个参数调用：字符串、开始和 n。start 将是一个非负的精确整数，n 将是一个正的精确整数，start 和 n 的总和不会超过字符串的长度。如果字符流位于文件末尾，则 r! 应返回正好 0。否则，它应该从流中读取至少一个字符，最多 n 个字符，将这些字符存储在从开头开始的字符串的连续位置，并以确切的正整数返回实际读取的字符数。

w!

调用是为了将输出发送到端口，例如，支持 put-char 或 put-string。它使用三个参数调用：字符串、开始和 n。start 和 n 将是非负精确整数，start 和 n 的总和不会超过字符串的长度。哇！应从字符串开始写入最多 n 个连续字符，并返回为精确的非负整数，即实际写入的字符数。

全科医生

调用以查询端口的位置。如果#f，则端口将不支持端口位置。如果未#f，则将传递零个参数，并应返回当前位置，这可能是任意值。

啪!

调用以设置端口的位置。如果#f，则端口将不支持设置端口位置！如果未#f，则将传递一个参数 pos，即表示新位置的值。如果 pos 是之前调用 gp 的结果，则 sp! 应将位置设置为 pos。

关闭

调用以关闭字符流。如果#f，则在关闭新端口时不会执行任何操作来关闭字符流。如果未#f，则将传

递零个参数，并应采取任何必要的操作来关闭字符流。

如果新端口是输入/输出端口，则在输入操作之后发生输出操作时，即使提供了 gp 和 sp！ 过程，实现也可能无法正确定位端口，因为必须执行输入缓冲以支持前瞻字符，并且通常无论如何都要这样做以提高效率。由于未指定端口位置的表示形式，因此实现无法调整 gp 返回值以考虑缓冲字符的数量。出于同样的原因，在输入操作之后调用端口位置可能不会返回准确的位置，即使提供了 sp！ 过程也是如此。

但是，如果位置重置为起始位置，则在读取后应该能够可靠地执行输出。因此，创建自定义文本输入/输出端口的程序通常应同时提供 gp 和 sp！ 过程，并且这些端口的使用者应在执行任何输入操作之前通过 port-position 获取起始位置，并在执行任何输出操作之前将位置重置回起始位置。

第 7.6 节. 港口运营

本节介绍对端口的各种操作，这些操作不直接涉及读取或写入端口。输入和输出操作将在后续部分中介绍。

过程：（端口？obj）
返回：如果obj是端口，#t，#f否则
库：（rnrs io ports），（rnrs）

过程：（输入端口？obj）
返回：#t如果obj是输入端口或输入/输出端口，#f否则
过程：（输出端口？obj）

返回：#t obj是输出端口还是输入/输出端口，#f否则
库：（rnrs io ports），（rnrs io simple），（rnrs）

过程：（二进制端口？obj）
返回：#t如果obj是二进制端口，#f否则
过程：（textual-port？obj）
返回：如果obj是文本端口，则#t，否则#f
库：（rnrs io ports），（rnrs）

过程：（关闭端口端口）
返回：未指定的
库：（rnrs io ports）、（rnrs）

如果 port 尚未关闭，则 close-port 将关闭它，如果端口是输出端口，请先将所有缓冲的字节或字符刷新到基础流中。一旦端口关闭，就不能再在该端口上执行输入或输出操作。由于操作系统可能会限制一次打开的文件端口数或限制对打开的文件的访问，因此最好关闭任何将不再用于输入或输出的文件端口。如果端口是输出端口，则显式关闭端口还可以确保将缓冲数据写入基础流。某些 Scheme 实现会在程序无法访问文件端口或 Scheme 程序退出后自动关闭文件端口，但最好尽可能显式关闭文件端口。关闭已关闭的端口不起作用。

过程：（转码端口二进制端口转码器）
返回：一个新的文本端口，其字节流与二进制端口
库相同：（rnrs io ports），（rnrs）

此过程返回一个新的文本端口，该端口具有转码器转码器和与二进制端口相同的基础字节流，位于二进制端口的当前位置。

作为创建文本端口的副作用，将关闭二进制端口以防止二进制端口上的读取或写入操作干扰新文本端口上的读取和写入操作。但是，在关闭文本端口之前，基础字节流将保持打开状态。

过程：（端口转码器端口）
返回：与端口关联的转码器（如果有），#f否则
库：（rnrs io ports），（rnrs）

此过程始终返回二进制端口的#f，并可能返回某些文本端口的#f。

过程：（端口-位置端口）
返回：端口的当前位置
过程：（端口具有端口位置？端口）
返回：#t端口是否支持端口位置，#f否则
库：（rnrs io ports），（rnrs）

端口可能允许查询确定其在基础字节或字符流中的当前位置。如果是这样，则过程“端口具有端口位置？”返回#t，端口位置返回当前位置。对于二进制端口，该位置始终是从字节流开始的精确非负整数字节位移。对于文本端口，位置的表示形式未指定;它可能不是一个精确的非负整数，即使它是，它也可能不表示基础流中的字节或字符位移。如果端口支持设置端口位置，则可以在以后的某个时间使用该位置来重置位置！如果在不支持端口的端口上调用了 port-position，则会引发条件类型 &断言的异常。

过程：（设置端口位置！port pos）
返回： 未指定的
过程： （port-has-set-port-position！？端口）

返回：#t端口是否支持 set-port-position!，#f否则
库：（rnrs io ports）、（rnrs）

端口可能允许将其当前位置直接移动到基础字节或字符流中的其他位置。如果是这样，则过程 port-has-set-port-position!? 返回#t，而 set-port-position! 将更改当前位置。对于二进制端口，位置 pos 必须是字节流开头的精确非负整数字节位移。对于文本端口，位置的表示形式未指定，如上面的端口位置条目中所述，但 pos 必须是文本端口的适当位置，通常只有当它是从同一端口上的停靠点获得的时，情况才会如此。如果在不支持 set-port-position! 的端口上调用，则会引发条件类型为&断言的异常。

如果 port 是二进制输出端口，并且该位置设置在基础流中数据的当前端之外，则在该位置写入新数据之前，不会扩展流。如果在该位置写入新数据，则每个干预位置的内容未指定。使用打开文件输出端口和打开文件输入/输出端口创建的二进制端口始终可以在基础操作系统的限制内以这种方式进行扩展。在其他情况下，尝试将端口设置到基础对象中当前数据末尾之外可能会导致条件类型&i/o 无效位置的异常。

过程：（使用端口调用端口过程）
返回：过程
库返回的值：（rnrs io ports），（rnrs）

以 port 为唯一参数的 port 调用过程。如果过程返回，则 port 调用将关闭端口并返回过程返回的值。

如果调用了在过程之外创建的延续，则“使用 port 调用”不会自动关闭端口，因为在过程内部创建的另一个延续可能会在以后被调用，从而将控制权返回给过程。如果

procedure 未返回，则仅当实现能够证明输出端口不再可访问时，它才可以自由地关闭该端口。

下面的示例将 infile 的内容复制到 outfile，如果存在，则覆盖 outfile。除非发生错误，否则在复制完成后将关闭端口。

```
(call-with-port (open-file-input-port “infile”
(file-options)
(buffer-mode block) (native-transcoder) )
(lambda (ip)
(call-with-port (open-file-output-port “outfile”
(file-options no-fail)
(buffer-mode block)
(native-transcoder) )
(lambda (op)
(do ([c (get-char ip) (get-char ip) ])
( (eof-object? c) )
(put-char op c) ) ) ) ) )
```

第 135 页给出了端口呼叫的定义。

过程：（输出端口缓冲区模式端口）
返回：表示端口
库的缓冲区模式的符号：（rnrs io ports），（rnrs）

第 7.7 节. 输入操作

本节介绍了主要目的是从输入端口读取数据的过程，以及用于识别或创建文件结尾（eof）对象的相关过程。

过程：（eof-object? obj）
返回：如果obj是eof对象，#t，#f否则
库：（rnrs io ports），（rnrs io simple），（rnrs）

当输入端口到达输入端时，文件末尾对象由输入操作返回，例如，get-datum。

过程： （eof-object）
返回： eof 对象
库： （rnrs io ports）， （rnrs io simple）， （rnrs）

```
（eof-object? （eof-object））⇒ #t
```

过程： （get-u8 二进制输入端口）
返回：二进制输入端口的下一个字节，或 eof 对象
库：（rnrs io ports）、 （rnrs）

如果二进制输入端口位于文件末尾，则返回 eof 对象。否则，下一个可用字节将作为无符号 8 位量返回，即小于或等于 255 的精确无符号整数，并且端口的位置提前一个字节。

过程： （lookahead-u8 二进制输入端口）
返回：来自二进制输入端口的下一个字节，或 eof 对象
库：（rnrs io ports）、（rnrs）

如果二进制输入端口位于文件末尾，则返回 eof 对象。否则，下一个可用字节将作为无符号 8 位量返回，即小于或等于 255 的精确无符号整数。与 get-u8 相反，lookahead-u8 不占用它从端口读取的字节，因此，如果端口上的下一个操作是调用 lookahead-u8 或 get-u8，则返回相同的字节。

过程： （get-bytevector-n binary-input-port n）
返回：包含最多n个字节的非空字节向量，或eof对象
库：（rnrs io ports），（rnrs）

n 必须是精确的非负整数。如果二进制输入端口位于文件末尾，则返回 eof 对象。否则，get-bytevector-n 将读取（就像使用 get-u8 一样）在端口位于文件末尾之前可用的字节数（最多 n 个），并返回包含这些字节的新（非空）字节。端口的位置将超过读取的字节数。

过程：（get-bytevector-n! 二进制输入端口字节向量开始 n）
返回：读取的字节数或对象
库的 eof：（rnrs io ports）、（rnrs）

start 和 n 必须是精确的非负整数，并且 start 和 n 的总和不得超过字节向量的长度。

如果二进制输入端口位于文件末尾，则返回 eof 对象。否则，get-bytevector-n! 读取（就像使用 get-u8 一样）在端口位于文件末尾之前可用的字节数（最多 n 个），将字节存储在从开始开始的字节向量的连续位置，并返回读取的字节数作为精确的正整数。端口的位置将超过读取的字节数。

过程：（get-bytevector-some binary-input-port）
返回：一个非空的字节向量或eof对象
库：（rnrs io ports），（rnrs）

如果二进制输入端口位于文件末尾，则返回 eof 对象。否则，get-bytevector-some 读取（就像使用 get-u8 一样）至少一个字节，甚至可能更多，并返回包含这些字节的字节。端口的位置将超过读取的字节数。此操作读取的最大字节数取决于实现。

过程：（get-bytevector-all binary-input-port）
返回：一个非空的字节向量或eof对象

库：（rnrs io ports），（rnrs）

如果二进制输入端口位于文件末尾，则返回 eof 对象。否则，get-bytevector-all 将读取（就像使用 get-u8 一样）端口位于文件末尾之前的所有可用字节，并返回包含这些字节的字节。端口的位置将超过读取的字节数。

过程：（get-char textual-input-port）
返回：来自textual-input-port的下一个字符，或eof对象
库：（rnrs io ports），（rnrs）

如果文本输入端口位于文件末尾，则返回 eof 对象。否则，将返回下一个可用字符，并且端口的位置是高级一个字符。如果文本输入端口是转码端口，则基础字节流中的位置可能会前进一个以上的字节。

过程：（lookahead-char textual-input-port）
返回：textual-input-port的下一个字符，或eof对象
库：（rnrs io ports），（rnrs）

如果文本输入端口位于文件末尾，则返回 eof 对象。否则，将返回下一个可用字符。与 get-char 相反，lookahead-char 不占用它从端口读取的字符，因此，如果端口上的下一个操作是对 lookahead-char 或 get-char 的调用，则会返回相同的字符。

为需要一个字符的“前瞻”的应用程序提供了“前瞻字符”。下面定义的过程 get-word 以字符串形式从文本输入端口返回下一个单词，其中单词定义为字母字符序列。由于 get-word 在看到单词以外的一个字符之前不知道它已经读取了整个单词，因此它使用 lookahead-char 来确定下一个字符，并使用 get-char 来消耗该字符。

```
（定义 get-word
（lambda （p）
（list->string
（let f （）
（let （[c （lookahead-char p）]）
（cond
[（eof-object? c）’（）]
[（char-alphabetic? c） （get-char p） （cons c
（f））]
[else ’（）]））））
```

过程：（get-string-n textual-input-port n）
返回：包含最多 n 个字符的非空字符串，或 eof 对象
库：（rnrs io ports）、（rnrs）

n 必须是精确的非负整数。如果文本输入端口位于文件末尾，则返回 eof 对象。否则，get-string-n 将读取（就像使用 get-char 一样）在端口位于文件末尾之前可用的字符数（最多 n 个），并返回包含这些字符的新（非空）字符串。端口的位置将超过读取的字符。

过程：（get-string-n! textual-input-port 字符串 start n）
返回：读取的字符数或对象
库的 eof：（rnrs io ports）、（rnrs）

start 和 n 必须是精确的非负整数，并且 start 和 n 的总和不得超过字符串的长度。

如果文本输入端口位于文件末尾，则返回 eof 对象。否则，get-string-n! 读取（就像使用 get-char 一样）在端口位于文件末尾之前可用的字符数（最多 n 个），将字符存储在从开头开始的字符串的连续位置，并返回作为

精确正整数读取的字符计数。端口的位置将超过读取的字符。

get-string-n! 可用于实现 string-set! 和 string-fill!，如下图所示，尽管这不是其主要目的。

```
(定义字符串集!
(lambda (s i c)
(let ([sip (open-string-input-port (string c))])
(get-string-n! sip s i 1)
; 返回未指定的值:
(if #f #f))))

(定义 string-fill!
(lambda (s c)
(let ([n (string-length s)])
(let ([sip (open-string-input-port (make-string n c))])
(get-string-n! sip s 0 n)
; 返回未指定的值:
(if #f #f))))

(let ([x (make-string 3)])
(string-fill! x #\-)
(string-set! x 2 #\))
(string-set! x 0 #\;)
x) ⇒ ";-)"
```

过程：(get-string-all textual-input-port)
返回：一个非空字符串或 eof 对象
库：(rnrs io ports)、(rnrs)

如果文本输入端口位于文件末尾，则返回 eof 对象。否则，get-string-all 将读取（就像使用 get-char 一样）在文件末尾的端口之前的所有可用字符，并返回包含这些字符的字符串。端口的位置将超过读取的字符。

过程：（get-line textual-input-port）
返回：一个字符串或 eof 对象
库：（rnrs io ports）、（rnrs）

如果文本输入端口位于文件末尾，则返回 eof 对象。否则，get-line 将读取（就像使用 get-char 一样）在端口位于文件末尾或已读取换行符之前的所有可用字符，并返回一个字符串，其中包含除读取的字符的换行符之外的所有字符。端口的位置将超过读取的字符。

```
(let ([sip (open-string-input-port
"one\ntwo\n")])
(let* ([s1 (get-line sip)] [s2 (get-line sip)])
(list s1 s2 (port-eof? sip))))⇒ ("one"
"two" #t)

(let ([sip (open-string-input-port "one\ntwo")])
(let* ([s1 (get-line sip)] [s2 (get-line sip)])
(list s1 s2 (port-eof? sip))))⇒ ("one"
"two" #t)
```

过程：（获取基准文本输入端口）
返回：方案基准对象或 eof 对象
库：（rnrs io ports）、（rnrs）

此过程扫描过去的空白和注释，以查找基准的外部表示的起点。如果在找到基准的外部表示的开始之前，文本输入端口到达文件的末尾，则返回 eof 对象。

否则，get-datum 将根据需要读取尽可能多的字符，而不是读取更多字符来解析单个基准，并返回一个新分配的对象，其结构由外部表示确定。端口的位置将超过读取的字符。如果在基准的外部表示完成之前到达文件末

尾，或者读取了意外字符，则会引发异常，条件类型 & 词法和 i/o 读取。

```
(let ([sip (open-string-input-port ";a\n\n one
(two) \n") ])
(let* ([x1 (get-datum sip) ]
[c1 (lookahead-char sip) ]
[x2 (get-datum sip) ])
(list x1 c1 x2 (port-eof? sip) ) ) ) ⇒ (一个
#\space (two) #f)
```

过程：（端口 eof? 输入端口）
返回：如果输入端口位于文件末尾，则#t，否则#f
库：（rnrs io ports），（rnrs）

此过程类似于二进制输入端口上的 lookahead-u8 或文本输入端口上的 lookahead-char，不同之处在于它返回一个布尔值以指示该值是否为 eof 对象，而不是返回下一个字节/字符或 eof 对象。

第 7.8 节.输出操作

本节介绍其主要目的是将数据发送到输出端口的过程。

过程：（put-u8 二进制输出端口八位字节）
返回：未指定的
库：（rnrs io ports），（rnrs）

八位字节必须是小于或等于 255 的精确非负整数。此过程将八位字节写入二进制输出端口，将端口的位置提前一个字节。

过程： (put-bytevector binary-output-port bytevector)
procedure: (put-bytevector binary-output-port bytevector start)
procedure: (put-bytevector binary-output-port bytevector start n)
返回： unspecated
libraries: (rnrs io ports), (rnrs)

start 和 n 必须是非负精确整数，并且 start 和 n 的总和不得超过字节向量的长度。如果未提供，则 start 默认为零，n 默认为字节向量和 start 的长度之差。

此过程将从开始到端口写入字节的 n 个字节，并将其位置提前到写入的字节末尾之后。

过程： (put-char textual-output-port char)
返回： 未指定的
库： (rnrs io ports), (rnrs)

此过程将 char 写入文本输出端口，将端口的位置提前一个字符。如果文本输出端口是转码端口，则基础字节流中的位置可能会前进一个以上的字节。

过程： (put-string textual-output-port string)
过程： (put-string textual-output-port string start)
过程： (put-string textual-output-port string start n)
返回： 未指定的
库： (rnrs io ports), (rnrs)

start 和 n 必须是非负整数，start 和 n 的总和不得超过字符串的长度。如果未提供，则 start 默认为零，n 默

认为字符串长度和 start 之间的差值。

此过程将从开头开始的字符串的 n 个字符写入端口，并将其位置移到所写字符的末尾之后。

过程： (put-datum textual-output-port obj)
返回： 未指定的
库： (rnrs io ports)， (rnrs)

此过程将 obj 的外部表示形式写入文本输出端口。如果 obj 没有作为基准的外部表示，则行为未指定。精确的外部表示依赖于实现，但是当obj确实有一个外部表示作为基准时，put-datum应该产生一系列字符，这些字符以后可以通过get-datum作为对象读取（在等于的意义上？）与obj等效。请参见第 12.5 节，了解放置基准、写入和显示的实现。

过程： (刷新输出端口输出端口)
返回：未指定的
库： (rnrs io ports)， (rnrs)

此过程强制将与输出端口关联的缓冲区中的任何字节或字符立即发送到基础流。

第 7.9 节.方便的 I/O

本节中的过程称为“方便”I/O 运算符，因为它们提供了用于创建文本端口并与之交互的简化界面。它们还提供了与修订版[5] 报告的向后兼容性，后者不支持单独的二进制和文本I/O。

可以使用或不使用显式端口参数来调用方便的输入/输出过程。如果在没有显式端口参数的情况下调用，则根据需要使用当前输入或输出端口。例如，（read-char）和（read-char（current-input-port））都从当前输入端口返回下一个字符。

过程：（打开输入文件路径）
返回：一个新的输入端口
库：（rnrs io simple），（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。open-input-file 为按路径命名的文件创建新的文本输入端口，就像通过具有默认选项、依赖于实现的缓冲区模式和依赖于实现的转码器打开文件输入端口一样。

下面演示了在表达式中使用打开输入文件、读取和关闭端口，该表达式从名为“myfile.ss”的文件中收集对象列表。

```
(let ([p (open-input-file “myfile.ss”)])
(let f ([x (read p)])
(if (eof-object? x)
(begin
(close-port p)
'())
(cons x (f (read p))))))
```

过程：（打开输出文件路径）
返回：新的输出端口
库：（rnrs io simple），（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。open-output-file 为按 path 命名的文件创建新的输出端口，就像通过 open-file-output-port（具有默认选

项、依赖于实现的缓冲区模式和依赖于实现的转码器）一样。

下面演示如何使用 open-output-file 将对象列表（待打印列表的值）（以换行符分隔）写入以“myfile.ss”命名的文件。

```
(let ([p (open-output-file "myfile.ss")])
(let f ([ls list-to-be-printed])
(if (not (null? ls))
(begin
(write (car ls) p)
(newline p)
(f (cdr ls)))))
(close-port p))
```

过程：（与输入文件路径的调用过程）
返回：过程
库返回的值：（rnrs io simple），（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。过程应接受一个参数。

与 input-file 的调用为按 path 命名的文件创建一个新的输入端口，就像使用 open-input-file 一样，并将此端口传递给过程。如果过程返回，则与输入文件的调用将关闭输入端口并返回过程返回的值。

如果调用了在过程之外创建的延续，则与 input-file 的调用不会自动关闭输入端口，因为在过程内部创建的另一个延续可能会在以后被调用，从而将控制权返回给过程。如果 procedure 未返回，则仅当实现能够证明输入端口不再可访问时，它才可以自由地关闭输入端口。如第 5.6

节所示，如果调用了在过程之外创建的延续，则可以使用动态风来确保关闭端口。

下面的示例演示如何在表达式中使用与 input 文件的调用，该表达式从由“myfile.ss”命名的文件中收集对象列表。它在功能上等效于上面为打开输入文件给出的示例。

```
(call-with-input-file “myfile.ss”
(lambda (p)
(let f ([x (read p)])
(if (eof-object? x)
’()
(cons x (f (read p)))))))
```

可以使用输入文件调用，但可以按如下方式定义而不进行错误检查。

```
(define call-with-input-file
(lambda (filename proc)
(let ([p (open-input-file filename)])
(let-values ([v* (proc p)])
(close-port p)
(apply values v*)))))
```

过程：（带输出文件路径的调用过程）
返回：过程
库返回的值：（rnrs io simple）、（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。过程应接受一个参数。

与输出文件的调用为按 path 命名的文件创建一个新的输出端口，就像使用 open-output-file 一样，并将此端口传

递给过程。如果过程返回，则与输出文件的调用将关闭输出端口并返回过程返回的值。

如果调用了在过程之外创建的延续，则与输出文件的调用不会自动关闭输出端口，因为在过程内部创建的另一个延续可能会在以后被调用，从而将控制权返回给过程。如果 procedure 未返回，则仅当实现能够证明输出端口不再可访问时，它才可以自由地关闭输出端口。如第 5.6 节所示，如果调用了在过程之外创建的延续，则可以使用动态风来确保关闭端口。

下面演示如何使用带输出的调用文件将对象列表（要打印的列表的值）写入以换行符分隔的文件，以“myfile.ss”命名。它在功能上等效于上面为打开输出文件给出的示例。

```
（调用与输出文件“myfile.ss”
（lambda （p）
（let f （[ls list-to-be-printed]）
（除非 （null? ls）
（write （car ls） p）
（newline p）
（f （cdr ls））））
```

可以定义带有输出文件的调用，而无需进行错误检查，如下所示。

```
（定义带输出文件
的调用 （lambda （文件名 proc）
（let （[p （open-output-file 文件名）]）
（let-values （[v* （proc p）]）
（close-port p）
（apply values v*））））））
```

过程：（with-input-from-file path thunk）
返回：thunk
库返回的值：（rnrs io simple）、（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。thunk 必须是一个过程，并且应该接受零参数。

with-input-from-file 在应用 thunk 期间，将当前输入端口临时更改为打开按路径命名的文件的结果，就像打开输入文件一样。如果 thunk 返回，则该端口将关闭，当前输入端口将恢复为其旧值。

如果在 thunk 返回之前调用了在 thunk 之外创建的延续，则 with-input-from-file 的行为是未指定的。实现可能会关闭端口并将当前输入端口还原到其旧值---但可能不会。

过程：（带有输出到文件路径 thunk）
返回：thunk
库返回的值：（rnrs io simple）、（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。thunk 必须是一个过程，并且应该接受零参数。

with-output-to-file 临时重新绑定当前输出端口，使其成为在应用 thunk 期间打开由 path 命名的文件的结果，就像打开输出文件一样。如果 thunk 返回，则关闭端口，并将当前输出端口恢复为其旧值。

如果在 thunk 返回之前调用了在 thunk 之外创建的延续，则无法指定 with-output-to-file 的行为。实现可能会关闭端口并将当前输出端口还原到其旧值---但可能不会。

过程： （读取）
过程： （读取文本输入端口）
返回： 方案基准对象或 eof 对象
库： （rnrs io simple）， （rnrs）

如果未提供文本输入端口，则默认为当前输入端口。否则，此过程等效于获取基准。

过程： （read-char）
过程： （read-char textual-input-port）
返回：文本输入端口
库中的下一个字符：（rnrs io simple）、 （rnrs）

如果未提供文本输入端口，则默认为当前输入端口。此过程在其他方面等效于 get-char。

过程： （peek-char）
过程： （peek-char textual-input-port）
返回： 来自 textual-input-port
库的下一个字符： （rnrs io simple）， （rnrs）

如果未提供文本输入端口，则默认为当前输入端口。此过程在其他方面等效于前瞻字符。

procedure: (write obj)
procedure: (write obj textual-output-port)
返回： uns specific
libraries: (rnrs io simple), (rnrs)

如果未提供文本输出端口，则默认为当前输出端口。此过程在其他方面等效于 put-datum，参数颠倒过来。请参见第 12.5 节，了解放置基准、写入和显示的实现。

过程： （显示 obj）
过程： （显示 obj textual-output-port）
返回： 未指定的
库： （rnrs io simple）， （rnrs）

如果未提供文本输出端口，则默认为当前输出端口。

display 类似于写入或放置基准，但直接打印在 obj 中找到的字符串和字符。字符串打印时不带引号，特殊字符不带转义，就像通过 put-string 一样，字符打印时不带 #\ 表示法，就像用 put-char 一样。使用显示时，三元素列表（a b c）和两元素列表（“a b” c）都打印为（a b c）。因此，不应使用 display 来打印要与读取一起读取的对象。display 主要用于打印消息，obj 通常是字符串。请参见第 12.5 节，了解放置基准、写入和显示的实现。

过程： （写字符字符）
过程： （写字符字符文本输出端口）
返回： 未指定的
库： （rnrs io simple）， （rnrs）

如果未提供文本输出端口，则默认为当前输出端口。此过程在其他方面等效于 put-char，参数颠倒过来。

过程： （换行符）
过程： （换行符文本输出端口）
返回： 未指定的
库： （rnrs io simple）， （rnrs）

如果未提供文本输出端口，则默认为当前输出端口。换行符将换行符发送到端口。

过程： （关闭输入端口输入端口）
过程： （关闭输出端口输出端口）
返回： 未指定的
库： （rnrs io simple）， （rnrs）

关闭输入端口关闭输入端口，关闭输出端口关闭输出端口。提供这些程序是为了向后兼容经修订的[5] 报告;它们实际上并不比近距离端口更方便使用。

第 7.10 节. 文件系统操作

除了文件输入/输出之外，Scheme 还有两个标准操作，用于与文件系统交互：file-exists? 和 delete-file。大多数实现都支持其他操作。

过程： （文件存在？路径）
返回：#t路径命名的文件是否存在，#f否则
库： （rnrs 文件）、（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。文件是否存在？是否遵循符号链接未指定。

过程： （删除文件路径）
返回：未指定的
库： （rnrs 文件）、（rnrs）

path 必须是字符串或命名文件的其他一些与实现相关的值。delete-file 将删除由 path 命名的文件（如果该文件存在并且可以删除），否则会引发条件类型为 &i/o-filename 的异常。删除文件是否遵循符号链接是未指定的。

第 7.11 节. 字节/字符串转换

本节中描述的过程对字符序列进行编码或解码，从字符串转换为字节向量或从字节向量转换为字符串。它们不一定涉及输入/输出，尽管它们可能使用字节向量输入和输出端口实现。

前两个过程（字节向量>字符串和字符串>字节向量）采用显式转码器参数，用于确定字符编码、eol 样式和错误处理模式。其他的则使用隐式 eol 样式 none 和错误处理模式替换执行特定的 Unicode 转换。

过程：（字节向量>字符串字节向量转码器）
返回：一个字符串，其中包含在字节向量
库中编码的字符：（rnrs io ports），（rnrs）

此操作至少在实际上会创建一个具有指定转码器的字节向量输入端口，从中读取所有可用字符，就像通过 get-string-all 一样，并将其放入输出字符串中。

```
(let ([tx (make-transcoder (utf-8-codec) (eol
style lf)
(error-handling-mode replace))])
(bytevector->string #vu8(97 98 99) tx)) ⇒ "abc"
```

过程：（字符串>字节向量字符串转码器）
返回：一个字节向量，其中包含字符串
库中字符的编码：（rnrs io ports），（rnrs）

此操作（至少在实际上）会创建一个字节向量输出端口，该端口具有指定的转码器，字符串的所有字符都将写入该转码器，然后提取包含累积字节的字节。

```
(let ([tx (make-transcoder (utf-8-codec) (eol-style none)
(error-handling-mode raise) )])
(string->bytevector "abc" tx) ) ⇒ #vu8(97 98 99)
```

过程：(string->utf8 字符串)
返回：包含字符串
库的 UTF-8 编码的字节向量：(rnrs bytevectors)、(rnrs)

过程：(string->utf16 字符串)
过程：(string->utf16 字符串字节序)
过程：(string->utf32 字符串)
过程：(string->utf32 字符串字节序)
返回：包含指定字节序的字节量字符串
库的编码：(rnrs bytevectors)，(rnrs)

字节序必须是大或小的符号之一。如果未提供字节序或符号较大，则 string->utf16 返回字符串的 UTF-16BE 编码，字符串>utf32 返回字符串的 UTF-32BE 编码。如果字节序是符号小，则 string->utf16 返回字符串的 UTF-16LE 编码，字符串>utf32 返回字符串的 UTF-32LE 编码。编码中不包含字节顺序标记。

过程：(utf8->string bytevector)
返回：包含 UTF-8 解码字节向量
库的字符串：(rnrs bytevectors)、(rnrs)

过程：(utf16->字符串字节序)
过程：(utf16->字符串字节序字节序-强制？)
过程：(utf32->字符串字节序)
过程：(utf32->字符串字节序 字节序 字节序-强制性？
返回：包含字节向量

库的指定解码的字符串：（rnrs bytevectors），（rnrs）

字节序必须是大或小的符号之一。这些过程返回字节向量的 UTF-16 或 UTF-32 解码，表示的字节序由字节序参数或字节顺序标记 （BOM） 确定。如果未提供字节序强制？？或#f字节序，则字节序由字节向量前面的 BOM 确定，或者，如果不存在 BOM，则由字节序确定。如果字节序强制？#t，则字节序由字节序确定，并且如果 BOM 出现在字节向量的前面，则将其视为常规字符编码。

UTF-16 BOM 是#xFE的双字节序列，#xFF指定“大”或#xFF的双字节序列，#xFE指定“小”。UTF-32 BOM 是四字节序列#x00，#x00#xFE，#xFF指定“大”或四字节序列#xFF，#xFE，#x00，#x00指定“小”。

ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93
订购本书 / 关于这本书

http://www.scheme.com