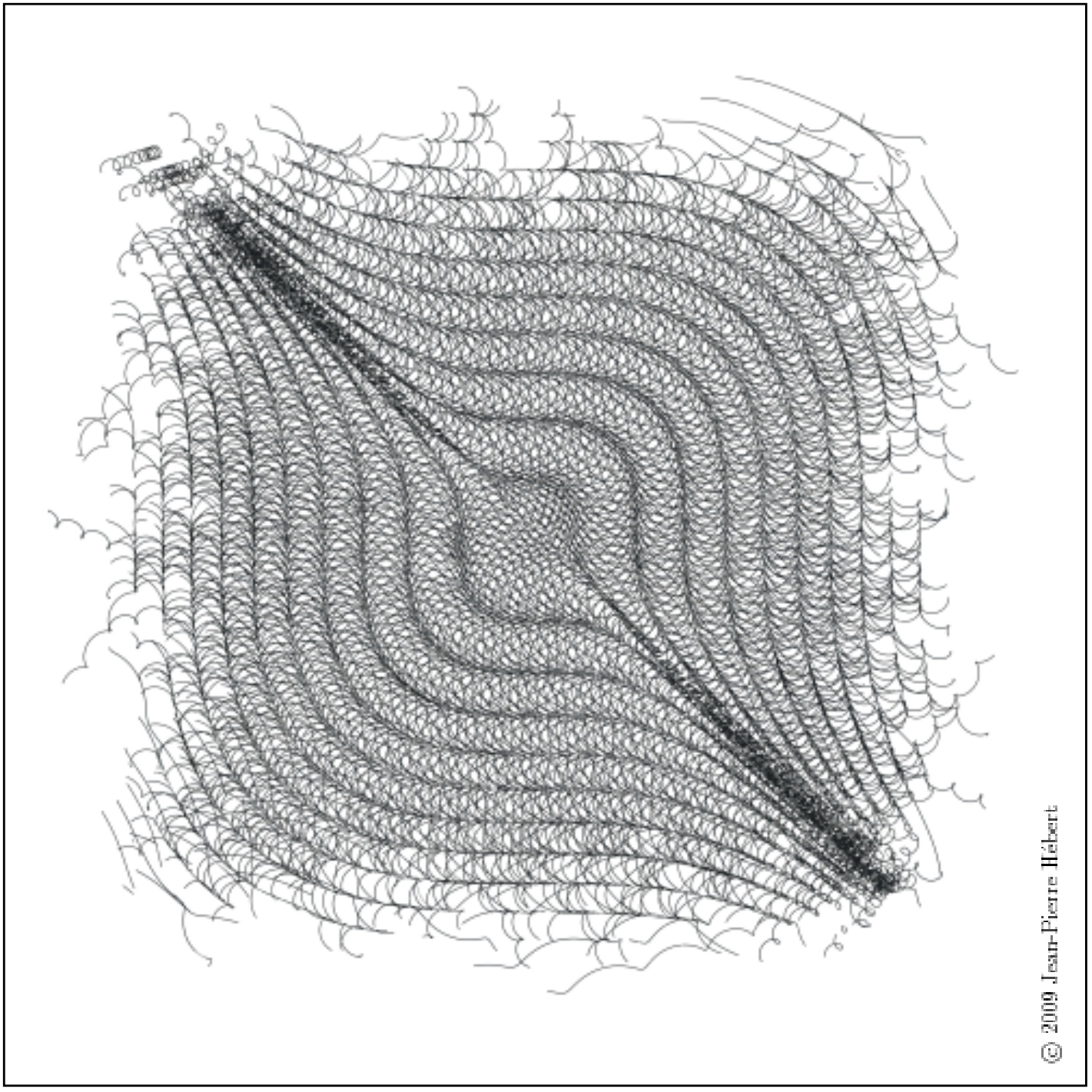

第 11 章。例外和条件

异常和条件为系统和用户代码提供了发出信号、检测程序运行时发生的错误并从中恢复的方法。

在各种情况下，标准语法形式和过程都会引发异常，例如，当传递给过程的参数数错误时，当传递给 `eval` 的

表达式的语法不正确时，或者当文件打开过程之一无法打开文件时。在这些情况下，将使用标准条件类型引发异常。

用户代码也可能通过 raise 或 raise-continuable 过程引发异常。在这种情况下，可以使用标准条件类型之一、标准条件类型之一的用户定义子类型（可能使用定义条件类型定义）或不是条件类型的任意 Scheme 值引发异常。

在程序执行期间的任何时候，单个异常处理程序（称为当前异常处理程序）负责处理引发的所有异常。默认情况下，当前异常处理程序是实现提供的处理程序。默认异常处理程序通常打印一条消息，描述引发异常的条件或其他值，对于任何严重情况，都会终止正在运行的程序。在交互式系统中，这通常意味着重置为读取-评估-打印循环。

用户代码可以通过保护语法或异常处理程序过程建立新的当前异常处理程序。在任一情况下，用户代码都可以处理所有异常，或者根据引发异常的条件或其他值，只处理部分异常，同时重新请求其他异常以供旧的当前异常处理程序处理。当防护表单和带异常处理程序调用动态嵌套时，将建立一个异常处理程序链，并且每个异常处理程序都可能遵从链中的下一个处理程序。

第 11.1 节. 引发和处理异常

过程： (raise obj)
过程： (raise-continuable obj)
返回： 参见下面的
库： (rnrs exceptions)， (rnrs)

这两个过程都会引发异常，从而有效地调用当前的异常处理程序，并将 obj 作为唯一的参数传递。对于 raise，异常是不可延续的，而对于 raise-continuable，异常是可延续的。异常处理程序可能会返回（具有零个或多个值）到可连续异常的延续。但是，如果异常处理程序尝试返回到不可连续异常的延续，则会引发条件类型为 &non-continuable 的新异常。因此，raise 永远不会返回，而 raise-continuable 可能会返回零个或多个值，具体取决于异常处理程序。

如果当前异常处理程序 p 是通过保护表单或对 with-exception-handler 的调用建立的，则当前异常处理程序将重置为在引发或提高可连续调用 p 之前建立 p 时的当前处理程序。这允许 p 仅通过重新引发异常来遵从预先存在的异常处理程序，并且当异常处理程序无意中导致引发不同的异常时，它有助于防止无限回归。如果 p 返回并且异常是可延续的，则 p 将恢复为当前异常处理程序。

```
(raise
(condition
(make-error)
(make-message-condition “no go”))) ⇒ 错误: no go
(raise-continuable
(condition
(make-violation)
(make-message-condition “oops”))) ⇒ violation: oops
(list
(call/cc
(lambda (k)
(vector
(with-exception-handler
(lambda (x) (k (+ x 5)))
(lambda () (+ (raise 17) 8)))))) ⇒ (22)
```

```
(list
(vector
(with-exception-handler
(lambda (x) (+ x 5))
(lambda () (+ (raise-continuable 17) 8)))))⇒
(#(30))
(列表
(向量
(带异常处理程序
(lambda (x) (+ x 5))
(lambda () (+ (raise 17) 8)))))⇒ 违规： 不可延续
```

程序： （错误谁味精刺激...）
过程： （断言违规谁 msg 刺激 ...）
库： （rnrs base）， （rnrs）

error 会引发条件类型 &error 的不可连续异常，应用于描述 &error 条件类型适合的情况，通常是涉及程序与程序外部某些内容交互的情况。断言冲突会引发条件类型为 &assertion 的不可连续异常，并且应用于描述 &断言条件类型适合的情况，通常是过程的无效参数或语法形式的子表达式的无效值。

引发异常的延续对象还包括 &who 条件，其 who 字段是 who（如果 who 不是#f），则为 &message 条件（其消息字段为 msg），以及其刺激性字段为（irritant ...）的 &irritants 条件。

必须是字符串、符号或#f标识报告错误的过程或语法形式，以代表谁报告错误。通常最好确定程序员调用的过程，而不是程序员可能不知道执行操作时涉及的其他一些过程。msg 必须是字符串，并且应描述异常情况。刺

激物可以是任何 Scheme 对象，并应包括可能导致或实质性参与特殊情况的值。

语法： （断言表达式）
返回： 请参阅下面的
库： （rnrs base）， （rnrs）

assert 计算表达式，如果表达式的值未#f，则返回表达式的值。如果表达式的值#f，则 assert 将引发一个不可连续的异常，其条件类型为 &assertion 和 &message，其消息字段中包含依赖于实现的值。鼓励实现尽可能提供有关条件中断言调用位置的信息。

过程： （语法冲突谁 msg 表单）
过程： （语法违规谁 msg 表单子窗体）
返回：不返回
库： （rnrs 语法大小写）， （rnrs）

此过程会引发一个不可延续的异常，条件为类型 &语法。它应用于报告语法扩展的转换器检测到的语法错误。条件的表单域的值为 form，其子窗体字段的值为子窗体，如果未提供子窗体，则#f。

引发异常的延续对象还包括一个 &who 条件，其 who 字段是 who，如果 who 不是 #f或从表单推断出来，以及消息字段为 msg 的 &message 条件。

谁必须是字符串、符号或#f。如果谁#f，则推断出表单的符号名称（如果表单是标识符）或表单的第一个子表单的符号名称（如果表单是列表结构表单，其第一个子表单是标识符）。消息必须是字符串。形式应该是发生语法冲突的句法形式的语法对象或基准面表示形式，子窗体（如果不是#f）应是更具体地参与冲突的子窗体的语法对象或

基准面表示。例如，如果在 lambda 表达式中找到重复的形式参数，则 form 可能是 lambda 表达式，子窗体可能是重复的参数。

某些实现将源信息附加到语法对象，例如，源自文件中的表单的行、字符和文件名，在这种情况下，此信息也可能作为条件对象中的某些依赖于实现的条件类型存在。

过程：（带有异常处理程序过程 thunk）
返回：请参阅下面的
库：（rnrs exceptions），（rnrs）

此过程建立应接受一个参数的过程，作为当前异常处理程序，以代替旧的当前异常处理程序 old-proc，并在没有参数的情况下调用 thunk。如果对 thunk 的调用返回，则将 old-proc 重新建立为当前异常处理程序，并返回 thunk 返回的值。如果控件离开或随后通过调用通过 call/cc 获取的延续来重新输入对 thunk 的调用，则在捕获延续时作为当前异常处理程序的过程将恢复。

```
(define (try thunk)
(call/cc
(lambda (k)
(with-exception-handler
(lambda (x) (if (error? x) (k #f) (raise
x) ) )
thunk) ) ) )
(try (lambda () 17) ) ⇒ 17
(try (lambda () (raise (make-error) ) ) ) ⇒ #f
(try (lambda () (raise (make-violation) ) ) ) ⇒
violation
(with-exception-handler
(lambda (x)
(raise
```

```
(apply condition
(make-message-condition "oops")
(simple-conditions x))))
(lambda ()
(try (lambda () (raise (make-violation))))))
⇒ 违规： 哎呀
```

语法： (保护 (var 子句 $_1$ 子句 $_2$ ...) b1 b2 ...)
返回：请参阅下面的
库：(rnrs exceptions)， (rnrs)

保护表达式建立一个新的当前异常处理程序，过程（如下所述），代替旧的当前异常处理程序old-proc，并计算主体b1 b2 ...。如果主体返回，则 guard 会将 old-proc 重新建立为当前异常处理程序。如果控件通过调用通过 call/cc 获取的延续离开或随后重新进入主体，则将恢复捕获延续时作为当前异常处理程序的过程。

由 guard 建立的过程过程将 var 绑定到它接收的值，并在该绑定的范围内处理子句 clause1 clause2 ......反过来，就好像包含在隐式 cond 表达式中一样。此隐式 cond 表达式在 guard 表达式的延续中计算，old-proc 作为当前异常处理程序。

如果未提供 else 子句，则 guard 将提供一个以相同值重新引发异常的子句，就像在继续调用过程时使用 raise-continuable 一样，将 old-proc 作为当前的异常处理程序。

```
(guard (x [else x]) (raise "oops")) ⇒ "oops"
(guard (x [#f #f]) (raise (make-error))) ⇒
error
(define-syntax try
(syntax-rules ()
```

```
[ (_ e1 e2 ...)
 (警卫 (x [(错误? x) #f]) e1 e2 ...) ) )
 (定义 open-one
 (lambda fn*
 (let loop ([ls fn*])
 (if (null? ls)
 (error 'open-one "all open trys failed" fn*)
 (or (try (open-input-file (car ls)) )
 (loop (cdr ls)) ) )
;假设 bar.ss 存在但未 foo.ss:
 (开放式"foo.ss" "bar.ss") ⇒ #<输入端口 bar.ss>
```

第 11.2 节. 定义条件类型

虽然程序可以通过 raise 或 raise-concontinuable 任何 Scheme 值，但描述异常情况的最佳方法通常是创建并传递一个条件对象。如果修订 6 报告要求实现引发异常，则传递给当前异常处理程序的值始终是 Section 11.3 中描述的一个或多个标准条件类型的条件对象。用户代码可以创建一个条件对象，该条件对象是一个或多个标准条件类型的实例，也可以创建扩展条件类型并创建该类型的条件对象。

条件类型类似于记录类型，但更灵活，因为条件对象可以是两个或多个条件类型的实例，即使两者都不是另一个条件类型的子类型。当条件是多个类型的实例时，它被称为复合条件。复合条件对于将有关异常的多条信息传达给异常处理程序非常有用。不是复合条件的条件称为简单条件。在大多数情况下，两者之间的区别并不重要，一个简单的条件被视为一个复合条件，它本身就是它唯一的简单条件。

语法： &条件
库： （rnrs 条件）， （rnrs）

&condition 是记录类型名称（第 9 章）和条件类型层次结构的根。所有简单条件类型都是此类型的扩展，所有条件（无论是简单条件还是复合条件）都被视为此类型的实例。

程序：（条件? obj）
返回：如果 obj 是条件对象，则#t，否则#f
库：（rnrs 条件）、（rnrs）

条件对象是 &condition 或复合条件的子类型的实例，可能由具有条件的用户代码创建。

```
（条件? “稳定”#f ⇒
（条件? （制造错误））⇒ #t
（条件? （make-message-condition “oops”））⇒ #t
（条件?
（条件
（制造错误）
（使消息条件“没有这样的元素”））））⇒ #t
```

过程： （条件条件 ...）
返回：一个条件，可能是复合
库：（rnrs 条件）、 （rnrs）

condition 用于创建可能包含多个简单条件的条件对象。每个参数条件可以是简单或复杂的;如果简单，则将其视为复合条件，其自身是唯一的简单条件。结果条件的简单条件是条件参数的简单条件，平展成单个列表并按顺序显示，第一个条件的简单条件后跟第二个条件的简单条件，依此类推。

如果列表只有一个元素，则结果条件可以是简单或复合的;否则就是复合的。简单条件和复合条件之间的区别通常并不重要，但如果使用定义记录类型而不是定义条件类型来扩展现有条件类型，则可以通过定义记录类型而不是定义条件类型来检测。

```
(条件) ⇒ #<condition>
(condition
(make-error)
(make-message-condition "oops")) ⇒ #<condition>

(define-record-type (&xcond make-xcond xcond
xcond?) (父和条件))
(xcond? (make-xcond)) ⇒ #t
(xcond? (条件 (make-xcond)) ⇒ #t还是#f
(xcond? (条件)) ⇒ #f
(xcond? (条件 (制造错误) (使 xcond))) ⇒ #f
```

过程：(简单条件条件)
返回：条件
库的简单条件列表：(rnrs 条件)、(rnrs)

```
(简单条件(条件)) ⇒ '()
(simple-conditions (make-error)) ⇒ (#<condition
&error>)
(simple-conditions (condition (make-error))) ⇒
(#<condition &error>)
(simple-conditions

(make-error)
(make-message-condition
"oops"))) ⇒ (#<condition &error> #<condition
&message>)

(let ([c1 (make-error)]
[c2 (制造者条件"f")]
[c3 (make-message-condition "invalid argument")]
```

```
[c4 （make-message-condition
“error happening when when read from file”）]
[c5 （make-irritants-condition ’（“a.ss”））])
（equal?
（简单条件
（条件（条件
c1 c2） c3）
（条件 c4（条件 c5））））
（列表 c1 c2 c3 c4 c5））） ⇒ #t
```

语法： （定义条件类型名称父构造函数 pred 字段 ...）
库： （rnrs 条件）， （rnrs）

定义条件类型形式是一个定义，可以出现在其他定义可能出现的任何位置。它用于定义新的简单条件类型。

子窗体名称、父项、构造函数和 pred 必须是标识符。每个字段的格式必须为（字段名称访问器名称），其中字段名称和访问器名称是标识符。

define-condition-type 将 name 定义为一种新的记录类型，其父记录类型为父记录类型，其构造函数名称为构造函数，其谓词名称为 pred，其字段为字段名称 ...，其字段访问器由访问器名称命名 ...。

除谓词和字段访问器外，define-condition-type 本质上是等效于

```
（定义记录类型（名称构造函数 pred）
（父父级）
（字段（（不可变字段名称访问器名称）...））））
```

谓词与由定义记录类型表单生成的谓词的不同之处在于，它不仅返回#t新类型的实例，还返回简单条件包括新类型实例的复合条件。同样，字段访问器接受新类型的实

例以及其简单条件至少包含一个新记录类型的实例的复合条件。如果访问器收到复合条件，其简单条件列表包含新类型的一个或多个实例，则访问器将对列表中的第一个实例进行操作。

```
（定义条件类型 &错误 &条件犯错误?
（键入错误类型））

（错误?'booboo） ⇒ #f

（定义 c1 （make-error 'spelling））
（error? c1） ⇒ #t
（error-type c1） ⇒ spelling

（define c2 （condition c1 （make-irritants-condition
'（eggregius））））
（error? c2） ⇒ #t
（error-type c2） ⇒ spelling
（irritants-condition? c2） ⇒ #t
（condition-irritants c2） ⇒ （eggregius）
```

过程：（条件谓词 rtd）
返回：条件谓词
过程：（条件访问器 rtd 过程）
返回：条件访问器
库：（rnrs 条件）、（rnrs）

这些过程可用于创建由定义记录类型从简单条件类型的记录描述符 rtd 或从简单条件类型派生的其他类型的定义记录类型创建的相同类型的特殊谓词和访问器。

对于这两个过程，rtd 必须是 &condition 子类型的记录类型描述符，对于条件访问器，过程应接受一个参数。

条件谓词返回的谓词接受一个参数，该参数可以是任何 Scheme 值。如果值是 rtd 描述的类型的条件，即 rtd 描述的类型的实例（或其子类型之一）或其简单条件包括 rtd 描述的类型的实例，则谓词返回#t。否则，谓词将返回#f。

条件访问器返回的访问器接受一个参数 c，该参数必须是 rtd 所描述的类型的条件。访问器将过程应用于单个参数，即 c 的简单条件列表的第一个元素，该元素是 rtd 描述的类型的实例（如果 c 是简单条件，则为 c 本身），并返回此应用程序的结果。在大多数情况下，过程是 rtd 所描述的类型的字段的记录访问器。

```
（定义记录类型（&错误犯错误$mistake?
（父级和条件）
（字段 （不可变类型 $mistake类型））)

;定义谓词和访问器，就好像我们使用了 define-condition-type
（define rtd （record-type-descriptor &error））
（定义 error? （条件谓词 rtd））
（定义错误类型（条件访问器 rtd $mistake-type））

（定义 c1 （make-error 'spelling））
（定义 c2 （condition c1 （make-irritants-condition '（eggregius））））
（list （error? c1） （error? c2）） ⇒ （#t #t）
（list （$mistake? c1） （$mistake? c2）） ⇒ （#t #f）
（error-type c1） ⇒ 拼写
（$mistake type c1） ⇒ 拼写
（error-type c2） ⇒拼写
（$mistake 型 c2） ⇒ 冲突
```

第 11.3 节.标准条件类型

语法： &严重
过程： （制造-严重-条件）
返回： 类型和严重
过程的条件： （严重条件？obj）
返回：如果 obj 是类型 &serious 的条件，则#t，#f否则
库：（rnrs 条件）、（rnrs）

这种类型的条件表示性质严重的情况，如果不被发现，通常会导致程序执行的终止。此类型的条件通常作为更具体的子类型 &error 或 &violation 之一出现。此条件类型可以按如下方式定义。

```
（定义条件类型&严重&条件
使严重条件严重条件？
```

语法： &违规
过程： （make-violation）
返回： 类型 &violation
procedure： （violation？obj）
返回：如果 obj 是类型 &violation 的条件，则#t，#f否则
库：（rnrs 条件）、（rnrs）

此类型的条件表示程序违反了某些要求，通常是由于程序中的 bug。此条件类型可以按如下方式定义。

```
（定义条件类型&违规&严重
违规？
```

语法： &断言
过程： （生成断言 - 违反）
返回： 类型和断言

过程的条件： （断言 - 违反？obj）
返回：如果 obj 是 &assertion 类型的条件，则#t，否则#f
库：（rnrs 条件）、（rnrs）

此条件类型指示程序向过程传递了错误数量或类型的参数的特定冲突。此条件类型可以按如下方式定义。

（定义条件类型 &断言 &违规
使断言 - 违规断言 - 违规？

语法： &error
过程： （make-error）
返回： 类型 &error
过程： （error？obj）
返回：如果 obj 是 &error 类型的条件，则#t，否则#f
库：（rnrs 条件）、（rnrs）

此类型的条件表示程序与其操作环境的交互时出错，如尝试打开文件失败。它不用于描述在程序中检测到错误的情况。此条件类型可以按如下方式定义。

（定义条件类型 &错误 &严重的
制造错误？

语法： &警告
过程： （发出警告）
返回： 类型 &警告
过程： （警告？obj）
返回：如果 obj 是类型 &warning 的条件，则#t，否则#f
库：（rnrs 条件）、（rnrs）

警告条件表示不会阻止程序继续执行的情况，但在某些情况下，可能会在以后的某个时间点导致更严重的问题。例如，编译器可能使用此类型的条件来指示它已处理对具有错误参数数的标准过程的调用;这不会成为一个严重的问题，除非在以后的某个时候实际评估呼叫。此条件类型可以按如下方式定义。

（定义条件类型和警告 &条件
制定警告警告？

语法： &message
过程： （make-message-condition message）
返回： 类型 &message
过程： （message-condition? obj）
返回：如果 obj 是 &message 类型的条件，#t，#f否则
过程：（条件-消息条件）
返回：条件的消息字段
库的内容：（rnrs 条件）、（rnrs）

这种类型的条件通常包含在&警告条件或&严重条件子类型之一中，以提供对特殊情况的更具体描述。构造函数的消息参数可以是任何 Scheme 值，但通常是字符串。此条件类型可以按如下方式定义。

（定义条件类型 &消息 &条件
生成消息条件消息条件？
（消息条件消息））

语法： &刺激物
程序： （制造刺激物 - 条件刺激物）
返回： 类型和刺激物
的条件程序： （刺激物条件？obj）
返回：#t如果 obj 是类型 &irritants 的条件，#f否则

过程：（条件刺激条件）
返回：条件的刺激物字段
库的内容：（rnrs 条件）、（rnrs）

这种类型的条件通常包含在 &message 条件中，以提供有关可能导致或严重涉及特殊情况的 Scheme 值的信息。例如，如果过程接收到错误类型的参数，则它可能会引发异常，其复合条件由断言条件、命名过程的 who 条件、声明接收了错误类型的参数的消息条件以及列出该参数的刺激条件组成。构造函数的刺激性参数应该是一个列表。此条件类型可以按如下方式定义。

```
（定义条件类型&刺激物&条件
制造刺激物-条件刺激物-条件？
（刺激物条件刺激物））
```

语法： &who
过程： （make-who-condition who）
返回： 类型 &who
条件的过程： （who-condition? obj）
返回：#t如果 obj 是类型 &who 的条件，#f否则
过程：（condition-who 条件）
返回：条件的内容 who 字段
库：（rnrs 条件）、 （rnrs）

此类型的条件通常包含在 &message 条件中，以标识检测到错误的语法形式或过程。构造函数的 who 参数应为符号或字符串。此条件类型可以按如下方式定义。

```
（定义条件类型&谁&条件
使谁条件谁条件？
（谁条件-谁））
```

语法： &不可连续
过程： （使-不可连续-违反）
返回： 类型 &不可连续
过程的条件： （不可连续 - 违反？obj）
返回：如果 obj 是类型 &不可连续的条件，则#t，#f否则
库：（rnrs 条件）、（rnrs）

此类型的条件表示发生了不可延续的冲突。如果当前异常处理程序返回，则 raise 将引发此类型的异常。此条件类型可以按如下方式定义。

```
（定义条件类型 &不可延续 &违规
使不可延续-违规
不可延续-不可延续-违规？
```

语法： &实现限制
过程： （make-implement-restriction-violation）
返回： 类型 &implement-restriction
procedure： （implementation-restriction-violation? obj）
返回：#t如果obj是类型&实现限制的条件，#f否则
库：（rnrs条件），（rnrs）

实现限制条件表示程序已尝试超过实现中的某个限制，例如，当 fixnum 加法操作的值将导致数字超出实现的 fixnum 范围时。它通常不表示实现中的缺陷，而是程序尝试执行的操作与实现可以支持的内容之间的不匹配。在许多情况下，实现限制由底层硬件决定。此条件类型可以按如下方式定义。

```
（定义条件类型 &实现-限制 &违规
使-实施-限制-违规
实施-限制-违规？
```

语法： &词法
过程： （使词法 - 违反）
返回： 类型和词法
过程的条件： （词法违规？obj）
返回：如果 obj 是 &lexical 类型的条件，#t，#f否则
库：（rnrs 条件）、（rnrs）

此类型的条件表示在解析 Scheme 程序或基准时发生了词法错误，例如括号不匹配或出现在数字常量中的无效字符。此条件类型可以按如下方式定义。

```
（定义条件类型 &词法 &违规
使词法违规 词法违规？
```

语法： &语法
过程： （生成语法违规表单子窗体）
返回： 类型 &语法
过程的条件： （语法冲突？obj）
返回：#t如果obj是类型&语法的条件，#f否则
过程：（语法-违规-表单条件）
返回：条件的表单字段
过程的内容：（语法-违规-子表单条件）
返回：条件的子窗体字段
库的内容：（rnrs 条件）、（rnrs）

此类型的条件表示在解析 Scheme 程序时发生了语法错误。在大多数实现中，宏扩展器会检测到语法错误。每个表单和子窗体参数都应该是语法对象（第 8.3 节）或基准，前者表示包含形式，后者表示特定的子窗体。例如，如果在 lambda 表达式中找到重复的形式参数，则 form 可能是 lambda 表达式，子窗体可能是重复的参数。如果不需要标识子窗体，则应#f子窗体。此条件类型可以按如下方式定义。

（定义条件类型 &语法 &违规
使语法违规 语法违规?
（表单语法-冲突-表单）
（子窗体语法冲突-子窗体））

语法： &未定义
过程： （make-undefined-violation）
返回： 类型 &undefined
procedure： （undefined-violation? obj）
返回：如果 obj 是类型 &undefined 的条件，则#t，#f否则
库：（rnrs 条件）、（rnrs）

未定义的条件表示尝试引用未绑定变量。此条件类型可以按如下方式定义。

（定义条件类型 &未定义 &违规
使未定义 -违规 未定义 -违规?

接下来的几个条件类型描述当输入或输出操作以某种方式失败时发生的条件。

语法： &i/o
过程： （make-i/o-error）
返回： 类型 &i/o
过程： （i/o-error? obj）
返回：如果 obj 是 &i/o 类型的条件，则#t，否则#f
库：（rnrs io ports）、（rnrs io simple）、（rnrs files）、（rnrs）

类型 &i/o 的条件表示发生了某种类型的输入/输出错误。这种类型的条件通常作为下面描述的更具体的亚型之一出现。此条件类型可以按如下方式定义。

```
（定义条件类型 &i/o &error
make-i/o-error i/o-error? ）
```

语法： &i/o-read
过程： （make-i/o-read-error）
返回： 类型 &i/o-read
过程的条件： （i/o-read-error? obj）
返回：如果 obj 是 &i/o-read 类型的条件，#t，#f否则
库：（rnrs io ports）、（rnrs io simple）、（rnrs files）、（rnrs）

此条件类型指示从端口读取时出错。此条件类型可以按如下方式定义。

```
（define-condition-type &i/o-read &i/o
make-i/o-read-error i/o-read-error? ）
```

语法： &i/o-write
过程： （make-i/o-write-error）
返回： 类型 &i/o-write
过程的条件： （i/o-write-error? obj）
返回：如果 obj 是 &i/o-write 类型的条件，则#t，否则#f
库：（rnrs io ports）、（rnrs io simple）、（rnrs files）、（rnrs）

此条件类型指示写入端口时发生错误。此条件类型可以按如下方式定义。

```
（define-condition-type &i/o-write &i/o
make-i/o-write-error i/o-write-error? ）
```

语法： &i/o-invalid-position
过程： （make-i/o-invalid-position-error position）

返回： 类型 &i/o-invalid-position
过程的条件： (i/o-invalid-position-error? obj)
返回：#t如果obj是类型&i/o-invalid-position的条件，#f否则
过程：(i/o-error-position条件)
返回：条件的位置字段
库的内容：(rnrs io ports)，(rnrs io simple)，(rnrs files)， (rnrs)

此条件类型表示尝试将端口的位置设置为超出基础文件或其他对象范围的位置。构造函数的位置参数应为无效位置。此条件类型可以按如下方式定义。

```
(define-condition-type &i/o-invalid-position &i/o
make-i/o-invalid-position-error
i/o-invalid-position-error?
(位置 i/o-错误-位置))
```

语法： &i/o-文件名
过程： (make-i/o-filename-error filename)
返回： &i/o-filename
过程： (i/o-filename-error? obj)
返回：#t如果 obj 是 &i/o-filename 类型的条件，#f否则
过程：(i/o-error-filename 条件)
返回：条件的文件名字段
库的内容：(rnrs io ports)、(rnrs io simple)、(rnrs files)， (rnrs)

此条件类型指示对文件进行操作时发生的输入/输出错误。构造函数的文件名参数应为文件的名称。此条件类型可以按如下方式定义。

（define-condition-type &i/o-filename &i/o
make-i/o-filename-error i/o-filename-error?
（文件名 i/o 错误文件名））

语法： &i/o-文件保护
过程： （make-i/o-file-protection-error 文件名）
返回： 类型 &i/o-file-protection
procedure： （i/o-file-protection-error? obj）
返回：如果obj是&i/o-file-protection类型的条件，#t#f否则
库：（rnrs io ports），（rnrs io simple），（rnrs files），（rnrs）

此类型的条件表示已尝试对程序没有适当权限的文件执行某些输入/输出操作。此条件类型可以按如下方式定义。

（定义条件类型 &i/o-文件保护 &i/o-文件名
make-i/o-file-protection-error
i/o-file-protection-error? ）

语法： &i/o-file-is-read-only
过程： （make-i/o-file-is-read-only-error filename）
返回： 类型为 &i/o-file-is-read-only
的过程： （i/o-file-is-read-only-error? obj）
返回：#t如果obj是类型&i/o-file-is-read-only的条件，#f否则
库：（rnrs io ports），（rnrs io simple），（rnrs文件），（rnrs）

此类型的条件表示尝试将只读文件视为可写文件。此条件类型可以按如下方式定义。

```
（define-condition-type &i/o-file-is-read-only &i/o-file-protection
make-i/o-file-is-read-only-error
i/o-file-is-read-only-error? ）
```

语法： &i/o-file-already-exists
过程： （make-i/o-file-already-error filename）
返回： 类型 &i/o-file-already-exists
的过程： （i/o-file-already-exists-error? obj）
返回：#t如果obj是类型&i/o-file-already-exists的条件，#f否则
库：（rnrs io ports），（rnrs io simple），（rnrs files），（rnrs）

此类型的条件表示对文件的操作失败的情况，因为该文件已存在，例如，尝试打开现有文件以进行输出，但没有“无失败文件”选项。此条件类型可以按如下方式定义。

```
（define-condition-type &i/o-file-already-exists &i/o-filename
make-i/o-file-already-exists-error
i/o-file-already-exists-error? ）
```

语法： &i/o-file-does-not-exist
过程： （make-i/o-file-does-not-exist-error 文件名）
返回： 类型为 &i/o-file-do-not-exist
过程： （i/o-file-do-not-exist-error? obj）
返回：如果 obj 是类型 &i/o-file-does-not-exist 的条件，#t否则库#f
：（rnrs io ports）、（rnrs io simple）、（rnrs files）、（rnrs）

此类型的条件表示对文件的操作失败的情况，因为该文件不存在，例如，尝试打开不存在的文件仅用于输入。此条件类型可以按如下方式定义。

```
(define-condition-type &i/o-file-do-not-exist &i/o-filename
make-i/o-file-do-not-exist-error
i/o-file-does-not-exist-error? )
```

语法： &i/o-port
过程： (make-i/o-port-error pobj)
返回： 类型 &i/o-port
过程的条件： (i/o-port-error? obj)
返回：如果 obj 是 &i/o-port 类型的条件，#t，#f否则
过程：(i/o-error-port 条件)
返回：条件的 pobj 字段
库的内容：(rnrs io ports)、(rnrs io simple)、(rnrs 文件)、 (rnrs)

此类型的条件通常包含在其他 &i/o 子类型之一的条件中，以指示异常情况下涉及的端口（如果涉及端口）。构造函数的 pobj 参数应为端口。此条件类型可以按如下方式定义。

```
(define-condition-type &i/o-port &i/o
make-i/o-port-error i/o-port-error?
(pobj i/o-error-port) )
```

语法： &i/o-解码
过程： (make-i/o-decoding-error pobj)
返回： 类型 &i/o-decoding
过程的条件： (i/o-decoding-error? obj)
返回：如果 obj 是 &i/o 解码类型的条件，则#t，否则#f
库：(rnrs io ports)、(rnrs)

此类型的条件表示在将字节转码为字符期间发生解码错误。构造函数的 pobj 参数应该是所涉及的端口（如果有）。端口应位于无效编码的上方。此条件类型可以按如下方式定义。

```
（定义条件类型 &i/o-解码 &i/o-port
make-i/o-decoding-error i/o-decoding-error? ）
```

语法： &i/o 编码
过程： （make-i/o-encoding-error pobj cobj）
返回： 类型 &i/o 编码
过程的条件： （i/o-encoding-error? obj）
返回：#t如果obj是类型&i/o编码的条件，#f否则
过程：（i/o-encoding-error-char条件）
返回：条件的cobj字段
库的内容：（rnrs io ports），（rnrs）

此类型的条件表示在将字符转码为字节的过程中发生了编码错误。构造函数的 pobj 参数应该是所涉及的端口（如果有），而 cobj 参数应该是编码失败的字符。此条件类型可以按如下方式定义。

```
（define-condition-type &i/o-encoding &i/o-port
make-i/o-encoding-error i/o-encoding-error?
（cobj i/o-encoding-error-char））
```

最后两种条件类型描述了当实现需要生成 NaN 或无穷大但没有这些值的表示形式时发生的条件。

语法： &no-infinities
过程： （make-no-infinities-violation）
返回： 类型 &no-infinities
过程的条件： （no-infinities-violation? obj）

返回：#t如果 obj 是类型和无穷大的条件，#f否则
库：（rnrs 算术 flonums）、（rnrs）

此条件指示实现没有无穷大的表示形式。此条件类型可以按如下方式定义。

```
（定义条件类型和无穷大 &实现限制
使无穷大违反
无穷大 违反？
```

语法： &no-nans
过程： （make-no-nans-violation）
返回： 类型 &no-nans
过程的条件： （no-nans-violation？obj）
返回：#t如果obj是类型&no-nans的条件，#f否则
库：（rnrs算术flums），（rnrs）

此条件表示实现没有 NaN 的表示形式。此条件类型可以按如下方式定义。

```
（定义条件类型 &no-nans &实现限制
make-no-nans-violation no-nans-violation？）
```

R. Kent Dybvig / The Scheme Programming Language， Fourth Edition

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

http://www.scheme.com