

Perl 操作文本的强大能力部分来源于对一个名为 正则表达式 的计算概念的囊括。正则 表达式（通常简化为 *regex* 或 *regexp*）是一个 模式，它描述了某文本字符串的 特征。正则表达式引擎 解释一个模式并将其应用到文本字符串上，以识别何者匹配。

*Perl* 的核心文档丰富而详细地描述了 *Perl* 的正则表达式；请参考 `perldoc perlretut` 教程、`perldoc perlre` 完整文档和 `perldoc perlreref` 参考指南。Jeffrey Friedl 的著作 精通正则表达式 解释了正则表达式工作背后的理论和机制。即便这些参考书看上去令人望而生畏，正则表达式却很像 Perl—— 你可以在懂得很少的情况下完成很多任务。

## 字面值

最简单的正则表达式就是简单的子字符串模式：

```
my $name = 'Chatfield';
say "Found a hat!" if $name =~ B</hat/>;
```

匹配操作符（`//` 或者，更为正式的 `m//`）中包含一正则表达式——在这个例子中是 `hat`。即使读上去像一个单词，它的意思其实是“`h` 字符，后接 `a` 字符，再接 `t` 字符，出现在该字符串的任何地方”。在 `hat` 中的每一个字符都是一个正则表达式中的 原子：该模式独立的单元。正则表达式绑定操作符（`=~`）是一个中缀操作符（fixity），它将位于其右的正则表达式应用于左边由表达式产生的字符串。当在标量上下文中求值时，一个成功的匹配将得到真值。

绑定操作符的否定形式（`!~`）在匹配成功时得到一个假值。

## `qr//` 操作符和正则表达式组合

在现代化的 Perl 中，当由 `qr//` 操作符创建时，正则表达式是第一级实体：

```
my $hat = B<qr/hat/>;
say 'Found a hat!' if $name =~ /$hat/;
```

来自 `Test::More` 的 `like()` 函数的工作就像 `is()` 一样，除了它的第二个参数是一个由 `qr//` 产生的正则表达式对象。

你可以通过内插和组合将他们变为更大更复杂的模式：

```
my $hat    = qr/hat/;
my $field  = qr/field/;

say 'Found a hat in a field!' if $name =~ /B<$hat$field>;

# 或

like( $name, qr/B<$hat$field>/, 'Found a hat in a field!' );
```

## 量词

正则表达式比前例演示的更为强大；你可以使用 `index` 操作符在字符串内搜索子字符串字面值。但用正则表达式引擎达成这项任务就像驾驶自治区战斗直升机去拐角小店购买备用奶酪。

通过使用 正则表达式量词，正则表达式可以变得更为强大，使你能够指定一个正则表达式组件在匹配字符串中出现的频率。最简单的量词是 零个或一个量词，或者说 `?`：

```
my $cat_or_ct = qr/caB<?>t/;

like( 'cat', $cat_or_ct, "'cat' matches /ca?t/" );
like( 'ct',  $cat_or_ct, "'ct' matches /ca?t/" );
```

在正则表达式中，任意原子后接 `?` 字符意味着“匹配此原子零次或一次”。这个正则表达式匹配 `c` 后立即跟随零个 `a` 字符再接一个 `t` 字符。它同时也匹配在 `c` 和 `t` 字符间有一个 `a`。

一个或多个量词，或者说 `+`，在字符串中只匹配量词前原子至少出现一次的情况：

```
my $one_or_more_a = qr/caB<+>t/;

like( 'cat',    $one_or_more_a, "'cat' matches /ca+t/" );
like( 'caat',   $one_or_more_a, "'caat' matches /ca+t/" );
like( 'caaat',  $one_or_more_a, "'caaat' matches /ca+t/" );
like( 'caaaat', $one_or_more_a, "'caaaat' matches /ca+t/" );

unlikely( 'ct', $one_or_more_a, "'ct' does not match /ca+t/" );
```

能匹配多少原子没有什么理论上的限制。

零个或多个量词 是 `*`；它匹配量化原子在字符串中出现的零个或多个实例：

```
my $zero_or_more_a = qr/caB<*>t/;

like( 'cat',    $zero_or_more_a, "'cat' matches /ca*t/" );
like( 'caat',   $zero_or_more_a, "'caat' matches /ca*t/" );
like( 'caaat',  $zero_or_more_a, "'caaat' matches /ca*t/" );
like( 'caaaat', $zero_or_more_a, "'caaaat' matches /ca*t/" );
like( 'ct',     $zero_or_more_a, "'ct' matches /ca*t/" );
```

这看上去可能不很有用，但是它和其他正则表达式功能可以组合得很好，让你不必关心在特定位置是否出现某模式。即便如此，大多数 正则表达式从使用 `?` 和 `+` 量词 中获益远多于 `*` 量词，因为它们可以避免昂贵的回溯，并将你的意图表达得更为清晰。

最后，你可以通过 数值量词 指定某原子匹配的次数。`{n}` 意味着确切匹配 `n` 次。

```
# equivalent to qr/cat/;
my $only_one_a = qr/caB<{1}>t/;

like( 'cat', $only_one_a, "'cat' matches /ca{1}t/" );
```

`{n,}` 意味着匹配次数必须至少为 `n` 次，同时可以匹配更多次数：

```
# equivalent to qr/ca+t/;
my $at_least_one_a = qr/caB<{1,}>t/;

like( 'cat',    $at_least_one_a, "'cat' matches /ca{1,}t/" );
like( 'caat',   $at_least_one_a, "'caat' matches /ca{1,}t/" );
like( 'caaat',  $at_least_one_a, "'caaat' matches /ca{1,}t/" );
```

```
like( 'caaaat', $at_least_one_a, "'caaaat' matches /ca{1,}t/" );
```

{n,m} 意味着必须至少匹配 n 次且不多于 m 次:

```
my $one_to_three_a = qr/caB<{1,3}>t/;

like( 'cat', $one_to_three_a, "'cat' matches /ca{1,3}t/" );
like( 'caat', $one_to_three_a, "'caat' matches /ca{1,3}t/" );
like( 'caaat', $one_to_three_a, "'caaat' matches /ca{1,3}t/" );
unlike( 'caaaat', $one_to_three_a, "'caaaat' does not match /ca{1,3}t/" );
```

## 贪心性

就 + 和 \* 自身来说, 它们是 贪心量词; 它们尽可能多地匹配输入字符串。在利用 .\* 来匹配“任何数量的任何字符”时尤其有害:

```
# a poor regex
my $hot_meal = qr/hot.*meal/;

say 'Found a hot meal!' if 'I have a hot meal' =~ $hot_meal;
say 'Found a hot meal!'
    if 'I did some one-shot, piecemeal work!' =~ $hot_meal;
```

贪心量词总是试图 先行 匹配尽可能多的输入字符串, 仅在匹配明显不成功时回退。如果你用如下方式查找单词“loam (土壤)”, 你将无法把所有结果塞进 7 Down 的四个盒子里面:

```
my $seven_down = qr/l${letters_only}*m/;
```

作为新手, 你将得到 Alabama、Belgium 以及 Bethlehem。那里的土壤也许不错, 但是它们全都太长了——并且, 匹配是从单词的中间开始的。

让贪心量词变成非贪心量词只需在其后加上 ? 量词:

```
my $minimal_greedy_match = qr/hot.*?meal/;
```

当给予非贪心量词, 正则表达式引擎将偏向 最短的、可能的潜在匹配, 并仅在目前数目无法满足匹配要求时, 增加由 .\*? 标记组合识别的字符数量。由于 \* 匹配零或更多次, 对应这个标记组合的最小潜在匹配是零个字符:

```
say 'Found a hot meal' if 'ilikeahotmeal' =~
/$minimal_greedy_match/;
```

使用 + 量词可以匹配某项一次或多次:

```
my $minimal_greedy_at_least_one = qr/hot.+?meal/;

unlike( 'ilikeahotmeal', $minimal_greedy_at_least_one );

like( 'i like a hot meal', $minimal_greedy_at_least_one );
```

? 量词修饰符也可以应用于 ? (零或一次匹配) 和范围量词。在每种情况下, 它使

得正则表达式尽可能地少匹配。贪心修饰符 `.+` 和 `.*` 是诱人但危险的。如果你编写了贪心匹配正则表达式，请用综合自动化测试套件和代表性数据完整地测试它，以将产生令人不快结果的可能性降到最小。

## 正则表达式锚点

正则表达式锚点 强制在字符串某位置进行匹配。字符串开头锚点 (`\A`) 确保任何匹配都将从字符串开头开始：

```
# 也匹配 "lammed"、"lawmaker" 和 "layman"
my $seven_down = qr/\A${letters_only}{2}m/;
```

字符串末尾锚点 (`\Z`) 确保任何匹配都将 结束 于字符串末尾：

```
# 也匹配 "loom", 它足够接近
my $seven_down = qr/\A${letters_only}{2}m\Z/;
```

单词边界元字符 (`\b`) 仅匹配一单词字符 (`\w`) 和另一非单词字符 (`\W`) 之间的边界。因此，查找 `loam` 而非 `Belgium`，可以使用加锚点的正则表达式：

```
my $seven_down = qr/\b${letters_only}{2}m\b/;
```

与 Perl 类似，达成某个目的的正则表达式也有许多写法。请考虑从中挑出最有表达力也是最易维护的一个。

## 元字符

正则表达式随着原子的一般化而变得更为强大。举例来说，在正则表达式内，`.` 字符的意思是“匹配除换行外的任意字符”。玩填字游戏时，如果你想要在一个单词列表里查找每一个匹配的 `7 Down` (“`Rich soil`”)，你可以这样写：

```
for my $word (@words)
{
    next unless length( $word ) == 4;
    next unless $word =~ /lB<..>m/;
    say "Possibility: $word";
}
```

当然，如果你的候选匹配列表由单词外的东西构成，这个元字符可能导致假阳性，因为它同时匹配标点符号、空格、数字以及其他的非单词字符。`\w` 元字符代表所有字母数字字符（按 Unicode 处理——`unicode`）还有下划线：

```
next unless $word =~ B</lB<\w\w>m/>;
```

`\d` 元字符匹配数字——不是你预期的 `0-9`，而是 Unicode 数字：

```
# 并非一个健壮的电话号码匹配器
next unless $potential_phone_number =~
/B<\d>{3}-B<\d>{3}-B<\d>{4}/;
say "I have your number: $potential_phone_number";
```

可以使用 `\s` 元字符匹配空白，无论是字面空格、制表符、硬回车、换页符或者换行：

```
my $two_three_letter_words = qr/\w{3}B<\s>\w{3}/;
```

这些元字符也有否定形式。要匹配 除 单词外的其他字符，使用 `\W`。要匹配非数字 字符，使用 `\D`。要匹配非空白字符，使用 `\S`。

正则表达式引擎将所有元字符作为原子对待。

## 字符类

如果允许字符的范围在上述四组里不够具体，通过把它们用中括号围起，你可以自行指定 字符类：

```
my $vowels      = qr/B<[>aeiouB<]>/;
my $maybe_cat = qr/c${vowels}t/;
```

标量变量名 `$vowels` 外的大括号帮助对变量名称消歧。如果没有它，语法分析器将把变量名解释为 `$vowelst`，这样不是因未知变量导致编译期错误就是把已存在 `$vowelst` 变量的内容内插进正则表达式。

如果字符集内的字符构成了一个连续的范围，你可以使用连字符（-）作为表达该范围的 快捷方式。

```
my $letters_only = qr/[a-zA-Z]/;
```

将连字符添加到字符类的开头或结尾可以将其包括进此字符类中：

```
my $interesting_punctuation = qr/[-!?!]/;
```

.....或对其进行转义：

```
my $line_characters = qr/[|=\\_]/;
```

就像单词和数字类元字符（`\w` 和 `\d`）有自己的否定形式，你也可以否定一个字符类。用插入符号（`^`）作为字符类的第一个元素意味着“除 这些外的所有字符”：

```
my $not_a_vowel = qr/[^\aeiou]/;
```

在此之外使用插入符号使其成为该字符类的一个成员。要在否定字符类里包含一个连字符，可以将其放在插入符号后，或者在字符类的最后，再或者对其转义。

## 捕获

通常的做法是先匹配字符串的一部分并在稍后对其进行处理；也许你想从一个字符串中提取一个地址或美国电话号码：

```
my $area_code      = qr/(\d{3})/;
my $local_number   = qr/\d{3}-?\d{4}/;
my $phone_number   = qr/$area_code\s?$local_number/;
```

正则表达式中的括号是元字符；`$area_code` 对它们进行了转义。

## 具名捕获

给出一个字符串，`$contact_info`，包含有联系信息，你可以将 `$phone_number` 正则表达式应用其上并通过 具名捕获 来将任何匹配结果 捕获 并存入变量中：

```
if ($contact_info =~ /(?(phone)>$phone_number)/)
```

```
{
    say "Found a number ${phone}";
}
```

捕捉结构可能看上去像是一大个摇晃的标点，当你可以将其作为整体认读时，它还是比较简单的：

```
(?<capture name> ... )
```

括号包围了整个捕获。`?< name >` 结构必须紧跟左括号。它为捕获缓冲区提供了名称。此结构位于括号内的其余部分是一个正则表达式。如果当正则表达式匹配该片段，Perl 将字符串被捕获的部分存储在神奇变量 `%+` 中：一个以捕获缓冲区名为键、匹配正则表达式的字符串部分为值的哈希。

对于 Perl 5 正则表达式来说，括号是特殊的；默认和常规的 Perl 代码一样，它们的行为就是进行分组。它们也将匹配部分组成的一个或多个原子包围在内。要在正则表达式内使用字面括号，你必须添加反斜杠，就像 `$area_code` 变量里那样。

## 编号捕获

具名捕获是 Perl 5.10 的新功能，但捕获早已在 Perl 中存在了许多年头。你也会碰到编号捕获：

```
if ($contact_info =~ /($phone_number)/)
{
    say "Found a number $1";
}
```

括号把要捕获片段包围在内，但是没有正则表达式元字符给出捕获的名称。作为代替，Perl 将捕获的子字符串存放在一系列以 `$1` 开头的神奇变量中，并延续至正则表达式中提供所有捕获组。Perl 找到的第一个匹配捕获存放在 `$1`，第二个存放在 `$2`，等等。捕获计数起始于捕获的左括号；因而第一个左括号将捕获存入 `$1`，第二个存入 `$2`，等等。

虽然具名捕获的语法比编号捕获来得长一些，但它提供了额外的清晰度。你不需要统计开括号的个数来指出某捕获会被存入 `$4` 还是 `$5`，并且基于较短的正则表达式编写更长的正则表达式相对容易一些，因为它们通常对位置的变更或是否出现在单个原子中不那么敏感。

具名捕获中仍可能发生名称冲突，虽然和发生在编号捕获中的编号冲突相比较少。考虑避免在正则表达式片段中使用捕获；将它们留给顶层正则表达式。

当你将一处匹配在列表上下文中求值时，编号捕获相对不那么令人沮丧：

```
if (my ($number) = $contact_info =~ /($phone_number)/)
{
    say "Found a number $number";
}
```

Perl 将按捕获顺序赋值给左值：

## 成组和选项

前面的例子将全部量词应用于简单原子上。它们也可以应用于一个更为复杂的子模式整体：

```
my $pork = qr/pork/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork?.*?$beans/,
    'maybe pork, definitely beans' );
```

如果手动扩展该正则表达式，结果可能令你感到惊讶：

```
like( 'pork and beans', qr/\Apork?.*?beans/,
      'maybe pork, definitely beans' );
```

这样仍然匹配，但考虑一个更为具体的模式：

```
my $pork = qr/pork/;
my $and  = qr/and/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork? $and? $beans/,
      'maybe pork, maybe and, definitely beans' );
```

一些正则表达式不是匹配这项就是匹配另一项。使用 选项 元字符 (|) 即可：

```
my $rice = qr/rice/;
my $beans = qr/beans/;

like( 'rice', qr/$rice|$beans/, 'Found some rice' );
like( 'beans', qr/$rice|$beans/, 'Found some beans' );
```

选项元字符意味着匹配前述任一片段。但请注意解释为正则表达式片段的内容：

```
like( 'rice', qr/rice|beans/, 'Found some rice' );
like( 'beans', qr/rice|beans/, 'Found some beans' );
unlike( 'rich', qr/rice|beans/, 'Found some weird hybrid' );
```

模式 `rice|beans` 可能会解释为 `ric`，后接 `e` 或 `b`，再跟上 `eans`——但是，这是不正确的。选项总是将离正则表达式分隔符最近的算作 整个 片段，无论该分隔符是模式开头和结尾，外围的括号，还是另一个选项字符或者中括号。

为了减少迷惑性，可以像变量 (`$rice|$beans`) 这样使用具名片段，或者将候选项 包括在非捕获分组 中：

```
my $starches = qr/(? :pasta|potatoes|rice)/;
```

(?:) 序列将一系列原子成组但跳过捕获行为。此例中，它包括了三个选项。

如果你打印一个编译后的正则表达式，你将看到它的字符串化形式包含在一个非捕获分组 内；`qr/rice|beans/` 字符串化为 `(?-xism:rice|beans)`。

## 其他转义序列

Perl 将正则表达式内的若干字符解释为 元字符，它们代表不同于它们字面形式的意义。中括号总是标示一个字符类，括号则将片段成组且可选地进行捕获。

要匹配一个元字符的 字面 实例，可以用反斜杠 (\) 对其进行 转义。因此，\`(` 意指单个左括号而 \`)` 意指单个右中括号。\`.` 指的是一个字面点号，而非“匹配除换行符 外所有字符”的原子。

其他通常需要转义的有用的元字符是管道符 (|) 和美元符号 (\$)。同时不要忘记量词： `*`、`+` 和 `?`。

为避免处处转义（和担心忘记转义内插的值），可以使用 元字符禁用字符。`\Q` 元字符禁用对元字符的处理直到它碰到 `\E` 序列。当取用你无法控制的正则表达式来匹配文本时，这个功能尤其有用：

```
my ($text, $literal_text) = @_;  
  
return $text =~ /\Q$literal_text\E/;
```

`$literal_text` 参数可以包含任何内容——例如字符串 `** ALERT **`。使用 `\Q` 和 `\E`，Perl 不会将“零或多个”量词解释为量词。相反，它会将此正则表达式解释为 `\*\* ALERT \*\*` 并试图匹配字面星号。

在处理来自不可信任的用户输入时须特别小心。构造一个恶意正则表达式对你的程序进行有效的拒绝服务攻击是完全可以办到的。

## 断言

正则表达式锚点（`\A` 和 `\Z`）是一种 正则表达式断言 的形式，字符串需要满足此条件，但并不实际匹配字符串中的某个字符。就是说，正则表达式 `qr/\A/` 将 一直匹配，无论字符串内容为何。元字符 `\b` 和 `\B` 也是断言。

零宽度断言 匹配一个 模式，不仅仅是一个字符串中的条件。最重要的是，它们不消耗它们匹配模式中的位置。例如，你只要找一只“cat（猫）”，你可以是用单词边界断言：

```
my $just_a_cat = qr/cat\b/;
```

.....但如果想找一非灾难性的“cat”，你也许会用到 零宽度否定前瞻断言：

```
my $safe_feline = qr/cat(?!astrophe)/;
```

`(?!...)` 结构仅在 `astrophe` 不紧随其后时匹配短语 `cat`。

零宽度否定前瞻断言：

```
my $disastrous_feline = qr/cat(=astrophe)/;
```

.....仅在短语 `astrophe` 紧随其后时匹配 `cat`。这看上去不怎么有用，一个普通的正则表达式就能完成同样的任务，但考虑下述情况，如果你想在字典中查找所有非“catastrophic”但以 `cat` 开头的单词。一种可能的情况是：

```
my $disastrous_feline = qr/cat(?!astrophe)/;  
  
while (<$words>)  
{  
    chomp;  
    next unless /\A(?<some_cat>$disastrous_feline.*)\Z/;  
    say "Found a non-catastrophe '${some_cat}'";  
}
```

因为断言宽度为零，它不消耗源字符串。因此，带锚点的 `.*\Z` 模式片段必须出现；否则就只将捕获源字符串中的 `cat` 部分。



零宽度后顾断言也是存在的。不像前瞻断言那样，这些断言的模式长度必须固定；你不能在这些模式中使用量词。

要对你的猫绝不会出现于行首做出断言，你可以使用 零宽度否定后顾断言：

```
my $middle_cat = qr/(?<!^>cat/;
```

.....此处的 (?!...) 结构包含定长模式。特别的，你可以用 零宽度肯定后顾断言 表达 cat 必须总是立即出现在空格符之后：

```
my $space_cat = qr/(?<=\s)cat/;
```

.....此处的 (?<=...) 结构包含定长模式。这种方式在用 \G 修饰符进行全局正则表达式匹配时非常有用，但这是一个你不会经常用到的高级特性。

## 正则表达式修饰符

正则表达式操作符允许若干修饰符改变匹配的行为。这些修饰符出现在匹配、替换和 qr// 操作符的结尾。例如，要启用大小写不敏感的匹配：

```
my $pet = 'CaMeLiA';

like( $pet, qr/Camelia/, 'You have a nice butterfly there'
);
like( $pet, qr/Camelia/i, 'Your butterfly has a broken shift key' );
```

第一个 like() 会失败，因为这些字符串包含不同的字母。第二个 like() 将通过，因为 /i 修饰符使得正则表达式忽略大小写的区别。因修饰符的关系，M 和 m 在第二例中是等价的。

你也可以在模式中内嵌修饰符：

```
my $find_a_cat = qr/(?<feline>(?!i)cat)/;
```

(?!i) 语法仅为它所包围的组启用大小写不敏感匹配：此例中，即整个 feline 捕获组。你可以以此形式使用多个修饰符（在模式合适的部分）。你也可以通过前缀 - 来禁用特定的修饰符：

```
my $find_a_rational = qr/(?<number>(?!-i)Rat)/;
```

多行操作符，/m，允许 ^ 和 \$ 锚点匹配字符串中任意行开头和结尾。

/s 修饰符将源字符串作为一行对待，如此 . 元字符便匹配换行符。Damian Conway 对助记符提出建议，/m 修改 多个(multiple) 正则表达式元字符的行为，而 /s 修改 单个(single) 正则表达式元字符的行为。

/x

修饰符允许你在模式中内嵌额外的空白和注释而不会改变它们的原意。此修饰符生效时，正则表达式引擎将空白和注释字符( # )及其后的内容统统作为注释并忽略它们。这允许你编写更可读的正则表达式：

```
my $attr_re = qr{
    ^                # 行首

    # 杂项
```

```

(?:
    [;\n\s]*          # 空白和伪分号
    (?:/\*.*?\*/)?    # C 注释
)*

# 属性标记
ATTR

# 类型
\s+
(
    U?INTVAL
    | FLOATVAL
    | STRING\s+\*
    | PMC\s+\*
    | \w*
)
}x;

```

这个正则表达式不 简单，但注释和空白提高了它的可读性。即便你利用已编译的片段一起编写正则表达式，`/x` 修饰符还是能够帮助提高你的代码质量。

`/g` 修饰符对字符串从头到脚执行某正则表达式行为。它和替换一起使用时候比较合理：

```

# appease the Mitchell estate
my $contents = slurp( $file );
$contents     =~ s/Scarlett O'Hara/Mauve Midway/g;

```

当和匹配一起使用时——并非替换——`\G` 元字符允许你在循环中按块处理字符串。`\G` 在最近一次匹配结束的位置进行匹配。为了按逻辑块处理一个全是美国电话号码的不正确编码文件，你可以编写：

```

while ( $contents =~ /\G(\w{3})(\w{3})(\w{4})/g )
{
    push @numbers, "($1) $2-$3";
}

```

注意 `\G` 锚点将从字符串中前一次迭代匹配的那一点开始着手。如果前一次匹配以诸如 `.*` 之类的贪心匹配结束，则接下来的可以用于匹配的部分将减少。前瞻断言的使用在这里很重要，因为它们不消耗欲匹配的字符串。

`/e` 修饰符允许你在替换操作右边写入任意 Perl 5 代码。如果成功匹配，正则表达式引擎将运行这段代码，并用它的返回值作为替换的值。前面的全局替换例子中，替换不幸主角姓名的部分可以变得更加健壮：

```

# appease the Mitchell estate
my $contents = slurp( $file );
$contents     =~ s{Scarlett( O'Hara)?}
                { 'Mauve' . defined $1 ? ' Midway' : '' }ge;

```

你可以向一次替换操作添加任意多的 `/e` 修饰符。每一处额外的修饰符将对表达式的结果进行又一次的求值，通常只有 Perl 高尔夫手才会使用 `/ee` 以及更加复杂的语句。

## 智能匹配

智能匹配操作符，`~~`，对两个操作符进行比较并在互相匹配时返回真值。定义的模糊恰好反映了此操作符的智能程度：比较操作由操作数两者共同决定。之前你已经见识了这种行为——`given (given_when)` 进行的的就是隐式智能匹配。

参见 `perldoc perlsyn` 中“智能匹配细节”一段以了解更多细节。一些智能匹配的语义 在 Perl 5.10.0 和 Perl 5.10.1 之间已经做出修改，因此在可能时，仅在 5.10.1 及更高版本中使用智能匹配。

智能匹配操作符是一个中缀操作符：

```
say 'They match (somehow)' if $loperand ~~ $roperand;
```

比较的类型大致先由右操作符的类型决定然后再是左操作符。例如，如果右操作符是一个带数值成分的标量，则比较将使用数值等于。如果右操作符是一个正则表达式，则比较将是一个 `grep` 操作或模式匹配。如果右操作符是一个数组，比较将是 `grep` 操作或递归的智能匹配。如果右操作符是一个哈希，比较操作将检查一个或多个键是否存在。

例如：

```
# 标量数值比较
my $x = 10;
my $y = 20;
say 'Not equal numerically' unless $x ~~ $y;

# 标量类数值比较
my $x = 10;
my $y = '10 little endians';
say 'Equal numeric-ishally' if $x ~~ $y;
```

.....以及：

```
my $needlepat = qr/needle/;

say 'Pattern match'          if $needle  ~~ $needlepat;
say 'Grep through array'     if @haystack ~~ $needlepat;
say 'Grep through hash keys' if %hayhash  ~~ $needlepat;
```

.....再及：

```
say 'Grep through array'          if $needlepat  ~~ @haystack;
say 'Array elements exist as hash keys' if %hayhash  ~~ @haystack;
say 'Array elements smart match'    if @strawstack ~~ @haystack;
```

.....又及：

```
say 'Grep through hash keys'          if $needlepat ~~ %hayhash;
say 'Array elements exist as hash keys' if @haystack ~~ %hayhash;
say 'Hash keys identical'              if %hayhash  ~~ %haymap;
```

这些比较操作在某操作数是给出数据类型的 引用 时也能正常工作。举例来说：

```
say 'Hash keys identical' if %hayhash ~~ \%hayhash;
```

你可以在对象上重载（overloading）智能匹配操作符。如果你不这样做，智能匹配

操作符在你尝试将某对象用作操作数时会抛出异常。你也可以使用如 `undef` 等其他数据类型，以及函数引用作为智能匹配操作数。请参考 `perldoc perlsyn` 中的表格来获取更多细节。