模式，然后输出那几行。比如，想看看某个文件在哪一行提到过flint，并且同一行内还跟着stone，可以用下面这条*grep*命令：

```
$ grep 'flint.*stone' chapter*.txt
chapter3.txt:a piece of flint, a stone which may be used to start a fire by striking
chapter3.txt:found obsidian, flint, granite, and small stones of basaltic rock, which
chapter9.txt:a flintlock rifle in poor condition. The sandstone mantle held several
```

$str =~ /\p{Space}/　　#有无空格
$str =~ /\p{Digit}/　　#数字
if($str =~ /\p{Hex}/)　　#16 进制数　【0-9A-Fa-f】
把 p 改为 P，上面表达式表示否定意义

元字符：
.　:匹配任意一个字符，换行符除外
\
*: 匹配 0 或多次
　　.*:匹配任意字符 0-无限次
+ ： 匹配 1 次以上
？： 匹配 0 次或 1 次
（）

模式分组：()
反向引用：\1,\2

圆括号同时也使得重新使用某些字符串成为可能。我们可以用反向引用（*back reference*）来引用圆括号中的模式所匹配的文字，这个行为我们称为捕获组（*capture*

（.）\1 匹配连续出现的两个同样的字符。

$_ = “yabba dabba doo”;
/y(....) d\1/　或者/y(.)(.)\2\1/　#匹配 abba
$_=”aa11bb”
/(.)\g{1}11/ 或者 g1　　#g{N}：N 为组号

|:或匹配

现在可以使用/fred(|\t)+barney/这样的模式来匹配fred和barney之间出现一次以上空格、制表符或两者混合的字符串。加号表示重复一次或更多。每次只要有重复，(|\t)就可能匹配空格或制表符[注13]。在这两个名字之间至少要有一个空格或制表符。

若要求fred与barney之间的字符必须都一样，你可以把上述模式改成/fred(+|\t+)barney/。如此一来，中间的分隔符就一定得全是空格或全是制表符。

字符集：
放在[]中，他只匹配字符集中的单个字符，exp:[abcdefg] 只匹配一个
[a-zA-Z]
[\000-\177]:匹配任意一个 7 位的 ASCII 字符
脱字符：^
[^def]:匹配除 def 以外的任何字符
[^\n-z]:匹配除了 n,-,z 以外的字符
字符集括号外面的连字符-没有特殊意义
\d :数字字符集　各种语言的数字
/a: 严格按照 ASCII 范围匹配数字字符时，(写在末尾) /HAL-[\d]+/a
\s :匹配空白符，同\p{Spacs}

在Perl5.6之前，\s仅能匹配以下5个空白符：换页符（form-feed）、水平制表符（tab）、换行符（newline）、回车符（carriage return）以及空格字符本身。所以，明确定义的字符集应该是[\f\t\n\r]。要在新版Perl当中使用严格表示此范围的字符集，可以使用之前\d例子中一样的办法：

```
use 5.014;

if (/\s/a) { # 按老的 ASCII 字符语义解释
    say 'The string matched ASCII whitespace.';
}
```

Perl 5.10还增加了范围更小的空白符集。比如简写\h，它只匹配水平空白符，而\v则只匹配垂直空白符。把\h和\v并起来，就成了\p{Space}：

\R: 断行符　不管是\r\n 或者\n 都行
\w:单词字符
/s 匹配任意字符，包括换行符

任意字符，那么要是我们只想其中几个点号匹配任意字符呢？可以换用字符集[^\n]，不过输入太麻烦，所以Perl 5.12开始引入了\N简写来表示\n的否定意义。

/x 可以在模式中添加空格，方便阅读
\s \s* \s+ 匹配空白符
/i 大小写无关
/a ascii 码方式
/u Unicode 方式 【更宽泛】
/l 遵从本地化语言设置

```
use 5.014;

/\w+/a        # 仅仅是 A-Z、a-z、0-9、_ 这些字符
/\w+/u        # 任何Unicode当中定义为单词的字符
/\w+/l        # 类同于ASCII的版本，但单词字符的定义取决于本地化设定
              # 所以如果设定为Latin-9的话，Œ也算单词字符
```

若使用两个 a，进一步表示仅仅采用 ascii 方式的大小写映射处理

```
/k/aai      # 只匹配ASCII字符K或k，但不匹配开尔文符号
/k/aia      # 其实/a不必相互紧挨，分开写的效果也是一样的
/ss/aai     # 只匹配ASCII的ss、Ss、sS、SS，不匹配ß
/ff/aai     # 只匹配ASCII的ff、Ff、fF、FF，不匹配ff
```

锚位：

\A: 匹配字符串的绝对开头 ，/\Ahttp:/i　:是否以 http：开头

\z: 匹配字符串的绝对末尾。 /\.png\z/i　:是否以.png 结尾

\Z 行末锚位，允许后面出现换行符

/\A\s*\Z/　:匹配空行【允许包含若干空白符，包括制表符和空格】

^:字符串开头锚位 /^barry/m

$:字符串结尾锚位　/fred$/m

$/m :对多行内容进行匹配

若无/m,^和$的行为同\A,\z

\b 单词[a-z,0-9,_]边界锚位\B　/\bfred\b/ 可匹配 fred 但不能匹配 frederick,afred

不捕获圆括号（?:）：Perl 正则表达式允许使用圆括号分组但不进行捕获，只是分组用

?四种用法：本身，数量可有可无，非贪婪匹配，放弃捕获

```
 #bronto 只是起到匹配作用，并不想用来捕获，$1 就可以捕获（|）了
If(/(?:bronto)?saurus (steak|burger)/){
    Print “Fred wants a $1\n”;
}
```

捕获内容直接命名：　?<LABEL>PATTERN

%+ ：保存捕获组捕获到的内容 ，键即为捕获时用的特殊标签

```
My $names = ‘fred or barney’;
If($names =~ m/(\w+)(and |or)(\w+)/) #对比
If($names =~ m/(?<name1>\w+)(?:and | or)(?<name2>\w+)/){
    Say “I saw $+{name1} and $+{name2}”;      #输出捕获
}
```

```
use 5.010;

my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?<last_name>\w+) and \w+ \g{last_name}/ ) {
    say "I saw $+{last_name}";
}
```

实际上，\k<label>与\g{label}有着细微差别。在两个以上的组有同名标签时，k<label>和\g{label}总是会指代最左边的那组，但\g{N}就可以实现相对反向引用。另外，如果你是Python的爱好者，也可以使用(?P=label)这样的语法。

\k<label> 等效于 \g{label}

```
my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?<last_name>\w+) and \w+ \g{last_name}/ ) {
    say "I saw $+{last_name}";
}
```

```
my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?<last_name>\w+) and \w+ \k<last_name>/ ) {
    say "I saw $+{last_name}";
}
```

不加()也能使用的捕获变量，保存在：$&、$`、$’

If(“hello there,neighbor” =~ /\s(\w+),/){

    Print “that actually matched ‘$&’”;   #输出形式 there

}

$&保存的是【_there,】

目标字符串保存在$1 中：【there】

目标字符串前的内容放在$`: 【_】

目标字符串后的内容放在$’: 【,】

如果你用的是Perl 5.10或以上的版本，那么就更方便了。修饰符/p只会针对特定的正则表达式开启类似的自动捕获变量，但它们的名字不再是$`，$&和$'，而是用${^PREMATCH}、${^MATCH}和${^POSTMATCH}表示。于是，之前的例子可以改写成：

```
use 5.010;
if ("Hello there, neighbor" =~ /\s(\w+),/p) {
    print "That actually matched '${^MATCH}'.\n";
}

if ("Hello there, neighbor" =~ /\s(\w+),/p) {
    print "That was (${^PREMATCH})(${^MATCH})(${^POSTMATCH}).\n";
}
```

模式中的量词：

* 【{0，}】，+【{1，}】，？【{0,1}】，{}

/a{5,15}/ 若 a 出现 20 次只会匹配前 15 次

中使用的元字符。对于表8-1所展示的优先级顺序，大致阐释如下：

1. 最高等级是圆括号（“()”），用于分组和捕获。圆括号里的东西总是比其他东西更富有紧密性。
2. 第二级是量词，也就是重复操作符：星号（*）、加号（+）、问号（?），以及使用花括号表示的量词，比如{5,15}、{3,}和{5}。它们都会和它前面的条目紧密相连。
3. 第三级是锚位和序列。我们已经介绍过的锚位有：\A、\Z、\z、^、$、\b和\B[注28]。序列（彼此相邻的条目）事实上也是操作符，就算没有使用元字符也是如此。这就是说，单词里字母之间的紧密程度和锚位与字母之间的紧密程度是相同的。
4. 第四级是择一竖线（|）。由于它位于优先级表底部，所以从效果上来看，它会把各种模式拆分成数个组件。另外，择一竖线的优先级之所以放在锚位与序列的下面，是因为我们希望类似/fred|barney/的模式中，单词里的字母间的紧密程度高于择一竖线。否则，该模式的解释方式就成了“匹配fre，后面所跟的字母必须是d或者b，然后再跟arney”。所以，择一竖线位于优先级表的底部，这样单词里的字母才会紧密连接在一起，成为一个整体。
5. 最低级别的称为原子（*atoms*）。正是由这些原子构成了大多数基本的模式，比如单独的字符、字符集以及反引用等等。

表8-1 正则表达式优先级

正则表达式特性	示例
圆括号（分组或捕获）	(…)，(?:…)，(?<LABEL>…)
量词	a*，a+，a?，a{n,m}
锚位和序列	abc，^，$，\A，\b，\z，\Z
择一竖线	a|b|c
原子	a，[abc]，\d，\1，\g{2}

```
$_ = "green scaly dinosaur";
s/(\w+) (\w+)/$2, $1/;  # 替换后为"scaly, green dinosaur"
s/^/huge, /;            # 替换后为"huge, scaly, green dinosaur"
s/,.*een//;             # 空替换：此时为"huge dinosaur"
s/green/red/;           # 匹配失败：仍为"huge dinosaur"
s/\w+$/($`!)$&/;        # 替换后为"huge (huge !)dinosaur"
s/\s+(!\W+)/$1 /;       # 替换后为"huge (huge!) dinosaur"
s/huge/gigantic/;       # 替换后为"gigantic (huge!) dinosaur"
```

s///返回的是布尔值，替换成功时为真，否则为假：

```
$_ = "fred flintstone";
if (s/fred/wilma/) {
    print "Successfully replaced fred with wilma!\n";
}
```

/g 可以让 s///进行全局替换

删除开头结尾的空白

S/^\s+//　开头

S/\s+$//　结尾

或者 s/^\s+|\s+$//g　效率慢

不同定界符的写法：

```
s{fred}{barney};
s[fred](barney);
s<fred>#barney#;
```

```
$file_name =~ s#^.*/##s;  # 将$file_name中所有的Unix风格的路径全部去除
```

```
my $original = 'Fred ate 1 rib';
my $copy = $original;
$copy =~ s/\d+ ribs?/10 ribs/;
```

也可以把后面两步并作一步，先做赋值运算，然后针对运算结果进行替换：

```
(my $copy = $original) =~ s/\d+ ribs?/10 ribs/;
```

看起来确实叫人眼花缭乱，因为很多人都会忘记其实左边的赋值运算就好比是普通的字符串，实际做替换的是变量$copy。Perl 5.14增加了一个/r修饰符，专门用于解决这类问题。原先s///操作完成后返回的是成功替换的次数，加上/r之后，就会保留原来字符串变量中的值不变，而把替换结果作为替换操作的返回值返回：

```
use 5.014;

my $copy = $original =~ s/\d+ ribs?/10 ribs/r;
```

形式上看起来和之前的例子差别不大，只不过拿掉了括号罢了。但在这个例子中，运算顺序却是相反的，先做替换，再做赋值。

\U 将它后面的字符全部转义成大写
\L 转成小写
\E 关闭大小写转换
使用小写（\l,\u）只影响紧跟其后的第一个字符

```
$_ = "I saw Barney with Fred.";
s/(fred|barney)/\U$1/gi;  # $_现在成了"I saw BARNEY with FRED."
```

类似地，\L转义符会将它后面的所有字符转换成小写的。沿用前面的例子：

```
s/(fred|barney)/\L$1/gi;  # $_ 现在成了 "I saw barney with fred."
```

默认情况下，它们会影响之后全部的（替换）字符串。你也可以用\E关闭大小写转换的功能：

```
s/(\w+) with (\w+)/\U$2\E with $1/i;  # $_ 替换后为 "I saw FRED with barney."
```

使用小写形式（\l与\u）时，它们只会影响紧跟其后的第一个字符：

```
s/(fred|barney)/\u$1/ig;  # $_ 替换后为 "I saw FRED with Barney."
```

你甚至可以将它们并用。同时使用\u与\L来表示“后续字符全部转为小写的，但首字母大写”[注6]：

```
s/(fred|barney)/\u\L$1/ig;  # $_ 现在成了 "I saw Fred with Barney."
```

附带一提，虽然这里介绍的是替换时的大小写转换，但它们同样可用在任何双引号内的字符串中：

```
print "Hello, \L\u$name\E, would you like to play a game?\n";
```

默认split会以空白符分隔$_中的字符串：

```
my @fields = split;  # 等效于 split /\s+/, $_;
```

一般来说，用在split中的模式就像之前看到的这样简单。但如果你用到更复杂的模式，请避免在模式里使用捕获圆括号，因为这会启动所谓的“分隔符保留模式”（详情请参考*perlfunc*文档）。如果需要在模式中使用分组匹配，请在split里使用非捕获圆括号(?:)的写法，以避免意外。

非贪婪量词：+？，*？，{5,10}？，{8，}？，？？
/fred.+?barney/　匹配最少的字符串

$^I:　内置控制编辑器的值