QRegExp Class

QRegExp 类使用正则表达式提供模式匹配。更多的...

Header:	**#include < QRegExp>**
CMake:	find_package(Qt6 REQUIRED COMPONENTS Core5Compat) target_link_libraries(mytarget PRIVATE Qt6::Core5Compat)
qmake:	QT += core5compat

- 所有成员的列表，包括继承的成员
- QRegExp 是隐式共享类的一部分。

注意：该类中的所有函数都是reentrant。

公共类型

enum	**CaretMode { CaretAtZero, CaretAtOffset, CaretWontMatch }**
enum	**PatternSyntax** { RegExp, RegExp2, Wildcard, WildcardUnix, FixedString, W3CXmlSchema11 }

公共方法

	QRegExp()
	QRegExp(const QString &*pattern*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive, QRegExp::PatternSyntax *syntax* = RegExp)
	QRegExp(const QRegExp &*rx*)
	~QRegExp()
QString	**cap**(int *nth* = 0) const
int	**captureCount**() const
QStringList	**capturedTexts**() const
Qt::CaseSensitivity	**caseSensitivity**() const
int	**countIn**(const QString &*str*) const
QString	**errorString**() const

	QRegExp()
bool	**exactMatch**(const QString &*str*) const
QStringList	**filterList**(const QStringList &*stringList*) const
int	**indexIn**(const QString &*str*, int *offset* = 0, QRegExp::CaretMode *caretMode* = CaretAtZero) const
int	**indexIn**(const QStringList &*list*, int *from*) const
bool	**isEmpty**() const
bool	**isMinimal**() const
bool	**isValid**() const
int	**lastIndexIn**(const QString &*str*, int *offset* = -1, QRegExp::CaretMode *caretMode* = CaretAtZero) const
int	**lastIndexIn**(const QStringList &*list*, int *from*) const
int	**matchedLength**() const
QString	**pattern**() const
QRegExp::PatternSyntax	**patternSyntax**() const
int	**pos**(int *nth* = 0) const
QString	**removeIn**(const QString &*str*) const
QString	**replaceIn**(const QString &*str*, const QString &*after*) const
QStringList	**replaceIn**(const QStringList &*stringList*, const QString &*after*) const
void	**setCaseSensitivity**(Qt::CaseSensitivity *cs*)
void	**setMinimal**(bool *minimal*)
void	**setPattern**(const QString &*pattern*)
void	**setPatternSyntax**(QRegExp::PatternSyntax *syntax*)
QStringList	**splitString**(const QString &*str*, Qt::SplitBehavior *behavior* = Qt::KeepEmptyParts) const
void	**swap**(QRegExp &*other*)
QVariant	**operator QVariant**() const
bool	**operator!=**(const QRegExp &*rx*) const
QRegExp &	**operator=**(const QRegExp &*rx*)
QRegExp &	**operator=**(QRegExp &&*other*)
bool	**operator==**(const QRegExp &*rx*) const

静态公共成员

QString	**escape(const QString &*str*)**

相关非成员

size_t	**qHash(const QRegExp &*key*, size_t *seed* = 0)**
QDataStream &	**operator<<**(QDataStream &*out*, const QRegExp &*regExp*)
QDataStream &	**operator>>**(QDataStream &*in*, QRegExp &*regExp*)

详细说明

该类在 Qt 6 中已弃用。请使用QRegularExpression而不是所有新代码。有关将旧代码从 QRegExp 移植到 QRegularExpression，请参阅{移植到QRegularExpression}

正则表达式或“regexp”是用于匹配文本中的子字符串的模式。这在许多情况下都很有用，例如

验证	**正则表达式可以测试子字符串是否满足某些条件，例如是整数或不包含空格。**
搜寻中	*正则表达式提供比简单子字符串匹配更强大的模式匹配，例如，匹配单词mail、letter或communications之一*，但不匹配单词*email*、*mailman*、*mailer*、*letterbox*等。
搜索和替换	正则表达式可以用不同的子字符串替换所有出现的子字符串，例如，将所有出现的*&*替换为*&* 除非*&*后面已经跟有*amp；*。
字符串分割	正则表达式可用于标识应在何处拆分字符串，例如拆分制表符分隔的字符串。

介绍了正则表达式、Qt 正则表达式语言的描述、一些示例以及函数文档本身。QRegExp 是根据 Perl 的正则表达式语言建模的。它完全支持 Unicode。*QRegExp 还可以在更简单的通配符模式*下使用，该模式类似于命令 shell 中的功能。QRegExp 使用的语法规则可以更改为setPatternSyntax()。特别是，模式语法可以设置为QRegExp::FixedString，这意味着要匹配的模式被解释为普通字符串，即特殊字符（例如反斜杠）不会被转义。

关于正则表达式的一本好书是Jeffrey EF Friedl 的*Mastering Regular Expressions* (Third Edition)，ISBN 0-596-52812-4。

注意：在 Qt 5 中，新的QRegularExpression类提供了正则表达式的 Perl 兼容实现，建议代替 QRegExp。

介绍

正则表达式是由表达式、量词和断言构建的。最简单的表达式是一个字符，例如**x**或**5**。表达式也可以是用方括号括起来的一组字符。**[ABCD]**将匹配**A**或**B**或**C**或**D**。我们可以将这个相同的表达式写为**[AD]**，并且匹配英语字母表中任何大写字母的表达式写为**[AZ]**。

量词指定必须匹配的表达式出现的次数。**x{1,1}**表示匹配一个且仅一个**x**。**x{1,5}**表示匹配包含至少一个**x**但不超过 5 个**x**字符的序列。

请注意，一般来说，正则表达式不能用于检查平衡括号或标签。例如，如果标签未嵌套，则可以编写正则表达式来匹配开始 html及其结束 html ，但如果标签嵌套，则相同的正则表达式将匹配具有错误结束的开始标签。对于片段，第一个将与第一个匹配，这是不正确的。但是，可以编写正确匹配嵌套括号或标签的正则表达式，但前提是嵌套级别的数量是固定且已知的。如果嵌套层数不固定且未知，则不可能编写不会失败的正则表达式。
``<b>````<b>````<b>bold <b>bolder</b></b>````

假设我们希望正则表达式匹配 0 到 99 范围内的整数。至少需要一位数字，因此我们从表达式**[0-9]{1,1}**开始，它只匹配一位数字一次。此正则表达式匹配 0 到 9 范围内的整数。要匹配最多 99 的整数，请将最大出现次数增加到 2，因此正则表达式变为**[0-9]{1,2}**。此正则表达式满足匹配 0 到 99 之间的整数的原始要求，但它也会匹配出现在字符串中间的整数。如果我们希望匹配的整数是整个字符串，则必须使用锚断言^（插入符号）和**$**（美元）。当^是正则表达式中的第一个字符，这意味着正则表达式必须从字符串的开头匹配。当**$**是正则表达式的最后一个字符时，意味着正则表达式必须匹配字符串的末尾。正则表达式变为**^[0-9]{1,2}$**。请注意，断言（例如^和**$**）不匹配字符，而是匹配字符串中的位置。

如果您看过其他地方描述的正则表达式，它们可能看起来与此处显示的不同。这是因为某些字符集和某些量词非常常见，因此需要使用特殊符号来表示它们。**[0-9]**可以用符号**\d**替换。精确匹配一次出现的量词**{1,1}**可以替换为表达式本身，即**x{1,1}**与*x**相同。所以我们的 0 到 99 匹配器可以写成*^\d{1,2}$*。也可以写成**\d\d0,1**，即从字符串的开头开始，匹配一个数字，后面紧跟0或1个数字。在实践中，它会写成^\d\d?$。的? 是量词{0,1}的简写，即出现 0 或 1 次。? 使表达式可选。正则表达式^\d\d?$表示*从字符串的开头开始，匹配一位数字，紧随其后的是 0 或 1 个以上的数字，紧接着是字符串的末尾*。

要编写一个匹配单词“mail”*或*“letter”*或*“correspondence”之一的正则表达式，但不匹配包含这些单词的单词，例如“email”、“mailman”、“mailer”和“letterbox”，从匹配“mail”的正则表达式开始。完整地表达，正则表达式为**m{1,1}a{1,1}i{1,1}l{1,1}**，但由于字符表达式会自动由**{1,1}**量化，因此我们可以简化**mail**的正则表达式，即“m”后跟“a”，后跟“i”，后跟“l”。现在我们可以使用竖线| ，这意味着**或**，包括其他两个词，。匹配“邮件”**或**“信件”**或**“信件”。虽然这个正则表达式将匹配我们想要匹配的三个单词之一，但它也会匹配我们不想匹配的单词，例如“电子邮件”。为了防止正则表达式匹配不需要的单词，我们必须告诉它在单词边界处开始和结束匹配。首先，我们将正则表达式括在括号中，**(mail|letter|correspondence)**。括号将表达式组合在一起，它们标识了我们希望的正则表达式的一部分capture。将表达式括在括号中允许我们将其用作更复杂的正则表达式中的组件。它还允许我们检查这三个单词中的哪一个实际上是匹配的。为了强制匹配在单词边界上开始和结束，我们将正则表达式括在**\b** *单词边界*断言中：**\b(mail|letter|correspondence)\b**。现在正则表达式的意思是：*匹配单词边界，后跟括号中的正则表达式，然后是单词边界*。**\b**断言匹配正则表达式中的*位置，而不是**字符*。单词边界是任何非单词字符，例如空格、换行符或字符串的开头或结尾。

如果我们想用 HTML 实体 & 替换 & 字符**;** ，要匹配的正则表达式很简单**&**。但此正则表达式还将匹配已转换为 HTML 实体的 & 符号。我们只想替换后面没有**amp;的 & 符号。**。为此，我们需要否定的前瞻断言**(?!** __ **)**。然后，正则表达式可以写为**&(?!amp;)**，即匹配后面**不***跟***amp;**的 & 符号。* 。

如果我们想要计算字符串中 'Eric' 和 'Eirik' 的所有出现次数，两个有效的解决方案是**\b(Eric|Eirik)\b**和**\bEi?ri[ck]\b**。需要使用单词边界断言“\b”来避免匹配包含任一名称的单词，例如“Ericsson”。请注意，第二个正则表达式匹配的拼写比我们想要的要多：“Eric”、“Erik”、“Eiric”和“Eirik”。

上面讨论的一些例子是在code examples部分。

字符集的字符和缩写

元素	意义
C	除非具有特殊的正则表达式含义，否则字符代表其自身。例如c匹配字符*c*。
\C	反斜杠后面的字符与字符本身匹配，除非下面指定。例如，要匹配字符串开头的文字插入符号，请编写\^。
\A	匹配 ASCII 响铃 (BEL, 0x07)。
\F	匹配 ASCII 换页符（FF、0x0C）。
\n	匹配 ASCII 换行符（LF、0x0A、Unix 换行符）。
\r	匹配 ASCII 回车符（CR、0x0D）。
\t	匹配 ASCII 水平制表符（HT，0x09）。
\v	匹配 ASCII 垂直制表符 (VT, 0x0B)。
\x*呵呵*	匹配与十六进制数字*hhhh*（0x0000 和 0xFFFF 之间）对应的 Unicode 字符。
\0 *ooo*（即 \零 *ooo*）	*与八进制数ooo*（0 到 0377 之间）的 ASCII/Latin1 字符匹配。
。（点）	匹配任何字符（包括换行符）。
\d	匹配一个数字 (QChar::isDigit())。
\D	匹配非数字。
\s	匹配空白字符 (QChar::isSpace())。
\S	匹配非空白字符。
\w	匹配单词字符 (QChar::isLetterOrNumber(),QChar::isMark（），或者 '_'）。
\W	匹配非单词字符。
\ *n*	第*n*个反向引用，例如 \1、\2 等。

注意： C++ 编译器会转换字符串中的反斜杠。要在正则表达式中包含**，请输入两次，即\\。要匹配反斜杠字符本身，请输入四次，即\\\\。

字符集

方括号表示匹配方括号中包含的任何字符。上述字符集缩写可以出现在方括号中的字符集中。除了字符集缩写和以下两种例外之外，方括号中的字符没有特殊含义。

^	**如果插入符号作为第一个字符出现（即紧接在左方括号之后），则它会否定字符集。[abc]匹配 'a' 或 'b' 或 'c'，但[^abc]匹配除'a' 或 'b' 或 'c'*以外的任何内容。***
	破折号表示字符范围。[WZ]匹配“W”或“X”或“Y”或“Z”

- 破折号表示字符范围。[WZ]匹配"W"或"X"或"Y"或"Z"。
  如果插入符号作为第一个字符出现（即紧接在左方括号之后），则它会否定字符集。[abc]匹配'a'或'b'

使用预定义的字符集缩写比使用跨平台和语言的字符范围更具可移植性。例如，**[0-9]**匹配西方字母表中的数字，但**\d匹配***任何*字母表中的数字。

注意：在其他正则表达式文档中，字符集通常称为"字符类"。

量词

默认情况下，表达式自动由**{1,1}**量化，即它应该只出现一次。在下面的列表中，***E***代表表达式。表达式是一个字符，或者一组字符的缩写，或者方括号中的一组字符，或者括号中的表达式。

***Ei*?**	**匹配零次或一次出现的*E*。这个量词意味着*前面的表达式是可选的*，因为无论是否找到该表达式它都会匹配。*埃*?* ***与E* {0,1}相同。例如，凹痕?* *匹配"dent"或"dents"。**
***E*+**	匹配一次或多次出现的*E*。***E* +与**E* {1,}相同。例如，**0+**匹配 '0'、'00'、'000' 等。
***E* ***	匹配零次或多次出现的*E*。***它与E* {0,}相同。*量词经常用在应该使用+的错误中。*例如，如果在表达式中使用*\s*$*来匹配以空格结尾的字符串，它将匹配每个字符串，因为***\s*$**表示*匹配零个或多个空格，后跟字符串末尾*。匹配至少有一个尾随空白字符的字符串的正确正则表达式是**\s+$**。
***E* {n}**	精确匹配*n*次出现的*E*。***E* {n}与重复**E* *n*次相同。例如，**x{5}与***xxxxx*相同。**它也与E* {n,n}**相同，例如**x{5,5}*。
***E* {n，}**	匹配至少出现*n*次的*E*。
***E* {,m}**	最多匹配*E*出现*m*次。***E* {,m}与*E* {0,m}**相同。
***E* {n,m}**	匹配至少出现*n* 次且最多出现*m*次的*E*。

要将量词应用于不仅仅是前面的字符，请使用括号将表达式中的字符分组在一起。例如，**tag+**匹配"t"，后跟"a"，后跟至少一个"g"，而**(tag)+**匹配至少出现一次"tag"。

注意：量词通常是"贪婪的"。它们总是匹配尽可能多的文本。例如，**0+**匹配它找到的第一个零以及第一个零之后的所有连续零。应用于"20005"，它匹配"2 000 5"。量词可以变得非贪婪，请参阅setMinimal()。

捕获文本

括号允许我们将元素组合在一起，以便我们可以量化和捕获它们。例如，如果我们有匹配字符串的表达式**mail|letter|correspondence**，我们知道*其中一个*单词匹配，但不知道是哪个单词匹配。使用括号允许我们"捕获"在其范围内匹配的任何内容，因此，如果我们使用**(mail|letter|correspondence)**并将此正则表达式与字符串"我向您发送了一些电子邮件"进行匹配，我们可以使用cap（） 或者capturedTexts() 函数提取匹配的字符，在本例中为"mail"。

我们可以在正则表达式本身中使用捕获的文本。为了引用捕获的文本，我们使用从 1 开始索引的*反向引用*，与 cap()。例如，我们可以使用**\b(\w+)\W+\1\b**搜索字符串中的重复单词，这意味着匹配单词边界，后跟一个或多个单词字符，后跟一个或多个非单词字符，后跟相同的字符文本作为第一个带括号的表达式，后跟单词边界。

如果我们想纯粹使用括号进行分组而不是捕获，我们可以使用非捕获语法，例如**(?:green|blue)**。非捕获括号以“(?:”开始并以“)”结束。在此示例中，我们匹配“绿色”或“蓝色”，但我们不捕获匹配，因此我们只知道我们是否匹配，但不知道我们实际找到了哪种颜色。使用非捕获括号比使用捕获括号更有效，因为正则表达式引擎必须执行更少的簿记操作。

捕获括号和非捕获括号都可以嵌套。

由于历史原因，适用于捕获括号的量词（例如* **）比其他量词更加“贪婪”。**例如，**a*(a*)**将匹配“aaa”和 cap(1) ==“aaa”。此行为与其他正则表达式引擎（尤其是 Perl）的行为不同。要获得更直观的捕获行为，请指定 QRegExp::RegExp2到 QRegExp 构造函数或调用setPatternSyntax（QRegExp::RegExp2）。

当无法预先确定匹配的数量时，常见的习惯用法是使用cap() 循环中。例如：

```
QRegExp rx("(\\d+)");
QString str = "Offsets: 12 14 99 231 7";
QStringList list;
int pos = 0;

while ((pos = rx.indexIn(str, pos)) != -1) {
    list << rx.cap(1);
    pos += rx.matchedLength();
}
// list: ["12", "14", "99", "231", "7"]
```

断言

断言在正则表达式中出现的位置对文本进行一些声明，但它们不匹配任何字符。在下面的列表中，***E***代表任何表达式。

^	**插入符号表示字符串的开头。如果您希望匹配文字，^则必须通过编写 来转义它\\^。例如，^#include将仅匹配以字符“#include”*开头的*字符串。（当插入符号是字符集的第一个字符时，它具有特殊含义，请参阅Sets of Characters.)**
$	美元表示字符串的结尾。例如，**\d\s*$**将匹配以数字结尾的字符串（可选后跟空格）。如果您希望匹配文字，$则必须通过编写 来转义它\\$。
\b	一个词的边界。例如，正则表达式**\bOK\b**表示匹配紧接在单词边界（例如字符串开头或空格）之后的字母“O”，然后匹配紧邻另一个单词边界（例如字符串结尾或空格）之前的字母“K”。但请注意，断言实际上并不匹配任何空格，因此如果我们编写**(\bOK\b)并且我们有一个匹配项，即使字符串是“It's** OK now”，它也只会包含“OK”。
\B	非单词边界。**当\b**为假时，该断言为真。例如，如果我们在“Left on”中搜索**\Bon\B，**则匹配将失败（空格和字符串结尾不是非单词边界），但在“t on ne”中会匹配。

^	**插入符号表示字符串的开头。如果您希望匹配文字，^则必须通过编写 来转义它\\^。例如，^#include将仅匹配以字符“#include”*开头的*字符串。（当插入符号是字符集的第一个字符时，它具有特殊含义，请参阅Sets of Characters.)**
(?=*E*)	积极的前瞻。如果表达式在正则表达式中的此时匹配，则此断言为 true。例如，只要**const(?=\s+char)**后面跟着“char”，就匹配“const”，如“static const char *”。（与**const\s+char**进行比较，它匹配“static const char *”。）
(?!*E*)	负向前瞻。如果表达式此时在正则表达式中不匹配，则此断言为 true。例如，**const(?!\s+char)**匹配“const”，*除非*它后面跟着“char”。

通配符匹配

大多数命令 shell（例如*bash*或*cmd.exe*）都支持“文件通配符”，即使用通配符识别一组文件的能力。这setPatternSyntax() 函数用于在正则表达式和通配符模式之间切换。通配符匹配比完整的正则表达式简单得多，并且只有四个功能：

C	**除下面提到的字符外，任何字符都代表其自身。因此c匹配字符*c*。**
?	匹配任何单个字符。它与 相同。在完整的正则表达式中。
*	匹配零个或多个任意字符。**它与完整正则表达式中的.***相同。
[...]	字符集可以用方括号表示，类似于完整的正则表达式。在字符类中，与外部一样，反斜杠没有特殊含义。

在Wildcard模式下，通配符不能被转义。在模式下WildcardUnix，字符 " 转义通配符。

例如，如果我们处于通配符模式并且有包含文件名的字符串，我们可以使用***.html**标识 HTML 文件。这将匹配零个或多个字符，后跟一个点，后跟“h”、“t”、“m”和“l”。

要针对通配符表达式测试字符串，请使用exactMatch()。例如：

```
QRegExp rx("*.txt");
rx.setPatternSyntax(QRegExp::Wildcard);
rx.exactMatch("README.txt");        // returns true
rx.exactMatch("welcome.txt.bak");   // returns false
```

Perl 用户注意事项

Perl 支持的大多数字符类缩写均受 QRegExp 支持，请参阅characters and abbreviations for sets of characters。

在 QRegExp 中，除了在字符类中之外，它^始终表示字符串的开头，因此必须始终对插入符进行转义，除非用于此目的。在 Perl 中，插入符号的含义会根据它出现的位置自动变化，因此很少需要对其进行转义。这同样适用于 $QRegExp 中始终表示字符串结尾的情况。

QRegExp 的量词与 Perl 的贪婪量词相同（但请参见note above）。非贪婪匹配不能应用于单个量词，但可以应用于模式中的所有量词。例如，要匹配 Perl 正则表达式**ro+?m**需要：

```
QRegExp rx("ro+m");
rx.setMinimal(true);
```

Perl/i选项的等价物是setCaseSensitivity（Qt::CaseInsensitive）。

Perl 的/g选项可以使用loop。

在 QRegExp 中。匹配任何字符，因此所有 QRegExp 正则表达式都具有与 Perl/s选项等效的功能。QRegExp 没有与 Perl 的/m选项等效的选项，但是可以通过多种方式来模拟，例如将输入拆分为行或使用搜索换行符的正则表达式进行循环。

因为 QRegExp 是面向字符串的，所以没有 \A、\Z 或 \z 断言。不支持 \G 断言，但可以在循环中模拟。

Perl 的 $,& 是 cap(0) 或capturedTexts()[0]。\`、' 或 $+ 没有 QRegExp 等价物。Perl 的捕获变量 1,2, ... 对应于 cap(1) 或capturedTexts()[1]、cap(2) 或capturedTexts()[2]等

替换模式使用QString::replace()。

/x不支持Perl 的扩展语法，也不支持指令（例如 (?i)）或正则表达式注释（例如 (?#comment)）。另一方面，C++ 的文字字符串规则可用于实现相同的目的：

```
QRegExp mark("\\b"      // word boundary
             "[Mm]ark" // the word we want to match
            );
```

零宽度正向和零宽度负向先行断言 (?=pattern) 和 (?!pattern) 都支持与 Perl 相同的语法。不支持 Perl 的后向断言、“独立”子表达式和条件表达式。

还支持非捕获括号，具有相同的 (?:pattern) 语法。

看QString::split（） 和QStringList::join() 相当于 Perl 的 split 和 join 函数。

注意：因为 C++ 转换 's，所以它们必须在代码中写入*两次*，例如**\b**必须写入**\b**。

代码示例

```
QRegExp rx("^\\d\\d?$");    // match integers 0 to 99
rx.indexIn("123");          // returns -1 (no match)
rx.indexIn("-6");           // returns -1 (no match)
rx.indexIn("6");            // returns 0 (matched at position 0)
```

第三个字符串匹配“ 6 ”。这是一个简单的验证正则表达式，适用于 0 到 99 范围内的整数。

```
QRegExp rx("^\\S+$");       // match strings without whitespace
rx.indexIn("Hello world");  // returns -1 (no match)
rx.indexIn("This_is-OK");   // returns 0 (matched at position 0)
```

第二个字符串匹配“ This_is-OK ”。我们使用字符集缩写“\S”（非空格）和锚点来匹配不包含空格的字符串。

在下面的示例中，我们匹配包含“mail”或“letter”或“correspondence”的字符串，但仅匹配整个单词，即不匹配“email”

```
QRegExp rx("\\b(mail|letter|correspondence)\\b");
rx.indexIn("I sent you an email");     // returns -1 (no match)
rx.indexIn("Please write the letter"); // returns 17
```

第二个字符串匹配“Please write the letter ”。单词“letter”也被捕获（因为括号）。我们可以这样查看我们捕获的文本：

```
QString captured = rx.cap(1); // captured == "letter"
```

这将从第一组捕获括号中捕获文本（从左到右计算捕获左括号）。括号从 1 开始计数，因为 cap(0) 是整个匹配的正则表达式（相当于大多数正则表达式引擎中的“&”）。

```
QRegExp rx("&(?!amp;)");      // match ampersands but not &amp;
QString line1 = "This & that";
line1.replace(rx, "&amp;");
// line1 == "This &amp; that"
QString line2 = "His &amp; hers & theirs";
line2.replace(rx, "&amp;");
// line2 == "His &amp; hers &amp; theirs"
```

这里我们将 QRegExp 传递给QString的replace()函数用新文本替换匹配的文本。

```
QString str = "One Eric another Eirik, and an Ericsson. "
              "How many Eiriks, Eric?";
QRegExp rx("\\b(Eric|Eirik)\\b"); // match Eric or Eirik
int pos = 0;    // where we are in the string
int count = 0;  // how many Eric and Eirik's we've counted
while (pos >= 0) {
    pos = rx.indexIn(str, pos);
    if (pos >= 0) {
        ++pos;      // move along in str
        ++count;    // count our Eric or Eirik
    }
}
```

我们已经使用了indexIn() 函数重复匹配字符串中的正则表达式。pos++请注意，我们可以编写pos += rx.matchedLength()跳过已经匹配的字符串，而不是一次向前移动一个字符。计数将等于 3，匹配“一个Eric、另一个Eirik和一个 Ericsson”。有多少埃里克人，埃里克？它与“Ericsson”或“Eiriks”不匹配，因为它们不受非单词边界的限制。

正则表达式的一种常见用途是将分隔数据行拆分为其组成字段。

```
str = "The Qt Company Ltd\tqt.io\tFinland";
QString company, web, country;
rx.setPattern("^([^\t]+)\t([^\t]+)\t([^\t]+)$");
if (rx.indexIn(str) != -1) {
    company = rx.cap(1);
    web = rx.cap(2);
    country = rx.cap(3);
}
```

在此示例中，我们的输入行的格式为公司名称、网址和国家/地区。不幸的是，正则表达式相当长并且通用性不强——如果我们添加更多字段，代码就会崩溃。一个更简单、更好的解决方案是查找分隔符（本例中为“\t”），并获取周围的文本。这QString::split() 函数可以将分隔符字符串或正则表达式作为参数，并相应地分割字符串。

```
QStringList field = str.split("\t");
```

这里 field[0] 是公司，field[1] 是网址等等。

为了模仿 shell 的匹配，我们可以使用通配符模式。

```
QRegExp rx("*.html");
rx.setPatternSyntax(QRegExp::Wildcard);
rx.exactMatch("index.html");                // returns true
rx.exactMatch("default.htm");               // returns false
rx.exactMatch("readme.txt");                // returns false
```

通配符匹配由于其简单性而非常方便，但任何通配符正则表达式都可以使用完整的正则表达式来定义，例如.***.html$**。请注意，我们不能使用通配符来匹配`.html`和文件，除非我们使用***.htm***，它也将匹配“test.html.bak”。完整的正则表达式为我们提供了所需的精度，**.*.html?$**。`.htm`

QRegExp 可以使用不区分大小写的方式进行匹配setCaseSensitivity()，并且可以使用非贪婪匹配，参见setMinimal()。默认情况下 QRegExp 使用完整的正则表达式，但这可以通过以下方式更改setPatternSyntax()。搜索可以通过以下方式进行indexIn() 或向后lastIndexIn()。可以使用以下方式访问捕获的文本capturedTexts() 返回所有捕获字符串的字符串列表，或使用cap() 返回给定索引捕获的字符串。这pos() 函数采用匹配索引并返回字符串中进行匹配的位置（如果没有匹配则返回 -1）。

移植到 QRegularExpression

这QRegularExpressionQt 5 中引入的类实现了与 Perl 兼容的正则表达式，并且在提供的 API、支持的模式语法和执行速度方面比 QRegExp 有了很大的改进。最大的区别在于QRegularExpression只是保存一个正则表达式，当请求匹配时它不会*被修改*。相反，一个QRegularExpressionMatch返回对象，以检查匹配结果并提取捕获的子字符串。这同样适用于全局匹配和QRegularExpressionMatchIterator。

其他差异概述如下。

笔记：QRegularExpression不支持与 Perl 兼容的正则表达式中可用的所有功能。最值得注意的是，不支持捕获组的重复名称，并且使用它们可能会导致未定义的行为。这可能会在 Qt 的未来版本中发生变化。

不同的模式语法

将正则表达式从 QRegExp 移植到QRegularExpression可能需要更改模式本身。

在特定场景中，QRegExp 过于宽松，接受的模式在使用时根本无效QRegularExpression。这些很容易检测到，因为QRegularExpression使用这些模式构建的对象无效（请参阅QRegularExpression::isValid()）。

在其他情况下，模式从 QRegExp 移植到QRegularExpression可能会默默地改变语义。因此，有必要审查所使用的模式。最值得注意的静默不兼容案例是：

- 需要花括号来使用十六进制转义符，例如`\xHHHH`超过 2 位数字。`\x2022`需要移植类似的模式`\x{2022}`，否则它将匹配空格 (`0x20`) 后跟字符串`"22"`。一般来说，强烈建议始终使用花括号进行`\x`转义，无论指定的位数是多少。
- `{,n}`需要移植0 到 n 的量化`{0,n}`以保留语义。否则，诸如 之类的模式`\d{,3}`将匹配后跟精确字符串 的数字`"{,3}"`。
- QRegExp 默认情况下进行 Unicode 感知匹配，而QRegularExpression需要单独的选项；请参阅下面的更多细节。

- QRegExp 中的 c{.} 默认匹配所有字符，包括换行符。QRegularExpression默认情况下排除换行符。要包含换行符，请设置QRegularExpression::DotMatchesEverythingOption图案选项。

有关支持的正则表达式语法的概述QRegularExpression，请参考pcrepattern(3)手册页，描述 PCRE（Perl 兼容正则表达式的参考实现）支持的模式语法。

从 QRegExp::exactMatch() 移植

QRegExp::exactMatch() 有两个目的：它将正则表达式与主题字符串精确匹配，并实现部分匹配。

从 QRegExp 的精确匹配移植

精确匹配表示正则表达式是否匹配整个主题字符串。例如，类在主题字符串上产生`"abc123"`：

	QRegExp::exactMatch()	QRegularExpressionMatch::hasMatch()
`"\\d+"`	**错误的**	**真的**
`"[a-z]+\\d+"`	**真的**	**真的**

精确匹配并没有体现在QRegularExpression。如果您想确保主题字符串与正则表达式完全匹配，您可以使用QRegularExpression::anchoredPattern （） 功能：

```
QString p("a .*|pattern");

// re matches exactly the pattern string p
QRegularExpression re(QRegularExpression::anchoredPattern(p));
```

从 QRegExp 的部分匹配移植

使用时QRegExp::exactMatch()，如果没有找到精确匹配，仍然可以通过调用来找出主题字符串与正则表达式匹配的部分QRegExp::matchedLength()。如果返回的长度等于主题字符串的长度，则可以得出结论：找到了部分匹配。

QRegularExpression通过适当的方式显式支持部分匹配QRegularExpression::MatchType。

全局匹配

由于 QRegExp API 的限制，无法正确实现全局匹配（即像 Perl 那样）。特别是，可以匹配 0 个字符的模式（例如`"a*"`）是有问题的。

QRegularExpression::globalMatch() 正确实现了 Perl 全局匹配，并且返回的迭代器可用于检查每个结果。

例如，如果您有如下代码：

```
QString subject("the quick fox");

int offset = 0;
QRegExp re("(\\w+)");
while ((offset = re.indexIn(subject, offset)) != -1) {
    offset += re.matchedLength();
    // ...
}
```

您可以将其重写为：

```
QString subject("the quick fox");

QRegularExpression re("(\\w+)");
QRegularExpressionMatchIterator i = re.globalMatch(subject);
while (i.hasNext()) {
    QRegularExpressionMatch match = i.next();
    // ...
}
```

Unicode 属性支持

使用 QRegExp 时，诸如`\w`、`\d`等字符类将字符与相应的 Unicode 属性进行匹配：例如，`\d`将任何字符与 Unicode `Nd`（十进制数字）属性进行匹配。

使用时这些字符类默认只匹配 ASCII 字符QRegularExpression: 例如，`\d`完全匹配 ASCII 范围内的字符`0-9`。可以通过使用来改变这种行为QRegularExpression::UseUnicodePropertiesOption图案选项。

通配符匹配

没有直接的方法可以进行通配符匹配 QRegularExpression 。但是，那QRegularExpression::wildcardToRegularExpression提供 () 方法将 glob 模式转换为可用于该目的的 Perl 兼容正则表达式。

例如，如果您有如下代码：

```
QRegExp wildcard("*.txt");
wildcard.setPatternSyntax(QRegExp::Wildcard);
```

您可以将其重写为：

```
auto wildcard = QRegularExpression(QRegularExpression::wildcardToRegularExpression("*.txt"));
```

但请注意，某些类似 shell 的通配符模式可能不会转换为您所期望的。如果简单地使用上述函数进行转换，以下示例代码将默默地中断：

```
const QString fp1("C:/Users/dummy/files/content.txt");
const QString fp2("/home/dummy/files/content.txt");

QRegExp re1("*/files/*");
re1.setPatternSyntax(QRegExp::Wildcard);
re1.exactMatch(fp1); // returns true
re1.exactMatch(fp2); // returns true

// but converted with QRegularExpression::wildcardToRegularExpression()

QRegularExpression re2(QRegularExpression::wildcardToRegularExpression("*/files/*"));
re2.match(fp1).hasMatch(); // returns false
re2.match(fp2).hasMatch(); // returns false
```

这是因为，默认情况下，返回的正则表达式QRegularExpression::wildcardToRegularExpression() 完全锚定。要获取未锚定的正则表达式，请传递QRegularExpression::UnanchoredWildcardConversion作为转换选项：

```
QRegularExpression re3(QRegularExpression::wildcardToRegularExpression(
                           "*/files/*", QRegularExpression::UnanchoredWildcardConversion));
re3.match(fp1).hasMatch(); // returns true
re3.match(fp2).hasMatch(); // returns true
```

最小匹配

QRegExp::setMinimal() 通过简单地反转量词的贪婪性来实现最小匹配（QRegExp 不支持惰性量词，如*?,+?等）。QRegularExpression相反，它支持贪婪、懒惰和所有格量词。这QRegularExpression::InvertedGreedinessOption模式选项可用于模拟效果QRegExp::setMinimal()：如果启用，它会反转量词的贪婪性（贪婪的量词变得懒惰，反之亦然）。

插入符号模式

这QRegularExpression::AnchorAtOffsetMatchOptionmatch 选项可用于模拟QRegExp::CaretAtOffset行为。没有其他的等价物QRegExp::CaretMode模式。

也可以看看QString,QStringList， 和QSortFilterProxyModel。

成员类型文档

enum QRegExp::CaretMode

CaretMode 枚举定义了正则表达式中插入符 (^) **的不同含义。**可能的值为：

持续的	价值	描述
`QRegExp::CaretAtZero`	`0`	插入符号对应于搜索字符串中的索引 0。
`QRegExp::CaretAtOffset`	`1`	插入符号对应于搜索的起始偏移量。
`QRegExp::CaretWontMatch`	`2`	插入符号从不匹配。

enum QRegExp::PatternSyntax

用于解释模式含义的语法。

持续的	价值	描述
`QRegExp::RegExp`	`0`	丰富的类似 Perl 的模式匹配语法。这是默认设置。
`QRegExp::RegExp2`	`3`	与 RegExp 类似，但带有greedy quantifiers。（在 Qt 4.2 中引入。）
`QRegExp::Wildcard`	`1`	这提供了一种简单的模式匹配语法，类似于 shell（命令解释器）用于“文件通配”的语法。看QRegExp wildcard matching。
`QRegExp::WildcardUnix`	`4`	这与通配符类似，但具有 Unix shell 的行为。通配符可以用字符“\”进行转义。

持续的	价值	描述
QRegExp::FixedString	2	模式是一个固定的字符串。这相当于在字符串上使用 RegExp 模式，其中所有元字符都使用转义escape()。
QRegExp::W3CXmlSchema11	5	该模式是 W3C XML Schema 1.1 规范定义的正则表达式。

也可以看看setPatternSyntax()。

成员函数文档

QRegExp::QRegExp()

构造一个空的正则表达式。

也可以看看isValid（） 和errorString()。

[explicit]QRegExp::QRegExp(const QString &pattern, Qt::CaseSensitivity cs = Qt::CaseSensitive, QRegExp::PatternSyntax syntax = RegExp)

构造给定的正则表达式对象*pattern*细绳。如果满足以下条件，则必须使用通配符表示法给出模式：*syntax*是Wildcard; 默认是RegExp。该模式区分大小写，除非*cs*是Qt::CaseInsensitive。匹配是贪婪的（最大），但可以通过调用来更改setMinimal()。

也可以看看setPattern(),setCaseSensitivity（）， 和setPatternSyntax()。

QRegExp::QRegExp(const QRegExp &rx)

构造一个正则表达式作为以下内容的副本*rx*。

也可以看看operator=()。

QRegExp::~QRegExp()

破坏正则表达式并清理其内部数据。

QString QRegExp::cap(int nth = 0) const

返回捕获的文本*nth*子表达式。整个匹配的索引为 0，带括号的子表达式的索引从 1 开始（不包括非捕获括号）。

```
QRegExp rxlen("(\\d+)(?:\\s*)(cm|inch)");
int pos = rxlen.indexIn("Length: 189cm");
if (pos > -1) {
    QString value = rxlen.cap(1); // "189"
    QString unit = rxlen.cap(2);  // "cm"
    // ...
}
```

cap() 匹配的元素顺序如下。第一个元素 cap(0) 是整个匹配字符串。每个后续元素对应于下一个捕获左括号。因此 cap(1) 是第一个捕获括号的文本，cap(2) 是第二个捕获括号的文本，依此类推。

也可以看看capturedTexts（） 和pos()。

int QRegExp::captureCount() const

返回正则表达式中包含的捕获数。

QStringList QRegExp::capturedTexts() const

返回捕获的文本字符串的列表。

列表中的第一个字符串是整个匹配的字符串。每个后续列表元素都包含一个与正则表达式的（捕获）子表达式匹配的字符串。

例如：

```
QRegExp rx("(\\d+)(\\s*)(cm|inch(es)?)");
int pos = rx.indexIn("Length: 36 inches");
QStringList list = rx.capturedTexts();
// list is now ("36 inches", "36", " ", "inches", "es")
```

上面的例子还捕获了可能存在但我们不感兴趣的元素。这个问题可以通过使用非捕获括号来解决：

```
QRegExp rx("(\\d+)(?:\\s*)(cm|inch(?:es)?)");
int pos = rx.indexIn("Length: 36 inches");
QStringList list = rx.capturedTexts();
// list is now ("36 inches", "36", "inches")
```

请注意，如果要迭代列表，则应该迭代副本，例如

```
QStringList list = rx.capturedTexts();
QStringList::iterator it = list.begin();
while (it != list.end()) {
    myProcessing(*it);
    ++it;
}
```

某些正则表达式可以匹配不确定的次数。例如，如果输入字符串是“Offsets: 12 14 99 231 7”并且正则表达式 ,rx是(\d+)+，我们希望获得所有匹配数字的列表。但是，调用 后rx.indexIn(str)， capturedTexts() 将返回列表 ("12", "12")，即整个匹配项是“12”，第一个匹配的子表达式是“12”。正确的方法是使用cap （） 在一个loop。

字符串列表中元素的顺序如下。第一个元素是整个匹配字符串。每个后续元素对应于下一个捕获左括号。因此，capturedTexts()[1] 是第一个捕获括号的文本，capturedTexts()[2] 是第二个捕获括号的文本，依此类推（对应于其他一些正则表达式语言中的 1、2 等）。

也可以看看cap （） 和pos()。

Qt::CaseSensitivity QRegExp::caseSensitivity() const

退货Qt::CaseSensitive正则表达式是否区分大小写匹配；否则返回Qt::CaseInsensitive。

也可以看看setCaseSensitivity(),patternSyntax(),pattern （） ， 和isMinimal()。

int QRegExp::countIn(const QString &str) const

返回此正则表达式匹配的次数*str*。

也可以看看indexIn(),lastIndexIn （） ， 和replaceIn()。

QString QRegExp::errorString() const

返回一个文本字符串，解释为什么正则表达式模式在这种情况下无效；否则返回“没有发生错误”。

也可以看看isValid()。

[static]QString QRegExp::escape(const QString &str)

返回字符串*str*每个正则表达式特殊字符都用反斜杠转义。特殊字符有 $、(,)、*、+、.、?、[、,]、^、{、|和 }。

例子：

```
s1 = QRegExp::escape("bingo");   // s1 == "bingo"
s2 = QRegExp::escape("f(x)");    // s2 == "f\\(x\\)"
```

此函数对于动态构造正则表达式模式很有用：

```
QRegExp rx("(" + QRegExp::escape(name) +
           "|" + QRegExp::escape(alias) + ")");
```

也可以看看setPatternSyntax()。

bool QRegExp::exactMatch(const QString &str) const

返回`true`如果*str*与该正则表达式完全匹配；否则返回`false`. 您可以通过调用确定匹配了多少字符串matchedLength()。

对于给定的正则表达式字符串 R，exactMatch("R") 相当于indexIn("^R$") 因为 exactMatch() 有效地将正则表达式包含在字符串的开头和字符串锚点的结尾，除了它设置matchedLength() 不同。

例如，如果正则表达式为**blue**，则 exactMatch()`true`仅返回 input `blue`。对于输入`bluebell`,`blutak`和`lightblue`，exactMatch() 返回`false`和matchedLength() 将分别返回 4、3 和 0。

虽然是 const，但是这个函数设置了matchedLength(),capturedTexts（），和pos()。

也可以看看indexIn（） 和lastIndexIn()。

QStringList QRegExp::filterList(const QStringList &stringList) const

返回与此正则表达式匹配的所有字符串的列表*stringList*。

int QRegExp::indexIn(const QString &str, int offset = 0, QRegExp::CaretMode caretMode = CaretAtZero) const

尝试在以下位置查找匹配项*str*从位置*offset*（默认为 0）。如果*offset*为-1时，从最后一个字符开始搜索；如果为 -2，则位于倒数第二个字符；ETC。

返回第一个匹配的位置，如果没有匹配则返回 -1。

这*caretMode*参数可用于指示^是否应在索引 0 或 at 处匹配*offset*。

您可能更喜欢使用QString::indexOf(),QString::contains（），甚至QStringList::filter()。要替换火柴，请使用QString::replace()。

例子：

```
QString str = "offsets: 1.23 .50 71.00 6.00";
QRegExp rx("\\d*\\.\\d+");    // primitive floating point matching
int count = 0;
int pos = 0;
while ((pos = rx.indexIn(str, pos)) != -1) {
    ++count;
    pos += rx.matchedLength();
}
// pos will be 9, 14, 18 and finally 24; count will end up as 4
```

虽然是 const，但是这个函数设置了matchedLength(),capturedTexts（） 和pos()。

如果QRegExp是一个通配符表达式（参见setPatternSyntax()) 并且想要针对整个通配符表达式测试字符串，请使用exactMatch() 来代替这个函数。

也可以看看lastIndexIn（） 和exactMatch()。

int QRegExp::indexIn(const QStringList &list, int from) const

返回此正则表达式的第一个完全匹配的索引位置*list*,从索引位置向前搜索*from*。如果没有匹配的项目，则返回 -1。

也可以看看lastIndexIn（） 和exactMatch()。

bool QRegExp::isEmpty() const

`true`如果模式字符串为空则返回；否则返回 false。

如果你打电话exactMatch() 在空字符串上使用空模式，它将返回 true；否则它会返回，`false`因为它对整个字符串进行操作。如果你打电话indexIn() *在任何字*符串上使用空模式，它将返回起始偏移量（默认为 0），因为空模式与字符串开头的“空”相匹配。在这种情况下，返回的匹配长度matchedLength() 将为 0。

看QString::isEmpty()。

bool QRegExp::isMinimal() const

`true`如果启用最小（非贪婪）匹配，则返回；否则返回`false`.

也可以看看caseSensitivity（） 和setMinimal()。

bool QRegExp::isValid() const

返回正true则表达式是否有效；否则返回 false。无效的正则表达式永远不会匹配。

模式**[az**是无效模式的示例，因为它缺少右方括号。

请注意，正则表达式的有效性还可能取决于通配符标志的设置，例如***.html**是有效的通配符正则表达式，但是无效的完整正则表达式。

也可以看看errorString()。

int QRegExp::lastIndexIn(const QString &str, int offset = -1, QRegExp::CaretMode caretMode = CaretAtZero) const

尝试向后查找匹配项*str*从位置*offset*。如果*offset*为-1（默认值），则从最后一个字符开始搜索；如果为 -2，则位于倒数第二个字符；ETC。

返回第一个匹配的位置，如果没有匹配则返回 -1。

这*caretMode*参数可用于指示^是否应在索引 0 或 at 处匹配*offset*。

虽然是 const，但是这个函数设置了matchedLength(),capturedTexts （） 和pos()。

警告：向后搜索比向前搜索慢得多。

也可以看看indexIn （） 和exactMatch()。

int QRegExp::lastIndexIn(const QStringList &list, int from) const

返回此正则表达式最后一次完全匹配的索引位置*list*，从索引位置向后搜索*from*。如果*from*为 -1（默认值），则从最后一项开始搜索。如果没有匹配的项目，则返回 -1。

也可以看看QRegExp::exactMatch()。

int QRegExp::matchedLength() const

返回最后一个匹配字符串的长度，如果没有匹配则返回 -1。

也可以看看exactMatch(),indexIn （） ， 和lastIndexIn()。

QString QRegExp::pattern() const

返回正则表达式的模式字符串。该模式具有正则表达式语法或通配符语法，具体取决于patternSyntax()。

也可以看看setPattern(),patternSyntax（），和caseSensitivity()。

QRegExp::PatternSyntax QRegExp::patternSyntax() const

返回正则表达式使用的语法。默认为QRegExp::RegExp。

也可以看看setPatternSyntax(),pattern（），和caseSensitivity()。

int QRegExp::pos(int nth = 0) const

返回的位置*nth*捕获搜索字符串中的文本。如果*nth*为 0（默认值），pos() 返回整个匹配的位置。

例子：

```
QRegExp rx("/([a-z]+)/([a-z]+)");
rx.indexIn("Output /dev/null");   // returns 7 (position of /dev/null)
rx.pos(0);                        // returns 7 (position of /dev/null)
rx.pos(1);                        // returns 8 (position of dev)
rx.pos(2);                        // returns 12 (position of null)
```

对于零长度匹配，pos() 始终返回 -1。（例如，如果 cap(4) 返回空字符串，则 pos(4) 返回 -1。）这是实现的一个功能。

也可以看看cap（） 和capturedTexts()。

QString QRegExp::removeIn(const QString &str) const

删除此正则表达式的所有出现位置*str*，并返回结果

与以下相同replaceIn(str, QString())。

也可以看看indexIn(),lastIndexIn（），和replaceIn()。

QString QRegExp::replaceIn(const QString &str, const QString &after) const

替换此正则表达式的每个出现位置*str*和*after*并返回结果。

对于包含以下内容的正则表达式capturing parentheses，出现**\1**，**\2**，...，在*after*替换为rx.cap(1), cap(2), ...

也可以看看indexIn(),lastIndexIn（），和QRegExp::cap()。

QStringList QRegExp::replaceIn(const QStringList &stringList, const QString &after) const

替换每个中出现的该正则表达式*stringList*与*after*。返回对字符串列表的引用。

void QRegExp::setCaseSensitivity(Qt::CaseSensitivity cs)

将区分大小写的匹配设置为*cs*。

如果*cs*是Qt::CaseSensitive, **.txt$**匹配`readme.txt`但不匹配`README.TXT`。

也可以看看caseSensitivity(),setPatternSyntax(),setPattern（），和setMinimal()。

void QRegExp::setMinimal(bool minimal)

启用或禁用最小匹配。如果*minimal*为 false，匹配为贪婪（最大），这是默认值。

例如，假设我们有输入字符串“我们必须**大胆**，非常**大胆**！”和模式**.**。*使用默认的贪婪（最大）匹配，匹配是“我们必须**大胆**，非常**大胆**！”。但通过最小（非贪婪）匹配，第一个匹配是：“我们必须**大胆**，非常**大胆**！”第二个匹配是“我们必须**大胆**，非常**大胆**！”。在实践中，我们可以使用模式*I*^<I*来代替，尽管这对于嵌套标签仍然会失败。

也可以看看isMinimal（） 和setCaseSensitivity()。

void QRegExp::setPattern(const QString &pattern)

将模式字符串设置为*pattern*。区分大小写、通配符和最小匹配选项不会更改。

也可以看看pattern(),setPatternSyntax（），和setCaseSensitivity()。

void QRegExp::setPatternSyntax(QRegExp::PatternSyntax syntax)

设置正则表达式的语法模式。默认为QRegExp::RegExp。

环境*syntax*到QRegExp::Wildcard实现简单的类似 shell 的功能QRegExp wildcard matching。例如，**r*.txt**匹配通配符模式下的字符串`readme.txt`，但不匹配`readme`。

环境*syntax*到QRegExp::FixedString意味着该模式被解释为纯字符串。那么特殊字符（例如反斜杠）不需要转义。

也可以看看patternSyntax(),setPattern(),setCaseSensitivity（），和escape()。

QStringList QRegExp::splitString(const QString &str, Qt::SplitBehavior behavior = Qt::KeepEmptyParts) const

分裂*str*转换为正则表达式匹配的子字符串，并返回这些字符串的列表。如果此正则表达式与字符串中的任何位置都不匹配，则 split() 返回一个单元素列表，其中包含*str*。

如果*behavior*被设定为Qt::KeepEmptyParts，空字段包含在结果列表中。

也可以看看QStringList::join（） 和QString::split()。

void QRegExp::swap(QRegExp &other)

交换正则表达式*other*用这个正则表达式。这个操作非常快并且永远不会失败。

QVariant QRegExp::operator QVariant() const

将正则表达式返回为QVariant

bool QRegExp::operator!=(const QRegExp &rx) const

`true`如果此正则表达式不等于则返回*rx*; 否则返回`false`.

也可以看看operator==()。

QRegExp &QRegExp::operator=(const QRegExp &rx)

复制正则表达式*rx*并返回对副本的引用。区分大小写、通配符和最小匹配选项也会被复制。

QRegExp &QRegExp::operator=(QRegExp &&other)

移动分配*other*对此QRegExp实例。

bool QRegExp::operator==(const QRegExp &rx) const

`true`如果此正则表达式等于则返回*rx*; 否则返回`false`.

二QRegExp如果对象具有相同的模式字符串以及相同的区分大小写、通配符和最小匹配设置，则它们是相等的。

相关非成员

size_t qHash(const QRegExp &key, size_t seed = 0)

返回哈希值*key*， 使用*seed*为计算提供种子。

QDataStream &operator<<(QDataStream &out, const QRegExp &regExp)

写入正则表达式*regExp*流式传输*out*。

也可以看看Serializing Qt Data Types。

QDataStream &operator>>(QDataStream &in, QRegExp &regExp)

从流中读取正则表达式*in*进入*regExp*。

也可以看看Serializing Qt Data Types。