QString Class

QString 类提供 Unicode 字符串。更多的...

Header:	#include
CMake:	find_package(Qt6 REQUIRED COMPONENTS Core) target_link_libraries(mytarget PRIVATE Qt6::Core)
qmake:	QT += core

- 所有成员的列表，包括继承的成员
- 已弃用的成员
- QString 是隐式共享类和字符串数据类的一部分。

注意：该类中的所有函数都是reentrant。

公共类型

	ConstIterator
	Iterator
enum	**NormalizationForm** { NormalizationForm_D, NormalizationForm_C, NormalizationForm_KD, NormalizationForm_KC }
enum	**SectionFlag** { SectionDefault, SectionSkipEmpty, SectionIncludeLeadingSep, SectionIncludeTrailingSep, SectionCaseInsensitiveSeps }
flags	**SectionFlags**
	const_iterator
	const_pointer
	const_reference
	const_reverse_iterator
	difference_type
	iterator
	pointer
	reference
	reverse_iterator

	ConstIterator
	size_type
	value_type

公共职能

	QString()
	QString(const QChar **unicode*, qsizetype *size* = -1)
	QString(QChar *ch*)
	QString(qsizetype *size*, QChar *ch*)
	QString(QLatin1StringView *str*)
	QString(const char8_t **str*)
	QString(const char **str*)
	QString(const QByteArray &*ba*)
	QString(const QString &*other*)
	QString(QString &&*other*)
	~QString()
QString &	**append**(const QString &*str*)
QString &	**append**(QChar *ch*)
QString &	**append**(const QChar **str*, qsizetype *len*)
QString &	**append**(QStringView *v*)
QString &	**append**(QLatin1StringView *str*)
QString &	**append**(QUtf8StringView *str*)
QString &	**append**(const char **str*)
QString &	**append**(const QByteArray &*ba*)
QString	**arg**(const QString &*a*, int *fieldWidth* = 0, QChar *fillChar* = u' ') const
QString	**arg**(qlonglong *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const
QString	**arg**(qulonglong *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const
QString	**arg**(long *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const
QString	**arg**(ulong *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const

	QString()
QString	**arg**(int *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const
QString	**arg**(uint *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const
QString	**arg**(short *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const
QString	**arg**(ushort *a*, int *fieldWidth* = 0, int *base* = 10, QChar *fillChar* = u' ') const
QString	**arg**(double *a*, int *fieldWidth* = 0, char *format* = 'g', int *precision* = -1, QChar *fillChar* = u' ') const
QString	**arg**(char *a*, int *fieldWidth* = 0, QChar *fillChar* = u' ') const
QString	**arg**(QChar *a*, int *fieldWidth* = 0, QChar *fillChar* = u' ') const
QString	**arg**(QStringView *a*, int *fieldWidth* = 0, QChar *fillChar* = u' ') const
QString	**arg**(QLatin1StringView *a*, int *fieldWidth* = 0, QChar *fillChar* = u' ') const
QString	**arg**(Args &&... *args*) const
const QChar	**at**(qsizetype *position*) const
QChar	**back**() const
QChar &	**back**()
QString::iterator	**begin**()
QString::const_iterator	**begin**() const
qsizetype	**capacity**() const
QString::const_iterator	**cbegin**() const
QString::const_iterator	**cend**() const
void	**chop**(qsizetype *n*)
QString	**chopped**(qsizetype *len*) const
void	**clear**()
int	**compare**(const QString &*other*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
int	**compare**(QLatin1StringView *other*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
int	**compare**(QStringView *s*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
int	**compare**(QChar *ch*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
QString::const_iterator	**constBegin**() const
const QChar *	**constData**() const
QString::const_iterator	**constEnd**() const

	QString()
bool	**contains**(const QString &*str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**contains**(QChar *ch*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**contains**(QLatin1StringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**contains**(QStringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**contains**(const QRegularExpression &*re*, QRegularExpressionMatch **rmatch* = nullptr) const
qsizetype	**count**(const QString &*str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**count**(QChar *ch*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**count**(QStringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**count**(const QRegularExpression &*re*) const
QString::const_reverse_iterator	**crbegin**() const
QString::const_reverse_iterator	**crend**() const
QChar *	**data**()
const QChar *	**data**() const
QString::iterator	**end**()
QString::const_iterator	**end**() const
bool	**endsWith**(const QString &*s*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**endsWith**(QStringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**endsWith**(QLatin1StringView *s*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**endsWith**(QChar *c*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
QString::iterator	**erase**(QString::const_iterator *first*, QString::const_iterator *last*)
QString::iterator	**erase**(QString::const_iterator *it*)
QString &	**fill**(QChar *ch*, qsizetype *size* = -1)
QString	**first**(qsizetype *n*) const
QChar	**front**() const
QChar &	**front**()
qsizetype	**indexOf**(QLatin1StringView *str*, qsizetype *from* = 0, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const

	QString()
qsizetype	**indexOf**(QChar *ch*, qsizetype *from* = 0, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**indexOf**(const QString &*str*, qsizetype *from* = 0, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**indexOf**(QStringView *str*, qsizetype *from* = 0, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**indexOf**(const QRegularExpression &*re*, qsizetype *from* = 0, QRegularExpressionMatch **rmatch* = nullptr) const
QString &	**insert**(qsizetype *position*, const QString &*str*)
QString &	**insert**(qsizetype *position*, QChar *ch*)
QString &	**insert**(qsizetype *position*, const QChar **unicode*, qsizetype *size*)
QString &	**insert**(qsizetype *position*, QStringView *str*)
QString &	**insert**(qsizetype *position*, QLatin1StringView *str*)
QString &	**insert**(qsizetype *position*, QUtf8StringView *str*)
QString &	**insert**(qsizetype *position*, const char **str*)
QString &	**insert**(qsizetype *position*, const QByteArray &*str*)
bool	**isEmpty**() const
bool	**isLower**() const
bool	**isNull**() const
bool	**isRightToLeft**() const
bool	**isUpper**() const
bool	**isValidUtf16**() const
QString	**last**(qsizetype *n*) const
qsizetype	**lastIndexOf**(const QString &*str*, qsizetype *from*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**lastIndexOf**(QChar *c*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**lastIndexOf**(QChar *ch*, qsizetype *from*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**lastIndexOf**(QLatin1StringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**lastIndexOf**(QLatin1StringView *str*, qsizetype *from*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**lastIndexOf**(const QString &*str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const

	QString()
qsizetype	**lastIndexOf**(QStringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**lastIndexOf**(QStringView *str*, qsizetype *from*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
qsizetype	**lastIndexOf**(const QRegularExpression &*re*, QRegularExpressionMatch **rmatch* = nullptr) const
qsizetype	**lastIndexOf**(const QRegularExpression &*re*, qsizetype *from*, QRegularExpressionMatch **rmatch* = nullptr) const
QString	**left**(qsizetype *n*) const
QString	**leftJustified**(qsizetype *width*, QChar *fill* = u' ', bool *truncate* = false) const
qsizetype	**length**() const
int	**localeAwareCompare**(const QString &*other*) const
int	**localeAwareCompare**(QStringView *other*) const
QString	**mid**(qsizetype *position*, qsizetype *n* = -1) const
QString	**normalized**(QString::NormalizationForm *mode*, QChar::UnicodeVersion *version* = QChar::Unicode_Unassigned) const
QString &	**prepend**(const QString &*str*)
QString &	**prepend**(QChar *ch*)
QString &	**prepend**(const QChar **str*, qsizetype *len*)
QString &	**prepend**(QStringView *str*)
QString &	**prepend**(QLatin1StringView *str*)
QString &	**prepend**(QUtf8StringView *str*)
QString &	**prepend**(const char **str*)
QString &	**prepend**(const QByteArray &*ba*)
void	**push_back**(const QString &*other*)
void	**push_back**(QChar *ch*)
void	**push_front**(const QString &*other*)
void	**push_front**(QChar *ch*)
QString::reverse_iterator	**rbegin**()
QString::const_reverse_iterator	**rbegin**() const
QString &	**remove**(qsizetype *position*, qsizetype *n*)
QString &	**remove**(QChar *ch*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)

	QString()
QString &	**remove**(QLatin1StringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**remove**(const QString &*str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**remove**(const QRegularExpression &*re*)
QString &	**removeAt**(qsizetype *pos*)
QString &	**removeFirst**()
QString &	**removeIf**(Predicate *pred*)
QString &	**removeLast**()
QString::reverse_iterator	**rend**()
QString::const_reverse_iterator	**rend**() const
QString	**repeated**(qsizetype *times*) const
QString &	**replace**(qsizetype *position*, qsizetype *n*, const QString &*after*)
QString &	**replace**(qsizetype *position*, qsizetype *n*, QChar *after*)
QString &	**replace**(qsizetype *position*, qsizetype *n*, const QChar **unicode*, qsizetype *size*)
QString &	**replace**(QChar *before*, QChar *after*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(const QChar **before*, qsizetype *blen*, const QChar** after*, qsizetype *alen*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(QLatin1StringView *before*, QLatin1StringView *after*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(QLatin1StringView *before*, const QString &*after*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(const QString &*before*, QLatin1StringView *after*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(const QString &*before*, const QString &*after*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(QChar *ch*, const QString &*after*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(QChar *c*, QLatin1StringView *after*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString &	**replace**(const QRegularExpression &*re*, const QString &*after*)
void	**reserve**(qsizetype *size*)
void	**resize**(qsizetype *size*)
void	**resize**(qsizetype *newSize*, QChar *fillChar*)

	QString()
QString	**right**(qsizetype *n*) const
QString	**rightJustified**(qsizetype *width*, QChar *fill* = u' ', bool *truncate* = false) const
QString	**section**(QChar *sep*, qsizetype *start*, qsizetype *end* = -1, QString::SectionFlags *flags* = SectionDefault) const
QString	**section**(const QString &*sep*, qsizetype *start*, qsizetype *end* = -1, QString::SectionFlags *flags* = SectionDefault) const
QString	**section**(const QRegularExpression &*re*, qsizetype *start*, qsizetype *end* = -1, QString::SectionFlags *flags* = SectionDefault) const
QString &	**setNum**(int *n*, int *base* = 10)
QString &	**setNum**(short *n*, int *base* = 10)
QString &	**setNum**(ushort *n*, int *base* = 10)
QString &	**setNum**(uint *n*, int *base* = 10)
QString &	**setNum**(long *n*, int *base* = 10)
QString &	**setNum**(ulong *n*, int *base* = 10)
QString &	**setNum**(qlonglong *n*, int *base* = 10)
QString &	**setNum**(qulonglong *n*, int *base* = 10)
QString &	**setNum**(float *n*, char *format* = 'g', int *precision* = 6)
QString &	**setNum**(double *n*, char *format* = 'g', int *precision* = 6)
QString &	**setRawData**(const QChar **unicode*, qsizetype *size*)
QString &	**setUnicode**(const QChar **unicode*, qsizetype *size*)
QString &	**setUtf16**(const ushort **unicode*, qsizetype *size*)
void	**shrink_to_fit**()
QString	**simplified**() const
qsizetype	**size**() const
QString	**sliced**(qsizetype *pos*, qsizetype *n*) const
QString	**sliced**(qsizetype *pos*) const
QStringList	**split**(const QString &*sep*, Qt::SplitBehavior *behavior* = Qt::KeepEmptyParts, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
QStringList	**split**(QChar *sep*, Qt::SplitBehavior *behavior* = Qt::KeepEmptyParts, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
QStringList	**split**(const QRegularExpression &*re*, Qt::SplitBehavior *behavior* = Qt::KeepEmptyParts) const

	QString()
void	**squeeze**()
bool	**startsWith**(const QString &*s*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**startsWith**(QStringView *str*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**startsWith**(QLatin1StringView *s*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
bool	**startsWith**(QChar *c*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive) const
void	**swap**(QString &*other*)
CFStringRef	**toCFString**() const
QString	**toCaseFolded**() const
double	**toDouble**(bool **ok* = nullptr) const
float	**toFloat**(bool **ok* = nullptr) const
QString	**toHtmlEscaped**() const
int	**toInt**(bool **ok* = nullptr, int *base* = 10) const
QByteArray	**toLatin1**() const
QByteArray	**toLocal8Bit**() const
long	**toLong**(bool **ok* = nullptr, int *base* = 10) const
qlonglong	**toLongLong**(bool **ok* = nullptr, int *base* = 10) const
QString	**toLower**() const
NSString *	**toNSString**() const
short	**toShort**(bool **ok* = nullptr, int *base* = 10) const
std::string	**toStdString**() const
std::u16string	**toStdU16String**() const
std::u32string	**toStdU32String**() const
std::wstring	**toStdWString**() const
uint	**toUInt**(bool **ok* = nullptr, int *base* = 10) const
ulong	**toULong**(bool **ok* = nullptr, int *base* = 10) const
qulonglong	**toULongLong**(bool **ok* = nullptr, int *base* = 10) const
ushort	**toUShort**(bool **ok* = nullptr, int *base* = 10) const
QList	**toUcs4**() const

	QString()
QString	**toUpper**() const
QByteArray	**toUtf8**() const
qsizetype	**toWCharArray**(wchar_t **array*) const
decltype(qTokenize(*this, std::forward(needle), flags...))	**tokenize**(Needle &&*sep*, Flags... *flags*) const &
decltype(qTokenize(std::move(*this), std::forward(needle), flags...))	**tokenize**(Needle &&*sep*, Flags... *flags*) const &&
decltype(qTokenize(std::move(*this), std::forward(needle), flags...))	**tokenize**(Needle &&*sep*, Flags... *flags*) &&
QString	**trimmed**() const
void	**truncate**(qsizetype *position*)
const QChar *	**unicode**() const
const ushort *	**utf16**() const
bool	**operator!=**(const char **other*) const
bool	**operator!=**(const QByteArray &*other*) const
QString &	**operator+=**(const QString &*other*)
QString &	**operator+=**(QChar *ch*)
QString &	**operator+=**(QStringView *str*)
QString &	**operator+=**(QLatin1StringView *str*)
QString &	**operator+=**(QUtf8StringView *str*)
QString &	**operator+=**(const char **str*)
QString &	**operator+=**(const QByteArray &*ba*)
bool	**operator<**(const char **other*) const
bool	**operator<**(const QByteArray &*other*) const
bool	**operator<=**(const char **other*) const
bool	**operator<=**(const QByteArray &*other*) const
QString &	**operator=**(const QString &*other*)
QString &	**operator=**(QChar *ch*)
QString &	**operator=**(QLatin1StringView *str*)
QString &	**operator=**(QString &&*other*)
QString &	**operator=**(const char **str*)

	QString()
QString &	**operator=**(const QByteArray &*ba*)
bool	**operator==**(const char **other*) const
bool	**operator==**(const QByteArray &*other*) const
bool	**operator>**(const char **other*) const
bool	**operator>**(const QByteArray &*other*) const
bool	**operator>=**(const char **other*) const
bool	**operator>=**(const QByteArray &*other*) const
QChar &	**[operator]**(qsizetype *position*)
const QChar	**[operator]**(qsizetype *position*) const

静态公共成员

QString	**asprintf**(**const char** ***cformat*, ...**)
int	**compare**(const QString &*s1*, const QString &*s2*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
int	**compare**(const QString &*s1*, QLatin1StringView *s2*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
int	**compare**(QLatin1StringView *s1*, const QString &*s2*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
int	**compare**(const QString &*s1*, QStringView *s2*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
int	**compare**(QStringView *s1*, const QString &*s2*, Qt::CaseSensitivity *cs* = Qt::CaseSensitive)
QString	**fromCFString**(CFStringRef *string*)
QString	**fromLatin1**(const char **str*, qsizetype *size*)
QString	**fromLatin1**(QByteArrayView *str*)
QString	**fromLatin1**(const QByteArray &*str*)
QString	**fromLocal8Bit**(const char **str*, qsizetype *size*)
QString	**fromLocal8Bit**(QByteArrayView *str*)
QString	**fromLocal8Bit**(const QByteArray &*str*)
QString	**fromNSString**(const NSString **string*)
QString	**fromRawData**(const QChar **unicode*, qsizetype *size*)
QString	**fromStdString**(const std::string &*str*)
QString	**fromStdU16String**(const std::u16string &*str*)

QString	**asprintf(const char *_cformat_, ...)**
QString	**fromStdU32String**(const std::u32string &_str_)
QString	**fromStdWString**(const std::wstring &_str_)
QString	**fromUcs4**(const char32_t *_unicode_, qsizetype _size_ = -1)
QString	**fromUtf8**(const char *_str_, qsizetype _size_)
QString	**fromUtf8**(QByteArrayView _str_)
QString	**fromUtf8**(const QByteArray &_str_)
QString	**fromUtf8**(const char8_t *_str_)
QString	**fromUtf8**(const char8_t *_str_, qsizetype _size_)
QString	**fromUtf16**(const char16_t *_unicode_, qsizetype _size_ = -1)
QString	**fromWCharArray**(const wchar_t *_string_, qsizetype _size_ = -1)
int	**localeAwareCompare**(const QString &_s1_, const QString &_s2_)
int	**localeAwareCompare**(QStringView _s1_, QStringView _s2_)
QString	**number**(long _n_, int _base_ = 10)
QString	**number**(int _n_, int _base_ = 10)
QString	**number**(uint _n_, int _base_ = 10)
QString	**number**(ulong _n_, int _base_ = 10)
QString	**number**(qlonglong _n_, int _base_ = 10)
QString	**number**(qulonglong _n_, int _base_ = 10)
QString	**number**(double _n_, char _format_ = 'g', int _precision_ = 6)
QString	**vasprintf**(const char *_cformat_, va_list _ap_)

相关非会员

qsizetype	**erase(QString &_s_, const T &_t_)**
qsizetype	**erase_if**(QString &_s_, Predicate _pred_)
bool	**operator!=**(const QString &_s1_, const QString &_s2_)
bool	**operator!=**(const QString &_s1_, QLatin1StringView _s2_)
bool	**operator!=**(const char *_s1_, const QString &_s2_)
QString	**operator""_s**(const char16_t *_str_, size_t _size_)
QString	**operator+**(const QString &_s1_, const QString &_s2_)

qsizetype	**erase**(QString &*s*, const T &*t*)
QString	**operator+**(QString &&*s1*, const QString &*s2*)
QString	**operator+**(const QString &*s1*, const char **s2*)
QString	**operator+**(const char **s1*, const QString &*s2*)
bool	**operator<**(const QString &*s1*, const QString &*s2*)
bool	**operator<**(const QString &*s1*, QLatin1StringView *s2*)
bool	**operator<**(QLatin1StringView *s1*, const QString &*s2*)
bool	**operator<**(const char **s1*, const QString &*s2*)
QDataStream &	**operator<<**(QDataStream &*stream*, const QString &*string*)
bool	**operator<=**(const QString &*s1*, const QString &*s2*)
bool	**operator<=**(const QString &*s1*, QLatin1StringView *s2*)
bool	**operator<=**(QLatin1StringView *s1*, const QString &*s2*)
bool	**operator<=**(const char **s1*, const QString &*s2*)
bool	**operator==**(const QString &*s1*, const QString &*s2*)
bool	**operator==**(const QString &*s1*, QLatin1StringView *s2*)
bool	**operator==**(QLatin1StringView *s1*, const QString &*s2*)
bool	**operator==**(const char **s1*, const QString &*s2*)
bool	**operator>**(const QString &*s1*, const QString &*s2*)
bool	**operator>**(const QString &*s1*, QLatin1StringView *s2*)
bool	**operator>**(QLatin1StringView *s1*, const QString &*s2*)
bool	**operator>**(const char **s1*, const QString &*s2*)
bool	**operator>=**(const QString &*s1*, const QString &*s2*)
bool	**operator>=**(const QString &*s1*, QLatin1StringView *s2*)
bool	**operator>=**(QLatin1StringView *s1*, const QString &*s2*)
bool	**operator>=**(const char **s1*, const QString &*s2*)
QDataStream &	**operator>>**(QDataStream &*stream*, QString &*string*)

宏

QStringLiteral(***str***)

	QStringLiteral(*str*)
	QT_NO_CAST_FROM_ASCII
	QT_NO_CAST_TO_ASCII
	QT_RESTRICTED_CAST_FROM_ASCII
const char *	**qPrintable**(const QString &*str*)
const wchar_t *	**qUtf16Printable**(const QString &*str*)
const char *	**qUtf8Printable**(const QString &*str*)

详细说明

QString存储一个16位的字符串QChars，其中每个QChar对应一个UTF-16编码单元。（代码值大于 65535 的 Unicode 字符使用代理对存储，即两个连续的QChars.）

Unicode是支持当今使用的大多数书写系统的国际标准。它是 US-ASCII (ANSI X3.4-1986) 和 Latin-1 (ISO 8859-1) 的超集，并且所有 US-ASCII/Latin-1 字符都可以在相同的代码位置使用。

QString 在幕后使用implicit sharing（写时复制）以减少内存使用并避免不必要的数据复制。这也有助于减少存储 16 位字符而不是 8 位字符的固有开销。

除了QString之外，Qt还提供了QByteArray类来存储原始字节和传统的 8 位“\0”终止字符串。对于大多数用途，QString 是您要使用的类。它在整个 Qt API 中使用，并且如果您想在某个时候扩展应用程序的市场，Unicode 支持可确保您的应用程序易于翻译。两种主要情况QByteArray当您需要存储原始二进制数据以及内存保护至关重要时（例如在嵌入式系统中），比较合适。

初始化一个字符串

初始化 QString 的一种方法是将 a 传递`const char *`给其构造函数。例如，以下代码创建一个大小为 5 的 QString，其中包含数据“Hello”：

```
QString str = "Hello";
```

QString`const char *`使用以下命令将数据转换为 UnicodefromUtf8（） 功能。

在所有采用`const char *`参数的 QString 函数中，`const char *`被解释为以 UTF-8 编码的经典 C 风格“\0”结尾的字符串。`const char *`参数为 是合法的`nullptr`。

您还可以提供字符串数据作为数组QChars:

```
static const QChar data[4] = { 0x0055, 0x006e, 0x10e3, 0x03a3 };
QString str(data, 4);
```

QString 进行深度复制QChar数据，以便您以后可以对其进行修改而不会产生副作用。（如果出于性能原因您不想对字符数据进行深层复制，请使用QString::fromRawData（） 反而。）

另一种方法是使用设置字符串的大小resize() 并初始化每个字符的数据字符。QString 使用从 0 开始的索引，就像 C++ 数组一样。要访问特定索引位置处的字符，您可以使用[operator]()。在非const弦上，[operator]() 返回对可在赋值左侧使用的字符的引用。例如：

```
QString str;
str.resize(4);

str[0] = QChar('U');
str[1] = QChar('n');
str[2] = QChar(0x10e3);
str[3] = QChar(0x03a3);
```

对于只读访问，另一种语法是使用at（） 功能：

```
QString str;

for (qsizetype i = 0; i < str.size(); ++i) {
    if (str.at(i) >= QChar('a') && str.at(i) <= QChar('f'))
        qDebug() << "Found character in range [a-f]";
}
```

这at() 函数可以比[operator]()，因为它永远不会导致deep copy发生。或者，使用first(),last（），或者sliced() 函数一次提取多个字符。

QString 可以嵌入 '\0' 字符（QChar::Null）。这size() 函数始终返回整个字符串的大小，包括嵌入的 '\0' 字符。

拨打电话后resize() 函数中，新分配的字符具有未定义的值。要将字符串中的所有字符设置为特定值，请使用fill（） 功能。

QString 提供了数十种旨在简化字符串使用的重载。例如，如果您想将 QString 与字符串文字进行比较，您可以编写如下代码，它将按预期工作：

```
QString str;

if (str == "auto" || str == "extern"
        || str == "static" || str == "register") {
    // ...
}
```

您还可以将字符串文字传递给以 QString 作为参数的函数，调用 QString(const char *) 构造函数。const char *类似地，您可以使用QString 将 QString 传递给采用参数的函数qPrintable() 宏，它将给定的 QString 返回为const char *. 这相当于调用。toLocal8Bit()。constData()。

操作字符串数据

QString提供了以下修改字符数据的基本函数：append(),prepend(),insert(),replace（），和remove()。例如：

```
QString str = "and";
str.prepend("rock ");     // str == "rock and"
str.append(" roll");        // str == "rock and roll"
str.replace(5, 3, "&");   // str == "rock & roll"
```

在上面的例子中replace() 函数的前两个参数是开始替换的位置和应替换的字符数。

当数据修改函数增加字符串的大小时，可能会导致 QString 对象重新分配内存。当这种情况发生时，QString 的扩展量会超过它立即需要的量，以便有空间用于进一步扩展而无需重新分配，直到字符串的大小大大增加。

这insert(),remove() 并且，当用不同大小的子字符串替换时，replace() 函数可能会很慢 (linear time）对于大字符串，因为它们需要将字符串中的许多字符在内存中至少移动一个位置。

如果您正在逐步构建 QString 并且提前知道 QString 将包含多少个字符，您可以调用reserve()，要求QString预先分配一定量的内存。您也可以致电capacity() 来找出 QString 实际分配了多少内存。

QString提供STL-style iterators （QString::const_iterator和QString::iterator）。实际上，在使用 C++ 标准库提供的通用算法时，迭代器非常方便。

注意：`const`当调用 QString 的任何非方法时，不能依赖 QString 上的迭代器以及对其中单个字符的引用保持有效。在调用非`const`方法之后访问此类迭代器或引用会导致未定义的行为。当需要类似迭代器的功能的稳定性时，您应该使用索引而不是迭代器，因为它们不依赖于 QString 的内部状态，因此不会失效。

注：由于implicit sharing`const`，给定 QString 上使用的第一个非运算符或函数可能会导致它在内部执行其数据的深层复制。这将使字符串上的所有迭代器以及对其中单个字符的引用无效。在第一个非`const`运算符之后，修改 QString 的操作可能会完全（在重新分配的情况下）或部分使迭代器和引用无效，但其他方法（例如begin （） 或者end （） ） 将不会。在迭代器或引用失效后访问它会导致未定义的行为。

一个常见的要求是从字符串中删除空格字符（'\n'、'\t'、' ' 等）。如果要删除 QString 两端的空格，请使用trimmed （） 功能。如果要删除字符串两端的空格并将多个连续的空格替换为字符串中的单个空格字符，请使用simplified()。

如果要查找 QString 中特定字符或子字符串的所有出现位置，请使用indexOf （） 或者lastIndexOf （） 功能。前者从给定的索引位置开始向前搜索，后者向后搜索。如果找到的话，两者都会返回字符或子字符串的索引位置；否则，它们返回-1。例如，下面是一个典型的循环，用于查找特定子字符串的所有出现位置：

```
QString str = "We must be <b>bold</b>, very <b>bold</b>";
qsizetype j = 0;

while ((j = str.indexOf("<b>", j)) != -1) {
    qDebug() << "Found <b> tag at index position" << j;
    ++j;
}
```

QString 提供了许多将数字转换为字符串以及字符串转换为数字的函数。请参阅arg() 函数， setNum() 函数，number() 静态函数，以及toInt(),toDouble() 以及类似的函数。

要获取字符串的大写或小写版本，请使用toUpper （） 或者toLower()。

字符串列表由QStringList班级。您可以使用以下命令将字符串拆分为字符串列表split() 函数，并使用可选分隔符将字符串列表连接成单个字符串QStringList::join()。您可以从字符串列表中获取包含特定子字符串或与特定子字符串匹配的字符串列表QRegularExpression使用QStringList::filter （） 功能。

查询字符串数据

如果你想查看 QString 是否以特定子字符串开头或结尾，请使用startsWith（） 或者endsWith()。如果您只想检查 QString 是否包含特定字符或子字符串，请使用contains（） 功能。如果您想找出特定字符或子字符串在字符串中出现的次数，请使用 count()。

要获取指向实际字符数据的指针，请调用data（） 或者constData()。这些函数返回一个指向开头的指针QChar数据。const在 QString 上调用非函数之前，指针保证保持有效。

比较字符串

可以使用重载运算符来比较 QString，例如operator<(),operator<=(),operator==(),operator>=（），等等。请注意，比较仅基于字符的 Unicode 数字值。它非常快，但不是人类所期望的；这QString::localeAwareCompare当可以进行此类比较时，() 函数通常是对用户界面字符串进行排序的更好选择。

在类 Unix 平台（包括 Linux、macOS 和 iOS）上，当 Qt 与 ICU 库（通常是这样）链接时，会使用其区域设置感知排序。否则，在 macOS 和 iOS 上，localeAwareCompare() 根据国际首选项面板中的“排序列表的顺序”设置进行比较。在其他没有 ICU 的类 Unix 系统上，比较会回溯到系统库的strcoll(),

编码字符串数据和 QString 之间的转换

QString 提供了以下三个函数，它们返回const char *字符串的版本：QByteArray:toUtf8(),toLatin1（），和toLocal8Bit()。

- toLatin1() 返回 Latin-1 (ISO 8859-1) 编码的 8 位字符串。
- toUtf8() 返回一个 UTF-8 编码的 8 位字符串。UTF-8 是 US-ASCII (ANSI X3.4-1986) 的超集，它通过多字节序列支持整个 Unicode 字符集。
- toLocal8Bit() 使用系统本地编码返回 8 位字符串。这与toUtf8() 在 Unix 系统上。

为了从这些编码之一进行转换，QString 提供了fromLatin1(),fromUtf8（），和fromLocal8Bit()。通过以下方式支持其他编码QStringEncoder和QStringDecoder类。

如上所述，QString 提供了很多函数和运算符，可以轻松地与const char *字符串进行互操作。但此功能是一把双刃剑：如果所有字符串都是 US-ASCII 或 Latin-1，它会使 QString 使用起来更方便，但始终存在使用错误的 8 位进行隐式转换的const char *风险编码。为了最大限度地降低这些风险，您可以通过定义以下一些预处理器符号来关闭这些隐式转换：

- QT_NO_CAST_FROM_ASCII禁用从 C 字符串文字和指针到 Unicode 的自动转换。
- QT_RESTRICTED_CAST_FROM_ASCII允许从 C 字符和字符数组自动转换，但禁用从字符指针到 Unicode 的自动转换。
- QT_NO_CAST_TO_ASCII禁用从 QString 到 C 字符串的自动转换。

然后你需要显式调用fromUtf8(),fromLatin1（），或者fromLocal8Bit() 从 8 位字符串构造 QString，或使用轻量级QLatin1StringView类，例如：

```
QString url = "https://www.unicode.org/"_L1;
```

同样，您必须调用toLatin1(),toUtf8（），或者toLocal8Bit() 显式地将 QString 转换为 8 位字符串。

C 程序员注意事项

由于 C++ 的类型系统和 QString 的事实implicitly shared、QStrings 可以像 s 或其他基本类型一样对待int。例如：QString Widget::boolToString(bool b) { QString result; if (b) result = "True"; else result = "False"; return result; }该result变量是分配在堆栈上的普通变量。Whenreturn被调用，并且因为我们按值返回，所以调用复制构造函数并返回字符串的副本。由于隐式共享，不会发生实际的复制。

空字符串和空字符串之间的区别

由于历史原因，QString 区分空字符串和空字符串。*空字符串*是使用 QString 的默认构造函数或通过将 (const char *)0 传递给构造函数来初始化的字符串。*空字符串*是任何大小为 0 的字符串。空字符串始终为空，但空字符串不一定为空：

```
QString().isNull();               // returns true
QString().isEmpty();              // returns true

QString("").isNull();             // returns false
QString("").isEmpty();            // returns true

QString("abc").isNull();          // returns false
QString("abc").isEmpty();         // returns false
```

所有功能，除了isNull() 将空字符串视为与空字符串相同。例如，toUtf8()。constData() 返回一个指向空字符串的“\0”字符的有效指针（*不是nullptr*）。我们建议您始终使用isEmpty()函数及避免isNull()。

数字格式

当一个QString::arg()'%'格式说明符包含'L'区域设置限定符，基数为 10（默认值），使用默认区域设置。这可以设置使用QLocale::setDefault()。要更精细地控制数字的本地化字符串表示形式，请参阅QLocale::toString()。QString 完成的所有其他数字格式均遵循 C 语言环境的数字表示形式。

什么时候QString::arg() 对数字应用左填充，填充字符'0'会被特殊处理。如果数字为负数，则其减号将出现在补零之前。如果该字段已本地化，则使用适合区域设置的零字符来代替'0'。对于浮点数，这种特殊处理仅适用于数字有限的情况。

浮点格式

在成员函数中（例如，arg(),number()) 将浮点数 (float或double) 表示为字符串，显示形式可以通过选择*格式*和*精度*来控制，其含义如下QLocale::toString（双精度、字符、整数）。

如果所选*格式*包括指数，则本地化形式将遵循区域设置对指数中数字的约定。对于非本地化格式，指数显示其符号并包含至少两位数字，如果需要，则在左侧填充零。

更高效的字符串构建

许多字符串在编译时是已知的。但是简单的构造函数 QString("Hello") 将复制字符串的内容，并将内容视为 Latin-1。为了避免这种情况，可以使用QStringLiteral宏在编译时直接创建所需的数据。从文字中构造 QString 不会在运行时造成任何开销。

效率稍低的方法是使用QLatin1StringView。此类包装 C 字符串文字，在编译时预先计算其长度，然后可用于比常规 C 字符串文字更快地与 QString 进行比较以及转换为 QString。

使用 QString'+'运算符，可以轻松地从多个子字符串构造复杂的字符串。你经常会写这样的代码：

```
    QString foo;
    QString type = "long";

    foo = "vector<"_L1 + type + ">::iterator"_L1;

    if (foo.startsWith("(" + type + ") 0x"))
        ...
```

这些字符串结构都没有任何问题，但存在一些隐藏的低效率问题。从 Qt 4.6 开始，您可以消除它们。

首先，操作符的多次使用'+'通常意味着多次内存分配。当连接*n*个子字符串时，其中*n* > *2*，可以有多达*n* - *1*次对内存分配器的调用。

`QStringBuilder`在 4.6 中，添加了一个内部模板类以及一些辅助函数。此类被标记为内部类，并且不会出现在文档中，因为您不打算在代码中实例化它。如下所述，其使用将是自动的。`src/corelib/tools/qstringbuilder.cpp`如果您想查看该类，可以在 中找到它。

`QStringBuilder`使用表达式模板并重新实现`'%'`运算符，以便当您使用`'%'` for 字符串连接而不是 时`'+'`，多个子字符串连接将被推迟，直到最终结果即将分配给 QString。至此，最终结果所需的内存量就已知了。然后调用一次内存分配器来获取所需的空间，并将子字符串一一复制到其中。

通过内联和减少引用计数可以获得额外的效率（从 a 创建的 QString`QStringBuilder`通常具有 1 的引用计数，而 QString::append() 需要额外测试）。

您可以通过两种方式访问这种改进的字符串构造方法。`QStringBuilder`最简单的方法是在您想要使用它的任何地方包含它，并在连接字符串时使用`'%'`运算符而不是`'+'`：

```
    #include <QStringBuilder>

    QString hello("hello");
    QStringView el = QStringView{ hello }.mid(2, 3);
    QLatin1StringView world("world");
    QString message =  hello % el % world % QChar('!');
```

一种更全局的方法是最方便但不完全源兼容的，是在 .pro 文件中定义：

并且`'+'`将自动执行为`QStringBuilder '%'`无处不在。

最大大小和内存不足情况

QString 的最大大小取决于体系结构。大多数 64 位系统可以分配超过 2 GB 的内存，典型限制为 2^63 字节。实际值还取决于管理数据块所需的开销。因此，在 32 位平台上，预计最大大小为 2 GB 减去开销，在 64 位平台上，最大大小为 2^63 字节减去开销。QString 中可以存储的元素数量是该最大大小除以QChar。

`std::bad_alloc`当内存分配失败时，如果应用程序是使用异常支持编译的，则QString 会引发异常。Qt 容器中内存不足的情况是 Qt 抛出异常的唯一情况。如果禁用异常，则内存不足是未定义的行为。

请注意，操作系统可能会对持有大量已分配内存（尤其是大的连续块）的应用程序施加进一步的限制。此类注意事项、此类行为的配置或任何缓解措施均超出了 Qt API 的范围。

也可以看看fromRawData(),QChar,QStringView,QLatin1StringView， 和QByteArray。

会员类型文档

QString::ConstIterator

Qt 风格的同义词QString::const_iterator。

QString::Iterator

Qt 风格的同义词QString::iterator。

enum QString::NormalizationForm

该枚举描述了 Unicode 文本的各种标准化形式。

持续的	价值	描述
`QString::NormalizationForm_D`	`0`	规范分解
`QString::NormalizationForm_C`	`1`	规范分解后进行规范组合
`QString::NormalizationForm_KD`	`2`	兼容性分解
`QString::NormalizationForm_KC`	`3`	兼容性分解，然后是规范组合

也可以看看normalized（） 和Unicode Standard Annex #15。

枚举 QString:: SectionFlag 标志 QString:: SectionFlags

该枚举指定可用于影响各个方面的标志section() 函数关于分隔符和空字段的行为。

持续的	价值	描述
`QString::SectionDefault`	`0x00`	计算空字段，不包括前导和尾随分隔符，并且区分大小写比较分隔符。
`QString::SectionSkipEmpty`	`0x01`	将空字段视为不存在，即就*开始*和*结束*而言，不考虑它们。
`QString::SectionIncludeLeadingSep`	`0x02`	在结果字符串中包含前导分隔符（如果有）。
`QString::SectionIncludeTrailingSep`	`0x04`	在结果字符串中包含尾随分隔符（如果有）。
`QString::SectionCaseInsensitiveSeps`	`0x08`	比较分隔符时不区分大小写。

SectionFlags 类型是QFlags 的类型定义。它存储SectionFlag 值的OR 组合。

也可以看看section()。

QString::const_iterator

也可以看看QString::iterator。

QString::const_pointer

QString::const_pointer typedef 提供了一个 STL 风格的 const 指针，指向一个QString元素 （QChar）。

QString::const_reference

QString::const_reverse_iterator

也可以看看QString::reverse_iterator和QString::const_iterator。

QString::difference_type

QString::iterator

也可以看看QString::const_iterator。

QString::pointer

QString::pointer typedef 提供了一个 STL 风格的指针，指向QString元素 （QChar）。

QString::reference

QString::reverse_iterator

也可以看看QString::const_reverse_iterator和QString::iterator。

QString::size_type

QString::value_type

成员函数文档

template QString QString::arg(Args &&... args) const

将此字符串中出现的 替换%N为相应的参数*args*。参数不是位置性的：第一个*args*将 替换%N为最低的N（全部），即第二个*args*与%N次N低等

Args可以由隐式转换为的任何内容组成QString,QStringView或者QLatin1StringView。

此外，还支持以下类型：QChar,QLatin1Char。

也可以看看QString::arg()。

[since 6.0]template <typename Needle, typename Flags>
decltype(qTokenize(std::move(this), std::forward(needle), flags...))
QString::tokenize(Needle &&sep, Flags... flags) &&*

[since 6.0]template <typename Needle, typename Flags>
decltype(qTokenize(this, std::forward(needle), flags...))
QString::tokenize(Needle &&sep, Flags... flags) const &*

[since 6.0]template <typename Needle, typename Flags>
decltype(qTokenize(std::move(this), std::forward(needle), flags...))
QString::tokenize(Needle &&sep, Flags... flags) const &&*

将字符串拆分为子字符串视图*sep*发生，并返回这些字符串的惰性序列。

相当于

```
return QStringTokenizer{std::forward<Needle>(sep), flags...};
```

除非它在编译器中未启用 C++17 类模板参数推导 (CTAD) 的情况下工作。

看QStringTokenizer如何*sep*和*flags*相互作用形成结果。

注意：虽然该函数返回QStringTokenizer，你永远不应该显式地命名它的模板参数。如果你可以使用C++17类模板参数推导（CTAD），你可以写

```
QStringTokenizer result = sv.tokenize(sep);
```

（没有模板参数）。如果无法使用 C++17 CTAD，则必须仅将返回值存储在auto变量中：

```
auto result = sv.tokenize(sep);
```

这是因为模板参数QStringTokenizer对特定的有非常微妙的依赖tokenize() 重载从中返回它们，并且它们通常与用于分隔符的类型不对应。

这个函数是在Qt 6.0中引入的。

也可以看看QStringTokenizer和qTokenize()。

[constexpr]QString::QString()

构造一个空字符串。空字符串也被视为空。

也可以看看isEmpty(),isNull（），和Distinction Between Null and Empty Strings。

[explicit]QString::QString(const QChar *unicode, qsizetype size = -1)

构造一个用第一个初始化的字符串*size*的字符QChar大批*unicode*。

如果*unicode*为 0，则构造空字符串。

如果*size*是负数，*unicode*假定指向一个以 \0' 结尾的数组，并且其长度是动态确定的。终止空字符不被视为字符串的一部分。

QString 对字符串数据进行深层复制。unicode 数据按原样复制，并保留字节顺序标记（如果存在）。

也可以看看fromRawData()。

QString::QString(QChar ch)

构造一个包含字符的大小为 1 的字符串*ch*。

QString::QString(qsizetype size, QChar ch)

构造给定的字符串*size*每个字符设置为*ch*。

也可以看看fill()。

QString::QString(QLatin1StringView str)

构造通过查看的 Latin-1 字符串的副本*str*。

也可以看看fromLatin1()。

[since 6.1]QString::QString(const char8_t *str)

构造一个使用 UTF-8 字符串初始化的字符串*str*。使用以下命令将给定的 const char8_t 指针转换为 UnicodefromUtf8（） 功能。

该功能是在 Qt 6.1 中引入的。

也可以看看fromLatin1(),fromLocal8Bit（），和fromUtf8()。

*QString::QString(const char *str)*

构造一个用 8 位字符串初始化的字符串*str*。使用以下命令将给定的 const char 指针转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用此构造函数QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

注：定义QT_RESTRICTED_CAST_FROM_ASCII也禁用此构造函数，但启用`QString(const char (&ch)[N])`构造函数。在这种情况下，使用非文字输入、嵌入 NUL 字符或非 7 位字符的输入是未定义的。

也可以看看fromLatin1(),fromLocal8Bit（），和fromUtf8()。

QString::QString(const QByteArray &ba)

构造一个用字节数组初始化的字符串*ba*。使用给定的字节数组转换为 UnicodefromUtf8()。

您可以通过定义禁用此构造函数QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

注意：：字节数组中的任何空（'\0'）字节都将包含在此字符串中，并转换为 Unicode 空字符 (U+0000)。此行为与 Qt 5.x 不同。

也可以看看fromLatin1(),fromLocal8Bit（），和fromUtf8()。

QString::QString(const QString &other)

构造一个副本*other*。

此操作需要constant time，因为 QString 是implicitly shared。这使得从函数返回 QString 的速度非常快。如果共享实例被修改，它将被复制（写时复制），这需要linear time。

也可以看看operator=()。

QString::QString(QString &&other)

Move 构造一个 QString 实例，使其指向与*other*正在指着。

QString::~QString()

破坏字符串。

QString &QString::append(const QString &str)

追加字符串*str*到该字符串的末尾。

例子：

```
QString x = "free";
QString y = "dom";

x.append(y);
// x == "freedom"
```

这与使用相同insert（） 功能：

```
x.insert(x.size(), y);
```

append() 函数通常非常快（constant time）， 因为QString在字符串数据的末尾预先分配额外的空间，以便它可以增长，而无需每次重新分配整个字符串。

也可以看看operator+=(),prepend（） ， 和insert()。

QString &QString::append(QChar ch)

该函数重载了append()。

追加字符*ch*到这个字符串。

*QString &QString::append(const QChar *str, qsizetype len)*

该函数重载了append()。

追加*len*字符来自QChar大批*str*到这个字符串。

[since 6.0]QString &QString::append(QStringView v)

该函数重载了append()。

附加给定的字符串视图*v*到这个字符串并返回结果。

这个函数是在Qt 6.0中引入的。

QString &QString::append(QLatin1StringView str)

该函数重载了append()。

附加通过查看的 Latin-1 字符串*str*到这个字符串。

[since 6.5]QString &QString::append(QUtf8StringView str)

该函数重载了append()。

追加 UTF-8 字符串视图*str*到这个字符串。

该功能是在 Qt 6.5 中引入的。

*QString &QString::append(const char *str)*

该函数重载了append()。

追加字符串*str*到这个字符串。使用以下命令将给定的 const char 指针转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用此功能QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QString &QString::append(const QByteArray &ba)

该函数重载了append()。

追加字节数组*ba*到这个字符串。使用以下命令将给定的字节数组转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用此功能QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QString QString::arg(const QString &a, int fieldWidth = 0, QChar fillChar = u' ') const

返回此字符串的副本，其中编号最小的位置标记被替换为字符串*a*，即%1，，，%2...，%99。

*fieldWidth*指定参数的最小空间量*a*应占据。如果*a*需要的空间小于*fieldWidth*，它被填充为*fieldWidth*有性格*fillChar*。积极的*fieldWidth*生成右对齐文本。负数*fieldWidth*生成左对齐文本。

此示例展示了我们如何status在处理文件列表时创建一个用于报告进度的字符串：

```
QString i;           // current file's number
QString total;       // number of files to process
QString fileName;    // current file's name

QString status = QString("Processing file %1 of %2: %3")
                 .arg(i).arg(total).arg(fileName);
```

首先，arg(i)替换%1. 然后arg(total)替换%2. 最后，arg(fileName)替换%3.

使用 arg() 的优点之一是asprintf() 的缺点是，如果应用程序的字符串被翻译成其他语言，则编号位置标记的顺序可以更改，但每个 arg() 仍将替换编号最低的未替换位置标记，无论它出现在何处。此外，如果位置标记%i在字符串中出现多次，则 arg() 会替换所有标记。

如果没有剩余未替换的位置标记，则会输出警告消息并且结果未定义。位置标记编号必须在 1 到 99 范围内。

QString QString::arg(qlonglong a, int fieldWidth = 0, int base = 10, QChar fillChar = u' ') const

该函数重载了 arg()。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

这*base*参数指定转换整数时使用的基数*a*成字符串。基数必须在 2 到 36 之间，其中 8 表示八进制数，10 表示十进制数，16 表示十六进制数。

也可以看看Number Formats。

QString QString::arg(qulonglong a, int fieldWidth = 0, int base = 10, QChar fillChar = u' ') const

该函数重载了 arg()。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

这*base*参数指定转换整数时使用的基数*a*成字符串。*base*必须介于 2 和 36 之间，其中 8 表示八进制数，10 表示十进制数，16 表示十六进制数。

也可以看看Number Formats。

QString QString::arg(long a, int fieldWidth = 0, int base = 10, *QChar* fillChar = u' ') const

该函数重载了 arg()。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

这*a*参数表达为给定的*base*，默认为 10，并且必须介于 2 到 36 之间。

“%”后面可以跟一个“L”，在这种情况下，该序列将替换为本地化表示*a*。转换使用默认区域设置。默认区域设置由应用程序启动时系统的区域设置确定。可以使用它来更改QLocale::setDefault()。'L' 标志被忽略，如果*base*不是 10。

```
QString str;
str = QString("Decimal 63 is %1 in hexadecimal")
        .arg(63, 0, 16);
// str == "Decimal 63 is 3f in hexadecimal"

QLocale::setDefault(QLocale(QLocale::English, QLocale::UnitedStates));
str = QString("%1 %L2 %L3")
        .arg(12345)
        .arg(12345)
        .arg(12345, 0, 16);
// str == "12345 12,345 3039"
```

也可以看看Number Formats。

QString QString::arg(*ulong* a, int fieldWidth = 0, int base = 10, *QChar* fillChar = u' ') const

该函数重载了 arg()。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

这*base*参数指定转换整数时使用的基数*a*到一个字符串。基数必须在 2 到 36 之间，其中 8 表示八进制数，10 表示十进制数，16 表示十六进制数。

也可以看看Number Formats。

QString QString::arg(int a, int fieldWidth = 0, int base = 10, *QChar* fillChar = u' ') const

该函数重载了 arg()。

这*a*参数以基数表示*base*，默认为 10，并且必须介于 2 到 36 之间。对于 10 以外的基数，*a*被视为无符号整数。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

“%”后面可以跟一个“L”，在这种情况下，该序列将替换为本地化表示*a*。转换使用默认区域设置，设置为QLocale::setDefault()。如果未指定默认区域设置，则使用系统区域设置。'L' 标志被忽略，如果*base*不是 10。

```
QString str;
str = QString("Decimal 63 is %1 in hexadecimal")
        .arg(63, 0, 16);
// str == "Decimal 63 is 3f in hexadecimal"

QLocale::setDefault(QLocale(QLocale::English, QLocale::UnitedStates));
str = QString("%1 %L2 %L3")
        .arg(12345)
        .arg(12345)
        .arg(12345, 0, 16);
// str == "12345 12,345 3039"
```

也可以看看Number Formats。

QString QString::arg(uint a, int fieldWidth = 0, int base = 10, QChar fillChar = u' ') const

该函数重载了 arg()。

这*base*参数指定转换整数时使用的基数*a*成字符串。底数必须介于 2 到 36 之间。

也可以看看Number Formats。

QString QString::arg(short a, int fieldWidth = 0, int base = 10, QChar fillChar = u' ') const

该函数重载了 arg()。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

这*base*参数指定转换整数时使用的基数*a*成字符串。基数必须在 2 到 36 之间，其中 8 表示八进制数，10 表示十进制数，16 表示十六进制数。

也可以看看Number Formats。

QString QString::arg(ushort a, int fieldWidth = 0, int base = 10, QChar fillChar = u' ') const

该函数重载了 arg()。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

这*base*参数指定转换整数时使用的基数*a*成字符串。基数必须在 2 到 36 之间，其中 8 表示八进制数，10 表示十进制数，16 表示十六进制数。

也可以看看Number Formats。

QString QString::arg(double *a*, int *fieldWidth* = 0, char *format* = 'g', int *precision* = -1, *QChar* *fillChar* = u' ') const

该函数重载了 arg()。

争论*a*按照指定的格式进行格式化*format*和*precision*。看Floating-point Formats了解详情。

*fieldWidth*指定最小空间量*a*被填充并充满字符*fillChar*。正值产生右对齐文本；负值会产生左对齐文本。

```
double d = 12.34;
QString str = QString("delta: %1").arg(d, 0, 'E', 3);
// str == "delta: 1.234E+01"
```

也可以看看QLocale::toString(),QLocale::FloatingPointPrecisionOption， 和Number Formats。

QString QString::arg(char *a*, int *fieldWidth* = 0, *QChar* *fillChar* = u' ') const

该函数重载了 arg()。

这*a*参数被解释为 Latin-1 字符。

QString QString::arg(*QChar* *a*, int *fieldWidth* = 0, *QChar* *fillChar* = u' ') const

该函数重载了 arg()。

QString QString::arg(*QStringView* *a*, int *fieldWidth* = 0, *QChar* *fillChar* = u' ') const

这是一个过载功能。

返回此字符串的副本，其中编号最小的位置标记被字符串替换*a*，即%1，，，%2...，%99。

*fieldWidth*指定最小空间量*a*应占据。如果*a*需要的空间小于*fieldWidth*，它被填充为*fieldWidth*有性格*fillChar*。积极的*fieldWidth*生成右对齐文本。负数*fieldWidth*生成左对齐文本。

此示例展示了我们如何status在处理文件列表时创建一个用于报告进度的字符串：

```
int i;                // current file's number
int total;            // number of files to process
QStringView fileName; // current file's name

QString status = QString("Processing file %1 of %2: %3")
                 .arg(i).arg(total).arg(fileName);
```

首先，`arg(i)`替换`%1`. 然后`arg(total)`替换`%2`. 最后，`arg(fileName)`替换`%3`.

使用 arg() 的优点之一是asprintf() 的缺点是，如果应用程序的字符串被翻译成其他语言，则编号位置标记的顺序可以更改，但每个 arg() 仍将替换编号最小的未替换位置标记，无论它出现在何处。此外，如果地点标记`%i`在字符串中出现多次，则 arg() 会替换所有这些标记。

如果没有剩余未替换的地点标记，则会打印一条警告消息，并且结果未定义。地点标记编号必须在 1 到 99 范围内。

QString QString::arg(QLatin1StringView a, int fieldWidth = 0, QChar fillChar = u' ') const

这是一个过载功能。

返回此字符串的副本，其中编号最小的位置标记被替换为通过查看的 Latin-1 字符串*a*，即`%1`，，，`%2...`，`%99`。

*fieldWidth*指定最小空间量*a*应占据。如果*a*需要的空间小于*fieldWidth*，它被填充为*fieldWidth*有性格*fillChar*。积极的*fieldWidth*生成右对齐文本。负数*fieldWidth*生成左对齐文本。

使用 arg() 的优点之一是asprintf() 的缺点是，如果应用程序的字符串被翻译成其他语言，则编号位置标记的顺序可以更改，但每个 arg() 仍将替换编号最小的未替换位置标记，无论它出现在何处。此外，如果地点标记`%i`在字符串中出现多次，则 arg() 会替换所有这些标记。

如果没有剩余未替换的地点标记，则会打印一条警告消息，并且结果未定义。地点标记编号必须在 1 到 99 范围内。

*[static]QString QString::asprintf(const char *cformat, ...)*

从格式字符串安全地构建格式化字符串*cformat*和任意参数列表。

格式字符串支持标准 C++ 库中 printf() 提供的转换说明符、长度修饰符和标志。这*cformat*字符串和`%s`参数必须采用 UTF-8 编码。

注意：转义序列`%lc`需要类型为 的 unicode 字符`char16_t`，或`ushort`（由QChar::unicode()）。转义`%ls`序列需要一个指向 或 ushort 类型的以零结尾的 unicode 字符数组的指针`char16_t`（由QString::utf16()）。这与标准 C++ 库中的 printf() 不一致，后者定义`%lc`打印 wchar_t 和`%ls`打印 a `wchar_t*`，并且还可能在 的大小`wchar_t`不是 16 位的平台上产生编译器警告。

警告：我们不建议在新的 Qt 代码中使用 QString::asprintf()。相反，请考虑使用QTextStream或者arg()，两者都无缝支持 Unicode 字符串并且是类型安全的。这是一个使用的示例QTextStream:

```
QString result;
QTextStream(&result) << "pi = " << 3.14;
// result == "pi = 3.14"
```

为了translations，特别是如果字符串包含多个转义序列，您应该考虑使用arg() 函数代替。这允许翻译器控制替换的顺序。

也可以看看arg()。

const QChar QString::at(qsizetype position) const

返回给定索引处的字符*position*在字符串中。

这*position*必须是字符串中的有效索引位置（即 0 <=*position*<size()）。

也可以看看[operator]()。

QChar QString::back() const

返回字符串中的最后一个字符。与 相同`at(size() - 1)`。

提供此函数是为了兼容 STL。

警告：在空字符串上调用此函数会构成未定义的行为。

也可以看看front(),at（），和[operator]()。

QChar &QString::back()

返回对字符串中最后一个字符的引用。与 相同`operator[](size() - 1)`。

提供此函数是为了兼容 STL。

警告：在空字符串上调用此函数会构成未定义的行为。

也可以看看front(),at（），和[operator]()。

QString::iterator QString::begin()

返回一个STL-style iterator指向字符串中的第一个字符。

警告：返回的迭代器在分离或当QString被修改。

也可以看看constBegin（） 和end()。

QString::const_iterator QString::begin() const

该函数重载了begin()。

qsizetype QString::capacity() const

返回在不强制重新分配的情况下可以存储在字符串中的最大字符数。

该函数的唯一目的是提供一种微调的手段QString的内存使用情况。一般来说，您很少需要调用此函数。如果你想知道字符串中有多少个字符，请调用size()。

注意：静态分配的字符串将报告容量为 0，即使它不为空。

注意：分配的内存块中的空闲空间位置是未定义的。换句话说，不应假设空闲内存始终位于初始化元素之后。

也可以看看reserve（） 和squeeze()。

QString::const_iterator QString::cbegin() const

返回一个常量STL-style iterator指向字符串中的第一个字符。

警告：返回的迭代器在分离或当QString被修改。

也可以看看begin（） 和cend()。

QString::const_iterator QString::cend() const

返回一个常量STL-style iterator指向字符串中最后一个字符之后。

警告：返回的迭代器在分离或当QString被修改。

也可以看看cbegin（） 和end()。

void QString::chop(qsizetype n)

删除*n*从字符串末尾开始的字符。

如果*n*大于或等于size()，结果为空字符串；如果*n*为负数，相当于传递零。

例子：

```
QString str("LOGOUT\r\n");
str.chop(2);
// str == "LOGOUT"
```

如果要从字符串*开头删除字符，请使用*remove（） 反而。

也可以看看truncate(),resize(),remove（） ， 和QStringView::chop()。

QString QString::chopped(qsizetype len) const

返回一个字符串，其中包含size()-*len*该字符串最左边的字符。

注意：如果出现以下情况，则行为未定义*len*为负数或大于size()。

也可以看看endsWith(),first(),last(),sliced(),chop（），和truncate()。

void QString::clear()

清除字符串的内容并使其为空。

也可以看看resize（）和isNull()。

[static]int QString::compare(const QString &s1, const QString &s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)

比较*s1*和*s2*并返回一个小于、等于或大于零的整数，如果*s1*小于、等于或大于*s2*。

如果*cs*是Qt::CaseSensitive（默认），比较区分大小写；否则比较不区分大小写。

区分大小写的比较完全基于字符的数字 Unicode 值，并且速度非常快，但这并不是人们所期望的。考虑对用户可见的字符串进行排序localeAwareCompare()。

```
int x = QString::compare("aUtO", "AuTo", Qt::CaseInsensitive);  // x == 0
int y = QString::compare("auto", "Car", Qt::CaseSensitive);     // y > 0
int z = QString::compare("auto", "Car", Qt::CaseInsensitive);   // z < 0
```

注意：该函数将空字符串视为空字符串，有关更多详细信息，请参见Distinction Between Null and Empty Strings。

也可以看看operator==(),operator<(),operator>（），和Comparing Strings。

int QString::compare(const QString &other, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了compare()。

词法上将此字符串与*other*字符串，如果该字符串小于、等于或大于另一个字符串，则返回小于、等于或大于零的整数。

与比较相同(*this*,other,cs*）。

int QString::compare(QLatin1StringView other, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了compare()。

与比较相同(*this*,other,cs*）。

int QString::compare(QStringView s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了compare()。

对此进行比较*s*，使用区分大小写设置*cs*。

int QString::compare(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了compare()。

对此进行比较*ch*，使用区分大小写设置*cs*。

[static]int QString::compare(const QString &s1, QLatin1StringView s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了compare()。

执行比较*s1*和*s2*，使用区分大小写设置*cs*。

[static]int QString::compare(QLatin1StringView s1, const QString &s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了compare()。

执行比较*s1*和*s2*，使用区分大小写设置*cs*。

[static]int QString::compare(const QString &s1, QStringView s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了compare()。

[static]int QString::compare(QStringView s1, const QString &s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了compare()。

QString::const_iterator QString::constBegin() const

返回一个常量STL-style iterator指向字符串中的第一个字符。

警告：返回的迭代器在分离或当QString被修改。

也可以看看begin（） 和constEnd()。

*const QChar *QString::constData() const*

返回一个指向存储在其中的数据的指针QString。该指针可用于访问组成字符串的字符。

请注意，仅当字符串未被修改时，指针才保持有效。

注意：返回的字符串不能以“\0”结尾。使用size() 来确定数组的长度。

也可以看看data(),[operator] （），和fromRawData()。

QString::const_iterator QString::constEnd() const

返回一个常量STL-style iterator指向字符串中最后一个字符之后。

警告：返回的迭代器在分离或当QString被修改。

也可以看看constBegin（） 和end()。

bool QString::contains(const QString &str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

返回`true`此字符串是否包含该字符串的出现*str*; 否则返回`false`.

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString str = "Peter Pan";
str.contains("peter", Qt::CaseInsensitive);    // returns true
```

也可以看看indexOf（） 和count()。

bool QString::contains(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了 contains()。

返回`true`此字符串是否包含该字符的出现*ch*; 否则返回`false`.

bool QString::contains(QLatin1StringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了 contains()。

返回`true`此字符串是否包含 latin-1 字符串的出现*str*; 否则返回`false`.

bool QString::contains(QStringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了 contains()。

返回`true`此字符串是否包含字符串视图的出现*str*; 否则返回`false`.

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

也可以看看indexOf（） 和count()。

*bool QString::contains(const QRegularExpression &re, QRegularExpressionMatch *rmatch = nullptr) const*

返回`true`if 正则表达式*re*匹配该字符串中的某处；否则返回`false`.

如果比赛成功并且*rmatch*不是`nullptr`，它还将匹配结果写入QRegularExpressionMatch指向的对象*rmatch*。

也可以看看QRegularExpression::match()。

qsizetype QString::count(const QString &str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

返回字符串出现的次数（可能重叠）*str*在这个字符串中。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

也可以看看contains（） 和indexOf()。

qsizetype QString::count(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了 count()。

返回字符出现的次数*ch*在字符串中。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

也可以看看contains（） 和indexOf()。

[since 6.0]qsizetype QString::count(QStringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了 count()。

返回字符串视图出现的次数（可能重叠）*str*在这个字符串中。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

这个函数是在Qt 6.0中引入的。

也可以看看contains（） 和indexOf()。

qsizetype QString::count(const QRegularExpression &re) const

该函数重载了 count()。

返回正则表达式出现的次数*re*字符串中的匹配项。

由于历史原因，该函数对重叠匹配进行计数，因此在下面的示例中，有四个“ana”或“ama”实例：

```
QString str = "banana and panama";
str.count(QRegularExpression("a[nm]a"));    // returns 4
```

此行为不同于使用以下命令简单地迭代字符串中的匹配项QRegularExpressionMatchIterator。

也可以看看QRegularExpression::globalMatch()。

QString::const_reverse_iterator QString::crbegin() const

返回一个常量STL-style反向迭代器以相反的顺序指向字符串中的第一个字符。

警告：返回的迭代器在分离或当QString被修改。

也可以看看begin(),rbegin（），和rend()。

QString::const_reverse_iterator QString::crend() const

返回一个常量STL-style反向迭代器以相反的顺序指向字符串中最后一个字符之后。

警告：返回的迭代器在分离或当QString被修改。

也可以看看end(),rend（），和rbegin()。

*QChar *QString::data()*

返回一个指向存储在其中的数据的指针QString。该指针可用于访问和修改组成字符串的字符。

不像constData（）和unicode() 时，返回的数据始终以 '\0' 结尾。

例子：

```
QString str = "Hello world";
QChar *data = str.data();
while (!data->isNull()) {
    qDebug() << data->unicode();
    ++data;
}
```

请注意，仅当字符串未被其他方式修改时，指针才保持有效。对于只读访问，constData() 更快，因为它永远不会导致deep copy发生。

也可以看看constData（） 和[operator]()。

*const QChar *QString::data() const*

这是一个过载功能。

注意：返回的字符串不能以“\0”结尾。使用size() 来确定数组的长度。

也可以看看fromRawData()。

QString::iterator QString::end()

返回一个STL-style iterator指向字符串中最后一个字符之后。

警告：返回的迭代器在分离或当QString被修改。

也可以看看begin（） 和constEnd()。

QString::const_iterator QString::end() const

该函数重载了end()。

bool QString::endsWith(const QString &s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

`true`如果字符串以以下结尾则返回*s*; 否则返回`false`.

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

```
QString str = "Bananas";
str.endsWith("anas");         // returns true
str.endsWith("pple");         // returns false
```

也可以看看startsWith()。

bool QString::endsWith(QStringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了endsWith()。

返回true字符串是否以字符串视图结尾*str*; 否则返回false.

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

也可以看看startsWith()。

bool QString::endsWith(QLatin1StringView s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了endsWith()。

bool QString::endsWith(QChar c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

true如果字符串以以下结尾则返回*c*; 否则返回false.

该函数重载了endsWith()。

[since 6.1]QString::iterator QString::erase(QString::const_iterator first, QString::const_iterator last)

从字符串中删除半开范围 [中的字符*first*,*last*）。返回一个迭代器，指向紧接最后一个擦除字符之后的字符（即由*last*擦除之前）。

该功能是在 Qt 6.1 中引入的。

[since 6.5]QString::iterator QString::erase(QString::const_iterator it)

it从字符串中删除 表示的字符。返回一个指向被擦除字符之后的字符的迭代器。

```
QString c = "abcdefg";
auto it = c.erase(c.cbegin()); // c is now "bcdefg"; "it" points to "b"
```

该功能是在 Qt 6.5 中引入的。

QString &QString::fill(QChar ch, qsizetype size = -1)

将字符串中的每个字符设置为字符*ch*。如果*size*与 -1（默认）不同，字符串大小调整为*size*预先。

例子：

```
QString str = "Berlin";
str.fill('z');
// str == "zzzzzz"

str.fill('A', 2);
// str == "AA"
```

也可以看看resize()。

[since 6.0]QString QString::first(qsizetype n) const

返回包含第一个的字符串*n*该字符串的字符。

注意：当以下情况时，行为未定义*n*< 0 或*n*>size()。

```
QString x = "Pineapple";
QString y = x.first(4);      // y == "Pine"
```

这个函数是在Qt 6.0中引入的。

也可以看看last(),sliced(),startsWith(),chopped(),chop（），和truncate()。

[static]QString QString::fromCFString(CFStringRef string)

构造一个新的QString包含一份副本*string*CFString。

注意：此功能仅在 macOS 和 iOS 上可用。

*[static]QString QString::fromLatin1(const char *str, qsizetype size)*

返回一个QString用第一个初始化*size*Latin-1 字符串的字符*str*。

如果*size*是-1,strlen(str)被用来代替。

也可以看看toLatin1(),fromUtf8（），和fromLocal8Bit()。

[static, since 6.0]QString QString::fromLatin1(QByteArrayView str)

这是一个过载功能。

返回一个QString使用 Latin-1 字符串初始化*str*。

注意：：字节数组中的任何空（'\0'）字节都将包含在此字符串中，并转换为 Unicode 空字符 (U+0000)。

这个函数是在Qt 6.0中引入的。

[static]QString QString::fromLatin1(const QByteArray &str)

这是一个过载功能。

返回一个QString使用 Latin-1 字符串初始化*str*。

注意：：字节数组中的任何空（'\0'）字节都将包含在此字符串中，并转换为 Unicode 空字符 (U+0000)。此行为与 Qt 5.x 不同。

[static]QString QString::fromLocal8Bit(const char *str, qsizetype size)

返回一个QString用第一个初始化*size*8位字符串的字符*str*。

如果*size*是-1,`strlen(str)`被用来代替。

在 Unix 系统上这相当于fromUtf8()，在 Windows 上，系统正在使用当前代码页。

也可以看看toLocal8Bit(),fromLatin1（），和fromUtf8()。

[static, since 6.0]QString QString::fromLocal8Bit(QByteArrayView str)

这是一个过载功能。

返回一个QString用8位字符串初始化*str*。

注意：：字节数组中的任何空（'\0'）字节都将包含在此字符串中，并转换为 Unicode 空字符 (U+0000)。

这个函数是在Qt 6.0中引入的。

[static]QString QString::fromLocal8Bit(const QByteArray &str)

这是一个过载功能。

返回一个QString用8位字符串初始化*str*。

注意：：字节数组中的任何空（'\0'）字节都将包含在此字符串中，并转换为 Unicode 空字符 (U+0000)。此行为与 Qt 5.x 不同。

[static]QString QString::fromNSString(const NSString *string)

构造一个新的QString包含一份副本*string*NSString。

注意：此功能仅在 macOS 和 iOS 上可用。

[static]QString QString::fromRawData(const QChar *unicode, qsizetype size)

构造一个QString使用第一个*size*数组中的 Unicode 字符*unicode*。数据在*unicode*不被复制。调用者必须能够保证*unicode*只要QString（或其未经修改的副本）存在。

任何修改的尝试QString或其副本将导致它创建数据的深层副本，确保原始数据不被修改。

这是我们如何使用的示例QRegularExpression内存中的原始数据，无需将数据复制到QString:

```
QRegularExpression pattern("\u00A4");
static const QChar unicode[] = {
        0x005A, 0x007F, 0x00A4, 0x0060,
        0x1009, 0x0020, 0x0020};
qsizetype size = sizeof(unicode) / sizeof(QChar);

QString str = QString::fromRawData(unicode, size);
if (str.contains(pattern) {
    // ...
}
```

警告：使用 fromRawData() 创建的字符串不是*以*'\0' 结尾的，除非原始数据在该位置包含 '\0' 字符*size*。这意味着unicode() 不会*返回*以 '\0' 结尾的字符串（尽管utf16() 确实如此，但以复制原始数据为代价）。

也可以看看fromUtf16（） 和setRawData()。

[static]QString QString::fromStdString(const std::string &str)

返回一个副本*str*细绳。使用以下命令将给定字符串转换为 UnicodefromUtf8（） 功能。

也可以看看fromLatin1(),fromLocal8Bit(),fromUtf8（）， 和QByteArray::fromStdString()。

[static]QString QString::fromStdU16String(const std::u16string &str)

返回一个副本*str*细绳。假定给定的字符串以 UTF-16 编码。

也可以看看fromUtf16(),fromStdWString（）， 和fromStdU32String()。

[static]QString QString::fromStdU32String(const std::u32string &str)

返回一个副本*str*细绳。假定给定的字符串采用 UCS-4 进行编码。

也可以看看fromUcs4(),fromStdWString（），和fromStdU16String()。

[static]QString QString::fromStdWString(const std::wstring &str)

返回一个副本*str*细绳。如果 wchar_t 的大小为 2 字节（例如在 Windows 上），则假定给定字符串以 utf16 编码；如果 wchar_t 的大小为 4 字节（大多数 Unix 系统），则假定以 ucs4 编码。

也可以看看fromUtf16(),fromLatin1(),fromLocal8Bit(),fromUtf8(),fromUcs4(),fromStdU16String（），和fromStdU32String()。

[static]QString QString::fromUcs4(const char32_t *unicode, qsizetype size = -1)

返回一个QString用第一个初始化*size*Unicode 字符串的字符*unicode*（ISO-10646-UCS-4 编码）。

如果*size*为 -1（默认），*unicode*必须以 \0' 结尾。

也可以看看toUcs4(),fromUtf16(),utf16(),setUtf16(),fromWCharArray（），和fromStdU32String()。

[static]QString QString::fromUtf8(const char *str, qsizetype size)

返回一个QString用第一个初始化*size*UTF-8 字符串的字节*str*。

如果*size*是-1,`strlen(str)`被用来代替。

UTF-8 是一种 Unicode 编解码器，可以表示 Unicode 字符串中的所有字符，例如QString。然而，UTF-8 可能存在无效序列，如果发现任何此类序列，它们将被一个或多个“替换字符”替换，或被抑制。其中包括非 Unicode 序列、非字符、超长序列或编码为 UTF-8 的代理代码点。

只要所有 UTF-8 字符都在传入数据中终止，此函数即可用于增量处理传入数据。字符串末尾的任何未终止字符都将被替换或抑制。为了进行状态解码，请使用QStringDecoder。

也可以看看toUtf8(),fromLatin1（），和fromLocal8Bit()。

[static, since 6.0]QString QString::fromUtf8(QByteArrayView str)

这是一个过载功能。

返回一个QString使用 UTF-8 字符串初始化*str*。

注意：：字节数组中的任何空（'\0'）字节都将包含在此字符串中，并转换为 Unicode 空字符 (U+0000)。

这个函数是在Qt 6.0中引入的。

[static]QString QString::fromUtf8(const QByteArray &str)

这是一个过载功能。

返回一个QString使用 UTF-8 字符串初始化*str*。

注意：：字节数组中的任何空（'\0'）字节都将包含在此字符串中，并转换为 Unicode 空字符 (U+0000)。此行为与 Qt 5.x 不同。

[static, since 6.1]QString QString::fromUtf8(const char8_t *str)

这是一个过载功能。

此重载仅在 C++20 模式下编译时可用。

该功能是在 Qt 6.1 中引入的。

[static, since 6.0]QString QString::fromUtf8(const char8_t *str, qsizetype size)

这是一个过载功能。

此重载仅在 C++20 模式下编译时可用。

这个函数是在Qt 6.0中引入的。

[static]QString QString::fromUtf16(const char16_t *unicode, qsizetype size = -1)

返回一个QString用第一个初始化*size*Unicode 字符串的字符*unicode*（ISO-10646-UTF-16 编码）。

如果*size*为 -1（默认），*unicode*必须以 \0' 结尾。

此函数检查字节顺序标记 (BOM)。如果缺少，则采用主机字节顺序。

与其他 Unicode 转换相比，此函数速度较慢。使用QString（常量QChar, *qsizetype) 或QString（常量QChar*） 如果可能的话。

QString制作 Unicode 数据的深层副本。

也可以看看utf16(),setUtf16（），和fromStdU16String()。

[static]QString QString::fromWCharArray(const wchar_t *string, qsizetype size = -1)

返回一个副本*string*，其中的编码*string*取决于 wchar 的大小。如果 wchar 是 4 个字节，则*string*被解释为 UCS-4，如果 wchar 是 2 个字节，则被解释为 UTF-16。

如果*size*为 -1（默认），*string*必须以 '\0' 结尾。

也可以看看fromUtf16(),fromLatin1(),fromLocal8Bit(),fromUtf8(),fromUcs4（），和fromStdWString()。

QChar QString::front() const

返回字符串中的第一个字符。与 相同`at(0)`。

提供此函数是为了兼容 STL。

警告：在空字符串上调用此函数会构成未定义的行为。

也可以看看back(),at（），和[operator]()。

QChar &QString::front()

返回对字符串中第一个字符的引用。与 相同`operator[](0)`。

提供此函数是为了兼容 STL。

警告：在空字符串上调用此函数会构成未定义的行为。

也可以看看back(),at（），和[operator]()。

qsizetype QString::indexOf(QLatin1StringView str, qsizetype from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

返回通过查看的 Latin-1 字符串第一次出现的索引位置*str*在此字符串中，从索引位置向前搜索*from*。如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString x = "sticky question";
QString y = "sti";
x.indexOf(y);               // returns 0
x.indexOf(y, 1);            // returns 10
x.indexOf(y, 10);           // returns 10
x.indexOf(y, 11);           // returns -1
```

如果*from*为-1时，从最后一个字符开始搜索；如果是 -2，则在倒数第二个字符处，依此类推。

也可以看看lastIndexOf(),contains（），和count()。

qsizetype QString::indexOf(QChar ch, qsizetype from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了indexOf()。

返回字符第一次出现的索引位置*ch*在此字符串中，从索引位置向前搜索*from*。如果返回 -1*ch*没有找到。

qsizetype QString::indexOf(const QString &str, qsizetype from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

返回字符串第一次出现的索引位置*str*在此字符串中，从索引位置向前搜索*from*。如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString x = "sticky question";
QString y = "sti";
x.indexOf(y);               // returns 0
x.indexOf(y, 1);            // returns 10
x.indexOf(y, 10);           // returns 10
x.indexOf(y, 11);           // returns -1
```

如果*from*为-1时，从最后一个字符开始搜索；如果是 -2，则在倒数第二个字符处，依此类推。

也可以看看lastIndexOf(),contains（），和count()。

qsizetype QString::indexOf(QStringView str, qsizetype from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了indexOf()。

返回字符串视图第一次出现的索引位置*str*在此字符串中，从索引位置向前搜索*from*。如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

如果*from*为-1时，从最后一个字符开始搜索；如果是 -2，则在倒数第二个字符处，依此类推。

也可以看看QStringView::indexOf(),lastIndexOf(),contains（），和count()。

*qsizetype QString::indexOf(const QRegularExpression &re, qsizetype from = 0, QRegularExpressionMatch *rmatch = nullptr) const*

返回正则表达式第一个匹配的索引位置*re*在字符串中，从索引位置向前搜索*from*。如果返回 -1*re*任何地方都不匹配。

如果比赛成功并且*rmatch*不是`nullptr`，它还将匹配结果写入QRegularExpressionMatch指向的对象*rmatch*。

例子：

```
QString str = "the minimum";
str.indexOf(QRegularExpression("m[aeiou]"), 0);       // returns 4

QString str = "the minimum";
QRegularExpressionMatch match;
str.indexOf(QRegularExpression("m[aeiou]"), 0, &match);       // returns 4
// match.captured() == mi
```

QString &QString::insert(qsizetype position, const QString &str)

插入字符串*str*在给定索引处*position*并返回对此字符串的引用。

例子：

```
QString str = "Meal";
str.insert(1, QString("ontr"));
// str == "Montreal"
```

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*str*。

也可以看看append(),prepend(),replace（），和remove()。

QString &QString::insert(qsizetype position, QChar ch)

该函数重载了 insert()。

刀片*ch*在给定索引处*position*在字符串中。

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*ch*。

*QString &QString::insert(qsizetype position, const QChar *unicode, qsizetype size)*

该函数重载了 insert()。

插入第一个*size*的字符QChar大批*unicode*在给定索引处*position*在字符串中。

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*size*的字符QChar大批*unicode*。

[since 6.0]QString &QString::insert(qsizetype position, QStringView str)

该函数重载了 insert()。

插入字符串视图*str*在给定索引处*position*并返回对此字符串的引用。

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*str*。

这个函数是在Qt 6.0中引入的。

QString &QString::insert(qsizetype position, QLatin1StringView str)

该函数重载了 insert()。

插入通过查看的 Latin-1 字符串*str*在给定索引处*position*。

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*str*。

[since 6.5]QString &QString::insert(qsizetype position, QUtf8StringView str)

该函数重载了 insert()。

插入 UTF-8 字符串视图*str*在给定索引处*position*。

注意： 插入可变宽度 UTF-8 编码的字符串数据在概念上比插入固定宽度字符串数据（例如 UTF-16）慢（QStringView) 或 Latin-1 (QLatin1StringView），因此应谨慎使用。

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*str*。

该功能是在 Qt 6.5 中引入的。

*QString &QString::insert(qsizetype position, const char *str)*

该函数重载了 insert()。

插入 C 字符串*str*在给定索引处*position*并返回对此字符串的引用。

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*str*。

此功能在以下情况下不可用QT_NO_CAST_FROM_ASCII被定义为。

QString &QString::insert(qsizetype position, const QByteArray &str)

该函数重载了 insert()。

解读内容*str*作为 UTF-8，在给定索引处插入它编码的 Unicode 字符串*position*并返回对此字符串的引用。

该字符串会增长以适应插入。如果*position*超出字符串末尾，将空格字符附加到字符串以达到此目的*position*， 其次是*str*。

此功能在以下情况下不可用QT_NO_CAST_FROM_ASCII被定义为。

bool QString::isEmpty() const

`true`如果字符串没有字符则返回；否则返回`false`.

例子：

```
QString().isEmpty();            // returns true
QString("").isEmpty();          // returns true
QString("x").isEmpty();         // returns false
QString("abc").isEmpty();       // returns false
```

也可以看看size()。

bool QString::isLower() const

返回`true`字符串是否为小写，即与其相同toLower（） 折叠式的。

请注意，这并不*意味着*字符串不包含大写字母（某些大写字母没有小写折叠；它们保持不变toLower()）。有关详细信息，请参阅 Unicode 标准第 3.13 节。

也可以看看QChar::toLower（） 和isUpper()。

bool QString::isNull() const

`true`如果该字符串为空则返回；否则返回`false`.

例子：

```
QString().isNull();             // returns true
QString("").isNull();           // returns false
QString("abc").isNull();        // returns false
```

由于历史原因，Qt 对空字符串和空字符串进行了区分。对于大多数应用程序来说，重要的是字符串是否包含任何数据，这可以使用isEmpty（） 功能。

也可以看看isEmpty()。

bool QString::isRightToLeft() const

`true`如果字符串是从右向左读取的，则返回。

也可以看看QStringView::isRightToLeft()。

bool QString::isUpper() const

返回`true`字符串是否为大写，即与其相同toUpper（） 折叠式的。

请注意，这并不*意味着*字符串不包含小写字母（某些小写字母没有大写折叠；它们保持不变toUpper()）。有关详细信息，请参阅 Unicode 标准第 3.13 节。

也可以看看QChar::toUpper（） 和isLower()。

bool QString::isValidUtf16() const

返回`true`字符串是否包含有效的 UTF-16 编码数据，否则返回`false`。

请注意，此函数不会对数据执行任何特殊验证；它只是检查是否可以成功从 UTF-16 解码。假定数据采用主机字节顺序；BOM 的存在是没有意义的。

也可以看看QStringView::isValidUtf16()。

[since 6.0]QString QString::last(qsizetype n) const

返回包含最后一个的字符串*n*该字符串的字符。

注意：当以下情况时，行为未定义*n*< 0 或*n*>size()。

```
QString x = "Pineapple";
QString y = x.last(5);      // y == "apple"
```

这个函数是在Qt 6.0中引入的。

也可以看看first(),sliced(),endsWith(),chopped(),chop（），和truncate()。

qsizetype QString::lastIndexOf(const QString &str, qsizetype from, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

返回字符串最后一次出现的索引位置*str*在此字符串中，从索引位置向后搜索*from*。

如果*from*为-1时，从最后一个字符开始搜索；如果是 -2，则在倒数第二个字符处，依此类推。

如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString x = "crazy azimuths";
QString y = "az";
x.lastIndexOf(y);           // returns 6
x.lastIndexOf(y, 6);        // returns 6
x.lastIndexOf(y, 5);        // returns 2
x.lastIndexOf(y, 1);        // returns -1
```

注意：当搜索0长度时*str*，数据末尾的匹配项被否定排除在搜索之外*from*，尽管-1通常被认为是从字符串末尾开始搜索：末尾的匹配位于最后一个字符*之后*，因此被排除。要包含这样的最终空匹配，请为*from*或省略*from*参数完全。

也可以看看indexOf(),contains（），和count()。

[since 6.3]qsizetype QString::lastIndexOf(QChar c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了lastIndexOf()。

该功能是在 Qt 6.3 中引入的。

qsizetype QString::lastIndexOf(QChar ch, qsizetype from, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了lastIndexOf()。

返回该字符最后一次出现的索引位置*ch*在此字符串中，从索引位置向后搜索*from*。

[since 6.2]qsizetype QString::lastIndexOf(QLatin1StringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了lastIndexOf()。

返回字符串最后一次出现的索引位置*str*在这个字符串中。如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString x = "crazy azimuths";
QString y = "az";
x.lastIndexOf(y);           // returns 6
x.lastIndexOf(y, 6);        // returns 6
x.lastIndexOf(y, 5);        // returns 2
x.lastIndexOf(y, 1);        // returns -1
```

该功能是在 Qt 6.2 中引入的。

也可以看看indexOf(),contains（），和count()。

qsizetype QString::lastIndexOf(QLatin1StringView str, qsizetype from, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了lastIndexOf()。

返回按以下方式查看的 Latin-1 字符串最后一次出现的索引位置*str*在此字符串中，从索引位置向后搜索*from*。

如果*from*为-1时，从最后一个字符开始搜索；如果是 -2，则在倒数第二个字符处，依此类推。

如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString x = "crazy azimuths";
QString y = "az";
x.lastIndexOf(y);           // returns 6
x.lastIndexOf(y, 6);        // returns 6
x.lastIndexOf(y, 5);        // returns 2
x.lastIndexOf(y, 1);        // returns -1
```

注意：当搜索0长度时*str*，数据末尾的匹配项被否定排除在搜索之外*from*，尽管-1通常被认为是从字符串末尾开始搜索：末尾的匹配位于最后一个字符之后，因此被排除。要包含这样的最终空匹配，请为*from*或省略*from*参数完全。

也可以看看indexOf(),contains（），和count()。

[since 6.2]qsizetype QString::lastIndexOf(const QString &str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了lastIndexOf()。

返回字符串最后一次出现的索引位置*str*在这个字符串中。如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString x = "crazy azimuths";
QString y = "az";
x.lastIndexOf(y);           // returns 6
x.lastIndexOf(y, 6);        // returns 6
x.lastIndexOf(y, 5);        // returns 2
x.lastIndexOf(y, 1);        // returns -1
```

该功能是在 Qt 6.2 中引入的。

也可以看看indexOf(),contains（），和count()。

[since 6.2]qsizetype QString::lastIndexOf(QStringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了lastIndexOf()。

返回字符串视图最后一次出现的索引位置*str*在这个字符串中。如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

该功能是在 Qt 6.2 中引入的。

也可以看看indexOf(),contains（），和count()。

qsizetype QString::lastIndexOf(QStringView str, qsizetype from, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载了lastIndexOf()。

返回字符串视图最后一次出现的索引位置*str*在此字符串中，从索引位置向后搜索*from*。

如果*from*为-1时，从最后一个字符开始搜索；如果是 -2，则在倒数第二个字符处，依此类推。

如果返回 -1*str*没有找到。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

注意：当搜索0长度时*str*，数据末尾的匹配项被否定排除在搜索之外*from*，尽管-1通常被认为是从字符串末尾开始搜索：末尾的匹配位于最后一个字符之后，因此被排除。要包含这样的最终空匹配，请为*from*或省略*from*参数完全。

也可以看看indexOf(),contains（），和count()。

*[since 6.2]qsizetype QString::lastIndexOf(const QRegularExpression &re, QRegularExpressionMatch *rmatch = nullptr) const*

该函数重载了lastIndexOf()。

返回正则表达式最后一个匹配的索引位置*re*在字符串中。如果返回 -1*re*任何地方都不匹配。

如果比赛成功并且*rmatch*不是`nullptr`，它还将匹配结果写入QRegularExpressionMatch指向的对象*rmatch*。

例子：

```
QString str = "the minimum";
str.lastIndexOf(QRegularExpression("m[aeiou]"));      // returns 8

QString str = "the minimum";
QRegularExpressionMatch match;
str.lastIndexOf(QRegularExpression("m[aeiou]"), -1, &match);      // returns 8
// match.captured() == mu
```

注意：由于正则表达式匹配算法的工作原理，该函数实际上会从字符串的开头重复匹配，直到到达字符串的结尾。

该功能是在 Qt 6.2 中引入的。

*qsizetype QString::lastIndexOf(const QRegularExpression &re, qsizetype from, QRegularExpressionMatch *rmatch = nullptr) const*

返回正则表达式最后一个匹配的索引位置*re*在字符串中，在索引位置之前开始*from*。

如果*from*为-1时，从最后一个字符开始搜索；如果是 -2，则在倒数第二个字符处，依此类推。

如果返回 -1*re*任何地方都不匹配。

如果比赛成功并且*rmatch*不是nullptr，它还将匹配结果写入QRegularExpressionMatch指向的对象*rmatch*。

例子：

```
QString str = "the minimum";
str.lastIndexOf(QRegularExpression("m[aeiou]"));      // returns 8

QString str = "the minimum";
QRegularExpressionMatch match;
str.lastIndexOf(QRegularExpression("m[aeiou]"), -1, &match);      // returns 8
// match.captured() == mu
```

注意：由于正则表达式匹配算法的工作原理，该函数实际上会从字符串的开头重复匹配到该位置*from*到达了。

注意：搜索正则表达式时*re*可能匹配 0 个字符，则数据末尾的匹配项将被排除在搜索之外*from*，尽管-1通常被认为是从字符串末尾开始搜索：末尾的匹配位于最后一个字符*之后*，因此被排除。要包含这样的最终空匹配，请为*from*或省略*from*参数完全。

QString QString::left(qsizetype n) const

返回包含以下内容的子字符串*n*字符串最左边的字符。

如果你知道的话*n*不能越界，使用first() 改为在新代码中，因为它更快。

如果返回整个字符串*n*大于或等于size()，或小于零。

也可以看看first(),last(),startsWith(),chopped(),chop（），和truncate()。

QString QString::leftJustified(qsizetype width, *QChar* fill = u' ', bool truncate = false) const

返回一个大小的字符串*width*包含由以下填充的字符串*fill*特点。

如果*truncate*是false并且size字符串的 () 大于*width*，那么返回的字符串是该字符串的副本。

```
QString s = "apple";
QString t = s.leftJustified(8, '.');    // t == "apple..."
```

如果*truncate*是true并且size字符串的 () 大于*width*，然后是该字符串副本中位置之后的任何字符*width*被删除，并返回副本。

```
QString str = "Pineapple";
str = str.leftJustified(5, '.', true);    // str == "Pinea"
```

也可以看看rightJustified()。

qsizetype QString::length() const

返回该字符串中的字符数。相当于size()。

也可以看看resize()。

[static]int QString::localeAwareCompare(const QString &s1, const QString &s2)

比较*s1*和*s2*并返回一个小于、等于或大于零的整数，如果*s1*小于、等于或大于*s2*。

比较以区域设置和平台相关的方式执行。使用此函数向用户呈现排序的字符串列表。

也可以看看compare(),QLocale， 和Comparing Strings。

int QString::localeAwareCompare(const QString &other) const

该函数重载了 localeAwareCompare()。

将此字符串与*other*字符串，如果该字符串小于、等于或大于，则返回小于、等于或大于零的整数*other*细绳。

比较以区域设置和平台相关的方式执行。使用此函数向用户呈现排序的字符串列表。

与 相同`localeAwareCompare(*this, other)`。

也可以看看Comparing Strings。

[since 6.0]int QString::localeAwareCompare(QStringView other) const

该函数重载了 localeAwareCompare()。

将此字符串与*other*字符串，如果该字符串小于、等于或大于，则返回小于、等于或大于零的整数*other*细绳。

比较以区域设置和平台相关的方式执行。使用此函数向用户呈现排序的字符串列表。

与 相同`localeAwareCompare(*this, other)`。

这个函数是在Qt 6.0中引入的。

也可以看看Comparing Strings。

[static, since 6.0]int QString::localeAwareCompare(QStringView s1, QStringView s2)

该函数重载了 localeAwareCompare()。

比较*s1*和*s2*并返回一个小于、等于或大于零的整数，如果*s1*小于、等于或大于*s2*。

比较以区域设置和平台相关的方式执行。使用此函数向用户呈现排序的字符串列表。

这个函数是在Qt 6.0中引入的。

也可以看看Comparing Strings。

QString QString::mid(qsizetype position, qsizetype n = -1) const

返回一个字符串，其中包含*n*该字符串的字符，从指定的位置开始*position*指数。

如果你知道的话*position*和*n*不能越界，使用sliced() 改为在新代码中，因为它更快。

如果返回空字符串*position*索引超出了字符串的长度。如果有少于*n*字符串中从给定开始的可用字符*position*， 或者如果*n*为 -1（默认），该函数返回指定的所有可用字符*position*。

也可以看看first(),last(),sliced(),chopped(),chop（），和truncate()。

QString QString::normalized(QString::NormalizationForm mode, QChar::UnicodeVersion version = QChar::Unicode_Unassigned) const

返回给定 Unicode 标准化的字符串*mode*，根据给定的*version*Unicode 标准。

[static]QString QString::number(long n, int base = 10)

返回与数字等效的字符串*n*根据指定的*base*。

默认基数为 10，且必须介于 2 到 36 之间。对于 10 以外的基数，*n*被视为无符号整数。

格式始终使用QLocale::C，即英语/美国。要获取数字的本地化字符串表示形式，请使用QLocale::toString() 与适当的区域设置。

```
long a = 63;
QString s = QString::number(a, 16);             // s == "3f"
QString t = QString::number(a, 16).toUpper();     // t == "3F"
```

也可以看看setNum()。

[static]QString QString::number(int n, int base = 10)

这是一个过载功能。

[static]QString QString::number(uint n, int base = 10)

这是一个过载功能。

[static]QString QString::number(ulong n, int base = 10)

这是一个过载功能。

[static]QString QString::number(qlonglong n, int base = 10)

这是一个过载功能。

[static]QString QString::number(qulonglong n, int base = 10)

这是一个过载功能。

[static]QString QString::number(double n, char format = 'g', int precision = 6)

返回表示浮点数的字符串*n*。

返回一个字符串，表示*n*，按照指定格式*format*和*precision*。

对于带有指数的格式，指数将显示其符号并且至少有两位数字，如果需要，可以在指数左侧填充零。

也可以看看setNum(),QLocale::toString(),QLocale::FloatingPointPrecisionOption， 和Number Formats。

QString &QString::prepend(const QString &str)

前置字符串*str*到该字符串的开头并返回对此字符串的引用。

此操作通常非常快（constant time）， 因为QString在字符串数据的开头预先分配额外的空间，因此它可以增长而无需每次重新分配整个字符串。

例子：

```
QString x = "ship";
QString y = "air";
x.prepend(y);
// x == "airship"
```

也可以看看append（） 和insert()。

QString &QString::prepend(QChar ch)

该函数重载了 prepend()。

前置字符*ch*到这个字符串。

*QString &QString::prepend(const QChar *str, qsizetype len)*

该函数重载了 prepend()。

前置*len*字符来自QChar大批*str*到此字符串并返回对此字符串的引用。

[since 6.0]QString &QString::prepend(QStringView str)

该函数重载了 prepend()。

前置字符串视图*str*到该字符串的开头并返回对此字符串的引用。

这个函数是在Qt 6.0中引入的。

QString &QString::prepend(QLatin1StringView str)

该函数重载了 prepend()。

前置 Latin-1 字符串*str*到这个字符串。

[since 6.5]QString &QString::prepend(QUtf8StringView str)

该函数重载了 prepend()。

前置 UTF-8 字符串视图*str*到这个字符串。

该功能是在 Qt 6.5 中引入的。

*QString &QString::prepend(const char *str)*

该函数重载了 prepend()。

前置字符串*str*到这个字符串。const char 指针使用以下命令转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用此功能QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QString &QString::prepend(const QByteArray &ba)

该函数重载了 prepend()。

前置字节数组*ba*到这个字符串。使用以下命令将字节数组转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用此功能QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

void QString::push_back(const QString &other)

提供此函数是为了兼容 STL，附加给定的*other*字符串到该字符串的末尾。它相当于`append(other)`.

也可以看看append()。

void QString::push_back(QChar ch)

这是一个过载功能。

追加给定的*ch*字符到该字符串的末尾。

void QString::push_front(const QString &other)

提供此函数是为了实现 STL 兼容性，预先考虑给定的*other*字符串到该字符串的开头。它相当于`prepend(other)`.

也可以看看prepend()。

void QString::push_front(QChar ch)

这是一个过载功能。

前置给定的*ch*字符到该字符串的开头。

QString::reverse_iterator QString::rbegin()

返回一个STL-style反向迭代器以相反的顺序指向字符串中的第一个字符。

警告：返回的迭代器在分离或当QString被修改。

也可以看看begin(),crbegin（），和rend()。

QString::const_reverse_iterator QString::rbegin() const

这是一个过载功能。

QString &QString::remove(qsizetype position, qsizetype n)

删除*n*字符串中的字符，从给定的位置开始*position*索引，并返回对字符串的引用。

如果指定*position*索引在字符串内，但是*position*+*n*超出字符串末尾，则字符串在指定位置被截断*position*。

如果*n*is <= 0 没有任何改变。

```
QString s = "Montreal";
s.remove(1, 4);
// s == "Meal"
```

元素删除将保留字符串的容量，并且不会减少分配的内存量。要释放额外容量并释放尽可能多的内存，请调用squeeze() 在最后一次更改字符串大小之后。

也可以看看insert（） 和replace()。

QString &QString::remove(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive)

删除每个出现的字符*ch*在此字符串中，并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString t = "Ali Baba";
t.remove(QChar('a'), Qt::CaseInsensitive);
// t == "li Bb"
```

这与 相同`replace(ch, "", cs)`。

元素删除将保留字符串的容量，并且不会减少分配的内存量。要释放额外容量并释放尽可能多的内存，请调用squeeze() 在最后一次更改字符串大小之后。

也可以看看replace()。

QString &QString::remove(QLatin1StringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive)

这是一个过载功能。

删除查看者查看的所有出现的给定 Latin-1 字符串*str*从此字符串，并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

这与 相同`replace(str, "", cs)`。

元素删除将保留字符串的容量，并且不会减少分配的内存量。要释放额外容量并释放尽可能多的内存，请调用squeeze() 在最后一次更改字符串大小之后。

也可以看看replace()。

QString &QString::remove(const QString &str, Qt::CaseSensitivity cs = Qt::CaseSensitive)

删除所有出现的给定值*str*string 在此字符串中，并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

这与 相同`replace(str, "", cs)`。

元素删除将保留字符串的容量，并且不会减少分配的内存量。要释放额外容量并释放尽可能多的内存，请调用squeeze() 在最后一次更改字符串大小之后。

也可以看看replace()。

QString &QString::remove(const QRegularExpression &re)

删除所有出现的正则表达式*re*在字符串中，并返回对该字符串的引用。例如：

```
QString r = "Telephone";
r.remove(QRegularExpression("[aeiou]."));
// r == "The"
```

元素删除将保留字符串的容量，并且不会减少分配的内存量。要释放额外容量并释放尽可能多的内存，请调用squeeze() 在最后一次更改字符串大小之后。

也可以看看indexOf(),lastIndexOf（），和replace()。

[since 6.5] QString &QString::removeAt(qsizetype pos)

删除索引处的字符*pos*。如果*pos*超出范围（即*pos*>=size()），这个函数什么也不做。

该功能是在 Qt 6.5 中引入的。

也可以看看remove()。

[since 6.5] QString &QString::removeFirst()

删除该字符串中的第一个字符。如果字符串为空，则该函数不执行任何操作。

该功能是在 Qt 6.5 中引入的。

也可以看看remove()。

[since 6.1] template <typename Predicate> QString &QString::removeIf(Predicate pred)

删除谓词所属的所有元素*pred*从字符串中返回 true。返回对字符串的引用。

该功能是在 Qt 6.1 中引入的。

也可以看看remove()。

[since 6.5] QString &QString::removeLast()

删除该字符串中的最后一个字符。如果字符串为空，则该函数不执行任何操作。

该功能是在 Qt 6.5 中引入的。

也可以看看remove()。

QString::reverse_iterator QString::rend()

返回一个STL-style反向迭代器以相反的顺序指向字符串中最后一个字符之后。

警告：返回的迭代器在分离或当QString被修改。

也可以看看end(),crend（），和rbegin()。

QString::const_reverse_iterator QString::rend() const

这是一个过载功能。

QString QString::repeated(qsizetype times) const

返回此字符串重复指定次数的副本*times*。

如果*times*小于 1，则返回空字符串。

例子：

```
QString str("ab");
str.repeated(4);            // returns "abababab"
```

QString &QString::replace(qsizetype position, qsizetype n, const QString &after)

取代*n*从索引开始的字符*position*用字符串*after*并返回对此字符串的引用。

注：如果指定*position*索引在字符串内，但是*position+n*超出字符串范围，然后*n*将被调整为停在字符串的末尾。

例子：

```
QString x = "Say yes!";
QString y = "no";
x.replace(4, 3, y);
// x == "Say no!"
```

也可以看看insert（） 和remove()。

QString &QString::replace(qsizetype position, qsizetype n, QChar after)

该函数重载了replace()。

取代*n*从索引开始的字符*position*与角色*after*并返回对此字符串的引用。

*QString &QString::replace(qsizetype position, qsizetype n, const QChar *unicode, qsizetype size)*

该函数重载了replace()。

取代*n*从索引开始的字符*position*与第一个*size*的字符QChar大批*unicode*并返回对此字符串的引用。

QString &QString::replace(QChar before, QChar after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了replace()。

替换每个出现的字符*before*与角色*after*并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

*QString &QString::replace(const QChar **before*, qsizetype blen, const QChar after*, qsizetype alen, Qt::CaseSensitivity cs = Qt::CaseSensitive)*

该函数重载了replace()。

替换该字符串中第一个出现的每个*blen*的字符*before*与第一个*alen*的字符*after*并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

QString &QString::replace(QLatin1StringView before, QLatin1StringView after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了replace()。

替换此字符串中查看的 Latin-1 字符串的每个出现位置*before*与 Latin-1 字符串查看*after*，并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

注意：替换后不会重新扫描文本。

QString &QString::replace(QLatin1StringView before, const QString &after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了replace()。

替换此字符串中查看的 Latin-1 字符串的每个出现位置*before*用字符串*after*，并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

注意：替换后不会重新扫描文本。

QString &QString::replace(const QString &before, QLatin1StringView after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了replace()。

替换每次出现的字符串*before*用字符串*after*并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

注意：替换后不会重新扫描文本。

QString &QString::replace(const QString &before, const QString &after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了replace()。

替换每次出现的字符串*before*用字符串*after*并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

例子：

```
QString str = "colour behaviour flavour neighbour";
str.replace(QString("ou"), QString("o"));
// str == "color behavior flavor neighbor"
```

注意：替换文本插入后不会重新扫描。

例子：

```
QString equis = "xxxxxx";
equis.replace("xx", "x");
// equis == "xxx"
```

QString &QString::replace(QChar ch, const QString &after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了replace()。

替换每个出现的字符*ch*在字符串中*after*并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

QString &QString::replace(QChar c, QLatin1StringView after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

该函数重载了replace()。

替换每个出现的字符*c*用字符串*after*并返回对此字符串的引用。

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

注意：替换后不会重新扫描文本。

QString &QString::replace(const QRegularExpression &re, const QString &after)

该函数重载了replace()。

替换每个出现的正则表达式*re*在字符串中*after*。返回对字符串的引用。例如：

```
QString s = "Banana";
s.replace(QRegularExpression("a[mn]"), "ox");
// s == "Boxoxa"
```

对于包含捕获组的正则表达式，出现**1**、**2**、 ...,*after*替换为相应捕获组捕获的字符串。

```
QString t = "A <i>bon mot</i>.";
t.replace(QRegularExpression("<i>([^<]*)</i>"), "\\emph{\\1}");
// t == "A \\emph{bon mot}."
```

也可以看看indexOf(),lastIndexOf(),remove(),QRegularExpression， 和QRegularExpressionMatch。

void QString::reserve(qsizetype size)

确保字符串至少有空间*size*人物。

如果您事先知道字符串有多大，则可以调用此函数来避免在构建字符串的过程中重复重新分配。这可以提高增量构建字符串时的性能。添加到字符串的一长串操作可能会触发多次重新分配，最后一次重新分配可能会给您留下比实际需要更多的空间，这比在开始时进行一次正确大小的分配效率要低。

如果对需要多少空间有疑问，通常最好使用上限*size*，或者最可能大小的高估计（如果严格的上限比这个大得多）。如果*size*是低估的，一旦超出保留大小，字符串将根据需要增长，这可能会导致分配的分配量大于最佳高估的分配量，并且会减慢触发它的操作。

警告： reserve() 保留内存，但不会更改字符串的大小。访问超出字符串末尾的数据是未定义的行为。如果需要访问超出字符串当前末尾的内存，请使用resize()。

此函数对于需要构建长字符串并希望避免重复重新分配的代码非常有用。在这个例子中，我们想要添加到字符串中，直到满足某个条件`true`，并且我们相当确定大小足够大，值得调用reserve()：

```
QString result;
qsizetype maxSize;
bool condition;
QChar nextChar;

result.reserve(maxSize);

while (condition)
    result.append(nextChar);

result.squeeze();
```

也可以看看squeeze(),capacity （） ， 和resize()。

void QString::resize(qsizetype size)

将字符串的大小设置为*size*人物。

如果*size*大于当前大小，则字符串被扩展以使其成为*size*字符长，并在末尾添加额外的字符。新字符未初始化。

如果*size*小于当前大小，字符超出位置*size*被排除在字符串之外。

注意： 虽然 resize() 会在需要时增加容量，但它永远不会缩小容量。要摆脱过剩产能，请使用squeeze()。

例子：

```
QString s = "Hello world";
s.resize(5);
// s == "Hello"

s.resize(8);
// s == "Hello???" (where ? stands for any character)
```

如果要向字符串附加一定数量的相同字符，请使用resize(qsizetype, QChar) 重载。

如果您想扩展字符串使其达到一定宽度并用特定字符填充新位置，请使用leftJustified （） 功能：

如果*size*为负数，相当于传递零。

```
QString r = "Hello";
r = r.leftJustified(10, ' ');
// r == "Hello     "
```

也可以看看truncate(),reserve（），和squeeze()。

void QString::resize(qsizetype newSize, QChar fillChar)

这是一个过载功能。

不像resize(qsizetype)，此重载将新字符初始化为*fillChar*:

```
QString t = "Hello";
r.resize(t.size() + 10, 'X');
// t == "HelloXXXXXXXXXX"
```

QString QString::right(qsizetype n) const

返回包含以下内容的子字符串*n*字符串最右边的字符。

如果你知道的话*n*不能越界，使用last() 改为在新代码中，因为它更快。

如果返回整个字符串*n*大于或等于size()，或小于零。

也可以看看endsWith(),last(),first(),sliced(),chopped(),chop（），和truncate()。

QString QString::rightJustified(qsizetype width, QChar fill = u' ', bool truncate = false) const

返回一个字符串size()*width*其中包含*fill*字符后跟字符串。例如：

```
QString s = "apple";
QString t = s.rightJustified(8, '.');    // t == "...apple"
```

如果*truncate*是false并且size字符串的 () 大于*width*，那么返回的字符串是该字符串的副本。

如果*truncate*是真的，并且size字符串的 () 大于*width*，然后结果字符串在位置被截断*width*。

```
QString str = "Pineapple";
str = str.rightJustified(5, '.', true);    // str == "Pinea"
```

也可以看看leftJustified()。

QString QString::section(*QChar* *sep*, *qsizetype* *start*, *qsizetype* *end* = -1, *QString::SectionFlags* *flags* = SectionDefault) const

该函数返回字符串的一部分。

该字符串被视为由字符分隔的字段序列，*sep*。返回的字符串由位置中的字段组成*start*定位*end*包括的。如果*end*未指定，所有字段来自位置*start*到字符串末尾都包含在内。字段从左开始编号为 0、1、2 等，从右到左编号为 -1、-2 等。

这*flags*参数可用于影响函数行为的某些方面，例如是否区分大小写、是否跳过空字段以及如何处理前导和尾随分隔符；看SectionFlags。

```
QString str;
QString csv = "forename,middlename,surname,phone";
QString path = "/usr/local/bin/myapp"; // First field is empty
QString::SectionFlag flag = QString::SectionSkipEmpty;

str = csv.section(',', 2, 2);   // str == "surname"
str = path.section('/', 3, 4);  // str == "bin/myapp"
str = path.section('/', 3, 3, flag); // str == "myapp"
```

如果*start*或者*end*为负数，我们从字符串右侧开始计数字段，最右边的字段为 -1，最右边的字段为 -2，依此类推。

```
str = csv.section(',', -3, -2);  // str == "middlename,surname"
str = path.section('/', -1); // str == "myapp"
```

也可以看看split()。

QString QString::section(const *QString* &*sep*, *qsizetype* *start*, *qsizetype* *end* = -1, *QString::SectionFlags* *flags* = SectionDefault) const

该函数重载了section()。

```
QString str;
QString data = "forename**middlename**surname**phone";

str = data.section("**", 2, 2); // str == "surname"
str = data.section("**", -3, -2); // str == "middlename**surname"
```

也可以看看split()。

QString QString::section(const QRegularExpression &re, qsizetype start, qsizetype end = -1, QString::SectionFlags flags = SectionDefault) const

该函数重载了section()。

该字符串被视为由正则表达式分隔的字段序列，*re*。

```
QString line = "forename\tmiddlename  surname \t \t phone";
QRegularExpression sep("\\s+");
str = line.section(sep, 2, 2); // str == "surname"
str = line.section(sep, -3, -2); // str == "middlename  surname"
```

警告：使用这个QRegularExpression版本比重载的字符串和字符版本昂贵得多。

也可以看看split（） 和simplified()。

QString &QString::setNum(int n, int base = 10)

将字符串设置为打印值*n*在指定的*base*，并返回对该字符串的引用。

默认基数为 10，并且必须介于 2 到 36 之间。

```
QString str;
str.setNum(1234);       // str == "1234"
```

格式始终使用QLocale::C，即英语/美国。要获取数字的本地化字符串表示形式，请使用QLocale::toString() 与适当的区域设置。

也可以看看number()。

QString &QString::setNum(short n, int base = 10)

这是一个过载功能。

QString &QString::setNum(ushort n, int base = 10)

这是一个过载功能。

QString &QString::setNum(uint n, int base = 10)

这是一个过载功能。

QString &QString::setNum(long n, int base = 10)

这是一个过载功能。

QString &QString::setNum(ulong n, int base = 10)

这是一个过载功能。

QString &QString::setNum(qlonglong n, int base = 10)

这是一个过载功能。

QString &QString::setNum(qulonglong n, int base = 10)

这是一个过载功能。

QString &QString::setNum(float n, char format = 'g', int precision = 6)

这是一个过载功能。

将字符串设置为打印值*n*，根据给定的格式*format*和*precision*，并返回对该字符串的引用。

格式始终使用QLocale::C，即英语/美国。要获取数字的本地化字符串表示形式，请使用QLocale::toString() 与适当的区域设置。

也可以看看number()。

QString &QString::setNum(double n, char format = 'g', int precision = 6)

这是一个过载功能。

将字符串设置为打印值*n*，根据给定的格式*format*和*precision*，并返回对该字符串的引用。

也可以看看number(),QLocale::FloatingPointPrecisionOption， 和Number Formats。

*QString &QString::setRawData(const QChar *unicode, qsizetype size)*

重置QString使用第一个*size*数组中的 Unicode 字符*unicode*。数据在*unicode*不被复制。调用者必须能够保证*unicode*只要QString（或其未经修改的副本）存在。

该函数可以用来代替fromRawData() 重用现有的QString对象来保存内存重新分配。

也可以看看fromRawData()。

*QString &QString::setUnicode(const QChar *unicode, qsizetype size)*

将字符串大小调整为*size*字符和副本*unicode*到字符串中。

如果*unicode*是`nullptr`，没有复制任何内容，但字符串的大小仍然调整为*size*。

也可以看看unicode（） 和setUtf16()。

*QString &QString::setUtf16(const ushort *unicode, qsizetype size)*

将字符串大小调整为*size*字符和副本*unicode*到字符串中。

如果*unicode*是`nullptr`，没有复制任何内容，但字符串的大小仍然调整为*size*。

请注意，与fromUtf16()，此函数不考虑 BOM 和可能不同的字节顺序。

也可以看看utf16（） 和setUnicode()。

void QString::shrink_to_fit()

提供此函数是为了兼容 STL。它相当于squeeze()。

也可以看看squeeze()。

QString QString::simplified() const

返回一个字符串，该字符串从开头和结尾删除了空格，并将每个内部空格序列替换为单个空格。

空白是指任何字符QChar::isSpace() 返回`true`. 这包括 ASCII 字符 '\t'、'\n'、'\v'、'\f'、'\r' 和 ' '。

例子：

```
QString str = "  lots\t of\nwhitespace\r\n ";
str = str.simplified();
// str == "lots of whitespace";
```

也可以看看trimmed()。

qsizetype QString::size() const

返回该字符串中的字符数。

字符串中的最后一个字符位于位置 size() - 1。

例子：

```
QString str = "World";
qsizetype n = str.size();   // n == 5
str.data()[0];              // returns 'W'
str.data()[4];              // returns 'd'
```

也可以看看isEmpty（） 和resize()。

[since 6.0]QString QString::sliced(qsizetype pos, qsizetype n) const

返回一个字符串，其中包含*n*该字符串的字符，从位置开始*pos*。

注意：当以下情况时，行为未定义*pos*< 0,*n*< 0，或*pos*+*n*>size()。

```
QString x = "Nine pineapples";
QString y = x.sliced(5, 4);           // y == "pine"
QString z = x.sliced(5);              // z == "pineapples"
```

这个函数是在Qt 6.0中引入的。

也可以看看first(),last(),chopped(),chop（），和truncate()。

[since 6.0]QString QString::sliced(qsizetype pos) const

这是一个过载功能。

返回一个字符串，其中包含该字符串从位置开始的部分*pos*并延伸至其末端。

注意：当以下情况时，行为未定义*pos*< 0 或*pos*>size()。

这个函数是在Qt 6.0中引入的。

也可以看看first(),last(),sliced(),chopped(),chop（），和truncate()。

QStringList QString::split(const QString &sep, Qt::SplitBehavior behavior = Qt::KeepEmptyParts, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

将字符串拆分为任意位置的子字符串*sep*发生，并返回这些字符串的列表。如果*sep*与字符串中的任何位置都不匹配，split() 返回包含该字符串的单元素列表。

*cs*指定是否*sep*应区分大小写或不区分大小写进行匹配。

如果*behavior*是Qt::SkipEmptyParts，空条目不会出现在结果中。默认情况下，保留空条目。

例子：

```
QString str = QStringLiteral("a,,b,c");

QStringList list1 = str.split(u',');
// list1: [ "a", "", "b", "c" ]

QStringList list2 = str.split(u',', Qt::SkipEmptyParts);
// list2: [ "a", "b", "c" ]
```

如果*sep*为空， split() 返回一个空字符串，后跟该字符串的每个字符，最后跟另一个空字符串：

```
QString str = "abc";
auto parts = str.split(QString());
// parts: {"", "a", "b", "c", ""}
```

要理解此行为，请回想一下空字符串在任何地方都匹配，因此上面的内容本质上与以下内容相同：

```
QString str = "/a/b/c/";
auto parts = str.split(u'/');
// parts: {"", "a", "b", "c", ""}
```

也可以看看QStringList::join（） 和section()。

QStringList QString::split(QChar sep, Qt::SplitBehavior behavior = Qt::KeepEmptyParts, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

这是一个过载功能。

QStringList QString::split(const QRegularExpression &re, Qt::SplitBehavior behavior = Qt::KeepEmptyParts) const

这是一个过载功能。

将字符串拆分为正则表达式所在的子字符串*re*匹配，并返回这些字符串的列表。如果*re*与字符串中的任何位置都不匹配，split() 返回包含该字符串的单元素列表。

下面是一个示例，我们使用一个或多个空白字符作为分隔符来提取句子中的单词：

```
QString str;
QStringList list;

str = "Some  text\n\twith  strange whitespace.";
list = str.split(QRegularExpression("\\s+"));
// list: [ "Some", "text", "with", "strange", "whitespace." ]
```

这是一个类似的示例，但这次我们使用任何非单词字符序列作为分隔符：

```
str = "This time, a normal English sentence.";
list = str.split(QRegularExpression("\\W+"), Qt::SkipEmptyParts);
// list: [ "This", "time", "a", "normal", "English", "sentence" ]
```

这是第三个示例，其中我们使用零长度断言\b（单词边界）将字符串拆分为非单词和单词标记的交替序列：

```
str = "Now: this sentence fragment.";
list = str.split(QRegularExpression("\\b"));
// list: [ "", "Now", ": ", "this", " ", "sentence", " ", "fragment", "." ]
```

也可以看看QStringList::join（） 和section()。

void QString::squeeze()

释放存储字符数据不需要的任何内存。

该函数的唯一目的是提供一种微调的手段QString的内存使用情况。一般来说，您很少需要调用此函数。

也可以看看reserve（） 和capacity()。

bool QString::startsWith(const QString &s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

`true`如果字符串开头则返回*s*; 否则返回`false`.

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

```
QString str = "Bananas";
str.startsWith("Ban");     // returns true
str.startsWith("Car");     // returns false
```

也可以看看endsWith()。

bool QString::startsWith(QStringView str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

这是一个过载功能。

返回`true`字符串是否以字符串视图开头*str*; 否则返回`false`.

如果*cs*是Qt::CaseSensitive（默认），搜索区分大小写；否则搜索不区分大小写。

也可以看看endsWith()。

bool QString::startsWith(QLatin1StringView s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载startsWith()。

bool QString::startsWith(QChar c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

该函数重载startsWith()。

`true`如果字符串开头则返回*c*; 否则返回`false`.

void QString::swap(QString &other)

交换字符串*other*用这个字符串。这个操作非常快并且永远不会失败。

CFStringRef QString::toCFString() const

从 a 创建一个 CFStringQString。

调用者拥有 CFString 并负责释放它。

注意：此功能仅在 macOS 和 iOS 上可用。

QString QString::toCaseFolded() const

返回字符串的大小写折叠等效项。对于大多数 Unicode 字符来说，这与toLower()。

*double QString::toDouble(bool *ok = nullptr) const*

返回转换为值的字符串double。

如果转换溢出，则返回无穷大；如果转换因其他原因（例如下溢）失败，则返回 0.0。

如果*ok*不是nullptr，通过设置*报告失败*ok*到*false*，并通过设置成功* ***ok*到true.

```
QString str = "1234.56";
double val = str.toDouble();   // val == 1234.56
```

警告：QString内容只能包含有效的数字字符，其中包括加号/减号、科学记数法中使用的字符 e 和小数点。包含单位或附加字符会导致转换错误。

```
bool ok;
double d;

d = QString( "1234.56e-02" ).toDouble(&ok); // ok == true, d == 12.3456

d = QString( "1234.56e-02 Volt" ).toDouble(&ok); // ok == false, d == 0
```

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toDouble()

```
d = QString( "1234,56" ).toDouble(&ok); // ok == false
d = QString( "1234.56" ).toDouble(&ok); // ok == true, d == 1234.56
```

由于历史原因，此函数不处理千组分隔符。如果您需要转换此类数字，请使用QLocale::toDouble()。

```
d = QString( "1,234,567.89" ).toDouble(&ok); // ok == false
d = QString( "1234567.89" ).toDouble(&ok); // ok == true
```

此函数忽略前导和尾随空格。

也可以看看number(),QLocale::setDefault(),QLocale::toDouble（），和trimmed()。

*float QString::toFloat(bool *ok = nullptr) const*

返回转换为值的字符串float。

如果转换溢出，则返回无穷大；如果转换因其他原因（例如下溢）失败，则返回 0.0。

如果*ok*不是nullptr，通过设置*报告失败*ok*到*false*，并通过设置成功* ***ok*到true.

警告：QString内容只能包含有效的数字字符，其中包括加号/减号、科学记数法中使用的字符 e 和小数点。包含单位或附加字符会导致转换错误。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toFloat()

由于历史原因，此函数不处理千组分隔符。如果您需要转换此类数字，请使用QLocale::toFloat()。

例子：

```
QString str1 = "1234.56";
str1.toFloat();             // returns 1234.56

bool ok;
QString str2 = "R2D2";
str2.toFloat(&ok);          // returns 0.0, sets ok to false

QString str3 = "1234.56 Volt";
str3.toFloat(&ok);          // returns 0.0, sets ok to false
```

此函数忽略前导和尾随空格。

也可以看看number(),toDouble(),toInt(),QLocale::toFloat（），和trimmed()。

QString QString::toHtmlEscaped() const

将纯文本字符串转换为带有 HTML 元字符<、>、&、 并"替换为 HTML 实体的 HTML 字符串。

例子：

```
QString plain = "#include <QtCore>"
QString html = plain.toHtmlEscaped();
// html == "#include &lt;QtCore&gt;"
```

*int QString::toInt(bool *ok = nullptr, int base = 10) const*

返回转换为int使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是nullptr，通过设置*报告失败*ok*到*false*，并通过设置成功* ***ok*到true.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toInt()

例子：

```
QString str = "FF";
bool ok;
int hex = str.toInt(&ok, 16);       // hex == 255, ok == true
int dec = str.toInt(&ok, 10);       // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number(),toUInt(),toDouble（），和QLocale::toInt()。

QByteArray QString::toLatin1() const

返回字符串的 Latin-1 表示形式QByteArray。

如果字符串包含非 Latin1 字符，则返回的字节数组未定义。这些字符可能会被隐藏或替换为问号。

也可以看看fromLatin1(),toUtf8(),toLocal8Bit（），和QStringEncoder。

QByteArray QString::toLocal8Bit() const

返回字符串的本地 8 位表示形式QByteArray。

在 Unix 系统上这相当于toUtf8()，在 Windows 上，系统正在使用当前代码页。

如果此字符串包含任何无法以本地 8 位编码进行编码的字符，则返回的字节数组未定义。这些字符可能会被抑制或被另一个字符取代。

也可以看看fromLocal8Bit(),toLatin1(),toUtf8（），和QStringEncoder。

*long QString::toLong(bool *ok = nullptr, int base = 10) const*

返回转换为long使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是nullptr，通过设置*报告失败*ok*到*false*，并通过设置成功* ***ok*到true.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toLongLong()

例子：

```
QString str = "FF";
bool ok;

long hex = str.toLong(&ok, 16);     // hex == 255, ok == true
long dec = str.toLong(&ok, 10);     // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number(),toULong(),toInt（），和QLocale::toInt()。

qlonglong QString::toLongLong(bool *ok = nullptr, int base = 10) const

返回转换为`long long`使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是`nullptr`，通过设置*报告失败*ok*到*`false`*，并通过设置成功* ***ok*到`true`.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toLongLong()

例子：

```
QString str = "FF";
bool ok;

qint64 hex = str.toLongLong(&ok, 16);      // hex == 255, ok == true
qint64 dec = str.toLongLong(&ok, 10);      // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number(),toULongLong(),toInt（），和QLocale::toLongLong()。

QString QString::toLower() const

返回字符串的小写副本。

```
QString str = "The Qt PROJECT";
str = str.toLower();        // str == "the qt project"
```

大小写转换始终发生在“C”语言环境中。用于依赖于区域设置的大小写折叠使用QLocale::toLower()

也可以看看toUpper（）和QLocale::toLower()。

*NSString *QString::toNSString() const*

从 a 创建一个 NSStringQString。

NSString 是自动释放的。

注意：此功能仅在 macOS 和 iOS 上可用。

*short QString::toShort(bool *ok = nullptr, int base = 10) const*

返回转换为short使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是nullptr，通过设置*报告失败*ok*到false，并通过设置成功* ***ok*到true.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toShort()

例子：

```
QString str = "FF";
bool ok;

short hex = str.toShort(&ok, 16);   // hex == 255, ok == true
short dec = str.toShort(&ok, 10);   // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number(),toUShort(),toInt（），和QLocale::toShort()。

std::string QString::toStdString() const

返回一个 std::string 对象，其中包含此对象中包含的数据QString。使用以下命令将 Unicode 数据转换为 8 位字符toUtf8（） 功能。

此方法主要用于传递QString到接受 std::string 对象的函数。

也可以看看toLatin1(),toUtf8(),toLocal8Bit（），和QByteArray::toStdString()。

std::u16string QString::toStdU16String() const

返回一个 std::u16string 对象，其中包含该对象的数据QString。Unicode 数据与返回的相同utf16（） 方法。

也可以看看utf16(),toStdWString（），和toStdU32String()。

std::u32string QString::toStdU32String() const

返回一个 std::u32string 对象，其中包含其中的数据QString。Unicode 数据与返回的相同toUcs4（） 方法。

也可以看看toUcs4(),toStdWString（），和toStdU16String()。

std::wstring QString::toStdWString() const

返回一个 std::wstring 对象，其中包含此对象中的数据QString。std::wstring 在 wchar_t 为 2 字节宽的平台（例如 windows）上以 utf16 编码，在 wchar_t 为 4 字节宽的平台（大多数 Unix 系统）上以 ucs4 编码。

此方法主要用于传递QString到接受 std::wstring 对象的函数。

也可以看看utf16(),toLatin1(),toUtf8(),toLocal8Bit(),toStdU16String（），和toStdU32String()。

*uint QString::toUInt(bool *ok = nullptr, int base = 10) const*

返回转换为`unsigned int`使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是`nullptr`，通过设置*报告失败*ok*到`false`，并通过设置成功* **ok*到`true`.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toUInt()

例子：

```
QString str = "FF";
bool ok;

uint hex = str.toUInt(&ok, 16);     // hex == 255, ok == true
uint dec = str.toUInt(&ok, 10);     // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number(),toInt（），和QLocale::toUInt()。

*ulong QString::toULong(bool *ok = nullptr, int base = 10) const*

返回转换为`unsigned long`使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是`nullptr`，通过设置*报告失败*ok*到`false`，并通过设置成功* **ok*到`true`.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toULongLong()

例子：

```
QString str = "FF";
bool ok;

ulong hex = str.toULong(&ok, 16);   // hex == 255, ok == true
ulong dec = str.toULong(&ok, 10);   // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number（） 和QLocale::toUInt()。

qulonglong QString::toULongLong(bool *ok = nullptr, int base = 10) const

返回转换为unsigned long long使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是nullptr，通过设置*报告失败*ok*到*false，*并通过设置成功* **ok*到true.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toULongLong()

例子：

```
QString str = "FF";
bool ok;

quint64 hex = str.toULongLong(&ok, 16);    // hex == 255, ok == true
quint64 dec = str.toULongLong(&ok, 10);    // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number(),toLongLong（），和QLocale::toULongLong()。

ushort QString::toUShort(bool *ok = nullptr, int base = 10) const

返回转换为unsigned short使用基数的字符串*base*，默认为 10，并且必须介于 2 到 36 或 0 之间。如果转换失败，则返回 0。

如果*ok*不是nullptr，通过设置*报告失败*ok*到*false，*并通过设置成功* **ok*到true.

如果*base*为0时，使用C语言约定：如果字符串以“0x”开头，则使用基数16；否则，如果字符串以“0b”开头，则使用基数 2；否则，如果字符串以“0”开头，则使用基数 8；否则，使用基数 10。

字符串转换始终发生在“C”语言环境中。对于依赖于语言环境的转换使用QLocale::toUShort()

例子：

```
QString str = "FF";
bool ok;

ushort hex = str.toUShort(&ok, 16);     // hex == 255, ok == true
ushort dec = str.toUShort(&ok, 10);     // dec == 0, ok == false
```

此函数忽略前导和尾随空格。

注意： Qt 6.4 中添加了对“0b”前缀的支持。

也可以看看number(),toShort（），和QLocale::toUShort()。

QList[uint](https://doc-qt-io.translate.goog/qt-6/qttypes.html?_x_tr_sl=auto&_x_tr_tl=zh-CN&_x_tr_hl=zh-CN&_x_tr_pto=wapp#uint-typedef) QString::toUcs4() const

返回字符串的 UCS-4/UTF-32 表示形式QList。

UCS-4 是一种 Unicode 编解码器，因此它是无损的。该字符串中的所有字符都将采用 UCS-4 进行编码。此字符串中任何无效的代码单元序列都将替换为 Unicode 的替换字符 (QChar::ReplacementCharacter，对应于U+FFFD）。

返回的列表不是 \0' 终止的。

也可以看看fromUtf8(),toUtf8(),toLatin1(),toLocal8Bit(),QStringEncoder,fromUcs4（），和toWCharArray()。

QString QString::toUpper() const

返回字符串的大写副本。

```
QString str = "TeXt";
str = str.toUpper();        // str == "TEXT"
```

大小写转换始终发生在“C”语言环境中。用于依赖于区域设置的大小写折叠使用QLocale::toUpper()

也可以看看toLower（）和QLocale::toLower()。

QByteArray QString::toUtf8() const

返回字符串的 UTF-8 表示形式QByteArray。

UTF-8 是一种 Unicode 编解码器，可以表示 Unicode 字符串中的所有字符，例如QString。

也可以看看fromUtf8(),toLatin1(),toLocal8Bit（），和QStringEncoder。

*qsizetype QString::toWCharArray(wchar_t *array) const*

填充*array*与此中包含的数据QString目的。该数组在 wchar_t 为 2 字节宽的平台（例如 Windows）上以 UTF-16 编码，在 wchar_t 为 4 字节宽的平台（大多数 Unix 系统）上以 UCS-4 编码。

*array*必须由调用者分配并包含足够的空间来容纳完整的字符串（分配与字符串长度相同的数组总是足够的）。

该函数返回字符串的实际长度*array*。

注意：此函数不会将空字符附加到数组中。

也可以看看utf16(),toUcs4(),toLatin1(),toUtf8(),toLocal8Bit(),toStdWString（），和QStringView::toWCharArray()。

QString QString::trimmed() const

返回从开头和结尾删除空格的字符串。

空白是指任何字符QChar::isSpace() 返回`true`. 这包括 ASCII 字符 '\t'、'\n'、'\v'、'\f'、'\r' 和 ' '。

例子：

```
QString str = "  lots\t of\nwhitespace\r\n ";
str = str.trimmed();
// str == "lots\t of\nwhitespace"
```

不像simplified()、trimmed() 单独保留内部空白。

也可以看看simplified()。

void QString::truncate(qsizetype position)

在给定的位置截断字符串*position*指数。

如果指定*position*索引超出了字符串末尾，没有任何反应。

例子：

```
QString str = "Vladivostok";
str.truncate(4);
// str == "Vlad"
```

如果*position*为负数，相当于传递零。

也可以看看chop(),resize(),first（），和QStringView::truncate()。

*const QChar *QString::unicode() const*

返回字符串的 Unicode 表示形式。在字符串被修改之前，结果一直有效。

注意：返回的字符串不能以“\0”结尾。使用size() 来确定数组的长度。

也可以看看setUnicode(),utf16（），和fromRawData()。

*const ushort *QString::utf16() const*

返回QString作为以 '\0' 结尾的无符号短裤数组。在字符串被修改之前，结果一直有效。

返回的字符串按主机字节顺序排列。

也可以看看setUtf16（） 和unicode()。

*[static]QString QString::vasprintf(const char *cformat, va_list ap)*

等效方法asprintf()，但需要一个 va_list*ap*相反，变量参数列表。请参阅asprintf() 解释文档*cformat*。

该方法不调用va_end宏，调用者负责调用va_end*ap*。

也可以看看asprintf()。

*bool QString::operator!=(const char *other) const*

该函数重载了operator!=()。

这*other*const char 指针转换为QString使用fromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

bool QString::operator!=(const QByteArray &other) const

该函数重载了operator!=()。

这*other*字节数组转换为QString使用fromUtf8（） 功能。如果任何 NUL 字符（'\0'）嵌入到字节数组中，它们将包含在转换中。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QString &QString::operator+=(const QString &other)

追加字符串*other*到该字符串的末尾并返回对此字符串的引用。

例子：

```
QString x = "free";
QString y = "dom";
x += y;
// x == "freedom"
```

此操作通常非常快（constant time），因为QString在字符串数据的末尾预先分配额外的空间，以便它可以增长，而无需每次重新分配整个字符串。

也可以看看append（）和prepend()。

QString &QString::operator+=(QChar ch)

该函数重载了operator+=()。

追加字符*ch*到字符串。

[since 6.0]QString &QString::operator+=(QStringView str)

该函数重载了operator+=()。

追加字符串视图*str*到这个字符串。

这个函数是在Qt 6.0中引入的。

QString &QString::operator+=(QLatin1StringView str)

该函数重载了operator+=()。

附加通过查看的 Latin-1 字符串*str*到这个字符串。

[since 6.5]QString &QString::operator+=(QUtf8StringView str)

该函数重载了operator+=()。

追加 UTF-8 字符串视图*str*到这个字符串。

该功能是在 Qt 6.5 中引入的。

*QString &QString::operator+=(const char *str)*

该函数重载了operator+=()。

追加字符串*str*到这个字符串。const char 指针使用以下命令转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用此功能QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QString &QString::operator+=(const QByteArray &ba)

该函数重载了operator+=()。

追加字节数组*ba*到这个字符串。使用以下命令将字节数组转换为 UnicodefromUtf8（） 功能。如果任何 NUL 字符（'\0'）嵌入在*ba*字节数组，它们将包含在转换中。

您可以通过定义禁用此功能QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

*bool QString::operator<(const char *other) const*

`true`如果该字符串在词法上小于 string，则返回*other*。否则返回`false`。

该函数重载了operator<()。

这*other*const char 指针转换为QString使用fromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

bool QString::operator<(const QByteArray &other) const

该函数重载了operator<()。

这*other*字节数组转换为QString使用fromUtf8（） 功能。如果任何 NUL 字符（'\0'）嵌入到字节数组中，它们将包含在转换中。

您可以禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

*bool QString::operator<=(const char *other) const*

该函数重载了operator<=()。

这*other*const char 指针转换为QString使用fromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

bool QString::operator<=(const QByteArray &other) const

该函数重载了operator<=()。

这*other*字节数组转换为QString使用fromUtf8（） 功能。如果任何 NUL 字符（'\0'）嵌入到字节数组中，它们将包含在转换中。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QString &QString::operator=(const QString &other)

分配*other*到此字符串并返回对此字符串的引用。

QString &QString::operator=(QChar ch)

该函数重载了operator=()。

将字符串设置为包含单个字符*ch*。

QString &QString::operator=(QLatin1StringView str)

该函数重载了operator=()。

分配 Latin-1 字符串查看者*str*到这个字符串。

QString &QString::operator=(QString &&other)

移动分配*other*对此QString实例。

*QString &QString::operator=(const char *str)*

该函数重载了operator=()。

分配*str*到这个字符串。const char 指针使用以下命令转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII或者QT_RESTRICTED_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QString &QString::operator=(const QByteArray &ba)

该函数重载了operator=()。

分配*ba*到这个字符串。使用以下命令将字节数组转换为 UnicodefromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

*bool QString::operator==(const char *other) const*

该函数重载了operator==()。

这*other*const char 指针转换为QString使用fromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

bool QString::operator==(const QByteArray &other) const

该函数重载了operator==()。

这*other*字节数组转换为QString使用fromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

返回`true`此字符串在词法上是否等于参数字符串*other*。否则返回`false`。

*bool QString::operator>(const char *other) const*

该函数重载了operator>()。

这*other*const char 指针转换为QString使用fromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

bool QString::operator>(const QByteArray &other) const

该函数重载了operator>()。

这*other*字节数组转换为QString使用fromUtf8（） 功能。如果任何 NUL 字符（'\0'）嵌入到字节数组中，它们将包含在转换中。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

*bool QString::operator>=(const char *other) const*

该函数重载了operator>=()。

这*other*const char 指针转换为QString使用fromUtf8（） 功能。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

bool QString::operator>=(const QByteArray &other) const

该函数重载了operator>=()。

这*other*字节数组转换为QString使用fromUtf8（） 功能。如果任何 NUL 字符（'\0'）嵌入到字节数组中，它们将包含在转换中。

您可以通过定义禁用该运算符QT_NO_CAST_FROM_ASCII当您编译应用程序时。如果您想确保所有用户可见的字符串都通过，这会很有用QObject::tr（），例如。

QChar &QString::operator

返回指定位置的字符*position*在字符串中作为可修改的引用。

例子：

```
QString str;

if (str[0] == QChar('?'))
    str[0] = QChar('_');
```

也可以看看at()。

const QChar QString::operator const

该函数重载了operator。

相关非会员

QString operator+(QString &&s1, const QString &s2)

QString operator+(const QString &s1, const QString &s2)

返回一个字符串，该字符串是连接的结果*s1*和*s2*。

[since 6.1]template qsizetype erase(QString &s, const T &t)

删除所有比较等于的元素*t*从字符串中*s*。返回删除的元素数（如果有）。

该功能是在 Qt 6.1 中引入的。

也可以看看erase_if。

[since 6.1]template qsizetype erase_if(QString &s, Predicate pred)

删除谓词所属的所有元素*pred*从字符串中返回 true*s*。返回删除的元素数（如果有）。

该功能是在 Qt 6.1 中引入的。

也可以看看erase。

bool operator!=(const QString &s1, const QString &s2)

返回`true`if 字符串*s1*不等于字符串*s2*; 否则返回`false`.

也可以看看Comparing Strings。

bool operator!=(const QString &s1, QLatin1StringView s2)

返回trueif 字符串*s1*不等于字符串*s2*。否则返回false。

该函数重载了operator!=()。

*bool operator!=(const char *s1, const QString &s2)*

返回true如果*s1*不等于*s2*; 否则返回false.

为了*s1*!= 0，这相当于compare(*s1*,*s2*) != 0。请注意，没有字符串等于*s1*为 0。

*[since 6.4]QString operator""_s(const char16_t *str, size_t size)*

创建一个的文字运算符QString从第一个*size*char16_t 字符串文字中的字符*str*。

这QString在编译时创建，生成的字符串数据存储在编译后的目标文件的只读段中。重复的文字可能共享相同的只读存储器。此功能可以与以下功能互换QStringLiteral，但是当代码中存在许多字符串文字时可以节省输入。

以下代码创建一个QString:

```
using namespace Qt::Literals::StringLiterals;

auto str = u"hello"_s;
```

该功能是在 Qt 6.4 中引入的。

也可以看看Qt::Literals::StringLiterals。

*QString operator+(const QString &s1, const char *s2)*

返回一个字符串，该字符串是连接的结果*s1*和*s2*（*s2*使用以下命令转换为 UnicodeQString::fromUtf8（） 功能）。

也可以看看QString::fromUtf8()。

*QString operator+(const char *s1, const QString &s2)*

返回一个字符串，该字符串是连接的结果*s1*和*s2*（*s1*使用以下命令转换为 UnicodeQString::fromUtf8（） 功能）。

也可以看看QString::fromUtf8()。

bool operator<(const QString &s1, const QString &s2)

该函数重载了operator<()。

返回`true`if 字符串*s1*在词法上小于字符串*s2*; 否则返回`false`.

也可以看看Comparing Strings。

bool operator<(const QString &s1, QLatin1StringView s2)

该函数重载了operator<()。

返回`true`如果*s1*从词法上来说小于*s2*; 否则返回`false`.

bool operator<(QLatin1StringView s1, const QString &s2)

该函数重载了operator<()。

返回`true`如果*s1*从词法上来说小于*s2*; 否则返回`false`.

*bool operator<(const char *s1, const QString &s2)*

返回`true`如果*s1*从词法上来说小于*s2*; 否则返回`false`. 为了*s1*!= 0，这相当于`compare(s1, s2) < 0`.

也可以看看Comparing Strings。

QDataStream &operator<<(QDataStream &stream, const QString &string)

写出给定的*string*到指定的*stream*。

也可以看看Serializing Qt Data Types。

bool operator<=(const QString &s1, const QString &s2)

返回`true`if 字符串*s1*在词法上小于或等于字符串*s2*; 否则返回`false`.

也可以看看Comparing Strings。

bool operator<=(const QString &s1, QLatin1StringView s2)

该函数重载了operator<=()。

返回`true`如果*s1*在词法上小于或等于*s2*; 否则返回`false`.

bool operator<=(QLatin1StringView s1, const QString &s2)

该函数重载了operator<=()。

返回`true`如果*s1*在词法上小于或等于*s2*; 否则返回`false`.

*bool operator<=(const char *s1, const QString &s2)*

返回`true`如果*s1*在词法上小于或等于*s2*; 否则返回`false`. 为了*s1*!= 0，这相当于`compare(s1, s2) <= 0`.

也可以看看Comparing Strings。

bool operator==(const QString &s1, const QString &s2)

该函数重载了operator==()。

返回`true`if 字符串*s1*等于字符串*s2*; 否则返回`false`.

注意：该函数将空字符串视为空字符串，有关更多详细信息，请参见Distinction Between Null and Empty Strings。

也可以看看Comparing Strings。

bool operator==(const QString &s1, QLatin1StringView s2)

该函数重载了operator==()。

返回`true`如果*s1*等于*s2*; 否则返回`false`.

bool operator==(QLatin1StringView s1, const QString &s2)

该函数重载了operator==()。

返回`true`如果*s1*等于*s2*; 否则返回`false`.

*bool operator==(const char *s1, const QString &s2)*

该函数重载了operator==()。

返回true如果*s1*等于*s2*; 否则返回false. 请注意，没有字符串等于*s1*为 0。

相当于`s1 != 0 && compare(s1, s2) == 0`.

bool operator>(const QString &s1, const QString &s2)

返回trueif 字符串*s1*在词法上大于字符串*s2*; 否则返回false.

也可以看看Comparing Strings。

bool operator>(const QString &s1, QLatin1StringView s2)

该函数重载了operator>()。

返回true如果*s1*在词法上大于*s2*; 否则返回false.

bool operator>(QLatin1StringView s1, const QString &s2)

该函数重载了operator>()。

返回true如果*s1*在词法上大于*s2*; 否则返回false.

*bool operator>(const char *s1, const QString &s2)*

返回true如果*s1*在词法上大于*s2*; 否则返回false. 相当于`compare(s1, s2) > 0`.

也可以看看Comparing Strings。

bool operator>=(const QString &s1, const QString &s2)

返回trueif 字符串*s1*在词法上大于或等于字符串*s2*; 否则返回false.

也可以看看Comparing Strings。

bool operator>=(const QString &s1, QLatin1StringView s2)

该函数重载了operator>=()。

返回`true`如果*s1*在词法上大于或等于*s2*; 否则返回`false`.

bool operator>=(QLatin1StringView s1, const QString &s2)

该函数重载了operator>=()。

返回`true`如果*s1*在词法上大于或等于*s2*; 否则返回`false`.

*bool operator>=(const char *s1, const QString &s2)*

返回`true`如果*s1*在词法上大于或等于*s2*; 否则返回`false`. 为了*s1*!= 0，这相当于`compare(s1, s2) >= 0`.

也可以看看Comparing Strings。

QDataStream &operator>>(QDataStream &stream, QString &string)

从指定的位置读取一个字符串*stream*进入给定的*string*。

也可以看看Serializing Qt Data Types。

宏文档

QStringLiteral(str)

该宏生成数据QString在字符串文字之外*str*在编译时。创建一个QString在这种情况下，from 是免费的，并且生成的字符串数据存储在编译后的目标文件的只读段中。

如果您的代码如下所示：

```
// hasAttribute takes a QString argument
if (node.hasAttribute("http-contents-length")) //...
```

然后是临时的QString将被创建为作为`hasAttribute`函数参数传递。这可能非常昂贵，因为它涉及内存分配以及将数据复制/转换为QString的内部编码。

通过使用 QStringLiteral 可以避免这种成本：

```
if (node.hasAttribute(QStringLiteral(u"http-contents-length"))) //...
```

在这种情况下，QString的内部数据会在编译时生成；运行时不会发生任何转换或分配。

使用 QStringLiteral 代替双引号纯 C++ 字符串文字可以显着加快创建QString来自编译时已知数据的实例。

笔记：QLatin1StringView当字符串传递给具有重载的函数时，仍然比 QStringLiteral 更有效QLatin1StringView并且这种重载避免了转换为QString。例如，QString::operator==() 可以与QLatin1StringView直接地：

```
if (attribute.name() == "http-contents-length"_L1) //...
```

注意：某些编译器在编码包含 US-ASCII 字符集之外的字符的字符串时存在错误。在这些情况下，请确保为字符串添加前缀u。否则它是可选的。

也可以看看QByteArrayLiteral。

QT_NO_CAST_FROM_ASCII

禁用从 8 位字符串（`char *`）到 Unicode QString 的自动转换，以及从 8 位`char`类型（`char`和`unsigned char`）到QChar。

也可以看看QT_NO_CAST_TO_ASCII,QT_RESTRICTED_CAST_FROM_ASCII，和QT_NO_CAST_FROM_BYTEARRAY。

QT_NO_CAST_TO_ASCII

禁用自动转换QString到 8 位字符串（`char *`）。

也可以看看QT_NO_CAST_FROM_ASCII,QT_RESTRICTED_CAST_FROM_ASCII，和QT_NO_CAST_FROM_BYTEARRAY。

QT_RESTRICTED_CAST_FROM_ASCII

禁用从源文字和 8 位数据到 unicode QString 的大多数自动转换，但允许使用`QChar(char)`and`QString(const char (&ch)[N]`构造函数以及`QString::operator=(const char (&ch)[N])`赋值运算符。这提供了大部分类型安全的好处QT_NO_CAST_FROM_ASCII但不需要用户代码来包装字符和字符串文字QLatin1Char,QLatin1StringView或类似的。

将此宏与 7 位范围之外的源字符串、非文字或嵌入 NUL 字符的文字一起使用是未定义的。

也可以看看QT_NO_CAST_FROM_ASCII和QT_NO_CAST_TO_ASCII。

const char qPrintable(const QString &str)*

退货*str*作为`const char *`. 这相当于*str*。toLocal8Bit()。constData()。

在使用 qPrintable() 的语句之后，char 指针将无效。这是因为返回的数组QString::toLocal8Bit() 将超出范围。

笔记：qDebug(),qInfo(),qWarning(),qCritical(),qFatal() 期望 %s 参数采用 UTF-8 编码，而 qPrintable() 则转换为本地 8 位编码。所以qUtf8Printable() 应该用于记录字符串而不是 qPrintable()。

也可以看看qUtf8Printable()。

const wchar_t qUtf16Printable(const QString &str)*

退货*str*作为 a `const ushort *`，但转换为 a`const wchar_t *`以避免警告。这相当于*str*。utf16() 加上一些铸造。

您可以对该宏的返回值做的唯一有用的事情是将其传递给QString::asprintf() 用于`%ls`转换。特别是，返回值不是*有效*的`const wchar_t*`！

一般情况下，在使用 qUtf16Printable() 的语句之后，指针将失效。这是因为指针可能是从临时表达式中获取的，这将超出范围。

例子：

```
qWarning("%ls: %ls", qUtf16Printable(key), qUtf16Printable(value));
```

也可以看看qPrintable(),qDebug(),qInfo(),qWarning(),qCritical（），和qFatal()。

const char qUtf8Printable(const QString &str)*

退货*str*作为`const char *`. 这相当于*str*。toUtf8()。constData()。

在使用 qUtf8Printable() 的语句之后，字符指针将无效。这是因为返回的数组QString::toUtf8() 将超出范围。

例子：

```
qWarning("%s: %s", qUtf8Printable(key), qUtf8Printable(value));
```

也可以看看qPrintable(),qDebug(),qInfo(),qWarning(),qCritical（），和qFatal()。