QList Class

template < typename T> class QList

QList类是一个提供动态数组的模板类。更多的...

Header:	**#include < QList>**
CMake:	find_package(Qt6 REQUIRED COMPONENTS Core) target_link_libraries(mytarget PRIVATE Qt6::Core)
qmake:	QT += core
Inherited By:	QBluetoothServiceInfo::Alternative, QBluetoothServiceInfo::Sequence, QByteArrayList, QItemSelection, QMqttUserProperties, QNdefMessage, QPolygon, QPolygonF, QQueue, QSignalSpy, QStack, QStringList, QTestEventList, QVector, QVulkanInfoVector, and QXmlStreamAttributes

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

注意：该类中的所有函数都是reentrant。

公共类型

class	**const_iterator**
class	**iterator**
	ConstIterator
	Iterator
	const_pointer
	const_reference
	const_reverse_iterator
	difference_type
	parameter_type
	pointer
	reference
	reverse_iterator

class	**const_iterator**
	rvalue_ref
	size_type
	value_type

公共职能

	QList()
	QList(qsizetype *size*)
	QList(qsizetype *size*, QList::parameter_type *value*)
	QList(std::initializer_list< T> *args*)
	QList(InputIterator *first*, InputIterator *last*)
	QList(QList< T> &&*other*)
	QList(const QList< T> &*other*)
	~QList()
void	**append**(QList::parameter_type *value*)
void	**append**(QList::rvalue_ref *value*)
void	**append**(const QList< T> &*value*)
void	**append**(QList< T> &&*value*)
QList::const_reference	**at**(qsizetype *i*) const
QList::reference	**back**()
QList::const_reference	**back**() const
QList::iterator	**begin**()
QList::const_iterator	**begin**() const
qsizetype	**capacity**() const
QList::const_iterator	**cbegin**() const
QList::const_iterator	**cend**() const
void	**clear**()
QList::const_iterator	**constBegin**() const
QList::const_pointer	**constData**() const

	QList()
QList::const_iterator	**constEnd**() const
const T &	**constFirst**() const
const T &	**constLast**() const
bool	**contains**(const AT &*value*) const
qsizetype	**count**(const AT &*value*) const
qsizetype	**count**() const
QList::const_reverse_iterator	**crbegin**() const
QList::const_reverse_iterator	**crend**() const
QList::pointer	**data**()
QList::const_pointer	**data**() const
QList::iterator	**emplace**(qsizetype *i*, Args &&... *args*)
QList::iterator	**emplace**(QList::const_iterator *before*, Args &&... *args*)
QList::reference	**emplaceBack**(Args &&... *args*)
QList::reference	**emplace_back**(Args &&... *args*)
bool	**empty**() const
QList::iterator	**end**()
QList::const_iterator	**end**() const
bool	**endsWith**(QList::parameter_type *value*) const
QList::iterator	**erase**(QList::const_iterator *pos*)
QList::iterator	**erase**(QList::const_iterator *begin*, QList::const_iterator *end*)
QList< T> &	**fill**(QList::parameter_type *value*, qsizetype *size* = -1)
T &	**first**()
const T &	**first**() const
QList< T>	**first**(qsizetype *n*) const
QList::reference	**front**()
QList::const_reference	**front**() const
qsizetype	**indexOf**(const AT &*value*, qsizetype *from* = 0) const
QList::iterator	**insert**(qsizetype *i*, QList::parameter_type *value*)
QList::iterator	**insert**(qsizetype *i*, qsizetype *count*, QList::parameter_type *value*)

	QList()
QList::iterator	**insert**(QList::const_iterator *before*, QList::parameter_type *value*)
QList::iterator	**insert**(QList::const_iterator *before*, qsizetype *count*, QList::parameter_type *value*)
QList::iterator	**insert**(QList::const_iterator *before*, QList::rvalue_ref *value*)
QList::iterator	**insert**(qsizetype *i*, QList::rvalue_ref *value*)
bool	**isEmpty**() const
T &	**last**()
const T &	**last**() const
QList< T>	**last**(qsizetype *n*) const
qsizetype	**lastIndexOf**(const AT &*value*, qsizetype *from* = -1) const
qsizetype	**length**() const
QList< T>	**mid**(qsizetype *pos*, qsizetype *length* = -1) const
void	**move**(qsizetype *from*, qsizetype *to*)
void	**pop_back**()
void	**pop_front**()
void	**prepend**(QList::rvalue_ref *value*)
void	**prepend**(QList::parameter_type *value*)
void	**push_back**(QList::parameter_type *value*)
void	**push_back**(QList::rvalue_ref *value*)
void	**push_front**(QList::rvalue_ref *value*)
void	**push_front**(QList::parameter_type *value*)
QList::reverse_iterator	**rbegin**()
QList::const_reverse_iterator	**rbegin**() const
void	**remove**(qsizetype *i*, qsizetype *n* = 1)
qsizetype	**removeAll**(const AT &*t*)
void	**removeAt**(qsizetype *i*)
void	**removeFirst**()
qsizetype	**removeIf**(Predicate *pred*)
void	**removeLast**()
bool	**removeOne**(const AT &*t*)

	QList()
QList::reverse_iterator	**rend**()
QList::const_reverse_iterator	**rend**() const
void	**replace**(qsizetype *i*, QList::parameter_type *value*)
void	**replace**(qsizetype *i*, QList::rvalue_ref *value*)
void	**reserve**(qsizetype *size*)
void	**resize**(qsizetype *size*)
void	**shrink_to_fit**()
qsizetype	**size**() const
QList< T>	**sliced**(qsizetype *pos*, qsizetype *n*) const
QList< T>	**sliced**(qsizetype *pos*) const
void	**squeeze**()
bool	**startsWith**(QList::parameter_type *value*) const
void	**swap**(QList< T> &*other*)
void	**swapItemsAt**(qsizetype *i*, qsizetype *j*)
T	**takeAt**(qsizetype *i*)
QList::value_type	**takeFirst**()
QList::value_type	**takeLast**()
T	**value**(qsizetype *i*) const
T	**value**(qsizetype *i*, QList::parameter_type *defaultValue*) const
bool	**operator!=**(const QList< T> &*other*) const
QList< T>	**operator+**(const QList< T> &*other*) const &
QList< T>	**operator+**(const QList< T> &*other*) &&
QList< T>	**operator+**(QList< T> &&*other*) const &
QList< T>	**operator+**(QList< T> &&*other*) &&
QList< T> &	**operator+=**(const QList< T> &*other*)
QList< T> &	**operator+=**(QList< T> &&*other*)
QList< T> &	**operator+=**(QList::parameter_type *value*)
QList< T> &	**operator+=**(QList::rvalue_ref *value*)
bool	**operator<**(const QList< T> &*other*) const

	QList()
QList< T> &	**operator<<**(QList::parameter_type *value*)
QList< T> &	**operator<<**(const QList< T> &*other*)
QList< T> &	**operator<<**(QList< T> &&*other*)
QList< T> &	**operator<<**(QList::rvalue_ref *value*)
bool	**operator<=**(const QList< T> &*other*) const
QList< T> &	**operator=**(std::initializer_list< T> *args*)
QList< T> &	**operator=**(const QList< T> &*other*)
QList< T> &	**operator=**(QList< T> &&*other*)
bool	**operator==**(const QList< T> &*other*) const
bool	**operator>**(const QList< T> &*other*) const
bool	**operator>=**(const QList< T> &*other*) const
QList::reference	**[operator]**(qsizetype *i*)
QList::const_reference	**[operator]**(qsizetype *i*) const

相关非 成员

qsizetype	**erase(QList< T> &*list*, const AT &*t*)**
qsizetype	**erase_if**(QList< T> &*list*, Predicate *pred*)
size_t	**qHash**(const QList< T> &*key*, size_t *seed* = 0)
QDataStream &	**operator<<**(QDataStream &*out*, const QList< T> &*list*)
QDataStream &	**operator>>**(QDataStream &*in*, QList< T> &*list*)

详细说明

QList< T> 是 Qt 的泛型之一container classes。它将其项目存储在相邻的内存位置并提供基于索引的快速访问。QVector< T> 曾经是 Qt 5 中的不同类，但现在是 QList 的简单别名。

QList< T> 和QVarLengthArray< T> 提供类似的 API 和函数。它们通常可以互换，但会产生性能影响。以下是用例概述：

- QList 应该是您默认的第一选择。
- QVarLengthArray提供一个在堆栈上保留空间的数组，但如果需要，可以动态增长到堆上。它非常适合用于通常较小的短期容器。
- 如果你需要一个真正的链表，这保证constant time在列表中间插入并使用项目迭代器而不是索引，请使用 std::list。

注： QList 和QVarLengthArray两者都保证 C 兼容的数组布局。

注意： Qt 5 中的 QList 并不总是具有 C 兼容的数组布局，我们经常建议使用QVector相反，为了获得更可预测的性能。Qt 6 中的情况不再如此，两个类现在共享一个实现并且可以互换使用。

下面是一个存储整数的 QList 和一个存储整数的 QList 的示例QString价值观：

```
QList<int> integerList;
QList<QString> stringList;
```

QList 将其项目存储在连续内存的数组中。通常，创建的列表具有初始大小。例如，以下代码构造一个包含 200 个元素的 QList：

```
QList<QString> list(200);
```

元素会自动初始化default-constructed value。如果要使用不同的值初始化列表，请将该值作为第二个参数传递给构造函数：

```
QList<QString> list(200, "Pass");
```

您也可以致电fill() 随时用值填充列表。

QList 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项目，可以使用运算符。在非常量列表上，operator 返回对可在赋值左侧使用的项的引用：

```
if (list[0] == "Liz")
    list[0] = "Elizabeth";
```

对于只读访问，另一种语法是使用at():

```
for (qsizetype i = 0; i < list.size(); ++i) {
    if (list.at(i) == "Alfonso")
        cout << "Found Alfonso at position " << i << endl;
}
```

at() 可能比运算符 [] () 更快，因为它永远不会导致deep copy发生。

访问 QList 中存储的数据的另一种方法是调用data()。该函数返回一个指向列表中第一项的指针。您可以使用指针直接访问和修改列表中存储的元素。如果您需要将 QList 传递给接受普通 C++ 数组的函数，则指针也很有用。

如果要查找列表中特定值的所有出现位置，请使用indexOf（） 或者lastIndexOf()。前者从给定的索引位置开始向前搜索，后者向后搜索。如果找到匹配项，两者都会返回匹配项的索引；否则，它们返回-1。例如：

```
qsizetype i = list.indexOf("Harumi");
if (i != -1)
    cout << "First occurrence of Harumi is at position " << i << endl;
```

如果您只想检查列表是否包含特定值，请使用contains()。如果您想找出特定值在列表中出现的次数，请使用count()。

QList 提供了以下基本函数来添加、移动和删除项目： insert(),replace(),remove(),prepend(),append()。除了append(),prepend（） 和replace()，这些函数可能会很慢（linear time）对于大型列表，因为它们需要将列表中的许多项目在内存中移动一个位置。如果您想要一个在中间提供快速插入/删除的容器类，请使用 std::list 代替。

与普通 C++ 数组不同，QList 可以通过调用随时调整大小resize()。如果新大小大于旧大小，QList 可能需要重新分配整个列表。QList 尝试通过预分配实际数据所需两倍的内存来减少重新分配的次数。

如果您正在逐步构建 QList 并提前知道它将包含多少个元素，您可以调用reserve()，要求QList预先分配一定量的内存。您也可以致电capacity() 找出 QList 实际分配了多少内存。

请注意，使用非常量运算符和函数可能会导致 QList 进行数据的深层复制，因为隐式共享。

QList 的值类型必须是assignable data type。这涵盖了大多数常用的数据类型，但是编译器不允许您存储例如QWidget作为一个值；相反，存储一个QWidget*。一些函数有额外的要求；例如，indexOf（） 和lastIndexOf() 期望值类型支持`operator==()`。这些要求是按函数记录的。

要迭代项目，请参阅Iterating over Containers。

除了QList之外，Qt还提供QVarLengthArray，一个非常低级的类，几乎没有针对速度进行优化的函数。

有关使用 Qt 容器的更多信息

有关 Qt 容器相互比较以及与 STL 容器比较的详细讨论，请参阅Understand the Qt Containers。

最大大小和内存不足情况

QList 的最大大小取决于架构。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。QList 中可以存储的元素数量是该最大大小除以所存储元素的大小。

当内存分配失败时，QList使用Q_CHECK_PTR宏，`std::bad_alloc`如果应用程序正在使用异常支持进行编译，则会引发异常。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。

成员类型文档

[alias]QList::ConstIterator

Qt 风格的同义词QList::const_iterator。

[alias]QList::Iterator

Qt 风格的同义词QList::iterator。

[alias]QList::const_pointer

提供 STL 兼容性。

[alias]QList::const_reference

提供 STL 兼容性。

[alias]QList::const_reverse_iterator

QList::const_reverse_iterator typedef 提供了一个 STL 风格的 const 反向迭代器QList。

警告：隐式共享容器上的迭代器的工作方式与 STL 迭代器不同。当迭代器在容器上处于活动状态时，您应该避免复制该容器。欲了解更多信息，请阅读Implicit sharing iterator problem。

警告：迭代器在以下情况下无效：QList被修改。考虑到默认情况下所有迭代器都是无效的。此规则的例外情况已明确记录。

也可以看看QList::rbegin(),QList::rend(),QList::reverse_iterator， 和QList::const_iterator。

[alias]QList::difference_type

提供 STL 兼容性。

[alias]QList::parameter_type

[alias]QList::pointer

提供 STL 兼容性。

[alias]QList::reference

提供 STL 兼容性。

[alias]QList::reverse_iterator

QList::reverse_iterator typedef 提供了一个 STL 风格的非常量反向迭代器QList。

警告：隐式共享容器上的迭代器的工作方式与 STL 迭代器不同。当迭代器在容器上处于活动状态时，您应该避免复制该容器。欲了解更多信息，请阅读Implicit sharing iterator problem。

警告：迭代器在以下情况下无效：QList被修改。考虑到默认情况下所有迭代器都是无效的。此规则的例外情况已明确记录。

也可以看看QList::rbegin(),QList::rend(),QList::const_reverse_iterator， 和QList::iterator。

[alias]QList::rvalue_ref

[alias]QList::size_type

提供 STL 兼容性。

[alias]QList::value_type

提供 STL 兼容性。

成员函数文档

void QList::prepend(QList::parameter_type value)

void QList::prepend(QList::rvalue_ref value)

*value*在列表的开头。

例子：

```
QList<QString> list;
list.prepend("one");
list.prepend("two");
list.prepend("three");
// list: ["three", "two", "one"]
```

这与 list.insert(0,*value*) 。

通常此操作相对较快（摊销constant time）。QList能够在列表数据的开头分配额外的内存并沿该方向增长，而无需在每次操作时重新分配或移动数据。但是，如果您想要一个保证的容器类constant time前置，使用 std::list 代替，但更喜欢QList否则。

也可以看看append（） 和insert()。

template < typename Args> QList::reference QList::emplaceBack(Args &&... args)

template < typename Args> QList::reference QList::emplace_back(Args &&... args)

将新元素添加到容器的末尾。这个新元素是使用就地构建的*args*作为其构建的论据。

返回对新元素的引用。

例子：

```
QList<QString> list{"one", "two"};
list.emplaceBack(3, 'a');
qDebug() << list;
// list: ["one", "two", "aaa"]
```

还可以使用返回的引用来访问新创建的对象：

```
QList<QString> list;
auto &ref = list.emplaceBack();
ref = "one";
// list: ["one"]
```

这与 list.emplace(list.size(),*args*） 。

也可以看看emplace。

QList::iterator QList::insert(qsizetype i, QList::parameter_type value)

QList::iterator QList::insert(qsizetype i, QList::rvalue_ref value)

*value*在索引位置*i*在列表中。如果*i*为 0 时，该值将添加到列表的前面。如果*i*是size()，该值被附加到列表中。

例子：

```
QList<QString> list = {"alpha", "beta", "delta"};
list.insert(2, "gamma");
// list: ["alpha", "beta", "gamma", "delta"]
```

对于大型列表，此操作可能会很慢（linear time），因为它需要移动索引处的所有项目*i*并在内存中再增加一位。如果您想要一个提供快速的容器类insert() 函数，请使用 std::list 代替。

也可以看看append(),prepend（），和remove()。

QList::iterator QList::insert(QList::const_iterator before, QList::parameter_type value)

QList::iterator QList::insert(QList::const_iterator before, QList::rvalue_ref value)

这是一个重载函数。

*value*在迭代器指向的项前面*before*。返回一个指向插入项的迭代器。

void QList::replace(qsizetype i, QList::parameter_type value)

void QList::replace(qsizetype i, QList::rvalue_ref value)

替换索引位置处的项目*i*和*value*。

*i*必须是列表中的有效索引位置（即，0 <=*i*<size()）。

也可以看看[operator]（） 和remove()。

void QList::push_front(QList::parameter_type value)

void QList::push_front(QList::rvalue_ref value)

提供此函数是为了兼容 STL。它相当于前置(*value*）。

QList< T> QList::operator+(QList< T> &&other) &&

QList< T> QList::operator+(const QList< T> &other) &&

QList< T> QList::operator+(QList< T> &&other) const &

QList< T> QList::operator+(const QList< T> &other) const &

返回一个列表，其中包含此列表中的所有项目，后跟*other*列表。

也可以看看operator+=()。

QList::QList()

构造一个空列表。

也可以看看resize()。

[explicit]QList::QList(qsizetype size)

构造一个初始大小为的列表*size*元素。

元素被初始化为default-constructed value。

也可以看看resize()。

QList::QList(qsizetype size, QList::parameter_type value)

构造一个初始大小为的列表*size*元素。每个元素都初始化为*value*。

也可以看看resize（） 和fill()。

QList::QList(std::initializer_list< T> args)

从 std::initializer_list 给出的列表构造一个列表*args*。

template < typename InputIterator> QList::QList(InputIterator first, InputIterator last)

使用迭代器范围 [中的内容构造一个列表*first*,*last*）。

的值类型`InputIterator`必须可转换为`T`。

[default]QList::QList(QList< T> &&other)

Move 构造一个 QList 实例，使其指向与*other*正在指着。

[default]QList::QList(const QList< T> &other)

构造一个副本*other*。

此操作需要constant time，因为 QList 是implicitly shared。这使得从函数返回 QList 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=()。

[default]QList::~QList()

销毁列表。

void QList::append(QList::parameter_type value)

*value*在列表的末尾。

例子：

```
QList<QString> list;
list.append("one");
list.append("two");
QString three = "three";
list.append(three);
// list: ["one", "two", "three"]
// three: "three"
```

这与调用 resize(size() + 1) 并赋值*value*到列表中新的最后一个元素。

这个操作比较快，因为QList通常会分配比所需更多的内存，因此它可以增长而无需每次重新分配整个列表。

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

void QList::append(QList::rvalue_ref value)

这是一个重载函数。

例子：

```
QList<QString> list;
list.append("one");
list.append("two");
QString three = "three";
list.append(std::move(three));
// list: ["one", "two", "three"]
// three: ""
```

void QList::append(const QList< T> &value)

这是一个重载函数。

追加以下项目*value*列到这个列表中。

也可以看看operator<<（）和operator+=()。

[since 6.0]void QList::append(QList< T> &&value)

这是一个重载函数。

移动项目*value*列表到此列表的末尾。

这个函数是在Qt 6.0中引入的。

也可以看看operator<<（） 和operator+=()。

QList::const_reference QList::at(qsizetype i) const

返回索引位置处的项目*i*在列表中。

*i*必须是列表中的有效索引位置（即，0 <=*i*<size())。

也可以看看value（） 和[operator]()。

QList::reference QList::back()

提供此函数是为了兼容 STL。它相当于last()。

QList::const_reference QList::back() const

这是一个重载函数。

QList::iterator QList::begin()

返回一个STL-style iterator指向列表中的第一项。

警告：返回的迭代器在分离或当QList被修改。

也可以看看constBegin（） 和end()。

QList::const_iterator QList::begin() const

这是一个重载函数。

qsizetype QList::capacity() const

返回在不强制重新分配的情况下可以存储在列表中的最大项目数。

该函数的唯一目的是提供一种微调的手段QList的内存使用情况。一般来说，您很少需要调用此函数。如果您想知道列表中有多少项目，请致电size()。

注意：静态分配的列表将报告容量为 0，即使它不为空。

警告：分配的内存块中的可用空间位置未定义。换句话说，您不应该假设空闲内存始终位于列表的末尾。您可以致电reserve() 确保末尾有足够的空间。

也可以看看reserve（） 和squeeze()。

QList::const_iterator QList::cbegin() const

返回一个常量STL-style iterator指向列表中的第一项。

警告：返回的迭代器在分离或当QList被修改。

也可以看看begin（） 和cend()。

QList::const_iterator QList::cend() const

返回一个常量STL-style iterator指向列表中最后一项的后面。

警告：返回的迭代器在分离或当QList被修改。

也可以看看cbegin（） 和end()。

void QList::clear()

从列表中删除所有元素。

如果不共享此列表，则capacity() 被保留。使用squeeze() 去化过剩产能。

注意：在 5.7 之前的 Qt 版本中（对于QVector）和 6.0（对于QList），该函数释放了列表使用的内存，而不是保留容量。

也可以看看resize（） 和squeeze()。

QList::const_iterator QList::constBegin() const

返回一个常量STL-style iterator指向列表中的第一项。

警告：返回的迭代器在分离或当QList被修改。

也可以看看begin（） 和constEnd()。

QList::const_pointer QList::constData() const

返回指向列表中存储的数据的常量指针。该指针可用于访问列表中的项目。

警告：指针在分离时或当QList被修改。

此函数主要用于将列表传递给接受普通 C++ 数组的函数。

也可以看看data（） 和[operator]()。

QList::const_iterator QList::constEnd() const

返回一个常量STL-style iterator指向列表中最后一项的后面。

警告：返回的迭代器在分离或当QList被修改。

也可以看看constBegin（） 和end()。

const T &QList::constFirst() const

返回对列表中第一项的常量引用。该函数假设列表不为空。

也可以看看constLast(),isEmpty（）， 和first()。

const T &QList::constLast() const

返回对列表中最后一项的常量引用。该函数假设列表不为空。

也可以看看constFirst(),isEmpty（）， 和last()。

template < typename AT> bool QList::contains(const AT &value) const

`true`如果列表包含出现则返回*value*; 否则返回`false`.

此函数要求值类型具有 的实现`operator==()`。

也可以看看indexOf（） 和count()。

template < typename AT> qsizetype QList::count(const AT &value) const

返回出现的次数*value*在列表中。

此函数要求值类型具有 的实现`operator==()`。

也可以看看contains（） 和indexOf()。

qsizetype QList::count() const

这是一个重载函数。

与...一样size()。

QList::const_reverse_iterator QList::crbegin() const

返回一个常量STL-style反向迭代器以相反的顺序指向列表中的第一项。

警告：返回的迭代器在分离或当QList被修改。

也可以看看begin(),rbegin（）， 和rend()。

QList::const_reverse_iterator QList::crend() const

返回一个常量STL-style反向迭代器以相反的顺序指向列表中最后一项的后面。

警告：返回的迭代器在分离或当QList被修改。

也可以看看end(),rend（）， 和rbegin()。

QList::pointer QList::data()

返回指向列表中存储的数据的指针。该指针可用于访问和修改列表中的项目。

例子：

```
QList<int> list(10);
int *data = list.data();
for (qsizetype i = 0; i < 10; ++i)
    data[i] = 2 * i;
```

警告：指针在分离时或当QList被修改。

此函数主要用于将列表传递给接受普通 C++ 数组的函数。

也可以看看constData（） 和[operator]()。

QList::const_pointer QList::data() const

这是一个重载函数。

template < typename Args > QList::iterator QList::emplace(qsizetype i, Args &&... args)

通过在位置插入新元素来扩展容器*i*。这个新元素是使用就地构建的*args*作为其构建的论据。

返回新元素的迭代器。

例子：

```
QList<QString> list{"a", "ccc"};
list.emplace(1, 2, 'b');
// list: ["a", "bb", "ccc"]
```

注意：保证元素将在开始时就位创建，但之后可能会被复制或移动到正确的位置。

也可以看看emplaceBack。

template < typename Args > QList::iterator QList::emplace(QList::const_iterator before, Args &&... args)

这是一个重载函数。

在迭代器指向的项前面创建一个新元素*before*。这个新元素是使用就地构建的*args*作为其构建的论据。

返回新元素的迭代器。

bool QList::empty() const

提供此函数是为了兼容 STL。它相当于isEmpty()，`true`如果列表为空则返回；否则返回`false`.

QList::iterator QList::end()

返回一个STL-style iterator指向列表中最后一项的后面。

警告：返回的迭代器在分离或当QList被修改。

也可以看看begin（） 和constEnd()。

QList::const_iterator QList::end() const

这是一个重载函数。

bool QList::endsWith(QList::parameter_type value) const

如果`true`此列表不为空且其最后一项等于*value*; 否则返回`false`.

也可以看看isEmpty（） 和last()。

QList::iterator QList::erase(QList::const_iterator pos)

删除迭代器指向的项*pos*从列表中，并返回一个迭代器到列表中的下一个项目（可能是end()）。

元素删除将保留列表的容量，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()。

注：当QList不是implicitly shared，此函数仅使指定位置或之后的迭代器无效。

也可以看看insert（） 和remove()。

QList::iterator QList::erase(QList::const_iterator begin, QList::const_iterator end)

这是一个重载函数。

从中删除所有项目*begin*最多（但不包括）*end*。返回一个指向相同项目的迭代器*end*在通话之前提及。

元素删除将保留列表的容量，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()。

注：当QList不是implicitly shared，此函数仅使指定位置或之后的迭代器无效。

QList< T> &QList::fill(QList::parameter_type value, qsizetype size = -1)

分配*value*到列表中的所有项目。如果*size*与 -1（默认值）不同，列表大小调整为*size*预先。

例子：

```
QList<QString> list(3);
list.fill("Yes");
// list: ["Yes", "Yes", "Yes"]

list.fill("oh", 5);
// list: ["oh", "oh", "oh", "oh", "oh"]
```

也可以看看resize()。

T &QList::first()

返回对列表中第一项的引用。该函数假设列表不为空。

也可以看看last(),isEmpty（），和constFirst()。

const T &QList::first() const

这是一个重载函数。

[since 6.0]QList< T> QList::first(qsizetype n) const

返回包含第一个的子列表*n*该列表的元素。

注意：当以下情况时，行为未定义*n*< 0 或*n*>size()。

这个函数是在Qt 6.0中引入的。

也可以看看last（） 和sliced()。

QList::reference QList::front()

提供此函数是为了兼容 STL。它相当于first()。

QList::const_reference QList::front() const

这是一个重载函数。

template < typename AT> qsizetype QList::indexOf(const AT &value, qsizetype from = 0) const

返回第一次出现的索引位置*value*在列表中，从索引位置向前搜索*from*。如果没有匹配的项目，则返回 -1。

例子：

```
QList<QString> list{"A", "B", "C", "B", "A"};
list.indexOf("B");            // returns 1
list.indexOf("B", 1);         // returns 1
list.indexOf("B", 2);         // returns 3
list.indexOf("X");            // returns -1
```

此函数要求值类型具有 的实现`operator==()`。

也可以看看lastIndexOf () 和contains()。

QList::iterator QList::insert(qsizetype i, qsizetype count, QList::parameter_type value)

这是一个重载函数。

*count*的副本*value*在索引位置*i*在列表中。

例子：

```
QList<double> list = {2.718, 1.442, 0.4342};
list.insert(1, 3, 9.9);
// list: [2.718, 9.9, 9.9, 9.9, 1.442, 0.4342]
```

QList::iterator QList::insert(QList::const_iterator before, qsizetype count, QList::parameter_type value)

*count*的副本*value*在迭代器指向的项前面*before*。返回一个指向第一个插入项的迭代器。

bool QList::isEmpty() const

`true`如果列表大小为 0，则返回；否则返回`false`.

也可以看看size（） 和resize()。

T &QList::last()

返回对列表中最后一项的引用。该函数假设列表不为空。

也可以看看first(),isEmpty（），和constLast()。

const T &QList::last() const

这是一个重载函数。

[since 6.0]QList< T> QList::last(qsizetype n) const

返回包含最后一个的子列表*n*该列表的元素。

注意：当以下情况时，行为未定义*n*< 0 或*n*>size()。

这个函数是在Qt 6.0中引入的。

也可以看看first（） 和sliced()。

template < typename AT> qsizetype QList::lastIndexOf(const AT &value, qsizetype from = -1) const

返回该值最后一次出现的索引位置*value*在列表中，从索引位置向后搜索*from*。如果*from*为 -1（默认值），则从最后一项开始搜索。如果没有匹配的项目，则返回 -1。

例子：

```
QList<QString> list = {"A", "B", "C", "B", "A"};
list.lastIndexOf("B");        // returns 3
list.lastIndexOf("B", 3);     // returns 3
list.lastIndexOf("B", 2);     // returns 1
list.lastIndexOf("X");        // returns -1
```

此函数要求值类型具有 的实现`operator==()`。

也可以看看indexOf()。

qsizetype QList::length() const

与...一样size（） 和count()。

也可以看看size（） 和count()。

QList< T> QList::mid(qsizetype pos, qsizetype length = -1) const

返回一个子列表，其中包含此列表中的元素，从位置开始*pos*。如果*length*是-1（默认值），之后的所有元素*pos*被包含在内; 否则*length*元素（或所有剩余元素，如果少于*length*元素）包括在内。

void QList::move(qsizetype from, qsizetype to)

将项目移动到索引位置*from*到索引位置*to*。

void QList::pop_back()

提供此函数是为了兼容 STL。它相当于removeLast()。

void QList::pop_front()

提供此函数是为了兼容 STL。它相当于removeFirst()。

void QList::push_back(QList::parameter_type value)

提供此函数是为了兼容 STL。它相当于附加（*value*）。

void QList::push_back(QList::rvalue_ref value)

这是一个重载函数。

QList::reverse_iterator QList::rbegin()

返回一个STL-style反向迭代器以相反的顺序指向列表中的第一项。

警告：返回的迭代器在分离或当QList被修改。

也可以看看begin(),crbegin（），和rend()。

QList::const_reverse_iterator QList::rbegin() const

这是一个重载函数。

void QList::remove(qsizetype i, qsizetype n = 1)

删除*n*列表中的元素，从索引位置开始*i*。

元素删除将保留列表的容量，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()。

注：当QList不是implicitly shared，此函数仅使指定位置或之后的迭代器无效。

也可以看看insert(),replace（），和fill()。

template < typename AT> qsizetype QList::removeAll(const AT &t)

删除所有比较等于的元素*t*从列表中。返回删除的元素数（如果有）。

元素删除将保留列表的容量，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()。

也可以看看removeOne()。

void QList::removeAt(qsizetype i)

删除索引位置的元素*i*。相当于

```
remove(i);
```

元素删除将保留列表的容量，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()。

注：当QList不是implicitly shared，此函数仅使指定位置或之后的迭代器无效。

也可以看看remove()。

void QList::removeFirst()

删除列表中的第一项。调用此函数相当于调用remove(0)。该列表不能为空。如果列表可以为空，则调用isEmpty()在调用该函数之前。

元素删除将保留列表的容量，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()。

也可以看看remove(),takeFirst（），和isEmpty()。

[since 6.1]template < typename Predicate> qsizetype QList::removeIf(Predicate pred)

删除谓词所属的所有元素*pred*从列表中返回 true。返回删除的元素数（如果有）。

该函数是在 Qt 6.1 中引入的。

也可以看看removeAll()。

void QList::removeLast()

删除列表中的最后一项。调用此函数相当于调用remove(size() - 1)。该列表不能为空。如果列表可以为空，则调用isEmpty() 在调用该函数之前。

元素删除将保留列表的容量，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()。

也可以看看remove(),takeLast(),removeFirst（），和isEmpty()。

template < typename AT> bool QList::removeOne(const AT &t)

删除第一个比较等于的元素*t*从列表中。返回元素是否确实被删除。

元素删除将保留列表的容量，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()。

也可以看看removeAll()。

QList::reverse_iterator QList::rend()

返回一个STL-style反向迭代器以相反的顺序指向列表中最后一项的后面。

警告：返回的迭代器在分离或当QList被修改。

也可以看看end(),crend（），和rbegin()。

QList::const_reverse_iterator QList::rend() const

这是一个重载函数。

void QList::reserve(qsizetype size)

尝试分配内存至少*size*元素。

如果您事先知道列表有多大，则应该调用此函数以防止重新分配和内存碎片。如果您经常调整列表的大小，您也可能会获得更好的性能。

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，一旦超过保留大小，列表将根据需要增长，这可能会导致分配比最佳高估更大的分配，并且会减慢触发它的操作。

警告： reserve() 保留内存，但不会更改列表的大小。访问列表当前末尾之外的数据是未定义的行为。如果需要访问列表当前末尾之外的内存，请使用resize()。

也可以看看squeeze(),capacity（），和resize()。

[since 6.0]void QList::resize(qsizetype size)

将列表的大小设置为*size*。如果*size*大于当前大小，元素添加到末尾；新元素初始化为default-constructed value。如果*size*小于当前大小，则从末尾删除元素。

如果不共享此列表，则capacity() 被保留。使用squeeze() 去化过剩产能。

注意：在 5.7 之前的 Qt 版本中（对于QVector;QList在 6.0 之前缺少 resize()），该函数释放列表使用的内存而不是保留容量。

这个函数是在Qt 6.0中引入的。

也可以看看size()。

void QList::shrink_to_fit()

提供此函数是为了兼容 STL。它相当于squeeze()。

qsizetype QList::size() const

返回列表中的项目数。

也可以看看isEmpty（） 和resize()。

[since 6.0]QList< T> QList::sliced(qsizetype pos, qsizetype n) const

返回一个子列表，其中包含*n*此列表的元素，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()。

这个函数是在Qt 6.0中引入的。

也可以看看first（） 和last()。

[since 6.0]QList< T> QList::sliced(qsizetype pos) const

这是一个重载函数。

返回一个子列表，其中包含此列表中从位置开始的元素*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（） 和last()。

void QList::squeeze()

释放存储项目不需要的任何内存。

该函数的唯一目的是提供一种微调的手段QList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 QList::startsWith(QList::parameter_type value) const

如果`true`此列表不为空且其第一项等于*value*; 否则返回`false`.

也可以看看isEmpty（） 和first()。

void QList::swap(QList< T> &other)

掉期清单*other*有了这个清单。这个操作非常快并且永远不会失败。

void QList::swapItemsAt(qsizetype i, qsizetype j)

交换索引位置的物品*i*项目位于索引位置*j*。该函数假设两者*i*和*j*至少为 0 但小于size()。为了避免失败，请测试两者*i*和*j*至少为 0 且小于size()。

T QList::takeAt(qsizetype i)

删除索引位置的元素*i*并返回它。

相当于

```
T t = at(i);
remove(i);
return t;
```

注：当QList不是implicitly shared，此函数仅使指定位置或之后的迭代器无效。

也可以看看takeFirst（） 和takeLast()。

QList::value_type QList::takeFirst()

删除列表中的第一项并将其返回。该函数假设列表不为空。为避免失败，请致电isEmpty() 在调用该函数之前。

也可以看看takeLast（） 和removeFirst()。

QList::value_type QList::takeLast()

删除列表中的最后一项并将其返回。该函数假设列表不为空。为避免失败，请致电isEmpty() 在调用该函数之前。

如果不使用返回值，removeLast() 效率更高。

也可以看看takeFirst（） 和removeLast()。

T QList::value(qsizetype i) const

返回索引位置处的值*i*在列表中。

如果索引*i*超出范围，函数返回default-constructed value。如果你确定*i*在范围内，您可以使用at() 代替，速度稍快一些。

也可以看看at（） 和[operator]()。

T QList::value(qsizetype i, QList::parameter_type defaultValue) const

这是一个重载函数。

如果索引*i*越界，函数返回*defaultValue*。

bool QList::operator!=(const QList< T> &other) const

返回`true`如果*other*不等于此列表；否则返回`false`.

如果两个列表包含相同顺序的相同值，则认为它们相等。

此函数要求值类型具有 的实现`operator==()`。

也可以看看operator==()。

QList< T> &QList::operator+=(const QList< T> &other)

追加以下项目*other*list 到此列表并返回对此列表的引用。

也可以看看operator+（） 和append()。

[since 6.0]QList< T> &QList::operator+=(QList< T> &&other)

这是一个重载函数。

这个函数是在Qt 6.0中引入的。

也可以看看operator+（） 和append()。

QList< T> &QList::operator+=(QList::parameter_type value)

这是一个重载函数。

追加*value*到列表中。

也可以看看append（） 和operator<<()。

QList< T> &QList::operator+=(QList::rvalue_ref value)

这是一个重载函数。

也可以看看append（） 和operator<<()。

bool QList::operator<(const QList< T> &other) const

`true`如果此列表是则返回lexically less than *other*; 否则返回`false`.

此函数要求值类型具有 的实现`operator<()`。

QList< T> &QList::operator<<(QList::parameter_type value)

追加*value*到列表并返回对此列表的引用。

也可以看看append（） 和operator+=()。

QList< T> &QList::operator<<(const QList< T> &other)

追加*other*到列表并返回对该列表的引用。

[since 6.0]QList< T> &QList::operator<<(QList< T> &&other)

这是一个重载函数。

这个函数是在Qt 6.0中引入的。

QList< T> &QList::operator<<(QList::rvalue_ref value)

这是一个重载函数。

也可以看看append（） 和operator+=()。

bool QList::operator<=(const QList< T> &other) const

`true`如果此列表是则返回lexically less than or equal to *other*; 否则返回`false`.

此函数要求值类型具有 的实现`operator<()`。

QList< T> &QList::operator=(std::initializer_list< T> args)

分配值的集合*args*对此QList实例。

[default]QList< T> &QList::operator=(const QList< T> &other)

分配*other*到此列表并返回对此列表的引用。

[default]QList< T> &QList::operator=(QList< T> &&other)

移动分配*other*对此QList实例。

bool QList::operator==(const QList< T> &other) const

返回`true`如果*other*等于这个列表；否则返回`false`.

如果两个列表包含相同顺序的相同值，则认为它们相等。

此函数要求值类型具有 的实现`operator==()`。

也可以看看operator!=()。

bool QList::operator>(const QList< T> &other) const

`true`如果此列表是则返回lexically greater than *other*; 否则返回`false`.

此函数要求值类型具有 的实现`operator<()`。

bool QList::operator>=(const QList< T> &other) const

`true`如果此列表是则返回lexically greater than or equal to *other*; 否则返回`false`.

此函数要求值类型具有 的实现`operator<()`。

QList::reference QList::operator

返回索引位置处的项目*i*作为可修改的参考。

*i*必须是列表中的有效索引位置（即，0 <=*i*<size())。

请注意，使用非常量运算符可能会导致QList进行深复制。

也可以看看at（） 和value()。

QList::const_reference QList::operator const

这是一个重载函数。

与在（*i*）。

相关非成员

[since 6.1]template <typename T, typename AT> qsizetype erase(QList< T> &list, const AT &t)

删除所有比较等于的元素*t*从列表中*list*。返回删除的元素数（如果有）。

注：不同于QList::removeAll,*t*不允许是对内部元素的引用*list*。如果您不能确定情况并非如此，请复印一份*t*并用副本调用此函数。

该函数是在 Qt 6.1 中引入的。

也可以看看QList::removeAll（） 和erase_if。

[since 6.1]template <typename T, typename Predicate> qsizetype erase_if(QList< T> &list, Predicate pred)

删除谓词所属的所有元素*pred*从列表中返回 true*list*。返回删除的元素数（如果有）。

该函数是在 Qt 6.1 中引入的。

也可以看看erase。

template < typename T> size_t qHash(const QList< T> &key, size_t seed = 0)

返回哈希值*key*， 使用*seed*为计算提供种子。

该函数需要为值类型重载qHash() `T`。

template < typename T> QDataStream &operator<<(QDataStream &out, const QList< T> &list)

写清单*list*流式传输*out*。

该函数需要值类型来实现`operator<<()`。

也可以看看Format of the QDataStream operators。

template < typename T> QDataStream &operator>>(QDataStream &in, QList< T> &list)

从流中读取列表*in*进入*list*。

该函数需要值类型来实现`operator>>()`。

也可以看看Format of the QDataStream operators。