

# Qt 中文文档翻译

作者: [QtDocumentCN](#)

2021 年 5 月 28 日



# 目录

第一章 参与翻译的人员	5
第二章 API Design Principles(未完结)	7
第三章 QAbstractAnimation	17
第四章 QAbstractAudioDeviceInfo	19
第五章 QAbstractAudioInput	21
第六章 QAbstractAudioOutput	25
第七章 QAbstractAxis	29
第八章 QAbstractBarSeries	43
第九章 QAbstractItemModel	47
第十章 QAbstractSocket	83
第十一章 QSql	85
第十二章 QSqlDatabase	87
第十三章 QWaitCondition	101
第十四章 QWebEngineHistory	105
第十五章 QWebEngineHistoryItem	109
第十六章 QWebEngineView	111
第十七章 QWriteLocker	115
第十八章 QX11Info	117



# 第一章 参与翻译的人员

Github 账号	翻译的章节	状态
CryFeiFei	QAbstractBarSeries	编写中
	QAbstractAxis	编写中
	QAbstractAudioOutput	待完善
	QAbstractAudioInput	待完善
	QAbstractAnimation	待完善
FlyWM	QAbstractItemModel	待完善
chenjiqing	QAbstractItemView	编写中
skykeyjoker	TODO	TODO.tex
xyz1001	TODO	TODO.tex
leytou	TODO	TODO.tex
froser	TODO	TODO.tex
顾晓	TODO	TODO.tex
吴冬亮	TODO	TODO.tex
Zgb1Kylin	TODO	TODO.tex
JackLove1	TODO	TODO.tex



## 第二章 API Design Principles(未完待续)

API 设计规范

译者注：

本文不来自于 Qt 文档，而是来自于 Qt Wiki: [API\\_Design\\_Principles](#)

API(Application Programming Interface)，应用开发接口，本文中也将 P 解释为 Programmer（开发者）。

Qt 最出名的特点之一是一致性强、易于学习、功能强大的 API。本文尝试对我们在设计 Qt 风格的 API 中积累的诀窍进行总结。其中许多准则都是通用的，其它的则是习惯性用法，我们主要为了保持接口一致性而继续遵循。

尽管这些准则主要面向公共接口，但也鼓励您在设计内部接口时使用相同的技术，这对与您合作开发的同僚会更加友好。

您可能也会有兴趣查阅 Jasmin Blanchette 的 [Little Manual of API Design \(PDF\)](#)，或它的前身，由 Matthias Ettrich 编写的 [Designing Qt-Style C++ APIs](#)。

优秀接口的六大特点

API 是面向开发者的，而 GUI 则是面向终端用户。API 中的 P 代表开发者（Programmer），而非程序（Program），目的是指出 API 由开发者，即人类（而非计算机）所使用这一特点。

Matthias 在 Qt 季刊第 13 期，关于 API 设计的文章中，声称他坚信 API 应该是最小化但完备的，具备清晰而简洁的语义，符合直觉，被开发者，易于记忆，能够引导开发者编写高可读性的代码。

最小化

最小化的 API 指包含尽可能少的公共成员和最少的类。这可让 API 更易于理解、记忆、调试和修改。

完备性完备的 API 意味着具备所有应有的功能。这可能会与最小化产生冲突。此外，若一个成员函数位于错误的类中，则大多数潜在的用户将无法找到它。

清晰简洁的语义

与其它设计协作时，应该让您的设计做到最小例外。通用化会让事情更简单。特例可能存在，但不

应成为关注的焦点。在处理特定问题时，不应让解决方案过度泛化。（例如，Qt 3 中的 `QMineSourceFactory` 本该被称作 `QImageLoader`，并且具备另一套 API。）

### 复合直觉

如同计算机上其它内容，API 应复合直觉。不同的开发经验和背景会导致对复合直觉与否有不同的感知。复合直觉的 API 应能让中等经验的开发者无需阅读文档并直接使用，并让不知道这个 API 的开发者可以理解使用它编写的代码。

### 易于记忆

为了让 API 易于记忆，请选择一组保持一致并足够精确的命名约定。使用可理解的模式和概念，并且避免缩写。

### 可读性导向

编写代码只需要一次，但阅读（以及调试和修改）则会非常频繁。高可读性的代码通常需要花更多时间编写，但可以在产品的生命周期中节约更多的时间。

最后需要谨记，不同类型的用户会使用 API 的不同部分。在单纯地创建一个 Qt 类实例能非常直观的同时，希望用户在尝试继承它之前先阅读文档则是很合理的。

### 静态多态

相似的代码类应具有相似的 API。可以使用继承来实现——当运行时多态支持时，这是很合理的。但多态同时也可以体现在设计截断。例如，若将代码中的 `QProgressBar` 换为 `QSlider`，或将 `QString` 换为 `QByteArray`，他们间相似的 API 会另替换操作变得非常容易。这便是为何我们称之为“静态多态”。

静态多态同样可让记忆 API 和开发模式变得更加简单。结果是，对于一组有关联的类，相似的 API 通常比为每个类独立设计的完美 API 更加好用。

在 Qt 中，当不具备有足够说服力的原因时，我们更倾向于使用静态多态，而非继承。这减少了 Qt 的公共类数量，并让新用户更容易在文档中找到需要的内容。

### 好的

`QDialogBox` 和 `QMessageBox` 具有相似的 API，以用于处理按钮（`addButton()`，`setStandardButtons()`），但无需继承自某些“`QAbstractButtonBox`”类。

### 坏的

`QAbstractSocket` 被 `QTcpSocket` 和 `QUdpSocket` 所继承，但这两个类的交互方式差异很大。看起来并没有人使用过（或能够使用）`QAbstractSocket` 指针来进行通用且有效的操作。

### 存疑

`QBoxLayout` 是 `QHBoxLayout` 和 `QVBoxLayout` 的基类。优点：可以在工具栏中使用 `QBoxLayout`，调用 `setOrientation()` 来令其水平/垂直排布。缺点：引入额外的类，用户可能会写出形如 `((QBoxLayout *)hbox)→setOrientation(Qt::Vertical)` 的代码，而这是不合理的。

基于属性的 API 比较新的 Qt 类倾向于使用基于属性的 API，例如：



```
QTimer timer;
timer.setInterval(1000);
timer.setSingleShot(true);
timer.start();
```

此处的属性，指的是作为对象状态一部分的任何概念性的特征——无论是否是实际的 `Q_PROPERTY`。只要可行，用户都应该允许以任何顺序设置属性，也就是说，这些属性应该是正交的。例如，上文的代码也可以写为：

```
QTimer timer;
timer.setSingleShot(true);
timer.setInterval(1000);
timer.start();
```

为了方便，我们也可以这样写：

```
timer.start(1000);
```

类似的，对于 `QRegExp`，我们可以：

```
QRegExp regExp;
regExp.setCaseSensitive(Qt::CaseInsensitive);
regExp.setPattern(".");
regExp.setPatternSyntax(Qt::WildcardSyntax);
```

为了实现此类 API，内部的对象需要被惰性构造。例如在 `QRegExp` 的案例中，不应该在还不知道表达式使用何种语法之前，就在 `setPattern()` 中过早地编译。表达式。

属性通常是级联的，此时我们应该谨慎地处理。仔细考虑下当前样式提供的“默认图标大小”与 `QToolButton` 的 `iconSize` 属性：

```
toolButton->iconSize(); // 返回当前样式表的默认大小
toolButton->setStyle(otherStyle);
toolButton->iconSize(); // 返回 otherStyle 的默认大小
toolButton->setIconSize(QSize(52, 52));
toolButton->iconSize(); // 返回 (52, 52)
toolButton->setStyle(yetAnotherStyle);
toolButton->iconSize(); // 返回 (52, 52)
```

注意，一旦 `iconSize` 被设置，它会被一直留存，此时修改当前样式不会影响它。这是好的。有时，提供重置属性的渠道会很方便，这有两种实现方式：

- 传递一个特定值（如 `QSize()`、`-1` 或 `Qt::Alignment(0)`）来指代“重置”；
- 提供显示的 `resetFoo()` 或 `unsetFoo()` 函数。

对于 `iconSize`，将 `QSize()`（即 `QSize(-1, -1)`）设为“重置”便足够了。

某些场景中，取值方法会返回值会与设置的内容不同。例如，若调用 `widget->setEnabled(true)`，可能通过 `widget->isEnabled()` 获得的依然是 `false`，因为父控件被禁用了。这没问题，因为通常这正是我们要检查的状态（父控件被禁用时，子控件也应该变灰，表现为也被禁用，但同时

在它内部，应该知道自己实际是“可用”的，并等待父控件可用后恢复状态)，但必须在文档中正确地进行描述。

## QProperty

译者注：该类型为 Qt 6.0 引入，需要参见 Qt 6 类文档。

本文原文中的内容与现有的 Qt 6.0 预览版存在出入，因此暂不翻译本节，待官方进一步维护更新原文。

## C++ 特性

### 值与对象

译者注：此条原文无内容，待官方更新

### 指针与引用

作为输出参数，指针与引用哪个更好？

```
void getHsv(int *h, int *s, int *v) const void getHsv(int &h, int &s, int &v) const
```

绝大多数 C++ 数据都推荐尽可能使用引用，因为引用在感知上比指针“更安全更漂亮”。事实上，我们在 Qt 软件中更倾向于使用指针，因为这会令代码更加已读。如对比：

```
color.getHsv(&h, &s, &v);  
color.getHsv(h, s, v);
```

第一行代码能很清晰地表示，h、s、v 对象有很大概率会被该函数调用所修改。

即便如此，由于编译器并不喜欢输出参数，在新 API 中应该避免此用法，而是返回一个小结构体：

```
struct Hsv { int hue, saturation, value }; Hsv getHsv() const;
```

译者注：对于可能失败的带返回值的函数，Qt 倾向于返回数值，使用 `bool* ok = 0` 参数来存储调用结果，以便在不关心时忽略之。同样在 Qt 6 以后，该方式不再被建议使用，而是改用 `std::optional<T>` 返回类型。

传递常引用与传递值若类型大于 16 字节，传递常引用。

若类型具有非平凡的拷贝构造函数或非平凡的析构函数，传递常引用来避免执行这些方法。

所有其它类型都应使用值传递。

范例：

```
void setAge(int age);  
void setCategory(QChar cat);  
void setName(QLatin1String name);  
void setAlarm(const QSharedPointer<Alarm> &alarm); // 常引用远快于拷贝构造和析构  
// QDate, QTime, QPoint, QPointF, QSize, QSizeF, QRect 都是其它应该值传递的好例子
```

## 虚函数

当 C++ 中的一个成员函数被声明为虚函数，这主要用于通过在子类中重写来自定义该函数的行为。将函数设为虚函数的目的是让对该函数的现有调用会被替代为访问自定义的代码分支。若在该类之外没人调用此函数，则在将其声明为虚函数之前需要小心斟酌：

```
// Qt 3 中的 QTextEdit: 成员函数不需要作为虚函数的成员函数
virtual void resetFormat();
virtual void setUndoDepth( int d );
virtual void setFormat( QTextFormat &f, int flags );
virtual void ensureCursorVisible();
virtual void placeCursor( const QPoint &pos, QTextCursorc = 0 );
virtual void moveCursor( CursorAction action, bool select );
virtual void doKeyboardAction( KeyboardAction action );
virtual void removeSelectedText( int selNum = 0 );
virtual void removeSelection( int selNum = 0 );
virtual void setCurrentFont( const QFont &f );
virtual void setOverwriteMode( bool b ) { overWrite = b; }
```

当 QTextEdit 从 Qt 3 迁移至 Qt 4 时，几乎所有虚函数都被移除。有趣的是（但并未预料到），并没有大量的抱怨。为什么？因为 Qt 3 并未使用 QTextEdit 的多态性，Qt 3 并不会调用这些函数——只有使用者会。简单来说，否则并没有任何理由去继承 QTextEdit 并重写这些方法——除非您自己会通过多态去调用它们。若您需要在您的应用程序，也就是 Qt 之外使用多态机制，您应该自行添加。

#### 避免使用虚函数

在 Qt 中，我们因为多种原因而尝试最小化虚函数的数量。每个虚函数调用都会让缺陷修复变得更难，因为会在调用图中插入一个不受控制的节点（使得调用结果无法预测）。人们会在虚函数中做非常疯狂的举措，例如：

- 发送事件
- 发送信号
- 重入事件循环（例如，打开一个模态文件对话框）
- 删除对象（例如，某些导致 `delete this` 的操作）

此外还有一些避免过度使用虚函数的原因：

- 无法在不破坏二进制兼容性的前提下增加、移动或删除虚函数
- 无法简便地重写虚函数
- 编译器通常几乎不会内联虚函数调用
- 调用虚函数需要查询虚表，导致其比常规函数调用慢 2-3 倍
- 虚函数另类对象更难进行值拷贝（可能做到，但会表现得很混乱且不被推荐）

过去的经验告诉我们，没有虚函数地类会产生更少的错误，通常也导致更少的维护。

一个通用的经验法则是，除非从工具集或该类的主要使用者角度需要调用它，否则一个函数不应该是虚函数。

虚对象与可复制性

多态对象和值类型的类并不是好朋友。

包含虚函数的类会有虚析构函数来避免基类析构时未清理子类数据导致的内存泄漏。

若需要以值语义拷贝、复制和对比一个类，则可能需要拷贝构造函数、赋值运算符重载和等于运算符重载：

```
class CopyClass {
public:
    CopyClass();
    CopyClass(const CopyClass &other);
    ~CopyClass();
    CopyClass &operator=(const CopyClass &other);
    bool operator==(const CopyClass &other) const;
    bool operator!=(const CopyClass &other) const;
    virtual void setValue(int v);
};
```

若创建该类的子类，则代码中会开始发生意料外的行为。通常来说，若没有虚函数和虚构造函数，则人们无法创建依赖于多态特性的子类。因此一旦虚函数或虚析构函数被添加，这会马上成为建立子类的理由，事情从此变得复杂。乍一看来，很容易觉得可以简单定义一下虚运算符重载。但顺着这条路深入下去，会导致混乱和毁灭（例如无可读性的代码）。请研究下这段代码：

```
class OtherClass {
public:
    const CopyClass &instance() const; // 此处会返回什么？我应该将返回值赋值给谁？
};
```

（此小节正在施工中）

不变性

C++ 提供了 `const` 关键字来标识不会改变或不会产生副作用的事物。它可被用于数值、指针和倍只想的内容，也可被作为成员函数的特殊属性来标识它不会修改对象的状态。

注意，`const` 自身并不提供太大的价值——许多语言甚至并未提供 `const` 关键字，但这并不会自动导致它们存在缺陷。事实上，若移除所有函数重载，并通过搜索替换移除 C++ 代码中的所有 `const` 标识，代码很可能依然能够编译并正确执行。使用实用主义导向来使用 `const` 是很重要的。

让我们看看 Qt 中使用 `const` 的 API 设计：

输入参数：`const` 指针

使用指针输入参数的 `const` 成员函数几乎总是使用 `const` 指针。

若一个成员函数确实被声明为 `const`，这意味着它不具有副作用，也不会修改对象对外可见的状态。那么，为什么要需要非 `const` 的输入参数？需要牢记，`const` 成员函数经常会被其它 `const` 成员函数，在这些调用场合中，非 `const` 的指针并不容易得到（除非使用 `const_cast`，而我们应该尽可能避免使用它）。

修改之前：

```
bool QWidget::isVisibleTo(const QWidget *ancestor) const;
bool QWidget::isEnabledTo(const QWidget *ancestor) const;
QPoint QWidget::mapFrom(const QWidget *ancestor, const QPoint &pos) const;
```

注意，我们在 `QGraphicsItem` 中修复了这些成员函数，但 `QWidget` 的修复需要等待 Qt 5:

```
bool isVisibleTo(const QGraphicsItem *parent) const;
QPointF mapFromItem (const QGraphicsItem *item, const QPointF &point) const;
```

返回值：`const` 值

函数的返回值，要么是引用类型，要么是右值。

非类类型的右值是不受 `cv` 限定符影响的。因此，即使在语法上允许为其添加 `const` 修饰，这并不会产生效果，因为由于其访问权不允许对其做出修改。大多数现代编译器在编译此类代码时会打印警告信息。

当为类类型的右值添加 `const` 时，对该类的非 `const` 的成员函数的访问会被禁止，对其成员变量的直接操作也会被禁止。

不添加 `const` 允许此类操作，但很少有此类需求，因为这些修改会伴随右值对象生命周期的结束而消失，这会在当前语句的分号结束后发生。

例如：

```
struct Foo {
    void setValue(int v) { value = v; } int value;
};

Foo foo() { return Foo(); }
const Foo cfoo() { return Foo(); }
int main() {
    // 下述代码可以编译，foo() 返回非 const 右值，无法
    // 成为赋值目标（这通常需要左值），但对成员变量的访问
    // 是左值：
    foo().value = 1; // 可以编译，但该临时值在这个完整的表达式结束后会被抛弃

    // 下述代码可以编译，foo() 返回非 const 右值，无法
    // 成为赋值目标，但可以调用（甚至于非 const 的）成员函数：
    foo().setValue(1); // 可以编译，但该临时值在这个完整的表达式结束后会被抛弃

    // 下述代码无法编译，cfoo() 返回 const 右值，因此其
    // 成员变量是 const 授权，无法被赋值：
    cfoo().value = 1; // 无法编译

    // 下述代码无法编译，cfoo() 返回 const 右值，无法调用
    // 其非 const 的成员函数：
    cfoo().setValue(1); // 无法编译
}
```

返回值：指针与 `const` 指针

`const` 成员函数应该返回指针还是 `const` 指针这个问题，令许多人发现 C++ 的“`const` 正确性”被瓦解了。该问题源于某些 `const` 成员函数，并不修改它们的内部状态，而是返回成员变量的非 `const` 指针。单纯返回一个指针并不会影响对象对外可见的状态，也不会修改它正在维护的状态。但这会令程序员获得间接地修改对象数据的权限。

下述范例展示了通过 `const` 成员函数返回的非 `const` 指针来规避不可变性的诸多方法之一：

```
QVariant CustomWidget::inputMethodQuery(Qt::InputMethodQuery query) const {  
    moveBy(10, 10); // 无法编译!  
    window()->childAt(mapTo(window(), rect().center()))->moveBy(10, 10); // 可以编译!  
}
```

返回 `const` 指针的函数，至少在一定程度上，避免了此类（可能并不希望/非预期的）副作用。但哪些函数会考虑返回 `const` 指针，或一组 `const` 指针？若我们使用 `const` 正确的方案，即令任何 `const` 成员函数返回成员变量（或一组成员变量的指针）时都是用 `const` 指针形式。很不幸的是，实际中这会造就无法使用的 API：

```
QGraphicsScene scene;  
// ... 初始化场景  
foreach (const QGraphicsItem *item, scene.items()) {  
    item->setPos(qrand() % 500, qrand() % 500); // 无法编译! item 是 const 指针  
}
```

`QGraphicsScene::items()` 是 `const` 成员函数，这可能会让人觉得应该返回 `const` 指针。

在 Qt 中，我们近乎只使用非 `const` 的模式。我们选择了实用主义之路：返回 `const` 指针更容易导致 `const_cast` 的过度使用，这比滥用返回的非 `const` 指针引发的问题更加频繁。

返回类型：值或 `const` 引用？

如果我们在返回对象时还保留了它的副本，返回 `const` 引用是最快的方法；然而，这在我们之后打算重构这个类时成为了限制（使用 `d` 指针惯用法，我们可以在任何时候修改 Qt 类的内存结构；但我们无法在不破坏二进制兼容性的前提下，将函数签名从 `const QFoo&` 改为 `QFoo`）。出于此原因，我们通常返回 `QFoo` 而非 `const QFoo&`，除了性能极端敏感，而重构并不是问题的少数场合（例如 `QList::at()`）。

`const` 与对象的状态

`const` 正确性是 C 中的一场“圣战”（译者注：原文为 `vi-emacs discussion`），因为该原则在一些领域（如基于指针的函数）中是失效了。

但 `const` 成员函数的常规含义是值不会修改一个类对外可见的状态，状态在此处值“我自己的和我负责的”。这并不意味着 `const` 成员函数会改变它们自己的私有成员变量，但也不代表不能这么做。但通常来说，`const` 成员函数不会产生可见的副作用。例如：

```
QSize size = widget->sizeHint(); // const  
widget->move(10, 10); // 非 const
```

代理对象负责处理对另一个对象的绘制工作，它的状态包含了它负责的内容，也就是包含它的绘制目标的状态。请求代理进行绘制是具有副作用的：这会改变正在被绘制的设备的外观（也意味着状态）。正因如此，令 `paint()` 成为 `const` 并不合理。任何视图控件或 `QIcon` 的 `paint()` 作

为 `const` 都很不合理。没人会在 `const` 成员函数中去调用 `QIcon::paint()`，除非他们明确地像规避当前函数的 `const` 性质。而在这种场景中，显示的 `const_cast` 会是更好的选择：

```
// QAbstractItemDelegate::paint 是 const
void QAbstractItemDelegate::paint(QPainter *painter,
const QStyleOptionViewItem &option,
const QModelIndex &index) const

// QGraphicsItem::paint 不是 const
void QGraphicsItem::paint(QPainter &painter,
const QStyleOptionGraphicsItem &option,
QWidget &widget = 0)
```

`const` 关键字不会为你做任何事，考虑将其移除，而非为一个成员函数提供 `const`/非 `const` 的重载版本。





## 第三章 QAbstractAnimation

**QAbstractAnimation** 是所有的动画相关的基类。

**QAbstractAnimation** 定义了所有动画类相关的基础功能，通过继承该类，您可以实现动画的其它功能，或者添加自定义的特效。

属性	方法
头文件	<code>#include&lt;QAbstractAnimation&gt;</code>
qmake	<code>QT+=core</code>
自从	Qt 4.6
继承	<code>QObject</code>
派生	<code>QAnimationGroup</code> , <code>QPauseAnimation</code> , <code>QVariantAnimation</code>

公共成员类型

类型	方法
enum	<code>DeletionPolicy { KeepWhenStopped, DeleteWhenStopped }</code>
enum	<code>Direction { Forward, Backward }</code>
enum	<code>State { Stopped, Paused, Running }</code>

属性

属性	类型	属性	类型
<code>currentLoop</code>	<code>const int</code>	<code>duration</code>	<code>const int</code>
<code>currentTime</code>	<code>int</code>	<code>loopCount</code>	<code>int</code>
<code>direction</code>	<code>Direction</code>	<code>state</code>	<code>const State</code>

公共成员函数

返回类型	函数名
	QAbstractAnimation(QObject *parent = Q_NULLPTR)
virtual	~QAbstractAnimation()
int	currentLoop() const
int	currentLoopTime() const
int	currentTime() const
Direction	direction() const
virtual int	duration() const = 0
QAnimationGroup *	group() const
int	loopCount() const
void	setDirection(Direction direction)
void	setLoopCount(int loopCount)
State	state() const
int	totalDuration() const

## 公共槽

返回类型	函数名
void	pause()
void	resume()
void	setCurrentTime(int msec)
void	setPaused(bool paused)
void	start(QAbstractAnimation::DeletionPolicy policy = KeepWhenStopped)
void	stop()

## 第四章 QAbstractAudioDeviceInfo

QAbstractAudioDeviceInfo 是音频后端的基类。

属性	方法
头文件	#include<QAbstractAudioDeviceInfo>
qmake	QT += multimedia
继承	QObject

简述

公共功能

类型	方法
virtual QString	deviceName() const = 0
virtual bool	isFormatSupported(const QAudioFormat &format) const = 0
virtual QAudioFormat	preferredFormat() const = 0
virtual QList<QAudioFormat::Endian>	supportedByteOrders() = 0
virtual QList	supportedChannelCounts() = 0
virtual QStringList	supportedCodecs() = 0
virtual QList	supportedSampleRates() = 0
virtual QList	supportedSampleSizes() = 0
virtual QList<QAudioFormat::SampleType>	supportedSampleTypes() = 0

类型

详细说明 QAbstractAudioDeviceInfo 是音频后端的基类。

该类实现了 QAudioDeviceInfo 的音频功能，即 QAudioDeviceInfo 类中会保留一个 QAbstractAudioDeviceInfo，并对其进行调用。关于 QAbstractAudioDeviceInfo 的实现的功能，您可以参考 QAudioDeviceInfo 的类与函数文档

成员函数文档

QString QAbstractAudioDeviceInfo::deviceName() const [纯虚函数]

返回音频设备名称

bool QAbstractAudioDeviceInfo::isFormatSupported(const QAudioFormat &format) const [纯虚函数]

传入参数 QAudioFormat（音频流）类，如果 QAbstractAudioDeviceInfo 支持的话，返回 true（真是不好翻译）

QAudioFormat QAbstractAudioDeviceInfo::preferredFormat() const [纯虚函数]  
返回 QAbstractAudioDeviceInfo 更加倾向于使用的音频流。

`QList<QAudioFormat::Endian> QAbstractAudioDeviceInfo::supportedByteOrders()`

[纯虚函数] 返回当前支持可用的字节顺序 (`QAudioFormat::Endian`) 列表

`QList QAbstractAudioDeviceInfo::supportedChannelCounts()` [纯虚函数]

返回当前可用的通道 (应该是这样翻译) 列表

`QStringList QAbstractAudioDeviceInfo::supportedCodecs()` [纯虚函数]

返回当前可用编解码器的列表

`QList QAbstractAudioDeviceInfo::supportedSampleRates()` [纯虚函数]

返回当前可用的采样率列表。(突然发现 Google 翻译真心吊啊)

`QList QAbstractAudioDeviceInfo::supportedSampleSizes()` [纯虚函数]

返回当前可用的样本大小列表。

`QList<QAudioFormat::SampleType> QAbstractAudioDeviceInfo::supportedSampleTypes()`  
[纯虚函数]

返回当前可用样本类型的列表。

## 第五章 QAbstractAudioInput

QAbstractAudioInput 类为 QAudioInput 类提供了访问音频设备的方法。(通过插件的形式)

属性	方法
头文件	<code>#include &lt;QAbstractAudioInput&gt;</code>
qmake	<code>QT += multimedia</code>
继承	QObject

简述

公有函数

类型	方法
virtual int	bufferSize() const = 0
virtual int	bytesReady() const = 0
virtual qint64	elapsedUSecs() const = 0
virtual QAudio::Error	error() const = 0
virtual QAudioFormat	format() const = 0
virtual int	notifyInterval() const = 0
virtual int	periodSize() const = 0
virtual qint64	processedUSecs() const = 0
virtual void	reset() = 0
virtual void	resume() = 0
virtual void	setBufferSize(int value) = 0
virtual void	setFormat(const QAudioFormat &fmt) = 0
virtual void	setNotifyInterval(int ms) = 0
virtual void	setVolume(qreal) = 0
virtual void	start(QIODevice *device) = 0
virtual QIODevice*	start() = 0
virtual QAudio::State	state() const = 0
virtual void	stop() = 0
virtual void	suspend() = 0
virtual qreal	volume() const = 0

信号

类型	方法
void	errorChanged(QAudio::Error error)
void	notify()
void	stateChanged(QAudio::State state)

#### 详细描述

`QAbstractAudioInput` 类为 `QAudioInput` 类提供了访问音频设备的方法。（通过插件的形式）`QAudioDeviceInput` 类中保留了一个 `QAbstractAudioInput` 的实例，并且调用的函数与 `QAbstractAudioInput` 的一致。

译者注：也就是说 `QAudioDeviceInput` 调用的函数实际上是 `QAbstractAudioInput` 的函数，就封装了一层相同函数名吧。可以自己看看源码。）

这意味着 `QAbstractAudioInput` 是实现音频功能的。有关功能的描述，可以参考 `QAudioInput` 类。另见 `QAudioInput` 函数

#### 成员函数文档

`int QAbstractAudioInput::bufferSize() const` [纯虚函数]

以毫秒为单位返回音频缓冲区的大小

`int QAbstractAudioInput::bytesReady() const` [纯虚函数]

以字节（bytes）为单位返回可读取的音频数据量

`qint64 QAbstractAudioInput**::elapsedUSecs() const` [纯虚函数]

返回调用 `start()` 函数以来的毫秒数，包括空闲时间与挂起状态的时间

`QAudio::Error QAbstractAudioInput::error() const` [纯虚函数]

返回错误的状态

`void QAbstractAudioInput::errorChanged(QAudio::Error error)` [信号 signal]

当错误状态改变时，该信号被发射

`QAudioFormat QAbstractAudioInput::format() const` [纯虚函数]

返回正在使用的 `QAudioFormat`（这个类是储存音频流相关的参数信息的）另参见 `setFormat()` 函数

`void QAbstractAudioInput::notify()` [信号 signal]

当音频数据的 `x ms` 通过函数 `setNotifyInterval()` 调用之后，这个信号会被发射。

`int QAbstractAudioInput::notifyInterval() const` [纯虚函数]

以毫秒为单位返回通知间隔

`int QAbstractAudioInput::periodSize() const` [纯虚函数]

以字节为单位返回其周期

`qint64 QAbstractAudioInput::processedUSecs() const` [纯虚函数]

返回自 `start()` 函数被调用之后处理的音频数据量（以毫秒为单位）

`void QAbstractAudioInput::reset()` [纯虚函数]

将所有音频数据放入缓冲区，并将缓冲区重置为零

`void QAbstractAudioInput::resume()` [纯虚函数]

在音频数据暂停后继续处理

`void QAbstractAudioInput::setBufferSize(int value)` [纯虚函数]

将音频缓冲区大小设置为 `value` 大小（以毫秒为单位）另参阅 `bufferSize()` 函数

`void QAbstractAudioInput::setFormat(const QAudioFormat &fmt)` [纯虚函数]

设置音频格式,设置格式的时候只能在 `QAudio` 的状态为 `StoppedState` 时 (`QAudio::StoppedState`)

`void QAbstractAudioInput::setNotifyInterval(int ms)` [纯虚函数]

设置发送 `notify()` 信号的时间间隔。这个 `ms` 时间间隔与操作系统平台相关，并不是实际的 `ms` 数。

`void QAbstractAudioInput::setVolume(qreal)` [纯虚函数]

另见 `volume()` 函数(设置这里应该是设置音量的值,Volume 在英文中有音量的意思,官方文档这里根本就没有任何说明,说去参考 `valume()` 函数,可是 `valume()` 说又去参考 `SetValume()` 函数,这是互相甩锅的节奏么??? 坑爹啊!!!)

`void QAbstractAudioInput::start(QIODevice *device)` [纯虚函数]

使用输入参数 `QIODevice *device` 来传输数据

`QIODevice *QAbstractAudioInput::start()` [纯虚函数]

返回一个指向正在用于正在处理数据 `QIODevice` 的指针。这个指针可以用来直接读取音频数据。

`QAudio::State QAbstractAudioInput::state() const` [纯虚函数]

返回处理音频的状态

`void QAbstractAudioInput::stateChanged(QAudio::State state)` [信号 signal]

当设备状态改变时，会发出这个信号

`void QAbstractAudioInput::stop()` [纯虚函数]

停止音频输入（因为这是个 `QAbstractAudioInput` 类啊，输入类啊，暂时这么解释比较合理。）

`void QAbstractAudioInput::suspend()` [纯虚函数]

停止处理音频数据，保存缓冲的音频数据

`qreal QAbstractAudioInput::volume() const` [纯虚函数]

另见 `setVolume()` (内心 os: 参考我解释 `setVolume()` 函数的说明, 这里应该是返回其音量)





## 第六章 QAbstractAudioOutput

QAbstractAudioOutput 类是音频后端的基类

属性	方法
头文件	#include <QAbstractAudioOutput>
qmake	QT += multimedia
继承	QObject

简述

public 公有函数

类型	方法
virtual int	bufferSize() const = 0
virtual int	bufferSize() const = 0
virtual int	bytesFree() const = 0
virtual QString	category() const
virtual qint64	elapsedUSecs() const = 0
virtual QAudio::Error	error() const = 0
virtual QAudioFormat	format() const = 0
virtual int	notifyInterval() const = 0
virtual int	periodSize() const = 0
virtual qint64	processedUSecs() const = 0
virtual void	reset() = 0
virtual void	resume() = 0
virtual void	setBufferSize(int value) = 0
virtual void	setCategory(const QString &)
virtual void	setFormat(const QAudioFormat &fmt) = 0
virtual void	setNotifyInterval(int ms) = 0
virtual void	setVolume(qreal volume)
virtual void	start(QIODevice *device) = 0
virtual QIODevice *	start() = 0
virtual QAudio::State	state() const = 0
virtual void	stop() = 0
virtual void	suspend() = 0
virtual qreal	volume() const

信号

类型	函数名
void	errorChanged(QAudio::Error error)
void	notify()
void	stateChanged(QAudio::State state)

### 详细描述

QAbstractAudioOutput 类是音频后端的基类。QAbstractAudioOutput 类是 QAudioOutput 类的实现类。QAudioOutput 的实现实际上是调用的 QAbstractAudioOutput 类，有关实现相关的功能，请参考 QAudioOutput() 类中的函数说明。

### 成员函数

`int QAbstractAudioOutput::bufferSize() const` [纯虚函数]

以字节为单位，返回音频缓冲区的大小。

另见 `setBufferSize()` 函数

`int QAbstractAudioOutput::bytesFree()const` [纯虚函数]

返回音频缓冲区的可用空间（以字节为单位）

`QString QAbstractAudioOutput::category() const` [虚函数 virtual]

音频缓冲区的类别(官方文档没有,这是我个人经验,当然可能有误,望指正)另见 `setCategory()`

`qint64 QAbstractAudioOutput::elapsedUSecs() const` [纯虚函数 pure virtual]

返回调用 `start()` 函数之后的毫秒数，包括处于空闲状态的时间和挂起状态的时间。

`QAudio::Error QAbstractAudioOutput::error() const` [纯虚函数 pure virtual]

返回错误状态。

`void QAbstractAudioOutput::errorChanged(QAudio::Error error)` [信号 signal]

当错误状态改变时，该信号被发射。

`QAudioFormat QAbstractAudioOutput::format()const` [纯虚函数 pure virtual]

返回正在使用的 `QAudioFormat()` 类另见 `setFormat()`

`void QAbstractAudioOutput::notify()` [信号 signal]

当函数 `setNotifyInterval(x)` 函数已经调用，即音频数据的时间间隔已经被设置时。该信号被发射。（就是调用 `setNotifyInterval(x)` 后，这个信号会被发射。官方文档讲的好详细啊 =。=）

`int QAbstractAudioOutput::notifyInterval() const` [纯虚函数 pure virtual]

以毫秒为单位，返回时间间隔另见函数 `setNotifyInterval()`

`int QAbstractAudioOutput::periodSize() const` [纯虚函数 pure virtual]

以字节为单位返回周期大小。

```
qint64 QAbstractAudioOutput::processedUSecs() const [纯虚函数 pure virtual]
```

返回自调用 `start()` 函数后处理的音频数据量（单位为毫秒）

```
void QAbstractAudioOutput::reset() [纯虚函数 pure virtual]
```

将所有音频数据放入缓冲区，并将缓冲区重置为零。

```
void QAbstractAudioOutput::resume() [纯虚函数 pure virtual]
```

继续处理暂停后的音频数据（也就是暂停后继续的意思呗）

```
void QAbstractAudioOutput::setBufferSize(int value) [纯虚函数 pure virtual]
```

重新设置音频缓冲区的大小（以字节为单位即输入参数 `value`）另见 `bufferSize()` 函数

```
void QAbstractAudioOutput::setCategory(const QString &)
```

[虚函数 `virtual`] 参见函数 `category()`

```
void QAbstractAudioOutput::setFormat(const QAudioFormat &fmt) [纯虚函数 pure virtual]
```

`QAbstractAudioOutput` 设置 `QAudioFormat` 类,只有当 `QAudio` 状态为 `QAudio::StoppedState` 时，音频格式才会被设置成功。另见函数 `format()`

```
void QAbstractAudioOutput::setNotifyInterval(int ms) [纯虚函数 pure virtual]
```

设置发送 `notify()` 信号的时间间隔。这个 `ms` 并不是实时处理的音频数据中的 `ms` 数。这个时间间隔是平台相关的。另见 `notifyInterval()`

```
void QAbstractAudioOutput::setVolume(qreal volume) [虚函数 virtual]
```

设置音量。音量的范围为 `[0.0 - 1.0]`。另见函数 `volume()`

```
void QAbstractAudioOutput::start(QIODevice *device) [纯虚函数 pure virtual]
```

调用 `start()` 函数时，输入参数 `QIODevice*` 类型的变量 `device`，用于音频后端处理数据传输。

```
QIODevice *QAbstractAudioOutput::start() [纯虚函数 pure virtual]
```

返回一个指向正在处理数据传输的 `QIODevice` 类型的指针，这个指针是可以被写入的，用于处理音频数据。（参考上边的函数是咋写入的）

```
QAudio::State QAbstractAudioOutput::state() const [纯虚函数 pure virtual]
```

返回音频处理的状态。

```
void QAbstractAudioOutput::stateChanged(QAudio::State state) [信号 signal]
```

当音频状态变化的时候，该信号被发射

```
void QAbstractAudioOutput::stop() [纯虚函数 pure virtual]
```

停止音频输出

```
void QAbstractAudioOutput::suspend() [纯虚函数 pure virtual]
```

停止处理音频数据，保存处理的音频数据。（就是暂停的意思啊 =。=）

```
qreal QAbstractAudioOutput::volume() const [虚函数 virtual]
```

返回音量。

音量范围为 [0.0 - 1.0] 另参阅函数 `setVolume()`

## 第七章 QAbstractAxis

**QAbstractAxis** 类是用于专门处理坐标轴的类

属性	方法
头文件	<code>#include &lt;QAbstractAxis&gt;</code>
实例化	<code>AbstractAxis</code>
继承	<code>QObject</code>
派生	<code>QBarCategoryAxis</code> , <code>QDateTimeAxis</code> , <code>QLogValueAxis</code> , and <code>QValueAxis</code>

简述

公共类型

类型	方法
enum	<code>AxisType</code> <code>AxisTypeNoAxis</code> , <code>AxisTypeValue</code> , <code>AxisTypeBarCategory</code> , <code>AxisTypeCatego</code>
flags	<code>AxisTypes</code>

属性

函数名	类型
alignment :	const Qt::Alignment
color :	QColor
gridLineColor :	QColor
gridLinePen :	QPen
gridVisible :	bool
labelsAngle :	int
labelsBrush :	QBrush
labelsColor :	QColor
labelsFont :	QFont
labelsVisible :	bool
linePen :	QPen
lineVisible :	bool
minorGridLineColor :	QColor
minorGridLinePen :	QPen
minorGridVisible :	bool
orientation :	const Qt::Orientation
reverse :	bool
shadesBorderColor :	QColor
shadesBrush :	QBrush
shadesColor :	QColor
shadesPen :	QPen
shadesVisible :	bool
titleBrush :	QBrush
titleFont :	QFont
titleText :	QString
titleVisible :	bool
visible :	bool

## 公共函数

类型	函数名
	~QAbstractAxis()
Qt::Alignment	alignment() const
QColor	gridLineColor()
QPen	gridLinePen() const
void	hide()
bool	isGridLineVisible() const
bool	isLineVisible() const
bool	isMinorGridLineVisible() const
bool	isReverse() const
bool	isTitleVisible() const
bool	isVisible() const
int	labelsAngle() const

QBrush	labelsBrush() const
QColor	labelsColor() const
QFont	labelsFont() const
bool	labelsVisible() const
QPen	linePen() const
QColor	linePenColor() const
QColor	minorGridLineColor()
QPen	minorGridLinePen() const
Qt::Orientation	orientation() const
void	setGridLineColor(const QColor &color)
void	setGridLinePen(const QPen &pen)
void	setGridLineVisible(bool visible = true)
void	setLabelsAngle(int angle)
void	setLabelsBrush(const QBrush &brush)
void	setLabelsColor(QColor color)
void	setLabelsFont(const QFont &font)
void	setLabelsVisible(bool visible = true)
void	setLinePen(const QPen &pen)
void	setLinePenColor(QColor color)
void	setLineVisible(bool visible = true)
void	setMax(const QVariant &max)
void	setMin(const QVariant &min)
void	setMinorGridLineColor(const QColor &color)
void	setMinorGridLinePen(const QPen &pen)
void	setMinorGridLineVisible(bool visible = true)
void	setRange(const QVariant &min, const QVariant &max)
void	setReverse(bool reverse = true)
void	setShadesBorderColor(QColor color)
void	setShadesBrush(const QBrush &brush)
void	setShadesColor(QColor color)
void	setShadesPen(const QPen &pen)
void	setShadesVisible(bool visible = true)
void	setTitleBrush(const QBrush &brush)
void	setTitleFont(const QFont &font)
void	setTitleText(const QString &title)
void	setTitleVisible(bool visible = true)
void	setVisible(bool visible = true)
QColor	shadesBorderColor() const
QBrush	shadesBrush() const
QColor	shadesColor() const
QPen	shadesPen() const
bool	shadesVisible() const

void	show()
QBrush	titleBrush() const
QFont	titleFont() const
QString	titleText() const
virtual AxisType	type() const = 0

信号

类型	函数名
void	colorChanged(QColor color)
void	gridLineColorChanged(const QColor &color)
void	gridLinePenChanged(const QPen &pen)
void	gridVisibleChanged(bool visible)
void	labelsAngleChanged(int angle)
void	labelsBrushChanged(const QBrush &brush)
void	labelsColorChanged(QColor color)
void	labelsFontChanged(const QFont &font)
void	labelsVisibleChanged(bool visible)
void	linePenChanged(const QPen &pen)
void	lineVisibleChanged(bool visible)
void	minorGridLineColorChanged(const QColor &color)
void	minorGridLinePenChanged(const QPen &pen)
void	minorGridVisibleChanged(bool visible)
void	reverseChanged(bool reverse)
void	shadesBorderLineColorChanged(QColor color)
void	shadesBrushChanged(const QBrush &brush)
void	shadesColorChanged(QColor color)
void	shadesPenChanged(const QPen &pen)
void	shadesVisibleChanged(bool visible)
void	titleBrushChanged(const QBrush &brush)
void	titleFontChanged(const QFont &font)
void	titleTextChanged(const QString &text)
void	titleVisibleChanged(bool visible)
void	visibleChanged(bool visible)

详细说明

**QAbstractAxis** 是专门用于处理坐标轴的基类。

每一个连续的序列可以绑定到一个或者多个水平轴和垂直轴，但是不同域的混合轴的类型是不支持的。比如在同个方向指定 **QValueAxis** 和 **QLogValueAxis**。

每个轴的元素（比如轴线，标题，标签，网格线，阴影，可见性）都是可以控制的。

成员变量 `enum QAbstractAxis::AxisType flags QAbstractAxis::AxisTypes`



这个枚举类型指定了轴对象的类型

变量	值
<code>QAbstractAxis::AxisTypeNoAxis</code>	<code>0x0</code>
<code>QAbstractAxis::AxisTypeValue</code>	<code>0x1</code>
<code>QAbstractAxis::AxisTypeBarCategory</code>	<code>0x2</code>
<code>QAbstractAxis::AxisTypeCategory</code>	<code>0x4</code>
<code>QAbstractAxis::AxisTypeDateTime</code>	<code>0x8</code>
<code>QAbstractAxis::AxisTypeLogValue</code>	<code>0x10</code>

`AxisTypes` 是 `QFlags<AxisType>` 的 typedef。它是 `AxisType` 类型的组合。（也就是个宏呗）

`alignment : const Qt::Alignment` 该属性是轴的对齐属性其值可以为 `Qt::AlignLeft`, `Qt::AlignRight`, `Qt::AlignBottom`, or `Qt::AlignTop`. 相关函数

类型	函数名
<code>Qt::Alignment</code>	<code>alignment() const</code>

.....

`color : QColor` 该属性是指坐标轴与刻度的颜色

相关函数

类型	函数名
<code>QColor</code>	<code>linePenColor() const</code>
<code>void</code>	<code>setLinePenColor(QColor color)</code>

通知信号

类型	函数名
<code>void</code>	<code>colorChanged(QColor color)</code>

.....

`gridLinePen : QPen` 该属性是指绘制网格线的笔

相关函数

类型	函数名
<code>QPen</code>	<code>gridLinePen() const</code>
<code>void</code>	<code>setGridLinePen(const QPen &amp;pen)</code>

通知信号

类型	函数名
<code>void</code>	<code>gridLinePenChanged(const QPen &amp;pen)</code>

.....

`labelsAngle : int` 该属性以度数保存轴坐标的角度

相关函数

类型	函数名
int	labelsAngle() const
void	setLabelsAngle(int angle)

通知信号

类型	函数名
void	labelsAngleChanged(int angle)

.....

**labelsBrush** : QBrush 该属性表示用于绘制标签的画笔只有画刷的颜色是相关的（这句话其实我不太理解 Only the color of the brush is relevant.）

相关函数

类型	函数名
QBrush	labelsBrush() const
void	setLabelsBrush(const QBrush &brush)

通知信号

类型	函数名
void	labelsBrushChanged(const QBrush &brush)

.....

**labelsColor** : QColor 该属性表示轴标签的颜色

相关函数

类型	函数名
QColor	labelsColor() const
void	setLabelsColor(QColor color)

通知信号

类型	函数名
void	labelsColorChanged(QColor color)

.....

**labelsFont** : QFont 该属性表示轴标签的字体信息

相关函数

类型	函数名
QFont	labelsFont() const
void	setLabelsFont(const QFont &font)

通知信号

类型	函数名
void	labelsFontChanged(const QFont &font)

.....

**labelsVisible** : bool 该属性表示轴标签是否可见

## 相关函数

类型	函数名
bool	labelsVisible() const
void	setLabelsVisible(bool visible = true)

## 通知信号

类型	函数名
void	labelsVisibleChanged(bool visible)

.....

**linePen** : QPen 该属性表示绘制轴线的笔相关

## 相关函数

类型	函数名
QPen	linePen() const
void	setLinePen(const QPen &pen)

## 通知信号

类型	函数名
void	linePenChanged(const QPen &pen)

.....

**lineVisible** : bool 该属性表示轴线是否可见

## 相关函数

类型	函数名
bool	isLineVisible() const
void	setLineVisible(bool visible = true)

## 通知信号

类型	函数名
void	lineVisibleChanged(bool visible)

.....

**minorGridLineColor** : QColor 该属性表示副格线的颜色仅适用于支持副网格线的轴

## 相关函数

类型	函数名
QColor	minorGridLineColor()
void	setMinorGridLineColor(const QColor &color)

## 通知信号

类型	函数名
void	minorGridLineColorChanged(const QColor &color)

.....

**minorGridVisible** : bool 该属性表示副格线是否可见仅适用于支持副网格线的轴

## 相关函数

类型	函数名
bool	isMinorGridLineVisible() const
void	setMinorGridLineVisible(bool visible = true)

## 通知信号

类型	函数名
void	minorGridVisibleChanged(bool visible)

orientation : const Qt::Orientation 该属性表示坐标轴的方向。当坐标轴被添加到图表时，该属性为 Qt::Horizontal 或者 Qt::Vertical

## 相关函数

类型	函数名
Qt::Orientation	orientation() const

reverse : bool 该属性表示是否使用反转轴。该值默认为 false。反转轴由直线，样条，散列图系列以及笛卡尔图表组成的区域支持。如果一个方向相反，或者行为为定义，则所有相同方向的所有轴必须保持一致。

## 相关函数

类型	函数名
bool	isReverse() const
void	setReverse(bool reverse = true)

## 通知信号

类型	函数名
void	reverseChanged(bool reverse)

shadesBorderColor : QColor 该属性表示坐标轴阴影的边框（笔）颜色

## 相关函数

类型	函数名
QColor	shadesBorderColor() const
void	setShadesBorderColor(QColor color)

## 通知信号

类型	函数名
void	shadesBorderColorChanged(QColor color)

shadesBrush : QBrush 该属性表示用于绘制轴阴影的画笔（网格线之间的区域）

## 相关函数

类型	函数名
QPen	shadesPen() const
void	setShadesPen(const QPen &pen)

通知信号

类型	函数名
void	shadesPenChanged(const QPen &pen)

.....

**shadesVisible** : bool 该属性表示轴阴影是否可见

相关函数

类型	函数名
bool	shadesVisible() const
void	setShadesVisible(bool visible = true)

通知信号

类型	函数名
void	shadesVisibleChanged(bool visible)

.....

**titleBrush** : QBrush 该属性表示用于绘制坐标轴标题文本的画笔。只影响画刷的颜色。

相关函数

类型	函数名
QBrush	titleBrush() const
void	setTitleBrush(const QBrush &brush)

通知信号

类型	函数名
void	titleBrushChanged(const QBrush &brush)

.....

**titleFont** : QFont 该属性表示坐标轴标题的字体。

相关函数

类型	函数名
QFont	titleFont() const
void	setTitleFont(const QFont &font)

通知信号

类型	函数名
void	titleFontChanged(const QFont &font)

.....

**titleText** : QString 该属性表示坐标轴的标题默认为空，坐标轴的标题支持 HTML 的格式。

相关函数

类型	函数名
QString	titleText() const
void	setTitleText(const QString &title)

通知信号

类型	函数名
void	titleTextChanged(const QString &text)

.....

**titleVisible** : bool 该属性表示坐标轴的可见性。默认值为 **true**。

相关函数

类型	函数名
bool	isTitleVisible() const
void	setTitleVisible(bool visible = true)

通知信号

类型	函数名
void	titleVisibleChanged(bool visible)

.....

**visible** : bool 该属性表示坐标轴的可见性。

相关函数

类型	函数名
bool	isVisible() const
void	setVisible(bool visible = true)

通知信号

类型	函数名
bool	isVisible() const
void	visibleChanged(bool visible)

.....

成员函数

**QAbstractAxis::~QAbstractAxis()**

析构函数，销毁轴对象，当一个坐标轴添加到图表时，该图表获得该坐标轴的所有权。

**void QAbstractAxis::colorChanged(QColor color)** [信号]

当坐标轴的颜色变化时，该信号被发射。注意：属性 **color** 的通知信号。

**void QAbstractAxis::gridLineColorChanged(const QColor &color)** [信号]

当绘制网格线的笔的颜色改变时，该信号被发射。注意：属性 **gridLineColor** 的通知信号。

**QPen QAbstractAxis::gridLinePen() const**

返回用于绘制网格的笔。注意：属性 `gridLinePen` 的 Getter 函数。另参见函数 `setGridLinePen()`。

`void QAbstractAxis :: gridLinePenChanged (const QPen &pen) [信号]`

当用于绘制网格线的笔变化时，会发出此信号。注意：属性 `gridLinePen` 的通知信号。

`void QAbstractAxis::gridVisibleChanged(bool visible) [信号]`

当坐标轴的网格线的可见性变化时，发出该信号。注意：属性 `gridVisible` 的通知信号。

`void QAbstractAxis::hide()` 使坐标轴，阴影，标签，网格线不可见。

`void QAbstractAxis::labelsAngleChanged(int angle) [信号]`

当坐标轴标签的角度变化时，发出该信号。注意：属性标签角度的通知信号

`QBrush QAbstractAxis::labelsBrush() const`

返回用于绘制标签的画笔。注意：属性 `labelsBrush` 的 Getter 函数。另参见 `setLabelsBrush()`。

`void QAbstractAxis::labelsBrushChanged(const QBrush &brush) [信号]`

当用于绘制坐标轴标签的画笔改变时，会发出此信号。属性 `Brush` 的通知信号。

`void QAbstractAxis :: labelsColorChanged (QColor color) [信号 signal]`

当坐标轴标签的颜色改变时，会发出此信号。属性 `labelsColor` 的通知信号。

`QFont QAbstractAxis::labelsFont() const`

返回用于绘制标签的字体。注意：属性 `labelsFont` 的 Getter 函数。另参见函数 `setLabelsFont()`。

`void QAbstractAxis :: labelsFontChanged (const QFont &font) [信号]`

当坐标轴的字体改变时，会发出此信号。属性 `labelsFont` 的通知信号。

`void QAbstractAxis::labelsVisibleChanged(bool visible) [信号]`

当坐标轴标签的可见性变化时，会发出此信号。属性 `labelsVisible` 的通知信号。

`QPen QAbstractAxis::linePen() const`

返回用于绘制轴线与刻度线的笔。注意：属性 `linePen` 的 Getter 函数。另参阅函数 `setLinePen()`。

`void QAbstractAxis::linePenChanged(const QPen &pen) [信号]`

当绘制坐标轴的笔变化时，会发出此信号。注意：属性 `linePen` 的通知信号。

`void QAbstractAxis::lineVisibleChanged(bool visible) [信号]`

当坐标轴线的可见性变化时，会发出此信号。注意：属性 `lineVisible` 的通知信号。

`void QAbstractAxis::minorGridLineColorChanged(const QColor &color) [信号]`

当绘制副格线的笔的颜色变化时，该信号被发射。注意：属性 `minorGridLineColor` 的通知信号。

`void QAbstractAxis::minorGridLinePenChanged(const QPen &pen)` [信号]

当绘制副格线的笔变化时，该信号被发射。注意：属性 `minorGridLinePen` 的通知信号。

`void QAbstractAxis::minorGridVisibleChanged(bool visible)` [信号]

当绘制副格线的可见性变化时，该信号被发射。注意：属性 `minorGridVisible` 的通知信号。

`Qt::Orientation QAbstractAxis::orientation() const`

返回坐标轴的方向（垂直或者水平）注意：坐标轴方向的 `Getter` 函数。

`void QAbstractAxis::setGridLinePen(const QPen &pen)`

设置绘制网格线的笔。注意：`gridLinePen` 的 `Setter` 函数。另参见函数 `gridLinePen()`。

`void QAbstractAxis::setLabelsBrush(const QBrush &brush)`

设置用于绘制标签的画笔。注意：属性 `LabelsBrush` 的 `Setter` 函数。另参阅 `labelsBrush()`。

`void QAbstractAxis::setLabelsFont(const QFont &font)`

设置用于绘制标签的字体相关注意：属性 `labelsFont` 的 `Setter` 函数。另参阅函数 `labelsFont()`。

`void QAbstractAxis::setLinePen(const QPen &pen)`

设置用于绘制坐标轴线和刻度线的笔。注意：属性 `linePen` 的 `Setter` 函数。另参见函数 `linePen()`。

`void QAbstractAxis::setLineVisible(bool visible = true)`

设置坐标轴与刻度线是否可见。注意：属性 `lineVisible` 的 `Setter` 函数另参见函数 `isLineVisible()`。

`void QAbstractAxis::setMax(const QVariant &max)`

设置坐标轴上显示的最大值。根据当前坐标轴的类型，最大值参数会被转换为适当的值。如果转化失败，该函数设置无效。

`void QAbstractAxis::setMin(const QVariant &min)`

设置坐标轴上显示的最小值。根据当前坐标轴的类型，最小值参数会被转换为适当的值。如果转化失败，该函数设置无效。

`void QAbstractAxis::setRange(const QVariant &min, const QVariant &max)`

设置坐标轴的范围。根据当前坐标轴的类型，最大值最小值会被转换为适当的值。如果转化失败，该函数设置无效。

`void QAbstractAxis::setShadesBrush(const QBrush &brush)`



设置用于绘制阴影的画笔。注意：属性 `shadesBrush` 的 `Setter` 函数。另参见函数 `shadesBrush()`。

```
void QAbstractAxis::setShadesPen(const QPen &pen)
```

设置用于绘制阴影的笔。注意：属性 `shadesPen` 的 `Setter` 函数。另参见函数 `shadesPen()`。

```
void QAbstractAxis::setTitleBrush(const QBrush &brush)
```

设置用于绘制标题的画笔。注意：属性 `titleBrush` 的 `Setter` 函数。另参见函数 `titleBrush()`。

```
void QAbstractAxis::setTitleFont(const QFont &font)
```

设置用于绘制标题的笔。注意：属性 `titleFont` 的 `Setter` 函数。另参见函数 `titleFont()`。

```
void QAbstractAxis::setVisible(bool visible = true)
```

设置坐标轴，阴影，标签，网格线是否可见。注意：属性 `visible` 的 `Setter` 函数。另参见函数 `isVisible()`。

```
void QAbstractAxis::shadesBorderColorsChanged(QColor color) [信号 signal]
```

当绘制坐标轴边框的颜色变化时，会发出此信号。注意：属性 `shadesBorderColors` 的通知信号。

```
QBrush QAbstractAxis::shadesBrush() const
```

返回用于绘制阴影的画笔。注意：属性 `shadesBrush()` 的 `Getter` 函数。另参阅函数 `setShadesBrush()`。

```
void QAbstractAxis::shadesBrushChanged(const QBrush &brush) [信号 signal]
```

当绘制坐标轴阴影的画刷变化时，会发出此信号。注意：属性 `shadesBrush()` 的通知信号。

```
void QAbstractAxis::shadesColorChanged(QColor color) [信号 signal]
```

当绘制坐标轴阴影颜色发生变化时，会发出此信号。注意：属性 `shadesColor()` 的通知信号。

```
QPen QAbstractAxis::shadesPen() const
```

返回用于绘制阴影的笔。注意：属性 `shadesPen()` 的 `Getter` 函数另参阅函数 `setShadesPen()`。

```
void QAbstractAxis::shadesPenChanged(const QPen &pen) [信号 signal]
```

当绘制坐标轴阴影的笔发生变化时，会发出此信号。注意：属性 `shadesPen()` 的通知信号。

```
void QAbstractAxis::shadesVisibleChanged(bool visible) [信号 signal]
```

当坐标轴的阴影可见性变化时，会发出此信号。注意：属性 `shadesVisible()` 的通知信号。

```
void QAbstractAxis::show()
```

使坐标轴，阴影，标签，网格线可见。

```
QBrush QAbstractAxis::titleBrush() const
```

返回用于绘制标题的画刷注意：属性 `titleBrush` 的 Getter 函数另参阅函数 `setTitleBrush()`

```
void QAbstractAxis::titleBrushChanged(const QBrush &brush) [信号 signal]
```

当绘制坐标轴标题的画刷变化时，该信号被发射注意：`titleBrush` 的通知信号。

```
QFont QAbstractAxis::titleFont() const
```

返回绘制标题的笔相关属性。注意：`titleFont` 的 Getter 函数另参阅函数 `setTitleFont()`

```
void QAbstractAxis::titleFontChanged(const QFont &font) [信号 signal]
```

当坐标轴标题的字体属性更改时，会发出此信号。注意：`titleFont` 的通知信号。

```
void QAbstractAxis::titleTextChanged(const QString &text) [信号 signal]
```

当坐标轴标题的文本内容改变时，会发出此信号。注意：`titleText` 的通知信号。

```
void QAbstractAxis::titleVisibleChanged(bool visible) [信号 signal]
```

当坐标轴的标题文本的可见性变化时，会发出此信号。注意：`titleVisible` 的通知信号

```
AxisType QAbstractAxis::type() const
```

返回坐标轴的类型

```
void QAbstractAxis::visibleChanged(bool visible)
```

当坐标轴的可见性变化时，会发出此信号。注意：坐标轴的 `visible` 的通知信号

# 第八章 QAbstractBarSeries

QAbstractBarSeries 是所有柱状图/条形图系列的基类

属性	方法
头文件	#include<QAbstractBarSeries>
实例化	AbstractBarSeries
继承	QAbstractSeries
派生	QBarSeries, QHorizontalBarSeries, QHorizontalPercentBarSeries, QHorizontalStackedBarSeries

.....

简述

公共类型

类型	方法
enum	LabelsPosition LabelsCenter, LabelsInsideEnd, LabelsInsideBase, LabelsOutside

属性

函数名	类型
barWidth :	qreal
count :	const int
labelsAngle :	qreal
5 个属性继承自 QAbstractSeries	
1 个属性继承自 QObject	

.....

Public Functions

函数名	类型
virtual	~QAbstractBarSeries()
bool	append(QBarSet *set)
bool	append(QList<QBarSet *> sets)
QList<QBarSet *>	barSets() const
qreal	barWidth() const
void	clear()
int	count() const
bool	insert(int index, QBarSet *set)
bool	isLabelsVisible() const
qreal	labelsAngle() const
QString	labelsFormat() const
QAbstractBarSeries::LabelsPosition	labelsPosition() const
bool	remove(QBarSet *set)
void	setBarWidth(qreal width)
void	setLabelsAngle(qreal angle)
void	setLabelsFormat(const QString &format)
void	setLabelsPosition(QAbstractBarSeries::LabelsPosition position)
void	setLabelsVisible(bool visible = true)
bool	take(QBarSet *set)
15 个公共函数继承自 QAbstractSeries	
32 个公共函数继承自 QObject	

.....

信号

函数名	类型
void	barsetsAdded(QList<QBarSet *> sets)
void	barsetsRemoved(QList<QBarSet *> sets)
void	clicked(int index, QBarSet *barset)
void	countChanged()
void	doubleClicked(int index, QBarSet *barset)
void	hovered(bool status, int index, QBarSet *barset)
void	labelsAngleChanged(qreal angle)
void	labelsFormatChanged(const QString &format)
void	labelsPositionChanged(QAbstractBarSeries::LabelsPosition position)
void	labelsVisibleChanged()
void	pressed(int index, QBarSet *barset)
void	released(int index, QBarSet *barset)

.....

额外继承的 1 个公共槽继承自 QObject 11 个静态成员函数继承自 QObject 9 个保护函数继承自 QObject

.....

## 详细说明

**QAbstractBarSeries** 类是所有条形柱的抽象类。

在条形图中，条形柱被定义为包含一种数据的集合。条形柱的位置由其类别与数值来决定。条形柱组合则是属于同一类别的条形柱。条形柱的显示则是由创建图表的时候决定的。

如果使用 **QValueAxis** 来代替 **QBarCategoryAxis** 当做图表的主轴。那么条形柱别按照索引值来分类。

可以参考 **Qt Example** (**example** 这里我还没有来得及翻译)

.....

## 成员类型

**enum QAbstractBarSeries::LabelsPosition\*\***

这个枚举值表示的是条形柱标签的位置：

枚举值	数值	描述
<b>QAbstractBarSeries::LabelsCenter</b>	<b>0</b>	中部
<b>QAbstractBarSeries::LabelsInsideEnd</b>	<b>1</b>	顶部
<b>QAbstractBarSeries::LabelsInsideBase</b>	<b>2</b>	底部
<b>QAbstractBarSeries::LabelsOutsideEnd</b>	<b>3</b>	外部



## 第九章 QAbstractItemModel

QAbstractItemModel 类为项模型类提供了抽象接口。 [更多...](#)

属性	方法
头文件	<code>#include &lt;QAbstractItemModel&gt;</code>
qmake	<code>QT+=core</code>
自从	Qt 4.6
继承	QObject
派生	QAbstractListModel、QAbstractProxyModel、QAbstractTableModel、QConcatenateTablesProxyModel、QDirModel、QFileSystemModel 和 QStandardItemModel

公有成员类型

类型	类型名称
enum class	CheckIndexOption NoOption, IndexIsValid, DoNotUseParent, ParentIsInvalid
flags	CheckIndexOptions
enum	LayoutChangeHint NoLayoutChangeHint, VerticalSortHint, HorizontalSortHint

公共成员函数

类型	函数名称
	QAbstractItemModel(QObject *parent = nullptr)
virtual	~QAbstractItemModel()
virtual QModelIndex	buddy(const QModelIndex &index) const
virtual bool	canDropMimeData(const QMimeData *data, Qt::DropAction action, int row, int column, const QModelIndex &parent) const
virtual bool	canFetchMore(const QModelIndex &parent) const

bool	checkIndex(const QModelIndex &index, QAbstractItemModel::CheckIndexOptions options = CheckIndexOption::NoOption) const
virtual int	columnCount(const QModelIndex &parent = QModelIndex()) const = 0
virtual QVariant	data(const QModelIndex &index, int role = Qt::DisplayRole) const = 0
virtual bool	dropMimeData(const QMimeData *data, Qt::DropAction action, int row, int column, const QModelIndex &parent)
virtual void	fetchMore(const QModelIndex &parent)
virtual Qt::ItemFlags	flags(const QModelIndex &index) const
virtual bool	hasChildren QModelIndex &parent = QModelIndex()) const
bool	hasIndex row, int column, const QModelIndex &parent = QModelIndex()) const
virtual QVariant	headerData section, Qt::Orientation orientation, int role = Qt::DisplayRole) const
virtual QModelIndex	index row, int column, const QModelIndex &parent = QModelIndex()) const = 0
bool	insertColumn column, const QModelIndex &parent = QModelIndex())
virtual bool	insertColumns column, int count, const QModelIndex &parent = QModelIndex())
bool	insertRow(int row, const QModelIndex &parent = QModelIndex())
virtual bool	insertRows row, int count, const QModelIndex &parent = QModelIndex())
virtual QMap<int, QVariant>	itemData QModelIndex &index) const
virtual QModelIndexList	match QModelIndex &start, int role, const QVariant &value, int hits = 1, Qt::MatchFlags flags = Qt::MatchFlags (Qt::MatchStartsWith
virtual QMimeData *	mimeData QModelIndexList &indexes) const
virtual QStringList	mimeTypes const
bool	moveColumn QModelIndex &sourceParent, int sourceColumn, const QModelIndex &destinationParent, int destinationChild)



virtual bool	moveColumns QModelIndex &sourceParent, int sourceColumn, int count, const QModelIndex &destinationParent, int destinationChild)
bool	moveRow QModelIndex &sourceParent, int sourceRow, const QModelIndex &destinationParent, int destinationChild)
virtual bool	moveRows QModelIndex &sourceParent, int sourceRow, int destinationChild)
bool	moveRow QModelIndex &sourceParent, int sourceRow, const QModelIndex &destinationParent, int destinationChild)
virtual bool	moveRows QModelIndex &sourceParent, int sourceRow, int count, const QModelIndex &destinationParent, int destinationChild)
virtual QModelIndex	parent QModelIndex &index) const = 0
bool	removeColumn column, const QModelIndex &parent = QModelIndex())
virtual bool	removeColumns column, int count, const QModelIndex &parent = QModelIndex())
bool	removeRow row, const QModelIndex &parent = QModelIndex())
virtual bool	removeRows row, int count, const QModelIndex &parent = QModelIndex())
virtual QHash<int, QByteArray>	roleNames const
virtual int	rowCount QModelIndex &parent = QModelIndex()) const = 0
virtual bool	setData QModelIndex &index, const QVariant &value, int role = Qt::EditRole)
virtual bool	setHeaderData section, Qt::Orientation orientation, const QVariant &value, int role = Qt::EditRole)
virtual bool	setItemData QModelIndex &index, const QMap<int, QVariant> &roles)
virtual QModelIndex	sibling row, int column, const QModelIndex &index) const
virtual void	sort column, Qt::SortOrder order = Qt::AscendingOrder)
virtual QSize	span QModelIndex &index) const

virtual Qt::DropActions	supportedDragActions const
virtual Qt::DropActions	supportedDropActions const

.....

#### 公共槽函数

类型	函数名称
virtual void	revert()
virtual bool	submit()

.....

#### 信号

类型	函数名称
void	columnsAboutToBeInserted(QModelIndex &parent, int first, int last)
void	columnsAboutToBeMoved(const QModelIndex &sourceParent, int sourceStart, int sourceEnd, const QModelIndex &destinationParent, int destinationColumn)
void	columnsAboutToBeRemoved(const QModelIndex &parent, int first, int last)
void	columnsInserted(QModelIndex &parent, int first, int last)
void	columnsMoved(const QModelIndex &parent, int start, int end, const QModelIndex &destination, int column)
void	columnsRemoved(const QModelIndex &parent, int first, int last)
void	dataChanged(const QModelIndex &topLeft, const QModelIndex &bottomRight, const QVector &roles = QVector())
void	headerDataChanged(Qt::Orientation orientation, int first, int last)
void	layoutAboutToBeChanged(const QList &parents = QList(), QAbstractItemModel::LayoutChangeHint hint = QAbstractItemModel::NoLayoutChangeHint)
void	layoutChanged(const QList &parents = QList(), QAbstractItemModel::LayoutChangeHint hint = QAbstractItemModel::NoLayoutChangeHint)
void	modelAboutToBeReset()
void	modelReset()
void	rowsAboutToBeInserted(const QModelIndex &parent, int start, int end)
void	rowsAboutToBeMoved(const QModelIndex &sourceParent, int sourceStart, int sourceEnd, const QModelIndex &destinationParent, int destinationRow)
void	rowsAboutToBeRemoved(const QModelIndex &parent, int first, int last)
void	rowsInserted(const QModelIndex &parent, int first, int last)
void	rowsMoved(const QModelIndex &parent, int start, int end, const QModelIndex &destination, int row)
void	rowsRemoved(const QModelIndex &parent, int first, int last)

.....

类型	函数名称
void	beginInsertColumns(const QModelIndex &parent, int first, int last)
void	beginInsertRows(const QModelIndex &parent, int first, int last)
bool	beginMoveColumns(const QModelIndex &sourceParent, int sourceFirst, int sourceLast, const QModelIndex &destinationParent, int destinationChild)
bool	beginMoveRows(const QModelIndex &sourceParent, int sourceFirst, int sourceLast, const QModelIndex &destinationParent, int destinationChild)
void	beginRemoveColumns(const QModelIndex &parent, int first, int last)
void	beginRemoveRows(const QModelIndex &parent, int first, int last)
void	beginResetModel()
void	changePersistentIndex(const QModelIndex &from, const QModelIndex &to)
void	changePersistentIndexList(const QModelIndexList &from, const QModelIndexList &to)
QModelIndex	createIndex(int row, int column, void *ptr = nullptr) const
QModelIndex	createIndex(int row, int column, quintptr id) const
void	endInsertColumns()
void	endInsertRows()
void	endMoveColumns()
void	endMoveRows()
void	endRemoveColumns()
void	endRemoveRows()
void	endResetModel()
QModelIndexList	persistentIndexList() const

受保护的槽函数

.....

类型	函数名称
void	resetInternalData()

详细描述

.....

**QAbstractItemModel** 类定义了项模型与模型/视图体系结构中的其他组件进行交互操作时必须

使用的标准接口。应该子类化该类创建新的模型，而不是直接实例化使用。

**QAbstractItemModel** 类是模型/视图类中的一个，也是 Qt 模型/视图框架的一部分。它可以用作 QML 中的项视图元素或 Qt Widgets 模块中的项视图类的底层数据模型。

如果需要一个模型来使用项视图，比如 QML 的 List View 元素或者 C++ widgets 的 **QListView** 或者 **QTableView**，应该考虑子类化 **QAbstractListModel** 或者 **QAbstractTableModel** 而不是使用该类。

底层数据模型作为表的层次结构暴露给视图和委托。如果不使用层次结构，那么模型就是一个简单的具有行和列的表。每个项都有一个由 **QModelIndex** 指定的惟一索引。

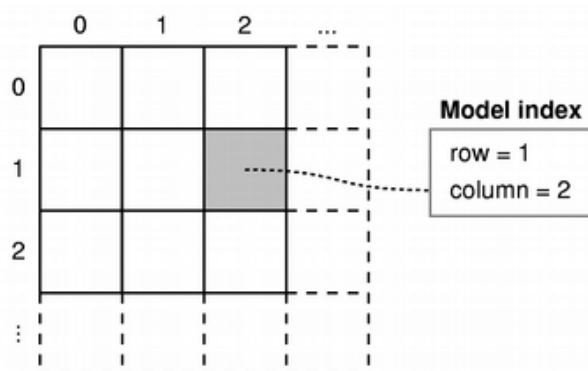


图 9.1: model index

每个数据项都可以通过包含一个关联的模型索引的模型进行访问。该索引可以通过 **index()** 函数获得。每个索引可能有一个 **sibling()** 索引；子项有一个 **parent()** 索引。

每个项目都有许多与之关联的数据元素，可以通过为模型的 **data()** 函数指定一个角色（请参阅 **Qt::ItemDataRole**）来检索它们。可以使用 **itemData()** 函数同时获取所有可用角色的数据。

使用特定的 **Qt::ItemDataRole** 设置每个角色的数据。可以使用 **setData()** 单独设置各个角色的数据，也可以使用 **setItemData** 设置所有角色的数据。

可以使用 **flags()** 查询项（请参阅 **Qt::ItemFlag**），以查看是否可以通过其他方式选择，拖动或操纵它们。

如果项具有子对象，则 **hasChildren** 为相应的索引返回 **true**。

该模型在层次结构的每个级别都有一个 **rowCount()** 和 **columnCount**。可以使用 **insertRows()**，**insertColumns()**，**removeRows()** 和 **removeColumns()** 插入和删除行和列。

模型发出信号以指示变化。例如，只要模型可用的数据项发生更改，就会发出 **dataChanged()**。对模型提供的标题的更改将发射 **headerDataChanged()** 信号。如果底层数据的结构发生了变化，则模型可以发出 **layoutChanged()** 来向任何附加的视图指示它们应该重新显示所显示的任何项，并需要考虑到新的结构。

可以使用 **match()** 函数在模型中搜索可用的项以查找特定数据。

要对模型进行排序，可以使用 **sort()**。

.....

## 子类化

注意： 在模型子类化参考中有一些关于模型子类化的通用指南。

子类化 `QAbstractItemModel` 时，至少必须实现 `index()`、`parent()`、`rowCount()`、`columnCount()` 和 `data()`。这些函数在所有的只读模型中使用，同样也构成了可编辑模型的基础。

还可以为模型重新实现 `hasChildren()` 来提供特殊的行为，而不是重新实现成本很高的 `rowCount()`。这使得模型可以限制视图请求的数据量，并且可以作为实现模型数据的惰性填充的一种方式。

要在模型中启用编辑，还必须实现 `setData` 和重新实现 `flags()` 以确保返回 `ItemIsEditable`。还可以重新实现 `headerData()` 和 `setHeaderData` 来控制呈现模型标题的方式。

当分别重新实现 `setData()` 和 `setHeaderData()` 函数时，必须显式发射 `dataChanged()` 和 `headerDataChanged()` 信号。

定制模型需要创建模型索引以供其他组件使用。为此，请使用适当的行号和列号以及相应的标识符调用 `createIndex()`，并将其作为指针或整数值。这些值的组合对于每个项都必须是唯一的。定制模型通常在其他重新实现的函数中使用这些唯一标识符，以检索项数据并访问有关该项目的父项和子项的信息。有关唯一标识符的更多信息，请参见简单树模型示例。

不必支持 `Qt::ItemDataRole` 中定义的角色。根据模型中包含的数据类型，可能只有实现 `data()` 函数以返回一些更常见角色的有效信息才有用。大多数模型至少为 `Qt::DisplayRole` 提供项数据的文本表示，行为良好的模型也应为 `Qt::ToolTipRole` 和 `Qt::WhatsThisRole` 提供有效信息。支持这些角色可使模型与标准 Qt 视图一起使用。但是，对于某些处理高度专业化数据的模型，仅为用户定义的角色提供数据可能是合适的。

提供可调整数据结构大小的接口的模型可以提供 `insertRows()`、`removeRows()`、`insertColumns()` 和 `removeColumns()` 的实现。在实现这些函数时，重要的是要在模型尺寸大小发生之前和之后将有关模型尺寸的更改通知所有连接的视图：

- `insertRows()` 的实现必须在将新行插入数据结构之前调用 `beginInsertRows()`，然后立即调用 `endInsertRows()`。
- `insertColumns()` 的实现必须在将新列插入数据结构之前调用 `beginInsertColumns()`，然后立即调用 `endInsertColumns()`。
- `removeRows()` 的实现必须在从数据结构中删除行之前调用 `beginRemoveRows()`，然后立即调用 `endRemoveRows()`。
- `removeColumns()` 的实现必须在列从数据结构中删除之前调用 `beginRemoveColumns()`，然后立即调用 `endRemoveColumns()`。

这些函数发出的私有信号使附加组件有机会在任何数据变得不可用之前采取行动。使用这些 `begin` 和 `end` 函数封装插入和删除操作还使模型能够正确地管理持久模型索引。如果希望正确处理选择，则必须确保调用了这些函数。如果插入或删除带有子项的项，则不需要为子项调用这些函数。换句话说，父项将管理其子项。

要创建增量填充的模型，可以重新实现 `fetchMore()` 和 `canFetchMore()`。如果 `fetchMore()`

的重新实现向模型中添加了行，则必须调用 `beginInsertRows()` 和 `endInsertRows()`。

参见模型类、模型子类化参考、`QModelIndex`、`QAbstractItemView`、在项视图中使用拖放、简单 DOM 模型示例、简单树模型示例、可编辑树模型示例和 `Fetch More` 示例。

## 成员类型文档

`CheckIndexOption` `CheckIndexOptions` enum class `QAbstractItemModel::CheckIndexOption`  
`flags` `QAbstractItemModel::CheckIndexOptions`

这个枚举可以用来控制 `QAbstractItemModel::checkIndex()` 执行的检查。

常量	值	描述
<code>QAbstractItemModel::CheckIndexOption::NoOptions</code>	0x0000	没有指定检查选项。
<code>QAbstractItemModel::CheckIndexOption::IndexIsValid</code>	0x0001	传递给 <code>QAbstractItemModel::checkIndex()</code> 的模型索引被检查为有效的模型索引。
<code>QAbstractItemModel::CheckIndexOption::DoNotUseParent</code>	0x0002	不执行任何涉及到传递给 <code>QAbstractItemModel::checkIndex()</code> 的父索引的使用的检查。

该枚举是在 Qt 5.11 中引入或修改的。

`CheckIndexOptions` 类型是一个 `QFlags<CheckIndexOption>` 的类型定义。它存储一个或组合的 `CheckIndexOption` 值。

## LayoutChangeHint

enum `QAbstractItemModel::LayoutChangeHint`

这个枚举描述了模型改变布局的方式。

常量	值	描述
<code>QAbstractItemModel::NoLayoutChangeHint</code>	0	没有任何提示。
<code>QAbstractItemModel::VerticalSortHint</code>	1	正在对行进行排序。
<code>QAbstractItemModel::HorizontalSortHint</code>	2	正在对列进行排序。

注意，`VerticalSortHint` 和 `HorizontalSortHint` 表示项在同一父级中移动，而不是移动到模型中的不同父级，也没有过滤掉或过滤进来。

## 成员函数文档

`QAbstractItemModel`

`QAbstractItemModel::QAbstractItemModel(QObject *parent = nullptr)`

构造指定父类对象 `parent` 的抽象项模型。

#### `columnsAboutToBeInserted`

```
[signal] void QAbstractItemModel::columnsAboutToBeInserted(const QModelIndex &parent, int first, int last)
```

在将列插入模型之前就发射此信号。新项将位于给定父项 `parent` 下的首 `first` 尾 `last` 之间。

注意：连接到这个信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 的实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。它可以用于信号连接，但不能由用户发射。

参见 `insertColumns()` 和 `beginInsertColumns()`。

#### `columnsAboutToBeMoved`

```
[ signal ] void QAbstractItemModel::columnsAboutToBeMoved(const QModelIndex &sourceParent, int sourceStart, int sourceEnd, const QModelIndex &destinationParent, int destinationColumn)
```

模型中的列被移动之前发射该信号。将要移动的项是在给定 `sourceParent` 下在 `sourceStart` 和 `sourceEnd` 之间（包括首尾）的项。它们将从 `destinationColumn` 列开始移动到 `destinationParent`。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

该函数在 Qt4.6 中被引入。

参见 `beginMoveRows()`。

#### `columnsAboutToBeRemoved`

```
[ signal ] void QAbstractItemModel::columnsAboutToBeRemoved(const QModelIndex &parent, int first, int last)
```

模型中的列被移除之前发射该信号。将要移除的项是在给定 `parent` 下在 `first` 和 `last` 之间（包括首尾）的项。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

参见 `removeColumns()` 和 `beginRemoveColumns()`。

#### `columnsInserted`

```
[ signal ] void QAbstractItemModel::columnsInserted(const QModelIndex &parent, int first, int last)
```



将列插入到模型之后发射该信号。新的项是在给定 `parent` 下在 `first` 和 `last` 之间（包括首尾）的项。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

参见 `insertColumns()` 和 `beginInsertColumns()`。

#### `columnsMoved`

```
[ signal ] void QAbstractItemModel::columnsMoved(const QModelIndex &parent, int start, int end, const QModelIndex &destination, int column)
```

模型中的列被移动之后发射该信号。新的项是在给定 `parent` 下在 `start` 和 `end` 之间（包括首尾）的项。它们将从 `column` 列开始移动到 `destination`。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

该函数在 Qt4.6 中被引入。

参见 `beginMoveRows()`。

#### `columnsRemoved`

```
[ signal ] void QAbstractItemModel::columnsRemoved(const QModelIndex &parent, int first, int last)
```

模型中的列被移除之后发射该信号。移除的项是在给定 `parent` 下在 `first` 和 `last` 之间（包括首尾）的项。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

参见 `removeColumns()` 和 `beginRemoveColumns()`。

#### `dataChanged`

```
[ signal ] void QAbstractItemModel::dataChanged(const QModelIndex &topLeft, const QModelIndex &bottomRight, const QVector<int> &roles = QVector())
```

现有的项的数据发生改变时发射该信号。

如果项是同一父项，则受影响的项是在 `topLeft` 和 `bottomRight`（包含）之间的项目。如果项没有相同的父项，则行为是不确定的。

当重新实现 `setData()` 函数时，必须显示地发射该信号。

可选的 **roles** 参数可用于指定实际修改了哪些数据角色。**Roles** 参数中的向量为空，表示应将所有角色视为已修改。角色参数中元素的顺序没有任何关联。

参见 `headerDataChanged()`、`setData()` 和 `layoutChanged()`。

#### `headerDataChanged`

```
[ signal ] void QAbstractItemModel::headerDataChanged(const Qt::Orientation
&orientation, int first, int last)
```

当标题改变时发射该信号。**orientation** 表示是横向标题还是竖向标题发生了改变。标题中从 **first** 到 **last** 的部分需要更新。

当重新实现 `setData()` 函数时，必须显示地发射该信号。

如果要更改列数或行数，则不需要发出此信号，而可以使用 `begin/end` 函数（有关详细信息，请参见 `QAbstractItemModel` 类描述中的子类化部分）。

参见 `headerData()`、`setHeaderData()` 和 `dataChanged()`。

#### `layoutAboutToBeChanged`

```
[ signal ] void QAbstractItemModel::layoutAboutToBeChanged(const QList<QPersistentMo
&parents = QList(), QAbstractItemModel::LayoutChangeHint hint = QAbstrac-
tItemModel::NoLayoutChangeHint)
```

这个信号会在模型的布局改变之前发出。连接到这个信号的组件使用它来适应模型布局的变化。

在发出 `layoutAboutToBeChanged()` 之后，子类应该更新所有的持久化模型索引。

可选的 **parents** 参数用于提供更具体的通知关于模型布局的哪些部分正在被改变。空列表表示对整个模型的布局进行了更改。**parents** 列表中元素的顺序不重要。可选的 **hint** 参数用于提示模型重新布局时都发生了什么。

该函数在 Qt 5.0 中被引入。

参见 `layoutChanged()` 和 `changePersistentIndex()`。

#### `layoutChanged`

```
[ signal ] void QAbstractItemModel::layoutChanged(const QList<QPersistentModelIndex>
&parents = QList(), QAbstractItemModel::LayoutChangeHint hint = QAbstrac-
tItemModel::NoLayoutChangeHint)
```

每当模型公开的项的布局发生变化时，就会发出这个信号，例如，对模型进行排序时。当视图接收到该信号时，应更新项的布局来反映此更改。

当对 `QAbstractItemModel` 或 `QAbstractProxyModel` 进行子类化时，请确保在更改项顺序或更改要公开给视图的数据的结构之前发出 `layoutAboutToBeChanged()` 信号，并在更改布局后发出 `layoutChanged()` 信号。

可选的 **parents** 参数用于给出有关模型布局的哪些部分正在更改的具体的通知。空列表表示更改了整个模型的布局。**parents** 列表中元素的顺序并不重要。可选的 **hint** 参数用于提示模型重新布局时发生的情况。

子类应在发出 `layoutChanged()` 信号之前更新所有持久模型索引。换句话说，当结构改变时：

- 发出 `layoutAboutToBeChanged`
- 记住将会改变的 `QModelIndex`
- 更新内部数据
- 调用 `changePersistentIndex()`
- 发出 `layoutChanged`

该函数在 Qt 5.0 中被引入。

参见 `layoutAboutToBeChanged()`、`dataChanged()`、`headerDataChanged()`、`modelReset()` 和 `changePersistentIndex()`。

`modelAboutToBeReset [ signal ] void QAbstractItemModel::modelAboutToBeReset()`

当调用 `beginResetModel()` 时，在模型的内部状态（例如持久模型索引）失效之前发出这个信号。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

该函数在 Qt 4.2 中被引入。

参见 `beginResetModel()` 和 `modelReset()`。

`modelReset`

`[ signal ] void QAbstractItemModel::modelReset()`

当调用 `endResetModel()` 时，在模型的内部状态（例如持久模型索引）失效之后发出这个信号。

注意，如果模型被重置，则应该认为之前从模型中检索的所有信息都是无效的。这包括但不限于 `rowCount()`、`columnCount()`、`flags()`、通过 `data()` 检索的数据和 `roleNames()`。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

该函数在 Qt 4.1 中被引入。

参见 `endResetModel()` 和 `modelAboutToBeReset()`。

`resetInternalData`

`[ protected slot ] void QAbstractItemModel::resetInternalData()`

该槽函数在模型的内部数据被清除并被重置之后被调用。

该槽函数为具体代理模型子类提供了便利，例如维护了额外的数据的

`QSortFilterProxyModel` 的子类。

```
class CustomDataProxy : public QSortFilterProxyModel
{
    Q_OBJECT
public:
```

```
CustomDataProxy(QObject *parent)
    : QSortFilterProxyModel(parent)
{
}

...

QVariant data(const QModelIndex &index, int role) override
{
    if (role != Qt::BackgroundRole)
        return QSortFilterProxyModel::data(index, role);

    if (m_customData.contains(index.row()))
        return m_customData.value(index.row());
    return QSortFilterProxyModel::data(index, role);
}
private slots:
    void resetInternalData()
    {
        m_customData.clear();
    }
private:
    QHash<int, QVariant> m_customData;
};
```

注意：由于错误，该槽函数没有出现在 Qt 5.0 中。

该函数在 Qt 4.8 中被引入。

参见 `modelAboutToBeReset()` 和 `modelReset()`。

## revert

[ virtual slot ] void QAbstractItemModel::revert()

让模型知道它应该丢弃缓存的信息。这个函数通常用于行编辑。

该函数在 Qt 4.2 中被引入。

参见 `submit()`。

## rowsAboutToBeInserted

[ signal ] void QAbstractItemModel::rowsAboutToBeInserted(const QModelIndex &parent, int start, int end)

在将行插入模型之前就发出该信号。新项将位于给定 `parent` 项目下的包含 `start` 和 `end` 之间。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发出，而不能在子类代码中显式发出。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

参见 `insertRows()` 和 `beginInsertRows()`。

#### `rowsAboutToBeMoved`

```
[ signal ] void QAbstractItemModel::rowsAboutToBeMoved(const QModelIndex
&sourceParent, int sourceStart, int sourceEnd, const QModelIndex &desti-
nationParent, int destinationRow)
```

模型中的行被移动之前发射该信号。将要移动的项是在给定 `sourceParent` 下在 `sourceStart` 和 `sourceEnd` 之间（包括首尾）的项。它们将从 `destinationRow` 列开始移动到 `destinationParent`。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

该函数在 Qt4.6 中被引入。

参见 `beginMoveRows()`。

#### `rowsAboutToBeRemoved`

```
[ signal ] void QAbstractItemModel::rowsAboutToBeRemoved(const QModelIndex
&parent, int first, int last)
```

模型中的行被移除之前发射该信号。将要移除的项是在给定 `parent` 下在 `first` 和 `last` 之间（包括首尾）的项。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

参见 `removeRows()` 和 `beginRemoveRows()`。

#### `rowsInserted`

```
[ signal ] void QAbstractItemModel::rowsInserted(const QModelIndex &par-
ent, int first, int last)
```

将行插入到模型之后发射该信号。新的项是在给定 `parent` 下在 `first` 和 `last` 之间（包括首尾）的项。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

参见 `insertRows()` 和 `beginInsertRows()`。

#### `rowsMoved`

```
[ signal ] void QAbstractItemModel::rowsMoved(const QModelIndex &parent,
int start, int end, const QModelIndex &destination, int column)
```

模型中的行被移动之后发射该信号。新的项是在给定 `parent` 下在 `start` 和 `end` 之间（包括首尾）的项。它们将从 `column` 列开始移动到 `destination`。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

该函数在 Qt4.6 中被引入。

参见 `beginMoveRows()`。

#### `rowsRemoved`

```
[ signal ] void QAbstractItemModel::rowsRemoved(const QModelIndex &parent, int first, int last)
```

模型中的行被移除之后发射该信号。移除的项是在给定 `parent` 下在 `first` 和 `last` 之间（包括首尾）的项。

注意：连接到该信号的组件使用它来适应模型尺寸的变化。它只能由 `QAbstractItemModel` 实现发射，不能在子类代码中显式发射。

注意：这是一个私有信号。仅用于信号连接，而不能由用户发射。

参见 `removeRows()` 和 `beginRemoveRows()`。

#### `submit`

```
[ virtual slot ] void QAbstractItemModel::submit()
```

让模型知道它应该将缓存的信息提交到永久存储。这个函数通常用于行编辑。

如果没有错误，返回 `true`；否则返回 `false`。

参见 `revert()`。

#### `~QAbstractItemModel`

```
[ virtual ] void QAbstractItemModel::QAbstractItemModel()
```

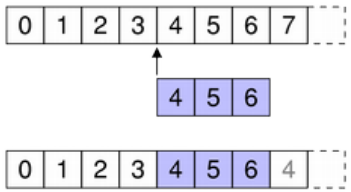
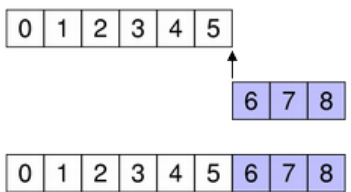
销毁抽象项模型。

#### `beginInsertColumns`

```
[ protected ] void QAbstractItemModel::beginInsertColumns(const QModelIndex &parent, int first, int last)
```

开始一个列插入操作。

在子类中重新实现 `insertColumns()` 时，必须在将数据插入模型的底层数据存储之前调用此函数。`parent` 索引对应于插入新列的父索引；`first` 和 `last` 是新列插入后的列号。

	<p>指定要插入到模型项中的列的范围的第一个和最后一个列号。例如，如图所示，我们在列 4 之前插入三列，所以 <b>first</b> 是 4, <b>last</b> 是 6: <code>beginInsertColumns(parent, 4, 6);</code> 这将插入三个新列作为列 4、5 和 6。</p>
	<p>要追加列，请将它们插入到最后一列之后。例如，如图所示我们，、将三列附加到一个包含六列的集合（以列 5 结尾），因此 <b>first</b> 是 6 and <b>last</b> 是 8: <code>beginInsertColumns(parent, 6, 8);</code> 这将追加两个新列作为列 6、7 和 8。</p>

注意：此函数发出 `columnAboutToBeInserted()` 信号，在插入数据之前，已连接的视图（或代理）必须处理该信号。否则，视图可能会以无效状态结束。

参见 `endInsertColumns()`。

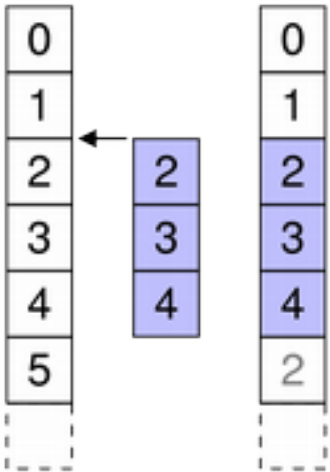
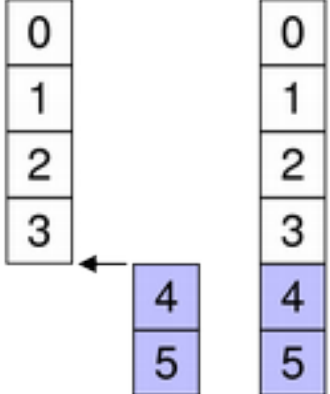
.....

## `beginInsertRows`

```
[ protected ] void QAbstractItemModel::beginInsertRows(const QModelIndex
&parent, int first, int last)
```

开始一个行插入操作。

在子类中重新实现 `insertRows()` 时，必须在将数据插入模型的底层数据存储之前调用此函数。**parent** 索引对应于插入新列的父索引；**first** 和 **last** 是新行插入后的行号。

	<p>为要插入模型中项的行范围指定第一行和最后一行。例如，如图所示，我们在第 2 行之前插入三行，因此 <b>first</b> 是 2，<b>last</b> 是 4：  <code>beginInsertRows(parent, 2, 4);</code> 这将插入三行新行，即第 2、3 和 4 行。</p>
	<p>要追加行，请将它们插入到最后一行之后。例如，如图所示，我们将两行附加到一个包含 4 个现有行的集合（以第 3 行结束），因此 <b>first</b> 是 4，<b>last</b> 是 5：  <code>beginInsertRows(parent, 4, 5);</code> 这将追加两个新行作为第 4 行和第 5 行。</p>

注意：此函数发出 `rowsAboutToBeInserted()` 信号，在插入数据之前，已连接的视图（或代理）必须处理该信号。否则，视图可能会以无效状态结束。

参见 `endInsertRows()`。

.....

## beginMoveColumns

```
[ protected ] void QAbstractItemModel::beginMoveColumns(const QModelIndex &sourceParent, int sourceFirst, int sourceLast, const QModelIndex &destinationParent, int destinationChild)
```

开始一个列移动操作。

当重新实现子类时，此方法简化了模型中实体的移动。此方法负责在模型中移动持久索引，否则您将需要自己执行此操作。使用 `beginMoveColumns` 和 `endMoveColumns()` 是直接发送与 `changePersistentIndex()` 一起的 `layoutAboutToBeChanged()` 和 `layoutChanged()` 的另一种选择。

**sourceParent** 索引对应于从其移出列的父级；**sourceFirst** 和 **sourceLast** 是要移动的列的第一列和最后一列。**destinationParent** 索引对应于将这些列移入的父级。**destinationChild** 是要将列移动到的列。也就是说，**sourceParent** 中 **sourceFirst** 列的索引将成为 **destinationParent** 中的 **destinationChild** 列，然后是所有其他列，直到 **sourceLast**。

但是，当在同一父目录下移动列时（**sourceParent** 和 **destinationParent** 是相等的），这



些列将被放置在 `destinationChild` 索引之前。也就是说，如果您希望移动列 0 和 1，使它们变成列 1 和列 2, `destinationChild` 应该是 3。在本例中，源列 `i`（位于 `sourceFirst` 和 `sourceLast` 之间）的新索引等于  $(\text{destinationChild} - \text{sourceLast} - 1 + i)$ 。

注意，如果 `sourceParent` 和 `destinationParent` 是相同的，您必须确保 `destinationChild` 不在 `sourceFirst` 和 `sourceLast + 1` 的范围内。还必须确保不会尝试将列移动到它自己的子列或祖先列中。如果任一条件为真，此方法将返回 `false`，在这种情况下，应中止移动操作。

该函数在 Qt4.6 中被引入。

参见 `endMoveColumns()`。

.....

## `beginMoveRows`

```
[ protected ] void QAbstractItemModel::beginMoveRows(const QModelIndex
&sourceParent, int sourceFirst, int sourceLast, const QModelIndex &des-
tinationParent, int destinationChild)
```

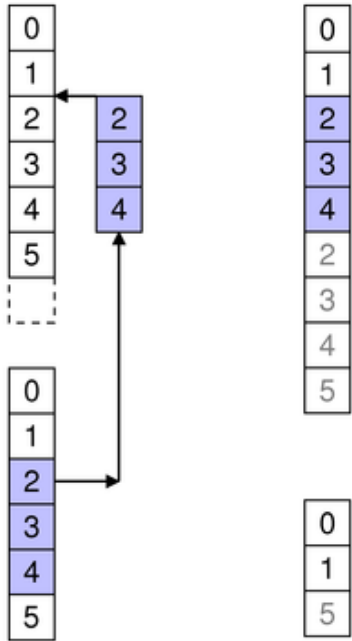
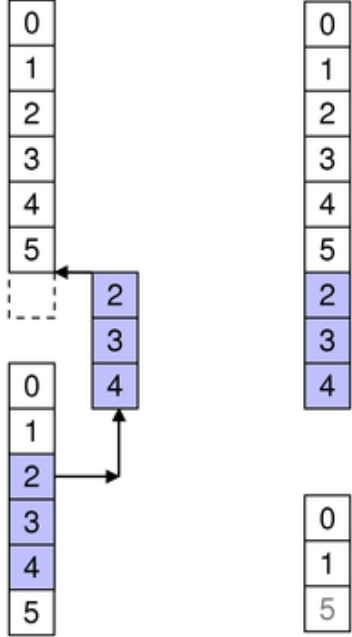
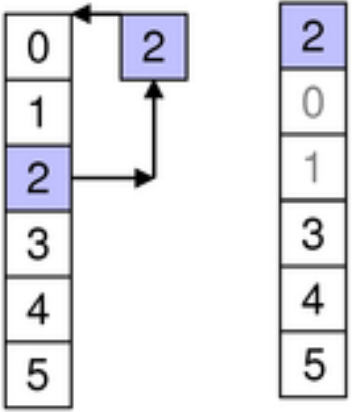
开始一个行移动操作。

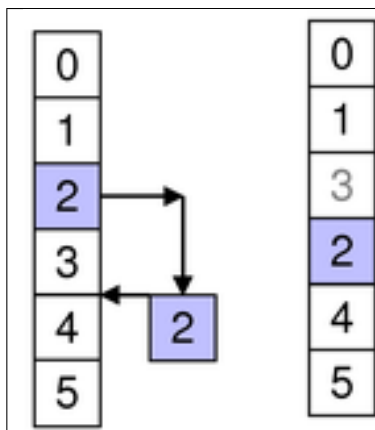
当重新实现子类时，此方法简化了模型中实体的移动。此方法负责在模型中移动持久索引，否则您将需要自己执行此操作。使用 `beginMoveRows` 和 `endMoveRows` 是直接发送与 `changePersistentIndex()` 一起的 `layoutAboutToBeChanged()` 和 `layoutChanged()` 的另一种选择。

`sourceParent` 索引对应于从其移出行的父级；`sourceFirst` 和 `sourceLast` 是要移动的第一行和最后一行。`destinationParent` 索引对应于将这些行移入的父级。`destinationChild` 是要将行移动到的行。也就是说，`sourceParent` 中的 `sourceFirst` 行的索引将成为 `destinationParent` 中的 `destinationChild` 行，然后是所有其他行，直到 `sourceLast`。

但是，当在同一父目录下移动列时（`sourceParent` 和 `destinationParent` 是相等的），这些行将被放置在 `destinationChild` 索引之前。也就是说，如果您希望移动列 0 和 1，使它们变成行 1 和行 2, `destinationChild` 应该是 3。在本例中，源行 `i`（位于 `sourceFirst` 和 `sourceLast` 之间）的新索引等于  $(\text{destinationChild} - \text{sourceLast} - 1 + i)$ 。

注意，如果 `sourceParent` 和 `destinationParent` 是相同的，您必须确保 `destinationChild` 不在 `sourceFirst` 和 `sourceLast + 1` 的范围内。还必须确保不会尝试将行移动到它自己的子列或祖先行中。如果任一条件为真，此方法将返回 `false`，在这种情况下，应中止移动操作。

	<p>指定源父行中您希望在模型中移动的行跨度的第一行和最后一行编号。还要在目标父级中指定要将范围内的行移动到的行。例如，如图所示，我们将源中的第 2 行到第 3 行移动了三行，因此 <code>sourceFirst</code> 为 2, <code>sourceLast</code> 为 4。我们将这些项移动到目标的第 2 行上方，因此 <code>destinationChild</code> 为 2。<code>beginMoveRows(sourceParent, 2, 4, destinationParent, 2);</code> 这会将源中的三行第 2、3 和 4 行移动到目标中的 2、3 和 4 行。其他受影响的同级项也因此被移位。</p>
	<p>若要将行追加到另一个父元素，请将它们移到最后一行的后面。例如，如图所示，我们将三行移动到一个包含 6 个现有行的集合中（以第 5 行结束），因此 <code>destinationChild</code> 为 6: <code>beginMoveRows(sourceParent, 2, 4, destinationParent, 6);</code> 这会将目标行移到目标父级的末尾，分别为 6、7 和 8。</p>
	<p>要在同一父级中移动行，请指定要将其移动到的行。例如，如图所示，我们将一项从第 2 行移至第 0 行，因此 <code>sourceFirst</code> 和 <code>sourceLast</code> 为 2, <code>destinationChild</code> 为 0。<code>beginMoveRows(parent, 2, 2, parent, 0);</code> 注意，其他行可能会相应移位。另请注意，在同一父级中移动项时，请勿尝试无效移动或无操作移动。在上面的示例中，项 2 位于移动之前的第 2 行，因此无法将其移动到第 2 行（已经存在）或第 3 行（空操作，因为第 3 行意味着已经在第 3 行之上）</p>



要在同一父级中移动行，请指定要将其移动到的行。例如，如图所示，我们将一项从第 2 行移至第 4 行，因此 `sourceFirst` 和 `sourceLast` 为 2，`destinationChild` 为 4。`beginMoveRows(parent, 2, 2, parent, 4);` 注意，其他行可能会相应移位。

该函数在 Qt4.6 中被引入。

参见 `endMoveRows()`。

### `beginRemoveColumns`

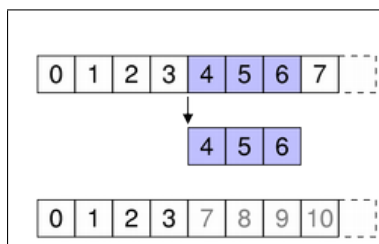
```
[ protected ] void QAbstractItemModel::beginRemoveColumns(const QModelIndex &parent, int first, int last)
```

开始一个列移除操作。

在子类中重新实现 `removeColumns()` 时，必须在从模型的底层数据存储中删除数据之前调用此函数。`parent` 索引对应于删除新列的父索引；`first` 和 `last` 是要删除的第一个和最后一个列的列号。

注意：此函数发出 `columnAboutToBeRemoved()` 信号，在删除数据之前，已连接的视图（或代理）必须处理该信号。否则，视图可能会以无效状态结束。

参见 `endRemoveColumns()`。



指定要从模型中的项中删除的列的范围的第一个和最后一个列号。例如，如图所示，我们将这三列从第 4 列移到第 6 列，因此 `first` 是 4，`last` 是 6：`beginRemoveColumns(parent, 4, 6);`

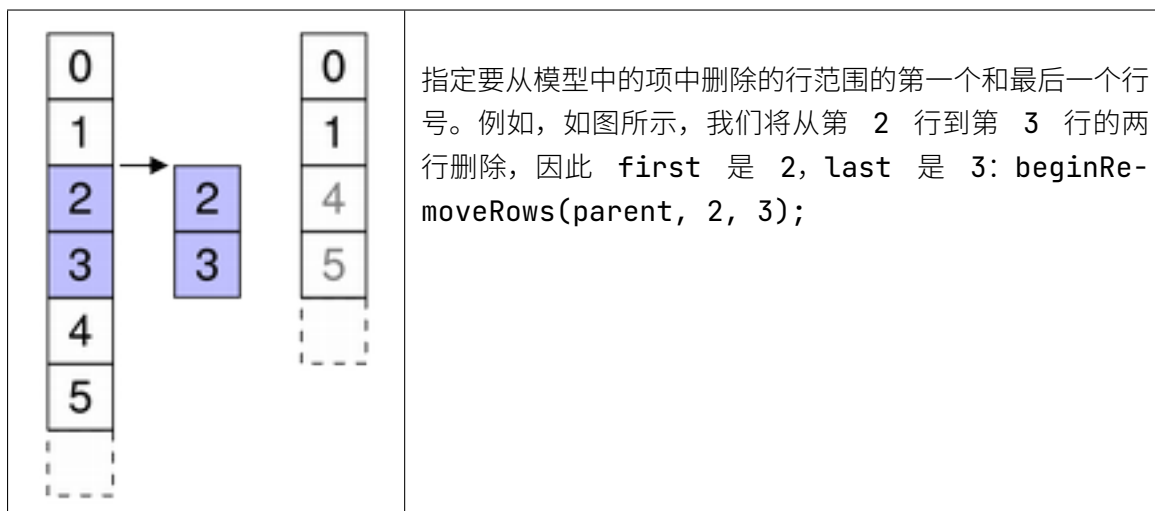
### `beginRemoveRows`

```
[ protected ] void QAbstractItemModel::beginRemoveRows(const QModelIndex &parent, int first, int last)
```

开始一个行移除操作。

在子类中重新实现 `removeRows()` 时，必须在从模型的基础数据存储中删除数据之前调用此函数。

`parent` 引对应于从中删除新行的父索引；`first` 和 `last` 是要删除的行的行号。



注意：此函数发出 `rowsAboutToBeRemoved()` 信号，连接的视图（或代理）必须在删除数据之前处理该信号。否则，视图可能会以无效状态结束。

参见 `endRemoveRows()`。

**beginResetModel**

```
[ protected ] void QAbstractItemModel::beginResetModel()
```

开始模型重置操作。

重置操作会在任何附加的视图将模型重置为当前状态。

注意：附加到这个模型的任何视图都将被重置。

当一个模型被重置时，这意味着以前从该模型报告的任何数据现在都是无效的，必须再次进行查询。这也意味着当前项和任何选定项都将无效。当模型从根本上更改其数据时，有时只需调用此函数比在底层数据源或其结构发生更改时发出 `dataChanged()` 通知其他组件更容易。

在重新设置模型或代理模型中的任何内部数据结构之前，必须调用此函数。

这个函数发出信号 `modelAboutToBeReset()`。

该函数在 Qt4.6 中被引入。

参见 `modelAboutToBeReset()`、`modelReset()` 和 `endResetModel()`。

**buddy**

```
[ virtual ] void QAbstractItemModel::buddy(const QModelIndex &index) const
```

返回由 `index` 表示的项的伙伴的模型索引。当用户想要编辑项目时，视图将调用此函数以检查是否应改为编辑模型中的另一个项。然后，视图将使用伙伴项返回的模型索引构造一个委托。

此功能的默认实现将每个项都作为自己的伙伴。

**canDropMimeData**

```
[ virtual ] bool QAbstractItemModel::canDropMimeData(const QMimeData *data,
Qt::DropAction action, int row, int column, const QModelIndex &parent)
const
```

如果模型接受放置 `data`，则返回 `true`。这个默认实现只检查 `mimeTypes()` 列表中数据是否至少有一种格式，以及操作是否在模型的 `supportedDropActions()` 中。

如果您想要测试是否可以在 `row`、`column`、`parent` 节点上放置 `data`，请在自定义模型中重新实现此函数。如果您不需要该测试，那么就没有必要重新实现此函数。

参见 `dropMimeData()` 和在项视图中使用拖放。

#### `canFetchMore`

```
[ virtual ] bool QAbstractItemModel::canFetchMore(const QModelIndex &parent) const
```

如果在 `parent` 索引下有更多可用的数据返回 `true`；否则返回 `false`；

默认的实现总是返回 `false`。

如果 `canFetchMore()` 返回 `true`，则应该调用 `fetchMore()` 函数。比如 `QAbstractItemView` 就是这样做的。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `fetchMore()`。

#### `changePersistentIndex`

```
[ protected ] void QAbstractItemModel::changePersistentIndex(const QModelIndex &from, const QModelIndex &to)
```

将等于给定的 `from` 的 `QPersistentModelIndex()` 模型索引更改为 `to` 模型索引。

如果没有找到与给定的模型索引 `from` 相等的持久模型索引，则什么也不会改变。

如果 `canFetchMore()` 返回 `true`，则应该调用 `fetchMore()` 函数。比如 `QAbstractItemView` 就是这样做的。

参见 `persistentIndexList()` 和 `changePersistentIndexList()`。

#### `changePersistentIndexList`

```
[ protected ] void QAbstractItemModel::changePersistentIndexList(const QModelIndexList &from, const QModelIndexList &to)
```

将等于给定的 `from` 的 `QPersistentModelIndex()`es 模型索引列表更改为 `to` 模型索引列表。

如果没有找到与给定的模型索引 `from` 相等的持久模型索引，则什么也不会改变。

该函数在 Qt4.1 中被引入。

参见 `persistentIndexList()` 和 `changePersistentIndex()`。

#### `checkIndex`

```
bool QAbstractItemModel::checkIndex(const QModelIndex &index, QAbstractItemModel::CheckIndexOptions options = CheckIndexOption::NoOption) const
```

此函数检查索引是否为此模型的合法模型索引。合法模型索引要么是无效的模型索引，要么是具有以下所有条件的有效模型索引：

- `index` 的模型就是 `this`；
- `index` 的行数大于等于零；
- `index` 的行数小于父索引的行数；
- `index` 的列数大于等于零；
- `index` 的列数小于父索引的列数。

`options` 参数可能会改变其中一些检查。如果 `options` 包含 `IndexIsValid`，那么 `index` 必须是一个有效的索引；这在重新实现 `data()` 或 `setData()` 等需要有效索引的函数时非常有用。

如果 `options` 包含 `DoNotUseParent`，那么将调用 `parent()` 的检查将被省略；允许在重新实现的 `parent()` 函数中调用此函数（否则，将导致无穷递归和崩溃）。

如果 `options` 不包含 `DoNotUseParent`，但包含 `IndexIsValid`，那么将执行额外的检查：检查父索引是否有效。这在实现平面模型（如列表或表）时非常有用，在这种情况下，模型索引不应该具有有效的父索引。

该函数如果检查成功返回 `true`，否则返回 `false`。允许在 `Q_ASSERT` 和类似的其他调试机制中使用该函数。如果某些检查失败，则将在 `qt.core.qabstractitemmodel.checkindex` 日志记录类别中显示一条警告消息，其中包含一些可能对调试失败有用的信息。

注意：这个函数是一个调试助手，用于实现您自己的项模型。在开发复杂模型时，以及在构建复杂的模型层次结构时（例如使用代理模型），调用这个函数来捕获传递给某个 `QAbstractItemModel()` API 的非法模型索引（如上定义）相关的 bug 是很有用的。

警告：请注意，将非法索引传递给项模型是未定义的行为，因此应用程序必须避免这样做，并且不能依赖于项模型可以用来优雅地处理非法索引的任何“防御性”编程。

该函数在 'Qt5.11' 中被引入。

参见 `QModelIndex`。

`columnCount`

```
[ pure virtual ] int QAbstractItemModel::columnCount(const QModelIndex &parent = QModelIndex()) const
```

返回给定 `parent` 索引的子项的列的数量。

在大多数子类中，列的数量独立于 `parent`。

例如：

```
int DomModel::columnCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return 3;
}
```

注意：在实现基于表的模型时，当父模型有效时，`columnCount()` 应该返回 0。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

该函数在 Qt4.1 中被引入。

参见 `rowCount()`。

#### `createIndex`

```
[ protected ] QModelIndex QAbstractItemModel::createIndex(int row, int
column, void *ptr = nullptr) const
```

使用内部指针 `ptr` 为给定的 `row` 和 `column` 创建模型索引。

当使用 `QSortFilterProxyModel` 时，它的索引有自己的内部指针。不建议在模型外部访问这个内部指针。使用 `data()` 函数代替。

这个函数提供了一个一致的接口，模型子类必须使用这个接口来创建模型索引。

参见 `QModelIndex::internalId()`。

#### `data`

```
[ pure virtual ] QVariant QAbstractItemModel::data(const QModelIndex &in-
dex, int role = Qt::DisplayRole) const
```

返回指定角色 `role` 和索引 `index` 的项数据。

注意：如果没有要返回的值，则返回无效的 `QVariant`，而不是返回 0。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `Qt::ItemDataRole`、`setData()` 和 `headerData()`。

#### `dropMimeData`

```
[ virtual ] bool QAbstractItemModel::dropMimeData(const QMimeData *data,
Qt::DropAction action, int row, int column, const QModelIndex &parent)
const
```

处理以给定操作 `action` 结束的拖放操作提供的数据 `data`。

如果数据和操作被模型处理了则返回 `true`，否则返回 `false`。

指定的行 `row`、列 `column` 和父索引 `parent` 指示模型中操作结束的项的位置。在正确的位置完成动作是模型的责任。

例如，`QTreeView` 中的一个项上的拖放操作可以导致新项作为行 `row`、列 `column` 和父索引 `parent` 指定的项的子项插入，或者作为项的兄弟项插入。

当行 `row` 和列 `column` 为 -1 时，这意味着放置的数据应该被认为是直接在 `parent` 上放置的。通常这意味着将数据附加为父项 `parent` 的子项。如果行 `row` 和列 `column` 大于或等于零，则表示放置发生在指定父索引 `parent` 的 `row` 和列 `column` 的前面。

调用 `mimeType()` 成员来获取可接受的 MIME 类型列表。这个默认实现假定 `mimeType()` 是默认实现，它返回一个默认 MIME 类型。如果您在自定义模型中重新实现 `mimeType()` 以返回多个 MIME 类型，那么您必须重新实现此函数以使用它们。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `supportedDropActions`、`canDropMimeData()` 和在项视图中使用拖放。

#### `endInsertColumns`

```
[ protected ] void QAbstractItemModel::endInsertColumns()
```

结束列插入操作。

在子类中重新实现 `insertColumns()` 时，必须在将数据插入模型的底层数据存储之后调用此函数。

参见 `beginInsertColumns()`。

#### `endInsertRows`

```
[ protected ] void QAbstractItemModel::endInsertRows()
```

结束行插入操作。

在子类中重新实现 `insertRows()` 时，必须在将数据插入模型的底层数据存储之后调用此函数。

参见 `beginInsertRows()`。

#### `endMoveColumns`

```
[ protected ] void QAbstractItemModel::endMoveColumns()
```

结束列移动操作。

在实现子类时，必须在模型的底层数据存储中移动数据之后调用此函数。

该函数在 Qt4.6 中被引入。

参见 `beginMoveColumns()`。

#### `endMoveRows`

```
[ protected ] void QAbstractItemModel::endMoveRows()
```

结束行移动操作。

在实现子类时，必须在模型的底层数据存储中移动数据之后调用此函数。

该函数在 Qt4.6 中被引入。

参见 `beginMoveRows()`。

#### `endRemoveColumns`

```
[ protected ] void QAbstractItemModel::endRemoveColumns()
```

结束列删除操作。



在子类中重新实现 `removeColumns()` 时，必须在从模型的底层数据存储中删除数据之后调用此函数。

参见 `beginRemoveColumns()`。

#### `endRemoveRows`

```
[ protected ] void QAbstractItemModel::endRemoveRows()
```

结束行删除操作。

在子类中重新实现 `removeRows()` 时，必须在从模型的底层数据存储中删除数据之后调用此函数。

参见 `beginRemoveRows()`。

#### `endResetModel`

```
[ protected ] void QAbstractItemModel::endResetModel()
```

完成模型重置操作。

在重置模型或代理模型中的任何内部数据结构后，必须调用此函数。

该函数发出 `modelReset()` 信号。

该函数在 Qt4.6 中被引入。

参见 `beginResetModel()`。

#### `fetchMore`

```
[ virtual ] void QAbstractItemModel::fetchMore(const QModelIndex &parent)
```

获取指定的 `parent` 父索引的项的任何可用的数据。

如果递增地填充模型，则需要重新实现该函数。

该函数的默认实现没有做任何事情。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `canFetchMore()`。

#### `flags`

```
[ virtual ] Qt::ItemFlags QAbstractItemModel::flags(const QModelIndex &index) const
```

返回给定索引的项标志。

基类的实现返回启用项 (`ItemIsEnabled`) 和允许选择项 (`ItemIsSelectable`) 的标志的组合。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `Qt::ItemFlags`。

#### `hasChildren`

```
[ virtual ] Qt::ItemFlags QAbstractItemModel::hasChildren(const QModelIndex &parent = QModelIndex()) const
```

如果父索引 `parent` 有任何子项则返回 `true`，否则返回 `false`。

在父索引上使用 `rowCount()` 来查找子项的数量。

注意，如果一个索引设置了 `Qt::ItemNeverHasChildren` 标志，那么用该索引调用该方法是未定义的行为。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `parent()` 和 `index()`。

#### `hasIndex`

```
[ virtual ] bool QAbstractItemModel::hasIndex(int row, int column, const QModelIndex &parent = QModelIndex()) const
```

如果模型返回一个指定父索引 `parent`、行 `row` 和列 `column` 的有效 `QModelIndex`，则返回 `true`，否则返回 `false`。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

#### `headerData`

```
[ virtual ] QVariant QAbstractItemModel::headerData(int section, Qt::Orientation orientation, int role = Qt::DisplayRole) const
```

返回标题中具有给定 `orientation` 和 `section` 的数据。

对于水平标题，节号对应于列号。类似地，对于垂直标题，节号对应于行号。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `Qt::ItemDataRole`、`setHeaderData()` 和 `QHeaderView`。

#### `index`

```
[ pure virtual ] QModelIndex QAbstractItemModel::index(int row, int column, const QModelIndex &parent = QModelIndex()) const
```

返回模型中指定 `row`、`column` 和 `parent` 索引的项的索引。

在子类中重新实现此函数时，调用 `createIndex()` 来生成模型索引，其他组件可以使用这些索引来引用模型中的项。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `createIndex()`。

#### `insertColumn`

```
bool QAbstractItemModel::insertColumn(int column, const QModelIndex &parent = QModelIndex()) const
```

在指定 `parent` 索引的子项的指定 `column` 之前插入一列。

如果插入了该列，则返回 `true`；否则，返回 `false`。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `insertColumns()`、`insertRow()` 和 `removeColumn()`。

## insertColumns

```
[ virtual ] bool QAbstractItemModel::insertColumns(int column, int count, const QModelIndex &parent = QModelIndex()) const
```

在支持此功能的模型上，在模型中的给定 `column` 之前插入 `count` 列新列。每个新列中的项将是由父模型索引 `parent` 表示的项的子项目。

如果 `column` 为 0，则这些列将添加到任何现有列的前面。

如果 `column` 为 `columnCount()`，则将这些列追加到任何现有列之后。

如果 `parent` 没有子项，插入带有 `count` 列的单行。

如果列成功插入，则返回 `true`，否则返回 `false`。

基类的实现没有做任何事情，并且返回 `false`。

如果您实现了自己的模型，希望支持插入，则可以重新实现此函数。或者，您可以提供自己的 API 来更改数据。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `insertRows()`、`removeColumns()`、`beginInsertColumns()` 和 `endInsertColumns()`。

## insertRow

```
bool QAbstractItemModel::insertRow(int row, const QModelIndex &parent = QModelIndex()) const
```

在指定 `parent` 索引的子项的指定 `row` 之前插入一列。

注意：该函数调用了虚函数 `insertRows()`。

如果插入了该行，则返回 `true`；否则，返回 `false`。

参见 `insertRows()`、`insertColumn()` 和 `removeRow()`。

## insertRows

```
[ virtual ] bool QAbstractItemModel::insertRows(int row, int count, const QModelIndex &parent = QModelIndex()) const
```

注意：基类的实现没有做任何事情，并且返回 `false`。

在支持此功能的模型上，将 `count` 行插入模型中给定的行之之前。新行中的项将是由父模型索引 `parent` 表示的项的子项。

如果 `row` 为 `0`，则这些行将添加到任何现有列的前面。

如果 `row` 为 `rowCount()`，则将这些列追加到任何现有行之后。

如果 `parent` 没有子项，插入带有 `count` 行的单列。

如果列成功插入，则返回 `true`，否则返回 `false`。

如果您实现了自己的模型，希望支持插入，则可以重新实现此函数。或者，您可以提供自己的 API 来更改数据。在任何一种情况下，您都需要调用 `beginInsertRows()` 和 `endInsertRows()` 来通知其他组件模型已经更改

参见 `insertColumns()`、`removeRows()`、`beginInsertRows()` 和 `endInsertRows()`。

#### itemData

```
[ virtual ] QMap<int, QVariant> QAbstractItemModel::itemData(const QModelIndex &index) const
```

为给定索引 `index` 处的项返回具有模型中所有预定义角色值的 `map`。

如果希望扩展此函数的默认行为以在 `map` 中包含自定义角色，请重新实现此函数。

参见 `setItemData()`、`Qt::ItemDataRole` 和 `data()`。

#### match

```
[ virtual ] QModelIndexList QAbstractItemModel::match(const QModelIndex &start, int role, const QVariant &value, int hits = 1, Qt::MatchFlags flags = Qt::MatchFlags(Qt::MatchStartsWith|Qt::MatchWrap)) const
```

返回一系列项的索引，这些索引在 `start` 索引的列中并且该索引下的数据在给定角色下存储的数据与指定值匹配。执行搜索的方式由给定的 `flags` 定义。返回的列表可能是空的。还要注意，如果使用了代理模型，列表中结果的顺序可能与模型中的顺序不一致。不能依赖结果的顺序。

搜索从 `start` 索引开始，直到匹配数据项的数量等于 `hits`，搜索到达最后一行，或者再次搜索到达 `start`—这取决于是否在 `flags` 中指定了 `MatchWrap`。如果希望搜索所有匹配项，请使用 `hits = -1`。

默认情况下，此函数将对所有项执行基于字符串的包装比较，搜索以 `value` 指定的搜索项开头的项。

注意：这个函数的默认实现只搜索列。重新实现此函数以包含不同的搜索行为。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

#### mimeData

```
[ virtual ] QMimeData QAbstractItemModel::mimeData(const QModelIndexList &indexes) const
```

返回一个对象，该对象包含与指定索引 `indexes` 列表对应的序列化数据项。用于描述编码数据的格式是从 `mimeTypeNames()` 函数获得的。这个默认实现使用 `mimeTypeNames()` 的默认实现返回的默认 **MIME** 类型。如果您在自定义模型中重新实现 `mimeTypeNames()` 以返回更多 **MIME** 类型，那么重新实现此函数以使用它们。

如果 `indexes` 为空，或者没有受支持的 **MIME** 类型，则返回 `0`，而不是序列化的空列表。

参见 `mimeTypeNames()` 和 `dropMimeData()`。

## `mimeTypeNames`

```
[ virtual ] QStringList QAbstractItemModel::mimeTypeNames() const
```

返回允许的 **MIME** 类型的列表。默认情况下，内置模型和视图使用内部 **MIME** 类型：`application / x-qabstractitemmodeldata`。

在自定义模型中实现拖放支持时，如果您将以默认内部 **MIME** 类型以外的格式返回数据，请重新实现此函数以返回您的 **MIME** 类型列表。

如果在自定义模型中重新实现这个函数，还必须重新实现调用它的成员函数：`mimeTypeNames()` 和 `dropMimeData()`。

参见 `mimeTypeNames()` 和 `dropMimeData()`。

## `moveColumn`

```
bool QAbstractItemModel::moveColumn(const QModelIndex &sourceParent, int
sourceColumn, const QModelIndex &destinationParent, int destinationChild)
```

在支持此功能的模型上，将 `sourceColumn` 从 `sourceParent` 移到 `destinationParent` 下的 `destinationChild`。

如果列被成功移动，则返回 `true`；否则返回 `false`。

参见 `moveColumns()` 和 `moveRow()`。

## `moveColumns`

```
[ virtual ] bool QAbstractItemModel::moveColumns(const QModelIndex &sourceParent, int sourceColumn, int count, const QModelIndex &destinationParent, int destinationChild)
```

在支持此功能的模型上，将 `count` 列从父索引 `sourceParent` 下的给定 `sourceColumn` 移到父索引 `destinationParent` 下的 `destinationChild` 列。

如果列被成功移动，则返回 `true`；否则返回 `false`。

基类的实现没有做任何事情，并且返回 `false`。

如果实现自己的模型，则如果要支持移动，则可以重新实现此功能。另外，您可以提供自己的 **API** 来更改数据。

参见 `beginMoveColumns()` 和 `endMoveColumns()`。

## `moveRow`

```
bool QAbstractItemModel::moveRow(const QModelIndex &sourceParent, int sourceRow,
const QModelIndex &destinationParent, int destinationChild)
```

在支持此功能的模型上，将 `sourceColumn` 从 `sourceParent` 移到 `destinationParent` 下的 `destinationChild`。

如果行被成功移动，则返回 `true`；否则返回 `false`。

参见 `moveRows()` 和 `moveColumn()`。

#### `moveRows`

```
[ virtual ] bool QAbstractItemModel::moveRows(const QModelIndex &source-
Parent, int sourceRow, int count, const QModelIndex &destinationParent,
int destinationChild)
```

在支持此功能的模型上，将 `count` 行从父索引 `sourceParent` 下的给定 `sourceColumn` 移到父索引 `destinationParent` 下的 `destinationChild` 行。

如果行被成功移动，则返回 `true`；否则返回 `false`。

基类的实现没有做任何事情，并且返回 `false`。

如果实现自己的模型，则如果要支持移动，则可以重新实现此功能。另外，您可以提供自己的 API 来更改数据。

参见 `beginMoveRows()` 和 `endMoveRows()`。

#### `parent`

```
[ pure virtual ] bool QAbstractItemModel::parent(const QModelIndex &in-
dex) const
```

返回具有给定索引 `index` 的模型项的父项。如果该项没有父项，则返回无效的 `QModelIndex`。

在公开树数据结构的模型中使用的常见约定是，只有第一列中的项才有子级。对于这种情况，当在子类中重新实现此函数时，返回的 `QModelIndex` 的列将为 0。

在子类中重新实现这个函数时，要小心避免调用 `QModelIndex` 成员函数，比如 `QModelIndex::parent()`，因为模型的索引将简单地调用实现，从而导致无限递归。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `createIndex()`。

#### `persistentIndexList`

```
[ protected ] QModelIndexList QAbstractItemModel::persistentIndexList()
const
```

返回作为模型中的持久索引存储的索引列表。

该函数在 Qt4.2 中被引入。

#### `removeColumn`

```
bool QAbstractItemModel::removeColumn(int column, const QModelIndex &parent = QModelIndex())
```

从指定的父项 `parent` 的子项中删除给定的列 `column`。

如果删除了该列，则返回 `true`；否则返回 `false`。

参见 `removeColumns()`、`removeRow()` 和 `insertColumn()`。

#### `removeColumns`

```
[ virtual ] bool QAbstractItemModel::removeColumns(int column, int count, const QModelIndex &parent = QModelIndex())
```

在支持此功能的模型上，从模型中删除以父项 `parent` 下给定列 `column` 开头的 `count` 列。

如果列被成功删除，返回 `true`；否则返回 `false`。

基类的实现没有做任何事情并返回了 `false`。

如果实现自己的模型，要支持删除，则可以重新实现此函数。另外，您可以提供自己的 API 来更改数据。

参见 `removeColumn()`、`removeRows()`、`insertColumns()`、`beginRemoveColumns()` 和 `endRemoveColumns()`。

#### `removeRow`

```
bool QAbstractItemModel::removeRow(int row, const QModelIndex &parent = QModelIndex())
```

从指定的父项 `parent` 的子项中删除给定的行 `row`。

如果删除了该行，则返回 `true`；否则返回 `false`。

这是一个调用 `removeRows()` 的便利函数。`QAbstractItemModel` 的 `removeRows()` 的实现不做任何事情。

参见 `removeRows()`、`removeColumn()` 和 `insertRow()`。

#### `removeRows`

```
[ virtual ] bool QAbstractItemModel::removeRows(int row, int count, const QModelIndex &parent = QModelIndex())
```

在支持此功能的模型上，从模型中删除以父项 `parent` 下给定列 `row` 开头的 `count` 行。

如果行被成功删除，返回 `true`；否则返回 `false`。

基类的实现没有做任何事情并返回了 `false`。

如果实现自己的模型，要支持删除，则可以重新实现此函数。另外，您可以提供自己的 API 来更改数据。

参见 `removeRow()`、`removeColumns()`、`insertColumns()`、`beginRemoveRows()` 和 `endRemoveRows()`。

## roleNames

```
[ virtual ] QHash<int, QByteArray> QAbstractItemModel::roleNames() const
```

返回模型的角色名称。

Qt 设置的默认角色名是：

Qt Role	QML Role Name
Qt::DisplayRole	display
Qt::DecorationRole	decoration
Qt::EditRole	edit
Qt::ToolTipRole	toolTip
Qt::StatusTipRole	statusTip
Qt::WhatsThisRole	whatsThis

参见 `removeRow()`、`removeColumns()`、`insertColumns()`、`beginRemoveRows()` 和 `endRemoveRows()`。

该函数在 Qt4.6 中被引入。

参见 `setRoleNames()`。

## rowCount

```
[ pure virtual ] int QAbstractItemModel::rowCount(const QModelIndex &parent = QModelIndex()) const
```

返回给定父节点 `parent` 下的行数。当父节点有效时，这意味着 `rowCount` 返回父节点的子节点数。

注意：在实现基于表的模型时，当父节点有效时，`rowCount()` 应该返回 0。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `columnCount()`。

## setData

```
[ virtual ] bool QAbstractItemModel::setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole)
```

将索引 `index` 处的项的角色数据设置为 `value`。

成功返回 `true`；否则返回 `false`。

如果数据被成功设置，应该发出 `dataChanged()` 信号。

基类的实现返回 `false`。对于可编辑的模型来说，该函数和 `data()` 必须被实现。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `Qt::ItemDataRole`、`data()` 和 `itemData()`。

## setHeaderData



```
[ virtual ] bool QAbstractItemModel::setHeaderData(int section, Qt::Orientation orientation, const QVariant &value, int role = Qt::EditRole)
```

设置指定 `section`、`orientation` 和 `role` 标题的数据为 `value`。

标题数据更新完成，返回 `true`；否则返回 `false`。

在重新实现此函数时，必须显式发出 `headerDataChanged()` 信号。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `Qt::ItemDataRole` 和 `headerData()`。

## setItemData

```
[ virtual ] bool QAbstractItemModel::setItemData(const QModelIndex &index, const QMap<int, QVariant> &roles)
```

对于每个 `Qt::ItemDataRole`，将索引 `index` 处的项目的角色数据设置为角色中的关联值。

设置成功，返回 `true`；否则返回 `false`。

不在角色中的角色将不会被修改。

参见 `setData()`、`data()` 和 `itemData()`。

## sibling

```
[ virtual ] QModelIndex QAbstractItemModel::sibling(int row, int column, const QModelIndex &) const
```

返回索引 `index` 项的行和列上的同级索引，如果该位置上没有同级索引，则返回无效的 `QModelIndex`。

`sibling()` 只是一个便捷函数，它找到项的父项，并使用它来检索指定行和列中子项的索引。

可以选择性地重写此方法以进行特定于实现的优化。

注意：该函数可以通过元对象系统和 QML 调用。请参阅 `Q_INVOKABLE`。

参见 `index()`、`QModelIndex::row()` 和 `QModelIndex::column()`。

## sort

```
[ virtual ] void QAbstractItemModel::sort(int column, Qt::SortOrder order = Qt::AscendingOrder)
```

按给定顺序 `order` 按列 `column` 对模型进行排序。

基类实现不执行任何操作。

## span

```
[ virtual ] QSize QAbstractItemModel::span(const QModelIndex &index) const
```

返回由索引 `index` 表示的项的行和列跨度。

注意：目前没有使用 `span`。

#### `supportedDragActions`

```
[ virtual ] Qt::DropActions QAbstractItemModel::supportedDragActions()  
const
```

返回此模型中数据支持的操作。

默认实现返回 `supportedDropActions()`。如果希望支持其他操作，请重新实现此函数。

当发生拖动时，`QAbstractItemView::startDrag()` 使用 `supportedDragActions()` 作为默认值。

注意：目前没有使用 `span`。

参见 `setSupportedDragActions()`、`Qt::DropActions` 和在项视图中使用拖放。

#### `supportedDropActions`

```
[ virtual ] Qt::DropActions QAbstractItemModel::supportedDropActions()  
const
```

返回此模型支持的放置动作。

默认实现返回 `Qt::CopyAction`。如果希望支持其他操作，请重新实现此函数。您还必须重新实现 `dropMimeData()` 函数来处理额外的操作。

该函数在 Qt4.2 中被引入。

参见 `dropMimeData()`、`Qt::DropActions` 和在项视图中使用拖放。

## 第十章 QAbstractSocket

QAbstractSocket 类是 Qt 中 Socket 通信类的基类，被 QTcpSocket 和 QUdpSocket 等类继承。QAbstractSocket 类为所有的 socket 通信类提供了最基本的功能。

属性	方法
头文件	<code>#include&lt;QAbstractSocket&gt;</code>
qmake	<code>QT+=network</code>
父类	<code>QIODevice</code>
子类	<code>QTcpSocket</code> 、 <code>QUdpSocket</code>



# 第十一章 QSql

## QSql 命名空间

QSql 命名空间里的各种各样的标识符，已经被运用在 Qt SQL 各个模块中。[更多](#)

属性	方法
头文件	<code>#include &lt;QSql&gt;</code>
qmake	<code>QT += sql</code>

## 类型

enum	Location { BeforeFirstRow, AfterLastRow }
enum	NumericalPrecisionPolicy { LowPrecisionInt32, LowPrecisionInt64,LowPrecisionDouble,HighPrecision}
flags	ParamType
enum	ParamTypeFlag { In, Out, InOut, Binary }
enum	TableType { Tables, SystemTables, Views, AllTables }

## 细节的介绍

查看 [Qt SQL](#)

## 类型文档

### enum QSql::Location

此枚举类型描述特殊的 sql 导航位置

常量	值	介绍
<code>QSql::BeforeFirstRow</code>	-1	在第一个记录之前
<code>QSql::AfterLastRow</code>	-2	在最后一个记录之后

另请参阅 [QSqlQuery::at\(\)](#)

### enum QSql::NumericalPrecisionPolicy

数据库中的数值可以比它们对应的 C++ 类型更精确。此枚举列出在应用程序中表示此类值的策略。

常量	值	介绍
<code>QSql::LowPrecisionInt32</code>	0x01	对于 32 位的整形数值。在浮点数的情况下，小数部分将会被舍去。
<code>QSql::LowPrecisionInt64</code>	0x02	对于 64 位的整形数值。在浮点数的情况下，小数部分将会被舍去。
<code>QSql::LowPrecisionDouble</code>	0x04	强制双精度值。这个默认的规则
<code>QSql::HighPrecision</code>	0	字符串将会维持精度

注意：如果特定的驱动发生溢出，这是一个真实行为。像 **Oracle** 数据库在这种情形下，就会返回一个错误。

```
enum QSql::ParamTypeFlag
```

```
flags QSql::ParamType
```

这个枚举用于指定绑定参数的类型

常量	值	介绍
<code>QSql::In</code>	<code>0x00000001</code>	这个参数被用于向数据库里写入数据
<code>QSql::Out</code>	<code>0x00000002</code>	这个参数被用于向数据库里获得数据
<code>QSql::InOut</code>	<code>In   Out</code>	这个参数被用于向数据库里写入数据；使用查询来向数据库里，重写数据
<code>QSql::Binary</code>	<code>0x00000004</code>	如果您想显示数据为原始的二进制数据，那么必须是 <code>OR'd</code> 和其他的标志——

类型参数类型定义为 **QFlags**。它被存放在一个 **OR** 与类型参数标志的值的组合。

## 第十二章 QSqlDatabase

QSqlDatabase 类用于处理数据库的连接

属性	方法
头文件	#include <QSqlDatabase>
qmake	QT += sql

列出所有的成员，包括继承成员

公共类型

返回值	函数名
	QSqlDatabase(const QSqlDatabase &other)
	QSqlDatabase()
QSqlDatabase &	operator=(const QSqlDatabase &other)
	QSqlDatabase()
void	close()
bool	commit()
QString	connectOptions() const
QString	connectionName() const
QString	databaseName() const
QSqlDriver *	driver() const
QString	driverName() const
QSqlQuery	exec(const QString &query = QString()) const
QString	hostName() const
bool	isOpen() const
bool	isOpenError() const
bool	isValid() const
QSqlError	lastError() const
QSql::NumericalPrecisionPolicy	numericalPrecisionPolicy() const
bool	open()
bool	open(const QString &user, const QString &password)
QString	password() const
int	port() const
QSqlIndex	primaryIndex(const QString &tablename) const
QSqlRecord	record(const QString &tablename) const
bool	rollback()
void	setConnectOptions(const QString &options = QString())
void	setDatabaseName(const QString &name)
void	setHostName(const QString &host)
void	setNumericalPrecisionPolicy(QSql::NumericalPrecisionPolicy precisionPolicy)
void	setPassword(const QString &password)
void	setPort(int port)
void	setUserName(const QString &name)
QStringList	tables(QSql::TableType type = QSql::Tables) const
bool	transaction()
QString	userName() const

## 静态公共成员

返回值	函数名
QSqlDatabase	<code>addDatabase(const QString &amp;type, const QString &amp;connectionName = QLatin1String(defaultConnection))</code>
QSqlDatabase	<code>addDatabase(QSqlDriver *driver, const QString &amp;connectionName = QLatin1String(defaultConnection))</code>
QSqlDatabase	<code>cloneDatabase(const QSqlDatabase &amp;other, const QString &amp;connectionName)</code>
QSqlDatabase	<code>cloneDatabase(const QString &amp;other, const QString &amp;connectionName)</code>
QStringList	<code>connectionNames()</code>
bool	<code>contains(const QString &amp;connectionName = QLatin1String(defaultConnection))</code>
QSqlDatabase	<code>database(const QString &amp;connectionName = QLatin1String(defaultConnection), bool open = true)</code>
QStringList	<code>drivers()</code>
bool	<code>isDriverAvailable(const QString &amp;name)</code>
void	<code>registerSqlDriver(const QString &amp;name, QSqlDriverCreatorBase *creator)</code>
void	<code>removeDatabase(const QString &amp;connectionName)</code>

## 受保护的成员函数

返回值	函数名
	<code>QSqlDatabase(QSqlDriver *driver)</code>
	<code>QSqlDatabase(const QString &amp;type)</code>

## 详细的介绍

**QSqlDatabase** 类提供接口用于数据库的连接。一个 **QSqlDatabase** 实例对象表示连接。这个连接提供数据库所需要的驱动，这个驱动来自于 **QSqlDriver**。换言之，您可以实现自己的数据库驱动，通过继承 **QSqlDriver**。查看[如何实现自己的数据库驱动](#)来获取更多的信息。

通过调用一个静态的 `addDatabase()` 函数，来创建一个连接（即：实例化一个 **QSqlDatabase** 类），并且可以指定驱动或者驱动类型去使用（依赖于数据库的类型）和一个连接的名称。一个连接是通过它自己的名称，而不是通过数据库的名称去连接的。对于一个数据库您可以有多个连接。**QSqlDatabase** 也支持默认连接，您可以不传递连接名参数给 `addDatabase()` 来创建它。随后，这个默认连接假定您在调用任何静态函数情况下，而不去指定连接名称。下面的一段代码片段展示了如何去创建和打开一个默认连接，去连接 PostgreSQL 数据库：

```
QSqlDatabase db = QSqlDatabase::database();
```

**QSqlDatabase** 是一个值类。通过一个 **QSqlDatabase** 实例对数据库连接所做的操作将影响表示相同连接的其他 **QSqlDatabase** 实例。使用 `cloneDatabase()` 在基于已存在数据库的连接来创建独立的数据库的连接。

警告：强烈建议不要将 **QSqlDatabase** 的拷贝作为类成员，因为这将阻止关闭时正确清理实例。如果需要访问已经存在 **QSqlDatabase**，应该使用 `database()` 访问。如果您选择使用作为成员变量的 **QSqlDatabase**，则需要在删除 **QCoreApplication** 实例之前删除它，否则可能会导致未定义的行为。

如果您想创建多个数据库连接，可以调用 `addDatabase()`，并且给一个独一无二的参数（即：连接名称）。使用带有连接名的 `database()` 函数，来获取该连接。使用带有连接名的 `removeDatabase()` 函数，来删除一个连接。如果尝试删除由其他 **QSqlDatabase** 对象引用的连接，**QSqlDatabase** 将输出警告。可以使用 `contains()` 查看给定的连接名是否在连接列表中。



	一些实用的方法
<code>tables()</code>	返回数据表的列表
<code>primaryIndex()</code>	返回数据表的主索引
<code>record()</code>	返回数据表字段的元信息
<code>transaction()</code>	开始一个事务
<code>commit()</code>	保存并完成一个事务
<code>rollback()</code>	取消一个事务
<code>hasFeature()</code>	检查驱动程序是否支持事务
<code>lastError()</code>	返回有关上一个错误的信息
<code>drivers()</code>	返回可用的数据库驱动名称
<code>isDriverAvailable()</code>	检查特定驱动程序是否可用
<code>registerSqlDriver()</code>	注册自定义驱动程序

注意: `QSqlDatabase::exec()` 方法已经被弃用。请使用 `QSqlQuery::exec()`

注意: 使用事务时, 必须在创建查询之前启动事务。

成员函数文档

**[protected]** `QSqlDatabase::QSqlDatabase(QSqlDriver *driver)`

这是一个重载函数

使用给定驱动程序来创建连接

**[protected]** `QSqlDatabase::QSqlDatabase(const QString &type)`

这是一个重载函数

通过引用所给的数据库驱动类型来创建一个连接。如果不给定数据库驱动类型, 那么这个数据库连接将会没有什么作用。

当前可用的驱动类型:

驱动类别	介绍
<code>QDB2</code>	IBM DB2
<code>QIBASE</code>	Borland InterBase 驱动
<code>QMYSQL</code>	MySQL 驱动
<code>QOCI</code>	Oracle 调用的接口驱动
<code>QODBC</code>	ODBC 驱动 (包含 Microsoft SQL Server)
<code>QPSQL</code>	PostgreSQL 驱动
<code>QSQLITE</code>	SQLite 第三版本或者以上
<code>QSQLITE2</code>	SQLite 第二版本
<code>QTDS</code>	Sybase Adaptive Server

其他第三方驱动程序, 包括自己自定义的驱动程序, 都可以动态加载。

请参阅 [SQL Database Drivers](#), `registerSqlDriver()` 和 `drivers()`。

```
QSqlDatabase::QSqlDatabase(const QSqlDatabase &other)
```

创建一个其它的副本

`QSqlDatabase::QSqlDatabase()` 创建一个无效的 `QSqlDatabase` 空对象。使用 `addDatabase()`, `removeDatabase()` 和 `database()` 来获得一个有效的 `QSqlDatabase` 对象。

```
QSqlDatabase &QSqlDatabase::operator=(const QSqlDatabase &other)
```

给这个对象赋一个其他其他对象的值

```
QSqlDatabase:: QSqlDatabase()
```

销毁这个对象，并且释放所有配置的资源注意：当最后的连接被销毁，这个析构函数就会暗中的调用 `close()` 函数，去删除这个数据库的其他连接。

另请参阅 `close()`。

```
[static] QSqlDatabase QSqlDatabase::addDatabase(const QString &type,const  
QString &connectionName = QLatin1String(defaultConnection))
```

使用驱动程序类型和连接名称，将数据库添加到数据库连接列表中。如果存在相同的连接名，那么这个连接将会被删除。

通过引用连接名，来返回一个新的连接。

如果数据库的类别不存在或者没有被加载，那么 `isValid()` 函数将会返回 `false`

如果我们没有指定连接名参数，那么应用程序就会返回默认连接。如果我们提供了连接名参数，那么可以使用 `database(connectionName)` 函数来获取该连接。

警告：如果您指定了相同的连接名参数，那么就会替换之前的那个相同的连接。如果您多次调用这个函数而不指定连接名参数，则默认连接将被替换。

在使用连接之前，它必须经过初始化。比如：调用下面一些或者全部 `setDatabaseName()`、`setUserName()`、`setPassword()`、`setHostName()`、`setPort()` 和 `setConnectOptions()`，并最终调用 `open()`

注意：这个函数是线程安全的

请查看 `database()`，`removeDatabase()` 以及 [线程和 SQL 单元](#)。

```
[static] QSqlDatabase QSqlDatabase::addDatabase(QSqlDriver *driver, const  
QString &connectionName = QLatin1String(defaultConnection))
```

这个重载函数是非常有用的，当您想创建一个带有[驱动](#)连接时，您可以实例化它。有可能您想拥有自己的数据库驱动，或者去实例化 Qt 自带的驱动。如果您真的想这样做，我非常建议您把驱动的代码导入到您的应用程序中。例如，您可用自己的 QPSQL 驱动来创建一个 PostgreSQL 连接，像下面这样：

```
PGconn *con = PQconnectdb("host=server@user=bart@password=simpson@dbname=springfield");  
QPSQLDriver *drv = new QPSQLDriver(con);  
QSqlDatabase db = QSqlDatabase::addDatabase(drv); // 产生成新的默认连接  
QSqlQuery query;  
query.exec("SELECT@NAME,@ID@FROM@STAFF");
```

上面的代码用于设置一个 PostgreSQL 连接和实例化一个 QPSQLDriver 对象。接下来，addDatabase() 被调用产生一个已知的连接，以便于它可以使用 Qt SQL 相关的类。Qt 假定您已经打开了数据库连接，当使用连接句柄（或一组句柄）实例化驱动程序时。

注意：我们假设 qtdir 是安装 Qt 的目录。假定您的 PostgreSQL 头文件已经包含在搜索路径中，然后这里才能引用所需要的 PostgreSQL 客户端库和去实例化 QPSQLDriver 对象。

请记住，必须将数据库客户端库到您的程序里。确保客户端库在您的链接器的搜索路径中，并且像下面这样添加到您的 .pro 文件里：

```
unix:LIBS += -lpq
win32:LIBS += libpqdll.lib
```

这里介绍了所有驱动支持的方法。只有驱动的构造参数有所不同。列举了一个关于 Qt 附带的程序，以及它们的源代码文件，和它们的构造函数参数的列表：

驱动	类名	构造器参数	用于导入的文件
QPSQL	QPSQLDriver	PGconn *connection	qsq_sql.cpp
QMYSQL	QMYSQLDriver	MYSQL *connection	qsq_mysql.cpp
QOCI	QOCIDriver	OCIEnv *environment, OCISvcCtx *serviceContext	qsq_oci.cpp
QODBC	QODBCDriver	SQLHANDLE environment, SQLHANDLE connection	qsq_odbc.cpp
QDB2	QDB2	SQLHANDLE environment, SQLHANDLE connection	qsq_db2.cpp
QTDS	QTDSDriver	LOGINREC *loginRecord, DBPROCESS *dbProcess, const QString &hostName	qsq_tds.cpp
QSQLITE	QSQLiteDriver	sqlite *connection	qsq_sqlite.cpp
QIBASE	QIBaseDriver	isc_db_handle connection	qsq_ibase.cpp

当构造用于为内部查询创建新连接的 QTDSDriver 时，需要主机名（或服务名）。这是为了防止在同时使用多个 QSqlQuery 对象时发生阻塞。

警告：添加一个存在连接名的连接时，这个新添加的连接将会替换另一个。警告：SQL 框架拥有驱动程序的所有权。它不能被删除。可以使用 removeDatabase()，去删除这个连接。另请参阅 drivers()

[protected] QSqlDatabase QSqlDatabase::cloneDatabase(const QString &other, const QString &connectionName)

克隆其他数据库连接并将其存储为 connectionName。原始数据库中的所有设置，例如 databaseName()、hostName() 等，都会被复制。如果其他数据库无效，则不执行任何操作。返回最新被创建的数据库连接。

注意：这个新的连接不能被打开。您必须调用 open()，才能使用这个新的连接。

[static] QSqlDatabase QSqlDatabase::cloneDatabase(const QString &other, const QString &connectionName) 这是个重载函数。

克隆其他数据库连接并将其存储为 connectionName。原始数据库中的所有设置，例如 databaseName()、hostName() 等，都会被复制。如果其他数据库无效，则不执行任何操作。返回最新被创建的数据库连接。

注意：这个新的连接不能被打开。您必须调用 open()，才能使用这个新的连接。

当我们在另一个线程克隆这个数据库，这个重载是非常有用的。

qt5.13 中引入了这个函数。

**void QSqlDatabase::close()** 关闭数据库连接，释放获取的所有资源，并使与数据库一起使用的任何现有 **QSqlQuery** 对象无效

这个函数也会影响它的 **QSqlDatabase** 对象副本。

另请参阅 **removeDatabase()**

**bool QSqlDatabase::commit()** 如果驱动支持事务和一个 **transaction()** 已经被启动，那就可以提交一个事务到这个数据库中。如果这个操作成功，就会返回 **true**。否则返回 **false**。

注意：对于一些数据库，如果对数据库使用 **SELECT** 进行查询操作，将会提交失败并且返回 **false**。在执行提交之前，使查询处于非活动状态。

调用 **lastError()** 函数获取错误信息。

另请参阅 **QSqlQuery::isActive()**, **QSqlDriver::hasFeature()**, 和 **rollback()**。

**QString QSqlDatabase::connectOptions() const** 返回用于此连接的连接选项字符串。这个字符串可能是空。

另请参阅 **setConnectOptions()**

**QString QSqlDatabase::connectionName() const** 返回连接名，它有可能为空。

注意：这个连接名不是数据库名

qt4.4 中引入了这个函数。

另请参阅 **addDatabase()**

**[static] QStringList QSqlDatabase::connectionNames()** 返回包含所有连接名称的列表。

注意：这个函数是线程安全的。

另请参阅 **contains()**, **database()**, 和线程和 **SQL** 模块

**[static] bool QSqlDatabase::contains(const QString &connectionName = QLatin1String(defaultConnectionName))**

如果所给的连接名，包含在所给的数据库连接列表里，那么就返回 **true**；否则返回 **false**。

注意：这个函数是线程安全的

另请参阅 **connectionNames()**, **database()** 和线程和 **SQL** 模块。

**[static] QSqlDatabase QSqlDatabase::database(const QString &connectionName = QLatin1String(defaultConnectionName), bool open = true)**

返回一个调用 **connectionName** 参数的数据库连接。这个数据库连接使用之前，必须已经通过 **addDatabase()** 函数进行添加。如果 **open** 为 **true**（默认值），并且数据库连接尚未打开，则现在打开它。如果未指定连接名参数，则使用默认连接。如果连接名不存在数据库列表中，那么将会返回一个非法的连接。

注意：这个函数是线程安全的

另请参阅 `isOpen()` 和线程和 SQL 模块。

```
QString QSqlDatabase::databaseName() const
```

返回连接的连接数据库名称，当然它也可能是空的。

注意：这个数据库名不是连接名

另请参阅 `setDatabaseName()`。

```
QSqlDriver *QSqlDatabase::driver() const
```

返回被使用的数据库连接的所使用的数据库驱动。

另请参阅 `addDatabase()` 和 `drivers()`

```
QString QSqlDatabase::driverName() const
```

返回连接的驱动名称

另请参阅 `addDatabase()` 和 `driver()`

```
[static] QStringList QSqlDatabase::drivers()
```

返回一个可使用的数据库驱动列表另请参阅 `registerSqlDriver()`

```
QSqlQuery QSqlDatabase::exec(const QString &query = QString()) const
```

在这个数据库里执行 SQL 表达式并返回一个 `QSqlQuery` 对象。使用 `lastError()` 来获取错误的信息。

如果查询为空，则返回一个空的、无效的查询。并且 `lastError()`。

另请参阅 `QSqlQuery` 和 `lastError()`。

```
QString QSqlDatabase::hostName() const
```

返回连接的主机名；它有可能为空。

另请参阅 `setHostName()`

```
[static] bool QSqlDatabase::isDriverAvailable(const QString &name)
```

如果调用一个叫 `name` 的驱动，是可以使用的，那么就返回 `true`；反之返回 `false`。

另请参阅 `drivers()`。

```
bool QSqlDatabase::isOpen() const
```

如果当前数据库连接是打开的，那么就返回 `true`，否则返回 `false`。

```
bool QSqlDatabase::isOpenError() const
```

如果打开数据库的连接有错误，那么就返回 `true`，否则返回 `false`。可以调用 `lastError()` 函数去获取相关的错误信息。

```
bool QSqlDatabase::isValid() const
```

如果 `QSqlDatabase()` 有一个有效的驱动，那么就返回 `true`。

例子：

```
QSqlDatabase db;
QDebug() << db.isValid(); // 返回 false

db = QSqlDatabase::database("sales");
QDebug() << db.isValid(); // 如果 "sales" 连接存在，就返回 true

QSqlDatabase::removeDatabase("sales");
QDebug() << db.isValid(); // 返回 false
```

**QSqlError** `QSqlDatabase::lastError() const`

返回这个数据库出现的最新错误信息。

使用 `QSqlQuery::lastError()` 函数来获取一个单个查询上的错误。

另请参阅 `QSqlError` and `QSqlQuery::lastError()`。

**QSql::NumericalPrecisionPolicy** `QSqlDatabase::numericalPrecisionPolicy() const`

返回数据库连接的当前默认精度策略。

qt4.6 中引入了这个函数。

另请参阅 `QSql::NumericalPrecisionPolicy`, `setNumericalPrecisionPolicy()`、

`QSqlQuery::numericalPrecisionPolicy()` 和 `QSqlQuery::setNumericalPrecisionPolicy()`。

**bool** `QSqlDatabase::open()`

使用当前连接值打开数据库连接。如果操作成功就返回 `true`；反之返回 `false`。可以调用 `lastError()` 来获取错误的信息。

另请参阅 `lastError()`、`setDatabaseName()`、`setUserName()`、

`setPassword()`、`setHostName()`、`setPort()` 和 `setConnectOptions()`。

**bool** `QSqlDatabase::open(const QString &user, const QString &password)`

这是一个重载函数。

使用所给的 `username` 和 `password` 两个参数，打开数据连接，如果成功就返回 `true`；反之返回 `false`。使用 `[lastError()]QSqlDatabase.md#qsqlerror-qsqldatabaselasterror-const` 来获取错误的信息。

这个函数不存放所给的 `password` 参数，相反的它会把 `password` 参数直接传给驱动用于打连接，然后销毁这个参数。

另请参阅 `lastError()`。

**QString** `QSqlDatabase::password() const`

返回连接的密码。如果没有使用 `setPassword()` 函数进行密码的设置并且没有调用 `open()` 以及没有使用密码，那么就返回空的字符串。

另请参阅 `setPassword()`

```
int QSqlDatabase::port() const
```

返回连接的端口号。如果端口号没有设置，那么这个值就是未定义的。

另请参阅 `setPort()`

```
QSqlIndex QSqlDatabase::primaryIndex(const QString &tablename) const
```

返回所给表格名的索引。如果索引不存在，那么就返回一个空的 `QSqlIndex`

注意：如果表在创建时没有被引用，一些驱动程序（如 `QPSQL` 驱动程序）可能要求您以小写的形式传递表格名。

查看关于 `Qt SQL driver` 文档的更多信息。

另请参阅 `tables()` 和 `record()`。

```
QSqlRecord QSqlDatabase::record(const QString &tablename) const
```

返回一个 `QSqlRecord`，其中填充了名为 `tablename` 的表（或视图）中所有字段的名称。字段在记录中出现的顺序未定义。如果没有这样的表格（或者视图）存在，将会返回一个空的记录。

注意：如果表在创建时没有被引用，一些驱动程序（如 `QPSQL` 驱动程序）可能要求您以小写的形式传递表格名。

查看 `Qt SQL driver` 文档的更多信息。

```
[static] void QSqlDatabase::registerSqlDriver(const QString &name, QSqlDriver-  
CreatorBase *creator)
```

这个函数在 `SQL` 框架中注册一个名为 `name` 的新 `SQL` 驱动程序。这个是非常有用的，如果您有一个自定义的驱动，并且您并不想把它编译作为一个插件。

例如：

```
QSqlDatabase::registerSqlDriver("MYDRIVER", new QSqlDriverCreator<QSqlDriver>);  
QVERIFY(QSqlDatabase::drivers().contains("MYDRIVER"));  
QSqlDatabase db = QSqlDatabase::addDatabase("MYDRIVER");  
QVERIFY(db.isValid());
```

`QSqlDatabase` 拥有 `creator` 指针的所有权，因此您不能自己删除它。

另请参阅 `drivers()`。

```
[static] void QSqlDatabase::removeDatabase(const QString &connectionName)
```

从数据库列表中，删除一个叫 `connectionName` 数据库连接。

警告：不应在数据库连接上打开查询的情况下调用此函数，否则将发生资源泄漏。



例子：

```
// 错误
QSqlDatabase db = QSqlDatabase::database("sales");
QSqlQuery query("SELECT NAME, DOB FROM EMPLOYEES", db);
QSqlDatabase::removeDatabase("sales"); // 将会输出警告
// “db” 现在是一个悬而未决的无效数据库连接，
// “查询” 包含无效的结果集
```

正确的做法：

```
{
    QSqlDatabase db = QSqlDatabase::database("sales");
    QSqlQuery query("SELECT NAME, DOB FROM EMPLOYEES", db);
}
// “db” 和 “query” 都被销毁，因为它们超出了范围
QSqlDatabase::removeDatabase("sales"); // 正确的
```

如果要删除默认连接，这个连接可能是通过调用 `addDatabase()` 函数而创建的，但未指定连接名称，可以通过对 `database()` 返回的数据库调用 `connectionName()` 来检索默认连接名称。注意，如果没有创建默认数据库，将返回一个无效的数据库。

注意：这个函数是线程安全的

另请查阅 `database()`，`connectionName()`，和线程和 SQL 模块。

### `bool QSqlDatabase::rollback()`

在数据库里回滚一个事务，如果驱动支持一个事务以及一个 `transaction()` 已经被启动。如果操作成功返回 `true`。否则返回 `false`。

注意：对于某些数据库，如果存在使用数据库进行选择的活动查询，则回滚将失败并返回 `false`。确保在执行回滚操作之前，查询是非活动的状态。

调用 `lastError()` 操作获得错误的相关信息。

另请查阅 `QSqlQuery::isActive()`，`QSqlDriver::hasFeature()` 和 `commit()`。

### `void QSqlDatabase::setConnectOptions(const QString &options = QString())`

设置一组数据库的具体的可选项。它必须在打之这个连接之前执行这个操作，否则是无效的。另一个可能的原因是调用 `QSqlDatabase::setConnectOptions()` 去关闭这个连接，并且调用 `open()` 再次关闭这个连接。

选项字符串的格式是以分号分隔的选项名称，或选项 = 值对的列表。这个选项依赖于所使用的客户端：



ODBC	MySQL	PostgreSQL
SQL_ATTR_ACCESS_MODE	CLIENT_COMPRESS	connect_timeout
SQL_ATTR_LOGIN_TIMEOUT	CLIENT_FOUND_ROWS	options
SQL_ATTR_CONNECTION_TIMEOUT	CLIENT_IGNORE_SPACE	tty
SQL_ATTR_CURRENT_CATALOG	CLIENT_ODBC	requiresssl
SQL_ATTR_METADATA_ID	CLIENT_NO_SCHEMA	service
SQL_ATTR_PACKET_SIZE	CLIENT_INTERACTIVE	
SQL_ATTR_TRACEFILE	UNIX_SOCKET	
SQL_ATTR_TRACE	MYSQL_OPT_RECONNECT	
SQL_ATTR_CONNECTION_POOLING	MYSQL_OPT_CONNECT_TIMEOUT	
SQL_ATTR_ODBC_VERSION	MYSQL_OPT_READ_TIMEOUT	
	MYSQL_OPT_WRITE_TIMEOUT	
	SSL_KEY	
	SSL_CERT	
	SSL_CA	
	SSL_CAPATH	
	SSL_CIPHER	

DB2	OCI	TDS
SQL_ATTR_ACCESS_MODE	OCI_ATTR_PREFETCH_ROWS	无
SQL_ATTR_LOGIN_TIMEOUT	OCI_ATTR_PREFETCH_MEMORY	

SQLite	Interbase
SQLITE_BUSY_TIMEOUT	ISC_DPB_LC_CTYPE
SQLITE_OPEN_READONLY	ISC_DPB_SQL_ROLE_NAME
SQLITE_OPEN_URI	
SQLITE_ENABLE_SHARED_CACHE	
SQLITE_ENABLE_REGEX	

例子:

```
db.setConnectOptions("SSL_KEY=client-key.pem;SSL_CERT=client-cert.pem;SSL_CA=ca-cert.pem;CLIENT_IGNORE_SPACESHACES");
if (!db.open()) {
    db.setConnectOptions(); // 清除连接的字符串
    // ...
}
// ...
// PostgreSQL 连接
db.setConnectOptions("requiressl=1"); // 确保 PostgreSQL 安全套接字连接
if (!db.open()) {
    db.setConnectOptions(); // 清除可选
    // ...
}
// ...
// ODBC 连接
db.setConnectOptions("SQL_ATTR_ACCESS_MODE=SQL_MODE_READ_ONLY;SQL_ATTR_TRACE=SQL_OPT_TRACE_ON"); // 设置 ODBC 连接
if (!db.open()) {
    db.setConnectOptions(); // 不要尝试去设置这个选项。
    // ...
}
}
```

查阅这个客户端库文档，获得更多关于不同可选项的更多信息。

另请查阅 `connectOptions()`。

`void QSqlDatabase::setDatabaseName(const QString &name)`

通过所给的 `name` 参数来设置所连接的数据库名称。必须在打开连接之前设置数据库名称。或者，可以调用 `close()` 函数关闭连接，设置数据库名称，然后再次调用 `open()`。

注意：这个数据库名不是连接名。必须在创建连接对象时将连接名称传递给 `addDatabase()`。

对于 **QSQLITE** 驱动，如果数据库名指定的名字不存在，然后它将会创建这个文件，除非您设置了 **QSQLITE\_OPEN\_READONLY**。

此外，可以把 `name` 参数设置为 `:memory:`，可以创建一个临时数据库，该数据库仅在应用程序的生命周期内可用。

对于 **QOCI (Oracle)** 驱动，这个数据库名是 **TNS Service Name**。

对于 **QODBC** 驱动程序，名称可以是 **DSN**，**DSN** 文件名（在这种情况下，文件扩展名必须为 **.DSN**）或者是一个连接字符串。

例如，**Microsoft Access** 可以使用下面的连接方式来直接打开 **.mdb** 文件，而不是在 **ODBC** 管理工具里创建一个 **DSN** 对象：

```
// ...
QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
db.setDatabaseName("DRIVER={Microsoft Access Driver (*.mdb, *.accdb)};
FIL={MS Access};DBQ=myaccessfile.mdb");
if (db.open()) {
    // 成功!
}
// ...
```

这个没有默认的值

另请查阅 `databaseName()`、`setUserName()`、`setPassword()`、`setHostName()`、`setPort()`、`setConnectOptions()` 和 `open()`。

```
void QSqlDatabase::setHostName(const QString &host)
```

通过 `host` 参数来设置连接的主机名。为了生效，必须在打开连接之前，设置主机名。或者，可以调用 `close()` 关闭连接，然后设置主机名，再次调用 `open()` 函数。

这个没有默认值。

另请查阅 `hostName()`、`setUserName()`、`setPassword()`、`setDatabaseName()`、`setPort()`、`setConnectOptions()` 和 `open()`。

```
void QSqlDatabase::setNumericalPrecisionPolicy(QSql::NumericalPrecisionPolicy
precisionPolicy)
```

设置在此数据库连接上创建的查询使用的默认数值精度策略。

注意：驱动程序不支持以低精度获取数值，将忽略精度策略。您可以使用 `QSqlDriver::hasFeature()` 来查找一个驱动是否支持这个功能。

注意：通过 `precisionPolicy` 来设置这个默认的精度策略，将不会响影任何当前的活动查询。

qt4.6 中引入了这个函数。

另请查阅 `QSql::NumericalPrecisionPolicy`、`numericalPrecisionPolicy()`、`QSqlQuery::setNumericalPrecisionPolicy` 和 `QSqlQuery::numericalPrecisionPolicy`。

```
void QSqlDatabase::setPassword(const QString &password)
```

通过 `password` 参数来设置连接的密码。为了生效，必须在打开连接之前来设置密码。或者，您可以调用 `close()` 关闭连接，然后设置密码，再次调用 `open()` 函数。

这个没有默认值。

警告：这个函数以明文的形式把密码存放到 `qt` 里。将密码作为参数来避免这个行为，然后使用 `open()` 进行调用。

另请查阅 `password()`、`setUserName()`、`setDatabaseName()`、`setHostName()`、`setPort()`、`setConnectOptions()` 和 `open()`。

```
void QSqlDatabase::setPort(int port)
```

通过 `port` 参数设置连接的端口号。为了生效，您必须在打开连接之前，进行端口号的设置。或者，您可以调用 `close()` 关闭连接，然后设置端口号，再次调用 `open()` 函数

这个没有默认的值。

另请查阅 `port()`, `setUserName()`, `setPassword()`,  
`setHostName()`, `setDatabaseName()`, `setConnectOptions()` 和 `open()`。

```
void QSqlDatabase::setUserName(const QString &name)
```

通过 `name` 参数来设置连接的用户名。为了生效，必须在打开连接之前设置用户名。或者，您可以调用 `close()` 函数来关闭连接，设置用户，然后再次调用 `open()`

这个没有默认值。

另请查阅 `userName()`, `setDatabaseName()`, `setPassword()`,  
`setHostName()`, `setPort()`, `setConnectOptions()` 和 `open()`。

```
QStringList QSqlDatabase::tables(QSql::TableType type = QSql::Tables) const
```

返回由 `parameter type` 参数指定的数据库的表格、系统表和视图的列表。

另请查阅 `primaryIndex()` 和 `record()`。

```
bool QSqlDatabase::transaction()
```

如果驱动程序支持事务，则在数据库上开始事务。如果操作成功的话，返回 `true`，否则返回 `false`。

另请查阅 `QSqlDriver::hasFeature()`, `commit()` 和 `rollback()`。

```
QString QSqlDatabase::userName() const
```

返回连接的用户名；它也许为空。

另请查阅 `setUserName()`。

# 第十三章 QWaitCondition

## QWaitCondition

QWaitCondition 提供一个用于同步线程的条件变量。[更多 ...](#)

属性	内容
头文件	#include<QWaitCondition>
qmake	QT += core

注意：此类中所有函数都是[线程安全](#)的。公共成员函数

返回类型	函数
int	appDpiX(int screen = -1)
	QWaitCondition()
	~QWaitCondition()
void	notify_all()
void	notify_one()
bool	wait(QMutex *lockedMutex, QDeadlineTimer deadline = QDeadlineTimer(QDeadlineTimer::infinity()))
bool	wait(QMutex *lockedMutex, unsigned long time)
bool	wait(QReadWriteLock *lockedReadWriteLock, QDeadlineTimer deadline = QDeadlineTimer::infinity())
bool	wait(QReadWriteLock *lockedReadWriteLock, unsigned long time)
void	wakeAll()
void	wakeOne()

### 详细描述

QWaitCondition 允许线程告诉其他线程某种条件已经满足。一个或多个线程可以被阻塞并等待 QWaitCondition 用 wakeOne() 或 wakeAll() 设置条件。使用 wakeOne() 唤醒一个随机选择的线程，或使用 wakeAll() 唤醒所有线程。

例，假设我们有三个任务，当用户按下一个键时，应该执行某些任务。每个任务可以分成一个线程，每个线程都有一个 run() 主体，如下所示：

```
    forever {
        mutex.lock();
        keyPressed.wait(&mutex);
        do_something();
        mutex.unlock();
    }
```

这里，`keyPressed` 变量是 `QWaitCondition` 类型的全局变量。第四个线程将读取按键，并在每次收到按键时唤醒其他三个线程，如下所示：

```
forever {
    getchar();
    keyPressed.wakeAll();
}
```

唤醒三个线程的顺序是未知的。另外，如果某些线程在按下键时仍在 `do_something()` 中，它们将不会被唤醒（因为它们没有等待条件变量），因此该按键不会执行任务。这个问题可以通过使用计数器和 `QMutex()` 来解决。例如，下面是工作线程的新代码：

```
forever {
    mutex.lock();
    keyPressed.wait(&mutex);
    ++count;
    mutex.unlock();

    do_something();

    mutex.lock();
    --count;
    mutex.unlock();
}
```

下面是第四个线程的代码：

```
forever {
    getchar();

    mutex.lock();
    // Sleep until there are no busy worker threads
    while (count > 0) {
        mutex.unlock();
        sleep(1);
        mutex.lock();
    }
    keyPressed.wakeAll();
    mutex.unlock();
}
```

互斥量是必需的，因为当两个线程同时更改同一变量的值时，结果是不可预测的。等待条件是一个强大的线程同步原语。`Wait Conditions` 示例演示了如何使用 `QWaitCondition` 作为 `QSemaphore()` 的替代品，来控制生产者消费者的共享循环缓冲区的访问。

另请参阅：`QMutex`、`QSemaphore`、`QThread()` 和 `Wait Conditions` 示例。

成员函数文档

`QWaitCondition::QWaitCondition()`

构造。

`QWaitCondition::QWaitCondition()`

析构。

`void QWaitCondition::notify_all()`

用于 STL 兼容。它相当于 `wakeAll()`。在 Qt 5.8 引入该函数。

`void QWaitCondition::notify_one()`

用于 STL 兼容。它相当于 `wakeOne()`。在 Qt 5.8 引入该函数。

`bool QWaitCondition::wait(QMutex *lockedMutex, QDeadlineTimer deadline = QDeadlineTimer(QDeadlineTimer::Forever))`

释放 `lockedMutex` 并等待条件。`lockedMutex` 最初必须由调用线程锁定。如果 `lockedMutex` 未处于锁定状态，则行为未定义。如果 `lockedMutex` 是递归互斥体，则此函数将立即返回。`lockedMutex` 将被解锁，调用线程将阻塞，直到满足以下任一条件：

另一个线程调用 `wakeOne()` 或 `wakeAll()` 发出信号。在这种情况下，此函数将返回 `true`。截止日期已到。如果 `deadline` 是默认值 `QDeadlineTimer::Forever`，则永远不会超时（必须用信号通知事件）。如果等待超时，此函数将返回 `false`。`lockedMutex` 将返回到相同的锁定状态。提供此函数是为了允许原子从锁定状态转换到等待状态。在 Qt 5.12 引入该函数。

另请参阅：`wakeOne()`、`wakeAll()`。

`bool QWaitCondition::wait(QMutex *lockedMutex, unsigned long time)` 重载。

`bool QWaitCondition::wait(QReadWriteLock **lockedReadWriteLock, QDeadlineTimer deadline = QDeadlineTimer(QDeadlineTimer::Forever))`

释放 `lockedReadWriteLock` 并等待条件。`lockedReadWriteLock` 最初必须由调用线程锁定。如果 `lockedReadWriteLock` 未处于锁定状态，则此函数将立即返回。`lockedReadWriteLock` 不能递归锁定，否则此函数将无法正确释放锁。`lockedReadWriteLock` 将被解锁，调用线程将阻塞，直到满足以下任一条件：

另一个线程调用 `wakeOne()` 或 `wakeAll()` 发出信号。在这种情况下，此函数将返回 `true`。截止日期已到。如果 `deadline` 是默认值 `QDeadlineTimer::Forever`，则永远不会超时（必须用信号通知事件）。如果等待超时，此函数将返回 `false`。`lockedReadWriteLock` 将返回到相同的锁定状态。提供此函数是为了允许原子从锁定状态转换到等待状态。在 Qt 5.12 引入该函数。

另请参阅：`wakeOne()`、`wakeAll()`。

`bool QWaitCondition::wait(QReadWriteLock *lockedReadWriteLock, unsigned long time)` 重载。

`void QWaitCondition::wakeAll()`

唤醒等待条件的所有线程。线程的唤醒顺序取决于操作系统的调度策略，无法控制或预测。

另请参阅：`wakeOne()`。

`void QWaitCondition::wakeOne()`

唤醒一个等待条件的线程。线程的唤醒顺序取决于操作系统的调度策略，无法控制或预测。如果要唤醒特定线程，解决方案通常是使用不同的等待条件，并让线程在不同的条件下等待。

另请参阅：`wakeAll()`。



## 第十四章 QWebEngineHistory

### QWebEngineHistory

表示 Web 引擎页面的历史记录。

属性	方法
头文件	<code>#include &lt;QX11Info&gt;</code>
qmake	<code>QT += webenginewidgets</code>

该类在 Qt 5.4 中引入。

公有成员函数

类型	函数名
void	back()
QWebEngineHistoryItem	backItem() const
QList	backItems(int maxItems) const
bool	canGoBack() const
bool	canGoForward() const
void	clear()
int	count() const
QWebEngineHistoryItem	currentItem() const
int	currentItemIndex() const
void	forward()
QWebEngineHistoryItem	forwardItem() const
QList	forwardItems(int maxItems) const
void	goToItem(const QWebEngineHistoryItem &item)
QWebEngineHistoryItem	itemAt(int i) const
QList	items() const

相关非成员函数

类型	方法
QDataStream &	operator«(QDataStream &stream, const QWebEngineHistory &history)
QDataStream &	operator»(QDataStream &stream, QWebEngineHistory &history)

详细描述

## 成员函数文档

每个 Web 引擎页面都包含访问过的页面的历史记录，可以由 `QWebEnginePage::history()` 得到。

历史记录使用当前项目的概念，通过使用 `back()` 和 `forward()` 函数来回导航，将访问的页面划分为可以访问的页面。当前项目可以通过调用 `currentItem()` 获得，并且历史记录中的任意项目都可以通过将其传递给 `goToItem()` 使其成为当前项目。

可以通过调用 `backItems()` 函数获得描述可以返回的页面的项目列表。类似的，可以使用 `forwardItems()` 函数获得描述当前页面之前页面的项目。项目的总列表是通过 `items()` 函数获得的。

与容器一样，可以使用函数来检查列表中的历史记录。可以使用 `itemAt()` 获得历史记录中的任意项目，通过 `count()` 给出项目的总数，并可以使用 `clear()` 函数清除历史记录。

可以使用 `>` 运算符将 `QWebEngineHistory` 的状态保存到 `QDataStream` 中，并使用 `<` 操作符进行加载。

另外参见 `QWebEngineHistoryItem` and `QWebEnginePage`。

## 成员函数文档

`void QWebEngineHistory::back()` 将当前项目设置为历史记录中的前一个项目，然后转到相应页面；也就是说，返回一个历史项目。

See also `forward()` and `goToItem()` .

`QWebEngineHistoryItem QWebEngineHistory::backItem() const`

返回历史记录中当前项目之前的项目。

`QList<QWebEngineHistoryItem> QWebEngineHistory::backItems(int maxItems) const`

返回向后历史记录列表中的项目列表。最多返回 `maxItems` 条目。

See also `forwardItems()`。

`bool QWebEngineHistory::canGoBack() const`

如果历史记录中当前项目之前有一个项目，则返回 `true`，否则返回 `false`。

See also `canGoForward()`。

`bool QWebEngineHistory::canGoForward() const`

如果我们有一个项目可以前进，则返回 `true`，否则返回 `false`。

See also `canGoBack()`。

`void QWebEngineHistory::clear()`

清除历史记录。

See also `count()` and `items()`。

```
int QWebEngineHistory::count() const
```

返回历史记录中的项目总数。

```
QWebEngineHistoryItem QWebEngineHistory::currentItem() const
```

返回历史记录中的当前项目。

```
int QWebEngineHistory::currentItemIndex() const
```

返回历史记录中当前项目的索引。

```
void QWebEngineHistory::forward()
```

将当前项目设置为历史记录中的下一个项目，然后转到相应页面；即，前进一个历史项目。

See also `back()` and `goToItem()`。

```
QWebEngineHistoryItem QWebEngineHistory::forwardItem() const
```

返回历史记录中当前项目之后的项目。

```
QList<QWebEngineHistoryItem> QWebEngineHistory::forwardItems(int maxItems) const
```

返回转发历史记录列表中的项目列表。最多返回 `maxItems` 条目。

See also `backItems()`。

```
void QWebEngineHistory::goToItem(const QWebEngineHistoryItem &item)
```

将当前项目设置为历史记录中的指定项目，然后转到页面。

See also `back()` and `forward()`。

```
QWebEngineHistoryItem QWebEngineHistory::itemAt(int i) const
```

返回历史记录中索引 `i` 处的项目。

```
QList<QWebEngineHistoryItem> QWebEngineHistory::items() const
```

返回历史记录中当前所有项目的列表。

See also `count()` and `clear()`。

相关非成员函数

```
QDataStream &operator<<(QDataStream &stream, const QWebEngineHistory &history)
```

将 Web 引擎历史记录历史记录保存到流中。

```
QDataStream &operator>>(QDataStream &stream, QWebEngineHistory &history)
```

将 Web 引擎历史记录从流加载到历史记录中。



## 第十五章 QWebEngineHistoryItem

`QWebEngineHistoryItem` 类表示 Web 引擎页面历史记录中的一项。

属性	方法
头文件	<code>#include &lt;QWebEngineHistoryItem&gt;</code>
qmake	<code>QT += webenginewidgets</code>
Since:	Qt 5.4

公有成员函数

类型	函数名
	<code>QWebEngineHistoryItem(const QWebEngineHistoryItem &amp;other)</code>
	<code>~QWebEngineHistoryItem()</code>
<code>QUrl</code>	<code>iconUrl() const</code>
<code>bool</code>	<code>isValid() const</code>
<code>QDateTime</code>	<code>lastVisited() const</code>
<code>QUrl</code>	<code>originalUrl() const</code>
<code>void</code>	<code>swap(QWebEngineHistoryItem &amp;other)</code>
<code>QString</code>	<code>title() const</code>
<code>QUrl</code>	<code>url() const</code>
<code>QWebEngineHistoryItem &amp;</code>	<code>operator=(const QWebEngineHistoryItem &amp;other)</code>

详细描述

`QWebEngineHistoryItem` 类表示 Web 引擎页面历史记录中的一项。

每个 Web 引擎历史记录项都代表一个网页历史记录堆栈中的一个条目，其中包含有关该页面，其位置以及上次访问时间的信息。

另请参见 `QWebEngineHistory` 和 `QWebEnginePage::history()`。

成员函数文档

`QWebEngineHistoryItem::QWebEngineHistoryItem(const QWebEngineHistoryItem &other)`

从其他构造一个历史项。新项目和其他项目将共享其数据，并且修改该项目或其他项目将修改两个实例。

`QWebEngineHistoryItem::~~QWebEngineHistoryItem()`

销毁历史记录项。

```
QUrl QWebEngineHistoryItem::iconUrl() const
```

返回与历史记录项关联的图标的 URL。

另见 `url()`, `originalUrl()`, 和 `title()`。

```
bool QWebEngineHistoryItem::isValid() const
```

返回这是否是有效的历史记录项。

```
QDateTime QWebEngineHistoryItem::lastVisited() const
```

返回上次访问与该项目关联的页面的日期和时间。

另见 `title()` 和 `url()`。

```
QUrl QWebEngineHistoryItem::originalUrl() const
```

返回与历史记录项关联的原始 URL。

另见 `url()`。

```
void QWebEngineHistoryItem::swap(QWebEngineHistoryItem &other)
```

将历史项与其他项交换。

```
QString QWebEngineHistoryItem::title() const
```

返回与历史记录项关联的页面的标题。

另见 `url()` 和 `lastVisited()`。

```
QUrl QWebEngineHistoryItem::url() const
```

 返回与历史记录项关联的 URL。

另见 `originalUrl()`, `title()` 和 `lastVisited()`。

```
QWebEngineHistoryItem &QWebEngineHistoryItem::operator=(const QWebEngine-  
HistoryItem &other)
```

为此分配另一个历史记录项。该项目和其他项目将共享其数据，并且修改该项目或其他项目将修改两个实例。

# 第十六章 QWebEngineView

## QWebEngineView

QWebEngineView 类提供了一个 widget，被使用去查看和编辑 web 元素。

属性	方法
头文件	#include <QWebEngineView>
qmake	QT += webenginewidgets
继承:	QWidget

## 特性

属性名	类型
hasSelection	const bool
title	const QString
icon	const QIcon
zoomFactor	qreal
selectedText	const QString

## 公共成员函数

类型	函数名
	QWebEngineView(QWidget *parent = Q_NULLPTR)
virtual	~QWebEngineView()
void	findText(const QString &subString, QWebEnginePage::FindFlags options = ..., const QWebEngineCallback &resultCallback = ...)
bool	hasSelection() const
QWebEngineHistory *	history() const
QIcon	icon() const
QUrl	iconUrl() const
void	load(const QUrl &url)
void	load(const QWebEngineHttpRequest &request) QWebEnginePage *page() const
QAction *	pageAction(QWebEnginePage::WebAction action) const
QString	selectedText() const
void	setContent(const QByteArray &data, const QString &mimeType = QString(), const QUrl &baseUrl = QUrl())
void	setHtml(const QString &html, const QUrl &baseUrl = QUrl())
void	setPage(QWebEnginePage *page)
void	setUrl(const QUrl &url)
void	setZoomFactor(qreal factor)
QWebEngineSettings *	settings() const
QString	title() const
void	triggerPageAction(QWebEnginePage::WebAction action, bool checked = false)
QUrl	url() const
qreal	zoomFactor() const

## 重实现公共成员函数

virtual QSize sizeHint() const override

- 216 个公共成员函数继承自 QWidget

- 31 个公共成员函数继承自 QObject
- 14 个公共成员函数继承自 QPaintDevice

## 公有槽函数

类型	函数名
void	back()
void	forward()
void	reload()
void	stop()

- 19 个公共槽函数继承自 QWidget
- 1 个公共槽函数继承自 QObject

## 信号

类型	函数名
void	iconChanged(const QIcon &icon)
void	iconUrlChanged(const QUrl &url)
void	iconUrlChanged(const QUrl &url)
void	loadProgress(int progress)
void	loadStarted()
void	renderProcessTerminated(QWebEnginePage::RenderProcessTerminationStatus terminationStatus, int exitCode)
void	selectionChanged()
void	titleChanged(const QString &title)
void	urlChanged(const QUrl &url)

- 3 信号继承自 QWidget
- 2 信号继承自 QObject

## 静态公有成员函数

- 5 个静态公有成员函数继承自 QWidget
- 9 个静态公有成员函数 QObject

## 保护成员函数

```
virtual QWebEngineView * createWindow(QWebEnginePage::WebWindowType type)
```

## 重实现保护成员函数

类型	函数名
virtual void	contextMenuEvent(QContextMenuEvent *event) override
virtual void	dragEnterEvent(QDragEnterEvent *e) override
virtual void	dragLeaveEvent(QDragLeaveEvent *e) override
virtual void	dragMoveEvent(QDragMoveEvent *e) override
virtual void	dropEvent(QDropEvent *e) override
virtual bool	event(QEvent *ev) override
virtual void	hideEvent(QHideEvent *event) override
virtual void	showEvent(QShowEvent *event) override



- 35 个保护成员函数继承自 `QWidget`
- 9 个保护成员函数继承自 `QObject`
- 1 个保护成员函数继承自 `QPaintDevice`

额外继承成员

- 一个保护槽继承自 `QWidget`

详细描述

`QWebEngineView` 类提供了一个小部件，被用去查看和编辑 `web` 文档。

一个 `web` 视图是组成 `Qt WebEngine web` 浏览模块的主要 `widget` 之一。它可以被用于多种应用去展示来自 `Internet` 的 `web` 内容。

一个 `web` 元素可以通过 `load()` 函数加载到 `web` 视图。一般总是使用 `Get` 方法去加载 `URLs`。

像所有 `Qt` 小部件一样，必须调用 `show()` 函数才能显示 `Web` 视图。下面的代码段说明了这一点：

```
QWebEngineView *view = new QWebEngineView(parent);
view->load(QUrl("http://qt-project.org/"));
view->show();
```

或者，`setUrl()` 可以被使用去加载 `web` 元素。如果你有可读的 `HTML` 内容，你可以使用 `setHtml()` 代替。

当视图开始加载的时候，发出 `loadStarted()` 信号，当 `web` 视图的一个元素被完整加载的时候，发出 `loadProgress()` 信号，例如一个 `embedded` 图像或一个脚本。当视图被完整加载时，发出 `loadFinished()` 信号。其参数 `true` 或 `false` 指示加载是成功还是失败。

`page()` 函数返回一个指向 `web` 页对象的指针。一个 `QWebEngineView` 包含一个 `QWebEnginePage`，允许依次访问 `QWebEnginePage` 在页的内容里。`HTML` 文档的标题可以用 `title()` 属性访问。另外，一个 `web` 元素可能是一个图标，可以使用 `icon()` 访问，使用 `iconUrl()` 访问图标的 `URL`。如果标题或是图标改变，将发出 `titleChanged()`，`iconChanged()` 和 `iconUrlChanged()` 信号来回应。`zoomFactor()` 属性通过缩放因子，能够缩放 `web` 页的内容。

该小部件适合于上下文菜单并包含在浏览器中有用的操作。对于自定义上下文菜单，或将动作嵌入菜单或工具栏中，可通过 `pageAction()` 获得单个动作。`web` 视图维护返回动作的状态，但允许修改动作属性，例如 `text` or `icon`。动作语义也可以通过 `triggerPageAction()` 直接触发。

对于 `web` 元素，如果你想要提供支持允许用户去打开新的窗口，例如 `pop-up windows`，你可以子类化 `QWebEngineView` 和实现 `createWindow()` 函数。

您也可以在 `WebEngine Widgets Simple Browser Example`，`WebEngine Content Manipulation Example`，和 `WebEngine Markdown Editor Example` 的文档中找到相关的信息。

特性文档编制

属性名	类型
<code>hasSelection</code>	<code>const bool</code>

此属性保存此页面是否包含所选内容。默认情况下，此属性为 `false`。

访问函数：

<code>bool</code>	<code>hasSelection() const</code>
-------------------	-----------------------------------

另见 `selectionChanged()`

`icon : const QIcon`

# 第十七章 QWriteLocker

QWriteLocker 是工具类，它简化了对读写锁，写访问的的锁定和解锁。[更多...](#)

属性	方法
头文件	#include<QWriteLocker>
qmake	QT += core

注意：此类中所有函数都是线程安全的。

公共成员函数

返回类型	函数
	QWriteLocker(QReadWriteLock *lock)
	~QWriteLocker()
	const QString
QReadWriteLock *	readWriteLock() const
void	relock()
void	unlock()

详细描述

QWriteLocker (和 QReadLocker) 的目的是简化 QReadWriteLock 的锁定和解锁。锁定和解锁语句、异常处理代码是很容易出错的，而且很难调试。QWriteLocker 可以确保在此类情况下，锁的状态始终定义良好。

下面是一个使用 QWriteLocker 锁定和解锁读写锁的示例：

```
QReadWriteLock lock;

void writeData(const QByteArray &data)
{
    QWriteLocker locker(&lock);
    ...
}
```

等价于以下代码：

```
QReadWriteLock lock;

void writeData(const QByteArray &data)
{
```

```
    lock.lockForWrite();  
    ...  
    lock.unlock();  
}
```

QMutexLocker 文档展示了使用 locker 对象来大大简化编程的示例。

另请参阅: QReadLocker、QReadWriteLock。

成员函数文档

**QWriteLocker::QWriteLocker(QReadWriteLock \*lock)**

构造一个 QWriteLocker 并锁定用于写入的锁。当 QWriteLocker 被销毁时，锁将被解锁。如果 lock = nullptr，则 QWriteLocker 不执行任何操作。

另请参阅: QReadWriteLock::lockForWrite()。

**QWriteLocker::~~QWriteLocker()**

销毁 QWriteLocker 并解锁传递给构造函数的锁。

另请参阅: QReadWriteLock::unlock()。

**QReadWriteLock \*QWriteLocker::readWriteLock() const**

返回传递给构造函数的读写锁的指针。

**void QWriteLocker::relock()**

重新锁定。

另请参阅: unlock()。

**void QWriteLocker::unlock()** 解锁。

另请参阅: QReadWriteLock::unlock()。

# 第十八章 QX11Info

## QX11Info

提供有关 X11 相关的相关配置信息（就是 linux 下的 x11 相关的配置信息

属性	方法
头文件	<code>#include &lt;QX11Info&gt;</code>
qmake	<code>QT += x11extras</code>
Since:	Qt5.1

简述

类型	函数名
int	appDpiX(int screen = -1)
int	appDpiY(int screen = -1)
unsigned long	appRootWindow(int screen = -1)
int	appScreen()
unsigned long	appTime()
unsigned long	appUserTime()
xcb_connection_t *	connection()
Display *	display()
unsigned long	getTimestamp()
bool	isCompositingManagerRunning(int screen = -1)
bool	isPlatformX11()
QByteArray	nextStartupId()
void	setAppTime(unsigned long time)
void	setAppUserTime(unsigned long time)
void	setNextStartupId(const QByteArray &id)

详细说明该类提供了关于 x window 相关的显式配置信息

该类提供了两类 API：一种是提供特定的 widget 或者特定的 pixmap 相关的非静态函数，一种是为应用程序提供默认信息的静态函数。（这个分类简直了!!!）

成员函数

int QX11Info::appDpiX(int screen = -1) static 函数

返回指定屏幕的水平分辨率。

参数 `screen` 是指哪个 `x` 屏幕（比如两个的话，第一个就是 `0`，第二个就是 `1`）。请注意，如果用户使用的系统是指 `Xinerama`（而不是传统的 `x11` 多屏幕），则只有一个 `x` 屏幕。请使用 `QDesktopWidget` 来查询有关于 `Xinerama` 屏幕的信息。

另参阅 `appDipY()`;

`int QX11Info::appDpiY(int screen = -1) static 函数`

返回指定屏幕的垂直分辨率。

参数 `screen` 是指哪个 `x` 屏幕（比如两个的话，第一个就是 `0`，第二个就是 `1`）。请注意，如果用户使用的系统是指 `Xinerama`（而不是传统的 `x11` 多屏幕），则只有一个 `x` 屏幕。请使用 `QDesktopWidget` 来查询有关于 `Xinerama` 屏幕的信息。

另参阅 `appDipX()`;

`unsigned long QX11Info::appRootWindow(int screen = -1) static 函数`

返回指定屏幕应用程序窗口的句柄

参数 `screen` 是指哪个 `x` 屏幕（比如两个的话，第一个就是 `0`，第二个就是 `1`）。请注意，如果用户使用的系统是指 `Xinerama`（而不是传统的 `x11` 多屏幕），则只有一个 `x` 屏幕。请使用 `QDesktopWidget` 来查询有关于 `Xinerama` 屏幕的信息。

`int QX11Info::appScreen() static 函数`

返回应用程序正在显示的屏幕编号。此方法是指每个原始的 `X11` 屏幕使用不同的 `DISPLAY` 环境变量。只有当您的应用程序需要知道它在哪个 `X` 屏幕上运行时，这个信息才有用。在典型的多个物理机连接到一个 `X11` 屏幕中时。意味着这个方法对于每台物理机来讲都是相同的编号。在这样的设置中，如果您对 `X11` 的 `RandR` 拓展程序感兴趣，可以通过 `QDesktopWidget` 和 `QScreen` 获得。

`unsigned long QX11Info::appTime() static 函数`

返回 `X11` 的时间

`unsigned long QX11Info::appUserTime() static 函数`

返回 `X11` 的用户时间

`xcb_connection_t *QX11Info::connection() static 函数`

返回应用程序默认的 `XCB` 信息。

`Display *QX11Info::display() static 函数`

返回应用程序默认的显式屏幕

`unsigned long QX11Info::getTimestamp() static 函数`

从 `X` 服务器上获取当前 `X11` 的时间戳。此方法创建一个事件来阻塞住 `X11` 服务器，直到它从 `X` 服务器接受回来。这个函数是从 `Qt5.2` 中引入的。

`bool QX11Info::isCompositingManagerRunning(int screen = -1)` static 函数

如果屏幕的合成管理器在运行时，则返回 `true` (ps, 合成管理器运行会有一些特殊的效果，比如一些透明色的绘制，可以用这个函数判断下。)，否则则返回 `false`。这个函数是从 Qt5.7 中引入的。

`bool QX11Info::isPlatformX11()` static 函数

如果应用程序运行在 X11 上则返回 `true`。这个函数是从 Qt5.2 开始引入的。

`QByteArray QX11Info::nextStartupId()`

返回此进程显式的下一个窗口的启动 ID。显式下一个窗口后，下一个启动 ID 则为空。

(Qt 官网很少给这种链接啊)<http://standards.freedesktop.org/startup-notification-spec/startup-notification-latest.txt>

这个函数在 Qt5.4 引入。

`void QX11Info::setAppTime(unsigned long time)` static 函数

将 X11 时间设置成指定的值。

`void QX11Info::setAppUserTime(unsigned long time)` static 函数

设置 X11 用户的时间

`void QX11Info::setNextStartupId(const QByteArray &id)` static 函数

设置下一个启动程序的 ID。第一个窗口的启动 ID 来自环境变量 `DESKTOP_STARTUP_ID`。当请求来自另一个进程（比如通过 `QDus`）时，此方法对于后续窗口很有用。

这个函数是从 Qt5.4 中引用的。