

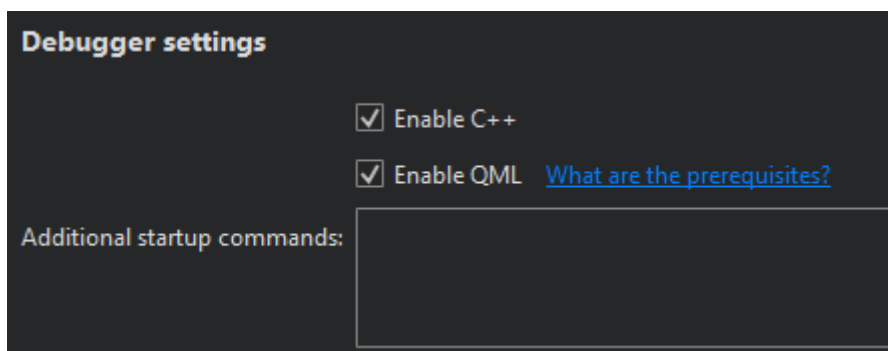
# 调试Qt快速项目

有关如何调试 Qt 快速项目的示例，请参见[调试 Qt 快速示例应用程序](#)。

**注意：**在本节中，您将使用高级菜单项。默认情况下，这些是不可见的。要切换高级菜单项的可见性，请参阅[自定义菜单](#)。

## 设置 QML 调试

若要调试 Qt 快速 UI 项目，请在“**项目模式的调试器设置**”中选中“**启用 QML**”复选框 **运行设置**。



## 启动 QML 调试

若要启动应用程序，请选择“**调试**”>“**启动调试**”>“**启动项目的启动调试**”或按 F5。应用程序开始运行后，其行为和执行将照常进行。然后，您可以执行以下任务：

- › 调试脚本函数
- › 执行脚本表达式以获取有关应用程序状态的信息
- › 检查 QML 属性和脚本变量，并在运行时临时更改它们

调试已在运行的应用程序：

1. 使用以下参数启动应用程序：

```
-qmljsdebugger=port:<port>[,host:<ip address>][,block]
```

其中（强制）指定调试端口，（可选）指定运行应用程序的主机的 IP 地址，并且（可选）阻止应用程序运行，直到调试客户端连接到服务器。这样就可以从头开始进行调试。portip addressblock

2. 选择“调试”>“启动调试”>“附加到 QML 端口”。

选择为运行要调试的应用程序的设备配置的工具包。应用程序启动时，要使用的端口号显示在标准输出中。

## 调试脚本函数

可以使用 Qt 设计工作室调试模式在调试时检查应用程序的状态。您可以通过以下方式与调试器进行交互：

- › 设置断点
- › 查看调用堆栈跟踪
- › 查看局部变量和函数参数
- › 计算表达式


## 设置断点


您可以将断点与以下各项相关联：

- › 源代码文件和行
- › 功能
- › 地址
- › 引发和捕获异常
- › 执行和分叉过程
- › 执行某些系统调用
- › 程序运行时特定地址的内存块的变化
- › 发射量子线信号
- › 引发脚本异常

The interruption of a program by a breakpoint can be restricted with certain conditions.

Breakpoints come in two varieties: `and` `.` An unclaimed breakpoint represents a task to interrupt the debugged program and passes the control to you later. It has two states: `and` `.unclaimedclaimedpendingimplanted`

Unclaimed breakpoints are stored as a part of a session and exist independently of whether a program is being debugged or not. They are listed in the **Breakpoint Preset** view and in the editor using the  (**Unclaimed Breakpoint**) icon, when they refer to a position in code.

Breakpoint Preset							
Debuggee	Function	File	Line	Address	Condition	Ignore	Threads
	-	...\\quickcontrols2\\gallery\\gallery.qml	155				(all)

When a debugger starts, the debugging backend identifies breakpoints from the set of unclaimed breakpoints that might be handled by the debugged program and claims them for its own exclusive use. Claimed breakpoints are listed in the **Breakpoints** view of the running debugger. This view only exists while the debugger is running.

appear as a pending breakpoint in the **Breakpoint** view.

At various times, attempts are made to implant pending breakpoints into the debugged process. Successful implantation might create one or more implanted breakpoints, each associated with an actual address in the debugged breakpoint. The implantation might also move a breakpoint marker in the editor from an empty line to the next line for which the actual code was generated, for example. Implanted breakpoint icons don't have the hourglass overlay.

When the debugger ends, its claimed breakpoints, both pending and implanted, will return to the unclaimed state and re-appear in the **Breakpoint Preset** view.

When an implanted breakpoint is hit during the execution of the debugged program, control is passed back to you. You can then examine the state of the interrupted program, or continue execution either line-by-line or continuously.

Number	Function	File	Line	Address	Condition	Ignore	Threads
2	-	f:\dd\vc\tools\crt\vcstartup\src\startup\exe_main.cpp	17	0x7ff7447b9e3d			(all)

## Adding Breakpoints

To add breakpoints:

1. Add a new breakpoint in one of the following ways:
  - › In the code editor, click the left margin or press **F9** (**F8** on macOS) on a particular line you want the program to stop.
  - › In the **Breakpoint Preset** view or the **Breakpoints** view:
    - › Double-click the empty part of the view.
    - › Right-click the view, and select **Add Breakpoint** in the context menu.
2. In the **Breakpoint type** field, select the location in the program code where you want the program to stop. The other options to specify depend on the selected location.

QC Add Breakpoint

Basic

Breakpoint type: File Name and Line Number

File name: 

Browse...

Line number: 0

Enabled: ☒

Address:

Expression:

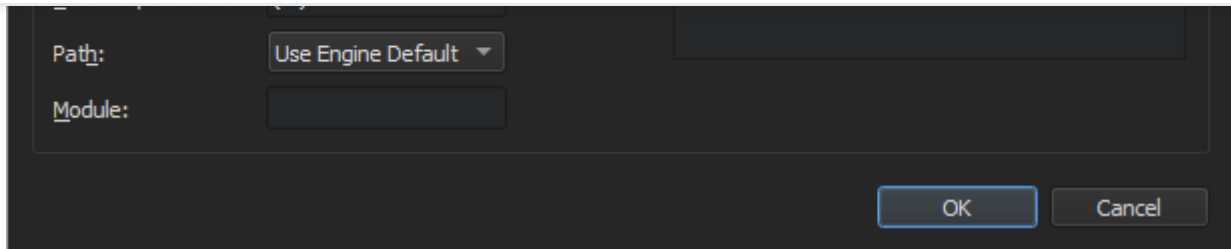
Function:

One shot only: ☐

Advanced

Condition:

Commands:



3. In the **Condition** field, set the condition to be evaluated before stopping at the breakpoint if the condition evaluates as true.
4. In the **Ignore** field, specify the number of times that the breakpoint is ignored before the program stops.
5. In the **Commands** field, specify the commands to execute when the program stops; one command on a line. GDB executes the commands in the order in which they are specified.

## Moving Breakpoints

To move a breakpoint:

- › Drag and drop a breakpoint marker to another line in the text editor.
- › In the **Breakpoint Preset** view or the **Breakpoints** view, select **Edit Selected Breakpoints**, and set the line number in the **Line number** field.

## Deleting Breakpoints

To delete breakpoints:

- › Click the breakpoint marker in the text editor.
- › In the **Breakpoint Preset** view or the **Breakpoints** view:
  - › Select the breakpoint and press **Delete**.
  - › Select **Delete Selected Breakpoints**, **Delete Selected Breakpoints**, or **Delete Breakpoints of File** in the context menu.

## Enabling and Disabling Breakpoints

To temporarily disable a breakpoint without deleting it and losing associated data like conditions and commands:

- › Right-click the breakpoint marker in the text editor and select **Disable Breakpoint**.
- › Select a line that contains a breakpoint and press **Ctrl+F9** (**Ctrl+F8** on macOS).
- › In the **Breakpoint Preset** view or the **Breakpoints** view:
  - › Select the breakpoint and press **Space**.
  - › Select **Disable Breakpoint** in the context menu.

A hollow breakpoint icon in the text editor and the views indicates a disabled breakpoint. To re-enable a breakpoint, use any of the above methods.

With the notable exception of data breakpoints, breakpoints retain their enabled or disabled state when the debugged program is restarted.

## Setting Data Breakpoints

△ *data breakpoint* stops the program when data is read or written at the specified address

1. In the **Breakpoint Preset** or **Breakpoints** view, select **Add Breakpoint** in the context menu.
2. In the **Breakpoint type** field, select **Break on data access at fixed address**.
3. In the **Address** field, specify the address of the memory block.
4. Select **OK**.

If the address is displayed in the **Locals** or **Expressions** view, you can select **Add Data Breakpoint at Object's Address** in the context menu to set the data breakpoint.

Data breakpoints will be disabled when the debugged program exits, as it is unlikely that the used addresses will stay the same at the next program launch. If you really want a data breakpoint to be active again, re-enable it manually.

## Viewing Call Stack Trace

When the program being debugged is interrupted, Qt Design Studio displays the nested function calls leading to the current position as a call stack trace. This stack trace is built up from call stack frames, each representing a particular function. For each function, Qt Design Studio tries to retrieve the file name and line number of the corresponding source file. This data is shown in the **Stack** view.

Stack <span>📄</span> <span>✕</span>			
Level	Function	File	Line
➔ 1	TextFinder::loadTextFile	textfinder.cpp	85
2	TextFinder::TextFinder	textfinder.cpp	64
3	main	main.cpp	59
4	invoke_main	exe_common.inl	79
5	__scrt_common_main_seh	exe_common.inl	283
6	__scrt_common_main	exe_common.inl	326
7	mainCRTStartup	exe_main.cpp	17
8	BaseThreadInitThunk	KERNEL32	
9	RtlUserThreadStart	ntdll	

Since the call stack leading to the current position may originate or go through code for which no debug information is available, not all stack frames have corresponding source locations. Stack frames without corresponding source locations are grayed out in the **Stack** view.

If you click a frame with a known source location, the text editor jumps to the corresponding location and updates the **Locals** and **Expressions** views, making it seem like the program was interrupted before entering the function.

To find out which QML file is causing a Qt Quick 2 application to crash, select **Load QML Stack** in the context menu

## Local Variables and Function Parameters

The Locals view consists of the **Locals** pane and the **Return Value** pane (hidden when empty).

Name	Value	Type
a	@0xc8b7fdc0	QApplication
argc	<not accessible>	int
argv	<0 items>	char **
w	@0xc8b7fd88	TextFinder
[QWidget]	@0xc8b7fd88	QWidget
staticMetaObject	@0x7ff73a7e3008	QMetaObject
[strings]	<at least 1 items>	
[raw]		
[properties]	<at least 0 items>	
[methods]	<1 items>	
[superdata]	@0x7ffec961bd48	*QMetaObject
[members]		
d	@0x7ff73a7e3008	QMetaObject::<unnamed-type-d>
data	8	unsigned int
extradata	0x0	void *
relatedMetaObjects	0x0	QMetaObject **
static_metacall	140699815024400	<function>
stringdata	"TextFinder"	QArrayData
superdata	@0x7ffec961bd48	QMetaObject
[strings]	<at least 34 items>	
[0]	"QWidget"	
[1]	"windowTitleChanged"	
[2]	""	
[3]	"title"	
[4]	"windowIconChanged"	
[5]	"icon"	
[6]	"windowIconTextChanged"	
[7]	"iconText"	
[8]	"customContextMenuRequested"	
[9]	"pos"	
[10]	"setEnabled"	
[11]	"setDisabled"	

Whenever a program stops under the control of the debugger, it retrieves information about the topmost stack frame and displays it in the **Locals** view. The **Locals** pane shows information about parameters of the function in that frame as well as the local variables. If the last operation in the debugger was returning from a function after pressing **Shift+F11**, the **Return Value** pane displays the value returned by the function.

When using GDB, you can specify whether the dynamic or the static type of objects will be displayed. Select **Use dynamic object type for display** in the context menu. Keep in mind that choosing the dynamic type might be slower.

## Evaluating Expressions

To compute values of arithmetic expressions or function calls, use expression evaluators in the **Expressions** view. To insert a new expression evaluator, either double-click on an empty part of the **Expressions** or **Locals** view, or select **Add New Expression Evaluator** from the context menu, or drag and drop an expression from the code editor.

Name	Value	Type
1+2	3	int
line	"These forms are processed at run-time to produce dynamically-generated user ..."	QString
[0]	'T'	QChar
[1]	'h'	QChar
[2]	'e'	QChar
[3]	's'	QChar
[4]	'e'	QChar
[5]	' '	QChar
[6]	'f'	QChar
[7]	'o'	QChar
[8]	'r'	QChar
[9]	'm'	QChar
[10]	's'	QChar
[11]	' '	QChar
[12]	'a'	QChar
[13]	'r'	QChar
[14]	'e'	QChar
[15]	' '	QChar
[16]	'p'	QChar
[17]	'r'	QChar
[18]	'o'	QChar
[19]	' '	QChar

**Note:** Expression evaluators are powerful, but slow down debugger operation significantly. It is advisable to not use them excessively, and to remove unneeded expression evaluators as soon as possible.

Expression evaluators are re-evaluated whenever the current frame changes. Note that functions used in the expressions are called each time, even if they have side-effects.

The QML debugger can evaluate JavaScript expressions.

## Inspecting Items

While the application is running, you can use the **Locals** view to explore the QML item structure.

Locals		
Name	Value	Type
QQuickView	object	QQuickView
Properties	list	
QQuickRootItem	object	QQuickRootItem
Properties	list	
root	object	Rectangle
Properties	list	
Transition	object	Transition
State	object	State
Properties	list	
changes	<unknown valu...	QQmlListProperty<QQuickStateOperation>
extend		QString
name	in-game	QString
objectName		QString
when		QQmlBinding *
gameOverTimer	object	Timer
Image	object	Image
gameCanvas	object	GameArea
menu	object	Item
scoreBar	object	Image
bottomBar	object	Image
Connections	object	Connections
stateChangeAnim	object	SequentialAnimation
Keys	object	Keys
QQmlEngine	object	QQmlEngine
QQmlFileSelector	object	QQmlFileSelector
Component	object	Component

To keep the application visible while you interact with the debugger, select **Debug > Show Application on Top**.

You can view a QML item in the **Locals** view in the following ways:

- Expand the item in the object tree.
- Select the item in the code editor.
- Select **Debug > Select** to activate selection mode and then click an item in the running application.

To change property values temporarily, without editing the source, double-click them and enter the new values. You can view the results in the running application.

## Inspecting User Interfaces

When you debug complex applications, you can jump to the position in code where an item is defined.

You can also view the item hierarchy in the running application:

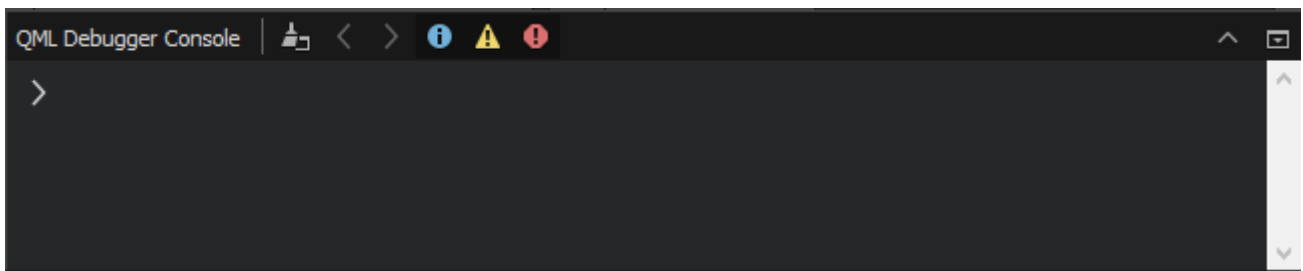
Double-click an item in the running application to cycle through the item stack at the cursor position.

To switch out of the selection mode, toggle the **Select** menu item.

To move the application running in Qt QML Viewer to the front, select **Debug > Show Application on Top**.

## Executing JavaScript Expressions

When the application is interrupted by a breakpoint, you can use the **QML Debugger Console** to execute JavaScript expressions in the current context. To open it, choose **View > Output > QML Debugger Console**.



You can change property values temporarily, without editing the source, and view the results in the running application. You can change the property values permanently in the [Properties](#) view.

## Applying QML Changes at Runtime

When you change property values in the **QML Debugger Console** or in the **Locals** or **Expression** view, they are immediately updated in the running application, but not in the source code.

[< Debugging and Profiling](#)

[Debugging a Qt Quick Example Application >](#)



Contact Us

### Company

[About Us](#)

[Investors](#)

[Newsroom](#)

### Licensing

[Terms & Conditions](#)

[Open Source](#)

[FAQ](#)





Support

- Support Services
- Professional Services
- Partners
- Training

For Customers

- Support Center
- Downloads
- Qt Login
- Contact Us
- Customer Success

Community

- Contribute to Qt
- Forum
- Wiki
- Downloads
- Marketplace