QTimer

The `QTimer` class provides repetitive and single-shot timers. More…

Synopsis

Functions

def `interval` ()
def `isActive` ()
def `isSingleShot` ()
def `remainingTime` ()
def `setInterval` (msec)
def `setSingleShot` (singleShot)
def `setTimerType` (atype)
def `timerId` ()
def `timerType` ()

Slots

def `start` ()
def `start` (msec)
def `stop` ()

Static functions

def `singleShot` (arg__1, arg__2)
def `singleShot` (msec, receiver, member)
def `singleShot` (msec, timerType, receiver, member)

Detailed Description

The `QTimer` class provides a high-level programming interface for timers. To use it, create a `QTimer` , connect its `timeout()` signal to the appropriate slots, and call `start()` . From then on, it will emit the `timeout()` signal at constant intervals.

Example for a one second (1000 millisecond) timer (from the Analog Clock example):

```
QTimer *timer = new QTimer(this);
```

```
connect(timer, &QTimer::timeout, this, QOverload<>::of(&An
timer->start(1000);
```

From then on, the `update()` slot is called every second.

You can set a timer to time out only once by calling `setSingleShot` (true). You can also use the static `singleShot()` function to call a slot after a specified interval:

```
QTimer.singleShot(200, self.updateCaption)
```

In multithreaded applications, you can use `QTimer` in any thread that has an event loop. To start an event loop from a non-GUI thread, use `exec()` . Qt uses the timer's `thread affinity` to determine which thread will emit the `timeout()` signal. Because of this, you must start and stop the timer in its thread; it is not possible to start a timer from another thread.

As a special case, a `QTimer` with a timeout of 0 will time out as soon as possible, though the ordering between zero timers and other sources of events is unspecified. Zero timers can be used to do some work while still providing a snappy user interface:

```
timer = QTimer(self)
timer.timeout.connect(self.processOneThing)
timer.start()
```

From then on, `processOneThing()` will be called repeatedly. It should be written in such a way that it always returns quickly (typically after processing one data item) so that Qt can deliver events to the user interface and stop the timer as soon as it has done all its work. This is the traditional way of implementing heavy work in GUI applications, but as multithreading is nowadays becoming available on more and more platforms, we expect that zero-millisecond `QTimer` objects will gradually be replaced by `QThread` s.

Accuracy and Timer Resolution

The accuracy of timers depends on the underlying operating system and hardware. Most platforms support a resolution of 1 millisecond, though the accuracy of the timer will not equal this resolution in many real-world situations.

The accuracy also depends on the `timer type` . For `PreciseTimer` , `QTimer` will try to keep the accuracy at 1 millisecond. Precise timers will also never time out earlier than expected.

For `CoarseTimer` and `VeryCoarseTimer` types, `QTimer` may wake up earlier than expected, within the margins for those types: 5%

of the interval for `CoarseTimer` and 500 ms for `VeryCoarseTimer` .

All timer types may time out later than expected if the system is busy or unable to provide the requested accuracy. In such a case of timeout overrun, Qt will emit `timeout()` only once, even if multiple timeouts have expired, and then will resume the original interval.

Alternatives to QTimer

An alternative to using `QTimer` is to call `startTimer()` for your object and reimplement the `timerEvent()` event handler in your class (which must inherit `QObject`). The disadvantage is that `timerEvent()` does not support such high-level features as single-shot timers or signals.

Another alternative is `QBasicTimer` . It is typically less cumbersome than using `startTimer()` directly. See Timers for an overview of all three approaches.

Some operating systems limit the number of timers that may be used; Qt tries to work around these limitations.

See also

QBasicTimer, QTimerEvent, timerEvent(), Timers, Analog Clock Example, Wiggly Example

class PySide2.QtCore.QTimer([*parent=None*])

```
param parent:
   PySide2.QtCore.QObject
```

Constructs a timer with the given `parent` .

PySide2.QtCore.QTimer.interval()

```
Return type:
   int
```

This property holds the timeout interval in milliseconds.

The default value for this property is 0. A `QTimer` with a timeout interval of 0 will time out as soon as all the events in the window system's event queue have been processed.

Setting the interval of an active timer changes its `timerId()` .

See also

singleShot

PySide2.QtCore.QTimer.isActive()

```
Return type:
   bool
```

This boolean property is `true` if the timer is running; otherwise false.

PySide2.QtCore.QTimer.isSingleShot()

```
Return type:
   bool
```

This property holds whether the timer is a single-shot timer.

A single-shot timer fires only once, non-single-shot timers fire every `interval` milliseconds.

The default value for this property is `false` .

See also

interval, singleShot()

PySide2.QtCore.QTimer.remainingTime()

```
Return type:
   int
```

This property holds the remaining time in milliseconds.

Returns the timer's remaining value in milliseconds left until the timeout. If the timer is inactive, the returned value will be -1. If the timer is overdue, the returned value will be 0.

See also

interval

PySide2.QtCore.QTimer.setInterval(*msec*)

```
Parameters:
   msec - int
```

This property holds the timeout interval in milliseconds.

The default value for this property is 0. A `QTimer` with a timeout interval of 0 will time out as soon as all the events in the window system's event queue have been processed.

Setting the interval of an active timer changes its `timerId()` .

See also

singleShot

Table of Contents

PySide2.QtCore.QTimer.setSingleShot(*singleShot*)

```
Parameters:
   singleShot - bool
```

This property holds whether the timer is a single-shot timer.

A single-shot timer fires only once, non-single-shot timers fire every `interval` milliseconds.

The default value for this property is `false` .

See also

interval, singleShot()

PySide2.QtCore.QTimer.setTimerType(*atype*)

```
Parameters:
   atype - TimerType
```

This property holds controls the accuracy of the timer.

The default value for this property is `Qt::CoarseTimer` .

See also

TimerType

static PySide2.QtCore.QTimer.singleShot(*arg__1*, *arg__2*)

```
Parameters:

      arg__1 - int
      arg__2 - PyCallable
```

static PySide2.QtCore.QTimer.singleShot(*msec*, *timerType*, *receiver*, *member*)

```
Parameters:

      msec - int
      timerType - TimerType
      receiver - PySide2.QtCore.QObject
      member - str
```

This is an overloaded function.

This static function calls a slot after a given time interval.

It is very convenient to use this function because you do not need to bother with a `timerEvent` or create a local `QTimer` object.

The `receiver` is the receiving object and the `member` is the slot. The time interval is `msec` milliseconds. The `timerType` affects the accuracy of the timer.

See also

start()

static PySide2.QtCore.QTimer.singleShot(*msec*, *receiver*, *member*)

```
Parameters:
    msec - int
    receiver - PySide2.QtCore.QObject
    member - str
```

This static function calls a slot after a given time interval.

It is very convenient to use this function because you do not need to bother with a `timerEvent` or create a local `QTimer` object.

Example:

```
from PySide2.QtCore import QApplication, QTimer

def main():

    app = QApplication([])
    QTimer.singleShot(600000, app, SLOT('quit()'))
    ...
    return app.exec_()
```

This sample program automatically terminates after 10 minutes (600,000 milliseconds).

The `receiver` is the receiving object and the `member` is the slot. The time interval is `msec` milliseconds.

See also

setSingleShot(), start()

PySide2.QtCore.QTimer.start()

This function overloads .

Starts or restarts the timer with the timeout specified in `interval` .

If the timer is already running, it will be `stopped` and restarted.

If `singleShot` is true, the timer will be activated only once.

PySide2.QtCore.QTimer.start(*msec*)

```
Parameters:
  msec - int
```

Starts or restarts the timer with a timeout interval of `msec` milliseconds.

If the timer is already running, it will be `stopped` and restarted.

If `singleShot` is true, the timer will be activated only once.

Note

Keeping the event loop busy with a zero-timer is bound to cause trouble and highly erratic behavior of the UI.

PySide2.QtCore.QTimer.stop()

Stops the timer.

See also

start()

PySide2.QtCore.QTimer.timerId()

```
Return type:
  int
```

Returns the ID of the timer if the timer is running; otherwise returns -1.

PySide2.QtCore.QTimer.timerType()

```
Return type:
  TimerType
```

This property holds controls the accuracy of the timer.

The default value for this property is `Qt::CoarseTimer` .

See also

TimerType