

# Qt Concurrent

Дмитрий Фёдоров  
Руководитель отдела разработки ПО



# Проверка связи





## Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



## Поставьте в чат:

-  если меня видно и слышно
-  если нет

# Дмитрий Фёдоров

О спикере:

- более 10 лет в разработке авиационных систем
- возглавляет отдел разработки ПО в НИЦ «ИРТ»



# Вспоминаем прошрое занятие

**Вопрос:** Что такое МОС?



# Вспоминаем прошрое занятие

**Вопрос:** Что такое МОС?

**Ответ:** Метаобъектный компилятор



# Вспоминаем прошрое занятие

**Вопрос:** Что такое сигналы?



# Вспоминаем прошрое занятие

**Вопрос:** Что такое сигналы?

**Ответ:** void методы



# Вспоминаем прошрое занятие

**Вопрос:** Какие отличия есть у слотов относительно обычных методов?





# Вспоминаем прошрое занятие

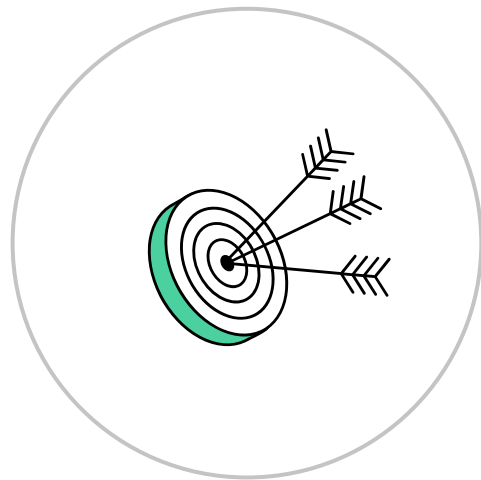
**Вопрос:** Какие отличия есть у слотов относительно обычных методов?

**Ответ:** Не могут быть статическими и нельзя задавать параметры по умолчанию



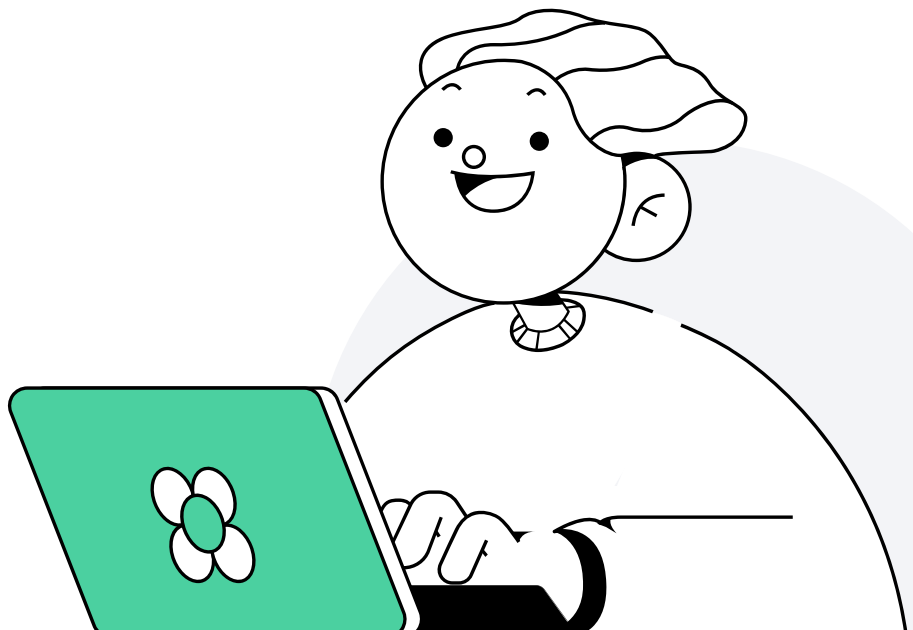
# Цели занятия

- Вспомним многопоточное программирование
- Изучим фреймворк Qt Concurrent
- Напишем программу с его использованием



# План занятия

- 1 Необходимость многопоточного программирования
- 2 Недостатки классического многопоточного программирования
- 3 Многопоточность в Qt
- 4 Фреймворк QtConcurrent
- 5 Итоги
- 6 Домашнее задание



\*Нажми на нужный раздел для перехода

# Необходимость многопоточного программирования



1

# Необходимость многопоточности

Интересный парадокс заключается в том, что при росте вычислительных мощностей, растет и время выполнения вновь возникающих задач.

# Необходимость многопоточности

Интересный парадокс заключается в том, что при росте вычислительных мощностей, растет и время выполнения вновь возникающих задач.

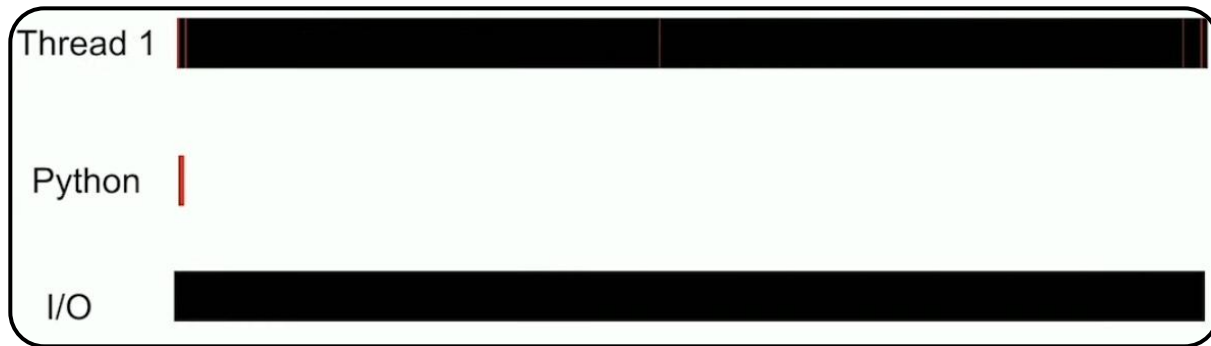
1. Необходимость распараллеливания емких задач.
2. Появляются графические приложения, в которых хочется чтобы форма на зависала (Привет Win 3.11)
3. Клиент-серверные приложения, где необходимо ожидать ответа от сервера.

# Клиент-серверные приложения

Как вы думаете в каком проценте можно соотнести время ожидания ответа от сервера и время обработки этого ответа?

# Клиент-серверные приложения

Как вы думаете в каком проценте можно соотнести время ожидания ответа от сервера и время обработки этого ответа?





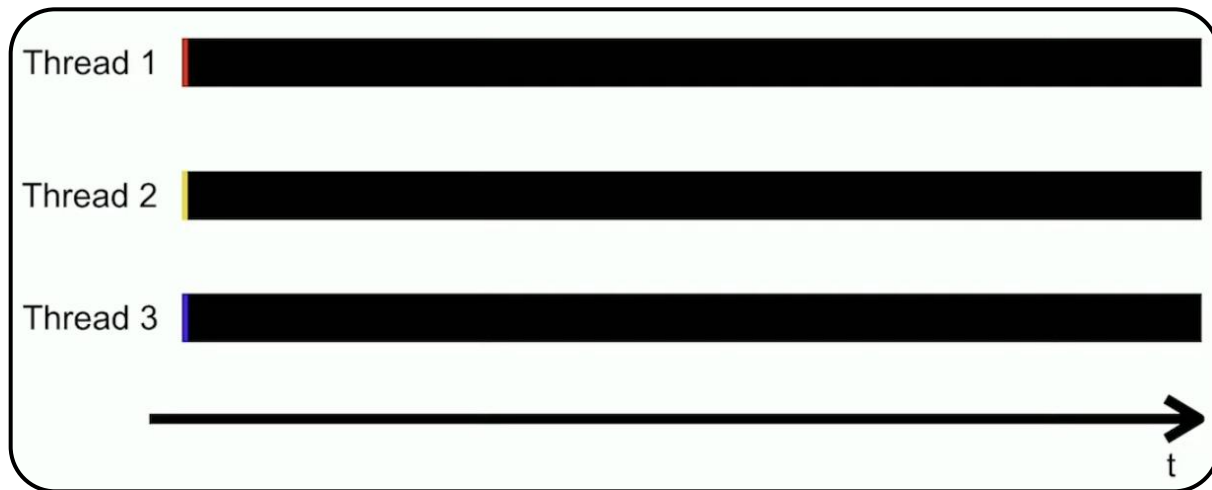
# Клиент-серверные приложения

Появление многоядерных процессоров, позволило по настоящему распараллелить задачи, в том числе и в клиент-серверных приложениях.

# Клиент-серверные приложения

Появление многоядерных процессоров, позволило по настоящему распараллелить задачи, в том числе и в клиент-серверных приложениях.

И теперь эта картина стала выглядеть вот так:



# Достоинства и недостатки многопоточного программирования



2

# Достоинства

- Упрощение программы в некоторых случаях, за счёт вынесения механизмов чередования выполнения различных слабо взаимосвязанных подзадач, требующих одновременного выполнения, в отдельную подсистему многопоточности.
- Повышение производительности процесса за счёт распараллеливания процессорных вычислений.
- Повышение удобства использования за счет распараллеливания процессорных вычислений и операций ввода-вывода(форма не зависает)
- Более эффективное использование ресурсов системы — программы, использующие два или более потоков, могут обращаться к одной области памяти (но это является и проблемой).

# Применение многопоточности

- Распараллеливание сложных инженерных задач.
- Пользовательские графические интерфейсы.
- Работа с клиент-серверными приложениями.
- Декомпозиция одной задачи на  $N$  менее сложных.

# Недостатки многопоточности

- Более сложная отладка приложения по сравнению с однопоточным, особенно при возникновении плавающей ошибки.
- При использовании единой области памяти возникают проблемы синхронизации потоков. В таких ситуациях может возникнуть состояние гонки или взаимная блокировка потоков.



**Примитивы синхронизации** — механизмы, позволяющие реализовать взаимодействие потоков.

Например, единовременный доступ только одного потока к критической области.

# Задачи примитивов синхронизации

- Взаимное исключение потоков — примитивы синхронизации гарантируют то, что одновременно с критической областью будет работать только один поток.
- Синхронизация потоков — примитивы синхронизации помогают отслеживать наступление тех или иных конкретных событий, то есть поток не будет работать, пока не наступило какое-то событие. Другой поток в таком случае должен гарантировать наступление данного события.



# Многопоточность в Qt



2

# Класс QThread

Объект класса QThread управляет одним потоком управления внутри программы.

QThreads начинают выполняться в run().

По умолчанию run() запускает цикл событий путем вызова exec() и запускает цикл событий Qt внутри потока.

# Класс QThread

Для выполнения кода в другом потоке в Qt используются два метода:

1. Создание класса-обертки для использования потока (рассмотрим в коде)
2. Переопределение метода `run()` класса `QThread`.

# Создание класса-обертки

В данном примере метод doWork выполняется в отдельном потоке. Для этого в классе-обертке Controller создается объект класса QThread, объект класса Worker передается потоку при помощи метода moveToThread.

При вызове сигнала operate выполнится метод doWork, а по его завершению вызовется сигнал resultReady и обработается слот handleResult

```
class Worker : public QObject
{
    Q_OBJECT

public slots:
    void doWork(const QString &parameter) {
        QString result;
        /* ... here is the expensive or blocking operation ... */
        emit resultReady(result);
    }

signals:
    void resultReady(const QString &result);
};

class Controller : public QObject
{
    Q_OBJECT
    QThread workerThread;
public:
    Controller() {
        Worker *worker = new Worker;
        worker->moveToThread(&workerThread);
        connect(&workerThread, &QThread::finished, worker, &QObject::deleteLater);
        connect(this, &Controller::operate, worker, &Worker::doWork);
        connect(worker, &Worker::resultReady, this, &Controller::handleResults);
        workerThread.start();
    }
    ~Controller() {
        workerThread.quit();
        workerThread.wait();
    }
public slots:
    void handleResults(const QString &);
signals:
    void operate(const QString &);
};
```

# Переопределение метода run()

В данном примере класс WorkerThread наследуется от QThread в нем переопределяется метод run(), после выполнения которого вызывается сигнал resultReady.

Метод run() вызывается методом start().

```
class WorkerThread : public QThread
{
    Q_OBJECT
    void run() override {
        QString result;
        /* ... here is the expensive or blocking operation ... */
        emit resultReady(result);
    }
signals:
    void resultReady(const QString &s);
};

void MyObject::startWorkInAThread()
{
    WorkerThread *workerThread = new WorkerThread(this);
    connect(workerThread, &WorkerThread::resultReady, this, &MyObject::handleResults);
    connect(workerThread, &WorkerThread::finished, workerThread, &QObject::deleteLater);
    workerThread->start();
}
```

# Qt Concurrent



2



**Конкурентность** — это свойство систем (программы, сети, компьютера и т.д.), допускающее одновременное выполнение нескольких вычислительных процессов, которые могут взаимодействовать друг с другом.

Вычисления запускаются, проходят и завершаются в пересекающихся промежутках времени; они также могут происходить абсолютно одновременно (параллелизм), но это не обязательно.

# Модуль QtConcurrent

Модуль QtConcurrent является высокоуровневым API, который позволяют писать многопоточные программы без использования низкоуровневых примитивов, таких как мьютексы, блокировки чтения-записи, условия ожидания или семафоры.

Также модуль QtConcurrent предоставляет такие методы как `map()`, `mapped()`, `mappedReduced()`, `filter()`, `filtered()` и `filteredReduced()` которые являются API в стиле функционального программирования.



# Подключение Qt Concurrent

```
find_package(Qt6 REQUIRED COMPONENTS Concurrent)  
target_link_libraries(mytarget PRIVATE Qt6::Concurrent)
```

```
QT += concurrent
```

# Concurrent Run

Функция `QtConcurrent::run()` запускает выполнение функции в отдельном потоке. Возвращаемое значение функции становится доступным через `QFuture` API.

Режимы работы `run()`:

- Базовый. Доступны только результаты вычисления пользовательских функций.
- С дополнительным `QPromise`. Доступны результаты промежуточных вычислений, возможности приостановки и возвращения вычислений.

# Класс QFuture

QFuture позволяет синхронизировать потоки с одним или несколькими результатами, которые будут готовы в более поздний момент времени.

QFuture это контейнер для хранения результатов вычисления другого потока

# Основные методы QFuture

- `isStarted()` – возвращает `true` если было запущено асинхронное вычисление представляемое объектом этого `QFuture`.
- `isSuspended()` – возвращает `true` если было запрошено приостановление асинхронного вычисления(доступно только с Qt 6)
- `isRunning()` – возвращает `true` если асинхронное вычисление выполняется.
- `isResultReadyAt(int index)` – возвращает `true` если результат вычисления по запрашиваемому индексу готов.
- `isFinished()` – возвращает `true` если асинхронное вычисление завершено.
- `result()` – возвращает результат асинхронного вычисления.
- `resultAt(int index)` – возвращает результат по запрашиваемому индексу

# Цепочка вычислений QFuture

При помощи метода `then()` можно связать в цепочку последовательность асинхронных операций. Таким образом каждая новая операция будет выполнена только по получению результата предыдущей.

```
QFuture<int> future = ...;  
future.then([](int res1){ ... }).then([](int res2){ ... })...
```

# Обработка исключений в цепочке QFuture

В случае неудачного выполнения асинхронных операций заключенных в цепочку, QFuture предоставляет возможность обработки таких ситуаций.

```
QFuture<int> testFuture = ...;
auto resultFuture = testFuture.then([](int res) {
    // Block 1
}).onFailed([] {
    // Block 3
}).then([] {
    // Block 4
}).onFailed([] {
    // Block 5
}).onCanceled([] {
    // Block 6
});
```

# Запуск вычислений

Запуск в отдельном потоке

```
extern void aFunction();  
QFuture<void> future = QtConcurrent::run(aFunction);
```

Запуск в пуле потоков

```
extern void aFunction();  
QThreadPool pool;  
QFuture<void> future = QtConcurrent::run(&pool, aFunction);
```

# Передача аргументов

```
extern void aFunctionWithArguments(int arg1, double arg2, const QString &string);  
  
int integer = ...;  
double floatingPoint = ...;  
QString string = ...;  
  
QFuture<void> future = QtConcurrent::run(aFunctionWithArguments, integer, floatingPoint, string);
```



# Перегруженные функции

QtConcurrent::run() не поддерживает прямой вызов перегруженных функций. Код с картинки не скомпилируется.

```
void foo(int arg);  
void foo(int arg1, int arg2);  
...  
QFuture<void> future = QtConcurrent::run(foo, 42);
```

```
QFuture<void> future = QtConcurrent::run([] { foo(42); });
```

```
QFuture<void> future = QtConcurrent::run(static_cast<void(*)>(int)>(foo), 42);
```

```
QFuture<void> future = QtConcurrent::run(qOverload<int>(foo), 42);
```

# Использование лямбда-функций

QtConcurrent::run() позволяет использовать лямбда-функции

```
QFuture<void> future = QtConcurrent::run([=]() {  
    // Code in this block will run in another thread  
});  
...
```

# Получение результата

Получение результата происходит при помощи объекта QFuture

```
extern QString someFunction(const QByteArray &input);

QByteArray bytearray = ...;

QFuture<QString> future = QtConcurrent::run(someFunction, bytearray);
...
QString result = future.result();
```

# Пример использования

**Задача.** Необходимо получить большой файл с данными, провести чтение этого файла и выборку необходимых значений, провести некие математические операции над полученными значениями и вывести результат.

**Условия.** Пользовательский интерфейс не должен зависать.

В нашем случае возьмем пример из жизни. Необходимо прочитать файл с результатами АЦП, выбрать определенный канал, перевести коды АЦП в физические величины и найти максимум и минимум в последовательности.

# Итоги



# Итоги занятия

- 1 Узнали как реализована многопоточность в Qt
- 2 Научились работать с объектами QFuture
- 3 Научились работать с QtConcurrent



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



# Дополнительные материалы

- Описание класса [QThread](#)
- Описание класса [QFuture](#)
- Описание модуля [Qt Concurrent](#)
- [Ссылка](#) на файл с тестовыми данными





**Задавайте вопросы  
и пишите отзыв о лекции**

