Felix Petriconi

- Started with C++ 1994
- Programmer and development manager since 2003 at MeVis Medical Solutions AG, Bremen, Germany
  - Development of medical devices in the area of mammography and breast cancer therapy (C++, Ruby)
- Programming activities:
  - Blog editor of ISO C++ website
  - Active member of C++ User Group Bremen
  - Contributor to stlab’s concurrency library
  - Member of ACCU conference committee
- Married with Nicole, having three children, living near Bremen, Germany
- Other interests: Classic film scores, composition
The [C++] language is too large for anyone to master
So everyone lives within a subset

Sean Parent, C++Now, 2012
Why I am here?

I saw how we used different ways to delegate work to different CPU cores
I saw how easy it is to make mistakes
I saw and still see the difficulties in maintaining the code

I listened 2015 to the CppCast with Sean Parent about Concurrency
I was impressed
I wanted to learn more

I started collaborating in his open source project for a new concurrency library
I’m having fun in learning there a lot
I care about sharing my knowledge
Why are you here?
Why do we have to talk about concurrency?
The free lunch is over!

*Herb Sutter, 2005*

---

1The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software
http://www.gotw.ca/publications/concurrency-ddj.htm
The free lunch is over

40 Years of Microprocessor Trend Data

Year


Transistors (thousands)

Single-Thread Performance (SpecINT x 10^3)

Frequency (MHz)

Typical Power (Watts)

Number of Logical Cores

Amdahl’s Law
Amdahl’s Law

\[ S(N) = \frac{1}{(1-P) + \frac{P}{N}} \]

\( S \) : Speed up

\( P \) : Synchronization \([0 - 1]\)

\( N \) : Number of Cores

---

\(^2\)Presented 1967 by Gene Myron Amdahl (1922-2015)
Amdahl’s Law

0% Synchronization

\[ S(N) = \frac{1}{(1-P)+\frac{P}{N}} \]

\[ P = 0 \]
Amdahl’s Law

10% Synchronization

\[ S(N) = \frac{1}{(1-P)+\frac{P}{N}} \]

\[ P = 0.1 \]
Amdahl’s Law

20% Synchronization

\[ S(N) = \frac{1}{(1-P) + \frac{P}{N}} \]

\[ P = 0.2 \]
Amdahl’s Law

90% Synchronization

\[ S(N) = \frac{1}{(1-P)+\frac{P}{N}} \]

\[ P = 0.9 \]
How to use multiple cores?

- Individual single threaded processes
- Multi threaded process without synchronization
- Multi threaded process with synchronization
  - Mutex
  - Atomic
  - Semaphore
  - Memory Fence
  - Transactional Memory

- Multi threaded process with higher level abstractions
  - Future
  - Channel
  - Actor
  - ...
Future Introduction

- Futures provide a mechanism to separate a function $f(...)$ from its result $r$
- After the function is called the result appears "magically" later in the future
- Futures, resp. promises were invented 1977/1978 by Daniel P. Friedman, David Wise, Henry Baker and Carl Hewitt
Future Introduction - Continuation
Future Introduction - Join
Future F3 is not needed any more (e.g. the user has canceled an operation)
Future Introduction - Cancellation
Future Introduction - Cancellation

- There is no need to execute task T3
Future Introduction - Cancellation

T1 \rightarrow F1 \rightarrow T2

F2a

F2b

T4

F4a

F4b

There Is A New Future
Felix Petriconi

Futures
Introduction
Continuation
Join
Split
Cancellation

C++ Standard - Futures
Futures
packaged_task
Exceptions
Capabilities

boost - Futures
Continuations
Join
Capabilities

stlab - Futures
Capabilities
Futures
Exceptions
Continuation
Reduction
Error Recovery
Join
Split
Executors
Conclusion
Then the futures F2a and F4a are not needed any more.
Future Introduction - Cancellation

- The graph collapses to its minimum
C++11 Standard - Futures

- boost::futures were added in boost 1.41, 2009
- std::future are mostly based on boost::futures
- Where added with C++11
C++11 Standard - Futures Overview
There Is A New Future
Felix Petriconi

C++ Standard - Futures

```cpp
#include <future>
#include <iostream>

using namespace std;

int main() {
    auto answer = [] { 
        return 42;
    };

    future<int> f = async(launch::async, answer);

    // Do other stuff, getting the answer may take longer
    cout << f.get() << 'n'; // access the value
}
```

Output

42
```cpp
#include <future>
#include <iostream>
#include <string>

using namespace std;

int main() {
    auto answer = [](string) { return 42; };
    packaged_task<int(string)> task(answer);

    future<int> f = task.get_future();
task("What is the answer ...?");

    // Do other stuff, getting the answer may take longer
    cout << f.get() << '\n'; // access the value
}
```

Output
42
```cpp
int main() {
    auto answer = [] {
        throw runtime_error("Bad things happened: Vogons appeared!");
        return 42;
    };

    future<int> f = async(launch::async, answer);

    // Do other stuff, getting the answer may take longer
    try {
        cout << f.get() << '\n'; // try accessing the value
        // re-throws the stored exception
    } catch (const runtime_error& ex) {
        cout << ex.what() << '\n';
    }
}
```

Output

Bad things happened: Vogons appeared!
What is the biggest problem within this code when our goal is best CPU utilization?

`future<T>.get()` is a blocking call! There is no direct way of checking if the future is ready! Only indirect with `.wait_for()` with zero timeout.
C++11/14/17 Future Capabilities

- No continuation — `future<T>::then()`
- No join — `when_all()` and `when_any()`
- No split — continuation in different directions
- No cancellation (but can be modelled)
- No automatic reduction (`future<future<T>>::future<T>`) 
- No progress monitoring (except ready)
- No custom executor
- Blocks on destruction (may even blocks until termination of used thread)
- Usage of `future<T>::get()` has two problems:
  1. One thread resource is consumed which increases contention and possibly causing a deadlock
  2. Any subsequent non-dependent calculations on the task are also blocked

- Don’t behave as a regular type

3. https://gist.github.com/sean-parent/24df3eefda5b3a482f6e71da2c2
4. Elements of Programming; Stepanov, McJones; Addison-Wesley 2009
boost Futures Overview
A Continuation on an existing future is realized through `future<T>.then()` which returns itself a future
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
    auto answer = []{ return 42; };
    auto report_answer = [] (auto a) { cout << a.get() << '
'; }

    boost::future<int> get_answer = boost::async(answer);

    boost::future<void> done = get_answer.then(report_answer);

    // do something else
    done.wait(); // waits until future done is fulfilled
}

Output
42
when_all() Returns a future that becomes ready when all future arguments are ready
when_any() Returns a future that becomes ready when the first future argument is ready
```
int main() {
    auto answer_a = []{ return 40; };
    auto answer_b = []{ return 2; };

    auto f_a = boost::async(answer_a);
    auto f_b = boost::async(answer_b);

    auto answer = boost::when_all(std::move(f_a), std::move(f_b))
        .then([](auto f) {
            auto t = f.get();
            return get<0>(t).get() + get<1>(t).get();
        });

    // do something else
    cout << answer.get() << '\n';
}
```

Output

42
int main() {
    auto answer_a = []{ return 40; };
    auto answer_b = []{ return 2; };

    auto f_a = boost::async(answer_a);
    auto f_b = boost::async(answer_b);

    auto answer = boost::when_all(std::move(f_a), std::move(f_b))
        .then([](auto f) {
            auto t = f.get();
            return get<0>(t).get() + get<1>(t).get();
        });

    // do something else
    cout << answer.get() << '
';
}

What is the type of f?

f is a future tuple of futures: future<tuple<future<int>, future<int>>>
C++17 TS / boost - Futures Capabilities

- Continuation — `future<T>.then()` ✓
- Join — `when_all()` and `when_any()` ✓
- No real split — continuations into different directions ✗
- No cancellation (but can be modelled) ✗
- No automatic reduction (`future<future<T>> ⇒ future<T>`) ✗
- No progress monitoring (except ready) ✗
- Custom executor ✓
- Blocks on destruction (may even blocks until termination of used thread) ✗
- Using `future<T>.get()` has two problems:
  1. One thread resource is consumed which increases contention and possibly causing a deadlock ✗
  2. Any subsequent non-dependent calculations on the task are also blocked ✗
- Don’t behave as a regular type ✗
- (C++17 TS is in namespace experimental and there is no interoperation between between std::experimental::future and std::future)
There Is A New Future
stlab Futures

stlab::future

Source: https://github.com/stlab/libraries
Documentation: http://www.stlab.cc/libraries
stlab - Futures Capabilities

- Continuation — `future<T>.then()` ✓
- Join — `when_all()` and `when_any()` ✓
- Split — continuation in different directions ✓
- Cancellation ✓
- Automatic reduction (`future<future<T>> ⇒ future<T>`) ✓
- No progress monitoring (except ready), more planned ✗
- Custom executor ✓
- Do not block on destruction ✓
- Behave as a regular type ✓
- Additional dependencies:
  - C++14: boost.optional
  - C++17: none
There Is A New Future

Felix Petriconi

Introduction
Continuation
Join
Split
Cancellation
C++ Standard - Futures
Futures
packaged_task
Exceptions
Capabilities
boost - Futures
Continuations
Join
Capabilities
stlab - Futures
Capabilities
Futures
Exceptions
Reduction
Error Recovery
Join
Split
Executors
Conclusion

#include <iostream>
#include <stlab/concurrency/default_executor.hpp>
#include <stlab/concurrency/future.hpp>
#include <stlab/concurrency/utility.hpp>

using namespace std;

int main() {
    auto answer = [] { return 42; };

    stlab::future<int> f =
        stlab::async(
            stlab::default_executor,// uses platform thread pool on Win/OSX
            // uses stlab task stealing
            // thread pool on other OS, e.g. Linux
            answer
        );
```cpp
int main() {
    auto answer = [] { return 42; };

    stlab::future<int> f =
        stlab::async(
            stlab::default_executor, // uses platform thread pool on Win/OSX
            // uses stlab task stealing
            // thread pool on other OS, e.g. Linux
            answer
        );

    while (!f.get_try()) {} // do something meaningful while waiting
    // Don't do busy waiting!
    std::cout << f.get_try().value() << '\n';
}
```

Output
---
42
```cpp
int main() {
  auto answer = [] { return 42; };

  stlab::future<int> f =
  stlab::async(
      stlab::default_executor, // uses platform thread pool on Win/OSX
      // uses stlab task stealing
      // thread pool on other OS, e.g. Linux
      answer
  );

  // access the value in a blocking way
  // try to avoid this whenever it is possible!
  cout << stlab::blocking_get(std::move(f)) << 'n';
}
```
There Is A New Future

Felix Petriconi

Futures
Introduction
Continuation
Join
Split
Cancellation

C++ Standard - Futures
Packaged task
Exceptions
Capabilities

boost - Futures
Continuations
Join
Capabilities

stlab - Futures
Capabilities
Futures
Exceptions
Capabilities

Reduction
Error Recovery
Join
Split
Executors
Conclusion

---

```cpp
int main() {
    auto answer = [] { 
        throw std::runtime_error("Bad thing happened: Vogons appeared!");
        return 42;
    };
    auto f = stlab::async(stlab::default_executor, answer);

    try {
        std::cout << stlab::blocking_get(std::move(f)) << '
';
    }
    catch (std::runtime_error const& ex) {
        std::cout << ex.what() << '
';
    }
}
```

Output

Bad things happened: Vogons appeared!
stlab Futures - Continuation

```cpp
using namespace stlab;

int main() {
    auto answer = []( ) { return 42; };

    auto report_answer = []( int a) { std::cout << a << '\n'; };

    future<void> done = async(default_executor, answer)
        .then(report_answer);  // Call by value and not by future

    int quit; std::cin >> quit;
}
```

Output

42
int main() {
    future<int> result =
        async(default_executor, [] { return 42; })
        .then(
            [](int x) {
                return async(default_executor,
                    [](int y) { return y + 42; },
                    x);
            });

    future<void> done = result.then(
        [](int v) { std::cout << v << '\n'; })
    );

    int quit; std::cin >> quit;
}

Output

84
int main() {
    auto answer = [] {} {
        throw std::runtime_error("Vogons appeared!");
        return 42;
    };

    auto handleTheAnswer = [] (int v) {
        if (v == 0) std::cout << "Oh! We have a problem!\n";
        else std::cout << "The answer is " << v << '\n';
    };
}
auto handleTheAnswer = [](int v) {
    if (v == 0) std::cout << "Oh! We have a problem!\n";
    else std::cout << "The answer is " << v << '\n';
};

auto f = stdlab::async(stlab::default_executor, answer)
    .recover([](stlab::future<int> result) {
    if (result.error()) {
        std::cout << "Listen to Vogon poetry!\n";
        return 0;
    }
    return result.get_try().value();
}).then(handleTheAnswer);

int quit; std::cin >> quit;
int main() {
    auto a = async(default_executor,[]{ return 40; });
    auto b = async(default_executor,[]{ return 2; });

    auto answer = when_all(
        default_executor,
        [](int x, int y) { return x + y; },
        a, b); // arguments as lvalues

    std::cout << stlab::blocking_get(std::move(answer)) << '\n';
}
A split is realized by creating multiple continuations on the same future
int main() {
    auto answer = async(default_executor, []{ return 42; });
    auto report_to_arthur = [] (int a) {
        printf("Tell the answer %d Arthur Dent\n", a);
    };
    auto report_to_marvin = [] (int a) {
        printf("May the answer %d shear up Marvin\n", a);
    };
    auto dent = answer.then(report_to_arthur);
    auto marvin = answer.then(report_to_marvin);
    blocking_get(dent);
    blocking_get(marvin);
}
int main() {
    auto answer = async(default_executor, [] { return 42; });

    auto report_to_arthur = [](int a) {
        printf("Tell the answer %d Arthur Dent\n", a);
    };
    auto report_to_marvin = [](int a) {
        printf("May the answer %d shear up Marvin\n", a);
    };

    auto dent = answer.then(report_to_arthur);
    auto marvin = answer.then(report_to_marvin);

    auto done = when_all(default_executor, [] {
        std::cout << "All know the answer!\n";
    }, marvin, dent);

    stlab::blocking_get(done);
}
Executors

- Executors are needed to customize where the task shall be executed
- Executors can be
  - thread pools
  - serial queues
  - main queues
  - dedicated task groups
  - etc.
- std executors are probably/hopefully coming with C++20
- In boost, executors derive from a common base class
- In stlab the executors must only implement the call operator
  `template <typename F> void operator()(F f)`
- stlab currently has
  - `default_executor (thread pool)`
  - `immediate_executor`
  - `main_executor`
  - `system_timer`
- See bonus slides for implementation of an executor for the Qt main-loop
stlab Futures - Continuation with Custom Executor

```cpp
#include <iostream>
#include <QLineEdit>
#include <stlab/concurrency/default_executor.hpp>
#include <stlab/concurrency/future.hpp>
#include "QtExecutor.h"

int main() { // Just illustrational example!
    QLineEdit theAnswerLineEdit;

    auto answer =
        stlab::async(stlab::default_executor, []{ return 42; });

    stlab::future<void> done = answer.then(
        QtExecutor{}, // different scheduler
        [&](int a) {
            theAnswerLineEdit.setValue(a); // update in Qt main thread
        });

    int quit; std::cin >> quit;
}
stlab Upcoming enhancements

- Coroutine support
- Performance optimization
- Progress monitoring
- Task promotion
Futures are a great tool to structure code in a well readable manner so that it runs in parallel with minimal contention.

But the graph can be used for a single execution only.

Channels are one concept that supports multiple invocations.
Channels allow the creation of persistent execution graphs
First published by Tony Hoare 1978
Channel Introduction

There Is A New Future
Felix Petriconi

Motivation

Channel - Stateless Process
Channel - Split
Channel - Join
Channel - Stateful Process

Conclusion
# include <iostream>
# include <stlab/concurrency/channel.hpp>
# include <stlab/concurrency/default_executor.hpp>

int main() {
    stlab::sender<int> send;       // sending part of the channel
    stlab::receiver<int> receiver; // receiving part of the channel
    std::tie(send, receiver) =    // combining both to a channel
        stlab::channel<int>(stlab::default_executor);

    auto printer =
        [](int x){ std::cout << x << '\n'; }; // stateless process

    auto printer_process =
        receiver | printer; // attaching process to the receiving
        // part
    receiver.set_ready();  // no more processes will be attached
                          // process starts to work
    send(1); send(2); send(3); // start sending into the channel

    int end; std::cin >> end;   // simply wait to end application
}
int main() {
    auto printer = [] (int x) { std::cout << x << '\n'; }; // stateless process

    auto printer_process = receiver | printer; // attaching process to the receiving part

    receiver.set_ready(); // no more processes will be attached

    send(1); send(2); send(3); // start sending into the channel

    int end; std::cin >> end; // simply wait to end application
}

Output

1
2
3
New edges are concatenated with the `operator |()` on the same receiver.
There Is A New Future

Felix Petriconi

Motivation

Channel - Stateless Process
Channel - Split Channel
Channel - Join Channel - Stateful Process
Conclusion

Channel - Split Process

```cpp
int main() {
    auto [send, receiver] = channel<int>(default_executor); // C++17

    auto printerA = [](int x){ printf("Process A %d\n", x); }; auto printerB = [](int x){ printf("Process B %d\n", x); };

    auto printer_processA = receiver | printerA; auto printer_processB = receiver | printerB;

    receiver.set_ready(); // no more processes will be attached
    // process may start to work
    send(1); send(2); send(3);
    int end; std::cin >> end;
}
```

Output

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Channel - Join

- `join()` The downstream process is invoked when all arguments are ready.
- `zip()` The downstream process is invoked in round robin manner with the incoming values.
- `merge()` The downstream process is invoked with the next value that is ready.
Additional channel options

- With `buffer_size\{n\}` within the concatenation it is possible to limit the incoming queue to size `n`.
- With `executor\{T\}` within the concatenation it is possible to specify a dedicated executor `T`. 
Some problems need a processor with state

Some problems have an n : m relationship from input to output

The picture becomes more complicated with states:
  ▸ When to proceed?
  ▸ How to handle situations when less than expected values come from upstream?
There Is A New Future
Felix Petriconi

Motivation

Channel - Stateless Process
Channel - Split
Channel - Join

Channel - Stateful Process

Conclusion

```
#include <stlab/concurrency/channel.hpp>

using process_state_scheduled = 
    std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close(); // optional

    void set_error(std::exception_ptr); // optional
};
```
Stateful Process Signature - await

```cpp
#include <stlab/concurrency/channel.hpp>

using process_stateScheduled = std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_stateScheduled state() const;

    void close(); // optional

    void set_error(std::exception_ptr); // optional
};
```
Stateful Process Signature - yield

```cpp
#include <stlab/concurrency/channel.hpp>

using process_state_scheduled = std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);
    U yield();

    process_state_scheduled state() const;

    void close();       // optional
    void set_error(std::exception_ptr); // optional
};
```
There Is A New Future

Felix Petriconi

Motivation
Channel - Stateless
Channel - Split
Channel - Join
Channel - Stateful
Process

Conclusion

Stateful Process Signature - state

```cpp
#include <stlab/concurrency/channel.hpp>

using process_state_scheduled = 
   std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close(); // optional

    void set_error(std::exception_ptr); // optional
};
```
Stateful Process Signature - close

```cpp
#include <stlab/concurrency/channel.hpp>

using process_state_scheduled = 
    std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close(); // optional

    void set_error(std::exception_ptr); // optional
};
```
# include <stlab/concurrency/channel.hpp>

using process_state_scheduled =
   std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close(); // optional

    void set_error(std::exception_ptr); // optional
};
Channel - Stateful Process Example

```cpp
struct adder {
};

int main() {
    auto [send, receiver] = channel<int>(default_executor);

    auto calculator = receiver | adder{} | [](int x) { std::cout << x << '
'; };

    receiver.set_ready();

    while (true) {
        int x;
        std::cin >> x;
        send(x);
    }
}
```
struct adder {
    int _sum = 0;
    process_state_scheduled _state = await_forever;

    void await(int x) {
        _sum += x;
        if (x == 0) {
            _state = yield_immediate;
        }
    }

    int yield() {
        int result = _sum;
        _sum = 0;
        _state = await_forever;
        return result;
    }

    auto state() const { return _state; }
};
Channels close the gap of multiple invocations where futures allow just one. With splits and the different kind of joins it is possible to build graphs of execution.
Take Away

Use high level abstractions like futures, channels or others (actors, etc.) to distribute work on available CPU cores.

Use thread pools from your operating system! Use highly optimized task stealing custom thread pools in case that the operating system does not provide one!

Design your application with the mindset that it can run dead-lock free on an 1-n core hardware!

Don’t let your application code be soaked with threads, mutex’ and atomics.
Acknowledgement

- My family, who supports me in my work on the concurrency library and this conference.
- Sean Parent, who taught me over time lots about concurrency and abstraction. He gave me the permission to use whatever I needed from his presentations for my own.
- My company MeVis Medical Solutions AG, that released me from work during this conference.
- All contributors to the stlab library.
Reference

- Concurrency library https://github.com/stlab/libraries
- Documentation http://stlab.cc/libraries
- Back to std2::future, C++ Standard Proposal by Bryce Adelstein Lelbach http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0701r0.html
Further reading I

Software Principles and Algorithms

- Elements of Programming by Alexander Stepanov, Paul McJones, Addison Wesley
- From Mathematics to Generic Programming by Alexander Stepanov, Daniel Rose, Addison Wesley
Further reading II

Concurrency and Parallelism

- HPX http://stellar-group.org/libraries/hpx/
- C++CSP https://www.cs.kent.ac.uk/projects/ofa/c++csp
- CAF_C++ Actor Framework http://actor-framework.org/
- C++ Concurrency In Action by Anthony Williams, Manning
Further listening and viewing

- Goals for better code by Sean Parent: Concurrency: https://youtu.be/au0xX4h8SCI?t=16354
- CppCast with Sean Parent http://cppcast.com/2015/06/sean-parent/
- Thinking Outside the Synchronization Quadrant by Kevlin Henney: https://vimeo.com/205806162
stlab Futures

stlab::future

Source: https://github.com/stlab/libraries
Documentation: http://www.stlab.cc/libraries
Thank’s for your attention!

- Mail: felix@petrironi.net
- GitHub: https://github.com/FelixPettriconi
- Web: https://petriconi.net
- Twitter: @FelixPettriconi
Q & A

- Mail: felix@petriconi.net
- GitHub: https://github.com/FelixPetriconi
- Web: https://petriconi.net
- Twitter: @FelixPetriconi

Feedback is always welcome!
#include <QApplication>
#include <Event>
#include <stlab/concurrency/task.hpp>

class QtExecutor
{
    using result_type = void;

class ExecutorEvent : public QEvent

class EventReceiver : public QObject
public:
    template <typename F>
    void operator()(F f) {
        auto event = std::make_unique<ExecutorEvent>();
        event->set_task(std::move(f))
        QApplication::postEvent(event->receiver(), event.release());
    }
};
class ExecutorEvent : public QEvent
{
    stlab::task<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

public:
    ExecutorEvent()
        : QEvent(QEvent::User), _receiver(new EventReceiver()) {
        _receiver()->moveToThread(QApplication::instance()->thread());
    }

    template<typename F>
    void set_task(F&& f) {
        _f = std::forward<F>(f);
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
};
class EventReceiver : public QObject
{
public:
    bool event(QEvent *event) override {
        auto myEvent = dynamic_cast<ExecutorEvent*>(event);
        if (myEvent) {
            myEvent->execute();
            return true;
        }
        return false;
    }
};