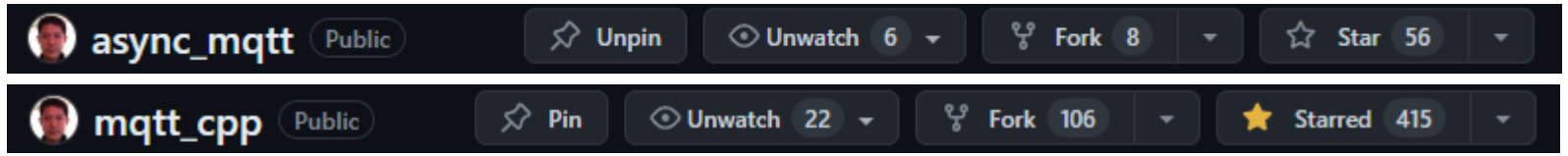
Boost.Asioにおける coroutineの活用法

C++20以降で使える、asioとcoroutineの組み合わせ

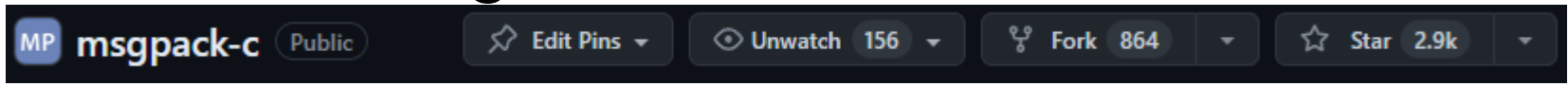
近藤 貴俊 redboltz

発表者について

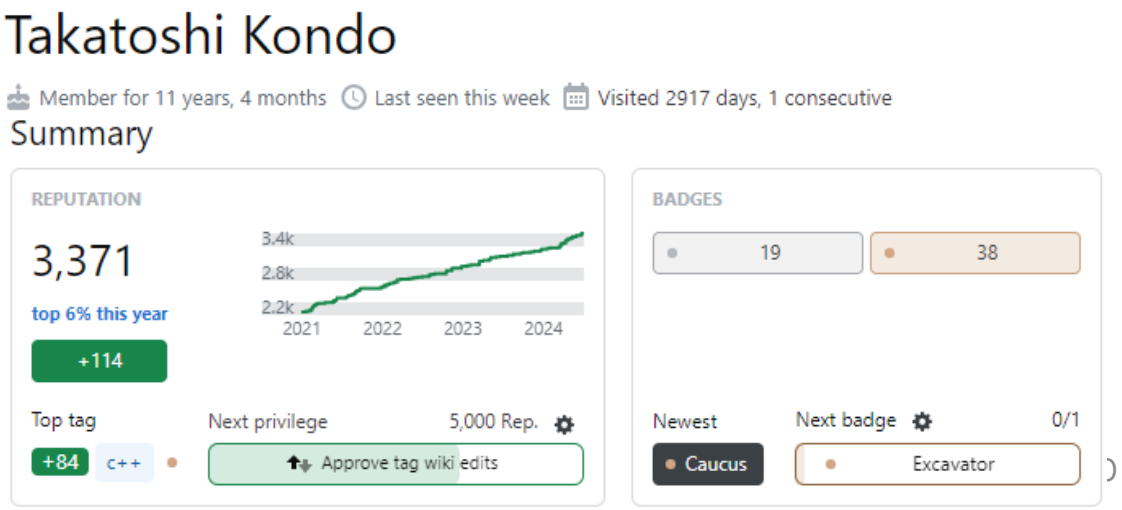
- 近藤貴俊 redboltz
- Boost Libraries コントリビュータ
- MQTT (IoT分野でよく使われる軽量プロトコル)
  ライブラリ作者
  async_mqtt, mqtt_cpp



- 軽量シリアライズフォーマット MessagePack の CおよびC++ 版
  msgpack-c メンテナ



- stackoverflowでQ&A活動中



Boost.Asioとは?

- Asynchronous IO で asio
- 非同期処理の基盤となるメカニズムを提供
- 非同期処理の各種応用を提供
  - ネットワーク通信
  - タイマ
  - コンソール入出力
  - シグナル
  - etc

非同期処理の完了を通知する手段

- コールバック
  - 関数、関数オブジェクト、ラムダ式 など
- CompletionToken
  - use_future, use_awaibale, deferred, experimental::use_promise など

本日のメインテーマ

asioスタイルの非同期関数の例

```
template <typename CompletionToken>
auto async_function(
  int p1,
  double p2,
  CompletionToken&& token
);
```

ここに何を渡すかで戻り値の型が変わる

callback vs coroutine

```cpp
void test(as::any_io_executor exe) {
    auto tim1 = std::make_shared<as::steady_timer>(exe, std::chrono::milliseconds{1});
    tim1->async_wait(
        [tim1, exe](auto ec1) {
            std::cout << ec1.message() << std::endl;
            auto tim2 = std::make_shared<as::steady_timer>(exe, std::chrono::milliseconds{1});
            tim2->async_wait(
                [tim2](auto ec2) {
                    std::cout << ec2.message() << std::endl;
                }
            );
        }
    );
}
```

https://godbolt.org/z/MfW5MoTcY

```cpp
as::awaitable<void> test() {
    auto exe = co_await as::this_coro::executor;

    as::steady_timer tim1{exe, std::chrono::milliseconds{1}};
    auto [ec1] = co_await tim1.async_wait(as::as_tuple(as::use_awaitable));
    std::cout << ec1.message() << std::endl;

    as::steady_timer tim2{exe, std::chrono::milliseconds{2}};
    auto [ec2] = co_await tim2.async_wait(as::as_tuple(as::use_awaitable));
    std::cout << ec2.message() << std::endl;

    co_return;
}
```

https://godbolt.org/z/M7rMe6ren

```cpp
// Resolve hostname
res.async_resolve(
    argv[1],
    argv[2],
    [&]
    (boost::system::error_code ec, as::ip::tcp::resolver::results_type eps) {
        std::cout << "async_resolve:" << ec.message() << std::endl;
        if (ec) return;
        // Layer
        // am::stream -> TCP

        // Underlying TCP connect
        as::async_connect(
            amep->next_layer(),
            eps,
            [&]
            (boost::system::error_code ec, as::ip::tcp::endpoint /*unused*/) {
                std::cout
                    << "TCP connected ec:"
                    << ec.message()
                    << std::endl;
                if (ec) return;
                // Send MQTT CONNECT
                amep->send(
                    am::v3_1_1::connect_packet{
                        true,   // clean_session
                        0x1234, // keep_alive
                        am::allocate_buffer("cid1"),
                        am::nullopt, // will
                        am::nullopt, // username set like am::allocate_buffer("user1"),
                        am::nullopt  // password set like am::allocate_buffer("pass1")
                    },
                    [&]
                    (am::system_error const& se) {
                        if (se) {
                            std::cout << "MQTT CONNECT send error:" << se.what() << std::endl;
                            return;
                        }
                        // Recv MQTT CONNACK
                        amep->recv(
                            [&]
                            (am::packet_variant pv) {
                                if (pv) {
                                    pv.visit(
                                        am::overload {
                                            [&](am::v3_1_1::connack_packet const& p) {
                                                std::cout
                                                    << "MQTT CONNACK recv"
                                                    << " sp:" << p.session_present()
                                                    << std::endl;
                                                // Send MQTT SUBSCRIBE
                                                amep->send(
                                                    am::v3_1_1::subscribe_packet{
                                                        *amep->acquire_unique_packet_id(),
                                                        { {am::allocate_buffer("topic1"), am::qos::at_most_once} }
                                                    },
                                                    [&]
                                                    (am::system_error const& se) {
                                                        if (se) {
                                                            std::cout << "MQTT SUBSCRIBE send error:" << se.what() << std::endl;
                                                            return;
                                                        }
                                                        // Recv MQTT SUBACK
                                                        amep->recv(
                                                            [&]
                                                            (am::packet_variant pv) {
                                                                if (pv) {
```

callback

VS

coroutine

```cpp
// Resolve hostname
auto eps = co_await res.async_resolve(host, port, as::use_awaitable);
std::cout << "async_resolved" << std::endl;

// Layer
// am::stream -> TCP

// Underlying TCP connect
co_await as::async_connect(
    amep->next_layer(),
    eps,
    as::use_awaitable
);
std::cout << "TCP connected" << std::endl;

// Send MQTT CONNECT
if (auto se = co_await amep->send(
        am::v3_1_1::connect_packet{
            true,   // clean_session
            0x1234, // keep_alive
            am::allocate_buffer("cid1"),
            am::nullopt, // will
            am::nullopt, // username set like am::allocate_buffer("user1"),
            am::nullopt  // password set like am::allocate_buffer("pass1")
        },
        as::use_awaitable
    )
) {
    std::cout << "MQTT CONNECT send error:" << se.what() << std::endl;
    co_return;
}

// Recv MQTT CONNACK
if (am::packet_variant pv = co_await amep->recv(as::use_awaitable)) {
    pv.visit(
        am::overload {
            [&](am::v3_1_1::connack_packet const& p) {
                std::cout
                    << "MQTT CONNACK recv"
                    << " sp:" << p.session_present()
                    << std::endl;
            },
            [](auto const&) {}
        }
    );
}
else {
    std::cout
        << "MQTT CONNACK recv error:"
        << pv.get<am::system_error>().what()
        << std::endl;
    co_return;
}

// Send MQTT SUBSCRIBE
std::vector<am::topic_subopts> sub_entry{
    {am::allocate_buffer("topic1"), am::qos::at_most_once}
};
if (auto se = co_await amep->send(
        am::v3_1_1::subscribe_packet{
            *amep->acquire_unique_packet_id(),
            am::force_move(sub_entry) // sub_entry variable is required to avoid g++ bug
        },
        as::use_awaitable
    )
) {
    std::cout << "MQTT SUBSCRIBE send error:" << se.what() << std::endl;
    co_return;
}
// Recv MQTT SUBACK
if (am::packet_variant pv = co_await amep->recv(as::use_awaitable)) {
```

co_await

- 非同期処理の完了を待つことができる
- どこで待てるのか

戻り値の型がawaitable<T> の関数の中

```
as::awaitable<void> async_func() {
  co_await ここで待てる
  co_return;
}
```

co_composedに渡すラムダ式の中

```
template <typename CompletionToken>
auto async_func(CompletionToken&& token) {
  return as::async_initiate<
    CompletionToken,
    void()
  >(
    as::experimental::co_composed<
      void()
    >(
      [](
        auto /*state*/
      ) -> void {
        co_await ここで待てる
        co_return {};
      }
    ),
    token
  );
}
```

co_await

- なにを待てるのか

戻り値の型がawaitable<T> の関数呼び出し

```
as::awaitable<void> async_func();
co_await async_func();
```

CompletionTokenとして、use_awaitableを渡した関数呼び出し

```
co_await as::post(as::use_awaitable);
```

CompletionTokenとして、deferredを渡した関数呼び出し

```
co_await as::post(as::deferred);
```

CompletionTokenとして、use_promiseを渡した関数呼び出し

```
co_await as::post(as::experimental::use_promise);
```

co_await

- どこでなにを待てるのか

戻り値の型がawaitable<T> の関数の中

```
as::awaitable<void> test() {
  co_await ここで待てる
  co_return;
}
```

全て待てる

https://godbolt.org/z/xEd4qWEj3

戻り値の型がawaitable<T> の関数呼び出し

```
as::awaitable<void> async_func();
```

```
co_await async_func();
```

CompletionTokenとして、use_awaitableを渡した関数呼び出し

```
co_await as::post(as::use_awaitable);
```

CompletionTokenとして、deferredを渡した関数呼び出し

```
co_await as::post(as::deferred);
```

CompletionTokenとして、use_promiseを渡した関数呼び出し

```
co_await as::post(as::experimental::use_promise);
```

co_await

- どこでなにを待てるのか

co_composedに渡すラムダ式の中

```
template <typename CompletionToken>
auto test(
  CompletionToken&& token
) {
  return as::async_initiate<
    CompletionToken,
    void()
  >(
    as::experimental::co_composed<
      void()
    >(
      [](
        auto /*state*/
      ) -> void {
        co_await ここで待てる
        co_return {};
      }
    ),
    token
  );
}
```

awaitable以外待てる

戻り値の型がawaitable<T> の関数呼び出し

```
as::awaitable<void> async_func();
```

```
co_await async_func();
```

CompletionTokenとして、use_awaitableを渡した関数呼び出し

```
co_await as::post(as::use_awaitable);
```

CompletionTokenとして、deferredを渡した関数呼び出し

```
co_await as::post(as::deferred);
```

CompletionTokenとして、use_promiseを渡した関数呼び出し

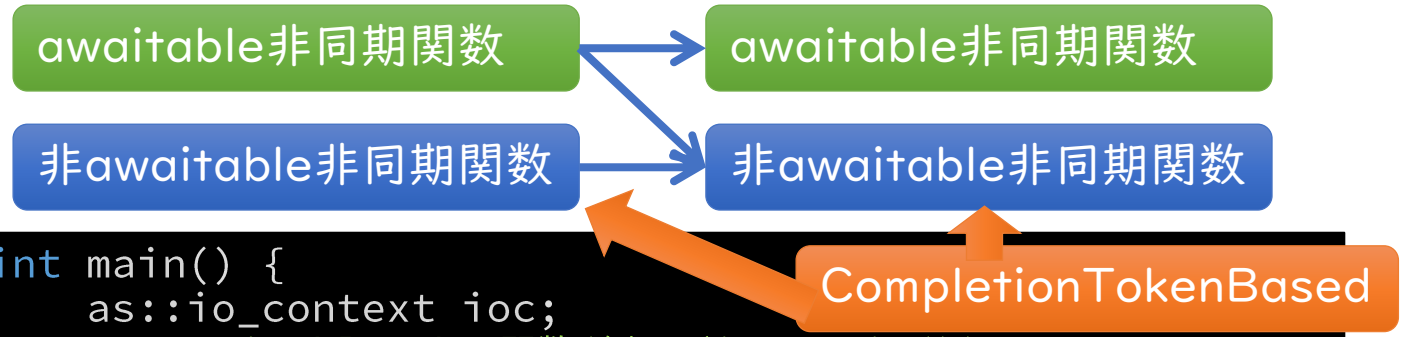
```
co_await as::post(as::experimental::use_promise);
```

https://godbolt.org/z/dbfhdY7jG

co_composedのうれしさ

- CompletionTokenベースの関数内で co_awaitが使える
  - awaitableの連鎖を断ち切ることができる
  - ただし、co_composed関数の中からawaitableな関数は呼び出せない

```
template <typename CompletionToken>
auto completion_token_based_async_func(
  CompletionToken&& token
) {
  return as::async_initiate<
    CompletionToken,
    void()
  >(
    as::experimental::co_composed<
      void()
    >(
      [](
        auto /*state*/
      ) -> void {
        co_await awaitable以外の非同期関数
        co_return {};
      }
    ),
    token
  );
}
```




```
int main() {
    as::io_context ioc;
    // awaitableでない関数(例えばint main())
    // から呼び出すことができる
    completion_token_based_async_func(
        ioc.get_executor(),
        []{ std::cout
            << "async_func() finished"
            << std::endl;
        }
    );
    ioc.run();
    std::cout << "main() finished" << std::endl;
}
```

https://godbolt.org/z/snx1643b4

co_awaitとマルチウェイト

```
as::awaitable<void> test() {
  co_await async_func1();
  co_await async_func2();
  co_return;
}
```

マルチウェイトになっていない例
async_func1()の非同期処理の完了を待ってから
async_func2()を開始してしまう

- 典型的なマルチウェイトのパターン

ひとつ待ち
- 複数の非同期関数を呼び出し、最初の結果が来たら他をキャンセルする
  - 例:タイムアウト付き受信処理

全部待ち
- 複数の非同期関数を呼び出し、全ての結果がそろってから処理を行う
  - 例: メッセージの一斉配信処理

- 待つ要素の種類と数

静的
- コンパイル時に決まっている
  - 例:1つの受信待ちに1つのタイムアウト

動的
- 実行時に決まる
  - 例:現在接続しているクライアント全員にメッセージ配信し、
    全ての非同期送信の結果を待つ

- awaitableか非awaitableかで方法が異なる

awaitable　非awaitable

co_awaitとマルチウェイト

awaitableな非同期関数群のマルチウェイト　https://godbolt.org/z/nxzPoEafz

静的

awaitable

全部待ち

co_await (aaa() && bbb())

```cpp
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/experimental/awaitable_operators.hpp>

namespace as = boost::asio;
using namespace as::experimental::awaitable_operators;

as::awaitable<std::tuple<int, double>> async_func() {
    co_return std::make_tuple(10, 20.1);
}

as::awaitable<void> test() {
    auto exe = co_await as::this_coro::executor;
    auto [res11, res12, res2] = co_await (
        async_func() && as::post(exe, as::append(as::use_awaitable, 30, 40.1, 50))
    );
    auto [res21, res22, res23] = res2;
    std::cout << res11 << " " << res12 << std::endl;
    std::cout << res21 << " " << res22 << " " << res23 << std::endl;
    co_return;
}
```

今ひとつ釈然としないが、こういう仕様

```
10 20.1
30 40.1 50
```

co_awaitとマルチウェイト

https://godbolt.org/z/5rb34T38Y

静的

awaitable

ひとつ待ち

co_await (aaa() || bbb())

```
略
as::awaitable<std::tuple<int, double>> async_func() {
    auto exe = co_await as::this_coro::executor;
    as::steady_timer tim{exe, std::chrono::milliseconds(1)};
    auto [ec] = co_await tim.async_wait(as::as_tuple(as::deferred));
    std::cout << ec.message() << std::endl;
    co_return std::make_tuple(10, 20.1);
}

as::awaitable<void> test() {
    auto exe = co_await as::this_coro::executor;
    auto var = co_await (
        as::post(exe, as::append(as::use_awaitable, 30, 40.1, 50)) || async_func()
    );
    if (auto const* p = std::get_if<0>(&var)) {
        auto [res11, res12, res13] = *p;
        std::cout << res11 << " " << res12 << " " << res13 << std::endl;
    }
    else if (auto const* p = std::get_if<1>(&var)) {
        auto [res21, res22] = *p;
        std::cout << res21 << " " << res22 << std::endl;
    }
    co_return;
}
```

```
Operation canceled
30 40.1 50
```

co_awaitとマルチウェイト

https://godbolt.org/z/1Tx4rE75n

静的
非awaitable
全部待ち

make_parallel_group(funcs...)
wait_for_all()

```
略
#include <boost/asio/experimental/parallel_group.hpp>
namespace as = boost::asio;
as::awaitable<void> test() {
    auto exe = co_await as::this_coro::executor;
    auto [orders, res11, res12, res21, res22, res23] =
        co_await as::experimental::make_parallel_group(
            as::post(exe, as::append(as::deferred, 10, 20.1)),
            as::dispatch(exe, as::prepend(as::deferred, 30, 40.1, 50))
        ).async_wait(
            as::experimental::wait_for_all(),
            as::deferred
        );
    for (auto order : orders) {
        std::cout << order;
        switch (order) {
        case 0:
            std::cout << " " << res11 << " " << res12 << std::endl;
            break;
        case 1:
            std::cout << " " << res21 << " " << res22 << " " << res23 << std::endl;
            break;
        }
    }
    co_return;
}
```

```
1 30 40.1 50
0 10 20.1
```

co_awaitとマルチウェイト

https://godbolt.org/z/hn8GMsIWe

静的
非awaitable
ひとつ待ち

```
as::awaitable<void> test() {
  auto exe = co_await as::this_coro::executor;
  as::steady_timer tim1{exe, std::chrono::milliseconds(20)};
  as::steady_timer tim2{exe, std::chrono::milliseconds(10)};
  auto [orders, ec1, res11, res12, ec2, res21, res22, res23] =
    co_await as::experimental::make_parallel_group(
      tim1.async_wait(as::append(as::deferred, 10, 20.1)),
      tim2.async_wait(as::append(as::deferred, 30, 40.1, 50))
    ).async_wait(
      as::experimental::wait_for_one(),
      as::deferred
    );
  for (auto order : orders) {
    std::cout << order;
    switch (order) {
    case 0:
      std::cout << " " << ec1.message() << " " << res11 << " " << res12 << std::endl;
      break;
    case 1:
      std::cout << " " << ec2.message() << " " << res21 << " " << res22 << " " << res23 << std::endl;
      break;
    }
  }
  co_return;
}
```

```
1 Success 30 40.1 50
0 Operation canceled 10 20.1
```

make_parallel_group(funcs...)
wait_for_one()

co_awaitとマルチウェイト

https://godbolt.org/z/c6ao6883c

動的

非awaitable

全部待ち

make_parallel_group(vec)
wait_for_all()

```
略
#include <boost/asio/experimental/parallel_group.hpp>
namespace as = boost::asio; as::awaitable<void> test() {
    auto exe = co_await as::this_coro::executor;
    using op_type = decltype(as::post(exe, as::append(as::deferred, int{})));
    std::vector<op_type> ops;
    for (int i = 0; i != 5; ++i) {
        ops.push_back(as::post(exe, as::append(as::deferred, i*10)));
    }
    auto [orders, values] =
        co_await as::experimental::make_parallel_group(
            ops
        ).async_wait(
            as::experimental::wait_for_all(),
            as::deferred
        );
    for (auto order : orders) {
        std::cout << order << " " << values[order] << std::endl;
    }
    co_return;
}
```

```
0 0
1 10
2 20
3 30
4 40
```

co_awaitとマルチウェイト

https://godbolt.org/z/69zWs7M5G

動的

非awaitable

ひとつ待ち

make_parallel_group(vec)
wait_for_one()

```
as::awaitable<void> test() {
    auto exe = co_await as::this_coro::executor;
    using op_type = decltype(
        std::declval<as::steady_timer>().async_wait(as::as_tuple(as::deferred))
    );
    std::vector<op_type> ops;
    std::vector<as::steady_timer> vtim;
    vtim.emplace_back(exe, std::chrono::milliseconds{50});
    vtim.emplace_back(exe, std::chrono::milliseconds{10});
    vtim.emplace_back(exe, std::chrono::milliseconds{30});
    for (auto& tim : vtim) {
        ops.push_back(tim.async_wait(as::as_tuple(as::deferred)));
    }
    auto [orders, values] =
        co_await as::experimental::make_parallel_group(
            ops
        ).async_wait(
            as::experimental::wait_for_one(),
            as::deferred
        );
    for (auto order : orders) {
        auto [ec] = values[order];
        std::cout << order << " " << ec.message() << std::endl;
    }
    co_return;
}
```

```
1 Success
0 Operation canceled
2 Operation canceled
```

co_awaitとマルチウェイト まとめ

要素数確定タイミング	awaitable非同期関数 awaitable	非awaitable非同期関数 非awaitable

コンパイル時
静的
	co_await (... && ...)
	全部待ち
	
	co_await (... || ...)
	ひとつ待ち
		co_await make_parallel_group(
		  func1(deferred),
		  func2(experimental::use_promise)
		).async_wait(
		  experimental::wait_for_all(), 全部待ち
		  // experimental::wait_for_one(), ひとつ待ち
		  deferred
		);

実行時
動的
	調査した限り手段無し	using op_type = decltype(post(exe, append(deferred, int{})));
		std::vector<op_type> ops;
		for (int i = 0; i != 5; ++i)
		  ops.push_back(post(exe, append(as::deferred, i*10)));
		  auto [orders, values] =
		  co_await experimental::make_parallel_group(
		  ops
		  ).async_wait(
		  experimental::wait_for_all(), 全部待ち
		  // experimental::wait_for_one(), ひとつ待ち
		  deferred
		  );

以上。時間があればBonus Slides

co_spawn

- coroutineを起動する

```cpp
#include <iostream>
#include <boost/asio.hpp>

namespace as = boost::asio;

as::awaitable<void> coro_main() {
    std::cout << "coro started" << std::endl;
    co_return;
}

int main() {
    as::io_context ioc;
    as::co_spawn(
        ioc.get_executor(),
        coro_main(), ←awaitableを渡している
        as::detached
    );
    ioc.run();
}
```

https://godbolt.org/z/1dTxc66do

```cpp
#include <iostream>
#include <boost/asio.hpp>

namespace as = boost::asio;

as::awaitable<void> coro_main() {
    std::cout << "coro started" << std::endl;
    co_return;
}

int main() {
    as::io_context ioc;
    as::co_spawn(
        ioc.get_executor(),
        coro_main,  ←callableを渡している
        as::detached
    );
    ioc.run();
}
```

https://godbolt.org/z/bjE4TrP3v

co_spawnの宣言

```
template<
    typename Executor,
    typename T,
    typename AwaitableExecutor,
    typename CompletionToken = DEFAULT>
DEDUCED co_spawn(
    const Executor & ex,
    awaitable< T, AwaitableExecutor > a,
    CompletionToken && token = DEFAULT,
    constraint_t<略>  = 0);
```

```
template<
    typename Executor,
    typename F,
    typename CompletionToken = DEFAULT>
DEDUCED co_spawn(
    const Executor & ex,
    F && f,
    CompletionToken && token = DEFAULT,
    constraint_t<略>  = 0);
```

https://www.boost.org/doc/libs/1_84_0/doc/html/boost_asio/reference/co_spawn/overload1.html

https://www.boost.org/doc/libs/1_84_0/doc/html/boost_asio/reference/co_spawn/overload5.html

CompletionToken指定時の注意

- 第1パラメタがboost::system::error_codeの場合の取り扱い
  - successのときは何も起こらず、その他errorの時、例外がthrowされる
  - これを防ぐには、as_tuple(deferred)のように、as_tupleで囲む
    こうすれば、successであってもerrorであっても、tupleの0番目の
    要素として、 boost::system::error_codeが取得できる

CompletionToken指定時の戻り値の型

```
as::awaitable<void> async_func();
```

```
boost::asio::awaitable<
  void,
  boost::asio::any_io_executor
>
```

```
as::post(as::use_awaitable);
```

```
boost::asio::awaitable<
  void,
  boost::asio::any_io_executor
>
```

```
as::post(as::deferred);
```

```
boost::asio::deferred_async_operation<
  void (),
  boost::asio::detail::initiate_dispatch
>
```

```
as::post(
  as::experimental::use_promise
);
```

```
boost::asio::experimental::promise<
  void (),
  boost::asio::basic_system_executor<
    boost::asio::execution::detail::blocking::possibly_t<0>,
    boost::asio::execution::detail::relationship::fork_t<0>,
    std::allocator<void>
  >,
  std::allocator<void>
>
```

```
boost::typeindex::type_id<
  decltype(as::post(as::deferred))
>().pretty_name()
```

この部分を差し替えて出力

https://godbolt.org/z/W9veEY35v

co_composedとexecutor

- co_composedがどのexecutorで動いているのか

```
template <typename CompletionToken>
auto async_func(
  CompletionToken&& token
) {
  return as::async_initiate<
    CompletionToken,
    void()
  >(
    as::experimental::co_composed<
      void()
    >(
      [](
        auto /*state*/
      ) -> void {
        co_await ここで待てる
        co_return {};
      }
    ),
    token
  );
}
```

https://godbolt.org/z/vs9cGdEeK