# 20分くらいで
# わかった気分になれる†
# C++20コルーチン

2019/9/4
C++MIX #5

†効果には個人差があります。C++20コルーチン 完全理解™ を保証するものではありません。

# はじめに

**誰？**

twitter @yohhoy / hatena id:yohhoy

**何を？**

C++20導入予定の **コルーチン** (Coroutines) 概要を紹介

**どうして？**

勉強会駆動 C++20コルーチン調査

本日のディスカッション ネタ提供

# Question: コルーチン？

並行タスク(Concurrency task)

非同期I/O(Asynchronous I/O)

ジェネレータ(Generator)

エラー伝搬(Error propagation)

# Answer: C++20コルーチン

~~並行タスク(Concurrency task)~~

~~非同期I/O(Asynchronous I/O)~~

~~ジェネレータ(Generator)~~

~~エラー伝搬(Error propagation)~~

# Answer: C++20コルーチン

~~並行タスク(Concurrency task)~~

~~非同期I/~~

~~ジェネレ~~

~~エラー伝搬(Error propagation)~~

ハイレベル機能の
実装に利用する
**低レベル言語仕様** と
**新しい構文** を定義

# C++20コルーチンのユーザは誰？

ライブラリ
プログラマ

並行タスク
ライブラリ

非同期I/O
ライブラリ

ジェネレータ,
etc.

アプリケーション
プログラマ

**コルーチン**
**言語仕様**

**co_await**
**co_yield**
**co_return**

# C++23, C++26, or someday

Parallel Algorithm
Ranges Library
`std::expected`
Executors TS
Networking TS

並行タスク
ライブラリ

非同期I/O
ライブラリ

ジェネレータ,
etc.

アプリケーション
プログラマ

# C++20コルーチン

**標準ヘッダ<coroutine>**

- `coroutine_traits<R,Args...>`
- **`coroutine_handle<Promise>`**
- `suspend_never`
- `suspend_always`
- `noop_coroutine()`

**新キーワード**

- **`co_await`**
- **`co_yield`**
- `co_return`

**カスタマイズポイント**

- **Promise type**
- **Awaitable type**
- `operator co_await`

# C++20コルーチン

**標準ヘッダ<coroutine>**

```
coroutine_traits<R,Args...>
coroutine_handle<Promise>
suspend_never
suspend_always
noop_coroutine()
```

**新キーワード**

```
co_await
co_yield
co_return
```

ライブラリ
プログラマ

**カスタマイズポイント**

**Promise type**
**Awaitable type**
operator co_await

# C++20コルーチン

**標準ヘッダ<coroutine>**

- coroutine_traits<*R*,*Args*...>
- **coroutine_handle<*Promise*>**
- suspend_never
- suspend_always
- noop_

**新キーワード**

- **co_await**
- **co_yield**
- co_return

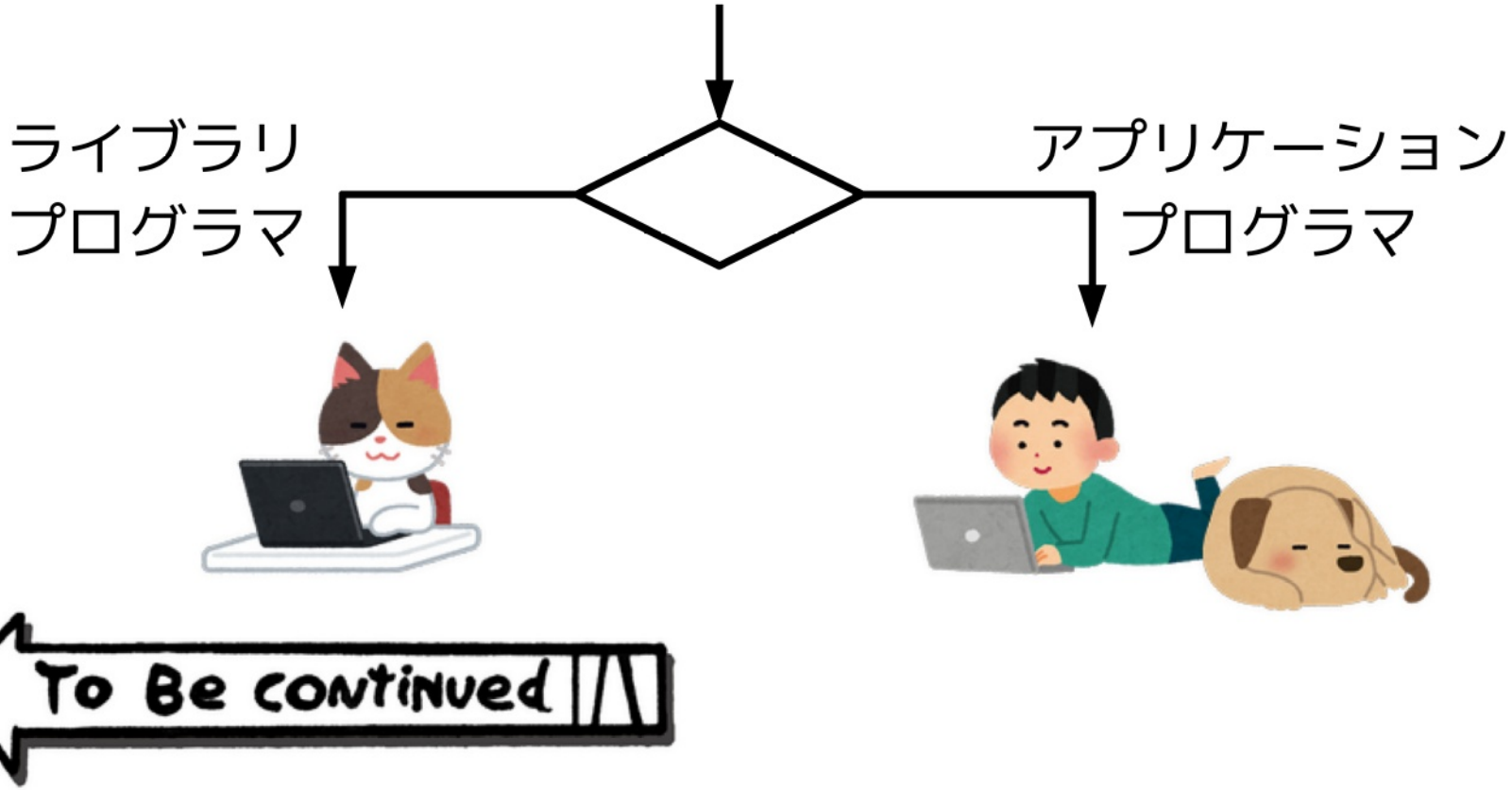
C++20コルーチンを
利用して実装された
ハイレベル・ライブラリ

**カスタマ**

- **Promis**
- **Awaitable type**
- operator co_await

アプリケーション
プログラマ

# If you are



# 関数とコルーチン

C++コルーチンは関数を拡張したもの

**関数(*function*)**

呼出(*call*)により処理開始／呼出元へ復帰(*return*)

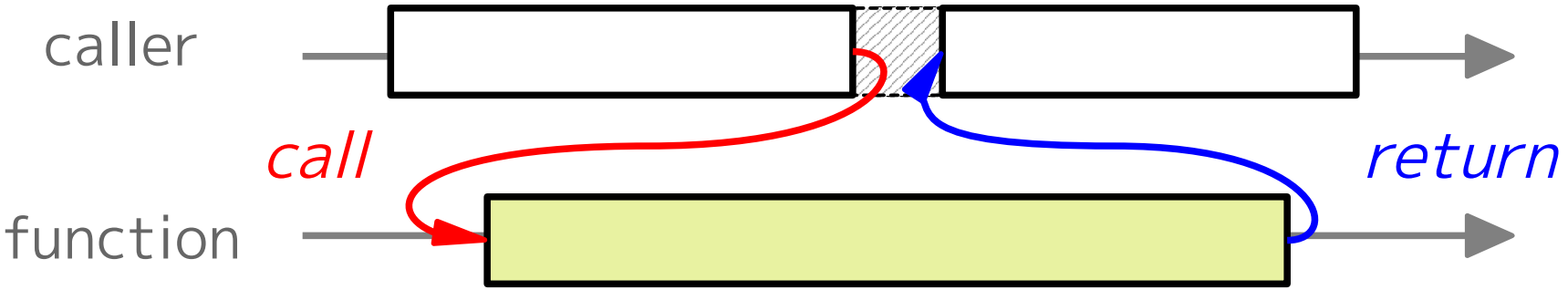
**コルーチン(*coroutine*)**

呼出(*call*)により処理開始／呼出元へ復帰(*return*)

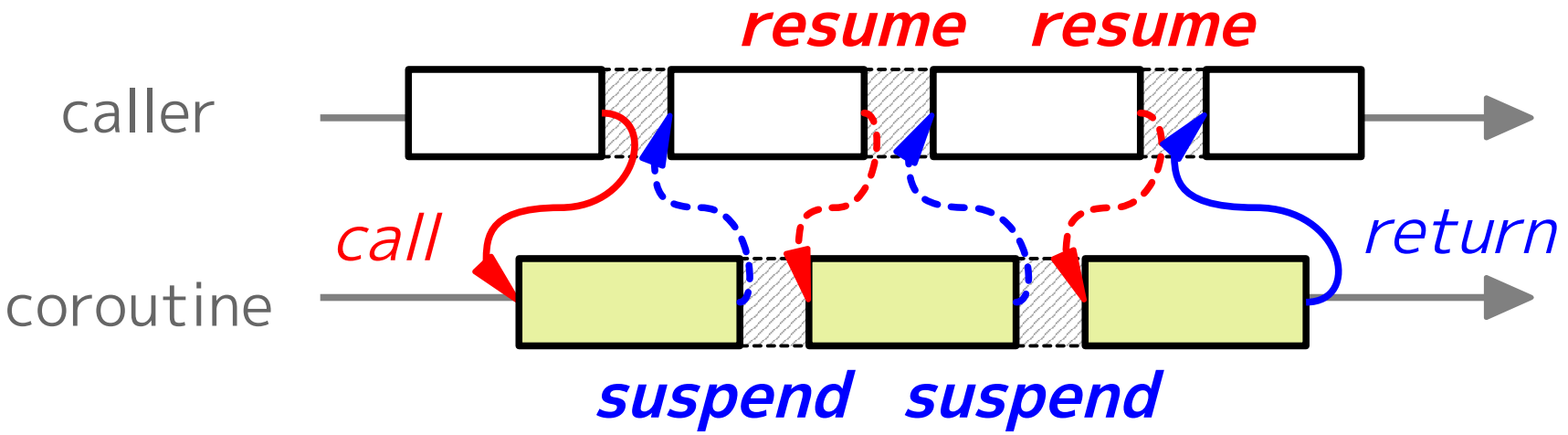
途中での**中断(*suspend*)**と**再開(*resume*)**をサポート

# 関数とコルーチン

## 関数(サブルーチン)

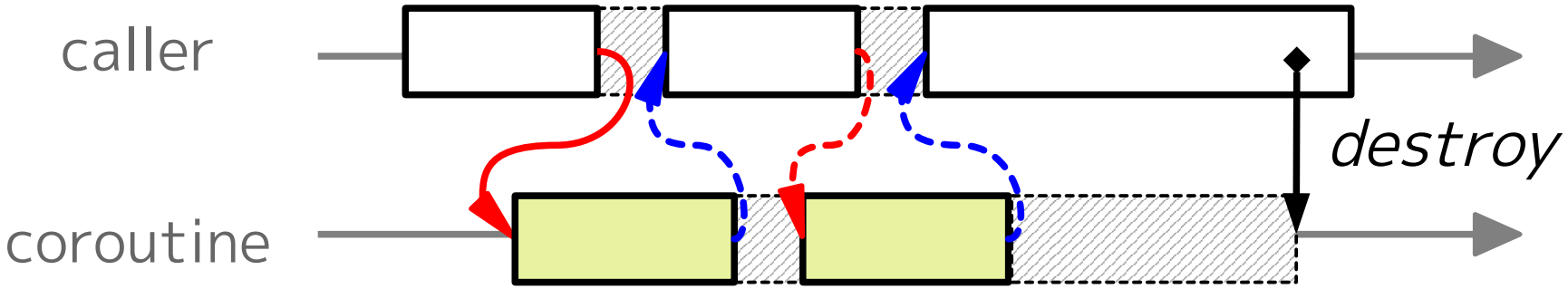



## コルーチン

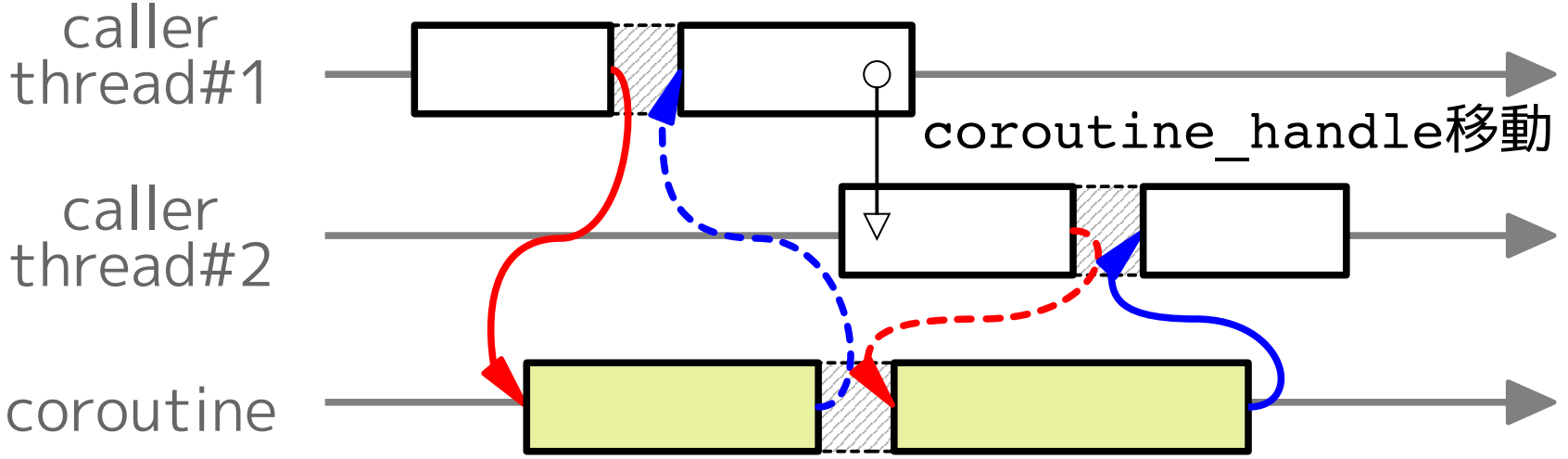


# 関数とコルーチン

## コルーチンを途中破棄




## 別スレッドから再開



# 関数とコルーチン

本体部にコルーチン構文が登場したらコルーチン
関数シグネチャからは区別しない/できない

```
future<int> c1() {
  co_yield 42;
}
```

```
future<int> f1() {
  return make_future(42);
}
```

```
task<void> c2() {
  co_await read();
  co_await write();
}
```

```
task<void> f2() {
  return []{
    read(); write(); };
}
```

**coroutine**

**function**

※上記コード片は説明用のイメージです。

# Asymmetric vs. Symmetric

**非対称(Asymmetric)／対称(Symmetric)コルーチン**の両方をサポートする

**非対称コルーチン**

現コルーチンを中断＝再開元(*resumer*)に戻る

大抵のケースでは非対称コルーチンが使われる

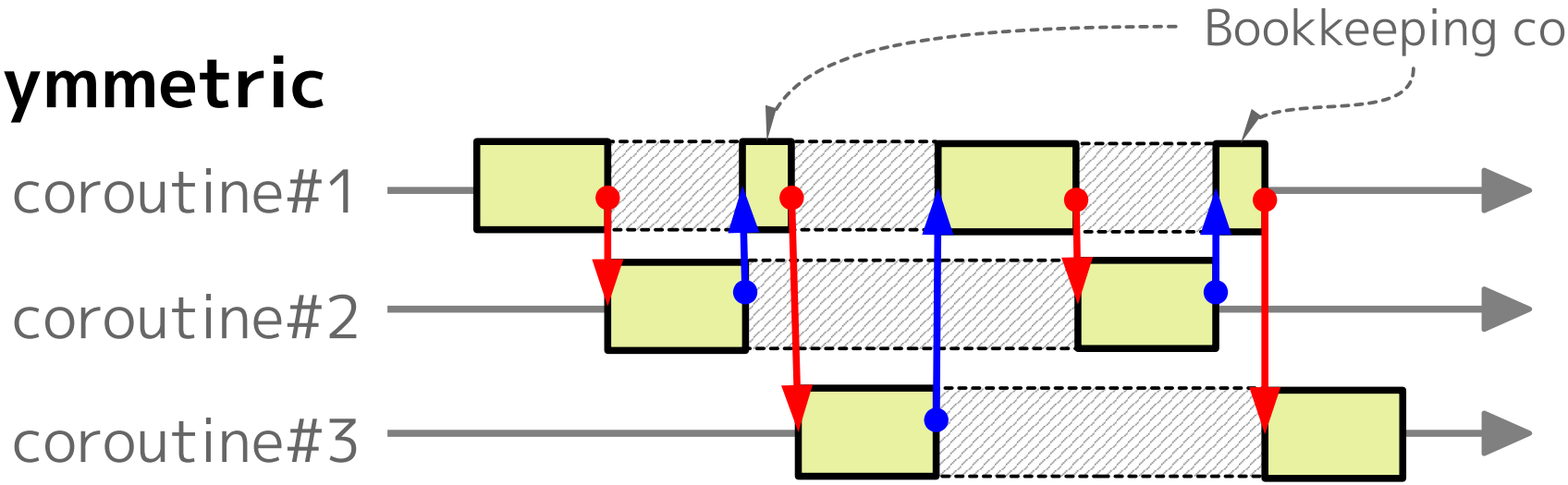
**対称コルーチン**

任意のコルーチンへと制御を移す(control transfer)

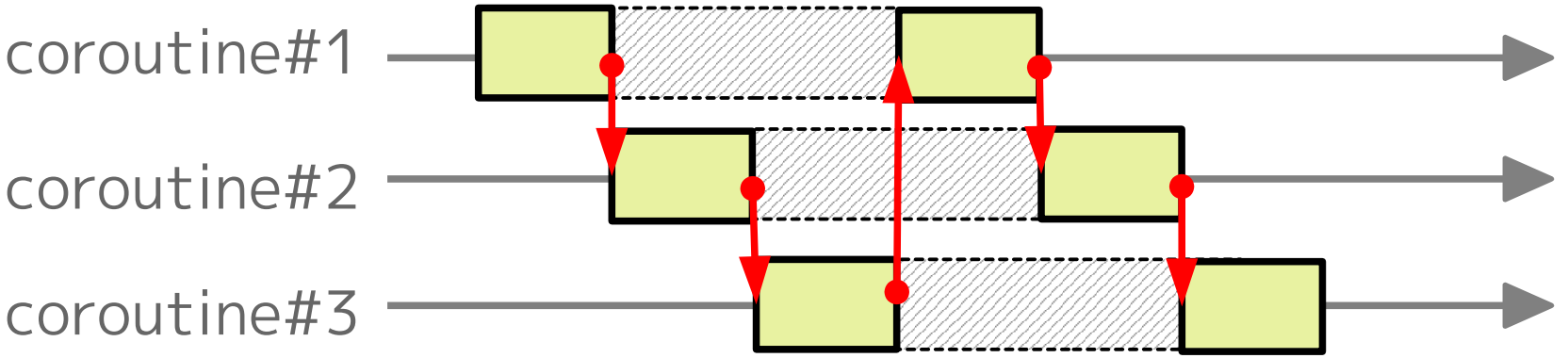
相手コルーチンを”知っている”ケースでのみ利用

# Asymmetric vs. Symmetric

## Asymmetric




## Symmetric



# Stackless vs. Stackful

C++コルーチンは**スタックレス(Stackless)**

- コルーチンでのみ **`co_await/co_yield`** を利用可能
- ネストしたラムダ式や呼出先の関数では利用不可
- 中断/再開用のコールスタック保持不要のため軽量

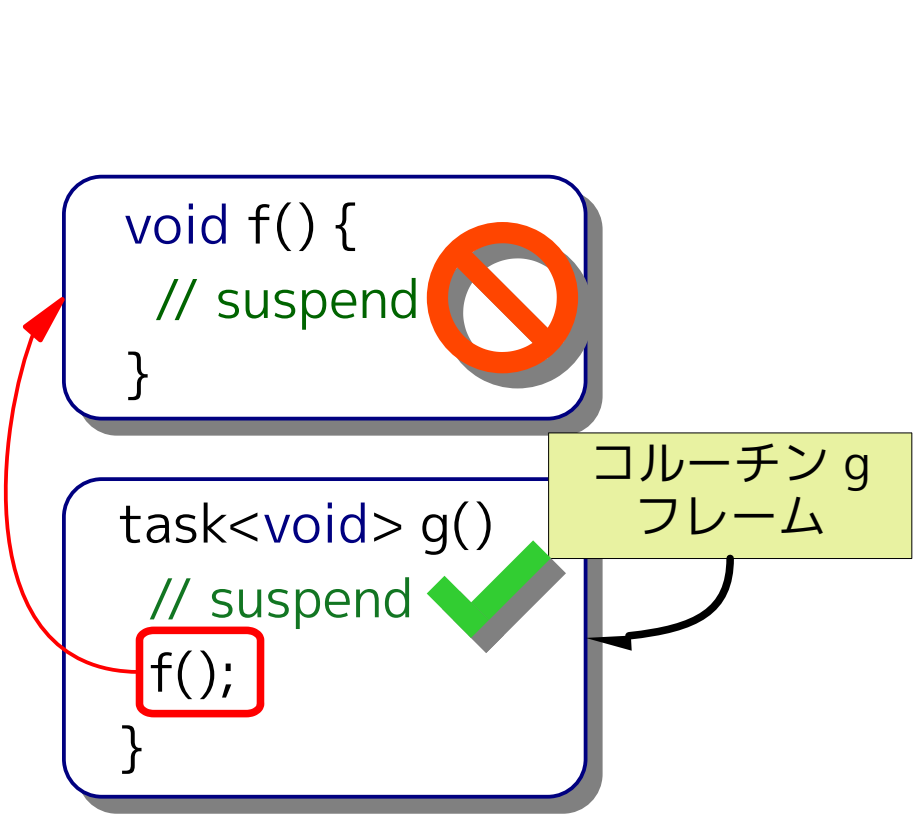
【参考：スタックフル(Stackful)コルーチン】

任意のタイミングでコルーチン中断/再開が可能

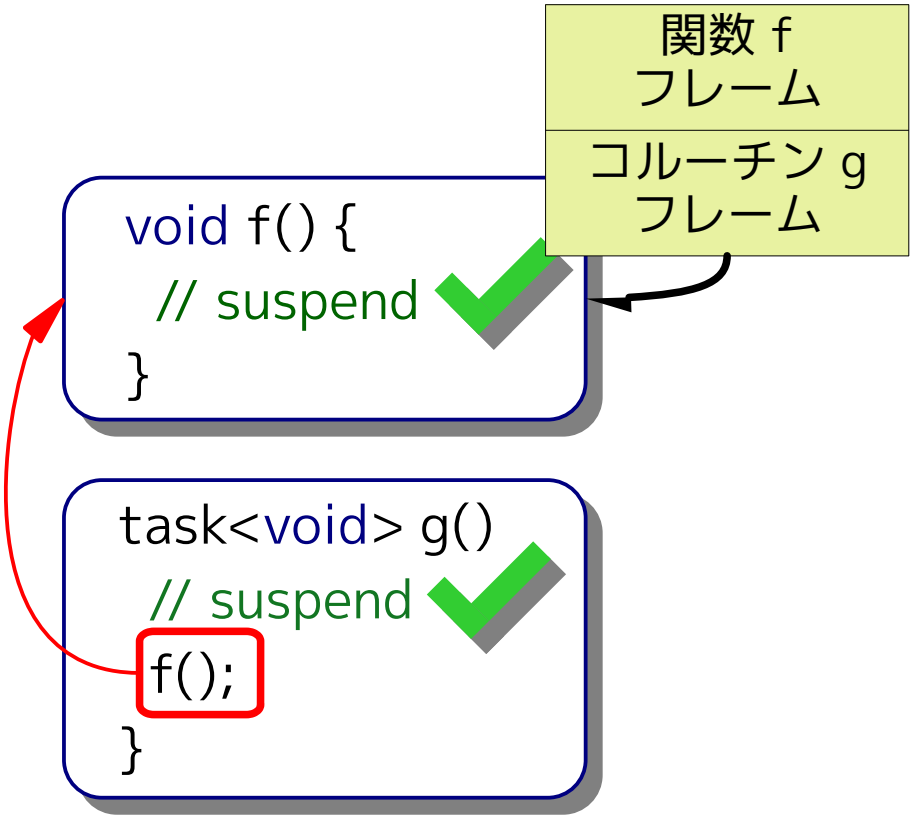
Go言語のgoroutineや協調的ファイバ(Fiber)など

中断/再開用のコールスタックを保持するため高コスト

# Stackless vs. Stackful




**Stackless**
(C++コルーチン)




**Stackful**

# C++20コルーチン (再掲)

**標準ヘッダ<coroutine>**

- `coroutine_traits<R,Args...>`
- **`coroutine_handle<Promise>`**
- `suspend_never`
- `suspend_always`
- `noop_coroutine()`

**新キーワード**

- **`co_await`**
- **`co_yield`**
- `co_return`

**カスタマイズポイント**

- **Promise type**
- **Awaitable type**
- `operator co_await`

# C++コルーチンの実例
## ～ アプリケーションプログラマ視点 ～

```
generator iota(int end)
{
  for (int n = 0; n < end; ++n)
    co_yield n;
}

int main()
{
  auto g = iota(10);
  for (auto e : g)
    std::cout << e << " ";
}
```

```
$ clang++ src.cpp -std=c++2a
$ ./a.out
0 1 2 3 4 5 6 7 8 9
```

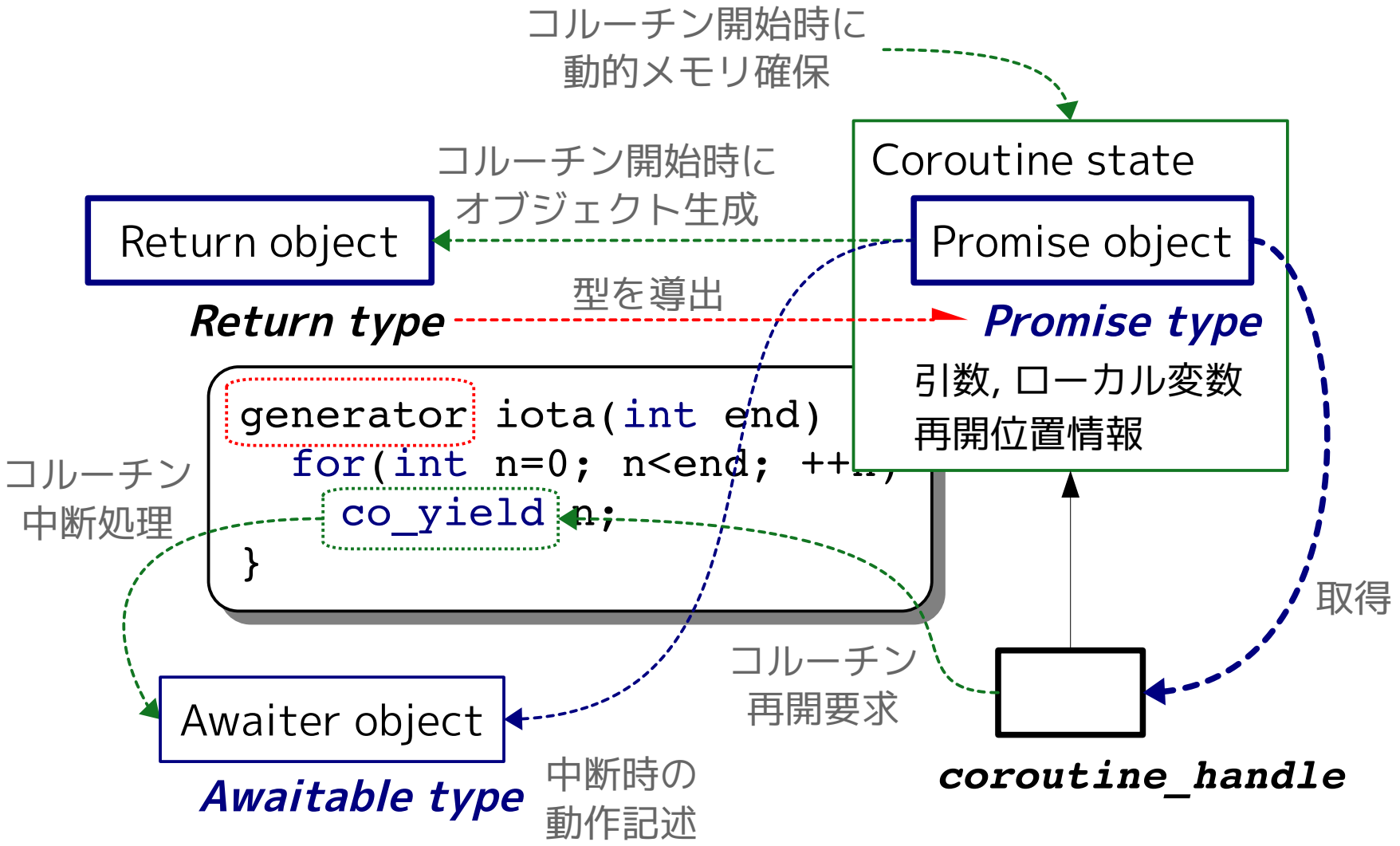
# C++コルーチンの実例
## ～ ライブラリプログラマ視点 ～

```
struct generator {
  struct promise_type {
    int value_;
    auto get_return_object()
      { return generator{*this}; }
    auto initial_suspend()
      { return std::suspend_always{}; }
    auto final_suspend()
      { return std::suspend_always{}; }
    auto yield_value(int v)
    {
      value_ = v;
      return std::suspend_always{};
    }
    void return_void() {}
    void unhandled_exception()
      { throw; }
  };
  using coro_handle =
    std::coroutine_handle<promise_type>;

  struct iterator {
    coro_handle coro_;
    bool done_;
    iterator& operator++()
    {
      coro_.resume();
      done_ = coro_.done();
      return *this;
    }
    bool operator!=(const iterator& rhs) const
      { return done_ != rhs.done_; }
    int operator*() const
      { return coro_.promise().value_; }
  };
  // (cont.)
```

```
  // (cont.)
  ~generator()
  {
    if (coro_)
      coro_.destroy();
  }

  generator(generator const&) = delete;
  generator(generator&& rhs)
    : coro_{std::exchange(rhs.coro_, nullptr)}
  {}

  iterator begin()
  {
    coro_.resume();
    return {coro_, coro_.done()};
  }
  iterator end()
    { return {{}, true}; }

private:
  explicit generator(promise_type& p)
    : coro_{coro_handle::from_promise(p)}
  {}

  coro_handle coro_;
};
```

https://wandbox.org/permlink/IpSC8my6j0A1gT0I

# C++コルーチン At-a-Glance



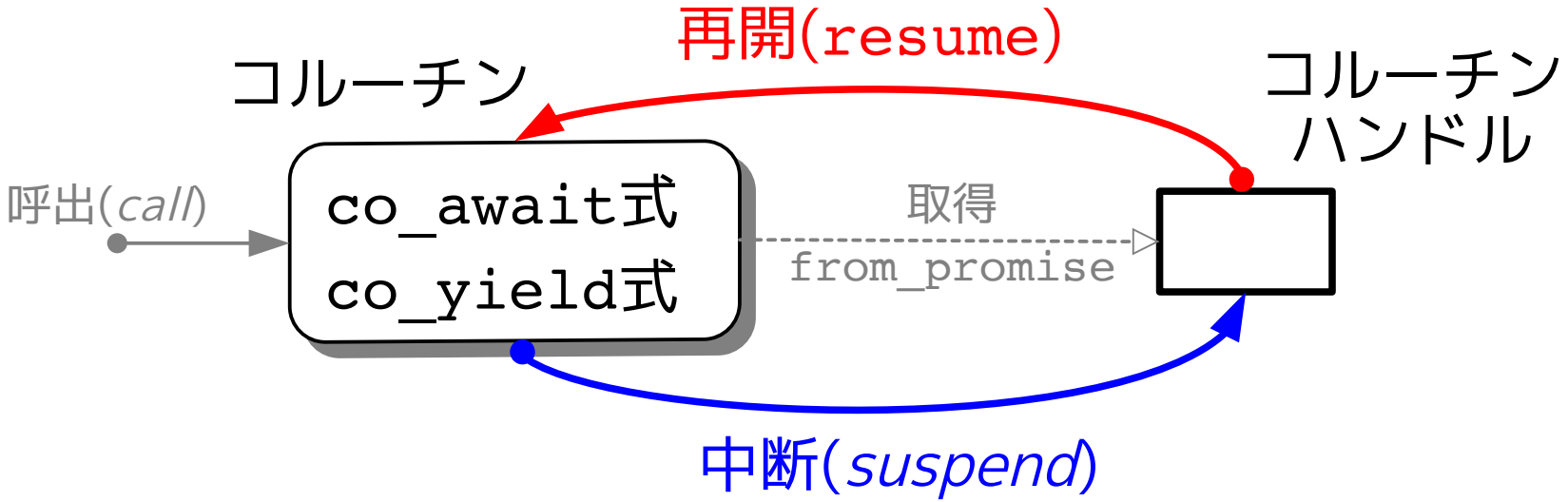
# コルーチンハンドル

## `std::coroutine_handle<Promise>`

コルーチン再開用の**ハンドル型**

**Promise object経由**でハンドルを取得する

コルーチンの実行に対してハンドルが1:1対応



# Promise

コルーチン動作仕様を記述する**ユーザ定義クラス**

- Promise typeはコルーチン**戻り値型から導出**
- Promise objectはコルーチン実行と1:1対応
- 動作の**カスタマイズポイント**を規定

```
(constructor)                 initial_suspend
get_return_object             final_suspend
yield_value                   return_value
await_transform               return_void
unhandled_exception
operator new/delete
get_return_object_on_allocation_failure
```

# Awaitable

中断/再開の動作仕様を記述する**ユーザ定義クラス**

コルーチン本体の`co_await`/`co_yield`動作を制御

コルーチン開始直後/終了直前の動作を制御

動作の**カスタマイズポイント**を規定

```
await_ready
await_suspend
await_resume
```

<coroutine>提供クラス

**Trivial Awaitable**

```
std::always_suspend
std::never_suspend
```

# その他

std::coroutine_traitsトレイツ特殊化
コルーチン引数とPromise typeコンストラクタ
co_yield式とyield_value
co_return式とreturn_value/return_void
co_await演算子オーバーロード
unhandled_exceptionと例外処理
型消去(Type Erasure)されたstd::coroutine_handle<>
std::coroutine_handleとvoidポインタ相互運用
ヒープメモリ確保省略最適化(“Halo”)
…

# C++20コルーチン対応状況

## コンパイラサポート状況

Clang, MSVC　：部分的にサポート

GCC, etc.　　：未対応

## 対応ライブラリ

**CppCoro - A coroutine library for C++**

https://github.com/lewissbaker/cppcoro

**Boost.ASIO** [Experimental]

https://github.com/boostorg/asio

**Facebook Folly** [Experimental]

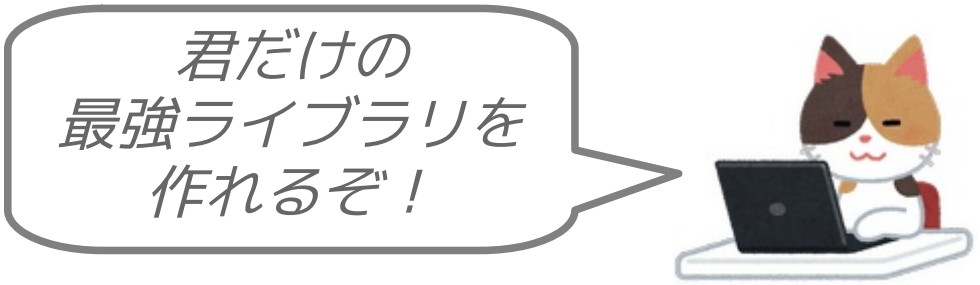
https://github.com/facebook/folly

# まとめ

C++20コルーチンは

- 処理の**中断／再開をサポート**する関数
- 軽量な**スタックレス**コルーチン
- ライブラリ実装用の**低レベル部品のみ提供**
- 多数の**カスタマイズポイント**を規定

C++20以後のコルーチン・ライブラリ発展に期待



# [おまけ] More information

**Working Draft, Standard for Programming Language C++**
http://eel.is/c++draft/

**Halo: coroutine Heap Allocation eLision Optimization**
https://wg21.link/p0981r0

**Impact of coroutines on current and upcoming library facilities**
https://wg21.link/p0975r0

**Coroutine Theory**
https://lewissbaker.github.io/2017/09/25/coroutine-theory

**How C++ coroutines work**
https://kirit.com/How%20C%2B%2B%20coroutines%20work

**luncliff/coroutine**
https://luncliff.github.io/coroutine/Home/

**C++コルーチン拡張メモ**
https://qiita.com/yohhoy/items/aeb3c01d02d0f640c067

# [おまけ] Stackless vs. Stackful

## Stackful Coroutines(Fiber)

**`fiber_context` - fibers without scheduler [Oliver K., Nat G.]**

https://wg21.link/p0876r8

**Fibers under the magnifying glass [Gor N.]**

https://wg21.link/p1364r0

**Response to “Fibers under the magnifying glass” [Nat G., Oliver K.]**

https://wg21.link/p0866r0

**Response to response to “Fibers under the magnifying glass” [Gor N.]**

https://wg21.link/p1520r0

Gorさん = C++20に統合されたCoroutines TS提案者
Oliverさん = Boost.Coroutine, Coroutine2, Fiber, Contextライブラリ作者