Делаем стейт машины из [redacted] шаблонов и корутин

Павел Новиков

@cpp_ape

R&D Align Technology

Термин «стейт машина» в этом докладе

finite-state machine (FSM)

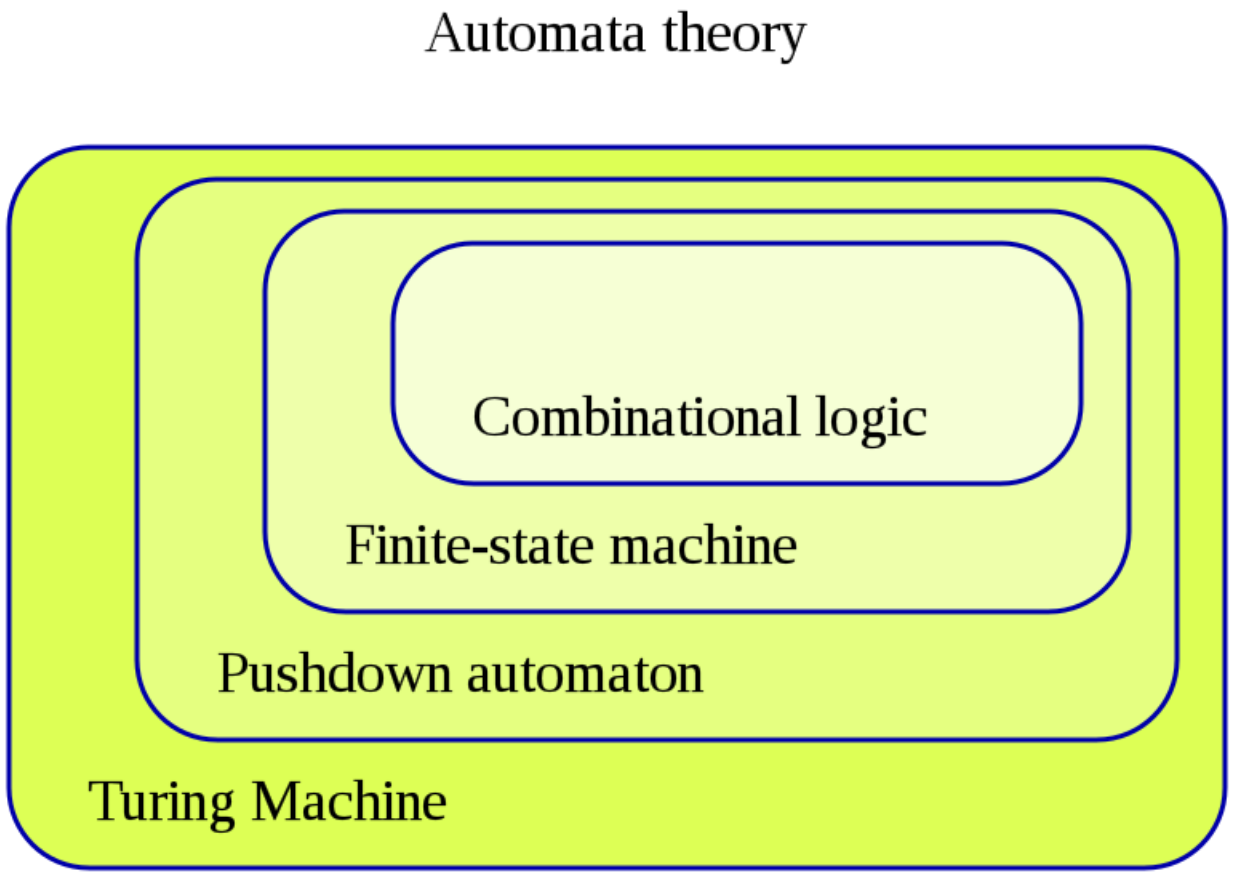
или

finite-state automaton (FSA)

или

конечный автомат

Абстрактная машина, которая может находится лишь в одном состоянии из конечного набора в любой конкретный момент времени.



Как приготовить ~~стейк~~ стейт машину

- из шаблонов
- из корутин

По дороге увидим практическое применение

- `if constexpr`
- `std::variant`
- `std::any` и type erasure

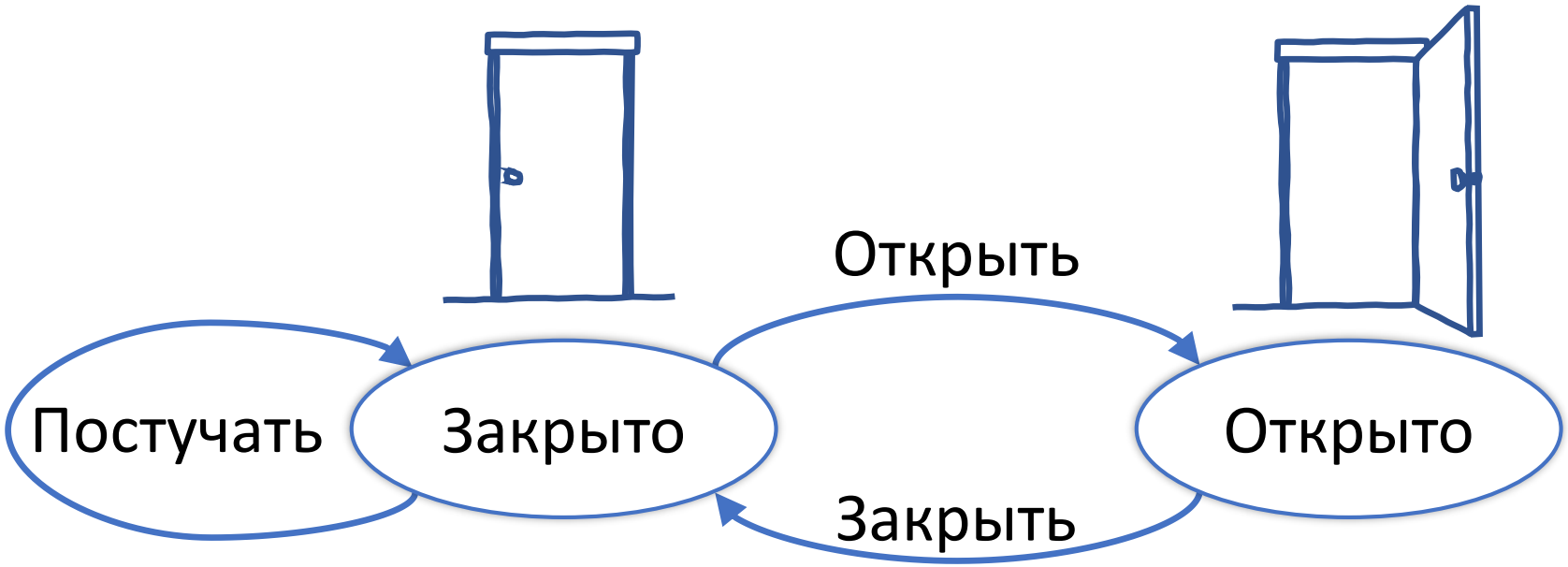
Но сначала...

События:

```
struct Open {};
struct Close {};
struct Knock {};
```

Состояние:

```
enum class State {
  Closed,
  Open
};
```



Но сначала...

События:

```
struct Open {};
struct Close {};
struct Knock {};
```

Состояние:

```
enum class State {
  Closed,
  Open
};

struct Door {
  State state = State::Closed;

  template<typename E>
  void onEvent(E);
};
```

Но сначала...

```
template<typename E>
void onEvent(E) {
  switch (state) {
  case State::Closed:
    if constexpr (std::is_same_v<E, Knock>) {
      shout("Come in, it's open!");  // нет перехода
    }
    else if constexpr (std::is_same_v<E, Open>) {
      state = State::Open;  // переход в состояние Open
    }
    break;
  case State::Open:
    if constexpr (std::is_same_v<E, Close>)
      state = State::Closed;  // переход в состояние Closed
  }
}
```

Использование

```
Door door;
door.onEvent(Open{});  // Closed -> Open
door.onEvent(Close{}); // Open -> Closed
door.onEvent(Knock{});
door.onEvent(Close{}); // Closed -> Closed
```

выводит:

```
Come in, it's open!
```


Стейт машина из шаблонов

Стейт машина из шаблонов

```
auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout("Come in, it's open!");
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Стейт машина из шаблонов

```
auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout("Come in, it's open!");
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Стейт машина из шаблонов

```
auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout("Come in, it's open!");
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Стейт машина из шаблонов

```
auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout("Come in, it's open!");
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Стейт машина из шаблонов

```
auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout("Come in, it's open!");
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Стейт машина из шаблонов

```
auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout("Come in, it's open!");
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Стейт машина из шаблонов

```
auto getDoor() {
  auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
    [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
      return {};
    },
    [](StateClosed &s, Knock event) /*нет перехода*/ {
      shout("Come in, it's open!");
    },
    [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
      return {};
    }
  );
  return sm;
}
```

Стейт машина из шаблонов

```
template<typename S, typename... Handlers>
struct StateMachine;

template<typename S>
struct TransitionTo {
  using TargetState = S;
};
```

Стейт машина из шаблонов

```
template<typename... States, typename... Handlers>
struct StateMachine<std::tuple<States...>, Handlers...> : Handlers...  {
  using Handlers::operator()...;

  template<typename... H>
  StateMachine(H&&...h) : Handlers(h)... {}

  template<typename E>
  void onEvent(E &&e);

  std::tuple<States...> states;
  std::variant<States*...> currentState = &std::get<0>(states);
};
```

Стейт машина из шаблонов

```
template<typename... States, typename... Handlers>
struct StateMachine<std::tuple<States...>, Handlers...> : Handlers...  {
  using Handlers::operator()...;

  template<typename... H>
  StateMachine(H&&...h) : Handlers(h)... {}

            template<typename... Ts>
            struct overloaded : Ts... {
              using Ts::operator()...;
            }
};
```

Как работает `overloaded`

```
template<typename... Ts>
struct overloaded : Ts... {
  using Ts::operator()...;
}
template<typename... Ts>
overloaded(Ts...)->overloaded<Ts...>;  // не нужен в C++20
```

Как работает overloaded

```
template<typename... Ts>
struct overloaded : Ts... {
  using Ts::operator()...;
}
template<typename... Ts>
overloaded(Ts...)->overloaded<Ts...>;  // не нужен в C++20

auto func = overloaded{
  [](int i) {
    std::cout << "int: " << i << '\n';
  },
  [](std::string s) {
    std::cout << "string: " << s << '\n';
  }
};
```

Как работает overloaded

```
struct lambda1 {
  inline /*constexpr */ void operator()(int i) const
  {
    std::cout << "int: " << i << '\n';
  }
};

struct lambda2 {
  inline /*constexpr */ void operator()(std::string s) const
  {
    std::cout << "string: " << s << '\n';
  }
};

overloaded<lambda1, lambda2> func = overloaded{ lambda1{}, lambda2{} };
```

Как работает `overloaded`

```
auto func = overloaded{
  [](int i) {
    std::cout << "int: " << i << '\n';
  },
  [](std::string s) {
    std::cout << "string: " << s << '\n';
  }
};
```

Как работает overloaded

```
overloaded<lambda1, lambda2> func = overloaded{
                                    [](int i) {
                                      std::cout << "int: " << i << '\n';
                                    },
                                    [](std::string s) {
                                      std::cout << "string: " << s << '\n';
                                    }
                                  };
```

Как работает overloaded

```
overloaded<lambda1, lambda2> func = overloaded{
                                    lambda1{},


                                    lambda2{}


                               };
```

Как работает `overloaded`

```
template<typename... Ts>
struct overloaded : Ts... {
  using Ts::operator()...;
}
```

```
template<>
struct overloaded<lambda1, lambda2> : lambda1, lambda2 {
  using lambda1::operator();
  using lambda2::operator();
};
```

Как работает overloaded

```
template<typename... Ts>
struct overloaded : Ts... {
  using Ts::operator()...;
}
```

```
template<>
struct overloaded<lambda1, lambda2> : lambda1, lambda2 {
  using lambda1::operator();
  using lambda2::operator();
};

overloaded<lambda1, lambda2> func = overloaded{ lambda1{}, lambda2{} };
func(42);
func("Hello, C++ Siberia 2021!");
```

Как работает overloaded

```
template<typename... Ts>
struct overloaded : Ts... {
  using Ts::operator()...;
}
```

```
template<>
struct overloaded<lambda1, lambda2> : lambda1, lambda2 {
  using lambda1::operator();
  using lambda2::operator();
};

overloaded<lambda1, lambda2> func = overloaded{ lambda1{}, lambda2{} };
func(42);
func("Hello, C++ Siberia 2021!");
```

Как работает overloaded

```
template<typename... Ts>
struct overloaded : Ts... {
  using Ts::operator()...;
}
```

```
template<>
struct overloaded<lambda1, lambda2> : lambda1, lambda2 {
  using lambda1::operator();
  using lambda2::operator();
};

overloaded<lambda1, lambda2> func = overloaded{ lambda1{}, lambda2{} };
func(42);
func("Hello, C++ Siberia 2021!");
```

Стейт машина из шаблонов

```
template<typename... States, typename... Handlers>
struct StateMachine<std::tuple<States...>, Handlers...> : Handlers...  {
  using Handlers::operator()...;

  template<typename... H>
  StateMachine(H&&...h) : Handlers(h)... {}

  template<typename E>
  void onEvent(E &&e);

  std::tuple<States...> states;
  std::variant<States*...> currentState = &std::get<0>(states);
};
```

Стейт машина из шаблонов

```
template<typename... States, typename... Handlers>
struct StateMachine<std::tuple<States...>, Handlers...> : Handlers...  {
  using Handlers::operator()...;

  template<typename... H>
  StateMachine(H&&...h) : Handlers(h)... {}

  template<typename E>
  void onEvent(E &&e);

  std::tuple<States...> states;
  std::variant<States*...> currentState = &std::get<0>(states);
};
```

Стейт машина из шаблонов

```
template<typename... States, typename... Handlers>
struct StateMachine<std::tuple<States...>, Handlers...> : Handlers...  {
  using Handlers::operator()...;

  template<typename... H>
  StateMachine(H&&...h) : Handlers(h)... {}

  template<typename E>
  void onEvent(E &&e);

  std::tuple<States...> states;
  std::variant<States*...> currentState = &std::get<0>(states);
};
```

Стейт машина из шаблонов

```
template<typename... States, typename... Handlers>
struct StateMachine<std::tuple<States...>, Handlers...> : Handlers...  {
  using Handlers::operator()...;

  template<typename... H>
  StateMachine(H&&...h) : Handlers(h)... {}

  template<typename E>
  void onEvent(E &&e);

  std::tuple<States...> states;
  std::variant<States*...> currentState = &std::get<0>(states);
};
```

```
template<typename E>
void onEvent(E &&e) {
  std::visit([this, &e](auto statePtr) {
    if constexpr (std::is_invocable_v<StateMachine, decltype(*statePtr), E &&>) {
      using ResultType = std::invoke_result_t<StateMachine, decltype(*statePtr), E &&>;
      if constexpr (std::is_same_v<ResultType, void>) {
        (*this)(*statePtr, std::forward<E>(e));
        std::cout << "(no transition)\n";
      }
      else {
        auto transitionTo = (*this)(*statePtr, std::forward<E>(e));
        currentState = &std::get<typename ResultType::TargetState>(states);
        std::cout << "(transitioned to " << currentState.index() << ")\n";
      }
    }
    else {
      std::cout << "(no rules invoked)\n";
    }
  }, currentState);
}
```

```
template<typename E>
void onEvent(E &&e) {
  std::visit([this, &e](auto statePtr) {
    if constexpr (std::is_invocable_v<StateMachine, decltype(*statePtr), E &&>) {
      using ResultType = std::invoke_result_t<StateMachine, decltype(*statePtr), E &&>;
      if constexpr (std::is_same_v<ResultType, void>) {
        (*this)(*statePtr, std::forward<E>(e));
        std::cout << "(no transition)\n";
      }
      else {
        auto transitionTo = (*this)(*statePtr, std::forward<E>(e));
        currentState = &std::get<typename ResultType::TargetState>(states);
        std::cout << "(transitioned to " << currentState.index() << ")\n";
      }
    }
    else {
      std::cout << "(no rules invoked)\n";
    }
  }, currentState);
}
```

```
template<typename E>
void onEvent(E &&e) {
  std::visit([this, &e](auto statePtr) {
    if constexpr (std::is_invocable_v<StateMachine, decltype(*statePtr), E &&>) {
      using ResultType = std::invoke_result_t<StateMachine, decltype(*statePtr), E &&>;
      if constexpr (std::is_same_v<ResultType, void>) {
        (*this)(*statePtr, std::forward<E>(e));
        std::cout << "(no transition)\n";
      }
      else {
        auto transitionTo = (*this)(*statePtr, std::forward<E>(e));
        currentState = &std::get<typename ResultType::TargetState>(states);
        std::cout << "(transitioned to " << currentState.index() << ")\n";
      }
    }
    else {
      std::cout << "(no rules invoked)\n";
    }
  }, currentState);
}
```

если можем позвать, т.е. если есть правило

иначе ничего не делаем

```
template<typename E>
void onEvent(E &&e) {
  std::visit([this, &e](auto statePtr) {
    if constexpr (std::is_invocable_v<StateMachine, decltype(*statePtr), E &&>) {
      using ResultType = std::invoke_result_t<StateMachine, decltype(*statePtr), E &&>;
      if constexpr (std::is_same_v<ResultType, void>) {
        (*this)(*statePtr, std::forward<E>(e));
        std::cout << "(no transition)\n";
      }
      else {
        auto transitionTo = (*this)(*statePtr, std::forward<E>(e));
        currentState = &std::get<typename ResultType::TargetState>(states);
        std::cout << "(transitioned to " << currentState.index() << ")\n";
      }
    }
    else {
      std::cout << "(no rules invoked)\n";
    }
  }, currentState);
}
```

если перехода нет

```
template<typename E>
void onEvent(E &&e) {
  std::visit([this, &e](auto statePtr) {
    if constexpr (std::is_invocable_v<StateMachine, decltype(*statePtr), E &&>) {
      using ResultType = std::invoke_result_t<StateMachine, decltype(*statePtr), E &&>;
      if constexpr (std::is_same_v<ResultType, void>) {
        (*this)(*statePtr, std::forward<E>(e));
        std::cout << "(no transition)\n";
      }
      else {
        auto transitionTo = (*this)(*statePtr, std::forward<E>(e));
        currentState = &std::get<typename ResultType::TargetState>(states);
        std::cout << "(transitioned to " << currentState.index() << ")\n";
      }
    }
    else {
      std::cout << "(no rules invoked)\n";
    }
  }, currentState);
}
```

иначе переход

```
template<typename E>
void onEvent(E &&e) {
  std::visit([this, &e](auto statePtr) {
    if constexpr (std::is_invocable_v<StateMachine, decltype(*statePtr), E &&>) {
      using ResultType = std::invoke_result_t<StateMachine, decltype(*statePtr), E &&>;
      if constexpr (std::is_same_v<ResultType, void>) {
        (*this)(*statePtr, std::forward<E>(e));
        std::cout << "(no transition)\n";
      }
      else {
        auto transitionTo = (*this)(*statePtr, std::forward<E>(e));
        currentState = &std::get<typename ResultType::TargetState>(states);
        std::cout << "(transitioned to " << currentState.index() << ")\n";
      }
    }
    else {
      std::cout << "(no rules invoked)\n";
    }
  }, currentState);
}
```

Стейт машина из шаблонов

```
template<typename S, typename... Handlers>
auto makeStateMachine(Handlers&&...h) {
  return StateMachine<S, std::decay_t<Handlers>...>{
    std::forward<Handlers>(h)...
  };
}
```

Стейт машина из шаблонов

```
struct StateOpen {};
struct StateClosed {};

auto getDoor() {
  auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
    [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
      return {};
    },
    [](StateClosed &s, Knock event) /*нет перехода*/ {
      shout("Come in, it's open!");
    },
    [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
      return {};
    }
  );
  return sm;
}
```

Использование

```
auto door = getDoor();
door.onEvent(Open{});  // Closed -> Open
door.onEvent(Close{}); // Open -> Closed
door.onEvent(Knock{});
door.onEvent(Close{}); // Closed -> Closed
```

выводит:

(transitioned to 1)

(transitioned to 0)

Come in, it's open!

(no transition)

(no rules invoked)

Сравним что получилось

switch

- императивное описание

```
switch (state) {
case State::Closed:
  //...
  else if constexpr (std::is_same_v<E, Open>) {
    state = State::Open;
  }
  break;
```

«шаблоны»

- декларативное описание

```
[](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
  return {};
},
[](StateClosed &s, Knock event) /*нет перехода*/ {
  shout("Come in, it's open!");
},
```

Сравним что получилось

`switch`

- императивное описание
- данные не изолированы

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний

Изоляция данных

```
struct Door {
  State state = State::Closed;
  std::string response = "Come in, it's open!";

  template<typename E>
  void onEvent(E) {
    switch (state) {
    case State::Closed:
      if constexpr (std::is_same_v<E, Knock>) {
        shout(response);
      }
      else if constexpr (std::is_same_v<E, Open>) {
        state = State::Open;
      }
      break;
    //...
```

Изоляция данных

```
struct StateOpen {};
struct StateClosed {
  std::string response = "Come in, it's open!";
};

auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout(s.response);
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Изоляция и разделение данных

```
auto myResponse = "Nobody's home. Come later.";

auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [myResponse](StateClosed &s, Knock event) /*нет перехода*/ {
    shout(myResponse);
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Сравним что получилось

`switch`

- императивное описание
- данные не изолированы

- трудно добавлять состояния

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний

- легко добавлять состояния

Сравним что получилось

`switch`

- императивное описание
- данные не изолированы

- трудно добавлять состояния

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний

- легко добавлять состояния

новое состояние: **заперто**

новые события:

```
struct Lock {};
struct Unlock {};
```

Добавление состояний

```
enum class State {
  Closed,
  Open,
  Locked
};
```

```
template<typename E>
void onEvent(E) {
  switch (state) {
  case State::Closed:
    if constexpr (std::is_same_v<E, Knock>) {
      shout("Come in, it's open!");  // нет перехода
    }
    else if constexpr (std::is_same_v<E, Open>) {
      state = State::Open;  // переход в состояние Open
    }



    break;
  case State::Open:
    if constexpr (std::is_same_v<E, Close>)
      state = State::Closed;  // переход в состояние Closed




  }
}
```

```
template<typename E>
void onEvent(E) {
  switch (state) {
  case State::Closed:
    if constexpr (std::is_same_v<E, Knock>) {
      shout("Come in, it's open!");  // нет перехода
    }
    else if constexpr (std::is_same_v<E, Open>) {
      state = State::Open;  // переход в состояние Open
    }
    else if constexpr (std::is_same_v<E, Lock>) {
      state = State::Locked;  // переход в состояние Locked
    }
    break;
  case State::Open:
    if constexpr (std::is_same_v<E, Close>)
      state = State::Closed;  // переход в состояние Closed
    break;
  case State::Locked:
    if constexpr (std::is_same_v<E, Unlock>)
      state = State::Closed;  // переход в состояние Closed
  }
}
```

Добавление состояний

```
auto sm = makeStateMachine<std::tuple<StateClosed, StateOpen, StateLocked>>(
  [](StateClosed &s, Open event) -> TransitionTo<StateOpen> {
    return {};
  },
  [](StateClosed &s, Knock event) /*нет перехода*/ {
    shout("Come in, it's open!");
  },
  [](StateOpen &s, Close event) -> TransitionTo<StateClosed> {
    return {};
  },
  [](StateClosed &s, Lock event) -> TransitionTo<StateLocked> {
    return {};
  },
  [](StateLocked &s, Unlock event) -> TransitionTo<StateClosed> {
    return {};
  }
);
```

Сравним что получилось

`switch`

- **императивное** описание
- данные **не изолированы**
- **трудно** добавлять состояния

«шаблоны»

- **декларативное** описание
- данные **изолированы** внутри состояний
- **легко** добавлять состояния


Стейт машина из корутин

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
template<typename... Events>
struct Event {};

struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E &&e);

  StateMachine(StateMachine&&) = default;
  StateMachine &operator=(StateMachine&&) = default;
  ~StateMachine() { coro.destroy(); }

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

Стейт машина из корутин

```
template<typename... Events>
struct Event {};
```

```
co_await Event<Open, Knock>{};
```

```
struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E &&e);

  StateMachine(StateMachine&&) = default;
  StateMachine &operator=(StateMachine&&) = default;
  ~StateMachine() { coro.destroy(); }

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

Стейт машина из корутин

```
template<typename... Events>
struct Event {};

struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E &&e);

  StateMachine(StateMachine&&) = default;
  StateMachine &operator=(StateMachine&&) = default;
  ~StateMachine() { coro.destroy(); }

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

нужен для поддержки корутин

Стейт машина из корутин

```
template<typename... Events>
struct Event {};

struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E &&e);

  StateMachine(StateMachine&&) = default;
  StateMachine &operator=(StateMachine&&) = default;
  ~StateMachine() { coro.destroy(); }

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

Стейт машина из корутин

```
template<typename... Events>
struct Event {};

struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E &&e);

  StateMachine(StateMachine&&) = default;
  StateMachine &operator=(StateMachine&&) = default;
  ~StateMachine() { coro.destroy(); }

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

Стейт машина из корутин

```
template<typename... Events>
struct Event {};

struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E &&e);

  StateMachine(StateMachine&&) = default;
  StateMachine &operator=(StateMachine&&) = default;
  ~StateMachine() { coro.destroy(); }

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

Стейт машина из корутин

```
template<typename... Events>
struct Event {};

struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E &&e);

  StateMachine(StateMachine&&) = default;
  StateMachine &operator=(StateMachine&&) = default;
  ~StateMachine() { coro.destroy(); }

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

StateMachine::promise_type

```
struct promise_type {
  using CoroHandle = std::coroutine_handle<promise_type>;
  StateMachine get_return_object() noexcept {
    return { CoroHandle::from_promise(*this) };
  }
  auto initial_suspend() const noexcept { return std::suspend_never{}; }
  auto final_suspend() const noexcept { return std::suspend_always{}; }
  void return_void() noexcept {}
  void unhandled_exception() noexcept {}

  template<typename... E>
  auto await_transform(Event<E...>) noexcept;

  std::any currentEvent;
  bool (*isWantedEvent)(const std::type_info&) = nullptr;
};
```

StateMachine::promise_type

```
struct promise_type {
  using CoroHandle = std::coroutine_handle<promise_type>;
  StateMachine get_return_object() noexcept {
    return { CoroHandle::from_promise(*this) };
  }
  auto initial_suspend() const noexcept { return std::suspend_never{}; }
  auto final_suspend() const noexcept { return std::suspend_always{}; }
  void return_void() noexcept {}
  void unhandled_exception() noexcept {}

  template<typename... E>
  auto await_transform(Event<E...>) noexcept;

  std::any currentEvent;
  bool (*isWantedEvent)(const std::type_info&) = nullptr;
};
```

StateMachine::promise_type

```
struct promise_type {
  using CoroHandle = std::coroutine_handle<promise_type>;
  StateMachine get_return_object() noexcept {
    return { CoroHandle::from_promise(*this) };
  }
  auto initial_suspend() const noexcept { return std::suspend_never{}; }
  auto final_suspend() const noexcept { return std::suspend_always{}; }
  void return_void() noexcept {}
  void unhandled_exception() noexcept {}

  template<typename... E>
  auto await_transform(Event<E...>) noexcept;

  std::any currentEvent;
  bool (*isWantedEvent)(const std::type_info&) = nullptr;
};
```

StateMachine::promise_type

```
struct promise_type {
  using CoroHandle = std::coroutine_handle<promise_type>;
  StateMachine get_return_object() noexcept {
    return { CoroHandle::from_promise(*this) };
  }
  auto initial_suspend() const noexcept { return std::suspend_never{}; }
  auto final_suspend() const noexcept { return std::suspend_always{}; }
  void return_void() noexcept {}
  void unhandled_exception() noexcept {}

  template<typename... E>
  auto await_transform(Event<E...>) noexcept;

  std::any currentEvent;
  bool (*isWantedEvent)(const std::type_info&) = nullptr;
};
```

StateMachine::promise_type

```
struct promise_type {
  using CoroHandle = std::coroutine_handle<promise_type>;
  StateMachine get_return_object() noexcept {
    return { CoroHandle::from_promise(*this) };
  }
  auto initial_suspend() const noexcept { return std::suspend_never{}; }
  auto final_suspend() const noexcept { return std::suspend_always{}; }
  void return_void() noexcept {}
  void unhandled_exception() noexcept {}

  template<typename... E>
  auto await_transform(Event<E...>) noexcept;

  std::any currentEvent;
  bool (*isWantedEvent)(const std::type_info&) = nullptr;
};
```

StateMachine::promise_type

```
struct promise_type {
  using CoroHandle = std::coroutine_handle<promise_type>;
  StateMachine get_return_object() noexcept {
    return { CoroHandle::from_promise(*this) };
  }
  auto initial_suspend() const noexcept { return std::suspend_never{}; }
  auto final_suspend() const noexcept { return std::suspend_always{}; }
  void return_void() noexcept {}
  void unhandled_exception() noexcept {}

  template<typename... E>
  auto await_transform(Event<E...>) noexcept;

  std::any currentEvent;
  bool (*isWantedEvent)(const std::type_info&) = nullptr;
};
```

StateMachine::promise_type::

```
template<typename... E>
auto await_transform(Event<E...>) noexcept {
  isWantedEvent = [](const std::type_info &type)->bool {
    return ((type == typeid(E)) || ...);
  };

  struct Awaitable {
    //...
  };

  return Awaitable{ &currentEvent };
}
```

StateMachine::promise_type::

```
co_await Event<Open, Knock>{};
```

```
template<typename... E>
auto await_transform(Event<E...>) noexcept {
  isWantedEvent = [](const std::type_info &type)->bool {
    return ((type == typeid(E)) || ...);
  };

  struct Awaitable {
    //...
  };

  return Awaitable{ &currentEvent };
}
```

StateMachine::promise_type::

```
co_await Event<Open, Knock>{};
```

```
template<typename... E>
auto await_transform(Event<E...>) noexcept {
  isWantedEvent = [](const std::type_info &type)->bool {
    return ((type == typeid(E)) || ...);
  };

  struct Awaitable {
    //...
  };

  return Awaitable{ &currentEvent };
}
```

StateMachine::promise_type::

```
template<typename... E>
auto await_transform(Event<E...>) noexcept {
  isWantedEvent = [](const std::type_info &type)->bool {
    return ((type == typeid(E)) || ...);
  };

  struct Awaitable {
    //...
  };

  return Awaitable{ &currentEvent };
}
```

StateMachine::promise_type::

```
template<typename... E>
auto await_transform(Event<E...>) noexcept {
  isWantedEvent = [](const std::type_info &type)->bool {
    return ((type == typeid(E)) || ...);
  };

  struct Awaitable {
    //...
  };

  return Awaitable{ &currentEvent };
}
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::promise_type::

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  auto await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
      (event = std::move(*std::any_cast<E>(currentEvent)), true) :
      false
      ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

StateMachine::

```
template<typename E>
void onEvent(E &&e) {
  auto &promise = coro.promise();
  if (promise.isWantedEvent(typeid(E))) {
    promise.currentEvent = std::forward<E>(e);
    coro();
  }
}
```

StateMachine::

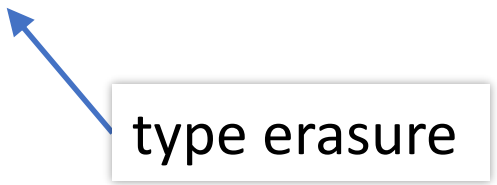
```
template<typename E>
void onEvent(E &&e) {
  auto &promise = coro.promise();
  if (promise.isWantedEvent(typeid(E))) {
    promise.currentEvent = std::forward<E>(e);
    coro();
  }
}
```

StateMachine::

```
template<typename E>
void onEvent(E &&e) {
  auto &promise = coro.promise();
  if (promise.isWantedEvent(typeid(E))) {
    promise.currentEvent = std::forward<E>(e);
    coro();
  }
}
```

StateMachine::

```
template<typename E>
void onEvent(E &&e) {
  auto &promise = coro.promise();
  if (promise.isWantedEvent(typeid(E))) {
    promise.currentEvent = std::forward<E>(e);
    coro();
  }
}
```



StateMachine::

```
template<typename E>
void onEvent(E &&e) {
  auto &promise = coro.promise();
  if (promise.isWantedEvent(typeid(E))) {
    promise.currentEvent = std::forward<E>(e);
    coro();
  }
}
```

возобновление корутины

StateMachine::

```
template<typename E>
void onEvent(E &&e) {
  auto &promise = coro.promise();
  if (promise.isWantedEvent(typeid(E))) {
    promise.currentEvent = std::forward<E>(e);
    coro();
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

```
door.onEvent(Open{});
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

```
door.onEvent(Open{});
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

```
door.onEvent(Open{});
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

```
door.onEvent(Open{});
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

```
door.onEvent(Open{});
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

```
door.onEvent(Open{});
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

```
door.onEvent(Open{});
```

Использование

```
auto door = getDoor();
door.onEvent(Open{});  // Closed -> Open
door.onEvent(Close{}); // Open -> Closed
door.onEvent(Knock{});
door.onEvent(Close{}); // Closed -> Closed
```

выводит:

```
Come in, it's open!
```

Сравним что получилось

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний
- легко добавлять состояния

корутины

- ??? описание, с неявными состояниями

Стейт машина из корутин

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Сравним что получилось

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний
- легко добавлять состояния

корутины

- ??? описание, с неявными состояниями
- данные могут быть изолированы внутри состояний с помощью областей видимости { }

Изоляция данных

```
StateMachine getDoor() {
  for (;;) {
    { // закрыто
      auto e = co_await Event<Open, Knock>{};
      if (std::holds_alternative<Knock>(e)) {
        shout("Come in, it's open!");
        continue;
      }
    }
    { // открыто
      co_await Event<Close>{};
    }
  }
}
```

Изоляция и разделение данных

```
StateMachine getDoor(std::string response) {
 closed:
   for (;;) {
     auto e = co_await Event<Open, Knock>{};
     if (std::holds_alternative<Knock>(e)) {
       shout(response);
     }
     else if (std::holds_alternative<Open>(e)) {
       goto open;
     }
   }
 open:
   co_await Event<Close>{};
   goto closed;
 }
```

Изоляция и разделение данных

```
auto door = getDoor("Occupied!");
door.onEvent(Open{});  // Closed -> Open
door.onEvent(Close{}); // Open -> Closed
door.onEvent(Knock{});
door.onEvent(Close{}); // Closed -> Closed
```

выводит:

```
Occupied!
```

Сравним что получилось

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний
- легко добавлять состояния

корутины

- ??? описание, с неявными состояниями
- данные могут быть изолированы внутри состояний с помощью областей видимости { }
- относительно легко добавлять состояния

```
StateMachine getLockableDoor() {
 closed:
   for (;;) {
     auto e = co_await Event<Open, Knock, Lock>{};
     if (std::holds_alternative<Knock>(e)) {
       shout("Come in, it's open!");
     }
     else if (std::holds_alternative<Open>(e)) {
       goto open;
     }
     else if (std::holds_alternative<Lock>(e)) {
       goto locked;
     }
   }
 open:
   co_await Event<Close>{};
   goto closed;
 locked:
   co_await Event<Unlock>{};
   goto closed;
 }
```

Добавление состояний

```
StateMachine getLockableDoor() {
    for (;;) {
      // закрыто
      auto e = co_await Event<Open, Knock, Lock>{};
      if (std::holds_alternative<Knock>(e)) {
        shout("Come in, it's open!");
      }
      else if (std::holds_alternative<Open>(e)) {
        // открыто
        co_await Event<Close>{};
      }
      else if (std::holds_alternative<Lock>(e)) {
        // заперто
        co_await Event<Unlock>{};
      }
    }
}
```

Сравним что получилось

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний
- легко добавлять состояния
- ресурсы чётко контролируются разработчиком

корутины

- ??? описание, с неявными состояниями
- данные могут быть изолированы внутри состояний с помощью областей видимости { }
- относительно легко добавлять состояния
- корутины требуют выделения памяти (в общем случае)

Сравним что получилось

«шаблоны»

- декларативное описание
- данные изолированы внутри состояний
- легко добавлять состояния
- ресурсы чётко контролируются разработчиком

корутины

- ??? описание, с неявными состояниями
- данные могут быть изолированы внутри состояний с помощью областей видимости { }
- относительно легко добавлять состояния
- корутины требуют выделения памяти (в общем случае)
- код может иметь естественный вид для данной задачи

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Disconnect, Request>
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Close>{};
    }
  }
}
```

Стейт машина из корутин

```
StateMachine getDoor() {
  for (;;) {
    // закрыто
    auto e = co_await Event<Disconnect, Request>
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Connect>{};
    }
  }
}
```

Стейт машина из корутин

```
Sl co_await Event<Connect>{};
  for (;;) {
    // закрыто
    auto e = co_await Event<Disconnect, Request>
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // открыто
      co_await Event<Connect>{};
    }
  }
}
```

Стейт машина из корутин

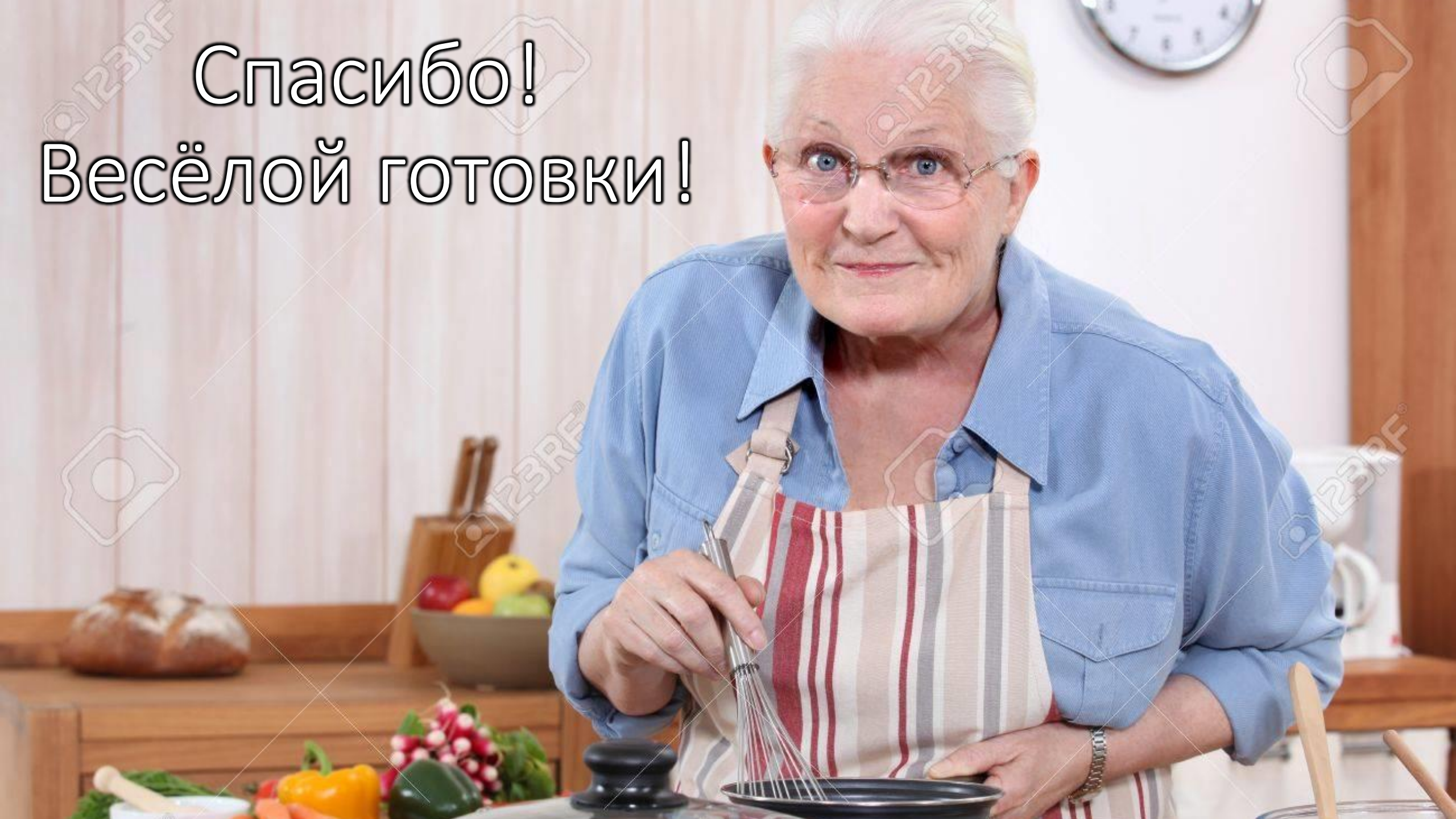
```
StateMachine getHandler() {
  co_await Event<Connect>{};
  for (;;) {
    auto e = co_await Event<Disconnect, Request>{};
    if (std::holds_alternative<Request>(e)) {
      handleRequest(std::get<Request>(e));
    }
    else if (std::holds_alternative<Disconnect>(e)) {
      co_await Event<Connect>{};
    }
  }
}
```

Использование

```
auto handler = getHandler();
handler.onEvent(Connect{});      // Disconnected -> Connected
handler.onEvent(Request{ 42 });
handler.onEvent(Disconnect{});   // Connected -> Disconnected
handler.onEvent(Disconnect{});   // Disconnected -> Disconnected
```

выводит:

```
handling request 42
```


Спасибо!
Весёлой готовки!

Делаем стейт машины из шаблонов и корутин

Павел Новиков

@cpp_ape

R&D Align Technology

Slides: https://git.io/JtSMn