# Await User Guide

Краткое руководство по concurrency в фреймворке Await.

## TL;DR

- Await позволяет пользователю параллельно запускать и координировать задачи (лямбды).
- Библиотека построена на трех ортогональных “китах”:
  - *Экзекуторы* (обычно – пулы потоков) запускают задачи.
  - *Файберы* позволяют задачам синхронизироваться с помощью привычных примитивов (мьютексов, ивентов, каналов), но не блокировать при этом потоки пула.
  - *Фьючи* позволяют декларативно комбинировать задачи и реализуют механизм отмены.
- Задачи исполняются *экзекуторами*. Базовые экзекуторы:
  - Пул потоков с общей очередью – для вычислительных задач
  - Шардированный work-stealing пул потоков – для IO-bound задач (файберов, корутин).
- Задачи планируются на исполнение в экзекуторах с помощью API futures: функции Execute или более многословного декларативного “заклинания” Just | Via | With | Map.
- Асинхронная задача всегда представлена фьючей.
- По умолчанию фьючи – ленивые: задача, представленная фьючей, отправится в экзекутор только в точке синхронного ожидания фьючи.

- Фьючи дают пользователю декларативный язык комбинаторов для композиции задач: последовательной (Map, Via, With, Flatten, и т.п.) и параллельной (Join, FirstOf и т.п.)
- Если в качестве экзекутора выбран executors::fibers::Pool, то все задачи будут исполняться в легковесных потоках – *файберах*. Файберы позволяют задачам синхронизироваться друг с другом с помощью привычных примитивов (мьютексы, каналы, ивенты, wait group и т.п.), и не блокировать при этом потоки пула, в котором они исполняются.
- futures – асинхронное API, декларативный язык комбинаторов, параллельная композиция; fibers – синхронное API, мьютексы и каналы, последовательная композиция.
- Всякую запущенную задачу можно [попросить] отменить, вызвав метод RequestCancel на соответствующей ей энергичной фьюче.
- Отмена кооперативная: задача должна проверять сигнал отмены вызывая executors::task::Checkpoint(), асинхронный сервис должен подписаться на отмену через futures::eager::Promise.
- Комбинаторы фьюч поддерживают structured concurrency: например, если один из входов для комбинатора FirstOf получает значение, то на другие входы автоматически отправляется сигнал отмены.
- С каждой задачей связан контекст – иммутабельный словарь для гетерогенных значений. Контекст доступен для задачи через executors::task::Context() и прозрачно передается через цепочки задач, построенные с помощью фьюч. Применение контекста – прозрачная для пользователя распределенная трассировка RPC.

# Напутствие

Чтение документации стоит совмещать с чтением / запуском примеров из examples/

Используется следующая цветовая нотация: классы и функции, пространства::имен, пути/в/репозитории

Пространства имен “отсчитываются” от корневого пространства имен await:: (выполните в уме using namespace await).

Пути в репозитории “отсчитываются” от await/.

# Словарь

Функцию назовем ***синхронной*** (*synchronous*), если логическая операция, которая стартует при вызове функции, завершается вместе с возвратом управления из вызова. В противном случае (т.е. когда операция длится дольше, чем вызов, который ее инициировал) функцию назовем ***асинхронной*** (*asynchronous*).

***Задача*** (*task*) – как правило*, это лямбда с пользовательским кодом.

* Внутри фреймворка это понятие шире: например, служебный файбер, запускающий задачи в экзекуторе `executors::fibers::Pool`, сам может стать задачей.

***Экзекутор*** (*executor*) – компонент, исполняющий запланированные в него задачи. Например, *пул потоков*.

***Пул потоков*** (*thread pool*) – экзекутор, запускающий задачи параллельно на фиксированном наборе потоков операционной системы.

***Файбер*** (*stackful fiber*) – поток исполнения, реализованный и планируемый в пространстве пользователя, без участия операционной системы.

Файберы ***останавливаются*** (*suspend*), тогда как потоки ***блокируются*** (*block*). Файберы ***кооперативны*** (*cooperative*), тогда как потоки ***вытесняемы*** (*preemptible*).

***Результат*** – значение типа T или ошибка, T + Error в алгебраической нотации

**Фьюча** (*future*) – представление будущего *результата* асинхронной задачи / произвольной асинхронной операции.

Фьючи делятся на ***энергичные*** (*eager*) и ***ленивые*** (*lazy*):

- *Энергичная* фьюча представляет уже запущенную задачу / уже стартовавшую асинхронную операцию.
- *Ленивая* фьюча представляет еще не запущенную задачу / операцию. Запуск происходит в точке использования результата (например, в точке синхронного ожидания фьючи).

***Комбинатор*** (*combinator*) – функция, получающая на вход одну или несколько фьюч и возвращающая новую фьючу.

***Сервис*** (*service*) – асинхронная функция (т.е. функция, возвращающая *фьючу*), или же класс / интерфейс с асинхронными функциями.

***Контекст*** (*context*) – иммутабельный словарь из строковых ключей в гетерогенные значения. Есть у каждой задачи / фьючи.

***Unit*** – тип с единственным значением.

# Навигация

Публичная часть библиотеки:

- await/ – алгоритм Await, реализация синхронного ожидания для фьюч / любых типов, реализующих концепт *Awaitable*
- executors/
  - impl/ – экзекуторы (пулы, файберные экзекуторы, Strand, Inline, Manual)
    - pools/ – пулы потоков
    - fibers/ – экзекуторы, запускающие задачи в файберах
  - task/ – текущая задача
    - Чекпоинт (для отмены)
    - Контекст
    - Экзекутор
- fibers/ – кооперативные файберы
  - sched/ – взаимодействие с планировщиком: Yield, Teleport, SleepFor
  - sync/ – примитивы синхронизации: Mutex, CondVar, OneShotEvent, WaitGroup, Channel
- futures/ – фьючи
  - types/ – объявление типов / концептов для фьюч: SomeFuture, Future<T>, BoxedFuture<T>, EagerFuture<T>
  - make/ – конструкторы фьюч: Execute, Contract, After, Just, Never и т.п.
  - combine/ – комбинаторы
    - seq/ – последовательная композиция
    - par/ – параллельная композиция: FirstOf, Join и др.
  - run/ – терминаторы цепочек фьюч: Await, Go, Unwrap, Sink
  - syntax/ – операторы для комбинирования фьюч

# Executors

*Executors are to function execution as allocators are to memory allocation.*

*Экзекутор* (или *экзекьютор*, *executor*) – компонент, исполняющий планируемые в него задачи. В коде представлен как реализация интерфейса `executors::IExecutor`.

## Запуск задачи

Задачи запускаются в экзекуторах с помощью асинхронного API фьюч.

### Ленивый

```
// Фьючи - ленивые!
auto future = futures::Execute(pool, []() {
  std::cout << “I’m running” << std::endl;
});

// ← К этому моменту задача еще не запущена (т.е. даже не отправлена в экзекутор `pool`), фьюча представляет лишь намерение запустить лямбду в пуле потоков

// Ожидание на фьюче приводит к планированию лямбды в экзекутор `pool` и запуску этой лямбды в потоке-воркере этого пула
threads::Await(std::move(future)).ExpectOk();
```

### Энергичный

Комбинатор `Start` на фьюче можно трансформирует ее из ленивой в энергичную и тем самым форсирует запуск задачи:

```
auto future = futures::Execute(pool, []() {
  // Будет напечатано из потока пула `pool`
  std::cout << “I’m running” << std::endl;
}) | futures::Start();  // ← Превращаем ленивую фьючу в энергичную

// У ленивой фьючи не будет такого метода
future.IAmEager();  // No-op

// ← Задача уже отправлена в экзекутор `pool` и возможно уже исполняется или даже завершилась
```

## Spawn

Для запуска задачи из другой задачи можно использовать более компактную форму `Execute` - `futures::Spawn`:

```
auto root = futures::Execute(pool, []() {
  // Запускаем задачу из задачи

  // Новая задача наследует экзекутор текущей, т.е. `pool`
  auto child = futures::Spawn([]() {
    std::cout << “Hi << std::endl;
  });

  fibers::Await(std::move(child)).ExpectOk();
});
```

## Управляемый

Функции `Execute` / `Spawn` реализованы через одну наиболее общую форму запуска пользовательского кода в Await - `Just | Via | With | Map`:

```
// Готовим лямбду к запуску в заданном экзекуторе и с заданным контекстом
auto task = futures::Just() |
            futures::Via(pool) |
            futures::With(context) |
            futures::Map([]() {
  std::cout << “I’m running” << std::endl;
});

// ← К этому моменту задача еще не запущена (т.е. не отправлена в экзекутор `pool`)

// Запускаем задачу и синхронно дожидаемся ее завершения
fibers::Await(std::move(task)).ExpectOk();
```

## Текущая задача

Путь в репозитории: executors/task/, ространство имен: executors::task

- Экзекутор: Executor()
- Чекпоинт (асинхронная отмена): Checkpoint()
- Пользовательский контекст: Context()

## Реализации

Реализации интерфейса IExecutor собраны в executors/impl / в пространстве имен executors::

- Пулы потоков (pools)

    - `pools::compute::ThreadPool` – пул потоков, разбирающих общую очередь задач, предназначен для вычислительных задач
    - `pools::fast::ThreadPool` – шардированный планировщик с work-stealing-ом, предназначен для запуска корутин / файберов, т.е. IO-bound задач
- `Strand` – декоратор над другим экзекутором, гарантирующий последовательное выполнение всех запланированных задач
- `ManualExecutor` – очередь задач с ручным запуском, позволяет писать детерминированные тесты для конкурентного кода, в том числе тестировать сам Await
- `InlineExecutor` – экзекутор по-умолчанию для фьюч, запускает задачу синхронно в точке планирования
- Файберные экзекуторы – запускают все планируемые задачи в служебных файберах
    - `fibers::Pool` – файберы запускаются параллельно, в пуле потоков
    - `fibers::ManualExecutor` – аналог простого (не-файберного) `ManualExecutor` для детерминированных тестов

См. примеры в `examples/executors/main.cpp`

# Futures

*Future* (*фьюча*) – представление будущего, еще не готового результата некой асинхронной операции.

Фьюча может представлять:

- Задачу, запланированную в экзекутор (например, вычисление в пуле потоков)
- Удаленный вызов (RPC)
- Таймер

“Распаковать” будущий результат, представляемый фьючей, можно двумя способами:

- либо синхронно дождаться завершения соответствующей ей операции,
- либо асинхронно запланировать в будущее *продолжение* (*continuation*).

# Result<T>

Фьюча может представлять вызов на удаленном сервере, и в общем случае такой вызов не обязательно будет успешен.

Поэтому результат “распакованной” фьючи представляется не просто значением типа T, а контейнером `::fallible::Result<T>`, который содержит либо значение типа T, либо ошибку (`::fallible::Error`).

См. примеры использования `::fallible::Result<T>`.

# Энергичность

Фьючи делятся на ***энергичные*** (*eager*) и ***ленивые*** (*lazy*)

*Энергичная* фьюча представляет уже запущенную задачу / уже стартовавшую асинхронную операцию.

*Ленивая* фьюча представляет еще не запущенную задачу / операцию. Запуск происходит в точке “распаковки” - в точке синхронного ожидания фьючи.

Фьючи в Await - ленивые!

## Типы

В futures/types/ перечислены концепты и типы фьюч.

### Future

Заголовок: future.hpp

Фьюча в Await - произвольный класс, удовлетворяющий концепту SomeFuture.

Концепт SomeFuture не фиксирует тип значения, вычисляемого фьючей и обычно используется как вход для комбинаторов.

Если тип вычисляемого фьючей значения известен, то вместо SomeFuture стоит использовать концепт Future<T>.

В примере ниже конструктор futures::Execute строит фьючу с уникальным типом, который параметризован типом пользовательской лямбды:

```
// Тип фьючи `f` выводится автоматически, но он должен удовлетворять концепту Future<T>
```

```
futures::Future<int> auto f = futures::Execute(pool, []() {
  return 42;
});
```

Помимо SomeFuture и Future<T>, определен концепт UnitFuture, который описывает произвольную фьючу, вычисляющую значение типа Unit:

```
futures::UnitFuture auto later = futures::After(1s);
```

### BoxedFuture

Заголовок: boxed.hpp

Для интерфейсов с асинхронными виртуальными функциями предусмотрен конкретный тип BoxedFuture<T> – фьюча со стертым типом (type erasure).

См. также пункт про упаковку / комбинатор Box.

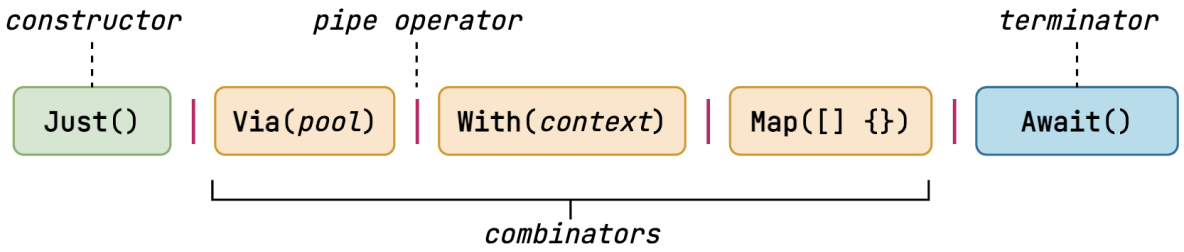
### EagerFuture

Заголовок: eager.hpp

Энергичная фьюча, полученная из ленивой применением комбинатора Start или оператора *bang*, представлена в Await конкретным типом EagerFuture<T>.

Энергичную фьючу можно проверить на готовность (метод HasResult) или отменить (метод RequestCancel).

## Цепочки и графы




(Запускаем в пуле *pool* лямбду [] {} с контекстом *context* и синхронно дожидаемся завершения)

С помощью API futures пользователь может планировать в экзекуторы цепочки задач:

1. Цепочка начинается с *конструктора* – функции, которая строит фьючу.
2. За конструктором следуют (*последовательные) комбинаторы*: очередной комбинатор потребляет фьючу, поданную на вход, и строит новую фьючу, добавляя новый шаг вычисления или декорируя его.
3. Цепочка завершается *терминатором*, который стартует построенную цепочку.

Цепочки можно объединять в ациклические направленные графы (DAG-и) с помощью *параллельных комбинаторов*, которые получают на вход сразу несколько фьюч.

## Конструкторы

В futures/make/ собраны *конструкторы* фьюч – функции, которые строят фьючи

- [execute.hpp] Запуск задачи
    - futures::Execute – запланировать лямбду на исполнение в заданный экзекутор

    - futures::Spawn - запланировать лямбду на исполнение в экзекутор текущей задачи
    - futures::Invoke - запланировать лямбду на исполнение в экзекутор Inline (т.е. построить фьючу, которая представляет ленивый синхронный вызов)
- Выполненные фьючи
    - [value.hpp] futures::Value(T) - выполненная фьюча с заданным значением типа T
    - [fail.hpp] futures::Fail(::fallible::Error) - выполненная фьюча с заданной ошибкой
    - [ready.hpp] futures::Ready(::fallible::Result<T>) - выполненная фьюча с заданным результатом
    - [unit.hpp] futures::Unit - UnitFuture, представляющая готовое значение типа Unit.
- [never.hpp] futures::Never - UnitFuture, которая никогда не выполнится (но может быть отменена)
- [after.hpp] futures::After - UnitFuture, которая выполнится через заданное число миллисекунд (для этого конструктора требуется планировщик таймеров)
- [contract.hpp] futures::Contract - энергичный контракт (пара EagerFuture<T> + Promise<T>) между *consumer*-ом и *producer*-ом.

## Аффинность

Построенную фьючу можно “потребить” (не важно, синхронно или асинхронно) не более одного раза.

Чтобы подчеркнуть это, все методы фьючи, которые потребляют будущий результат, определены для rvalue reference и требуют от пользователя явной передачи владения:

```
// std::move здесь означает передачу владения состоянием фьючи
std::move(future) | futures::Map([]() { ...});
// ← Теперь фьюча `f` потреблена асинхронным продолжением,
обращаться к ней больше нельзя
```

Свободные функции, которые получают фьючи на вход и потребляют их (например, комбинаторы типа futures::All и futures::FirstOf, а также алгоритм fibers::Await), получают их по значению, т.е. требуют от пользователя явной передачи владения:

```
// Передаем владение с помощью std::move
auto result = fibers::Await(std::move(future));
```

Если энергичная фьюча не будет явно потреблена пользователем, то при разрушении она отменит связанную с ней операцию / задачу.

## Комбинаторы

*Комбинаторы* - функции, которые получают одну или несколько фьюч на вход и возвращают новую фьючу.

### Последовательная композиция

Путь в репозитории: futures/combine/seq/

### Pipe operator

Строго говоря, последовательные комбинаторы представлены в Await не функциями, а *билдерами*, которые применяются к фьючам через *pipe operator* |.

```
// | - pipe operator
auto f = futures::After(1s) | futures::AndThen([](wheels::Unit)
{
  std::cout << “1s later” << std::endl;
});
```

Map

Заголовок: map.hpp или map_more.hpp (Map + вариации)

API futures позволяет планировать в экзекуторы не только отдельные задачи, но и цепочки задач – с помощью комбинатора Map:

```
futures::Future<B> auto f = futures::Execute(pool, []() → A {
  return ComputeA();
}) | futures::Map([](Result<A> input) → Result<B> {
  if (input.IsOk()) {
    return ComputeB(*input);
  } else {
    return fallible::PropagateError(input);
  }
});
```

На Map можно смотреть как на способ асинхронно “передать” будущий, еще не готовый результат, представленный в виде Future<T>, в лямбду-*продолжение*, которая получает на вход просто Result<T>, т.е. не будущий, а “настоящий”, уже материализованный результат.

Комбинатор Map применяется без ожидания: мы планируем запуск продолжения, не дожидаясь готовности входного результата для него!

Доступные варианты *продолжений* для Map:

- Result<T> → Result<U> (концепт *ResultMapper*)
- T → U (концепт *ValueMapper*)
- T → Result<U> (концепт *FaultyMapper*)
- Error → Result<T> (концепт *ErrorHandler*)

Мапперы, получающие на вход значение (T), не вызываются на ошибках, а обработчики ошибок - (зеркально) не вызываются на значениях.

Перед применением Map следует явно с помощью комбинатора Via указать экзекутор, в котором запустится продолжение. По умолчанию экзекутор наследуется от предшествующей в цепочке фьючи.

AndThen / OrElse

Заголовок: and_then.hpp / or_else.hpp или map_more.hpp

Для большей наглядности у Map есть две вариации: AndThen и OrElse, которые разделяют успешный путь / обработку ошибки:

- AndThen принимает только ValueMapper и FaultyMapper
- OrElse - только ErrorHandler

```
futures::Future<B> auto f = futures::Execute(pool, []() → int
{
  throw std::runtime_error(“Fail”);
}) | futures::AndThen([](int value) {
  return value + 1;  // не будет вызван
}) | futures::OrElse([](::fallible::Error) → Result<int> {
```

```
  return ::fallible::Ok(42);  // fallback
});
```

Anyway

Заголовок: anyway.hpp

Асинхронная операция, которую представляет фьюча, может завершиться успехом (т.е. значением), ошибкой или может быть отменена.

С помощью комбинатора Anyway можно запланировать выполнение кода для любого из этих исходов:

```
auto g = std::move(f) | futures::Anyway([semaphore]() {
  // Возвращаем ранее полученный permit в асинхронный семафор
  semaphore.Release();
});
```

Комбинатор Anyway служит аналогом RAII в C++ / defer в Golang, но для асинхронного API.

Flatten

Заголовок: flatten.hpp

Если продолжение само является асинхронным, т.е. возвращает фьючу, то следует избавиться от “вложенности” с помощью комбинатора Flatten:

```
// Value type для `f` - просто int, а не фьюча
futures::Future<int> auto f = futures::Execute(pool, [&]() {
  return futures::Execute(pool, []() {
```

```
    return 7;
  });
}) | futures::Flatten();  // ← “Уплощение” фьючи
```

FlatMap

Заголовок: flat_map.hpp или map_more.hpp

Комбинация Map + Flatten

Via

Заголовок: via.hpp

С помощью комбинатора Via можно указать экзекутор для запуска продолжения:

```
auto f = futures::Execute(pool, []() {
  // Исполняется в пуле потоков `pool`
}) | futures::Via(strand) | futures::Map([]() {
  // Исполняется в экзекуторе `strand`
});
```

Установленный через Via экзекутор наследуется по цепочке продолжений.

У комбинатора Via есть вариация Inline, которая устанавливает для продолжения экзекутор executors::Inline().

With

Заголовок: with.hpp

Комбинатор `With` устанавливает пользовательский контекст (`::carry::Context`) для продолжения фьючи.

### Hint / Yield

Заголовок: `yield.hpp`

С помощью комбинатора `Hint` можно дать подсказку планировщику (в виде `executors::SchedulerHint`) о том, как именно стоит планировать продолжение (например, использовать LIFO- или FIFO-планирование).

Комбинатор разумно применять только при использовании экзекутора `pools::fast::ThreadPool` (или производного от него).

Возможные подсказки (`executors::SchedulerHint`):

- *`Next`* - продолжение цепочки задач (запланировать продолжение через LIFO-слот локального потока-воркера)
- *`New`* - начало новой цепочки задач (хвост локальной очереди задач)
- *`Yield`* - уступить управление другим задачам (хвост разделяемой очереди задач)

По умолчанию для цепочки `Map`-ов используется LIFO-планирование (*`Next`*).

Комбинатор `Yield` = `Hint`(*`Yield`*) - асинхронный аналог `std::this_thread::yield`.

### Box

Заголовок: `box.hpp`

Комбинатор Box стирает конкретный тип фьючи Future<T> до типа BoxedFuture<T> (перемещая состояние фьючи на кучу либо используя SBO):

```
futures::BoxedFuture<int> f = futures::Execute(pool, []() {
  return 42;
}) | futures::Box();
```

Явно применять Box не обязательно - упаковка выполняется автоматически:

```
futures::BoxedFuture<int> f = futures::Execute(pool, []() {
  return 42;
});
```

## Start

Заголовок: start.hpp

Комбинатор Start запускает асинхронную операцию, представленную ленивой фьючей:

```
futures::EagerFuture<int> f = futures::Execute(pool, []() {
  return 42;
}) | futures::Start();
```

Результат применения комбинатора представлен специальным типом futures::EagerFuture<T> (*энергичная* фьюча).

См. также оператор ! (bang).

## WithTimeout

Заголовок: timeout.hpp

Комбинатор WithTimeout добавляет таймаут к асинхронной операции:

```
futures::Future<int> auto f = futures::Execute(pool, []() {
  return Compute();
}) | futures::WithTimeout(1s);
```

Если асинхронная операция не успевает завершиться за отведенное ей время, то фьюча выполняется с ошибкой ::fallible::errors::TimedOut, а операции отправляется сигнал отмены (см. главу Cancellation).

### Sequence

Заголовок: sequence.hpp

Комбинатор Sequence последовательно выполняет две фьючи, которые он получил на вход:

```
// Отложенное на 1s вычисление в пуле потоков
futures::Future<int> auto f = futures::Sequence(
  futures::After(1s),
  futures::Execute(pool, []() { return 42; }));
```

См. также оператор >> (sequence).

### Параллельная композиция

Путь в репозитории: futures/combine/par/

### FirstOf

Заголовок: first_of.hpp

Пусть у нас есть две фьючи, представляющие два параллельных RPC, и мы хотим дождаться первого ответа (*хэджирование* запросов).

Фьючи дают нам элегантное решение этой задачи – комбинатор `FirstOf`:

```
auto first = futures::FirstOf(std::move(f), std::move(g));
```

Вызов `FirstOf` завершается без ожидания.

Вызов `FirstOf` строит фьючу-комбинатор, который “подписан” на входы `f` и `g`, и который выполнится тогда, когда получит первый результат от одного из этих входов.

См. также оператор `||` (or).

### All

Заголовок: `all.hpp`

Параллельный комбинатор `All` получает на вход набор однородных по *ValueType* (обозначим его `T`) фьюч и строит фьючу `Future<vector<T>>`, которая “собирает” значения со всех входов `All`.

См. также оператор `+` (sum).

### Join

Заголовок: `join.hpp`

Параллельный комбинатор `Join` получает на вход набор произвольных фьюч и строит `UnitFuture`, которая

- успешно выполняется тогда, когда успешно выполняются все входы Join,
- выполняется с ошибкой при первой ошибке от любого из входов.

См. также оператор && (and).

## Терминаторы

Терминаторы завершают цепочки асинхронных операций / запускают фьючи.

Путь в репозитории: futures/run/

### Await

Заголовок: await.hpp

Терминатор Await *синхронно* “распаковывает” Future<T> в ::fallible::Result<T> : стартует асинхронную операцию, которую представляет фьюча + останавливает текущую задачу (т.е. останавливает файбер или блокирует поток) до завершения этой операции / получения от нее результата.

```
fallible::Result<T> result =
    futures::After(1s) |
    futures::Map([](wheels::Unit) {
      return 42;
    }) |
    futures::Await();
```

Синхронное ожидание реализовано через алгоритм ::await::Await и работает как для файберов, так и для системных потоков.

## Go

Заголовок: go.hpp

Терминатор Go стартует асинхронную операцию, которую представляет Future<T>, не дожидаясь ее завершения.

Состояние фьючи перемещается на кучу и будет автоматически разрушено после ее выполнения (завершения операции).

```
// Запускаем фоновую задачу в пуле потоков и забываем про нее
futures::Execute(pool, []() {
  std::cout << “Background” << std::endl;
}) | futures::Go();
```

См. также оператор ~ (go).

## Unwrap

Заголовок: unwrap.hpp

Терминатор Unwrap оптимистично “распаковывает” фьючу, предполагая, что она завершится синхронно в точке старта.

```
// Конструктор Invoke строит фьючу, представляющую ленивый запуск лямбды
::fallible::Result<int> result = futures::Invoke([]() {
  return 42;
}) | futures::Unwrap();
```

Основное применение Unwrap – юнит-тесты для фьюч.

См. также оператор * (unwrap).

### Sink

Заголовок: sink.hpp

Терминатор Sink замыкает цепочку, которую представляет фьюча, конвертируя ее в объект Chain.

Замкнутая цепочка запускается прямым вызовом Start:

```
executors::fibers::ManualExecutor manual;

auto chain = futures::Execute(manual, []() {
  for (int i = 0; i < 3; ++i) {
    fibers::Yield();
  }
}) | futures::Sink();

// Отправляем задачу в экзекутор
chain.Start();

// Выполняем задачу в файбере
manual.Drain();
```

## Операторы

В заголовочных файлах futures/syntax/ определены перегрузки операторов для более идиоматического комбинирования фьюч:

| Заголовок | Оператор | Определение |
|---|---|---|
| or.hpp | f or g (f \|\| g) | FirstOf(f, g) |
| join.hpp | f and g (f && g) | Join(f, g) |

| | | |
|---|---|---|
| sequence.hpp | f >> g | Sequence(f, g) |
| bang.hpp | !f | f \| Start() |
| unwrap.hpp | *f | f \| Unwrap() |
| go.hpp | ~f | f \| Go() |

Оператор | (*pipe*) активируется вместе с подключением любого последовательного комбинатора.

# Fibers

*Файберы* (*stackful fiber*) – легковесные потоки, реализованные в пространстве пользователя и планируемые без участия операционной системы.

Если вы знакомы с языком Golang, то можно (грубо) считать, что файберы – это *горутины* (*goroutines*).

## Мотивация

*The problem is that the thread, the software unit of concurrency, cannot match the scale of the application domain's natural units of concurrency — a session, an HTTP request, or a single database operation — nor can it match the scale of concurrency that modern hardware can support.*

## Кооперативность

Файберы – *кооперативные*: запущенный файбер не может быть вытеснен* в пользу другого файбера, он может уступить управление лишь добровольно:

- вызвав `fibers::Yield()`,
- вызвав `mutex.Lock()` на мьютексе, который в данный момент залочен другим файбером,
- начав ожидать фьючу: `fibers::Await(std::move(future))`,
- и т.п.

* Но может быть вытеснен вместе с потоком операционной системы по истечению кванта планирования.

# API

Файберы не взаимодействуют с планировщиком операционной системы и не блокируют потоки, поэтому для их планирования и синхронизации нужно использовать специальные функции / примитивы.

## Обращение к планировщику

- Yield - уступить текущий поток другой задаче / файберу: остановить текущий файбер и перепланировать его в текущий планировщик (например, если это пул потоков, то файбер переместится в конец очереди пула).
- SleepFor - остановить файбер на указанное время (для этой операции нужен планировщик таймеров).
- TeleportTo - перепланировать файбер в другой экзекутор (например, перепрыгнуть из IO-пула в CPU-пул для тяжелого вычисления).

## Синхронизация

- Mutex - мьютекс, взаимное исключения для файберов
- CondVar - условная переменная
- OneShotEvent - флаг для уведомления о наступившем событии / ожидания события. Одноразовый.
- Semaphore - ограничивает конкурентный доступ к разделяемому ресурсу.
- WaitGroup - счетчик с функцией ожидания. Позволяет дождаться завершения группы подзадач. Одноразовый.
- Channel<T> - буферизированный канал для передачи значений типа T между файберами + ограниченная версия Select - ожидание значения из нескольких каналов

Mutex, OneShotEvent, WaitGroup - lock-free, они реализованы без взаимного исключения на уровне потоков (т.е. без мьютексов / спинлоков)

Остановка файбера на ожидании в перечисленных выше примитивах не приводит к блокировке потока, на котором был запущен файбер!

## Прозрачность

API для прямого запуска файберов у пользователя нет: вместо него Await предоставляет специальный экзекутор - пул файберов - executors::fibers::Pool.

### Пул файберов

Экзекутор executors::fibers::Pool запускает планируемые задачи в служебных файберах на заданном количестве потоков.

При использовании пула файберов всякая запущенная задача может синхронизироваться с другими задачами с помощью привычных “блокирующих” примитивов (например, мьютексов), но не блокировать при этом потоки операционной системы!

### Эффективность

executors::fibers::Pool эффективно управляет служебными файберами:

- Если задача не перепланирует / не останавливает исполняющий ее служебный файбер, то после ее завершения этот файбер сразу же запустит следующую (в очереди планировщика) задачу, без дополнительных накладных расходов (переключений контекста).

- Если задача пользователя остановила файбер, в котором была запущена (и тем самым “приклеилась” к нему), то после ее завершения служебный файбер вернется в пул и по необходимости будет использован повторно.

## Stackful / Stackless

У stackful файберов есть альтернатива – *stackless* корутины или сопрограммы (*stackless coroutines*).

С одной стороны, stackless корутины не требуют аллокации стеков и избегают накладных расходов на переключение контекста исполнения (context switching).

С другой, они требуют от разработчика явной маркировки асинхронных вызовов специальными ключевыми словами (await / co_await / .await т.п.). В результате корутины раскрашивают код в два цвета, что приводит к неоднородности API.

Файберы, напротив, не требуют от разработчика маркировать вызовы, которые могут остановить файбер.

Если продолжать цветовую метафору, то можно сказать, что в случае файберов все функции одноцветные (например, розовые), так что API получается однородным.

Пример языка, который пошел по stackful пути – Golang, по stackless – Rust.

В C++ на уровне языка поддержаны stackless корутины, а stackful файберы могут быть реализованы в виде библиотеки.

## Фьючи и файберы

Фьючи и файберы – два механизма, которые можно использовать по отдельности, но если мы хотим достичь максимальной выразительности конкурентного кода, то эффективнее будет их комбинировать.

Если нам нужна последовательная композиция (A ; B), то с ней лучше справятся файберы:

- Последовательные шаги гораздо проще описывать не на специальном языке комбинаторов `Map` / `AndThen` / `OrElse`, а непосредственно на языке программирования, используя обычные синхронные вызовы и исключения.
- Кроме того, декларативность языка комбинаторов становится препятствием, когда требуется выразить простые императивные конструкции: циклы и ветвления.

Но если мы комбинируем операции параллельно (A || B), то тут декларативный язык комбинаторов `Join` и `FirstOf` оказывается более выразительным, чем синхронное API файберов.

# Алгоритм Await

Библиотека пользователю универсальный API для синхронного ожидания – алгоритм `Await`:

```
// Останавливаем файбер до готовности фьючи `future`
auto result = Await(std::move(future));
```

Алгоритм `fibers::Await` может дожидаться не только фьючу, он расширяется на любой тип *Awaitable*, для которого реализована функция `GetAwaiter`, возвращающая *Awaiter*, который умеет “дожидаться” события, связанного с *Awaitable*, т.е. возобновлять остановленный файбер при наступлении этого события.

В частности, *Awaiter* для *Awaitable* = `futures::SomeFuture` подписывается на фьючу и в коллбэке возобновляет остановленный поток / файбер.

Алгоритм `Await` инвариантен относительно потоков / файберов, он работает с абстракцией виртуального потока, за которой может находиться как поток операционной системы, так и stackful файбер.

# Cancellation

## Инициаторы

### Параллельные комбинаторы

Инициаторами отмены асинхронной задачи / операции чаще всего являются параллельные комбинаторы фьюч: например, если комбинатор `FirstOf` получил значение от одного из входов, то он инициирует отмену всех остальных – их результаты уже не понадобятся.

### Пользователь

Инициатором отмены уже запущенной (а значит – представленной энергичной фьючей) операции / задачи может быть и сам пользователь: энергичная фьюча предоставляет ему (потребляющий фьючу) метод `RequestCancel`.

## Best-Effort

Отмена фьючи – best-effort, а не гарантия отмены стоящей за ней операции / задачи!

Отмена со стороны пользователя / параллельного комбинатора и выполнение фьючи со стороны сервиса / завершение задачи со стороны пула потоков – в общем случае конкурирующие события, так что пользователь не может рассчитывать на какой-то определенный исход.

## Кооперативность

Отмена – кооперативная:

Отмена фьючи сам по себе не приводит к отмене соответствующей ей задачи или асинхронной операции.

Это лишь сигнал, уведомление, и этот сигнал требует проверки / подписки со стороны задачи / сервиса, построившего фьючу.

## Отмена задачи

Пусть за фьючей стоит задача, и она уже запустилась в экзекуторе.

Чтобы поддерживать отмену, задача должна периодически вызывать служебную функцию `executors::task::Checkpoint()`.

Любой такой вызов может обернуться служебным исключением `cancel::CancelledException`, которое «размотает» стек текущей задачи и отменит соответствующую ей фьючу.

Будьте осторожны: перехватывать такое исключение нельзя!

## Неявные чекпоинты

Чекпоинты могут быть расставлены внутри файберных функций планирования (`fibers/sched`) и примитивов синхронизации (`fibers/sync`).

Запуск очередного маппера в цепочке фьюч также является чекпоинтом.

## Отмена контракта

Пусть теперь фьюча порождена абстрактным сервисом с помощью `futures::Contract` и представляет некую асинхронную операцию.

Для отмены контракта асинхронный сервис должен

- либо периодически опрашивать futures::Promise с помощью метода IsCancelRequested
- либо подписаться на сигнал отмены с помощью метода CancelSubscribe

### Двойственность

Как можно заметить, EagerFuture и Promise двойственны:

- От Promise к EagerFuture передается результат, в обратную сторону - от EagerFuture к Promise - сигнал отмены.
- Клиент сервиса подписывается на результат через EagerFuture, реализация сервиса - на сигнал отмены через Promise.

## Structured Concurrency

### Параллельные комбинаторы

- Если комбинатор FirstOf получил значение от одного из своих входов, то он выполнится сам + инициирует отмену остальных входов - их результаты больше не нужны.
- Аналогично, если один из входов комбинатора All выдал ошибку, то All проваливается сам + отправляет сигнал отмены на остальные входы - операция уже провалена.
- Если комбинатор WithInterrupt получил “прерывание”, то он отправит сигнал отмены на основной вход.

### WaitGroup

Если одна из задач в fibers::WaitGroup завершилась ошибкой, то WaitGroup инициирует отмену всех задач группы.

См. Notes on structured concurrency, or: Go statement considered harmful

# Context

***Контекст*** (*context*) – иммутабельный словарь из строковых ключей в гетерогенные значения, который несет с собой каждая задача.

Представлен типом `::carry::Context`.

## Применение

Пользуется контекстом как правило не пользователь, а фреймворк, в котором этот пользователь пишет код.

Например, фреймворк RPC, который запускает в файберах задачи-обработчики удаленных вызовов (на стороне сервера) и строит фьючи / контракты при старте RPC (на стороне клиента), может использовать `Context` задач для прозрачной для пользователя передачи (*propagation*) контекста распределенной трассировки (trace-id, span-id).

## API

Сконструировать новый контекст можно с помощью `::carry::New`, а создать контекст на основе уже существующего – при помощи `::carry::Wrap`.

Установить контекст для запускаемой задачи (маппера) можно с помощью комбинатора `futures::With`.

Обратиться к контексту текущей задачи можно с помощью `::executors::task::Current()`