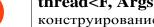
Параллелизм в С + +

future<RetType>









RetType | RetType& | void get() возвращает результат типа ResType, блокируется пока результат не доступен

bool valid()

true если get не был вызван

shared future<RetType> share()

конвертирует future в shared_future

void wait()

блокируется, пока результат не будет доступен

future_status wait_for(const duration&)

ожидает результат в течение какого-то времени

future_status wait_until(const time_point&)

ожидает результат до определённого времени

shared future<RetType>











shared_future(future<RetType>&&)

move-создание из future

RetType | RetType& | void get()

возвращает результат типа ResType, блокируется пока результат не доступен

bool valid()

true если get не был вызван

shared future<RetType> share()

конвертирует future в shared_future

void wait()

блокируется, пока результат не будет доступен

future_status wait_for(const duration&)

ожидает результат в течение какого-то времени

future status wait until(const time point&)

ожидает результат до определённого времени

Legend



Конструктор по-умолчанию



Конструктор копирования



Конструктор перемещения



Оператор присваивания



Оператор перемещения



Метод swap



thread ()









thread<F, Args...>(F&&, Args&&...)

конструирование из F и Args

bool joinable()

true если поток не был отделен (detached)

блокируется пока поток не завершится

void detach()

отказаться от контроля над потоком

id get id()

возвращает ID потока

native handle type native handle()

возвращает платформенно-зависимый handle потока

static unsigned hardware_concurrency()

возвращает количество аппаратных потоков

this thread *namespace*

thread::id get id()

возвращает ID вызывающего потока

void vield()

уступает процессорное время другим потокам

void sleep until(const time point&)

блокирует вызывающий поток до определённого времени

void sleep for(const duration&)

блокирует вызывающий поток на определённое время

Free functions

future<RetTypeOfF> async([launch], F&&, Args&&...)

Возвращает future и вызывает F с Args в соответствии с политикой запуска (если задана) или с launch::async / launch::deferred иначе

void lock<L1, L2, L3...>(L1&, L2&, L3&...)

захватывает все аргументы без дедлоков, в случае неудачи освобождает все аргументы и завершает работу

int try_lock<L1, L2, L3...>(L1&, L2&, L3&...)

вызывает try lock для каждого аргумента если аргумент не может быть захвачен освобождает все предыдущие аргументы и возвращает индекс незахваченного

void call_once(once_flag&, F&&, Args&&...)

вызывает F с Args только однажды

lock_guard<Mutex>

lock_guard(Mutex&, [adopt_lock_t])

Захватывает мьютекс при создании и освобождает при удалении

packaged task<RetType,

ArgTypes...> () M

Alloc&, F&&)

bool valid()

void reset()

старое

состояния

из потока

создании

мьютекс

(exception_ptr)

выходу из потока

выходе из потока

данных (если задано)

future<RetType> get_future()

void operator()(ArgTypes...)

promise<RetType>

future<RetType> get_future()

void set value(const RetType&)

возвращает future для данного promise

устанавливает результат и дёргает future

устанавливает исключение и дёргает future

устанавливает исключение и дёргает future по

void set_exception(exception_ptr)

void set_exception_at_thread_exit

unique_lock<Mutex>

try_to_lock_t | adopt_lock_t])

true если мьютекс заблокирован

возвращает указатель на мьютекс

mutex type* release()

mutex type* mutex()

bool owns lock()

native_handle)

unique lock(Mutex&, [defer lock t |

по возможности захватывает мьютекс при

разблокирует и возвращает указатель на

Также имеет все методы timed_mutex(кроме

возвращает future для данной задачи

выполняет задачу и даёт об этом знать future

true если задача имеет разделяемое состояние

void make_ready_at_thread_exit(ArgTypes...)

выполняет задачу и даёт об этом знать future по

создаёт новое разделяемое состояние, удаляет

promise<Alloc>(allocator_arg_t, const Alloc&)

void set_value(RetType&& | RetType& | void)

void set value at thread exit(const RetType&)

устанавливает результат и дёргает future по выходу

конструирует, используя Alloc для разделяемого



packaged_task<F, Alloc>(allocator_arg_t, const

конструирует из *F*, использует *Alloc* для внутренних









разблокирует один из ожидающих потоков

void notify_all()

void notify one()

разблокирует все ожидающие потоки

void wait(unique_lock<mutex>&, [Predicate])

освобождает мьютекс и блокирует поток, пока условная переменная не изменится; использует Predicate для проверки ложных пробуждений

cv status | bool wait until

condition variable

(unique lock<mutex>&, const time point&, [Predicate])

как wait, но ожидает только до определённого времени; возвращает cv status unu, если Predicate задан, значение Predicate

cv status | bool wait for

(unique lock<mutex>&, const duration&, [Predicate])

как wait, но ожидает только определённое время; возвращает су status или, если Predicate задан, значение Predicate

native handle type native handle()

возвращает платформенно-зависимый handle

condition_variable_any



Тоже что *condition_variable*, но *wait**-методы позволяют задать определённый lock-класс вместо unique_lock. Метод native_handle не доступен

mutex/recursive mutex



void lock()

recursive_mutex позволяет множественные вызовы из вложенных функций

bool try lock()

немедленно возвращает false если захват невозможен

void unlock()

native_handle_type native_handle()

возвращает платформенно-зависимый handle

timed mutex/ recursive timed mutex



То же самое, что и mutex/recursive_mutex, с двумя дополнительными методами:

bool try lock for(const duration&)

Попытка захватить мьютекс (за какое-то время)

bool try lock until(const time point&)

Попытка захватить мьютекс (до какого-то времени)