

Современные средства разработки ПО

Синхронизация потоков.

Фетисов Михаил Вячеславович

fetisov.michael@bmstu.ru

Переменные условия

Основное

- Переменные условия позволяют синхронизировать потоки посредством обмена сообщениями.
- Для их использования нужно подключить заголовочный файл `<condition_variable>`.
- Один поток выступает отправителем сообщения, а другой поток (или несколько потоков) – получателем.
- Получатель ждёт, пока не придёт сообщение.
- Чаще всего переменные условия применяются, когда нужно реализовать способ обработки данных по типу издателя и подписчика или производителя и потребителя.
- Переменная условия служит связующим звеном между отправителем и получателем сообщения.

Переменные условия

Методы класса

Функция	Описание
<code>notify_one</code>	Оповестить один ожидающий поток о наступлении события
<code>notify_all</code>	Оповестить все ожидающие потоки
<code>wait</code>	Ожидать сообщения, держа блокировщик открытым
<code>wait_for</code>	Ожидать сообщения, держа блокировщик открытым, но не более заданного промежутка времени
<code>wait_until</code>	Ожидать сообщения, держа блокировщик открытым, но не более, чем до заданного момента времени
<code>native_handle</code>	Возвращает системный дескриптор переменной условия

Переменные условия

Пример 1

```
// ConditionVariable.cpp
#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

std::condition_variable cond_var;
std::mutex cond_var_mutex;
bool is_data_ready {false};

void doTheWork() {
    std::cout << "Processing shared data." << std::endl;
}

void waitingForWork() {
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> locker(cond_var_mutex);
    cond_var.wait(locker, []{ return is_data_ready; });
    doTheWork();
    std::cout << "Work done." << std::endl;
}
```

```
void setDataReady() {
    {
        std::lock_guard<std::mutex> locker(cond_var_mutex);
        is_data_ready = true;
    }
    std::cout << "Sender: Data is ready." << std::endl;
    cond_var.notify_one();
}

int main() {
    std::thread t1(waitingForWork);
    std::thread t2(setDataReady);
    t1.join(); t2.join();
}
```

Переменные условия

Значение предиката

- Предикат наделяет переменную условия состоянием.
- Функция ожидания всегда должна сначала проверить истинность предиката.
- Предикат, таким образом, помогает бороться с двумя известными слабыми местами переменных условия: утерянным пробуждением и ложным пробуждением.

Переменные условия

Пример 2

```
// ConditionVariableBlock.cpp
#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

std::condition_variable cond_var;
std::mutex cond_var_mutex;

void waitingForWork() {
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> locker(cond_var_mutex);
    cond_var.wait(locker);
    // do the work
    std::cout << "Work done." << std::endl;
}

void setDataReady() {
    std::cout << "Sender: Data is ready." << std::endl;
    cond_var.notify_one();
}

int main() {
    std::thread t1(setDataReady);
    std::thread t2(waitingForWork);
    t1.join(); t2.join();
}
```

Переменные условия

Утерянные и ложные пробуждения

- **Утерянным пробуждением** называется ситуация, когда поток-отправитель успевает послать оповещение до того, как получатель начинает его ожидать. Как следствие оповещение оказывается утерянным.
- **Ложное пробуждение** — это пробуждение ожидающего потока, когда отправители никаких оповещений не посылали.
- (одна из причин такого явления — *похищенное пробуждение*: перед тем как пробуждённый поток-адресат получает шанс запуститься, другой поток успевает вклиниться первым и начинает выполнение)

Семафор

Определение

- **Семафор** (англ. *semaphore*) — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является недопустимой.
- Служит для построения сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты *критических секций*, а также для управления доступом к аппаратному обеспечению.

Критическая секция

определение

- **Критическая секция** — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком выполнения. При нахождении в критической секции двух (или более) потоков возникает состояние «гонки» («состязания»).

Семафор

Появление понятия

- Понятие семафора предложил в 1965 г. нидерландский учёный в области информатики и программирования Эдсгер Вибе Дейкстра.
- Семафор — это структура данных, содержащая очередь и счётчик.
- Счётчик инициализируется значением, большим или равным нулю.
- Семафор поддерживает две операции: **wait** и **signal**.
- Операция **wait** захватывает семафор, уменьшая значение счётчика, если оно положительно, или блокирует поток в противном случае.
- Операция **signal** освобождает семафор путём увеличения счётчика и, если очередь заблокированных потоков не пуста, пробуждает первый поток из неё.
- Постановка заблокированных потоков в очередь необходима для предотвращения *ресурсного голода*.

Ресурсный голод

Определение

- **Ресурсный голод** — ситуация, когда поток или процесс не может продолжить работу и проводит неопределённо долгое время в ожидании из-за того, что система каждый раз отказывает ему в предоставлении некоторого ресурса

promise-futures

- Стандартная библиотека предоставляет механизм для получения возвращаемых значений и перехвата исключений, генерируемых асинхронными задачами (т.е. функциями, запускаемыми в отдельных потоках).
- Эти значения передаются в общем состоянии, в котором асинхронная задача может записать свое возвращаемое значение или сохранить исключение, и которые могут быть проверены, ожидаться и иным образом обрабатываться другими потоками, которые содержат экземпляры `std::future` или `std::shared_future`, которые ссылаются на это общее состояние.

promise-future как каналы передачи данных

Обещание: отправитель

Фьючерс: получатель



поместить



Канал



извлечь

Задания как каналы передачи данных

std-классы, связанные с механизмом promise-futures

promise	сохраняет значение для асинхронного извлечения
packaged_task	упаковывает функцию для сохранения ее возвращаемого значения для асинхронного извлечения
future	ожидает значения, которое устанавливается асинхронно
shared_future	ожидает значения (возможно, на которое ссылаются другие фьючерсы), которое устанавливается асинхронно
async	запускает функцию асинхронно (потенциально в новом потоке) и возвращает std::future, который будет содержать результат
launch	определяет политику запуска для std::async

future

- Шаблон класса `std::future` предоставляет механизм доступа к результату асинхронных операций:
 - Асинхронная операция (созданная с помощью `std::async`, `std::packaged_task` или `std::promise`) может предоставить объект `std::future` создателю этой асинхронной операции.
 - Создатель асинхронной операции может затем использовать различные методы для запроса, ожидания или извлечения значения из `std::future`. Эти методы могут блокироваться, если асинхронная операция еще не предоставила значение.
 - Когда асинхронная операция готова отправить результат создателю, она может сделать это, изменив общее состояние (например, `std::promise::set_value`), которое связано с `std::future` создателя.
- Обратите внимание, что `std::future` ссылается на общее состояние, которое не используется совместно ни с какими другими объектами асинхронного возврата (в отличие от `std::shared_future`).

```
#include <future>
#include <iostream>
#include <thread>

int main()
{
    // future from a packaged_task
    std::packaged_task<int()> task([]{ return 7; }); // wrap the function
    std::future<int> f1 = task.get_future(); // get a future
    std::thread t(std::move(task)); // launch on a thread

    // future from an async()
    std::future<int> f2 = std::async(std::launch::async, []{ return 8; });

    // future from a promise
    std::promise<int> p;
    std::future<int> f3 = p.get_future();
    std::thread([&p]{ p.set_value_at_thread_exit(9); }).detach();

    std::cout << "Waiting..." << std::flush;
    f1.wait();
    f2.wait();
    f3.wait();
    std::cout << "Done!\nResults are: "
              << f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';
    t.join();
}
```

Output:

```
Waiting...Done!  
Results are: 7 8 9
```

```
#include <future>
#include <iostream>
#include <thread>

int main()
{
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t([&p]
    {
        try
        {
            // code that may throw
            throw std::runtime_error("Example");
        }
        catch (...)
        {
            try
            {
                // store anything thrown in the promise
                p.set_exception(std::current_exception());
            }
            catch (...) {} // set_exception() may throw too
        }
    });

    try
    {
        std::cout << f.get();
    }
    catch (const std::exception& e)
    {
        std::cout << "Exception from the thread: " << e.what() << '\n';
    }
    t.join();
}
```


Output:

```
Exception from the thread: Example
```

Асинхронные задания

Определение

- **Асинхронные задания (механизм будущих результатов)** — позволяет выполнять две операции параллельно, пока результат выполнения одной операции не понадобятся для другой.
- Задание характеризуется пакетом работы, которую предстоит выполнить, и обладает двумя частями: *обещанием* (promise) и *ожидаемым результатом* (future).

Асинхронные задания

Описание

- Задания ведут себя подобно каналам, по которым данные проходят от входного разъёма до выходного. Один конец канала называется **обещанием** (promise), другой — **будущим** (future).
- Этими разъёмами может управлять один и тот же канал, а могут — разные.
- Обещание помещает данные в канал.
- Будущее в неопределённый момент в будущем извлекает из канала результат их преобразования.

Асинхронные задания

Пример 1

```
// JustAsync.cpp
#include <future>
#include <iostream>

int main()
{
    auto fut= std::async([]{ return 2000 + 11; });
    std::cout << "fut.get(): " << fut.get() << std::endl;
}
```

Асинхронные задания

Пример 2

```
// AsyncVersusThread.cpp
#include <future>
#include <thread>
#include <iostream>

int main()
{
    int res;
    std::thread t([&]{ res = 2000 + 11; });
    t.join();
    std::cout << "res: " << res << std::endl;

    auto fut= std::async([]{ return 2000 + 11; });
    std::cout << "fut.get(): " << fut.get() << std::endl;
}
```


Отличие заданий от потоков

Таблица

Критерий	Потоки	Задания
Основные сущности	Родительский и дочерний потоки	Обещание и фьючерс
Способ передачи данных	Общая переменная	Канал
Отдельный поток	Всегда	Иногда
Синхронизация	Функция join ожидает завершения потока	Функция get блокирует выполнение
Исключение в дочернем потоке (задании)	Оба потока завершаются вместе со всем процессом	Передаётся через обещание и фьючерс
Передаваемые данные	Значения	Значения, оповещения и исключения

Отличие заданий от потоков

Пояснение

- Для обмена данными между родительским и дочерним потоками нужна переменная, к которой имеют доступ оба потока. Взаимодействие с заданием происходит через канал. Как следствие заданиям не нужны примитивы синхронизации наподобие мьютексов.
- Если общей переменной, через которую обмениваются данными родительский и дочерний потоки, можно злоупотребить (поскольку два потока имеют к ней доступ, любой из них может, меняя значение переменной, влиять на поведение другого), взаимодействие с заданием носит более явный характер.
- Результат выполнения задания можно запросить через фьючерс только один раз, вызвав его функцию **get**. Повторный вызов этой функции на том же фьючерсе приводит к неопределённому поведению. Это не относится, однако, к классу **std::shared_future**, из которого значение можно запрашивать многократно.
- Родительский поток ждёт завершения дочернего, вызывая функцию **join**. С фьючерсом нужно использовать функцию **get**, которая блокирует выполнение до тех пор, пока результат задания не станет доступен.
- Если исключение возникает и не перехватывается в потоке, завершается и этот поток, и создавший его, и весь процесс. Для сравнения: обещание умеет отправить своё исключение фьючерсу, откуда его можно достать и обработать.
- Обещание может обслуживать один или несколько фьючерсов. Оно может посылать значение, исключение или просто оповещение. Обещания можно использовать в качестве безопасной замены для переменной условия.

Всегда следует предпочитать асинхронные вызовы (`std::async`)

- Реализация стандартной библиотеки C++ сама решает, выполнять асинхронный вызов в отдельном потоке или нет.
- Это решение может зависеть от числа доступных ядер процессора, загруженности системы, размера пакета работы.
- Вызывая функцию **`std::async`**, программист лишь передаёт ей задание, которое должно быть выполнено.
- Вся работа по возможному созданию потока и управлению временем его жизни перекладывается на внутренние механизмы реализации.
- Помимо того, при вызове функции **`std::async`** можно необязательным параметром передать политику запуска.

Асинхронные задания

Политика запуска

- Посредством политики можно в явном виде указать, каким образом реализации следует выполнить асинхронный вызов: в том же потоке, который создал вызов (`std::launch::deferred`), или в другом потоке (`std::launch::async`).
- Особенность выражения вида
`auto fut = std::async(std::launch::deferred, ...)`
 - состоит в том, что обещание не запускается немедленно. Вместо этого оно будет выполнено ленивым образом только в момент вызова **`fut.get()`**. Иными словами, обещание запускается только тогда, когда фьючерс в явном виде запрашивает его результат.

Строгие и ленивые вычисления

- Строгое и ленивое вычисления — это две противоположных способа вычислять значение выражения.
- При **строгой стратегии** выражение вычисляется немедленно, тогда как при **ленивой стратегии** вычисление откладывается до тех пор, пока значение не станет необходимо.
- Строгое вычисление также называют *жадным*, а ленивое — *отложенным*, или вычислением *по требованию*.
- Ленивая стратегия вычислений часто помогает сберечь время и ресурсы процессора, предотвращает вычисление данных, которые могут не понадобиться в будущем.

Строгие и ленивые вычисления

Пример

```
// AsyncLazy.cpp
#include <chrono>
#include <future>
#include <iostream>

int main() {
    auto begin = std::chrono::system_clock::now();
    auto async_lazy = std::async(std::launch::deferred,
        []{ return std::chrono::system_clock::now(); });
    auto async_eager = std::async(std::launch::async,
        []{ return std::chrono::system_clock::now(); });

    std::cout << "Waiting one second." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));

    auto lazy_start = async_lazy.get() - begin;
    auto eager_start = async_eager.get() - begin;
    auto lazy_duration = std::chrono::duration<double>(lazy_start).count();
    auto eager_duration = std::chrono::duration<double>(eager_start).count();

    std::cout << "async_lazy evaluated after : " << lazy_duration
        << " seconds." << std::endl;
    std::cout << "async_eager evaluated after: " << eager_duration
        << " seconds." << std::endl;
}
```

Строгие и ленивые вычисления

Пример (результат выполнения программы)

```
Waiting one second.  
async_lazy evaluated after : 1.00016 seconds.  
async_eager evaluated after: 6.5047e-05 seconds.
```

Асинхронные задания

Запустить и забыть

- Особый случай составляют фьючерсы, о которых забывают сразу после создания. Они не сохраняются в каких-либо переменных и должны запускаться немедленно в момент создания. Существенно, что обещания таких фьючерсов должны выполняться в отдельном потоке, чтобы фьючерс мог начать работу немедленно. Для этого нужно использовать политику запуска **std::launch::async**.

```
std::async(std::launch::async, []{ std::cout << "fire and forget" << '\n';  
});
```

- Короткоживущие фьючерсы выглядят удобными, но имеют существенный недостаток — они выполняются на самом деле последовательно.

Promise

Обещание

- Шаблон класса `std::promise` предоставляет средство для хранения значения или исключения, которое позже будет получено асинхронно с помощью объекта `std::future`, созданного объектом `std::promise`.
- Объект `std::promise` предназначен для использования только один раз.
- Задание характеризуется пакетом работы, которую предстоит выполнить, и обладает двумя частями: *обещанием* (`promise`) и *ожидаемым результатом* (`future`).

Promise

Обещание

- Каждое обещание связано с общим состоянием, которое содержит некоторую информацию о состоянии и результат, который может быть еще не оценен, оценен как значение (возможно, недействительное) или оценен как исключение.
- Обещание может выполнять три действия с общим состоянием:
 - Подготовить (make ready): обещание сохраняет результат или исключение в общем состоянии. Помечает состояние готовности и разблокирует любой поток, ожидающий future, связанном с общим состоянием.
 - Освободить (release): обещание отказывается от своей ссылки на общее состояние. Если это была последняя такая ссылка, общее состояние уничтожается. Если это не было общее состояние, созданное `std::async`, которое еще не готово, эта операция не блокируется.
 - Отказаться (abandon): обещание сохраняет исключение типа `std::future_error` с кодом ошибки `std::future_errc::broken_promise`, делает общее состояние готовым, а затем освобождает его.

Promise

Обещание

- Обещание является "отправным" концом канала связи `promise-future`: операция, которая сохраняет значение в общем состоянии, синхронизируется с (как определено в `std::memory_order`) успешным возвратом из любой функции, ожидающей общего состояния (например, `std::future::get`).
- Параллельный доступ к одному и тому же общему состоянию может привести к конфликту в противном случае: например, несколько вызывающих `std::shared_future::get` должны либо быть доступны только для чтения, либо обеспечивать внешнюю синхронизацию.

Скалярное умножение векторов с помощью асинхронных вызовов

- DotProductAsync.cpp

Использование обещаний и фьючерсов

Примеры

- `PromiseFuture.cpp`
- `Future.cpp`
- `FutureException.cpp`
- `Promise.cpp`

Вопросы?

Фетисов Михаил Вячеславович
fetisov.michael@bmstu.ru