4.2.2. Связывание задачи с фьючерсом

Шаблон класса `std::packaged_task<>` привязывает фьючерс к функции или вызываемому объекту. Когда вызывается объект `std::packaged_task<>`, он вызывает связанную функцию или объект и приводит фьючерс в состояние *готовности* с возвращаемым значением, сохраненным в качестве связанных данных. Этим можно воспользоваться как строительным блоком для пула потоков (см. главу 9) или других схем управления задачами, например для запуска каждой задачи в собственном потоке или последовательного запуска всех задач в специально выделенном потоке, работающем в фоновом режиме. Если крупную операцию возможно разбить на автономные подзадачи, каждую из подзадач можно заключить в экземпляр `std::packaged_task<>`, который затем будет передан диспетчеру задач или пулу потоков. Тем самым удается абстрагироваться от подробностей задач — диспетчер имеет дело только с экземплярами `std::packaged_task<>`, а не с отдельно взятыми функциями.

Параметром шаблона для шаблона класса `std::packaged_task<>` является сигнатура функции, например `void()` для функции, не получающей параметры и не имеющей возвращаемых значений, или `int(std::string&,double*)` для функции, получающей не-`const`-ссылку на `std::string` и указатель на `double` и возвращающей значение типа `int`. При создании экземпляра `std::packaged_task` ему следует передать функцию или вызываемый объект, принимающий указанные параметры, а затем возвращающий тип, который можно преобразовать в указанный тип возвращаемого значения. Точного совпадения типов не требуется, можно сконструировать объект `std::packaged_task<double(double)>` из функции, принимающей значение типа `int` и возвращающей значение типа `float`, поскольку типы способны проходить подразумеваемое преобразование.

Тип возвращаемого значения, указанный в сигнатуре функции, определяет тип объекта `std::future<>`, возвращаемого из компонентной функции `get_future()`, а заданный в сигнатуре список аргументов используется для определения сигнатуры оператора вызова в классе упакованной задачи. Например, в листинге 4.8 приведено частичное определение класса для `std::packaged _task <std::string(std::vector<char>*,int)>`.

Листинг 4.8. Частичное определение класса для специализации std::packaged_task< >

```
template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
    template<typename Callable>
    explicit packaged_task(Callable&& f);
    std::future<std::string> get_future();
    void operator()(std::vector<char>*,int);
};
```

Объект `std::packaged_task` является вызываемым, значит, его можно обернуть объектом `std::function`, передать `std::thread` в качестве функции потока, передать другой функции, требующей вызываемого объекта, или даже вызвать напрямую.

Когда `std::packaged_task` вызывается в качестве функционального объекта, аргументы, предоставленные оператору вызова функции, передаются содержащейся в этом объекте функции, а возвращаемое значение сохраняется в качестве асинхронного результата в объекте `std::future`, полученном от `get_future()`.

Таким образом, задачу можно заключить в объект `std::packaged_task` и извлечь фьючерс перед передачей объекта `std::packaged_task` в то место, где он надлежащим образом будет вызван. Когда понадобится результат, можно будет дождаться готовности фьючерса. Практическое применение рассмотренного алгоритма показано в следующем примере.

Передача задач между потоками

Многие среды графических пользовательских интерфейсов (GUI) требуют, чтобы обновления GUI выполнялись из строго определенных потоков, поэтому, если какому-то потоку требуется обновить GUI, он должен отправить сообщение потоку, определенному для выполнения таких обновлений. В листинге 4.9 показано, что один из способов решения данной задачи, не требующий специальных сообщений для каких-либо действий, связанных с GUI-интерфейсом, обеспечивается шаблоном `std:packaged_task`.

Листинг 4.9. Запуск кода в GUI-потоке с помощью std::packaged_task

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>
std::mutex m;
std::deque<std::packaged_task<void()> > tasks;
bool gui_shutdown_message_received();
void get_and_process_gui_message();
void gui_thread()                                    ❶
{
    while(!gui_shutdown_message_received())          ❷
    {
        get_and_process_gui_message();               ❸
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lk(m);
            if(tasks.empty())                        ❹
                continue;
            task=std::move(tasks.front());           ❺
            tasks.pop_front();
        }
        task();                                      ❻
    }
}
std::thread gui_bg_thread(gui_thread);
template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
```

```
{
    std::packaged_task<void()> task(f);  ❼
    std::future<void> res=task.get_future();  ❽
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));  ❾
    return res;  ❿
}
```

Код не отличается особой сложностью: поток GUI ❶ выполняет цикл, пока не будет получено сообщение, предписывающее GUI прекратить работу ❷, многократно проводя опрос на наличие GUI-сообщений на обработку ❸, например, щелчков кнопкой мыши, и на наличие задач в очереди. Если в очереди нет задач ❹, цикл повторяется, в противном случае задача извлекается из очереди ❺, с очереди снимается блокировка, после чего запускается задача ❻. Когда задача будет выполнена, фьючерс, связанный с ней, перейдет в состояние готовности.

Так же просто происходит и помещение задачи в очередь: предоставленная функция создает новую упакованную задачу ❼, вызовом компонентной функции `get_future()` из этой задачи получается фьючерс ❽ и задача помещается в список ❾ перед тем, как фьючерс возвращается вызывающему коду ❿. Затем код, отправивший сообщение потоку GUI, может ожидать фьючерс, если ему требуется узнать о завершении задачи, или же проигнорировать фьючерс, если эта информация не нужна.

В данном примере для задач используется экземпляр класса `std::packaged_task<void()>`, в который заключается функция или другой вызываемый объект, не принимающий параметров и возвращающий значение `void` (если он возвращает что-то иное, это значение игнорируется). Это самая простая из возможных задач, но из ранее увиденного легко сделать вывод, что `std::packaged_task` применяется и в более сложных ситуациях — указав в качестве параметра шаблона другую сигнатуру функции, можно изменить тип возвращаемого значения (и, стало быть, тип данных, хранящихся в состоянии, связанном с фьючерсом), а также типы аргументов в операторе вызова функции. Этот пример легко можно распространить на задачи, предназначенные для запуска в потоке GUI, чтобы получить аргументы и возвратить значение в `std::future`, а не просто воспользоваться им в качестве индикатора завершения задачи.

А как быть с такими задачами, которые нельзя выразить в виде простого вызова функции, или с теми, где результат может приходить из более чем одного места? В таких случаях приходится обращаться к третьему способу создания фьючерса — к использованию `std::promise` для явного задания значения.

4.2.3. Создание промисов std::promise

Когда есть приложение, нуждающееся в обработке большого количества сетевых подключений, оно зачастую пытается обработать каждое подключение в отдельном потоке, так как это позволяет легче воспринимать сетевое подключение и упрощает программирование. При небольшом количестве подключений и, соответственно,

небольшом количестве потоков проблем не возникает. К сожалению, по мере роста числа подключений такой подход утрачивает привлекательность: большое количество потоков потребляет немалый объем ресурсов операционной системы и потенциально становится причиной множества переключений контекста (когда количество потоков превышает доступный объем ресурсов оборудования для конкурентности), что ухудшает производительность. В экстремальных случаях операционная система может исчерпать свои ресурсы по запуску новых потоков до того, как будут растрачены ее объемы для сетевых подключений. Поэтому в приложениях с большим количеством сетевых подключений для их обработки принято обходиться небольшим количеством потоков (возможно, только одним потоком), каждый из которых имеет дело с несколькими подключениями.

Рассмотрим один из таких потоков, обрабатывающих подключения. Пакеты данных будут поступать из разных подключений и обрабатываться, по сути, в произвольном порядке, и точно так же в произвольном порядке пакеты данных будут помещаться в очередь на отправку. Во многих случаях другим частям приложения придется ждать либо успешной отправки данных, либо успешного получения новых пакетов данных через конкретные сетевые подключения.

Шаблон `std::promise<T>` позволяет устанавливать значение (типа `T`), которое позже можно прочитать через связанный объект `std::future<T>`. Один из возможных механизмов такого рода будет реализован парой `std::promise` — `std::future`, ожидающий поток может заблокироваться на фьючерсе, а поток, предоставляющий данные, — воспользоваться другой половиной пары, промисом (promise, иногда называют обещанием), для установки связанного значения и приведения фьючерса в состояние готовности.

Получить объект фьючерса `std::future`, связанный с заданным промисом `std::promise`, можно вызовом компонентной функции `get_future()`, как это было с `std::packaged_task`. Когда значение промиса установлено (с помощью компонентной функции `set_value()`), фьючерс приводится в состояние готовности и может использоваться для извлечения сохраненного значения. Если объект `std::promise` уничтожить без установки значения, вместо него будет сохранено исключение. Передача исключений между потоками рассматривается в подразделе 4.2.4.

В листинге 4.10 показан пример кода для потока, обрабатывающего подключения в соответствии в приведенным ранее описанием. В данном примере пара `std::promise<bool>` — `std::future<bool>` используется для идентификации успешной передачи блока исходящих данных: значение, связанное с фьючерсом, представляет собой простой флаг успеха/сбоя. Для входящих пакетов данными, связанными с фьючерсом, становится полезная нагрузка пакета.

Листинг 4.10. Обработка с помощью промисов сразу нескольких подключений в одном потоке

```
#include <future>
void process_connections(connection_set& connections)
{
    while(!done(connections))  ←❶
    {                                  ❷
        for(connection_iterator        ←
```

```
            connection=connections.begin(),end=connections.end();
        connection!=end;
        ++connection)
    {
        if(connection->has_incoming_data()) ◄──❸
        {
            data_packet data=connection->incoming();
            std::promise<payload_type>& p=
                connection->get_promise(data.id); ◄──❹
            p.set_value(data.payload);
        }
        if(connection->has_outgoing_data()) ◄──❺
        {
            outgoing_packet data=
            connection->top_of_outgoing_queue();
            connection->send(data.payload);
            data.promise.set_value(true);     ◄──❻
        }
    }
}
```

Функция `process_connections()` выполняет цикл, пока функция `done()` не возвратит значение `true` ❶. При каждом проходе цикла поочередно проверяется каждое подключения ❷, извлекаются входящие данные, если они есть ❸, или отправляются любые стоящие в очереди исходящие данные ❺. Предполагается, что у пакета входящих данных есть идентификатор, а у него — полезная нагрузка. Идентификатор отображается на объект `std::promise` (возможно, поиском в ассоциативном контейнере) ❹, а в качестве значения устанавливается информационная нагрузка пакета. Исходящий пакет извлекается из очереди исходящих данных и отправляется по подключению в сеть. Когда отправка завершится, промис, связанный с исходящими данными, устанавливается равным `true`, что говорит об успешной передаче ❻. Насколько хорошо все это будет перенесено на сетевой протокол, зависит от самого протокола: такой стиль структуры «промис — фьючерс» для конкретного сценария может и не сработать, хотя данная структура похожа на структуру поддержки асинхронного ввода-вывода некоторых операционных систем.

В приводимом до сих пор коде проигнорированы возможности выдачи исключений. Приятно, конечно, представлять себе мир, в котором все всегда работает идеально, но так не бывает. Иногда возникает дефицит дискового пространства, или еще нет искомых данных, или сбоит сеть, или пропадает доступ к базе данных. Если бы операция выполнялась в том потоке, которому нужен результат, код мог бы просто отрапортовать об ошибке путем выдачи исключения, но было бы неразумно ограничиваться требованием, чтобы все шло как надо, только потому, что возникла потребность использовать `std::packaged_task` или `std::promise`. Поэтому стандартная библиотека C++ задает четкий путь решения проблем с выдачей исключений при таком сценарии развития событий, позволяя сохранять эти исключения в качестве части связанного результата.