Часть XVI. Паттерны проектирования

Следующие библиотеки реализуют паттерны проектирования.

- Boost.Flyweight реализует паттерн "приспособленец" (flyweight). Полезен в том случае, если в программе используется множество идентичных объектов, и необходимо уменьшить использование памяти.
- Boost.Signals2 делает паттерн "наблюдатель" (observer) простым в использовании. Эта библиотека называется Boost.Signals2 потому, что она реализует концепцию "сигнал/слот".
- Boost.MetaStateMachine дает возможность использовать конечные автоматы из UML в C++.

Содержание

- ~~66. Boost.Flyweight~~
- 67. Boost.Signals2
- ~~68. Boost.MetaStateMachine~~

Глава 67. Boost.Signals2

Содержание

- Сигналы
- Соединения
- Многопоточность

Boost.Signals2 реализует концепцию "сигнал/слот". Одна или несколько функций под названием *слоты* связаны с объектом, который может "посылать" сигнал. Каждый раз, когда посылается сигнал, связанные с ним функции будут вызваны.

Концепция "сигнал/слот" может быть полезна, например, при разработке приложений с графическим интерфейсом. Кнопки могут быть сделаны так, чтобы посылать сигнал, когда пользователь нажимает на них. Они могут поддерживать связи сразу с несколькими функциями для обработки пользовательского ввода. Таким образом можно гибко обрабатывать события.

Класс `std::function` так же может быть использован для обработки событий. Важное различие между `std::function` и Boost.Signals2 состоит в том, что Boost.Signals2 позволяет связать более одного обработчика с одним событием. Поэтому, библиотека Boost.Signals2 лучше для событийно-ориентированной разработки и должна быть в приоритете, в случае, если необходимо обрабатывать события.

Boost.Signals2 сменяет библиотеку Boost.Signals, которая устарела и не рассматривается в данной главе.

Сигналы

Boost.Signals2 предоставляет класс `boost::signals2::signal`, который используется для создания сигналов. Этот класс определен в файле `boost/signals2/signal.hpp`. Также, вы можете использовать основной заголовочный файл `boost/signals2.hpp`, в котором определены все классы и функции библиотеки Boost.Signals2.

Boost.Signals2 определяет `boost::signals2::signal` и другие классы, так же, как и все функции, в пространстве имен `boost::signals2`.

Пример 67.1. “Hello, world!” с использованием boost::signals2::signal

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
}
```

`boost::signals2::signal` в качестве шаблонного параметра принимает сигнатуру функции, используемой в качестве обработчика событий. В Примере 67.1, только функции с сигнатурой `void()` могут быть связаны с сигналом **s**.

Лямбда-функция связывается с сигналом **s** с помощью функции `connect()`. Так как лямбда-функция удовлетворяет требуемой сигнатуре

`void()`, связывание произошло успешно. Лямбда-функция будет вызвана всякий раз, когда сработает сигнал **s**.

Сигнал посылается с помощью вызова объекта **s** так же, как вы вызываете обычную функцию. Сигнатура этой функции совпадает с той, которая была передана в качестве шаблонного параметра. Скобки пусты потому, что `void()` не принимает никаких аргументов. Вызов **s** приводит к срабатыванию сигнала, что в свою очередь вызывает лямбда-функцию, которая была ранее связана с помощью `connect()`.

Пример 67.1 так же может быть реализован с помощью `std::function`, так как показано в Примере 67.2.

Пример 67.2. “Hello, world!” с использованием std::function

```
#include <functional>
#include <iostream>

int main()
{
  std::function<void()> f;
  f = []{ std::cout << "Hello, world!\n"; };
  f();
}
```

В Примере 67.2 лямбда-функция также вызывается, когда **f** будет вызвана. В то время как `std::function` может использоваться только как в Примере 67.2, библиотека Boost.Signals2 предоставляет гораздо больше разнообразия. Например, она может связать несколько функций с отдельным сигналом (см. Пример 67.3).

Пример 67.3. Несколько обработчиков событий с использованием boost::signals2::signal

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello"; });
  s.connect([]{ std::cout << ", world!\n"; });
  s();
}
```

`boost::signals2::signal` позволяет связать несколько функций с отдельным сигналом с помощью многократных вызовов `connect()`. Каждый раз, когда срабатывает сигнал, функции будут вызваны в том порядке, в котором они были связаны с помощью `connect()`.

Так же, порядок может быть определен явно с помощью перегруженной функции `connect()`, которая принимает значение типа `int` в качестве дополнительного аргумента (Пример 67.4).

Пример 67.4. Обработчики событий с явным порядком

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect(1, []{ std::cout << ", world!\n"; });
  s.connect(0, []{ std::cout << "Hello"; });
  s();
}
```

Как и предыдущий пример, Пример 67.4 выведет на экран `Hello, world!`.

Чтобы отсоединить функцию от сигнала, вызовете `disconnect()`.

Пример 67.5. Отсоединение обработчиков событий от boost::signals2::signal

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

void hello() { std::cout << "Hello"; }
void world() { std::cout << ", world!\n"; }

int main()
{
  signal<void()> s;
  s.connect(hello);
  s.connect(world);
  s.disconnect(world);
  s();
}
```

Пример 67.5 выведет на экран только `Hello`, потому что связь с `world()` была разорвана до того как сигнал был послан.

Помимо `connect()` и `disconnect()`, `boost::signals2::signal` предлагает ещё несколько функций (см. Пример 67.6).

Пример 67.6. Дополнительные функции boost::signals2::signal

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello"; });
  s.connect([]{ std::cout << ", world!"; });
  std::cout << s.num_slots() << '\n';
  if (!s.empty())
    s();
  s.disconnect_all_slots();
}
```

`num_slots()` возвращает число связанных функций. Если связанных функций нет, то `num_slots()` возвращает 0. `empty()` сообщит вам, связаны ли с сигналом обработчики событий или нет. А функция `disconnect_all_slots()` делает именно то, что говорит её имя: она разрывает все существующие связи.

Пример 67.7. Обработка значений, возвращаемых слотами при срабатывании сигнала.

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<int()> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::cout << *s() << '\n';
}
```

В Примере 67.7 две лямбда-функции связаны с сигналом **s**. Первая лямбда-функция возвращает 1, а вторая возвращает 2.

Пример 67.7 выведет `2` в стандартный поток вывода. Оба возвращаемых значения были правильно приняты сигналом **s**, но все, кроме последнего, были проигнорированы. По умолчанию, возвращается только последнее возвращаемое значение всех связанных функций.

Обратите внимание, что `s()` не возвращает результат последней вызванной функции напрямую. Возвращается объект типа `boost::optional`, который при разыменовывании возвращает число 2. Посланный сигнал, который не связан с какой-либо функцией, не возвращает никакого значения. Таким образом, в данном случае, `boost::optional` позволяет Boost.Signals2 вернуть пустой объект. С классом `boost::optional` мы познакомились в Главе 21.

Можно настроить сигнал таким образом, что каждое возвращаемое значение будет соответствующим образом обработано. Чтобы сделать это, необходимо передать *объединитель* (combiner) в `boost::signals2::signal` в качестве второго шаблонного параметра.

Пример 67.8. Поиск наименьшего возвращаемого значения с помощью пользовательского объединителя

```
#include <boost/signals2.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace boost::signals2;

template <typename T>
struct min_element
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    std::vector<T> v(first, last);
    return *std::min_element(v.begin(), v.end());
  }
};

int main()
{
  signal<int(), min_element<int>> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::cout << s() << '\n';
}
```

Объединитель - это класс с перегруженным оператором `operator()`. Этот оператор будет автоматически вызван с двумя итераторами в качестве аргументов, которые используются для доступа к функциям, связанным с определенным сигналом. При разыменовывании итераторов, происходит вызов соответствующих функций, и их возвращаемые значения становятся доступны в объединителе. В дальнейшем, обычный алгоритм из стандартной библиотеки, такой как `std::min_element()` может быть использован для расчета и возврата наименьшего значения (см. Пример 67.8).

`boost::signals2::signal` использует `boost::signals2::optional_last_value` как объединитель по умолчанию. Этот объединитель возвращает объект типа `boost::optional`. Пользователь может определить объединитель с возвращаемым значением любого типа. Например, объединитель `min_element` в Примере 67.8 возвращает значение такого же типа, который был передан в качестве шаблонного параметра в `min_element`.

Невозможно передать такой алгоритм, как `std::min_element()` в качестве шаблонного параметра напрямую в `boost::signals2::signal`. `boost::signals2::signal` ожидает, что в объединителе объявлен тип `result_type`, который указывает тип возвращаемого значения оператора `operator()`. Так как этот тип не определен в стандартных алгоритмах, компилятор выдаст сообщение об ошибке.

Обратите внимание, что не представляется возможным передать итераторы **first** и **last** в `std::min_element()` напрямую, потому что этот алгоритм ожидает итераторы прямого доступа (forward iterators), в то время как объединитель работает с итераторами ввода (input iterators). Поэтому, перед тем как определить наименьшее значение с помощью `std::min_element()`, для хранения всех возвращаемых значений используется вектор.

В Примере 67.9 объединитель изменен так, чтобы хранить все возвращаемые значения в контейнере, вместо того чтобы сравнивать их. Он размещает все возвращаемые значения в векторе, который будет возвращен при вызове `s()`.

Пример 67.9. Получение всех возвращаемых значений при помощи пользовательского объединителя

```
#include <boost/signals2.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace boost::signals2;

template <typename T>
struct return_all
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    return T(first, last);
  }
};

int main()
{
  signal<int(), return_all<std::vector<int>>> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::vector<int> v = s();
  std::cout << *std::min_element(v.begin(), v.end()) << '\n';
}
```

Соединения

Функциями можно управлять с помощью функций `connect()` и `disconnect()`, предоставляемых классом `boost::signals2::signal`. Поскольку `connect()` возвращает объект типа `boost::signals2::connection`, соединениями можно управлять по-другому (см. Пример 67.10).

Пример 67.10. Управление соединениями с помощью boost::signals2::connection

```
#include <boost/signals2.hpp>
#include <iostream>

int main()
{
  boost::signals2::signal<void()> s;
  boost::signals2::connection c = s.connect(
    []{ std::cout << "Hello, world!\n"; });
  s();
  c.disconnect();
}
```

Функции `disconnect()` класса `boost::signals2::signal` в качестве аргумента требуется указатель на функцию. Обойти это можно путем вызова `disconnect()` с объектом `boost::signals2::connection`.

Для того, чтобы заблокировать функцию на короткий промежуток времени, не разрывая связи с сигналом, может быть использован `boost::signals2::shared_connection_block`.

Пример 67.11. Блокировка соединений с помощью shared_connection_block

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  connection c = s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
  shared_connection_block b{c};
  s();
  b.unblock();
  s();
}
```

В Примере 67.11 лямбда-функция вызывается дважды. Сигнал **s** посылается три раза, но лямбда-функция не будет вызвана во второй раз потому что был создан объект типа `boost::signals2::shared_connection_block` для того чтобы заблокировать вызов. Как только объект выходит из области видимости, блокировка автоматически удаляется. Блокировка также может быть удалена явно с помощью вызова функции `unblock()`. Так как `unblock()` вызывается до последней отправки сигнала, последний вызов лямбда-функции будет выполнен.

В дополнение к `unblock()`, `boost::signals2::shared_connection_block` предоставляет функции `block()` и `blocking()`. Первая используется для блокирования соединения после вызова `unblock()`, а последняя дает возможность проверить заблокировано ли соединение в данный момент или нет.

Обратите внимание, что `boost::signals2::shared_connection_block` содержит слово “shared” по причине того, что несколько объектов типа `boost::signals2::shared_connection_block` могут быть инициализированы с помощью одного и того же соединения.

Пример 67.12. Многократная блокировка соединений

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  connection c = s.connect([]{ std::cout << "Hello, world!\n"; });
  shared_connection_block b1{c, false};
  {
    shared_connection_block b2{c};
    std::cout << std::boolalpha << b1.blocking() << '\n';
    s();
  }
  s();
}
```

В Примере 67.12 сигнал **s** посылается дважды, но лямбда-функция будет вызвана только во второй раз. Программа выведет `Hello, world!` в стандартный поток вывода только один раз.

Так как `false` передан в конструктор в качестве второго аргумента, первый объект типа `boost::signals2::shared_connection_block` не блокирует соединение с сигналом **s**. Следовательно, вызов `blocking()` объекта **b1** вернет `false`.

Тем не менее, лямбда-функция не выполняется при первом вызове **s**, так как доступ происходит только после того как второй объект типа `boost::signals2::shared_connection_block` будет инстанцирован. Соединение будет заблокировано, так как конструктору не был передан второй параметр. Когда **s** вызывается во второй раз, лямбда-функция выполнится, так как блокировка была автоматически удалена, когда **b2** вышел из области видимости.

Boost.Signals2 может разрывать соединения, как только объект, чьим членом является функция, связанная с сигналом, будет разрушен.

Пример 67.13. Функция-член в качестве обработчика событий

```
#include <boost/signals2.hpp>
#include <memory>
#include <functional>
#include <iostream>

using namespace boost::signals2;

struct world
{
  void hello() const
  {
    std::cout << "Hello, world!\n";
  }
};

int main()
{
  signal<void()> s;
  {
    std::unique_ptr<world> w(new world());
    s.connect(std::bind(&world::hello, w.get()));
  }
  std::cout << s.num_slots() << '\n';
  s();
}
```

В Примере 67.13 функция-член объекта связывается с сигналом с помощью `std::bind()`. Объект разрушен до того, как сигнал будет послан, что является проблемой, так как вместо того чтобы передать объект типа `world` в `std::bind()`, будет передан только адрес. К тому времени как `s()` будут вызван, объект, на который мы ссылались, не будет существовать.

Возможно изменить программу так, чтобы соединение автоматически разрывалось, как только объект был уничтожен. В Примере 67.14 происходит именно это.

Пример 67.14. Автоматический разрыв связи с функцией-членом

```
#include <boost/signals2.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::signals2;

struct world
{
  void hello() const
  {
    std::cout << "Hello, world!\n";
  }
};

int main()
{
  signal<void()> s;
  {
    boost::shared_ptr<world> w(new world());
    s.connect(signal<void()>::slot_type(&world::hello, w.get()).track(w));
  }
  std::cout << s.num_slots() << '\n';
  s();
}
```

Теперь `num_slots()` вернет `0`. В Примере 67.14 при отправки сигнала, не происходит попытки вызвать функцию-член объекта, который не существует. Изменение состоит в том, что мы обернули объект типа `world` умным указателем `boost::shared_ptr`, который передаётся в `track()`. Эта функция-член вызывается для слота, который был передан в `connect()`, для запроса отслеживания соответствующего объекта.

Функция или функция-член связанная с сигналом называется *слотом*. Тип для задания слота не был использован в предыдущих примерах, потому что передать указатель на функцию или функцию-член в `connect()` было достаточно. Соответствующий слот был создан и связан с сигналом автоматически.

Тем не менее, в Примере 67.14 умный указатель связывается со слотом путем вызова `track()`. Так как тип слота зависит от сигнала, `boost::signals2::signal` предоставляет тип `slot_type` для доступа к требующемуся типу. `slot_type` ведет себя так же, как `std::bind`, что позволяет передавать оба параметра для описания слота напрямую. `track()` может связать слот с умным указателем типа `boost::shared_ptr`. Затем объект отслеживается, что приводит к тому, что слот будет автоматически удален, как только отслеживаемый объект разрушится.

Для того, чтобы управлять объектами с другими умными указателями, слоты предоставляют функцию под названием `track_foreign()`. В то время как `track()` принимает умный указатель типа `boost::shared_ptr`, `track_foreign()` позволяет, например, использовать умный указатель типа `std::shared_ptr`. Поддержка умных указателей, кроме `std::shared_ptr` и `std::weak_ptr`, должна быть добавлена в Boost.Signals2, прежде чем эти указатели могут быть переданы в `track_foreign()`.

Подписчик конкретного события может получить доступ к объекту типа `boost::signals2::signal` для создания новых связей или разрыва связи существующих.

Пример 67.15. Создание новых соединений в обработчике событий

```
#include <boost/signals2.hpp>
#include <iostream>

boost::signals2::signal<void()> s;

void connect()
{
  s.connect([]{ std::cout << "Hello, world!\n"; });
}

int main()
{
  s.connect(connect);
  s();
}
```

В Примере 67.15 мы получаем доступ к **s** внутри функции `connect()` для связывания лямбда-функции с сигналом. Так как `connect()` вызывается, когда сигнал уже послан, возникает вопрос, будет ли вызвана лямбда-функция.

Программа не выводит ничего, что означает, что лямбда-функция ни разу не была вызвана. Хотя Boost.Signals2 и поддерживает связывание функций с сигналами во время срабатывания сигнала, новые связи будут использованы только при следующем срабатывании сигнала.

Пример 67.16. Разрыв соединения в обработчике событий

```
#include <boost/signals2.hpp>
#include <iostream>

boost::signals2::signal<void()> s;

void hello()
{
  std::cout << "Hello, world!\n";
}

void disconnect()
{
  s.disconnect(hello);
}

int main()
{
  s.connect(disconnect);
  s.connect(hello);
  s();
}
```

В Примере 67.16 не создаются новые связи, а вместо этого - разрываются существующие. Как и в Примере 67.15, ничего не будет выведено в стандартный поток вывода.

Такое поведение можно объяснить довольно просто. Представьте себе, что всякий раз, когда срабатывает сигнал создается временная копия всех слотов. Вновь созданные связи не будут добавлены во временную копию и, следовательно, могут быть вызваны только при

следующим срабатывании сигнала. А разорванные связи по-прежнему будут частью временной копии и, для того, чтобы избежать вызова функции-члена уже разрушенного объекта, будут проверяться объединителем при разыменовывании.

Многопоточность

Почти все классы, предоставляемые Boost.Signals2, являются потокобезопасными и могут использоваться в многопоточных приложениях. Например, к объектам типа `boost::signals2::signal` и `boost::signals2::connection` можно обращаться из разных потоков.

С другой стороны, `boost::signals2::shared_connection_block` не потокобезопасен. Это ограничение не важно, потому что несколько объектов типа `boost::signals2::shared_connection_block` могут быть созданы в различных потоках и могут использовать одни и те же объекты соединений.

Пример 67.17. Потокобезопасность boost::signals2::signal

```
#include <boost/signals2.hpp>
#include <thread>
#include <mutex>
#include <iostream>

boost::signals2::signal<void(int)> s;
std::mutex m;

void loop()
{
  for (int i = 0; i < 100; ++i)
    s(i);
}

int main()
{
  s.connect([](int i){
    std::lock_guard<std::mutex> lock{m};
    std::cout << i << '\n';
  });
  std::thread t1{loop};
  std::thread t2{loop};
  t1.join();
  t2.join();
}
```

В Примере 67.17 создается два потока, выполняющие функцию `loop()`, которая обращается к **s** сто раз, что вызывает связанную лямбда-функцию. Boost.Signals2 явно поддерживает одновременный доступ из разных потоков к объектам типа `boost::signals2::signal`.

Пример 67.17 выведет числа от 0 до 99. Поскольку **i** инкрементируется в двух потоках и выводится в стандартный поток вывода в лямбда-функции, числа не только будут выводится дважды, они, к тому же, будут пересекаться. Однако, так как доступ к `boost::signals2::signal` можно получить из разных потоков, программа не завершится с ошибкой.

Тем не менее, в Примере 67.17 по-прежнему требуется синхронизация. Поскольку два потока обращаются к **s**, связанная лямбда-функция работает параллельно в двух потоках. Для того, чтобы два потока не прерывали друг друга вовремя вывода в стандартный поток, для синхронизации доступа к **std::cout** используется мьютекс.

Для однопоточных приложений, поддержка многопоточности в Boost.Signals2 может быть отключена.

Пример 67.18. boost::signals2::signal без потокобезопасности

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

signal<void()> s;

int main()
{
  typedef keywords::mutex_type<dummy_mutex> dummy_mutex;
  signal_type<void(), dummy_mutex>::type s;
  s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
}
```

Последний из многих шаблонных параметров, поддерживаемых `boost::signals2::signal`, определяет тип мьютекса, используемого для синхронизации. К счастью, Boost.Signals2 предлагает более простой способ отключить синхронизацию, чем передача всего списка параметров.

Пространство имен `boost::signals2::keywords` содержит классы, которые позволяют передавать шаблонные параметры по имени. `boost::signals2::keywords::mutex_type` может быть использован для передачи типа мьютекса в качестве второго шаблонного параметра для `boost::signals2::signal_type`. Обратите внимание, что в данном случае используется `boost::signals2::signal_type`, а не `boost::signals2::signal`. Тип эквивалентный типу `boost::signals2::signal`, необходимый для определения сигнала **s**, извлекается с помощью `boost::signals2::signal_type::type`.

Boost.Signals2 предоставляет пустую реализацию мьютекса под названием `boost::signals2::dummy_mutex`. Если сигнал определяется с этим классом, он больше не будет поддерживать многопоточность (см. Пример 67.18).