Dyspozytor w C++11

(czyt. dispatcher w C++11)

Bartosz ’BaSz’ Szurgot

bartek.szurgot@baszerr.eu

2014-06-26

Cel

- Obsługa przychodzących wiadomości
- Odbiór poprzez dyspozytora
- Wiadomości bez wspólnej bazy
- Użytkownik otrzymuje docelowy typ

Format wiadomości

```
1  #pragma once
2  struct Hello
3  {
4    static constexpr int type() { return 142; }
5    // some user data
6  };
7  struct Say
8  {
9    static constexpr int type() { return 675; }
10   // more user data
11 };
12 struct Bye
13 {
14   static constexpr int type() { return 234; }
15   // even more user data
16 };
```

Format przesyłowy

```
1  #pragma once
2  #include <string>
3
4  struct BinaryMsg
5  {
6    int         type_;
7    std::string data_;
8  };
9
10 template<typename M>
11 BinaryMsg serialize(M const& m);
12
13 template<typename M>
14 M deserialize(BinaryMsg const& b);
```

Klasa bazowa

```
1  #pragma once
2  #include "BinaryMsg.hpp"
3
4  struct Dispatcher
5  {
6    virtual ~Dispatcher(void) { }
7    virtual void dispatch(BinaryMsg const& bin) = 0;
8  };
```

Nagłówek

```
1  #pragma once
2  #include "Dispatcher.hpp"
3  #include "messages.hpp"
4
5  struct ManualDispatcher: public Dispatcher
6  {
7    virtual void dispatch(BinaryMsg const& bin);
8    void handle(Hello const& msg);
9    void handle(Say const& msg);
10   void handle(Bye const& msg);
11 };
```

Implementacja z piekła rodem

```
1  #include <stdexcept>
2  #include "ManualDispatcher.hpp"
3
4  void ManualDispatcher::dispatch(BinaryMsg const& bin)
5  {
6    if( bin.type_ == Hello::type() )
7    {
8      handle( deserialize<Hello>(bin) );
9      return;
10   }
11   if( bin.type_ == Say::type() )
12   {
13     handle( deserialize<Say>(bin) );
14     return;
15   }
16   if( bin.type_ == Bye::type() )
17   {
18     handle( deserialize<Bye>(bin) );
19     return;
20   }
21   throw std::runtime_error{"unknnown message"};
22 }
```

Prowizorka – zawsze najtrwalsza

API marzeń. . .

```
1  #include "Dispatcher.hpp"
2  #include "messages.hpp"
3
4  struct DispatcherImpl: public Dispatcher
5  {
6    virtual void dispatch(BinaryMsg const& bin) final
7    { /* >>> here the miracle occurs <<< */ }
8  };
9
10 struct PerfectDispatcher: public DispatcherImpl
11 {
12   void handle(Hello const& msg);
13   void handle(Say const& msg);
14   void handle(Bye const& msg);
15 };
```

Realizowalne API

```
1  #include "Dispatcher.hpp"
2  #include "messages.hpp"
3
4  template<typename FinalType, typename ...Msgs>
5  struct DispatcherImpl: public Dispatcher
6  {
7    DispatcherImpl()
8    { /* registers type to ID->handler map */ }
9    virtual void dispatch(BinaryMsg const& bin) final
10   { /* using registered ID to find proper handlers */ }
11 };
12
13 struct DoableDispatcher: public DispatcherImpl<DoableDispatcher,
14                                                Hello, Say, Bye>
15 {
16   void handle(Hello const& msg);
17   void handle(Say const& msg);
18   void handle(Bye const& msg);
19 };
```

Faktyczna implementacja

```
1  #include <unordered_map>
2  #include <boost/cast.hpp>
3  #include "Dispatcher.hpp"
4  #include "Reg.hpp"
5
6  template<typename FinalType, typename ...M>
7  struct DispatcherImpl: public Dispatcher
8  {
9    DispatcherImpl(void)
10   {
11     Reg<FinalType, M..., void>::call(handlers_);
12     assert( handlers_.size() == sizeof...(M) && "non-unique message ids" );
13   }
14
15   virtual void dispatch(BinaryMsg const& bin) final override
16   {
17     auto it = handlers_.find(bin.type_);
18     if(it==std::end(handlers_))
19       throw std::runtime_error{"unknown message"};
20     auto h = it->second;
21     assert(h);
22     (*h)( *boost::polymorphic_downcast<FinalType*>(this), bin );
23   }
24
25 private:
26   std::unordered_map<int, void(*)(FinalType&, BinaryMsg const&)> handlers_;
27 };
```

Pomoce rejestracyjne

```
1   #pragma once
2   #include <unordered_map>
3   #include <boost/cast.hpp>
4   #include "BinaryMsg.hpp"
5
6   template<typename FinalType, typename M>
7   void caller(FinalType& ft, BinaryMsg const& bin)
8   {
9     ft.handle( deserialize<M>(bin) );
10  }
11
12  template<typename FinalType, typename H, typename ...T>
13  struct Reg
14  {
15    template<typename C>
16    static void call(C& c)
17    {
18      c[H::type()] = caller<FinalType,H>;
19      Reg<FinalType, T...>::call(c);
20    }
21  };
22
23  template<typename FinalType>
24  struct Reg<FinalType, void>
25  {
26    template<typename C>
27    static void call(C& c) { }
28  };
```

Lepszy automat == mniej pracy

Cechy proponowanego rozwiązania

+ Banalne w użyciu
+ Brak konieczności powielania kodu
+ Szybsze dla większych liczb wiadomości (vs. *if-else*)
- Nieznacznie dłuższa kompilacja (vs. *if-else*)
- Większe zużycie pamięci (vs. *if-else*)
- Nieco większy kod (lambda vs. funkcja)

Wariacje na temat...

- Weryfikacja unikalności listy wiadomości w czasie kompilacji
- Generowanie kodu szukającego zamiast tablicy hashującej
- Generowanie kodu szukającego logarytmicznie
- Zewnętrzny „getter” ID wiadomości
- Zewnętrzny (de)serializator wiadomości

Pytania?