Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 9
13.11.2018

Visitor: предпосылки

Зададимся простой иерархией...

```
class TAnimal {
  public:
    virtual ~TAnimal() {}
    virtual void Talk() const = 0;
    virtual void Move() const = 0;
};
```

Visitor: предпосылки

```
using  TAnimalPtr = std::shared_ptr<TAnimal>;

class TCat : public TAnimal {
  public:
    virtual void Talk() const override{
        std::cout << "meow" << std::endl;
    }
    virtual void Move() const override{
        std::cout << "cat jumps" << std::endl;
    }
};

class TDog : public TAnimal {
  public:
    virtual void Talk() const override{
        std::cout << "woof" << std::endl;
    }
    virtual void Move() const override{
        std::cout << "dog moves" << std::endl;
    }
};
```

Visitor: предпосылки

Простая фабрика по созданию объектов:

```
TAnimalPtr CreateAnimal() {
    std::string subj;
    std::cin >> subj;
    if (subj == "cat") {
        return static_cast<TAnimalPtr>(new TCat);
    } else if (subj == "dog") {
        return static_cast<TAnimalPtr>(new TDog);
    }
    return nullptr;
}
```

Обычный динамический полиморфизм:

```
int main() {
    TAnimalPtr animal = CreateAnimal();
    animal->Talk();
    animal->Move();
    return 0;
}
```

Visitor: операции

Вынесем операции:

```
class TOperation {
  public:
    virtual ~TOperation() {}
    virtual void Apply(const TAnimal &animal) const = 0;
};
using TOperationPtr = std::shared_ptr<TOperation>;

class TTalkOperation : public TOperation {
  public:
    virtual void Aplly(const TAnimal &animal) const override {
    }
};
class TMoveOperation : public TOperation {
  public:
    virtual void Apply(const TAnimal &animal) const override{
    }
};
```

Visitor: операции

Фабрика операций:

```
TOperationPtr CreateOperation() {
    std::string op;
    std::cin >> op;
    if (op == "talk") {
        return static_cast<TOperationPtr>(new TTalkOperation);
    } else if (op == "move") {
        return static_cast<TOperationPtr>(new TMoveOperation);
    }
}
```

А теперь хочется делать так:

```
TAnimalPtr animal = CreateAnimal();
TOperationPtr operation = CreateOperation();
// (animal, operation) -> Apply();  ???
```

Visitor: двойная диспетчеризация

Можно сделать так:

```
class TTalkOperation : public TOperation {
  public:
    virtual void Aplly(const TAnimal &animal) const {
        if (const TCat* cat = dynamic_cast<const TCat*>(&animal) {
            std::cout << "meow" << std::endl;
        } else if (const TDog* dog = dynamic_cast<const TDog*>(&animal)
            std::cout << "woof" << std::endl;
        }
    }
};
```

Но:

- много дублирования,
- важен порядок условий.

Visitor

- В основной иерархии только виртуальный метод `Accept`; Переопределяется в наследниках

```
TCat::Accept(op) { op.Visit(*this); }
```

- В `Operation(Visitor)` определяется набор методов Visit, они все виртуальные и их можно переопределять (Visit(Animal), Visit(Cat), Visit(Dog)).

Visitor

```
class TOperation {
  public:
    virtual ~TOperation() {}
    virtual void Visit(const TAnimal &animal) const = 0;
    virtual void Visit(const TCat &cat) const;
    virtual void Visit(const TDog &dog) const;
};

void TOperation::Visit(const TCat &cat) const {
    Visit(static_cast<const TAnimal&>(cat));
}

void TOperation::Visit(const TDog &dog) const {
    Visit(static_cast<const TAnimal&>(dog));
}
```

Visitor

```
class TTalkOperation : public TOperation {
  public:
    virtual void Visit(const TAnimal &animal) const override {
        std::cout << "unknown operation" << std::endl;
    }
    virtual void Visit(const TCat &cat) const override{
        std::cout << "meow" << std::endl;
    }
    virtual void Visit(const TDog &dog) const override{
        std::cout << "woof" << std::endl;
    }
};

class TMoveOperation : public TOperation {
  public:
    virtual void Visit(const TAnimal &animal) const override {
        std::cout << "unknown operation" << std::endl;
    }
    // ...
};
```

Visitor

```
class TCat : public TAnimal,
             public std::enable_shared_from_this<TCat> {
  public:
    virtual void Accept(const TOperation& operation) const override{
        std::shared_ptr<TCat> p{shared_from_this()};
        operation.Visit(p);
    }
};

class TDog : public TAnimal,
             public std::enable_shared_from_this<TDog> {
  public:
    virtual void Accept(const TOperation& operation) const override{
        std::shared_ptr<TDog> p{shared_from_this()};
        operation.Visit(p);
    }
};
```

Visitor

Использование:

```
int main() {
    TAnimalPtr animal = CreateAnimal();
    TOperationPtr operation = CreateOperation();
    animal->Accept(*operation);
    return 0;
}
```

Двойная диспетчеризация: при вызове `animal->Accept` находится правильный класс `TAnimal` (механизм виртуальных функций), а затем при вызове `operation->visit(*this)` управление передается правильному visitor'у.

Visitor

Дублирование в Accept наследниках можно устранить:

```
template <typename T>
class TFinalAnimal : public T {
  public:
    virtual void Accept(const TOperation& operation) const {
        operation.Visit(*this);
    }
};
```

Идиома Type Erasure

Есть класс, сохраняющий объекты произвольного типа:

```
template <typename T>
class TValue {
    T v;
  public:
    T Get() const;
    template <typename U>
    void Set(U&& val) {v = std::forward<U>(val);}
};
```

Идиома Type Erasure

Есть класс, сохраняющий объекты произвольного типа:

```
template <typename T>
class TValue {
    T v;
  public:
    T Get() const;
    template <typename U>
    void Set(U&& val) {v = std::forward<U>(val);}
};
```

И мы хотим «подписаться» на изменения Set:

```
    TValue<int> v;
    v.DoOnChange(
        [](int newVal) {std::cout << "set " << newVal << std::endl;}
    );
    v.Set(1);
```

Идиома Type Erasure

Абстрактный интерфейс вызова:

```
template <typename T>
class IFunctor {
  public:
    virtual ~IFunctor() = default;
    virtual void Call(const T&t) = 0;
};
```

Идиома Type Erasure

Абстрактный интерфейс вызова:

```
template <typename T>
class IFunctor {
  public:
    virtual ~IFunctor() = default;
    virtual void Call(const T&t) = 0;
};
```

Класс для вызова:

```
template <typename T, typename F>
class TFunctor: public IFunctor<T> {
  private:
    std::decay_t<F> f;
  public:
    TFunctor(F f) : f(std::move(f)){}
    void Call(const T& t) override { f(t); }
};
```

Идиома TypeErasure

Создание объекта, который будет вызывать нужную функцию:

```
template <typename T, typename F>
std::shared_ptr<IFunctor<T>> CreateFunctor(F&& f) {
    return std::make_shared<TFunctor<T, F>> (std::forward<F>(f));
}
```

Идиома TypeErasure

```
template <typename T>
class TValue {
    T v;
    std::shared_ptr<IFunctor<T>> invPtr;
  public:
    T Get() const;

    template <typename U>
    void Set(U&& val) {
        if (val != v) {
            v = std::forward<U>(val);
            invPtr->Call(v);
        }
    }

    template <typename F>
    void DoOnChange(F&& f) {
        invPtr = CreateFunctor<T>(std::forward<F>(f));
    }
};
```

Идиома TypeErasure

Пример: `shared_ptr`. Какой тип стирается?

Идиома TypeErasure

Пример: shared_ptr. Какой тип стирается?

```
template <typename T>
class TSharedPtr {
    struct TDeleterBase {
        virtual void apply(void*) = 0;
        virtual ~TDeleterBase() {}
    };

    template <typename D>
    struct Deleter: public TDeleterBase {
        Deleter(D d) : deleter(d) {}
        virtual void apply(void *p) {deleter(static_cast<T*>(p));}
        D deleter;
    };
    ...
```

Идиома TypeErasure

```
template <typename T>
class TSharedPtr {
    ...
  public:
    template <typename D>
    TSharedPtr(T* ptr, D dlt): p(ptr), d(new Deleter<D>(dlt)) {}

    ~TSharedPtr() {d->apply(p); delete d;}

    T* operator->() {return p;}
    const T* operator->() const {return p;}
  private:
    T* p;
    TDeleterBase* d;

};
```

Проблема перегрузки и универсальных ссылок

Параметр-универсальная ссылка обычно обеспечивает точное соответствие для всего, что бы ни было передано:

```
using TStringSet = std::set<std::string>;
template <typename T>
void Do(TStringSet& strings, T&& str) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}
void Do(TStringSet& strings, int x) {
    std::cout << x << std::endl;
    strings.emplace(std::to_string(x));
}
```

Ломается код:

```
    short x = 2;
    Do(strings, x);
```

Проблема перегрузки и универсальных ссылок

Добавим новую функцию, которая вызывает две другие:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<T>()
  )
}
```

Какая проблема?

Проблема перегрузки и универсальных ссылок

Добавим новую функцию, которая вызывает две другие:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<T>()
  )
}
```

Какая проблема? `std::is_integral<int&>` имеет ложное значение.

Проблема перегрузки и универсальных ссылок

Добавим новую функцию, которая вызывает две другие:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<T>()
  )
}
```

Какая проблема? `std::is_integral<int&>` имеет ложное значение.

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<typename std::remove_reference<T>::type>()
  )
}
```

Проблема перегрузки и универсальных ссылок

Меньше символов с C++14:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<std::remove_reference_t<T>>()
  )
}
```

Проблема перегрузки и универсальных ссылок

Меньше символов с C++14:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<std::remove_reference_t<T>>()
  )
}
```

Диспетчеризация дескрипторов: вызов перегруженных функций «диспетчеризует» передачу работы правильной функции путем создания нужного объекта дескриптора.

Проблема перегрузки и универсальных ссылок

Две перегрузки для DoImpl:

```
template <typename T>
void DoImpl(TStringSet& strings, T&& str, std::false_type /*f*/) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}
void DoImpl(TStringSet& strings, int x, std::true_type /*t*/) {
    std::cout << x << std::endl;
    strings.emplace(std::to_string(x));
}
```

Диспетчеризация дескрипторов

- Решаем проблемы перегрузку и оставляем универсальные ссылки,
- неперегружаемая функция диспетчеризации Do принимает параметр, являющийся универсальной ссылкой,
- перегружаемая impl-функция имеет параметр дескриптора, который спроектирован так, что не существует более одной перегрузки,
- вызов нужной функции определяется дескриптором.