Mxx_ru и Ruby как DSL

Е.А. Охотников

2006.08.02

Аннотация

Статья рассказывает об опыте создания небольшого Mxx_ru-генератора C++ кода. В качестве описаний для данного генератора используется простой Domain Specific Language, реализованный средствами языка Ruby.

1 Задача

В одном из C++ проектов потребовалось строить DLL библиотеку `filters_repository`, в которой располагался код фильтров текстовых сообщений. В один прекрасный момент в данную библиотеку потребовалось включать код нестандартных (custom) фильтров. Эти фильтры представляют из себя статические библиотеки с фабриками объектов некоторого типа. В простейшем случае библиотека содержит реализацию наследников специального типа `filter_t` и функцию `create_filter` вида:

```
std::auto_ptr< filter_t >
create_filter(
  const std::string & cfg_file_name,
  std::string & error_desc );
```

В `filters_repository` каждая подобная функция `create_filter` должна регистрироваться в какой-то глобальной таблице вместе с осмысленным текстовым именем. Например, имя "test::custom_filter::odd"обозначает фильтр, за создание которого отвечает функция `create_filter` из пространства имен `test::custom_filter::odd`. В самом простом варианте DLL должна была содержать функцию поиска фабрики фильтров по имени фильтра, например, такого вида:

```
pfn_filter_factory_t
find_custom_filter_factory( const std::string & factory_name )
  {
    static custom_filter_factory_info_t factories[] =
      {
        { "test::custom_filter::odd",
          &test::custom_filter::odd::create_filter }
      , { "test::custom_filter::even",
          &test::custom_filter::even::create_filter }
      ...
      , { 0, 0 }
      };
    for( const custom_filter_factory_info_t * p = factories;
```

```
14          p->m_factory_name; ++p )
15        if( p->m_factory_name == factory_name )
16          return p->m_factory;
17      return 0;
18    }
```

Особенностью задачи было то, что код `find_custom_filter_factory` нельзя было писать вручную. Список фильтров должен был определяться только во время компиляции на основании информации из конфигурационного файла (список фильтров зависел от конфигурации, в которой компилировался весь проект).

Второй особенностью являлось то, что фильтры располагались в статических библиотеках, но могли нуждаться в дополнительных как статических, так и динамических библиотеках. И все эти дополнительные библиотеки должны были автоматически подключаться к результирующей библиотеке `filters_repository`. Например, если фильтр использует регулярные выражения, то в `filters_repository` должны были попасть используемые им библиотеки для работы с регулярными библиотеками.

Вопрос был в том, как задавать набор необходимых в конкретной конфигурации фильтров и как использовать это описание во время компиляции, чтобы правильным образом формировать код `find_custom_filter_factory` и список дополнительных библиотек для линковки `filters_repository`. Проект компилировался при помощи Mxx_ru поэтому можно было использовать все возможности языка Ruby.

2 Решение

Основные черты решения были понятны изначально, поскольку задача сразу оформилась как генерация фрагмента C++ кода по какому-то конфигурационному файлу. Вероятно, общую задачу построения библиотеки `filters_repository` можно было решить и каким-нибудь иным методом, а не кодогенерацией, но кодогенерация казалась самым простым и гибким способом.

Поэтому основной вопрос состоял в том, как задавать описания фильтров и как внутри Mxx_ru проекта библиотеки `filters_repository` эти описания обрабатывать.

2.1 Описание используемых фильтров

Поскольку Ruby является отличным инструментом для создания Domain Specific Language (DSL), то вместо использования конфигурационных файлов какого-либо формата (XML[3], YAML[4] или JSON[5]) было решено задавать описания прямо в виде Ruby кода:

```
1 # Задает фильтр, реализация которого находится в проекте
2 # test/custom_filter/odd/prj.rb (имя проекта автоматически
3 # вычисляется на основании имени фильтра).
4 custom_filter 'test::custom_filter::odd'
5
6 # Задает фильтр и проектный файл с его реализацией.
7 custom_filter 'test::custom_filter::even',
8   :no_default_project => true do |f|
9   f.required_prj 'test/custom_filter/odd/even.prj.rb'
```

```
end

# Задает фильтр, реализация которого находится в проекте
# test/custom_filter/regex/prj.rb (имя вычисляется автоматически),
# и который нуждается в проекте regex/prj.rb:
custom_filter 'test::custom_filter::regex' do |f|
  f.required_prj 'regex/prj.rb'
end
```

Его реализация тривиальна:

```
FiltersAndProjects = Struct.new( :filters, :projects )

class MapProcessor
  class FilterDescriptor
    attr_reader :projects

    def initialize
      @projects = Set.new
    end

    def required_prj( name )
      @projects << name
    end
  end

  def initialize( map_file_name )
    @names = Set.new
    @projects = Set.new
    @file_name = map_file_name
  end

  def process
    instance_eval( File.read( @file_name ), @file_name )

    FiltersAndProjects.new( @names, @projects )
  end

  # Этот метод будет вызываться из DSL-я для определения очередного
  # custom-фильтра.
  #
  # Если в options нет ключа :no_default_project (со значением true),
  # то для фильтра автоматически формируется одно имя подпроекта: все
  # пары :: заменяются на /. Например, из имени test::custom_filter::odd
  # будет автоматически сгенерировано имя test/custom_filter/odd/prj.rb.
  #
  def custom_filter( name, options = {} )
    fail "#{@file_name}: filter '#{name}' already defined" if
        @names.member?( name )

    d = FilterDescriptor.new
    if !( options[ :no_default_project ] )
      default_project = File.join( name.split( /::/ ), 'prj.rb' )
      d.required_prj( default_project )
```

```
44      end
45
46      yield( d ) if block_given?
47
48      fail "#{@file_name}: filter '#{name}': no required projects defined" if
49          0 == d.projects.size
50
51      @names << name
52      @projects.merge( d.projects )
53    end
54  end # class MapProcessor
```

Для разбора Ruby-файла с описанием фильтров нужно создать объект типа `MapProcessor` и передать ему в конструктор имя файла с описанием. После чего вызвать метод `MapProcessor#processor` который возвращает объект `FiltersAndProjects` с именами фильтров и необходимых проектов. Фактически, инициирование разбора описания выполняется одной строкой:

```
1  parsing_result = MapProcessor.new( @map_file ).process
```

Принцип работы `MapProcessor` состоит в том, что метод `process` загружает содержимое файла описания и выполняет его как Ruby код через собственный метод `instance_eval`. Это ключевой момент. Получается, что как будто все содержимое файла описания просто записано в теле метода `MapProcessor#process`. И вызовы `custom_filter` в описании — это вызовы метода объекта класса `MapProcessor`.

Метод `MapProcessor#custom_filter` сам получает в качестве необязательного параметра блок кода. Этот блок предназначен для перечисления необходимых фильтру подпроектов. Для чего в методе `MapProcessor#custom_filter` создается вспомогательный объект типа `MapProcessor::FilterDescriptor`, который и передается в блок кода в качестве параметра. Блок кода может вызывать у этого объекта метод `required_prj` для каждого необходимого фильтру подпроекта. Например:

```
1  custom_filter 'custom_filter::regex' do |f|
2    f.required_prj 'pcre/prj.rb'
3    f.required_prj 'pcrecpp/prj.rb'
4  end
```

При своем небольшом объеме функция `MapProcessor#custom_filter` выполняет ряд действий по обработке и валидации описаний:

- проверяет уникальность имени фильтра;
- строит из имени фильтра имя проекта, в котором находится реализация фильтра (если только не задан аргумент `:no_default_project`, который как раз указывает, что имя этого проекта будет задано явно);
- проверяет, что хотя бы одно имя необходимого фильтру проекта задано.

Т.е. в небольшом фрагменте Ruby кода реализован разбор и валидация конфигурационного файла с параметрами фильтров.

2.2 Генерация кода функции find_custom_filter_factory

После того, как задача хранения описаний используемых фильтров была разрешена с помощью простого Ruby DSL осталось решить задачу генерации исходного текста функции `find_custom_filter_factory` во время компиляции проекта `filters_repository` и подключения к проекту всех нужных фильтров подпроектов. В Mxx_ru за решение подобных задач отвечают специальные объекты-генераторы, поэтому для создания `find_custom_filter_factory` был создан соответствующий генератор:

```
class CustomFilterFinderGenerator < MxxRu::AbstractGenerator
  DEFAULT_RESULT_FILE = 'filters/find_custom_filter.cpp'
  DEFAULT_MAP_FILE = 'aag_3.smsc_map.custom_filters.rb'

  def initialize(
      target,
      params = { :result_file => DEFAULT_RESULT_FILE,
          :map_file => DEFAULT_MAP_FILE } )
    ...# Обсуждается ниже...
  end

  def build( target )
    generator = FinderGenerator.new( @result_file, @map_file, @filters )
    MxxRu::show_brief( "checking generation necessary: #{@result_file}" )
    if generator.need_generation?
      MxxRu::show_brief( "generating: #{@result_file}" )
      generator.generate
    end
  end

  def clean( target )
    MxxRu::Util::delete_file( @result_file )
  end

  ...
end
```

Класс `CustomFilterFinderGenerator` наследуется от общего базового класса всех Mxx_ru-генераторов, `MxxRu::AbstractGenerator`. Главной целью `CustomFilterFinderGenerator` является реализация собственных методов `build` (генерация исходного текста) и `clean` (удаление результатов генерации при очистке проекта). Но наиболее важные действия класс `CustomFilterFinderGenerator` выполняет в своем конструкторе:

```
def initialize(
    target,
    params = { :result_file => DEFAULT_RESULT_FILE,
        :map_file => DEFAULT_MAP_FILE } )

  # Результирующий файл должен быть добавлен к списку
  # исходных файлов проекта.
  @result_file = target.create_full_src_file_name(
```

```
9       params[ :result_file ] || DEFAULT_RESULT_FILE )
10    target.cpp_source( @result_file )
11
12    # Нужно разобрать описания фильтров.
13    @map_file = params[ :map_file ] || DEFAULT_MAP_FILE
14    filters_and_projects = load_filters_map
15
16    # Имена фильтров нужно сохранить на будущее, а проекты сразу нужно
17    # добавить в список необходимых подпроектов цели.
18    @filters = filters_and_projects.filters
19    target.required_prjs( filters_and_projects.projects )
20  end
```

Конструктор получает два параметра. Первый, обязательный, является объектом-целью (target) Mxx_ru-проекта, в котором `CustomFilterFinderGenerator` будет использоваться как генератор. В конструкторе объект `target` используется для:

- определения полного имени результирующего cpp-файла (который укладывается в подкаталог проекта) посредством обращения к методу `create_full_src_file_name` и для включения результирующего cpp-файла в список исходных текстов проекта (посредством `cpp_source`);
- включения в проект всех необходимых подпроектов с фильтрами и нужными им библиотеками (через вызов `required_prjs`).

Второй параметр не обязателен и служит для передачи генератору необязательных параметров в виде `Hash`. По умолчанию, генератор предназначен для генерации файла `filters/find_custom_filter.cpp`, а описания фильтров задаются в файле `aag_3.smsc_map.custom_filters.rb`. Если `CustomFilterFinderGenerator` используется в конфигурации по умолчанию, то задавать эти значения не нужно, достаточно в проекте указать:

```
1  # Добавляем проекты с custom-фильтрами.
2  generator Aag_3::SmscMap::CustomFilterFinderGenerator.new( self )
```

и все. Но существует необходимость использовать этот генератор и в unit-тестах, в которых имена файлов с результатами генерации и описаниями фильтров могут быть совсем другими. В этих случаях потребуется задавать значение аргументу `params`:

```
1  generator Aag_3::SmscMap::CustomFilterFinderGenerator.new(
2      self,
3      :result_file => 'custom_filter_finder.cpp',
4      :map_file => 'test/smsc_map/routing/custom_filters.rb' )
```

Из-за подобных особенностей аргумента `params` в коде конструктора `CustomFilterFinderGenerator` сделаны проверки наличия в `params` соответствующих значений и, если это не так, используются значения имен файлов по умолчанию.

Загрузка описаний фильтров производится функцией `load_filters_map`:

```
# Загружает содержимое карты custom-фильтров, если карта существует.
# Если карты фильтров нет, то возвращает пустой объект FiltersAndProjects.
def load_filters_map
  # Загрузку производим только, если файл существует.
  if File.exists?( @map_file )
    MapProcessor.new( @map_file ).process
  else
    # В противном случае возвращаем пустые списки.
    FiltersAndProjects.new( [], [] )
  end
end
```

Поскольку отсутствие файла описания фильтров является обычным делом (библиотека `filters_repository` содержит только стандартные фильтры), то `load_filters_map` в этом случае возвращает пустой список фильтров и подпроектов.

Генерация cpp-файла с кодом `find_custom_filter_factory` производится в методе `build` с помощью объекта типа `FinderGenerator`.

```
class FinderGenerator
  def initialize( result_file, map_file, filters )
    @result_file = result_file
    @map_file = map_file
    @filters = filters
  end

  def need_generation?
    MxxRu::TargetState::EXISTS != MxxRu::TargetState.detect(
        @result_file, @map_file ).state
  end

  def generate
    tmp_name = @result_file + '.tmp'
    File.open( tmp_name, 'w' ) do |file|
      generate_header( file )
      generate_custom_filters_prototypes( file )
      generate_filter_finder( file )
      generate_footer( file )
    end

    FileUtils.mv( tmp_name, @result_file, :force => true )
  end

  def generate_header( file ); ...; end

  def generate_custom_filters_prototypes( file ); ...; end

  def generate_filter_finder( file ); ...; end

  def generate_footer( file ); ...; end
end # class FinderGenerator
```

Именно этот объект знает детали кодогенерации.

В реализации `FinderGenerator` интерес представляют всего два момента:

- метод `need_generation?` использует функциональность класса `MxxRu::TargetState` для того, чтобы определить, нуждается ли файл в перегенерации (либо он отсутствует, либо описания фильтров изменились с момента последней генерации).
- в методе `generate` сначала создается временный файл, и только если его генерация прошла успешно, временный файл копируется под именем результирующего.

3 Заключение

В статье был рассмотрен небольшой Mxx_ru-генератор C++ кода, созданный для решения частной нестандартной задачи в проекте, компиляция которого осуществлялась с помощью Mxx_ru. Данный генератор использовал в своей работе описания, сделанные непосредственно в Ruby коде, а не в конфигурационном файле какого-то формата (XML, YAML или JSON). Т.е. была продемонстрирована возможность использования Ruby для создания простых DSL.

Список литературы

[1] `http://www.rubyforge.org/projects/mxx-ru`

[2] `http://www.ruby-lang.org`

[3] `http://www.xml.org`

[4] `http://www.yaml.org`

[5] `http://www.json.org`