Разработка модульных приложений на C/C++ с использованием аннотаций

В моей первой статье я рассказал об использовании препроцессора для организации модульности на уровне исходных текстов в языках C/C++. Вкратце этот способ сводится к написанию специфических метаданных внутри исходников, которые анализируются внешним инструментом и используются для генерации glue-исходников, позволяющих реализовать модульность. Детали реализации описаны в упомянутой статье, поэтому не буду здесь повторяться. В данной статье я пойду чуть дальше и попытаюсь показать, что с помощью метаданных или аннотаций можно реализовать не только модульность, но и некоторые другие полезные фичи. Должно получиться что-то вроде Google Guice или Spring для C (той его части, которая связана с модульностью и аспектами). Отдельно подчеркиваю, что эта статья — дополнение и улучшение первой, поэтому тут я буду говорить не столько технических деталях реализации, сколько о том, как это все выглядит для пользователя. Если эта тема вызовет интерес, то я напишу продолжение с пояснениями о том, как устроено внутри само приложение-конфигуратор.

Универсальные аннотации

Предыдущий подход был основан на использовании директив *#pragma* в качестве способа записи метаданных внутри исходников. Формат этих метаданных был выбран произвольно и затачивался под конкретное применение (описание связи между исходником и абстрактным модулем, с которым он ассоциирован). На самом деле понятие аннотаций несколько более широко. В частности, можно провести параллели с языками вроде Java или C#, где аннотации могут описывать произвольные утверждения о коде, в котором они содержатся, поэтому было бы удобно использовать некоторый универсальный формат.

В качестве примеров существующих реализаций подобного

подхода можно привести штуки типа Keil configuration wizard, которые позволяют описывать в виде XML различные переменные внутри файла (например, какие-то константы, размеры массивов и т.п.). Затем, прямо в редакторе, можно перейти на вкладку configuration и настраивать эти опции в графическом виде, а не искать их внутри файла. Помимо этого, в исходниках часто имеется и другая информация, которую можно было бы назвать аннотациями (например то, что пишется внутри __*attribute*__ в GCC, всевозможные __*declspec* и тому подобное).

По ряду причин было решено отказаться от записи аннотаций в комментариях, поскольку последние, по моему глубокому убеждению, должны всегда выполнять только одну функцию. Как известно, " so much complexity in software comes from trying to make one thing do two things ". С другой стороны, использование *#pragma* также сопряжено с проблемами: неизвестные *#pragma* вызывают предупреждение, что при большом числе файлов в проекте выглядит не очень красиво: (несколько предупреждений в каждом файле). Чтобы этого избежать было принято решение обернуть аннотации в макрос, который при компиляции отображать в пустой. Если какой-то компилятор вдруг начнет поддерживать такой формат для своих внутренних нужд и потребуется *#pragma*, то стандарт C99 наконец-то вводит возможность определения *#pragma* через макрос, так что макрос в этом смысле более универсальное решение.

Второй вопрос был связан с форматом самих аннотаций, записываемых внутри макроса. Хотя, в последнее время, все большее распространение для данных задач получает XML. Он используется, помимо приведенного выше примера с Keil, еще и в проекте FreeRTOS (тоже в комментариях). Надо сказать, что XML все-таки не очень хорош по эстетическим соображениям — он довольно многословен, поэтому человеку не очень удобно работать с информацией в таком формате. В то же время, существует машинночитаемый язык разметки, который адаптирован для чтения и записи человеком, а также является совместимым с синтаксисом C — это JSON. Однако, необходимость заключать в кавычки ключи приводила к не очень красивому внешнему виду аннотаций в коде, поэтому был использован

расширенный вариант — YAML. Следует особо отметить, что из всех возможностей YAML используется ТОЛЬКО возможность писать ключи без кавычек. Аннотации, перед тем как передаваться парсеру, всегда преобразуются в одну строку, поэтому возможности YAML, связанные с переносом строк, использовать нельзя. Поскольку JSON является подмножеством YAML, то можно все писать и на чистом JSON, возможность опустить кавычки у ключей следует рассматривать как маленький приятный бонус.

Таким образом, в сухом остатке, мы имеем следующее: исходники на C могут содержать макрос **FX_METADATA(x)** (с одним аргументом), внутри которого записываются аннотации в формате JSON. Аннотации заключены в скобки, для того, чтобы избежать ошибок в случае использования закрывающей скобки внутри самих аннотаций. По соглашению, аннотации всегда представляют собой набор пар ключ-значение, то есть, в терминах JSON, представляют собой хэш. Пример:

```
FX_METADATA(( { annotation: "hello world!" } ))
```

Как вы уже догадались, прагмы из первой статьи были «переведены» в новый формат, и теперь то, что раньше записывалось как:

```
#pragma fx interface MY_INTERFACE:MY_IMPLEMENTATION
```

Теперь записывается так:

```
FX_METADATA(( { interface: [MY_INTERFACE, MY_IMPLEMENTATION]
 } ))
```

По части реализации модульности, изменения чисто косметические, просто все то, что раньше записывалось как попало, стало записываться в более-менее универсальном формате.

Что касается #include, то все осталось почти без изменений: в качестве аргумента по-прежнему используется макрос FX_INTERFACE, только теперь он имеет всегда один аргумент — всегда подразумевается включение интерфейса по-умолчанию, без возможности указать какой-то конкретный интерфейс.

Подытожим: Исходники содержат аннотации, в которых описывается принадлежность исходника к какому-либо модулю. Модуль — это набор исходников, содержащий один заголовочный файл (описывающий интерфейс) и произвольное количество файлов исходных текстов (*.c, *.cpp и т.д.). Информация о принадлежности файлов к модулям извлекается на этапе до компиляции. После этого, внешним приложением генерируется файл, который отображает имена заголовочных файлов на имена модулей, что и позволяет *#include* работать в терминах модулей, а не просто имен файлов, как это происходит обычно. Поскольку у нас есть способ узнать, какие модули должны войти в состав системы, а каждый исходник при этом содержит информацию о принадлежности к модулям, то мы можем автоматически узнать и все файлы, которые нужно скомпилировать. То есть, написав в каком-то исходнике **#include FX_INTERFACE(MY_MODULE)**, получим, что в систему включатся все исходники, которые содержат метку о принадлежности к интерфейсу MY_MODULE.

Конфигуратор

Естественно, то, что в управляемых языках делает *runtime*, здесь тоже должен кто-то делать, поэтому я расскажу немного о приложении, которое это реализует. Фреймворк работы с аннотациями реализован на C# как набор классов (DLL). Вопрос о том, стоило ли его писать именно на C# пока оставим за кадром. Фреймворк создавался таким образом, чтобы использование метаданных не было ограничено только сборкой. Помимо прочего, он предоставляет интерфейсы для того, чтобы доступ к хранилищу метаданных мог быть получен и извне. Это открывает возможности для создания сторонних модулей, которые, на основе метаданных, могут делать те или иные действия. В перспективе, все плагины планируется подключать через MEF и отвязать их от структуры проекта.

Помимо набора DLL, существуют также два приложения, реализующих функции конфигуратора: консольное и графическое. Последнее находится в сильно прототипном состоянии и рассматривать его я сейчас не буду, консольное же будет рассмотрено вкратце.

Конфигуратор имеет 7 ключей командной строки, причем обязательными из них являются только 3, список путей к папкам исходников, целевой модуль (тот, с которого начнется разрешение зависимостей) и выходной файл/папка. Например, если наши аннотированные исходники лежат в папках c:\src1 и d:\src2, и нужно собрать модуль MY_MODULE, то команда будет выглядеть примерно так:

```
fx_mgr -p c:\src1,d:\src2 -t MY_MODULE -o output_folder
```

Выполнение команды приведет к тому, что из исходников будут извлечены метаданные, разрешены зависимости начиная с модуля

MY_MODULE, определены все файлы, которые надо скомпилировать, после чего будет сформирован вывод файлов для компиляции.

Существует два способа «вывода».

Первый способ

Он подходит для небольших проектов, а также удобен для поставки чего-либо в виде исходных текстов. Когда в качестве аргумента с ключом **-o** была указана существующая папка, в этом случае все нужные файлы будут скопированы в указанную папку, причем заголовочные файлы будут переименованы в соответствии с названием реализуемого ими интерфейса. Так как система может содержать только одну реализацию каждого интерфейса, то конфликта имен не возникнет и при размещении файлов для компиляции в плоском виде в одной папке, а макрос **FX_INTERFACE** (аргумент *#include*) может быть определен очень просто:

```
#define FX_INTERFACE(i) <i.h>
```

Никаких дополнительных файлов не требуется. В embedded довольно часто приходится использовать библиотеки поставляемые в виде исходников, из-за необходимости компилировать их со специфическими ключами компилятора. Такой подход с плоским набором файлов упрощает как использование make, так и добавление файлов в какую-либо IDE и т.д.

Второй способ

Разумеется, каждый раз что-то копировать при сборке это плохой вариант, поэтому существует еще один способ: если то, что было указано с ключом **-o**, не является папкой, то создается новый файл с указанным именем (либо перезаписывается существующий) и в него помещается список абсолютных путей к файлам, которые нужно скомпилировать. После этого данный файл можно использовать в системе сборки (в том числе в make или MSBuild) для компиляции файлов. Здесь возникает вопрос, если файлы содержат аннотации, а include используют модули, откуда же возьмутся макросы FX_INTERFACE/FX_METADATA? Действительно, ниоткуда. Для их получения используется ключ **-h**. Если он указан, то создается т.н. общий заголовочный файл, тот самый, который содержит отображения имен файлов на имена модулей, а также определения FX_INTERFACE и FX_METADATA. Данный файл должен быть принудительно включен директивой компилятора во все компилируемые файлы, если используется такой тип сборки.

Это все были основы, которые были реализованы и ранее и прекрасно работали на *#pragma* без универсальных аннотаций, теперь рассмотрим вкратце внедрение зависимостей, и перейдем к вопросам, для которых уже необходимо наличие структурированных метаданных.

Внедрение зависимостей

Стоит уделить пару слов необходимости поддерживать несколько реализаций одного интерфейса. Изначально весь фреймворк вырос из системы конфигурирования для RTOS, которая должна была идеально подстраиваться под требования приложения. Настолько идеально, что какой-то одной реализации каждого интерфейса могло быть уже недостаточно. Самой ОС, как единой сущности, не существовало, она строилась из более мелких блоков. В таком случае делалось несколько реализаций модуля, каждая из которых «затачивалась» под определенные требования. Затем, поскольку интерфейсы у всех реализаций

одинаковые, можно было подбирать идеальные для приложения реализации каждого интерфейса. Иными словами, это статическое внедрение зависимостей в C на уровне исходников. Исходники в дереве проекта не содержат никакой информации о том, какие фактически модули будут использованы, они импортируют абстрактные интерфейсы без указания реализаций. Фактические реализации указываются конфигуратору на этапе сборки системы в виде файла, в котором описана эта информация, по аналогии с тем, как в «энтерпрайз»-языках она записывается во внешних XML (при желании, ничего не мешает и здесь написать такие XML). Наконец, пользуясь теми же метаданными можно определить какие-то дополнительные признаки, по которым конфигуратор мог бы сам определить, какие реализации нужно использовать.

Если исходники содержат только одну реализацию каждого интерфейса, то конфигуратор сам понимает, что альтернативы нет и в случае если данный интерфейс кем-то импортирован, автоматически использует единственный имеющийся модуль с указанным интерфейсом. Если же у каждого интерфейса множество реализаций, то, в отсутствии подсказок, конфигуратор выбирает первую попавшуюся реализацию (и выдает в консоль warning о произошедшем). Так как для данной конфигурации системы обычно известно, какие реализации интерфейсов надо использовать, конфигуратору можно дать подсказку, она дается с помощью ключа **-a** <*имя файла*>. Указанный файл содержит строки вида INTERFACE = VERSION, где INTERFACE это имя интерфейса, а VERSION — имя реализации, которую нужно использовать. Это все близко к тому, что было описано в первой части, только немного упрощен синтаксис, так что за техническими подробностями лучше обратиться к первоисточнику.

Аспекты

Ну, вот и подобрались к самому интересному, ради чего все и затевалось. Существует множество определений аспектов в

программировании, поэтому я не буду пытаться объять необъятное в одной короткой заметке. Аспект это часть чего-то, что должно собираться в зависимости от модулей, входящих в проект. Например, если есть несколько типов каких-то объектов, иногда очень хочется иметь enum, содержащий все эти типы, причем при добавлении нового типа (просто включения файла в проект), этот enum должен автоматически расширяться. Может потребоваться узнать количество каких-то модулей, входящих в данный проект и так далее.

Очевидно, что никакими ухищрениями с препроцессором и кодом достичь этого не удастся, т.к. информация о модулях которые входят в данный конкретный проект недоступна даже компилятору. Наверно, лучше тут начать с предыстории: среди требований к ОС была возможность подстраивать таблицу системных вызовов под имеющийся в ОС набор сервисов, при этом, если какой-то модуль включался в сборку и имел функционал, экспортируемый в user-mode, таблица системных вызовов должна была это учесть. Это все не новость, подобные методы используются в некоторых ОС (например, в NuttX), однако, как правило, все это делается на препроцессоре. Если какой-то модуль включен, то определен некоторый *#define*, если он определен, то работает *#ifdef*, в котором перечислены функции этого модуля внутри таблицы. Сама таблица, таким образом, состоит из *#ifdef*-ов. Данный подход, помимо затрудненной поддерживаемости, имеет также такую проблему, что если появляется новый модуль, то соответствующие ему функции надо вручную добавить в эту таблицу, и таблица, таким образом, содержит знания обо всем, что только может появляться в системе. Аспекты призваны решить проблему, когда для добавления чего-то в проект, нужно сделать множество мелких правок в разных местах: расширение enum-ов, добавление записей в конфигурацию системы сборки и так далее.

В предлагаемом конфигураторе реализован другой подход. Раз уж у нас есть модули, почему бы не сделать некоторые модули генерируемыми самим конфигуратором? Ведь он обладает знаниями о структуре системы и о метаданных всех модулей, после того, как было произведено внедрение зависимостей.

Исходники содержат в качестве специального ключа аннотаций набор пар вида «ключ-массив значений». После разрешения конфигуратором всех зависимостей, становятся известны модули, входящие в данную конфигурацию. Далее все эти модули исследуются на предмет наличия в них ключа aspects, и если таковой найден, то создается #define с именем ключа, в котором перечисляются все значения (из всех модулей в системе). То есть, если один из модулей содержит например вот такую запись:

```
FX_METADATA(({ aspects: [  { key: [  val1, val2, val3 ] } ] }))
```

А другой, вот такую:

```
FX_METADATA(({ aspects: [  { key: [  val4, val5, val6 ] } ] }))
```

То будет сгенерирован #define вида

```
#define key val1 val2 val3 val4 val5 val6
```

Все значения (и ключи и значения) рассматриваются как текст, поэтому можно написать, например, так:

```
FX_METADATA({ aspects:
 [  { key: [  "val1,"  ,  "val2,"  , "val3," ] } ] }))
```

И тогда значения внутри #define будут идти через запятую. Ну, вы поняли идею.

Где будет сгенерирован этот #define? В заголовочном файле, который указывается конфигуратору в качестве ключа -e <имя файла>. Если указан этот ключ, то конфигуратор добавляет сгенерированный интерфейс в хранилище метаданных. Этот интерфейс называется CFG_ASPECTS и доступен для включения как #include FX_INTERFACE(CFG_ASPECTS), там содержатся аспекты из всех модулей. Важно отметить, что если такой интерфейс уже есть (то есть пользователь уже написал его вручную и он содержится в пуле исходников после их анализа), то генерироваться ничего не будет, а будет использоваться имеющийся модуль.

Опции

Опции — это некоторые внешние #define, которые определяют некоторые конфигурационные константы для данного модуля. По аналогии с тем, что у Keil, только не в терминах файлов, а в терминах модулей. Основное их предназначение — установка этих параметров через GUI, поэтому использовать опции в консольном приложении довольно затруднительно. Как и в случае с аспектами, опции представлены генерируемым интерфейсом CFG_OPTIONS, который содержит все опции для данной конфигурации модулей, которые были описаны в виде соответствующих аннотаций внутри исходников. Консольное приложение генерирует файл с опциями установленными в

значения по-умолчанию, графическое же приложение позволяет их задавать.

Модель использования опций предполагалась следующей: пользователь получает ОС в виде набора исходников, запускает конфигуратор, который анализирует эти исходники и показывает имеющиеся опции, пользователь их настраивает под свои нужды, и конфигуратор генерирует тот самый файл CFG_OPTIONS. После этого у пользователя есть полный набор необходимых сконфигурированных исходников. То есть в контексте разработки от них в общем-то толку немного, поэтому на этом я думаю стоит закончить.

Ограничения

В текущей версии используется упрощенная модель извлечения метаданных — из файла просто исключаются комментарии (регулярными выражениями) и дальше там производится поиск include и метаданных (тоже регулярными выражениями). В 98% случаев этот подход работает безотказно, но в каких-то сложных случаях, вроде переопределения макросов и вкючаемых файлов, он работать не будет. В этом случае надо использовать полноценный поставщик метаданных, который препроцессит каждый файл (сишным препроцессором) и извлекает метаданные и включаемые файлы уже со 100% гарантией, даже в случае любых трюков с препроцессором. Но, разумеется, в случае большого количества файлов, это будет довольно медленно (секунды для проекта в несколько сотен файлов, что при отладке довольно утомительно).

В текущей реализации поддерживается параллельное извлечение метаданных: так как эта операция идеально распараллеливается, извлечение и парсинг метаданных происходит с использованием цикла Parallel.ForEach, который работает на основе пула потоков, так что, при наличии нескольких процессоров, эта операция серьезно ускоряется.

Также есть небольшая сложность с использованием IDE, дело в том, что сишные IDE предполагают, что все файлы, которые находятся в текущем проекте, нужно компилировать. В предлагаемом подходе, конфигуратор, по списку путей к папкам, проекта сам определяет, какие файлы должны быть скомпилированы, поэтому если IDE не позволяет как-то фильтровать входные файлы (с помощью какого-то плагина, который мог бы в момент билда сказать, что надо, а что не надо компилировать), то использовать такую IDE будет несколько проблематично. Я сам IDE не пользуюсь, в качестве редактора использую Sublime Text или Visual Studio Code, а сборка настроена так, чтоб запускаться из командной строки. К тому же RTOS поставляется в виде исходников для конкретной конфигурации, поэтому с ней проблем нет, но тот, кто не может жить без IDE, может быть разочарован, это да. По сути, вводится дополнительная фаза сборки, выполняющаяся до компиляции — фаза конфигурирования, и не все IDE совместимы с этим подходом.

Заключение

Прошу прощения за некоторую сумбурность изложения, аннотации, аспекты и модульность это большие отдельные темы, которые очень сложно объять и полностью раскрыть в одном посте, поэтому я рассматриваю данный пост не как нечто законченное, а как начало дискуссии. Если кому-то описываемые вещи показались интересными, то можно перейти к следующему этапу и посмотреть на примеры использования и исходники конфигуратора. Все это доступно на гитхабе. На этом пока все, всем спасибо за внимание.