Санкт-Петербургский государственный университет

Кафедра системного программирования

Кита Михаил Евгеньевич

Генерация фрагментов исходного кода с помощью статистических методов

Курсовая работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Санкт-Петербург
2017

Оглавление

Введение **3**

1. Постановка задачи **4**

2. Обзор существующих решений **5**

3. Данные **8**

4. Алгоритм **10**

4.1. Сортировка последовательности 10

4.2. Генерация кода 11

5. Результаты **14**

Заключение **15**

Список литературы **16**

Введение

С каждым годом сложность разработки программных продуктов возрастает, поэтому всё более востребованными становятся инструменты, облегчающие разработку и повышающие производительность труда программистов. Однако до сих пор в случае возникновения затруднений разработчики вынуждены тратить много времени на поиск подходящего решения для конкретной задачи.

Исследование [1] выявило, что исходный код в значительной степени повторяется. И это неудивительно: при решении какой-либо практической задачи разработчики зачастую используют схожие последовательности команд языка. Отсюда следует, что по текстовому описанию задачи можно предсказать, какие функции потребуются для её реализации. На основе этой информации можно предоставлять целые фрагменты кода по запросу.

Инструмент, умеющий производить такую работу, будет весьма полезен в разработке. Во-первых, он упростит и ускорит написание кода, поскольку разработчикам не нужно будет проводить много времени в поисках решения. Во-вторых, поможет улучшить качество самого кода, рекомендуя наилучший доступный вариант.

Описанную выше задачу можно разделить на две существенных части. Первая часть заключается в построении последовательности методов, необходимых для решения сформулированной пользователем задачи, по её текстовому описанию. Вторая — предполагает генерацию исходного кода по готовой цепочке вызовов API (API usage sequence), полученной в качестве результата первой части. Обе части достаточно независимы, и могут разрабатываться отдельно друг от друга.

В рамках данной работы рассматривается именно вторая часть. Будут исследованы различные подходы для реализации такого рода задач с целью их дальнейшего улучшения. В качестве итога планируется создать рабочий прототип, позволяющий генерировать фрагменты исходного кода на основе последовательности вызовов API.

1. Постановка задачи

Целью данной работы является создание модуля для генерации фрагментов исходного кода на языке C# по набору вызовов API. Для её достижения были поставлены следующие задачи:

- реализовать инструмент для сбора данных;
- собрать данные, необходимые для работы алгоритма;
- реализовать алгоритм генерации кода.

2. Обзор существующих решений

API mining — интересная и относительно свежая область исследований. Научные коллективы на протяжении уже нескольких десятилетий занимаются разработками в этом направлении, накопив на текущий момент большой багаж знаний. Основная цель подобных исследований — упростить изучение библиотек и фреймворков, предлагая список наиболее подходящих методов для решения возникающих задач. Однако сам по себе список методов не слишком удобен для работы. Хотелось бы иметь возможность получать на выходе алгоритма фрагмент исходного кода, в который, возможно, нужно будет внести незначительные изменения, чтобы он заработал.

Уже в ранних исследованиях упоминается возможность генерации кода по цепочке вызовов. Так, в [6] авторы в качестве основных целей будущих работ рассматривают синтез фрагментов кода. Тем не менее, долгое время публикаций по данной теме не было. Лишь в последние годы стали появляться исследования, где в качестве конечного результата генерировался фрагмент исходного кода.

Одной из первых таких работ стал SWIM [4]. Алгоритм умел не только строить цепочки методов, но также и генерировать код. Эту задачу выполнял специальный модуль, который авторы назвали синтезатором кода. В его основе лежала модель структурированных последовательностей вызовов: каждая последовательность содержала описание того, какие методы и в каких конструкциях языка используются. Данные для модели были собраны из более чем 25000 open-source проектов.

По текстовому запросу пользователя алгоритм строил цепочку вызовов API, которая затем поступала на вход синтезатору кода. Он принимал решение, каким образом объединить методы из входной цепочки, чтобы получить корректный код. Затем синтезатор, используя заранее собранные данные, подбирал наиболее подходящие структурированные последовательности вызовов и восстанавливал из них код. Отдельное внимание было уделено именованию переменных: по возможности имена подбирались в соответствии с хранимыми данными.

SWIM генерировал код с достаточно высокой точностью. Тем не менее, используемая в исследовании модель была намеренно ограничена ввиду сложности работы с некоторыми конструкциями языка. Таким образом, SWIM позволял решать лишь некоторые несложные задачи.

Следующим шагом стал алгоритм T2API [5]. Основные идеи, использованные для его реализации, остались прежними, однако в основе алгоритма кодогенерации была задействована совершенно иная модель. В отличие от предыдущих исследований она имела графовую структуру, что значительно лучше подходит для исходного кода.

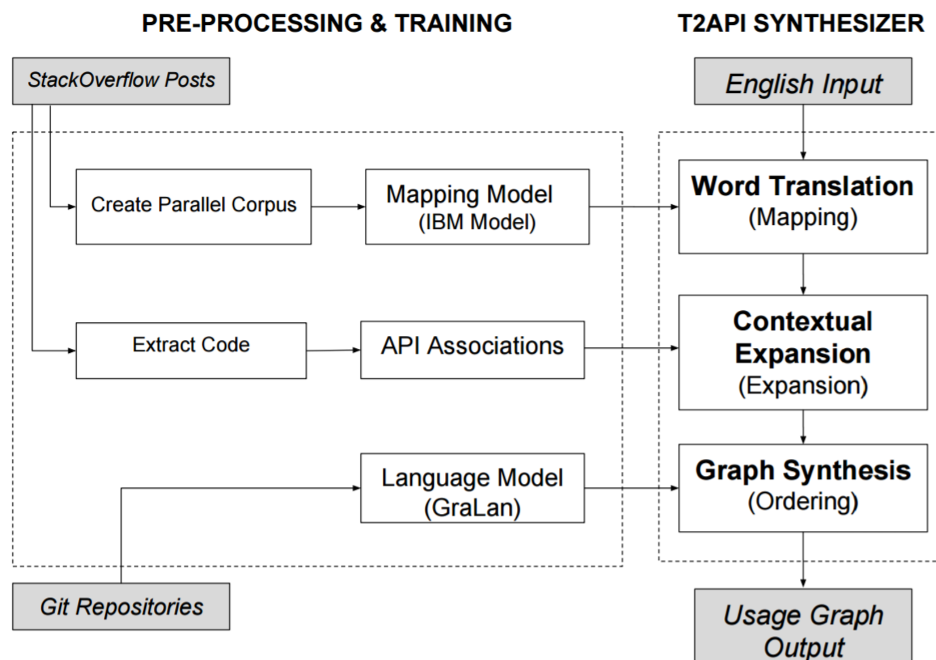

Рис. 1: Архитектура T2API

Архитектура T2API изображена на Рис. 1. Наиболее интересен нам будет модуль, который занимается синтезом графов (Graph Synthesis). Для работы он использует модель GraLan [3], которая позволяет представлять цепочку вызовов API в виде графа. Каждый элемент входной цепочки постепенно добавлялся в граф, занимая наиболее подходящее для него место. Модель вычисляла вероятность добавления нового узла к уже имеющимся на основе данных, собранных из большого количества проектов с GitHub. Полученный после завершения данного процесса граф отражал структуру будущего фрагмента кода.

Благодаря использованию новой модели и улучшенному распознаванию контекста входного запроса генерируемый код стал более релевантным конкретной задаче. Новые подходы также позволили решать более сложные задачи в сравнении со SWIM.

Таким образом, результаты исследований в области генерации фрагментов кода уже сейчас показывают свою высокую эффективность. Однако в силу новизны области остаётся широкий простор для исследования способов улучшить существующие решения.

3. Данные

Перед началом основной работы необходимо было собрать достаточное количество данных. Авторы предыдущих исследований извлекали исходный код из проектов с GitHub, Stack Overflow и других открытых источников. Несмотря на простоту и очевидность данного метода, он имеет ряд недостатков. Как известно, в языке C#, начиная с версии 3.0, переменные могут иметь неявный тип var, поэтому для выведения явного типа таких переменных требуется компиляция собранных проектов. В силу некоторых причин, сделать это автоматически может быть затруднительно. Также открытым остаётся вопрос о качестве собранного таким способом кода.

Учитывая всё сказанное, решено было использовать принципиально другой метод. Необходимые данные извлекались из готовых сборок, которые в большом количестве доступны в репозитории NuGet [2]. При таком подходе не возникает проблем с типами, поскольку они уже в явном виде представлены в библиотеках. Используя NuGet Package Manager Console, встроенный в Visual Studio, был получен список всех доступных пакетов в порядке убывания популярности: предполагалось, что более популярные пакеты должны содержать более качественный код. Пакеты скачивались в автоматическом режиме, после чего содержащиеся в них библиотеки собирались в одну директорию, чтобы в дальнейшем обеспечить разрешение зависимостей. В случае возникновения коллизии имён предпочтение отдавалось более новой версии библиотеки.

Далее следовал процесс обработки собранных файлов. Для чтения информации, содержащейся в библиотеках, использовался Mono.Cecil. Этот инструмент позволял получить доступ ко всем классам, полям и методам из библиотеки. Для каждого найденного метода средствами ICSharpCode.Decompiler строилось синтаксическое дерево, после чего производился его разбор. Из собранной таким образом информации извлекались последовательности вызовов, которые затем сохранялась на диск для дальнейшего использования.

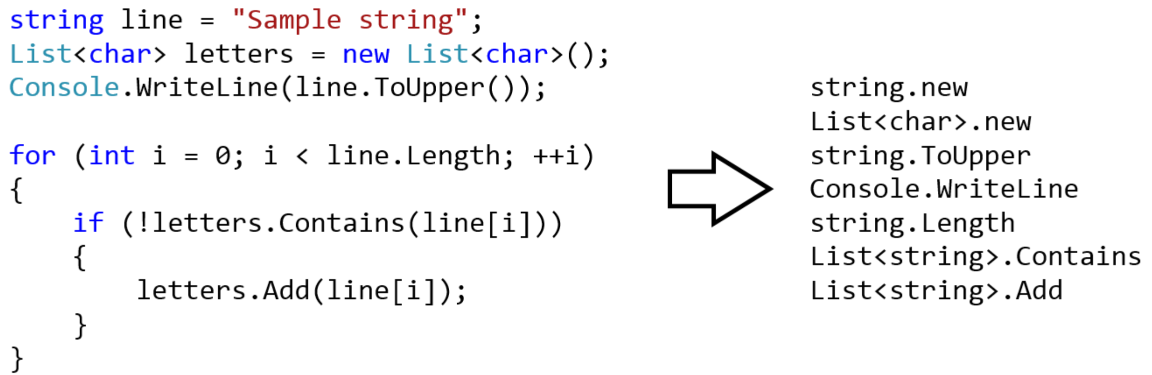

Рис. 2: Процесс извлечения данных на примере одного метода

Как показала практика, описанный метод сбора данных оказался достаточно эффективным. Было скачано 4100 пакетов, содержащих в общей сложности 4278 библиотек. Итоговое количество обработанных методов составило 611945, что сопоставимо с объёмом данных, использованных при реализации модели GraLan.

Сам же разработанный модуль получился универсальным: с небольшими доработками он сможет получать различную дополнительную информацию, что может пригодиться для решения ряда других задач. Так, например, можно извлекать комментарии к методам из xml-файлов, поставляемых вместе с библиотеками. Тем не менее, скорость обработки данных на текущий момент оставляет желать лучшего, что показывает необходимость оптимизации разработанного решения.

4. Алгоритм

В качестве отправной точки исследования решено было повторить эксперимент, проведённый в T2API, а затем исследовать возможности его улучшения. В оригинальной статье авторы используют язык Java и упоминают, что работа только с одним языком программирования может дать необъективные результаты. Чтобы убедиться в их достоверности, основным языком данной работы стал C#.

Алгоритм можно разделить на две основные части. Сначала входная цепочка сортируется, после чего производится синтез фрагмента. В следующих разделах каждая часть рассматривается подробнее.

4.1. Сортировка последовательности

Перед началом генерации кода входную цепочку требуется упорядочить. Это необходимо, чтобы обеспечить правильную с точки зрения языка последовательность команд. Так, например, инициализация объекта всегда должна предшествовать его использованию.

Как уже говорилось ранее, авторы T2API представляли цепочку вызовов в виде графа управления. Однако в плане реализации такая модель оказалась значительно более сложной, чем ожидалось изначально. Возникало множество проблем, связанных с хранением и обработкой данных. На этапе сбора данных извлечённые цепочки необходимо было сохранять в виде графов, для чего требовалась специально созданная база данных. Проблемы возникали и при дальнейшей обработке: согласно алгоритму для графов из базы необходимо было искать подграфы, изоморфные заданному, что само по себе является вычислительно сложной задачей. По утверждению авторов для решения этих проблем они использовали эффективные алгоритмы поиска и хранения.

Учитывая описанные причины, пришлось отступить от оригинальной статьи. Входные цепочки оставались линейными, благодаря чему работа с большим объёмом собранных данных заметно облегчилась. Эксперименты показали, что даже упрощённая модель позволяет расположить связанные вызовы в верном порядке.

Сам же алгоритм довольно прост. Изначально все элементы входной цепочки добавлялись в список рассматриваемых. Используя собранные данные, на каждой итерации выбирался следующий элемент списка, который будет добавлен в итоговый набор. Для этого оценивалась релевантность всех последовательностей из набора данных. Последовательность считалась тем более подходящей, чем больше в ней содержится элементов из списка и чем меньше — посторонних элементов. На основании этого формула вычисления оценки j-ой последовательности была следующей:

$$relevance(S_j) = \frac{|L \cap S_j|}{|S_j|}, \tag{1}$$

где L — текущий список рассматриваемых элементов, S_j — j-ая последовательность из набора данных. Полученное значение прибавлялось к результату того элемента из списка, который встретился в S_j первым. Элемент, набравший наибольшую сумму баллов, добавлялся в итоговый набор и удалялся из списка рассматриваемых. Алгоритм завершался, когда список становился пустым.

Стоит также заметить, что на каждой итерации фактическая работа производилась не со всем набором данных, а лишь с небольшой его частью — участвовали только те последовательности, которые содержали хотя бы один вызов из входной цепочки. Чтобы увеличить скорость работы алгоритма, из общего набора выделялось указанное подмножество, и вся дальнейшая работа производилась с ним.

4.2. Генерация кода

После подготовки последовательности можно было перейти генерации кода. Однако, как выяснилось, данный процесс не был в достаточной мере освещён в статьях. Например, авторы не упоминают, каким образом из входной цепочки вызовов может быть получена информация об управляющих конструкциях языка (условные операторы, циклы, и т.д.). Это важное упущение, поскольку без такого рода информации получаемый в результате код будет строго линейным.

Чтобы решить данную задачу, потребовалось улучшить данные. Для каждого вызова была добавлена информация о том, в какой конструкции он находится и содержит ли аргументы. Также был доработан алгоритм сортировки: после выбора наиболее подходящего элемента входной цепочки набор данных просматривался ещё раз, чтобы определить, внутри какой конструкции разместить текущий элемент. Кроме того, на основе собранной статистики добавлялась информации о наличии аргументов у данного элемента. После завершения работы алгоритма получалась упорядоченная последовательность, расширенная дополнительной информацией.

Далее следовал непосредственно процесс синтеза фрагмента. Прежде всего обрабатывалась информации об управляющих конструкциях. Определялось, где должен располагаться текущий элемент, и при необходимости добавлялись нужные конструкции. На этом этапе возникла интересная задача — синтез логических выражений для условных операторов и циклов. В общем случае создание значимых условий потребует глубокого анализа семантики, поэтому на текущий момент данная проблема была решена лишь частично. Сейчас логические выражения для циклов задаются по умолчанию, а для условного оператора остаются пустыми.

После определения контекста для каждого элемента создавались соответствующие ему строки кода. Если текущий элемент отвечал за создание нового объекта, то для него генерировалась переменная, имя которой подбиралось на основе имени класса объекта. Чтобы избежать использования уже занятых идентификаторов, поддерживался список ранее созданных переменных, а также список ключевых слов языка. При возникновения коллизии к имени новой переменной приписывались дополнительные символы. После подбора подходящего имени объявление переменной добавлялось в код фрагмента. Если же рассматриваемый элемент представлял собой инструкцию, то имя класса в нём заменялось на имя соответствующей переменной из списка (она должна найтись, если цепочка была упорядочена верно). Готовая инструкция также добавлялась во фрагмент.

```
string varString = new string();
StreamReader streamReader = new StreamReader();
Queue<string> queue = new Queue<string>();

while (true)
{
    streamReader.ReadLine();

    if (...)
    {
        streamReader.Close();
    }

    queue.Enqueue(...);
}
```

Рис. 3: Пример сгенерированного фрагмента кода.

На Рис. 3 представлен готовый фрагмент кода, полученный с помощью описанного алгоритма. Как можно заметить, он содержит некоторое количество пропусков, однако достаточно легко понять, как сделать его корректным.

5. Результаты

После завершения работы необходимо было оценить точность алгоритма. Поскольку результат его работы представляет собой фрагмент кода, возникли некоторые сложности: две реализации одной задачи могут значительно различаться структурой и порядком команд, даже если они используют общий набор вызовов API. Поэтому для оценки точности работы алгоритма был применён эвристический метод.

В качестве тестовых данных использовалась выборка небольших фрагментов кода с MSDN для решения наиболее распространённых задач. В их числе работа со стандартными коллекциями, файловой системой, строками и регулярными выражениями. Из собранных фрагментов извлекались цепочки вызовов, которые поступали на вход алгоритму. Затем полученные фрагменты сравнивались с исходными, при этом учитывался порядок следования команд и управляющих конструкций, а также контекст, в котором они расположены. Отличия в порядке следования инструкций, не нарушающие общую логику работы фрагмента, считались верными. Результат работы вычислялся как отношение верно расположенных команд ко всем командам в сгенерированном фрагменте. Итоговым результатом стало среднее значение полученных оценок, составившее 58.63%.

Помимо точности важно было также измерить время работы. Измерение проводилось на той же выборке и учитывало время от ввода запроса до получения готового фрагмента. В среднем для генерации одного фрагмента требуется 0.231 секунды, что достаточно для практического использования алгоритма.

Заключение

В ходе работы достигнуты следующие результаты:

- создан инструмент, позволяющий автоматически собирать данные для статистических задач, связанных с исходным кодом;
- собран набор данных, достаточный для работы алгоритма;
- реализован алгоритм, позволяющий по цепочкам вызовов API генерировать фрагменты кода.

Дальнейшее направление работы

Алгоритм показал достаточно хорошие результаты, однако он далёк от совершенства. Уже сейчас есть множество идей его улучшения. Так, например, для достижения более высоких результатов можно увеличить количество собираемой информации и усложнить модель. Стоит также отметить, что текущая реализация сложна для использования другими разработчиками. В качестве финального результата хотелось бы видеть расширение для популярной IDE, реализующее алгоритм в удобной для использования форме.

Список литературы

[1] Gabel M., Su Z. A study of the uniqueness of source code // Proceedings of the 18th ACM international symposium on Foundations of software engineering, ser. FSE '10. — ACM, 2010. — P. 147 – 156.

[2] Microsoft. NuGet Gallery. — 2017. — URL: `http://www.nuget.org` (online; accessed: 13.05.2017).

[3] Nguyen Anh Tuan, Nguyen Tien N. Graph-based Statistical Language Model for Code // Proceedings of the 37th International Conference on Software Engineering. — IEEE, 2015. — P. 858 – 868.

[4] Raghothaman M., Wei Y., Hamadi Y. SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis // Proceedings of the 38th International Conference on Software Engineering. — ACM, 2016. — P. 357 – 367.

[5] T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation / Thanh Nguyen, Peter C. Rigby, Anh Tuan Nguyen et al. // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — ACM, 2016. — P. 1013 – 1017.

[6] Xie Tao, Pei Jian. MAPO: Mining API usages from open source repositories // Proceedings of the 2006 international workshop on Mining software repositories. — ACM.