МЕХАНИЗМЫ РАБОТЫ И ВНУТРЕННЕЕ УСТРОЙСТВО СИСТЕМ УПРАВЛЕНИЯ ПАКЕТАМИ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

Вавилов И.П.

Вавилов Иван Павлович – технический руководитель мобильной разработки,
Компания Swiftlane, г. Москва

Аннотация: *в статье анализируются различные виды систем управления пакетами, описываются внутренние алгоритмы работы.*
Ключевые слова: *менеджер зависимости, система управления пакетами.*

В этой статье я расскажу, в чем системы управления пакетами (сокращенно СУП или менеджер зависимости, по-английски – package manager) схожи по внутреннему устройству, алгоритму работы, и в чем их принципиальные отличия. Я рассматривал те СУП, предназначенные для разработки под операционные системы (сокращенно ОС) iOS и OS X, но содержание статьи с некоторыми допущениями применимо и к другим СУП.

Разновидности систем управления пакетами

- Системные менеджеры зависимостей – устанавливают недостающие утилиты в операционную систему. Например, Homebrew.
- Менеджеры зависимостей языка – собирают исходный код, написанный на одном из языков программирования, в конечные исполняемые программы. Например, go build.
- Менеджеры зависимостей проекта – управляют зависимостями в разрезе конкретного проекта. То есть, в их задачи входит описание зависимостей, скачивание, обновление их исходного кода. Это, например, Cocoapods.

Основное отличие между ними в том, кому они «служат». Системные менеджеры – пользователям, проекта – разработчикам, а языка – и тем, и тем сразу. Далее я буду рассматривать СУП проекта – они используются разработчиками чаще всего.

Схема проекта при использовании СУП

Рассмотрим на примере популярной СУП Cocoapods. Обычно выполняется команду *pod install*, а затем менеджер зависимостей все делает за разработчика. Рассмотрим, из чего должен состоять проект, чтобы эта команда завершилась успешно.

Рис. 1. Типовая схема проекта при использовании СУП

1. Есть код, в котором разработчик использует ту или иную зависимость, скажем, библиотеку Alamofire.
2. Из manifest-файла менеджер зависимостей знает, какие зависимости разработчик использует в исходном коде. Если он забудет указать там какую-либо библиотеку, зависимость не установится, и проект в итоге не скомпилируется.
3. Lock-файл – генерируемый менеджером зависимостей файл определенного формата, в котором перечисляются все зависимости, успешно установленные в проект.
4. Код зависимостей – внешний исходный код, который скачивает СУП и который будет вызываться из кода разработчика.

Это было бы невозможно без конкретного алгоритма, который запускается каждый раз после команды установки зависимостей. Все 4 компонента перечислены друг за другом, т.к. последующий компонент формируется исходя из предыдущего.

Рис. 2. Типовая схема проекта при использовании СУП с указанием последовательности исполнения алгоритма

Не у всех СУП есть все 4 компонента, но с учетом функций менеджера зависимостей наличие всех — оптимальный вариант. После установки зависимостей все 4 компонента идут на вход компилятору либо интерпретатору в зависимости от языка программирования.

Рис. 3. Типовая схема проекта при использовании СУП с указанием шага компиляции или интерпретации

Также обращу внимание, что за первые две составляющие ответственны разработчики – они пишут этот код, а за оставшиеся две – сама СУП – она генерирует файлы и скачивает исходный код зависимостей.

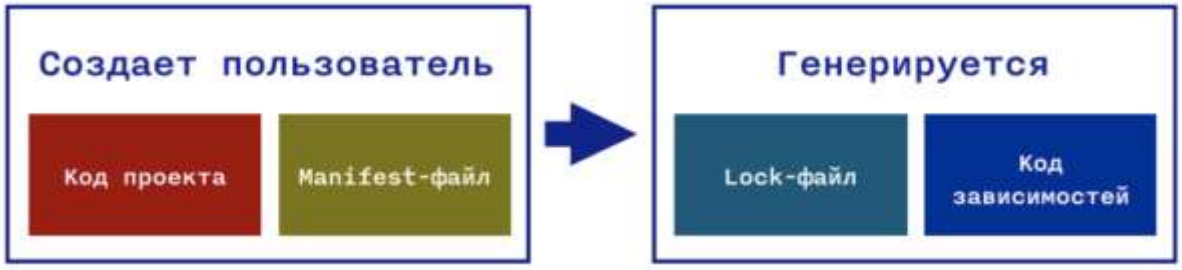

Рис. 4. Типовая схема проекта при использовании СУП с разделением шагов разработчика (пользователя) и СУП

Алгоритм работы менеджера зависимостей

Перейдем к алгоритмической части работы СУП. Типовой алгоритм работы выглядит следующим образом:

1. Проверка проекта и среды окружения. За это отвечает объект, который именуется Analyzer.
2. Построение графа. Из зависимостей СУП должна выстроить граф. Этим занимается объект Resolver.
3. Скачивание зависимостей. Исходный код зависимостей должен быть скачан для того, чтобы разработчик его использовал в своем коде.
4. Интеграция зависимостей. Того, что исходный код зависимостей лежит в директории на диске, может быть недостаточно, поэтому их нужно связать с проектом.
5. Обновление зависимостей. Этот шаг выполняется не сразу за шагом 4, а только при необходимости обновиться на новые версии библиотек. Здесь есть свои особенности, поэтому я выделил их в отдельный шаг — о них далее.

Проверка проекта и среды окружения

Проверка включает сравнение версий ОС, вспомогательных утилит, которые необходимы СУП, а также настроек проекта и manifest-файла: начиная от проверки на синтаксис, заканчивая несовместимыми настройками. Возможные предупреждения и ошибки при проверке podfile:

- Не найдена зависимость ни в одном из spec-репозитории;
- Явно не указана ОС и версия;
- Некорректное имя проекта.

Построение графа зависимостей

Так как у нужных проекту зависимостей могут быть свои зависимости, а у тех в свою очередь — собственные вложенные зависимости или подзависимости, СУП должна это все свести в корректное состояние с правильными версиями библиотек. Схематично все зависимости в результате должны выстроиться в направленный ацикличный граф.

Построение направленного ацикличного графа сводится к задаче топологической сортировки. У нее есть несколько алгоритмов решения.

1. Алгоритм Кана – перебор вершин, сложность $O(n)^1$.
2. Алгоритм Тарьяна – на основе поиска в глубину, сложность $O(n)^2$.
3. Алгоритм Демукрона – послойное разбиение графа.
4. Параллельные алгоритмы, использующие полиномиальное количество процессоров. В таком случае сложность «упадет» до $O(\log^2 n)$.

Сама по себе задача является NP-полной, тот же алгоритм используется в компиляторах и машинном обучении.

Результатом решения является созданный lock-файл, который полностью описывает отношения между зависимостями.

Рис. 5. Шаг, на котором происходит построение графа зависимостей

Рассмотрим, как это выглядит у наиболее популярных менеджеров зависимостей для iOS.

Cocoapods

Реализация алгоритма построения графа выделена в отдельный репозиторий. Здесь же реализация графа и Resolver. В Analyzer можно найти, что проверяется соответствие версий cocoapods системы и lock-файла. Из исходного кода также видно, что Analyzer генерирует таргеты для зависимостей.

В секции PODS перечисляются прямые и вложенные зависимости с указанием версий, далее подсчитываются их контрольные суммы в отдельности и вместе и указывается версия cocoapods, которая использовалась для установки.

Скачивание зависимостей

После успешного построения графа и создания lock-файла, СУП переходит к их скачиванию. Необязательно это будут исходные коды, это могут быть так же исполняемые файлы или собранные фреймворки в бинарном формате. Также все менеджеры зависимостей как правило поддерживают возможность установки по локальному пути.

Рис. 6. Шаг, на котором происходит скачивание зависимостей

Нет ничего сложного, чтобы их скачать по ссылке (которую нужно откуда-то взять), поэтому я не буду описывать, как происходит само скачивание, а остановлюсь на вопросах централизации и безопасности.

Централизация

СУП имеет два пути при скачивании зависимостей:

1. «Сходить» в какой-то перечень доступных зависимостей и по названию получить ссылку для скачивания.
2. Разработчик явно указывает источник для каждой зависимости в manifest-файле.

По первому пути идут централизованные менеджеры зависимостей, по второму – децентрализованные.

Безопасность

Если разработчик скачивает зависимости по протоколам https или ssh, то проблем возникнуть не должно. Тем не менее, часто разработчики предоставляют http-ссылки на свои официальные библиотеки. И здесь можно столкнуться с атакой «человек посередине», когда злоумышленник подменит исходный код, исполняемый файл или фреймворк. Какие-то менеджеры зависимостей не защищаются от этого, а некоторые делают это следующим образом:

- Homebrew: проверка версии утилиты curl в устаревших версиях OS X. Также есть проверка хэша SHA256 при скачивании по протоколу http. Также настройкой можно запретить небезопасные редиректы на http (переменная HOMEBREW_NO_INSECURE_REDIRECT).

- Carthage и Cocoapods: нельзя использовать протокол http при скачивании исполняемых файлов[3].
- Swift Package Manager: ничего, связанного с безопасностью, найти не удалось, но в предложениях по развитию есть короткое упоминание про некий механизм подписи пакетов с помощью сертификатов[4].

Интеграция зависимостей

Под интеграцией я понимаю подключение зависимостей к проекту таким образом, чтобы мы беспрепятственно могли их использовать, и они компилировались с основным кодом приложения. Интеграция может быть либо ручной (Carthage), либо автоматической (Cocoapods). Плюсы автоматической – минимум лишних настроек со стороны разработчика, но может добавиться много непонятного кода и файлов в проект.

В случае ручной разработчик полностью контролирует процесс добавления зависимостей в проект.

Обновление зависимостей

Контролировать исходный код зависимостей в проекте можно с помощью их версий. В менеджерах зависимостей используются 3 способа:

1. Версии библиотеки. Наиболее удобный и распространенный способ. Можно указать как конкретную версию, так и интервал. Вполне предсказуемый способ для поддержки совместимости зависимостей при условии корректного изменения версий авторами библиотек.
2. Название ветка в системе контроля версий git. При обновлении ветки и дальнейшем обновлении зависимости можно предсказать, какие изменения произойдут.
3. Хэш коммита или тэг в системе контроля версий git. При выполнении команды на обновление, зависимости со ссылками на конкретный коммит или тэг (если его не изменят) никогда не будут обновляться.

Заключение

В статье я описал внутреннее устройство систем управления пакетами. Если хотите узнать больше, стоит подробнее изучить исходный код других СУП. Описанная схема является типовой, но в отдельно взятом менеджере зависимостей может что-то отсутствовать или наоборот появиться новое.

Список литературы

1. *Левитин А.В.* Глава 5. Метод уменьшения размера задачи: Топологическая сортировка // Алгоритмы. Введение в разработку и анализ, 2006. С. 220-224.
2. *Седжвик Роберт.* Алгоритмы на графах. 3-е изд., 2002. С. 496.
3. Исходный код системы управления пакетами Cocoapods, Github. [Электронный ресурс], 2021. Режим доступа: https://github.com/CocoaPods/CocoaPods/blob/master/lib/cocoapods/validator.rb/ (дата обращения: 25.06.2021).
4. Предложение по улучшению Swift Package Manager, Github. [Электронный ресурс], 2021. Режим доступа: https://github.com/apple/swift-package-manager/blob/57c5be1db1c1e12e089dff02241ffbce5722fb0e/Documentation/PackageManagerCommunityProposal.md#security-and-signing/ (дата обращения: 25.06.2021).