Готовьсь, цельсь, пли! Как не обжечься при сборке Gradle-приложения и настолько ли всё серьезно?

Доброго дня, читатель! Меня зовут Стручков Михаил и я Android-разработчик в команде мобильного оператора Yota.

В последнее время особенности нашего приложения способствуют частой и кропотливой работе с Gradle. В своем опыте работы с ним я успел пройти через стадию поломанных сборок, отчаяния в попытках понять причину очередного фейла при билде, и неподдельной радости после успешной реализации собственных задумок.

Предлагаю вам, дорогие читатели, упростить тернистый путь к пониманию сборки Gradle-приложений, разобрать основные этапы и их особенности, и попробовать совместно сократить трафик stackoverflow на тему Gradle. В качестве бонуса, немного коснемся Gradle-плагинов и разберемся, как к ним подходить. Добро пожаловать под кат!

Невозможно рассмотреть работу с Gradle в одной статье, поэтому планирую последовательно пополнять список:

1. Готовьсь, цельсь, пли! Как не обжечься в процессе сборке Gradle-приложения
2. Пишем свой Gradle-плагин или делаем обмундирование для боевого слона
3. Пишем эффективные Gradle-таски

Статья получилось довольно объемной, поэтому для удобства ниже представлено оглавление:

Gradle приобрел широкую популярность в качестве системы сборки не только JVM-приложений. Широта предоставляемого инструментария, возможности его адаптации под кастомные задачи, возможности для сборки крупных проектов, простота интеграции с различными CI-системами делают свое дело.

В этой статье поговорим про основные этапы сборки Gradle-приложения - инициализацию, конфигурацию и сборку, и про то, как нам с ними обращаться при работе.

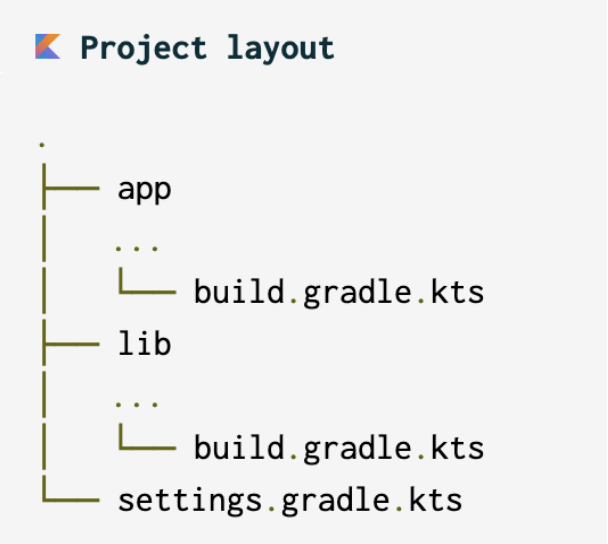
Для своих примеров я буду использовать Kotlin и конфигурацию на Kotlin DSL. ~~Стильно, модно~~. Kotlin в сравнении с Groovy привносит много удобств. Ведь теперь мы можем легко выяснить - что и где, и какого оно типа, а Gradle API наконец начинает нам в этом помогать. С Groovy этого определенно не хватало. Кто еще не в курсе, в документации хорошо рассказано о том, как попробовать Kotlin DSL.

Структура Gradle-проекта

Итак, главная сущность в мире Gradle – это Project. Project-ом выступает само наше приложение, которое мы только что создали. У project-а могут быть subproject-ы, у subproject-а могут быть свои subproject-ы, и таким образом получается древовидная структура. Хорошей практикой является подход, при котором степень вложенности не превышает двух. Такая структура подходит для большинства проектов. При этом subproject-ы можно просто называть Gradle-модулями.

Если вы пользуетесь IDE от JetBrains или Android Studio, по умолчанию они не дадут вам создать иерархию больше двух, что уже наталкивает на мысль что так делать не стоит.

На картинке снизу subproject-ы (Gradle-модули) это app и lib. Корнем Gradle-структуры является сам проект (вот эта маленькая точка сверху):



Project узнает обо всех своих subproject из файла конфигурации settings.gradle (.kts). Написанный здесь скрипт будет выполняться на этапе инициализации проекта.

Инициализация

Итак, в settings.gradle (.kts) мы определяем содержимое нашего приложения, и учим Gradle к нему подступаться. В самом начале задаем имя нашего проекта:

```
rootProject.name = "myproject"
```

Теперь мы можем легко его получить из любого subproject-а на последующих этапах.

Конфигурация плагинов

Далее мы можем определить блок pluginManagement, где есть возможность сконфигурировать плагины, которые используются в нашем приложении:

```
pluginManagement {
	plugins {
	}

	repositories {
	}

	resolutionStrategy {
	}
}
```

Внутри блока plugins есть возможность указать дефолтную версию для плагина, если не используется никакая другая. При этом эту версию можно брать извне, например, из gradle.properties:

```
val helloPluginVersion: String by settings
...
plugins {
	id("com.example.hello") version "${helloPluginVersion}"
}

gradle.properties:
helloPluginVersion=1.0.0
```

Подключать дефолтные версии плагинов из внешних источников, на мой взгляд, уникальный кейс, но если это ваш случай - вы теперь знаете как это сделать. Если вы уже используете такую возможность в вашем проекте - напишите в комментариях, кейс интересный.

По умолчанию все наши плагины подключаются из gradlePluginPortal. Если же плагин необходимо достать из другого репозитория, в блоке repositories можно его определить. Например, мы хотим подключить Android Gradle Plugin. Он лежит в гугловском репозитории, что и объявляем явно:

```
repositories {
        gradlePluginPortal()
        google()
}
```

Готово! Теперь можно добавить зависимость Android Gradle Plugin в classpath (о котором чуть позже) и затем успешно подключать Android-плагины в билдскриптах.

В блоке resolutionStrategy мы можем определить правила для подключения плагинов, используемых проекте. На коде ниже приведен пример того, можно хитрым образом динамически подключить Android Gradle Plugin на тот случай, если в проекте он начнёт использоваться:

```
resolutionStrategy {
     eachPlugin {
         if (requested.id.namespace == "com.android") {
                 useModule("com.android.tools.build:gradle:${requested.version}")
          }
         }
```

После такого финта ушами, AGP автоматически подключится в classpath по необходимости.
В поле requested находится реквест плагина, который мы явно сформировали в том модуле, где собираемся его использовать:

root-project build.gradle.kts:

```
plugins {
   id("com.android.application") version "4.1.0"
}
```

Чтобы узнать какая версия плагина будет запрошена в конечном итоге, следует использовать поле target.

Важный момент: версию плагинов в явном виде стоит определять именно в рутовом build.gradle (.kts). Если сделать это как-то по-другому, чаще всего будет появляться следующая ошибка:

Caused by: org.gradle.plugin.management.internal.InvalidPluginRequestException: Plugin request for plugin already on the classpath must not include a version…

Реже подобная:
Caused by: java.lang.IllegalArgumentException: org.gradle.api.internal.initialization.DefaultClassLoaderScope@68d65269 must be locked before it can be used to compute a classpath!

Полный зоопарк. Я не буду загружать подробностями (ибо у автора тоже с них взрывается мозг), но суть такова, что рутовый build.gradle (.kts) является для этого централизованным местом.

Подключение Gradle-проектов и композитная сборка

Далее с помощью include объявляем существующие в нашем проекте подпроекты (они же по-простому gradle-модули). При этом, если модуль лежит где-то в другом месте и ~~вообще он гриб~~, можно явно указать, как и где его найти с помощью project:

```
include("some-subproject")
project("some-subproject").projectDir = file("../somesubproject")
project(":some-subproject ").buildFileName = "some-subproject.gradle"
```

И последнее, но не менее интересное – includeBuild. С помощью него можно определить проект, который мы бы хотели собрать и определить интерфейс до сборки основного проекта. Таким образом, имеем возможность организовать композитную сборку:

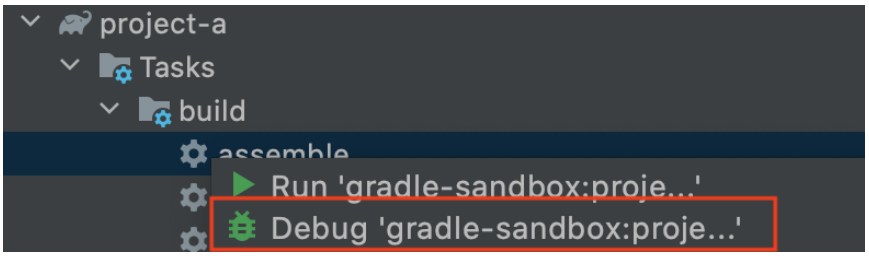
```
includeBuild("some-other-project") {
   dependencySubstitution {
        substitute(module("org.sample:mysample")).with(project(":"))
        }
}
```

С помощью лямбды в dependencySubstitution можно подменить зависимости, используемые в includeBuild-проекте, в том числе на те, что предоставляет наш проект, как на примере выше.
Проект из композитной сборки предоставляет те же возможности

Спойлер: Где можно использовать композитную сборку?

Из своего опыта могу сказать, что композитная сборка может быть очень полезна, когда мы хотим отдебажить какой-нибудь Gradle-проект из другого репозитория. Например, это может быть наш собственный Gradle-плагин. Для

этого просто выбираем Debug у Gradle-таски из нашего плагина, который мы подключаем через композитную сборку:



После этого можно пользоваться всеми радостями отладки от IDE: ставить брейкпоинты, просматривать значения переменных и т.д. при выполнении кода Gradle-плагина.

P.S. Отладить таким образом можно не только Gradle-плагин, но и все содержимое build.gradle (kts), и в том числе на Groovy!

Где ещё может пригодиться композитная сборка, поговорим немного дальше по ходу статьи.

Также settings.gradle (.kts) может с успехом отсутствовать. В этом случае Gradle поймет что других модулей проекте нет и на этом успокоится.

Многие уже привыкли видеть settings.gradle как скрипт, где перечислены все известные в приложении Gradle-проекты. По опыту починки различных поломок с Gradle могу сказать, что сюда посмотрят в последнюю очередь. Если есть возможность не дописывать логику сюда, лучше этого не делать. Практически всегда эту логику можно реализовать в build.gradle (.kts) рутового Gradle-проекта, что будет более очевидно для других разработчиков.

После выполнения инициализации Gradle создает объекты типа Project, с которыми мы продолжаем работу уже на этапе конфигурации.

Конфигурация

Также известна под именами “Sync Project With Gradle Files”, “Load Gradle Changes”, “Reload All Gradle Projects”. На данном этапе главными игроками выступают билдскрипты build.gradle (.kts).

Механизм достаточно простой - для построенного дерева Gradle-проектов на этапе инициализации последовательно вызывается соответствующий билдскрипт. Здесь выполняется следующее:

1. Резолвятся и загружаются зависимости для каждого из известного в settings.gradle (.kts) Gradle-проекта;
2. Строится граф выполнения Gradle-тасок для этапа сборки;
3. Определяются переменные и конфигурируются данные для выполнения тасок на этапе сборки;

О подкапотном пространстве 2 и 3 этапов предлагаю поговорить в статье про Gradle-таски и в комментариях, а сейчас давайте немного сосредоточимся на том, как нам быть с зависимостями.

Управление зависимостями

Для подключения зависимостей в Gradle-проект существует два ключевых блока: repositories и dependencies.

```
repositories {
        mavenCentral()
        maven(url = "https://www.myrepo.io")
        flatDir {
         dirs("lib1", "lib2")
  }

}
dependencies {
        implementation(kotlin(“stdlib”))
}
```

В блоке repositories определяем список репозиториев, в которые нужно сходить для загрузки зависимостей. Gradle предоставляет возможности использовать внешние репозитории, внутренние репозитории, и локальные в виде путей к папкам. Возможности для конфигурации здесь достаточно большие.

При поиске зависимостей, Gradle осуществляет поиск по репозиториям в том порядке, в котором мы эти репозитории у себя объявили. Из этого вытекает, что репозитории с максимальным числом зависимостей стоит выносить в самый верх, и в целом стараться держать как можно меньше репозиториев.

Сами зависимости мы определяем в блоке dependencies. Здесь правило похожее - чем меньше, тем лучше.

В блоке buildscript определяются зависимости, которые необходимо загрузить и положить в classpath для их доступности на этапе конфигурации:

```
buildscript {
repositories {
  }
  dependencies {
        classpath("com.android.tools.build:gradle:$agpVersion")
  }
}
```

На примере выше я добавляю зависимость от Android Gradle Plugin чтобы использовать com.android.* плагины для конфигурации Android-приложения.

Для подключения зависимостей на этапе сборки, блоки dependencies и repositories добавляются в корень билдскрипта build.gradle (.kts). В качестве зависимостей могут быть как другие gradle-проекты, так и внешние/локальные библиотеки:

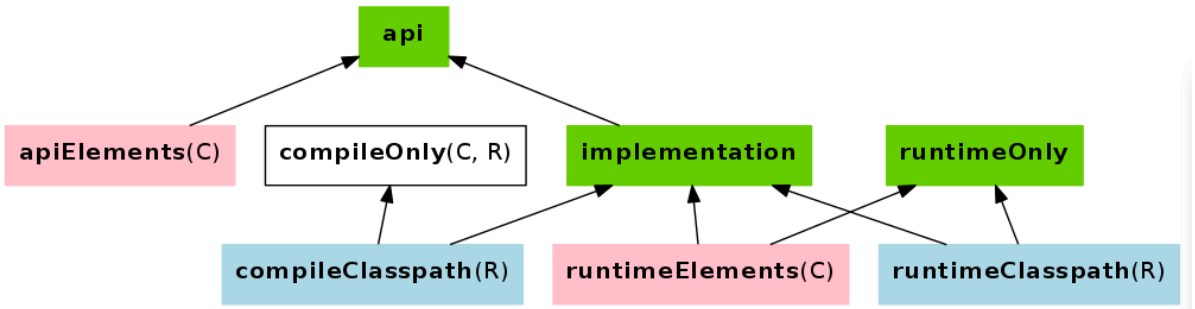
```
dependencies {
   runtimeOnly(“group = "org.springframework", name = "spring-core", version = "2.5"”)
   implementation(fileTree(mapOf("include" to listOf("*.jar"), "dir" to "libs")))
   implementation(kotlin("stdlib"))
   implementation(project(“:project-a”))
}
```

Для подключения gradle-проектов используется символ “:”. С помощью него мы определяем путь, по которому лежит подключаемый gradle-проект относительно рутового gradle-проекта.

Важно следить за тем, чтобы не организовать цикл между зависимостями. Если такое произойдет, Gradle-сборка начнет падать с NullPointerException без объяснения причин, а искать проблему будет сложно.

Типы зависимостей

У Gradle бывают два типа зависимостей - runtime и compile-time. Хорошо объясняет положение вещей следующий рисунок:



При подключении зависимостей в compileClasspath, мы получим runtime exception при попытке достучаться до кода зависимости во время выполнения, поскольку зависимость не попала в приложение. Но в то же время код зависимости будет доступен на этапе сборки проекта. Подключая зависимости в runtimeClasspath, мы гарантируем их попадание в приложение, а значит, и безопасность выполнения кода в runtime. Здесь-то и приходит понимание, что implementation и api добавляют зависимости в оба classpath-а. При этом api также позволяет получить доступ к коду gradle-проекта в случае его транзитивного подключения.

В качестве apiElements и runtimeElements на рисунке обозначен код, который мы хотим отдать на использование в другие gradle-модули.

Производительность конфигурации

Gradle старается, насколько это возможно, оптимизировать процесс конфигурации. Наша задача этому не мешать, а именно, стараться не взаимодействовать с другими gradle-проектами на этапе конфигурации напрямую. Это приводит к связности проектов и замедляет скорость конфигурации.
Самый простой способ сделать проекты связными – определить блоки allprojects { } или subprojects { }.
Чаще всего эти блоки используются чтобы добавить общее поведение, например, определить общую зависимость/репозиторий, определить Gradle-таску, которую мы бы хотели выполнять для каждого проекта и.т.д.:

```
subprojects {
  repositories {
     mavenCentral()
  }
  }
}
```

Эти блоки можно определять в root-проекте, поскольку он и так неявно связан со всеми подпроектами. Но стоит помнить, что выполнение подобного трюка в дочернем проекте оставит негативный след на скорости конфигурации. Если потребность всё же есть, лучше сделать зависимость между Gradle-тасками на этапе сборки.

Как сделать задачу для этапа конфигурации?

У Gradle есть на этот случай колбэки. Самый простой способ добавить колбэк для этапа конфигурации – определить лямбду у функций afterEvaluate/beforeEvaluate. Блок afterEvaluate можно использовать, например, в случае, когда мы хотим добавить таску в граф выполнения и хотим чтобы она выполнялась по определенному правилу. Например, после myTask, как на примере ниже:

```
afterEvaluate {
        tasks.all {
            if (this is org.jetbrains.kotlin.gradle.tasks.KotlinCompile) {
            dependsOn(tasks.named("myTask"))
        }
}
```

Спойлер: Почему не стоит использовать dependsOn

Делая так, мы должны быть уверены в том, что, как и когда выполняет Gradle-таска, от которой мы хотим зависеть. Нет никаких гарантий что при её изменении, наш код не сломается.

Чтобы избежать подобных проблем, таски следует связывать через их inputs и outputs. В этом случае в качестве бонуса мы получим еще и инкрементальность (aka up-to-date проверки), что значительно снизит время повторной сборки. Чтобы во всем разобраться, можно посмотреть доклад Степана Гончарова (таймкод) и пример из документации.

Немного о buildSrc

Еще одна вещь, о которой я не могу не упомянуть, это buildSrc. Модуль buildSrc собирается каждый раз перед конфигурацией нашего рутового проекта и поставляется на этап конфигурации в виде jar. Его довольно удобно использовать для объявления зависимостей, а также содержать общую логику

для билдскриптов. Подробнее о том, как использовать buildSrc в Gradle-проект хорошо описано в этой статье.
Но к сожалению, не всё так гладко, как хотелось бы. У buildSrc есть одна достаточно весомая проблема, связанная с тем, что при любом его изменении мы теряем наш билд-кеш, и как следствие, заставляем проект пересобираться на холодную. Если проект большой, это может быть особенно критично. О том, как решать проблему buildSrc, можно почитать в статье от ребят из Badoo. (Спойлер - решается миграцией на композитный билд при помощи includeBuild). Сейчас наш проект не настолько большой, чтобы проблема buildSrc ощутимо ударила. Но я не исключаю, что с его ростом композитный билд действительно станет нашим будущим.

Куда можно вынести общую логику билдскриптов?

Если вы используете Groovy, можно вынести логику в отдельный билдскрипт и подключать его в нужные места вот так:

build.gradle:
```
apply from: ‘common.gradle’
```

При этом скрипт можно положить напрямую в папку с проектом. Поскольку Groovy интерпретируемый язык, то ничего заранее компилировать не придется.

Если вы пишете подключаемые билдскрипты на Kotlin DSL, то они обязательно должны лежать в buildSrc. Связано это с тем, что Kotlin-скрипт должен быть скомпилирован перед использованием. При этом, для подключения нужно выполнить небольшой финт и в build.gradle (.kts) модуля buildSrc добавить следующее:

buildSrc / build.gradle.kts:
```
plugins {
  `kotlin-dsl`
  `kotlin-dsl-precompiled-script-plugins`
}
```

Теперь можем добавить Kotlin-скрипт в buildSrc:

common.gradle.kts:
```
dependencies {
  add("implementation", "org.sample:some-shared-dependency:1.0.0")
}
```

И подключить его:
build.gradle.kts:

```
apply(id = “common”)
```

Скрипт, подключаемый образом, компилируется всего один раз перед использованием, что не повлияет на скорость конфигурации, и даже ускорит её, если вынести достаточно жирный кусочек.

Для того, чтобы связать Kotlin и Groovy, в Kotlin-скрипты можно подключать стандартные {*}.gradle с помощью того же apply:

```
apply(from = “common.gradle”)
```

Сборка

Вот мы и плавно подобрались к тому, что чаще всего ~~поджигает наш ноутбук~~ рождает приложение в свет. На этом этапе Gradle начинает выполнять таски, которые были добавлены в граф зависимости на этапе конфигурации в нужном порядке.

Коротко про Gradle Task

Gradle Task представляет собой единицу работы, которую выполняет сборка. К примеру, это может быть компиляция классов, создание JAR, создание документации Javadoc или публикация в репозиторий.

Простейшим вариантом написать таску для этапа сборки является реализация лямбды в функциях doFirst/doLast, как на примере ниже:

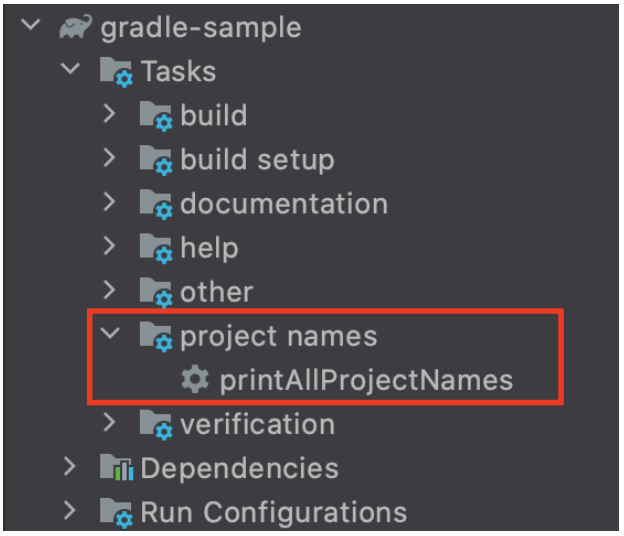
```
tasks.register("printAllProjectNames") {
    group = “project names”
    description = “Prints all projects names”

  doFirst {
   println(“Start execution”)
   }
  doLast {

    println("Root project name is: ${project.name}")

    project.subprojects.forEach { project ->
      println("Child project name is: ${project.name}")
    }
  }
}
```

Таске очень желательно добавлять description и group чтобы было без лишних телодвижений было понятно чем она занимается. После добавления group таску будет удобно искать и запускать:



Таску можно выполнить также и через консоль с помощью команды:

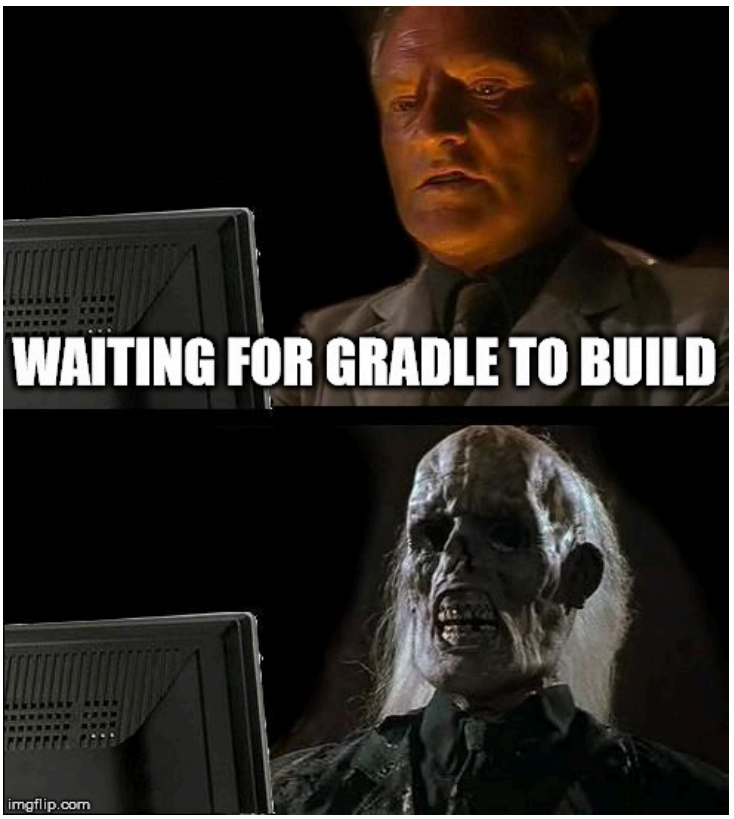
gradle {your_task_name}

Gradle Daemon

Непосредственным выполнением сборки занимается Gradle Deamon. Он включен по умолчанию для Gradle версии 3.0 и выше. Gradle Daemon является долгоживущим системным процессом, периодически осуществляющим сборку когда мы этого хотим. Внутри него происходит много in-memory кеша, оптимизации работы с файловой системой и оптимизации кода выполнения сборки. Если коротко - всё идет на пользу. Пожалуй, исключение только одно - он довольно прожорлив, и Gradle любит держать несколько демонов на разные случаи жизни. Если система начинает ощутимо лагать, всегда можно всех за раз прибить командой

gradle --stop.

Несколько советов по оптимизации скорости сборки



Нетрудно понять, что сборка - самый трудозатратный этап, занимающий львиную долю всего времени работы Gradle. Есть несколько рекомендаций для того, как нам её ускорить, ~~первая из которых - запастись терпением~~:

1. Параллельное выполнение тасок.

Всё просто - org.gradle.parallel=true в вашем gradle.properties, и Gradle будет стараться максимально распараллелить выполнения тасок на этапе сборки.

2. Обновление Gradle

Gradle важно периодически обновлять. Связано это с тем, что с каждой версией Gradle привносят улучшения по скорости сборки, исправления багов и новые интересные плюшки. Для удобства обновления, был придуман Gradle Wrapper (или просто Wrapper).
Это ни что иное, как обычная Gradle-таска. Теперь для обновления версии можно написать в рутовом build.gradle (.kts) следующее:

```
tasks.withType<Wrapper> {
  gradleVersion = "{gradle_version}"
}
```

, а затем выполнить команду gradle wrapper. Или же выполнить

```
$ gradle wrapper --gradle-version ${gradle_version}
```

В обоих случаях вместо gradle_version указываем желаемую версию Gradle. Тем самым Gradle сам загрузит нужную версию и положит её в папку с проектом. Она же будет использоваться в дальнейшем по умолчанию, а счастливые мы сможем запускать таски с помощью скрипта gradlew, который появится в папке с нашим проектом.

3. Правильное использование api/implementation.

При изменении реализации зависимости, Gradle пересобирает на холодную все зависящие от неё модули. Я уже упомянул про api и implementation в разговоре о способах подключения зависимостей и теперь понятно, что при подключении зависимости через api, она также транзитивно попадает в classpath-ы модулей, в которые мы подключили наш модуль. Тем самым увеличивается количество модулей, которые gradle будет пересобирать при изменении зависимости. Если нет необходимости в транзитивном использовании кода из зависимости, для её подключения следует использовать implementation.

4. Инкрементальность и build-кеш.

Оптимизация Gradle позволяет нам пропускать выполнение таски при определенных условиях. У Gradle существует 5 состояний тасок - EXECUTED, UP-TO-DATE, FROM-CACHE, SKIPPED и NO-SOURCE. В разговоре про кеш нам интересны два из них - UP-TO-DATE и FROM-CACHE. Они сигнализируют о том, что результаты выполнения наших тасок были успешно подтянуты гредлом из результатов предыдущей сборки. Об остальных состояниях можно почитать в документации.

UP-TO-DATE. Возникает в случае, если разработчик таски позаботился о ней и самостоятельно реализовал инкрементальность её выполнения (проверки на up-to-date), или же все зависимости этой таски, если они есть, были успешно взяты из кеша, или признаны гредлом up-to-date.

FROM-CACHE. Это состояние возникает в случае, если Gradle смог восстановить outputs нашей таски из билд-кеша. При этом по умолчанию build-cache выключен. Build-cache довольно удобно использовать на CI, таким образом ускорив выполнение пайплайнов Gradle-сборки на холодную. Но я бы не рекомендовал использовать его для локальных сборок, поскольку знаменитый Clean/Rebuild в этом случае теряет свою силу - параметры для тасок будут тянуться из Gradle-кеша, хранящегося на вашей локальной машине. Как организовать build-cache всегда можно узнать тут.

Бонус для дочитавших до конца: Gradle-плагины

Gradle-плагины являются своеобразными контейнерами, в которых может содержаться логика инициализации, конфигурации и сборки. Как правило, плагины используются для внедрения какой-то законченной бизнес-логики в Gradle. На самом деле, плагин есть ни что иное как обычный интерфейс с одним методом apply, где в качестве generic-типа чаще всего выступает Project:

```
public interface Plugin<T> {

    void apply(T target);
}
```

Для написания плагина можно пойти тремя путями. Каждый из них рассмотрим в следующей статье, а сейчас лишь скажу что самый простой из них мы уже видели в разговаривая про то, куда еще можно вынести общую логику билдскриптов. Да, это есть ни что иное, как script-плагин для Gradle.

Пожалуй, самыми яркими примерами будут являться представители из Android Gradle Plugin: "com.android.application" и "com.android.library". Подключая их в билдскрипты, мы получаем возможность собирать и конфигурировать сборку Android-приложений.

Подключать плагины в наши билдскрипты мы можем с помощью apply:

```
apply(plugin = "com.android.application")
```

, или в блоке plugins:

```
plugins {
        `jacoco`
}
```

На примере выше видно, как красиво Kotlin позволяет подключать плагины с помощью extension-функций. По аналогии можно написать свои экстеншены и использовать их для подключения плагинов. Красота!
Кому хочется попробовать себя в разработки плагинов, добро пожаловать в документацию. Также можно посмотреть доклад с AppsConf 2019. В нем раскрываются способы реализации плагинов, а также идет разговор об областях их применения. Возможно, после просмотра вы поймете, что с их помощью вам будет проще решить производственную задачу.

Заключение

Gradle действительно мощный инструмент для сборок проектов, хоть и работа с ним не всегда проста и очевидна. После появления возможности использовать Kotlin для реализации своих Gradle-задумок, магия постепенно начинает уходить и на stackoverflow хочется заходить реже. К тому же Gradle ведут работу над созданием хорошей документации, что несомненно очень радует (правда, её космический объем не остановит только настоящих энтузиастов)

И конечно же, Gradle это open-source. Можете с удовольствием скрасить один из вечеров в репозитории Gradle на Github.

К сожалению, показать весь Gradle в одной, и даже в двух статьях очень сложно. Однако кушая слона по кусочкам, можно прийти к успеху.
В следующей статье хотел бы подойти к практике и рассмотреть реализацию тасок и плагинов, а также рассказать про то, для чего мы их используем в своем проекте. Что еще вы бы хотели видеть в следующей статье? Пишите в комментариях любые замечания и предложения как по текущей статье, так и по будущим, буду рад!

Список докладов

Степан Гончаров - Gradle [A-Z]. Доклад про Gradle под капотом. Спикер рассказывает про устройство Gradle, реализацию тасков и плагинов, и (о боже!) затрагивает недостатки Kotlin DSL.
Stefan Oehme - Composite Builds with Gradle. Доклад про суть композитной сборки в Gradle и способах ее применения (на английском).
Филипп Уваров - Gradle Plugin Development. Доклад с AppsConf про разработку Gradle-плагинов. Раскрывает область применения плагинов, способы их реализации и по каждому рассматривает преимущества/недостатки.