Gradle 构建过程包括三阶段：

1. 初始化：Gradle 支持单项目和多项目构建。在初始化阶段，Gradle 确定哪些项目将参与构建，并为每个项目创建一个 Project 实例。
2. 配置：在此阶段配置 Project 对象。执行作为构建一部分的所有项目的构建脚本。
3. 执行：Gradle 确定在配置阶段创建和配置的要执行的任务子集。该子集由传递给 gradle 命令和当前目录的任务名称参数确定。 Gradle 然后执行每个选定的任务。

大多数规则是关于在一个阶段或另一个阶段允许做什么。每个阶段都有不同的限制。

1. 不要在配置阶段进行昂贵的计算

它会减慢构建速度。此类计算应封装在任务操作中。

1. 避免在 Gradle 的容器类型上使用 create 方法

T create​(String name, Action<? super T> configureAction)

throws InvalidUserDataException

使用给定名称创建一个新项目，将其添加到此容器中，然后使用给定操作对其进行配置。

推荐使用 register 方法

<T extends Task> TaskProvider<T> register​(String name,

Class<T> type,

Action<? super T> configurationAction)

throws InvalidUserDataException

定义一个新任务，将在需要时创建和配置。当使用诸如 TaskCollection.getByName(java.lang.String) 之类的查询方法定位任务时，或者当任务被添加到任务图中以执行时，或者在此方法的返回值上调用 Provider.get() 时。

使用此方法通常比使用 create(java.lang.String, java.lang.Class, org.gradle.api.Action) 或 create(java.lang.String, java.lang.Class) 更有效，因为这些方法将急切地创建和配置任务，无论当前构建是否需要该任务。另一方面，此方法将推迟创建和配置，直到需要。

1. 避免 Gradle 容器类型的 all 回调

void all​(Action<? super T> action)

对该集合中的所有对象以及随后添加到该集合的任何对象执行给定的操作。

void all​(Closure action)

对该集合中的所有对象以及随后添加到该集合的任何对象执行给定的闭包。该对象作为闭包委托传递给闭包。或者，它也作为参数传递。

推荐使用 configureEach 方法

void configureEach​(Action<? super T> action)

使用给定的操作配置此集合中的每个元素，因为每个元素都是必需的。操作按添加的顺序运行。

1. 不要假设某个插件是在另一个插件之后应用的。相反，使用 pluginManager.withPlugin()。

void withPlugin(String id, Action<? super AppliedPlugin> action)

应用指定插件时执行给定的操作。如果已应用具有指定 ID 的插件，则提供的操作将立即执行。否则，该操作将在应用具有指定 ID 的插件后立即执行。给定的动作总是在应用插件后执行。

1. 避免做出任何类型的排序假设

惰性配置、回调和提供者链是游戏的名称。

随着构建复杂性的增加，了解特定值的配置时间和位置可能变得难以推理。 Gradle 提供了几种使用惰性配置来管理这种复杂性的方法。

Gradle 提供了惰性属性，它会延迟属性值的计算，直到实际需要它。这些为构建脚本和插件作者提供了三个主要好处：

1.构建作者可以将 Gradle 模型连接在一起，而不必担心何时会知道特定属性的值。例如，可能希望根据扩展的源目录属性设置任务的输入源文件，但在构建脚本或其他插件配置它们之前，扩展属性值是未知的。

2.构建作者可以将任务的输出属性连接到其他任务的输入属性，Gradle 会根据此连接自动确定任务依赖关系。属性实例携带有关哪个任务（如果有）产生其值的信息。构建作者无需担心使任务依赖项与配置更改保持同步。

3.构建作者可以避免在配置阶段进行资源密集型工作，这会对构建性能产生很大影响。例如，当配置值来自解析文件但仅在运行功能测试时使用时，使用属性实例来捕获这意味着仅在运行功能测试时才解析文件，而不是在例如，clean运行。

Gradle 用两个接口表示惰性属性：

1.Provider 代表一个只能查询不能更改的值。

这些类型的属性是只读的。

Provider.get() 方法返回属性的当前值。

可以使用 Provider.map(Transformer) 从另一个 Provider 创建 Provider。

许多其他类型扩展了 Provider 并且可以在任何需要 Provider 的地方使用。

2.Property表示可以查询和更改的值。

这些类型的属性是可配置的。

Property扩展了 Provider 接口。

方法 Property.set(T) 为属性指定一个值，覆盖任何可能存在的值。

方法 Property.set(Provider) 为属性的值指定一个 Provider，覆盖任何可能存在的值。这允许在配置值之前将 Provider 和 Property 实例连接在一起。

可以通过工厂方法 ObjectFactory.property(Class) 创建Property。

惰性属性旨在传递并且仅在需要时进行查询。通常，这将发生在执行阶段。

1. 不要在任务操作中访问 Project 实例

它破坏了配置缓存，最终将被弃用。

1. 不要访问另一个项目的 Project 实例

这称为跨项目配置，非常脆弱。它在项目之间创建了隐含的、几乎不可建模的依赖关系，并且只会导致悲伤。相反，通过声明依赖项来跨项目共享工件。

它还破坏了实验性项目隔离功能，但这暂时不会真正相关。

在多项目构建中，一个常见的模式是一个项目使用另一个项目的工件。一般来说，Java生态系统中最简单的消费形式是，当A依赖B时，那么A会依赖项目B产生的jar。如本章前面所述，这是由A依赖B的一个变体来建模的，这里的变体是根据 A 的需要来选择的。为了编译，我们需要 B 的 API 依赖，由 apiElements 变体提供。对于运行时，我们需要由 runtimeElements 变体提供的 B 的运行时依赖项。

但是，如果需要与主要工件不同的工件怎么办？例如，Gradle 提供了对依赖另一个项目的测试装置的内置支持，但有时需要依赖的工件根本不会作为变体公开。

为了在项目之间安全共享并允许最大性能（并行性），必须通过传出配置公开此类工件。

不要直接引用其他项目任务

声明跨项目依赖关系的常见反模式是：

dependencies {

// this is unsafe!

implementation project(":other").tasks.someOtherJar

}

这种发布模型是不安全的，并且可能导致不可复制且难以并行化构建。

1. Avoid afterEvaluate

它引入了微妙的排序问题，调试起来非常具有挑战性。使用Provider或Property（另请参阅惰性配置）。

1. 不要在任务操作之外对 Provider 调用 get()

使用Provider的全部意义在于尽可能晚地对其进行评估。如果过早调用 get()（评估它）将导致痛苦的排序问题。相反，使用map或flatMap。

<S> Provider<S> map​(Transformer<? extends S,​? super T> transformer)

返回一个新的 Provider，其值是使用给定函数转换的此Provider的值。结果Provider将是实时的，因此每次查询它时，它都会查询原始（此）Provider并将转换应用于结果。每当原始Provider没有值时，新的Provider也将没有值并且不会调用转换。当此Provider表示任务或任务的输出时，新Provider将被视为任务的输出，并将携带 Gradle 可用于自动将任务依赖项附加到使用新Provider作为输入值的任务的依赖项信息。

<S> Provider<S> flatMap​(Transformer<? extends Provider<? extends S>,​? super T> transformer)

返回一个新的 Provider，其值是使用给定函数转换的此Provider的值。虽然在功能上与常规 map 操作非常相似，但此方法提供了一种将任务输入和输出连接在一起的便捷方式。任务输入和输出通常采用Provider或Property的形式，后者是Provider的特例，其值可以随意更改。使用 flatMap 连接此类属性的示例如下：

class Producer extends DefaultTask {

@OutputFile

abstract RegularFileProperty getOutputFile()

//irrelevant details omitted

}

class Consumer extends DefaultTask {

@InputFile

abstract RegularFileProperty getInputFile()

//irrelevant details omitted

}

def producer = tasks.register("producer", Producer)

def consumer = tasks.register("consumer", Consumer)

consumer.configure {

inputFile = producer.flatMap { it.outputFile }

}

像这样连接输入和输出属性的另一个好处是 Gradle 可以基于此类连接自动检测任务依赖关系。为了在代码级别实现这一点，与此提供者（调用 flatMap 的那个）相关的任何任务细节都将被忽略。新的提供者将使用与转换的返回值相关联的任何任务细节。

flatMap 返回的新提供者将是实时的，因此每次查询它时，它都会查询此Provider并将转换应用于结果。每当这个Provider没有价值时，新的Provider也将没有价值，转换也不会被调用。

1. 不要使用内部 API

Gradle 认为内部 API 是公平的游戏，即使在较小的版本中也可以进行重大更改。因此，使用这样的 API 本质上是脆弱的，并且会导致严重的、完全可以避免的头痛。

1. 不要在公共 API 中使用 Kotlin lambda

请改用 Action<T>。 Gradle 在运行时增强字节码，为插件用户提供更好的 DSL 体验。

Interface Action<T>

对 T 类型的对象执行一些操作。

1. 不要自己创建对象

请改用 ObjectFactory。 （这是配置时间问题。）

Interface ObjectFactory

用于创建各种模型对象的工厂。通过使用 javax.inject.Inject 注释公共构造函数或属性 getter 方法，可以将工厂实例注入任务、插件或其他对象。它也可以通过 Project.getObjects() 获得。

1. 不要在自定义扩展中使用列表

请改用域对象容器。再一次，Gradle 能够以这种方式提供增强的 DSL 支持。

<T> NamedDomainObjectContainer<T> domainObjectContainer​(Class<T> elementType)

创建一个新的 NamedDomainObjectContainer 用于管理指定类型的命名对象。

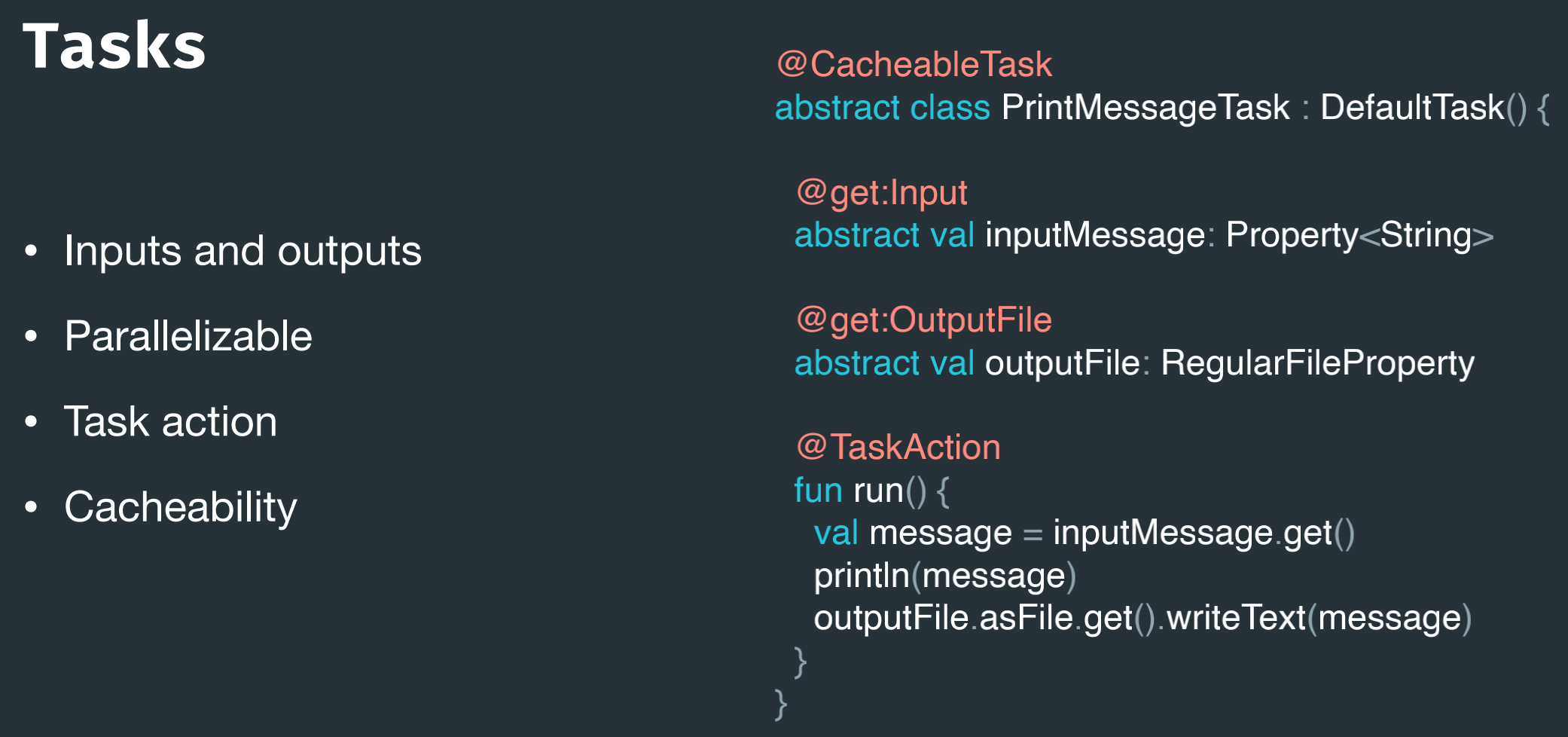
指定的元素类型必须有一个将名称作为字符串参数的公共构造函数。类型必须是非最终的并且是类或抽象类。

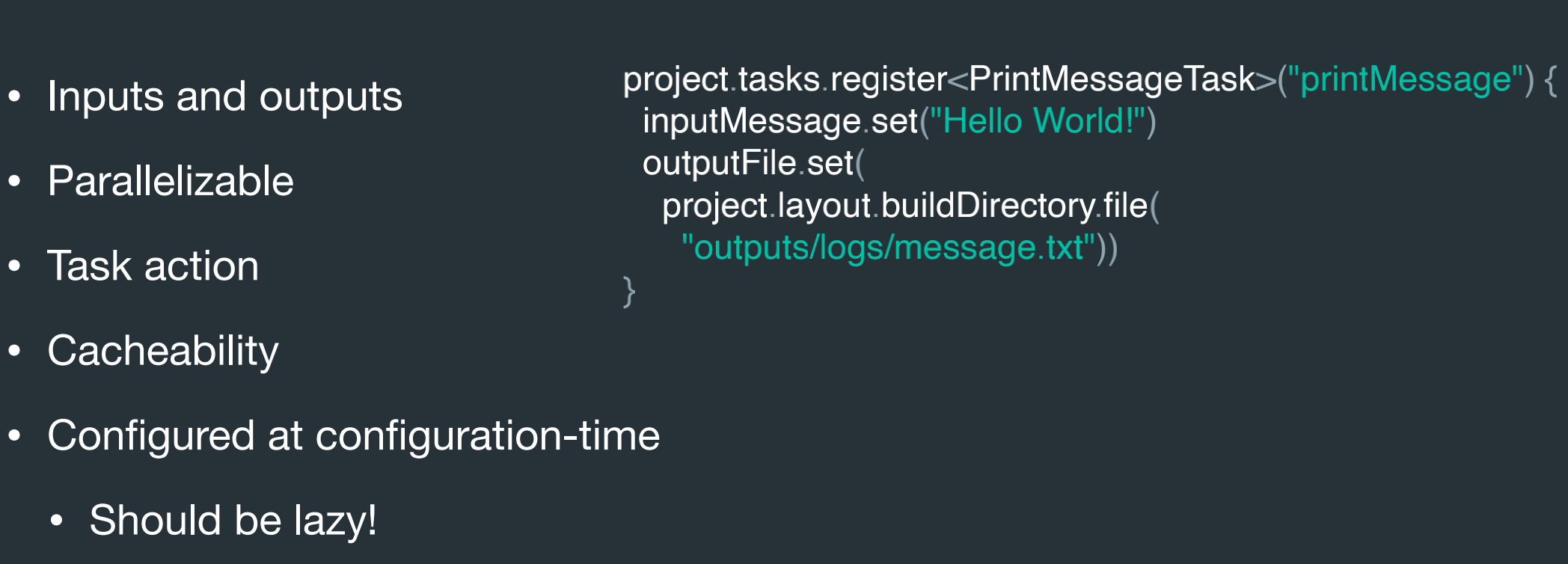
如果接口声明了 String 类型的只读名称属性，则支持接口，否则为空或完全由托管属性组成。

所有对象都必须将它们的名称公开为名为“name”的 bean 属性。该名称必须在对象的生命周期内保持不变。

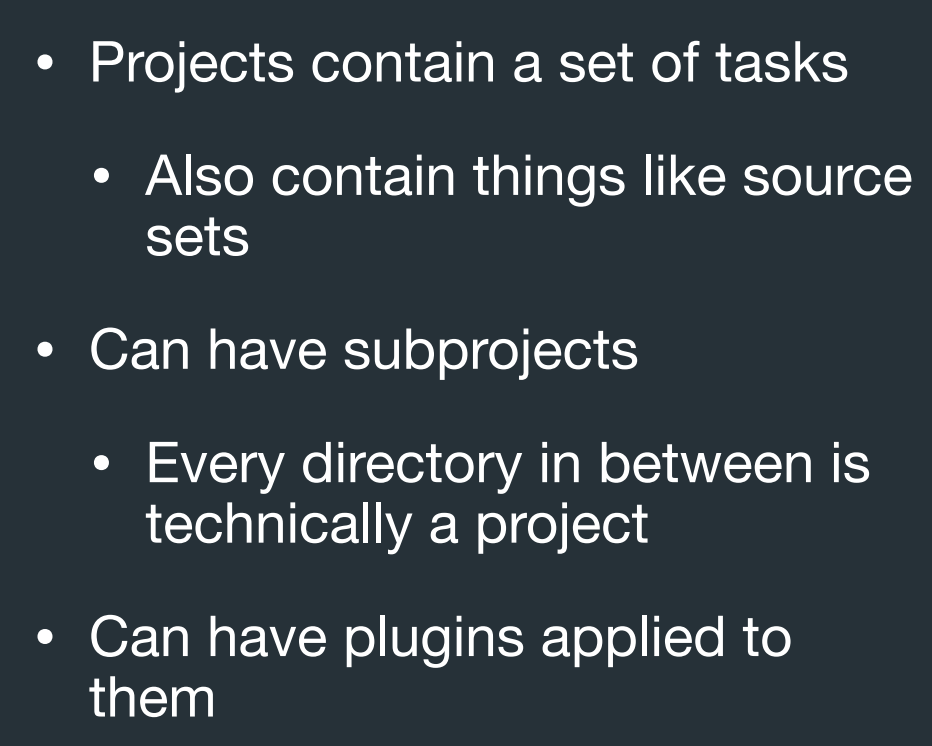
容器创建的对象是可修饰和可扩展的，并且具有可用于注入的服务。

1. Task

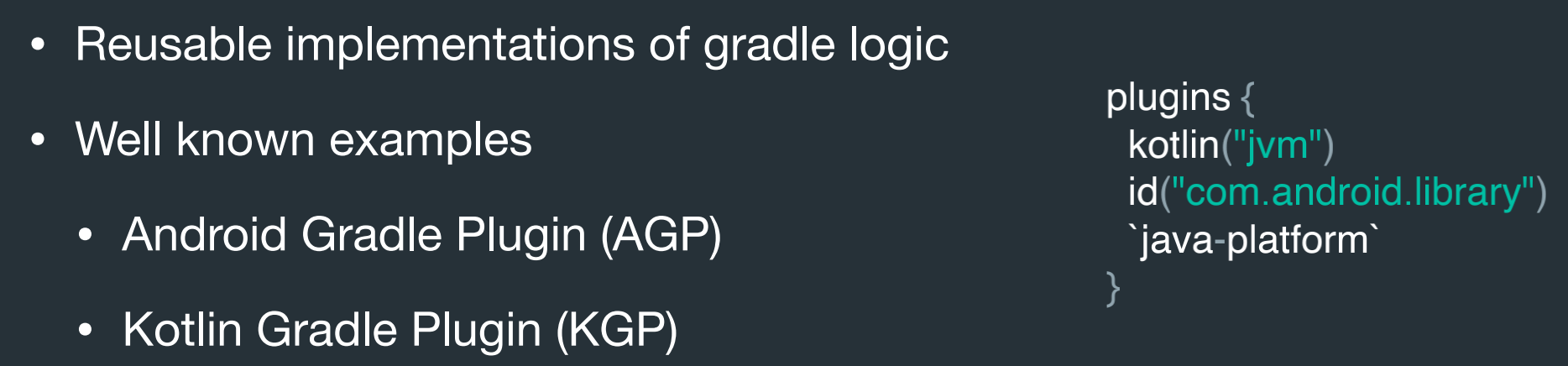




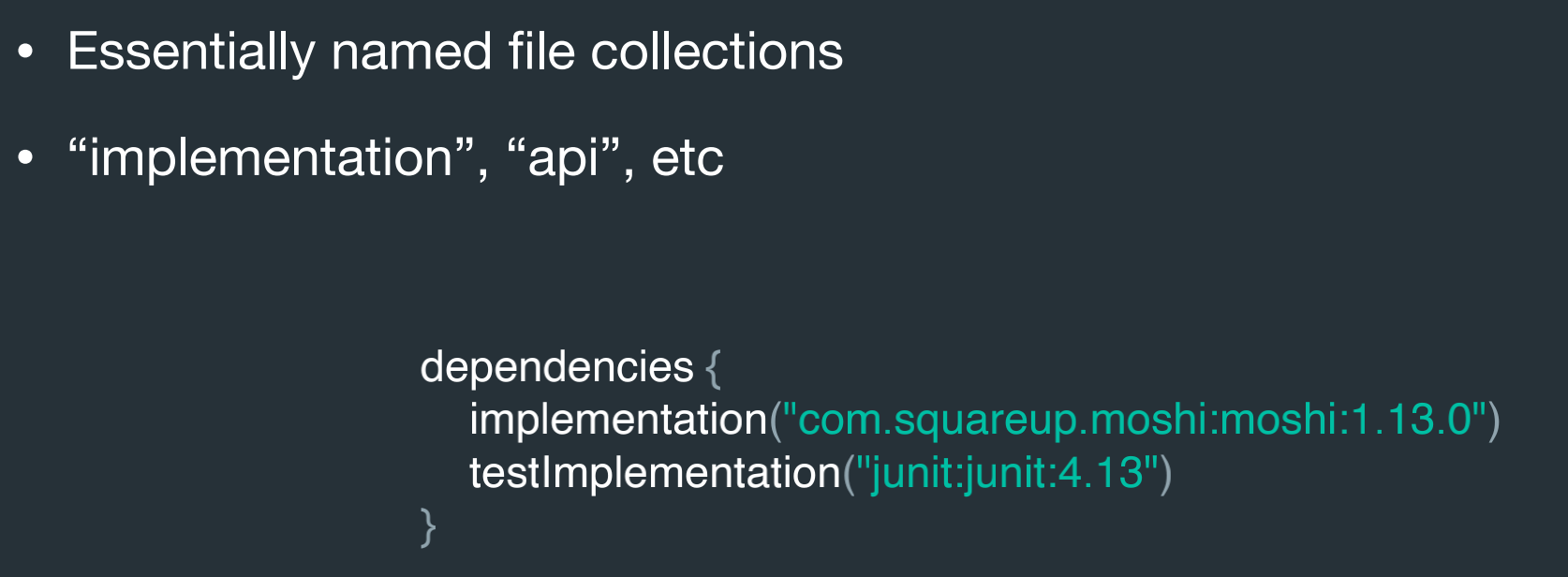
1. Project



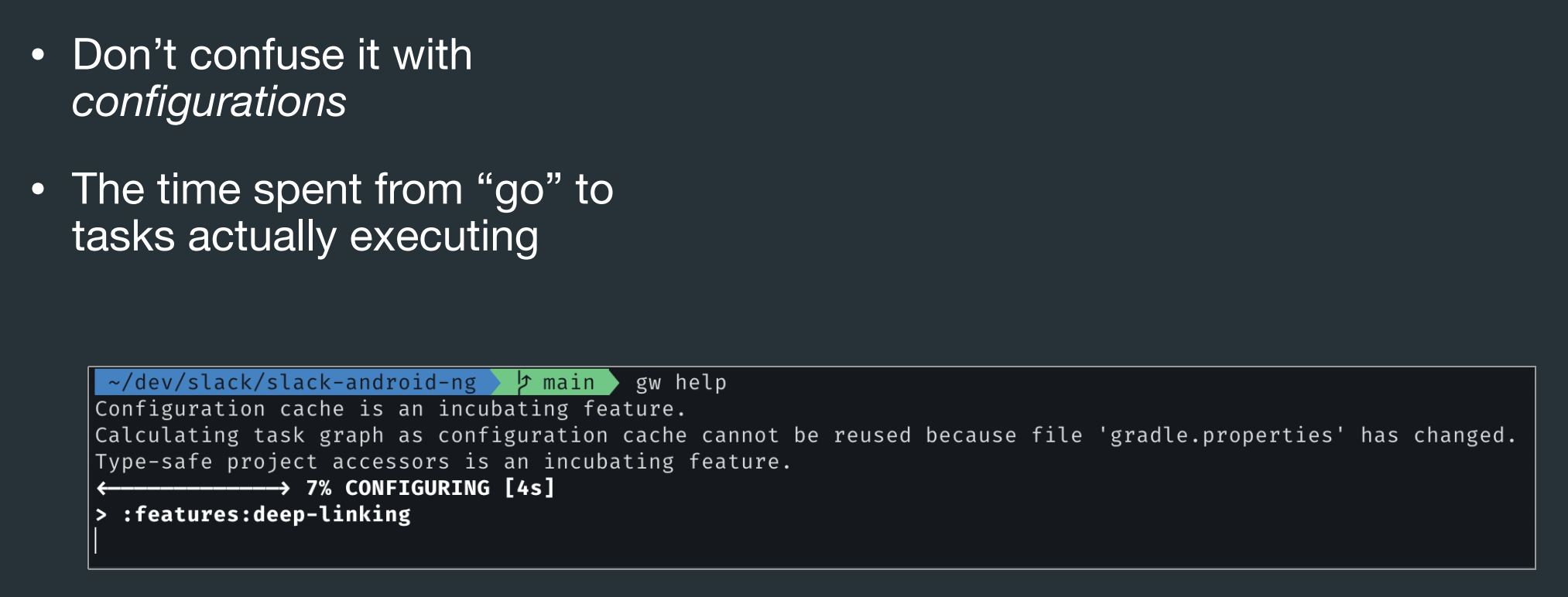
1. Plugin



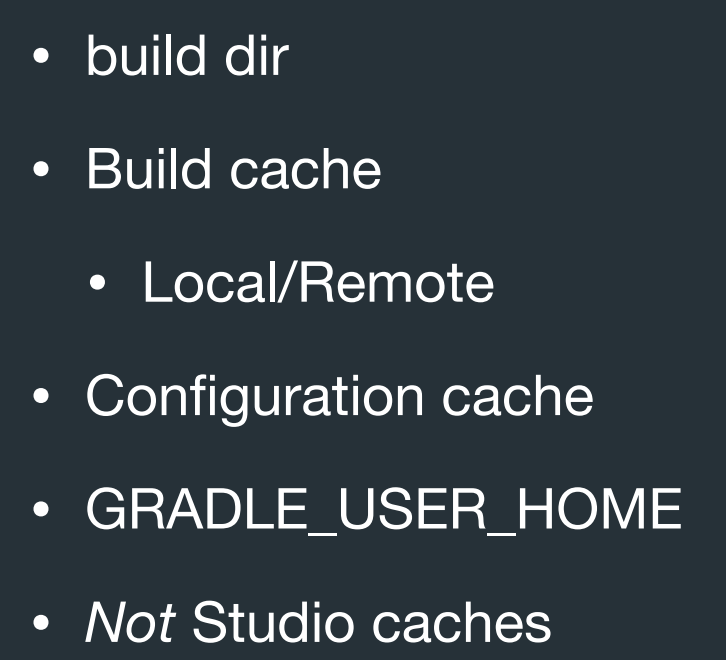
1. Configurations



1. Configuration



1. Caching/Incremental Builds



Gradle 本地构建缓存 - 一种通过将给定输入状态的输出存储/恢复到磁盘来跳过 Gradle 任务执行阶段的机制。例如，如果您运行一个可缓存的任务，检测 build/ 目录，然后再次运行该任务，Gradle 将跳过运行它并直接恢复第一个任务的输出。这在分支之间切换时特别有用。此功能很稳定，但遗憾的是默认关闭，您应该采用它

Gradle 远程构建缓存 - 与本地构建缓存等效的功能，除了使用远程服务器来存储任务输入/输出而不是本地磁盘。这种类型的缓存通常通过让持续集成 (CI) 构建器推送到此服务器以及读取此缓存的所有本地开发人员来使用。

Gradle 配置缓存 - Gradle 功能用于存储/恢​​复您可能从终端知道的配置阶段的状态：

<-------------> 0% 配置 [4s]

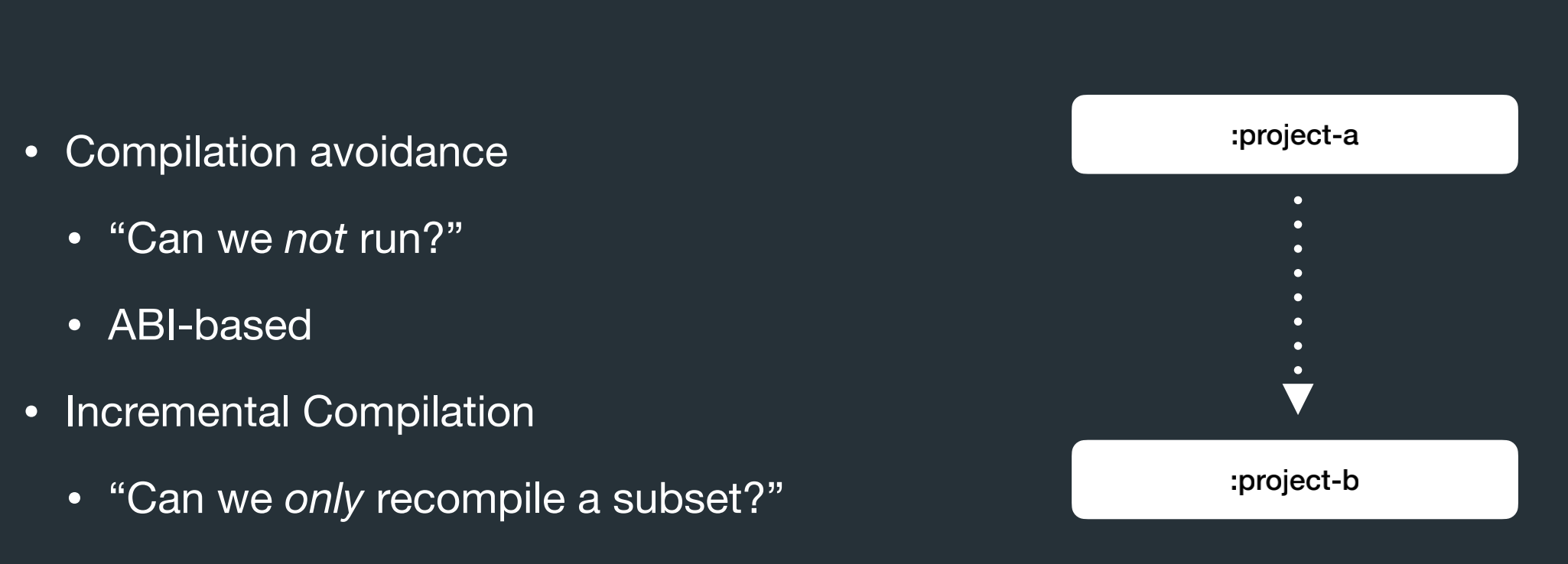
Gradle 使用所有输入序列化所有任务，这些输入可能会使第一次运行期间任务的设置方式无效，并且对于以后的运行，它会从磁盘重新加载这些任务，而不是再次重建整个任务图，本质上是这样 ./gradlew第二次运行时的测试只运行测试，跳过弄清楚如何配置此任务的步骤。此功能目前（截至 2022 年 4 月）正在孵化，但效果很好，应该采用。

GRADLE\_USER\_HOME - Gradle 用于存储各种非项目特定全局状态的目录（默认为 ~/.gradle），包括 Gradle 包装缓存（提取的 gradle-X.X-bin.zip）和工件缓存（以避免重新下载 maven 依赖项）

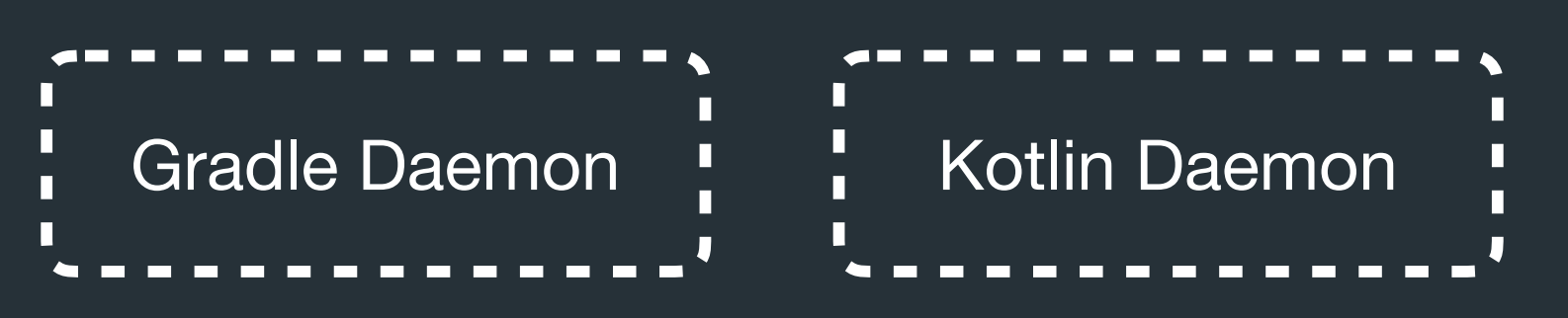
擦除 Studio 缓存不修复你的构建

1. 增量编译（IC）



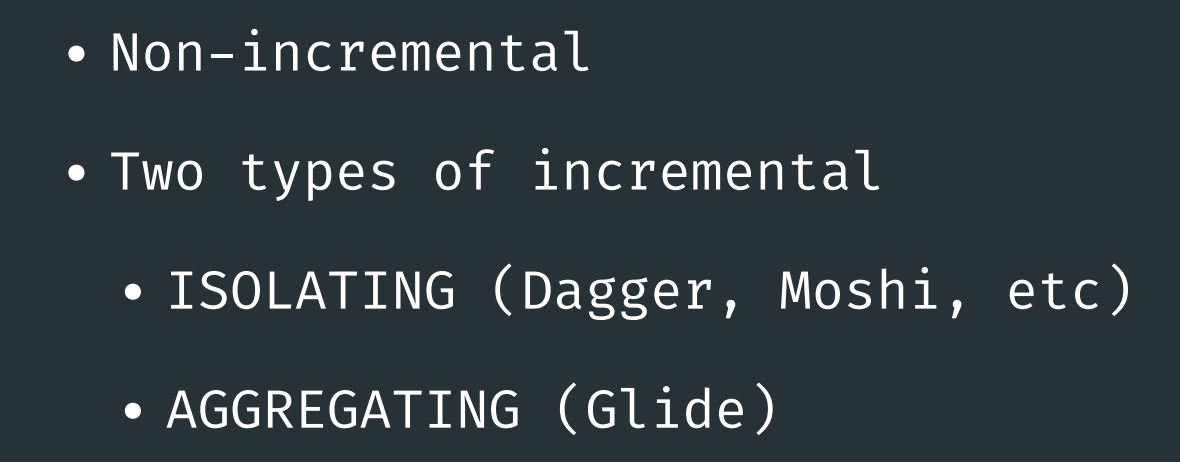


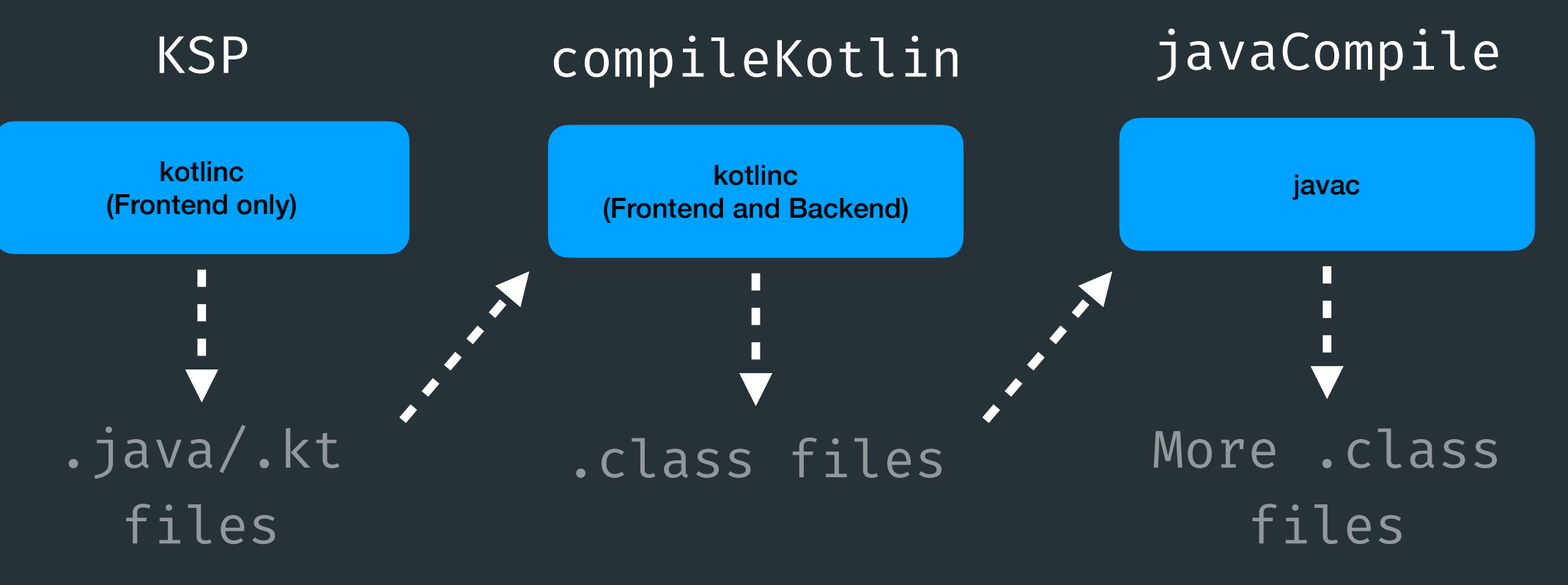
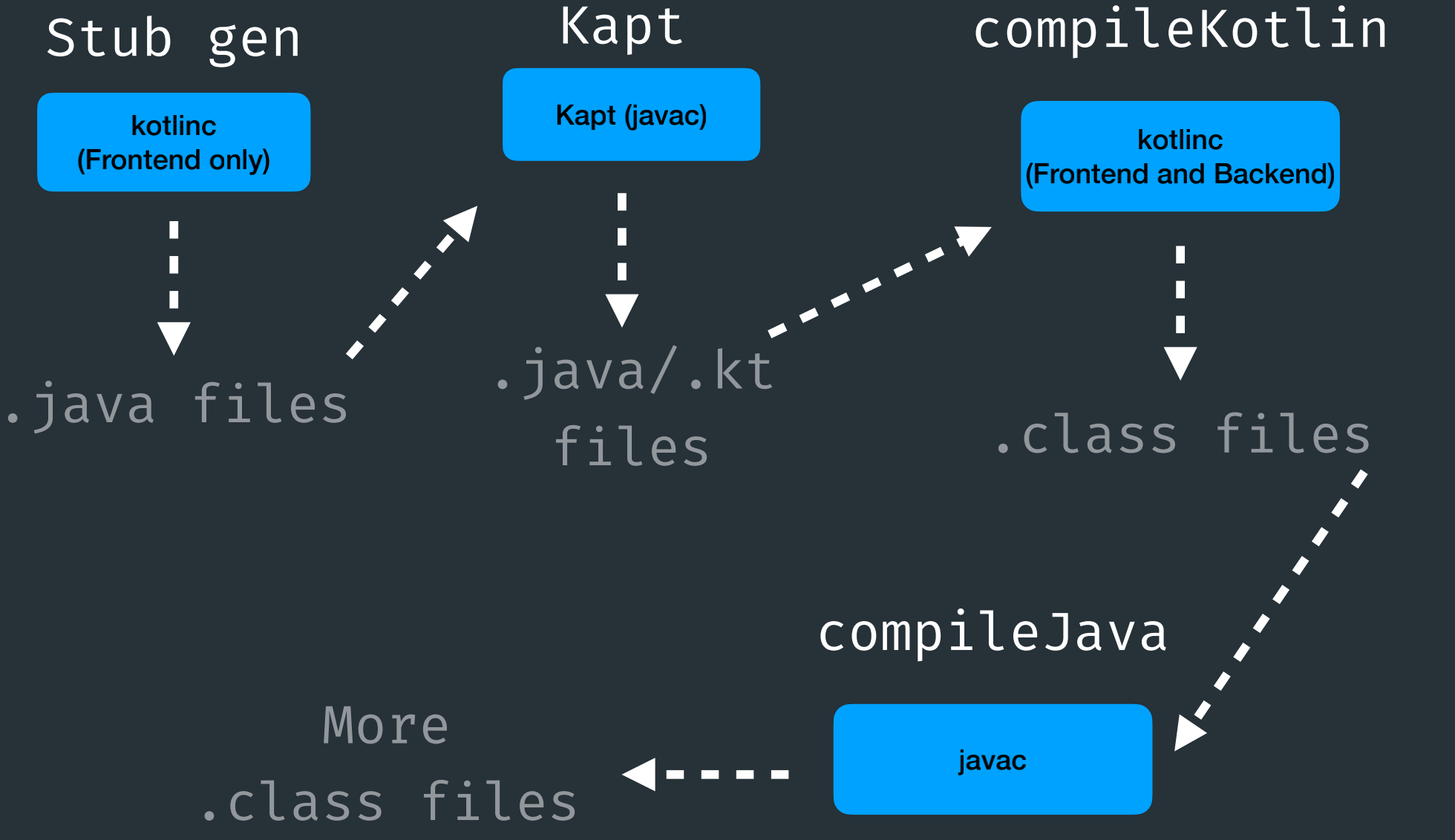
1. Daemons



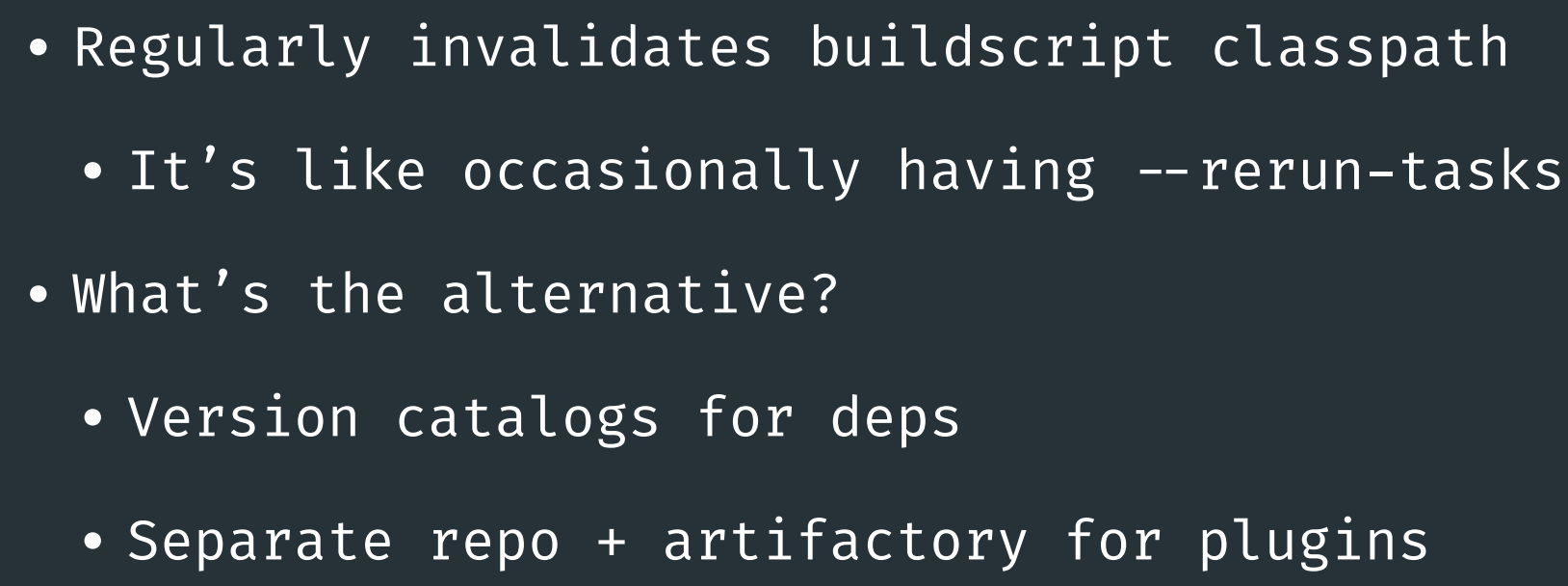
1. Annotation Processing

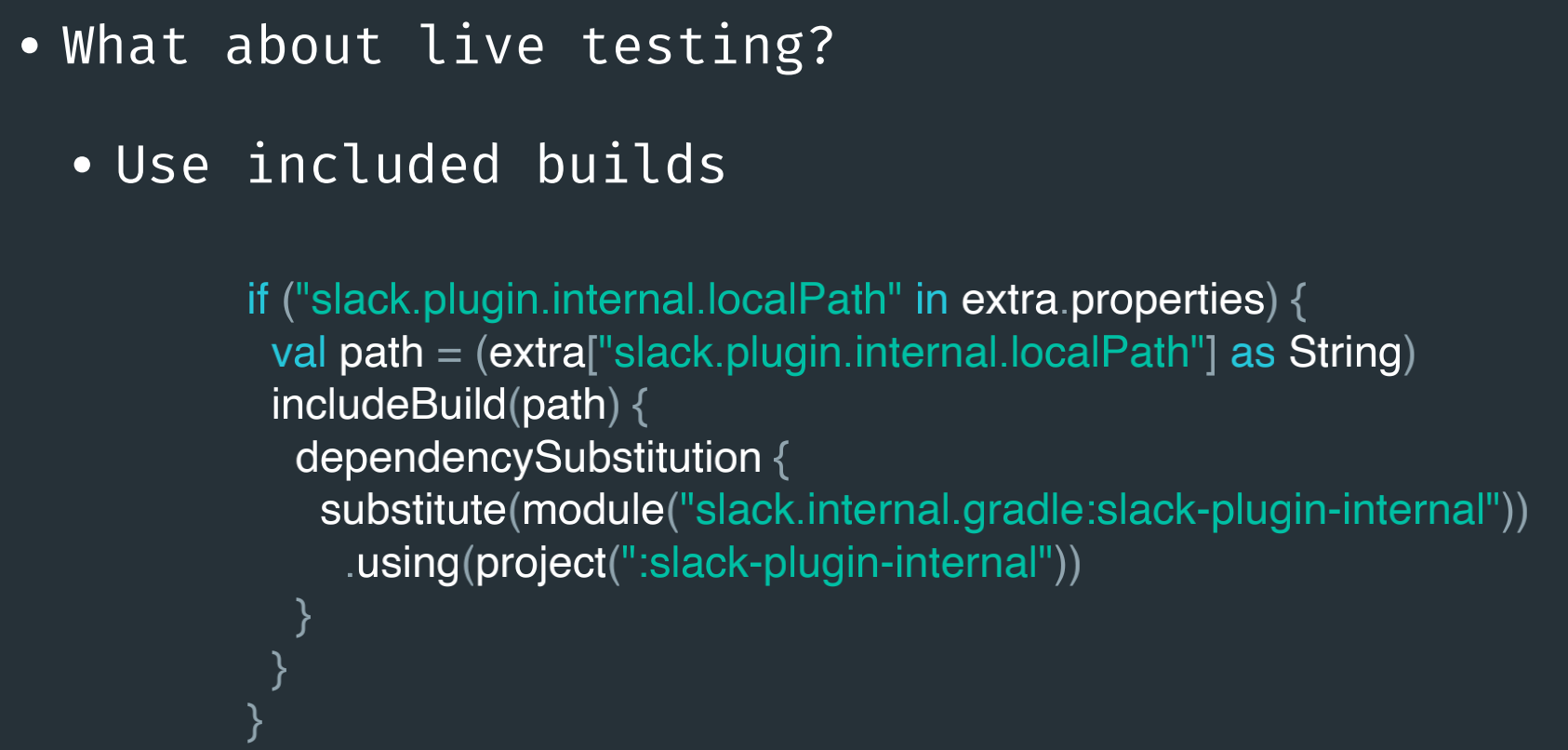




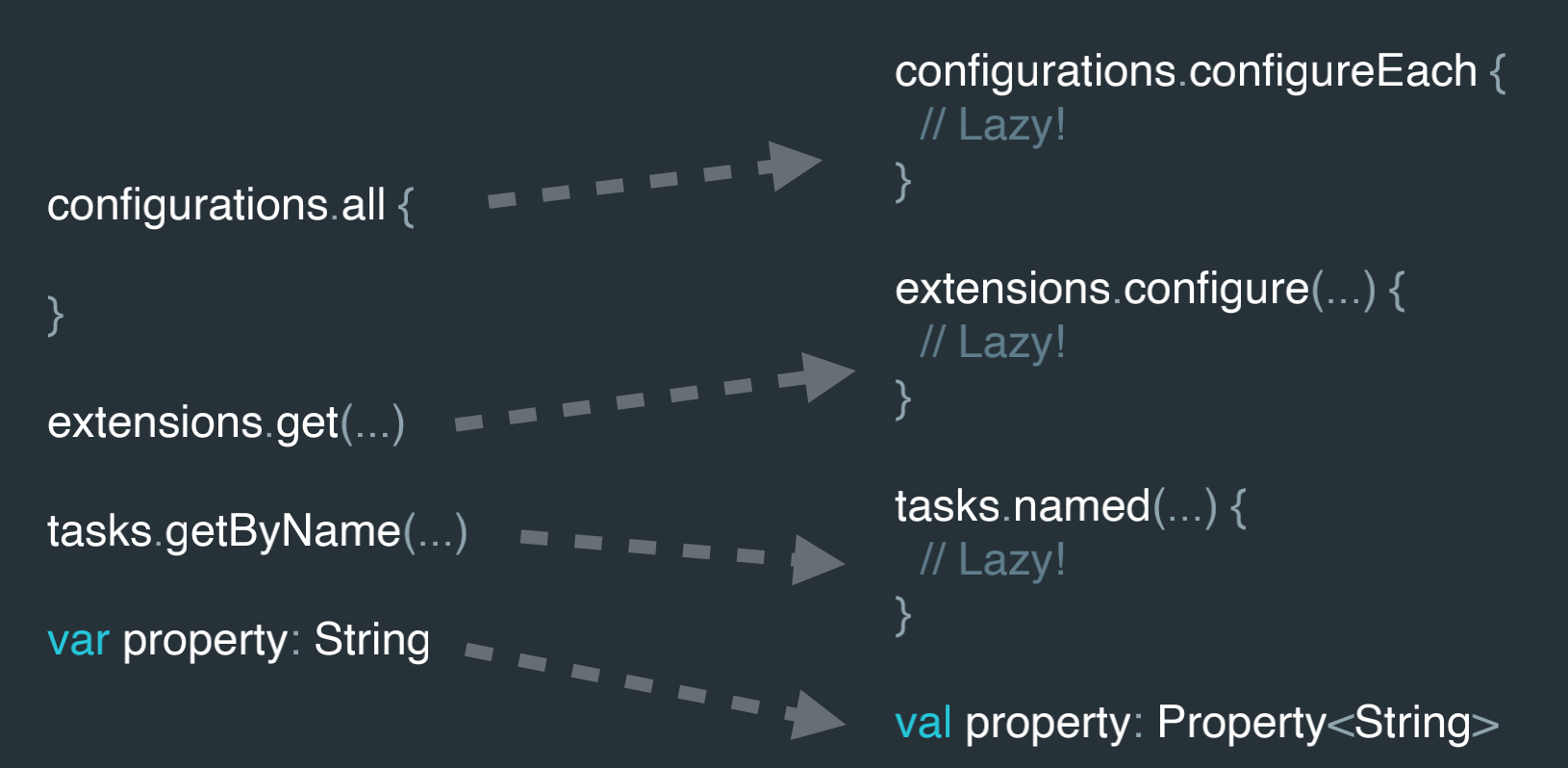


1. buildSrc

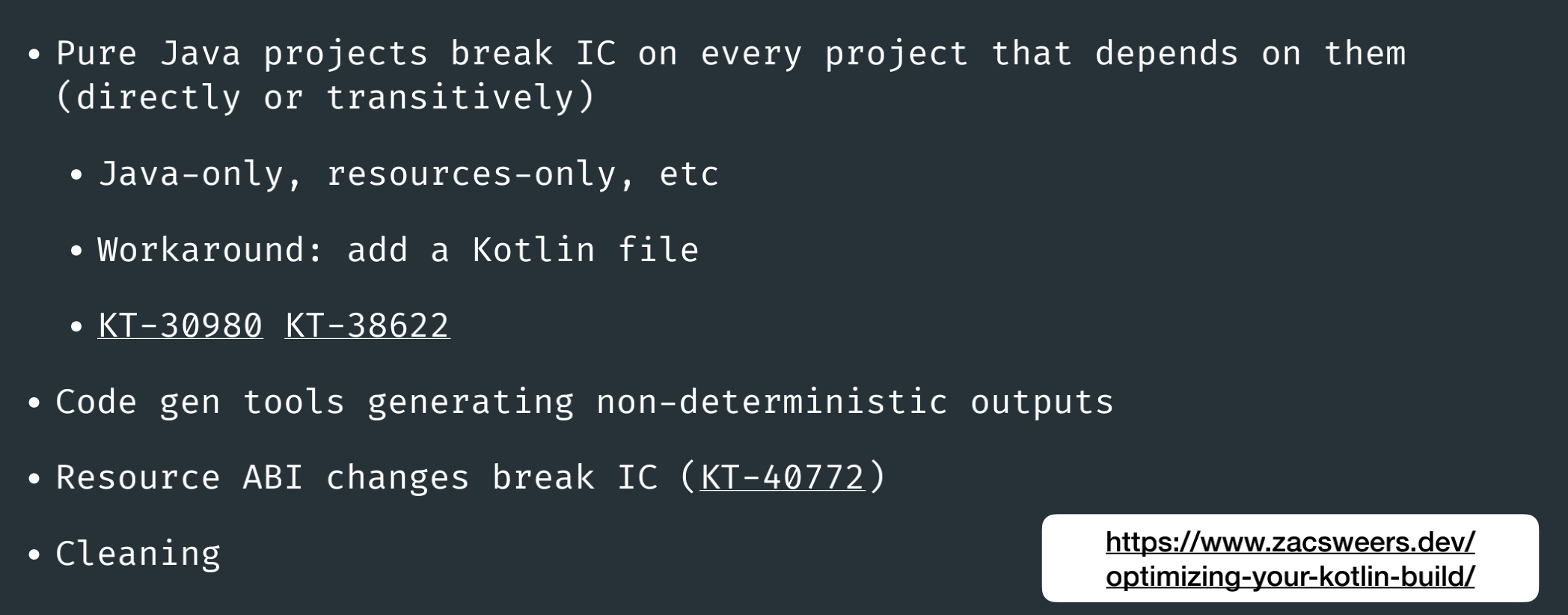




1. ./gradlew clean
2. 将 NDK 项目移动到单独的项目
3. Gradle API



1. IC



1. 配置

