@droidcon | @zhaosiruo | @GrouponEng

LINT一生推

你的360全方位贴身保镖

Hi 大家好! 欢迎来到lint一生推，你的360全方位贴身保镖。我是赵思若，或者Snow。我来自硅谷的Groupon。这里是我的联系方式。(next Slide)

联系方式

赵思若（SNOW）

sirzhao@groupon.com

@zhaosiruo

+zhao siruo

snow赵思若

欢迎大家email我， 或者在twitter上@赵思若，在g＋@zhao空格siruo。另外考虑到国内twitter和gmail都不方便，我昨天特意开通了一个微博，snow赵思若。

另外有一个小小的声明，我的计算机术语都是在美国上学的时候学到的，为了这次在DroidCon Beijing能用中文来讲，我查阅了很多资料，也请教了很多朋友，但是如果还是有翻译不到位的地方希望大家见谅，并且欢迎在结束后找我指正。

主办方，改slides，学习，lint发展，新feature，反馈，keynote，以中文为准

梗概

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析,
(Click)什么是Lint,
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs,
(Click)和如何测试我们的Lint Check.
(Click)那之后, 我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT
- 私人定制时间－LINT

  JavaConstructorDetector.java

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT
- 私人定制时间－LINT

  JavaConstructorDetector.java

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT
- 私人定制时间－LINT

JavaConstructorDetector.java

LINT RULE的组成

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT
- 私人定制时间－LINT

JavaConstructorDetector.java

LINT RULE的组成

AST 和 UAST

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析,
(Click)什么是Lint,
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs,
(Click)和如何测试我们的Lint Check.
(Click)那之后, 我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT
- 私人定制时间－LINT

JavaConstructorDetector.java

LINT RULE的组成

AST 和 UAST

测试

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT
- 私人定制时间－LINT

  JavaConstructorDetector.java

  LINT RULE的组成

  AST 和 UAST

  测试
- 常用安卓静态代码分析工具对比

那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

梗概

- 静态代码分析介绍
- 什么是LINT
- 私人定制时间－LINT

  JavaConstructorDetector.java

  LINT RULE的组成

  AST 和 UAST

  测试
- 常用安卓静态代码分析工具对比

问答

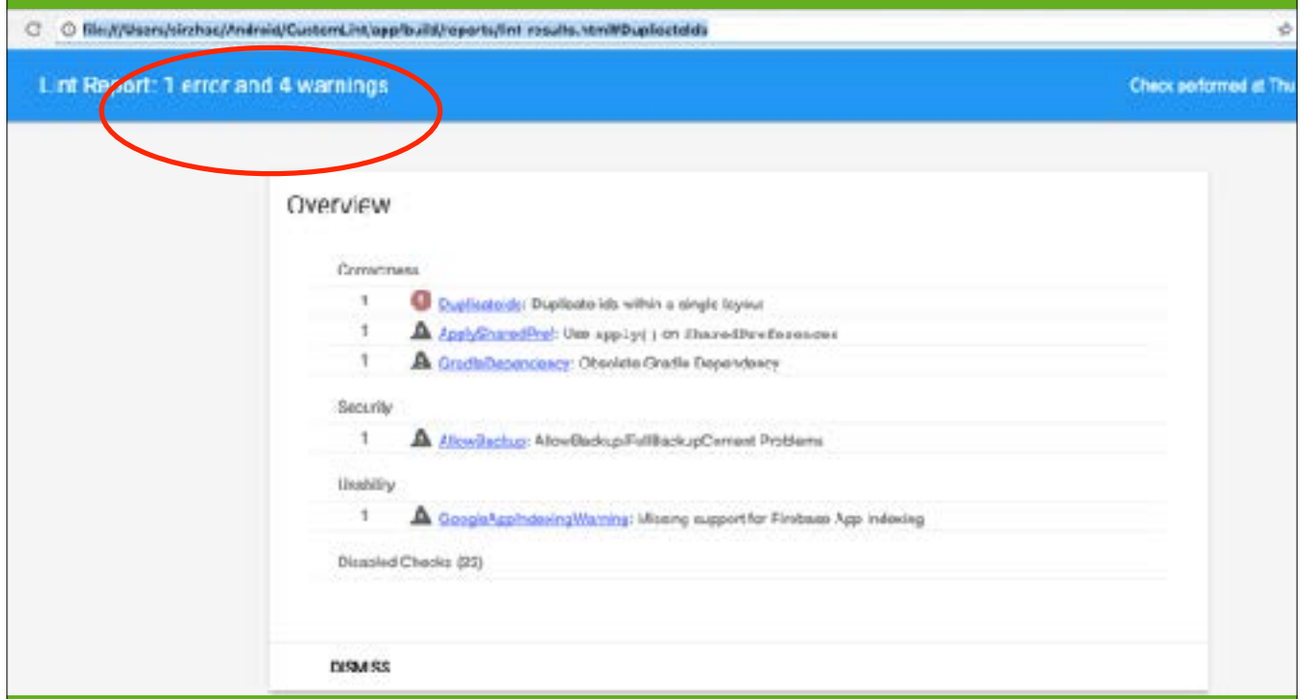
那么今天我们的话题是静态代码分析工具Lint，具体点说是Android Lint（当然啊，因为这个是DroidCon嘛）(break)
(Click)我会先简单介绍一下静态代码分析，
(Click)什么是Lint，
(Click)然后直接进入我们的重头戏- 定制你的私人专属Lint!
我会用lint来为参数过多的构造函数建议一个生成器模式
(Click)在这个例子里，我会讲解一个Lint Rule的组成部分
(Click)Lint使用的AST and UAST APIs，
(Click)和如何测试我们的Lint Check.
(Click)那之后，我会介绍一些常用的静态代码分析工具，并且给出他们的具体用法指导意见。
(Click)最后，大家的最爱， Q&A。请大家把问题都留在这里，因为我们今天要讲的内容信息量确实很大.
（看观众）那么，大家和我一起做一个深呼吸，进入Lint的世界！(new slide)

这是什么?

见过这个的同学，请举手。

谢谢，请放下。如果你有开发安卓一段时间，这个应该很熟悉了。这是一个Lint报告的截图。在这份报告里有(click)1个错误和4个警告，它们的类别是(click)Correctness，(click)Security，和(click)Usability。如果我想要具体去看某一个问题，比如第一个，(click)DuplicateIds，我只要点击这个问题，然后就会被指引到相关的部分。

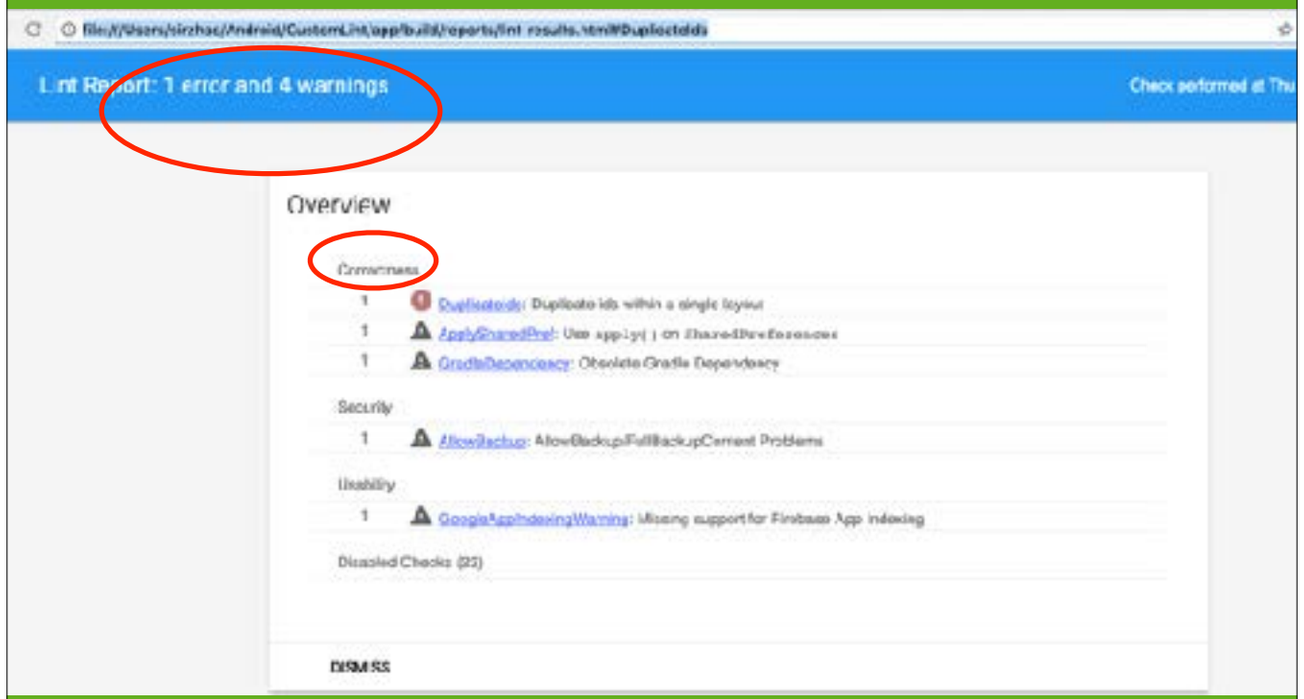
这是什么?



见过这个的同学，请举手。

谢谢，请放下。如果你有开发安卓一段时间，这个应该很熟悉了。这是一个Lint报告的截图。在这份报告里有(click)1个错误和4个警告，它们的类别是(click)Correctness，(click)Security，和(click)Usability。如果我想要具体去看某一个问题，比如第一个，(click)DuplicateIds，我只要点击这个问题，然后就会被指引到相关的部分。

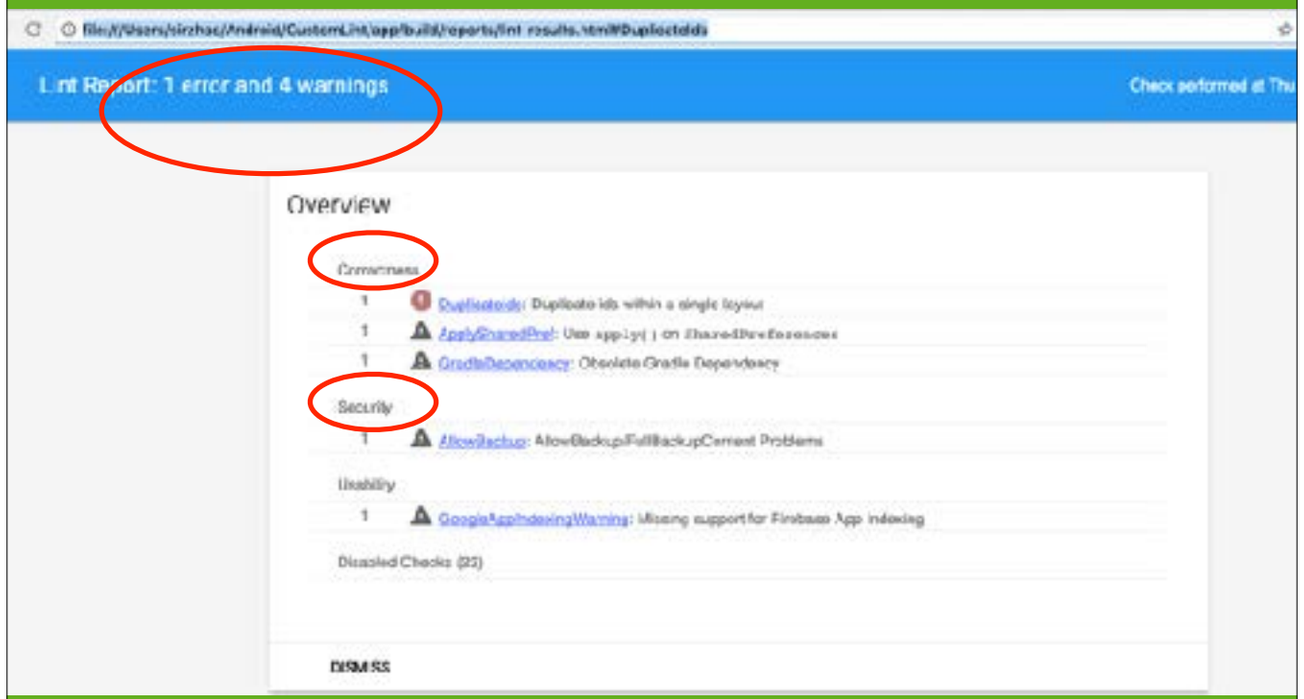
这是什么?



见过这个的同学，请举手。

谢谢，请放下。如果你有开发安卓一段时间，这个应该很熟悉了。这是一个Lint报告的截图。在这份报告里有(click)1个错误和4个警告，它们的类别是(click)Correctness，(click)Security，和(click)Usability。如果我想要具体去看某一个问题，比如第一个，(click)DuplicateIds，我只要点击这个问题，然后就会被指引到相关的部分。

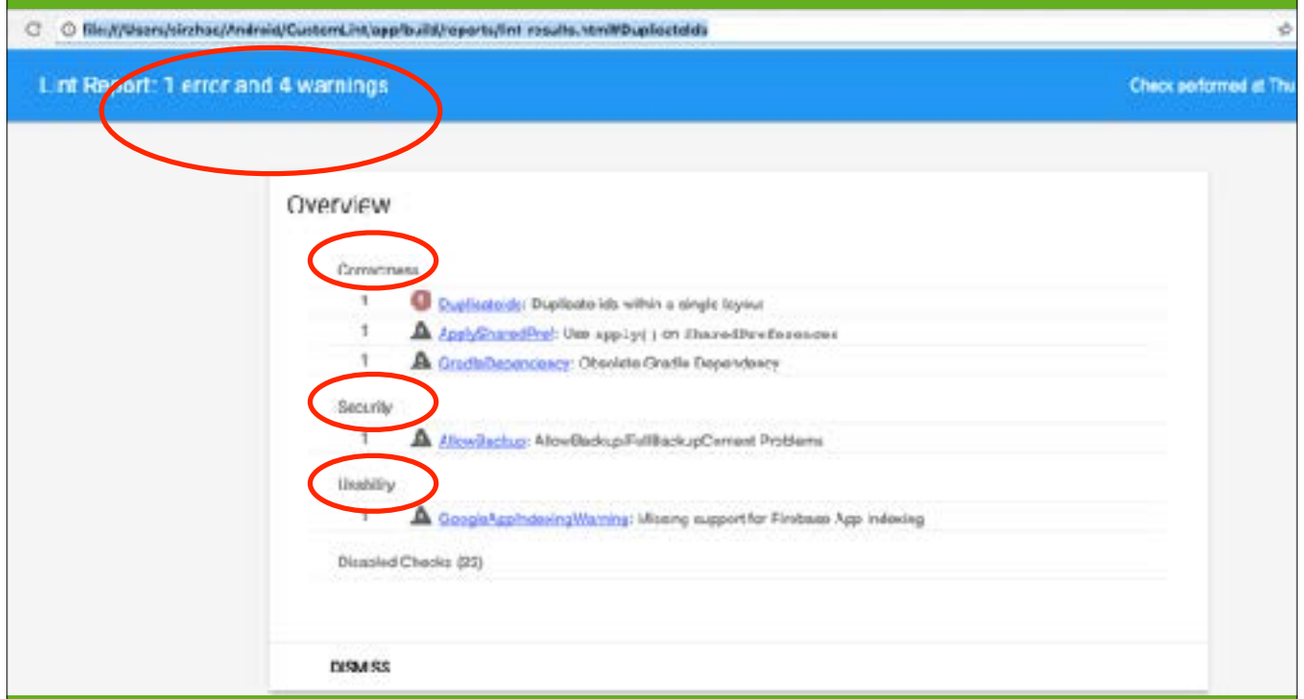
这是什么?



见过这个的同学，请举手。

谢谢，请放下。如果你有开发安卓一段时间，这个应该很熟悉了。这是一个Lint报告的截图。在这份报告里有(click)1个错误和4个警告，它们的类别是(click)Correctness，(click)Security，和(click)Usability。如果我想要具体去看某一个问题，比如第一个，(click)DuplicateIds，我只要点击这个问题，然后就会被指引到相关的部分。

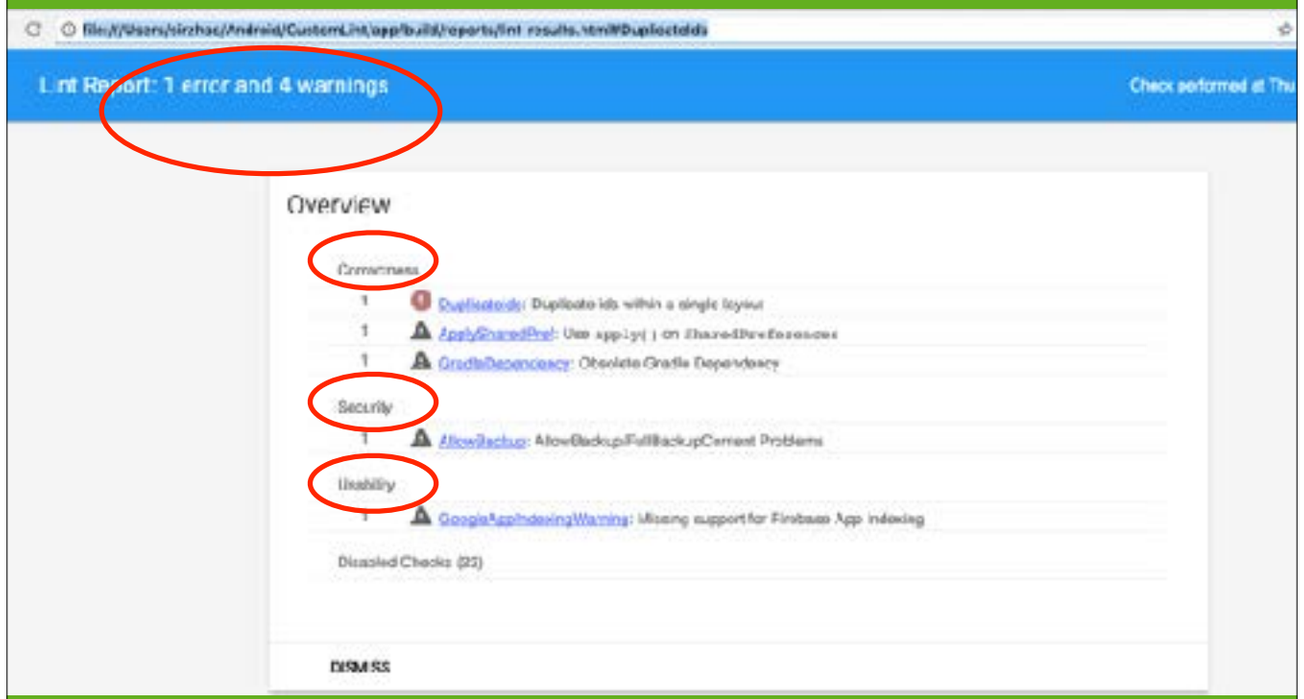
这是什么?



见过这个的同学，请举手。

谢谢，请放下。如果你有开发安卓一段时间，这个应该很熟悉了。这是一个Lint报告的截图。在这份报告里有(click)1个错误和4个警告，它们的类别是(click)Correctness，(click)Security，和(click)Usability。如果我想要具体去看某一个问题，比如第一个，(click)DuplicateIds，我只要点击这个问题，然后就会被指引到相关的部分。

这是什么?



见过这个的同学，请举手。

谢谢，请放下。如果你有开发安卓一段时间，这个应该很熟悉了。这是一个Lint报告的截图。在这份报告里有(click)1个错误和4个警告，它们的类别是(click)Correctness，(click)Security，和(click)Usability。如果我想要具体去看某一个问题，比如第一个，(click)DuplicateIds，我只要点击这个问题，然后就会被指引到相关的部分。

LINT 报告

像这样。这里我们可以看到(click)这个问题具体发生的位置, (click)代码节选, (click)这个问题的id, (click)类别, (click)严重程度 和 (click)优先级.

LINT 报告

像这样。这里我们可以看到(click)这个问题具体发生的位置, (click)代码节选, (click)这个问题的id, (click)类别, (click)严重程度 和 (click)优先级.

LINT 报告

像这样。这里我们可以看到(click)这个问题具体发生的位置, (click)代码节选, (click)这个问题的id, (click)类别, (click)严重程度 和 (click)优先级.

LINT 报告

像这样。这里我们可以看到(click)这个问题具体发生的位置, (click)代码节选, (click)这个问题的id, (click)类别, (click)严重程度 和 (click)优先级.

LINT 报告

像这样。这里我们可以看到(click)这个问题具体发生的位置, (click)代码节选, (click)这个问题的id, (click)类别, (click)严重程度 和 (click)优先级.

LINT 报告

像这样。这里我们可以看到(click)这个问题具体发生的位置, (click)代码节选, (click)这个问题的id, (click)类别, (click)严重程度 和 (click)优先级.

LINT 报告

像这样。这里我们可以看到(click)这个问题具体发生的位置, (click)代码节选, (click)这个问题的id, (click)类别, (click)严重程度 和 (click)优先级.

LINT 报告

这个时候如果我点击“explain”, 就会跳出更具体的解释, 像这样 (play the video). 希望这样我们就有足够的信息来解决这个问题了。

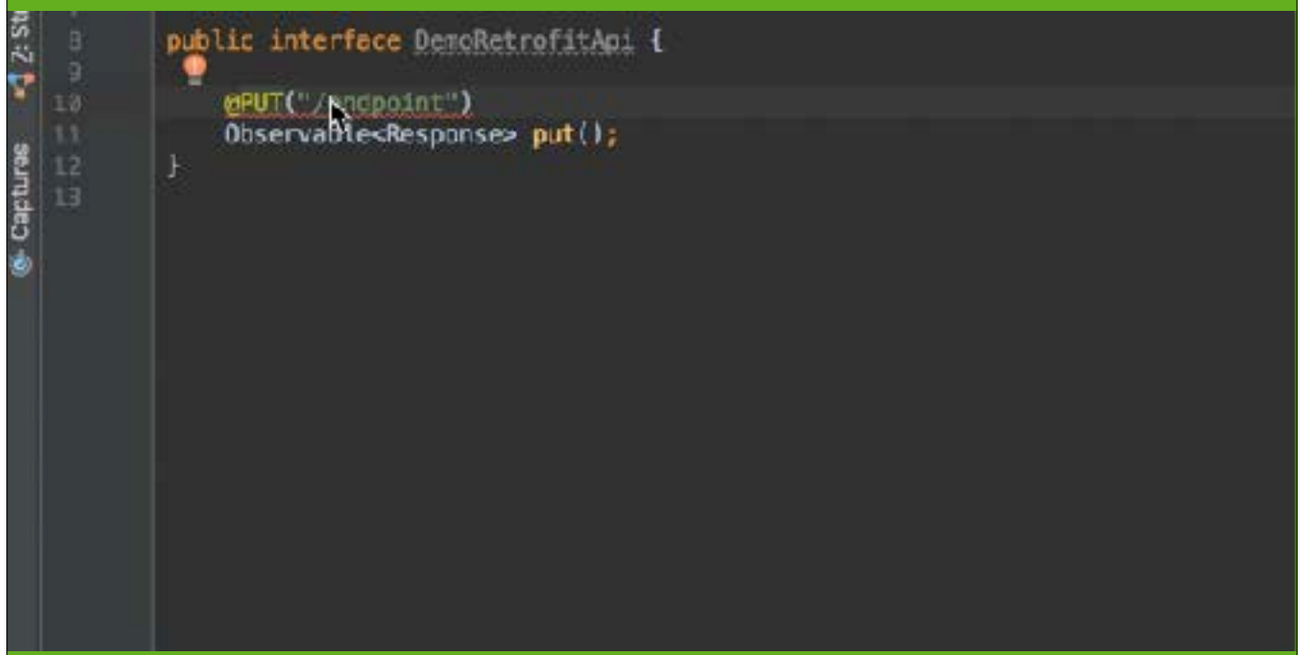
Duplicate Ids是一个现有规则，但是在之后的定制Lint代码中我们会看到这些功能也同样适用。
(look at audience and break)那么，你准备好迎接lint的魔法了吗?

LINT 报告

这个时候如果我点击“explain”, 就会跳出更具体的解释, 像这样 (play the video). 希望这样我们就有足够的信息来解决这个问题了。

Duplicate Ids是一个现有规则，但是在之后的定制Lint代码中我们会看到这些功能也同样适用。
(look at audience and break)那么，你准备好迎接lint的魔法了吗?

LINT 魔法



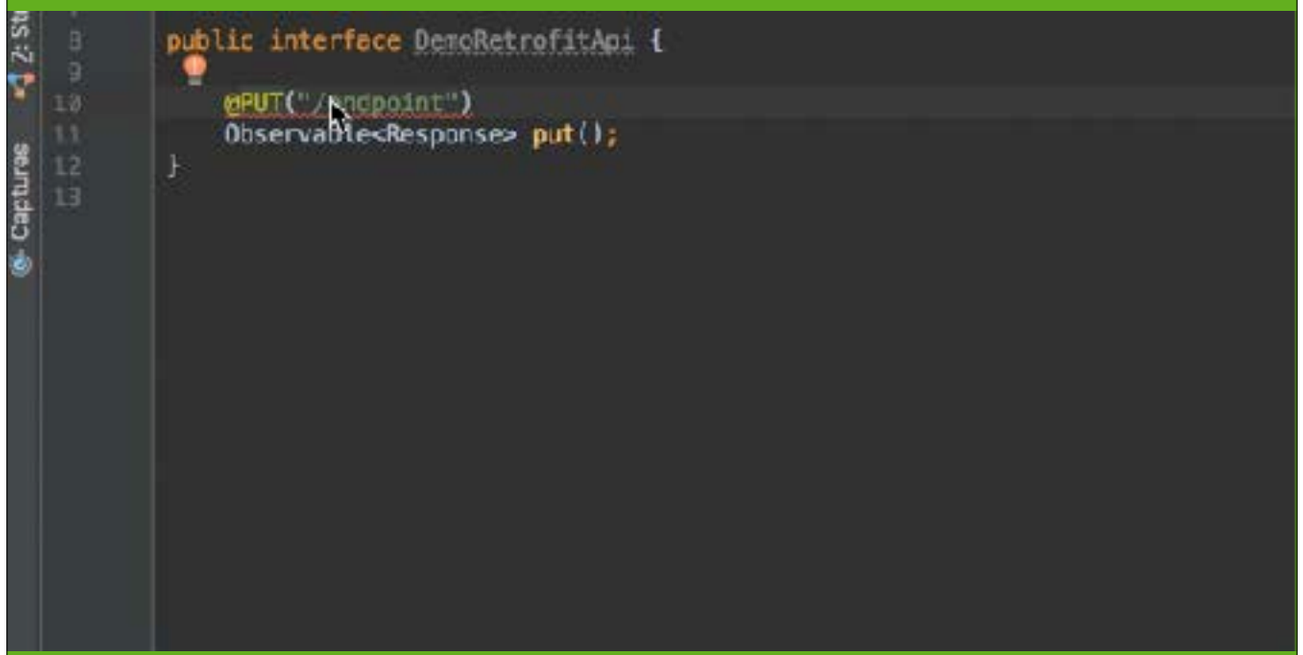
Groupon在今年年初的时候开始使用Retrofit用来处理网络请求。为此我编写了一个定制Lint规则。它是用来禁止端点以斜线开始，以便于保存完整的请求url。这个函式库目前正在开源申请过程中，届时我会在Twitter上发布。到时候欢迎Retrofit的用户们使用这个简单但非常有用的函式库。

在这段视频中，你将会看到Lint不仅会指出这个问题，而且还可以一键修正。(play the video)

如果你不小心错过了Android Studio Editor里面的警告，这个规则的警告或者错误提示还会出现在你的Lint报告中. 就像我们在前几页看到的那样。

是不是很方便?
我刚才提到Lint是个静态代码分析工具. 那么，什么是静态代码分析?(next slide)

LINT 魔法



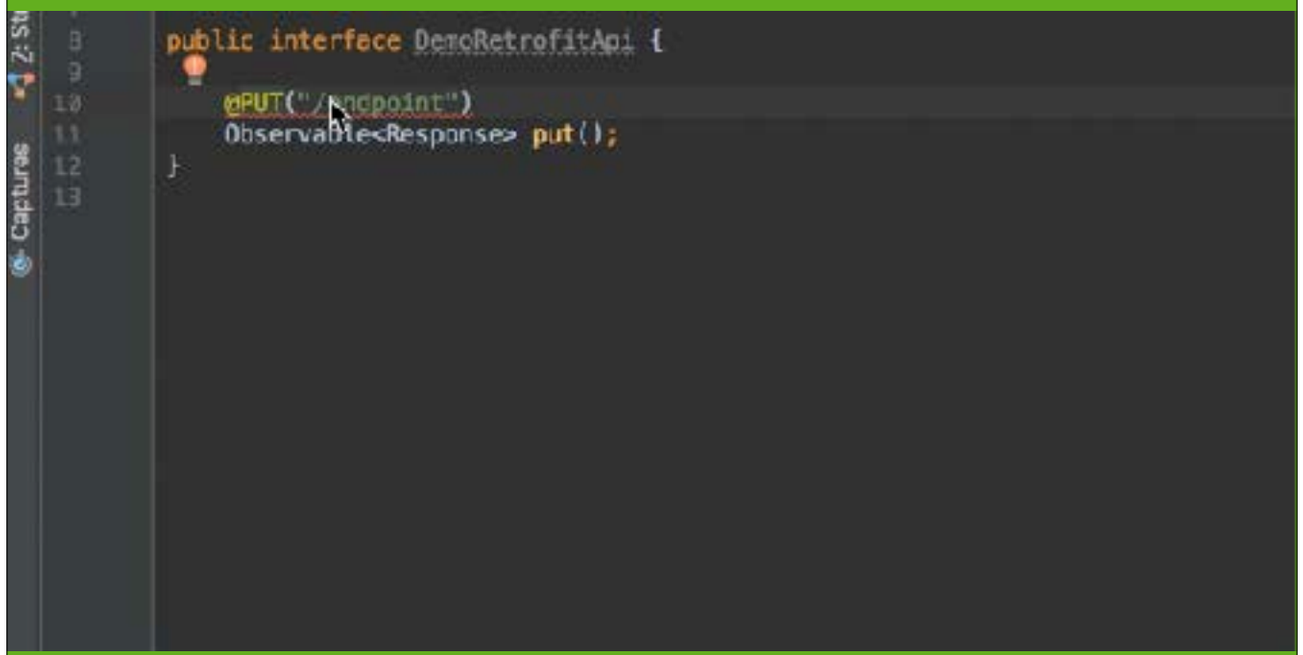
Groupon在今年年初的时候开始使用Retrofit用来处理网络请求。为此我编写了一个定制Lint规则。它是用来禁止端点以斜线开始，以便于保存完整的请求url。这个函式库目前正在开源申请过程中，届时我会在Twitter上发布。到时候欢迎Retrofit的用户们使用这个简单但非常有用的函式库。

在这段视频中，你将会看到Lint不仅会指出这个问题，而且还可以一键修正。(play the video)

如果你不小心错过了Android Studio Editor里面的警告，这个规则的警告或者错误提示还会出现在你的Lint报告中. 就像我们在前几页看到的那样。

是不是很方便?
我刚才提到Lint是个静态代码分析工具. 那么，什么是静态代码分析?(next slide)

LINT 魔法



Groupon在今年年初的时候开始使用Retrofit用来处理网络请求。为此我编写了一个定制Lint规则。它是用来禁止端点以斜线开始，以便于保存完整的请求url。这个函式库目前正在开源申请过程中，届时我会在Twitter上发布。到时候欢迎Retrofit的用户们使用这个简单但非常有用的函式库。

在这段视频中，你将会看到Lint不仅会指出这个问题，而且还可以一键修正。(play the video)

如果你不小心错过了Android Studio Editor里面的警告，这个规则的警告或者错误提示还会出现在你的Lint报告中. 就像我们在前几页看到的那样。

是不是很方便?
我刚才提到Lint是个静态代码分析工具. 那么，什么是静态代码分析?(next slide)

静态代码分析

很简单，静态代码分析其实就是自动代码审查，它可以由多种静态代码分析工具来完成，比如，喵星人(click)(来跟我家Mika打个招呼，也许是时候告诉大家其实我同事的代码审查都是Mika完成的, 这样子我就有更多时间来钻研更有趣的项目，比如，Lint)!

那么回到静态代码分析, 它能够帮助防止很多问题的产生，自然也就能提高代码质量.

(click)虽然计算机科学这门学科的历史并不长，但是静态代码分析的起源就可以追溯到70年代。现在，越来越多的静态代码分析工具正在走向IDE，提供即时自动代码审查。Android Lint是其中的佼佼者, 它就好像一个全天候的贴身保镖。一直在那里帮助我们向完美代码走近。

静态代码分析

很简单，静态代码分析其实就是自动代码审查，它可以由多种静态代码分析工具来完成，比如，喵星人(click)(来跟我家Mika打个招呼，也许是时候告诉大家其实我同事的代码审查都是Mika完成的, 这样子我就有更多时间来钻研更有趣的项目，比如，Lint)!

那么回到静态代码分析, 它能够帮助防止很多问题的产生，自然也就能提高代码质量.

(click)虽然计算机科学这门学科的历史并不长，但是静态代码分析的起源就可以追溯到70年代。现在，越来越多的静态代码分析工具正在走向IDE，提供即时自动代码审查。Android Lint是其中的佼佼者, 它就好像一个全天候的贴身保镖。一直在那里帮助我们向完美代码走近。

自动化代码审查

很简单，静态代码分析其实就是自动代码审查，它可以由多种静态代码分析工具来完成，比如，喵星人(click)(来跟我家Mika打个招呼，也许是时候告诉大家其实我同事的代码审查都是Mika完成的, 这样子我就有更多时间来钻研更有趣的项目，比如，Lint)!

那么回到静态代码分析, 它能够帮助防止很多问题的产生，自然也就能提高代码质量.

(click)虽然计算机科学这门学科的历史并不长，但是静态代码分析的起源就可以追溯到70年代。现在，越来越多的静态代码分析工具正在走向IDE，提供即时自动代码审查。Android Lint是其中的佼佼者, 它就好像一个全天候的贴身保镖。一直在那里帮助我们向完美代码走近。

自动化代码审查

编译时间-> IDE

很简单，静态代码分析其实就是自动代码审查，它可以由多种静态代码分析工具来完成，比如，喵星人(click)(来跟我家Mika打个招呼，也许是时候告诉大家其实我同事的代码审查都是Mika完成的, 这样子我就有更多时间来钻研更有趣的项目，比如，Lint)!

那么回到静态代码分析, 它能够帮助防止很多问题的产生，自然也就能提高代码质量.

(click)虽然计算机科学这门学科的历史并不长，但是静态代码分析的起源就可以追溯到70年代。现在，越来越多的静态代码分析工具正在走向IDE，提供即时自动代码审查。Android Lint是其中的佼佼者, 它就好像一个全天候的贴身保镖。一直在那里帮助我们向完美代码走近。

LINT 介绍

(click)Lint 是谷歌的为安卓量身打造的静态分析工具(click)。它的创造者是谷歌安卓工具组的Tor Norbye. 如果你和我一样是“Android Developer Backstage” 这个Podcast的忠实用户，你应该对作为主持的他很熟悉了。顺便说一句我强烈向大家推荐这个Podcast，上下班路上带着耳机就能了解到安卓的内部消息和最新动向。他们最近就有一期tooling around讲到了lint的最新进展。

我之前说过，静态代码分析其实可以追溯到70年代，1979年的Unix V7 Lint就是先驱者之一。据Tor说，他之所以会取名为Lint一方面是向前辈致敬，另一方面也是方便大家一目了然这是一个静态代码分析工具。不过千万不要误会，安卓Lint和Unix / Linux Lint并不共享源代码，而是彼此独立的项目。

那么，因为Lint最初是为安卓设计的，很多现有规则都带有强烈的“安卓色彩”, 比如 (click)缺少权限许可（像我使用了相机的api但是app本身并没有相机许可）, (click)布局过深(默认的最大深度是10，但是也可以根据自己的需要用环境变量来设定这个值，比如我可以说我只允许5层深度，不然不仅影响performance，也会大大降低代码可读性), (click)shared preferences要用apply() 而不是commit() , (click) rtl 兼容性检查，这个对于为国际市场打造的产品来说非常方便.

Lint 是在2012时被加入到ADT 16的，那时只有大概100个自带规则，现在这个数字已经直逼300. (click)欢迎大家查阅官方目录.

那么你还等什么呢？快来将Lint整合进入你的安卓代码。

LINT 介绍

谷歌为安卓量身定制的静态代分析工具

(click)Lint 是谷歌的为安卓量身打造的静态分析工具(click)。它的创造者是谷歌安卓工具组的Tor Norbye. 如果你和我一样是“Android Developer Backstage” 这个Podcast的忠实用户，你应该对作为主持的他很熟悉了。顺便说一句我强烈向大家推荐这个Podcast，上下班路上带着耳机就能了解到安卓的内部消息和最新动向。他们最近就有一期tooling around讲到了lint的最新进展。

我之前说过，静态代码分析其实可以追溯到70年代，1979年的Unix V7 Lint就是先驱者之一。据Tor说，他之所以会取名为Lint一方面是向前辈致敬，另一方面也是方便大家一目了然这是一个静态代码分析工具。不过千万不要误会，安卓Lint和Unix / Linux Lint并不共享源代码，而是彼此独立的项目。

那么，因为Lint最初是为安卓设计的，很多现有规则都带有强烈的“安卓色彩”, 比如 (click)缺少权限许可（像我使用了相机的api但是app本身并没有相机许可）, (click)布局过深(默认的最大深度是10，但是也可以根据自己的需要用环境变量来设定这个值，比如我可以说我只允许5层深度，不然不仅影响performance，也会大大降低代码可读性), (click)shared preferences要用apply() 而不是commit() , (click) rtl 兼容性检查，这个对于为国际市场打造的产品来说非常方便.

Lint 是在2012时被加入到ADT 16的，那时只有大概100个自带规则，现在这个数字已经直逼300. (click)欢迎大家查阅官方目录.

那么你还等什么呢？快来将Lint整合进入你的安卓代码。

LINT 介绍

谷歌为安卓量身定制的静态代分析工具

– 缺少权限许可

(click)Lint 是谷歌的为安卓量身打造的静态分析工具(click)。它的创造者是谷歌安卓工具组的Tor Norbye. 如果你和我一样是“Android Developer Backstage” 这个Podcast的忠实用户，你应该对作为主持的他很熟悉了。顺便说一句我强烈向大家推荐这个Podcast，上下班路上带着耳机就能了解到安卓的内部消息和最新动向。他们最近就有一期tooling around讲到了lint的最新进展。

我之前说过，静态代码分析其实可以追溯到70年代，1979年的Unix V7 Lint就是先驱者之一。据Tor说，他之所以会取名为Lint一方面是向前辈致敬，另一方面也是方便大家一目了然这是一个静态代码分析工具。不过千万不要误会，安卓Lint和Unix / Linux Lint并不共享源代码，而是彼此独立的项目。

那么，因为Lint最初是为安卓设计的，很多现有规则都带有强烈的“安卓色彩”, 比如 (click)缺少权限许可（像我使用了相机的api但是app本身并没有相机许可）, (click)布局过深(默认的最大深度是10，但是也可以根据自己的需要用环境变量来设定这个值，比如我可以说我只允许5层深度，不然不仅影响performance，也会大大降低代码可读性), (click)shared preferences要用apply() 而不是commit() , (click) rtl 兼容性检查，这个对于为国际市场打造的产品来说非常方便.

Lint 是在2012时被加入到ADT 16的，那时只有大概100个自带规则，现在这个数字已经直逼300. (click)欢迎大家查阅官方目录.

那么你还等什么呢？快来将Lint整合进入你的安卓代码。

LINT 介绍

谷歌为安卓量身定制的静态代分析工具

– 缺少权限许可

– 布局过深

(click)Lint 是谷歌的为安卓量身打造的静态分析工具(click)。它的创造者是谷歌安卓工具组的Tor Norbye. 如果你和我一样是“Android Developer Backstage” 这个Podcast的忠实用户，你应该对作为主持的他很熟悉了。顺便说一句我强烈向大家推荐这个Podcast，上下班路上带着耳机就能了解到安卓的内部消息和最新动向。他们最近就有一期tooling around讲到了lint的最新进展。

我之前说过，静态代码分析其实可以追溯到70年代，1979年的Unix V7 Lint就是先驱者之一。据Tor说，他之所以会取名为Lint一方面是向前辈致敬，另一方面也是方便大家一目了然这是一个静态代码分析工具。不过千万不要误会，安卓Lint和Unix / Linux Lint并不共享源代码，而是彼此独立的项目。

那么，因为Lint最初是为安卓设计的，很多现有规则都带有强烈的“安卓色彩”, 比如 (click)缺少权限许可（像我使用了相机的api但是app本身并没有相机许可）, (click)布局过深(默认的最大深度是10，但是也可以根据自己的需要用环境变量来设定这个值，比如我可以说我只允许5层深度，不然不仅影响performance，也会大大降低代码可读性), (click)shared preferences要用apply() 而不是commit() , (click) rtl 兼容性检查，这个对于为国际市场打造的产品来说非常方便.

Lint 是在2012时被加入到ADT 16的，那时只有大概100个自带规则，现在这个数字已经直逼300. (click)欢迎大家查阅官方目录.

那么你还等什么呢？快来将Lint整合进入你的安卓代码。

LINT 介绍

谷歌为安卓量身定制的静态代分析工具

- 缺少权限许可
- 布局过深
- SHAREDPREFERENCES 中使用 APPLY()

(click)Lint 是谷歌的为安卓量身打造的静态分析工具(click)。它的创造者是谷歌安卓工具组的Tor Norbye. 如果你和我一样是“Android Developer Backstage” 这个Podcast的忠实用户，你应该对作为主持的他很熟悉了。顺便说一句我强烈向大家推荐这个Podcast，上下班路上带着耳机就能了解到安卓的内部消息和最新动向。他们最近就有一期tooling around讲到了lint的最新进展。

我之前说过，静态代码分析其实可以追溯到70年代，1979年的Unix V7 Lint就是先驱者之一。据Tor说，他之所以会取名为Lint一方面是向前辈致敬，另一方面也是方便大家一目了然这是一个静态代码分析工具。不过千万不要误会，安卓Lint和Unix / Linux Lint并不共享源代码，而是彼此独立的项目。

那么，因为Lint最初是为安卓设计的，很多现有规则都带有强烈的“安卓色彩”, 比如 (click)缺少权限许可（像我使用了相机的api但是app本身并没有相机许可）, (click)布局过深(默认的最大深度是10，但是也可以根据自己的需要用环境变量来设定这个值，比如我可以说我只允许5层深度，不然不仅影响performance，也会大大降低代码可读性), (click)shared preferences要用apply() 而不是commit() , (click) rtl 兼容性检查，这个对于为国际市场打造的产品来说非常方便.

Lint 是在2012时被加入到ADT 16的，那时只有大概100个自带规则，现在这个数字已经直逼300. (click)欢迎大家查阅官方目录.

那么你还等什么呢？快来将Lint整合进入你的安卓代码。

LINT 介绍

谷歌为安卓量身定制的静态代分析工具

– 缺少权限许可

– 布局过深

– SHAREDPREFERENCES 中使用 APPLY()

– RTL 兼容性

….

(click)Lint 是谷歌的为安卓量身打造的静态分析工具(click)。它的创造者是谷歌安卓工具组的Tor Norbye. 如果你和我一样是“Android Developer Backstage” 这个Podcast的忠实用户，你应该对作为主持的他很熟悉了。顺便说一句我强烈向大家推荐这个Podcast，上下班路上带着耳机就能了解到安卓的内部消息和最新动向。他们最近就有一期tooling around讲到了lint的最新进展。

我之前说过，静态代码分析其实可以追溯到70年代，1979年的Unix V7 Lint就是先驱者之一。据Tor说，他之所以会取名为Lint一方面是向前辈致敬，另一方面也是方便大家一目了然这是一个静态代码分析工具。不过千万不要误会，安卓Lint和Unix / Linux Lint并不共享源代码，而是彼此独立的项目。

那么，因为Lint最初是为安卓设计的，很多现有规则都带有强烈的“安卓色彩”, 比如 (click)缺少权限许可（像我使用了相机的api但是app本身并没有相机许可）, (click)布局过深(默认的最大深度是10，但是也可以根据自己的需要用环境变量来设定这个值，比如我可以说我只允许5层深度，不然不仅影响performance，也会大大降低代码可读性), (click)shared preferences要用apply() 而不是commit() , (click) rtl 兼容性检查，这个对于为国际市场打造的产品来说非常方便.

Lint 是在2012时被加入到ADT 16的，那时只有大概100个自带规则，现在这个数字已经直逼300. (click)欢迎大家查阅官方目录.

那么你还等什么呢？快来将Lint整合进入你的安卓代码。

LINT 介绍

谷歌为安卓量身定制的静态代分析工具

– 缺少权限许可

– 布局过深

– SHAREDPREFERENCES 中使用 APPLY()

– RTL 兼容性

….

官方指导

自带规则

(click)Lint 是谷歌的为安卓量身打造的静态分析工具(click)。它的创造者是谷歌安卓工具组的Tor Norbye. 如果你和我一样是“Android Developer Backstage” 这个Podcast的忠实用户，你应该对作为主持的他很熟悉了。顺便说一句我强烈向大家推荐这个Podcast，上下班路上带着耳机就能了解到安卓的内部消息和最新动向。他们最近就有一期tooling around讲到了lint的最新进展。

我之前说过，静态代码分析其实可以追溯到70年代，1979年的Unix V7 Lint就是先驱者之一。据Tor说，他之所以会取名为Lint一方面是向前辈致敬，另一方面也是方便大家一目了然这是一个静态代码分析工具。不过千万不要误会，安卓Lint和Unix / Linux Lint并不共享源代码，而是彼此独立的项目。

那么，因为Lint最初是为安卓设计的，很多现有规则都带有强烈的“安卓色彩”, 比如 (click)缺少权限许可（像我使用了相机的api但是app本身并没有相机许可）, (click)布局过深(默认的最大深度是10，但是也可以根据自己的需要用环境变量来设定这个值，比如我可以说我只允许5层深度，不然不仅影响performance，也会大大降低代码可读性), (click)shared preferences要用apply() 而不是commit() , (click) rtl 兼容性检查，这个对于为国际市场打造的产品来说非常方便.

Lint 是在2012时被加入到ADT 16的，那时只有大概100个自带规则，现在这个数字已经直逼300. (click)欢迎大家查阅官方目录.

那么你还等什么呢？快来将Lint整合进入你的安卓代码。

整合 LINT

那么我们需要做些什么?

你可能已经知道了，我们不需要做任何事! 没错，正如我刚才所说，现有的规则是包含在ADT里的，所以(Click mouse to show the picture of Mika) 你可以像我的Mika一样，闭着眼建立一个Android项目，而不费吹灰之力得到这200多条规则!

(Pause) 等一下, 难道这就是说你会永远被RTL的检查所限制? 即使你只是单纯地需要一个从左至右的中文App? 那么，这个时候Lint配置就是你的好朋友.

那么我们需要做些什么?

你可能已经知道了，我们不需要做任何事! 没错，正如我刚才所说，现有的规则是包含在ADT里的，所以(Click mouse to show the picture of Mika) 你可以像我的Mika一样，闭着眼建立一个Android项目，而不费吹灰之力得到这200多条规则!

(Pause) 等一下, 难道这就是说你会永远被RTL的检查所限制? 即使你只是单纯地需要一个从左至右的中文App? 那么，这个时候Lint配置就是你的好朋友.

LINT 配置

你可以从多种级别对Lint进行配置:(click)
(click)全局 (也就是整个项目)
(click)对于某一个项目模块
(click)或者是生产模块
(click)测试模块(click)
甚至是针对现在打开的文件等等

今天我会展示三个我使用最多也最为推荐的方式。

LINT 配置

不同级别的配置:

你可以从多种级别对Lint进行配置:(click)
(click)全局 (也就是整个项目)
(click)对于某一个项目模块
(click)或者是生产模块
(click)测试模块(click)
甚至是针对现在打开的文件等等

今天我会展示三个我使用最多也最为推荐的方式。

LINT 配置

不同级别的配置:

– 全局

你可以从多种级别对Lint进行配置:(click)
(click)全局 (也就是整个项目)
(click)对于某一个项目模块
(click)或者是生产模块
(click)测试模块(click)
甚至是针对现在打开的文件等等

今天我会展示三个我使用最多也最为推荐的方式。

LINT 配置

不同级别的配置:

– 全局

– 项目模块

你可以从多种级别对Lint进行配置:(click)
(click)全局 (也就是整个项目)
(click)对于某一个项目模块
(click)或者是生产模块
(click)测试模块(click)
甚至是针对现在打开的文件等等

今天我会展示三个我使用最多也最为推荐的方式。

LINT 配置

不同级别的配置:

- 全局
- 项目模块
- 生产模块

你可以从多种级别对Lint进行配置:(click)
(click)全局 (也就是整个项目)
(click)对于某一个项目模块
(click)或者是生产模块
(click)测试模块(click)
甚至是针对现在打开的文件等等

今天我会展示三个我使用最多也最为推荐的方式。

LINT 配置

不同级别的配置:

- 全局
- 项目模块
- 生产模块
- 测试模块

...

你可以从多种级别对Lint进行配置:(click)
(click)全局 (也就是整个项目)
(click)对于某一个项目模块
(click)或者是生产模块
(click)测试模块(click)
甚至是针对现在打开的文件等等

今天我会展示三个我使用最多也最为推荐的方式。

LINT 配置

不同级别的配置:

- 全局
- 项目模块
- 生产模块
- 测试模块

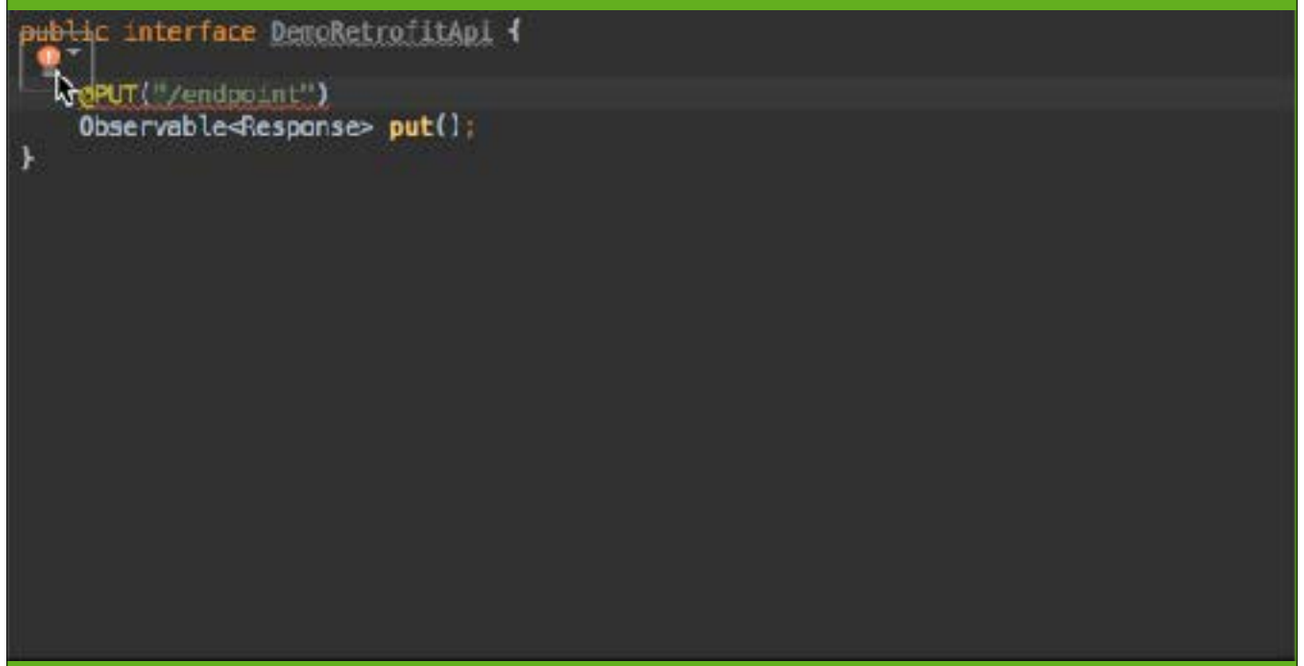
...

官方指导

你可以从多种级别对Lint进行配置:(click)
(click)全局 (也就是整个项目)
(click)对于某一个项目模块
(click)或者是生产模块
(click)测试模块(click)
甚至是针对现在打开的文件等等

今天我会展示三个我使用最多也最为推荐的方式。

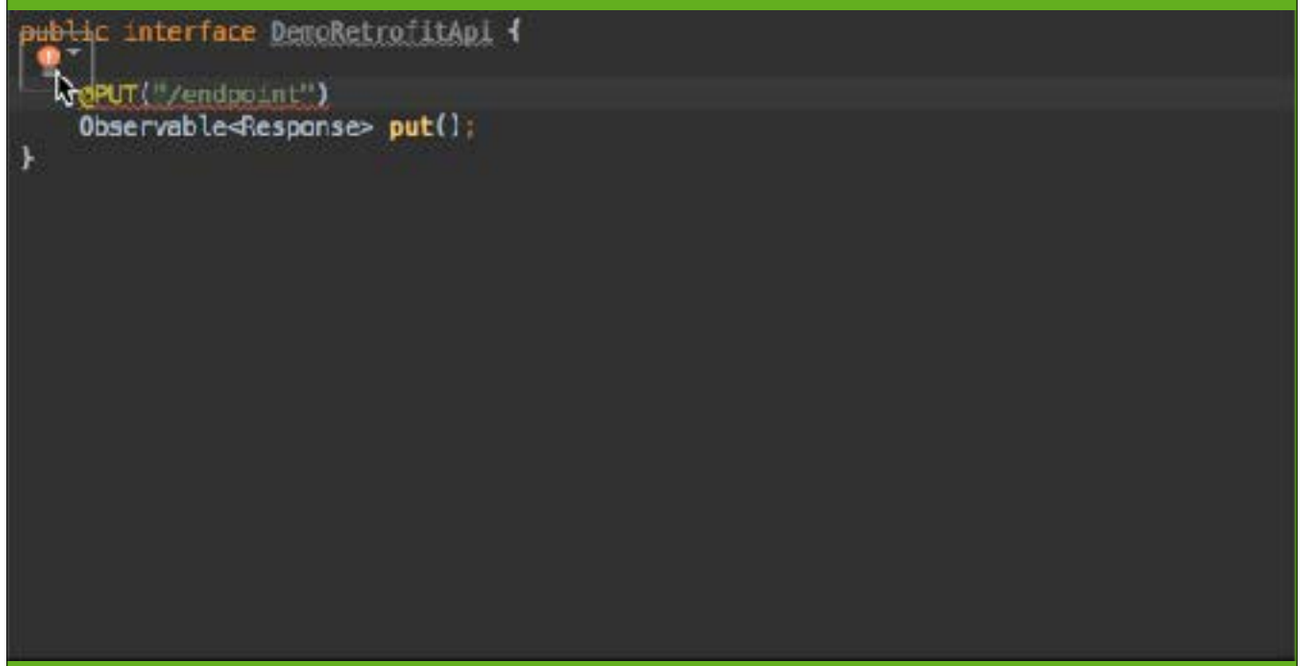
@SUPPRESSLINT



第一个是在源代码中的某一处禁用某一条规则，这个要用到一个注释，@SuppressLint。回到我们的RetroFit规则，如果决定这是一个特殊情况，我确实需要一个以斜线开始的端点，那我可以选择suppress它。这样我可以继续保留我的斜线，但同时也不会在运行时出错。无独有偶，这次我也不过需要点一点鼠标。(play the video)

这个注释使用方便，很适合对于某条规则的特殊处理。但是，很多时候我们可能需要对某一条规则进行更为整体化的处理，那就需要用到我最喜欢的也是使用最多的方法，lint.xml.

@SUPPRESSLINT



第一个是在源代码中的某一处禁用某一条规则，这个要用到一个注释，@SuppressLint。回到我们的RetroFit规则，如果决定这是一个特殊情况，我确实需要一个以斜线开始的端点，那我可以选择suppress它。这样我可以继续保留我的斜线，但同时也不会在运行时出错。无独有偶，这次我也不过需要点一点鼠标。(play the video)

这个注释使用方便，很适合对于某条规则的特殊处理。但是，很多时候我们可能需要对某一条规则进行更为整体化的处理，那就需要用到我最喜欢的也是使用最多的方法，lint.xml.

LINT.XML


```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--use 'lint - -list' to show all possible ids-->

<lint>
  <issue id="SpUsage" severity="ignore"/>
  <issue id="PilgrimSdkWith" severity="ignore"/>
  <issue id="Typos" severity="ignore"/>
  <issue id="MenuTitle" severity="ignore"/>

  <issue id="UnusedResources" severity="warning">
    <ignore path="src/main/res/values-*/strings.xml" />
    <ignore path="src/main/res/values/strings.xml" />
    <ignore path="src/main/res/xml/shortcuts.xml" />
  </issue>
```


这是 Groupon app的lint.xml节选，它在我们的root directory下面. 正如你所见(click)我可以选择通过降低优先等级完全忽略某一条规则。当然，反之我们也可以提高它的优先等级。

然后， (click)我也可以通过设定“ignore path=”来忽略某一个文件或者文件夹。同时这条规则在其他文件里还照样执行。

LINT.XML


```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--use 'lint - -list' to show all possible ids-->

<lint>
  <issue id="SpUsage" severity="ignore"/>
  <issue id="PilgrimSdkWith" severity="ignore"/>
  <issue id="Typos" severity="ignore"/>
  <issue id="MenuTitle" severity="ignore"/>

  <issue id="UnusedResources" severity="warning">
    <ignore path="src/main/res/values-*/strings.xml" />
    <ignore path="src/main/res/values/strings.xml" />
    <ignore path="src/main/res/xml/shortcuts.xml" />
  </issue>
```


这是 Groupon app的lint.xml节选，它在我们的root directory下面. 正如你所见(click)我可以选择通过降低优先等级完全忽略某一条规则。当然，反之我们也可以提高它的优先等级。

然后， (click)我也可以通过设定“ignore path=”来忽略某一个文件或者文件夹。同时这条规则在其他文件里还照样执行。

LINT.XML


```
<?xml version="1.0" encoding="UTF-8"?>
<!--use 'lint - -list' to show all possible ids-->

<lint>
  <issue id="SpUsage" severity="ignore"/>
  <issue id="PilgrimSdkWith" severity="ignore"/>
  <issue id="Typos" severity="ignore"/>
  <issue id="MenuTitle" severity="ignore"/>

  <issue id="UnusedResources" severity="warning">
    <ignore path="src/main/res/values-*/strings.xml" />
    <ignore path="src/main/res/values/strings.xml" />
    <ignore path="src/main/res/xml/shortcuts.xml" />
  </issue>
```


这是 Groupon app的lint.xml节选，它在我们的root directory下面. 正如你所见(click)我可以选择通过降低优先等级完全忽略某一条规则。当然，反之我们也可以提高它的优先等级。

然后， (click)我也可以通过设定“ignore path=”来忽略某一个文件或者文件夹。同时这条规则在其他文件里还照样执行。

LINTOPTIONS


```
lintOptions {
    // DO NOT TURN THIS OFF!
    // If you need to skip a lint error either ignore it in lint.xml
    // or suppress the lint warning
    abortOnError true
    quiet false
    htmlReport true
    xmlReport true
    baseline file("${lint_baseline_file_directory}/lint-baseline.html")
    lintConfig file("${lint_config_file}")
    htmlOutput file("${lint_reports_directory}/lint-results.html")
    xmlOutput file("${lint_reports_directory}/lint-results.xml")
}
```


下面要介绍一个非常实用的配置，LintOptions baseline。因为如果你和我一样是在编写一个“历史悠久”的大型app，尤其是在Lint出现之前就有的app，你会发现，第一次运行lint，报告上会有成百上千的错误和警告。那么我们应该怎么办?

我既不想让新的问题继续增加，但也实在没有时间和精力一次性把所有现有问题修好。

这个时候就要感谢LintOptions baseline的存在。只需要运行一边Lint，然后在gradle 文件里设置好这个现有的报告内容，之后Lint就只会在出现这个报告之外的问题时警告。

而如果需要改变baseline的内容，只需要更新这个文件即可。

(look at audience and pause)

定制Lint规则可以和自带规则一样配置。那么让我们一起来定制我们的贴身保镖!

LINTOPTIONS


```
lintOptions {
    // DO NOT TURN THIS OFF!
    // If you need to skip a lint error either ignore it in lint.xml
    // or suppress the lint warning
    abortOnError true
    quiet false
    htmlReport true
    xmlReport true
    baseline file("${lint_baseline_file_directory}/lint-baseline.html")
    lintConfig file("${lint_config_file}")
    htmlOutput file("${lint_reports_directory}/lint-results.html")
    xmlOutput file("${lint_reports_directory}/lint-results.xml")
}
```


下面要介绍一个非常实用的配置，LintOptions baseline。因为如果你和我一样是在编写一个“历史悠久”的大型app，尤其是在Lint出现之前就有的app，你会发现，第一次运行lint，报告上会有成百上千的错误和警告。那么我们应该怎么办?

我既不想让新的问题继续增加，但也实在没有时间和精力一次性把所有现有问题修好。

这个时候就要感谢LintOptions baseline的存在。只需要运行一边Lint，然后在gradle 文件里设置好这个现有的报告内容，之后Lint就只会在出现这个报告之外的问题时警告。

而如果需要改变baseline的内容，只需要更新这个文件即可。

(look at audience and pause)

定制Lint规则可以和自带规则一样配置。那么让我们一起来定制我们的贴身保镖!

定制 LINT

今天的灵感来自于Java的四大名著之首的Effective Java 第二条：遇到多个构造参数时要考虑用构建器。我们会编写一个 (click)JavaConstructorDetector 来对参数过多的构造器建议(click)使用builder 模式.

顺便说一下，Effective Java不仅是Java开发者的好朋友，也是一个寻找定制Lint规则灵感的绝佳起点。

定制 LINT

Effective Java 第2条:

遇到多个构造参数时要考虑用构建器

今天的灵感来自于Java的四大名著之首的Effective Java 第二条：遇到多个构造参数时要考虑用构建器。我们会编写一个 (click)JavaConstructorDetector 来对参数过多的构造器建议(click)使用builder 模式.

顺便说一下，Effective Java不仅是Java开发者的好朋友，也是一个寻找定制Lint规则灵感的绝佳起点。

构想: JavaConstructorDetector.java

Effective Java 第2条:

遇到多个构造参数时要考虑用构建器

今天的灵感来自于Java的四大名著之首的Effective Java 第二条：遇到多个构造参数时要考虑用构建器。我们会编写一个 (click)JavaConstructorDetector 来对参数过多的构造器建议(click)使用builder 模式.

顺便说一下，Effective Java不仅是Java开发者的好朋友，也是一个寻找定制Lint规则灵感的绝佳起点。

定制 LINT

为了让今天的演示集中在如何编写Lint而不是复杂的逻辑，我们需要如下假设(click)：
我们假设 (click)每个类只有一个构造器(所以我们只需要考虑一个构造器里的构造参数, 但是 /effective java里有提到，builder 模式在由多个构造器形成重叠构造器的情况下尤为推荐，所以敬请期待我将完整逻辑开源).
(click)接下来，我们假设只有可选域(所以不需要考虑必要参数的build函式).
(click)然后，每个构造参数和相应的域命名相同(就不需要检查它们之间的一一对应关系).
(click)最后，我们假设构造器只负责域的赋值.

那么，来看一个简单的代码例子!

定制 LINT

假设:

为了让今天的演示集中在如何编写Lint而不是复杂的逻辑，我们需要如下假设(click)：
我们假设 (click)每个类只有一个构造器(所以我们只需要考虑一个构造器里的构造参数, 但是 /effective java里有提到，builder 模式在由多个构造器形成重叠构造器的情况下尤为推荐，所以敬请期待我将完整逻辑开源).
(click)接下来，我们假设只有可选域(所以不需要考虑必要参数的build函式).
(click)然后，每个构造参数和相应的域命名相同(就不需要检查它们之间的一一对应关系).
(click)最后，我们假设构造器只负责域的赋值.

那么，来看一个简单的代码例子!

定制 LINT

假设:

- 每个类只有一个构造器

为了让今天的演示集中在如何编写Lint而不是复杂的逻辑，我们需要如下假设(click)：
我们假设 (click)每个类只有一个构造器(所以我们只需要考虑一个构造器里的构造参数, 但是 /effective java里有提到，builder 模式在由多个构造器形成重叠构造器的情况下尤为推荐，所以敬请期待我将完整逻辑开源).
(click)接下来，我们假设只有可选域(所以不需要考虑必要参数的build函式).
(click)然后，每个构造参数和相应的域命名相同(就不需要检查它们之间的一一对应关系).
(click)最后，我们假设构造器只负责域的赋值.

那么，来看一个简单的代码例子!

定制 LINT

假设:

- 每个类只有一个构造器
- 只有可选域

为了让今天的演示集中在如何编写Lint而不是复杂的逻辑，我们需要如下假设(click)：
我们假设 (click)每个类只有一个构造器(所以我们只需要考虑一个构造器里的构造参数, 但是 /effective java里有提到，builder 模式在由多个构造器形成重叠构造器的情况下尤为推荐，所以敬请期待我将完整逻辑开源).
(click)接下来，我们假设只有可选域(所以不需要考虑必要参数的build函式).
(click)然后，每个构造参数和相应的域命名相同(就不需要检查它们之间的一一对应关系).
(click)最后，我们假设构造器只负责域的赋值.

那么，来看一个简单的代码例子!

定制 LINT

假设:

- 每个类只有一个构造器
- 只有可选域
- 每个构造参数和相应的域命名相同

为了让今天的演示集中在如何编写Lint而不是复杂的逻辑，我们需要如下假设(click)：
我们假设 (click)每个类只有一个构造器(所以我们只需要考虑一个构造器里的构造参数, 但是 /effective java里有提到，builder 模式在由多个构造器形成重叠构造器的情况下尤为推荐，所以敬请期待我将完整逻辑开源).
(click)接下来，我们假设只有可选域(所以不需要考虑必要参数的build函式).
(click)然后，每个构造参数和相应的域命名相同(就不需要检查它们之间的一一对应关系).
(click)最后，我们假设构造器只负责域的赋值.

那么，来看一个简单的代码例子!

定制 LINT

假设:

- 每个类只有一个构造器
- 只有可选域
- 每个构造参数和相应的域命名相同
- 构造器内只负责域的赋值

为了让今天的演示集中在如何编写Lint而不是复杂的逻辑，我们需要如下假设(click)：
我们假设 (click)每个类只有一个构造器(所以我们只需要考虑一个构造器里的构造参数, 但是 /effective java里有提到，builder 模式在由多个构造器形成重叠构造器的情况下尤为推荐，所以敬请期待我将完整逻辑开源).
(click)接下来，我们假设只有可选域(所以不需要考虑必要参数的build函式).
(click)然后，每个构造参数和相应的域命名相同(就不需要检查它们之间的一一对应关系).
(click)最后，我们假设构造器只负责域的赋值.

那么，来看一个简单的代码例子!

```java
public class Foo {
    private int a;
    private boolean b;
    private String c;

    public Foo(int a, boolean b, String c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
}
```


大家来看Foo的构造器. (click)竟然有三个构造参数!

```java
public class Foo {
    private int a;
    private boolean b;
    private String c;

    public Foo(int a, boolean b, String c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
}
```


大家来看Foo的构造器. (click)竟然有三个构造参数!

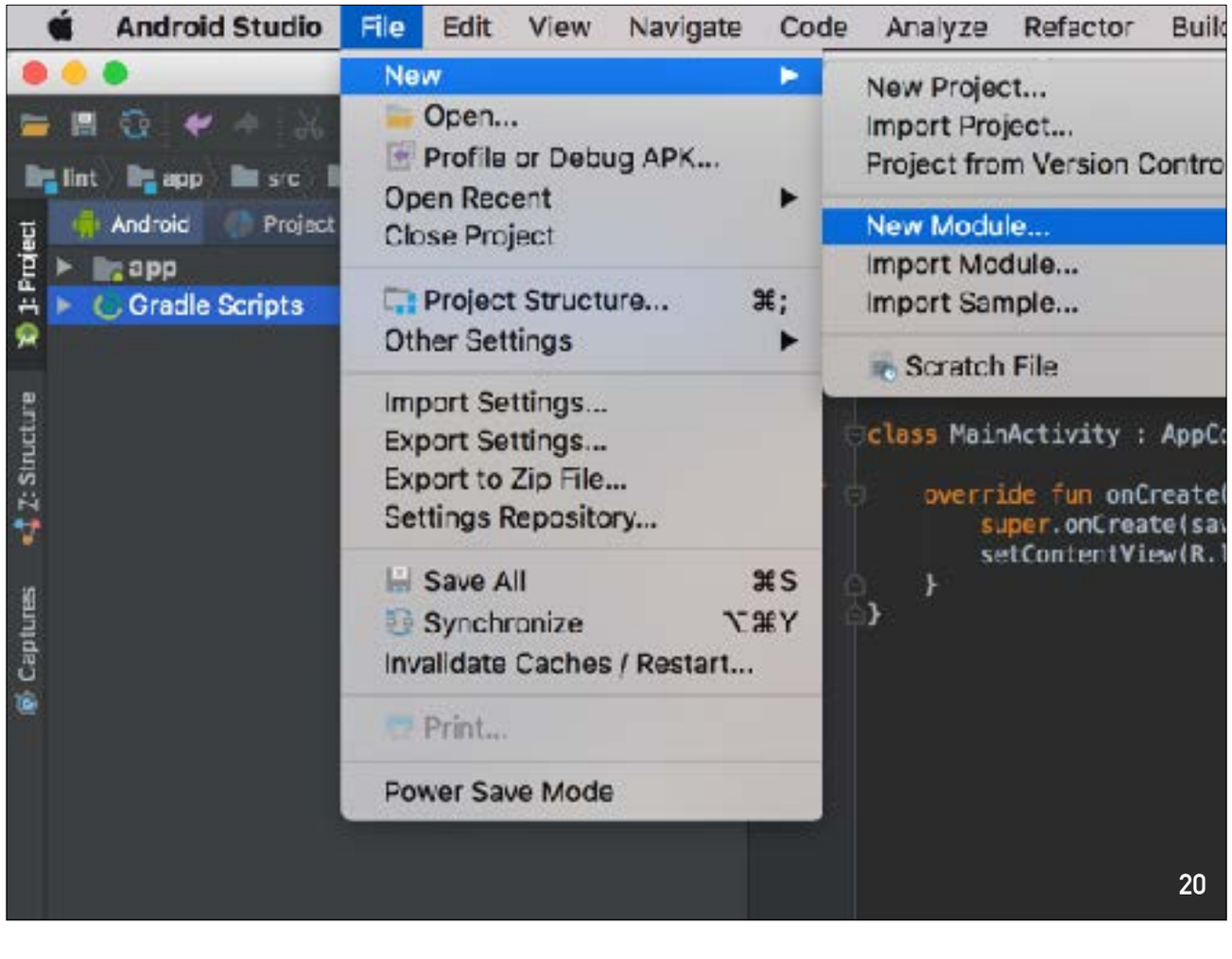
```java
public class Foo {
    …
    public Foo(int a, boolean b, String c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
}
```


当然了，我们不会允许这种这种情况的发生。

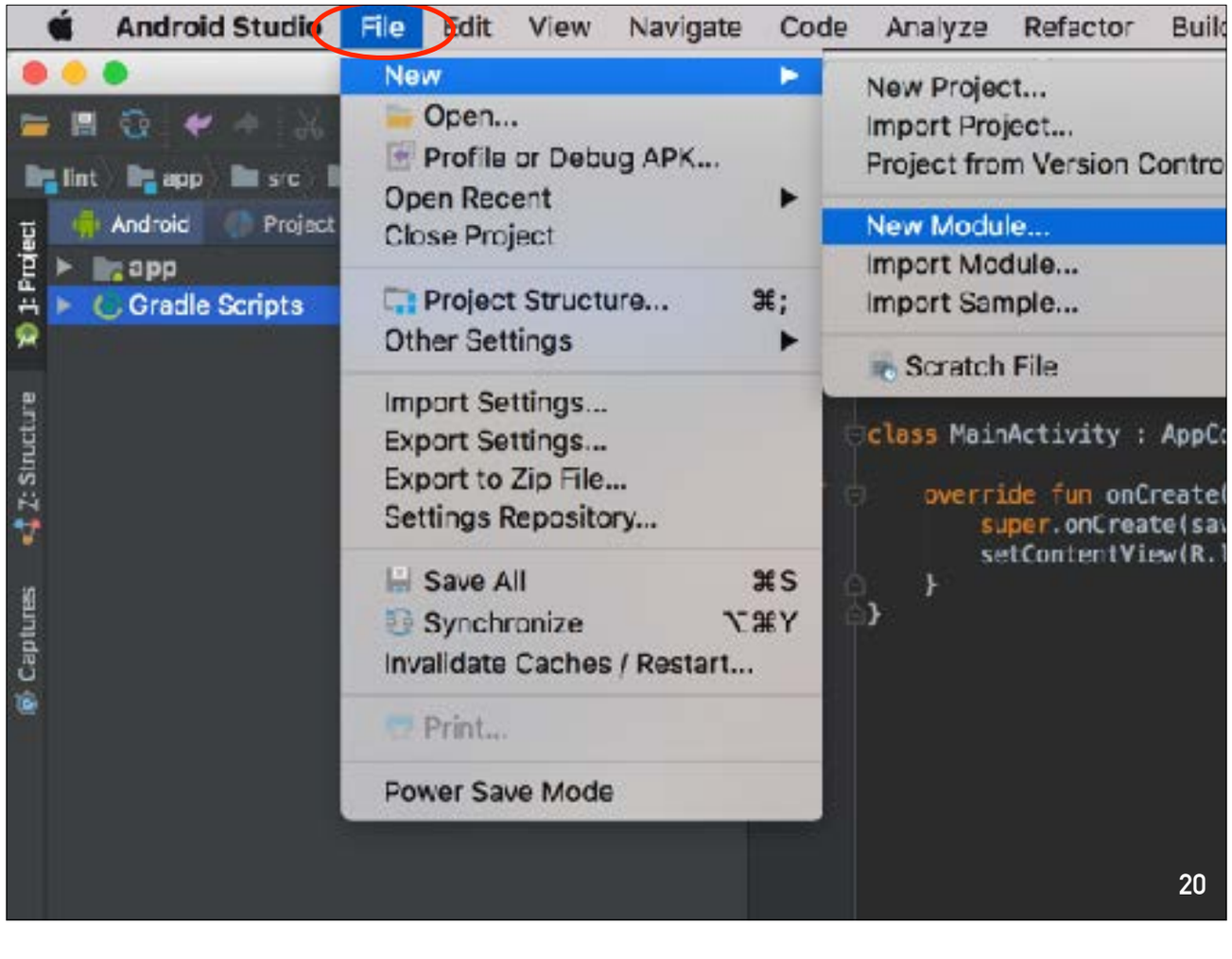
```java
public class Foo {
   …

    private Foo(Builder builder) {
      this.a = builder.a;
      this.b = builder.b;
      this.c = builder.c;
   }
   public static class Builder {
      private int a;
      private boolean b;
      private String c;
      public Builder() {
      }
      public Builder a(int a) {
         this.a = a;
         return this;
      }
      public Builder b(boolean b) {
         this.b = b;
         return this;
      }
      public Builder c(String c) {
         this.c = c;
         return this;
      }
      public Foo build() {
         return new Foo(this);
      }
   }
}
```


我们需要将这个构造器改成builder模式。

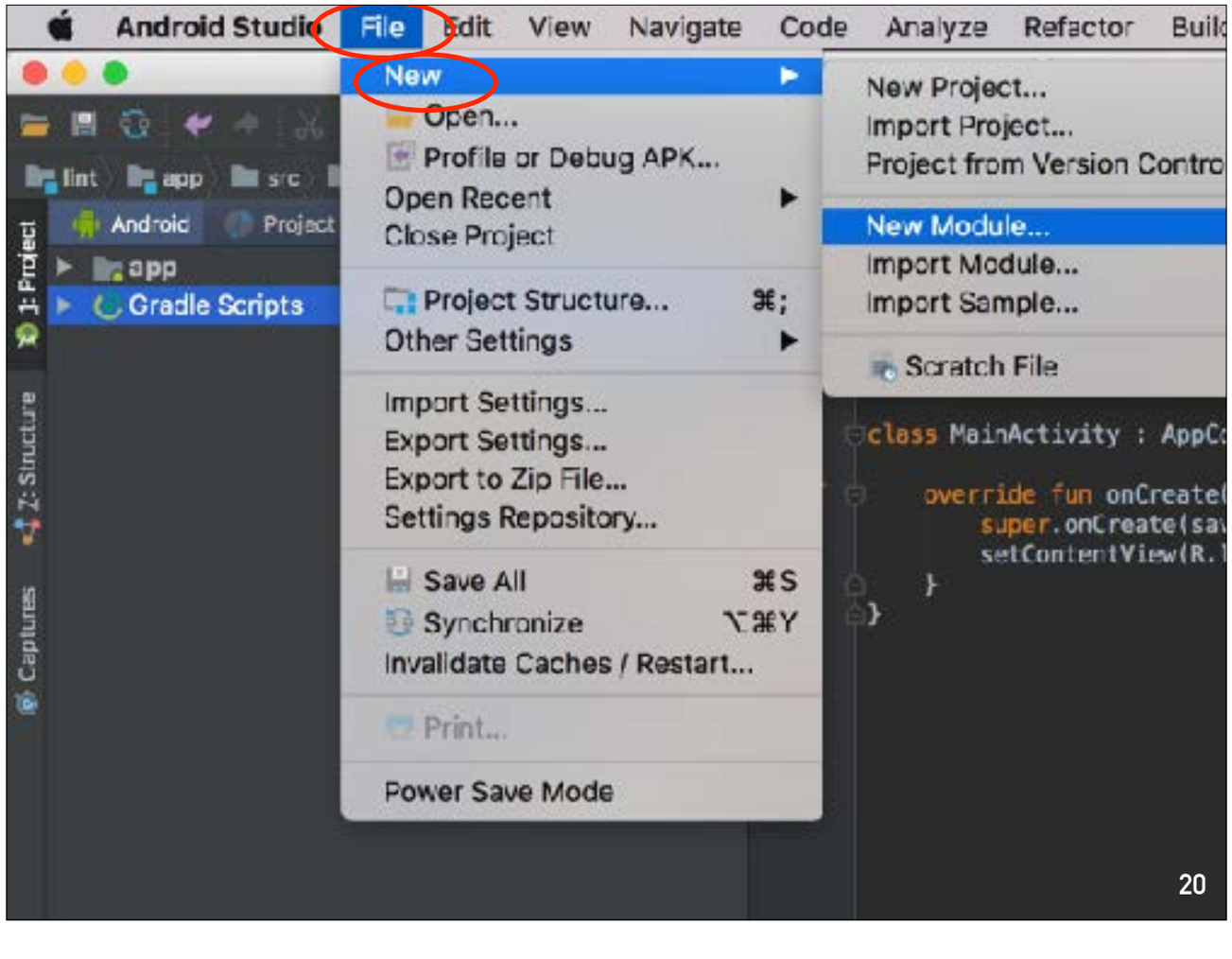
简单明了，是不是? 那么是时候进入Android Studio了.



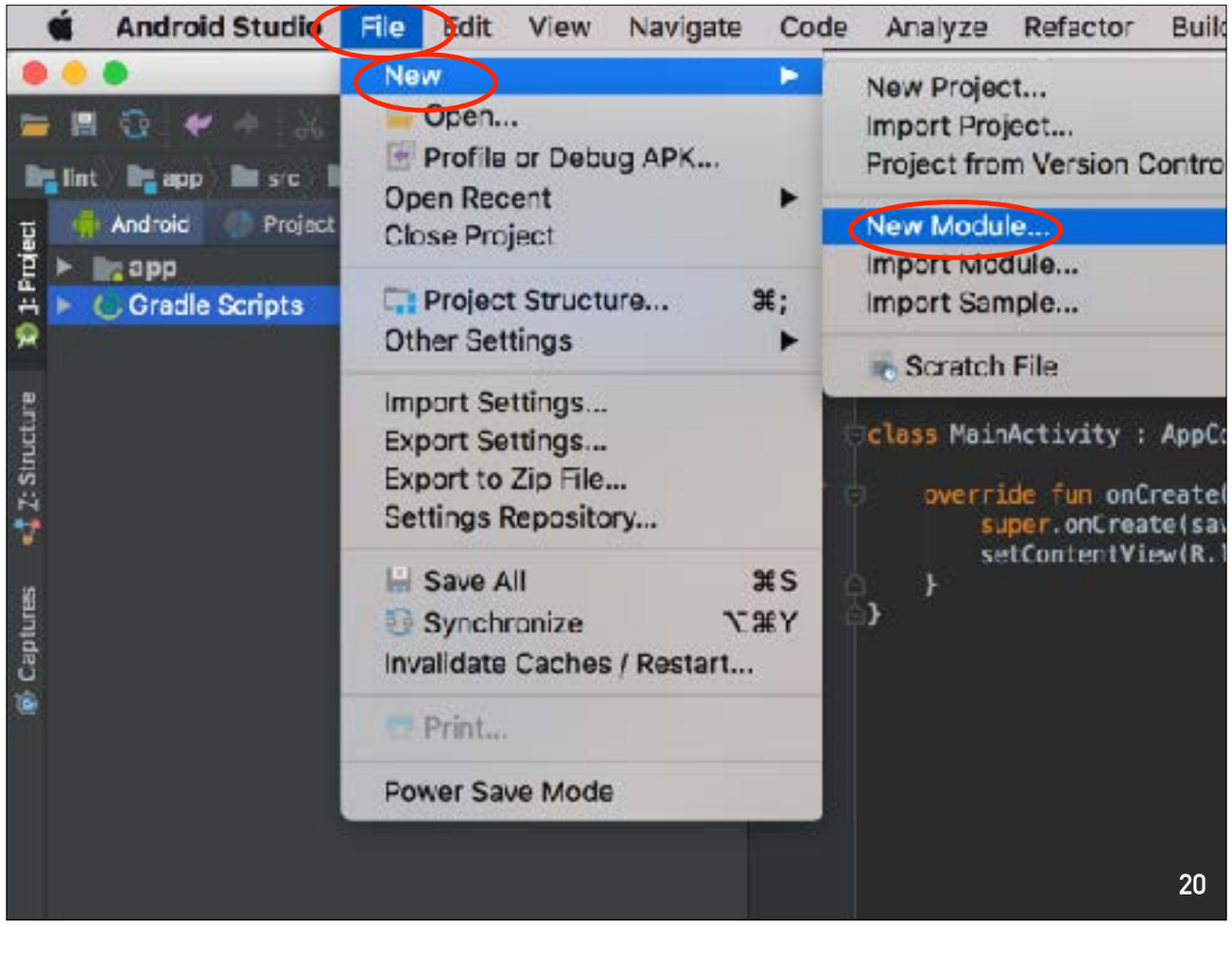
只需要轻击鼠标, 我们就能为定制Lint的函式库设置好模块. (click)File- (click)New- (click)New Module.(next slide)



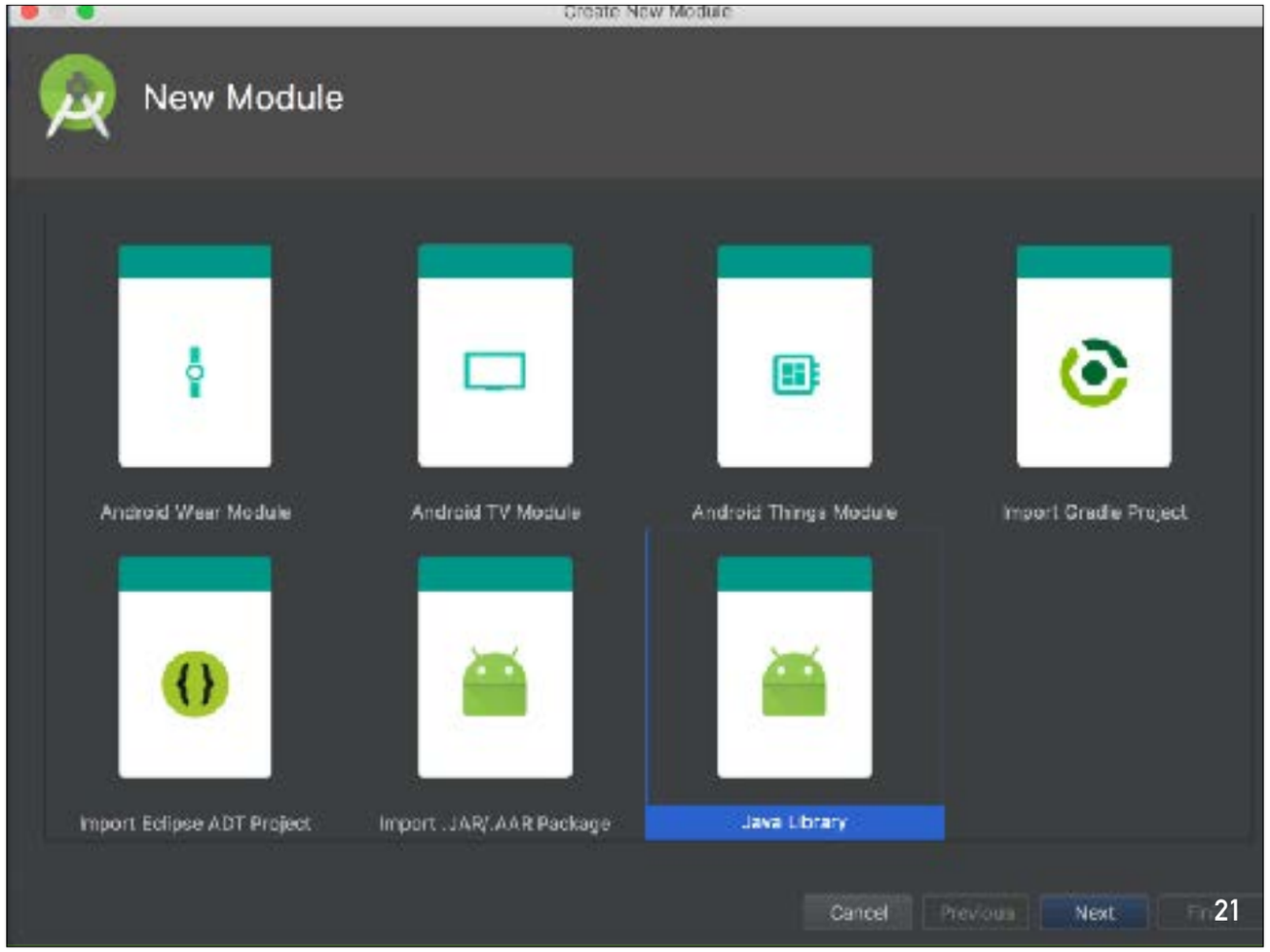
只需要轻击鼠标, 我们就能为定制Lint的函式库设置好模块. (click)File- (click)New- (click)New Module.(next slide)



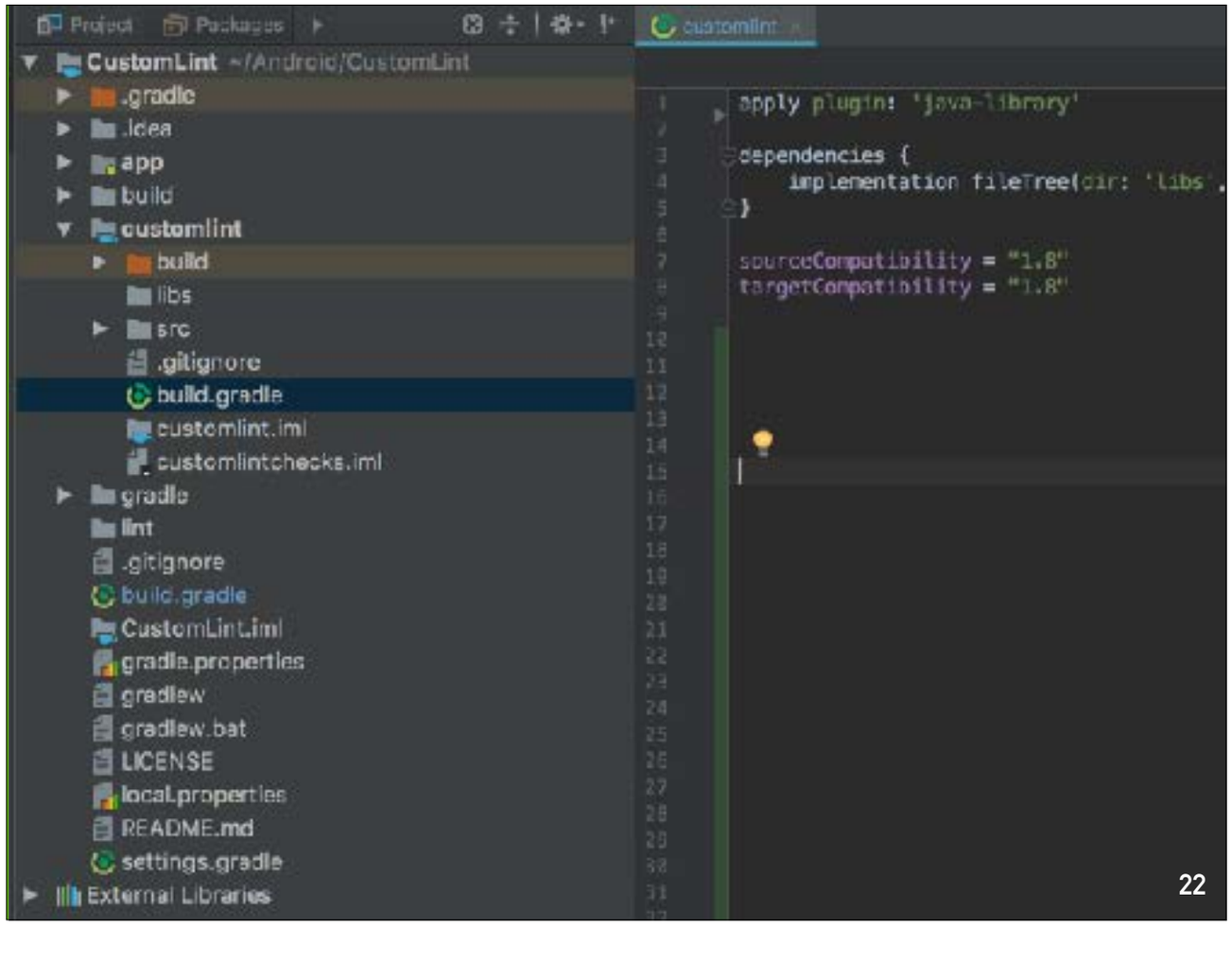
只需要轻击鼠标, 我们就能为定制Lint的函式库设置好模块. (click)File- (click)New- (click)New Module.(next slide)



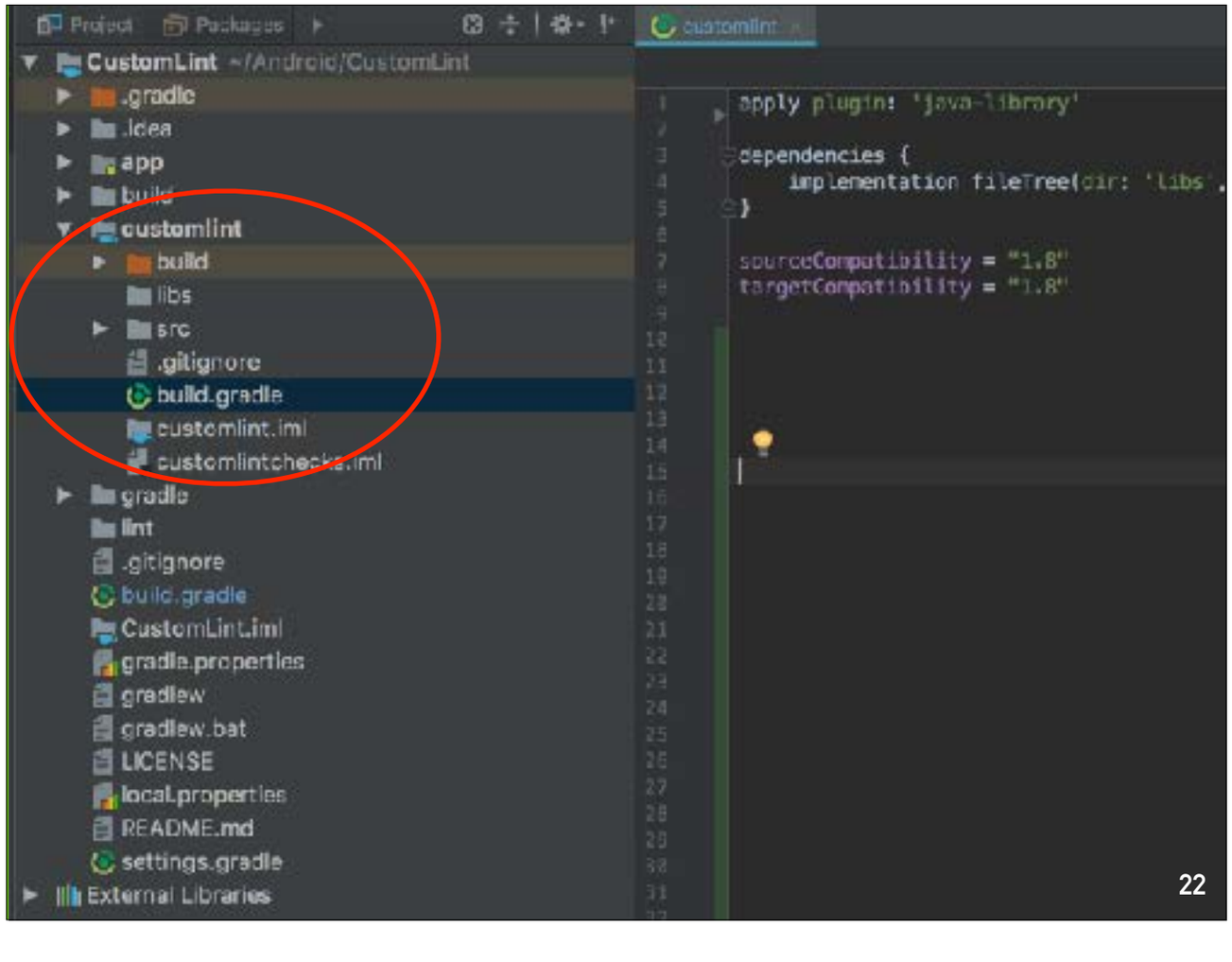
只需要轻击鼠标, 我们就能为定制Lint的函式库设置好模块. (click)File- (click)New- (click)New Module.(next slide)



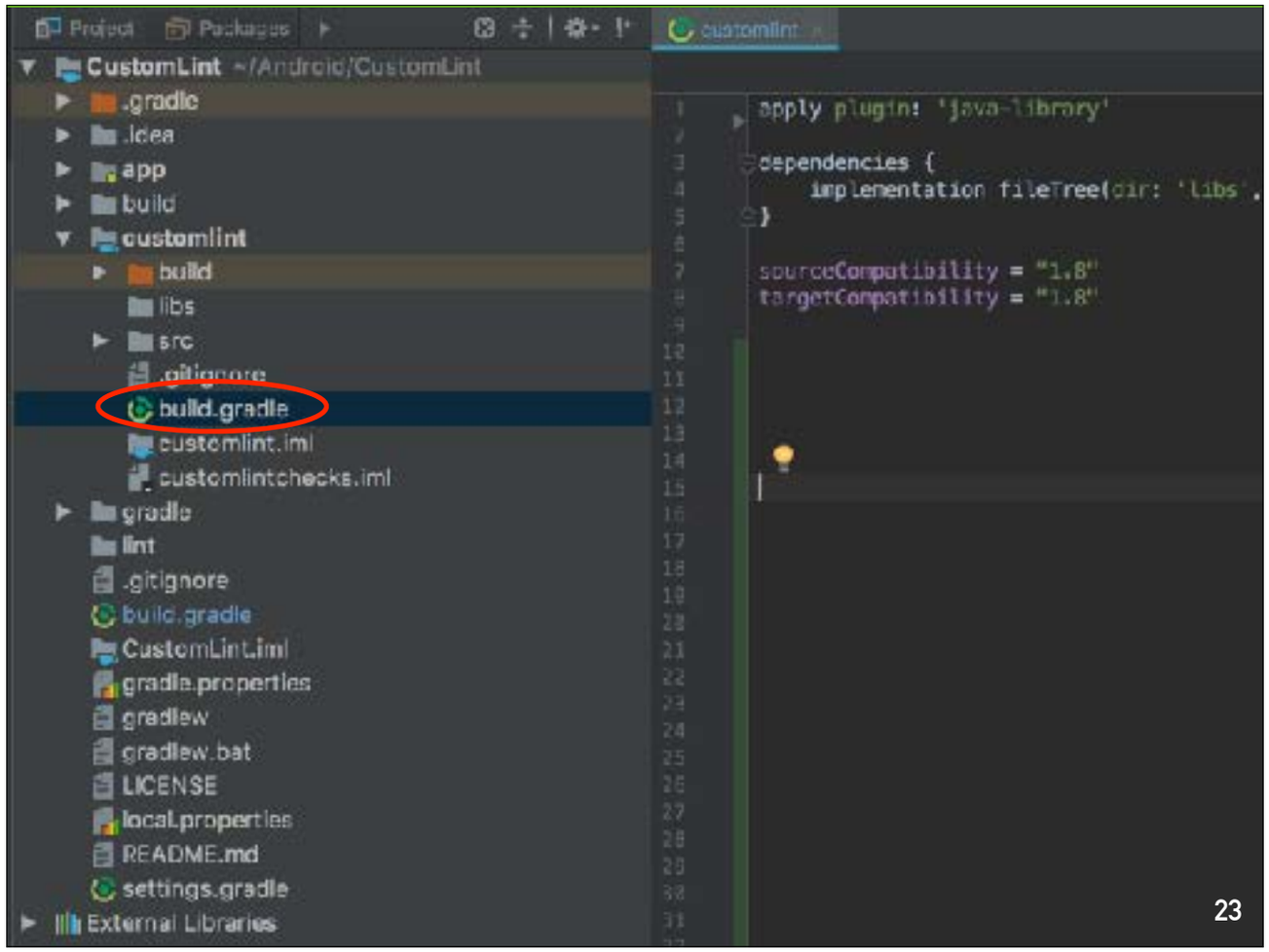
然后记得选 java library. 命名. 你就会在自己的项目中看到这个模块.



(click)就像这样. 下一步就是将 lint api dependencies加入这个模块的gradle.build



(click)就像这样. 下一步就是将 lint api dependencies加入这个模块的gradle.build


Project
Packages
CustomLint ~/Android/CustomLint
.gradle
.idea
app
build
customlint
build
libs
src
.gitignore
build.gradle
customlint.iml
customlintchecks.iml
gradle
lint
.gitignore
build.gradle
CustomLint.iml
gradle.properties
gradlew
gradlew.bat
LICENSE
local.properties
README.md
settings.gradle
External Libraries
customlint
apply plugin: 'java-library'
dependencies {
implementation fileTree(dir: 'libs',
}
sourceCompatibility = "1.8"
targetCompatibility = "1.8"

CustomLint/customlint/build.gradle

```
apply plugin: 'java-library'

…

}
```


Android Studio应该已经有在gradle.build里面加入一些内容。我们只需要在这基础之上添加.

CustomLint/customlint/build.gradle

```
apply plugin: 'java-library'

…

dependencies {
    api 'com.android.tools.lint:lint-api:26.0.0-rc1'
    api 'com.android.tools.lint:lint-checks:26.0.0-rc1'
    testImplementation 'junit:junit:4.12'
    testImplementation 'com.android.tools.lint:lint:26.0.0-rc1'
    testImplementation 'com.android.tools.lint:lint-tests:26.0.0-rc1'
    testImplementation 'com.android.tools:testutils:26.0.0-rc1'
}
```


这里我们说需要 lint api dependencies, 一定不要忘记test dependencies.

那么在编写Java代码之前，让我们不得不提一下lint规则的组成.

LINT 规则的组成

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

LINT 规则的组成

– DETECTOR: LINT 规则的主要逻辑

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

LINT 规则的组成

– DETECTOR: LINT 规则的主要逻辑

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

LINT 规则的组成

– DETECTOR: LINT 规则的主要逻辑

Issue: 设立所检测的问题

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

LINT 规则的组成

– DETECTOR: LINT 规则的主要逻辑

Issue: 设立所检测的问题

Scanner: check 和 fix的逻辑

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

LINT 规则的组成

– DETECTOR: LINT 规则的主要逻辑

Issue: 设立所检测的问题

Scanner: check 和 fix的逻辑

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

LINT 规则的组成

- DETECTOR: LINT 规则的主要逻辑
  - Issue: 设立所检测的问题
  - Scanner: check 和 fix的逻辑
- ISSUEREGISTRY: ISSUES 的 LIST

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

LINT 规则的组成

- DETECTOR: LINT 规则的主要逻辑
  - Issue: 设立所检测的问题
  - Scanner: check 和 fix的逻辑
- ISSUEREGISTRY: ISSUES 的 LIST
- 测试!

(click)Detector包含一个规则或多个规则的主要逻辑。没错，一个detector可以同时用来监测一个或多个规则。(example in the built in checks)
每个Detector内都会有issue，和scanner.
(click)每一个issue都分别是一个需要检索的问题，比如我们一开始看到的Duplicate ids，和retrofit 端点以斜线开始.
(click)Scanner 包含着detector的主要逻辑，它会定义如何发现并且报告这个issue，同时也可以提出改进建议.

(click)当detector写好以后，我们需要将它检测的issue / issues放进一个IssueRegistry，这样Lint才知道要检测它或者它们。IssueRegistry有点像AndroidManifest，只不过它包含的是所有lint issues 而不是Android activities。

最后，不要忘记测试！Lint的测试api非常方便实用，所以我们可以在将定制规则运用到实际项目之前先模拟实战.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner
…
}
```


那么，这里是我们的JavaConstructorDetector, 它需要 extend Detector 并且 implement UastScanner.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        …);

    …
}
```


现在我们来建立issue, 你可以看到其实Lint Issue的构造也是使用了类似builder的模式。第一个必要参数是id。 (click)Id会出现在 lint 报告和suppress lint中，所以尽量简明扼要，我就叫它TooManyParametersConstructor。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        …);

    …
}
```

现在我们来建立issue, 你可以看到其实Lint Issue的构造也是使用了类似builder的模式。第一个必要参数是id。 (click)Id会出现在 lint 报告和suppress lint中，所以尽量简明扼要，我就叫它TooManyParametersConstructor。

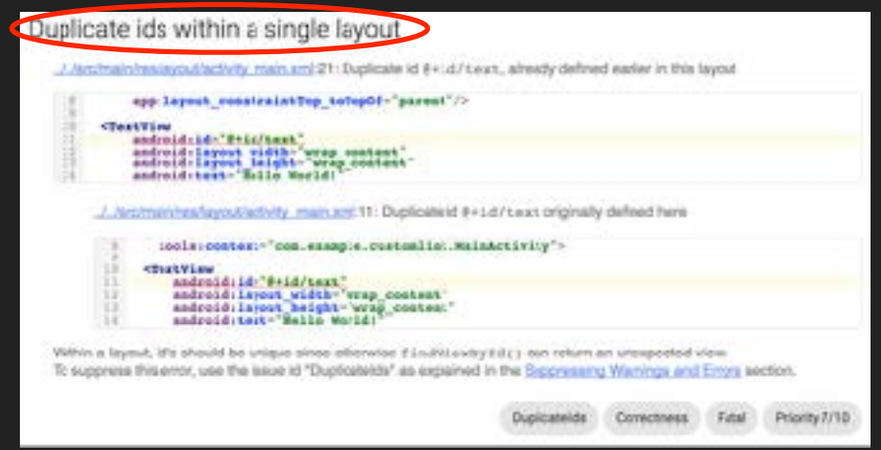
.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        …);
    …
}
```


下一个是berif description，简短的问题描述—也会出现在 lint 报告和IDE的错误提示中，就是我们一开始在Retrofit视频中看到的小灯泡的那个内容框里.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        …);

    …
}
```



下一个是berif description，简短的问题描述—也会出现在 lint 报告和IDE的错误提示中，就是我们一开始在Retrofit视频中看到的小灯泡的那个内容框里.

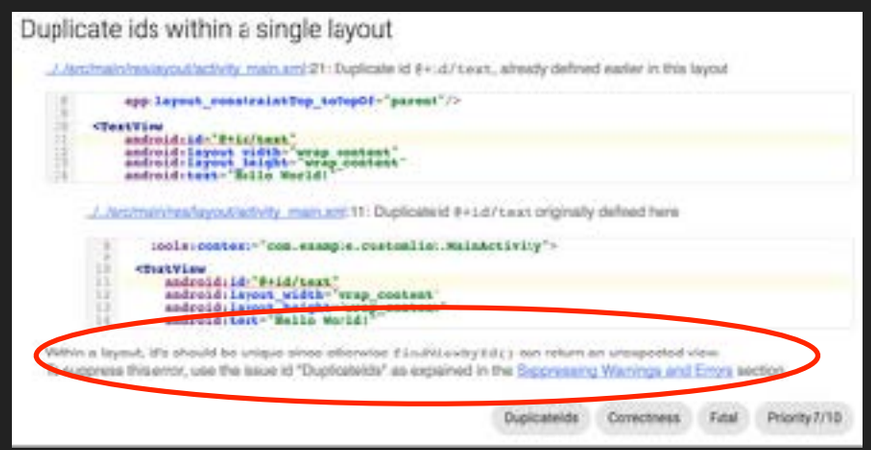
.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.”,
        …);
   …
}
```


然后是full description，详细的问题描述, 我们在一开始的Lint报告视频中有看到这个出现在点击explain之后。根据你的对象，这个可以不止一行，也可以设置粗体，斜体等等。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        …);
   …
}
```



然后是full description，详细的问题描述, 我们在一开始的Lint报告视频中有看到这个出现在点击explain之后。根据你的对象，这个可以不止一行，也可以设置粗体，斜体等等。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        Category.CORRECTNESS,
        …);

    …
}
```


Category，也就是问题的类别，Lint api包含很多已有的类别，performance, correctness, security, etc。当然你也可以建立自定义类别。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        Category.CORRECTNESS,
        ...);

    ...
}
```

Category，也就是问题的类别，Lint api包含很多已有的类别，performance, correctness, security, etc。当然你也可以建立自定义类别。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        Category.CORRECTNESS,
        6,
        …);
   …
}
```


Priority 或者优先级是一个1到10之间的数字，代表这个问题的重要性。这里设置为6是为了匹配severity.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        Category.CORRECTNESS,
        6,
        ...);
    ...
}
```

Priority 或者优先级是一个1到10之间的数字，代表这个问题的重要性。这里设置为6是为了匹配severity.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        Category.CORRECTNESS,
        6,
        Severity.WARNING,
        …);
    …
}
```


Severity 或者严重性决定了你的项目运行行为。这里我设定的是warning，所以它不会让运行失败。但是如果我设置成了retrofit规则那样的 fatal，代码就会无法运行。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        Category.CORRECTNESS,
        6,
        Severity.WARNING,
        ...);

    ...
}
```

Severity 或者严重性决定了你的项目运行行为。这里我设定的是warning，所以它不会让运行失败。但是如果我设置成了retrofit规则那样的 fatal，代码就会无法运行。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    public static final Issue TOO_MANY_PARAMETERS_ISSUE =
Issue.create(
        "TooManyParametersConstructor",
        "Constructor has too many parameters.",
        "Switching to builder pattern improves readability and
scalability.",
        Category.CORRECTNESS,
        6,
        Severity.WARNING,
        new Implementation(JavaConstructorDetector.class,
Scope.JAVA_FILE_SCOPE));

    …
}
```


最后一个是Implementation，定义了查找这个issue的范围。这个可以是 Java 文件，Kotlin 文件，Xml 文件，甚至gradle文件等等。值得一提的是，多个issues可以共用一个implementation，所以如果需要，记得建造一个域。

那么这里就是我们的issue，现在一起来看scanner.

AST and UAST

为了帮助大家更好地理解Lint的工作原理，我不得不提到两个重要的概念：AST 和 UAST.

作为移动开发者，我们多半不常听到这两个词。但是它们与我们每天每一次编译都息息相关。它们就是编译器看待我们代码的方式。(click)AST, 或者抽象语法树，是某一种语言源代码抽象语法结构的树状表现形式。数上的每一个节点都是一个逻辑元素。我们之所以叫它抽象就是因为一些细节会被抛开，比如括号，逗号，分号之类的。

这里有一个简单的AST例子: 代码是(click)b != 0.(Pause)
这个其实是一个大AST中的一个分支，完整的AST在这里: (next slide)

AST and UAST

AST -ABSTRACT SYNTAX TREE（抽象语法树）：

某一种语言源代码抽象语法结构的树状表现形式

为了帮助大家更好地理解Lint的工作原理，我不得不提到两个重要的概念：AST 和 UAST.

作为移动开发者，我们多半不常听到这两个词。但是它们与我们每天每一次编译都息息相关。它们就是编译器看待我们代码的方式。(click)AST, 或者抽象语法树，是某一种语言源代码抽象语法结构的树状表现形式。数上的每一个节点都是一个逻辑元素。我们之所以叫它抽象就是因为一些细节会被抛开，比如括号，逗号，分号之类的。

这里有一个简单的AST例子: 代码是(click)b != 0.(Pause)
这个其实是一个大AST中的一个分支，完整的AST在这里: (next slide)

定制 LINT

AST and UAST

AST -ABSTRACT SYNTAX TREE（抽象语法树）：

某一种语言源代码抽象语法结构的树状表现形式

例子: b != 0

为了帮助大家更好地理解Lint的工作原理，我不得不提到两个重要的概念：AST 和 UAST.

作为移动开发者，我们多半不常听到这两个词。但是它们与我们每天每一次编译都息息相关。它们就是编译器看待我们代码的方式。(click)AST, 或者抽象语法树，是某一种语言源代码抽象语法结构的树状表现形式。数上的每一个节点都是一个逻辑元素。我们之所以叫它抽象就是因为一些细节会被抛开，比如括号，逗号，分号之类的。

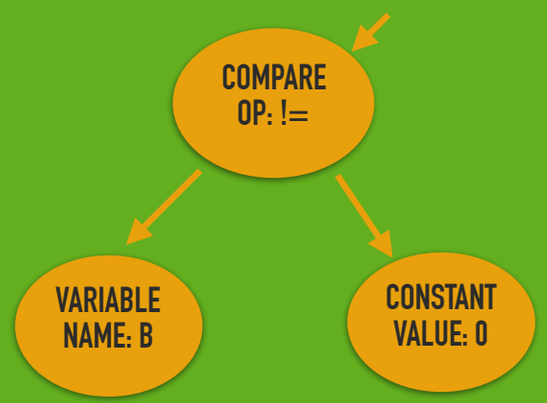
这里有一个简单的AST例子: 代码是(click)b != 0.(Pause)
这个其实是一个大AST中的一个分支，完整的AST在这里: (next slide)

AST and UAST

AST -ABSTRACT SYNTAX TREE（抽象语法树）：

某一种语言源代码抽象语法结构的树状表现形式

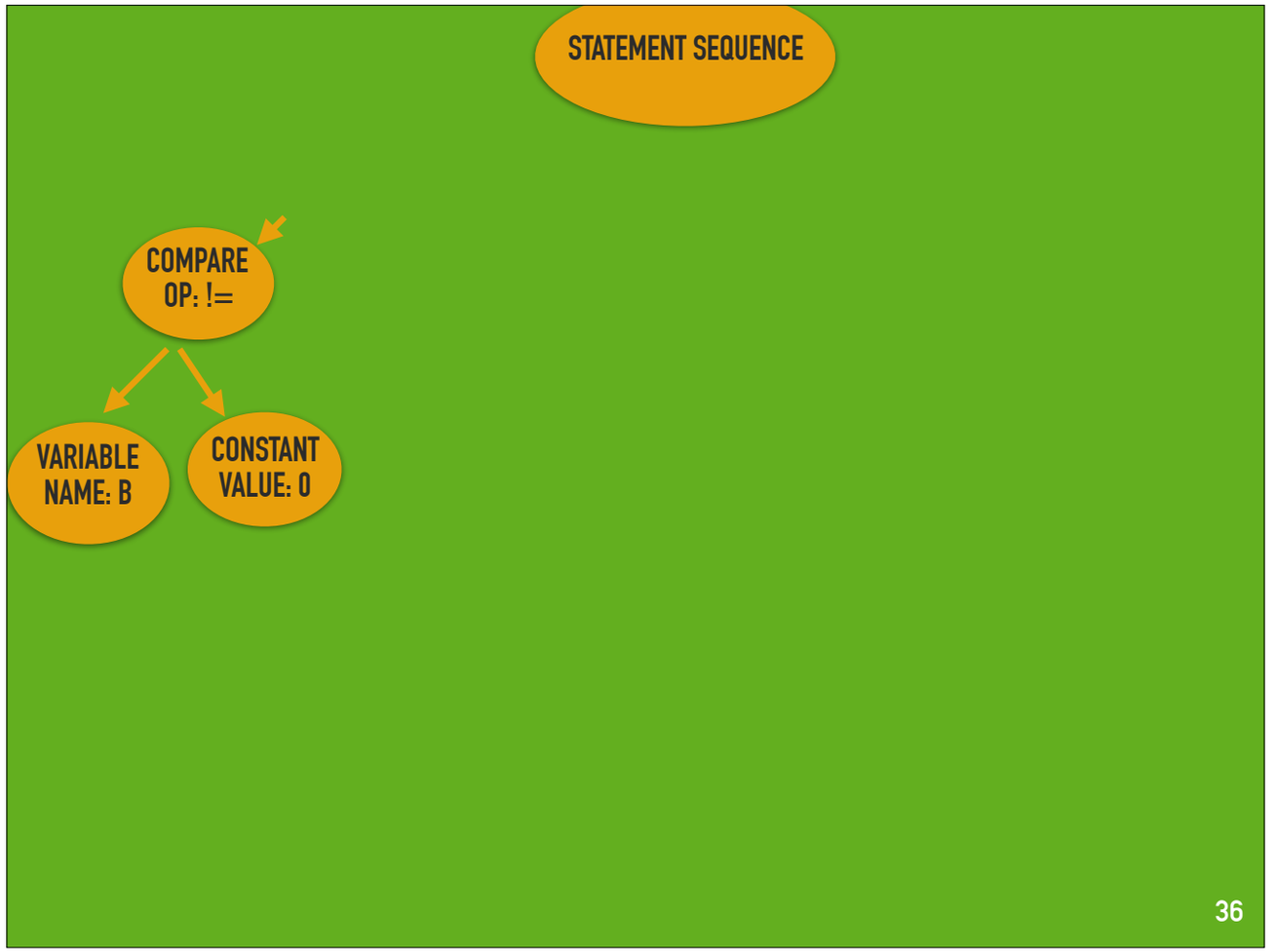
例子: b != 0



为了帮助大家更好地理解Lint的工作原理，我不得不提到两个重要的概念：AST 和 UAST.

作为移动开发者，我们多半不常听到这两个词。但是它们与我们每天每一次编译都息息相关。它们就是编译器看待我们代码的方式。(click)AST, 或者抽象语法树，是某一种语言源代码抽象语法结构的树状表现形式。数上的每一个节点都是一个逻辑元素。我们之所以叫它抽象就是因为一些细节会被抛开，比如括号，逗号，分号之类的。

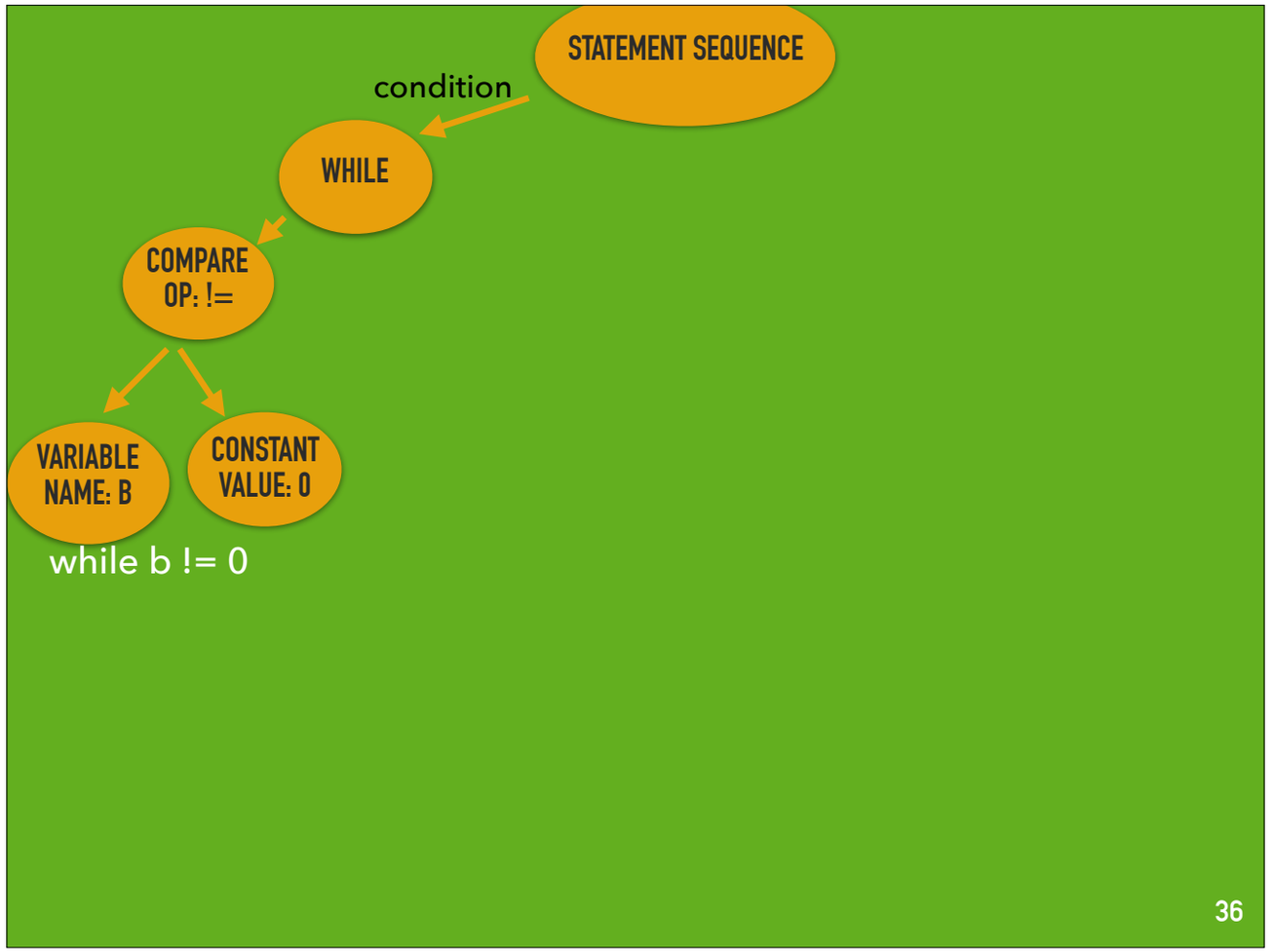
这里有一个简单的AST例子: 代码是(click)b != 0.(Pause)
这个其实是一个大AST中的一个分支，完整的AST在这里: (next slide)



那我们来做一个小练习，看看这个AST代表什么

所以我们有 (click) while b ≠ 0
然后我们来到while loop的body,
(click) 比较 a and b
(click) 如果a 比较大
(click) 赋值a 等于 a-b
(click) 否则赋值b 等于 b-a
(click) 如果我们走出了 loop, return a

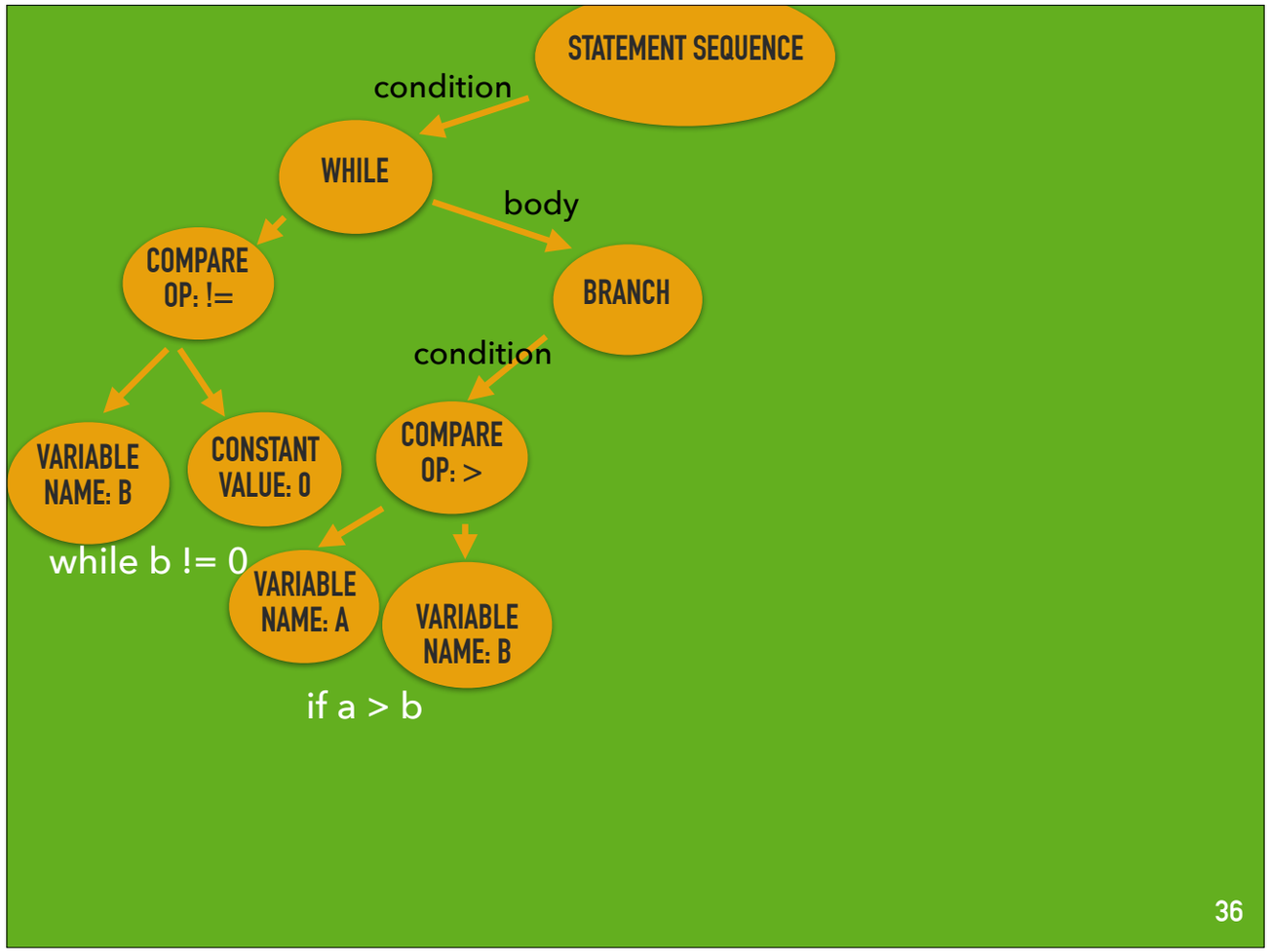
看起来很大的一棵树，其实不过6行代码



那我们来做一个小练习，看看这个AST代表什么

所以我们有 (click) while b ≠ 0
然后我们来到while loop的body,
(click) 比较 a and b
(click) 如果a 比较大
(click) 赋值a 等于 a-b
(click) 否则赋值b 等于 b-a
(click) 如果我们走出了 loop, return a

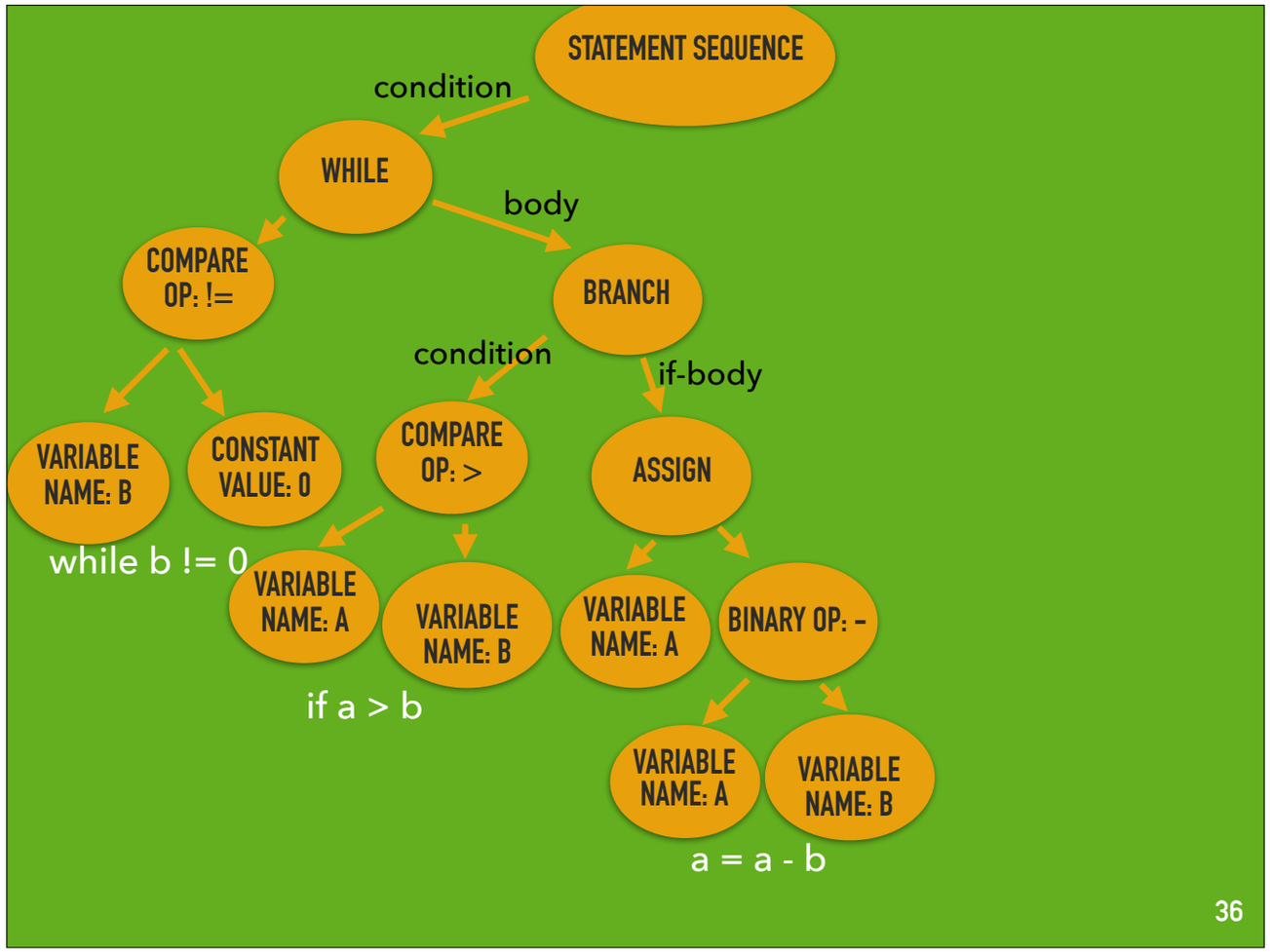
看起来很大的一棵树，其实不过6行代码



那我们来做一个小练习，看看这个AST代表什么

所以我们有 (click) while b ≠ 0
然后我们来到while loop的body,
(click) 比较 a and b
(click) 如果a 比较大
(click) 赋值a 等于 a-b
(click) 否则赋值b 等于 b-a
(click) 如果我们走出了 loop, return a

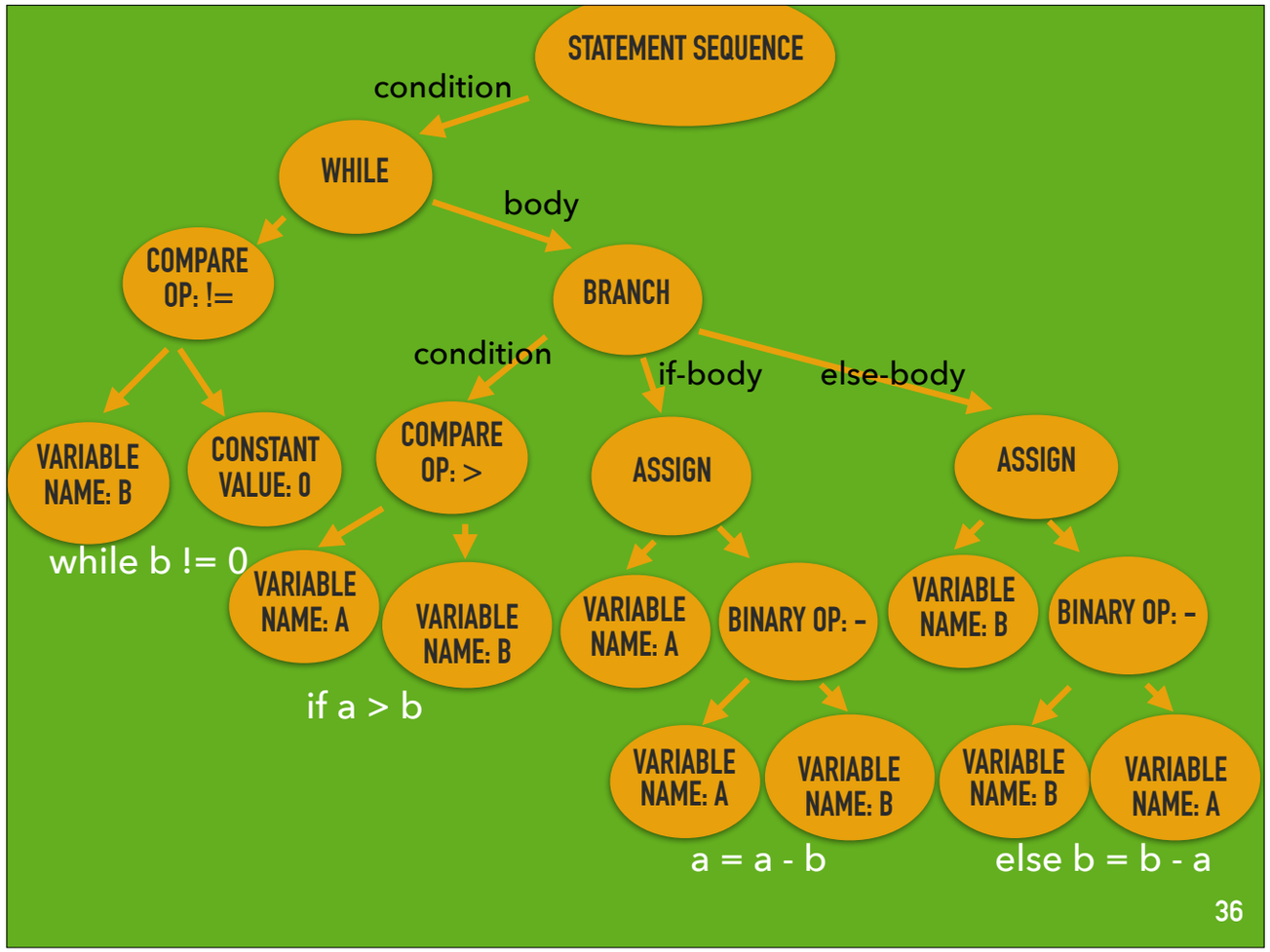
看起来很大的一棵树，其实不过6行代码



那我们来做一个小练习，看看这个AST代表什么

所以我们有 (click) while b ≠ 0
然后我们来到while loop的body,
(click) 比较 a and b
(click) 如果a 比较大
(click) 赋值a 等于 a-b
(click) 否则赋值b 等于 b-a
(click) 如果我们走出了 loop, return a

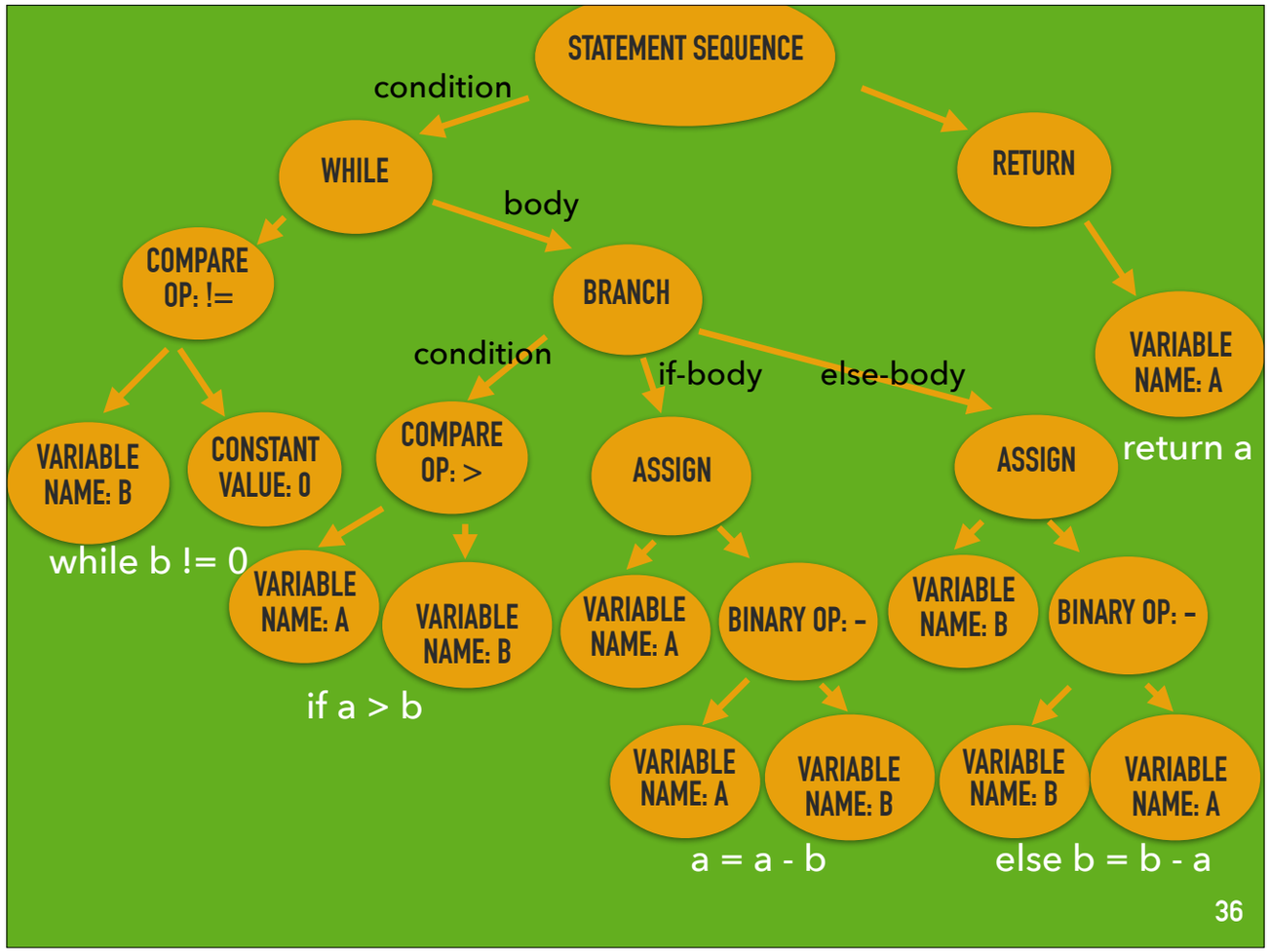
看起来很大的一棵树，其实不过6行代码



那我们来做一个小练习，看看这个AST代表什么

所以我们有 (click) while b ≠ 0
然后我们来到while loop的body,
(click) 比较 a and b
(click) 如果a 比较大
(click) 赋值a 等于 a-b
(click) 否则赋值b 等于 b-a
(click) 如果我们走出了 loop, return a

看起来很大的一棵树，其实不过6行代码



那我们来做一个小练习，看看这个AST代表什么

所以我们有 (click) while b ≠ 0
然后我们来到while loop的body,
(click) 比较 a and b
(click) 如果a 比较大
(click) 赋值a 等于 a-b
(click) 否则赋值b 等于 b-a
(click) 如果我们走出了 loop, return a

看起来很大的一棵树，其实不过6行代码

```
while b != 0
    if a > b
        a = a - b
    else
        b = b - a
return a
```


是不是很眼熟？有没有同学（举手）知道这个代码究竟在做什么？

没错这个就是著名的欧几里得算法，它给出了最大的公除数，也给出了一个很棒的AST例子。

```
while b != 0
    if a > b
        a = a - b
    else
        b = b - a
return a
```

欧几里得算法

是不是很眼熟？有没有同学（举手）知道这个代码究竟在做什么？

没错这个就是著名的欧几里得算法，它给出了最大的公除数，也给出了一个很棒的AST例子。

AST and UAST

那么明白了AST，Uast就很简单. Uast, 通用抽象语法树, 是某几种语言源代码抽象语法结构的树状表现形式。

UAST的存在源于很多计算机语言在语法结构上的相似性。比如刚才我们看到的欧几里得算法数就通用于java和c。对于我们今天的内容而言，因为kotlin语言是由一群java大神们所开发，所以lint的uast api对于kotlin和java都适用。

AST and UAST

UAST -UNIVERSAL ABSTRACT SYNTAX TREE（通用抽象语法树）：
某几种语言源代码抽象语法结构的树状表现形式

那么明白了AST，Uast就很简单. Uast, 通用抽象语法树, 是某几种语言源代码抽象语法结构的树状表现形式。

UAST的存在源于很多计算机语言在语法结构上的相似性。比如刚才我们看到的欧几里得算法数就通用于java和c。对于我们今天的内容而言，因为kotlin语言是由一群java大神们所开发，所以lint的uast api对于kotlin和java都适用。

定制 LINT

Android Lint 的 AST 和 UAST

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)

Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:

(click)UElement - the common interface for all Uast elements.

(click)UClass

(click)UField

(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

- UElement

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

- UElement
- UClass

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

- UElement
- UClass
- UField

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

- UElement
- UClass
- UField
- UMethod

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

- UElement
- UClass
- UField
- UMethod

...

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

- UElement
- UClass
- UField
- UMethod

...

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

定制 LINT

Android Lint 的 AST 和 UAST

LOMBOK AST -> PSI -> UAST

常见元素:

- UElement
- UClass
- UField
- UMethod

...

UAST API

最初时期，Android Lint 使用 Lombok AST API, 然后在 2.2 PSI完全取而代之. 现在 Uast 也被加入来支持Kotlin.(click)
Android Lint的 Uast 函式库是由kotlin的拥有者JetBrains开发. /一些常见元素有:
(click)UElement - the common interface for all Uast elements.
(click)UClass
(click)UField
(click)UMethod

Android Lint 的 AST 和 UAST

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

- PsiElement

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

- PsiElement
- PsiClass

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

- PsiElement
- PsiClass
- PsiField

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

- PsiElement
- PsiClass
- PsiField
- PsiMethod

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

- PsiElement
- PsiClass
- PsiField
- PsiMethod

...

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

- PsiElement
- PsiClass
- PsiField
- PsiMethod

...

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

Android Lint 的 AST 和 UAST

PSI API 仍然和 UAST 并存:

常见元素:

- PsiElement
- PsiClass
- PsiField
- PsiMethod

...

PSI API

另一方面，PSI API，同样来自JetBrains, 仍然和UAST并存，接下来的代码中你会看到这对与一些java特有的功能非常有帮助.

(click) 常见元素和Uast很相像：

PsiElement - the common interface for all Psi objects.
PsiClass
PsiField
PsiMethod

定制 LINT

这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。

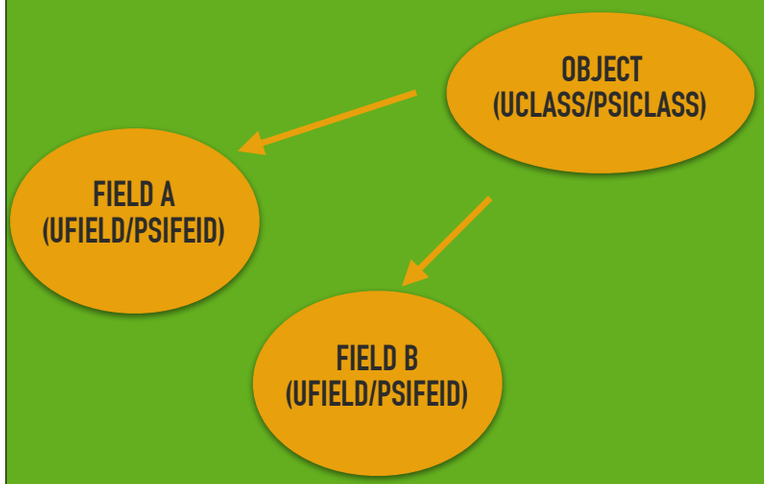
定制 LINT

这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。

定制 LINT

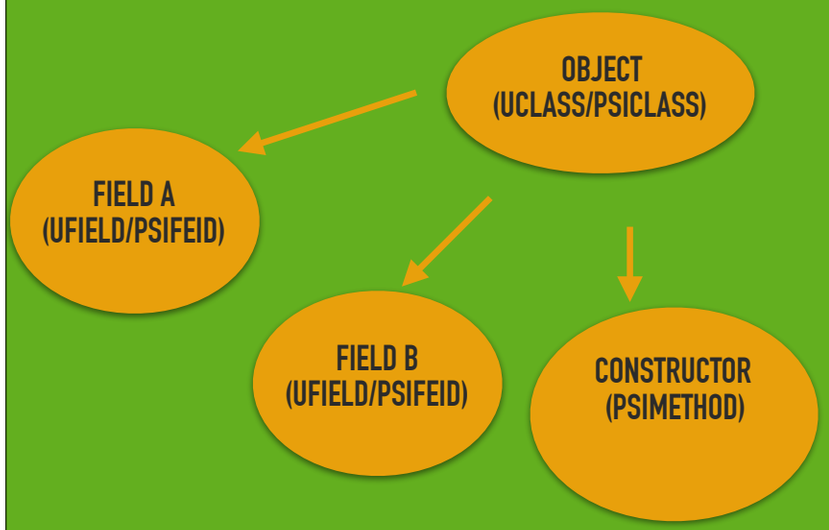
这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。

定制 LINT



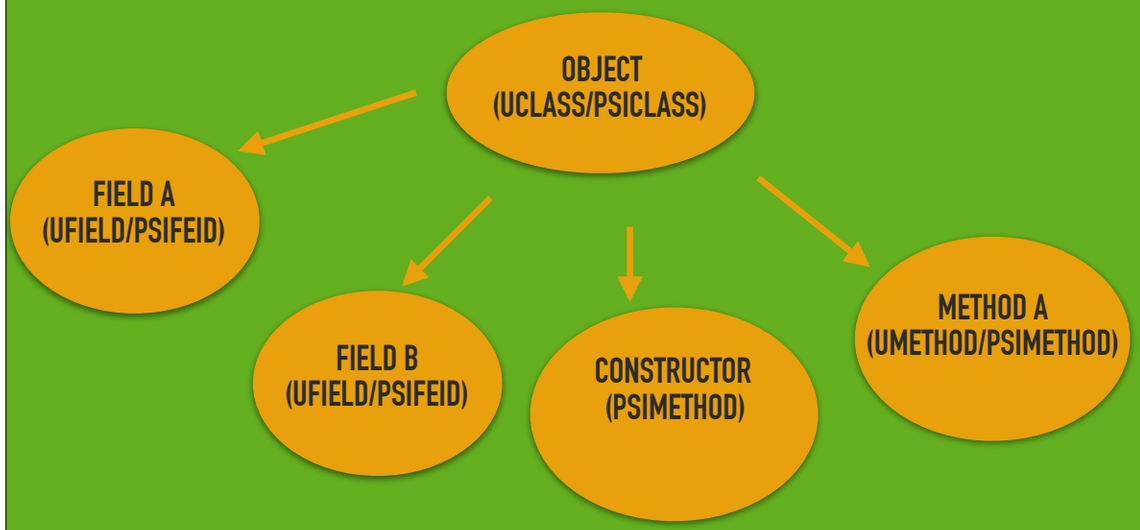
这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。

定制 LINT



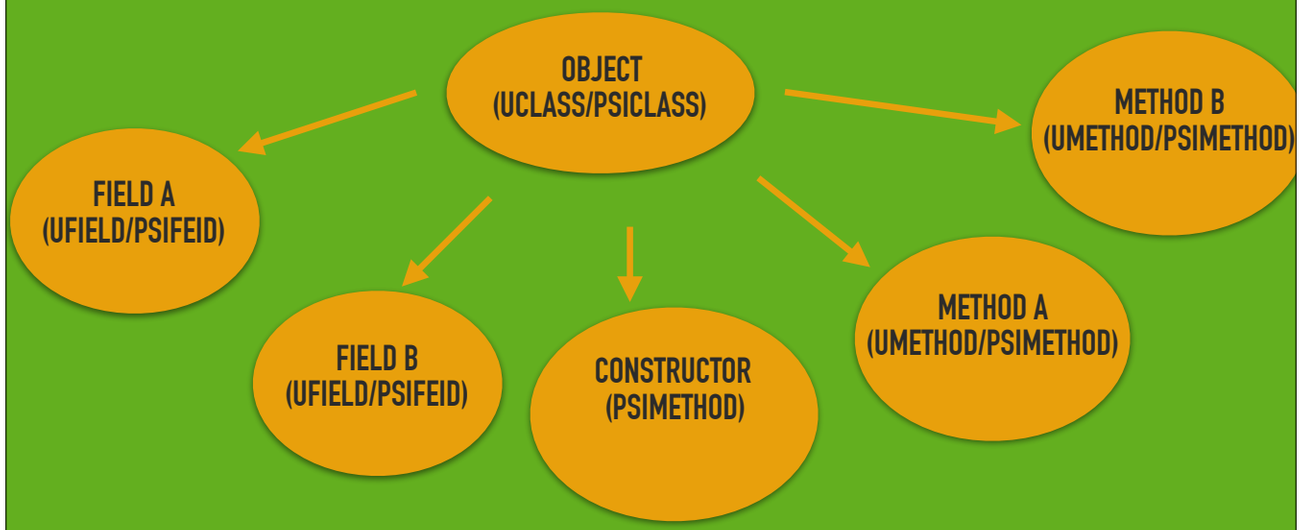
这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。

定制 LINT

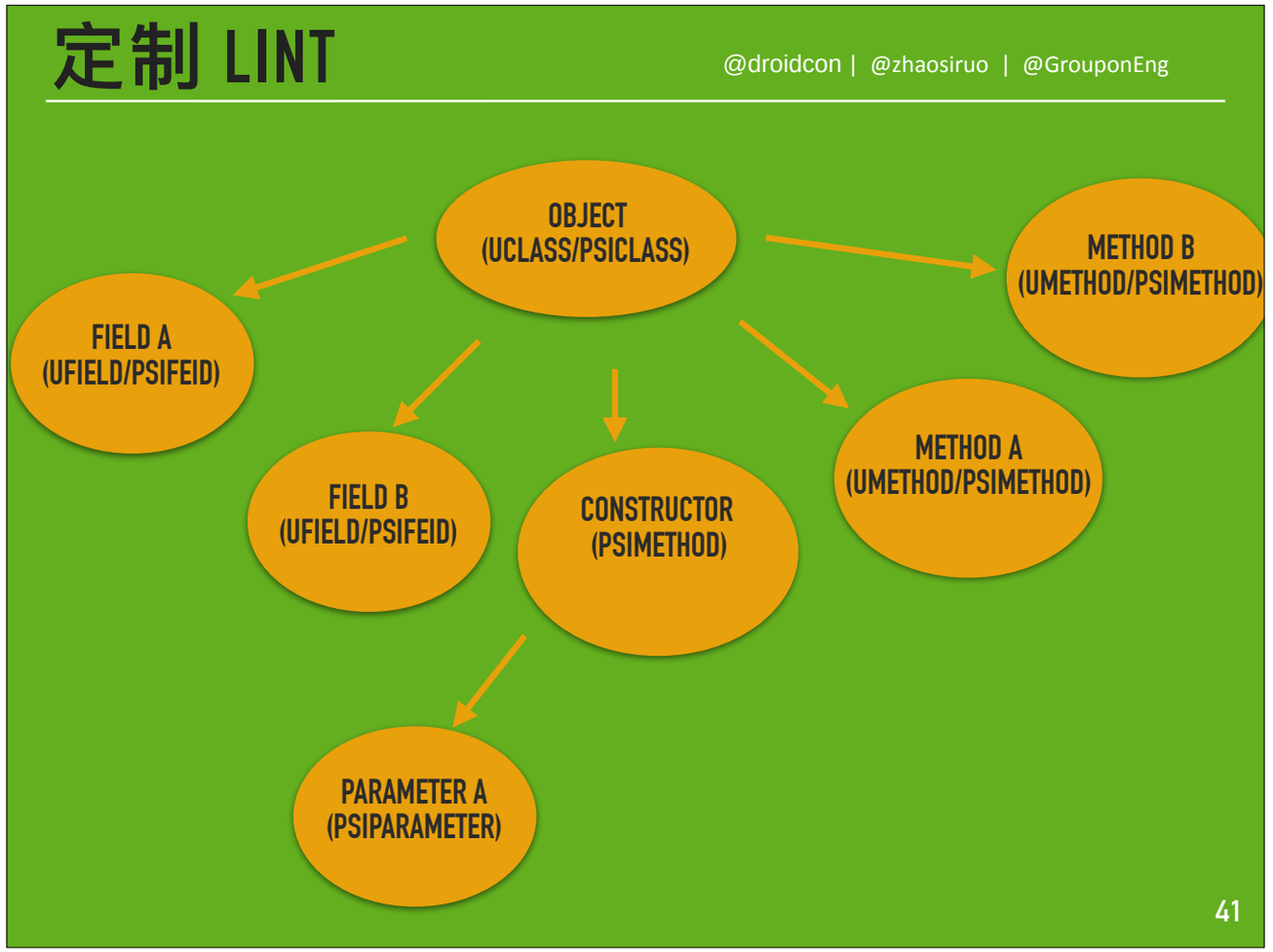


这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。

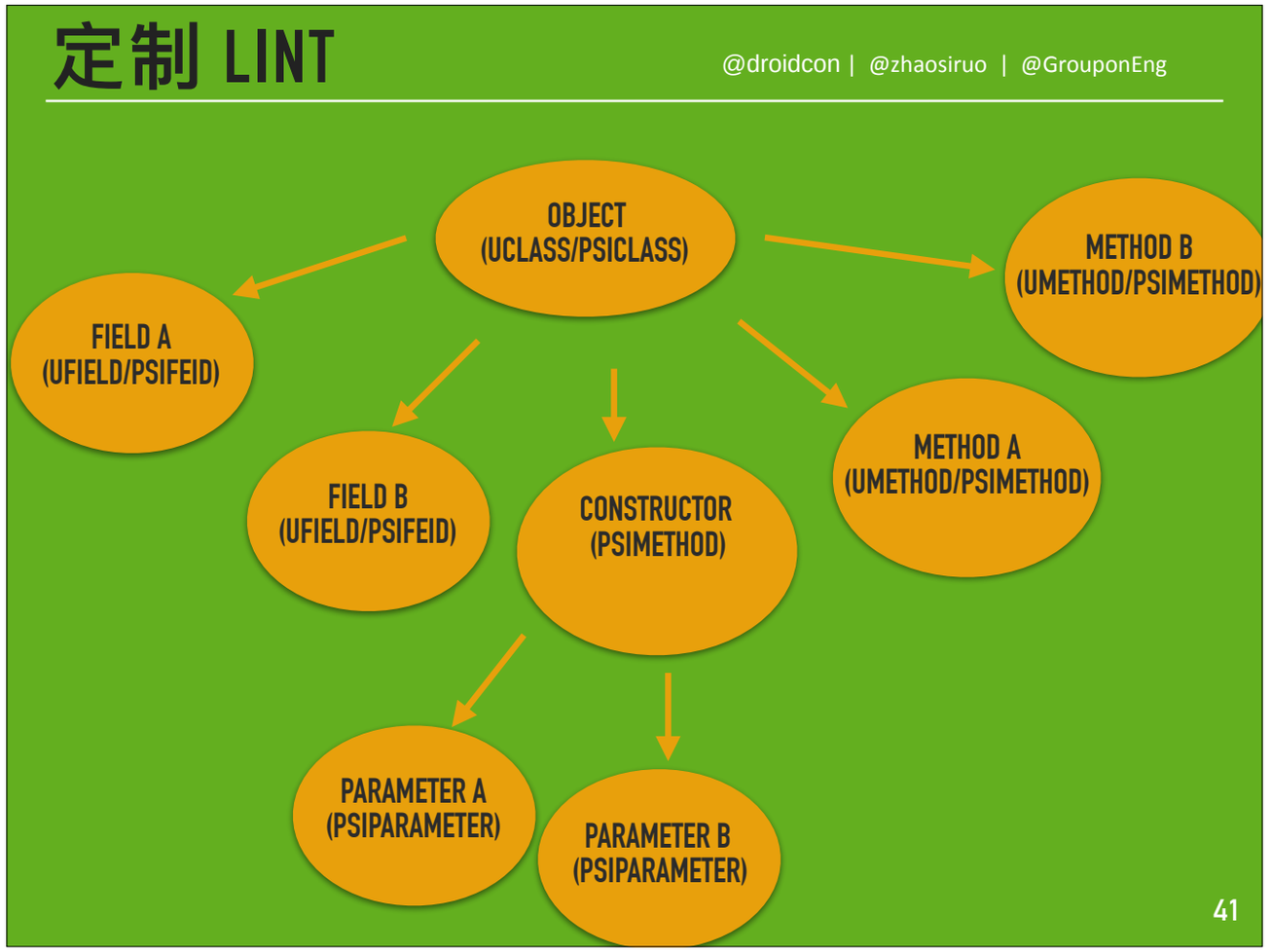
定制 LINT



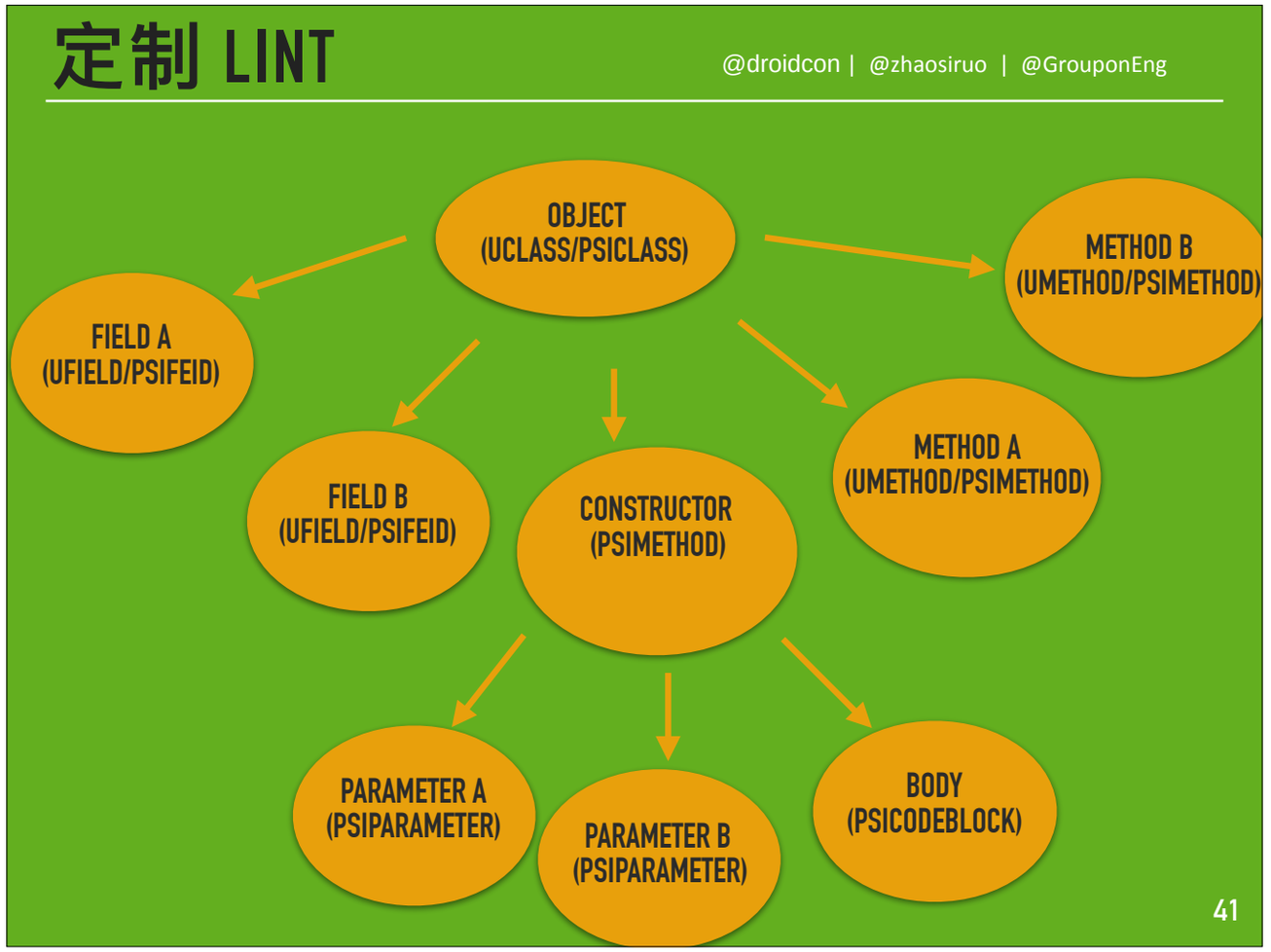
这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。



这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。



这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。



这里是我们构造器的语法树示例。注意 Psi 和 Uast的混用。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    @Override
    public List<Class<? extends UElement>> getApplicableUastTypes() {
        return Collections.singletonList(UClass.class);
    }

    …
}
```

现在回到我们的scanner，我们需要override getApplicableUastTypes() 并且 return Class，以便于检察它的构造器。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

    @Override
    public List<Class<? extends UElement>> getApplicableUastTypes() {
        return Collections.singletonList(UClass.class);
    }

    …
}
```

```
public interface UastScanner {

    @Nullable
    List<Class<? extends UElement>> getApplicableUastTypes();
    …
}
```

现在回到我们的scanner，我们需要override getApplicableUastTypes() 并且 return Class，以便于检察它的构造器。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {
    …

    @Override
    public UElementHandler createUastHandler(final JavaContext
context) {

        return new UElementHandler() {…}
    }
}
```


然后override UastScanner 的createUastHandler()。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {
    …

    @Override
    public UElementHandler createUastHandler(final JavaContext
context) {

        return new UElementHandler() {…}
    }
}
```

```
public interface UastScanner {

   @Nullable
   UElementHandler createUastHandler(@NonNull JavaContext context);
   …
}
```


然后override UastScanner 的createUastHandler()。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {
  …

  @Override
  public void visitClass(UClass uClass) {
    …
  }
}
```


然后我们需要override 这个handler的visitClass. UElemenrHandler里面有很多visit 函式来供我们使用，比如visannotiation，visitAttribute。但是一定记住不要return error，它会返回一个error.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {
  …

  @Override
  public void visitClass(UClass uClass) {
    …
  }
}
```

OBJECT
(UCLASS/PSICLASS)

然后我们需要override 这个handler的visitClass. UElemenrHandler里面有很多visit 函式来供我们使用，比如visannotiation，visitAttribute。但是一定记住不要return error，它会返回一个error.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {
  …

  @Override
  public void visitClass(UClass uClass) {
    …
  }
}
```

OBJECT
(UCLASS/PSICLASS)

```
public class UElementHandler {

    public void visitAnnotation(@NonNull UAnnotation uAnnotation) {
        error(UAnnotation.class);
    }

    public void visitClass(@NonNull UClass uClass) {
        error(UClass.class);
    }
    …
}
```


然后我们需要override 这个handler的visitClass. UElemenrHandler里面有很多visit 函式来供我们使用，比如visannotiation，visitAttribute。但是一定记住不要return error，它会返回一个error.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    …
  }
}
```


Lint 自带很多方便的功能，比如我这里使用的uClass.getConstructors()。如果自己编写这个的话会非常繁琐。所以这里一个小建议就是，在自己动手之前，先看看Lint是不是已经为你准备好了捷径，毕竟站在巨人的肩上才能看得更远。

注意，uClass.getConstructors返回的是一个PsiMethod array。

同时，在AST的角度来看，我们已经从object层面到达了constructor层面。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    …
  }
}
```

OBJECT
(UCLASS/PSICLASS)

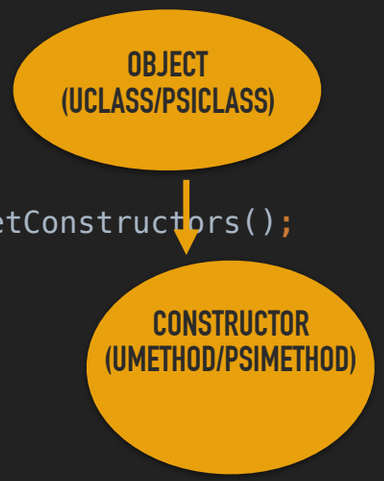
Lint 自带很多方便的功能，比如我这里使用的uClass.getConstructors()。如果自己编写这个的话会非常繁琐。所以这里一个小建议就是，在自己动手之前，先看看Lint是不是已经为你准备好了捷径，毕竟站在巨人的肩上才能看得更远。

注意，uClass.getConstructors返回的是一个PsiMethod array。

同时，在AST的角度来看，我们已经从object层面到达了constructor层面。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    …
  }
}
```



Lint 自带很多方便的功能，比如我这里使用的uClass.getConstructors()。如果自己编写这个的话会非常繁琐。所以这里一个小建议就是，在自己动手之前，先看看Lint是不是已经为你准备好了捷径，毕竟站在巨人的肩上才能看得更远。

注意，uClass.getConstructors返回的是一个PsiMethod array。

同时，在AST的角度来看，我们已经从object层面到达了constructor层面。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    if (constructors.length > 0) {
      for (PsiMethod constructor : constructors) {
        …
      }
    }
  }
}
```


然后这里如果我们确定有一些constructors, 我们就需要遍历。因为我们之前已经假设只有一个constructor，所以其实这里只是拿到了第一个。

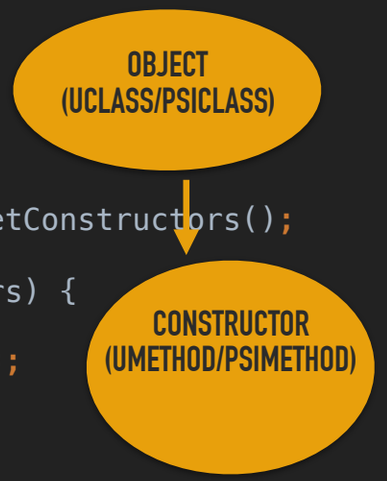
.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    if (constructors.length > 0) {
      for (PsiMethod constructor : constructors) {
        final PsiParameterList parameters =
                constructor.getParameterList();
        …
      }
    }
  }
}
```


然后另一个方便的lint自带函式，PsiMethod.getparameterList。
这时我们已经又到了AST的更深一层，parameters层面。

.../src/main/java/com/example/customlint/JavaConstructorDetector

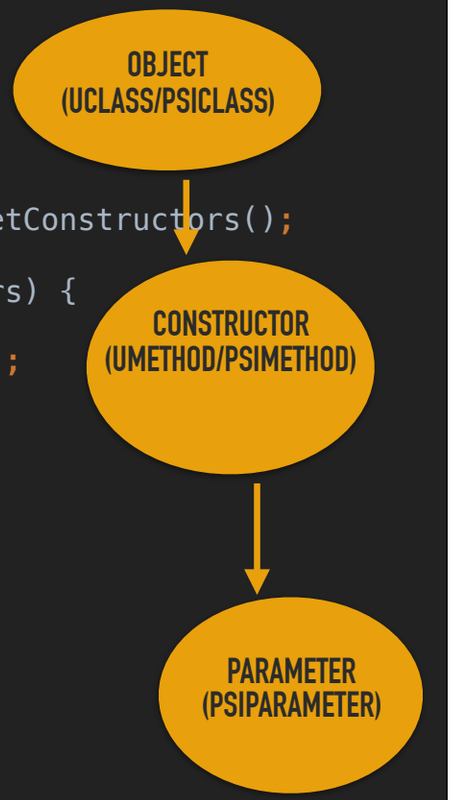
```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    if (constructors.length > 0) {
      for (PsiMethod constructor : constructors) {
        final PsiParameterList parameters =
                constructor.getParameterList();
        …
      }
    }
  }
}
```



然后另一个方便的lint自带函式，PsiMethod.getparameterList。
这时我们已经又到了AST的更深一层，parameters层面。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    if (constructors.length > 0) {
      for (PsiMethod constructor : constructors) {
        final PsiParameterList parameters =
                constructor.getParameterList();
        …
      }
    }
  }
}
```



然后另一个方便的lint自带函式，PsiMethod.getparameterList。
这时我们已经又到了AST的更深一层，parameters层面。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    if (constructors.length > 0) {
      for (PsiMethod constructor : constructors) {
        final PsiParameterList parameters =
                constructor.getParameterList();
        if (parameters.getParametersCount() > 2) {
          …
        }
      }
    }
  }
}
```


现在我就定义我不想要两个以上的构造参数。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    if (constructors.length > 0) {
      for (PsiMethod constructor : constructors) {
        final PsiParameterList parameters =
                constructor.getParameterList();
        if (parameters.getParametersCount() > 2) {
          context.report(TOO_MANY_PARAMETERS_ISSUE,
                         constructor,
                         context.getLocation(constructor),

    TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT));
          }
        }
      }
    }
  }
}
```


否则就要向Lint报告。context.report(issue id, scope 也就是出错的PsiElement, 和 location, 和issue的简短描述.
这就是我们用来检查too many parameters constrcutor issue的scanner。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    final PsiMethod[] constructors = uClass.getConstructors();
    if (constructors.length > 0) {
      for (PsiMethod constructor : constructors) {
        final PsiParameterList parameters =
                constructor.getParameterList();
        if (parameters.getParametersCount() > 2) {
          context.report(TOO_MANY_PARAMETERS_ISSUE,
                         constructor,
                         context.getLocation(constructor),

    TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT));
          }
        }
      }
    }
  }
}
```

```
@JvmOverloads fun report(
            issue: Issue,
            scope: PsiElement?,
            location: Location,
            message: String) {

}
```


否则就要向Lint报告。context.report(issue id, scope 也就是出错的PsiElement, 和 location, 和issue的简短描述.
这就是我们用来检查too many parameters constrcutor issue的scanner。

定制 LINT

LINT 规则的组成

- DETECTOR: LINT 规则的主要逻辑
  - Issue: 设立所检测的问题
  - Scanner: check 和 fix的逻辑
- ISSUEREGISTRY: ISSUES 的 LIST
- 测试!

复习一下Lint规则的组成，有了issue和scanner，detector就完成了。

LINT 规则的组成

- DETECTOR: LINT 规则的主要逻辑
  - Issue: 设立所检测的问题
  - Scanner: check 和 fix的逻辑
- ISSUEREGISTRY: ISSUES 的 LIST
- 测试!

下一步就需要向issue registry报备这个新建的issue.

.../src/main/java/com/example/customlint/CustomIssueRegistry

```
public class CustomIssueRegistry extends IssueRegistry {

  public CustomIssueRegistry() {
  }

  …
}
```


和detector相比IssueRegistry就简单多了. 命名，extend IssueRegistry, 建一个构造器.

.../src/main/java/com/example/customlint/CustomIssueRegistry

```
public class CustomIssueRegistry extends IssueRegistry {
  private static final int INITIAL_CAPACITY = 1;

  public CustomIssueRegistry() {
  }

  static {
    List<Issue> issues = new ArrayList<>(INITIAL_CAPACITY);
  …}


}
```


初始化我们的 issue list.

.../src/main/java/com/example/customlint/CustomIssueRegistry

```
public class CustomIssueRegistry extends IssueRegistry {
  private static final List<Issue> lintIssues;
  private static final int INITIAL_CAPACITY = 1;

  public CustomIssueRegistry() {
  }

  static {
    List<Issue> issues = new ArrayList<>(INITIAL_CAPACITY);

    issues.add(JavaConstructorDetector.TOO_MANY_PARAMETERS_ISSUE);
      lintIssues = Collections.unmodifiableList(issues);
    }

}
```


然后加入我们的定制lint issue.

.../src/main/java/com/example/customlint/CustomIssueRegistry

```
public class CustomIssueRegistry extends IssueRegistry {
  private static final List<Issue> lintIssues;
  private static final int INITIAL_CAPACITY = 1;

  public CustomIssueRegistry() {
  }

  static {
    List<Issue> issues = new ArrayList<>(INITIAL_CAPACITY);

    issues.add(JavaConstructorDetector.TOO_MANY_PARAMETERS_ISSUE);
      lintIssues = Collections.unmodifiableList(issues);
    }

  @Override
  public List<Issue> getIssues() {
    return lintIssues;
  }
}
```


最后 override getIssues 并且 return 我们的 list.

LINT 规则的组成

- DETECTOR: LINT 规则的主要逻辑
  - Issue: 设立所检测的问题
  - Scanner: check 和 fix的逻辑
- ISSUEREGISTRY: ISSUES 的 LIST
- 测试!

所以这就是IssueRegistry。

LINT 规则的组成

- DETECTOR: LINT 规则的主要逻辑
  - Issue: 设立所检测的问题
  - Scanner: check 和 fix的逻辑
- ISSUEREGISTRY: ISSUES 的 LIST
- 测试!

终于到了激动人心的测试环节!

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    ...
}
```


我前面说过，Lint的test api设计合理，使用方便. 我们首先需要 extend LintDetectorTest.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    …
    @Override
    protected Detector getDetector() {
        return new JavaConstructorDetector();
    }



}
```


Override getDetector() 并且 return JavaConstructorDetector.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    …

    @Override
    protected List<Issue> getIssues() {
        return Collections.singletonList(JavaConstructorDetector.
                TOO_MANY_PARAMETERS_ISSUE);
    }
}
```


Override getIssues() 并且 return detector的issue list. 我们目前只有一个.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    …

    @Override
    protected boolean allowCompilationErrors() {
        return true;
    }

}
```


然后override allowCompilationErrors 并且 return true 所以你不需要一份完美的代码以供测试，你马上就会明白为什么要这样.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    public void testNoArgConstructor() throws Exception {
        …
    }
}
```


那么set up工作完成，首先, 我们要测试没有构造参数的情况.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    public void testNoArgConstructor() throws Exception {
        @Language("JAVA") final String SOURCE = …;

        …
    }
}
```


这里需要一份用来测试的代码，定义语言为Java.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    public void testNoArgConstructor() throws Exception {
        @Language("JAVA") final String SOURCE = ""
            + "package test.pkg;\n"
            + "public class NoArgConstructorTestObject {\n"
            + "  public NoArgConstructorTestObject() {\n"
            + "    System.out.println(\"This is a no argument
              constructor\");\n"
            + "  }\n"
            + "}";

        …
    }
}
```


然后这里是一个没有构造参数的java类.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    public void testNoArgConstructor() throws Exception {
        @Language("JAVA") final String SOURCE = ""
            + "package test.pkg;\n"
            + "public class NoArgConstructorTestObject {\n"
            + "  public NoArgConstructorTestObject() {\n"
            + "    System.out.println(\"This is a no argument
constructor\");\n"
            + "  }\n"
            + "}";

        lint()
            .files(java(SOURCE))
            .allowMissingSdk(true)
            .run()
            .expect("No warnings.");
    }
}
```


当然这里不应该报错.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public void testThreeParametersConstructor() throws Exception {
    @Language("JAVA") final String SOURCE = ""
        + "package test.pkg;\n"
        + "public class ThreeParametersConstructorTestObject {\n"
        + "\tpublic ThreeParametersConstructorTestObject(int a, long
b, boolean c) {\n”
        …
        + "\t}\n"
        + "}";

    lint()
        .files(java(SOURCE))
        .run()
        .expect("src/test/pkg/
ThreeParametersConstructorTestObject.java:3: Warning: Constructor has
too many parameters. [TooManyParametersConstructor]\n" +
                    " public ThreeParametersConstructorTestObject(int
a, long b, boolean c) {\n" +
                    " ^\n" +
                    "0 errors, 1 warnings\n”)
}
```


然后测试过多构造参数的情况。我们的测试代码中出现了一个有三个构造参数的构造器(click). 这里我们希望lint有警告(click)。这里需要注意警告信息的格式，最好的方法恐怕就是运行一遍然后对比实际的警告。

这就是我们detector的测试。IssueRegistry的测试和普通的单元测试并无不同，只要看一眼github里面的例子你就会明白。

现在我们已经有了一个完整的定制lint规则，如何让它和自带lint同时运行呢?
回到我们的gradle.build。

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public void testThreeParametersConstructor() throws Exception {
    @Language("JAVA") final String SOURCE = ""
        + "package test.pkg;\n"
        + "public class ThreeParametersConstructorTestObject {\n"
        + "\tpublic ThreeParametersConstructorTestObject(int a, long
b, boolean c) {\n”
        …
        + "\t}\n"
        + "}";

    lint()
        .files(java(SOURCE))
        .run()
        .expect("src/test/pkg/
ThreeParametersConstructorTestObject.java:3: Warning: Constructor has
too many parameters. [TooManyParametersConstructor]\n" +
                    " public ThreeParametersConstructorTestObject(int
a, long b, boolean c) {\n" +
                    " ^\n" +
                    "0 errors, 1 warnings\n”)
}
```


然后测试过多构造参数的情况。我们的测试代码中出现了一个有三个构造参数的构造器(click). 这里我们希望lint有警告(click)。这里需要注意警告信息的格式，最好的方法恐怕就是运行一遍然后对比实际的警告。

这就是我们detector的测试。IssueRegistry的测试和普通的单元测试并无不同，只要看一眼github里面的例子你就会明白。

现在我们已经有了一个完整的定制lint规则，如何让它和自带lint同时运行呢?
回到我们的gradle.build。

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public void testThreeParametersConstructor() throws Exception {
    @Language("JAVA") final String SOURCE = ""
        + "package test.pkg;\n"
        + "public class ThreeParametersConstructorTestObject {\n"
        + "\tpublic ThreeParametersConstructorTestObject(int a, long
b, boolean c) {\n”
        …
        + "\t}\n"
        + "}";

    lint()
        .files(java(SOURCE))
        .run()
        .expect("src/test/pkg/
ThreeParametersConstructorTestObject.java:3: Warning: Constructor has
too many parameters. [TooManyParametersConstructor]\n" +
                    " public ThreeParametersConstructorTestObject(int
a, long b, boolean c) {\n" +
                    " ^\n" +
                    "0 errors, 1 warnings\n”)
}
```


然后测试过多构造参数的情况。我们的测试代码中出现了一个有三个构造参数的构造器(click). 这里我们希望lint有警告(click)。这里需要注意警告信息的格式，最好的方法恐怕就是运行一遍然后对比实际的警告。

这就是我们detector的测试。IssueRegistry的测试和普通的单元测试并无不同，只要看一眼github里面的例子你就会明白。

现在我们已经有了一个完整的定制lint规则，如何让它和自带lint同时运行呢?
回到我们的gradle.build。

CustomLint/customlint/build.gradle

```
apply plugin: 'java-library'

…

jar {
    baseName 'CustomLint'
    version '1.0'

    manifest {
        attributes 'Manifest-Version': 1.0
        attributes('Lint-Registry':
                   'com.example.customlint.CustomIssueRegistry')
    }
}
```


我需要把这个模块建成一个jar文件，(click) 并且指向 我们的CustomIssueRegistry.

CustomLint/customlint/build.gradle

```
apply plugin: 'java-library'

…

task copyLintJar(type: Copy) {
    description = 'Copies the lint jar file into the
{user.home}/.android/lint folder.'
    from('build/libs/')
    into(System.getProperty("user.home") + '/.android/lint')
    include("*.jar")
}
```


然后需要一个 copyLintJar task 来把我们的定制规则放进 Android Lint 文件夹。这里是相应的gradle wrapper指令 (click)
你也可以将这个函式库放在jcenter或者maven中共享，方法我有在最后的资源页提到。

现在让我们运行一下。

CustomLint/customlint/build.gradle

```
apply plugin: 'java-library'

…

task copyLintJar(type: Copy) {
    description = 'Copies the lint jar file into the
{user.home}/.android/lint folder.'
    from('build/libs/')
    into(System.getProperty("user.home") + '/.android/lint')
    include("*.jar")
}
```

./gradlew clean :customlint:copyLintJar

然后需要一个 copyLintJar task 来把我们的定制规则放进 Android Lint 文件夹。这里是相应的gradle wrapper指令 (click)
你也可以将这个函式库放在jcenter或者maven中共享，方法我有在最后的资源页提到。

现在让我们运行一下。

定制 LINT

Run!!!

Yay!!! Build Successful!

让我们在真实代码中操练一下。

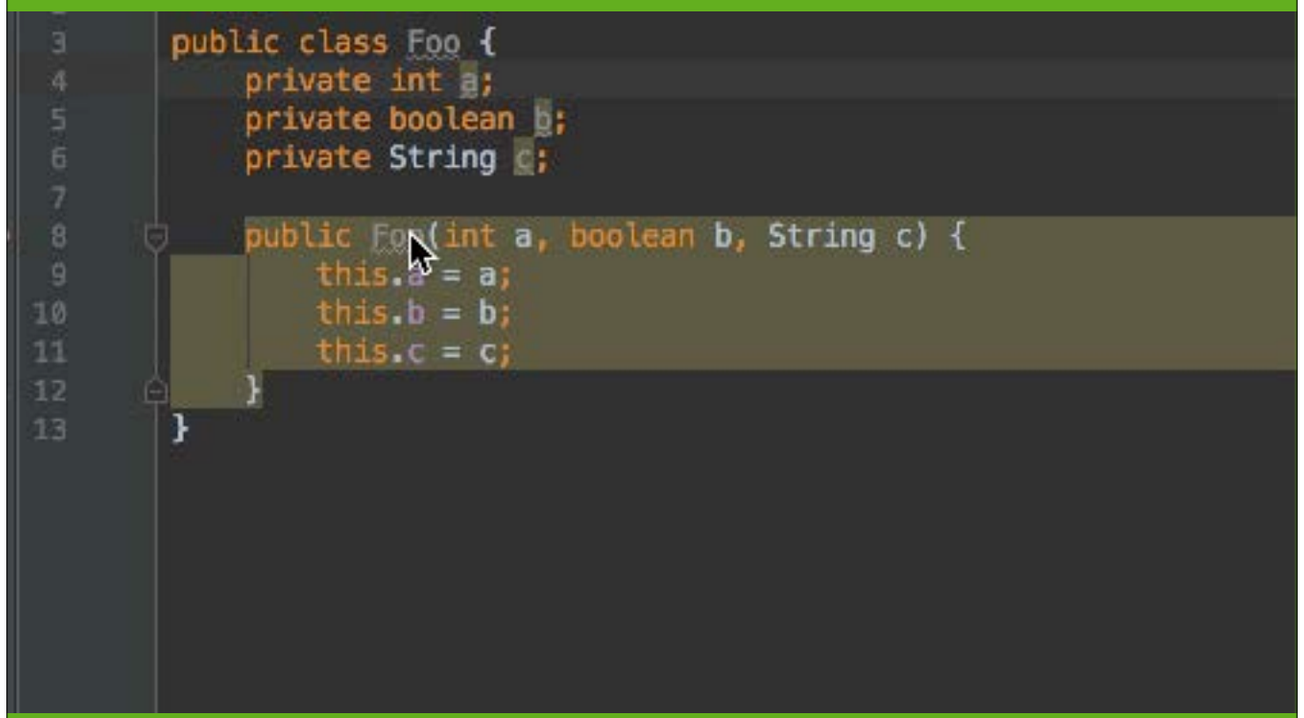
定制 LINT

Run!!!

Yay!!! Build Successful!

让我们在真实代码中操练一下。

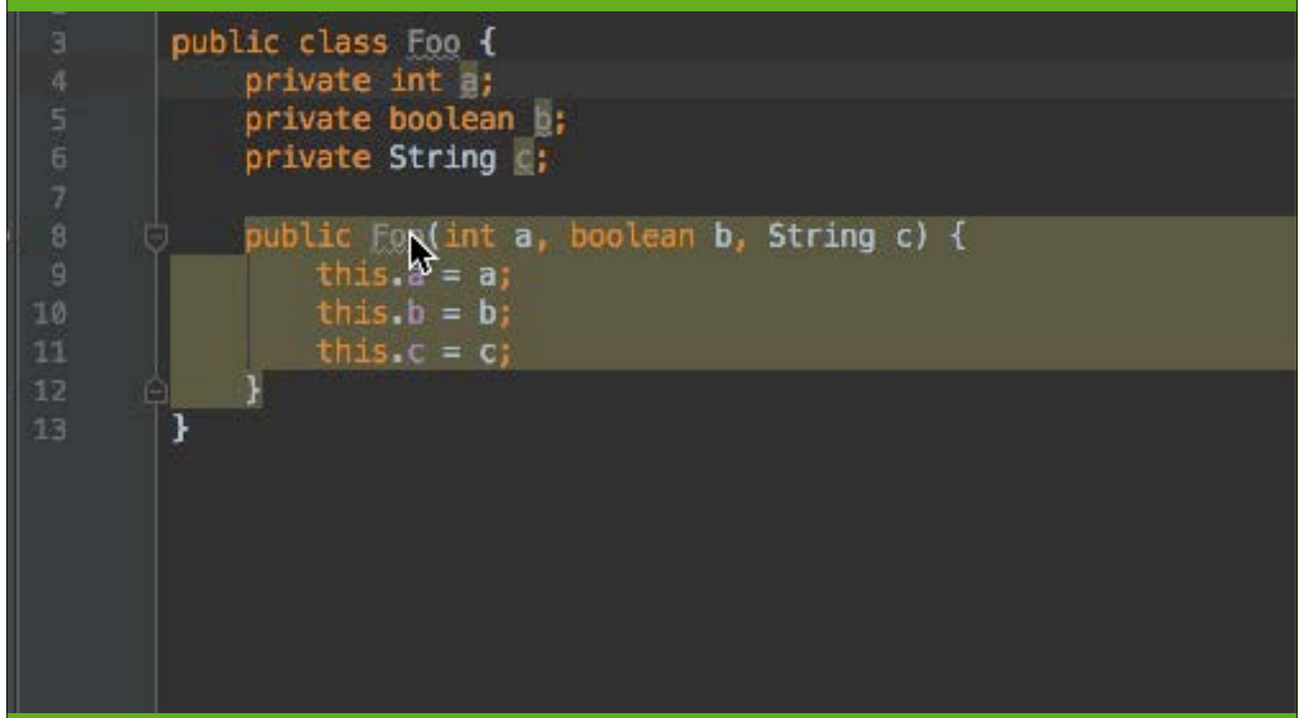
定制 LINT



我们的定制lint完美地指出了问题。那么，我们结束了嘛?

我之前有提到，从Gradle 3.0开始，定制规则也可以提出修改建议。目前为止，修改方式仅限于 修改出错处的代码，或者xml的attributes。幸运的是这适用于绝大多数情况，现在让我们来看看如何建议一个builder模式

定制 LINT



我们的定制lint完美地指出了问题。那么，我们结束了嘛?

我之前有提到，从Gradle 3.0开始，定制规则也可以提出修改建议。目前为止，修改方式仅限于 修改出错处的代码，或者xml的attributes。幸运的是这适用于绝大多数情况，现在让我们来看看如何建议一个builder模式

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > 2) {

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

     TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT));

    }
}
```


回到Detector的visitClass.

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > 2) {

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

     TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT));

    }
}
```

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > 2) {
      final LintFix fix = fix()
                          .replace()
                          .text(constructor.getText())
                          .with(getBuilderPatternText(constructor))
                          .reformat(false)
                          .build();

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

     TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT),
     fix);

    }
}
```


我们需要加入一个LintFix 类。这里我说我需要将现有的构造器代码换成由private helper函式得到的builder模式代码。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > 2) {
      final LintFix fix = fix()
                          .replace()
                          .text(constructor.getText())
                          .with(getBuilderPatternText(constructor))
                          .reformat(false)
                          .build();

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

     TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT),
     fix);

    }
}
```

```
@JvmOverloads fun report(
            issue: Issue,
            scope: PsiElement?,
            location: Location,
            message: String,
            quickfixData: LintFix? = null) {
}
```


我们需要加入一个LintFix 类。这里我说我需要将现有的构造器代码换成由private helper函式得到的builder模式代码。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
private String getBuilderPatternText(final PsiMethod constructor) {
    final StringBuilder stringBuilder = new StringBuilder();
    final String objectName = constructor.getName();
    final PsiParameterList parameterList =
constructor.getParameterList();
    final PsiParameter[] parameters = parameterList.getParameters();

    stringBuilder.append(getBuilderConstructorText(objectName,
parameters));

    stringBuilder.append(getStaticFactoryText(objectName,
parameters));

    return stringBuilder.toString();
}
```

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
private String getStaticFactoryText(@NonNull final String objectName,
@NonNull final PsiParameter[] parameters) {
    final StringBuilder stringBuilder = new StringBuilder();

    //    public static class Builder {
    stringBuilder.append(TAB);
    stringBuilder.append(PUBLIC);
    stringBuilder.append(SPACE);
    stringBuilder.append(STATIC);
    stringBuilder.append(SPACE);
    stringBuilder.append(CLASS);
    stringBuilder.append(SPACE);
    stringBuilder.append(BUILDER);
    stringBuilder.append(SPACE);
    stringBuilder.append(LEFT_CURLY_BRACKET);
    stringBuilder.append(NEW_LINE);
    ...
}
```


你可以看到其实一字一句地构造java代码是很枯燥冗长的，好在 Square 的Java Poet 可以将这项工作变得简单轻松，感兴趣可以自己看一下。

所以现在fix就搞定了。不要忘记相应地修改测试。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
private String getStaticFactoryText(@NonNull final String objectName,
@NonNull final PsiParameter[] parameters) {
    final StringBuilder stringBuilder = new StringBuilder();

    //    public static class Builder {
    stringBuilder.append(TAB);
    stringBuilder.append(PUBLIC);
    stringBuilder.append(SPACE);
    stringBuilder.append(STATIC);
    stringBuilder.append(SPACE);
    stringBuilder.append(CLASS);
    stringBuilder.append(SPACE);
    stringBuilder.append(BUILDER);
    stringBuilder.append(SPACE);
    stringBuilder.append(LEFT_CURLY_BRACKET);
    stringBuilder.append(NEW_LINE);
    ...
}
```

JAVA POET

你可以看到其实一字一句地构造java代码是很枯燥冗长的，好在 Square 的Java Poet 可以将这项工作变得简单轻松，感兴趣可以自己看一下。

所以现在fix就搞定了。不要忘记相应地修改测试。

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    public void testNoArgConstructor() throws Exception {
        …

        lint()
            .files(java(SOURCE))
            .allowMissingSdk(true)
            .run()
            .expect("No warnings.”);
    }
}
```


这是我们的no arg constructor 情况.

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public class JavaConstructorDetectorTest extends LintDetectorTest {

    public void testNoArgConstructor() throws Exception {
        …

        lint()
            .files(java(SOURCE))
            .allowMissingSdk(true)
            .run()
            .expect("No warnings.”)
            .expectFixDiffs("");
    }
}
```


我们只需要在最后添加expectFixDiffs empty string

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public void testThreeParametersConstructor() throws Exception {
    …
    lint()
        .files(java(SOURCE))
        .run()
        .expect("…")
        .expectFixDiffs(""
                    + "Fix for src/test/pkg ThreeParameters-
ConstructorTestObject.java line 2: Replace with "
                    + "private ThreeParametersConstructorTestObject-
(Builder builder) {\n"
                    + "\t\tthis.a = builder.a;\n"
                    + "\t\tthis.b = builder.b;\n"
                    + "\t\tthis.c = builder.c;\n"
                    + "\t}\n"…);
}
```


同理，在过多构造参数的情况下，我们只需要加入预期的fix diffs。(click)实际的fix diffs比这个要长很多，所以最佳方法还是运行一遍来对比实际输出。请格外注意空格和tap的区别。

.../src/test/java/com/example/customlint/JavaConstructorDetectorTest

```
public void testThreeParametersConstructor() throws Exception {
    …
    lint()
        .files(java(SOURCE))
        .run()
        .expect("…")
        .expectFixDiffs(""
                + "Fix for src/test/pkg ThreeParameters-
ConstructorTestObject.java line 2: Replace with "
                + "private ThreeParametersConstructorTestObject-
(Builder builder) {\n"
                + "\t\tthis.a = builder.a;\n"
                + "\t\tthis.b = builder.b;\n"
                + "\t\tthis.c = builder.c;\n"
                + "\t}\n"…);
}
```


同理，在过多构造参数的情况下，我们只需要加入预期的fix diffs。(click)实际的fix diffs比这个要长很多，所以最佳方法还是运行一遍来对比实际输出。请格外注意空格和tap的区别。

定制 LINT

Run!!!

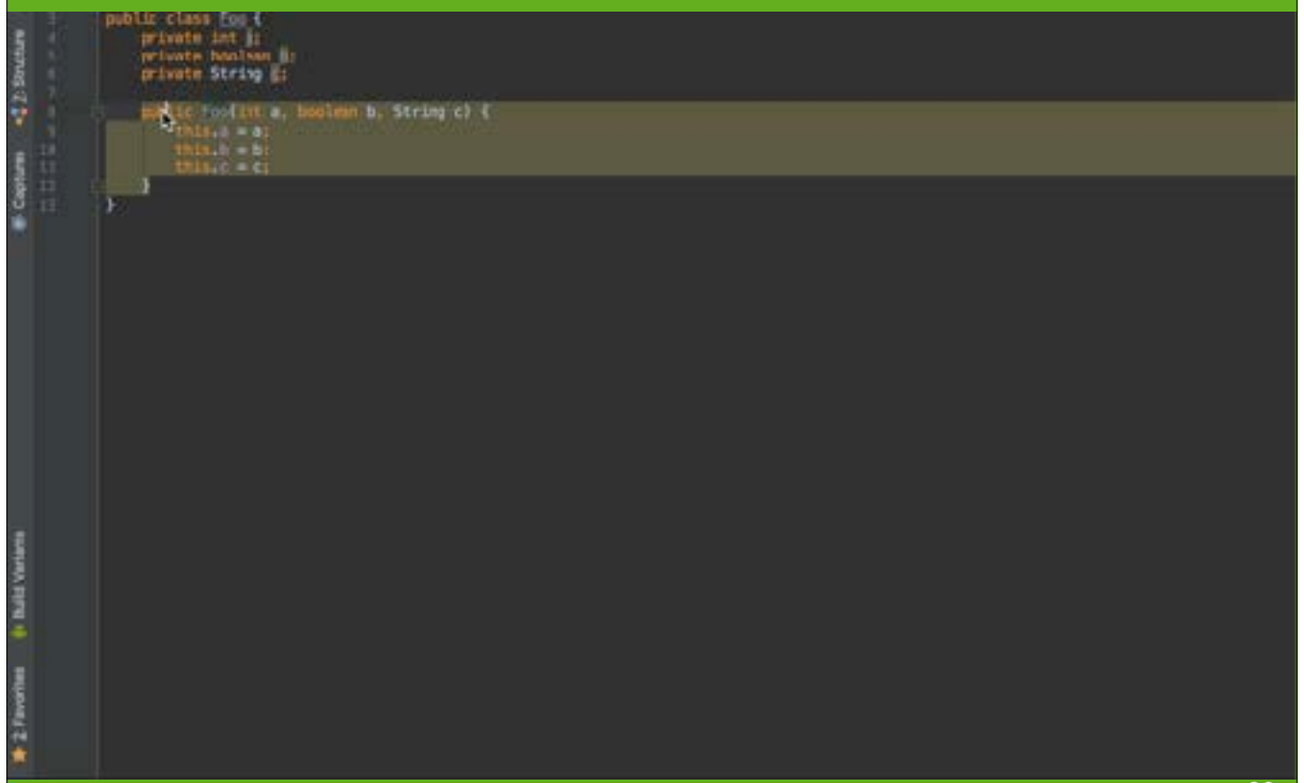
Build Successful!!!

定制 LINT

Run!!!

Build Successful!!!

定制 LINT

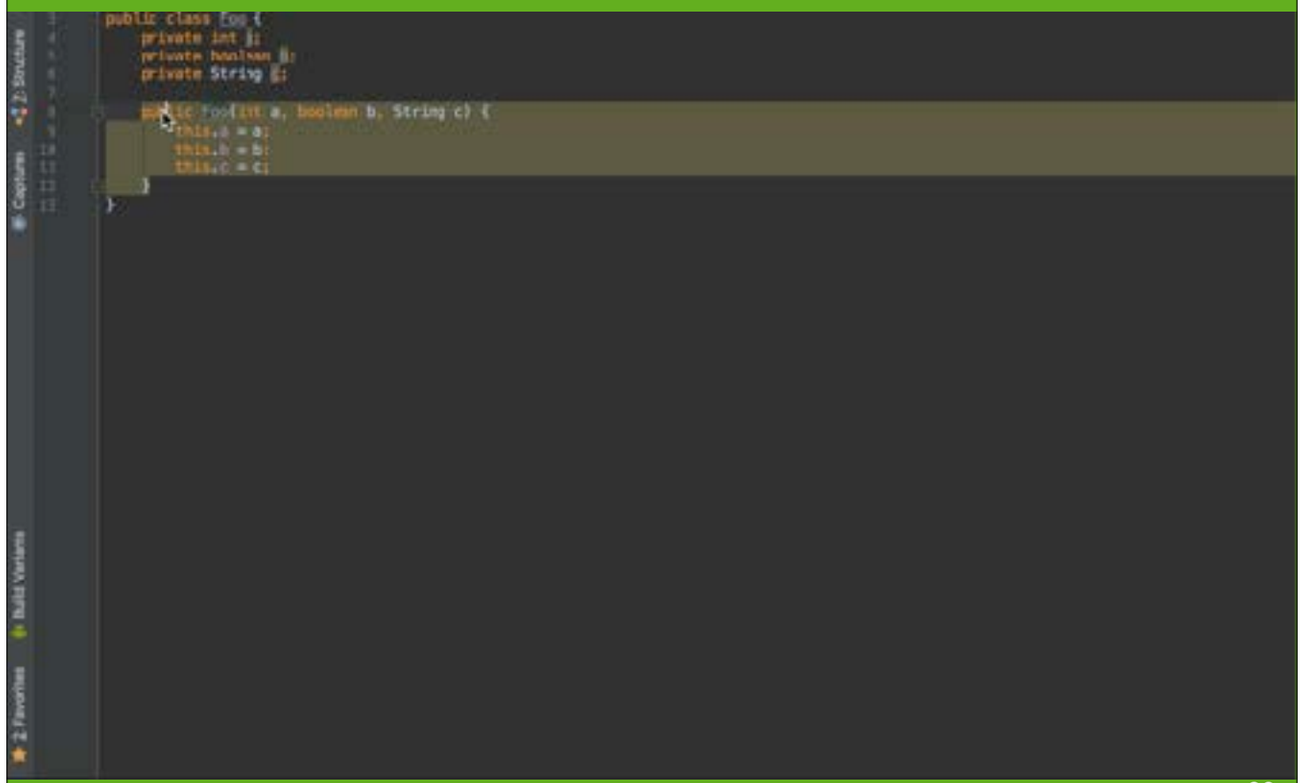


Yay!!! Done!

...Wait...Really?

还记得我在布局过深的例子中说，Lint规则可以用环境参数配置。但是现在我只是hard coding 构造参数的数量限制。

定制 LINT



Yay!!! Done!

...Wait...Really?

还记得我在布局过深的例子中说，Lint规则可以用环境参数配置。但是现在我只是hard coding 构造参数的数量限制。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > 2) {
      final LintFix fix = fix()
                          .replace()
                          .text(constructor.getText())
                          .with(getBuilderPatternText(constructor))
                          .reformat(false)
                          .build();

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

     TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT),
                     fix);

    }
}
```


在这里。

那么为什么不用环境参数呢?

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > 2) {
      final LintFix fix = fix()
                          .replace()
                          .text(constructor.getText())
                          .with(getBuilderPatternText(constructor))
                          .reformat(false)
                          .build();

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

      TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT),
                     fix);

    }
}
```


在这里。

那么为什么不用环境参数呢?

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
public class JavaConstructorDetector extends Detector implements
Detector.UastScanner {

  private static final int PARAMETERS_COUNT_LIMIT;
  static {
    int parametersCount = 0;

    String countValue = System.getenv("ANDROID_LINT_CONSTRUCTOR_
                          PARAMETERS_COUNT_LIMIT");
    if (countValue != null) {
      try {
        parametersCount = Integer.parseInt(countValue);
      } catch (NumberFormatException e) {
        // pass: set to default below
      }
    }
    if (parametersCount <= 0) {
      parametersCount = 2;
    }

  PARAMETERS_COUNT_LIMIT = parametersCount;
  }
  …
}
```


所以，在detector里，我定义了一个 PARAMETERS_COUNT_LIMIT。然后尝试用System.getenv(ANDROID_LINT_CONSTRUCTOR_PARAMETERS_COUNT_LIMIT) 来为其赋值。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > PARAMETERS_COUNT_LIMIT) {
      final LintFix fix = fix()
                              .replace()
                              .text(constructor.getText())
                              .with(getBuilderPatternText(constructor))
                              .reformat(false)
                              .build();

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

     TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT
),
                     fix);

    }
}
```


别忘记在visitclass里将hardcoded limit改成 PARAMETERS_COUNT_LIMIT.

/ / 测试里的改动很简单，只要使用setEnv来 mock这个环境参数就好，所以还是请看github.

很简单的改动，但是它大大提高了我们定制规则的用户友好度。

.../src/main/java/com/example/customlint/JavaConstructorDetector

```
return new UElementHandler() {

  @Override
  public void visitClass(UClass uClass) {
    …
    if (parameters.getParametersCount() > PARAMETERS_COUNT_LIMIT) {
      final LintFix fix = fix()
                              .replace()
                              .text(constructor.getText())
                              .with(getBuilderPatternText(constructor))
                              .reformat(false)
                              .build();

      context.report(TOO_MANY_PARAMETERS_ISSUE,
                     constructor,
                     context.getLocation(constructor),

     TOO_MANY_PARAMETERS_ISSUE.getBriefDescription(TextFormat.TEXT
),
                     fix);

    }
}
```


别忘记在visitclass里将hardcoded limit改成 PARAMETERS_COUNT_LIMIT.

/ / 测试里的改动很简单，只要使用setEnv来 mock这个环境参数就好，所以还是请看github.

很简单的改动，但是它大大提高了我们定制规则的用户友好度。

常用安卓静态代码分析工具

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具：

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具：
- CHECKSTYLE

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具：

- CHECKSTYLE
- PMD

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具：

- CHECKSTYLE
- PMD
- FINDBUGS

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具：
- CHECKSTYLE
- PMD
- FINDBUGS

原生工具:

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具:
- CHECKSTYLE
- PMD
- FINDBUGS

原生工具:
- LINT

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具:
- CHECKSTYLE
- PMD
- FINDBUGS

原生工具:
- LINT

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

常用安卓静态代码分析工具

第三方工具:
- CHECKSTYLE
- PMD
- FINDBUGS

原生工具:
- LINT

今天的代码时间到此结束，让我们放松一下(click for mika picture)/带着对lint深入全面的了解，看一看常用安卓静态代码分析工具。
我会介绍一些 (click)groupon实际使用的第三方工具 (click)checkstyle, (click)pmd, (click)findbags.
(click)然后做一个比较 (click).

CHECKSTYLE

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

CHECKSTYLE

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

规范 JAVA 编码风格。

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

CHECKSTYLE

规范 JAVA 编码风格。

– 缩进

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

CHECKSTYLE

规范 JAVA 编码风格。

– 缩进

– 空格

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

CHECKSTYLE

规范 JAVA 编码风格。

– 缩进

– 空格

– 括号

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

CHECKSTYLE

规范 JAVA 编码风格。

– 缩进

– 空格

– 括号

…

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

CHECKSTYLE

规范 JAVA 编码风格。

– 缩进

– 空格

– 括号

…

[官方指导]

[谷歌 JAVA 风格指导]

代码风格一直是我们学习新语言的第一课。正如其名字所提到的，于2001年问世的checkstyle 代劳了Java工程师们繁琐的代码风格检查，比如, 缩进，空格，括号等等。。。

CheckStyle同样拥有极高的可配置性，谷歌Java风格指导就是一个极佳的实例。

PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

检索源代码中的常见错误。

PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

检索源代码中的常见错误。
– 空白的 TRY/CATCH

PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

检索源代码中的常见错误。
– 空白的 TRY/CATCH
– 使用.EQUALS() 而不是 '=='

PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

PMD

检索源代码中的常见错误。

- 空白的 TRY/CATCH
- 使用.EQUALS() 而不是 '=='
- 未使用的参数和导入

PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

PMD

检索源代码中的常见错误。

– 空白的 TRY/CATCH

– 使用.EQUALS() 而不是 '=='

– 未使用的参数和导入

…

PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

PMD

检索源代码中的常见错误。

– 空白的 TRY/CATCH

– 使用.EQUALS() 而不是 '=='

– 未使用的参数和导入

…

官方指导

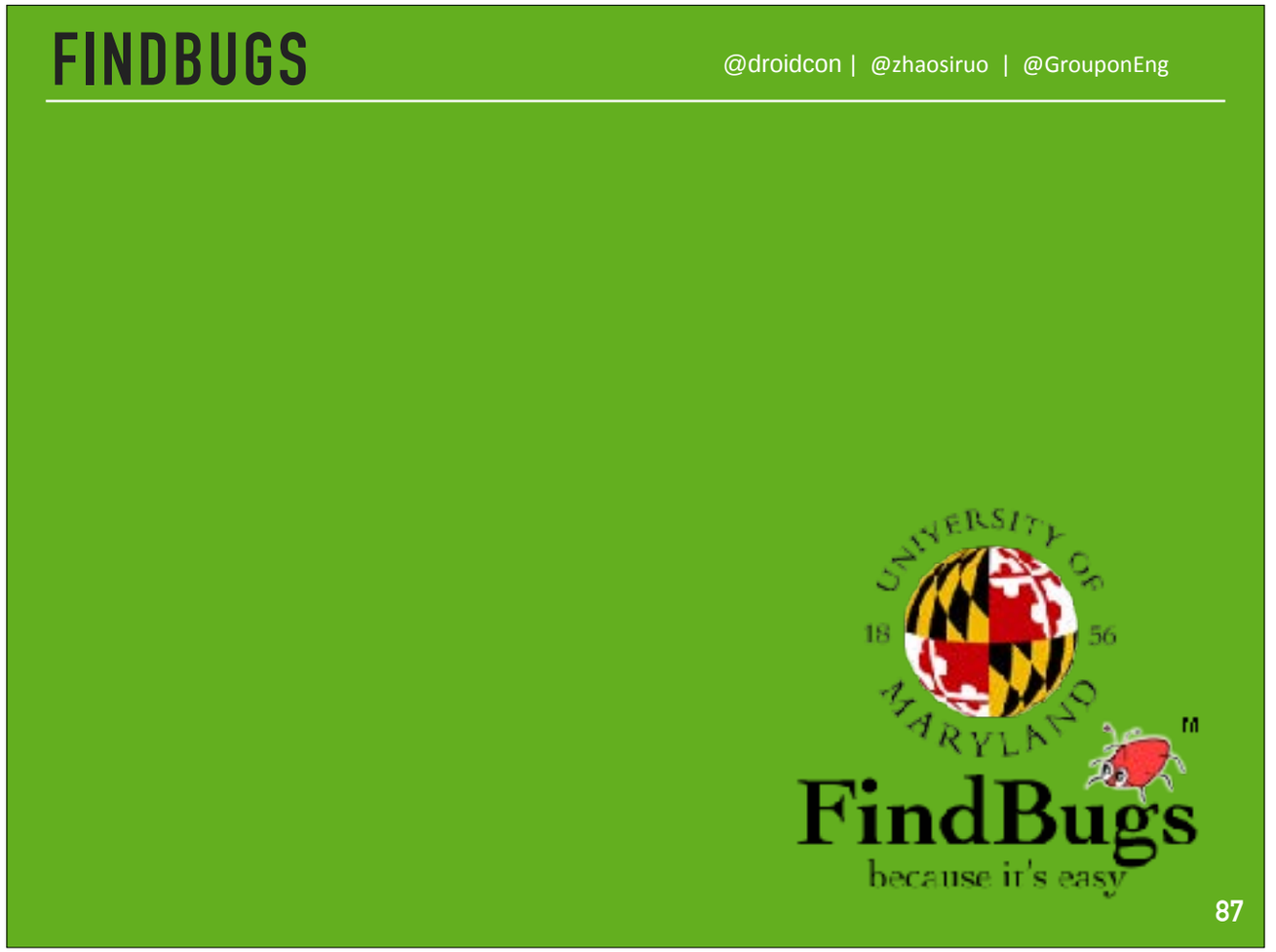
PMD 于2002问世。一个小趣闻就是它并没有一个官方全称，有的工程师就戏称它为（Pretty Much Done）（快要完成了）或者 Project Meets Deadline（按时交工）。听起来最为正经的一个是 Programming Mistake Detector（程序错误检测）。PMD负责检索源代码中的常见错误，比如 空白的 try/catch blocks, 使用.equals() instead of '==', 和人见人爱的未使用参数和导入。

PMD 非常强大，它能够覆盖包括 Java, JavaScript, XML等的8种语言。另外，PMD也是利用AST来检索java文件的。

FINDBUGS

FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?

因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。



FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?

因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。

FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?

因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。

检索字节码中的常见错误。
– 不可能的类型转化

FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?
因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。

FINDBUGS

检索字节码中的常见错误。

– 不可能的类型转化

– 自我包含的集合

FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?

因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。

FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?

因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。

FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?

因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。

FindBugs是2006年问世的Java静态代码分析工具。从名字来看Findbugs和PMD功能相似，那么为什么我们会同时使用呢?

因为Findbugs的工作原理在于检索字节码而不是源代码。所以它能找到和PMD不同的问题，比如不可能的类型转化，自我包含的集合，和无限循环等等。

	语言	可配置度	用途

	语言	可配置度	用途
CheckStyle			

	语言	可配置度	用途
CheckStyle	Java		

	语言	可配置度	用途
CheckStyle	Java		

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD			

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs			

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs	Java		

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs	Java		

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs	Java		字节码纠错

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs	Java		字节码纠错
Lint			

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs	Java		字节码纠错
Lint	Java, XML, Groovy/Gradle, Kotlin		

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs	Java		字节码纠错
Lint	Java, XML, Groovy/Gradle, Kotlin		

	语言	可配置度	用途
CheckStyle	Java		Java 风格规范
PMD	Java, XML		源代码纠错
FindBugs	Java		字节码纠错
Lint	Java, XML, Groovy/Gradle, Kotlin	 ???	

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML)

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑

89

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

	语言	可配置度	用途
Lint	Java, XML, Groovy/Gradle, Kotlin		风格规范 / 纠错 – 安卓特色 (JAVA, XML) – GRADLE – KOTLIN – 商业逻辑 – 程式库测试 ...

另外，就在两周前，谷歌宣布Lint的建议用途不应该止于安卓。 所以我相信，凭借和IDE的无缝对接，和谷歌的不懈支持，Lint将会在未来肩负起越来越重要的责任，而掌握Lint也一定会帮助我们成为更加优秀的谷歌开发者。

LINT

你的360全方位贴身保镖

https://github.com/siruozhao/android-custom-lint


@zhaosiruo


+zhao siruo


snow赵思若

你，准备好试水了没?

你，准备好试水了没?

GROUPON

永远向优秀的工程师敞开大门：

jobs.groupon.com/careers/

最后说一下，Groupon永远会向优秀的工程师敞开大门，所以欢迎随时联系我了解细节。

问答

问题？

建议？

资源

LINT OFFICIAL GUIDE

OFFICIAL SAMPLE CUSTOM LINT RULE

UAST API

PSI API

CHECKSTYLE OFFICIAL GUIDE

PMD OFFICIAL GUIDE

FINDBUGS OFFICIAL GUIDE

HOW TO CREATE ANDROID LIBRARY

LINT & CI

资源

- LINT OFFICIAL GUIDE
- OFFICIAL SAMPLE CUSTOM LINT RULE
- UAST API
- PSI API
- CHECKSTYLE OFFICIAL GUIDE
- PMD OFFICIAL GUIDE
- FINDBUGS OFFICIAL GUIDE
- HOW TO CREATE ANDROID LIBRARY
- JAVA POET
- LINT & CI