GitHook的执行的脚本位于项目.git/hooks文件夹下，默认未开启的脚本以.smaple结尾，如果要激活某个脚本，只需要将后缀删除即可。

这些能够执行的脚本分为两类，一种是在本地执行另一种是在服务器执行。

**本地执行（有序执行）：**

pre-commit：commit时最先执行，一般用来进行代码检测，可被git commit --no-verify命令绕过

prepare-commit-msg：一般用来生成模板commit，需要将生成的模板写入.git/COMMIT\_EDITMSG文件，输入的内容会替换git commit –m”备注”命令中输入的备注。

commit-msg：一般用来检测最终的commit，但此时仍然可以修改COMMIT\_EDITMSG文件

post-commit：commit成功后进行触发，一般用作通知

触发器示例：

**不允许代码量过大**

实际中还要判断文件是否是类文件，具体的类解析一般是执行一个python脚本或者运行check-style.jar来验证文件是否符合要求

.git/hooks/pre-commit

#!/bin/bash

diffFile=$(git diff --name-only)  
for file in $diffFile  
do  
 line="$(awk '{print NR}' $file|tail -n1)"  
 if [ $line -gt 500 ]  
 then  
 echo "$file 代码行数超过500行"  
 exit 1  
 fi  
done  
exit 0

checkstyle.jar检测：

1.下载checkstyle.jar <https://github.com/checkstyle/checkstyle/releases>

2.配置检测规则：

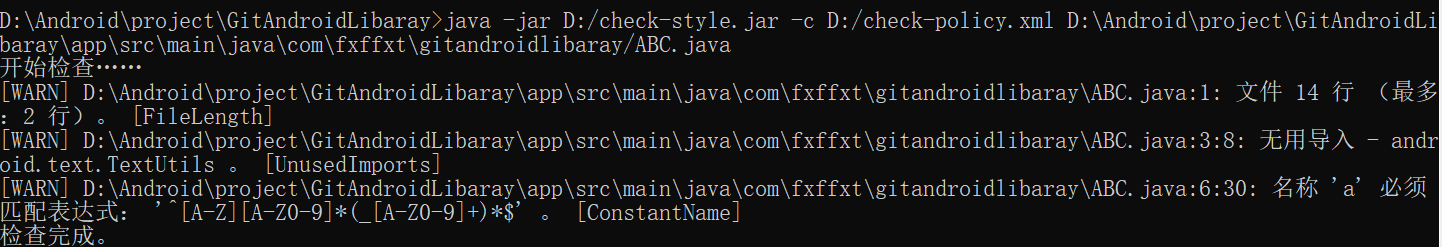
Checkstyle的检测规则是一个xml配置文档，可以根据需求进行配置（https://checkstyle.org/config.html）

<?xml version="1.0"?><!DOCTYPE module PUBLIC  
 "-//Puppy Crawl//DTD Check Configuration 1.3//EN"  
 "http://www.puppycrawl.com/dtds/configuration\_1\_3.dtd">  
  
<module name="Checker">  
 <property name="charset" value="utf-8" />  
 <property name="severity" value="warning" />  
 <property name="fileExtensions" value="java" />  
 <!—限制文件行数-->  
 <module name="FileLength">  
 <property name="max" value="2" />  
 </module>  
 <module name="TreeWalker">  
 <!-- 避免.\*,重复多余的和不使用的import-->  
 <module name="AvoidStarImport" />  
 <!-- 避免无效的import-->  
 <module name="UnusedImports" />  
 <!-- 常量全部用大写-->  
 <module name="ConstantName" />  
 <!-- 方法名称 -->  
 <module name="MethodName" />  
 </module>  
  
</module>

3.使用以下命令即可对文件进行检测：

java -jar checkstyle.jar -c 规则文件 待检测文件

检测结果样式：



可根据输出的结果来判断是否通过本次commit

如果内置的规则无法满足，则需要自定规则，比如MethodName规则中就无法排除对native方法的检测，需要自定义（待更新）

CheckStyle添加自定义规则：

首先需要了解DetailAST对象，DetailAST是一个类似于树的结构，该对象保存了整个Java文件的详细信息，可通过java -jar D:/check-style.jar –t xxx.java 的方式查看java文件信息

1. 定义CustomCommentCheck继承AbstractCheck，如果是对已有的规则进行加强，可以直接继承已有的规则，规则名+Check为具体规则实现类。
2. 确定要接收的类型：TokenTypes类中有类型常量

@Override  
public int[] getDefaultTokens() {//默认接收的类型  
 return getRequiredTokens();  
}  
  
@Override  
public int[] getAcceptableTokens() {  
 return getRequiredTokens();  
}  
  
@Override  
public int[] getRequiredTokens() {  
 return new int[]{TokenTypes.*SINGLE\_LINE\_COMMENT*};//接收单行注释  
}

@Override  
public boolean isCommentNodesRequired() {//开启注释接收[只在进行注释规则检测时使用]  
 return true;  
}

4．实现规则判断

@Override  
public void visitToken(DetailAST ast) {  
 DetailAST detailAST = ast.findFirstToken(TokenTypes.*COMMENT\_CONTENT*);  
 if (detailAST != null) {  
 String comment = detailAST.getText();  
 if (comment.length()>maxLength){  
 log(ast, "单行注释超过"+maxLength);  
 }  
 }  
}

以上实现了对单行注释(//)长度的限制

5．添加可通过外部配置的属性

CustomCommentCheck添加一个maxLength属性和对应的set方法

然后就可以在规则文件中使用了

<module name="com.example.check.CustomCheck">  
 <property name="maxLength" value="2"/>  
</module>

内部通过反射将property的值设置给CustomCheck

6．执行

将自定义的代码打包成myCheckStyle.jar

命令行运行：java –classpath ./checkstyle.jar;./myCheckStyle.jar com.puppycrawl.tools.checkstyle.Main –c ./checkstyle.xml ./JavaClassForCheck.java

7．还可以添加CustomChcek描述文件（非必要），可直接参考checkstyle.jar的内部实现

**组装commit模板**

可以对commit信息进行修饰，比如创建一个固定的commit格式

.git/hooks/prepare-commit-msg

#!/bin/bash  
echo "前缀 $(cat $1) 后缀" > $1  
exit 0

**commit信息进行检测**

对应一些不合适的commit信息可以驳回

./git/hooks/commit-msg

#!/bin/bash  
msg=$(cat $1)  
if [ ${#msg} -lt 8 ]  
then  
 echo "commit信息太短小于8个字符"  
 exit 1  
fi  
exit 0

补丁应用hook（用途待研究）

patch指某两次提交之间的差异，存储的是对代码的修改

applypathc-msg：接收单个参数， 包含请求合并信息的临时文件的名字，在执行git am xxx.patch后调用。可通过.git/rebase-apply/final-commit文件得到补丁的commit信息，同时可修改该文件的内容，产生新的commit提交信息，此时产生的提交不会走commit hook流程。

pre-applypatch：运行于应用补丁之后，产生提交之前，可以用它在提交前检查快照。

post-applypatch：用于通知，此时无法暂停工作流。

其他客户端hook（用途待研究）

pre-rebase：在变基操作之前执行，可以通过返回非零值来中止变基操作，。

post-merge：执行于git merge操作之后。

post-checkout：在进行切换分支或创建新分支时触发（git checkout）。可以根据项目环境来调整工作目录。 比如放入大文件、自动生成文档或进行其他操作。

pre-push：在git push操作成功前运行，可中止push流程。

服务端执行：

pre-receive：在服务器接收到更新推送时触发，可以检测推送信息是否合法，通过返回非零阻止推送更新到git仓库。

update：在分支更新时触发，如果一次提交有多个分支需要更新，会每个分支各执行一次，可中止流程。

post-update：整个更新结束后触发，无法中止流程。

**kotlin的代码检测**

checkstyle只能进行java代码的检测，对kotlin的检测需要ktlint插件。

需要先下载ktlint插件

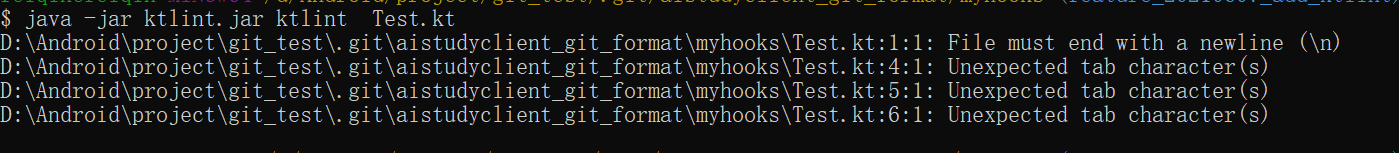
Ktlint本身支持对代码格式的检测，如操作符两边要空格、不能有连续的换行、类名必须和文件名一致、以换行结束文件等。

要想用ktlint对kotlin文件进行检测，只需要一行命令

Java –jar ktlint路径 ktlint 被检测文件路径1,被检测文件路径2

如果想进行多个文件的检测，可以用逗号分隔

注意：被检测文件的路径只能是相对路径，不能取绝对路径



检测结果

Ktlint支持配置规则文件.editorconfig，通过在命令行后面添加—editorconfig 配置文件路径 的方式设置自己的规则集合。editorconfig支持配置的规则是有限的，ktlint的默认规则都是开启的，如果需要禁用某个规则，需要通过命令的方式禁用规则，如：

--disabled\_rules= final-newline, max-line-length

禁用多个规则也是用逗号分隔，至于规则的名称，可以去ktlint源码里看，或者查看最后整理的规则列表。

Ktlint支持的规则有限，如果需要其他规则，就需要自定义。

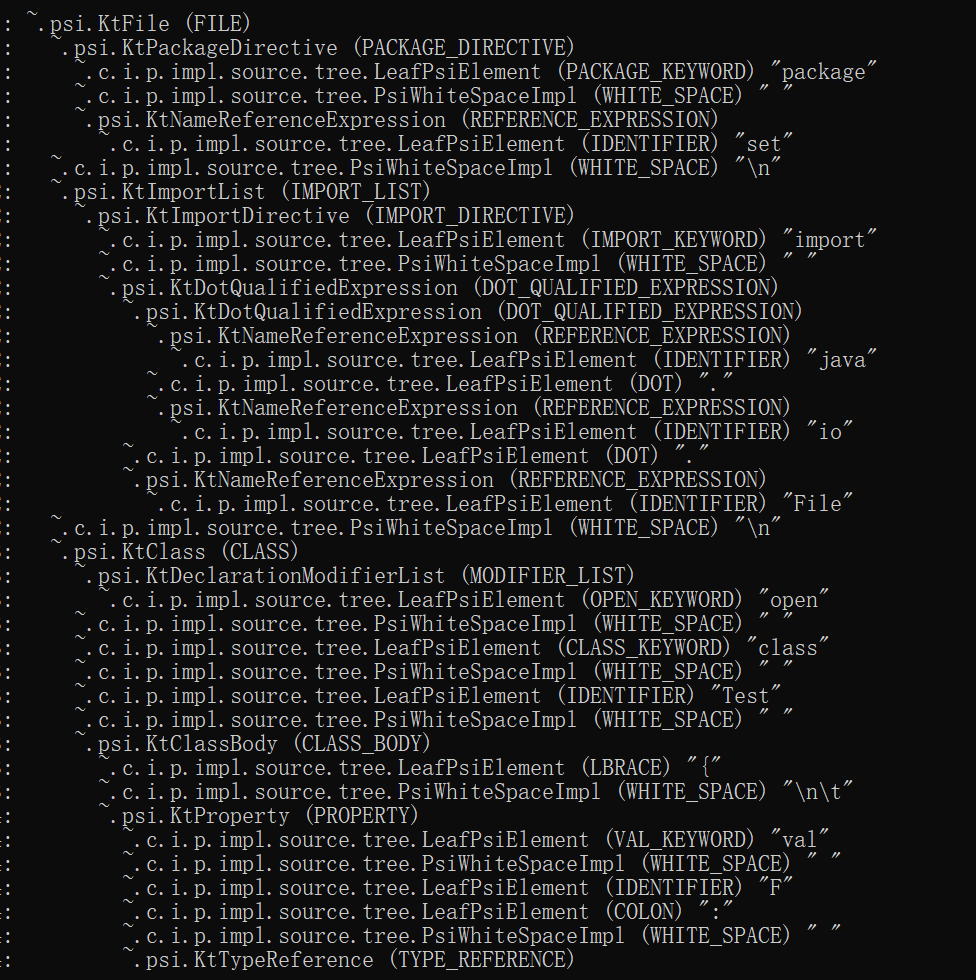
自定义步骤：

1. 创建java或者kotlin工程、依赖ktlint
2. 创建规则类继承Rule类，需要填写id和实现visit方法
3. 在visit方法中实现自己的规则逻辑
4. 创建规则provider类继承RuleSetProvider，设置provider的id和包含的自定义规则
5. 在META-INF文件夹下创建services/com.pinterest.ktlint.core.RuleSetProvider纯文本文件，里面的内容为自定义provider的全量类名
6. 打包成jar
7. 在正常检测的命令中添加 –R my-rules.jar 命令，运行命令即可添加自定义的检测规则
8. 禁用自定义规则，和禁用默认规则一样，但是得加上自定义规则provider的id前缀：disabled\_rules=my-rules:method-name-rule,my-rules:property-name-rule

自定义的核心是Rule 的visit方法，要自定义规则得了解ktlint检测的原理。

和checkstyle检测的原理一样，ktlint也是将kotlin代码进行分解，最终组成一颗树，查看分解树的结构可以用以下命令完成

Java –jar ktlint路径 ktlint 检测文件 –print-ast



检测的结构

树的跟节点是一个FILE点，所有的kotlin文件都有这个节点，往下就是按照代码的顺序进行扫描，显示扫描到包名产生就产生了PACKAGE\_DIRECTIVE节点，这个节点下就包含了包名信息。继续扫描，扫描到同级的import，就产生了IMPORT\_LIST节点，IMPORT\_LIST节点下包含了import信息。然后又扫描产生了同级的CLASS节点，内部包含class的详细信息。从CLASS节点往内部扫描的话，就会产生属性节点（PROPERTY），方法节点FUN。

树上的所有节点都是一个ASTNode，ASTNode的一些重要方法有：

getElementType 获取当前节点的类型

findChildByType 根据传入的类型获取子节点

getText 当前节点所包含的内容（如果当前节点是FILE，那么结果就是整个kt文件的纯文本）

getStartOffset 当前节点在整个文本中的位置，如果检测到文件的代码不合规，这个方法可提示用户需要在哪里修改

ASTNode的类型在ElementType类里面能找到

一个简单的例子，禁止Long类型变量使用小写的L：

verride fun onVisit(node: ASTNode, autoCorrect: Boolean, emit: (offset: Int, errorMessage: String, canBeAutoCorrected: Boolean) -> Unit) {

//找到int或者long数字  
 if (node.*elementType* == ElementType.INTEGER\_LITERAL){  
 if (node.*text*.*endsWith*("l")){//判断是否以小写L结尾  
 emit(node.*startOffset*,"use 'L' instead of 'l'", false)  
 }  
 }  
}

如果要制定其他的规则，如方法名检测，变量名检测，都可以按照这个方法来制定。

总结：

需要ktlint.jar

自定义规则要继承Rule

使规则能够生效，需要继承RuleSetProvider

使规则能够被ktlint.jar 扫描到需要在MTEA-INF/services文件夹下添加能被加载的RuleSetProvider的全限定名

调用java命令即可完成对特定文件的检测：

java –jar ktlint.jar ktlint MyKt.kt –R myRules.jar –editorconfig my\_editorconfig\_path –disabled\_rules=final\_newline

对代码的检测可以配置到本地也可以配置到git服务器，配置到本地和配置check style的方式一模一样。配置到服务器则还需要一步额外的操作：生成提交文件。

git服务器仓库和本地仓库的区别是，本地有对应的代码文件，而服务器只有本地的.git目录里面的内容，并没有直接存储完整的文件，要得到对应版本的完整文件，需要使用git命令先得到提交的文件列表：

git log -1 –name-only –pretty=format: <version\_hash>

然后再使用命令得到文件内容：

git show <version\_hash> :file\_name\_you\_want

再将得到的内容写入到一个文件中

最后才能通过检测命令检测对应文件的代码规范，检测完成之后需要记得删除。

流程注意事项：

脚本用python写需要加上声明 #!/usr/bin/python

出现中文需要添加声明 # -\*- coding:utf-8 -\*-

Python popen执行命令需要有COMSPEC环境变量指向cmd.exe文件（windows电脑测试需要，其他不知）

最终的hook部署时，是使用的脚本部署，脚本会检测当前位置，如果正则继续，会判断是否安装Java环境，如果没有就自动安装.

服务端的java环境需要写死（目前服务器存在java环境但是运行时java命令无法使用，需要java的绝对路径），安装脚本在安装时会对代码中需要的路径进行替换。