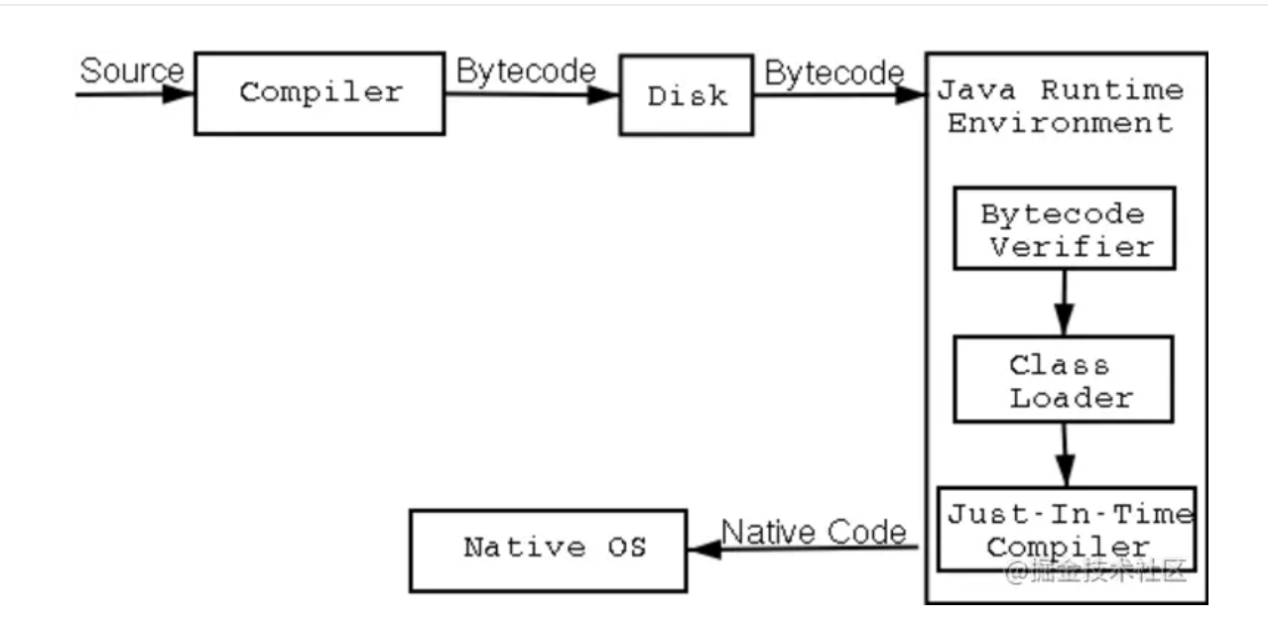
**什么是Processor**

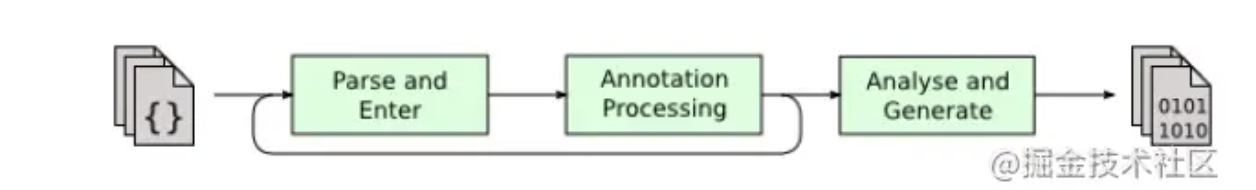
AbstractProcessor是一个抽象类，它的父类是Processer。

Processor会在编译阶段初始化，然后对当前模块内的代码进行一次扫描，然后获取到对应的注解，之后调用process方法，然后根据这些注解类来做一些后续操作。

**java的编译流程**



上图是一张简单的编译流程图，compiler代表我们的javac。这张图其实缺少了一个流程，在source -> complier的过程中就应该把我们的Processor补充上去。



把两张图结合就是整个java的编译流程了。整个编译过程就是 source(源代码) -> processor（处理器） -> generate （文件生成）-> javacompiler -> .class文件 -> .dex(只针对安卓)。

先从接口的方法介绍起把。

| **变量和类型** | **方法** | **描述** |
| --- | --- | --- |
| Iterable<? extends Completion> | getCompletions​(Element element, AnnotationMirror annotation, ExecutableElement member, String userText) | 返回一个空的迭代完成。 |
| Set<String> | getSupportedAnnotationTypes() | 如果处理器类使用SupportedAnnotationTypes进行批注，则返回与注释具有相同字符串集的不可修改集。 |
| Set<String> | getSupportedOptions() | 如果处理器类使用SupportedOptions进行批注，则返回具有与批注相同的字符串集的不可修改集。 |
| SourceVersion | getSupportedSourceVersion() | 如果处理器类使用SupportedSourceVersion进行批注，请在批注中返回源版本。 |
| void | init​(ProcessingEnvironment processingEnv) | 通过将 processingEnv字段设置为 processingEnv参数的值，使用处理环境初始化处理器。 |
| boolean | process​(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) | 处理源自前一轮的类型元素的一组注释类型，并返回此处理器是否声明了这些注释类型。 如果返回true ，则声明注释类型，并且不会要求后续处理器处理它们; 如果返回false ，则注释类型无人认领，可能会要求后续处理器处理它们。 处理器可以总是返回相同的布尔值，或者可以根据其自己选择的标准改变结果。 |

## ProcessingEnvironment

这个类会在函数init的时候被传入，主要的工具类方法都在这个类上。

public interface ProcessingEnvironment {

Map<String, String> getOptions();

Messager getMessager();

Filer getFiler();

Elements getElementUtils();

Types getTypeUtils();

SourceVersion getSourceVersion();

Locale getLocale();

}

Filer 就是文件流输出路径，当我们用AbstractProcess生成一个java类的时候，我们需要保存在Filer指定的目录下。

Messager 输出日志工具，需要输出一些日志相关的时候我们就要使用这个了。

Elements 获取元素信息的工具，比如说一些类信息继承关系等。

Types 类型相关的工具类，processor java代码不同的是，当process执行的时候，class的由于类并没有被传递出来，所以大部分都行都是用element来代替了，所以很多类型比较等等的就会转化成type相关的进行比较了。

类型相关的都被转化成了一个叫TypeMirror，其getKind方法返回类型信息，其中包含了基础类型以及引用类型。

private Elements elementUtils;

private Types types;

@Override

public synchronized void init(ProcessingEnvironment processingEnv) {

super.init(processingEnv);

types = processingEnv.getTypeUtils();

elementUtils = processingEnv.getElementUtils();

}

public boolean isSubType(Element element, String className) {

return element != null && isSubType(element.asType(), className);

}

public TypeMirror typeMirror(String className) {

return typeElement(className).asType();

}

public TypeElement typeElement(String className) {

return elementUtils.getTypeElement(className);

}

public boolean isSubType(TypeMirror type, String className) {

return type != null && types.isSubtype(type, typeMirror(className));

}

其中isSubType方法是判断传入的Element是不是一个接口的实现类。首先我们要将对象都转化成Element, 然后将两个element转化成TypeMirror，之后调用Types的isSubtype方法对两个TypeMirror进行比较，如果发现类型一样，则该输入的Element是特定接口的实现类。

**process**

扫描代码的时候会把当前获取到的annotations传入当前方法。一个模块的processer可能会有多个。其中process方法返回值返回true ，则声明注释类型，并且不会要求后续处理器处理它们; 如果返回false ，则注释类型无人认领，可能会要求后续处理器处理它们。

@Override

public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

if (CollectionUtils.isNotEmpty(annotations)) {

initRouter(moduleName, roundEnv);

return true;

}

return false;

}

private void initRouter(String name, RoundEnvironment roundEnv) {

Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(BindRouter.class);

if (elements.isEmpty()) {

return;

}

//生成一个init 的final 静态方法

MethodSpec.Builder initMethod = MethodSpec.methodBuilder("init")

.addModifiers(Modifier.PUBLIC, Modifier.FINAL, Modifier.STATIC);

TypeMirror type\_Activity = elementUtils.getTypeElement(Const.ACTIVITY).asType();

TypeMirror fragmentTm = elementUtils.getTypeElement(Const.FRAGMENT).asType();

TypeMirror callBackTm = elementUtils.getTypeElement(Const.RUNNABLE).asType();

//一、收集信息

int count = 0;

//遍历Element元素

for (Element element : elements) {

//检查element类型

//field type

BindRouter router = element.getAnnotation(BindRouter.class);

ClassName className;

if (element.getKind() == ElementKind.CLASS) {

className = ClassName.get((TypeElement) element);

} else if (element.getKind() == ElementKind.METHOD) {

className = ClassName.get((TypeElement) element.getEnclosingElement());

} else {

throw new IllegalArgumentException("unknow type");

}

// 一个页面可以注册多个路由

String[] id = router.urls();

for (String format : id) {

int weight = router.weight();

TypeMirror type = element.asType();

//判断是不是一个方法

if (types.isSubtype(type, callBackTm)) {

String callbackName = "callBack" + count;

initMethod.addStatement(className + " " + callbackName + "=new " + className + "()");

CodeBlock interceptorBlock = buildInterceptors(getInterceptors(router));

initMethod.addStatement("com.kronos.router.Router.map($S,$L$L)", format, callbackName, interceptorBlock);

count++;

continue;

}

if (weight > 0) {

String bundleName = "bundle" + count;

initMethod.addStatement("android.os.Bundle " + bundleName + "=new android.os.Bundle();");

String optionsName = "options" + count;

initMethod.addStatement("com.kronos.router.model.RouterOptions " + optionsName + "=new com.kronos.router.model.RouterOptions("

+ bundleName + ")");

initMethod.addStatement(optionsName + ".setWeight(" + weight + ")");

CodeBlock interceptorBlock = buildInterceptors(getInterceptors(router));

initMethod.addStatement("com.kronos.router.Router.map($S,$T.class," + optionsName + "$L)",

format, className, interceptorBlock);

} else {

CodeBlock interceptorBlock = buildInterceptors(getInterceptors(router));

initMethod.addStatement("com.kronos.router.Router.map($S,$T.class,$L)", format, className, interceptorBlock);

}

}

count++;

}

//将方法插入到一个特定包名的类名下

String moduleName = "RouterInit\_" + name;

TypeSpec routerMapping = TypeSpec.classBuilder(moduleName)

.addModifiers(Modifier.PUBLIC, Modifier.FINAL)

.addMethod(initMethod.build())

.build();

try {

//生成java文件

JavaFile.builder("com.kronos.router.init", routerMapping)

.build()

.writeTo(filer);

} catch (IOException ignored) {

}

}

获取所有的注解，然后根据把注解上的url以及类名，拦截器等信息收集起来，通过javapoet将这些信息注册到一个注册类上。

其中判断了下注解类的类型，如果是一个函数的话，则调用函数注册的方式，如果是activity的话则调用另外一个注册规则。

apt能做的事情还是比较有限的

1. javapoet只能新增一个类，而不能对当前类进行更改。
2. proessor在javac执行之前，所以只能对当前moudule生效
3. 当Module一多，可能会有类名冲突的问题

但是apt还是能帮助我们解决很多问题的，我们可以把一些机械化的操作，通过anntation的方式去简化，比如butterknife，这样开发就可以有更多的精力去专注做写别的事情。一部分abtest赋值的操作其实也可以用同样的方式去调整。