注解生成代码与Analyzer Ast解析生成代码

Analyzer_x插件使用

analyzer_x插件使用方法

方法一：在主项目下执行dart run build_runner build --delete-conflicting-outputs

方法二：在analyzer_x(插件目录下)下执行dart run build_runner build --delete-conflicting-outputs

方法三：运行插件analyzer_x下lib/test/main.dart文件

原理简介

一、注解生成代码

1. 概述

flutter禁用了反射功能，因此无法直接获取对象本身的语法信息（属性名、构造器等），而通过为一个类添加注解的方式，在编译期间我们可以得到一个类的基本语法信息，并可利用这些信息构造代码生成器。

2. 获取范围

注解生成代码依赖于第三方库'source_gen'，在注解之下的类，我们能够得到如下有效信息：

（1）注解类中的所有信息

（2）类中有在Element中声明的语法节点

（3）构建位置信息

针对（2），以如下图片进行展示,在Element中提供了visitChildren用以遍历一个element的所有子节点，从图中可以看出element访问的信息比较有效，只是当前注解对象的直接子节点：

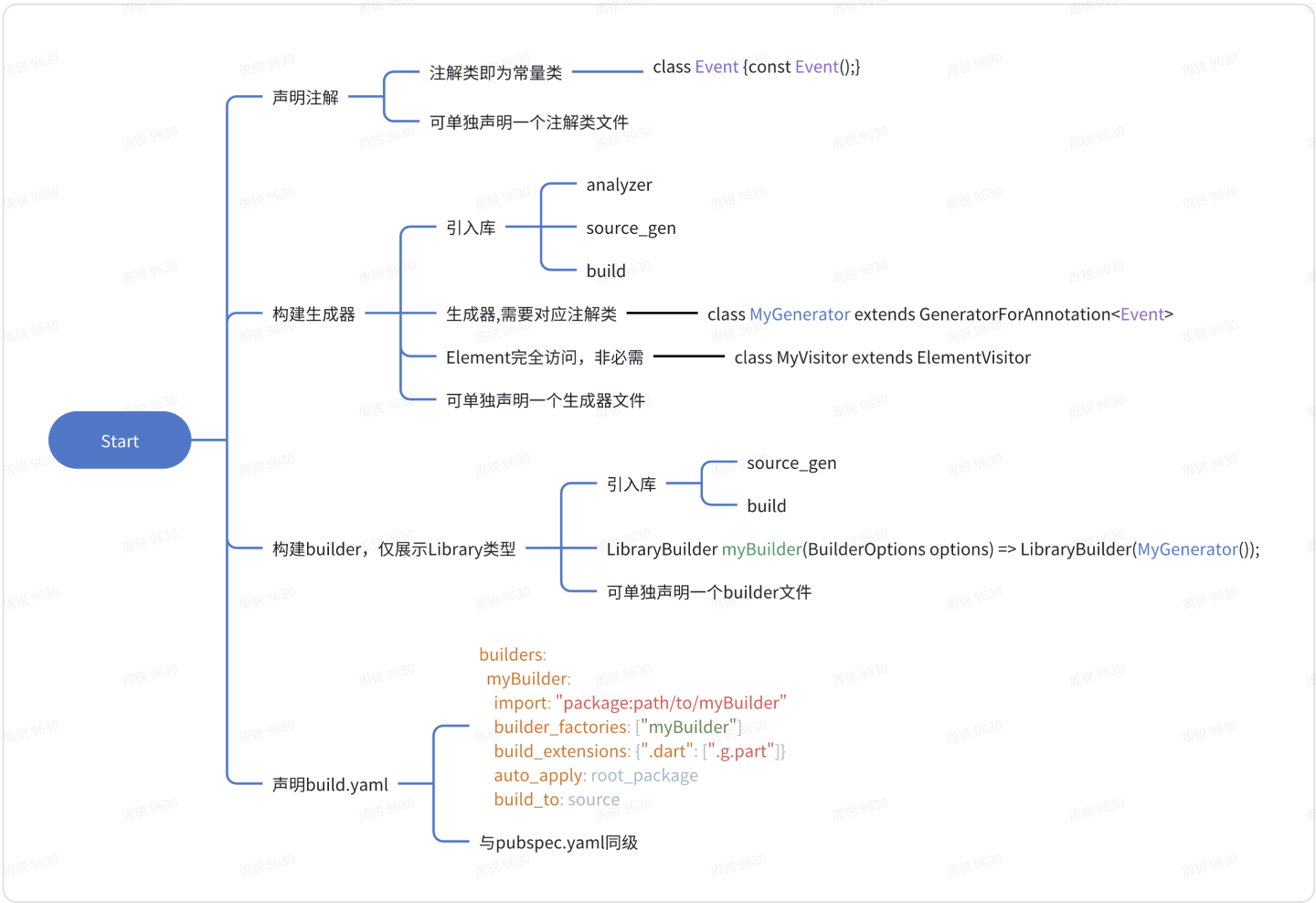
```dart
@EventABC()
class AfPurchaseCoinSuccesszEvent extends BaseEvent {
  AfPurchaseCoinSuccesszEvent({
    required this.priceUsdParam,
  }) : super(name: 'purchase_coin_success');

  final PriceUsdParam priceUsdParam;

  @override
  List<BaseParam> get preDefineParams => [
        priceUsdParam,
      ];

  @override
  List<EventPanel> get deliverPanels => [
        EventPanel.appEvent,
        EventPanel.xlog,
        EventPanel.grpc,
        EventPanel.mixpanel,
        EventPanel.firebase,
        EventPanel.appsflyer,
      ];
}
```

注解类

```dart
// GENERATED CODE - DO NOT MODIFY BY HAND

// **************************************************************************
// MYGenerator
// **************************************************************************

//AfPurchaseCoinSuccesszEvent  AfPurchaseCoinSuccesszEvent   CLASS
//priceUsdParam  priceUsdParam   GETTER
//preDefineParams  preDefineParams   GETTER
//deliverPanels  deliverPanels   GETTER
//priceUsdParam  priceUsdParam   FIELD
//preDefineParams  preDefineParams   FIELD
//deliverPanels  deliverPanels   FIELD
//AfPurchaseCoinSuccesszEvent     CONSTRUCTOR
//priceUsdParam  priceUsdParam   PARAMETER
```

遍历可获取的所有类信息

3. 快速生成



二、Analyzer Ast解析

1. 概述

通过第三方库'analyzer'，我们可以实现针对指定dart文件的Ast（抽象语法树）解析，以此获取一个文件中所有的语法信息。与注解生成代码相比，analyzer具有更强大的信息获取能力、自由度更高，但没有与build_runner相结合，原本需要自定义输入路径、输出路径、库引入信息以及执行入口。但在此处可以与注解生成代码相结合，通过添加注解的方式，让该方法生成代码在build命令下可执行。

2. 获取范围

（1）指定文件中的所有语法信息

具体包含信息，可以查看'analyzer'库下lib/dart/ast/ast.dart文件，其中给出了每一个语法节点的文法定义，如下图所示：

```
/// An as expression.
///
///    asExpression ::=
///        [Expression] 'as' [TypeAnnotation]
///
/// Clients may not extend, implement or mix-in this class.
abstract class AsExpression implements Expression {...}
```

as表达式定义

3. 遍历Ast所有节点

（1）Ast是一棵双向树，因此可以从任一节点出发访问整个Ast。

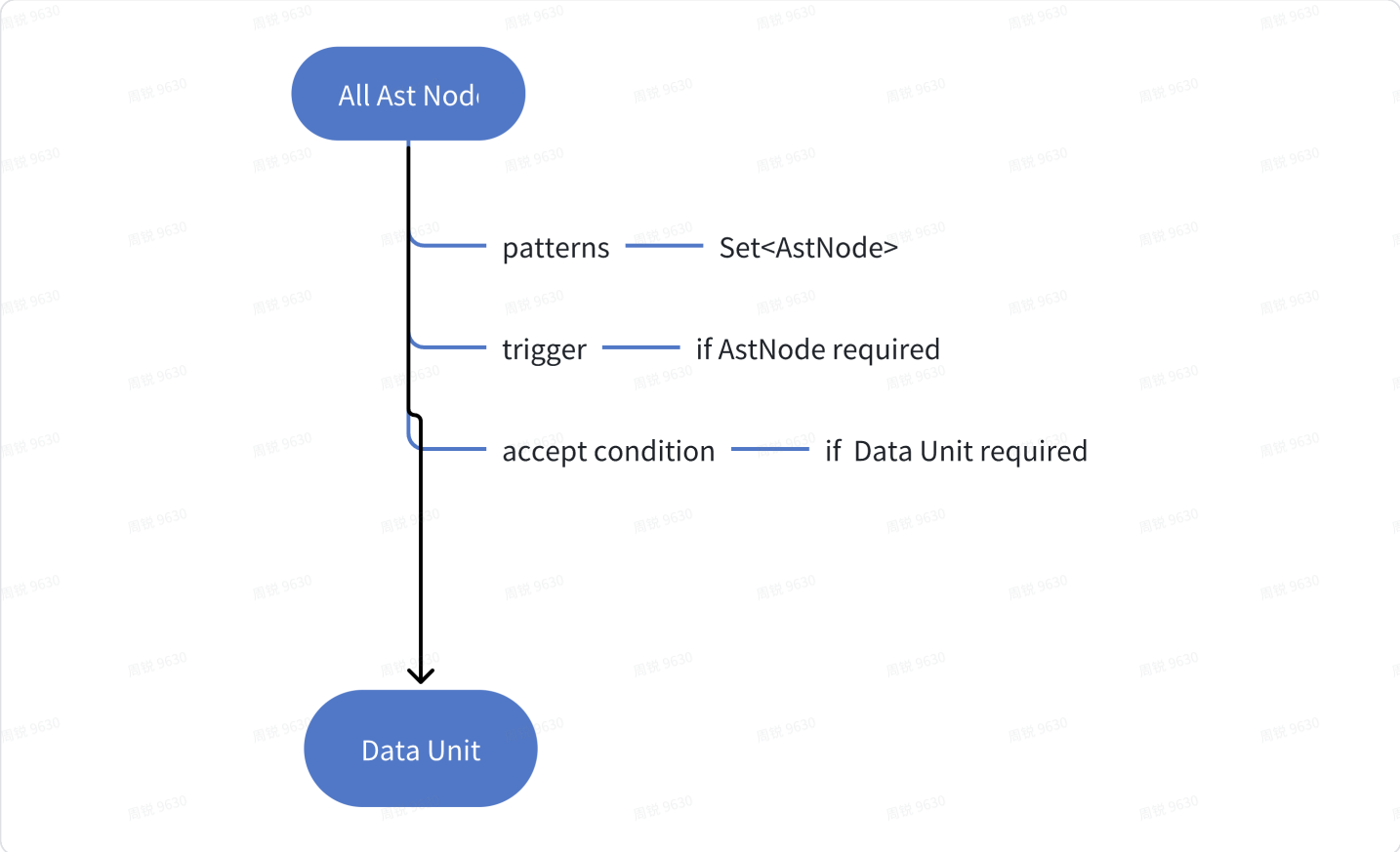
（2）对Ast的遍历通过对analyzer提供的AstVisitor进行覆写实现，遍历是一个深度优先的过程。

基于以上两点，提出了如下的访问策略：

（1）patterns: 一系列需要访问的AstNode的集合，当遍历到的节点在其中时，进入下一步过滤

（2）trigger: 针对单个符合patterns的AstNode，判断该节点是否符合要求

（3）accept condition: 经历步骤二，节点可被表达为Map{AstNode: if required}，通过各个AstNode的需求情况，判断这些的AstNode是否能构造出符合需求的数据单元



管道过滤获取需求数据

4. 自寻输入文件/输出文件/import路径

当已知需求数据所对应的语法结构时，我们便可以自动找寻相应的输入文件；同时，可以将输出文件路径指定为获取需求数据最多的一个输入文件；针对import，则需要单独构建用于获取包信息相关语法结构的Getter。

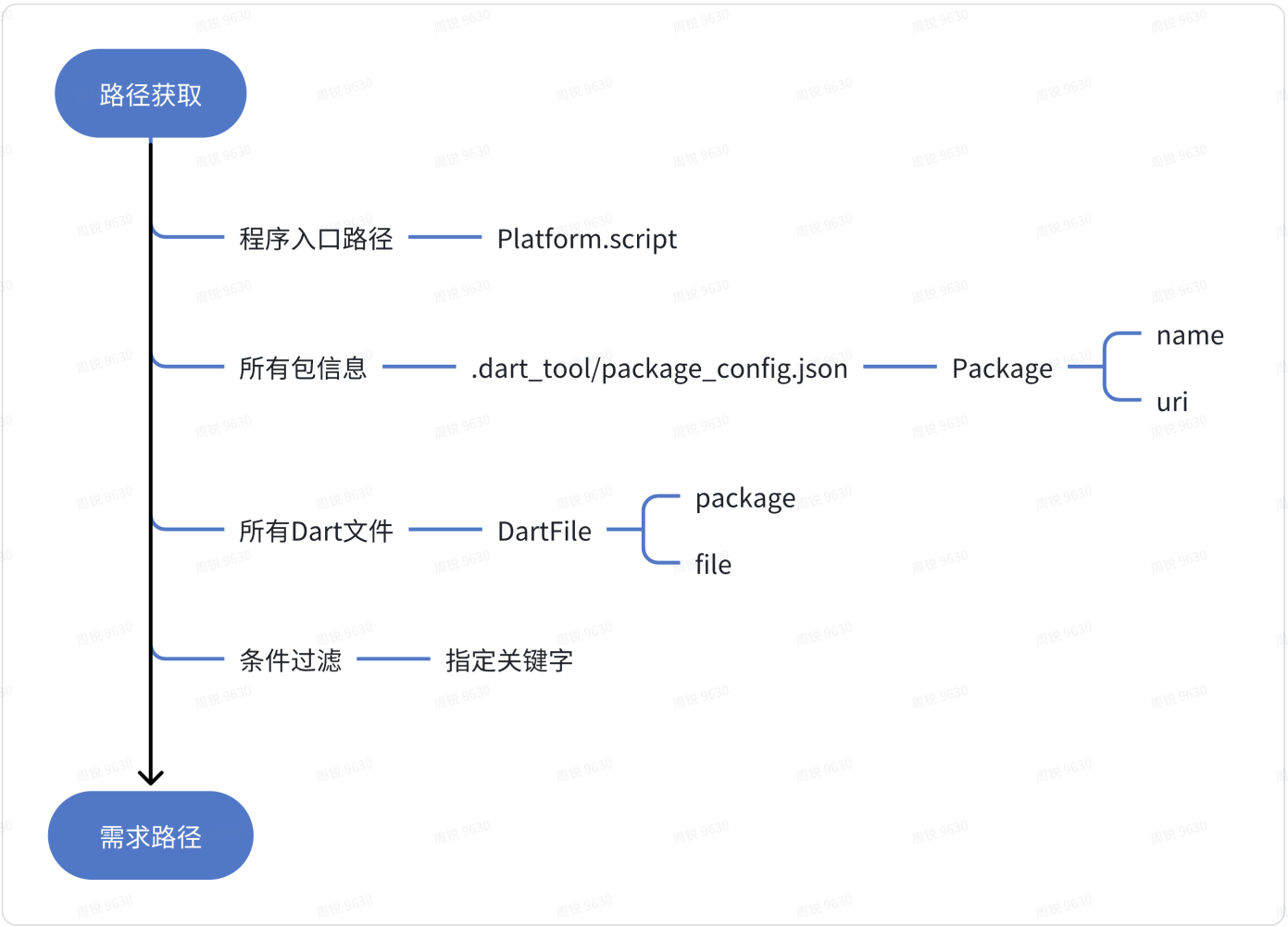
那么，路径获取可以通过以下步骤被逐步确定：

（1）程序入口路径：指main函数所在文件路径

（2）所有包信息：通过项目文件package_config可以获取到主项目所有的package依赖

（3）所有Dart文件：在各个包中找寻扩展为'.dart'的文件

（4）条件过滤：通过一些关键字，提前过滤掉不会出现需求数据的dart文件



路径自动获取

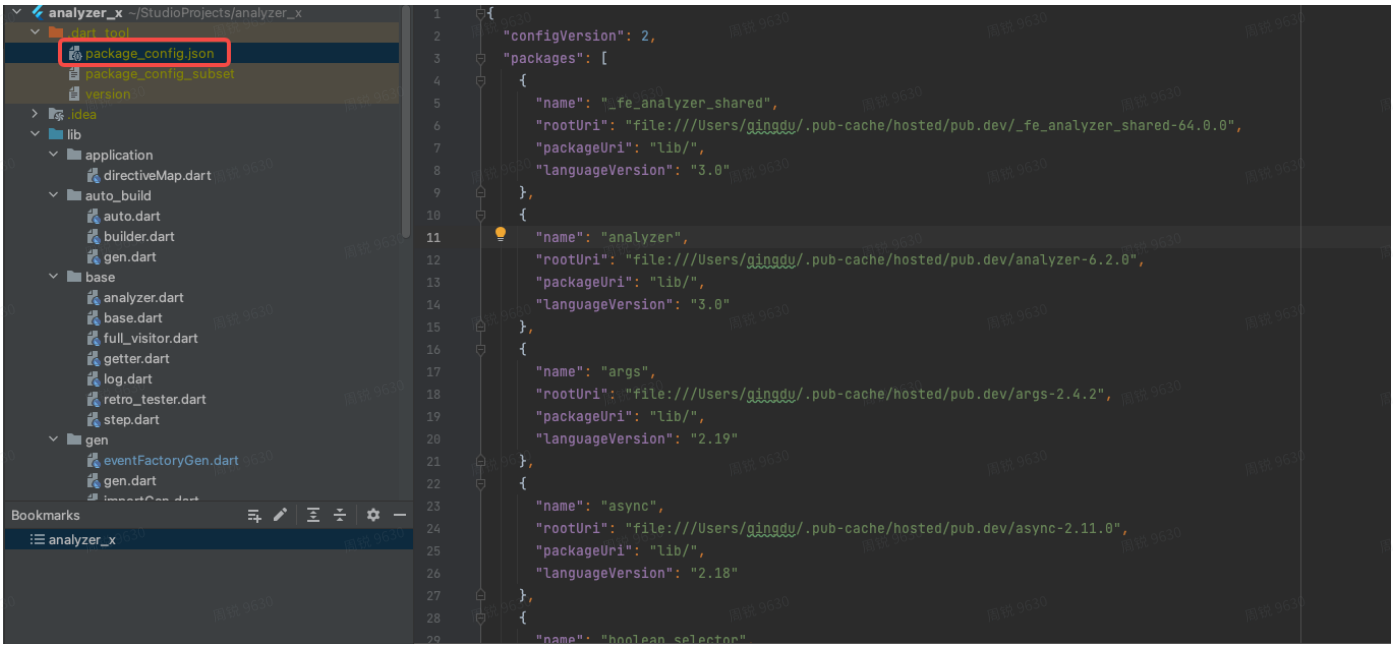
插件简介

在实际应用过程中，路径查询、数据获取与代码生成实际为三个相互独立的工作。

一、路径查询(lib/path)

1. 获取包信息

主要操作为获取主项目中文件*package_config.json，解析其中的包内容*



package_config.json

2. 获取所有包中dart文件

递归遍历包中lib文件夹下的所有文件，找到所有dart文件并存储

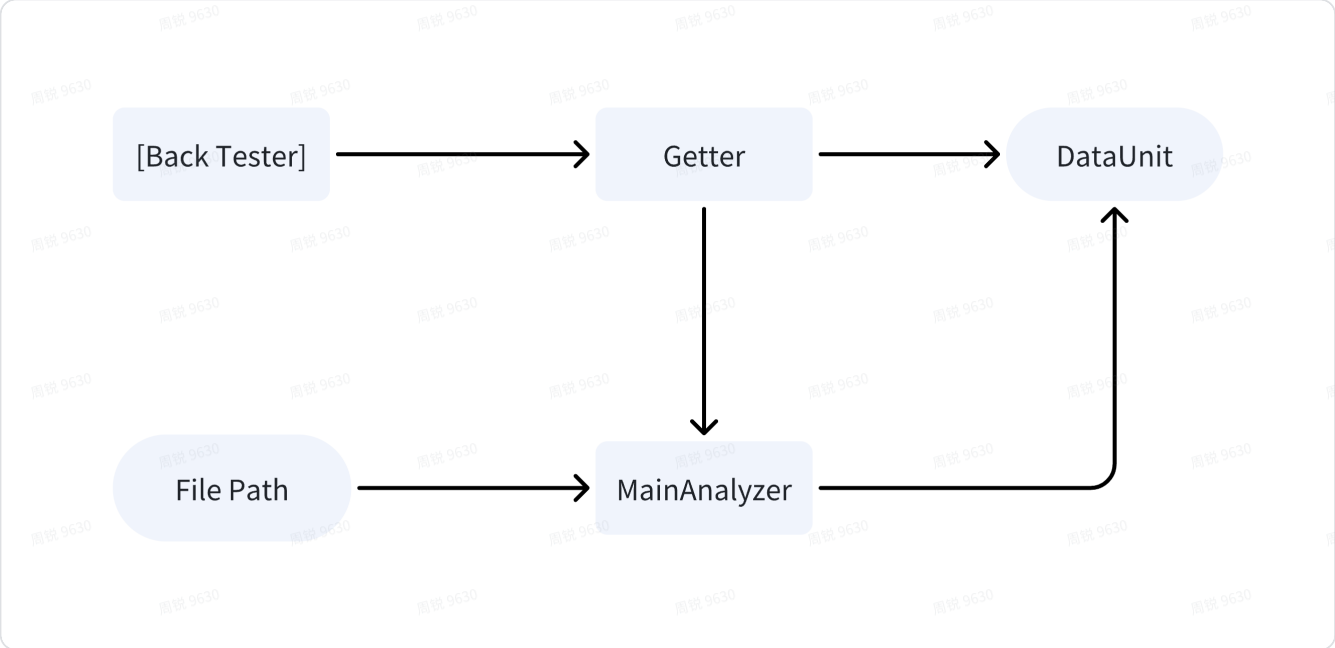
```dart
///获取主进程中的所有dart文件
List<DartFile> getDartFiles({bool Function(String fileString)? isTarget}) {
  if (_files.isEmpty) {
    PackageConfig config = PackageConfig.fromProj();
    for (var package in config.packages) {
      if (!filterExternPackage || !package.isAbsolutePath) {
        _getDartFile(package, package.absolutePackagePath, _files);
      }
    }
  }
  return _files
      .where((e) => isTarget?.call(File(e.filePath).readAsStringSync()) ?? true)
      .toList();
}

///定义一个dart文件
class DartFile {
  final Package package;
  final String filePath;

  String get importName {
    return 'package:${filePath.replaceAll(package.absolutePackagePath, package.name + |
  }

  DartFile(this.package, this.filePath);
}
```

获取dart文件与dart file定义

二、数据获取

1. 抽象层（lib/base）

抽象层定义了从文本中获取数据的完整流程，但以抽象类的方式进行表达，其架构如下图所示：



数据获取抽象

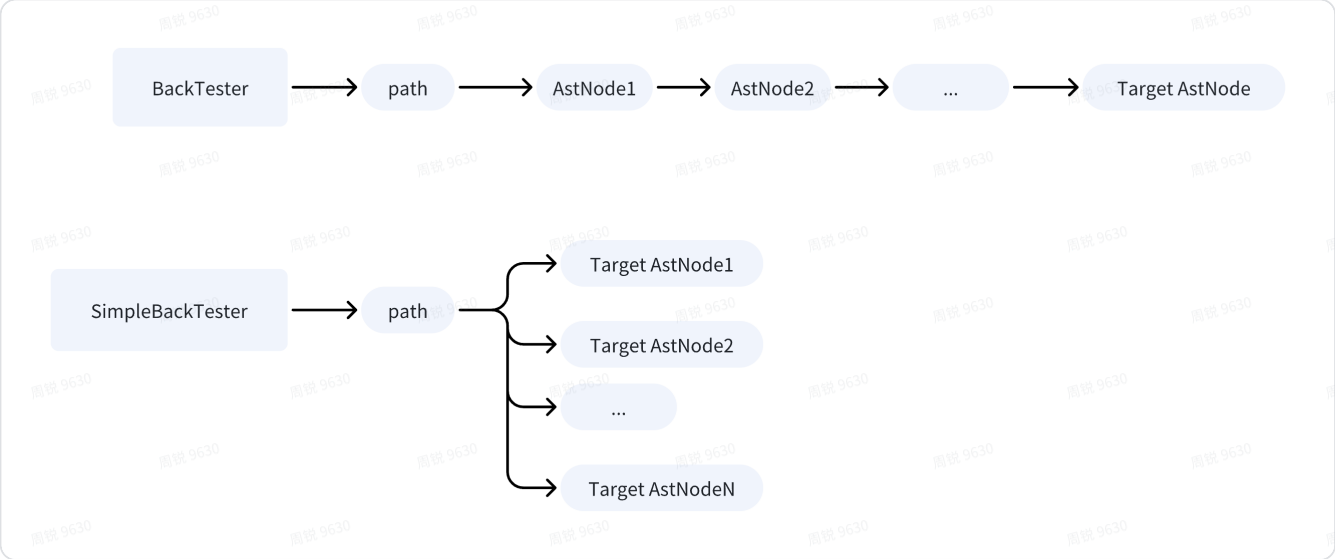
2. Back Tester

为了获取dart文件中的语法信息是否包含我们所需的结构，我们可以构建一条访问路径，如果在遍历过程中有此条路径，则通过RetroTester存储该路径的叶子结点（AstNode）。因为AstNode是双向树，我们可以通过一个节点遍历整棵树，所以仅存储叶子结点即可。于此同时，Retro Tester提供了多个方法便于回溯找寻需求节点。

获取的需求信息并非所有的都是需要经历一条路径，例如，我们可能获取'part'与'part of'的语法节点集合，此时收集的数据呈现为多个单结点。为应对此类情况，在RetroTester基础上构造一个SimpleRetroTester抽象类。

1. PartDirective： *[Annotation] part [StringLiteral] ;*
2. *PartOfDirective：[Annotation] part of [Identifier] ;*

上述两种情况可以表示为如下两种状态机，每一个Tester被设计为只接受一条路径或者只接受多个单节点的集合。



接受AstNode的不同形式

BackTester和SimpleBackTester都是用于获取我们指定的语法节点，指定在path中完成，这需要对analyzer中语法节点的定义有一定了解。两者的区别在于SimpleBackTester是存储不同类型AstNode的集合，当语法树遍历到该节点时，便会将其加入；BackTester是仅存储path的首结点(在语法树上则靠下)，同时需要将存储类型进行声明——当语法树上找到这样一条路径之后，你需要调用accept方法判断是否接受，接受则将其存储下来，否则丢弃。

构建一个SimpleBackTester：

```
///获取库/文件引用信息
class DirectiveBTester extends SimpleBackTester {
  @override
  List<AnalyzerStep> get path => [
        AnalyzerStep.importDirective,
        AnalyzerStep.libraryDirective,
        AnalyzerStep.partDirective,
        AnalyzerStep.partOfDirective,
        AnalyzerStep.exportDirective,
      ];
}
```

SimpleBackTester定义

构建一个BackTester：

```dart
///获取一个继承[superClass]的构造器超集中[labelName]的[SimpleStringLiteral]取值
class SuperNameBTester extends BackTester<SimpleStringLiteral> {
  final String superClass;
  final String labelName;

  SuperNameBTester({
    required this.superClass,
    required this.labelName,
  });

  @override
  bool accept(node) {
    ClassDeclaration classDeclaration = backNode<ClassDeclaration>(node);
    ExtendsClause? extendsClause = classDeclaration.extendsClause;
    NamedExpression namedExpression = backNode<NamedExpression>(node);
    Label label = namedExpression.name;
    if (extendsClause == null ||
        extendsClause.superclass.name2.toString() != superClass) {
      return false;
    }
    if (label.label.name != labelName) {
      return false;
    }
    return true;
  }

  ///状态机为:
  ///[ClassDeclaration], [ExtendsClause]=>
  ///[ConstructorDeclaration]=>
  ///[SuperConstructorInvocation]=>
  ///[ArgumentList]=>
  ///[NamedExpression], [Label]=>
  ///[SimpleStringLiteral]
  @override
  List<AnalyzerStep> get path => [
        AnalyzerStep.simpleStringLiteral,
        AnalyzerStep.namedExpression,
        AnalyzerStep.argumentList,
        AnalyzerStep.superConstructorInvocation,
        AnalyzerStep.constructorDeclaration,
        AnalyzerStep.classDeclaration,
      ];
}
```

BackTester定义

3. Getter

getter是获取需求数据的核心类，它从Back Tester得到的节点中获取需要的数据信息并保存起来。简单来说,此处用于实现数据的转换。也因此，需要在Getter中引用BackTester，并在reset方法中实现数据存储。

```dart
///获取一个dart文件中的[Class]|||[Enum]|||[Function] 声明
class DeclarationGetter extends Getter {
  DeclarationUnit unit = DeclarationUnit();

  @override
  void reset() {
    unit.dClass.addAll(getClasses());
    unit.dFunction.addAll(getFunctions());
    unit.dEnum.addAll(getEnums());
  }

  @override
  List<BackTester<AstNode>> testers = [
    DeclarationBTester(),
  ];

  List<String> getClasses() => tester<DeclarationBTester>()
      .tList<ClassDeclaration>()
      .map((e) => e.name.toString())
      .toList();

  List<String> getFunctions() => tester<DeclarationBTester>()
      .tList<FunctionDeclaration>()
      .map((e) => e.name.toString())
      .toList();

  List<String> getEnums() => tester<DeclarationBTester>()
      .tList<EnumDeclaration>()
      .map((e) => e.name.toString())
      .toList();
}

class DeclarationUnit {
  List<String> dClass = [];
  List<String> dFunction = [];
  List<String> dEnum = [];
}
```

实现对class/funcion/enum声明中类名称的获取

同时，Getter提供testerAccept方法用于判断某一个BackTester是否已经接受了某一结点，接受需要两个条件：（1）在路径上/集合内 （2）BackTester的accept方法返回true。

如下图所示：EventGetter用于获取一个继承了BaseEvent类，且声明label包含'name'的结构（要求1）。当SuperNameBTester接受时，说明符合了要求1，这时再从各AstNode之中获取我们需要的数据，组成数据单元。

```dart
///获取一个Class类型的数据信息，该类需满足一下条件
///(1)继承自类[BaseEvent]
///(2)构造器中的super中应包含[name]字段
///(3)[name]字段类型为简单字符串
class EventGetter extends Getter {
  List<EventUnit> units = [];

  @override
  void reset() {
    if (testerAccept<SuperNameBTester>()) {
      units.add(EventUnit(
        className: className,
        eventName: eventName,
        classParameters: classParameters,
        classParameterQuestions: classParameterQuestions,
        constructorParameters: constructorParameters,
      ));
    }
  }

  @override
  List<BackTester> testers = [
    SuperNameBTester(
      superClass: 'BaseEvent',
      labelName: 'name',
    ),
    ClassParametersBTester(),
    ConstructorParametersBTester(),
  ];
```

4. MainAnalyzer

MainAnalyzer是文件分析与与数据获取的入口，每次调用会分析指定路径的文件或指定内容的文本，配合第一部分的路径查询，可以自动得到需要分析的路径/文本；除此之外，需要传入定义好的Getter（允许复数个），用于获取指定文件/文本中是否有符合要求的数据。

```dart
Map<DartFile, List<EventUnit>> parseEvent() {
  List<DartFile> inputFilePath = getDartFiles(isTarget: EventGetter.mayTarget);
  Map<DartFile, List<EventUnit>> unitsMap = {};

  for (var file in inputFilePath) {
    var getter = EventGetter();
    MainAnalyzer(getters: [getter], filePath: file.filePath);
    unitsMap[file] = getter.units;
  }
  return unitsMap;
}
```

通过MainAnalyzer获取数据

三、代码生成（lib/gen）

通过第二部分得到的数据，我们可以生成需求的代码，目前生成需求尚不复杂，所以直接书写在dart中，因为在第一部分获取了各个Dart文件的路径信息，因此，输出路径也是容易指定的。除此之外，调用了DartFormatter帮助代码格式化，自写方法实现了一个并不完善的自动import。

```dart
void execute() {
  analyzerLog(DateTime.now());
  analyzerLog('parse event');
  Map<DartFile, List<EventUnit>> units = parseEvent();
  assert(units.values.expand((e) => e).toList().isNotEmpty);
  ///生成代码
  String fileString = classBlock(
    methods: [
      fromMethod(
        units.values.expand((e) => e).toList(),
      ),
      whereTypeOneMethod(),
      whereTypeOneOrNullMethod(),
      onEventCheckFailMethod(),
    ],
  );
  ///自动引入
  analyzerLog(DateTime.now());
  analyzerLog('auto import');
  importFiles.clear();
  importFiles.addAll(ImportGen.instance.analyse(fileString));
  fileString = dartFile(header: header(), classBlock: [fileString]);
  ///代码格式化
  DartFormatter formatter = DartFormatter(indent: 0);
  String code = formatter.format(fileString);
  ///设置生成[EventUnit]最多的文件为输出文件
  DartFile? outFile;
  int count = 0;
  units.forEach((key, value) {
    if (value.length > count) {
      outFile = key;
      count = value.length;
    }
  });
  File(outFile!.filePath.replaceAll('.dart', '.g.dart')).writeAsString(code);
  analyzerLog(DateTime.now());
  analyzerLog(
      'EventFactoryGen: ${outFile!.importName.replaceAll('.dart', '.g.dart')}');
}
```