**静态分析工具设计文档**



**撰写者：南京大学 吴晓阳**

**南京大学 陈昊东**

**南京大学 施超烜**

**南京大学 程珂**

**南京大学 周若衡**

目录

[1. 引言 3](#_Toc32547)

[1.1. 编写目的 3](#_Toc20833)

[1.2. 预期读者 3](#_Toc85)

[1.3. 术语表 3](#_Toc9571)

[1.3.1. CSA Clang Static Analyer 3](#_Toc30628)

[2. 总体设计 3](#_Toc30478)

[2.1. 开发人员 3](#_Toc9980)

[2.2. 开发环境和工具 4](#_Toc6819)

[2.3. 基本设计描述 4](#_Toc19342)

[2.4. 输入命令设计 4](#_Toc4761)

[2.5. 输出类型设计 5](#_Toc30655)

[2.6. 用例图 5](#_Toc620)

[3. 模块设计 6](#_Toc14646)

[3.1. 静态分析模块 6](#_Toc16598)

[3.1.1. 整体框架 6](#_Toc21680)

[3.1.2. Checker设计 9](#_Toc19677)

[3.2. 命令行解析模块 15](#_Toc27989)

[3.2.1. 功能介绍 15](#_Toc17506)

[3.2.2. 代码结构和处理流程 15](#_Toc23436)

[3.3. 统计模块 16](#_Toc12306)

[3.3.1. 功能介绍 16](#_Toc22438)

[3.3.2. 相关接口和功能 17](#_Toc22499)

[3.4. 输出模块 18](#_Toc32490)

[3.4.1. 功能介绍 18](#_Toc11405)

[3.4.2. 类图 19](#_Toc11697)

[3.4.3. 相关接口和功能 19](#_Toc30097)

[3.5. Json模块 19](#_Toc16312)

[3.5.1. 功能介绍 19](#_Toc12506)

[3.5.2. 类图 20](#_Toc4065)

[3.5.3. 核心算法 22](#_Toc14326)

[3.6. GUI模块 24](#_Toc6888)

[3.6.1. 功能介绍 24](#_Toc8306)

[3.6.2. 代码结构和处理流程 25](#_Toc1245)

[4. 模块交互 26](#_Toc14321)

[4.1. 核心模块 26](#_Toc32615)

[4.2. GUI模块 26](#_Toc16865)

# 引言

## 编写目的

本静态分析工具基于Clang CSA架构进行开发，在开发过程中，会涉及到与CSA 相关接口的交互、命令行的解析、输出逻辑和输出格式、checker的设计、gui设计等相关设计需求。本设计文档为程序模块提供相关设计方案和模块交互方案，以在实际设计过程、后期维护过程中提供程序结构的参考和借鉴。

本设计文档的编写目的是说明程序模块的设计方案，包括程序描述、输入/输出、算法和流程逻辑等，为软件编程和软件维护提供基础。

## 预期读者

本说明书的预期读者为软件设计人员、软件开发人员、软件测试人员。

## 术语表

### CSA Clang Static Analyer

Clang静态代码分析器是clang原生的代码分析框架，其使用符号执行技术进行路径敏感的代码分析，采用挂载在引擎上的Checker对程序状态进行检查并报错。

# 总体设计

## 开发人员

吴晓阳、陈昊东、程珂、周若衡、施超烜：负责核心相关部分开发和测试

程珂、周若衡：负责GUI部分的设计和实现

吴晓阳：负责网站和在线分析工具的实现和维护。

## 开发环境和工具

|  |  |
| --- | --- |
| 工具 | 作用 |
| Clang | 基于CSA进行静态分析 |
| QT | 进行GUI开发 |
| VisualStudio | 在Windows环境下进行开发和部署 |

## 基本设计描述

1. 能够通过命令形式获取用户输入文件、输入参数、输出参数等数据，进行相应代码分析，并将分析结果呈现在对应的输出界面上。
2. 能够通过Shell、GUI与用户进行交互，方便用户使用和操作。
3. 提供官网进行软件信息的查看和软件下载

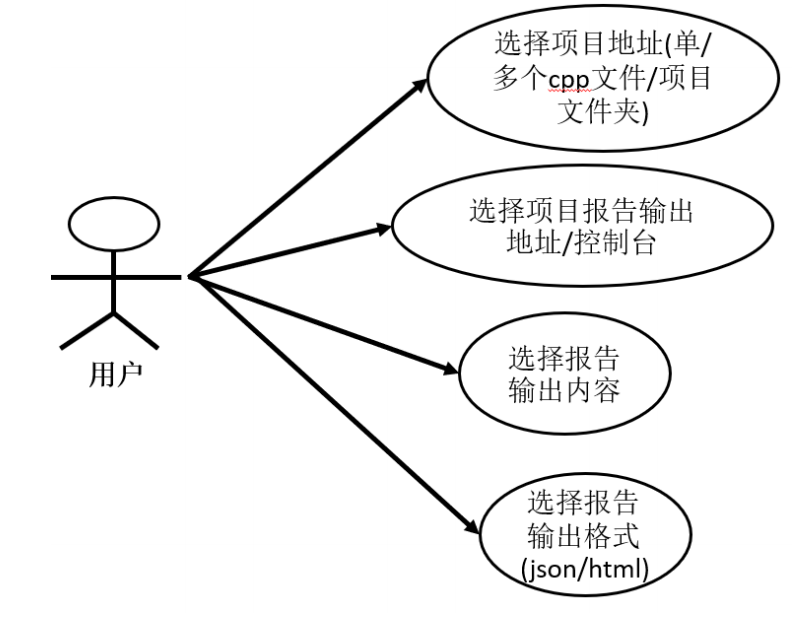
## 输入命令设计

|  |  |  |
| --- | --- | --- |
| 命令 | 参数 | 描述 |
| -help |  | 打印帮助信息 |
| -g |  | 开启god mode |
| -debug |  | 开启debug mode |
| -input | File1, file2... | 制定输入文件 |
| -compilation-database-path | Path | 指定compile\_commands.json路径 |
| -output --path | Path | 指定输出路径 |
| -output --format | Format1,format2... | 指定输出格式【json/console/text/html】 |
| -output --info | Opt1, opt2... | 指定输出包含的信息【time/checker/desc/stats】，  若指定time，则默认添加stats |
| -enable-checker | Checker1,checker2... | 指定开启的checker |
| -checker-list |  | 列举支持的checker列表 |

## 输出类型设计

|  |  |
| --- | --- |
| 输出类型 | 解释 |
| HTML | 以HTML方式输出到文件夹中，供网页生成和在线浏览。 |
| Json | 以Json格式输出到文件夹中，供用户利用以进行二次开发 |
| MD/text/Console | 以markdown/文本/命令行形式输出，供用户对缺陷分析结果进行查看。 |

## 用例图



## 模块设计

## 静态分析模块

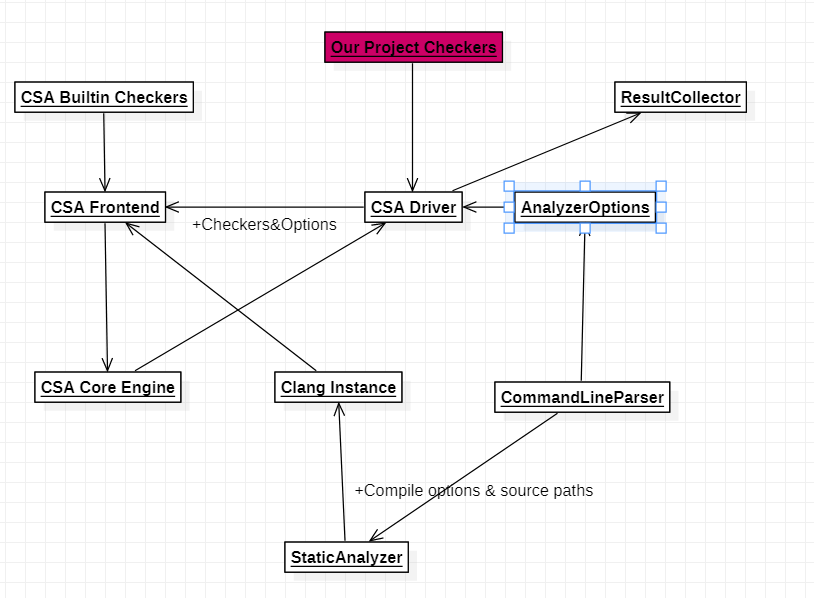
### 整体框架

#### 功能介绍

静态分析模块用于调用CSA相关api进行开发、注册相应checker、运行静态分析算法等功能。

#### 代码结构和处理流程

##### 模块关联



##### 各模块功能

###### StaticAnalyzer

StaticAnalyzer负责完成整个缺陷检测的驱动，具体而言，其需要：

1. 正确构造符合libToolingAPI规范的ClangTool
2. 根据用户配置，正确的传递编译选项和待处理文件给ClangTool
3. 将用户关于本软件的配置选项传递给CSADriver
4. 将准备完成的CSADriver注册为ClangTool的运行行为
5. 运行ClangTool，使软件陷入Clang运行时环境

###### CSADriver

CSADriver为本项目接口对Clang内部API的适配器，负责Checker的注册、将本软件配置注入CSA Framework和CSA符号执行引擎的驱动。

###### CSACheckerRegistry

单例类，负责注册、管理本项目Checker，并将Checker注册进CSA实例，移交管理权至CSA

###### CSAAction

负责创建CSA运行时、开启统计计时并将统计信息传给ResultCollector、将CSA的Checker管理器、注册器传递给CSACheckerRegistry并正确的启用和配置CSA执行选项

###### CSAFactory

CSAAction的工厂类，负责AnalyzerOptions的传递

###### CSADiagnostics

负责获取CSA的缺陷报告，并将其处理生成为Diagnostic传给ResultCollector

###### ResultCollector

ResultCollector作为单例类负责管理所有的缺陷报告信息及统计信息，是本框架对外输出的统一数据接口

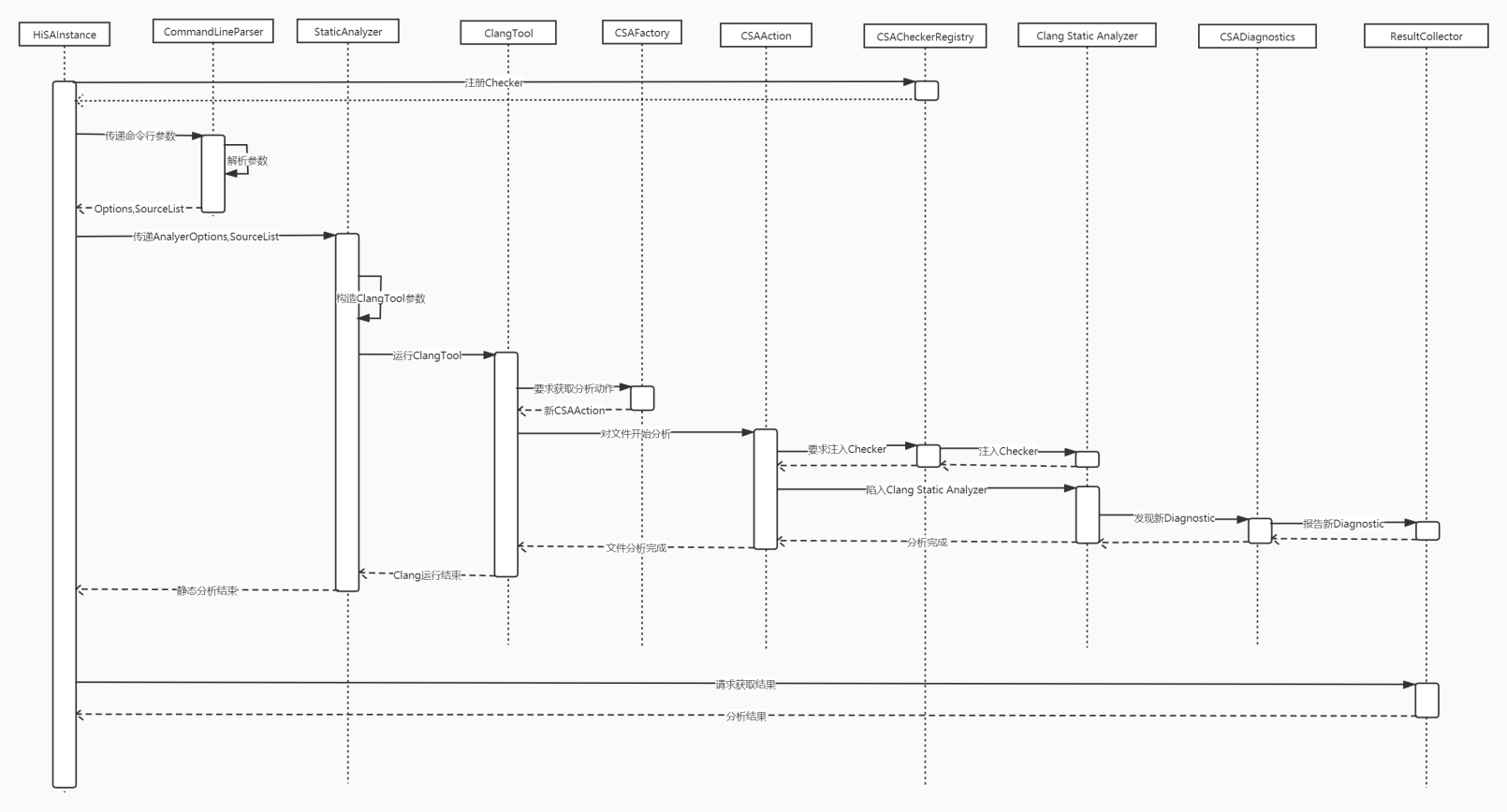
###### CommandLineParser

输入模块的主要类，负责解析命令行

###### HisaInstance

类似于ClangTool，提供统一访问接口

##### 运作序列图



### Checker设计

#### 功能介绍

该模块基于CSA Checker继承进行设计，负责对代码进行符号分析以找出并报告相应的代码缺陷。包括MemoryChecker、DeferenceChecker、DivModZeroChecker和UninitObjChecker。

#### MemoryChecker设计

##### 功能介绍

本checker用于对内存相关的缺陷进行检测，包括double delete、double free、memory leak、use after free四种缺陷。相关checker注入位置包括dead symbol、new allocator、pre call、post call、new的post statement、delete的pre statement。

##### 算法流程

###### CheckDeadSymbol

本回调对new分配完内存，调用构造函数前执行。其内部调用handle\_new()来标记分配的内存为allocated状态

###### CheckLocation

本回调在对指针使用的时候执行，若使用的指针指向的内存区域为deallocated，那么存在use-after-free缺陷

###### CheckPostCall

本回调函数在函数调用后执行，在本函数中，我们对allocation族函数和deallocation族函数进行检查，对于allocation族函数，我们调用handle\_allocation()来更新其内存状态，对于deallocation族函数，我们调用handle\_deallocation()来更新其内存状态并检测double-free缺陷

###### CheckPostStmt

本回调函数在语句执行前执行，在本函数中，对于delete\_expr我们调用handle\_deallocation()来更新其内存状态并检测double-free缺陷，对于return语句，我们认为返回deallocated的内存区域是use-after-free缺陷

###### CheckPreCall

本回调在函数调用前执行，在本函数中，我们对除deallocation函数外的所有函数的参数进行检查，若参数对应符号对应的内存区域为deallocated状态，那么存在use-after-free缺陷，若函数调用是类方法调用，那么若this指针指向的内存区域为deallocated状态，则同样存在use-after-free缺陷

###### CheckPreStmt

本回调函数在语句执行后执行，在本函数中，对于new\_expr我们调用handle\_new()来更新其内存状态

###### Handle allocation

本函数创建一个代表内存区域的符号，并令其状态为allocated，该符号是与allocation site绑定的

###### Handle deallocation

本函数将符号对应内存区域由allocated状态更新为deallocated状态，若原先状态为deallocated，则此次是重复释放，存在double-free缺陷

###### Handle new

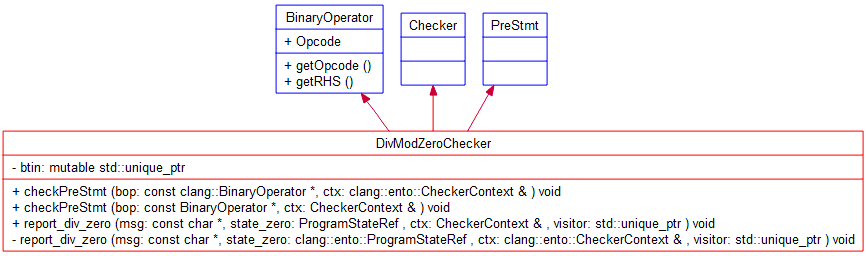
本函数用于将符号对应的内存区域状态更新为allocated，符号本身应该是一个Location

#### DivModZeroChecker设计

##### 功能介绍

该模块负责处理除0模0的缺陷。相关checker位置为location、PreStmt。

##### 类图



##### 算法流程

###### CheckPreStmt

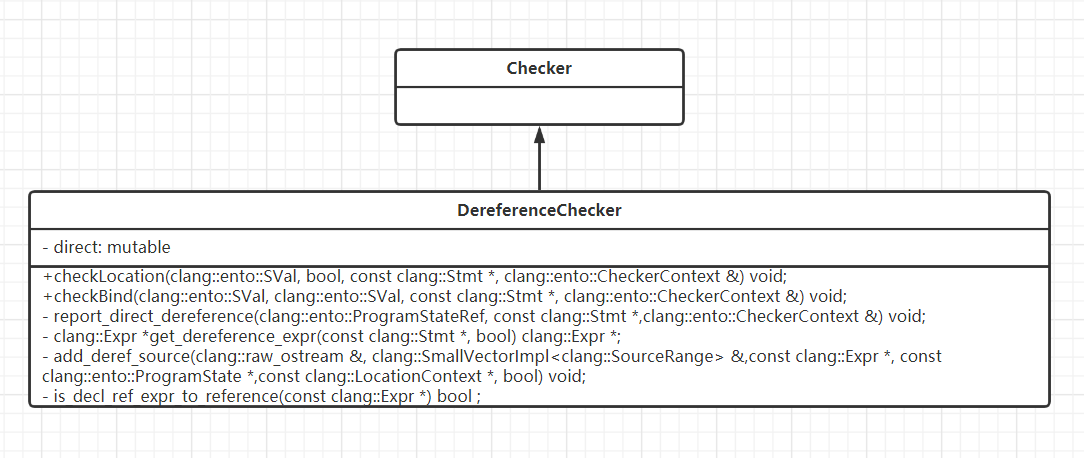
传入的是 二元运算符 bop 和相应的语句 ctx，根据**设计**中的原理对语句是否存在缺陷进行判断

#### DereferenceChecker设计

##### 功能介绍

该模块负责解引用的软件缺陷检测，相关checker注入位置包括bind、location

##### 类图



##### 算法流程

###### CheckBind

本回调函数在对符号进行绑定值的时候被调用，对于空指针解引用缺陷，只需要检查对引用变量进行初始化绑定的情况，因此需要对被绑定的符号进行类型的判断。经过判断过后，与checkLocation的情况类似，通过对进行绑定的值进行非空情况和为空情况的判断，并根据判断结果选择是否报告错误。

###### checkLocation

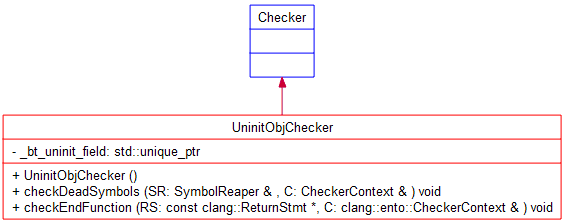
本回调函数在对指针进行读写操作时被调用，state为当前程序状态，loc为当前指针的符号，通过语句std::tie(not\_null\_state, null\_state) = state->assume(loc);调用符号执行引擎对loc所代表的值进行判断，并将非空和为空的情况之后的程序状态分别赋予not\_null\_state和null\_state，如果只存在为空情况不存在非空情况，则报告一个空指针解引用错误，如果为空情况和非空情况同时存在，则报告一个可能隐形存在的空指针解引用错误。

#### UninitObjChecker设计

##### 功能介绍

该checker负责对未初始化对象的使用进行缺陷检测

##### 类图



##### 算法流程

## 命令行解析模块

### 功能介绍

该部分负责对输入命令进行解析，获取输入数据，扇出至静态分析模块进行操作。

### 代码结构和处理流程

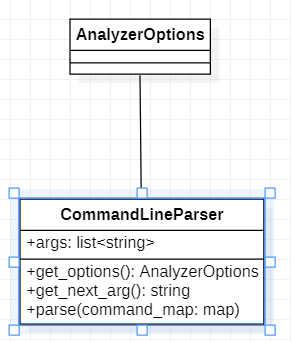
#### 核心逻辑

模块通过函数指针的形式储存所有的相关命令，并使用map进行命令名和处理函数之间的映射。在从输入中扫描到相关命令时进行回调，处理输入并保存所需数据。

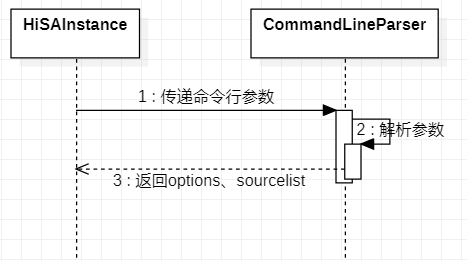
所包括的map有：

|  |  |
| --- | --- |
| Map名称 | 功能 |
| Command\_map | 储存核心命令，包括checker选择等 |
| Debug\_map | 储存debug模式下使用的命令 |
| Output\_map | 储存与输出相关的map |

#### 类图



#### 序列图



## 统计模块

### 功能介绍

该模块负责对静态分析模块生成的结果进行统计分析，统计静态分析模块产生的数据，包括：

总体数据：总项目的缺陷数量和缺陷类型数量

单源文件数据：分析时间、单文件缺陷数量和单文件缺陷类型数量

以提供给输出模块进行输出。

### 相关接口和功能

#### StatisticsHelper类

|  |  |
| --- | --- |
| 接口描述 | 解释 |
| Instance() | 数据统计接口单实例 |
| collect\_generate(const vector<Diagnostic>&, const vector<Stats> &) | 收集静态分析的数据，并产生各类统计信息 |
| get\_file\_stats\_list() | 得到所有文件统计数据的列表。 |
| get\_filename\_list() | 得到所有文件路径名 |
| get\_all\_bug\_type\_list() | 得到项目所有缺陷类型 |
| get\_all\_bug\_stats\_list() | 得到项目所有缺陷的统计数据 |
| get\_all\_bug\_num() | 得到项目所有缺陷的数量 |
| get\_all\_bug\_type\_num() | 得到项目缺陷的所有类型数量 |
| get\_all\_file\_num() | 得到项目分析的文件数量 |
| get\_all\_analysis\_time() | 得到项目分析总时间 |
| clear | 清楚所有统计数据 |

#### FileStats类

|  |  |
| --- | --- |
| 接口描述 | 解释 |
| get\_filename() | 获取文件路径名 |
| get\_analysis\_time() | 获取文件分析时间 |
| get\_all\_bug\_num() | 得到文件的缺陷数量 |
| get\_bug\_type\_num() | 得到文件缺陷的类型数量 |
| get\_bug\_stats\_list() | 得到文件的缺陷统计数据列表 |
| get\_bug\_type\_list() | 得到文件的缺陷类型列表 |
| clear() | 清楚文件的统计数据 |

#### BugStats类

|  |  |
| --- | --- |
| 接口描述 | 解释 |
| get\_bug\_type() | 获取缺陷统计数据中的缺陷类型 |
| get\_num() | 获取缺陷统计数据中的bug数量 |
| get\_bug\_desc\_list() | 获取此类型缺陷详细信息的列表 |
| clear() | 清楚此缺陷类型统计数据 |

#### BugStatsDesc类

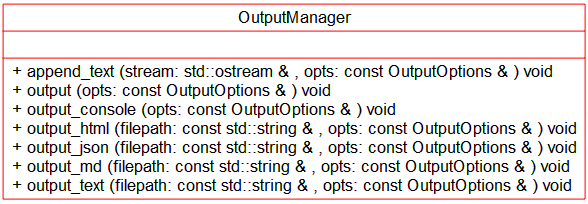
|  |  |
| --- | --- |
| 接口描述 | 解释 |
| get\_file\_name() | 获取缺陷所在文件 |
| get\_checker\_name() | 获取缺陷检测所用checker |
| get\_description() | 获取缺陷信息的描述 |
| get\_line\_number() | 获取缺陷在所在文件中的行数 |
| get\_column\_number() | 获取缺陷在所在文件中的列数 |
| clear() | 清除此缺陷的描述信息 |

## 输出模块

### 功能介绍

该模块负责处理对程序运行后获得的缺陷检测数据及其统计数据进行格式化，输出成为不同类型的文件，包括html格式、Json格式、text格式、markdown格式和命令台格式。

### 类图



### 相关接口和功能

|  |  |
| --- | --- |
| 接口 | 功能 |
| Output | 根据opts的选项值输出内容 |
| Output\_console | 输出到控制台 |
| Output\_html | 输出到对应html文件夹 |
| output\_json | 输出到json文件 |
| output\_md | 输出到markdown文件 |
| Output\_text | 输出成文本文件 |

## Json模块

### 功能介绍

作为工具类，负责将Json字符文件转换成对应Json数据结构，或将Json数据结构转换成字符串，并输出到相应文件中。

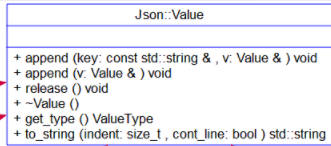
在本软件中，用于OutputManager输出Json文件，以及读取compile\_commands.json文件至程序中。

在设计Json树结构时，采用了组合模式的设计模式，以Value类作为Json元素单元组成Json树，并在此基础上进行一系列操作。

### 类图

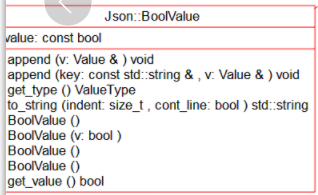
通用的Json处理：

Value基类：

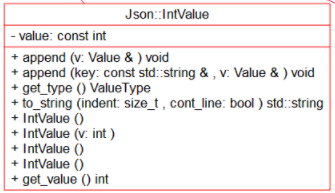


该基类是所有Json元素的抽象类，每一个Value都对应一个Json元素，例如对象、数组、整数、字符串等。

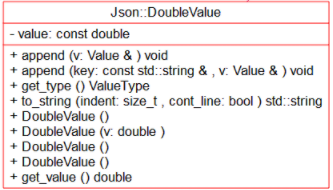
布尔值Value：



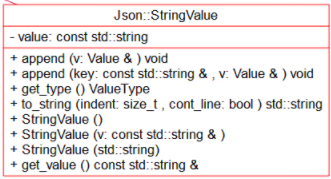
整型值Value：



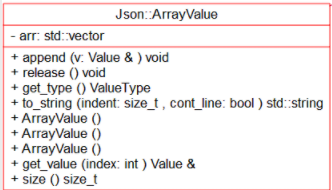
浮点型Value：



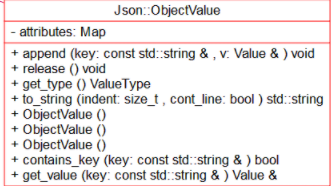
字符串型Value：



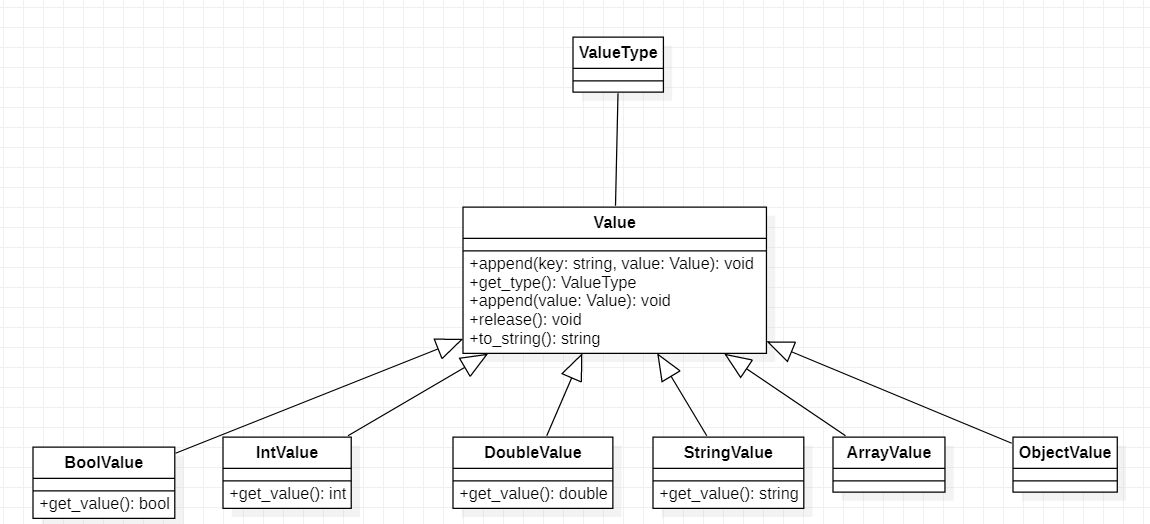
数组型Value：



对象型Value：



整体关系：

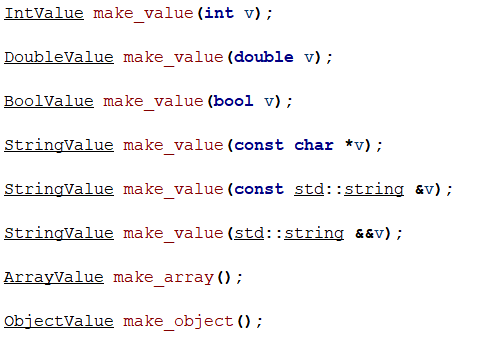


### 核心算法

#### 生成Json数据结构

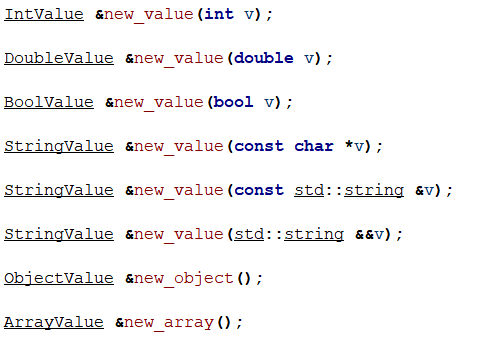
##### 在栈中生成

通过make\_value函数，利用函数重载实现Value的生成，通过利用ArrayValue和ObjectValue的append方法添加子节点，生成Json树。



##### 在堆中生成

通过new\_value函数，利用函数重载实现Value的生成，通过利用ArrayValue和ObjectValue的append方法添加子节点，生成Json树。



PS：生成结束并使用完成后，需手动通过release方式释放Json树所占用的空间。

#### 将Json数据结构转化为字符串

提供stringify函数进行转换，该函数会深度递归调用Json树中每个节点的to\_string方法以进行字符串转化：

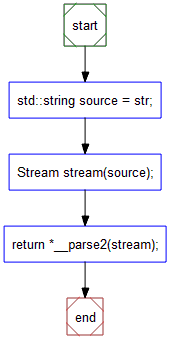
对于ObjectValue，会依次输出其包含的属性名，并调用该Value的to\_string方法进行输出。

对于ArrayValue，则会以此调用存储的Value的to\_string方法进行输出。

对于其余实值类型，则to\_string方法直接输出本身的值。

#### 将Json字符串转化为Json数据结构

使用parse函数进行转换：



其中\_\_parse2函数会通过逐个读取字符的方式进行字符串解析，并生成对应Json树。

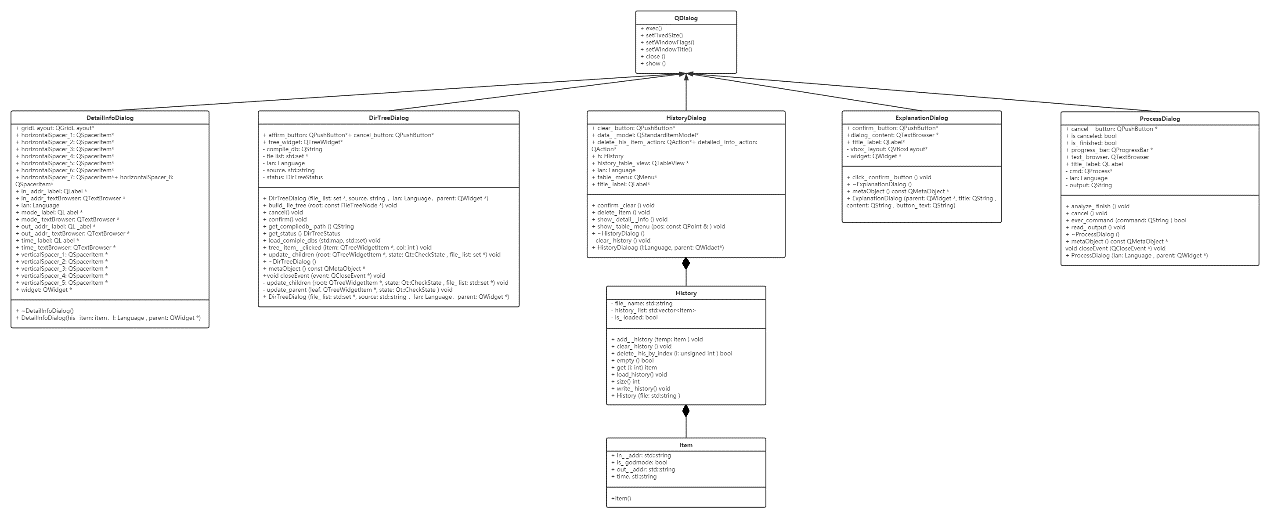
#### 将统计数据输出为Json文件

首先，算法会通过parse函数将统计数据在堆中组装成Json树，并返回树根节点的Value引用，随后，会调用根节点的to\_string方法，转化成string类型，最后使用文件流写入对应的Json文件。

## GUI模块

### 功能介绍

该模块负责调用核心模块的相关api，提供可视化界面供用户操作。



DetailInfoDialog:展示历史记录详细信息的界面

DirTreeDialog:文件树对话框界面，检测前用户需要在这个界面中选择项目对应的 compile\_commands.json 文件并选择需要检测的C++/C文件

HistoryDialog:历史数据展示界面，用户可以在这个界面查看历史记录详细信息和删除历史记录

History:负责对历史记录文件进行增删改查一系列操作并向HistoryDialog类提供历史数据记录

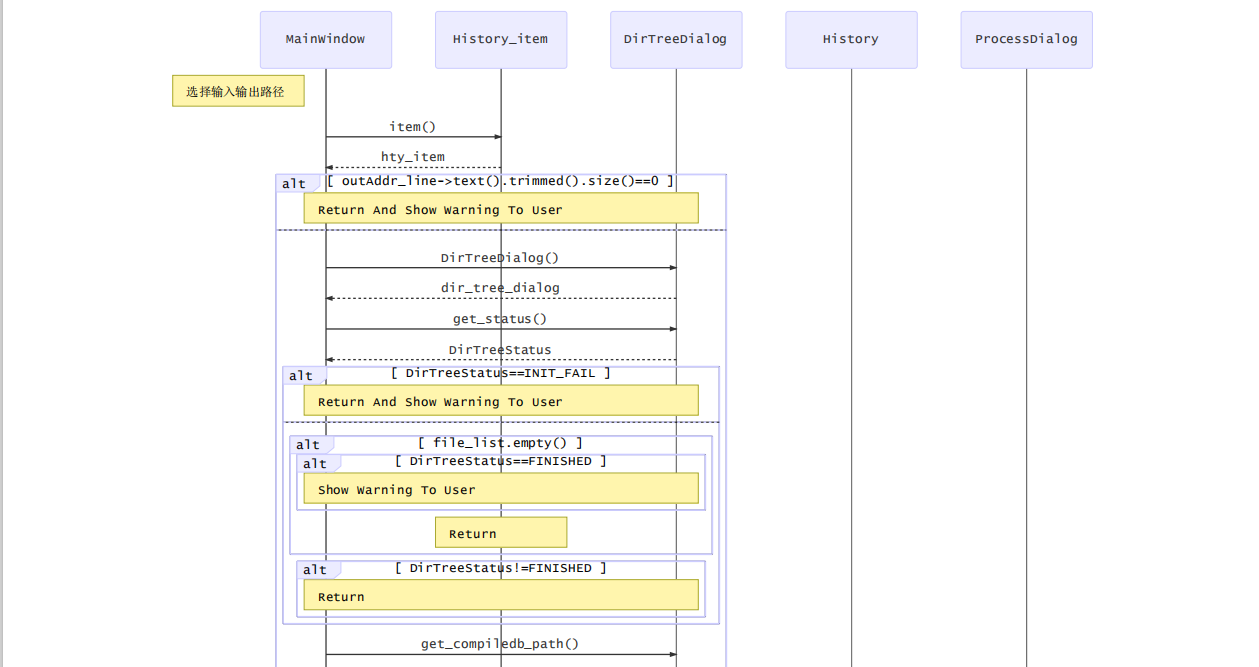
item:历史记录信息，存储了历史记录的基本信息

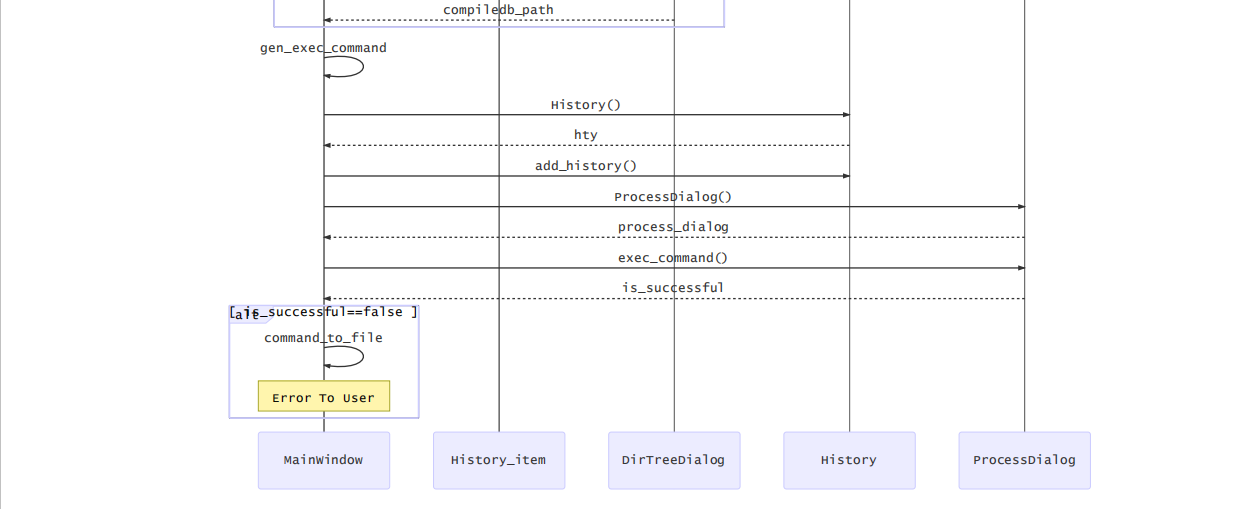
ExplanationDialog:帮助信息展示界面

ProcessDialog:进度界面，提供用户随时查看检测进度和取消检测的功能

MainWindow:主页面，用户可以在该界面选择待检测项目路径与报告输出路径并自定义检测设置

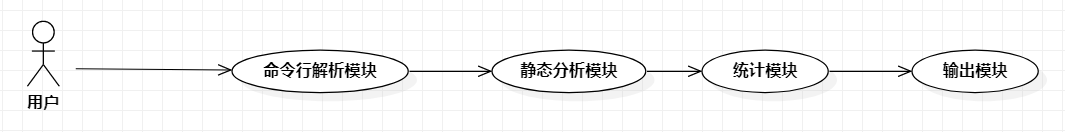
### 代码结构和处理流程





# 模块交互

## 核心模块



## GUI模块

GUI会通过调用核心模块的可执行文件进行静态分析，以获取生成的报告并在GUI中反馈给用户：

