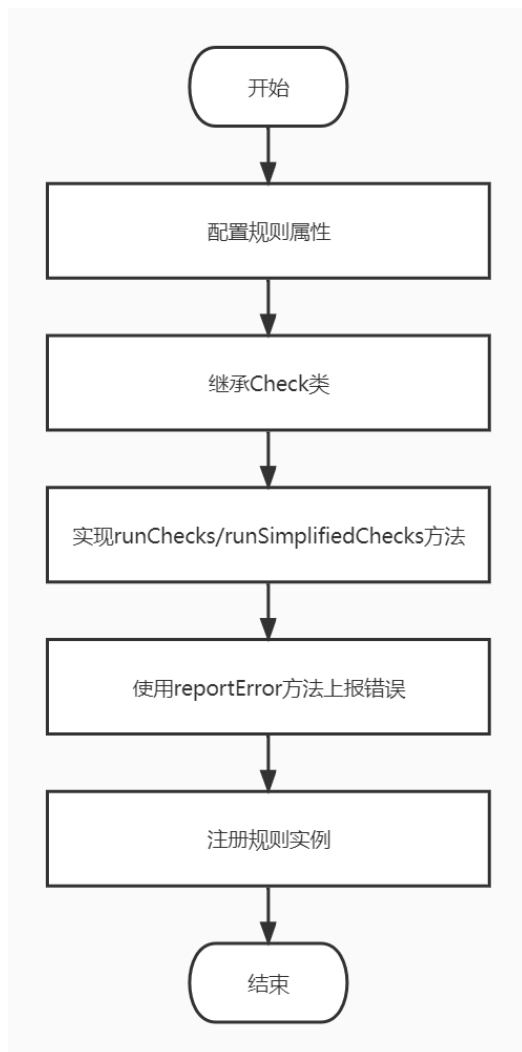


【技术调研】 TscanCode自定义规则开发

自定义规则开发流程



如上图所示，自定义规则开发需要经过五个步骤，接下来根据自定义规则样例，详细介绍自定义规则的开发流程。

自定义规则样例


需求描述

如下代码所示，当IRecordset类型的指针没有使用SAFE_RELEASE宏释放时给予警告提示。

C++

```
1  #include "recordsetpointerrelease.h"
2  void Demo()
3  {
4      IRecordset* pRecord = new Recordset();
5      //SAFE_RELEASE(pRecord);
6  }
```

实现源码

 checkrecordsetpointerrelease.h

C++

```
1  #pragma once
2  #include "checkcustom.h"
3  /** @brief 检查IRecordset类型的指针是否使用SAFE_RELEASE宏释放 */
4  class TSCANCODELIB CheckIRecordsetPointerRelease : public CheckCustom {
5  public:
6      CheckIRecordsetPointerRelease() : CheckCustom() {
7      }
8
9      CheckIRecordsetPointerRelease(const Tokenizer* tokenizer, const Settings*
settings, ErrorLogger* errorLogger)
10         : CheckCustom(tokenizer, settings, errorLogger) {
11      }
12      std::string GetRuleId() const override {
13          return "recordsetPointerRelease";
14      }
15
16      void runChecks(const Tokenizer* tokenizer, const Settings* settings,
ErrorLogger* errorLogger) override {
17          CheckIRecordsetPointerRelease checker(tokenizer, settings,
errorLogger);
18              checker.Check();
19      }
20
21      void runSimplifiedChecks(const Tokenizer* tokenizer, const Settings*
settings, ErrorLogger* errorLogger) override {
22      }
23
24      void getErrorMessage(ErrorLogger* errorLogger, const Settings*
settings) const override {
25      }
26
27      void Check();
28
29  };
```



checkrecordsetpointerrelease.cpp

C++

```
1  #include "check.h"
2  #include "checkrecordsetpointerrelease.h"
3  #include "symboldatabase.h"
4
```

```

5 // Register this check class (by creating a static instance of it)
6 namespace {
7     CheckIRecordsetPointerRelease instance;
8 }
9
10 void CheckIRecordsetPointerRelease::Check() {
11     const SymbolDatabase *symbolDatabase = _tokenizer->getSymbolDatabase();
12     const std::size_t functionSize = symbolDatabase->functionScopes.size();
13     for (std::size_t i = 0; i < functionSize; ++i) {
14         const Scope* functionScope = symbolDatabase->
15 >functionScopes[i];
16         const Token* tok = functionScope->classStart;
17         const Token* end = functionScope->classEnd;
18         if (tok && end) {
19             end = end->next();
20             std::map<std::string, const Token *> vars_map;
21             for (; tok && tok != end; tok = tok->next()) {
22                 if (Token::Match(tok, "IRecordset * %var%")) {
23                     tok = tok->next()->next();
24                     if (tok && tok->variable() && tok->variable()-
25 >isPointer()) {
26                         vars_map[tok->str()] = tok;
27                     }
28                 }
29                 if (Token::Match(tok, "SAFE_RELEASE ( %var%
30 )")) {
31                     tok = tok->next()->next();
32                     if (tok && tok->variable() && tok->variable()-
33 >isPointer()) {
34                         vars_map.erase(tok->str());
35                     }
36                 }
37             }
38             //处理SAFE_RELEASE宏被解析成实际代码时的情况
39             if (Token::Match(tok, "%var% . Release ( )"))
40 {
41                 if (tok && tok->variable() && tok->
42 >variable()->isPointer()) {
43                     vars_map.erase(tok->str());
44                 }
45             }
46         }
47         if (vars_map.empty() == false) {
48             for (auto iter = vars_map.begin(); iter != vars_map.end();
49 ++iter) {
50                 std::stringstream ss;
51                 ss << "Pointer " << iter->first << " is not freed using
52 the SAFE_RELEASE method.";
53                 ReportError(iter->second,

```

```

        ss.str().c_str());
45         }
46     }
47 }
48 }
49 }

```

配置规则属性

```

9 <subid name="possibleUninitVar" value="0" severity="Serious" rule_name="可能的变量未初始化" desc="未初始化变量用作等号右值或者函数参数"/>
0 <subid name="uninitPtr" value="1" severity="Serious" rule_name="指针变量未初始化" desc="未初始化的指针"/>
1 <subid name="possibleUninitPtr" value="0" severity="Serious" rule_name="可能的指针变量未初始化" desc="未初始化的指针变量用作等号右值或者函数参数"/>
2 <subid name="uninitMemberInCtor" value="1" severity="Serious" rule_name="构造函数使用未初始化成员变量" desc="构造函数使用未初始化成员变量"/>
3 <subid name="possibleUninitStruct" value="0" severity="Serious" rule_name="可能的未初始化结构体变量" desc="未初始化的结构体变量用作等号右值或者函数参数"/>
4 <subid name="UninitStruct" value="0" severity="Serious" rule_name="未初始化结构体变量" desc="结构体变量没有初始化, 或者部分成员变量未初始化"/>
5 <subid name="possibleUninitMemberInCtor" value="0" severity="Serious" rule_name="构造函数中成员变量可能未初始化" desc="构造函数中未初始化成员变量用作等号右值或
6 <subid name="uninitMemberVar" value="1" severity="Warning" rule_name="构造函数中成员变量未初始化" desc="构造函数中部分成员变量未初始化"/>
7 </id>
8 <id name="custom" value="1">
9 <subid name="recordsetPointerRelease" value="1" severity="Serious" rule_name="IRecordset指针释放" desc="IRecordset类型的指针没有使用SAFE_RELEASE方法进行释放"/>
0 </id>

```

在trunk/cfg/cfg.xml文件中配置自定义规则的相关属性，属性说明如下：

Plain Text

```

1 <id name="<规则分组ID>" value="<规则分组是否开启：1开启、0关闭>">
2     <subid name="<规则ID>" value="<规则是否开启：1开启、0关闭>" severity="<规则扫描
   的严重级别>" rule_name="<规则名称>" desc="<规则描述>"
3     </subid>
4 </id>

```

后续就可以在代码中获取规则的相关属性，用于判断规则是否开启、获取规则的严重级别、上报错误等。

继承Check类

自定义规则需要继承Check类，由于自定义规则有不少通用操作，因此封装了一个CheckCustom基类，后续自定义规则只需要继承CheckCustom基类即可。

C++

```
1  #pragma once
2  #include "check.h"
3  /**
4   * @brief 自定义规则类封装
5   */
6  class TSCANCODELIB CheckCustom : public Check
7  {
8  public:
9      /** @brief This constructor is used when registering the CheckClass */
10     CheckCustom() :Check(GetRuleId()) {
11     }
12
13     /** @brief This constructor is used when running checks. */
14     CheckCustom(const Tokenizer* tokenizer, const Settings* settings,
15     ErrorLogger* errorLogger)
16         : Check(GetRuleId(), tokenizer, settings, errorLogger) {
17     }
18
19     std::string classInfo() const {
20         return GetRuleId();
21     }
22
23     /**
24     * @brief 获取规则ID
25     * @return 返回规则ID
26     */
27     virtual std::string GetRuleId()const {
28         return "";
29     };
30
31     /**
32     * @brief 上报错误
33     * @param token 发生错误的符号
34     * @param msg 错误消息
35     * @return 无
36     */
37     void ReportError(const Token* token, std::string msg) {
38         Check::reportError(token, Severity::error, mErrorType,
39         GetRuleId(), msg, 0U, true);
40     }
41
42 protected:
43     const ErrorType::ErrorTypeEnum mErrorType = ErrorType::UserCustom;
44 };
```

实现runChecks/runSimplifiedChecks方法

C++

```
1      void runChecks(const Tokenizer* tokenizer, const Settings* settings,
2      ErrorLogger* errorLogger) override {
3          CheckIRecordsetPointerRelease checker(tokenizer, settings,
4          errorLogger);
5          checker.Check();
6      }
7
6      void runSimplifiedChecks(const Tokenizer* tokenizer, const Settings*
7      settings, ErrorLogger* errorLogger) override {
```

自定义规则需要继承CheckCustom类并实现runChecks或者runSimplifiedChecks方法。runChecks和runSimplifiedChecks方法的参数类型和个数都一样，唯一的区别在于runSimplifiedChecks方法中tokenizer提供的符号列表是经过简化处理的。实现自定义规则一般通过tokenizer获取符号数据库，然后获取每个方法的作用域，对每个方法中的符号进行扫描判断是否符合规则特征。

使用reportError方法上报错误

当方法中的符号匹配规则特征时，需要使用reportError方法上报错误，其中使用到了规则属性中的规则ID。

PHP

```
1      /**
2      * @brief 上报错误
3      * @param token 发生错误的符号
4      * @param msg 错误消息
5      * @return 无
6      */
7      void ReportError(const Token* token, std::string msg) {
8          Check::reportError(token, Severity::error, mErrorType,
9          GetRuleId(), msg, 0U, true);
10     }
```

注册规则实例

自定义规则类开发完成后，还需要生成一个实例注册给框架调用。

TypeScript

```
1
2 // Register this check class (by creating a static instance of it)
3 namespace {
4     CheckIRecordsetPointerRelease instance;
5 }
```