本 科 实 验 报 告

实验名称：clang-tidy 插件开发

学　　员：马坚　　学　　号：201912509012

培养类型：指挥　　年　　级：2019 级

专　　业：软件工程　　所属学院：计算机学院

指导教员：陈振邦　　职　　称：

实 验 室：306-705　　实验日期：2022-6-18

国防科技大学训练部制

1. 实验目的和要求

1.1. 目的

通过实验掌握*llvm*环境配置，能够通过语法树来分析程序代码。了解掌握*clang*二次开发，通过编写代码实现两个语法检查，加深对知识点的理解。

1.2. 要求

1) 每人完成2个指定*check*
2) 将*check*注册到*gjb*中
3) 根据*GJB*标准完成对单个文件的测试
4) 完成对*Benchmarks*的测试
5) 如有无法完成的*check*，注明详细原因

2. 实验内容和原理

2.1. 内容

下载*LLVM*源码进行安装编译，并借助*Clion*对*clang-tidy*进行二次开发，编写代码实现两个不同的*check*，分别是检测函数是否有返回语句，主过程中是否含有未调用的子过程。

1）函数必须有返回语句

一个函数应该有一个返回语句，否则函数会返回一个随机数，这个随机数通常是堆栈顶端值。例如：

```
/*************************************************************
 * 函数必须有返回语句
 *************************************************************/
unsigned int static_p(unsigned int p_1, unsigned short p_2)
{
    unsigned int y=p_1;
    /* Not returning a value */
}
```

图2-1 函数必须有返回语句详解图

2）主过程所在文件中禁止有未被该文件中任何过程调用的子过程

主过程所在文件中如果有未被该文件中任何过程调用的子过程，那么这个子过程有可能被其他文件中的过程调用，应该把这个子过程移到有过程调用它的那个文件中去。例如：

```
/*****************************************************************************
 * 主过程所在文件中禁止有未被该文件中任何过程调用的子过程
 *****************************************************************************/
unsigned int static_p(unsigned int p)
{
    unsigned int x =1u;
    /*...*/
    x=x+p;
    return x;
}
int main(void)
{
    /*...*/
    return(0);
}
```

图 2-2 主过程不得包含未被调用的子过程详解图

2. 2. 原理

AST 作为语法分析树的一种简写方式，它独立于具体编程语言而且与语法分析树的建立过程无关是联系编译器前端、后端的重要接口。

Clang 的 *AST* 与其他一些编译器生成的 *AST* 不同，因为它与编写的 *C* ++代码和 *C* ++标准非常相似。这使得 *Clang* 的 *AST* 非常适合重构工具。

通过命令 *clang -Xclang -ast-dump -fsyntax-only *.cpp* 可以生成其对应的语法树，通过语法树可以分析代码。例如 *a.cpp*：

```
jackyma@cyber:~/Study/Tools/LLVM$ cat a.cpp
int main(){
    int x = 0;
    return x;
}
```

图 2-3 *a.cpp* 代码内容

生成的语法树如下：

```
`-FunctionDecl 0x55862eed4170 <a.cpp:1:1, line:4:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x55862eed4380 <col:11, line:4:1>
    |-DeclStmt 0x55862eed4320 <line:2:5, col:14>
    | `-VarDecl 0x55862eed4298 <col:5, col:13> col:9 used x 'int' cinit
    |   `-IntegerLiteral 0x55862eed4300 <col:13> 'int' 0
    `-ReturnStmt 0x55862eed4370 <line:3:5, col:12>
      `-ImplicitCastExpr 0x55862eed4358 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0x55862eed4338 <col:12> 'int' lvalue Var 0x55862eed4298 'x' 'int'
```

图 2-4 *a.cpp* 生成的语法树

其中 *FunctionDecl* 节点对应的是函数节点；*CompoundStmt* 节点对应的是函数的主题部分，也就是大括号内的内容，可以通过有无 *CompoundStmt* 节点来判断是否为函数的声明语句，在第一个 *check* 中有所体现；*ReturnStmt* 节点对应的

就是返回语句，这就是第一个*check*要检查的内容。

*clang-tidy*就是通过对*AST*中的节点匹配，搭配一些判断语句，进行语法分析，并通过框架内的接口*diag*在控制台打印输出相关信息，例如：

```
diag( Loc: MatchedDecl->getLocation(),
      Description: "Function '%0' has never be used in main file")
    << MatchedDecl->getName();
```

图 2-5 *diag*接口用法格式

此语句将会在控制台打印*MatchedDecl*节点对应的位置，并输出后面的描述语句，就像这样：

```
/home/jackyma/Study/Tools/LLVM/cmake-build-release/bin/../../test.cpp:
5:5: warning: Function 'foo' has no return [gjb-my-first-check]
int foo(int x){
    ^
Suppressed 1 warnings (1 with check filters).
```

图 2-6 控制台打印结果

3. 实验步骤

3.1. 环境配置

详见 clang-tidy 安装编译及开发指南。版本：Ubuntu 22.04 LTS，LLVM 14.0.0.0

3.2. 代码编写

3.2.1. *FuncReturnCheck*

首先添加需要匹配的节点并为其绑定一个标识。在下图的代码中，接口*addMatcher*用来添加新的匹配节点；参数*isExpansionInMainFile*用于判断是否为待测文件夹内节点，如果没有这个参数，那么检测出来的*warning*数量就是包含了头文件和库文件的所有警告；参数*hasDescendant*用于判断该节点是否存在子节点，这里还用到了*anyOf*和*Unless*，此处同时用到，是为了无论有没有*return*语句都要匹配*FunctionDecl*节点，否则如果没有*ReturnStmt*节点，那么*FunctionDecl*节点也不会被匹配到，这是不符合我们的预期的；最后*bind*是设置表示。

```
19  void MyFirstCheckCheck::registerMatchers(MatchFinder *Finder) {
20    // FIXME: Add matchers.
21    Finder->addMatcher(
22  NodeMatch: functionDecl( Arg1: isExpansionInMainFile(),
23                anyOf(hasDescendant( InnerMatcher: compoundStmt().bind( ID: "cpd")),
24                      unless(hasDescendant( InnerMatcher: compoundStmt().bind( ID: "cpd")))),
25                anyOf(hasDescendant( InnerMatcher: returnStmt().bind( ID: "ret")),
26                      unless(hasDescendant( InnerMatcher: returnStmt().bind( ID: "ret")))))
27      .bind( ID: "func"),
28  Action: this);
29  }
```

图 3-1 *check*1绑定匹配节点代码

在*check*方法里匹配节点，当匹配到函数定义的节点后，首先判断该函数返回类型以及是否是主函数，如果是*void*、*void* *、*main*函数，那么不再进行接下来的判断。随后判断，该函数是否是声明语句，在*AST*语法树中，函数的声明语句也被视为*FunctionDecl*节点。二者的区别在于声明语句的*FunctionDecl*节点没有*CompoundStmt*子节点，因此可以通过是否含有*CompoundStmt*子节点来判断是否是声明语句。最后判断是否含有*ReturnStmt*节点，即是否有*return*返回语句，如果没有，则报出*warning*。

```
void MyFirstCheckCheck::check(const MatchFinder::MatchResult &Result) {
  // FIXME: Add callback implementation.

  /// get Node which we want to get
  if(const auto *MatchedDecl :const FunctionDecl* =
          Result.Nodes.getNodeAs<FunctionDecl>( ID: "func")){
    /// ReturnType is not 'void' or 'void *'
    if(MatchedDecl->getReturnType().getAsString() != "void"
        && MatchedDecl->getReturnType().getAsString() != "void *"
        /// Func Name is not "main"
        && MatchedDecl->getName() != "main"){
      /// Func has definition
      if(const auto *MatchedCpd :const CompoundStmt* =
              Result.Nodes.getNodeAs<CompoundStmt>( ID: "cpd")){
        /// Func has ReturnStmt
        if(const auto *MatchedStmt :const ReturnStmt* =
                Result.Nodes.getNodeAs<ReturnStmt>( ID: "ret")){
        }
        /// Func does not have ReturnStmt
        else{
          diag( Loc: MatchedDecl->getLocation(),
            Description: "Function '%0' has no return") << MatchedDecl->getName();
        }
      }
    }
  }
}
```

图 3-2 *check*1 详细代码

3.2.2. *FuncUseCheck*

首先添加需要匹配的节点并为其绑定标识，这里是遍历所有待测文件中的*FunctionDecl*节点。

```
void MySecondCheckCheck::registerMatchers(MatchFinder *Finder) {
  // FIXME: Add matchers.
  Finder->addMatcher( NodeMatch: functionDecl( Arg1: isExpansionInMainFile())
                          .bind( ID: "func"),
                      Action: this);
}
```

图 3-3 *check*2 绑定匹配节点代码

匹配到节点后，判断是否为*main*函数，如果不是则调用接口*isUsed*来判断是否被调用，如果没有被调用，则打印*warning*。

```
void MySecondCheckCheck::check(const MatchFinder::MatchResult &Result) {
  // FIXME: Add callback implementation.

  if (const auto *MatchedDecl :const FunctionDecl* =
          Result.Nodes.getNodeAs<FunctionDecl>( ID: "func")){
    /// Func doesn't used in main file
    if (!MatchedDecl->isUsed() && MatchedDecl->getName() != "main")
      diag( Loc: MatchedDecl->getLocation(),
            Description: "Function '%0' has never be used in main file")
          << MatchedDecl->getName();
  }
}
```

图 3-4 *check2*详细代码

4. 实验结果与分析

4.1. 简单用例测试

编写一段简单的代码，仅通过此代码即可验证两个*check*的正确性，代码的内容如下：

```
#include "head.h"

int test(int x);

int foo(int x){ /*return 2*x;*/ }

int foo1(int x){ return x; }

int foo2(int x){ return x*x; }

int main(){
        test();
        //int x = foo(1);
        int y = foo1(1);
        int z = foo2(1);
        return 0;
}

int test(int x){ return 1; }
```

图 4-1 简单测试用例

1）函数必须有返回语句

第 3 行的声明函数不应报出*warning*，第 5 行函数没有*return*语句，应当报出*warning*。检测结果如下：

```
jackyma@cyber:~/Study/Tools/LLVM/cmake-build-release/bin$ ./clang-tidy
 -checks=-*,gjb-my-first-check ../../test.cpp --
2 warnings generated.
/home/jackyma/Study/Tools/LLVM/cmake-build-release/bin/../../test.cpp:
5:5: warning: Function 'foo' has no return [gjb-my-first-check]
int foo(int x){ /*return 2*x;*/ }
    ^
Suppressed 1 warnings (1 with check filters).
```

图 4-2 检查结果 1

通过控制台打印的信息，符合预期结果。

2）主过程中不能有未被调用的子过程

函数*test*和*foo*都没有被调用，结果应报出这两个子过程没有被调用。这里*test*出现了两次，一次是声明，一次是定义，最后决定还是两处都报（如果只报一次，可以通过判断是否有*CompoundStmt*节点来判断属于哪一类）。结果如下：

```
jackyma@cyber:~/Study/Tools/LLVM/cmake-build-release/bin$ ./clang-tid
y -checks=-*,gjb-my-second-check ../../test.cpp --
4 warnings generated.
/home/jackyma/Study/Tools/LLVM/cmake-build-release/bin/../../test.cpp
:3:5: warning: Function 'test' has never be used in this file [gjb-my
-second-check]
int test(int x);
    ^
/home/jackyma/Study/Tools/LLVM/cmake-build-release/bin/../../test.cpp
:5:5: warning: Function 'foo' has never be used in this file [gjb-my-
second-check]
int foo(int x){ /*return 2*x;*/ }
    ^
/home/jackyma/Study/Tools/LLVM/cmake-build-release/bin/../../test.cpp
:19:5: warning: Function 'test' has never be used in this file [gjb-m
y-second-check]
int test(int x){ return 1; }
    ^
Suppressed 1 warnings (1 with check filters).
```

图 4-3 检查结果 2

控制台的输出也是符合预期的。

4.2. *Benchmarks* 测试

1) *C-master*

a) *FuncReturnCheck*

No warning

b) *FuncUseCheck*

No warning

2) *disque-master*

a) *FuncReturnCheck*

No warning

b) *FuncUseCheck*

测试时发现如下的*warning*，在主过程中存在未被调用的子过程，顺着提示的路径找到对应文件，在文件中搜索定位到函数*serverDebug*，发现只有一条记录且是函数的定义，因此断定该子过程未在主过程中使用。

```
clang-tidy --use-color -checks=-*,gjb-my-second-check -p=/home/jackyma/Study/Cou
rses/Software-Quality-Assurance/规则检查实验-jun/bemchmarks/disque-master /home/
jackyma/Study/Courses/Software-Quality-Assurance/规则检查实验-jun/bemchmarks/dis
que-master/src/server.c
server.c:211:6: warning: Function 'serverDebug' has never be used in this file [
gjb-my-second-check]
void serverDebug(const char *fmt, ...) {
     ^
server.c:281:10: warning: Function 'randomTimeError' has never be used in this f
ile [gjb-my-second-check]
mstime_t randomTimeError(mstime_t milliseconds) {
         ^
server.c:289:6: warning: Function 'exitFromChild' has never be used in this file
 [gjb-my-second-check]
void exitFromChild(int retcode) {
     ^
```

图 4-4 控制台打印信息

图 4-5 源文件对应信息

3) *giflib-bug-74*

a) *FuncReturnCheck*

No warning

b) *FuncUseCheck*

No warning

4) *glfw-master*

a) *FuncReturnCheck*

No warning

b) *FuncUseCheck*

No warning

5) *redis-unstable*

a) *FuncReturnCheck*

测试时发现如下的*warning*，存在函数没有*return*语句。同样，依据路径提示找到相应的文件，全局搜索函数*redis_check_aof_main*，发现该函数是以*exit*来结束，并没有*return*语句，检测符合要求。

```
clang-tidy --use-color -checks=-*,gjb-my-first-check -p=/home/jackyma/Study/Cour
ses/Software-Quality-Assurance/规则检查实验-jun/bemchmarks/redis-unstable /home/
jackyma/Study/Courses/Software-Quality-Assurance/规则检查实验-jun/bemchmarks/red
is-unstable/src/redis-check-aof.c
redis-check-aof.c:522:5: warning: Function 'redis_check_aof_main' has no return
[gjb-my-first-check]
int redis_check_aof_main(int argc, char **argv) {
    ^
1 warning generated.
```

图 4-6 控制台打印信息

图 4-7 源文件信息

b) *FuncUseCheck*

测试时发现如下的*warning*，在主过程中存在未被调用的子过程，找到对应文件，在文件中搜索定位到函数*validateProcTitleTemplate*，发现只有一条记录且是函数的定义，因此断定该子过程未在主过程中使用。

图 4-8 控制台信息

图 4-9 源文件信息

5. 总结与致谢

通过本次实验，我进行了第一次源码编译安装软件，在源码编译的过程中我了解到了*Debug*和*Release*版本的区别。*Debug*版本编译速度慢、占用空间大，但是能提供较为精准的报错信息，总的来说，*Debug*版本是开发阶段的版本，在这个阶段开发人员希望能够获得更多的报错信息，以此来提高软件的质量。而*Release*版本是开发完成后的发行版本，此版本面向的对象是用户，此版本在底层加了许多优化，最终展现出来的就是编译速度快、占用空间小，但是报错信息并不会那么详细。另外，通过源码编译，体验到了原汁原味的开发环境，第一次看到如此庞大的工程项目，内心还是颇有触动。

*clang-tidy*二次开发的过程中，熟悉了其基本框架，并通过代码编写，节点匹配，强化了对语法树的认识，虽谈不上掌握了其底层原理，但是拿到一段程序，对

照语法树能说出一二来。同时也学习到了许多相关的命令，比如：

*clang -Xclang -ast-dump -fsyntax-only *.cpp* 可以生成对应的语法树；对单个文件的分析和批量文件分析又各有其对应的语法，这里就不详细介绍了。

在实验的过程中有许多不懂的地方，感谢教员与教辅的耐心指导，使我在面对困难时有了更多的信心，最终成功完成实验。也感谢我身边的同学，当我有疑惑时，总是热情第帮助我分析问题并给予解决方案，没有同学的帮助也不会有现在的成果。