uglify设计方案

版本: 0.5

2023.7.12

上海芯栈通信科技有限公司

修订历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 日期 | 作 者 | 审 核 | 说 明 |
| 0.1 | 2023.3.13 | Andy.cao |  | uglify设计报告 |
| 0.2 | 2023.3.14 | Andy.cao |  |  |
| 0.3 | 2023.6.12 | Andy.cao |  |  |
| 0.4 | 2023.6.30 | Andy.cao |  | uglify设计报告（终稿） |
| 0.5 | 2023.7.12 | Andy.cao |  | uglify设计报告（修订） |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

重要声明

**版权声明**

**版权所有 © 2021, 上海芯栈通信科技有限公司，保留所有权利。**

**目 录**

[1 丑化代码需求背景 5](#_Toc20195)

[2 丑化代码设计思路 6](#_Toc5262)

[2.1 查找所有待处理的文件 6](#_Toc298)

[2.2 变量名和函数名替换 6](#_Toc7920)

[2.3 反向astyle 6](#_Toc1406)

[3 实现代码 8](#_Toc4645)

[3.1 利用find命令找到当前目录下的所有C文件和头文件 8](#_Toc27497)

[3.2 将处理文件进行格式标准化 8](#_Toc24929)

[3.3 进行变量名和函数名的替换 8](#_Toc15325)

[3.3.1 匹配C文件中的变量名 8](#_Toc29235)

[3.3.2 匹配C文件中的函数名 8](#_Toc19714)

[3.3.3 保证函数名和变量名的首字符符合C语言规范 8](#_Toc5519)

[3.3.4 生成变量名替换规则 8](#_Toc31146)

[3.3.5 生成函数名替换规则 9](#_Toc31239)

[3.3.6 合并sed脚本 9](#_Toc278)

[3.3.7 利用sed脚本处理文件 9](#_Toc26853)

[3.3.8 删除临时文件 9](#_Toc14898)

[3.4 随机删除注释 10](#_Toc27914)

[3.5 随机插入花指令 10](#_Toc23930)

[3.6 随机删除空行 14](#_Toc14771)

[3.7 随机删除每行行首的空格 15](#_Toc793)

[3.8 Uglify总处理 16](#_Toc6305)

[4 输出结果演示 17](#_Toc7949)

[4.1 运行过程演示 17](#_Toc8751)

[4.2 输出前后部分对比 18](#_Toc21324)

[4.3 处理后文件astyle还原验证 18](#_Toc22357)

# 丑化代码需求背景

生成的代码经常需要反汇编，这个时候就需要丑化代码使阅读变得困难，也不易被反汇编读取其中代码。

通过使用ugilfy源代码，特别是C语言，既保持了开放源代码的坦诚，又可以阻碍其他人快速的进行阅读和维护的阅读难度，因此具备市场需求价值。

# 丑化代码设计思路

## 查找所有待处理的文件

使用find命令找出当前路径下的所有C文件和头文件

## 变量名和函数名替换

找出代码文件中的函数名和变量名，并根据函数名和变量名长度生成相同长度的随机名称，并利用sed命令进行全局匹配和替换，降低代码的可阅读性。

注意:

·随机生成的函数名和变量名应符合C语言规范，即首字母不能是数字

·主函数main不可被替换

## 反向astyle

由于astyle的执行思路是将代码进行规范化处理，使其变得美观易读，所以uglify的设计思路便是反向执行astyle，使其变得丑化难读。需要做的便是：1、删除空行，使代码变得紧凑，不再分段，使阅读变得困难；2、删除每一行前面的空格或制表符，使代码变得不再美观，使读者阅读到括号匹配相关问题时变得困难；3、对变量名和函数名进行替换，加大阅读难度。

# 实现代码

## 利用find命令找到当前目录下的所有C文件和头文件

|  |
| --- |
| find $file\_path -type f \( -name "\*.c" -o -name "\*.h" \) |

## 将处理文件进行格式标准化

|  |
| --- |
| astyle $source |

## 进行变量名和函数名的替换

### 匹配C文件中的变量名

|  |
| --- |
| grep -oh '[a-zA-Z\_][a-zA-Z0-9\_ ]\*=' $source | sort | uniq | sed 's/=$//' |awk '{print $NF}' >var\_names.txt |

#grep -o：只输出匹配到的部分。-E：使用扩展的正则表达式进行匹配。'^[a-zA-Z\_][a-zA-Z0-9\_]\*[[:space:]]+[a-zA-Z\_][a-zA-Z0-9\_]\*[[:space:]]\*+[a-zA-Z\_][a-zA-Z0-9\_]\*[[:space:]]\*\('：正则表达式用于匹配具有一定格式的函数声明。它匹配以字母或下划线开头的函数返回类型，后跟一个或多个字母、数字或下划线的函数名，然后是零或多个空格，最后以括号 "(" 结尾。这样的匹配会输出整个函数声明。

#awk：用于对输入进行文本处理的命令。'{print $NF}'：打印每行的最后一个字段。在这种情况下，由于之前的grep命令匹配到整个函数声明，因此这个命令将提取函数声明中的最后一个字段，即函数名。

#sed：用于进行文本替换和编辑的命令。's/($//'：替换每行中的 "(" 符号为空字符串，从而去除函数名后面的括号。

### 匹配C文件中的函数名

|  |
| --- |
| grep -ohE '^[a-zA-Z\_][a-zA-Z0-9\_]\*[[:space:]]+[a-zA-Z\_][a-zA-Z0-9\_]\*[[:space:]]\*[a-zA-Z\_][a-zA-Z0-9\_]\*\(' $source | awk '{print $NF}' | sed 's/($//'> func\_names.txt |

### 保证函数名和变量名的首字符符合C语言规范

|  |
| --- |
| first\_char=$(cat /dev/urandom | tr -dc 'a-zA-Z\_' | fold -w 1 | head -n 1) |

### 生成变量名替换规则

|  |
| --- |
| cat var\_names.txt | while IFS= read -r line  do  var\_length=${#line}  temp\_replace\_var=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w "$var\_length" | head -n 1)  replace\_var="$first\_char$temp\_replace\_var"  var\_sed\_rules="s/\<$line\>/$replace\_var/g"  echo $var\_sed\_rules  done >var\_sed\_rules.txt |

#cat /dev/urandom：/dev/urandom 是 Linux 系统中的一个伪随机数生成设备。通过使用 cat 命令读取 /dev/urandom，我们可以获取随机的字节序列。

#tr -dc 'a-zA-Z0-9'：tr 命令用于字符转换或删除操作。在这里，我们使用 -dc 选项指定要删除的字符集，即只保留字母和数字字符。

#fold -w "$length"：fold 命令用于将文本进行折行处理。使用 -w 选项指定每行的最大宽度为 $length，即指定生成的随机字符串的长度。

#head -n 1：head 命令用于显示文件的开头部分。通过使用 -n 1 选项，我们只显示第一行，即生成的随机字符串

### 生成函数名替换规则

|  |
| --- |
| cat func\_names.txt | while IFS= read -r line  do  func\_length=${#line}  temp\_replace\_function=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w "$func\_length" | head -n 1)  replace\_function="$first\_char$temp\_replace\_function"  func\_sed\_rules="s/\<$line\>/$replace\_function/g"  echo $func\_sed\_rules  done >func\_sed\_rules.txt |

### 合并sed脚本

|  |
| --- |
| cat func\_sed\_rules.txt var\_sed\_rules.txt >sed\_rules.txt |

### 利用sed脚本处理文件

|  |
| --- |
| for file in $source  do  sed -i -f sed\_rules.txt $file  done |

### 删除临时文件

#可用于查看匹配的函数名和变量名并在执行sed脚本前修改敏感的变量名和函数名，例如：

###########################################################

#删除敏感的变量名

sed -i '/\<h\>/d' var\_names.txt

#删除敏感的函数名

sed -i '/\<main\>/d' func\_names.txt

##########################################################

|  |
| --- |
| rm var\_names.txt func\_names.txt  rm func\_sed\_rules.txt var\_sed\_rules.txt  rm sed\_rules.txt |

## 随机删除注释

|  |
| --- |
| # 使用Python脚本删除注释  python3 - <<EOF  import random  import re  # 读取C文件内容  with open("$source", "r") as file:  content = file.read()  # 正则表达式匹配注释  pattern = r"/\\*.\*?\\*/|//.\*?$"  comment\_blocks = re.findall(pattern, content, re.DOTALL | re.MULTILINE)  # 随机删除一部分注释  for block in comment\_blocks:  if random.random() < 0.5: # 随机删除概率为50%  content = content.replace(block, "")  # 保存修改后的文件  with open("$source", "w") as file:  file.write(content)  print("删除注释完成")  EOF |

## 随机插入花指令

|  |
| --- |
| #显示函数体的函数名所在的行号  function\_name\_line=$(grep -n -E '^[a-zA-Z\_][a-zA-Z0-9\_]\*[[:space:]]+[a-zA-Z\_][a-zA-Z0-9\_]\*[[:space:]]\*[a-zA-Z\_][a-zA-Z0-9\_]\*\(' $source | cut -d ':' -f 1)  for line\_num in $function\_name\_line  do  # 生成一个随机数（范围：0到1）  random\_number=$((RANDOM % 2))  # 插入随机判断  if [ $random\_number -eq 1 ]; then    # 使用 sed 命令处理数组  processed\_line=$(echo "$line" | sed 's/foo/bar/g')  # 输出处理后的数组  # 生成第二个随机数（范围：0到4）  random\_number1=$((RANDOM % 5))  # 利用随机数进行判断花指令类型根据随机数的大小随机插入五种类型的花指令  case $random\_number1 in  0)  #插入花指令  line\_number=$((processed\_line+countline+2)) # 要插入花指令的起始行号  message1=" while(0)"  message2=" {"  message3=" int abcd = 1;dcba = 128"  message4=" for(abcd;;abcd++)"  message5=" {"  message6=" dcba+=2;"  message7=" }"  message8=" }"  #构建要插入的多行花指令  insert\_lines="${message1}\n"  insert\_lines+="${message2}\n"  insert\_lines+="${message3}\n"  insert\_lines+="${message4}\n"  insert\_lines+="${message5}\n"  insert\_lines+="${message6}\n"  insert\_lines+="${message7}\n"  insert\_lines+="${message8}\n"  #在指定行之前插入多行花指令  sed -i "${line\_number}i ${insert\_lines}" $source  ((countline+=8))  echo 插入第一种花指令  ;;  1)  #插入花指令  line\_number=$((processed\_line+countline+2)) # 要插入花指令的起始行号  message1="/\*This code seems to have an error.\*/"  #构建要插入的多行花指令  insert\_lines="${message1}\n"  #在指定行之前插入多行花指令  sed -i "${line\_number}i ${insert\_lines}" $source  ((countline+=1))  echo 插入第二种花指令  ;;  2)  #插入花指令  line\_number=$((processed\_line+countline+2)) # 要插入花指令的起始行号  message1=" do"  message2=" {"  message3=" int suspect = 1;certain = 256"  message4=" for(suspect;;suspect++)"  message5=" {"  message6=" certain+=suspect;"  message7=" }"  message8=" }while(0);"  #构建要插入的多行花指令  insert\_lines="${message1}\n"  insert\_lines+="${message2}\n"  insert\_lines+="${message3}\n"  insert\_lines+="${message4}\n"  insert\_lines+="${message5}\n"  insert\_lines+="${message6}\n"  insert\_lines+="${message7}\n"  insert\_lines+="${message8}\n"  #在指定行之前插入多行花指令  sed -i "${line\_number}i ${insert\_lines}" $source  ((countline+=8))  echo 插入第三种花指令  ;;  3)  #插入花指令  line\_number=$((processed\_line+countline+2)) # 要插入花指令的起始行号  message1="/\*Warning:\*/"  message2="/\*This is just a joke.\*/"  message3="/\*You were deceived.\*/"  #构建要插入的多行花指令  insert\_lines="${message1}\n"  insert\_lines+="${message2}\n"  insert\_lines+="${message3}\n"    #在指定行之前插入多行花指令  sed -i "${line\_number}i ${insert\_lines}" $source  ((countline+=3))  echo 插入第四种花指令  ;;  4)  #插入花指令  line\_number=$((processed\_line+countline+2)) # 要插入花指令的起始行号  message1=" int bookhad = 1"  message2=" if(bookhad = 0)"  message3=" {"  message4=" for(int i = 1;i < 100;i++)"  message5=" {"  message6=" bookhad+=0;"  message7=" }"  message8=" }else"  message9=" {"  message10=" bookhad-=0;"  message11=" }"  #构建要插入的多行花指令  insert\_lines="${message1}\n"  insert\_lines+="${message2}\n"  insert\_lines+="${message3}\n"  insert\_lines+="${message4}\n"  insert\_lines+="${message5}\n"  insert\_lines+="${message6}\n"  insert\_lines+="${message7}\n"  insert\_lines+="${message8}\n"  insert\_lines+="${message9}\n"  insert\_lines+="${message10}\n"  insert\_lines+="${message11}\n"  #在指定行之前插入多行花指令  sed -i "${line\_number}i ${insert\_lines}" $source  ((countline+=11))  echo 插入第五种花指令  ;;  esac  else    # 重新生成一个新的随机数  random\_number=$((RANDOM % 2))  fi    done |

## 随机删除空行

|  |
| --- |
| #创建一个变量用来匹配删除后的行数  countzeroline=0  # 使用grep命令匹配所有空行，并将结果存储在空行列表中  empty\_lines=($(grep -n '^$' $source | cut -d ':' -f 1))  # 循环处理每个空行  for line\_number in "${empty\_lines[@]}"; do  # 生成一个随机数（范围：1到2）  random\_number=$((1 + RANDOM % 2))  # 根据随机数决定是否删除当前空行  if [ $random\_number -eq 2 ]; then  # 使用sed命令删除当前空行并覆盖原始文件  deletline=$((line\_number-countzeroline))  sed -i "${deletline}d" $source  countzeroline=$((countzeroline+1))  echo 删除空行$deletline  fi  done |

## 随机删除每行行首的空格

|  |
| --- |
| # 生成一个随机数（范围：1到6）以随机确定进行数次删除  times\_random=$((1 + RANDOM % 5))  #创建一个中间临时文件  tmp\_file=$(mktemp)  # 随机循环执行  for(( i=1; i<=times\_random; i++ ))  do  # 循环处理每一行  while IFS= read -r line; do  # 判断当前行是否以空格开头  if [[ $line =~ ^[[:space:]] ]]; then  # 生成一个随机数（范围：0到1）  delete\_random=$((RANDOM % 2))  # 根据随机数决定是否删除当前行首空格  if [ $delete\_random -eq 1 ]; then  # 使用sed命令删除当前行首空格  line=$(sed 's/^[[:space:]]//' <<< "$line")  echo 删除当前行首空格  fi  fi  # 将处理后的行写入临时文件  echo "$line" >> "$tmp\_file"  done < $source    # 将临时文件覆盖原始文件  mv "$tmp\_file" $source  done |

## Uglify总处理

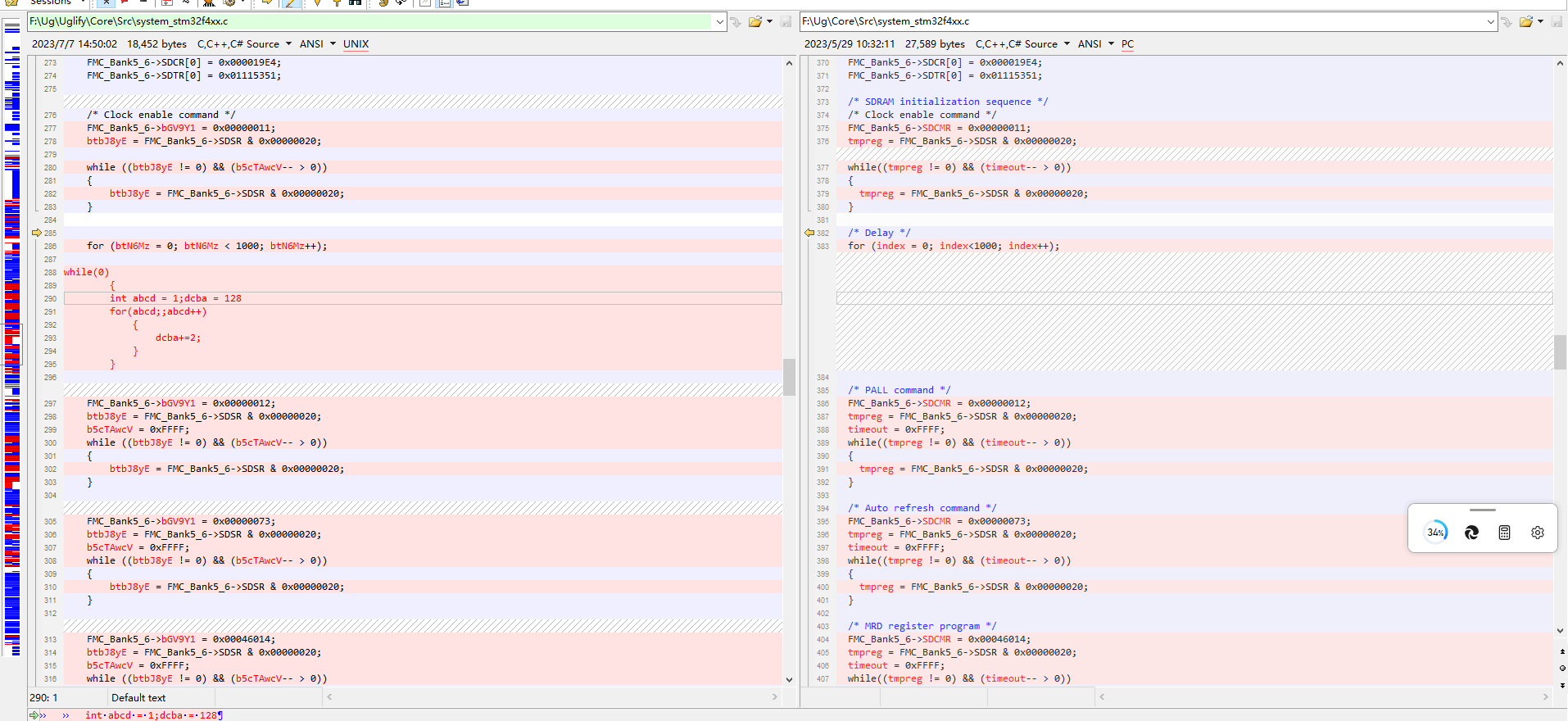
|  |
| --- |
| #!/bin/bash  #待处理文件所在路径  file\_path="$1"  echo 正在执行Uglify...  #利用find命令查找所有的C文件和头文件并执行变量名和函数名的丑化  find $file\_path -type f \( -name "\*.c" -o -name "\*.h" \)| xargs bash replace.sh  #利用find命令查找所有的C文件和头文件并执行格式的丑化  find $file\_path -type f \( -name "\*.c" -o -name "\*.h" \)| xargs bash layout.sh  #make时允许编译过程中出现未使用值的警告而不将其视为错误  find $file\_path -type f -name "makefile"| xargs sed -i '/^WARNINGS/ a -Wno-unused-value -Wno-unused-variable \\'  echo Uglify完成 |

# 输出结果演示

## 运行过程演示



## 输出前后部分对比



## 处理后文件astyle还原验证

