temp

Java语言程序的前端编译与反编译

---

1 Javap

这是接下来要使用的代码，`temp.java`：

```
class temp {
    public static void main(String[] args) {
        System.out.println("temp");
    }
}
```

首先使用`javap -v`命令，输出如下：

```
Classfile /Users/shenye/Desktop/Compiler Materia/temp.class
  Last modified Nov 29, 2023; size 399 bytes
  SHA-256 checksum a103e8dff66c34577f4c06a3d1b35c547b4f0e71475c2395ab8740ed193a89a4
  Compiled from "temp.java"
class temp
  minor version: 0
  major version: 65
  flags: (0x0020) ACC_SUPER
  this_class: #21                         // temp
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Fieldref           #8.#9          // java/lang/System.out:Ljava/io/PrintStream;
   #8 = Class              #10            // java/lang/System
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
  #10 = Utf8               java/lang/System
  #11 = Utf8               out
  #12 = Utf8               Ljava/io/PrintStream;
  #13 = String             #14            // temp
  #14 = Utf8               temp
  #15 = Methodref          #16.#17        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = Class              #18            // java/io/PrintStream
  #17 = NameAndType        #19:#20        // println:(Ljava/lang/String;)V
  #18 = Utf8               java/io/PrintStream
  #19 = Utf8               println
  #20 = Utf8               (Ljava/lang/String;)V
  #21 = Class              #14            // temp
  #22 = Utf8               Code
  #23 = Utf8               LineNumberTable
  #24 = Utf8               main
  #25 = Utf8               ([Ljava/lang/String;)V
  #26 = Utf8               SourceFile
  #27 = Utf8               temp.java
{
  temp();
    descriptor: ()V
    flags: (0x0000)
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #13                 // String temp
         5: invokevirtual #15                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 3: 0
        line 4: 8
}
SourceFile: "temp.java"
```

图1：`javap -v` 的输出内容

`javap -v` 命令用于查看 Java 类文件的详细信息，下面用我写的一个类来说明输出的组成部分：

1. 文件信息
    - 最近修改时间
    - 文件完整性检验方式：`SHA-256`

- 文件名

2. Java类信息
   - 类名: `temp`
   - 最小版本: 0 (从这个虚拟机的版本开始，向后兼容)
   - 最大版本: 65 (在这个虚拟机的版本之前，向前兼容)
   - Flags: `ACC_SUPER` (0x0020) (flags 指明了权限信息和类和接口的特性， 在这里， `ACC_SUPER` 表明这个类被 `invokespecial` 指令编译)
   - 继承关系：这个类继承 `java/lang/Object`
   - 接口，变量， 方法， 特性数量
3. 常量池
   - 一个表结构包含字符串常量，类名和方法名还有其他变量
4. 方法
   - `temp()` 方法:
     - 构造器描述: `()V`. （`()` 表明没有参数， `V` 表明返回值类型为 `void`）
     - 方法行数表：提供一个表来建立内部使用的方法与对应行数
     - 代码行数表 （字节码与行数对应）
   - `public static void main(java.lang.String[])`:
     - Descriptor `([Ljava/lang/String;)V` （表明接受 `String array` 返回 `void`）
     - 方法行数表：

```
   0: getstatic     #7                  // 变量。
java/lang/System.out:Ljava/io/PrintStream;
   3: ldc           #13                   // temp字符串
   5: invokevirtual #15                   // 方法
java/io/PrintStream.println:(Ljava/lang/String;)V
   8: return
```

- `getstatic` 从(`System.out`)获取静态变量
- `ldc` 加载字符串 (`temp`) 到栈上
- `invokevirtual` 激发 `println` 方法 到 `System.out`
- `return` 返回
- 代码行数表 （字节码与行数对应）

然后我们执行`javap -help`命令，看看还有什么参数可以使用：

```
Usage: javap <options> <classes>
where possible options include:
  --help -help -h -?              Print this help message
  -version                        Version information
  -v  -verbose                    Print additional information
  -l                              Print line number and local variable tables
  -public                         Show only public classes and members
  -protected                      Show protected/public classes and members
  -package                        Show package/protected/public classes
                                  and members (default)
  -p  -private                    Show all classes and members
  -c                              Disassemble the code
  -s                              Print internal type signatures
  -sysinfo                        Show system info (path, size, date, SHA-256 hash)
                                  of class being processed
  -constants                      Show final constants
  --module <module>  -m <module>  Specify module containing classes to be disassembled
  -J<vm-option>                   Specify a VM option
  --module-path <path>            Specify where to find application modules
  --system <jdk>                  Specify where to find system modules
  --class-path <path>             Specify where to find user class files
  -classpath <path>               Specify where to find user class files
  -cp <path>                      Specify where to find user class files
  -bootclasspath <path>           Override location of bootstrap class files
  --multi-release <version>       Specify the version to use in multi-release JAR files

GNU-style options may use = instead of whitespace to separate the name of an option
from its value.

Each class to be shown may be specified by a filename, a URL, or by its fully
qualified class name. Examples:
   path/to/MyClass.class
   jar:file:///path/to/MyJar.jar!/mypkg/MyClass.class
   java.lang.Object
```

图二：`javap -help`的输出结果

从这张图中我们可以看到`javap`这个命令有两个可以指定的环境变量：`option`，`classes`。图中展示了很多`option`的参数，可以查看版本，包，变量等等。对于`classes`， 我们可以通过文件名，URL或者验证过的类名，图中分别对应这三种方式举了三个例子。下面我们来看看其他参数的作用。

下面用一段新的代码，`Person.java`：

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

首先使用`javap -public`，这是输出：

```
Compiled from "Person.java"
public class Person {
  public Person(java.lang.String, int);
  public void displayInfo();
}
```

图三：`javap - public`的输出结果

可以看到正如所介绍的，输出是所有公共的类和它的公共成员，注意到这个包含在`javap -v`的结果里。

其次使用`javap -l`：

```
Compiled from "Person.java"
public class Person {
  public Person(java.lang.String, int);
    LineNumberTable:
      line 5: 0
      line 6: 4
      line 7: 9
      line 8: 14

  public void displayInfo();
    LineNumberTable:
      line 11: 0
      line 12: 15
      line 13: 30
}
```

图四：`javap -l`的输出结果

输出是代码行数表，记录了源代码行数与字节码的对应关系，这个也被包含了。

参数`-private, -package, -protected`的作用和`-public`差不多，输出对应权限的类和其类成员，在这里就不展示了。

接下来使用`javap -c`：

```
Compiled from "Person.java"
public class Person {
  public Person(java.lang.String, int);
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: aload_1
       6: putfield      #7                  // Field name:Ljava/lang/String;
       9: aload_0
      10: iload_2
      11: putfield      #13                 // Field age:I
      14: return

  public void displayInfo();
    Code:
       0: getstatic     #17                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_0
       4: getfield      #7                  // Field name:Ljava/lang/String;
       7: invokedynamic #23,  0             // InvokeDynamic #0:makeConcatWithConstants:(Ljava/lang/String;)Ljava/lang/String;
      12: invokevirtual #27                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      15: getstatic     #17                 // Field java/lang/System.out:Ljava/io/PrintStream;
      18: aload_0
      19: getfield      #13                 // Field age:I
      22: invokedynamic #33,  0             // InvokeDynamic #1:makeConcatWithConstants:(I)Ljava/lang/String;
      27: invokevirtual #27                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      30: return
}
```

图五：`javap -c`的输出结果

这个指令反汇编代码， 输出方法行数表。表记录了源代码中使用的方法与字节码的对应关系，这个也在`javap -v`中。

使用`javap -s`：

```
Compiled from "Person.java"
public class Person {
  public Person(java.lang.String, int);
    descriptor: (Ljava/lang/String;I)V

  public void displayInfo();
    descriptor: ()V
}
```

图六：`javap -s`的输出结果

输出了类内部两个成员的构造描述，包括参数和返回类型，这个同样也在里面。

使用`javap -sysinfo`：

```
  Last modified Nov 30, 2023; size 1034 bytes
  SHA-256 checksum 9d2746a949150b7b295f59c9de0b45ae8e1e1bd050d88c9b67116e324447d
2c5
  Compiled from "Person.java"
public class Person {
  public Person(java.lang.String, int);
  public void displayInfo();
}
```

图七：`javap -sysinfo`的输出结果

输出了文件的修改日期，完整性，文件名，类和成员。

总结

可以看出使用`javap -v`可以得到关于一个类的全部额外信息，而使用其他参数可以得到对应的信息。如果你想详细了解一个类，可以使用`javap -v`指令，但如果你对这个类有了一定认识，只想知道特定方面的信息，更推荐使用对应参数，这样结果更少也不用去寻找。

Javac

使用`javac -help`后的输出：

```
Usage: javac <options> <source files>
where possible options include:
  @<filename>                  Read options and filenames from file
  -Akey[=value]                Options to pass to annotation processors
  --add-modules <module>(,<module>)*
        Root modules to resolve in addition to the initial modules,
        or all modules on the module path if <module> is ALL-MODULE-PATH.
  --boot-class-path <path>, -bootclasspath <path>
        Override location of bootstrap class files
  --class-path <path>, -classpath <path>, -cp <path>
        Specify where to find user class files and annotation processors
  -d <directory>               Specify where to place generated class files
  -deprecation
        Output source locations where deprecated APIs are used
  --enable-preview
        Enable preview language features.
        To be used in conjunction with either -source or --release.
  -encoding <encoding>         Specify character encoding used by source files
  -endorseddirs <dirs>         Override location of endorsed standards path
  -extdirs <dirs>              Override location of installed extensions
  -g                           Generate all debugging info
  -g:{lines,vars,source}       Generate only some debugging info
  -g:none                      Generate no debugging info
  -h <directory>
        Specify where to place generated native header files
  --help, -help, -?            Print this help message
  --help-extra, -X             Print help on extra options
  -implicit:{none,class}
        Specify whether to generate class files for implicitly referenced files
  -J<flag>                     Pass <flag> directly to the runtime system
  --limit-modules <module>(,<module>)*
        Limit the universe of observable modules
  --module <module>(,<module>)*, -m <module>(,<module>)*
        Compile only the specified module(s), check timestamps
  --module-path <path>, -p <path>
        Specify where to find application modules
  --module-source-path <module-source-path>
        Specify where to find input source files for multiple modules
  --module-version <version>
        Specify version of modules that are being compiled
  -nowarn                      Generate no warnings
  -parameters
        Generate metadata for reflection on method parameters
  -proc:{none,only,full}
        Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...]
        Names of the annotation processors to run;
        bypasses default discovery process
  --processor-module-path <path>
        Specify a module path where to find annotation processors
  --processor-path <path>, -processorpath <path>
        Specify where to find annotation processors
  -profile <profile>
        Check that API used is available in the specified profile.
        This option is deprecated and may be removed in a future release.
  --release <release>
        Compile for the specified Java SE release.
        Supported releases:
            8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
  -s <directory>               Specify where to place generated source files
  --source <release>, -source <release>
        Provide source compatibility with the specified Java SE release.
        Supported releases:
            8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
  --source-path <path>, -sourcepath <path>
        Specify where to find input source files
  --system <jdk>|none          Override location of system modules
  --target <release>, -target <release>
        Generate class files suitable for the specified Java SE release.
        Supported releases:
            8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21

  --upgrade-module-path <path>
        Override location of upgradeable modules
  -verbose                     Output messages about what the compiler is doing
  --version, -version          Version information
  -Werror                      Terminate compilation if warnings occur
```

图一：`javac -help`的输出结果

可以看出`javac`指令也是有两个环境变量：`option`，`source file`. 参数后的介绍基本上解释了参数的作用，因为参数太多，所以这里选择几个平常会使用的参数来尝试。

使用 `javac -verbose`：

```
[parsing started SimpleFileObject[/Users/shenye/Desktop/Compiler Materia/Person.java]]
[parsing completed 11ms]
[loading /modules/java.base/module-info.class]
[loading /modules/jdk.security.jgss/module-info.class]
[loading /modules/jdk.internal.le/module-info.class]
[loading /modules/jdk.internal.vm.ci/module-info.class]
[loading /modules/jdk.internal.jvmstat/module-info.class]
[loading /modules/java.xml.crypto/module-info.class]
[loading /modules/jdk.crypto.cryptoki/module-info.class]
[loading /modules/java.smartcardio/module-info.class]
[loading /modules/jdk.unsupported.desktop/module-info.class]
[loading /modules/java.prefs/module-info.class]
[loading /modules/jdk.jdeps/module-info.class]
[loading /modules/java.datatransfer/module-info.class]
[loading /modules/jdk.crypto.ec/module-info.class]
[loading /modules/jdk.unsupported/module-info.class]
[loading /modules/jdk.jconsole/module-info.class]
[loading /modules/java.logging/module-info.class]
[loading /modules/java.sql/module-info.class]
[loading /modules/jdk.charsets/module-info.class]
[loading /modules/jdk.httpserver/module-info.class]
[loading /modules/jdk.jshell/module-info.class]
[loading /modules/jdk.javadoc/module-info.class]
[loading /modules/jdk.security.auth/module-info.class]
[loading /modules/java.transaction.xa/module-info.class]
[loading /modules/jdk.random/module-info.class]
[loading /modules/jdk.editpad/module-info.class]
[loading /modules/java.instrument/module-info.class]
[loading /modules/jdk.jdwp.agent/module-info.class]
[loading /modules/jdk.sctp/module-info.class]
[loading /modules/jdk.dynalink/module-info.class]
[loading /modules/jdk.jdi/module-info.class]
[loading /modules/jdk.incubator.vector/module-info.class]
[loading /modules/java.xml/module-info.class]
[loading /modules/jdk.accessibility/module-info.class]
[loading /modules/jdk.jartool/module-info.class]
[loading /modules/java.compiler/module-info.class]
[loading /modules/java.management.rmi/module-info.class]
[loading /modules/jdk.jsobject/module-info.class]
[loading /modules/java.scripting/module-info.class]
[loading /modules/jdk.net/module-info.class]
[loading /modules/jdk.management/module-info.class]
[loading /modules/jdk.naming.dns/module-info.class]
[loading /modules/java.sql.rowset/module-info.class]
[loading /modules/java.net.http/module-info.class]
[loading /modules/jdk.jpackage/module-info.class]
[loading /modules/java.se/module-info.class]
[loading /modules/jdk.management.agent/module-info.class]
[loading /modules/jdk.jcmd/module-info.class]
[loading /modules/java.security.sasl/module-info.class]
[loading /modules/jdk.internal.vm.compiler.management/module-info.class]
[loading /modules/jdk.nio.mapmode/module-info.class]
[loading /modules/java.desktop/module-info.class]
[loading /modules/jdk.jlink/module-info.class]
[loading /modules/jdk.jstatd/module-info.class]
[loading /modules/jdk.localedata/module-info.class]
[loading /modules/jdk.internal.vm.compiler/module-info.class]
[loading /modules/jdk.naming.rmi/module-info.class]
[loading /modules/jdk.management.jfr/module-info.class]
[loading /modules/jdk.attach/module-info.class]
[loading /modules/java.management/module-info.class]
[loading /modules/jdk.hotspot.agent/module-info.class]
[loading /modules/jdk.internal.opt/module-info.class]
[loading /modules/jdk.internal.ed/module-info.class]
[loading /modules/java.naming/module-info.class]
[loading /modules/java.rmi/module-info.class]
[loading /modules/jdk.jfr/module-info.class]
[loading /modules/jdk.xml.dom/module-info.class]
[loading /modules/jdk.zipfs/module-info.class]
```

图二： `javac -verbose` 的输出结果

从`help`信息中得知这个参数的作用是输出编译器的编译信息，从这个结果可以看出编译器在编译时在不断的加载源代码所需的模块。

使用`javac -d "/Users/shenye/Desktop"  Person.java`可以将编译后的`.class`文件输出到你指定的位置。

使用`javac -g  Person.java`输出所有调试信息，这个功能最好与IDE一起使用，在IDE里设置断点，接着使用这条指令就会看到断点位置，上下文。

`-nowarn`和`-Werror`， 第一个参数作用是编译时不生成任何警告，第二个作用是出现警告时终止编译。

使用`javac -encoding utf8  Person.java`来 指定源文件使用的字符编码

Clang

因为MacOS自带`clang`,所以环境不需要配置。

这是接下来要使用的代码:
`temp.c:`

```
#include<stdio.h>

int main(void){
int sum = 0;
for(int i = 0;i < 1000;i++){
        sum += i;
}
printf("The sum of 0..1000 is %d\n",sum);
}
```

使用`clang -help`:

```
where possible options include:
  @<filename>                Read options and filenames from file
  -Akey[=value]              Options to pass to annotation processors
  --add-modules <module>(,<module>)*
        Root modules to resolve in addition to the initial modules,
        or all modules on the module path if <module> is ALL-MODULE-PATH.
  --boot-class-path <path>, -bootclasspath <path>
        Override location of bootstrap class files
  --class-path <path>, -classpath <path>, -cp <path>
        Specify where to find user class files and annotation processors
  -d <directory>             Specify where to place generated class files
  -deprecation
        Output source locations where deprecated APIs are used
  --enable-preview
        Enable preview language features.
        To be used in conjunction with either -source or --release.
  -encoding <encoding>       Specify character encoding used by source files
  -endorseddirs <dirs>       Override location of endorsed standards path
  -extdirs <dirs>            Override location of installed extensions
  -g                         Generate all debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -g:none                    Generate no debugging info
  -h <directory>
        Specify where to place generated native header files
  --help, -help, -?          Print this help message
  --help-extra, -X           Print help on extra options
  -implicit:{none,class}
        Specify whether to generate class files for implicitly referenced files
  -J<flag>                   Pass <flag> directly to the runtime system
  --limit-modules <module>(,<module>)*
        Limit the universe of observable modules
  --module <module>(,<module>)*, -m <module>(,<module>)*
        Compile only the specified module(s), check timestamps
  --module-path <path>, -p <path>
        Specify where to find application modules
  --module-source-path <module-source-path>
        Specify where to find input source files for multiple modules
  --module-version <version>
        Specify version of modules that are being compiled
  -nowarn                    Generate no warnings
  -parameters
        Generate metadata for reflection on method parameters
  -proc:{none,only,full}
        Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...]
        Names of the annotation processors to run;
        bypasses default discovery process
  --processor-module-path <path>
        Specify a module path where to find annotation processors
  --processor-path <path>, -processorpath <path>
        Specify where to find annotation processors
  -profile <profile>
        Check that API used is available in the specified profile.
        This option is deprecated and may be removed in a future release.
  --release <release>
        Compile for the specified Java SE release.
        Supported releases:
            8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
  -s <directory>             Specify where to place generated source files
  --source <release>, -source <release>
        Provide source compatibility with the specified Java SE release.
        Supported releases:
            8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
  --source-path <path>, -sourcepath <path>
        Specify where to find input source files
  --system <jdk>|none        Override location of system modules
  --target <release>, -target <release>
        Generate class files suitable for the specified Java SE release.
        Supported releases:
            8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
  --upgrade-module-path <path>
```

图一：`clang -help`的输出结果

这只是一页的结果，`clang`有太多的参数，在这里选取几个常用的进行尝试。

可以使用`clang -S -mllvm --x86-asm-syntax=intel temp.c`来生成对应的汇编代码，注意这里可以通过调整`--x86-asm-syntax`参数来生成`intel`风格或者`AT&T`：

```
.section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 13, 0     sdk_version 14, 0
        .globl  _main                           ; -- Begin
function main
        .p2align        2
_main:                                  ; @main
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #48
        .cfi_def_cfa_offset 48
        stp     x29, x30, [sp, #32]             ; 16-byte
Folded Spill
        add     x29, sp, #32
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
```

```
        stur    wzr, [x29, #-4]
        stur    wzr, [x29, #-8]
        stur    wzr, [x29, #-12]
        b       LBB0_1
LBB0_1:                                 ; =>This Inner Loop
Header: Depth=1
        ldur    w8, [x29, #-12]
        subs    w8, w8, #1000
        cset    w8, ge
        tbnz    w8, #0, LBB0_4
        b       LBB0_2
LBB0_2:                                 ;   in Loop:
Header=BB0_1 Depth=1
        ldur    w9, [x29, #-12]
        ldur    w8, [x29, #-8]
        add     w8, w8, w9
        stur    w8, [x29, #-8]
        b       LBB0_3
LBB0_3:                                 ;   in Loop:
Header=BB0_1 Depth=1
        ldur    w8, [x29, #-12]
        add     w8, w8, #1
        stur    w8, [x29, #-12]
        b       LBB0_1
LBB0_4:
        ldur    w9, [x29, #-8]
                                        ; implicit-def: $x8
        mov     x8, x9
        mov     x9, sp
        str     x8, [x9]
        adrp    x0, l_.str@PAGE
        add     x0, x0, l_.str@PAGEOFF
        bl      _printf
        ldur    w0, [x29, #-4]
        ldp     x29, x30, [sp, #32]             ; 16-byte
Folded Reload
        add     sp, sp, #48
        ret
        .cfi_endproc
                                        ; -- End function
        .section        __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
```

```
        .asciz  "The sum of 0..1000 is %d\n"

.subsections_via_symbols
```

我们可以在这里探究一下 `clang` 的代码优化效果。使用 `-o` 参数，后面跟着值：

- 0: 没有优化
- 1: 基本优化
- 2: 中等优化
- 3: 激进优化
- “s”: 对代码大小优化
  在这里我们使用 `clang -O2 -S -mllvm --x86asm-syntax=intel temp.c` 对源代码进行二级优化：

```
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 13, 0     sdk_version 14, 0
        .globl  _main                           ; -- Begin function main
        .p2align        2
_main:                                  ; @main
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #32
        .cfi_def_cfa_offset 32
        stp     x29, x30, [sp, #16]             ; 16-byte Folded Spill
        add     x29, sp, #16
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
        mov     w8, #40748
        movk    w8, #7, lsl #16
        str     x8, [sp]
Lloh0:
        adrp    x0, l_.str@PAGE
Lloh1:
        add     x0, x0, l_.str@PAGEOFF
        bl      _printf
        mov     w0, #0
        ldp     x29, x30, [sp, #16]             ; 16-byte Folded Reload
        add     sp, sp, #32
        ret
        .loh AdrpAdd    Lloh0, Lloh1
        .cfi_endproc
                                        ; -- End function
        .section        __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
        .asciz  "The sum of 0..1000 is %d\n"

.subsections_via_symbols
```

图二：优化后的汇编代码

可以看到汇编代码明显短了很多，没有经过优化的代码有两个循环，而这里优化后的代码并没有循环，编译器看到源代码的循环分析出这是在求和，于是直接相加并没有用循环。

我们来看一下反汇编代码会怎么样，有一个工具叫`objdump`可以对二进制文件进行反汇编，我们来看看它生成的汇编代码是怎样的，使用`objdump -d -M intel temp.exe`：

```
temp.exe:       file format mach-o arm64

Disassembly of section __TEXT,__text:

0000000100003f00 <_main>:
100003f00: d100c3ff     sub     sp, sp, #48
100003f04: a9027bfd     stp     x29, x30, [sp, #32]
100003f08: 910083fd     add     x29, sp, #32
100003f0c: b81fc3bf     stur    wzr, [x29, #-4]
100003f10: b81f83bf     stur    wzr, [x29, #-8]
100003f14: b81f43bf     stur    wzr, [x29, #-12]
100003f18: 14000001     b       0x100003f1c <_main+0x1c>
100003f1c: b85f43a8     ldur    w8, [x29, #-12]
100003f20: 710fa108     subs    w8, w8, #1000
100003f24: 1a9fb7e8     cset    w8, ge
100003f28: 37000168     tbnz    w8, #0, 0x100003f54 <_main+0x54>
100003f2c: 14000001     b       0x100003f30 <_main+0x30>
100003f30: b85f43a9     ldur    w9, [x29, #-12]
100003f34: b85f83a8     ldur    w8, [x29, #-8]
100003f38: 0b090108     add     w8, w8, w9
100003f3c: b81f83a8     stur    w8, [x29, #-8]
100003f40: 14000001     b       0x100003f44 <_main+0x44>
100003f44: b85f43a8     ldur    w8, [x29, #-12]
100003f48: 11000508     add     w8, w8, #1
100003f4c: b81f43a8     stur    w8, [x29, #-12]
100003f50: 17fffff3     b       0x100003f1c <_main+0x1c>
100003f54: b85f83a9     ldur    w9, [x29, #-8]
100003f58: aa0903e8     mov     x8, x9
100003f5c: 910003e9     mov     x9, sp
100003f60: f9000128     str     x8, [x9]
100003f64: 90000000     adrp    x0, 0x100003000 <_main+0x64>
100003f68: 913e3000     add     x0, x0, #3980
100003f6c: 94000005     bl      0x100003f80 <_printf+0x100003f80>
100003f70: b85fc3a0     ldur    w0, [x29, #-4]
100003f74: a9427bfd     ldp     x29, x30, [sp, #32]
100003f78: 9100c3ff     add     sp, sp, #48
100003f7c: d65f03c0     ret

Disassembly of section __TEXT,__stubs:

0000000100003f80 <__stubs>:
100003f80: b0000010     adrp    x16, 0x100004000 <__stubs+0x4>
100003f84: f9400210     ldr     x16, [x16]
100003f88: d61f0200     br      x16
```

图三：反汇编的代码

可以看出虽然源代码只是一个很简单的函数，反汇编后的代码依然与前两个代码不同。

像`javac`一样，`-g`符号是生成调试信息，这个需要设置断点等信息，推荐使用`GDB`对C代码调试，非常方便。

还有一些其他参数：

- `-Werror`：把警告当成错误
- `Wall`：输出警告信息
- `-arch` ：选择目标架构，不同架构的芯片对应的指令集不一样。
- `-std`：指定语言标准，一些新语法会在旧标准里报错

最常用的编译命令是：`clang -Wall your_source.c -o your_binary`.

使用`clang -S -emit-llvm temp.c -o temp.ll`指令生成LLVM IR代码：

```
; ModuleID = 'temp.c'
source_filename = "temp.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"  目标数据格式
target triple = "arm64-apple-macosx13.0.0"  目标架构和系统

@.str = private unnamed_addr constant [26 x i8] c"The sum of 0..1000 is %d\0A\00", align 1
  全局字符串常量
; Function Attrs: noinline nounwind optnone ssp uwtable(sync)    函数特性
define i32 @main() #0 {      定义main函数
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 0, ptr %2, align 4
  store i32 0, ptr %3, align 4
  br label %4

4:                                                ; preds = %11, %0
  %5 = load i32, ptr %3, align 4
  %6 = icmp slt i32 %5, 1000
  br i1 %6, label %7, label %14

7:                                                ; preds = %4
  %8 = load i32, ptr %3, align 4
  %9 = load i32, ptr %2, align 4
  %10 = add nsw i32 %9, %8
  store i32 %10, ptr %2, align 4
  br label %11

11:                                               ; preds = %7
  %12 = load i32, ptr %3, align 4
  %13 = add nsw i32 %12, 1
  store i32 %13, ptr %3, align 4
  br label %4, !llvm.loop !6

14:                                               ; preds = %4
  %15 = load i32, ptr %2, align 4
  %16 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %15)
  %17 = load i32, ptr %1, align 4
  ret i32 %17
}

declare i32 @printf(ptr noundef, ...) #1    定义printf函数

attributes #0 = { noinline nounwind optnone ssp uwtable(sync) "frame-pointer"="non-leaf" "min-legal-
vector-width"="0" "no-trapping-math"="true" "probe-stack"="__chkstk_darwin" "stack-protector-buffer-
size"="8" "target-cpu"="apple-m1" "target-features"="+aes,+crc,+crypto,+dotprod,+fp-armv8,+fp16fml,
+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+sm4,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+zcm,
+zcz" }                  特性0列表
attributes #1 = { "frame-pointer"="non-leaf" "no-trapping-math"="true" "probe-
stack"="__chkstk_darwin" "stack-protector-buffer-size"="8" "target-cpu"="apple-m1" "target-
features"="+aes,+crc,+crypto,+dotprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,
+sha2,+sha3,+sm4,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+zcm,+zcz" }
                  特性1列表
!llvm.module.flags = !{!0, !1, !2, !3, !4}  模块标志和标识符
!llvm.ident = !{!5}
```

"Screenshot 2023-11-30 at 20.01.04.png" is not created yet. Click to create.

图四：LLVM IR代码

生成的`.ll`文件结构已经在图片中标出，下面对一些图中值得注意的点进行讲解：

1. 定义函数
   在图片中我们可以看到，使用了`define 132 @main() #0 {}`定义了`main`函数：
   - `define`是关键字
   - `i32`是返回值类型
   - `@<name>`是函数名
   - `()`括号里的是函数参数列表
   - `#<number>`是函数的特性号，在文件下面会再次列出来详细版本
2. 特性列表：
   因为函数特性太多，这里挑几个特性进行讲解：

- `noinline`: 没有内联，内联是用函数本体替代使用
- `nounwind`: 不抛出异常
- `optnone`: 不优化代码
- `ssp`: 这个表明编译器应用了栈保护，防止栈溢出

下面我们来了解一下`LLVM IR`语法：

1. 数据类型： 有不同大小的整数, 浮点数, 指针, 向量, 数组，还有代码自己定义的类型
2. 变量：变量基本上用寄存器代表，寄存器是指令所使用的暂时存储位置，寄存器名字一般是：`%1`,`%2`... 跟上图一样
3. 指令：一般是低等级的类似汇编的指令，如：`sub`，`add`，`mul`...
4. 函数：上面讲过了，用`define`进行声明
5. 全局变量：用`@`进行声明
6. 注释：跟汇编一样用`;`

我们知道分析代码分为静态分析和动态分析,而`clang`两个分析的参数都有，我们可以组合这两种分析，完成对代码的检查：

```
clang  -fsanitize=address -coverage <your_program.c> -o
<your_program>
```

这个指令检查内存错误和代码覆盖，当你遇到`page fault`而不知道哪里错误的时候，不妨试试这个。

LLVM

配置环境：

```
brew install llvm
```

下面是在按照官方教程的指令得出的结果，所用源代码还是上面的`temp.c`：

```
% lli temp.bc
The sum of 0..1000 is 499500

% llvm-dis < temp.bc | less

; ModuleID = '<stdin>'
source_filename = "temp.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx13.0.0"
; ModuleID = '<stdin>'
@.str = private unnamed_addr constant [26 x i8] c"The sum of
0..1000 is %d\0A\00", align 1
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
; Function Attrs: nofree nounwind ssp uwtable(sync)
define i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 (ptr, ...) @printf(ptr noundef nonnull
dereferenceable(1) @.str, i32 noundef 499500)
  ret i32 0
}
; Function Attrs: nofree nounwind ssp uwtable(sync)
; Function Attrs: nofree nounwindaddr #0 {
declare noundef i32 @printf(ptr nocapture noundef readonly,
...) local_unnamed_addr #1
.str, i32 noundef 499500)
attributes #0 = { nofree nounwind ssp uwtable(sync) "frame-
pointer"="non-leaf" "min-legal-vector-width"="0" "no-trapping-
```

```
math"="true" "probe-stack"="__chkstk_darwin" "stack-protector-
buffer-size"="8" "target-cpu"="apple-m1" "target-
features"="+aes,+crc,+crypto,+dotprod,+fp-
armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha
3,+sm4,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+zcm,+zcz" }

attributes #1 = { nofree nounwind "frame-pointer"="non-leaf"
"no-trapping-math"="true" "probe-stack"="__chkstk_darwin"
"stack-protector-buffer-size"="8" "target-cpu"="apple-m1"
"target-features"="+aes,+crc,+crypto,+dotprod,+fp-
armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha
3,+sm4,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+zcm,+zcz" }

declare noundef i32 @printf(ptr nocapture noundef readonly,
...) local_unnamed_a!llvm.module.flags = !{!0, !1, !2, !3, !4}

!llvm.ident = !{!5}

attributes #0 = { nofree nounwind ssp uwtable(sync) "frame-
pointer"="non-leaf" "!0 = !{i32 2, !"SDK Version", [2 x i32]
[i32 14, i32 0]}robe-stack"="__chkstk_da!1 = !{i32 1,
!"wchar_size", i32 4}="8" "target-cpu"="apple-m1" "target-
features!2 = !{i32 8, !"PIC Level", i32
2}armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc!3 = !{i32 7,
!"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 1}
!5 = !{!"Apple clang version 15.0.0 (clang-1500.0.40.1)"}

% llc temp.bc -o temp.s

% gcc temp.s -o temp.native

% ./temp.native
The sum of 0..1000 is 499500
```

下面来尝试其他工具.

我们先来试试`opt`工具，这个工具是用来优化中间代码的。先使用`opt -O2 -S temp.ll -o tempp.ll`语句对中间代码进行优化，再使用`llc`工具生成汇编代码，这个工具可以将中间代码转换为汇编。使用`llc -march=x86 tempp.ll -o tempp.s`.结果如下，很奇怪的一点是，似乎并没有优化，还是两个循环。

```
        .build_version macos, 13, 0
        .globl  _main                           ; -- Begin
function main
        .p2align        2
_main:                                  ; @main
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #48
        stp     x29, x30, [sp, #32]             ; 16-byte
Folded Spill
        add     x29, sp, #32
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
        stur    wzr, [x29, #-4]
        stur    wzr, [x29, #-8]
        stur    wzr, [x29, #-12]
LBB0_1:                                 ; =>This Inner Loop
Header: Depth=1
        ldur    w8, [x29, #-12]
        cmp     w8, #1000
        b.ge    LBB0_4
; %bb.2:                                ;   in Loop:
Header=BB0_1 Depth=1
        ldur    w8, [x29, #-12]
        ldur    w9, [x29, #-8]
        add     w8, w9, w8
        stur    w8, [x29, #-8]
; %bb.3:                                ;   in Loop:
Header=BB0_1 Depth=1
        ldur    w8, [x29, #-12]
        add     w8, w8, #1
        stur    w8, [x29, #-12]
        b       LBB0_1
LBB0_4:
        ldur    w8, [x29, #-8]
        str     x8, [sp]
        adrp    x0, l_.str@PAGE
        add     x0, x0, l_.str@PAGEOFF
        bl      _printf
        ldur    w0, [x29, #-4]
```

```
        ldp     x29, x30, [sp, #32]             ; 16-byte
Folded Reload
        add     sp, sp, #48
        ret
        .cfi_endproc
                                        ; -- End function
        .section        __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
        .asciz  "The sum of 0..1000 is %d\n"

.subsections_via_symbols
```

我还写了一个小工具能够将中间代码转换为`LLVM IR`，但其实更好的方法是直接生成`LLVM IR`.

项目三

例1

因为`open64`不对`apple m1`芯片支持，我选择使用`gcc`和`clang`,做出相同的实验，首先我们选择一个复杂一点的源代码，这里我选择了快排：

```c
#include <stdio.h>

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
```

```
    arr[high] = temp;

    return i + 1;
}

void quicksort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array and get the pivot index
        int pivot = partition(arr, low, high);

        // Recursively sort the subarrays
        quicksort(arr, low, pivot - 1);
        quicksort(arr, pivot + 1, high);
    }
}

int main() {
    int arr[] = {12, 4, 5, 6, 7, 3, 1, 15, 8, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    quicksort(arr, 0, n - 1);

    printf("\nSorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

下面进行各种优化等级，并进行对比。优化时间的获取使用`mac`自带的`time`指令。

	o1 time	o1 size	o2 time	o2 size	o3 time	o3 size
gcc	0.386	37kb	0.281	35kb	0.286	34kb

	o1 time	o1 size	o2 time	o2 size	o3 time	o3 size
clang	1.183	37kb	0.328	34kb	0.101	34kb

可以看出因为我的源代码还不太复杂，没有很多冗余的代码，所以不管启用多大等级，生成的可执行文件大小并没有太大的变化，而对于时间来说。优化等级越高，时间似乎越少，我的猜测是优化等级越高，编译器可以采取更激进的优化，而不会顾虑改变了源代码的一些东西，比如反汇编后可能根本不知道这个代码是干什么的。

我们还可以继续探究编译器的尾递归优化，利用一下代码，一个简单的阶乘：

```c
int f(int n, int accumulator) {
    if (n == 0) {
        return accumulator;
    } else {
        return f(n - 1, n * accumulator);
    }
}
```

使用 `clang` 生成中间代码：

```
; ModuleID = 'tail.c'
source_filename = "tail.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx13.0.0"

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable(sync)
define i32 @f(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, ptr %4, align 4
  store i32 %1, ptr %5, align 4
  %6 = load i32, ptr %4, align 4
  %7 = icmp eq i32 %6, 0
  br i1 %7, label %8, label %10

8:                                                ; preds = %2
  %9 = load i32, ptr %5, align 4
  store i32 %9, ptr %3, align 4
  br label %17

10:                                               ; preds = %2
  %11 = load i32, ptr %4, align 4
  %12 = sub nsw i32 %11, 1
  %13 = load i32, ptr %4, align 4
  %14 = load i32, ptr %5, align 4
  %15 = mul nsw i32 %13, %14
  %16 = call i32 @f(i32 noundef %12, i32 noundef %15)
  store i32 %16, ptr %3, align 4
  br label %17

17:                                               ; preds = %10, %8
  %18 = load i32, ptr %3, align 4
  ret i32 %18
}

; Function Attrs: noinline nounwind optnone ssp uwtable(sync)
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = call i32 @f(i32 noundef 100, i32 noundef 1)
  store i32 %2, ptr %1, align 4
  %3 = load i32, ptr %1, align 4
  %4 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %3)
  ret i32 0
}

declare i32 @printf(ptr noundef, ...) #1

attributes #0 = { noinline nounwind optnone ssp uwtable(sync) "frame-pointer"="non-leaf" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="apple-m1" "target-features"="+aes,+crc,+dotprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+zcm,+zcz" }
attributes #1 = { "frame-pointer"="non-leaf" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="apple-m1" "target-features"="+aes,+crc,+dotprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+zcm,+zcz" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 1}
!3 = !{i32 7, !"frame-pointer", i32 1}
!4 = !{!"Homebrew clang version 17.0.6"}
```

可以看到在 `main` 函数直接使用了递归，没有进行任何优化，下面使用 `-O2` 优化等级：

```
; Function Attrs: nofree norecurse nosync nounwind ssp memory(none) uwtable(sync)
define i32 @f(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = icmp eq i32 %0, 0
  br i1 %3, label %45, label %4

4:                                                ; preds = %2
  %5 = icmp ult i32 %0, 16
  br i1 %5, label %36, label %6

6:                                                ; preds = %4
  %7 = and i32 %0, -16
  %8 = and i32 %0, 15
  %9 = insertelement <4 x i32> <i32 poison, i32 1, i32 1, i32 1>, i32 %1, i64 0
  %10 = insertelement <4 x i32> poison, i32 %0, i64 0
  %11 = shufflevector <4 x i32> %10, <4 x i32> poison, <4 x i32> zeroinitializer
  %12 = add <4 x i32> %11, <i32 0, i32 -1, i32 -2, i32 -3>
  br label %13

13:                                               ; preds = %13, %6
  %14 = phi i32 [ 0, %6 ], [ %27, %13 ]
  %15 = phi <4 x i32> [ %9, %6 ], [ %23, %13 ]
  %16 = phi <4 x i32> [ <i32 1, i32 1, i32 1, i32 1>, %6 ], [ %24, %13 ]
  %17 = phi <4 x i32> [ <i32 1, i32 1, i32 1, i32 1>, %6 ], [ %25, %13 ]
  %18 = phi <4 x i32> [ <i32 1, i32 1, i32 1, i32 1>, %6 ], [ %26, %13 ]
  %19 = phi <4 x i32> [ %12, %6 ], [ %28, %13 ]
  %20 = add <4 x i32> %19, <i32 -4, i32 -4, i32 -4, i32 -4>
  %21 = add <4 x i32> %19, <i32 -8, i32 -8, i32 -8, i32 -8>
  %22 = add <4 x i32> %19, <i32 -12, i32 -12, i32 -12, i32 -12>
  %23 = mul <4 x i32> %15, %19
  %24 = mul <4 x i32> %16, %20
  %25 = mul <4 x i32> %17, %21
  %26 = mul <4 x i32> %18, %22
  %27 = add nuw i32 %14, 16
  %28 = add <4 x i32> %19, <i32 -16, i32 -16, i32 -16, i32 -16>
  %29 = icmp eq i32 %27, %7
  br i1 %29, label %30, label %13, !llvm.loop !5

30:                                               ; preds = %13
  %31 = mul <4 x i32> %24, %23
  %32 = mul <4 x i32> %25, %31
  %33 = mul <4 x i32> %26, %32
  %34 = tail call i32 @llvm.vector.reduce.mul.v4i32(<4 x i32> %33)
  %35 = icmp eq i32 %7, %0
  br i1 %35, label %45, label %36

36:                                               ; preds = %4, %30
  %37 = phi i32 [ %1, %4 ], [ %34, %30 ]
  %38 = phi i32 [ %0, %4 ], [ %8, %30 ]
  br label %39

39:                                               ; preds = %36, %39
  %40 = phi i32 [ %43, %39 ], [ %37, %36 ]
  %41 = phi i32 [ %42, %39 ], [ %38, %36 ]
  %42 = add nsw i32 %41, -1
  %43 = mul nsw i32 %40, %41
  %44 = icmp eq i32 %42, 0
  br i1 %44, label %45, label %39, !llvm.loop !8


45:                                               ; preds = %39, %30, %2
  %46 = phi i32 [ %1, %2 ], [ %34, %30 ], [ %43, %39 ]
  ret i32 %46
}

; Function Attrs: nofree nounwind ssp uwtable(sync)
define i32 @main() local_unnamed_addr #1 {
  %1 = tail call i32 (ptr, ...) @printf(ptr noundef nonnull dereferenceable(1) @.str, i32 noundef 0)
  ret i32 0
}
```

可以看到这里明显出现了尾递归优化，代码里有了循环结构，将递归改成了循环。虽然整个代码体积增大，但栈上情况肯定会小很多。

还有一条启示是似乎这两种编译器的优化路线是相似的，代码体积都会随着优化等级提高而减小，没有书上`open64`所出现的那种情况。

评价：

- 这个实验不足以反映编译器的性能，测试样例太少，测试样本单调。正确方法，使用蒙特卡洛法则，使用不同大小，不同功能的足够多的样本，随后进行统计，可以预想到的是，不同编译器优化所擅长的代码类型必定不相同，所以要给数据集分类，最后得出结果。

例2

为了测量时间，我将三个主要函数暂时集成到第一个函数里，并在第一个函数开始执行之前，添加`console.time('myCode');`,在最后一个函数执行完之后，使用`console.timeEnd('myCode');`,这样就可以得到从源代码到中间代码所需的时间。

首先还是使用`temp.c`,不过把打印函数删除。

```
int $sum$, $i$;
$sum$ := 0;
$i$ := 0;

while($i$ < 1000):{
        $sum$ = $sum$ + $i$;
        $i$ = $i$ + 1;
}
```

测量`clang`时间和上面相同：`time clang -S -emit-llvm temp.c -o temp.ll`.

接着我又添加了一个无用的循环，里面声明了两个变量.

接下来我想用一个最简单的源代码，声明六个变量，并赋值：

```
int a,b;
char *c, *d;
a = 1;
```

```
b = 2;
c = "asd";
d = "fgh";
```

```
int $a$,$b$;
string $c$,$d$;
$a$ := 1;
$b$ := 2;
$c$ := "asd";
$d$ := "fgh";
```

以下所有实验结果均是实验五次并且取平均数的结果：

	time1	size 1	time 2	size 2	time3	size 3
my compiler	1.0009765625 ms	195 bytes	1.00098013575 ms	201 bytes	2.08ms	229 bytes
clang	194ms	1728 bytes	348.6 ms	1857 bytes	340 ms	1204 bytes

可以看到明显`clang`编译器比我的编译器要慢很多，生成文件也要大很多，这里我的C代码没有链接任何库，没有这个因素干扰。有很多可能的原因：

- C代码语言定义非常复杂，即使是相同的代码，所需要的时间也会相差很多，这个非常有可能。
- 相同功能的源代码，在两种定义下所生成的中间代码并不一样，因为C代码复杂，相应的中间代码需要更多的位数去编码
- 在编译器前端阶段，相比简单的三阶段，`clang`完成了更多的事，1 Introduction.t同时为了提高前端的性能，编译器通常采用复杂的算法、数据结构和优化技术。这会使得所花费的时间大大增加

例三

下面使用工具`compilerRun`来把中间代码翻译成可执行文件，而测量时间依然使用自带的`time`工具。

结果如下：

	time1	size 1	time 2	size 2
my compiler	0.071	23500	0.065	21532
clang	0.078	33432	0.075	31578

出现这种情况的原因：

- 在时间上没有出现相差较大的现象，可能是编译器后端只是把中间代码翻译成机器吗，而用来测试的代码非常简单，还没有使用到栈管理等一些更复杂的阶段，所以时间上差不多。
- 大小有明显差异，