第五章　程序插桩技术

蔡彦

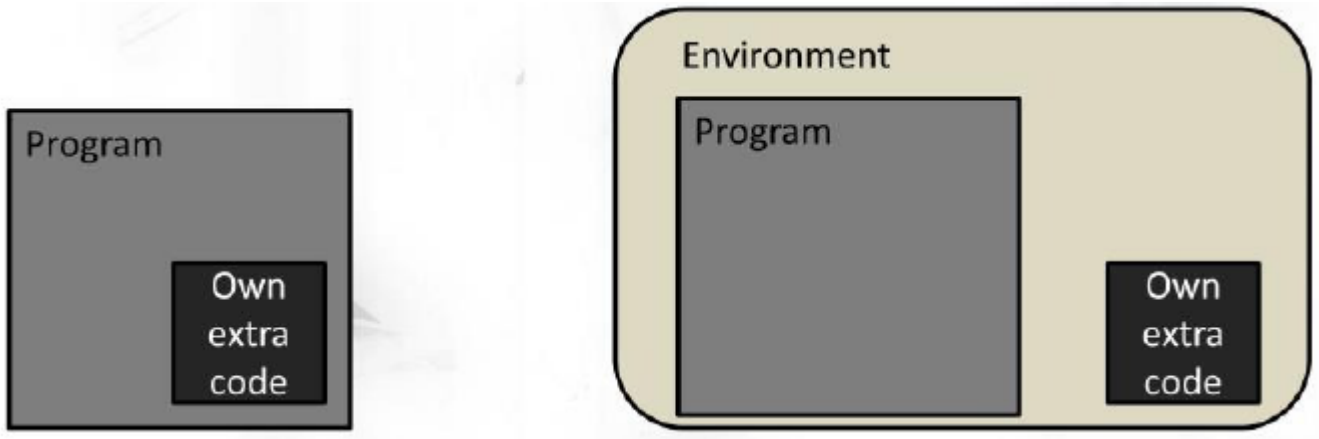
计算机科学国家重点实验室
中国科学院软件研究所

yancai@ios.ac.cn

目录

1.程序插桩技术

- 基本概念
  - **插桩（Instrumentation）**是一种用于动态软件测试等领域方法
  - 向目标程序或运行环境插入特定的代码片段（称为**桩**、**探针**）
  - 程序运行时通过探针的执行获取程序关键信息



https://www.slideserve.com/rhiannon/light-and-dark-side-of-code-instrumentation

1.程序插桩技术

- 实例：
  - 调试程序时，在程序中的某个点位插入一些输出语句，查看程序的运行状态
  - 本质上是手动插桩

```
1  Value *SrcValue = possibleValues.front();
2
3  printf("hit\n");
4
5  possibleValues.pop();
```

1.1 技术原理

- 基本原理
  - 通过插入代码片段（探针）将目标程序P变换为P′
  - 保证目标程序P的逻辑完整性
  - 运行时通过探针的执行获取目标程序P的运行时信息

基本原理

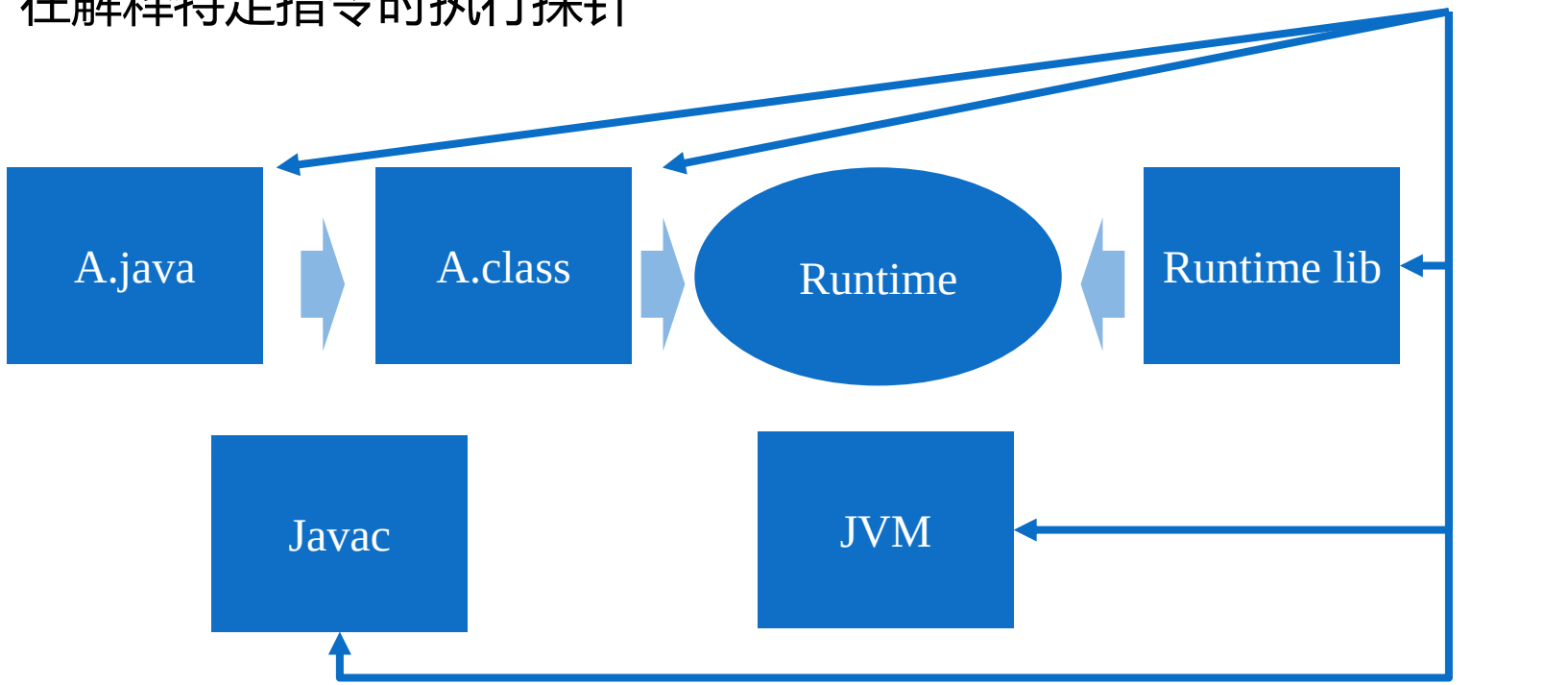
- 手动插桩效率低，在实际场景中不可行
- **程序插桩技术**的目的是实现**自动化**插桩
  - 按照**预先配置**的规则
  - 在程序**运行时**或**编译时**的不同阶段
  - 对所有满足要求的点位**自动**插入探针

基本原理

- 基本流程
  1. 确定需要收集的运行时信息
  2. 根据1，设计要插入的代码片段（探针），探针不能影响目标程序的逻辑完整性
  3. 在程序编译或运行时的不同阶段，选择合适的点位，插入2中设计的探针

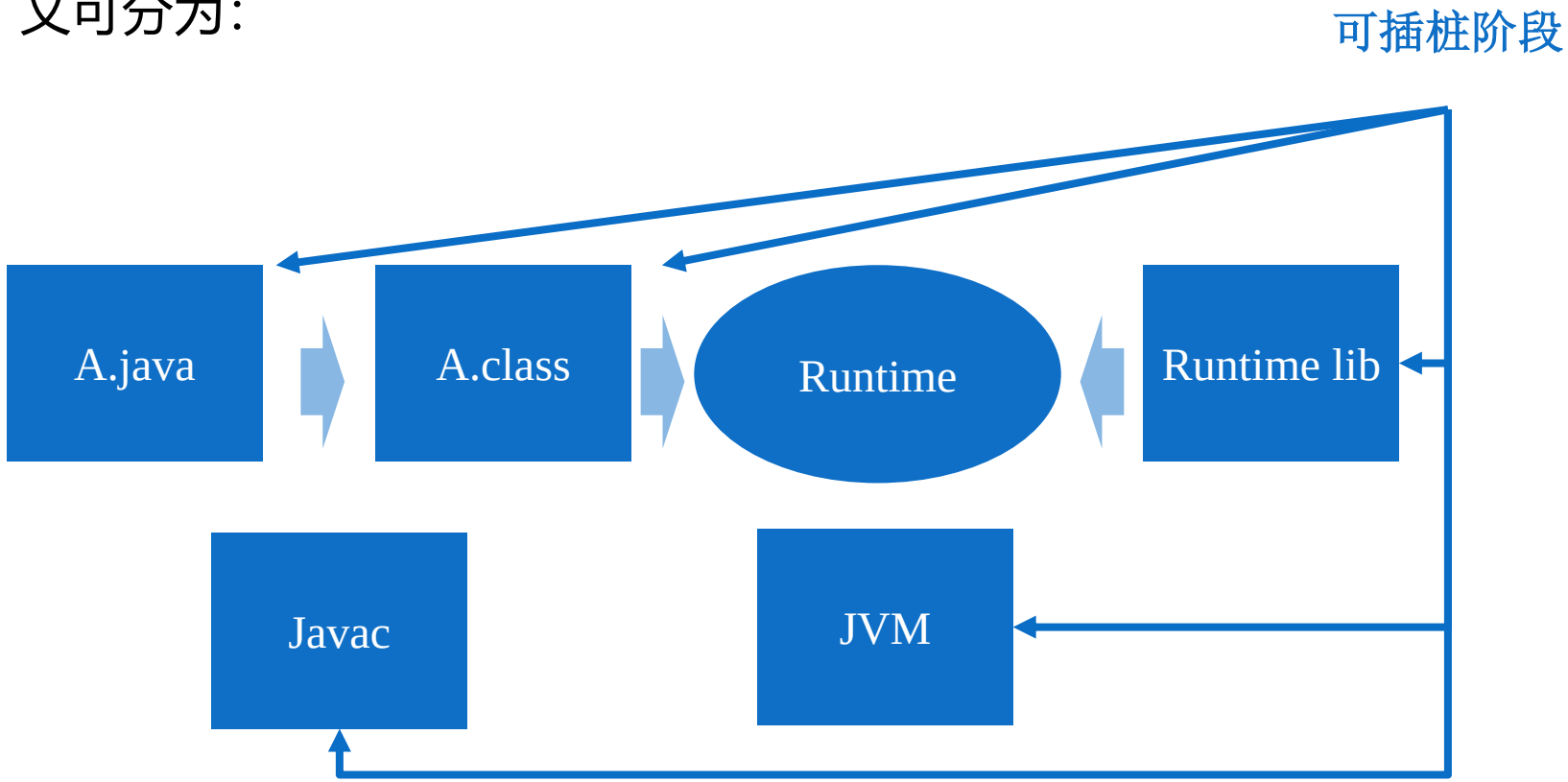
分类

- 按照插桩目标分类：
  - 直接插桩
    - 直接插桩目标程序，是最常用的插桩方式
  - 间接插桩（解释器）
    - 使用解释器调试选项，在解释特定指令时执行探针



分类

- 按照插桩阶段分类：
  - 静态插桩：
    - 在编译阶段进行插桩
    - 根据编译阶段的不同，又可分为：
      - 源码级
      - 中间表示(IR)级
      - 二进制级
  - 动态插桩
    - 在运行阶段进行插桩



分类

- 动态插桩（续）：
  - 一般用于二进制程序
  - 通过运行时插入二进制探针、指令覆盖等，进行插桩

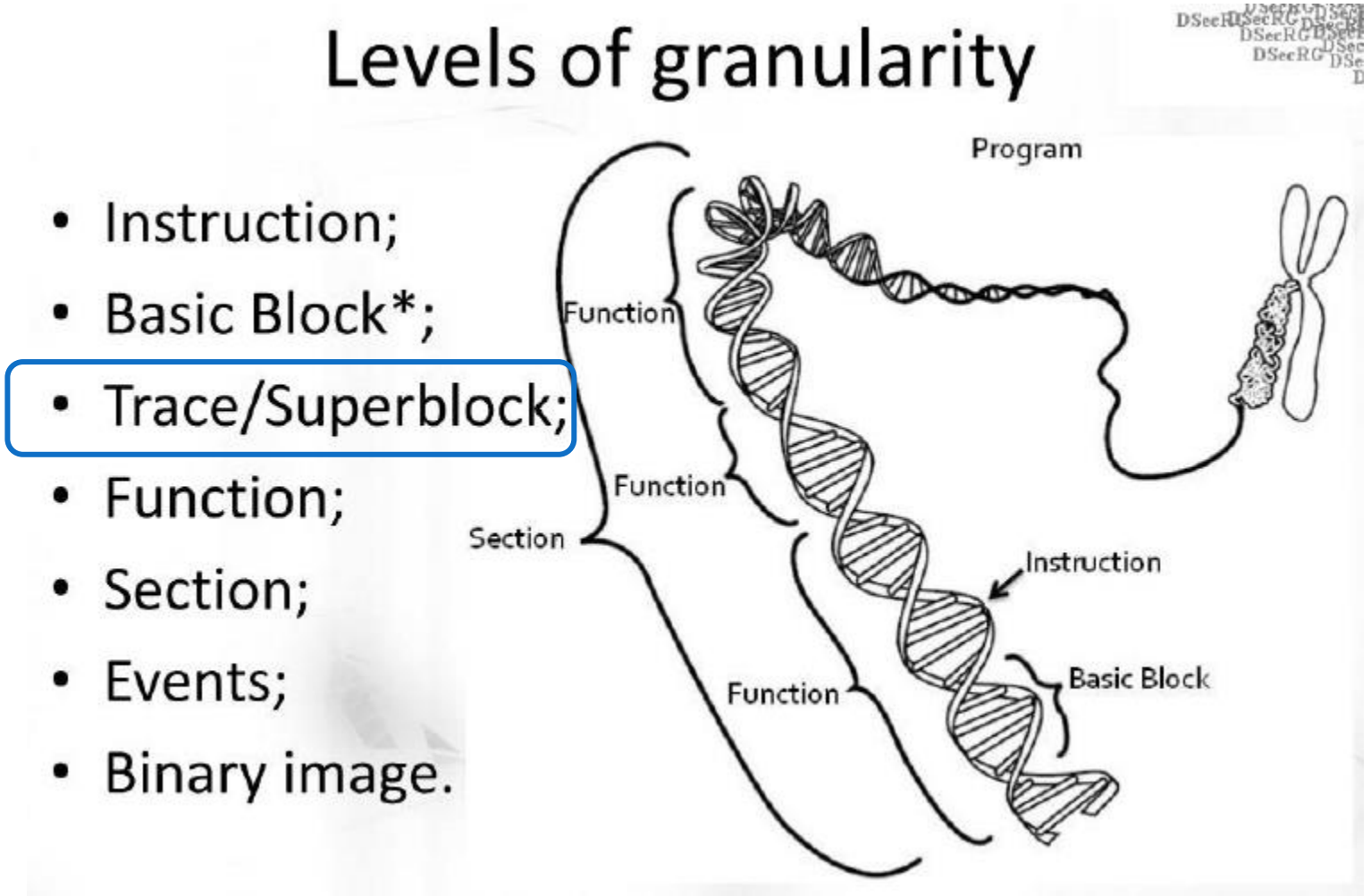
- 动态插桩可分为：
  - 即时模式(Just-In-Time mode)：
    - 执行一段二进制代码时
    - 借助Just-In-Time编译器，生成插桩过的二进制代码
    - 替代原始二进制代码执行

分类

- 解释模式(Interpretation mode)：
  - 执行一段二进制代码代码时
  - 在替换表中查找对应的替换规则
  - 如果存在，则根据替换规则执行替换后的二进制指令
  - 否则执行原始二进制指令
- 探测模式(Probe mode)：
  - 向程序中加入探针
  - 探针用于检测程序状态是否满足一定的条件
  - 条件满足时，探针将被触发，程序执行一段预定义好的二进制指令

分类

- 按插桩粒度分类：
  - 指令（语句）级
  - 基本块级
  - 函数级
  - 其他

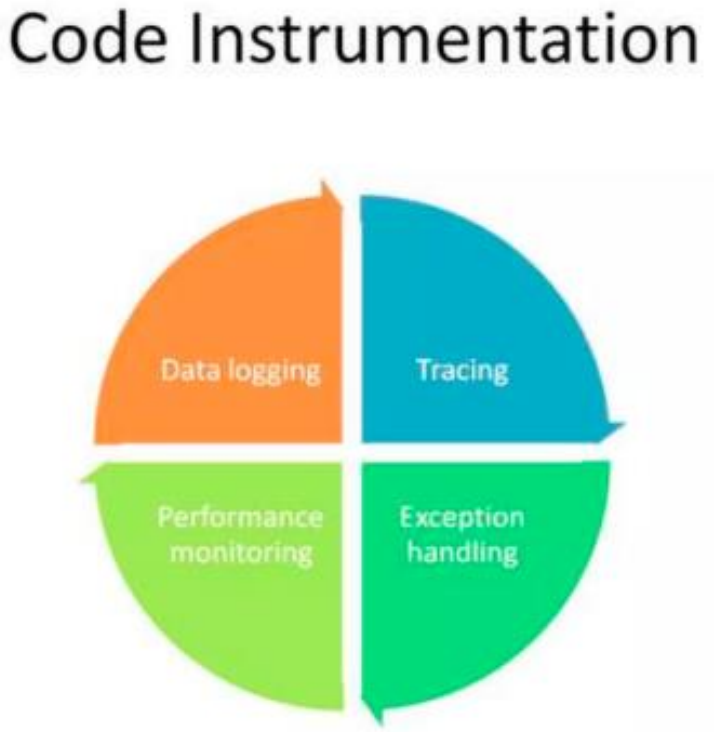


Traces usually begin at the target of a taken branch and end with an unconditional branch, including calls and returns.

https://www.slideserve.com/rhiannon/light-and-dark-side-of-code-instrumentation

分类

- 按插桩目的分类：
  - 控制流信息
    - 获取程序的分支覆盖情况
  - 数据流信息
    - 获取运行时变量的值
  - 性能评估
    - 评估程序性能（用时、内存等）
  - 异常处理
  - 其他



https://www.slideshare.net/mtekbir/code-instrumentation

简单示例

- 例子：在函数调用时打印提示信息

```
public void func1()
{
    int a;
    …
}
```

printf("function func1 is called");

- 对C/C++代码进行静态插桩
  - 使用clang进行源码插桩
  - 使用clang/llvm进行中间表示（IR）插桩
  - 直接对汇编码进行插桩

简单实例：源码插桩

- 使用clang进行源码插桩
- 编写clang tool

Clang Static Analyzer： *handleXXX(YYY)*

```
class MyASTConsumer : public ASTConsumer {
    public:
        MyASTConsumer(){}
        virtual bool HandleTopLevelDecl(DeclGroupRef DR) {
            for (DeclGroupRef::iterator b = DR.begin(), e = DR.end(); b != e; ++b) {
                if(FunctionDecl * funcDecl = dyn_cast<FunctionDecl> (*b)){
                    if(funcDecl->hasBody()){
                        if(CompoundStmt * bodyStmt =
                            dyn_cast<CompoundStmt> (funcDecl->getBody())){
                            string call_printf = string("\n\tprintf(\"function ") +
                                                 funcDecl->getName().str() +
                                                 string(" is called\");");
                            MyRewriter.InsertTextAfterToken(
                                bodyStmt->getBeginLoc(),
                                call_printf);
                        }
                    }
                }
            }
            return true;
        }
};
```

遍历所有函数定义

需要插入的printf语句

使用Rewriter，在函数体开始处插入目标语句

printf("function func1 is called");

```
class MyFrontendAction : public ASTFrontendAction {
    public:
        MyFrontendAction() {}
        std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI, StringRef file) override {
            MyLangOpts = CI.getLangOpts();
            ptrMySourceMgr= &(CI.getSourceManager());
            MyRewriter = Rewriter(*ptrMySourceMgr, MyLangOpts);
            return std::make_unique<MyASTConsumer>();
        }
        void EndSourceFileAction() override {
            const RewriteBuffer *RewriteBuf =
                MyRewriter.getRewriteBufferFor(ptrMySourceMgr -> getMainFileID());
            RewriteBuf->write(llvm::outs());
        }
};
```

初始化Rewriter

输出插桩后的代码

简单实例：源码插桩

- 插桩效果

```c
void f(){

}

int main(){
    f();
    return 0;
}
```

源文件

```c
void f(){
    printf("function f is called");

}

int main(){
    printf("function main is called");
    f();
    return 0;
}
```

插桩后源文件

简单实例：IR插桩

- 使用LLVM Pass进行IR插桩

- 需要插入的log函数：

```
void __funtion_call_log(const char * funcname){
    printf("function %s is called\n", funcname);
}
```

简单实例：IR插桩

- 编写LLVM Pass:

```
const char * LOG_CALL_STR = "__funtion_call_log";
struct Instru: public ModulePass {
    static char ID;
    Instru() : ModulePass(ID) {}
    bool runOnModule(Module &M) override {
        declare_log_functions(M);
        for(Function & F : M){
            if(!F.isIntrinsic() && !F.isDeclaration()){
                errs() << F.getName() << '\n';
                instrumentCall(M, F);
            }
        }
        return true;
    }
    void declare_log_functions(Module &M) {…
    void instrumentCall(Module &M, Function &f) {…
};
```

插入log函数的声明

在每个函数开始处插入log函数的调用

简单实例：IR插桩

- 插入log函数的声明

定义log函数的类型

```
void declare_log_functions(Module &M) {
    LLVMContext &C = M.getContext();
    Type *voidType = Type::getVoidTy(C);
    Type *StringType = Type::getInt8PtrTy(C);

    std::vector<Type*> callFunctionCallParams;
    callFunctionCallParams.push_back(StringType);
    FunctionType *callFunctionType = FunctionType::get(
        voidType, callFunctionCallParams, false
    );

    M.getOrInsertFunction(LOG_CALL_STR, callFunctionType);
}
```

向模块中插入
log函数的声明

```
void __funtion_call_log(const char * funcname){
    printf("function %s is called\n", funcname);
}
```

简单实例：IR插桩

- 插入log函数的调用

获取插入位置

获取log函数

设定传入参数

插入函数调用语句

```
void instrumentCall(Module &M, Function &f) {
    BasicBlock &entryBlock = f.getEntryBlock();
    Instruction *firstInstr = entryBlock.getFirstNonPHI();

    Function *logFunction = M.getFunction(LOG_CALL_STR);

    IRBuilder<> builder(firstInstr);
    Value *strPointer = builder.CreateGlobalStringPtr(f.getName());
    std::vector<Value *> args;
    args.push_back(strPointer);

    CallInst::Create(logFunction, args, "", firstInstr);
}
```

```
void __funtion_call_log(const char * funcname){
    printf("function %s is called\n", funcname);
}
```

简单实例：IR插桩

- 插桩效果：

```
; Function Attrs: noinline nounwind uwtable
define dso_local void @f() #0 {
entry:
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  call void @f()
  ret i32 0
}
```

IR

```
; Function Attrs: noinline nounwind uwtable
define dso_local void @f() #0 {
entry:
  call void @__funtion_call_log(i8* getelementptr inbound
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
entry:
  call void @__funtion_call_log(i8* getelementptr inbound
  %retval = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  call void @f()
  ret i32 0
}

declare void @__funtion_call_log(i8*)
```

插桩后IR

简单实例：汇编码插桩

- 要插入的代码片段

```
call_printf = [
    '    movabsq    $%s, %%rdi',
    '    callq    printf',
]
ro_section = '    .section    .rodata.str1.1,"aMS",@progbits,1'
str_section = [
    '    .type    %s,@object',
    '%s:',
    '    .asciz    \"%s\"',
    '    .size    %s, %d'
]
```

调用printf

rodata段开头

要打印的内容

```
for line in src_lines:
    if ':' in line and not (line.startswith('.') or line.startswith('#')):
        func_to_instrument = line.split(':')[0]
        instrumented_asm += line + '\n'
    elif func_to_instrument != None and 'movq\t%rsp, %rbp' in line:
        str_name = '.str' + str(funcCnt)

        instrumented_asm += line + '\n'
        instrumented_asm += ('\n'.join(call_printf)) % (str_name,) + '\n'

        log_str = 'function %s is called\\n' % (func_to_instrument, )

        str_sections.append(('\n'.join(str_section)) % \
                            (str_name, str_name, log_str, str_name, len(log_str) + 1))

        funcCnt += 1
        func_to_instrument = None
    elif '.section' in line and '.note.GNU-stack' in line:
        instrumented_asm += ro_section + '\n'
        for sec in str_sections:
            instrumented_asm += sec + '\n'
    else:
        instrumented_asm += line + '\n'
```

查找需要插桩的函数

在函数开头处插入printf的调用

准备好要打印的字符串

将要打印的字符串加入rodata段

简单实例：汇编码插

- 插桩效果：

```
f:
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
```

```
main:
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    movl    $0, -4(%rbp)
    callq   f
    xorl    %eax, %eax
    addq    $16, %rsp
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
```

汇编码

```
f:
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    movabsq     $.str0, %rdi
    callq   printf
    .cfi_def_cfa_register %rbp
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
```

```
main:
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    movabsq     $.str1, %rdi
    callq   printf
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    movl    $0, -4(%rbp)
    callq   f
    xorl    %eax, %eax
    addq    $16, %rsp
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
```

```
    .section    .rodata.str1.1,"aMS",@progbits,1
    .type   .str0,@object
str0:
    .asciz   "function f is called\n"
    .size   .str0, 23
    .type   .str1,@object
str1:
    .asciz   "function main is called\n"
    .size   .str1, 26
```

插桩后汇编码

1.2 常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA, Soot
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

1.2 常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA, Soot
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

编译后运行的程序

- 一个程序的编译过程可以划分为前端与后端两部分：
  - 前端把源代码翻译成中间表示 (IR)
  - 后端把IR编译成目标平台的机器码

- 经典的编译器（如gcc）
  - 在设计上没有显式地区分前端与后端，只提供端到端（输入源码，输出可执行文件）的编译服务。
  - 用户不需要知道它使用的IR是什么样的，它也不会暴露中间接口来让用户操作它的IR。
  - 从前端到后端，这些编译器的大量代码都是强耦合的。

编译后运行的程序

- 经典的编译器（续）：
  - 优势：为特定平台提供专属优化策略
  - 劣势：每当一种新语言或一个新的平台出现，就从头至尾地重新设计整个编译器
  - 如果有M种语言、N种目标平台，那么最坏情况下要实现 M*N 套编译器。这是很低效的。

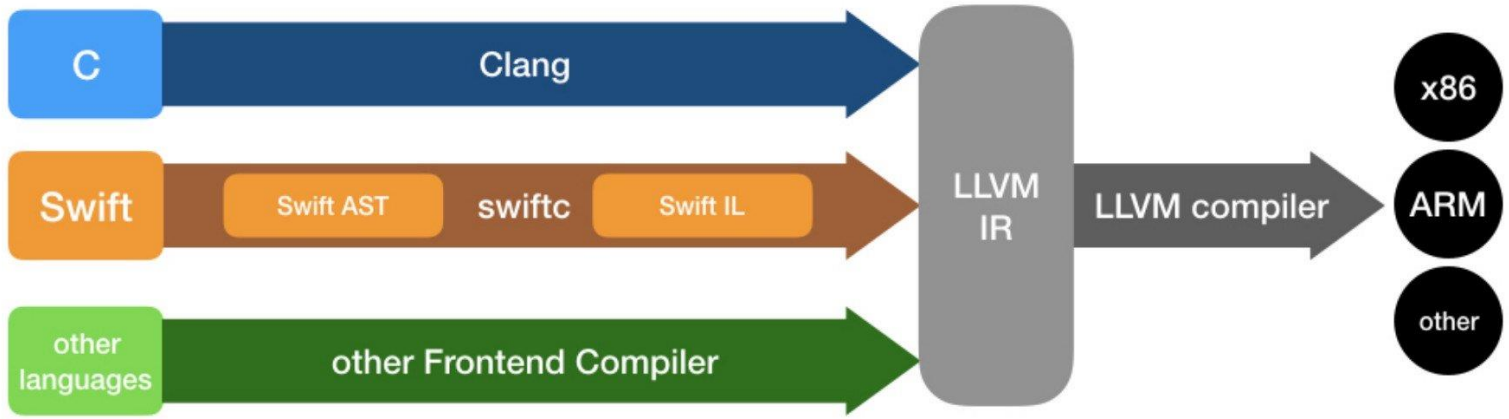
Clang/LLVM

- 如果大家都共用一种IR呢?
  - 每当增加一种新的语言，我们只需要重新设计一个从这个语言到IR的前端;
  - 每当增加一个新的目标平台，我们只需要重新设计一个从IR到这个目标平台的后端。
  - 如果有M种语言、N种目标平台，那么我们至多只需要实现 M+N 个前后端!

- LLVM就是这样一个项目。

Clang/LLVM

- LLVM的核心设计了一个叫 **LLVM IR** 的中间表示， 并以**库(Library)** 的方式提供一系列接口， 为用户提供诸如操作IR、生成目标平台代码等等后端的功能。
- Clang是一个基于LLVM的编译器驱动。它提供了把C/C++/OC等语言翻译成LLVM IR的前端，并使用LLVM的库实现了LLVM IR到目标平台的后端。



Clang/LLVM

- 实现了一系列周边工具，帮助开发者完成编译器后端的翻译任务：
  - **llvm-as**：把LLVM IR从人类能看懂的文本格式汇编翻转换成二进制格式。注意：此处得到的不是目标平台的机器码。
  - **llvm-dis**：llvm-as的逆过程，即反汇编。 不过这里的反汇编的对象是LLVM IR的二进制格式，而不是机器码。
  - **opt**：优化LLVM IR。输出新的LLVM IR。
  - **llc**：把LLVM IR编译成汇编码。需要用as进一步得到机器码。
  - **lli**：解释执行LLVM IR。

Clang/LLVM

- LLVM IR有三种表示：
  - **.ll** 格式：人类可以阅读的、汇编风格的文本。
  - **.bc** 格式：适合机器存储的二进制文件。
  - 内存表示：LLVM核心库中存储、访问LLVM IR的格式。

Clang/LLVM

- LLVM IR有三种表示：
  - .ll 格式：人类可以阅读的、汇编风格的文本。

```
; Function Attrs: mustprogress noinline norecurse optnone uwtable
define dso_local noundef i32 @main() #2 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 1234, ptr %2, align 4
  %4 = call i32 @_setjmp(ptr noundef @Jmpbuf) #6
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %8
6:                                                ; preds = %0
  call void @_Z5func1Pi(ptr noundef %2)
  %7 = call noundef i32 @_Z5func2Pi(ptr noundef %2)
  store i32 %7, ptr %3, align 4
  br label %8
```

Clang/LLVM

- .c格式源码文件、.ll格式IR和.bc格式IR相互之间的转换方式：
  - .c -> .ll： clang -emit-llvm -S a.c -o a.ll
  - .c -> .bc： clang -emit-llvm -c a.c -o a.bc
  - .ll -> .bc： llvm-as a.ll -o a.bc
  - .bc -> .ll： llvm-dis a.bc -o a.ll
  - .bc -> .s： llc a.bc -o a.s
- 通过向clang指定**-emit-llvm**参数， 使得原本要生成汇编以及机器码的指令生成了LLVM IR的.ll格式和.bc格式。
- 理解为：对于LLVM IR来说，.ll文件就相当于汇编，.bc文件就相当于机器码。

Clang/LLVM

- 在LLVM中，通常通过LLVM Pass对LLVM IR进行修改和变形。
- LLVM Pass：就是“遍历一遍IR，可以同时对它做一些操作”的意思。

- 在实现上，用户可以通过以下步骤设计、调用Pass，实现诸如优化、插桩、静态分析等等多种目的：
  - 继承由LLVM的核心库提供的一些基础Pass类；
  - 根据目的，修改并实现继承的Pass类的一些方法；
  - 使用LLVM的编译器调用自己设计的Pass类，遍历和修改LLVM IR，达成目的。

Clang/LLVM

```cpp
struct MyPass : public PassInfoMixin<MyPass>
{
  PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM)
  {
    if (F.getName() != "printHello")
    {
      FunctionCallee TestHello;
      auto &M = *F.getParent();
      AttributeList Attr;
      Attr = Attr.addFnAttribute(M.getContext(), Attribute::NoUnwind);
      IRBuilder<>
          IRB(M.getContext());

      // 查找符号名为printHello，返回值类型为void，参数列表类型为[void*]的函数，注册到TestHello中
      TestHello = F.getParent()->getOrInsertFunction("printHello", Attr, IRB.getVoidTy(), IRB.getInt8PtrTy());

      // 调整插入点至函数入口
      IRB.SetInsertPoint(&F.getEntryBlock(), F.getEntryBlock().getFirstInsertionPt());

      // 插入对IRB.getVoidTy()的函数调用，并传递一个函数参数（此处参数值为NULL）
      IRB.CreateCall(TestHello, {IRB.CreateBitOrPointerCast(IRB.getInt8(0), IRB.getInt8PtrTy())});
    }
    return PreservedAnalyses::all();
  }
};
```

Clang/LLVM

```
1  struct MyPass : public PassInfoMixin<MyPass>
2  {
3    PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM)
4    {
5      if (F.getName() != "printHello")
6      {
7        FunctionCallee TestHello;
8        auto &M = *F.getParent();
9        AttributeList Attr;
10       Attr = Attr.addFnAttribute(M.getContext(), Attribute::NoUnwind);
11       IRBuilder<>
12           IRB(M.getContext());
13
14       // 查找符号名为printHello，返回值类型为void，参数列表类型为[void*]的函数，注册到TestHello中
15       TestHello = F.getParent()->getOrInsertFunction("printHello", Attr, IRB.getVoidTy(), IRB.getInt8PtrTy());
16
17       // 调整插入点至函数入口
18       IRB.SetInsertPoint(&F.getEntryBlock(), F.getEntryBlock().getFirstInsertionPt());
19
20       // 插入对IRB.getVoidTy()的函数调用，并传递一个函数参数（此处参数值为NULL）
21       IRB.CreateCall(TestHello, {IRB.CreateBitOrPointerCast(IRB.getInt8(0), IRB.getInt8PtrTy())});
22     }
23     return PreservedAnalyses::all();
24   }
25 };
```

继承LLVM核心库提供的基础Pass类

Clang/LLVM

重载run()方法，实现对函数的扫描

```
struct MyPass : public PassInfoMixin<MyPass>
{
  PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM)
  {
    if (F.getName() != "printHello")
    {
      FunctionCallee TestHello;
      auto &M = *F.getParent();
      AttributeList Attr;
      Attr = Attr.addFnAttribute(M.getContext(), Attribute::NoUnwind);
      IRBuilder<>
          IRB(M.getContext());

      // 查找符号名为printHello，返回值类型为void，参数列表类型为[void*]的函数，注册到TestHello中
      TestHello = F.getParent()->getOrInsertFunction("printHello", Attr, IRB.getVoidTy(), IRB.getInt8PtrTy());

      // 调整插入点至函数入口
      IRB.SetInsertPoint(&F.getEntryBlock(), F.getEntryBlock().getFirstInsertionPt());

      // 插入对IRB.getVoidTy()的函数调用，并传递一个函数参数（此处参数值为NULL）
      IRB.CreateCall(TestHello, {IRB.CreateBitOrPointerCast(IRB.getInt8(0), IRB.getInt8PtrTy())});
    }
    return PreservedAnalyses::all();
  }
};
```

Clang/LLVM

```
1  struct MyPass : public PassInfoMixin<MyPass>
2  {
3    PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM)
4    {
5      if (F.getName() != "printHello")
6      {
7        FunctionCallee TestHello;
8        auto &M = *F.getParent();
9        AttributeList Attr;
10       Attr = Attr.addFnAttribute(M.getContext(), Attribute::NoUnwind);
11       IRBuilder<>
12           IRB(M.getContext());
13
14       // 查找符号名为printHello，返回值类型为void，参数列表类型为[void*]的函数，注册到TestHello中
15       TestHello = F.getParent()->getOrInsertFunction("printHello", Attr, IRB.getVoidTy(), IRB.getInt8PtrTy());
16
17       // 调整插入点至函数入口
18       IRB.SetInsertPoint(&F.getEntryBlock(), F.getEntryBlock().getFirstInsertionPt());
19
20       // 插入对IRB.getVoidTy()的函数调用，并传递一个函数参数（此处参数值为NULL）
21       IRB.CreateCall(TestHello, {IRB.CreateBitOrPointerCast(IRB.getInt8(0), IRB.getInt8PtrTy())});
22     }
23     return PreservedAnalyses::all();
24   }
25 };
```

扫描过程中，修改LLVM IR，在函数入口插入对printHello的调用

Clang/LLVM

```
1  PassPluginLibraryInfo getPassPluginInfo()
2  {
3    const auto callback = [](PassBuilder &PB)
4    {
5      PB.registerPipelineEarlySimplificationEPCallback(
6          [&](ModulePassManager &MPM, auto)
7          {
8            MPM.addPass(createModuleToFunctionPassAdaptor(MyPass()));
9            return true;
10         });
11   };
12
13   return {LLVM_PLUGIN_API_VERSION, "name", "0.0.1", callback};
14 };
15
16 extern "C" LLVM_ATTRIBUTE_WEAK PassPluginLibraryInfo llvmGetPassPluginInfo()
17 {
18   return getPassPluginInfo();
19 }
```

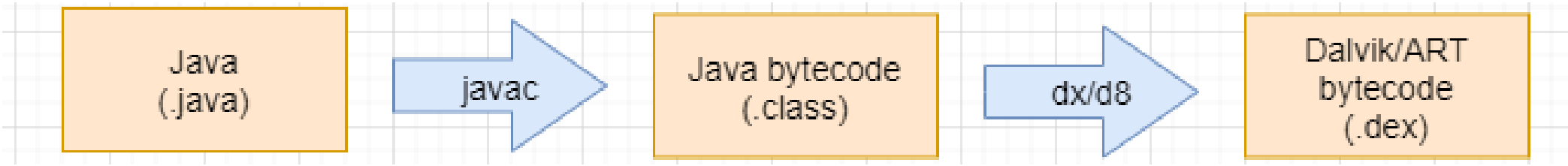
向PassManager注册新添加的Pass

常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA , Soot
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

ASM

- 在Java的编译流程中，源码（.java格式代码）先被翻译成统一格式的字节码（.class格式代码），然后再进一步被翻译为目标机器代码（例如用于Android虚拟机的.dex格式代码）

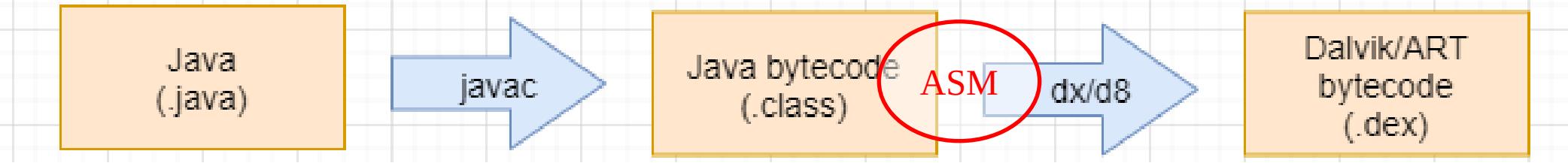


ASM

- ASM是一个通用的Java字节码操作和分析框架。
  - 直接以二进制形式用于修改现有类或动态生成类
  - 提供了一些常见的字节码转换和分析算法，可以从中构建定制的复杂转换和代码分析工具
  - 提供了与其他Java字节码框架类似的功能，但是侧重于性能。因为它的设计和实现是尽可能的**小**和尽可能**快**，所以它非常适合以动态插桩方式使用（当然也可以以静态插桩方式使用，例如在编译器中）

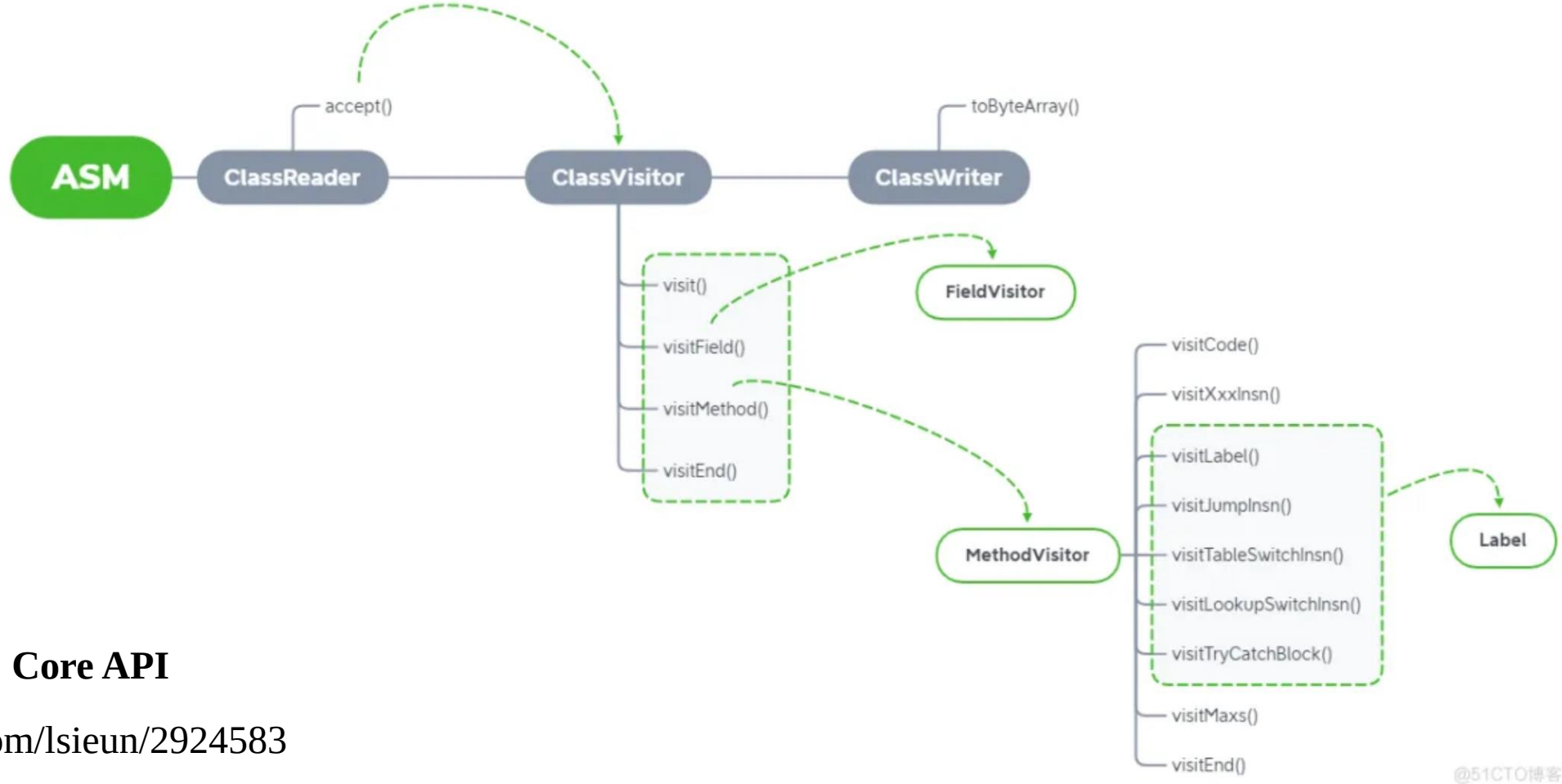
ASM

- ASM是一个通用的Java字节码操作和分析框架。
  - 直接以二进制形式用于修改现有类或动态生成类
  - 提供了一些常见的字节码转换和分析算法，可以从中构建定制的复杂转换和代码分析工具
  - 提供了与其他Java字节码框架类似的功能，但是侧重于性能。因为它的设计



ASM

- 访问者模式



Java ASM系列一： Core API

https://blog.51cto.com/lsieun/2924583

ASM

- 访问者模式
- ASM 主要依赖以下三个核心功能类：
  - ClassReader：读取并分析字节码
  - ClassVisitor：抽象类，访问类的解析，如注解、方法、成员变量的解析，对应的具体子类分别是 AnnotationVisitor、MethodVisitor（会解析方法上的注解、参数、代码等）、FieldVisitor
  - ClassWriter：ClassVisitor 的子类，可以获得解析结果

ASM

```
private byte[] modifyClass(FileInputStream fis) throws IOException {
    ClassReader classReader = new ClassReader(fis);
    ClassWriter classWriter = new ClassWriter(0);
    // 在 ClassVisitor 中修改
    ClassVisitor classVisitor = new ClassVisitor(ASM7, classWriter) {
        // 要修改方法，就在 visitMethod() 中进行
        @Override
        public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[]
            // 删除 print 方法
            if ("print".equals(name)) {
                return null;
            }

            // 将 setName 方法的访问负设为 private
            if ("setName".equals(name)) {
                access = ACC_PRIVATE;
            }

            // super 内调用的其实是 ClassVisitor 构造方法中 classWriter 的 visitMethod()
            return super.visitMethod(access, name, descriptor, signature, exceptions);
        }
```

使用ClassReader类读取并解析.class字节码文件

ASM

```
private byte[] modifyClass(FileInputStream fis) throws IOException {
    ClassReader classReader = new ClassReader(fis);
    ClassWriter classWriter = new ClassWriter(0);
    // 在 ClassVisitor 中修改
    ClassVisitor classVisitor = new ClassVisitor(ASM7, classWriter) {
        // 要修改方法，就在 visitMethod() 中进行
        @Override
        public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[]
            // 删除 print 方法
            if ("print".equals(name)) {
                return null;
            }

            // 将 setName 方法的访问负设为 private
            if ("setName".equals(name)) {
                access = ACC_PRIVATE;
            }

            // super 内调用的其实是 ClassVisitor 构造方法中 classWriter 的 visitMethod()
            return super.visitMethod(access, name, descriptor, signature, exceptions);
        }
```

使用ClassWrite类记录修改后的字节码

ASM

```
private byte[] modifyClass(FileInputStream fis) throws IOException {
    ClassReader classReader = new ClassReader(fis);
    ClassWriter classWriter = new ClassWriter(0);
    // 在 ClassVisitor 中修改
    ClassVisitor classVisitor = new ClassVisitor(ASM7, classWriter) {
        // 要修改方法，就在 visitMethod() 中进行
        @Override
        public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[]
            // 删除 print 方法
            if ("print".equals(name)) {
                return null;
            }

            // 将 setName 方法的访问负设为 private
            if ("setName".equals(name)) {
                access = ACC_PRIVATE;
            }

            // super 内调用的其实是 ClassVisitor 构造方法中 classWriter 的 visitMethod()
            return super.visitMethod(access, name, descriptor, signature, exceptions);
        }
```

修改已有的方法

ASM

新增一个String类型的变量

```
// 要增加方法或字段，最好在 visitEnd() 中进行，避免破坏之前排好的类结构
@Override
public void visitEnd() {
    // 增加一个字段 private String address
    FieldVisitor fieldVisitor = cv.visitField(ACC_PRIVATE, "address", "Ljava/lang/String;", null, null);
    fieldVisitor.visitEnd();

    // 增加一个方法 public String getAddress()
    MethodVisitor methodVisitor = cv.visitMethod(ACC_PUBLIC, "getAddress", "()Ljava/lang/String;", null,
    methodVisitor.visitCode();
    methodVisitor.visitVarInsn(ALOAD, 0);
    methodVisitor.visitFieldInsn(GETFIELD, "com/example/asm/Demo1", "address", "Ljava/lang/String;");
    methodVisitor.visitInsn(IRETURN);
    methodVisitor.visitMaxs(1, 1);
    methodVisitor.visitEnd();

    super.visitEnd();
}
```

ASM

```
// 要增加方法或字段，最好在 visitEnd() 中进行，避免破坏之前排好的类结构
@Override
public void visitEnd() {
    // 增加一个字段 private String address
    FieldVisitor fieldVisitor = cv.visitField(ACC_PRIVATE, "address", "Ljava/lang/String;", null, null);
    fieldVisitor.visitEnd();

    // 增加一个方法 public String getAddress()
    MethodVisitor methodVisitor = cv.visitMethod(ACC_PUBLIC, "getAddress", "()Ljava/lang/String;", null,
    methodVisitor.visitCode();
    methodVisitor.visitVarInsn(ALOAD, 0);
    methodVisitor.visitFieldInsn(GETFIELD, "com/example/asm/Demo1", "address", "Ljava/lang/String;");
    methodVisitor.visitInsn(IRETURN);
    methodVisitor.visitMaxs(1, 1);
    methodVisitor.visitEnd();

    super.visitEnd();
}
```

新增一个方法getAddress()

ASM

```java
// 要增加方法或字段，最好在 visitEnd() 中进行，避免破坏之前排好的类结构
@Override
public void visitEnd() {
    // 增加一个字段 private String address
    FieldVisitor fieldVisitor = cv.visitField(ACC_PRIVATE, "address", "Ljava/lang/String;", null, null);
    fieldVisitor.visitEnd();

    // 增加一个方法 public String getAddress()
    MethodVisitor methodVisitor = cv.visitMethod(ACC_PUBLIC, "getAddress", "()Ljava/lang/String;", null,
    methodVisitor.visitCode();
    methodVisitor.visitVarInsn(ALOAD, 0);
    methodVisitor.visitFieldInsn(GETFIELD, "com/example/asm/Demo1", "address", "Ljava/lang/String;");
    methodVisitor.visitInsn(IRETURN);
    methodVisitor.visitMaxs(1, 1);
    methodVisitor.visitEnd();

    super.visitEnd();
}
```

向方法getAddress()中逐条插入字节码指令

ASM

```
private byte[] modifyClass(FileInputStream fis) throws IOException {
    ClassReader classReader = new ClassReader(fis);
    ClassWriter classWriter = new ClassWriter(0);
    // 在 ClassVisitor 中修改
    ClassVisitor classVisitor = new ClassVisitor(ASM7, classWriter) {
        // 要修改方法，就在 visitMethod() 中进行
        @Override
        public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[]
            // 删除 print 方法
            if ("print".equals(name)) {
                return null;
            }

            // 将 setName 方法的访问负设为 private
            if ("setName".equals(name)) {
                access = ACC_PRIVATE;
            }

            // super 内调用的其实是 ClassVisitor 构造方法中 classWriter 的 visitMethod()
            return super.visitMethod(access, name, descriptor, signature, exceptions);
        }
    };
    classReader.accept(classVisitor, 0);
    return classWriter.toByteArray();
}
```

使用classVisitor类访问原有字节码，进行插桩，并将结果记录到classWriter类中

常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA , Soot
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

WALA

- WALA是用于静态和动态程序分析的Java库。
- 也可用于对Java字节码进行插桩。
- 最初诞生于IBM TJ Watson研究中心，2006年，IBM将其捐赠给开源社区。

WALA

- WALA的核心功能：
  - Java类型系统和类层次结构分析
  - 支持Java和JavaScript的源语言框架
  - 过程间数据流分析（RHS求解器）
  - 基于上下文的基于列表的切片器
  - 指针分析和调用图构造
  - 基于SSA的寄存器传输语言IR
  - 迭代数据流的通用框架
  - 通用分析工具和数据结构
  - 字节码检测库（Shrike）和Java的动态加载时检测库（Dila）

WALA

- 一个使用WALA打印所有类名的例子

```
public class analysisclass {

    public static void classanalysis() throws IOException, ClassHierarchyException {
        AnalysisScope scope = AnalysisScopeReader.instance.makeJavaBinaryAnalysisScope("h2.jar", (new FileProvider()).
        ClassHierarchy cha = ClassHierarchyFactory.make(scope);
        for(IClass klass : cha){
            System.out.println(klass.getName());
        }
    }
    public static void main(String[] args) throws IOException, ClassHierarchyException {
        classanalysis();
    }
}
```

WALA

- 一个使用WALA打印所有类名的例子
  - 指定分析域为“h2.jar”

```
public class analysisclass {

    public static void classanalysis() throws IOException, ClassHierarchyException {
        AnalysisScope scope = AnalysisScopeReader.instance.makeJavaBinaryAnalysisScope("h2.jar", (new FileProvider()).
        ClassHierarchy cha = ClassHierarchyFactory.make(scope);
        for(IClass klass : cha){
            System.out.println(klass.getName());
        }
    }
    public static void main(String[] args) throws IOException, ClassHierarchyException {
        classanalysis();
    }
}
```

WALA

- 一个使用WALA打印所有类名的例子
  - 读取“h2.jar”中的类层次结构

```
public class analysisclass {

    public static void classanalysis() throws IOException, ClassHierarchyException {
        AnalysisScope scope = AnalysisScopeReader.instance.makeJavaBinaryAnalysisScope("h2.jar", (new FileProvider()).
        ClassHierarchy cha = ClassHierarchyFactory.make(scope);
        for(IClass klass : cha){
            System.out.println(klass.getName());
        }
    }
    public static void main(String[] args) throws IOException, ClassHierarchyException {
        classanalysis();
    }
}
```

WALA

- 一个使用WALA打印所有类名的例子
  - 遍历打印所有类名

```
public class analysisclass {

    public static void classanalysis() throws IOException, ClassHierarchyException {
        AnalysisScope scope = AnalysisScopeReader.instance.makeJavaBinaryAnalysisScope("h2.jar", (new FileProvider()).
        ClassHierarchy cha = ClassHierarchyFactory.make(scope);
        for(IClass klass : cha){
            System.out.println(klass.getName());
        }
    }
    public static void main(String[] args) throws IOException, ClassHierarchyException {
        classanalysis();
    }
}
```

常用插桩工具

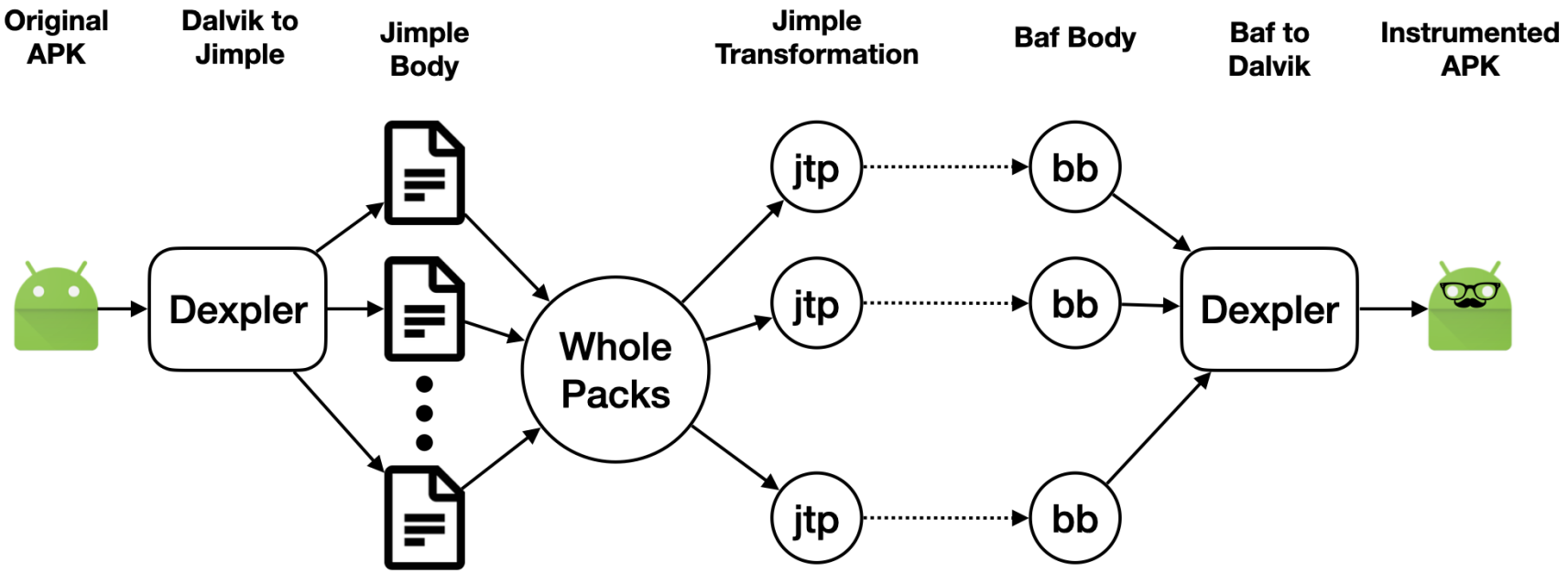
- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA, Soot
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

Soot

- Soot作为一种Java程序优化框架被开发。
- Soot现在可用于对Java和Android程序的程序分析、插桩、优化、可视化等多项任务。
- 一个“重量级”的工具。
- Soot即将停止维护，开发团队将转向新的SootUp项目（为原先Soot项目的一个扩展）

Soot

- Soot插桩Android APK的工作流程：
  - Dalvik->Jimple->插桩后的Jimple->Baf-> Dalvik



常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

Dexlib2

- Dexlib2是一个使用Java编写的Dex文件（Android机器码文件）的编辑库。
- 常用来修改Dex文件，合并Dex文件等。
- 通过修改Dex文件，可实现对Android程序的插桩。

- **安卓虚拟机 vs Java 虚拟机**
  - Java虚拟机的模型：基于栈的机器
  - 安卓虚拟机的模型：基于寄存器的机器（ARM）
  - 两者均实现解释执行到本地执行的设计，提高性能
  - 导致插桩差异很大

常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

Qemu

- QEMU是一款仿真软件，可进行应用与系统级软件仿真
  - 提供包括处理器、存储器、总线以及中断控制器、MMU(memory management unit)、FPU(float point unit)以及其他外设的全系统仿真，可以不作修改运行任何客户机系统软件。
- QEMU在运行过程中主要执行两方面任务：
  - 执行调度
  - 动态二进制翻译
- 判断当前基本翻译块(translate block, TB)是否已经在TB Cache中：
  - 不在：则进行动态二进制翻译
  - 在：根据该基本翻译块执行宿主机指令

Qemu

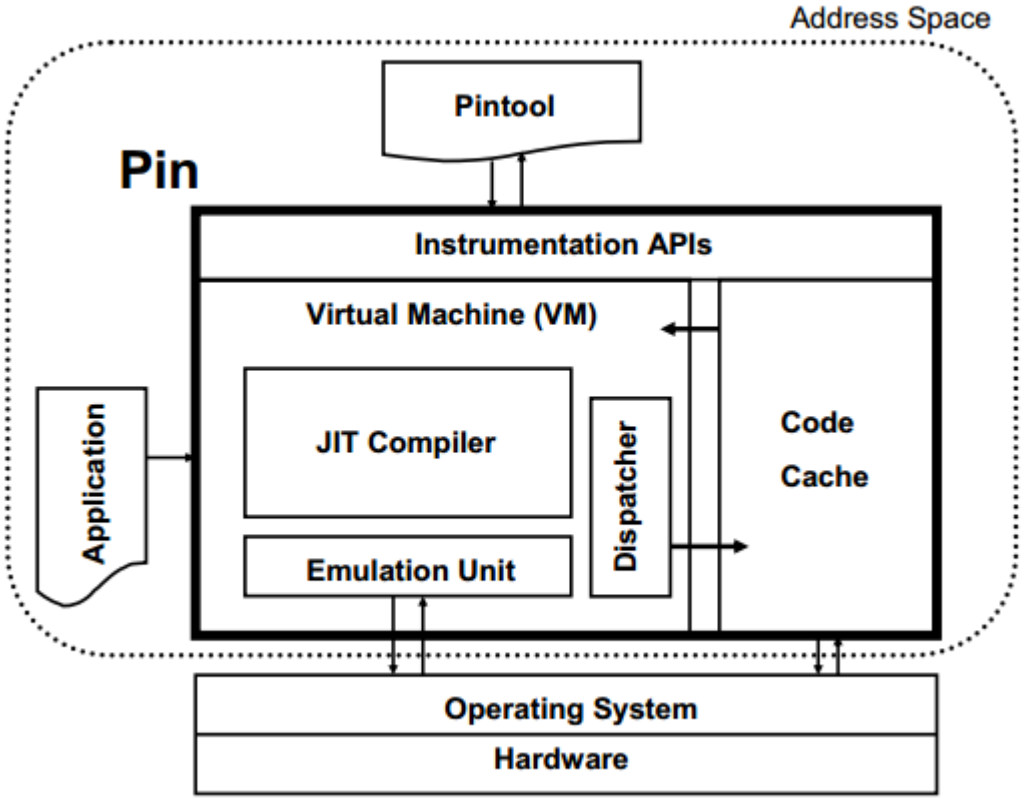
- 动态二进制翻译包含翻译前端和翻译后端2个阶段
  - 在第一阶段，读取客户机指令序列，进行反汇编，并产生中间码。
  - 在第二阶段，遍历中间码，将中间码转换成宿主机指令，保存翻译块并加入TB Cache。
- 通过修改第一阶段生成的中间码，可实现**动态二进制插桩**。

常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA, Soot
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

Pintool

- Pin是由英特尔公司开发的动态二进制插桩框架
  - 支持Windows，Linux和OSX等
- Pin 允许在可执行文件 文件的任意位置插入任意的代码 (C/C++ 编写)
- 优势：
  - 代码可以**被动态的添加到正在执行的可执行文件中**
  - 可以将 pin **附加 (attach) 到正在运行的进程中**



Pintool

```
ofstream OutFile;

// 全局变量统计二进制指令条数
static UINT64 icount = 0;

// 每次 Instruction 函数都会调用这个函数进行 icount 变量自增
VOID docount() { icount++; }

// PIN 每次执行一条 instruction 前都会执行这个函数
VOID Instruction(INS ins, VOID* v)
{
    // INS_InsertCall 函数对指令插入指定的函数
    // ins 是当前可执行文件准备执行的指令
    // IPOINT_BEFFORE 指的是 instruction 执行前时间点
    // docount 是注册的函数，在上面定义，统计指令条数
    // IARG_END 指明传入 docount 的参数的结尾，这里 docount 传入参数为空
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

// 在命令行中指明 -o xxx 可以改变输出信息存储文件路径
KNOB< string > KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "inscount.out", "specify output file name");
```

Pintool

```
ofstream OutFile;

// 全局变量统计二进制指令条数
static UINT64 icount = 0;

// 每次 Instruction 函数都会调用这个函数进行 icount 变量自增
VOID docount() { icount++; }

// PIN 每次执行一条 instruction 前都会执行这个函数
VOID Instruction(INS ins, VOID* v)
{
    // INS_InsertCall 函数对指令插入指定的函数
    // ins 是当前可执行文件准备执行的指令
    // IPOINT_BEFFORE 指的是 instruction 执行前时间点
    // docount 是注册的函数，在上面定义，统计指令条数
    // IARG_END 指明传入 docount 的参数的结尾，这里 docount 传入参数为空
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

// 在命令行中指明 -o xxx 可以改变输出信息存储文件路径
KNOB< string > KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "inscount.out", "specify output file name");
```

想要向程序中插桩docount函数

Pintool

```
ofstream OutFile;

// 全局变量统计二进制指令条数
static UINT64 icount = 0;

// 每次 Instruction 函数都会调用这个函数进行 icount 变量自增
VOID docount() { icount++; }

// PIN 每次执行一条 instruction 前都会执行这个函数
VOID Instruction(INS ins, VOID* v)
{
    // INS_InsertCall 函数对指令插入指定的函数
    // ins 是当前可执行文件准备执行的指令
    // IPOINT_BEFFORE 指的是 instruction 执行前时间点
    // docount 是注册的函数，在上面定义，统计指令条数
    // IARG_END 指明传入 docount 的参数的结尾，这里 docount 传入参数为空
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

// 在命令行中指明 -o xxx 可以改变输出信息存储文件路径
KNOB< string > KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "inscount.out", "specify output file name");
```

调用Pin提供的接口，在执行每一条指令前插入对docount()的调用

Pintool

```
int main(int argc, char* argv[])
{
    // 初始化 pin
    if (PIN_Init(argc, argv)) return Usage();

    // 打开输出文件
    OutFile.open(KnobOutputFile.Value().c_str());

    // 注册上面的 Instruction 函数，每次执行指令前调用
    INS_AddInstrumentFunction(Instruction, 0);

    // 注册终止函数
    PIN_AddFiniFunction(Fini, 0);

    // 开始执行应用，永远不会返回
    PIN_StartProgram();

    return 0;
}
```

向Pin框架注册插桩函数

Pintool

```
int main(int argc, char* argv[])
{
    // 初始化 pin
    if (PIN_Init(argc, argv)) return Usage();

    // 打开输出文件
    OutFile.open(KnobOutputFile.Value().c_str());

    // 注册上面的 Instruction 函数，每次执行指令前调用
    INS_AddInstrumentFunction(Instruction, 0);

    // 注册终止函数
    PIN_AddFiniFunction(Fini, 0);

    // 开始执行应用，永远不会返回
    PIN_StartProgram();

    return 0;
}
```

启动程序，Pin框架在运行过程中自动完成插桩

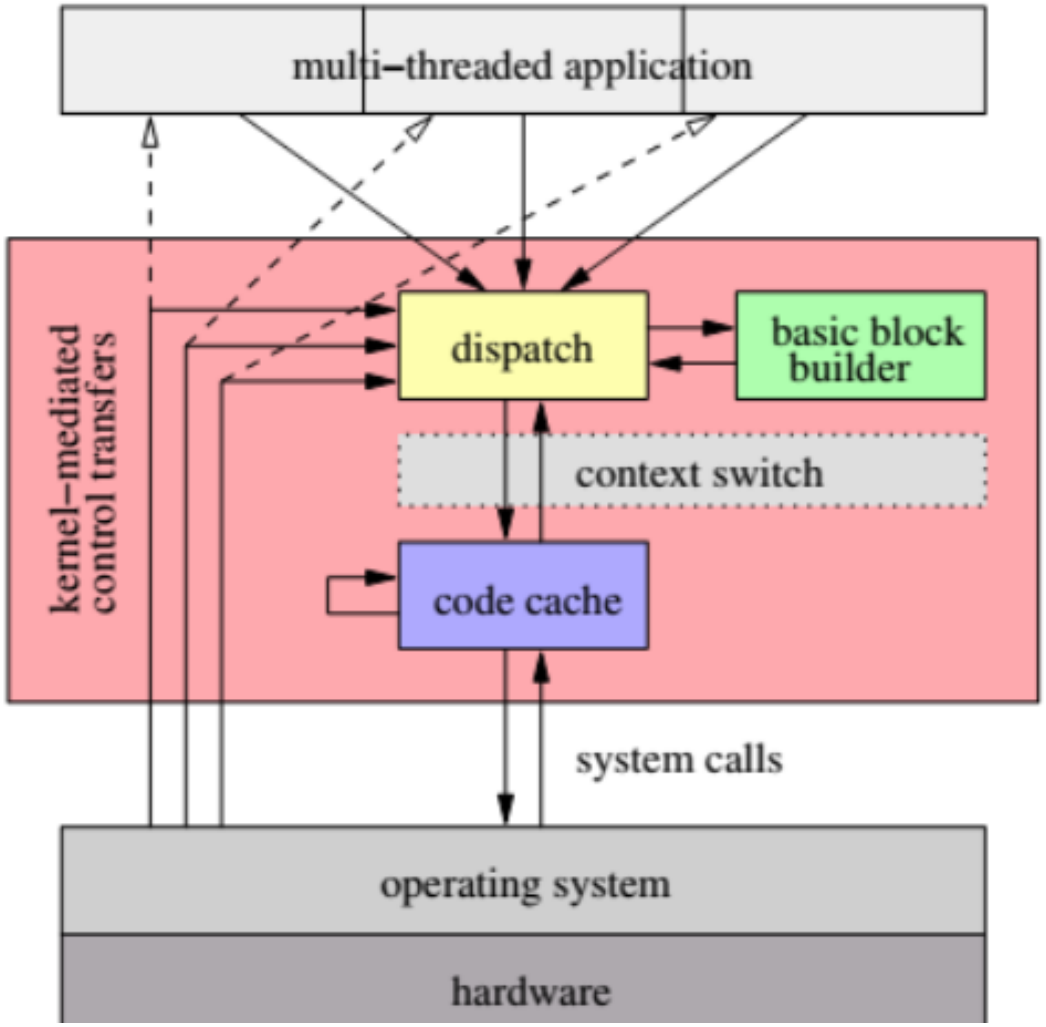
常用插桩工具

- 源代码（字节码）插桩工具：
  - C/C++： Clang/LLVM
  - Java： ASM, WALA
  - Android： Soot, Dexlib2
- 二进制插桩工具：Qemu, Pintool, DynamoRIO

DynamoRIO

- DynamoRIO是一个使用动态二进制插桩框架的平台。
  - 由惠普的Dynamo优化系统和麻省理工学院的Runtime Introspection and Optimization（RIO）研究小组合作开发
- 支持Windows和Linux
- DynamoRIO是一个运行时刻代码（ runtime code）操作系统。
- 它在操作系统和应用程序之间，构成一个中间层，允许我们在程序运行时对程序的任何部分进行代码转换。

DynamoRIO


multi−threaded application
kernel−mediated control transfers
dispatch
basic block builder
context switch
code cache
system calls
operating system
hardware

DynamoRIO

- DynamoRIO的内核由三个模块组成
  - 调度器（Dispatch）：截获被测程序的指令，完成内存事件的分发并协调各个模块之间的工作
  - 基本块构建器（Basic block builder）：从被测程序中切分出基本块并完成插桩
  - 代码缓存器（Code cache）：缓存并管理已经插桩完毕的基本块

- DynamoRIO vs Pintool

插桩工具比较

	GCC	Clang	LLVM	ASM	WALA	SOOT	Dexlib2	QEMU	Pintool	DynamoRIO
语言	C/C++		IR	Java			Dex	二进制		
插桩级别	源码		中间码	字节码		中间码		二进制		
成熟度	高									中
难易度	难								中	难
静态分析	有			无	有		无			
	GCC	Clang	LLVM	ASM	WALA	SOOT	Dexlib2	QEMU	Pintool	DynamoRIO

对机器模型的知识

1.3 困难与挑战

- 自动化程序插桩技术面临的挑战主要包括：
  - 减少插桩引入的额外开销：
    - 插桩的代码段本身会带来一定开销
    - 这些额外开销会降低程序性能，影响性能评估的准确性
    - 因此需要设计高效的插桩代码

1.3 困难与挑战

- 选择合适的插桩粒度：
  - 较细的插桩粒度（语句级）
    - 能更全面地收集程序信息
    - 会引入更大的预处理和运行开销
  - 较粗的插桩粒度（基本块级、函数级）
    - 只能收集到粗粒度的运行时信息
    - 开销较小，不会显著影响程序性能
  - 需要对二者进行权衡

Q & A

问答