**FART简介**

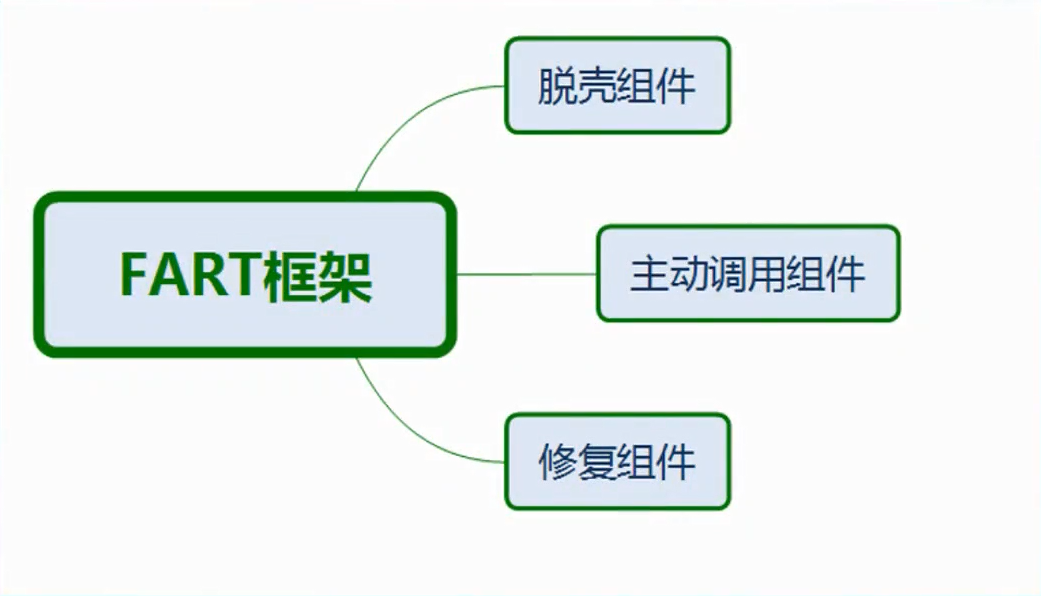
ART 环境下基于主动调用的自动化脱壳方案，可以解决函数抽取壳。

关于函数抽取壳的实现原理可以参考：[基于 art 下的类加载机制，实现函数抽取壳](https://cyrus-studio.github.io/blog/posts/%E5%9F%BA%E4%BA%8E-art-%E4%B8%8B%E7%9A%84%E7%B1%BB%E5%8A%A0%E8%BD%BD%E6%9C%BA%E5%88%B6%E5%AE%9E%E7%8E%B0%E5%87%BD%E6%95%B0%E6%8A%BD%E5%8F%96%E5%A3%B3/)

FART 的作用就是所有这些被抽空的函数的还原和修复，把加固的 dex 整体 dump 下来。

项目地址：<https://github.com/hanbinglengyue/FART>

**FART 框架**



脱壳组件：将内存中的 Dex 数据完整 dump 出来

主动调用组件：构造主动调用链，完成对函数粒度的主动调用并完成 CodeItem 的 dump

修复组件：利用脱壳组件得到的 dex 和主动调用 dump 下来的函数体，完成函数粒度的修复

**FART 中的脱壳点**

其中 FART 脱壳组件 选择 Execute 作为脱壳点，它是 Interpreter 模式执行所有 Java 方法的统一入口，能够稳定截获和提取所有解释执行的真实方法，从而达到通用脱壳的目的。

ART 下函数在运行时可能是解释执行（Interpreter 模式）或编译执行（Quick 模式）。

为何选择 Execute 作为脱壳点？

**dex2oat 编译流程**

[关于 dex2oat 以及 vdex、cdex、dex 格式转换](https://cyrus-studio.github.io/blog/posts/%E5%85%B3%E4%BA%8E-dex2oat-%E4%BB%A5%E5%8F%8A-vdexcdexdex-%E6%A0%BC%E5%BC%8F%E8%BD%AC%E6%8D%A2/)

dex2oat 编译流程入口函数：

int main(int argc, char\*\* argv) {

int result = static\_cast<int>(art::Dex2oat(argc, argv));

// Everything was done, do an explicit exit here to avoid running Runtime destructors that take

// time (bug 10645725) unless we're a debug or instrumented build or running on a memory tool.

// Note: The Dex2Oat class should not destruct the runtime in this case.

if (!art::kIsDebugBuild && !art::kIsPGOInstrumentation && !art::kRunningOnMemoryTool) {

\_exit(result);

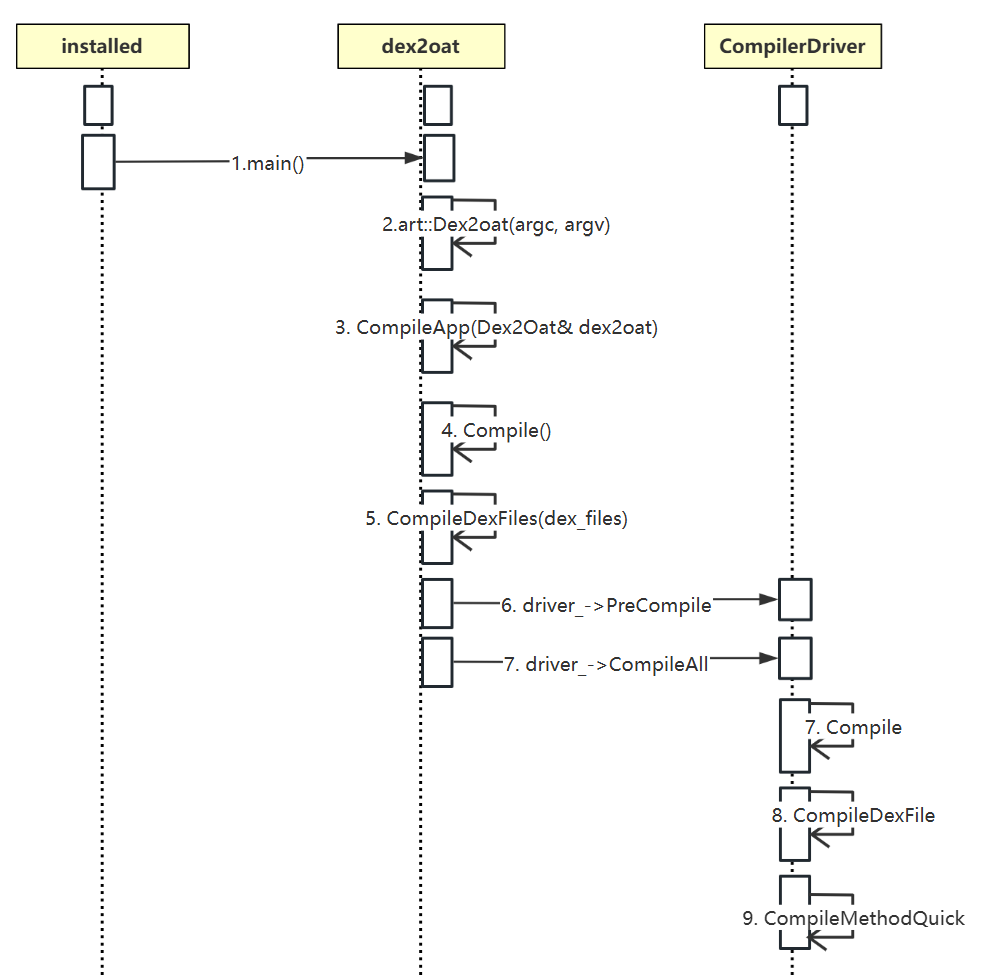
}

return result;

}

<https://cs.android.com/android/platform/superproject/+/android10-release:art/dex2oat/dex2oat.cc;l=3027>

apk 安装时进行的 dex2oat 编译流程，dex2oat 的编译驱动会对函数逐个编译



<https://www.processon.com/i/6825c4e0c168ca282c1b9fb4?full_name=PO_iNeoB2>

模板函数 CompileDexFile：编译 dex 文件中的所有类和方法，最后会调用 compile\_fn 进行函数粒度编译

// 模板函数：编译 dex 文件中的所有类和方法

template <typename CompileFn>

static void CompileDexFile(CompilerDriver\* driver,

jobject class\_loader,

const DexFile& dex\_file,

const std::vector<const DexFile\*>& dex\_files,

ThreadPool\* thread\_pool,

size\_t thread\_count,

TimingLogger\* timings,

const char\* timing\_name,

CompileFn compile\_fn) {

// 用于性能分析记录这段编译过程的时间

TimingLogger::ScopedTiming t(timing\_name, timings);

// 创建一个用于并行编译的上下文管理器

ParallelCompilationManager context(Runtime::Current()->GetClassLinker(),

class\_loader,

driver,

&dex\_file,

dex\_files,

thread\_pool);

// 编译单个类的回调函数

auto compile = [&context, &compile\_fn](size\_t class\_def\_index) {

const DexFile& dex\_file = \*context.GetDexFile();

ClassLinker\* class\_linker = context.GetClassLinker();

jobject jclass\_loader = context.GetClassLoader();

ClassReference ref(&dex\_file, class\_def\_index);

const dex::ClassDef& class\_def = dex\_file.GetClassDef(class\_def\_index);

ClassAccessor accessor(dex\_file, class\_def\_index);

CompilerDriver\* const driver = context.GetCompiler();

// 跳过验证失败的类（这些类在运行时也会失败）

if (driver->GetCompilerOptions().GetVerificationResults()->IsClassRejected(ref)) {

return;

}

// 进入托管代码环境，访问 Java 对象

ScopedObjectAccess soa(Thread::Current());

StackHandleScope<3> hs(soa.Self());

// 解码 class\_loader 对象

Handle<mirror::ClassLoader> class\_loader(

hs.NewHandle(soa.Decode<mirror::ClassLoader>(jclass\_loader)));

// 查找类对象

Handle<mirror::Class> klass(

hs.NewHandle(class\_linker->FindClass(soa.Self(), accessor.GetDescriptor(), class\_loader)));

Handle<mirror::DexCache> dex\_cache;

if (klass == nullptr) {

// 类加载失败，清除异常并使用 dex cache

soa.Self()->AssertPendingException();

soa.Self()->ClearException();

dex\_cache = hs.NewHandle(class\_linker->FindDexCache(soa.Self(), dex\_file));

} else if (SkipClass(jclass\_loader, dex\_file, klass.Get())) {

// 判断是否跳过该类（如外部类、系统类等）

return;

} else if (&klass->GetDexFile() != &dex\_file) {

// 重复类（已从另一个 dex 文件加载），跳过

return;

} else {

dex\_cache = hs.NewHandle(klass->GetDexCache());

}

// 没有方法的类无需编译

if (accessor.NumDirectMethods() + accessor.NumVirtualMethods() == 0) {

return;

}

// 进入 native 状态，避免阻塞 GC

ScopedThreadSuspension sts(soa.Self(), kNative);

// 判断是否启用 dex-to-dex 编译（可能是省略优化过程）

optimizer::DexToDexCompiler::CompilationLevel dex\_to\_dex\_compilation\_level =

GetDexToDexCompilationLevel(soa.Self(), \*driver, jclass\_loader, dex\_file, class\_def);

// 编译类中所有 direct 和 virtual 方法

int64\_t previous\_method\_idx = -1;

for (const ClassAccessor::Method& method : accessor.GetMethods()) {

const uint32\_t method\_idx = method.GetIndex();

if (method\_idx == previous\_method\_idx) {

// 处理非法 smali 文件：可能多个 method 共用同一个 method\_idx（重复定义）

continue;

}

previous\_method\_idx = method\_idx;

// 调用外部传入的 compile\_fn 进行实际方法编译

compile\_fn(soa.Self(),

driver,

method.GetCodeItem(),

method.GetAccessFlags(),

method.GetInvokeType(class\_def.access\_flags\_),

class\_def\_index,

method\_idx,

class\_loader,

dex\_file,

dex\_to\_dex\_compilation\_level,

dex\_cache);

}

};

// 并发执行 compile 回调，对 dex 文件中的所有 class 进行编译

context.ForAllLambda(0, dex\_file.NumClassDefs(), compile, thread\_count);

}

<https://cs.android.com/android/platform/superproject/+/android10-release:art/dex2oat/driver/compiler_driver.cc;l=2559>

compile\_fn 是一个回调函数，用于处理每个方法的编译过程（通常是 JIT/AOT 编译器提供的函数指针或 Lambda）。

<https://cs.android.com/android/platform/superproject/+/android10-release:art/dex2oat/driver/compiler_driver.cc;l=2632>

在 dex2oat 编译流程中，compile\_fn 是 CompileMethodQuick 函数，它是 Quick 编译器（AOT） 编译每个方法的核心入口。

// 快速编译指定方法的包装函数，用于传入 CompileDexFile 进行批量编译。

static void CompileMethodQuick(

Thread\* self,

CompilerDriver\* driver,

const dex::CodeItem\* code\_item,

uint32\_t access\_flags,

InvokeType invoke\_type,

uint16\_t class\_def\_idx,

uint32\_t method\_idx,

Handle<mirror::ClassLoader> class\_loader,

const DexFile& dex\_file,

optimizer::DexToDexCompiler::CompilationLevel dex\_to\_dex\_compilation\_level,

Handle<mirror::DexCache> dex\_cache) {

// 实际执行编译的 lambda 函数，传给 CompileMethodHarness 执行

auto quick\_fn = [](

Thread\* self,

CompilerDriver\* driver,

const dex::CodeItem\* code\_item,

uint32\_t access\_flags,

InvokeType invoke\_type,

uint16\_t class\_def\_idx,

uint32\_t method\_idx,

Handle<mirror::ClassLoader> class\_loader,

const DexFile& dex\_file,

optimizer::DexToDexCompiler::CompilationLevel dex\_to\_dex\_compilation\_level,

Handle<mirror::DexCache> dex\_cache) {

DCHECK(driver != nullptr);

CompiledMethod\* compiled\_method = nullptr;

MethodReference method\_ref(&dex\_file, method\_idx); // 方法引用（用于 profile 与验证等）

// 如果是 native 方法

if ((access\_flags & kAccNative) != 0) {

// 如果禁用了 JNI 编译但目标平台支持通用 stub，则跳过生成 stub，使用默认实现

if (!driver->GetCompilerOptions().IsJniCompilationEnabled() &&

InstructionSetHasGenericJniStub(driver->GetCompilerOptions().GetInstructionSet())) {

// 什么也不做，走 generic jni stub

} else {

// 读取方法上的 @FastNative 或 @CriticalNative 注解（优化调用约定）

access\_flags |= annotations::GetNativeMethodAnnotationAccessFlags(

dex\_file, dex\_file.GetClassDef(class\_def\_idx), method\_idx);

// 使用编译器生成 JNI stub（桥接 Java 和 native 函数的中间代码）

compiled\_method = driver->GetCompiler()->JniCompile(

access\_flags, method\_idx, dex\_file, dex\_cache);

CHECK(compiled\_method != nullptr); // 确保 JNI 编译成功

}

// 如果是 abstract 方法，无需编译（没有实现体）

} else if ((access\_flags & kAccAbstract) != 0) {

// Do nothing

// 普通 Java 方法

} else {

const VerificationResults\* results = driver->GetCompilerOptions().GetVerificationResults();

DCHECK(results != nullptr);

const VerifiedMethod\* verified\_method = results->GetVerifiedMethod(method\_ref);

// 判断该方法是否应该被编译

bool compile =

results->IsCandidateForCompilation(method\_ref, access\_flags) &&

verified\_method != nullptr &&

!verified\_method->HasRuntimeThrow() && // 验证阶段没有失败

(verified\_method->GetEncounteredVerificationFailures() &

(verifier::VERIFY\_ERROR\_FORCE\_INTERPRETER | verifier::VERIFY\_ERROR\_LOCKING)) == 0 &&

driver->ShouldCompileBasedOnProfile(method\_ref); // 在 profile 中标记为热点

if (compile) {

// 编译方法（返回 CompiledMethod 对象）

compiled\_method = driver->GetCompiler()->Compile(code\_item,

access\_flags,

invoke\_type,

class\_def\_idx,

method\_idx,

class\_loader,

dex\_file,

dex\_cache);

// 根据设置校验 profile 方法是否一定要被编译成功

ProfileMethodsCheck check\_type =

driver->GetCompilerOptions().CheckProfiledMethodsCompiled();

if (UNLIKELY(check\_type != ProfileMethodsCheck::kNone)) {

bool violation = driver->ShouldCompileBasedOnProfile(method\_ref) &&

(compiled\_method == nullptr);

if (violation) {

std::ostringstream oss;

oss << "Failed to compile "

<< method\_ref.dex\_file->PrettyMethod(method\_ref.index)

<< " as expected by profile";

switch (check\_type) {

case ProfileMethodsCheck::kNone:

break;

case ProfileMethodsCheck::kLog:

LOG(ERROR) << oss.str(); // 仅记录错误日志

break;

case ProfileMethodsCheck::kAbort:

LOG(FATAL\_WITHOUT\_ABORT) << oss.str(); // 直接终止程序

\_exit(1);

}

}

}

}

// 如果 Quick 编译失败，且允许 Dex-to-Dex 编译，则走 D2D 优化路径

if (compiled\_method == nullptr &&

dex\_to\_dex\_compilation\_level !=

optimizer::DexToDexCompiler::CompilationLevel::kDontDexToDexCompile) {

DCHECK(!Runtime::Current()->UseJitCompilation()); // AOT 模式

driver->GetDexToDexCompiler().MarkForCompilation(self, method\_ref); // 标记用于 D2D

}

}

return compiled\_method; // 返回最终生成的 CompiledMethod 对象或 nullptr

};

// 使用通用包装器调用 lambda，用于处理计时、线程控制、异常处理等

CompileMethodHarness(self,

driver,

code\_item,

access\_flags,

invoke\_type,

class\_def\_idx,

method\_idx,

class\_loader,

dex\_file,

dex\_to\_dex\_compilation\_level,

dex\_cache,

quick\_fn);

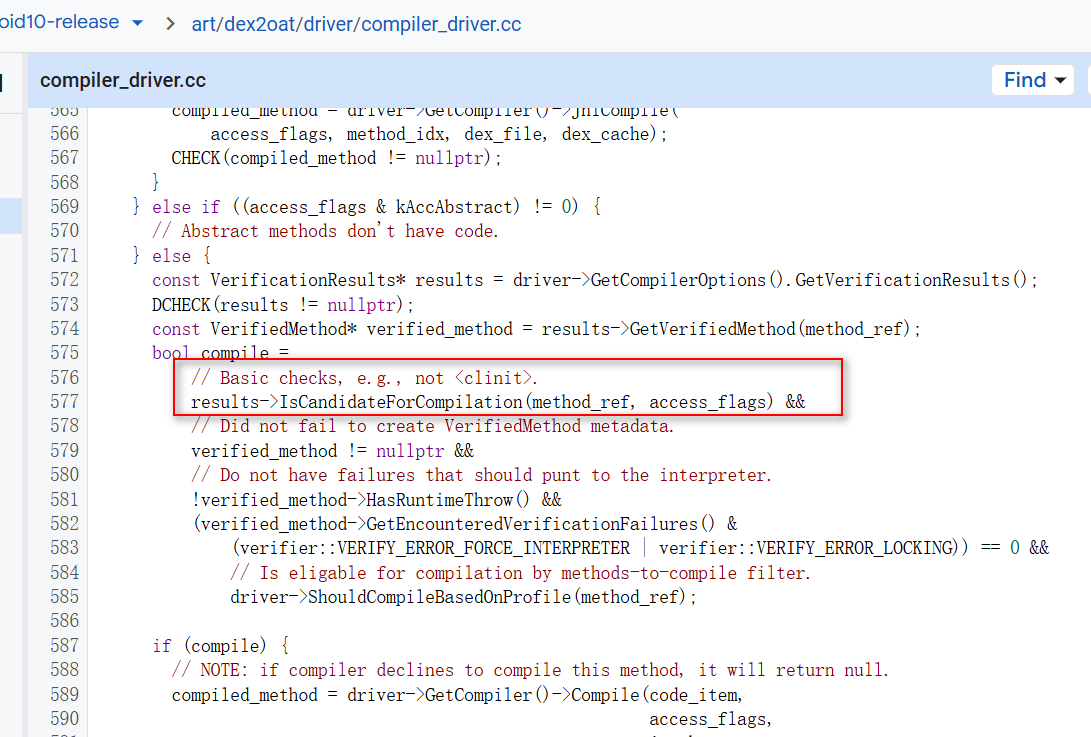
}

<https://cs.android.com/android/platform/superproject/+/android10-release:art/dex2oat/driver/compiler_driver.cc;l=2671>

<https://cs.android.com/android/platform/superproject/+/android10-release:art/dex2oat/driver/compiler_driver.cc;l=527>

并不是所有函数都会被编译！

比如类的初始化函数 <clinit>。因此，对于当一个类被初始化时，该类的初始化函数始终运行在 Interpreter 模式



<https://cs.android.com/android/platform/superproject/+/android10-release:art/dex2oat/driver/compiler_driver.cc;l=577>

**ART 下函数执行模式**

ART 下函数执行模式：

Interpreter 模式：使用 ART 自带的解释器逐条解释执行 DEX 字节码

Quick 模式：直接运行 DEX 字节码 通过 dex2oat 编译后的 平台相关的机器码（如 ARM64 指令）

调用 ArtMethod::Invoke 执行一个 Java 方法，执行流程大概如下：

ArtMethod::Invoke(...)

├─ 判断是否需要解释执行（Interpreter 模式）

│ └─ 是：调用 EnterInterpreterFromInvoke(...)

│ └─ 构造 shadow frame（解释器需要的栈帧）

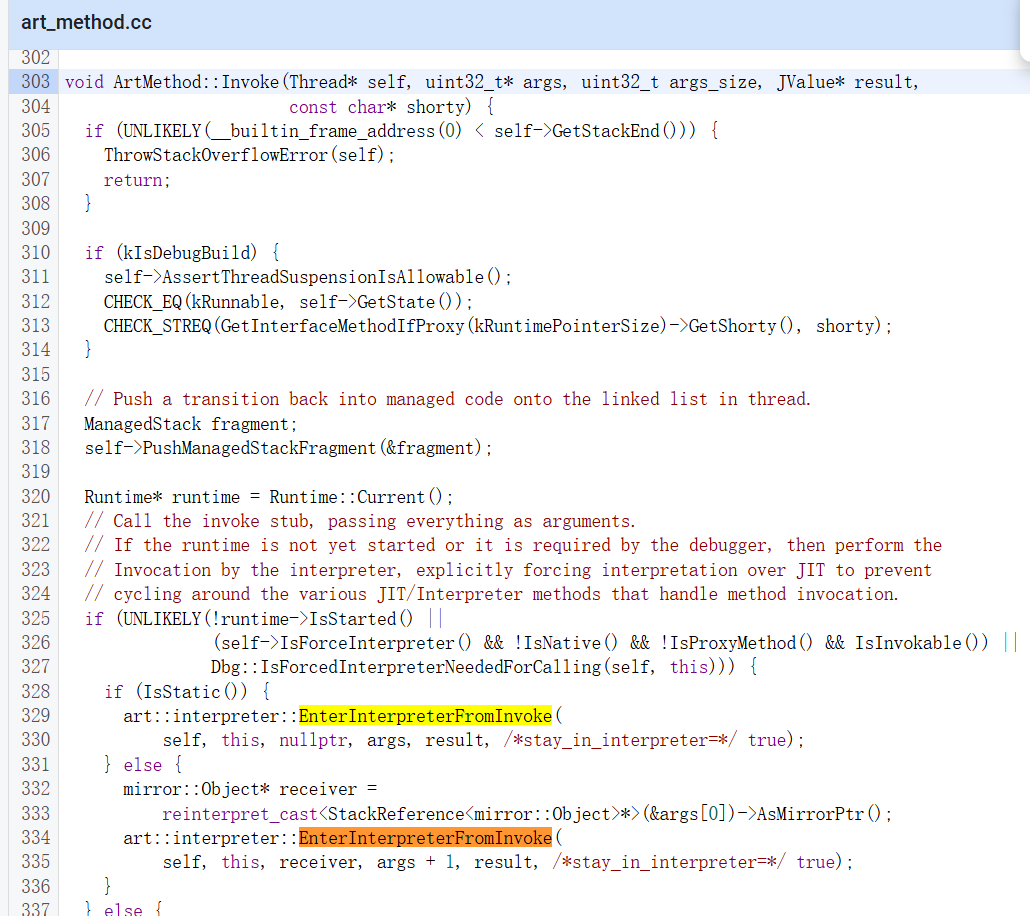
│ └─ 如果是非 native 方法：调用 Execute(...) 开始解释执行

│ └─ 如果是 native 方法：走 InterpreterJni(...)

└─ 否：调用快速入口点 art\_quick\_invoke\_stub 或 art\_quick\_invoke\_static\_stub

**Interpreter 模式流程**

从 ArtMethod 类中的 Invoke 方法开始



<https://cs.android.com/android/platform/superproject/+/android10-release:art/runtime/art_method.cc;l=303>

art::interpreter::EnterInterpreterFromInvoke 是解释器模式的入口方法。

void EnterInterpreterFromInvoke(Thread\* self,

ArtMethod\* method,

ObjPtr<mirror::Object> receiver,

uint32\_t\* args,

JValue\* result,

bool stay\_in\_interpreter) {

DCHECK\_EQ(self, Thread::Current());

// 🔒 检查是否栈溢出（防止非法栈访问）

bool implicit\_check = !Runtime::Current()->ExplicitStackOverflowChecks();

if (UNLIKELY(\_\_builtin\_frame\_address(0) < self->GetStackEndForInterpreter(implicit\_check))) {

ThrowStackOverflowError(self);

return;

}

// ⚠️ 检查是否调用了已经过时（Obsolete）的方法

if (UNLIKELY(method->IsObsolete())) {

ThrowInternalError("Attempting to invoke obsolete version of '%s'.",

method->PrettyMethod().c\_str());

return;

}

// 🧵 禁止线程挂起，防止中间被 GC 打断

const char\* old\_cause = self->StartAssertNoThreadSuspension("EnterInterpreterFromInvoke");

// 🎯 获取该方法的 CodeItem 数据，用于获取寄存器数和入参数

CodeItemDataAccessor accessor(method->DexInstructionData());

uint16\_t num\_regs;

uint16\_t num\_ins;

if (accessor.HasCodeItem()) {

// Java 方法：从 CodeItem 中获取寄存器数和参数数

num\_regs = accessor.RegistersSize();

num\_ins = accessor.InsSize();

} else if (!method->IsInvokable()) {

// 方法无法被调用（比如 abstract），抛出错误

self->EndAssertNoThreadSuspension(old\_cause);

method->ThrowInvocationTimeError();

return;

} else {

// Native 方法：计算参数数量（静态方法不需要接收者）

DCHECK(method->IsNative());

num\_regs = num\_ins = ArtMethod::NumArgRegisters(method->GetShorty());

if (!method->IsStatic()) {

num\_regs++;

num\_ins++;

}

}

// 🧱 创建 ShadowFrame（栈帧结构体），用于解释器执行

ShadowFrame\* last\_shadow\_frame = self->GetManagedStack()->GetTopShadowFrame();

ShadowFrameAllocaUniquePtr shadow\_frame\_unique\_ptr =

CREATE\_SHADOW\_FRAME(num\_regs, last\_shadow\_frame, method, /\* dex pc \*/ 0);

ShadowFrame\* shadow\_frame = shadow\_frame\_unique\_ptr.get();

self->PushShadowFrame(shadow\_frame); // 压入当前线程的 shadow frame 栈

// 📦 将参数填充到 shadow frame 中（包括 this/receiver）

size\_t cur\_reg = num\_regs - num\_ins;

if (!method->IsStatic()) {

// 实例方法的第一个参数是 receiver

CHECK(receiver != nullptr);

shadow\_frame->SetVRegReference(cur\_reg, receiver);

++cur\_reg;

}

// 根据方法签名（shorty）将参数依次放入 shadow frame 的寄存器中

uint32\_t shorty\_len = 0;

const char\* shorty = method->GetShorty(&shorty\_len);

for (size\_t shorty\_pos = 0, arg\_pos = 0; cur\_reg < num\_regs; ++shorty\_pos, ++arg\_pos, cur\_reg++) {

DCHECK\_LT(shorty\_pos + 1, shorty\_len);

switch (shorty[shorty\_pos + 1]) {

case 'L': { // 对象引用

ObjPtr<mirror::Object> o =

reinterpret\_cast<StackReference<mirror::Object>\*>(&args[arg\_pos])->AsMirrorPtr();

shadow\_frame->SetVRegReference(cur\_reg, o);

break;

}

case 'J': case 'D': { // long 或 double，占两个寄存器

uint64\_t wide\_value = (static\_cast<uint64\_t>(args[arg\_pos + 1]) << 32) | args[arg\_pos];

shadow\_frame->SetVRegLong(cur\_reg, wide\_value);

cur\_reg++; // 多占一个寄存器

arg\_pos++;

break;

}

default: // 其他基本类型（int、float等）

shadow\_frame->SetVReg(cur\_reg, args[arg\_pos]);

break;

}

}

self->EndAssertNoThreadSuspension(old\_cause); // 恢复线程挂起状态

// 🧪 如果是静态方法，确保类已初始化（可能触发类初始化）

if (method->IsStatic() && UNLIKELY(!method->GetDeclaringClass()->IsInitialized())) {

ClassLinker\* class\_linker = Runtime::Current()->GetClassLinker();

StackHandleScope<1> hs(self);

Handle<mirror::Class> h\_class(hs.NewHandle(method->GetDeclaringClass()));

if (UNLIKELY(!class\_linker->EnsureInitialized(self, h\_class, true, true))) {

// 初始化失败，抛出异常，退出

CHECK(self->IsExceptionPending());

self->PopShadowFrame();

return;

}

}

// 🧠【解释器执行路径】

if (LIKELY(!method->IsNative())) {

// 🎯 执行解释器主函数

// 🔥 这一步真正进入 Execute（根据配置进入 mterp 或 switch 实现）

JValue r = Execute(self, accessor, \*shadow\_frame, JValue(), stay\_in\_interpreter);

if (result != nullptr) {

\*result = r; // 保存执行结果

}

} else {

// 💡 Native 方法：JNI 函数不会走普通解释器路径

// 但在 image 写入或测试时会通过 InterpreterJni 执行

args = shadow\_frame->GetVRegArgs(method->IsStatic() ? 0 : 1);

if (!Runtime::Current()->IsStarted()) {

// image 构建期模拟调用 native 方法

UnstartedRuntime::Jni(self, method, receiver.Ptr(), args, result);

} else {

// 正常 JNI 方法调用

InterpreterJni(self, method, shorty, receiver, args, result);

}

}

// 🧹 弹出 shadow frame，恢复执行栈

self->PopShadowFrame();

}

<https://cs.android.com/android/platform/superproject/+/android10-release:art/runtime/interpreter/interpreter.cc;l=399>

Interpreter 模式下，Java 函数最终都会走到 Execute。

**总体调用流程**

ArtMethod::Invoke

↓

interpreter::EnterInterpreterFromInvoke

↓

interpreter::Execute ← 解释器执行核心

↓

[use\_mterp ?] → ExecuteMterpImpl

ExecuteSwitchImpl

可以看到，对于任何一个运行在 interpreter 模式的 java 函数来说，最终都会进入到 ART 下的解释器中进行解释执行。

**Execute**

Execute 函数负责在 ART 虚拟机中根据当前执行环境选择合适的解释器（如 Mterp 或 Switch）执行指定的 Java 方法字节码。

static inline JValue Execute(

Thread\* self,

const CodeItemDataAccessor& accessor,

ShadowFrame& shadow\_frame,

JValue result\_register,

bool stay\_in\_interpreter = false,

bool from\_deoptimize = false) REQUIRES\_SHARED(Locks::mutator\_lock\_) {

// 方法不能是 abstract 或 native，因为解释器无法执行它们

DCHECK(!shadow\_frame.GetMethod()->IsAbstract());

DCHECK(!shadow\_frame.GetMethod()->IsNative());

// 检查当前线程是否使用了正确类型的解释器（比如 mterp）

if (kIsDebugBuild && self->UseMterp() != CanUseMterp()) {

MutexLock tll\_mu(self, \*Locks::thread\_list\_lock\_);

DCHECK\_EQ(self->UseMterp(), CanUseMterp());

}

// 如果不是从 deoptimization 进入（正常调用路径）

if (LIKELY(!from\_deoptimize)) {

if (kIsDebugBuild) {

// 新进入方法，DexPC 应为 0，且不能有异常待处理

CHECK\_EQ(shadow\_frame.GetDexPC(), 0u);

self->AssertNoPendingException();

}

// 获取当前运行时的 instrumentation 组件（用于调试/监控方法调用）

instrumentation::Instrumentation\* instrumentation = Runtime::Current()->GetInstrumentation();

ArtMethod\* method = shadow\_frame.GetMethod();

// 如果注册了方法进入监听器，则调用监听逻辑

if (UNLIKELY(instrumentation->HasMethodEntryListeners())) {

instrumentation->MethodEnterEvent(self,

shadow\_frame.GetThisObject(accessor.InsSize()),

method,

0);

// 如果 instrumentation 指定需要强制退出该帧

if (UNLIKELY(shadow\_frame.GetForcePopFrame())) {

DCHECK(Runtime::Current()->AreNonStandardExitsEnabled());

DCHECK(PrevFrameWillRetry(self, shadow\_frame));

return JValue(); // 不执行，直接返回

}

// 如果 instrumentation 导致了异常，也直接返回

if (UNLIKELY(self->IsExceptionPending())) {

instrumentation->MethodUnwindEvent(self,

shadow\_frame.GetThisObject(accessor.InsSize()),

method,

0);

return JValue();

}

}

// 如果允许切换到 JIT 编译执行（非强制 stay\_in\_interpreter 且非强制解释器）

if (!stay\_in\_interpreter && !self->IsForceInterpreter()) {

jit::Jit\* jit = Runtime::Current()->GetJit();

if (jit != nullptr) {

// 通知 JIT 方法已进入

jit->MethodEntered(self, method);

// 如果该方法已经被编译过了，则可以直接调用机器码

if (jit->CanInvokeCompiledCode(method)) {

JValue result;

// 先弹出 ShadowFrame

self->PopShadowFrame();

// 计算参数偏移量（输入参数寄存器在高位）

uint16\_t arg\_offset = accessor.RegistersSize() - accessor.InsSize();

// 通过桥接方法跳转到已编译代码执行

ArtInterpreterToCompiledCodeBridge(self, nullptr, &shadow\_frame, arg\_offset, &result);

// 执行完成后重新压回 ShadowFrame

self->PushShadowFrame(&shadow\_frame);

return result;

}

}

}

}

// 获取当前方法

ArtMethod\* method = shadow\_frame.GetMethod();

// 验证方法静态状态

DCheckStaticState(self, method);

// 如果启用了访问检查，则必须关闭锁计数器检查

DCHECK(!method->SkipAccessChecks() || !method->MustCountLocks());

bool transaction\_active = Runtime::Current()->IsActiveTransaction();

// 判断是否跳过访问权限检查

if (LIKELY(method->SkipAccessChecks())) {

// === 进入 "无需访问检查" 模式 ===

if (kInterpreterImplKind == kMterpImplKind) {

// 解释器是 mterp

if (transaction\_active) {

// mterp 不支持事务，回退到 switch 模式

return ExecuteSwitchImpl<false, true>(self, accessor, shadow\_frame, result\_register, false);

} else if (UNLIKELY(!Runtime::Current()->IsStarted())) {

// Runtime 尚未启动，mterp 不可用，回退 switch 模式

return ExecuteSwitchImpl<false, false>(self, accessor, shadow\_frame, result\_register, false);

} else {

// 可使用 mterp

while (true) {

// mterp 不支持调试/断点等，所以如果当前线程不允许用 mterp，就退回 switch

if (!self->UseMterp()) {

return ExecuteSwitchImpl<false, false>(self, accessor, shadow\_frame, result\_register, false);

}

// 调用 mterp 执行指令

bool returned = ExecuteMterpImpl(self,

accessor.Insns(), // 获取指令序列

&shadow\_frame,

&result\_register);

if (returned) {

// mterp 执行成功（正常返回）

return result\_register;

} else {

// mterp 无法处理该指令，改用 switch 解释器单步执行

result\_register = ExecuteSwitchImpl<false, false>(self, accessor, shadow\_frame, result\_register, true);

if (shadow\_frame.GetDexPC() == dex::kDexNoIndex) {

// 已执行 return 或发生未捕获异常，直接返回

return result\_register;

}

}

}

}

} else {

// 当前解释器类型是 switch

DCHECK\_EQ(kInterpreterImplKind, kSwitchImplKind);

if (transaction\_active) {

return ExecuteSwitchImpl<false, true>(self, accessor, shadow\_frame, result\_register, false);

} else {

return ExecuteSwitchImpl<false, false>(self, accessor, shadow\_frame, result\_register, false);

}

}

} else {

// === 进入 "需要访问检查" 模式 ===

// 启动路径不应该运行到这里，除非是软验证失败或者在 AOT 编译中

DCHECK(method->GetDeclaringClass()->GetClassLoader() != nullptr

|| Runtime::Current()->IsVerificationSoftFail()

|| Runtime::Current()->IsAotCompiler())

<< method->PrettyMethod();

if (kInterpreterImplKind == kMterpImplKind) {

// mterp 不支持访问检查，强制使用 switch 模式

if (transaction\_active) {

return ExecuteSwitchImpl<true, true>(self, accessor, shadow\_frame, result\_register, false);

} else {

return ExecuteSwitchImpl<true, false>(self, accessor, shadow\_frame, result\_register, false);

}

} else {

// switch 模式解释器分支

DCHECK\_EQ(kInterpreterImplKind, kSwitchImplKind);

if (transaction\_active) {

return ExecuteSwitchImpl<true, true>(self, accessor, shadow\_frame, result\_register, false);

} else {

return ExecuteSwitchImpl<true, false>(self, accessor, shadow\_frame, result\_register, false);

}

}

}

}

<https://cs.android.com/android/platform/superproject/+/android10-release:art/runtime/interpreter/interpreter.cc;l=247>

**ExecuteMterpImpl 和 ExecuteSwitchImpl**

ExecuteMterpImpl 和 ExecuteSwitchImpl 是 ART 虚拟机中两种解释器的执行实现方式，它们的主要区别在于 执行效率、可调试性 和 支持的功能。

ExecuteMterpImpl：使用 高性能的汇编模板代码（如 x86、arm64 手写汇编）执行字节码，效率非常高。

ExecuteSwitchImpl：基于 C++ 的 switch-case 控制流，每条字节码指令有一个 case 分支。

它们是如何被选择执行的？ 在 Execute 函数中：

if (kInterpreterImplKind == kMterpImplKind) {

if (transaction\_active) {

return ExecuteSwitchImpl<...>(); // Mterp 不支持事务，退回 Switch

} else if (!Runtime::Current()->IsStarted()) {

return ExecuteSwitchImpl<...>(); // VM 没启动也不能用 Mterp

} else {

while (true) {

if (!self->UseMterp()) {

return ExecuteSwitchImpl<...>(); // 当前线程禁用了 Mterp

}

bool returned = ExecuteMterpImpl(...);

if (returned) {

return result\_register; // Mterp 成功执行完成

} else {

// Mterp 遇到不支持的指令或状态，单步回退到 Switch

result\_register = ExecuteSwitchImpl(...);

if (shadow\_frame.GetDexPC() == dex::kDexNoIndex) {

return result\_register; // 已返回或异常抛出

}

}

}

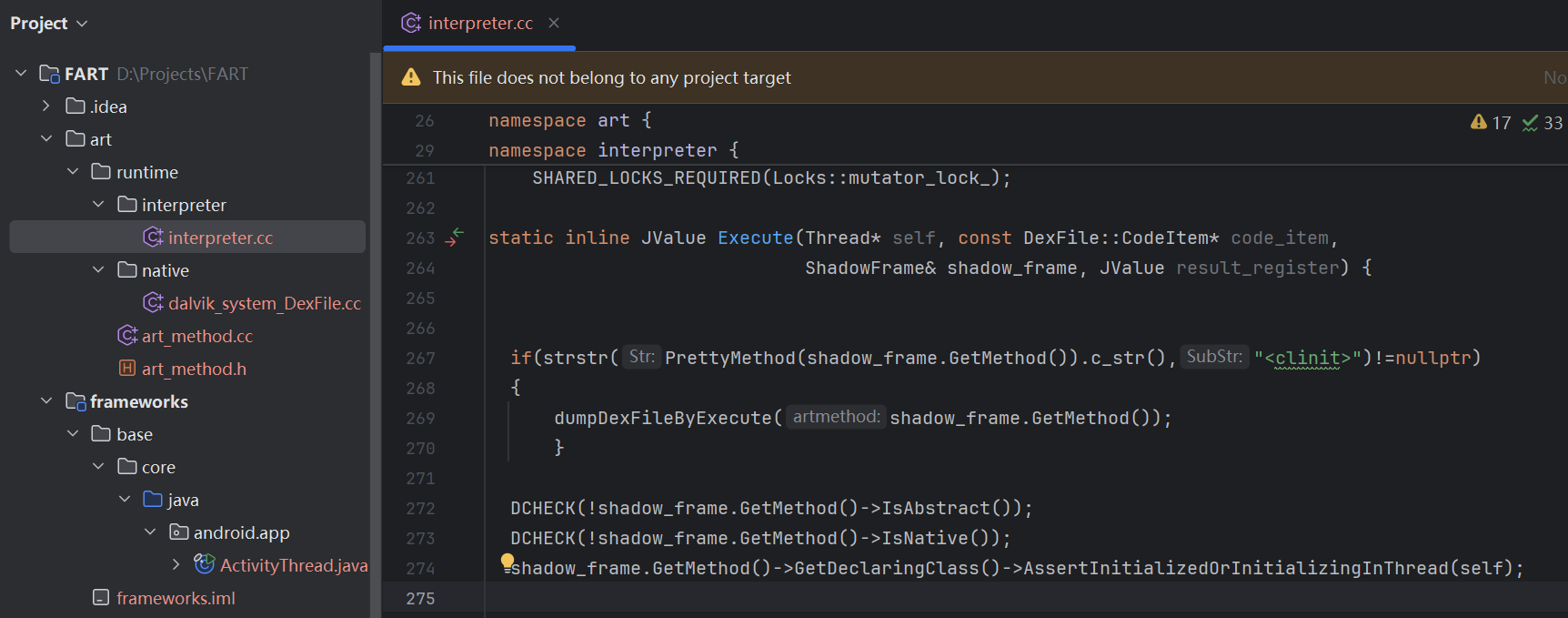
}

}

**脱壳实现**

对于函数抽取壳的 dex 来说，因为需要禁用了 dex2oat ，所以都会以解释模式运行，进入到 Execute 函数里面。

Execute 参数中有 ShadowFrame，能拿到 ArtMethod，再通过 GetDexFile() 函数获得 DEX 字节码等信息。



在 dex 中第一个类初始化的时候调用 dumpDexFileByExecute 脱壳

static inline JValue Execute(Thread\* self, const DexFile::CodeItem\* code\_item,

ShadowFrame& shadow\_frame, JValue result\_register) {

// 在类初始化的时候脱壳

if(strstr(PrettyMethod(shadow\_frame.GetMethod()).c\_str(),"<clinit>")!=nullptr) {

dumpDexFileByExecute(shadow\_frame.GetMethod());

}

...

}

在 dumpDexFileByExecute 中判断 如果 dex 文件不存在就 dump

// 该函数在 ART 执行期间调用，用于在 Execute 函数内完成 dex 脱壳

extern "C" void dumpDexFileByExecute(ArtMethod \*artmethod)

SHARED\_LOCKS\_REQUIRED(Locks::mutator\_lock\_) {

// 为 dex 文件保存路径分配内存

char \*dexfilepath = (char \*) malloc(sizeof(char) \* 2000);

if (dexfilepath == nullptr) {

// 分配失败，打印日志并返回

LOG(INFO) << "ArtMethod::dumpDexFileByExecute, methodname: "

<< PrettyMethod(artmethod).c\_str()

<< " malloc 2000 byte failed";

return;

}

// 获取当前进程的 cmdline 名称，用于后续命名脱壳文件

int fcmdline = -1;

char szCmdline[64] = { 0 };

char szProcName[256] = { 0 };

int procid = getpid();

sprintf(szCmdline, "/proc/%d/cmdline", procid);

fcmdline = open(szCmdline, O\_RDONLY, 0644);

if (fcmdline > 0) {

read(fcmdline, szProcName, 256);

close(fcmdline);

}

// 若成功获取到进程名

if (szProcName[0]) {

// 获取当前 ArtMethod 所属 dex 文件及其起始地址和大小

const DexFile \*dex\_file = artmethod->GetDexFile();

const uint8\_t \*begin\_ = dex\_file->Begin(); // dex 文件起始地址

size\_t size\_ = dex\_file->Size(); // dex 文件大小

int size\_int\_ = (int) size\_; // 用于命名文件

// 创建保存路径：/sdcard/fart/<process\_name>/

memset(dexfilepath, 0, 2000);

sprintf(dexfilepath, "/sdcard/fart");

mkdir(dexfilepath, 0777); // 创建 fart 目录

memset(dexfilepath, 0, 2000);

sprintf(dexfilepath, "/sdcard/fart/%s", szProcName);

mkdir(dexfilepath, 0777); // 创建子目录为进程名

// 拼接最终保存路径，如：/sdcard/fart/com.xxx.xxx/123456\_dexfile\_execute.dex

memset(dexfilepath, 0, 2000);

sprintf(dexfilepath,

"/sdcard/fart/%s/%d\_dexfile\_execute.dex",

szProcName, size\_int\_);

// 检查该文件是否已经存在，若存在则跳过写入

int dexfilefp = open(dexfilepath, O\_RDONLY, 0666); // 以只读方式尝试打开指定路径的 dex 文件

if (dexfilefp > 0) {

close(dexfilefp); // 已存在，关闭文件

dexfilefp = 0;

} else {

// 不存在则创建并写入 dex 内容

dexfilefp = open(dexfilepath, O\_CREAT | O\_RDWR, 0666);

if (dexfilefp > 0) {

write(dexfilefp, (void \*) begin\_, size\_);

fsync(dexfilefp); // 刷新到磁盘

close(dexfilefp); // 关闭文件

}

}

}

// 释放申请的内存

if (dexfilepath != nullptr) {

free(dexfilepath);

dexfilepath = nullptr;

}

}

路径：art/runtime/art\_method.cc

这时候已经可以把 Dex 整体 dump 下来了，但是还没有把抽空的函数修复，这个就需要 FART 中的主动调用组件来解决了。

关于 FART 中的主动调用组件设计与分析参考：[FART 主动调用组件设计和源码分析](https://cyrus-studio.github.io/blog/posts/fart-%E4%B8%BB%E5%8A%A8%E8%B0%83%E7%94%A8%E7%BB%84%E4%BB%B6%E8%AE%BE%E8%AE%A1%E5%92%8C%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90/)

相关文章：[拨云见日：安卓APP脱壳的本质以及如何快速发现ART下的脱壳点](https://bbs.kanxue.com/thread-254555.htm)