Kaiti SC

Android ART 分析

周坤[1]

中国科学技术大学 计算机科学与技术学院

March 22, 2017

[1]984040887@qq.com

目录

1 Android ART analysis

Android 源码目前更新到 5.1.1_r2,http://socialcompare.com/en/comparison/android-versions-comparison 上给出了 Android 历史版本的比较，这里截取了 5.0 版本之后的比较，如表??所示：

表 1: android version comparison

version	key user features added	key developer features added	release date
android5.0	New design (Material design); Speed improvement; Battery consumption improvement	Several new API; Tracking battery consumption app	2014 Oct 17
android5.0.1	bug fixes, fix issues with video playback and password failures		2014 Dec 2
android5.0.2	Performance improvements and bug fixes		2014 Dec 19
android5.1	Multiple SIM cards support; Quick settings shortcuts to join Wi-Fi networks or control Bluetooth devices; Lock protection if lost or stolen; High Definition voice call Stability and performance enhancements		2015 Mar 9
android5.1.1	Speed improvement; Bug fixes		2015 April 21

本文档分析的源码版本是 android5.0.0_r7, 简单看了一下 5.0 到 5.1.1 版本的更新日志 http://changes.droidsec.org/, 关于 ART 的修改是在 5.0.0 到 5.0.1；5.0.2 到 5.1.0，主要是一些细节的修改。本文档主要是分析 ART 运行过程的大体框架，没有涉及到一些细节的深入分析，所以与最新版本应该没有太大出入（后面再重新结合最新的源码修改本文档）。

1.1 Android Runtime

Android 系统 init 进程启动过程中会运行 app_process 进程，app_process 对应的源码位于/frameworks/base/cmds/app_main.cpp, 在 main 函数中会启动 Zygote:

```
if (zygote) {
          runtime.start("com.android.internal.os.ZygoteInit", args);
}
```

runtime 是 AppRuntime 的实例，AppRuntime 继承自 AndroidRuntime. 进入 AndroidRuntime 类的 start 函数（位于/frameworks/base/core/jni/AndroidRuntime.cpp），部分代码如下：

```
void AndroidRuntime::start(const char* className, const Vector<String8
       >& options)
{
  ......

  JniInvocation jni_invocation;
  jni_invocation.Init(NULL);
  JNIEnv* env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
  }

  ......
}
```

调用 JniInvocation 类的 Init 函数（位于/libnativehelper/JniInvocation.cpp），部分代码如下：

```
bool JniInvocation::Init(const char* library) {
  library = GetLibrary(library);

  handle_ = dlopen(library, RTLD_NOW);

  ......

  if (!FindSymbol(reinterpret_cast<void**>(&
       JNI_GetDefaultJavaVMInitArgs_),
                  "JNI_GetDefaultJavaVMInitArgs")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_CreateJavaVM_),
                  "JNI_CreateJavaVM")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetCreatedJavaVMs_),
                  "JNI_GetCreatedJavaVMs")) {
    return false;
```

```
  }
  return true;
}
```

Init 通过 GetLibrary 函数从属性系统获得名为 persist.sys.dalvik.vm.lib 的属性值，dalvik 模式，其值为 libdvm.so,art 模式下，其值为 libart.so. 通过 dlopen 打开对应的库文件，这里我们只考虑 libart.so. 然后调用 FindSymbol 函数从打开的 libart.so 文件中搜索到对应的导出符号，比如 JNI_CreateJavaVM 对应的是 /art/runtime/jni_internal.cc 中的 JNI_CreateJavaVM 函数。回到 AndroidRuntime::start 函数，接着会调用 startVM 函数启动虚拟机，该函数最终会调用 JNI_CreateJavaVM 函数，ART 模式对应的部分代码如下：

```
extern "C" jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void*
       vm_args) {
  const JavaVMInitArgs* args = static_cast<JavaVMInitArgs*>(vm_args);
  ......
  RuntimeOptions options;
    for (int i = 0; i < args->nOptions; ++i) {
    JavaVMOption* option = &args->options[i];
      options.push_back(std::make_pair(std::string(option->
       optionString),
                option->extraInfo));
    }
  if (!Runtime::Create(options, ignore_unrecognized)) {
    ......
  }
  Runtime* runtime = Runtime::Current();
  bool started = runtime->Start();
  ......
}
```

JNI_CreateJavaVM 函数首先解析虚拟机启动参数存入到 Runtime::Options 实例中，根据解析的参数信息，调用 Create 函数创建 Runtime 实例，获取 Runtime 的当前实例，接着调用 Runtime 的 Start 函数 (位于 art/runtime/runtime.cc)，该函数代码如下：

```
bool Runtime::Start() {
  ......
  Thread* self = Thread::Current();
    self->TransitionFromRunnableToSuspended(kNative);
    ......
  InitNativeMethods();
  ......
  if (is_zygote_) {
```

```
9       if (!InitZygote()) {
10          return false;
11      }
12     } else {
13       DidForkFromZygote(...);
14       }
15    StartDaemonThreads();
16      ......
17  }
```

Start 函数获取当前运行线程，将该线程状态从 Runnable 切换到 Suspend，通过 InitNativeMethods 完成 Native 函数的注册。调用 InitZygote 完成一些文件系统的增加。最后调用 StartDaemonThreads 启动守护进程。

1.2 PackageManagerService

PackageManagerService 在 Android 系统中负责 APK 包的管理和相关信息查询及应用程序的安装、卸载，这个服务负责扫描系统中特定的目录，找到里面的应用程序文件，即以 Apk 为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息，完成应用程序的安装过程。dex 到 oat 的转换过程就是在 PackageManagerService 中完成的。PackageManagerService (frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java) 由 SystemServer (frameworks/base/services/java/com/android/server/SystemServer.java) 创建。在创建 PackageManagerService 的实例时，会在 PackageManagerService 类的构造函数中开始执行安装应用程序的过程。通过调用 scanDirLI 函数扫描以下几个目录中的 apk 文件:

/system/framework

/system/app

/system/priv-app

/vendor/app

/vendor/overlay

/oem/app

接着调用 scanPackageLI 函数对 apk 文件进行解析和安装。scanPackageLI 通过创建 PackageParser 实例，调用这个实例的 parsePackage 方法解析 apk 文件。parsePackage 代码位于: frameworks/base/core/java/android/content/pm/PackageParser.java。该函数通过解析 apk 中的配置文件 AndroidManifest.xml 完成解析任务，最后将解析得到的信息保存到 PackageManagerService 服务中。这样，应用程序都注册到了 PackageManagerService

中，接着需要将这些应用程序在桌面上显示出来。Android 中 Launcher 负责从 PackageManagerService 服务中把这些安装好的应用程序取出来，并在桌面上展现出来。Launcher 由 ActivityManagerService 启动，ActivityManagerService 和 PackageManagerService 一样，都是在开机时由 SystemServer 组件启动的。

在 PackageManagerService 处理过程中，会通过调用另外一个类 Installer 的成员函数 dexopt 对 APK 内的 dex 字节码进行优化（frameworks/base/services/core/java/com/android/server/pm/Installer.java）。Installer 向守护进程 installd 发送一个 dexopt 请求，由 installd 内的 do_dexopt 函数来处理,do_dexopt 调用 dexopt 函数（位于 frameworks/native/cmds/installd/commands.c). dexopt 函数的部分代码如下：

```
int dexopt(const char *apk_path, uid_t uid, bool is_public,
           const char *pkgname, const char *instruction_set,
           bool vm_safe_mode, bool is_patchoat)
{
    ......

    property_get("persist.sys.dalvik.vm.lib.2",
        persist_sys_dalvik_vm_lib, "libart.so");

    ......

    if (strncmp(persist_sys_dalvik_vm_lib, "libdvm", 6) == 0) {
            run_dexopt(input_fd, out_fd, input_file, out_path);
        } else if (strncmp(persist_sys_dalvik_vm_lib, "libart", 6) ==
        0) {
            if (is_patchoat) {
                run_patchoat(input_fd, out_fd, input_file, out_path,
        pkgname, instruction_set);
            } else {
                run_dex2oat(input_fd, out_fd, input_file, out_path,
        pkgname, instruction_set,
                            vm_safe_mode);
            }
        } else {
            exit(69);   /* Unexpected persist.sys.dalvik.vm.lib value
        */
        }

    ......
}
```

dexopt 通过 property_get 函数读取系统属性 persist.sys.dalvik.vm.lib 的值。如

果该值为 libdvm.so, 就会调用函数 run_dexopt 即运行/system/bin/dexopt 将 dex 文件优化成 odex 文件；如果是 libart.so，接着判断 is_patchoat 的值，为真，调用 run_patchoat 即运行/system/bin/patchoat 对已经编译过的 oat 文件进行重定位 relocation，改变其基地址 base address；为假，则调用 run_dex2oat 级运行/system/bin/dex2oat 将 dex 文件转换为 oat 文件。

1.3 dex2oat

/system/bin/dex2oat 对应的源码文件位于/art/dex2oat/dex2oat.cc。main 函数直接调用 art 命名空间下的 dex2oat 函数，dex2oat 函数的主要结构如下所示：

```
//command-line argument parsing
......

//create oat file but not write data
std::unique_ptr<File> oat_file;
......

//create Dex2Oat
Dex2Oat::Create(...);
......

//extract classes.dex file
std::vector<const DexFile*> dex_files;
......

//create oat format file
dex2oat->CreateOatFile(...);
......

//create image file
dex2oat->CreateImageFile(...);
......
```

dex2oat 函数代码比较长，前面一部分都是在解析参数，对一些不正确的参数组合会调用 Usage 函数打印 dex2oat 的使用方法并退出。在完成参数的解析判断之后会创建一个指向 oat_location 的文件指针，代码如下：

```
std::unique_ptr<File> oat_file;
bool create_file = !oat_unstripped.empty();
if (create_file) {
  oat_file.reset(OS::CreateEmptyFile(oat_unstripped.c_str()));
  if (oat_location.empty()) {
```

```
    oat_location = oat_filename;
  }
 } else {
   oat_file.reset(new File(oat_fd, oat_location));
   oat_file->DisableAutoClose();
   oat_file->SetLength(0);
 }
```

声明文件指针变量 oat_file,if 和 else 分支都是在创建一个 File 实例并赋值给 oat_file , 这部分代码仅仅是创建了一个 File 实例， 没有真正写入 oat 格式的文件数据。

接着开始完成 dex 到 oat 的转换工作, 声明指向 Dex2Oat 的指针 p_dex2oat, 类 Dex2Oat 的定义位于 art/dex2oat/dex2oat.cc 中, 接着调用 Dex2Oat 的 Create 方法,

```
Dex2Oat* p_dex2oat;
if (!Dex2Oat::Create(&p_dex2oat,
                     runtime_options,
                     *compiler_options,
                     compiler_kind,
                     instruction_set,
                     instruction_set_features,
                     verification_results.get(),
                     &method_inliner_map,
                     thread_count)) {
    LOG(ERROR) << "Failed to create dex2oat";
    return EXIT_FAILURE;
  }
}
```

Dex2Oat 的 Create 方法代码如下:

```
static bool Create(Dex2Oat** p_dex2oat,...){
  SHARED_TRYLOCK_FUNCTION(true, Locks::mutator_lock_) {
    CHECK(verification_results != nullptr);
    CHECK(method_inliner_map != nullptr);
    std::unique_ptr<Dex2Oat> dex2oat(new  Dex2Oat(&compiler_options
        ,...))
    if (!dex2oat->CreateRuntime(runtime_options,instruction_set)) {
      *p_dex2oat = nullptr;
      return false;
    }
    *p_dex2oat = dex2oat.release();
    return true;
  }
```

```
13 }
```

Create 方法调用 CreateRuntime 函数获取 Runtime 类的实例，Runtime 使用单例模式，所以获取 Runtime 实例前会先获取锁。接着创建 Dex2Oat 的实例，Dex2Oat 的构造函数比较简单，只是一些类成员的赋值，如下所示：

```
1  explicit Dex2Oat(const CompilerOptions* compiler_options,...)
2        : compiler_options_(compiler_options),
3          compiler_kind_(compiler_kind),
4          instruction_set_(instruction_set),
5          instruction_set_features_(instruction_set_features),
6          verification_results_(verification_results),
7          method_inliner_map_(method_inliner_map),
8          runtime_(nullptr),
9          thread_count_(thread_count),
10         start_ns_(NanoTime()) {
11     CHECK(compiler_options != nullptr);
12     CHECK(verification_results != nullptr);
13     CHECK(method_inliner_map != nullptr);
14   }
```

其中 start_ns_ 记录实例化的时间。

回到 dex2oat 函数，接着将 p_dex2oat 赋值给 dex2oat 变量，获取当前线程，将线程从 runnable 切换到 suspend 状态，释放掉之前 Dex2Oat 创建时获得的锁。WellKnownClasses::Init 函数用来完成一些 JNI 类、函数、以及字段的初始化操作，函数代码位于 art/runtime/well_known_classes.cc.

接着声明 vector 变量 dex_files 保存相应的 DexFile 实例，if 分支的 GetBootClassPath 函数以及 else 分支的 DexFile::OpenFromZip 函数都是获取相应的 dex 文件。

接着通过 EnableWrite() 将 dex 文件设置为可写，保证能够进行 dex 到 dex 的优化转换。

最后调用 CreateOatFile 和 CreateImageFile 函数来分别创建 oat 文件和镜像文件。这里我们只关心 CreateOatFile 函数的实现过程。

```
1  const CompilerDriver* CreateOatFile(...) {
2      ......
3      std::unique_ptr<CompilerDriver> driver(new CompilerDriver(...));
4      driver->GetCompiler()->SetBitcodeFileName(*driver.get(),
           bitcode_filename);
5      driver->CompileAll(class_loader, dex_files, &timings);
6
7      OatWriter oat_writer(...);
8      ......
```

```
9
10      if (!driver->WriteElf(android_root, is_host, dex_files, &
            oat_writer, oat_file)) {
11      ......
12      }
13
14      ......
15  }
```

CreateOatFile 函数创建 CompilerDriver 实例，调用 CompilerDriver 类的 CompileAll 方法（源码位于/art/compiler/driver/compiler_driver.cc）来完成 dex 到 IR 的转换。CompileAll 方法的代码如下：

```
1  void CompilerDriver::CompileAll(jobject class_loader,
2                                  const std::vector<const DexFile*>&
           dex_files,
3                                  TimingLogger* timings) {
4    DCHECK(!Runtime::Current()->IsStarted());
5    std::unique_ptr<ThreadPool> thread_pool(new ThreadPool("Compiler␣
           driver␣thread␣pool", thread_count_ - 1));
6    PreCompile(class_loader, dex_files, thread_pool.get(), timings);
7    Compile(class_loader, dex_files, thread_pool.get(), timings);
8    if (dump_stats_) {
9      stats_->Dump();
10   }
11 }
```

CompileAll 主要包含两个函数：PreCompile 和 Compile。ProCompile 函数的代码：

```
1  void CompilerDriver::PreCompile(jobject class_loader,
2              const std::vector<const DexFile*>& dex_files,
3              ThreadPool* thread_pool, TimingLogger* timings) {
4     LoadImageClasses(timings);
5     Resolve(class_loader, dex_files, thread_pool, timings);
6     if (!compiler_options_->IsVerificationEnabled()) {
7     LOG(INFO) << "Verify␣none␣mode␣specified,␣skipping␣verification.";
8      SetVerified(class_loader, dex_files, thread_pool, timings);
9      return;
10    }
11
12   Verify(class_loader, dex_files, thread_pool, timings);
13
14   InitializeClasses(class_loader, dex_files, thread_pool, timings);
15
16    UpdateImageClasses(timings);
```

```
}
```

PreCompile 主要是进行编译前的一些预操作，主要是类的加载、解析、校验和初始化等。

Compile 函数直接调用 CompileDexFile 函数，该函数的代码：

```
void CompilerDriver::CompileDexFile(jobject class_loader, const
       DexFile& dex_file,
                                    const std::vector<const DexFile*>&
        dex_files,
                                    ThreadPool* thread_pool,
       TimingLogger* timings) {
  TimingLogger::ScopedTiming t("Compile␣Dex␣File", timings);
  ParallelCompilationManager context(Runtime::Current()->
       GetClassLinker(), class_loader, this,
                                     &dex_file, dex_files, thread_pool
       );
  context.ForAll(0, dex_file.NumClassDefs(), CompilerDriver::
       CompileClass, thread_count_);
}
```

通过类 ParallelCompilationManager 并行编译 dex 文件中的各个类。CompileClass 通过调用 CompilerDriver 类的 CompileMethod 来编译每个类对应的 direct methods 和 vitual methods。CompileMethod 方法的代码如下：

```
void CompilerDriver::CompileMethod(const DexFile::CodeItem* code_item,
        uint32_t access_flags,
                                   InvokeType invoke_type, uint16_t
       class_def_idx,
                                   uint32_t method_idx, jobject
       class_loader,
                                   const DexFile& dex_file,
                                   DexToDexCompilationLevel
       dex_to_dex_compilation_level) {
  CompiledMethod* compiled_method = nullptr;
  uint64_t start_ns = kTimeCompileMethod ? NanoTime() : 0;

  if ((access_flags & kAccNative) != 0) {
    if (!compiler_options_->IsCompilationEnabled() &&
        (instruction_set_ == kX86_64 || instruction_set_ == kArm64)) {
    } else {
      compiled_method = compiler_->JniCompile(access_flags, method_idx
       , dex_file);
      CHECK(compiled_method != nullptr);
    }
  } else if ((access_flags & kAccAbstract) != 0) {
```

```
17    } else {
18      MethodReference method_ref(&dex_file, method_idx);
19      bool compile = verification_results_->IsCandidateForCompilation(
            method_ref, access_flags);
20      if (compile) {
21        compiled_method = compiler_->Compile(code_item, access_flags,
            invoke_type, class_def_idx,
22                                             method_idx, class_loader,
            dex_file);
23      }
24      if (compiled_method == nullptr && dex_to_dex_compilation_level !=
            kDontDexToDexCompile) {
25        (*dex_to_dex_compiler_)(*this, code_item, access_flags,
26                                invoke_type, class_def_idx,
27                                method_idx, class_loader, dex_file,
28                                dex_to_dex_compilation_level);
29      }
30    }
31
32    ......
33
34    Thread* self = Thread::Current();
35    if (compiled_method != nullptr) {
36      MethodReference ref(&dex_file, method_idx);
37      DCHECK(GetCompiledMethod(ref) == nullptr) << PrettyMethod(
            method_idx, dex_file);
38      {
39        MutexLock mu(self, compiled_methods_lock_);
40        compiled_methods_.Put(ref, compiled_method);
41      }
42      DCHECK(GetCompiledMethod(ref) != nullptr) << PrettyMethod(
            method_idx, dex_file);
43    }
44
45    if (self->IsExceptionPending()) {
46      ScopedObjectAccess soa(self);
47      LOG(FATAL) << "Unexpected␣exception␣compiling:␣" << PrettyMethod(
            method_idx, dex_file) << "\n"
48          << self->GetException(nullptr)->Dump();
49    }
50  }
```

对于本地方法 native method，如果编译选项 IsCompilationEnabled 为假并且指令集为 64 位 X86 或者 64 位 Arm 时不做任何处理会触发通用的 JNI 调用，其他情况会调用编译器的 JniCompile 方法，Android ART 中存在两种 JniCompile 方法：一种位于 /art/compiler/compiler.cc, 通过调用 ArtL-

LVMJniCompileMethod （位于/art/compiler/llvm/compiler_llvm.cc） 完成。ArtLLVMJniCompileMethod 是通过调用 llvm JNI 编译器的 Compile 方法（源码位于 /art/compiler/jni/portable/jni_compiler.cc） 来完成本地方法的编译。另外一种位于/art/compiler/compilers.cc，通过调用 ArtQuickCompileMethod（位于/art/compiler/dex/frontend.cc） 完成。ArtQuickCompileMethod 最终是通过调用 CompileMethod 方法完成编译工作。CompileMethod （/art/compiler/dex/frontend.cc） 主要是通过创建 MIR Graph，对 MIR Graph 进行处理来完成方法的编译。

对于抽象方法，不做任何处理。

其他方法通过调用编译器的 Compile 方法进行处理。因为编译器选项有 3 个值：Quick、Optimizing、Portable，所以对应的有 3 种 Compile 方法。QuickCompiler::Compile(源码位于/art/compiler/compilers.cc） 先调用 TryCompileWithSeaIR 得到 method，若 method 为空，则返回 ArtQuickCompileMethod。TryCompileWithSeaIR 最终是调用 CompileMethodWithSeaIr（源码位于/art/compiler/sea_ir/frontend.cc），通过得到 dex 文件的 SeaGraph，基于 SeaGraph 进行操作，完成方法的编译。

OptimizingCompiler::Compile（源码位于/art/compiler/compilers.cc）先调用 TryCompile，如果得到的 method 为空，则调用 QuickCompiler::Compile 完成编译。TryCompile 位于/art/compiler/optimizing/optimizing_compiler.cc,TryCompile 的代码比较复杂，主要包含两个部分：通过 HGraphBuilder 构建 HGraph（方法对应的控制流图），基于 HGraph 进行一系列处理包括构建支配树、转换为 SSA 形式、活跃分析等；通过 CodeGenerator 创建代码生成器。

PortableCompiler::Compile（源码位于/art/compiler/compiler.cc） 先调用 TryCompileWithSeaIR 得到 method，若 method 为空，则返回 ArtCompileMethod。ArtCompileMethod 调用 CompileMethod 方法完成方法的编译。CompileMethod 的处理流程：

1、创建 CompilationUnit 对象来存放一次编译中需要的信息，compilationunit 相应成员的初始化；

2、将 dex 文件中的 Dalvik 字节码解码为 DecodedInstruction（这个结构体的定义位于 /art/compiler/dex/mir_graph.h），并创建对应的 MIR 节点；

3、定位基本块的边界，并创建相应的 BasicBlock 对象，将 MIR 塞进去；

4、确定控制流关系，将基本块连接起来构成控制流图（CFG），并添加恢复解释器状态和异常处理用的基本块；

5、将基本块都加到 CompilationUnit 里去；

6、MIR 优化（带有局部优化和全局优化）

1.4 oat 执行

ART 运行时提供了一个 OatFile 类，通过调用它的静态成员函数 Open 可以在本进程中加载 OAT 文件，这个函数定义在文件 art/runtime/oat_file.cc 中，

```
OatFile* OatFile::Open(const std::string& filename,
                       const std::string& location,
                       byte* requested_base,
                       bool executable,
                       std::string* error_msg) {
  ..............

}
```

参数 filename 和 location 指向要加载的 OAT 文件，参数 requested_base 是一个可选参数，用来描述要加载的 OAT 文件里面的 oatdata 段要加载的位置，参数 executable 表示要加载的 OAT 是不是应用程序的主执行文件。一般来说，一个应用程序只有一个 classes.dex 文件，这个 classes.dex 文件经过编译后，就得到一个 OAT 主执行文件。不过，应用程序也可以在运行时动态加载 DEX 文件。这些动态加载的 DEX 文件在加载的时候同样会被翻译成 OAT 再运行，就不属于主执行文件了。

ART 运行时利用 LLVM 编译框架将 DEX 字节码翻译成本地机器指令，这些生成的机器指令就保存在 ELF 文件格式的 OAT 文件的 oatexec 段中.ART 运行时会为每一个类方法都生成一系列的本地机器指令，这些本地机器指令不是孤立存在的，因为它们可能需要其它的函数来完成自己的功能。这要求 Backend 为类方法生成本地机器指令时，要处理调用其它模块提供的函数的问题。ART 运行时支持两种类型的 Backend：Portable 和 Quick。

Portable 类型的 Backend 通过集成在 LLVM 编译框架里面的一个称为 MCLinker 的链接器来生成本地机器指令。这些 OAT 文件要通过系统的动态链接器提供的 dlopen 函数来加载。函数 dlopen 在加载 OAT 文件的时候，会通过重定位技术来处理好它与其它模块的依赖关系，使得它能够调用其它模块提供的接口。

Quick 类型的 Backend 生成的本地机器指令用另外一种方式来处理依赖模块之间的依赖关系,ART 运行时会在每一个线程的 TLS（线程本地区域）提供一个函数表,Quick 类型的 Backend 生成的本地机器指令通过引用这个函数表来调用其它模块的函数。也就是说，Quick 类型的 Backend 生成的本地机器指令要依赖于 ART 运运时提供的函数表，这使得 Quick 类型的 Backend

生成的 OAT 文件在加载时不需要重定位，因此就不需要通过系统的动态链接器提供的 dlopen 函数来加载。由于省去重定位这个操作，Quick 类型的 Backend 生成的 OAT 文件在加载时就会更快，这也是称为 Quick 的缘由。

如果在编译 ART 运行时时，定义了宏 ART_USE_PORTABLE_COMPILER，那么就表示要使用 Portable 类型的 Backend 来生成 OAT 文件，否则就使用 Quick 类型的 Backend 来生成 OAT 文件。Open 函数的实现过程：

1. 如果编译时指定了 ART_USE_PORTABLE_COMPILER 宏，并且参数 executable 为 true，那么就通过 OatFile 类的静态成员函数 OpenDlopen 来加载指定的 OAT 文件。OatFile 类的静态成员函数 OpenDlopen 直接通过动态链接器提供的 dlopen 函数来加载 OAT 文件。

2. 其余情况下，通过 OatFile 类的静态成员函数 OpenElfFile 来加载指定的 OAT 文件。这种方式是按照 ELF 文件格式来解析要加载的 OAT 文件的，并且根据解析获得的信息将 OAT 里面相应的段加载到内存中来。

OatFile 类的成员函数 Dlopen 首先是通过动态链接器提供的 dlopen 函数将参数 elf_filename 指定的 OAT 文件加载到内存中来，接着同样是通过动态链接器提供的 dlsym 函数从加载进来的 OAT 文件获得两个导出符号 oatdata 和 oatlastword 的地址，最后调用成员函数 Setup 来解析已经加载内存中的 oatdata 段，以获得 ART 运行所需要的更多信息。

函数 Setup 定义在文件 art/runtime/oat_file.cc 中，通过分析 Setup 函数更好地了解 oat 文件格式。

函数 Setup 的一开始调用了函数 GetOatHeader，因此 OAT 文件里面的 oatdata 段的开始储存着一个 OAT 头，这个 OAT 头通过类 OatHeader 描述，定义在文件 art/runtime/oat.h 中

然后调用函数 GetImageFileLocationSize 得到正在打开的 OAT 依赖的 Image 空间文件的路径大小，因此紧接着在 OAT 头后面的是 Image 空间文件路径

接着的代码获得包含在 oatdata 段的 DEX 文件描述信息，每一个 DEX 文件记录在 oatdata 段的描述信息包括：

DEX 文件路径大小，保存在变量 dex_file_location_size 中

DEX 文件路径，保存在变量 dex_file_location_data 中

DEX 文件检验和，保存在变量 dex_file_checksum 中

DEX 文件内容在 oatdata 段的偏移，保存在变量 dex_file_offset 中

DEX 文件包含的类的本地机器指令信息偏移数组，保存在变量 methods_offsets_pointer 中

上述得到的每一个 DEX 文件的信息都被封装在一个 OatDexFile 对象

中，以便以后可以直接访问，在 OAT 文件中，每一个 DEX 文件包含的每一个类的描述信息都通过一个 OatClass 对象来描述

```
OatFile::OatClass::OatClass(const OatFile* oat_file,
                            mirror::Class::Status status,
                            OatClassType type,
                            uint32_t bitmap_size,
                            const uint32_t* bitmap_pointer,
                            const OatMethodOffsets* methods_pointer)
    : oat_file_(oat_file), status_(status), type_(type),
      bitmap_(bitmap_pointer), methods_pointer_(methods_pointer) {}
```

参数 oat_file 描述的是宿主 OAT 文件，参数 status 描述的是 OAT 类状态，参数 methods_pointer 是一个数组，描述的是 OAT 类的各个方法的信息，它们被分别保存在 OatClass 类的相应成员变量中。通过这些信息，我们就可以获得包含在该 DEX 文件里面的类的所有方法的本地机器指令信息。