OatFile类的静态成员函数Open的实现虽然只有寥寥几行代码，但是要理解它还得先理解宏ART\_USE\_PORTABLE\_COMPILER的的作用。在前面Android运行时ART简要介绍和学习计划一文中提到，ART运行时利用LLVM编译框架来将DEX字节码翻译成本地机器指令，其中要通过一个称为Backend的模块来生成本地机器指令。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。例如，它们可能需要调用ART运行时的堆管理系统提供的接口来为对象分配内存空间。这样就会涉及到一个模块依赖性问题，就好像我们在编写程序时，需要依赖C库提供的接口一样。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。关于MCLinker的更多知识，可以参考https://code.google.com/p/mclinker。简单来说，假设我们有一个模块A，它依赖于模块B、C和D，那么在为模块A生成本地机器指令时，指出它依赖于模块B、C和D就行了。在生成的OAT文件中会记录好这些依赖关系，这是ELF文件格式本来就支持的特性。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。这个实际上就通用的编译器、静态连接器以及动态链接器合作在一起干的事情，MCLinker扮演的就是静态链接器的角色。既然是通用的技术，因为就称能产生这种OAT文件的Backend为Portable类型的。

另一方面，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。简单来说，就是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。有了这个函数表之后，Quick类型的Backend生成的本地机器指令就可以通过它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运行时提供的函数表。这使得Quick类型的Backend生成的OAT文件在加载时不需要再处理模式之间的依赖关系。再通俗一点说的就是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。

关于ART运行时类型为Portable和Quick两种类型的Backend，我们就暂时讲解到这里，后面分析ART运行时执行类方法的时候，我们再详细分析。现在我们需要知道的就是，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。默认情况下，使用的是Quick类型的Backend。

接下就可以很好地理解OatFile类的静态成员函数Open的实现了：

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来手动加载指定的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。

接下来我们就分别看看OatFile类的静态成员函数OpenDlopen和OpenElfFile的实现，以便可以对OAT文件有更清楚的认识。

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，分别保存在当前正在处理的OatFile对象的成员变量begin\_和end\_中。根据图1所示，符号oatdata的地址即为OAT文件里面的oatdata段加载到内存中的开始地址，而符号oatlastword的地址即为OAT文件里面的oatexec加载到内存中的结束地址。符号oatlastword本身也是属于oatexec段的，它自己占用了一个地址，也就是sizeof(uint32\_t)个字节，于是将前面得到的end\_值加上sizeof(uint32\_t)，得到的才是oatexec段的结束地址。

实际上，上面得到的begin\_值指向的是加载内存中的oatdata段的头部，即OAT头。这个OAT头描述了OAT文件所包含的DEX文件的信息，以及定义在这些DEX文件里面的类方法所对应的本地机器指令在内存的位置。另外，上面得到的end\_是用来在解析OAT头时验证数据的正确性的。此外，如果参数requested\_base的值不等于0，那么就要求oatdata段必须要加载到requested\_base指定的位置去，也就是上面得到的begin\_值与requested\_base值相等，否则的话就会出错返回。

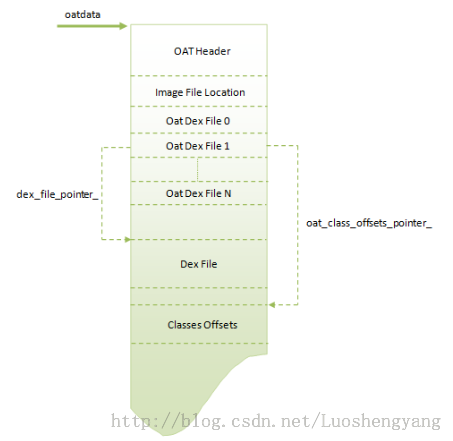
最后，OatFile类的成员函数Dlopen通过调用另外一个成员函数Setup来解析已经加载内存中的oatdata段

OatFile类的静态成员函数OpenElfFile的实现与前面分析的成员函数Dlopen是很类似的，唯一不同的是前者通过ElfFile类来手动加载参数file指定的OAT文件，实际上就是按照ELF文件格式来解析参数file指定的OAT文件，并且将文件里面的oatdata段和oatexec段加载到内存中来。我们可以将ElfFile类看作是ART运行时自己实现的OAT文件动态链接器。一旦参数file指定的OAT文件指定的文件加载完成之后，我们同样是通过两个导出符号oatdata和oatlastword来获得oatdata段和oatexec段的起止位置。同样，如果参数requested\_base的值不等于0，那么就要求oatdata段必须要加载到requested\_base指定的位置去。

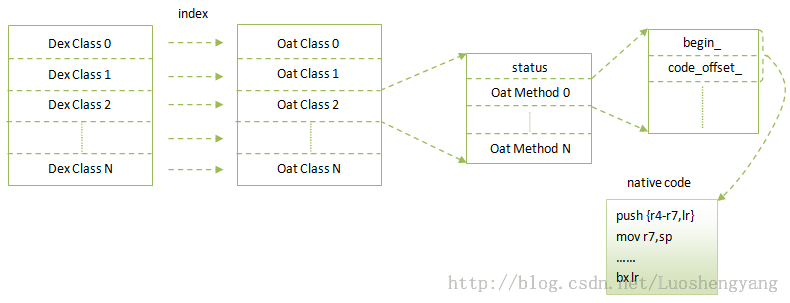
将参数file指定的OAT文件加载到内存之后，OatFile类的静态成员函数OpenElfFile最后也是调用OatFile类的成员函数Setup来解析其中的oatdata段。OatFile类的成员函数Setup定义在文件art/runtime/oat\_file.cc中，我们分三部分来阅读，以便可以更好地理解OAT文件的格式。

 通过第5个信息我们可以找到DEX文件里面的每一个类方法对应的本地机器指令。这个数组的大小等于header->class\_defs\_size\_，即DEX文件里面的每一个类在数组中都对应有一个偏移值。这里的header指向的是DEX文件头，它的class\_defs\_size\_描述了DEX文件包含的类的个数。在DEX文件中，每一个类都是有一个从0开始的编号，该编号就是用来索引到上述数组的，从而获得对应的类所有方法的本地机器指令信息。

        最后，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。如果我们使用OatDexFile来描述每一个DEX文件的描述信息，那么就可以通过图4看到这些描述信息在oatdata段的位置：



  通过上面对OAT文件加载过程的分析，我们就可以清楚地看到OAT文件的格式，以及如何在OAT文件中找到一个类方法的本地机器指令。我们通过图7来总结在OAT文件中找到一个类方法的本地机器指令的过程：



我们从左往右来看图7。首先是根据类签名信息从包含在OAT文件里面的DEX文件中查找目标Class的编号，然后再根据这个编号找到在OAT文件中找到对应的OatClass。接下来再根据方法签名从包含在OAT文件里面的DEX文件中查找目标方法的编号，然后再根据这个编号在前面找到的OatClass中找到对应的OatMethod。有了这个OatMethod之后，我们就根据它的成员变量begin\_和code\_offset\_找到目标类方法的本地机器指令了。其中，从DEX文件中根据签名找到类和方法的编号要求对DEX文件进行解析，这就需要利用Dalvik虚拟机的知识了。