# 历史背景

Graal虚拟机以及Graal编译器仍在实验室中尚未商用，但未来其有望代替或成为HotSpot下一代技术基础。Graal编译器最初是在Maxine虚拟机中作为C1X编译器的下一代编译器而设计的，所以它理所当然地使用于Java语言来编写。2012年，Graal编译器从Maxine虚拟机项目中分离，成为一个独立发展的Java编译器项目，Oracle Labs希望它最终能够成为一款高编译效率、高输出质量、支持提前编译和即时编译，同时支持应用于包括HotSpot在内的不同虚拟机的编译器。

由于这个编译器使用Java编写，代码清晰，又继承了许多来自HotSpot的服务端编译器的高质量优化技术，所以无论是科技企业还是高校研究院，都愿意在它上面研究和开发新编译技术。HotSpot服务端编译器的创造者Cliff Click自己就对Graal编译器十分推崇，并且公开表示再也不会用C、C++去编写虚拟机和编译器了。Twitter的Java虚拟机团队也曾公开说过C2目前犹如一潭死水，亟待一个替代品，因为在它上面开发、改进实在太困难了。

Graal编译器在JDK 9时以Jaotc提前编译工具的形式首次加入到官方的JDK中，从JDK 10起，Graal编译器可以替换服务端编译器，成为HotSpot分层编译中最顶层的即时编译器。这种可替换的即时编译器架构的实现，得益于HotSpot编译器接口的出现。

早期的Graal曾经同C1及C2一样，与HotSpot的协作是紧耦合的，这意味着每次编译Graal均需重新编译整个HotSpot。JDK 9时发布的JEP 243：Java虚拟机编译器接口（Java-Level JVM Compiler Interface，JVMCI）使得Graal可以从HotSpot的代码中分离出来。JVMCI主要提供如下三种功能：

* 响应HotSpot的编译请求，并将该请求分发给Java实现的即时编译器。
* 允许编译器访问HotSpot中与即时编译相关的数据结构，包括类、字段、方法及其性能监控数据等，并提供了一组这些数据结构在Java语言层面的抽象表示。
* 提供HotSpot代码缓存（Code Cache）的Java端抽象表示，允许编译器部署编译完成的二进制机器码。

综合利用上述三项功能，我们就可以把一个在HotSpot虚拟机外部的、用Java语言实现的即时编译器（不局限于Graal）集成到HotSpot中，响应HotSpot发出的最顶层的编译请求，并将编译后的二进制代码部署到HotSpot的代码缓存中。

此外，单独使用上述第三项功能，又可以绕开HotSpot的即时编译系统，让该编译器直接为应用的类库编译出二进制机器码，将该编译器当作一个提前编译器去使用（如Jaotc）。

# 构建编译调试环境

由于Graal编译器要同时支持Graal VM下的各种子项目，如Truffle、Substrate VM、Sulong等，还要支持作为HotSpot和Maxine虚拟机的即时编译器，所以只用Maven或Gradle的话，配置管理过程会相当复杂。为了降低代码管理、依赖项管理、编译和测试等环节的复杂度，Graal团队专门用Python 2写了一个名为mx的小工具来自动化做好这些事情。我们要构建Graal的调试环境，第一步要先把构建工具mx安装好，这非常简单，进行如下操作即可：

|  |
| --- |
| $ git clone https://github.com/graalvm/mx.git  $ export PATH=`pwd`/mx:$PATH |

'mx vm'等同于'java'

既然Graal编译器是以Java代码编写的，那第二步自然是要找一个合适的JDK来编译。考虑到Graal VM项目是基于OpenJDK 8开发的，而JVMCI接口又在JDK 9以后才会提供，所以Graal团队提供了一个带有JVMCI功能的OpenJDK 8版本，我们可以选择这个版本的JDK 8来进行编译。

|  |
| --- |
| $ export JAVA\_HOME=/development/oraclejdk1.8.0\_212-jvmci-20-b01 |

第三步是获取Graal编译器代码，编译器部分的代码是与整个Graal VM放在一块的，我们把Graal VM复制下来，大约有700MB，操作如下：

|  |
| --- |
| $ git clone https://github.com/graalvm/graal.git |

其他目录中存放着Truffle、Substrate VM、Sulong等其他项目，这些在本次实战中不会涉及。进入compiler子目录，使用mx构建Graal编译器，操作如下：

|  |
| --- |
| $ cd graal/compiler  $ mx build |

但编译的时候提示“Graal requires JDK21 or later”，所以就切换到jdk21来编译了

由于整个构建过程需要的依赖项都可以自动处理，需要手动处理的只有OpenJDK一个，所以编译一般不会出现什么问题，大概两三分钟编译即可完成。此时其实已经可以修改、调试Graal编译器了，但写Java代码不同于C、C++，应该没有人会直接用VIM去做Java开发调试，我们还是需要一个IDE来支持本次实战的。

mx工具能够支持Eclipse、Intellij IDEA和NetBeans三种主流的Java IDE项目的创建（Graal团队中使用Eclipse占多数，支持也最好），操作如下：

|  |
| --- |
| $ cd graal/compiler  # 生成指定编辑器的工程配置文件  $ mx eclipseinit  $ mx intellijinit  $ mx netbeansinit |

成功后：

|  |
| --- |
| Please restart Eclipse instances for this workspace to see some of the effects.  ----------------------------------------------  Eclipse project generation successfully completed for:  /development/graal/compiler  /development/graal/regex  /development/graal/sdk  /development/graal/truffle  The recommended next steps are:  1) Open Eclipse with workspace path: /development  2) Open project import wizard using: File -> Import -> Existing Projects into Workspace -> Next.  3) For "select root directory" enter path /development  4) Make sure "Search for nested projects" is checked and press "Finish".  hint) If you select "Close newly imported projects upon completion" then the import is more efficient.  Projects needed for development can be opened conveniently using the generated Suite working sets from the context menu.  5) Update the type filters (Preferences -> Java -> Appearance -> Type Filters) so that `jdk.\*` and `org.graalvm.\*` are not filtered.  Without this, code completion will not work for JVMCI and Graal code.  Note that setting MX\_BUILD\_EXPLODED=true can improve Eclipse build times. See "Exploded builds" in the mx README.md.  ----------------------------------------------  Ensure that these Execution Environments have a Compatible JRE in Eclipse (Preferences -> Java -> Installed JREs -> Execution Environments):  JavaSE-21 |

无论使用哪种IDE，都需要把IDE配置中使用的Java堆修改到2GB或以上，才能保证Graal在IDE中的编译构建能够顺利进行。譬如Eclipse默认配置（eclipse.ini文件）下的Java堆最大为1GB，，在Eclipse中选择File->Open Projects from File System，再选择Graal项目的根目录，将

会导入整个Graal VM，导入的工程如图：