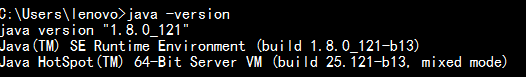
## Part 1 OpenJDK

### 1.1 Hotspot简介

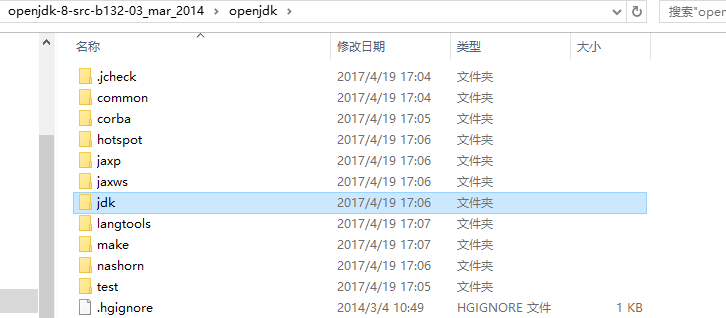
[Reference](https://www.cnblogs.com/dennyzhangdd/p/6734933.html)

Hotspot：全称 Java HotSpot Performance Engine，是 Java 虚拟机的一个实现，包含了服务器版和桌面应用程序版。利用 JIT 及自适应优化技术（自动查找性能热点并进行动态优化）来提高性能。

使用 java -version 可以查看 Hotspot 的版本



### 1.2 目录结构



—— corba：不流行的多语言、分布式通讯接口   
—— hotspot：Java 虚拟机   
—— jaxp：XML 处理   
—— jaxws：一组 XML web services 的 [**Java**](http://lib.csdn.net/base/java)API   
—— jdk：java 开发工具包   
—— —— 针对[**操作系统**](http://lib.csdn.net/base/operatingsystem)的部分   
—— —— share：与平台无关的实现   
—— langtools：Java 语言工具   
—— nashorn：JVM 上的 [**JavaScript**](http://lib.csdn.net/base/javascript) 运行时

VM目录结构



## Part 2 oop-klass model

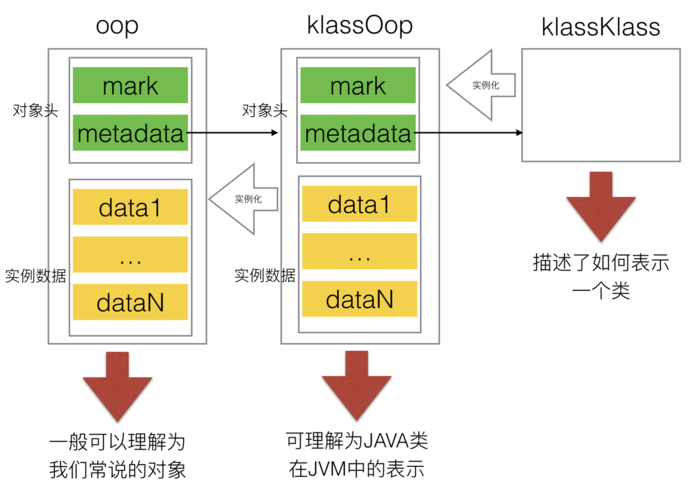
[Reference](http://www.hollischuang.com/archives/1910)

HotSpot JVM并没有根据Java对象直接通过虚拟机映射到新建的C++对象，而是设计了一个oop/klass model。

oops模块可以分成两个相对独立的部分：OOP框架和Klass框架。oop表示对象的实例信息；klass用来保存描述元数据。

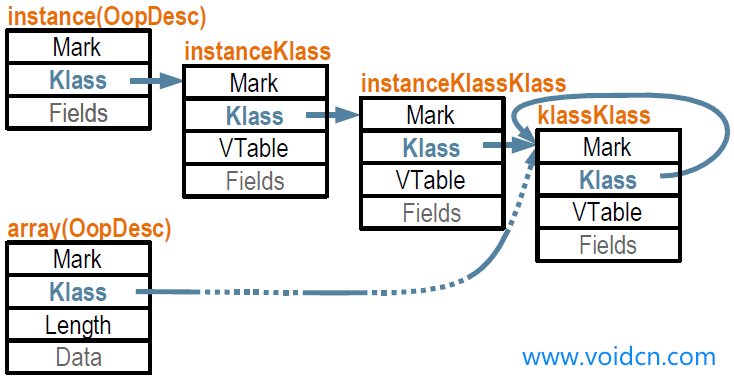
关于为何要设计oop/klass这种二分模型的实现，一个原因是不想让每个对象都包含vtbl(虚方法表)，其中oop中不含有任何虚函数，虚函数表保存于klass中，可以进行method dispatch。

### 2.3 关系图



klass链

在JVM中，对象在内存中的基本存在形式就是oop。那么，对象所属的类，在JVM中也是一种对象，因此它们实际上也会被组织成一种oop，即klassOop。同样的，对于klassOop，也有对应的一个klass来描述，它就是klassKlass，也是klass的一个子类。klassKlass作为oop的klass链的端点。关于对象和数组的klass链大致如下图：



图中instance即为instanceOopDesc

instanceOopDesc🡪instanceKlass--

### 2.1 oop

存在于VM中的对象

1. //定义了oops共同基类
2. typedef **class**   oopDesc\*                            oop;
3. //表示一个Java类型实例
4. typedef **class**   instanceOopDesc\*            instanceOop;
5. //表示一个Java方法
6. typedef **class**   methodOopDesc\*                    methodOop;
7. //表示一个Java方法中的不变信息
8. typedef **class**   constMethodOopDesc\*            constMethodOop;
9. //记录性能信息的数据结构
10. typedef **class**   methodDataOopDesc\*            methodDataOop;
11. //定义了数组OOPS的抽象基类
12. typedef **class**   arrayOopDesc\*                    arrayOop;
13. //表示持有一个OOPS数组
14. typedef **class**   objArrayOopDesc\*            objArrayOop;
15. //表示容纳基本类型的数组
16. typedef **class**   typeArrayOopDesc\*            typeArrayOop;
17. //表示在Class文件中描述的常量池
18. typedef **class**   constantPoolOopDesc\*            constantPoolOop;
19. //常量池告诉缓存
20. typedef **class**   constantPoolCacheOopDesc\*   constantPoolCacheOop;
21. //描述一个与Java类对等的C++类
22. typedef **class**   klassOopDesc\*                    klassOop;
23. //表示对象头
24. typedef **class**   markOopDesc\*                    markOop;

#### 2.1.1 OopDesc

对象基类 instanceOopDesc ：oopDesc。

[**jdk7u-hotspot**](https://github.com/openjdk-mirror/jdk7u-hotspot/tree/50bdefc3afe944ca74c3093e7448d6b889cd20d1)**/[src](https://github.com/openjdk-mirror/jdk7u-hotspot/tree/50bdefc3afe944ca74c3093e7448d6b889cd20d1/src)/**[**share**](https://github.com/openjdk-mirror/jdk7u-hotspot/tree/50bdefc3afe944ca74c3093e7448d6b889cd20d1/src/share)**/[vm](https://github.com/openjdk-mirror/jdk7u-hotspot/tree/50bdefc3afe944ca74c3093e7448d6b889cd20d1/src/share/vm)/**[**oops**](https://github.com/openjdk-mirror/jdk7u-hotspot/tree/50bdefc3afe944ca74c3093e7448d6b889cd20d1/src/share/vm/oops)**/oop.hpp**

1. **class** oopDesc {
2. friend **class** VMStructs;
3. **private**:   //Header
4. **volatile** markOop  \_mark;
5. union \_metadata {
6. wideKlassOop    \_klass;
7. narrowOop       \_compressed\_klass;
8. } \_metadata;
10. **private**:
11. // field addresses in oop
12. **void**\*     field\_base(**int** offset)        **const**;
14. jbyte\*    byte\_field\_addr(**int** offset)   **const**;
15. jchar\*    char\_field\_addr(**int** offset)   **const**;
16. jboolean\* bool\_field\_addr(**int** offset)   **const**;
17. jint\*     int\_field\_addr(**int** offset)    **const**;
18. jshort\*   short\_field\_addr(**int** offset)  **const**;
19. jlong\*    long\_field\_addr(**int** offset)   **const**;
20. jfloat\*   float\_field\_addr(**int** offset)  **const**;
21. jdouble\*  double\_field\_addr(**int** offset) **const**;
22. address\*  address\_field\_addr(**int** offset) **const**;
23. }

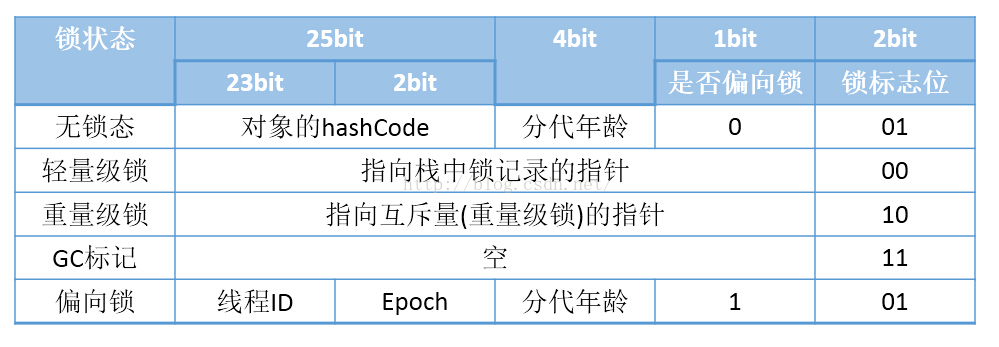
26. **class** instanceOopDesc : **public** oopDesc {
27. }
29. **class** arrayOopDesc : **public** oopDesc {
30. }

可见对象在内存中结构为三块区域：对象头、实例数据和对齐填充。

对象头包括\_mark和\_metadata

##### \_mark

32位的HotSpot虚拟机对象头存储结构：



用于存储对象自身的运行时数据。

1. **class** markOopDesc: **public** oopDesc {
2. **private**:
3. // Conversion
4. uintptr\_t value() **const** { **return** (uintptr\_t) **this**; }
6. **public**:
7. // Constants
8. **enum** { age\_bits                 = 4,
9. lock\_bits                = 2,
10. biased\_lock\_bits         = 1,
11. max\_hash\_bits
12. hash\_bits
13. cms\_bits                 = LP64\_ONLY(1) NOT\_LP64(0),
14. epoch\_bits               = 2
15. };

**age\_bits：**对象的分代年龄

**lock\_bits：** 锁状态标识位

**biased\_lock：** 偏向锁标识位

**hash\_bits：** 保存对象的哈希码  
**JavaThread\*：**保存持有偏向锁的线程ID  
**epoch：** 保存偏向时间戳

##### \_metadata

其中\_klass是普通指针，指向它的类的元数据。\_compressed\_klass是压缩类指针

#### 2.1.2 klassOop

#### 2.1.2 instanceOopDesc

1. **class** instanceOopDesc : **public** oopDesc

new创建Java对象实例时，JVM会创建一个instanceOopDesc对象来表示这个Java对象。

### 2.2 klass

实现语言层面的Java类。描述java的类元数据。不会存在VM中

1. //klassOop的一部分，用来描述语言层的类型
2. **class**  Klass;
3. //在虚拟机层面描述一个Java类
4. **class**   instanceKlass;
5. //专有instantKlass，表示java.lang.Class的Klass
6. **class**     instanceMirrorKlass;
7. //专有instantKlass，表示java.lang.ref.Reference的子类的Klass
8. **class**     instanceRefKlass;
9. //表示methodOop的Klass
10. **class**   methodKlass;
11. //表示constMethodOop的Klass
12. **class**   constMethodKlass;
13. //表示methodDataOop的Klass
14. **class**   methodDataKlass;
15. //最为klass链的端点，klassKlass的Klass就是它自身
16. **class**   klassKlass;
17. //表示instanceKlass的Klass
18. **class**     instanceKlassKlass;
19. //表示arrayKlass的Klass
20. **class**     arrayKlassKlass;
21. //表示objArrayKlass的Klass
22. **class**       objArrayKlassKlass;
23. //表示typeArrayKlass的Klass
24. **class**       typeArrayKlassKlass;
25. //表示array类型的抽象基类
26. **class**   arrayKlass;
27. //表示objArrayOop的Klass
28. **class**     objArrayKlass;
29. //表示typeArrayOop的Klass
30. **class**     typeArrayKlass;
31. //表示constantPoolOop的Klass
32. **class**   constantPoolKlass;
33. //表示constantPoolCacheOop的Klass
34. **class**   constantPoolCacheKlass;

#### instanceKlass

instanceKlass对象用来在JVM层表示Java类。

1. //类拥有的方法列表
2. objArrayOop     \_methods;
3. //描述方法顺序
4. typeArrayOop    \_method\_ordering;
5. //实现的接口
6. objArrayOop     \_local\_interfaces;
7. //继承的接口
8. objArrayOop     \_transitive\_interfaces;
9. //域
10. typeArrayOop    \_fields;
11. //常量
12. constantPoolOop \_constants;
13. //类加载器
14. oop             \_class\_loader;
15. //protected域
16. oop             \_protection\_domain;
17. ....

## Part 3 Load JVM

### 3.1 解释器

### 3.2 load过程

[Reference](https://www.jianshu.com/p/b91258bc08ac)

虚拟机的启动入口位于share/tools/launcher/java.c的main方法，整个流程分为如下几个步骤：  
1、配置JVM装载环境  
2、解析虚拟机参数  
3、设置线程栈大小  
4、执行Java main方法

#### 1 配置JVM装载环境

**CreateExecutionEnvironment**

**LoadJavaVM**

if (!LoadJavaVM(jvmpath, &ifn)) {

exit(6);

}

#### 2解析虚拟机参数

装载完JVM环境之后，需要对启动参数进行解析  
SetClassPath(s);🡪AddOption

#### 3 设置线程栈大小

1. **if** (threadStackSize == 0) {
2. **struct** JDK1\_1InitArgs args1\_1;
3. memset((**void**\*)&args1\_1, 0, **sizeof**(args1\_1));
4. args1\_1.version = JNI\_VERSION\_1\_1;
5. ifn.GetDefaultJavaVMInitArgs(&args1\_1);  /\* ignore return value \*/
6. **if** (args1\_1.javaStackSize > 0) {
7. threadStackSize = args1\_1.javaStackSize;
8. }
9. }

如果启动参数未设置-Xss，即threadStackSize为0，则调用InvocationFunctions的GetDefaultJavaVMInitArgs方法获取JavaVM的初始化参数，即调用JVM.dll函数JNI\_GetDefaultJavaVMInitArgs，定义在share\vm\prims\jni.cpp，实现如下

#### 4 执行Java main方法

**创建新线程，并执行JavaMain函数**

1. { /\* Create a new thread to create JVM and invoke main method \*/
2. **struct** JavaMainArgs args;
3. args.argc = argc;
4. args.argv = argv;
5. args.jarfile = jarfile;
6. args.classname = classname;
7. args.ifn = ifn;
9. **return** ContinueInNewThread(JavaMain, threadStackSize, (**void**\*)&args);
10. }

**加载主类Class**

Java运行方式有两种：jar方式和class方式

in Javamain

1. **if** (jarfile != 0) {  //jar方式
2. mainClassName = GetMainClassName(env, jarfile);
3. classname = (**char** \*)(\*env)->GetStringUTFChars(env, mainClassName, 0);
4. mainClass = LoadClass(env, classname);
5. (\*env)->ReleaseStringUTFChars(env, mainClassName, classname);
6. } **else** {
7. mainClassName = NewPlatformString(env, classname);
8. classname = (**char** \*)(\*env)->GetStringUTFChars(env, mainClassName, 0);
9. mainClass = LoadClass(env, classname);
10. (\*env)->ReleaseStringUTFChars(env, mainClassName, classname);
11. }

**查找main方法**

1. /\* Get the application's main method \*/
2. mainID = (\*env)->GetStaticMethodID(env, mainClass, "main",
3. "([Ljava/lang/String;)V");

**执行main**

1. mainArgs = NewPlatformStringArray(env, argv, argc);
2. /\* Invoke main method. \*/
3. (\*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs)

## Part 3 initialize

### 3.2 类加载

[Reference](https://www.jianshu.com/p/252e27863822)

Java ClassLoader

1. **public** abstract **class** ClassLoader {
3. **private** native Class defineClass1(String name, byte[] b, **int** off, **int** len,               ProtectionDomain pd, String source);
5. }

可以发现loadClass方法最终会执行native方法defineClass1进行类的加载，即读取对应class文件的二进制数据到虚拟机中进行解析；

#### class文件的解析

实现位于ClassLoader.c文件中，最终调用jvm.cpp中的jvm\_define\_class\_common方法实现，核心的实现逻辑如下

#### 创建instanceKlass实例

TODO

### 3.3 实例化

[Reference](https://www.jianshu.com/p/0009aaac16ed)

Java中的new关键字对应jvm中的new指令，定义在InterpreterRuntime类中

\src\share\vm\Interpreter\InterpreterRuntime.cpp

1. IRT\_ENTRY(**void**, InterpreterRuntime::\_new(JavaThread\* **thread**, constantPoolOopDesc\* pool, **int** index))
2. klassOop k\_oop = pool->klass\_at(index, CHECK);
3. instanceKlassHandle klass (THREAD, k\_oop);
5. // Make sure we are not instantiating an abstract klass
6. klass->check\_valid\_for\_instantiation(**true**, CHECK);
8. // Make sure klass is initialized
9. klass->initialize(CHECK);
11. oop obj = klass->allocate\_instance(CHECK);
12. **thread**->set\_vm\_result(obj);
13. IRT\_END

pool->klass\_at 返回BBB对应的klassOop对象

klass->initialize实现

### 3.4 GC垃圾收集器

## 第四篇 实现细节

### 4.1 锁

[Reference](http://www.cnblogs.com/dennyzhangdd/p/6734638.html)

对象头**lock\_bits**锁标志位枚举：

1. **enum** {   locked\_value             = 0,//00 轻量级锁
2. unlocked\_value           = 1,//01 无锁
3. monitor\_value            = 2,//10 监视器锁，也叫膨胀锁，也叫重量级锁
4. marked\_value             = 3,//11 GC标记
5. biased\_lock\_pattern      = 5 //101 偏向锁
6. };

markOopDesc引用ObjectMonitor

1. **class** markOopDesc: **public** oopDesc {
2. **class** ObjectMonitor;
4. bjectMonitor\* monitor() **const** {
5. assert(has\_monitor(), "check");
6. // Use xor instead of &~ to provide one extra tag-bit check.
7. **return** (ObjectMonitor\*) (value() ^ monitor\_value);
8. }
10. }

#### ObjectMonitor类

1. **private**:
2. **friend** **class** ObjectSynchronizer;
3. **friend** **class** ObjectWaiter;
4. **volatile** markOop   \_header;
5. **volatile** **intptr\_t**  \_waiters;
6. ObjectWaiter \* **volatile** \_EntryList ;
7. ObjectMonitor() {
8. \_header       = NULL;//markOop对象头
9. \_count        = 0;
10. \_waiters      = 0,//等待线程数
11. \_recursions   = 0;//重入次数
12. \_object       = NULL;//监视器锁寄生的对象。
13. \_owner        = NULL;//指向获得ObjectMonitor对象的线程或基础锁
14. \_WaitSet      = NULL;//处于wait状态的线程，会被加入到wait set；
15. \_WaitSetLock  = 0 ;
16. \_Responsible  = NULL ;
17. \_succ         = NULL ;
18. \_cxq          = NULL ;
19. FreeNext      = NULL ;
20. \_EntryList    = NULL ;//处于等待锁block状态的线程，会被加入到entry set；
21. \_SpinFreq     = 0 ;
22. \_SpinClock    = 0 ;
23. OwnerIsThread = 0 ;// \_owner is (Thread \*) vs SP/BasicLock
24. \_previous\_owner\_tid = 0;// 监视器前一个拥有者线程的ID
25. }

#### 加锁过程

 synchronized修饰的代码段，在JVM被编译为monitorenter、monitorexit指令来获取和释放互斥锁.。

#### monitorenter

解释器执行monitorenter时会进入到InterpreterRuntime.cpp的InterpreterRuntime::monitorenter函数

TODO

### 4.2 String\_intern

#### hashTable.hpp

\openjdk7\hotspot\src\share\hashTable.hpp

1. **class** BasicHashtableEntry : **public** CHeapObj {
2. unsigned **int**         \_hash;
3. BasicHashtableEntry\* \_next;
4. }
6. template <**class** T> **class** HashtableEntry : **public** BasicHashtableEntry {
7. T   \_literal;    // ref to item in table.
8. }
10. **class** BasicHashtable : **public** CHeapObj {
11. **private**:
12. HashtableBucket\*  \_buckets;
13. BasicHashtableEntry\* \_free\_list;
14. **int**               \_table\_size;
15. **public**:
16. **int** table\_size() { **return** \_table\_size; }
17. **void** add\_entry(**int** index, BasicHashtableEntry\* entry);
19. }
21. **class** Hashtable : **public** BasicHashtable {
22. HashtableEntry<T>\* new\_entry(unsigned **int** hashValue, T obj);
23. }

#### symbolTable.cpp

\openjdk7\hotspot\src\share\vm\classfile\symbolTable.cpp

1. **class** StringTable : **public** Hashtable<oop> ;
2. **class** SymbolTable : **public** Hashtable<Symbol\*> ;
4. oop StringTable::intern(Handle string\_or\_null, jchar\* name, **int** len, TRAPS) { //561
5. unsigned **int** hashValue = java\_lang\_String::hash\_string(name, len);
6. **int** index = the\_table()->hash\_to\_index(hashValue);
7. oop string = the\_table()->lookup(index, name, len, hashValue);
9. // Found
10. **if** (string != NULL) **return** string;
12. // Otherwise, add to symbol to table
13. //the\_table ==>StringTable\* StringTable::\_the\_table = NULL;
14. **return** the\_table()->basic\_add(index, string\_or\_null, name, len, hashValue, CHECK\_NULL);
16. }

19. oop StringTable::basic\_add(**int** index, Handle string\_or\_null, jchar\* name,**int** len, unsigned **int** hashValue, TRAPS) {  //516
21. oop test = lookup(index, name, len, hashValue); // calls lookup(u1\*, int)
22. **if** (test != NULL) {
23. // Entry already added
24. **return** test;
25. }
27. HashtableEntry<oop>\* entry = new\_entry(hashValue, string());
28. add\_entry(index, entry);
29. **return** string();
30. }
32. oop StringTable::lookup(**int** index, jchar\* name,**int** len, unsigned **int** hash) { //503
33. **for** (HashtableEntry<oop>\* l = bucket(index); l != NULL; l = l->next()) {
34. **if** (l->hash() == hash) {
35. **if** (java\_lang\_String::equals(l->literal(), name, len)) {
36. **return** l->literal();
37. }
38. }
39. }
40. **return** NULL;
41. }
43. template <**class** T> HashtableEntry<T>\* Hashtable<T>::new\_entry(unsigned **int** hashValue, T obj) {
44. HashtableEntry<T>\* entry;
46. entry = (HashtableEntry<T>\*)BasicHashtable::new\_entry(hashValue);
47. entry->set\_literal(obj);
48. }
50. inline **void** BasicHashtable::add\_entry(**int** index, BasicHashtableEntry\* entry) {
51. entry->set\_next(bucket(index));
52. \_buckets[index].set\_entry(entry);
53. ++\_number\_of\_entries;
54. }

\openjdk7\hotspot\src\sharevm\classFile\javaClasses.hpp

1. **class** java\_lang\_String : AllStatic {

4. }