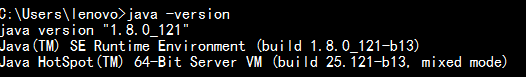
## Part 1 OpenJDK

### 1.1 Hotspot简介

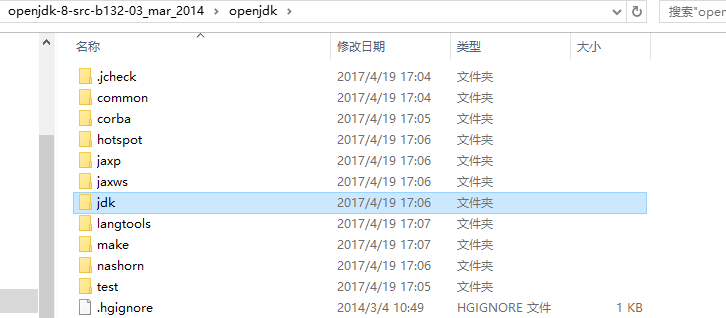
[Reference](https://www.cnblogs.com/dennyzhangdd/p/6734933.html)

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" \t "_blank" \o "操作系统知识库)**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。

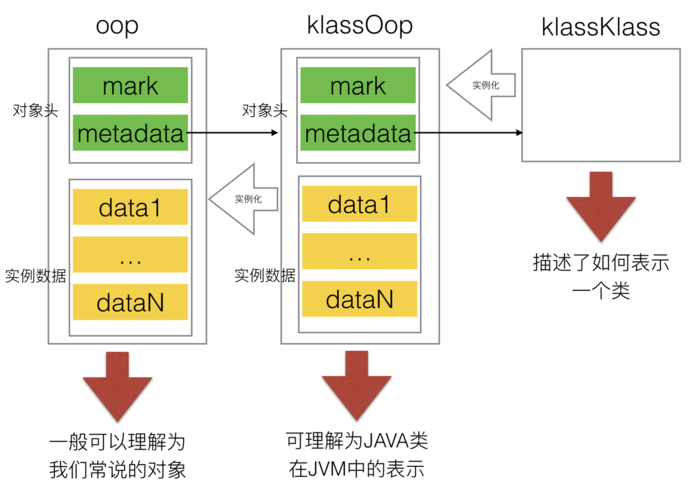
关于为何要设计oop/klass这种二分模型的实现，一个原因是不想让每个对象都包含vtbl(虚方法表)，其中oop中不含有任何虚函数，虚函数表保存于klass中，可以进行method dispatch。

### 2.1 关系图

instanceKlass VS klassOop

Klass是java类的元数据，描述java类结构。OOP是VM中数据结构。

instanceKlass在虚拟机层面描述一个Java类，instanceKlass在heap生成的实例klassOop相当于Java中的class实例。

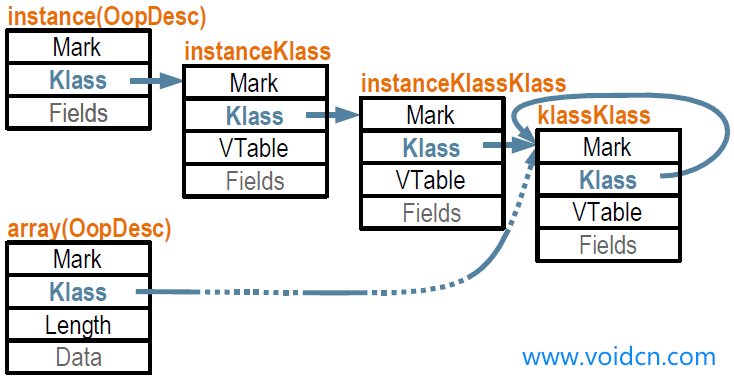


可见对象头klass指针指向的是对象。

klass链

在JVM中，对象在内存中的基本存在形式就是oop。**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：



在这种设计下，JVM对内存的分配和回收，都可以采用统一的方式来管理。

图中instance即为klassOop.

### 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**

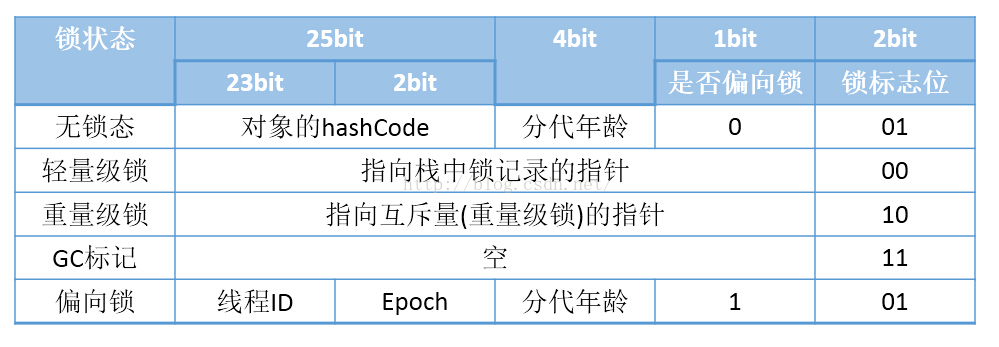
1. **class** oopDesc {
2. friend **class** VMStructs;
3. **private**:   //Header
4. **volatile** markOop  \_mark;
5. union \_metadata {
6. wideKlassOop    \_klass;
7. narrowOop       \_compressed\_klass;
8. } \_metadata;
10. **private**:
11. // field addresses in oop
12. **void**\*     field\_base(**int** 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. }

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

**A klassOop is the C++ equivalent of a Java class.**

##### ****layout****

// klassOop object layout:

// [header ]

// [klass\_field]

// [KLASS ]

##### klass\_part()

获取对应的Klass对象，并强制转换成instanceKlass类型的对象

1. **class** klassOopDesc : **public** oopDesc {
2. **public**:
3. // returns the Klass part containing dispatching behavior
4. Klass\* klass\_part() **const** {
5. **return** (Klass\*)((address)**this** + **sizeof**(klassOopDesc));
6. }
7. // Convenience wrapper
8. **inline** oop java\_mirror() **const**;
10. }

#### 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

java类的元数据

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;

##### layout

// Klass layout:

// [header ] klassOop

// [klass pointer ] klassOop

// [C++ vtbl ptr ] (contained in Klass\_vtbl)

// [layout\_helper ]

// [super\_check\_offset ] for fast subtype checks

// [secondary\_super\_cache] for fast subtype checks

// [secondary\_supers ] array of 2ndary supertypes

// [primary\_supers 0]

// [primary\_supers 1]

// [primary\_supers 2]

// ...

// [primary\_supers 7]

// [java\_mirror ]

// [super ]

// [name ]

// [first subklass]

// [next\_sibling ] link to chain additional subklasses

// [modifier\_flags]

// [access\_flags ]

// [verify\_count ] - not in product

// [alloc\_count ]

// [last\_biased\_lock\_bulk\_revocation\_time] (64 bits)

// [prototype\_header]

// [biased\_lock\_revocation\_count]

// [trace\_id]

##### Klass

1. **class** Klass : **public** Klass\_vtbl {
2. **protected**:
3. // Cache of last observed secondary supertype
4. klassOop    \_secondary\_super\_cache;
5. // Array of all secondary supertypes
6. objArrayOop \_secondary\_supers;
7. // Ordered list of all primary supertypes
8. klassOop    \_primary\_supers[\_primary\_super\_limit];
9. // java/lang/Class instance mirroring this class
10. oop       \_java\_mirror;
11. // Superclass
12. klassOop  \_super;
13. // First subclass (NULL if none); \_subklass->next\_sibling() is next one
14. klassOop \_subklass;
15. // Sibling link (or NULL); links all subklasses of a klass
16. klassOop \_next\_sibling;
17. AccessFlags \_access\_flags;
18. **public**:
19. **const** Klass\_vtbl& vtbl\_value() **const** { **return** \***this**; }
20. // java mirror
21. oop java\_mirror() **const**              { **return** \_java\_mirror; }
23. // Casting
24. **static** Klass\* cast(klassOop k) {
25. **return** k->klass\_part();
26. }
27. **public**:
28. // type testing operations
29. **virtual** **bool** oop\_is\_instance\_slow()       **const** { **return** **false**; }
30. **virtual** **bool** oop\_is\_instanceMirror()      **const** { **return** **false**; }
31. **virtual** **bool** oop\_is\_instanceRef()         **const** { **return** **false**; }
32. **virtual** **bool** oop\_is\_array()               **const** { **return** **false**; }
33. **virtual** **bool** oop\_is\_objArray\_slow()       **const** { **return** **false**; }
34. **virtual** **bool** oop\_is\_klass()               **const** { **return** **false**; }
35. **virtual** **bool** oop\_is\_thread()              **const** { **return** **false**; }
36. **virtual** **bool** oop\_is\_method()              **const** { **return** **false**; }
38. **bool** is\_public() **const**          { **return** \_access\_flags.is\_public(); }
39. **bool** is\_final() **const**           { **return** \_access\_flags.is\_final(); }
40. **bool** is\_interface() **const**         { **return** \_access\_flags.is\_interface(); }
41. **bool** is\_abstract() **const**        { **return** \_access\_flags.is\_abstract(); }
42. **bool** is\_super() **const**           { **return** \_access\_flags.is\_super(); }
43. markOop prototype\_header() **const**      { **return** \_prototype\_header; }
45. }

##### base\_create\_klass

\src\share\vm\oop\Klass.cpp

1. KlassHandle Klass::base\_create\_klass(KlassHandle& klass, **int** size,
2. **const** Klass\_vtbl& vtbl, TRAPS) {
3. klassOop ek = base\_create\_klass\_oop(klass, size, vtbl, THREAD);
4. **return** KlassHandle(THREAD, ek);
5. }

base\_create\_klass\_oop

1. klassOop Klass::base\_create\_klass\_oop(KlassHandle& klass, **int** size,
2. **const** Klass\_vtbl& vtbl, TRAPS) {
3. size = align\_object\_size(size);
4. // allocate and initialize vtable
5. Klass\*   kl = (Klass\*) vtbl.allocate\_permanent(klass, size, CHECK\_NULL);
6. klassOop k  = kl->as\_klassOop();
8. { // Preinitialize supertype information.
9. // A later call to initialize\_supers() may update these settings:
10. kl->set\_super(NULL);
11. **for** (juint i = 0; i < Klass::primary\_super\_limit(); i++) {
12. kl->\_primary\_supers[i] = NULL;
13. }
14. kl->set\_secondary\_supers(NULL);
15. oop\_store\_without\_check((oop\*) &kl->\_primary\_supers[0], k);
16. kl->set\_super\_check\_offset(in\_bytes(primary\_supers\_offset()));
17. }
19. kl->set\_java\_mirror(NULL);
20. kl->set\_modifier\_flags(0);
21. kl->set\_layout\_helper(Klass::\_lh\_neutral\_value);
22. kl->set\_name(NULL);
23. AccessFlags af;
24. af.set\_flags(0);
25. kl->set\_access\_flags(af);
26. kl->set\_subklass(NULL);
27. kl->set\_next\_sibling(NULL);
28. kl->set\_alloc\_count(0);
29. kl->set\_alloc\_size(0);
30. TRACE\_SET\_KLASS\_TRACE\_ID(kl, 0);
32. kl->set\_prototype\_header(markOopDesc::prototype());
33. kl->set\_biased\_lock\_revocation\_count(0);
34. kl->set\_last\_biased\_lock\_bulk\_revocation\_time(0);
36. **return** k;
37. }

allocate\_permanent

allocate\_permanent方法默认在PermGen分配内存，instanceKlass对象保存在永久代区域；

##### as\_klassOop

获取对应的klassOop

1. klassOop as\_klassOop() **const** {
2. // see klassOop.hpp for layout.
3. **return** (klassOop) (((**char**\*) **this**) - **sizeof**(klassOopDesc));
4. }

KlassOop.cpp

1. Klass\* klass\_part() **const** {
2. **return** (Klass\*)((address)**this** + **sizeof**(klassOopDesc));
3. }

##### Klass\_vtbl

1. **class** Klass\_vtbl {
2. **public**:
3. **virtual** **void**\* allocate\_permanent(KlassHandle& klass, **int** size, TRAPS) **const** = 0;
4. **void** post\_new\_init\_klass(KlassHandle& klass, klassOop obj, **int** size) **const**;
5. **protected**:
6. **void**\* operator **new**(**size\_t** ignored, KlassHandle& klass, **int** size, TRAPS);
7. }

#### instanceKlass

An instanceKlass is the VM level representation of a Java class.

It contains all information needed for at class at execution runtime.

##### fields

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

##### ClassState

1. **class** instanceKlass: **public** Klass {
2. **friend** **class** VMStructs;
3. **public**:
5. **enum** ClassState {
6. // object is not yet parsable by gc. Value of \_init\_state at object
7. unparsable\_by\_gc = 0,
8. allocation.
9. allocated,// allocated (but not yet linked)
10. loaded,   // loaded and inserted in class hierarchy (but not linked yet)    linked,      // successfully linked/verified (but not initialized yet)
11. being\_initialized, // currently running class initializer
12. fully\_initialized, // initialized (successfull final state)
13. initialization\_error // error happened during initialization
14. }
15. }

可见每个instanceKlass对象都有一个ClassState状态，用来标识当前class的加载进度

 Klass\_vtbl

1. **class** Klass\_vtbl {
2. **virtual** **void**\* allocate\_permanent(KlassHandle& klass, **int** size, TRAPS) **const** = 0;
3. **void** post\_new\_init\_klass(KlassHandle& klass, klassOop obj, **int** size) **const**;
5. // Every subclass on which vtbl\_value is called must include this macro.
6. // Delay the installation of the klassKlass pointer until after the
7. // the vtable for a new klass has been installed (after the call to new()).
8. #define DEFINE\_ALLOCATE\_PERMANENT(thisKlass) \
9. **void**\* allocate\_permanent(KlassHandle& klass\_klass, **int** size,TRAPS) **const**{\  **void**\* result = **new**(klass\_klass, size, THREAD) thisKlass();\
10. **if** (HAS\_PENDING\_EXCEPTION) **return** NULL;\
11. klassOop new\_klass = ((Klass\*) result)->as\_klassOop();\
12. OrderAccess::storestore();\
13. post\_new\_init\_klass(klass\_klass, new\_klass, size);\
14. **return** result;\
15. }
16. **bool** null\_vtbl() { **return** \*(**intptr\_t**\*)**this** == 0; }
18. **protected**:
19. **void**\* operator **new**(**size\_t** ignored, KlassHandle& klass, **int** size, TRAPS); };

#### instanceKlassKlass

##### create\_klass

1. klassOop instanceKlassKlass::create\_klass(TRAPS) {
2. instanceKlassKlass o;
3. KlassHandle h\_this\_klass(THREAD, Universe::klassKlassObj());
4. KlassHandle k = base\_create\_klass(h\_this\_klass, header\_size(), o.vtbl\_value(), CHECK\_NULL);
5. java\_lang\_Class::create\_mirror(k, CHECK\_NULL); // Allocate mirror
6. **return** k();
7. }

##### allocate\_instance\_klass

初始化一个空instanceKlass对象，并由后续逻辑进行数据的填充；

\src\share\vm\oop\instanceKlassKlass.cpp

1. klassOop instanceKlassKlass::allocate\_instance\_klass(){  Symbol\* name, int vtable\_len, int itable\_len, int static\_field\_size,
2. unsigned nonstatic\_oop\_map\_count,ReferenceType rt, TRAPS){
3. KlassHandle k;
4. instanceKlass o;
5. k = base\_create\_klass(h\_this\_klass, size, o.vtbl\_value(), CHECK\_NULL);
6. instanceKlass\* ik = (instanceKlass\*) k()->klass\_part();
7. //set default value
8. ik->set\_fields(NULL, 0);
9. ik->set\_constants(NULL);
10. ik->set\_class\_loader(NULL);
11. **return** k();
12. }

这里创建了instanceKlass,同时创建了klassOop。

Klass.cpp: const Klass\_vtbl& vtbl\_value() const { return \*this; }

#### klassKlass

1. **class** klassKlass: **public** Klass {
3. klassOop klassKlass::create\_klass(TRAPS) {
4. KlassHandle h\_this\_klass;
5. klassKlass o;
6. // for bootstrapping, handles may not be available yet.
7. klassOop k = base\_create\_klass\_oop(h\_this\_klass, header\_size(), o.vtbl\_value(), CHECK\_NULL);
8. k->set\_klass(k); // point to thyself
9. // Do not try to allocate mirror, java.lang.Class not loaded at this point. // See Universe::fixup\_mirrors()
10. **return** k;
11. }
12. }

klassOop  \_klassKlassObj = klassKlass::create\_klass(CHECK);

klassOop \_instanceKlassKlassObj= instanceKlassKlass::create\_klass(CHECK)

### KlassHandle

1. **class** Handle VALUE\_OBJ\_CLASS\_SPEC {
2. **private**:
3. oop\* \_handle;
5. **protected**:
6. oop     obj() **const**     { **return** \_handle == NULL ? (oop)NULL : \*\_handle; }  oop     non\_null\_obj() **const**       {  **return** \*\_handle; }
8. **public**:
9. // Constructors
10. Handle()                                       { \_handle = NULL; }
11. Handle(oop obj);
13. // General access
14. oop     operator () () **const**         { **return** obj(); }
15. oop     operator -> () **const**                   { **return** non\_null\_obj(); }
16. **bool**    operator == (oop o) **const**             { **return** obj() == o; }
17. **bool**    operator == (**const** Handle& h) **const**   { **return** obj() == h.obj(); }    // Null checks
18. **bool**    is\_null() **const**           { **return** \_handle == NULL; }
19. **bool**    not\_null() **const**          { **return** \_handle != NULL; }};
20. **class** KlassHandle: **public** Handle {
21. **protected**:
22. klassOop    obj() **const**            { **return** (klassOop)Handle::obj(); }
23. klassOop non\_null\_obj() **const** { **return** (klassOop)Handle::non\_null\_obj(); }  Klass\*      as\_klass() **const**    { **return** non\_null\_obj()->klass\_part(); }
24. }

### Universe

1. **class** Universe: AllStatic {
2. **private**:
3. // Known classes in the VM
4. **static** klassOop \_klassKlassObj;
5. **static** klassOop \_instanceKlassKlassObj;
6. **static** klassOop \_constantPoolKlassObj;
8. **void** Universe::genesis(TRAPS) {
9. \_klassKlassObj          = klassKlass::create\_klass(CHECK);
10. \_instanceKlassKlassObj  = instanceKlassKlass::create\_klass(CHECK);
11. }

## 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

\src\share\tools\launcher\java.c

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

\src\share\tools\launcher\java.c

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. }

GetStaticMethodID

**查找main方法**

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

NewPlatformStringArray

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native jvm.cpp中的jvm\_define\_class\_common

##### jvm\_define\_class\_common

\src\share\vm\prims\jvm.cpp

1. **static** jclass jvm\_define\_class\_common(JNIEnv \*env, **const** **char** \*name,
2. jobject loader, **const** jbyte \*buf,jsize len, jobject pd, **const** **char** \*source,jboolean verify, TRAPS) {
4. klassOop k = SystemDictionary::resolve\_from\_stream(class\_name,class\_loader , protection\_domain, &st,verify != 0,CHECK\_NULL);
5. return (jclass) JNIHandles::make\_local(env,Klass::cast(k)->java\_mirror());
6. }

resolve\_from\_stream处理stream数据流，并生成Klass对象。

##### resolve\_from\_stream

\src\share\vm\prims\classfile\SystemDictionary.cpp

1. klassOop SystemDictionary::resolve\_from\_stream(Symbol\* class\_name,
2. Handle class\_loader,
3. Handle protection\_domain,
4. ClassFileStream\* st,
5. **bool** verify,
6. TRAPS) {
8. instanceKlassHandle k = ClassFileParser(st).parseClassFile(class\_name,
9. class\_loader, protection\_domain, parsed\_name,verify,  THREAD);
10. define\_instance\_class(k, THREAD);
11. return k();
12. }

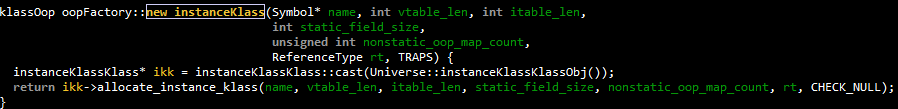
##### parseClassFile

\src\share\vm\prims\classfile\ClassFileParser.cpp

1. instanceKlassHandle ClassFileParser::parseClassFile(Symbol\* name, Handle class\_loader, Handle protection\_domain,KlassHandle host\_klass, GrowableArray<Handle>\* cp\_patches, TempNewSymbol& parsed\_name, **bool** verify, TRAPS) {
2. ClassFileStream\* cfs = stream();
3. // Magic value
4. u4 magic = cfs->get\_u4\_fast();
5. guarantee\_property(magic == JAVA\_CLASSFILE\_MAGIC,
6. "Incompatible magic value %u in class file %s",
7. magic, CHECK\_(nullHandle));
8. // Version numbers
9. u2 minor\_version = cfs->get\_u2\_fast();
10. u2 major\_version = cfs->get\_u2\_fast();
11. // ...
12. // Constant pool
13. constantPoolHandle cp = parse\_constant\_pool(CHECK\_(nullHandle));
14. ConstantPoolCleaner error\_handler(cp); // set constant pool to be cleaned up.
16. AccessFlags access\_flags;
17. jint flags = cfs->get\_u2\_fast() & JVM\_RECOGNIZED\_CLASS\_MODIFIERS;
18. access\_flags.set\_flags(flags);
20. // Interfaces
21. u2 itfs\_len = cfs->get\_u2\_fast();
22. //...
23. // Fields (offsets are filled in later)
24. typeArrayHandle fields = parse\_fields(class\_name, cp, access\_flags.is\_interface(), &fac, &fields\_annotations,                                            &java\_fields\_count,   CHECK\_(nullHandle));
25. // We can now create the basic klassOop for this klass
26. // ik’s type is instanceClass
27. klassOop ik = oopFactory::new\_instanceKlass(name, vtable\_size, itable\_si ze,  static\_field\_size, total\_oop\_map\_count, rt, CHECK\_(nullHandle));
28. this\_klass->set\_access\_flags(access\_flags);
29. this\_klass->set\_should\_verify\_class(verify);
30. jint lh = Klass::instance\_layout\_helper(instance\_size, **false**);
31. this\_klass->set\_layout\_helper(lh);
32. // Not yet: supers are done below to support the new subtype-checking fields
33. //this\_klass->set\_super(super\_klass());
34. this\_klass->set\_class\_loader(class\_loader());
35. this\_klass->set\_nonstatic\_field\_size(nonstatic\_field\_size);
36. this\_klass->set\_has\_nonstatic\_fields(has\_nonstatic\_fields);
37. this\_klass->set\_static\_oop\_field\_count(fac.count[STATIC\_OOP]);
38. cp->set\_pool\_holder(this\_klass());
39. error\_handler.set\_in\_error(**false**);   // turn off error handler for cp
40. this\_klass->set\_constants(cp());
41. this\_klass->set\_local\_interfaces(local\_interfaces());
42. this\_klass->set\_fields(fields(), java\_fields\_count);
43. this\_klass->set\_methods(methods());
44. this\_klass->set\_initial\_method\_idnum(methods->length());
45. this\_klass->set\_name(cp->klass\_name\_at(this\_class\_index));
46. this\_klass->set\_protection\_domain(protection\_domain());
47. this\_klass->set\_fields\_annotations(fields\_annotations());
48. this\_klass->set\_methods\_annotations(methods\_annotations());
49. this\_klass->set\_methods\_parameter\_annotations(methods\_parameter\_annotations());
50. this\_klass->set\_methods\_default\_annotations(methods\_default\_annotations());
52. this\_klass->set\_minor\_version(minor\_version);
53. this\_klass->set\_major\_version(major\_version);
55. // Create new handle outside HandleMark
56. instanceKlassHandle this\_klass (THREAD, preserve\_this\_klass);
57. debug\_only(this\_klass->as\_klassOop()->verify();)
59. **return** this\_klass;
61. }

##### new\_instanceKlass

\src\share\vm\memery\oopFactory.cpp



allocate\_instance\_klass不仅创建了KlassOoop还创建了instanceKlass

#### 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. }