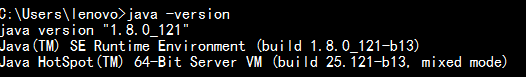
## Part 1 OpenJDK

### 1.1 Hotspot简介

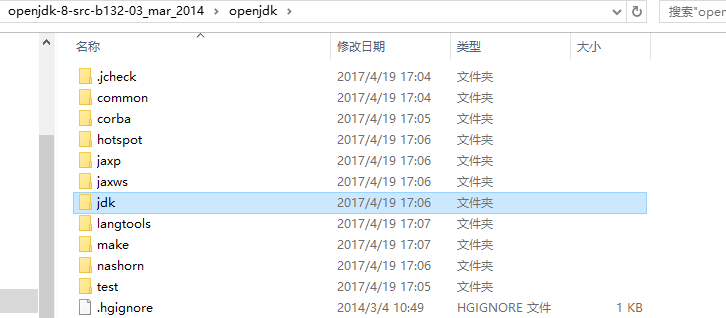
[Reference](https://www.cnblogs.com/dennyzhangdd/p/6734933.html)

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



### 1.3 解释器

[Reference](https://www.jianshu.com/p/22714cd32c90)

将字节码翻译成汇编代码

#### 模板解释器

TemplateInterpreter。hotspot默认使用的是模板解释器。

#### C++解释器

cppinterpreter，也叫做zero解释器。

UPDATE\_PC\_AND\_CONTINUE(1):指针类型PC+=1, 获取PC更新后对应代码,继续

SET\_STACK\_OBJECT/INT/FLOAT: 将Object/int/double入栈

SET\_LOCALS\_INT

初始化？

/src/share/vm/interpreter/cppinterpreter.cpp

1. **void** CppInterpreter::initialize() {
2. **if** (\_code != NULL) **return**;
3. AbstractInterpreter::initialize();
4. // generate interpreter
5. {
6. **int** code\_size = InterpreterCodeSize;
7. NOT\_PRODUCT(code\_size \*= 4;)  // debug uses extra interpreter code space
8. \_code = **new** StubQueue(**new** InterpreterCodeletInterface, code\_size, NULL,                            "Interpreter");
9. InterpreterGenerator g(\_code);
10. }
12. // Allow c++ interpreter to do one initialization now that switches are set, etc.
13. BytecodeInterpreter start\_msg(BytecodeInterpreter::initialize);
14. **if** (JvmtiExport::can\_post\_interpreter\_events())
15. BytecodeInterpreter::runWithChecks(&start\_msg);
16. **else**
17. BytecodeInterpreter::run(&start\_msg);
18. }

#### BytecodeInterpreter

### 1.4 JIT

### 1.5 JNI

[Reference](https://blog.csdn.net/kisimple/article/details/44204201)

实现原理

### 1.6 HotSpot SA

[Reference](https://www.jianshu.com/p/809bba0865a2)

## Part 2 oop-klass model

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

### 2.1 概述

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

原因是不想让每个对象都包含vtbl(虚方法表)，减少虚函数表的消耗，所以就拆成klass和oop，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。

为什么 C++ 中使用虚函数时会影响效率？

按照the C++PL的说法，虚函数调用比普通成员函数慢至多25%

oop是 Ordinary Object Pointer（普通对象指针），表示对象的实例信息， klass用来保存描述元数据。oopDesc和klass区别

#### 继承关系

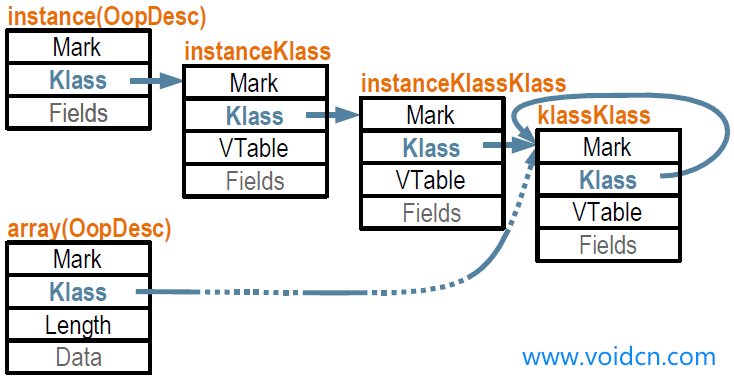
instanceKlassKlass : klassKlass: Klass

instanceKlass：Klass。

它们都可包装成KlassOop。

**klass链：**

图中Oop在heap，其他所有的Klass在permanent。



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。

#### 创建关系

1.Universe在permanent创建全局的KlassKlass和instanceKlassKlass。

2.instanceKlassKlass 在permanent创建instanceKlass(allocate\_instance\_klass)。加载时间：类加载

3.instanceKlass在heap中创建instanceOop(allocate\_instance)。加载时间：类加载

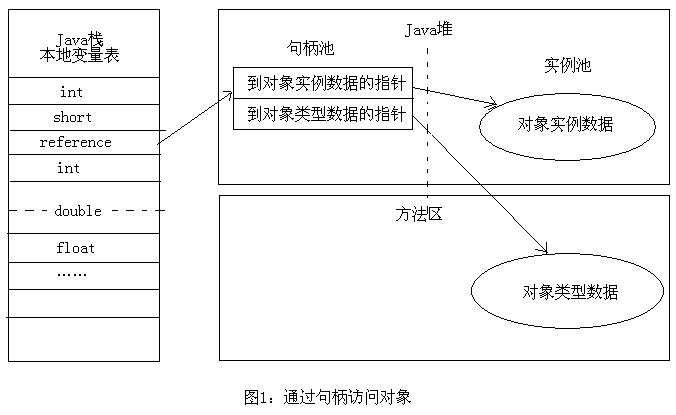
#### 对象结构

Java虚拟机规范不强制规定对象的内部结构应当如何表示 。

#### 访问方式

对象在不同内存中的有的访问方式，不同VM实现方式不同，主流两种：

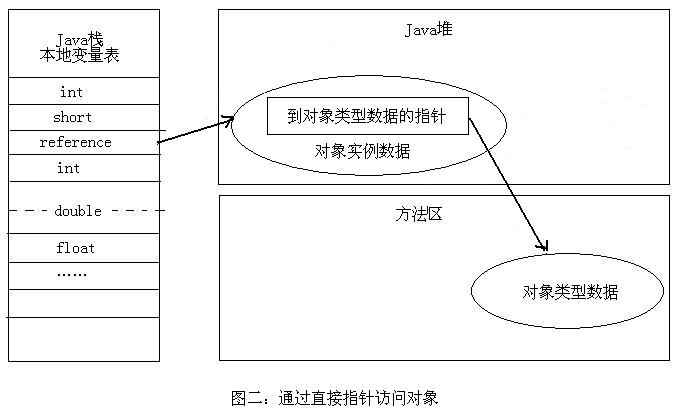
使用句柄



相比直接引用多一个指向对象实例的指针。

直接指针

hotspot采用这种方式



### 2.2 Oop结构

所有对象的基类。比如 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**;
13. jbyte\*    byte\_field\_addr(**int** offset)   **const**;
14. jchar\*    char\_field\_addr(**int** offset)   **const**;
15. ...
17. }

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

##### markOop

详细见part4的锁结构。

##### \_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

##### klassOop

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

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. }

### 2.3 klass结构

类的元数据

class klassOopDesc : public oopDesc

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. //表示methodOop的Klass
8. **class**   methodKlass;
9. //最为klass链的端点，klassKlass的Klass就是它自身
10. **class**   klassKlass;
11. //表示instanceKlass的Klass
12. **class**     instanceKlassKlass;
13. //表示arrayKlass的Klass
14. **class**     arrayKlassKlass;

##### 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]

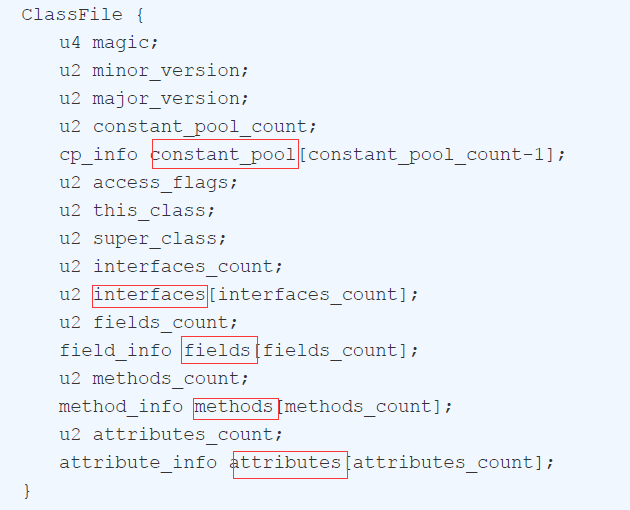
##### Klass

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. }

#### ClassFile结构

[Reference](http://blog.csdn.net/ns_code/article/details/17675609)1

[Reference2](https://blog.csdn.net/zhangjg_blog/article/details/22432599)



对于instanceClass.

* 包含了类的字段、方法、接口、版本等描述信息外，还有constant\_pool.

Note

u1，u2和u4，分别代表了1、2和4个字节的无符号数

##### as\_klassOop

获取对应的klassOop

Klass.cpp

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. }
8. }

#### instanceKlass

instanceKlassKlass::allocate\_instance\_klass()创建了instanceKlass

allocate\_instance 创建instanceOopDesc对象

new\_instanceKlass

##### 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); };

### 2.4 其他

#### 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. \_constantPoolKlassObj = constantPoolKlass::create\_klass(CHECK);
12. }

Universe生成全局的KlassKlassOop和instanceKlassKlass

#### handles.hpp

##### instanceKlassHandle

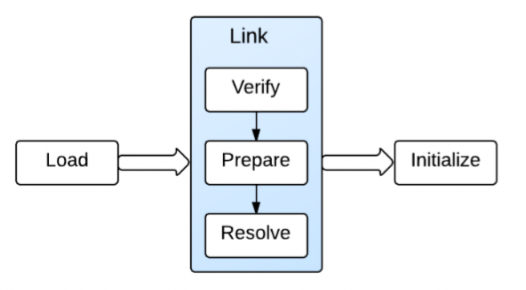
##### KlassHandle

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(); }

## Part 3 Klass生命周期

### 3.1 Summary

[Reference](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html)



**1) Load**

加载并解析字节码，最后调用instanceKlassKlass::allocate\_instance\_klass()在permanent中创建instanceKlass

**2) Linking**

**Verify**:

**Prepare**

给instanceKlass的类变量（不包括实例变量）分配内存并赋初始值。

**Resolution**

解析阶段是虚拟机将常量池中的符号引用转化为直接引用的过程。

**3) initialize**

instanceKlass::initialize()初始化。执行类构造器<clinit>()

类加载方式：

一是隐式加载：new/getstatic/putstatic/reflect/父类/main方法的类；

二是显式加载：在代码中调用loadClass()，Class.forName，ClassLoader的findClass方法等，显式加载中也可能包含隐式加载；

TODO：两种方式分别调用哪个API，进而指导jvm实现的逻辑。

#### 3.1.2 Java Classloader

**Bootstrap**:负责装载JRE的核心类库，如JRE目录下的rt.jar,charsets.jar等。C++实现。

**ExtClassLoader** 负责装载JRE扩展目录ext下的jar类包。getParent()为null。

**AppClassLoader**：装载应用程序的类

**三个loader是父子关系，不是继承关系？**

**API**

##### loadClass

findClass🡪 defineClass🡪 resolveClass, AppClassLoader加载时调用。

1. **protected** Class<?> loadClass(String name, **boolean** resolve){
2. **synchronized** (getClassLoadingLock(name)) {
3. // First, check if the class has already been loaded
4. Class c = findLoadedClass(name);
5. **if** (c == **null**) {
6. **if** (parent != **null**) {
7. c = parent.loadClass(name, **false**);
8. } **else** {
9. c = findBootstrapClassOrNull(name);
10. }
11. **if** (c == **null**) {
12. c = findClass(name);
13. }
14. }
15. **if** (resolve) {
16. resolveClass(c);
17. }
18. **return** c;
19. }
20. }

##### findClass

1. **protected** Class<?> findClass(String name) throws ClassNotFoundException {}

取得要加载class文件的字节流(未加载到vm)。

findClass底层实现

findLoadedClass底层实现

1. JVM\_ENTRY(jclass, JVM\_FindLoadedClass(JNIEnv \*env, jobject loader, jstring name))
2. TempNewSymbol klass\_name=SymbolTable::new\_symbol(str, str\_len, CHECK\_NULL); Handle h\_loader(THREAD, JNIHandles::resolve(loader));
3. klassOop k = SystemDictionary::find\_instance\_or\_array\_klass(klass\_name,
4. h\_loader,
5. Handle(),
6. CHECK\_NULL);
8. **return** (k == NULL) ? NULL :
9. (jclass) JNIHandles::make\_local(env, Klass::cast(k)->java\_mirror());
10. JVM\_END

find\_instance\_or\_array\_klass

##### defineClass

class文件字节流解析生成Class对象(该Class对象并未进行resolve)。此解析非类加载中那个解析。defineClass隶属Load阶段。

1. **private** native Class defineClass1(String name, byte[] b, **int** off, **int** len,
2. ProtectionDomain pd, String source);

实现位于ClassLoader.c文件中，

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

1. JVM\_ENTRY(jclass, JVM\_DefineClassWithSource(JNIEnv \*env, **const** **char** \*name, jobject loader, **const** jbyte \*buf, jsize len, jobject pd, **const** **char** \*source))
2. JVMWrapper2("JVM\_DefineClassWithSource %s", name);
4. **return** jvm\_define\_class\_common(env, name, loader, buf, len, pd, source, **true**, THREAD);
5. JVM\_END

最终调用native jvm.cpp中的jvm\_define\_class\_common

##### resolveClass

1. **private** native **void** resolveClass0(Class c);

链接指定的类。这个方法给Classloader用来链接一个类，如果这个类已经被链接过了，那么这个方法只做一个简单的返回。否则，这个类将被按照 Java™规范中的Execution描述进行链接。

1. JVM\_ENTRY(**void**, JVM\_ResolveClass(JNIEnv\* env, jclass cls))
2. JVMWrapper("JVM\_ResolveClass");
3. **if** (PrintJVMWarnings) warning("JVM\_ResolveClass not implemented");
4. JVM\_END

linking实现了link的三个子阶段。hotspot没提供该子阶段的单独实现？

Class.forName 和 ClassLoader.loadClass区别

forName: Load and Link

loadClass: just load,在newInstance执行static块。

### 3.3 Load

调用instanceKlassKlass::allocate\_instance\_klass()在permanent中创建instanceKlass

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

#### jvm\_define\_class\_common

defineClasss最终会调用jvm\_define\_class\_common

parseClassFile 🡪 allocate\_instance\_klass(instanceKlassKlass)

\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

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

解析byte流，并在heap中生成Oop。r esolve\_from\_stream处理stream数据流，并在permanent中生成Klass对象。

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

1. klassOop oopFactory::new\_instanceKlass(Symbol\* name, **int** vtable\_len, **int** itable\_len,**int** static\_field\_size,unsigned **int** nonstatic\_oop\_map\_count,                           ReferenceType rt, TRAPS) {
2. instanceKlassKlass\* ikk = instanceKlassKlass::cast(Universe::instanceKlassKlassObj());
3. **return** ikk->allocate\_instance\_klass(name, vtable\_len, itable\_len, static\_f ield\_size, nonstatic\_oop\_map\_count, rt, CHECK\_NULL);
4. }

allocate\_instance\_klass创建了instanceKlass(包装成KlassOop返回)。

TODO:解析器Interpreter

#### allocate\_instance\_klass

处于Linking的准备阶段

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。

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

#### 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. kl->set\_java\_mirror(NULL);
12. kl->set\_modifier\_flags(0);
13. kl->set\_layout\_helper(Klass::\_lh\_neutral\_value);
14. kl->set\_name(NULL);
15. AccessFlags af;
16. af.set\_flags(0);
17. kl->set\_access\_flags(af);
18. kl->set\_subklass(NULL);
19. kl->set\_next\_sibling(NULL);
20. kl->set\_alloc\_count(0);
21. kl->set\_alloc\_size(0);
22. TRACE\_SET\_KLASS\_TRACE\_ID(kl, 0);
24. kl->set\_prototype\_header(markOopDesc::prototype());
25. kl->set\_biased\_lock\_revocation\_count(0);
26. kl->set\_last\_biased\_lock\_bulk\_revocation\_time(0);
28. **return** k;
29. }

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；

#### create\_klass

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

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. }

### 3.4 link\_class

instanceKlass::link\_class\_impl。此方法实现linking三个子节点（hotspot没提供子阶段的单独实现？）

1. // link all interfaces implemented by this class before linking this class
2. objArrayHandle interfaces (THREAD, this\_oop->local\_interfaces());
3. **int** num\_interfaces = interfaces->length();
4. **for** (**int** index = 0; index < num\_interfaces; index++) {
5. HandleMark hm(THREAD);
6. instanceKlassHandle ih(THREAD, klassOop(interfaces->obj\_at(index)));
7. link\_class\_impl(ih, throw\_verifyerror, CHECK\_false);
8. }

link该类实现的 interfaces

#### verify

1. // verification & rewriting
2. {
3. ObjectLocker ol(this\_oop, THREAD);
4. // rewritten will have been set if loader constraint error found
5. // on an earlier link attempt
6. // don't verify or rewrite if already rewritten
7. **if** (!this\_oop->is\_linked()) {
8. **if** (!this\_oop->is\_rewritten()) {
9. {
10. **bool** verify\_ok = verify\_code(this\_oop, throw\_verifyerror, THREAD);
11. **if** (!verify\_ok) {
12. **return** **false**;
13. }
14. }

#### ResolveClass

解析阶段是虚拟机将常量池中的符号引用转化为直接引用的过程。

重写、重定位和方法链接。

重写是为支持更好的解释器运行性能，向常量池添加缓存，并调整相应字节码的常量池索引重新指向常量池 Cache 索引；重定位是 relocate\_and\_link\_methods()，其中包含方法链接，是为 Java 方法配置编译器或解释器入口。

1. //relocate jsrs and link methods after they are all rewritten
2. this\_oop->relocate\_and\_link\_methods(CHECK\_false);

**relocate\_and\_link\_methods**

1. // Now relocate and link method entry points after class is rewritten.
2. // This is outside is\_rewritten flag. In case of an exception, it can be
3. // executed more than once.
4. **void** instanceKlass::relocate\_and\_link\_methods(TRAPS) {
5. assert(is\_loaded(), "must be loaded");
6. instanceKlassHandle this\_oop(THREAD, **this**->as\_klassOop());
7. Rewriter::relocate\_and\_link(this\_oop, CHECK);
8. }

**initialize tables**

1. // Initialize the vtable and interface table after
2. // methods have been rewritten since rewrite may
3. // fabricate new methodOops.
4. // also does loader constraint checking
5. **if** (!this\_oop()->is\_shared()) {
6. ResourceMark rm(THREAD);
7. this\_oop->vtable()->initialize\_vtable(**true**, CHECK\_false);
8. this\_oop->itable()->initialize\_itable(**true**, CHECK\_false);
9. }

由于方法被重写后会产生新的 methodOops，初始化虚函数表 vtable 和接口表 itable。

1. this\_oop->set\_init\_state(linked);

设为linked状态

### 3.5 initialize

对load阶段生成的instanceKlass初始化。调用类构造器<clinit>()

initialize()

\src\share\vm\oop\instanceKlass.cpp

1. **void** instanceKlass::initialize\_impl(instanceKlassHandle this\_oop, TRAPS) {
2. // Make sure klass is linked (verified) before initialization
3. this\_oop->link\_class(CHECK);
4. // refer to the JVM book page 47 for description of steps
5. // Step 1
6. { ObjectLocker ol(this\_oop, THREAD);
7. ...
8. }

**Step 1** 过ObjectLocker在初始化前进行加锁，防止多个线程并发初始化。

1. { ObjectLocker ol(this\_oop, THREAD);
3. Thread \*self = THREAD; // it's passed the current thread

**Step 4**如果当前instanceKlass处于fully\_initialized状态，说明已经初始化完成，则直接返回

1. // Step 4
2. **if** (this\_oop->is\_initialized()) {
3. **return**;
4. }

**Step 5** 当前instanceKlass处于initialization\_error状态，说明初始化(clinit)失败了，抛出异常。

1. // Step 5
2. **if** (this\_oop->is\_in\_error\_state()) {
3. **const** **char**\* desc = "Could not initialize class ";
4. **const** **char**\* className = this\_oop->external\_name();
5. **size\_t** msglen = strlen(desc) + strlen(className) + 1;
6. **char**\* message = NEW\_RESOURCE\_ARRAY(**char**, msglen);
7. **if** (NULL == message) {
8. // Out of memory: can't create detailed error message
9. THROW\_MSG(vmSymbols::java\_lang\_NoClassDefFoundError(), className);
10. } **else** {
11. jio\_snprintf(message, msglen, "%s%s", desc, className);
12. THROW\_MSG(vmSymbols::java\_lang\_NoClassDefFoundError(), message);
13. }
14. }

**Step 6** 设置当前instanceKlass的状态为 **being\_initialized**；设置初始化线程为当前线程。

1. // Step 6
2. this\_oop->set\_init\_state(being\_initialized);
3. this\_oop->set\_init\_thread(self);

**Step 7** 如果当前instanceKlass不是接口类型，并且父类不为空，且还未初始化，则执行父类的初始化。

1. // Step 7
2. klassOop super\_klass = this\_oop->super();
3. **if** (super\_klass != NULL && !this\_oop->is\_interface() && Klass::cast(super\_klass)->should\_be\_initialized()) {
4. Klass::cast(super\_klass)->initialize(THREAD);
5. }
6. }

**Step 8** 通过call\_class\_initializer方法执行静态块代码

1. // Step 8
2. {
3. this\_oop->call\_class\_initializer(THREAD);
4. }
6. }

**call\_class\_initializer\_impl**

1. **void** instanceKlass::call\_class\_initializer\_impl(instanceKlassHandle this\_oop, TRAPS) {
2. methodHandle h\_method(THREAD, this\_oop->class\_initializer());
4. **if** (h\_method() != NULL) {
5. JavaCallArguments args; // No arguments
6. JavaValue result(T\_VOID);
7. JavaCalls::call(&result, h\_method, &args, CHECK); // Static call (no args)
8. }
9. }

this\_oop->class\_initializer()可以获取静态代码块入口，最终通过JavaCalls::call执行代码块逻辑，再下一层就是具体操作系统的实现了。

1. // Step 9
2. **if** (!HAS\_PENDING\_EXCEPTION) {
3. this\_oop->set\_initialization\_state\_and\_notify(fully\_initialized, CHECK);  }

如果初始化过程没有异常，说明instanceKlass对象已经初始完成，则设置当前instanceKlass的状态为 **fully\_initialized**，最后通知其它线程初始化已经完成；否则执行

S**tep10 and 11**

如果初始化发生异常，则设置当前instanceKlass的状态为 **initialization\_error**，并通知其它线程初始化发生异常。

1. **else** {
2. // Step 10 and 11
3. Handle e(THREAD, PENDING\_EXCEPTION);
4. CLEAR\_PENDING\_EXCEPTION;
5. {
6. EXCEPTION\_MARK;
7. this\_oop->set\_initialization\_state\_and\_notify(initialization\_error, THREAD);
8. CLEAR\_PENDING\_EXCEPTION;   // ignore any exception thrown, class initialization error is thrown below
9. }
10. DTRACE\_CLASSINIT\_PROBE\_WAIT(error, instanceKlass::cast(this\_oop()), -1,wait);
11. **if** (e->is\_a(SystemDictionary::Error\_klass())) {
12. THROW\_OOP(e());
13. } **else** {
14. JavaCallArguments args(e);
15. THROW\_ARG(vmSymbols::java\_lang\_ExceptionInInitializerError(),
16. vmSymbols::throwable\_void\_signature(),
17. &args);
18. }
19. }

### 3.6 实例化

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

类加载阶段instanceKlassKlass::allocate\_instance\_klass()创建了instanceKlass，在此阶段

instanceKlass::allocate\_instance 创建instanceOopDesc对象

#### 3.6.1 Oop内存分配

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

java对象的分配，分为快速分配（Fast allocation）和慢速分配（Slow allocation）两种形式。

##### Slow allocation

慢速分配根据堆的实现方式、GC的实现方式、代的实现方式不同而具有不同的分配调用层次。

\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);
7. // Make sure klass is initialized
8. klass->initialize(CHECK);
10. oop obj = klass->allocate\_instance(CHECK);
11. **thread**->set\_vm\_result(obj);
12. IRT\_END

pool->klass\_at返回BBB对应的klassOop对象

##### allocate\_instance

创建instanceOopDesc对象

\src\share\vm\oop\instanceKlass.cpp

1. instanceOop instanceKlass::allocate\_instance(TRAPS) {
2. **bool** has\_finalizer\_flag = has\_finalizer(); // Query before possible GC
3. **int** size = size\_helper();  // Query before forming handle.
5. KlassHandle h\_k(THREAD, as\_klassOop());
7. instanceOop i;
9. i = (instanceOop)CollectedHeap::obj\_allocate(h\_k, size, CHECK\_NULL);
10. **if** (has\_finalizer\_flag && !RegisterFinalizersAtInit) {
11. i = register\_finalizer(i, CHECK\_NULL);
12. }
13. **return** i;
14. }

has\_finalizer判断当前类是否包含不为空的finalize方法；

size\_helper确定创建当前对象需要分配多大内存；

**CollectedHeap::obj\_allocate**

从堆中申请指定大小的内存，并创建instanceOopDesc对象，实现如下：

1. oop CollectedHeap::obj\_allocate(KlassHandle klass, **int** size, TRAPS) {
2. HeapWord\* obj = common\_mem\_allocate\_init(size, CHECK\_NULL);
3. post\_allocation\_setup\_obj(klass, obj, size);
4. NOT\_PRODUCT(Universe::heap()->check\_for\_bad\_heap\_word\_value(obj, size));
5. **return** (oop)obj;
6. }

##### Fast allocation

[Reference](http://www.cnblogs.com/iceAeterNa/p/4877741.html)

已经解析。快速分配使用无锁的指针碰撞技术在新生代的Eden区上进行分配。

\src\share\vm\interpreter\bytecodeInterpreter.cpp

1. CASE(\_new): {
2. u2 index = Bytes::get\_Java\_u2(pc+1);
3. constantPoolOop constants = istate->method()->constants();
4. **if** (!constants->tag\_at(index).is\_unresolved\_klass()) {
5. // Make sure klass is initialized and doesn't have a finalizer
6. oop entry = constants->slot\_at(index).get\_oop();
7. klassOop k\_entry = (klassOop) entry;
8. assert(k\_entry->klass\_part()->oop\_is\_instance());
9. instanceKlass\* ik = (instanceKlass\*) k\_entry->klass\_part();

查询常量池获取klassOOp，确保oop\_is\_instance。resolved klass将通过fast allocation的方式创建Oop

1. **if** ( ik->is\_initialized() && ik->can\_be\_fastpath\_allocated() ) {
2. **size\_t** obj\_size = ik->size\_helper();
3. oop result = NULL;
4. // If the TLAB isn't pre-zeroed then we'll have to do it
5. **bool** need\_zero = !ZeroTLAB;
6. **if** (UseTLAB) {
7. result = (oop) THREAD->tlab().allocate(obj\_size);
8. }

使用TLAB技术给obj分配内存。TLAB是每个线程在Java堆中预先分配了一小块内存

1. **if** (result == NULL) {
2. need\_zero = **true**;
3. // Try allocate in shared eden
4. retry:
5. HeapWord\* compare\_to = \*Universe::heap()->top\_addr();
6. HeapWord\* new\_top = compare\_to + obj\_size;
7. **if** (new\_top <= \*Universe::heap()->end\_addr()) {
8. **if** (Atomic::cmpxchg\_ptr(new\_top, Universe::heap()->top\_addr(), compare\_to) != compare\_to) {
9. **goto** retry;
10. }
11. result = (oop) compare\_to;
12. }
13. }

**指针碰撞技术** allocate in shared eden

Universe::heap()->top\_addr() 返回的是Eden区空闲块的起始地址变量\_top的地址

1. //是否需要填0选项，对分配空间的对象数据区进行填0
2. **if** (result != NULL) {
3. // Initialize object (if nonzero size and need) and then the header
4. **if** (need\_zero ) {
5. HeapWord\* to\_zero = (HeapWord\*) result + **sizeof**(oopDesc) / oopSize;
6. obj\_size -= **sizeof**(oopDesc) / oopSize;
7. **if** (obj\_size > 0 ) {
8. memset(to\_zero, 0, obj\_size \* HeapWordSize);
9. }
10. }

1. **if** (UseBiasedLocking) {
2. result->set\_mark(ik->prototype\_header());
3. } **else** {
4. result->set\_mark(markOopDesc::prototype());
5. }
6. result->set\_klass\_gap(0);
7. result->set\_klass(k\_entry); //设置对象的klassOop引用
8. }

根据是否使用偏向锁，设置对象头信息

1. //把对象地址引入栈，并继续执行下一个字节码
2. SET\_STACK\_OBJECT(result, 0);
3. UPDATE\_PC\_AND\_TOS\_AND\_CONTINUE(3, 1);
4. //类型没有被解析，就会调用InterpreterRuntime的\_new函数完成慢速分配
5. CALL\_VM(InterpreterRuntime::\_new(THREAD, METHOD->constants(), index),
6. handle\_exception);
7. SET\_STACK\_OBJECT(THREAD->vm\_result(), 0);
8. THREAD->set\_vm\_result(NULL);
9. UPDATE\_PC\_AND\_TOS\_AND\_CONTINUE(3, 1);

### 3.3 GC垃圾收集器

jdk1.7 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

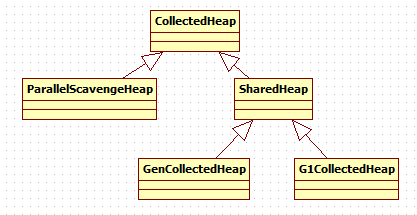
jdk1.9 默认垃圾收集器G1

-XX:+PrintCommandLineFlags -version查看默认设置收集器类型

-XX:+PrintGCDetails亦可通过打印的GC日志的新生代、老年代名称判断

JDK1.7 -XX:+UseParallelGC

1. jint Universe::initialize\_heap() {
2. **if** (UseParallelGC) {
3. Universe::\_collectedHeap = **new** ParallelScavengeHeap();
4. } **else** **if** (UseG1GC) {
5. G1CollectorPolicy\* g1p = **new** G1CollectorPolicy();
6. G1CollectedHeap\* g1h = **new** G1CollectedHeap(g1p);
7. Universe::\_collectedHeap = g1h;
8. } **else** {  to line23
9. GenCollectorPolicy \*gc\_policy;
10. **if** (UseSerialGC) {
11. gc\_policy = **new** MarkSweepPolicy();
12. } **else** **if** (UseConcMarkSweepGC) {
13. **if** (UseAdaptiveSizePolicy) {
14. gc\_policy = **new** ASConcurrentMarkSweepPolicy();
15. } **else** {
16. gc\_policy = **new** ConcurrentMarkSweepPolicy();
17. }
18. } **else** { // default old generation
19. gc\_policy = **new** MarkSweepPolicy();
20. }
22. Universe::\_collectedHeap = **new** GenCollectedHeap(gc\_policy);
23. }
25. jint status = Universe::heap()->initialize();
26. **if** (status != JNI\_OK) {
27. **return** status;
28. }
30. **if** (UseTLAB) {
31. ThreadLocalAllocBuffer::startup\_initialization();
32. }
33. **return** JNI\_OK;
34. }



## Part 4 实现细节

### 4.2 Load JVM

Launcher是一种用于启动JVM进程的启动器，Launcher负责维护HotSpot的整个生命周期，包括初始化环境以及销毁虚拟机等。

[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)

### 4.3 常量池

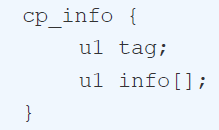
符号引用

[Reference](https://www.zhihu.com/question/30300585/answer/51335493)

转换过程

[Reference](https://www.zhihu.com/question/50258991)

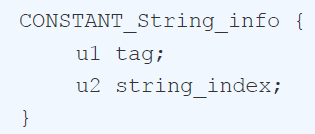
通用格式



tag项说明

|  |  |
| --- | --- |
| **常量类型** | **值** |
| CONSTANT\_Class | 7 |
| CONSTANT\_Fieldref | 9 |
| CONSTANT\_String | 8 |
| CONSTANT\_Integer | 3 |

**CONSTANT\_String\_info**



这里指Class文件中的常量池，用于存储编译期就生成的常量、符号引用。



**运行时的常量池**

存在于方法区，具有动态性。类加载的解析阶段会将符号引用转化为直接引用。

符号引用 VS 直接引用

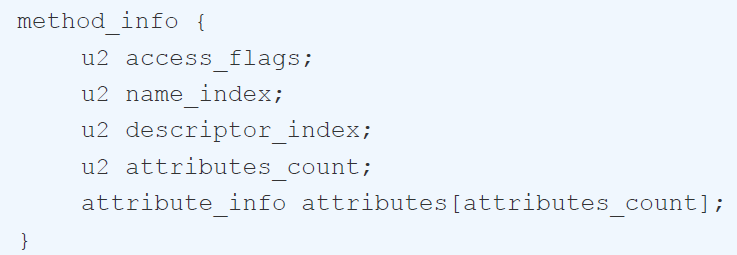
符号引用：编译后的class文件用字符串表示某个变量、接口的位置

直接引用：符号引用翻译出来的地址，**类加载阶段会将符号引用转化为直接引用。**

#### field\_info

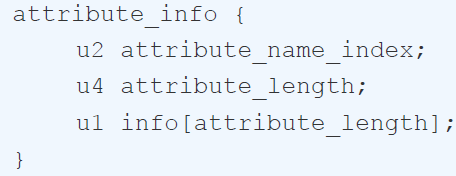
成员变量信息。field的操作会加入构造器中，不管引用类型还是基本类型,都用putfileld/getfield。

#### method\_info



规范定义的method\_info结构中，属性表可出现的成员有Code, Exception等

#### attribute\_info



### 4.4 锁

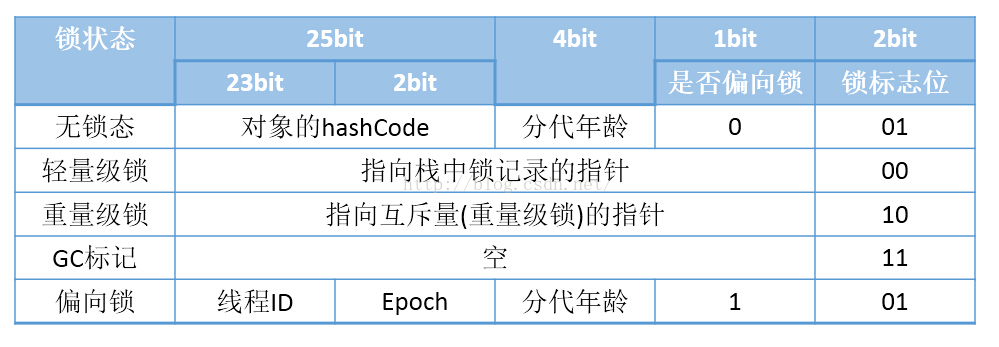
#### 4.4.1 锁结构

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

##### markOop

对象头用于存储对象自身的运行时数据。比如年代信息和锁。

32位的HotSpot Vm对象头存储结构：



实现源码

markOopDesc即对象头部的\_mark

1. **class** BasicLock;
2. **class** ObjectMonitor;
3. **class** JavaThread;
5. **class** markOopDesc: **public** oopDesc {
6. **private**:
7. // Conversion
8. **uintptr\_t** value() **const** { **return** (**uintptr\_t**) **this**; }
10. **public**:
11. // Constants 年代
12. **enum** {
13. age\_bits                 = 4,
14. lock\_bits                = 2,
15. biased\_lock\_bits         = 1,
16. max\_hash\_bits            = ,
17. hash\_bits                = max\_hash\_bits > 31 ? 31 : max\_hash\_bits,
18. cms\_bits                 = LP64\_ONLY(1) NOT\_LP64(0),
19. epoch\_bits               = 2
20. };
21. //锁
22. **enum** {
23. locked\_value             = 0,//00轻量级锁
24. unlocked\_value           = 1,//01无锁
25. monitor\_value            = 2,//10重量级锁
26. marked\_value             = 3,//11GC标记
27. biased\_lock\_pattern      = 5 //101偏向锁
28. };
30. ObjectMonitor\* monitor() **const** {
31. **return** (ObjectMonitor\*) (value() ^ monitor\_value);
32. }

35. }

可见每个对象头部都引用一个 monitor对象（所以Java中任意对象可以作为锁）。

##### ObjectMonitor

1. ObjectMonitor() {
2. \_header       = NULL;//markOop对象头
3. \_count        = 0;
4. \_waiters      = 0,//等待线程数
5. \_recursions   = 0;//重入次数
6. \_object       = NULL;//监视器锁寄生的对象。
7. \_owner        = NULL;//指向获得ObjectMonitor对象的线程或基础锁
8. \_WaitSet      = NULL;//处于wait状态的线程，会被加入到wait set；
9. \_EntryList    = NULL ;//处于等待锁block状态的线程，会被加入到entry set；
10. }

* \_owner指向持有ObjectMonitor对象的线程
* \_EntryList 等待锁block状态的线程
* \_WaitSet 调用wait()

#### 4.4.2 实现逻辑

加锁值存在对象头部，即mark中。cas操作实现轻量级锁,但仅用于交替执行同步块的情况。对于同一Thread可用偏向锁，减少cas操作。多个线程在同一时刻进入临界区膨胀为重量级锁。依赖OS层面实现。

[Reference](https://www.jianshu.com/p/c5058b6fe8e5)  
monitorenter

解释器执行monitorenter时会进入monitorenter函数。

/src/share/vm/interpreter/InterpreterRuntime.cpp

1. IRT\_ENTRY\_NO\_ASYNC(**void**, InterpreterRuntime::monitorenter(JavaThread\* **thread**, BasicObjectLock\* elem))
3. Handle h\_obj(**thread**, elem->obj());
4. **if** (UseBiasedLocking) {
5. // Retry fast entry if bias is revoked to avoid unnecessary inflation
6. ObjectSynchronizer::fast\_enter(h\_obj, elem->lock(), **true**, CHECK);
7. } **else** {
8. ObjectSynchronizer::slow\_enter(h\_obj, elem->lock(), CHECK);
9. }
10. IRT\_END

##### 偏向锁

引入偏向锁是为了减少不必要的轻量级锁执行，轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只依赖一次CAS原子指令置换ThreadID。

1. **void** ObjectSynchronizer::fast\_enter(Handle obj, BasicLock\* lock, **bool** attempt\_rebias, TRAPS) {
2. **if** (UseBiasedLocking) {
3. **if** (!SafepointSynchronize::is\_at\_safepoint()) {
4. BiasedLocking::Condition cond = BiasedLocking::revoke\_and\_rebias(obj, attempt\_rebias, THREAD);
5. **if** (cond == BiasedLocking::BIAS\_REVOKED\_AND\_REBIASED) {
6. **return**;
7. }
8. } **else**
9. BiasedLocking::revoke\_at\_safepoint(obj);//撤销偏向锁
10. }
11. assert(!obj->mark()->has\_bias\_pattern(), "biases should be revoked by now");
12. }
13. //升级到轻量级锁
14. slow\_enter (obj, lock, THREAD) ;
15. }

**revoke\_and\_rebias**

通过CAS原子指令设置mark中JavaThread为当前线程ID，下次执行时判断是同一Thread，直接return.

##### 轻量级锁

在多线程交替执行同步块的情况下，但是如果多个线程在同一时刻进入临界区，会导致轻量级锁膨胀升级重量级。

1. **void** ObjectSynchronizer::slow\_enter(Handle obj, BasicLock\* lock, TRAPS) {
2. markOop mark = obj->mark();

5. **if** (mark->is\_neutral()) { //当无锁状态01
6. // Anticipate successful CAS -- the ST of the displaced mark must
7. // be visible <= the ST performed by the CAS.
8. lock->set\_displaced\_header(mark);
9. **if**(mark==(markOop)Atomic::cmpxchg\_ptr(lock,obj()->mark\_addr(), mark)) {        TEVENT (slow\_enter: release stacklock) ;
10. **return** ;
11. }
12. // Fall through to inflate() ...
13. } **else** **if** (mark->has\_locker() && THREAD->is\_lock\_owned((address)mark->locker())) {
14. //已经加了轻量级锁
15. lock->set\_displaced\_header(NULL);
16. **return**;
17. }
18. //升级到重量级锁
19. lock->set\_displaced\_header(markOopDesc::unused\_mark());
20. ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
21. }

##### 重量级锁

重量级锁通过对象内部的监视器（monitor）实现，其中monitor的本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高

### 4.4 String\_intern

#### 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. }

**basic\_add**

1. oop StringTable::basic\_add(**int** index, Handle string\_or\_null, jchar\* name,**int** len, unsigned **int** hashValue, TRAPS) {  //516
3. oop test = lookup(index, name, len, hashValue); // calls lookup(u1\*, int)
4. **if** (test != NULL) {
5. // Entry already added
6. **return** test;
7. }
9. HashtableEntry<oop>\* entry = new\_entry(hashValue, string());
10. add\_entry(index, entry);
11. **return** string();
12. }

**new\_entry**

1. **template** <**class** T> HashtableEntry<T>\* Hashtable<T>::new\_entry(unsigned **int** hashValue, T obj) {
2. HashtableEntry<T>\* entry;
3. entry = (HashtableEntry<T>\*)BasicHashtable::new\_entry(hashValue);
4. entry->set\_literal(obj);
5. }

**add\_entry**

1. **inline** **void** BasicHashtable::add\_entry(**int** index, BasicHashtableEntry\* entry) {
2. entry->set\_next(bucket(index));
3. \_buckets[index].set\_entry(entry);
4. ++\_number\_of\_entries;
5. }

lookup

1. oop StringTable::lookup(**int** index, jchar\* name,**int** len, unsigned **int** hash) { //503
2. **for** (HashtableEntry<oop>\* l = bucket(index); l != NULL; l = l->next()) {
3. **if** (l->hash() == hash) {
4. **if** (java\_lang\_String::equals(l->literal(), name, len)) {
5. **return** l->literal();
6. }
7. }
8. }
9. **return** NULL;
10. }

#### 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. }

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

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

4. }