http://www.zhihu.com/question/21574535/answer/18998914
Java instanceof 关键字是如何实现的？

RednaxelaFX,从事JVM研发
戴翔、咚咔、evenX86 等人赞同
在进入正题前先得提一句:既然楼主提问的背景是“被面试问到”,那很重要的一点是揣摩面试官提问的意图. 按照楼主的简单描述,面试官问的是“Java的instanceof关键字是如何实现的”,那要选择怎样的答案就得看面试官有怎样的背景. 比较安全的应对方法是请求面试官澄清/细化他的问题——他想要的“底层”底到什么程度.

情形1:
你在面月薪3000以下的Java码农职位. 如果面试官也只是做做Java层开发的,他可能只是想让你回答Java语言层面的instanceof 运算符的语义. Java语言的“意思”就已经是“底层”. 这样的话只要参考Java语言规范对 instanceof 运算符的定义就好:
15.20.2 Type Comparison Operator instanceof, Java语言规范Java SE 7版
当然这实际上回答的不是“如何实现的”,而是“如何设计的”. 但面试嘛…

如果用Java的伪代码来表现Java语言规范所描述的运行时语义,会是这样:

```
// obj instanceof T
boolean result;
try {
  T temp = (T) obj; // checkcast
  result = true;
} catch (ClassCastException e) {
  result = false;
}
```

用中文说就是:如果有表达式 obj instanceof T ,那么如果 (T) obj 不抛 ClassCastException 异常则该表达式值为 true ,否则值为 false .
注意这里完全没提到JVM啊Class对象啊啥的. 另外要注意 instanceof 运算符除了运行时语义外还有部分编译时限制,详细参考规范.

如果这样回答被面试官说“这不是废话嘛”,请见情形2.

情形2:
你在面月薪6000-8000的Java研发职位. 面试官也知道JVM这么个大体概念,但知道的也不多. JVM这个概念本身就是“底层”. JVM有一条名为 instanceof 的指令,而Java源码编译到Class文件时会把Java语言中的 instanceof 运算符映射到JVM的instanceof 指令上.

你可以知道Java的源码编译器之一javac是这样做的:

1. instanceof 是javac能识别的一个关键字,对应到Token.INSTANCEOF的token类型. 做词法分析的时候扫描到"instanceof"关键字就映射到了一个Token.INSTANCEOF token. jdk7u/jdk7u/langtools: 5c9759e0d341 src/share/classes/com/sun/tools/javac/parser/Token.java
2. 该编译器的抽象语法树节点有一个JCTree.JCInstanceOf类用于表示instanceof运算. 做语法分析的时候解析到instanceof运算符就会生成这个JCTree.JCInstanceof类型的节点. jdk7u/jdk7u/langtools: 5c9759e0d341 src/share/classes/com/sun/tools/javac/parser/JavacParser.java term2Rest()
3. 中途还得根据Java语言规范对instanceof运算符的编译时检查的规定把有问题的情况找出来.
4. 到最后生成字节码的时候为JCTree.JCInstanceof节点生成instanceof字节码指令. jdk7u/jdk7u/langtools: 5c9759e0d341 src/share/classes/com/sun/tools/javac/jvm/Gen.javavisitTypeTest()

(Java语言君说:“instanceof 这问题直接交给JVM君啦”)
(面试官:你还给我废话…给我进情形3!)

其实能回答到这层面就已经能解决好些实际问题了,例如说需要手工通过字节码增强来实现一些功能的话,知道JVM有这么条instanceof 指令或许正好就能让你顺利的使用 ASM 之类的库完成工作.

情形3:
你在面月薪10000的Java高级研发职位. 面试官对JVM有一些了解,想让你说说JVM会如何实现 instanceof 指令. 但他可能也没看过实际的JVM是怎么做的,只是臆想过一下而已. JVM的规定就是“底层”. 这种情况就给他JVM规范对 instanceof 指令的定义就好:

Chapter 6. The Java Virtual Machine Instruction Set, JVM规范Java SE 7版
根据规范来臆想一下实现就能八九不离十的混过这题了.

该层面的答案就照@敖琪前面给出的就差不多了,这边不再重复.

情形4:
你可能在面真的简易JVM的研发职位,或许是啥嵌入式JVM的实现. 面试官会希望你对简易JVM的实现有所了解. JVM的直观实现就是"底层". 这个基本上跟情形3差不多,因为简易JVM通常会用很直观的方式去实现. 但对具体VM实现得答对一些小细节,例如说这个JVM是如何管理类型信息的.

这个情形的话下面举点例子来讲讲.

情形5:
你在面试月薪10000以上的Java资深研发职位,注重性能调优啥的. 这种职位虽然不直接涉及JVM的研发,但由于性能问题经常源自"抽象泄漏",对实际使用的JVM的实现的思路需要有所了解. 面试官对JVM的了解可能也就在此程度. 对付这个可以用一篇论文:Fast subtype checking in the HotSpot JVM. 之前有个讨论帖里讨论过对这篇论文的解读:请教一个share/vm/oops下的代码做fast subtype check的问题

情形6:
你在面试真的高性能JVM的研发职位,例如 HotSpot VM 的研发. JVM在实际桌面或服务器环境中的具体实现是"底层". 呵呵这要回答起来就复杂了,必须回答出JVM实现中可能做的优化具体的实现. 另外找地方详细写.

我觉得会问这种问题的还是情形1和2的比例比较大,换句话说面试官也不知道真的JVM是如何实现这instanceof指令的,可能甚至连这指令的准确语义都无法描述对. 那随便忽悠忽悠就好啦不用太认真. 说不定他期待的答案本身就雾很大(逃
碰上情形4、6的话,没有忽悠的余地,是怎样就得怎样.
情形5可能还稍微有点忽悠余地呃呵呵.
==========
看俩实际存在的简易JVM的实现,Kaffe和JamVM. 它们都以解释器为主,JIT的实现非常简单,主要功能还是在VM runtime里实现,所以方便考察.
主要考察的是:它们中Java对象的基本结构(如何找到类型信息),类型信息自身如何记录(内部用的类型信息与Java层的java.lang.Class对象的关系),以及instanceof具体是怎样实现的.

Kaffe
https://github.com/kaffe/kaffe/blob/master/include/native.h
Kaffe中Java对象由Hjava_lang_Object结构体表示,里面有个struct _dispatchTable*类型的字段vtable,下面再说.

https://github.com/kaffe/kaffe/blob/master/kaffe/kaffevm/classMethod.h
Java层的java.lang.Class实例在VM里由Hjava_lang_Class结构体表示. Kaffe直接使用Hjava_lang_Class来记录VM内部的类型信息. 也就是说在Kaffe上运行的Java程序里持有的java.lang.Class的实例就是该JVM内部存类型信息的对象.
前面提到的_dispatchTable结构体也在该文件里定义. 它是一个虚方法分派表,主要用于高效实现invokevirtual.
假如有Hjava_lang_Object* obj,要找到它对应的类型信息只要这样:

```
obj->vtable->class
```

https://github.com/kaffe/kaffe/blob/master/kaffe/kaffevm/soft.c
instanceof的功能由soft.c的soft_instanceof()函数实现. 该函数所调用的函数大部分都在这个文件里.
https://github.com/kaffe/kaffe/blob/master/kaffe/kaffevm/intrp/icode.h#L295
这边定义了softcall_instanceof宏用于在解释器或者JIT编译后的代码里调用soft_instanceof()函数
https://github.com/kaffe/kaffe/blob/master/kaffe/kaffevm/kaffe.def#L3299
这边定义了instanceof字节码指令的处理要调用softcall_instanceof宏

JamVM
http://jamvm.cvs.sourceforge.net/viewvc/jamvm/jamvm/src/jam.h?view=markup
JamVM中Java对象由Object结构体表示,Java层的java.lang.Class实例在VM里由Class表示(是个空Object),VM内部记录的类信息由ClassBlock结构体表示(类型名、成员、父类、实现的接口、类价值器啥的都记录在ClassBlock里). 比较特别的是每个Class与对应的ClassBlock实际上是粘在一起分配的,所以Class*与ClassBlock*可以很直接的相互转换. 例如说如果有Class* c想拿到它对应的ClassBlock,只要:

```
ClassBlock* cb = CLASS_CB(c);
```

即可.

Object结构体里有Class*类型的成员class,用于记录对象的类型.

http://jamvm.cvs.sourceforge.net/viewvc/jamvm/jamvm/src/cast.c?view=markup
instanceof的功能由cast.c第68行的isInstanceOf()函数实现. 该函数所调用的函数大部分都在这个文件里.
http://jamvm.cvs.sourceforge.net/viewvc/jamvm/jamvm/src/interp.c?view=markup
解释器主循环的代码主要在interp.c里. 把instanceof指令的参数所指定的常量池索引解析为实际类指针的逻辑在OPC_INSTANCEOF的实现里. JamVM做了个优化,在解析好类之后会把instanceof字节码改写为内部字节码instanceof_quick;调用isInstanceOf()的地方在2161行OPC_INSTANCEOF_QUICK的实现里,可以看到它调用的是isInstanceOf(class, obj->class).

上面介绍了Kaffe与JamVM里instanceof字节码的实现相关的代码在哪里. 接下来简单分析下它们的实现策略.

两者的实现策略其实几乎一样,基本上按照下面的步骤:
(假设要检查的对象引用是obj,目标的类型对象是T)

1. obj如果为null,则返回false;否则设S为obj的类型对象,剩下的问题就是检查S是否为T的子类型
2. 如果S == T,则返回true;
3. 接下来分为3种情况,S是数组类型、接口类型或者类类型. 之所以要分情况是因为instanceof要做的是“子类型检查”,而Java语言的类型系统里数组类型、接口类型与普通类类型三者的子类型规定都不一样,必须分开来讨论. 到这里虽然例中两个JVM的具体实现有点区别,但概念上都与JVM规范所描述的 instanceof的基本算法 几乎一样. 其中一个细节是:对接口类型的instanceof就直接遍历S里记录的它所实现的接口,看有没有跟T一致的;而对类类型的instanceof则是遍历S的super链(继承链)一直到Object,看有没有跟T一致的. 遍历类的super链意味着这个算法的性能会受类的继承深度的影响.

关于Java语言里子类型关系的定义,请参考:Chapter 4. Types, Values, and Variables
类类型和接口类型的子类型关系大家可能比较熟悉,而数组类型的子类型关系可能会让大家有点意外.

4.10.3. Subtyping among Array Types

The following rules define the direct supertype relation among array types:

- If S and T are both reference types, then S[] >1 T[] iff S >1 T.
- Object >1 Object[]
- Cloneable >1 Object[]
- java.io.Serializable >1 Object[]
- If P is a primitive type, then:
    - Object >1 P[]
    - Cloneable >1 P[]
    - java.io.Serializable >1 P[]

这里稍微举几个例子. 以下子类型关系都成立(“<:”符号表示左边是右边的子类型,“=>”符号表示“推导出”):

1. String[][][] <: String[][][] (数组子类型关系的自反性)
2. String <: CharSequence => String[] <: CharSequence[] (数组的协变)
3. String[][][] <: Object (所有数组类型是Object的子类型)
4. int[] <: Serializable (原始类型数组实现java.io.Serializable接口)
5. Object[] <: Serializable (引用类型数组实现java.io.Serializable接口)
6. int[][][] <: Serializable[][] <: Serializable[] <: Serializable (上面几个例子的延伸…开始好玩了吧?)
7. int[][][] <: Object[][] <: Object[] <: Object

好玩不?实际JVM在记录类型信息的时候必须想办法把这些相关类型都串起来以便查找.

另外补充一点:楼主可能会觉得很困惑为啥说到这里只字未提ClassLoader——因为在这个问题里还轮不到它出场.
在一个JVM实例里,"(类型的全限定名, defining class loader)"这个二元组才可以唯一确定一个类. 如果有两个类全限定名相同,也加载自同一个Class文件,但defining class loader不同,从VM的角度看它们就是俩不同的类,而且相互没有子类型关系. instanceof运算符只关心“是否满足子类型关系”,至于类型名是否相同之类的不需要关心.

通过Kaffe与JamVM两个例子我们可以看到简单的JVM实现很多地方就是把JVM规范直观的实现了出来. 这就解决了前面提到的情形4的需求.

==========
至于情形5、6,细节讲解起来稍麻烦所以这里不想展开写. 高性能的JVM跟简易JVM在细节上完全不是一回事.

简单来说,优化的主要思路就是把Java语言的特点考虑进来:由于Java的类所继承的超类与所实现的接口都不会在运行时改变,整个继承结构是稳定的,某个类型C在继承结构里的“深度”是固定不变的. 也就是说从某个类出发遍历它的super链,总是会遍历到不变的内容.
这样我们就可以把原本要循环遍历super链才可以找到的信息缓存在数组里,并且以特定的下标从这个数组找到我们要的信息. 同时,Java的类继承深度通常不会很深,所以为这个缓存数组选定一个固定的长度就足以优化大部分需要做子类型判断的情况.

HotSpot VM具体使用了长度为8的缓存数组,记录某个类从继承深度0到7的超类. HotSpot把类继承深度在7以内的超类叫做“主要超类型”(primary super),把所有其它超类型(接口、数组相关以及超过深度7的超类)叫做“次要超类型”(secondary super).
对“主要超类型”的子类型判断不需要像Kaffe或JamVM那样沿着super链做遍历,而是直接就能判断子类型关系是否成立. 这样,类的继承深度对HotSpot VM做子类型判断的性能影响就变得很小了.
对“次要超类型”,则是让每个类型把自己的“次要超类型”混在一起记录在一个数组里,要检查的时候就线性遍历这个数组. 留意到这里把接口类型、数组类型之类的子类型关系都直接记录在同一个数组里了,只要在最初初始化secondary_supers数组时就分情况填好了,而不用像Kaffe、JamVM那样每次做instanceof运算时都分开处理这些情况.

举例来说,如果有下述类继承关系:
Apple <: Fruit <: Plant <: Object
并且以Object为继承深度0,那么对于Apple类来说,它的主要超类型就有:
0: Object
1: Plant
2: Fruit
3: Apple
这个信息就直接记录在Apple类的primary_supers数组里了. Fruit、Plant等类同理.

如果我们有这样的代码:

```
Object f = new Apple();
boolean result = f instanceof Plant;
```

也就是变量f实际指向一个Apple实例,而我们要问这个对象是否是Plant的实例.
可以知道f的实际类型是Apple;要测试的Plant类的继承深度是1,拿Apple类里继承深度为1的主要超类型来看是Plant,马上就能得出结论是true.
这样就不需要顺着Apple的继承链遍历过去一个个去看是否跟Plant相等了.

对此感兴趣的同学请参考前面在情形5提到的两个链接. 先读第一个链接那篇论文,然后看第二个链接里的讨论(没有ACM帐号无法从第一个链接下载到论文的同学可以在第二个链接里找到一个镜像).

JDK6至今的HotSpot VM实际采用的算法是:

```
S.is_subtype_of(T) := {
  int off = T.offset;
  if (S == T) return true;
  if (T == S[off]) return true;
  if (off != &cache) return false;
  if ( S.scan_secondary_subtype_array(T) ) {
    S.cache = T;
    return true;
  }
  return false;
}
```

(具体是什么意思请务必参考论文)

这边想特别强调的一点是:那篇论文描述了HotSpot VM做子类型判断的算法,但其实只有HotSpot VM的解释器以及java.lang.Class.isInstance() 的调用是真的完整按照那个算法来执行的. HotSpot VM的两个编译器,Client Compiler (C1) 与Server Compiler (C2) 各自对子类型判断的实现有更进一步的优化. 实际上在这个JVM里,instanceof的功能就实现了4份,VM runtime、解释器、C1、C2各一份.

VM runtime的:
jdk7u/jdk7u/hotspot: e087a2088970 src/share/vm/oops/oop.inline.hpp oopDesc::is_a()
jdk7u/jdk7u/hotspot: e087a2088970 src/share/vm/oops/klass.hpp is_subtype_of()
jdk7u/jdk7u/hotspot: e087a2088970 src/share/vm/oops/klass.cppKlass::search_secondary_supers()

```
inline bool oopDesc::is_a(klassOop k)        const { return blueprint()->is_subtype_of(k); }
```

```
 bool is_subtype_of(klassOop k) const {
   juint    off = k->klass_part()->super_check_offset();
   klassOop sup = *(klassOop*)( (address)as_klassOop() + off );
   const juint secondary_offset = in_bytes(secondary_super_cache_offset());
   if (sup == k) {
     return true;
   } else if (off != secondary_offset) {
     return false;
   } else {
     return search_secondary_supers(k);
   }
 }
 bool search_secondary_supers(klassOop k) const;
```

```
bool Klass::search_secondary_supers(klassOop k) const {
 // Put some extra logic here out-of-line, before the search proper.
 // This cuts down the size of the inline method.

 // This is necessary, since I am never in my own secondary_super list.
 if (this->as_klassOop() == k)
   return true;
 // Scan the array-of-objects for a match
 int cnt = secondary_supers()->length();
 for (int i = 0; i < cnt; i++) {
   if (secondary_supers()->obj_at(i) == k) {
     ((Klass*)this)->set_secondary_super_cache(k);
     return true;
   }
 }
 return false;
}
```

解释器的(以x86-64的template interpreter为例):
jdk7u/jdk7u/hotspot: e087a2088970 src/cpu/x86/vm/templateTable_x86_64.cpp TemplateTable::instanceof()
jdk7u/jdk7u/hotspot: e087a2088970 src/cpu/x86/vm/interp_masm_x86_64.cpp InterpreterMacroAssembler::gen_subtype_check()
(太长,不把代码贴出来了. 要看代码请点上面链接)

C1和C2对instanceof的优化分散在好几个地方,以C1为例,
C1把Java字节码parse成HIR(High-level Intermediate Representation)的逻辑在
GraphBuilder::iterate_bytecodes_for_block(),它先把instanceof字节码解析成了InstanceOf节点:
jdk7u/jdk7u/hotspot: e087a2088970 src/share/vm/c1/c1_GraphBuilder.cpp
然后在优化过程中,InstanceOf节点会观察它的对象参数是否为常量null或者是固定的已知类型,并相应的尝试做常量折叠:
jdk7u/jdk7u/hotspot: e087a2088970 src/share/vm/c1/c1_Canonicalizer.cpp Canonicalizer::do_InstanceOf
如果已经常量折叠了的话就没后续步骤了. 反之则继续下去生成LIR:
jdk7u/jdk7u/hotspot: e087a2088970 src/cpu/x86/vm/c1_LIRGenerator_x86.cpp LIRGenerator::do_InstanceOf
jdk7u/jdk7u/hotspot: e087a2088970 src/share/vm/c1/c1_LIR.cpp LIR_List::instanceof
最后生成机器码:
jdk7u/jdk7u/hotspot: e087a2088970 src/cpu/x86/vm/c1_LIRAssembler_x86.cpp LIR_Assembler::emit_opTypeCheck
jdk7u/jdk7u/hotspot: e087a2088970 src/cpu/x86/vm/c1_LIRAssembler_x86.cpp
LIR_Assembler::emit_typecheck_helper
生成的机器码逻辑跟解释器版本基本上是一样的,只是寄存器使用上稍微不同.

而在C2中,

最初处理instanceof字节码生成C2的内部节点的逻辑主要在:
jdk7u/jdk7u/hotspot: e087a2088970 src/share/vm/opto/graphKit.cpp GraphKit::gen_instanceof()
它会调用 GraphKit::gen_subtype_check() 来生成检查逻辑的主体,而后者会根据代码的上下文所能推导出来的类型信息把类型检查尽量优化到更简单的形式,甚至直接就得出结论.

对这部分细节感兴趣的同学请单独联系我或者另外开帖讨论吧.

下面两个patch是我对HotSpot VM在子类型检查相关方面做的小优化,两个都在JDK7u40/JDK8里发布:

[#JDK-7170463] C2 should recognize "obj.getClass() == A.class" code pattern
Request for review (S): C2 should recognize "obj.getClass() == A.class" code pattern
http://hg.openjdk.java.net/hsx/hotspot-comp/hotspot/rev/8f6ce6f1049b

[#JDK-7171890] C1: add Class.isInstance intrinsic
Request for review (M): 7171890: C1: add Class.isInstance intrinsic
hsx/hotspot-comp/hotspot: 8f37087fc13f
lambda/lambda/hotspot: e1635876b206

举个例子,经过JDK-7170463的patch之后,HotSpot VM的C2会把下面这样的代码:

```
if (obj.getClass() == A.class) {
  boolean isInst = obj instanceof A;
}
```

优化为:

```
if (obj.getClass() == A.class) {
  boolean isInst = true;
}
```

那个instanceof运算就直接被常量折叠掉了. 楼主可以看看,当时面试你的面试官是否了解到这种细节了,而他又是否真的要在面试种考察这种细节.

==========

楼主的问题原本有提到 BytecodeInstanceOf.java . 它是 Serviceability Agent 的一部分,不是 HotSpot VM 内的逻辑. 关于 Serviceability Agent 请从这帖里的链接找资料来读读:记GreenTeaJUG第二次线下活动(杭州)

SA是HotSpot VM自带的一个用来调试、诊断HotSpot VM运行状态的工具. 它是一个“进程外”条调试器,也就是说假如我们要调试的HotSpot VM运行在进程A里,那么SA要运行在另一个进程B里去调试进程A. 这样做的好处是SA与被调试进程不会相互干扰,于是调试就可以更干净的进行;就算SA自己崩溃了也(通常)不会连带把被调试进程也弄崩溃.

SA在HotSpot VM内部的C++代码里嵌有一小块,主要是把HotSpot的C++类的符号信息记录下来;SA的主体则是用Java来实现的,把可调试的HotSpot里C++的类用Java再做一层皮. 楼主看到的BytecodeInstanceOf类就是这样的一层皮,它并不包含HotSpot的执行逻辑,纯粹是为调试用的.

直接放些外部参考资料链接方便大家找:
The HotSpot™ Serviceability Agent: An Out-of-Process High-Level Debugger for a Java™ Virtual Machine, USENIX JVM '01 这篇是描述 HotSpot Serviceability Agent 的原始论文,要了解 SA 的背景必读.

HotSpot source: Serviceability Agent (A. Sundararajan's Weblog)
提到了hotspot/agent目录里的代码都是 Serviceability Agent 的实现. 注意 SA 并不是 HotSpot VM 运行时必要的组成部分.

Serviceability in HotSpot, OpenJDK
这是OpenJDK馆网上的相关文档页面.