jdk源码解析-基础类-Object类源码解析

讲师简介

smart哥，互联网悍将，历经从传统软件公司到大型互联网公司的洗礼，入行即在中兴通讯等大型通信公司担任项目leader，后随着互联网的崛起，先后在美团支付等大型互联网公司担任架构师，公派旅美期间曾与并发包大神Doug Lea探讨java多线程等最底层的核心技术。对互联网架构底层技术有相当的研究和独特的见解，在多个领域有着丰富的实战经验。

一、源码解析

Java的Object是所有其他类的父类，从继承的层次来看它就是最顶层根，所以它也是唯一一个没有父类的类。它包含了对象常用的一些方法，比如`getClass`、`hashCode`、`equals`、`clone`、`toString`、`notify`、`wait`等常用方法。所以其他类继承了Object后就可以不用重复实现这些方法。这些方法大多数是native方法，下面具体分析。

主要的代码如下：

```
public class Object {

  private static native void registerNatives();
  static {
        registerNatives();
  }

  public Object() {}

  public final native Class<?> getClass();

  public native int hashCode();

  public boolean equals(Object obj) {
    return (this == obj);
  }

  protected native Object clone() throws CloneNotSupportedException;

  public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
  }

  public final native void notify();

  public final native void notifyAll();

  public final native void wait(long timeout) throws InterruptedException;

  public final void wait(long timeout, int nanos) throws InterruptedException {
    if (timeout < 0) {
      throw new IllegalArgumentException("timeout value is negative");
    }

    if (nanos < 0 || nanos > 999999) {
      throw new IllegalArgumentException("nanosecond timeout value out of
range");
    }

    if (nanos > 0) {
      timeout++;
    }

    wait(timeout);
  }

  public final void wait() throws InterruptedException {
    wait(0);
  }

  protected void finalize() throws Throwable {}
}
```


registerNatives方法

由于registerNatives方法被static块修饰，所以在加载Object类时就会执行该方法，对应的本地方法为`Java_java_lang_Object_registerNatives`，如下，

```
JNIEXPORT void JNICALL
Java_java_lang_Object_registerNatives(JNIEnv *env, jclass cls)
{
    (*env)->RegisterNatives(env, cls,
                            methods, sizeof(methods)/sizeof(methods[0]));
}
```

(*env)-> 表示访问的jni函数。JNIEnv是指向类型为JNIEnv*的一个特殊NI数据结构的指针，当由c++编译器编译时NIEnv*结构其实被定义为一个类，这个类中定义了很多内嵌函数，通过使用“->”符号，可以很方便的使用这些函数。

可以看到它间接调用了`JNINativeInterface_`结构体的方法，简单可以看成是这样：它干的事大概就是将Java层的方法名和本地函数对应起来，方便执行引擎在执行字节码时根据这些对应关系表来调用C/C++函数，如下面，将这些方法进行注册，执行引擎执行到`hashCode`方法时就可以通过关系表来查找到VM的`JVM_IHashCode`函数，其中`()I`还可以得知Java层上的类型应该转为int类型。这个映射其实就可以看成将字符串映射到函数指针。

```
static JNINativeMethod methods[] = {
  {"hashCode",    "()I",  (void *)&JVM_IHashCode},
  {"wait",    "(J)V",  (void *)&JVM_MonitorWait},
  {"notify",    "()V",  (void *)&JVM_MonitorNotify},
  {"notifyAll", "()V",(void *)&JVM_MonitorNotifyAll},
  {"clone","()Ljava/lang/Object;", (void *)&JVM_Clone},
};
```

getClass方法

getClass方法也是个本地方法，对应的本地方法为`Java_java_lang_Object_getClass`，如下：

```
JNIEXPORT jclass JNICALL
Java_java_lang_Object_getClass(JNIEnv *env, jobject this)
{
    if (this == NULL) {
        JNU_ThrowNullPointerException(env, NULL);
        return 0;
    } else {
        return (*env)->GetObjectClass(env, this);
    }
}
```

所以这里主要就是看`GetObjectClass`函数了，Java层的Class在C++层与之对应的则是`klassOop`，所以关于类的元数据和方法信息可以通过它获得。

```
JNI_ENTRY(jclass, jni_GetObjectClass(JNIEnv *env, jobject obj))
  JNIWrapper("GetObjectClass");
  DTRACE_PROBE2(hotspot_jni, GetObjectClass__entry, env, obj);
  klassOop k = JNIHandles::resolve_non_null(obj)->klass();
  jclass ret =
    (jclass) JNIHandles::make_local(env, Klass::cast(k)->java_mirror());
  DTRACE_PROBE1(hotspot_jni, GetObjectClass__return, ret);
  return ret;
JNI_END
```

hashCode方法

由前面registerNatives方法将几个本地方法注册可知，hashCode方法对应的函数为`JVM_IHashCode`，即

```
JVM_ENTRY(jint, JVM_IHashCode(JNIEnv* env, jobject handle))
  JVMWrapper("JVM_IHashCode");
  // as implemented in the classic virtual machine; return 0 if object is NULL
  return handle == NULL ? 0 : ObjectSynchronizer::FastHashCode (THREAD,
JNIHandles::resolve_non_null(handle)) ;
JVM_END
```

对于hashcode生成的逻辑由`synchronizer.cpp`的`get_next_hash`函数决定，实现比较复杂，根据hashcode的不同值有不同的生成策略，最后使用一个hash掩码处理。见Xorshift算法

```
static inline intptr_t get_next_hash(Thread * Self, oop obj) {
  intptr_t value = 0 ;
  if (hashCode == 0) {
     value = os::random() ;
  } else
  if (hashCode == 1) {
     intptr_t addrBits = intptr_t(obj) >> 3 ;
     value = addrBits ^ (addrBits >> 5) ^ GVars.stwRandom ;
  } else
  if (hashCode == 2) {
     value = 1 ;            // for sensitivity testing
  } else
  if (hashCode == 3) {
     value = ++GVars.hcSequence ;
  } else
  if (hashCode == 4) {
     value = intptr_t(obj) ;
  } else {
     unsigned t = Self->_hashStateX ;
     t ^= (t << 11) ;
     Self->_hashStateX = Self->_hashStateY ;
     Self->_hashStateY = Self->_hashStateZ ;
     Self->_hashStateZ = Self->_hashStateW ;
     unsigned v = Self->_hashStateW ;
     v = (v ^ (v >> 19)) ^ (t ^ (t >> 8)) ;
     Self->_hashStateW = v ;
     value = v ;
  }

  value &= markOopDesc::hash_mask;
  if (value == 0) value = 0xBAD ;
  assert (value != markOopDesc::no_hash, "invariant") ;
  TEVENT (hashCode: GENERATE) ;
  return value;
}
```


equals方法

这是一个非本地方法，判断逻辑也十分简单，直接==比较。

clone方法

由本地方法表知道clone方法对应的本地函数为`JVM_Clone`，clone方法主要实现对象的克隆功能，根据该对象生成一个相同的新对象（我们常见的类的对象的属性如果是原始类型则会克隆值，但如果是对象则会克隆对象的地址）。Java的类要实现克隆则需要实现Cloneable接口，`if (!klass->is_cloneable())`这里会校验是否有实现该接口。然后判断是否是数组分两种情况分配内存空间，新对象为new_obj，接着对new_obj进行copy及C++层数据结构的设置。最后再转成jobject类型方便转成Java层的Object类型。

```
JVM_ENTRY(jobject, JVM_Clone(JNIEnv* env, jobject handle))
  JVMWrapper("JVM_Clone");
  Handle obj(THREAD, JNIHandles::resolve_non_null(handle));
  const KlassHandle klass (THREAD, obj->klass());
  JvmtiVMObjectAllocEventCollector oam;

  if (!klass->is_cloneable()) {
    ResourceMark rm(THREAD);
    THROW_MSG_0(vmSymbols::java_lang_CloneNotSupportedException(), klass->external_name());
  }

  const int size = obj->size();
  oop new_obj = NULL;
  if (obj->is_javaArray()) {
    const int length = ((arrayOop)obj())->length();
    new_obj = CollectedHeap::array_allocate(klass, size, length, CHECK_NULL);
  } else {
    new_obj = CollectedHeap::obj_allocate(klass, size, CHECK_NULL);
  }
  Copy::conjoint_jlongs_atomic((jlong*)obj(), (jlong*)new_obj,
                               (size_t)align_object_size(size) / HeapWordsPerLong);
  new_obj->init_mark();

  BarrierSet* bs = Universe::heap()->barrier_set();
  assert(bs->has_write_region_opt(), "Barrier set does not have write_region");
  bs->write_region(MemRegion((HeapWord*)new_obj, size));

  if (klass->has_finalizer()) {
    assert(obj->is_instance(), "should be instanceOop");
    new_obj = instanceKlass::register_finalizer(instanceOop(new_obj), CHECK_NULL);
  }

  return JNIHandles::make_local(env, oop(new_obj));
JVM_END
```

toString方法

逻辑是获取class名称加上@再加上十六进制的hashCode。

notify方法

此方法用来唤醒线程，final修饰说明不可重写。与之对应的本地方法为`JVM_MonitorNotify`，`ObjectSynchronizer::notify`最终会调用`ObjectMonitor::notify(TRAPS)`，这个过程是ObjectSynchronizer会尝试当前线程获取free ObjectMonitor对象，不成功则尝试从全局中获取。

```
JVM_ENTRY(void, JVM_MonitorNotify(JNIEnv* env, jobject handle))
  JVMWrapper("JVM_MonitorNotify");
  Handle obj(THREAD, JNIHandles::resolve_non_null(handle));
  assert(obj->is_instance() || obj->is_array(), "JVM_MonitorNotify must apply to
an object");
  ObjectSynchronizer::notify(obj, CHECK);
JVM_END
```

ObjectMonitor对象包含一个`_WaitSet`队列对象，此对象保存着所有处于wait状态的线程，用ObjectWaiter对象表示。notify要做的事是先获取`_WaitSet`队列锁，再取出`_WaitSet`队列中第一个ObjectWaiter对象，再根据不同策略处理该对象，比如把它加入到`_EntryList`队列中。然后再释放`_WaitSet`队列锁。它并没有释放synchronized对应的锁，所以锁只能等到synchronized同步块结束时才释放。

```
void ObjectMonitor::notify(TRAPS) {
  CHECK_OWNER();
  if (_WaitSet == NULL) {
     TEVENT (Empty-Notify) ;
     return ;
  }
  DTRACE_MONITOR_PROBE(notify, this, object(), THREAD);

  int Policy = Knob_MoveNotifyee ;

  Thread::SpinAcquire (&_WaitSetLock, "WaitSet - notify") ;
  ObjectWaiter * iterator = DequeueWaiter() ;
  if (iterator != NULL) {
     TEVENT (Notify1 - Transfer) ;
     guarantee (iterator->TState == ObjectWaiter::TS_WAIT, "invariant") ;
     guarantee (iterator->_notified == 0, "invariant") ;
     if (Policy != 4) {
        iterator->TState = ObjectWaiter::TS_ENTER ;
     }
     iterator->_notified = 1 ;

     ObjectWaiter * List = _EntryList ;
     if (List != NULL) {
        assert (List->_prev == NULL, "invariant") ;
        assert (List->TState == ObjectWaiter::TS_ENTER, "invariant") ;
        assert (List != iterator, "invariant") ;
     }

     if (Policy == 0) {       // prepend to EntryList
         if (List == NULL) {
             iterator->_next = iterator->_prev = NULL ;
             _EntryList = iterator ;
         } else {
             List->_prev = iterator ;
             iterator->_next = List ;
             iterator->_prev = NULL ;
             _EntryList = iterator ;
        }
```

```
    } else
    if (Policy == 1) {      // append to EntryList
        if (List == NULL) {
            iterator->_next = iterator->_prev = NULL ;
            _EntryList = iterator ;
        } else {
           // CONSIDER:  finding the tail currently requires a linear-time walk
of
           // the EntryList.  We can make tail access constant-time by
converting to
           // a CDLL instead of using our current DLL.
           ObjectWaiter * Tail ;
           for (Tail = List ; Tail->_next != NULL ; Tail = Tail->_next) ;
           assert (Tail != NULL && Tail->_next == NULL, "invariant") ;
           Tail->_next = iterator ;
           iterator->_prev = Tail ;
           iterator->_next = NULL ;
       }
    } else
    if (Policy == 2) {      // prepend to cxq
        // prepend to cxq
        if (List == NULL) {
            iterator->_next = iterator->_prev = NULL ;
            _EntryList = iterator ;
        } else {
           iterator->TState = ObjectWaiter::TS_CXQ ;
           for (;;) {
               ObjectWaiter * Front = _cxq ;
               iterator->_next = Front ;
               if (Atomic::cmpxchg_ptr (iterator, &_cxq, Front) == Front) {
                   break ;
               }
           }
        }
    } else
    if (Policy == 3) {      // append to cxq
       iterator->TState = ObjectWaiter::TS_CXQ ;
       for (;;) {
           ObjectWaiter * Tail ;
           Tail = _cxq ;
           if (Tail == NULL) {
               iterator->_next = NULL ;
               if (Atomic::cmpxchg_ptr (iterator, &_cxq, NULL) == NULL) {
                  break ;
               }
           } else {
               while (Tail->_next != NULL) Tail = Tail->_next ;
               Tail->_next = iterator ;
               iterator->_prev = Tail ;
               iterator->_next = NULL ;
               break ;
           }
       }
    } else {
       ParkEvent * ev = iterator->_event ;
       iterator->TState = ObjectWaiter::TS_RUN ;
       OrderAccess::fence() ;
       ev->unpark() ;
```

```
      }

      if (Policy < 4) {
        iterator->wait_reenter_begin(this);
      }

      // _WaitSetLock protects the wait queue, not the EntryList.  We could
      // move the add-to-EntryList operation, above, outside the critical section
      // protected by _WaitSetLock.  In practice that's not useful.  With the
      // exception of  wait() timeouts and interrupts the monitor owner
      // is the only thread that grabs _WaitSetLock.  There's almost no
contention
      // on _WaitSetLock so it's not profitable to reduce the length of the
      // critical section.
    }

    Thread::SpinRelease (&_WaitSetLock) ;

    if (iterator != NULL && ObjectMonitor::_sync_Notifications != NULL) {
       ObjectMonitor::_sync_Notifications->inc() ;
    }
}
```

notifyAll方法

与notify方法类似，只是在取`_WaitSet`队列时不是取第一个而是取所有。

wait方法

wait方法是让线程等待，它对应的本地方法是`JVM_MonitorWait`，间接调用了`ObjectSynchronizer::wait`，与notify对应，它也是对应调用ObjectMonitor对象的wait方法。该方法较长，这里不贴出来了，大概就是创建一个ObjectWaiter对象，接着获取`_WaitSet`队列锁将ObjectWaiter对象添加到该队列中，再释放队列锁。**另外，它还会释放synchronized对应的锁，所以锁不会等到synchronized同步块结束时才释放。**

```
JVM_ENTRY(void, JVM_MonitorWait(JNIEnv* env, jobject handle, jlong ms))
  JVMWrapper("JVM_MonitorWait");
  Handle obj(THREAD, JNIHandles::resolve_non_null(handle));
  assert(obj->is_instance() || obj->is_array(), "JVM_MonitorWait must apply to
an object");
  JavaThreadInObjectWaitState jtiows(thread, ms != 0);
  if (JvmtiExport::should_post_monitor_wait()) {
    JvmtiExport::post_monitor_wait((JavaThread *)THREAD, (oop)obj(), ms);
  }
  ObjectSynchronizer::wait(obj, ms, CHECK);
JVM_END
```

finalize方法

这个方法用于当对象被回收时调用，这个由JVM支持，Object的finalize方法默认是什么都没有做，如果子类需要在对象被回收时执行一些逻辑处理，则可以重写finalize方法。

二、深度挖掘

1、你真的懂Object类吗？一个简单的类Person并没有显式继承Object类，jvm是如何关联的？

- 编译器处理（帮你加上extends Object）
- JVM处理

2、Object类的native方法深层原理及手写java程序调用c或者c++的方法扩展Object类。（详细代码演示见视频讲解）

https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

在介绍 native 之前，我们先了解什么是 JNI。

一般情况下，我们完全可以使用 Java 语言编写程序，但某些情况下，Java 可能会不满足应用程序的需求，或者是不能更好的满足需求，比如：

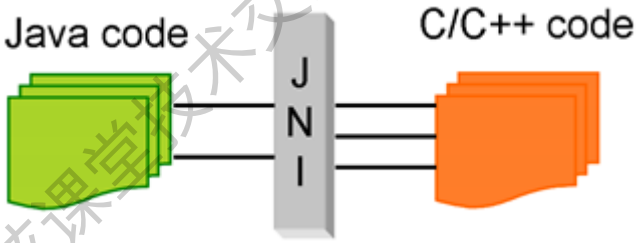
①、标准的 Java 类库不支持应用程序平台所需的平台相关功能。

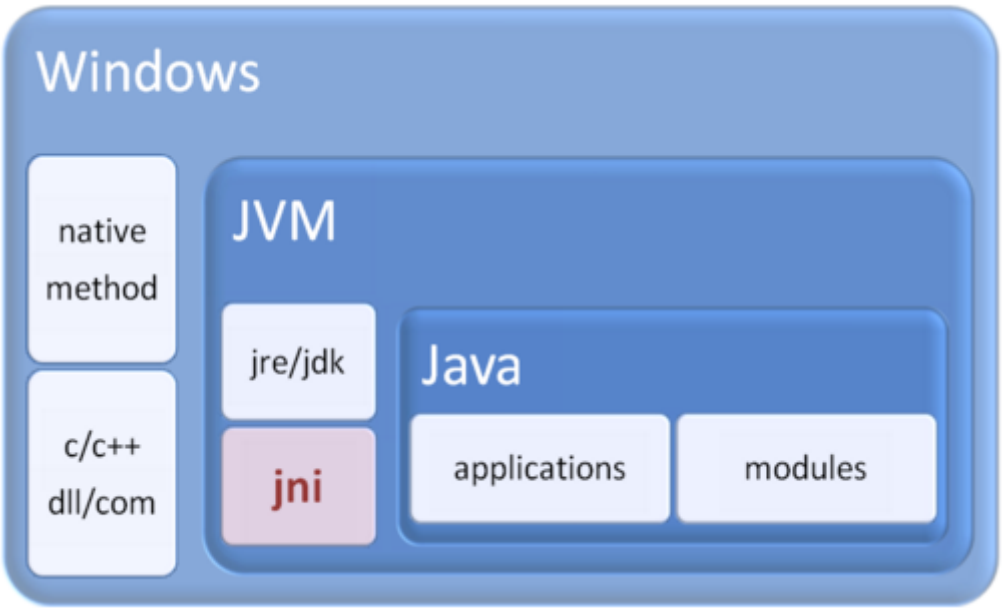
②、我们已经用另一种语言编写了一个类库，如何用Java代码调用？

③、某些运行次数特别多的方法代码，为了加快性能，我们需要用更接近硬件的语言（比如汇编）编写。

上面这三种需求，其实说到底就是如何用 Java 代码调用不同语言编写的代码。那么 JNI 应运而生了。

从Java 1.1开始，Java Native Interface (JNI)标准就成为java平台的一部分，它允许Java代码和其他语言写的代码进行交互。JNI一开始是为了本地已编译语言，尤其是C和C++而设计 的，但是它并不妨碍你使用其他语言，只要调用约定受支持就可以了。使用java与本地已编译的代码交互，通常会丧失平台可移植性。但是，有些情况下这样做是可以接受的，甚至是必须的，比如，使用一些旧的库，与硬件、操作系统进行交互，或者为了提高程序的性能。JNI标准至少保证本地代码能工作在任何Java 虚拟机实现下。





步骤如下：

- 编写带有 **native** 声明的方法的java类，生成.java文件；(注意这里出现了 native 声明的方法关键字）
- **使用** javac命令编译所编写的java类，生成.class文件；
- 使用 **javah -jni java类名** 生成扩展名为 h 的头文件，也即生成.h文件；
- 使用C/C++（或者其他编程想语言）实现本地方法，创建.h文件的实现，也就是创建.cpp文件实现.h文件中的方法；
- 将C/C++编写的文件生成动态连接库，生成dll文件；

.h 文件就是接口，cpp文件就是实现，dll文件就是jar

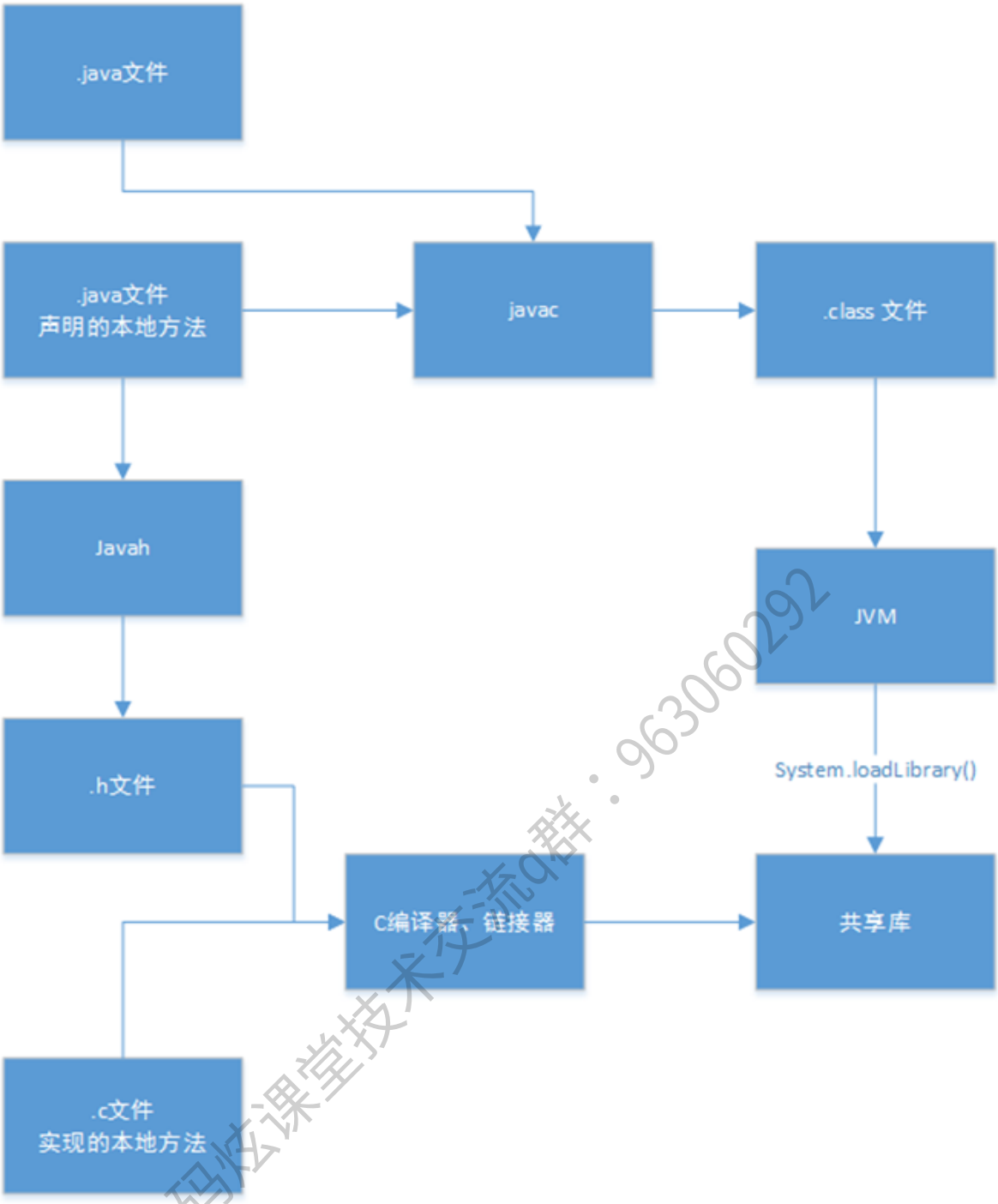
win下动态库 dll

linux 下 so

手写java native方法调用c++实现

需求：新安装一台扫描仪，java程序调用打印机驱动（c写的），传入2个整数（打印和复印张数），返回两数之和。

总结：对应流程图



3、hashCode究竟是不是内存地址？

代码验证

linux的创始人linus 。talk is cheap ，show me the code。

4、什么情况下需要重写equals()和hashCode()。

主要涉及hashmap的put和get源码

hashmap.put(new Person(1),"sss");

hashmap.put(new Person(2),"sss");

Hash算法

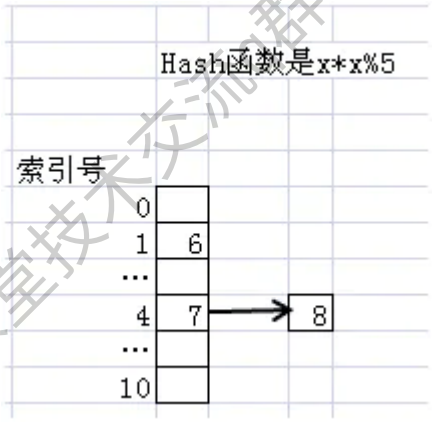
先复习一下数据结构里的一个知识点：在一个长度为 n（假设是 10000）的线性表（假设是ArrayList）里，存放着无序的数字；如果我们要找一个指定的数字，就不得不通过从头到尾依次遍历来查找。

我们再来观察Hash表（这里的Hash表纯粹是数据结构上的概念，和Java无关）。它的平均查找次数接近于 1，代价相当小，关键是在Hash表里，存放在其中的数据和它的存储位置是用Hash函数关联的。

我们假设一个Hash函数是 `x * x % 5`。当然实际情况里不可能用这么简单的Hash函数，这里纯粹为了说明方便，而Hash表是一个长度是 11的线性表。如果我们要把 6放入其中，那么我们首先会对 6用Hash函数计算一下，结果是 1，所以我们就把 6放入到索引号是 1这个位置。同样如果我们要放数字 7，经过Hash函数计算， 7的结果是 4，那么它将被放入索引是 4的这个位置。这个效果如下图所示。

这样做的好处非常明显。比如我们要从中找 6这个元素，我们可以先通过Hash函数计算 6的索引位置，然后直接从 1号索引里找到它了。

不过我们会遇到“Hash值冲突”这个问题。比如经过Hash函数计算后， 7和 8会有相同的Hash值，对此Java的HashMap对象采用的是"链地址法"的解决方案。效果如下图所示：



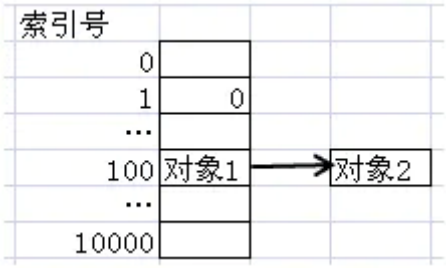
具体的做法是，为所有Hash值是 i的对象建立一个同义词链表。假设我们在放入 8的时候，发现 4号位置已经被占，那么就会新建一个链表结点放入 8。同样，如果我们要找 8，那么发现 4号索引里不是 8，那会沿着链表依次查找。

虽然我们还是无法彻底避免Hash值冲突的问题，但是Hash函数设计合理，仍能保证同义词链表的长度被控制在一个合理的范围里。这里讲的理论知识并非无的放矢，大家能在后文里清晰地了解到重写hashCode方法的重要性。

为什么要重写equals和hashCode算法？

当我们用 HashMap存入自定义的类时，如果不重写这个自定义类的equals和hashCode方法，得到的结果会和我们预期的不一样。

我们定义了一个 Key类；在其中定义了唯一的一个属性 id。当前我们先注释 equals()方法和 hashCode()方法。



结语

码炫课堂3位1体大体系课之-**【jdk源码解析&多线程&高并发】**第一期即将开启，主讲老师smart哥将会带大家彻底扒光jdk。jdk源码是一切组件的源头及基础，如果对jdk源码没有一个清晰的认识你将无法阅读tomcat，netty，dubbo，spring，springboot等优秀组件及框架的源码，你将永远只是一个CURDer，升职加薪基本与你无缘。

后期会陆陆续续开启课程如下：

【springboot&spring源码解析】

【tomcat源码解析&手写tomcat】

【netty源码解析&手写reactor框架】

【dubbo源码解析&手写RPC框架】

【redis源码解析】

最后，**纯讲源码的都是耍流氓，割韭菜。**学习源码是为了更清晰的认识原理并理解其背后的思想，如果是为了学源码而学源码，那是极其愚蠢的行为。

码炫课堂技术交流群：963060292