**交流1：**

1）clone是怎么做到的？原型模式在android里的可能在哪些地方用？

2）原型模式在C++层怎么应用？

答：

1）源码

@HotSpotIntrinsicCandidate

protected native Object clone() throws CloneNotSupportedException;

  从源码可以看出，clone()也是一个本地方法，在上一小节中，我们在Object.c文件中看到了它被定义到一个JNINativeMethod数组中：

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},

};

  由此可见，它的实现也是由JNI来处理。JNINativeMethod的结构体声明在jni.h中：

/\*

\* used in RegisterNatives to describe native method name, signature,

\* and function pointer.

\*/

typedef struct {

char \*name;

char \*signature;

void \*fnPtr;

} JNINativeMethod;

回头看clone的函数指针，它指向了JVM\_Clone，我们在jvm.h(jdk9\src\java.base\share\native\include\jvm.h)中可以看到它的声明：

JNIEXPORT jobject JNICALL

JVM\_Clone(JNIEnv \*env, jobject obj);

  而关于它的实现在jvm.cpp(hotspot-jdk9\src\share\vm\prims\jvm.cpp)中可以找到：

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;

#ifdef ASSERT

// Just checking that the cloneable flag is set correct

if (obj->is\_array()) {

guarantee(klass->is\_cloneable(), "all arrays are cloneable");

} else {

guarantee(obj->is\_instance(), "should be instanceOop");

bool cloneable = klass->is\_subtype\_of(SystemDictionary::Cloneable\_klass());

guarantee(cloneable == klass->is\_cloneable(), "incorrect cloneable flag");

}

#endif

// Check if class of obj supports the Cloneable interface.

// All arrays are considered to be cloneable (See JLS 20.1.5)

if (!klass->is\_cloneable()) {

ResourceMark rm(THREAD);

THROW\_MSG\_0(vmSymbols::java\_lang\_CloneNotSupportedException(), klass->external\_name());

}

// Make shallow object copy

const int size = obj->size();

oop new\_obj\_oop = NULL;

if (obj->is\_array()) {

const int length = ((arrayOop)obj())->length();

new\_obj\_oop = CollectedHeap::array\_allocate(klass, size, length, CHECK\_NULL);

} else {

new\_obj\_oop = CollectedHeap::obj\_allocate(klass, size, CHECK\_NULL);

}

// 4839641 (4840070): We must do an oop-atomic copy, because if another thread

// is modifying a reference field in the clonee, a non-oop-atomic copy might

// be suspended in the middle of copying the pointer and end up with parts

// of two different pointers in the field. Subsequent dereferences will crash.

// 4846409: an oop-copy of objects with long or double fields or arrays of same

// won't copy the longs/doubles atomically in 32-bit vm's, so we copy jlongs instead

// of oops. We know objects are aligned on a minimum of an jlong boundary.

// The same is true of StubRoutines::object\_copy and the various oop\_copy

// variants, and of the code generated by the inline\_native\_clone intrinsic.

assert(MinObjAlignmentInBytes >= BytesPerLong, "objects misaligned");

Copy::conjoint\_jlongs\_atomic((jlong\*)obj(), (jlong\*)new\_obj\_oop,

(size\_t)align\_object\_size(size) / HeapWordsPerLong);

// Clear the header

new\_obj\_oop->init\_mark();

// Store check (mark entire object and let gc sort it out)

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\_oop, size));

Handle new\_obj(THREAD, new\_obj\_oop);

// Special handling for MemberNames. Since they contain Method\* metadata, they

// must be registered so that RedefineClasses can fix metadata contained in them.

if (java\_lang\_invoke\_MemberName::is\_instance(new\_obj()) &&

java\_lang\_invoke\_MemberName::is\_method(new\_obj())) {

Method\* method = (Method\*)java\_lang\_invoke\_MemberName::vmtarget(new\_obj());

// MemberName may be unresolved, so doesn't need registration until resolved.

if (method != NULL) {

methodHandle m(THREAD, method);

// This can safepoint and redefine method, so need both new\_obj and method

// in a handle, for two different reasons. new\_obj can move, method can be

// deleted if nothing is using it on the stack.

m->method\_holder()->add\_member\_name(new\_obj(), false);

}

}

// Caution: this involves a java upcall, so the clone should be

// "gc-robust" by this stage.

if (klass->has\_finalizer()) {

assert(obj->is\_instance(), "should be instanceOop");

new\_obj\_oop = InstanceKlass::register\_finalizer(instanceOop(new\_obj()), CHECK\_NULL);

new\_obj = Handle(THREAD, new\_obj\_oop);

}

return JNIHandles::make\_local(env, new\_obj());

JVM\_END

  代码会首先判断准备被clone的类是否实现了Cloneable接口，若为否则会抛CloneNotSupportException异常。接着根据要clone的对象是否数组进行新对象的内存分配以及信息写入，然后copy内存块信息到新对象。接下来初始化对象头，进行存储检查标记新对象分配堆栈，然后将需要特别注册的方法进行注册，最后将复制完成的内存块转换成本地对象并将其返回。

  这里主要执行copy的代码是Copy::conjoint\_jlongs\_atomic，这个方法在hotspot-jdk9\src\share\vm\utilities\copy.hpp中实现：

static void conjoint\_jlongs\_atomic(jlong\* from, jlong\* to, size\_t count) {

assert\_params\_ok(from, to, LogBytesPerLong);

pd\_conjoint\_jlongs\_atomic(from, to, count);

}

  方法中执行了pd\_conjoint\_jlongs\_atomic()方法，其位于hotspot-jdk9\src\cpu\zero\vm\copy\_zero.hpp中：

static void pd\_conjoint\_jlongs\_atomic(jlong\* from, jlong\* to, size\_t count) {

\_Copy\_conjoint\_jlongs\_atomic(from, to, count);

}

  再接着执行\_Copy\_conjoint\_jlongs\_atomic()方法，在hotspot-jdk9\src\os\_cpu\linux\_zero\vm\os\_linux\_zero.cpp中：

void \_Copy\_conjoint\_jlongs\_atomic(jlong\* from, jlong\* to, size\_t count) {

if (from > to) {

jlong \*end = from + count;

while (from < end)

os::atomic\_copy64(from++, to++);

}

else if (from < to) {

jlong \*end = from;

from += count - 1;

to += count - 1;

while (from >= end)

os::atomic\_copy64(from--, to--);

}

}

  至此，我们看到了整个对象copy的过程，就是把from指针指向的内存的值赋给to指针指向的内存，这是一个简单的拷贝操作。可以知道，在经过了clone()方法生成的新对象并不是通过构造函数来创建，而是直接在内存层面进行了copy操作。

2）C++层原型模式的简单实现，利用拷贝构造函数。

1. #include <cstdio>
3. //接口
4. **class** CPrototype
5. {
6. **public**:
7. CPrototype(){}
8. **virtual** ~CPrototype(){}
10. **virtual** CPrototype\* Clone() = 0;
11. };
13. //实现
14. **class** CConcretePrototype : **public** CPrototype
15. {
16. **public**:
17. CConcretePrototype():m\_counter(0){}
18. **virtual** ~CConcretePrototype(){}
20. //拷贝构造函数
21. CConcretePrototype(**const** CConcretePrototype& rhs)
22. {
23. m\_counter = rhs.m\_counter;
24. }
26. //复制自身
27. **virtual** CPrototype\* Clone()
28. {
29. //调用拷贝构造函数
30. **return** **new** CConcretePrototype(\***this**);
31. }
33. **private**:
34. **int** m\_counter;
35. };
37. **int** main(**int** argc, **char** \*\*argv)
38. {
39. //生成对像
40. CPrototype\* conProA = **new** CConcretePrototype();
41. //复制自身
42. CPrototype\* conProB = conProA->Clone();
44. **delete** conProA; conProA=NULL;
45. **delete** conProB; conProB=NULL;
47. **return** 0;
48. }

在C++层原型模式和拷贝构造函数的主要区别是，原型模式clone后返回的基类对象，用户在写代码时不需要知道具体的派生类。而拷贝构造函数必须传入派生类对象才可以。

**交流2：**

1）开闭原则和策略模式很像，它们具体的差异是什么？

https://www.cnblogs.com/machine/p/pattern.html#sec-2-1