InterfaceInjection

Patch name: inti.patch
Description: inti.txt

Implementation Notes

See InterfaceCalls for a discussion of how interface call sites work.

Each `klass` structure already includes (allocated inline) a 2-dimensional ragged table of its statically defined interfaces. The spine of the table has pairs, <`oop iklass, offset_t itable`>. The spine is terminated by a tuple <NULL, 0>.. The itable offsets are relative to the enclosing `klass` (the receiver type).

Each itable is an array of the form `methodOop target[N]`, where `N` is the number of methods in the interface.

Interface dispatch (for polymorphic call sites) searches the spine of the table, and dips into the matching itable, picking out the target method corresponding to the index (in 0..N-1) of the abstract method in the interface.

See definition of `klassItable::setup_itable_offset_table` around line 1100 of klassVtable.cpp. The pair type is `itableOffsetEntry`, present in every concrete `instanceKlass`.

Since the offset entries on the spine of the table have two words, one word of the terminating entry can have a sentinel value, and the other is free to point to a linked list of extension records. So we repurpose the second word head of a linked list of extension records: <NULL, extension>.

There is one extension record for one interface injection event. They are linked through "next" fields (in an arbitrary order), and the incoming interface `klass` is matched against their "key" fields. A matching extension record has N additional words, all methodOops; those N words are exactly like a statically defined embedded itable. The extension records are actually just system arrays (type `Object[]`) whose lengths are `2+N`, with key at element 0, "next" link at element 1, and the implementation methods (if any) at elements 2 through `2+N-1`.

In order to pack these dynamic itables, MethodHandles must be lowered to methodOops. For the special case of direct method handles, the original methodOop can be reused. For other (adapted or bound) method handles, a new methodOop must be created to wrap the method handle. This is a dark and dirty secret that nobody but the JVM will know about. (See the auto-generation of invoke methods in methodOop.cpp of meth.patch.)

A wrapper methodOop for a method handle `mh` of type `R(A...)` could look like bytecodes for this pseudocode:

structure of a wrapper method

```
static final MethodHandle MH = mh; // stored directly in m's constant pool
static R m(A... a...) {
  // ldc #MH
  // push l0; push l1; ... for all A
  // invokevirtual MethodHandle.invoke(A...)R
  return MH.<R>invoke(a);
  // return
}
```

The meth.patch code already uses the oop-in-a-constant-pool technique for autogenerated `MethodHandle.invoke` methods (of which there is an infinite variety).

Design decisions and considerations

- The need to be able to do dynamic invocations in `<clinit>` of an interface is less than minimal. It should be safe to reuse the `_bootstrap_method oop` in `instanceKlass` for storing the injector for injectable interfaces.
- The injector for an injectable interface is defined by the `<clinit>` method on the injectable interface. This is done by invoking `InterfaceInjector.setInjector(InterfaceInjector)` from `<clinit>`.
- How should an interface be marked as injectable? Options include:
  - Adding a flag that marks an interface as injectable. The following flags have no meaning for classes yet:
    - `0x0040` - `JVM_ACC_VOLATILE` for fields, `JVM_ACC_BRIDGE` for methods
    - `0x0080` - `JVM_ACC_TRANSIENT` for fields, `JVM_ACC_VARARGS` for methods
    - `0x0100` - `JVM_ACC_NATIVE` for methods
  - Adding an injectable annotation to the class file, and make sure that this annotation gets loaded early enough.
  - Eagerly load all interfaces. The interfaces that define an injector during class initialization are defined as injectable.
    - To reduce the need for eager loading the constant pool could be scanned to determine if the interface references the `setInjector`-method.

Other tricky parts

Retry path for `invokeinterface`

When the linear-table and linked-list lookups fail, the invoking code needs to back up to a safe place, call the (C code of the) JVM, find out whether an injection is to be made, and either patch the `klass` (for next time) or throw an error. The tricky part is finding a backoff point where it's safe to call out of Java.

`instanceof`/`checkcast` logic

(This logic is duplicated in a couple of places, notably `graphKit.cpp`.)

This needs a linked list search and a whole new negative logic side. As with invokeinterface, if the initial searches fail, there needs to be a backoff and upcall to the JVM, to possibly inject the interface. Since negative interface checks are too common to handle this way (via an upcall) some negative filtering needs to be put in. We could do it in a couple of ways:

- Interface klasses which are not injectable (the vast majority) should have bits in their header which identifies them as such, so that instanceof can return false more quickly. A good way to do this may be to add a second `Klass::secondary_super_cache` just for injectable interfaces; then the secondary_super_offset for an interface will take one of two distinct values (instead of the single value it takes today), depending on which secondary_super_cache it uses. This simultaneously makes it easy to detect injectables (by `secondary_super_offset == offsetof (&secondary_super_cache[1]))` and also allocates a word in every `klass` to optimize the lookup of injected interfaces.
- (Can delay this for the POC.) Introduce a negative super type cache: `Klass::secondary_non_super_cache`. Use it as a first resort, to avoid upcalls on negative type tests of injectables.

negative injection record on each `klass`

Somehow we must record on each `klass` which interfaces have *refused* to inject. (They are the guys that show up in `secondary_non_super_cache`.) Probably a chunky linked list: A linked list of arrays, like the extension records above. (Or something else?) The important thing is not to ask the same injection question twice; record yesses as extension records and noes as entries on the negative injection list.

Weak-pointer-ization

(Can delay this a while; we'll get a GC guy to help fix it.) References to interface klasses on both positive and negative sides should be weak, so that the GC can delete entries for unreachable interfaces.

Compiler integration

(Can delay this, as long as the calls truly bottom out in methodOops.) The ciMethod type should be able to resolve against injected interfaces, so as to be able to inline such calls. It may be easiest, both now and in the long run, to reify (in the CI, at least) method handles as if they were in fact methods. That means generating throwaway bytecodes inside the JVM, exactly as above.

Security

(Can delay this, as long as we expose the API only to privileged code, like `Unsafe`.) Needs some analysis, especially in the case of non-public interfaces. Perhaps we punt on non-public interfaces, but probably there's a more permissive solution, such as allowing injection if the target class itself would have been able to access the interface. (Or, maybe non-public interfaces are an important use case, precisely for adding new aspects to classes that would not be able to add them to themselves.)

Java API design.

Requires great precision to say exactly when injection requests happen, how they are resolved, and how the resolutions affect subsequent execution. The basic idea is to have each *exact*, *concrete* type get injected at most once, and exactly once if an (exact) instance of that type (exactly) gets an invokeinterface, checkcast, or instanceof. (Or reflective or optimized versions thereof!)

If two types are related by inheritance, it may be best to query supertypes before subtypes. (But never `Object`.)

Compile-time optimizations

If interfaces are marked as being injectable or not, then the JIT would not have to emit the code for looking up the interface in the extension list if it's not injectable. To do this
the method that emits the instructions needs to have access to the `klass` representing the interface; this needs to be done through the `ciKlass` API.

Customizing code like that is usually only done by the JIT's optimizer. (Esp. the server JIT.) The interpreter can usually afford (until proven otherwise!) to use the most generic code sequences.

There are five places where interface types must be checked by the JIT:

1. invokeinterface
2. instanceof
3. checkcast
4. checkcast implicit in aastore checks
5. reflective versions of 1, 2, or 3.

In cases 1, 3, and 4, there is no strong need to customize for non-injectable interfaces, because the cost of failure is always an expensive exception, so it doesn't matter if you did a useless check of the extension records.

In case 2, there is definitely a need for a negative supertype cache in class Klass.

In case 5, if there is a JIT intrinsic which folds the reflective idiom to a non-reflective one, then it can be reduced to one of the previous cases (1,2,3). Otherwise, it is probably slow, and can be treated in full generality all the time. A possible exception is that the negative supertype cache should be consulted for reflective instanceof.

Cases 4 and 5 are interesting in that the interface type being tested against is non-constant (aastore checks against a varying array element type). In those cases, you either use full generality all the time, or use both code sequences, selected by a dynamic test for injectability.

Bottom line, per case:
1. always check for extension records
2. always use the negative cache (after fast positive tests), and customize the code in the JIT (`GraphKit::gen_subtype_check`)
3. customize in the JIT; negative cache buys nothing
4. customize in the JIT if possible; negative cache buys nothing
5. let the JIT do its thing for intrinsics; use the negative cache for `Class.isInstance` and `Class.isAssignableFrom`

References

- http://blogs.oracle.com/jrose/entry/interface_injection_in_the_vm
- http://journal.thobe.org/2008/07/my-jvm-whishlist-pt-1-interface.html