Ⅰ. 引论

反射和 Unsafe

1. 反射回顾

在 Java 中，反射是一种很灵活的特性，它允许用户在运行时通过 Class 对象获取任意类的结构信息（如字段、方法、构造函数等），并能动态调用对象的方法或修改字段值。在使用自定义 ClassLoader 动态加载字节码时，也需要通过反射去使用被加载的类（因为自定义 ClassLoader 通常都不和代码上下文 ClassLoader 在同一条委托链上，或位于上下文 ClassLoader 之下，导致在代码上下文中无法找到这些动态加载的 class 字节码）。在实际应用中获取、修改 private 字段一般也会通过反射进行。反射使用方法中的正常符合规范的做法不在本文的讨论范围之内。

随着 Java 版本的更新，安全机制越来越严格，从 Java 17 开始已经不再允许反射修改字段和方法的修饰符。例如，在早期可以像下面这样通过修改字段的修饰符来修改 final 变量的值：

```
Field field = TargetClass.class.getDeclaredField("any_field");
field.setAccessible(true);

Field modifiersField = Field.class.getDeclaredField("modifiers");
modifiersField.setAccessible(true);
modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);

field.set(target_obj, new_value);
```

Java 17 之后设置 modifiers 变量不再生效，final 语义的保证由更底层的机制实现。

但是，这个案例中依然有值得注意的地方，从中可以看出用户让变量可以访问的**核心方法是 setAccessible()**，它可以设置目标是否可以被用户访问（包括修改字段值、方法调用等），它的底层实现实际上是设置了 AccessibleObject 类（Field、Method、Constructor 类等的父类）的 **override 成员**，该变量控制这个对象是否有访问权限检查（Java 的安全相关的核心类几乎都设置了访问权限检查，用户是不可访问的），具有访问权限检查的对象不可被用户操作。该变量为 true 则表示无权限检查，设置为 false 代表有权限检查。这个 override 成员很重要，本系列文章的一切技巧根源就源自修改该变量。

```java
@CallerSensitive  // overrides in Method/Field/Constructor are @CS
public void setAccessible(boolean flag) {
    AccessibleObject.checkPermission();
    setAccessible0(flag);
}

/**
 * Sets the accessible flag and returns the new value
 */
boolean setAccessible0(boolean flag) {
    this.override = flag;
    return flag;
}
```

上面的案例展示了使用反射修改字段的基本原理。往后的版本想要修改 final 字段依然是可行的，但需要借助 Unsafe。需要注意的是 static final 修饰的变量可能会被内联优化，例如原生类型、String 字面值或者 null 值。如果 Unsafe 修改目标变量后发现使用该变量的地方依旧未被修改（尤其是在跨类修改一个未在该类中使用的字段时），那么就说明目标变量的值已经被内联。例如声明 static final Object obj = null; 并在后期修改 obj 时，如果打印 obj 的值，会发现修改前后 obj 一直是 null，这并不是修改失败，而是 null 值已经被内联进了打印方法中，修改前后打印的都是 null 值本身而不是 obj。在修改方法所在类里显示引用 null 字面值的目标 static final 字段（只需要引用，不需要使用），可以防止修改失败。

2. Unsafe

Unsafe 如其名称一样不够安全，它为用户提供了直接进行底层操作的接口，需要谨慎使用，不当的使用会造成严重后果。sun.misc.Unsafe 是 JDK 的内部 Unsafe 类 jdk.internal.misc.Unsafe（通常称之为 InternalUnsafe）的部分功能封装（并且一些功能还会添加限制），后者为 JDK 内部使用的类，支持更多底层操作但不允许用户使用：不论你 import 它或者声明变量，编译器均会报错编译不通过。

Unsafe 实例可以通过反射获取。

```java
public static Unsafe unsafe;

theUnsafe = Unsafe.class.getDeclaredField("theUnsafe");
theUnsafe.setAccessible(true);
unsafe = (Unsafe) theUnsafe.get(null);
```

同样，InternalUnsafe 也可以通过反射获取，但是储存其实例的变量类型必须是 Object，这是因为 jdk.internal.misc.Unsafe 不允许用户使用，即使声明变量也不行。

```
static Class<?> internalUnsafeClass;
static Object internalUnsafe;

private static Method objectFieldOffset$Field;// 没有检查的jdk.internal.misc.Unsafe.objectFieldOffset()
private static Method objectFieldOffset$Class$String;
private static Method staticFieldBase;
private static Method staticFieldOffset;


internalUnsafeClass = Class.forName("jdk.internal.misc.Unsafe");
theInternalUnsafe = Unsafe.class.getDeclaredField("theInternalUnsafe");
theInternalUnsafe.setAccessible(true);
internalUnsafe = theInternalUnsafe.get(null);
```

得到 jdk.internal.misc.Unsafe 对象后，还需要将其 Method 对象的 override 成员设置为 true 才可以调用。

使用 sun.misc.Unsafe 访问或修改变量通常遵循以下步骤：

1. 获取目标字段的 java.lang.reflect.Field 对象；
2. 获取 Field 内存偏移量 offset，这可以通过 objectFieldOffset()实现（如果是静态成员，需要获取 staticFieldBase 对象并通过 staticFieldOffset()获取内存偏移量）；
3. 使用 sun.misc.Unsafe 的 get*()函数族读取或使用 put*()函数族修改目标对象指定 offset 的值。**该方法直接操作内存，无视 Java 的规范定义的修饰符语义，这意味着它可以修改任何变量。**如果目标对象或者 offset 值错误则会造成不可预见的后果，绝大部分情况下是 JVM 直接崩溃。

实际上，这三个步骤要同时做到并不容易，尤其是涉及 Java 核心类。

获取字段 Field

获取 Field 对象是受限制的，并不是所有字段都允许用户去获取。在 JDK 内部存在反射类 jdk.internal.reflect.Reflection，它为反射引入了过滤限制，其内部声明了 fieldFilterMap 和 methodFilterMap 这两个 Map<Class<?>, Set<String>> 对象，所有位于该过滤 Map 内的字段或方法均无法通过反射获取（包括这两个 Map 本身）。

```
/** Used to filter out fields and methods from certain classes from public
    view, where they are sensitive or they may contain VM-internal objects.
    These Maps are updated very rarely. Rather than synchronize on
    each access, we use copy-on-write */
private static volatile Map<Class<?>, Set<String>> fieldFilterMap;
private static volatile Map<Class<?>, Set<String>> methodFilterMap;
private static final String WILDCARD = "*";
public static final Set<String> ALL_MEMBERS = Set.of(WILDCARD);

static {
    fieldFilterMap = Map.of(
        Reflection.class, ALL_MEMBERS,
        AccessibleObject.class, ALL_MEMBERS,
        Class.class, Set.of("classLoader", "classData"),
        ClassLoader.class, ALL_MEMBERS,
        Constructor.class, ALL_MEMBERS,
        Field.class, ALL_MEMBERS,
        Method.class, ALL_MEMBERS,
        Module.class, ALL_MEMBERS,
        System.class, Set.of("security")
    );
    methodFilterMap = Map.of();
}
```

但是正如 Java 所倡导的，反射和句柄是相辅相成的，本系列后面的文章会讲解使用 Handle 去获取这两个字段和方法过滤 Map 的 VarHandle 从而修改它们的方法，这将可以实现移除 Java 反射过滤用以获取所需的任何字段或方法。

AccessibleObject 类的 override 成员理所当然地也在反射过滤 Map 里，因此是不能通过反射获取其 Field 的，进而也就无法获取其内存偏移量 offset。正确的方法是自己构造一个 AccessibleObject 的镜像类，且该镜像类拥有和 AccessibleObject 类完全一致的继承关系和成员变量排布，保证两个类在内存中的布局完全一致，这样镜像类的 override 成员内存偏移量就是实际的 AccessibleObject 类的 override 成员偏移量。除此以外也可以根据 Java 的对象内存模型推断出成员的内存偏移量 offset，但值得注意的是从 Java 18 版本开始 sun.misc.Unsafe.objectFieldOffset()方法标注了@Deprecated 注解，并说明以后的 Java 版本中类的成员偏移量可能将不再保持继承不变。**截至目前 Java 21 版本，不论继承关系如何，override 的内存偏移量固定为 12**，这意味着不论是 Field、Method 还是 Constructor 类，内存偏移量为 12 的 boolean 值就是成员 override。

获取字段内存偏移量

在不构造镜像类或直接推断内存偏移量的情况下，如果无法获取到 Field，那么就无法借助 sun.misc.Unsafe 来获取 offset 了，因为它的唯一一个 objectFieldOffset()方法只接收 Field 类型的参数。但是，jdk.internal.misc.Unsafe 类是可以直接不经过反射（这意味着目标字段即使位于过滤 Map 也没有影响）获取字段 offset 的。当然，要使用这个方法还是需要先找到 override 的内存偏移

量，才能移除该方法的权限检查。

```
/**
 * Reports the location of the field with a given name in the storage
 * allocation of its class.
 *
 * @throws NullPointerException if any parameter is {@code null}.
 * @throws InternalError if there is no field named {@code name} declared
 *         in class {@code c}, i.e., if {@code c.getDeclaredField(name)}
 *         would throw {@code java.lang.NoSuchFieldException}.
 *
 * @see #objectFieldOffset(Field)
 */
public long objectFieldOffset(Class<?> c, String name) {
    if (c == null || name == null) {
        throw new NullPointerException();
    }

    return objectFieldOffset1(c, name);
}
```

从 Java 14 开始引入的 record 类，是一种由 Java 语言规范保证了的实例化后不可修改的类，这时通过 sun.misc.Unsafe 修改变量的方法也不再适用，这是因为 **sun.misc.Unsafe. objectFieldOffset()方法会检查目标字段是否是隐藏类或者 record 的字段，如果是则直接抛出异常，不返回 offset。**

```
@Deprecated(since="18")
@ForceInline
public long objectFieldOffset(Field f) {
    if (f == null) {
        throw new NullPointerException();
    }
    Class<?> declaringClass = f.getDeclaringClass();
    if (declaringClass.isHidden()) {
        throw new UnsupportedOperationException("can't get field offset on a hidden class: " + f);
    }
    if (declaringClass.isRecord()) {
        throw new UnsupportedOperationException("can't get field offset on a record class: " + f);
    }
    return theInternalUnsafe.objectFieldOffset(f);
}
```

想要得到隐藏类或 record 类的字段偏移量的唯一方法是使用 JDK 内部 jdk.internal.misc.Unsafe 的 objectFieldOffset()方法。

使用 Unsafe 访问或修改目标字段

好在 sun.misc.Unsafe 类的 get*()和 put*()函数族是没有添加额外限制的，和内部的 jdk.internal.misc.Unsafe 类提供的功能完全一致。使用 Unsafe 的 put*()函数族时需要用户自行指定目标偏移量地址的值类型，这点和 C++的指针一样。错误的类型会导致对象数据损坏，引发一系列不可预测的后果。

3. 使用 InternalUnsafe

从上面的讨论中可以看出，jdk.internal.misc.Unsafe 是一个功能强大的类，目前我们已经可以获取到它的实例，接下来就需要通过反射去获取其方法 Method 对象，并通过设置其 override 为 true 来调用它的方法。幸运的是，jdk.internal.misc.Unsafe 的方法均不在 Reflection 的过滤 Map 里面，可以使用反射正常获取其 Method 对象。

作为教程的起点，第一步就是利用 **override 内存偏移量为 12** 的事实编写一个移除 AccessibleObject 对象访问权限检查的方法。下面的代码使用 sun.misc.Unsafe 的 putBoolean()方法直接在目标 AccessibleObject 对象（这里是 Method）的内存偏移量为 12 的位置设置为指定的值。

```
public static <AO extends AccessibleObject> AO setAccessible(AO accessibleObj, boolean accessible) {
    unsafe.putBoolean(accessibleObj, java_lang_reflect_AccessibleObject_override_offset, accessible);
    return accessibleObj;
}

public static Field setAccessible(Class<?> cls, String field_name, boolean accessible) {
    Field f = Reflection.getField(cls, field_name);
    setAccessible(f, accessible);
    return f;
}
```

```
public static <AO extends AccessibleObject> AO removeAccessCheck(AO access_obj) {
    return InternalUnsafe.setAccessible(access_obj, true);
}
```

进一步封装可以得到直接调用目标 Method 的方法：

```
public static Object invoke(Object obj, Method method, Object... args) {
    try {
        return ObjectManipulator.removeAccessCheck(method).invoke(obj, args);
    } catch (IllegalArgumentException | IllegalAccessException | SecurityException ex) {
        System.err.println("invoke failed. obj=" + obj.toString() + ", method_name=" + method.getName());
        ex.printStackTrace();
    } catch (InvocationTargetException ex) {
        System.err.println("invoke method throws exception. obj=" + obj.toString() + ", method_name=" + method.getName());
        ex.getCause().printStackTrace();
    }
    return null;
}
```

这样，只要获取到任意一个 Method 对象就可以无视访问权限调用。

字段需要使用 jdk.internal.misc.Unsafe.objectFieldOffset()方法去获取 offset，这样就可以获取到 record 和隐藏类的字段内存偏移量，从而实现修改其成员变量的效果。先为要用到的 jdk.internal.misc.Unsafe 方法 Method 对象取消权限检查：

```
objectFieldOffset$Field = setAccessible(internalUnsafeClass.getDeclaredMethod("objectFieldOffset", Field.class), true);
objectFieldOffset$Class$String = setAccessible(internalUnsafeClass.getDeclaredMethod("objectFieldOffset", Class.class, String.class), true);
staticFieldBase = setAccessible(internalUnsafeClass.getDeclaredMethod("staticFieldBase", Field.class), true);
staticFieldOffset = setAccessible(internalUnsafeClass.getDeclaredMethod("staticFieldOffset", Field.class), true);
```

```
public static long objectFieldOffset(Field field) {
    try {
        return (long) objectFieldOffset$Field.invoke(internalUnsafe, field);
    } catch (IllegalAccessException | InvocationTargetException e) {
        e.printStackTrace();
    }
    return INVALID_FIELD_OFFSET;
}

public static long objectFieldOffset(Class<?> cls, String field_name) {
    try {
        return (long) objectFieldOffset$Class$String.invoke(internalUnsafe, cls, field_name);
    } catch (IllegalAccessException | InvocationTargetException e) {
        e.printStackTrace();
    }
    return INVALID_FIELD_OFFSET;
}

public static Object staticFieldBase(Field field) {
    try {
        return staticFieldBase.invoke(internalUnsafe, field);
    } catch (IllegalAccessException | InvocationTargetException e) {
        e.printStackTrace();
    }
    return null;
}

public static long staticFieldOffset(Field field) {
    try {
        return (long) staticFieldOffset.invoke(internalUnsafe, field);
    } catch (IllegalAccessException | InvocationTargetException e) {
        e.printStackTrace();
    }
    return INVALID_FIELD_OFFSET;
}
```

之后进一步封装可以得到直接读取和修改目标 Field 的方法：

```
public static Object access(Object obj, String field_name) {
    try {
        Field field = ObjectManipulator.removeAccessCheck(Reflection.getField(obj, field_name));
        return field.get(obj);
    } catch (IllegalArgumentException | IllegalAccessException | SecurityException ex) {
        System.err.println("access failed. obj=" + obj.toString() + ", field_name=" + field_name);
        ex.printStackTrace();
    }
    return null;
}

public static Object access(Object obj, Field field) {
    try {
        return ObjectManipulator.removeAccessCheck(field).get(obj);
    } catch (IllegalArgumentException | IllegalAccessException ex) {
        System.err.println("access failed. obj=" + obj.toString() + ", field_name=" + field.getName());
        ex.printStackTrace();
    }
    return null;
}
```

```
public static boolean setObject(Object obj, Field field, Object value) {
    if (field == null)
        return false;
    InternalUnsafe.putObject(obj, field, value);
    return true;
}

public static boolean setObject(Object obj, String field, Object value) {
    return setObject(obj, Reflection.getField(obj, field), value);
}

public static Object getObject(Object obj, Field field) {
    if (field == null)
        return false;
    return InternalUnsafe.getObject(obj, field);
}

public static Object getObject(Object obj, String field) {
    return getObject(obj, Reflection.getField(obj, field));
}
```

这里只展示了 Object 类型值的读取和修改，其他原生类型也使用一样的思路实现。如果要修改数组元素，则需要使用 arrayBaseOffset()方法，在此不再赘述。

至此，只要可以获取到目标 Field、Method 或 Constructor 对象，就可以对它们进行访问、修改。当然，要获取它们并不容易，后面将使用 Handle 去获取被过滤的字段和方法，以及 native 方法。利用这些技巧可以实现相当多正常情况下不可实现的功能。

附录

Open JDK 中关于 Unsafe 工作原理的部分函数源码追溯

```
UNSAFE_ENTRY(void, Unsafe_PutReference(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jobject x_h)) {
  oop x = JNIHandles::resolve(x_h);
  oop p = JNIHandles::resolve(obj);
  assert_field_offset_sane(p, offset);
  HeapAccess<ON_UNKNOWN_OOP_REF>::oop_store_at(p, offset, x);
} UNSAFE_END
```

jdk/src/hotspot/share/prims/unsafe.cpp

```
template <typename T>
static inline void oop_store_at(oop base, ptrdiff_t offset, T value) {
   verify_heap_oop_decorators<store_mo_decorators>();
   typedef typename AccessInternal::OopOrNarrowOop<T>::type OopType;
   OopType oop_value = value;
   AccessInternal::store_at<decorators | INTERNAL_VALUE_IS_OOP>(base, offset, oop_value);
}
```

jdk/src/hotspot/share/oops/access.hpp

```
template <DecoratorSet decorators, typename T>
inline void store_at(oop base, ptrdiff_t offset, T value) {
  verify_types<decorators, T>();
  using DecayedT = std::decay_t<T>;
  DecayedT decayed_value = value;
  const DecoratorSet expanded_decorators = DecoratorFixup<decorators |
                                           (HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value ?
                                            INTERNAL_CONVERT_COMPRESSED_OOP : DECORATORS_NONE)>::value;
  PreRuntimeDispatch::store_at<expanded_decorators>(base, offset, decayed_value);
}
```

jdk/src/hotspot/share/oops/accessBackend.hpp

```
template <DecoratorSet decorators, typename T>
inline static typename EnableIf<
  HasDecorator<decorators, AS_RAW>::value>::type
store_at(oop base, ptrdiff_t offset, T value) {
  store<decorators>(field_addr(base, offset), value);
}

template <DecoratorSet decorators, typename T>
inline static typename EnableIf<
  !HasDecorator<decorators, AS_RAW>::value>::type
store_at(oop base, ptrdiff_t offset, T value) {
  if (is_hardwired_primitive<decorators>()) {
    const DecoratorSet expanded_decorators = decorators | AS_RAW;
    PreRuntimeDispatch::store_at<expanded_decorators>(base, offset, value);
  } else {
    RuntimeDispatch<decorators, T, BARRIER_STORE_AT>::store_at(base, offset, value);
  }
}
```

jdk/src/hotspot/share/oops/accessBackend.hpp

关于这部分的内容较复杂，涉及到底层 oop 的访问和操作。另外可参阅与 oop 紧密相关的 jdk/src/hotspot/share/oops/accessDecorators.hpp。