摆脱 --add-opens，使用 Unsafe 突破 Java17 强封装

前言

众所周知，在 Java16 版本之后，Java 使用了强封装，任何对 JDK 内部类的反射、使用，都要在编译期添加`--add-exports`，在运行时添加`--add-opens`，否则就无法通过编译或者运行时抛出`IllegalAccessException`，但是这种封装对于框架或工具组件的开发带来了很多麻烦

偶然发现一个叫 [Burningwave Core](http://cxyroad.com/”https://github.com/burningwave/core”) 的库，可以在不添加任何参数的情况下，实现 `ALL Module add-opens to ALL` 的效果，经过一番研究，提取了纯 java 实现的版本（其组件 [jvm-driver](http://cxyroad.com/”https://github.com/toolfactory/jvm-driver”) 还提供了 JNI 等多种后备模式的版本），让我们来打开潘多拉的魔盒吧

1. JDK 不得不暴露的弱点

由于使用 `sun.misc.Unsafe` 框架/组件实在太多，迫于对社区和生态适应的压力，Java 只得开放对于 `sun.misc.Unsafe` 的访问，而这就是我们的突破点

在 Java17 （不添加参数）获取 Unsafe 对象的方法如下：

```
import sun.misc.Unsafe;
// 获取Unsafe实例
Field theUnsafe = Unsafe.class.getDeclaredField(”theUnsafe”);
theUnsafe.setAccessible(true);
Unsafe unsafe = (Unsafe) theUnsafe.get(null);
```

正常 JDK 内部类 `setAccessible` 是会抛异常的，不过JDK特意暴露了 `sun.misc` 给 `EVERYONE_MODULE`，这里可以正常运行

2. 再见了，`setAccessible`

众所周知，`setAccessible` 如果成功了，修改的是`java.lang.reflect.AccessibleObject#override`字段，但是由于反射获取的字段，都会经过 `jdk.internal.reflect.Reflection#filterFields` 过滤，很遗憾，`AccessibleObject`就在过滤清单上：

而`sun.misc.Unsafe` 恰恰去掉了在`jdk.internal.misc.Unsafe`中仍保留的`objectFieldOffset(Class<?> c, String name)`接口，只留下了`objectFieldOffset(Field f)`，所以我们不能直接修改它，得绕点路了

搜索 override 地址偏移

通过对两个相同 `Field` 对象（反射获取Field都会拷贝一个新Field），一个设置 `setAccessible(true)`，一个设置 `setAccessible(false)` 然后通过 `Unsafe` 循环读取该对象的内存，直到对比出不一样的部分，我们就得到了`override`字段的内存偏移，而任何子类中父类字段的内存偏移都是一样的

```
private static final int overrideOffset;
public static final Unsafe UNSAFE;

static {
    /*
     * 通过反射获取 Unsafe 实例，这是JDK故意保留的使用方式
     * 通过 Unsafe setAccessible 的 Field 和 未 setAccessible 的 Field 逐一对比获取 override 字段的内存偏移（字段偏移在所有子类型中固定）
     * 通过 override 偏移，即可绕过权限校验强行设置所有 setAccessible
     */
    try {
        Field accessible = Unsafe.class.getDeclaredField("theUnsafe");
        Field notAccessible = Unsafe.class.getDeclaredField("theUnsafe");
        accessible.setAccessible(true);
        notAccessible.setAccessible(false);
```

```
        Unsafe unsafe = (Unsafe) accessible.get(null);
        // override 布尔型字节偏移量。在java17应该是 12
        int i = 0;
        while (unsafe.getBoolean(accessible, i) ==
unsafe.getBoolean(notAccessible, i)) {i++;}
        overrideOffset = i;
        UNSAFE = unsafe;
    } catch (Throwable e) {
        throw Throws.sneakyThrows(e);
    }
}

```

```
import lombok.experimental.UtilityClass;

/**
 * 通过泛型抛任何异常，详见文章《一些Java 泛型使用经验，使用泛型优化接
口设计》最后一段
 */
@UtilityClass
public class Throws {
    @SuppressWarnings("unchecked")
    public static <T extends Throwable> RuntimeException
sneakyThrows(Throwable throwable) throws T {
        throw (T) throwable;
    }
}

```

然后我们就可以对任意`AccessibleObject`调用`setAccessible` 了！

```
@SuppressWarnings({"deprecation", "UnusedReturnValue"})
static <T extends AccessibleObject> T setAccessible(T object) {
    if (object == null) {
        return null;
    }
    if (object.isAccessible()) {
        return object;
    }
    UNSAFE.putBoolean(object, overrideOffset, true);
    return object;
```

}

```

3. `LookUp` 与 `MethodHandle`，让反射重新伟大
==========

`Reflection#filterFields`、`Reflection#filterMethods`过滤了太多东西，导致反射不再能轻易得到JDK内部字段方法了，我们需要绕过过滤

### 构造 `Lookup`

为了减少安全检查，同时提高性能，反射调用方法的最佳方式就是通过`MehodHandle`，而为了获取JDK内部的`MehodHandle`对象，我们需要一个高权限的`MethodHandles.Lookup`对象

```
private static final MethodHandle newLookUp;

static {
    //noinspection SpellCheckingInspection
    try {
        // 通过反射获取 MethodHandles.Lookup 的构造方法
        Constructor<MethodHandles.Lookup> constructor =
MethodHandles.Lookup.class.getDeclaredConstructor(Class.class, Class.
class, int.class);
        setAccessible(constructor);
        // 获取构造方法的 MethodHandle ， MethodHandle 只在获取时检查权
限
        // setAccessible 之后 unreflect 不再检查权限，任意 lookup 均可
        newLookUp =
MethodHandles.lookup().unreflectConstructor(constructor);
    } catch (Throwable e) {
        throw Throws.sneakyThrows(e);
    }
}

//使用 ClassValue 缓存持有 Class 强引用的对象，防止 Class 无法卸载导致
内存泄漏
private static final ClassValue<MethodHandles.Lookup> LOOKUP = new
ClassValue<>() {
    @Override
    @SneakyThrows
```
```

```
    protected MethodHandles.Lookup computeValue(Class<?> type) {
        return (MethodHandles.Lookup) newLookUp.invokeExact(type,
(Class<?>) null, /* Lookup.TRUSTED*/-1);
    }
};

// 最高权限 LOOKUP
static final MethodHandles.Lookup IMPL_LOOKUP =
LOOKUP.get(Object.class);

/**
 * 获取拥有 指定类的最高权限 MethodHandles.Lookup 对象
 */
@SneakyThrows
static MethodHandles.Lookup getLookUp(Class<?> lookupClass) {
    return LOOKUP.get(lookupClass);
}
```

获取任意类的全部字段和方法

`MethodHandle` 通过 `invokeExact` 调用是最快也是类型匹配要求最严格的

```
private static final MethodHandle getDeclaredMethods0;
private static final MethodHandle getDeclaredFields0;
private static final MethodHandle forName0;

static {
    try {
        getDeclaredMethods0 = IMPL_LOOKUP.findSpecial(Class.class,
”getDeclaredMethods0”,
                MethodType.methodType(Method[].class, boolean.class),
Class.class);
        getDeclaredFields0 = IMPL_LOOKUP.findSpecial(Class.class,
”getDeclaredFields0”,
                MethodType.methodType(Field[].class, boolean.class),
Class.class);
        forName0 = IMPL_LOOKUP.findStatic(Class.class, ”forName0”,
                MethodType.methodType(Class.class, String.class,
boolean.class, ClassLoader.class, Class.class));
    } catch (Throwable e) {
        throw Throws.sneakyThrows(e);
    }
```

```
}

@SneakyThrows
static Method[] getDeclaredMethods(Class<?> clazz) {
    return (Method[]) getDeclaredMethods0.invokeExact(clazz, false);
}

@SneakyThrows
static Field[] getDeclaredFields(Class<?> clazz) {
    return (Field[]) getDeclaredFields0.invokeExact(clazz, false);
}

@SneakyThrows
@SuppressWarnings("unchecked")
static <T> Class<T> getClassByName(String className, boolean
initialize, ClassLoader classLoader, Class<?> caller) {
    return (Class<T>) forName0.invokeExact(className, initialize,
classLoader, caller);
}
```

```
至此，我们绕开了 Java17 强封装对反射的影响，但是对JDK内部类的使用还会在类加载阶段进行检查，我们需要彻底消灭`--add-opens`

4. Field 与 Method 的实用工具
=========================

*注：如下代码引用的 `$Unsafe` 类代码均在上述列出，不再赘述*

### `JFields` 工具，读取设置任意字段

**[点击展开/折叠代码块]**

```
import lombok.SneakyThrows;
import lombok.experimental.UtilityClass;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.lang.invoke.VarHandle;
import java.lang.reflect.Field;
```
```

```
import java.lang.reflect.Modifier;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author muyuanjin
 * @since 2024/5/13
 */
@UtilityClass
public class JFields {
    public static Field getField(Class<?> clazz, String name) {
        return getFieldInfo(clazz, name).field;
    }

    public static VarHandle getVarHandle(Class<?> clazz, String name) {
        return getFieldInfo(clazz, name).varHandle;
    }

    public static FieldInfo getFieldInfo(Class<?> clazz, String name) {
        return FIELDS.get(clazz).computeIfAbsent(name, key ->
searchFields(DECLARED_FIELDS.get(clazz), key));
    }

    public static List<Field> getFields(Class<?> clazz) {
        return
Collections.unmodifiableList(Arrays.asList(DECLARED_FIELDS.get(clazz))
);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T getValue(Object target, String name) {
        FieldInfo fieldValue = getFieldInfo(target.getClass(), name);
        if (!fieldValue.isVolatile) {
            return (T) fieldValue.varHandle.get(target);
        }
        return (T) fieldValue.varHandle.getVolatile(target);
    }

    public static void setValue(Object target, String name, Object value) {
        FieldInfo fieldValue = getFieldInfo(target.getClass(), name);
        if (!fieldValue.isVolatile) {
            fieldValue.varHandle.set(target, value);
        }
        fieldValue.varHandle.setVolatile(target, value);
```

```
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T getStaticValue(Class<?> clazz, String name) {
        FieldInfo fieldValue = getFieldInfo(clazz, name);
        if (!fieldValue.isVolatile) {
            return (T) fieldValue.varHandle.get();
        }
        return (T) fieldValue.varHandle.getVolatile();
    }

    public static void setStaticValue(Class<?> clazz, String name, Object
value) {
        FieldInfo fieldValue = getFieldInfo(clazz, name);
        if (!fieldValue.isVolatile) {
            fieldValue.varHandle.set(value);
        }
        fieldValue.varHandle.setVolatile(value);
    }

    @SneakyThrows
    private static FieldInfo searchFields(Field[] fields, String name) {
        if (fields.length != 0) {
            Class<?> declaringClass = fields[0].getDeclaringClass();
            for (Field field : fields) {
                if (field.getName().equals(name)) {
                    return FieldInfo.of(field);
                }
            }
        }
        throw Throws.sneakyThrows(new NoSuchFieldException(name));
    }

    private static final ClassValue<Map<String, FieldInfo>> FIELDS = new
ClassValue<>() {
        @Override
        protected Map<String, FieldInfo> computeValue(Class<?> type) {
            return new ConcurrentHashMap<>();
        }
    };

    private static final MethodHandle copyFields;

    static {
        try {
            copyFields = $Unsafe.IMPL_LOOKUP.findStatic(Class.class,
"copyFields", MethodType.methodType(Field[].class, Field[].class));
```

```
        } catch (Throwable e) {
            throw Throws.sneakyThrows(e);
        }
    }

    private static final ClassValue<Field[]> DECLARED_FIELDS = new
ClassValue<>() {
        @Override
        @SneakyThrows
        @SuppressWarnings("ConfusingArgumentToVarargsMethod")
        protected Field[] computeValue(Class<?> type) {
            Field[] declaredFields = (Field[])
copyFields.invokeExact($Unsafe.getDeclaredFields(type));
            for (Field declaredField : declaredFields) {
                $Unsafe.setAccessible(declaredField);
            }
            return declaredFields;
        }
    };

    public record FieldInfo(Field field, VarHandle varHandle, boolean
isVolatile) {
        private static final MethodHandle makeFieldHandle;
        private static final MethodHandle newMemberName;
        private static final MethodHandle getFieldType;

        static {
            try {
                ClassLoader loader = FieldInfo.class.getClassLoader() == null
? ClassLoader.getSystemClassLoader() :
FieldInfo.class.getClassLoader();
                Class<Object> varHandlesClass =
$Unsafe.getClassByName("java.lang.invoke.VarHandles", true, loader,
MethodHandles.class);
                Class<Object> memberNameClass =
$Unsafe.getClassByName("java.lang.invoke.MemberName", true, loader,
MethodHandles.class);
                makeFieldHandle =
$Unsafe.IMPL_LOOKUP.findStatic(varHandlesClass, "makeFieldHandle",
                        MethodType.methodType(VarHandle.class,
memberNameClass, Class.class, Class.class, boolean.class))
                        .asType(MethodType.methodType(VarHandle.class,
Object.class, Class.class, Class.class, boolean.class));
                newMemberName =
$Unsafe.IMPL_LOOKUP.unreflectConstructor(memberNameClass.getCon
structor(Field.class, boolean.class))
                        .asType(MethodType.methodType(Object.class,
Field.class, boolean.class));
```

```
                getFieldType =
$Unsafe.IMPL_LOOKUP.findVirtual(memberNameClass, "getFieldType",
MethodType.methodType(Class.class))
                        .asType(MethodType.methodType(Class.class,
Object.class));
            } catch (Throwable e) {
                throw Throws.sneakyThrows(e);
            }
        }

        @SneakyThrows
        public static FieldInfo of(Field field) {
            Class<?> clazz = field.getDeclaringClass();
            Object memberName = newMemberName.invokeExact(field,
false);
            // 绕过 trustedFinal 检查
            VarHandle handle = (VarHandle)
makeFieldHandle.invokeExact(memberName, clazz, (Class<?>)
getFieldType.invokeExact(memberName), true);
            return new FieldInfo(field, handle,
Modifier.isVolatile(field.getModifiers()));
        }
    }
}
```

`JMethods` 工具，读取调用任意方法

[点击展开/折叠代码块]

```
import lombok.SneakyThrows;
import lombok.experimental.UtilityClass;

import java.lang.invoke.MethodHandle;
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author muyuanjin
 * @since 2024/5/13
 */
```

```
@UtilityClass
public class JMethods {
    /**
     * 获取方法
     *
     * @param targetClass    目标类
     * @param methodName     方法名
     * @param parameterTypes 参数类型
     * @return 方法
     */
    public static Method getMethod(Class<?> targetClass, String
methodName, Class<?>... parameterTypes) {
        return getMethodValue(targetClass, methodName,
parameterTypes).method;
    }

    @SneakyThrows
    public static List<Method> getMethods(Class<?> targetClass) {
        return
Collections.unmodifiableList(Arrays.asList(DECLARED_METHODS.get(tar
getClass)));
    }

    /**
     * 获取方法句柄
     *
     * @param targetClass    目标类
     * @param methodName     方法名
     * @param parameterTypes 参数类型
     * @return 方法句柄
     */
    public static MethodHandle getMethodHandle(Class<?> targetClass,
String methodName, Class<?>... parameterTypes) {
        return getMethodValue(targetClass, methodName,
parameterTypes).methodHandle;
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invokeStatic(Class<?> targetClass, String
methodName, Class<?>[] parameterTypes, Object... args) {
        return (T) getMethodHandle(targetClass, methodName,
parameterTypes).invokeWithArguments(args);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invokeStatic(Class<?> targetClass, String
```

```
methodName, Object... args) {
        if (args.length == 0) {
            return invokeStatic(targetClass, methodName,
EMPTY_CLASS_ARRAY, EMPTY_OBJECT_ARRAY);
        }
        return (T) findMethodValue(targetClass, methodName,
args).methodHandle.invokeWithArguments(args);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName,
Class<?>[] parameterTypes, Object... args) {
        Object[] vars = new Object[args.length + 1];
        vars[0] = target;
        System.arraycopy(args, 0, vars, 1, args.length);
        return (T) getMethodHandle(target.getClass(), methodName,
parameterTypes).invokeWithArguments(vars);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName) {
        return (T) getMethodHandle(target.getClass(),
methodName).invokeWithArguments(target);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName,
Class<?> parameterType, Object arg1) {
        return (T) getMethodHandle(target.getClass(), methodName,
parameterType).invokeWithArguments(target, arg1);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName,
Class<?>[] parameterTypes, Object arg1, Object arg2) {
        return (T) getMethodHandle(target.getClass(), methodName,
parameterTypes).invokeWithArguments(target, arg1, arg2);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName,
Class<?>[] parameterTypes, Object arg1, Object arg2, Object arg3) {
        return (T) getMethodHandle(target.getClass(), methodName,
```

```
parameterTypes).invokeWithArguments(target, arg1, arg2, arg3);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName,
Class<?>[] parameterTypes, Object arg1, Object arg2, Object arg3,
Object arg4) {
        return (T) getMethodHandle(target.getClass(), methodName,
parameterTypes).invokeWithArguments(target, arg1, arg2, arg3, arg4);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName,
Class<?>[] parameterTypes, Object arg1, Object arg2, Object arg3,
Object arg4, Object arg5) {
        return (T) getMethodHandle(target.getClass(), methodName,
parameterTypes).invokeWithArguments(target, arg1, arg2, arg3, arg4,
arg5);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName, Object
arg1) {
        return (T) findMethodValue(target.getClass(), methodName,
arg1).methodHandle.invokeWithArguments(target, arg1);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName, Object
arg1, Object arg2) {
        return (T) findMethodValue(target.getClass(), methodName, arg1,
arg2).methodHandle.invokeWithArguments(target, arg1, arg2);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName, Object
arg1, Object arg2, Object arg3) {
        return (T) findMethodValue(target.getClass(), methodName, arg1,
arg2, arg3).methodHandle.invokeWithArguments(target, arg1, arg2,
arg3);
    }

    @SneakyThrows
```

```
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName, Object
arg1, Object arg2, Object arg3, Object arg4) {
        return (T) findMethodValue(target.getClass(), methodName, arg1,
arg2, arg3, arg4).methodHandle.invokeWithArguments(target, arg1, arg2,
arg3, arg4);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName, Object
arg1, Object arg2, Object arg3, Object arg4, Object arg5) {
        return (T) findMethodValue(target.getClass(), methodName, arg1,
arg2, arg3, arg4, arg5).methodHandle.invokeWithArguments(target, arg1,
arg2, arg3, arg4, arg5);
    }

    @SneakyThrows
    @SuppressWarnings("unchecked")
    public static <T> T invoke(Object target, String methodName,
Object... args) {
        if (args.length == 0) {
            return invoke(target, methodName, EMPTY_CLASS_ARRAY,
EMPTY_OBJECT_ARRAY);
        }
        Object[] vars = new Object[args.length + 1];
        vars[0] = target;
        System.arraycopy(args, 0, vars, 1, args.length);
        return (T) findMethodValue(target.getClass(), methodName,
args).methodHandle.invokeWithArguments(vars);
    }

    private static Method findMethod(Class<?> targetClass, String
methodName, Object... args) {
        Method[] declaredMethods =
DECLARED_METHODS.get(targetClass);
        List<Method> methods = new ArrayList<>();
        for (Method declaredMethod : declaredMethods) {
            if (declaredMethod.getName().equals(methodName)) {
                methods.add(declaredMethod);
            }
        }
        if (methods.size() == 1) {
            return methods.get(0);
        }
        out:
        for (Method method : methods) {
            Class<?>[] parameterTypes = method.getParameterTypes();
```

```
            if (parameterTypes.length == args.length) {
                for (int i = 0; i < parameterTypes.length; i++) {
                    if (!parameterTypes[i].isInstance(args[i])) {
                        continue out;
                    }
                }
                return method;
            } else {
                Parameter[] parameters = method.getParameters();
                if (parameters[parameters.length - 1].isVarArgs()) {
                    for (int i = 0; i < parameters.length - 1; i++) {
                        if (!parameterTypes[i].isInstance(args[i])) {
                            continue out;
                        }
                    }
                    if (args.length == parameters.length - 1) {
                        return method;
                    }
                    Class<?> componentType =
parameterTypes[parameters.length - 1].getComponentType();
                    for (int i = parameters.length - 1; i < args.length; i++) {
                        if (!componentType.isInstance(args[i])) {
                            continue out;
                        }
                    }
                    return method;
                }
            }
        }
        throw Throws.sneakyThrows(new
NoSuchMethodError(methodName + " " + Arrays.toString(args)));
    }

    private static MethodValue findMethodValue(Class<?> targetClass,
String methodName, Object... args) {
        MethodKey key = new MethodKey(methodName,
getParameterNames(args));
        return METHODS.get(targetClass).computeIfAbsent(key, k -> {
            try {
                Method method = findMethod(targetClass, methodName,
args);
                return new MethodValue(method,
$Unsafe.IMPL_LOOKUP.unreflect(method));
            } catch (Throwable e) {
                throw Throws.sneakyThrows(e);
            }
        });
    }
```

```
    private static MethodValue getMethodValue(Class<?> targetClass,
String methodName, Class<?>[] parameterTypes) {
        MethodKey key = new MethodKey(methodName,
getParameterTypeNames(parameterTypes));
        return METHODS.get(targetClass).computeIfAbsent(key, k -> {
            try {
                Method method = (Method)
searchMethods.invokeExact(DECLARED_METHODS.get(targetClass),
methodName, parameterTypes);
                if (method == null) {
                    throw new NoSuchMethodException(methodName + " " +
Arrays.toString(parameterTypes));
                }
                return new MethodValue(method,
$Unsafe.IMPL_LOOKUP.unreflect(method));
            } catch (Throwable e) {
                throw Throws.sneakyThrows(e);
            }
        });
    }

    private static List<String> getParameterNames(Object... args) {
        String[] names = new String[args.length];
        for (int i = 0; i < args.length; i++) {
            Object arg = args[i];
            names[i] = arg == null ? "null" : arg.getClass().getName();
        }
        return Arrays.asList(names);
    }

    private static List<String> getParameterTypeNames(Class<?>[]
parameterTypes) {
        String[] names = new String[parameterTypes.length];
        for (int i = 0; i < parameterTypes.length; i++) {
            names[i] = parameterTypes[i].getName();
        }
        return Arrays.asList(names);
    }

    private static final Class<?>[] EMPTY_CLASS_ARRAY = new Class[0];
    private static final Object[] EMPTY_OBJECT_ARRAY = new Object[0];

    private static final MethodHandle copyMethods;
    private static final MethodHandle searchMethods;

    static {
        try {
```

```
            copyMethods = $Unsafe.IMPL_LOOKUP.findStatic(Class.class,
"copyMethods", MethodType.methodType(Method[].class,
Method[].class));
            searchMethods = $Unsafe.IMPL_LOOKUP.findStatic(Class.class,
"searchMethods", MethodType.methodType(Method.class,
Method[].class, String.class, Class[].class));
        } catch (Throwable e) {
            throw Throws.sneakyThrows(e);
        }
    }

    private static final ClassValue<Method[]> DECLARED_METHODS =
new ClassValue<>() {
        @Override
        @SneakyThrows
        @SuppressWarnings("ConfusingArgumentToVarargsMethod")
        protected Method[] computeValue(Class<?> type) {
            Method[] declaredMethods = (Method[])
copyMethods.invokeExact($Unsafe.getDeclaredMethods(type));
            for (Method declaredMethod : declaredMethods) {
                $Unsafe.setAccessible(declaredMethod);
            }
            return declaredMethods;
        }
    };

    private static final ClassValue<Map<MethodKey, MethodValue>>
METHODS = new ClassValue<>() {
        @Override
        protected Map<JMethods.MethodKey, MethodValue>
computeValue(Class<?> type) {
            return new ConcurrentHashMap<>();
        }
    };

    private record MethodKey(String method, List<String>
parameterTypes) {}

    private record MethodValue(Method method, MethodHandle
methodHandle) {}
}

```

5. 有了 `exportAllToAll`，再也不用`--add-opens`了
==========

下面就是通过反射修改模块信息了（从 [Burningwave Core](http://cxyroad.com/ ”https://github.com/burningwave/core”) ~~抄~~ 搬运过来的）

`JModules` 工具，任意模块 export to 任意模块

[点击展开/折叠代码块]

```
/*
 * This file is part of Burningwave Core.
 *
 * Author: Roberto Gentili
 *
 * Hosted at: https://github.com/burningwave/core
 *
 * --
 *
 * The MIT License (MIT)
 *
 * Copyright (c) 2019 Roberto Gentili
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated
 * documentation files (the ”Software”), to deal in the Software without restriction, including without
 * limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
 * the Software, and to permit persons to whom the Software is furnished to do so, subject to the following
 * conditions:
 *
 * The above copyright notice and this permission notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN
 * AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
```

```
 * OR OTHER DEALINGS IN THE SOFTWARE.
 */
import lombok.SneakyThrows;
import lombok.experimental.UtilityClass;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import java.util.stream.Stream;

/*
 * Copyright (c) Burningwave Core. and/or Roberto Gentili.
 * The original file is org.burningwave.core.classes.Modules
 * Modifications made by muyuanjin on 2024/5/14.
 */
@UtilityClass
@SuppressWarnings("DuplicatedCode")
public class JModules {
    private static final Class<?> moduleClass;
    private static final Set<?> allSet;
    private static final Set<?> everyOneSet;
    private static final Set<?> allUnnamedSet;
    private static final Map<String, ?> nameToModule;

    static {
        try {
            ClassLoader loader = JModules.class.getClassLoader() == null ?
ClassLoader.getSystemClassLoader() : JModules.class.getClassLoader();
            moduleClass = $Unsafe.getClassByName("java.lang.Module",
false, loader, JModules.class);
            Class<?> moduleLayerClass =
$Unsafe.getClassByName("java.lang.ModuleLayer", false, loader,
JModules.class);
            Object moduleLayer = JMethods.invokeStatic(moduleLayerClass,
"boot");
            nameToModule = JFields.getValue(moduleLayer,
"nameToModule");
            allSet = new HashSet<>();
            allSet.add(JFields.getStaticValue(moduleClass,
"ALL_UNNAMED_MODULE"));
            allSet.add(JFields.getStaticValue(moduleClass,
"EVERYONE_MODULE"));
            everyOneSet = new HashSet<>();
            everyOneSet.add(JFields.getStaticValue(moduleClass,
"EVERYONE_MODULE"));
            allUnnamedSet = new HashSet<>();
            allUnnamedSet.add(JFields.getStaticValue(moduleClass,
```

```
"ALL_UNNAMED_MODULE"));
        } catch (Throwable e) {
            throw Throws.sneakyThrows(e);
        }
    }

    private static volatile boolean initialized = false;

    @SneakyThrows
    public static synchronized void makeSureExported() {
        if (!initialized) {
            exportAllToAll();
            initialized = true;
        }
    }

    public static synchronized void exportAllToAll() {
        try {
            nameToModule.forEach((name, module) ->
(JMethods.<Set<String>>invoke(module,
"getPackages")).forEach(pkgName -> {
                exportToAll("exportedPackages", module, pkgName);
                exportToAll("openPackages", module, pkgName);
            }));
        } catch (Throwable e) {
            throw Throws.sneakyThrows(e);
        }
    }


    public static synchronized void exportToAllUnnamed(String name) {
        exportTo(name, JModules::exportToAllUnnamed);
    }

    public static synchronized void exportToAll(String name) {
        exportTo(name, JModules::exportToAll);
    }

    public static synchronized void exportPackage(String
moduleFromName, String moduleToName, String... packageNames) {
        Object moduleFrom = checkAndGetModule(moduleFromName);
        Object moduleTo = checkAndGetModule(moduleToName);
        exportPackage(moduleFrom, moduleTo, packageNames);
    }


    public static synchronized void exportPackageToAll(String
moduleFromName, String... packageNames) {
```

```
        Object moduleFrom = checkAndGetModule(moduleFromName);
        exportPackage(moduleFrom, everyOneSet.iterator().next(),
packageNames);
    }


    public static synchronized void exportPackageToAllUnnamed(String
moduleFromName, String... packageNames) {
        Object moduleFrom = checkAndGetModule(moduleFromName);
        exportPackage(moduleFrom, allUnnamedSet.iterator().next(),
packageNames);
    }


    public static synchronized void export(String moduleFromName,
String moduleToName) {
        try {
            Object moduleFrom = checkAndGetModule(moduleFromName);
            Object moduleTo = checkAndGetModule(moduleToName);
            (JMethods.<Set<String>>invoke(moduleFrom,
"getPackages")).forEach(pkgName -> {
                export("exportedPackages", moduleFrom, pkgName,
moduleTo);
                export("openPackages", moduleFrom, pkgName, moduleTo);
            });
        } catch (Throwable e) {
            throw Throws.sneakyThrows(e);
        }
    }


    static void exportPackage(Object moduleFrom, Object moduleTo,
String... packageNames) {
        Set<String> modulePackages = JMethods.invoke(moduleFrom,
"getPackages");
        Stream.of(packageNames).forEach(pkgName -> {
            if (!modulePackages.contains(pkgName)) {
                throw new PackageNotFoundException("Package " +
pkgName + " not found in module " + JFields.getValue(moduleFrom,
"name"));
            }
            export("exportedPackages", moduleFrom, pkgName, moduleTo);
            export("openPackages", moduleFrom, pkgName, moduleTo);
        });
    }

    static Object checkAndGetModule(String name) {
        Object module = nameToModule.get(name);
```

```
        if (module == null) {
            throw new NotFoundException("Module named name " + name
+ " not found");
        }
        return module;
    }

    static void exportTo(String name, ThrConsumer<String, Object,
String> exporter) {
        try {
            Object module = checkAndGetModule(name);
            (JMethods.<Set<String>>invoke(module,
"getPackages")).forEach(pkgName -> {
                exporter.accept("exportedPackages", module, pkgName);
                exporter.accept("openPackages", module, pkgName);
            });
        } catch (Throwable e) {
            throw Throws.sneakyThrows(e);
        }
    }


    static void exportToAll(String fieldName, Object module, String
pkgName) {
        Map<String, Set<?>> pckgForModule = JFields.getValue(module,
fieldName);
        if (pckgForModule == null) {
            pckgForModule = new HashMap<>();
            JFields.setValue(module, fieldName, pckgForModule);
        }
        pckgForModule.put(pkgName, allSet);
        if (fieldName.startsWith("exported")) {
            JMethods.invokeStatic(moduleClass, "addExportsToAll0",
module, pkgName);
        }
    }


    static void exportToAllUnnamed(String fieldName, Object module,
String pkgName) {
        Map<String, Set<?>> pckgForModule = JFields.getValue(module,
fieldName);
        if (pckgForModule == null) {
            pckgForModule = new HashMap<>();
            JFields.setValue(module, fieldName, pckgForModule);
        }
        pckgForModule.put(pkgName, allUnnamedSet);
        if (fieldName.startsWith("exported")) {
```

```
            JMethods.invokeStatic(moduleClass,
"addExportsToAllUnnamed0", module, pkgName);
        }
    }

    static void export(String fieldName, Object moduleFrom, String
pkgName, Object moduleTo) {
        Map<String, Set<Object>> pckgForModule =
JFields.getValue(moduleFrom, fieldName);
        if (pckgForModule == null) {
            pckgForModule = new HashMap<>();
            JFields.setValue(moduleFrom, fieldName, pckgForModule);
        }
        Set<Object> moduleSet = pckgForModule.get(pkgName);
        if (!(moduleSet instanceof HashSet)) {
            if (moduleSet != null) {
                moduleSet = new HashSet<>(moduleSet);
            } else {
                moduleSet = new HashSet<>();
            }
            pckgForModule.put(pkgName, moduleSet);
        }
        moduleSet.add(moduleTo);
        if (fieldName.startsWith("exported")) {
            JMethods.invokeStatic(moduleClass, "addExports0",
moduleFrom, pkgName, moduleTo);
        }
    }

    @FunctionalInterface
    interface ThrConsumer<T, U, R> {
        void accept(T t, U u, R r);
    }

    public static class NotFoundException extends RuntimeException {
        public NotFoundException(String message) {
            super(message);
        }
    }

    public static class PackageNotFoundException extends
RuntimeException {
        public PackageNotFoundException(String message) {
            super(message);
        }
    }
}
```

```

6. 使用方法
=======

只要在使用了JDK内部类的工具调用/加载，确保调用过`JModules#makeSureExported` 方法即可
对于没有继承内部类，只是使用来说，可以在工具类开头加上一行即可，对于继承内部类的类，只能封装好，确保不对外暴露并且在即将加载该类前调用`makeSureExported`

```
public class XXXUtil {
    static {JModules.makeSureExported();}
```

对于Spring应用程序，可以在已有的`EnvironmentPostProcessor`或者实现一个空的`EnvironmentPostProcessor` 去调用`makeSureExported`，`EnvironmentPostProcessor`在Spring中会是最早加载的一批类，基本不用担心程序运行时的`--add-opens`问题了

对于测试方法，如果遇到`IllegalAccessException`(如果工具封装的好就不会)，显式调用一次`makeSureExported`即可

这样就完美突破了java17的（运行时）强封装，只要编译过得去（`--add-exports`还是跑不了，好在只在POM中配一下就好了），组件的任何黑科技代码都可以顺利运行而不需要引用的程序添加A参数B参数的了！

原文链接: https://juejin.cn/post/7368740273788616731
```