1. 说说你对 Java 反射的理解（源码）

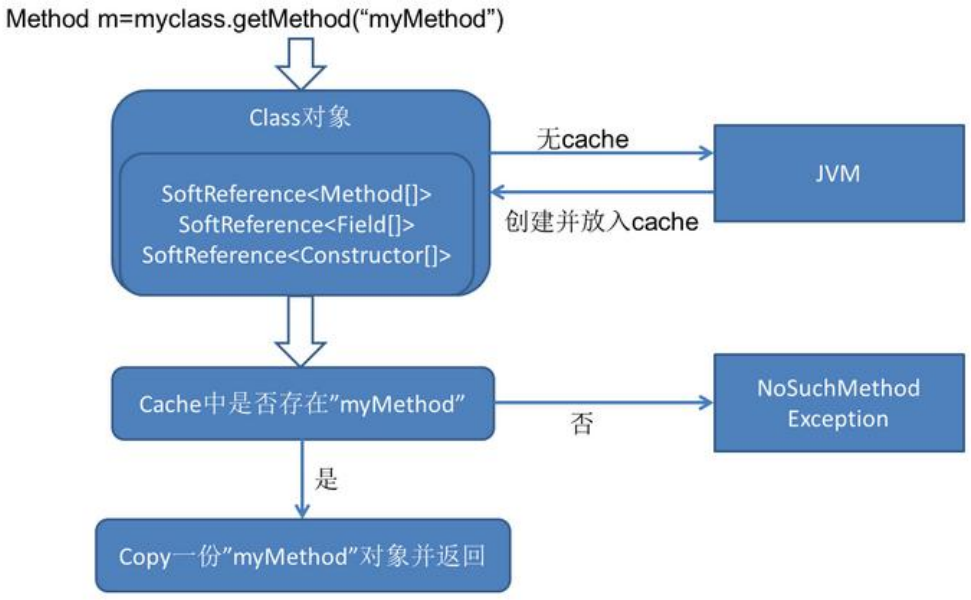
https://www.jianshu.com/p/6277c1f9f48d
https://www.jianshu.com/p/1a21a9cb5bea
http://www.importnew.com/23902.html

```
public class ReflectCase {
    public static void main(String[] args) throws Exception {
        Proxy target = new Proxy();
        Method method = Proxy.class.getDeclaredMethod("run");
        method.invoke(target);
    }

    static class Proxy {
        public void run() {
            System.out.println("run");
        }
    }
}
```

主要有两部分：Method 的获取、Method 的使用

一、Method 获取



我们会调用 getDeclaredMethod（）方法

```java
@CallerSensitive
public Method getDeclaredMethod(String name, Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException {
    // be very careful not to change the stack depth of this
    // checkMemberAccess call for security reasons
    // see java.lang.SecurityManager.checkMemberAccess
    checkMemberAccess(Member.DECLARED, Reflection.getCallerClass(), true);
    Method method = searchMethods(privateGetDeclaredMethods(false), name, parameterTypes);
    if (method == null) {
        throw new NoSuchMethodException(getName() + "." + name + argumentTypesToString(parameterTypes));
    }
    return method;
}
```

其中 privateGetDeclaredMethods 方法从缓存或 JVM 中获取该 Class 中申明的方法列表，searchMethods 方法将从返回的方法列表里找到一个匹配名称和参数的方法对象。

（1）searchMethods（从方法列表中找到相应的对象）

```java
private static Method searchMethods(Method[] methods,
                                    String name,
                                    Class<?>[] parameterTypes)
{
    Method res = null;
    String internedName = name.intern();
    for (int i = 0; i < methods.length; i++) {
        Method m = methods[i];
        if (m.getName() == internedName
            && arrayContentsEq(parameterTypes, m.getParameterTypes())
            && (res == null
                || res.getReturnType().isAssignableFrom(m.getReturnType())))
            res = m;
    }

    return (res == null ? res : getReflectionFactory().copyMethod(res));
}
```

如果找到一个匹配的 Method，则重新 copy 一份返回，即 Method.copy()方法

```
/**
 * Package-private routine (exposed to java.lang.Class via
 * ReflectAccess) which returns a copy of this Method. The copy's
 * "root" field points to this Method.
 */
Method copy() {
    // This routine enables sharing of MethodAccessor objects
    // among Method objects which refer to the same underlying
    // method in the VM. (All of this contortion is only necessary
    // because of the "accessibility" bit in AccessibleObject,
    // which implicitly requires that new java.lang.reflect
    // objects be fabricated for each reflective call on Class
    // objects.)
    Method res = new Method(clazz, name, parameterTypes, returnType,
                            exceptionTypes, modifiers, slot, signature,
                            annotations, parameterAnnotations, annotationDefault);
    res.root = this;
    // Might as well eagerly propagate this if already present
    res.methodAccessor = methodAccessor;
    return res;
}
```

所次每次调用 getDeclaredMethod 方法返回的 Method 对象其实都是一个新的对象，且新对象的 root 属性都指向原来的 Method 对象（如果需要频繁调用，最好把 Method 对象缓存起来）。

（2）privateGetDeclaredMethods（获取该 class 中声明的方法列表）

从缓存或 JVM 中获取该 Class 中申明的方法列表，实现如下：

```
// Returns an array of "root" methods. These Method objects must NOT
// be propagated to the outside world, but must instead be copied
// via ReflectionFactory.copyMethod.
private Method[] privateGetDeclaredMethods(boolean publicOnly) {
    checkInitted();
    Method[] res;
    ReflectionData<T> rd = reflectionData();
    if (rd != null) {
        res = publicOnly ? rd.declaredPublicMethods : rd.declaredMethods;
        if (res != null) return res;
    }
    // No cached value available; request value from VM
    res = Reflection.filterMethods(this, getDeclaredMethods0(publicOnly));
    if (rd != null) {
        if (publicOnly) {
            rd.declaredPublicMethods = res;
        } else {
            rd.declaredMethods = res;
        }
    }
    return res;
}
```

先利用 reflectionData（）方法从缓存中获取该 Class 中声明的方法列表，如果有就直接返回，不用执行下一步骤。

如果从缓存中找不到，则会利用 getDeclaredMethods0（）方法从 JVM 中获取该类中的所有方法，然后找出特定方法

```
private native Method[]      getDeclaredMethods0(boolean publicOnly);
```

（1）从缓存中获取：

先利用 reflectionData（）方法从缓存中获取该 Class 中声明的方法列表。

其中 reflectionData()方法实现如下：

```
// Lazily create and cache ReflectionData
private ReflectionData<T> reflectionData() {
    SoftReference<ReflectionData<T>> reflectionData = this.reflectionData;
    int classRedefinedCount = this.classRedefinedCount;
    ReflectionData<T> rd;
    if (useCaches &&
        reflectionData != null &&
        (rd = reflectionData.get()) != null &&
        rd.redefinedCount == classRedefinedCount) {
        return rd;
    }
    // else no SoftReference or cleared SoftReference or stale ReflectionData
    // -> create and replace new instance
    return newReflectionData(reflectionData, classRedefinedCount);
}
```

这里有个比较重要的数据结构 ReflectionData，用来缓存从 JVM 中读取类的如下属性数据：

```
// reflection data that might get invalidated when JVM TI RedefineClasses() is called
static class ReflectionData<T> {
    volatile Field[] declaredFields;
    volatile Field[] publicFields;
    volatile Method[] declaredMethods;
    volatile Method[] publicMethods;
    volatile Constructor<T>[] declaredConstructors;
    volatile Constructor<T>[] publicConstructors;
    // Intermediate results for getFields and getMethods
    volatile Field[] declaredPublicFields;
    volatile Method[] declaredPublicMethods;
    // Value of classRedefinedCount when we created this ReflectionData instance
    final int redefinedCount;

    ReflectionData(int redefinedCount) { this.redefinedCount = redefinedCount; }
}
```

从 reflectionData()方法实现可以看出：

ReflectionData **对象是** SoftReference **类型的，**说明在内存紧张时可能会被回收（不过也可以通过-XX:SoftRefLRUPolicyMSPerMB 参数控制回收的时机，只要发生 GC 就会将其回收）。

如果 ReflectionData 被回收之后，又执行了反射方法，那只能通过 newReflectionData（）方法重新创建一个这样的对象了。newReflectionData（）方法实现如下：

```
private ReflectionData<T> newReflectionData(SoftReference<ReflectionData<T>> oldReflectionData,
                                            int classRedefinedCount) {
    if (!useCaches) return null;

    while (true) {
        ReflectionData<T> rd = new ReflectionData<>(classRedefinedCount);
        // try to CAS it...
        if (Atomic.casReflectionData(this, oldReflectionData, new SoftReference<>(rd))) {
            return rd;
        }
        // else retry
        oldReflectionData = this.reflectionData;
        classRedefinedCount = this.classRedefinedCount;
        if (oldReflectionData != null &&
            (rd = oldReflectionData.get()) != null &&
            rd.redefinedCount == classRedefinedCount) {
            return rd;
        }
    }
}
```

通过 unsafe.compareAndSwapObject 方法重新设置 reflectionData 字段；

如果通过 reflectionData() 获得的 ReflectionData 对象不为空，则尝试从 ReflectionData 对象中获取 declaredMethods 属性。如果是第一次，或则被 GC 回收之后，重新初始化后的类属性为空，则需要重新到 JVM 中获取一次，并赋值给 ReflectionData，下次调用就可以使用缓存数据了。

（2）从 JVM 中查找

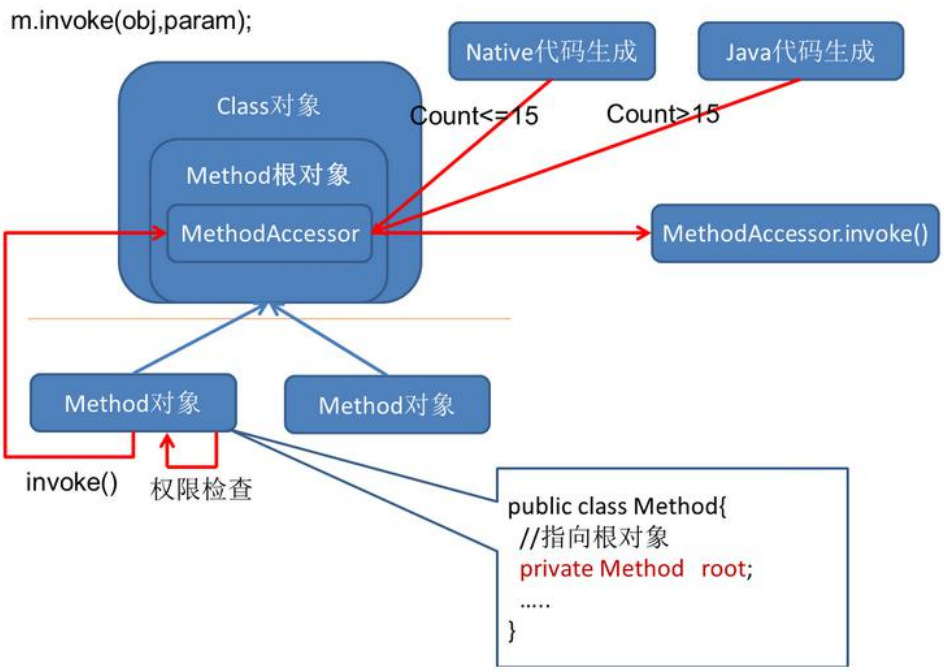
如果从缓存中找不到，则会利用 getDeclaredMethods0（）方法从 JVM 中获取该类中的所有方法

```
private native Method[]      getDeclaredMethods0(boolean publicOnly);
```

然后调用 Reflection.filterMethods（）方法找出特定方法

```
res = Reflection.filterMethods(this, getDeclaredMethods0(publicOnly));
```

二、Method 调用



获取到指定的方法对象 Method 之后，就可以调用它的 invoke 方法了，invoke 实现如下：

```
@CallerSensitive
public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
       InvocationTargetException
{
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            // Until there is hotspot @CallerSensitive support
            // can't call Reflection.getCallerClass() here
            // Workaround for now: add a frame getCallerClass to
            // make the caller at stack depth 2
            Class<?> caller = getCallerClass();
            checkAccess(caller, clazz, obj, modifiers);
        }
    }
    MethodAccessor ma = methodAccessor;             // read volatile
    if (ma == null) {
        ma = acquireMethodAccessor();
    }
    return ma.invoke(obj, args);
}
```

主要是 MethodAccessor 对象中的 invoke（）方法调用

（1）获取 MethodAccessor 对象：一开始 methodAccessor 为空，需要调用 acquireMethodAccessor 生成一个新的 MethodAccessor 对象。

MethodAccessor 本身就是一个接口，实现如下：

```
public interface MethodAccessor {
    Object invoke(Object var1, Object[] var2) throws IllegalArgumentException, InvocationTargetException;
}
```

acquireMethodAccessor()方法：

在 acquireMethodAccessor 方法中，会通过 ReflectionFactory 类的 newMethodAccessor()方法创建一个实现了 MethodAccessor 接口的对象，实现如下：

```
public MethodAccessor newMethodAccessor(Method method) {
    checkInitted();

    if (noInflation) {
        return new MethodAccessorGenerator().
            generateMethod(method.getDeclaringClass(),
                           method.getName(),
                           method.getParameterTypes(),
                           method.getReturnType(),
                           method.getExceptionTypes(),
                           method.getModifiers());
    } else {
        NativeMethodAccessorImpl acc =
            new NativeMethodAccessorImpl(method);
        DelegatingMethodAccessorImpl res =
            new DelegatingMethodAccessorImpl(acc);
        acc.setParent(res);
        return res;
    }
}
```

(在 ReflectionFactory 类中，有 2 个重要的字段：

noInflation(默认 false)和 inflationThreshold(默认 15)

在 checkInitted 方法中可以通过

-Dsun.reflect.inflationThreshold=xxx

和-Dsun.reflect.noInflation=true

对这两个字段重新设置，而且只会设置一次；）

如果 noInflation 为 false，方法 newMethodAccessor 都会返回 DelegatingMethodAccessorImpl 对象

注：

DelegatingMethodAccessorImpl 的类实现：

```
class DelegatingMethodAccessorImpl extends MethodAccessorImpl {
    private MethodAccessorImpl delegate;

    DelegatingMethodAccessorImpl(MethodAccessorImpl delegate) {
        setDelegate(delegate);
    }

    public Object invoke(Object obj, Object[] args)
        throws IllegalArgumentException, InvocationTargetException
    {
        return delegate.invoke(obj, args);
    }

    void setDelegate(MethodAccessorImpl delegate) {
        this.delegate = delegate;
    }
}
```

其实，DelegatingMethodAccessorImpl 对象就是一个代理对象，负责调用被代理对象 delegate 的 invoke 方法，其中 delegate 参数目前是 NativeMethodAccessorImpl 对象。

由于 DelegatingMethodAccessorImpl 对象内部实现是一个代理对象，被代理的对象是 NativeMethodAccessorImpl 对象。所以最终

Method 的 invoke 方法调用的是 NativeMethodAccessorImpl 对象 invoke 方法。

NativeMethodAccessorImpl 对象 invoke 方法实现如下：

```
class NativeMethodAccessorImpl extends MethodAccessorImpl {
    private Method method;
    private DelegatingMethodAccessorImpl parent;
    private int numInvocations;

    NativeMethodAccessorImpl(Method method) {
        this.method = method;
    }

    public Object invoke(Object obj, Object[] args)
        throws IllegalArgumentException, InvocationTargetException
    {
        if (++numInvocations > ReflectionFactory.inflationThreshold()) {
            MethodAccessorImpl acc = (MethodAccessorImpl)
                new MethodAccessorGenerator().
                    generateMethod(method.getDeclaringClass(),
                                   method.getName(),
                                   method.getParameterTypes(),
                                   method.getReturnType(),
                                   method.getExceptionTypes(),
                                   method.getModifiers());
            parent.setDelegate(acc);
        }

        return invoke0(method, obj, args);
    }

    void setParent(DelegatingMethodAccessorImpl parent) {
        this.parent = parent;
    }

    private static native Object invoke0(Method m, Object obj, Object[] args);
}
```

这里用到了 ReflectionFactory 类中的 inflationThreshold，

创建机制采用了一种名为 inflation 的方式（JDK1.4 之后）：

（1）调用次数小于等于 15 次：如果该方法的累计调 0.用次数<=15，会创建出 NativeMethodAccessorImpl，它的实现就是直接调用 native 方法实现反射；

（2）调用次数大于 15：如果该方法的累计调用次数>15，会由 java

代码创建出字节码组装而成的 MethodAccessorImpl。（改变 DelegatingMethodAccessorImpl 类中的代理对象）

这里需要注意的是：

MethodAccessorGenerator#generateMethod（）方法在生成 MethodAccessorImpl 对象时，会在内存中生成对应的字节码，并调用 ClassDefiner.defineClass 创建对应的 class 对象，实现如下：

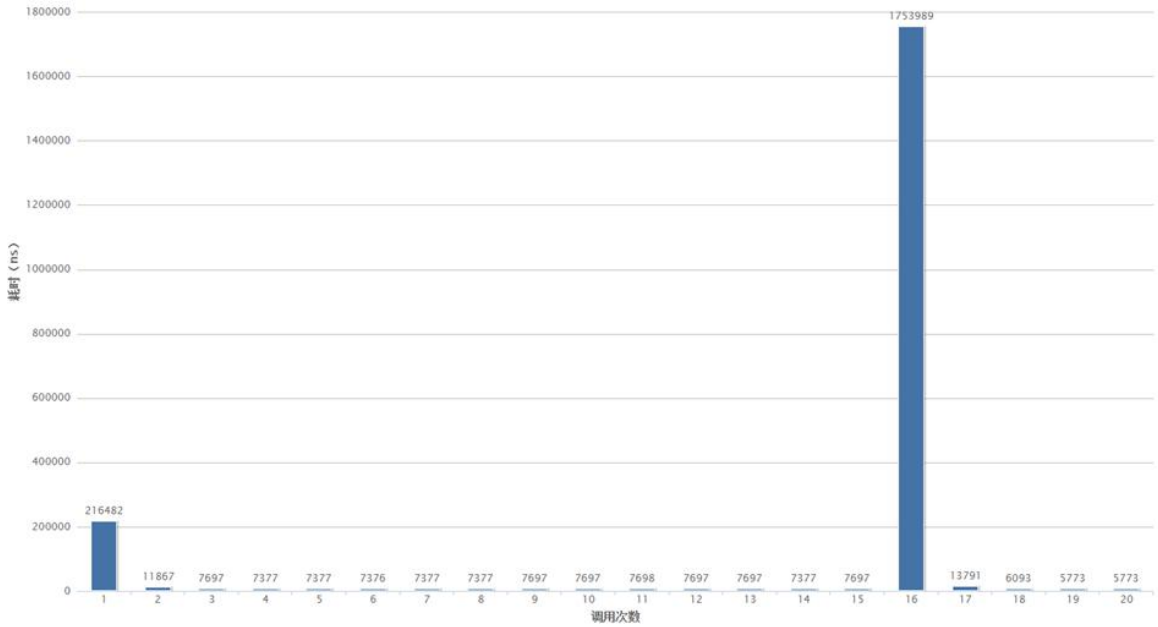
```
return AccessController.doPrivileged(
    new PrivilegedAction<MagicAccessorImpl>() {
        public MagicAccessorImpl run() {
                try {
                return (MagicAccessorImpl)
                ClassDefiner.defineClass
                        (generatedName,
                         bytes,
                         0,
                         bytes.length,
                         declaringClass.getClassLoader()).newInstance();
                } catch (InstantiationException e) {
                    throw (InternalError)
                        new InternalError().initCause(e);
                } catch (IllegalAccessException e) {
                    throw (InternalError)
                        new InternalError().initCause(e);
                }
            }
        });
```

在 ClassDefiner.defineClass 方法实现中，每被调用一次都会生成一个 DelegatingClassLoader 类加载器对象

```
static Class defineClass(String name, byte[] bytes, int off, int len,
                         final ClassLoader parentClassLoader)
{
    ClassLoader newLoader = AccessController.doPrivileged(
        new PrivilegedAction<ClassLoader>() {
            public ClassLoader run() {
                return new DelegatingClassLoader(parentClassLoader);
            }
        });
    return unsafe.defineClass(name, bytes, off, len, newLoader, null);
}
```

这里每次都生成新的类加载器，是为了性能考虑，在某些情况下可以卸载这些生成的类，因为类的卸载是只有在类加载器可以被回收

的情况下才会被回收的，如果用了原来的类加载器，那可能导致这些新创建的类一直无法被卸载，从其设计来看本身就不希望这些类一直存在内存里的，在需要的时候有就行了。



从变化趋势上看，第 1 次和第 16 次调用是最耗时的（初始化 NativeMethodAccessorImpl 和字节码拼装 MethodAccessorImpl）。毕竟初始化是不可避免的，而 native 方式的初始化会更快，因此前几次的调用会采用 native 方法。

随着调用次数的增加，每次反射都使用 JNI 跨越 native 边界会对优化有阻碍作用，相对来说使用拼装出的字节码可以直接以 Java 调用的形式实现反射，发挥了 JIT 优化的作用，避免了 JNI 为了维护 OopMap（HotSpot 用来实现准确式 GC 的数据结构）进行封装/解封装的性能损耗。因此在已经创建了 MethodAccessor 的情况下，使用 Java 版本的实现会比 native 版本更快。所以当调用次数到达一定次数（15 次）后，会切换成 Java 实现的版本，来优化未来可能的更频繁的反

射调用。

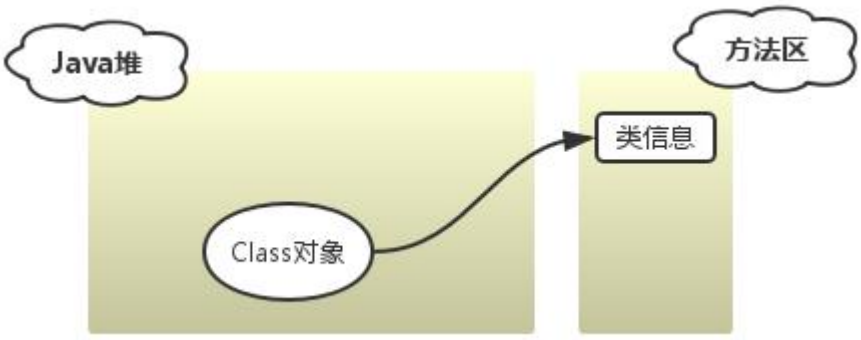
（2）调用 MethodAccessor 对象的 invoke（）方法：

1）调用次数小于等于 15，那么 MethodAccessor 会是 NativeMethodAccessorImpl，即调用 NativeMethodAccessorImpl#invoke（）方法

2）调用次数大于 15，那么 MethodAccessor 会是字节码组装而成的 MethodAccessorImpl，即调用 MethodAccessorImpl#invoke（）方法

Class 对象

虚拟机在 class 文件的加载阶段，把类信息保存在方法区数据结构中，并在 Java 堆中生成一个 Class 对象，作为类信息的入口。



反射的性能问题

1、代码的验证防御逻辑过于复杂，本来这块验证时在链接阶段实现的，使用反射 reflect 时需要在运行时进行；

2、产生过多的临时对象，影响 GC 的消耗；

3、由于缺少上下文，导致不能进行更多的优化，如 JIT；

不过现代 JVM 已经运行的足够快，我们应该把主要重心放在复杂的代码逻辑上，而不是一开始就进行各种性能优化。