1.**方法反射实例**

|  |
| --- |
| 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");          }      }  } |

通过java反射机制可以在运行期间调用对象的任何方法。

如果大量使用这种方式进行调用，会有性能或内存隐患么？为了彻底了解方法的反射机制，只能从底层代码入手了。

2.**Method获取**

调用Class类的getDeclaredMethod可以获取指定方法名和参数的方法对象Method

|  |
| --- |
| **public** Method getDeclaredMethod(String name, Class<?>... parameterTypes)  **throws** NoSuchMethodException, SecurityException {  checkMemberAccess(Member.***DECLARED***, Reflection.*getCallerClass*(), **true**);  **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。**  Method method = *searchMethods*(privateGetDeclaredMethods(**false**), name, parameterTypes);  **if** (method == **null**) {  **throw** **new** NoSuchMethodException(getName() + "." + name + *argumentTypesToString*(parameterTypes));  }  **return** method;  }  **Rd表示缓存数据，如果rd非空，publicOnly为true取rd中所有的public方法，如果publicOnly为false，就取rd中的所有方法，如果rd为空，就从JVM中把符合publicOnly的方法拿出来赋值为res，并按照publicOnly对rd的进行赋值，返回res；**，  **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;  }  **如果reflectionData不为null，且reflectionData.get（）不为null，就返回ReflectionData<T> rd**  **否则，newReflectionData(reflectionData, classRedefinedCount)返回一个新的；**  **private** eflectionData<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);  }  **这里讲一个重要的数据结构ReflectionData，会从JVM中得到下面这些属性，成员，方法，构造器等**  **private** **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;  **volatile** Class<?>[] interfaces;  // Value of classRedefinedCount when we created this ReflectionData instance  **final** **int** redefinedCount;  ReflectionData(**int** redefinedCount) {  **this**.redefinedCount = redefinedCount;  }  }  **reflectionData实际上是个软引用对象，说明在内存紧张时，可能会被回收，如果reflectionData被回收后又执行了反射方法，那么导致reflectionData里面数据为空，只能用下面的方法重新创建一个对象。**  **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；**  -------------------------------------------------------------------------------  **在methods中找到一个方法，名字和参数列表能够匹配的方法，然后赋值一个方法对象出来，如果没有就返回null**  **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));  } |

由此可见，我们每次通过调用getDeclaredMethod方法返回的Method对象其实都是一个新的对象，所以不宜多调哦，如果调用频繁最好缓存起来。

**3.Method的调用**

获取到指定方法的对象Method之后，就可以调用它的invoke方法了

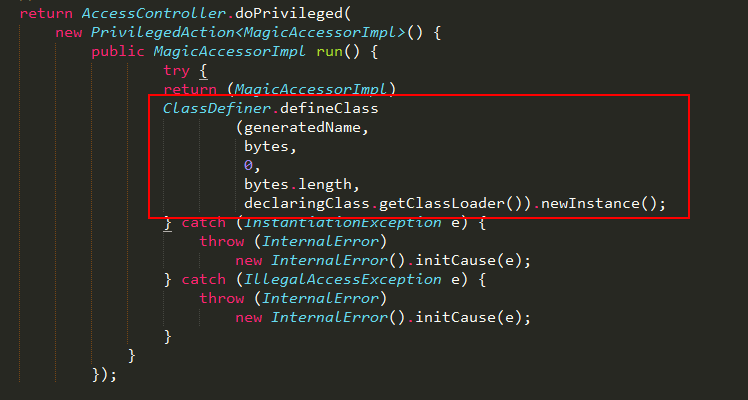
|  |
| --- |
| **public** Object invoke(Object obj, Object... args)  **throws** IllegalAccessException, IllegalArgumentException,  InvocationTargetException  {  **if** (!override) {  **if** (!Reflection.*quickCheckMemberAccess*(clazz, modifiers)) {  Class<?> caller = Reflection.*getCallerClass*();  checkAccess(caller, clazz, obj, modifiers);  }  }  MethodAccessor ma = methodAccessor; // read volatile  **if** (ma == **null**) {  ma = acquireMethodAccessor();  }  **return** ma.invoke(obj, args);  } |

这里的MethodAccessor对象时invoke的关键，一开是methodAccess为空，需要调用acquireMethodAccessor生成一个新的MethodAccessor对象，该对象是一个接口

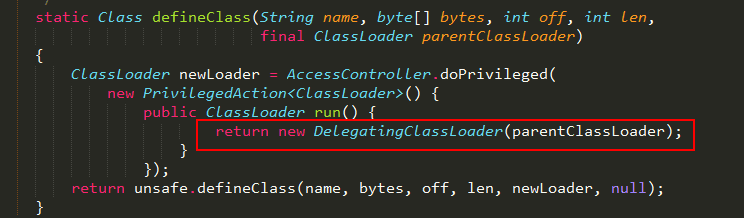


|  |
| --- |
| **Root是Method类中的一个Method对象，在copy方法中会对root进行赋值，getMethodAccessor返回的就是methodAccessor，如果tmp卫康，调用reflectionFactory.newMethodAccessor方法创建一个MethodAccessor对象**  **private MethodAccessor acquireMethodAccessor() {**  **// First check to see if one has been created yet, and take it**  **// if so**  **MethodAccessor tmp = null;**  **if (root != null) tmp = root.getMethodAccessor();**  **if (tmp != null) {**  **methodAccessor = tmp;**  **} else {**  **// Otherwise fabricate one and propagate it up to the root**  **tmp = reflectionFactory.newMethodAccessor(this);**  **setMethodAccessor(tmp);**  **}**  **return tmp;**  **}**  **--------------------------------------------------------------------------------**    **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true，那么使用MethodAccessorGeneratro的generateMethod构造一个MethodAccessor返回。**  **如果noinflation为false，返回DelegatingMethodAccessorImpl，它是MethodAccessorImpl的子类**  http://upload-images.jianshu.io/upload_images/2184951-4e12c4b088111f47.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240  **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，所以最终Method的invoke方法调用的是NativeMethodAccessorImpl对象invoke方法**  http://upload-images.jianshu.io/upload_images/2184951-ebd0c029bbe607a4.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240  **这里用到了ReflectionFactory类中的inflationThreshold，当delegate调用了15次invoke方法之后，如果继续调用就通过MethodAccessorGenerator类的generateMethod方法生成MethodAccessorImpl对象，并设置为delegate对象，这样下次执行Method.invoke时，就调用新建的MethodAccessor对象的invoke()方法了。** |

这里需要注意的是：  
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：



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



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。