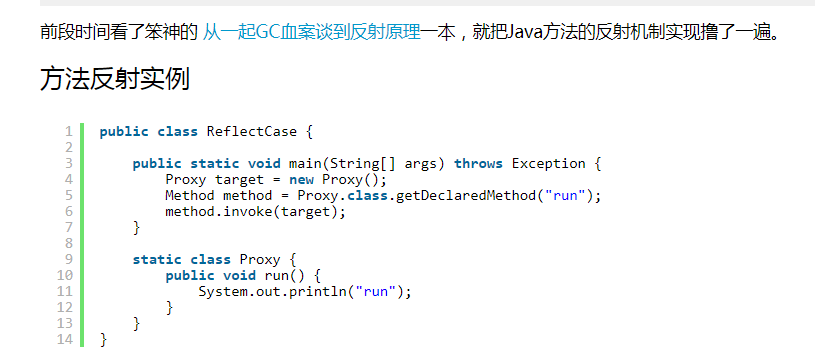
方法反射实例

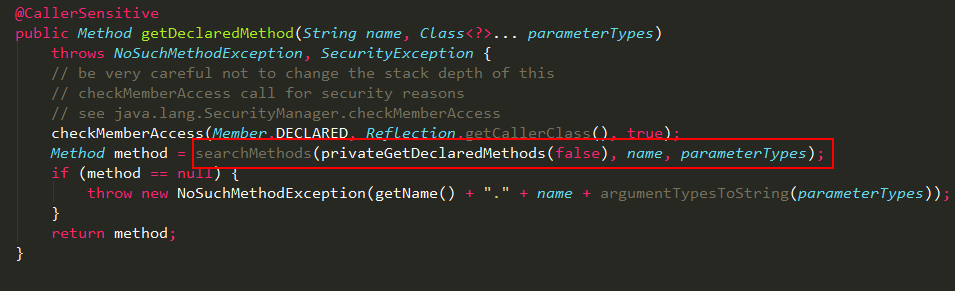


通过Java的反射机制，可以在运行期间调用对象的任何方法，如果大量使用这种方式进行调用，会有性能或内存隐患么？为了彻底了解方法的反射机制，只能从底层代码入手了。

Method获取

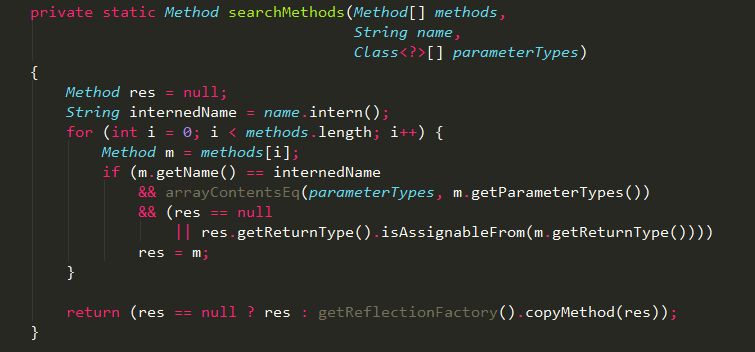
调用Class类的getDeclaredMethod可以获取指定方法名和参数的方法对象Method。

getDeclaredMethod

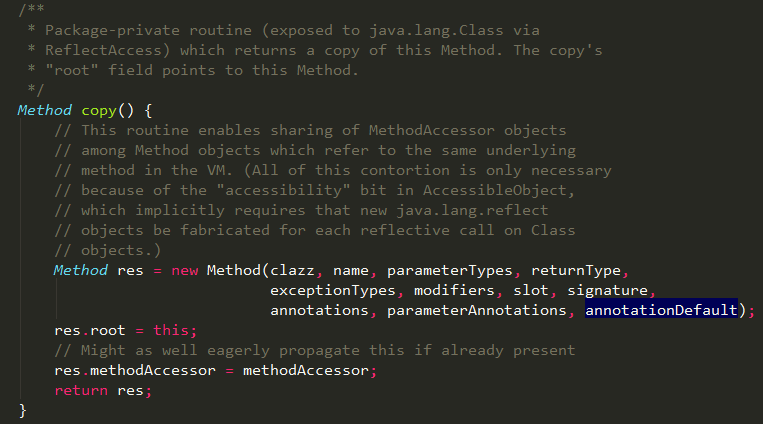


其中privateGetDeclaredMethods方法从缓存或JVM中获取该Class中申明的方法列表，searchMethods方法将从返回的方法列表里找到一个匹配名称和参数的方法对象。

searchMethods



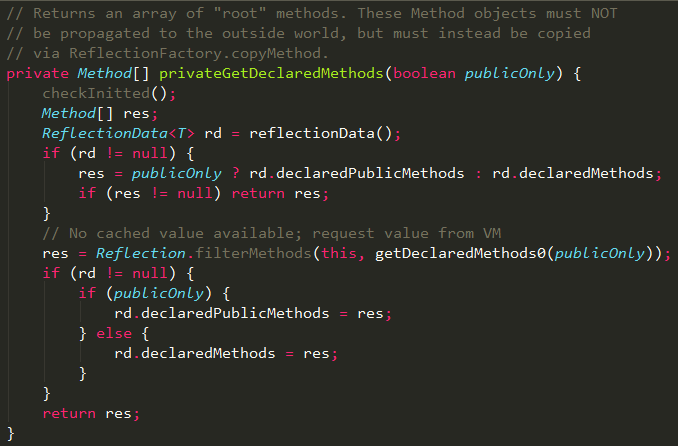
如果找到一个匹配的Method，则重新copy一份返回，即Method.copy()方法



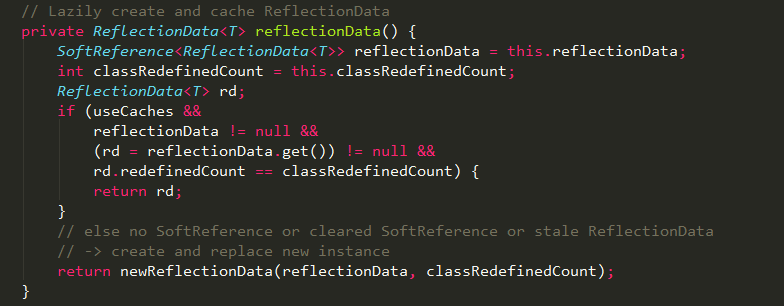
所次每次调用getDeclaredMethod方法返回的Method对象其实都是一个新的对象，且新对象的root属性都指向原来的Method对象，如果需要频繁调用，最好把Method对象缓存起来。

privateGetDeclaredMethods

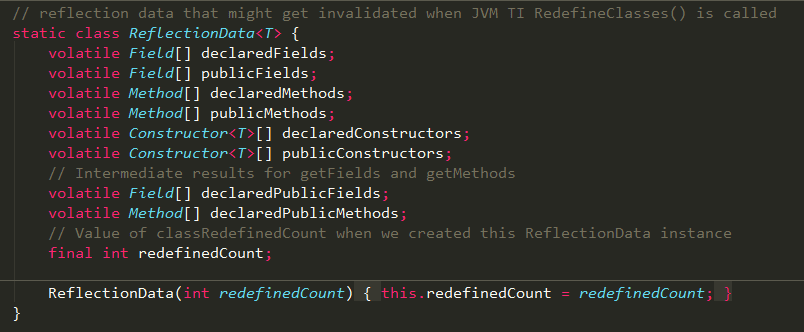
从缓存或JVM中获取该Class中申明的方法列表，实现如下：



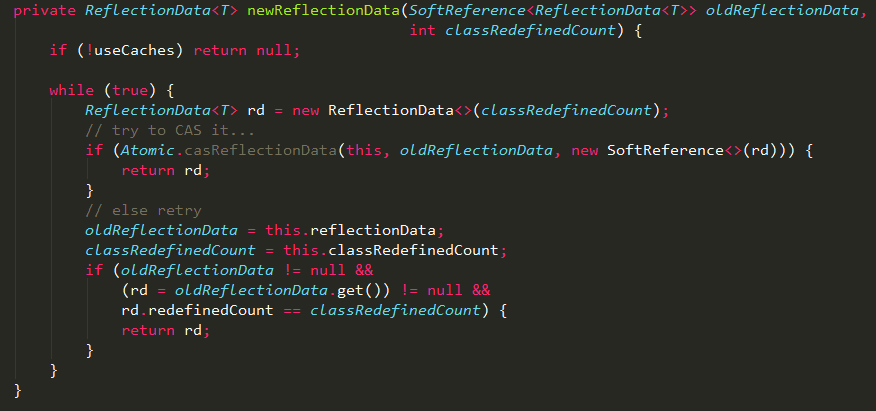
其中reflectionData()方法实现如下：



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：



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：

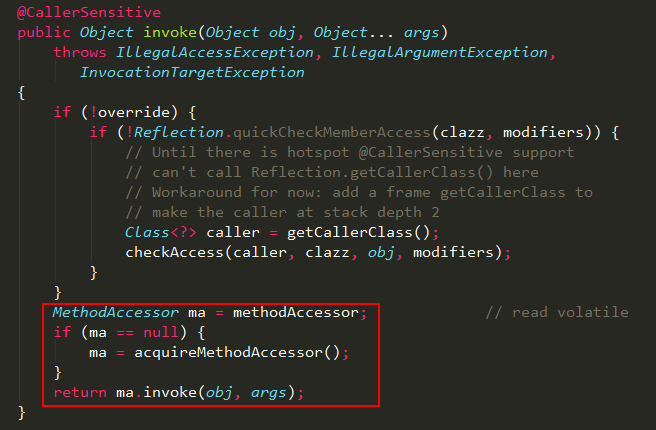


通过unsafe.compareAndSwapObject方法重新设置reflectionData字段；

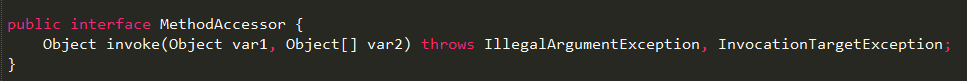
在privateGetDeclaredMethods方法中，如果通过reflectionData()获得的ReflectionData对象不为空，则尝试从ReflectionData对象中获取declaredMethods属性，如果是第一次，或则被GC回收之后，重新初始化后的类属性为空，则需要重新到JVM中获取一次，并赋值给ReflectionData，下次调用就可以使用缓存数据了。

Method调用

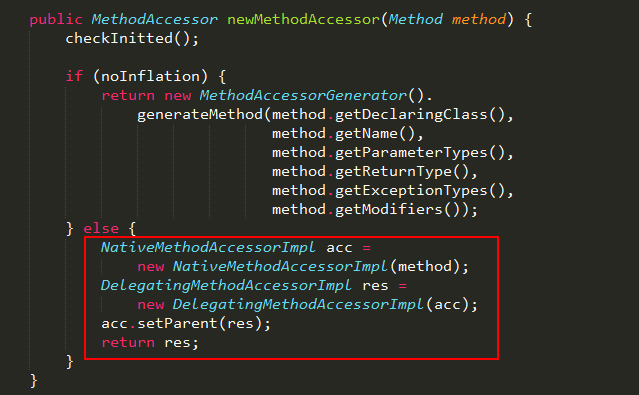
获取到指定的方法对象Method之后，就可以调用它的invoke方法了，invoke实现如下：



应该注意到：这里的MethodAccessor对象是invoke方法实现的关键，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：

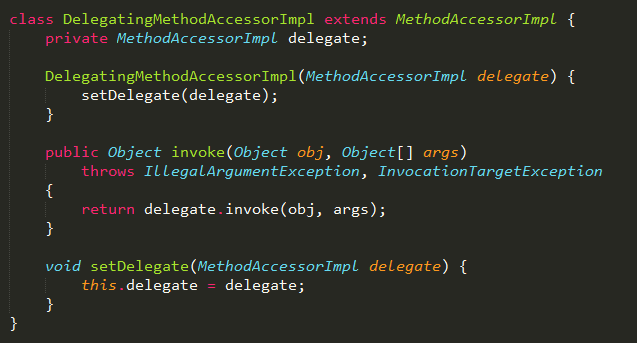


在acquireMethodAccessor方法中，会通过ReflectionFactory类的newMethodAccessor创建一个实现了MethodAccessor接口的对象，实现如下：

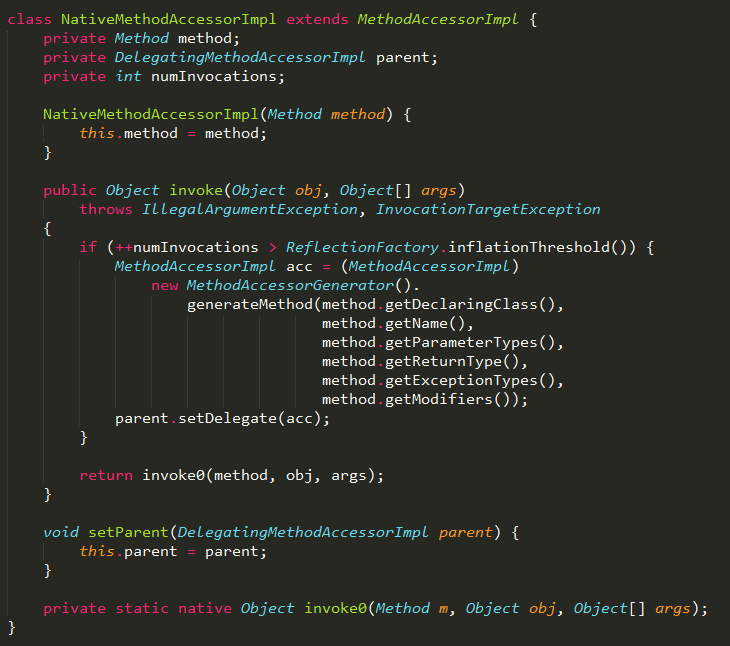


在ReflectionFactory类中，有2个重要的字段：noInflation(默认false)和inflationThreshold(默认15)，在checkInitted方法中可以通过-Dsun.reflect.inflationThreshold=xxx和-Dsun.reflect.noInflation=true对这两个字段重新设置，而且只会设置一次；

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

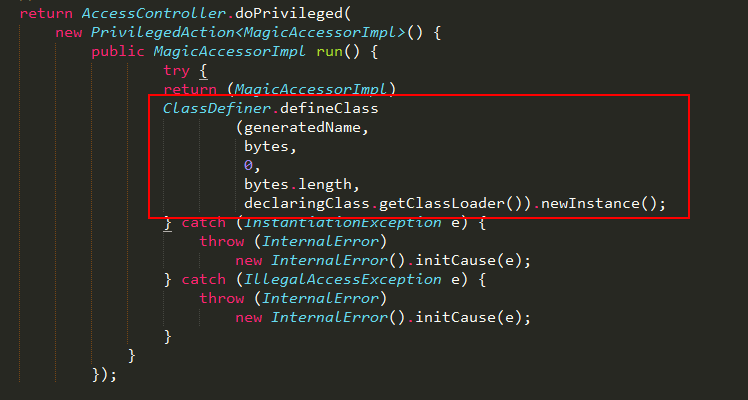


其实，DelegatingMethodAccessorImpl对象就是一个代理对象，负责调用被代理对象delegate的invoke方法，其中delegate参数目前是NativeMethodAccessorImpl对象，所以最终Method的invoke方法调用的是NativeMethodAccessorImpl对象invoke方法，实现如下：

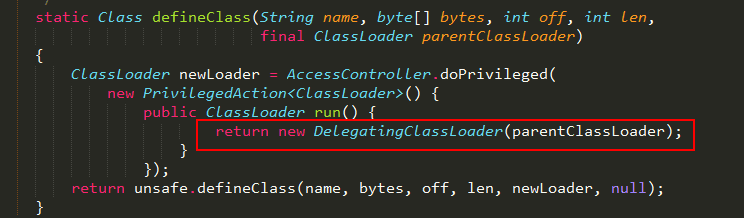


这里用到了ReflectionFactory类中的inflationThreshold，当delegate调用了15次invoke方法之后，如果继续调用就通过MethodAccessorGenerator类的generateMethod方法生成MethodAccessorImpl对象，并设置为delegate对象，这样下次执行Method.invoke时，就调用新建的MethodAccessor对象的invoke()方法了。

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。