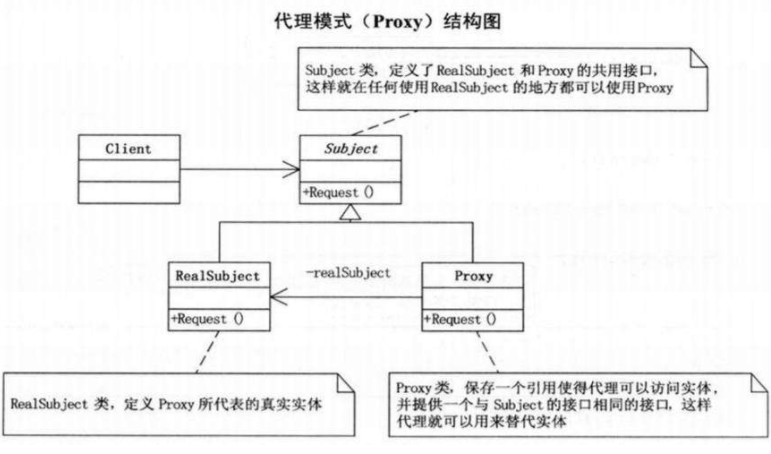
**1.代理模式**



Proxy里面有个RealSubject对象，初始化的时候将RealSubject对象传入，进行Proxy中RealSubject的对象进行初始化，然后Proxy中的方法，都是调用Proxy对象的方法，可在前后加需求，客户端只需使用多态，构造一个接口，即可。调用接口的方法，实际上就是调用Proxy方法，即RealSubject方法。

**2.静态代理**

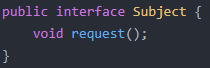
按上述类图进行操作就是静态代理，静态代理如果要对多个方法进行处理，就得在多个方法前后进行修改，不方便，因此产生动态代理。

**3.动态代理**

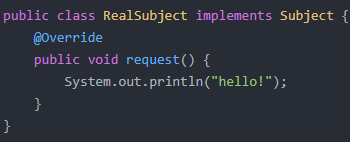
代理类在程序运行时创建的代理方式，可以方便的对代理类的函数进行统一的处理，而不用修改代理类的每个方法。

**3.1整体的用法**

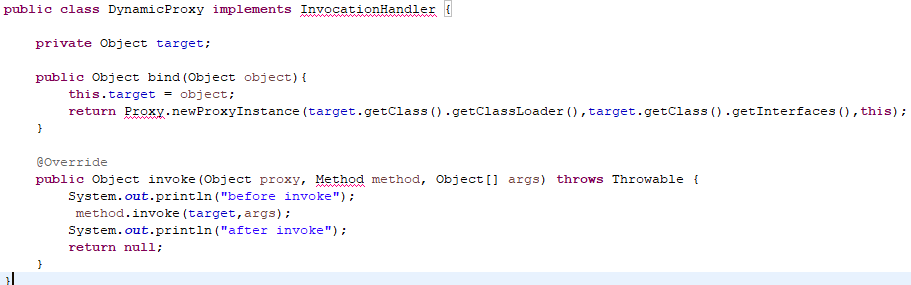
接口



具体对象



代理类



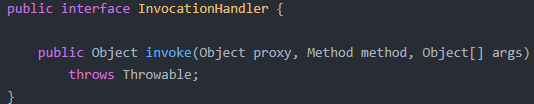
测试类



**3.2底层原理**

主要关于InvocationHandler和Proxy

InvocationHandler接口



子类重写的时候，需要将具体实现类对象（不是代理类）传递进去，然后用反射调用具体实现类的方法，同时增加其他业务逻辑。

Proxy类

|  |
| --- |
| **public static Object newProxyInstance(ClassLoader loader,**  **Class<?>[] interfaces,**  **InvocationHandler h)**  **throws IllegalArgumentException**  **{**  **Objects.*requireNonNull*(h);handler的非空判断**  **final Class<?>[] intfs = interfaces.clone();**  **final SecurityManager sm = System.*getSecurityManager*();**  **if (sm != null) {**  ***checkProxyAccess*(Reflection.*getCallerClass*(), loader, intfs);**  **}**    **Class<?> cl = *getProxyClass0*(loader, intfs);根据classloader和接口数组生成代理类的class对象**    **try {**  **if (sm != null) {**  ***checkNewProxyPermission*(Reflection.*getCallerClass*(), cl);**  **}**    **获取动态生成的代理类中参数为InvocationHandler的构造方法**  **//private static final Class<?>[] constructorParams = { InvocationHandler.class };**  **final Constructor<?> cons = cl.getConstructor(*constructorParams*);**  **final InvocationHandler ih = h;**  **得到cl代理类class中类的修饰符，然后判断是不是public类型，如果是，就返回指定构造器cons的实例，即代理对象的实例，如果不是，就将cons构造方法设置为可访问。**  **if (!Modifier.*isPublic*(cl.getModifiers())) {**  **AccessController.*doPrivileged*(new PrivilegedAction<Void>() {**  **public Void run() {**  **cons.setAccessible(true);**  **return null;**  **}**  **});**  **}**  **return cons.newInstance(new Object[]{h});**  **} catch (IllegalAccessException|InstantiationException e) {**  **throw new InternalError(e.toString(), e);**  **} catch (InvocationTargetException e) {**  **Throwable t = e.getCause();**  **if (t instanceof RuntimeException) {**  **throw (RuntimeException) t;**  **} else {**  **throw new InternalError(t.toString(), t);**  **}**  **} catch (NoSuchMethodException e) {**  **throw new InternalError(e.toString(), e);**  **}**  **}** |

1、根据传递进来的ClassLoader，以及我们的代理对象的父接口数组，来动态创建二进制的class文件，然后根据创建好的Class二进制文件，获取到创建的动态代理类的Class对象。

2、通过代理类的class对象，获取class对象中参数为InvocationHandler的构造方法

3、判断构造方法的访问修饰符，如果不是public的，将其设置成可以访问的

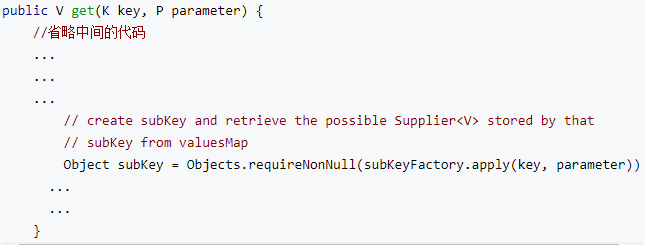
4、调用构造器的newInstance方法，参数为InvocationHandler，返回代理类的实例

|  |
| --- |
| **private** **static** Class<?> getProxyClass0(ClassLoader loader, Class<?>... interfaces) {  **if** (interfaces.length > 65535) {  **throw** **new** IllegalArgumentException("interface limit exceeded");  }  ***proxyClassCache是WeakCache类型，如果缓存中包含接口数组和classloader的class类已经存在，就返回缓存副本，否则通过ProxyClassFactory创建一个对应的class对象***  **return** ***proxyClassCache***.get(loader, interfaces);  } |

WeakCache这个对象当中会在get取不到值时，去生成一个值，放入进去；



ProxyClassFactory是Proxy的一个静态内部类，主要就是用来根据classLoader和接口数组来生成Class对象的。

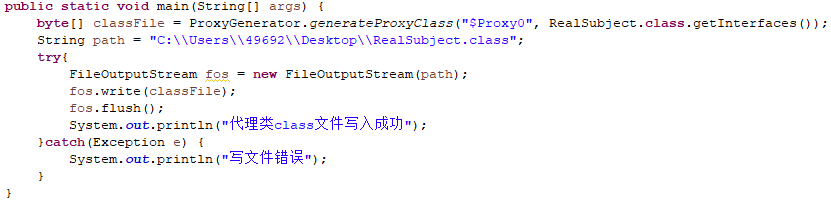


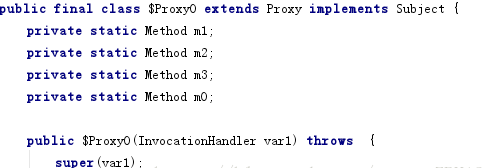
subKeyFactory.apply(key, parameter) 方法调用的是ProxyClassFactory@apply(ClassLoader loader, Class<?>[] interfaces) 方法

|  |
| --- |
| **public** Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {  Map<Class<?>, Boolean> interfaceSet = **new** IdentityHashMap<>(interfaces.length);用来存传入的接口  **for** (Class<?> intf : interfaces) {    Class<?> interfaceClass = **null**;  **try** {  根据接口全限定类名和classLoader，获取到接口class对象  interfaceClass = Class.*forName*(intf.getName(), **false**, loader);  } **catch** (ClassNotFoundException e) {  }  如果两次接口class对象不一致，直接抛出异常，说明创建错误  **if** (interfaceClass != intf) {  **throw** **new** IllegalArgumentException(  intf + " is not visible from class loader");  }  判断创建出来的接口是不是接口类型，不是就抛异常  **if** (!interfaceClass.isInterface()) {  **throw** **new** IllegalArgumentException(  interfaceClass.getName() + " is not an interface");  }  判断接口的set集合是否已经存在了该接口类，存在抛出异常，不存在就添加进去  **if** (interfaceSet.put(interfaceClass, Boolean.***TRUE***) != **null**) {  **throw** **new** IllegalArgumentException(  "repeated interface: " + interfaceClass.getName());  }  }  String proxyPkg = **null**; // package to define proxy class in  **int** accessFlags = Modifier.***PUBLIC*** | Modifier.***FINAL***;  判断非public接口是不是在同一个包内，如果不是，抛出异常；  声明代理类所在的包位置，  **for** (Class<?> intf : interfaces) {  **int** flags = intf.getModifiers();  **if** (!Modifier.*isPublic*(flags)) {  accessFlags = Modifier.***FINAL***;  String name = intf.getName();  **int** n = name.lastIndexOf('.');  String pkg = ((n == -1) ? "" : name.substring(0, n + 1));  **if** (proxyPkg == **null**) {  proxyPkg = pkg;  } **else** **if** (!pkg.equals(proxyPkg)) {  **throw** **new** IllegalArgumentException(  "non-public interfaces from different packages");  }  }  }  如果都是public接口设定全限定类名com.sun.proxy.$proxy0  **if** (proxyPkg == **null**) {  // if no non-public proxy interfaces, use com.sun.proxy package  proxyPkg = ReflectUtil.***PROXY\_PACKAGE*** + ".";  }    **long** num = ***nextUniqueNumber***.getAndIncrement();  String proxyName = proxyPkg + ***proxyClassNamePrefix*** + num;  根据代理类全限定类名，接口数组，访问修饰符，生成代理类的字节码  **byte**[] proxyClassFile = ProxyGenerator.*generateProxyClass*(  proxyName, interfaces, accessFlags);  **try** {  根据生成的字节码，创建class对象并返回。Native方法  **return** *defineClass0*(loader, proxyName,  proxyClassFile, 0, proxyClassFile.length);  } **catch** (ClassFormatError e) {  **throw** **new** IllegalArgumentException(e.toString());  }  }  } |
|  |

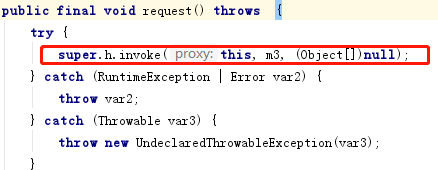
可以看到这段代码就是设置好需要生成的类的类名，然后调用ProxyGenerator.generateProxyClass来生成代理类的字节码

|  |
| --- |
| public static byte[] generateProxyClass(final String var0, Class<?>[] var1, int var2) {  ProxyGenerator var3 = new ProxyGenerator(var0, var1, var2);  final byte[] var4 = var3.generateClassFile();  //中间省略掉一部分代码  return var4;  }  private byte[] generateClassFile() {    //将object类当中的 hashcode，equals，toString方法添加到动态代理类当中  this.addProxyMethod(hashCodeMethod, Object.class);  this.addProxyMethod(equalsMethod, Object.class);  this.addProxyMethod(toStringMethod, Object.class);  Class[] var1 = this.interfaces;  int var2 = var1.length;    int var3;  Class var4;    **//遍历父接口数据**  for(var3 = 0; var3 < var2; ++var3) {  var4 = var1[var3];    **//获取每个接口当中的方法**  Method[] var5 = var4.getMethods();  int var6 = var5.length;    **//遍历接口当中的方法，将接口当中的方法都添加至动态代理类当中**  for(int var7 = 0; var7 < var6; ++var7) {  Method var8 = var5[var7];  this.addProxyMethod(var8, var4);  }  }    Iterator var11 = this.proxyMethods.values().iterator();    **//检查代理类当中的返回类型**  List var12;  while(var11.hasNext()) {  var12 = (List)var11.next();  checkReturnTypes(var12);  }    Iterator var15;  try {  **// 将构造方法添加至代理类当中的方法集合中**  this.methods.add(this.generateConstructor());  var11 = this.proxyMethods.values().iterator();    **//遍历代理类当中的方法，此处使用两层循环，是因为方法签名相同的，可能有多个方法**  while(var11.hasNext()) {  var12 = (List)var11.next();  var15 = var12.iterator();    while(var15.hasNext()) {  ProxyGenerator.ProxyMethod var16 = (ProxyGenerator.ProxyMethod)var15.next();  this.fields.add(new ProxyGenerator.FieldInfo(var16.methodFieldName, "Ljava/lang/reflect/Method;", 10));  this.methods.add(var16.generateMethod());  }  }  **// 将静态代码块添加进去**  this.methods.add(this.generateStaticInitializer());  } catch (IOException var10) {  throw new InternalError("unexpected I/O Exception", var10);  }    if(this.methods.size() > '\uffff') {  throw new IllegalArgumentException("method limit exceeded");  } else if(this.fields.size() > '\uffff') {  throw new IllegalArgumentException("field limit exceeded");  } else {  /\*\*  \* 省略部分代码  \*\*/  return var13.toByteArray();  } catch (IOException var9) {  throw new InternalError("unexpected I/O Exception", var9);  }  }  } |

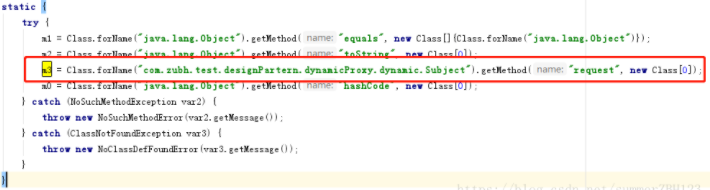




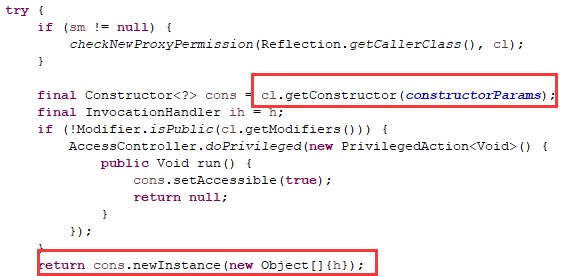
生成了代理类对象，继承自Proxy，实现了subject接口，其构造方法是传递了InvocationHandler参数，同时实现了equals ，hashcode，toString方法，以及我们自己定义的request方法。



这里就能清楚的看到，代理类中，request方法，其实就是调用InvocationHandler实现类中invoke方法，m3就是RealSubject中定义的request方法



这里就能知道，调用Subject的request的时候，为什么能够调用动态代理类的invoke方法了。因为在调用bind中的Proxy.newProxyInstance的时候，传入的是我们classLoader，代理类的父接口，和自定义的InvocationHandler，因此生成的代理类对象中的h就是自定义的InvocationHandler。



**整体流程**：

**使用方法**：

定义一个接口Subject，里面有抽象方法request，一个接口的实现类RealSubject，里面有真正的request方法，定义个ProxyHandler，实现InvocationHandler接口，有ProxyHandler中有个目标对象target，用于传入RealSubject的实例；里面有两个方法，bind方法，将传入的RealSubject对象，赋值给target，并调用Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), this)返回代理类对象的实例；还有一个方法，invoke（Object proxy, Method method, Object[] args）方法，在method.invoke前后可以加逻辑功能；method方法对象包含方法的相关信息，method.invoke就会去找target对象中，指定参数与该方法匹配的方法。

**底层原理**：

首先分析代理对象的产生过程，最后分析代理对象的产生后的样子

产生过程：

Proxy.newInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), this) 方法，该方法主要做了

①获取代理类的接口Subject.class对象的副本intfs。

②调用getProxyClass0(loader, intfs)方法，获取代理类的class对象

--2.1 getProxyClass方法会去调用proxyClassCache.get(loader,interfaces)方法，该方法的作用，如果缓存中已经有该接口对应的代理类的副本，那么返回代理类的副本，否则使用ProxyClassFactory的apply方法创建一个class对象；get方法中有一句表现了这个过程，如下图



SubkeyFactory与ProxyClassFactory都是BiFunction接口的的实现类，此处调用的是ProxyClassFactory的apply方法。

----2.1.1 apply方法：根据接口的全限定类名以及指定的classLoader，调用class.forName方法得到接口的class对象interfaceClass，进行判断是否是接口类型，是否有重复，

----2.1.2 设定代理类所在的包路径，如果有非public接口，且不是在同一个包内，抛异常，如果在同一个包内，就设定代理类的包路径为proxyPkg，如果都是public接口，那么生成的代理类的包路径proxyPkg设定为com.sum.proxy.

----2.1.3 根据包路径，类名前缀和num数字，确定代理类的名字

----2.1.4 调用ProxyGenerator.generateProxyClass(proxyName, interfaces, accessFlags);内部生成一个ProxyGenerator对象，调用ProxyGenerator.generateClassFile()得到代理类的字节码文件；

------2.1.4.1 ProxyGenerator.generateClassFile()方法，为代理类添加了三个object类的hashcode，equals，toString方法，遍历父接口，获取每个接口中的方法，添加方法，添加构造方法等，都加入到代理中；返回二进制字节码文件

----2.1.5 调用defineClass0(loader, proxyName, proxyClassFile, 0,proxyClassFile.length);将字节码文件创建为class对象，返回

③获取到代理类的class对象后，得到代理类参数为InvocationHandler的构造方法，如果是非public类型，就使用setAccessible设定为true，然后调用cons.newInstance(new Object[]{h})来生成代理类对象；

④在代理类对象中，有个request方法，方法中默认会调用Proxy类里面的InvocationHandler也就是我们自定义的InvocationHandler的invoke方法。这就实现了动态代理

**Cglib动态代理**

具体类对象

|  |
| --- |
| **package** cn.huangwei.classLoader.Proxy;  **public** **class** CglibService {  **public** CglibService(){  System.***out***.println("CglibService构造器");  }  **public** **void** getCglib(){  System.***out***.println("CglibService的get方法");  }  } |

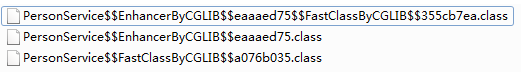
拦截器对象

|  |
| --- |
| **package** cn.huangwei.classLoader.Proxy;  **import** java.lang.reflect.Method;  **import** net.sf.cglib.proxy.MethodInterceptor;  **import** net.sf.cglib.proxy.MethodProxy;  **public** **class** CglibIntercepter **implements** MethodInterceptor{  @Override  **public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {  // **TODO** Auto-generated method stub  System.***out***.println("执行前。。。。");  Object object = proxy.invokeSuper(obj, args);  System.***out***.println("执行后。。。。");  **return** object;  }  } |

测试类

|  |
| --- |
| **package** cn.huangwei.classLoader.Proxy;  **import** net.sf.cglib.proxy.Enhancer;  **public** **class** TestCglibProxy {  **public** **static** **void** main(String[] args) {  //实例化增强器，类生成器  Enhancer eh = **new** Enhancer();  //设置目标类  eh.setSuperclass(CglibService.**class**);  //设置拦截对象  CglibIntercepter ci = **new** CglibIntercepter();  eh.setCallback(ci);  //生成代理类对象，返回实例  CglibService cs = (CglibService) eh.create();  cs.getCglib();  }  } |

可以得到Cglib生成的三个文件



第一个是代理类的fastclass，第二个是被代理类CglibService的fastclass

Aaaed75.class就是cglib生成的代理类，继承了CglibService类（CglibService类）实现了Factory，只给出部分关键代码

|  |
| --- |
| public class CglibService$$EnhancerByCGLIB$$eaaaed75  extends CglibService  implements Factory  {    **private MethodInterceptor CGLIB$CALLBACK\_0;//拦截器**  **private static final Method CGLIB$ getCglib $0$Method;//被代理方法**  **private static final MethodProxy CGLIB$ getCglib $0$Proxy;//代理方法**      static void CGLIB$STATICHOOK1()  {  CGLIB$THREAD\_CALLBACKS = new ThreadLocal();  CGLIB$emptyArgs = new Object[0];  **Class localClass1 = Class.forName("cn.huangwei.classLoader.Proxy.** **CglibService $$EnhancerByCGLIB$$eaaaed75");//代理类**  **Class localClass2;//被代理类CglibService**  。。。  **Method[] tmp223\_220 = ReflectUtils.findMethods(new String[] { " getCglib ", "()V" }, (localClass2 =**  **Class.forName("cn.huangwei.classLoader.Proxy.** **CglibService ")).getDeclared- Methods());**  **CGLIB$ getCglib $0$Method = tmp223\_220[0];**  **CGLIB$ getCglib $0$Proxy = MethodProxy.create(localClass2, localClass1, "()V", " getCglib ", "CGLIB$ getCglib $0");**  tmp223\_220;  return;  }  public class MethodProxy {  private Signature sig1;  private Signature sig2;  private MethodProxy.CreateInfo createInfo;  private final Object initLock = new Object();  private volatile MethodProxy.FastClassInfo fastClassInfo;  //c1:被代理对象Class  //c2:代理对象Class  //desc：入参类型  //name1:被代理方法名  //name2:代理方法名  public static MethodProxy create(Class c1, Class c2, String desc, String name1, String name2) {  MethodProxy proxy = new MethodProxy();  proxy.sig1 = new Signature(name1, desc);//被代理方法签名  proxy.sig2 = new Signature(name2, desc);//代理方法签名  proxy.createInfo = new MethodProxy.CreateInfo(c1, c2);  return proxy;  }  private static class CreateInfo {  Class c1;  Class c2;  NamingPolicy namingPolicy;  GeneratorStrategy strategy;  boolean attemptLoad;   public CreateInfo(Class c1, Class c2) {  this.c1 = c1;  this.c2 = c2;  AbstractClassGenerator fromEnhancer = AbstractClassGenerator.getCurrent();  if(fromEnhancer != null) {  this.namingPolicy = fromEnhancer.getNamingPolicy();  this.strategy = fromEnhancer.getStrategy();  this.attemptLoad = fromEnhancer.getAttemptLoad();  }   } }  我们通过代理类的源码可以看到，代理类会获得所有在父类继承来的方法，并且会有MethodProxy与之对应，比如 Method CGLIB$getCglib $0$Method、MethodProxy CGLIB$getCglib $0$Proxy;  //被代理方法（methodProxy.invokeSuper会调用）  final void CGLIB$ getCglib $0() {  super. getCglib ();  }  //代理方法  public final void getCglib () {  MethodInterceptor var10000 = this.CGLIB$CALLBACK\_0;  if(this.CGLIB$CALLBACK\_0 == null) {  CGLIB$BIND\_CALLBACKS(this);  var10000 = this.CGLIB$CALLBACK\_0;  }  if(var10000 != null) {  //调用拦截器  var10000.intercept(this, CGLIB$setPerson$0$Method, CGLIB$emptyArgs, CGLIB$setPerson$0$Proxy);  } else {  super. getCglib ();  }  }  主函数使用代理对象调用getCglib方法->调用拦截器->methodProxy.invokeSuper->CGLIB$getCglib $0->被代理对象getCglib方法  public class MethodProxy {  private Signature sig1;  private Signature sig2;  private MethodProxy.CreateInfo createInfo;  private final Object initLock = new Object();  private volatile MethodProxy.FastClassInfo fastClassInfo;  //c1:被代理对象Class  //c2:代理对象Class  //desc：入参类型  //name1:被代理方法名  //name2:代理方法名  public static MethodProxy create(Class c1, Class c2, String desc, String name1, String name2) {  MethodProxy proxy = new MethodProxy();  proxy.sig1 = new Signature(name1, desc);//被代理方法签名  proxy.sig2 = new Signature(name2, desc);//代理方法签名  proxy.createInfo = new MethodProxy.CreateInfo(c1, c2);  return proxy;  }  private static class CreateInfo {  Class c1;  Class c2;  NamingPolicy namingPolicy;  GeneratorStrategy strategy;  boolean attemptLoad;   public CreateInfo(Class c1, Class c2) {  this.c1 = c1;  this.c2 = c2;  AbstractClassGenerator fromEnhancer = AbstractClassGenerator.getCurrent();  if(fromEnhancer != null) {  this.namingPolicy = fromEnhancer.getNamingPolicy();  this.strategy = fromEnhancer.getStrategy();  this.attemptLoad = fromEnhancer.getAttemptLoad();  }   } }  public Object invoke(Object obj, Object[] args) throws Throwable {  try {  init();  FastClassInfo fci = fastClassInfo;  return fci.f1.invoke(fci.i1, obj, args);  } catch (InvocationTargetException e) {  throw e.getTargetException();  } catch (IllegalArgumentException e) {  if (fastClassInfo.i1 < 0)  throw new IllegalArgumentException("Protected method: " + sig1);  throw e;  }  }  public Object invokeSuper(Object obj, Object[] args) throws Throwable {  try {  this.init();  MethodProxy.FastClassInfo fci = this.fastClassInfo;  return fci.f2.invoke(fci.i2, obj, args);  } catch (InvocationTargetException var4) {  throw var4.getTargetException();  }  }  **上面代码调用过程就是获取到代理类对应的fastclass，并执行了代理方法，init方法就是用于生成fastclass对象，如果已存在就不创建；**  -------------------------------------------------------------------------------------  Cglib动态代理执行代理方法效率之所以比JDK的高是因为Cglib采用了FastClass机制，它的原理简单来说就是：为代理类和被代理类各生成一个Class，这个Class会为代理类或被代理类的方法分配一个index(int类型)。getIndex就是用于给index赋值的方法，该方法在init中会被调用，这个index当做一个入参，FastClass就可以直接定位到要调用的方法直接进行调用省去了反射的过程、  private static class FastClassInfo {   **FastClass f1;//被代理类FastClass  FastClass f2;//代理类FastClass  int i1; //被代理类的方法签名(index)  int i2;//代理类的方法签名**   private FastClassInfo() {  } }  **//根据方法签名获取index，在init方法中会被调用**  public int getIndex(Signature var1) {  String var10000 = var1.toString();  switch(var10000.hashCode()) {  case -2077043409:  if(var10000.equals("getPerson(Ljava/lang/String;)Lcom/demo/pojo/Person;")) {  return 21;  }  break;  case -2055565910:  if(var10000.equals("CGLIB$SET\_THREAD\_CALLBACKS([Lnet/sf/cglib/proxy/Callback;)V")) {  return 12;  }  break;  case -1902447170:  if(var10000.equals("getCglib()V")) {  return 7;  }  break;  //省略部分代码..... 　  **//根据index直接定位执行方法，在methodProxy的invoke或者invokesuper中调用**  public Object invoke(int var1, Object var2, Object[] var3) throws InvocationTargetException {  eaaaed75 var10000 = (eaaaed75)var2;  int var10001 = var1;  try {  switch(var10001) {  case 7:  var10000.setPerson();  return null;  。。。。。。  case 19:  var10000.CGLIB$setPerson$0();  return null;  //省略部分代码....  } catch (Throwable var4) {  throw new InvocationTargetException(var4);  }  throw new IllegalArgumentException("Cannot find matching method/constructor");  }  //MethodProxy invoke/invokeSuper都调用了init() private void init() {  if(this.fastClassInfo == null) {  Object var1 = this.initLock;  synchronized(this.initLock) {  if(this.fastClassInfo == null) {  MethodProxy.CreateInfo ci = this.createInfo;  MethodProxy.FastClassInfo fci = new MethodProxy.FastClassInfo();  **fci.f1 = helper(ci, ci.c1);/**/如果缓存中就取出，没有就生成新的FastClass  fci.f2 = helper(ci, ci.c2);  **fci.i1 = fci.f1.getIndex(this.sig1);//获取方法的index**  **fci.i2 = fci.f2.getIndex(this.sig2);**  this.fastClassInfo = fci;  this.createInfo = null;  }  }  } |

总过程：生成一个enhancer的实例，设定被代理类和拦截器，调用enhancer的create方法，之后能得到代理类对象proxy，并完成一些列的初始化工作，代理类对象继承了CglibService类，代理对象含有MethodInterceptor拦截器、表示被代理的方法Method对象和表示被代理的方法的MethodProxy；以及两个方法CGLIB$ getCglib $0和getCglib()方法；当主函数中调用proxy的getCglib时，会先获取拦截器，此时的拦截器，就是通过enhancer设定拦截器即CglibInterceptor的实例；然后调用intercept方法，完成回调；在CglibInterceptor类中的intercept方法中，调用methodProxy.invokeSuper方法；内部执行是，初始化fastclass，然后通过fastclass机制，获取到方法对应的index值，根据调index值用fastclass的invoke方法，此时将会调用CGLIB$ getCglib $0，即调用super父类CglibService的getCglib方法；调用结束。如果methodProxy.invoke方法，根据调index值用fastclass的invoke方法，此时将会调用getCglib造成循环调用；

1.JDK动态代理是实现了被代理对象的接口，Cglib是继承了被代理对象。  
2.JDK和Cglib都是在运行期生成字节码，JDK是直接写Class字节码，Cglib使用ASM框架写Class字节码，Cglib代理实现更复杂，生成代理类比JDK效率低。  
3.JDK调用代理方法，是通过反射机制调用，Cglib是通过FastClass机制直接调用方法，Cglib执行效率更高。