# 设计模式



## 工厂模式

不使用工厂模式

```
1  public class BMW320 {
2      public BMW320(){
3          System.out.println("制造-->BMW320");
4      }
5  }
6
7  public class BMW523 {
8      public BMW523(){
9          System.out.println("制造-->BMW523");
10     }
11 }
12
13 public class Customer {
14     public static void main(String[] args) {
15         BMW320 bmw320 = new BMW320();
16         BMW523 bmw523 = new BMW523();
```

```
    }
}
```

使用简单工厂模式

产品类

```
abstract class BMW {
    public BMW(){

    }
}

public class BMW320 extends BMW {
    public BMW320() {
        System.out.println("制造-->BMW320");
    }
}
public class BMW523 extends BMW{
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

工厂类

```
public class Factory {
    public BMW createBMW(int type) {
        switch (type) {

        case 320:
            return new BMW320();

        case 523:
            return new BMW523();

        default:
```

```
            break;
        }
        return null;
    }
}
```

客户类

```
public class Customer {
    public static void main(String[] args) {
        Factory factory = new Factory();
        BMW bmw320 = factory.createBMW(320);
        BMW bmw523 = factory.createBMW(523);
    }
}
```

1) 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑，用来创建产品
2) 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。
3) 具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现。

工厂方法模式

工厂方法模式去掉了简单工厂模式中工厂方法的静态属性，使得它可以被子类继承。这样在简单工厂模式里集中在工厂方法上的压力可以由工厂方法模式里不同的工厂子类来分担。

工厂方法模式组成：

1)抽象工厂角色： 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。

2)具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。

3)抽象产品角色：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。

4)具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。

(开闭原则)当有新的产品产生时，只要按照抽象产品角色、抽象工厂角色提供的合同来生成，那么就可以被客户使用，而不必去修改任何已有 的代码。

产品类

```
abstract class BMW {
    public BMW(){
```

```
3
4       }
5   }
6   public class BMW320 extends BMW {
7       public BMW320() {
8           System.out.println("制造-->BMW320");
9       }
10  }
11  public class BMW523 extends BMW{
12      public BMW523(){
13          System.out.println("制造-->BMW523");
14      }
15  }
```

创建工厂类：

```
1   interface FactoryBMW {
2       BMW createBMW();
3   }
4
5   public class FactoryBMW320 implements FactoryBMW{
6
7       @Override
8       public BMW320 createBMW() {
9
10          return new BMW320();
11      }
12
13  }
14  public class FactoryBMW523 implements FactoryBMW {
15      @Override
16      public BMW523 createBMW() {
17
18          return new BMW523();
19      }
20  }
```

客户类

```
public class Customer {
    public static void main(String[] args) {
        FactoryBMW320 factoryBMW320 = new FactoryBMW320();
        BMW320 bmw320 = factoryBMW320.createBMW();

        FactoryBMW523 factoryBMW523 = new FactoryBMW523();
        BMW523 bmw523 = factoryBMW523.createBMW();
    }
}
```

# 策略模式

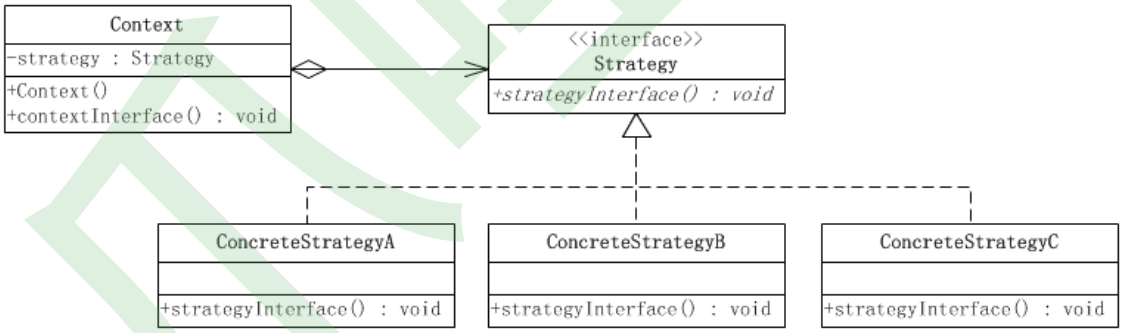
定义

定义一组算法，将每一个算法封装起来，从而使它们可以相互切换。

特点

1）一组算法，那就是不同的策略。

2）这组算法都实现了相同的接口或者继承相同的抽象类，所以可以相互切换。




策略模式涉及到的角色有三个：

– 封装角色：上层访问策略的入口，它持有抽象策略角色的引用。

– 抽象策略角色：提供接口或者抽象类，定义策略组必须拥有的方法和属性。

– 具体策略角色：实现抽象策略，定义具体的算法逻辑。

```
// Context持有Strategy的引用，并且提供了调用策略的方法，
  public class Context {

    private Strategy strategy;

    /**
     * 传进的是一个具体的策略实例
     * @param strategy
     */
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    /**
     * 调用策略
     */
    public void contextInterface() {
        strategy.algorithmLogic();
    }

}
```

```
// 抽象策略角色，定义了策略组的方法
public interface Strategy {

    public void algorithmLogic();

}
```

```
// 具体策略角色类
public class ConcreteStrategyA implements Strategy{

    @Override
    public void algorithmLogic() {
```

```
        // 具体的算法逻辑（）
    }
}
```

```
// 客户端
public class Client {

    public static void main(String[] args) {
        // 操控比赛，这场要输
        Context context = new Context(new ConcreteStrategyA());
        context.contextInterface();
    }
}
```

策略模式的优点

（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免代码重复。

（2）使用策略模式可以避免使用多重条件(if-else)语句。多重条件语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重条件语句里面，比使用继承的办法还要原始和落后。

策略模式的缺点

（1）客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道算法或行为的情况。

（2）由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

# 工厂结合策略实战

课堂代码

# 单例设计模式(面试重点)

概念：单例对象的类必须保证只有一个实例存在

适用场景： 单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等。如：

1. 需要频繁实例化然后销毁的对象。
2. 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
3. 有状态的工具类对象。
4. 频繁访问数据库或文件的对象。

常见写法：

1. 饿汉式

```
public class Singleton {
     /**
     * 优点：没有线程安全问题，简单
     * 缺点：提前初始化会延长类加载器加载类的时间；如果不使用会浪费内存空间；不能传递参数
     */
     private static final Singleton instance = new Singleton();
     private Singleton(){};

     public static Singleton getInstance(){
         return instance;
     }
}
```

2. 懒汉式

```

    public class Singleton{
    /**
     * 优点:解决线程安全，延迟初始化( Effective Java推荐写法)
     */
    private Singleton(){}

    public static Singleton getInstance () {
        return Holder.SINGLE_TON;
    }
```

```
11
12      private static class Holder{
13          private static final Singleton SINGLE_TON = new Singleton
    ();
14      }
15  }
```

1. 双重检查锁（double checked locking）

```
1  public class Singleton {
2      private volatile static Singleton uniqueSingleton;
3
4      private Singleton() {
5      }
6
7      public Singleton getInstance() {
8          if (null == uniqueSingleton) {
9              synchronized (Singleton.class) {
10                 if (null == uniqueSingleton) {
11                     uniqueSingleton = new Singleton();
12                 }
13             }
14         }
15         return uniqueSingleton;
16     }
17 }
```

volatile指令重排序

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

1. 在单线程环境下不能改变程序运行的结果；
2. 存在数据依赖关系的不允许重排序

uniqueSingleton = new Singleton();

分配内存空间

初始化对象

将对象指向刚分配的内存空间

但是有些编译器为了性能的原因，可能会将第二步和第三步进行重排序，顺序就成了：
分配内存空间
将对象指向刚分配的内存空间
初始化对象
现在考虑重排序后，两个线程发生了以下调用：

| Time | Thread A | Thread B |
| --- | --- | --- |
| T1 | 检查到uniqueSingleton为空 | |
| T2 | 获取锁 | |
| T3 | 再次检查到uniqueSingleton为空 | |
| T4 | 为uniqueSingleton分配内存空间 | |
| T5 | 将uniqueSingleton指向内存空间 | |
| T6 | | 检查到uniqueSingleton不为空 |
| T7 | | 访问uniqueSingleton（此时对象还未完成初始化） |
| T8 | 初始化uniqueSingleton | |

在这种情况下，T7时刻线程B对uniqueSingleton的访问，访问的是一个初始化未完成的对象。
使用了volatile关键字后，重排序被禁止，所有的写（write）操作都将发生在读（read）操作之前。

1. 单例模式的破坏

```
Singleton sc1 = Singleton.getInstance();
Singleton sc2 = Singleton.getInstance();
System.out.println(sc1); // sc1, sc2是同一个对象
System.out.println(sc2);
/*通过反射的方式直接调用私有构造器*/
Class<Singleton> clazz = (Class<Singleton>) Class.forName("com.learn.example.Singleton");
Constructor<Singleton> c = clazz.getDeclaredConstructor(null);
c.setAccessible(true); // 跳过权限检查
Singleton sc3 = c.newInstance();
```

```
Singleton sc4 = c.newInstance();
System.out.println("通过反射的方式获取的对象sc3: " + sc3);  // sc3, sc4不是同一个对象
System.out.println("通过反射的方式获取的对象sc4: " + sc4);

//防止反射获取多个对象的漏洞
private Singleton() {
    if (null != SingletonClassInstance.instance)
        throw new RuntimeException();
}
```

1. spring中bean的单例
    a. 当多个用户同时请求一个服务时，容器会给每一个请求分配一个线程，这时多个线程会并发执行该请求对应的业务逻辑（成员方法），此时就要注意了，如果该处理逻辑中有对单例状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。
    有状态就是有数据存储功能。有状态对象(Stateful Bean)，就是有实例变量的对象，可以保存数据，是非线程安全的。在不同方法调用间不保留任何状态。
    无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象 .不能保存数据，是不变类，是线程安全的。
    b. 实现

```
public abstract class AbstractBeanFactory implements ConfigurableBeanFactory{
/**
* 充当了Bean实例的缓存，实现方式和单例注册表相同
*/
private final Map singletonCache=new HashMap();
public Object getBean(String name)throws BeansException{
   return getBean(name,null,null);
}
...
public Object getBean(String name,Class requiredType,Object[] args)throws BeansException{
  //对传入的Bean name稍做处理，防止传入的Bean name名有非法字符(或则做转码)
  String beanName=transformedBeanName(name);
  Object bean=null;
  //手工检测单例注册表
  Object sharedInstance=null;
```

```
  //使用了代码锁定同步块，原理和同步方法相似，但是这种写法效率更高
  synchronized(this.singletonCache){
     sharedInstance=this.singletonCache.get(beanName);
   }
  if(sharedInstance!=null){
     ...
     //返回合适的缓存Bean实例
     bean=getObjectForSharedInstance(name,sharedInstance);
  }else{
    ...
    //取得Bean的定义
    RootBeanDefinition mergedBeanDefinition=getMergedBeanDefiniti
on(beanName,false);
     ...
    //根据Bean定义判断，此判断依据通常来自于组件配置文件的单例属性开关
    //<bean id="date" class="java.util.Date" scope="singleton"/>
    //如果是单例，做如下处理
    if(mergedBeanDefinition.isSingleton()){
       synchronized(this.singletonCache){
        //再次检测单例注册表
         sharedInstance=this.singletonCache.get(beanName);
         if(sharedInstance==null){
            ...
           try {
              //真正创建Bean实例
              sharedInstance=createBean(beanName,mergedBeanDefini
tion,args);
              //向单例注册表注册Bean实例
               addSingleton(beanName,sharedInstance);
           }catch (Exception ex) {
              ...
           }finally{
              ...
        }
       }
      bean=getObjectForSharedInstance(name,sharedInstance);
    }
   //如果是非单例，即prototpye，每次都要新创建一个Bean实例
   //<bean id="date" class="java.util.Date" scope="prototype"/>
```

```
54     else{
55         bean=createBean(beanName,mergedBeanDefinition,args);
56     }
57 }
58 ...
59 return bean;
60 }
61 }
```

## 命令模式

1. 背景:当需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，使得请求发送者与请求接收者消解耦

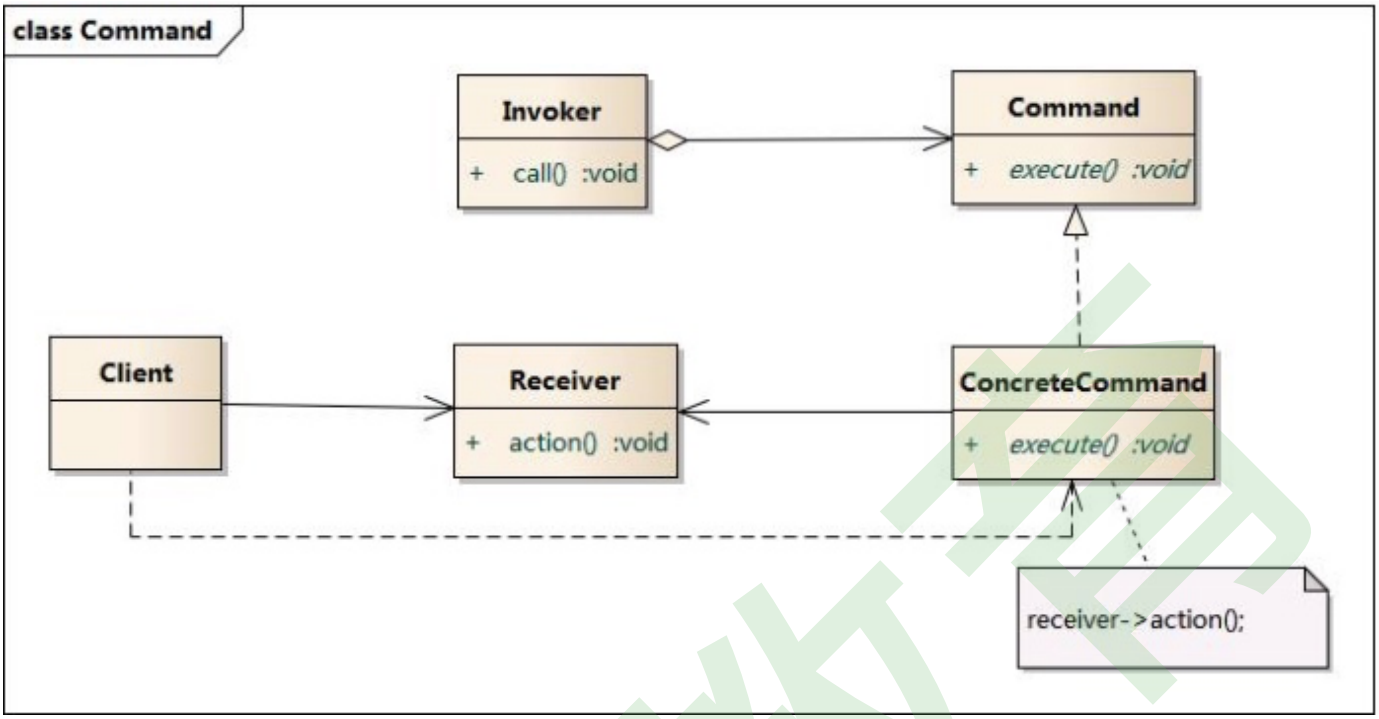
2. 模式定义

命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。

3. 模式结构

命令模式包含如下角色：

Command: 抽象命令类
ConcreteCommand: 具体命令类
Invoker: 调用者
Receiver: 接收者
Client:客户类




4. 命令模式的优点

降低系统的耦合度。

新的命令可以很容易地加入到系统中。

可以比较容易地设计一个命令队列和宏命令（组合命令）。

可以方便地实现对请求的Undo和Redo。

5. 命令模式的缺点

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

6. 样例代码

```
// 客户端，请求者，命令接口，命令实现，接受者，
public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command commandOne = new ConcreteCommandOne(receiver);
        Command commandTwo = new ConcreteCommandTwo(receiver);
        Invoker invoker = new Invoker(commandOne, commandTwo);
        invoker.actionOne();
        invoker.actionTwo();
    }
}

```

```
13 public class Invoker {
14     private Command commandOne;
15     private Command commandTwo;
16 
17     public Invoker(Command commandOne, Command commandTwo) {
18         this.commandOne = commandOne;
19         this.commandTwo = commandTwo;
20 
21     }
22 
23     public void actionOne() {
24         commandOne.execute();
25 
26     }
27 
28     public void actionTwo() {
29         commandTwo.execute();
30 
31     }
32 }
33 
34 public interface Command {
35     void execute();
36 }
37 
38 public class ConcreteCommandOne implements Command {
39     private Receiver receiver
40 
41     public ConcreteCommandOne(Receiver receiver) {
42         this.receiver = receiver;
43 
44     }
45 
46     public void execute() {
47         receiver.actionOne();
48 
49     }
50 }
51 
52 public class ConcreteCommandTwo implements Command {
```

```
53      private Receiver receiver
54
55      public ConcreteCommandTwo(Receiver receiver) {
56          this.receiver = receiver;
57
58      }
59
60      public void execute() {
61          receiver.actionTwo();
62
63      }
64  }
65
66  public class Receiver {
67      public Receiver() {
68          //
69
70      }
71
72      public void actionOne() {
73          System.out.println("ActionOne has been taken.");
74
75      }
76
77      public void actionTwo() {
78          System.out.println("ActionTwo has been taken.");
79
80      }
81  }
```

为什么使用命令模式

```
1
2  public class Client {
3      public static void main(String[] args) {
4          Receiver receiver = new Receiver();
5          receiver.actionOne();
6          receiver.actionTwo();
```

```
    }
}

public class Receiver {
    public Receiver() {
        //
    }

    public void actionOne() {
        System.out.println("ActionOne has been taken.");
    }

    public void actionTwo() {
        System.out.println("ActionTwo has been taken.");
    }
}
```

(1)我们须要Client和Receiver同时开发,而且在开发过程中分别须要不停重购，改名
(2)如果我们要求Redo ,Undo等功能
(3)我们须要命令不按照调用执行，而是按照执行时的情况排序，执行
(4)在上边的情况下，我们的接受者有很多，不止一个

```
public class Invoker {
    private List cmdList = new ArrayList();

    public Invoker() {
    }

    public add(Command command) {
        cmdList.add(command);
    }

    public remove(Command command) {
        cmdList.remove(command);
    }

    public void action() {
        Command cmd;
```

```
        while ((cmd = getCmd()) != null) {
            log("begin" + cmd.getName());
            cmd.execute();
            log("end" + cmd.getName());
        }
    }

    public Command getCmd() {
        //按照自定义优先级，排序取出cmd
    }
}

public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command commandOne = new ConcreteCommandOne(receiver);
        Command commandTwo = new ConcreteCommandTwo(receiver);
        Invoker invoker = new Invoker();
        invoker.add(commandOne);
        invoker.add(commandTwo);
        iinvoker.action();
    }
}
```

redo undo

```
public class ConcreteCommandOne implements Command {
    private Receiver receiver
    private Receiver lastReceiver;

    public ConcreteCommandOne(Receiver receiver) {
        this.receiver = receiver;
    }

    public void execute() {
        record();
        receiver.actionOne();
    }
```

```
13
14      public void undo() {
15          //恢复状态
16      }
17
18      public void redo() {
19          lastReceiver.actionOne();
20          //
21      }
22
23      public record() {
24          //记录状态
25      }
26  }
```

# 代理模式

## 静态代理

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。
代理模式一般涉及到的角色有：
抽象角色：声明真实对象和代理对象的共同接口；
代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。
真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

```
1 /**
2  * 抽象角色
3  */
4 public abstract class Subject {
5     public abstract void request();
```

```
1 /**
2  * 真实的角色
3  */
```

```java
public class RealSubject extends Subject {

    @Override
    public void request() {
        // TODO Auto-generated method stub

    }

}
```

```java
/**
 * 静态代理，对具体真实对象直接引用
 * 代理角色，代理角色需要有对真实角色的引用，
 * 代理做真实角色想做的事情
 */
public class ProxySubject extends Subject {

    private RealSubject realSubject = null;

    /**
     * 除了代理真实角色做该做的事情，代理角色也可以提供附加操作，
     * 如：preRequest()和postRequest()
     */
    @Override
    public void request() {
        preRequest();  //真实角色操作前的附加操作

        if(realSubject == null){
            realSubject =  new RealSubject();
        }
        realSubject.request();

        postRequest();  //真实角色操作后的附加操作
    }

    /**
     *  真实角色操作前的附加操作
     */
```

```
    private void postRequest() {
        // TODO Auto-generated method stub

    }

    /**
     *  真实角色操作后的附加操作
     */
    private void preRequest() {
        // TODO Auto-generated method stub

    }

}
```

```
/**
 *  客户端调用
 */
public class Main {
    public static void main(String[] args) {
        Subject subject = new ProxySubject();
        subject.request();  //代理者代替真实者做事情
    }
}
```

优点：可以做到在不修改目标对象的功能前提下,对目标功能扩展.
缺点：每一个代理类都必须实现一遍委托类（也就是realsubject）的接口，如果接口增加方法，则代理类也必须跟着修改。其次，代理类每一个接口对象对应一个委托对象，如果委托对象非常多，则静态代理类就非常臃肿，难以胜任。

## jdk动态代理

动态代理解决静态代理中代理类接口过多的问题，通过反射来实现的，借助Java自带的java.lang.reflect.Proxy,通过固定的规则生成。
步骤如下：

1. 编写一个委托类的接口，即静态代理的（Subject接口）
2. 实现一个真正的委托类，即静态代理的（RealSubject类）
3. 创建一个动态代理类，实现InvocationHandler接口，并重写该invoke方法

4. 在测试类中，生成动态代理的对象。

第一二步骤，和静态代理一样,第三步：

```
public class DynamicProxy implements InvocationHandler {
    private Object object;
    public DynamicProxy(Object object) {
        this.object = object;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] ar
gs) throws Throwable {
        Object result = method.invoke(object, args);
        return result;
    }
}
```

创建动态代理的对象

```
Subject realSubject = new RealSubject();
DynamicProxy proxy = new DynamicProxy(realSubject);
ClassLoader classLoader = realSubject.getClass().getClassLoader();
Subject subject = (Subject) Proxy.newProxyInstance(classLoader, ne
w  Class[]{Subject.class}, proxy);
subject.visit();
```

上述代码的关键是Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler)方法，该方法会根据指定的参数动态创建代理对象。三个参数的意义如下：

1. loader，指定代理对象的类加载器；
2. interfaces，代理对象需要实现的接口，可以同时指定多个接口；
3. handler，方法调用的实际处理者，代理对象的方法调用都会转发到这里（*注意1）。

原理：

```
public class Proxy implements java.io.Serializable {
    public static Object newProxyInstance(ClassLoader loader,
                                           Class<?>[] interfaces,
                                           InvocationHandler h)
        throws IllegalArgumentException
    {
        Objects.requireNonNull(h);

        // 准备一份所有被实现的业务接口
        final Class<?>[] intfs = interfaces.clone();
        final SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            checkProxyAccess(Reflection.getCallerClass(), loader,
intfs);
        }

        /*
         * 代理类生成的核心代码
         */
        Class<?> cl = getProxyClass0(loader, intfs);
```

classLoader 的作用是将字节码文件加载进虚拟机并生成相应的 class
interfaces 就是被实现的那些接口
h 就是 InvocationHandler

```
public class Proxy implements java.io.Serializable {
/**
     * a cache of proxy classes
     */
    private static final WeakCache<ClassLoader, Class<?>[], Class
<?>>
        proxyClassCache = new WeakCache<>(new KeyFactory(), new P
roxyClassFactory());

    private static Class<?> getProxyClass0(ClassLoader loader,Cla
ss<?>... interfaces) {

         //代理接口数限制
```

```
        if (interfaces.length > 65535) {
            throw new IllegalArgumentException("interface limit exceeded");
        }

        //如果由给定的加载器实现给定的接口定义的代理类存在，返回缓存;否则，它将通过ProxyClassFactory创建代理类
        return proxyClassCache.get(loader, interfaces);
    }
```

```
//通过ClassLoader和接口列表，生成和定义一个proxy class
    private static final class ProxyClassFactory
        implements BiFunction<ClassLoader, Class<?>[], Class<?>>
    {
        // 所有代理类的命名前缀
        private static final String proxyClassNamePrefix = "$Proxy";

        // 一个唯一的number作为proxy class的名称标识
        private static final AtomicLong nextUniqueNumber = new AtomicLong();
        //proxy class生成方法
        @Override
        public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

            Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>(interfaces.length);
            for (Class<?> intf : interfaces) {
                //确认类加载器解析了这个名字的接口到相同的Class对象。
                Class<?> interfaceClass = null;
                try {
                    interfaceClass = Class.forName(intf.getName(), false, loader);
                } catch (ClassNotFoundException e) {
                }
                if (interfaceClass != intf) {
                    throw new IllegalArgumentException(
```

```
                        intf + " is not visible from class loade
r");
                }
                //确认代理的Class是接口，从这可看出JDK动态代理的劣势
                if (!interfaceClass.isInterface()) {
                    throw new IllegalArgumentException(
                        interfaceClass.getName() + " is not an in
terface");
                }
                //接口重复校验
                if (interfaceSet.put(interfaceClass, Boolean.TRUE
) != null) {
                    throw new IllegalArgumentException(
                        "repeated interface: " + interfaceClass.g
etName());
                }
            }

            String proxyPkg = null;     // 定义proxy class 所在的包
            int accessFlags = Modifier.PUBLIC | Modifier.FINAL;

            //记录所有non-public的 proxy interfaces都在同一个package
中
            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");
                    }
                }
            }
```

```
57
58              if (proxyPkg == null) {
59                  // 如果没有non-public的interfaces，默认包为com.sun.pr
   oxy
60                  proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
61              }
62
63              /*
64               * Choose a name for the proxy class to generate.
65               */
66              long num = nextUniqueNumber.getAndIncrement();
67              //最终大概名字为：com.sun.proxy.$Proxy1
68              String proxyName = proxyPkg + proxyClassNamePrefix +
   num;
69
70              /*
71               * 生成特殊的proxy class.
72               */
73              byte[] proxyClassFile = ProxyGenerator.generateProxyC
   lass(
74                  proxyName, interfaces, accessFlags);
75              try {
76                  return defineClass0(loader, proxyName,
77                                      proxyClassFile, 0, proxyClass
   File.length);
78              } catch (ClassFormatError e) {
79                  /*
80                   * A ClassFormatError here means that (barring bu
   gs in the
81                   * proxy class generation code) there was some ot
   her
82                   * invalid aspect of the arguments supplied to th
   e proxy
83                   * class creation (such as virtual machine limita
   tions
84                   * exceeded).
85                   */
86                  throw new IllegalArgumentException(e.toString());
87              }
88          }
```

```
    }
```

```
public static byte[] generateProxyClass(String arg, Class<?>[] arg0, int arg1) {
      ProxyGenerator arg2 = new ProxyGenerator(arg, arg0, arg1);
      byte[] arg3 = arg2.generateClassFile();
      if(saveGeneratedFiles) {
         AccessController.doPrivileged(new 1(arg, arg3));
      }

      return arg3;
   }
   //私有构造器
   private ProxyGenerator(String arg0, Class<?>[] arg1, int arg2) {
        this.className = arg0;
        this.interfaces = arg1;
        this.accessFlags = arg2;
    }
```

```
private byte[] generateClassFile() {
          //添加从 Object基类中继承的方法
        this.addProxyMethod(hashCodeMethod, Object.class);
        this.addProxyMethod(equalsMethod, Object.class);
        this.addProxyMethod(toStringMethod, Object.class);
        //添加接口中的方法实现
        Class[] arg0 = this.interfaces;
        int arg1 = arg0.length;

        int arg2;
        Class arg3;
        for (arg2 = 0; arg2 < arg1; ++arg2) {
            arg3 = arg0[arg2];
            Method[] arg4 = arg3.getMethods();
            int arg5 = arg4.length;

            for (int arg6 = 0; arg6 < arg5; ++arg6) {
```

```
18                  Method arg7 = arg4[arg6];
19                  this.addProxyMethod(arg7, arg3);
20              }
21          }
22
23          Iterator arg10 = this.proxyMethods.values().iterator();
24
25          List arg11;
26          while (arg10.hasNext()) {
27              arg11 = (List) arg10.next();
28              checkReturnTypes(arg11);
29          }
30
31          Iterator arg14;
32          try {
33                  //构造方法
34              this.methods.add(this.generateConstructor());
35              arg10 = this.proxyMethods.values().iterator();
36
37              while (arg10.hasNext()) {
38                  arg11 = (List) arg10.next();
39                  arg14 = arg11.iterator();
40
41                  while (arg14.hasNext()) {
42                      ProxyMethod arg15 = (ProxyMethod) arg14.next
    ();
43                      this.fields.add(new FieldInfo(this, arg15.me
    thodFieldName, "Ljava/lang/reflect/Method;", 10));
44                      this.methods.add(ProxyMethod.access$100(arg1
    5));
45                  }
46              }
47
48              this.methods.add(this.generateStaticInitializer());
49          } catch (IOException arg9) {
50              throw new InternalError("unexpected I/O Exception",
    arg9);
51          }
52
53          if (this.methods.size() > '□') {
```

```
54              throw new IllegalArgumentException("method limit exc
eeded");
55          } else if (this.fields.size() > '□') {
56              throw new IllegalArgumentException("field limit exce
eded");
57          } else {
58              this.cp.getClass(dotToSlash(this.className));
59              this.cp.getClass("java/lang/reflect/Proxy");
60              arg0 = this.interfaces;
61              arg1 = arg0.length;
62
63              for (arg2 = 0; arg2 < arg1; ++arg2) {
64                  arg3 = arg0[arg2];
65                  this.cp.getClass(dotToSlash(arg3.getName()));
66              }
67
68              this.cp.setReadOnly();
69              //生成文件
70              ByteArrayOutputStream arg12 = new ByteArrayOutputStr
eam();
71              DataOutputStream arg13 = new DataOutputStream(arg12)
;
72
73              try {
74                  arg13.writeInt(-889275714);
75                  arg13.writeShort(0);
76                  arg13.writeShort(49);
77                  this.cp.write(arg13);
78                  arg13.writeShort(this.accessFlags);
79                  arg13.writeShort(this.cp.getClass(dotToSlash(thi
s.className)));
80                  arg13.writeShort(this.cp.getClass("java/lang/ref
lect/Proxy"));
81                  arg13.writeShort(this.interfaces.length);
82                  Class[] arg16 = this.interfaces;
83                  int arg17 = arg16.length;
84
85                  for (int arg18 = 0; arg18 < arg17; ++arg18) {
86                      Class arg21 = arg16[arg18];
87                      arg13.writeShort(this.cp.getClass(dotToSlash
```

```
    (arg21.getName()));
                }

                arg13.writeShort(this.fields.size());
                arg14 = this.fields.iterator();

                while (arg14.hasNext()) {
                    FieldInfo arg19 = (FieldInfo) arg14.next();
                    arg19.write(arg13);
                }

                arg13.writeShort(this.methods.size());
                arg14 = this.methods.iterator();

                while (arg14.hasNext()) {
                    MethodInfo arg20 = (MethodInfo) arg14.next()
;
                    arg20.write(arg13);
                }

                arg13.writeShort(0);
                //返回字节流
                return arg12.toByteArray();
            } catch (IOException arg8) {
                throw new InternalError("unexpected I/O Exceptio
n", arg8);
            }
        }
    }
```

生成的proxy.class

```
import com.proxy.api.PeopleService;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;
/**
```

```
 * 生成的类继承了Proxy,实现了要代理的接口PeopleService
 *
 */
public final class $Proxy11
  extends Proxy
  implements PeopleService
{
  private static Method m1;
  private static Method m3;
  private static Method m2;
  private static Method m4;
  private static Method m0;

  public $proxy11(InvocationHandler paramInvocationHandler)
  {
    //调用基类Proxy的构造器
    super(paramInvocationHandler);
  }

  public final boolean equals(Object paramObject)
  {
    try
    {
      return ((Boolean)this.h.invoke(this, m1, new Object[] { paramObject })).booleanValue();
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }

  public final void mainJiu()
  {
    try
    {
```

```
46        this.h.invoke(this, m3, null);
47        return;
48      }
49      catch (Error|RuntimeException localError)
50      {
51        throw localError;
52      }
53      catch (Throwable localThrowable)
54      {
55        throw new UndeclaredThrowableException(localThrowable);
56      }
57    }
58
59    public final String toString()
60    {
61      try
62      {
63        return (String)this.h.invoke(this, m2, null);
64      }
65      catch (Error|RuntimeException localError)
66      {
67        throw localError;
68      }
69      catch (Throwable localThrowable)
70      {
71        throw new UndeclaredThrowableException(localThrowable);
72      }
73    }
74
75    public final void printName(String paramString)
76    {
77      try
78      {
79        this.h.invoke(this, m4, new Object[] { paramString });
80        return;
81      }
82      catch (Error|RuntimeException localError)
83      {
84        throw localError;
85      }
```

```
86      catch (Throwable localThrowable)
87      {
88        throw new UndeclaredThrowableException(localThrowable);
89      }
90    }
91
92    public final int hashCode()
93    {
94      try
95      {
96        return ((Integer)this.h.invoke(this, m0, null)).intValue()
;
97      }
98      catch (Error|RuntimeException localError)
99      {
100       throw localError;
101     }
102     catch (Throwable localThrowable)
103     {
104       throw new UndeclaredThrowableException(localThrowable);
105     }
106   }
107
108   static
109   {
110     try
111     {
112     //利用反射生成5个方法，包括Object中的equals、toString、hashCode以及
PeopleService中的mainJiu和printName
113       m1 = Class.forName("java.lang.Object").getMethod("equals",
new Class[] { Class.forName("java.lang.Object") });
114       m3 = Class.forName("com.proxy.api.PeopleService").getMetho
d("mainJiu", new Class[0]);
115       m2 = Class.forName("java.lang.Object").getMethod("toStrin
g", new Class[0]);
116       m4 = Class.forName("com.proxy.api.PeopleService").getMetho
d("printName", new Class[] { Class.forName("java.lang.String") }
);
117       m0 = Class.forName("java.lang.Object").getMethod("hashCod
e", new Class[0]);
```

```
118       return;
119     }
120     catch (NoSuchMethodException localNoSuchMethodException)
121     {
122       throw new NoSuchMethodError(localNoSuchMethodException.get
    Message());
123     }
124     catch (ClassNotFoundException localClassNotFoundException)
125     {
126       throw new NoClassDefFoundError(localClassNotFoundException
    .getMessage());
127     }
128   }
129 }
```

现在再看生成的$Proxy.class反编译的java代码中，调用mainJiu时会this.h.invoke(this, m3, null);调用Proxy类中InvocationHandler的invoke方法

newProxyInstance()通过反射生成含有接口方法的proxy class（继承了Proxy类，实现了需要代理的接口）
因此对该对象的所有方法调用都会转发InvocationHandler.invoke()方法

## cglib动态代理

假设我们有一个没有实现任何接口的类HelloConcrete

```
1 public class HelloConcrete {
2     public String sayHello(String str) {
3         return "HelloConcrete: " + str;
4     }
5 }
```

```
1 // CGLIB动态代理
2 // 1. 首先实现一个MethodInterceptor，方法调用会被转发到该类的intercept()
  方法。
3 class MyMethodInterceptor implements MethodInterceptor{
4   ...
5     @Override
```

```
    public Object intercept(Object obj, Method method, Object[] a
rgs, MethodProxy proxy) throws Throwable {
        logger.info("You said: " + Arrays.toString(args));
        return proxy.invokeSuper(obj, args);
    }
}
// 2. 然后在需要使用HelloConcrete的时候，通过CGLIB动态代理获取代理对象。
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(HelloConcrete.class);
enhancer.setCallback(new MyMethodInterceptor());

HelloConcrete hello = (HelloConcrete)enhancer.create();
System.out.println(hello.sayHello("I love you!"));
```

通过CGLIB的Enhancer来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用create()方法得到代理对象，对这个对象所有非final方法的调用都会转发给MethodInterceptor.intercept()方法，在intercept()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；通过调用MethodProxy.invokeSuper()方法，我们将调用转发给原始对象，具体到本例，就是HelloConcrete的具体方法。
对于从Object中继承的方法，CGLIB代理也会进行代理，如hashCode()、equals()、toString()等，但是getClass()、wait()等方法不会，因为它是final方法，CGLIB无法代理。

原理：
CGLIB是一个强大的高性能的代码生成包，底层是通过使用一个小而快的字节码处理框架ASM，它可以在运行期扩展Java类与实现Java接口
Enhancer是CGLIB的字节码增强器，可以很方便的对类进行拓展
创建代理对象的几个步骤:
1、生成代理类的二进制字节码文件
2、加载二进制字节码，生成Class对象( 例如使用Class.forName()方法 )
3、通过反射机制获得实例构造，并创建代理类对象

## 关系

## 总结

1. jdk动态代理：利用拦截器(拦截器必须实现InvocationHanlder)加上反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。只能对实现了接口的类生成代理只能对实现了接口的类生成代理

2. cglib：利用ASM开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。主要是对指定的类生成一个子类，覆盖其中的方法，并覆盖其中方法实现增强，但是因为采用的是继承，对于final类或方法，是无法继承的。
3. 选择
    a. 如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。
    b. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP。
    c. 如果目标对象没有实现了接口，必须采用CGLIB库，Spring会自动在JDK动态代理和CGLIB之间转换。

尚硅谷