* 学习目标

* 能够理解代理模式概述

  * 经纪人，中介

  * 额外添加功能

* 能够掌握代理模式之静态代理

* 能够掌握代理模式之动态代理中的JDK代理

* 能够掌握代理模式之动态代理中的子类代理

* 能够掌握自定义连接池

---

* 回顾

  * 存储过程

    create or replace procedure name()

    as

    begin

    end;

  * CallableStatement

    * {call name(?,?)}

  * 存储函数

    * function,返回值

    * { ? =call name(?.?)}

  * 程序包

    * create package pac1

    * create package body pac1

    * pac1.存储名字

  * 触发器

* dml触发的时候：trigger

* 序列自动增长

* ：old.name,:new.name

* 视图

* 虚拟表

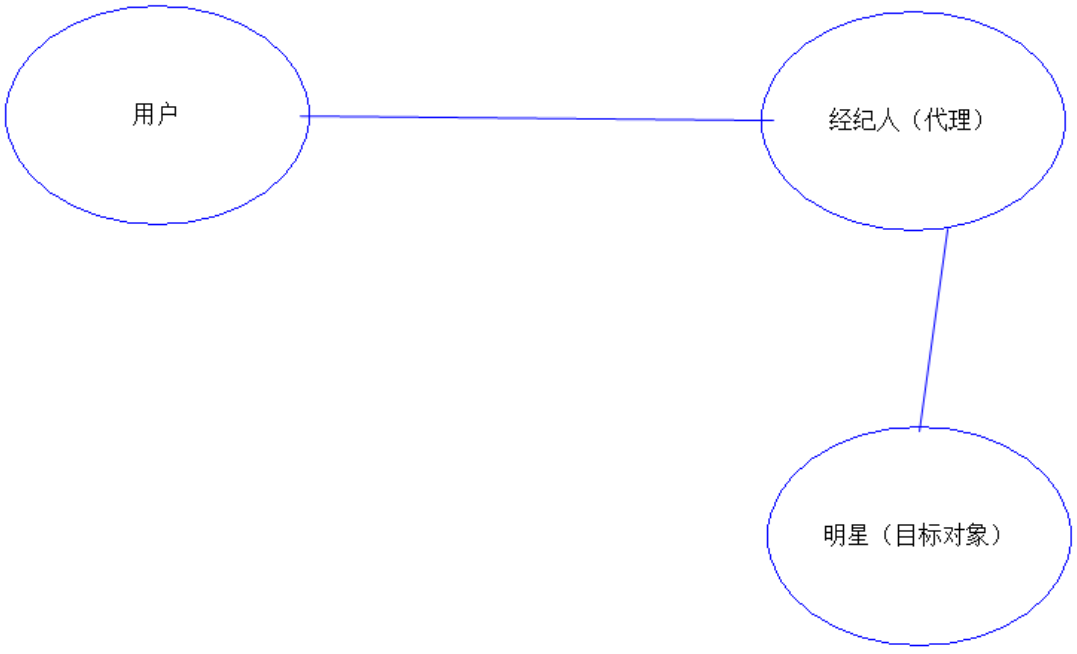
* 安全性，简单性，隔离性，角度性

* 数据库闪回，备份，恢复

* 能够理解代理模式概述

* 生活代理：经纪人

* 假设我们想邀请一位明星,那么并不是直接联系明星,而是联系明星的经纪人,来达到同样的目的.明星就是一个目标对象,他只要负责活动中的节目,而其他琐碎的事情就交给他的代理人(经纪人)来解决.这就是代理思想在现实中的一个例子。



* 代理模式概述

* 代理(Proxy)是一种设计模式,提供了对目标对象另外的访问方式;即通过代理对象访问目标对象.

* 代理模式作用

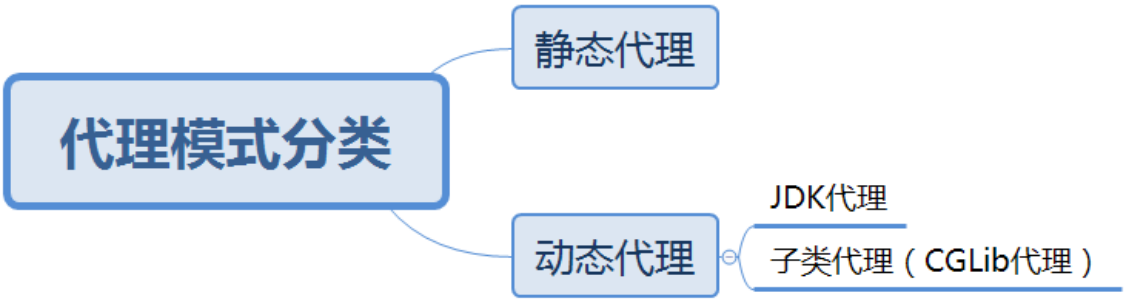
  * 可以在目标对象实现的基础上,增强额外的功能操作，即扩展目标对象的功能

  * 温馨提醒：

    * 在编程中不要随意去修改别人已经写好的代码或者方法

    * 如果需改修改,可以通过代理的方式来扩展该方法

* 代理模式的分类



* 能够掌握代理模式之静态代理

  * 静态代理概述

  * 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.

  * 静态代理在使用时：

    * 需要定义接口

      * 被代理对象（目标对象）与代理对象一起实现相同的接口。

```
* 业务需求：
  * 添加一个用户到数据库
  * 在添加用户需求上，额外添加一个记录日志
* 准备
 * 表，序列，触发器的创建
```

```
drop table t_user;
create table t_user(
   id number primary key,
   name varchar2(50),
   password varchar2(50)
);
create sequence seq_user;
create trigger tri_user_id
   before insert on t_user for each row
begin
   select seq_user.nextval into :new.id from dual;
end;
insert into t_user(name,password) values('xiaohei','123');
select * from t_user;

 * 项目搭建
   * 数据库连接环境
   * 日志的环境
 * 添加一个用户到数据库
  * 实体类
  User:id，name，password
  * UserDao
  public interface UserDao {
    boolean add(User user) throws SQLException;
 }
 * UserDaoImpl
 public class UserDaoImpl implements UserDao{

    @Override
    public boolean add(User user) throws SQLException {
        Connection conn = ConnectionUtils.getConn();
        String sql="insert into t_user(name,password) values(?,?)";
        PreparedStatement pst=null;
        try {
            pst = conn.prepareStatement(sql);
            pst.setString(1, user.getName());
            pst.setString(2, user.getPassword());
            int result=pst.executeUpdate();
            return result>0;
        } catch (SQLException e) {
```

```
            e.printStackTrace();
            throw new SQLException("添加失败");
        }finally {
            ConnectionUtils.close(conn, pst, null);
        }
    }
}
 * 单元测试
@Test
    public void test1() throws SQLException {
         UserDao userDao=new  UserDaoImpl();
         User user=new User();
         user.setName("刘备");
         user.setPassword(MD5Utils.md5("123"));
         userDao.add(user);
    }

* 在添加用户需求上，额外添加一个记录日志
  * 封装L工具类
 public class L {
    public static Logger logger = LogManager.getLogger("mylog");

    public static void t(String msg) {
        logger.trace(msg);
    }
    public static void d(String msg) {
        logger.debug(msg);
    }
    public static void i(String msg) {
        logger.info(msg);
    }
    public static void w(String msg) {
        logger.warn(msg);
    }
    public static void e(String msg) {
        logger.error(msg);
    }
}
 * 添加静态代理类
 public class UserDaoStaticProxy implements UserDao{
```

```
    private UserDao userDao=new UserDaoImpl();
    @Override
    public boolean add(User user) {
        L.d("开始add User");
        try {
            userDao.add(user);
        } catch (SQLException e) {
            e.printStackTrace();
            L.e("添加user出问题了:"+e.getMessage());
        }
        L.d("结束add User");
        return false;
    }
}
 * 单元测试
  @Test
    public void test2() throws SQLException {
         UserDao userDao=new UserDaoStaticProxy();
         User user=new User();
         user.setName("刘备");
         user.setPassword(MD5Utils.md5("123"));
         userDao.add(user);
    }

 * 静态代理的问题
  * 需要建很多代理类
    * UserDaoImpl---UserDaoImplProxy
    * PetDaoImpl---PetDaoImplProxy
     ...
  * 一旦接口增加方法,目标对象与代理对象都要维护.
```

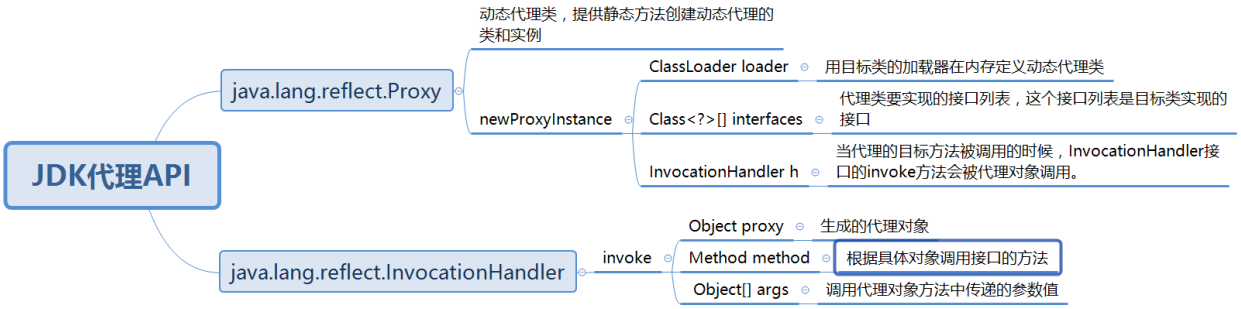
* 能够掌握代理模式之动态代理中的JDK代理

  * 动态代理特点

    * 动态代理也叫做:JDK代理,接口代理

    * 代理对象是由JDK代理API动态的在内存中构建的

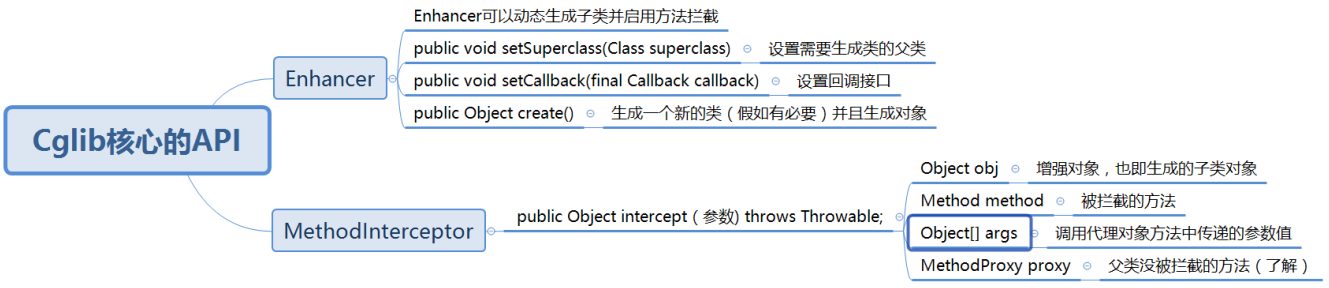
  * JDK代理API




```
* 代理对象工厂的创建
public class DaoProxyFactory {
    private Object target;

    public DaoProxyFactory(Object target) {
        this.target=target;
    }

    /**
     * @return
     * 获得Dao的代理对象
     */
    public Object getDaoProxy() {

        Object proxy=Proxy.newProxyInstance(target.getClass().getClassLoader(),
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) th
                L.d(method.getName()+"执行之前");
                Object result=null;
                try{
                     result=method.invoke(target, args);
                }catch (Exception e) {
                    e.printStackTrace();
                    L.e("出现问题："+e.getMessage());
                }
                L.d(method.getName()+"执行之后");
                return result;
```

```
            }
        });
        return proxy;
    }
}

* 测试
@Test
    public void test3() throws SQLException {
        DaoProxyFactory daoProxy=new DaoProxyFactory(new UserDaoImpl());
        UserDao userDao=(UserDao) daoProxy.getDaoProxy();
        User user=new User();
        user.setName("关羽");
        user.setPassword(MD5Utils.md5("123"));
        userDao.add(user);
    }
* 断点提示
* 存在问题
 * 静态代理和动态代理模式都是要求目标对象是实现一个接口的目标对象，假如目标对象没有实现
```

* 能够掌握代理模式之动态代理中的子类代理
  * 子类代理（cglib代理）概述
    * 通过目标对象子类的方式实现代理,这种方法就叫做:子类代理(Cglib代理)
    * 子类代理(Cglib代理)是在内存中构建一个子类对象从而实现对目标对象功能的扩展
    * Cglib是一个强大的高性能的代码生成包,它可以在运行期扩展java类与实现java接口
  * Cglib核心的API



```
* 代理工厂的创建
public class DaoProxyFactory2 {
    private Object target;

    public DaoProxyFactory2(Object target) {
        this.target=target;
    }

    /**
     * @return
     * 获得Dao的代理对象
     */
    public Object getDaoProxy() {
        Enhancer enhancer=new Enhancer();
        enhancer.setSuperclass(target.getClass());
        enhancer.setCallback(new MethodInterceptor() {

            @Override
            public Object intercept(Object obj, Method method, Object[] args, M
                L.d(method.getName()+"执行之前");
                Object result=null;
                try{
                     result=method.invoke(target, args);
                }catch (Exception e) {
                    e.printStackTrace();
                    L.e("出现问题："+e.getMessage());
                }
                L.d(method.getName()+"执行之后");
                return result;
            }
        });
        Object obj = enhancer.create();
        return obj;
    }
}

* 测试
    @Test
    public void test4() throws SQLException {
         DaoProxyFactory2 daoProxy=new DaoProxyFactory2(new UserDaoImpl());
```

```
41          UserDao userDao=(UserDao) daoProxy.getDaoProxy();
42          User user=new User();
43          user.setName("关羽123");
44          user.setPassword(MD5Utils.md5("123"));
45          userDao.add(user);
46      }
47
```

* JDK代理与Cglib代理的对比

  * JDK动态代理的原理是根据定义好的规则，用传入的接口创建一个新类

  * CGLib采用的是用创建一个继承实现类的子类，用asm库动态修改子类的代码来实现的

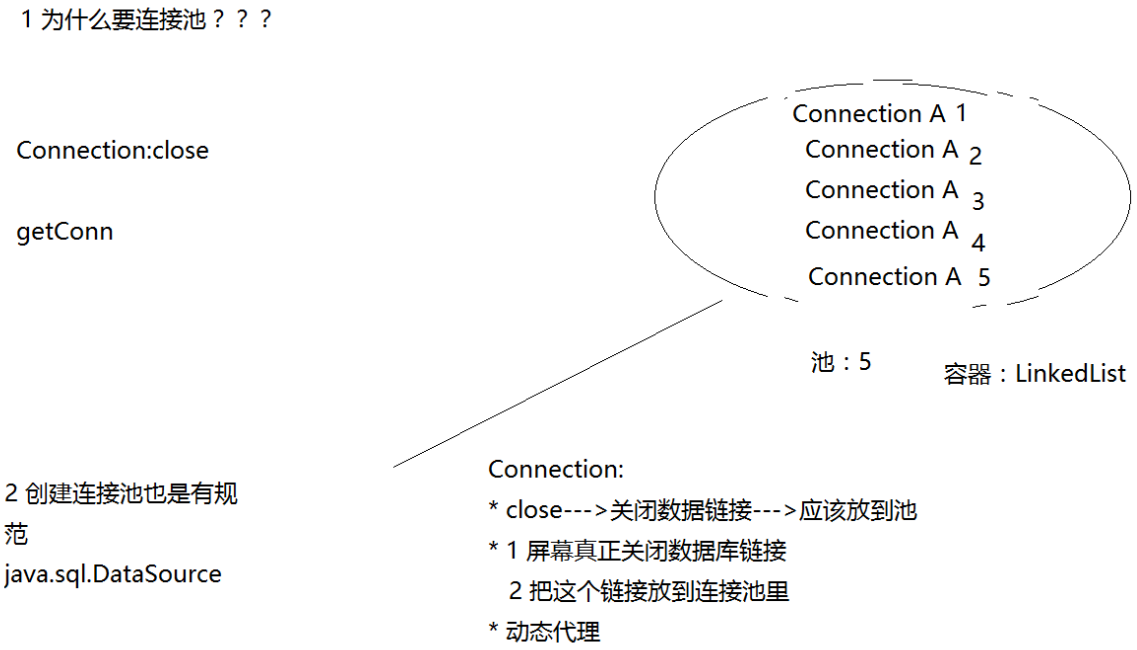
  * CGLib创建的动态代理对象性能比JDK创建的动态代理对象的性能高不少

  * CGLib在创建代理对象时所花费的时间却比JDK多得多

    * 适合单例

  * CGLib由于是采用动态创建子类的方法，对于final方法，无法进行代理

* 能够掌握自定义连接池



```
* 自定义DataSourceAdapter类
public abstract class DataSourceAdapter implements DataSource{
    @Override
    public PrintWriter getLogWriter() throws SQLException {
        return null;
    }
    @Override
    public void setLogWriter(PrintWriter out) throws SQLException {
    }
    @Override
    public void setLoginTimeout(int seconds) throws SQLException {
    }
    @Override
    public int getLoginTimeout() throws SQLException {
        return 0;
    }
    @Override
    public Logger getParentLogger() throws SQLFeatureNotSupportedException {
        return null;
    }
    @Override
    public <T> T unwrap(Class<T> iface) throws SQLException {
        return null;
    }
    @Override
    public boolean isWrapperFor(Class<?> iface) throws SQLException {
        return false;
    }
    @Override
    public Connection getConnection(String username, String password) throws SQ
        return null;
    }
}

* 自定义NBDataSource
public class NBDataSource extends DataSourceAdapter{
    public static int size=5;
    public static LinkedList<Connection> pools=new LinkedList<Connection>();
    private static String username;
    private static String password;
```

```
41      private static String url;
42      static {
43          Properties prop=new Properties();
44          try {
45              prop.load(NBDataSource.class.getClassLoader().getResourceAsStream('
46              username=prop.getProperty("username");
47              password=prop.getProperty("password");
48              url=prop.getProperty("url");
49          } catch (IOException e) {
50              e.printStackTrace();
51          }
52          for (int i = 0; i < size; i++) {
53              try {
54                  pools.addFirst(DriverManager.getConnection(url, username, passw
55              } catch (SQLException e) {
56                  e.printStackTrace();
57              }
58          }
59          System.out.println("数据库连接池链接数："+pools.size());
60      }
61      @Override
62      public Connection getConnection() throws SQLException {
63          Connection conn = pools.removeLast();
64          System.out.println("获取完数据库连接之后链接数："+pools.size());
65          Connection proxyConn=(Connection) Proxy.newProxyInstance(conn.getClass(
66
67              @Override
68              public Object invoke(Object proxy, Method method, Object[] args) th
69                  if("close".equals(method.getName())) {
70  //                      method.invoke(conn, args);
71                      pools.addFirst(conn);
72                      System.out.println("释放完数据库连接之后链接数："+pools.size(
73                  }else {
74                      return method.invoke(conn, args);
75                  }
76                  return null;
77              }
78          });
79          return proxyConn;
80      }
```

```
81
82 }
83
84 * 自定义链接工具类
85 public class ConnectionUtilsNB {
86     public static DataSource dataSource=new NBDataSource();
87     // 获得链接
88     public static Connection getConn() {
89         try {
90             return dataSource.getConnection();
91         } catch (SQLException e) {
92             e.printStackTrace();
93         }
94         return null;
95     }
96     // 释放资源
97     public static void close(Connection con,Statement st,ResultSet rs) {
98         if(rs!=null) {
99             try {
100                 rs.close();
101             } catch (SQLException e) {
102                 e.printStackTrace();
103             }
104         }
105         if(st!=null) {
106             try {
107                 st.close();
108             } catch (SQLException e) {
109                 e.printStackTrace();
110             }
111         }
112         if(con!=null) {
113             try {
114                 con.close();
115             } catch (SQLException e) {
116                 e.printStackTrace();
117             }
118         }
119     }
120
```

```
    // 关闭连接
    public static void closeConn(Connection conn) {
        close(conn, null, null);
    }

    // 关闭SQL执行对象
    public static void closeSt(Statement st) {
        close(null, st, null);
    }

    //关闭结果集
    public static void closeRs(ResultSet rs) {
        close(null, null, rs);
    }

}
public class UserDaoImpl implements UserDao{

    @Override
    public boolean addUser(User user) throws SQLException {
        Connection conn = ConnectionUtilsNB.getConn();
        PreparedStatement pst =null;
        String sql="insert into t_user(name,password) values(?,?)";
        try {
            pst = conn.prepareStatement(sql);
            pst.setString(1, user.getName());
            pst.setString(2, user.getPassword());
            int result=pst.executeUpdate();
            return result>0;
        } finally {
            ConnectionUtils.close(conn, pst, null);
        }
    }
}
```