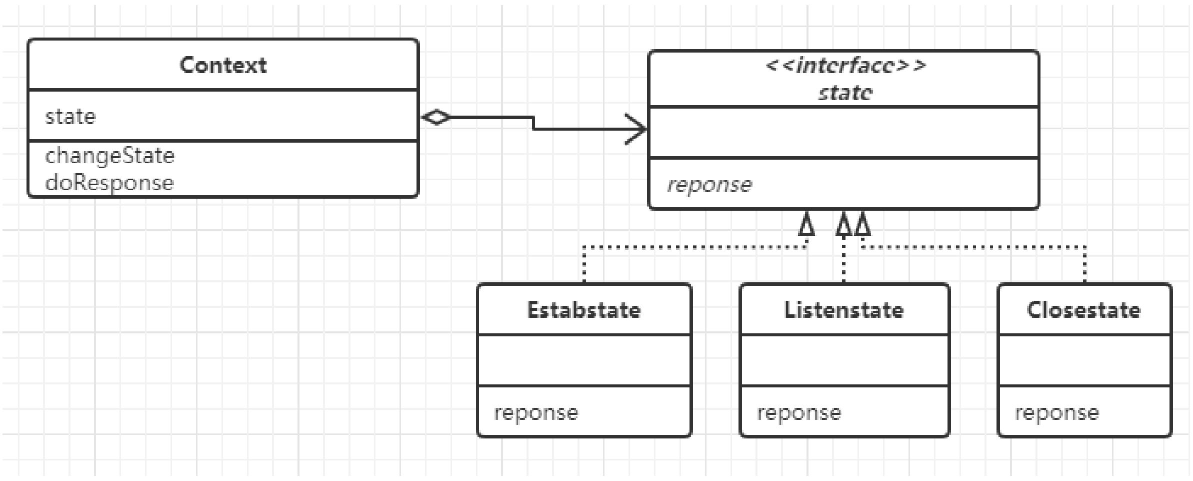
1. 状态模式

在某网络管理软件中， TCP 连接（TCP Connection）具有建立（Established）、监听（Listening）、关闭（Closed）等多种状态，在不同的状态下 TCP 连接对象具有不同的行为，连接对象还可以从一个状态转换到另一个状态。当一个连接对象收到其他对象的请求时，它根据自身的当前状态做出不同的反应。现采用状态模式对 TCP 连接进行设计，绘制对应的类图并编程模拟实现。



```
package state_moudle;

public interface State {
    public void response();
}
```

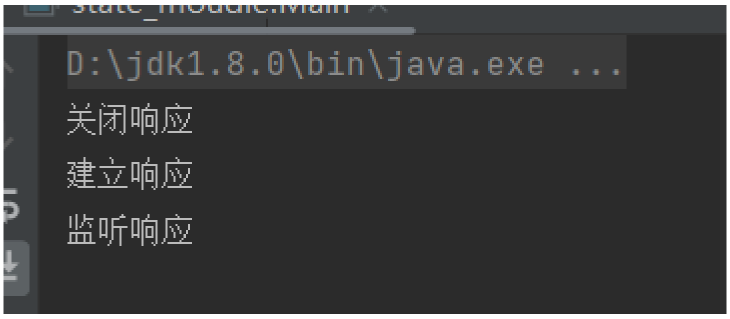
```
package state_moudle;

public class Listenstate implements State{
    @Override
    public void response() {
        System.out.println("监听响应");
    }
}
```

```
package state_moudle;

public class Closestate implements State{
    @Override
    public void response() {
        System.out.println("关闭响应");
    }
}
```

```java
package state_moudle;

public class Estabstate implements State{
    @Override
    public void response() {
        System.out.println("建立响应");
    }
}
```

```java
package state_moudle;

public class Context {
    State state;
    public void changeState(State state){
        this.state=state;
    }
    public void doResponse(){
        state.response();
    }
}
```

```java
package state_moudle;

public class Main {
    public static void main(String[] args) {
        State[] s=new State[3];
        s[0]=new Closestate();
        s[1]=new Estabstate();
        s[2]=new Listenstate();
        Context c=new Context();

        for(int i=0;i<3;i++){
            c.changeState(s[i]);
            c.doResponse();
        }
    }
}
```

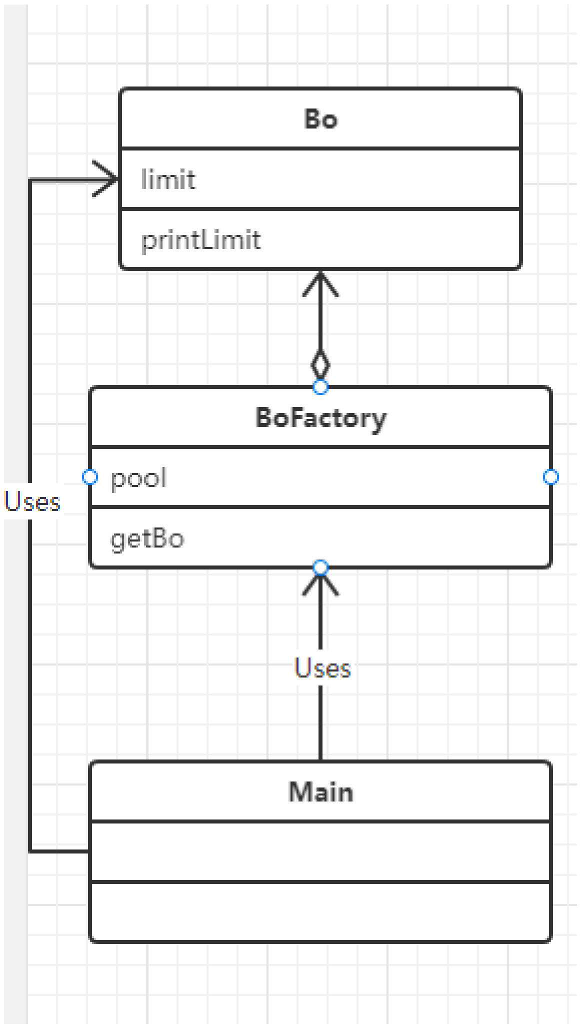


```
D:\jdk1.8.0\bin\java.exe ...
关闭响应
建立响应
监听响应
```

2. 享元模式

某 OA 系统采用享元模式设计权限控制与管理模块，在该模块中，将与系统功能相对应的业务类设计为享元类并将相应的业务对象存储到享元池中（提示；可使用Map 实现，key 为业务对象对应的权限编码， value 为业务对象）。用户身份验证成功后，系统通过存储在数据库中的该用户的权限编码集从享元池获取相应的业务对象并构建权限列表，在界面上显示用户所拥有的权限。根据以上描述，绘制对应

的类图并编程模拟实现。



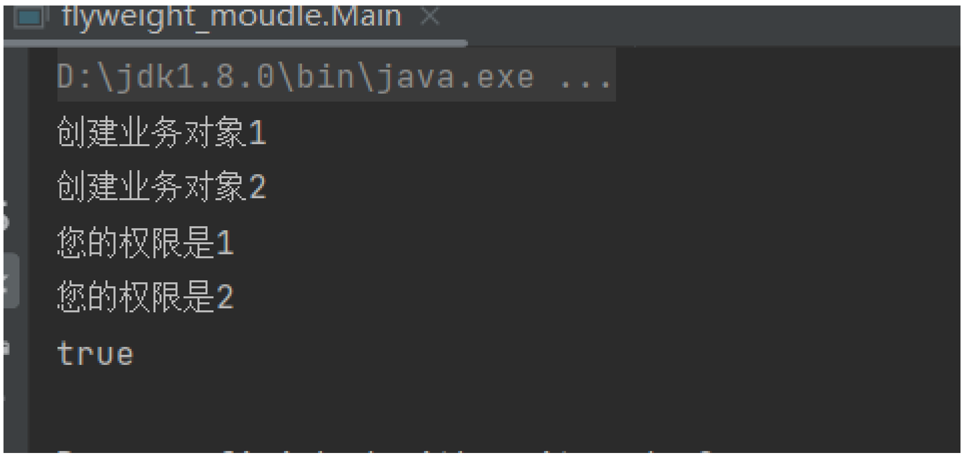
```java
package flyweight_moudle;

public class Bo {
    String limit;
    public Bo(String limit){
        this.limit=limit;
    }
    public void printLimit(){
        System.out.println("您的权限是"+limit);
    }

}
```

```java
package flyweight_moudle;

import java.util.HashMap;
import java.util.Map;

public class BoFactory {
    private Map<Integer,Bo> pool=new HashMap<Integer,Bo>();

    public Bo getBo(int i){
        Bo bo=pool.get(i);
        if(bo==null){
            bo=new Bo(String.valueOf(i));
```

```java
                pool.put(i,bo);
            }
            return bo;
        }

    }
```

```java
package flyweight_moudle;

public class Main {
    public static void main(String[] args) {
        BoFactory bf=new BoFactory();
        Bo b1=bf.getBo(1);
        Bo b2=bf.getBo(2);
        b1.printLimit();
        b2.printLimit();
        Bo b3=bf.getBo(1);
        System.out.println(b1==b3);

    }
}
```

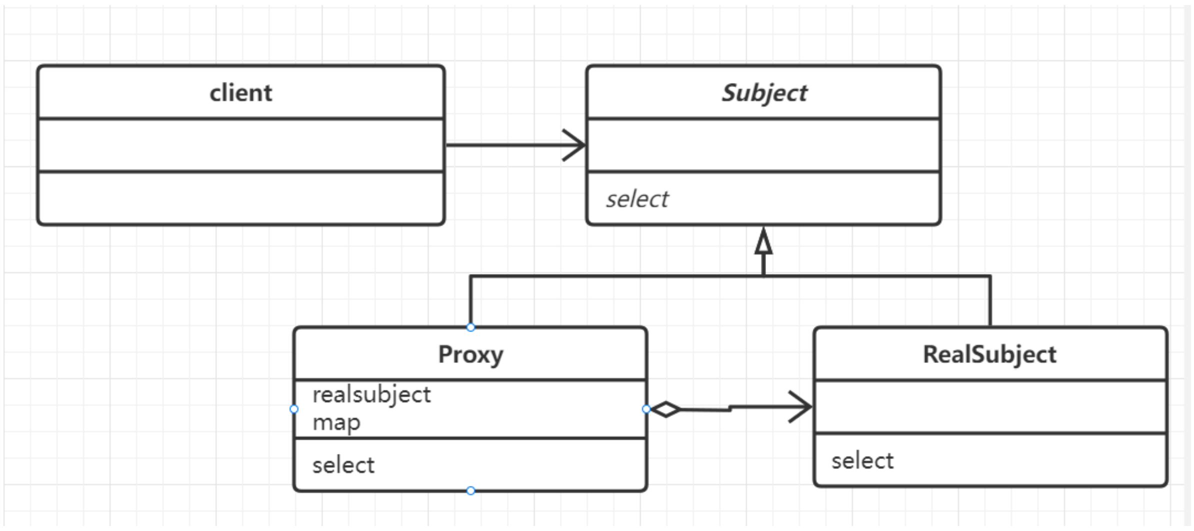
运行main结果



```
D:\jdk1.8.0\bin\java.exe ...
创建业务对象1
创建业务对象2
您的权限是1
您的权限是2
true
```

3. 代理模式

在某电子商务系统中，为了提高查询性能，需要将一些频繁查询的数据保存到内存的辅助存储对象中（提示：可使用 Map 实现）。用户在执行查询操作时，先判断辅助存储对象中是否存在待查询的数据，如果不存在，则通过数据操作对象查询并返回数据，然后将效据保存到辅助存储对象中，否则直接返回存储在辅助存储对象中的数据。现采用代理模式中的缓冲代理实现该功能，要求绘制对应的类图并编程模拟实现。



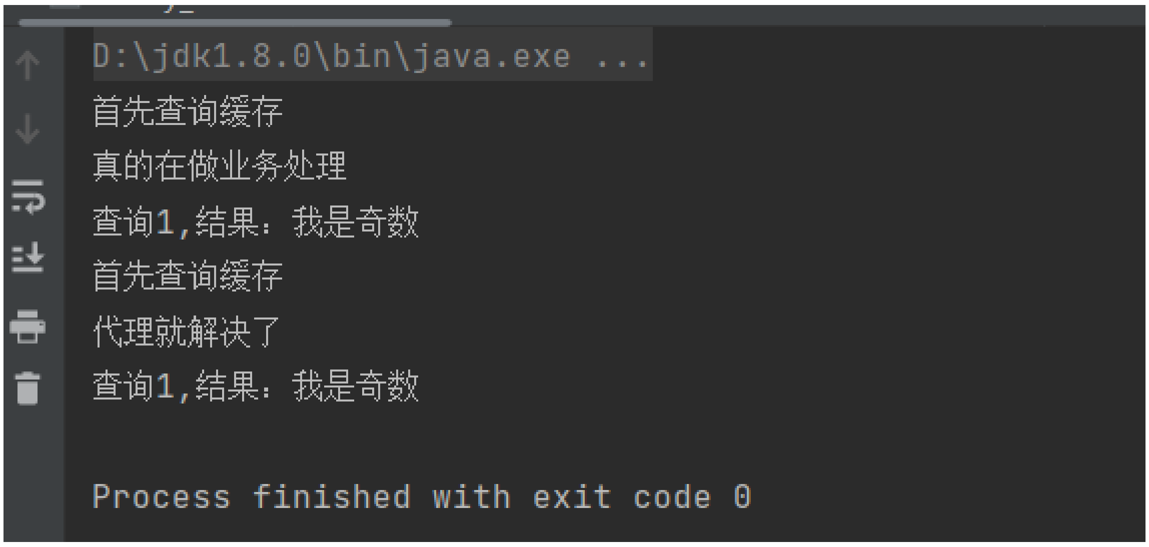
```java
package Proxy_moudle;

public abstract class Subject {
    public abstract String select(int i);
}
```

```java
package Proxy_moudle;

public class RealSubject extends Subject{

    @Override
    public String select(int i) {
        System.out.println("真的在做业务处理");
        if(i%2==0){
            return "我是偶数";
        }else {
            return "我是奇数";
        }
    }
}
```

```java
package Proxy_moudle;

import java.util.HashMap;
import java.util.Map;

public class Proxy extends Subject{
    private RealSubject realsubject=new RealSubject();
    private Map<Integer,String> map=new HashMap<Integer,String>();
    @Override
    public String select(int i) {
        System.out.println("首先查询缓存");
        String res=map.get(i);
        if(res==null){
            res=realsubject.select(i);
            map.put(i,res);
        }else {
            System.out.println("代理就解决了");
        }
        System.out.printf("查询%d,结果: %s\n",i,res);
        return res;
    }
}
```

```
package Proxy_moudle;

public class Main {
    public static void main(String[] args) {
        Subject s=new Proxy();
        s.select(1);
        s.select(1);
    }


}
```

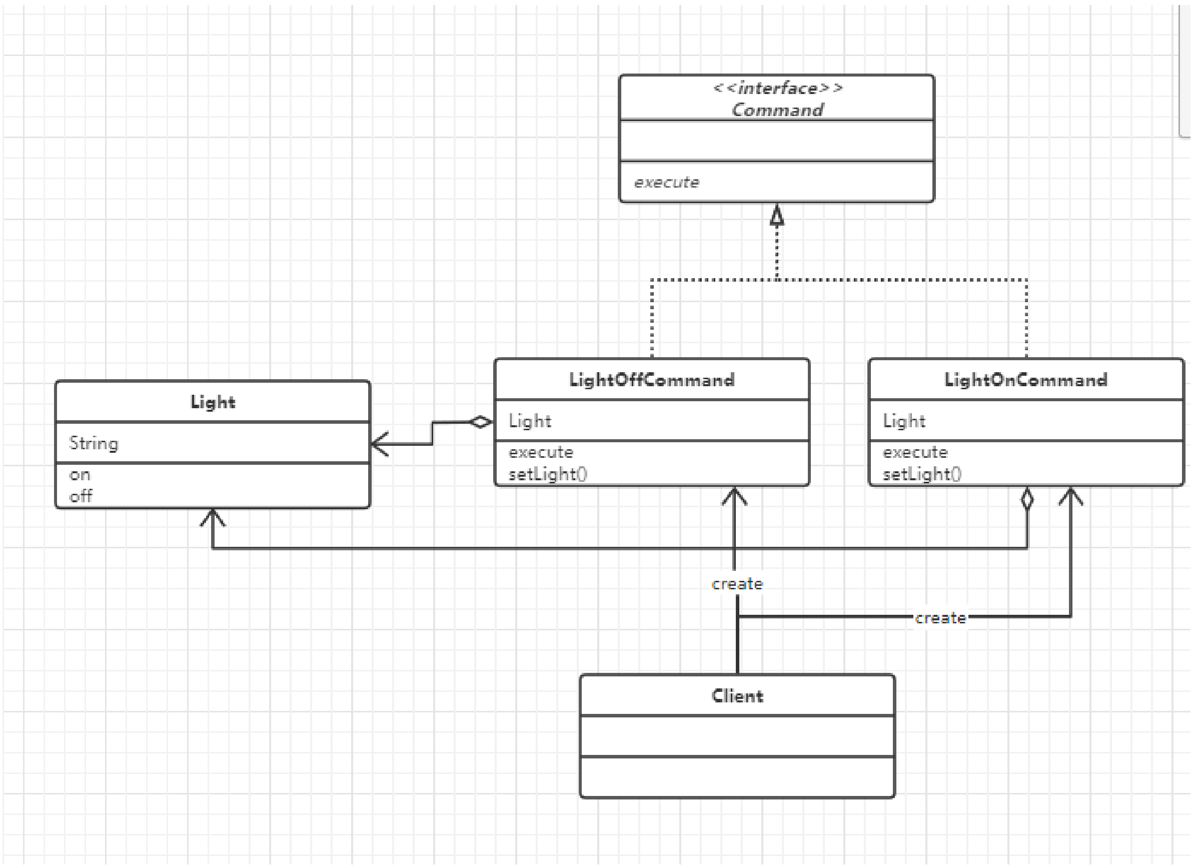
运行main:



```
D:\jdk1.8.0\bin\java.exe ...
首先查询缓存
真的在做业务处理
查询1,结果: 我是奇数
首先查询缓存
代理就解决了
查询1,结果: 我是奇数

Process finished with exit code 0
```

4. 命令模式

某灯具厂商要生产一个智能灯具遥控器，该遥控器具有 5 个可编程的插槽，每个插槽都有一个控制灯具的开关，这5个开关可以通过蓝牙技术控制5个不同房间灯光的打开和关闭，用户可以自行设置每一个开关所对应的房间。现采用命令模式实现该智能遥控器的软件部分，绘制对应的类图并编程模拟实现。



可以将插槽上的开关对应为命令，灯具去执行命令。

```
package command_moudle;

public interface Command {
    public void execute();
}
```

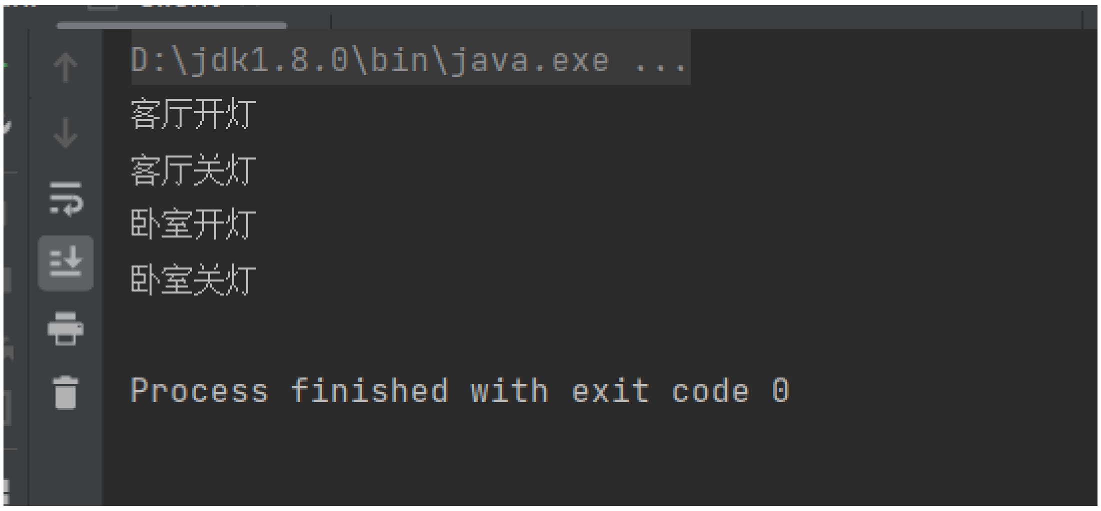
```
package command_moudle;

public class LightOffCommand implements Command{
    public Light light;
    public void setLight(Light l){
        this.light=l;
    }

    @Override
    public void execute() {
        this.light.lightOff();
    }
}
```

```
package command_moudle;


public class LightOnCommand implements Command{

    public Light light;
    public void setLight(Light l){
        this.light=l;
    }

    @Override
    public void execute() {
        this.light.lightOn();
    }
}
```

```
package command_moudle;

public class Light {
    String name;
    public Light(String name){
        this.name=name;
    }

    public void lightOn(){
        System.out.println(name+"开灯");
    }
    public void lightOff(){
        System.out.println(name+"关灯");
    }
}
```

```
package command_moudle;

public class Client {
    public static void main(String[] args) {
        Light l1=new Light("客厅");
        Light l2=new Light("卧室");
        LightOnCommand c1=new LightOnCommand();
        c1.setLight(l1);
        LightOffCommand c2=new LightOffCommand();
        c2.setLight(l1);

        LightOnCommand c3=new LightOnCommand();
        c3.setLight(l2);
        LightOffCommand c4=new LightOffCommand();
        c4.setLight(l2);

        c1.execute();
        c2.execute();
        c3.execute();
        c4.execute();

    }
}
```

运行main结果



```
D:\jdk1.8.0\bin\java.exe ...
客厅开灯
客厅关灯
卧室开灯
卧室关灯

Process finished with exit code 0
```

5. 解释器模式

某软件公司要为数据库备份和同步开发一套简单的数据库同步指令，通过指令可以对数 据库中的数据和结构进行备份。例如，输入指令“ COPY VIEW FROM srcDB TO desDB”，表示将数据库 srcDB 中的所有视图（View）对象都拷贝至数据库desDB； 输人指令“ MOVETABLE Student FROM srcDB TO desDB ”，表示将数据库 srcDB 中的Student 表移动至数据库 desDB 。现使用解释器模式来设计并编程模拟实现该数据库同步指令系统。

指令的语法比较简单，因为没有重复的需求，所以其实用一个非终结符展开式就能满足样例的需求了，但是要考虑到这些指令可能会有扩充的需求，于是在此设计：

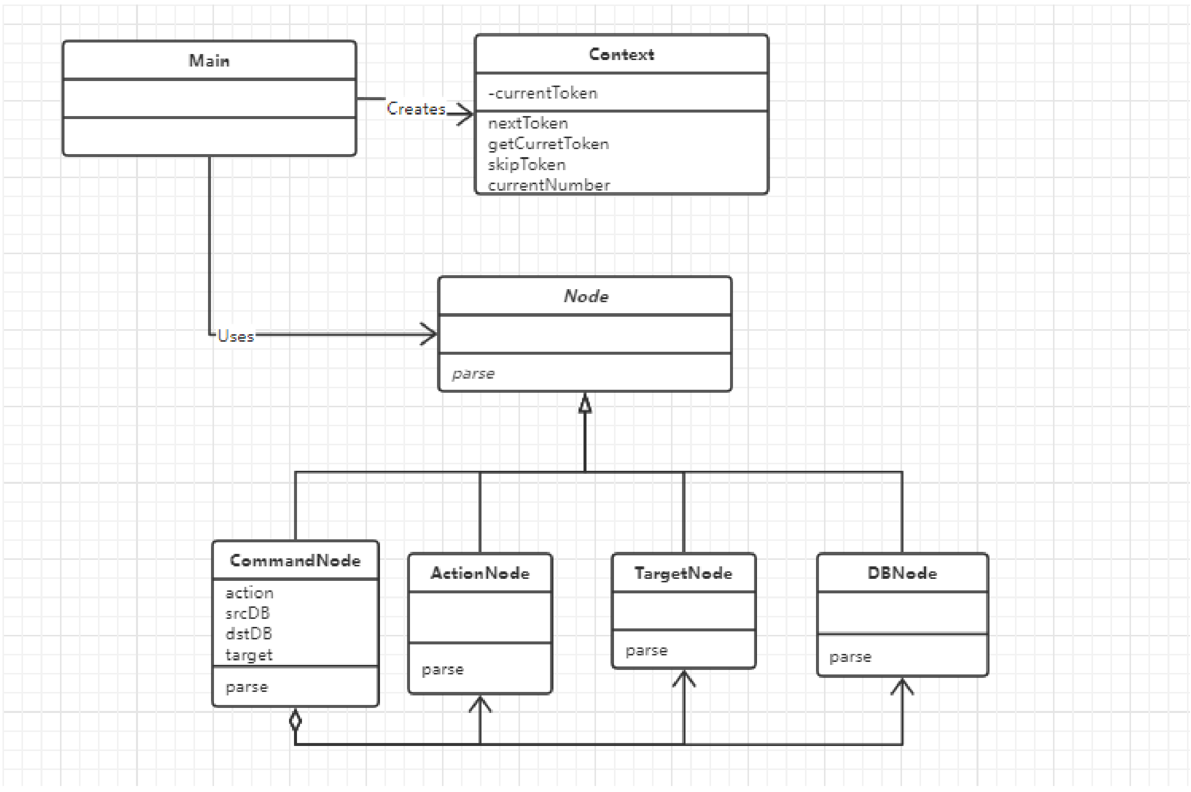
首先一条指令会有动作（action），还会有处理对象（target），和库（database）

<command> ::= <action> <target> FROM <database> TO <database>

<action>::=COPY | MOVETABLE

<target> ::=VIEW | Student

<database>::=srcDB |desDB



```java
package interpreter_moudle;

public abstract class Node {
    public abstract void parse(Context c) throws ParseException;
}
```

```java
package interpreter_moudle;

public class TargetNode extends Node{
    private String name;
    public String toString(){
        return name;
    }
    @Override
    public void parse(Context c) throws ParseException {
        name=c.currentToken();
        c.skipToken(name);
        switch (name){
            case("VIEW"):break;
            case("Student"):break;
            default:throw new ParseException();
        }
    }
}
```

```java
package interpreter_moudle;

public class ActionNode extends Node{
    private String name;
    public String toString(){
        return name;
    }
    @Override
    public void parse(Context c) throws ParseException {
        name=c.currentToken();
        c.skipToken(name);
        switch (name){
            case("COPY"):break;
            case("MOVETABLE"):break;
            default:throw new ParseException();
        }
    }


}
```

```java
package interpreter_moudle;

public class CommandNode extends Node{
    Node action=new ActionNode(),src=new DBNode(),dst=new DBNode(),target=new
TargetNode();
```

```
    @Override
    public void parse(Context c)throws ParseException {
        action.parse(c);
        target.parse(c);
        String from=c.currentToken();
        c.nextToken();
        if(!from.equals("FROM")){
            throw new ParseException();
        }
        src.parse(c);
        String to=c.currentToken();
        c.nextToken();
        if(!to.equals("TO")){
            throw new ParseException();
        }
        dst.parse(c);
    }

    public String toString(){
        return "识别出"+"从"+src+"到"+dst+"对"+target+"做这个动作"+action;
    }
}
```

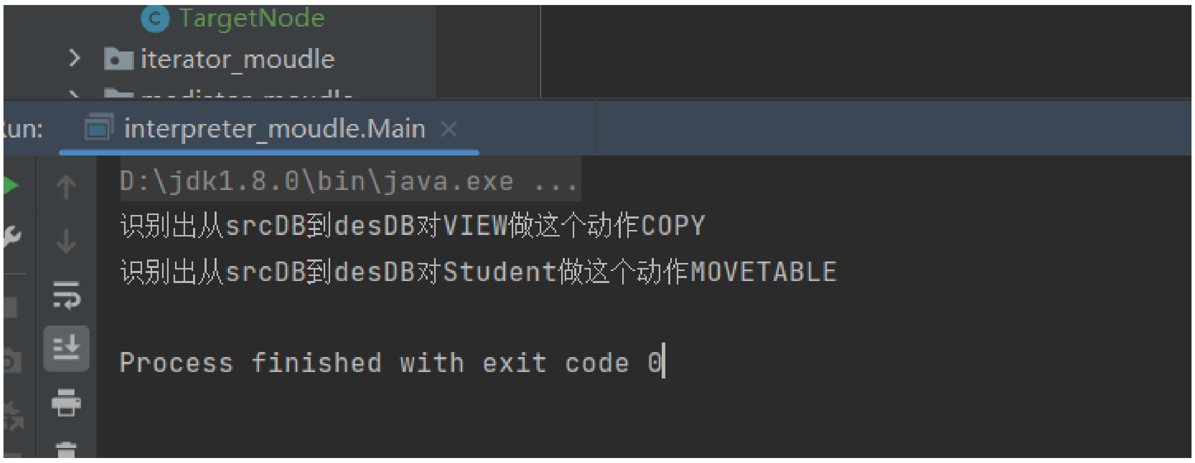
```
package interpreter_moudle;

public class DBNode extends Node{
    private String name;
    public String toString(){
        return name;
    }
    @Override
    public void parse(Context c) throws ParseException {
        name=c.currentToken();
        c.skipToken(name);

    }
}
```

```
package interpreter_moudle;

import java.util.StringTokenizer;

public class Context {
    private StringTokenizer tokenizer;
    private String currentToken;
    public Context(String text) {
        tokenizer = new StringTokenizer(text);
        nextToken();
    }
    public String nextToken() {
        if (tokenizer.hasMoreTokens()) {
            currentToken = tokenizer.nextToken();
        }
        else {
            currentToken = null;
        }
        return currentToken;
    }
    public String currentToken() {
        return currentToken;
    }
    public void skipToken(String token) {
        if (!token.equals(currentToken)) {
            System.out.println("Equal to currentToken!");
        }
        nextToken();
    }
}
```

```
package interpreter_moudle;


public class Main {
    public static void main(String[] args) throws ParseException {
        String text1 = "COPY VIEW FROM srcDB TO desDB";
        String text2 = "MOVETABLE Student FROM srcDB TO desDB";
        CommandNode c=new CommandNode();
        c.parse(new Context(text1));
        System.out.println(c);
        CommandNode c1=new CommandNode();
        c1.parse(new Context(text2));
        System.out.println(c1);
    }
}
```

```
package interpreter_moudle;

public class ParseException extends Exception{
}
```

运行main结果



```
D:\jdk1.8.0\bin\java.exe ...
识别出从srcDB到desDB对VIEW做这个动作COPY
识别出从srcDB到desDB对Student做这个动作MOVETABLE

Process finished with exit code 0
```