实验1 对象与类2

1 学习目标

- 理解方法重载的概念，并掌握其用法；
- 理解继承的概念，并掌握其用法；
- 理解方法重写的概念，并掌握其用法；
- 理解子类型多态的概念，并掌握其用法；
- 理解嵌套类的概念，并掌握其用法。

2 验证性内容

2.1 方法重载

方法重载是一种特设多态性，使用一个方法名，支持不同的参数类型。

（1）支持双精度和整型参数的加法器。

```
public class AddTest {
  public static void main(String[] args) {
    int x = 10, y = 10;
    double a = 10.0, b = 10.0;
    System.out.printf("x + y : %d\n", add(x, y)); // 选择整型加法器
    System.out.printf("a + b : %.2f\n", add(a, b)); // 选择双精度加法器
  }
  // 双精度参数列表
  public static double add(double x, double y) {
    System.out.println("双精度加法器");
    return x + y;
  }
  // 整数参数列表
  public static int add(int x, int y) {
    System.out.println("整型加法器");
    return x + y;
  }
}
```

（2）可能存在的匹配冲突。

```
public class AddTest {
  public static void main(String[] args) {
    int x = 10, y = 10;
    add(10.0, y); // 选择add(double x, int y)
    add(x, 10.0); // 选择add(int x, double y)
    add(x, y); // 错误：匹配不明确
  }
  public static double add(double x, int y) {
    return x + y;
  }
  public static double add(int x, double y) {
    return x + y;
  }
}
```

2.2 继承与子类型

场景描述：几何形状（Geometry）存在多种类型，包括圆形、矩形、三角形等。尝试定义一个几何父类，并定义具体的几何子类。

（1）定义几何形状类。

```
public class Geometry {
  protected int type = TYPE_GEOM; // 类型字段
  public static final int TYPE_GEOM = 0; // 默认类型
  public static final int TYPE_CIRCLE = 1; // 圆形
  public static final int TYPE_RECTANGLE = 2; // 矩形
  public int getType() {
    return type;
  }
}
```

```
public class Circle extends Geometry{
  {
    this.type = TYPE_CIRCLE; // 初始化类型字段为圆形
  }
  private double radius; // 半径
  public double getRadius() {
    return radius;
  }
  public void setRadius(double radius) {
    this.radius = radius;
  }
}
```

```
public class Rectangle extends Geometry{
  {
    this.type = TYPE_RECTANGLE; // 初始化类型字段为矩形
  }
  private double width, height; // 宽和高
  public double getWidth() {
    return width;
  }
  public void setWidth(double width) {
    this.width = width;
  }
  public double getHeight() {
    return height;
  }
  public void setHeight(double height) {
    this.height = height;
  }
}
```

(2) 父类型变量引用子类型对象。

```
public class Test {
  public static void main(String[] args) {
    Circle c = new Circle(); // 创建一个圆形对象
    Geometry g = c; // 将圆形对象赋值给父类型变量，这里类型会向上转换（Up-Casting）
    System.out.println(g == c); // ==用于比较两个变量是否指向同一个对象
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Geometry g = new Circle(); // 创建一个几何对象，并赋值给父类型变量
    Circle c = (Circle) g; // 将父类型变量赋值给子类型变量，这里类型需要向下强制转换
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Geometry g = new Circle();
    if(g instanceof Circle) { // 通过类型判断操作来确保类型安全
      Circle c = (Circle) g;
    }
  }
}
```

注意：虽然父类变量能够存放子类型引用，但通过父类变量访问子类型对象时，受限于父类的定义。

```
public class Test {
  public static void main(String[] args) {
    Geometry g = new Circle();
    g.getRadius(); // 错误：无法访问子类型定义的方法
}
```

（3）重写父类函数。

这里以Object类的toString()函数为例，其子类Geometry、Circle和Rectangle对该函数进行重写。

```
public class Geometry {
  @Override // 重写注解，编译器帮助检查重写合法性
  public String toString() {
    return "几何形状的类型未定义"; // 默认返回值
  }
}
public class Circle extends Geometry{
  @Override
  public String toString() {
    return "半径为" + getRadius() + "的圆形"; // 圆形信息
  }
}
public class Rectangle extends Geometry{
  @Override
  public String toString() {
    return "宽高为(" + getWidth() + "," + getHeight() + ")" + "的矩形"; // 矩形信息
  }
}
```

测试类：

```
public class Test {
  public static void main(String[] args) {
    Geometry g1 = new Geometry(); // 创建一个几何对象
    Geometry g2 = new Circle(); // 创建一个圆形对象，由父类型变量引用
    Geometry g3 = new Rectangle(); // 创建一个矩形对象，由父类型变量引用
    System.out.println(g1.toString()); // 调用Geometry的toString()重写函数
    System.out.println(g2.toString()); // 调用Circle的toString()重写函数
    System.out.println(g3.toString()); // 调用Rectangle的toString()重写函数
  }
}
```

这里已经体现出子类型多态特性，其具体的技术实现条件：

- 需要存在子类型关系；
- 父类型变量可以引用子类型对象，即Up-Casting；
- 对父类型的函数进行重写；
- 动态绑定技术，具体被调用的函数实现是由对象类型决定，而不是变量类型。

（4）子类型多态的应用。

子类型多态能实现系统行为的参数化，解耦应用层与底层实现，增强系统的扩展性。例如我们要对一组几何形状的面积进行求和操作。

先在父类定义一个面积计算的基本行为：

```
// 由于还没有学习抽象类和抽象方法，暂时用具体的方法来实现
public class Geometry {
  public double area() {
    return 0; // 假设形状没有确定则返回面积0
  }
}
```

在两个子类中重写父类方法：

```
public class Rectangle extends Geometry{
  @Override
  public double area() {
    return width * height; // 矩形面积
  }
  public Rectangle(double width, double height) {
    setWidth(width);
    setHeight(height);
  }
}
```

```
public class Circle extends Geometry{
  @Override
  public double area() {
    return radius * radius * Math.PI; // 圆形面积
  }
  public Circle(double radius) {
    setRadius(radius);
  }
}
```

在应用层编写一个几何形状面积求和的函数：

```
public class Test {
  public static void main(String[] args) {
    Geometry[] a = {new Geometry(), new Circle(2.3), new Rectangle(1.2,5.6)};
    System.out.printf("面积和为%.2f", areaSum(a));
  }
  // 充当应用功能，对一组几何形状面积求和
  public static double areaSum(Geometry[] geoList) {
    double sum = 0;
    for(Geometry g : geoList) {
      sum += g.area(); // g.area()是动态绑定，根据对象类型决定调用哪个函数
    }
    return sum;
  }
}
```

思考：当系统需要扩展，例如增加新的几何形状，这时应用层代码areaSum()需要修改吗？

2.3 嵌套类

嵌套类包含成员类、局部类、匿名类等多种形式。除了静态环境相关的嵌套类，其他都属于内部类，与具体的对象关联。

（1）静态成员类。

静态成员类不依赖于具体对象，其访问性与静态成员变量一致，可以将外部类看作静态成员类的名称空间。

例如，定义一个静态成员类：

```
public class TopClass {
  // 静态成员类
  public static class StaticNestedClass{
  }
}
```

访问静态成员类：

```
import oop.task.exp.e02.TopClass.StaticNestedClass; // 引入
public class Test {
  public static void main(String[] args) {
    StaticNestedClass c = new StaticNestedClass(); // 可以省略类名，直接访问
  }
}
```

（2）对象成员类。

对象成员类属于内部类，依赖于具体对象，在其内部隐式的存在外部对象的引用，可以访问外部对象成员。

例如，定义一个对象成员类：

```
public class TopClass {
  private int x;
  // 对象成员类，没有static修饰符
  public class InnerClass{
    public void accessOuterObject() {
      int a = x; // 直接访问外部对象的成员变量
    }
  }
}
```

内部类的实例化依赖于具体的外部类对象：

```
public class Test {
  public static void main(String[] args) {
    InnerClass inner01 = new TopClass().new InnerClass(); // 实例化依赖于具体对象
    TopClass top0 = new TopClass();
    InnerClass inner02 = top0.new InnerClass(); // 实例化依赖于具体对象
  }
}
```

（3）局部类。

局部类定义在方法内部，其作用域为方法内部，无法在方法外访问。

例如，定义一个局部类：

```
public class TopClass {
  private int x;
  public void method() {
    int localValue = 1;
    class LocalClass {
      // 局部类的定义
      int a = x; // 与对象成员类相似，也可以直接访问外部类的成员
      int b = localValue; // 也可以直接访问局部变量，但不能对局部变量进行修改
    };
    LocalClass lc = new LocalClass(); // 实例化局部类
  }
}
```

（4）匿名类。

匿名类与局部类相似，定义在方法内部，其作用域为方法内部，无法在方法外访问。匿名类没有类名，定义和实例化是一体的。

例如，定义一个匿名类：

```
public class TopClass {
  private int x;
  public void method() {
    int localValue = 1;
    // 匿名类没有类名，而是借助父类（或接口）和new关键词来定义
    Object o = new Object() {
      // 局部类的定义
      int a = x; // 与对象成员类相似，也可以直接访问外部类的成员
      int b = localValue; // 也可以直接访问局部变量，但不能对局部变量进行修改
    };
  }
}
```

3 设计性内容

3.1 图谱管理

1 项目背景

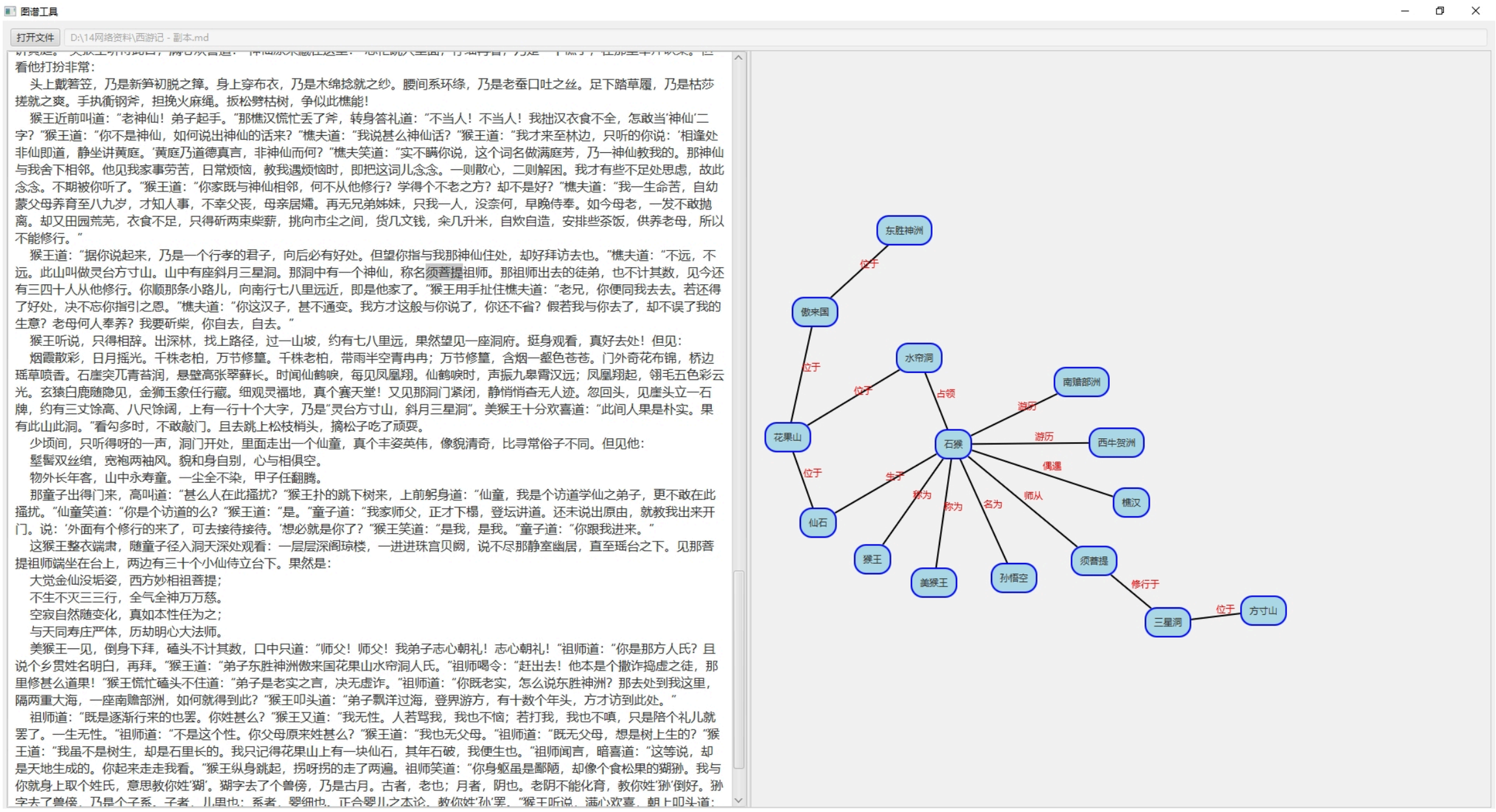
知识图谱（Knowledge Graph）是一种通过结构化方式表示现实世界中实体及其关系的技术。它使用图模型，将实体作为节点，实体之间的关系作为有向边，形成一个连接性强的网络，用于表达多样化的信息及其相互联系，被广泛应用于信息检索、推荐系统、智能问答等领域。虽然自动化构建技术在知识图谱领域取得了很大进展，但人工方式仍然是保证知识图谱高质量、精准性和可控性的核心手段。因此，自动化与人工结合的方式，将继续成为未来知识图谱构建与管理的主流模式。

2 项目需求

设计并实现一个简单的人工知识图谱构建与管理软件，可以基于文本进行实体及关系的提取与管理。本实验的主要功能需求：

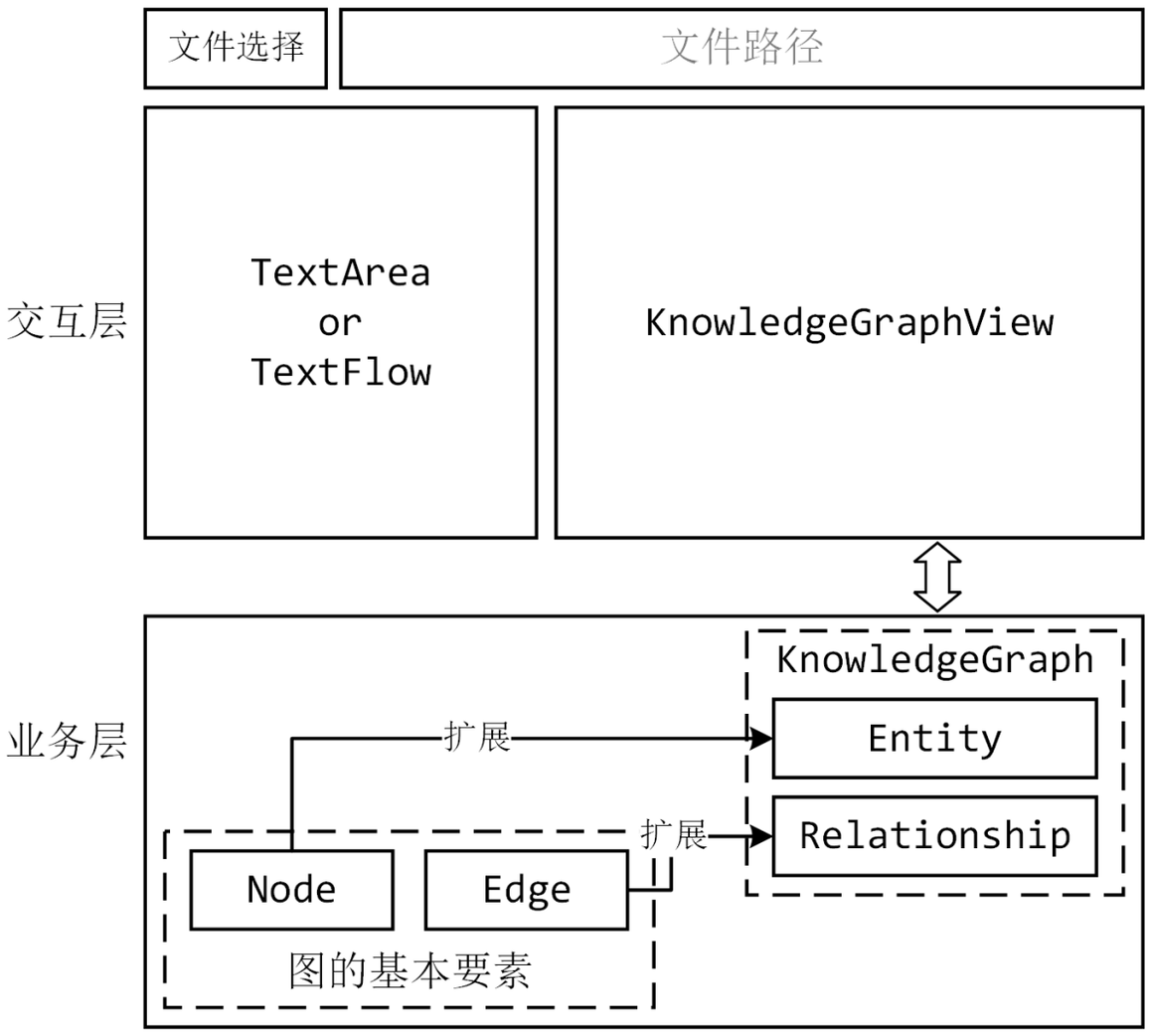
（1）能加载和显示一个文本；
（2）通过划词的方式来提取实体；
（3）能对实体和关系进行交互展示；
（4）能基于两个选中的实体创建关系；
（5）能对实体和关系的信息进行编辑。

软件预览图如下：



3 设计思路

如下图所示，程序分为交互层和业务层两部分。



业务层包含Node、Edge、Entity、Relationship和KnowledgeGraph等类，其中Node与Edge定义图的节点和边，Entity和Relationship在次基础上扩展知识信息，KnowledgeGraph负责知识图谱的管理，包括创建实体和关系以及信息编辑功能。

交互层用于接收用户输入，并将请求转发给业务层进行处理，主要包含文本区域和图谱区域，具体交互逻辑：
（1）文本区域划词提取实体，通过事件（例如键盘空格键）触发实体创建请求；
（2）图谱区域将实体创建请求发送给业务层，并将创建结果显示在图谱区域；
（3）实体在图谱区域可以左键拖动，以改变其位置；
（4）右键选择两个实体，通过事件（例如键盘空格）触发关系创建请求；
（5）图谱区域将关系创建请求发送给业务层，并将创建结果显示在图谱区域；
（6）双击实体，弹出实体信息编辑对话框，支持编辑；
（7）双击关系，弹出关系信息编辑对话框，支持编辑。

4 具体实现

（1）节点与边

节点Node存在一个编号字段，用于唯一标识一个节点。重写equals和hashCode方法的目的是在Set中能基于编号来判断是否为同一个节点。（学习集合框架后再来理解）

```
public class Node {
  private int id; // 唯一编号
  public Node(int id){
    setId(id);
  }
  // 省略Getter、Setter方法
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Node node = (Node) o;
    return id == node.id;
  }
  @Override
  public int hashCode() {
    return Integer.hashCode(id);
  }
}
```

有向边存在from和to两个Node类型字段，用于表示起始节点和终止节点。重写equals和hashCode方法的目的是在Set中能基于编号来判断是否为同一个边。

```
public class Edge {
  private Node from, to; // 起始节点和终止节点
  public Edge(Node from, Node to) {
    setFrom(from);
    setTo(to);
  }
  // 省略Getter、Setter方法
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Edge edge = (Edge) o;
    return edge.getFrom().equals(this.getFrom()) && edge.getTo().equals(this.getTo());
  }
  @Override
  public int hashCode() {
    return Integer.hashCode(Math.addExact(getFrom().getId(), getTo().getId()));
  }
}
```

（2）实体与关系

实体类Entity继承于Node，增加类型（type）、名称（name）、属性（attribute）等字段。

```
public class Entity extends Node {
  // 字段声明，String类型
  // 单参数构造器
  // 全参数构造器
  // 省略Getter、Setter方法
}
```

关系类Relationship继承于Edge，增加名称（name）、属性（attribute）等字段。

```
public class Relationship extends Edge{
  // 字段声明，String类型
  // 单参数构造器
  // 全参数构造器
  // 省略Getter、Setter方法
}
```

（3）知识图谱

图谱由实体和关系组成，为了实现实体与关系的快速查询，增加了两个索引。

```
public class KnowledgeGraph {
  private Set<Entity> entities; // 实体集合
  private Set<Relationship> relations; // 关系集合
  private Map<Entity, Set<Relationship>> outMap; // 出发索引
  private Map<Entity, Set<Relationship>> inMap;  // 进入索引
  private int seqNumber; // 实体自动编号
  public KnowledgeGraph() {
    seqNumber = 1; // 初始化编号为1
    entities = new HashSet<Entity>();
    relations = new HashSet<Relationship>();
    outMap = new HashMap<>();
    inMap = new HashMap<>();
  }
}
```

创建和添加实体过程都由图谱来管理：

```
public class KnowledgeGraph {
  // 创建和添加实体
  public Optional<Entity> addEntity(String type, String name, String attribute) {
    Entity e = new Entity(seqNumber++, type, name, attribute); // 创建实体
    if(addEntity(e)) { // 若添加成功则返回实体对象
      return Optional.ofNullable(e); // 添加成功
    }else {
      return Optional.empty(); // 添加失败，则返回空
    }
  }
  // 重载形式，若添加成功则返回true
  public boolean addEntity(Entity e) {
    return this.entities.add(e);
  }
}
```

创建和添加实体关系过程也由图谱来管理：

```
public class KnowledgeGraph {
  // 创建与添加关系，若添加成功则返回关系引用
  public Optional<Relationship> addRelation(Entity from, Entity to) {
    Optional<Relationship> r = Optional.empty(); // 空的关系引用
    if(entities.contains(from) && entities.contains(to)) { // 两个实体都存在
      Relationship rel = new Relationship(from, to); // 创建关系对象
      if(relations.add(rel)) { // 添加成功
        outMap.computeIfAbsent(from, f -> new HashSet<>()).add(rel); // 更新出索引
        inMap.computeIfAbsent(to, f -> new HashSet<>()).add(rel); // 更新入索引
        r = Optional.ofNullable(rel);
      }
    }
    return r;
  }
}
```

（4）创建并显示实体

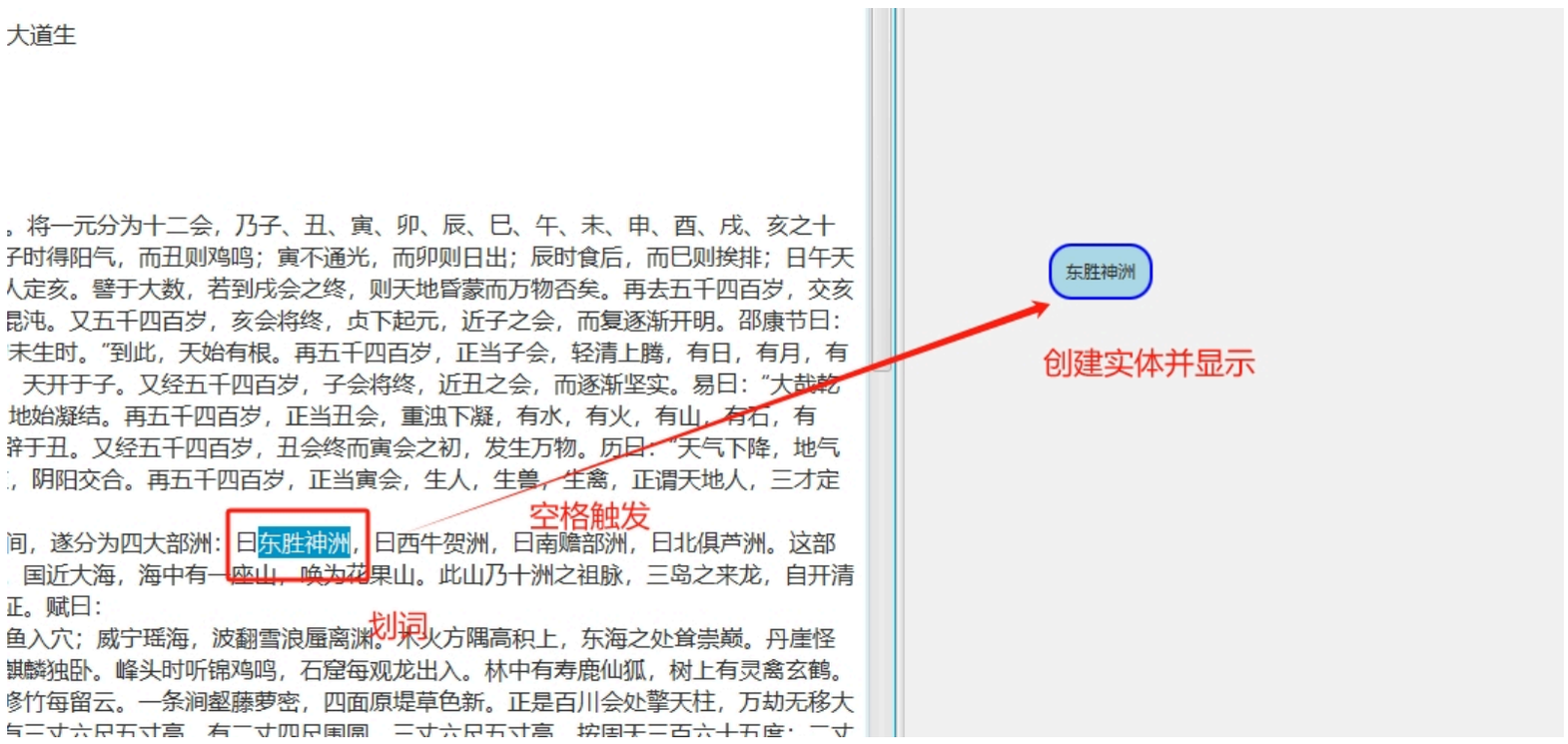
交互逻辑：在主界面的文本区域，增加一个事件处理器，当用户按下空格键时，将当前文本区域选中的文本发送给图谱视图来创建实体。

```
public class KGApp extends Application{
  @Override
  public void start(Stage primaryStage) throws Exception {
    // 空格键触发实体创建事件
    textArea.setOnKeyPressed(e -> {
      switch(e.getCode()) {
        case SPACE : {
          view.add(textArea.getSelectedText()); // 向图谱视图发送创建实体请求
        }break;
        default : break;
      }
      e.consume();
    });
  }
}
```

图谱视图负责创建实体，并将实体添加到图谱中：

```
public class KnowledgeGraphView extends Pane{
  private KnowledgeGraph graph; // 每个图谱视图对应一个图谱对象
  private static EntityLabel selectedStart, selectedEnd; // 当前选中的实体标签
  // 根据文本创建一个实体
  public void add(String text) {
    graph.addEntity("", text, "").ifPresent(e -> {
      getChildren().add(new EntityLabel(e)); // 创建实体标签用于显示
    });
  }
  // 展示实体的标签类，作为图谱视图的对象成员类来定义
  public class EntityLabel extends Label{
    private Entity entity; // 实体
    private double offsetX, offsetY; // 移动参考坐标
  }
}
```

基本效果：



（5）创建并显示关系

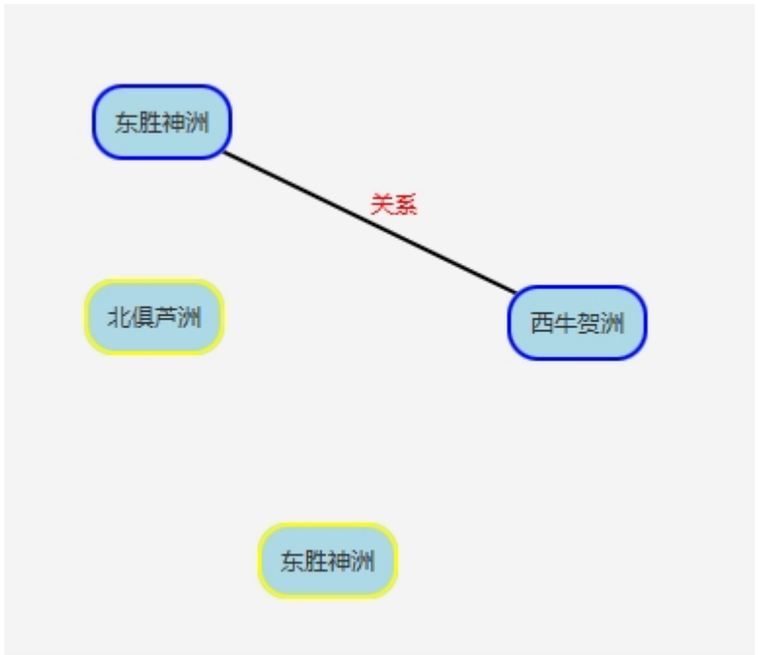
创建关系时，需要选择两个实体，然后创建关系，并将关系添加到图谱中。

实体选择过程中，实际是选择实体标签，由于左键用于拖动实体标签，这里用右键来选择实体标签作为起始点或终止点。若当前没有选中任何实体标签，则选中的实体标签为起始点；若当前有一个实体标签被选中，则当前选择的实体标签为终止点；若当前有两个实体标签被选中，则取消第一个选中的实体标签，将第二实体标签作为起始点，当前选择的标签作为终止点。当选中两个实体标签后，通过空格事件触发创建关系请求。

```
public class KnowledgeGraphView extends Pane{
  public KnowledgeGraphView() {
    graph = new KnowledgeGraph(); // 暂不考虑加载已有图谱的情况
    actionInitialize();
  }
  public void actionInitialize() {
    this.setOnMousePressed(event -> {
      if (event.getButton() == MouseButton.PRIMARY) {
        clear(); // 清理选择
      }
      event.consume();
    });
    this.setOnMouseClicked(event -> this.requestFocus()); // 鼠标点击后获取焦点
    this.setOnKeyPressed(e -> {
      System.out.println(e.getCode());
      switch(e.getCode()) {
        case SPACE:{
          // 创建关系并显示
          createRelationship().ifPresent(line -> {
            this.getChildren().add(0, line);
            clear();
          });
        }break;
        default: break;
      }
      e.consume();
    });
  }
  // 选择一个实体标签
  public void select(EntityLabel e) {
    if(selectedStart == null) {
      selectedStart = e;
      selectedStart.selectedStyle(); // 选择样式
    }else if(selectedStart != null && selectedEnd == null){
      selectedEnd = e;
      selectedEnd.selectedStyle(); // 选择样式
    }else {
      selectedStart.nonSelectedStyle(); // 未选择样式
      selectedStart = selectedEnd;
      selectedEnd = e;
      selectedEnd.selectedStyle(); // 选择样式
    }
  }
  // 清除选择
  public void clear() {
    if(selectedStart != null) {
      selectedStart.nonSelectedStyle();
      selectedStart = null;
    }
    if(selectedEnd != null) {
      selectedEnd.nonSelectedStyle();
      selectedEnd = null;
    }
  }
  // 创建关系并显示关系线
  public Optional<RelationLine> createRelationship() {
    if(selectedStart != null && selectedEnd != null) {
      Entity from = selectedStart.getEntity();
      Entity to = selectedEnd.getEntity();
      Optional<RelationLine> line = Optional.empty();
      Optional<Relationship> relation = graph.addRelation(from, to);
      if(relation.isPresent()) {
        line = Optional.ofNullable(new RelationLine(selectedStart, selectedEnd, relation.get()));
      }
      return line;
    }else {
      return Optional.empty();
    }
```

```
  }
  public class EntityLabel extends Label{
    public EntityLabel(Entity entity){
      // 处理双击事件，改变标签的背景颜色
      this.setOnMouseClicked(event -> {
        if (event.getClickCount() == 2) {
          // 省略
        }else if(event.getButton() == MouseButton.SECONDARY){
          select(this); // 右键触发向图谱视图发送标签选择请求；内部类直接访问外部类成员函数。
        }
      });
    }
  }
  // 展示关系的线段类
  public class RelationLine extends Group{
    private Relationship relation;
    private Line line = new Line();
    private Text label; // 线上的标签
  }
}
```

基本效果：



（6）修改实体信息

创建实体后，可以通过双击实体标签弹出对话框，对实体信息进行修改，将修改的状态同时更新标签。

用于编辑和修改实体信息的对话框：

```
public class EntityDialog extends Dialog<Entity>{
  private Entity entity;

  public EntityDialog(Entity e) {
    entity = e;
    this.setTitle("编辑实体信息");
    // 添加确认和取消按钮
    ButtonType okButtonType = new ButtonType("确认", ButtonType.OK.getButtonData());
    getDialogPane().getButtonTypes().addAll(okButtonType, ButtonType.CANCEL);
    // 使用格网布局，展示实体的属性
    // 创建文本输入框
    // 将输入框放入格网中
    // 设置确认按钮的行为
    setResultConverter(dialogButton -> {
      if (dialogButton == okButtonType) {
        // 更新实体信息
        entity.setType(typeField.getText());
        entity.setName(nameField.getText());
        entity.setAttribute(attrField.getText());
        return entity;
      }
      return null;
    });
  }
}
```

双击实体标签显示编辑框：

```
public class KnowledgeGraphView extends Pane{
  // 展示实体的标签类
  public class EntityLabel extends Label{
    public EntityLabel(Entity entity){
      // 处理双击事件，改变标签的背景颜色
      this.setOnMouseClicked(event -> {
        if (event.getClickCount() == 2) {
          // 显示对话框，并等待结果
          new EntityDialog(entity).showAndWait().ifPresent(update -> {
            this.updateText(); // 更新标签文本
          });
        }else if(event.getButton() == MouseButton.SECONDARY){
          select(this);
        }
      });
    }
    public void updateText() {
      this.setText(entity.getName());
    }
  }
}
```

基本效果：

（7）修改关系信息

与实体类似，通过对关系线的双击触发，弹出对话框，对关系信息进行修改，将修改的状态同时更新关系线。

```
public class RelationshipDialog extends Dialog<Relationship>{
  private Relationship relation;
  public RelationshipDialog(Relationship rel) {
    relation = rel;
    this.setTitle("编辑关系信息");
    // 添加确认和取消按钮
    // 使用格网布局，展示实体的属性
    // 创建文本输入框
    // 将输入框放入格网中
    getDialogPane().setContent(grid);

    // 设置确认按钮的行为
    setResultConverter(dialogButton -> {
      if (dialogButton == okButtonType) {
        // 更新实体信息
        relation.setName(nameField.getText());
        relation.setAttribute(attrField.getText());
        return relation;
      }
      return null;
    });
  }
}
public class KnowledgeGraphView extends Pane{
  // 展示关系的线段类
  public class RelationLine extends Group{
    public RelationLine(EntityLabel start, EntityLabel end, Relationship rel) {
      this.setOnMouseClicked(event -> {
        if (event.getClickCount() == 2) {
          new RelationshipDialog(relation).showAndWait().ifPresent(update -> {
            this.updateText(); // 更新线标签
          });
        }
      });
    }
    // 更新线标签
    public void updateText() {
      label.setText(relation.getName());
    }
  }
}
```

基本效果：

4 实验要求

完成第2部分验证性实验，理解源代码背后的逻辑；完成第3部分设计性实验，理解程序的设计与实现逻辑，完成类图和部分源代码。实验成果材料具体包含以下两个材料：

（1）实验报告按报告模板提示和要求填写，包含两部分内容：

- 第2部分验证性内容自查；
- 第3部分核心类的`plantuml`类图代码。

（2）源代码主要完成两个类：

- 实体类：Entity.java

- 关系类：Relationship.java

实验成果上传目录：`FTP服务器地址/2024-2025-1/面向对象高级编程/学号/实验2`

5 其他说明

暂无