



中山大學
SUN YAT-SEN UNIVERSITY

FJE-GO：基于 GO 实现的 JSON 文件可视化工具

姓名 李 博

学号 21307278

学院 计 算 机 学 院

专业 计算机科学与技术

目录

1	项目代码	2
2	设计文档	3
2.1	类图与说明	3
2.1.1	Funny JSON Explorer	3
2.1.2	Builder	4
2.1.3	Factory	4
2.1.4	TreeFactory 和 RectangleFactory	4
2.1.5	Component	5
2.1.6	Leaf	5
2.1.7	Container	5
2.1.8	IconFamily	6
2.1.9	Config	6
2.2	设计模式	6
2.2.1	抽象工厂模式	6
2.2.2	工厂方法模式	7
2.2.3	建造者模式	8
2.2.4	组合模式	9
2.2.5	策略模式	9
3	效果截图	10

1 项目代码

【GitHub 仓库链接】

本项目使用的编程语言是 GO 语言，项目的源代码以及相关的测试文件、配置文件都已经在 GitHub 中给出，下面是整个项目的大致目录树以及文件的相关说明：

```
Design Pattern 习题（实验要求）
├── tex-report （实验报告）
├── src（源代码）
│   ├── version-1 ： 第一版本代码
│   ├── verison-2 ： 基于设计模式以及实验要求优化后的第二版本代码
│   │   ├── component.go ： 定义 Component 接口、Leaf 和 Container 类
│   │   ├── factory.go ： 定义 Factory 接口及其具体实现
│   │   ├── builder.go ： 定义 Builder 类
│   │   ├── explorer.go ： 定义 FunnyJSONExplorer 类
│   │   ├── main.go ： 主程序入口
│   │   ├── loadConfig.go ： 定义处理 Icon 配置文件的相关类
│   │   ├── treeDraw.go ： 实现对于 tree 风格的 Draw 方法
│   │   └── rectangleDraw.go ： 实现对于 rectangle 风格的 Draw 方法
│   └── json ： 用于测试的 JSON 文件
```

└─ config : 配置文件, 可以用于 IconFamily 的配置

2 设计文档

2.1 类图与说明

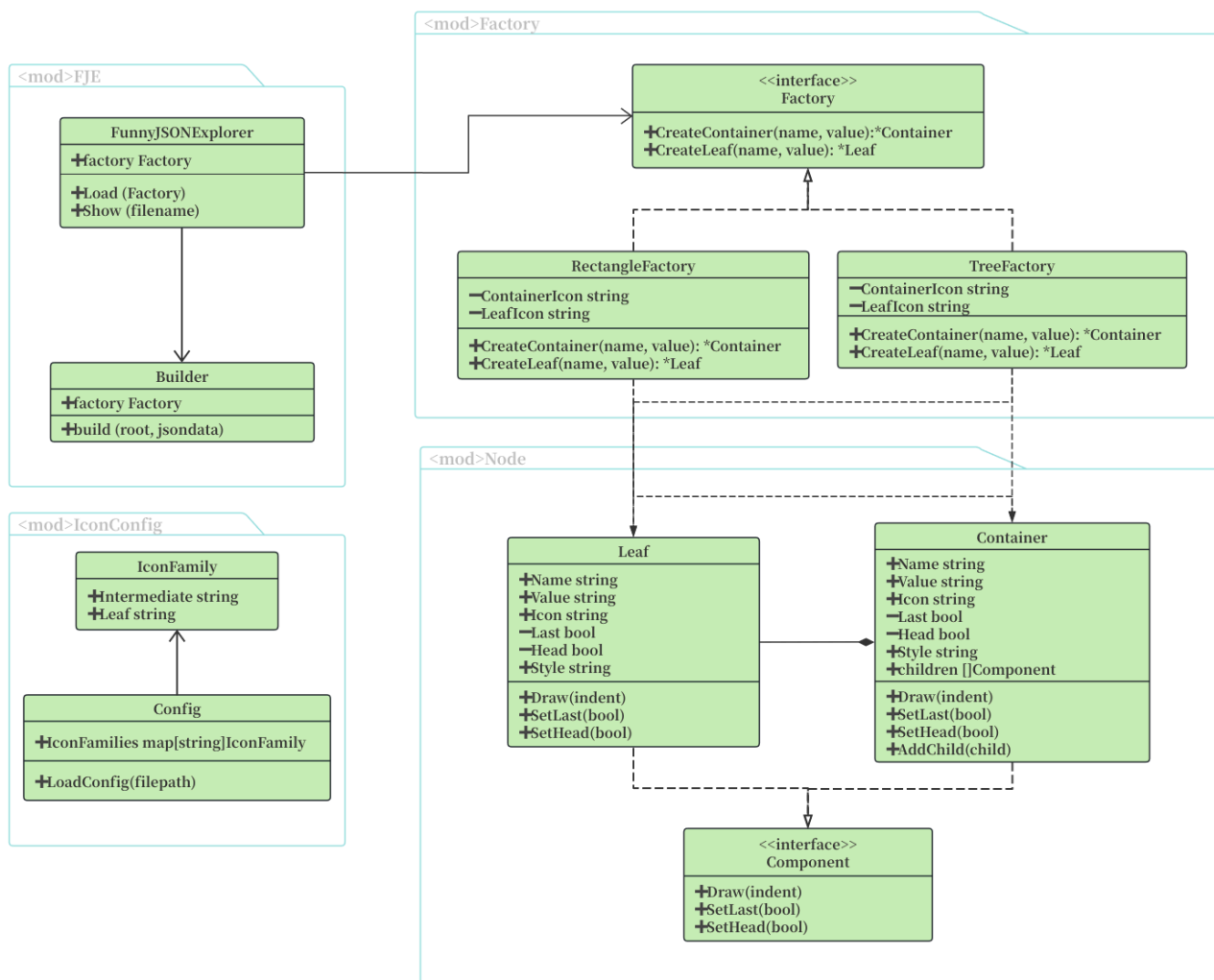


图 1: FJE-GO 项目的 UML 类图

FJE-GO 项目的 UML 类图如上图 1 所示, 其中类的命名参考了实验要求中的领域模型, 下面对其进行具体说明。

2.1.1 Funny JSON Explorer

- 描述:

- 这是程序的主类, 主要实现读取 JSON 文件并对其进行可视化的主过程。
- 其定义了一个 `factory` 的抽象工厂对象, 主要目的是通过抽象工厂接口来接受 `main` 函数中根据具体风格指定的具体工厂对象 (例如 `TreeFactory` 以及 `RectangleFactory`)。

- 在对 JSON 文件的读取和创建过程中，为了将复杂的自定义 Component 结构的创建解耦，根据建造者设计模式可以创建一个 Builder 类，在 builder 类中会根据不同的工厂类型创建不同风格的树结构，从而将创建过程分离出去。

- 主要方法:

- **Load(factory Factory):** 用来设置使用的抽象工厂对象。
- **Show(filename string):** 读取 JSON 文件并进行可视化。具体来说，实现对指定路径下的 JSON 文件进行读取并调用 Builder 的 build 方法将 JSON 结构转换为自定义的 Component 结构，最后调用 root 节点的 Draw 方法进行可视化。

2.1.2 Builder

- 描述:

- 主要实现根据 JSON 数据构建树结构，并返回创建的根节点，从而实现创建过程在主过程中的解耦。

- 属性:

- factory: 创建节点所使用的工厂，可以根据具体工厂类型进行创建。

- 方法:

- **Build(root Component, data map[string]interface):** 基于 JSONdata 以及传入的引用根节点进行树结构的创建。采用的方法是 DFS 思想，通过深度优先遍历整个 JSON 结构创建所有中间节点 Container 以及叶节点 Leaf，最后返回根节点。

2.1.3 Factory

- 描述:

- 这是抽象工厂接口，用于创建中间节点 Container 和叶子节点 Leaf。这是一种抽象工厂设计模式的思想，可以通过抽象接口来支持新的 Style 以及新的具体工厂类型，实现良好的可扩展性。

- 方法:

- **CreateContainer(name string, value string) *Container:** 通过节点键值创建一个中间节点。
- **CreateLeaf(name string, value string) *Leaf:** 通过节点键值创建一个叶子节点。

2.1.4 TreeFactory 和 RectangleFactory

- 描述:

- Factory 接口的具体实现，用于创建特定 Style 的节点。

- 属性:

- ContainerIcon: 中间节点图标，用于在可视化指定具体的中间节点使用的图标。

- LeafIcon: 叶子节点图标, 用于在可视化指定具体的叶子节点使用的图标。

- 方法:

- **CreateContainer(name string, value string) *Container**: 通过节点键值创建一个具体 Style 的中间节点。
- **CreateLeaf(name string, value string) *Leaf**: 通过节点键值创建一个具体 Style 的叶子节点。

2.1.5 Component

- 描述:

- 抽象节点接口类型, 用于确保节点的继承可以实现 Draw 方法。

- 方法:

- **Draw(indent string)**: 递归实现节点可视化, 其中 indent 为递归传递的可视化前缀。
- **SetLast(b bool)**: 设置是否为最后一个子节点。
- **SetHead(b bool)**: 设置是否为头节点。

2.1.6 Leaf

- 职责:

- 是对抽象节点类型的具体实现, 表示一个树结构中的叶子节点。

- 属性:

- icon: 叶子节点使用的图标类型。
- name: 叶子节点的名称。
- Head: 是否为头节点, 用于可视化判断。
- Last: 是否为尾节点, 用于可视化判断。

- 方法:

- **Draw(indent string)**: 根据前缀 indent 可视化叶子节点。
- **SetHead(b bool)**: 设置是否为头节点。
- **SetLast(b bool)**: 设置是否为尾节点。

2.1.7 Container

- 职责:

- 是对抽象节点类型的具体实现, 表示一个树结构中的中间节点, 其拥有子节点列表。

- 属性:

- icon: 中间节点使用的图标类型。
- name: 中间节点的名称。

- Head: 是否为头节点，用于可视化判断。
- Last: 是否为尾节点，用于可视化判断。
- children: 中间节点的维护子节点列表。

- 方法:

- **Draw(indent string)**: 根据前缀 indent 可视化中间节点。
- **AddChild(child Component)**: 为中间节点添加子节点。
- **SetHead(b bool)**: 设置是否为头节点。
- **SetLast(b bool)**: 设置是否为最后一个子节点。

2.1.8 IconFamily

- 描述:

- 用于表示所使用的图表族，主要包括对中间节点或者叶子节点两种不同的 Icon。

- 属性:

- Intermediate: 中间节点图标。
- Leaf: 叶节点图标。

2.1.9 Config

- 描述:

- 主要用于存储在 config.json 文件中所指定的所有图标族，导入给 main 函数用于用户选取。

- 属性:

- IconFamilies: 所有配置图表族组，用 map[string]IconFamily 存储。

- 方法:

- **LoadConfig(filePath string) (*Config, error)**: 从 config.json 文件导入所有的图标族，加载至本地的 IconFamilies 中。

2.2 设计模式

2.2.1 抽象工厂模式

抽象工厂模式是一种创建型设计模式，它提供一个接口，用于创建一系列相关或相互依赖的对象，而无需指定它们具体的类。核心思想是将对象的创建过程与对象的使用分离，确保在不修改客户端代码的情况下，可以方便地更换具体的工厂，从而创建不同的对象族。

在本 FJE-GO 项目中，使用了抽象工厂接口，它定义了创建抽象产品的方法，包括中间节点 (Container) 和叶子节点 (Leaf):

```

1 type Factory interface {
2     CreateContainer(name, value string) *Container
3     CreateLeaf(name, value string) *Leaf
4 }

```

并提供了两个具体工厂的实现 `TreeFactory` 和 `RectangleFactory`，分别用于创建树形风格和矩形风格的节点。下面给出了 `TreeFactory` 的具体代码实现：

```

1 type TreeFactory struct {
2     ContainerIcon string
3     LeafIcon      string
4 }
5
6 func (f *TreeFactory) CreateContainer(name string, value string) *Container {
7     return &Container{Name: name, Value: value, Icon: f.ContainerIcon, Style: "tree", Last: false}
8 }
9
10 func (f *TreeFactory) CreateLeaf(name string, value string) *Leaf {
11     return &Leaf{Name: name, Value: value, Icon: f.LeafIcon, Style: "tree", Last: false}
12 }

```

使用抽象工厂接口的优势在于，可以根据依赖抽象工厂接口来使用具体工厂创建具体风格的节点，而不需要直接依赖具体的节点类，这样有利于实现和具体产品的解耦，极大地提高了项目的可扩展性，符合软件工程设计模式的开闭原则。

在客户端可以通过不同的风格类型创建不同的工厂来实现可拓展性：

```

1 // style-dealing
2 var factory Factory
3 switch *style {
4 case "tree":
5     factory = &TreeFactory{icon.Intermediate, icon.Leaf}
6 case "rectangle":
7     factory = &RectangleFactory{icon.Intermediate, icon.Leaf}
8 //TODO: 在这里可以添加新的样式的抽象工厂
9 default:
10     fmt.Println("Unknown style")
11     return
12 }

```

2.2.2 工厂方法模式

工厂方法模式（Factory Method Pattern）是一种创建型设计模式，提供了一种创建对象的接口，但由子类决定要实例化的类是哪一个。以此试图将对象的实例化推迟到子类。它主要用于解决对象创建的问题，通过将对象创建的细节封装起来，使得客户端代码无需知道具体的创建过程。

在本 FJE-GO 项目中，在定义的抽象工厂接口（如 2.2.1 所述）之中定义了创建节点（包括中间节点 `Container` 和叶子节点 `Leaf`）的抽象接口，在抽象工厂不直接指定创建方法实例化哪种风格的节点，而是依赖于具体工厂的实现，体现了工厂方法模式的运用。

```

1 type Factory interface {
2     CreateContainer(name, value string) *Container
3     CreateLeaf(name, value string) *Leaf
4 }

1 func (f *TreeFactory) CreateContainer(name string, value string) *Container {
2     return &Container{Name: name, Value: value, Icon: f.ContainerIcon, Style: "tree"
3         ↪ ", Last: false}
4 }

5 func (f *TreeFactory) CreateLeaf(name string, value string) *Leaf {
6     return &Leaf{Name: name, Value: value, Icon: f.LeafIcon, Style: "tree", Last:
7         ↪ false}
8 }

```

2.2.3 建造者模式

建造者模式 (Builder Pattern) 主要用于将一个复杂对象的构建与表示分离, 使得同样的构建过程可以创建不同的表示。

在本 FJE-GO 项目中使用了建造者设计模式的思想, 通过 Builder 类来负责根据 JSON 数据构建树结构, 并返回根节点, 从而实现在 JSE 主过程中将构建过程解耦。

```

1 type Builder struct {
2     factory Factory
3 }

4

5 // Build方法 根据 JSON 数据构建树结构, 返回根节点
6 func (b *Builder) Build(root Component, data map[string]interface{}) {
7     for key, value := range data {
8         var child Component
9         if _, ok := value.(map[string]interface{}); ok {
10             child = b.factory.CreateContainer(key, val)
11             root.(*Container).AddChild(child)
12         } else {
13             // Leaf and NoneTy
14             child := b.factory.CreateLeaf(key, val)
15             root.(*Container).AddChild(child)
16         }
17
18         if v, ok := value.(map[string]interface{}); ok {
19             b.Build(child, v)
20         }
21     }
22 }

```

优势在于通过建造者模式可以将构建过程解耦分离并封装, 从而可以根据不同的 JSON 数据构建不同的树结构, 而不需要修改构建过程的代码; 另外, 它提供了更加良好的扩展性, 由于建造者模式将构建过程进行了封装, 可以更容易地扩展构建过程。

2.2.4 组合模式

组合模式（Composite Pattern）是一种结构型设计模式，它允许将对象组合成树形结构以表示“部分-整体”的层次结构。使用组合模式可以很好地利用递归和多态的机制来使用树形结构。

在本 FJE-GO 项目中，Container 类表示树形结构中的中间节点，Leaf 类表示叶子节点，它们共同实现了 Component 接口。其中，Container 作为中间节点拥有 children 数据结构，其有 Component 组合而成。代码如下：

```

1  type Component interface {
2      Draw(string)
3      SetLast(bool) /
4      SetHead(bool)
5  }
6
7  type Container struct {
8      Name      string
9      Value     string
10     Icon      string
11     Last      bool
12     Head      bool
13     Style     string
14     Children  []Component
15 }

```

对于 JSON 结构（树形结构），使用组合模式允许在节点中包含其他节点，从而可以递归地构建复杂的树形结构。通过将 Container 类中的 Children 属性设置为 Component 接口的实现类，实现了树形结构的递归组合，从而用简单的节点类型实现负责树形整体。

2.2.5 策略模式

策略模式（Strategy Pattern）是一种行为设计模式，定义了一系列算法，并将每个算法封装起来，使它们可以互相替换。策略模式使得算法可以独立于使用它的客户而变化。

在本 FJE-GO 项目中，用户可以指令使用的风格类型并通过命令行输入，程序可以根据指定类型来实施不同的策略类型，并创建相应风格的具体工厂。使用策略模式可以根据用户的指定而进行策略的切换，从而灵活地创建不同风格的组件。

```

1  var factory Factory
2  switch *style {
3  case "tree":
4      factory = &TreeFactory{icon.Intermediate, icon.Leaf}
5  case "rectangle":
6      factory = &RectangleFactory{icon.Intermediate, icon.Leaf}
7  default:
8      fmt.Println("Unknown style")
9      return
10 }

```

3 效果截图

使用如下简单的 JSON 代码可以进行测试：

```

1 {
2   "oranges": {
3     "mandarin": {
4       "clementine": null,
5       "tangerine": "cheap & juicy!"
6     }
7   },
8   "apples": {
9     "gala": null,
10    "pink lady": null
11  }
12 }
```

使用两种图标族 (weather 以及 poker) 以及两种风格 (tree 以及 rectangle) 效果如下：

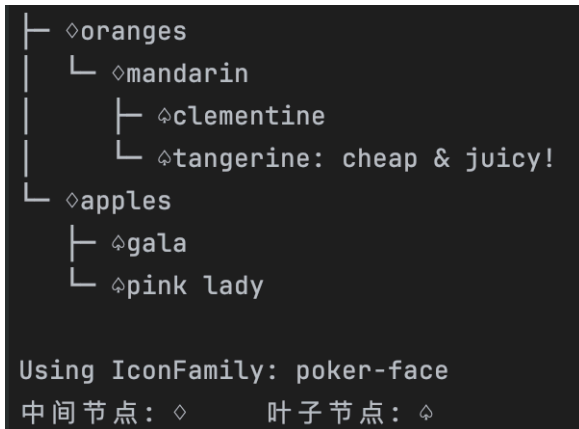


图 2: Tree-Style Poker-Icon JSON 可视化

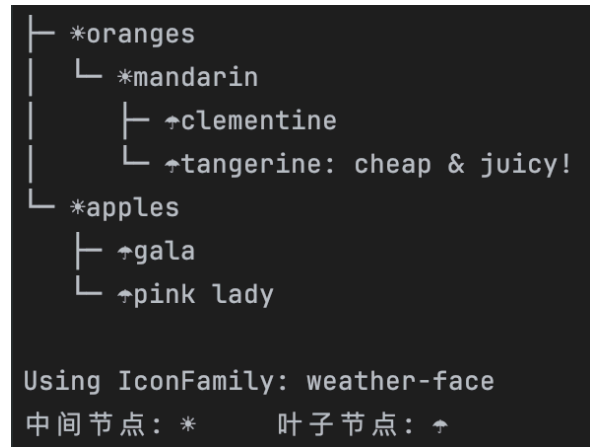


图 3: Tree-Style Weather-Icon JSON 可视化



图 4: Rectangle-Style Poker-Icon JSON 可视化



图 5: Rectangle-Style Weather-Icon JSON 可视化

为了验证可视化效果的正确性，可以使用层数更加深的文件进行可视化，如图 6 以及图 7 所示，可以看出效果是符合预期的。

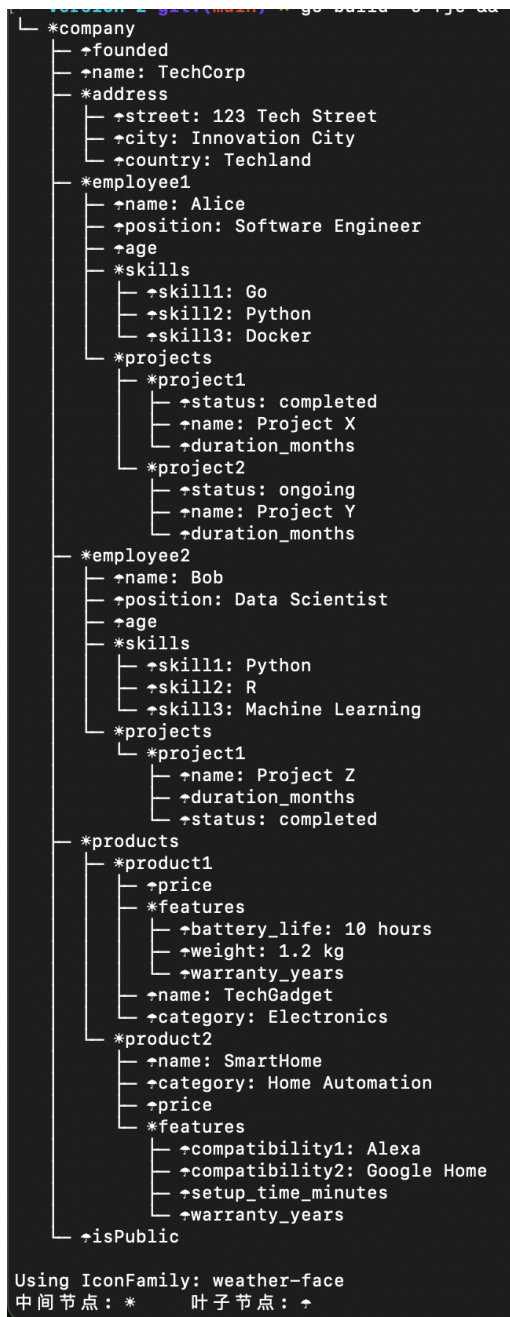


图 6: 加深的 Tree-Style Weather-Icon
JSON 可视化效果

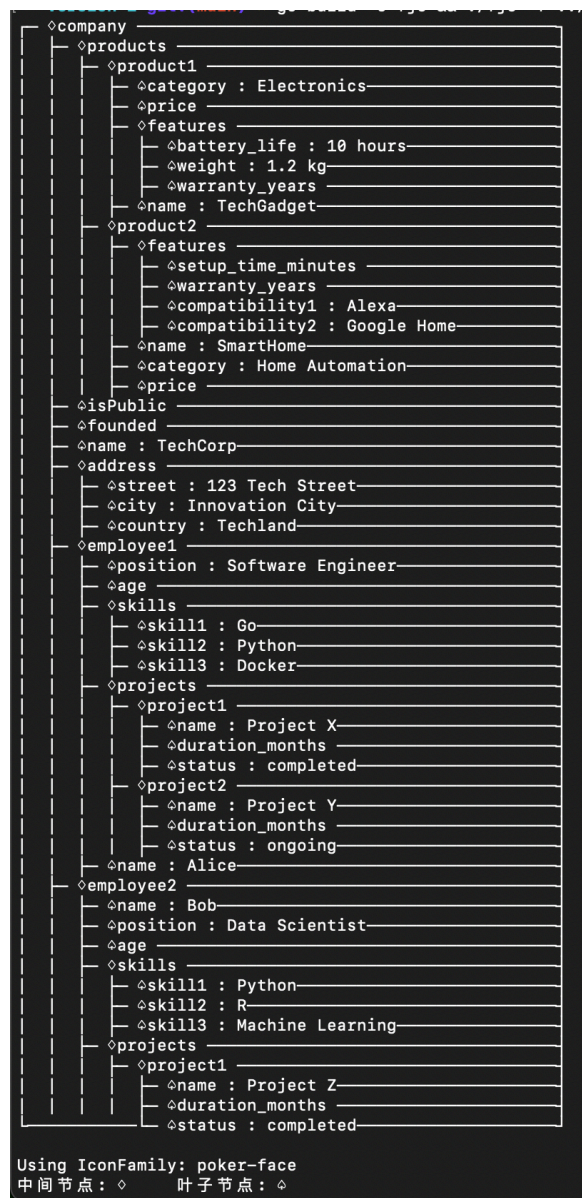


图 7: 加深的 Rectangle-Style Poker-Icon
JSON 可视化效果