

# AI 驱动 软件研发 全面进入数字化时代

中国·深圳 11.24-25

AI+  
software  
Development  
Digital  
summit



## 打造LLM高质量代码生成的神兵利器

陈雅菲 中兴通讯

# 科技生态圈峰会 + 深度研习



—1000+ 技术团队的选择



K+全球软件研发行业创新峰会

会议时间: 2024.05.24-25



K+全球软件研发行业创新峰会

会议时间: 2024.09.20-21



AI+ 软件研发数字峰会

会议时间: 2023.11.24-25



AI+ 软件研发数字峰会

会议时间: 2024.07.19-20



AI+ 软件研发数字峰会

会议时间: 2024.11.15-16

# ▶ 演讲嘉宾



## 陈雅菲

中兴通讯 公司资深软件架构师 资深敏捷教练

---

深耕软件领域20余年，拥有丰富的大型软件项目架构、设计与开发经验，在软件设计开发理论与实践上均有独到见解。

长期致力于敏捷技术实践指导与推广，多次获得公司十佳敏捷教练称号。目前研究方向是LLM辅助设计编码。

# 目录

## CONTENTS

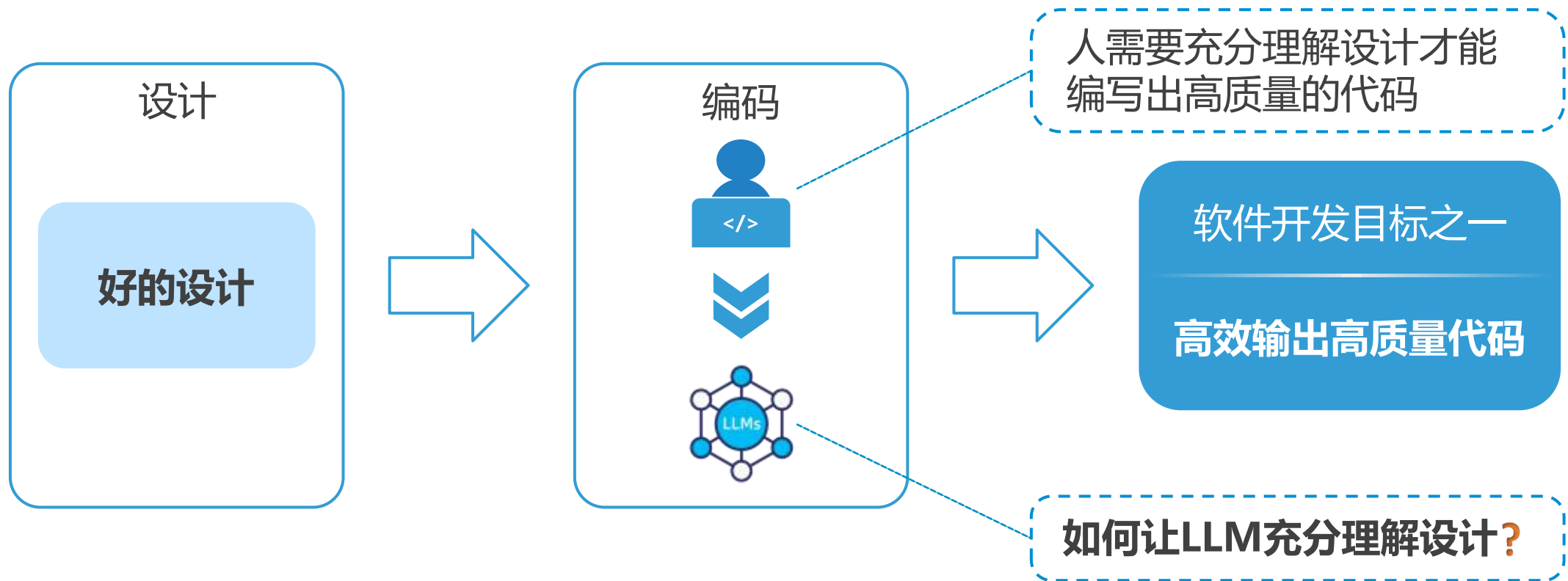
1. LLM代码生成面临的问题
2. 问题的解决思路与方案
3. 实践情况与效果评估
4. 总结&展望



## PART 01

# LLM代码生成面临的问题

# LLM代码生成面临的问题

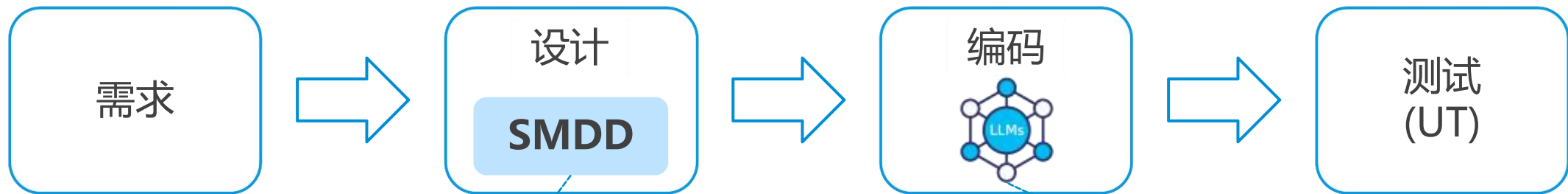


如何获得好的设计，并让LLM充分理解设计，从而高效生成高质量代码

## PART 02

# 问题的解决思路与方案

# 解决思路



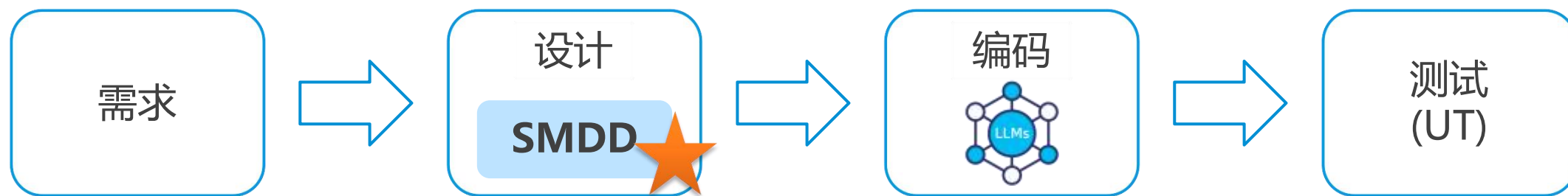
- **语义模型驱动设计**(Semantic Model Driven Design)
- 一种应对高软件复杂度的设计方法
- 抽象出针对特定问题领域的语义模型
- 采用结构化表达描述语义模型

- 不同LLM有不同使用约束
- 根据不同使用约束提炼**提示模式**

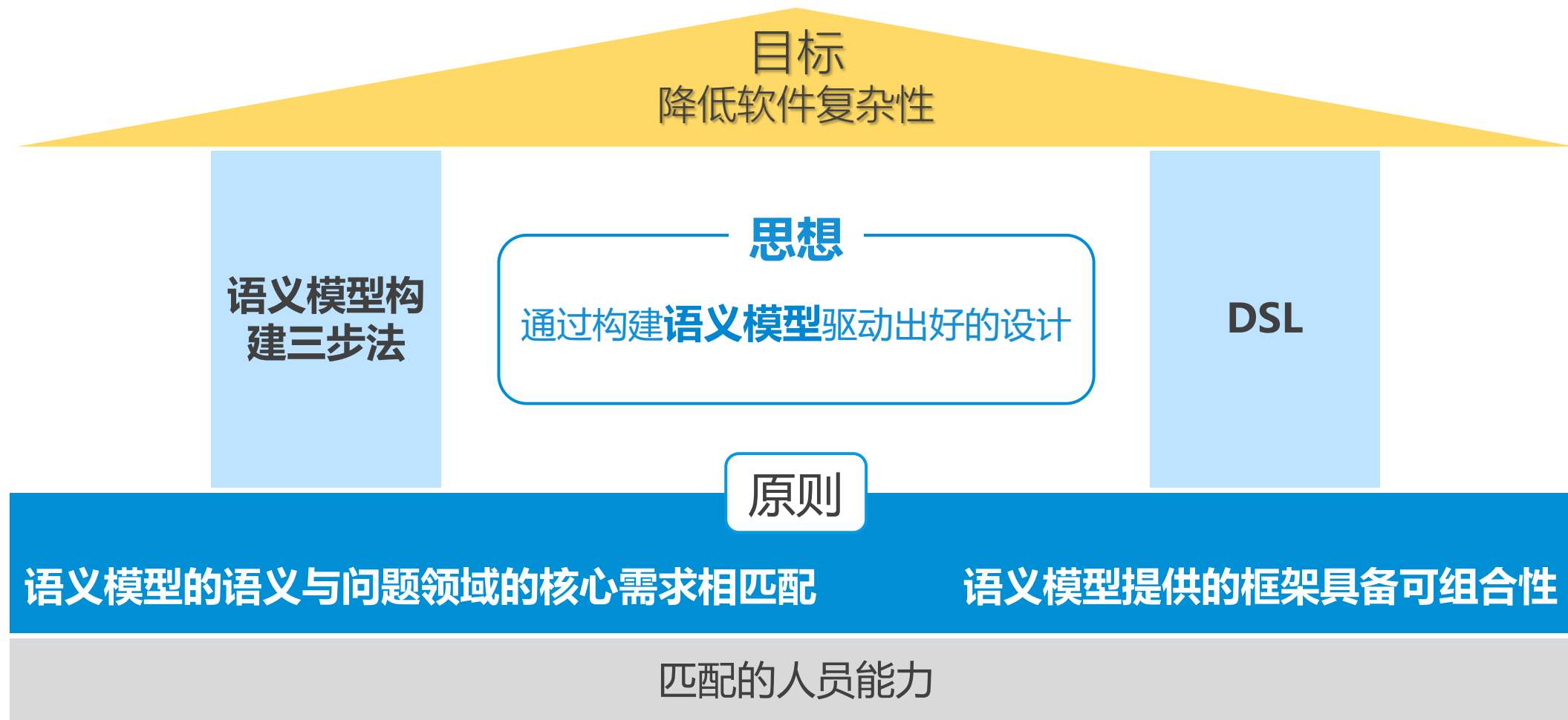
“SMDD+提示模式” 是LLM高质量代码生成的神兵利器



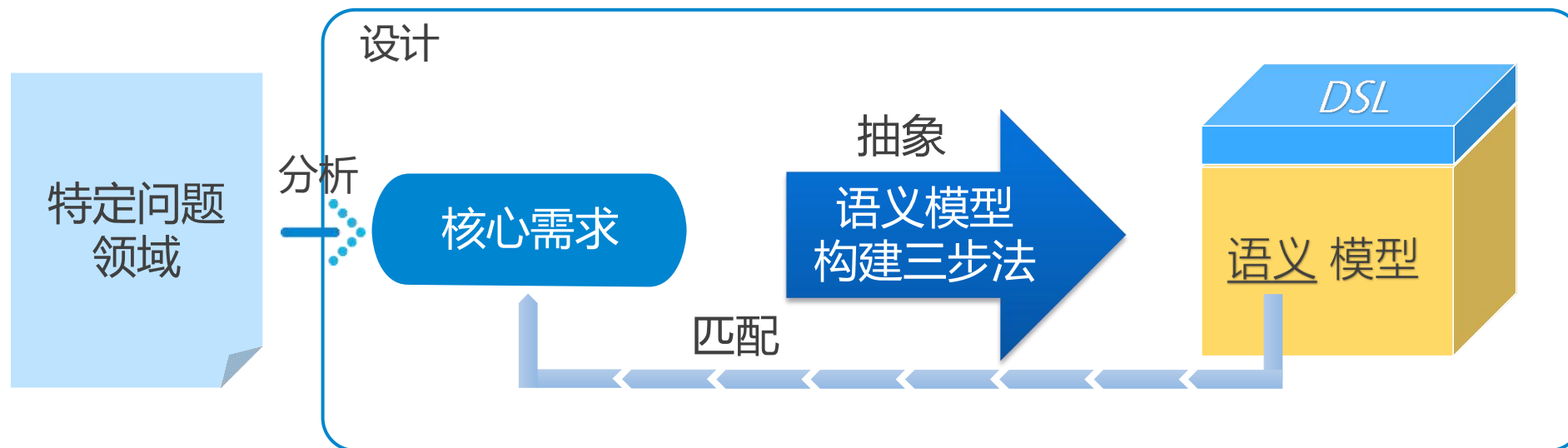
## ▶ 解决方案：SMDD



# ▶ 语义模型驱动设计(SMDD)总览



# 语义模型驱动设计(SMDD)框架



A

深入分析特定问题领域、挖掘问题领域**核心需求**

B

使用**语义模型构建三步法**，从核心需求抽象出**语义模型**，其语义要与特定问题领域的核心需求相匹配

C

基于语义模型提供一套**DSL**(Domain-Specific Language, 领域特定语言)

# ▶ 语义模型

语义模型是提供了与问题领域核心需求相匹配的语义、框架具备可组合性的计算模型



## 原子

最简单、最基本的实体



## 组合方法

把原子组合起来构成更复杂实体的方法



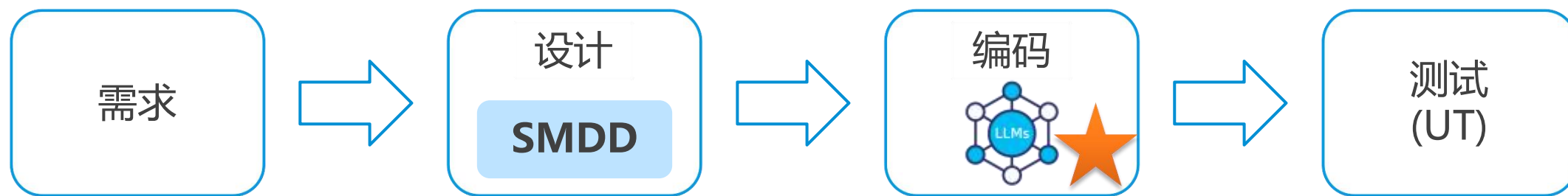
## 抽象手段

命名复杂实体的手段，命名后的复杂实体可以像原子一样通过组合方法组合成更为复杂的实体



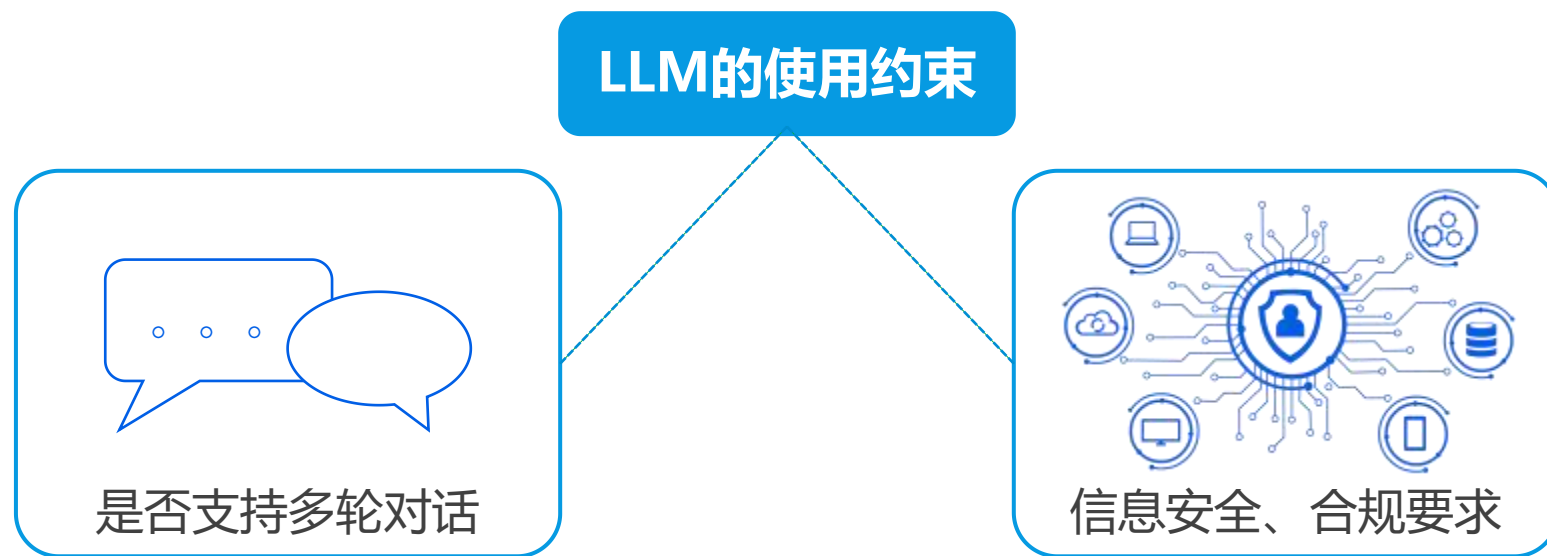
可组合性

## ▶ 解决方案：提示模式





# 提示模式：不同使用约束、不同提示模式



提示模式 = 交互方式 + 提示模板

## 提示模式一

**交互方式：**多轮对话 + 无代码上传

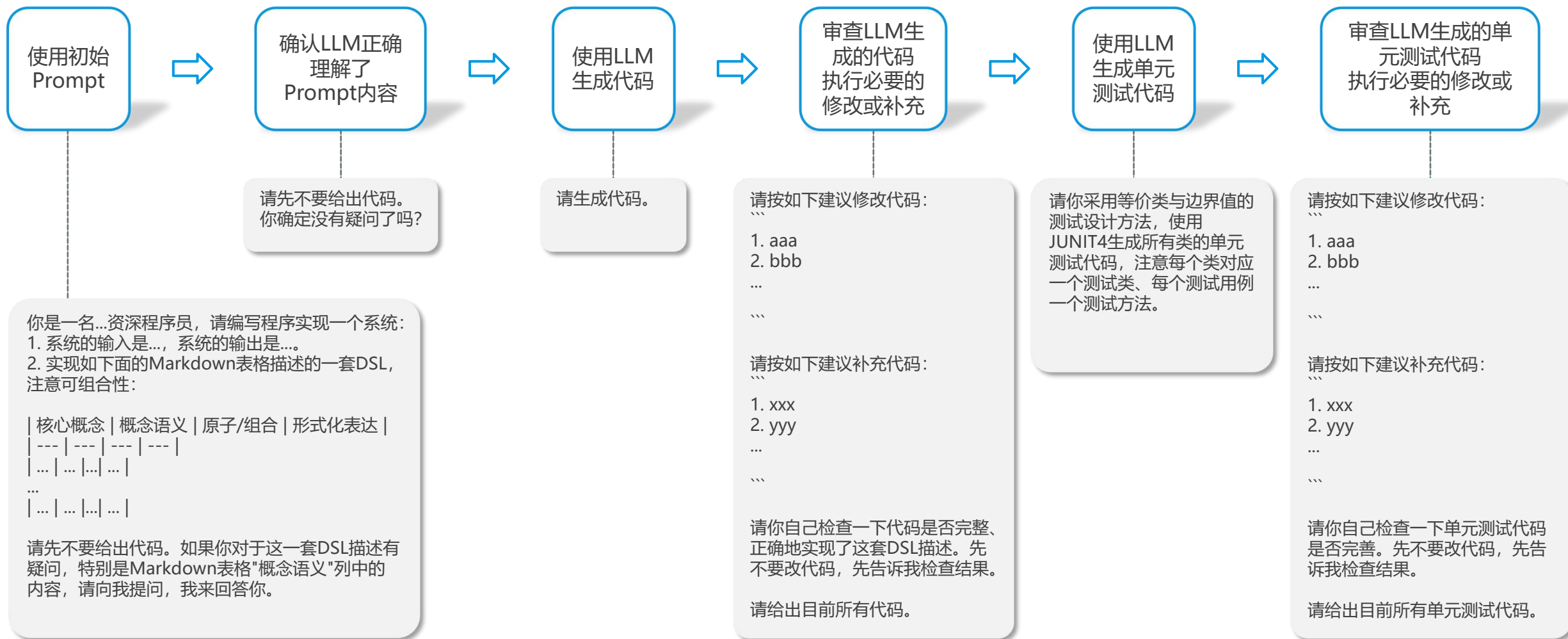
**提示模板：**每轮对话对应一个提示模板

## 提示模式二

**交互方式：**单轮对话 + 有代码上传

**提示模板：**每个场景对应一个提示模板

# 提示模式一：多轮对话+无代码上传



## ▶ 提示模式二：场景

### 提示模式二

**交互方式：**单轮对话 + 有代码上传

**提示模板：**每个场景对应一个提示模板

#### 代码生成

全量代码生成

增量代码生成

代码缺陷修复

#### 单元测试代码生成

无依赖代码的单元测试代码生成

有依赖代码的单元测试代码生成

# 提示模式二：代码生成

## 全量代码生成

### ## 任务概述

你是一名...资深程序员，请编写程序实现一个系统：

1. 系统的输入是...，系统的输出是...
2. 实现如下面的Markdown表格描述的一套DSL，注意可组合性：

核心概念	概念语义	原子/组合	形式化表达
---	---	---	---
...	...	...	...
...	...	...	...

### ## 任务重申

1. 请编写程序实现上述系统。

...

初始全量代码生成

### 说明：

1. 使用**Markdown语法**、按**总分总结构**书写提示。
2. 在**任务重申**部分强调一下需要LLM做的事情和注意事项。

## 增量代码生成

### ## 任务概述

你是一名...资深程序员，请编写程序实现一个系统：

1. 系统的输入是...，系统的输出是...
2. 实现如下面的Markdown表格描述的一套DSL，注意可组合性：

核心概念	概念语义	原子/组合	形式化表达
---	---	---	---
...	...	...	...
...	...	...	...

3. 已实现的条件或规则未列入表格。相关的实现代码在``已有代码``中。

4. 请在``//TODO``处新增代码。

### ## 已有代码

```
```java
```

```
...
```

```
...
```

基于已有代码，增量生成新的代码

### ## 待实现的代码

```
```java
```

```
...
```

```
//TODO: ...
```

```
...
```

```
...
```

### ## 任务重申

1. 请在``//TODO``处按要求新增代码。

...

## 代码缺陷修复

### ## 任务概述

你是一名...资深程序员，请编写程序实现一个系统：

1. 系统的输入是...，系统的输出是...
2. 实现如下面的Markdown表格描述的一套DSL，注意可组合性：

核心概念	概念语义	原子/组合	形式化表达
---	---	---	---
...	...	...	...
...	...	...	...

3. 已实现的条件或规则未列入表格。相关的实现代码在``已有代码``中。

4. 请在``//FIXME``处修复代码缺陷。

### ## 已有代码

```
```java
```

```
...
```

```
...
```

基于已有代码，修复已有代码缺陷

### ## 待修复的代码

```
```java
```

```
...
```

```
//FIXME: ...
```

```
...
```

```
...
```

### ## 任务重申

1. 请在``//FIXME``处按要求修复代码缺陷。

...

# 提示模式二：单元测试代码生成

## 无依赖代码的单元测试代码生成

### ## 任务概述

你是一名Java资深程序员。请你采用等价类与边界值的测试设计方法，使用JUNIT4生成所有``被测代码``的单元测试代码，注意每个类对应一个测试类、每个测试用例一个测试方法。

### ## 被测代码

```
``java
```

...

### ## 任务重申

1. 请使用JUNIT4生成所有``被测代码``的单元测试代码。

\* 注意采用等价类与边界值的测试设计方法。

\* 每个类对应一个测试类、每个测试用例一个测试方法。

...

1. 初始全量单元测试代码生成
2. 针对对其他代码无依赖的被测代码生成单元测试代码

## 有依赖代码的单元测试代码生成

### ## 任务概述

你是一名Java资深程序员。请你采用等价类与边界值的测试设计方法，**基于``已有代码``**使用JUNIT4生成``被测代码``的单元测试代码，注意每个类对应一个测试类、每个测试用例一个测试方法。

### ## 已有代码

```
``java
```

...

### ## 被测代码

```
``java
```

...

### ## 任务重申

1. 请基于``已有代码``使用JUNIT4生成所有``被测代码``的单元测试代码。

\* 注意采用等价类与边界值的测试设计方法。

\* 每个类对应一个测试类、每个测试用例一个测试方法。

...

针对对其他代码有依赖的被测代码生成单元测试代码



# ▶ LLM高质量代码生成的神兵利器

## SMDD(语义模型驱动设计)

做出好的设计，输出结构化表达的**语义模型**，使得LLM能够充分理解设计

LLM高质量  
代码生成的  
神兵利器

## 提示模式

交互方式+提示模板，使得LLM基于**语义模型**高效生成高质量代码

# **PART 03**

## **实践情况与效果评估**

## ▶ 实践案例：字符串转换重构

```
if(fieldName.equals("A")) {  
    if(beforeValue.equals("1")) {  
        afterValue = "a1";  
    } else {  
        afterValue = "a2";  
    }  
}
```

```
if(fieldName.equals("B")) {  
    if(beforeValue.equals("1")) {  
        afterValue = "b1";  
    } else if(val.equals("2")) {  
        afterValue = "b2";  
    } else {  
        afterValue = "N/A";  
    }  
}
```

重复

```
if(fieldName.equals("C")) {  
    try {  
        double tmp = Double.parseDouble(beforeValue);  
        DecimalFormat df = new DecimalFormat("##0.00");  
        afterValue = df.format(tmp);  
    } catch(Exception ex) {  
        ex.printStackTrace();  
    }  
}
```

```
if(fieldName.equals("D")) {  
    String tmp = beforeValue.toString();  
    int index = tmp.indexOf(".");  
    afterValue = tmp.subSequence(0, index + 2);  
}
```

相同逻辑的不同实现

# ▶ 字符串转换重构：语义模型驱动设计示例

核心概念	概念语义	原子/组合	形式化表达
规则	规则的输入是"源字符串", 输出是"Optional<String>", 其中String是转换后的字符串	/	rule: srcStr -> Optional<String>
"一一映射"规则	该规则由映射列表构造而成, 映射列表的每个元素由一对mappingFrom和mappingTo构成。该规则接受的源字符串即mappingFrom, 找到对应的mappingTo并输出	原子规则	oneToOne([(mappingFrom, mappingTo)])
"保留指定位数小数"规则	该规则由"保留小数位数"构造而成。该规则对接受的源字符串进行保留指定位数小数处理后并输出	原子规则	decimalDigits(digitNum)
"条件应用分支"规则	该规则由"条件"、"then规则"、"else规则"构造而成。当条件满足时应用then规则, 否则应用else规则	组合规则	whenThenElse(condition, thenRule, elseRule)
"顺序应用"规则	该规则由"规则列表"构造而成。按照给定的顺序依次应用多条规则。当某条规则应用成功, 则返回该条规则的应用结果; 如果该条规则应用失败, 则继续应用下一条规则	组合规则	seq([rule])
条件	条件的输入是"源字符串"、输出boolean, 条件满足时输出true、条件不满足时输出false	/	condition: srcStr -> { true   false }
"null"条件	该条件无构造参数。当该条件接受的源字符串为null时为true, 否则为false	原子条件	nullString()
"空白字符串"条件	该条件无构造参数。当该条件接受的源字符串仅含空白字符或为空时为true, 否则为false	原子条件	blankString()
"或"条件	布尔代数的或	组合条件	or(condition1, condition2)
"与"条件	布尔代数的与	组合条件	and(condition1, condition2)
.....	.....	.....	.....

# ▶ 字符串转换重构：提示模式一

提示模式 = 交互方式 + 提示模板

## 提示模式一



**交互方式：**多轮对话 + 无代码上传

**提示模板：**每轮对话对应一个提示模板

## 提示模式二

**交互方式：**单轮对话 + 有代码上传

**提示模板：**每个场景对应一个提示模板



# 字符串转换重构：代码生成

Q: 你是一名Java资深程序员，请编写程序实现一个系统：

1. 系统的输入是"源字符串"。系统的输出是"Optional<String>"，其中String是转换后的字符串，如果转换成功，输出非empty的Optional<String>，如果转换失败，输出Optional.empty。

2. 实现如下面的Markdown表格描述的一套DSL，注意可组合性：

| 核心概念 | 概念语义 | 原子/组合 | 形式化表达 |

| --- | --- | --- | --- |

| 规则 | 规则的输入是"源字符串"，输出是"Optional<String>"，其中String是转换后的字符串 |  
/ | rule: srcStr -> Optional<String> |

| "——映射"规则 | 该规则由映射列表构造而成，映射列表的每个元素由一对mappingFrom和mappingTo构成。该规则接受的源字符串即mappingFrom，找到对应的mappingTo并输出 | 原子规则 | oneToOne([(mappingFrom, mappingTo)]) |

...

| 条件 | 条件的输入是"源字符串"、输出boolean，条件满足时输出true、条件不满足时输出false

| 组合条件 | condition: srcStr -> { true | false } |

| "null"条件 | 该条件无构造参数。当该条件接受的源字符串为null时为true，否则为false | 原子条件 | nullString() |

...

请先不要给出代码。如果你对于这一套DSL描述有疑问，特别是Markdown表格"概念语义"列中的内容，请向我提问，我来回答你。

A: 好的，我理解了你对这一套DSL的描述。如果你对其中的任何部分有疑问，我会尽力回答。请告诉我如果你需要任何进一步的解释或有其他问题。

Q: 请先不要给出代码。你确定没有疑问了吗？

A: 我已经仔细阅读了您提供的DSL描述，并没有特别的疑问。我明白这是一个规则转换系统，其中包含了原子规则和组合规则，以及一些条件用于判断是否应用规则。您可以继续说明您的需求，我将等待您的指示。如果您需要对DSL的某些方面进行澄清或有其他问题，请随时告诉我，我会尽力回答。

Q: 请生成代码。

A: 好的，下面是用Java实现这套DSL的代码：

```
public class RuleEngine {  
  
    // 不变规则  
    30 usage  
    public static Rule unchanged() {  
        return srcStr -> Optional.of(srcStr);  
    }  
  
    // 后缀规则  
    43 usage 4 usages  
    public static Rule suffix() {  
        return srcStr -> srcStr + "suffix";  
    }  
  
    // 原子条件: 永真  
    1 usage  
    public static Condition alwaysTrue() {  
        return srcStr -> true;  
    }  
  
    // 百分比规则  
    1 usage  
    public static Condition percentage(int percentage) {  
        return srcStr -> srcStr.length() * percentage / 100 > 0;  
    }  
  
    // 保留指定  
    1 usage  
    public static Condition reserved(String reserved) {  
        return srcStr -> srcStr.contains(reserved);  
    }  
  
    // 组合条件: 或  
    2 usages  
    public static Condition or(Condition condition1, Condition condition2) {  
        return srcStr -> condition1.apply(srcStr) || condition2.apply(srcStr);  
    }  
  
    // 组合条件: 与  
    1 usage  
    public static Condition and(Condition condition1, Condition condition2) {  
        return srcStr -> condition1.apply(srcStr) && condition2.apply(srcStr);  
    }  
  
    // 组合条件: null或空白字符串  
    1 usage  
    public static Condition nullOrBlankString() {  
        return or(nullString(), blankString());  
    }  
}
```

# 字符串转换重构：单元测试代码生成

Q: 请你采用等价类与边界值的测试设计方法，使用JUNIT4生成单元测试代码，注意每个类对应一个测试类、每个测试用例一个测试方法。

A: 好的，我会使用等价类与边界值的测试设计方法，并使用JUnit 4生成单元测试代码。请注意，由于描述的规则和条件比较复杂，测试覆盖所有情况可能会比较繁琐。因此，我将选择一些典型的等价类和边界值来生成测试用例，以便展示测试的基本思路。你可以在此基础上进一步扩展测试用例，以确保代码的正确性。  
首先，我们将每个规则和条件对应的测试代码放在独立的测试类中，并使用JUnit 4进行测试。以下是测试类和测试方法的示例代码：

...  
上述代码中，每个规则和条件对应一个测试类，测试类中包含了若干测试方法，每个方法对应一个测试用例。我选择了一些简单的测试用例来演示测试方法的设计，你可以根据实际需求添加更多测试用例。在实际测试中，需要覆盖各种等价类和边界值来确保代码的正确性。

请注意，上述测试代码仅用于演示目的，实际测试中应该尽可能全面地覆盖各种情况，以保证系统的稳健性和正确性。

```
import org.junit.Assert;
import org.junit.Test;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Optional;

public class RuleEngineTest {

    // 测试——映射规则
    @Test
    public void testOneToOne() {
        List<String[]> mappings = Arrays.asList(
            new String[]{"A", "Apple"},
            new String[]{"B", "Banana"}
        );
        RuleEngine.Rule rule = RuleEngine.or(
            RuleEngine.in(Arrays.asList("A", "B")),
            RuleEngine.contains("X")
        );
        Assert.assertEquals(Optional.of("Apple"), rule.apply("A"));
        Assert.assertEquals(Optional.of("Banana"), rule.apply("B"));
        Assert.assertEquals(Optional.empty(), rule.apply("C"));
    }
}
```

```
// 测试“与”条件
@Test
public void testAndCondition() {
    RuleEngine.Condition condition1 = RuleEngine.in(Arrays.asList("A", "B", "C"));
    RuleEngine.Condition condition2 = RuleEngine.contains("X");
    RuleEngine.Condition condition = RuleEngine.and(condition1, condition2);

    Assert.assertFalse(condition.apply("A"));
    Assert.assertFalse(condition.apply("B"));
    Assert.assertFalse(condition.apply("C"));
    Assert.assertTrue(condition.apply("XYZ"));
    Assert.assertFalse(condition.apply("D"));
}

// 测试“null或空白字符串”条件
@Test
public void testNullOrBlankStringCondition() {
    RuleEngine.Condition condition = RuleEngine.nullOrBlankString();
    Assert.assertTrue(condition.apply(null));
    Assert.assertTrue(condition.apply(""));
    Assert.assertTrue(condition.apply(" "));
    Assert.assertFalse(condition.apply("Not Blank"));
}
}
```

```
// 测试后缀规则
@Test
public void testSuffix() {
    RuleEngine.Rule rule = RuleEngine.suffix("_Suffix");
    Assert.assertEquals(Optional.of("Source_String_Suffix"), rule.apply("Source_String"));
}
}
```

```
rule.apply("58"));
s( digitNum: 2);
rule.apply("3.14159");
}
```

# ▶ 字符串转换重构：提示模式二

提示模式 = 交互方式 + 提示模板

## 提示模式一

**交互方式：**多轮对话 + 无代码上传

**提示模板：**每轮对话对应一个提示模板

## 提示模式二

**交互方式：**单轮对话 + 有代码上传

**提示模板：**每个场景对应一个提示模板



# 字符串转换重构：全量代码生成

## ## 任务概述

你是一名Java资深程序员，请编写程序实现一个系统：

1. 系统的输入是"源字符串"。系统的输出是"Optional<String>"，其中String是转换后的字符串，如果转换成功，输出非empty的Optional<String>，如果转换失败，输出Optional.empty。

2. 实现如下面的Markdown表格描述的一套DSL，注意可组合性：

| 核心概念 | 概念语义 | 原子/组合 | 形式化表达 |

| --- | --- | --- | --- |

| 规则 | 规则的输入是"源字符串"，输出是"Optional<String>"，

其中String是转换后的字符串 | / / rule: srcStr ->

Optional<String> |

| "——映射"规则 | 该规则由映射列表构造而成，映射列表的每个元素由一对mappingFrom和mappingTo构成。该规则接受的源字符串即mappingFrom，找到对应的mappingTo并输出 | 原子

规则 | oneToOne([(mappingFrom, mappingTo)]) |

...

| 条件 | 条件的输入是"源字符串"、输出boolean，条件满足时输出true、条件不满足时输出false | 组合条件 | condition: srcStr ->

{ true | false } |

| "null"条件 | 该条件无构造参数。当该条件接受的源字符串为null时为true，否则为false | 原子条件 | nullString() |

...

...

## ## 任务重申

1. 请编写程序实现上述系统。

```
public class StringTransformationSystem {
```

```
// 规则接口
```

```
interface Rule extends Function<String, Optional<String>> {}
```

```
// 条件接口
```

```
interface Condition extends Predicate<String> {}
```

```
// 构建规则：——映射规则
```

```
static Rule oneToOne(List<Pair<String, String>> mapping) {
```

```
    return srcStr -> {
```

```
        for (Pair<String, String> mapping : mapping) {
```

```
            if (srcStr.equals(mappingFrom)) {
```

```
                return Optional.of(mappingTo);
```

```
            }
```

```
        }
```

```
    };
```

```
};
```

```
// 构建规则：提供者规则
```

```
static Rule supplier(String srcStr, Supplier<String> supplier) {
```

```
    return srcStr -> Optional.of(supplier.get());
```

```
};
```

```
// 构建规则：不变规则
```

```
static Rule unchanged() {
```

```
    return Optional::of;
```

```
};
```

```
// 构建规则：顺序应用规则
```

```
static Rule seq(List<Rule> rules) {
```

```
    return srcStr -> {
```

```
        for (Rule rule : rules) {
```

```
            Optional<String> result = rule.apply(srcStr);
```

```
            if (result.isPresent()) {
```

```
                return result;
```

```
            }
```

```
        }
```

```
    };
```

```
};
```

```
// 构建条件：包含条件
```

```
static Condition contains(String contained) {
```

```
    return str -> str != null && str.contains(contained);
```

```
};
```

```
// 构建条件：永真条件
```

```
static Condition alwaysTrue() {
```

```
    return str -> true;
```

```
};
```

```
// 构建条件：或条件
```

```
static Condition or(Condition condition1, Condition condition2) {
```

```
    return str -> condition1.test(str) || condition2.test(str);
```

```
};
```

```
// 构建条件：与条件
```

```
static Condition and(Condition condition1, Condition condition2) {
```

```
    return str -> condition1.test(str) && condition2.test(str);
```

```
};
```

```
// 构建条件：null或空白字符串条件
```

```
static Condition nullOrBlankString() {
```

```
    return or(nullString(), blankString());
```

```
};
```

# ▶ 字符串转换重构：增量代码生成

## ## 任务概述

你是一名Java资深程序员，请编写程序实现一个系统：

1. 系统的输入是"源字符串"。系统的输出是"Optional<String>"，其中String是转换后的字符串，如果转换成功，输出非empty的Optional<String>，如果转换失败，输出Optional.empty。
2. 实现如下面的Markdown表格描述的一套DSL，注意可组合性。  
| 核心概念 | 概念语义 | 原子/组合 | 形式化表达 |  
| --- | --- | --- | --- |  
| "null或空白字符串"条件 | 该条件应基于或条件生成，"null"条件与"空白字符串"条件相或。 | 组合条件 | or(nullString(), blankString()) |
3. 已实现的条件或规则未列入表格。相关的实现代码在``已有代码``中。
4. 请在``//TODO``处新增代码。

## ## 已有代码

```
``java
//规则
@FunctionalInterface
public interface Rule {
    Optional<String> apply(String srcStr);
}
...``
```

## ## 待实现的代码

```
``java
// "日期时间"规则。
static Rule dateTime(String format) {
    return srcStr -> {
        //TODO: 请完成"日期时间"规则的实现。该规则对接受的源字符串按照日期时间格式字符串进行日期时间格式处理后并输出。

        return Optional.of(srcStr);
    };
}
...``
```

## ## 任务重申

1. 请在``//TODO``处按要求新增代码。

```
``// "日期时间"规则
static Rule dateTime(String format) {
    return srcStr -> {
        // 使用DateTimeFormatter解析日期时间
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern(format);
        try {
            LocalDateTime dateTime = LocalDateTime.parse(srcStr, formatter);
            // 将日期时间转换为字符串并返回
            return Optional.of(dateTime.format(formatter));
        } catch (Exception e) {
            // 解析失败，返回Optional.empty
            return Optional.empty();
        }
    };
}
...``
```



# 字符串转换重构：代码缺陷修复

## ## 任务概述

你是一名Java资深程序员，请编写程序实现一个系统。请使用OpenJDK 8实现。请使用函数式编程，不要使用面向对象编程。

1. 系统的输入是"源字符串"。系统的输出是"Optional<String>"，其中String是转换后的字符串，如果转换成功，输出非empty的Optional<String>，如果转换失败，输出Optional.empty。

2. 实现如下面的Markdown表格描述的一套DSL，注意可组合性：

核心概念	概念语义	原子/组合	形式化表达
---	---	---	---

| "顺序应用"规则 | 该规则由"规则列表"构造而成。按照给定的顺序依次应用多条规则。当某条规则应用成功，则返回该条规则的应用结果；如果该条规则应用失败，则继续应用下一条规则 | 组合规则 | seq([rule]) |

3. 已实现的条件或规则未列入表格。相关的实现代码在"已有代码"中。

4. 请在"//FIXME"处修复代码缺陷。

## ## 已有代码

```
```java
//规则
@FunctionalInterface
public interface Rule {
    Optional<String> apply(String srcStr);
}
```
```

## ## 待修复的代码

```
```java
public static Rule seq(List<Rule> rules) {
    //FIXME: 按照给定的顺序依次应用多条规则。当某条规则应用成功，则返回该条规则的应用结果；如果该条规则应用失败，则继续应用下一条规则
}
```
```

## ## 任务重申

1. 请在"//FIXME"处按要求修复代码缺陷。

```
public static Rule seq(List<Rule> rules) {
    return srcStr -> {
        for (Rule rule : rules) {
            Optional<String> result = rule.apply(srcStr);
            if (result.isPresent()) {
                return result;
            }
        }
        return Optional.empty();
    };
}
```

# 字符串转换重构：无依赖代码的单元测试代码生成

## ## 任务概述

你是一名Java资深程序员。请你采用等价类与边界值的测试设计方法，使用JUNIT4生成所有“被测代码”的单元测试代码，注意每个类对应一个测试类、每个测试用例一个测试方法。

## ## 被测代码

```
``java
//规则
@FunctionalInterface
public interface Rule {
    Optional<String> apply(String srcStr);
}

class Rules {
    ...
}

//条件
@FunctionalInterface
public interface Condition {
    boolean apply(String srcStr);
}

class Conditions {
    ...
}
``
```

## ## 任务重申

1. 请使用JUNIT4生成所有“被测代码”的单元测试代码。
- \* 注意采用等价类与边界值的测试设计方法。
  - \* 每个类对应一个测试类、每个测试用例一个测试方法。

```
public class RulesTest {
```

```
@Test
public void testSeqRule() {
    List<P
    mappin
    mappin
    mappin
```

```
@Test
public void testSeqRule() {
    Rule rule1 = Rules.supplier("Value1");
    Rule rule2 = Rules.supplier("Value2");
    Rule rule3 = Rules.supplier("Value3");

    Rule rule = Rules.seq(Arrays.asList(rule1, rule2, rule3));

    assertEquals("Value1", rule.apply("Sample").orElse(null));
}

assert
assert
assert
assert
```

```
@Test
public void testChainRule() {
    Rule rule1 = Rules.suffix(" A");
    Rule rule2
    Rule rule3
```

```
Rule rule
assertEquals
```

```
public class ConditionsTest {
```

```
@Test
public void testOrCondition() {
```

```
@Test
public void testOrCondition() {
    Condition condition1 = Conditions.contains("A");
    Condition condition2 = Conditions.contains("B");
    Condition orCondition = Conditions.or(condition1, condition2);

    assertTrue(orCondition.apply("SampleA"));
    assertTrue(orCondition.apply("SampleB"));
    assertFalse(orCondition.apply("SampleC"));
}

asse
asse
```

# 字符串转换重构：有依赖代码的单元测试代码生成

## ## 任务概述

你是一名Java资深程序员。请你采用等价类与边界值的测试设计方法，基于“已有代码”使用JUNIT4生成“被测代码”的单元测试代码，注意每个类对应一个测试类、每个测试用例一个测试方法。

## ## 已有代码

```
```java
//条件
@FunctionalInterface
public interface Condition {
    boolean apply(String srcStr);
}
...
```

## ## 被测代码

```
```java
class Conditions {
    //“或”条件
    public static Condition or(Condition condition1, Condition condition2) {
        return srcStr -> condition1.apply(srcStr) || condition2.apply(srcStr);
    }
}
...
```

## ## 任务重申

1. 请基于“已有代码”使用JUNIT4生成“被测代码”的单元测试代码。
- \* 注意采用等价类与边界值的测试设计方法。
  - \* 每个类对应一个测试类、每个测试用例一个测试方法。

```
public class ConditionsTest {

    @Test
    public void testOrWithTrueTrue() {
        // 测试两个条件都返回true的情况
        Condition condition1 = srcStr -> true;
        Condition condition2 = srcStr -> true;
        Condition orCondition = Conditions.or(condition1, condition2);

        assertTrue(orCondition.apply("test"));
    }

    @Test
    public void testOrWithTrueFalse() {
        // 测试第一个条件返回true, 第二个条件返回false的情况
        Condition condition1 = srcStr -> true;
        Condition condition2 = srcStr -> false;
        Condition orCondition = Conditions.or(condition1, condition2);

        assertTrue(orCondition.apply("test"));
    }
}
```

```
@Test
public void testOrWithFalseTrue() {
    // 测试第一个条件返回false, 第二个条件返回true的情况
    Condition condition1 = srcStr -> false;
    Condition condition2 = srcStr -> true;
    Condition orCondition = Conditions.or(condition1, condition2);

    assertTrue(orCondition.apply("test"));
}

@Test
public void testOrWithFalseFalse() {
    // 测试两个条件都返回false的情况
    Condition condition1 = srcStr -> false;
    Condition condition2 = srcStr -> false;
    Condition orCondition = Conditions.or(condition1, condition2);

    assertFalse(orCondition.apply("test"));
}
```

```
@Test
public void testOrWithMixedCases() {
    // 测试混合情况
    Condition condition1 = srcStr -> srcStr.contains("a");
    Condition condition2 = srcStr -> srcStr.contains("b");
    Condition orCondition = Conditions.or(condition1, condition2);

    assertTrue(orCondition.apply("apple"));
    assertTrue(orCondition.apply("banana"));
    assertFalse(orCondition.apply("carrot"));
}
}
```

# ▶ 神兵利器效果评估

## SMDD(语义模型驱动设计)

做出好的设计，输出结构化表达的**语义模型**，使得LLM能够充分理解设计

LLM 高质量  
代码生成的  
神兵利器

## 提示模式

交互方式+提示模板，使得LLM基于**语义模型**高效生成高质量代码



### 一致性

- LLM生成的代码与设计的核心概念实现了一一对应
- LLM生成的代码与人编写的代码是相似的，甚至更好



### 可理解性

- LLM生成的代码容易理解，一方面是因为SMDD做出了好的设计，另一方面是LLM充分理解了该设计



### 可维护性

- 消除了原有代码中的“重复”与“相同逻辑不同实现”等问题，使得可维护性大幅度提升



# PART 04

## 总结&展望



# 总结

## SMDD(语义模型驱动设计)

做出好的设计，输出结构化表达的**语义模型**，使得LLM能够充分理解设计

LLM 高质量  
代码生成的  
神兵利器

## 提示模式

交互方式+提示模板，使得LLM基于**语义模型**高效生成高质量代码



### 适用场景

- 适用于软件复杂度高的场景



### 亮点

- 提示模式简单易学
- 多人多轮多案例验证，结果稳定、有效



# ▶ 展望

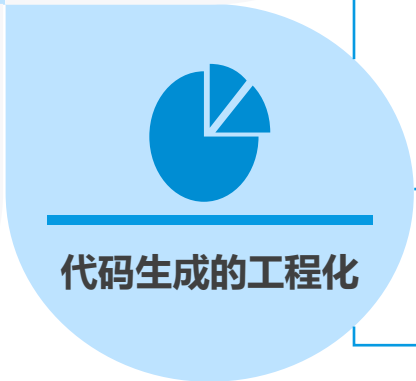
- 根据不同的LLM采用相应的提示模式
- Prompt中关于SMDD的核心内容是不变的、适用于多种大模型
- Prompt中的其他内容以及与LLM的交互方式需要调整，以适应不同的LLM



- 充分应用提示工程的基本原则
- 像跟人沟通一样跟LLM沟通



- 对软件架构提出了更高的要求
- 人员能力要求不降反升，既要具备软件设计、编码技能，又要掌握提示工程技能



- 提炼可落地、可推广的提示模式，尽量让LLM能够稳定生成代码，稳定不是100%准确
- 不建议过度追求100%的代码可用率
- 需要人来兜底

# THANKS

