第 1 章
低代码平台概述

本章是对低代码平台的全面概述，旨在为读者提供一个深入而清晰的认知框架。通过阅读本章内容，你将能够精准把握低代码平台的定义，洞悉其发展历程与当前态势，同时领略其在多领域中的广泛应用及显著优势。我们期望，通过本章内容的引导，能够激发你对低代码平台的浓厚兴趣，并让你对这一行业的未来发展充满信心与期待。

1.1 低代码平台的定义

低代码平台（Low-Code Platform）是一种软件开发工具，它允许用户（包括非研发角色）通过图形化界面和预构建的功能或模块，快速搭建业务功能，实现高效的投产上线。当然，在某些定制化场景下，可能需要辅助代码开发。该平台降低了编程的复杂性，使开发者能够专注于业务逻辑和用户体验，而无须关注底层的编程细节。下面来看一下笔者在日常工作中经常被问及的关于低代码平台的3个问题。

疑问1　低代码平台能实现所有业务场景需求吗？

低代码平台并不是万能的，它是为了解决特定业务场景而设计的。笔者不止一次听朋友提及，他们的研发总监希望将所有业务迁移到低代码平台，并要求团队自行调研和实施迁移。然而，目前还没有哪个低代码平台能够适用于所有的业务场景。在实际应用中，应以具体业务为切入点，逐步完善低代码平台的功能，尽量避免一开始就设计一个“大而全”的平台。这样不仅会加大研发投入，还会拉长整个研发周期。

举个例子，如果我们企业负责保险代理业务，每当与新的保险公司合作并代理其保险产品时，都需要对接新保险公司的接口，配置保险商品信息，并开发相应的产品推广页面。在未使用低代码平台时，每次对接一家新保险公司，研发团队都需要处理保险公司的鉴权、核保、投保、退保等接口对接工作。假如对接和联调一个接口需要3人天，那么整个流程下来需要12人天。

针对这个问题，我们可以先解决对接的难题，建立一个低代码保险公司对接平台。每当需要对接新保险公司或者对接旧保险公司的新接口时，只需要实施人员将接口协议按规则输入对接平台即可完成对接。对于保险产品的快速上线，可以开发一个低代码配置中心，快速生成配置页面。最后，解决C端不同保险产品的差异化展示问题，搭建一个拖曳式的低代码内容管理平台，生成C端页面链接进行推广投放。

通过一系列平台的组合，最终组成一个完善的低代码平台矩阵。后续若有新产品接入，无须研发介入即可实现。

疑问2　低代码平台只是供研发用的吗？

从上一个问题可以看出，低代码平台的使用对象并不仅限于研发人员。我们在开发过程中经常提到的抽象能力、通用架构和中台设计，主要是针对研发人员而言的。这些概念要么是为了减少研发投入，要么是为了解决后续开发的扩展性，或者是为了增强代码的可读性。

相比之下，低代码平台的核心聚焦于业务领域。其主要目标是实现项目的快速部署与投产，旨在最大限度地降低甚至完全消除研发成本，为研发部门以外的用户赋能，使他们能够迅速构建和部署各种业务场景。

疑问3　低代码平台会导致研发失业吗？

低代码平台并不等同于无代码（零代码）平台。其开发与后续的升级更新对研发人员提出了较高要求，尤其需要研发人员具备低代码平台的设计能力。这对于具有低代码开发经验的研发人员来说，是一个显著的优势。

对于业务逻辑复杂的企业，低代码平台能显著减轻研发负担，但并不能完全取代所有研发工作。然而，在业务相对简单、研发团队规模小于20人的小型企业中，低代码平台的应用使得项目开发完成后，仅需一名运维人员即可满足日常运营需求。

1.2　低代码平台的发展历史与现状

低代码平台作为软件开发领域的一次重大革新，自诞生以来便以其独特的方式和优势，深刻改变了应用程序开发的格局。其历史可以追溯到20世纪90年代至21世纪初，第四代编程语言和快速应用开发工具的出现为低代码平台的诞生奠定了基础。

随着时间的推移，低代码平台的概念在2014年正式确立，并开始引起业界的广泛关注。其核心理念是通过图形化界面和预构建模块，使开发者能够以少量甚至无须手动编程的方式，快速构建和部署应用程序。这一创新大幅提升了开发效率，降低了技术门槛，使更多非技术背景的业务人员也能够参与到应用程序的开发中。

在过去的几年里，低代码平台经历了快速发展。随着云计算、大数据、人工智能等技术的进步，低代码平台的功能日益丰富，应用场景愈发广泛。从企业内部的运营后台、数据面板、

办公系统，到B端的商品管理、广告投放、商铺搭建，再到C端的活动页面、促销频道、广告频道等，低代码平台展现出了强大的适应性和灵活性。

然而，随着市场的扩大和竞争加剧，低代码平台也面临一些挑战和机遇。首先，尽管低代码平台降低了技术门槛，但开发者仍需具备一定的业务知识和技术素养，才能充分发挥其优势。其次，每个平台往往具有特定的业务属性，适用于不同的行业和企业，因此市场上缺乏能够适用于所有场景的通用平台。此外，随着越来越多企业进入低代码平台市场，如何保持技术创新和服务质量成为每个平台都需要面对的问题。

展望未来，低代码平台将继续保持快速发展。一方面，随着技术不断进步和创新，低代码平台能够提供更加丰富的功能和更加灵活的配置选项，以满足不同行业和企业的需求。另一方面，随着数字化和智能化的深入推进，低代码平台将与云计算、大数据、人工智能等技术深度融合，为企业提供更加全面和高效的解决方案。同时，随着市场竞争加剧，低代码平台也需要不断提升技术实力和服务水平，以赢得更多市场份额和用户信任。

1.3　低代码平台与传统开发的比较

低代码平台与传统开发模式在多个方面存在显著的差异，这些差异使得低代码平台在快速应用开发、降低技术门槛以及提高开发效率方面展现出了独特的优势。

首先，从开发方式来看，传统开发主要依赖于程序员手动编写代码来构建应用程序。这要求开发者具备深厚的编程技术背景和丰富的项目经验。而低代码平台通过提供图形化界面和预构建模块，使开发者能够以拖曳、配置和编写少量代码的方式快速搭建应用程序。这种方式不仅降低了技术门槛，还让非技术背景的业务人员也能够参与到应用程序的开发中来。

其次，从开发效率来看，传统开发模式通常需要较长的开发周期和较高的成本。由于需要手动编写大量代码，并进行反复测试和调试，因此开发过程往往烦琐且耗时。而低代码平台通过预构建模块和自动化工具，大幅减少了手动编写代码的工作量，从而显著提高了开发效率。这使得企业能够快速响应市场需求，加速产品迭代和创新。

再次，从可维护性和扩展性方面来看，低代码平台也具有明显优势。由于低代码平台采用模块化和组件化的设计思想，应用程序的各个部分都是独立的且可重用的。这使得应用程序的维护和扩展变得更加容易和灵活。当需要修改或添加功能时，只需要调整相应的模块或组件即可，无须对整个应用程序进行大规模重构。

最后，从团队协作和项目管理方面来看，低代码平台也提供了更加高效和便捷的工具。通过可视化的项目管理界面和协作工具，团队成员可以实时查看项目进度、分配任务、进行代码审查等。这不仅提高了团队协作效率，还降低了沟通成本和出错率。

综上所述，低代码平台在开发方式、开发效率、可维护性和扩展性以及团队协作和项目管理等方面相比传统开发模式都展现出了明显的优势。这些优势使得低代码平台成为企业快速构建和部署应用程序的重要工具。

1.4 低代码平台的应用场景与优势

1.4.1 应用场景

低代码平台的应用场景十分广泛，不仅适用于大型企业内部的复杂系统构建，也能满足中小企业业务快速迭代和创新的需求。

在企业内部，低代码平台可用于快速开发企业级应用，如客户关系管理（Customer Relationship Management，CRM）、企业资源规划（Enterprise Resource Planning，ERP）、办公自动化（Office Automation，OA）等系统，提升企业内部管理的效率和灵活性。此外，对于电商、金融、医疗等特定行业，低代码平台能够快速搭建符合行业特性的定制化应用，如电商平台的商品管理系统、金融行业的风险管理系统、医疗行业的病历管理系统等。同时，低代码平台还适用于快速构建移动应用、微信小程序等，满足企业跨平台、多终端的服务需求。

无论是初创企业还是大型企业，低代码平台都能提供高效、灵活、定制化的解决方案，助力企业实现数字化转型和业务创新。

1.4.2 低代码平台的优势

在当今的程序员就业市场中，面对就业形势的不断变化，学习低代码技术显得尤为重要。低代码平台不仅为企业提供了降本增效、减少重复建设、加速业务上线的机会，也为个人开发者提供了提升技术能力、增强就业竞争力，并专注于核心业务代码输出的途径。

1. 对企业的优势

从企业的角度来看，推动低代码建设具有重要意义，这主要体现在以下几个方面。

1）为企业降本增效

低代码平台是实现降本增效的重要工具。随着市场竞争的加剧，企业需要迅速响应市场变化，推出新产品和服务以维持竞争力。低代码平台通过预构建模块和可视化界面，大幅减少了手动编程的需求，从而缩短了开发周期，降低了开发成本。此外，低代码平台还提供了丰富的功能和灵活的扩展性，使得企业能够根据自身需求定制应用程序，减少重复建设，提升整体效率。

2）实现业务快速上线

在业务快速上线的需求下，低代码平台发挥了关键作用。传统的开发模式通常需要经历冗长的开发周期和复杂的测试流程，而低代码平台能够大大缩短这一流程，使企业能够更快地推出新产品和服务，抢占市场先机。

3）保障系统稳定性

低代码平台通过模块化和组件化的设计，有助于保证系统质量的稳定性。在低代码平台中，应用程序的各个部分都是独立的、可重用的。这不仅降低了系统的复杂性，还提高了系统

的可维护性和可扩展性。当系统出现问题时，开发人员可以迅速定位并修复问题，从而保证系统的稳定运行。

4）为企业增收

对于希望将产品研发转向SaaS（Software as a Service，软件即服务）以实现增收的企业来说，低代码平台同样具有重要意义。SaaS模式要求企业能够快速、灵活地为客户提供定制化的服务。低代码平台能够快速构建和部署应用程序，并支持多租户架构，使得企业能够轻松地为客户提供个性化的服务，从而实现增收。

2. 对个人的优势

从个人的角度来看，学习低代码技术同样具有重要意义，这主要体现在以下几个方面。

1）提升个人竞争力

随着技术的不断发展和市场需求的变化，掌握低代码技术将使个人在就业市场上更具竞争力。低代码技术以其高效、灵活的特点，正逐渐成为企业构建应用程序的首选。因此，具备低代码技术能力的开发者将更受企业青睐。

学习低代码技术意味着掌握了一种新的开发工具和方法论。这不仅使个人在职业生涯中更加多元化和灵活，还增强了适应不同工作和项目需求的能力。

2）加快职业成长

低代码平台以其独特的优势，对研发人员的技术广度提出了更高的要求。为了充分利用平台的强大功能并应对多样化的项目需求，研发人员不得不深入学习各种相关技术知识，包括平台特有的开发框架、集成技术、数据处理方法以及第三方服务接口等。这一过程不仅丰富了个人技术能力，还促使研发人员在实践中不断积累经验，加速职业成长的步伐。低代码平台成为研发人员提升自我、拓宽职业道路的宝贵工具。

3）提高工作效率

低代码平台通过预构建模块和可视化界面，大幅减少了手动编程的需求，从而提高了开发效率。对于个人而言，这意味着可以在更短的时间内完成更多的工作，提高工作效率。

在开发过程中，低代码平台还提供了丰富的功能和灵活的扩展性，使得开发者能够更快地实现业务逻辑和满足用户需求，减少返工和修改的时间。

4）增强创新能力

低代码平台鼓励开发者通过可视化的方式思考和解决问题，这有助于激发个人的创新思维和创造力。通过快速构建和测试应用程序，开发者可以更快地验证想法和创意，从而实现产品的快速迭代和创新。

低代码平台还支持与其他技术和工具的无缝集成，如云计算、大数据、人工智能等。这使得开发者能够利用更多先进的技术和工具来构建更加智能、高效的应用程序，进一步提升个人的创新能力。

第 2 章
低代码平台入门指引

学习低代码技术是一个从理论到实践，再由实践深化理论的过程。对于没有接触过低代码平台、缺乏低代码研发经验的研发人员来说，入门并掌握这一技术无疑是一个挑战。本章将为读者提供一份详细的低代码技术入门与实战指南，帮助你逐步掌握低代码技术，从理论到实践，再由实践深化理论。

本章为你提供一条详细的学习路径，助你有效掌握低代码技术，实现从入门到精通的跨越。

首先，你需要夯实基础知识，本章将介绍低代码领域中的关键技术点，并引导你举一反三，灵活运用所学知识。其次，你需要学会将所学知识与实际开发场景相结合，这不仅能加深你对知识点的理解，还能提升你的实际应用能力。再次，我们将通过深入剖析经典实战案例，让你在理解案例设计的同时，掌握其背后的思考逻辑和最佳实践。最后，建议你亲自动手，尝试搭建一套低代码系统，将所学知识付诸实践，真正做到学以致用，将学习成果转换为实际能力。

通过这一完整的学习流程，你将逐步掌握低代码技术，为未来的职业发展奠定坚实基础。

2.1 掌握基础技术

学习低代码技术，首先需要掌握其基本概念和核心技术。这包括了解低代码平台的基本架构、功能组件以及常用的开发语言等。通过系统地学习这些基本知识，你可以对低代码技术有一个全面的认识，为后续的学习和实践打下基础。

在学习过程中，本书会为你介绍常见的低代码技术，并教你如何运用这些技术。但需要记住，学习不仅仅是记忆知识点，更重要的是学会举一反三。当你掌握了一种技术的运用方法后，要尝试将其应用到其他类似的场景中，通过不断的实践来加深理解。

2.2　掌握部分架构知识

低代码平台的设计除需要掌握基本的架构设计原则和常用的设计模式外，还非常注重架构的抽象能力。架构的抽象能力能够将复杂的业务逻辑、技术细节和系统组件简化为易于理解和操作的概念或模型，这对于实现低代码平台内部功能的模块化至关重要。

2.3　应用场景分析

在学习低代码技术的过程中，要学会将理论知识与实际的开发场景相结合。你可以尝试将所学知识应用到自己的项目中，通过实际操作来加深对知识点的理解。同时，也要关注行业内的最新动态和案例，了解低代码在实际应用中的最新发展和趋势。

为了将知识更好地融入日常开发场景，可以参加线上或线下的技术交流活动，与其他开发者分享经验和问题。通过交流和讨论，可以拓宽视野，获取更多的灵感和解决方案。

2.4　学习实战案例

学习实战案例是掌握低代码技术的关键步骤。通过深入剖析经典实战案例，你可以了解低代码平台的设计思路、实现过程以及优化方法。在学习过程中，不仅要理解案例的设计，更要学会如何将这些设计应用到自己的项目中。

后面的章节将详细讲解这些实战案例，并提供相应的代码和操作步骤。你可以跟随书中的指引，一步一步完成案例的学习和实践。通过实际操作，你可以更好地掌握低代码技术的核心要点，提升实战能力。

2.5　应用到工作场景

学以致用是学习低代码技术的最终目标。掌握基本知识、将知识融入日常开发场景并学习了实战案例后，就可以尝试自行搭建一套低代码系统。

在搭建系统的过程中，你需要综合运用所学知识，从需求分析、系统设计到开发实现，全程参与并主导。通过这个过程，你可以更深入地了解低代码技术的实际应用和挑战，同时锻炼自己的项目管理和团队协作能力。

最后，记住学习是一个持续的过程，不要害怕遇到问题或困难。通过持续的学习和实践，你会逐渐掌握低代码技术，并在实际项目中发挥出它的巨大价值。

第 3 章
低代码基础技术讲解

对于初次接触低代码平台的读者而言，学习低代码往往感到无从下手，不清楚需要掌握哪些基础技术和技能。为此，本章将详细介绍一系列常见的基础技术，旨在为你的低代码学习之旅打下坚实的基础。这些技术包括规则引擎、流程引擎、动态脚本语言、模板引擎、常见的数据交换格式等。通过掌握这些基础技术，你将在后续的案例实战中更加得心应手，将理论知识转换为实践能力。本章的讲解将为你的低代码学习之旅提供强有力的支持，帮助你在低代码开发的道路上迈出坚实的步伐。

3.1 规 则 引 擎

3.1.1 什么是规则引擎

规则引擎是一种应用程序中的组件，它从推理引擎发展而来，并被嵌入应用程序中。它使得业务决策能够从应用程序代码中分离出来，并通过预定义的语义模块来编写。简而言之，你可以提前定义好规则，当程序执行到相关代码时，会按照规则响应相应的结果。这些规则可以是POJO上定义的注释，以脚本的形式存放在文档中，或者是表达式和代码函数。

3.1.2 规则引擎在低代码平台中的作用

这里为什么要讲规则引擎呢？因为在诸如会员权益或活动任务等场景，通过低代码实现时，规则引擎可以减少因新增权益或新增活动任务而产生的开发工作量。以会员权益为例，产品经理为了提高日活，经常会新增各种权益，而每种权益又有各种各样的使用限制，也就是我们说的使用规则。按照传统的研发逻辑，每次产品经理需要新增一种或几种权益，都需要研发人员对权益进行开发，比如创建权益和编写权益的所有规则逻辑。然而，权益的规则通常是通

用的，比如使用门槛（等级门槛、注册时间、黑白名单、用户池）、使用限制（年月日可用、次数限制）、封顶控制（年月日次数上限、总上限）、金额计算、时长计算等；还可能需要对规则进行组合或切换顺序判断。如果使用规则引擎，只需要提前编写各种规则元素，然后就可以组装成一个权益。具体实现将在后面的案例中详细讲解。

3.1.3　有哪些规则引擎

目前市面上的规则引擎非常多，分为商业和开源两种类型，常见的有Drools、Easy Rules、URule、Aviator等。为了帮助读者了解和选择不同的规则引擎，笔者整理了一个表格供参考，如表3-1所示。

表 3-1　不同规则引擎的区别和选用

	Drools（JBoss Rules）	Easy Rules	Aviator	URule
语言/国家	Java/国外	Java/国外	Java/国内	Java/国内
定　　位	规则引擎	规则引擎	表达式求值引擎	规则引擎
特　　性	重量级、集成功能多、易用性好，支持Java代码嵌入规则文件；基于Rete算法，执行速度快	轻量级库、API 学习成本低；基于注解编程模型；定义抽象的业务规则并轻松应用它们；支持从简单规则创建组合规则的能力；支持使用表达式语言，如MVEL、SpEL 和 JEXL	支持数字、字符串、正则表达式、布尔值等基本类型，完整支持所有Java运算符及优先级等	以Rete算法为基础，提供了向导式规则集、脚本式规则集、决策表、交叉决策表（PRO版提供）、决策树、评分卡及决策流7种类型的规则定义方式，配合基于Web的设计器，可快速实现规则的定义、维护与发布
是否开源	开源	开源	开源	开源/商业
社区活跃	活跃度高	不活跃，2020年进入维护阶段	活跃度高	不活跃，2018年后未更新
规则配置方式	DRL规则文件 Excel决策树文件	JavaBean&注解 表达式语言 YML规则描述文件	脚本语言	Java类
可 视 化	WorkBench	不支持	不支持	网页版设计器
热 部 署	支持	支持	支持	开源版不支持
缺　　点	规则仍需要开发工程师维护；规则规模大了之后也不好维护；规则语法只适合扁平规则，对于嵌套规则的支持并不好；学习成本高	基于注解的POJO编程模型，类似于策略模式的实现，方便业务逻辑的隔离；如果使用YML文件的方式配置规则，功能上就没有Drools强大；未使用Rete算法，规则匹配效率较低	严格来说，只是表达式求值引擎，不适合复杂的业务逻辑使用	开源版本功能阉割较为严重，缺乏社区支持度。商业版本功能齐全，规则配置可视化程度高
协　　议	Apache License 2.0	MIT License	LGPL	Apache License 2.0

3.1.4 低代码平台推荐使用的规则引擎

日常工作对规则引擎的要求可能不高，读者可按业务需求选择。但是，低代码平台对规则引擎的选用会有一些特殊要求，并不是所有规则引擎都能够满足，比如以下要求。

1. 入参的灵活性

不同业务或权益会有不同的入参，没有固定的POJO；基于POJO注解编程模型的规则引擎无法支持这种灵活性。

2. 规则的多样性，也要求返回形式的多样性

在低代码平台中，规则可分为过滤型、过程型和展示型。过滤型规则返回 TRUE 或 FALSE；过程型规则返回特定数据结果；展示型规则返回前端展示的数据。规则结果的多样性要求规则引擎不仅能返回简单的 TRUE 或 FALSE，还能增强平台的拓展性。

3. 能够支持第三方调用和数据库查询

在众多业务场景中，规则常常需要调用容器内的接口，或者操作数据库，甚至调用外部接口，简单的表达式规则无法形成业务的闭环。例如，在会员权益业务中，假如权益内容为金卡会员，每日只能领取一张抵用券，我们需要调用会员中台接口，判断当前会员是不是金卡会员，还需要查询数据库，获取指定会员今日已领取该权益的次数。

4. 尽量减少多个规则引擎的使用

在进行规则引擎选型时，需要考虑业务未来发展的需求，尽量避免后续因缺少某项功能引入其他规则引擎。规则引擎配置方式的多样性，导致随意切换规则引擎会带来许多风险，同时会提高规则引擎切换的成本。

5. 自定义函数强大，支持指定语言编码（比如Java）

提供强大的自定义函数功能，允许研发人员根据产品需求通过代码实现复杂的规则运算和额外能力补充。例如，如果规则设定为打卡6次即可获得60积分，那么规则引擎需要能够查询数据库以计算打卡次数。此外，在某些场景下，可能还需要规则引擎支持日志上报统计通过规则的用户数，或者在规则执行异常时发送MQ消息进行异步重试。

6. 规则能够嵌套和灵活搭配

在实际应用场景中，我们经常需要使用嵌套和组合的规则。例如，在一个停车场业务场景中，商场原本对所有金卡用户免收停车费，最近为了促进银卡用户的消费，增加了一项权益：消费满300元的银卡用户也可免收停车费。这样的规则可以通过逻辑表达式组合来实现，如“会员等级=金卡 ||（会员等级=银卡 && 消费金额>300）”。

综合上述要求，在3.1.3节介绍的规则引擎中，可选的有Drools和Aviator。然而，Aviator基于LGPL协议开发，这可能导致部分企业在选型时有所顾虑。如果业务需求不是特别复杂，

可以选择Drools作为降级选项。当然，如果需要，也可以直接使用Java来实现简单的规则引擎。

3.1.5　Aviator 使用介绍

本小节介绍Aviator的使用。Aviator支持大部分运算操作符，包括算术操作符（+、−、*、/、%）、比较运算符（>、>=、==、!=、<、<=）、逻辑操作符（&&、||、！）、位运算操作符（&、|、^、<<、>>）、三元表达式（？:）、正则表达式（=~），还支持操作符的优先级和使用括号来强制优先级，并且支持对传入的参数进行运算。

1. 引入依赖

```
<dependency>
<groupId>com.googlecode.aviator</groupId>
<artifactId>aviator</artifactId>
<version>5.3.0</version>
</dependency>
```

2. 常见的使用方法

```
public static void main(String[] args) {
    // 支持算术运算符
    System.out.println(AviatorEvaluator.execute("(1 + 2 - 0) * 3 / 3 % 2")); //1
    // 支持比较运算符和逻辑运算符
    System.out.println(AviatorEvaluator.execute("3 > 2 && 2 != 4 || true")); //true
    // 支持位运算符
    System.out.println(AviatorEvaluator.execute("104 ^ 111")); //7
    // 支持三元表达式
    System.out.println(AviatorEvaluator.execute("6 > 2 ? 1 : 0")); //1
    //支持正则表达式: 正则表达式需要放在//之间
    System.out.println(AviatorEvaluator.execute("'128' =~ /[0-9]{3}/")); //true
    // 支持调用函数
    System.out.println(AviatorEvaluator.execute("string.length('abc')")); //3
    System.out.println(AviatorEvaluator.execute("string.contains('ABC', 'B')"));
    //true
    System.out.println(AviatorEvaluator.execute("math.pow(-3, 2)")); //9.0
    System.out.println(AviatorEvaluator.execute("math.sqrt(9.0)")); //3.0

    // 支持传参
    // 传参方式1: 通过exec，无须传递Map格式，不推荐使用
    String str = "这是内容";
    System.out.println(AviatorEvaluator.exec("'输出内容: '+ str", str)); // 输出内
容: 这是内容
    // 传参方式1: 通过execute，需要传递Map格式
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("paramA", "参数A");
```

```
        map.put("paramB", "参数");
        System.out.println(AviatorEvaluator.execute(" string.startsWith(paramA,
paramB)",map)); //true
    }
```

3. 自定义函数

要使用自定义函数，首先需要继承AbstractFunction类，重写getName和call方法。其中，getName方法用于定义函数的名称，而call方法用于实现函数的具体逻辑运算。

1）限制用户每日使用权益一次

```
    // 继承AbstractFunction，重写getName和call方法
    public class UserDayLimitFunction extends AbstractFunction {
        @Override
        public AviatorObject call(Map<String, Object> param) { //获取用户id
            String userId = String.valueOf(param.get("userId")); //获取用户当日已使用次数
            int times = ElementUtil.queryDayTimesByUserId(userId);
            int success = times > 1 ? 0 : 1;
            return new AviatorBigInt(success);
        }

        @Override
        public String getName() {
            return "userDayLimit";
        }
    }
```

2）限制每月用户使用权益 5 次

```
    // 继承AbstractFunction，重写getName和call方法
    public class MonthLimitFunction extends AbstractFunction {
        @Override
        public AviatorObject call(Map<String, Object> param) {
            // 获取用户id
            String userId = String.valueOf(param.get("userId"));
            // 获取用户当月已使用次数
            int times = ElementUtil.queryMonthTimesByUserId(userId);
            int success= times > 1 ? 0 : 1;
            return new AviatorBigInt(success);
        }
        @Override
        public String getName() {
            return "userMonthLimit";
        }
    }
```

4. 调试

```
public static void main(String[] args) {
    // 加载自定义函数
    AviatorEvaluator.addFunction(new UserDayLimitFunction());
    AviatorEvaluator.addFunction(new MonthLimitFunction());
    // 设置基本参数
    Map<String, Object> params = new HashMap<>();
    params.put("userId", "张三");
    // 判断用户是否有权益A
    // 通过数据库获取权益A规则表达式：用户只要未达月上限就可以使用权益
    String ruleA = "userMonthLimit()>0";
    System.out.println(AviatorEvaluator.execute(ruleA, params));
    // 判断用户是否有权益B
    // 通过数据库获取权益B规则表达式：用户不能超过月上限且不能超过日上限才可以使用权益
    String ruleB = "userDayLimit()>0 && userMonthLimit()>0";
    System.out.println(AviatorEvaluator.execute(ruleB, params));
}
```

通过上述案例，当运营人员熟悉Aviator表达式后，如果业务方想新增一个用户权益，运营人员只需通过配置表达式即可创建新的权益。举一反三，如果我们将配置表达式的能力转换为前端组件配置的形式：平台用户在创建新权益时，每个组件代表一个规则，用户只需将组件拖曳进权益配置界面中，然后平台用户只需配置每个规则所需的参数，即可完成新权益的创建。

3.2　流 程 引 擎

3.2.1　什么是流程引擎

流程引擎（Process Engine）是一种用于管理和执行业务流程的软件技术或工具。流程引擎基于一组节点与执行界面，通过人机交互的形式自动地执行与协调各个任务和活动。它可以实现任务的分配、协作、路由和跟踪。通过流程引擎，组织能够实现业务流程的优化、标准化和自动化，从而提高工作效率和质量。总的来说，它是一套低代码工具，能够帮助我们可视化地设计和修改业务流程。低代码平台、办公自动化（Office Automation，OA）、BPM平台和工作流系统均需要流程引擎功能。

流程引擎通常包括以下几个组成部分。

（1）流程设计器（Process Designer）：这是用来设计流程图的工具，它提供了一系列的节点、连线和规则，方便用户从画布中拖曳出工业流程图。用户可以使用建模工具创建业务流程模型，包括流程图、活动、决策、条件等元素，并定义流程的执行顺序、条件和规则。

（2）执行引擎：负责根据流程设计器的设计来执行实际的业务流程。执行引擎将业务流程抽象成可执行的流程模型，并自动化执行流程。

（3）监控工具：用于实时监控流程的执行情况，并提供报告。

流程引擎的核心功能包括流程建模、流程执行和流程监控。在流程建模阶段，用户可以使用建模工具来定义业务流程。在流程执行阶段，执行引擎会根据定义的流程来执行实际的业务操作。在流程监控阶段，监控工具会提供实时的流程执行情况和报告，以便用户了解流程的执行细节并进行相应的调整。

3.2.2 流程引擎在低代码平台中的作用

流程引擎是一个低代码工具，它允许我们在不编写代码的情况下，通过可视化界面来控制流程的转变。在OA系统中，流程引擎的应用非常广泛，而在企业级的低代码系统中，它同样具有重要的价值。例如，在CRM系统中，如果原有的流程包括：线索→拜访→机会→合同→门店→勘探→发货，那么当需要在合同签订后增加回访环节时，只需在合同和门店之间插入一个回访的执行模块，即可实现：线索→拜访→机会→合同→回访→门店→勘探→发货，而无须重新编写代码。

流程引擎的作用不仅限于此，它还可以根据当前流程的结果来控制流程的前进与后退。例如，如果机会未能转换为合同，流程引擎可以配置为自动触发第二次拜访的任务。

实际上，流程引擎可以发挥更大的作用，解决代码逻辑的问题。在代码开发过程中，我们经常进行if判断、for循环、调用接口、操作数据库等操作。如果将这些操作转换为可视化的操作，那么我们甚至可能不再需要敲代码。为了加深读者的理解，后面的案例中将对此进行详细讲解。

3.2.3 有哪些流程引擎

目前市面上存在多种流程引擎，其中一些主流的流程引擎包括Activiti、Flowable、Camunda、OSWorkflow、jBPM等。以下是这些流程引擎的简要介绍和对比。

1. Activiti

- 基于Java的轻量级业务流程引擎。
- 提供图形化的流程设计器和管理界面。
- 支持BPMN 2.0规范。

2. Flowable

- Activiti的分支，专注于企业级应用。
- 提供了更多的定制选项和扩展性。
- 提供了可视化建模工具。

3. Camunda

- 强大的BPM平台，支持微服务架构。
- 提供了对CMMN（Case Management Model and Notation，案例管理模型与符号）和DMN（Decision Model and Notation，决策模型与符号）的支持。
- 有强大的社区支持和丰富的文档。

4. OSWorkflow

- 完全用Java编写的开放源代码工作流引擎。
- 显著的灵活性，面向技术背景用户。
- 用户可以根据需求设计简单或复杂的工作流。

5. jBPM

- 提供灵活且可扩展的工具和API。
- 支持图形化的流程设计器。
- 可与规则引擎（如Drools）集成。

1）流程引擎的比较

Activiti、Flowable、Camunda、OSWorkflow 和 jBPM 五种流程引擎的比较如表 3-2 所示。

表 3-2　流程引擎的比较

特　　性	Activiti	Flowable	Camunda	OSWorkflow	jBPM
起　　源	基于jBPM派生	Activiti分支	全新开发	自主开发	自主开发
BPMN 2.0支持	是	是	是	是（通过插件）	是
图形化建模	是	是	是（Camunda Modeler）	否	是
规则引擎集成	有限	有限	是（DMN支持）	否	是（与Drools集成）
微服务架构支持	有限	有限	优秀	有限	有限
社区支持	适中	适中	强大	适中	适中
定制性和扩展性	适中	高	高	高	高
商业支持	有限	有限	提供	有限	提供

2）使用率

- Activiti：由于该引擎的轻量级和易用性，在中小企业和开源社区中较为流行。
- Flowable：作为Activiti的分支，由于该引擎在企业级应用中的优势和更多的定制选项，获得了较高的使用率。
- Camunda：由于该引擎强大的功能和支持微服务架构的能力，在大型企业和复杂业务流程管理场景中较为流行。
- OSWorkflow：虽然该引擎在特定领域（如ERP、CRM等）有一定的用户基础，但相对于其他引擎，其整体使用率可能较低。

- jBPM：作为一个成熟的BPM解决方案，该引擎在Java社区中有一定的用户群体，特别是在需要与规则引擎集成的场景中。

3.2.4 低代码平台推荐的流程引擎

如果只是实现简单的流程切换，前面介绍的流程引擎可以根据实际业务需求进行选择。然而，在低代码平台中，特别是在大企业的低代码平台中（小企业可能不会投入大量资源），我们往往会选择自行搭建一个符合业务需求的流程引擎。正如前面提到的，在低代码平台中，我们不仅需要处理if和for这样的基本逻辑，还需要处理参数透传、方法调用、消息同步、数据库操作等复杂操作，以及各种触发器。这些操作可能需要我们自行开发才能满足特定的业务需求。

3.3 动态脚本语言

3.3.1 什么是动态脚本语言

动态脚本语言（Dynamic Scripting Language）是一类在运行时解释执行的编程语言，通常具有灵活、易读、易写和易于学习的特点。这些语言通常不需要编译为机器码，而是直接由解释器或虚拟机执行源代码。动态脚本语言通常用于快速开发、原型设计、网页开发、自动化脚本、数据处理等多种应用场景。

动态脚本语言的主要特点如下。

- 解释执行：动态脚本语言不需要像传统编译型语言那样先编译成机器码再执行，而是直接由解释器读取源代码并执行。这使得代码的开发和调试过程更加快速和便捷。
- 动态类型：动态脚本语言通常支持动态类型，即在运行时确定变量的类型。这意味着程序员不需要在声明变量时指定其类型，而是在运行时根据变量的值来推断其类型。这种灵活性提高了编程的便捷性，但也可能牺牲一些性能。
- 易读易写：动态脚本语言的语法通常较为简洁和直观，易于学习和使用。这使得它们成为初学者和快速开发项目的理想选择。
- 跨平台性：由于动态脚本语言通常不依赖于特定的硬件或操作系统，因此它们通常具有很好的跨平台性。只需安装相应的解释器或虚拟机，就可以在任何支持该语言的平台上运行脚本。
- 快速开发：动态脚本语言的快速开发和迭代能力使它们非常适合用于原型设计和快速构建应用程序。开发人员可以快速编写和测试代码，然后根据需要进行调整和优化。
- 与Web开发密切相关：许多动态脚本语言（如JavaScript、PHP、Python等）在Web开发领域具有广泛应用。它们可以用于构建Web应用程序、处理用户输入、与数据库交互等任务。
- 扩展性：动态脚本语言通常具有强大的扩展能力，可以通过插件、库和框架等方式扩展其功能。这使得它们能够适应各种复杂的业务需求和技术挑战。

3.3.2　动态脚本语言在低代码平台中的作用

由于动态脚本语言具有解释执行的特性，结合低代码平台的优势，我们可以了解到动态脚本语言在低代码平台中的应用具有重要意义。在流程引擎中，我们通常需要调用其他方法或外部接口。如果每次都需要编写代码来对接外部接口，那么使用动态脚本语言可以直接在流程引擎中编写动态脚本函数，无须重启服务。当流程执行过程中检测到动态脚本语言时，可以直接执行这些脚本。

3.3.3　有哪些动态脚本语言

常见的动态脚本语言有Python、JavaScript、Ruby、Perl、PHP和Groovy，它们的区别如下。

1. Python

特性：Python是一种解释型的高级通用编程语言，其设计哲学特别强调代码的可读性，使用缩进来定义代码块。此外，Python支持多种编程范式，包括面向过程和面向对象编程。

优点：简洁易读、跨平台、拥有庞大的库和框架生态系统（如Django、TensorFlow等）、支持多种应用场景（如Web开发、数据分析、人工智能等）。

缺点：相对于编译型语言，执行速度可能较慢。

2. JavaScript

特性：JavaScript是一种客户端脚本语言，主要用于Web浏览器，使得开发者能够在网页上实现动态内容和交互功能。

优点：与HTML和CSS紧密结合，可以直接操作DOM；支持异步编程和事件驱动模型；广泛应用于前端开发和单页面应用（Single Page Application，SPA）。

缺点：浏览器兼容性问题；全局作用域和变量提升可能导致一些不易察觉的错误。

3. Ruby

特性：Ruby是一种面向对象的解释型编程语言，以简洁、易读和优雅著称，旨在让程序员感到快乐和富有生产力。

优点：语法简洁易读；具有丰富的库和框架（如Ruby on Rails）；元编程能力强。

缺点：执行速度相对较慢；社区相对较小（与Python和JavaScript相比）。

4. Perl

特性：Perl是一种解释型的通用编程语言，特别擅长文本处理，并且支持正则表达式。

优点：强大的文本处理能力；适合进行系统管理、网络编程和CGI编程。

缺点：语法较复杂，学习曲线较陡峭；在某些应用场景下，性能可能不如其他语言。

5. PHP

特性：PHP是一种服务器端脚本语言，主要用于Web开发。它支持多种数据库，可以轻松地与MySQL等数据库进行交互。

优点：易于学习；与MySQL结合紧密；拥有大量的Web开发框架和CMS系统（如WordPress）。

缺点：安全性问题（如SQL注入等）；在某些复杂的应用场景下，性能可能受限。

6. Groovy

特性：Groovy是一种基于JVM（Java Virtual Machine，Java虚拟机）的面向对象编程语言。它的语法与Java非常相似，但更加简洁和灵活，且可以与Java无缝集成，允许在Java项目中直接使用Groovy代码。

优点：简洁易读的语法；与Java无缝集成；支持动态类型；可以编写DSL（Domain-Specific Language，领域特定语言）。

缺点：相对于原生Java，性能可能略低；社区相对较小（与Java和Python相比）。

为了让读者更直观地对比它们之间的差异，整理了表3-3方便读者查阅。

表 3-3　6 种动态脚本语言之间的对比

脚本语言	特　　性	优　　点	缺　　点	主要应用场景
Python	简洁易读、面向对象、跨平台	易于学习、丰富的库和框架、可读性强	执行速度相对较慢	网页开发、数据分析、人工智能、机器学习等
JavaScript	浏览器内执行、事件驱动、面向对象	交互性强、与HTML/CSS紧密结合	安全性问题、依赖浏览器	Web前端开发、动态网页、移动应用前端
Ruby	简洁易读、面向对象、元编程能力强	易于学习、开发效率高、社区活跃	执行速度相对较慢	Web开发（Ruby on Rails）、脚本编写
Perl	文本处理能力强、正则表达式支持	强大的文本处理能力、系统管理工具	语法较复杂、学习曲线较陡峭	系统管理、网络编程、Web开发
PHP	服务器端脚本语言、支持多种数据库	易于学习、与MySQL结合紧密	安全性问题、性能瓶颈	Web开发、电子商务平台、内容管理系统
Groovy	基于JVM、与Java兼容、性能优越	简洁的语法、与Java无缝集成	相对于原生Java性能略低	脚本编写、与Java协同开发、DSL开发、Web开发

3.3.4　低代码平台推荐

基于国内庞大的Java开发群体，从Java的兼容性、学习成本和广泛应用考虑，毫无疑问我们推荐使用Groovy。对于已经拥有Java开发环境的团队来说，引入Groovy几乎不需要额外的配置和学习成本。这种无缝集成使得Groovy在低代码平台中备受青睐，因为它可以充分利用现有的Java生态系统和工具。Groovy以其简洁的语法、与Java的无缝集成、动态编程和灵活性以

及丰富的特性和库等特点，成为低代码平台推荐使用的编程语言之一。下面列举了Groovy在低代码平台中的几个主要应用场景。

1. 流程引擎自定义逻辑

在前面讨论流程引擎时提到，流程引擎需要执行方法调用、消息同步、数据库操作以及触发器操作等任务。为了满足不同业务流程的定制化需求，流程引擎必须具备高度的灵活性和可配置性。Groovy作为一种动态脚本语言，在这方面展现出了独特的优势。具体表现如下：

- 接口暴露：Java服务只需暴露一些通用的接口（如消息队列操作、数据库操作、触发器操作等），供Groovy调用。
- 脚本配置：在流程引擎的各个流程节点中，可添加Groovy脚本，并在保存流程时将Groovy脚本保存到数据库或文件中。
- 脚本执行：当流程引擎启动并按流程定义顺序执行各个节点时，一旦遇到包含Groovy脚本的节点，就会触发该脚本的执行。Groovy脚本根据节点配置和传入参数执行相应的操作，并将结果返回给流程引擎。流程引擎根据执行结果继续执行后续节点或进行相应的处理。

2. API网关接入

Groovy在API网关领域的应用得益于其作为灵活脚本语言的特性，能够为API网关提供动态配置、自定义逻辑以及扩展功能。具体来说，Groovy在API网关中的应用场景包括以下几个方面。

- 动态路由和流量控制：Groovy可以用于编写自定义的路由规则，根据请求的属性（如请求头、请求路径、请求参数等）进行动态路由，从而实现流量的分发和控制。例如，可以根据请求的来源IP、用户身份等信息，将请求路由到不同的后端服务或进行限流处理。
- 请求和响应的转换：在API网关中，Groovy可以用于编写自定义的请求和响应转换逻辑。例如，可以对入站请求进行解析、验证和转换，以适应后端服务的接口要求；对出站响应进行格式化、加密和压缩，以满足客户端的需求。
- 自定义身份验证和授权：Groovy可以用于编写自定义的身份验证和授权逻辑。例如，可以实现基于JWT（JSON Web Token）的身份验证，或者根据用户的角色和权限信息进行访问控制。
- 日志和监控：Groovy还可以用于编写自定义的日志和监控逻辑。例如，可以记录每个请求的详细信息，包括请求时间、请求路径、请求参数、响应状态码等，便于后续分析和排查问题。
- 扩展和定制功能：由于Groovy是一种灵活的脚本语言，开发者可以根据具体需求扩展和定制API网关的功能。例如，可以编写自定义的过滤器、拦截器或插件，以实现对API调用的特殊处理或增强功能。

Groovy在API网关中的应用可以为开发者提供更大的灵活性和扩展性，使其能够更快速地构建和部署满足业务需求的API网关。同时，由于Groovy与Java的无缝集成，开发者可以充分利用Java生态系统的丰富资源和工具，从而提高开发效率和系统稳定性。

3. 规则引擎

Groovy在实现规则引擎方面有着天然的优势，因为它是一种基于JVM的、灵活且易于理解的动态语言。规则引擎通常用于将业务逻辑从代码中分离出来，以便在不修改代码的情况下修改业务规则。以下是一个简单的步骤，说明如何使用Groovy来实现一个简单的规则引擎。

1）定义规则

首先，需要定义自己的业务规则。这些规则可以以Groovy脚本的形式存在，因为Groovy允许编写类似于Java的代码，但使用更加简洁和灵活的语法。

例如，定义一个名为getDiscount.Groovy的Groovy脚本文件，其中包含一个或多个规则函数：

```
// 定义检查折扣条件的函数
def checkDiscount(order) {
    // 如果订单金额大于1000且积分大于60，返回true，否则返回false
    order.amount > 1000 && order.point > 60
}

// 定义计算折扣的函数
def getDiscount(order) {
    if (checkDiscount(order)) {        // 如果满足折扣条件
        return order.amount * 0.2      // 计算并返回折扣金额：订单金额的20%
    } else {
        return 0.0                     // 如果不满足折扣条件，返回0.0
    }
}
```

2）加载和执行 Groovy 脚本

在Java应用程序中，可以使用Groovy的GroovyScriptEngine或GroovyShell来加载和执行Groovy脚本。

例如，使用GroovyShell：

```
import Groovy.lang.GroovyShell;  // 导入GroovyShell类，用于加载和执行Groovy脚本
import Groovy.lang.Script;       // 导入Script类，用于表示Groovy脚本
public class RuleEngine {
    // 声明GroovyShell对象，用于加载和执行Groovy脚本
    private GroovyShell GroovyShell;
    public RuleEngine() {         // 构造函数，初始化GroovyShell对象
        this.GroovyShell = new GroovyShell();  // 创建GroovyShell实例
    }

    // 执行Groovy脚本并返回结果
    public Object executeScript(String scriptName, Object... args) {
        try {
```

```
                // 加载Groovy脚本，解析指定路径的Groovy脚本文件
                Script script = GroovyShell.parse(new File(scriptName));
                // 执行脚本中的方法，并传入参数，调用Groovy脚本中的getDiscount方法并传入参数
                return script.invokeMethod("getDiscount", args);
            } catch (Exception e) {
                e.printStackTrace();        // 捕获异常并打印堆栈信息
            }
            return null;                    // 如果执行过程中出现异常，返回null
        }
        public static void main(String[] args) {
            // 创建RuleEngine实例
            RuleEngine ruleEngine = new RuleEngine();
            // 创建一个Order对象，模拟订单信息
            Order order = new Order(90,61);
            // 调用executeScript方法执行Groovy脚本，计算折扣
            double discount = (double) ruleEngine.executeScript("getDiscount.Groovy",
order);
            // 输出计算出的折扣金额
            System.out.println("Discount: " + discount);
        }
    }
```

注意：你需要处理Groovy脚本的加载与可能的异常。此外，为简化示例，在这里直接调用了getDiscount方法。在实际应用中，可能需要根据规则名称或标识符等更复杂的逻辑来调用不同的规则函数。

3）扩展和维护

由于规则是以Groovy脚本的形式存在的，因此可以在不修改Java代码的情况下轻松修改或添加新的规则。只需编辑Groovy脚本文件并重新运行应用程序即可实现，这大幅增加了系统的灵活性和可维护性。

4）性能和安全性考虑

虽然Groovy提供了很多便利，但在生产环境中使用时，仍然需要关注性能和安全性问题。例如，确保你的Groovy脚本是安全的，不会执行恶意代码。此外，对于需要高性能的应用程序，可能需要仔细评估Groovy的性能表现，并考虑使用其他技术或优化策略来提高性能。

4. 拓展Saas功能

在SaaS（Software-as-a-Service）应用中，为了满足不同客户的需求，提供灵活且易于定制的解决方案至关重要。尽管SaaS平台通常提供一系列标准化的功能作为其核心服务，但每个客户都有其独特的业务流程和特定的业务需求，这要求SaaS应用必须具备定制化和可拓展性。Groovy作为一种强大的脚本语言，在这方面发挥了重要作用。

5. 定制化操作

当客户需要对SaaS应用进行定制化操作时，Groovy脚本的引入极大地简化了这一过程。通过编写Groovy脚本，客户能够直接针对特定的业务流程或功能进行定制，而无须依赖开发团队进行代码级的修改。这种灵活性使SaaS应用能够更好地适应不同客户的需求，提高客户满意度和忠诚度。

例如，一个CRM（Customer Relationship Management，客户关系管理）SaaS应用可能提供了通用的客户信息管理、销售机会跟踪等功能。然而，某个客户可能希望根据自身的业务特点添加特定的字段、验证规则或业务逻辑。通过Groovy脚本，客户可以轻松实现这些定制化需求，而无须等待开发团队进行漫长的开发和部署。

除定制化操作外，Groovy还允许客户对SaaS应用的功能进行拓展。当客户发现现有的功能无法满足其业务需求时，他们可以通过编写Groovy脚本来添加新的功能或拓展现有功能。这种拓展性使得SaaS应用能够保持与时俱进，满足不断变化的业务需求。

以ERP（Enterprise Resource Planning，企业资源规划）SaaS应用为例，它可能提供了库存管理、采购管理、生产管理等核心功能。然而，随着业务的发展，客户可能希望添加新的模块或功能，如质量管理、项目管理等。通过Groovy脚本，客户可以自主开发这些新功能，并将其集成到现有的ERP系统中，从而实现功能的快速拓展和升级。

3.4 模 板 引 擎

3.4.1 什么是模板引擎

作为Web开发领域的重要工具，模板引擎是构建动态内容生成流程的核心组件。它的核心作用是将静态的模板文件与动态的数据进行智能结合，以生成多样化的、用户定制的HTML、XML或其他文档类型。这一过程不仅提升了内容生成的效率，同时大幅简化了开发和维护的复杂度。

模板引擎的工作原理基于“占位符”机制。在模板文件中，开发者预先定义了多个占位符，这些占位符将用于填充动态数据。当模板引擎接收到实际的数据输入时，它会遍历整个模板文件，查找所有占位符，并用对应的数据进行替换。这一过程确保了内容的动态性和个性化。

除简单的数据替换外，模板引擎还支持更为复杂的逻辑操作。通过向模板文件中插入变量、条件语句、循环结构等控制语句，开发者可以定义更复杂的逻辑规则。这些规则允许模板引擎根据数据的不同属性或状态，动态地改变输出内容的结构和样式。例如，当某个数据项的值超过某个阈值时，模板引擎可以自动改变页面的颜色或显示特定的警告信息。

模板引擎的引入使得Web开发过程更加灵活和高效。开发者可以专注于编写清晰、易于维护的模板文件，而将复杂的数据处理和逻辑运算交给后端程序来处理。这种分工合作的方式不

仅提高了开发效率，也使得整个Web应用更加易于扩展和维护。同时，由于模板引擎支持多种文档类型的输出，因此它可以广泛应用于需要动态内容生成的各种场景，如电子邮件、配置文件、报告等。

3.4.2　模板引擎的应用场景

模板引擎在低代码平台中的应用具有诸多优势，特别是当使用FreeMarker等强大模板引擎时，其应用变得更加广泛和高效。下面介绍几个具体的应用场景。

- 渲染页面结构：模板引擎允许开发人员通过创建可复用的模板文件，轻松定义用户界面结构。这些模板可以包含HTML标记、CSS样式和JavaScript脚本，并通过变量、宏等特性实现动态内容的插入。开发人员只需将模板与业务数据相结合，即可快速渲染出符合需求的用户界面，极大地提高了开发效率。
- API服务平台：在调用第三方接口时，数据格式和字段名称的匹配往往是一个烦琐的任务。然而，通过使用模板引擎，开发人员可以轻松地将输入的数据映射成第三方接口所需的层次结构和字段名称。通过定义模板文件，开发人员可以指定数据如何被格式化、转换和传递给API，从而简化与第三方服务的集成过程。
- 约束输出内容：在设计通用数据库查询接口时，直接输出查询结果可能会导致敏感信息泄露或格式混乱。通过使用模板引擎，开发人员可以在查询结果返回之前，通过模板定义约束输出的内容。例如，可以使用模板过滤掉敏感字段、格式化日期和数字、添加HTML标记等。这样，开发人员可以确保仅呈现格式正确的信息，提高了系统的安全性和可读性。
- 数据变动无须重启服务：部分模板引擎支持模板的实时编译和加载。当模板文件发生变动时，模板引擎可以自动重新编译模板并更新缓存。这意味着在开发过程中，开发人员可以随时修改模板文件，而无须重启整个服务或应用程序。这种热更新能力使开发人员能够更快地看到修改后的效果，并缩短开发周期。

3.4.3　有哪些模板引擎

本小节将介绍几种常见的模板引擎。

- FreeMarker是一个用Java语言编写的模板引擎，可以生成HTML、XML或任何文本文件。它支持丰富的模板语言，允许开发人员通过变量、宏、条件语句等来动态生成内容。
- Thymeleaf是一个用于Web和独立环境的现代服务器端Java模板引擎。它完全支持HTML5，并允许在模板中使用HTML属性进行模板化。此外，它与Spring Framework紧密集成。
- Velocity是一个基于Java的模板引擎，允许开发人员使用模板语言来引用由Java代码定义的对象。它常用于动态内容生成，如Web页面、电子邮件和源代码等。
- JSP(JavaServer Pages)是Java EE平台的标准技术之一，允许在HTML页面中嵌入Java代码。JSP页面被编译成Servlet来执行，并可以生成动态内容。
- Handlebars是一个简单、高效且易于使用的模板引擎，采用Mustache模板语法。Handlebars提供了条件、迭代和辅助函数等功能，使模板编写变得简单直观。

这几种常见的模板引擎的特点如下。

- FreeMarker：功能强大，支持丰富的模板语言；易于集成到Java项目中；模板文件与代码分离，易于维护。
- Thymeleaf：完全支持HTML5；可以与Spring Framework紧密集成；支持国际化；提供丰富的表达式语言。
- Velocity：简单易用；性能良好；可以与Java代码紧密结合；支持模板继承和包含。
- JSP：Java EE标准技术；可以直接在HTML页面中嵌入Java代码；支持自定义标签库；性能良好。
- Handlebars：语法简洁直观；易于学习和使用；支持条件、迭代和辅助函数等功能；可以与多种后端语言结合使用。

常见的模板引擎的对比如表3-4所示。

表 3-4　常见的模板引擎的对比

模板引擎	语言支持	模板语言丰富性	与Java集成	性　　能	HTML5支持
FreeMarker	Java	高	良好	中等	良好
Thymeleaf	Java	中等	优秀（与Spring集成）	中等	优秀
Velocity	Java	中等	良好	优秀	良好
JSP	Java	高（Java代码嵌入）	优秀（Java EE标准）	优秀	良好
Handlebars	多种语言	中等	取决于后端实现	中等	优秀

请注意，表3-4中的“性能”和“模板语言丰富性”等指标是相对的，并且可能因具体使用情况而有所不同。此外，模板引擎的选择还应考虑项目的具体需求和开发环境。

3.4.4　推荐模板引擎

在低代码平台中，模板引擎之间的差异并不显著。对于前面介绍的模板引擎，读者可以根据自己的业务需求进行选择。但是，在低代码平台中，需要特别关注热更新问题，尽量减少服务重启，避免影响用户体验和业务的稳定性。FreeMarker本身并不直接处理热重载（hot-reloading），但可以在应用程序或框架层面实现热重载来解决这个问题。另一方面，Thymeleaf在Spring Boot环境中可以通过结合使用spring-boot-devtools，并在application.yml文件中设置spring.thymeleaf.cache:false来实现热更新。这样，在修改Thymeleaf模板后，无须重启服务，改动即可生效。

至于其他模板引擎，如Velocity、JSP和Handlebars，它们是否支持热更新，取决于具体的实现和使用环境。在某些框架或开发环境中，可能会提供热更新的支持或插件。因此，如果你需要使用热更新功能，建议查看所使用模板引擎的官方文档或社区资源，以了解如何在项目中实现这一功能。

3.5　数据交换格式JSON和Protobuf协议

3.5.1　为什么需要 JSON 和 Protobuf 协议

在讨论模板引擎的同时，我们也提及JSON和Protobuf（Protocol Buffer）协议。尽管本小节的重点并非深入探讨序列化协议，但由于它们都是序列化协议，因此一并讨论。选择在本节讲解JSON和Protobuf，是因为它们在低代码平台中扮演着特定的角色。JSON已经广为人知，因此无须赘述。对于Protobuf，我们将简要介绍。

Protobuf是Google开发的一种数据序列化协议，提供了一种语言无关、平台无关、可扩展的机制，用于以向前兼容和向后兼容的方式序列化结构化数据。与JSON和XML类似，Protobuf也是一种数据交换格式，但它更加紧凑、快速且简单。Protobuf的主要特点如下：

- 语言无关：支持多种语言，包括Java、Python、C++、JavaScript、Go以及其他语言。
- 平台无关：可以生成不同语言的代码，并在任何环境中运行。
- 性能好、扩展性好：相比JSON和XML等，Protobuf的序列化和反序列化性能更高，平均每秒可以处理10万条消息。
- 版本兼容性：具有一定的向后兼容性，可以在不破坏现有数据结构的情况下扩展和修改数据格式。
- 强类型：数据结构在编译时定义，有助于防止数据类型错误，从而提高代码的稳定性。
- 可读性和可维护性：尽管Protobuf的二进制编码不像XML和JSON那样易于人类阅读，但Protobuf的定义是文本格式的，易于理解和维护。

Protobuf的应用场景包括数据存储和交换、网络通信、序列化和反序列化等。由于其紧凑的二进制格式和高效的解析能力，Protobuf在需要处理大量数据的场景中具有显著优势。

3.5.2　JSON 和 Protobuf 协议的应用场景

JSON和Protobuf协议在低代码平台中的应用较为广泛，以下通过两个案例进行讲解。

1. 映射API接口协议

结合模板引擎，可以定义API接口协议。以下是使用FreeMarker模板引擎定义的模板内容，采用JSON格式。JSON结构与第三方接口的格式一致，JSON中的Key字段为第三方接口字段，${}中的数据则是服务内的数据。

```
{
    "name": ${categoryName},
    "category": ${category},
    "field_1PTYg__c": ${productCode},
    "field_zbG9h__c": ${userName},
```

```
    "field_sAsr6__c": ${field1},
    "field_09Ig2__c": ${field2},
    "dataObjectApiName": "ProductObj",
    "owner": [],
    "record_type": "default__c"
}
```

在调用第三方接口中，首先需要从数据库中查询数据并组装成实体对象。接着，将这些数据传入FreeMarker模板引擎进行渲染，转换为最终的JSON格式。最后，我们使用这些JSON数据来调用第三方接口，完成整个接口调用流程。具体的代码实现和详细步骤将在后续的实战案例中进行讲解，此处不再展开。

2. 定义页面结构：生成式页面和管理后台

管理后台和配置中心的工作虽然技术含量不高，但却占用了我们大部分的开发时间。通过采用低代码技术构建这些系统，可以显著释放研发团队的人力资源。此外，JSON和Protobuf在定义生成式页面方面发挥了重要作用，它们能够提供页面结构的精确描述。例如，在创建一个商品管理后台页面时，该页面包含商品名称、商品图片和商品类型等元素。我们可以使用JSON来定义这些内容，如下所示：

```
{
  "tag": "商品",
  "item": [
    {
      "field": "goodsName",
      "describe": "商品名称",
      "type": "String",
      "required": true,
      "uniq": true
    },
    {
      "field": "goodsImg",
      "describe": "商品图片",
      "type": "Image",
      "required": true,
      "uniq": false
    },
    {
      "field": "goodsType",
      "describe": "商品类型",
      "type": "Select",
      "options": {
        "Card": "卡",
        "Coupon": "券"
```

```
      },
      "required": true,
      "uniq": false
    }
  ]
}
```

也可以使用Protobuf来定义：

```
syntax = "proto3";
package configpb;
/**
 * name: 商品
 */
message Goods {
    /**
     * name: 商品名称
     * uniq: true
     * required: true
     * type: String
     */
    string goodsName = 1;
    /**
     * name: 商品图片
     * required: true
     * type: Image
     */
    string goodsImg = 2;
    /**
     * name: 商品类型
     * required: true
     * type: Select
     * options:
     *     - text: 卡
     *       value: 'Card'
     *     - text: 券
     *       value: 'Coupon'
     */
    string goodsType = 3;
}
```

3.5.3　是否有其他替代方案

除JSON和Protobuf外，还有其他一些常用的数据交换格式，具体说明如下：

- YAML（Yet Another Markup Language）：一种轻量级的文本格式，易于阅读和使用，常用于配置文件和数据交换。YAML通过缩进来表示结构，而不是使用复杂的标签或嵌套代码。
- CSV（Comma-Separated Values）：一种简单的表格形式的数据格式，用逗号分隔不同的值。CSV常用于将数据从一个软件传输到另一个软件，如电子表格程序之间的数据交换。
- XML（eXtensible Markup Language）：一种可扩展的标记语言，用于描述和传输结构化数据。XML具有自我描述性，支持多种编程语言，常用于Web服务和配置文件中。
- HTML：虽然HTML主要用于网页内容的展示，但在某些情况下，也可以作为数据交换格式使用。例如，通过Ajax等技术，可以在客户端和服务器之间传输HTML格式的数据。
- Smile：一种二进制格式的JSON，类似于Protobuf，但更接近JSON的结构。Smile的序列化结果相较于JSON更为紧凑和高效，并且支持JSON的所有特性。
- BSON（Binary JSON）：一种类JSON的二进制序列化格式，用于存储和传输数据。BSON在MongoDB数据库中广泛使用，因为它支持更丰富的数据类型，并且具有更高的性能和更小的存储需求。
- Avro：一种数据序列化系统，用于支持大量数据的存储、处理和传输。Avro数据文件是自描述的，并且具有紧凑的二进制格式，可以跨语言进行读写。

每种数据交换格式都有其独特的优点和适用场景，可根据具体需求选择合适的格式进行数据传输和交换。常见数据交换格式的对比如表3-5所示。

表 3-5　常见数据交换格式的对比

数据交换格式	描　　述	特　　点	优　　点	缺　　点
YAML	轻量级文本格式	简洁、易读、易写	适用于配置文件和数据交换	对复杂的嵌套结构支持较弱
CSV	表格形式数据格式	通用性强、纯文本、结构简单	易于在软件间传输数据	不支持复杂的数据结构
XML	可扩展标记语言	自描述性、跨平台、支持多种语言	数据结构清晰、易于人类阅读	数据冗余、解析性能较差
HTML	网页内容标记语言	丰富的标签和属性	适用于网页内容的展示和传输	不是专门的数据交换格式
Smile	二进制JSON	紧凑、快速、支持JSON特性	序列化结果小、速度快	相对于JSON可读性差
BSON	二进制JSON	支持更多数据类型、性能高	适用于MongoDB等NoSQL数据库	不是通用数据交换格式
Avro	数据序列化系统	自描述、紧凑二进制格式	跨语言、支持复杂数据结构	需要预先定义Schema

这些格式各有其特点，并适用于不同的场景。例如，YAML和CSV常用于配置文件和数据交换，XML在Web服务和配置文件中广泛使用，Smile和BSON在需要高性能和紧凑格式的场景中表现出色，而Avro则更适用于需要跨语言的数据交换和复杂数据结构的场景。在选择数据交换格式时，需要根据具体的应用场景和需求来权衡各种格式的优缺点。

3.5.4　不同场景的推荐

在笔者的开发经历中，主要使用的是JSON和Protobuf。JSON以其简洁性和高可读性在众多项目中备受青睐。无论是产品研发还是测试，团队成员都能轻松驾驭这种格式。JSON的直观性和易理解性使得数据交换变得更加直观和高效。在快速迭代和新增页面的场景中，使用JSON可以显著减少后端配置的依赖，让其他角色人员能够更加独立和自主地进行配置。这种灵活性极大地提高了开发效率，使得项目能够快速响应市场变化和用户需求。

然而，随着项目规模的扩大和团队语言的多样化，JSON在某些方面可能无法满足需求。此时，Protobuf便成为另一种理想的选择。在笔者所在的企业团队中，由于涉及Java、Golang和Python等多种编程语言，不同语言间的通信和数据交换成为一个重要的问题。而Protobuf恰好提供了一种跨语言的解决方案。基于GRPC协议进行通信，Protobuf能够方便地在不同语言间进行数据传输和交换。更为重要的是，Protobuf支持自动生成对应语言的代码，使得开发人员可以更加专注于业务逻辑的实现，而无须花费大量时间处理数据格式。这种自动化生成的特性不仅提高了开发效率，还确保了代码的一致性和可维护性。

此外，Protobuf在低代码平台上的应用也是其优势之一。在低代码平台上，开发人员可以通过简单的配置和拖曳操作来快速生成页面和业务逻辑。而Protobuf作为主要的序列化协议，可以方便地与低代码平台进行集成。开发人员可以将Protobuf数据直接导入低代码平台中，并通过简单的配置来生成对应的页面和交互逻辑。这种无缝对接的特性使得Protobuf在低代码平台上的应用更加广泛和深入。