中图分类号：TP393.09
论 文 编 号 ： 10006SY1006321

北京航空航天大學

硕 士 学 位 论 文

高性能内容整合引擎的研究与实现

作 者 姓 名　　徐 静 波

学 科 专 业　　计算机应用技术

指 导 教 师　　刘 旭 东　教 授

培 养 院 系　　计算机学院

Research and Implementation of High Performance Mashup Runtime Engine

A Dissertation Submitted for the Degree of Master

Candidate：Xu Jingbo

Supervisor：Prof. Liu Xudong

School of Computer Science and Engineering

Beihang University, Beijing, China

图分类号：TP393.09
论文编号：10006SY1006321

硕 士 学 位 论 文

高性能内容整合引擎的研究与实现

作者姓名	徐静波	申请学位级别	工学硕士
指导教师姓名	刘旭东	职　　称	教　授
学科专业	计算机应用技术	研究方向	Web 服务计算
学习时间自	年　月　日	起至	年　月　日止
论文提交日期	年　月　日	论文答辩日期	年　月　日
学位授予单位		学位授予日期	年　月　日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：_____ 日期： 年 月 日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名：_____ 日期： 年 月 日

指导教师签名：_____ 日期： 年 月 日

摘　要

随着信息技术的发展，互联网正逐渐从一系列网站的集合转变为一个成熟的、为终端用户提供网络应用的服务平台。终端用户作为互联网数据提供、分享和使用的主体，对个性化互联网应用的开发需求也越来越迫切。在这种背景下，出现了一种新的软件开发模式：Mashup。Mashup 是指将不同来源的数据或服务进行组合，从而构建出一种具有新型功能的网络应用。它既可以完成服务的快速构建，也可以满足用户自主参与进行数据处理的需求，从而迅速发展，并成为 Web 2.0 的代表技术之一。

Mashup 应用开发平台为终端用户提供了信息整合的应用开发环境。Mashup 执行引擎作为开发平台的核心部分，其性能的好坏将直接影响平台所提供的服务质量。本文首先考虑了 Mashup 结构和运行的特殊性，提出了一种 Mashup 调度单元模型：mashlet 及其相应的性能衡量指标，作为后续研究的基础。其次，本文创新性地提出了一种基于“懒启动”的动态调度策略，在 Mashup 引擎的执行过程中动态地、有序地对 Mashup 执行请求进行调度，通过节约引擎瓶颈资源的占用，提升了引擎的吞吐量。本文还针对 Mashup 平台面向终端用户带来的低效编程问题，提出了对 Mashup 应用片段的静态优化策略，通过重组织 Mashup 应用片段提升其执行效率，作为对内容整合引擎性能提升的补充方法。最后，在上述研究的基础上，本文设计并实现了一个高性能的 Mashup 应用执行引擎，在其中应用了基于懒启动的动态调度策略和静态优化方法，通过一系列实验及分析，验证了它们对于引擎性能提升的有效性及引擎整体的高效性。

关键词： 个性化，Mashup，信息整合，性能优化

Abstract

With the development of information technology, the Internet is gradually transforming from the collection of a series of websites into a platform with variety of applications and services for end-users. It is a very challenging issue for service providers to meet the requirement of end-users to compose their own programs in an easy way. Mashup is considered to be a feasible approach to address this issue. A mashup is a web application that combines the data or the functionality from several sources to build a new service. Since it can be built quickly and used easily, Mashup has developed rapidly and becomes one of the representative technologies of Web 2.0.

A Mashup development platform offers customers to build their own services by integrating some available information. Mashup runtime engine works as the core part of the platform and its performance directly affects the quality of service built based on the platform. First of all, this paper put forward a mashup model and its composition. Secondly, according to characteristics of the Mashup, this paper states a dynamic scheduling policy based on the technique of "lazy start", which could save the memory consumption of the engine and improve throughput of the platform. Last but not least, this paper proposes a set of static optimization rules to adjust the structure of Mashups, to make Mashups can be executed efficiently.

With this research, a Mashup execution engine with high performance is developed, which implements the dynamics scheduling policy and the static optimization. A set of experiments have conducted and proved the effectiveness and efficiency of these methods.

Keywords: Personalization, Mashup, Data Integration, Performance optimization

目 录

图　目

表　目

第一章　绪论

1.1 研究背景与意义

随着信息技术的发展，以网络技术为代表的新兴计算技术给全球经济带来了巨大影响，信息系统已渗透到企业生产的各个领域和阶段，成为企业竞争力的重要体现。然而，在信息化建设的历程中，诸多异构系统之间的数据源因为拥有各自独立的数据格式、元数据以及元模型，造成了信息孤岛大量存在的事实，让信息化建设和发展面临着巨大的挑战。Gartner 于 1996 年提出 SOA（Service Oriented Architecture，面向服务的体系结构）的理念，为信息化产业带来新的解决方案[1]。SOA 是一种组件模型，其核心思想是将企业信息系统组件封装并发布成为可通过互联网访问的标准的 Web 服务，并针对新的应用需求或环境变化快速组合这些服务形成新的应用软件，从而有效降低软件的开发成本，提高网络软件的生产效率。面向服务的体系结构带来了一场信息化的变革。特别是 2000 年 Web Service 出现后，SOA 的松散耦合、数据交互与协同、信息集成等高效性特点被实践所验证。随着 Web Service 的流行，SOA 被誉为下一代 Web 服务的基础框架，目前已经成为计算机信息领域的一个新的发展方向。

另一方面，随着互联网技术的发展，互联网已经从早先的一系列网站，慢慢转变为一个成熟的、由终端用户主导内容、同时也为终端用户提供网络应用的服务平台，并以此标志着 Web 2.0 时代的到来[2]。Web 2.0 既是一次以技术创新为基础的应用，也是一种以应用为导向的技术创新，在信息主体、传播方式、组织方式等方面都产生了一系列突破和变化。它以用户为中心，以自组织为主要形式，以构建参与、互动、分享、真实化的网络平台或节点为手段。

在 Web 2.0 时代，网络成为新的平台，内容因为每位终端用户的参与而产生，这些带着明显个性化的内容通过用户间的分享，构建了整个 Web 2.0 的网络[3]。由此可见，每一个用户可能都是一个潜在的消费者。他们一方面创造数据，一方面也使用互联网上的各种数据和服务。如何将 SOA 的系统构架用于解决这些用户的数据整合、使用以及构建小型业务的需求一度成为互联网技术的研究课题[4]。源于语义 Web 领域的数据建模技术和松耦合、面向服务、与平台无关的通信协议相结合，将可以提供一种可充分利用大量 Web 信息的应用程序基础设施[5]。内容整合系统 Mashup 正是在这样的背景下提出的。

Mashup 是指将不同来源的数据或服务进行组合，从而构建出一种具有新型功能的网络应用[6]。它既可以完成服务的快速构建，也可以满足用户自主参与进行数据处理的需求，从而迅速发展，并成为 Web 2.0 的代表技术之一。

Mashup 迅速发展是长尾理论在软件领域的一次成功应用：它关注着软件开发领域的“长尾群体”，即那些有开发应用需求的众多终端用户[7]。这些用户普遍缺乏编程经验，也无法消费应用开发的巨大时间、精力、金钱成本。Mashup 将复杂的技术逻辑隐藏，只为终端用户提供一些必要的界面和选项，简单的建模很好地迎合了终端用户的需求。在这种方式下，可能以极低的成本关注了分布曲线的“长尾”，产生巨大的总体收益，发挥“长尾”效益的价值。

Mashup 迅速发展的另一个原因是它在一定程度上满足了企业级用户的需求。在企业内部，常常面临着越来越频繁的组织结构与业务流程变化，信息技术支持需要随时调整以适应业务或结构变更所带来的影响。与此同时，由于市场变化加快，业务协作时间缩短，使用传统的应用开发模式消耗过多的时间在应用系统集成工作，减少了获得投资回报的时间与收益。为此，需要寻求能够快速构建应用且成本合理的解决方案，Mashup 方式构建的信息整合应用以开发周期短，开发门槛低等特点满足了企业对这类应用的需求。一般的企业级应用通常用来解决一系列复杂的问题，功能较多，用户群体范围广，通用性强，但开发需要较长的周期，详细的项目计划，较多的人力资源。Mashup 应用则能以一种灵活小巧的方式满足某个特定的需求。同时具有开发时间短，所需人员少，针对性强等优点。

Mashup 涵盖了众多理论、技术与方法，并大量运用 Web2.0 技术。在资源共享方面，它通过使用内容提供商的开放 API、RSS/Atom 聚合数据、网页内容以及 XML 数据等方式获得数据源；在用户参与、协作方面，它使用一些已有的成熟技术，如 Web Service、REST、JSR168 等技术，在聚集与复用方面，它表现出明显的 Web 2.0 特征，如标签标记、用户分享等；在用户体验方面，Mashup 从客户端角度出发，关注软件的交互性与用户体验，通常结合 AJAX 和 RIA 技术将应用呈现给用户。

综上，Mashup 是 Web 2.0 发展过程中的一个必然产物。互联网用户日益迫切和深化的个性化服务需求是催生 Mashup 的内在动因，互联网相关技术的进步和原有技术的充分挖掘与组合，是 Mashup 蓬勃发展的外在条件。对 Mashup 平台及其相关技术的研究，有助于我们认识互联网的典型特征和经验，也有助于对互联网技术的进步趋势做出展望。这也是本论文研究 Mashup 的意义所在。

而在 Mashup 的各种应用模式中，Mashup 引擎总是最核心的一部分。Mashup 引擎一方面面向用户，屏蔽所有技术细节，透明地为用户完成数据和服务的整合；另一方面，又作为底层支撑，负责实现所有的技术细节。Mashup 引擎的性能高低直接关系着服务质量的好坏。Mashup 引擎的高性能，体现在充分利用有限服务资源，提供尽可能多的 Mashup 执行服务，具有较高的吞吐率，能应答较高的请求并发量。在一些已有 Mashup 平台上，访问数据统计也显示出对 Mashup 引擎的性能有着现实的需求。如据不完全统计，在 Yahoo! Pipes[8]平台上有近 90000 名活跃的终端用户，每天接受近 5000000 的 Mashup 执行请求。由此可见，对 Mashup 引擎性能的研究具有十分重要的实践意义。对 Mashup 引擎性能的研究，有助于我们深入分析 Mashup 应用的运行机制，了解引擎的核心原理；探索引擎性能的有效提升方法，有助于为已有的 Mashup 应用平台提供借鉴和反馈，并有可能带来直接的经济效益。

1.2 国内外研究现状

Mashup 概念最早是一个用于音乐中“糅合”的术语。在计算机领域中 Mashup 被赋予了新的含义。Mashup 从提出到业界的普遍接受和理解，再到后来的广泛存在，这个发展过程与工业界、学术界对它的推广与研究是分割不开的。本小节将简述国内外对 Mashup 的研究现状，具体地将从 Web 内容整合模式的探索、产业界对 Mashup 的应用、学术界对 Mashup 平台性能研究三个方面展开论述。

1.2.1 Web内容整合模式研究

Mashup 是数据、逻辑和表现的联合，它是基于 SOA 架构在应用层的一种衍生[9]。相对于 SOA 架构中服务组合关注服务间功能的结合以实现用户需求，Mashup 更加关注对数据的操作。和现有的 SOA 服务组合技术相比，Mashup 应用粒度更大，开发方式更简单，灵活性更强。

目前，大多数 Mashup 应用开发的描述语言参考工作流技术。工作流技术是指一类能够完全自动执行的经营过程，根据一系列过程规则，将文档、信息或任务在不同的执行者之间进行传递与执行。工作流领域受到业界广泛支持的标准是 2002 年 8 月由 IBM 和微软联合发布的 BPEL4WS（Business Process Execution Language for Web Services），该规范被结构化信息标准促进组织（OASIS）接受后，在 2004 年 9 月进一步发展为 WS-BPEL2.0 规范。WS-BPEL 是一种定义可执行抽象业务过程的语言，它

通过拓展 Web Service 的交互模型来支持业务的事务操作。在 Mashup 应用的开发过程中，主要进行的操作是数据间信息的提取、传递与处理，使用企业级的业务流程执行语言来描述 Mashup 应用逻辑信息显得过于复杂，因此需要在借鉴工作流描述的基础上进行简化。Xproc[10]提出了一种面向数据流的设计模型，该模型拓展了 XML，用一组对接的输入输出表示一种类似管道的关系，适用于表示一个序列的操作和处理。Mashroom[11]提出一种基于表格的 Mashup 描述语言，将 Mashup 流程关系映射为二元组，分别表示流程的数据和操作，通过表格中顶点与边的关系来描述数据整合流程的逻辑。MashMaker[12]则提出了一个全面而复杂的描述模型，这个模型在设计时参考了文件系统、表格、映射集、浏览器等多种经典数据模型，最终基于树与目录结构，经一系列形式化演绎设计了一种语言，用以描述 Mashup 的过程。

在应用模式方面，学术界也有诸多讨论。文献[13]提出一种 Mashup 应用框架，它是现有 SOA 架构的扩展，用于实现快速的服务组合。对应于 SOA 中的服务提供者、服务代理和服务使用者，这种框架下将 Mashup 的参与者划分为 Mashup 组件生成器、Mashup 服务器和 Mashup 使用者三个角色与之对应。Mashup 组件生成器负责从服务目录中获取服务，将它们处理输出为标准的容器模型，并注册在资源库中。Mashup 服务器负责存储、发布组件生成器注册的组件，并维护一系列 Mashup 组件的使用状态。Mashup 使用者选择组件用以整合成为 Mashup 应用。将 Mashup 应用到时下热门的移动服务领域将带来一系列新的研究问题。文献[14]指出，由于包含互联网和数据服务商网络两部分来源，移动 Mashup 的数据源相对于 Web 应用层级的 Mashup 将更加丰富，但由此也将在安全性和构架设计方面迎来更大的挑战。文献[15]利用 SOA 的灵活性，设计了一种基于 SOA 的移动 Mashup 平台，探索运营级的移动 Mashup 系统构架。该构架分为接入层、控制层、应用层和适配层四个部分，并由一个服务管理框架从可用性、健壮性以及对业务的动态适配能力等几个方面来保证整个平台的高效运行。

1.2.2 内容整合的产业化发展

Mashup 在以用户为中心的 Web 2.0 时代迅速兴起并深入人心。当今的互联网上，Mashup 技术的应用随处可见。如数据信息的位置标注与地图服务的 Mashup，让各种不同的数据集在地图上以图形化的方式呈现；图片服务如 Flickr 与社交网络的 Mashup，可以识别照片中的朋友并分析一个人的社交图谱；搜索和购物的 Mashup，消费者可以

通过比较来自各个商场的零售价数据而选择一个最经济的方案；还有新闻的 Mashup，创建个性化的报纸，从而满足读者独特的阅读兴趣等。据 ProgrammableWeb 的不完全统计，网络上有约 7800 组开放数据 API，每一组 API 都有惊人的活跃度和使用量，Mashup 的热度可见一斑[16]。

Mashup 的概念被提出后，雅虎、谷歌、微软、英特尔、IBM 等各大互联网公司也纷纷推出自己的平台或产品挖掘它的商业价值[17]。

2007 年 2 月雅虎公司推出第一款成型的 Mashup 商业应用平台 Yahoo! Pipes[8]。Yahoo! Pipes 提供基于 Web 页面的模块拖拽式交互引导用户制作 Mashup。Yahoo pipe 将对数据源的一系列处理称之为“管道”，每一个管道处理一组特殊的任务。Yahoo! Pipes 致力于满足终端用户的业务处理设计与运行。具有一定编程基础的用户可以在 Yahoo! Pipes 上编辑一个包含获取数据、处理数据、展示数据等模块的管道，再交由 Yahoo! Pipes 引擎运行以完成信息融合的操作。2011 年 8 月，雅虎公司在原有运行引擎的基础上做了商业上的优化与升级，推出了引擎的 V2 版本。图 1 是 Yahoo! Pipes 平台上创建的一个 Mashup 应用实例。

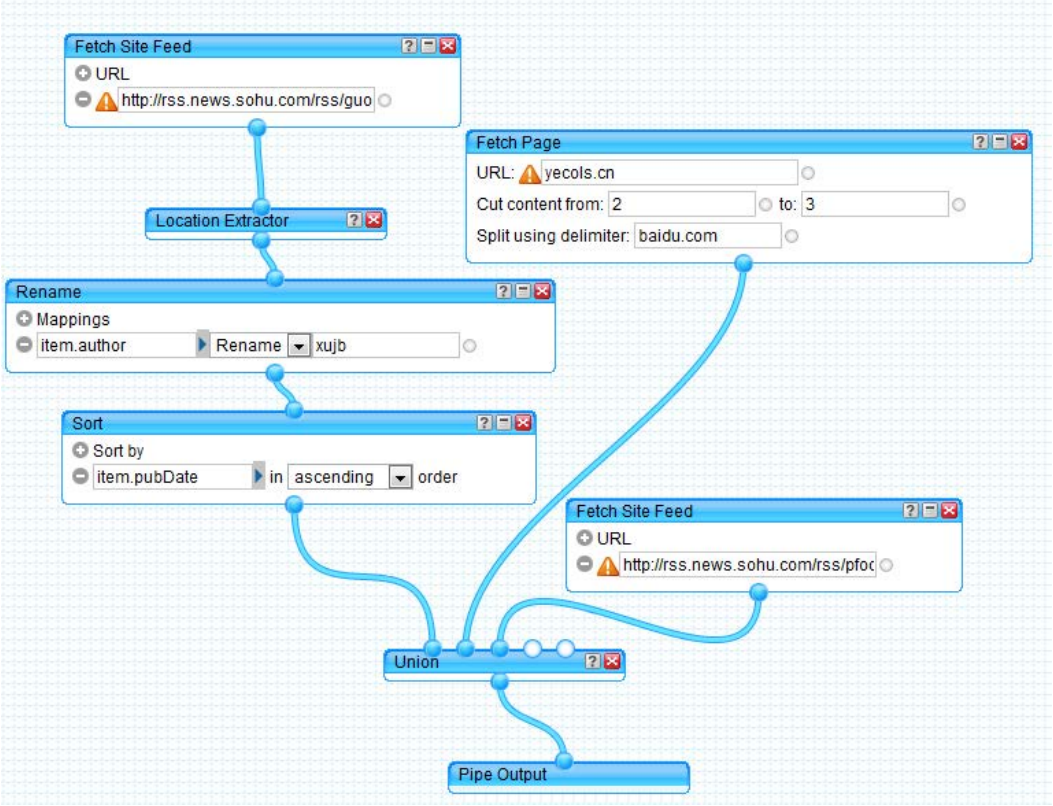

图 1　Yahoo! Pipes 创建的 Mashup 应用示例

Google 公司的 Google Mashup Maker 则提供一种基于模版的 Mashup 开发环境。

它提供了一系列的标准模块用于填充外部数据，例如，List 模块用于表示 RSS 或 Atom 等条目列表，Item 模块用于表示单项数据项目。当用户创建 Mashup 时，需要创建一个展现界面、一系列标准组件的控制逻辑以及配置数据源，GME 在运行时，从数据源获取数据并按照逻辑操作，将最终结果填充到展现界面并显示给用户。2009 年，Google 公司进行业务调整，一方面以 API 的形式将各种服务数据向开发者开放，另一方面，发布了 Google App Engine 作为开发者应用的托管引擎。这实际上是将 Mashup 的编辑和运行两个阶段服务分开，同时给了开发者更大的开发自由度。Google 随即关停了 Google Mashup Maker。

微软公司则提供了一种基于组件的 Mashup 可视化开发环境，Microsoft Popfly。Popfly 基于微软自有技术 Silverlight 构建，通过浏览器为用户提供一种华丽的浏览效果和操作体验。在 Popfly 的环境下，可重用的组件被称之为“blocks”，每一个 block 其实相当于外部服务的中间件，包含着一组输入和一组输出。它们或连接了 Web Service 的调用，或被作为一个实用函数完成某项功能处理，或直接负责结果的界面展示。每一个 Mashup 即由一组 blocks 组成。

Mash Maker 是英特尔公司的 Mashup 产品。Intel Mash Maker 脱胎于英特尔研究院的一个内部项目，它致力于将 Mashup 向终端用户更加推进一步。不同于其他 Mashup 平台需要专门的工具和环境，Mash Maker 以浏览器插件的方式提供交互[19]。在安装该插件后，任何用户都可以对当前浏览的页面进行数据抓取和信息提取。提取页面信息时，Mash Maker 首先尝试依据 RDF 描述抽取结构化数据；当网站不提供 RDF 信息时，它依据一个由社区维护的网页类型描述数据库，从 HTML 网页的原始数据中抽取有用信息，从而保证抽取到的网页信息尽可能完整和有效。用户可以对抽取到的页面信息进行保存或者二次处理。该插件还允许用户组合 Web 页面，从而提供个性化的、整合式的网站浏览方式。图 2 为 Mash Maker 的运行效果，其中左侧为 Mashup 辅助面板。

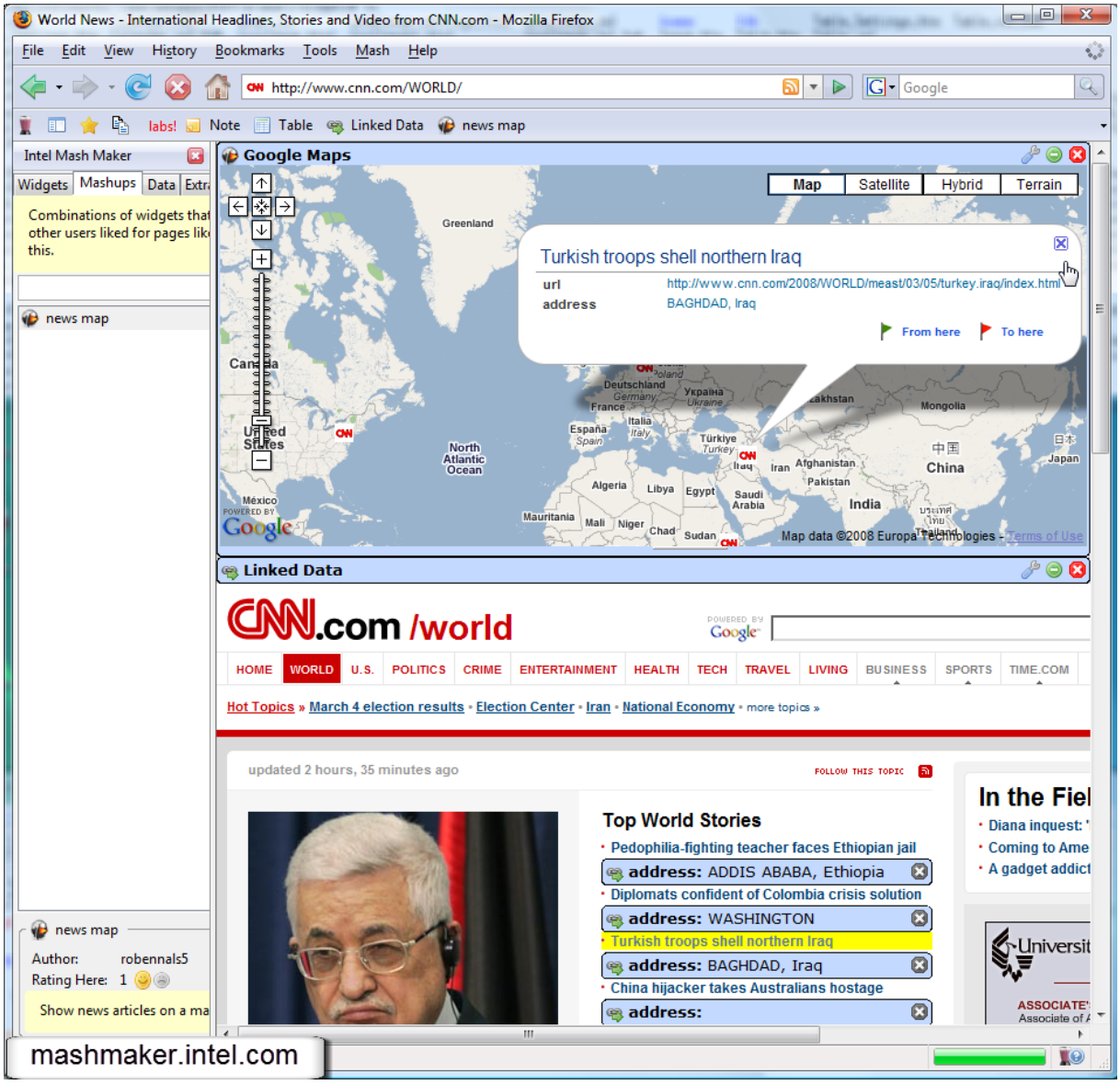

图 2　Intel Mash Maker 运行效果图

作为解决方案提供商，IBM 也推出企业级的 Mashup 应用产品：IBM Mashup Center。IBM Mashup Center 致力于为企业内部提供信息整合服务。业务人员可使用 IBM Mashup Center 自由组装动态的情景应用，通过灵活支配企业内部各部门之间的信息资产，应对易变的业务需求，以更加便捷与灵活的方式创造新的业务价值。

综上，Mashup 在互联网具有十分广泛的应用，多数用户在获取信息的过程中，直接或间接地使用了 Mashup 技术。以各大互联网公司的产品为代表的一系列 Mashup 平台或工具，为终端用户提供了直接参与 Mashup 创建和使用的机会，也说明了 Mashup 技术在产业界被普遍接受，正在蓬勃发展。

1.2.3 内容整合平台的性能研究

Mashup 引擎处于平台的整合层。作为一个 Mashup 平台的核心部分，引擎的设计和实现直接影响平台性能。学术界对 Mashup 的引擎性能有一些研究，传统软件领域和中间件的技术为 Mashup 平台的性能优化提供了借鉴方案。

通过结合传统缓存与引擎执行的耗时分析，有学者提出一种动态的 Mashup 缓存框架：Mace[20]。在 Mace 的设计中，作者使用缓存 Mashup 片段的中间结果数据来进行性能的提升。相比于传统的缓存应用情景，Mashup 有一些独有的特点。由于 Mashup 通过一系列的操作来融汇数据，故 Mashup 应用程序可以看作是一种倒过来的树形结构。其中，叶子节点即为输入数据源，根节点为输出。内节点表示一系列操作。作者创新性地提出，在 Mashup 中应该以内节点的子树作为缓存对象，并基于树形结构的性质提出，在一个 Mashup 中，选择一个节点作为缓存点并缓存其子树，对于一个并发运行 Mashup 的平台来说是最高效的。在缓存点的选择上，作者在考虑了节点的输入数据规模、输出数据规模以及耗时后，提出了一种评估模型，并进一步抽象为一个可用贪心求解的算法。从而在极短的时间内可以最优选择一个缓存点进行缓存。实验证明，该动态缓存框架在 Mashup 并发量增长以及数据源更新频繁时，较之不用缓存与静态缓存的方法可以显著提升平台运行引擎的性能。

也有一些观点提出，应该从 Mashup 引擎的实现上采取更优质的策略。考虑到 Mashup 运行时的时序性，在引擎中使用事件驱动的方式或许是一种好的方法[21]。在这种实现中，Mashup 的获取数据源、处理数据等模块都被看作事件源与引擎的一个事件控制中心进行通信。一个 Mashup 在运行时，向事件控制中心注册一个实例，并在其控制下开始运行其中无依赖的模块。当这些模块运行完成后，向事件控制中心发送一个通知，后者接受通知后，更新相应的状态并调动后序模块运行。通过这种方式，对一个 Mashup 进行了拆分，并通过线程等手段，使得模块的运行与其所属的 Mashup 运行独立开来，从而达到模块流水线式复用的目的，从构架的角度提升了引擎性能。

Mashup 的面向对象是较为普通的用户，这个群体没有较为系统的编程经验。基于这点假设，可以预见用户创建或编辑的 Mashup 是效率低下的[22]。例如用户可能通过简单的拖拽等方式编写 Mashup，由于误操作等原因其中有一些模块处于失效的状态，或者因为其能力所限，只是通过直观逻辑来放置和编排模块的先后顺序。这样的个性化导致一个 Mashup 直接交付引擎执行是效率低下的。有一些观点提出，借鉴软件工程中的系统方法，可以对其 Mashup 片段本身进行优化。实验中，通过对 Yahoo! Pipes

上的 Mashup 进行抓取和分析，证明该方法在一定程度上是可行的，通过优化后的执行片段能高效使用引擎资源。

还有一种观点认为，在 Mashup 引擎的实现中，应该引入重用的机制[23]。在 Mashup 中主要的创建、编辑方式就是对数据和操作模块的重组，而一个平台提供的模块和数据源一般是有限的。在这样的情况下，它们的组合也会分布在一个有限集中，即用户创建和编辑的 Mashup 会有一定程度的重复性。如果两个 Mashup 中的某块片段，操作模块与输入参数相同，即可认为这两个片段是相同的。当引擎的负载达到一定规模时，来自不同 Mashup 的相同执行片段会在同一个时间片内重复执行，如果引擎能把这种重复的执行片段分拣出来并合并它们的执行，将能削减总共执行次数，从而在平台的整体上达到更好的执行效率。

综上，学术界对 Mashup 引擎的性能方面有所关注，也从各种角度提出了提升性能的方法。这些方法对本论文的工作具有一定的借鉴意义。

1.3 研究目标和内容

1.3.1 课题来源

本课题来源于北京航空航天大学可信网络计算技术国防重点学科实验室承担的国家 863 高科技发展计划课题“大规模服务化软件的组合开发和运行演化技术及平台”（项目编号：SS2012AA010103）。课题旨在提供以可信软件生产环境为基础的大型服务化生产和运行平台，构建大型网构化软件的协同构造、运行管理、可信评估和持续演化的技术体系与公共服务环境。其中，一子课题旨在研究按需服务化软件建模与组合的技术与方法。

1.3.2 目标与内容

本论文的主要研究目标是通过分析已有平台的性能效率，研究一种Mashup平台引擎运行的综合策略，并将之应用，实现一个高性能的即时信息整合平台，该平台引擎拥有较高的吞吐率和较低的用户平均等待时间。

本文主要的研究重点将集中在三个方面，分别是Mashup引擎动态调度策略的研究，Mashup应用静态优化策略的研究，以及综合运用该两种策略的引擎设计与实现。

本论文的第一个研究工作是Mashup引擎动态调度策略的研究。本文将从平台的资源有限性讨论出发，分析Mashup应用运行和结构的特点，找出平台引擎在执行Mashup应

用过程中的性能瓶颈。并以此性能瓶颈为突破，提出一种调度策略，使得引擎可以合理、有序地分配、使用平台资源，并特别照顾瓶颈资源，从而提升引擎的并发处理能力，提升平台的Mashup应用执行吞吐率。

本论文的第二个研究工作是Mashup应用的静态优化。本文将通过在运行前尽可能的优化结构，规避资源使用过程中的冗余和浪费，让Mashup以一种高效率的组织和结构发出执行请求。Mashup应用的静态优化策略作为引擎动态调度策略的补充，可以进一步提升Mashup平台的整体性能。

本论文的第三个研究工作是系统实现。基于以上两个研究工作，本文将设计和实现一个高性能的Mashup执行引擎。Mashup执行引擎是一个Mashup平台的核心部分，平台一般面向普通的终端用户，通过提供一个图形化的交互界面，让终端用户根据个性化的业务需求自由、简单地建模。引擎负责将用户的建模结果运行起来，通过一系列Web资源的调用和内部的处理，输出为用户想要的结果。这个Mashup的运行过程涉及引擎对模型文件的解析、处理，引擎和外部资源的交互、对内部数据的处理等技术和方法。本文将在这些技术方法的研究上，在考虑满足基本功能性需求的同时，应用上述的动态调度策略和静态优化方法，设计并实现Mashup的执行引擎。

1.4 本文的组织结构

本文的组织结构如下：

第一章 绪论

本章主要阐明本文的研究动机。首先介绍 Mashup 技术的起源和发展，分析现阶段 Web 内容整合模式的探索、Mashup 产品的实践和产业化发展以及学术界对 Mashup 性能的研究。最后介绍本论文的研究目标和组织结构。

第二章 内容整合引擎的关键技术分析

本章主要介绍内容整合引擎所涉及的关键技术。首先给出一个 Mashup 的一般模型，介绍经典的树形 Mashup 表示法。然后介绍 Mashup 平台的经典组成，从功能上将其划分为三个层次：编辑层、表现层和运行层。之后对三层的划分分别介绍关键技术以及本项目中选用的技术方案。最后详细介绍 Mashup 运行引擎的外部资源获取的几种方式和内部数据处理时的几个关键技术点。

第三章 内容整合引擎的高性能动态调度策略研究

本章主要研究一种基于“懒启动”的高性能内容整合引擎动态调度策略。首先从

平台的资源有限性出发，基于 Mashup 应用的结构和运行特点，分析了 Mashup 应用执行时的性能瓶颈。接着提出一种同时将运行时间和占用空间考虑在内的衡量方法，并基于此提出动态调度策略的设计目标。然后，根据此目标，提出一种“懒启动”的调度机制，并根据实际运行的情况，提出两种修正。最后，对基于“懒启动”调度策略的效果进行了理论分析。

第四章 内容整合引擎对执行对象的静态优化研究

本章主要研究内容整合引擎对 Mashup 应用的静态优化方法。首先对静态优化 Mashup 应用进行可行性分析。然后对具体的静态优化策略开始设计，依次从设计思路、优化的规则和处理的时机进行论述。最后，对静态优化效果也做出理论评估。

第五章 高性能内容整合引擎的设计与实现

本章主要论述高性能内容整合引擎的设计与实现。首先给出系统的总体设计，这个设计中除了包含第二章中所述的三层结构外，还对运行层进行详细的模块划分，包含了第三章、第四章中描述的性能提升工具和方法。然后对运行层的几个主要模块的设计和实现进行详细描述。最后，通过一些系统截图，展示内容整合引擎的系统实现和运行效果。

第六章 实验与评价

本章主要通过一系列实验，测试平台在功能上的高可用性和可依赖性。同时通过一些对比实验，验证动态调度策略和静态优化对于高性能内容整合引擎实现的效果和意义。

总结与展望

总结论文的主要工作，并展望进一步工作。

第二章 内容整合引擎的关键技术分析

本章主要对内容整合引擎所涉及的关键技术进行介绍。本章首先给出内容整合引擎的工作对象，即 Mashup 的模型、结构、及其表示方法，其次介绍论文依托项目的平台构成及每一组成部分中的核心技术，重点对运行引擎部分的外部资源获取和内部数据处理过程进行技术分析。

2.1 Mashup模型

普遍被接受的观点认为，Mashup 应用是指这样一类应用：这种应用以获取网络上的数据源为起点，将获得的数据经由一系列操作处理，最终将结果反馈给用户。这里的操作处理是指在 Mashup 平台上预先定义和支持的功能模块，在本论文的讨论情景下，称之为“操作子”。每种操作子的任务被设计为完成一种特定的功能。如 Fetch 表示从某个地址请求网页、数据的操作；Sort 表示以某种规则对数据进行排序的操作；Filter 表示对数据进行筛选的操作；GeoTag 表示对数据进行地理信息标记的操作；Merge 表示合并数据的操作；Cut 表示对数据进行截取的操作；End 表示一个 Mashup 应用的处理终点，此时 Mashup 的结果应被返回给用户。

如图 3，表示了一个 Mashup 应用示例。其中左图来自用户的自然需求，某用户在 Mashup 平台上创建了该 Mashup 应用，主要功能是通过 Fetch 操作子从互联网上获取一些新闻 RSS 数据，并按需做一些排序、截取、合并、提取地理信息等处理。右图是对左图的一个操作映射和结构规整。

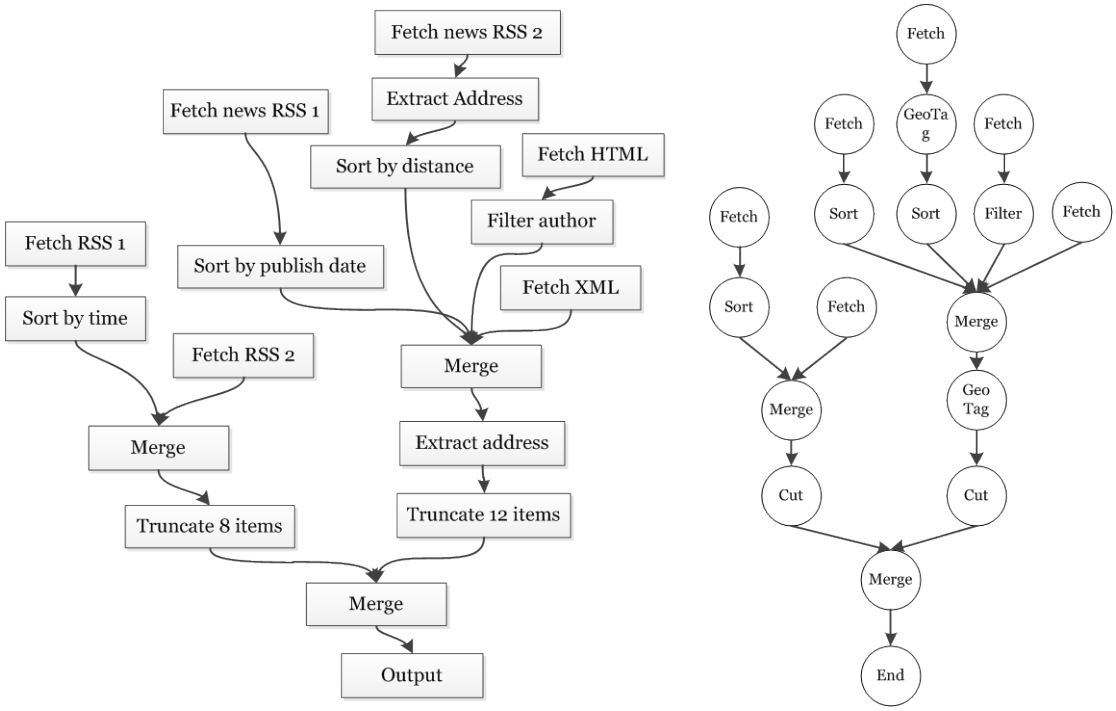

图 3　Mashup 应用示例及其树形结构

从图 3 的 Mashup 应用示例中不难发现，Mashup 应用的数据结构具有一般规律：一个 Mashup 应用可以表示为一个拓展的、倒着的树型结构。在这棵树中，每一个节点表示一个操作子，运行逻辑从叶子节点开始，到根节点结束。其中，有一些特殊位置的节点，它们只能是特定的操作。如叶子节点总是 Fetch 操作子，以从互联网上获取数据源；根节点总是一个 End 操作子，作为该 Mashup 应用处理的终点；分支岔口的节点总是一个 Merge 操作子，表示将数据流的合并。在本论文的情景下，将 Mashup 树称作对树结构的拓展是因为相对于普通的树，Mashup 树中还有一些不处在分支处的节点，如示例中的 GeoTag 操作子，它们表示对数据流的其他操作。

Mashup 应用可以表示为表达式 Mp = {N, E}，其中 N 表示用户所选择的操作子的集合，E 表示在 Mashup 树中连接各操作子的边的集合，用于表示所连接的各操作子的输入和输出关系。

一个 Mashup 应用由叶子节点获取外部数据源，将从外部数据源获取到的数据作为原始数据。其中，叶子节点即为入度为零的节点，入度表示终止于该节点的边的数量，因此，入度为零表示该节点上没有连接终止于其上的边，是整个树形结构的起始点。叶子节点将获取到的原始数据发送给相应的操作子进行处理，数据经由树枝上的各操作子的节点处理之后，最终达到树根的节点，并将此时的运行结果输出。

对于树形结构中对数据流进行汇聚的 Merge 节点来说，需要等待各输入数据流均

准备完毕之后，才能进行运行。也就是说，在 Merge 节点所连接的各节点分别完成对数据流的处理之后，该 Merge 节点才能够对输入的两个或多个数据流进行处理。

2.2 Mashup平台的构成

Mashup 平台作为一个完整的系统，可以从功能上划分为三个层次[24]。

首先，Mashup 平台必须提供一种面向终端用户的建模方式，作为用户使用 Mashup 平台功能的起点，即 Mashup 的编辑层。其次，Mashup 平台必须提供一种向用户展现应用处理结果的方式，即 Mashup 的展现层。当前，主流的展现层通过浏览器实现，而内容整合应用的特点分析表明，很多 Mashup 应用对实时性和地域性有较高的要求，用户也希望可以通过手机随时地访问之前编排的 Mashup 应用。这种需求可以通过移动互联网和移动设备应用在展现层而得到满足。最后，Mashup 平台还必须有一个核心的运行层，负责 Mashup 应用的运行。

在本论文的依托项目中，平台构成如图 4 所示。平台根据上述功能层次，划分为 Mashup 编辑层、Mashup 表现层和 Mashup 运行层。各层次的设计及技术要点将在以下各小节中阐述。

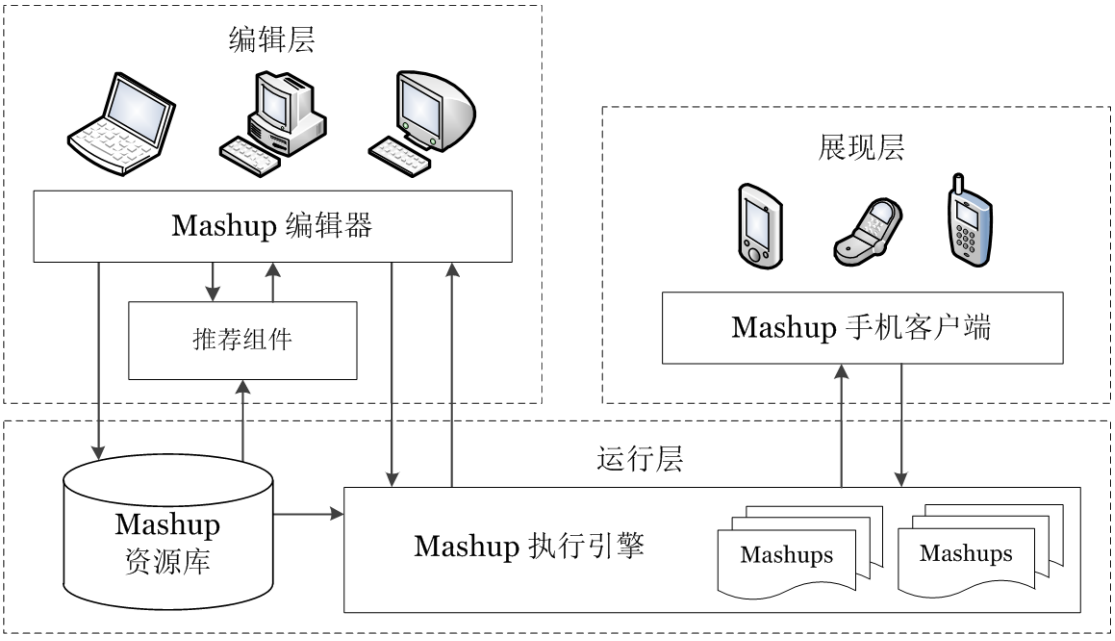

图 4 Mashup 平台构成及层次划分

2.2.1 Mashup编辑层

Mashup 平台必须提供一种面向终端用户的建模方式，作为用户使用 Mashup 平台

功能的起点，即 Mashup 的编辑层。在这个层次，平台主要满足用户创建、编辑、保存 Mashup 应用的功能。

在本论文的依托项目中，Mashup 编辑层的具体载体是一个以 Flex 实现的 Web 端编辑器[25]。在该编辑器中，提供了预定义的功能模块（如获取数据、排序、截取等操作）和零配置的数据模块（如一些新闻网站的 RSS）。用户在创建 Mashup 应用时，只需通过简单的拖曳操作，选择所需的数据或功能，并以连接线连接即可表示各模块的处理关系。

在该 Mashup 编辑器的设计中，引入了编辑期间的推荐机制[26]。这种推荐机制可以在用户创建 Mashup 应用的每一步过程中提供强交互的推荐方案。由于要针对用户创建 Mashup 的每一步做出响应提供推荐方案，故推荐算法的实时性要求很高。该机制的实现通过三个组件：数据分析器、数据加载器、和推荐引擎。

数据分析器的目的在于离线地分析 Mashup 平台中的所有 Mashup 应用，找出一些重复率高的可重用片段。并在这种片段中建立特殊的关联性，以作为推荐的基础。这种算法依据的是用户的群体智慧：一些操作总是以相同的前后时序被用户选择并组装，说明他们必然存在一定的语义相关性。分析器在通过大量的分析计算之后，针对 Mashup 平台当前状态，将生成一张带权重的有向图，称之为操作子关联权重图。操作子关联权重图以平台上备选的操作子为节点，指向相关的操作子，边上的权重表示两个操作子间的关联度。

数据加载器加载分析器算出的操作子关联权重图，实现了一系列实时统计机制和参数记录。

以前两步为基础，平台的推荐引擎在编辑器工作的全过程中起到辅助作用。推荐引擎根据用户当前编辑的 Mashup 应用片段，在操作子关联权重图中寻找用户最可能选择的下一步操作，返回候选列表。引擎可能推荐下一个操作子，也可能直接推荐一个 Mashup 片段，这都基于用户当前编辑状态。

实践证明，该 Mashup 推荐机制在准确性、实用度、多样性等多种指标上都取得了较好效果。进一步满足了本论文依托项目面向终端用户的建模需求。

此外，该 Mashup 编辑层的设计遵循了 MVC 模式（Model-View-Controller，模型-视图-控制器模式）。MVC 模式将业务模型、业务逻辑和输出展现互相分离。其中，模型层表示业务流程或数据的处理，视图层表示用户的交互界面，控制层从用户处接受请求，从视图层接口中获取用户的交互映射为业务的处理逻辑，并通过模型操作接口

完成操作[27]。

该 Mashup 编辑器在 Flex 的实现中使用了 PureMVC 框架[28]，实现了用户操作、模型数据和控制操作的分离。用户完成 Mashup 应用的创建和编辑后，编辑器将生成两个文件，一个用于编辑器复现该 Mashup 应用界面，一个记录了该 Mashup 应用的相关逻辑，可供交付执行引擎操作。

2.2.2 Mashup表现层

在传统的系统集成方案中，软件的表现层技术通常采用图形化的用户界面 GUI，Mashup 则专注于流畅、友好的用户体验。在 Mashup 平台中，表现层用于展现整合结果，现有的实现方式通常基于 Web，使用 HTML、AJAX 等技术构建网页。在这种方式下，Mashup 平台的表现层在自动化方面还是有一定问题，包括组件的事件响应和绑定机制等[29]。

另一方面，移动互联网当下正在飞速发展。传统互联网的份额不断被移动互联网蚕食，并迫使传统网站向移动设备兼容。Mashup 面向终端用户，致力于解决用户一些简单的业务和数据整合的需求，智能手机无疑是一种便携的表现媒介。一些研究开始探索将 Mashup 表现层拓展到移动设备上来，以迎合用户的需求[30]。

在本论文的实际项目中，表现层被设计为一款手机的应用程序。手机客户端与 Mashup 平台的交互包含以下三个部分：客户端通过与用户信息服务的交互完成用户登录和身份认证；通过与 Mashup 应用目录服务的交互，获取该用户创建过的 Mashup 应用目录；通过与执行引擎的交互获取 Mashup 应用的执行结果。

手机客户端与 Mashup 平台的交互通过 JSON 的数据格式进行。JSON 是一种轻量级的数据交换格式，它基于 JavaScript 的一个子集，采用完全独立与语言的文本格式。JSON 的建构只有两种，要么是名值对的集合，要么是值的有序列表。由于它在编写和阅读上的简易性以及对机器的友好性，JSON 是一种理想的数据交换语言。

手机客户端的实现，将用户交互的事件响应从 Mashup 的平台上抽取出来，依赖于手机 SDK。而手机平台如 Android 对用户交互事件的响应与处理封装地十分完善。从而间接地降低了表现层的处理逻辑。另外，手机端较之浏览器能更方便的调用位置信息服务和地图服务。在某些依赖位置信息的 Mashup 应用场景，手机端的表现层将凸显优势。

2.2.3 Mashup运行层

Mashup 的运行层主要负责 Mashup 应用的获取数据源、逻辑处理、内部数据流转等操作。是 Mashup 平台三个层次中最核心、最复杂的部分。也是本论文的研究对象。

Mashup 运行层要负责解析编辑层交付的执行文件。运行层需要从序列化的执行请求中逆向生成 Mashup 应用，并识别、处理该应用的功能逻辑。在本论文依托项目的情景下，面向终端用户的建模需要运行层在用户创建、编辑 Mashup 应用的阶段即反馈实时的、局部的 Mashup 结果。这对运行层提出了更高的挑战。

Mashup 运行层需要负责获取外部数据源。编辑层在引导创建 Mashup 应用后，经过用户配置的 Fetch 模块实际上由运行层执行。运行层需要实现一系列的获取数据源操作，并向用户隐藏技术细节。

Mashup 运行层需要负责处理内部数据的流转。Mashup 在引擎内部实际以数据流的形式执行，引擎负责维护、管理数据流在各操作子之间的流转。面对从外部网络获取的大量异构的资源与数据，Mashup 引擎必须协商一种内部数据结构，并负责将异构的外部数据向其映射，作为内部流转时的处理对象。

Mashup 运行层还需负责与表现层手机的交互。运行层以约定格式向手机客户端发送 Mashup 的运行结果，并负责运行结果实时性和动态性的实现。

Mashup 运行层所涉及的核心技术可分为两个方面：外部资源的获取技术和内部数据的处理技术，本论文将在后续两个小节中分别阐述。

2.3 Mashup运行引擎的外部资源获取技术

终端用户对 Mashup 应用的需求主要为数据的处理以及二次利用，故而数据源是 Mashup 生存的起点。一个 Mashup 运行引擎必须对数据源及其获取有所实现和封装，并以一种简单直白的方式提供给终端用户。外部资源的获取也是 Mashup 运行引擎与外部网络资源的主要交互，外部信息的获取技术主要可归纳为以下三种方式。

2.3.1 Web Feed方式

Web Feed 是一种轻量级、多用途、可扩展的元数据描述方式。代表技术有 RSS、Atom 等。RSS 和 Atom 两种技术相似，都是基于 XML 的文档格式，描述为摘要的相关信息列表[31]。在 Mashup 中，存在大量关于文本内容的集成。Web Feed 这种简单内容联合的方式，为 Mashup 开发者提供了相对分组的数据源。如用户关注体育新闻

领域，Mashup 平台可以选择性地将自身维护的体育方面订阅源展现给用户。用户不必重新开发自己的文件格式、传输协议和软件来实现内容的联合。

2.3.2 第三方API方式

API（Application Programming Interface，应用程序编程接口）是一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件的以访问一组例程的能力，而又无需访问源码，或理解内部工作机制的细节[32]。在 Web 2.0 时代，API 的应用让互联网上的公司、社区有条件创造一个共享数据和内容的开放架构。互联网上的内容和数据从它的创建开始，即可通过 API 的方式，在互联网上被多次浏览和使用。大多数互联网数据服务商均已通过 API 方式开放自己的数据，如 Google、Flickr，以及中国的豆瓣、新浪微博等[34]。这些 API 对于 Mashup 平台来说是十分合适的数据来源。

在 Mashup 平台中，API 的管理和发现是一个值得深度挖掘的问题[33]。用户如何在 Mashup 平台上各种数据 API 中找到自己所需的数据源，以及如何让用户快速清晰地理解各数据 API 之间的关系等，都是 Mashup 平台引入第三方数据 API 后必须考虑的问题。目前，API 的管理和发现有两种方案：自动完成和推荐[35]。自动完成适用于这样的场景：用户对使用哪些数据 API 有明确的要求，以及对 Mashup 的结果有明确的预期。在这样的情况下，用户只需选择和配置一些起始的数据源和指明某一些关键处理步骤，Mashup 平台将会基于模版，帮助用户自动生成一个 Mashup 应用[36]。推荐适用的场景更加广泛。推荐的主要思想在于基于用户建模的当前步骤，为用户推荐下一步可能选用的 API。生成推荐的依据有根据当前操作步骤的语义[37]、以当前步骤产生的结构化结果限定下一个步骤的输入[38]、社会化的协同推荐[39]等。

调用第三方 API 也会引入未知的问题。一个 Mashup 平台在引入了第三方 API 的同时，也将整个平台的健壮性和可用性严重依赖于提供 API 的数据服务商[40]。该 API 可能因为并发性支持、调用频率等种种原因不能提供很好的服务质量，这将导致 Mashup 平台上所有使用该 API 的 Mashup 应用收到影响。在引入第三方 API 的同时，系统应该实现一个管理框架，监控第三方 API 可用性状态和运行质量。甚至在 API 不可用时，无缝、透明地过渡到某种替换方案，以保证用户 Mashup 应用得以正常运行。

2.3.3 Web服务方式

Web Service 是一种平台独立的、松耦合的、基于可编程的 Web 的应用程序。它可以使用开放的 XML 标准来描述、发布、发现、协调和配置，十分适用于开发分布

式互操作性强的应用程序。Web Service 实现了在不同的系统之间通过软件对话的方式互相调用，将 Web Service 引入 Mashup 平台的数据源，可以实现基于 Web 无缝集成的目标[41]。Web Service 作为数据源有 SOAP 方式与 REST 方式[42]。

SOAP（Simple Object Access Protocol，简单对象访问协议）是用于交换 XML 信息的轻量级协议。SOAP 被设计成在互联网上交换结构化的和固化的信息，它可以和现存的其他互联网协议结合使用，包括 HTTP、SMTP、MIME 等。在 Mashup 的应用中，SOAP 主要用于远程过程调用。远程过程调用（RPC）用于在应用之间完成 CORBA 等对象之间的通信，但 HTTP 协议不是为此而设计的，由此带来了兼容性和安全性问题，防火墙和代理服务器甚至会阻止此类流量。通过 SOAP 与 HTTP 的结合，RPC 得以基于 HTTP 通信，并回避了此类问题[43]。

REST（Representational State Transfer，表述性状态转移）描述了一个架构样式的互联系统，它定义了一组架构约束条件和原则[44]。REST 强调信息本身，它将信息称为资源，每一个资源都有唯一的 URI 标识，当服务接到请求时，根据 URI 来决定响应。由于它简便、轻量级以及通过 HTTP 直接传输数据的特性，RESTful 成为基于 SOAP 服务的一个最有前途的替代方案。越来越多的服务供应商都提供对 REST 的支持，如 Amazon、Yahoo、Google 等。在 Mashup 平台上将 REST 引为数据源的同时挖掘 RESTful 服务中的语义，将可以提供更好的数据源服务[45]。

2.4 Mashup运行引擎的内部数据处理技术

Mashup 引擎在获取外部数据和资源后，还需对数据进行内部处理。

内部处理包括 Mashup 应用流程文件的解析、内部数据格式的定义、数据流流转的控制等。流程应用文件的解析是指引擎从数据库或前台编辑器捕获 Mashup 应用的运行请求，该请求中附带着序列化的 Mashup 应用构成及其配置参数，引擎需要通过反序列化、对象生成、参数配置等一系列步骤，将 Mashup 应用置于运行环境中。经过实践和探索，内部数据格式被设计为列表的形式，该列表中的每一个对象由一系列名值对构成，具备良好的拓展性和适配性。数据流的控制逻辑在 Mashup 应用中规定：应用中每一个操作子以指针的形式告诉引擎前驱后继，引擎在执行过程中根据该指针控制 Mashup 运行实体的数据流向。

此外，在本论文的依托项目中，Mashup 平台主要针对新闻这个垂直领域的数据聚合与应用。在新闻信息领域中，以下三个内部数据的处理技术是实际项目设计中所遇

到的关键问题。包括 Mashup 语义的预处理、地理信息的提取以及多源数据的去重。

2.4.1 Mashup语义的提取

Mashup 引擎在获取数据后，进一步还需根据用户的定制处理数据。面向终端用户和隐藏技术细节的定位，对引擎的智能化提出了一定的要求。例如，用户希望将新闻以发生地为地理坐标信息和地图进行整合，将新闻标记在地图上。再如，用户可能在聚合实时新闻后以新闻的价值度进行排序，并希望只阅读最具爆炸性的 5 个新闻。这一切都需要 Mashup 引擎对获取到的数据做进一步的分拣、处理甚至理解。

中文分词技术是语义理解的基础。在英文的语法中，词与词之间以空格分隔，计算机可以很容易地通过连接词义理解一句话的意思。而中文以字为单位，没有一个明显的标识告诉计算机意义分隔的位置，如何把中文的汉字序列切分成有意义的词，这就是中文分词技术。

现有的中文分词算法可以分为三类：基于字符串匹配的算法、基于理解的算法和基于统计的算法[46]。基于字符串匹配的算法又称为机械分词方法，它是按照一定的策略将汉字序列与一个机器词典进行匹配，若在词典中找到某个字符串，则匹配成功，即识别出一个词。它按照扫描的方式又可细分为正向最大匹配法、逆向最大匹配法和最少切分法。基于理解的算法是通过让计算机模拟人对句子的理解，达到识别词的效果。其基本思想就是在分词的同时进行句法、语义分析，利用句法信息和语义信息来处理歧义现象。它通常包括三个部分：分词子系统、句法语义子系统、总控部分。在总控部分的协调下，分词子系统可以获得有关词、句子等的句法和语义信息来对分词歧义进行判断，即它模拟了人对句子的理解过程。这种分词方法需要使用大量的语言知识和信息。由于汉语语言知识的笼统、复杂性，难以将各种语言信息组织成机器可直接读取的形式，因此目前基于理解的分词系统还处在试验阶段。基于统计的算法基于这样一种理论：从形式上看，词是稳定的字的组合，因此在上下文中，相邻的字同时出现的次数越多，就越有可能构成一个词。因此字与字相邻共现的频率或概率能够较好的反映成词的可信度。基于统计的算法对语料中相邻共现的各个字的组合的频度进行统计，计算它们的互现信息。由于中文是一门十分复杂的语言，让计算机理解中文语言十分困难。当前在中文分词领域，歧义识别和新词识别这两大难题一直没有完全突破。

在本平台的 Mashup 引擎中，使用一款开源的 Java 轻量级中文分词工具包 IK

Analyzer 作为分词功能的实现。IK Analyzer 在 2006 年 12 月推出，最初它是以开源项目 Luence 为应用主体的。结合词典分词和文法分析算法的中文分词组件，凭借歧义分析算法优化查询关键字的搜索排列组合，它能极大的提高 Luence 检索的命中率。后续 IK Analyzer 发展为面向 Java 的公用分词组件，独立于 Luence 项目。它采用了特有的正向迭代最细粒度切分算法，具有高效、快速的处理能力。

Mashup 引擎对内部数据进行分词，有助于引擎进一步理解 Mashup 中的语义信息，为后续地理信息提取和新闻的去重提供了基础。

2.4.2 Mashup地理信息的提取

Mashup 平台引擎负责对信息的地理位置标记。而地理位置信息隐藏在新闻数据中，引擎需根据新闻所描述事件提取出实际发生地点。一个普适性的做法就是根据新闻数据的分词结果与地理信息数据库进行匹配查找。这里的问题主要涉及两个方面，其一是地理位置关键字的查找效率，其二是地理位置关键字的查找准确性。

本平台使用的地理信息数据库涵盖中国四级行政单位以及大部分常用世界地名。表项接近四万项。在查找效率方面，通过哈希索引、集合加载等数据结构级别的优化来保证匹配和查找的速度。

一种基本的地点查找匹配方法是简单的将新闻文本中按顺序找到的第一个地理位置的名词作为查找结果输出。当一篇新闻中包含有不止一个地理位置名词时，依据这种方法获取的地理位置信息准确度将大大降低。假设一篇新闻中顺序包含“中国”、“上海”、“静安区”三个地理位置的名词，当匹配到“中国”时就将其作为该新闻的地理坐标显然是不够合理的。

本平台对地理信息数据库的地名按省市区的层级关系组织，并用类似邮编的机制标记一个地理信息描述的准确度。当 Mashup 应用中的新闻数据分析到多个地名时，比较获得的各地名精确度，优选最精确的地理信息描述。如上例中，选择了“静安区”。并回溯获得完整地址“中国上海市静安区”。

2.4.3 Mashup新闻的去重

Mashup 的一个核心功能是数据的聚合。当 Mashup 应用在新闻领域，一个问题不容忽视：当发生重大新闻时，用户编排的 Mashup 应用中来自不同新闻源的新闻数据在报道同一个内容。为用户优先提供重大新闻、以及如何去除重大新闻的重复性是 Mashup 引擎必须面对和解决的技术问题。

多源新闻数据的相似度计算模型建立在多维向量空间模型之上。向量空间模型在20 世纪 60 年代末由 Salton 等人提出，其基本思想是假设词与词之间是不相关的，以向量来表示文本，从而简化了文本中关键词之间的复杂关系，使得模型具备了可计算性[47]。

在这种模型中，文本 Document 用 D 表示，特征项 Term 用 T 表示。特征项是指出现在文档 D 中且能够代表该文档内容的基本语言单位，主要是由词或者短语构成。文本可以用特征项集合表示为 $D(T_1,T_2,\cdots,T_n)$，其中 T_k 是特征项，$1\leqslant k\leqslant n$。对含有 n 个特征项的文本而言，通常会给每个特征项赋予一定的权重表示其重要程度。即 $D(T_1,W_1;T_2,W_2;\cdots T_n,W_n)$，简记为 $D=D(W_1,W_2,\cdots,W_n)$，将其作为文本 D 的向量表示。其中 W_k 是 T_k 的权重，$1\leqslant k\leqslant n$。在向量空间模型中，两个文本 D_1 和 D_2 间的内容相关度 $Sim(D_1,D_2)$用向量之间夹角的余弦值表示：

$$Sim(D_1,D_2)=\cos\theta=\frac{\sum_{k=1}^{n}W_{1k}\times W_{2k}}{\sqrt{(\sum_{k=1}^{n}W_{1k}^2)\times(\sum_{k=1}^{n}W_{2k}^2)}}$$

其中，W_{1k}、W_{2k} 分别表示文本 D_1 和 D_2 第 K 个特征项的权值，$1\leqslant k\leqslant n$。

在本平台的新闻数据相似度计算中，权值计算通过如下公式进行：

$$W=\frac{n}{N}\times\frac{100000000}{freq}$$

W 表示一个词语在多维向量空间中所对应的某一个方向上的权值。其中，n 代表在一个新闻中某特征项（即某个词语）在这条新闻中出现的次数，N 代表这个新闻中特征项的总数目（即分词后的词语个数），n/N 代表新闻中的一个特征项对这条新闻影响的比例大小，freq 为该词语基于统计的词频。常数为修正因子。

本平台的词频来自于搜狗实验室统计的数据[48]。搜狗实验室通过搜狗搜索引擎索引到的中文互联网语料统计分析，在一亿页面以上规模的数据中统计出的词条约150000 项。

通过上述方法，引擎计算来自不同新闻源的新闻之间是否具有一定的相似度。具体操作时，考虑到效率和效果的平衡，以不同的权重，使用标题和摘要进行相似度比较。当两篇新闻相似时，取其中一篇新闻作为 Mashup 融合结果，并提升其权重值，以表示该新闻被多源报道，具有更高的热度和重要性。

2.5 传统的系统调度方法

Mashup 应用平台的处理规模有限，当请求并发多时，需要采取一定的分配策略来分时地满足请求。这就涉及到调度方法。针对 Mashup 应用提出一种针对性的高性能调度方法是本文的主要研究内容之一，将在下一章中阐述。在本小节中将分析传统领域的系统调度算法。

传统的系统调度中，有一些经典的调度算法。主要包括先来先服务算法、短作业优先调度法、高响应比优先调度算法、时间片轮转调度算法、多级反馈调度算法和优先级调度算法等[49]。

先来先服务算法是一种非抢占式的调度算法，它按照作业提交或进程变为就绪状态的先后次序来分派资源。当前作业或进程如果占用资源，那么直到执行完或阻塞，才出让所占资源。先来先服务算法有利于长作业，而不利于短作业；有利于 CPU 繁忙的作业，而不利于 I/O 繁忙的作业。

短作业优先调度算法是对先来先服务算法的改进，它对预计执行时间短的作业进行优先分派资源，目标在于减少平均周转时间。短作业优先的调度算法对长作业非常不利，长作业有可能长时间得不到执行，而且该算法也未能依据作业的紧迫程度来划分执行的优先级。

高响应比优先算法是对先来先服务方式和短作业优先方式的一种综合平衡。先来先服务方式只考虑每个作业的等待时间而未考虑执行时间的长短，而短作业优先方式只考虑执行时间而未考虑等待时间的长短。因此，这两种调度算法在某些极端情况下会带来某些不便。高响应比调度策略同时考虑每个作业的等待时间长短和估计需要的执行时间长短，从中选出响应比最高的作业投入执行。

时间片轮转调度算法是让每个进程在就绪队列中的等待时间与享受服务的时间成正比例。每一个作业执行相同时长的时间片，最大程度的执行了公平的原则。

多级反馈队列算法是对时间片方式的发展。该算法设置了多个就绪队列，分别赋予不同的优先级。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长，如逐级加倍。新进程进入内存后，先投入队列 1 的末尾，按先来先服务算法调度；若按队列 1 一个时间片未能执行完，则降低投入到队列 2 的末尾，同样按先来先服务算法调度；如此下去，降低到最后的队列，则按时间片算法调度直到完成。多级反馈队列算法为提高系统吞吐率和缩短平均周转时间而照顾短进程，具有较多的优点。

优先级算法是多级队列算法的改进，平衡进程对响应时间的要求。可分成抢先式和非抢先式。从优先级的计算方法来看又可以分为静态优先级和动态优先级。其中静态优先级根据作业的紧急程度、作业的类型或作业的要求资源情况等确定，动态优先级根据进程占用资源时间长短来确定并即时反馈和调整。

上述这些经典调度算法中，没有可直接适用于 Mashup 引擎的调度,这是由 Mashup 的结构特点和运行特点所决定的。但这些传统领域的调度算法为引擎调度算法的设计提供了十分有意义的借鉴。

2.6 本章小结

本章首先对 Mashup 应用的模型、结构和表达方法进行了描述，作为本论文后续工作的展开基础。本章也对本论文所依托的的 Mashup 项目平台进行了层次划分，按功能划分为编辑层、展现层和运行层，并分别介绍了各层次所设计的关键技术。其中，运行层作为本论文的主要研究对象，其内外部数据处理技术被详细阐述。本章为本论文的工作探讨了技术方案，奠定了系统实现的基础。

第三章　内容整合引擎的高性能动态调度策略研究

本章将着力于内容整合引擎的高性能动态策略研究。本章首先根据平台的实际需求，引出提高内容整合引擎性能的必要性，然后，通过对 Mashup 应用的结构、运行特点的分析，提出一种基于“懒启动”的运行调度策略，对待执行的请求进行高效调度，从而达到运行引擎的高性能设计的目的。

3.1　调度策略设计的目标

随着用户量的增长，Mashup 平台上的 Mashup 应用会越来越多，随之带来负载和效率方面的问题。其中，Mashup 运行引擎遇到的问题尤其明显。随着用户量的增加，Mashup 应用的执行请求也随之增长，这就要求 Mashup 运行引擎能具备一定的并发处理能力。但另一方面，并发量的增加又带来资源的消耗和额外的开销。一个平台的资源总是有限的，如何利用有限的资源服务更多的 Mashup 应用，成为运行引擎的挑战。调度策略的合理设计，有助于提高 Mashup 应用的运行并发量，提升 Mashup 运行引擎的效率。

3.1.1　性能瓶颈的分析

如第二章中所述，一个 Mashup 应用可以记作树形结构，运行时由它的叶子节点获取外部数据源作为原始数据，然后经由树枝上的各个操作子的处理到达树根的节点作为最终结果输出。其中完成“合并”功能的操作子作为特殊的节点，它们处于树结构的分岔处，运行时将来自两个或多个不同的分支的中间结果汇聚。

通过观察和分析 Mashup 应用的树形结构描述，不难发现以下两个特点：

1. 每一个节点都代表着一个操作，或是从外部数据源获取数据，或是对当前数据流进行某种处理。这个操作相对独立。

2. 汇聚点的节点操作对其它节点有依赖：必须等待各个输入流数据完备才能运行。

Mashup 应用在运行时是数据流的形态。数据源从 Mashup 树叶子节点处被填充、之后再沿着树的枝干向根节点汇聚。结合上述第一个特点，可以推论出 Mashup 应用在平台中所占用的最主要资源为内存。即在高并发的情况下，内存资源比其他资源更容易遇到性能上的瓶颈。而第二个特点，又意味着 Mashup 数据流在这些汇聚节点会被阻塞，从而造成了不必要的内存占用。阻塞是这样引起的：如果汇聚节点的多个输

入数据流中有一个或多个未准备完毕，即有数据流未输入该汇聚节点，则即使其他各数据流已输入该汇聚节点，该汇聚节点也无法进行相应的处理，使得已准备完毕的各输入的数据流依然占用内存，造成内存资源无法及时得到释放，造成数据流的阻塞。

因此，现有技术中 Mashup 平台的内存资源的利用率较低。在这种情况下，释放 Mashup 在阻塞期对资源的占用可以显著提升平台性能。

3.1.2 性能衡量指标

为了合理消除内存瓶颈，提升内存的有效利用率，以“同步”的时间尺度作为切入点，引入“时延内存积”PMT 的概念：

Product of Memory and Time (PMT) = Memory * Holding Time

其中 M 表示一个调度单元占用内存的大小，T 表示该调度单元占用内存资源的时间。PMT 直接在时间和空间两个尺度上度量了内存的使用。在本论文论述的场景下，能够很好地用以衡量一个调度单元同步阻塞的资源代价。

3.1.3 调度设计目标

结合以上的分析，可以归纳出对于 Mashup 运行引擎，一个高效的调度策略应该具备以下特点：

1. 在一段时间内，内存资源具有较高的有效利用率。该策略应尽可能让真正需要处理的数据流装载在内存，让内存中的 Mashup 数据保持活跃的状态，一旦数据不再活跃应尽快清出内存；

2. 适当地保证公平性。该策略能让每个调度单元都有机会获取并相对公平地占用平台的内存资源；

3. 对一个调度单元来说，允许其自身做一些“以时间换空间”或“以空间换时间”的权衡；

4. 保证一个平台的 Mashup 运行结果产出率。

本论文的调度策略设计就致力于满足上面的目标。即设计一种调度策略，降低 Mashup 运行周期内的 PMT 占用，从而让平台的有限内存资源在一段时间内为更多的 Mashup 服务。亦即通过内存使用的优化，提升整个平台的 Mashup 运行吞吐率。

3.2 调度单元mashlet的设计

对于用户来说，一个 Mashup 的执行等待时间为整棵 Mashup 树执行完毕所需的时间，而对于执行引擎来说，实际上是以一个 Mashup 内的各个节点为单位运行的。这种情况导致在调度单元粒度划分上的矛盾：若将每个节点作为调度单元，则没有考虑到一个 Mashup 的整体性;若将整个 Mashup 作为一个调度单元又没有考虑到每个节点都是一个相对独立的执行操作子这个特点。因此,需要考虑一种兼顾两者的平衡机制。

本论文将 Mashup 中的汇聚操作子视作一个特殊的节点，对其上的分支进行拆解，将得到的多个 Mashup 分支称之为 mashlet。并将 mashlet 这种数据结构作为调度的基本单元。mashlet 是 Mashup 的片段，它是通过切割 Mashup 树上的分支生成的。

Mashup 切割成 mashlet 的过程具体如下描述：Mashup 应用的树形表达记作 M_p ={E, N} ，将 N 中的所有 Merge 操作子的集合记作 Mer，特别地，我们也将 End 节点放入 Mer 中。对于 Mer 中的每一个操作子节点 n，均可以初始化一个新的 mashlet，$mashlet_i$={N_i, E_i}，其中 N_i={n}，E_i=∅，然后，迭代地将 n 的前驱节点添加到 N_i 中，直到其前驱是∅或者 Merge 节点。与此同时，将相应的边也加入 E_i。当对 Mer 中的每一个节点都做如此处理后，一个 Mashup 应用相应的即被划分为若干 mashlet。

以第二章中的 Mashup 应用为例，如图 5 所示。左边是原 Mashup 树状结构图，以三个 Merge 节点作为分割点，Mashup 树被表示为右上图的形式，在这种表示中，每一个节点表示一个 mashlet。右下图是例举了 $mashlet_4$ 和 $mashlet_8$ 的构成，他们分别由三个操作子及其连接关系组成。

一个 Mashup 可以表示为 M_p={$mashlet_{p0}$,$mashlet_{p1}$,…,$mashlet_{pn}$}。其中，$mashlet_i$ 由一组操作子 $oprSeq_i$={opr_1,opr_2,…,opr_k}组成，它们决定了处理数据的逻辑。同时有一组输入流 $inputSet_i$={$input_i^1$, $input_i^2$,…, $input_i^n$}作为待处理数据，以及唯一一个输出流 $output_i$ 作为输出结果。故而一个 mashlet 可以表示为一个四元组 mashlet={id,inputSet,output,oprSeq}。一个 mashlet 的执行就是各输入数据流通过操作子的一系列处理最终形成输出流流出。当一个 Mashup 处理完它最后一个 mashlet 时也标志着该 Mashup 处理的完成。

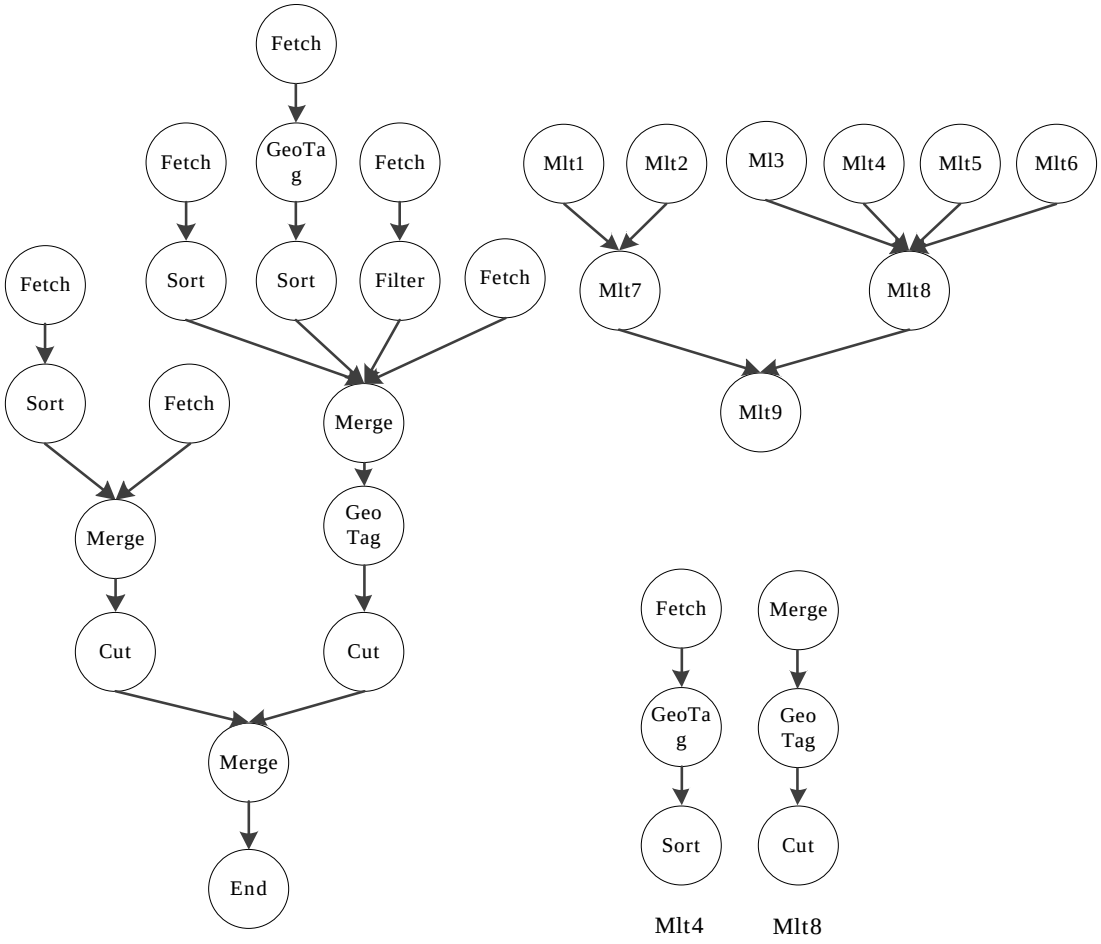

图 5 mashlet 及其拆分过程示例

3.3 基于懒启动的调度策略设计

基于上述的调度设计目标和调度单元设计，本节将提出基于懒启动的调度策略。

3.3.1 懒启动的调度设计

在引入 mashlet 的概念后，重新考察 PMT 值的意义。

对某个 mashlet 的 PMT 进行计算时，公式中的 Memory 表示该 mashlet 占用内存的大小，也就是该 mashlet 在运行时数据流预计或实际占用内存的大小；Holding Time 表示该 mashlet 占用内存资源的时间，也就是该 mashlet 在运行时数据流预计或实际占用内存的时间。PMT 的数值可以表征 mashlet 对内存资源的占用和消耗。

为了让整个平台的内存资源尽可能得到有效的利用，本论文的调度设计遵循这样一个原则：当一个 mashlet 的 PMT 值越大，该 mashlet 的运行优先级就越高。从而让

平台运行时引擎尽可能早地调度执行该 mashlet，释放该单元持有的资源。

由于 mashlet 由平台中的单一、稳健的操作子组成，基于数据分析和历史统计，可以得到一个大致准确的 mashlet 运行预估时间。将操作子 opr_i 的预估时间记作 T^i_{opr}，那么 $mashlet_i$ 的预估执行耗时为 $prTime_i = \sum T_{opr}$，即 $oprSeq_i$ 中所有 opr 的预计耗时之和。

记一个 mashlet 运行时预计占用或实际占用的 PMT 为 PMT_{run}，则

$$PMT_{run} = prTime * \sum_i input^i.MemSize$$

需要说明的是，一个 mashlet 所占内存基本等于各输入数据流的所占内存的大小之和，这是因为该 mashlet 利用其包括的操作子对数据流进行处理并交付到下一个 mashlet 之前，数据量不会发生大幅的变化。考虑到入度为零的 mashlet 一般以需要获取数据的操作子为起始操作子，例如具有 Fetch 功能的操作子，故对于该类操作子，将获取到数据之后该操作子所占用内存的大小，作为该操作子所占用内存的大小。

由于 PMT 的计算是在 mashlet 被运行之前，因此，该 mashlet 所占用内存的大小和占用的时间可以采用预估值。也就是，可以基于数据分析和历史统计，根据操作子被其他 Mashup 应用调用时实际运行中所占用的内存大小和运行时间，计算出基本准确的各操作子运行预计占用的内存大小和预计占用的时间。

当某个 mashlet 的 PMT 的数值较大时，说明该 mashlet 需要占用内存的时间较长，或该 mashlet 需要占用的内存资源较多，则该 mashlet 的优先级较高。相应地，运行引擎会优先运行该类 mashlet。

懒启动是指，由一个 Mashup 输出点所在的 mashlet 开始，倒推各个 mashlet 的开始执行时间，使得 mashlet 能在各个汇聚点几乎同时到达。从而消除同步等待所造成的阻塞。理想情况下懒启动会异步地启动 mashlet，在时间上更紧凑地使用引擎的内存资源，从而提升内存利用率。

懒启动的效果如图 6 所示，依然以图 5 所示的 Mashup 应用为例。在图中以横坐标表示占用内存，纵坐标表示持有时间，区块面积就表示了一个 mashlet 的 PMT 值。由于在该 Mashup 应用中 $mashlet_3$，$mashlet_4$，$mashlet_5$，$mashlet_6$ 作为相邻 mashlet，都将数据流转给 $mashlet_8$，$mashlet_8$ 则必须等待四个输入流均到达才可开始运行，否则即使获得一部分数据流入也必须持有等待。考察 $mashlet_4$ 中，有一个地理信息提取模块，该模块可能需要较长的时间，由此导致了 $mashlet_4$ 运行时间较长，实际的 PMT 占用如

图 6 上半部分中所示。$mashlet_3$，$mashlet_5$，$mashlet_6$ 被阻塞。浪费了浅灰色的内存时空。

通过使用懒启动的调度方式，$mashlet_3$，$mashlet_4$，$mashlet_5$，$mashlet_6$ 在通过逆推计算后被执行引擎异步地启动和执行，从而消除浅灰色部分所示的时空，优化内存的使用。直观效果如图 6 下半部分所示。

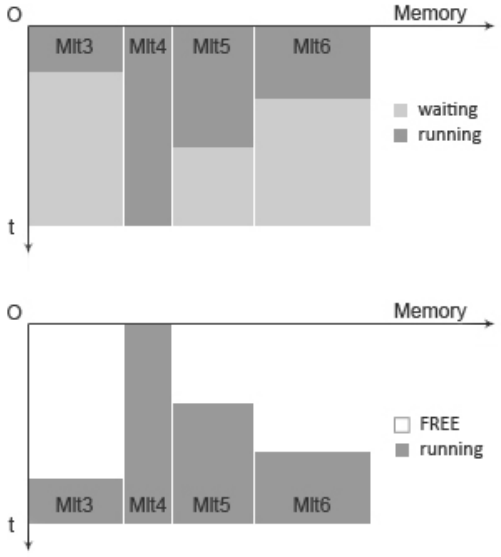

图 6 mashlet 懒启动对时空的优化效果

3.3.2 懒启动算法的修正

懒启动算法从一个 mashup 输出点所在的 mashlet 开始，倒推各个 mashlet 的开始执行时间，使得 mashlet 能在各个汇聚点几乎同时到达。理想情况下可以异步地启动 mashlet，从而消除同步等待所造成的阻塞，提升内存利用率。

然而，在实际调度情况中，有两个因素制约着懒启动算法的执行效果。

其一，运行引擎在初始时计算出 mashlet 的 PMT 数值，是根据预估的 mashlet 占用内存大小和时间作为计算依据的。根据预估的占用内存时间，理论上具有相同输出节点的 mashlet 运行完毕时的数据流基本可以同时到达作为输出节点的 mashlet。但是在实际运行过程中，无法保证理论上预计的准确程度，也就是说，在实际运行过程中，具有相同输出节点的 mashlet 运行完毕后到达作为输出节点的 mashlet 时间，可能存在时间差，仍然存在同步等待的问题。

其二，由于引擎的运行负载是有限的，不能保证每一个 mashlet 在懒启动的开始时间即被执行。即 mashlet 在预设时间 t_0 进入待执行状态，但不一定在该时刻被调度执行，从而导致了到达汇聚点的时候比预估的晚，重新带来了阻塞。

上述两个原因制约了懒启动的应用效果，导致其执行效果与理论设想下的效果存在偏差。本论文以一种修正机制来动态调整 mashlet 的 PMT 值，从而尽量保证懒启动调度的内存优化效果。

为了明确问题的描述，我们将 mashlet 分为两种类型，并暂称为 A 类 mashlet 和 B 类 mashlet。其中 A 类 mashlet 包含源 Mashup 树形结构的叶子节点，它们没有输入，自身的第一个操作子是获取数据源的操作；B 类的 mashlet 以 Merge 节点作为起点操作子，它有若干个输入。在以图 5 为例的 Mashup 应用中，$mashlet_1$，$mashlet_2$，$mashlet_3$，$mashlet_4$，$mashlet_5$，$mashlet_6$ 都是 A 类 mashlet；$mashlet_7$，$mashlet_8$，$mashlet_9$ 为 B 类 mashlet。在本论文中，在考察一支 mashlet 时，不考虑其他相关 mashlet 执行时延带来的连锁变化。

当一个 A 类型的 $mashlet_a$ 进入调度队列后，按预估时间对其安排的开始执行时间为 t_0^a，但是在实际的执行过程中，$mashlet_a$ 在 t_0^a 并没能被调度执行。那么在 t 时刻，$mashlet_a$ 的相邻调度单元都由于不能进行下一步而被阻塞，阻塞时间为 $(t-t_0^a)$，阻塞的内存大小为各相邻调度单元输出的内存尺寸。所谓相邻调度单元是具有相同输出节点的 mashlet 之间互相的称呼。在以图 3 为例的 Mashup 应用中，$mashlet_1$ 和 $mashlet_2$ 互为相邻调度单元。

用 PMT_{delay}^a 来表示这个同步等待的时空资源代价，有：

$$PMT_{delay}^a=(t-t_0^a)*\sum_i output_{adj}^i.MemSize$$

再来看 B 类型的 $mashlet_b$，当它有调度延时，给平台空间带来的耗费来自于它各个就绪的输入流。

$$PMT_{delay}^b=(t-t_0^b)*\sum_j output_{prec}^j.MemSize$$

将上述两个计算式写作更一般性的表达，用来表示一个 mashlet 没有按预设时间启动而带来的资源代价，来自两个部分：分别是 a 部分代表的来自相邻 mashlet 和 b 部分代表的以自身作为输出点的前驱 mashlet。记为 PMT_{delay}

$$PMT_{delay}=(t-t_0)*(\sum_i output_{adj}^i.MemSize+\sum_j output_{prec}^j.MemSize)$$

引入 PMT_{delay} 后，我们将 PMT 的计算值调整 $PMT=PMT_{run}+PMT_{delay}$。当调度系统

再以 PMT 作为调度因子计算优先级时，PMT 更能表现一个 mashlet 被执行的急迫度，并根据当前 mashlet 运行状态动态地调整和重计算着 PMT 的值，从而能消除本节提出的现实影响，让调度策略也更加高效、准确。

3.4 基于懒启动的调度策略效果评估

基于懒启动的调度策略归纳的说可以认为是一种执行引擎动态地根据 PMT 的值，从高到低地运行拆分 Mashup 所获得的 mashlet。具体 PMT 的计算方法如前文所述。

观察 PMT 的组成我们发现，一个 mashlet 进入调度队列后，随着时间的增长，它的 PMT_{delay} 将会显著增大，这种动态性是保证 mashlet 不会饥饿的安全机制。

一个 Mashup 应用在被运行引擎分解为若干个 mashlet 之后，各 mashlet 被加入队列中进行无差别的公平竞争，各 mashlet 通过 PMT 数值的大小进行竞争的过程看似独立，但是 PMT_{delay} 的计算方法能够有助于保持 Mashup 应用的整体性：当一个 Mashup 应用中的一个 mashlet 被运行之后，势必由于它占用了内存资源，导致与它属于同一个 Mashup 应用的其他 mashlet 的 PMT 的数值增大，相应地 PMT 数值增大的 mashlet 优先级被提高。从而使同属于一个 Mashup 的未运行 mashlet 优先级提高，并被尽快被运行。这样的连锁反应会随着该 Mashup 应用中的各 mashlet 逐个被运行而越发明显。即一个 Mashup 应用的运行整体具有较好的连贯性。从 Mashup 平台的整体上来看，调度过程中对队列中来自同一个 Mashup 应用的 mashlet 有类聚的效果，从而保证了平台的结果产出率。

据此，基于懒启动的调度算法能满足前文提出的高效调度算法设计需求。

3.5 本章小结

本章首先根据 Mashup 应用的结构和运行特点，分析了 Mashup 运行时所遇到的性能瓶颈。继而根据性能瓶颈的分析，提出了一种基于懒启动的动态调度策略。在这种策略的理论计算下，Mashup 可以被高效有序地调度，提升 Mashup 内容整合引擎的性能。在实际应用中，本文又给出了该策略和算法的一种修正，以更加精准地调度 Mashup。最后，对懒启动的调度算法进行了效果的评估，证实了该算法的有序和高效。

第四章　内容整合引擎对执行对象的静态优化研究

本章将阐述内容整合引擎在静态优化策略上的研究。本章首先根据平台的面向对象，分析出平台上的 Mashup 应用具有静态优化的可能和必要；其次，详细设计一组静态优化的理论方法和顺序规则；最后对效果进行了评估。

4.1　静态优化的可行性分析

Mashup 成为 Web 2.0 时代广泛流行的技术，原因之一在于它有广泛的群众基础。Mashup 目的在于为没有编程基础的普通终端用户提供一种信息二次处理的解决方案，用快速构建、即时使用的 Mashup 应用替代了功能专业、开发冗长的传统软件，满足了终端普通用户对于数据整合和处理的简单需求。

面向终端用户的建模带来了操作便利、门槛低等优点的同时，也带来了一些软件工程领域的问题。

由于 Mashup 的用户是没有编程基础的终端用户，他们不具备程序员所具有的一般编程基础甚至编程逻辑。这部分用户在平台上创建和编辑，将会给平台中带来低效、冗余，甚至是错误的 Mashup 应用。

假设用户在平台上随意地创建了如图 7 的 Mashup 应用。在该应用中，用户的逻辑需求是从三个 RSS 订阅源订阅了新闻，并将他们与一个来自网页的新闻数据源整合，其后做了一些例如提取地理信息、过滤了非今日新闻、条目限制等操作，最终返回一个通过整合后的个性化新闻条目和内容的展示列表。

该 Mashup 是用户随意拖拽生成的，在组织上起码可以有三处优化。

在 A 处，用户的组织是先对来自网页的新闻源进行“按作者姓氏排序”，然后对新闻进行按发布时间的过滤，滤除非今日的新闻条目。可以注意到，排序的关键字与过滤的条件不是同一个数据项，粗略看来先排序后过滤和先过滤后排序在这里没有影响。假设从网页获取的新闻数据项有 30 个条目，再次考虑这两种顺序可以看出效率上的差别：在先排序后过滤的情况下，排序操作子对 30 个数据项进行操作，过滤操作子对 30 个数据项操作，最后满足条件为 9 个数据项。而在先过滤后排序的情况下，过滤操作子的操作没有变化，而排序操作子只需对过滤后的结果，即 9 个数据项进行操作。

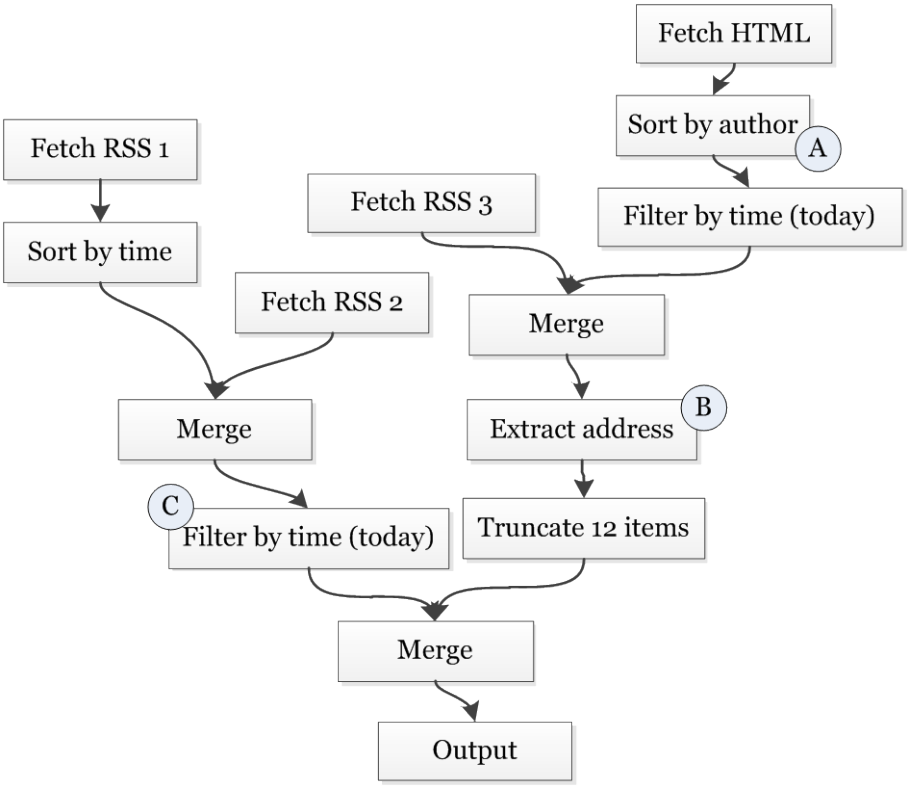

图 7 普通用户创建的可优化 Mashup 应用示例

同理，在 B 处，截取操作也可以和地理信息提取的操作交换位置。从而减少地理信息提取操作子的操作项数。

而在 C 处，过滤操作子出现在合并操作子之后，在此处，可以将 Filter 操作子移动到合并之前的两个 mashlet 中，这样一则让来自 RSS1 的数据源可以如 A 一般先过滤后排序以减少排序的项数，二则减少了该 Mashup 应用的 mashlet 调度支。即 C 处之前的 Merge 操作子可以取消，让它的输入项直接成为 C 处之后 Merge 操作子的输入项。

经过这三处的调整，该 Mashup 的执行过程将更有效率。

由此可以看出，用户直接编辑并提交的 Mashup 应用具有效率低下、逻辑不清等问题。这种问题在面向普通用户的 Mashup 平台具有一定的普遍性。

用户在创建和编辑 Mashup 应用后，会将 Mashup 应用提交到平台即时运行或保存留待之后再次运行。在留待后续运行时，Mashup 应用以文件的形式保存在平台上的 Mashup 资源库，这种机制为 Mashup 应用的静态优化提供了时机。Mashup 平台可以在用户离线时，透明地为用户优化 Mashup 应用，以将性能低下的应用转化为高效的应用，从而提升该 Mashup 应用在交付引擎执行时的资

源占用，变相地提升引擎的吞吐性能。

4.2 静态优化策略的设计

由上一节分析可以看出，个性化的 Mashup 应用执行效率低下的主要原因在于Mashup应用的编写者没有编程基础，带来了一些逻辑上和组织上的冗余，静态优化的目的即在于能消除这部分冗余，让 Mashup 应用以一种高效的方式存在和运行。

在这里，Mashup 应用的逻辑是指用户创建该 Mashup 的意图以及该 Mashup 完成的业务流程。如用户从新浪、腾讯等几个网站获取新闻数据源，在按时间排序后截取前十条显示。Mashup 应用的组织是指用户在平台上创建这个 Mashup 的过程中，具体选用的操作子，以及操作子的前后关系、连接关系等。

在逻辑上的冗余往往难以通过程序判断。由于 Mashup 应用是一种个性化的 Web 应用，无法从一个 Mashup 应用的逻辑本身来推测和判断用户创建该 Mashup 应用的本意。于此同时，作为一个提供服务的平台，考虑到用户隐私等问题，也不应擅自分析、更改用户创建的 Mashup 逻辑。

组织上的冗余则是平台可以处理的。平台处理组织上的冗余只是为了让 Mashup 引擎能以更高效的方法执行，并不改变用户指定的 Mashup 逻辑。

4.2.1 静态优化的设计思路

本论文主要采用操作子的重新排序来对 Mashup 应用进行静态优化。

操作子的重新排序是指，通过标准化 Mashup 应用的表现形式，规定操作子在一个 Mashup 中出现的先后顺序，以此为准则对 Mashup 重新组织，同时保持 Mashup 的逻辑和语义不变。

如前文中所述，Mashup 应用由一些操作子组成，这些操作子由 Mashup 平台提供。在分析中发现，有一些操作子在 Mashup 应用中出现的顺序与 Mashup 最终的结果并无关系。

例如，Sort 排序和 Filter 过滤两个操作子，对于一个数据流，先对其以某个数据项为主键排序还是先对数据项进行满足某种规则的过滤，对操作之后的输出数据流完全没有影响。也即排序和过滤这两个操作子的出现位置是可互换的。

再如 Rename 重命名操作子，它的出现与单输入单输出的操作子基本都可

以互换位置，而不影响结果。

这类有可能进行位置调整而不影响最终结果的操作子是将重排序应用于的静态优化的基础。

4.2.2 静态优化的规则设计

针对这些可调位置的操作子，可以设计一组静态优化的规则，规定某些操作子出现的标准顺序。静态优化的过程也就是将 Mashup 应用匹配这些规则并调整的过程。

本论文主要提出以下两个顺序规则，作为静态优化的依据。

1. 当操作子出现在一个Mashup中的某个分支中时，将其规整为一定顺序；
2. 某些可互换操作子尽量往 Mashup 树的叶端移动。

针对 Mashup 中的分支，亦即 mashlet，应用第一个顺序规则。即将该 mashlet 中所出现的操作子按一定顺序互换位置，规整为统一顺序。一些具体的规整方法如下，符号“→”表示其左边的操作子顺序应在右边操作子之前。

过滤→截取

当一个 mashlet 中出现过滤操作子和截取操作子时，将其重新组织为先过滤后截取的顺序。过滤操作子和截取操作子是可交换的，这是因为过滤操作根据中间结果是否满足某项条件而进行筛选，而截取操作仅仅是对最终结果做条目限制处理。将顺序限定为先过滤后截取有助于保证内部结构顺序的一致性。虽然可能对结果的保留个数有些微变动和差异，但是基本没有改变用户意图。

过滤→排序

当一个 mashlet 中出现过滤操作子和排序操作子时，将其重新组织为先过滤后排序的顺序。排序是一个耗时操作，对单规则进行排序时，通常使用的快速排序算法也需要 O(nlogn)的复杂度，在 Mashup 应用中，用户可能指定了一系列排序规则，故更应该考虑排序的有效项数。当过滤操作子和排序操作子一起出现时，应该先进行过滤操作，尽早排除不符合规则的条目，减少排序操作处理的项数。例如，用户处理一组来自网上的 CD 库目录数据，当用户编写该 Mashup 应用时，并不知道这组数据的数量级，故编排的顺序是，先按 CD 的发行年份进行时间排序，然后滤取音乐人是“孙燕姿”的 CD 条目。实际处理时，CD 库目录的数量级为 10000，而孙燕姿的 CD 数为 8。如果不进行顺序规整，

引擎需对 10000 个 CD 条目进行排序，然后过滤出 8 条有效数据。在这样的情景下，顺序调整结果明显，调整后引擎只需过滤出 8 条有效数据，后对该 8 条数据进行排序即可。

截取→重命名、提取地理信息

在一个 mashlet 中，将截取操作换序到重命名、提取地理信息等原项操作子之前。重命名、提取地理信息等操作称为原项操作是指，这类操作只对 Mashup 中间结果中的每一项数据内进行操作，而不涉及数据项之间的操作。这类操作子的出现顺序，不影响中间结果的项数、顺序和其他特征。将此类操作子置于截取操作子之后，提高了操作的有效性，避免了一些不必要操作。如用户对一个包含 30 个条目的新闻源做地理标记和截取前 5 条的操作，当规整为先截取后提取地理信息的顺序时，提取地理信息这个耗时的模块可以只处理对结果有效的前 5 条数据。

此外，还可应用第二条静态优化规则，将可互换操作子尽量往 Mashup 树的叶端移动。此类移动涉及 mashlet 间的移动，最佳时机应该在 Mashup 剪枝成为 mashlet 之前进行。

该规则适用于过滤、截取等明显降低中间结果项数的操作子。

这一类的优化在 PMT 的计算方法下效果十分明显。如图 8 所示，一个 Mashup 应用从两个电商网站获取了一系列价格数据，汇合后筛选了 200 元以下价格数据并排序显示。原始的 Mashup 应用如左图所示。在静态优化后转化为右图的结构。具体来说，即阴影部分的过滤操作子向叶端移动，经过汇聚点时分别移动到两个分支。从结构上看，由于多了重复的过滤操作子好像变得更加冗余。其实从本文论述的调度单元 mashlet 及其 PMT 的分析可以看到，优化前后的两个 Mashup 都将被剪枝成三个 mashlet，而在调度时，由于右图的前两个 mashlet 都已经完成数据的过滤，减少了一些对结果无效的数据项，故可以在调度等待时占用更少的内存，由此也减少了整个 Mashup 应用执行过程中的内存占用，让它的执行更加高效。

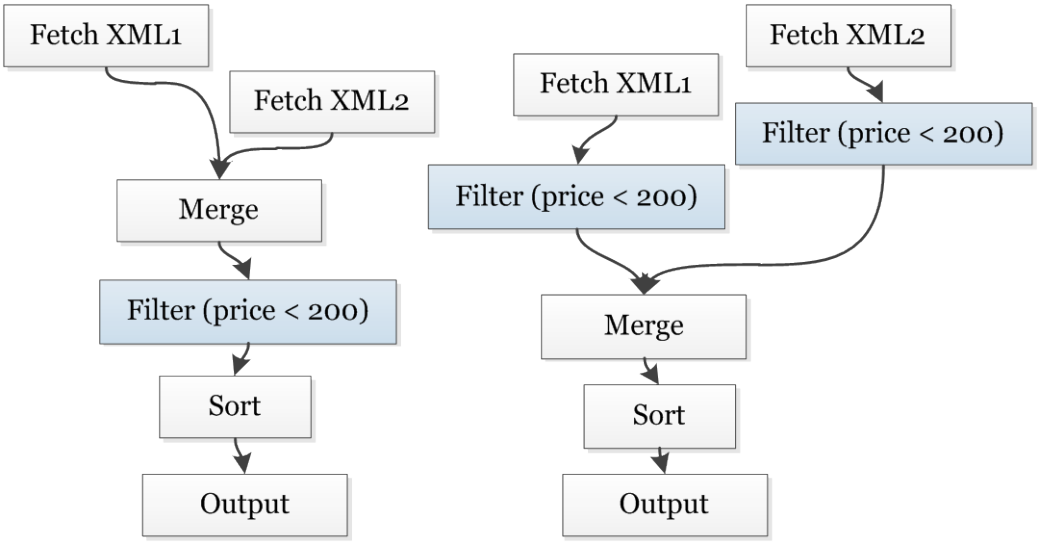

图 8 静态优化前后 Mashup 结构对比

4.2.3 静态优化的处理时机

静态优化由引擎中的结构优化器模块进行处理。静态优化主要有两个时机处理用户的 Mashup 应用。

Mashup 平台有一个 Mashup 资源库，保存所有用户创建、编辑过的 Mashup 应用。Mashup 应用以文件的形式在资源库中。当没有用户调用时，这些 Mashup 应用属于离线状态。引擎可以在用户请求少的时间段开启定时任务，对资源库中的 Mashup 应用进行分析和优化。在这种情况下，静态优化的过程应采用事务机制。当引擎对某一个 Mashup 应用正在处理时，用户突然访问该 Mashup，需对优化的处理进行回滚，以防止访问冲突。

静态优化的另一个时机在于运行时。引擎接收到的 Mashup 运行请求有超过一半是没有经过资源库，而是用户在创建、编辑完成后直接提交请求执行的。这种情况下的 Mashup 应用没有经过资源库的离线优化。这时，引擎中的结构优化器在 Mashup 解构为 mashlet 后，针对每个 mashlet 进行快速的分析处理，在将 mashlet 提交引擎执行前完成其静态优化。

4.3 静态优化策略的效果评估

静态排序无论是对 Mashup 应用整体，还是对 mashlet 调度支，都起到了一定的优化作用。对于 Mashup 应用的优化将可能减少 mashlet 调度支，从而减少

运行一个 Mashup 所需要的总共调度次数和运行步骤；对于 mashlet 的优化将可能减少一个 mashlet 运行时所需的总体空间，减少对平台的资源占用，从而使拥有一定资源的 Mashup 平台在单位时间内服务更多的 mashlet，提升 Mashup 执行吞吐率。

操作子的重排序，可以让 Mashup 应用更加高效。由于所有的排序规则是由平台设计者经过研究调查事先规定，并通过分析确认确实有效，相比于终端用户随性、直观的组织更具科学性，相应的，效率也就得到保证。

操作子的重排序，可以规整平台上用户创建的所有 Mashup 应用，让 Mashup 资源库中的应用都以一种标准化的形态存在。这样的好处在于平台在面向终端用户时，是个性化、灵活编程的。但在平台的内部，实际上具有一定的标准性和通用性，为处理、运行、研究 Mashup 应用提供了进一步的可能。例如后续可否将相同的 mashlet 或者更小的划分单元归并执行一次再分发结果等，这将在本论文最后的工作展望中阐述。

4.4 本章小结

本章详细介绍了内容整合引擎的静态优化方法，作为引擎性能提升的辅助。首先分析了静态优化的可行性和必要性，之后对优化策略进行了详细的设计，提出了一组优化规则，作为引擎对 Mashup 应用进行优化的依据。此外，还分析了静态优化的最佳处理时机。最后，对静态优化的效果进行了评估和阐述。

第五章　高性能内容整合引擎的设计与实现

本章将结合之前两章的论述，将动态和静态两种性能优化机制引入平台，首先给出一个系统的总体设计，其次将描述主要模块的功能设计和功能实现。本章还将提供一些实际环境中的平台运行效果，以说明高性能内容整合引擎的成功运用。

5.1　系统的总体设计

本论文将致力实现一个高性能的 Mashup 引擎，所谓的高性能从两个方面实现，分别是基于懒启动的动态调度方法和静态的 Mashup 优化，具体的方法已在前两章中详述。

本论文所实现的 Mashup 运行引擎作为整个 Mashup 平台的核心部分，与 Mashup 编辑器和 Mashup 手机客户端共同组成 Mashup 平台的整体。编辑器与手机客户端的工作不属于本论文的研究内容，故在此不再详述。

高性能 Mashup 引擎的结构设计及其在 Mashup 平台中的角色如图 9 所示。

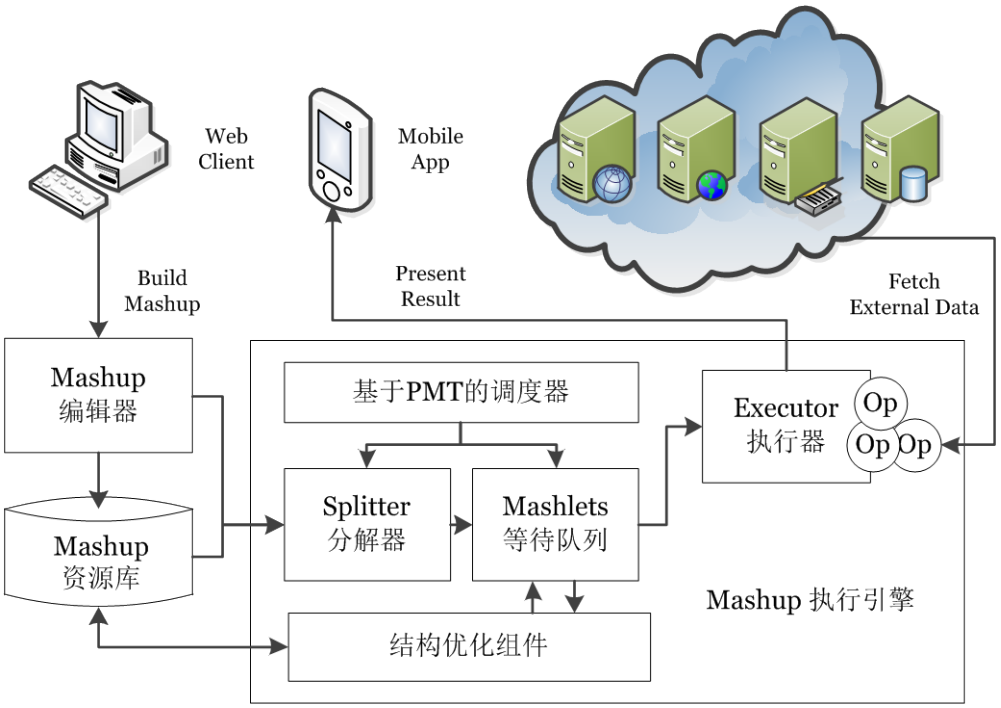

图 9　内容整合引擎的总体设计

图 9 中描绘了本论文依托项目中整个 Mashup 平台的结构，其中 Mashup 编辑器，供用户自由地创建、编辑 Mashup 应用；Mashup 资源库保存所有用户的 Mashup 应用。Mobile App 是一个适用于 Android 系统的一个移动客户端，用户可以通过该客户端随

时查看自建 Mashup 应用的实时结果。Mashup 执行引擎即为本论文研究和设计的高性能 Mashup 运行引擎。

高性能 Mashup 运行引擎由五个部分组成。它们分别是 Splitter 分解器、Mashlets 待执行队列、Executor 执行器、基于 PMT 的调度器和结构优化器。

- Splitter 分解器运行在引擎的起始，直接与用户交互。它在收到用户的 Mashup 应用运行请求后，负责将用户创建的 Mashup 分割成一组 mashlet，具体的算法如第三章中所描述。
- Mashlets 等待队列是一个待执行 mashlet 队列，在该队列中所有的 mashlet 都处于 ready 状态，随时等待被调入执行器执行。
- Executor 执行器是引擎中真正执行 mashlet 操作子的部分。它维持一系列操作子线程池，根据当前的 mashlet 中的操作子序列，选择相应操作子对 mashlet 的数据进行操作。这些操作子可能需要获取外部资源或数据，由执行器发起、维护、和结束与外部服务器的交互。
- 基于 PMT 的调度器负责从 Mashlets 等待队列中选择合适的 mashlets 交付给执行器执行，同时负责维护和刷新 Mashlet 等待队列中所有 mashlet 的 PMT 值。
- 结构优化器负责处理第四章中描述的对 Mashup 应用的静态优化。它通过对仓库中的 Mashup 应用进行离线分析，进行操作子顺序调优等操作；或对用户编辑后直接提交的 Mashup 请求进行重排优化。

Mashup 运行请求被引擎接收后，处理流程如下：

1. 运行平台每接受一个 Mashup 应用的执行请求，由 Splitter 分解器在汇聚节点将其分解成一组 mashlet；分解器同时将这些 mashlet 做一些初始化配置，如预估每一支 mashlet 的运行时间和内存占用，作为原始的 PMT 值写入该 mashlet 属性；
2. 这些 mashlet 进入待调度队列；
3. 随着时间的变化，调度队列中 mashlet 的 PMT 不断重新计算和刷新，同时待调度队列中优先级最高的 mashlet 弹出并被引擎执行。

这里，PMT 的刷新和重计算主要包括两个方面：一是当输出产生内存占用时，以实际内存占用与实际持有时间代替预置的估算值；二是随着时间的变化 PMT 的增大。

内容整合平台执行 Mashup 应用的流程图如图 10 所示。

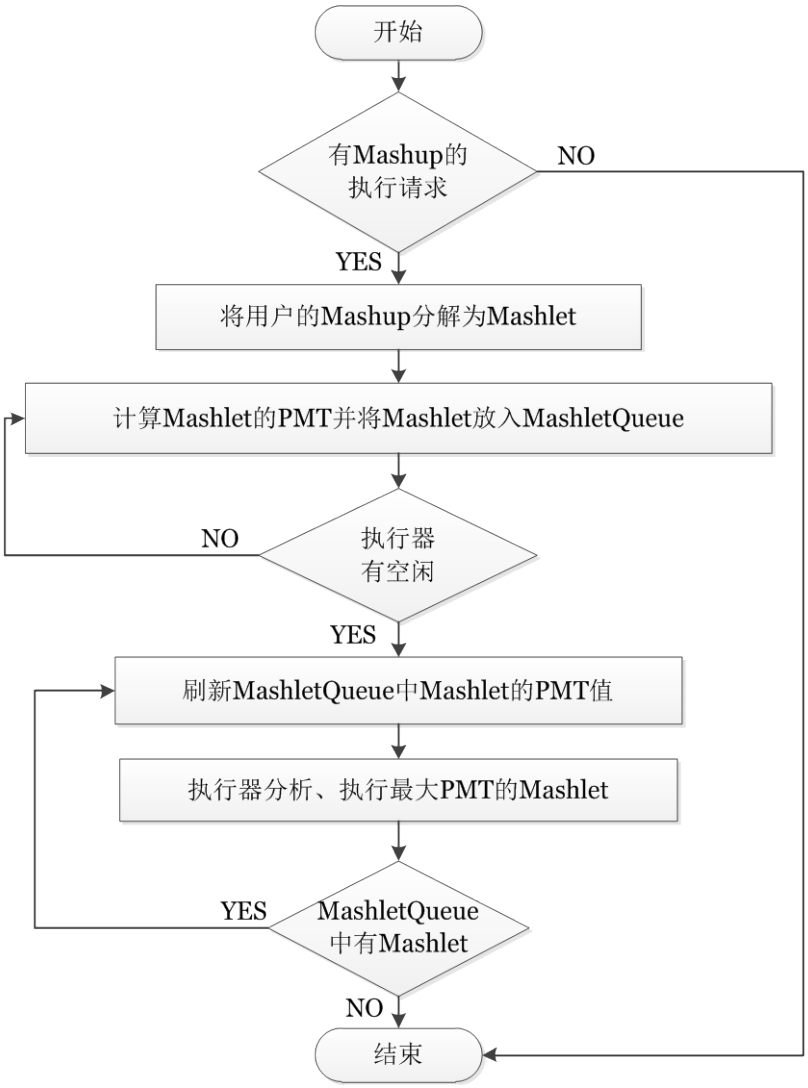

图 10 Mashup 应用的处理流程图

5.2 主要模块的设计与实现

内容整合引擎由 Splitter 分解器、Mashlets Queue 待执行队列、Executor 执行器、基于 PMT 的调度器和结构优化器五部分组成。这其中，又以分解器、调度器和执行器为重要的核心部分。

5.2.1 分解器的设计与实现

Splitter 分解器负责将 Mashup 应用剪枝处理为 mashlet。并需配置 mashlet 的一些初始参数和记录一些统计信息。

Splitter 分解器在收到用户创建的 Mashup 应用后，首先根据 End 操作子标记将该 Mashup 树的根节点压入一个临时节点集 beginNodes。beginNodes 标记了该 Mashup 应

用可生成的所有 mashlet 的种子节点。然后，分解器对 beginNodes 节点集合中的每一个节点，依序取其前置操作节点。当该前置节点是另外的 Merge 操作子时，将其压入 beginNodes 作为另一个 mashlet 的种子节点留待之后操作，否则作为当前 mashlet 的一部分。每生成一个 mashlet 对象分解器还需加入额外的信息，如基于统计数据的 PMT 估值。在处理完 beginNodes 中的所有种子节点后，一个 Mashup 即完成了 mashlet 的切分，算法描述如表 1。

表 1　Mashlet 的切割和生成算法

Algorithm: Split Mashup to Mashlets

Require: a tree representation of mashup and the root node.

Output: a set of mashlet.

```
1    beginNodes ← root
2    for node in beginNodes do
3        NewMashlet(mashlet, node)
4        while node.prevNode do
5            if node.prevNode ≠ Merge do
6                MashletAddNode(mashlet, node.prevNode)
7                node ← node.prevNode
8            else do
9                beginNodes ← node.prevNode
10           end
11       end
12       MashletInit(mashlet)
13   end
```

5.2.2　调度器的设计与实现

调度器主要负责对 mashlet 的 PMT 进行实时计算和刷新，并根据当前的引擎容纳量，将适量的 mashlet 交付给引擎执行。

待执行 mashlet 的 PMT 计算时机是一个需要权衡的设计点。当 PMT 被频繁计算时，

可能获得更精确的值，但频繁计算带来了额外的 CPU 开销；而当 PMT 的刷新频率不高时，又可能因为计算的滞后导致 mashlet 的 PMT 没有随着实际变化而被更新，导致不准确。在实践中，刷新频率根据统计采用了一个经验性的数值。

Mashlet 调度器的工作算法如表 2 所示。

表 2 Scheduler 调度器调度 mashlet 的算法

Algorithm: Schedule Mashlets

Require: a set of mashlets to run.

Output: void

```
1   for mashlet in mashletSet do
2       CalculatePMT(mashlet)
3       AddToQueue(mashlet)
4   end
5   while queue ≠ NULL and execPool.isNotFull do
6       t ← pool.remainSize
7       for mashlet in queue do
8           UpdatePMT(mashlet)
9       end
10      execPool ← SelectMashlet(queue, t)
11  end
```

5.2.3 执行器的设计与实现

执行器负责执行 Mashup 应用中每一个操作子的具体数据操作。

内容整合平台为用户提供了一个操作子的集合，用户在创建 Mashup 应用时，可以自由地选择和组合这些操作子。每个操作子可以进行一定的配置，执行器根据用户提交的 Mashup 应用，解构其操作子序列并执行。平台提供的操作子有：

获取聚合源（Fetch Feed）

获取聚合源的操作子用于引入一个或多个 RSS/Atom 数据源。用户选取 Fetch Feed 操作子后需填写作为数据源的 RSS 地址，执行器在运行阶段中前往该地址获取实时聚

合源，并映射为内部数据格式。当用户填写多个 RSS 数据源地址时，执行器负责从多源获取数据，然后不做区分的将其合并为一组内部数据。

获取 XML 数据源（Fetch XML）

获取 XML 操作子用于引入 XML 格式的数据。在 Mashup 编辑器上，用户选用 Fetch XML 操作子后，同样需要进行配置，且该配置过程与用户具有多步的交互操作。用户首先给定 XML 文件的 URL，引擎的实时支持模块负责获取该 XML 文件，并将其结构和数据按层次展现。用户继续在该层次图上选取若干节点，指定它们与引擎内部数据结构的映射关系。编辑器将用户的选择和指定作为该操作子的属性，一并记入 Mashup 应用的流程文件。在执行器的运行阶段，执行器根据原始的 XML 文件和用户感兴趣的数据域，输出符合内部数据结构的数据。

获取网页数据源（Fetch HTML）

获取网页数据源操作子用于获取网页内容。网页是网络上最普遍、最常见的信息载体，将其作为数据源可以极大的提升平台的可用性和普遍性。但是网页是一种非结构化的数据，作为内容整合应用的输入源又具有一定的挑战。

通过对网页的分析我们注意到，一些网页的局部有一些结构化的特征。如，在搜狐的首页，左侧有一大块区域是新闻板块。在该板块中，一组新闻标题以行的形式排列，并均指向一个超链接。在这种形式表现的背后，该网页所对应的 HTML 代码上，也有一组在同一层次的标题标签。这种局部结构化的特征在网页中普遍存在。

当用户对这类网页数据感兴趣时，可以选用 Fetch HTML 操作子，并借助编辑器提供的内容框选工具，选择感兴趣的网页区域。该工具可以帮助分析 HTML 网页的组成，并抽取出一个 XPath，记录用户感兴趣的区域在 HTML 文档结构中的位置。在执行器的执行阶段，该 XPath 作为参数，协助执行器在获取 HTML 文档后抽取出用户感兴趣的区域数据作为数据源。

过滤（Filter）

过滤操作子用于对输入数据进行过滤和筛选操作。用户选用该操作子时需要指定一组筛选原则。其中，每一条筛选原则包含一个匹配关键字、匹配逻辑（包含或不包含，大于或小于等）。筛选原则之间又以“与”、“或”的关系耦合。执行器的执行阶段，根据所有规则进行数据的顺序匹配，并筛除不符合规则的数据。

排序（Sort）

排序操作子将输入数据按规则组进行排序。用户选用该操作子时指定一组排序规

则，每个规则指定排序关键字和升序/降序。执行器执行时按规则组的先后顺序作为优先级，依次对所有数据进行排序。

合并（Merge）

合并操作子用于将多个数据流合一成为新的数据流，是内容整合引擎的标志性操作子。由于引擎的内部数据具有统一的结构，所以来自网络上的异构数据源可以进行合并。一个合并操作子支持最多五个数据流的合并，更多的数据流合并可以用该类操作子的级联进行实现。此外，合并操作子提供一个参数标记用户是否需要去除来自不同数据源的重复类数据项。数据项间的相似性检测和去重即在此操作子上完成。

截取（Clip）

截取操作子用于当数据项过多时的裁剪。可配置参数为保留的项目数。截取操作子常常被组合在排序操作子或联合操作子之后，用以精选数据。

地理信息标注（GeoTag）

地理信息标注操作子用于对输入流中的文本进行地理位置检测。当检测到地理信息时，将地理信息附上数据项。本操作子同时提供一个参数标记用户是否需求将检测不到地理信息的数据项剔除。

此外，还有本地搜索（Local Search）、网络搜索（Search）、多媒体扩充（Media Enrich）、翻译（Translate）、重命名（Rename）等数据操作子。

执行器的实现应用了 Java 中的反射机制。Java 中的反射机制是指在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性。由于 Executor 执行器在运行前并不知道每一个操作子所具有的属性、配置、以及解析这些配置的方法，故需要在运行时环境中，用反射机制去动态地使用在运行池中各操作子的属性、方法等。

通过这样的方式，执行器对 mashlet 四元组中 oprSeq 所列的每一个操作子 opr 依次进行处理，并将最后的结果作为 output 输出。该 mashlet 将一直停留在内存中，直至该 mashlet 结束生命周期。Mashlet 的生命周期终结以 output 作为其它 mashlet 的输入流完成传递，或该 mashlet 的最终操作子为 End 作为标志。

执行器的算法描述如表 3 所示。

表 3　Executor 执行器执行 mashlet 的算法

Algorithm: Execute Mashlets

Require: a sequence representation of mashlet.

Output: mashup result.

```
1    for node in mashlet do
2        while node ≠ End do
3            if threadPoolGet(node.oprType) = NULL do
4                Reflect(node.oprType)
5            end
6            threadPoolRun(node)
7        end
8        MashupOut(mashlet)
9    end
```

5.3 实现效果演示

本节展示 Mashup 平台的实现效果和使用方法。本论文项目最终将通过一站式的平台为终端用户提供服务，平台项目名为 Mashroom，取 Mashup 工坊之意。其中三个层次划分在网站上也有不同的体现。网站提供一个基于 Flex 的 Mashup 编辑器，负责编辑层的功能；网站后端以 Tomcat 作为应用服务器，运行一个执行引擎，提供运行层的功能，负责 Mashup 应用的执行；同时，网站提供一个下载安装包，提供 Android 平台上 Mashup 表现客户端的下载。

在浏览网页前首先需在服务器端配置应用环境，包括 Tomcat 6.0 和 MySQL 5.1。之后输入首页网址，可看到项目首页。

图 11 Mashroom 网站主界面

其中，Mashup 编辑器的界面如图 12 所示。

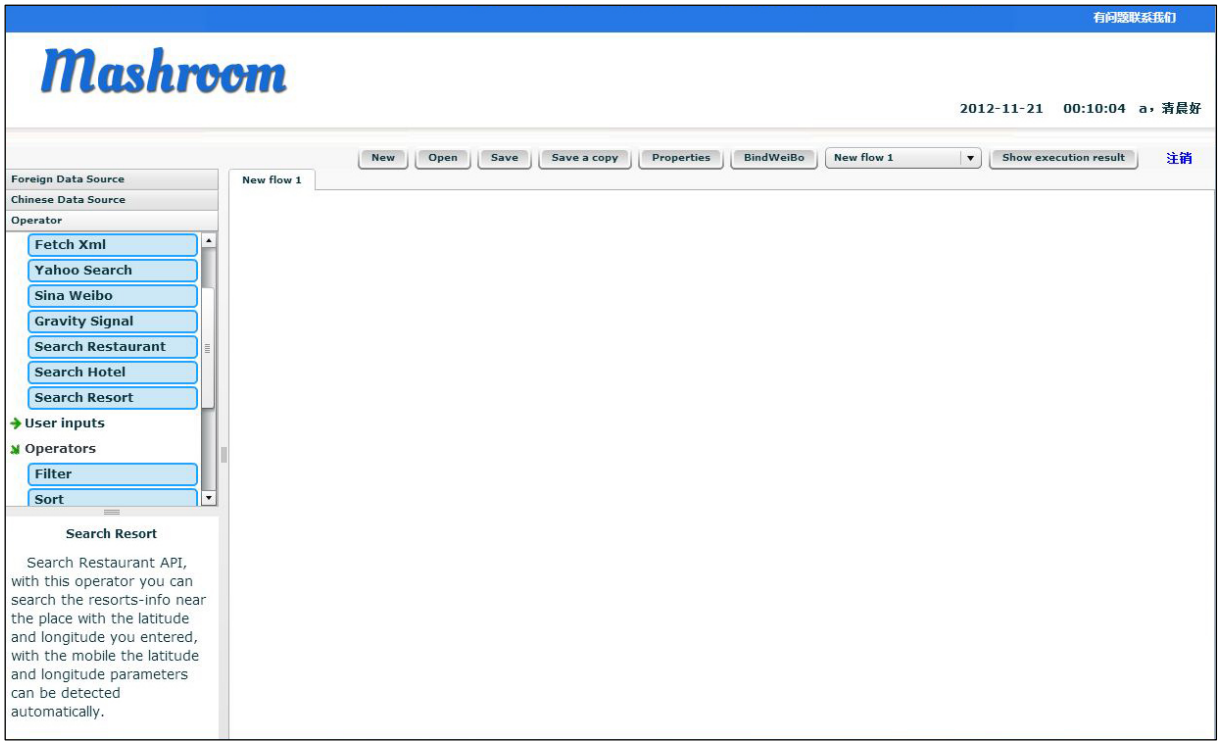

图 12 Mashroom 项目中的 Mashup 编辑器界面

在该界面中，左侧面板提供了可供用户选择的操作子和预配置的数据源，及其使用介绍。右侧空白面板为编辑 Mashup 的工作空间。用户可以从左侧面板中通过拖拽的方式选择模块到工作空间，如图 13 所示。

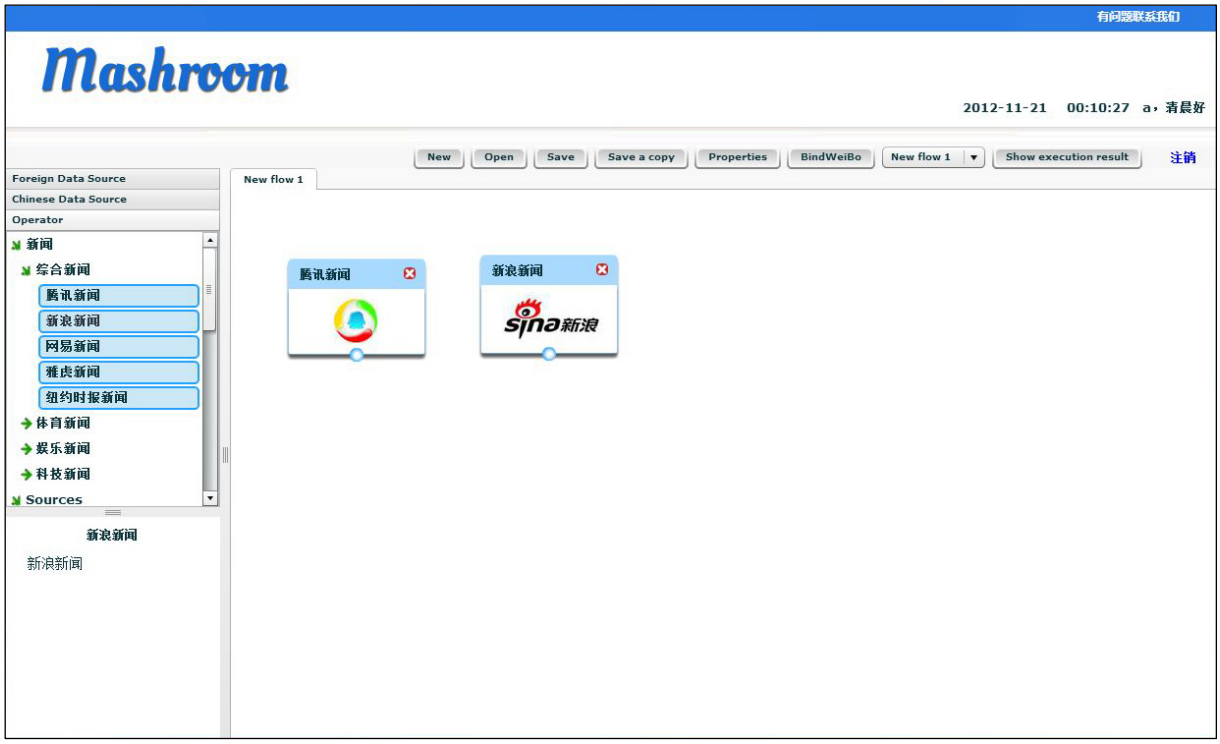

图 13　Mashup 编辑器界面中选择操作子步骤演示

每一个模块有可连接的输入口或输出口，用户通过连线的方式来指示模块间的数据流转关系，如图 14 中所示。

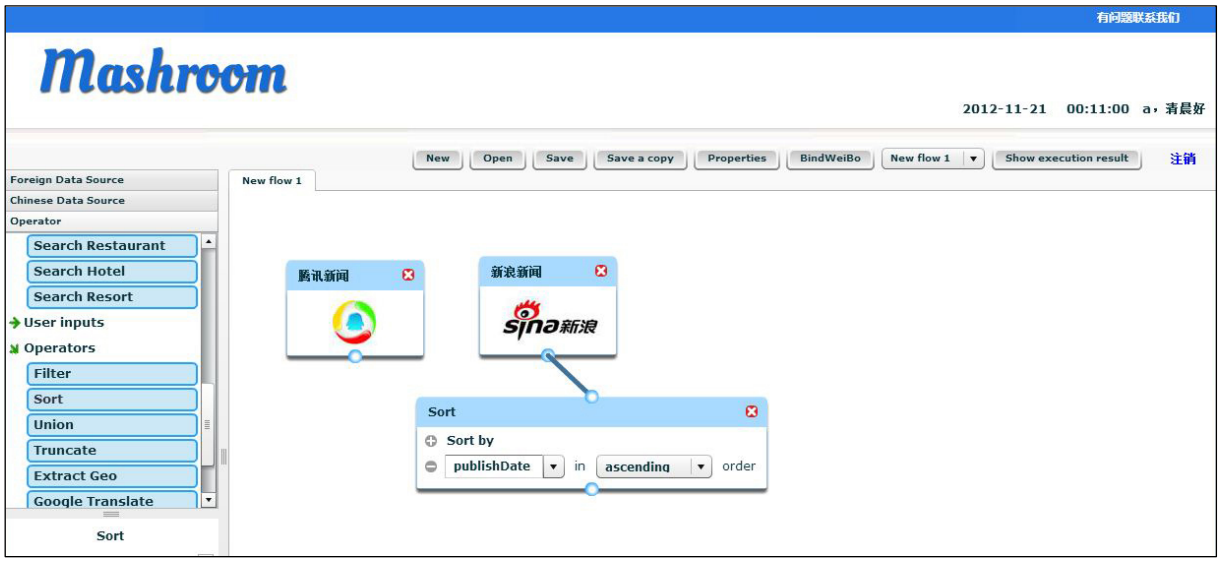

图 14　Mashup 编辑器界面中连接操作子步骤演示

通过用户一系列选择、连接操作子的操作，最后的 Mashup 应用形态可能如图 15。

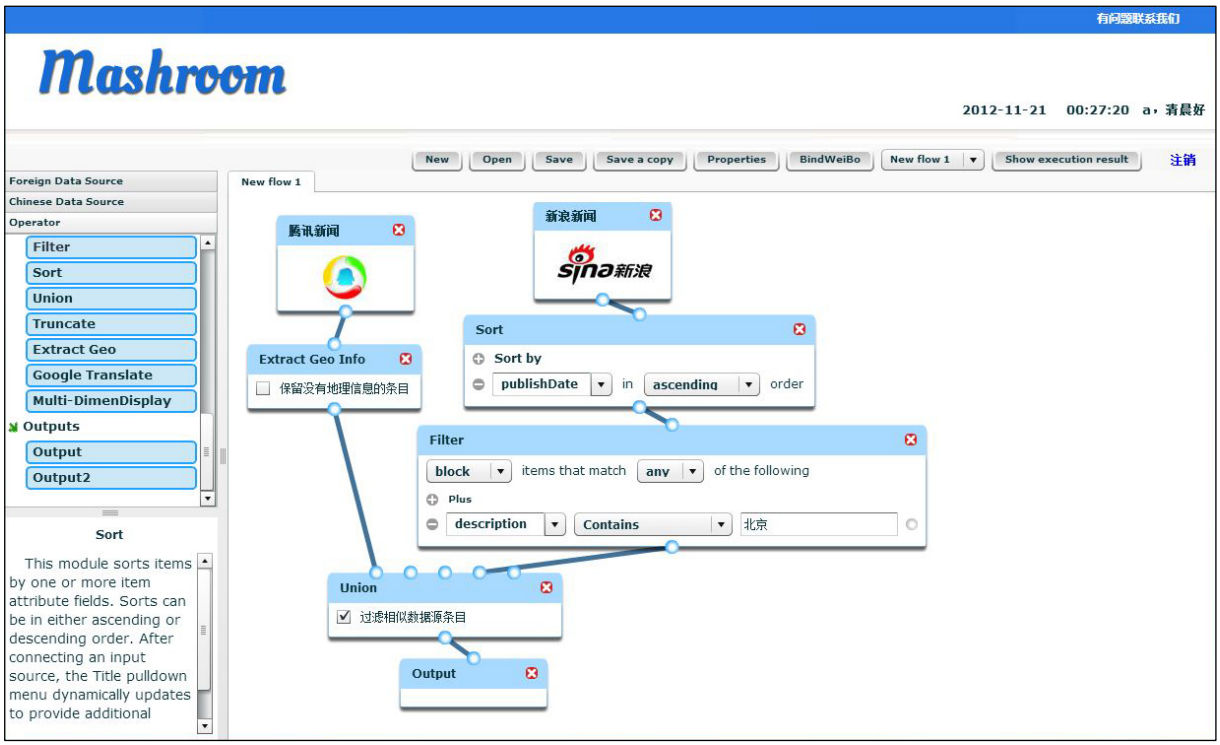

图 15 Mashup 编辑器界面中用户创建的 Mashup 示例

用户在完成以上步骤的编辑后，可以选择保存该 Mashup 应用以备以后的使用和手机端的即时信息获取，或者以调试模式查看该 Mashup 的运行结果。

Mashup 编辑器会将图形化界面中的对象序列化为流程文件保存或交付引擎。流程文件为 XML 的形式，其中片段如图 16 所示。

```
<?xml version="1.0" encoding="gb2312" ?>
- <em:enginemodel xsi:schemaLocation="http://www.example.org/EngineModel EngineModel.xsd htt
    output="30" xmlns:em="http://www.example.org/EngineModel" xmlns:gf="http://www.example.
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  + <em:figure gf:dynamic="0" gf:classid="FetchRss" gf:id="2">
  - <em:figure gf:dynamic="0" gf:classid="FetchHtml" gf:id="2">
      <gf:AttributeInput />
    - <gf:LogicalAttribute>
        <gf:url>http://www.sina.com.cn/</gf:url>
        <gf:xPath>/body[@id="body"]/div[@id="page"]/div[@id="PartA"]/div[@class="rightbox"]/
          [@id="news"]/span[@id="news_con_1"]/div[@id="newsListTop"]/ul[@id="impoLisTop"]
      </gf:LogicalAttribute>
    + <gf:interfaces>
    </em:figure>
  + <em:figure gf:dynamic="0" gf:classid="Filter" gf:id="6">
  + <em:figure gf:dynamic="0" gf:classid="Sort" gf:id="6">
  + <em:figure gf:dynamic="0" gf:classid="ExtractGeo" gf:id="10">
  + <em:figure gf:dynamic="0" gf:classid="Crop" gf:id="14">
  - <em:figure gf:dynamic="0" gf:classid="Merge" gf:id="19">
```

图 16 Mashup 流程片段文件示例

在调试模式下，执行引擎将 Mashup 应用的执行结果以引擎内部格式通过 XML 传回并做简单展示，如图 17 所示。

图 17　Mashup 应用调试模式下执行结果展示

在发布模式下，引擎通过 JSON 格式将结果传到手机端，用户通过手机客户端的软件完成身份认证后，可以拉取之前创建过的所有 Mashup 应用，并通过多种维度查看结果，如地图展示、列表展示等。手机应用效果如图 18 展示。从左到右四个界面分别是用户登录界面、执行结果单条目显示界面、执行结果所有条目的列表页和执行结果所有条目的地图展示页。

图 18　Mashup 手机端应用的展示

5.4 本章小结

本章介绍了高性能内容整合引擎的设计与实现。引擎的高性能主要通过之前两章中阐述的动态调度和静态优化两种方法实现。本章首先给出了系统的总体设计，之后描述了主要模块的功能设计和功能实现。最后通过一些系统环境中的平台运行效果，说明了高性能内容整合引擎的成功实现和实际运用。

第六章　实验与评价

本章通过一系列实验和测试数据，首先验证本论文所述引擎的有效性和可用性。其次，针对本论文所述的动态调度性能提升方法设计实验，通过对比分析，说明该方法的性能提升效果，最后分析了静态优化的效果。从而验证了论文的研究成果。

6.1　实验平台与工具

本论文项目作为 Web 工程部署在戴尔工作站的服务容器中，依赖 Windows 系统和 Java 环境。实验平台的其他运行环境配置如表 4 所示。

表 4　实验平台的运行环境配置

硬件环境	软件环境
Dell Precision T5100 工作站	Microsoft® Windows® 7
Intel® Core® i7 860 2.80GHz*4	Sun Java(TM)2 SE Version 6
3GB RAM	Apache Tomcat 6.0.16
10/100M 自适应无线以太网卡	Oracle® MySQL 5.1

本论文使用 Apache JMeter 作为实验的辅助工具，用于在短时间内发起批量的 Mashup 执行请求。

Apache JMeter 是 Apache 组织开发的基于 Java 的压力测试工具。它被设计用于对软件进行压力测试。所谓压力测试就是通过增加并发访问量来探知在什么条件下应用程序的性能会变得不可接受[50]。JMeter 可以用于测试静态和动态资源，例如 HTML 网页、Java 服务程序、CGI 脚本、Java 对象、数据库对象等。JMeter 还可以用于对服务器、网络或对象模拟巨大的负载，在不同压力类别下测试它们的强度和分析整体性能。本项目中正是应用了 JMeter 的这种功能，对 Mashup 执行请求的并发进行模拟。

JMeter 可以提供可视化反馈，通过各种图表或目录树的形式显示运行结果。在本章实验中，只是用到了 JMeter 的并发过程，其自带的结果展示对实验分析没有意义。本章实验和评价通过 Mashroom 项目的系统日志进行性能分析。

本论文设计和实现的日志系统记录了时间戳、执行耗时、相应完成时间、所占内存、PMT 值、调用模块等足够信息用于性能分析。本章所有评价基于该日志系统数据。日志的片段示例数据如下。

表 5 Mashroom 系统日志片段示例

时间戳	时移	类型	线程编号	调用类	功能	备注	PMT 或耗时
13:25:34	9719	INFO	[Thread-546]	(AbstractListModule.java:46)	Filter	t	7548
13:25:34	9719	INFO	[Thread-546]	(AbstractListModule.java:47)	Filter	m	4627
13:25:33	9558	INFO	[Thread-514]	(AbstractListModule.java:47)	Sort	m	9254
13:25:34	9724	INFO	[Thread-514]	(Mashlet.java:289)	Mashup ends	Mashup172	12058
13:25:33	9558	INFO	[Thread-239]	(AbstractListModule.java:47)	FetchRss	m	4627
13:25:33	9478	INFO	[Thread-458]	(AbstractListModule.java:46)	FetchRss	t	11591
13:25:34	9726	INFO	[Thread-458]	(AbstractListModule.java:47)	FetchRss	m	4627
13:25:33	9478	INFO	[Thread-421]	(Mashlet.java:289)	Mashup ends	Mashup141	11971
13:25:33	9436	INFO	[Thread-146]	(AbstractListModule.java:46)	FetchRss	t	11575
13:25:34	9726	INFO	[Thread-146]	(AbstractListModule.java:47)	FetchRss	m	4627
13:25:33	9676	INFO	[Thread-494]	(AbstractListModule.java:47)	FetchRss	m	4627
13:25:34	9739	INFO	[Thread-337]	(AbstractListModule.java:46)	Sort	t	370
13:25:34	9739	INFO	[Thread-337]	(AbstractListModule.java:47)	Sort	m	9254
13:25:34	9739	INFO	[Thread-337]	(Mashlet.java:289)	Mashup ends	Mashup119	12342
13:25:34	9811	INFO	[Thread-156]	(AbstractListModule.java:46)	FetchRss	t	11947
13:25:34	9811	INFO	[Thread-156]	(AbstractListModule.java:47)	FetchRss	m	4627
13:25:34	10032	INFO	[Thread-118]	(Mashlet.java:231)	Mashlet PMT	Mashlet116	56347606
13:25:34	10032	INFO	[Thread-118]	(Mashlet.java:231)	Mashlet PMT	Mashlet117	56347606
13:25:34	10040	INFO	[Thread-578]	(AbstractListModule.java:46)	FetchRss	t	12145
13:25:34	10040	INFO	[Thread-578]	(AbstractListModule.java:47)	FetchRss	m	4627
13:25:34	10032	INFO	[Thread-508]	(Mashlet.java:231)	Mashlet PMT	Mashlet506	56171780
13:25:34	10042	INFO	[Thread-508]	(Mashlet.java:231)	Mashlet PMT	Mashlet507	56171780
13:25:34	10085	INFO	[Thread-303]	(AbstractListModule.java:46)	Filter	t	7991
13:25:34	10085	INFO	[Thread-303]	(AbstractListModule.java:47)	Filter	m	4627
13:25:34	10032	INFO	[Thread-535]	(Mashlet.java:231)	Mashlet PMT	Mashlet533	56162526
13:25:34	10085	INFO	[Thread-535]	(Mashlet.java:231)	Mashlet PMT	Mashlet534	56157899
13:25:34	10097	INFO	[Thread-6]	(AbstractListModule.java:46)	FetchRss	t	12239
13:25:34	10097	INFO	[Thread-6]	(AbstractListModule.java:47)	FetchRss	m	4627
13:25:34	10099	INFO	[Thread-581]	(AbstractListModule.java:46)	FetchRss	t	12201

6.2 引擎的性能评估

该引擎部署在 Mashroom 平台以来，无故障连续运行时间超过 1000 多小时，具有较好的强壮型和鲁棒性。

实验首先选取了几个 Mashup 平台的用户应用测试引擎最终的整体性能，选用的 Mashup 如图 19 所示。这几个 Mashup 的选用具有一定的代表性，分别涉及了平台上的主要操作子，用以兼顾功能性测试。

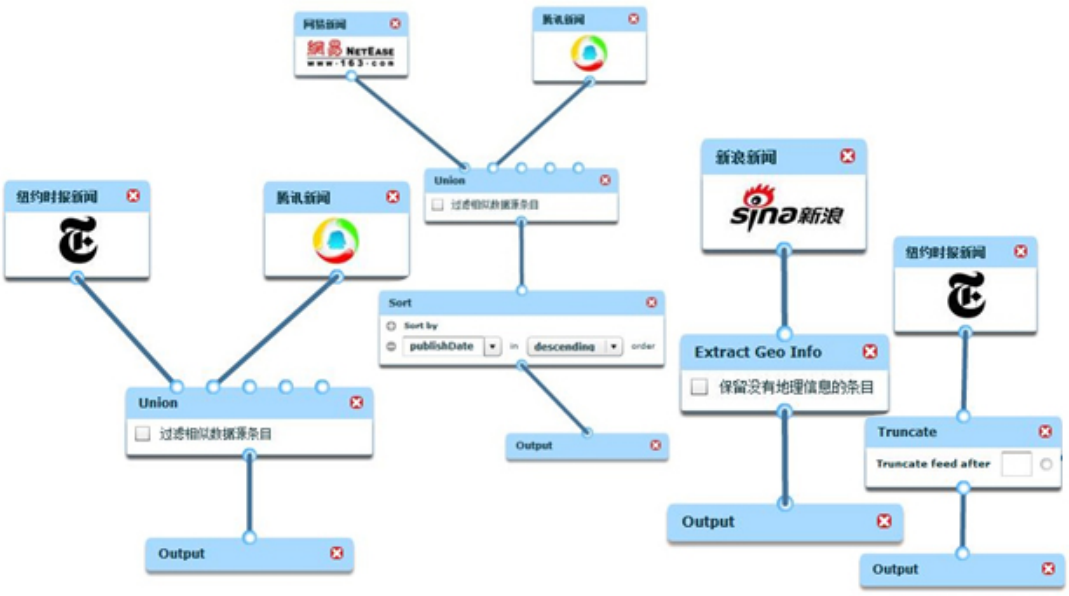

图 19　用于整体性能测试的 Mashup 应用

该四个 Mashup 应用的编号从左到右分别为 1 到 4。在实验中，JMeter 分别模拟了 10，100，200，一直到 1000 的并发量下，引擎的性能表现。结果如表 6 所示，表中，总耗时一项表示的是，在该并发量下，引擎从接到第一个执行请求起，到该并发量的最后一个执行请求完成的所耗时间。

从该表中可以看出，引擎在请求并发量较少时，具有极快的响应速度。当请求并发量多至 1000 时，引擎也保持在用户可接受的相应速度。

表 6　所选 Mashup 在引擎中的执行性能测试结果

并发量	总耗时（ms）			
	Mashup 1	Mashup 2	Mashup 3	Mashup 4
10	843	68	170	829
100	3236	1486	643	1447
200	2046	2114	723	2035
300	2847	2418	1279	2879
400	3229	2938	1659	3401
500	3348	3572	1945	4358
600	4245	4767	2270	4759
700	4669	4799	4780	5428
800	5509	5297	3904	6702
900	6624	6369	2823	7791
1000	7263	7051	3185	7876

6.3 动态调度方法的实验与效果评价

在 Mashroom 的高性能引擎实现中，应用了基于“懒启动”的动态调度策略，在用户 Mashup 执行并发请求时进行响应和资源分配，该策略对引擎的性能提升效果明显。本节将通过一组对比实验，展示该策略对引擎性能的影响效果。

由于该方法主要作用于 Mashup 的执行调度上，与 Mashup 本身的结构无关。为了让结果更加清晰和突出，本实验使用图 20 所示的 Mashup 应用进行高并发模拟。该 Mashup 从腾讯网和新浪网获取数据源，对其做若干操作。该 Mashup 会被分为 3 个 mashlet 进行处理。在下文计算 PMT 的过程中，内存尺寸的单位为字节，计时以毫秒。

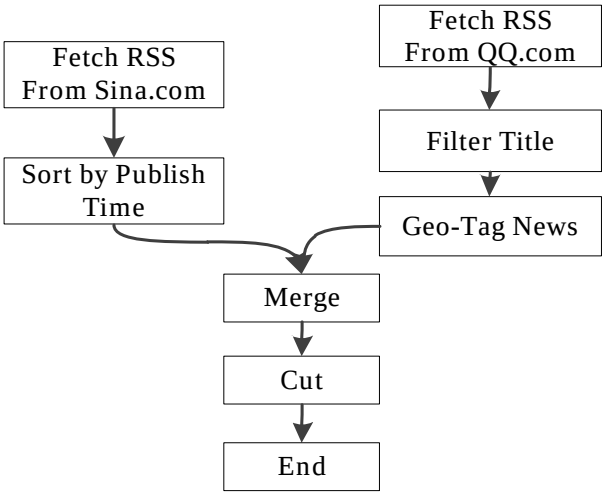

图 20 动态调度效果实验所用的 Mashup 应用

本组实验设置三个对比组，标识及意义分别是：

1. FIFO：按照先进先出的顺序调度 Mashup 应用的执行。在该种方式下，Mashup 引擎不进行调度工作，只尽量的处理当前的 Mashup 请求。
2. Max-First：完全以 PMT_{run} 的估值作为调度依据，进入 Mashlet Queue 后不作动态调整，即不加修正的基于 PMT 方法。
3. Lazy Start：本论文所述的完整的基于懒启动的调度方法。

在第一组实验中，目标在于使用 PMT 的指标来评测内存的利用率。实验过程是使用 JMeter 并发 200 次请求，并记录相应 mashlet 的 PMT。结果图 21 所示。该图的横坐标表示 mashlet 的运行时序，纵坐标表示对应 mashlet 的 PMT 值。故而对曲线下的面积积分即表示各对比组的设置下，200 次执行请求的累积 PMT。图中可以显著看出采用懒启动的动态调度方式下，所用 PMT 最少。这是因为 mashlet 被异步的调度启

动，更加有序的调用减少了阻塞时间。数据显示，PMT 累积值减少了 28.65%。

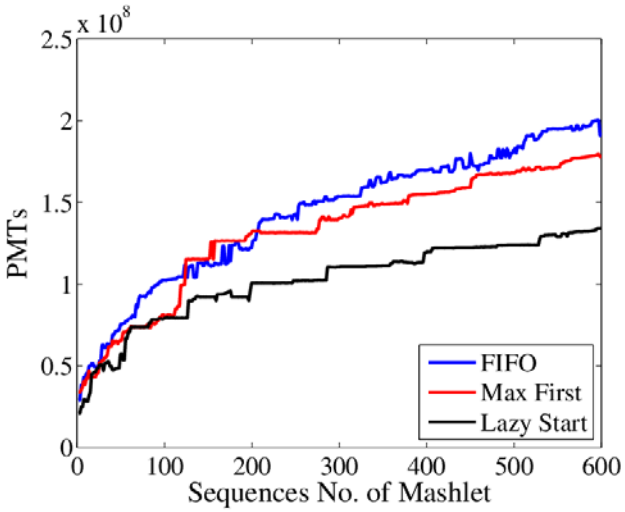

图 21　并发量为 200 时 mashlet 的 PMT 对比

同时，在引入懒启动动态调度之后 Mashup 执行时间的变化也值得注意。在图 22 中，横坐标是 Mashup 执行结束的时序号，纵坐标是耗时。可以看到懒启动的效果对 Mashup 执行耗时效果明显，数据显示，Mashup 执行的平均耗时减少了 28.53%。这是因为 PMT 计算方式的相互影响带来了聚集作用，可以让来自同一个 Mashup 的 mashlet 临近执行，从而加速了单个 Mashup 的执行，提升了单位时间的引擎吞吐量。

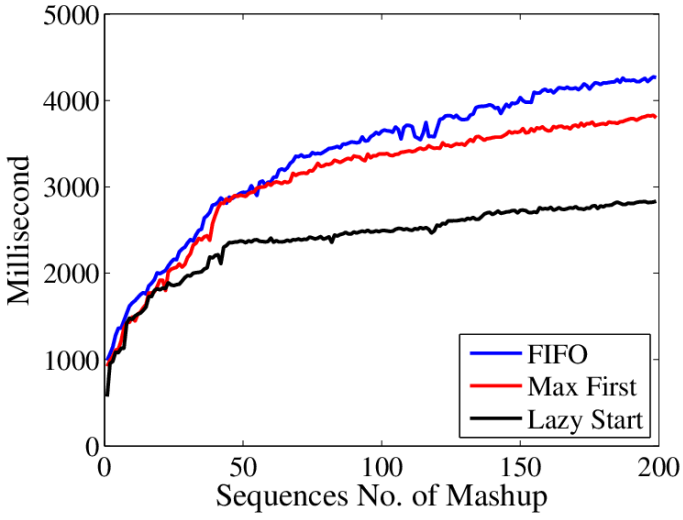

图 22　并发量为 200 时 Mashup 执行耗时对比

在第二组实验中，着重评估懒启动动态调度在不同并发量情况下对性能的影响。实验中，设置了从 50 到 300 不等的并发量请求数。在图 23 和图 24 所示的结果中，

可以看到作用于 mashlet 的 PMT 和 Mashup 的执行时间上的减少效果均与并发量正相关。即并发量越大，性能提升效果越明显，提升效果在 12%到 44%之间。这是因为由于调度策略的的职能所决定的，并发量越高，对于有效、合理的调度策略就越有需求。

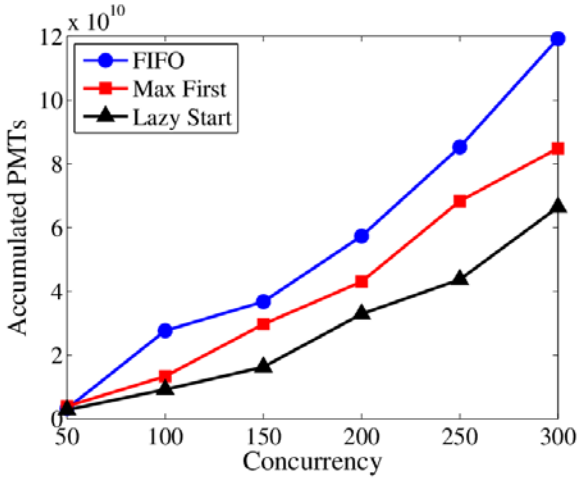

图 23 不同并发量下 mashlet 累计消耗 PMT 对比

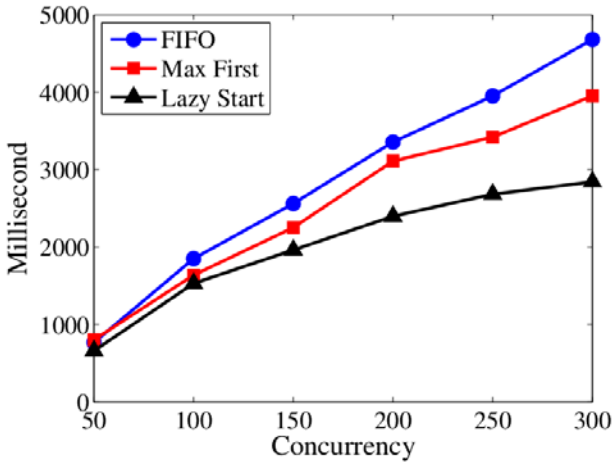

图 24 不同并发量下的 Mashup 执行耗时对比

6.4 静态优化方法的实验与效果评价

在 Mashroom 的高性能引擎实现中，应用了静态优化策略对待执行的 Mashup 应用进行组织和结构上的静态优化。应用该策略后对 Mashup 应用的效率提升效果明显。本节将通过一组对比实验，就静态优化对执行性能的影响进行评估。

由于静态优化的有效性并非对所有 Mashup 应用都能明显体现。为了让结果更加清晰和突出，本实验使用图 25 所示的 Mashup 应用进行评估，以测试静态优化前后的效果。

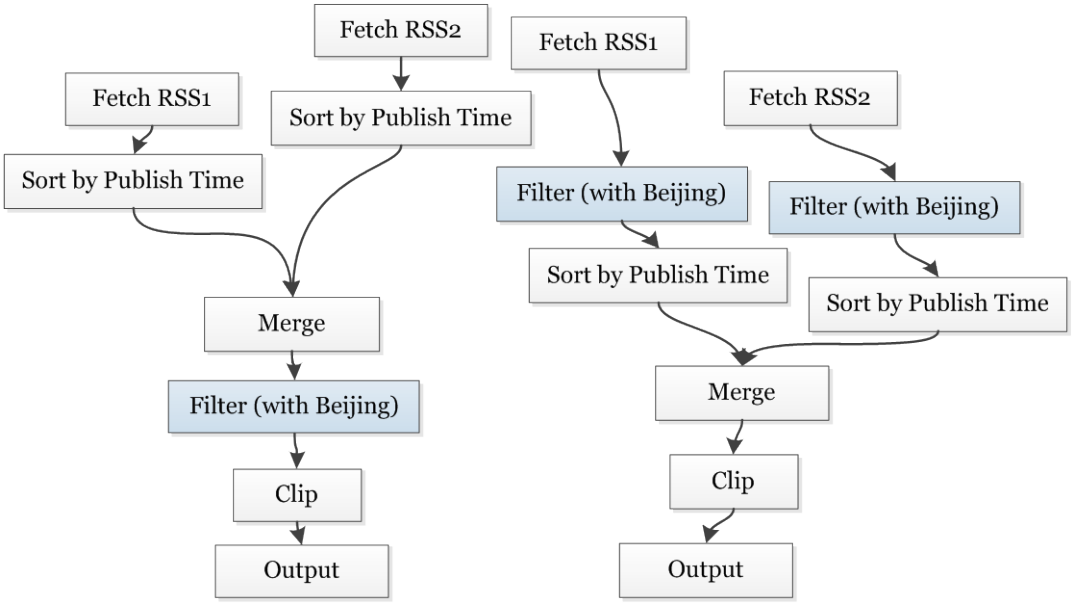

图 25　静态优化实验所用 Mashup 应用

通过重复实验后取平均值的对比，发现静态优化对于内存占用的减少效果明显。上例中，mashlet 的 PMT 占用对比如图 26 所示。

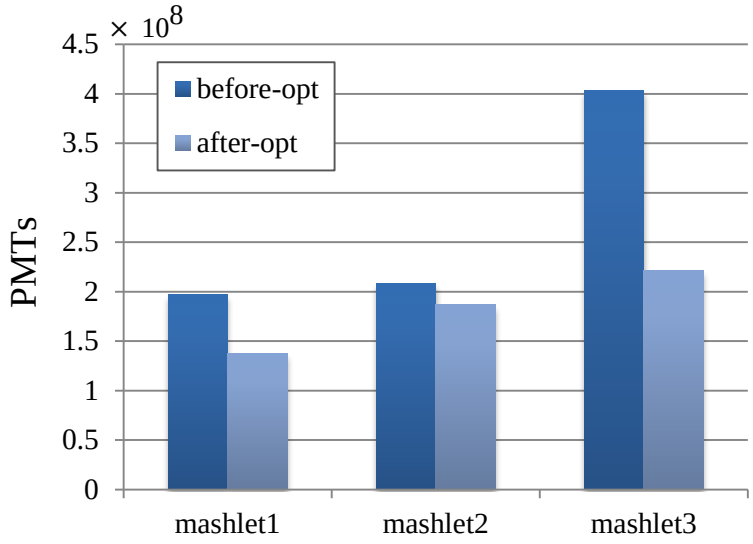

图 26　静态优化通过 PMT 值反应的效果

在图 26 中所示中，深色柱和浅色柱分别表示静态优化前后的 PMT 占用值。优化前后的 Mashup 均将剪枝成三个 mashlet，在图上分别进行对比。第一个 mashlet 中，实际上过滤器对数据项起了较大的过滤筛选作用，将中间结果所占内存的均值从 4603K 减少到 1003K，但是由于过滤功能也耗费了一些时间，故在 PMT 这种综合考虑时间和内存的衡量值上表现的不太明显；第二个 mashlet 中，虽然有做优化，但是筛

除数据不多故在柱状图中无明显体现。在第三个 mashlet 中，由于流入数据量的减少，以及在处理过程中不再进行过滤而节省的耗时，故而在 PMT 的柱状图上看出静态优化具有十分明显的效果。

实验表明，静态优化的引入，对于单个可优化的 Mashup 执行效率及对引擎内存的占用优化有良好的效果。而文献[22]通过对 Yahoo! Pipes 平台上的 Mashup 应用做海量分析和统计后指出，很大一部分终端用户编写的 Mashup 应用可进行操作子顺序组织上的优化，该类 Mashup 应用占 Yahoo! Pipes 应用资源库总量的 19%。由此可见，静态优化在 Mashup 平台上的应用效果也会十分可观。

6.5 本章小结

本章主要介绍了实验和效果评价，用实际数据来验证本文所研究的高性能 Mashup 执行引擎的效果。本章首先介绍了实验环境设定、实验工具和实验数据分析方法，之后分别通过一些图表，直观地展示了实验结果，并对其进行一定的分析和论证。通过实验数据说明了动态调度和静态调整两种方法对于性能优化的意义，验证了论文的研究成果。

总结与展望

论文总结

SOA 的出现，代表着软件由以产品为中心转向以消费者和用户为中心的模式。它通过将企业信息系统组件封装并发布为标准 Web 服务的形式，打破了信息化建设历程中诸多异构系统间的数据、模型的独立性，从而成为下一代 Web 服务的基础框架。

在 SOA 技术相对成熟、网络化软件应用高速发展的今天，以 Web Service 为代表的软件技术已成为新的研究热点。Web Service 技术极大地降低了应用之间的耦合度，增加了企业应用的灵活性，在不断的发展和成熟中逐渐成为了 SOA 的最佳实践方式。

传统的 Web Service 以 SOAP 协议为基础，以 WSDL 文档描述，通过 UDDI 向注册中心定义接口，从而对外提供服务。传统 Web Service 组合方式适用于企业级应用开发，但由于其复杂的特性设计和高深的技术实现使得面向非产品级的个人应用仍然停留在初级阶段。而一种遵循面向资源的体系结构的新风格：REST，正以其轻量化、无状态、基于 HTTP 等优势逐渐取代传统的 Web Service。REST 方式给出了设计与开发网络应用的新型解决方案，它降低了开发的复杂性，提高了系统的可伸缩性。

在这种背景下，一种新型的基于 Web 的数据集成应用程序开发方法 Mashup 逐渐兴起，取代了传统的软件开发模式，并已经成为 Web 2.0 的标志性技术之一。Mashup 提倡以普通使用者为中心，通过将不同来源的数据与服务进行组合，构建个性化的网络应用程序。它专注于没有编程基础的终端用户，致力于帮助该类用户构建小型业务、处理数据、使用资源，满足快速响应、即时构建、个性化定制等非功能属性需求。

各大厂商纷纷推出各自的 Mashup 平台或产品作为产业化的实践。在 Mashup 平台中，最核心的部分为 Mashup 执行引擎。引擎性能的高低直接决定了 Mashup 平台的吞吐率，影响 Mashup 平台的系统负载。本论文致力于研究与实现一个高性能的 Mashup 执行引擎。在本论文的系统工作中，首先实现了一个 Mashup 执行引擎，然后通过对其执行过程的研究与分析，提出一种基于“懒启动”的 Mashup 执行调度策略，该策略通过在运行时动态地计算优先级权重，合理地调度 Mashup 的执行。在此基础之上，本文还提出了一种静态优化的方法，优化 Mashup 的组织结构。最后，将此两种性能提升的方法在引擎中应用，通过实验验证，具有较好效果。

本文的主要工作包括：

1. 通过研究 Mashup 的产业化发展过程与学术界对于 Mashup 性能问题的探讨，总结出了 Mashup 的一般性模型及其形式化定义。针对 Mashup 的结构在调度上需要进行粒度划分的问题，创新性的提出了 Mashup 的调度单元：mashlet，同时给出了 mashlet 的属性定义。
2. 针对 Mashup 应用平台的高性能需求，通过分析 Mashup 应用的结构特点和运行特点，创新性地提出了一种基于“懒启动”的动态调度策略，在 Mashup 引擎的执行过程中动态地、有序地对待执行 Mashup 请求进行调度，通过节约引擎瓶颈资源的占用，提升引擎的吞吐量，为内容整合引擎的性能提升提供了可依赖的指导方法。
3. 针对 Mashup 平台面向终端用户的定位而带来的低效编程，基于相关论文的研究，提出了对 Mashup 应用片段的静态优化策略，制定了一组静态优化的规则，优化了 Mashup 应用片段的执行效率，作为对内容整合引擎性能提升的补充方法。
4. 基于以上研究，设计并实现了高性能内容整合引擎。通过系统设计，将引擎分为几个关联的功能模块，结合了动态和静态两种优化方法，最终实现了引擎，应用于实际中的生产环境，作为主项目的一部分，满足了功能需求和非功能需求，并针对所实现的高性能内容整合引擎，通过实验设计，验证了功能属性，以及动态优化和静态优化的实际效果。实验效果证明了所研究的性能优化策略切实有效。

工作展望

本文设计实现的内容整合引擎，能够满足普通用户对 Mashup 应用的开发和使用，也能满足一定负载量下的性能需求。但研究中仍存在一些不足和需要完善之处，进一步的工作可以从以下几个方面进行考虑。

1. 在应用模式探索方面，可以考虑实现一种以用户为中心的应用协同工作的框架[51]。考虑到 Mashup 应用对服务和内容的整合只是用于满足用户简单的业务整合功能，一些操作子均由平台自身提供，对数据和服务做一些简单的操作。在今后的工作中，可以考虑一种具有内在适配性的协同框架，由用户任意的选用互联网上的应用，由首位用户进行数据异构的转化。通过这种方式，该框架可以动态的组合互联网上的应用及其调用，更加简便地为用户提供组合服务的功能。在用户的使用过程中，应用的数据、服务的内容也可以考虑支持用户间的协同操作。
2. 在性能方面，一个可以持续研究的方向在于对引擎的去中心化和引入分布式[52]。

在云计算高速发展的今天，分布式的部署对系统的可靠性、扩展性等有着十分重要的战略意义。而对于 Mashup 来说，由于它的运行关键在于对网络上已有服务和信息的整合，故分布式带来的性能优势更加明显，同时问题也更加复杂。如由于各种操作子对性能的要求不同，可以考虑由分布式的机器各自专注处理一种操作子，该机器从配置上就可进行针对该操作子进行专有优化；如网络上各 API 的分布和服务器的地缘位置不同影响速度，也可以考虑通过分布式的部署地缘进行消除影响。

3. 在性能方面，还可以在静态优化的基础之上再做进一步的研究工作。在静态优化的过程中，个性化的 Mashup 应用已经被规整为相对统一的内部格式。这样的规整有助于引擎判断在某个时刻，或一段时间内是否有一组相同的 Mashup 执行请求。如，来自两个用户都有从搜狐获取新闻源并提取其中地理信息这样一个操作步骤。当引擎判断到这类短时间内相同的执行请求时，可以将其合并只执行一次。通过这样的方法，可以大幅减低引擎的执行总次数，提升吞吐率。

参考文献

[1] YV Natis. Service-oriented architecture scenario[EB/OL]. http://info.lnpu.edu.cn/website/jpkc/rjgc/ywzl/Service-Oriented%20Architecture%20Scenario.pdf, 2012-11-21

[2] Schroth, Christoph, and Till Janner. Web 2.0 and soa: Converging concepts enabling the internet of services[J]. IT Professional 9, no. 3, 2007: 36-41

[3] Wikipedia, Web 2.0[EB/OL]. http://en.wikipedia.org/wiki/Web_2.0, 2012-11-21

[4] OReilly, Tim. What is Web 2.0: Design patterns and business models for the next generation of software[J]. Communications & strategies, 1, 2007: 17

[5] Bianchini Devis, Valeria De Antonellis, and Michele Melchiori. Towards Semantic-Assisted Web Mashup Generation[A]. Database and Expert Systems Applications, 2012, 23rd International Workshop[C], IEEE, 2012: 279-283

[6] Wikipedia, Mashup[EB/OL]. http://en.wikipedia.org/wiki/Mashup, 2012-11-21

[7] Hoyer, Volker, Katarina Stanoesvka-Slabeva, et al. Enterprise mashups: Design principles towards the long tail of user needs[A]. SCC'08 IEEE International Conference[C]. vol. 2, 2008: 601-602

[8] Yahoo Inc, Yahoo! Pipes[EB/OL]. http://pipes.yahoo.com/pipes/, 2012-11-20

[9] Nestler, Tobias. Towards a mashup-driven end-user programming of SOA-based applications[A]. Proceedings of the 10th international Conference on information integration and Web-Based Applications & Services[C]. ACM, 2008: 551-554

[10] Walsh, Norman. Xproc: An xml pipeline language[J]. Conference on XML, 2007: 13

[11] Guiling Wang, Shaohua Yang, and Yanbo Han. Mashroom: end-user mashup programming using nested tables[A]. Proceedings of the 18th international conference on World Wide Web[C], ACM, 2009: 861-870

[12] Ennals, Rob, and David Gay. User-friendly functional programming for web mashups[A]. ACM SIGPLAN Notices[C]. vol. 42, no. 9, ACM, 2007: 223-234

[13] Liu, Xuanzhe, Yi Hui, Wei Sun, et al. Towards service composition based on mashup[A]. Services 2007 IEEE Congress[A]. IEEE, 2007: 332-339

[14] Ke Xu, Xiaoqi Zhang, Meina Song, and Junde Song. Mobile mashup: Architectu

re, challenges and suggestions[A]. Management and Service Science, 2009 International Conference[C]. IEEE, 2009: 1-4

[15] Luo, Xiaoxiang, Huiyang Xu, Meina Song, et al. Research on SOA-based platform to enable mobile mashups[A]. Communication Technology, 11th IEEE International Conference[C]. IEEE, 2008: 607-610

[16] ProgrammableWeb[EB/OL]. http://www.programmableweb.com/, 2012-11-20

[17] Yu, Jin, Boualem Benatallah, Fabio Casati, et al. Understanding mashup development[A]. Internet Computing, IEEE 12[C]. 2008: 44-52

[18] Intel Inc, Intel Mash Maker[EB/OL]. http://software.intel.com/en-us/articles/intel-mash-maker-mashups-for-the-masses, 2012-11-20

[19] Ennals, Rob, Eric Brewer, Minos Garofalakis, et al. Intel Mash Maker: join the web[A]. ACM SIGMOD[C]. Record 36, no. 4, 2007: 27-33

[20] Hassan, OA-H., Lakshmish Ramaswamy, et al. Mace: A dynamic caching framework for mashups[A]. ICWS 2009. IEEE International Conference[C]. IEEE, 2009: 75-82

[21] Stecca, Michele, and Massimo Maresca. An execution platform for event driven mashups[A]. Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services[C]. ACM, 2009: 33-40

[22] Stolee, Kathryn T., and Sebastian Elbaum. Refactoring pipe-like mashups for end-user programmers[A]. Proceedings of the 33rd International Conference on Software Engineering[C]. ACM, 2011: 81-90

[23] Hassan, OA-H., Lakshmish Ramaswamy, and John A. Miller. Enhancing Scalability and Performance of Mashups Through Merging and Operator Reordering[A]. Web Services, 2010 IEEE International Conference[C]. IEEE, 2010: 171-178

[24] Miah, Shah J., and John Gammack. A mashup architecture for web end-user application designs[A]. Digital Ecosystems and Technologies, 2008, 2nd IEEE International Conference[C]. IEEE, 2008: 532-537

[25] 路跃. 面向最终用户的即时信息整合平台的研究与实现[D]. 北京航空航天大学, 2011

[26] Yang, Jianyu, Jun Han, Xu Wang, et al. MashStudio: An On-the-fly Environmen

t for Rapid Mashup Development[A]. Internet and Distributed Computing Systems[C]. Springer, 2012: 160-173

[27] Cornell, Gary, and Cay S. Horstmann. Core Java[M]. Prentice-Hall, Inc., 1997

[28] PureMVC Framework[EB/OL]. http://puremvc.org/, 2012-11-20

[29] Daniel, Florian, Maristella Matera, Jin Yu, et al. Understanding UI integration: A survey of problems, technologies, and opportunities[J]. Internet Computing, IEEE 11, no. 3, 2007: 59-66

[30] Chaisatien, Prach, and Takehiro Tokuda. A web-based mashup tool for information integration and delivery to mobile devices[J]. Web Engineering, Springer, 2009: 489-492

[31] Wikipedia, Web feed[EB/OL]. http://en.wikipedia.org/wiki/Web_feed, 2012-11-21

[32] Wikipedia, Open API[EB/OL]. http://en.wikipedia.org/wiki/Open_API, 2012-11-21

[33] Ebner, Hannes, and Matthias Palmér. A mashup-friendly resource and metadata management framework[A]. Proceedings of the First International Workshop on Mashup Personal Learning Environments. 2008: 48-56

[34] Löh, Andres, and Ralf Hinze. Open data types and open functions[A]. Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming[C]. ACM, 2006: 133-144

[35] Cao, Jiawei, and Chunxiao Xing. Data Source Recommendation for Building Mashup Applications[A]. Web Information Systems and Applications Conference[C]. IEEE, 2010: 220-224

[36] Chen, Huajun, Bin Lu, Yuan Ni, et al. Mashup by surfing a web of data APIs [A]. Proceedings of the VLDB Endowment 2[C]. ACM, no. 2, 2009: 1602-1605

[37] Bianchini, Devis, Valeria De Antonellis, and Michele Melchiori. A recommendation system for semantic mashup design[A]. In Database and Expert Systems Applications (DEXA), 2010 Workshop[C]. IEEE, 2010: 159-163

[38] Elmeleegy, Hazem, Anca Ivan, et al. Mashup advisor: A recommendation tool for mashup development[A]. Web Services 2008, IEEE International Conference [C]. IEEE, 2008:337-344

[39] Maaradji, Abderrahmane, Hakim Hacid, et al. Social composer: a social-aware mashup creation environment[A]. Proceedings of the ACM Conference on Computer Supported Cooperative Work[C]. 2010

[40] Lu Bin, Zhaohui Wu, Yuan Ni, et al. sMash: semantic-based mashup navigation for data API network[A]. Proceedings of the 18th international conference on World Wide Web[C]. ACM, 2009: 1133-1134

[41] Benslimane Djamal, Schahram Dustdar, and Amit Sheth. Services mashups: The new generation of web applications[A]. Internet Computing[C]. IEEE 12, no. 5 2008: 13-15.

[42] Curbera Francisco, Frank Leymann, Tony Storey, et al. Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more[M]. Prentice Hall PTR, 2005

[43] Curbera Francisco, Matthew Duftler, Rania Khalaf, et al. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI[A]. Internet Computing, IEEE 6[C]. IEEE, no. 2, 2002: 86-93

[44] Richardson, Leonard, and Sam Ruby. RESTful Web Services[M]. O'Reilly Media, Incorporated, 2007

[45] Lathem Jon, Karthik Gomadam, and Amit P. Sheth. Sa-rest and (s) mashups: Adding semantics to restful services[A]. Semantic Computing, 2007, International Conference[C]. IEEE, 2007: 469-476

[46] 欧振猛, 余顺争. 中文分词算法在搜索引擎应用中的研究[J]. 计算机工程与应用, 36, no. 8, 2000: 80-82

[47] 吴栋, 滕育平. 中文信息检索引擎中的分词与检索技术[J]. 计算机应用, 24, no. 7 2004: 128-131.

[48] Wang Canhui, Min Zhang, Shaoping Ma, et al. Automatic online news issue construction in web environment[A]. Proceedings of the 17th international conference on World Wide Web[C]. ACM, 2008: 457-466

[49] 汤小丹, 梁红兵, 哲凤屏等. 计算机操作系统[M]. 西安电子科技大学出版社, 2007

[50] 罗婧婷, 赵轶群, 郑小军等. 开放源 Web 应用开发中的一种测试解决方案[J]. 计

算机与现代化，2005：25-28，32

[51] Vasko Martin and Schahram Dustdar. Introducing Collaborative Service Mashup Design[A]. Lightweight Integration on the Web (ComposableWeb'09)[C], 2009: 51-62

[52] Meng, Jian, and Jinlong Chen. A Mashup Model for Distributed Data Integratio n[A]. Management of e-Commerce and e-Government, ICMECG'09, International Conference[C]. IEEE, 2009: 168-171

攻读硕士学位期间取得的学术成果

[1] 徐静波，孙海龙，刘旭东，王旭，张日崇. Optimizing Pipe-like Mashup Execution for Improving Resource Utilization [A]，IEEE，SOCA 2012[C]. 2012，Taipei（已录用）

[2] 孙海龙，刘旭东，徐静波，王旭. 内容整合引擎调度方法及装置[P]. 中华人民共和国专利申请（已受理，专利号 201210337027.0）

[3] 徐静波，孙海龙，刘旭东，王旭，张日崇. A High Performance Schedule Design of Mashup Runtime Based on Lazy Start [J]，北京航空航天大学第九届研究生学术论坛论文集. 2012

致　谢

在此论文即将完成的时候，我想对我的导师刘旭东教授表示衷心的感谢。在硕士学习的两年半生涯中，刘老师极大的科研热情和对学生尽心尽责的态度深深感染了我。在刘老师的精心指导和悉心培养下，我在科研工作上加深了对专业知识和研究方法的理解和掌握，在我进行课题研究的过程中，刘老师时刻关注着研究进展，在关键问题和方法上提出指导和建议，使我能够顺利的完成研究课题。

非常感谢孙海龙副教授。在我就读研究生期间，孙老师在学习、生活上给予我莫大帮助，也给予我足多的信任和支持。孙老师的指导贯穿着我的科研工作始终，也是在孙老师的带领和鼓舞下，我才能在顺利地选取课题的研究点，并在该研究点上攻坚克难。在我撰写小论文期间，孙老师一直对我的工作持续关注、倾力指导，甚至帮我审阅论文到深夜，令我深受感动。

特别感谢怀进鹏院士、马殿富教授，他们以渊博的学识、严谨的治学作风，准确地把握当前研究动态和发展趋势，为实验室创造了积极活跃的学术讨论氛围和良好的研究学习环境。感谢张日崇老师、沃天宇老师给我的帮助。同时感谢 ACT 实验室的教师团队为我们实验室营造了良好的科研工作氛围，两年半的实验室生活让我受益匪浅。

感谢王旭博士在整个毕设过程中对我的帮助，不仅为我的研究课题提供了宏观上的指导建议，同时认真而耐心地帮助我解决实际设计和实现中碰到的种种困难和问题，对我提出了很多宝贵的建议。感谢路跃师兄、杨建宇、张芊、周子龙，你们在研究课题和系统工作上给予了我很大的帮助。

感谢周义、陆阳、毛泽鹏、孟子德，我们在学习生活中共同探讨问题、共同钻研，互相帮助，结成了深厚的友谊。感谢张萌、张帆、闫敏之、赵文敏、杨蕾、曾志兴、郭大路、张世龙，以及所有的 ACT 实验室成员，大家共同营造了良好的学术氛围和愉快的生活环境。

最后感谢各位评阅老师抽出宝贵的时间对本文进行审评。