软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(7):1419−1435 [doi: 10.3724/SP.J.1001.2013.04319]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种面向富客户端应用的运行时自适应中间件*

赵 祺[1,2], 刘譞哲[1,2], 王旭东[1,2], 黄 罡[1,2], 梅 宏[1,2]

[1](北京大学 信息科学技术学院 软件研究所,北京 100871)

[2](高可信软件技术教育部重点实验室(北京大学),北京 100871)

通讯作者: 刘譞哲, E-mail: liuxzh@sei.pku.edu.cn , http://www.sei.pku.edu.cn/~liuxzh

摘 要: 随着 Internet 的发展、应用需求的日趋复杂,传统浏览器-服务器模式下的瘦客户端不再能够满足应用这种需求,进而促使了具有良好用户体验、可以有效利用本地存储计算资源的富客户端应用的出现.富客户端应用遵循“模型-视图-控制器(model-view-controller,简称 MVC)”体系结构风格,运行在客户端的软、硬件运行环境中.随着移动设备硬件、浏览器软件的发展,不同富客户端运行环境能力差异很大.另一方面,因为 Internet 的开放性、动态性,富客户端应用开发人员不可能预知其运行环境的特点.因此,富客户端不可避免地面临异构运行环境造成的适应性问题.提出一种富客户端运行环境自适应中间件,可提供一个符合富客户端应用体系结构风格的 MVC 构件模型,并利用构件的数据模型、控制器与视图这 3 部分,有针对性地处理富客户端存储环境、计算环境以及显示环境中的适应性问题,提供相应的自适应解决方案.提出的自适应中间件封装了以上构件模型与自适应机制,保证运行于中间件上的富客户端可以适应运行环境,合理、高效地利用运行环境中的存储、计算以及显示资源.

关键词: 富客户端;运行时自适应

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 赵祺,刘譞哲,王旭东,黄罡,梅宏.一种面向富客户端应用的运行时自适应中间件.软件学报,2013,24(7):1419−1435. http://www.jos.org.cn/1000-9825/4319.htm

英文引用格式: Zhao Q, Liu XZ, Wang XD, Huang G, Mei H. Rich client middleware for runtime self-adaption. Ruan Jian Xue Bao/Journal of Software, 2013,24(7):1419−1435 (in Chinese). http://www.jos.org.cn/1000-9825/4319.htm

Rich Client Middleware for Runtime Self-Adaption

ZHAO Qi[1,2], LIU Xuan-Zhe[1,2], WANG Xu-Dong[1,2], HUANG Gang[1,2], MEI Hong[1,2]

[1](Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871, China)

Corresponding author: LIU Xuan-Zhe, E-mail: liuxzh@sei.pku.edu.cn, http://www.sei.pku.edu.cn/~liuxzh

Abstract: As the Internet has rapidly grown, rich Internet application (RIA) has become the mainstream application since it uses local storage and computes resources more effectively and therefore has better user experience. The client-side of RIA, i.e. rich client, can be thought of a sub-application which adopts “model-view-controller” architecture style and runs on the client-side runtime. Along with the development of mobile device and web browser, there are large differences among different client-side runtimes. However, rich client cannot predict the characteristics of its runtime due to the open nature of the Internet, and therefore suffers from the heterogeneous runtimes. This paper proposes a rich client middleware for runtime self-adaption. The middleware provides a MVC component model and decompose the problem of heterogeneous runtimes into three sub-problems: The problem of heterogeneous storage runtimes, the problem of heterogeneous computation runtimes and the problem of heterogeneous display runtimes. The middleware encapsulates a series of common self-adaption services which solve the problems correspondingly.

* 基金项目: 国家重点基础研究发展计划(973)(2009CB320703); 国家自然科学基金(61003010); 国家高技术研究发展计划(863)(2012AA011207)

收稿时间: 2012-02-25; 定稿时间: 2012-08-10

Key words: rich client; runtime self-adaption

随着 Internet 的快速发展,面向服务、云计算、移动计算等新技术的涌现,Internet 环境下最主流的应用形态——Web 应用也发生了巨大的变化.传统 Web 应用遵循“浏览器-服务器体系结构”,将数据与功能集中在服务器,浏览器只负责渲染呈现简单的 HTML 页面.这种体系结构保证 Web 应用具备易维护升级、安装部署成本低、支持跨平台访问的优势.然而,数据与功能的集中,使 Web 应用无法有效利用客户端存储计算资源;所有计算必须在服务器执行也导致大量不必要的网络延迟,降低了用户体验的良好感觉.这些都使 Web 应用无法满足 Internet 环境中日趋复杂的应用需求.

富客户端是对传统 Web 应用客户端的改进[1],支持在保持浏览器跨平台、易部署、易维护优点的同时,在富客户端中提供桌面客户端富用户体验、本地存储、本地计算等特性,进而满足 Internet 环境中的复杂需求.近年来,富客户端逐步得到了学术界与产业界的关注,主流的互联网应用提供商(例如谷歌、Facebook、百度、腾讯等)均或多或少地开始在自身的 Web 应用中支持富客户端.

一个完整的富客户端由富客户端应用与运行环境两部分组成,其中,前者是为满足用户需求、完成特定任务而开发的计算机程序,后者是支撑富客户端应用运行的软、硬件运行环境.

从体系结构上看,富客户端应用对传统 Web 应用的改进集中在如下 3 个方面:

首先,富客户端应用与服务器间交换的信息由数据界面混杂的 HTML 页面转变为结构化数据集.因此,富客户端应用可以在客户端建立数据模型(model),对数据进行进一步的加工、计算、存储.这使得应用可以增量地获取数据提供响应速度与网络利用率,并且在客户端数据源的帮助下使离线应用成为可能.

其次,应用不仅仅渲染静态 HTML 页面,而是通过服务器返回的数据生成友好的视图(view).这种视图能够及时响应用户的操作,异步请求服务器的数据或功能,避免网络延迟造成的视图无响应状态,提升了应用的用户体验.

最后,随着数据模型和视图复杂度的增加,常常由单独的程序模块协调视图与数据模型之间的关系,在客户端处理复杂的业务逻辑.这不仅进一步利用客户端计算资源提供了应用的响应速度,也降低了应用对服务器的计算资源的依赖,提高了整个 Web 应用的伸缩性.这种负责协调视图与数据模型、处理业务逻辑的模块,实际上在客户端即完成了在传统 Web 应用中控制器(controller)的任务.

综上,富客户端应用本身也遵循 Web 应用的“模型-视图-控制器(model-view-controller,简称 MVC)”体系结构风格.

而从上面分析也可以看出,为了更好地满足复杂的需求,富客户端应用需要使用运行环境中的存储计算资源.近年来,手机、平板电脑的不断发展和普及,逐渐成为重要的访问 Internet 的硬件设备.手机、平板电脑虽然相比传统 PC 具有更好的便携性,但同时,其存储计算能力较弱;另一方面,为适应富客户端应用的需求,浏览器的能力不断增强,也出现了新型浏览器、浏览器插件、基于浏览器内核的桌面运行环境等能力各异的富客户端软件运行环境.在软、硬件环境愈发复杂的情况下,Internet 环境开放动态的特性却使得富客户端应用不能预知其运行环境的特点.因此,富客户端应用不可避免地面临异构的运行环境.

如果从富客户端应用的 MVC 体系结构透视其异构运行环境,可以发现,在数据模型、控制器、视图这 3 方面,富客户端需要解决 3 类不同的适应性问题:

- 异构的存储环境:富客户端数据模型需要访问在客户端与服务器中存储的数据,因此,其需要能够访问传输协议、数据格式异构的本地与远程数据源.同时,移动设备的发展还要求富客户端能够运行在速度、稳定性不同的网络环境中.不同的网络环境也会影响数据模型对数据源的访问.因此,异构的数据源和异构的网络环境构成了富客户端数据模型面对的异构存储环境.
- 异构的计算环境:复杂富客户端的控制器需要进行复杂的本地计算(例如图形渲染、复杂的业务逻辑等),然而,手机、平板电脑、PC 这 3 类设备的计算能力差别很大.因此,富客户端控制器必须能够适应异构的计算环境,既要充分利用性能优异设备的本地计算能力,又要尽可能地保证在性能较差的设备

上可以顺利地运行.

- 异构的显示环境:富客户端视图面临着因不同设备不同尺寸屏幕所造成的显示环境适应性问题.手机等便携设备往往只配置尺寸较小的屏幕,因此,富客户端视图必须考虑如何适应不同尺寸的屏幕,在手机、PC 上都能提供良好的用户体验.

针对上述问题,本文提出一种富客户端运行环境自适应中间件(以下简称“富客户端中间件”或“中间件”).

本文的主要贡献在于:

- 首先,根据富客户端遵循的“模型-视图-控制器”体系结构风格,富客户端中间件提供一种符合该风格的 MVC 构件模型以及相应的支撑机制.
- 以此 MVC 构件模型为基础,中间件将富客户端运行环境适应性问题分解为数据模型、控制器、视图这 3 部分分别面临的存储环境、计算环境和显示环境的适应性问题,并提出解决这些问题的针对性自适应机制,进而解决整个富客户端的运行环境自适应问题.

本文第 1 节给出富客户端自适应中间件的概述.第 2 节介绍中间件使用的 MVC 构件模型与组装模型.第 3 节介绍中间件封装的存储、计算、显示异构环境自适应机制.第 4 节给出实验评估.第 5 节介绍相关工作.第 6 节总结全文并讨论下一步的工作设想.

1 富客户端运行环境自适应中间件概述

目前的富客户端应用遭遇复杂适应性问题的主要原因在于:应用直接运行在富客户端运行环境之上,运行环境的差异被直接暴露给应用.为此,本文提出富客户端运行环境自适应中间件.中间件作为一种运行于应用软件与底层运行环境之间的软件形态,通过引入中间层,应用与运行环境可以被有效地隔离(如图 1 所示).从体系结构上看,富客户端中间件的运行位置处于富客户端应用与运行环境中的软件运行时(通常是 Web 浏览器)之间.

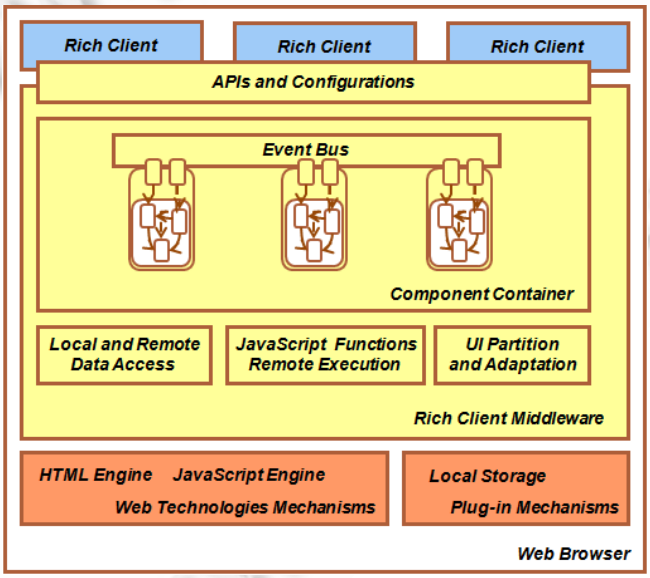


Fig.1 Overview of rich client middleware

图 1 富客户端中间件概览

富客户端中间件的实现依赖于富客户端运行时内置的支撑机制,基于 JavaScript,HTML,HTTP 等 Web 标准.中间件处于最接近富客户端运行环境的位置,因此可以直接了解运行环境的特征,进而提供最为有效的适应方案;另一方面,在引入富客户端中间件之后,富客户端应用运行于中间件之上,原本需要应用自身处理的异构性

问题被中间件屏蔽.在中间件的自适应公共服务的协助下,富客户端应用无需再关心其所处运行环境的异构性问题.

富客户端中间件基于浏览器内置及插件机制,由 Web 标准的 JavaScript 语言来实现.它主要包括两部分内容:

- 构件容器:本文提出一种符合富客户端体系结构风格的 MVC 构件模型.开发人员遵循该构件模型开发构件、提供构件接口定义.构件容器负责管理构件定义,并根据上层应用的请求,实例化构件、管理构件的生命周期;另一方面,为实现富客户端应用的具体功能,构件之间需要传递消息、调用彼此的功能.因此,构件容器遵循浏览器异步交互模式,提供基于事件的构件组装模型.该组装模型允许一个构件订阅其他构件发布的事件,并根据事件调用构件自身的方法完成组装.
- 自适应机制:自适应机制封装了富客户端应用适应异构软、硬件运行环境所需的功能.如前所述,通过 MVC 构件模型,可以将自适应机制分为 3 个子部分:
  - ✧ 多存储介质数据访问机制:多介质数据访问机制首先对浏览器的本地与远程数据源进行封装,统一异构数据源访问的 API.对于不支持本地存储的浏览器,使用远程数据源模拟.简言之,数据访问机制首先将各浏览器中异构的数据源抽象为一致的数据源.在此基础上,开发人员可以使用配置文件规定数据模型的存储位置、缓存策略.实际运行时,中间件会根据当前运行环境的特点以及开发人员提供的配置文件,自动选择使用最优数据源访问策略,使得富客户端应用可以良好地适应异构的存储环境;
  - ✧ 控制器函数服务器端运行机制:控制器函数服务器端运行机制在服务器端架设富客户端 JavaScript 脚本的运行环境,建立富客户端-服务器 JavaScript 上下文(context)同步机制.开发人员通过配置文件来标示富客户端应用中可以在服务器端运行的函数,在运行时刻,函数运行机制会自动地将当前环境中运行较慢、耗费计算资源较多的 JavaScript 函数及其运行时上下文移动到服务器端运行.通过这一机制,富客户端应用可以同时适应计算能力较强及能力较弱的异构计算环境;
  - ✧ 视图分割与屏幕适配机制:视图分割与屏幕适配机制会自动检测当前运行环境的屏幕尺寸,根据该尺寸对富客户端应用视图中的文档对象模型(document object model,简称 DOM)树进行分割,使得视图以可用、用户友好的方式显示.同时,屏幕适配机制会检测富客户端应用对 DOM 树的修改,针对每次修改重新分割 DOM 树,保证富客户端应用视图对屏幕的适应.

在使用中间件时,开发人员首先需要遵循中间件提供的构件模型开发构件,并利用组装模型对构件加以组装实现富客户端应用.此外,开发人员还需要提供必要的配置文件,使中间件了解应用中各个构件的元信息,以便在运行时刻确定应用的适应策略.

当一个遵循中间件构件模型组装而成的富客户端应用启动时,浏览器会首先加载中间件相关 JavaScript 脚本文件;而后,中间件会加载富客户端实现.根据富客户端应用实现的不同,构件容器会实例化需要的构件并加以组装;最后,中间件将根据当前运行环境的特征以及开发人员提供的配置文件,调整富客户端应用使用的数据源、函数的运行位置以及视图的显示规则,使得应用得以适应实际运行环境.

2 富客户端中间件构件模型及其容器

2.1 富客户端应用体系结构风格

在上文中,我们通过分析富客户端应用的特征指出,富客户端应用遵循模型-视图-控制器 MVC 的体系结构风格.其中,

- 数据模型是对富客户端应用需要使用的数据结构的建模.数据模型包含一系列的属性,在运行时刻,应用可能从客户端或服务器的数据源中取得数据,并利用这些数据根据数据模型构造一系列的数据实例;另一方面,应用可能新建或修改这些数据实例,如果被修改的数据实例是一个持久化的数据实例,那

么数据实例之上的修改将被保存回数据源.

- 视图是富客户端应用的用户界面(UI).视图一方面负责呈现所有数据实例中的信息,另一方面负责提供应用功能的操作入口.对于富客户端应用,视图可能提供相当复杂的交互操作方式.
- 控制器则是对富客户端应用中应用逻辑的封装.当用户在视图中进行操作时,控制器会把这一操作翻译为一系列对数据实例或视图本身的修改操作;另一方面,当应用从数据源生成数据实例或对数据实例的操作成功完成时,控制器也需要根据数据实例修改视图.

除了富客户端应用的结构外,我们还关注应用各个模块间的通信方式.富客户端应用的主要职责包括:1) 与服务器通信;2) 与用户交互.实际应用中,这两类交互通常需要较长的等待时间.原因是:首先,客户端与服务器的通信必须经过网络传输并导致较长的网络延时;其次,用户的多个操作之间通常也存在较大的时间间隔.

基于以上原因,富客户端运行环境实现了基于事件(event)以及回调(callback)的异步通信(asynchronous communication)风格,以便对应用中需要“长等待”的操作进行支持.

MVC 体系结构风格与基于事件的异步通信是富客户端应用最重要的特征.通过对这一风格的明晰,本文提出中间件使用的 MVC 构件模型与基于事件的组装模型.

2.2 构件模型

富客户端构件容器提供了一个遵循 MVC 模式的构件模型,该构件模型封装了应用的数据模型、控制逻辑以及视图,保证构件能够具备良好的结构,并同时作为自适应机制的基础.富客户端构件模型如图 2 所示.

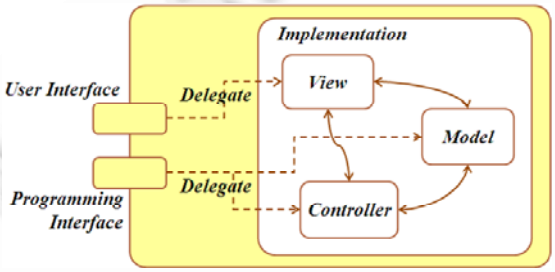


Fig.2 Component model

图 2 构件模型

富客户端构件与传统构件类似,由两部分构成:构件实现与构件接口.富客户端构件同时封装了功能与视图,因此,接口不仅提供传统构件的编程接口(programming interface),还提供用户接口(user interface).编程接口暴露构件的功能,构件可以通过编程接口相互调用,实现构件组装.用户接口负责与用户交互,响应用户的操作,调用构件实现中的相应功能.

构件实现遵循 MVC 模式,其中,

(1) 数据模型封装了构件需要的数据,定义其数据结构,提供访问本地、远程数据源以及将获取的结构化数据集解析为数据模型实例的访问.数据模型还提供对数据进行过滤、合并、缓存的功能.

(2) 视图是一段 HTML 文档片段.构件视图会在运行时刻被实例化为 DOM 树,并插入富客户端应用的完整视图中.视图还定义了如何将对视图元素进行用户操作转换为构件事件.

(3) 控制器负责连接数据模型与视图,其将视图发起的事件翻译为一系列对不同数据模型的创建、删除、查询、修改等操作,并根据操作结果通知视图进行更新.当操作数据模型的流程逻辑比较复杂时,控制器往往会消耗较多的计算资源.

编程接口显示出构件数据模型以及控制器的功能,它包含两个部分:方法与事件.方法用于调用控制器功能,或查询、修改数据模型的状态;事件定义了构件状态的变化,可以被发布到组装模型的事件总线中,其他构件可以订阅事件,以获知某个构件的状态改变并执行必要的方法.用户接口显示出构件视图,开发人员可以通过用

户接口提供视图中元素的元信息,例如:复合元素不可分割;元素可以被隐藏等.

2.3 组装模型

富客户端中间件的构件容器还提供一个符合富客户端应用特征的基于事件的发布/订阅构件组装模型.

为了实现基于事件的组装模型,构件容器提供统一的事件模型以及事件总线(event bus),如图 3 所示.

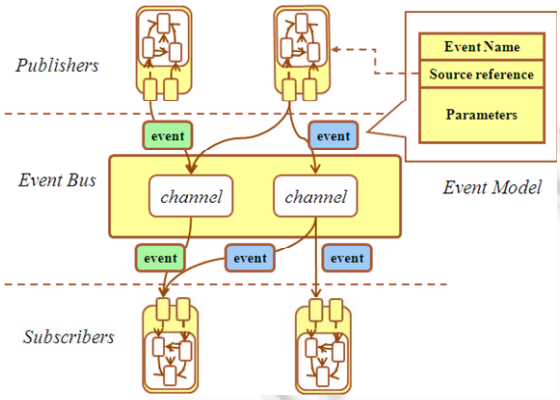


Fig.3 Event model and event bus

图 3 事件模型与事件总线

2.3.1 事件模型

浏览器提供 DOM 事件模型.每一个 DOM 节点上可以触发大量 DOM 事件,例如鼠标点击、鼠标移入/移出等.当事件被触发时,浏览器会创建一个 DOM 事件实例,并将这个事件实例传递给回调函数.浏览器中还存在与 DOM 节点无关的异步操作,这些异步操作也可能触发事件,例如 Ajax 调用、setTimeInterval 方法等,但这类事件不属于 DOM 事件,也不会生成 DOM 事件实例.此外,除了浏览器内置的事件,构件中还可能存在其他事件,例如数据模型状态的改变等.

为了简化构件组装,构件容器为所有不同类型的事件提供抽象的统一事件模型.该事件模型由事件名称、事件的源构件以及事件参数构成.构件需要根据浏览器内置事件或者构件私有事件创建统一事件实例,再将事件实例发布到事件总线中.

2.3.2 事件总线

构件容器提供一条事件总线用以支持事件发布-订阅的机制.基于事件的功能组装是富客户端构件组装的最主要方式,因此,一个富客户端应用中可能存在大量事件订阅.为了保证性能,事件总线以一种轻量的方式实现.实际上,经过测试,在较为复杂的情况下,尽管在不同的浏览器中有所差异,事件总线依然能在 2s 内处理超过 200 个构件实例以及大约 500 种订阅方法.在实践中,大部分富客户端应用使用的构件数量以及组装的复杂程度通常不会达到这一级别.因此可以说,本文提供的富客户端中间件具有良好的性能和伸缩性.

在进行构件功能组装时,开发人员可以在事件总线上定义若干个频道(channel),每个频道由一个唯一的标识符命名.构件接口中定义的事件可以被指定发布到任意一个或多个频道中.在设置发布时,构件会记录自身事件与事件总线频道的映射关系.构件也可以订阅某个频道,并规定事件到达时需执行构件接口中定义的哪种方法.事件总线会记录所有的构件对频道订阅以及订阅的方法.

在富客户端应用运行过程中:当一个事件被触发时,构件会首先检查事件与频道之间的映射关系,以确定事件应被发布到哪些频道中;之后,构件会生成事件实例,并将其发布到事件总线的特定频道中;然后,当事件抵达总线频道后,总线会检查所有订阅该频道的构件,并以事件实例为参数,依次调用订阅该频道构件指定的方法,实现构件之间的功能组装.

3 富客户端运行环境自适应机制

3.1 富客户端多存储介质数据访问机制

在富客户端应用中,数据既可以被存放在服务器端的数据库中,也可以被存储在客户端的本地数据源中[2].因此,根据存储物理位置的不同,应用可以访问两类数据源——位于服务器的远程数据源以及位于客户端的本地数据源.两类不同的数据源面向不同的存储需求,因此,富客户端应用需要可对两者进行访问[3].

富客户端应用相比传统 Web 应用的一大优势在于能够在本地数据源中存储数据,这可以帮助应用减少不必要的网络访问,提供更好的用户体验.近年来,出现了大量富客户端本地存储的解决方案,例如 Flash LSO 以及 HTML 5.然而,不同的方案也给客户端数据源的访问带来问题:

- 首先,不同的本地数据源可能是异构的.目前,本地数据源可以分为两大类:第 1 类被称为“基于 SQL 的本地数据源”,它将轻量级的 SQL 数据库(例如 Sqlite)嵌入到浏览器内,并提供相关的 SQL 查询接口;第 2 类被称为“基于键值表的本地数据源”,它为每个富客户端应用提供一张两列的键值表,允许应用对键值表进行增删查改操作.很显然,以上两类数据源是异构的.
- 其次,即使同一类数据源,也可能使用不兼容的 API.例如,Google Gears 与 HTML 5 WebSQL 都提供基于 SQL 的本地数据源,然而在 Google Gears,以同步的方式执行 SQL 查询;反之,HTML 5 WebSQL 要求查询异步执行,并在执行完成后调用之前注册的回调函数处理结果.因此,富客户端应用也会面临不兼容的 API 造成的问题.

在实际应用中,不同的浏览器对本地数据源的支持程度也各不相同,这使得富客户端应用需要考虑当前运行的浏览器实际支持的本地数据源,选择当前运行环境中可用的乃至最优的数据源.不同的浏览器对不同本地数据源的支持程度可见表 1.

Table 1 Browser support on the local data sources

表 1 浏览器对本地数据源的支持情况

	IE	Firefox	Chrome	Safari	IE (Mobile)	Safari (iPhone)
IE userData**	5.5+	N/A	N/A	N/A	N/A	N/A
Flash LSO***	Plug-in	Plug-in	Plug-in	Plug-in	N/A	N/A
HTML 5 Local****	8.0+	3.5+	3.0+	3.1+	N/A	3.1+ (OS 2+)
HTML 5 Database	N/A	N/A	3.0+	3.1+	N/A	3.1+ (OS 2+)
Google Gears*****	6.0+, Plug-in	1.5+, Plug-in	Default	N/A	N/A	N/A

另一方面,虽然通过使用标准的 HTTP 以及 RESTful 服务在服务器与富客户端应用之间进行数据传输,可以屏蔽大部分远程数据源的异构问题.但富客户端应用在访问远程数据源时,依然需要处理在不同的网络环境中富客户端应用与远程数据源的连接问题.特别地,出于性能的考虑,应用需要有效地对远程数据进行缓存.因为富客户端应用运行在浏览器中,因此可以使用浏览器内置的 HTTP 缓存机制(HTTP/1.1,http://www.w3.org/Protocols/rfc2616/rfc2616.html).然而,HTTP 缓存规范要求缓存策略在服务器端设置,在浏览器中生效.

富客户端应用会从不同的 URL 地址获取多个数据集合.对于 HTTP 缓存机制,这些数据集合被作为不可分割的整体缓存.然而在实际应用中,这些数据集合往往存在交集.HTTP 缓存机制不能有效地利用重复的数据,造成不必要的远程数据源访问.此外,移动计算的发展,使得富客户端应用可能运行在不稳定的网络环境中.在这样的环境里,应用可能处于离线状态,离线状态下的应用需要尽可能地使用缓存数据,直到恢复连接.HTTP 缓存机制也无法适应网络环境的状态,将可用的临时数据标志为“过期”,造成应用的不可用.

** http://msdn.microsoft.com/en-us/library/ms531424%28v=vs.85%29.aspx

*** http://www.adobe.com/products/flashplayer/systemreqs/

**** http://en.wikipedia.org/wiki/Comparison_of_layout_engines_%28HTML_5%29#cite_note-177

***** http://en.wikipedia.org/wiki/Google_Gears

综上,富客户端多存储介质数据访问机制分别提供针对本地与远程数据源的自适应机制.

3.1.1 本地数据源自适应访问机制

本地数据源自适应访问机制提供两项功能:

1) 提供一组统一的本地数据源访问 API 以屏蔽数据源的异构性;

2) 根据当前富客户端应用与浏览器的特点,选择当前场景下最适合的本地数据源.

针对前者,适应机制首先基于构件中的数据模型实现了 ActiveRecord(http://en.wikipedia.org/wiki/Active_record_patter)模式;接着,开发人员可以为每一个数据模型指定模型元数据,包括每个数据模型需要被持久化的属性名称、类型等信息.依据这些信息,当富客户端应用初始化时,适应机制会将数据访问方法(例如 *User.find*, *user.save*,*user.update*)插入到数据对象中,应用可以直接调用这些方法.利用 ActiveRecord 模式,适应机制可以屏蔽同类数据源中的不兼容 API,并将同一个数据对象映射到不同类型的数据源中——一个数据对象或者被映射为 SQL 数据库中的一行,或者被映射为键值表中的由 id 以及类型索引的一项.

支持统一数据访问后,适应机制还需要选择最适应当前场景的本地数据源.选择过程分为两个步骤:首先,根据当前富客户端应用运行的浏览器选择可用的本地数据源.通过浏览器中的"user-agent"属性可以判断浏览器版本;通过一系列条件语句(例如 if (window['google'] && window['google']['gears']))则可以判断浏览器是否支持某种本地数据源.考虑到在最坏情况下,浏览器可能不支持任何本地数据源,此时,适应机制会提供一个远程数据源来模拟本地数据源,以保证任一浏览器都至少有一个可用的本地数据源.经过上面的步骤,适应机制选择出多个可用的本地数据源.接着,需要从所有候选中选择最适合当前浏览器与应用的数据源.此时,需要将不同数据源的性能和容量限制因素纳入考虑范围.如图 4 与表 2 所示,各数据源在不同浏览器中有着不同的性能表现以及容量限制.

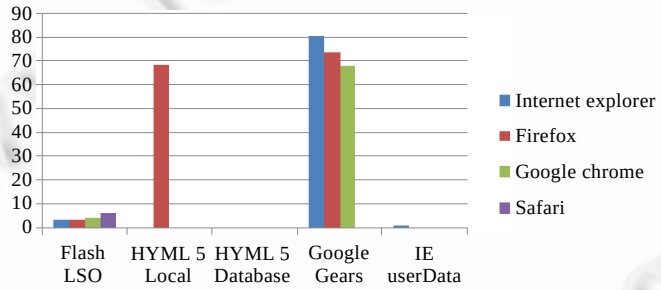


Fig.4 Insertion performances of different local storages in different browser

图 4 不同浏览器中各数据源插入操作的性能

Table 2 Capacity limit of different data resources

表 2 不同数据源的容量限制

Data source	Flash LSO	IE userData	HTML 5 Local	HTML5 Database	Google Gears
Size limit	100K	250K	Depend on Impl.	Depend on Impl.	No limit

适应机制以向量$\langle P_c,P_r,P_u,P_d\rangle$表示不同数据源增删查改操作的性能,以 $S_{\max}$ 表示数据源的容量限制.同时,设定特征向量$\langle W_c,W_r,W_u,W_d\rangle$表示富客户端应用中增删查改操作的权重,$S_{\text{app-max}}$ 表示应用的最大存储容量.因此,不同数据源的适用性可以通过如下的评估函数来表示:

$$E=\begin{cases}\dfrac{1}{\langle P_c,P_r,P_u,P_d\rangle\times\langle W_c,W_r,W_u,W_d\rangle}, & \text{if } (S_{\max}\geqslant S_{\text{app-max}})\\ 0, & \text{if } (S_{\max}<S_{\text{app-max}})\end{cases}.$$

可以看出,最适用的数据源拥有最大的 E 值.在富客户端应用运行过程中,适应机制会记录一定时间窗口内每一次的数据访问操作,并根据历史记录设定$\langle W_c,W_r,W_u,W_d\rangle$.同时,$S_{\text{app-max}}$ 会被设为应用运行以来存储容量的最

大值.适应机制会每隔一段时间重新计算评估函数,当发现更合适的数据源时,进行数据源迁移.

3.1.2 远程数据源自适应缓存机制

缓存机制支持富客户端应用在数据模型上定制缓存策略[4].同时,缓存机制可能会依据应用的行为实时地计算数据模型对象的一些属性(如访问频率),并依据这些属性动态优化指定的策略.缓存机制支持的策略见表 3[5].

Table 3 Cache strategy

表 3 缓存策略

策略名称	策略含义
Time since last access (TLA)	最后一次请求的时间戳
Entry time (ET)	进入缓存池的时间戳
Frequency of access (FOA)	访问频率
Time to live (TTL)	生存时间
Enable pre-fetch (EP)	开启预取
Enable adaptive (EA)	开启缓存自适应

缓存框架会实时地计算下列信息,见表 4.

Table 4 Arguments of cache strategy

表 4 缓存策略涉及的参数

参数名称	参数含义
Frequency of access (Foa)	用户对某个缓存对象的访问频率
Frequency of update (Fou)	服务器对某个缓存对象的更新频率
Computation cost (CC)	更新这一缓存对象的开销
Delivery cost (DC)	从服务器端返回请求结果所需开销

通过上述信息,缓存机制会计算是否缓存对象、缓存多长时间、是否进行预取;而后,通过调整之前所述的缓存策略,将这一计算结果反映于数据模型上.

实际应用中,用户通常会无规律地访问缓存数据.一个固定的缓存过期时间因而会给系统性能带来一定的问题:一方面,如果缓存时间固定得太短,则数据还没使用便已过期,导致较低的命中率和较高的请求数目.另一方面,如果缓存时间设置得太长,用户会被迫使用过老的数据,使得整个缓存系统的价值有所下降.

此外,从实验中我们发现,除了命中率外,还应有更多的因素影响过期时间的取值.这些因素来源于用户的行为模式和 Web 服务的行为模式.对于前者,通过将命中率和访问频率作为数据集,对过期时间加以训练,可以获得更符合用户行为的过期时间;而对于后者,Web 服务中数据结果的变化频率也应对缓存时间的设置有所影响.在这样的条件下,过期时间显然应该相应地加以变化.尽管在缓存框架中加入以上所有因素的考虑会更加准确,但由于我们的缓存框架定位于富客户端应用,在取得 Web 服务的行为方面存在一定限制,因此适应机制仅考虑用户行为模式.综上,我们提出一种改进的缓存策略.在这个策略中,适应机制会根据用户的命中率和访问频率动态地改变缓存数据的过期时间.具体而言,通过下列的用户行为模式来不断优化过期时间:

- 用户每次的请求要么命中,要么缺失,从而影响该缓存模型的总命中率.命中率如果有变低/高的趋势,则缓存时间也应相应加以调整,这一调整应取决于命中率的变化率而非单个点的命中率大小.
- 用户的请求频率往往会发生变化,从而影响单位时间内对缓存模型的请求次数,即访问频率.

在实践中我们发现,模拟退火算法能够很好地适应上述情况.我们借鉴了文献[6]中提出的算法,但与之不同的是,我们的模拟退火算法着重于找到最适合用户的过期时间.我们设计算法的目标是,过期时间会动态适应变化的命中率和访问频率,从而动态地适应用户行为模式.

估计值 V_i 代表了缓存模型对象 i 的过期时间 I_i.我们用 V_a 代表一个阈值,通过下面的公式计算 V_i.

$$V_i=\begin{cases}\mathrm{e}^{\frac{\alpha\times hit\_ratio}{temperature}}, & \text{if } (V_i>V_a)\\ NA, & \text{if } (V_i\leqslant V_a)\end{cases} \tag{1}$$

公式(1)中的变量具有下述含义:

- 变量 *hit_ratio* 代表缓存对象 I_i 的命中率.我们使用常数以扩大命中率的影响力,这一值在我们的实现中取常数 1 000.随着命中率的上升,过期时间应相应增加.这一结果对应于如下事实:即,如果我们总能反复地找到所需缓存数据,那么这一数据应缓存更长时间;相反地,如果命中率较低,也就意味着现有的缓存对象很少被访问,那么根据局部性原理,它往往位于"局部"以外,因此在最近的将来被访问的可能性也较小(对于过远的将来则不然).在这种情形下,我们应该减少该模块的过期时间,使其尽快移出缓存池.由于过短的缓存时间使得缓存数据失去意义,因此我们设置一个阈值 V_a(如前所述),以直接淘汰过期时间比阈值还短的缓存模块对象.
- 变量 *temperature* 代表温度,对应于过期时间对命中率变化的弹性.另外,我们认为,温度从本质上来源于对服务的访问频率,因而具有用户行为模式的影响.例如,用户可能在短时间内大量使用同一请求访问同一服务,导致命中率显著提高.如果过期时间对命中率弹性较大,则这一结果对导致过期时间在短时间内的提升过快;相反地,如果用户在相当长的时间内不请求某个数据块,则无法采集足够的数据使过期时间适应用户模式.基于上述原因,我们认为,模拟退火算法中的温度在我们的场景下对应于用户对服务的访问频率.

在一个退火过程中,温度通过下列方式来加以调整:

用'T'代表温度,min$T$ 为温度的最小值,使用'*Foa*'代表用户对服务的访问频率(frequency of accessing),则有:

$$T=\mathrm{Max}\{\mathrm{min}T,Foa^2\} \tag{2}$$

公式(2)对应于下列事实:即,随着访问频率的增加,温度上升;随其减少而下降.然而,温度改变的弹性不应过大,否则会导致极端的结果值.因此,我们设置了一个下限以约束温度的取值.退火算法在温度下降到 min$T$ 时结束(对应现实退火中的降温完成).*Foa* 过高,意味着在短时期的将来更多的服务可能被访问.在这样的情况下,如果过期时间随命中率变化的弹性较大,则它会很快变得极端化.例如生成一个长达 1 年或者短至 1ms 的过期时间.这两种极端情况显然都应避免.因此,我们必须升温以减慢反应速度,即减少此时过期时间对于命中率的弹性;相反地,如果 *Foa* 过低,意味着系统需要很长的时间收集数据使得缓存时间适应用户行为.在这种情况下,我们必须降温以增加过期时间对命中率的弹性.亦即,温度应与 *Foa* 正相关.

3.1.3 控制器函数服务器端运行机制

富客户端构件的控制器中往往包含大量复杂的业务逻辑,这些业务逻辑的运行需要消耗大量客户端运行环境的计算资源.然而,随着移动计算的发展以及移动设备的普及,富客户端运行环境的计算能力千差万别.因此,控制器函数服务器端运行机制(以下简称"函数运行机制")在富客户端应用运行过程时,动态地将 JavaScript 函数迁移到服务器执行,借以减轻应用对运行环境计算资源的依赖.其核心思想是:在富客户端应用加载时替换耗时较多的函数,使其在调用时实际调用服务器端的函数副本.

函数运行机制需要开发人员以配置文件的形式提供富客户端应用的元信息,包括服务器预加载的文件、需要被迁移的函数列表、上下文环境同步需要用到的部分类的构造函数.之后,当应用初始化时,服务器会根据开发人员提供的元信息完成 JavaScript 上下文环境的初始化以及与客户端运行环境的上下文对接.同时,函数运行机制会提取需要迁移的函数,将函数副本迁移到服务器 JavaScript 上下文环境中.当应用实际调用被迁移函数时,调用请求会被转发到服务器.服务器在接受到请求后,首先会将其上下文环境与富客户端应用同步,而后执行函数.而在函数运行的结果返回时,富客户端应用同样需要同步上下文,再解析函数返回结果并执行之后的代码.以上流程中,函数运行机制最重要的任务是保证服务器与富客户端应用拥有相同的函数运行上下文.

3.1.4 服务器端与富客户端 JavaScript 上下文对接

为了使富客户端 JavaScript 可以在服务器正确运行,首先需要实现服务器与富客户端 JavaScript 上下文环境的对接.这要求服务器支持 JavaScript 这一客户端脚本语言的运行.为此,函数运行机制将 JavaScript 语言引擎集成到 Web 服务器中.当 Web 服务器接收到富客户端应用中 JavaScript 函数运行的请求后,会将请求交由嵌入服务器的 JavaScript 引擎模块.该模块负责 JavaScript 环境运行时上下文环境的同步以及解释执行富客户端应

用发送的 JavaScript 代码,并将结果返回给 Web 服务器,Web 服务器再将结果返回到富客户端.服务器与富客户端应用的交互情况如图 5 所示.

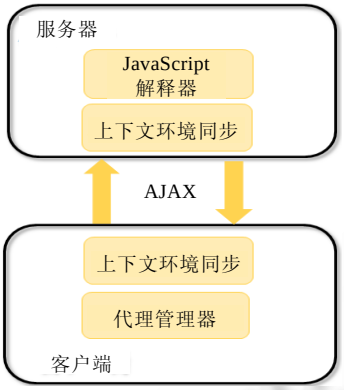


Fig.5 Communication between Web server and rich client

图 5 Web 服务器与富客户端的交互

由于服务器和富客户端应用为一对多的关系,并且 JavaScript 上下文在运行时刻会根据具体应用的执行结构发生变化,因此,服务器在接收到应用请求时必须区分不同富客户端应用的 JavaScript 上下文,这要求在服务器与富客户端应用之间建立会话(session).然而与浏览器级别共享的 HTTP 会话不同,同一浏览器中多个富客户端应用实例的 JavaScript 上下文内容并不相同.因此,函数运行机制在服务器与每一个运行在浏览器页面(page)中的富客户端应用实例建立页面会话(page session).

函数运行机制以如下方式维护页面会话:

- 首先,服务器端会为每一个富客户端应用实例建立单独的 JavaScript 上下文,以保证不同的上下文之间不会互相干扰.
- 接着,服务器端会为每一个上下文生成一个会话 ID,并返回给富客户端应用.而所有会话 ID 共同构成一个字典,用以对上下文进行检索.
- 在以后的交互过程中,富客户端应用的每一次对服务器的请求都会同时发送会话 ID;而服务器在接受到富客户端应用请求后,通过会话 ID 来获取该应用对应的服务器 JavaScript 上下文,将需要执行的函数放入该上下文中运行.
- 最后,富客户端应用关闭之前,会请求服务器销毁失效的 JavaScript 上下文.
- 同时,服务器的会话 ID 字典中会记录每个上下文最近一次运行的时间,并采用轮询的方式查看是否有上下文超时,若超时,也将销毁失效的 JavaScript 上下文.

3.1.5 服务器端与富客户端 JavaScript 上下文同步

在上下文对接完成后,为了使被迁移的函数可以正确运行,还需要在每次函数运行前将该函数依赖的上下文同步至服务器端对应的 JavaScript 上下文中.同理,当服务器端的函数执行完毕后,也需要将上下文信息同步回富客户端应用.

JavaScript 语言中,一个函数的执行依赖于其上下文中的 4 类变量:

- this 指针:JavaScript 函数调用时必须传入一个 this 指针,例如 *Person*1.*getName*()语句,*getName* 函数被执行时,*Person*1 即为该次执行时 this 指针指向的对象.在 *getName* 内部作用域中使用关键词 this,即可获取 *Person*1 的引用.当函数没有显式使用‘.’操作符被调用时,this 指针默认指向 JavaScript 上下文中的最高层级对象——富客户端应用中的 window 对象;
- 参数:JavaScript 函数调用时可以传入若干个参数,如 *add*(*a*,*b*)中,*a*,*b* 即为 *add* 函数本次执行的参数.JavaScript 函数允许传入任意多个参数,在函数体可以通过 *arguments* 变量获得参数数组;

- 全局变量:即函数执行过程中依赖的全局作用域中的变量;
- 闭包变量:JavaScript 语言中,函数自身的作用域构成一个闭包(closure).当函数定义嵌套时,被嵌套的函数可以获取上层函数作用域(闭包)中的变量.例如下面的代码:

```
function f(){
    var a=0;
    return function(){
        a=a+1;
        return a;
    }
}
var g=f(), h=f();
g();  //返回值为 1
g();  //返回值为 2
h();  //返回值为 1
```

f 函数的返回值为一个匿名函数,在该匿名函数内部调用了 *f* 闭包中的变量 *a*——将 *a* 加 1 并返回 *a* 的值.上面代码中,*g* 和 *h* 为 *f* 两次执行的结果.由于 *f* 每次运行时都会新建一个闭包,因而 *g* 和 *h* 在各自运行时用到的 *a* 为不同闭包内的变量.

在函数执行过程中,可能会使用上层函数闭包中的变量.

因此,为了保证服务器、富客户端 JavaScript 上下文的同步,必须在函数运行前后将这 4 类变量在富客户端应用和服务器之间同步.同步过程可分为如下两步:

首先,函数运行机制会构造对象图.

JavaScript 中的变量可能是基本数据类型(如 int,string)或者对象引用,其中,对象具备唯一性(即,两个属性值完全相同的对象用“==”操作符判断的结果也为假).对象在 JavaScript 中可以被多个变量引用.因此,在函数执行依赖的 this 指针、参数、全局变量、闭包变量这 4 类变量之间,很可能存在多个变量或变量的子属性引用同一个对象的情况.一个典型对象图结构如图 6 所示,其中,箭头表示包含于关系:*a* 包含 *b*,*c* 两个对象,*b*,*c* 都包含 *e*,*e* 包含 *d*,*d* 包含 *b*.其中,*b*,*d*,*e* 的嵌套关系构成了一个回路.

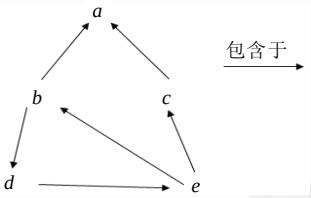


Fig.6 Objects graph sample

图 6 对象图示例

为了在同步时保持变量与对象间的对应关系,需要将上述 4 类变量所指向的所有对象以及对象之间的关系以对象图的形式记录.对象图构造过程如下:

(1) 遍历所有变量,将其中的对象加入对象图中,并为其分配唯一标识 UID;

(2) 遍历新加入对象图的对象的属性值,将其中未被标识的对象加入对象图中,并对其进行标识;

(3) 重复步骤(2),直至所有对象均已被标记.

其中,UID 的作用为:1) 标识此对象已被添加到对象图中;2) 通过 UID 记录对象的嵌套关系.

当对象图构造完成之后,函数运行机制会使用一种基于 JSON 的方式将对象图序列化为一个字符串.

该序列化方法对于基本的数据类型(如 int,string),直接按 JSON 的方式处理.但对于对象,标准 JSON 无法重

现对象的类型信息以及对象之间的嵌套关系.因此,函数运行机制在处理对象时增加了其类型信息,例如{type:“Class_*T*”,value:{/*JSON value*/}},Class_*T* 即为该对象的类.此外,因为对象图中已经标记了所有需要序列化的对象,所以在序列化类型为对象的属性值时,只用记录该对象的 UID;而在反序列化时,即可根据 UID 在对象图中找到对应的对象,从而重新建立对象图.在反序列化对象图时,对原有对象图的操作均为更新操作.即:如果该对象已存在且其类型信息未变动,则更新其属性值;如果该对象尚未创建,则创建一个新的该类对象.

3.1.6 视图分割与屏幕适配机制

随着移动计算、便携设备的发展,手机成为 PC 之外的另一种重要的互联网访问设备.截止到 2010 年底,手机互联网用户在全体互联网用户中的比例超过 2/3[7].然而作为运行富客户端应用的两类重要设备,手机和 PC 的显示环境存在极大差异——目前,PC 的主流显示分辨率为 1280×800 或 1440×900;而手机的主流分辨率为 320×240 或 480×320.这导致在开发富客户端应用时,需要针对两类设备分别设计视图.

一些已有的研究工作关注于如何使为 PC 屏幕设计的传统 Web 应用 HTML 页面自动地适配手机屏幕[8,9],但是,这些工作主要针对服务器传输的 HTML 页面.富客户端应用视图与传统 Web 应用的 HTML 页面不同,富客户端视图是在运行时刻根据服务器发送的数据集动态生成的,再根据用户的操作动态地加以变化.在针对传统 Web 应用的方法看来,富客户端应用的 HTML 页面往往是仅包含简单框架的空白页面.

富客户端视图分割与屏幕适配机制提出一种可在运行时刻分割动态生成视图,并在视图更新后动态重分割的方法.这种方法的要点在于其分割行为基于富客户端视图的运行时刻 DOM 树,而非基于传统的 HTML 页面.DOM 树可以忠实地反映运行时刻的视图结构,从而保证了分割适配机制的正确性.

3.1.7 DOM 树分析

为了进行视图分割适配,我们首先需要对 DOM 树进行分析.在富客户端视图的 DOM 树中,存在多种不同类型的节点,其中某些节点虽然存在子节点,但它们却是逻辑上的原子节点,因此不需要或者不可分割.例如:一个视图中的下拉菜单对应 DOM 树中的 select 节点,该节点下包含多个 option 节点代表菜单中的不同选项,然而在 DOM 树中,select 作为一个整体不可分割.此外,在 DOM 中还存在一些节点并不在视图中显示,例如 script 类型的节点,这些节点不需要参与到分割适配的过程中.

综上,我们将视图 DOM 树中的节点分为如下 3 类:

- 可分割的节点:以这类节点为根节点的子树可以被分割.这类节点包括⟨div⟩,⟨table⟩等;
- 不可分割的节点:以这类节点为根节点的子树不可分割,无论其面积是否超过了屏幕尺寸.这些节点还细分为多种类型,例如⟨input⟩,⟨button⟩等 DOM 树中实际的原子节点,或者上文提到的⟨select⟩等 DOM 树中的复合节点、逻辑上的原子节点;
- 可忽略的节点:在视图中不可见的节点.这些节点在分割过程中可被忽略,例如⟨script⟩,⟨style⟩节点.

对 DOM 树节点进行分类后,当视图被动态生成完毕之后,分割适配机制会计算所有不可分割节点以及其父节点的长宽尺寸.因为如图 7 所示,DOM 树中子节点可能超过父节点的范围.所以,分割适配机制不能简单地只计算一个节点的尺寸,而需要递归地计算给定节点所有子节点的尺寸及其相对父节点的偏移量,利用这些信息计算出一个节点在实际显示中的尺寸.

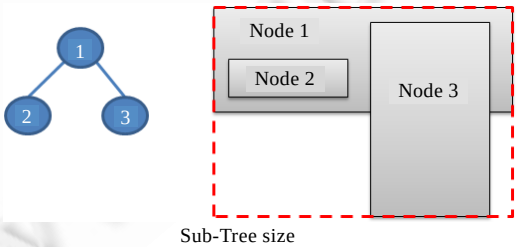


Fig.7 Size of a node

图 7 节点的尺寸

3.1.8 DOM 树分割

当计算出每个 DOM 节点的类型与尺寸信息后,分割适配机制会依据节点与屏幕的长宽尺寸来进行分割.其基本策略是尽可能地保证分割后每个子视图的长宽小于屏幕的长宽,这样可以使得分割后的视图无需滚动即可在屏幕中完整显示.此外,在长度、宽度两个属性中,会优先保证子视图的宽度小于屏幕宽度,因为在实际应用中,用户较不适应水平滚动条.图 8 给出一个分割的示例.

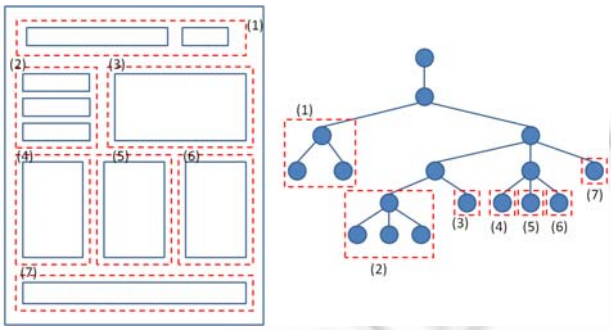


Fig.8 An example of partition pattern

图 8 DOM 树分割示例

分割完成后,分割适配机制会通过设置 DOM 节点的显示隐藏属性,保证任一时刻只有一个子视图对用户可见.此外,在大多数情况下,子视图的长宽不会和屏幕的长宽精确匹配.因此,分割适配机制还会适当地对子视图进行缩放,以减少用户使用滚动条的情况,提高富客户端应用的用户体验.

最后,富客户端应用可能在运行时刻动态修改视图,因此,分割适配机制会监视应用对视图的修改,并在视图发生改变后,重新对被修改的 DOM 子树进行分割,以保证适配的正确性.

4 实验评估

在使用富客户端中间件时,开发人员首先关心的是中间件的开销与性能.为此,我们设计一组仿真实验:在不同浏览器中加载空中间件以及实例化 10,50,100,200 个构件,检测浏览器内存消耗.我们还对实例化的构件进行组装,测试订阅当有 500 种~25 000 种方法订阅事件时,全部方法运行完毕所需要的时间,以此判断中间件组装模型的性能.从图 9 中可以看到:以富客户端应用 Gmail 为参照,中间件在除 IE 外的所有浏览器中,实例化 200 个构件的内存开销不会超过 Gmail 的内存消耗(在 IE 中,200 个构件比 Gmail 略高);而在所有浏览器中,执行 25 000 种订阅方法的时间不超过 400ms.以我们在实际使用中的经验来看,复杂富客户端应用中构件数量与一个事件的订阅方法都不会超过 100 种.由此可见,中间件具有良好的开销和性能.

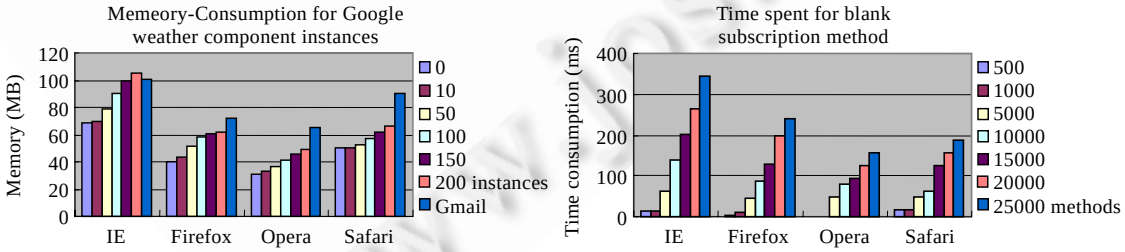


Fig.9 Performance of component model and composition model

图 9 构件模型与组装模型性能

然后,我们需要评估 3 种自适应机制的实际效果.对于多存储介质数据访问机制,实验首先实例化一组使用

本地数据源的构件,接着编写仿真脚本让构件模拟用户操作反复读写本地数据源.实验记录不同数据源选择策略下数据源访问总的耗时,结果如图 10 所示.在不进行自适应的情况下,默认数据源选择策略会选取了容量较大但速度较慢的本地数据源,而“读优先”策略选取了读取速度快但写入速度慢的数据源.因为应用本身的特点是读写平衡,所以默认和“读优先”策略的性能很差.而使用自适应策略达到了与手工指定“读写均衡”策略相似的性能,说明自适应策略可以很好地帮助富客户端适应应用本身与浏览器的特点.

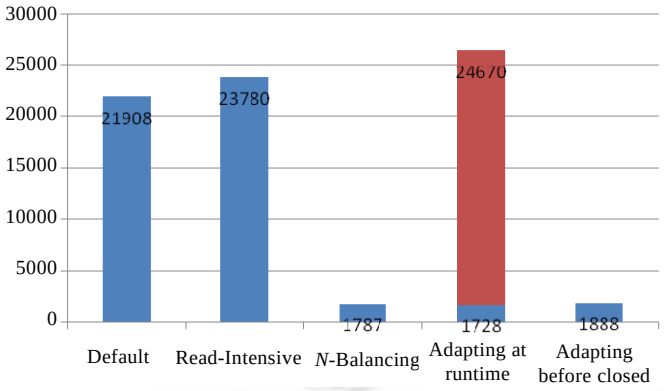


Fig.10 Performance of data access mechanism

图 10 多存储介质数据访问机制性能

对于视图分割与屏幕适配机制,我们使用该机制的分割方法对若干主流的 Web 应用视图进行分割,其结果见表 5.可以看出:分割适配机制取得了较好的效果,绝大部分视图没有变形,出现水平滚动条的概率在可接受范围内.同时,大部分视图没有被过度分割,具备合理的、保证用户体验的子视图数量.

Table 5 Result of view partition mechanism

表 5 视图分割与屏幕适配机制效果

Rich client	Partition units	Horizontal overflow (%)	Vertical overflow (%)	Fragment (total)	Deformation (total)
Google (search result)	11	75.71	22.19	4	0
Yahoo (home)	8	63.8	58.67	3	0
Baidu (search result)	6	135.94	33.33	2	0
Wikipedia (article)	7	101.28	50.63	1	0
Window live (home)	4	110.57	55.86	1	0
QQ.com (home)	30	90.35	26.82	9	0
MSN (home)	13	61.3	45.19	1	0
Linkedln (personal home page)	12	50.56	51.09	3	0
Amazon (home)	9	90.37	43.99	0	0
Taobao (home)	10	102.98	40.22	2	2
Sina (home)	36	26.55	47.69	5	6
Wordpress (blog page)	35	7.52	218.11	7	2
Ebay (home)	12	71.63	36.04	1	0
163.com (home)	24	38.39	55.43	0	3
Paypal (home)	9	34.03	16.91	3	0

5 相关工作

文献[1,10,11]提出富客户端构件组装的概念,指出,可以通过开发一系列的富客户端构件并对这些构件进行组装,以实现一个富客户端应用.本文提出的中间件中运行的富客户端应用即是以这种富客户端构件组装作为基本应用形态.文献[12,13]提出了一种类似本文的 MVC 的富客户端构件模型,但文献[12,13]中 MVC 构件模型的目的在于对已有的其他种类构件进行封装,而本文则从富客户端体系结构风格出发推演出 MVC 构件的必要性,其目的在于通过 MVC 构件模型分解富客户端的自适应问题.在文献[14,15]中,我们还进一步探讨了富客户端构件组装框架的设计与实现细节.

文献[3,16]最早关注于富客户端数据建模.文献[3]论述了富客户端应用中数据在客户端与服务器存储的需求,并给出一系列指导原则,用于帮助开发人员决定数据的存储位置[16].然而,这些工作更关注于如何通过数据模型以及其他 Web 模型直接生成富客户端应用;相反地,本文关注于如何在异构的运行环境中有效地访问数据.

文献[17]等工作关注于如何将应用的一部分移动到另一个运行环境中执行,以提高整个应用的性能.这类工作的基本思路与本文类似,但是这些工作主要面向服务器集群,其网络中不同节点上具有相同的应用运行环境.然而在富客户端场景中,服务器与客户端的运行环境并不相同,需要在服务器模拟客户端的运行环境并同步应用上下文,这正是本文工作的难点所在.与之类似,文献[18−21]虽然各自提出了面向浏览器-服务器体系结构的应用切分移动机制,但这些工作仍然通过文献中定义的私有模型或语言统一浏览器-服务器的运行环境;相反,本文直接使用 Web 标准的富客户端开发 JavaScript 语言,因此具备更好的适用性.

文献[8,9]提供将为 PC 屏幕设计的 HTML 页面自动适配到手机屏幕的方法,但这些工作主要针对服务器传输的 HTML 页面.如文中所述,富客户端应用的视图与传统 Web 应用的 HTML 页面的不同在于:富客户端视图是在运行时刻根据服务器数据动态生成,根据用户操作动态变化的.本文提供的方法不基于 HTML 页面,而是基于动态生成 DOM 树,并提供监视 DOM 变化随时重分割的机制,可以更好地适应富客户端视图适配的应用场景.

6 结束语

本文提出了一种富客户端运行环境自适应中间件.我们首先提供一种符合富客户端 MVC 体系结构风格的构件模型及其支撑机制作为中间件的基础;通过 MVC 构件模型,我们有效地将富客户端运行环境自适应问题分解为富客户端数据模型、控制器、视图所面临的异构存储、计算、显示环境的自适应问题;针对 3 个子问题,我们将多存储介质数据访问机制、控制器函数服务器端运行机制、视图分割与屏幕适配机制集成到中间件中.3 种机制的协作,使得富客户端应用得以适应其运行环境;最后,我们通过一组实验验证了中间件具备良好的性能与开销,同时,各机制确实可以帮助富客户端适应异构的运行环境.我们下一阶段将重点关注进一步完整几种自适应机制,重点考虑如何让数据访问机制支持复杂数据查询以及在控制器函数迁移机制中符合自动检测判断客户端函数的可迁移性.

References:

[1] Zhao Q, Liu XZ, Huang JY, Huang G. A browser-based middleware for service-oriented rich client. In: O'Conner L, ed. Proc. of the 2010 Int'l Conf. on Service Sciences. Washington: IEEE Computer Society, 2010. 22−27. [doi: 10.1109/ICSS.2010.14]

[2] Bozzon A, Comai S. Conceptual modeling and code generation for rich Internet applications. In: Wolber D, *et al*., eds. Proc. of the 6th Int'l Conf. on Web Engineering. New York: ACM Press, 2006. 353−360. [doi: 10.1145/1145581.1145649]

[3] Preciadol JC, Linajel M, Comai S, Sanchez-Figueroal F. Designing rich internet applications with Web engineering methodologies. In: Huang SH, *et al*., eds. Proc. of the 9th IEEE Int'l Workshop on Web Site Evolution. Washington: IEEE Computer Society, 2007. 23−30. [doi: 10.1109/WSE.2007.4380240]

[4] Zhao Q, Liu XZ, Chen XR, Huang JY. Towards a data access framework for service-oriented rich clients. In: Proc. of the 2010 IEEE Int'l Conf. on Service-Oriented Computing and Applications. Washington: IEEE Computer Society, 2010. 1−8. [doi: 10.1109/SOCA.2010.5707150]

[5] Mahdavi M, Shepherd J. Enabling dynamic content caching in Web portals. In: Martin DC, ed. Proc. of the 14th Int'l Workshop on Research Issues in Data Engineering. Washington: IEEE Computer Society, 2004. 129−136. [doi: 10.1109/RIDE.2004.1281712]

[6] Russsel SJ, Norvi P. Artificial Intelligence: A Modern Approach. Prentice-Hall, 1995.

[7] CNNIC. China Internet Development Report (2011). Beijing: Electronic Industry Press, 2011.

[8] Hua ZG, Xie X, Liu H, Lu HQ, Ma WY. Design and performance studies of an adaptive scheme for serving dynamic Web content in a mobile computing environment. IEEE Trans. on Mobile Computing, 2006,5(12):1650−1662. [doi: 10.1109/TMC.2006.182]

[9] Chen Y, Ma WY, Zhang HJ. Detecting Web page structure for adaptive viewing on small form factor devices. In: Proc. of the 12th Int'l Conf. on World Wide Web. New York: ACM Press, 2003. 225−233. [doi: 10.1145/775152.775184]

[10] Yu J, Benatallah B, Casati F, Daniel F. Understanding mashup development. IEEE Internet Computing, 2008,12(5):44−52. [doi: 10.1109/MIC.2008.114]

[11] Daniel F, Yu J, Benatallah B, Casati F, Matera M. Understanding UI integration: A survey of problems, technologies, and opportunities. IEEE Internet Computing, 2007,11(3):59−66. [doi: 10.1109/MIC.2007.74]

[12] Yu J, Benatallah B, Saint-Paul R, Casati F, Daniel F, Matera M. A framework for rapid integration of presentation components. In: Williamson C, *et al*., eds. Proc. of the 16th Int'l Conf. on World Wide Web. New York: ACM Press, 2007. 923−932. [doi: 10.1145/1242572.1242697]

[13] Yu J, Benatallah B, Casati F, Daniel F, Matera M, Saint-Paul R. Mixup: A development and runtime environment for integration at the presentation layer. In: Luciano B, *et al*., eds. Proc. of the 7th Int'l Conf. on Web engineering. Berilin/Heidelberg: Springer-Verlag, 2007. 479−484. [doi: 10.1007/978-3-540-73597-7_40]

[14] Huang G, Zhao Q, Huang JY, Liu XZ, Teng T, Zhang Y, Yuan HG. Towards service composition middleware embedded in Web browser. In: Proc. of the Cyber-Enabled Distributed Computing and Knowledge Discovery. 2009. 93−100. [doi: 10.1109/CYBERC.2009.5342195]

[15] Zhao Q, Huang G, Liu XZ, Huang JY, Mei H. A Web-based mashup environment for on-the-fly service composition. In: Lee J, *et al*., eds. Proc. of the 2008 IEEE Int'l Symp. on Service-Oriented System Engineering. Washington: IEEE Computer Society, 2008. 32−37. [doi: 10.1109/SOSE.2008.9]

[16] Huang JY, Liu XZ, Zhao Q, Ma JZ, Huang G. A browser-based framework for data cache in Web-delivered service composition. In: Proc. of the 2010 IEEE Int'l Conf. on Service-Oriented Computing and Applications. Washington: IEEE Computer Society, 2010. 1−8. [doi: 10.1109/SOCA.2010.5707138]

[17] Tilevich E, Smaragdakis Y. J-Orchestra: Automatic Java application partitioning. In: Magnusson B, ed. Proc. of the 16th European Conf. on Object-Oriented Programming. London: Springer-Verlag, 2002. 178−204. [doi: 10.1007/3-540-47993-7_8]

[18] Cooper E, Lindley S, Wadler P, Yallop J. Links: Web programming without tiers. In: de Boer FS, *et al*., eds. Proc. of the 5th Int'l Conf. on Formal Methods for Components and Objects. Berilin/Heidelberg: Springer-Verlag, 2006. 266−296. [doi: 10.1007/978-3-540-74792-5_12]

[19] Chong S, Liu J, Myers AC, Qi X, Vikram K, Zheng LT, Zheng X. Building secure Web applications with automatic partitioning. Communications of the ACM, 2009,52(2):79−87. [doi: 10.1145/1461928.1461949]

[20] Yang F, Gupta N, Gerner N, Qi X, Demers A, Gehrke J, Shanmugasundaram J. A unified platform for data driven Web applications with automatic client-server partitioning. In: Williamson C, *et al*., eds. Proc. of the 16th Int'l Conf. on World Wide Web. New York: ACM Press, 2007. 341−350. [doi: 10.1145/1242572.1242619]

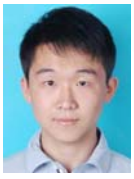
[21] Wohlstadter E, Li P, Cannon B. Web service mashup middleware with partitioning of XML pipelines. In: Proc. of the 2009 IEEE Int'l Conf. on Web Services. Washington: IEEE Computer Society, 2009. 91−98. [doi: 10.1109/ICWS.2009.102]

赵祺(1984－),男,北京人,博士,CCF 会员,主要研究领域为服务计算,Web 技术.
E-mail: zhaoqi06@sei.pku.edu.cn

黄罡(1975－),男,博士,教授,博士生导师,CCF 会员,主要研究领域为软件工程,软件中间件.
E-mail: hg@pku.edu.cn

刘譞哲(1980－),男,博士,副教授,主要研究领域为服务计算,Web 技术.
E-mail: liuxzh@sei.pku.edu.cn

梅宏(1963－),男,博士,教授,博士生导师,中国科学院院士,CCF 会士,主要研究领域为软件工程,系统软件.
E-mail: meih@pku.edu.cn



王旭东(1990－),男,学士,主要研究领域为服务计算,Web 技术.
E-mail: wangxd10@sei.pku.edu.cn