《模型-视图-视图框架模型基于多任务的改进方案》

绪论 目录:

1.JavaScript简介 2.JavaScript单线程异步 3.JavaScript异步产生的问题,以及现有解决方案 **1 JavaScript**的简介

JavaScript是一个轻量级解释型向对象但却将函数视为语言中一级公民的编程语言。JavaScript同时支持基于原型的面向对象编 程、命令式编程风格和函数式编程风格,是一种基于原型的、多范式的动态脚本语言。

JavaScript由网景公司雇佣的Brendan Eich实现。它的名字借鉴了当时大行其道的Java并且命名为JavaScript,基本语法借鉴C语言 与Java,数据结构借鉴了Java,函数借鉴了Scheme和Awk,将函数视为一等公民,并且引入了闭包概念。原型链的继承模式借鉴 了Self语言。正则表达式则借鉴了Perl。一些函数和方法借鉴了Python。但是为了保持简单,语言本身缺少了一些关键功能,比如 命名空间、模块等。但是因为语言的动态特性,这些语言都是可以利用现有的功能实现的。所以也造就了学习别的语言通常是学习 如何使用语言的功能,而学习JavaScript通常是学习各种解决问题的模式。

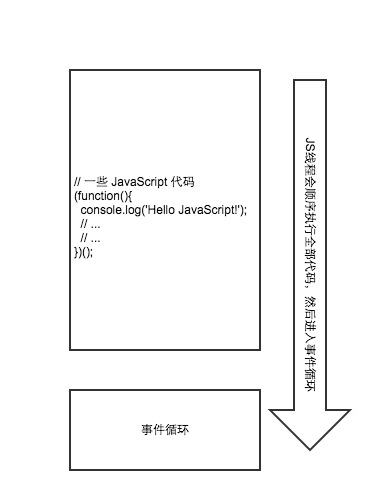
JavaScript最常被用在编写网页脚本上,但是随着Google的V8引擎的发布,JavaScript的效率被瞬间提高了降级20倍,并且接近原 生代码的执行速度。现在JavaScirpt已经别运用在了各个领域当中,比如用于开发移动智能的应用、桌面计算机上的应用、服务器 上的应用、游戏引擎中的脚本,跟甚至跟随这Intel进入了嵌入式应用的领域。比起Java,JavaScript最接近当年Java对世人许下的 承诺:“一处编写,到处运行”。

**2 JavaScript**的单线程与异步事件驱动 **2.1 JavaScript**单线程

JavaScript的单线程是指JavaScript运行时有且只有一个线程在执行JavaScript代码。正因为JavaScript的运行时是单线程的,一旦 被阻塞将会阻塞整个JavaScript引擎。所以JavaScript非常多的API被设计成了非阻塞(no-blocked)的模式,例如AJAX等。当 JavaScript引擎执行到异步非阻塞的AJAX时并不会阻塞等待AJAX请求的结果,而是继续执行接下来的代码。当AJAX请求返回结果 时,会产生一个事件压入JavaScript引擎的消息队列中,最终在事件循环中执行事件的处理的操作。

所以在代码中马上获取异步非阻塞的AJAX是不可能的。(注:虽然大多数浏览器都支持同步的AJAX,但是因为同步的AJAX会因为 等待请求返回而使得整个web页面失去响应,所以绝大多数规范中都命令禁止使用同步的AJAX)。

JavaScript线程会逐行执行所有的JavaScript代码后进入事件循环,通过轮询的方式等待以及处理事件。



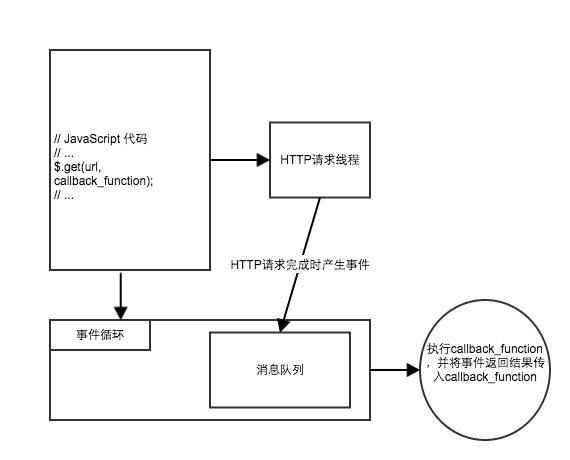
**2.2** 异步事件驱动

正如前面所说,JavaScript的单线程是指JavaScript的运行时有且只有一条线程在执行JavaScript代码。但是我们的浏览器并不是单 线程的,也不可能是单线程的。以Webkit为例,一个页面有JavaScript引擎线程、界面渲染线程、浏览器事件触发线程、Http请求 线程等线程。浏览器很多行为被设计成了异步的,例如,点击鼠标事件,定时器事件,以及XMLHttpRequest的回调等。当一个产 生一个异步事件了以后,它就被会压入一个消息队列。然后由事件循环(Event Loop)中对消息队列进行轮询并进行事件处理事 件。

以AJAX为例,当通过AJAX发出了一个请求,浏览器会开一个线程去请求相应的数据,当请求完成了以后会产生一个事件压入消息 队列。在事件循环中处理AJAX,执行回调并且将AJAX请求返回的结果作为参数传入回调函数。

**3 JavaScript**异步产生的问题,以及现有解决方案

**3.1** 回调地狱(**Callback Hell**) 回调地狱是指在编写JavaScript异步代码时产生的一个导致可维护差的问题。其中一个典型的例子是多重嵌套回调。





这种嵌套的回调函数难以理解,开发人员需要仔细分析哪些代码用于应用的业务逻辑,而哪些代码处理异步函数调用的,代码结构

支离破碎。错误处理也分解了,我们需要在各个地方检测错误的发生并作出相应的处理。

**3.2** 现有解决方案 由于回调地狱产生的非常多的现有解决方案:

**3.2.1 Promise**

Promise最初在E语言中被提出,它是基于并列/并行处理设计的一种编程语言。Promise是一种抽象异步处理对象以及对其进行各种 操作的组件。Promise也是一种同步函数与异步函数联系和通讯的方式。可以让异步函数返回值,并且能够抛出异常。

**3.2.2 Async**

Async主要解决异步的迭代难题以及异步函数工作次序的的难题,主要在Node.js上,最常见的是做IO操作上。

**3.2.3 Step**

Step是极简主义的工作流库。相比Async,Step的实现非常简单,功能也十分单一却十分实用。

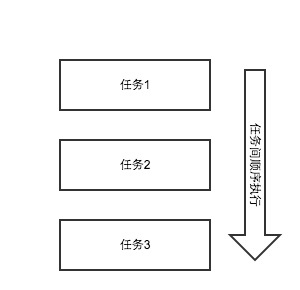
**2** 需求分析

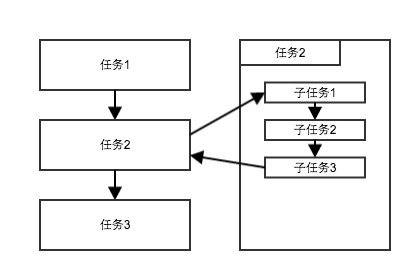
上一章已经介绍过JavaScript回调在工程应用产生的问题,同时也介绍了几种解决问题的方 案。但是各种各种方案都有其不足的地方,所以我们尝试设计一个更为优秀的解决方案来解决 不仅仅是几个异步操作顺序工作流这么简单的问题了,而是更高层次的抽象,将整个 JavaScript工程的控制流全部统一抽象出来,通过管道相连接,所有的操作都封装成一个任 务。

**1** 功能模型 **1.1** 任务间顺序执行

任务间严格按照顺序串行执行。

**1.2** 子任务 一个任务可以拥有子任务,住任务工作流会等待子任务全部按顺序执行完成后才继续执行主工



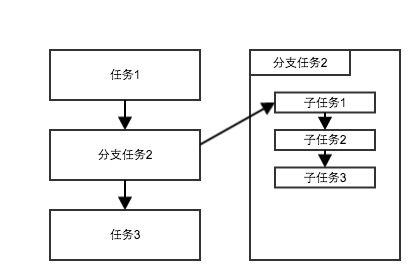


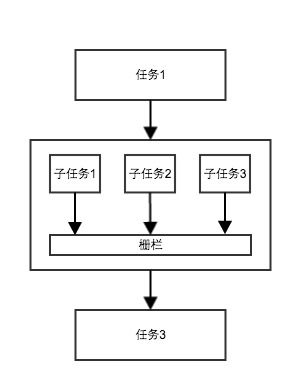
作流。

**1.3** 分支任务 分支类似于子任务,但是主任务工作流并不会等待分支任务执行完成,而是开启新的分支任务

后立即执行下一个任务。

**1.3** 栅栏 等待当多个任务都执行完成后(无顺序先后),才继续执行下一个任务。





# 3架构设计

模型 - 视图 - 控制器  
  
模型 - 视图 - 控制器（MVC）是一种软件架构模式实现用户界面。它把一个给定的软件应用程序分为三个相互关联的部分，以便分离从该信息提供给或从用户接受的方式信息的内部表示。[1] [2]  
  
内容  
  
    1概述  
        1.1组件  
        1.2相互作用  
    2使用Web应用程序  
    3历史  
    4参见  
    5参考  
    6外部链接  
  
概观  
  
与其他软件模式，MVC的表达到的问题在“解决方案的核心”，同时允许它适于每个系统。[3]的特殊的MVC架构可以与传统的描述在这里显著变化。[4]  
组件  
MVC组件的典型合作  
  
MVC中，模型，所述的中心部件捕获该应用程序的行为在其问题域，独立于用户界面的方面。[5]该模型直接管理的数据，逻辑和应用的规则。视图可以是信息，诸如图表或图中的任何输出表示;同一信息多种视图是可能的，如条形图管理和会计表格视图。的第三部分，所述控制器，接受输入，并将其转换为模型或视图的命令。[6]  
互动  
  
除了将应用程序划分成3种成分中，模型 - 视图 - 控制器设计定义它们之间的相互作用。[7]  
  
    控制器可以发送命令到模型以更新模型的状态（例如，编辑文档）。它也可以发送命令到它的相关联的视图改变模型的视图的呈现（例如，通过一个文档滚动）。  
    所检索由控制器，并在视图中显示的模型存储数据。每当有一个变化的数据是由控制器进行更新。  
    视图从它使用，以产生输出的表示给用户的模型请求信息。  
  
在使用Web应用程序  
  
虽然最初开发用于桌面计算，模型 - 视图 - 控制器已被广泛采用，作为在主要的编程语言万维网应用的架构。几个商业和非商业的Web应用程序框架已创建的执行模式。这些框架各有不同，解释，主要的方式，MVC的职责是在客户端和服务器之间进行分配。[8]  
  
早期的Web MVC框架，采取了瘦客户机的办法，放在了几乎整个模型，视图和控制器逻辑在服务器上。在这种方法中，客户端发送任何超链接的请求或形式输入到控制器，并然后接收从视图的完整的和更新的网页（或其他文件）;该模型完全存在于服务器上。[8]作为客户端技术已经成熟，框架，如AngularJS，Ember.js，JavaScriptMVC和骨干已创建允许MVC组件部分在客户机上执行（另见阿贾克斯）。  
历史  
  
MVC是一家在图形用户界面的早期发展的开创性的见解，并描述和实施其职责方面的软件构造的第一途径之一。[9]  
  
特里夫Reenskaug引入到MVC的Smalltalk-76，而来访的Xerox PARC的[10] [11] 20世纪70年代。在20世纪80年代，吉姆阿尔索夫和其他实施了Smalltalk的-80类库版本的MVC。只是到了后来，在对象技术杂志1988年的一篇文章，说的MVC表示为一个总的概念。[12]  
  
MVC模式已经随之进化，[13]从而产生变种，如HMVC，MVA，MVP，MVVM，和其他人适合模型视图控制器不同的语境。

# Model View ViewModel

模型视图模型（MVVM）是一种架构模式进行软件开发。  
  
MVVM是Martin Fowler的演示模型设计模式的变体。[1] [2]象福勒的演示模式，MVVM抽象视图的状态和行为。[1]然而，尽管呈现模型抽象的视图（即创建一个视图模型）在不依赖于特定的用户界面平台的方式，MVVM开发在微软专门通过接口 - 利用Windows演示基础（WPF）（微软的.NET图形系统）和功能，以简化用户的事件驱动编程的Silverlight（WPF的互联网应用导数）。[1]  
  
约翰Gossman，微软的WPF和Silverlight的建筑师之一，MVVM宣布他在2005年的博客。  
  
MVVM和演示模型无论是从模型 - 视图 - 控制器模式（MVC）获得。 MVVM方便的图形用户界面的发展的一个分离（无论是作为标记语言或GUI代码）从业务逻辑或后端逻辑（数据模型）的发展。 MVVM的视图模型是一个值转换器; [3]，这意味着该视图模型负责暴露在这样​​一种方式，对象被容易管理和消耗从模型的数据对象。在这方面，认为模型比视图多个模型，并处理最如果不是全部的视图的显示逻辑。[3]视图模型还可以实现调解模式，组织访问后端逻辑周围的用例由视图支持。  
  
模型 - 视图 - 视图模型也被称为模型 - 视图 - 粘合剂，特别是在实现，不涉及.NET平台。 ZK（用Java编写的Web应用框架）和KnockoutJS（JavaScript库）使用模型 - 视图 - 粘合剂。[1] [4] [5]  
  
内容  
  
    1组件的MVVM模式  
    2基本原理  
    3批评  
    4参考  
    5外部链接  
  
的MVVM模式组件  
  
模型  
    模型是指无论是一个域模型，它代表了真实情况的内容（一个面向对象的方法），或者，表示的内容（数据为中心的方法）的数据访问层。[来源请求]  
  
视图  
    如在MVC和MVP模式，视图是用户界面（UI）。[进一步解释需要]  
  
视图模型  
    视图模型是公开的公共属性和命令视图的抽象。取而代之的是MVC模式的控制器，或MVP模式的主持人，MVVM有粘合剂。在视图模型，这个粘合剂介导的视图和数据粘合剂之间的通信。[澄清需要]视图模型已被描述为在模型中的数据的状态。[6]  
  
粘合剂  
    声明数据和命令绑定是隐含在MVVM模式。在微软溶液堆栈，粘合剂是一种标记语言，称为XAML。[7]的粘合剂释放被被迫写锅炉板逻辑同步视图模型和视图显影剂。当微软之外实现堆叠的声明式数据绑定技术的存在是一个关键推动的格局。[4] [8] [澄清需要]  
  
合理  
  
MVVM旨在利用数据绑定功能在WPF更好的方便视图层的发展，从图案的其余部分分离，通过从视图层移除几乎所有的GUI代码（“代码隐藏”）。[9]相反需要的用户体验（UX）开发人员编写GUI代码，他们可以使用该框架标记语言（如XAML），并创建数据绑定到视图模型，这是书面和应用程序开发人员维护。这种角色的分离使得互动设计师专注于UX需求，而不是业务逻辑编程。应用程序的层因此可开发在多个工作流为提高生产率。即使当一个开发人员的工作对整个代码库从模型视图的适当分离，更高效的用户界面通常是经常变化和根据最终用户的反馈开发周期的后期。  
  
MVVM模式尝试获得既由MVC提供的功能开发的分离的优点，同时利用数据绑定的优点和框架通过结合数据作为接近纯应用模型越好。[9] [10] [11] [需要澄清]它使用的粘合剂，视图模型，以及任何业务层“数据检查功能来验证输入的数据。其结果是，该模型和框架驱动尽可能多的操作的可能，消除或减少其直接操纵的视图应用逻辑（例如，代码隐藏）。  
批评  
  
的模式的批评来自MVVM创始人John Gossman自己，[12]谁指出，在实施MVVM的开销是“杀鸡用牛刀”的简单的UI操作。他还指出，对于较大的应用，推广视图模型变得更加困难。此外，他说明了数据非常大的应用程序绑定可以造成相当大的内存消耗。

管道（软件）  
维基百科，自由的百科全书  
这篇文章是关于软件管道一般。对于原执行贝壳，看到管道（UNIX）。  
  
在软件工程中，管道由一连串的处理元件（进程，线程，协程，函数等），设置成，使得每个元件的输出是下一个的输入;该名称是由类似于物理管道。通常缓冲一定量的连续元件之间提供的。在这些管道中流动的信息是通常的记录，字节或比特，和一个管道可以被称为滤波器的元流;这也被称为管道和过滤器的设计模式。连接元件进入管道类似于函数组合。  
  
狭义地讲，一个管道是线性的，单向的，尽管有时将术语适用于更一般的流动。例如，一个主要的单向管线可以具有在其他方向一些通信，被称为一个返回信道或反向信道，如在词法分析器劈或管道可以是完全双向的。与一个方向的树和向无环图的拓扑流动行为类似于（线性）管道 - 缺乏循环使得它们简单 - 并且因此可以被松散地称为“管道”。  
  
内容  
  
    1日实施  
    2 VM / CMS和MVS  
    3管线对象  
    4管线的图形用户界面  
    5其他注意事项  
    6参见  
    7笔记  
    8外部链接  
  
履行  
  
管线通常在一个多任务操作系统实现，通过启动所有的元素同时随着流程，并自动维修数据读取由每个过程与写的上游过程中的数据的请求 - 这可以被称为一个多道管道。在这种方式中，CPU将自然的处理由调度器之间切换，以便最大限度地减少它的空闲时间。在其他常见车型，元素被实现为轻量级线程或协程，以减少操作系统开销往往涉及流程。根据不同的操作系统时，线程可以直接由OS或由线程管理器调度。协程总是定于某种形式的协同程序经理。  
  
通常，读取和写入请求被阻塞操作，这意味着在源处理的执行，书写时，暂停，直到所有的数据可以被写入到目的过程中，以及，同样地，在目标处理的执行，通过阅读，被暂停，直到至少一些所请求的数据的可从源方法得到。这不能导​​致死锁，其中两个进程将无限期地等待对方响应，因为这两个过程中的至少一个将在此后不久有其请求的服务由操作系统，并继续运行。  
  
出于性能，实现管大多数操作系统都使用管道缓冲区，它允许源过程中提供更多的数据比目标的过程是目前能够或愿意接受。在大多数的Unices和类Unix操作系统，一个特殊的命令也可实现其潜在的更大，配置大小的管道缓冲区，通常被称为“缓冲”。这个命令可以是有用的，如果在目标过程比源过程显著慢，但是它无论如何需要使源进程可以尽快完成其任务。例如，如果源过程包括其中从CD读取的音频轨道和目的地过程包括其中压缩波形音频数据，如MP3格式的命令的命令。在这种情况下，缓冲整个轨道的管道缓冲区将允许CD驱动器更快地降速，并且使用户能够从驱动器中取出光盘之前，编码过程已经完成。  
  
可使用系统来实现，例如一个缓冲器命令调用用于读取和写入数据。浪费忙等待，可避免使用的设施，如投票或选择或多线程。  
VM / CMS和MVS  
  
CMS管道是管道想法VM / CMS和MVS系统的端口。它支持更复杂的管道结构比Unix shell中，与步骤服用多种输入流，并产生多个输出流。 （这种功能是由Unix内核的支持，但很少使用的程序，因为它使得复杂的语法和阻塞模式，虽然有些炮弹通过做任意的文件描述符分配支持它）。由于IBM的大型机操作系统的不同性质，它实现了内部CMS管道而在Unix中是独立的外部程序很多步骤，也可以要求其功能独立的外部程序。此外，由于文件在IBM大型机上的面向记录的性质，管道在面向记录的，而不是面向流的方式。[编辑]操作  
管道对象  
  
旁字节流基管线，也有对象管道。在一个对象的管道，处理元件的输出对象，而不是文字。 Windows PowerShell中包含了PowerShell运行时内部功能之间传输的.NET对象的内部对象的管道。频道，在凌波编程语言，IPython的ipipe扩展发现是这个比喻的其他例子。  
管线的GUI  
  
图形化的环境中，如RISC OS和ROX桌面还利用管道。而不是提供一个包含文件管理器让用户指定的程序应该写数据保存​​对话框中，RISC OS和ROX提供了保存包含的图标（和一个字段指定的名称）对话框。目的是通过拖放图标指定。用户可以在任何地方放下图标已经保存的文件可能被丢弃，包括到其他程序的图标。如果图标拖放到一个程序的图标，它的加载和，否则将被保存在内容上的新程序的标准输入流传递。  
  
例如，用户浏览万维网可能会遇到他们想要编辑和重新上传一个.gz的压缩图像。使用GUI管道，他们可以拖动链接到他们去归档程序，拖动代表所提取的内容，以他们的图像编辑器的图标，编辑，打开另存为对话框，然后拖动图标到他们上传软件。  
  
从概念上讲，这种方法可与常规的保存对话框使用，但是这将需要用户的方案，以具有在可导航到文件系统明显和易于访问的位置。在实践中，这通常是不是这种情况，那么GUI管道是罕见的。  
其他注意事项  
  
名称'管道'来自粗略类比与该管道的物理管道通常[1]允许信息流只在一个方向，像水经常流动的管道。  
  
管道和过滤器可以被看作是功能的编程的一种形式，使用字节流数据对象;更具体地，它们可以被看作是单子用于I / O的一种特殊形式。[2]  
  
管线的概念也是中央对茧的Web开发框架或任何Xproc的（W3C的标准）的实现，它允许一个源流最终显示之前进行修改。  
  
这种模式鼓励使用文本流作为节目的输入和输出。这依赖于文字具有创建图形壳时文本程序进行核算。

# 概要设计

# 详细设计

# 系统实现

# 系统测试

# 总结