# 绪论

1.1 研究背景与意义

随着云计算、物联网、移动互联、社交媒体等新兴信息技术和应用模式的快速发展，推动人类社会不断地向大数据时代迈进。2010年，全球数据量就已经跨入了ZB时代，据IDC预测，至2020年全球将拥有35ZB的数据量，大量数据实时地影响着我们的工作、生活，甚至国家经济、社会发展，大数据时代已经到来。大数据具有数据量巨大、数据类型多样、流动速度快和价值密度低的特点，大数据技术为我们分析问题和解决问题提供了新的方法和思路，其研究已经成为热点。一般意义上来讲，对大数据就是利用现有的理论知识以及各种技术手段和工具都很难再可以接受的时间之类完成数据分析计算，而整体都呈现价值巨大的海量复杂的数据集合体。与此同时，这些巨大的价值量往往都是隐藏在海量的数据当中，变现出了价值密度低、分布极其不规律、信息隐藏深度极深、发现价值极其困难等特点，这些特征必然会为大数据的研究和分析带来前所未有的挑战和机遇。

大数据的分析计算模式主要分为批量计算（batch computing）、流式计算（stream computing）、交互式计算（interactive computing）、图形计算（graph computing）等等。其中批量计算和流式计算这两种计算模式不管是在学术界还是在工业界都是主要的研究模式，同时各自都有广泛的大数据应用场景。其中批量计算是一种适用于大估摸并行批量处理作业的分布式计算模式，也就是我们大家都十分熟悉的MapReduce计算模式。MapReduce的简单易用性使其成为目前大数据处理最成功的主流并行计算模式。在开源社区的努力下，开源的Hadoop系统目前已成为较为成熟的大数据处理平台，并已发展成一个包括众多数据处理工具和环境的完整的生态系统。目前几乎国内外的各个著名IT企业都在使用Hadoop平台进行企业内大数据的计算处理。此外，Spark系统也具备批处理计算的能力。而流式计算是一种高实时性的计算模式，需要对一定时间窗口内应用系统产生的新数据完成实时的计算处理，避免造成数据堆积和丢失。很多行业的大数据应用，如电信、电力、道路监控等行业应用以及互联网行业的访问日志处理，都同时具有高流量的流式数据和大量积累的历史数据，因而在提供批处理计算模式的同时，系统还需要能具备高实时性的流式计算能力。流式计算的一个特点是数据运动、运算不动，不同的运算节点常常绑定在不同的服务器上。Facebook的Scribe和Apache的Flume都提供了一定的机制来构建日志数据处理流图。而更为通用的流式计算系统是Twitter公司的Storm、Yahoo公司的S4以及Apache Spark Steaming。

大数据时代下的流式计算呈现出了鲜明的高带宽、低时延的应用需求，传统的流式计算平台的构造往往是建立在传统的数据库的基础之上的，难以满足大数据流式计算的需求。如何构建一个低时延、高带宽、持续可靠、长期运行的大数据流式计算系统成为了当前亟待解决的问题。Redis这种基于内存计算的、可进行数据持久化的Key-Value存储系统的诞生，为大数据流式计算带来了很好的一个解决方案。Redis数据库最初是为了解决像SNS类网站在数据存取过程中的实时性等刚性需求的，而传统的关系型数据库越来越难以胜任了，这也使得redis这种数据库也越来越受到人们的关注。如今redis数据库已经得到了广泛的应用，不论是在高速缓存系统中，还是在海量文件的实时检索中，甚至是在如何如荼的各种推荐系统中，redis都起着中心地位的作用。Redis的基于内存的数据计算和高效的数据存储策略也能够很好的满足实时流计算问题中的低时延的刚性需求。因此，研究redis的内存计算以及存储策略并将其运用到实时流式计算模型中具有重要的意义和实用价值。

在流式数据处理中，它不像传统的批量数据处理，它无需事先存储数据，无法确定数据是什么时候到来以及按什么顺序到来，因此，不需要事先对流式数据进行存储，而是当流动的数据到来后在内存中直接进行数据的实时计算和分析。就像我们熟悉的Twitter的Storm、Yahoo的S4就是典型的流式数据处理框架，数据在任务拓扑中被计算，最后输出有价值的信息。目前这些流行的流式处理框架都有一个共同的缺点就是，没有一个方便的能够快速根据业务构建数据任务的拓扑计算流程，也就是我们所说的计算流（flow），同时也缺乏数据的流化功能。Node-red是基于Node.js的，可视化流程编辑框架，它允许开发人员仅仅使用一个基于浏览器的可视化界面流程编辑器来完成设备、服务器以及API应用的连接。Node-red本身是IBM Emerging Technology团队创建的一个新型开源工具，它允许用户通过组合各种部件来编写应用程序。这些部件可以是硬件设备、Web API或者是在线服务。Node-red被广泛用于物联网领域，实现数据的流式传输。在node-red中从数据的接入，到数据的解析分析，最后到结果的输出都是通过各种各样的节点来完成的，IBM Emerging Technology团队在开发这个工具的时候只引入了少量的大众化的节点，比如常用的http节点、tcp节点、udp节点、debug等数据输入输出节点，还有一些用于数据分析的节点比如sentiment节点，还有一些用于访问存储设备的节点；node-red除了原始已经提供的这些节点外，还运行用户自己按照开发原则开发自己需要的节点。为了能够充分利用node-red的可视化流程编辑的直观性，结合redis数据库的内存计算的特点，探索开发适应于流式数据分析的数据输入输出和数据处理节点，这对流式数据分析有着重要的实际意义。

1.2 国内外研究现状

1.2.1实时流数据处理模型的研究应用现状

大数据时代下的数据处理主要的两种方式就是实时流式处理和批量处理。实时流数据处理主要适合于那些无需事先进行数据存储，可以直接进行数据分析处理，实时性要求比较严格，但数据的准确度要求比较宽松的应用场景。而对于传统的批量数据处理，首先要进行数据的存储，然后再对存储的静态数据进行集中或者分布式计算。目前，对于传统的批量数据处理模型的技术和研究成果已经相对成熟了，最初有Google公司的MapReduce并行编程模型的提出，再有后来在开源社区的努力下开发的Hadoop系统为代表的批处理系统，都已经是稳定而高效的批处理系统。而对于流式数据处理模型的研究仅仅处于一个初级阶段，在早期关于流式数据的研究也主要集中在以数据库为中心而开展的，主要是研究了数据计算的流式化，数据规模也比较小，数据对象也比较单一，很难适应在大数据时代下流式数据处理所呈现出来的新特性。因为，在新时期的流式数据主要呈现出实时性、突发性、无序性等特点，对新的流式计算系统就有了更高更严格的要求。

在国外，Yahoo推出了S4流式数据处理系统，随后在2011年，Twitter也推出了自己的流式数据处理系统Storm,还有就是近年来开源社区新兴的MOA（Massive Online Analysis）、Spark Stream都是流式处理系统，这在一定程度上推动了流式数据处理的发展和应用。但是像S4、Strom这样的流式数据处理系统在可伸缩性、容错性、数据吞吐量等方面存在着明显的不足，而对于MOA，Spark Stream这样的系统，虽然功能和API十分丰富，但是在稳定性和易用性上不尽如人意。所以，如果构建一个低延迟、高吞吐、易用且能持续可靠地运行的流式数据处理系统，是一个亟待解决的问题。

在国内，目前关于流式数据处理模型的研究还比较少，但目前国内主要有百度公司自主研发的Dstream和TM实时计算平台，在学术界主要是有一下关于流式数据挖掘算法的研究。但是，流式数据的可视化分析已经在很多场景得到了应用，比如各大银行都陆续建立的大屏监控系统，就是实时地监控银行的业务状况、系统运行状况、用户行为分析等，又比如政府网站群的监控，也是通过实时监控网站的访问数据，分析用户的行为。在这些应用的背后，如何建立一个高效、稳定、易于维护的实时处理模型显得尤为重要。

文献[1]，

文献[2]，

1.2.2 Node-red的研究应用现状

Node-red作为一种在物联网时代的新型产物，是一种用来快速搭建物联网应用程序的流式处理框架，在信息无处不在的时代，Node-red也越来越受到业界的关注和研究。

它是由IBM Emerging Technologies团队发起的一个开源项目，其中Nick O’Leary 和Dave Conway-Jones工程师为Node-red的设计和开发做出了巨大的贡献。2013年，Node-red以开源项目的形式被发布，经过一年的发展，Node-red已经拥有了一大批活跃的用户和开发人员。Node-red依然是一个新型科技，时至今日，但凡用过Node-red的制造商、实验人员和一大批大大小小的公司，都已经见证了Node-red极具价值的应用之处。

在国外，IBM公司率先将Node-red应用起来，Node-red被集成到IBM公司的最新云产品Bluemix上。通过Bluemix提供的云服务，用Node-red来建立和管理一个实例（也就是一个应用流程），就可以实现消息的推送服务。Node-RED 的使用，与 Bluemix 中简单的 Push 服务相结合，使整个流程变得非常简单，需要调整的部分也少得多。

在国内，目前也有很多智能设备制造公司在使用Node-red，可以很方便地通过Node-red节点来控制硬件设备的状态，比如拿Node-RED搭配Arduino，是一个快速原型化的好用工具.例如控制RPI的某根脚位去点亮LED。只要简单的拉四个node，串一串再写一点程序代码即可做到。因为Node-red还在进一步完善当中，原始开发的节点可能很难满足实际的需求，所以，我们在运用Node-red来管理数据流程的时候，还需要自己开发需要的功能节点。在这一点上，目前在不少银行的业务监控系统中引入了redis的访问节点。

1.2.3 Redis的研究应用现状

Redis作为存储系统之中的后起之秀，由于其数据结构丰富、基于内存计算、支持网络又可进行数据持久化等特点，迅速为许多企业和开发者所爱戴。不论是在学术界还是在工业界，对Redis的研究都从未停止过。

Redis是由Salvatore Sanfilippo为实时统计系统LLOOGG量身定制的一个数据库，在2009年的时候将Redis开源发布，并开始于另外一位Redis代码贡献者Pieter Noordhuis一起继续Redis的开发，知道现在。随着Redis内存数据库的发布，短短几年内的时间就拥有了一个庞大的用户群体。在国外，像GitHub、Viacom、Pinterest等都是Redis的用户，Github利用Redis集群，来统计用户项目跟进状况。而在国内，新浪微博研究了Redis数据库的源码，搭建了有号称史上最大的Redis集群，实现了传统的SQL数据库难以实现的计数分析（counting）、反向缓存（reverse cache）、top 10 list等功能。近年来，也有不少银行，在自己的实时数据监控平台引入了Redis数据库，实现了数据的实时处理和分析，还有就是随着国家电子政务系统的逐渐推行，不少的地方政府也在自己的数据中心监控系统中引入了Redis数据库，来实现数据实时计算和处理。

文献[3]，

1.3 主要工作和研究内容

本文对大数据背景下流式数据处理过程中所遇到的挑战和难题进行了研究分析，详细研究了Node-red流式处理框架的编程模型和消息推送机制，Redis数据库的存储原理及其基于内存计算的原理。设计了一种新的基于Node-red的流式管理和Redis的内存计算的流式数据处理模型，并通过实现网站访问实时监控系统来验证了该模型的可行性。主要工作内容如下：

1. 本文对当下流行的几款流式处理框架Storm、Spark stream与Node-red流式管理框架进行了对比分析，详细阐述了各自的编程模型，同时对结合MQTT详细阐述了消息推送机制为什么适合流式数据处理。
2. 对Redis数据库做了深入研究，尤其是Redis的存储原理以及内存计算原理，还对Redis的Pub/Sub机制做了深入研究，阐述了在流式数据处理中为什么选择Redis作为数据计算和中间数据缓存的工具。
3. 在研究分析了流式数据的特点和流式数据处理的基本原理后，结合Node-red的编程模型和消息推送机制，设计了一种新的基于Node-red的流式管理和Redis的内存计算的流式数据处理模型。由于原始的Node-red缺乏对redis数据库的访问节点以及redis的pub/sub节点，重新设计了新的数据输入、输出节点以及数据处理节点，并成功发布到Node-red框架当中，实现数据的流式处理，和数据流的管理。
4. 本文最后还对设计好的流式处理模型，加以了应用验证。使用该模型对某政府网站的访问数据进行实时监控分析，设计了一套数据监控系统，包括了数据的采集，数据分析和处理，以及最后的数据可视化展示，并对结果进行了有效性分析。实现了从模型设计到模型应用的全过程。

1.4 结构安排

本论文共分为六章，其章节结构安排如下：

1. ，绪论，首先介绍了本论文的研究背景和意义，通过阅读大量相关文献和论文资料，总结了国内外流式数据处理模型的研究现状，以及Node-red、Redis的研究应用现状。然后简单的介绍了本论文的主要研究内容和全文的章节结构安排。
2. ，实时流数据处理的理论基础和技术，本章详细介绍了目前几个主流的流式处理框架的编程模型，有Storm的流式编程模型、Spark stream的流式编程模型、Node-red的编程模型以及他们各自的特点和不足之处。还介绍了MQTT协议以及在消息推送服务中的应用。最后还介绍了Node-red的异步非阻塞模式与事件驱动机制，以及Redis的存储技术和Pub/Sub机制。
3. ，基于Node-red和Redis的实时流数据处理模型的设计，本章首先对在实际场景中的流式数据处理应用做了详尽的需求分析，然后对模型的总体架构做了详细设计，最后对于Node-red中原本缺少的用于流式数据的输入和输出节点以及数据处理节点做了重新设计，并将设计的各个节点重新部署到Node-red框架当中，使其成为一个能够胜任流式数据处理的完整框架。
4. ，基于node-red与redis的实时流数据处理模型在网站访问监控方面的应用，本章主要是对设计的新模型加以应用，以此来验证模型的可行性。为此设计了一个实时网站访问的监控系统，数据处理就用到了我们设计好的流式数据处理模型，将数据处理的结果输出到前端页面Board做可视化展示。本章详细阐述了系统的功能，各个模块的设计与实现以及系统的展示。
5. ，系统测试与性能分析，这一章是整个模型的测试环节，主要是分析了模型对流式数据的处理能力，并与Storm和Spark stream流式处理系统在吞吐量和时延上做了对比分析。
6. ，全文总结与展望，是对本论文的主要工作进行最后总结，并对后续工作做了一些说明。

# 实时流数据处理的基础理论和技术

2.1 实时流数据处理的编程模型

2.1.1 Storm的编程模型

Storm是一个分布式的，可靠的，容错的数据流处理系统。它会把工作任务委托给不同类型的组件，每个组件负责处理一项简单特定的任务。Storm集群的输入流由一个被称作spout的组件管理，spout把数据传递给bolt， bolt要么把数据保存到某种存储器，要么把数据传递给其它的bolt。你可以想象一下，一个Storm集群就是在一连串的bolt之间转换spout传过来的数据。

在Storm编程框架中，只要不是人为干预，Storm就一直实时不断地进行数据处理。值得注意的是，并不是Storm去处理，而是它可以将我们程序的很多jar包，业务程序，同时放到不同的服务器中并发的运行，最终得到的结果就是不同系统的海量数据就会分散到不同的服务器中并发的进行处理，负载能力很强。所以真正进行数据处理的是我们写好的数据处理程序，Storm的强大作用之一就是它为这些程序提供了运行温床，将应用程序上传到Storm集群中，在多台机器上并发运行，这样就可以扩展程序的负载处理能力，实现流式计算。

接下来我们看一下Storm的集群组件，Storm集群主要有两个角色扮演者，一个就是Nibus，是集群的主节点，主要负责任务分配、响应客户端提交topology请求以及任务失败的调度。另外一个就是Supervisor，是集群的从节点，主要负责启动、停止业务逻辑组件程序进程。主从节点之间通过zookeeper集群进行连接，主从节点之间是fail-fast（java的一种错误机制）、无状态的，主从节点的状态信息均保存到zookeeper中或者本地硬盘里。这样的好处就在于，哪怕是主节点kill掉了，storm会自动起一个备份主节点，因为无状态的关系，所以任意一个节点都可以充当Nimbus一角。下图展示了Storm集群的架构，这种设计使得Storm十分稳定。



Storm集群架构图

在[Storm](https://storm.apache.org/" \t "http://www.csdn.net/article/2015-03-09/_blank)中，先要设计一个用于实时计算的图状结构，我们称之为拓扑（topology），这也是Storm处理流式数据计算的核心编程模型。正如下图所示，这个拓扑将会被提交给集群，由集群中的主控节点（master node）分发代码，将任务分配给工作节点（worker node）执行。一个拓扑中包括spout和bolt两种角色，其中spout发送消息，负责将数据流以tuple元组的形式发送出去；而bolt则负责转换这些数据流，在bolt中可以完成计算、过滤等操作，bolt自身也可以随机将数据发送给其他bolt。由spout发射出的tuple是不可变数组，对应着固定的键值对。



Storm拓扑结构图

简单理解，Storm中的拓扑结构（topology）就是包含了数据源、逻辑处理组件的一个外在集合框架，使用storm可以定义一个topology里set多少个数据源组件，多少个逻辑处理组件。

2.1.2 Spark streaming的编程模型

Spark是一个类似于MapReduce的分布式计算框架，其核心是弹性分布式数据集，提供了比MapReduce更丰富的模型，可以在快速在内存中对数据集进行多次迭代，以支持复杂的数据挖掘算法和图形计算算法。Spark Streaming是一种构建在Spark上的实时计算框架，它扩展了Spark处理大规模流式数据的能力。

Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备。下图显示了Spark Streaming的整个流程。



Spark streaming 架构图

使用Spark Streaming编写的程序与编写Spark程序非常相似，在Spark程序中，主要通过操作RDD（Resilient Distributed Datasets弹性分布式数据集）提供的接口，如map、reduce、filter等，实现数据的批处理。而在Spark Streaming中，则通过操作DStream（表示数据流的RDD序列）提供的接口，这些接口和RDD提供的接口类似。图3和图4展示了由Spark Streaming程序到Spark jobs的转换图。





在图3中，Spark Streaming把程序中对DStream的操作转换为DStream Graph，图4中，对于每个时间片，DStream Graph都会产生一个RDD Graph；针对每个输出操作（如print、foreach等），Spark Streaming都会创建一个Spark action；对于每个Spark action，Spark Streaming都会产生一个相应的Spark job，并交给JobManager。JobManager中维护着一个Jobs队列, Spark job存储在这个队列中，JobManager把Spark job提交给Spark Scheduler，Spark Scheduler负责调度Task到相应的Spark Executor上执行。

Spark Streaming的另一大优势在于其容错性，RDD会记住创建自己的操作，每一批输入数据都会在内存中备份，如果由于某个结点故障导致该结点上的数据丢失，这时可以通过备份的数据在其它结点上重算得到最终的结果。

正如Spark Streaming最初的目标一样，它通过丰富的API和基于内存的高速计算引擎让用户可以结合流式处理，批处理和交互查询等应用。因此Spark Streaming适合一些需要历史数据和实时数据结合分析的应用场合。当然，对于实时性要求不是特别高的应用也能完全胜任。另外通过RDD的数据重用机制可以得到更高效的容错处理。

2.1.3 Node-red的编程模型

Node-Red是IBM公司开发的一个可视化的编程工具。它允许程序员通过组合各部件来编写应用程序。这些部件可以是硬件设备(如：Arduino板子)、Web API(如：WebSocket in和WebSocket out)、功能函数(如：range)或者在线服务(如：twitter)。Node-Red提供基于网页的编程环境。通过拖拽已定义node到工作区并用线连接node创建数据流来实现编程。程序员通过点击‘Deploy’按钮实现一键保存并执行。程序以JSON字符串的格式保存，方便用户分享、修改。Node-Red基于Node.js，它的执行模型和Node.js一样，也是事件驱动非阻塞的。理论上，Node.js的所有模块都可以被封装成Node-Red的一个或几个node。

2.2 Node的事件驱动和非阻塞机制

与Nginx的服务原理类似，Node采用事件驱动的运行方式。不过nginx式多进程单线程，而Node通过事件驱动的方式处理请求时无需为每一个请求创建额外的线程。在事件驱动的模型当中，每一个IO工作被添加到事件队列中，线程循环地处理队列上的工作任务，当执行过程中遇到来堵塞(读取文件、查询数据库)时，线程不会停下来等待结果，而是留下一个处理结果的回调函数，转而继续执行队列中的下一个任务。这个传递到队列中的回调函数在堵塞任务运行结束后才被线程调用。下图为Node的事件驱动原理图。



Node Async IO = CPS + Callback，这一套实现开始于Node开始启动的进程，在这个进程中Node会创建一个循环，每次循环运行就是一个Tick周期，每个Tick周期中会从事件队列查看是否有事件需要处理，如果有就取出事件并执行相关的回调函数。事件队列事件全部执行完毕，node应用就会终止。Node对于堵塞IO的处理在幕后使用线程池来确保工作的执行。Node从池中取得一个线程来执行复杂任务，而不占用主循环线程。这样就防止堵塞IO占用空闲资源。当堵塞任务执行完毕通过添加到事件队列中的回调函数来处理接下来的工作。

我们知道，IO操作无疑是耗时的，当服务器端接收到大量请求时，为每一个请求创建进程或线程的同时，也增加了额外的内存开销，也可能浪费更多的时间资源。由于Node.js是事件驱动的，于是它使用了事件循环来解决IO操作带来的瓶颈问题。在Node.js中，一个IO操作通常会带有一个回调函数，当IO操作完成并返回时，就会调用这个回调函数，而主线程则继续执行接下来的代码。下面我们说明一下Node.js异步非阻塞I/O的运行原理。

在Windows平台上，Node.js是直接利用Windows下的IOCP（I/O Completion Port）通常称为I/O完成端口来实现的，在IOCP的内部其实是利用了线程池的原理，这些线程是由Windows系统内核自动管理，不需要我们手动加以管理。而在Linux平台上，Node.js从v0.9.3版本以后，都是通过自行实现的线程池来完成异步非阻塞I/O的。由于这种跨平台的差异性，Node.js通过构建一个平台层架构libuv，来屏蔽平台的差异性，所有平台的兼容性问题都是由这一层来完成。下图就是Node.js的异步执行架构图。



以上就异步非塞模型和事件驱动机制进行了总结。而这个事件循环的机制并不仅仅是Node.js所独有的，并且Node.js的代码是单线程执行的，在面对大量并发请求的时候，Node.js是有着自己独特优势的，下图我们给出了Node.js的架构图。



从该架构图中可以看出，Node.js的底层有一个模块负责维护线程池，当一个IO请求发出的时候，Node.js的底层模块将新建一个线程来处理请求，完成后再将结果交还给上层。那么，当有多个请求的时候，Node.js的底层模块将利用尽可能少的线程来完成最多的任务，如果存在空闲的线程，它将继续被利用来做其他的事情，这对于前面说的针对每个请求开一个新的进程或线程而言，无疑“聪明”许多，也更加高效了。

2.3 Node-red可视化流式处理框架

2.3.1 Node-red的概述

Node-Red是IBM公司开发的一个可视化的编程工具。它允许程序员通过组合各部件来编写应用程序。这些部件可以是硬件设备(如：Arduino板子)、Web API(如：WebSocket in和WebSocket out)、功能函数(如：range)或者在线服务(如：twitter)。Node-Red提供基于网页的编程环境。通过拖拽已定义node到工作区并用线连接node创建数据流来实现编程。程序员通过点击‘Deploy’按钮实现一键保存并执行。程序以JSON字符串的格式保存，方便用户分享、修改。Node-Red基于Node.js，它的执行模型和Node.js一样，也是事件驱动非阻塞的，这一点在上一节关于Node的事件驱动和非阻塞机制已经作了详尽的阐述。理论上，Node.js的所有模块都可以被封装成Node-Red的一个或几个节点（node）。接下来，我们将详细地阐述如何搭建Node-red的环境，以及如何编写和管理自己的flow。

2.3.2 Node-red的编译与安装

由于本文要对Node-red的原始节点进行补充，增加新的节点使其适合流式数据的传输和计算。因此我们在安装Node-red的时候选择了源码安装。安装的平台选择了Ubuntu14.04长期支持版。具体的安装流程如下：

依赖的安装：

由于Node-red是基于Node.js的，所以在安装Node-red前必须先安装Node.js。这里推荐使用源码安装。

1. 在github上获取Node.js的源码：

$ sudo git clone https://github.com/nodejs/node.git

1. 修改目录权限：

$ sudo chmod -R 755 node

1. 使用./configure创建编译文件，并按如下命令安装：

$ cd node

$ sudo ./configure

$ sudo make

$ sudo make install

1. 验证是否安装成功：

在按照上面三步安装之后，执行node -v命令之后，结果如下图所示表示安装成功。



1. 检验npm（NodePackagedModule）是否安装成功：

由于新版的Node.js里面已经集成了npm，所以不需要另外单独安装npm，执行npm -v命令之后，结果如下图所示表示安装成功。



Node-red的安装：

因为我们要对Node-red的原始节点进行补充，所以这里我们必须选择源码安装。

1. 在github上获取Node-red的源码：

$ sudo git clone https://github.com/node-red/node-red.git

1. 安装Node-red：

$ cd node-red

$ npm install

1. 安装grunt-cli：

为了能够利用Node-red顺利的创建应用，这里我们还需要安装grunt-cli，并且不需要安装成全局模块。

$ npm install -g grunt-cli

1. 创建应用并运行Node-red：

$ grunt bulid

$ node red

1. 验证Node-red是否运行成功：

按照上述步骤安装结束之后，如果执行node red命令之后，控制台出现如下图所以表示安装成功：



2.3.3 Node-red的基本配置

安装好Node-red后，在Node-red的安装目录下有很多文件，从这些源码文件就可以看出Node-red的目录是十分清晰，各个模块的划分也是仅仅有条的。在这些文件当中，绝大部分文件用户都不需要关心，但是有几个重要文件我们需要尤其注意，详细了解这些文件的作用和实用方法，对我们后面开发新节点有很重要的帮助，首先我们来看一下Node-red的目录结构，下图展示了Node-red的目录结构。



Node-red目录结构图

下面简单地介绍一下各个目录文件存储的内容和作用：

（1）在/public目录下是一些关于Node-red本身的静态文件，包括资源文件、css样式文件、以及前端页面的html文件；

（2）/node-modules目录下面是一些外部依赖库，也就是Node-red需要的一些Node.js模块。

（3）/red目录下面就是真正的Node-red代码，主要是一些核心api、事件驱动程序、服务器端主程序、系统设计程序以及Node-red的入口程序等。

（4）/test目录下面主要是放了一些用于测试的Node以及flow；

（5）/nodes目录是一个极其重要的目录，Node-red中所有的节点都是存放在这个目录下的，包括各个节点的html和js文件，本文中重新设计的数据节点也会放在这个目录下。

（6）settings.js文件是整个Node-red的系统配置文件，该文件描述了启动的参数细节、端口和ip设置以及各个启动目录的设置。

了解了各个目录文件的作用之后我们再来介绍一下如何配置Node-red。

2.4 基于内存计算的数据库Redis

2.4.1 Redis数据库的概述

redis是一个key-value[存储系统](http://baike.baidu.com/view/51839.htm" \t "http://baike.baidu.com/_blank)。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list([链表](http://baike.baidu.com/view/549479.htm" \t "http://baike.baidu.com/_blank))、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些[数据类型](http://baike.baidu.com/view/675645.htm" \t "http://baike.baidu.com/_blank)都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

与其他key-value存储系统相比，Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

2.4.4 Redis数据库的储存原理

Redis存储机制分成两种Snapshot 和 AOF。无论是那种机制,Redis都是将数据存储在内存中。

Snapshot工作原理: 是将数据先存储在内存，然后当数据累计达到某些设定的伐值的时候，就会触发一次DUMP操作，将变化的数据一次性写入数据文件（RDB文件）。

AOF 工作原理: 是将数据也是先存在内存，但是在存储的时候会使用调用fsync来完成对本次写操作的日志记录，这个日志揭露文件其实是一个基于Redis网络交互协议的文本文件。AOF调用fsync也不是说全部都是无阻塞的，在某些系统上可能出现fsync阻塞进程的情况，对于这种情况可以通过配置修改，但默认情况不要修改。AOF最关键的配置就是关于调用fsync追加日志文件的平率，有两种预设频率，每次记录进来都添加，每秒添加一次。

这两种存储策略各有优点，从性能上讲，Snapshot方式的性能明显要高于AOF方式，因为前者是采用2进制方式存储数据，数据文件比较小，加载快速，存储的时候是按照配置中的save策略来存储，每次都是聚合很多数据批量存储，写入的效率很高，而AOF则一般都是工作在实时存储或者准实时模式下。相对来说存储的频率高，效率却偏低。但是从数据安全性上来讲，AOF数据安全性高于Snapshot存储，因为Snapshot存储是基于累计批量的思想，也就是说在允许的情况下，累计的数据越多那么写入效率也就越高，但数据的累计是靠时间的积累完成的，那么如果在长时间数据不写入RDB，但Redis又遇到了崩溃，那么没有写入的数据就无法恢复了，但是AOF方式偏偏相反，根据AOF配置的存储频率的策略可以做到最少的数据丢失和较高的数据恢复能力。

2.4.5 Redis数据库的pub与sub机制

pub/sub功能也就是publish/subscribe功能，也就是发布订阅功能。基于事件的[系统](http://www.2cto.com/os/" \t "http://www.2cto.com/database/201502/_blank)中，pub/sub是目前广泛使用的通信模型，它采用事件作为基本的通信机制，提供大规模系统所要求的松散耦合的交互模式：订阅者比如客户端以事件订阅的方式表达出它有兴趣接收的一个事件或一类事件，发布者比如服务器可以将订阅者兴趣的事件随时通知相关订阅者。

Redis数据库也支持pub/sub机制，本论文所设计的流式数据处理模型中，新引入了数据的输入节点也就是redis的subscribe节点，以及数据的输出节点publish节点，将这两个节点在采集数据的时候用。订阅者可以订阅多个Channal，而发布者可以通过Channel，向订阅者发送消息。但是发布者所发的消息是异步的，不需要等待订阅者订阅，也不关心订阅者是否订阅，简单地说就是订阅者只能收到发布者后续所发送到该Channel上的消息，如果之前发送的消息没有接收，那么也再也接收不到了，下面是Redis数据库的发布订阅命令。

IMG_256

PUBLISH：向channel\_1发布消息Hello。

SUBSCRIBE：订阅channel\_1消息，会收到发布者所发送的Hello消息。

2.5 本章总结

本章主要是对本论文的相关技术进行了介绍，首先详细介绍了当前流行的几种流式数据处理框架的编程模型和一些基本概念。然后对Node的事件驱动和非阻塞机制进行了详细阐述，主要是为后面Node-red的节点设计打下理论基础，再是对Node-red进行了详细介绍，最后还详细介绍了Redis数据库的基本概念、存储原理以及发布/订阅消息的模型。

# 基于Node-red与Redis的实时流数据处理模型的设计

3.1 需求分析

本论文主要是研究和设计一种基于Node-red与Redis的实时流数据处理模型，应用场景为实际项目中的网站群的实时访问监控。在本项目中旨在实时了解用户访问网站群的行为。捕捉用户请求并跟踪其所有响应，收集、处理并显示用户行为的细节数据，并可视化展示数据和挖掘数据背后的信息。针对该流式计算模型在该实际应用场景下提出如下的需求。

（1）高可扩展性；数据采集点可扩展:数据采集部分能够根据采集数据量的要求，通过添加服务器和配置的方式，无需编程实现水平扩展。数据存储可扩展：数据存储部分能够根据数据量的要求，通过添加服务器和配置的方式，无需编程实现水平扩展。分析/处理部分可扩展：数据分析部分能够根据应用的要求，通过添加服务器和配置的方式，无需编程实现水平扩展。

（2）高性能；支持流数据采集：能够实时采集业务日志、流水和各种软件运行状况的数据；业务数据采集速度不低于50笔/秒，每种软件运行状况数据不低于1笔/秒。细节数据存储：存储每笔业务活动记录（包括查询）。实时动态展示：业务统计数据实时变化延时不超过1分钟，软件系统统计数据变化延时不超过1分钟。

（3）高可用；系统可以通过集群等方式实现分布式部署，避免单点故障。

（4）可扩展；数据量、计算量会随着业务的不断扩展而不断增大，这就要求我们的模型需要有良好的扩展性。

（5）分布式；为了提高数据的处理能力和计算效率，模型还需要具备分布式的处理能力；

（6）安全性；数据安全是任何系统的一个首要前提，流式数据处理模型也必须要保证数据的安全性。

本论文要在这些需求的基础之上，要在Node-red上设计出高效的数据接入和输出节点，同时也要有高效的数据处理计算的节点，并充分利用Redis的pub/sub机制来实现数据的流式异步传输。并最终将这套模型应用到实际系统中去加以验证。

3.2 模型的总体架构

基于Node-red与Redis的实时流式数据处理模型的设计是搭建在Ubuntu环境下的，也可以部署在分布式环境上以提高流式数据的处理能力和计算效率。该模型通过重新设计数据输入、输出以及数据计算节点，以完成对实时流数据的处理。整个模型的架构如下图所示：



流数据处理模型架构图

从该模型的架构图中可以看出，redis数据库充当了数据交换的中心，而整个数据流的处理逻辑都交给计算节点群去完成。数据首先通过redis的channel（通道）进入redis server，然后Node-red利用redisSub节点去订阅相应通道（channel）的数据交给计算节点（function nodes）集群进行数据计算，而计算节点集群所产生的中间结果集，通过redis\_in节点传给redis server进行统计，最后产生的最终计算结果通过redisPub节点发布到前端可视化模块中。

在原始的Node-red中是没有任何节点可以与redis进行交互，为此，新增加了redisSub、redisPub、redis\_in和redis\_out节点。为了，用户可以自定义数据的处理逻辑，引入了函数节点，多个函数节点构成了整个流式计算的计算节点群。有了这些节点，就可以方便快捷地在Node-red上编写流式数据处理的业务代码，更为重要的是，这些业务代码可以实现一次编写多次使用，方便移植和维护。

3.3 节点处理模块的设计

节点是Node-red的重要组成元素，所有的flow都是通过一个一个的节点组成的，在Node-red中有三类基本的Node，数据输入节点、输出节点以及数据处理节点。为了设计出适合流式数据处理的模型，这里必须对这三类节点进行补充设计，在这一节中主要是对整个流式数据处理模型所需要的节点给出详细的设计方案。Node-red节点本身一般包括两份文件：js文件和html文件，js文件主要定义了组件具体做些什么事情，有什么样的功能；html文件主要定义了组件的属性，组件编辑框格式和帮助信息，下图为一个Node的设计方案：



Node-red节点设计方案图

为了保证所创建的Node能够提供一致的用户体验，这里给出Node-red的Node设计原则。

1. 要求创建的Node要对各种数据类型的输入数据进行必要的处理，即使某些类型并不是这个Node所需要的。这样做有两个目的，一是为了便于对原始数据进行追加额外说明信息，二是为了便于节点的扩展。
2. 由于Node-Red在识别和处理node的时候使用了大量的字符串匹配操作，所以在node的定义中有一些名字的字符串是必须保持一致的，否则Node-Red在解析的时候就会出错。
3. .html文件分为3部分：node的定义，node的编辑模板和node的帮助信息。Node的定义主要用于：确定node的类型，可编辑的属性，在浏览器中显示的样式，是一段可执行的js代码，RED.nodes.registerType；编辑模板主要是生成用户编辑该node的实例时的界面(由data-template-name包括的一段HTML代码)，用户的输入最终会保存在node的定义中；
4. 在.html文件中，data-template-name、node-input-xx、data-help-name都是Node-Red系统保留字。data-template-name、data-help-name的值必须和文件名字的name部分一致。RED.nodes.registerType的第一个参数也必须和文件名字的name部分一致。
5. 每个Node的可编辑的域在defaults中声明，data-template-name所包含的node-input-xx负责生成输入框。defaults的每个域的名字必须和node-input-xx中的名字保持一致。在.js文件中使用可编辑域的值的时候，直接访问defaults的域就可以，不必添加defaults前缀。
6. 在.js文件中，RED.nodes.registerType用来注册一个node实例的生成函数，它的第一个参数必须和文件名字的name部分一致。传给生成函数的参数是node可编辑域的值(已编辑完成)及node共享域的值。
7. input的callback是node输入的处理函数。需要注意的是，Node-Red节点间数据传输使用的是名字为payload的域，这个也是Node-Red系统保留的。

3.3.1 数据输入节点的设计

数据的输入节点（input node），主要是用于从外部设备或者其他外部接口获取数据到Node-red中进行数据分析。在Node-red的一个flow中，输入节点是所有message的入口，为下一个Node产生新的message。由于Node-red自带的输入节点很有限，而且不适合流式数据的输入，所以在这里必须补充设计数据的输入节点。

为了满足流式数据的输入需求，数据的输入节点的设计必须要满足一下几个原则：

1. 流式化数据，为了让成批到达的数据也能够在这样一个模型中得到计算，我们在设计数据输入节点的时候就要考虑到这点，也就是说让批量到达的数据逐条进入Node-red的flow。
2. 统一的数据格式，在一个数据处理模型中，数据格式的好与坏意味着后序进行数据计算的简与繁。
3. 高吞吐量，由于流式数据的产生是源源不断的，所以在设计输入节点的时候要充分考虑节点的数据吞吐量问题，不然会造成大量数据的堆积，从而影响后续的数据分析与计算。

（4）高稳定性，输入节点是数据的入口，稳定性是必须考虑的一个因素。

（5）可移植性，为了能够将自己设计的数据输入节点共享给其他用户，节点的可移植性也十分重要。

为了设计出高效的适合流式数据传输的输入节点，考虑到流式数据的特点，结合redis数据库的sub机制，我们可以为Node-red新增一个redisSub节点。从上一小节的总体架构图中我们可以看出，我们尽量让所有的数据通过redis的发布订阅机制来进行收集，把采集到的数据按类别放到不同的redis通道（channel）中，然后在Node-red中通过我们新增加的redisSub节点去订阅相应channel的数据，这样我们就可以把数据引入Node-red中，完成了数据的接入工作。同样redisSub节点也包括两个文件，一个是编写具体功能的实现代码的文件js文件，另一个是用于界面设计和帮助文档描述的html文件。由于Node-red原始节点的存在，所以在进行文件命名标号的时候从52号开始，因为文件名编号和节点的ID是紧密相关的，所以节点的标号必须唯一。设计好新的节点后需要重新安装部署新节点到Node-red中，在利用npm安装的时候，Node-red的节点注册模块会去检测setting.js配置文件，依次加载配置文件中的其他外部模块。下图是整个redisSub节点的设计图。



对于redisSub节点的界面ui设计，需要考虑有哪些信息需要用户输入该节点。因为每个节点都有自己的名字，所以首先需要的一个信息就是用户为该节点取一个名字，需要用户输入Name字段。由于数据是存放在redis server上的，所以还需要redisSub节点的描述redis server的ip地址和端口号。当redisSub节点连接上redis server后，不知道数据是位于redis的哪一个channel上，因此还必须给出通道名称，这些都是redisSub节点所需要的最基本的信息。另外还有就是redisSub节点的帮助信息也必须给出一定的说明。ui界面主要是定义在52\_redisSub.html文件中。

<script type="text/x-red" data-template-name="redisSub">

<div class="form-row">

<label for="node-input-name"><i class="icon-tag"></i> Name</label>

<input type="text" id="node-input-name">//节点的名字

</div>

<div class="form-row">

<label for="node-input-name"><i class="icon-tag"></i> host</label>

<input type="text" id="node-input-host">//redis server的ip地址

</div>

<div class="form-row">

<label for="node-input-name"><i class="icon-tag"></i> port</label>

<input type="text" id="node-input-port">//redis server 的端口号

</div>

...//其他说明信息

<div class="form-row">

<label for="node-input-name"><i class="icon-tag"></i>Channel</label>

<input type="text" id="node-input-channel">//redis 通道名

</div>

</script>

而对于redisSub节点的具体功能，是在52\_redisSub.js文件中实现的。首先，要调用Node-red提供的节点创建函数createNode（）创建一个节点，并把配置信息告诉节点。节点接收到这些信息后，创建一个数据库连接池函数redisConnectionPool，将redis server的ip和port，和createNode函数内部所产生的uuid传递给连接池函数。数据库连接池主要是通过一个connections数组的\_nodeCount来记录有多少redisSub节点连接redis server，当有一个新节点连接redis时，该值就会加一，同样当有一个节点断开了解的时候就会减一。当有close请求到的时候首先要判断\_nodeCount的值是否为0，来决定是否删除connections对象数组。关于redis数据库连接池函数的执行流程如下图所示：



Redis数据库连接池函数执行流程

有了数据库连接池函数，就可以来实现redisSub的功能了。redisSub节点的前端页面将用户输入的redis server的信息保存起来，然后通过参数传给连接池函数连接redis server。连接数据库后调用client.subscribe()方法去订阅指定的通道，如果订阅成功，就让client去监听一个message事件，看通道是否有数据发送过来，如果有数据就封装在msg.payload中，让node的send()方法发送出来供下一个节点接收。与此同时，client还要去监听redis的close事件，当redisSub节点断开与redis server的连接的时候，就要调用redisConnectionPool.close()方法去断开连接。

下面就是redisSub的功能函数的伪代码：

function redisSub(config) {

//创建节点

RED.nodes.createNode(this,config);

//将html文件中定义的节点属性保存下来。

this.host = config.host;

this.port = config.port;

...

this.channel = config.channel;

//为节点生成一个唯一的id

var uuid1 = uuid.v4();

this.client = redisConnectionPool.get(this.host,this.port,uuid1);

...

this.client.subscribe( this.channel);

//监听相应通道的message时间，将通道发送过来的数据封装到msg.payload中，并通过node.send()方法发送出来。

this.client.on("message", function (channel, message) {

var msg = {};

msg.payload = message;

node.send(msg);

});

this.on("close", function() {

redisConnectionPool.close(node.client);

});

}

3.3.2 数据输出节点的设计

数据进入Node-red后，经过各个计算节点的数据计算、封装等工作，然后打包成系统规定的数据格式后，需要从Node-red中输出，进入后续的数据可视化展示。数据的输出就用到了Node-red的输出节点。Node-red的输出节点允许把数据输出到Node-red的flow以外的其他服务和应用上去，对内有一个数据输入的左断点，对外暴露一个公共接口。

在Node-red中有一个常用的输出节点就是debug节点，这个节点是在编写flow的时候调试的时候用的，主要显示打印出数据经过上一节点处理之后的具体信息。Debug节点是一个具有开关的节点，允许程序员手动开启或者禁用该节点。debug节点的使用也非常简单，只需要在Node-red左侧的节点栏中找到该节点然后拖拽到相应节点的后面，并用线连接起来就可以实现数据的传输，最后开启debug的启动按钮，当部署了所编写的flow后，就可以在Node-red的最右侧的debug面板中看到打印出来的具体数据。值得注意的是，debug节点的只有一个数据的入口，而没有数据的输出端，在设计debug的时候，重新封装了sendDebug()函数，用来发送消息，将消息直接发送到Node-red的网页编辑器debug视图上直接显示，而不是交由下游节点做数据处理。下面给出debug节点的设计逻辑的部分伪代码。

function DebugNode(n) {

...//创建节点并定义complete属性，用来判断数据封装是否完成。

RED.nodes.createNode(this,n);

this.name = n.name;

this.complete = n.complete||"payload";

...

this.on("input",function(msg) {

if (this.complete === "true") {

//debug节点完成了msg的接收进行封装

...

msg={id:this.id,name:this.name,topic:msg.topic,msg:msg,\_path:msg.\_path}

sendDebug(msg);

} else {

//debug的用户需要自己定义msg的属性

...

var output = msg[property];

if (this.complete !== "false" && typeof this.complete !== "undefined") {

output = propertyParts.reduce(function (obj, i)

...

if (this.active)

sendDebug({output});

}

...

});

});

有了debug节点，可以方便用户在编写自己的flow的时候，及时查看数据的处理情况。本文在第四章中应用该模型来解决实际问题的时候，将大量应用到debug节点。

为了保证数据的实时地输出到Node-red的flow以外的其他服务和应用上，这里我们新引入了redisPub节点。顾名思义，redisPub节点就是将redis的publis功能嵌入到Node-red中，通过设计一个新的节点来将经过Node-red处理和计算过的数据输出来，这里之所以选择redis的publis发布数据，一方面保证了数据的异步传输，另一方面也保证了数据的隔离（原因是各个redis的通道数据是相互隔离的，互补干预）。在坚持节点的设计原则的前提下，下图给出了redisPub的设计方案。



redisPub节点设计图

结合上一小节数据输入节点的设计可知，redisPub节点和redisSub节点的设计恰好相反，redisPub节点只具有一个数据的输入接口，也就是只有数据的输入端点，这一端是连接上一个数据处理节点的，在redisPub节点中也必须定位redis的位置，也就是redis服务器的ip，端口号，不管是在redis集群还是在单点的redis服务器中都必须要指定，同时还要指定数据输出到哪个redis的channel中。所以redisPub节点的ui设计与redisSub节点的ui设计十分相识，不同的是他们的功能代码不一样，体现在js文件中。下面给出redisPub节点的设计逻辑的具体流程图。



RedisPub设计逻辑流程图

同样，在实现redisPub节点的时候，也用到了数据库连接池函数，关于这个函数的设计思想在上一小节redisSub的设计中已经做了详细阐述。从上面流程图可以看出，当redisPub节点成功连接redis后，将去监听input事件，当事件有数据输入该节点后，就会调用this.client.publish()发布函数，将封装好的数据（message对象）发布到指定的channel上。

3.3.3 数据计算节点的设计

数据计算节点在Node-red中起着举足轻重的作用，几乎所有的flow中都会用到数据计算节点。数据计算节点允许用户编写JavaScript函数来处理进入Node-red中的数据，编写自己的业务代码，将定义好的数据类型转化为在Node-red中流动的message对象。在Node-red中的message实际上就是一个JavaScript对象，message对象至少要包含payload属性，用来保存具体的数据。就像下面这样一个最基本的Node-red的massage数据格式：

msg={

payload:”massage payload”

}

计算节点接收到message后，主要处理的也是payload字段中保存的信息，处理后的数据也会封装成一个message对象传到下一个节点。然而，message对象不仅只具有payload字段，还可以扩展出更多的其他字段来补充说明message对象的属性。比如下面这个message对象：

msg={

payload:”massage payload”

topic:”error”

location:”somewhere in space and time”

}

计算节点通常包含一个数据输入端点和一个或多个数据输出端点，在Node-red中提供了部分具有特殊功能的数据处理节点，比如change\_node，可以用来增加或者删除message的字段，再如switch\_node，可以用来做开关节点使用，它是通过判断message对象的某一字段是否存在或者真假来决定最后输出什么样的message对象。为了能够进一步扩展Node-red的功能，方便利用JavaScript函数加载外部的js模块，这里引入function\_node，也就是函数节点。可以说function\_node在Node-red中就像一把瑞士军刀，可以使用户不必依赖于现有的数量有限的几个节点来处理数据。顾名思义，函数节点其实就是暴露出来的一个JavaScript函数，用户可用通过编写一个JavaScript函数来处理从上游节点流下来的message，并返回处理后的一个或多个massage。函数节点是用来做数据处理和数据格式化的利器，引入函数节点使得Node-red的对流式数据进行处理变得简单容易。下图是function\_node的设计图：



用户可以通过function\_node内置的编辑器sandBox，编写用户自己的JavaScript函数来处理message。在function\_node内编写的JavaScript函数内部是调用本机上的JavaScript运行环境来解释执行的，同时在函数节点中可以去调用外部的js模块，但是这首先会去配置文件setting.js文件中找到要包含的模块。所以function\_node在执行每一个函数的时候首先会去检查这个配置文件，在这个文件中去查找全局的函数模块。在setting.js中，通过functionGlobalContext支持全局模块：

functionGlobalContext: {

// bonescript:require('bonescript'),

// arduino:require('duino')

lodash:require('lodash')

}

对于自己编写的JavaScript函数要求每一个函数都有一个返回值，也就是一个message对象，即使没有显式地返回，每个函数都会默认返回一个payload字段为空字符串的message对象。

3.3.4 数据库访问节点的设计

数据库访问节点

3.4 节点的重新部署

节点设计和实现完之后，一步重要的工作就是要将新设计的节点部署到Node-red中。节点可以作为模块打包或者发布到npm库中，这使得它们易于安装其所有依赖的模块。为了解决安装包的依赖关系，在打包节点的时候就要严格按照npm包管理规则来打包。下图是一个redisSub节点打包的目录结构：



节点package目录结构

本文采取的是本地模块安装的方式，在本地安装节点模块，就用到了npm link命令。将节点在本地目录,链接到一个本地Node-red安装目录,这和npm安装是一样的。本地部署节点按照如下两个步骤即可完成部署。

1. 在包含有package.json的目录下执行sudo npm link命令；
2. 在Node-red的运行运行目录下执行npm link <节点模块的名字>。

部署成功后，重新启动Node-red，然后浏览器中打开编辑界面，在最右侧的节点视图就可以看到新增加的节点，这样就完成了节点的设计和部署工作。为接下来改模型的应用提供了技术支持。

3.5 本章总结

# 基于redis有序集合的去重统计原理的研究

## 4.1 Skip List的基本原理

Skip List是由William Pugh提出的一种基于并联链表的、随机化的数据结构。其效率可以与二叉查找树相提并论，可以实现平均复杂度为O（longN）的插入、删除和查找操作。一般而言，跳表是对有序的链表增加上附加的前进链接，增加是以随机化的方式进行的，这里的随机化都是以对数随机化的方式来实现的，所以在列表中的查找可以快速的跳过部分列表(因此得名)。众所周知，对于有序链表的查找操作，其时间复杂度为O(n)，尽管真正插入与删除节点的操作的复杂度只有O(1)，但是，这些操作都需要首先查找到节点的位置，换句话说，是查找拉低了有序链表的整体性能。而Skip List采用“空间换时间”的设计思想，除了原始链表外还保存一些“跳跃”的链表，达到加速查找的效果。可以很好解决有序链表查找特定值的困难。

## 4.2 redis有序集合zset的源码分析

## 4.3 zset的去重统计原理

## 4.4 本章小结

# 基于Node-red与Redis的实时网站访问监控系统

4.1 实时网站访问监控系统介绍

该系统的所有数据是来源于某地方政府的电子政务网站群的访问流量，数据真实可靠、说服力强、具有重要的实际意义和研究价值。同时数据具有通用性，因为这是截取的服务器端的访问流量，也就是通用的HTTP报文，适合各类网站群的实时监控与数据分析。

随着各级地方政府的电子政务系统的不断发展，但是信息收集与数据分析能力还比较薄弱，急需要一个统一的实时数据收集、储存、分析、应用的平台。因此，本文首先提出一个基于Node-red与redis的实时流数据处理模型，随后应用这个模型来解决网站群的流量数据的实时收集和分析问题。在本章中重点是应用该模型来实现一个可视化的实时网站访问监控系统，同时也将详细阐述系统中的用户行为监控和网站群页面监控的分析方法。

4.1.1 实时网站访问监控系统的功能介绍

本文所设计的系统是通过实时采集网站群访问流量，通过本文所设计的流式数据处理模型来解析处理实时数据，并从中挖掘出用户关心的有价值的信息，用于实时监控分析网站群的访问情况。该系统主要包括以下几个功能：

1.用户行为监控

实时了解用户访问网站群的行为。捕捉用户请求并跟踪其通过所以响应，收集、处理并显示用户行为的细节数据。

具体实现以下功能：

（1）错误页面跟踪，对返回码为404,500等出错页面进行统计跟踪；

（2）用户终端类型，对用户访问网站群的终端进行统计；

（3）受访页面统计，用户访问网站所浏览的页面统计；

（4）来路页面，用户通过页面浏览网站群统计；

（5）地区分布，根据用户IP统计访问网站群的地区分布，并区分内外网用户（内网IP地址范围及相关部门的对照表需信息中心提供）；

（6）IP/PV,一天之内独立IP数，相同IP数被计数一次；

（7）页面浏览量，用户每次刷新被计算一次；

（8）重复访问率，同一IP，不在同一天内访问同一页面的访问量/总访问量；

2.网站群页面监控

（1）关键词搜索频率，用户搜索关键词的频率；

（2）热门关键词统计；

（3）二级域名访问统计（需信息中心提供二级域名对照表）；

（4）频道访问统计（需信息中心提供频道名称对照表）；

（5）热点页面统计；

4.2 实时网站访问监控平台的设计

4.2.1 实时数据采集模块设计

整个系统作为一个实时数据的交互处理中心，除了自己内部的数据通信以外，还需要对网站群的访问流量进行采集。这种数据具有实时性、连续性、非机构化等特点，同时数据量也非常巨大。由于其实时性明显，同时也要求系统能够实时展示分析出网站群的访问情况，所以不能采用传统的先收集后处理的方案，需要重新设计一套实时流数据收集方案。

考虑到访问流量数据是一种非结构化的数据，为了能够更加准确地收集有效的信息，需要在采集数据的时候进行原始数据的预处理。因为原始的访问流量就是HTTP请求和响应报文，如果仅仅是收集到了这些报文，它都是以一种字符串的形式存在的，为了方便解析，更好更准确的处理这些数据，有必要进行初步地结构化处理。由于json格式的数据能够有效地反映数据的特点，同时与JavaScript对象能够实现无损转换，所以在进行数据格式化的时候选择json格式，同时在后面处理和存储中间数据的时候也选择json格式。

因此，我们设计出如下的实时数据采集方案：



实时数据采集方案图

在客户的服务器端，将网站的访问流量做一份拷贝，利用http\_tracer将这部分流量截取到redis集群中，专门用一个redis节点来采集数据。由于原始的数据报结构混乱，难以分析，所以在进行下一步数据分析之前进行了预处理。原始数据通过redis\_sub节点从redis server上被订阅，交给msgToJSON模块（这个模块是Node-red中的function\_node实现的）。msgToJSON模块把原始数据包分为请求包和响应包两类，最后只是在message对象中增加一个type字段加一区分。最终产生的数据就是一个JSON对象，继续传递给下游的数据处理中心，进行数据处理。下表就是预处理前的原始数据报格式：

原始数据格式

HTTP\_TRACE\_REP|1419840350182.825|172.16.1.1:18083|42.91.9.230:24735|#8|HTTP1.1|GET|/emall/css/jquery.alert.css|304|NotModified|15.491943359375ms|33.83KBAccept:\*/\*Referer:http://emall.lzbank.com/emall/myorder/queryMyOrder.doAccept-Language:zh-cnUser-Agent:Mozilla/4.0(compatible;MSIE8.0;WindowsNT5.1;Trident/4.0)Accept-Encoding:gzip,deflateHost:emall.lzbank.comConnection:Keep-AliveCookie:JSESSIONID:9986FBA4B93382AB77946439738F1714|emall\_shop\_car:""|jiathis\_rdc: %7B%22http%3A//emall.lzbank.com/emall/goods/goodsinfo.do%3Fgoodsid%3D3436814040014%22%3A-1798085429%2C%22http%3A//emall.lzbank.com/emall/goods/goodsinfo.do/....

采集到的数据都publish到redis的一个channel中。得到的数据是原始的HTTP请求和响应报文，根据客户需求说明，以及后面的数据处理的需要，从在从redis的channel中去订阅（subscribe）这些数据做初步的格式化，经过预处理的HTTP报文变成形如下面这样的json对象。

结构化后的JSON格式

{

"type": "HTTP\_TRACE\_REP", //标识报文类型

"http\_version": "HTTP1.1",

"http\_method": "GET",

"userIP": "42.91.9.230", //用户的ip地址

"hostIP": "172.16.1.1", //服务器ip地址

"target": "/emall/css/jquery.alert.css", //请求的目标文件

"status": "304", //http状态码

"host": "emall.lzbank.com\n", //网站域名

"user\_agent":"Mozilla/4.0(compatible;MSIE8.0;WindowsNT5.1;Trident/4.0)\n",//用户浏览器类型

"referer":"http://emall.lzbank.com/emall/myorder/queryMyOrder.do"

}

4.2.2 用户行为分析模块设计

4.2.2.1 用户行为分析的总体架构设计

用户行为分析模块是搭建在第三章所设计的基于Node-red与redis的实时流数据处理模型上的，数据在不同模块之间的流动是利用redis的publish和subscribe机制来完成。位于网关的抓包模块抓取原始的报文信息，然后把数据发布到数据分析系统的redis server上。然后在Node-red中利用在第三章设计的数据输入节点redis\_sub节点，从redis server的一个通道（http\_tracer）订阅原始数据。数据进入Node-red之后，经过计算节点进行数据处理和封装，最后通过redis\_pub将处理结果publish到redis的指定通道中，供可视化模块去接收这些数据。在数据处理过程中，需要用到redis做中间结果集的保存和初步的计算工作。在节点之间的数据是通过node.js的emit和on事件机制进行，这种机制已经被集成到Node-red中，因为Node-red也是基于node开发的。

在进行用户行为分析的时候，本文把计算节点按照功能的不同划分为7个计算节点，形成一个计算节点群。整个用户行为分析模块除了这7个用于数据处理和计算的节点外，还包括redis数据库的访问节点用于传输中间结果集到redis数据库中，还包括一个功能节点（定时节点）用于定时清理redis的中间结果集，以减轻redis server的负担。在用户行为分析模块中，主要讲述了4个数据分析节点，他们分别是refererCount、countUserAgent、repeatVisit以及userIP。这些节点主要完成来路页面统计，用户的浏览器类型统计，重复访问页面统计，以及独立访问的IP地址，其中，在repeatVisit和userIP中整合了pv，uv的计算。



用户行为分析模块总体架构图

从用户行为分析模块的总体架构图可以看出，redis是整个模块数据交换的纽带，也是进行数据计算的中心。原始数据通过http\_tracer从网关截取到后，发布到redis server的http\_trace通道，各个计算节点在从http\_trace通道去订阅（subscribe）这些数据，进行数据处理和分析，然后将中间结果集重新发布（publish）到指定的通道，各个通道相互对立、互补干扰。中间结果集通过redis的计算功能进行统计计算，最终形成前端可视化模块需要的数据格式，而前端可视化模块通过node.js的socket.io来取得这些数据，进行实时展示。

由于客户的要求，需要对频道的更新情况进行持久化，并且能够随时查询到频道的更新情况。为此，引入了mongo数据库提供数据持久化功能，同时前端可视化模块中增加了统计查询页面，通过socket.io来查询mongo中的频道更新数据。

由于巨大的数据量和繁重的数据计算，导致redis server的负担也异常。为此，需要定时清理redis server上的中间结果集，系统选择在每天的凌晨清理数据，这样一方面可以减轻redis server的负担，提高运行效率，另一方面也不会影响数据分析的结果和前端可视化模块的展示。

4.2.2.2 redis数据库结果集设计

用户行为分析模块得到的最终分析结果都存放在redis中，在redis中进行统计和初级计算。这些数据在redis中是一系列的字符串、集合、有序集合以及hash表，用它们表示这用户行为数据。

redis中的哈希表（hash）是一钟键值（key-value）结构，一个键对应一个值，根据键计算存储地址，访问速度很快。在用户数据记录时，用户的ip、sessionid和网页错误类型等数据存放为hash格式。redis中的有序集合（zset）是以跳跃表为底层结构的键值结构。跳跃表可以高效的对某随机键的值进行加法运算。因此是数据分析中用户pv、uv等累加型数据的理想存储结构。在本文构建的用户行为分析系统中使用较多。

这些表中的数据也是下一阶段前端可视化模块中的图表（线图、饼图）的原始数据。数据分析模块向结果集中单向写入数据，而数据可视化模块从结果集中单向读取数据，两个模块之间并无直接交互，redis是连接它们的纽带。下表示对redis中的数据结构说明。

|  |  |  |
| --- | --- | --- |
| Redis表名 | 表类型 | 功能描述 |
| errPage.zset | 排序集合 | 错误页面 |
| errType.zset | 排序集合 | 错误类型 |
| visitPage.zset | 排序集合 | 热点访问页面 |
| refererPage.zset | 排序集合 | 来路页面 |
| userIP.zset | 排序集合 | 用户的ip统计 |
| repeatVisit.zset | 排序集合 | 重复访问的页面统计 |
| errWebHostName.string | 字符串 | 错误页面所属的网站 |
| HostName.set | 集合 | 网站的域名 |
| IPBelong.hash | 哈希表 | 访问ip所属区域 |

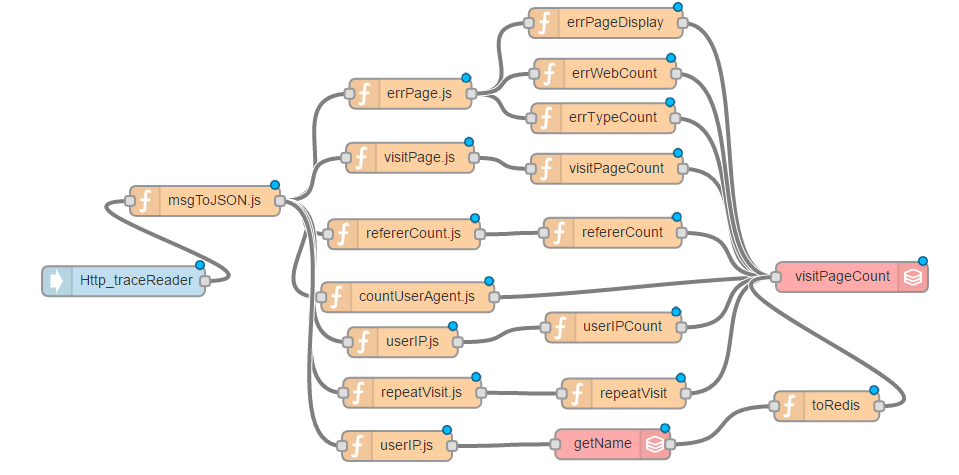
中间结果集的数据表以zset和hash表为主。所有的表都是单独承担某一功能的数据存放任务，这样可以减少不同功能模块之间数据的相互干扰，同时易于维护。

4.2.2.3 用户行为分析的算法流程

用户行为分析

4.2.2.4 在Node-red中的数据流程

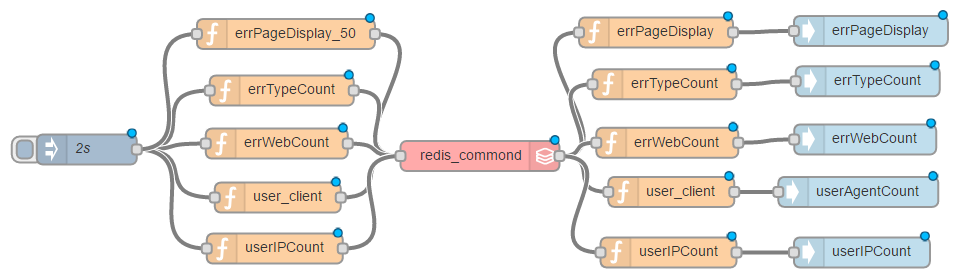
数据通过redis\_sub节点从redis通道获得元素数据，再经过初始化模块对原始数据进行初步的结构化处理，然后就进入后续的数据分析处理阶段，最后将数据处理结果通过redis\_pub节点发送的redis server的指定通道上供可视化模块接收并展示。在Node-red中，整个用户行为分析模块的flow如下图所示：



用户行为分析模块的flow数据流程图

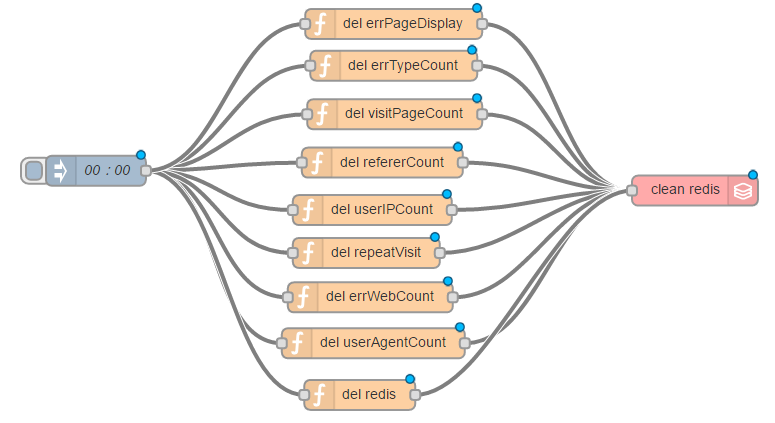
在整个用户行为分析模块的flow中，各计算节点从预处理节点msgToJSON中得到数据后，将数据分发到各个功能单一的计算节点上。这里有7个功能节点，他们主要完成错误页面统计、受访页面统计、访问来由页面统计等。从原始数据中解析出相应的数据后，再经过下游的数据封装节点（同样是function节点来完成）将数据封装成redis数据库访问节点能够识别的数据结构，将这些数据暂存到redis server上的中间结果集中进行统计，图中的visitPageCount就是redis中间结果集节点。

经过redis server的计算后，结果会保存到redis所提供的数据结构中，有set、zset、字符串以及hash表，其中关于统计的数据都保存到zset中，因为redis提供的zset是一个自动排序的集合，每次进行count后，都会重新对数据进行排序，这样就能够使得结果数据具有鲜明的对比性，让后面的数据可视化变得一目了然。为了能够实时地更新数据，在Node-red中设置了一个定时器，规定每两秒钟去redis server上去取一次数据，将取得的数据封装成可视化模块需要是数据结构，实际上就是highcharts所需要的数据结构。下图为定时取数据和封装数据的flow数据流程图。



实时推送数据的flow流程图

由于数据是不间断的到来，数据量十分巨大，所以每天在redis server中都会堆积大量的中间结果，再加之redis还要完成基本的统计计算工作。因此，redis server一般都是超负荷运行的。为了减轻redis server的运行负担，提高redis的计算效率，必须对redis server上的中间结果集进行定时清理，同时也要保证被清理的数据不会影响分析结果。为此，系统在Node-red中同样设计了一个定时器，规定每天的凌晨清理redis。因为，凌晨清理数据不会影响第二天的数据分析，同时也不会影响前端可视化模块的展示。下图为清理redis server的中间结果集的flow。



清理redis server中间结果集的flow

4.2.3 网站群页面监控模块设计

4.2.3.1 网站群页面监控的总体架构设计

同样，网站群页面监控模块也是搭建在第三章所设计的基于Node-red与redis的实时流数据处理模型上的。网站群页面监控模块主要是完成用户关键词搜索频率统计、热门关键词统计、热点页面统计、二级域名访问统计（需信息中心提供二级域名对照表）以及频道访问统计（需信息中心提供频道名称对照表）。在本节中主要涉及4个数据分析节点，分别是keyWordCount、hostCount、hotVisitPage以及channelVisit，他们分别完成关键词统计、二级域名访问统计、热点页面统计以及频道访问统计。整体架构和用户行为分析模块的架构类似，区别在于无需对数据进行持久化，同时在可视化模块中也无需统计查询模块。下图为该模块的总体架构图：



网站群页面监控模块架构图

从该模块的总体架构图可以看出，redis server仍然是数据交换和数据计算的中心，各个计算节点都是通过redis的发布订阅机制与redis server进行数据交换。可视化模块不再是直接去向redis server取得最终数据，而是在他们之间引入了push server，由push server去取数据并通过socket.io与前端可视化模块进行数据交换。从软件设计角度上来讲，这样可以减少模块与模块之间的依赖，另一方面，从运行效率上来讲，可以减轻前端可视化模块的负担，提高运行效率。

4.2.3.2 redis数据库结果集设计

由于网站群页面监控模块需要监控热门关键词的搜索情况，需要用到redis的zset数据结构。为了找出热点页面所属的网站，需要提前准备好各个网站的host\_name对照表，需要用到hash。因此，该模块主要用到的redis数据结构是排序集合以及哈希表。下表给出了该模块所用到的表名称和功能对照情况。

|  |  |  |
| --- | --- | --- |
| Redis表名 | 表类型 | 功能描述 |
| KeyWorldCount.zset | 排序集合 | 关键词统计 |
| HotVisitPageCount.zset | 排序集合 | 热点页面统计 |
| HostVisitCount.zset | 排序集合 | 访问网站统计 |
| HostName.hash | 哈希表 | 网站名称与域名对照 |
| ChannelName.hash | 哈希表 | 频道连接与频道名对照 |
| ChannelVisitCount.zset | 排序集合 | 频道访问统计 |

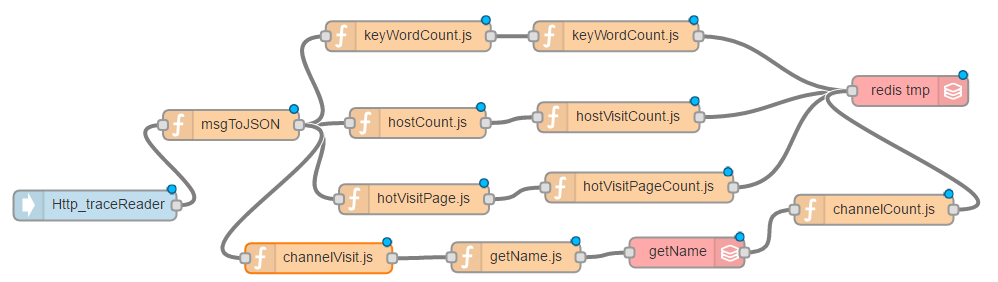
在该模块中主要需要实时记录一些统计信息，比如每个被搜索的关键字的搜索次数，页面的访问次数，并实时给出排名情况，以此来找出热门关键词以及热点页面。这都需要用到redis的排序集合，当有新的同样的数据到来的时候，就向redis server发送一次zadd指令，redis就会在相应字段的value上进行一次加加操作，由于zset会底层数据结构是一种特殊的跳表结构，因此会进行快速的自动排序。从而可以根据value的大小判断出关键词的搜索频率和页面的访问频率。为了能够快速找到每个页面所属网站的名字，提供了一张关于网站域名与网站名字的对照表，存放在redis的hash数据结构中，可以提高查找效率。

4.2.3.3 网站群页面监控的算法流程

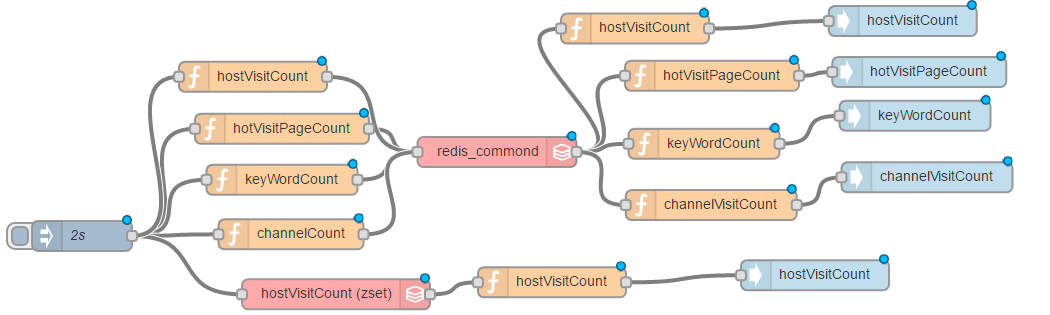
4.2.3.4 在Node-red中的数据流程

网站群页面监控的原始数据仍然是来自于Http\_tracer从网关截取的http报文。经过初始化模块msgToJSON后的数据进入计算节点集群，进行数据处理。keyWorldCount.js节点从msgToJSON节点接收到message对象后，从msg.target中解析出以“&q”开头的字符串，这个字符串就是搜索的关键字，让后封装成新的message对象传给下一个节点keyWorldCountToRedis.js节点，该节点将数据封装为redis的zset存储的数据结构存储到redis中间结果集keyWorldCount.zset中。同样，hotVisitPage.js和hostCount.js节点以类似的过程将计算出来的结果存储到redis中间结果集中。

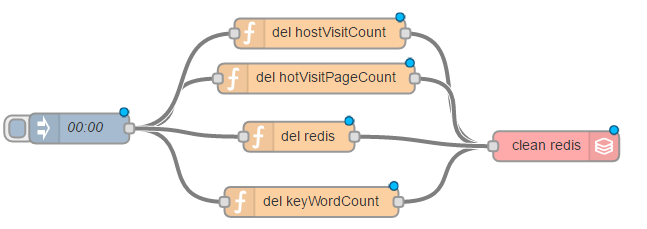
在频道访问统计这一模块中，channelVisit.js从msg.target中解析出频道页面后，又需要知道所访问的频道的名字，这是，将channelVisit.js节点的msg对象交给getName.js节点，又它去redis的hash表中查找频道名字。当找到名字后，重新再封装存储到channelVisitCount.zset中存储。下图为网站群页面监控模块在Node-red中的真实流程图。



经过各个计算节点的处理后，数据都存放在redis中的中间结果集中。为了能够在前端可视化系统中实时动态展现网站群页面监控的情况，需要定实从redis server中去的数据，这里通过设置一个定时节点inject\_node，每两秒钟取一次数据，并通过数据封装的函数节点封装为前端可视化模块需要的数据格式，交个redis\_pub节点发布到相应通道上。下图展示了在Node-red中定时取数据的真实数据流图。



由于数据是不间断的到来，数据量十分巨大，所以每天在redis server中都会堆积大量的中间结果，再加之redis还要完成基本的统计计算工作。因此，redis server一般都是超负荷运行的。为了减轻redis server的运行负担，提高redis的计算效率，必须对redis server上的中间结果集进行定时清理，同时也要保证被清理的数据不会影响分析结果。为此，系统在Node-red中同样设计了一个定时器，规定每天的凌晨清理redis。因为，凌晨清理数据不会影响第二天的数据分析，同时也不会影响前端可视化模块的展示。下图为清理redis server的中间结果集的flow。



4.2.4 数据可视化模块设计

4.2.4.1 数据可视化模块的功能需求

经过用户行为分析之后，得到的结果是代表用户的访问量、新用户的数量、访问的来源、访问的深度等信息。而网站群页面监控分析，最后的结果是代表关键词搜索频率、热点页面统计、错误页面统计等信息。所有最终结果的数据都是一系列统计表和集合的数据结构，这些数据不能直接交由用户，因为用户不了解分析的过程，就不知道这些数据代表什么含义，可视化模块的功能就是要把这些结果直观易懂的显示出来。同时，可视化模块要实时更新，当新的用户行为数据和网站群监控数据被分析出来后，要能立刻在可视化模块中看到。

需要支持的图表形式有：

1. 曲线图，用于显示流量变化。

2. 饼图，用于显示用户设备和浏览器使用比例。

3. 柱状图，用于显示用户访问深度和用户停留时间以及热点页面访问情况。

4. 表，用于显示关键词统计、错误页面跟踪、访问来路页面登信息。

4.2.4.2 可视化模块的架构设计

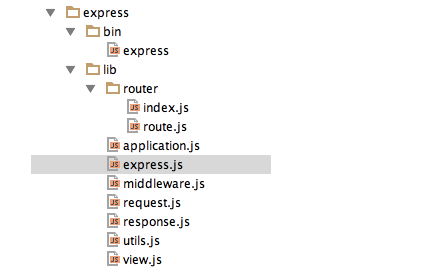
监控系统的可视化模块使用了MVC架构的web网站来实现的。使用MVC架构的web网站，就是通过模型(model)－视图(view)－控制器(controller)这样一种典型的软件设计范式而设计出来的一种网站结构模式。model（模型）是应用程序的核心部分（比如数据库记录列表）。view（视图）是显示数据的模块（图表、线图、饼图）。controller（控制器）处理输入（响应restful 请求的接口，接收消息）。



监控系统的MVC架构图

上图是监控系统网站的架构模型，功能上也分为model、view和controller三个功能模块。使用MVC架构的网站前端，具有数据格式统一并且显示风格多变的特点，因此，十分适合多样化数据呈现的网站群用户行为数据的展示。前端页面设计为一个web应用，使得显示的界面可以在任何一台机器上都可以查看，不需要而外安装单独的应用软件，这样就减少了安装软件和更新软件带来的麻烦，同时提高了系统的灵活性。

整个前端的显示系统是实用Node.js的express框架来实现的，express是一个简洁、灵活的 web应用框架，它的强大特性有助于快速创建各种web应用，以及丰富的http工具。下图为express框架的代码结构：



（1）bin/express是在命令行下，用来生成express 框架的目录结构。

（2）lib/express是整个框架的入口

文件。

（3）lib/router是路由模块，主要是进行路由分发，比对，执行callback回调函数。

（4）lib/middleware是中间件模块文件，主要是用于对response，request进行二次改写。

（5）lib/request是请求。

（6）lib/response是响应。

（7）lib/utils是工具函数集，它是对connect模块的一个补充，比如地址处理正则表达式等。

（8）lib/view是处理视图。

监控系统前端网站所使用的express框架的各个文件的调用关系如下图所示：



Express框架的调度关系图

从express框架各个文件的调度关系图中可以看到，它首先调用中间件，中间件的作用主要是改写改写request，response 请求的。将这2个请求导出，方便后面的模板渲染，后再调用路由模块。路由模块只要是根据path调用路由分发函数和分发路由，执行callback回调函数。最后调用view 模块，渲染事先编辑好的模板。

4.2.4.3 数据显示方法的设计

为了使数据显示更加生动、直观，能够一目了然地洞见数据背后的信息。在系统的可视化模块中提供了曲线图、饼图、柱状图、表格等显示方法，同时具有交互式的显示效果。为了支持这些图表的显示，系统采用了第三方插件highcharts。highcharts是一个非常流行，界面美观的纯Javascript图表库，可以为网站或Web应用程序提供直观，互动式的图表。Highcharts和其他许多JavaScript库一样沿用jQuery,MooTool以及Prototype等Javascript框架来处理基本的Javascript任务。因此，在使用Highcharts之前，需要在页面头部引用这些脚本文件。如表所示，js文件可以引入在线的，也可以引入本地的，这里以在线的为例。

<script src="http://cdn.hcharts.cn/jquery/jquery-1.8.3.min.js"></script>

<script src="http://cdn.hcharts.cn/highcharts/highcharts.js"></script>

在配置好highcharts之后，我们就看调用它提供的highcharts接口来填写图标的数据结构。

$(function () {

$('#container').highcharts ({ //图表展示容器，与 div的 id保持一致

chart: {

type: 'bar' //指定图表的类型，默认是折线图（line）

},

title: {

text: '我的第一个图表' //指定图表标题

},

xAxis: {

categories: ['苹果', '香蕉', '橙子'] //指定x轴分组

},

yAxis: {

title: {

text: 'something' //指定y轴的标题

}

},

series: [{ //指定数据列

name: '小明', //数据列名

data: [1, 0, 4] //数据

}, {

name: '小红',

data: [5, 7, 3]

}]

});

});

图表的数据主要是通过series字段来表示，其余的字段是用来描述图表的样式以及图表说明。所有的series字段的数据是在server端封装完成后，通过socket.io来向前端监控页面push。前端页面运行客户端代码，监听一个socket.io的端口，这样就可以在socket.io的客户端接收到数据，然后封装到highcharts模块中，就可以绘制出相应的图表，并及时更新图表上的数据。socket.io实时地从server端接收数据，使前端监控页面的图表实时动态变化。

4.3 实时网站访问监控平台的实现

4.3.1 syslog采集器mtail的实现

4.3.2 数据统计与封装模块的实现

4.3.3 日志监控页面的实现

4.4 本章小结

# 系统测试与性能分析

5.1 性能测试指标

5.2 测试环境

5.3 测试结果与分析

5.4 本章小结

# 全文总结与展望

6.1 研究工作总结

6.2 研究工作展望

致 谢

三年研究生的生活，带给自己的不仅是学习能力的提升，同样也带来个人素养以及认知水平的提高，自己的努力的同时，也得到了良师益友的帮助，在论文即将结束的地方，对他们表示衷心的感谢。

首先，首先要感谢的是本人的研究生导师，\*\*\*教授。\*老师治学严谨，待人和蔼，同时学术知识渊博。在三年的学习过程以及项目方面，给了我很大的指导，并且在人生以及就业方面，提出了自己的经验和看法，让我学习到了很多。论文的研究过程中，\*老师也对其进行指导，否则论文不会这么顺利的完成。再一次特别感谢恩师\*\*教授。

其次，要感谢实验室以及项目组的\*老师、\*老师、\*老师、\*老师等各位老师。在学习和论文的完成过程中，给我很多研究建议，解决其中的难点。

另外，感谢我的师兄，同学以及项目相关人员。在课题的研究过程中，帮助我克服了很多困难。一起的学习生涯，让我学到了很多，在此向他们表示衷心的感谢和祝福。

同样，论文期间，家人朋友的支持也是我的精神支柱，帮我客服生活、学习方面的困难，再次表达我真诚的感谢。

最后，十分感谢评阅论文以及答辩过程的老师和专家，谢谢你们的辛苦付出。