**基于HTTP2的微服务快速部署实践**

1绪论

说道微服务，要先从上世纪90年代开始逐渐广泛关注的新型软件开发方法，敏捷开发开始，其更强调团队与业务专家之间的紧密写作，频繁交付新的软件版本，注重软件开发过程。作为运维团队，为了更好的适应开发团队的敏捷开发，从运维的架构开始逐渐提出面向服务架构(SOA)的运维方式。 从2014年起，一种叫做微服务的架构方式逐渐兴起。 微服务是更为具体，微服务其本身并没有一个严格的定义，微服务中的应用由自动化的、可独立部署、扩展和管理的服务组成，提供了所必需的可扩展、弹性、跨平台等基础，微服务只需要在特定的某种服务中增加所需功能，而不影响整体进程。

REST(Representational State Transfer)这样的网络服务协议已成为定义微服务中API的首选，REST的资源是由URI来指定的，通过操作资源的表现形式来操作资源。

从2015年末开始， HTTP2也被各种浏览器支持，在这次实践中我们讲进行如何部署微服务的同时又在前端支持http2，并适用于现代软件开发模式，敏捷开发中快速部署RESTful微服务。

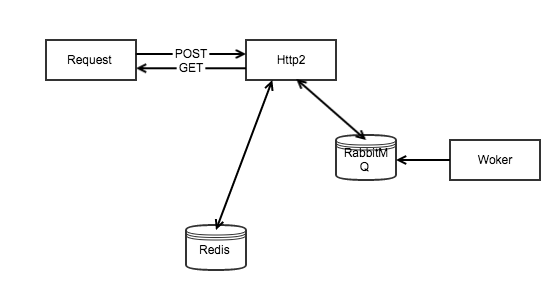
源码：<https://github.com/leitu/mse_advancedweb>

2 需求分析

在微服务架构下，根据各个服务之间暴露的HTTP公开服务，以JSON作为数据交换格式，相互转递信息。各个服务间只需将端点细节信息放到配置文件中。

前端web服务器提供GET,POST方法接口，接受内容，并进行分解和添加后，之后由后端的worker进行执行，减少各个不同的服务之前的相互影响，同时各个服务又能被其他服务所复用。

2.1 系统流程图



**3 工具介绍**

**3.1 HTTP2**

3.1.1 HTTP2介绍

几乎所有互联网上的内容都采用了HTTP 1.1作为通信协议, 但是目前的HTTP1.1协议过于庞大，而且很难榨干TCP协议中所能提供的所有性能，特别是HTTP1.1对网络延迟非常敏感，即便是我们拥有了高连接的速率，也难有优质快速的体验。虽然有像Spriting、Inlining这样的解决方案，但效果总是不让人满意，因此HTTP/2就诞生了（超文本传输协议第2版，最初命名为HTTP 2.0），是HTTP协议的的第二个主要版本，使用于万维网。HTTP/2是HTTP协议自1999年HTTP 1.1发布后的首个更新，主要基于SPDY协议。它由互联网工程任务组（IETF）的Hypertext Transfer Protocol Bis（httpbis）工作小组进行开发。该组织于2014年12月将HTTP/2标准提议递交至IESG进行讨论，于2015年2月17日被批准。 HTTP2标准于2015年5月以RFC 7540正式发表。

HTTP2包含了大部分HTTP1.1的特性，保留原有的URI，又提供如下:

1. HTTP2 采用二进制格式传输数据，已Frame作为二进制基础
2. HTTP2 对消息头采用 HPACK 进行压缩传输，能够节省消息头占用的网络的流量。
3. 多路复用，所有的请求都是通过一个 TCP 连接并发完成, 还支持优先级和流量控制。
4. Server Push：这个是HTTP2最重要的特性，在服务端能够更快的把资源推送给客户端。例如服务端可以主动把 JS 和 CSS 文件推送给客户端。
5. HTTP2 基于TLS。

可以根据[demo](https://http2.akamai.com/demo)看出HTTP2在性能加载上有绝对的优势。

HTTP2可以使用curl这样的工具来调试协议，而如果要进一步地分析网络数据流则需要诸如Wireshark这样的http2解析器。

**3.2 Microservices**

3.2.1 Microservices介绍

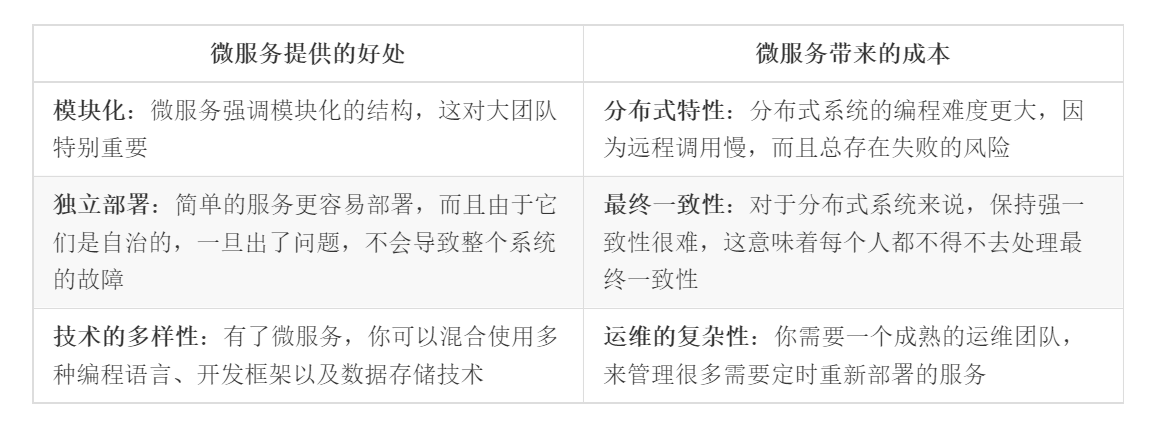
Microservices， 微服务是一种以业务功能为主的服务设计概念，每一个服务都具有自主运行的业务功能，对外开放不受语言限制的 API (最常用的是 HTTP)，应用程序则是由一个或多个微服务组成。

3.2.2 Microservices的特性

Microservices秉承了Unix理论“Do one thing and do it well", 还包含如下:

1. 每个服务都容易被取代。
2. 服务是以能力来组织的。
3. 服务可由不同的编程语言、数据库、硬件与软件环境实作，只在乎是否适配。
4. 架构是对称而非分层。

微服务是协同工作的小型自管理的服务。



**3.3 Docker**

3.3.1 Docker介绍

容器技术并不新鲜，早在1979年出现的在 Unix 系统中的 chroot 便是容器技术的雏形。而随后出现的 BSD Jail、Solaris Containers 和 OpenVZ 都算是容器技术的先驱。但容器技术开始普及却是在2007年，这一年 Google 贡献出 cgroups，并且从 2.6.4 开始，Linux 内核包含了这一组件，随后容器技术开始逐渐普及。但容器技术真正大放异彩则要等到2013年 Docker 0.10 版本发布。

Docker是一个开放源代码软件专案，让应用程序布署在软件容器下的工作可以自动化进行。

1. 借此在Linux操作系统上，提供一个额外的软件抽象层，以及操作系统层虚拟化的自动管理机制。
2. Docker利用Linux核心中的资源分离机制，例如cgroups，以及Linux核心命名空间（name space），来建立独立的软件容器（containers）。这可以在单一Linux实体下运作，避免启动一个虚拟机器造成的额外负担。Linux核心对命名空间的支援完全隔离了工作环境中应用程序的视野，包括行程树、网络、用户ID与挂载档案系统，而核心的cgroup提供资源隔离，包括CPU、内存、block I/O与网络。

Docker 的出现并非创造了一个新的容器技术，而是在 LXC (LinuX Container)、cgroups、namespaces 技术之上所构建的一种技术，Docker 包含两方面技术：

(1)容器技术： 有效分配与管理物理资源 实现资源隔离

(2)镜像技术： 打破“代码即应用”的观念 从系统环境开始，自底至上打包应用。

2.3.2 Docker Compose

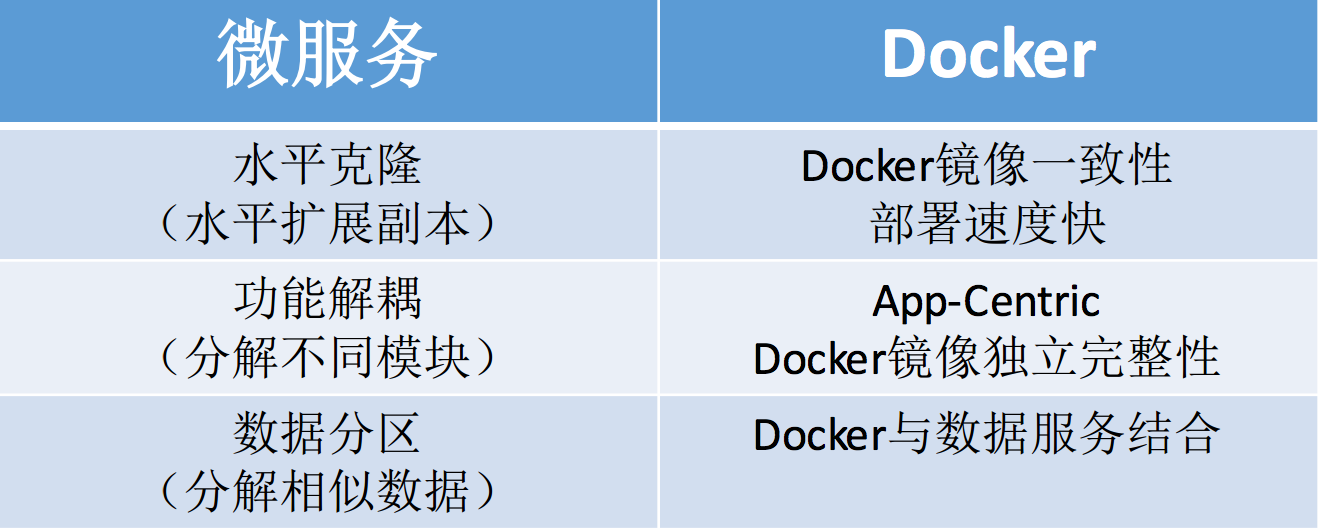
Docker Compse的前身是Fig, 可以让用户在集群中部署分布式应用.用compose file配置应用服务，快速部署多个应用程序。Docker Compose属于一个“应用层”的服务，用户可以定义哪个容器组运行哪个应用，它支持动态改变应用，并在需要时扩展。

2.3.3 Docker Hub

Docker Hub是docker提供的存放镜像的地方，许多官方提供和私人提供的镜像都存在与此，当然也可以自己搭registory.

2.3.4 微服务与Docker

微服务对于开发而言，只需要开发简单有效的模块，配置一个运行中的文件，不再是异常复杂的应用，对于运维而言，只需要管理硬件设施，监控并及时反馈，



通过用dockerfile的定义，build出docker镜像，然后在docker容器中运行。

**2.4** GO

2.4.1 Go介绍

Go,又称golang，是Google开发的一种静态强类型、编译型，并发型，并具有垃圾回收功能的编程语言, 目标之一就是迎合多核编程的挑战使得并发编程更加简便. Go不使用多线程模型,而是通过Go的routine（基于CSP的communication通道).任何Go语言中的函数,能够作为串行编程模型中的普通routine或者用关键字go在前端执行一个Goroutine,在运行时里面一个被调用的Goroutine是和调用者routine并发执行的，而不管他们是否运行在同一个CPU上.从编程这的角度来说,Go routine是并发的归宿,而且对比于之前的锁方式等等，有更加良好的确定性.Go在CSP的communication通道的基础上,扩展了无缓冲区大小的通道来实现异步发送(或写)操作.通道是Go语言的预定义类型(first-class),能够从一个routine传递到另一个routine.Goroutine和CSP通道的结合提供了一个强大的机制来来保证达到一个期望的确定性的并发计算。

**2.5** Redis

2.5.1 Redis介绍

Redis(REmote DIctionary Server)是一个开源、支持网络、基于内存、键值对存储数据库，将全部的数据存储在内存中,支持主从同步，基于 BSD 许可的，高级键值 (key-value) 缓存 (cache) 和存储 (store) 系统。由于 Redis 的键包括 string，hash，list，set，sorted set，bitmap 和 hyperloglog，所以常常被称为数据结构服务器。你可以在这些类型上面运行原子操作，例如，追加字符串，增加哈希中的值，加入一个元素到列表，计算集合的交集、并集和差集，或者是从有序集合中获取最高排名的元素。

为了满足高性能，Redis 采用内存 (in-memory) 数据集 (dataset)。根据你的使用场景，你可以通过每隔一段时间转储数据集到磁盘，或者追加每条命令到日志来持久化。持久化也可以被禁用，如果你只是需要一个功能丰富，网络化的内存缓存。Redis 还支持主从异步复制，非常快的非阻塞初次同步、网络断开时自动重连局部重同步。 其他特性包括：

1. 事务
2. 订阅/发布
3. Lua 脚本
4. 带 TTL 的键
5. LRU 回收健
6. 自动故障转移 (failover)

**2.6 RabbitMQ**

2.6.1 RabbitMQ介绍

RabbitMQ是实现了高级消息队列协议（AMQP）的开源消息代理软件，具有可伸缩性：群集服务 消息持久化：从内存持久化消息到硬盘，再从硬盘加载到内存。

RabbitMQ是实现AMQP（高级消息队列协议）的消息中间件的一种，最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

RabbitMQ主要是为了实现系统之间的双向解耦而实现的。当生产者大量产生数据时，消费者无法快速消费，那么需要一个中间层。保存这个数据。

几个概念说明：

Broker：简单来说就是消息队列服务器实体。

Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列。

Queue：消息队列载体，每个消息都会被投入到一个或多个队列。

Binding：绑定，它的作用就是把exchange和queue按照路由规则绑定起来。

Routing Key：路由关键字，exchange根据这个关键字进行消息投递。

Producer：消息生产者，就是投递消息的程序。

Consumer：消息消费者，就是接受消息的程序。

Channel：消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务。

**2.7** JSON

2.7.1 JSON介绍

JSON（JavaScript Object Notation）是一种由[道格拉斯·克罗克福特](https://zh.wikipedia.org/wiki/%E9%81%93%E6%A0%BC%E6%8B%89%E6%96%AF%C2%B7%E5%85%8B%E7%BE%85%E5%85%8B%E7%A6%8F%E7%89%B9)构想设计、轻量级的[数据交换语言](https://zh.wikipedia.org/w/index.php?title=%E8%B3%87%E6%96%99%E4%BA%A4%E6%8F%9B%E8%AA%9E%E8%A8%80&action=edit&redlink=1)，以文字为基础，且易于让人阅读。尽管JSON是[Javascript](https://zh.wikipedia.org/wiki/Javascript)的一个子集，但JSON是独立于语言的[文本格式](https://zh.wikipedia.org/wiki/%E6%96%87%E6%9C%AC%E6%96%87%E4%BB%B6)，并且采用了类似于[C语言](https://zh.wikipedia.org/wiki/C%E8%AA%9E%E8%A8%80)家族的一些习惯。JSON最开始被广泛的应用于WEB应用的开发。不过目前JSON使用在[JavaScript](https://zh.wikipedia.org/wiki/JavaScript)、[Java](https://zh.wikipedia.org/wiki/Java)、[Node.js](https://zh.wikipedia.org/wiki/Node.js)应用的状况比较多,目前也有用在NOSQL中，相对于传统的[关系型数据库](https://zh.wikipedia.org/wiki/%E5%85%B3%E7%B3%BB%E5%9E%8B%E6%95%B0%E6%8D%AE%E5%BA%93)，一些基于文档存储的[NoSQL](https://zh.wikipedia.org/wiki/NoSQL)非关系型数据库选择JSON作为其数据存储格式，比较出名的产品有：[MongoDB](https://zh.wikipedia.org/wiki/MongoDB)、[CouchDB](https://zh.wikipedia.org/wiki/CouchDB)、[RavenDB](https://zh.wikipedia.org/w/index.php?title=RavenDB&action=edit&redlink=1)等。

* 1. REST

REST(REpresentational State Transfer)这个概念，首次出现是在 2000年Roy Thomas Fielding（他是HTTP规范的主要编写者之一）的博士论文中，它指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful的。

要理解什么是REST，我们需要理解下面几个概念:

1. 资源（Resources） REST是"表现层状态转化"，其实它省略了主语。"表现层"其实指的是"资源"的"表现层"。
2. 那么什么是资源呢？就是我们平常上网访问的一张图片、一个文档、一个视频等。这些资源我们通过URI来定位，也就是一个URI表示一个资源。
3. 表现层（Representation）
4. 资源是做一个具体的实体信息，他可以有多种的展现方式。而把实体展现出来就是表现层，例如一个txt文本信息，他可以输出成html、json、xml等格式，一个图片他可以jpg、png等方式展现，这个就是表现层的意思。URI确定一个资源，但是如何确定它的具体表现形式呢？应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。
5. 状态转化（State Transfer）
6. 访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，肯定涉及到数据和状态的变化。而HTTP协议是无状态的，那么这些状态肯定保存在服务器端，所以如果客户端想要通知服务器端改变数据和状态的变化，肯定要通过某种方式来通知它。
7. 客户端能通知服务器端的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。

综合上面的解释，我们总结一下什么是RESTful架构：

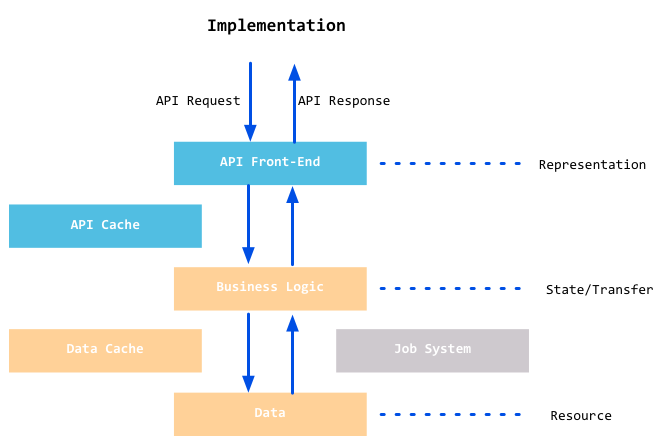
（1）每一个URI代表一种资源；

（2）客户端和服务器之间，传递这种资源的某种表现层；

（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

Web应用要满足REST最重要的原则是:客户端和服务器之间的交互在请求之间是无状态的,即从客户端到服务器的每个请求都必须包含理解请求所必需的信息。如果服务器在请求之间的任何时间点重启，客户端不会得到通知。此外此请求可以由任何可用服务器回答，这十分适合云计算之类的环境。因为是无状态的，所以客户端可以缓存数据以改进性能。

另一个重要的REST原则是系统分层，这表示组件无法了解除了与它直接交互的层次以外的组件。通过将系统知识限制在单个层，可以限制整个系统的复杂性，从而促进了底层的独立性。



REST是一种架构风格，汲取了WWW的成功经验：无状态，以资源为中心，充分利用HTTP协议和URI协议，提供统一的接口定义，使得它作为一种设计Web服务的方法而变得流行。在某种意义上，通过强调URI和HTTP等早期Internet标准，REST是对大型应用程序服务器时代之前的Web方式的回归。目前Go对于REST的支持还是很简单的，通过实现自定义的路由规则，我们就可以为不同的method实现不同的handle，这样就实现了REST的架构。

**4 详细设计**

前端服务页面，接受到POST 请求，解析POST请求后，将信息修改并加入其他特定的内容，将存入信息到RabbitMQ中，后端的则由worker 监听当前RabbitMQ的队列，当发现监听队列中出现新的消息后，取出消息中的特定项， 解析后由worker执行，并返回处理结果，redis模块作为key-value数据库，记录每个POST请求。 将各个服务封装在docker中，使用docker compose 对服务进行快速部署，节省开发时间介绍中间环节，并达到开发环境和真实生产环境的一致性。

4.1 前端RESTful

该模块利用go语言实现。

4.1.1 http服务

Go语言自带的net库,可以通过几行代码就启动一个http服务，在go语言升级大版本号至1.6以后，自带的net库支持http2，通过几行代码就能够实现一个web服务器的启动。

import “"golang.org/x/net/http2"

func main() {

var srv http.Server

http2.VerboseLogs = true

srv.Addr = ":8443"

// This enables http2 support

http2.ConfigureServer(&srv, nil)

http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "你在使用HTTP2协议访问 %q\n", html.EscapeString(r.URL.Path))

ShowRequestInfoHandler(w, r)

})

}

引入http2的库，启动编译后的二进制文件，访问本机<https://localhost:8443> 就会如图所示， 页面端已提供http2的服务，并强制使用TLS加密。



图中打印了欢迎句，HTTP版本， URL及Header信息。

3.1.2 REST实现

http1.1中定义了8种方法，在RESTful中最常用的就是GET,POST,PUT,DELETE这四种方法。

我们将指定 URL /api/post 为接受POST方法。

http.HandleFunc("/api/post", func(w http.ResponseWriter, r \*http.Request) {

//get "POST" request

if r.Method == "POST" {

decoder := json.NewDecoder(r.Body)

var t jsonData

err := decoder.Decode(&t)

if err != nil {

panic(err)

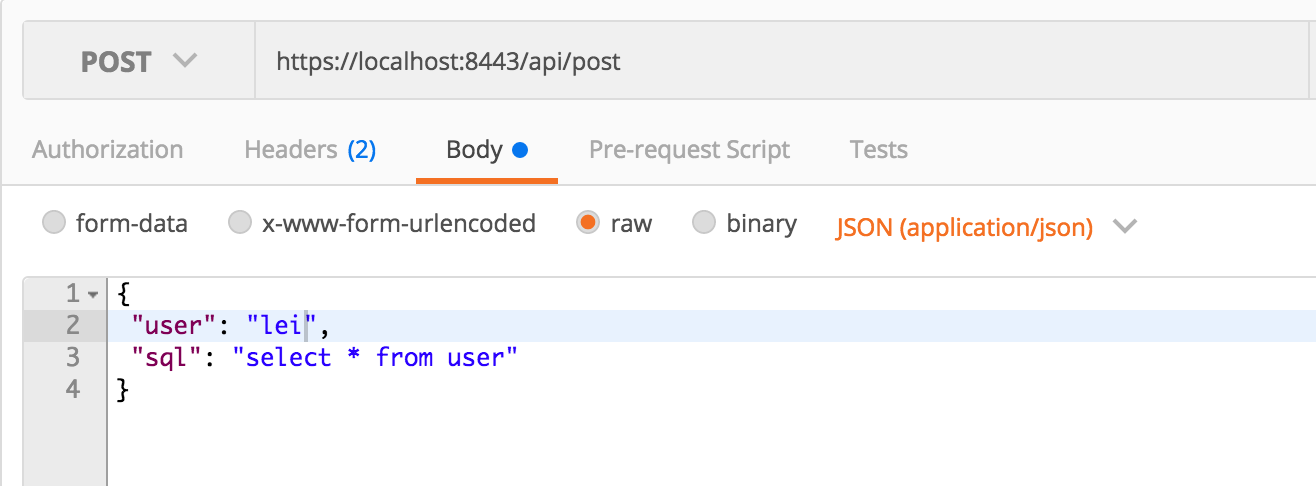
}

log.Println(t.Test)

}

})

当前端使用curl 或者 postman 将测试代码已JSON格式提交到<https://localhost:8443/api/post>的url当中。



此时服务器后台的log 就会打印如下日志记录:

2016/05/20 11:15:59 http2: server connection from 10.0.2.2:50529 on 0xc820012100

2016/05/20 11:15:59 http2: server: error reading preface from client 10.0.2.2:50529: EOF

2016/05/20 11:15:59 http2: server connection from 10.0.2.2:50531 on 0xc820012100

2016/05/20 11:15:59 http2: server: client 10.0.2.2:50531 said hello

2016/05/20 11:15:59 http2: server read frame SETTINGS len=12, settings: MAX\_CONCURRENT\_STREAMS=1000, INITIAL\_WINDOW\_SIZE=6291456

2016/05/20 11:15:59 http2: server processing setting [MAX\_CONCURRENT\_STREAMS = 1000]

2016/05/20 11:15:59 http2: server processing setting [INITIAL\_WINDOW\_SIZE = 6291456]

2016/05/20 11:15:59 http2: server read frame WINDOW\_UPDATE len=4 (conn) incr=15663105

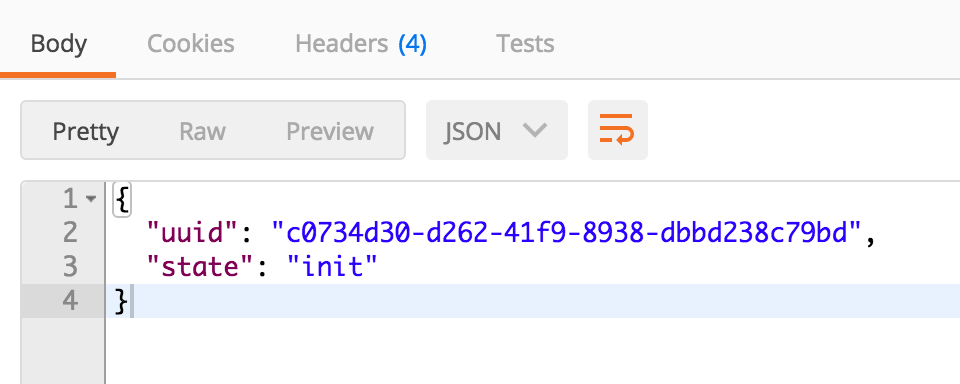
2016/05/20 11:15:59 http2: server read frame HEADERS flags=END\_HEADERS|PRIORITY stream=1 len=267

2016/05/20 11:15:59 http2: server read frame DATA flags=END\_STREAM stream=1 len=48 data="{\n \"user\": \"lei\",\n \"sql\": \"select \* from user\"\n}"

2016/05/20 11:15:59 http2: server read frame SETTINGS flags=ACK len=0

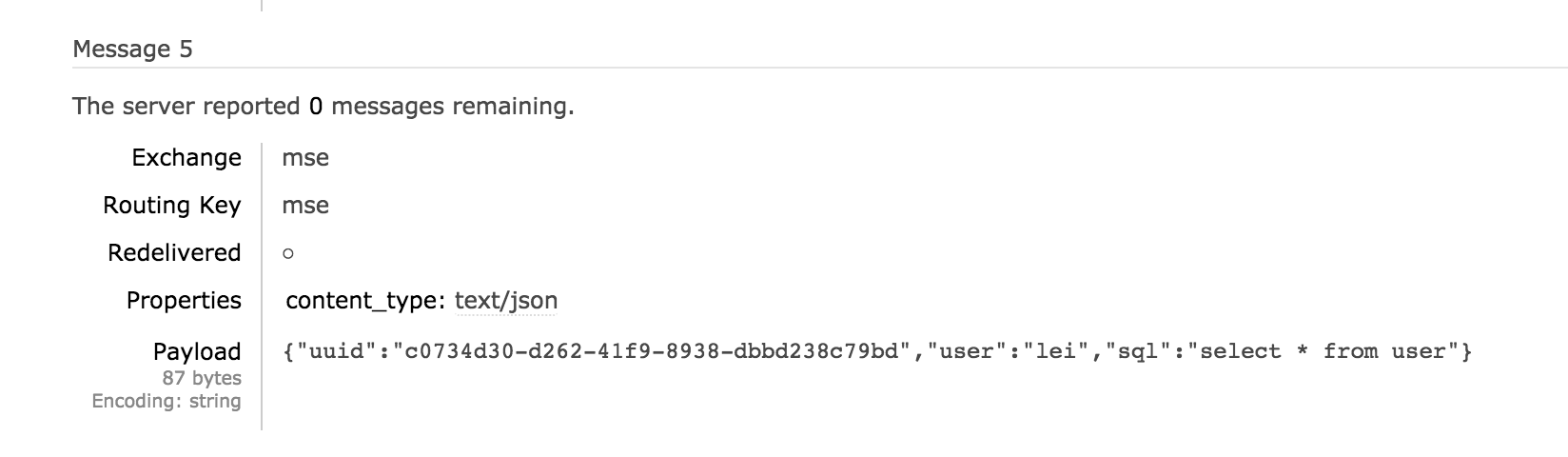
2016/05/20 11:15:59 {lei select \* from user}

同时对页面返回JSON数据，表示这个任务已被接受，并已经分发到rabbitmq中开始这个任务。我会对每个可执行任务分配一个uuid，这个uuid将会作为关键值，存在信息中，并在后台的记录中这个uuid作为唯一标志符，对数据进行传输。



3.2 后端消息处理

当数据从前端的request body 进行解析后，将所需要部分并附上之前所产生的uuid 一并传入RabbitMQ的队列中， 具体信息可以从RabbitMQ的后台管理页面中找到，在后端的worker就可以监听数据消息的传入，将数据payload中的sql 语句进行提取，在后端执行，并返回结果。



Redis将会作为一个数据记录的NOSQL数据库， 记录该时间的状态。对于redis的键值，同样的已uuid作为key， 并传入status和httpstatus的返回值， 200则表示http的状态是好的。

127.0.0.1:6379> HMGET c0734d30-d262-41f9-8938-dbbd238c79bd status

1) "init"

127.0.0.1:6379> HMGET c0734d30-d262-41f9-8938-dbbd238c79bd httpStatus

1) "200"

3.3 快速部署

何为快速部署，这要从敏捷开发说起， 敏捷开发是一种已人为核心，迭代，循序渐进的开发方式。目标是持续提升整个团队的产品能力。Docker在业内已其轻量级的设置， 逐渐成为开发人员在迭代开发中的主要手段，同时在部署中也为开发人员带来方便。

在部署中已ubuntu的14.04 版本作为运行系统， 将预先生成的server.crt/server.key和编译后的二进制文件放在同一目录之中， 挂载当前目录并启动该服务。生成对应的dockerfile，将配置文件写入。

Dockerfile

FROM ubuntu:14:04

RUN mkdir /code

WORKDIR /code

ADD . /code/

CMD [“./code”]

简单的通过dockerfile就完成了镜像的打包，后端使用官方提供的redis及RabbitMQ镜像,减少对外界的依赖，同时保证数据的正确性。

3.3.2 使用docker-compose进行部署

Docker-compose是在docker收购fig 这一开源工具后，适时的推出了基于docker本身的快速部署工具，写法简单，语义明确。利用yaml文本格式，简单明了的表达了各个container 之间的关系。

version: '2'

services:

web:

build: .

volumes:

- .:/code

ports:

- "8443:8443"

depends\_on:

- redis

- rabbitmq

links:

- redis

- rabbitmq

redis:

image: redis:latest

ports:

- "6379:6379"

rabbitmq:

image: rabbitmq:latest

ports:

- "5672:5672"

environment:

RABBITMQ\_DEFAULT\_USER: test

RABBITMQ\_DEFAULT\_PASS: test

只要运行docker-compose run --service-ports web，则服务器端就会被快速部署，不需要人工参与。docker-compose会自己找到docker-compose.yml文件，根据语义，在web模块会根据dockerfile进行前端的编译，同时将rabbitmq和redis镜像下载到本地，并暴露端口给web。访问https://localhost:8443即可达到效果。

**结论**

在本次实践过程中，系统版本是基于ubuntu的，安装了docker-engine，通过postman进行POST方法后，在终端会返回机器的日志。 在这个实践中并没有增加监控功能， 微服务的架构中产生的分布式及模块化，在运维过程中因为这些的出现导致这些服务严重依赖于系统日志和系统状况， 对微服务的监控就显得格外的重要，因为在生产环境中出现的失败，和迭代的生产比较不容易被发现，各种微服务之间的联系节点也变得更加复杂，如果一个节点消失对整个环境将会造成不可预估的后果，所以如何实现一个完整的监控体系，合理的收集日志并进行利用分析。

与此同时google在实验QUIC(快速UDP互联网连接）协议，它在很大程度上继承了SPDY的衣钵。QUIC是一个UDP版的TCP + TLS + HTTP/2替代实现。QUIC可以创建更低延迟的连接，并且也像HTTP/2一样，通过仅仅阻塞部分流解决了包裹丢失这个问题，让连接在不同网络上建立变得更简单.

参考：

[1] https://http2.akamai.com/demo