Nginx HTTP Load-Balance

# 概述

负载均衡包括二个方面：

* Worker Process

Worker进程之间的负载均衡，具体是指Worker处理连接(Connection)数量的平衡。它由Accept Mutex实现。

* Upstream Server

Nginx作为反向代理服务器运行时，负载均衡的目的是将Client的请求均匀地分流到上游服务器，以达到上游服务器群中的各台服务器的负荷比较平均。

具体有Round Robin、IP Hash、Least Conn 等策略。

# Accept Mutex

## 惊群与竞争

所有的Worker都监听相同的Listen Socket以获得新的连接(Connection)。当一个新的连接到达时，所有的Worker都会从epoll\_wait()被唤醒。

都收到一个Accept事件，然后开始竞争Accept。只有一个Worker能accept成功，其余的Worker都accept失败，做了一次无用功。

这种现象形象地称为“惊群(Thundering Herd)”。

因为竞争的随机性，有可能导致的结果是某个Worker进程Accept很多Connection，而某个Worker却很少，造成负载不均衡。

## 负载度量

以最大连接数的7/8为基准，如果实际连接数超过它则表示Worker过载。

ngx\_accept\_disabled = used\_connection\_n – connection\_n \* 7/8

= connection\_n – free\_connection\_n – connection\_n \* 7/8

= connection\_n / 8 – free\_connection\_n

ngx\_accept\_disabled是Worker负载轻重的量化。数值越大，表示负载越重，数值越少，表示负载越轻。大于0，表示过载。

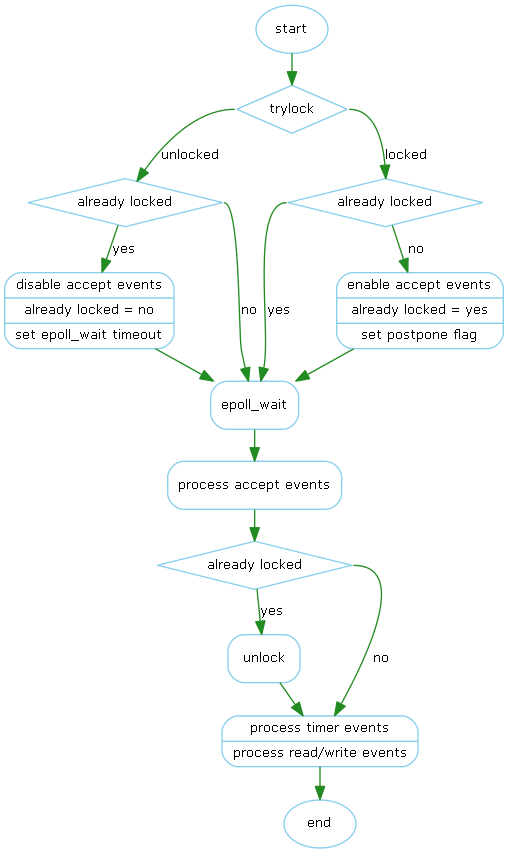
经过上述公式的推导，降低了计算量。

ngx\_accept\_disabled在接受一个新连接时重新计算。

## 解决方案

只要满足同一时刻只有一个Worker在监听Listen Socket，就不会出现“惊群”问题。

设置一把锁，Worker获得（加锁）锁，便有监听Listen Socket的权力，否则没有。于是“惊群”问题转化为竞争锁问题。



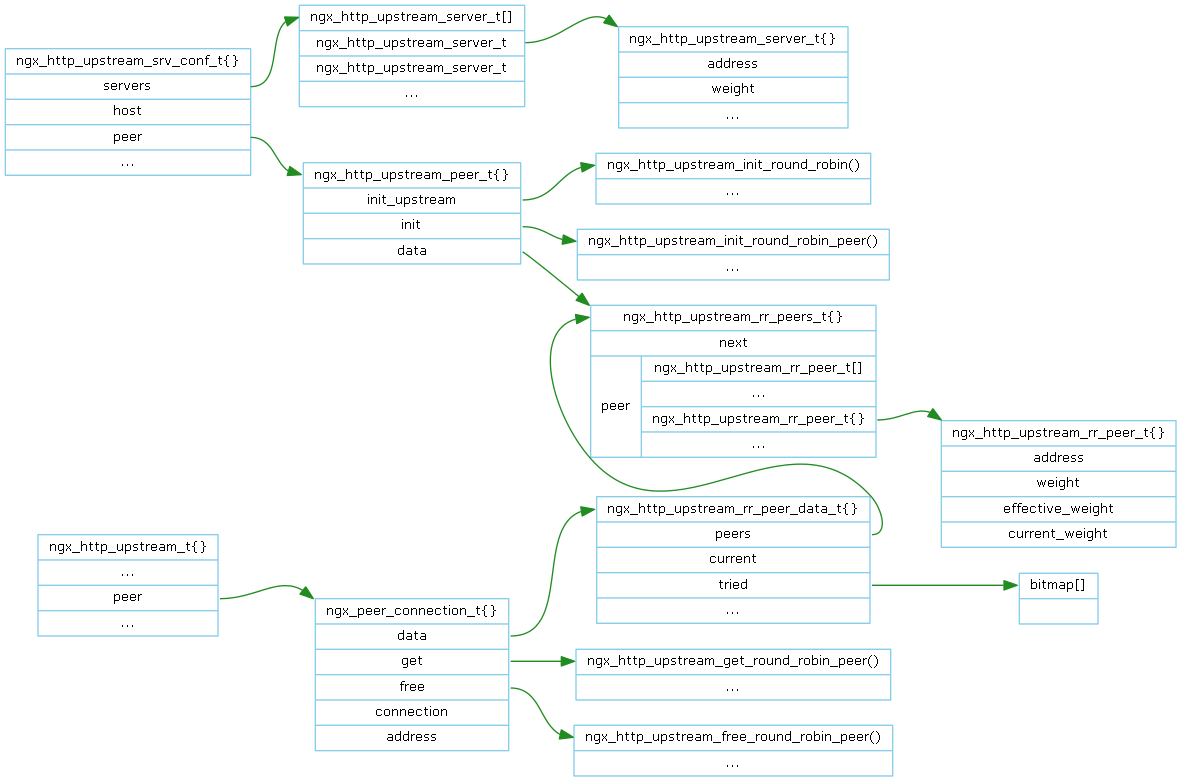
## 负载控制

TODO

# Round Robin

Round Robin以轮询的方式选择上游服务器，Nginx采用了Weighted Round Robin（WRR）算法。

## 数据结构



在<Nginx HTTP Proxy>中，我们已经知道ngx\_http\_upstream\_srv\_conf\_t是一个上游服务器群。

负载均衡是服务器群上的一个方法，于是由ngx\_http\_upstream\_srv\_conf\_t来管理它的负载均衡策略非常合理。

ngx\_http\_upstream\_peer\_t管理各种负载均衡策略的抽象数据结构。

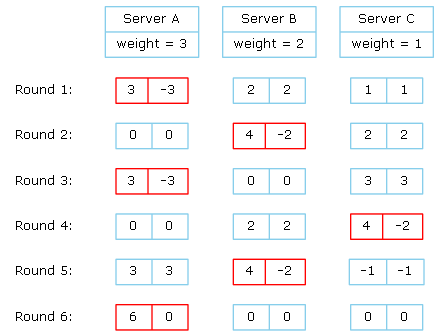
ngx\_http\_upstream\_rr\_peers\_t和ngx\_http\_upstream\_rr\_peer\_t是实现Round Robin策略的关键数据结构。

ngx\_http\_upstream\_rr\_peer\_t代表一台上游服务器，包含address和权重信息。

ngx\_http\_upstream\_rr\_peers\_t代表一个上游服务器群，包含一个ngx\_http\_upstream\_rr\_peer\_t数组。

Ngx\_http\_upstream\_rr\_peer\_data\_t执行负载均衡的数据结构。

## 算法实现



算法伪码如下：

FOR EACH peer IN peers :

peer.current\_weight += peer.effective\_weight

IF peer.current\_weight is heaviest :

best = peer

ENDIF

ENDFOR

best.current\_weight -= total\_weight

RETURN best

## 调用过程

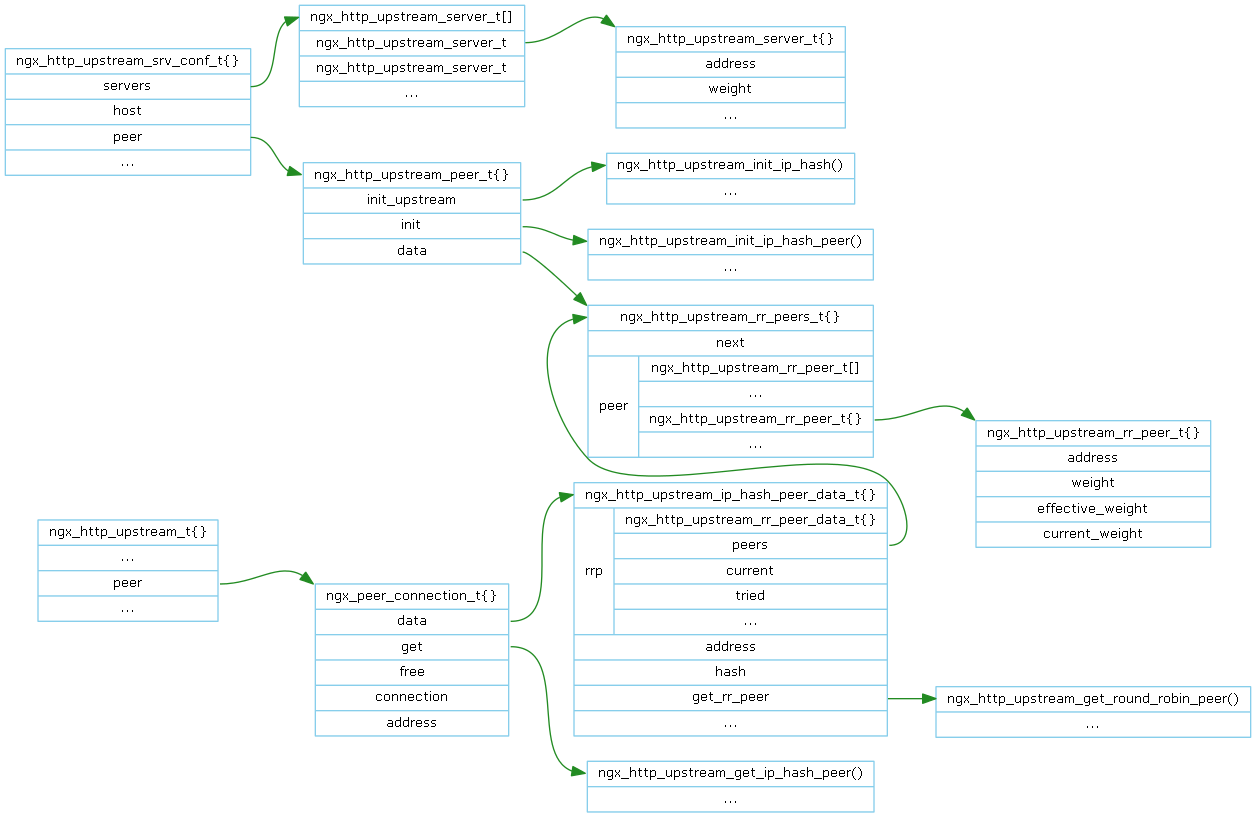
ngx\_http\_upstream\_connect()🡪ngx\_event\_connect\_peer()🡪ngx\_http\_upstream\_get\_round\_robin\_peer()

## 容错处理

# IP Hash

IP Hash根据Client的IP地址选择上游服务器，保证相同Client IP地址的不同请求被分配到同一个上游服务器。

## 数据结构



## 算法实现

### IP地址HASH

hash = 89

FOR EACH 3 bytes IN ip\_address :

hash = (hash \* 113 + byte) % 6271;

ENDFOR

RETURN hash

### 无权重

index = hash % number

RETURN peers[index]

### 有权重

w = hash % total\_weight

FOR EACH peer IN peers :

w -= peer.weight

IF w < 0 :

RETURN peer

ENDIF

ENDFOR

# Least Conn

略

# 自定义策略

从上述几个策略的实现，我们可以看出Load Balance框架和接口。

## 一个框架

## http_load_balance_interface.png四个接口

* init\_upstream

配置过程中设置（参考ip\_hash directive）。它的主要责任是建立负载均衡策略的数据结构（由ngx\_http\_upstream\_peer\_t的data管理）。

设置init接口。

* init

它由init\_upstream设置。它的主要责任是建立运行时状态数据结构（由ngx\_peer\_connection\_t的data管理）。

设置get/free接口。

* get

它由init设置。它的主要责任是执行算法选择一个服务器，并将地址address写入ngx\_peer\_connection\_t的address。

* free

它由init设置。TODO

# Prototype

暂无

# Latest revision

https://github.com/lingjf/nginx\_analyse/blob/master/doc/