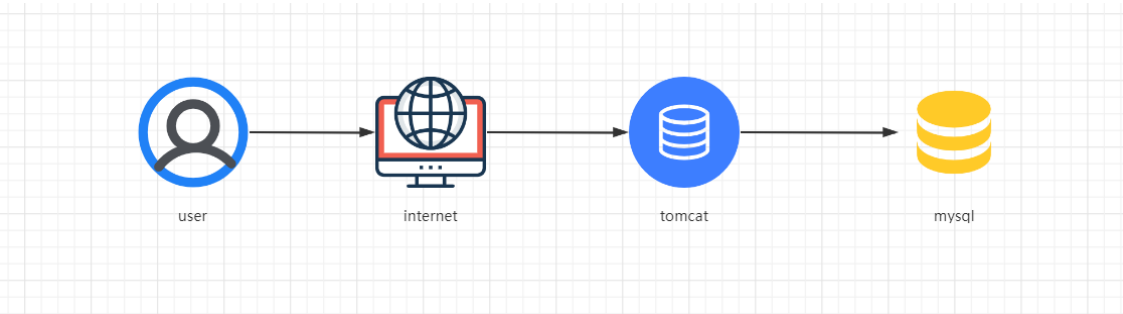
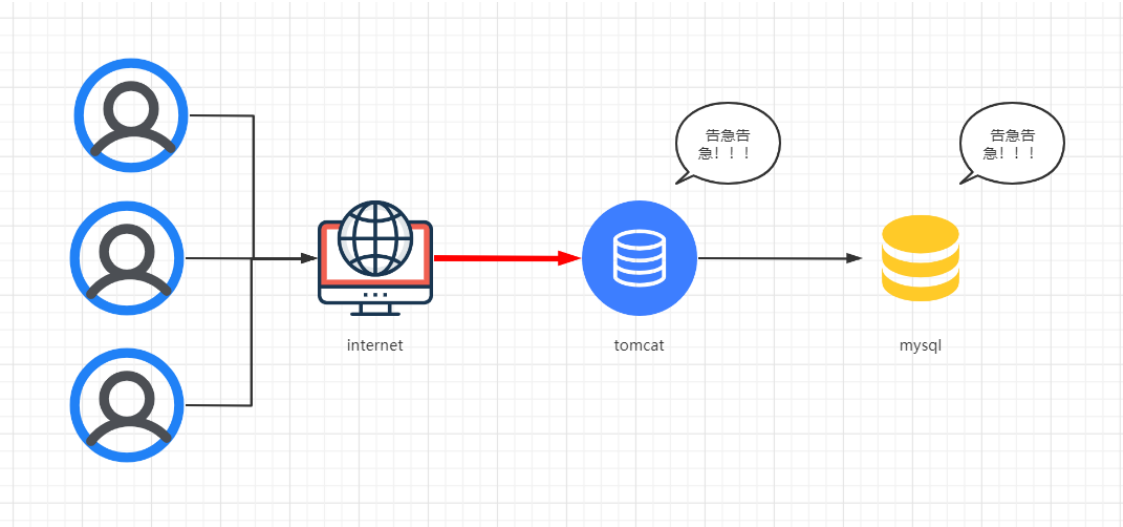
什么是集群？什么是负载均衡器？什么是四层负载均衡？什么是七层负载均衡？他两区别是什么？Nginx作为一个负载均衡器如何使用？对负载均衡算法还有点模糊？本篇文章，将这些基础知识点一网打尽！

一、集群和负载均衡概述

相对于集群，单体是咱们平时接触最多的，比如笔者曾经自己搭建的商城项目，架构是：



但是，当用户暴增的时候，就会出现服务器压力过大，扛不住那么多的请求同时进来了：



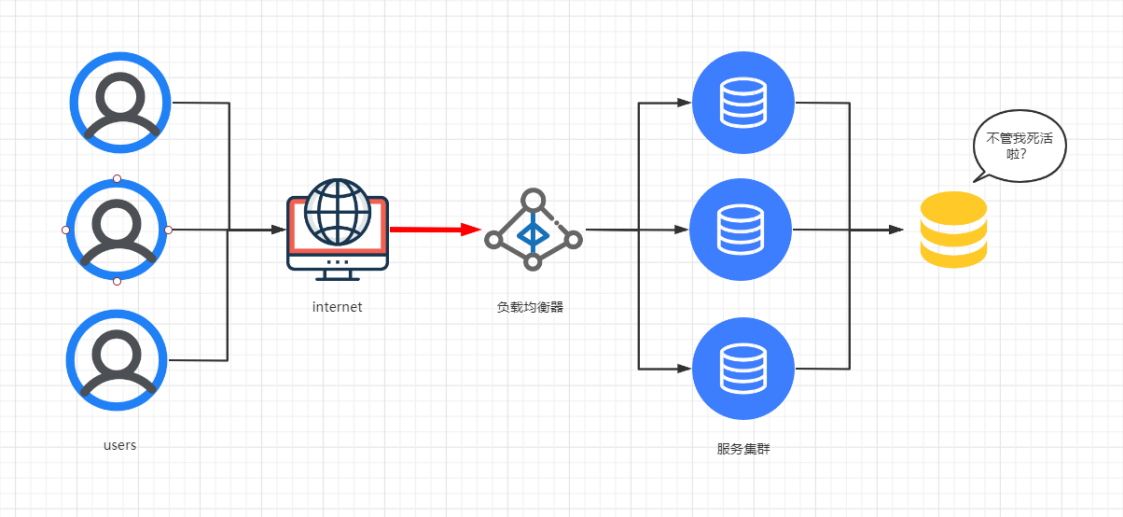
MySQL压力大时，往往会采取分库分表、增加缓存层、消息队列层等方式来做，我们不做细致讨论，仅针对咱们的tomcat服务进行考虑，可以将其横向扩展为集群服务。

什么叫集群？集群简单来说，就是单机处理到达瓶颈后，需要将单机复制几份，这样就构成了集群。**每个节点提供相同的服务**，整体处理能力和稳定性可以得到较大提升。

用上了集群，系统扩展就十分简单了，但是有个问题是：**用户的请求究竟由哪个节点来处理呢**？最好能够让此时此刻负载较小的节点来处理，这样使得每个节点的压力都比较平均。要实现这个功能，就需要在所有节点之前增加一个“调度者”的角色，用户的所有请求都先交给它，然后它根据当前所有节点的负载情况，决定将这个请求交给

哪个节点处理。这个“调度者”有个牛逼了名字—**负载均衡服务器**，而如何调度就取决于使用哪种**负载均衡算法**。

那么理论上，我们的商城用上集群后，将会变成这个样子：



当然了，细心的读者会发现，负载均衡器也可能需要用上集群，没错，如果负载均衡器都挂了，那么后面堆再多的机器也枉然！不过我们先聚焦应用服务器扩展为集群，关于负载均衡器比如nginx的高可用后续文章就会说到。

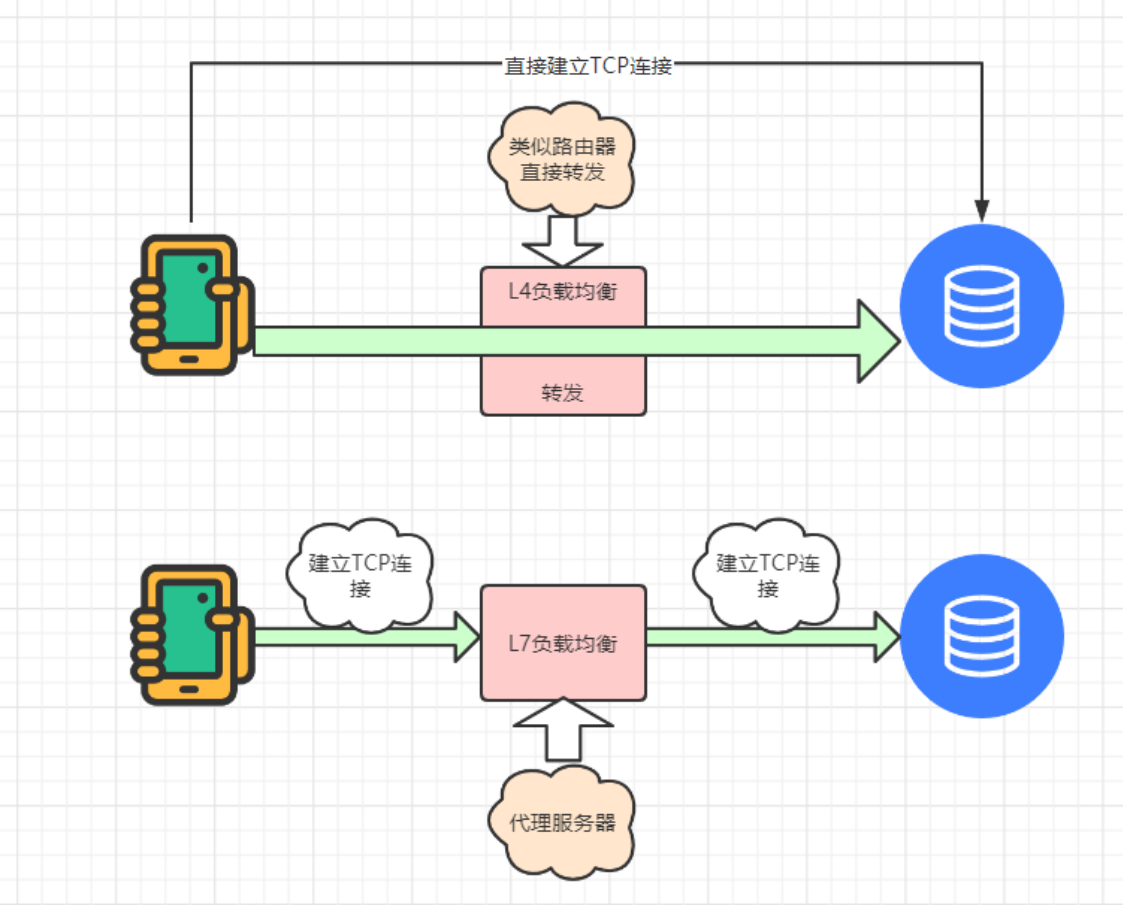
> 软件的世界就是如此简单而又复杂，简单之处在于往往可以通过增加一层来解决，不过增加一层，又要考虑引入的代价，整个系统就在这样的迭代中，逐渐变得复杂，但是可以达成最终的目标。

二、四层和七层负载均衡原理简述和区别

2.1、四层和七层负载均衡

负载均衡方案的选择有两种：四层负载均衡和七层负载均衡。熟悉TCP/IP网络分层模型的话，很容易知道四层对应传输层，七层对应应用层。

他们两者是什么原理呢？有什么区别呢？先贴一张图，下面进行详细说明：



所谓四层负载均衡，也就是主要通过**IP和端口**，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

> 四层负载均衡具体实现方式为：通过报文中的IP地址和端口，再加上负载均衡设备所采用的负载均衡算法，最终确定选择后端哪台服务器。以TCP为例，客户端向负载均衡发送SYN请求建立第一次连接，通过配置的负载均衡算法选择一台后端服务器，并且将报文中的IP地址信息修改为后台服务器的IP地址信息，因此TCP三次握手连接是与后端服务器直接建立起来的。

因此我们的四层负载均衡器主要充当的是转发的角色，类似于路由器这样的设备，仅仅是转发，区别于下面要说的七层负载均衡器。

所谓七层负载均衡，也称为“内容交换”，也就是主要通过**报文中的真正有意义的应用层内容**，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

上面我们说四层负载均衡器主要充当的是转发的角色，**但是七层负载均衡器就不一样了，它的身份更像是一个代理服务器，请求方需要先与负载均衡设备建立TCP连接，然后负载均衡设备再与后端服务器建立另外一条TCP连接通道**。

因此，**七层设备在网络性能损耗会更多一些**，处理七层的能力也必然会低于四层模式的部署方式。

2.2、四层和七层的对比

四层负载均衡与服务器直接建立起TCP连接，很容易遭受`SYN Flood`攻击。`SYN Flood`是一种广为人知的`DDoS`（分布式拒绝服务攻击）的方式之一，这是一种利用TCP协议缺陷，发送大量伪造的TCP连接请求，从而使得被攻击方资源耗尽的攻击方式。从技术实现原理上可以看出，**四层负载均衡很容易将垃圾流量转发至后台服务器，而七层设备则可以过滤这些恶意并清洗这些流量**，但要求设备本身具备很强的抗`DDOS`流量的能力。

四层负载工作模式简单，负载性能高，后台服务器都必须承载相同的业务。

七层负载工作模式复杂，性能消耗高，但带来了更好的灵活度，更有效的利用资源，加速对资源的使用。

常见的四层负载均衡一般可以通过如下方式：

- F5硬负载均衡
- LVS四层负载均衡
- Haproxy四层负载均衡
- Nginx四层负载均衡

七层负载均衡一般可以通过如下方式：

- Nginx七层负载均衡
- Haproxy七层负载均衡
- apache七层负载均衡

我们有个大概印象即可。

2.3、基于其他层次的负载均衡

同理，有**基于MAC地址信息**(通过一个虚拟MAC地址接收请求，转发给真实MAC地址)进行转发的**二层负载均衡**。

还有**基于IP地址**(通过一个虚拟IP地址接收请求，然后再分配到真实的IP地址)的**三层负载均衡**。

- **二层负载均衡**会通过一个虚拟 MAC 地址接收请求，然后再分配到真实的 MAC 地址；
- **三层负载均衡**会通过一个虚拟 IP 地址接收请求，然后再分配到真实的 IP 地址；
- **四层负载均衡**通过虚拟 IP + 端口接收请求，然后再分配到真实的服务器；
- **七层通过均衡**虚拟的 URL 或主机名接收请求，然后再分配到真实的服务器；

2.4、基于DNS地域负载均衡

所谓DNS地域是基于地理的负载均衡，例子：北方的用户访问北京的机房，南方的用户访问深圳的机房,DNS负载均衡的本质是DNS解析同一个域名可以访问不同的IP地址。

例如：同样是`www.baidu.com`,北京用户解析后获取的地址是`61.135.165.224`(北京机房IP),南方用户解析后获取的地址是`14.215.177.38`(深圳机房IP)

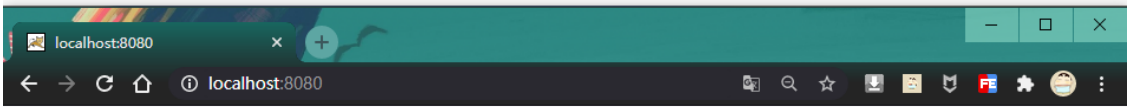
三、搭建Nginx+Tomcat集群

本地准备两个tomcat，监听的端口分别是8080和8090两个端口，我对ROOT目录下的`index.jsp`做下简单修改，比如：

```
<!DOCTYPE html>
<html lang="en">
    <head>
    </head>
    <body>
         <h1>I am Tomcat1,listening on 8080!</h1>
    </body>
</html>
```

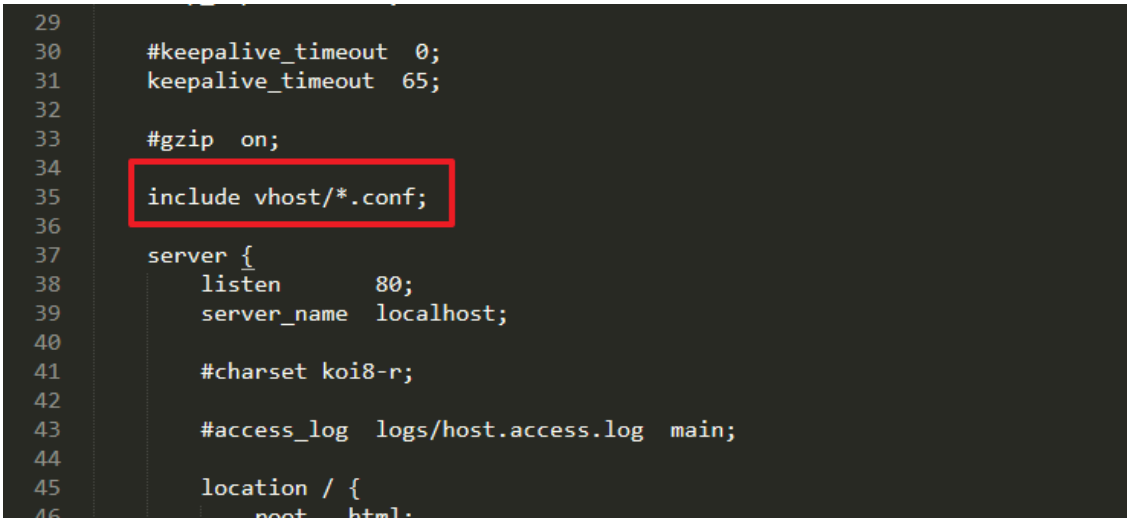
我可以通过首页轻松分别出哪台tomcat给我们做了响应，效果如下：

I am Tomcat2,listening on 8090!



I am Tomcat1,listening on 8080!

好了，下面就是配置 `Nginx` 来做负载均衡器。我直接在 `nginx.conf` 中增加 `include vhost/*.conf;` 配置：



那么我在 `nginx.conf` 同级目录中新建文件夹 `vhost`，然后在新的文件夹下新建文件，比如 `mytomcats.conf`，只要后缀是 `.conf` 的文件，`nginx` 加载的时候就会扫描到。

在 `Nginx` 中，有个关键词叫做 `upstream`，中文翻译过来叫做“上游”。我们的两个tomcat作为nginx的后端服务器，就是所谓的上游。我们可以通过这个关键词配置后端多个服务器集群，让他们轮流为用户服务。废话不多说，我的 `mytomcats.conf` 配置为：

```
upstream mytomcats{
    server www.tomcats.com:8080;
    server www.tomcats.com:8090;
}

server {
  listen 80;
  autoindex on;
  server_name www.tomcats.com;

  location / {
    proxy_pass http://mytomcats;
  }
}
```

可以看到，我可以对 `upstream` 随便起一个名字，我这里叫做 `mytomcats`，下面配置了两个 `server`，就是代表着我们的两个后端tomcat，这里的 `www.tomcats.com` 也是我随便起的。下面还要配置 `server`，因为我要通过 `nginx` 去访问就需要配置，`nginx` 我配置的监听端口是默认的80端口，下面的 `server_name` 我写的是上面起的域名。所以很显然，我要想访问这个域名，就需要

在本地做下域名解析，因此我会在我的 `hosts` 文件中配置：

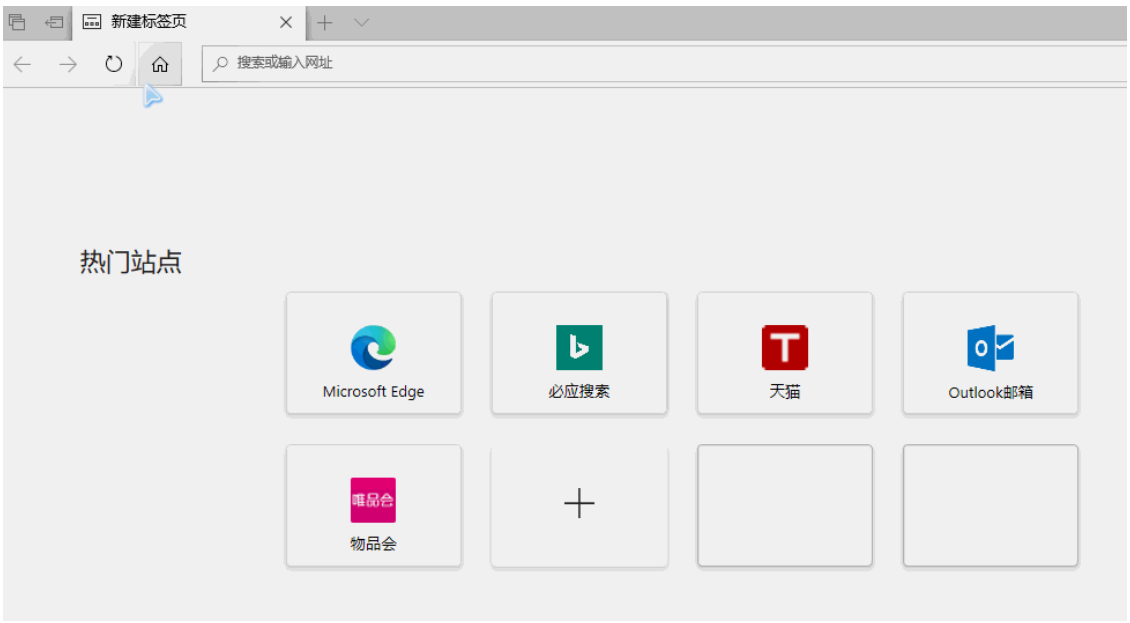
```
127.0.0.1 www.tomcats.com
```

接下来就是 `location` 模块，核心的是 `proxy_pass` 字段，中文翻译过来意思是代理转发，也就是说，我请求了 `nginx` 的80端口后，它会帮助我转发给 `http://mytomcats` 服务，而这个 `mytomcats` 恰好就是咱们上面配置的 `upstream` ，那么就会根据 `upstream` 配置的后端服务器响应用户请求。

下面就是启动 `nginx` ，启动前先去检测下配置文件是否有问题：

```
swg@LAPTOP-QVVADAEN MINGW64 /d/nginx-1.12.2
$ ./nginx.exe -t
nginx: the configuration file D:\nginx-1.12.2/conf/nginx.conf
syntax is ok
nginx: configuration file D:\nginx-1.12.2/conf/nginx.conf test is
successful
```

好了，看来我们的配置没啥大毛病，那么就启动吧： `./nginx.exe` 。



好了，我们可以看到，默认使用的是轮询算法，也就是说，大家的活平摊干，公平公正，最简单。但是现实往往没有这么美好。

四、upstream模块指令参数

官方文档：`http://nginx.org/en/docs/http/ngx_http_upstream_module.html` ，这里说明的都是关于 `ngx_http_upstream_module` 模块的内容。

可以使用 `upstream` 定义一组服务器，通过 `server` 指令指定每台服务器：

```
http {
    upstream backend {
        server backend1.example.com;
        server backend2.example.com;
    }
}
```

4.1、max_conns和queue

可以通过max_conns参数来限制连接数，如果达到了最大连接数，请求就可以放到queue中来等待后续的处理 ，queue指令设置了可以放到队列中的最大的请求数。

queue参数：如果在处理请求时无法立即选择上游服务器，则该请求将被放入队列中。该指令指定number可以同时在队列中的最大请求数。如果队列已满，或者在timeout参数指定的时间段内无法选择将请求传递给的服务器，则会将502（错误网关）错误返回给客户端。

当使用默认循环方法以外的负载平衡器方法时，有必要在queue指令之前将其激活。该timeout参数的默认值为60秒。

配置参考如下：

```
# worker进程设置1个，便于测试观察成功的连接数 worker_processes  1;

upstream tomcats {
    server 192.168.1.173:8080 max_conns=2;
    server 192.168.1.174:8080 max_conns=2;
    server 192.168.1.175:8080 max_conns=2;

    queue 100 timeout=70;
}
```

值得注意的是对于max_conns参数：

> Since version 1.5.9 and prior to version 1.11.5, this parameter was available as part of our commercial subscription.

即1.5.9-1.11.5是作为商用版本提供的，不过现在的版本已经远超1.11.5了，所以我们可以免费使用。

queue参数直接就是商用版本才可使用：

> This directive is available as part of our commercial subscription.

4.2、slow_stat

> sets the time during which the server will recover its weight from zero to a nominal value, when unhealthy server becomes healthy, or when the server becomes available after a period of time it was considered unavailable. Default value is zero, i.e. slow start is disabled.

- 即服务器慢启动，对于刚刚恢复的服务器，如果一下子被请求淹没可能会再次宕机。
- 可以通过server指令的slow_start参数来让其权重从0缓慢的恢复到正常值。
- 默认值为0，表示禁用慢启动。

配置参考如下：

```
upstream tomcats {
    server 192.168.1.173:8080 weight=6 slow_start=60s;
    # server 192.168.1.190:8080;
    server 192.168.1.174:8080 weight=2;
    server 192.168.1.175:8080 weight=2;
}
```

注意：

- 该参数不能使用在 hash，ip_hash，random load balancing 中。
- 如果在 upstream 中只有一台 server，则该参数失效。

4.3、down和back_up

down 用于标记服务节点不可用：

```
upstream tomcats {
    server 192.168.1.173:8080 down;
    server 192.168.1.174:8080 weight=1;
    server 192.168.1.175:8080 weight=1;
}
```

backup 表示当前服务器节点是备用机，只有在其他的服务器都宕机以后，自己才会加入到集群中，被用户访问到：

```
upstream tomcats {
    server 192.168.1.173:8080 backup;
    server 192.168.1.174:8080 weight=1;
    server 192.168.1.175:8080 weight=1;
}
```

注意：backup 参数不能与 hash，ip_hash和随机 负载平衡方法一起使用。

4.4、max_fails和fail_timeout

当nginx认为一个server不可用，它会暂时停止向这个server转发请求直至nginx再次认为它是可用的。

nginx通过两个参数来控制nginx的判断：

- fail_timeout，当在fail_timeout时间段内，失败次数达到一定数量则认为该server不可用。并在接下来的fail_timeout时间内不会再将请求打到这个server上。
- max_fails，这个max_fails就是上面说的失败的一定数量。
- 默认的值是10秒和1次尝试。

假设目前设置如下：

```
max_fails=2 fail_timeout=15s
```

则代表在15秒内请求某一server失败达到2次后，则认为该server已经挂了或者宕机了，随后再过15秒，这15秒内不会有新的请求到达刚刚挂掉的节点上，而是会达到正常运作的server上，15秒后会再有新请求尝试连接挂掉的server，如果还是失败，重复上一过程，直到恢复。

五、负载均衡算法实践

我们说，轮询是最简单的方式，每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。虽然这种方式简便、成本低廉。但不一定适合于所有场景。下面我们介绍几种其他的负载均衡策略。

5.1、权重

由于我 `tomcat1` 性能更加强劲，而 `tomcat2` 实属弱鸡，所谓能者多劳，我能不能让强者多承担一点工作？那么就可以通过配置权重的方式来照顾下 `tomcat2` 。我改为了如下配置，注意看到我在后面增加了 `weight` ，翻译过来是体重，没错，吃的多，干的就要多！

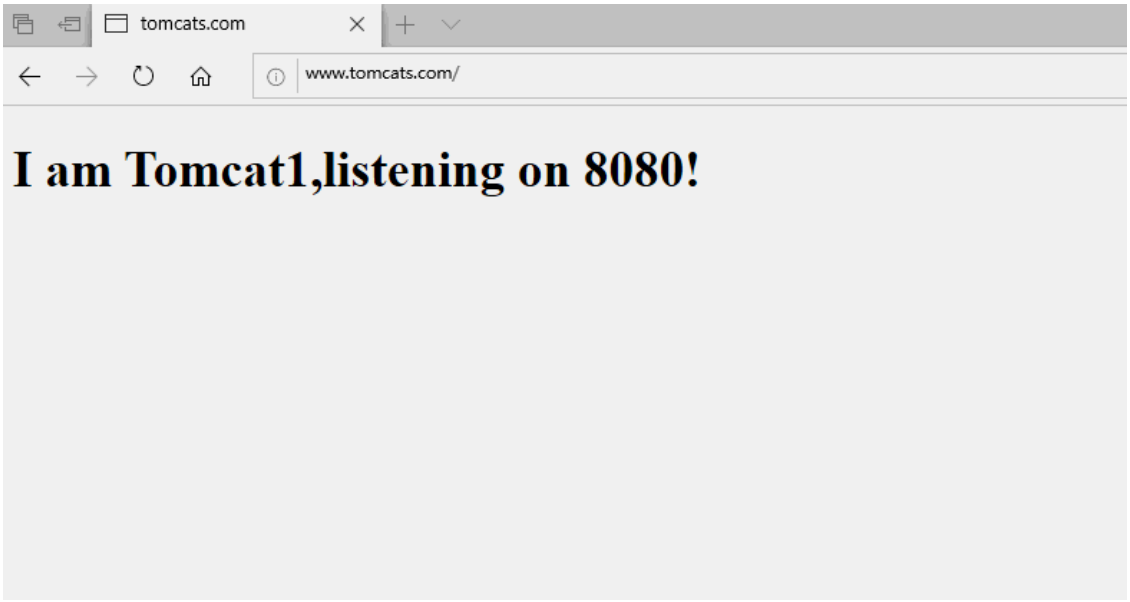
```
```

```
upstream mytomcats{
    server www.tomcats.com:8080 weight=5;
    server www.tomcats.com:8090 weight=2;
}

server {
  listen 80;
  autoindex on;
  server_name www.tomcats.com;

  location / {
    proxy_pass http://mytomcats;
  }
}
```

我们来实际看下效果：



可以看到，`tomcat1`的出勤次数明显增多，出现的频次大概是`tomcat2`的两倍多。

5.2、ip_hash

如果出现这个需求：要求用户进来后，每次分配的服务器都不变，如何做？可以用`ip_hash`来实现，**使得每个访客固定访问一个后端服务器**。

可以看到，永远都是1为我服务了。

ip_hash 可以保证用户访问可以请求到上游服务中的固定的服务器，前提是用户ip没有发生更改。 （但是处于公网的用户一般IP都是动态变化的）

结合官方文档，我们可以看到有一个注意点说明：

> Specifies that a group should use a load balancing method where requests are distributed between servers based on client IP addresses. The first three octets of the client IPv4 address, or the entire IPv6 address, are used as a hashing key. The method ensures that requests from the same client will always be passed to the same server except when this server is unavailable. In the latter case client requests will be passed to another server. Most probably, it will always be the same server as well.
> 这个负载均衡算法是根据客户端IP地址在服务器之间分配请求。**客户端IPv4地址的前三个八位位组或整个IPv6地址用作哈希密钥**。该方法确保了来自同一客户端的请求将始终传递到同一服务器，除非该服务器不可用。在后一种情况下，客户端请求将传递到另一台服务器。最有可能的是，它也将永远是同一台服务器。

因此如果是前三位都一样的IPV4地址，比如局域网中做实验，都是`192.168.0.X`打头的话，那么都将得到同一台server的响应。

还有一个注意点：

> If one of the servers needs to be temporarily removed, it should be marked with the down parameter in order to preserve the current hashing of client IP addresses.

不能把后台服务器直接移除，只能标记 down ，配置参考如下：

```
upstream tomcats {
    ip_hash;
    server 192.168.1.173:8080;
    server 192.168.1.174:8080 down;
    server 192.168.1.175:8080;
}
```

5.3、least_conn

最少连接法：每次都选择连接数最少的server，如果有多个server的连接数相同，再根据这几个server的权重分发请求。配置参考如下：

```
upstream backend {
    least_conn;

    server backend1.example.com;
    server backend2.example.com;
}
```

5.4、其他方式

我们介绍了轮询、权重、ip_hash三种方式了，这个是`Nginx`本身支持的算法，也是咱们用的最多的配置方式。当然了，世界是多样性的，通过引入第三方模块可以进行拓展，比如`fair`和 `url_hash`等算法。我们这里简单介绍下即可，结合上面介绍的，我相信也不需要做演示也可以理解。

`fair`：用于为响应时间分配服务器组内的服务器，他是按后端服务器的响应时间来分配请求，**响应时间越短的越优先分配**，需要第三方模块的支持`nginx-upstream-fair-master`.

`url_hash`：**按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器**，后端服务器为缓存时比较有效。注意：**使用hash后不能使用weight**。需要第三方模块的支持`ngx_http_upstream_hash_module`

配置参考如下：

```
upstream tomcats {
    # url hash
    hash $request_uri;
    server 192.168.1.173:8080;
    server 192.168.1.174:8080;
    server 192.168.1.175:8080;
}
```

不过普通的hash会带来一个问题：当集群结点增加或减少时，对整体hash的结果都要重新计算，影响范围过大，我们可以借用一致性hash来缩小因集群扩容或缩容（节点故障等原因）带来的影响，可使用一致性hash算法。关于一致性hash算法读者朋友们不了解的也可自行学习下。我们可以通过consistent参数使用ketama一致哈希算法来减少增减服务器对客户端的影响。

```
upstream backend {
    hash $request_uri consistent;

    server backend1.example.com;
    server backend2.example.com;
}
```