亿级流量秒杀接入层架构

参考链接

系统架构知识图谱（一张价值10w的系统架构知识图谱）

https://www.processon.com/view/link/60fb9421637689719d246739

秒杀系统的架构

https://www.processon.com/view/link/61148c2b1e08536191d8f92f

分层的高并发策略

常见互联网分布式架构如上，分为：

（1）**客户端层：**客户端层是浏览器browser或者手机应用APP；

（2）**接入层**：系统入口，反向代理；

（3）**服务层**：实现核心应用逻辑，返回html或者json；

（4）**服务层**：如果实现了服务化，就有这一层；

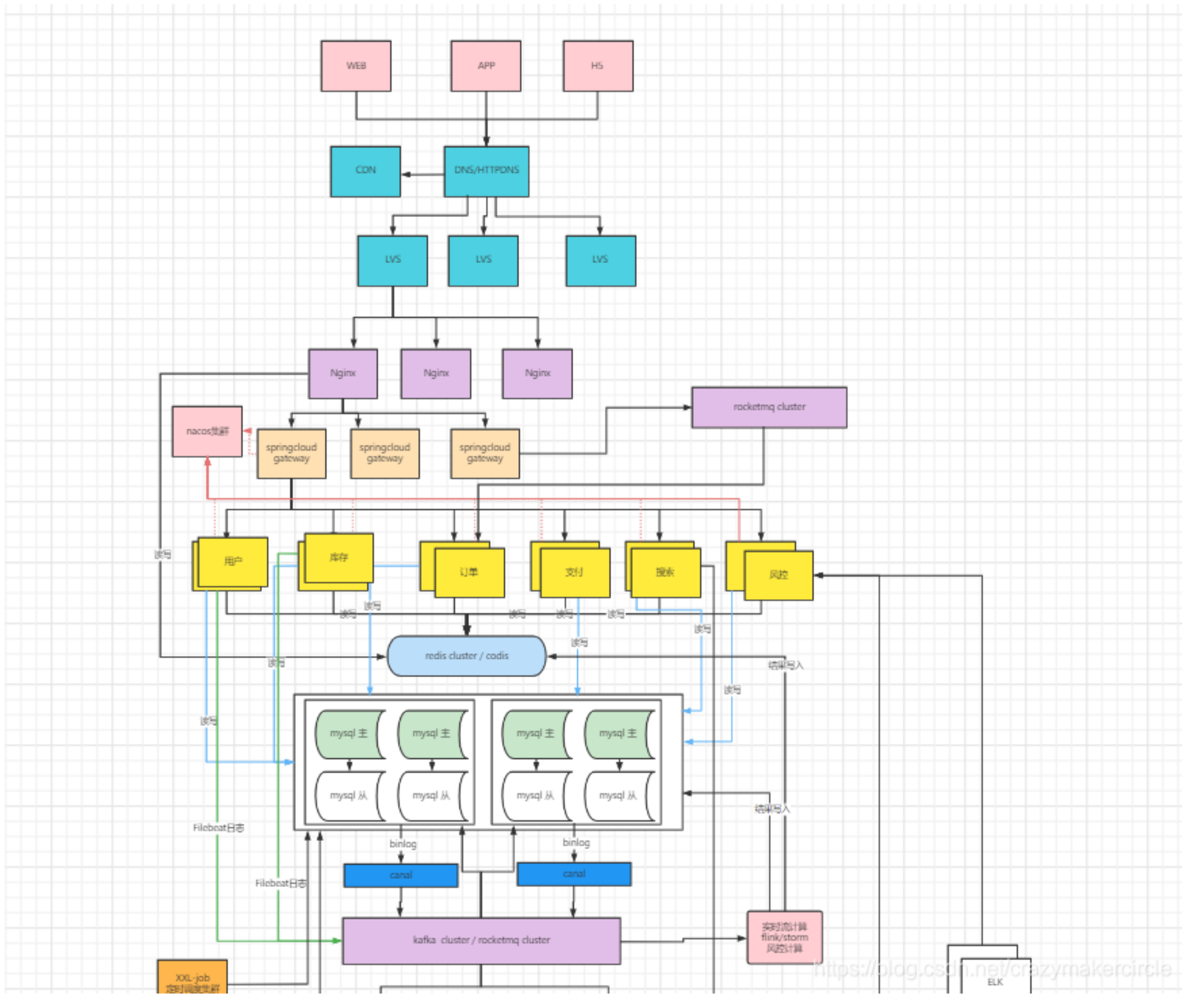
（5）**缓存层**：缓存加速访问存储；

（6）**数据库层**：数据库固化数据存储；

（7）**中间件：** zk、xxl-job，rocketmq

架构图：

https://www.processon.com/view/link/61148c2b1e08536191d8f92f



接入层SLB 负载均衡架构

根据流量而定：

》10W qps

三级以上slb： gslb（智能dns）-》 lvs-》ng-》SpringCloud gateway

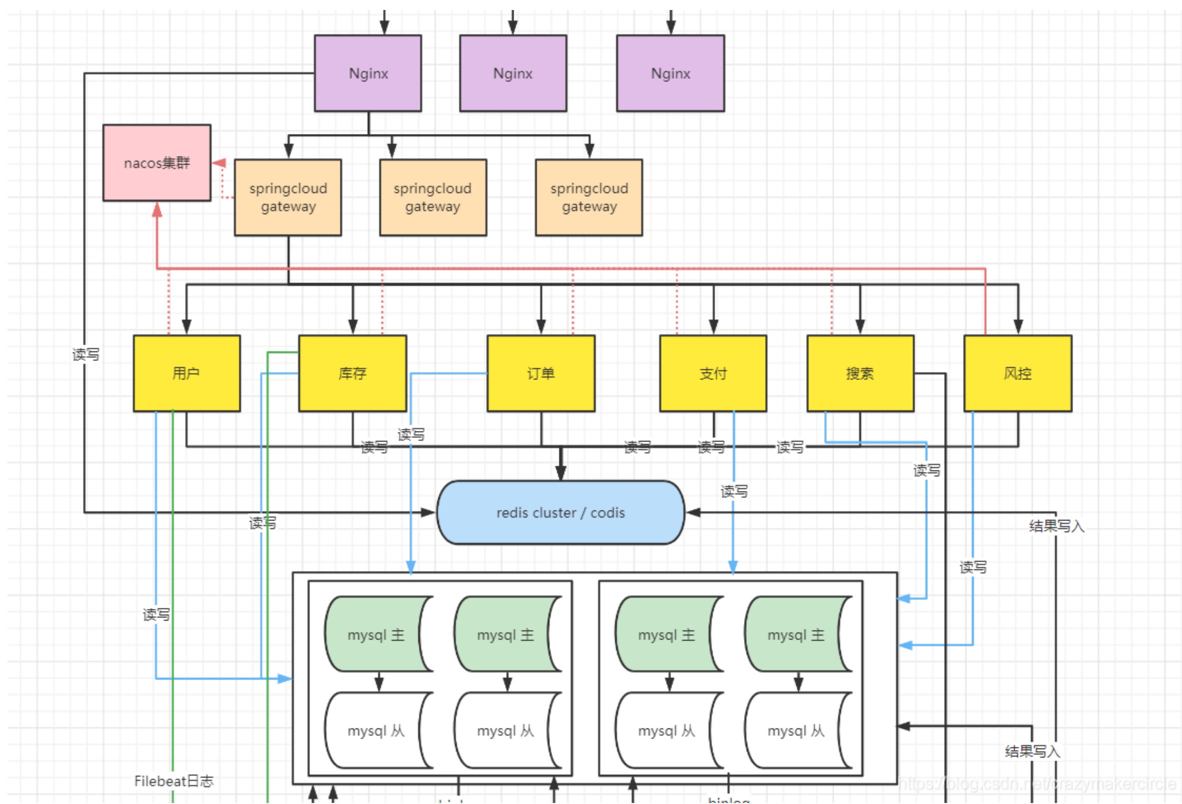
》5W qps

两级以上slb： lvs-》ng-》SpringCloud gateway

<= 5W qps

一级slb： ng- 》SpringCloud gateway

Nginx到服务层网关的负载均衡



nignx的基础知识：

centos 下ng的安装

https://www.cnblogs.com/crazymakercircle/p/12115651.html

nignx的基础知识：

《SpringCloud、Nginx高并发核心编程》

微服务网关的高并发与负载均衡

> 微服务网关与接入层网关(ng)的关系

站点层的水平扩展，是通过“nginx”实现的。通过修改nginx.conf，可以设置多个SpringCloud gateway 网关服务。

当SpringCloud gateway 后端成为瓶颈的时候，只要增加服务器数量 5k-1w，5W 新增SpringCloud gateway 服务的部署，在nginx配置中配置上新的web后端，就能扩展站点层的性能，做到理论上的无限高并发。

上游服务网关 的负载均衡

```
#内部网关的代理，内部网关带有 token 认证
```

```
  upstream zuul {
    # idea 开发环境
    #    server 192.168.56.121:7799;
    # centos 自验证环境
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";
    server "192.168.56.121:8888";

    keepalive 1000;
  }
```

在location中使用upstream负载均衡节点

```
    # 开发调试： 库存服务
    location  ^~ /stock-provider/ {
      proxy_pass http://zuul/stock-provider/ ;
           proxy_set_header Connection "";
        proxy_http_version 1.1;
        proxy_ignore_client_abort on;
        #下面的timeout跟自己的业务相关设置对应的timeout
        proxy_connect_timeout 600;
        proxy_read_timeout 600;
        proxy_send_timeout 600;
    }
```

测试环境的启动ng:

```
[root@cdh1 ~]# /vagrant/LuaDemoProject/sh/linux/openresty-restart.sh
shell dir is: /vagrant/LuaDemoProject/sh/linux
openrestry/nginx is not running!
OPENRESTRY_PATH:/usr/local/openresty
PROJECT_PATH:/vagrant/LuaDemoProject/src
nginx: [alert] lua_code_cache is off; this will hurt performance in
/vagrant/LuaDemoProject/src/conf/nginx-seckill.conf:90
openrestry/nginx starting succeeded!
pid is 29219
```

原始的

http://cdh1:7711/stock-provider/swagger-ui.html

微服务网关

http://cdh1:8888/stock-provider/swagger-ui.html

反向代理

http://cdh1:8080/stock-provider/swagger-ui.html

不同的上游节点之间的负载均衡策略：

《SpringCloud、Nginx高并发核心编程》

上游连接池的设置

Nginx Upstream长连接由upstream模式下的keepalive指令控制，并指定可用于长连接的连接数，配置样例如下：

```
#内部网关的代理，内部网关带有 token 认证
upstream zuul {
  # idea 开发环境
  #    server 192.168.56.121:7799;
  # centos 自验证环境
  server "192.168.56.121:8888";
  keepalive 10000;
}
```

一旦与后端服务器建立连接，则在当前请求连接结束之后不会立即关闭连接，而是把用完的连接保存在一个keepalive connection pool里面，以后每次需要建立向后连接的时候，只需要从这个连接池里面找，如果找到合适的连接的话，就可以直接来用这个连接，不需要重新创建socket或者发起connect()。

> 这样既省下建立连接时在握手的时间消耗，又可以避免TCP连接的slow start。

如果在keepalive连接池找不到合适的连接，那就按照原来的步骤重新建立连接。假设连接查找时间可以忽略不计，那么这种方法肯定是有益而无害的（当然，需要少量额外的内存）。

为支持长连接，需要配置使用HTTP1.1协议（虽然HTTP 1.0可通过设置Connection请求头为“keep-alive”来实现长连接，但这并不推荐）。

此外，由于HTTPPROXY模块默认会将反向代理请求的connection头部设置成Close，因此这里也需要清除connection头部（清除头部即不发送该头部，在HTTP 1.0中默认为长连接）。

```
location  ^~ /seckill-provider/ {
        proxy_pass http://zuul/seckill-provider/;
         proxy_set_header Connection "";
        proxy_http_version 1.1;
        proxy_ignore_client_abort on;
        #下面的timeout跟自己的业务相关设置对应的timeout
        proxy_connect_timeout 600;
        proxy_read_timeout 600;
        proxy_send_timeout 600;
}
```

接入层静态内容服务：

“秒杀开始前几分钟，大量用户开始进入秒杀商品详情页面，很多人开始频繁刷新秒杀商品详情页，这时秒杀商品详情页访问量会猛增”。如果请求全部打到后端服务，那后端服务的压力会非常大（后端服务要处理业务逻辑，而且还要访问数据库，吞吐量比较低）。

秒杀的前端工程

客户端需要完成秒杀商品的静态化展示。无论在桌面浏览器、移动端App展示秒杀商品，秒杀商品的图片和文字元素，需要尽可能静态化，尽量减少动态元素。这样，就可以通过CDN来提速和抗峰值。

另外，在客户端这一层的用户交互上需要具备一定的控制用户行为和禁止重复秒杀的能力。比如，当用户提交秒杀请求之后，可以将秒杀按钮置灰，禁止重复提交。

简化的UI设计

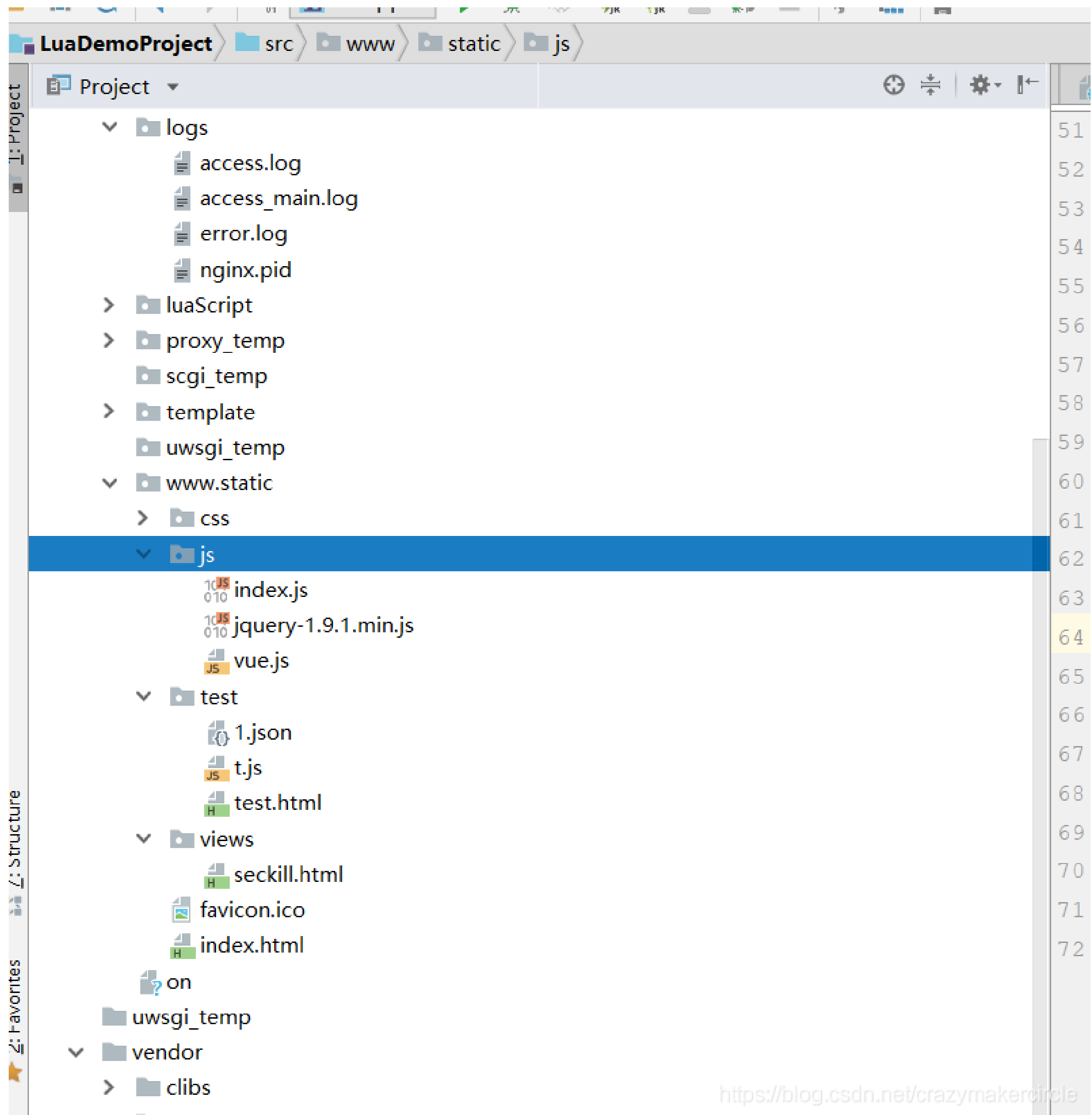
分布式秒杀测试 LUA 版本

请填写秒杀信息

用户ID 1 | 1 设置用户
用户信息 点击右边设置用户,获取用户
秒杀ID 1157197244718385152 | 2 获取商品详细
商品名称 秒杀商品-1
库存数量 10001
原始库存 10001
暴露地址 4b70903f6e1aa87788d3ea962f8b2f0e
秒杀令牌 请获取秒杀令牌 | Lua获取令牌
秒杀结果 | 4 开始秒杀

前端工程的代码结构

> nginx下的静态内容工程: js+html的形式。html负责静态内容的渲染，js负责数据的动态渲染



前端代码的访问配置

```
    location ~ .*\.(htm|html)$ {        # 自动匹配到(htm|html)格式
      ## 开发阶段，配置页面不缓存html和htm结尾的文件
      add_header Cache-Control "private, no-store, no-cache, must-revalidate,
proxy-revalidate";
      root /vagrant/LuaDemoProject/src/www/static; #服务器路径
      default_type 'text/html';
    }


    location ~ .*\.(js|script)$ {        # 自动匹配到(jpg|gif|png)格式
      root /vagrant/LuaDemoProject/src/www/static; #服务器路径
      default_type 'application/javascript';
    }
```

实验地址

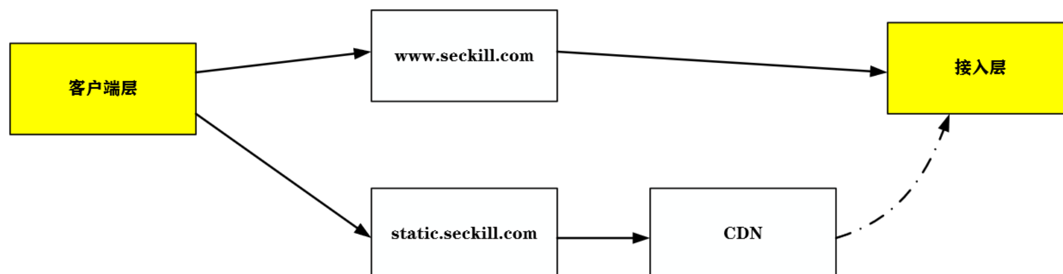
http://cdh1:8080/

CDN加速

CDN相当于加上一层缓存， 加载离用户最近的idc机房， 由cdn的运营商提供， 比如电信等

https://www.cnblogs.com/crazymakercircle/p/14978513.html

秒杀的静态页面通过到CDN上预热（CDN是内容分发网络，可以简单理解成互联网上的巨大的缓存，用于存放静态页面、图片、视频等，可以显著提高访问速度），用CDN扛流量，这样大量的商品详情页的访问请求就不用访问自己的网站（源站）。这样既可以提高访问速度，也没有给网站增加压力，同时也减少了网站带宽压力。

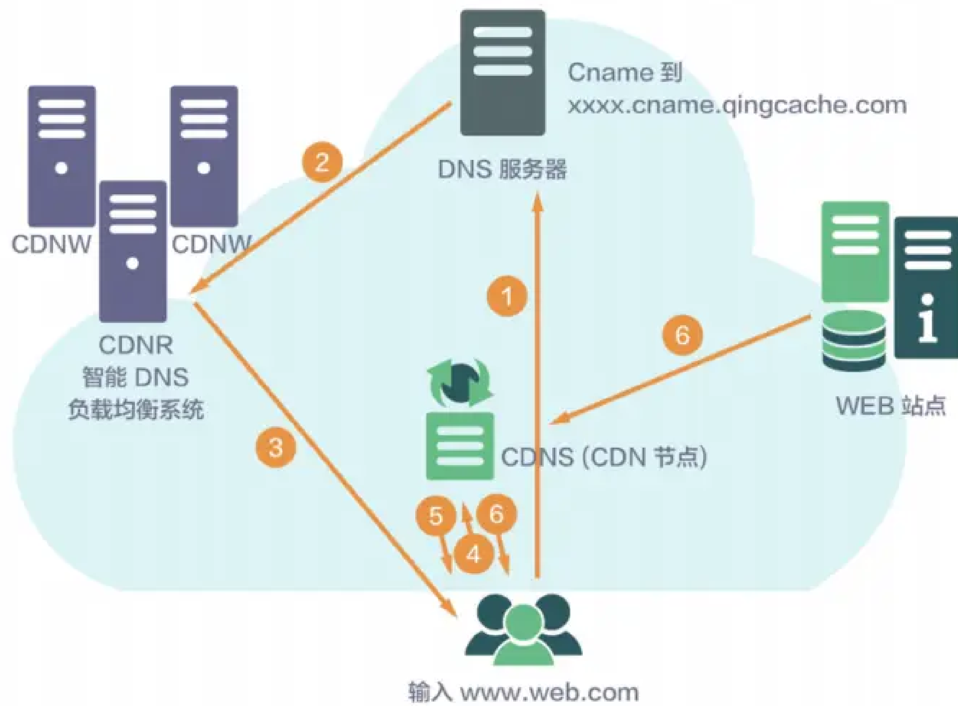


CDN加速的流程示意图

CDN是为了解决用户网络访问时的“最后一公里”效应，本质是一种“以空间换时间”的加速策略，即将内容缓存在离用户最近的地方，用户访问的是缓存的内容，而不是站点实时访问的内容。

由于CDN部署在网络运营商的机房，这些运营商又是终端用户的网络提供商，因此用户请求路由的第一跳就到达了CDN服务器，当CDN中存在浏览器请求的资源时，从CDN直接返回给浏览器，最短路径返回响应，加快用户访问速度。

下面是简单的CDN请求流程示意图：



CDN能够缓存的一般是静态资源，如图片、文件、CSS、Script脚本、静态网页等，但是这些文件访问频度很高，将其缓存在CDN可极大改善网页的打开速度。

CDN准备

秒杀前要和网络运营商、CDN服务商提前申请带宽。

阿里云cdn加速操作实战

记录一下备忘：

- 加速域名和源站域名

源站域名就是你的网站**原本**对外访问静态资源的域名，比如

- 网站主域名www.seckil.com，
- 静态文件域名 static.seckil.com，

加速域名就是你想用CDN来代替你原来静态文件域名的新域名，比如随便起一个 cdn-static.seckil.com，

1填 cdn-static.seckil.com, 2填 static.seckil.com

注意，要想起到cdh效果，访问的静态的地址变成 cdn-static.seckil.com，

> 说明：以上cdn加速的例子，仅仅为示意，参考作用。

接入层使用本地缓存大大提高读并发的吞吐量

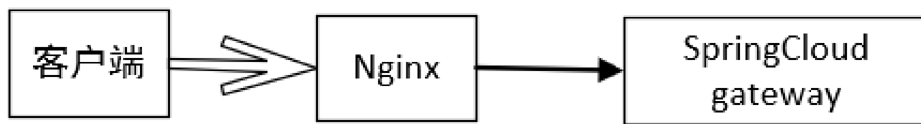
或者：

如何10倍_100倍的提升读并发的吞吐量

问题：提升秒杀商品详细信息的读并发

http://cdh1:8080/stock-provider/swagger-ui.html

```
http://zuul/stock-provider/api/seckill/sku/detail/v1
```



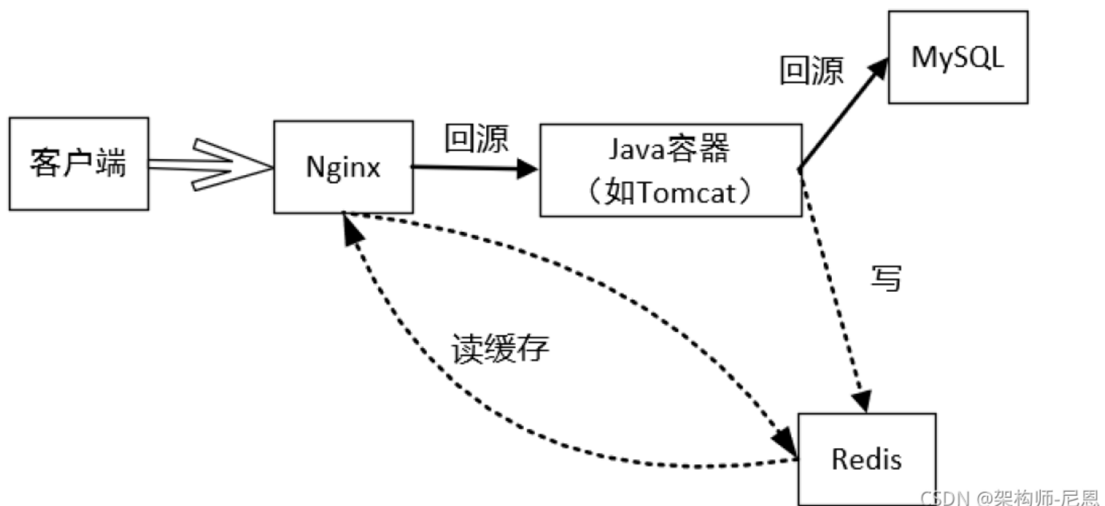
CSDN @架构师-尼恩

如何提升读并发的吞吐量?

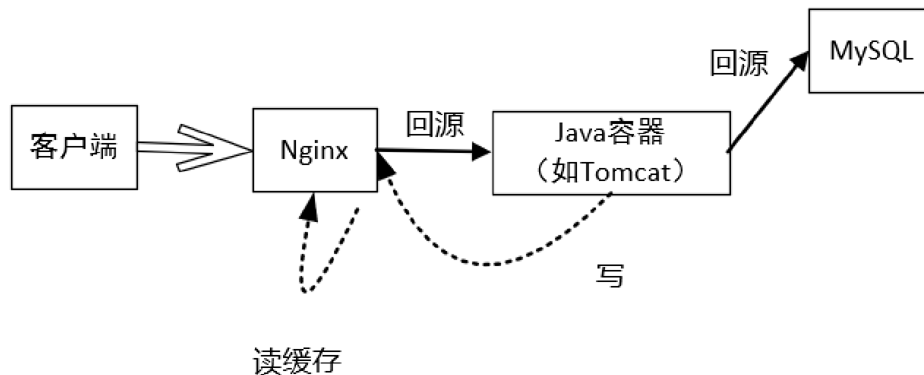
如何提升读并发的吞吐量？一个提高读请求吞吐量的原则：

> 尽可能的命中缓存，最好是本地缓存，如果数据量较大，可以是二级缓存

如果数据量较大，可以是二级缓存



尽可能的命中缓存，最好是本地缓存，对于数据量不大的极热数据



CSDN @架构师-尼恩

将参加秒杀的商品信息放入ng的进程缓存中，这样可以大大的提高系统的吞吐量。

实操：利用本地缓存（L3）级缓存提升秒杀商品的读并发

本地缓存的数据一致性策略

过期删除+数据回源

被动的更新策略，过了一段时间之后，缓存会过期，然后回源到上游服务获取数据

主动更新+主动删除

通过消息队列，业务系统主动更新+删除 ng缓存的书

前置知识：

lua基础

lua操作缓存

lua操作redis

《SpringCloud、Nginx高并发核心编程》

接入层如何10倍_100倍的提升写并发的吞吐量

如何提升写并发的吞吐量?

如何提升写并发的吞吐量？一个提高写并发吞吐量的原则：

1 有效流量锁定，过滤出有效请求

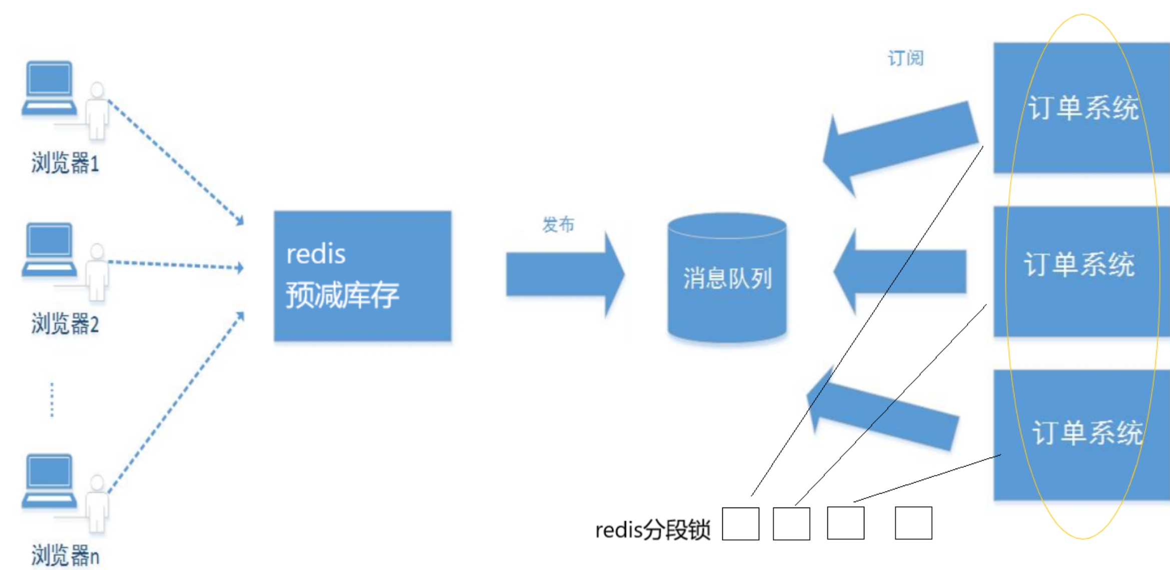
2 有效流量削峰，有效请求 达到或者超出 下一层的能力瓶颈， 同步处理，变成（降级为）异步处理

3 有效流量限流，1000Qps， 接入层、服务层、数据库

秒杀的两段式下单方案

高并发操作迁移到并发量更高的nginx+redis+lua，提交操作变成两段式：

- 第一阶段为锁定有效流量， 将有效的流量识别出来，过滤掉无效的流量， 申请令牌，申请预减减库，申请成功之后，进入服务层
- 第二阶段为有效流量落地，进入服务层之后，进入消息队列，秒杀服务从消息队列拉取令牌，确认令牌，然后完成下单操作。 查库存 -> 创建订单 -> 扣减库存。通过分布式锁保障解决多个provider实例并发下单产生的超卖问题。
-



异步的两段式下单

将提交操作变成两段式：

- 第一阶段申请令牌阶段，模拟库存的扣减操作，在redis中预减减库，预减成功，发出令牌，申请成功之后，返回客户端端；
- 第二阶段秒杀下单阶段，从消息队列消费申请令牌，然后完成下单操作。 查库存 -> 创建订单 -> 扣减库存。通过分布式锁保障解决多个provider实例并发下单产生的超卖问题。

申请令牌阶段：

将存库从MySQL前移到Redis中，所有的预减库存的操作放到内存中，由于Redis中不存在锁故不会出现互相等待，并且由于Redis的写性能和读性能都远高于MySQL，这就解决了高并发下的性能问题。

秒杀下单阶段：

然后通过队列等异步手段，将变化的数据异步写入到DB中。

引入队列，然后数据通过队列排序，按照次序更新到DB中，完全串行处理。当达到库存阀值的时候就不在消费队列，并关闭购买功能。这就解决了超卖问题。

两段式下单的优化效果：

一个高性能秒杀的场景：

> 假设一个商品6000个库存，假设10W用户1分钟之内抢完，1分钟6000订单，每秒的 600个下单操作。
>
> 申请令牌阶段，每秒的 600个预减库存的操作，对于 Redis 来说，没有任何压力。甚至每秒的6000个预减库存的操作，对于 Redis 来说，也是压力不大。
>
> 秒杀下单阶段，就不一样了。假设加锁之后，释放锁之前，查库存 -> 创建订单 -> 扣减库存，经过优化，每个IO操作100ms，大概200毫秒，一秒钟5个订单。如果不经过性能提升和优化， 600个订单需要120s，2分钟才能彻底完成。

申请令牌阶段的Lua脚本

初始化秒杀的令牌发放的结构

暴露一下，创建一个结果

```
http://cdh1:8888/swagger-ui.html

     {
     "exposedKey": "4b70903f6e1aa87788d3ea962f8b2f0e",
     "newStockNum": 10000,
     "seckillSkuId": 1157197244718385152,
     "seckillToken": "0f8459cbae1748c7b14e4cea3d991000",
     "userId": 37
     }
```

getToken_v3.lua脚本的介绍

getToken_v3.lua脚本并没有判断和设置秒杀令牌的核心逻辑

仅仅调用缓存在Redis内部的seckill.lua脚本的setToken方法设置和获取秒杀令牌

然后对seckill.lua脚本的返回值进行判断，并根据不同的返回值作出不同的响应。

> 问题：什么是seckill.lua脚本，seckill.lua脚本为啥要执行在redis

Nginx Lua与Redis Lua之间的关系

getToken_v3.lua脚本和seckill.lua脚本都是Lua脚本，但是执行的地点不同：

- getToken.lua脚本被执行在Nginx中

- seckill.lua脚本被执行在Redis中，

getToken.lua通过evalsha方法，去调用缓存在Redis中的seckill.lua脚本。getToken.lua脚本和seckill.lua脚本的关系，具体如图10-16所示。

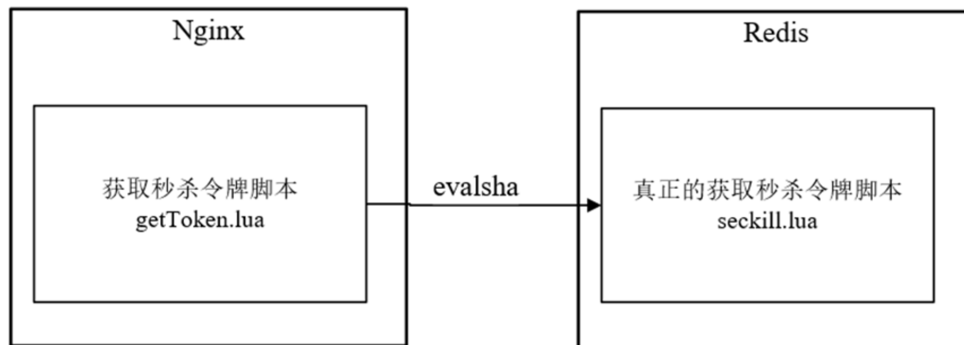


图10-16 getToken.lua脚本和seckill.lua脚本关系

什么时候在Redis中加载seckill.lua脚本呢?

和限流脚本一样，该脚本是在Java程序启动商品秒杀时完成其在Redis的加载和缓存的。

并且，Java程序会将seckill.lua脚本加载完成之后的sha1编码，去通过自定义的key（具体为"lua:sha1:seckill"）缓存在Redis中，以方便Nginx中的getToken.lua脚本去获取，并且在调用evalsha方法时使用。

什么是sha1编码呢?

Redis在缓存完Lua脚本后，会返回该脚本的固定长度的sha1编码，作为Lua脚本的摘要提供给外部调用Lua脚本使用。

sha1摘要是通过SHA-1（Secure Hash Algorithm 1、安全散列算法1）散列算法生成的。SHA-1算法是第一代"安全散列算法"的缩写，其本质就是一个Hash算法，主要用于生成字符串摘要（摘要经加密后成为数字签名），该算法曾被认为是MD5算法的后继者。

SHA-1算法能将一个最大2^64比特的字符串散列成一串160位（20字节）的散列值，散列值通常的呈现形式为40个十六进制数。SHA-1算法始终能保证任何两组不同的字符串产生的摘要是不同的。

> 注意：使用redis集群，因此每个节点都需要各自load一份脚本数据

```
/**
* 由于使用redis集群，因此每个节点都需要各自缓存一份脚本数据
* @param slotKey 用来定位对应的slot的slotKey
*/
public void storeScript(String slotKey){
if (StringUtils.isEmpty(unlockSha1) || !jedisCluster.scriptExists(unlockSha1,
slotKey)){
   //redis支持脚本缓存，返回哈希码，后续可以继续用来调用脚本
    unlockSha1 = jedisCluster.scriptLoad(DISTRIBUTE_LOCK_SCRIPT_UNLOCK_VAL,
slotKey);
   }
}
```

getToken_v3.lua脚本的使用演示

高并发系统的限流架构

限流的概念：
限流的目的是在大促或者流量突增期间，我们的后端服务假设某个接口能够扛住的的QPS为10000，这时候同时有20000个请求进来，经过限流模块，会先放10000个请求，其余的请求会阻塞一段时间。

不简单粗暴的返回404，让客户端重试，同时又能起到流量销峰的作用。

限流是面试中的常见的面试题（尤其是大厂面试、高P面试）

限流的总体架构

前面提到，秒杀系统中秒杀商品总是有限的。除此之外，服务节点处理能力、数据库的处理能力也都是有限的，因此，需要根据系统的负载能力进行秒杀限流。

两个级别的限流策略

总体来说，在接入层可以进行两个级别的限流策略：

（1）应用级别的限流

（2）接口级别的限流

什么是应用级别的限流策略呢？对于整个应用系统来说，一定会有一个QPS的极限值，如果超了极限值，则整个应用就会不响应或响应的非常慢。因此，需要在整个应用的维度，做好应用级别的限流配置。

应用级别的限流，应该配置在最顶层的反向代理，具体如图10-4所示。

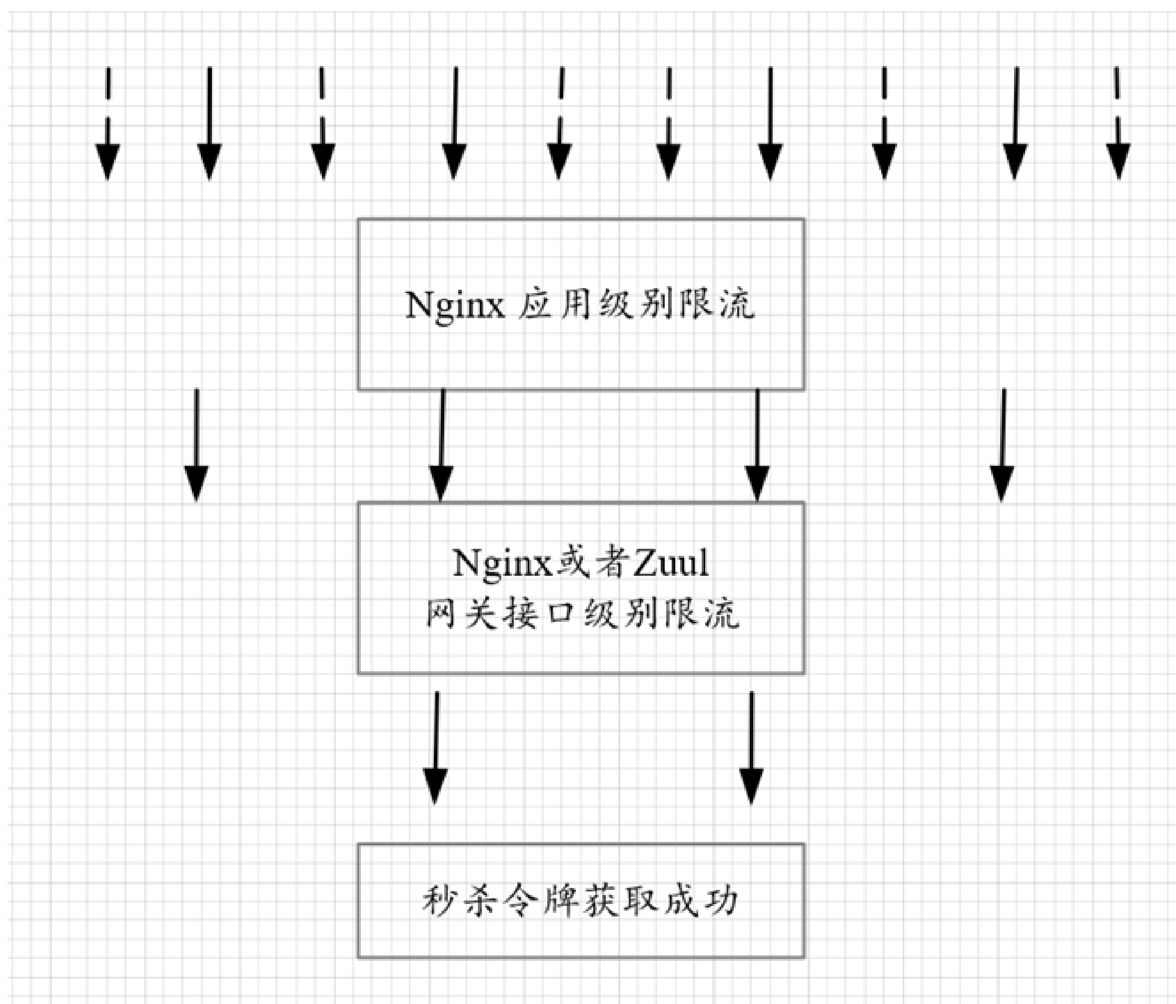


图10-4 接入层的限流架构

应用级别的流量限制

应用级别的流量限制，可以通过Nginx的limit_req_zone和limit_req两个指令完成。假定要配置Nginx虚拟主机的限流规则为单IP限制为每秒1次请求，整个应用限制为每秒10次请求，则具体的配置如下：

limit_req 应用在server块

```
limit_req_zone  $binary_remote_addr  zone=perip:10m      rate=1r/s;
limit_req_zone  $server_name        zone=perserver:1m   rate=10r/s;

server {
    ...
    limit_req  zone=perip  burst=5;
    limit_req  zone=perserver  burst=10;
}
```

接口级别的限流策略

什么是接口级别的限流策略呢?

如果单个接口可能会有突发访问情况，可能会由于突发访问量太大造成系统崩溃，典型的就是本章所介绍的秒杀类接口。接口级别的限流就是在配置单个接口的请求速率，是细粒度的限流。

接口级别的限流

- Nginx的limit_req_zone和limit_req两个指令配合完成

接口级别的限流也可以通过Nginx的limit_req_zone和limit_req两个指令配合完成，对获取秒杀令牌的接口，同时进行用户Id和商品Id进行限流的配置大致如下:

limit_req 应用在location 块

- 也可以使用分布式限流组件，例如redis+Lua来限流。

秒杀的接口基本限流架构

应用维度的限流:

单ng限制在50000QPS每秒

对用户的限流:

用户维度的限流，可以在ngix 上进行，因为:

- 使用nginx限流内存来存储用户id，比用redis 的key，来存储用户id，效率高。
- 大流量往往都需要做流量分片，在进行大流量分片路由的时候，同一个用户id， 我们尽量路由到同一个接入网关

对商品的限流：

商品维度的限流，可以在redis上进行，因为：

- 不需要大量的计算访问次数的key，
- 不同的接入层网关，需要通过redis，发出同一个商品的令牌

初始化的限流配置

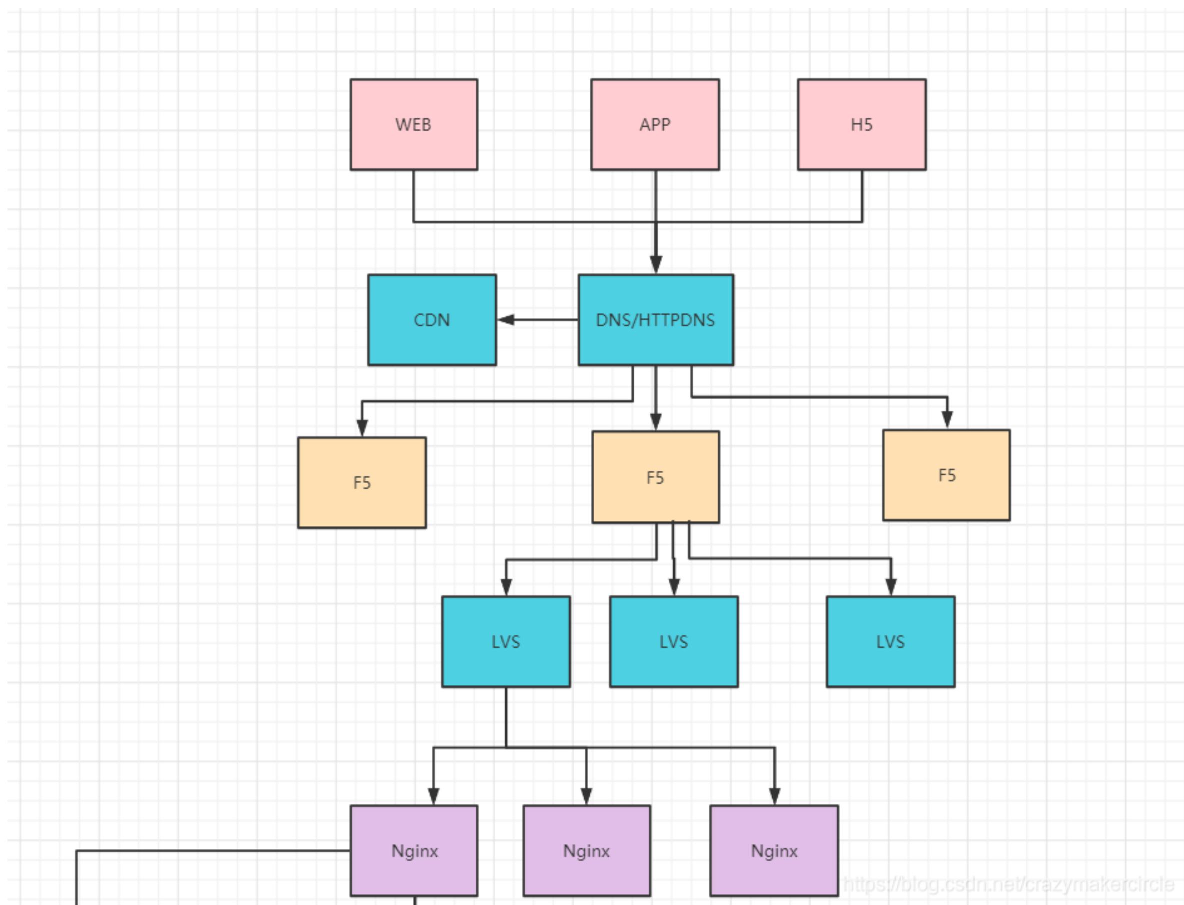
```
# Nginx+lua 秒杀：获取秒杀 token
location  ~ /seckill-lua/(.*)/getToken/v3 {
  default_type 'application/json';
  charset utf-8;
  set $skuId $1;
  limit_req  zone=userzone;

  # 限流的 lua 脚本
  access_by_lua_file luaScript/module/seckill/getToken_access_limit.lua;
  # 获取秒杀token lua 脚本
  content_by_lua_file luaScript/module/seckill/getToken_v3.lua;
}
```

初始化秒杀的限流器

```
        //初始化秒杀的限流器
        rateLimitService.initLimitKey(
                "seckill",
                String.valueOf(skuId),
                10000000,//总数 SeckillConstants.MAX_ENTER,
                1000// 1000/s  SeckillConstants.PER_SECKOND_ENTER
        );
```

秒杀暴露的时候，初始化了这个结构

LVS到Nginx负载均衡



LVS的作用

一、LVS主要用于多服务器的负载均衡。

二、它工作在网络层，可以实现高性能，高可用的服务器集群技术。

三、它可把许多低性能的服务器组合在一起形成一个超级服务器。

四、它配置非常简单，且有多种负载均衡的方法。

五、它稳定可靠，即使在集群的服务器中某台服务器无法正常工作，也不影响整体效果。

六、可扩展性也非常好。

LVS和Nginx的配合使用

一、nginx工作在网络的应用层，主要做反向代理；lvs工作在网络层，主要做负载均衡。nginx也同样能承受很高负载且稳定，但负载度和稳定度不及lvs。

二、nginx对网络的依赖较小，lvs就比较依赖于网络环境。

三、在使用上，一般最前端所采取的策略应是lvs。 nginx可作为lvs节点机器使用。

鉴权

所有请求需要鉴权，校验合法身份

- 可以结合jwt 令牌进行用户鉴权

接入层安全

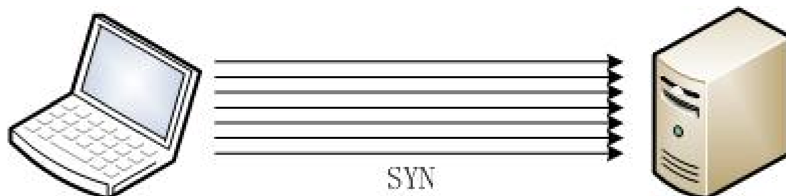
相信不少站长或多或少都经历过DDoS攻击（DDoS攻击是通过大量合法的请求占用大量网络资源，以达到瘫痪网络的目的），一旦你的网站被人DDoS攻击后，网站就会无法被访问了，严重时服务器都能卡死

DoS攻击与DDoS攻击

拒绝服务攻击DoS(Denial of Service)：使系统过于忙碌而不能执行有用的业务并且占尽关键系统资源。它是基于这样的思想：用数据包淹没本地系统，以打扰或严重阻止捆绑本地的服务响应外来合法的请求，甚至使本地系统崩溃。实现Dos攻击，常见的方式有：TCP SYN泛洪(SYN Flood)，ping泛洪(ping-Flood)，UDP泛洪(UDP-Flood)，分片炸弹(fragmentation bombs)，缓冲区溢出(buffer overflow)和ICMP路由重定向炸弹(ICMP routeing redirect bomb)。

拒绝服务攻击DoS(Denial of Service)：使系统过于忙碌而不能执行有用的业务并且占尽关键系统资源。它是基于这样的思想：用数据包淹没本地系统，以打扰或严重阻止捆绑本地的服务响应外来合法的请求，甚至使本地系统崩溃。实现Dos攻击，常见的方式有：**TCP SYN泛洪**(SYN Flood)，**ping泛洪**(ping-Flood)，**UDP泛洪**(UDP-Flood)，**分片炸弹**(fragmentation bombs)，**缓冲区溢出**(buffer overflow)和**ICMP路由重定向炸弹**(ICMP routeing redirect bomb)。

1、TCP SYN泛洪
TCP SYN泛洪是利用TCP建立连接时需要进行三次握手的过程，并结合**IP源地址欺骗**实现的。如下图：



攻击者将其自身的源地址伪装成一个私有地址向本地系统的TCP服务发起连接请求，本地TCP服务回复一个**SYN-ACK**作为响应，然而该响应发往的地址并非攻击者的地址(真实地址)，而是攻击者伪装的私有地址。由于该私有地址是不存在的于本地服务器所在的网络的，所有本地系统将收不到**RST**消息(以结束这个半打开连接)。

本地TCP服务接下来要等待接收一个ACK回应，但是该回应永远不会到来，该半打开连接会保持打开状态直至连接尝试超时，因此有限的连接资源被消耗了。攻击者连接请求的到来比TCP超时释放资源更快，利用一次又一次的连接请求淹没本地连接资源(如通过listen()创建的大小有限的连接队列等)，以致本地服务无法接收更多的连接请求。

SYN攻击属于DOS攻击的一种，它利用TCP协议缺陷，通过发送大量的半连接请求，耗费CPU和内存资源。TCP协议建立连接的时候需要双方相互确认信息,来防止连接被伪造和精确控制整个数据传输过程数据完整有效。所以TCP协议采用三次握手建立一个连接。

第一次握手：建立连接时，客户端发送syn包到服务器，并进入SYN_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN 同时自己也发送一个SYN包 即SYN+ACK包，此时服务器进入SYN_RECV状态；

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

问题是：

假设一个用户向服务器发送了SYN报文后突然死机或掉线，那么服务器在发出SYN+ACK应答报文后是无法收到客户端的ACK报文的（第三次握手无法完成），

这种情况下，服务器端一般会重试（再次发送SYN+ACK给客户端）并等待一段时间后丢弃这个未完成的连接，这段时间的长度称为SYN Timeout，一般来说这个时间是分钟的数量级（大约为30秒-2分钟）；一个用户出现异常导致服务器的一个线程等待1分钟并不是什么很大的问题，但如果有一个恶意的攻击者大量模拟这种情况，服务器端将为了维护一个非常大的半连接列表而消耗非常多的资源----数以万计的半连接，

即使是简单的保存并遍历也会消耗非常多的CPU时间和内存，何况还要不断对这个列表中的IP进行SYN+ACK的重试。实际上如果服务器的TCP/IP栈不够强大，最后的结果往往是堆栈溢出崩溃---即使服务器端的系统足够强大，服务器端也将忙于处理攻击者伪造的TCP连接请求而无暇理睬客户的正常请求（毕竟客户端的正常请求比率非常之小），此时从正常客户的角度看来，服务器失去响应，这种情况称做：服务器端受到了SYN Flood攻击（SYN洪水攻击）

2、ping泛洪
攻击者通过ping发送的ICMP的echo请求消息也是常见的DoS攻击方式之一，其原理是**强制让系统消耗大多数时间进行无用的应答，降低系统网络质量**。主要实现的方法有：①将ping包的源地址伪装成受害者的地址并向整个主机所在的网络广播echo请求，这样的请求消息能够造成很多的响应发送的受害者机器；②通过互联网在受害者机器安装木马程序并在某时刻向某主机发送大量echo请求；③攻击者发送更多简单的ping泛洪来淹没数据连接。

一种更为古老的攻击方式叫做死亡之ping，攻击者会发送巨大的ping数据包给受害机器，易受攻击的系统可能因此崩溃，Linux等类UNIX系统并没有此漏洞。对于受害主机来说，丢弃ping请求并不是一个很好的解决方案，因为不论是对到来的ping包做什么反应，系统或者网络依旧会被淹没在检测/丢弃数据包的过程中。

3、UDP泛洪
不同于TCP，UDP是**无状态**的，没有任何信息被维护以指明下一个期望到来的数据包，所以UDP服务更易受这些类型攻击的影响，许多站点都禁止用所有非必要的UDP端口。

4、缓冲区溢出
缓冲区溢出攻击是无法通过过滤防火墙进行保护的，通过覆盖程序的数据空间或者运行时的堆栈导致系统或者服务崩溃，这需要专业技术以及对硬件、系统软件的了解。举个例子，早期有服务器程序使用了sprintf()函数而被攻击者攻击，因为缓冲区溢出而使得程序崩溃，所以更为妥当的做法是使用**snprintf**()代替sprintf()。

5、ICMP路由重定向炸弹
我们知道**ICMP的消息类型中的类型5是告知目标系统改变内存中的路由表以获得更短的路由**，以通知主机有更多的路径可用。重定向很少发源主机附近的路由器，对于连接到ISP(运营商)的住宅或商用站点来说，主机附近的路由器产生一个重定向的ICMP消息可能性非常小。如果我们主机使用静态路由且收

到了重定向消息，这可能是有人在攻击我们系统，欺骗主机以转发所有流量到另一远处主机处。

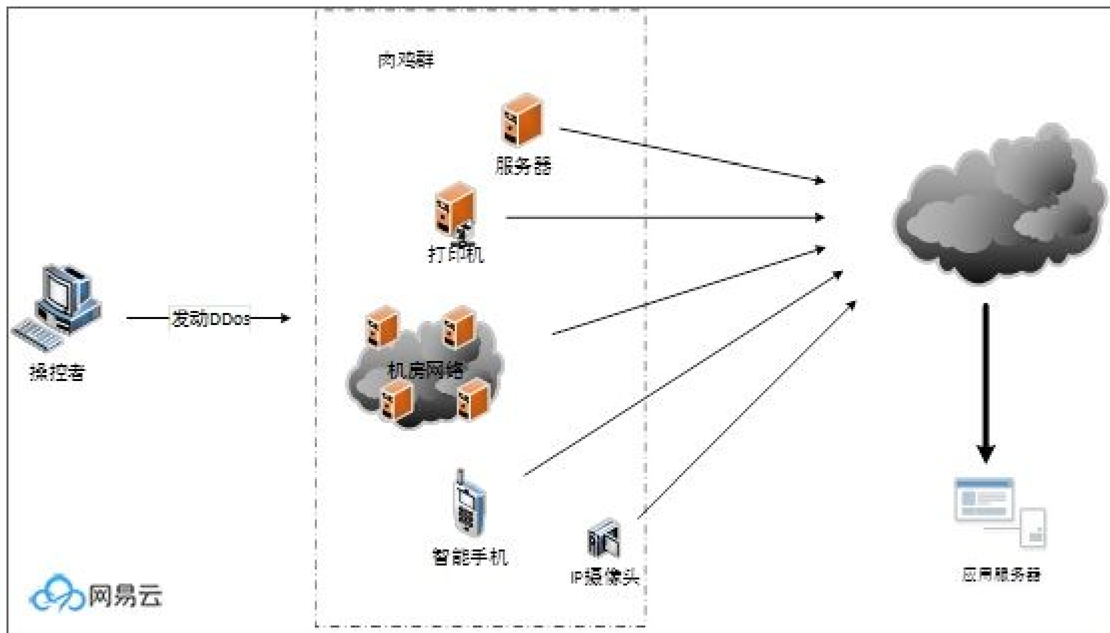
6、分片炸弹
我们知道数据包从一个路由器沿路径(源计算机到目的计算机的路径)到下一个路由器时，网关路由器可能需要在它们进行传递到下一个网络前将数据包分切为更小的片段(超过**MTU值**会被分片)，在这些片段里第一个分片会包含UDP或者TCP报头中的源端口号和目的端口号，接下来的分片并不包含。当数据包被分片时中间路由器不会重组数据包，数据包到达目的主机或邻近路由器时才会重新组装。

分片炸弹的实现是构造一种非常小的数据包导致系统或者程序崩溃的操作：比如构造一个最初的分片使得UDP或者TCP的源端口和目的端口被包含在第二个分片中。许多防火墙并不检查第一个分片之后的分片，然而第一个分片由于防火墙所要过滤的信息还未呈现，所以得以通过，那么最终所有分片都在主机中得以组装。

另外，因为中间进行分片基本上比发送更小的无需分片的数据包代价更高，所以一般会在发送在IP报头中设置了**不分片标志**，设置后系统会向目标主机发起连接前进行MTU发现，如果中间路由器必须对数据包进行分片那么它会丢弃数据包并赶回**ICMP 3错误消息**，即“需要分片”。

DDoS 攻击

全称Distributed Denial of Service，中文意思为“分布式拒绝服务”，就是利用大量合法的分布式服务器对目标发送请求，从而导致正常合法用户无法获得服务。通俗点讲就是利用网络节点资源如：IDC服务器、个人PC、手机、智能设备、打印机、摄像头等对目标发起大量攻击请求，从而导致服务器拥塞而无法对外提供正常服务，只能宣布game over，详细描述如下图所示：



DDoS的攻击方式

一种服务需要面向大众就需要提供用户访问接口，这些接口恰恰就给了黑客有可乘之机，如：可以利用TCP/IP协议握手缺陷消耗服务端的链接资源，可以利用UDP协议无状态的机制伪造大量的UDP数据包阻塞通信信道……可以说，互联网的世界自诞生之日起就不缺乏被DDoS利用的攻击点，从TCP/IP协议机制到CC、DNS、NTP反射类攻击，更有甚者利用各种应用漏洞发起更高级更精确的攻击。

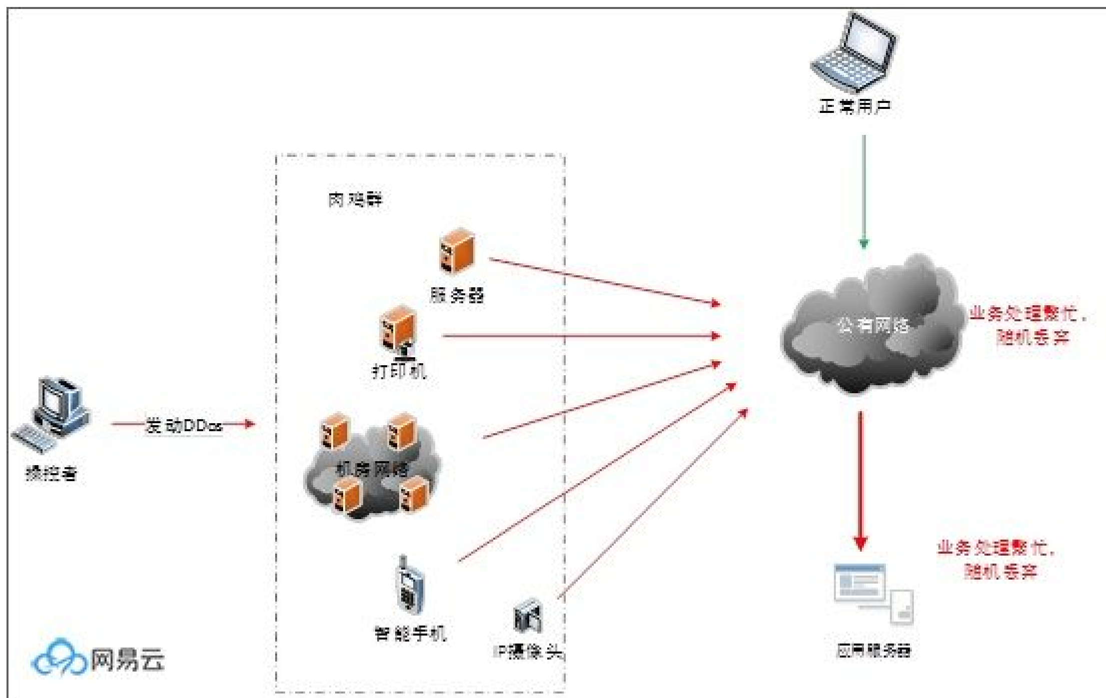
从DDoS的危害性和攻击行为来看，我们可以将DDoS攻击方式分为以下几类：

a）资源消耗类攻击

资源消耗类是比较典型的DDoS攻击，最具代表性的包括：Syn Flood、Ack Flood、UDP Flood。这类攻击的目标很简单，就是通过大量请求消耗正常的带宽和协议栈处理资源的能力，从而达到服务端无法正常工作的目的。

b）服务消耗性攻击

相比资源消耗类攻击，服务消耗类攻击不需要太大的流量，它主要是针对服务的特点进行精确定点打击，如web的CC，数据服务的检索，文件服务的下载等。这类攻击往往不是为了拥塞流量通道或协议处理通道，它们是让服务端始终处理高消耗型的业务的忙碌状态，进而无法对正常业务进行响应，详细示意图如下：



黑客为什么选择DDoS

不同于其他恶意篡改数据或劫持类攻击，DDoS简单粗暴，可以达到直接摧毁目标的目的。另外，相对其他攻击手段DDoS的技术要求和发动攻击的成本很低，只需要购买部分服务器权限或控制一批肉鸡即可，而且攻击相应速度很快，攻击效果可视。另一方面，DDoS具有攻击易防守难的特征，服务提供商为了保证正常客户的需求需要耗费大量的资源才能和攻击发起方进行对抗。这些特点使得DDoS成为黑客们手中的一把很好使的利剑，而且所向霹雳。

从另一个方面看，DDoS虽然可以侵蚀带宽或资源，迫使服务中断，但这远远不是黑客的正真目的。所谓没有买卖就没有杀害，DDoS只是黑客手中的一枚核武器，他们的目的要么是敲诈勒索、要么是商业竞争、要么是要表达政治立场。在这种黑色利益的驱使下，越来越多的人参与到这个行业并对攻击手段进行改进升级，致使DDoS在互联网行业愈演愈烈，并成为全球范围内无法攻克的一个顽疾。

接入层的安全策略

用防火墙防御DDoS

如果网站遇到DDoS攻击时，该如何解决呢？相信大家都知道，要靠防火墙（防火墙是位于内网和外网之间的屏障隔离技术）来处理一部分攻击流量。 这是不是就说明防火墙可以防御DDoS呢？其实没有这么简单。

首先我要说的是，不是有所防火墙都可以有效抵御DDoS攻击。DDoS攻击和其它攻击不同，它本身也算是一种合法的网络请求！

防火墙种类很多，主要有：软件防火墙（软防）、硬件防火墙。这两类防火墙在对待DDoS时的表现也不同。

1、软件防火墙（软防）

我们一般用户都接触过软件防火墙，像Windows上的"防火墙"、Linux上的"iptables"等。它们以软件的形式时刻检查系统上的所有数据包，然后决定放行哪些数据包或者拦截哪些数据包。

软防在应对小流量DDoS时，可以搞得住。但一旦遇到大流量的DDoS，软防是抗不住的，可以把系统资源消耗尽，服务器直接卡死。

从成本上说，软防成本很低。

2、硬件防火墙（硬防）

和防软不同，硬防是把防火墙程序做到硬件芯片中，由单独的硬件执行防护功能，减少了CPU的负担，性能上比软防高出很多，当然了，成本也比软防高得多。

综上，不管是软防还是硬防，其原理上都差不多；但是软防是基于系统的，硬防是基于硬件的。在面对稍大的DDoS时，软防基本上是无能为力的，而硬防也不是能抗得住所有的DDoS，不同配置的硬防能承受的DDoS流量不同。假设硬防只能抗住1G的流量，而你的网站受到了2G的DDoS流量，硬防也是白搭！

总体来说：

防火墙防不了DDoS攻击，DDoS只是访问量增加，除非你不让别人访问，当流量超出服务器的网络带宽时，服务器就会拥堵到无法访问，也就是瘫痪。

最常见的不过是ping攻击，也是最简单，防御ping攻击很简单，服务器IP禁ping即可,设置禁ping还可以防御其他攻击，有的黑客是否停止攻击的依据就是你的服务器是否ping的通，因为发起DDoS攻击也是需要成本的，需要大量的肉鸡等等，不可能长时间持续对一个目标攻击，除非是针对性的！

> 所以，禁ping是一种简单而又有效的手段。禁用服务器的ICMP响应，这样别人无法通过Ping来判断你的服务器IP是否存活。

第二就是真实IP的影藏。 网站启用CDN来隐藏后端服务器的真实IP，CDN本身也带有一定的抗洪能力。但是这个仅仅对于静态资源而言，比较有效。

接入层可以加一层高防服务，来防御 DDos 攻击

如果有钱，解决DDoS攻击最根本的办法就是买高防，比如墨者安全高防防御流量1000G，国内DDOS一般最多几十到几百G左右吧。

顾名思义，“高防服务器”就是能够为企业抵御 DDoS/CC 攻击的服务器。在云计算时代，游戏、APP、金融、电商等有需求的业务可以通过接入 DDoS 高防服务来获得这种高防的能力。DDoS 是一种耗尽攻击目标的系统资源导致其无法响应正常的服务请求的攻击方式，DDoS 的防护系统，本质上是一个基于资源较量和规则过滤的智能化系统。针对流行的 DDoS/CC 流量型攻击，DDoS 高防服务可以通过云端清洗集群、数据库监控牵引系统等技术进行有效的削弱。

市面上的“高防服务器”、“高防服务”有很多，但资源是无底洞，选择高防服务可以从安全性、易用性、成本效益等方面来考量。其中安全性是核心，不能缓解攻击的，再便宜也是白扔钱。安全性的决定因素：

- **服务器和带宽资源。**主流云服务商的高峰服务都可以防护 SYN Flood、ACK Flood、ICMP Flood、UDP Flood、DNS Flood、HTTP Flood等常见的攻击类型，以及针对应用层的 CC 攻击，但最终能否防护成功还要看厂商可防护带宽的大小。比如网易云易盾 DDoS 高防服务，提供 **1T 超大防护带宽，单 IP 防护能力最大可达数百 G**，就是认为超大带宽才能从容应对超大流量攻击。
- **流量分析能力。**唯有基于海量数据分析，进而对合法用户进行模型化，才能对 DDoS 流量进行精确清洗。借助网易云易盾 DDoS 高防服务，用户可以通过配置高防 IP，将攻击流量引流到高防 IP，抵御超大流量 DDoS 攻击。
- **弹性扩展能力。**DDoS 流量巨大，防护系统支持弹性扩展，采用基础预付费+弹性后付费，既可以在避免带宽浪费的前提下进行有效防护，又可以节约成本。