顺风车高并发处理

1. 高并发概述

> 高并发（High Concurrency）是互联网分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计保证系统能够同时并行处理很多请求。

高并发一方面可以提高资源利用率，加快系统响应速度，但是同时也会带来稳定性，分布式事务、死锁等问题。

- 并发：处理器视角，能不能一个人同时做多件事？
- 并行：外部视角，多件事是不是在同时进行且互不干扰？

1.1 度量指标

> 并发的指标一般有QPS，TPS，IOPS，并发用户数，PV，响应时间等。

1.1.1 QPS

每秒响应请求数，是一台服务器每秒能够相应的查询次数，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准, 即每秒的响应请求数。

1.1.2 TPS

Transactions Per Second，也就是事务数/秒。一个事务是指一个客户机向服务器发送请求然后服务器做出反应的过程。客户机在发送请求时开始计时，收到服务器响应后结束计时，以此来计算使用的时间和完成的事务个数。

QPS基本类似于TPS，但是不同的是，对于一个页面的一次访问，形成一个TPS；但一次页面请求，可能产生多次对服务器的请求，服务器对这些请求，就可计入“QPS”之中。

如果访问一个接口，请求只有一个，这种情况下TPS=QPS

1.1.3 并发用户数

同时承载正常使用系统功能的用户数量。例如一个即时通讯系统，同时在线量一定程度上代表了系统的并发用户数。

1.1.4 平均响应时间

系统对请求做出响应的时间。例如系统处理一个HTTP请求需要200ms，这个200ms就是系统的响应时间。一般而言，用户感知友好的高并发系统，时延应该控制在250毫秒以内。

QPS = 并发量 / 平均响应时间

1.1.5 PV

PV（Page View）：页面访问量，即页面浏览量或点击量，用户每次刷新即被计算一次。可以统计服务一天的访问日志得到。

1.1.5 UV

独立访客，也就是独立IP（Unique Visitor）,按访问来源统计。1个人哪怕访问10次，也是1个uv，但是是10个pv

1.1.7 小结

tps和qps区别？

tps重业务视角，qps重服务器视角。一般 qps > tps

pv和uv的区别？

pv是访问量，uv是用户量

并发量和吞吐量区别？

并发量指的是同一时刻向服务器的请求数量。吞吐量是指单位时间内，成功传输的数据量。

机器数如何推算？

假设峰值为每天80%的访问集中在20%的时间里，即峰值时间。（总PV * 80%）/（1天的总秒数 * 20%）---> 峰值时间QPS。 峰值时间每秒QPS / 单台机器能承受的最大QPS ---> 需要多少机器 。

1.2 设计思路

> 互联网分布式架构设计，提高系统并发能力的方式，方法论上主要有两种：垂直扩展（Scale Up，也叫竖向扩展）与水平扩展（Scale Out，也叫横向扩展）。

1.2.1 垂直方向:提升单机能力

> 提升单机处理能力又可分为硬件和软件两个方面

1.2.1.1 硬件方向

升级服务器硬件，购买多核高频机器，大内存，大容量磁盘等

1.2.1.2 软件方向

包括用更快的数据结构（编程语言级别的并发编程），改进架构，应用多线程、协程（select/poll/epoll等IO多路复用技术），以及上性能优化各种手段，但是这种方式很容易出现瓶颈。

1.2.2 水平方向:分布式集群

> 为了解决分布式系统的复杂性问题，一般会用到架构分层和服务拆分，通过分层做隔离，通过微服务解耦。

这个理论上没有上限，只要做好层次和服务划分，加机器扩容就能满足需求，但实际上并非如此，一方面分布式会增加系统复杂性，另一方面集群规模上去之后，也会引入一堆服务发现、服务治理的新问题。

因为垂直向的限制，所以，我们通常更关注水平扩展，高并发系统的实施也主要围绕水平方向展开。

2. 应对措施

我们的打车系统在正式上线后必将会面对大量用户访问，面对各种层级的高并发请求，因此我们会采用高性能的服务器、高性能的数据库、高效率的编程语言、高性能的Web容器等。但是这几个方面，还无法从根本解决大型网站面临的高负载和高并发问题。因此我们必须对此做出相应的策略和技术解决方案

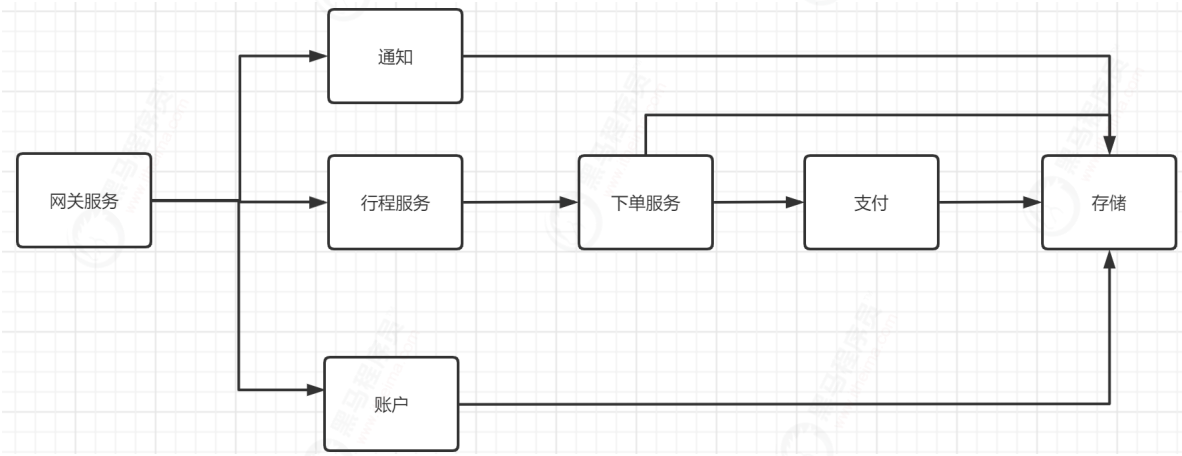
2.1 微服务体系

> 高内聚，低耦合，灵活部署与升级

使用集群是网站解决高并发、海量数据问题的常用手段。当一台服务器的处理能力、存储空间不足时，不要企图去更换更强大的服务器，对大型网站而言，不管多么强大的服务器，都满足不了网站持续增长的业务需求。这种情况下，更恰当的做法是增加一台服务器分担原有服务器的访问及存储压力。 对网站架构而言，只要能通过增加一台服务器的方式改善负载压力，就可以以同样的方式持续增加服务器不断改善系统性能，从而实现系统的可伸缩性。

2.1.1 集群架构

> 具体调用流程如下图



2.1.2 服务列表

> 我们整体服务分为以下服务

- 网关服务:主要负责外部流量进入,以及限流和权限校验
- 账户服务:服务用户生成Token,用户验证,以及用户管理
- 通知服务:主要负责客户和司机端的实时聊天沟通以及消息暂存功能
- 行程服务:最核心的模块,主要负责行程管理,发布行程,接单等操作
- 下单服务:主要管理订单,行程完成后生成订单等操作
- 支付服务:主要完成订单的支付工作,对接微信支付
- 存储服务:存储服务主要负责数据的存储以及文件存储

2.1.3 快速部署（供参考）

└build.sh :

git获取代码并编译打包，借助maven的docker插件推送到镜像库

使用方式：

build.sh order ，后面order为服务名，可以是order，storage.....等模块，如果不指定，则打包全部

如果指定web则不需要打包，前端页面拉取一下即可

```
cd /opt/hitch/src/hitch/
#git pull origin master
module=""
if [ "$1" != "" ];then
    module="hitch-$1"
fi
if [ "$1" != "web" ];then
    cd backend/$module
    mvn clean install -Dmaven.test.skip=true -Ppro
fi
cd /opt/hitch
```

deploy.sh :

部署指定的服务

使用方式：

deploy.sh order , 参数同上

```
docker ps | grep "hitch-$1" | grep -v nacos | awk '{print $1}' | xargs docker rm
-f
docker images | grep "hitch-$1" | awk '{print $3}' | xargs docker rmi -f
cd /opt/hitch/src/hitch
docker-compose -f /opt/hitch/src/hitch/docker-compose.yml up -d
cd /opt/hitch
```

2.2 负载均衡

> 负载均衡将是大型网站解决高负荷访问和大量并发请求采用的终极解决办法

单个重负载的运算分担到多台节点设备上做并行处理，每个节点设备处理结束后，将结果汇总，返回给用户，系统处理能力得到大幅度提高。

大量的并发访问或数据流量分担到多台节点设备上分别处理，减少用户等待响应的时间，这主要针对Web服务器、FTP服务器、企业关键应用服务器等网络应用。

2.2.1 负载均衡的作用

2.2.1.1 转发功能

按照一定的算法【权重、轮询】，将客户端请求转发到不同应用服务器上，减轻单个服务器压力，提高系统并发量。

2.2.1.2 故障移除

通过心跳检测的方式，判断应用服务器当前是否可以正常工作，如果服务器期宕掉，自动将请求发送到其他应用服务器。

2.2.1.3 恢复添加

如检测到发生故障的应用服务器恢复工作，自动将其添加到处理用户请求队伍中。

2.2.2 负载均衡方案

> 负载均衡可以分为3种：

2.2.2.1 DNS负载均衡

客户端通过URL发起网络服务请求的时候，会去DNS服务器做域名解释，DNS会按一定的策略（比如就近策略）把URL转换成IP地址，同一个URL会被解释成不同的IP地址，这便是DNS负载均衡，它是一种粗粒度的负载均衡，它只用URL前半部分，因为DNS负载均衡一般采用就近原则，所以通常能降低时延，但DNS有cache，所以也会更新不及时的问题。

2.2.2.2 硬件负载均衡

> 通过布置F5，A10等专门的负载均衡设备到机房做负载均衡，性能高，但是价格昂贵。

2.2.2.3 软件负载均衡

> 利用软件实现四层负载均衡（LVS）和七层负载均衡（Nginx）

我们的请求通过网关将数据分配到了不同的服务节点，我们使用了springcloud微服务架构，任意一个节点可以随着业务量的增加来扩充节点使得我们的服务能够经受住更到访问量的压力

2.2.3 负载均衡算法

upstream{

a 192.168.1.1 weight=4

b 192.168.1.2 weight=2

c 192.168.1.3 weight=1

}

2.2.3.1 轮询

> 轮询为负载均衡中较为基础也较为简单的算法，它不需要配置额外参数。

假设配置文件中共有 台服务器，该算法遍历服务器节点列表，并按节点次序每轮选择一台服务器处理请求。当所有节点均被调用过一次后，该算法将从第一个节点开始重新一轮遍历。

特点：由于该算法中每个请求按时间顺序逐一分配到不同的服务器处理，因此适用于服务器性能相近的集群情况，其中每个服务器承载相同的负载。但对于服务器性能不同的集群而言，该算法容易引发资源分配不合理等问题。

2.2.3.2 加权轮询

> 为了避免普通轮询带来的弊端，加权轮询应运而生。在加权轮询中，每个服务器会有各自的`weight`。

一般情况下，`weight`的值越大意味着该服务器的性能越好，可以承载更多的请求。该算法中，客户端的请求按权值比例分配，当一个请求到达时，优先为其分配权值最大的服务器。

特点：加权轮询可以应用于服务器性能不等的集群中，使资源分配更加合理化

拓展：为解决机器平滑出现的问题，nginx的源码中使用了一种平滑的加权轮询的算法，规则如下：

- 每个节点两个权重，weight和currentWeight，weight永远不变是配置时的值，current不停变化
- 变化规律如下：选择前所有current+=weight，选current最大的响应，响应后让它的current-=total

次数	响应前	被选中	响应后
1	4，2，1	a	-3，2，1
2	1，4，2	b	1，-3，2
3	5，-1，3	a	-2，-1，3
4	2，1，4	c	2，1，-3
5	6，3，-2	a	-1，3，-2
6	3，5，-1	b	3，-2，-1
7	7，0，0	a	0，0，0

2.2.3.3 IP 哈希

`ip_hash` 依据发出请求的客户端 IP 的 hash 值来分配服务器，该算法可以保证同 IP 发出的请求映射到同一服务器，或者具有相同 hash 值的不同 IP 映射到同一服务器。

特点：该算法在一定程度上解决了集群部署环境下 Session 不共享的问题。

2.2.4 Nginx负载均衡

2.2.4.1 负载均衡配置

> 我们采用加权轮询的方式进行nginx的负载均衡

```
upstream myServer {
    server 192.168.72.49:8081 down;
    server 192.168.72.49:8082 weight=2;
    server 192.168.72.49:8083;
    server 192.168.72.49:8084 backup;
}
```

2.2.4.2 参数解释

- down： 表示单前的server暂时不参与负载
- Weight： 默认为1.weight越大，负载的权重就越大。
- max_fails： 允许请求失败的次数默认为1.当超过最大次数时，返回proxy_next_upstream 模块定义的错误
- fail_timeout： max_fails 次失败后，暂停的时间
- Backup： 其它所有的非backup机器down或者忙的时候，请求backup机器。所以这台机器压力会最轻。

2.2.5 GateWay负载均衡

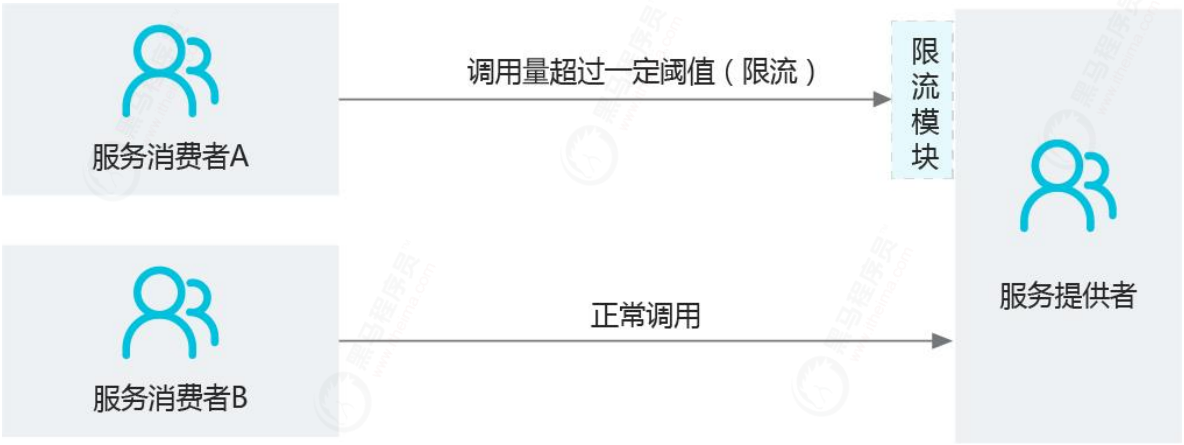
> GateWay本身是支持负载均衡功能的，我们只需要在gateway中配置的代理路径加上`lb`的前缀就会自动进行负载均衡

2.2.5.1 GateWay配置

```
server:
  port: @project.server.prot@
```

```yaml
spring:
  application:
    name: @project.server.name@
  cloud:
    nacos:
      server-addr: @nacos.addr@
    gateway:
      routes:
        # 账户服务API代理
        - id: hitch-account-server
          uri: lb://hitch-account-server
          predicates:
            - Path=/account/**
        # 库存服务API代理
        - id: hitch-storage-server
          uri: lb://hitch-storage-server
          predicates:
            - Path=/storage/**
        # 行程服务
        - id: hitch-stroke-server
          uri: lb://hitch-stroke-server
          predicates:
            - Path=/stroke/**
        # 订单服务
        - id: hitch-order-server
          uri: lb://hitch-order-server
          predicates:
            - Path=/order/**
        # 支付管理
        - id: hitch-payment-server
          uri: lb://hitch-payment-server
          predicates:
            - Path=/payment/**
        # 存储管理
        - id: hitch-storage-server
          uri: lb://hitch-storage-server
          predicates:
            - Path=/storage/**
        # 通知服务
        - id: hitch-notice-server
          uri: lb://hitch-notice-server
          predicates:
            - Path=/notice/**
        # 消息推送服务 websocket代理
        - id: hitch-notice-service-ws
          uri: lb:ws://hitch-notice-service
          predicates:
            - Path=/ws/**
```

2.4 限流



限流策略通常是用来在高qps下进行流量限制的，常见的方式有计数器、令牌桶、漏桶。在这次活动中我负责的模块是控制的对下游的流量，我们可以让那些请求选择丢弃、等待或者降级这些限流算法可以自行实现也可以利用现有的限流工具，比如说Guava的令牌桶，具体看场景需求吧，下面来看一下这几种限流策略，再说说我写的限流方式。

2.4.1 限流算法

> 常见的限流算法有：计数器、漏桶和令牌桶算法。

2.4.1.1 计数器限流

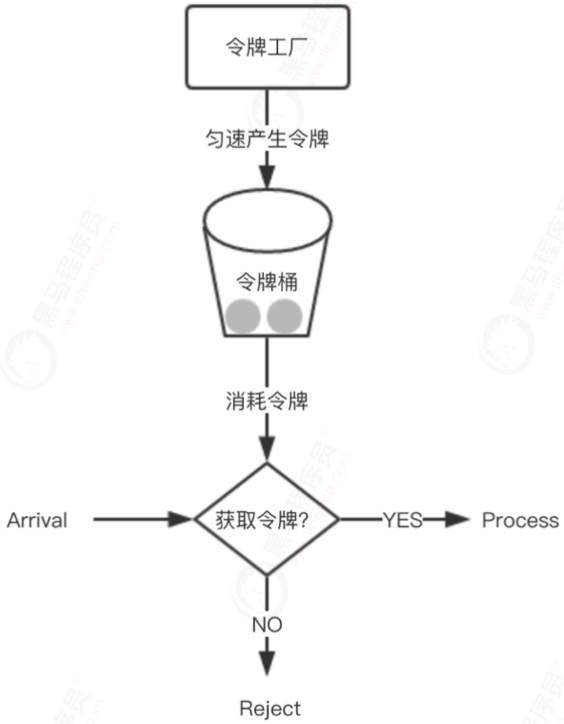
> 这种方式比较粗暴，相当于有一个计数器来控制单秒请求数，也就是qps定义的方式。

比如说限流2000，每秒钟对于计数器进行置0操作，当一秒内到达2000时就不再接受请求了。这样能保证较长时间短的流量均匀，但是单秒内部实际上是不均匀的，可能这2000个请求，在前0.1s就处理完成了，后面的都是被丢掉的，并且峰值qps 可能是达到2w的。

2.4.1.2 令牌桶限流

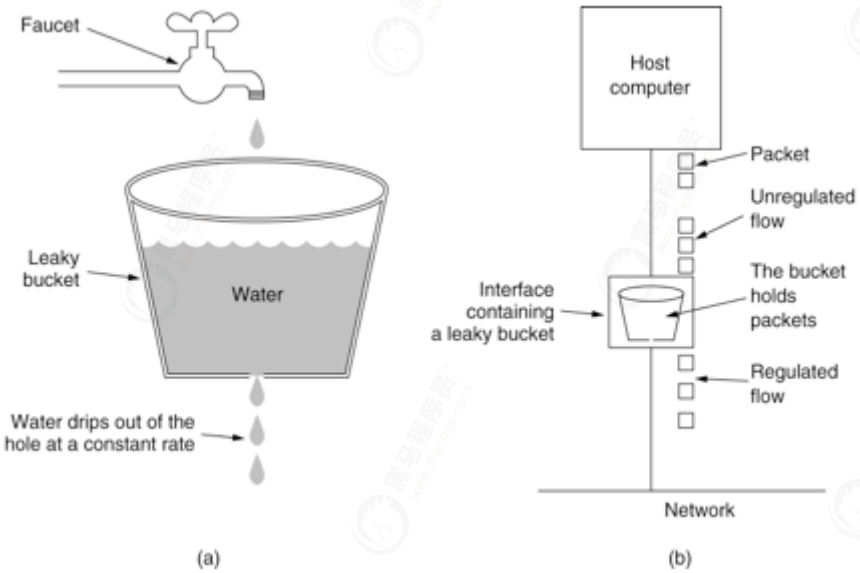
> 令牌桶限流是指我们可以设立一个令牌桶，然后以固定的速率往令牌桶中添加令牌，令牌桶满则不添加。

请求到来时检查如果令牌桶中有令牌则取走令牌，发起请求。假设要限定 2000 qps，则1/2000 的速率向令牌桶添加令牌，也可以1/1000 一次性添加两个令牌，以此类推。令牌桶在持续高qps 下是没问题的，可以把流量限制的比较均匀。但是面对突发流量时，流量桶里是满的，可能一瞬间把令牌抢空完成请求，这里的问题和计数器限流实际上是一样的，峰值可能远远大于2000，所以对于突发流量是限不住的。下面看看令牌桶的示意图：其实我感觉令牌桶更像是优化后的计数器限流，只不过时间窗口由1s变的更细了



2.4.1.3 漏斗限流

这个是使用最多的一种限流算法，通常用来流量整形或者流量控制，看起来和令牌桶比较像，但是差异还是比较大的。漏斗往桶里加的是请求，不是令牌，相当于新请求到达时放到桶中，如果桶满了则溢出请求，桶以匀速漏出请求进行处理，比如qps 2000，则1/2000 s 漏出一个请求进行处理。这种限流方式比较稳定，但是需要维护一个请求队列或者任务队列。



2.4.1.4 漏斗和令牌桶比较

1. 令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时则拒绝新的请求；
2. 漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；
3. 令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量；
4. 漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率；

5. 令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率；
6. 两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。

2.4.2 nginx漏斗限流

> Nginx官方版本限制IP的连接和并发分别有两个模块：

- limit_req_zone 用来限制单位时间内的请求数，即速率限制,采用的漏桶算法 "leaky bucket"。
- limit_req_conn 用来限制同一时间连接数，即并发限制。

2.4.2.1 nginx配置

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    server {
        location /search/ {
            limit_req zone=one burst=3 nodelay;
        }
    }
```

2.4.2.2 参数解释

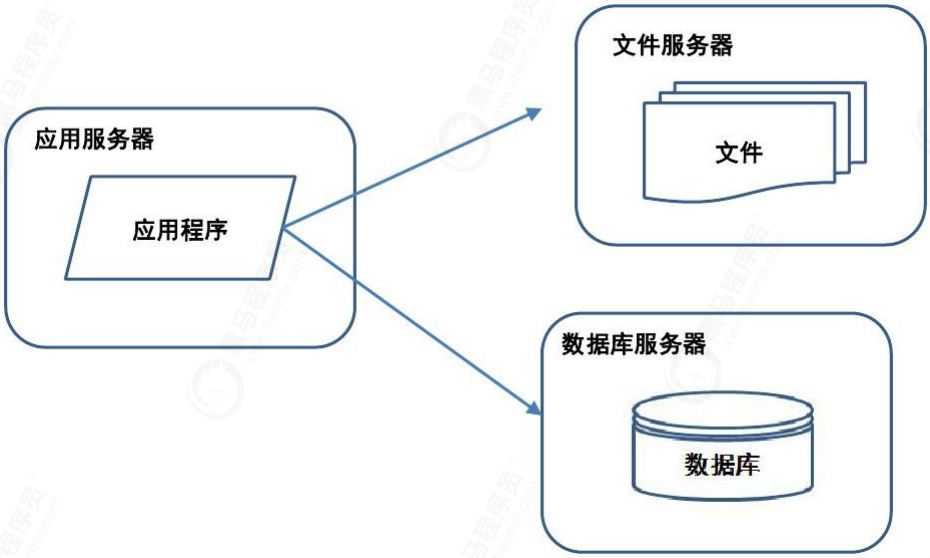
```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
```

- 第一个参数：$binary_remote_addr 表示通过remote*addr这个标识来做限制，“binary”*的目的是缩写内存占用量，是限制同一客户端ip地址。
- 第二个参数：zone=one:10m表示生成一个大小为10M，名字为one的内存区域，用来存储访问的频次信息。
- 第三个参数：rate=1r/s表示允许相同标识的客户端的访问频次，这里限制的是每秒1次，还可以有比如30r/m的。

```
limit_req zone=one burst=5 nodelay;
```

- 第一个参数：zone=one 设置使用哪个配置区域来做限制，与上面limit_req_zone 里的name对应。
- 第二个参数：burst=3，重点说明一下这个配置，burst爆发的意思，这个配置的意思是设置一个大小为3的缓冲区当有大量请求（爆发）过来时，超过了访问频次限制的请求可以先放到这个缓冲区内。
- 第三个参数：nodelay，如果设置，超过访问频次而且缓冲区也满了的时候就会直接返回503，如果没有设置，则所有请求会等待排队。

2.5 图片服务器判重



架构体系： 入口nginx - 动静分离 - 图片 - fastdfs。

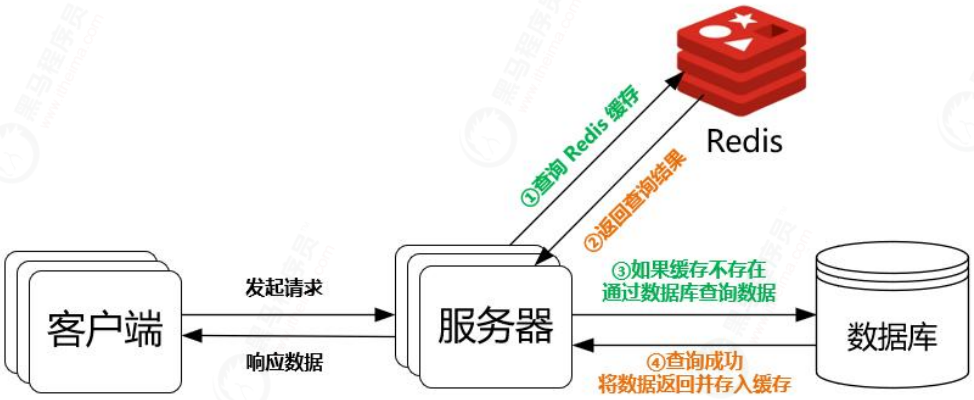
2.5.1 计算文件MD5

> 上传文件的时候有一个保存文件记录的的表,保存有文件的元数据以及文件的MD5

如果出现上传文件的Md5和数据库中的值一致,说明文件已经上传,只要将表中保存的文件的URL返回即可,这样可以提高文件的存储性能以及节省磁盘空间

```
public ResponseVO<AttachmentPO> uploadFile(MultipartFile file) throws Exception
{
    //校验文件是否为空
    if (file.isEmpty()) {
        throw new BusinessRuntimeException(BusinessErrors.DATA_NOT_EXIST, "文件不
存在");
    }
    //构建附件对象
    AttachmentPO attachmentPO = getAttachmentPO(file);
    //根据文件签名查询文件是否存在
    AttachmentPO tmp = attachmentMapper.selectByMd5(attachmentPO.getMd5());
    //如果存在直接将附件对象返回
    if (null != tmp) {
        return ResponseVO.success(tmp);
    }
    //如果不存在则上传文件以及添加数据到附件表
    String url = dfsUploadFile(file);
    attachmentPO.setUrl(url);
    attachmentMapper.insert(attachmentPO);
    return ResponseVO.success(attachmentPO);
}
```

2.6 使用缓存



> 使用缓存改善网站性能

网站访问的特点和现实世界的财富分配一样遵循二八定律：80%的业务访问集中在20%的数据上，既然大部分业务访问集中在一小部分数据上，那么如果把这一小部分数据缓存在内存中，就可以减少数据库的访问压力，提高整个网站的数据访问速度，改善数据库的写入性能了。

2.6.1 使用SpringCache

> 这里我们使用了springCache来作为我们的应用缓存

2.6.1.1 SpringCache简介

Spring从`3.1`开始定义了`org.springframework.cache.Cache`和`org.springframework.cache.CacheManager`接口来统一不同的缓存技术；并支持使用JCache（JSR-107）注解简化我们开发；Cache接口为缓存的组件规范定义，包含缓存的各种操作集合；Cache接口下Spring提供了各种xxxCache的实现；如`RedisCache`，`EhCacheCache`，`ConcurrentMapCache`等

2.6.2.2 缓存中的重要概念

注解	说明
Cache	缓存接口，定义缓存操作。实现有:RedisCache、EhcacheCache、ConcurrentMapCache等
CacheManager	缓存管理器，管理各种缓存组件
@Cacheable	注意针对方法配置，能够根据方法的请求参数对其进行缓存
@CacheEvict	清空缓存
@CachePut	保证方法被调用，又希望结果被缓存。与@Cacheable区别在于是否每次都调用方法，常用于更新
@EnableCaching	开启基于注解的缓存
keyGenerator	缓存数据时key生成策略
serialize	缓存数据时value序列化策略
@CacheConfig	统一配置类的缓存注解属性

> @Cacheable/@CachePut/@CacheEvict 主要的参数

属性	说明
value	缓存的名称，在 spring 配置文件中定义，必须指定至少一个 例如：@Cacheable(value=”mycache”) 或者 @Cacheable(value={”cache1”,”cache2”}
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合 例如：@Cacheable(value=”testcache”,key=”#id”)
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存例如：@Cacheable(value=”testcache”,condition=” #userName.length()>2”)
unless	否定缓存。当条件结果为TRUE时，就不会缓存 @Cacheable(value=”testcache”,unless=” #userName.length()>2”)
allEntries (@CacheEvict)	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存 例如：@CachEvict(value=”testcache”,allEntries=true)
beforeInvocation(@CacheEvict)	是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存例如：@CachEvict(value=”testcache”，beforeInvocation=true)

2.6.2.3 启动SpringCache

> 在springboot的启动类中加入`@EnableCaching`注解来启用SpringCache的功能

```
@SpringBootApplication
@MapperScan("com.heima.storage.mapper")
//开启缓存注解
@EnableCaching
public class StorageApplication {
    public static void main(String[] args) {
        SpringApplication.run(StorageApplication.class, args);
    }
}
```

2.6.2.4 缓存数据

> 在要缓存的方法上面添加@Cacheable注解，即可缓存这个方法的返回值。
>
> @Cacheable注解会先查询是否已经有缓存，有会使用缓存，没有则会执行方法并缓存

```
@RequestMapping("/selectByID/{id}")
@Cacheable(cacheNames = "com.heima.modules.po.StrokePO", key = "#id")
public StrokePO select(@PathVariable("id") String id) {
    return strokeMapper.selectByPrimaryKey(id);
}
```

2.6.2.5 删除缓存

> @CacheEvict能够删除一条缓存记录

```
@RequestMapping("/update")
@CacheEvict(cacheNames = "com.heima.modules.po.StrokePO", key = "#strokePO.id")
public void update(@RequestBody StrokePO strokePO) {
    strokeMapper.updateByPrimaryKeySelective(strokePO);
}
```

2.6.2.6 配置

```
  cache:
    type: REDIS
    redis:
      cache-null-values: false
      time-to-live: 10000ms
      use-key-prefix: true
```

2.7 消息队列

> 分布式缓存在读多写少的场景性能优异，对于写操作较多的场景可以采用消息队列集群，它可以很好地做写请求异步化处理，实现削峰填谷的效果。

MQ是分布式架构中的解耦神器，应用非常普遍。有些分布式事务也是利用MQ来做的。由于其高吞吐量，在一些业务比较复杂的情况，可以先做基本的数据验证，然后将数据放入MQ，由消费者异步去处理后续的复杂业务逻辑，这样可以大大提高请求响应速度，提升用户体验。如果消费者业务处理比较复杂，也可以独立集群部署，根据实际处理能力需求部署多个节点。

2.7.1 业务分析

> 因为用户打车成功后需要实时汇报当前的GEO轨迹坐标，进行后续的行程计算以及大数据分析以及风控监控

因为打车轨迹数据的数据量太大，假如有一万个用户在打车，轨迹需要每秒汇报一次，那这个系统需要承担每秒一万的并发，并且这个并发可能一直持续，将会产生海量的数据，并且用户对于这些轨迹数据是不太关心只有后台用轨迹数据进行分析数据，基本上不需要考虑延时。

2.7.2 使用MQ削峰

这里就要使用MQ来进行异步处理，将我们的轨迹数据先扔到MQ中然后后台在慢慢消费，MQ在其中充当一个消费填谷的作用。

2.7.2.1 代码实现

> 调用该接口的坐标会添加到MQ等待后续的处理

```
/**
  * 发送实时位置服务
  *
  * @param locationVO
  * @return
  */
public ResponseVO<LocationVO> realtimeLocation(LocationVO locationVO) {
    mqProducer.sendLocation(JSON.toJSONString(locationVO));
    return ResponseVO.success(locationVO);
}
```

> 消费者消费消息队列,并将大量的GEO数据批量添加到数据库中

```
/**
  * 行程位置监听
  *
  */

@RabbitListener(
    bindings =
    {
        @QueueBinding(value = @Queue(value = RabbitConfig.STROKE_LOCATION_QUEUE,
durable = "true"),
                      exchange = @Exchange(value =
RabbitConfig.STROKE_LOCATION_QUEUE_EXCHANGE), key =
RabbitConfig.STROKE_LOCATION_KEY)
    }, concurrency = "batchQueueRabbitListenerContainerFactory")
@Override
public void onMessageBatch(List<Message> messages) {
    List<LocationPO> locationPOList = getLocationPOList(messages);
    locationService.batchSaveLocation(locationPOList);
}
```

> 这里面我们将数据添加到了MongoDB,下面我们介绍下为什么使用MongoDB

2.8 使用NoSQL

> 因为我们系统需要涉及到大量的数据的读写,用传统的数据库可能存在性能问题,对于大量的消息,GEO数据,也不是特别重要,丢失一些也是可以接受的,我们这里采用NOSQL来存储消息以及GEO的坐标数据
>
> -- 容忍丢失，照顾吞吐

2.8.1 MongoDB方案

使用MongoDB存储数据在保证用户数据的存储速度以及数据量和安全性上面找到一个平衡，因为mongodb是一个nosql数据库，支付水平扩展，数据量上面理论上可以无限扩展，并且MongoDB的存储速度可以很快，因为数据量太大，可能处理的数据不及时，我们可以采用批量插入的方式进行批量插入。

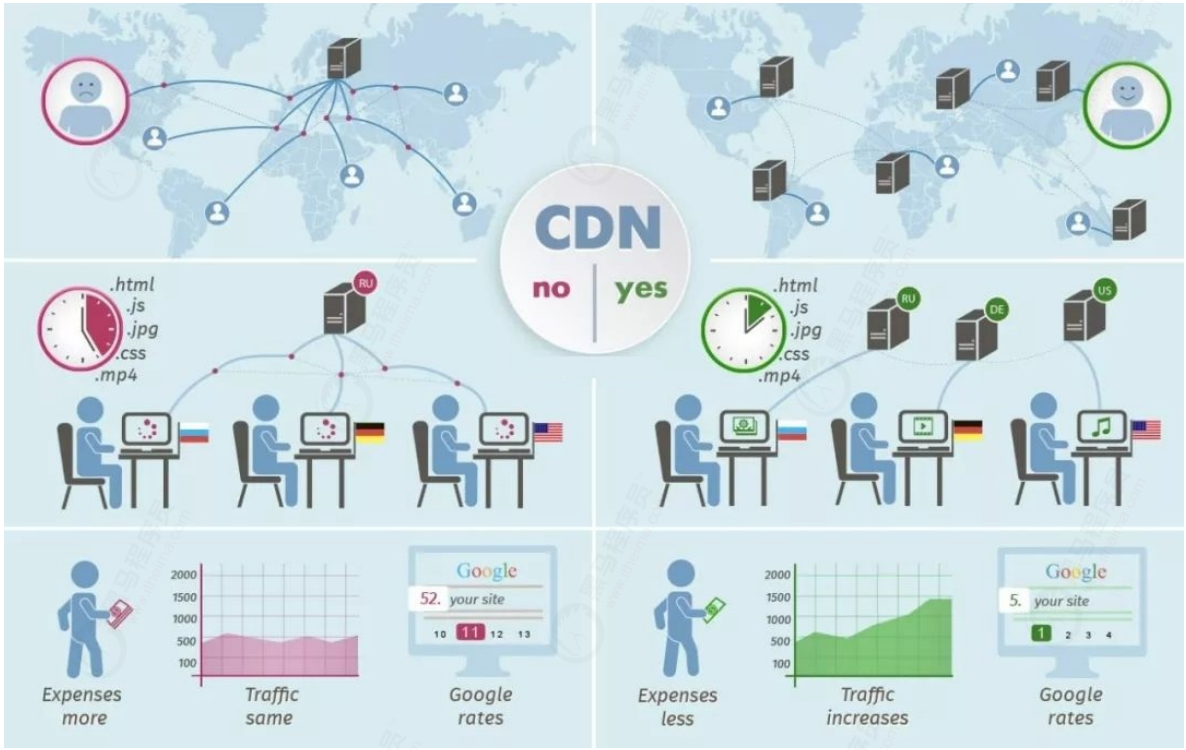
2.8.2 批量插入GEO数据

```
/**
  * 批量添加位置信息
  *
  * @param locationPOList
  */
@Override
public void batchSaveLocation(List<LocationPO> locationPOList) {
    mongoTemplate.insert(locationPOList, HtichConstants.LOCATION_COLLECTION);
}
```

3. 其他技术（拓展）

> 因为环境原因,一些问题我们自己无法实现,可以将技术分享给大家

3.1 CDN网站加速



3.1.1 什么是CDN

对于CDN加速，简单而言，就是把目前网站的内容能够发送到其他靠近用户的服务器上，使用户能够就近获取所需要的信息内容，提升用户浏览网站时的访问速度。而且，CDN加速成本低，速度快，能够大大提高网站信息内容流动的效率，确保网站能够正常的浏览，提升网站访问速度。

3.1.2 为什么使用CDN

3.1.2.1 提高访问速度

CDN加速能够让用户就近就能获得所需要的内容，很好的解决了因分布、带宽、服务器所带来的浏览延时等问题，极大的提高了用户浏览网站的访问速度和成功率，对当下而言，能够控制延迟问题是一项非常重要的指标。

3.1.2.2 提高搜索排名

网站访问速度提高是有利于搜索引擎排名，因为很多搜素引擎都会把网站的打开速度当做一个比较重要的指标，因此当网站的访问速度过慢时就会影响搜索排名，而使用CDN加速后，网站不仅打开速度变快了，跳出率也减少了，而且也增加用户对网站的友好体验。

3.1.2.3 对用户友好

CDN加速就像是护航者和加速者，初衷就是为了能够顺利保障信息内容的连贯性，尽可能的减少资源在转发、传输、链路抖动等情况下受到影响，更加快准狠的触发用户的需求，给用户更好的使用体验。

3.1.3 CDN相关技术

> CDN的实现需要依赖多种网络技术的支持，其中最主要的包括负载均衡技术、动态内容分发与复制技术、缓存技术等。

3.1.3.1 负载均衡技术

> 负载均衡技术不仅仅应用于CDN中，在网络的很多领域都得到了广泛的应用，如服务器的负载均衡、网络流量的负载均衡。

顾名思义，网络中的负载均衡就是将网络的流量尽可能均匀分配到几个能完成相同任务的服务器或网络节点上，由此来避免部分网络节点过载。这样既可以提高网络流量，又提高了网络的整体性能。在CDN中，负载均衡又分为服务器负载均衡和服务器整体负载均衡(也有的称为服务器全局负载均衡)。服务器负载均衡是指能够在性能不同的服务器之间进行任务分配，既能保证性能差的服务器不成为系统的瓶颈，又能保证性能高的服务器的资源得到充分利用。而服务器整体负载均衡允许Web网络托管商、门户站点和企业根据地理位置分配内容和服务。通过使用多站点内容和服务来提高容错性和可用性，防止因本地网或区域网络中断、断电或自然灾害而导致的故障。在CDN的方案中服务器整体负载均衡将发挥重要作用，其性能高低将直接影响整个CDN的性能。

3.1.3.2 动态分发与复制技术

> 众所周知，网站访问响应速度取决于许多因素，如网络的带宽是否有瓶颈、传输途中的路由是否有阻塞和延迟、网站服务器的处理能力及访问距离等。

多数情况下，网站响应速度和访问者与网站服务器之间的距离有密切的关系。如果访问者和网站之间的距离过远的话，它们之间的通信一样需要经过重重的路由转发和处理，网络延误不可避免。一个有效的方法就是利用内容分发与复制技术，将占网站主体的大部分静态网页、图像和流媒体数据分发复制到各地的加速节点上。所以动态内容分发与复制技术也是CDN所需的一个主要技术。

3.1.3.3 缓存技术

> 缓存技术已经不是一种新鲜技术。Web缓存服务通过几种方式来改善用户的响应时间，如代理缓存服务、透明代理缓存服务、使用重定向服务的透明代理缓存服务等。

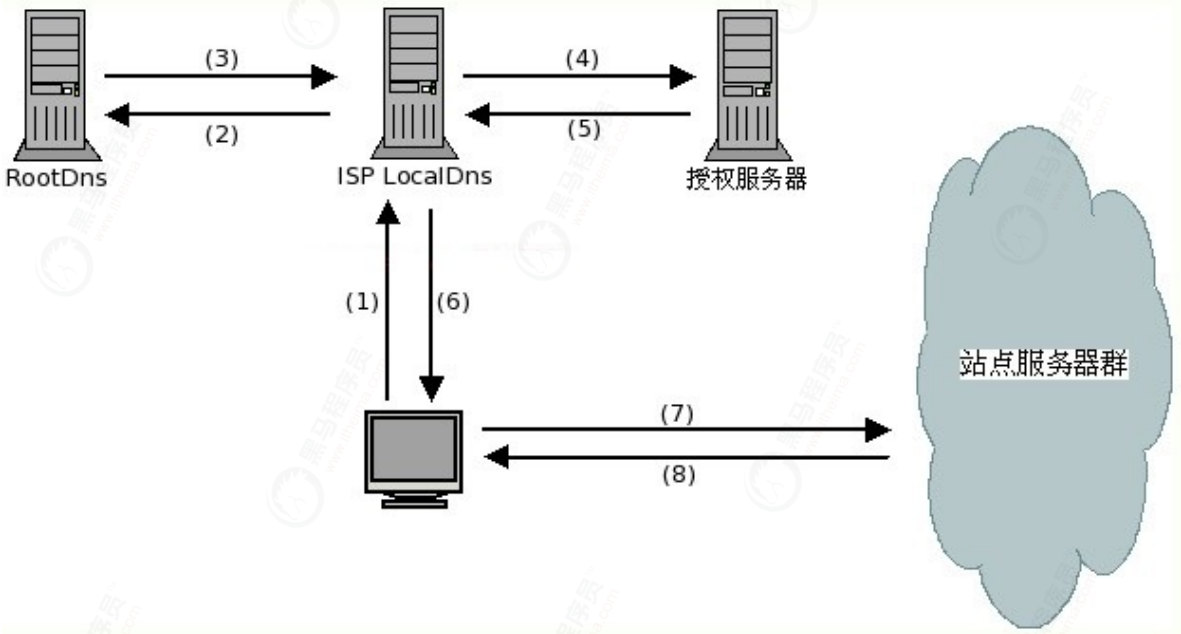
通过Web缓存服务，用户访问网页时可以将广域网的流量降至最低。对于公司内联网用户来说，这意味着将内容在本地缓存，而无须通过专用的广域网来检索网页。对于Internet用户来说，这意味着将内容存储在他们的ISP的缓存器中，而无须通过Internet来检索网页。这样无疑会提高用户的访问速度。CDN的核心作用正是提高网络的访问速度，所以，缓存技术将是CDN所采用的又一个主要技术。

3.1.4 工作原理

> CDN网络是在用户和服务器之间增加Cache层，主要是通过接管DNS实现，将用户的请求引导到Cache上获得源服务器的数据，从而降低网络的访问时间。

3.1.4.1 传统的访问过程

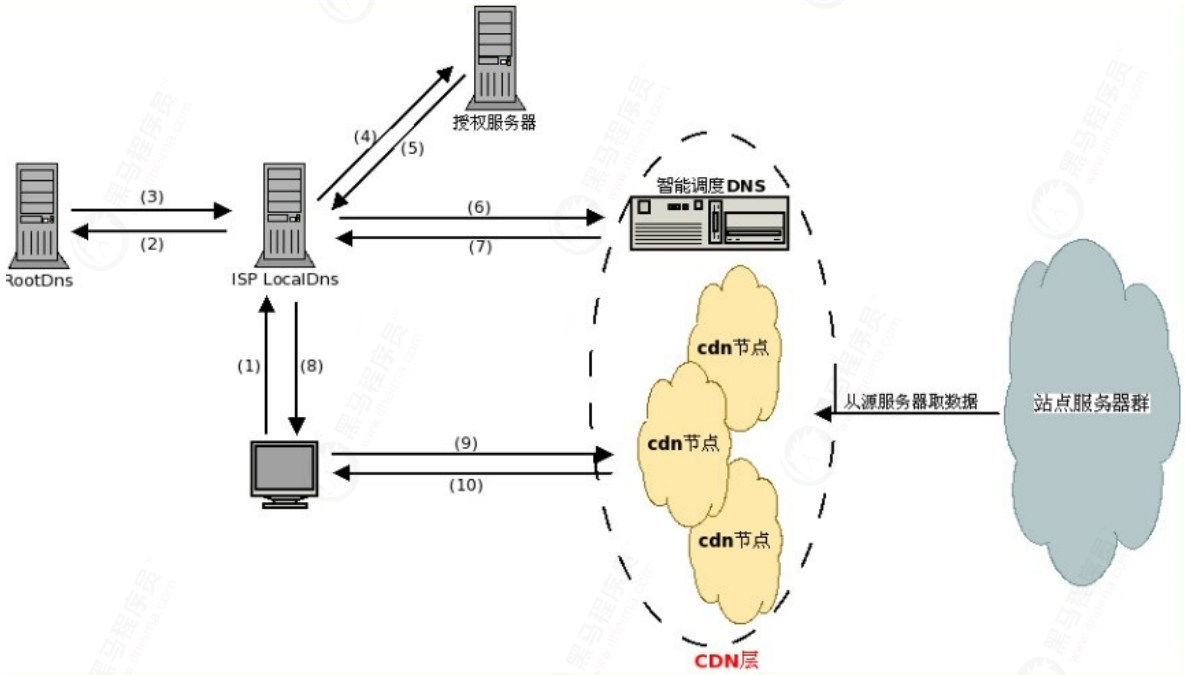
> 首先，让我们看一下传统的未加缓存服务的访问过程：



> 如图可以看出，传统的网络访问的流程如下：

1. 用户输入访问的域名,操作系统向 LocalDns 查询域名的ip地址；
2. LocalDns向 ROOT DNS 查询域名的授权服务器(这里假设LocalDns缓存过期)；
3. ROOT DNS将域名授权dns记录回应给 LocalDns；
4. LocalDns得到域名的授权dns记录后，继续向域名授权dns查询域名的ip地址；
5. 域名授权dns 查询域名记录后，回应给 LocalDns；
6. LocalDns 将得到的域名ip地址，回应给用户端；
7. 用户得到域名ip地址后，访问站点服务器；
8. 站点服务器应答请求，将内容返回给客户端.

3.1.4.2 CDN加速访问过程



1. 用户输入访问的域名,操作系统向 LocalDns 查询域名的ip地址；
2. LocalDns向 ROOT DNS 查询域名的授权服务器(这里假设LocalDns缓存过期)；
3. ROOT DNS将域名授权dns记录回应给 LocalDns；
4. LocalDns得到域名的授权dns记录后,继续向域名授权dns查询域名的ip地址；
5. 域名授权dns 查询域名记录后(一般是CNAME)，回应给 LocalDns；
6. LocalDns 得到域名记录后,向智能调度DNS查询域名的ip地址；
7. 智能调度DNS 根据一定的算法和策略(比如静态拓扑，容量等),将最适合的CDN节点ip地址回应给 LocalDns；
8. LocalDns 将得到的域名ip地址，回应给用户端；
9. 用户得到域名ip地址后，访问站点服务器。