如何设计一个高并发&高可用系统

一、高可用业务架构设计实践

1、系统设计的一些原则

在互联网项目开发中，总是不断针对新的需求去研发新的系统，而很多系统的设计都是可以触类旁通的：

海恩法则

- 事故的发生是量的积累的结果 （并发量，数据量，服务量.....）
- 再好的技术、再完美的规章， 在实际操作层面也无法取代人自身的素质和责任心 。

墨菲定律

- 任何事情都没有表面看起来那么简单 。
- 所有事情的发展都会比你预计的时间长 。
- 会出错的事总会出错。
- 如果你担心某种情况发生，那么它更有可能发生。

警示我们，在互联网公司里，对生产环境发生的任何怪异现象和问题 都不要轻易忽视，对于其背后的原因一定要彻查。

同样，海恩法则也强调任何严重事故的背后 都是多次小问题的积累，积累到一定的量级后会导致质变，严重的问题就会浮出水面 。

那么，我们需要对线上服务产生的任何征兆，哪怕是一个小问题，也要刨根问底: 这就需要我们有技术攻关的能力，对任何现象都要秉着以下原则： 为什么发生？ 发生了怎么应对？ 怎么恢复？ 怎么避免？ 对问题要彻查，不能因为问题的现象不明显而忽略 。

2、软件架构中的高可用设计

2.1、什么是高可用？

高可用 HA（High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，

通过设计减少系统不能提供服务的时间。

假设系统一直能够提供服务，我们说系统的可用性是 100%。

如果系统每运行 100 个时间单位，会有 1 个时间单位无法提供服务，我们说系统的可用性是 99%。

很多公司的高可用目标是 4 个 9，也就是 99.99%，这就意味着，系统的年停机时间为 8.76 个小时。

百度的搜索首页，是业内公认高可用保障非常出色的系统，甚至人们会通过 www.baidu.com 能不能访问来判断“网络的连通性”，百度高可用的服务让人留下啦“网络通畅，百度就能访问”，“百度打不开，应该是网络连不上”的印象，这其实是对百度 HA 最高的褒奖。

2.2、可用性度量和考核

所谓业务可用性(availability)也即系统正常运行时间的百分比，架构组最主要的 KPI (Key Performance Indicators ，关键业绩指标)。对于我们提供的服务（web，api）来说，现在业界更倾向用 N 个 9 来量化可用性， 最常说的就是类似 “4 个 9(也就是 99.99%)” 的可用性。

描述	通俗叫法	可用性级别	年度停
基本可用性	2个9	99%	87.6小
较高可用性	3个9	99.9%	8.8小时
具有故障自动恢复能力的可用性	4个9	99.99%	53分钟
极高可用性	5个9	99.999%	5 分钟

故障时间=故障修复时间点-故障发现（报告）时间点
服务年度可用时间%=（1-故障时间/年度时间）× 100%

故障的度量与考核
对管理者而言：可用性是产品的整体考核指标。 每个工程师而言：使用故障分来考核：

类别	描述
高危S级事故故障	一旦出现故障，可能会导致服务整体不可用
严重A级故障	客户明显感知服务异常：错误的回答
中级B级故障	客户能够感知服务异常：响应比较慢
一般C级故障	服务出现短时间内抖动

服务级别可用性：
如果是一个分布式架构设计，系统由很多微服务组成，所有的服务可用性不可能都是统一的标准。
为了提高我们服务可用性，我们需要对服务进行分类管理并明确每个服务级别的可用性要求。

类别	服务	可用性要求	描述
一级核心服务	核心产品或者服务	99.99%（全年53分钟不可用）	系统引擎部分：一旦出现故障，
二级重要服务	重要的产品功能	99.95%（全年260分钟不可用）	类比汽车轮子：该服务出现问题
三级一般服务	一般功能	99.9%（全年8.8小时不可用）	类比汽车倒车影像：该部分出现
四级工具服务	工具类是服务	99%	非业务功能：比如爬虫、管理后

2.2、如何保障系统的高可用？

我们都知道，单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。

方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他 backup 能够顶上。

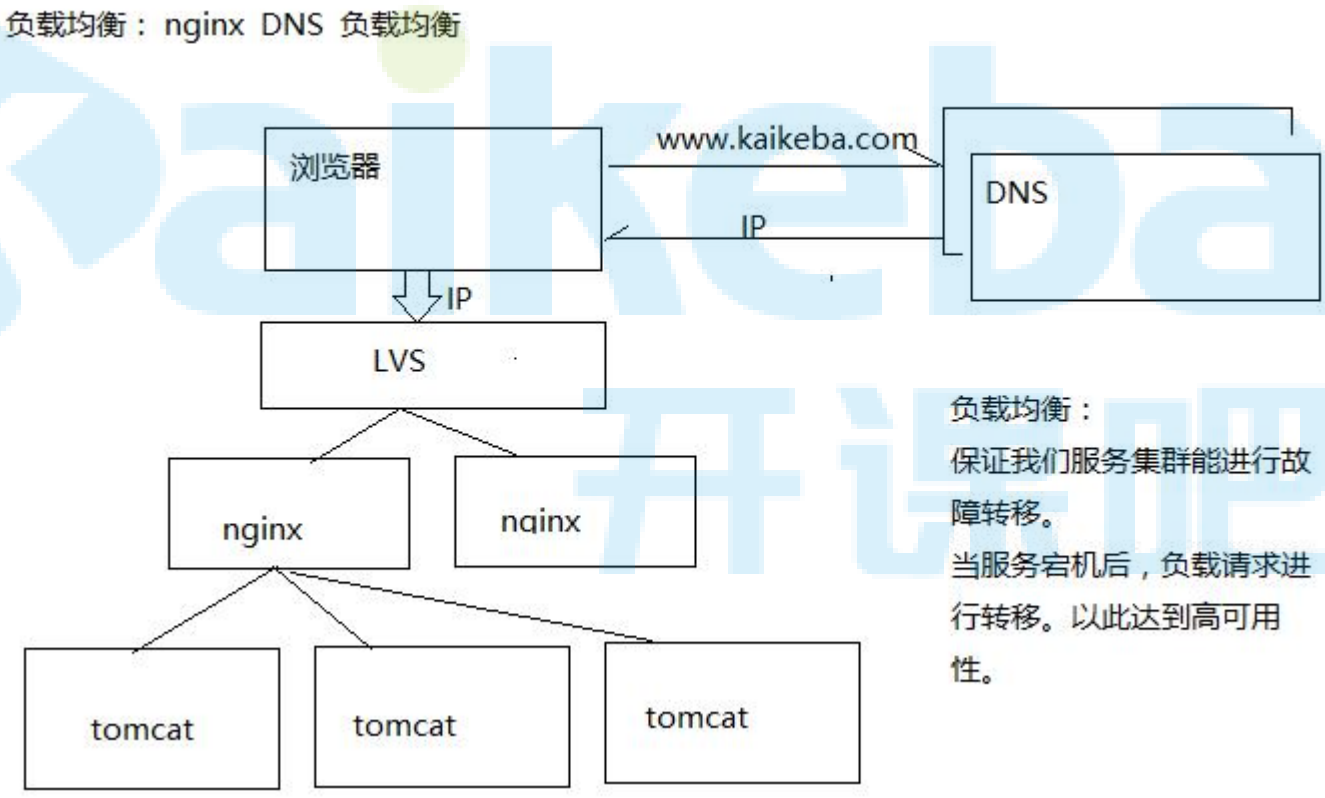
保证系统高可用，架构设计的核心准则是：冗余。有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

接下来我们看下典型互联网架构中，解决高可用问题具体有哪些方案：

1、负载均衡
2、限流
3、降级
4、隔离
5、超时与重试
6、回滚
7、压测与预案

3、负载均衡

3.1、DNS&nginx 负载均衡



以上负载均衡方案是接入层的方案，实际上负载均衡的地方还有很多：
1、服务和服务 RPC --- RPC 框架 提供负载方案 （DUBBO，SpringCloud）
2、数据集群需要负载均衡（mycat,haproxy）

3.2、upstream 配置

第一步我们需要给 Nginx 配置上游服务器，即负载均衡到的真实处理业务的服务器，通过在

http 指令下配置 upstream 即可。

```
upstream backend {
    server 192.168.61.1:9080  weight=
    server 192.168.61.1:9090  weight=
}
```

proxy_pass 来处理用户请求。

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

3.3、负载均衡算法

负载均衡用来解决用户请求到来时如何选择 upstream server 进行处理，默认采用的是 round-robin（轮询），同时支持其他几种算法。

- round-robin： 轮询，默认负载均衡算法，即以轮询的方式将请求转发到上游服务器，通过配合 weight 配置可以实现基于权重的轮询

- ip_hash ：根据客户 IP 进行负载均衡，即相同的 IP 将负载均衡到同一个 upstream server。

```
upstream backend {
    ip_hash;
    server 192.168.61.1:9080  weigh
    server 192.168.61.1:9090  weigh
}
```

- hash key [consistent]： 对某一个 key 进行哈希或者使用一致性哈希算法进行负载均衡。使用 Hash 算法存在的问题是，当添加/删除一台服务器时，将导致很多 key 被重新负载均衡到不同的服务器（从而导致后端可能出现问题）；因此，建议考虑使用一致性哈希算法，这样当添加/删除一台服务器时，只有少数 key 将被重新负载均衡到不同的服务器。

哈希算法： 此处是根据请求 uri 进行负载均衡，可以使用 Nginx 变量，因此，可以实现复杂的算法。

```
upstream backend {
    hash $uri;
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2;
}
```

一致性哈希算法： consistent_key 动态指定。

```
upstream nginx_local_server {
    hash $consistent_key consistent;
    server 192.168.61.1:9080  weight=
    server 192.168.61.1:9090  weight=
}
```

如下 location 指定了一致性哈希 key，此处会优先考虑请求参数 cat（类目），如果没有，则再根据请求 uri 进行负载均衡。

```
location / {
    set $consistent_key $arg_cat;
    if ($consistent_key = "") {
        set $consistent_key $request_ur
    }
}
```

而实际我们是通过 Lua 设置一致性哈希 key

```
set_by_lua_file $consistent_key "lua_balancing.lua

local consistent_key = args.cat
if not consistent_key or consistent_key == '' then
   consistent_key = ngx_var.request_uri
end

local value = balancing_cache:get(consistent_key)
if not value then
```

3.4、失败重试

```
upstream backend {
    server 192.168.61.1:9080 max_fails=2 fail_timeout=10s we
    server 192.168.61.1:9090 max_fails=2 fail_timeout=10s we
}
```

通过配置上游服务器的 max_fails 和 fail_timeout，来指定每个上游服务器，当 fail_timeout 时间内失败了 max_fails 次请求，则认为该上游服务器不可用/不存活，然后将摘掉该上游服

务器，fail_timeout 时间后会再次将该服务器加入到存活上游服务器列表进行重试。

```
location /test {
    proxy_connect_timeout 5s;
    proxy_read_timeout 5s;
    proxy_send_timeout 5s;

    proxy_next_upstream error
    proxy_next_upstream_timeou
    proxy_next_upstream_tries
```

然后进行 proxy_next_upstream 相关配置，当遇到配置的错误时，会重试下一台上游服务器。

3.5、监控检查

Nginx 可以集成 nginx_upstream_check_module（https://github.com/yaoweibin/nginx_upstream_check_module）模块来进行主动健康检查。nginx_upstream_check_module 支持 TCP 心跳和 HTTP 心跳来实现健康检查。

TCP 心跳检测：

```
upstream backend {
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2;
    check interval=3000 rise=1 fall=3 timeout=2000
}
```

此处配置使用 TCP 进行心跳检测。

- interval： 检测间隔时间，此处配置了每隔 3s 检测一次。
- fall： 检测失败多少次后，上游服务器被标识为不存活。
- rise： 检测成功多少次后，上游服务器被标识为存活，并可以处理请求。

http 心跳检测：

```
upstream backend {
        server 192.168.61.1:9080  weight=1;
        server 192.168.61.1:9090  weight=2;

        check interval=3000 rise=1 fall=3 timeout=2000 type=ht
        check_http_send "HEAD /status HTTP/1.0\r\n\r\n";
        check_http_expect_alive http_2xx http_3xx;
    }
```

HTTP 心跳检查有如下两个需要额外配置。

· check_http_send： 即检查时发的 HTTP 请求内容。

· check_http_expect_alive： 当上游服务器返回匹配的响应状态码时，则认为上游服务器存活。

此处需要注意，检查间隔时间不能太短，否则可能因为心跳检查包太多造成上游服务器挂掉，同时要设置合理的超时时间。

3.6、其他配置

1）备份服务器

```
upstream backend {
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2 back
}
```

将 9090 端口上游服务器配置为备上游服务器，当所有主上游服务器都不存活时，请求会转发给备上游服务器。
如通过缩容上游服务器进行压测时，要摘掉一些上游服务器进行压测，但为了保险起见会配置一些备上游服务器，当压测的上游服务器都挂掉时，流量可以转发到备上游服务器，从而不影响用户请求处理。

2）不可用服务器

```
upstream backend {
    server 192.168.61.1:9080  weigh
    server 192.168.61.1:9090  weigh
}
```

9090 端口上游服务器配置为永久不可用，当测试或者机器出现故障时，暂时通过该配置临时摘掉机器。

4、隔离术

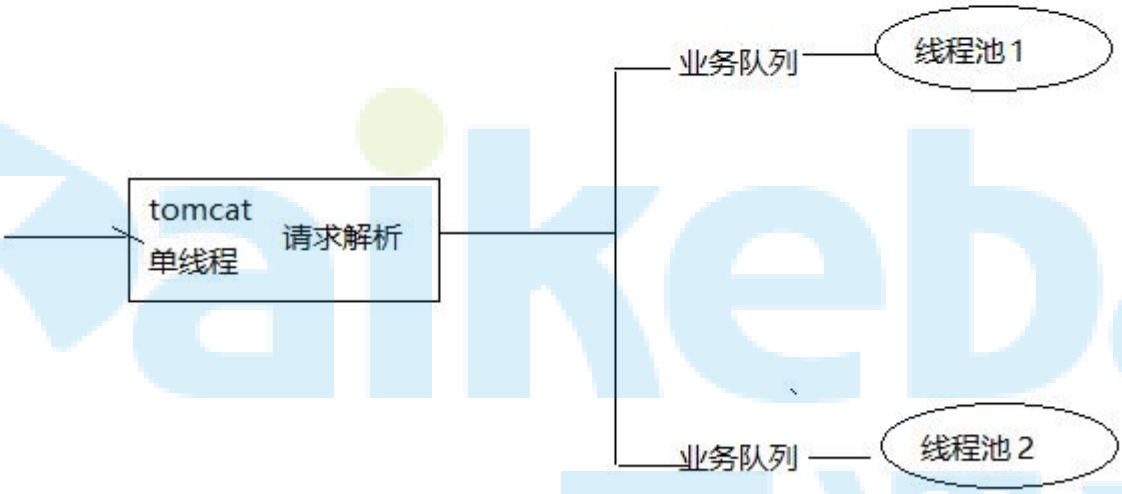
隔离是指将系统或资源分割开，系统隔离是为了在系统发生故障时，能限定传播范围和影响范围，即发生故障后不会出现滚雪球效应，从而保证只有出问题的服务不可用，其他服务还是可用的。

资源隔离通过隔离来减少资源竞争，保障服务间的相互不影响和可用性。

在实际生产环境中，比较多的隔离手段有线程隔离、进程隔离、集群隔离、机房隔离、读写隔离、快慢隔离、动静隔离等。

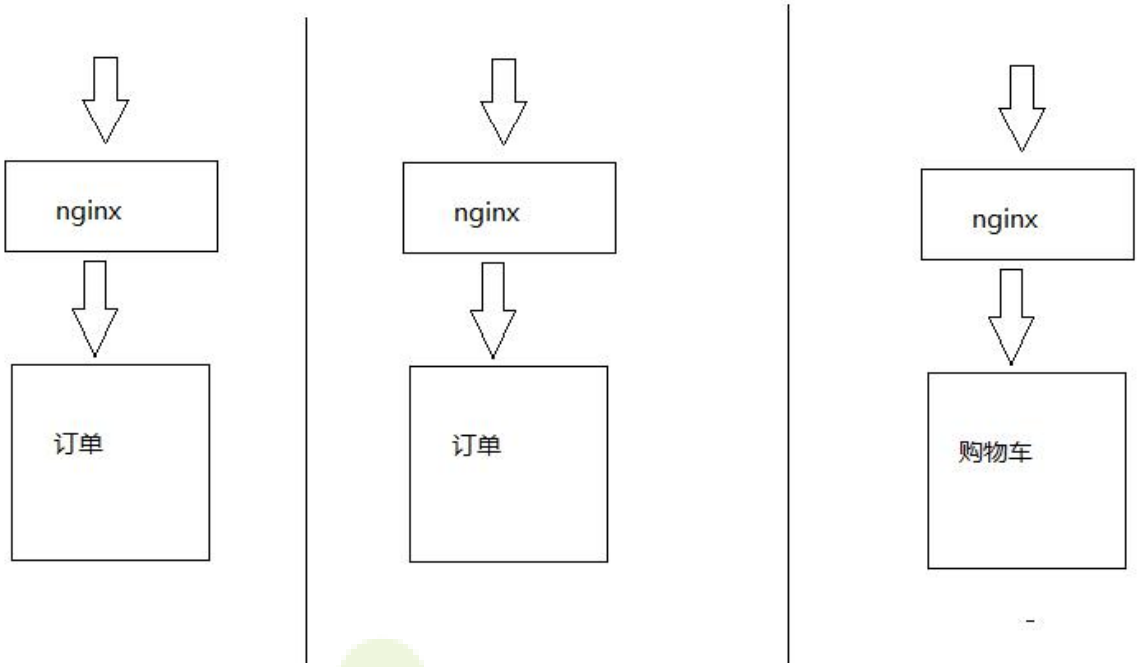
4.1、线程隔离

线程隔离主要指的是 线程池 隔离。 请求分类，交给不同的线程池进行处理。 一个请求出现异常，不会导致故障扩散到其他线程池。
这样就保证系统的高可用性。



4.2、进程隔离

把项目拆分为一个个的子项目，然后让这些子项目进行物理隔离。项目和项目之间没有调用关系

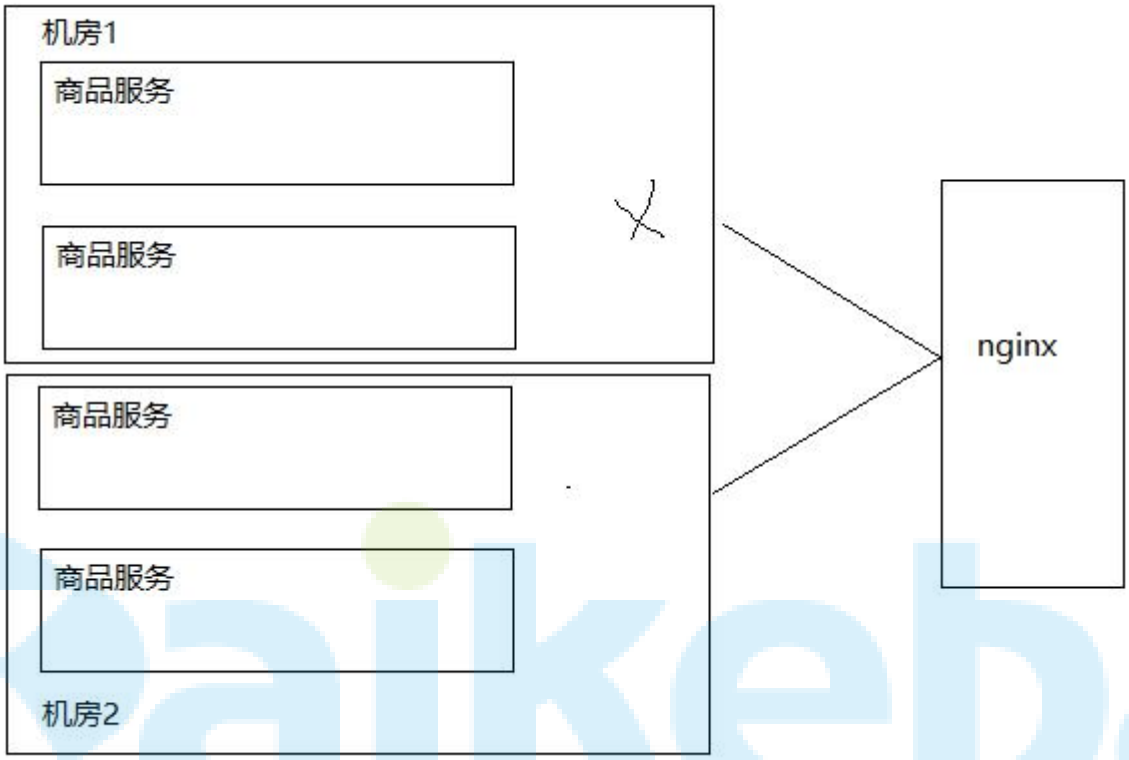


4.3、集群隔离

项目上线后，一定会进行集群部署，为了提高服务高可用性，采用集群隔离术。

对项目进行集群部署；每一个部署结构都是一个进程，当一个进程发生问题的时候，其他进程不会受到影响；

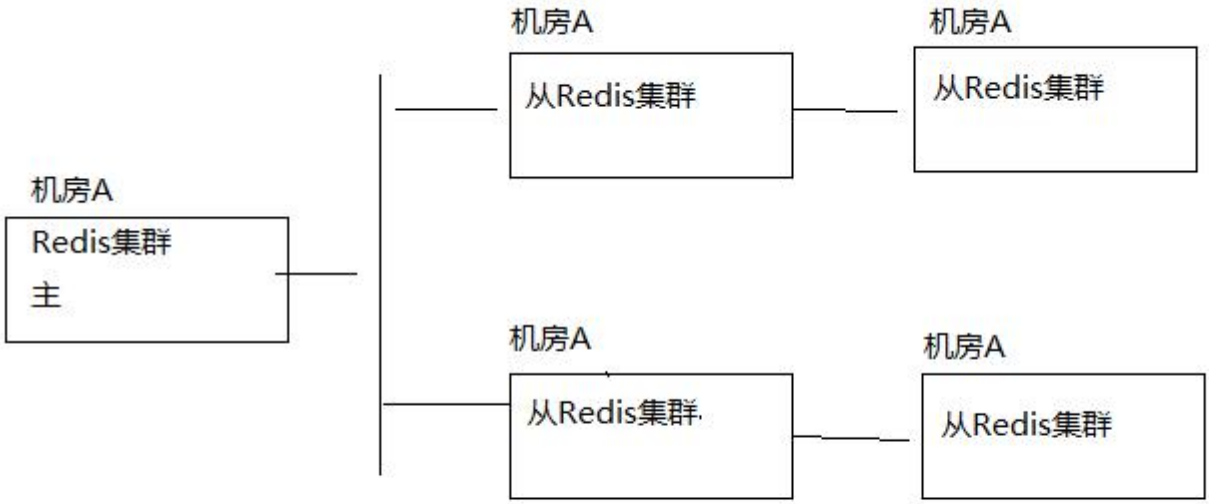
4.4、机房隔离



杭州机房；北京机房；上海机房；（同一个机房，物理隔离）

4.5、读写隔离

Redis 主从 – 读写分离

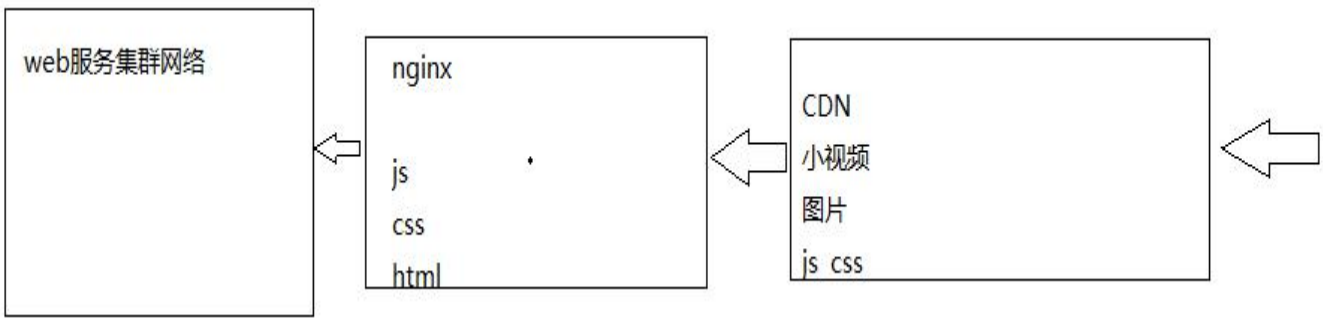


互联网项目中： 读多，写少 ， 读首先成为瓶颈， 扩展读的能力，采用读写分离方式，扩展读的能力

本质上来说： 读写分离就是用来扩展 读的 性能瓶颈

4.6、动静隔离

把静态资源放入 nginx，CDN 服务。达到动静隔离。防止有页面直接加载大量静态资源。。。，
因为访问量大，导致网络带宽打满，
导致卡死，出现不可用。



4.7、热点隔离

秒杀、抢购属于非常合适的热点例子，对于这种热点，是能提前知道的，所以可以将秒杀和抢购做成独立系统或服务进行隔离，从而保证秒杀/抢购流程出现问题时不影响主流程。

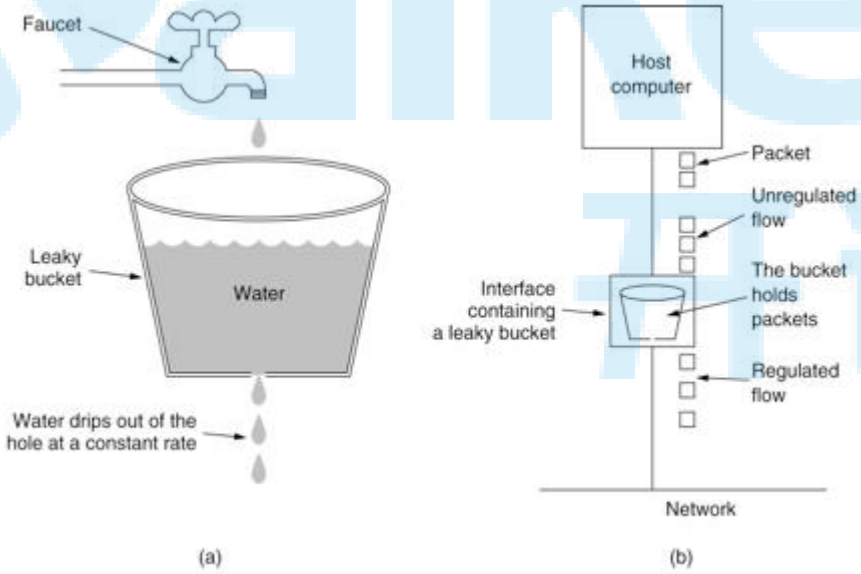
还存在一些热点，可能是因为价格或突发事件引起的。对于读热点，笔者使用多级缓存来搞定，而写热点我们一般通过缓存+队列模式削峰

5、限流

5.1、限流算法

（1）漏桶算法

把请求比作是水，水来了都先放进桶里，并以限定的速度出水，当水来得过猛而出水不够快时就会导致水直接溢出，即拒绝服务。



漏斗有一个进水口 和 一个出水口，出水口以一定速率出水，并且有一个最大出水速率：

在漏斗中没有水的时候：

- 如果进水速率小于等于最大出水速率，那么，出水速率等于进水速率，此时，不会积水
- 如果进水速率大于最大出水速率，那么，漏斗以最大速率出水，此时，多余的水会积在漏斗中

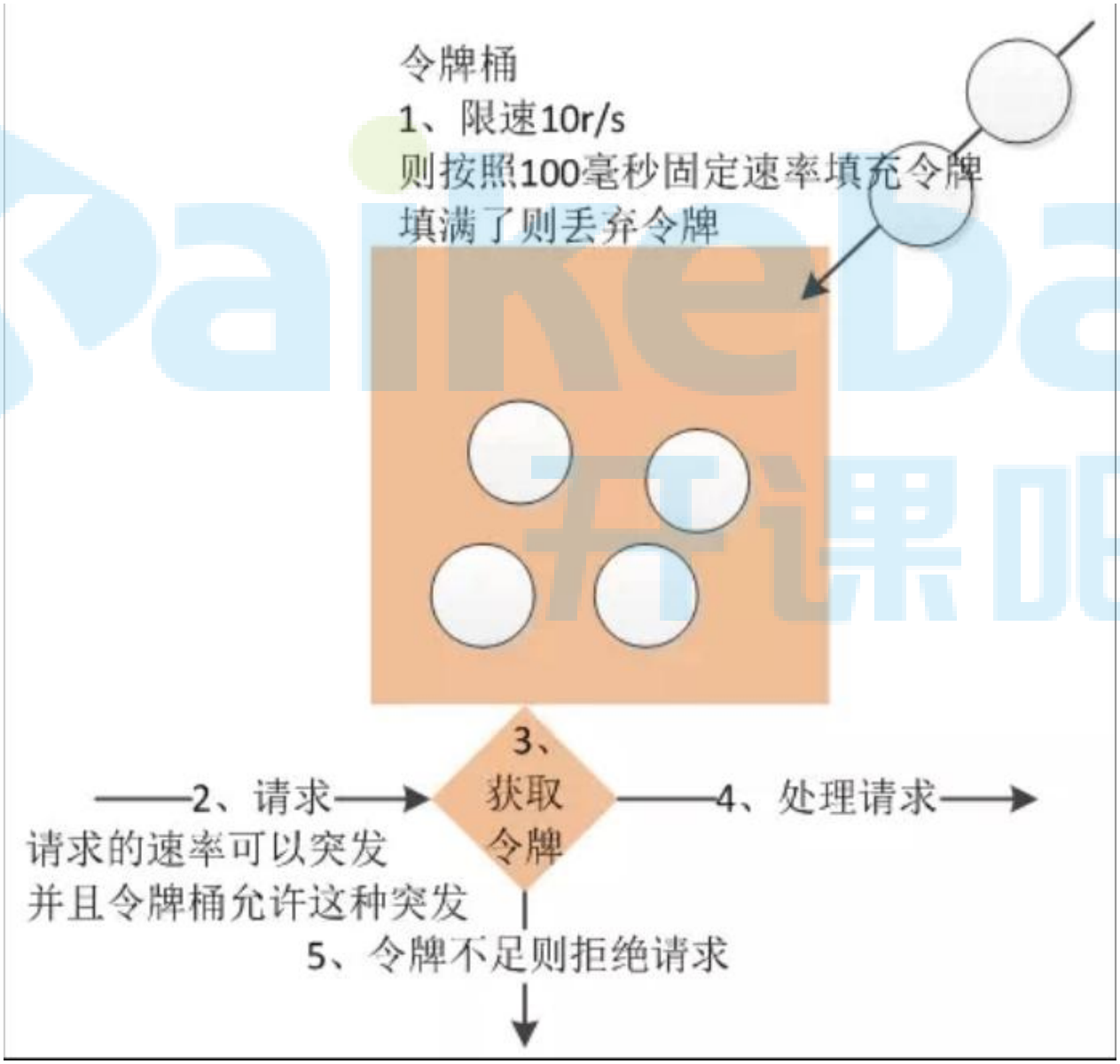
在漏斗中有水的时候

- 出水口以最大速率出水
- 如果漏斗未满，且有进水的话，那么这些水会积在漏斗中
- 如果漏斗已满，且有进水的话，那么这些水会溢出到漏斗之外

（2）令牌桶算法

对于很多应用场景来说，除了要求能够限制数据的平均传输速率外，还要求允许某种程度的突发传输。这时候漏桶算法可能就不合适了，令牌桶算法更为适合。

令牌桶算法的原理是系统以恒定的速率产生令牌，然后把令牌放到令牌桶中，令牌桶有一个容量，当令牌桶满了的时候，再向其中放令牌，那么多余的令牌会被丢弃；当想要处理一个请求的时候，需要从令牌桶中取出一个令牌，如果此时令牌桶中没有令牌，那么则拒绝该请求。

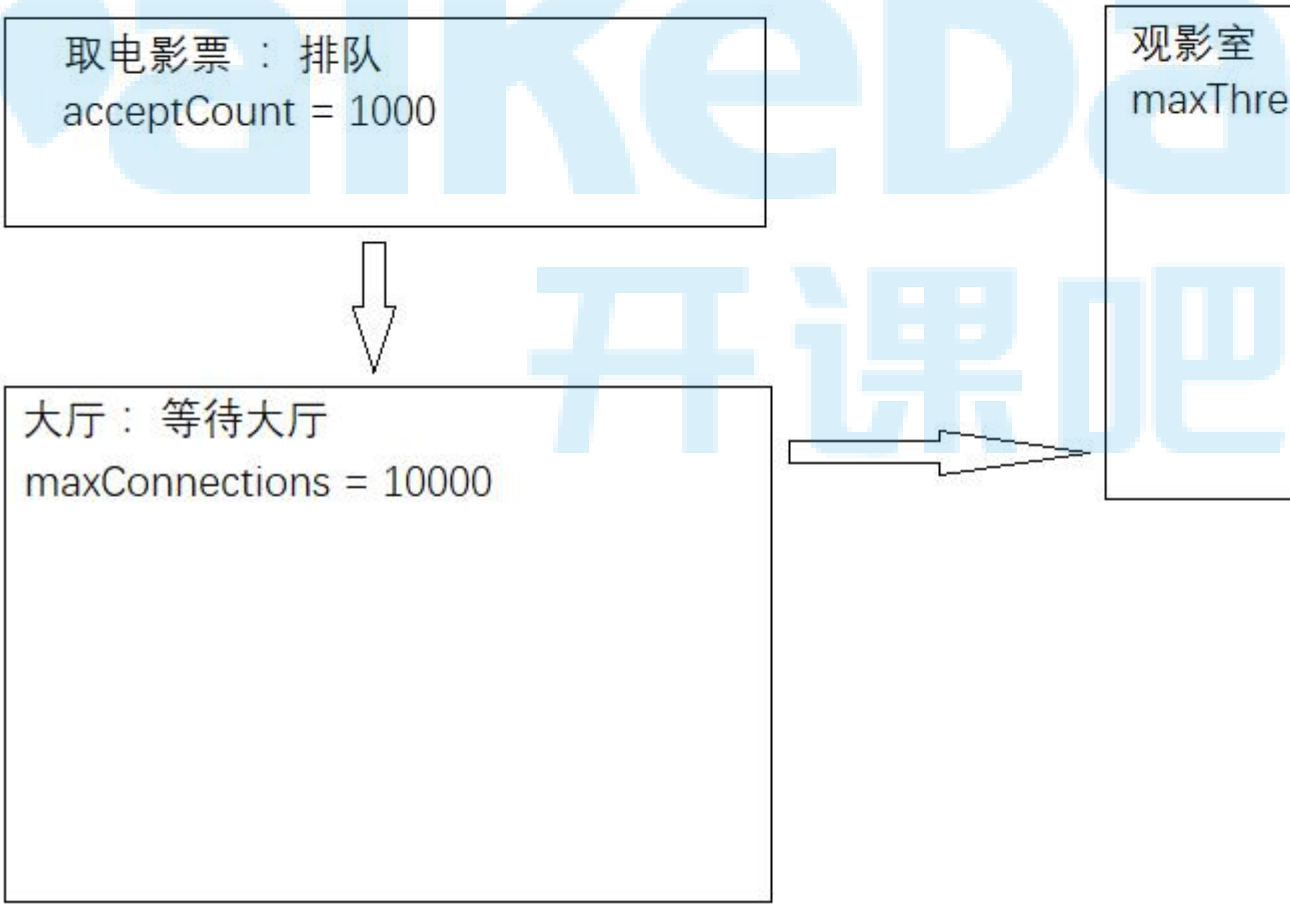


5.2、Tomcat 限流

对于一个应用系统来说，一定会有极限并发/请求数，即总有一个 TPS/QPS 阈值，如果超了阈值，则系统就会不响应用户请求或响应得非常慢，因此我们最好进行过载保护，以防止大量请求涌入击垮系统。

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" maxThreads="800" maxConnections="2000" acce
<!-- A "Connector" using the shared thread pool-->
```

- acceptCount： 如果 Tomcat 的线程都忙于响应，新来的连接会进入队列排队，如果超出排队大小，则拒绝连接；默认值为 100
- maxConnections： 瞬时最大连接数，超出的会排队等待；
- maxThreads： Tomcat 能启动用来处理请求的最大线程数，即同时处理的任务个数，默认值为 200，如果请求处理量一直远远大于最大线程数，则会引起响应变慢甚至会僵死。



5.3、接口限流

限制某个接口/服务每秒/每分钟/每天的请求数/调用量。如一些基础服务会被很多其他系统调用，比如商品详情页服务会调用基础商品服务调用，但是更新量比较大有可能将基础服务打挂。

```
long limit = 1000;
while(true) {
    //得到当前秒
    long currentSeconds = System.currentTimeMillis() / 1
    if(counter.get(currentSeconds).incrementAndGet() > l
        System.out.println("限流了:" + currentSeconds);
        continue;
    }
    //业务处理
```

hystrix 框架进行限流….(限流，熔断….)

5.4、Redis 限流

在分布式环境下，使用第三方限流方法，具体限流方法，使用 Redis+lua 代码，保证操作的原子性：保证线程安全

```
local key = KEYS[1] --限流 KEY（一秒一个）
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call("INCRBY", key, "1")) --请求数+1
if current > limit then --如果超出限流大小
    return 0
elseif current == 1 then --只有第一次访问需要设置 2 秒的过期时间
    redis.call("expire", key,"2")
end
return 1
```

如上操作因是在一个 Lua 脚本中，又因 Redis 是单线程模型，因此线程安全。

5.5、Nginx 限流

对于 Nginx 接入层限流可以使用 Nginx 自带的两个模块：
连接数限流模块 ngx_http_limit_conn_module
漏桶算法实现的请求限流模块 ngx_http_limit_req_module。

ngx_http_limit_conn_module

limit_conn 是对某个 key 对应的总的网络连接数进行限流。可以按照 IP 来限制 IP 维度的总连接数，或者按照服务域名来限制某个域名的总连接数。但是，记住不是每个请求连接都会被计数器统计，只有那些被 Nginx 处理的且已经读取了整个请求头的请求连接才会被计数器统计。

```
http {
    limit_conn_zone $binary_remote_addr zone=addr:10m;
    limit_conn_log_level error;
    limit_conn_status 503;
    ...
    server {
    ...
    location /limit {
        limit_conn addr 1;
    }
```

limit_conn： 要配置存放 key 和计数器的共享内存区域和指定 key 的最大连接数。此处指定的最大连接数是 1，表示 Nginx 最多同时并发处理 1 个连接。

limit_conn_zone：用来配置限流 key 及存放 key 对应信息的共享内存区域大小。此处的 key 是$binary_remote_addr”，表示 IP 地址，也可以使用$server_name 作为 key 来限制域名级别的最大连接数。

limit_conn_status： 配置被限流后返回的状态码，默认返回 503。

limit_conn_log_level： 配置记录被限流后的日志级别，默认 error 级别。

ngx_http_limit_req_module

limit_req 是漏桶算法实现，用于对指定 key 对应的请求进行限流，比如，按照 IP 维度限制请求速率。配置示例如下

```
limit_conn_log_level error;
limit_conn_status 503;
...
server {
...
location /limit {
    limit_req zone=one burst=5 nodela
}
```

limit_req： 配置限流区域、桶容量（突发容量，默认为 0）、是否延迟模式（默认延迟）。

limit_req_zone： 配置限流 key、存放 key 对应信息的共享内存区域大小、固定请求速率。此处指定的 key 是“$binary_remote_addr”，表示 IP 地址。固定请求速率使用 rate 参数配置，支持 10r/s 和 60r/m，即每秒 10 个请求和每分钟 60 个请求。不过，最终都会转换为每秒的固定请求速率（10r/s 为每 100 毫秒处理一个请求，60r/m 为每 1000 毫秒处理一个请求）。

limit_conn_status： 配置被限流后返回的状态码，默认返回 503。

limit_conn_log_level： 配置记录被限流后的日志级别，默认级别为 error。

6、降级

在开发高并发系统时，有很多手段来保护系统，如缓存、降级和限流等。本章来聊聊降级策略。当访问量剧增、服务出现问题（如响应时间长或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。本文将介绍一些笔者在实际工作中遇到的或见到过的一些降级方案，供大家参考。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。降级也需要根据系统的吞吐量、响应时间、可用率等条件进行手工降级或自动降级。

6.1、降级预案

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅，从而梳理出哪些必须誓死保护，哪些可降级。比如，可以参考日志级别设置预案。

- **一般：** 比如，有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级。

- **警告：** 有些服务在一段时间内成功率有波动（如在 95~100%之间），可以自动降级或人工降级，并发送告警。

- **错误：** 比如，可用率低于 90%，或者数据库连接池用完了，或者访问量突然猛增到系统能承受的最大阈值，此时，可以根据情况自动降级或者人工降级。

- **严重错误：** 比如，因为特殊原因数据出现错误，此时，需要紧急人工降级。

降级按照是否自动化可分为：自动开关降级和人工开关降级。

降级按照功能可分为：读服务降级和写服务降级。

降级按照处于的系统层次可分为：多级降级。

降级的功能点主要从服务器端链路考虑，即根据用户访问的服务调用链路来梳理哪里需要降级。

6.2、页面降级

在大促或者某些特殊情况下，某些页面占用了一些稀缺服务资源，在紧急情况下可以对其整个降级，以达到丢卒保帅的目的。

6.3、页面片段降级

比如，商品详情页中的商家部分因为数据错误，此时，需要对其进行降级。

6.4、页面异步请求降级

比如，商品详情页上有推荐信息/配送至等异步加载的请求，如果这些信息响应慢或者后端服务有问题，则可以进行降级。

6.5、服务功能降级

比如，渲染商品详情页时，需要调用一些不太重要的服务（相关分类、热销榜等），而这些服务在异常情况下直接不获取，即降级即可。

6.6、读降级

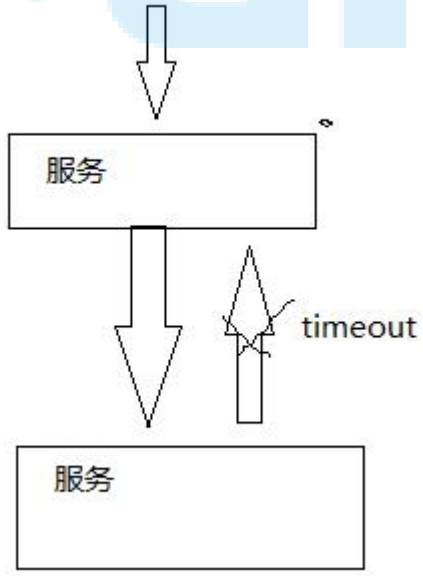
比如，多级缓存模式，如果后端服务有问题，则可以降级为只读缓存，这种方式适用于对读一致性要求不高的场景。

6.7、写降级

比如，秒杀抢购，我们可以只进行 Cache 的更新，然后异步扣减库存到 DB，保证最终一致性即可，此时可以将 DB 降级为 Cache。

自动降级

（1） 超时降级



（2） 服务失败频率较高

统计服务出现错误次数，当出现错误次数达到阀值（99.99%）,对服务进行降级，发出警告。

（3）故障降级

网络故障

服务抛出异常

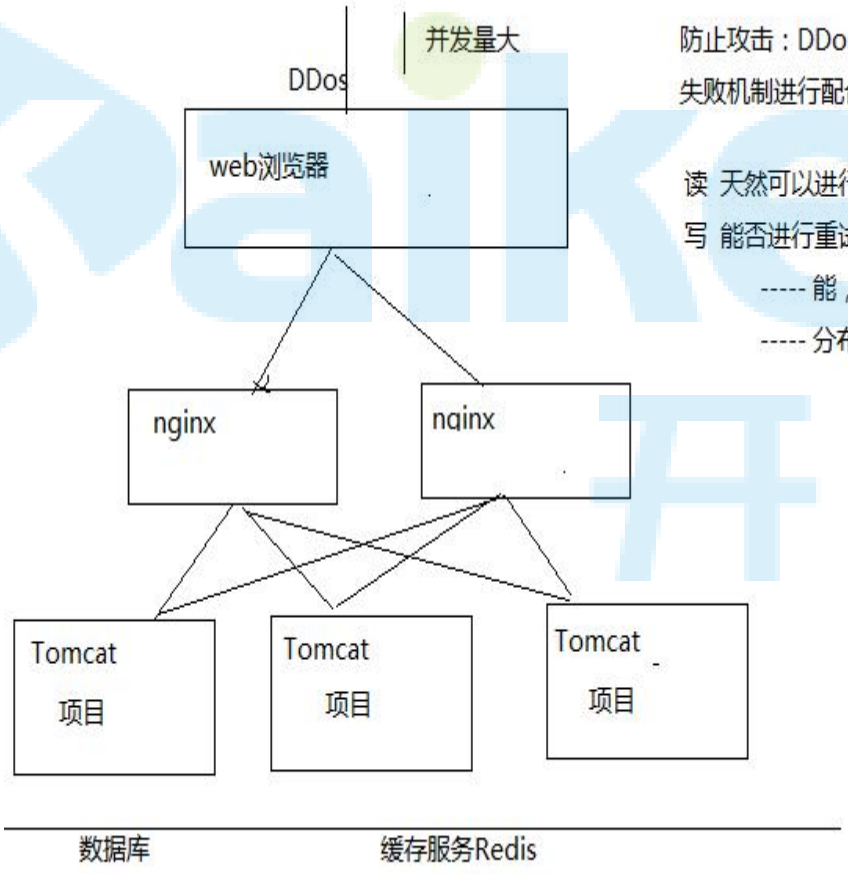
人工降级

读降级

写降级

页面降级

7、超时与重试



防止攻击：DDos攻击，设置合理超时重试机制、要和服务熔断机
失败机制进行配合，防止攻击。造成整个系统宕机。

读 天然可以进行重新

写 能否进行重试？？？？写订单--提交订单 退款 -- 多次重试

----- 能，但是必须保证接口的幂等性

----- 分布式锁 详细讲接口幂等性设计

（1）代理层超时与重试： nginx
（2）web 容器超时与重试
（3）中间件和服务之间超时与重试
（4）数据库连接超时与重试
（5）nosql 超时与重试
（6）业务超时与重试

（7）前端浏览器 ajax 请求超时与重试

退款： 按钮 – 手抖，多点击几次---------- 接口 幂等性

8、压测与预案

8.1、系统压测

压测一般指性能压力测试，用来评估系统的稳定性和性能，通过压测数据进行系统容量评估，从而决定是否需要进行扩容或缩容。

压测之前要有压测方案〔如压测接口、并发量、压测策略（突发、逐步加压、并发量）、压测指标（机器负载、QPS/TPS、响应时间）〕，之后要产出压测报告〔压测方案、机器负载、QPS/TPS、响应时间（平均、最小、最大）、成功率、相关参数（JVM 参数、压缩参数）等〕，最后根据压测报告分析的结果进行系统优化和容灾。

（1）线下压测

通过如 JMeter、Apache ab 压测系统的某个接口（如查询库存接口）或者某个组件（如数据库连接池），然后进行调优（如调整 JVM 参数、优化代码），实现单个接口或组件的性能最优。

线下压测的环境（比如，服务器、网络、数据量等）和线上的完全不一样，仿真度不高，很难进行全链路压测，适合组件级的压测，数据只能作为参考。

（2）线上压测

线上压测的方式非常多，按读写分为读压测、写压测和混合压测，按数据仿真度分为仿真压测和引流压测，按是否给用户提供服务分为隔离集群压测和线上集群压测。

读压测是压测系统的读流量，比如，压测商品价格服务。写压测是压测系统的写流量，比如下单。写压测时，要注意把压测写的数据和真实数据分离，在压测完成后，删除压测数据。只进行读或写压测有时是不能发现系统瓶颈的，因为有时读和写是会相互影响的，因此，这种情况下要进行混合压测。

仿真压测是通过模拟请求进行系统压测，模拟请求的数据可以是使用程序构造、人工构造（如提前准备一些用户和商品），或者使用 Nginx 访问日志，如果压测

的数据量有限，则会形成请求热点。而更好的方式可以考虑引流压测，比如使用 TCPCopy 复制

8.2、系统优化和容灾

拿到压测报告后，接下来会分析报告，然后进行一些有针对性的优化，如硬件升级、系统扩容、参数调优、代码优化（如代码同步改异步）、架构优化（如加缓存、读写分离、历史数据归档）等。不要把别人的经验或案例拿来直接套在自己的场景下，一定要压测，相信压测数据而不是别人的案例。

在进行系统优化时，要进行代码走查，发现不合理的参数配置，如超时时间、降级策略、缓存时间等。在系统压测中进行慢查询排查，包括 Redis、MySQL 等，通过优化查询解决慢查询问题。

在应用系统扩容方面，可以根据去年流量、与运营业务方沟通促销力度、最近一段时间的流量来评估出是否需要进行扩容，需要扩容多少倍，比如，预计 GMV 增长 100%，那么可以考虑扩容 2~3 倍容量。

8.3、应急预案

在系统压测之后会发现一些系统瓶颈，在系统优化之后会提升系统吞吐量并降低响应时间，容灾之后的系统可用性得以保障，但还是会存在一些风险，如网络抖动、某台机器负载过高、某个服务变慢、数据库 Load 值过高等，为了防止因为这些问题而出现系统雪崩，需要针对这些情况制定应急预案，从而在出现突发情况时，有相应的措施来解决掉这些问题。

应急预案可按照如下几步进行：首先进行系统分级，然后进行全链路分析、配置监控报警，最后制定应急预案。

二、高并发业务架构设计实践

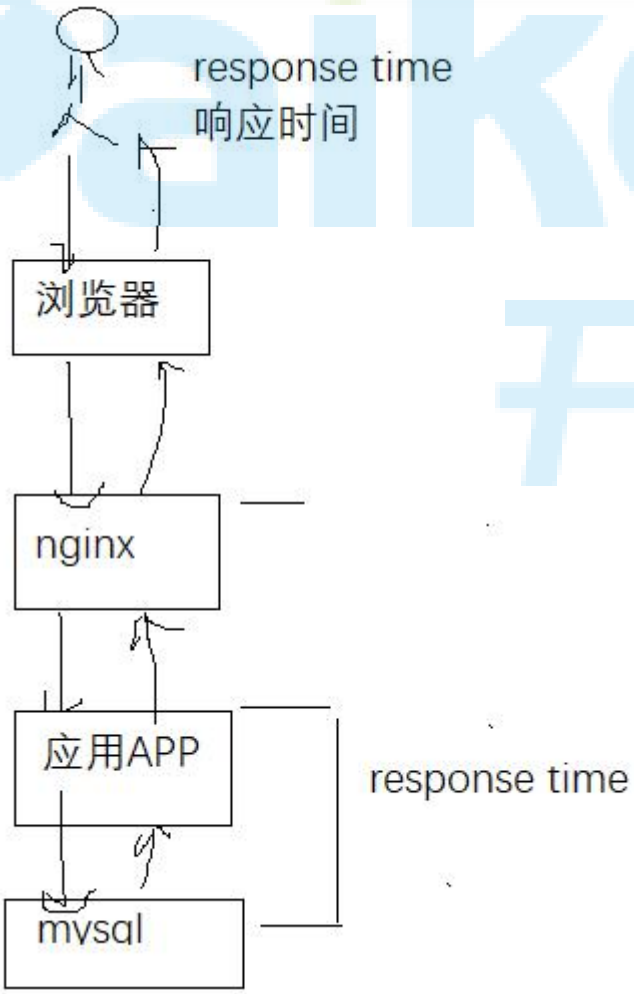
1、软件架构中的高并发思考

1.1、什么是高并发？

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。

高并发相关常用的一些指标有响应时间（Response Time），吞吐量（Throughput），每秒查询率 QPS（Query Per Second），并发用户数等。

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。



吞吐量：单位时间内处理的请求数量（并发数量）

QPS：每秒响应请求数。在互联网领域，这个指标和吞吐量区分的没有这么明显。

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。

重点：

1、QPS ： 每秒查询数量

2、TPS ： 每秒事务数量，一个接口请求从发送请求到接收到响应为止，代表一个 tps

3、吞吐量： 每秒请求数量

QPS ,TPS，吞吐量有何区别？？

解释： 在大多数情况系下，qps = tps = 吞吐量

例如： 访问/index.html (css,js,index.do) ➔ QPS = 3

⇨ TPS = 1

1.2、高并发下系统设计问题的思考

大家也许开发过高并发的系统或者一些高并发类似秒杀程序，但肯定都有接触过，像电商平台的秒杀、抢购等活动，还有 12306 春运抢票。

互联网公司，做一些有奖活动，而且数量有限，奖品给力，如果是先到先得的策略，那就类似秒杀系统了。

这类系统最大的问题就是活动周期短，瞬间流量大（高并发），很少人可以成功下单，绝大多数人都是很遗憾。所以从运营体验上，没有成功下单的人，心里肯定是不好受的，如果这时候，因为技术、网络问题，影响用户体验，那就更是骂声一片。

下面是基本的概念的建立。

第一：高并发

技术要做的事，一方面优化程序，让程序性能最优，单次请求时间能从 50ms 优化到 25ms，那就可以在一秒钟内成功响应翻倍的请求了。

另一方面就是增加服务器，用更大的集群来处理用户请求，设计好一个可靠且灵活扩充的分布式方案就更加重要了。

1、加机器

2、高性能机器

3、架构、性能优化

第二：时间短(Throughput)

火热的秒杀活动，真的是一秒钟以内就会把商品抢购一空，而大部分用户的感受是，提交订单的过程却要等待好几秒、甚至十几秒，更糟糕的当然是请求报错。

那么一个好的秒杀体验，当然希望尽可能减少用户等待时间，准确的提示用户当前是否还有商品库存。而这些，也是需要有优秀的程序设计来保证的。

第三：系统容量预估(Throughput) – 硬件层面

系统设计的时候，都需要有一个容量预估，那就是要提前计算好，我们设计的系统，要承载多大的数量级。

例如：

8 核 16G 内存服务器

订单处理能力：100ms

QPS = 1000ms / 100ms * 8 = 80 QPS ---- 单台服务器，非常低

优化程序： 订单请求处理耗时： 10ms

QPS = 1000ms /10 ms * 8 = 800 QPS

订单业务：判断库存----默认库存是无限的---下单时间：1ms

第四：好的分布式方案(Throughput)---应用架构

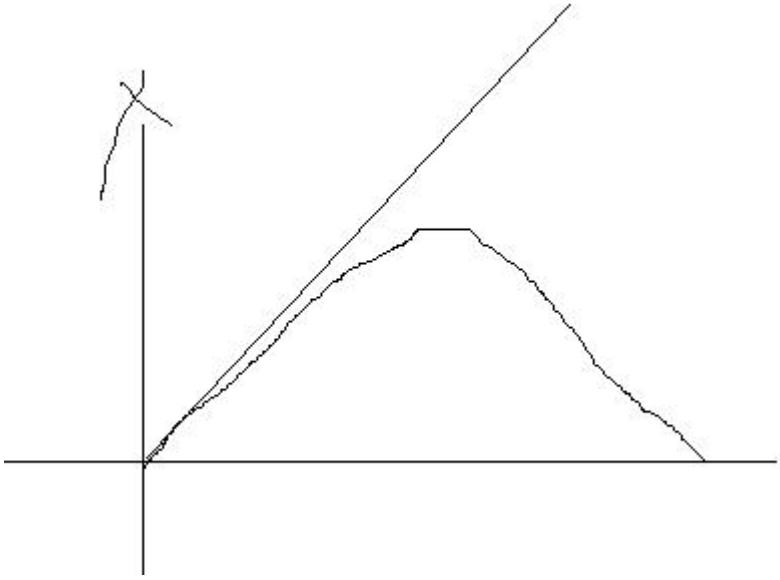
一个好的分布式方案，首先当然是稳定可靠，不要出乱子，然后就是方便扩充，最好的效果当然是增加一台服务器，并发处理量可以 1:1 线性增长。

例如：

单机 QPS : 1000

10 台服务器： 1w

100 台服务器： 10w/s



由于 nginx 出现性能瓶颈，造成吞吐量下降。

第五：关注系统的瓶颈

大家先有几个基本的共识，系统的处理速度

程序内数据读写 > redis > mysql > 磁盘

单机网络请求 > 局域网内请求 > 跨机房请求

我们优化程序的时候，尽量用最快的方式，尽量用最简短的逻辑。

用 redis 替代 mysql 来保存订单处理中依赖的数据，用程序中的提交的数据代替从 redis 中二次获取数据，比如：商品库存信息，用户订单信息。

逻辑处理中，把速度快且提前中断的逻辑放在最前面，比如：验证登录，验证问答。

我们做分布式方案的时候，尽量把资源调用放在最近的地方。

前端服务器依赖的数据尽量就在局域网内，如果能在单机都有读的 redis 服务当然更好，程序维护数据响应会复杂些。

不要出现跨机房网络请求，不要出现跨机房网络请求，不要出现跨机房网络请求，重要的事情说三遍。

2.2、如何保障系统的高并发？

1、服务尽量进行拆分部署（分布式：SOA,微服务）; --- 业务纵向拆分，化整为零，资源拆分，横向扩展

2、尽量将请求拦截在系统上游（越上游越好）; --- 限流，缓存

3、读多写少多使用缓存（缓存抗读压力）; --- 读缓存，写异步

4、浏览器和 APP：做限速（漏桶原理） --- 限速，限流

5、站点层：按照 uid 做限速，做页面缓存

6、服务层：按照业务做写请求队列控制流量，做数据缓存 – 队列缓冲

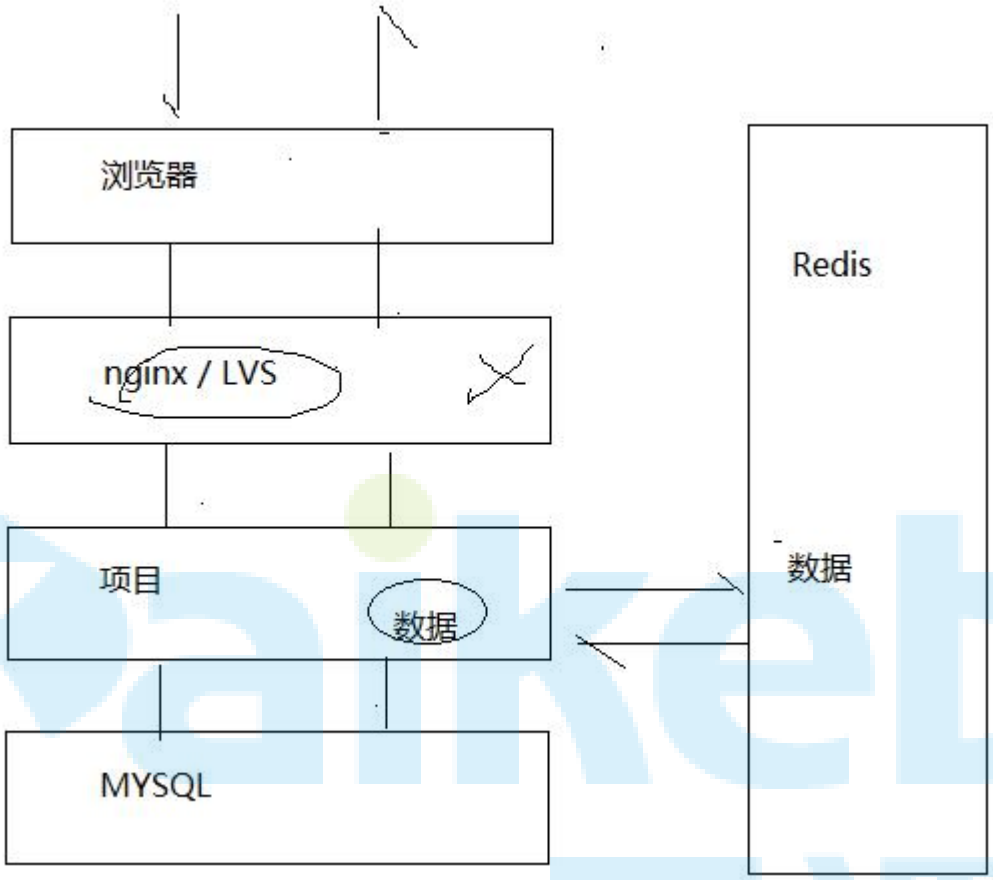
7、数据层：压力就小了，无忧无虑 —数据库优化

其他方面：结合业务做优化

具体的解决方案：
1、缓存（应用级别缓存、http 缓存、多级缓存）
2、连接池
3、异步

4、扩容
5、队列

项目简单逻辑分层图示结构：优化策略具体可以分为 数据层优化，应用层优化，前端优化



数据层优化：
数据库优化：--- RT --- 吞吐量
1、集群（分表、分库、读写分离【解决读压力】）
2、索引
3、开启缓存
4、SQL 优化
5、冗余设计
6、防止写复杂 SQL
7、冷热数据分离
分布式文件系统：
1、开源文件系统 FastDFS
2、云服务
日志数据/搜索数据/简单业务数据：
elasticSearch ms
Redis 缓存

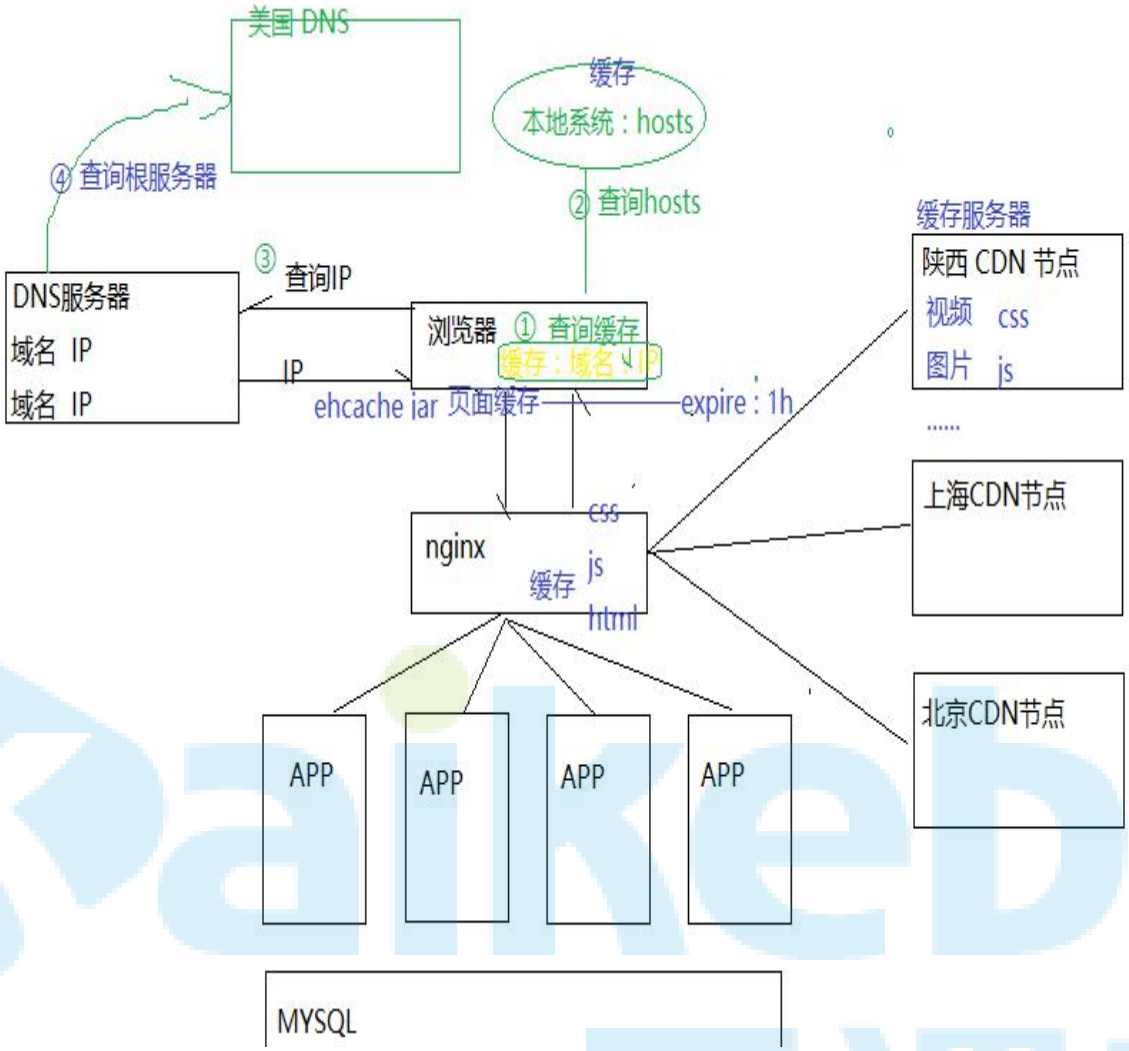
应用层优化：

1、web 服务器优化 （线程池，连接队列）
2、JVM 优化
3、代码结构优化（code review）
4、分布式拆分（提高吞吐量，提高集群部署网络数量）
5、异步架构
6、异步并发编程
7、队列
8、线程池
9、nosql

前端优化工作：
1、DNS 缓存
2、CDN 缓存
3、浏览器缓存
4、nginx 缓存

2、多级缓存应用

http 缓存：
1、DNS 缓存
2、CDN 缓存
3、浏览器缓存
4、nginx 缓存



应用（application program）级别缓存：代码级别缓存

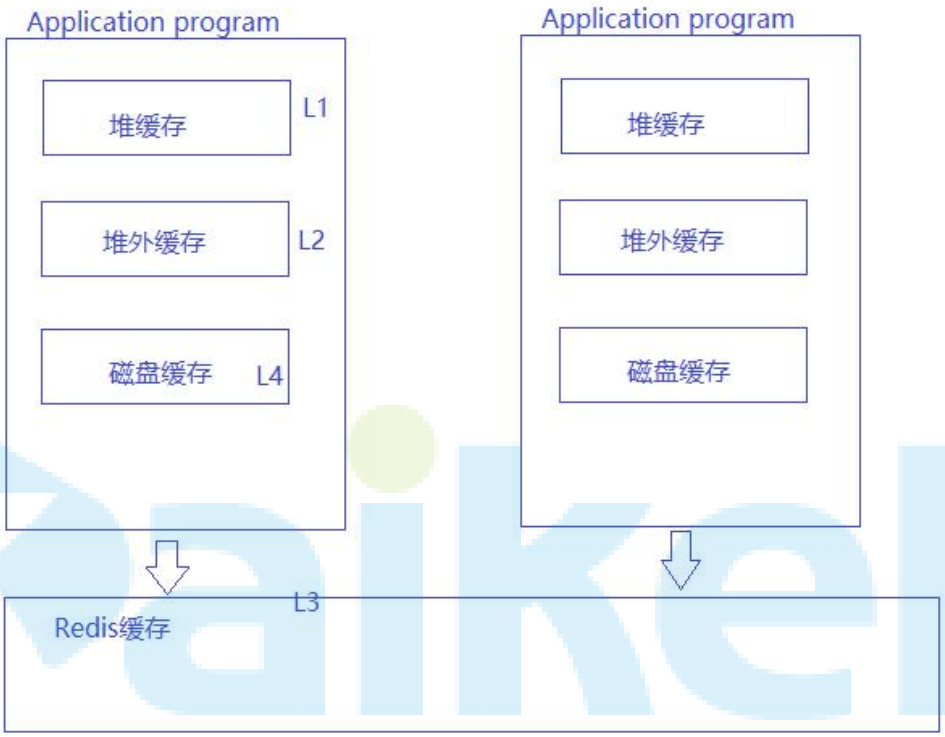
所谓缓存：让数据离用户更近，访问速度更快，RT 时间更短，吞吐量更高。

例如：
CPU 架构（三级缓存）： L1(查询) / L2（查询）/ L3（查询）
Maven: 先查询本地仓库，远程仓库

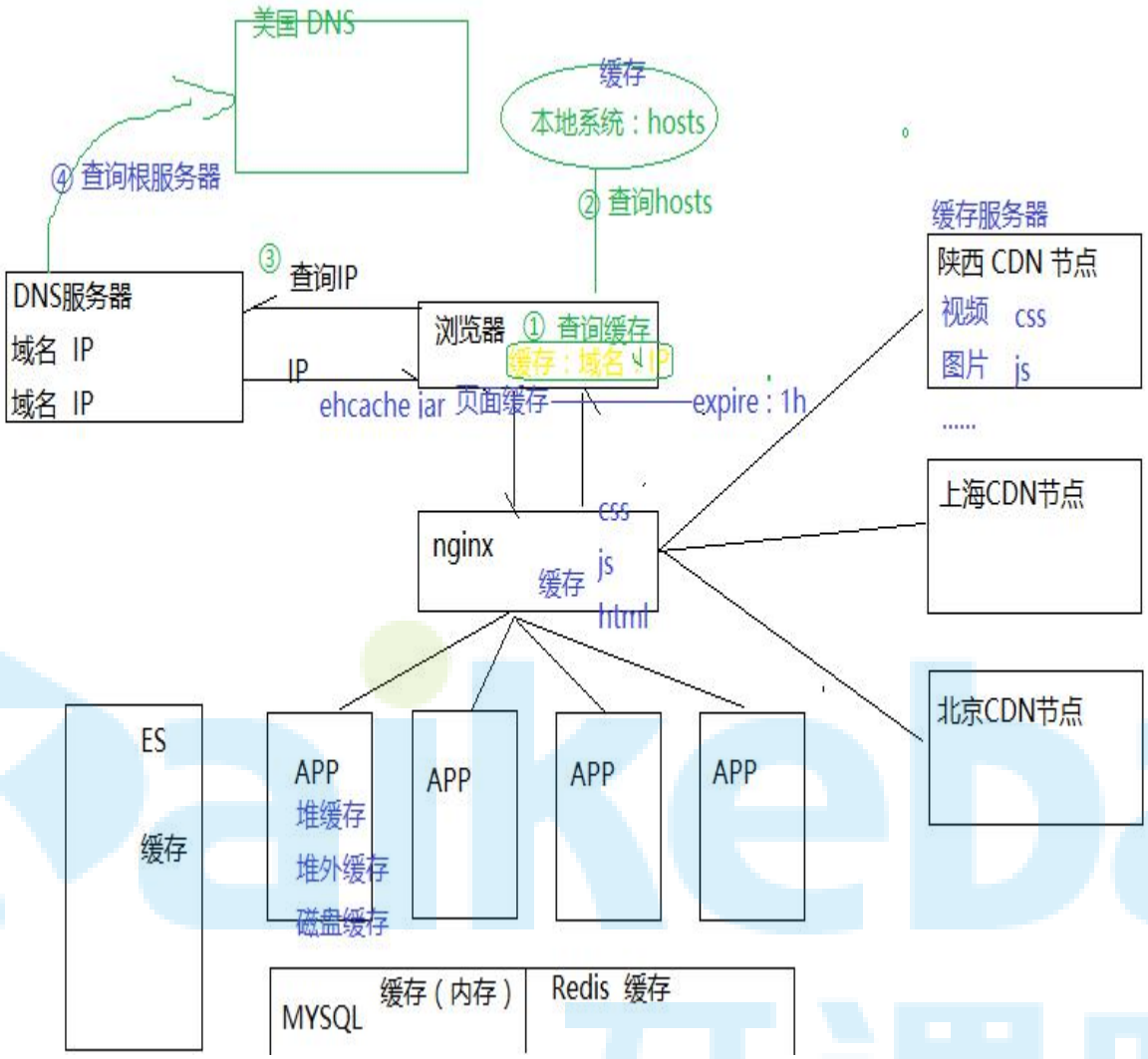
项目来说： 构造项目缓存 – mybatis：一级缓存，二级缓存
1、堆缓存
使用 Java 堆内存缓存对象。 使用技术方案： Ehcache 实现缓存。
2、内存缓存
内存缓存，缓存的大小取决于内存大小。 技术方案：Ehcache 实现缓存
3、磁盘缓存
缓存数据存储在磁盘，JVM 启动后，还能从磁盘加载。技术方案：Ehcache 实现缓存，设置过期时间
4、分布式缓存

使用第三方缓存服务器：Redis,memcached

堆缓存： 对象不需要序列化/反序列化
堆外数据： 反序列化



多级缓存（整体）：



大型项目中：构造高性能项目，一定重视缓存应用。非常重要

3、连接池详解

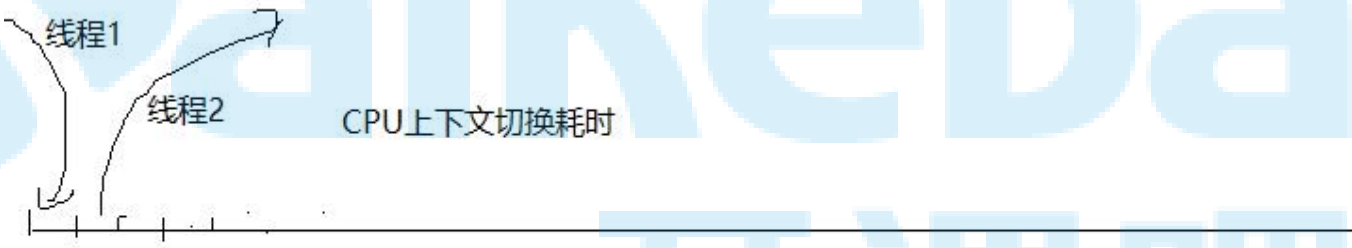
连接池目的：通过连接池减少频繁创建连接，释放连接，降低消耗，提升性能 （吞吐量）
连接池： 数据库连接，Redis 连接池，Http 连接池.....
技术方案： Apache commons pool2 , jedis, druid, dbcp.....
各位程序员： 是否知道连接池怎么配置？ 设置多少个队列，设置多少个线程？？

例如：
1w QPS 2W TPS 连接池应用设置多少？？？
回答：尽量设置大些吧，设置个 500 吧 ---- 此话是否正确？？

Oracle 数据测试数据分析：
https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing

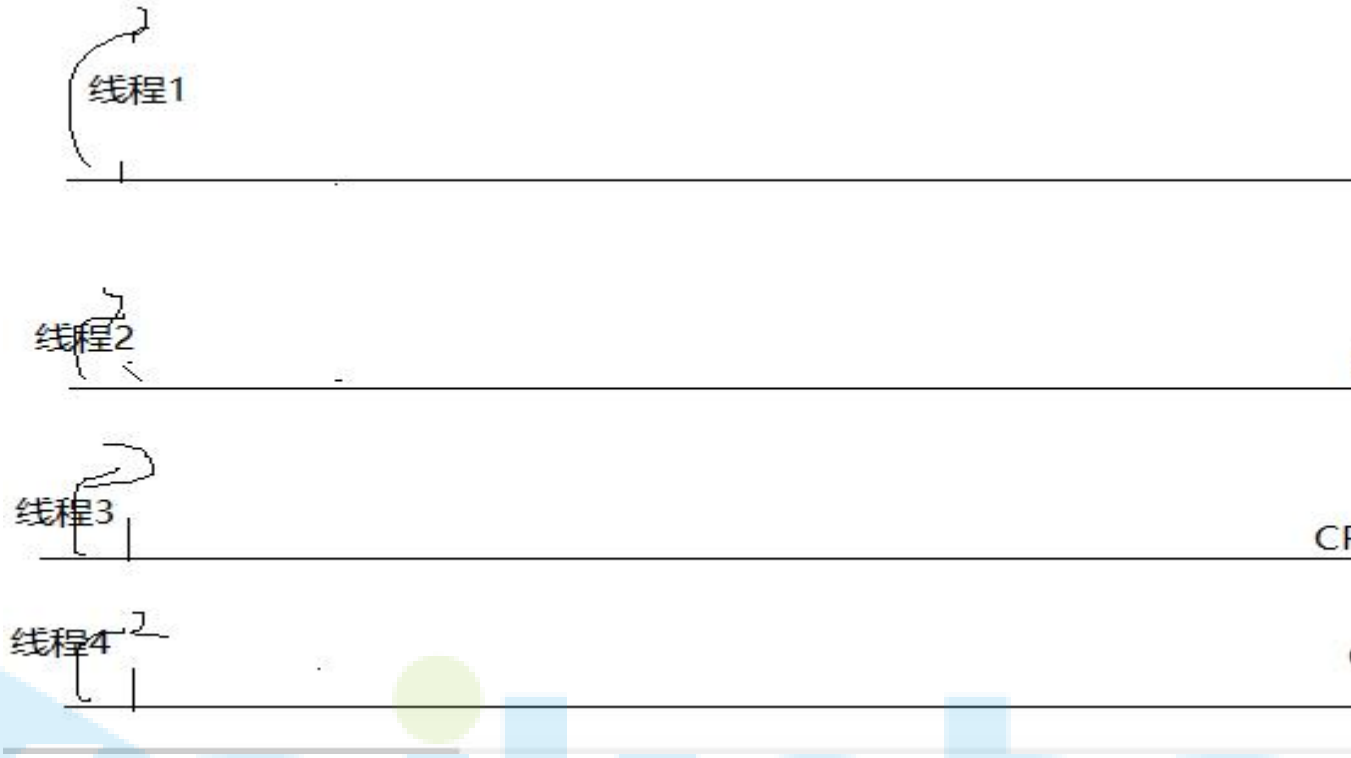
条件	线程池设置大小	每个请求在连接池队列里平均等待间
Oracle、9600并发线程、每次数据库操作sleep 550ms	2048	33ms
Oracle、9600并发线程、每次数据库操作sleep 550ms	1024	33ms
Oracle、9600并发线程、每次数据库操作sleep 550ms	96	1ms

通过 oracle 公司公布的测试数据发现：
连接池设置变小了，执行数据反而变快了，RT 时间变短了..... ，性能更好了..... 这是什么原因造成的？？？
回答： 线程池数据量太大，就会造成创建的线程过多，CPU 使用的是时间分片切换线程的使用权，多个线程的频繁切换会消耗大量的时间。



线程什么时候能执行：抢到CPU时间分片的时候，才能获得执行权限

线程池是否越少越好呢？？ 肯定不行，线程池一定要设置一个合适值。
理想状态：
4 核 CPU 机器： CPU 密集型 --- 线程池数量 == CPU 核心数量



IO 密集型： 占用大量的 IO 资源，CPU 占用比较少

线程池： 2*4 = 8

以上算法：粗略的估算方法。

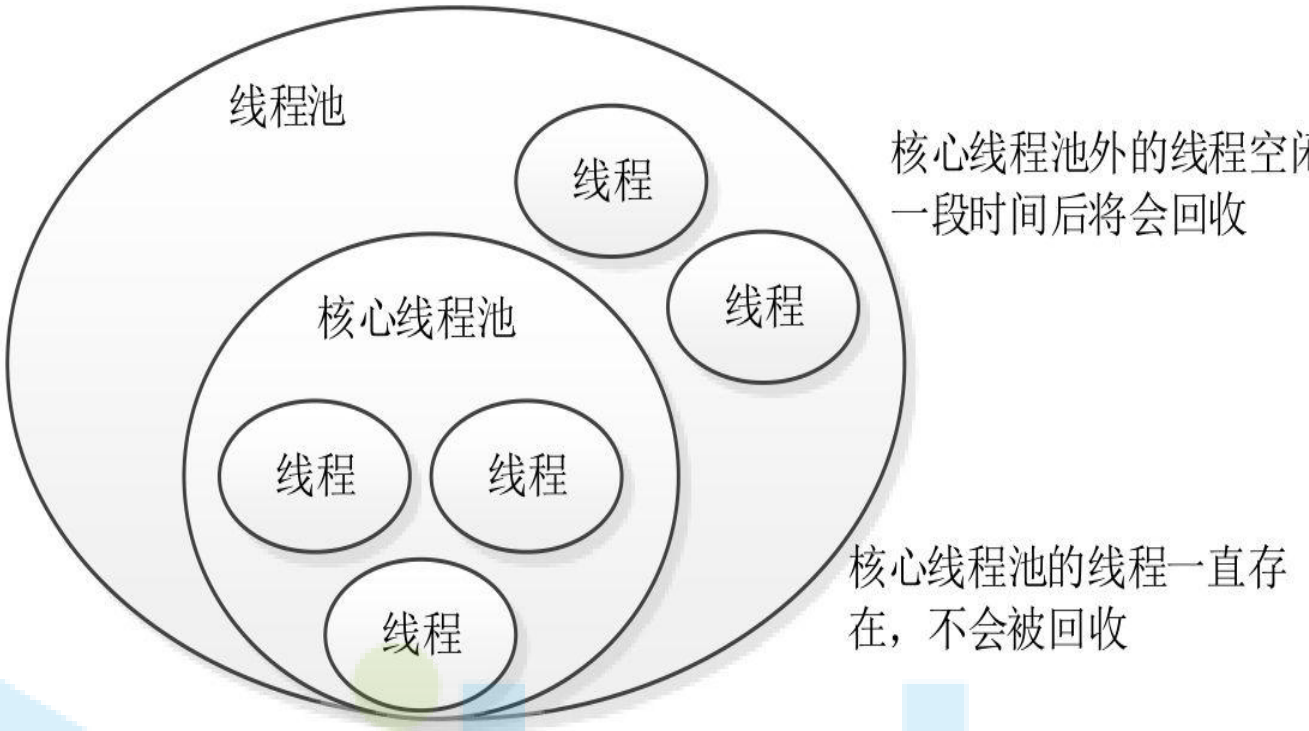
4、线程池详解

线程池的目的类似于连接池，通过减少频繁创建和销毁线程来降低性能损耗。每个线程都需要一个内存栈，用于存储如局部变量、操作栈等信息，可以通过-Xss 参数来调整每个线程栈大小，通过调整该参数可以创建更多的线程，不过 JVM 不能无限制地创建线程。

通过使用线程池可以限制创建的线程数，从而保护系统。线程池一般配合队列一起工作，使用线程池限制并发处理任务的数量。

然后设置队列的大小，当任务超过队列大小时，通过一定的拒绝策略来处理，这样可以保护系统免受大流量而导致崩溃——只是部分拒绝服务，还是有一部分是可以正常服务的。

线程池一般有核心线程池大小和线程池最大大小配置，当线程池中的线程空闲一段时间时将会被回收，而核心线程池中的线程不会被回收。



多少个线程合适呢？

建议根据实际业务情况来压测决定，或者根据利特尔法则来算出一个合理的线程池大小，其定义是，在一个稳定的系统中，长时间观察到的平均用户数量 L，等于长时间观察到的有效到达速率 λ 与平均每个用户在系统中花费的时间的乘积，即 L= λW。但实际情况是复杂的，如存在处理超时、网络抖动都会导致线程花费时间不一样。因此，还要考虑超时机制、线程隔离机制、快速失败机制等，来保护系统免遭大量请求或异常情况的冲击。

Java 提供了 ExecutorService 的三种实现。

- **ThreadPoolExecutor：** 标准线程池。
- **ScheduledThreadPoolExecutor：** 支持延迟任务的线程池。
- **ForkJoinPool：** 类似于 ThreadPoolExecutor，但是使用 work-stealing 模式，其会为线程池中的每个线程创建一个队列，从而用 work-stealing（任务窃取）算法使得线程可以从其他线程队列里窃取任务来执行。即如果自己的任务处理完成了，则可以去忙碌的工作线程那里窃取任务执行。

4.1、Java 线程池

使用 Executors 来创建线程池。

（1）创建单线程的线程池。

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
```

（2）创建固定数量的线程池。

```
ExecutorService executorService = Executors.newFixedThreadPool (10);
```

（3)创建可缓存的线程池，初始大小为 0，线程池最大大小为 Integer.MAX_VALUE

其使用 SynchronousQueue 队列，一个没有数据缓冲的阻塞队列。对其执行 put 操作后必须等待 take 操作消费该数据，反之亦然。该线程池不限制最大大小，如果线程池有空闲线程则复用，否则会创建一个新线程。如果线程池中的线程空闲 60 秒，则将被回收。该线程默认最大大小为 Integer.MAX_VALUE，请确认必要后再使用该线程池。

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

（4）支持延迟执行的线程池，其使用 DelayedWorkQueue 实现任务延迟。

```
ScheduledExecutorService scheduledExecutorService =
                        Executors.newScheduledThreadPool(10);
```

（5）work-stealing 线程池

```
ExecutorService executorService = Executors.newWorkStealingPool
(5);
```

ThreadPoolExecutor 配置。

• **corePoolSize：** 核心线程池大小，线程池维护的线程最小大小，即没有任务处理情况下，线程池可以有多个空闲线程，类似于 DBCP 中的 minIdle。

• **maximumPoolSize：** 线程池最大大小，当任务数非常多时，线程池可创建的最大线程数量。

keepAliveTime： 线程池中线程的最大空闲时间，存活时间超过该时间的线程会被回收，线程池会一直缩小到 corePoolSize 大小。

• **workQueue：** 线程池使用的任务缓冲队列，包括有界阻塞数组队列 ArrayBlockingQueue、有界/无界阻塞链表队列 LinkedBlockingQueue、优先级

阻塞队列 PriorityBlockingQueue、无缓冲区阻塞队列 SynchronousQueue。有界阻塞队列须要设置合理的队列大小。

• **threadFactory：** 创建线程的工厂，我们可以设置线程的名字、是否是后台线程。

• **rejectedExecutionHandler：** 当缓冲队列满后的拒绝策略，包括 Abort（直接抛出 RejectedExecutionException）、Discard（按照 LIFO 丢弃）、DiscardOldest（按照 LRU 丢弃）、CallsRun（主线程执行）。

4.2、Tomcat 线程池

以 Tomcat 8 为例配置如下，配置方式一；

```
<Connector port="8080" acceptCount="100" maxConnec
           minSpareThreads ="10" maxThreads="20
```

• **acceptCount：** 请求等待队列大小。当 Tomcat 没有空闲线程处理连接请求时，新来的连接请求将放入等待队列，默认为 100。当队列超过 acceptCount 后，新连接请求将被拒绝。

• **maxConnections：** Tomcat 能处理的最大并发连接数。当超过后还是会接收连接并放入等待队列（acceptCount 控制），连接会等待，不能被处理。BIO 默认是 maxThreads 数量。NIO 和 NIO2 默认是 10000，ARP 默认是 8192。

• **minSpareThreads：** 线程池最小线程数，默认为 10。该配置指定线程池可以维持的空闲线程数量。

• **maxThreads：** 线程池最大线程数，默认为 200。当线程池空闲一段时间后会释放到只保留 minSpareThreads 个线程。

举例，假设 maxThreads=100，maxConnections=50，acceptCount=50，假设并发请求为 200，则有 50 个线

程并发处理 50 个并发连接，50 个连接进入等待队列，剩余 100 个将被拒绝。也就是说 Tomcat 最大并发线程数是由 maxThreads 和 maxConnections 中最小的一个决定。BIO 场景下 maxConnections 和 maxThreads 是一样的，当我们需要长连接场景时，应使用 NIO 模式，并发连接数是大于线程数的。

第二种配置方式：

```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
        minSpareThreads="25" maxThreads="200" maxIdleTime="
        maxQueueSize= "Integer.MAX_VALUE"
        prestartminSpareThreads="false"/>
<Connector port="8080" executor="tomcatThreadPool"
        executorTerminationTimeoutMillis ="5000"/>
```

此处我们使用了 org.apache.catalina.Executor 实现，其表示一个可在多个 Connector 间共享的线程池，而且有更丰富的配置。

- **namePrefix：** 创建的 Tomcat 线程名字的前缀。
- **deamon：** 是否守护线程运行，默认为 true。
- **minSpareThreads：** 线程池最小线程数，默认为 25。
- **maxThreads：** 线程池最大线程数，默认为 200。
- **maxIdleTime：** 空闲线程池的存活时间，默认为 60s。当线程空闲超过该时间后，线程将被回收。
- **maxQueueSize：** 任务队列最大大小，默认为 Integer.MAX_VALUE，建议改小。可以认为是 maxConnections。
- **prestartminSpareThreads：** 是否在 Tomcat 启动时就创建 minSpareThreads 个线程放入线程池，默认为 false。
- **executorTerminationTimeoutMillis：** 在停止 Executor 时，等待请求处理线程终止的超时时间。

最后，要根据业务场景和压测来配置合理的线程池大小，配置太大的线程池在并发量较大的情况下会引起请求处理不过来导致响应慢，甚至造成 Tomcat 僵死。

4.3、线程数计算

<Java Concurrency In practice>

For compute-intensive tasks, an N_{cpu}-processor system usually achieves optimum utilization with a thread
+1 threads. (Even compute-intensive threads occasionally take a page fault or pause for some other re
"extra" runnable thread prevents CPU cycles from going unused when this happens.) For tasks that also in
other blocking operations, you want a larger pool, since not all of the threads will be schedulable at all time
size the pool properly, you must estimate the ratio of waiting time to compute time for your tasks; this es
not be precise and can be obtained through pro-filing or instrumentation. Alternatively, the size of the thr
be tuned by running the application using several different pool sizes under a benchmark load and observing
CPU utilization.

Given these definitions:

$$N_{cpu} = \text{number of CPUs}$$
$$U_{cpu} = \text{target CPU utilization, } 0 \le U_{cpu} \le 1$$
$$\frac{W}{C} = \text{ratio of wait time to compute time}$$

The optimal pool size for keeping the processors at the desired utilization is:

$$N_{threads} = N_{cpu} * U_{cpu} * \left(1 + \frac{W}{C}\right)$$

You can determine the number of CPUs using `Runtime`:

```
int N_CPUS = Runtime.getRuntime().availableProcessors();
```

Of course, CPU cycles are not the only resource you might want to manage using thread pools. Other resou
contribute to sizing constraints are memory, file handles, socket handles, and database connections. Calc
size constraints for these types of resources is easier: just add up how much of that resource each task
divide that into the total quantity available. The result will be an upper bound on the pool size.

When tasks require a pooled resource such as database connections, thread pool size and resource pool siz
other. If each task requires a connection, the effective size of the thread pool is limited by the connecti
Similarly, when the only consumers of connections are pool tasks, the effective size of the connection pool
the thread pool size.

公式一：
Ncpu = CPU 核心数量
Ucpu = cpu 使用率
W/C = 线程等待时间 /(等待时间+计算时间)

线程池数量 = Ncpu * Ucpu * (1 + w/c)

公式二：

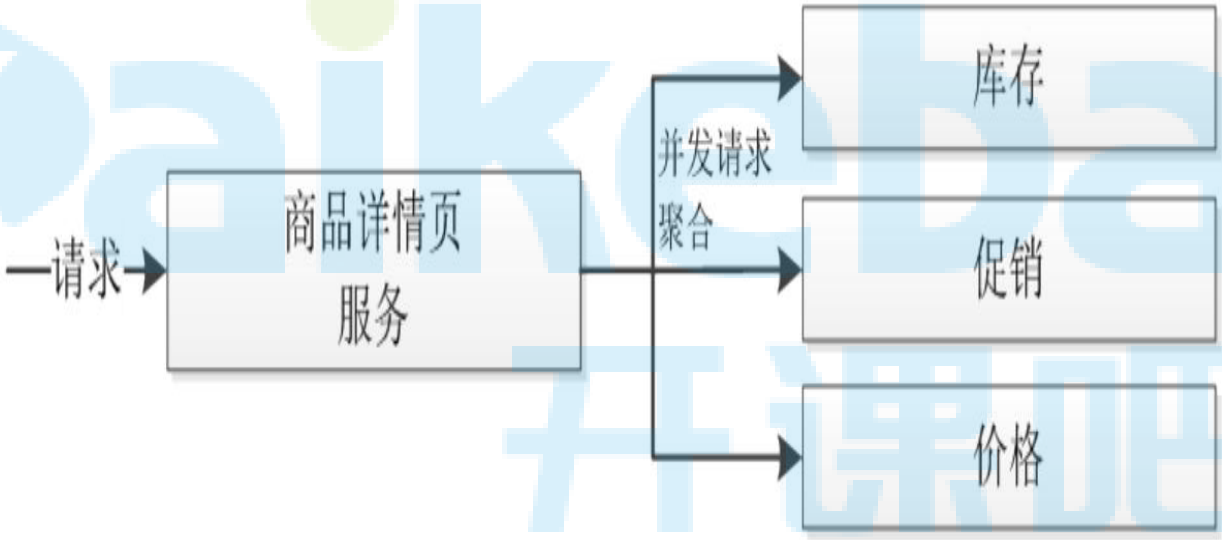
线程池数量 = Ncpu / (1 – 阻塞系数)　阻塞系数 = waittime / (waittime + cpu time)
阻塞系数： 计算密集型的任务阻塞系数是 0　　IO 密集型阻塞系数：无限制靠近 1

例如： 阻塞系数= 0.9
线程数=16 /(1-0.9) = 160 线程

经验值设置：
Cpu 密集型： threads = N + 1
Io 密集型： theads = 2*N

5、异步并发

在应用中一个服务可能会调用多个依赖服务来处理业务，而这些依赖服务是可以同时调用的。如果顺序调用的话需要耗时 100ms，而并发调用只需要 50ms，那么可以使用 Java 并发机制来并发调用依赖服务，从而降低该服务的响应时间。



在开发应用系统过程中，通过异步并发并不能使响应变得更快，更多是为了提升吞吐量、对请求更细粒度控制，或是通过多依赖服务并发调用降低服务响应时间。当一个线程在处理任务时，通过 Fork 多个线程来处理任务并等待这些线程的处理结果，这种应用并不是真正的异步。

异步是针对 CPU 和 IO 的，当 IO 没有就绪时要让出 CPU 来处理其他任务，这才是异步。

5.1、异步 Future

线程池配合 Future 实现，但是阻塞主请求线程，高并发时依然会造成线程数过多、CPU 上下文切换。

通过 Future 可以并发发出 *N* 个请求，然后等待最慢的一个返回，总响应时间为最慢的

一个请求返回的用时。如下请求如果并发访问，则响应可以在 30ms 后返回。

实际代码为：

```
public class Test {
    final static ExecutorService executor =
                                        Executors.newFixedThreadPool(2);
    public static void main(String[] args) {
        RpcService rpcService = new RpcService();
        HttpService httpService = new HttpService();
        Future<Map<String, String>> future1 = null;
        Future<Integer> future2 = null;
        try {
            future1 = executor.submit(() -> rpcService.getRpcResult());
            future2 = executor.submit(() -> httpService.getHttpResult());
            //耗时为10ms
            Map<String, String> result1 =
                                future1.get(300, TimeUnit.MILLISECONDS);
            //耗时为20ms
            Integer result2 = future2.get(300, TimeUnit.MILLISECONDS);
```

```
            //总耗时为 20ms
        } catch (Exception e) {
            if (future1 != null) {
                future1.cancel(true);
            }
            if (future2 != null) {
                future2.cancel(true);
            }
            throw new RuntimeException(e);
        }
    }
    static class RpcService {
        Map<String, String> getRpcResult() throws Exception {
            //调用远程方法（远程方法耗时约 10ms，可以使用 Thread.sleep 模拟）
        }
    }
    static class HttpService {
        Integer getHttpResult() throws Exception {
            //调用远程方法（远程方法耗时约 20ms，可以使用 Thread.sleep 模拟）
        }
    }
}
```

5.2、异步 Callback

通过回调机制实现，即首先发出网络请求，当网络返回时回调相关方法，如 HttpAsyncClient 使用基于 NIO 的异步 I/O 模型实现，它实现了 Reactor 模式，摒弃阻塞 I/O 模型 one thread per connection，采用线程池分发事件通知，从而有效支撑大量并发连接。这种机制并不能提升性能，而是为了支撑大量并发连接或者提升吞吐量。

```
public class AsyncTest {
    public static HttpAsyncClient httpAsyncClient;
    public static CompletableFuture<String> getHttpData(S
        CompletableFuture asyncFuture = new CompletableFut

        HttpAsyncRequestProducer producer =
                                HttpAsyncMethods.create(new Ht
        BasicAsyncResponseConsumer consumer =
```

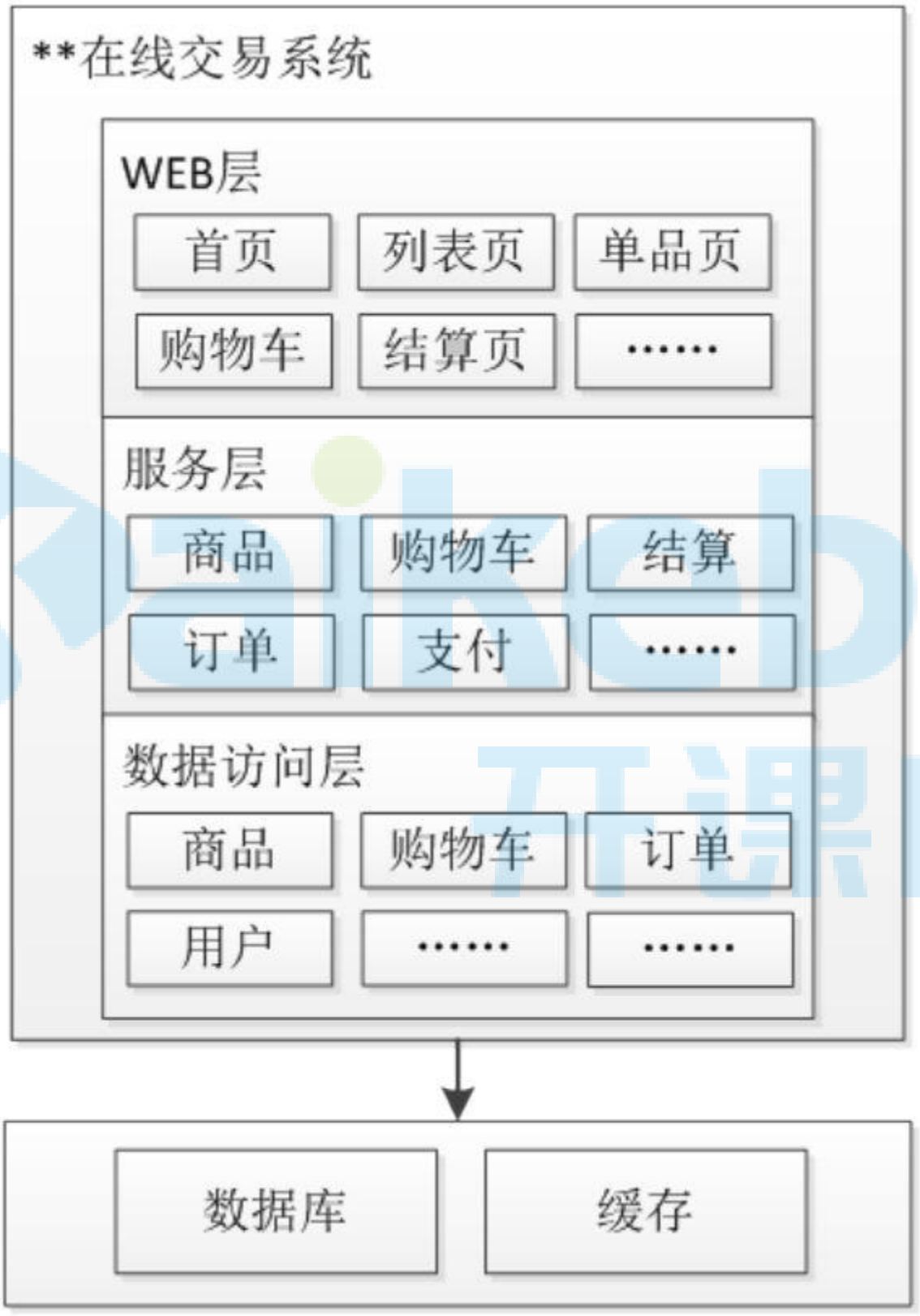
```
                                new BasicAsyncResponseConsume

        FutureCallback callback = new FutureCallback<Http
            public void completed(HttpResponse response)
                asyncFuture.complete(response);
            }
            public void failed(Exception e) {
                asyncFuture.completeExceptionally(e);
            }
            public void cancelled() {
                asyncFuture.cancel(true);
            }
        };

        httpAsyncClient.execute(producer, consumer, callb
        return asyncFuture;
    }

    public static void main(String[] args) throws Except
        CompletableFuture<String> future =
                                AsyncTest.getHttpData("http://
```

6、服务扩容


**在线交易系统
WEB层
首页
列表页
单品页
购物车
结算页
……
服务层
商品
购物车
结算
订单
支付
……
数据访问层
商品
购物车
订单
用户
……
……
数据库
缓存

对于这样一个系统，随着产品使用的用户越来越多，网站的流量会增加，最终单台服务器无法处理那么大的流量，此时就需要用分而治之的思想来解决问题。
第一步是尝试通过简单扩容来解决。
第二步，如果简单扩容搞不定，就需要水平拆分和垂直拆分数据/应用来提升系统的伸缩性，即通过扩容提升系统负载能力。
第三步，如果通过水平拆分/垂直拆分还是搞不定，那就需要根据现有系统特性，从架构层面进行重构甚至是重新设计，即推倒重来。
对于系统设计，理想的情况下应支持线性扩容和弹性扩容，即在系统瓶颈时，只需要增加机器就可以解决系统瓶颈，如降低延迟提升吞吐量，从而实现扩容需求。
如果你想扩容，则支持水平/垂直伸缩是前提。在进行拆分时，一定要清楚知道自己的目的是什么，拆分后带来的问题如何解决，拆分后如果没有得到任何收益就不要为了拆而拆，即不要过度拆分，要适合自己的业务。本章主要从应用和数据层面讲解如何按照业务和功能进行应用或数据层面的拆分

7、队列应用---RocketMQ（十万-单机）-- 提高吞吐量

队列，在数据结构中是一种线性表，从一端插入数据，然后从另一端删除数据。本书的目的不是讲解各种队列及如何实现，而是讲述在应用层面使用队列能解决哪些场景问题。

在我们的系统中，不是所有的处理都必须实时处理，不是所有的请求都必须实时反馈结果给用户，不是所有的请求都必须 100%一次性处理成功，不知道哪个系统依赖“我”来实现其业务处理，保证最终一致性，不需要强一致性。此时，我们应该考虑使用队列来解决这些问题。当然我们也要考虑是否需要保证消息处理的有序性及如何保证，是否能重复消费及如何保证重复消费的幂等性。在实际开发时，我们经常使用队列进行异步处理、系统解耦、数据同步、流量削峰、扩展性、缓冲等。

应用场景

• **异步处理** ：使用队列的一个主要原因是进行异步处理，比如，用户注册成功后，需要发送注册成功邮件/新用户积分/优惠券等；缓存过期时，先返回过期数据，然后异步更新缓存、异步写日志等。通过异步处理，可以提升主流程响应速度，而非主流程/非重要处理可以集中处理，这样还可以将任务聚合批量处理。因此，可以使用消息队列/任务队列来进行异步处理。

• **系统解耦** ：比如，用户成功支付完成订单后，需要通知生产配货系统、发票系统、库存系统、推荐系统、搜索系统等进行业务处理，而未来需要支持哪些业务是不知道的，并且这些业务不需要实时处理、不需要强一致，只需要保证最终一致性即可，因此，可以通过消息队列/任务队列进行系统解耦。

• **数据同步** ：比如，想把 MySQL 变更的数据同步到 Redis，或者将 MySQL 的数据同步到 Mongodb，或者让机房之间的数据同步，或者主从数据同步等，此时可以考虑使用 databus、canal、otter 等。使用数据总线队列进行数据同步的好处是可以保证数据修改的有序性。

• **流量削峰** ：系统瓶颈一般在数据库上，比如扣减库存、下单等。此时可以考虑使用队列将变更请求暂时放入队列，通过缓存+队列暂存的方式将数据库流量削峰。同样，对于秒杀系统，下单服务会是该系统的瓶颈，此时，可以使用队列进行排队和限流，从而保护下单服务，通过队列暂存或者队列限流进行流量削峰。

队列的应用场景非常多，以上只列举了一些常见用法和场景。

7.1、缓冲队列

使用缓冲队列应对突发流量时，并不能使处理速度变快，而是使处理速度变平滑，从而不会因瞬间压力太大而压垮应用。

通过缓冲区队列可以实现批量处理、异步处理和平滑流量。

7.2、任务队列

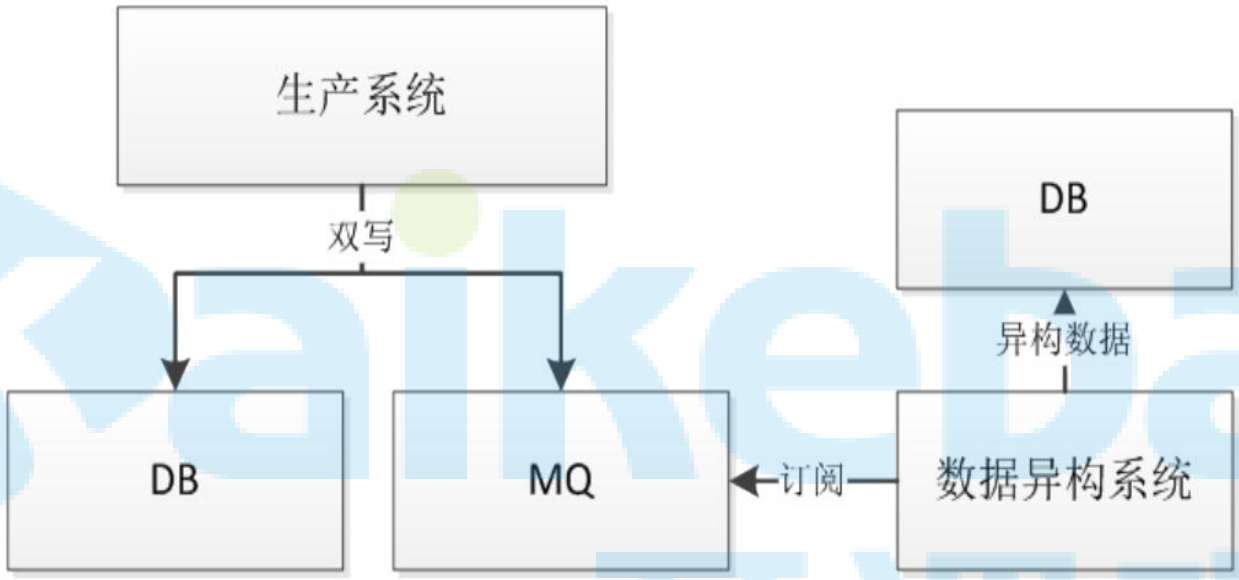
使用任务队列可以将一些不需要与主线程同步执行的任务扔到任务队列进行异步处理。笔者用得最多的是线程池任务队列（默认为 LinkedBlockingQueue）和 Disruptor 任务队列（RingBuffer）。

如用户注册完成后，将发送邮件/送积分/送优惠券任务扔到任务队列进行异步处理；刷数据时，将任务扔到队列异步处理，处理成功后再异步通知用户，处理成功后异步通知用户。以及查询聚合时，将多个可并行处理的任务扔到队列，然后等待最慢的一个任务返回。

7.3、消息队列

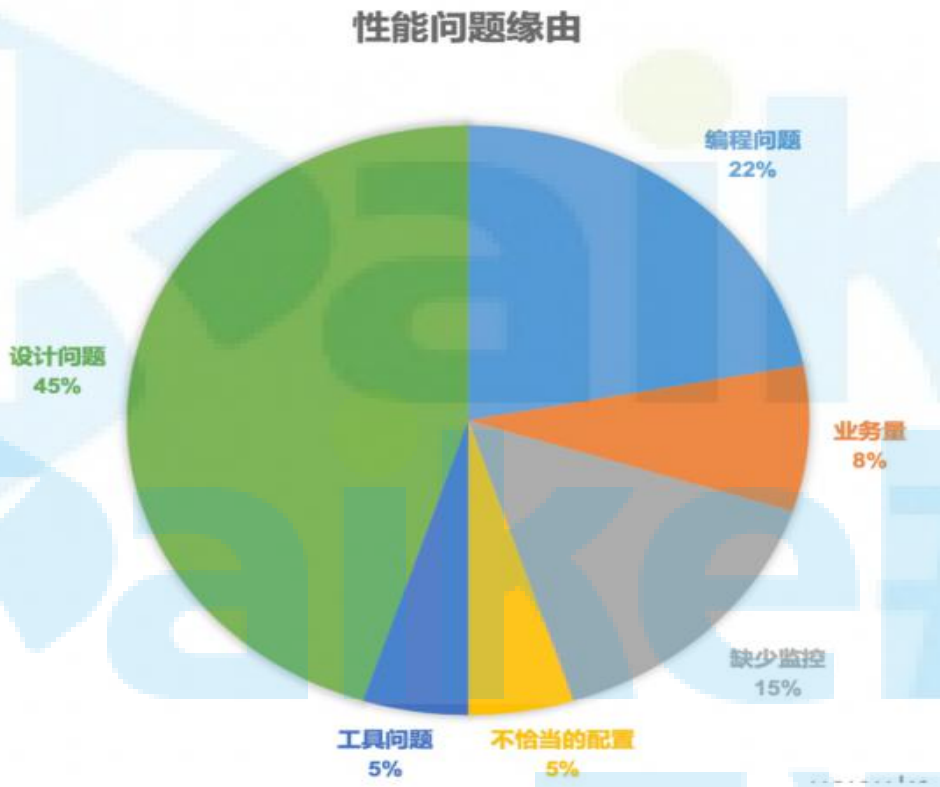
使用消息队列存储各业务数据，其他系统根据需要订阅即可比如，修改商品数据、变更订单状态时，都应该将变更信息发送到消息队列，如果其他系统有需要，则直接订阅该消息队列即可。

一般我们会在应用系统中采用双写模式，同时写 DB 和 MQ，然后异构系统可以订阅 MQ 进行业务处理（见下图）。因为在双写模式下没有事务保证，所以会出现数据不一致的情况，如果对一致性要求没那么严格，则这种模式是没问题的，而且在实际应用中这种模式也非常多。



三、系统性能优化实践

1、性能问题的来源



人祸>>

解决问题的思维定势：

非业务高峰，瞬时峰值已87%CPU，是否可以优先考虑给 机器增加服务器。

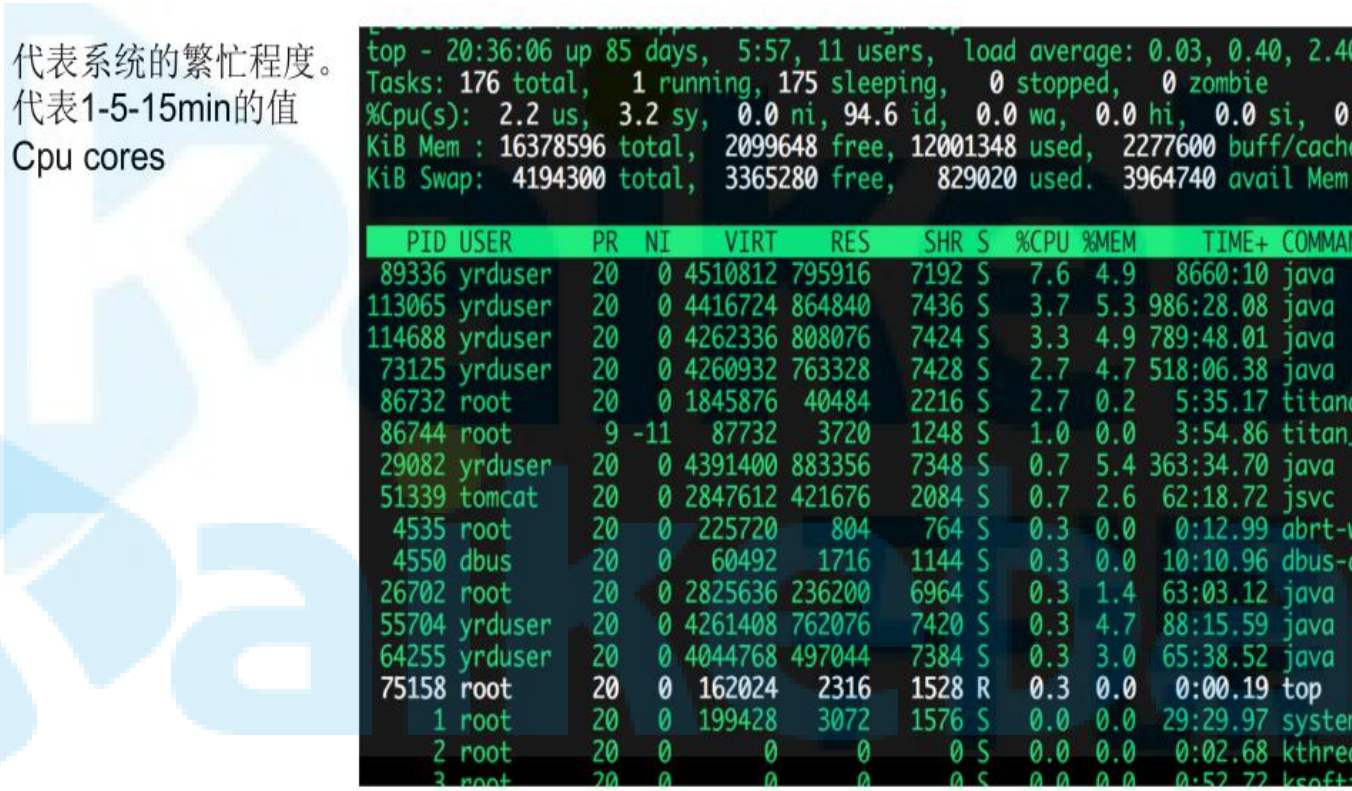
```
top - 11:17:31 up 285 days, 16:28,  1 user,  load average: 0.43, 0.38, 0.40
Tasks: 111 total,   1 running, 109 sleeping,   1 stopped,   0 zombie
%Cpu(s): 19.2 us,  2.7 sy,  0.0 ni, 61.6 id,  0.0 wa,  0.0 hi,  0.0 si, 16.4 st
KiB Mem :  8173708 total,   133660 free,  4383332 used,  3656716 buff/cache
KiB Swap: 16777212 total, 15533820 free,  1243392 used.  3422736 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
18395 r   t     20   0  894016  52852   5192 S  87.5  0.6  36357:15 filebeat
25862 j   user  20   0 8109644   3.9g   6644 S  37.5 50.3   4462:50 java
31907 root       9 -11   87728   3652   1
    1 root      20   0  190972   2816   1
    2 root      20   0       0      0
    3 root      20   0       0      0
    5 root       0 -20       0      0
    7 root      rt   0       0      0
    8 root      20   0       0      0
    9 root      20   0       0      0
   10 root       0 -20       0      0
```

业务已在4月中旬发出预警，受资源限制，一直未分配资源，目前风险已经相当高了，请尽快帮忙处理。

2、常见的性能分析方法

2.1、Load average



单核 cpu:

Load average < 1 , cpu 比较空闲，没有出现线程等待 cpu 执行现象；

Load average = 1 , cpu 刚刚占满，没有空闲空间；

Load average > 1 , cpu 已经出现了线程等待，比较繁忙

Load average > 3 , cpu 阻塞非常严重，出现了严重线程等待

4 核心 cpu:

Load average < 4 , cpu 比较空闲，没有出现线程等待 cpu 执行现象；

Load average = 4 , cpu 刚刚占满，没有空闲空间；

Load average > 4 , cpu 已经出现了线程等待，比较繁忙

Load average > 9 , cpu 阻塞非常严重，出现了严重线程等待

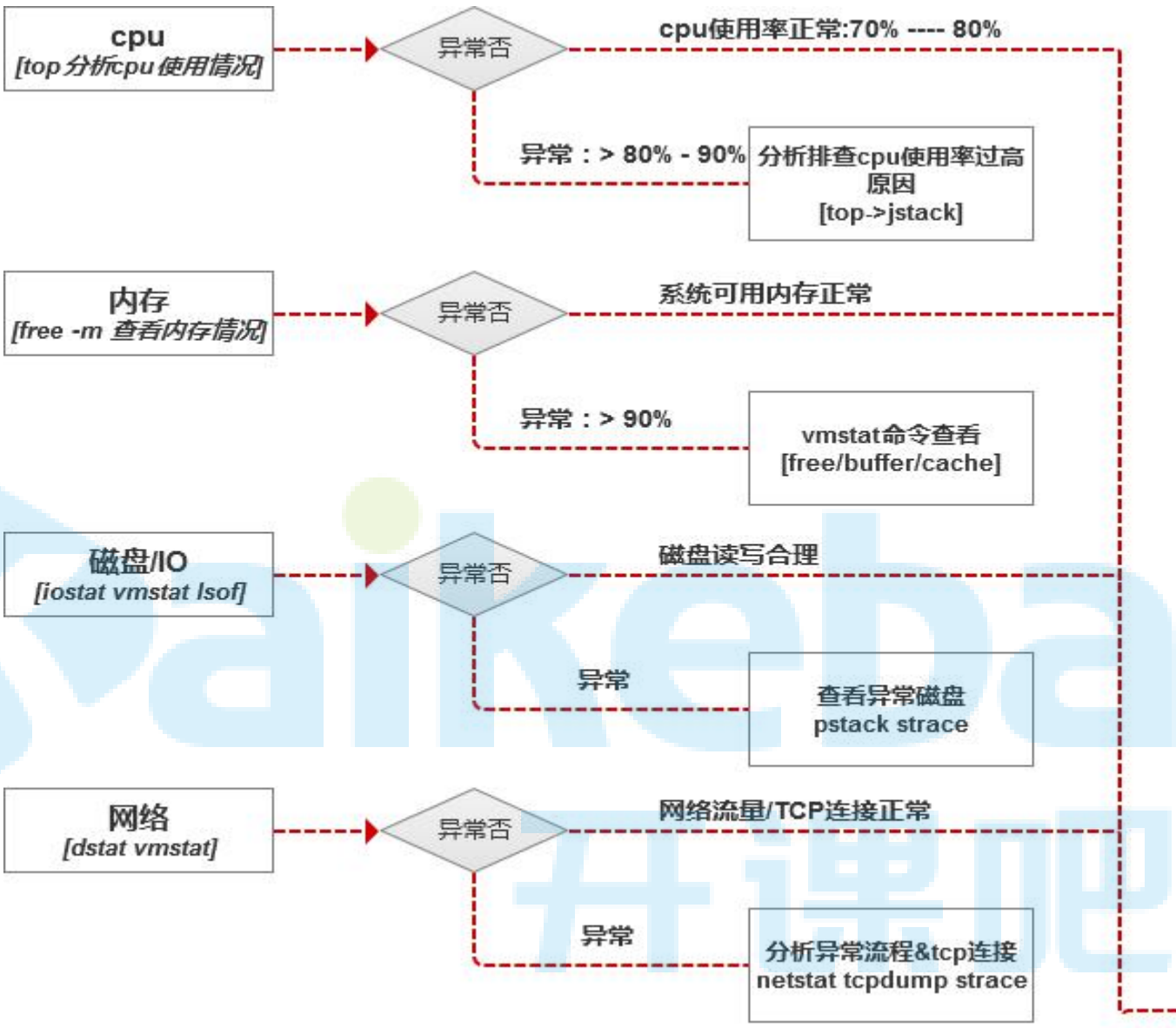
2.2、60s 快速排错

服务上线运行后：

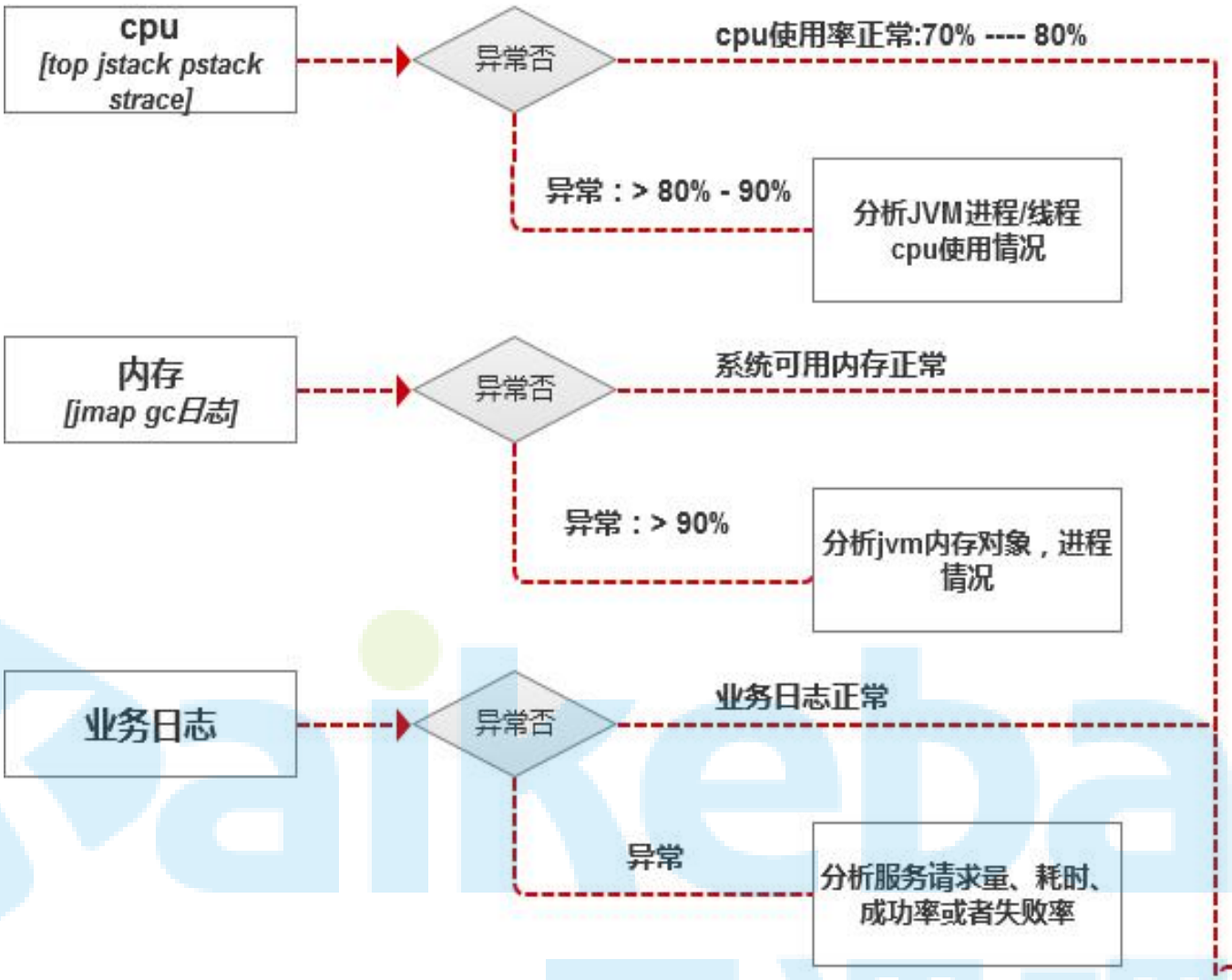
1、系统异常（磁盘满了，cpu 持续飙高.....）

2、业务异常（bug,fullgc,gc）

系统异常：


cpu
[top 分析cpu 使用情况]
异常否
cpu使用率正常:70% ---- 80%
异常：> 80% - 90%
分析排查cpu使用率过高原因
[top->jstack]
内存
[free -m 查看内存情况]
异常否
系统可用内存正常
异常：> 90%
vmstat命令查看
[free/buffer/cache]
磁盘/IO
[iostat vmstat lsof]
异常否
磁盘读写合理
异常
查看异常磁盘
pstack strace
网络
[dstat vmstat]
异常否
网络流量/TCP连接正常
异常
分析异常流程&tcp连接
netstat tcpdump strace

业务异常：



3、实现高性能系统考虑的问题

实现高并发需要考虑：

（1）系统的架构设计，如何在架构层面减少不必要的处理（网络请求，数据库操作等）例如：使用 Cache 来减少 IO 次数，使用异步来增加单服务吞吐量，使用无锁数据结构来减少响应时间；

（2）网络拓扑优化减少网络请求时间、如何设计拓扑结构，分布式如何实现？

分布式，微服务，servicemesh ,serverless

（3）系统代码级别的代码优化，使用什么设计模式来进行工作？哪些类需要使用单例，哪些需要尽量减少 new 操作？

Gc – stw(整个程序停止，进行 gc)

（4）提高代码层面的运行效率、如何选取合适的数据结构进行数据存取？如何设计合适的算法？

（5）任务执行方式级别的同异步操作，在哪里使用同步，哪里使用异步？

读缓存，写异步

（6）JVM 调优，如何设置 Heap、Stack、Eden 的大小，如何选择 GC 策略,控制 Full GC 的频率？

响应时间优先，并发优先

（7）服务端调优（线程池，等待队列）

（8）数据库优化减少查询修改时间。数据库的选取？数据库引擎的选取？数据库表结构的设计？数据库索引、触发器等设计？是否使用读写分离？还是需要考虑使用数据仓库？

连接池优化，MySQL 查询语句优化，表结构设计优化；架构优化（分表分库）

（9）缓存数据库的使用，如何选择缓存数据库？是 Redis 还是 Memcache？如何设计缓存机制？

（10）数据通信问题，如何选择通信方式？是使用 TCP 还是 UDP，是使用长连接还是短连接？NIO 还是 BIO？netty、mina 还是原生 socket？

内网： tcp

连接： 长连接，保证连接的复用性，提高性能

（11）操作系统选取，是使用 winserver 还是 Linux？或者 Unix？

（12）硬件配置？是 8G 内存还是 32G，网卡 10G 还是 1G？例如：增加 CPU 核数如 32 核，升级更好的网卡如万兆，升级更好的硬盘如 SSD，扩充硬盘容量如 2T，扩充系统内存如 128G；