大型互联网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：

架构师具备什么样的能力：
1、架构设计能力 ： 30%
2、技术能力： 30%
3、管理能力： 10%
4、沟通能力： 5%
.....
思考： 每一行代码都有出现异常的可能？？？

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？

服务永远可用：

伪命题 ---- 服务器硬件会坏，操作系统崩溃，软件服务崩溃，程序异常

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	年度停机时间
基本可用性	2个9	99%	87.6小时
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。

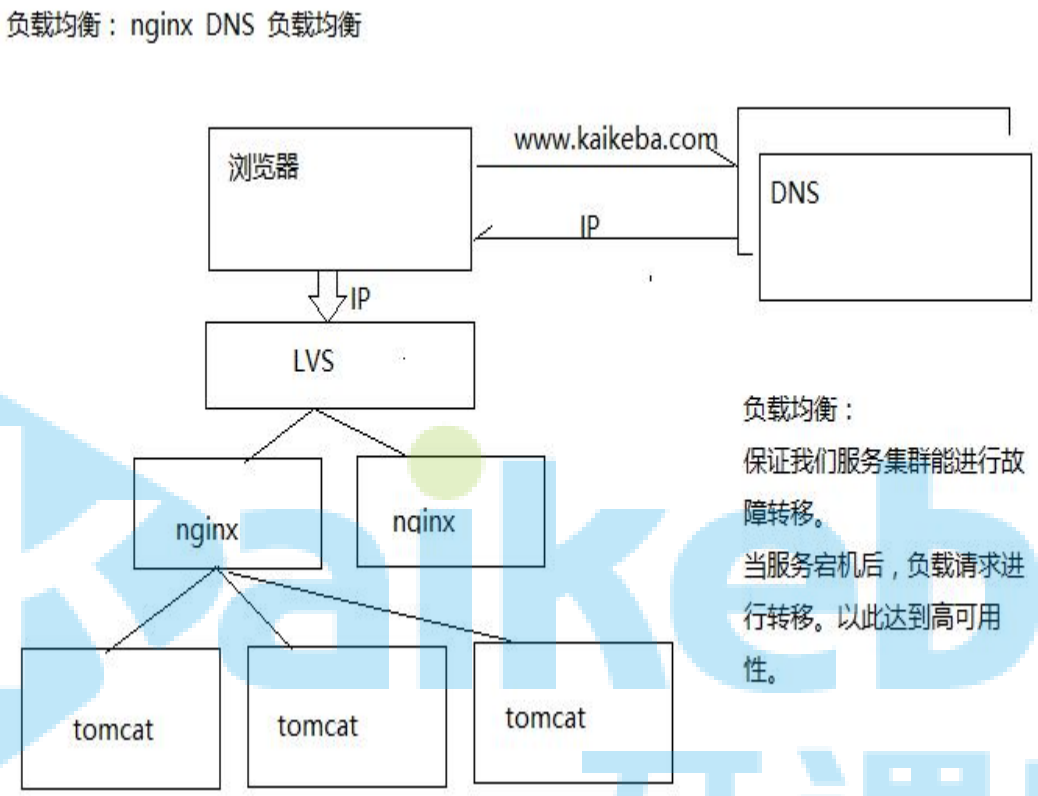
保证系统高可用，架构设计的核心准则是：冗余。有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

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



服务宕机：nginx 实现故障转移，宕机服务可以使用 k8s 方式，快速从新创建一个 pod,提高服务可用性。

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=1;
    server 192.168.61.1:9090  weight=2;
}
```

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  weight=1;
    server 192.168.61.1:9090  weight=2;
}
```

- 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=1;
    server 192.168.61.1:9090  weight=2;
}
```

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_uri;
    }
}
```

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 weight=1;
    server 192.168.61.1:9090 max_fails=2 fail_timeout=10s weight=1;
}
```

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、健康检查

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 type=tcp;
}
```

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 backup;
}
```

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  weight=1;
    server 192.168.61.1:9090  weight=2 down;
}
```

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

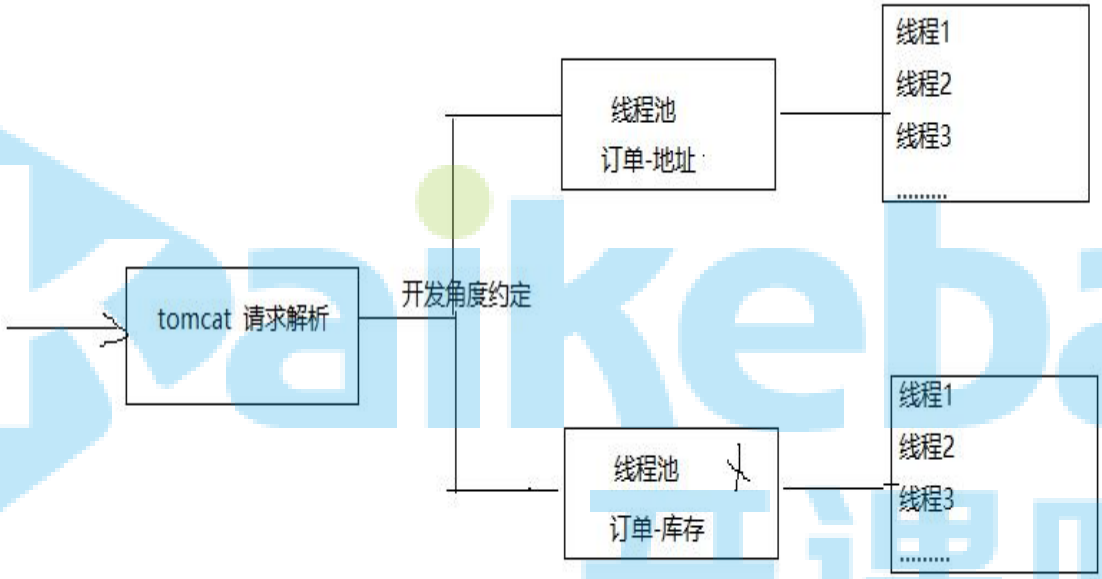
隔离是指将系统或资源分割开，系统隔离是为了在系统发生故障时，能限定传播范围和影响范围，即发生故障后不会出现滚雪球效应，从而保证只有出问题的服务不可用，其他服务还是可用的。

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。

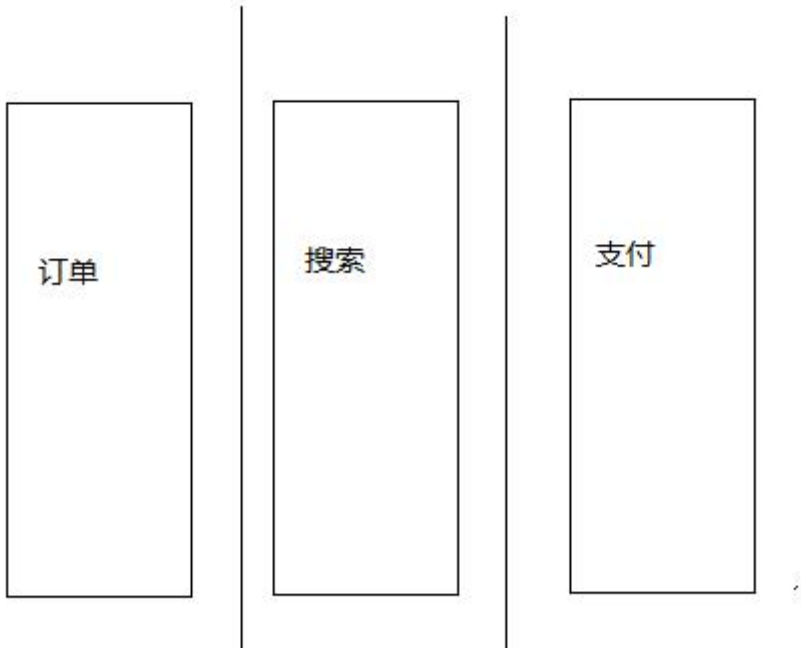
4.1、线程隔离

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

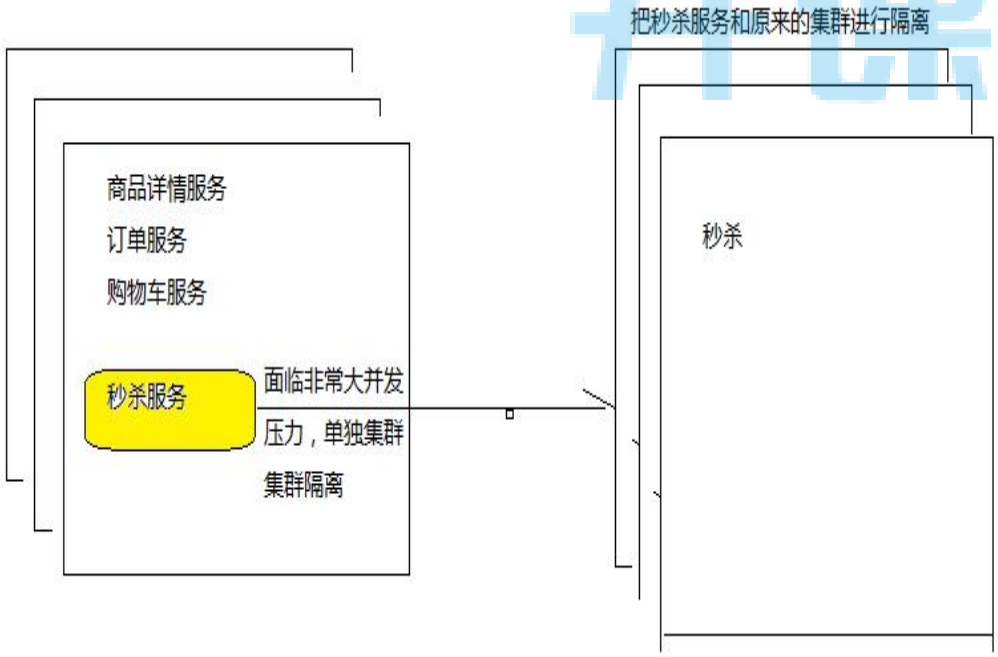
把项目拆分为一个个的子项目，然后让这些子项目进行物理隔离。项目和项目之间没有调用关系



当一个项目模块出现异常，不影响其他项目模块，从而提高项目高可用性。

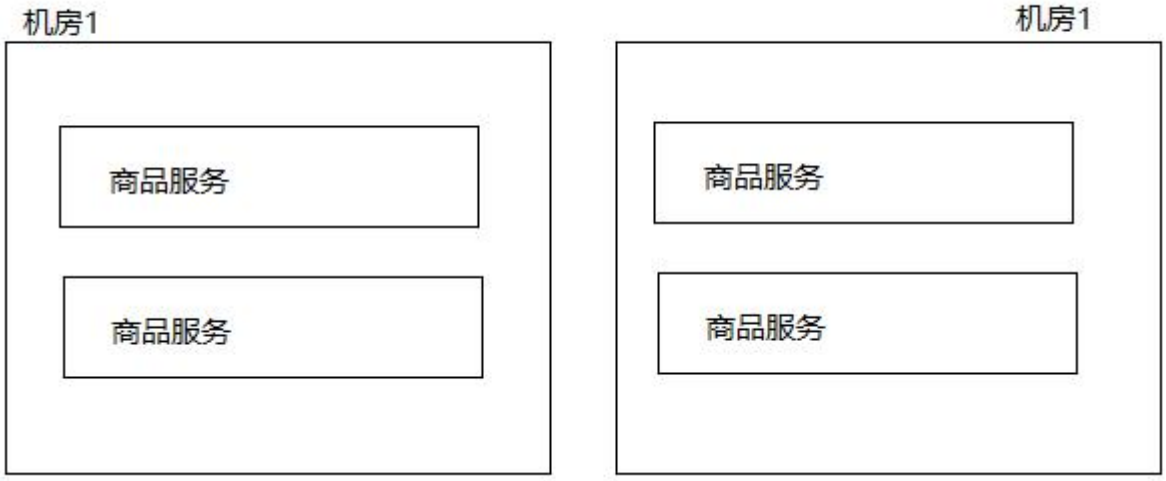
4.3、集群隔离

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。



4.4、机房隔离

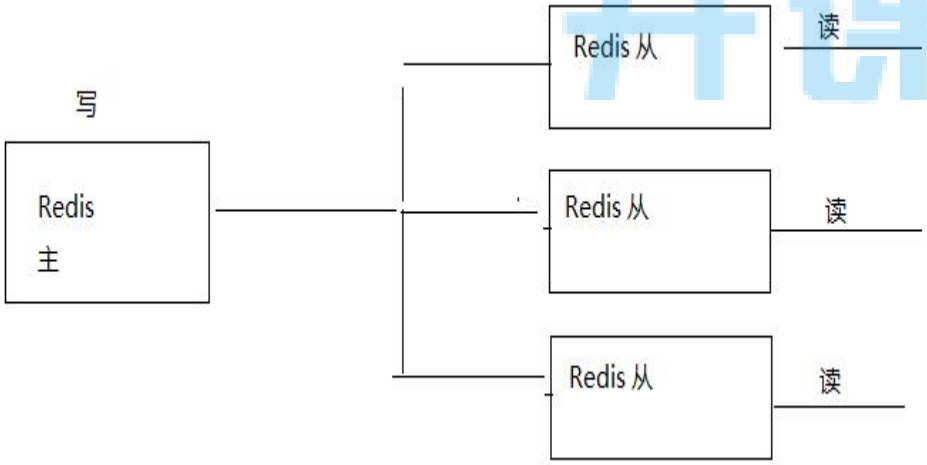
通过 DNS/负载均衡服务器进行故障转移。把请求切换到另一个机房。



根据 IP 地址进行分组，IP+组名

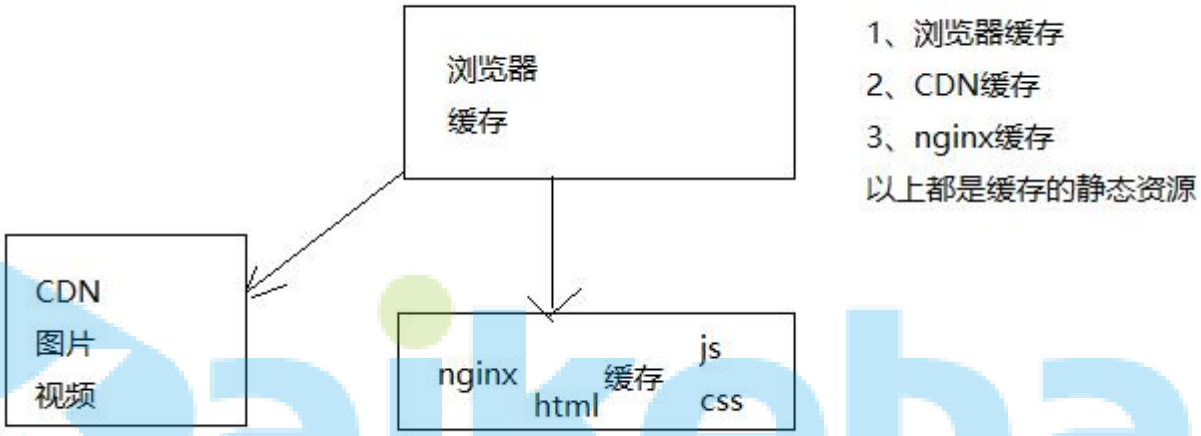
4.5、读写隔离

Redis 主从 – 读写分离
互联网项目： 读多，写少。



4.6、动静隔离

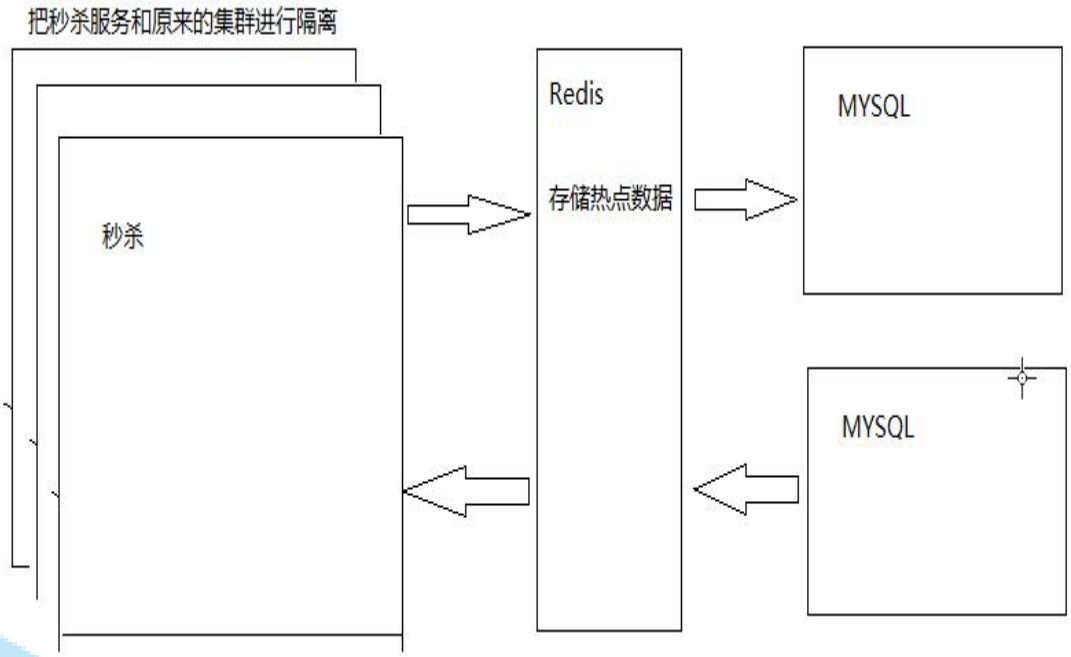
把静态资源放入 nginx，CDN 服务。达到动静隔离。防止有页面直接加载大量静态资源。。。，
因为访问量大，导致网络带宽打满，
导致卡死，出现不可用。



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



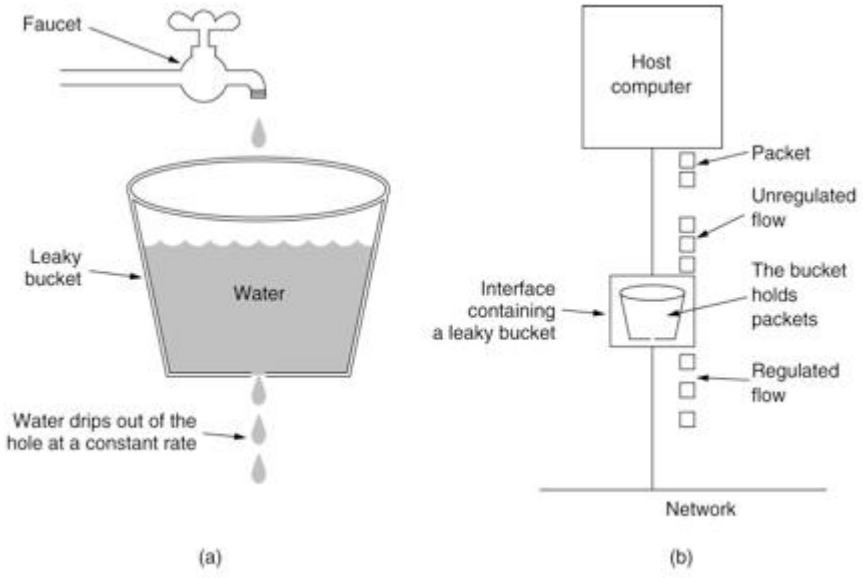
热点数据隔离，减轻数据库压力，提高项目高可用。

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

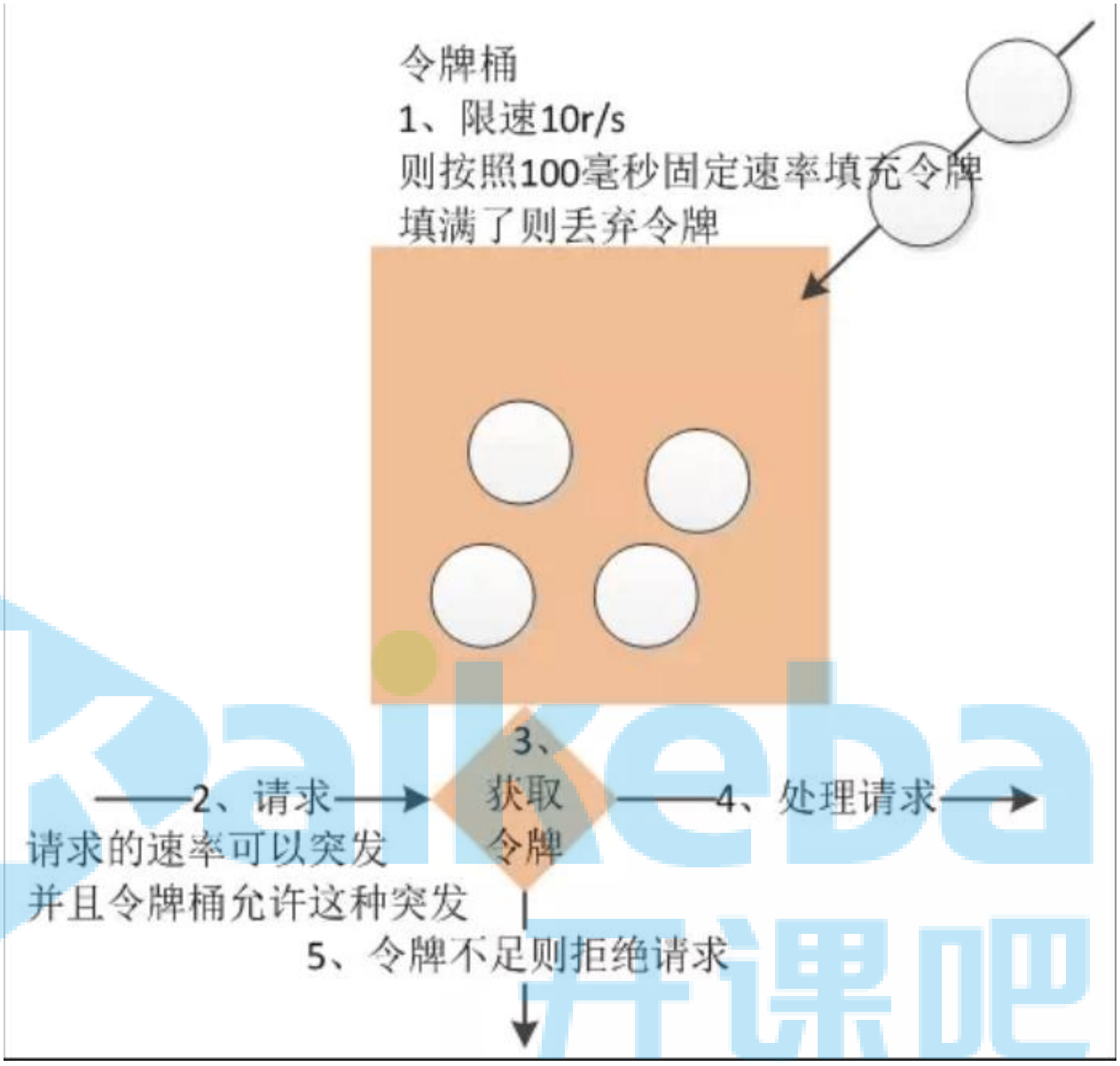
在漏斗中有水的时候

- 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" acceptCount="1000"/>
<!-- 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() / 1000;
    if(counter.get(currentSeconds).incrementAndGet() > limit) {
        System.out.println("限流了:" + currentSeconds);
        continue;
    }
    //业务处理
```

Hystrix 框架进行限流….(熔断….)

spirngcloud 限流：限流算法 ---- 单机版限流 ---- 在分布式架构项目中，要使用分布式限流方式。

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 nodelay;
}
```

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。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。

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。

6.8、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

总结：熔断 降级 到底有什么区别？？？

相同点：

1、目的很一致，都是从可靠性角度来处理服务，为了防止系统整体缓慢甚至崩溃，采用的技术手段

2、表现形式非常类似，最终让用户的体验某些功能暂时不可达或不可用。

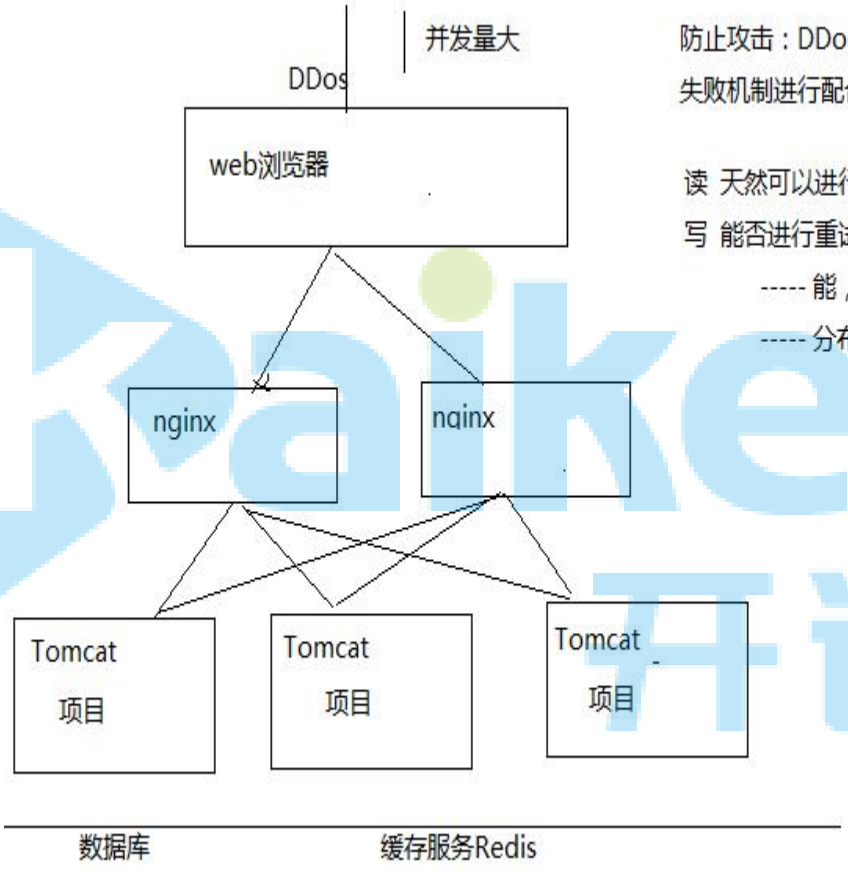
3、熔断模式基于策略自动触发，降级也可以进行自动触发。

不同点：

1、触发原因不一样，服务熔断一般情况都是下游服务故障引起。而服务降级一般都是从系统整体负荷考虑

2、实现方式不一样，服务降级代码侵入性比较高（由控制器实现开关降级，人工降级），熔断一般都是自我熔断。

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

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。