总的目录：

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

流量架构（流量预研）

流量架构包含的主要内容：

1 按照未来一段时间的用户规模，做系统的流量（吞吐量）预估。

2 按照流量预估值和系统各层组件的性能**基线值**，做各层的**组件的部署架构规划**，确保系统的高性能、高可用。

所以，最终的流量架构，就是系统在不同用户量、不同场景下（如平时、战时）的**分层规划、组件规划。**

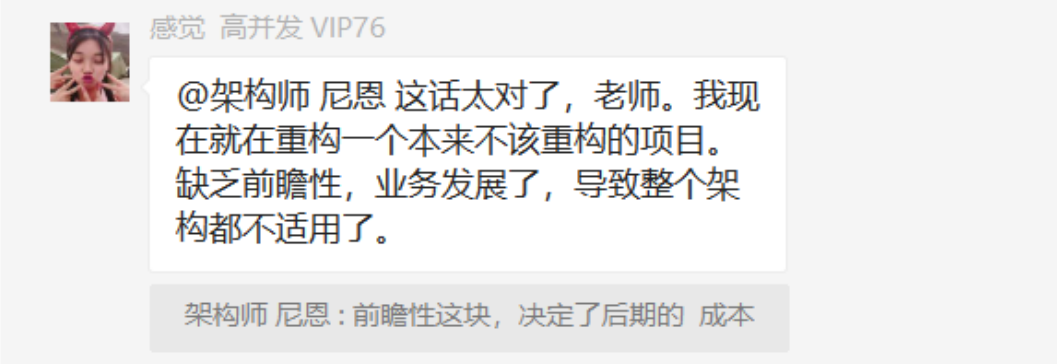
低并发系统，如何弄出高并发的经验

大部分小伙伴，都在做CRUD的业务开发工作，参与的都是低并发系统，怎么办？

现在只有1w用户，可能未来两年会有10W用户、100W用户、10000W用户

所以，提前得做好架构的规划

反过来说，没有提前做好架构的规划，等到后期性能跟不上， 又要进行升级改造，甚至推倒重来



流量架构的类型

老系统的流量架构

如果是老系统，做流量架构的时候，可以参考现有的监控数据，服务能力，流量指标 。对着老的架构版本，进行偏离指标的计算，折算成冗余系数，完成流量架构的工作：

（1）做出系统在不同用户量、不同场景（高峰、平峰、低峰）下的流量（吞吐量）预估，含未来一段时期如两年。

（2）做出系统在不同用户量、不同场景下的**各层组件的部署架构。**

新系统的流量架构

接下来我们重点说一下新系统，也是在实际工作中遇到最多的情况。

> 有的朋友可能说：我们公司的项目就是啥都没有，三无产品，没有业务监控、没有中间件日志，也没有日活数据，那怎么评估预期指标。

新项目是还没有上线，在上线前希望先进行一轮压测，评估项目性能是否能支撑当前的用户，这个时候性能预期指标更为重要。

如果是新系统，线上并没有任何的历史监控数据和日志数据，所以之前介绍的方法就不再适用，这个时候需要使用另外一种方法来评估性能指标，那就是“二八定律”。

对新系统来说，完成流量架构的工作：

（1）根据二八定律，做出系统在不同用户量、不同场景下的流量（吞吐量）预估，含未来一段时期如两年。

（2）做出系统在不同用户量、不同场景下的**各层组件的部署架构。**

亿级用户量场景下的流量预估

根据二八定律，做出系统在不同用户量、不同场景下的流量（吞吐量）预估，含未来一段时期如两年。

二八定律

什么是二八定律?

> 先来看一下定义。 二八定律又名80/20定律、帕累托法则（Pareto‘s principle）、巴莱特定律、朱伦法则（Juran’s Principle）、关键少数法则（Vital FeRule）、不重要多数法则（Trivial Many Rule）最省力的法则、不平衡原则等，被广泛应用于社会学及企业管理学等。

二八定律是19世纪末20世纪初意大利经济学家帕累托发现的。他认为，在任何一种事物中，最重要的只占其中一小部分，约20%，其余80%尽管是多数，却是次要的。

使用场景：

软件测试

软件测试理论中，常提到2-8原则

80%的测试成本花在20%的软件模块中

所谓2-8原则，即80%的bug多发生在软件的20%的模块。所以，在回归测试的时候，这20%的高发地带是关注的重点！

80%的错误是由20%的模块引起的

站在用户角度，并非研发实现的角度，正确地选择重要模块作为测试重点，从而不偏离方向。

80%的测试时间花在20%的软件模块中

软件测试执行过程中需要将时间倾斜在重要模块的测试用例中，从而使测试更加有效，发现bug

经济学场景

从经济学上看，世界上80%的财富，都集中的20%的人手里

心理学场景

从心理学来说，人类80%的智慧，都集中在20%人身上。

二八定律是一种社会准则，符合大多数社会现象的规律。同样也适用于互联网领域。

互联网行为场景

一个网站有成千上万的用户，但是80%的用户请求是发生在20%的时间内，比如大家去网上购物，基本也都集中在中午休息或晚上下班后。

二八定律的核心原则是关注重要部分，忽略次要部分。系统性能如果能支撑发生在20%时间内的高并发请求，必然也能支持非高峰期的访问。

二八定律来计算预期指标

具体来说下怎么通过二八定律来计算预期指标。

通过用户量来预估QPS

首先先预估系统的每日总请求数，这个没有固定的方法，如果没有任何历史数据参考，一般是通过用户量或者其他关联系统来评估。

术语说明：
QPS = req/sec = 请求数/秒

> 说明：对于单次接口调用的请求， QPS=TPS

【通过用户量来推算PV】

公式：(总用户数 * 20%) * 每天的大致点击次数（淘宝经验30-50次）= pv数

问：用户数是1000万，pv量是多少?

答：1000万 * 20% *30= 6000万

【PV推算QPS的公式】

公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

问：每天6000万 PV ， 多少QPS?
答：(6000万 * 0.8) / (86400 * 0.2) = 4800W/17280(QPS) =2700

【乘上**冗余系数**】

评估出来指标后，为了更加保险一些，最好再乘以一个冗余系数（偏离系数），提高预期指标，防止人为评估造成预期指标偏低的情况。

这个冗余系数一般定为2-5之间（行业经验），上面计算出来的tps指标为2700，如果再乘以一个冗余系数4，那么最终tps指标就定为10800。

2700（QPS）* 4 =10800（QPS)

总结一下，二八定律的算法为 80%的请求 / 20%的时间 * 冗余系数

冗余系数的迭代

同时，将来项目上线后，可以通过对项目接口的峰值监控，来对比之前评估的算法结果，调整冗余系数，最终随着不断的数据积累，将会形成一套本项目的性能模型。

十万级用户量的压力预估

个假设这个网站预估的用户数是10万，那么根据28法则，每天会来访问这个网站的用户占到20%，也就是2万用户每天会过来访问。

【通过用户量来推算PV】

公式：(总用户数 * 20%) / 每天的大致点击次数（淘宝经验30-50次）= pv数

问：用户数是10万，pv量是多少?

答：10万 * 20% * 30= 60万

> 通常假设平均每个用户每次过来会有30次的点击，那么总共就有60万的点击（PV）。

【PV推算QPS的公式】

公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

问：5小时内会有48万点击 ， 多少QPS?
答：60W * 0.8 / 5 * 3600 = 27 (QPS)

【乘上冗余系统 】

27 (QPS) * 4 =108 (QPS)

百万级用户量的压力预估

个假设这个网站预估的用户数是100万，那么根据28法则，每天会来访问这个网站的用户占到20%，也就是2万用户每天会过来访问。

【通过用户量来推算PV】

公式：(总用户数 * 20%) / 每天的大致点击次数（淘宝经验30-50次）= pv数

问：用户数是100万，pv量是多少?

答：100万 * 20% * 30= 600万

> 通常假设平均每个用户每次过来会有30次的点击，那么总共就有600万的点击（PV）。

【PV推算QPS的公式】

公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

问：5小时内会有48万点击 ， 多少QPS?
答：600W * 0.8 / 5 * 3600 = 270 (QPS)

【乘上冗余系统 】

270 (QPS) * 4 =1080 (QPS)

千万级用户量的压力预估

这个假设这个网站预估的用户数是1000万，那么根据28法则，每天会来访问这个网站的用户占到20%，也就是200万用户每天会过来访问。

【通过用户量来推算PV】

公式：(总用户数 * 20%) / 每天的大致点击次数（淘宝经验30-50次）= pv数

问：用户数是1000万，pv量是多少?

答：1000万 * 20% * 30= 6000万

> 通常假设平均每个用户每次过来会有30次的点击，那么总共就有6000万的点击（PV）。

【PV推算QPS的公式】

公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

问：5小时内会有48万点击 ， 多少QPS?
答：6000 W * 0.8 / 5 * 3600 = 2700 (QPS)

【加上冗余系统 】

2700 (QPS) * 4 =10800 (QPS)

亿级用户量的压力预估

这个假设这个网站预估的用户数是10000万，那么根据28法则，每天会来访问这个网站的用户占到20%，也就是2000万用户每天会过来访问。

【通过用户量来推算PV】

公式：(总用户数 * 20%) / 每天的大致点击次数（淘宝经验30-50次）= pv数

问：用户数是1000万，pv量是多少?

答：10000万 * 20% * 30= 60000万

> 通常假设平均每个用户每次过来会有30次的点击，那么总共就有60000万的点击（PV）。

【PV推算QPS的公式】

公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

问：5小时内会有48万点击 ， 多少QPS?
答：60000 W * 0.8 / 5 * 3600 = 27000 (QPS)

【加上冗余系统 】

27000 (QPS) * 4 =108000 (QPS)

实际与理论的差距

那么将来项目上线后，接口的访问量真的和计算的一模一样吗?

这个肯定不会，大家一定得知道一个原则，**性能测试从来都不是一门非常精确的技术。**

二八定律也并不是100%适用于所有业务场景。在没有任何历史数据参考的背景下，二八定律相对来说是一种相对来说靠谱的算法，最起码有一定的理论依据，比拍脑袋猜的值靠谱多了。

亿级用户量场景下的流量架构

根据上面的二八定律和冗余系数，假设1亿用户，大概知道了高峰期每秒钟可能会有10万左右的QPS之后，来看一下系统中各层组件的部署架构。

各组件的并发能力基线值（参考值）

1）Nginx：一个高性能的Web-Server和实施反向代理的软件

2）LVS：Linux Virtual Server，使用集群技术，实现在Linux操作系统层面的一个高性能、高可用、负载均衡服务器

3）Keepalived：一款用来检测服务状态存活性的软件，常用来做高可用

4）F5：一个高性能、高可用、负载均衡的硬件设备

5）DNS轮询：通过在DNS-Server上对一个域名设置多个IP解析，来扩充Web-Server性能及实施负载均衡的技术

Tomcat

tomcat 默认配置的最大请求数是150，也就是说同时支持150个并发。具体能承载多少并发，需要看硬件的配置，CPU 越多性能越高，分配给JVM的内存越多性能也就越高，但也会加重GC的负担。当

某个应用拥有 250 个以上并发的时候，应考虑应用服务器的集群。

> 一般来说，虽然tomcat的io线程最多控制在400以内，如果每个请求要300ms，一个线程3qps，那么一个tomcat到达1000qps，还是可以的。

所以，tomcat参考的并发能力为 1000qps

Nginx的并发能力

看到这里，这个框架的弱点仍然是Nginx结点的并发问题和单点故障。

对于Nginx的抗并发能力，官方给出的是**5w并发量**，即轻轻松松处理5w的并发访问。

对比tomcat，之所以有这么大抗并发，主要原因有两个：

一是Nginx只做请求和响应的转发而没有业务逻辑处理，大部分的时间花在与其他计算的I/O上；

二是Nginx的I/O采用的是单线程或少线程、异步非阻塞的模式（Tomcat是一个连接一个线程，同步阻塞的模式），避免了打开I/O通道等待数据传输的过程（仅仅是在数据传到了，再来接收即可），极大的缩短了线程调度和I/O处理的时间。

MySQL

MySQL数据库查询能力

- 主键查询：千万级别数据 = 1~10 ms ， 4核心 8线程 为 1000qps* 8=8000qps

- 唯一索引查询：千万级别数据 = 10~100 ms ， 4核心 8线程 为 100qps* 8=800qps
- 非唯一索引查询：千万级别数据 = 100~1000 ms ， 4核心 8线程 为 10qps* 8=80qps
- 无索引：百万条数据 = 1000 ms+

综合来说，mysql的并发能力，大概在1500qps左右

MySQL数据库事务能力

- 更新删除（与查询相同）
- 插入操作，依赖于配置优化，比查询的效率低

MySQL的连接数限制

MySQL默认配置的最大连接数是151。可以将最大连接数设置的最大值为100000。一般情况在Linux系统中建议设置为500-1000。

https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html

- `max_connections`

Command-Line Format	--max-connections=#
System Variable	max_connections
Scope	Global
Dynamic	Yes
Type	Integer
Default Value	151
Minimum Value	1
Maximum Value	100000

> The maximum permitted number of simultaneous client connections. For more information, see Section 5.1.11.1, “Connection Interfaces”.

Redis 单机性能测试

> Redis 单机为5w QPS左右。

由于生产环境下业务服务器总响应延迟需要控制在100ms内，为了尽量减少日志输出环节的耗时，考虑将日志吐到redis中缓存，由其他程序异步的从redis中取数据。在生产环境改造日志数据系统之前，对单机的redis读写性能做了测试。

测试单实例 redis的 读写list 数据结构性能

本地/局域网	读写类型	测试命令	client连接数	qps	延迟响应 <= 1ms比例	延迟响应 <= 2ms比例	延迟响应 <= 4ms比例	延迟响应 <= 8ms比例
本地	写	lpush	1	35714	100%	100%	100%	100%
本地	写	lpush	2	63000	100%	100%	100%	100%
本地	写	lpush	4	153846	100%	100%	100%	100%
本地	写	lpush	8	155914	100%	100%	100%	100%
本地	写	lpush	16	151788	99.99%	100%	100%	100%
本地	读	rpop	1	38971	100%	100%	100%	100%
本地	读	rpop	2	65832	100%	100%	100%	100%
本地	读	rpop	4	162469	100%	100%	100%	100%
本地	读	rpop	8	184928	100%	100%	100%	100%
本地	读	rpop	16	171452	100%	100%	100%	100%
局域网	写	lpush	1	10009	99.99%	99.99%	100%	100%
局域网	写	lpush	2	13563	99.98%	99.98%	99.99%	100%
局域网	写	lpush	4	25601	99.98%	99.99%	100%	100%
局域网	写	lpush	8	43830	99.98%	99.98%	99.99%	100%
局域网	写	lpush	16	68820	99.94%	99.97%	99.99%	100.00%

本地/局域网	读写类型	测试命令	client连接数	qps	延迟响应 <= 1ms比例	延迟响应 <= 2ms比例	延迟响应 <= 4ms比例	延迟响应 <= 8ms比例
局域网	读	rpop	1	6214	99.97%	99.98%	99.99%	100%
局域网	读	rpop	2	13289	99.94%	99.97%	99.99%	100%
局域网	读	rpop	4	24900	99.98%	99.98%	99.99%	100%
局域网	读	rpop	8	37118	99.96%	99.98%	99.99%	100%
局域网	读	rpop	16	69492	99.93%	99.96%	99.99%	100%

测试环境和测试工具

CPU：8核

内存：8G

Redis版本：3.2.6

测试工具：redis官方基准测试工具 redis-benchmark

Springcloud Gateway

> 单节点可以按照 预估。

一次8核8G压测结果：

并发数：300；

netty工作线程数（reactor.netty.ioWorkerCount）：8（默认）

样本数据：返回1.5k大小

服务端响应时间：10ms左右

测试时长：5分钟

JVM内存：2G

Requests	Executions			Response Times (ms)							Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Throughput	Received	Sent
Total	3199314	0	0.00%	29.41	1	7038	24.00	30.00	60.00	9668.17	7798.74	3332.88
HTTP Request	3199314	0	0.00%	29.41	1	7038	24.00	30.00	60.00	9668.17	7798.74	3332.88

JMeter报告

```
top - 09:51:20 up 15:19,  3 users,  load average: 11.48, 9.28, 8.05
Tasks: 123 total,   2 running, 121 sleeping,   0 stopped,   0 zombie
%Cpu0  : 47.9 us, 17.7 sy,  0.0 ni, 32.6 id,  0.0 wa,  0.0 hi,  0.4 si,  1.4 st
%Cpu1  : 53.1 us, 16.1 sy,  0.0 ni, 30.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.7 st
%Cpu2  : 46.3 us, 15.9 sy,  0.0 ni, 36.4 id,  0.0 wa,  0.0 hi,  0.0 si,  1.4 st
%Cpu3  : 48.4 us, 17.3 sy,  0.0 ni, 32.5 id,  0.0 wa,  0.0 hi,  0.0 si,  1.8 st
%Cpu4  : 54.2 us, 16.4 sy,  0.0 ni, 29.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
%Cpu5  : 50.2 us, 17.5 sy,  0.0 ni, 30.5 id,  0.0 wa,  0.0 hi,  0.0 si,  1.8 st
%Cpu6  : 53.5 us, 18.1 sy,  0.0 ni, 27.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.7 st
%Cpu7  : 37.5 us, 14.5 sy,  0.0 ni,  4.6 id,  0.0 wa,  0.0 hi, 42.4 si,  1.1 st
KiB Mem :  8008212 total,  6342456 free,  1355016 used,   310740 buff/cache
KiB Swap:  8388604 total,  8388604 free,        0 used.  6391932 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
20412 root      20   0 8328764   1.2g  14268 S 577.7 15.3  21:50.25 java
```

CPU负载

netty工作线程数调整为（reactor.netty.ioWorkerCount）：12；

Requests	Executions			Response Times (ms)							Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Throughput	Received	Sent
Total	3395650	0	0.00%	27.74	1	7041	26.00	36.00	67.00	10240.66	8260.53	3530.23
HTTP Request	3395650	0	0.00%	27.74	1	7041	26.00	36.00	67.00	10240.66	8260.53	3530.23

JMeter报告

```
top - 09:35:35 up 15:03,  2 users,  load average: 15.07, 12.44, 6.81
Tasks: 121 total,   2 running, 119 sleeping,   0 stopped,   0 zombie
%Cpu0  : 54.1 us, 18.3 sy,  0.0 ni, 26.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.7 st
%Cpu1  : 50.7 us, 18.4 sy,  0.0 ni, 30.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.7 st
%Cpu2  : 52.7 us, 20.4 sy,  0.0 ni, 26.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
%Cpu3  : 53.7 us, 21.3 sy,  0.0 ni, 24.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.7 st
%Cpu4  : 55.9 us, 19.0 sy,  0.0 ni, 24.1 id,  0.0 wa,  0.0 hi,  0.0 si,  1.0 st
%Cpu5  : 52.8 us, 19.7 sy,  0.0 ni, 26.4 id,  0.0 wa,  0.0 hi,  0.0 si,  1.1 st
%Cpu6  : 54.0 us, 18.3 sy,  0.0 ni, 27.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.7 st
%Cpu7  : 35.3 us, 16.3 sy,  0.0 ni,  5.5 id,  0.0 wa,  0.0 hi, 41.9 si,  1.0 st
KiB Mem :  8008212 total,  6221104 free,  1476768 used,   310340 buff/cache
KiB Swap:  8388604 total,  8388604 free,        0 used.  6270084 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
19239 root      20   0 8304224   1.3g  14260 S 605.6 16.8  34:25.70 java
```

系统的流量架构规划

系统必备：进行部署架构的规划

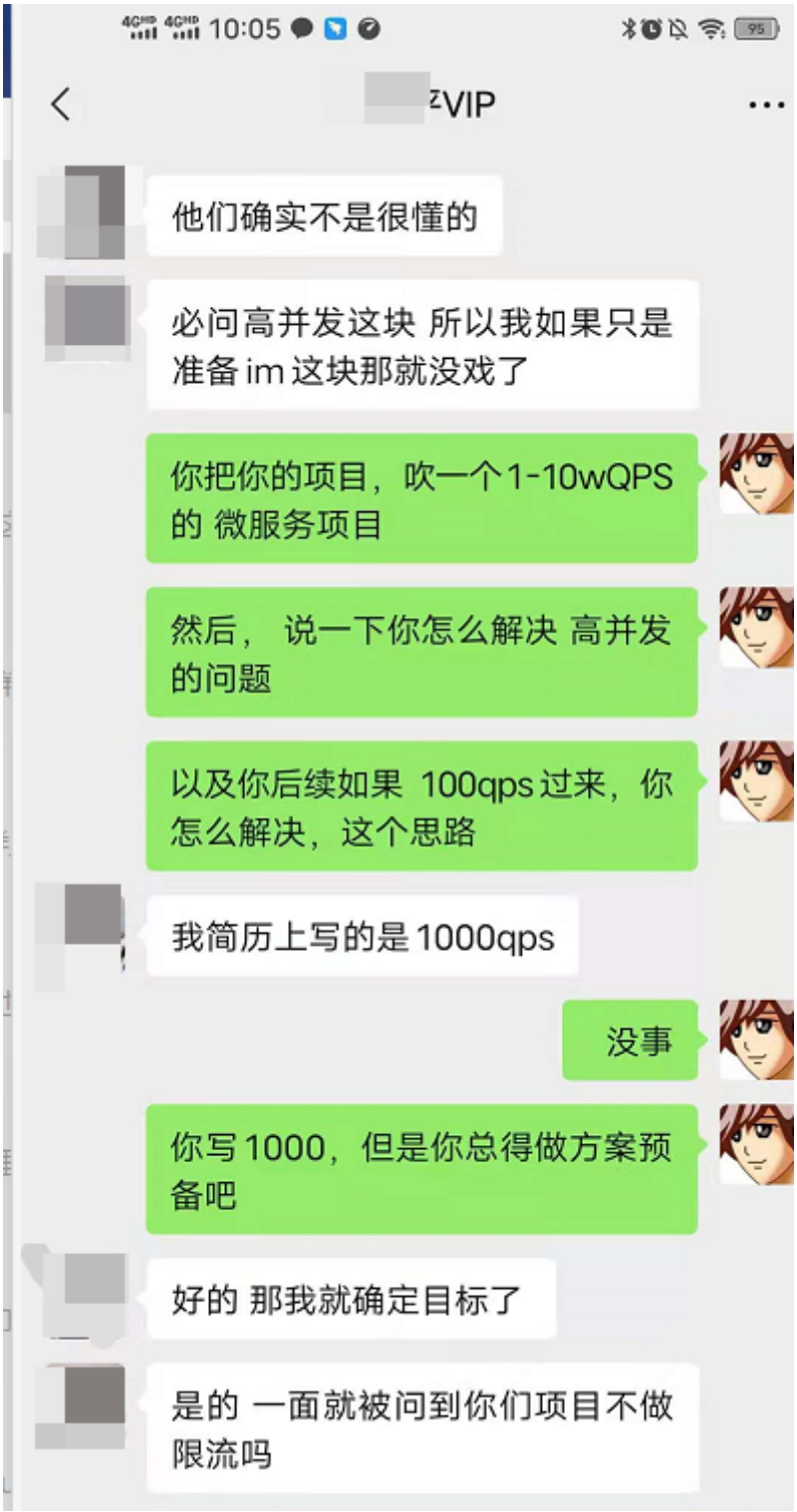
按照系统未来的用户规模，进行部署架构的规范

比如说 每月增长10W，1年之内100W用户，未来的架构如何？

然后 2年之内1000W用户，未来的架构如何？

然后 3年之内10000W用户，未来的架构如何？

生产项目，没有那么高的并发量，怎么办？

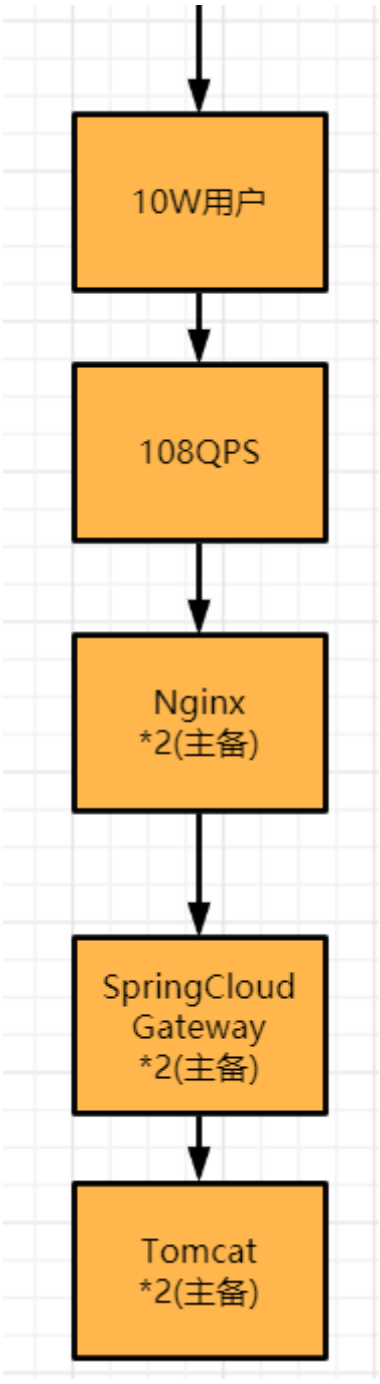


答案是：**规划和预研**

参考资料：

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

10W用户 各层的部署架构



100W用户的各层的部署架构

1080qps

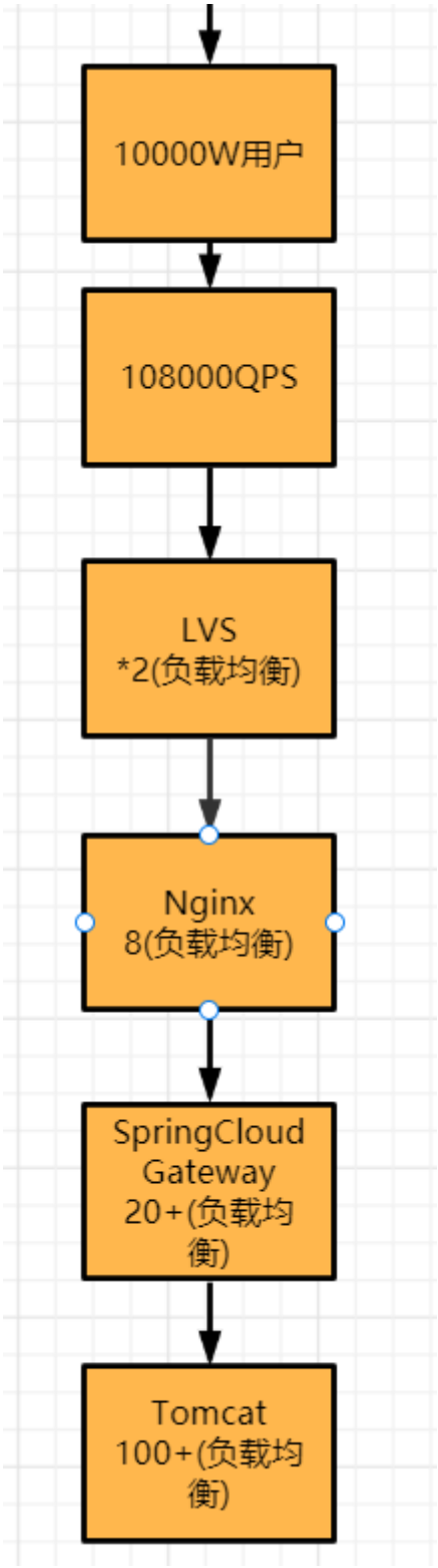
1000W用户的各层的部署架构

10800qps

10000W用户的各层的部署架构

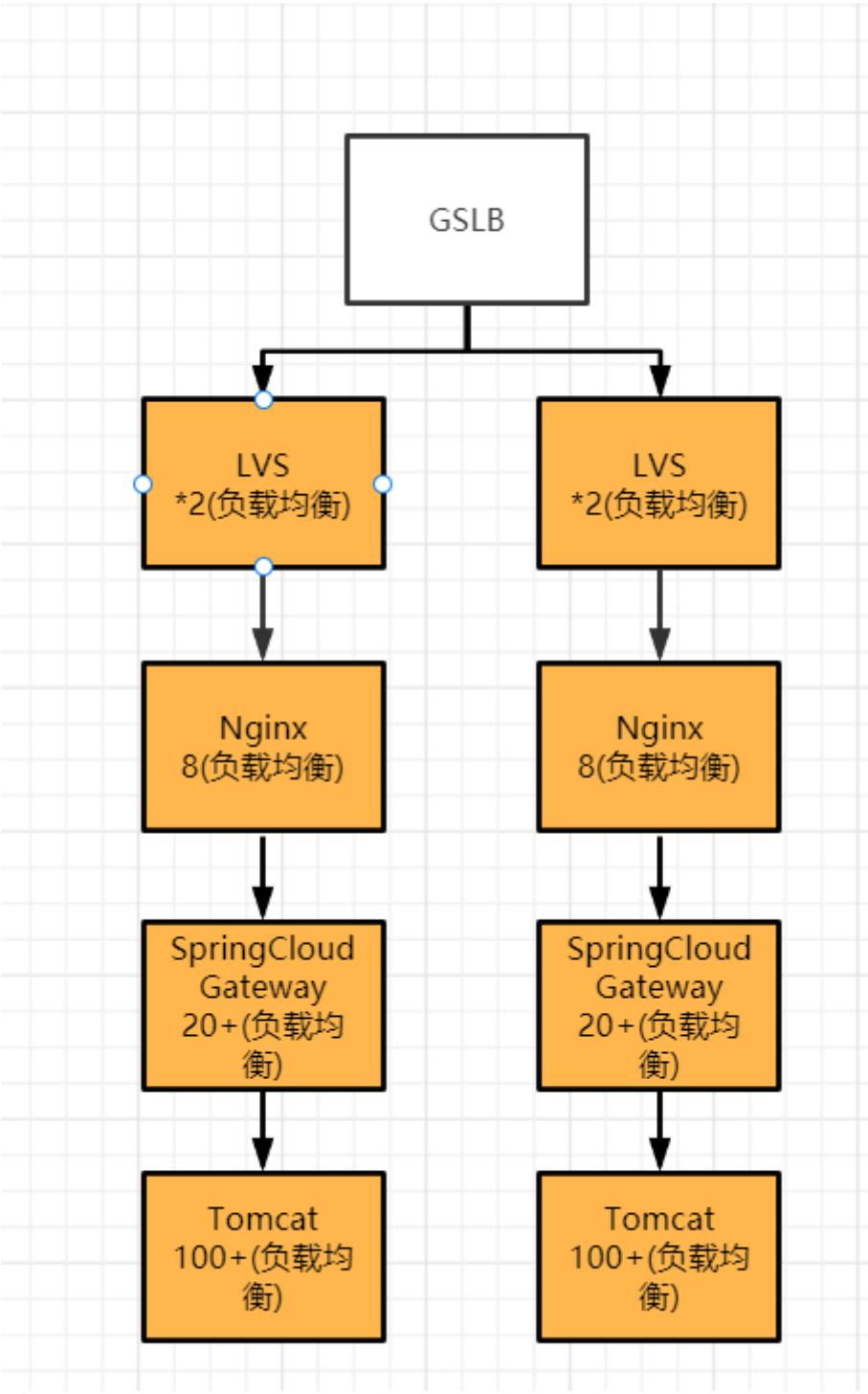
108000qps

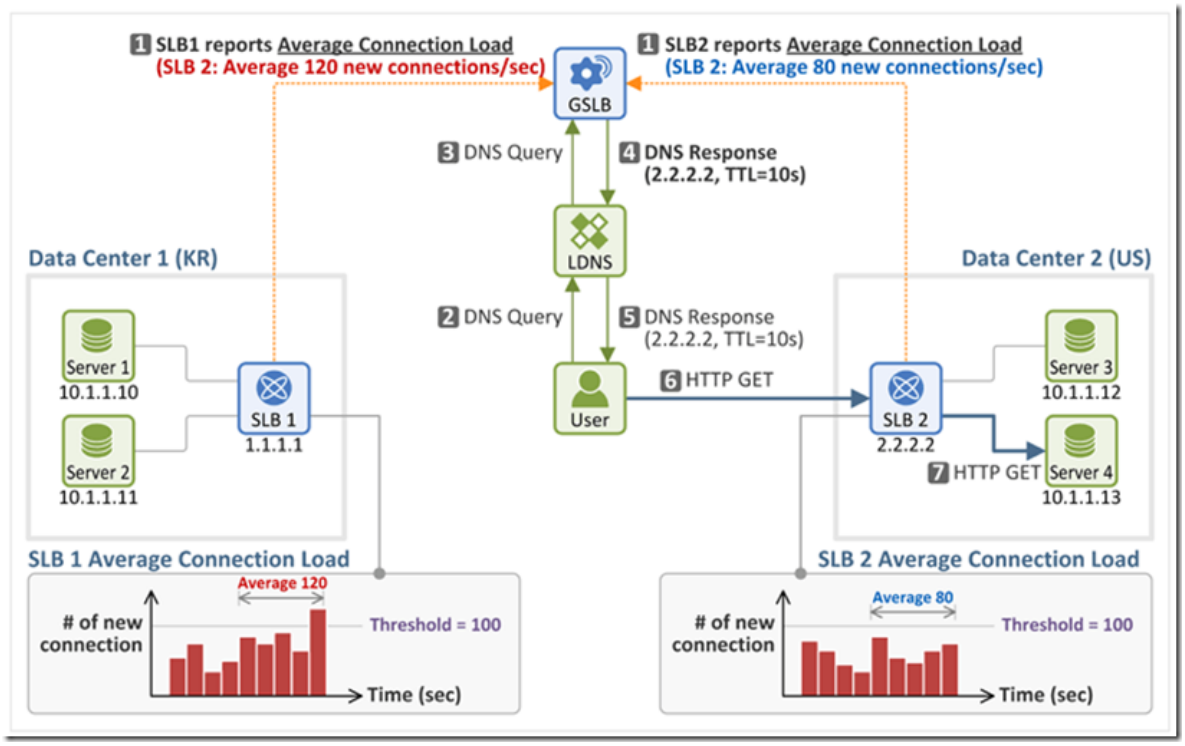
###



异地多活的流量架构

- 通过 dns + gslb （智能ldns）做负载均衡，将请求路由到最近的可用服务的机房。
- 服务全部机房内自治，不进行跨机房访问
- 不同的机房，进行数据异步双向复制


GSLB
LVS
*2(负载均衡)
LVS
*2(负载均衡)
Nginx
8(负载均衡)
Nginx
8(负载均衡)
SpringCloud
Gateway
20+(负载均
衡)
SpringCloud
Gateway
20+(负载均
衡)
Tomcat
100+(负载均
衡)
Tomcat
100+(负载均
衡)


1 SLB1 reports Average Connection Load
(SLB 2: Average 120 new connections/sec)
1 SLB2 reports Average Connection Load
(SLB 2: Average 80 new connections/sec)
GSLB
3 DNS Query
4 DNS Response
(2.2.2.2, TTL=10s)
LDNS
2 DNS Query
5 DNS Response
(2.2.2.2, TTL=10s)
User
6 HTTP GET
Data Center 1 (KR)
Server 1
10.1.1.10
Server 2
10.1.1.11
SLB 1
1.1.1.1
Data Center 2 (US)
Server 3
10.1.1.12
SLB 2
2.2.2.2
7 HTTP GET
Server 4
10.1.1.13
SLB 1 Average Connection Load
Average 120
# of new connection
Threshold = 100
Time (sec)
SLB 2 Average Connection Load
Average 80
# of new connection
Threshold = 100
Time (sec)