第 5 周 非功能（上）

秦金卫/KimmKing

资深架构师

KK架构训练营讲师

讲师介绍

- Apache Dubbo/ShardingSphere PMC
- 资深开源参与者/前阿里巴巴/京东架构师
- 《高可用可伸缩微服务架构》作者
- 《JVM 核心技术32讲》专栏作者
- 《面向性能的系统架构设计》作者
- 阿里云MVP/腾讯云TVP/TGO会员

目录

1. 非功能概述

① 非功能介绍

② 非功能测试

1. 非功能概述

1.1 非功能介绍

什么是非功能

软件系统的非功能性特性（也称为质量属性或质量非功能性需求）是指对系统性能、可靠性、安全性等方面的要求，而非直接影响系统功能的需求。

- 性能（Performance）： 包括响应时间、吞吐量、并发性等，确保系统在给定条件下能够有效执行。
- 可靠性（Reliability）： 系统在运行时的稳定性和可靠性，以及其在发生故障时的恢复能力。
- 可用性（Availability）： 衡量系统在一定时间内可用的程度，通常以百分比表示。
- 可维护性（Maintainability）： 衡量系统容易进行修改、维护和支持的程度。
- 安全性（Security）： 保护系统免受未经授权的访问、数据泄露和其他安全威胁。
- 可移植性（Portability）： 系统能够在不同的硬件、操作系统或环境中运行的能力。
- 可扩展性（Scalability）： 系统能够有效地处理不断增长的工作负载或数据量。
- 兼容性（Compatibility）： 系统与其他系统、硬件或软件的集成和互操作性。
- 易用性（Usability）： 用户界面友好、易于学习和使用的程度。
- 效率（Efficiency）： 系统在资源利用方面的效率，包括内存、处理器和存储器的使用。
- 合规性（Compliance）： 系统符合特定的法规、标准或行业规范的程度。

非功能的重要意义

非功能性特性对于确保软件系统的质量、性能和可用性至关重要。

在软件开发过程中，对这些特性进行明确定义和测试是确保系统满足用户期望的关键步骤。

功能如果说是满足了用户的某种表面需求，那么非功能则是对这些功能的效用边界做了兜底。

但是一个悖论在于，功能能看见，非功能看不见。

所以，非功能往往容易被忽视，工作量无法预估，以及出了问题难以解决。

但是架构师需要考虑这些，所以有人说，架构是非功能的总和。

非功能的侧重点

特别是某些场景下，非功能特性会变成非常核心的本质要求。

比如安全性对于任何涉及资金账户的系统。

互联网的某些应用，需要的第一要点可能是快，足够的快，性能就很重要。

而传统金融的某些核心系统，高可用和稳定性则是最重要的。

对于一个高频使用的手机APP，那么易用性可能非常重要。

性能

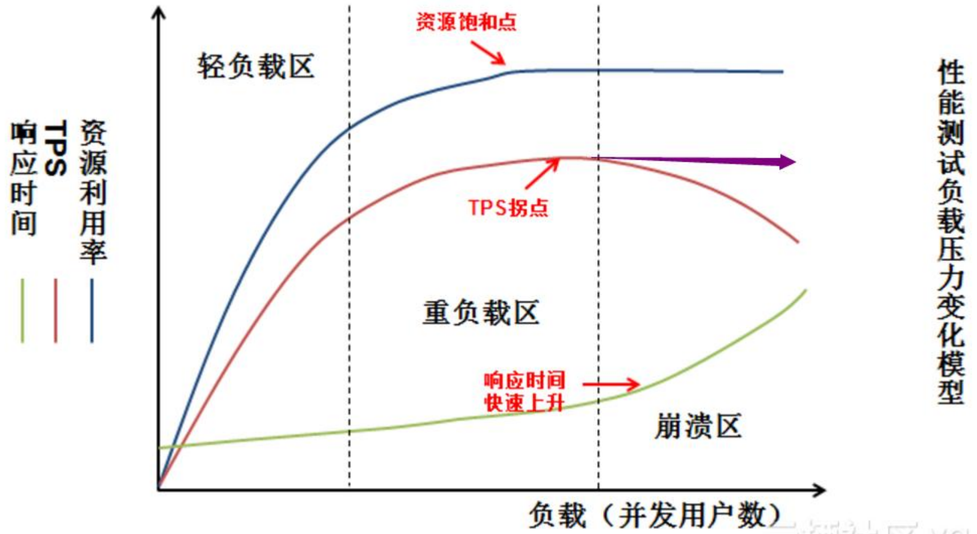
性能包括包括吞吐量，延迟，容量。

吞吐量：单位时间内处理的业务数量。

延迟：每笔业务处理的时间。

并发：系统支持的并发数量。

容量：系统在特定并发下支持的最大吞吐量。



高可用性

可用性定义：系统的可用时间占比。几个9，宕机次数与总时长的关系。
相关概念，可靠性：稳定与恢复能力。RPO与RTO。

可用性的问题在于，成本和效益的权衡。对比安排几个人干活。
后面单独讲，不多解释。

一致性

一致性一般不算到非功能中，但是很多时候，我们去考虑一致性的不同情况，就不可以看做是非功能了。

强一致性：实时一致性。

弱一致性：对一致性要求不高，可接受部分数据丢失。

最终一致性：中间过程中，数据可以不一致，最终在某个时间点需要数据一致。

一致性的妥协，可以大大简化系统的设计复杂度，并且提高性能等其他非功能性。

扩展性

可扩展性（Scalability）是指软件系统在面对不断增长的负载或需求时，能够有效地处理这些增加，而不影响系统的性能、效率和用户体验。可扩展性是一个系统的重要非功能性特性，涉及到系统设计和架构的方方面面，以确保系统在不同规模和复杂度下都能够保持高效运行。

- 水平扩展性（Horizontal Scalability）： 指通过增加系统的节点或实例数量来扩展系统。例如，通过增加服务器的数量来分散负载，从而提高系统的整体性能。
- 垂直扩展性（Vertical Scalability）： 指通过增加单个节点的资源（如增加服务器的处理能力、内存或存储容量）来扩展系统。这通常涉及到升级硬件来提高系统的性能。
- 弹性（Elasticity）： 指系统能够根据负载的变化自动调整其规模，以保持性能。这可以是动态地增加或减少计算资源，以适应变化的需求。
- 容量规划（Capacity Planning）： 涉及对系统未来增长的需求进行预测和规划，以确保系统具有足够的资源来满足这些需求。
- 分布式架构（Distributed Architecture）： 通过将系统划分为多个独立的组件或服务，使系统能够更容易地进行水平扩展，并充分利用分布式计算的优势。
- 负载均衡（Load Balancing）： 将请求分发到多个服务器或节点，以确保负载在整个系统中均匀分布，避免单一节点成为性能瓶颈。

实现可扩展性可以提高系统的灵活性、稳定性和可维护性，同时降低系统维护和运行的成本。在设计和开发阶段，考虑和实施合适的可扩展性策略对于应对未来的增长是至关重要的。

安全性

安全性（Security）是指软件系统的一种性质，其目标是保护系统中的数据、功能和资源不受未经授权的访问、破坏或泄露。安全性是一个软件系统的重要非功能性需求，对于保护用户信息、维护业务机密性、确保系统的可靠性和可信度至关重要。

1. 身份认证（Authentication）： 确保只有经过授权的用户能够访问系统。这通常涉及使用用户名、密码、多因素认证等方式验证用户身份。

2. 授权（Authorization）： 确保经过认证的用户只能访问其有权访问的资源和功能，限制未经授权用户的权限。

3. 数据保护（Data Protection）： 保护数据的机密性、完整性和可用性，通过加密、访问控制等手段来防止数据泄露或损坏。

4. 网络安全（Network Security）： 保护系统在网络中的通信安全，防止恶意攻击、拒绝服务攻击等网络威胁。

5. 漏洞和攻击防护（Vulnerability and Attack Prevention）： 防范系统中的漏洞，采取措施来防止各种类型的攻击，如注入攻击、跨站脚本攻击等。

6. 安全审计和监控（Security Auditing and Monitoring）： 记录和监视系统的活动，以便检测潜在的安全问题，并在发现异常时进行响应。

7. 物理安全（Physical Security）： 保护存储设备、服务器和其他硬件资源，以防止未经授权的物理访问。

8. 应急响应和恢复（Incident Response and Recovery）： 制定和实施应急响应计划，以便在发生安全事件时能够快速、有效地应对和恢复。

综合考虑这些安全性方面，软件系统可以更好地保护用户和组织的利益，防止潜在的威胁和风险。在软件开发的过程中，采用安全性最佳实践，进行安全性评估和测试是确保系统安全性的重要步骤。

可维护性

可维护性（Maintainability）是指软件系统在其生命周期内易于进行修改、调整、扩展、诊断和修复的能力。一个具有良好可维护性的系统能够使开发人员更容易理解、改进和维护代码，从而降低开发和维护成本，提高系统的可靠性和稳定性。

- 可读性（Readability）： 代码易于理解，有清晰的结构和注释，使开发人员能够迅速了解代码的功能和逻辑。
- 模块化（Modularity）： 系统的组件和模块之间存在清晰的接口和关系，使得单个模块的修改不会对其他模块产生不必要的影响。
- 松散耦合（Loose Coupling）： 模块之间的依赖关系较弱，使得修改一个模块不会对其他模块造成连锁反应。
- 高内聚性（High Cohesion）： 模块内部的功能高度相关，一个模块专注于一个特定的任务，提高了代码的清晰度和可维护性。
- 文档化（Documentation）： 有充分的文档，包括代码注释、技术文档和用户文档，以帮助开发人员理解系统的设计和功能。
- 易测试性（Testability）： 系统的组件易于测试，有良好的测试覆盖率，这有助于及早发现和修复潜在问题。
- 可追踪性（Traceability）： 可追踪的代码修改历史，使得可以了解每个变更是如何影响系统的。
- 灵活性（Flexibility）： 系统容易适应新的需求和变化，不需要大规模的代码重构。
- 易调试性（Debuggability）： 提供良好的调试信息和工具，使得开发人员能够迅速定位和修复问题。

良好的可维护性不仅有助于降低系统的总体成本，还有助于应对需求变化和技术演进，确保系统长期有效运行。在软件开发的过程中，注重可维护性是一个重要的设计和编码原则。

易用性

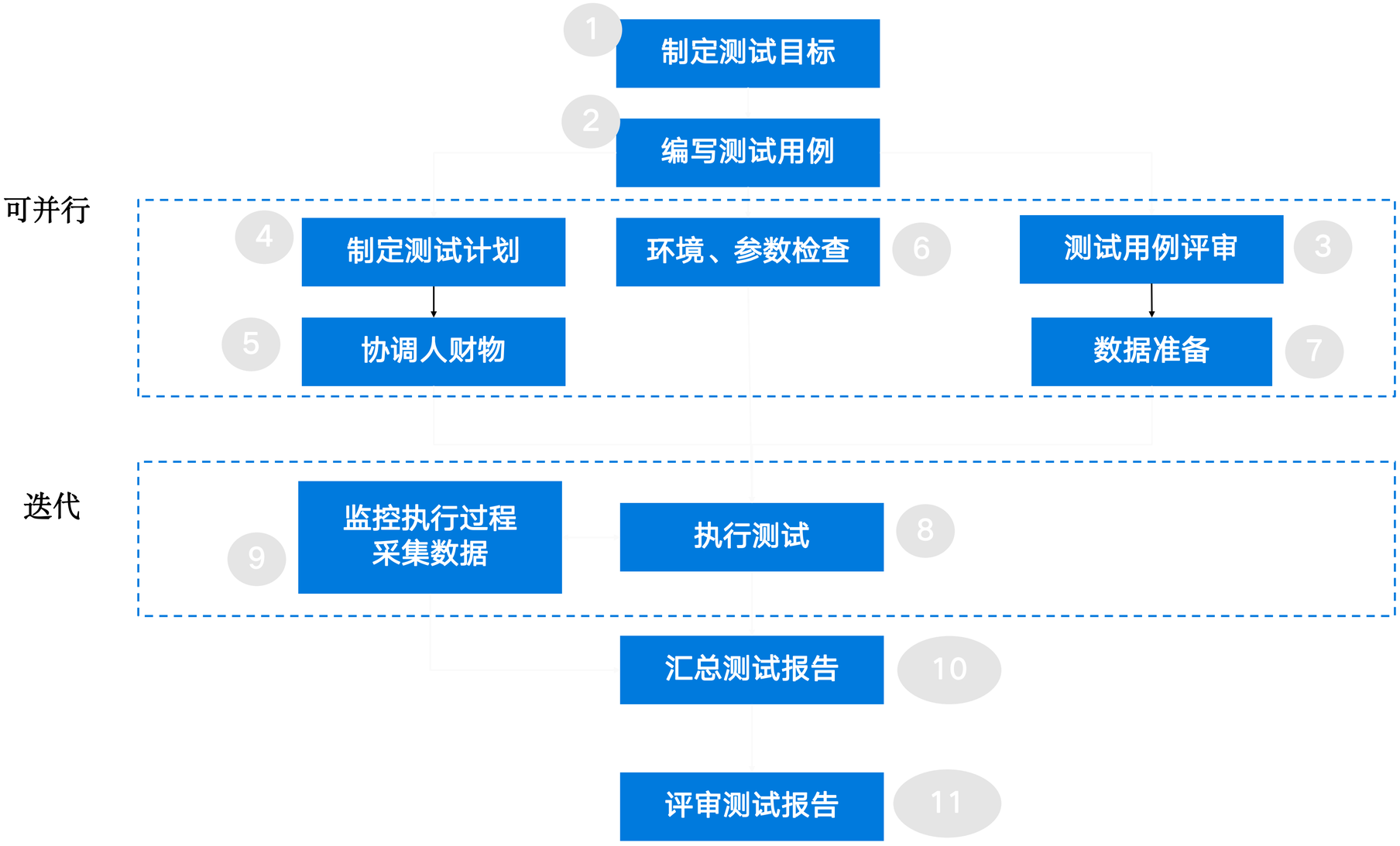
易用性（Usability）是指软件系统或产品的用户界面和交互设计是否符合用户的期望、容易使用以及是否提供了良好的用户体验。一个具有良好易用性的系统应该能够使用户轻松地完成任务，减少用户的学习曲线，并提供愉悦的使用体验。易用性是软件设计中重要的非功能性需求之一。

- 学习性（Learnability）： 用户是否能够迅速学会如何使用系统，减少学习成本。
- 效率（Efficiency）： 用户在使用系统时能够快速完成任务，不需要过多的步骤或操作。
- 记忆性（Memorability）： 用户在一段时间不使用系统后，是否仍然能够轻松地记得如何使用系统。
- 错误率和容错性（Error Rate and Tolerance）： 用户在使用系统时是否容易犯错，以系统是否提供了良好的错误提示和纠正机制。
- 满意度（Satisfaction）： 用户对系统的整体满意度和使用体验感受。
- 可用性（Accessibility）： 系统是否对所有用户，包括有特殊需求的用户，都是可访问的。
- 一致性（Consistency）： 系统的界面和交互设计是否一致，使得用户在不同部分或模块中能够轻松理解和操作。
- 反馈（Feedback）： 系统是否能够及时地向用户提供反馈，例如成功或失败的信息，以及用户的操作是否被系统正确理解。
- 直观性（Intuitiveness）： 系统的设计是否直观，使得用户能够直观地理解如何使用系统，而无需繁琐的说明或培训。

良好的易用性设计可以提高用户的满意度，降低用户的挫折感，增强系统的可接受性。在软件开发中，团队通常会进行用户体验研究、用户测试和迭代设计来确保系统具有良好的易用性。

1. 非功能概述
1.2 非功能测试

非功能测试整体流程

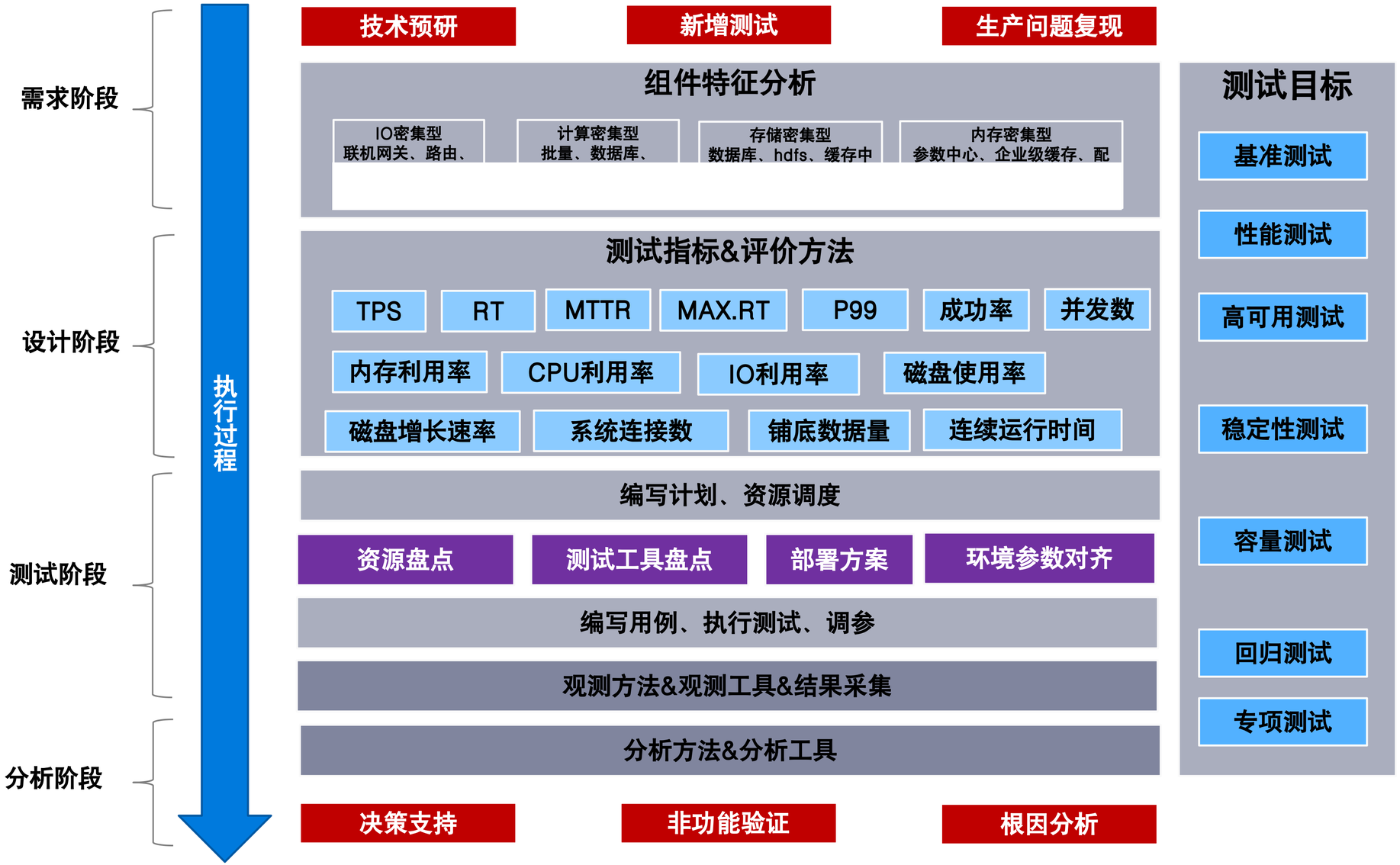

1
制定测试目标
2
编写测试用例
可并行
4
制定测试计划
环境、参数检查
6
测试用例评审
3
5
协调人财物
数据准备
7
迭代
9
监控执行过程
采集数据
执行测试
8
汇总测试报告
10
评审测试报告
11

非功能测试难点与分类


选对方法
版本变化
根因分析
时间进度
资源调度
环境干扰


容量测试
基准测试
性能测试
非功能测试
稳定性测试
高可用测试
专项测试

非功能测试工作内容


需求阶段
设计阶段
测试阶段
分析阶段
执行过程
技术预研
新增测试
生产问题复现
组件特征分析
IO密集型
联机网关、路由、
计算密集型
批量、数据库、
存储密集型
数据库、hdfs、缓存中
内存密集型
参数中心、企业级缓存、配
测试指标&评价方法
TPS
RT
MTTR
MAX.RT
P99
成功率
并发数
内存利用率
CPU利用率
IO利用率
磁盘使用率
磁盘增长速率
系统连接数
铺底数据量
连续运行时间
编写计划、资源调度
资源盘点
测试工具盘点
部署方案
环境参数对齐
编写用例、执行测试、调参
观测方法&观测工具&结果采集
分析方法&分析工具
决策支持
非功能验证
根因分析
测试目标
基准测试
性能测试
高可用测试
稳定性测试
容量测试
回归测试
专项测试

非功能测试目的

容量测试	基准测试	稳定性	性能	高可用	专项测试
• 找到TPS、RT极限用于帮助制定指标	• 1种交易梯+度并发压力观察RT与TPS，用于帮助制定指标	• 很多应用在长期运行的情况下可能会发生异常不到的情况。	• 主要分为观察TPS、RT、QPS。但是不同的应用观察方法不一样	• 宕机、网路延迟、故障发生后系统的表现	• 深度测试某个组件，需要进行端到端测试。

制定测试指标-容量

如果系统已经上线：则部署模型、铺底数据

要根据生产环境*等比例缩放*。

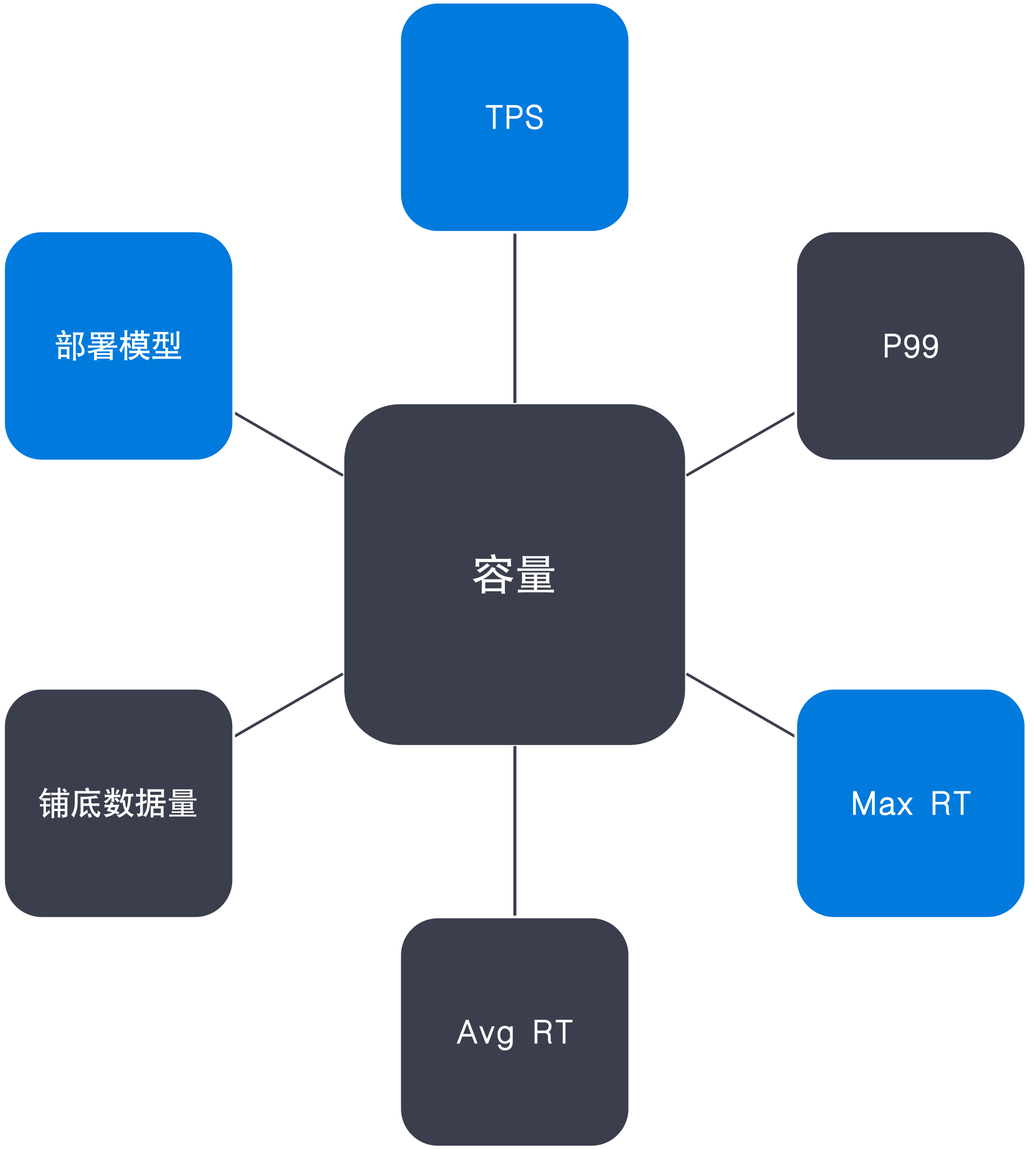
等比例缩放后：TPS应该线性变化。P99、ART不应该高于生产环境

例如，路由服务在生产上双中心40台，数据库约40亿数据。在测试环境完全1:1复刻生产环境。

一直测试到出错为止，无需考虑CPU、内存使用率。

如果系统未上线：则部署模型、铺底数据应按照设计的极限进行等比例缩放。无需考虑CPU、内存使用率

测试后的结果，作为制定指标的基础依据。

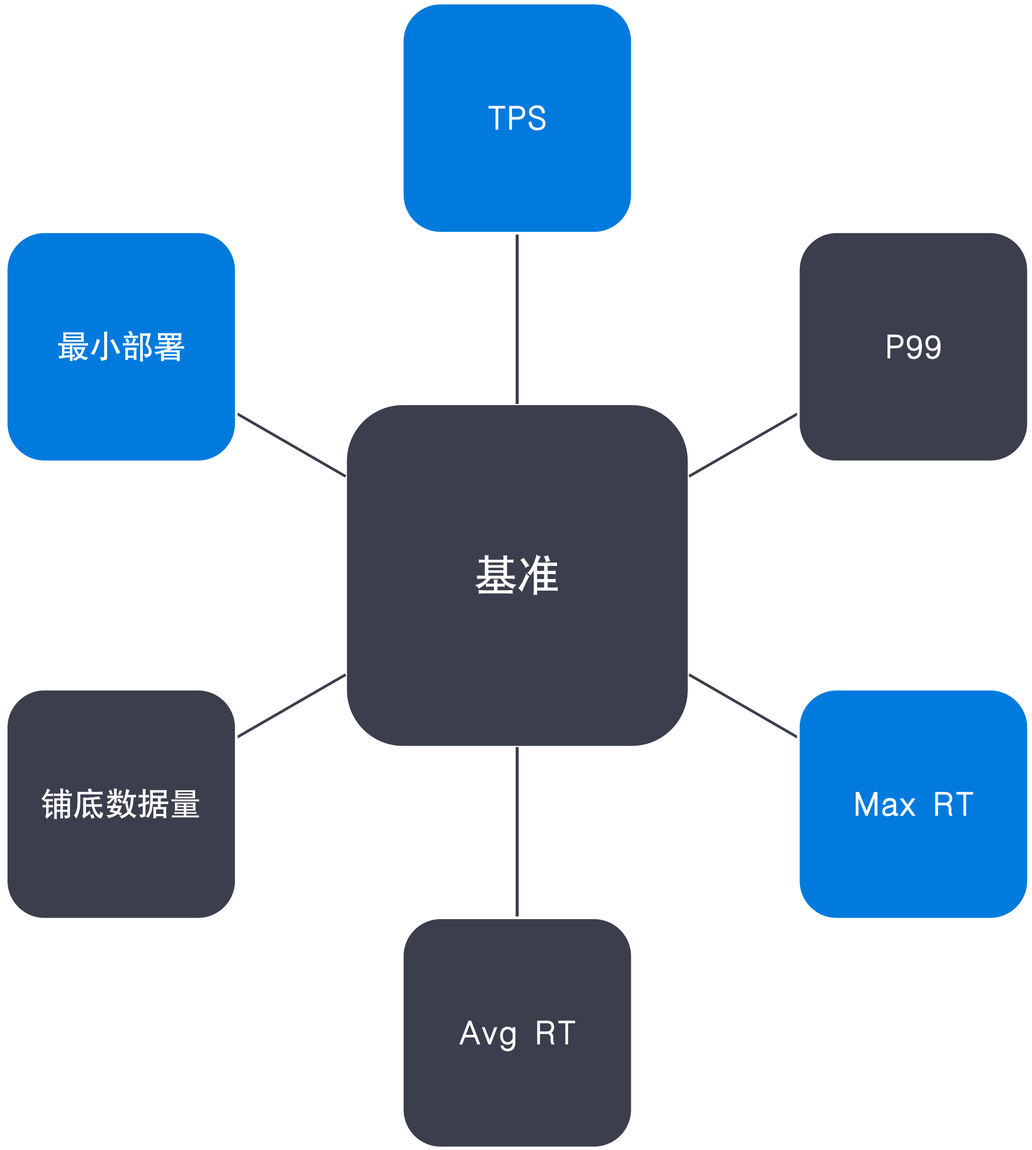


制定测试指标-容量

基准测试采用最小部署模型(MDU)，铺底数据按照容量测试方案，从1并发测试开始进行梯度压力，一直压到开始出现报错，或者监控到资源使用率超过75%（经验值）

观察在每个梯度下，TPS，P99，ART，MAR的表现，找到其性能拐点。则该拐点为基准测试结果。

作为制定指标的参考，并作为未来性能、测试的基准。



压力模型示意图

无负载区：

RT无变化，TPS可快速升高

攀升区：

RT略微升高，TPS依旧可快速攀升；

满载区：

RT逐渐升高，TPS增速放缓（拐点）

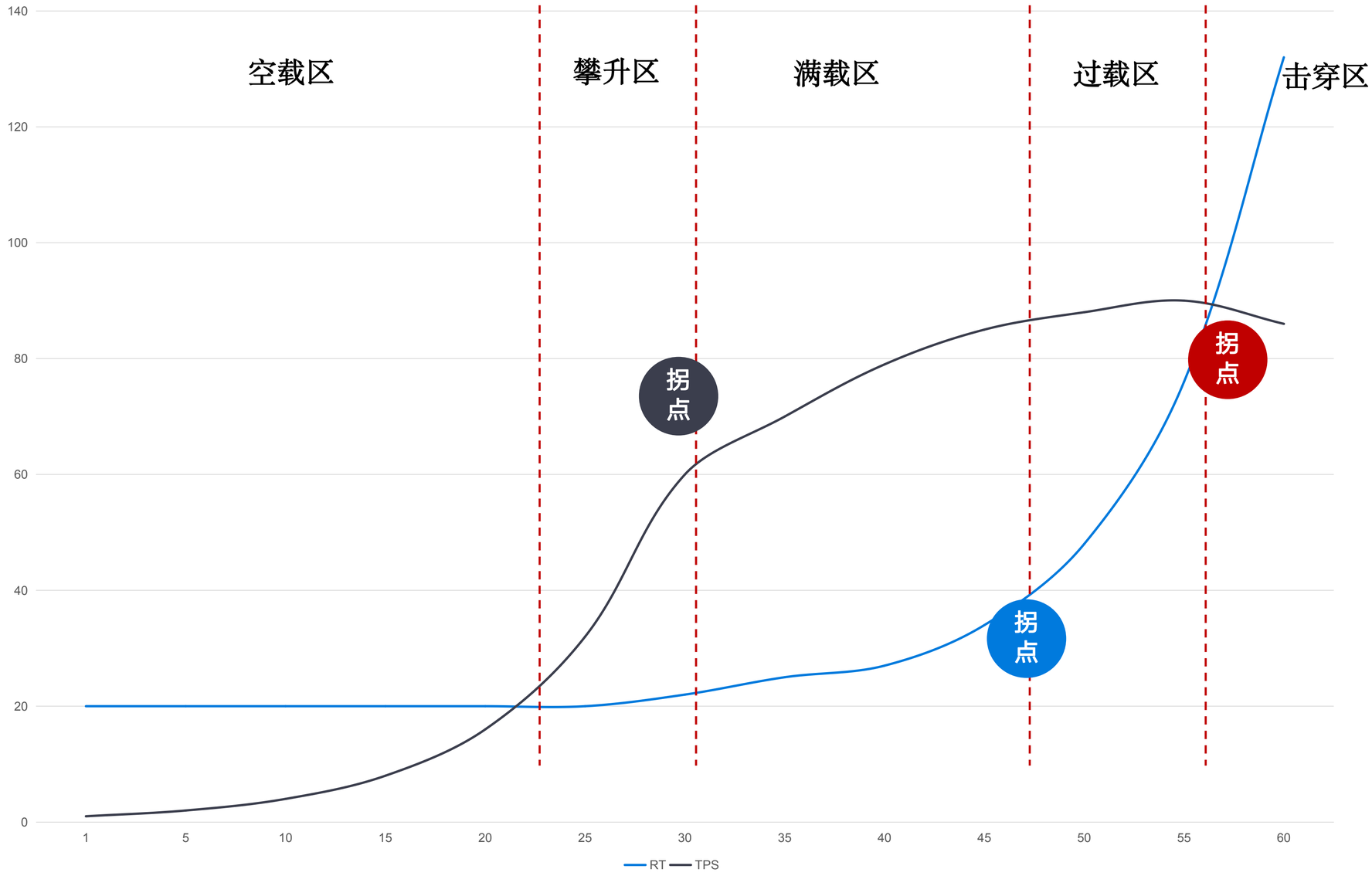
过载区：

RT快速攀升（拐点），TPS缓慢增长

击穿区：

RT快速攀升，TPS略微下落（拐点）

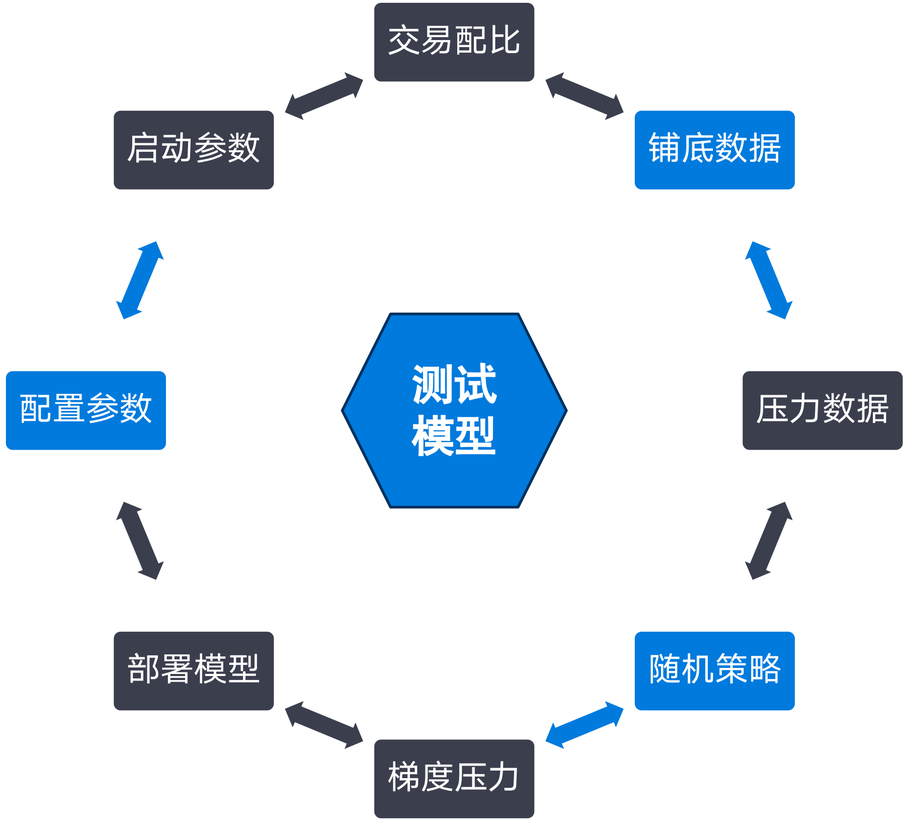
TPS = 并发 * 1/ART



测试模型示例

测试模型指的是：交易配比如何，铺底数据用哪些，压测数据用哪些，每轮迭代如何选择数据，压力发多大，部署多少节点，配置参数是什么？启动参数是什么？每一轮测试，只改变一个变量！具体的测试案例还会有更多的维度，也需要编排进测试模型中

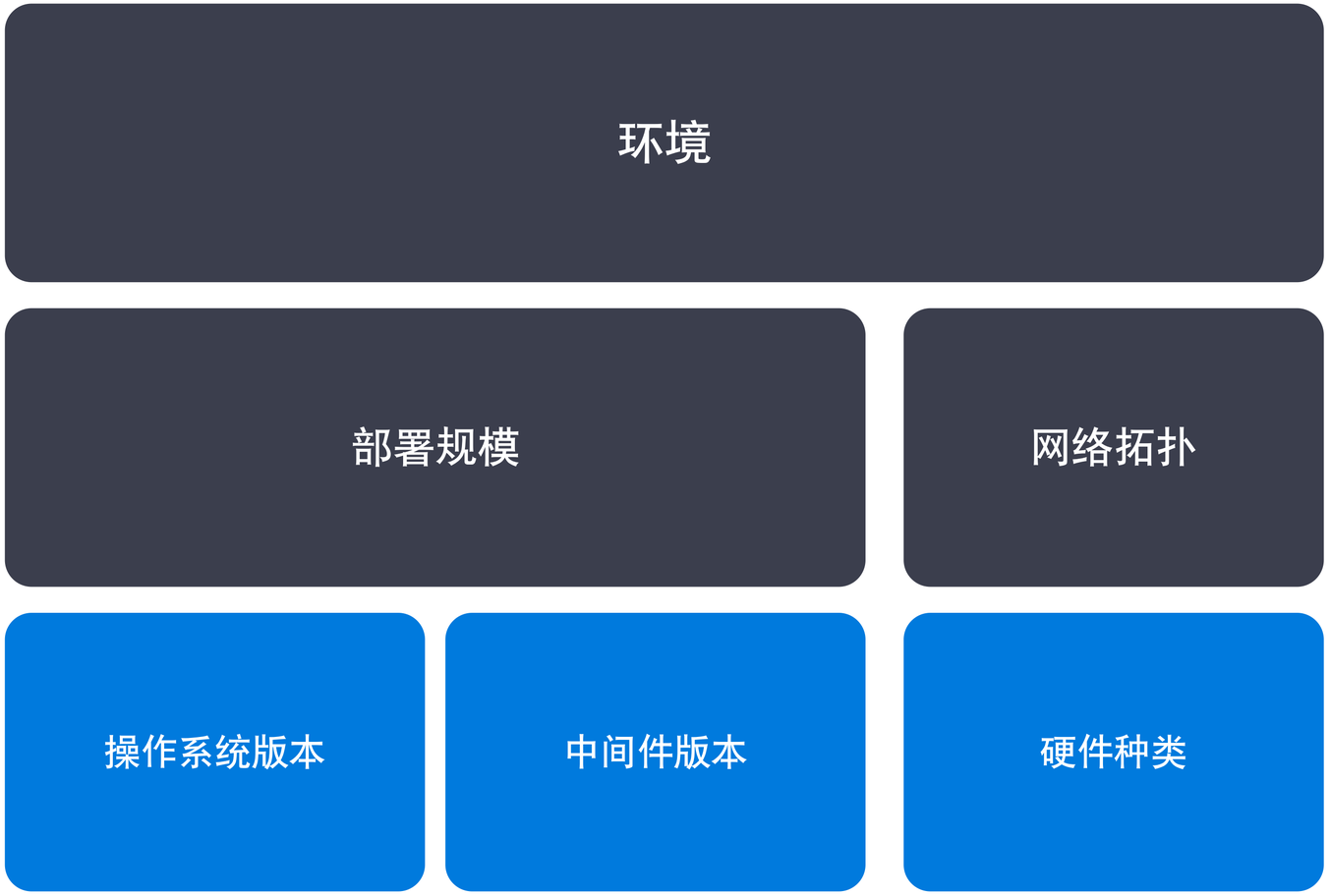
模型	交易配比	铺底数据	压力数据	随机策略	梯度压力	部署模型	配置参数	启动参数
模型1	查询、新增、修改 8:1:1	铺底1000万条数据，每条1K字节	每条**404**字节	数据进行shuffle，每次迭代选择唯一	1-100并发，每次增加10并发，每梯度10分钟	30Demo-6Proxy-9redis-cluster	默认参数，双中心同步	JVM参数，logfree
模型2	查询、新增、修改 8:1:1	铺底1000万条数据每条1K字节	每条1K字节	数据进行shuffle，每次迭代选择唯一	1-100并发，每次增加10并发，每梯度10分钟	30Demo-6Proxy-9redis-cluster	默认参数，双中心同步	JVM参数，logfree
模型3	查询、新增、修改 10:0:0	铺底1000万条数据每条1K字节	每条1K字节	数据进行shuffle，每次迭代选择唯一	1-100并发，每次增加10并发，每梯度10分钟	30Demo-6Proxy-9redis-cluster	默认参数，双中心同步	JVM参数，logfree



环境准备

在非功能测试中，约有1/2的缺陷是由于环境导致的

- 首先需要保证硬件、操作系统版本、中间件版本与目标（生产）环境尽量对齐，否则结果会严重失真。
- 其次，网络拓扑必须与目标（生产）环境尽量对齐。若无法对齐应该保证网络拓扑尽量简单，可以尽力排除网络因素的干扰。
- 再次，部署规模需要根据测试模型等比例缩放，但是节点的比例应该保证相同。
- 最后，尽量保证一套环境独享，如果无法保证独享，至少保证分时独享，否则多个测试同时运行会导致结果是真严重。



部署配置

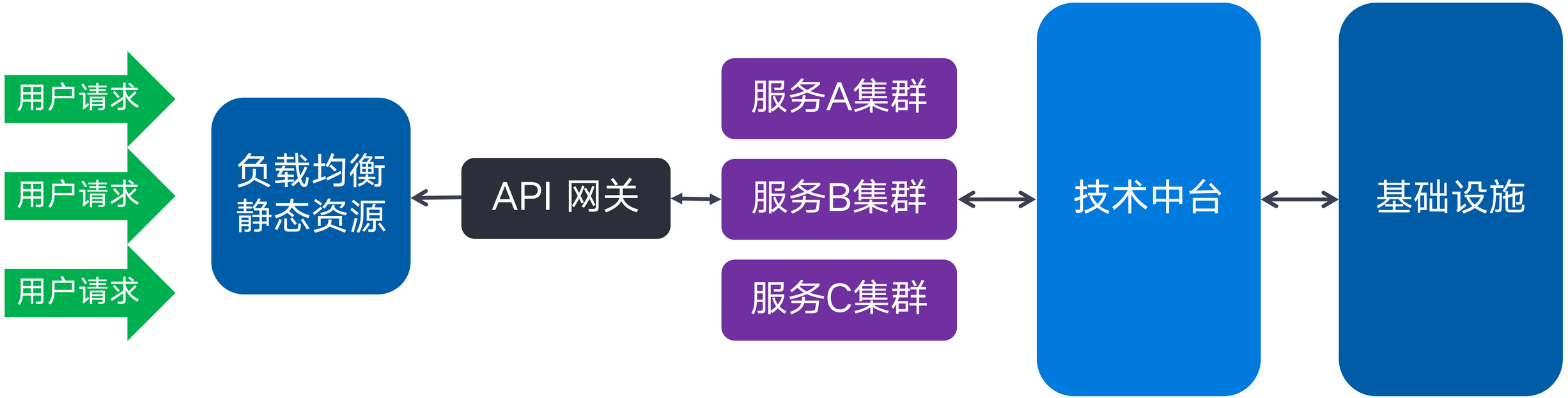
在非功能测试中，约有1/6的缺陷是由于部署导致的

- 首先要保证应用侧部署模型与目标环境上的模型是等比例缩放的，端到端的链路模型应该相同。
- 其次，Foundation层的数据和中间件部署模型应该与生产目标环境的部署模型等比例缩放。
- 再次，应用侧与foundation的比例，尽量与目标环境保持一直，或者foundation不应该成为测试瓶颈。
- 最后，保证启动参数、配置参数应与目标环境对齐。对于数值、地址等跟环境相关的参数应适配测试环境。

2. 面向性能的架构

性能

各个环节都可能会出现性能问题，进而进行性能优化。



性能的关注点

高并发用户（Concurrent Users）

高吞吐量（Throughout）

低延迟（Latency）

~ 容量？？并发？？

性能的副作用

如果实现了高性能，有什么副作用呢？

1、系统复杂度 x10以上

2、建设与维护成本++++

3、故障或 BUG 导致的破坏性 x10以上

性能的经验

- 吞吐与延迟： 有些结论是反直觉的，指导我们关注什么
- 没有量化就没有改进：监控与度量指标，指导我们怎么去入手
- 80/20原则：先优化性能瓶颈问题，指导我们如何去优化
- 过早的优化是万恶之源：指导我们要选择优化的时机
- 脱离场景谈性能都是耍流氓：指导我们对性能要求要符合实际

性能的关注点

- 业务系统的分类：计算密集型、数据密集型
- 业务处理本身无状态，数据状态最终要保存到数据库
- 一般来说，DB/SQL操作的消耗在一次处理中占比最大
- 业务系统发展的不同阶段和时期，性能瓶颈要点不同，类似木桶装水

前端性能

1、连接数优化

2、资源优化

3、开gzip

4、小文件合并

5、PageSpeed工具可以检测前端性能

后端性能

业务逻辑优化

框架优化

中间件优化

线程池优化

。。。。。。

数据库性能

数据库参数优化

SQL查询优化

数据批量插入优化

缓存优化

。。。。。。

网络性能

操作系统TCP参数优化

Netty相关优化

操作系统参数优化

内存相关参数

修改/etc/sysctl.conf 文件添加如下配置后使用sysctl -p生效：

```
//调大共享内存
kernel.shmmax = 4294967295
//降低swap几率，或者关闭掉swap：直接运行swapoff命令
vm.swappiness = 0
//允许分配所有物理内存
vm.overcommit_memory = 1
```

命令行中关闭透明大页以降低分配内存对性能的影响：

```
$ echo never > /sys/kernel/mm/transparent_hugepage/enabled
$ echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

配额相关参数

使用`ulimit -n 100000`修改。

再修改/etc/security/limit.conf以提升资源限制阈值：

```
* soft nofile 65535
* hard nofile 65535
```

网络相关参数

同样在/etc/sysctl.conf文件中添加如下参数修改网络相关配置。这里最重要的连接数、半连接数调大到65535，以容纳更多的客户端连接。

```
net.core.somaxconn = 65535
net.core.netdev_max_backlog = 65535
net.ipv4.tcp_max_syn_backlog = 65535
net.ipv4.tcp_fin_timeout = 10
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
net.core.wmem_default = 87380
net.core.wmem_max = 16777216
net.core.rmem_default = 87380
net.core.rmem_max = 16777216
net.ipv4.tcp_keepalive_time = 120
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_keepalive_probes = 3
```

中间件性能

MQ优化

Redis优化

ES优化

。。。。。。

性能测试

基准测试

通常的过程是单个事务单用户运行一定时间或者多次重复一个事务获得平均事务响应时间（ART）和每秒事务数（TPS），作为性能测试后续压测的参考基准。

负载测试

负载测试是典型的性能测试，对应用程序施加负载直到达到目标，但通常不会进一步施压。负载测试最接近真实的使用场景，接近真实生产环境的性能压力。

压力测试

压力测试和负载测试的目的完全不同，压力测试会导致应用程序或者部分支撑硬件的崩溃，压力测试的目的是确定硬件的支撑大小和上限。

稳定性测试

疲劳测试也叫稳定性测试,目的是找出那些可能只出现在一段较长时间里的问题。一般测试持续8-12小时以上。

性能测试

TPS（每秒交易数）：要求为日峰值交易量的80%发生在当天的峰值处理时间（20%的时间，约2小时）之内，即：

投产时：峰值TPS=60万×80%/（2×3600）=66.6笔/秒

一年后：峰值TPS=160万×80%/（2×3600）= 177.7笔/秒

三年后：峰值TPS=500万×80%/（2×3600）= 555.5笔/秒

ART(交易响应时间)：例如，90%的交易耗时小于1.5秒，99%的交易耗时小于2.5秒；报表查询耗时不高于5秒。

并发用户数：1.经典公式：平均并发用户C=nL/T，n是登录会话数，L是session的平均时长，T是交易的时间长度；并发用户峰值C'=C+3* $\sqrt{C}$

2.经验值，使用客户数，同时在线用户数，并发用户数=系统最大在线用户数的5%-15%

交易成功率：例如，核心交易类系统成功率≥99.99%。运营支撑类成功率≥99.9%。

性能测试

分析方法	适用场景	举例
直接观测法	新增测试	观察TPS，观察RT，MAXRT
因果分析法	新增测试、生产问题复现	梯度压力观察网关TPS增幅和响应时间变化，一次观察压力与性能的关系
单变量法	生产问题复现	依次调整JVM参数、压力、平台参数。观察每次调整后，TPS，RT的变化
相关性分析法	生产问题复现、技术预言	同时调整部署结构、平台参数、JVM参数，观察系统的P99，TPS和系统日志。综合分析这些变化量与性能之间的关系
统计分析法	生产问题复现、技术预言	采集所有被测节点的IO使用率、GC平均耗时，数据库操作时间。结合不同压力情况下的P99。统计这些被测结果与P99之间的关系。
时序分析法	技术预研、新增测试	例如经过一段时间的观察，可以看到TPS的波形。发现RT和GC的上升是同时上升，TPS本来是线性增长，而RT和GC上升后TPS增速下降。
对比分析法	问题复现	对比多个版本，多个时段找到差异，并根据差异分析根因。

直接观察法

因果分析法

单变量分析法

相关性分析法

统计分析法

时序分析法

对比分析法

性能测试


压力测试工具
LoadRunner / JMeter
监控工具
nmon、nmon analyser、prometheus
日志分析工具
elasticsearch
链路追踪工具
skywalking
单元测试工具
Junit、jmockit
接口测试工具
Postman、JMeter
自动化测试工具
自动化测试平台（软研提供）
混沌工程工具
ChaosBlade
Linux工具
top、vmstat、pcstat、perf 、iostat、sar 等
网络工具
netstat（网卡饱和度）
数据库工具
postgresql 配套工具（报告生成）
JVM工具
jprofiler（热点代码分析）、JFR（Flight Recorder，热点代码分析）、jstack（javacore收集）、jstat（查看 gc 信息）、jmap（ Heapdump 收集）、jvisualvm（查看线程运行过程中是否有锁）、jconsole、Eclipse Memory Analysis Tools（堆内存文件分析）、IBM Pattern Modeling and Analysis Tool for Java Garbage Collector（gc分析工具）

全链路压测

稳定性模块详细讲

第五周-作业实践

（选做）

1、思考一下自己负责的系统，都在哪些非功能方面有一定的要求。

2、思考一下自己负责的系统，有什么地方的性能是可以改进的。

Q&A

谢谢各位

kimmking@163.com