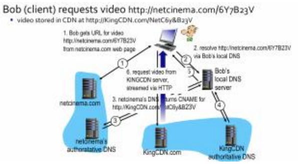
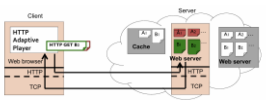
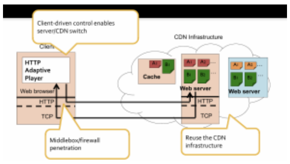
软件架构
大型网站架构发展历程
- 大型网站软件系统特点：高并发，大流量；高可用，海量数据；用户分布广泛，网络情况复杂；安全情况恶劣，需求快速变更，发布频繁，渐进式发展
- **架构演化价值观**：1. **核心价值：随网站所需灵活应对** 2. 驱动发展的主要力量是网站业务发展
- **架构设计误区**： 1. 一味追寻大公司解决方案 2. 为了技术而技术 3. 企图用技术解决所有问题
- 发展历程：1. 初始阶段：程序，文件，sql 都在一台服务器上->2. 应用服务和数据服务分离(不同特性的服务器承担不同的服务角色，**网站的并发处理能力和数据存储空间都得到了很大改善** but IO 还是慢)->3. 用缓存改善网站性能（减少数据访问压力，但是单一应用服务器可以处理的请求连接有限->应用服务器成为高峰期瓶颈）->**4. 使用应用服务器集群改善网站的并发处理能力**，来自用户浏览器的访问请求分发到应用服务器集群中的任何一台服务器上，服务器的负载压力不再成为整个网站的瓶颈（负载过高的时候，部分数据库读操作（缓存访问不命中、缓存过期）和全部的写操作需要访问数据库，瓶颈变成数据库）->5. **数据库读写分离**（主从数据库：写数据的时候访问主数据库，通过主从复制机制将数据同步到从数据库，读数据的时候从从数据库读入）->6. 反向代理+CDN 加快响应，反代和 CDN 基本原理都是缓存，加快用户访存，减少后端服务器负载，CDN in 网络提供商机房，反代 in 网站中心机房->7. 分布式文件系统，分布式数据库系统->8. NoSQL 和搜索引擎->9. 业务拆分和分布式系统->10. 分布式服务
流媒体架构
- 流媒体是网络带宽的主要消耗者
- 挑战：scale:要reach到很多用户；异质性：用户上网环境区别大
- solution：分布式应用架构
- 推送存下来的视频：（video server->Internet->client）主要挑战：带宽随网络阻塞情况波动；丢包
  - 其他挑战：要求连续放映（但是网络带宽不定，需要 client-side buffer; solve：用 client-side buffering 和放映的延迟中和网络中延迟的波动）；客户交互；重传视频包等要求

- DASH（Dynamic Adaptive Streaming over HTTP）：
  - 服务器：视频文件分块->每个块以多种不同的比特率进行编码->不同比特率的编码存储在不同的文件中->文件在多个 CDN 节点处复制->用清单文件保存不同块的 URL
  - 用户：定期估计服务器和用户之间带宽->问询清单文件，每次获取一个块的URL->选择当前带宽下可持续的最大编码率，可以在不同时间用不同编码率，或者通过不同服务器选择
  - 用户端具有“智能”：何时请求 chunk 请求什么样的编码率 从哪里请求chunk
  - 推送视频==编码+DASH+playout buffering
- CDN（内容分发网络）
  - 挑战：怎么同时把内容（成千上万的视频信息）推送到成百上千不同用户那里。单个大的超级服务器不work，原因：它失败整体就gg了；可能产生拥塞；到达很远的用户路径长
  - 策略2：在地理位置不同的多个地方存视频的多个副本->CDN
  - CDN:在CDN节点存了内容的多个副本。subscriber 请求内容，服务提供者返回 manifest->根据 manifest，客户可以在最高比特率下获取内容，如果网络堵塞，可以从不同服务器获取或者换帧率获取
  - OTT 挑战：（Over the Top）:每个CDN node 放哪些信息，用户从哪个 CDN node 以什么帧率获取信息





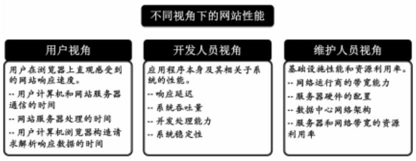


Client-Server and P2P
- Client-Server: Client 端发出请求，Server 端相应请求，这种模式指定Client 和 Server 端的可复用行为，但是不指定该如何实现 client server
- P2P：所有实体同时作为 client 和 server，好处是比较灵活，但是难以实现同时胜任两者工作的软件
  - 无需一直开服务器，端系统直接进行交流，对等节点向其他节点请求服务也提供服务，（自我扩展性：新的对等节点提供新的能力和需求），但是对等节点连接不是永久的，IP地址也会变化，所以相对复杂
派发文件
如何把大小为F的文件从 server 发到 n 个 peer
- client-server: server端：连续上传n个文件副本，用时 NF/U_s; client 端：每个 client 下载文件样本，设最小下载率为d_min，需要时间为F/dmin
- 所以用这种方式派发文件的时间： D_c-s> max{NF/us,,F/dmin}
- P2P: server: 至少上传一份文件样本，用时F/U_s; client:min client download time: F/dmin; clients: as aggregate must download NF bits，最大上传时间和最大下载时间都是U_s+sigma(U_i)，所以D_P2P>=max(F/U_s,F/dmin,NF/(Us+sigma(U_i)))
- 比特币是用 P2P 形式传递的

Software as a Service(SaaS) and Cloud Computing
- SaaS:
  - 数据在云端存储；丢失设备不会丢数据，组间易于交换/协作
  - 一个软件副本，单一硬件/操作系统环境，兼容性有保障，但是不可以比如说对1%用户去 beta test,升级的时候就重新部署就行
  - 便于快速相应用户反馈，和尽快实施改变，符合敏捷开发
  - SaaS对硬件基础的需求：交流能力；可扩展性和弹性；稳定性
  - 集群上服务的优势：
    - 集群上，大量普通以太网交换机连接的普通计算机
    - 可扩展性；便宜，通过冗余更具有可靠性，数千台服务器只需少数操作员
- Utility computing(2007)
  - Buy computing, storage, communication by the hour
  - Public/Utility Cloud Computing(由专家维护的基础设施可以降低total cost of ownership)
软件设计五个原则
- 软件架构聚焦于系统组件如何互相交互
- 软件架构模式：子系统如何连接在一起符合程序的功能性和非功能性需求
- 软件架构定义：系统的基本组织结构体现在其组件中，以及这些组件与彼此及环境的关系，并且受到指导其设计和发展的原则的约束。
- 软件架构重要性：
  - 系统的骨架，对于一个工程，没有对的架构，只有适配度最高的架构
  - 影响质量属性，"enables or inherits qualities"
- 目标：高性能，高可用性，高扩展性（容量，变化），高异质性，高安全性，低成本
- 软件设计：定义系统或者组件的架构，成员或者接口
- 设计过程：创造性设计更多是同时开发和精细化问题的结构和解决办法的思路，伴随着不断的对问题和解决方法的分析，假设和评估
  - 设计的逻辑：寻找替代方案；设计的形态：层次结构；流程作为风格的决定性因素
- 设计结果：由宏观更加具体化，定义实现一个系统的细节（架构，子系统，交流方式，组件，内外在接口）
五个原则
KISS: keep it simple stupid

简化接口，模块间依赖解耦
结构和配置简化
模块化
每个模块都简单到可以让人完全理解。
一个模块的变化，并不影响另外一个（模块对改变解耦）
一个模块的修改，尽量别改接口
一个模块改好了可以和其它新或老的模块一起工作。这也是现在敏捷开发和持续交付的基础。
原则：把复杂系统分割成**强内聚松耦合**的简单模块
好处：更方便规划发展；软件增量可以被定义和交互；更加适应变化；更高效测试和debug;便于长期并且没有副作用的运维；合适模块数可以减少开销
选择合适的模块数，总开销随模块数量增加先减后增
各司其职
- 分而治之：很多问题间有强依赖关系，可以独立的聚焦于问题的不同方面，比如质量，开发过程，运行约束等
- crosscutting concerns: 横切多个关键程序的concerns（就像 assert 或者 log）所以改变concerns的时候 需要在多个地方都改变。->solution: 每个函数元素做且只做一件事
- 各司其职：系统化思考来解决复杂问题
  - 分解：复杂问题分解为小问题；小，简单，可控
  - 合成：子问题融合成系统化解答
Design for change
- 背景：变化不可避免：e.g 加功能，fix，加强用户体验，新科技，系统融合

- 策略：把可能的变化当成系统设计的一部分；把可能的变化锁定在系统的一部分，所以到时候只用修改部分系统
- 为了扩展和压缩来设计系统：预判变化，不是设计一个程序，而是设计a family of programs(不同的硬件，输入输出格式，资源使用量，数据集大小，事件频率，只用部分的功能)
- 可变性：改变的开销，接受变化的难易程度
可复用
"with and for reuse"
- 软件复用优势：降低生产运维开销；质量提升；更快交付
- 复用的东西：设计，代码，测试；问题和设计的解决方案；方法和过程；或者复用用已实现的成熟开源组件
- 使用 product line:
  - 软件产品线（或应用程序系列）是一组软件密集型系统，它们共享一组公共的、受管理的特性，满足特定市场细分或任务的具体需求，并且按照规定的方式，从一组共同的核心资产中开发而来;创造新应用的时候 核心特性被复用，每个程序也是核心特性的一个实例化
  - 核心+定制+管理；过程：核心特性积累+产生产品+协调
架构设计
- 架构设计的主要目的是解决软件系统复杂度带来的问题
- 软件设计目标也可能带来软件的复杂度
- 架构设计三原则：合适优于业界领先；简单优于复杂；渐进式优于一步到位
网站高性能架构
- 性能：系统架构进步的 driving factor



- 性能指标：响应时间，性能计数器（被服务器/操作系统用，看系统负载（正在被CPU执行和等待被执行的进程数目总和），对象与线程数，内存使用，CPU使用，磁盘与网络I/O），并发数（系统能同时处理请求的数量），吞吐量（单位时间内系统处理的请求数量，体现系统的整体处理能力）

architecture patterns
高性能关系数据库集群
- 读写分离，将数据库读写操作分散在不同的节点上，分配机制：程序代码封装/中间件封装
- 按照业务模块将数据分散到不同的数据库服务器（单台数据库server可以支撑10万用户量级的服务）,缺点：操作复杂，代码工作量++，计算资源++
- 分表：垂直分表：某些不常用且占了大量空间的列拆分出去 水平分表：适合表行数特别大的表
- 用 NoSQL(Not only SQL)
  - 文档数据库（e.g MongoDB）：no-schema，可以存储和读取任意的数据，一般用json格式
    - 优势：新增字段 无需修改表结构； 历史数据无新增的字段时，返回空值，不会出错。复杂数据可以用json表述
    - 劣势：不支持 transaction ，不直接支持 join
  - 全文索引数据库
    - 用倒排索引建立单词到文档的索引
高性能缓存架构
缓存系统要解决：缓存穿透，缓存雪崩，缓存热点
- 缓存穿透
  - 对于缓存和数据库中都没有的数据，而用户不断发起请求，导致数据库压力过大
  - 接口层增加校验，比如用户鉴权，非法值的检验

  - 从缓存取不到的数据，在数据库中也没有取到，这时也可以将keyvalue对写为key-null，缓存有效时间可以设置短点，如30秒。这样可以防止攻击用户反复用同一个id暴力攻击
- 缓存雪崩
  - 当缓存失效（过期）后引起系统性能急剧下降的情况。缓存过期的时候，业务系统需要重新生成缓存，所以再次访问存储，这个过程需要几十到上百毫秒，期间可能也会接到成百上千请求，而且这些请求的线程都不知道有另一个线程在生成缓存，所以所有线程都会去生成缓存从而对存储系统压力过大，压力又会拖慢整个系统
  - solve:1. 过期时间随机设置，防止同一时间大量过期 2. 热点数据分散布置在不同sql中 3. 设置热点数据永不过期
- 缓存热点
  - 虽然缓存系统本身的性能比较高，但对于一些特别热点的数据，如果大部分甚至所有的业务请求都命中同一份缓存数据，则这份数据所在的缓存服务器的压力也很大
  - solve:
    - 复制多份缓存副本，将请求随机分散到多个缓存服务器上，且不同缓存副本的过期时间不要一样
负载均衡技术
DNS 负载均衡，硬件负载均衡，软件负载均衡
典型架构：3种技术互补
算法:
1. 轮询：服务器不宕机，服务器和负载均衡器不断连
2. 加权轮询：服务器性能有差异
3. 负载最低优先：考虑连接数，请求数，CPU 负载，I/O 负载；但是这个需要采样
4. 性能最优优先：考虑处理速度，但是这个需要采样

5. Hash：对源地址或者ID hash
Web 前端性能优化
浏览器访问优化
减少请求数，启用浏览器缓存，启用压缩，减少cookie传输，CSS 放在页面最上，Javascript放在最下
CDN 加速
数据缓存在离用户最近的地方
反代
位于网站机房一侧，代理网站Web服务器接收HTTP请求
应用服务器性能优化
- 异步操作，采用消息队列使得调用异步，可以用来削峰
高可用
错误不可避免
- 可用性：系统去掩盖或者修补失败的能力，来使得一定时间范围内，积累性的服务不可用的时间小于所需要的值，可以用 avg(time between failures)/(avg(time_between_failures)+avg(time_to_repair))来算
- 容错性：在硬件/软件/系统可靠性低的时候分布式系统保持可用性，容错性本身不应该涉及用户或者admin，可以由恢复（依赖于同等的服务），组件冗余来实现
- CAP：在分布式系统中，以下三者只可能保证其中之二：Consistency(the most recent write or an error)，Availability(a non-error reasonable response,但是可能不是最recent的)，Partition Tolerance（node 之间的网络可能丢包，这种情况下还可以保证可用）；P 一定需要的 所以是 CP/AP组合

  - CP 和 AP 区别在于，当分区现象发生的时候，两个节点不能同步，CP 返回的是 error，AP 返回的是更新前结果
- FMEA：故障模式和影响分析
  - 初始架构设计图->假设架构中某个部件发生故障->该故障对系统功能的影响->判断架构是否需要优化
- 高可用存储-双机
  - 类型：主备复制：双机变更频率低，像学生管理系统这样，正常情况下主机写，读，主机故障的时候从机写/读；主从复制：适合写少读多，比如论坛这样，是从主机写&读，从机读；主主复制：临时性，可丢失，可覆盖，两个主机都可以读写
  - 主备切换：状态监测和状态传递，状态判断，切换时机，切换策略，自动程度
  - 数据集中式集群：备机对主机状态判断不一致的情况；主机故障后，如何决定新的主机
  - 数据分散集群：可伸缩性更好
  - 高可用存储：数据分区：不同分区地理位置不同，每个分区有一部分数据，来规避地理级别的故障
  - 对称集群：负载均衡；非对称集群：不同角色服务器承担不同职责
  - 业务高可用的保障：异地多活架构：对于关键应用，用户无论访问哪一个地点的业务系统，都期待能够得到正确的业务服务。即使某个地方业务异常的时候，用户访问其他地方正常的业务系统，也能够得到正确的业务服务

高可扩展
可扩展性：通过修改和扩展，不断地让软件系统具备更多的功能和特性，满足新的需求或者顺应技术发展的趋势
基本思想：拆，将原本大一统的系统拆分成多个小规模部分，扩展时可以只改其中一部分。流程>服务>功能，可以面向流程/服务/功能拆分
- 面向流程拆分对应分层架构，like Client-Server划分整个业务系统或者MVC(Model层-View层-Controller层）划分单个业务子系统；还有逻辑分层架构，按职责从顶向下依赖

  - 特点：1. 各层之间差异足够清晰，边界足够明显；2. 隔离关注点，每层组件只会处理本层逻辑 3. 层层传递，相邻层依赖，降低整体系统复杂度
  面向服务拆分对应 SOA和微服务
  - SOA:Service Oriented Architecture
    - 一组独立的、可组合的服务，可以作为黑盒进行组合。
    - 特点：类似于 client-server 但是组件之间松耦合，而且是用独立接口。且 SOA 从接口定义出发，把整个程序建造为一个接口，接口实现，接口调用的图拓扑，"interface-oriented architecture"
  - **微服务架构**
    - 定义：微服务架构风格是开发单一应用程序的一种方法，将应用程序作为一组小型服务来构建，每个服务都在自己的进程中运行，并通过轻量级机制进行通信，通常是HTTP资源API。
    - 特点：这些服务围绕业务能力构建，并可以通过完全自动化的部署机制独立部署。
    - 单体应用和微服务应用的区别：
      - 单体应用把所有功能塞到同一个流程中，然后把这个单体放在不同的server上面；微服务架构把每个服务元素放到一个单独的service里面，把这些service跨server分发，并且在需要的时候进行冗余
      - 模块化：单体：不同模块放在同一个过程中；微服务：不同模块放在不同的过程中
      - 数据库：单体：单一数据库；微服务：按照应用进行数据库划分
      - 容器化：在共享的操作系统上面运行独立服务，就像（需要的lib和设置，安全性，程序环境这样），使得微服务成为可能
      - 特点：是一种实现持续交付/部署的达到目的的手段；通过发布的接口使用服务作为构建块（组件）；围绕业务能力组织；开发团队对生产中的软件负全部责任；端点智能化，管道简单化，程序语言的去中心化控制
      - 实现特点：数据库去中心化：每个服务维护自身的数据库，利用 Polyglot Persistence;基础设施自动化；design for failure;渐进化发展

- 4代微服务架构：容器编排->服务发现和容错->边车和服务网格->无服务器架构
- 面向功能拆分对应微内核架构，这些架构也可以组合使用
  - 微内核架构：微内核架构（Microkernel Architecture），也被称为插件化架构（Plug-in Architecture），是一种面向功能进行拆分的可扩展性架构，通常用于实现基于下载安装的应用。比如 bowser，vscode,unix 系统这类的。最终目标：更快送达更好的软件

软件需求

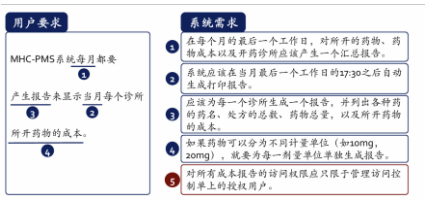
- 定义：需求是对应当执行任务的规范说明，描述系统行为特性或者属性，是一种对系统开发进程的约束。开发软件系统的前提是明确用户的期望和要求，即是软件需求（考虑到用户的目标和期望，对开发人员进行规范说明）
- 需求工程
  - 定义:用工程的理念和方法来指导软件需求实践，它提供了一系列的过程、策略、方法学和工具，帮助需求工程师加强对业务或领域问题及其环境的理解，获取和分析软件需求，指导软件需求的文档化和评审，以尽可能获得准确、一致和完整的软件需求，产生软件需求的相关软件制品
  - 需求工程过程：可行性分析（出可行性报告）->需求获取和分析（出需求建模）<->需求规约（进行需求规格说明）<->需求确认验证
  - 需求工程特点：1. 知识密集型工作，需要交叉多学科知识 2. 多方共同参与 3. 需求获取的多种形式和源头 4. 持续迭代和逐步推进
- 方法学：
  - 抽象：结构化数据和数据流抽象(1970s)->面向对象抽象（1990s）
  - 建模：建模语言
  - 分析：精化和分析软件需求，循序渐进获得需求细节，逐步获得详细软件需求。提供策略和手段

需求获取

以用户需求为驱动的产品设计，理解探求用户目标，提取必要系统功能，“有所为有所不为”

过程

1. 识别用户类别：belike:用户群访问权限或安全级别;在业务操作中执行的任务;使用的特性;使用产品的频率;在应用领域和计算机专业技能经验，使用平台，语言，和系统交互方式等
2. 确定于系人代表：干系人是指 积极参与项目的某个人，群体或者组织，可能会受项目过程和结果的影响或者影响项目的过程或者结果，可能在 开发组织，开发组织以外，项目组
3. 确定决策者：设定一定的决策规则，在软件项目中，需要在关键路径上做决定，或是解决一些冲突，接受（或拒绝）某个需求变更或者批准一组即将发布的需求，这有利于决定工作方向，保证项目进度
4. 需求达成共识：不能盲目接受客户所有请求，对需求/其中某部分达成一致是客户-开发关系的核心。客户确认：需求描述了他们的需要；开发人员确认：理解需求并且可以实现；测试人员确认：需求可以验证；管理层确认：需求满足业务目标。需求基线：一个特定时间点的需求快照，是作为后续开发基础的一组需求



需求建模

泳道图

理解成有主体的流程图就行了，看考试复习课件上的那个样题 注意它查看本人的当前考勤状态以及历史信息的箭头方向
流程：操作步骤，角色协同，事务，还有判定条件

判定表和判定树

条件组合：功能需求是在所有可能条件的组合下对系统响应动作的描述，忽略条件组合可能引起功能遗漏
这个看P48页图 注意美观 不难画

用例图

UML：可视化，描述，构造和文档化软件密集型系统的各种产品，支持不同人员之间的交流
Unified:可视化标准化建模表示 Modeling：可视化建模 language：图形化语言
用例：系统和某外部角色之间的一系列交互，角色：和系统交互执行某个用例的人，用例图：对用户需求的概要性可视化表达
1. 确定角色，分析和系统间交互活动 2. 分析业务过程，场景概括为样例 3. 识别系统外部事件，关联到角色和用例
2. 可以用用例场景说明来补充详细信息，belike:1. 唯一的ID和简洁的名称（指明用户目标） 2. 简短文字说明，用来描述用例的意图 3. 开始执行用例的触发条件 4. 用例开始的零个或多个前提条件 5. 后置条件，描述用例成功完成后的系统的状态 6. 一个有编号的步骤序列，展示了角色与系统之间的交互顺序

用户故事和顺序图

用户故事表述：角色+活动+业务价值
用例更像是史诗（Epics），一组围绕一些共同目标的用户故事
用户故事的问题：缺乏语境，尝试覆盖所有和目标接近的base，没有对接下来工作的前瞻性

顺序图

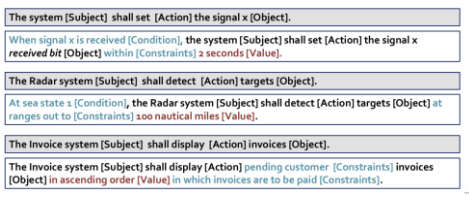
定义一个用例中单场景的事件序列，看物体间的交互和交互发生次序
图中，从上往下：时间顺序，从左往右：不同物体
看 P69 图，就像TCP 三次握手或者IPsec 密钥交换的图那样画

画图方法：把句子中所有谓语提出来，确定主语宾语，主语画一条线指向宾语，箭头上面标谓语，看 P73 图

需求要求

软件需求规格说明 SRS：开发前，中，后相关人员沟通的基础
定义良好的需求要求：
- 可验证性： 可以被验证
- 必要性：必须被系统满足，来实现利益相关者目标或者解决要求
- 可量化性：可以被可量化的指标评估，并且有约束限制

需求样例：[条件][主体][动作][客体][约束][值]（值和约束是一起的，和原本需求相比，只需加上条件，约束。值）



constraints 可以是专用，通用或者独立的，对延迟，范围，形式进行要求

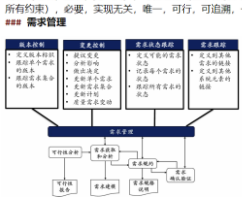
需求缺陷的代价

敏维斯假设：软件生命周期中，需求缺陷发现越晚，修复它的代价越高
- 缺陷发现阶段&&相对修复代价：需求分析：0.1~0.2 设计：0.5 编码：1 单测：2，系统测试：5 软件维护：20

需求质量特性

明确（准确，正确，无二义），可验证性（基于客观手法，检验是否正确实现），完整（单个需求包括必要信息，需求文档覆盖所有约束），必要，实现无关，唯一，可行，可追溯，一致

需求管理



软件运维

运维技术不同阶段：人力驱动->自动驱动+人力决策->AI取代人力，自主快速准确决策
软件运维在全生命周期开销占比：40%~90%

必要性

1. 运维（Ops）传统上需要的技能集与开发（Dev）所需的技能集大不相同：组装现有软件组件并部署它们，使其协同工作以提供服务;运行服务并根据实际发生的事件和更新进行响应;负责可用性、延迟、性能、效率、变更管理、监控、应急响应和容量规划
2. 大部分中断由某种变化产生，比如新feature 新配置啥的

运维开销

程序大小，事件数，警告级别一定递增；每年都会寄掉20% server;云端，移动端，微服务的数字化，还有IoT啥的

SRE(Site Reliability Engineering)

让一个软件工程师做一个运维程序，Ops Automation

tenets of SRE

1. 确保对工程的持久关注
运维时间不超过50%;每个8-12小时的值班班次中，事件发生次数不多于两次，不少于一次；所有重大事件都应编写事后分析报告。
2. 追求最大的变更速度，同时不违反服务的服务水平目标（SLO）
业务/产品确定系统的非100%可用性目标，错误预算 = 1 - 可用性目标。中断不再是一件“坏事”—它是创新过程中的预期部分。
3. 监控
紧急性：警报（立即行动）> 工单（行动）> 日志（仅供参考）。
4. 应急响应
可靠性是故障平均时间（MTTF）和修复平均时间（MTTR）的函数。人为因素会增加延迟 - 尽可能自动化。轮值“手册”（MTTR提高3倍）和“灾难角色扮演”。

5. 变更管理
70%的中断是由现有系统的变更引起的。从循环中移除人为操作（像是渐进式发布，快速准确检测问题，问题出现时安全回滚）
6. 需求预测和容量规划
确保有足够的容量和冗余度来满足预测的未来需求，并保证所需的可用性。
三个必要步骤：（一个准确的有机需求预测，其预测期限需超过获取容量所需的前置时间；将无机需求来源准确纳入需求预测；定期进行系统负载测试，将原始容量（服务器、硬盘等）与服务容量相关联）
7. 配置
结合变更管理和容量规划
8. 效率和性能
高效利用资源可以节省（大量）成本；性能（响应速度）应密切监控并保持

拥抱风险原则

1. 平衡不可用的风险与快速创新和高效服务运营的目标（优化用户的整体幸福感—功能、服务、性能、**可靠性**。用户通常无法察觉服务的**高可靠性与极端可靠性**之间的差异）
2. 通过管理风险来管理服务可靠性（减少系统故障可能，确保可靠性的成本随着可靠性的增加而非线性增长如加入冗余机器/计算资源的成本，考虑机会成本（用于新产品功能））
3. 衡量服务风险：1. 低可靠性的成本：用户不满、伤害或失去信任；直接或间接的收入损失；品牌或声誉影响；不良的媒体报道 2. 基于时间的可用性 = 运行时间 /（运行时间 + 停机时间）3. 合计可用性（谷歌首选） = 成功请求 / 总请求，例如，99.99%的可用性意味着每250万请求中有250个错误
4. MTBF(mean time between failures)=uptime / num_of_failures;MTTR(mean time to repair)=downtime/num_of_failures;可用性=MTBF/(MTBF+MTTR)

5. 一些核心观点：1. 管理可靠性主要是风险管理 2. 将服务的风险配置与企业愿意承担的风险相匹配 没必要刻意追求100%
3. **错误预算**对齐激励，强调 SRE 和项目发展的共同所有权，便于**确定发布速率**和绩解和利益相关者关于中断的讨论，便于团队就生产风险达成相同的结论。

服务质量目标

服务级别术语

1. 服务级别指标（SLI）：对提供的服务级别的某些方面进行仔细定义的定量测量，例如，请求延迟、错误率、系统吞吐量、可用性
通常在一定时间窗口内聚合：率、平均值或百分位
2. 服务级别目标（SLO）：由SLI测量的服务级别的目标值或值范围。应成为优先处理工作的主要驱动力。例如，每次请求的平均延迟应低于100毫秒，例如，Google Compute Engine的99.95%可用性
3. 服务级别协议（SLA）：与用户之间的显式或隐式合约，包括达到（或未达到）其中包含的SLOs的后果。

实践中指标

- 是一系列有代表性的指标，belike:
  - 面向用户的服务系统中的可用性，延迟，吞吐量；存储系统中的延迟，可用性，持久性；大的数据系统：吞吐量；端到端延迟
  - 如何收集这些指标：serverside：Prometheus,Zabbix,Borgmon;或者client-side 插装；数据聚合的方法是平均值或者用百分位数

总结

1. 定义目标
2. 选择目标：不要根据当前性能选择目标；保持简单，不用过于复杂的聚合；尽可能少设置 SLO ；不追求完美
3. SLOs设置期望：保持安全余量，不要过度实现

尽量消除琐事

琐事：**是指与运行生产服务相关的工作类型，这种工作往往是手动的、重复的、可自动化的、战术性的，缺乏持久价值，并且随着服务的增长而线性扩展**
SRE 组织推荐把 toil 这种工程性工作的占比降低到50%以下；每个SRE的时间至少有50%应该是开发性工作，像是减少未来琐事或者加feature，google的这个占比是33%，这个开发性工作要求新颖，本质，而且需要人类的判断力
SRE activities 一般有如下几类：软件工程（写代码，设计，写文档），系统工程（配置），琐事，overhead(与直接运营无关的行政工作)

分布式系统的监控

白盒监控：看内部指标比如说log,JVM log HTTP log
黑盒监控：测试面向外部的特征
- 为什么需要监控：
- 1. 分析长期趋势 2. 按时间或者和对照组对比 3. 便于警报：有东西坏了，有人需要马上处理（警报最好有很好的信号而且不会吵；滴滴别人干活的开销有点大，而且大经常的警报会造成“紧急警报疲劳”）从而延长中断时间 4. 建造dashboard：4 golden signals（延迟，流量，错误，饱和度（可以用某个小窗口内第99百分位响应时间表示））；不需要人监控 dashboard
5. 便于回溯分析
- 白盒 vs 黑盒：黑盒：面向症状，表征现存问题；白盒：取决于透视系统内在结构的能力：可以探测即将来临的问题，被重试掩盖住的错误设施
- 监视百分位数指标：（如果你运行的网络服务在每秒1000个请求的情况下平均延迟为100毫秒，那么最后1%的请求可能需要5秒钟）
  - 选择合适的测量分辨率，细粒度监控不总是必须的，高分辨率不一定需要低延时
  - 尽可能简单化，为长期做监控

紧急警报的哲学

- 每次寻呼器响起时，我（Pager）应该能够以紧迫感做出反应。一天之内我只能几次以紧迫感反应，之后我会感到疲劳
- 每个寻呼都应该是可操作的
- 每次响应寻呼都应该需要人类智能。如果一个寻呼只是值得机械式的响应，那么它不应该是一个寻呼。
- 寻呼应该关于一个新问题或一个之前未见过的事件。
- 应该有人找到并消除问题的根本原因；如果无法做到这一点，那么警报响应应该完全自动化。

服务质量具象化

- 最基本要求：功能性
- 高阶要求：自我实现，主动把握服务的方向。这些需要一系列活动，比如监控系统，容量，紧急相应，判断中断的根源
- 监控的好处：1. 更理性的看到改变对服务的影响 2. 对相应事件更加理性 3. 评估服务和商业目标的对齐程度
- 监控大系统的挑战：1. components 多 2. 需要减轻人员运维负担->需要监控及时警报高等级事件，但是不必过于精细监控个别组件

轮值：概念

在工作和非工作时间都可响应，以保持其服务的可靠性和可用性
值班工程师工作流程：一旦接到并确认了寻呼，值班工程师预期将对问题进行分类并努力解决问题，可能涉及其他团队成员并根据需要进行升级->比如说9.99% 的可用性 -> 每季度错误预算为13分钟 -> 在5分钟内开始操作 ->对于不太敏感的事件，预留30分钟时间->非寻呼事件在工作时间内以较低优先级处理
平衡值班：1. 数量上的平衡：我们坚信“SRE”中的“E（工程）”。SRE至少有50%的时间用于工程工作，其余时间中最多25%可用于值班 2. 质量上的平衡：每次值班，工程师应有足够时间处理任何紧急事件和后续活动，如编写事后总结 3. 每次紧急事件6小时，这是一系列与同一根本原因相关的事件和警报，将作为同一事后总结的一部分讨论

轮值：feeling safe

1. 减少轮值的压力

2. 理性，专注，深思熟虑的思考便于中断处理
   1. 避免未经反思和考虑的行动；过于依赖直觉可能会导致工程师浪费时间追求一条从一开始就错误的推理线索
3. 理想的方法论是在有足够数据做出合理决定时以期望的速度采取措施，同时严格审视你的假设，从而达到完美的平衡
4. 依赖值班资源：明确的升级路径，明确定义的事故管理程序和相应的工具，无责备的事后总结文化。但在重大事故后，SREs必须编写事后总结报告，并详细说明发生的事件的完整时间线
5. 避免过量运维负担：
   1. 可能由配置不合适的监控引起 2. 控制值班工程师因单一紧急事件接收的警报数量；抑制和压缩 3. 与应用程序开发者合作设定共同目标以改进系统 4. 在系统达到SRE标准之前，将寻呼器交还给开发者

有效排查问题

- 需要两方面的知识：通用的排查问题的能力，对系统的深刻了解
- 通用策略：假设演绎法
  1. 给定一组关于系统的观察和理解系统行为的理论基础，我们迭代地假设故障的潜在原因并尝试测试这些假设。
  2. 将理论与证据进行比较；处理系统并进行观察
  3. 直到识别出原因
- 可能会遇到的问题
  1. 关注不相关的症状或误解系统指标的含义 2. 误解如何改变系统、其输入或其环境，以便安全有效地测试假设 3. 出关于问题所在的极不可能的理论，或者抓住过去问题的原因 4. 在所有条件相同的情况下，更倾向于简单的解释。（奥卡姆剃刀原则） 5. 追查似是而非的相关性 6. 相关性！=因果性
- 实践：
  - 问题报告：告诉你预期的行为、实际的行为，以及如果可能的话，如何复现该行为。目应为将来参考保留调查和修复活动的日志

  - 分级处理：先评估问题严重性，top priority应当停止损失；让系统在当前情况下尽可能正常运行(比如将流量从故障集群转移到仍在工作的其他集群，整体丢弃流量以防止级联失败，禁用子系统来减轻负载)
  - 查看：查看每个系统中组件，比如系统和应用的 log,调用的 trace change logs，workflow logs 这些
- 诊断：
  - 简化和减少：1. 观察组件之间的连接或者观察数据流动，来确定某个组件是否正常工作 2. 从调用栈的一端系统地开始，逐步向另一端工作，依次检查每个组件 3. 问“what”,“where”,“why”,(找出出现故障的系统正在做什么，询问它为什么这样做，看它的资源在哪里被使用，或者它的输出去向何处)
  - 看哪个最后改变它，比如配置或者负载类型改变，看伴随变化的系统性能或者行为也有帮助

总结

1. 从基础开始为每个组件构建可观察性—包括日盒指标和结构化日志 2. 记录工作流程和变更 3. 设计具有易于理解和可观察的组件间接口的系统 4. 确保信息在系统中以一致的方式可用，可以加快诊断和恢复的时间 5. 采用系统化的故障排除方法—而不是依靠运气或经验—可以帮助限定服务恢复时间，从而为用户带来更好的体验

紧急事件响应

1. 三种可能：由测试引发的紧急事件，由改变引发的紧急事件，由过程引发的紧急事件
2. 原则：一切问题都有解答（系统一定会以未曾设想的方式出问题；有需要的话，尽快求助队友；解决之后要清理总结）；从过去学习，不要重复犯错
3. well-designed management 的元素
   1. 递归分责（事件指挥官，运维工作组（背锅的人），沟通（事件响应组的公众面孔），规划（支撑运维工作））2. 明确指挥部（作战室或者 IRC）3.实时事件状态文档 4. 事件指挥官的明确、实时交接
4. 事后分析
   1. 是对事件、其影响、采取的缓解或解决措施、根本原因以及为防止事件重复发生而采取的后续行动的书面记录
   2. 目标：事件被记录下来，所有有助于根本原因的因素都被充分理解，有效的预防措施得以实施，以减少事件重复发生的可能性和/或影响