# 服务架构演进史

- ## 原始分布式时代
  - **背景：**早期计算机资源相对较小，为了突破硬件能力的限制，寻找多台计算机共同协作来支撑同一套软件系统的可行方案。
  - **成果：**网络运算架构NCA（远程服务调用雏形），
    AFS文件系统（分布式文件系统最早实现），
    Kerberos协议（服务认证和访问控制基础性协议）
  - **后续：**制定了分布式技术体系DCE。包含基于NCA的远程服务调用规范DCE/RPC，基于AFS的分布式文件系统规范DCE/DFS等。
  - **缺陷：**强求分布式操作的透明性，即开发人员不必关注
    访问的方法和资源是否远程或本地，使得技术难度极高。
- ## 单体系统时代
  - **优点：**因为更容易开发、部署、测试而获得更好便捷性
  - **缺点：**由于隔离能力缺失，难以阻断错误传播、不便于动态更新、难以技术异构。关键在于当规模变大，交付一个可靠的单体系统变得极具挑战性和困难。
- ## SOA时代
  - **拆分单体的历史架构：**烟囱式架构（信息孤岛）、微内核架构（公共服务、数据资源等集中为核心，其他服务作为插件）、事件驱动架构（通过事件管道进行服务间通信）
  - **特征：**有清晰的软件设计指导原则，采用SOAP作为远程调用协议、使用企业服务总线ESB进行交互等。
  - **缺陷：**过于严格的规范定义带来过度复杂性，过于
    精密的流程和理论需要理解深刻才能驾驭
  - **贡献：**提出服务间的松散耦合、注册发现、治理、隔离、编排等微服务的概念。解决分布式环境中提出的主要技术问题。
- ## 微服务时代
  - **概念：**微服务是一种通过多个小型服务组合来构建
    单个应用的架构风格，服务围绕业务能力而非技术
    标准来构建，服务可以采用不同语言和技术实现。
    服务采用轻量级通信机制和自动化部署。
  - **业务与技术特征**
    - **围绕业务能力构建**
    - **分散治理**
    - **通过服务来实现独立自治的组件**
    - **产品化思维**
    - **数据去中心化**
    - **强终端弱管道**
    - **容错性设计**
    - **演进式设计**
    - **基础设施自动化**
- ## 后服务时代（云原生）
  - **意义：**用虚拟化技术和容器技术来解决分布式架构问题。未来，K8S成为服务端标准运行环境，服务网格成为微服务之间通信交互的主流模式，将通信协议、认证授权等技术问题隔离于程序代码，微服务只关注业务本身的逻辑。
- ## 无服务时代
  - **愿景**
    - 不需要考虑技术组件，后端技术组件现成
    - 不需要考虑如何部署，完全托管至云端
    - 不需要考虑计算能力，由数据中心支撑
    - 不需要考虑运维操作，由云服务商负责
  - **内容**
    - **后端设施：**数据库、消息队列、日志、存储等。运行在云中。
    - **函数：**指业务逻辑代码，无服务中函数即服务
  - **特点：**无服务架构适合于短链接、无状态、适合事件驱动的交互形式。对于具有复杂业务逻辑、依赖服务端状态、响应速度要求较高、需要长链接等特征应用不太合适（无服务按使用量计费，因此函数不会一直以活动状态常驻服务器，不便依赖服务端状态）。

# 访问远程服务

- **远程服务调用RPC**
  - **进程间通信IPC**
    - **通信方式：**管道或具名管道，负责在进程间传递字符流或字节流。
    - **通信载体**
      - 信号
      - 信号量
      - 消息队列
      - 共享内存
      - 本地套接字接口
  - **通信成本**
    - **通过网络进行分布式运算的八宗罪**
      - “网络是可靠的”
      - “延迟是不存在的”
      - “带宽是无限的”
      - “网络是安全的”
      - “拓扑结构是一成不变的”
      - “总会有一个管理员”
      - “不必考虑传输成本”
      - “网络都是同质化的”
    - **RPC定义：**指位于互不相同的内存地址空间中的两个程序，在**语言层面**上，以同步的方式来使用带宽有限的信道来传输程序控制信息。
    - **RPC与IPC区别：**RPC是一种高层次的或者说语言层次的特征，而IPC是低层次的或者说系统层次的特征。
  - **RPC协议三个基本问题**
    - **如何表示数据：**包括传递给方法的参数和方法返回值，其实就是**序列化协议**。如Protocol Buffer、JSON、XML等序列化。
    - **如何传递数据：**指如何通过网络在两个服务的EndPoint之间交换数据，称为Wire Protocol。如SOAP、HTTP协议等。
    - **如何表示方法：**指如何用跨语言的标准表示特定方法。如使用UUID、Web服务描述语言WSDL、接口描述语言IDL等。
  - **统一的RPC**
    - **跨系统、跨语言的CORBA：**设计繁琐，大量无效冗余代码。指定的规范脱离实际，导致后续各个语言厂商有自己解读，最终各不兼容。
    - **Web Service：**基于XML，缺点是性能较差，且包含一大堆其他协议，学习困难。
  - **分裂的RPC**
    - **朝着面向对象发展：**不满足于RPC将面向过程编码方式带到分布式，希望能进行跨进程的面向对象编程。如RMI、.NET Remoting。
    - **朝着性能发展：**性能取决于序列化效率和信息密度。序列化输出结果的容量越小，速度越快，效率越高。信息密度取决于有效负载占总传输数据的比例大小，使用的传输协议层次越高，密度越低。如gRPC和Thrift都有专用序列化器，gRPC基于HTTP/2，支持多路复用和Header压缩，Thrift基于传输层TCP协议。
    - **朝着简化发展：**协议简单轻便，接口和格式更为通用，尤其适合Web浏览器这种不会有额外协议支持的应用场合。代表有JSON-RPC。
    - **向更高层次与插件化发展：**不再追求独立解决RPC三个问题，将一部分功能设计为扩展点，由用户选择。聚焦于提供核心的更高层次功能，如负载均衡、服务注册等。代表为阿里的Dubbo，支持多种传输协议选择，支持多种序列化器选择。
- **REST设计风格**
  - **RPC和REST区别**
    - **思想上：**面向过程和面向资源的编程思想不同
    - **概念上：**REST并不是一种远程服务调用协议，只是一种风格，不带有一定规范性和强制性。
    - **范围上：**RPC的发展方向包括分布式对象、提升调用效率、简化调用复杂性。其他，分布式对象的应用与REST毫无关系；而REST应用最多的浏览器可选传输协议和序列化器不多，提升调用效率困难；基于简化调用复杂性，几乎只有JSON-RPC可与REST竞争。
  - **REST相关概念**
    - **资源：**信息数据本身
    - **表征：**资源的不同表现形式
    - **状态：**基于特定场景的上下文信息
    - **转移：**服务端将表征经过某种操作实现状态转移。即”表征状态转移“。
    - **统一接口：**HTTP中GET、PUT、POST等，表征状态转移的具体方式
  - **REST六大原则**
    - **客户端与服务端分离：**将用户界面关注逻辑与存储关注的逻辑分离，提高用户界面跨平台的可移植性。

- **无状态：**服务端不用负责维护状态，客户端来处理或维护必要的上下文信息、会话信息。但绝大多数系统都无法满足，因为大型系统上下文信息庞大。
- **可缓存：**无状态提升了系统可见性、可靠性和可伸缩性，但降低网络性，使得有状态的设计在无状态时需要多次请求。REST希望客户端与中间的通信传递者如代理可以将部分服务端的应答缓存起来。
- **分层系统：**客户端无需知道是否直接连接到最终服务器或中间服务器，中间服务器可以使用负载均衡和共享缓存提高系统可扩展性，便于缓存、伸缩和安全策略部署。代表应用是内容分发网络CDN。
- **统一接口：**REST希望面向资源编程，侧重于系统该有哪些资源，使用HTTP的统一接口来实现资源相关的操作
- **按需代码：**可选原则，指任何按照客户端请求将可执行代码从服务端发送至客户端的技术，如Java Applet。

- **REST好处**
  - **降低服务接口学习成本**
  - **资源天然具有集合与层次结构**
  - **REST绑定与HTTP协议，既是好处又是缺点**
- **REST成熟度RMM**
  - **第0级：**完全与REST不相关
  - **第1级：**引入资源概念，通过资源ID作为主要线索与服务交互
  - **第2级：**引入统一接口，使用HTTP协议的Status Code、Header等信息
  - **第3级：**超文本驱动，请求应该自己描述清楚后续可能发生的状态转移
- **不足与争议**
  - **面向资源编程对CURD之外操作设计麻烦**，面向过程思想符合计算机世界主流交互方式，面向对象思想符合现实世界主流交互方式，面向资源思想符合网络世界主流交互方式。
  - **REST与HTTP完全绑定，不适合应用于要求高性能传输场景**
  - **REST协议不具有直接支持分布式服务对多个数据同时提交的统一协调能力**，但实现最终一致性是可以的。
  - **REST没有传输可靠性支持**，若未收到响应，无法确定服务端操作进行到哪一步，只能重发一次请求，这要求服务具有幂等性。
  - **REST缺乏对资源进行部分和批量处理的能力**
  - GraphQL可以解决上述问题，但是脱离了HTTP，推广成为问题

# 事务处理

- **ACID概念**
  - **一致性Consistency：**保证系统中所有的数据都是符合期望的，且相互关联的数据之间不会产生矛盾。
  - **原子性Atomic：**在同一项业务处理过程中，事务保证了对多个数据的修改，要么同时成功，要么同时失败。
  - **隔离性Isolation：**在不同的业务处理过程中，事务保证了各业务在读写的数据相互独立，不会影响彼此。
  - **持久性Durability：**事务保证所有成功被提交的数据修改都能正确地被持久化，不丢失数据。
- **本地事务**
  - **定义：**指仅操作单一事务资源的、不需要全局事务管理器进行协调的事务。仅适用于单服务使用单个数据源的场景，直接依赖于数据源本身提供的事务能力来工作。
  - **实现原子性和持久性**
    - **难处：**实现原子性和持久性的最大困难在于“写入磁盘”这个操作不是原子的
    - **可能出现情形**
      - **未提交事务，写入后崩溃：**程序还没完整修改完数据，但数据库将其中几个数据的变动写入了磁盘，若此时出现了崩溃，一旦重启后，数据库必须有办法知道崩溃前做了一次不完整的磁盘写入，需要将已经修改过的数据从磁盘恢复成未修改状态。
      - **已提交事务，写入前崩溃：**程序已完成修改完数据，但数据库并未全部将数据变动写入磁盘，若此时出现崩溃，一旦重启后，数据库必须有办法知道崩溃前发生过一次完整的操作，需要将没写入磁盘的部分数据重新写入。
    - **解决方法**
      - **提交日志Commit Logging：**事务不直接修改磁盘，而是将数据修改操作记录在日志中，日志记录全部写入后，添加Commit Record标志，数据库根据此信息才开始真正修改磁盘数据，修改完成后在日志中添加End Record标志。
      - **Shadow Paging：**对数据的变动写入数据的副本，当事务成功提交时，数据的修改都成功持久化后，最后一步是修改数据的引用指针。但是涉及隔离性和并发锁时，事务并发能力较弱，因此不适用于高性能数据库。
    - **Commit Logging改进**
      - **缺陷：**所有对数据的修改必须发生在事务提交之后，此时即使磁盘I/O足够空闲也不被允许在事务提交前写入磁盘数据。因此，引入提前写入日志Write-Ahead Logging。
      - **Write-Ahead Logging相关概念**
        - **FORCE：**是否要求事务提交后同步写入数据
        - **STEAL：**是否允许事务提交前写入数据
      - **Undo Log：**提前写入增加了Undo Log日志类型，在变动数据写入磁盘前，必须记录在Undo Log中哪些数据发生了什么变动，以便在事务回滚或者崩溃恢复时根据记录进行数据变动的擦除。此时，Commit Logging的日志称为Redo Log。
    - **Write-Ahead Logging**
      **崩溃恢复**
      - **分析阶段：**从最后一次检查点开始扫描日志（检查点前的记录都已正常持久化），找出没有End Record的记录，组成待恢复的事务集合，包括事务表和脏页表。
      - **重做阶段Redo：**从待恢复的事务集合中，找出包含Commit Record的日志，将这些日志的修改写入磁盘，并在日志后添加End Record标志，移出待恢复事务集合。
      - **回滚阶段Undo：**经过重做阶段后的事务集合中只剩下需要回滚的事务，根据Undo Log将提交到磁盘的数据变动进行恢复
    - **FORCE和STEAL组合**
      - **FORCE+NO-STEAL：**最慢，但是不需要日志
      - **FORCE+STEAL：**需要回滚日志
      - **NO-FORCE+NO-STEAL：**需要重做日志，即Commit Logging
      - **NO-FORCE+STEAL：**最快，需要重做日志和回滚日志，即提前写入
  - **实现隔离性**
    - **三种锁**
      - **写锁/排他锁X-Lock：**数据有写锁时，只有持有写锁的事务才能对数据进行写入操作。数据持有写锁时，其他事务不能写入，也不能添加写锁。
      - **读锁S-Lock：**多个事务可以对同一个数据添加读锁，数据被添加上读锁之后无法被加上写锁。若一个数据只有一个事务添加了读锁，则允许将其升级为写锁。但是读取数据不一定要添加读锁。
      - **范围锁：**对某个范围直接加排他锁，在这个范围内的数据不能被写入。
    - **隔离级别**
      - **可串行化Serializable：**对事务所有读、写的数据都加上锁，并且使用范围锁。
      - **可重复读Repeatable Read：**对事务所有读、写的数据都加上锁，但是不使用范围锁。因此相比较可串行化，会出现**幻读问题**，指的是在事务执行过程中，两个完全相同的范围查询会得到不同的结果。基于ARIES理论，实际情况有出入，如MYSQL还有MVCC等支持，只读事务不会出现幻读，读写事务才会出现幻读。
      - **读已提交Read Commit：**数据加的写锁会持续到事务结束，但读锁在查询操作之后就直接释放。因此相比较可重复读，会出现**不可重复读问题**，指的是在事务执行过程中对同一行数据的两次查询得到不同的结果。
      - **读未提交Read UnCommitted：**只对事务涉及的数据加写锁并持续到事务结束，但不添加读锁（即不通过读锁来读取数据，直接读

取）。因此相比较读已提交，会出现**脏读问题**，指的是事务读取到其他事务还未提交的数据。

- **MVCC**
  - **定义：**多版本并发控制，是一种无锁读取优化方案，指的是读取时不需要加锁。针对读+写场景，写+写场景还是要上锁，只是区分乐观锁和悲观锁。
  - **基本思路：**对数据库的修改不直接覆盖之前的数据，而是产生新版本和老版本。
  - **实现原理**
    - **隐藏字段：**DB_TRX_ID（最后一次修改该记录的事务ID）、DB_ROLL_PTR（配合undo log指向上一个旧版本）、DB_ROW_ID（隐藏主键）
    - **Read View：**事务进行快照读操作的时候生产的读视图，在该事务执行快照读的那一刻，会生成一个数据系统当前的快照，记录并维护系统当前活跃事务的id。包含三个全局属性，**trx_list**（维护正活跃事务ID）、**up_limit_id**（trx_list中最小ID）、**low_limit_id**（Read View生成时系统尚未分配的下一个事务ID）。
    - **比较规则：判断某一行是否被判断为可读取的数据**。
      当DB_TRX_ID小于up_limit_id时，则为可读取数据；
      当DB_TRX_ID大于low_limie_id时，则为不可读取数据；
      当DB_TRX_ID处于它们之间时，若DB_TRX_ID在trx_list中，
      则不可读取，在trx_list之外则可读取
  - **RR和RC**
    - **RR可重复读：同一个事务中的第一个快照读才会创建Read View**，之后的快照读获取的都是同一个Read View，保证可重复读的特性，避免不可重复读。
    - **RC读已提交：每个快照读都会生成并获取最新的Read View**，不可避免会出现不可重复度现象。

## 全局事务

- **定义：**与本地事务相对应，指适用于单个服务多个数据源的事务解决方案。是一种在分布式环境中仍然追求强一致性的事务处理方案。
- **X/Open XA**
  **事务处理架构**
  - **组件**
    - 全局的事务管理器Transaction Manager，用于协调全局事务
    - 局部的资源管理器Resource Manager，用于驱动本地事务
  - **两段式提交2PC**
    - **准备阶段：**协调者询问参与者是否准备好，参与者进行回复Prepared或Non-Prepared。数据库进行准备操作时，**在重做日志中进行到事务提交前的最后一步Commit Record不写入**，此时事务未完整提交，仍然持有锁，维持隔离性。
    - **提交阶段：**协调者若收到所有的Prepared消息，则将自己的本地事务状态标为Commit，之后向所有参与者发送Commit消息，让参与者进行事务提交操作；若收到Non-Prepared消息或有参与者超时未回复，则协调者将自身事务状态标为Abort，并且向参与者发送Abort消息，让参与者进行回滚。
  - **前提条件**
    - **必须假设网络在提交阶段的短时间内是可靠的，不会丢失消息**。在投票阶段失败了可以回滚补救，但是若提交阶段失败了则无法补救，只能等崩溃的节点重新恢复。因此该阶段尽可能耗时较短。
    - **必须假设因为网络分区、机器崩溃或其他原因而导致失联的节点最终都能恢复**
  - **缺点**
    - **单点问题：**协调者重要性太大，参与者等待协调者指令无法做超时处理，若协调者宕机则所有参与者都会受影响。
    - **性能问题：**所有参与者都被绑定为一个统一调度的整体，在2PC需要经过两次远程服务调用、三次数据持久化（准备阶段写重做日志、协调者做状态持久化、提交阶段写入事务提交），整个过程必须等待参与者中最慢的那一个。
    - **一致性风险：**若协调者在持久化本地事务后，网络突然断开，无法向所有参与者发出Commit消息，则会出现协调者数据已提交，但参与者的数据未提交，且无法回滚，导致数据不一致。
  - **三段式提交3PC**
    - **背景：**解决2PC中的单点问题和性能问题
    - **过程：**3PC将准备阶段拆分为CanCommit和PreCommit阶段，将提交阶段称为DoCommit阶段。其中，CanCommit是询问阶段，让参与者评估事务是否可以顺利完成，避免2PC时有一个参与者准备失败，其他参与者所作工作都白费的情况发生。若PreCommit之后协调者宕机，则默认操作是参与者提交事务，避免单点问题。

## 共享事务

- **定义：**与全局事务刚好相反，指多个服务共用一个数据源
- **实现方式**
  - 直接让各个服务共享数据库链接，但要求数据源的使用者在同一个进程内。增加一个中间数据源服务器的角色，其他服务都通过它来与数据库交互
  - 使用消息队列服务器来代替中间数据源服务器，即通过消息的消费者来统一完成本地事务
- **缺点：**数据库一般是压力最大且最不易伸缩扩展的资源，现实中几乎没有反过来代理一个数据库为多个服务提供事务协调。

## 分布式事务

- **定义：**指多个服务同时访问多个数据源的事务处理机制，即在分布式服务环境下的事务处理机制。
- **CAP**
  - **一致性Consistency：**代表数据在任何时刻、任何分布式节点所看到的都是符合预期的。
  - **可用性Availability**：代表系统不间断地提供服务的能力。包括可靠性和可维护性两个指标，**可靠性使用平均无故障时间MTBF度量，可维护性用平均可修复时间MTTR来度量**。可用性A=MTBF/(MTBF+MTTR)
  - **分区容忍性Partition Tolerance：**代表分布式环境中部分节点因为网络原因而彼此失联，与其他节点形成网络分区时，系统仍能正确地提

供服务的能力。

- **放弃CAP的影响**
  - **放弃分区容忍性：**即假设节点之间的通信永远可靠，但是这是不可能存在的
  - **放弃可用性：**意味着一旦发生网络分区，则节点之间信息的同步时间可以无限延长
  - **放弃一致性：**意味着一旦发生网络分区，节点之间提供的数据可能不一致。
    AP是当前分布式系统的主流选择，毕竟选择分布式就是为了追求高可用。
    选择CP是在一些宁愿中断服务也不允许数据不一致的情况。
    **选用AP不意味着完全放弃一致性，而是追求最终一致性。此时ACID成为刚性事务，其余分布式事务称为柔性事务。**
- **可靠事件队列**
  - **定义：**也叫做最大努力一次提交，将最有可能出错的业务以本地事务的方式完成后，采用不断重试的方式来促使同一个分布式事务中的其他关联业务全部完成。
  - **过程**
    - 1、最终用户发出一个请求，涉及多个服务，生成分布式事务
    - 2、根据服务的出错概率进行动态排序，选择最容易出错的最先进行
    - 3、将最容易出错的服务的业务数据操作和消息写入作为本地事务写入数据库
    - 4、在系统中建立消息服务，定时轮询消息表，将进行中的消息发给其他服务
    - 4-1、若其他关联服务完成了各自的事务，则向最初的服务返回执行结果，初始服务将消息状态改为已完成，至此分布式事务结束。
    - 4-2、若有关联服务未收到消息，则消息服务器将在轮询时向未响应的服务重发发生消息，因此消息处理必须是幂等性的。
    - 4-3、若关联服务无法完成工作，此时消息服务器依旧会持续发生消息，直至操作成功或人工介入。因此可靠事件队列一旦第一个服务事务成功了，就没有失败回滚的概念，只允许后续操作成功。
    - 4-4、若关联服务完成了事务，但响应消息丢失，则消息服务器将持续发送消息，直到收到响应消息。
  - **缺点：**整个过程没有任何隔离性，一些业务若缺乏隔离性将导致出现异常情况。例如，存在购买业务，两个用户各自购买的数量未超过商品数量，但是合起来的数量却超过了，而未做隔离性设置，将导致这两个业务都成功，但实际上关联业务出现异常。
- **TCC事务**
  - **定义：**Try-Confirm-Cancel事务，相比较于可靠事件队列，增加了隔离性。是一种业务侵入性较强的事务方案。
  - **阶段**
    - **Try：**尝试执行阶段，完成所有业务可执行性的检查，并且预留好全部需要用到的业务资源，保障隔离性，必要时冻结资源。
    - **Confirm：**确认执行阶段，直接使用Try预留的资源来完成业务处理，由于此阶段需要循环直至业务完成，因此要求操作需要幂等性。
    - **Cancel：**取消执行阶段，释放Try预留的资源，同样要求操作具有幂等性。
  - **优势和不足：**TCC类似于2PC的准备和提交阶段，TCC在业务执行时只操作预留资源，几乎不会涉及锁和资源的竞争，有一定的性能潜力。但是TCC也带来更高的开发成本和业务入侵性，一般由分布式事务中间件来完成。
  - **限制：**它的业务入侵性很强，但是存在有些第三方业务是不被允许锁定资源的情况，因此第一阶段Try就无法进行。
- **Saga事务**
  - **定义：**提升长时间事务运作效率的方式，将大事务拆分成可以交错执行的一系列子事务。相比较TCC，Saga不需要冻结资源，补偿操作一般比冻结操作简单。
  - **操作**
    - 1、将大事务拆分成多个小事务，即将事务T拆分为T1~TN
    - 2、为每一个子事务设计补偿动作，要求子事务和补偿操作都具备幂等性，同时补偿操作必须能提交成功
  - **恢复策略**
    - **正向恢复：**一直重试直到所有子事务完成
    - **反向恢复：**从失败的子事务开始，逆序执行补偿操作
  - **前提：**Saga必须保证所有子事务都能提交或补偿，由于Saga本身也有可能崩溃，因此被设计为与数据库类似的日志机制，保证系统恢复后可以追踪到子事务的执行情况。
- **AT事务模式**
  - **做法：**在业务数据提交时拦截所有SQL，将SQL对数据修改前、修改后的结果保存快照，生成行锁，通过本地事务一起提交到操作的数据源中，相当于自动记录了重做和回滚日志。若分布式事务成功提交，则自动清理每个数据源中对应的日志数据，若分布式事务需要回滚，则根据日志数据自动生成补偿的逆向SQL
  - **优势：**基本逆向SQL的补偿方式，分布式事务的每一个数据源都可以独立提交，然后立即释放锁和资源。相比较2PC，提升了系统的吞吐量。
  - **缺点：**大幅度牺牲了隔离性，甚至影响了原子性。在缺乏隔离性的前提下，以补偿代替回滚并不是总是成功的，可能本地事务提交后，分布式事务完成之前，该数据被补偿前被其他操作修改了，出现了脏写，此时分布式事务若需要回滚，则只能人工介入。因此GTS增加了全局锁来实现写隔离，避免脏写。

# 透明多级分流系统

- **前言**
  - **不同部件价值**
    - 一些位于客户端或网络边缘的部件，能够迅速响应用户的请求，避免给后方I/O与CPU带来压力，如本地缓存、内容分发网络CDN、反向代理等。
    - 一些部件的处理能力能够线性拓展、易于伸缩，可以使用较小的代价堆叠机器来获得与用户数量相匹配的并发能力，应尽量作为业务逻辑的主要载体，如集群中能够自动扩缩的服务节点。
    - 一些部件稳定服务对系统运行有全局性的影响，要时刻保持容错备份，维护高可用性，如服务注册中心、配置中心。
    - 一些部件是天生的单点部件，只能依靠升级机器本身的网络、存储和运算性能来提升处理能力，如位于系统入口的路由、网关或负载均衡器（它们可以做集群，但一次请求中无可避免有一个是单点的部件）、位于请求调用链末端的传统关系数据库。
  - **设计原则**
    - **尽可能减少单点部件。若单点无可避免，则尽量减少到达单点部件的流量**。适当引导请求分流到最合适的组件中，避免绝大多数流量汇聚到单点部件如数据库中，同时依然能够在绝大多数时候保证处理结果的准确性，使单点系统在出现故障时可以自动且迅速实施补救措施，即**系统多级分流的意义所在**。
    - **奥卡姆剃刀原则，在能够满足需求的前提下，最简单的系统就是最好的系统。**
- **客户端缓存**
  - **状态缓存：**不经过服务器，客户端直接根据缓存信息对目标网站的状态判断，对应响应码301Moved Permanently永久重定向。如资源已经永久移到新地址，缓存信息在第二次访问时会根据缓存自动向新地址发出请求。
  - **强制缓存**
    - **定义：**假设在某个时间点来临前，资源的内容和状态一定不会改变，因此客户端可以无须经过任何请求，在时间点到来前一直持有并使用该资源的本地副本。
    - **生效场景：**在浏览器的地址输入、页面链接跳转、新开窗口、前进后退中均可生效，但用户主动刷新页面时应该自动失效。
    - **Expires**
      - **定义：**当服务器返回某个资源带有此Header时，则意味着服务器保证资源在截止时间之前不会发生改变。浏览器可以直接缓存该数据，不再重新发出请求。从HTTP 1.0开始提供。
      - HTTP/1.1 200 OK
        Expires: Wed, 8 Apr 2020 07:28:00 GMT
      - **缺陷：**受限于客户端本地时间；无法处理设计用户身份的私有资源；无法描述不缓存的语义。
    - **Cache-Control**
      - **定义：**与Expires类似，从HTTP 1.1开始提供。与Expires相比，可同时存在于请求和响应Header中，与Expires同时存在时以Cache-Control为准。
      - HTTP/1.1 200 OK
        Cache-Control: max-age=600
      - **参数max-age和s-maxage：**表示资源缓存有效期的相对时间，单位为秒。s-maxage为共享缓存的有效时间。
      - **参数public和private：**public表示可以被CDN、代理等缓存的资源，private表示只允许被用户的客户端缓存的私有资源。
      - **参数no-cache和no-store：**no-cache表示不使用缓存过期的数据，即每次请求都判断资源是否过期，从服务器获取资源还是从客户端缓存获取。no-store表示彻底不缓存资源，每次请求都从服务器获取资源。
      - **参数no-transform：**禁止以任何形式修改资源，例如禁止CDN、透明代理压缩资源。
      - **参数min-fresh和only-if-cached：**仅用于客户端请求Header，min-fresh用于建议服务器返回一个不少于该时间的缓存资源，即包含在max-age内，但不少于min-fresh。only-if-cached表示客户端仅使用事先缓存的资源进行响应，若缓存不命中则返回503/Service Unavaliable。
      - **参数must-revalidate和proyx-revalidate：**must-revalidate表明资源过期后，一定要从服务器进行获取，即超过max-age后等同于no-cache。proxy-revalidate与前者类似，用于提示代理和CDN等设备。
  - **协商缓存**
    - **定义：**不同于强制缓存基于时效性，协商缓存基于变化检测，在一致性上有更好表现，但需要多一次变化检测的交互开销，性能稍差一些。
    - **生效场景：**在浏览器的地址输入、页面链接跳转、新开窗口、前进后退中均可生效，**并且用户主动刷新页面时同样生效**，只有用户强制刷新时才会失效。
    - **与强制缓存并存：**当强制缓存存在时，直接从强制缓存返回资源，无需进行变化检测；当强制缓存超时失效时，或者使用no-cache和must-revalidate时，协商缓存仍可以正常工作。
    - **Last-Modified和If-Modified-Since**
      - **定义：**Last-Modified用于响应Header，表明资源的最后修改时间。客户端对该资源的再次请求时，会使用If-Modified-Since加上这个最后修改时间发送回服务器。若服务器发现资源在该时间后没有修改，则直接返回304/Not Modified；若修改则返回完整资源信息和新的最后修改时间。
      - HTTP/1.1 304 Not Modified
        Cache-Control：public，max-age=600
        Last-Modified：Wed, 8 Apr 2020 15:31:30 GMT
      - HTTP/1.1 200 OK
        Cache-Control：public，max-age=600

Last-Modified：Wed, 8 Apr 2020 15:31:30 GMT
Content

- **ETag和 If-None-Match**
  - **定义：**ETag是服务端的响应Header，用于告诉客户端资源的唯一标识，由服务端来实现标识的生成。对于该资源，客户端进行再次请求时会一并发送此标识，检测资源是否发生了变化。
  - HTTP/1.1 304 Not-Modified
    Cache-Control：public，max-age=600
    Etag："asdiw-jias-asjidwn"
  - HTTP/1.1 200 OK
    Cache-Control：public，max-age=600
    Etag："asdiw-jias-asjidwn"
    Content
  - **优势：**Etag是HTTP中一致性最强的缓存机制，Last-Modified只能精确到秒级，若某个文件在一秒内被修改多次或者某些文件定期生成但内容并未发生变化，在这些情况下Last-Modified没有太有效的作用。
  - **缺陷：**Etag是HTTP中性能最差的缓存机制，每次请求时，服务端都要进行资源的Etag计算。Etag与Last-Modified可以共存，优先验证ETag。

## 域名解析

- **DNS解析步骤**
  - 1、客户端先检查本地的 DNS 缓存，查看是否存在可用的该域名地址记录。DNS 以存活时间TTL来衡量缓存有效情况，并依靠 TTL 超期后重新获取来保证一致性。
  - 2、客户端将地址发送给本机操作系统中配置的本地 DNS
  - 3、本地 DNS 收到查询请求后，按照
    “是否有www.icyfenix.com.cn的权威服务器”→
    “是否有icyfenix.com.cn的权威服务器”→
    “是否有com.cn的权威服务器”→
    “是否有cn的权威服务器”的顺序，依次查询自己的地址记录，如果都没有查询到，就会一直找到最后点号代表的根域名服务器为止。
  - 4、假设本地 DNS 是全新的，不存在任何域名的权威服务器记录，当 DNS 查询请求按顺序一直查到根域名服务器之后，将会得到“cn的权威服务器”的地址记录，逐级获取下一级的地址记录，最后找到能够解释www.icyfenix.com.cn的权威服务器地址。
  - 5、通过“www.icyfenix.com.cn的权威服务器”，查询www.icyfenix.com.cn的地址记录，地址记录并不一定就是指 IP 地址。
- **DNS服务器概念**
  - **权威域名服务器（Authoritative DNS）：**是指负责翻译特定域名的 DNS 服务器，“权威”意味着这个域名应该翻译出怎样的结果是由它来决定的。
  - **根域名服务器（Root DNS）**是指固定的、无需查询的顶级域名服务器，可以默认已内置在操作系统代码之中。全世界一共有 13 组根域名服务器。 DNS 主要采用 UDP 传输协议来进行数据交换，未分片的 UDP 数据包在 IPv4 下最大有效值为 512 字节，最多可以存放 13 组地址记录。
- **优势：**每种记录类型中可以包括多条记录，一个域名下可以配置多条不同的 A 记录为例，权威服务器可以根据策略来进行选择，如**智能线路**，根据访问者所处的不同地区、不同服务商等因素来确定返回最合适的 A 记录，达到智能加速的目的。
- **缺陷：**DNS系统多级分流的设计使得DNS能够经受住全球网络流量不间断的冲击，而典型的问题是**响应速度**，当极端情况（各级服务器均无缓存）下的域名解析可能导致每个域名都必须递归多次才能查询到结果，影响传输的响应速度。常用**DNS预取**的前端优化方式来避免此问题。
- **安全风险：**DNS 的分级查询意味着每一级都有可能受到中间人攻击的威胁，产生被劫持的风险。要攻陷位于顶层的服务器和链路非常困难，但很多位于底层或者来自本地运营商的 Local DNS 服务器的安全防护则相对松懈，甚至不少地区的运营商自己就会主动进行劫持。
- **DNS新方式HTTPDNS：**将原本的 DNS 解析服务开放为一个基于 HTTPS 协议的查询服务，替代基于 UDP 传输协议的 DNS 域名解析，通过程序代替操作系统直接从权威 DNS 或者可靠的 Local DNS 获取解析数据，从而绕过传统 Local DNS。

## 传输链路

- **背景：**经过客户端缓存的节流、经过 DNS 服务的解析指引，程序发出的请求流量便正式离开客户端，经过传输链路到达服务端。而**程序发出的请求能否与应用层、传输层协议提倡的方式相匹配，对传输的效率也会有极大影响**。
- **连接数优化**
  - **HTTP和TCP设计的”不协调“：** HTTP/3 以前是以 TCP 为传输层的应用层协议，而**HTTP 传输对象的主要特征是数量多、时间短、资源小、切换快，而TCP 协议本身是面向于长时间、大数据传输来设计的**（慢启动和三次握手），在长时间尺度下，它连接建立的高昂成本才不至于成为瓶颈。
  - **解决方案：**减少发出的请求数量，同时增加客户端到服务端的连接数量。除了使用前端的一些优化技巧，更重要的是从协议层面去解决连接成本过高的问题。
  - **Keep-Alive**
    - **定义：**HTTP1.1时默认打开的连接复用技术，原理是让客户端对同一个域名长期持有一个或多个不会用完即断的 TCP 连接。典型做法是在客户端维护一个 FIFO 队列，每次取完数据之后一段时间内不自动断开连接，以便获取下一个资源时直接复用，避免创建 TCP 连接的成本。
    - **缺陷：**常见队首阻塞问题，若队列中有10个资源请求和1个TCP连接，即使它们可以并发请求，但是缺乏对返回的资源包归属的判定，只能按照顺序逐渐返回资源的数据包，若第一个资源请求操作未收到所有资源数据包，其他资源请求只能等待。
  - **HTTP管道：**HTTP 服务器中也建立类似客户端的 FIFO 队列，让客户端一次将所有要请求的资源名单全部发给服务端，由服务端来安排返回顺序，管理传输队列。虽然无法完全避免队首阻塞的问题，但服务端能够较为准确地评估资源消耗情况，进而能够更紧凑地安排资源传输。由于 HTTP 管道需要多方共同支持，协调起来相当复杂，推广得并不算成功。
  - **HTTP/2多路复用**
    - **定义：**在 HTTP/1.x 中，HTTP 请求就是传输过程中最小粒度的信息单位，在 HTTP/2 中，帧才是最小粒度的信息单位。每个帧都附带一个流 ID 以标识这个帧属于哪个流。在同一个 TCP 连接中传输的多个数据帧就可以根据流 ID 轻易区分出开来，在客户端毫不费力地将不同流中的数据重组出不同 HTTP 请求和响应报文来
    - **优势：**有了多路复用的支持，HTTP/2 就可以对每个域名只维持一个 TCP 连接来以任意顺序传输任意数量的资源，既减轻了服务器的连

接压力，开发者也不用去考虑域名分片等突破浏览器对每个域名的连接数限制了；没有了 TCP 连接数的压力，就无须刻意压缩 HTTP 请求了，在有些情况下压缩请求、节省Header并不奏效。

- **传输压缩**
  - **背景：**HTTP 支持GZip压缩，由于 HTTP 传输的主要内容是文本数据，压缩的收益较高，传输数据量一般会降至20%左右。而对于不适合压缩的资源，Web 服务器则能根据 MIME 类型来判断是否对响应进行压缩，已经采用过压缩算法存储的资源，便不会被二次压缩，空耗性能。早期，采用**“静态预压缩”**，静态资源先预先压缩存放起来，而现在由服务器对符合条件的请求将在输出时进行**“即时压缩”**。
  - **压缩与持久连接的冲突**
    - **冲突：**使用即时压缩后，服务器再也没有办法给出Content-Length 这个响应 Header 了；使用持久连接后，不再依靠 TCP 连接是否关闭来判断资源请求是否结束。因此，HTTP/1.0的话，由于缺乏判断资源请求是否结束的标志，持久链接和即时压缩只能二选其一。
    - **解决方式：**HTTP/1.1增加了“**分块传输编码**”的资源结束判断机制，在响应 Header 中加入“Transfer-Encoding: chunked”之后，报文中的 Body 改为用一系列分块来传输。每个分块包含十六进制的长度值和对应长度的数据内容，长度值独占一行，数据从下一行开始。最后以一个长度值为 0 的分块来表示资源结束。
- **快速UDP网络连接**
  - **背景：**从职责上讲，持久连接、多路复用、分块编码这些能力，已经或多或少超过了应用层的范畴。要从根本上改进 HTTP，必须直接替换掉 HTTP over TCP 的根基，即 TCP 传输协议，这便最新一代 HTTP/3 协议的设计重点。
  - **优势**
    - QUIC 会以 UDP 协议为基础，没有丢包自动重传的特性，可靠传输能力完全由自己来实现，**可以对每个流能做单独控制，即使某个流发生错误，仍然可以独立地继续为其他流提供服务。**TCP 协议接到数据包丢失或损坏通知之前，可能已经收到了大量数据，在纠正错误之前，其他的正常请求都会等待甚至被重发，这也是HTTP/2 未能解决传输大文件慢的根本原因。
    - QUIC 在移动设备上的优势体现在网络切换时的响应速度上，譬如当移动设备在不同IP之间切换，若使用 TCP 协议，现存所有连接都必定会超时、中断，然后重新创建。**QUIC 提出了连接标识符的概念，唯一标识客户端与服务器之间的连接，无须依靠 IP 地址，只需向服务端发送包含标识符的数据包即可重用既有连接**

## 内容分发网络CDN

- **定义：**客户端缓存、域名解析、传输链路的综合应用，内容分发网络的工作过程，主要涉及路由解析、内容分发、负载均衡和所能支持的 CDN 应用内容四个方面。
- **互联网系统的速度决定**
  **因素（网络传输角度）**
  - **网站服务器**接入网络运营商的链路所能提供的**出口带宽**。
  - **用户客户端**接入网络运营商的链路所能提供的**入口带宽**。
  - 从网站到用户之间经过的**不同运营商之间互联节点的带宽**，一般来说两个运营商之间只有固定的若干个点是互通的，所有跨运营商之间的交互都要经过这些点。
  - 从网站到用户之间的**物理链路传输时延ping**
  - 除了用户客户端入口带宽只能通过换更好的宽带才能解决之外，其余三个都能通过内容分发网络来显著改善。一个运作良好的内容分发网络，**能为互联网系统解决跨运营商、跨地域物理距离所导致的时延问题，能为网站流量带宽起到分流、减负的作用。**
- **路由解析**
  - **方式：**CDN将用户请求路由到特定资源服务器上是依靠DNS服务器来实现的。
  - **工作过程**
    - 1、架设好“icyfenix.cn”的服务器后，将IP地址在CDN服务商上注册为源站，注册后会得到一个 CNAME，即“icyfenix.cn.cdn.dnsv1.com.”。
    - 2、将得到的CNAME在购买域名的DNS服务商上注册为一条CNAME 记录。
    - 3、当第一位用户来访站点时，首先发生一次未命中缓存的 DNS 查询，域名服务商解析出 CNAME 后，返回给本地 DNS，至此之后链路解析的主导权就开始由内容分发网络的调度服务接管了。
    - 4、本地 DNS 查询 CNAME 时，由于能解析该 CNAME 的权威服务器只有 CDN 服务商所架设的权威DNS，因此DNS服务将根据特定均衡策略和参数，在全国各地能提供服务的 CDN 缓存节点中挑选一个最适合的，将它的 IP 代替源站的 IP 地址，返回给本地 DNS。
    - 5、浏览器从本地 DNS 拿到 IP 地址，将该 IP 当作源站服务器来进行访问，此时该 IP 的 CDN 节点上可能没有缓存过源站的资源，经过内容分发后的 CDN 节点，就有能力代替源站向用户提供所请求的资源。
- **内容分发**
  - **背景：**CDN的缓存节点接管了用户向服务器发出的资源请求，而缓存节点中必须有用户请求的资源副本，才可能代替源站来响应用户请求。其中，包括了两个子问题，即“如何获取源站资源（内容分发）”和“如何管理/更新资源”。
  - **主流内容分发方式**
    - **主动分发Push：**分发由源站主动发起，将内容从源站或者其他资源库推送到用户边缘的各个 CDN 缓存节点上。可以采用任何传输方式、推送策略、推送时间，只要与更新策略相匹配即可。由于主动分发通常需要源站、CDN 服务双方提供程序 API 接口层面的配合，所以它对源站并不是透明的，只对用户一侧单向透明。主动分发一般用于网站要预载大量资源的场景。
    - **被动回源Pull：**被动回源是由用户访问所触发全自动、双向透明的资源缓存过程。当资源首次被请求时CDN缓存节点发现没有该资源，将实时从源站中获取，响应时间可认为是资源从源站到 CDN 缓存节点+资源从 CDN 发送到用户的时间之和，首次访问通常比较慢的，但并不一定比用户直接访问源站更慢。被动回源不适合应用于数据量较大的资源。优点是不需要源站在程序上做任何的配合，使用方便，是小型站点使用 CDN 服务的主流选择。
  - **管理/更新资源**
    - **背景：**HTTP 协议中关于缓存的 Header 定义中有对 CDN 这类共享缓存的一些指引性参数，如Cache-Control的 s-maxage，但是否要遵循，完全取决于 CDN 本身的实现策略。如果CDN完全照着 HTTP Headers 来控制缓存失效和更新，效果反而会相当的差，因此，CDN 缓存的管理就不存在通用的准则
    - **做法：**最常见的做法是超时被动失效与手工主动失效相结合。超时失效是指给予缓存资源一定的生存期，超过了生存期就在下次请求时重新被动回源一次。而手工失效是指 CDN 服务商一般会提供给程序调用来失效缓存的接口，在网站更新时，由持续集成的流水线自动调用该接口来实现缓存更新。
- **CDN应用**

- **加速静态资源**
- **安全防御：**CDN 在广义上可以视作网站的堡垒机，源站只对 CDN 提供服务，由 CDN 来对外界其他用户服务，这样恶意攻击者就不容易直接威胁源站。
- **协议升级：**CDN可以实现源站是 HTTP 协议的，对外开放的网站是基于 HTTPS。
  可以实现源站到 CDN 是 HTTP/1.x 协议，CDN 提供的外部服务是 HTTP/2 3；
  可以实现源站是基于 IPv4 网络的，CDN 提供的外部服务支持 IPv6 网络。
- **修改资源：**CDN 可以在返回资源给用户的时候修改任何内容，实现不同的目的。
- **访问控制：**CDN 可以实现 IP 黑/白名单功能，根据不同的来访 IP 提供不同的响应结果、IP的访问流量来实现 QoS 控制、根据 HTTP 的 Referer 来实现防盗链等。

## 负载均衡

- **背景：**除了DNS层面的负载均衡，还有网络请求进入数据中心入口之后的其他级次的负载均衡。
- **基础概念**
  - **四层负载均衡：**四层的意思是说这些工作模式的共同特点是维持着同一个 TCP 连接，不是说只工作在第四层。这些模式主要都是工作在二层（数据链路层，改写 MAC 地址）和三层（网络层，改写 IP 地址）上。
  - **七层负载均衡：**流量已经到达目标主机上，谈不上流量转发，只能做代理。
  - 四层负载均衡的优势是性能高，七层负载均衡的优势是功能强；做多级混合负载均衡，通常应是低层的负载均衡在前，高层的负载均衡在后。
- **数据链路层负载均衡**
  - **工作原理：**修改请求的数据帧中的 MAC 目标地址，让原本发送给负载均衡器的请求的数据帧，被二层交换机根据新的 MAC 目标地址转发到服务器集群中对应的服务器的网卡上，即真实服务器获得了一个原本目标并不是发送给它的数据帧。
  - **限制：**由于第三层的IP数据包中包含了源和目标的 IP 地址，只有真实服务器IP 地址与数据包中的目标 IP 地址一致，数据包才能被正确处理。因此，需要把真实物理服务器集群所有机器的虚拟 IP 地址配置成与负载均衡器的虚拟 IP 一样，这样经均衡器转发后的数据包就能在真实服务器中顺利地使用。
  - **优势：**响应结果不再需要通过负载均衡服务器进行地址交换，可将响应结果的数据包直接从真实服务器返回给用户的客户端，避免负载均衡器网卡带宽成为瓶颈，因此数据链路层的负载均衡效率是相当高的。
  - **缺陷：**二层负载均衡器直接改写目标 MAC 地址的工作原理决定了它与真实的服务器的通信必须是二层可达的，通俗地说就是必须位于同一个子网当中，无法跨 VLAN。
  - **适用场景：**优势效率高和劣势不能跨子网共同决定了数据链路层负载均衡最适合用来做数据中心的第一级均衡设备，用来连接其他的下级负载均衡器。
- **网络层负载均衡**
  - **工作原理：**与数据链路层负载均衡原理类似，通过改变IP数据包的 IP 地址来实现数据包的转发。
  - **修改方式**
    - **IP隧道：**保持原来的数据包不变，新创建一个数据包，把原来数据包的 Headers 和 Payload 整体作为另一个新的数据包的 Payload，在这个新数据包的 Headers 中写入真实服务器的 IP 作为目标地址，然后把它发送出去。
    - **IP隧道优势：**由于没有修改原有数据包中的任何信息，I 隧道的转发模式仍然具备三角传输的特性，即负载均衡器转发来的请求，可以由真实服务器去直接应答。且由于 IP 隧道工作在网络层，所以可以跨越 VLAN，因此摆脱了直接路由模式中网络侧的约束。
    - **IP隧道缺陷：**要求真实服务器必须支持IP 隧道协议，所幸几乎所有的 Linux 系统都支持 IP 隧道协议；这种模式仍必须通过专门的配置，必须保证所有的真实服务器与均衡器有着相同的虚拟 IP 地址，因为回复该数据包时，需要使用这个虚拟 IP 作为响应数据包的源地址，这样客户端收到这个数据包时才能正确解析。
    - **改变目标数据包：**把数据包 Headers 中的目标地址改掉，通过三层交换机转发给真实服务器。但是如果真实服务器直接将应答包返回客户端的话，应答数据包的源 IP 是客户端不认识的真实服务器的 IP，无法正常处理。因此，只能让应答流量继续回到负载均衡，由负载均衡把应答包的源 IP 改回自己的 IP，再发给客户端。这种通过网络地址转换的负载均衡器成为NAT负载均衡器。
    - **NAT模式优势：**真实服务器不必支持IP隧道，真实服务器与负载均衡器不必有相同虚拟IP。
    - **NAT模式缺陷：**流量压力比较大的时候，NAT 模式的负载均衡会带来较大的性能损失，比起直接路由和 IP 隧道模式，甚至会出现数量级上的下降。
    - **SNAT：**均衡器在转发时，不仅修改目标 IP 地址，连源 IP 地址也一起改了，源地址就改成均衡器自己的 IP。好处是真实服务器无须配置网关就能够让应答流量经过正常的三层路由回到负载均衡器上。缺点是真实服务器处理请求时无法拿到客户端IP 地址，有一些需要根据目标 IP 进行控制的业务逻辑就无法进行。
- **应用层负载均衡**
  - **与四层负载均衡区别：**四层负载均衡时，客户端到响应请求的真实服务器维持着同一条 TCP 通道。工作在四层之后的负载均衡模式无法进行转发，只能进行代理，此时真实服务器、负载均衡器、客户端三者之间由两条独立的 TCP 通道来维持通信。
  - **七层负载均衡优势：**七层负载均衡器属于反向代理中的一种，只论网络性能比不过四层均衡器，它比四层均衡器至少多一轮 TCP 握手，有着跟 NAT 转发模式一样的带宽问题，而且通常要耗费更多的 CPU，因为可用的解析规则远比四层丰富。它工作在应用层，可以感知应用层通信的具体内容，往往能够做出更明智的决策。
  - **可实现功能**
    - 所有 CDN 可以做的缓存方面的工作七层均衡器全都可以实现，譬如静态资源缓存、协议升级、安全防护、访问控制等。
    - 七层均衡器可以实现更智能化的路由。譬如根据 Session 路由，以实现亲和性的集群；根据 URL 路由，实现专职化服务，相当于网关；根据用户身份路由，实现对部分用户的特殊服务，如某些站点的贵宾服务器等。
    - 某些安全攻击可以由七层均衡器来抵御
    - 链路治理措施都需要在七层中进行，譬如服务降级、熔断、异常注入等
- **均衡策略与实现**
  - 轮循均衡（Round Robin）
  - 权重轮循均衡（Weighted Round Robin）
  - 随机均衡（Random）

- 权重随机均衡（Weighted Random）
- 一致性哈希均衡（Consistency Hash）
- 响应速度均衡（Response Time）
- 最少连接数均衡（Least Connection）

- **服务端缓存**
  - **缺陷**
    - **从开发角度来说：**引入缓存会提高系统复杂度，因为你要考虑缓存的失效、更新、一致性等问题。
    - **从运维角度来说：**缓存会掩盖掉一些缺陷，让问题在更久的时间以后，出现在距离发生现场更远的位置上。
    - **从安全角度来说：**缓存可能泄漏某些保密数据，也是容易受到攻击的薄弱点。
  - **引入服务端缓存理由**
    - **为缓解 CPU 压力而做缓存：**譬如把方法运行结果存储起来、把原本要实时计算的内容提前算好、把公用数据进行复用，节省 CPU 算力，顺带提升响应性能。
    - **为缓解 I/O 压力而做缓存：**譬如把原本对网络、磁盘等较慢介质的读写访问变为对内存等较快介质的访问，将原本对单点部件如数据库的读写访问变为到可扩缩部件如缓存中间件的访问，顺带提升响应性能。
    - 缓存是典型以空间换时间来提升性能的手段，但**出发点是缓解 CPU 和 I/O 资源在峰值流量下的压力，“顺带”地提升响应性能** 。如果可以通过增强 CPU、I/O 本身的性能来满足需要的话，那升级硬件往往是更好的解决方案，即使需要一些额外的投入成本，也通常要优于引入缓存后可能带来的风险。
  - **缓存属性**
    - **吞吐量：**缓存的吞吐量使用每秒操作数OPS 值来衡量，反映了对缓存进行并发读、写操作的效率，即缓存本身的工作效率高低。
      - **缓存中最主要数据竞争：**读取数据的同时，也会伴随着对数据状态的写入操作，写入数据的同时，也会伴随着数据状态的读取操作。例如，读取时要同时更新数据的最近访问时间和访问计数器的状态，以实现缓存的淘汰策略；又或者读取时要同时判断数据的超期时间等信息，以实现失效重加载等其他扩展功能。
      - **状态维护方式一：**以 Guava Cache 为代表的同步处理机制，即在访问数据时一并完成缓存淘汰、统计、失效等状态变更操作，通过分段加锁等来尽量减少竞争。
      - **状态维护方式二：**以 Caffeine 为代表的异步日志提交机制，将对数据的读、写过程看作是日志即对数据的操作指令的提交过程。异步提交的日志已经将原本在 Map 内的锁转移到日志的追加写操作上，日志里优化的余地就比在 Map 中要大得多。
    - **命中率：**缓存的命中率即成功从缓存中返回结果次数与总请求次数的比值，命中率越低，引入缓存的收益越小，价值越低。与命中率相关的是缓存的淘汰策略。
      - **FIFO：**优先淘汰最早进入被缓存的数据。越是频繁被用到的数据，可能会越早被存入缓存之中。如果采用这种淘汰策略，很可能会大幅降低缓存的命中率。
      - **LRU：**优先淘汰最久未被使用访问过的数据。LRU 通常会采用如 LinkedHashMap来实现。LRU尤其适合处理短时间内频繁访问的热点对象。但如果热点数据在系统中经常被频繁访问，但最近一段时间因为某种原因未被访问过，此时这些热点数据依然要面临淘汰的命运，LRU 依然可能错误淘汰价值更高的数据。
      - **LFU：**优先淘汰最不经常使用的数据。LFU 会给每个数据添加一个访问计数器，需要淘汰时就清理计数器数值最小的那批数据。LFU可以解决热点数据间隔一段时间不访问就被淘汰的问题，但需要对每个缓存数据维护一个计数器，每次访问都要更新，这样做会带来高昂的维护开销；另一个问题是不便于处理随时间变化的热度变化，譬如某个曾经频繁访问的数据现在不需要了，它也很难自动被清理出缓存。
      - **TinyLFU：**TinyLFU 是 LFU 的改进版本。为了避免访问修改计数器带来的性能负担，采用 Sketch 对访问数据进行分析；采用了基于“滑动时间窗”的热度衰减算法，以此解决“旧热点”数据难以清除的问题。
      - **W-TinyLFU：**TinyLFU 的改进版本。应对短时间的突发访问是 LRU 的强项，它结合了 LRU 和 LFU 两者的优点，将新记录暂时放入一个前端 LRU 缓存里面，让这些对象累积热度，如果能通过 TinyLFU 的过滤器，再进入主缓存中存储。
    - **扩展功能：**缓存除了基本读写功能外，还提供哪些额外的管理功能，譬如最大容量、失效时间、失效事件、命中率统计，等等。
    - **分布式支持：**缓存可分为“进程内缓存”和“分布式缓存”两大类，前者只为节点本身提供服务，无网络访问操作，速度快但缓存的数据不能在各个服务节点中共享，后者则相反。
      - **复制式缓存：**复制式缓存可以看作是“能够支持分布式的进程内缓存”，缓存中所有数据在分布式集群的每个节点里面都存在有一份副本，读取数据时直接从当前节点的进程内存中返回；当数据发生变化时，将变更同步到集群的每个节点中，复制性能随着节点的增加呈现平方级下降，变更数据的代价十分高昂。
      - **集中式缓存：**分布式缓存的主流形式，读写都需要网络访问，但不会随着集群节点数量增加产生额外负担，坏处是不可能达到进程内缓存的高性能。与使用缓存的应用分处在独立的进程空间中，能够为异构语言提供服务。如果要缓存对象等复杂类型的话，基本上就只能靠序列化来支撑具体语言的类型系统，不仅有序列化的成本，还很容易导致传输成本也显著增加。主流的是Redis。
      - 根据分布式缓存集群是否能保证数据一致性，可以将它分为 AP 和 CP 两种类型，Redis是典型的AP缓存集群。
      - **多级缓存：**使用进程内缓存做一级缓存，分布式缓存做二级缓存，如果能在一级缓存中查询到结果就直接返回，否则便到二级缓存中去查询，再将二级缓存中的结果回填到一级缓存，以后再访问该数据就没有网络请求了。如果二级缓存也查询不到，就发起对最终数据源的查询，将结果回填到一、二级缓存中去。
  - **缓存风险**
    - **缓存穿透：**如果查询的数据在数据库中根本不存在的话，缓存里自然也不会有，这类请求的流量每次都不会命中，每次都会触及到末端的数据库，缓存就起不到缓解压力的作用了。
      - 对于业务逻辑本身就不能避免的缓存穿透，可以约定在一定时间内对返回为空的 Key 值依然进行缓存，使得在一段时间内缓存最多被穿透一次。后续业务在数据库中对该 Key 值插入了新记录，那应当在插入之后主动清理掉缓存的 Key 值。
      - 对于恶意攻击导致的缓存穿透，通常会在缓存之前设置一个布隆过滤器来解决。如果布隆过滤器给出的判定结果是请求的数据不存在，那就直接返回即可，连缓存都不必去查。
    - **缓存击穿：**如果缓存中某些热点数据忽然因某种原因失效了，譬如由于超期而失效，此时又有多个针对该数据的请求同时发送过来，这些请求将全部未能命中缓存，都到达真实数据源中去，导致其压力剧增。
      - **加锁同步：**以请求该数据的 Key 值为锁，使得只有第一个请求可以流入到真实的数据源中，其他线程采取阻塞或重试策略。进程内缓存施加普通互斥锁，分布式缓存施加分布式锁，数据源就不会同时收到大量针对同一个数据的请求了。
      - **手动管理：**缓存击穿是仅针对热点数据被自动失效才引发的问题，对于这类数据，可以由开发者通过代码来有计划地完成更新、失效，避

免由缓存的策略自动管理。

- **缓存雪崩：**由于大批不同的数据在短时间内一起失效，导致了这些数据的请求都击穿了缓存到达数据源，同样令数据源在短时间内压力剧增。
  - 提升缓存系统可用性，建设分布式缓存的集群。
  - 启用透明多级缓存，各个服务节点一级缓存中的数据通常会具有不一样的加载时间，也就分散了它们的过期时间。
  - 将缓存的生存期从固定时间改为一个时间段内的随机时间，譬如原本是一个小时过期，缓存不同数据时，设置生存期为 55 分钟到 65 分钟之间的某个随机时间。
- **缓存污染：**指缓存中的数据与真实数据源中的数据不一致的现象。为了尽可能的提高使用缓存时的一致性，已经总结不少更新缓存可以遵循设计模式，譬如 Cache Aside等。
  - 读数据时，先读缓存，没有的话再读数据源，然后将数据放入缓存，再响应请求。
  - 写数据时，先写数据源，然后失效（而不是更新）掉缓存。

# 架构安全性

- **认证**
  - **定义：**系统如何正确分辨出操作用户的真实身份。
  - **认证方式**
    - **通信信道上的认证：**建立通信连接之前，要先证明你是谁。在网络传输场景中的典型是基于 SSL/TLS 传输安全层的认证。
    - **通信协议上的认证：**请求获取资源之前，要先证明你是谁。在互联网场景中的典型是基于 HTTP 协议的认证。
    - **通信内容上的认证：**使用提供服务之前，要先证明你是谁。在万维网场景中的典型是基于 Web 内容的认证。
  - **HTTP认证**
    - **前提**
      - **认证方案**：是指生成用户身份凭证的某种方法，源于 HTTP 协议的认证框架
      - HTTP通用认证框架要求所有支持 HTTP 协议的服务器，**在未授权的用户意图访问服务端保护区域的资源时，应返回 401 Unauthorized，同时在响应报文头里附带以下两个分别代表网页认证和代理认证的 Header 之一**，告知客户端应该采取何种方式产生能代表访问者身份的凭证信息，**验证失败返回403 Forbidden 错误。**
    - **服务端响应Header**
      - **WWW-Authenticate: <认证方案> realm=<保护区域的描述信息>**
        **Proxy-Authenticate: <认证方案> realm=<保护区域的描述信息>**
    - **客户端请求Header**
      - **Authorization: <认证方案> <凭证内容>**
        **Proxy-Authorization: <认证方案> <凭证内容>**
    - **常见认证方案**
      - **HTTP Basic：**以演示为目的的认证方案，也应用于一些不要求安全性的场合。
      - **Digest：**HTTP 摘要认证，可视为 Basic 认证的改良版本，针对 Base64 明文发送的风险，把用户名和密码加盐后再通过 MD5/SHA 等哈希算法取摘要发送出去。
      - **Bearer：**基于 OAuth 2（同时涉及认证与授权的协议） 规范来完成认证。
      - **HOBA：**是一种基于自签名证书的认证方案。一类是采用 CA层次结构的模型，由 CA 中心签发证书；另一种是以 IETF 的 Token Binding 协议为基础的 OBC自签名证书模型。
      - **AWS4-HMAC-SHA256：**亚马逊 AWS 基于 HMAC-SHA256 哈希算法的认证。
      - **NTLM / Negotiate：**这是微软公司NTLM用到的两种认证方式。
      - **Windows Live ID：**微软开发并提供的“统一登入”认证。
      - **Twitter Basic：**Twitter网站所改良的 HTTP 基础认证。
  - **Web认证**
    - **背景：**以 HTTP 协议为基础的认证框架也只能面向传输协议设计，依靠内容来实现的认证方式肯定希望是由系统本身的功能去完成的。
    - **WebAuthn**
      - **背景：**万维网联盟W3C批准了由FIDO（安全、开放、防钓鱼、无密码认证标准的联盟）领导起草的世界首份 Web 内容认证的标准“WebAuthn”。直接采用生物识别或者实体密钥来作为身份凭证，从根本上消灭了用户输入错误产生的校验需求和防止机器人模拟产生的验证码需求等问题，甚至可以省掉表单界面。
      - **注册流程**
        - 1、服务端先暂存用户提交的用户信息，生成一个随机字符串Challenge和用户的 UserID，返回给客户端
        - 2、客户端的 WebAuthn API 接收到 Challenge 和 UserID，把这些信息发送给验证器，可理解为用户设备上 TouchID、FaceID、实体密钥等认证设备统一接口。
        - 3、验证器提示用户进行验证，结果是生成一个密钥对（公钥和私钥），由验证器存储私钥、用户信息以及当前的域名。然后使用私钥对 Challenge 进行签名，并将签名结果、UserID 和公钥一起返回客户端。
        - 4、服务器核验信息，检查 UserID 与之前发送的是否一致，并用公钥解密后得到的结果与之前发送的 Challenge 相比较，一致即表明注册通过，由服务端存储该 UserID 对应的公钥。
      - **登录流程**
        - 1、服务器返回随机字符串 Challenge、用户 UserID
        - 2、浏览器将 Challenge 和 UserID 转发给验证器
        - 3、验证器提示用户进行认证操作，通过后以存储的私钥加密 Challenge，然后返回给浏览器
        - 4、服务端接收到浏览器转发来的被私钥加密的 Challenge，以此前注册时存储的公钥进行解密，如果解密成功则宣告登录成功
      - **优势**
        - 采用非对称加密的公钥、私钥替代传统的密码，私钥是保密的，只有验证器需要知道它，没有人为泄漏的可能；除了得知私钥外，没有其他途径能够生成可被公钥验证为有效的签名，可以通过公钥是否能够解密来判断最终用户的身份是否合法。
        - 解决了传统密码在网络传输上的风险
        - 为登录过程带来极大的便捷性，而且彻底避免了用户在一个网站上泄漏密码，所有使用相同密码的网站都受到攻击的问题。
  - **JAVA安全框架**

- **主流框架**
  - **Apache Shiro**
  - **Spring Security**
- **功能**
  - **认证功能：**以 HTTP 协议中定义的各种认证、表单等认证方式确认用户身份。
  - **安全上下文：**用户获得认证之后，应用可以得知用户的基本资料、权限、角色等。
  - **授权功能：**判断并控制认证后的用户对什么资源拥有哪些操作许可
  - **密码的存储与验证：**密码是烫手的山芋，存储、传输还是验证都应谨慎处理

## 授权

- **定义：**系统如何控制一个用户该看到哪些数据、能操作哪些功能。
- **包含问题**
  - **确保授权的过程可靠：**对于单一系统来说，授权的过程是比较容易做到可控的。而在涉及多方的系统中，需要既让第三方系统能够访问到所需的资源，又能保证其不泄露用户的敏感数据。常用的多方授权协议主要有 OAuth2 和 SAML 2.0。
  - **确保授权的结果可控：**授权的结果用于对程序功能或者资源的访问控制，成理论体系的权限控制模型有自主访问控制DAC、强制访问控制MAC、基于属性的访问控制ABAC，还有最为常用的基于角色的访问控制RBAC。
- **RBAC**
  - **背景：**访问控制模型，实质上都是在解决“谁拥有什么权限去操作哪些资源”。为了避免对每一个用户设定权限，RBAC 将权限从用户身上剥离，改为绑定到“角色”上，将权限控制变为解决“角色拥有操作哪些资源的许可”。
  - **优势**
    - 不仅简化配置操作，还满足了“最小特权原则”。角色拥有许可的数量是根据完成该角色工作职责所需的最小权限来赋予的。
    - RBAC 允许对不同角色之间定义关联与约束，进一步强化它的抽象描述能力。如不同的角色之间可以有继承性，不同角色之间也可以具有互斥性。
  - **建立访问控制模型的基本目的是为了管理垂直权限和水平权限**。垂直权限即功能权限，水平权限即数据权限管理。数据权限是很难抽象与通用的，基本只能由信息系统自主来来完成。
- **OAuth2**
  - **直接将密码告知第三方应用出现的问题**
    - **密码泄漏：**如果第三方应用被黑客攻破，将导致密码也同时被泄漏。
    - **访问范围：**第三方应用将有能力读取、修改、删除、更新源服务端上的所有资源。
    - **授权回收：**只有修改密码才能回收授予给第三方应用的权力，授权的应用可能有许多，修改了密码意味着所有别的第三方的应用程序的授权会全部失效。
  - **解决问题方法：**以令牌Token代替用户密码作为授权的凭证。令牌被泄漏，也不会导致密码的泄漏；令牌上可以设定访问资源的范围以及时效性；每个应用都持有独立的令牌，哪个失效都不会波及其他。
  - **授权码模式**
    - **授权过程**
      - 1、第三方应用将资源所有者（用户）导向授权服务器的授权页面，并向授权服务器提供 ClientID 及用户同意授权后的回调 URI，这是一次客户端页面转向。
      - 2、授权服务器根据 ClientID 确认第三方应用的身份，用户决定是否进行授权。
      - 3、用户同意授权，授权服务器转向第1步中提供的回调 URI，并附带上一个授权码和获取令牌的地址作为参数，这是第二次客户端页面转向。
      - 4、第三方应用通过回调地址收到授权码，然后将授权码与自己的 ClientSecret 一起作为参数，向授权服务器提供的获取令牌的服务地址发起请求，换取令牌。要求服务器的地址应与注册时提供的域名处于同一个域中。
      - 5、授权服务器核对授权码和 ClientSecret，向第三方应用授予令牌。令牌中必定要有的是访问令牌，可选的是刷新令牌。访问令牌用于到资源服务器获取资源，有效期较短，刷新令牌用于在访问令牌失效后重新获取，有效期较长。
      - 6、资源服务器根据访问令牌所允许的权限，向第三方应用提供资源。
    - **不同情况的应对**
      - **其他应用冒充第三方应用骗取授权：**ClientID是可以完全公开的，但ClientSecret 应当只有应用自己才知道。发放令牌时必须能够提供ClientSecret才能成功完成。
      - **先发放授权码，再用授权码换令牌：**客户端转向对于用户是可见的，授权码可能会暴露，但由于并没有 ClientSecret也无法换取到令牌，避免了令牌在传输转向过程中被泄漏的风险。
      - **不能直接把访问令牌的时间调长：**通常访问令牌一旦发放，除非超过了令牌中的有效期，否则很难有其他方式让它失效，所以访问令牌的时效性一般设计的比较短，如果还需要继续用，那就定期用刷新令牌去更新。
    - **缺陷：**虽然最严谨，但要求第三方应用必须有应用服务器向授权服务器获取令牌。
  - **隐式授权模式**
    - **与授权码模式区别：**省略掉了通过授权码换取令牌的步骤，第三方应用不需要服务端支持。同时，由于缺乏ClientSecret的保存地，授权服务器不再去验证第三方应用身份。但还会限制第三方应用的回调 URI 地址必须与注册时提供的域名一致。
    - **安全考虑：**要求第三方应用在注册时提供的回调域名与接受令牌的服务处于同一个域内；在隐式模式中明确禁止发放刷新令牌；令牌必须通过 Fragment 带回，Fragment 是不会跟随请求被发送到服务端的，尽最大努力地避免了令牌从操作代理到第三方服务之间的链路存在被攻击而泄漏出去的可能性。
    - **缺陷：**认证服务器到操作代理之间链路的安全，则只能通过 TLS即 HTTPS来保证中间不会受到攻击了，但无法要求第三方应用同样都支持 HTTPS。
  - **密码模式**

        - **与前两种授权模式区别：**授权码模式和隐私模式属于纯粹的授权模式，与认证没有直接的联系。在密码模式里，认证和授权被整合成了同一个过程，第三方应用拿着用户名和密码直接向授权服务器换令牌。
        - **使用场景：**仅限于用户对第三方应用是高度可信任的场景中使用，或者把第三方视作是系统中与授权服务器相对独立的子模块，在物理上独立于授权服务器部署，但是在逻辑上与授权服务器仍同属一个系统。
    - **客户端模式**
        - **定义：**指第三方应用以自己的名义，向授权服务器申请资源许可。通常用于管理操作或者自动处理类型的场景中。微服务架构并不提倡同一个系统的各服务间有默认的信任关系，客户端模式便是一种常用的服务间认证授权的解决方案。
        - **授权过程：**OAuth2 中还有一种与客户端模式类似的授权模式，即设备码模式。设备从授权服务器获取URI地址和用户码，到验证 URI 中输入用户码。设备会一直循环尝试获取令牌，直到拿到令牌或者用户码过期为止。
- **凭证**
    - **定义：**系统如何保证它与用户之间的承诺是双方当时真实意图的体现，是准确、完整且不可抵赖的？
    - **背景：**对待共享状态信息有两种思路，状态维护在服务端还是客户端？以 HTTP 协议的 Cookie-Session 机制为代表的服务端状态存储在三十年来都是主流的解决方案。不过由于分布式系统中共享数据必然会受到 CAP 不兼容的限制，使得客户端状态存储重新得到关注，如之前只在多方系统中采用的 JWT 令牌方案。
    - **Cookie-Session**
        - **背景：**HTTP为了让服务器有办法区分出发送请求的用户是谁，定义了 HTTP 的状态管理机制，在 HTTP 协议中增加了 Set-Cookie 指令，以键值对的方式向客户端发送一组信息，在此后一段时间内的每次 HTTP 请求中，以名为 Cookie Header 附带着重新发回给服务端，以便服务端区分来自不同客户端的请求。
        - **Set-Cookie: id=icyfenix; Expires=Wed, 21 Feb 2020 07:28:00 GMT; Secure; HttpOnly**

          **GET /index.html HTTP/2.0**
          **Host: icyfenix.cn**
          **Cookie: id=icyfenix**
        - **过程：**系统会把状态信息保存在服务端，在 Cookie 里只传输一个无字面意义的、不重复的字符串，习惯上以sessionid/jsessionid为名，服务器在内存中以 Key/Entity 的结构存储每一个在线用户的上下文状态，使用如超时清理等管理措施。
        - **优点**
            - 状态信息都存储于服务器，只要依靠客户端的同源策略和 HTTPS 的传输层安全，保证 Cookie 中的键值不被窃取而出现被冒认身份的情况，就能完全规避掉上下文信息在传输过程中被泄漏和篡改的风险。
            - 服务端有主动的状态管理能力，可根据自己的意愿随时修改、清除任意上下文信息，譬如很轻易就能实现强制某用户下线的这样功能。
        - **单体服务到多节点方案**
            - **牺牲集群一致性：**让均衡器采用亲和式的负载均衡算法，如根据用户 IP 或者 Session 来分配节点，特定用户发出的请求都被分配到固定节点，每个节点都不重复地保存着一部分用户的状态，若这个节点崩溃了，里面的用户状态便完全丢失。
            - **牺牲集群可用性：**让各个节点之间采用复制式的 Session，每一个节点中的 Session 变动都会发送到组播地址的其他服务器上，但 Session 之间组播复制的同步代价高昂，节点越多时，同步成本越高。
            - **牺牲集群分区容忍性：**让普通的服务节点中不再保留状态，将上下文集中放在一个所有服务节点都能访问到的数据节点中进行存储。此时的矛盾是数据节点就成为了单点，一旦数据节点损坏或出现网络分区，整个集群都不再能提供服务。
    - **JWT**
        - **背景：**当服务器存在多个时，把状态信息存储在客户端，每次随着请求发回服务器去。缺点是无法携带大量信息，有泄漏和篡改的风险。前者不好解决，但使用JWT确保信息不被中间人篡改则还是可以实现的。常见的使用方式是附在名为 Authorization的 Header 发送给服务端。
        - **JWT组成部分**
            - **令牌头Header：**包括令牌的类型，统一为typ:JWT以及令牌签名的算法
            - **负载Payload：**真正向服务端传递的信息。针对认证问题，至少应该包含用户信息，针对授权问题，至少应该包含用户拥有什么角色/权限的信息。
            - **签名Signature：**使用在对象头中公开的特定签名算法，通过特定的密钥Secret，由服务器进行保密，对前面两部分内容进行加密计算。确保负载中的信息是可信的、没有被篡改的。
        - **散列消息认证码HMAC：**带有密钥的哈希摘要算法，不仅保证了内容未被篡改过，还保证了该哈希确实是由密钥的持有人所生成的。
        - **优点**
            - 不需要任何一个服务节点保留任何一点状态信息，就能够保障认证服务与用户之间的承诺是双方当时真实意图的体现，是准确、完整、不可篡改、且不可抵赖的。
            - 本身可以携带少量信息，有利于 RESTful API 的设计，较容易地做成无状态服务，在做水平扩展时就不需要像前面 Cookie-Session 方案那样考虑如何部署的问题
        - **缺陷**
            - **令牌难以主动失效：**JWT 令牌一旦签发，理论上就和认证服务器再没有什么瓜葛了，在到期之前就会始终有效。服务器使用管理策略较困难。
            - **相对更容易遭受重放攻击：**Cookie-Session 也是有重放攻击问题的，只是因为 Session 中的数据控制在服务端手上，应对重放攻击会相对主动一些。
            - **只能携带相当有限的数据：**在令牌中存储过多的数据不仅耗费传输带宽，还有额外的出错风险。
            - **必须考虑令牌在客户端如何存储**
            - **无状态也不总是好的**
- **保密**
    - **定义：**系统如何保证敏感数据无法被包括系统管理员在内内外部人员所窃取滥用？
    - **保密的强度**
        - **以摘要代替明文：**如果密码本身比较复杂，简单的哈希摘要至少可以保证即使传输过程中有信息泄漏，也不会被逆推出原信息；即使密码在一个系统中泄漏了，也不至于威胁到其他系统的使用，但这种处理不能防止弱密码被彩虹表攻击所破解。

- **先加盐值再做哈希：**是应对弱密码的常用方法，盐值可以替弱密码建立一道防御屏障，一定程度上防御已有的彩虹表攻击，但并不能阻止加密结果被监听、窃取后，攻击者直接发送加密结果给服务端进行冒认。
- **将盐值变为动态值能有效防止冒认：**如果每次传输时都掺入了动态盐值，让加密的结果都不同，那即使被窃取了，也不能冒用来进行另一次调用。在双方通信均可能泄漏的前提下协商出只有双方才知道的保密信息是可行的，但协商出盐值的过程将变得复杂，而且每次协商只保护一次操作，也难以阻止对其他服务的重放攻击。
- **给服务加入动态令牌：**在网关或其他流量公共位置建立校验逻辑，服务端愿意付出在集群中分发令牌信息等代价的前提下，可以做到防止重放攻击，但是依然不能抵御传输过程中被嗅探而泄漏信息的问题。
- **启用 HTTPS 可以防御链路上的恶意嗅探：**也能在通信层面解决了重放攻击的问题。但依然有因客户端被攻破产生伪造根证书、有因服务端被攻破产生的证书泄漏而被中间人冒认、有因CRL更新不及时或者OCSP Soft-fail 产生吊销证书被冒用、有因 TLS 的版本过低或密码学套件选用不当产生加密强度不足的风险。
- **为了抵御上述风险，保密强度还要进一步提升：**譬如银行会使用独立于客户端的存储证书的物理设备（俗称的 U 盾）来避免根证书被恶意程序窃取伪造；涉及到账号、金钱等操作时，会使用双重验证开辟一条独立于网络的信息通道如手机验证码来显著提高冒认的难度；甚至一些关键企业（如国家电网）或机构（如军事机构）会专门建设遍布全国各地的与公网物理隔离的专用内部网络来保障通信安全。

- **密码存储和验证**
  - **普通安全强度**
    **密码创建过程**
    - 1、用户在客户端注册，输入明文密码：123456
    - 2、客户端对用户密码进行简单哈希摘要，可选的算法有 MD2/4/5、SHA1/256等
    - 3、为了防御彩虹表攻击应加盐处理，客户端加盐只取固定的字符串即可，最多用伪动态的盐值，即服务端不需要额外通信可以得到的信息。
    - 4、假设攻击者截获了信息，得到了摘要结果和盐值，就可以枚举遍历弱密码，然后对每个密码再加盐计算，就得到一个针对固定盐值的对照彩虹表。为了应对这种暴力破解，并不提倡在盐值上做动态化，更理想的方式是引入慢哈希函数（函数执行时间是可以调节的哈希函数，通常是以控制调用次数来实现的。）来解决。
    - 5、只需防御被拖库后针对固定盐值的批量彩虹表攻击，为每一个密码即客户端传来的哈希值产生一个随机的盐值。常用密码学安全伪随机数生成器CSPRNG来生成一个长度与哈希值长度相等的随机字符串。
    - 6、将动态盐值混入客户端传来的哈希值再做一次哈希，产生出最终的密文，并和上一步随机生成的盐值一起写入到同一条数据库记录中。由于慢哈希算法占用大量处理器资源，并不推荐在服务端中采用。
  - **密码验证过程**
    - 1、客户端用户在登录页面中输入密码明文，经过与注册相同的加密过程，向服务端传输加密后的结果。（与创建过程第3或第4步生成的结果相同）。
    - 2、服务端接受到哈希值，从数据库中取出登录用户对应的密文和盐值，采用相同的哈希算法，对客户端传来的哈希值、服务端存储的盐值计算摘要结果。
    - 3、比较上一步的结果和数据库储存的哈希值是否相同
- **客户端加密**
  - **背景：**为了保证信息不被黑客窃取而做客户端加密没有太多意义，对绝大多数的信息系统来说，启用 HTTPS 可以说是唯一的实际可行的方案。但是为了保证密码不在服务端被滥用，在客户端就开始加密是很有意义的，例如避免被拖库等。
  - **客户端加密对防御泄密会没有意义：**原因是网络通信并非由发送方和接收方点对点进行的，客户端无法决定用户送出的信息能不能到达服务端，或者会经过怎样的路径到达服务端，在传输链路必定是不安全的假设前提下，无论客户端做什么防御措施，最终都会沦为“马其诺防线”。
  - 在客户端加盐传输的做法通常都得不偿失，客户端无论是否动态加盐，都不可能代替 HTTPS。真正防御性的密码加密存储确实应该在服务端中进行，但是为了防御服务端被攻破而批量泄漏密码的风险，并不是为了增加传输过程的安全。

## 传输

- **定义：**系统如何保证通过网络传输的信息无法被第三方窃听、篡改和冒充?
- **摘要、加密与签名**
  - **摘要：**也称之为数字摘要或数字指纹。信息通过哈希算法计算出来摘要值，理想的哈希算法都具备易变性和不可逆性。可以在源信息不泄漏的前提下辨别其真伪。
  - **加密：**加密是可逆的，现代密码学安全建立在特定问题的计算复杂度之上。
    - **对称加密算法：**为保证两两通信都采用独立的密钥，密钥数量与成员数量平方成正比；最关键的是如何才能将一个只能让通信双方才能知道的密钥传输给对方。若能保证传输安全，也就没加密算法什么事了。
    - **非对称加密算法：**公钥加密，私钥解密，这种就是**加密，用于向私钥所有者发送信息**，这个信息可能被他人篡改，但是无法被他人得知；私钥加密，公钥解密，这种就是**签名，用于让所有公钥所有者验证私钥所有者的身份**，并且用来防止私钥所有者发布的内容被篡改。但是不用来保证内容不被他人获得。
    - **安全保密：**如果甲想给乙发一个安全保密的数据，那么应该甲乙各自有一个私钥，甲先用乙的公钥加密这段数据，再用自己的私钥加密这段加密后的数据。最后再发给乙，这样确保了内容即不会被读取，也不能被篡改。
    - **非对称加密算法缺陷：**计算复杂度都相当高，性能比对称加密要有数量级差距，因此难以支持分组，所以主流的非对称加密算法都只能加密不超过密钥长度的数据，这决定了非对称加密不能直接用于大量数据的加密。
    - **密码学套件：**用非对称加密来安全地传递少量数据给通信的另一方，然后再以这些数据为密钥即“密钥协商”，采用对称加密来安全高效地大量加密传输数据。
  - **JWT 令牌的签名不能保证负载中的信息不可篡改、不可抵赖：**即使有了哈希摘要、对称和非对称加密，但公钥在网络传输过程中可能已经被篡改，如果获取公钥的网络请求被攻击者截获并篡改，返回了攻击者自己的公钥，那以后攻击者就可以用自己的私钥来签名，让资源服务器无条件信任它的所有行为了。
- **数字证书**
  - **现实世界达成信任的方法：**基于共同私密信息的信任和基于权威公证人的信任
  - **网络世界达成信任的方法：**由于不能假设授权服务器和资源服务器是互相认识的，因此采用基于权威公证人的信任方式，即目前标准的保证公钥可信分发的公开密钥基础设施PKI。
  - **数字证书认证中心CA：**承担公钥体系中公钥的合法性检验的责任，且一般预置系统中或可安装，避免伪装CA。CA负责签发证书 Certificate，是对特定公钥信息的一种公证载体，是权威 CA 对特定公钥未被篡改的签名背书。

- **传输安全层**
  - **背景：**如果从确定加密算法、生成密钥、公钥分发、CA 认证、核验公钥、签名、验证，每一个步骤都要由最终用户来完成的话，将是十分繁琐且极难推广的。最合理的做法就是在传输层之上、应用层之下加入专门的安全层来实现，这样对上层原本基于 HTTP 的 Web 应用来说影响甚至是无法察觉的，即构建传输安全层。
  - **发展：**SSL 协议（Secure Sockets Layer）->TLS（Transport Layer Security）
  - **TLS1.2**
    - **意义：**传输安全层是保障所有信息都是第三方无法窃听（加密传输）、无法篡改（一旦篡改通信算法会立刻发现）、无法冒充（证书验证身份）的。
    - **过程**
      - **客户端请求Client Hello：**客户端向服务器请求进行加密通信，以明文的形式，向服务端提供支持的协议版本、生成的 32 Bytes 随机数（用于产生加密的密钥）、SessionID（用于复用TLS连接）、支持的密码学套件和数据压缩算法、扩展信息。
      - **服务器回应Server Hello：**服务器接收到客户端的通信请求后，支持的协议版本和加密算法组合与客户端相匹配的话，向客户端发出回应，包括服务端确认使用的 TLS 协议版本、第二个 32 Bytes 的随机数（用于产生加密的密钥）、SessionID、列表中选定的密码学算法套件、列表中选定的数据压缩方法、扩展信息。如果协商出的加密算法组合是依赖证书认证的，服务端还要发送出自己的 X.509 证书。
      - **客户端确认Client Handshake Finished：**客户端收到服务器应答后，验证服务器的证书合法性，从证书中取出服务器的公钥，并向服务器发送客户端证书（可选，部分服务端并不是面向全公众）、第三个 32 Bytes 的随机数（以服务端传过来的公钥加密）将与前两次发送的随机数一起，根据特定算法后续内容传输时的对称加密算法所采用的私钥等。
      - **服务端确认Server Handshake Finished：**服务端向客户端回应最后的确认通知，包括编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。
    - **HTTPS：**建立在安全传输层之上的 HTTP 协议，就被称为“HTTP over SSL/TLS”。采用不同的协议版本、不同的密码学套件、证书是否有效、服务端/客户端对面对无效证书时的处理策略如何都导致了不同 HTTPS 站点的安全强度的不同。

- **验证**
  - **定义：**系统如何确保提交到每项服务中的数据是合乎规则的，不会对系统稳定性、数据一致性、正确性产生风险？
  - 在 Java 里有验证的标准做法，提倡的做法是把校验行为从分层中剥离出来，不是在哪一层做，而是在 Bean 上做。即 Java Bean Validation。

# 分布式共识算法

- ## 相关概念
  - 在软件系统里，要**保障系统的可靠性**，软件系统必须有多台机器能够拥有一致的数据副本，才有可能对外提供可靠的服务。
  - 在软件系统里，要**保障系统的可用性**，必须考虑动态的数据如何在不可靠的网络通信条件下，依然能在各个节点之间正确复制的问题。
  - **状态转移：**以同步为代表的数据复制方法，较符合人类思维的可靠性保障手段，但通常要以牺牲可用性为代价，建设分布式系统的时候，往往不能承受这样的代价。
  - **操作转移：**分布式系统里主流的数据复制方法，通过某种操作，令源状态转换为目标状态。能够使用确定的操作，促使状态间产生确定的转移结果的计算模型，在计算机科学中被称为状态机State Machine。
  - **状态机模型：**根据状态机特性，要让多台机器最终状态一致，只要确保初始状态是一致的，并且接收到的操作指令序列也是一致的即可。将任何行为理解为将操作日志正确地广播给各个分布式节点。广播指令与指令执行期间，不要求所有节点每一条指令都同时开始、同步完成，只要求在此期间的内部状态不能被外部观察到。
  - **Quorum 机制：**一旦系统中过半数的节点中完成了状态的转换，就认为数据的变化已经被正确地存储在系统当中，这样就可以容忍少数（通常是不超过半数）的节点失联，使得增加机器数量对系统整体的可用性变成是有益的。
  - **协商共识：**让系统各节点不受局部的网络分区、机器崩溃、执行性能或者其他因素影响，都能最终表现出整体一致的过程。能够让分布式系统内部暂时容忍存在不同的状态，但最终能够保证大多数节点的状态达成一致；同时，能够让分布式系统在外部看来始终表现出整体一致的结果。
- ## Paxos
  - **定义：**一种基于消息传递的协商共识算法，是当今分布式系统最重要的理论基础。
  - **分布式系统节点分类**
    - **提案节点Proposer：**提出对某个值进行设置操作的节点，设置值这个行为称为提案Proposal，值一旦设置成功就不会丢失也不可变。Paxos 是典型的基于操作转移模型来设计的算法，“设置值”应该类比成日志记录操作。
    - **决策节点Acceptor：**是应答提案的节点，决定该提案是否可被投票、是否可被接受。提案一旦得到过半数决策节点的接受，即称该提案被批准，提案被批准即意味着该值不能再被更改，也不会丢失，且最终所有节点都会接受该它。
    - **记录节点Learner：**不参与提案，也不参与决策，单纯地从提案、决策节点中学习已经达成共识的提案，譬如少数派节点从网络分区中恢复时，将会进入这种状态。
    - **原则：**所有的节点都是平等的，都可以承担以上某一种或者多种的角色，为了便于确保有明确的多数派，决策节点的数量应该被设定为奇数个，且在系统初始化时，网络中每个节点都知道整个网络所有决策节点的数量、地址等信息。
  - **分布式系统值达成一致影响因素**
    - 系统内部各个节点通信是不可靠的，不论对于系统中企图设置数据的提案节点抑或决定是否批准设置操作的决策节点，其发出、收到的信息可能延迟送达、也可能会丢失，但不去考虑消息有传递错误的情况。
    - 系统外部各个用户访问是可并发的，如果系统只会有一个用户，或者每次只对系统进行串行访问，那单纯地应用 Quorum 机制，就已经足以保证值被正确地读写。
  - **分布式系统加锁分析：**对同一个变量的并发修改必须先加锁后操作，在分布式的环境下要考虑到可能出现的通信故障，如果一个节点在释放锁之前发生崩溃失联，这将导致整个操作被无限期的等待所阻塞，因此算法必须提供一个其他节点能抢占锁的机制，以避免因通信问题而出现死锁，因此分布式环境中的锁必须是可抢占的。
  - **阶段**
    - **准备Prepare**
      - 承诺不会再接受提案 ID 小于或等于 n 的 Prepare 请求。
      - 承诺不会再接受提案 ID 小于 n 的 Accept 请求。
      - 不违背承诺的前提下，回复已经批准过的提案中 ID 最大的提案设定的值和提案 ID，如果该值从来没有被提案设定过则返回空值。如果违反此前的承诺，收到的提案 ID 并不是决策节点收到过的最大的，那允许直接对此 Prepare 请求不予理会。
    - **批准Accept**
      - 1-1、如果提案节点发现所有响应的决策节点此前都没有批准过该值（即为空），那说明它可以随意地决定要设定的值，将自己选定的值与提案 ID，构成一个二元组“(id, value)”，再次广播给全部的决策节点（称为 Accept 请求）
      - 1-2、如果提案节点发现响应的决策节点中，已经有至少一个节点的应答中包含有值了，必须无条件地从应答中找出提案 ID 最大的那个值并接受，构成一个二元组“(id, maxAcceptValue)”，再次广播给全部的决策节点（称为 Accept 请求）
      - 2、每一个决策节点收到 Accept 请求时，会在不违背作出的承诺的前提下，接收并持久化对当前提案 ID 和提案附带的值。如果违反此前做出的承诺，即收到的提案 ID 并不是决策节点收到过的最大的，那允许直接对此 Accept 请求不予理会。
      - 3、当提案节点收到了多数派决策节点的应答（称为 Accepted 应答）后，协商结束，共识决议形成，将形成的决议发送给所有记录节点进行学习。
  - **缺陷：**Basic Paxos 只能对单个值形成决议，并且决议的形成至少需要两次网络请求和应答（准备和批准阶段各一次），高并发情况下将产生较大的网络开销，极端情况下甚至可能形成活锁。
- ## Multi Paxos
  - **改进：**增加了“选主”的过程，提案节点通过定时轮询（心跳），发现没有主提案节点存在时就会在心跳超时后使用准备、批准的两轮网络交互过程，向所有其他节点广播竞选主节点的请求，希望各节点对由该节点作为主节点这件事情协商达成一致共识。如果得到了决策节点中多数派的批准便竞选成功。之后除非主节点失联之后发起重新竞选，否则从此往后，就只有主节点本身才能够提出提案。
  - **过程：**提案节点接收到客户端的操作请求，将请求转发给主节点来完成提案，而主节点提案的时无需再次经过准备过程，可理解为选主过后，相当于是处于无并发的环境当中进行的有序操作，此时要对某个值达成一致，只要进行一次批准的交互。
  - **广播的数据区别：**二元组变成三元组(id, i, value)，给主节点增加了任期编号，且保证严格单调递增，以应付主节点陷入网络分区后重新恢复，但另外一部分节点仍然有多数派且已经完成了重新选主的情况，此时以任期编号大的主节点为准。

- **分布式系统中如何对某个值达成一致问题划分**
  - **如何选主Leader Election：**即使用Basic Paxos的准备、批准阶段完成。
  - **如何把数据复制到各个节点上Entity Replication**
    - **正常情况下：**客户端向主节点发起一个操作请求，主节点将变更的值写入自己的变更日志，接着把变更的信息在下一次心跳包中广播给所有的从节点，从节点收到信息后，将操作写入自己的变更日志，然后给主节点发送确认签收的消息，主节点收到过半数的签收消息后，提交自己的变更、应答客户端并且给从节点广播可以提交的消息，从节点收到提交消息后提交自己的变更，数据在节点间的复制宣告完成。
    - **异常情况下：**网络出现了分区，部分节点失联，但只要仍能正常工作的节点的数量能够满足多数派（过半数）的要求，分布式系统就仍然可以正常工作。
  - **如何保证过程是安全的Safety**
    - **协定性Safety：**所有的坏事都不会发生
    - **终止性Liveness：**所有的好事都终将发生，但不知道是啥时候。
    - 譬如以选主问题为例，Safety 保证了选主的结果一定是有且只有唯一的一个主节点；而 Liveness 则要保证选主过程是一定可以在某个时刻能够结束的。

# Gossip 协议

- **背景：**Paxos、Raft、ZAB 等分布式算法经常会被称作是“强一致性”的分布式共识协议（尽管系统内部节点可以存在不一致的状态，但从不一致的情况并不会被外部观察到）。还有另一类“最终一致性”的分布式共识协议，这表明系统中不一致的状态有可能会在一定时间内被外部直接观察到，比如DNS系统和Gossip协议。
- **工作过程**
  - 如果有某一项信息需要在整个网络中所有节点中传播，那从信息源开始，选择一个固定的传播周期，随机选择它相连接的 k 个节点（称为 Fan-Out）来传播消息。
  - 每一个节点收到消息后，如果这个消息是它之前没有收到过的，将在下一个周期内，选择除了发送消息给它的那个节点外的其他相邻 k 个节点发送相同的消息，直到最终网络中所有节点都收到了消息。
- **优势**
  - 对网络节点的连通性和稳定性几乎没有任何要求，一开始就将网络某些节点只能与一部分节点部分连通而不是以全连通网络为前提。
  - 能够容忍网络上节点的随意地增加或者减少，随意地宕机或者重启，新增加或者重启的节点的状态最终会与其他节点同步达成一致。
  - 把网络上所有节点都视为平等而普通的一员，没有中心化节点或者主节点的概念，这些特点使得 Gossip 具有极强的鲁棒性，而且非常适合在公众互联网中应用。
- **缺陷**
  - 消息通过多个轮次散播而到达全网，必然存在全网各节点状态不一致的情况。
  - 由于随机选取发送消息的节点，尽管可以在整体上测算出传播速率，但对于个体消息来说，无法准确地预计到需要多长时间才能达成全网一致。
  - 由于随机选取发送消息的节点，不可避免的存在消息重复发送给同一节点的情况，增加了网络的传输的压力，也给消息节点带来额外的处理负载。

# 从类库到服务

- ## 引入微服务产生的问题
  - **服务发现：**对消费者来说，外部的服务由谁提供？具体在什么网络位置？
  - **服务的网关路由：**对生产者来说，内部哪些服务需要暴露？哪些应当隐藏？应当以何种形式暴露服务？以什么规则在集群中分配请求？
  - **服务的负载均衡：**对调用过程来说，如何保证每个远程服务都接收到相对平均的流量，获得尽可能高的服务质量与可靠性？
- ## 服务发现
  - **背景：**所有的远程服务调用都使用全限定名（代表了网络中某台主机的精确位置）、端口号（代表了主机上提供了 TCP/UDP 网络服务的程序）与服务标识（代表了该程序所提供的某个具体的方法入口）来确定远程服务的精确坐标。全限定名、端口号的含义对所有的远程服务来说都一致，而服务标识则与具体的应用层协议相关，如 REST 的标识是 URL 地址，RMI 的标识是 Stub 类中的方法。
  - **分类：**远程服务标识的多样性，决定了服务发现也可以有两种不同的理解。一种是以 UDDI 为代表的的服务发现、另一种是类似于 DNS的服务发现。
  - **历史发展**
    - 服务发现原本只依赖 DNS 将全限定名翻译为一至多个 IP 地址或其他类型的记录，位于 DNS 之后的负载均衡器也实质上承担了一部分服务发现的职责，完成了外部 IP 地址到各个服务内部实际 IP 的转换。
    - 随着微服务的逐渐流行，服务的非正常宕机、重启和正常的上线、下线变得越发频繁，仅靠着 DNS 服务器和负载均衡器等无法跟上服务变动的步伐了。
    - 尝试使用 ZooKeeper 这样的分布式 K/V 框架，通过软件自身来完成服务注册与发现，是微服务早期的主流选择，但毕竟 ZooKeeper 是很底层的分布式工具，用户自己还需要做相当多的工作才能满足服务发现的需求。
    - 到后面的Eureka、Nacos等，服务发现框架发展得相当成熟，不仅支持通过 DNS 或者 HTTP 请求进行符号与实际地址的转换，还支持多种服务健康检查方式，支持集中配置、K/V 存储、跨数据中心的数据交换等多种功能。
    - 云原生时代来临，基础设施的灵活性得到大幅度的增强，最初的使用基础设施来透明化地做服务发现的方式又重新被人们所重视。
  - **可用与可靠**
    - **服务发现操作**
      - **服务的注册Service Registration：**当服务启动的时通过某些形式将自己的坐标信息通知到服务注册中心，分为自注册模式（由应用程序本身来完成）和第三方注册模式（由容器编排框架或第三方注册工具来完成）。
      - **服务的维护Service Maintaining：**服务发现框架必须维护服务列表的正确性，以避免告知服务的坐标后，服务却不能使用的情况。现在的服务发现框架，往往都能支持多种协议（HTTP、TCP 等）、多种方式（长连接、心跳、探针、进程状态等）去监控服务是否健康存活，将不健康的服务自动从服务注册表中剔除。
      - **服务的发现Service Discovery：**特指消费者从服务发现框架中，把一个符号转换为服务实际坐标的过程，一般通过 HTTP API 请求或DNS Lookup 操作来完成。
    - **服务发现的重要性和处理**
      - 注册中心被所有其他服务共同依赖，是系统中最基础的服务，几乎不可能在业务层面进行容错。服务注册中心一旦崩溃，整个系统都不再可用。实际上服务注册中心都是以集群的方式进行部署的，通常使用三个到七个节点（更多了会导致日志复制的开销太高）来保证高可用。
    - **可用和可靠的矛盾**
      - 期望服务注册中心一直可用永远健康的同时，也能够在访问每一个节点中都能取到可靠一致的数据，而不是从注册中心拿到的服务地址可能已经下线，这两个需求就构成了 CAP 矛盾，不可能同时满足。
    - **Eureka选择优先保证高可用性AP**
      - **节点间采用异步复制来交换服务注册信息**，新服务注册时不等待信息同步，而是马上在该服务发现节点宣告服务可见，但不保证在其他节点上多长时间后才会可见。
      - **旧的服务发生下线或者断网时由超时机制来控制何时从服务注册表中移除**，变动信息不会实时同步给所有服务端与客户端。使得不论是服务端还是客户端，都能够持有自己的服务注册表缓存，并以 TTL机制来进行更新，哪怕服务注册中心完全崩溃，客户端在仍然可以维持最低限度的可用。
      - Eureka对节点关系相对固定，服务不会频繁上下线的系统很合适，以较小的同步代价换取了最高的可用性。且即使客户端拿到了已经发生变动的错误地址，也**能够通过 Ribbon 和 Hystrix 模块配合来兜底**，实现故障转移或者快速失败。
    - **Consul选择优先保证高可靠性CP**
      - 采用Raft 算法，**要求多数派节点写入成功后服务的注册或变动才算完成**，严格地保证了在集群外部读取到的服务发现结果必定是一致的。
      - 同时采用 Gossip 协议，**支持多数据中心之间更大规模的服务同步**。
      - Consul 优先保证高可靠性基于它没有着全家桶式的微服务组件，万一从服务发现中取到错误地址，就没有其他组件为它兜底了。
    - **CP和AP的选择**
      - **场景：**假设系统形成了 A、B 两个网络分区后，A/B区的服务只能从区域内的服务发现节点获取到 A/B 区的服务坐标，这对你的系统会有什么影响？
      - **AP：**假设 A、B 是不同的机房，由于网络交换机导致服务发现集群出现分区问题，但每个分区中的服务仍然能独立提供完整且正确服务，但网络分区在事实上避免了跨机房的服务请求，反而还带来了服务调用链路优化的效果。
      - **CP：**譬如系统中大量依赖了集中式缓存、消息总线、或者其他有状态的服务，一旦这些服务全部或者部分被分隔到某一个分区中，会对整个系统的操作的正确性产生直接影响，干脆直接停机避免生成错误数据。
  - **注册中心实现**

- **在分布式 K/V 存储框架上自己开发服务发现**，如ZooKeeper、Doozerd、Etcd。Etcd/Raft算法和Zookeeper/Multi Paxos派生算法决定了这类型服务发现都是CP的，基础的能力如服务如何注册、如何做健康检查，等等都必须自己去实现。
- **以基础设施主要指 DNS 服务器来实现服务发现**，如SkyDNS、CoreDNS。K8S 1.11 版，采用的CoreDNS，是 CP 还是 AP 取决于后端采用的存储，如基于 Etcd 实现CP，基于内存异步复制的方案实现AP。以基础设施来做服务发现，好处是对应用透明，任何语言、框架、工具都支持 HTTP、DNS ，不受技术选型约束；坏处是必须考虑如何去做客户端负载均衡、如何调用远程方法等这些问题，且必须遵循或者说受限于这些基础设施本身所采用的实现机制。
- **专门用于服务发现的框架和工具**，如Eureka、Consul 和 Nacos。它们可以被应用程序感知，所以还需要考虑所用的程序语言技术框架的集成问题

## 网关路由

- **存在原因：**微服务架构下，每个服务节点都可能由不同团队负责，有着独立互不相同的接口，如果服务集群缺少统一对外交互的代理人角色，那外部的服务消费者就必须知道所有微服务节点在集群中的精确坐标，导致消费者受到网络限制（不能确保每个节点都有外网连接）、安全限制（服务节点的安全，外部受到如浏览器同源策略的约束）、依赖限制（服务坐标这类信息随时可能变动，不应该依赖它）。
- **职责：**微服务中网关的首要职责是作为统一的出口对外提供服务，将外部访问网关地址的流量，根据适当的规则路由到内部集群中正确的服务节点之上，因此，微服务中的网关，也常被称为“服务网关”或者“API 网关”。在此基础上还可以根据需要作为流量过滤器来使用，提供某些额外的可选的功能。
- **关注点**
  - **网络协议层次**
    - **定义：**指负载均衡中介绍过的四层流量转发与七层流量代理
    - **从技术实现角度来看：**对于路由这项工作，负载均衡器与服务网关在实现上是没有什么差别的，很多服务网关本身就是基于老牌的负载均衡器来实现的。
    - **从目的角度看：**负载均衡器是为了根据均衡算法对流量进行平均地路由，服务网关是为了根据流量中的某种特征进行正确地路由。
    - **网关必须能够识别流量中的特征，即网关能够支持的网络通信协议的层次将会直接限制后端服务节点能够选择的服务通信方式**。如果服务集群只提供像 Etcd 这样直接基于 TCP 的访问的服务，那只部署四层网关便可，网关以 IP 报文中源地址、目标地址为特征进行路由；如果服务集群要提供 HTTP 服务的话，那就必须部署一个七层网关，网关根据 HTTP 报文中的 URL、Header 等信息为特征进行路由；如果服务集群还要提供更上层的 WebSocket、SOAP 等服务，那就必须要求网关同样能够支持这些上层协议，才能从中提取到特征。
  - **性能与可用性**
    - **影响因素：**网关的性能与工作模式和自身实现算法都有关系，但工作模式是最关键的因素，若采用 DSR 三角传输模式，原理上决定了性能一定会比代理模式来的强
    - **与网络I/O模型关系：**由于REST 和 JSON-RPC 等基于 HTTP 协议的服务接口在对外部提供的服务中占主流的地位，所以服务网关默认都必须支持七层路由，通常就默认无法直接进行流量转发，只能采用代理模式。在这个前提约束下，网关的性能主要取决于它们如何代理网络请求，也即它们的网络 I/O 模型。
  - **网络I/O模型**
    - **定义：**在套接字接口抽象下，网络 I/O 的出入口就是 Socket 的读和写，Socket 在操作系统接口中被抽象为数据流，网络 I/O 可以理解为对流的操作。当发生一次网络请求发生后，将会按顺序经历“等待数据从远程主机到达缓冲区”和“将数据从缓冲区拷贝到应用程序地址空间”两个阶段。
    - **异步I/O：**数据到达缓冲区后，不需要由调用进程主动进行从缓冲区复制数据的操作，而是复制完成后由操作系统向线程发送信号，所以它一定是非阻塞的。
    - **同步I/O**
      - **阻塞 I/O：**在数据没有到达应用程序地址空间时线程休眠，即线程被挂起。阻塞 I/O 是最直观的 I/O 模型，逻辑清晰，比较节省 CPU 资源，但缺点就是线程休眠所带来上下文切换，这是一种需要切换到内核态的重负载操作，不应当频繁进行。
      - **非阻塞 I/O：**非阻塞 I/O 能够避免线程休眠，线程每隔一段时间去检查数据是否到达应用程序地址空间，对于一些很快就能返回结果的请求，非阻塞 I/O 可以节省切换上下文切换的消耗，但是对于较长时间才能返回的请求，非阻塞 I/O 反而白白浪费了 CPU 资源，所以目前并不常用。
      - **多路复用 I/O：**多路复用 I/O 本质上是阻塞 I/O 的一种，但它可以在同一条阻塞线程上处理多个不同端口的监听。当某个端口的数据到达后，即马上通知它，然后继续监听其他端口，多路复用 I/O 是目前的高并发网络应用的主流。
      - **信号驱动 I/O：**信号驱动 I/O 与异步 I/O 的区别是“从缓冲区获取数据”这个步骤的处理，前者收到的通知是可以开始进行复制操作了，在复制完成之前线程处于阻塞状态，所以它仍属于同步 I/O 操作，而后者收到的通知是复制操作已经完成。
  - **网关可用性**
    - 由于网关的地址具有唯一性，就不像之前服务发现那些注册中心那样直接做个集群
    - 网关应尽可能轻量，尽管网关作为服务集群统一的出入口，可以很方便地做安全、认证、授权、限流、监控，等等的功能，但给网关附加这些能力时还是要仔细权衡，取得功能性与可用性之间的平衡，过度增加网关的职责是危险的。
    - 网关选型时，应该尽可能选择较成熟的产品实现，譬如 Nginx Ingress Controller、KONG、Zuul 这些经受过长期考验的产品，而不能一味只考虑性能选择最新的产品，性能与可用性之间的平衡也需要权衡。
    - 在需要高可用的生产环境中，应当考虑在网关之前部署负载均衡器或者等价路由器（ECMP），让那些更成熟健壮的设施（往往是硬件物理设备）去充当整个系统的入口地址，这样网关也可以进行扩展了。

## 客户端负载均衡

- **背景：**以前负载均衡器大多只部署在整个服务集群的前端，将用户的请求分流到各个服务进行处理，即集中式的负载均衡。随着微服务日渐流行，越来越多的访问请求是由集群内部的某个服务发起，由集群内部的另一个服务进行响应的，对于这类流量的负载均衡，针内部流量的特点，直接在服务集群内部消化掉，是更合理的。它与服务端负载均衡器的关键差别所在：客户端均衡器是和服务实例一一对应的，而且与服务实例并存于同一个进程之内。
- **客户端负载均衡优势**
  - 均衡器与服务之间信息交换是进程内的方法调用，不存在任何额外的网络开销。
  - 不依赖集群边缘的设施，所有内部流量都仅在服务集群的内部循环，避免了出现集群内部流量要“绕场一周”的尴尬局面。
  - 分散式的均衡器意味着天然避免了集中式的单点问题，它的带宽资源将不会像集中式均衡器那样敏感，这在以七层均衡器为主流、不能通过 IP 隧道和三角传输这样方式节省带宽的微服务环境中显得更具优势。
  - 客户端均衡器要更加灵活，能够针对每一个服务实例单独设置均衡策略等参数，访问某个服务，是不是需要具备亲和性，选择服务的策略是随

机、轮询、加权还是最小连接等等，都可以单独设置而不影响其它服务。

- **客户端负载均衡缺陷**
  - 它与服务运行于同一个进程之内，意味着它的选型受到服务所使用的编程语言的限制，要为每种语言都实现对应的能够支持复杂网络情况的均衡器是非常难的。
  - 从个体服务来看，由于是共用一个进程，均衡器的稳定性会直接影响整个服务进程的稳定性，消耗的资源也同样影响到服务的可用资源。从集群整体来看，在服务数量达成千乃至上万规模时，客户端均衡器消耗的资源总量是相当可观的。
  - 由于请求的来源可能是来自集群中任意一个服务节点，而不再是统一来自集中式均衡器，这就使得内部网络安全和信任关系变得复杂，当攻破任何一个服务时，更容易通过该服务突破集群中的其他部分。
  - 服务集群的拓扑关系是动态的，每一个客户端均衡器必须持续跟踪其他服务的健康状况，以实现服务队列维护。由于这些操作都需要通过访问服务注册中心来完成，数量庞大的客户端均衡器一直持续轮询服务注册中心，也会为它带来不小的负担。
- **代理负载均衡器**
  - **定义：**对此前的客户端负载均衡器的改进是将原本嵌入在服务进程中的均衡器提取出来，作为一个进程之外，同一 Pod 之内的特殊服务，放到边车代理中去实现。
  - 代理均衡器与服务实例不再是进程内通信，而是通过网络协议栈进行数据交换的，数据要经过操作系统的协议栈，要进行打包拆包、计算校验和、维护序列号等网络数据的收发步骤，流量比起之前的客户端均衡器确实多增加了一系列处理步骤。
  - **优势**
    - 代理均衡器不再受编程语言的限制。集中不同编程语言的使用者的力量，更容易打造出能面对复杂网络情况的、高效健壮的均衡器。且独立于服务进程的均衡器也不会由于自身的稳定性影响到服务进程的稳定。
    - 在服务拓扑感知方面代理均衡器也要更有优势。由于边车代理接受控制平面的统一管理，当服务节点拓扑关系发生变化时，控制平面就会主动向边车代理发送更新服务清单的控制指令，这避免了此前客户端均衡器必须长期主动轮询服务注册中心的浪费。
    - 在安全性、可观测性上，由于边车代理都是一致的实现，有利于在服务间建立双向 TLS 通信，也有利于对整个调用链路给出更详细的统计信息。

# 流量治理

- **前提**
  - **容错性设计：**是微服务的另一个核心原则，由于服务随时都有可能崩溃，因此快速的失败检测和自动恢复就显得至关重要。
  - **拆分服务越来越多会遇到的问题**
    - **雪崩效应：**由于某一个服务的崩溃，导致所有用到这个服务的其他服务都无法正常工作，一个点的错误经过层层传递，最终波及到调用链上与此有关的所有服务。
    - **突发流量：**服务没有崩溃，但由于处理能力有限，面临超过预期的突发请求时，大部分请求直至超时都无法完成处理。这种现象产生的后果跟交通堵塞是类似的，如果一开始没有得到及时的治理，后面就需要长时间才能使全部服务都恢复正常。
- **服务容错**
  - **容错策略**
    - **调用失败如何弥补**
      - **故障转移：**是指如果调用的服务器出现故障，系统不会立即向调用者返回失败结果，而是自动切换到其他服务副本，尝试能否返回成功调用的结果，保证整体的高可用性。故障转移应该有调用次数限制，超过次数报错依旧返回调用失败。因为重试是有执行成本的，且过度的重试反而可能让系统处于更加不利的状况。
        - **优点：**系统自动处理，调用者对失败的信息不可见
        - **缺陷：**增加调用时间，额外的资源开销
        - **应用场景：**调用幂等服务，对调用时间不敏感的场景
      - **快速失败：**尽快让服务报错，坚决避免重试，尽快抛出异常，由调用者自行处理。故障转移的前提是要求服务具备幂等性，对于非幂等的服务，重复调用就可能产生脏数据，引起的麻烦远大于服务调用直接失败。比如银行扣款错误，很难判断是扣款指令发送给银行时出现的网络异常，还是扣款后返回结果给服务时的网络异常。
        - **优点：**调用者有对失败的处理完全控制权，不依赖服务的幂等性
        - **缺陷：**调用者必须正确处理失败逻辑，如果一味只是对外抛异常，容易引起雪崩
        - **应用场景：**调用非幂等的服务，超时阈值较低的场景
      - **安全失败：**在一个调用链路中的服务通常有主路和旁路之分，有部分服务失败了也不影响核心业务的正确性。即使旁路逻辑调用实际失败了，也当作正确来返回，也可以返回一个符合要求的零值，然后自动记录一条服务调用出错的日志备查。
        - **优点：**不影响主路逻辑
        - **缺陷：**只适用于旁路调用
        - **应用场景：**调用链中的旁路服务
      - **故障恢复：**一般作为其他容错策略的补充措施，如果设置容错策略为故障恢复的话，通常默认会采用快速失败加上故障恢复的策略组合。当服务调用出错了以后，将该次调用失败的信息存入一个消息队列中，然后由系统自动开始异步重试调用。
        - 尽力促使失败的调用最终能够被正常执行，也为服务注册中心和负载均衡器及时提供服务恢复的通知信息。要求服务必须具备幂等性，由于重试是后台异步进行，一般用于对实时性要求不高的主路逻辑，也适合处理不需要返回值的旁路逻辑。为了避免内存中异步调用任务堆积，与故障转移一样，应该有最大重试次数的限制。
        - **优点：**调用失败后自动重试，也不影响主路逻辑
        - **缺陷：**重试任务可能产生堆积，重试仍然可能失败
        - **应用场景：**调用链中的旁路服务，对实时性要求不高的主路逻辑也可以使用
      - **沉默失败：**如果大量的请求需要等到超时或者长时间处理后才宣告失败，很容易由于请求堆积而消耗大量的线程、内存、网络等资源，进而影响到整个系统的稳定。此时合理的失败策略是当请求失败后，就默认服务提供者一定时间内无法再对外提供服务，不向它分配请求流量，将错误隔离开来，避免对系统其他部分产生影响。
        - **优点：**控制错误不影响全局
        - **缺陷：**出错的地方将在一段时间内不可用
        - **应用场景：**频繁超时的服务
    - **调用之前如何获得最大的成功概率**
      - **并行调用：**一开始就同时向多个服务副本发起调用，只要有其中任何一个返回成功，便宣告成功。在关键场景中使用更高的执行成本换取执行时间和成功概率。
        - **优点：**尽可能在最短时间内获得最高的成功率
        - **缺陷：**额外消耗机器资源，大部分调用可能都是无用功
        - **应用场景：**资源充足且对失败容忍度低的场景
      - **广播调用：**同时发起多个调用，广播调用要求所有的请求全部都成功，通常会被用于实现“刷新分布式缓存”这类的操作。
        - **优点：**支持同时对批量的服务提供者发起调用
        - **缺陷：**资源消耗大，失败概率高
        - **应用场景：**只适用于批量操作的场景
  - **容错设计模式**

- **断路器模式**
  - **基本思路：**通过断路器对象与远程服务一一对应接管服务调用者的远程请求，持续监控并统计服务返回的各种结果，当故障次数达到断路器的阈值时，断路器打开，后续代理的远程访问都将直接返回调用失败。通过断路器对远程服务的熔断，避免因持续的失败而消耗资源或持续超时而堆积请求，避免雪崩效应的出现。
  - **状态转变为OPEN**
    - 一段时间内请求数量达到一定阈值。这个条件的意思是如果请求本身就很少，那就用不着断路器介入。
    - 一段时间内请求的故障率到达一定阈值。这个条件的意思是如果请求本身都能正确返回，也用不着断路器介入。
  - **服务熔断与服务降级**
    - 服务熔断是一种快速失败的容错策略的实现方法。在明确反馈了故障信息给上游服务以后，上游服务必须能够主动处理调用失败的后果，而不是坐视故障扩散，这里的“处理”指的就是一种典型的服务降级逻辑，降级逻辑可以包括，但不应该仅仅限于是把异常信息抛到用户界面去，而应该尽力想办法通过其他路径解决问题。
    - 服务降级不一定是在出现错误后才被动执行的，许多场景里面降级更可能是指需要主动迫使服务进入降级逻辑的情况。如应对可预见的峰值流量，或者是系统检修等原因，关闭系统部分功能或部分旁路服务，此时有可能会主动迫使这些服务降级。
- **舱壁隔离模式**
  - **定义：**是常用的实现服务隔离的设计模式，符合容错策略中沉默失败策略
  - **调用外部服务故障**
    - **失败：**如400 Bad Request、500 Internal Server Error
    - **拒绝：**如 401 Unauthorized、403 Forbidden
    - **超时：**如 408 Request Timeout、504 Gateway Timeout
  - **超时带来全局性风险：**由于目前主流的网络访问大多是基于Thread per Request并发模型来实现的，只要请求不结束就一直占用着线程不能释放。而线程是典型的全局性资源，尤其是 Java 这类将线程映射为操作系统内核线程来实现的语言，为了不让某一个远程服务的局部失败演变成全局性的影响，就必须引入服务隔离。
  - **使用局部的线程池来控制服务的最大连接数：**为每个服务单独设立线程池，默认不预置活动线程。服务的超时故障就只会最多阻塞最大连接数的线程。
    - **优点：**当服务出问题时能够隔离影响，当服务恢复后，还可以通过清理掉局部线程池，瞬间恢复该服务的调用。
    - **缺陷：**额外增加了 CPU 的开销，每个独立的线程池都要进行排队、调度和下文切换工作
  - **更轻量的可以用来控制服务最大连接数办法：**信号量机制。如果不考虑清理线程池、客户端主动中断线程这些额外的功能，可以只为每个远程服务维护一个线程安全的计数器即可，并不需要建立局部线程池。
  - 舱壁隔离模式还可以在更高层、更宏观的场景中使用，不是按调用线程，而是按功能、按子系统、按用户类型等条件来隔离资源都是可以的，譬如，根据用户等级、用户是否 VIP、用户来访的地域等各种因素，将请求分流到独立的服务实例去。
- **重试模式**
  - **使用场景：**适合解决系统中的瞬时故障，即有可能自愈的临时性失灵，网络抖动、服务临时过载如503 Bad Gateway 错误等。实现故障转移和故障恢复容错策略。
  - **使用的前提条件**
    - **仅在主路逻辑的关键服务上进行同步的重试**，不是关键的服务，一般不把重试作为首选容错方案，尤其不该进行同步重试。
    - **仅对由瞬时故障导致的失败进行重试**。关于故障是否属于可自愈的瞬时故障可从 HTTP 的状态码上获得一些初步的结论。
    - **仅对具备幂等性的服务进行重试**。判断服务是否幂可以找到一些总体上通用的原则。如RESTful 服务中的 POST 请求是非幂等的，而 GET、PUT、HEAD、OPTIONS、TRACE 由于不会改变资源状态，这些请求应该被设计成幂等的。
    - **重试必须有明确的终止条件**，如超时终止或次数终止。
  - **风险：**由于重试模式可以在网络链路的多个环节中去实现，如客户端调用的、网关中的、负载均衡器中的自动重试等，若不注意可能导致重试次数达到指数级别。

## 流量控制

- **涉及的问题**
  - **限流的依据：**要不要控制流量，要控制哪些流量，控制力度要有多大，等等这些操作都没法在系统设计阶段静态地给出确定的结论，必须根据系统此前一段时间的运行状况，甚至未来一段时间的预测情况来动态决定。
  - **具体如何限流：**解决系统具体是如何做到允许一部分请求能够通行，而另外一部分流量实行受控制的失败降级，必须了解掌握常用的服务限流算法和设计模式。
  - **超额流量如何处理：**有不同的处理策略，可以直接返回失败如 429 Too Many Requests，或者被迫使它们进入降级逻辑（否决式限流）。也可能让请求排队等待，暂时阻塞一段时间后继续处理（阻塞式限流）。
- **流量统计指标**
  - **每秒事务数Transactions per Second：**TPS 是衡量信息系统吞吐量的最终标准。事务可以理解为一个逻辑上具备原子性的业务操作。
  - **每秒请求数Hits per Second：**HPS 是指每秒从客户端发向服务端的请求数。如果只要一个请求就能完成一笔业务，那 HPS 与 TPS 是等价的，但在一些场景（尤其常见于网页中）里，一笔业务可能需要多次请求才能完成。
  - **每秒查询数Queries per Second：**QPS 是指一台服务器能够响应的查询次数。如果只有一台服务器来应答请求，那 QPS 和 HPS 是等价的，但在分布式系统中，一个请求的响应往往要由后台多个服务节点共同协作来完成。
  - 整体上最希望能够基于 TPS 来限流，但真实业务操作的耗时受限于用户交互带来的不确定性。目前主流系统大多倾向使用 HPS 作为首选的限流指标，它是相对容易观察统计的，而且能够在一定程度上反应系统当前以及接下来一段时间的压力。
- **限流设计模式**
  - **流量计数器模式**
    - **思路：**设置计算器，根据当前时刻的流量计数结果是否超过阈值来决定是否限流。
    - **每一秒的统计流量都没有超过阈值不能说明系统没有遇到超过阈值的流量压力**。如果系统连续两秒都收到 60 TPS 的访问请求，但分别是前 1 秒里面的后 0.5 秒，以及后 1 秒中的前面 0.5 秒所发生的。虽然每个周期的流量都不超过 80 TPS 请求的阈值，但是系统确实曾

经在 1 秒内实在在发生了超过阈值的 120 TPS 请求。

- **连续若干秒统计流量都超过了阈值也不能说明流量压力就超过了系统承受能力**。如果 10 秒中，前 3 秒 TPS为100，而后 7 秒为30，此时系统仍然能够处理完这些请求。超时时间是 10 秒，而最慢的请求也能在 8 秒左右处理完毕。此时限制阈值，反而会误杀一部分请求，造成部分请求出现原本不必要的失败。
- **缺陷：**只是针对时间点进行离散的统计。

- **滑动时间窗模式**
  - **思路：**与流量计数器模式区别，实现平滑的基于时间片段统计。在向前流淌的时间轴上，漂浮着固定大小的窗口。任何时刻静态地通过窗口内观察到的信息，都等价于一段长度与窗口大小相等、动态流动中时间片段的信息。
  - **工作过程**
    - 1、将数组最后一位的元素丢弃掉，并把所有元素都后移一位，然后在数组第一个插入一个新的空元素。这个步骤即为“滑动窗口”。
    - 2、将计数器中所有统计信息写入到第一位的空元素中。
    - 3、对数组中所有元素进行统计，并复位清空计数器数据供下一个统计周期使用。
  - **优点：**滑动时间窗口模式的限流完全解决了流量计数器的缺陷，可以保证任意时间片段内，只需经过简单的调用计数比较，就能控制住请求次数一定不会超过限流的阈值，在单机限流或者分布式服务单点网关中的限流中很常用。
  - **缺陷：**通常只适用于否决式限流，超过阈值的流量就必须强制失败或降级，很难进行阻塞等待处理，也就很难在细粒度上对流量曲线进行整形，无法削峰填谷。
- **流量整型-漏桶模式**
  - **思路：**其实就是一个以请求对象作为元素的先入先出队列，队列长度就相当于漏桶的大小，当队列已满时便拒绝新的请求进入。
  - **缺陷：**困难在于如何确定漏桶的两个参数：桶的大小和水的流出速率。如果桶设置得太大，那服务依然可能遭遇到流量过大的冲击，不能完全发挥限流的作用；如果设置得太小，那很可能就会误杀掉一部分正常的请求。
- **流量整型-令牌桶模式**
  - **思路：**与漏桶模式刚好相反，前者是从水池里往系统出水，后者是系统往排队机中放入令牌。
  - **工作过程**
    - 1、让系统以一个由限流目标决定的速率向桶中注入令牌，譬如要控制系统的访问不超过100 次/秒，速率即设定为 100 个令牌/秒，每个令牌注入间隔为10 毫秒。
    - 2、桶中最多可以存放 N 个令牌，N 的具体数量是由超时时间和服务处理能力共同决定的。如果桶已满，第 N+1 个进入的令牌会被丢弃掉。
    - 3、请求到时先从桶中取走 1 个令牌，如果桶已空就进入降级逻辑。
  - **实现：**不需要真的用一个专用线程或者定时器来每隔一段时间放入令牌，只要在令牌中增加一个时间戳记录，每次获取令牌前，比较一下时间戳与当前时间，就可以轻易计算出这段时间需要放多少令牌进去，然后一次性放入即可。

- **分布式限流**
  - **背景：**前面讨论过的那些限流算法，直接使用在单体架构的集群上是完全可行的，但到了微服务架构下，它们就最多只能应用于集群最入口处的网关上，对整个服务集群进行流量控制，而无法细粒度地管理流量在内部微服务节点中的流转情况。
  - **与单机限流差别：**核心差别在于如何管理限流的统计指标，单机限流指标存储在服务内存当中，而分布式限流无论是将限流功能封装为专门的远程服务，抑或是在系统采用的分布式框架中有专门的限流支持，都需要将原本在每个服务节点自己内存当中的统计数据给开放出来，让全局的限流服务可以访问到才行。
  - **一种分布式限流方法是将所有服务的统计结果都存入集中式缓存如 Redis中**，以实现在集群内的共享，并通过分布式锁、信号量等机制解决读写访问时的并发控制。此时单机限流模式也可以应用于分布式环境中。代价是每次服务调用都必须要额外增加一次网络开销，流量压力大时反倒会显著降低系统的处理能力。
  - **缓解集中式缓存性能损耗：**在令牌桶限流模式基础上把令牌看作数值形式的货币额度。当请求进入集群时，首先在 API 网关处领取到一定数额的货币，访问每个服务时都要求消耗一定量的货币。
    - **限流指标：**把剩余额度作为内部限流的指标，只要一旦剩余额度≤0 时，就不再允许访问其他服务了。此时必须先发生一次网络请求，重新向令牌桶申请一次额度，成功后才能继续访问，不成功则进入降级逻辑。
    - **缺陷：**基于额度的限流方案对限流的精确度有一定的影响，可能存在业务操作已经进行了一部分服务调用，却无法从令牌桶中再获取到新额度导致业务操作失败，白白浪费了部分已经完成了的服务资源。但总体来说，它仍是一种并发性能和限流效果上都相对折衷可行的分布式限流方案。

# 可靠通讯

- **零信任网络**
  - **边界安全**
    - **基于边界的安全模型：**根据某类与宿主机相关的特征，如机器所处的位置、IP 地址、子网等，把网络划分为不同的区域，对应于不同风险级别和允许访问的网络资源权限，将安全防护措施集中部署在各个区域的边界之上，重点关注跨区域的网络流量。相关VPN、DMZ、防火墙、内网、外网等概念。
    - **思路：**在可用性和安全性之间权衡取舍，着重对经过网络区域边界的流量进行检查，对可信任区域内部机器之间的流量则给予直接信任或较为宽松的处理策略，减小了安全设施对整个应用系统复杂度的影响，以及网络传输性能的额外损耗。
    - **缺陷：**边界上的防御措施即使自身能做到永远滴水不漏牢不可破，一旦可信的网络区域中的某台服务器被攻陷，那边界安全措施就成了马其诺防线，攻击者很快就能以一台机器为跳板，侵入到整个内网。
  - **零信任安全模型**
    - **中心思想：**不应当以某种固有特征来自动信任任何流量，除非明确得到了能代表请求来源的身份凭证，否则一律不会有默认的信任关系。
    - **特征**
      - **零信任网络不等同于放弃在边界上的保护设施**
      - **身份只来源于服务：**服务所部署的 IP 地址、服务实例的数量随时都可能发生变化，因此，身份只能来源于服务本身所能够出示的身份凭证（通常是数字证书），而不再是服务所在的 IP 地址、主机名或者其它特征。
      - **服务之间也没有固有的信任关系：**只有已知明确授权的调用者才能访问服务，阻止攻击者通过某个服务节点中的代码漏洞来越权调用到其他服务。如果某个服务节点被成功入侵，这一原则可阻止攻击者执行扩大其入侵范围。
      - **集中、共享的安全策略实施点：**微服务分散治理，但涉及安全的非功能性需求最好除外。要写出高度安全的代码极为不易，为此付出的精力甚至可能远高于业务逻辑本身；让服务各自处理安全问题很容易会出现实现不一致或者出现漏洞时要反复修改多处地方，还有一些安全问题如果不立足于全局是很难彻底解决的。
      - **受信的机器运行来源已知的代码：**限制了服务只能使用认证过的代码和配置，并且只能运行在认证过的环境中。零信任安全针对软件供应链（编写代码、自动化测试、自动集成、漏洞扫描、发布上线）的每一步都加入了安全控制策略。
      - **自动化、标准化的变更管理：**独立于应用的安全基础设施，可以让运维人员轻松地了解基础设施变更对安全性的影响，在不影响生产环境的情况下发布安全补丁。
      - **强隔离性的工作负载**
    - **最终目的：**实现整个基础设施之上的自动化安全控制，服务所需的安全能力可以与服务自身一起，以相同方式自动进行伸缩扩展。对于程序来说，做到安全是日常，风险是例外；对于人类来说，做到袖手旁观是日常，主动干预是例外。
  - **Google探索**
    - 为了在网络边界上保护内部服务免受 DDoS 攻击，设计了名为 Google Front End（名字意为“最终用户访问请求的终点”）的边缘代理，负责保证此后所有流量都在 TLS 之上传输，并自动将流量路由到适合的可用区域之中。
    - 为了强制身份只来源于服务，设计了名为 Application Layer Transport Security（应用层传输安全）的服务认证机制，这是一个用于双向认证和传输加密的系统，自动将服务与它的身份标识符绑定。
    - 为了确保服务间不再有默认的信任关系，设计了 Service Access Policy（服务访问策略）来管理一个服务向另一个服务发起请求时所需提供的认证、鉴权和审计策略，并支持全局视角的访问控制与分析。
    - 为了实现仅以受信的机器运行来源已知的代码，设计了名为 Binary Authorization的部署时检查机制，确保在软件供应链的每一个阶段，都符合内部安全检查策略，并对此进行授权与鉴权。同时设计了名为 Host Integrity的机器安全启动程序，在创建宿主机时自动验证包括 BIOS、BMC、Bootloader 和系统内核数字签名。
    - 为了工作负载能够具有强隔离性，设计了名为gVisor的轻量级虚拟化方案，解决容器共享操作系统内核而导致隔离性不足的安全缺陷，做法都是为每个容器提供了一个独立的虚拟 Linux 内核。
- **服务安全**
  - **背景：**介绍在前微服务时代和云原生时代分别是如何实现安全传输、认证和授权的，通过这两者的对比，探讨在微服务架构下，应如何将业界的安全技术标准引入并实际落地，实现零信任网络下安全的服务访问。
  - **建立信任**
    - **前提：**网络世界达成信任只能采用基于权威公证人的信任，即公开密钥基础设施PKI，是构建传输安全层TLS的必要基础。**启用 TLS 对传输通道本身进行加密是让发送者发出的内容只有接受者可以解密是唯一具备可行性的方案**。
    - 建立 TLS 传输只要在部署服务器时预置好CA 根证书，以后用该 CA 为部署的服务签发 TLS 证书。这属于必须集中在基础设施中自动进行的安全策略实施点，面对数量庞大且自动扩缩的服务节点，依赖运维人员手工去部署轮换根证书是不可能的。
    - 微服务中...
      - **单向 TLS 认证：**只需要服务端提供证书，客户端通过服务端证书验证服务器的身份，但服务器并不验证客户端的身份。单向 TLS 用于公开的服务，即任何客户端都被允许连接到服务进行访问，它保护的重点是客户端免遭冒牌服务器的欺骗。
      - **双向 TLS 认证：**客户端、服务端双方都要提供证书，双方各自通过对方提供的证书来验证对方的身份。双向 TLS 用于私密的服务，它除了保护客户端不连接到冒牌服务器外，也保护服务端不遭到非法用户的越权访问。
  - **认证**
    - **服务认证**
      - **云原生：...**
        - 如果每一个服务提供者、调用者均受 Istio 管理，那 mTLS 就是最理想的认证方案，只需要简单的配置，即可对某个K8S名称空间范围内流量均启用 mTLS。
        - 仍存在部分不受 Istio 管理的服务端或者客户端，也可以将 mTLS 传输声明为宽容模式，即受 Istio 管理的服务会允许同时接受明文

和 mTLS 两种流量，明文流量仅用于与那些不受 Istio 管理的节点进行交互，自行想办法解决明文流量的认证问题；而对于服务网格内部的流量，就可以使用 mTLS 认证。一旦所有服务都完成Istio迁移，便可将整个系统设置为严格 TLS 模式。

- **前微服务时代：...**
  - 客户端与认证服务器约定好一组只有自己知道的密钥（Client Secret），由运维人员在线下自行完成，通过参数传给服务，而不是由开发人员在源码或配置文件中直接设定。**密钥就是客户端的身份证明，客户端调用服务时，会先使用该密钥向认证服务器申请到 JWT 令牌，然后通过令牌证明自己的身份，最后访问服务。**
  - **每一个对外提供服务的服务端，都扮演着 OAuth 2 中的资源服务器的角色，它们均声明为要求提供客户端模式的凭证。**
  - Spring Security 提供的过滤器会自动拦截请求、驱动认证、授权检查的执行，申请和验证 JWT 令牌等操作在开发期对程序员，运行期对用户都能做到相对透明。
  - **缺陷：**仍然是一种应用层面的不加密传输的解决方案，对传输过程中内容被监听、篡改、以及被攻击者在传输途中拿到 JWT 令牌后再去冒认调用者身份调用其他服务都是无法防御的。**不适用于零信任安全模型，只能在默认内网节点间具备信任关系的边界安全模型上才能良好工作。**

- **请求认证**
  - **云原生：**Istio仍然能做到单纯依靠基础设施解决问题，整个认证过程无需应用程序参与。当来自最终用户的请求进入服务网格时，Istio会自动根据配置中的JSON Web Key Set来验证令牌的合法性，如果令牌没有被篡改过且在有效期内，就信任 Payload 中的用户身份，并从令牌的 Iss 字段中获得 Principal。
    - **JWKS ：** 是一组 JWK（存储密钥的纯文本格式） 的集合，能通过 JWT 令牌 Header 中的 KID（Key ID）来自动匹配出应该使用哪个 JWK 来验证签名
  - **前微服务时代：**依然是采用 JWT 令牌作为用户身份凭证的载体，认证过程依然在 Spring Security 的过滤器里中自动完成。Spring Security 已经做好了认证所需的绝大部分的工作，真正要开发者去编写的代码是令牌的具体实现。

- **授权**
  - **云原生：**Istio可以通过配置，限制名空间的内部流量只能流经哪些资源或操作。虽然不可能跟由代码实现的授权相比，但多数场景已经够用了。在便捷性、安全性、无侵入、统一管理等，Istio 这种在基础设施上实现授权方案显然就要更具优势。
  - **前微服务时代：**Spring Cloud中授权控制还是使用 Spring Security、通过应用程序代码来实现的。Spring Security 授权方法有两种，一种是使用Configurer来进行集中配置，另一种写法是通过 Spring 的全局方法级安全以及@RolesAllowed注解来做授权控制。后者对代码的侵入性更强，要以注解的形式分散写到每个服务甚至是每个方法中，但好处是能以更方便的形式做出更加精细的控制效果。

# 可观测性

- **背景**
  - **相关元素特征**
    - **日志：**职责是记录离散事件，通过这些记录事后分析出程序的行为。输出日志较为容易，但收集和分析日志却可能会很复杂。
    - **追踪：**微服务时代，追踪就不只局限于调用栈了，一个外部请求完整的调用轨迹将跨越多个服务，同时包括服务间的网络传输信息与各个服务内部的调用堆栈信息。分布式追踪常被称为全链路追踪。追踪的主要目的是排查故障，如分析调用链的哪一部分、哪个方法出现错误或阻塞，输入输出是否符合预期，等等。
    - **度量：**指对系统中某一类信息的统计聚合。度量的主要目的是监控和预警，如某些度量指标达到风险阈值时触发事件，以便自动处理或者提醒管理员介入。
  - **相关产品**
    - 日志收集和分析大多被统一到 Elastic Stack（ELK）技术栈上，其中的 Logstash 有可能被 Fluentd 取代。
    - 度量方面，Prometheus即将成为云原生时代度量监控的事实标准
    - 各个服务之间是使用 HTTP 或 gRPC通信会直接影响追踪的实现，各个服务是使用哪种语言来编写，也会直接影响到进程内调用栈的追踪方式。这决定了追踪工具本身有较强的侵入性，通常是以插件式的探针来实现；也决定了追踪领域很难出现一家独大的情况，通常要有多种产品来针对不同的语言和网络。
- **事件日志**
  - **背景：**复杂的分布式系统很难只依靠 tail、grep、awk 来从日志中挖掘信息，往往还要有专门的全局查询和可视化功能。此时，从打印日志到分析查询之间，还隔着收集、缓冲、聚合、加工、索引、存储等步骤。
  - **输出日志原则**
    - **避免打印敏感信息**
    - **避免引用慢操作：**日志中打印的信息应该是上下文中可以直接取到的，如果需要专门调用远程服务或者从数据库获取、通过大量计算才能取到，需要考虑必要性。
    - **避免打印追踪诊断信息：**日志中不要打印方法输入参数、输出结果、方法执行时长之类的调试信息。日志的职责是记录事件，追踪诊断应由追踪系统去处理。
    - **应该包含处理请求时的 TraceID：**分布式追踪通过它把请求在各个服务中的执行过程串联起来，即使是单体系统，在日志对快速定位错误时仍然有价值。
    - **应该包含系统运行过程中的关键事件：**日志的职责就是记录事件，进行了哪些操作、发生了与预期不符的情况、运行期间出现未能处理的异常或警告、定期自动执行的任务等。但应判断清楚事件的重要程度，选定相匹配的日志的级别。
    - **应该包含启动时输出配置信息：**对于系统启动时或者检测到配置中心变化时更新的配置，应将非敏感的配置信息输出到日志中，初始化配置的逻辑一般只会执行一次，不便于诊断时复现，所以应该输出到日志中。
  - **收集和缓冲**
    - **日志收集发展历史**
      - 最初，ELK 中日志收集与加工聚合的职责都由 Logstash 来承担的，Logstash 除了部署在各个节点中作为收集的客户端以外，它还同时设有独立部署的节点，扮演归集转换日志的服务端角色。但是 Logstash 与它的插件是基于 JRuby 编写的，要跑在单独的 Java 虚拟机进程上，而且 Logstash 的默认的堆大小就到了 1GB。
      - 后来，Elastic.co 公司将所有需要在服务节点中处理的工作整理成以Libbeat为核心的Beats 框架，并使用 Golang 重写了一个功能较少，却更轻量高效的日志收集器即Filebeat。框架还包括度量和追踪工具，ELK可在一定程度上代替度量和追踪。
    - **缓冲必要性：**日志收集器不仅要保证能覆盖全部数据来源，虽然不追求绝对的完整精确，但追求在代价可承受的范围内保证尽可能地保证较高的数据质量。
    - **缓冲实现：**一种最常用的缓解压力的做法是将日志接收者从 Logstash 和 Elasticsearch 转移至抗压能力更强的队列缓存，如在 Logstash 之前架设一个 Kafka 或者 Redis 作为缓冲层，面对突发流量，Logstash 或 Elasticsearch 处理能力出现瓶颈时自动削峰填谷，甚至当它们短时间停顿，也不会丢失日志数据。
  - **存储和查询**
    - **背景：**收集、缓冲、加工、聚合后的日志数据，可以放入ES 中索引存储。
    - **ElasticSearch优势**
      **（不可取代的原因）**
      - **从数据特征的角度看：**日志是基于时间的数据流，虽然增长速度很快，但已写入的数据几乎没有再发生变动的可能。日志的数据特征决定了所有用于日志分析的 Elasticsearch 都会使用时间范围作为索引，由于能准确地预知未来的日期，因此索引都可以预先创建，免去了动态创建的寻找节点、创建分片、在集群中广播变动信息等开销。又由于所有新的日志都是“今天”的日志，只要建立logs_current这样的索引别名来指向当前索引，就能避免代码因日期而变动。
      - **从数据价值的角度看：**日志基本上只会以最近的数据为检索目标，随着时间推移，早期的数据将逐渐失去价值。这点决定了可以很容易区分出冷数据和热数据，进而对不同数据采用不一样的硬件策略。
      - **从数据使用的角度看：**分析日志很依赖全文检索和即席查询，对实时性的要求是处于实时与离线两者之间，即不强求日志产生后立刻能查到，但也不能接受日志产生之后按小时或天的频率来更新，而检索能力和近实时性，也正好都是ES的强项。
    - Elasticsearch 只提供了 API 层面的查询能力，它通常搭配Kibana 一起使用，可以将 Kibana 视为 Elastic Stack 的 GUI 部分。Kibana能探索数据并可视化，即把存储在ES中的数据被检索、聚合、统计后，定制形成各种图形、表格、指标、统计，以此观察系统的运行状态，找出日志事件中潜藏规律和隐患。
  - **加工和聚合**
    - **必要性：**日志是非结构化数据，一行日志中通常会包含多项信息，如果不做处理，那在 Elasticsearch 就只能以全文检索的原始方式去使用日志，既不利于统计对比，也不利于条件过滤。

- **加工：**Logstash将日志行中的非结构化数据，通过 Grok 表达式语法转换为结构化数据，还可能会根据需要调用其他插件来完成时间处理、查询归类等额外处理工作，然后以 JSON 格式（普遍）输出到 Elasticsearch 中。因此，Elasticsearch 便可针对不同的数据项来建立索引，进行条件查询、统计、聚合等操作。
- **聚合：**如果想从离散的日志中获得统计信息，再生成可视化统计图表，一种解决方案是通过 Elasticsearch 本身的处理能力做实时的聚合统计，便捷但消耗ES服务器的运算资源。另一种解决方案是在收集日志后自动生成某些常用的、固定的聚合指标，这种聚合就会在 Logstash 中通过聚合插件来完成。这两种聚合方式都有不少实际应用，前者一般用于应对即席查询，后者用于应对固定查询。

## 链路追踪

- **定义：**广义上讲，一个完整的分布式追踪系统应该由**数据收集、数据存储和数据展示**三个相对独立的子系统构成，而狭义上讲的追踪则特指链路追踪数据的收集部分。广义的追踪系统常被称为“APM 系统”。
- **追踪与跨度**
  - **追踪：**从客户端发起请求抵达系统的边界开始，记录请求流经的每一个服务，直到到向客户端返回响应为止，这整个过程就称为一次追踪Trace。
  - **跨度：**为了能够记录具体调用了哪些服务，以及调用的顺序、开始时点、执行时长等信息，每次调用服务前都要先埋入一个调用记录，这个记录称为一个跨度Span。Span 的数据结构应该足够简单，以便于能放在日志或者网络协议的报文头里；也至少含有时间戳、起止时间、Trace ID、当前 Span 的 ID、父 Span 的 ID 等。
  - **从目标来看：**链路追踪的目的是为排查故障和分析性能提供数据支持，系统对外提供服务的过程中同时持续地生成 Trace，按次序整理好 Trace 中每一个 Span 所记录的调用关系，便能绘制出一幅系统的服务调用拓扑图。根据拓扑图中 Span 记录的时间信息和响应结果就可以定位到缓慢或者出错的服务；将 Trace 与历史记录进行对比统计，就可以从系统整体层面分析服务性能，定位性能优化的目标。
  - **从实现来看：...**
    - **功能上的挑战：**来源于服务的异构性，各个服务可能采用不同程序语言，服务间交互可能采用不同的网络协议，每兼容一种场景，都会增加功能实现方面的工作量
    - **非功能性挑战**
      - **低性能损耗：**分布式追踪不能对服务本身产生明显的性能负担。
      - **对应用透明：**追踪系统通常是运维期才事后加入的系统，应该尽量以非侵入或者少侵入的方式来实现追踪，对开发人员做到透明化。
      - **随应用扩缩：**现代的分布式服务集群都有根据流量压力自动扩缩的能力，这要求当业务系统扩缩时，追踪系统也能自动跟随，不需要运维人员人工参与。
      - **持续的监控：**要求追踪系统必须能够 7x24 小时工作，否则就难以定位到系统偶尔抖动的行为。
- **数据收集**
  - **基于日志的追踪：...**
    - **优点：**日志追踪对网络消息完全没有侵入性，对应用程序只有很少量的侵入性，对性能影响也非常低。
    - **缺陷：**直接依赖于日志归集过程，日志本身不追求绝对的连续与一致，这也使得基于日志的追踪往往不如其他两种追踪实现来的精准；业务服务调用与日志归集不是同时完成的，也通常不由同一个进程完成，有可能发生业务调用已经顺利结束了，但日志归集不及时或精度丢失，导致日志出现延迟或缺失记录，产生追踪失真。
    - **代表产品：**Spring Cloud Sleuth
  - **基于服务的追踪：...**
    - **优点：**追踪的精确性与稳定性都有所保证，不必再依靠日志归集来传输追踪数据。也可以通过这种方式手机追踪诊断信息。
    - **缺陷：**比基于日志的追踪消耗更多的资源，也有更强的侵入性
    - **代表产品：**Pinpoint、SkyWalking
  - **基于边车代理的追踪：**...
    - **优点：**与程序语言无关，只要通过网络HTTP或gRPC来访问服务就可以被追踪到；有自己独立的数据通道，追踪数据通过控制平面进行上报，避免了追踪对程序通信或者日志归集的依赖和干扰，保证了最佳的精确性。
    - **缺陷：**服务网格现在还不够普及；边车代理本身的对应用透明的工作原理决定了它只能实现服务调用层面的追踪，像本地方法调用级别的追踪诊断是做不到的。
    - **代表产品：**Envoy

## 聚合度量

- **定义：**度量Metrics的目的是揭示系统的总体运行状态，度量总体上可分为**客户端的指标收集**、**服务端的存储查询**以及**终端的监控预警**三个相对独立的过程。
- **指标收集**
  - **如何定义指标：...**
    - **计数度量器Counter：**对有相同量纲、可加减数值的合计量，如服务调用次数、网站访问人数。
    - **瞬态度量器Gauge：**表示某个指标在某个时点的数值，如虚拟机堆内存的使用量、网站在线人数。
    - **吞吐率度量器Meter：**用于统计单位时间的吞吐量，即单位时间内某个事件的发生次数，如每秒发生了多少笔事务交易。
    - **直方图度量器Histogram：**两个坐标分别是统计样本和该样本对应的某个属性的度量，以长条图的形式表示具体数值。如地区历年GDP变化。
    - **采样点分位图度量器Quantile Summary：**统计学中通过比较各分位数的分布情况的工具，用于验证实际值与理论值的差距，评估理论值与实际值之间的拟合度。
  - **如何将这些指标告诉服务端：**拉取式采集Pull指度量系统主动从目标系统中拉取指标，推送式采集Push由目标系统主动向度量系统推送指标。
  - **指标应该以怎样的网络访问协议、取数接口、数据结构来获取：**出现过如网络管理中的SNMP、Windows 硬件的WMI、以及此前提到的Java 的JMX，但收集器支持哪种协议没有强求。
  - **Prometheus**
    - **背景：**一般来说，度量系统只会支持其中一种指标采集方式，因为度量系统的网络连接数量，以及对应的线程或者协程数可能非常庞大，

如何采集指标将直接影响到整个度量系统的架构设计。而Prometheus 基于 Pull 架构的同时还能够有限度地兼容 Push 式采集，是因为它有 Push Gateway 的存在。

- **Push Gateway**
  - **定义：**一个位于 Prometheus Server 外部的相对独立的中介模块，将外部推送来的指标放到 Push Gateway 中暂存，然后再等候 Prometheus Server 从 Push Gateway 中去拉取。
  - **目的：**解决 Pull 的一些固有缺陷，譬如目标系统位于内网，外网的 Prometheus 是无法主动连接目标系统的，只能由目标系统主动推送数据；又譬如某些小型短生命周期服务，可能还等不及 Prometheus 来拉取，服务就已经结束运行了，因此也只能由服务自己 Push 来保证度量的及时和准确。
- **Exporter**
  - **背景：**Prometheus只允许通过 HTTP 访问度量端点这一种访问方式。如果目标提供了 HTTP 的度量端点，否则就需要一个专门的 Exporter 来充当媒介。
  - **定义：**是目标应用的代表，既可以独立运行，也可以与应用运行在同一个进程中。Exporter 以 HTTP 协议返回符合 Prometheus 格式要求的文本数据给服务器。

- **存储查询**
  - **定义：**指标从目标系统采集过来之后，应存储在度量系统中，以便被后续的分析界面、监控预警所使用。
  - **指标特征：**每一个度量指标由时间戳、名称、值和一组标签构成，除了时间之外，指标不与任何其他因素相关。指标的数据总量固然是不小的，但它没有嵌套、没有关联、没有主外键，不必关心范式和事务。
  - **时序数据库：**用于存储跟随时间而变化的数据，并且以时间（时间点或者时间区间）来建立索引的数据库。
  - **根据特征做出改进策略：**写操作，时序数据通常只是追加，很少删改或者根本不允许删改。针对数据热点只集中在近期数据、多写少读、几乎不删改、数据只顺序追加这些特点，时序数据库被允许做出很激进的存储、访问和保留策略。
    - 以日志结构的合并树LSM-Tree代替传统关系型数据库中的B+Tree作为存储结构，LSM 适合的应用场景就是写多读少，且几乎不删改的数据。
    - 设置激进的数据保留策略，譬如根据过期时间（TTL）自动删除相关数据以节省存储空间，同时提高查询性能。对于普通数据库来说，数据会存储一段时间后就会被自动删除这种事情是不可想象的。
    - 对数据进行再采样（Resampling）以节省空间，譬如最近几天的数据可能需要精确到秒，而查询一个月前的数据时，只需要精确到天，查询一年前的数据时，只要精确到周就够了，这样将数据重新采样汇总就可以极大节省存储空间。
    - 时序数据库中甚至还有一种并不罕见却更加极端的形式，叫作轮替型数据库，以环形缓冲的思路实现，只能存储固定数量的最新数据，超期或超过容量的数据就会被轮替覆盖，因此也有着固定的数据库容量，却能接受无限量的数据输入。
- **监控预警**
  - Prometheus 应算是处于狭义和广义的度量系统之间，在生产环境下，大多是 Prometheus 配合 Grafana 来进行展示的，这是 Prometheus 官方推荐的组合方案，但该组合也并非唯一选择，如果要搭配 Kibana 甚至 SkyWalking。
  - **监控目的：**良好的可视化能力对于提升度量系统的产品力十分重要，长期趋势分析、对照分析、故障分析等分析工作，既需要度量指标的持续收集、统计，还需要对数据进行可视化，才能让人更容易地从数据中挖掘规律。
  - **预警：**Prometheus 提供了专门用于预警的 Alert Manager，将 Alert Manager 与 Prometheus 关联后，可以设置某个指标在多长时间内达到何种条件就会触发预警状态，触发预警后，根据路由中配置的接收器，譬如邮件接收器、Slack 接收器、微信接收器、或者更通用的 WebHook接收器等来自动通知用户。