Distribution

2021年9月1日　　9:16

https://github.com/donnemartin/system-design-primer/blob/master/README-zh-Hans.md

CAP
Consistency, Availability, Partition Tolerance

一个分布式计算系统中，只能同时满足下面2点：
　　一致性　每次访问都能获得最新数据，但是可能收到错误响应
　　可用性　每次访问都能收到非错响应，但是不能保证获取到最新数据
　　分区容错性　在任意分区网络故障的情况下系统仍能继续运行

网络并不可靠，所以应该支持分区容错性。所以需要在 一致性和可用性 之间做出取舍。

CP 一致性和分区容错性
等待分区节点的响应可能导致超时错误。如果你的业务需要 原子读写，CP是不错的选择

AP 可用性和分区容错性
响应节点上的可用数据 可能不是最新的。当分区解析完后，写入操作可能需要一点时间来传播。
如果业务允许最终一致性，或者有外部故障时要求系统继续运行，AP是不错的选择。

一致性模式
一份数据有多份副本，我们需要决定 怎么同步它们，以便让客户端有一致的显示数据。

　　弱一致性
在数据写入后，访问可以看到，也可能看不到数据。尽力优化，使得能访问最新数据。
这种方式可以在memcached等系统中看到，弱一致性在VoIP，视频聊天，实时多人游戏等真实用例中表现不错。比如，在通话中丢失信号几秒钟，当重新连接时，你是听不到这几秒种的话的。

　　最终一致性
写入后，访问最终能看到写入的数据(几毫秒内)。数据被异步复制。
DNS和email 等系统使用的是这种方式。
最终一致性在高可用性系统中效果不错。

　　强一致性
在写入后，访问立刻可见。数据被同步复制。
文件系统和关系型数据库(RDBMS)中使用的是此种方式。
强一致性在需要记录的系统中运作良好。

可用性模式
有2种支持高可用性的模式：故障切换(fail-over)和复制(replication)。

故障切换
工作到备用切换
流程是，工作服务器发送周期信号给待机中的备用服务器。如果信号中断，备用服务器切换成工作服务器的IP并恢复服务。
宕机时间取决于备用服务器处于热待机状态还是需要从冷待机状态进行启动。
只有工作服务器处理流量。
工作到备用的故障切换也被称为主从切换。

双工作切换
在双工作切换中，双方都在管控流量，在它们之间分散负载。
如果是外网服务器，DNS将需要对双方都了解。如果是内网服务器，应用程序逻辑将需要对双方都了解。
也被称为主主切换。

故障切换的缺陷
故障切换需要添加额外的硬件并增加复杂性。
如果新写入数据在 被复制到备用系统之前，工作系统出现故障，则有可能会丢失数据。

复制
主-从复制

主-主复制
。。这2个在数据库里 讲到。

DNS
域名系统是把 域名 转换成IP地址。
DNS结果可以缓存在浏览器或操作系统中一段时间，时间长短取决于存活时间TTL
NS记录（域名服务） 指定解析域名或子域名的DNS服务器
MX记录（邮件交换） 指定接收信息的邮件服务器
A记录（地址） 指定域名对应的IP地址
CNAME（规范） 一个域名映射到另一域名或CNAME记录。

某些DNS服务器通过几种方式来路由流量：
加权轮询调度
防止流量进入维护中的服务器
在不同大小集群间负载均衡
A/B测试
基于延迟路由
基于地理位置路由

内容分发网络CDN
CDN是一个全球性的代理服务器分布式网络，它从靠近用户的位置提供内容。
通常，HTML/CSS/JS，图片，视频等静态内容由CDN提供。

负载均衡器

random
round robin
least busy
sticky session / cookies
by request parameters

负载均衡器将传入的请求分发到应用服务器和数据库等计算资源。
效果在于：
防止请求进入不好的服务器
防止资源过载
帮助消除单一的故障点

可以通过 硬件(昂贵) 或 HAProxy 等软件 实现。增加的好处包括：
SSL终结 解密传入的请求并加密服务器响应，这样的话后端就不必再执行这些潜在的高消耗运算了。
Session留存 如果web应用程序不追踪会话，发出cookie并将特定客户端的请求路由到同一实例。

通常会设置采用 工作-备用 和 双工作 模式的多个负载均衡器，以免发生故障。

负载均衡器能基于多种方式来路由流量
随机
最少负载
session/cookie
轮询调度 或 加权轮询调度
四层负载均衡
七层负载均衡

四层负载均衡
根据 监听传输层的信息 来决定如何分发请求。
通常，这会涉及来源，目标IP地址，和请求头中的端口，但不包括数据包内容。
四层负载均衡执行 网络地址转换(NAT) 来向上游服务器转发网络数据包。

七层负载均衡器
监控 应用层 来决定怎样分发请求。
这会涉及 请求头的内容，消息和cookie。
七层负载均衡器终结网络流量，读取消息，做出负载均衡判断，然后传送给特定服务器。
比如，一个七层负载均衡器能直接将视频流量连接到托管视频的服务器，同时将更敏感的用户账单流量引导到安全性更强的服务器。

以损失灵活性为代价，四层比七层负载均衡 花费更少时间和计算资源，虽然这对现代商用硬件的性能影响甚微

水平扩展
负载均衡器还能帮助水平扩展，提高性能和可用性。
使用商用硬件的性价比更高，并且比在单台硬件上垂直扩展更贵的硬件 具有更高的可用性。

水平扩展缺陷

水平扩展引入了复杂度并涉及服务器复制

服务器应该是无状态的：它们也不该包含像session或资料图片等用户关联的数据。

session可以集中存储在数据库或持久化缓存(redis,memcached)的数据存储区中

缓存和数据库等下游服务器需要随着上游服务器进行扩展，以处理更多的并发连接。

负载均衡器缺陷

如果没有足够的资源配置或配置错误，负载均衡器会变成一个性能瓶颈。

引入负载均衡器以帮助 消除单点故障，但增加了复杂性

单个负载均衡器会导致单点故障，但配置多个负载均衡器会进一步增加复杂性。

反向代理(web服务器)

反向代理是一种可以集中地调用内部服务，并提供统一接口给公共客户的web服务器

来客客户端的请求向被反向代理服务器转发到可以相应请求的服务器，然后代理再把服务器的相应结果返回给客户端。

。。。负载均衡 应该是 对 一个服务的 大量请求的 均衡。 反向代理 是对 多个服务包装成 一个接口，当客户端请求的时候， 反向代理需要将请求转发 到 正确的服务上。

。。所以 应该先 反向代理，然后 负载均衡。 岂不是2个瓶颈了。。

。。不， 负载均衡器，必然是反向代理的。

。。而且，上面也没有说 反向代理 是 封装不同的服务 到同一个接口。。 估计反向代理也是 封装多个相同的服务实例 到一个接口。。。

。。确实，封装不同的接口 到同一个接口 干什么。。闲的。。

反向代理好处：

增加安全性 隐藏后端服务器的信息，屏蔽黑名单，限制客户端的连接数

提高可扩展性和灵活性 客户端只需要知道反向代理服务器的IP，这样你可以修改后端服务器的数量或配置。

本地终结SSL会话 解密传入请求，加密服务器响应，这样后端就不需要执行这些潜在的高消耗的操作。

免除了在每个服务器上安装X.509证书的需要

压缩 压缩服务器响应

缓存 直接返回命中的缓存结果

静态内容 直接提供静态内容：html/js/css,图片，视频等。

负载均衡器与反向代理

当你有多个服务器时，部署负载均衡非常有用，通常，负载均衡器将流量路由给一组功能相同的服务器上。

即使只有一台服务器，反向代理也有用。

nginx，HAProxy 等解决方案 可以同时 支持 7层反向代理和负载均衡。

反向代理不利之处

增加系统复杂度

单独一个反向代理服务器仍可能发生单点故障，多台反向代理服务器会增加复杂度

应用层

将Web服务层 与 应用层(也成为平台层) 分离，可以独立缩放和配置这两层。

添加新的API只需要添加应用服务器，而不必添加额外的Web服务器。
。。看配图是，request - load balancer - web server - platform server - database
。。或许 web服务层 是 业务逻辑， 应用层 是 docker吧。。。不，下面说到了 应用层 可以异步化。。估计 应用层 是抽取出来的 服务吧。。或者是Saas Paas 那种吧。

单一职责原则 提倡 小型的，自治的 服务 共同合作。

应用层中的工作进程也 可以实现异步化。

微服务
可以描述为 一系列可以独立部署的小型的模块化服务。每个服务运行在一个独立的线程中，通过明确定义的轻量级机制通讯，共同实现业务目标。

例如，可以有以下微服务：用户资料，关注者，图片上传，搜索 等。

服务发现
Consul，Etcd，Zookeeper 可以通过 追踪注册名，地址，端口 等信息 来帮助服务互相发现对方。
Health check 可以帮助确定服务的完整性和可达性
Consul，Etcd 都有一个内建的 k-v存储 来存储配置信息和其他的共享信息。

应用层缺点
添加由多个松耦合服务组成的应用层，从架构，运营，流程 等方面来讲 将非常不同(相对于单体系统)。。。估计是说 改变太大。
微服务会增加部署和运营的复杂度。

数据库
配图： 多个web instance 都指向同一个cache(cache只有一个)。 也都指向 3个数据库(一共就3个数据库，一个write master，2个read replica)。

关系型数据库管理系统 (RDBMS)
一系列以表的形式组织的数据项集合。

ACID是描述 关系型数据库事务的特性
原子性 每个事务内部所有操作要么全部完成，要么全部不完成
一致性 任何事务都使数据库从一个有效状态转换到另一个有效状态
隔离性 并发执行事务的结果 和顺序执行事务的结果相同
持久性 事务提交后，对系统的影响是永久的。

关系型数据库扩展包括许多技术，主从复制，主主复制，联合，分片，非规范化，sql调优。

主从复制
配图，一个master，2个slave，master指向slave(作为replication)。client的读可以从master和slave上读取，但是写只能到master上写。

主库同时负责读取和写入操作，并复制写入到一个或多个从库中，从库只负责读操作。
树状形式的从库再将写入复制到更多的从库中。
如果主库离线，系统可以以只读模式运行，直到某个从库被提升为主库，或有新的主库出现。

主从复制缺点
将从库提升为主库需要额外的逻辑
参考 复制缺点 (这个是下面主主复制 也会碰到的，所以在主主复制里)

主主复制

配图，2个master，互相指向(作为replication)，client的读、写 都可以在2个master上执行。

2个主库都负责读和写，写入时相互协调。 如果一个主库挂机，系统可以继续读取和写入。

主主复制缺点
需要添加负载均衡器或在应用逻辑中做改动，来确定写入哪一个数据库
多数主-主系统 要么不能保证一致性(违反ACID)，要么因为同步而产生写入延迟
随着更多写入节点的加入和延迟的提高，如何解决冲突显得越发重要。
参考 复制缺点

复制缺点
如果主库在将新写入的数据复制到其他节点前挂掉，则数据可能丢失。
写入会 重新在负责读取的副本 上执行，副本可能因为过多的写操作阻塞住。
读取从库越多，需要复制的写入数据就越多，导致更严重的复制延迟。
某些数据库系统中，写入主库的操作可能多个线程并行写入，但是 读取副本 只支持单线程顺序写入。
复制意味着跟多的硬件和额外的复杂度。

联合

。。分库。。不 这个不是分库 也不是分表。。不，是分库， 垂直分库。
联合(或按功能划分) 将数据库按对应功能分割。
例如，你有3个数据库：论坛，用户和产品，而不是仅仅一个单体数据库，从而减少每个数据库的读取和写入流量，减少复制延迟。
较小的数据库意味着更多适合放入内存的数据，从而提高缓存命中几率。。。。估计是说DBMS自己的缓存，数据库小，那么DBMS内存中的数据更容易被命中。
没有只能串行写入的中心化主库，你可以并行写入，提高负载能力

联合缺点
如果你的数据库模式需要 大量的功能和数据表，联合的效率并不好。
你需要更新应用程序的逻辑 来确定要读取和写入哪个数据库。
用server link 从2个数据库 联结数据更复杂。。。。(left join 跨2个数据库)
联合需要更多的硬件 和额外的复杂度。

分片
。。分库。　水平分库
配图是application 需要user的数据，访问load balance，load balance根据所查询用户的首字母去不同的数据库里查询。
分片将数据分配到不同的数据库上，使得每个数据库仅管理整个数据集的一个子集。
类似联合的优点，分片可以减少读取和写入流量，减少复制并提高缓存命中率。也减少了索引，通常意味着查询更快，性能更好。
如果一个分片出问题，其他仍能运行，你可以使用某种形式的冗余来防止数据丢失。
类似联合，没有只能串行写入的中心化主库，你可以并行写入，提高负载能力。
常见做法是 用户姓氏的首字母或用户的地理位置来分隔用户表。

分片缺点
你需要修改应用程序的逻辑来实现分片，这会带来复杂的sql查询。
分片不合理可能导致数据负载不均衡。例如，被频繁访问的用户数据会导致其所在分片的负载相对其他分片高。
　　再平衡会引入额外的复杂度。基于 一致性哈希 的分片算法可以减少这种情况
联结多个分片的数据操作更复杂。
分片需要更多的硬件和额外的复杂度。

非规范化
非规范化试图以写入性能为代价来换取读取性能。
在多个表中冗余数据副本，以避免高成本的联结操作。
一些关系型数据库，比如PostgreSQL 和 Oracle 支持物化视图，可以处理冗余信息存储和保证冗余副本一致。

当数据使用诸如 联合 和 分片 等技术被分割，进一步提高了处理跨数据中心的联结操作复杂度。　非规范化可以避免这种复杂的联结操作。

大多数系统中，读取操作的频率远高于写入操作，比例可达到100:1，甚至1000:1。需要复杂的数据库联结的读取操作成本非常高，在磁盘操作上消耗了大量的时间。

非规范化缺点
数据会冗余
约束可以帮助冗余的信息副本保持同步，但这会增加数据库设计的复杂度。
非规范化的数据库再高写入负载下性能可能比规范化的数据库差。

sql调优
利用 基准测试 和 性能分析 来模拟和发现系统瓶颈很重要。
　　基准测试　用ab等工具模拟高负载情况 。。。apache bench
　　性能分析　通过启用 慢查询日志 等工具来辅助追踪性能问题。

基准测试和性能分析 可能会指引你到下面的优化方案。

改进模式
为了实现快速访问，MySQL在磁盘上用连续的块存储数据
使用char类型存储固定长度的字段，不要用varchar
　　char在快速，随机访问中效率很高。使用varchar，如果你想读取下一个字符串，你不得不先读取到当前字符串的末尾。

使用text类型存储大块的文本，例如博客正文。text还允许布尔搜索。使用text字段需要在磁盘上存储一个用于定位文本块的指针。
使用int存储2^32 或 40亿 的较大数字。
使用decimal 存储货币
避免blobs存储实际对象，而是用来存储存放对象的位置。。。这个还能设置？
varchar(255) 是以8位数字存储的最大字符数，在某些关系型数据库中，最大限度地利用字节。。。。？
在适合的场景中设置 not null 约束 来提高搜索性能。

使用正确索引
你正查询(select, group by, order by, join)的列 如果用了索引会更快。
索引通常表示为自平衡的B树，可以保持数据有序，并允许在对数时间内进行搜索，顺序访问，插入，删除操作。
设置索引，会将数据存在内存中，占用了更多内存空间。。。第一次知道，索引在内存中。
写入操作会变慢，因为索引需要被更新。
加载大量数据时，禁用索引再加载数据，然后重建索引，这样也许会更快。

避免高成本的联结操作
有性能需要，可以进行非规范化。

分割数据表
将热点数据拆分到单独的数据表中，可以有助于缓存。

调优查询缓存
某些情况下，查询缓存可能会导致性能问题。

NoSQL
是 键-值数据库，文档型数据库，列型数据库，图数据库 的统称。
非规范化的，表联结大多在应用程序代码中完成。
大多数NoSQL无法实现 真正符合ACID的事务，支持最终一致。

BASE 通常用来表述NoSQL 数据库的特性。相比CAP理论，BASE强调可用性超过一致性。
- 基本可用(basically available) 系统保证可用性
- 软状态(soft state) 即使没有输入，系统状态也可能随着时间变化。
- 最终一致性(eventual consistency) 经过一段时间后，系统最终会变一致。

NoSQL有很多类型，所以不仅要在SQL NoSQL中选择，也要在NoSQL的多种类型中选择。

键-值存储
抽象模型：哈希表
通常可以实现O(1) 读写， 用内存或SSD存储数据。
数据存储可以按 字典顺序维护键，从而实现键的高效检索。
键-值存储可以用于存储元数据。

性能很高，通常用于存储简单数据模型 或频繁修改的数据，如放在内存中的缓存。
键-值存储提供的操作有限，如果需要更多操作，复杂度将转嫁到应用程序层面。

键-值存储 是 文档存储的基础， 有时，甚至是图存储等更复杂的存储系统的基础。

文档类型存储
抽象模型：将文档作为值 的 键-值存储
文档类型存储以 文档(xml,json,二进制文件等) 为中心，文档存储了 指定对象的全部信息。文档存储根据 文档自身的内部结构提供API或查询语句来实现查询。

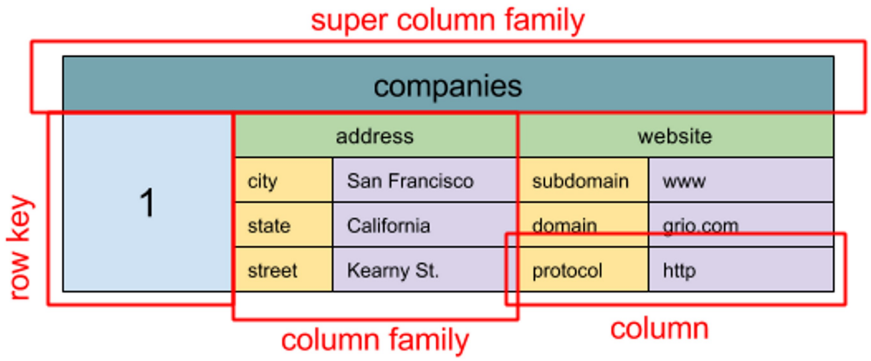
很多 键-值存储数据库 有用值存储元数据的特性，这也模糊了 这2种存储类型的界限。

基于底层实现，文档可以根据集合，标签，元数据或文件夹组织。
尽管不同文档可以被组织在一起或者 分成一组，但是相互之间可能具有完全不同的字段。

MongoDB 和 CouchDB 等一些文档类型存储还提供了类似sql的查询语句来实现复杂查询。DynamoDB同时支持 键-值存储 和 文档类型存储。

文档类型存储具备高度的灵活性，常用于处理 偶尔变化的数据。
。。。估计是说 可以存任何东西，但是 这个东西不能一直变化(变化会很消耗CPU资源)。

列型存储



抽象模型：嵌套的 ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>> 映射
类型存储的基本数据单元是列(名/值对)。列可以在列族(类似SQL的数据表) 中被分组。超级列族 再分组普通列族。你可以使用行键 独立访问每一列，具有相同行键值的列组成一行。每个值都包含版本的时间戳用于解决版本冲突。

Google发布了第一个列型存储数据库 Bigtable， 它影响了Hadoop生态系统中活跃的开源数据库HBase 和 Fasebook的 Cassandra。
以上3种列存储数据库 把 键以字母顺序存储，可以高效地读取键列。

列型存储 具备高可用性 和 高可扩展性。通常被用于大数据相关存储。

图数据库
抽象模型：图
。。。。。图论的图。。不是 picture。。

在图数据库中，一个节点对应一条记录，一条边对应2个节点之间的关系。
图数据库被 优化用于表示 外键繁多的复杂关系或 多对多关系。

图数据库为 存储复杂关系的数据模型，如社交网络，提供了很高的性能。
相对较新，尚未广泛应用，查找开发工具或资源相对较难。许多图只能通过rest api访问。

SQL 还是 NoSQL

选择SQL的原因

- 结构化数据
- 严格的模式
- 关系型数据
- 需要复杂的联结操作
- 事务
- 清晰的扩展模式
- 既有资源更丰富：开发者，社区，代码库，工具等。
- 通过索引进行查询非常快。

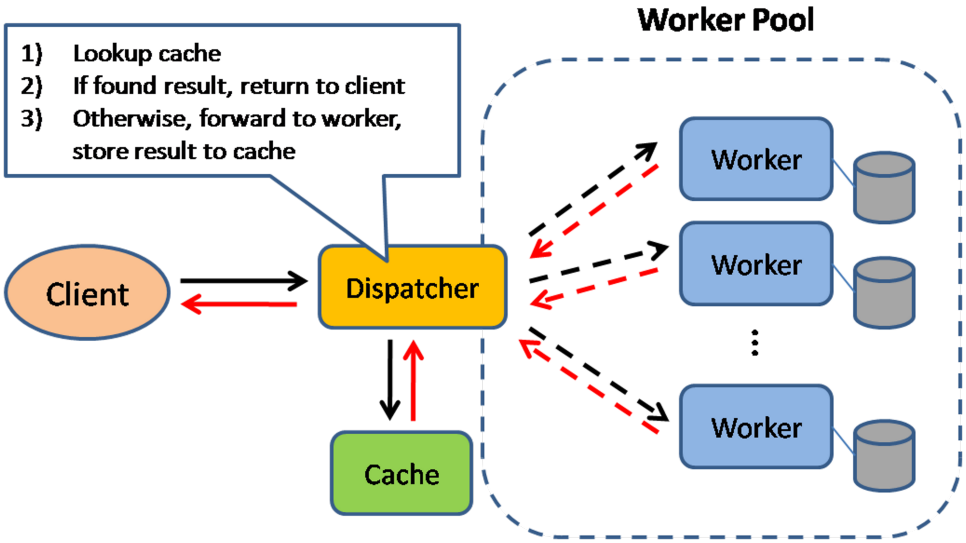
选择NoSQL的原因

- 半结构化数据
- 动态或灵活的模式
- 非关系型数据
- 不需要复杂的联结操作
- 存储TB(甚至PB)级别的数据
- 高数据密集的工作负载
- IOPS高吞吐量

适合NoSQL的实例：

- 埋点数据和日志数据
- 排行榜或者得分数据
- 临时数据，如购物车
- 频繁访问的表
- 元数据/查找表

缓存



缓存可以提高页面加载速度，并减少服务器和数据库的负载。在上面的模型中，分发器先查看请求 是否被响应过，如果有则将之前的结果直接返回。

数据库分片均匀分布的读取是最好的。但是热数据会让读取分布不均匀，这样就会造成瓶颈，如果在数据库前面加一个缓存，就可以抹平不均匀的负载 和 突发流量对数据库的影响。

客户端缓存
cache可以位于客户端(os或浏览器)，服务端 或不同的缓存层。

CDN缓存
CDN也被视为一种缓存。

Web服务器缓存
反向代理和缓存(如Varnish) 可以直接提供静态和动态内容。Web服务器同样可以缓存请求，返回响应结果 而不必连接应用服务器。
。。Web服务器是指 load balance？不
。。web服务器是为了提供http内容， 应用服务器也可以提供http内容，还可以提供其他协议支持，如rmi/rpc
。。web服务器主要是为 提供静态内容而设计的，不过大多数web服务器都有插件来支持脚本语言(perl,php,asp,jsp)，通过这些插件，这些服务器就可以生成动态的http内容。

数据库缓存
数据库的默认配置中通常包含缓存级别，针对一般用例进行了优化。调整配置，在不同情况下使用不同的模式可以进一步提高性能。

应用缓存
基于内存的缓存，如Memcached,Redis，是应用程序和数据存储之间的 一种键值存储。由于数据保存在RAM中，它比存储在磁盘上的典型数据库要快多了。
RAM比磁盘限制跟多，所以例如LRU的算法可以将 热数据放在RAM中。

Redis有以下附加功能：
- 持久性选项
- 内置数据结构，如有序集合，列表。

有多个缓存级别，分为2大类：数据库查询和对象：
- 行级别
- 查询级别
- 完整的可序列化对象
- 完全渲染的html

一般来说，应该尽量避免 基于文件的缓存，因为这使得复制和自动缩放很困难。

数据库查询级别的缓存
当你查询数据库的时候，将查询语句的hash值与 查询结果存储到缓存中。这种方法会遇到下面的问题：
- 很难用复杂的查询删除已缓存结果。
- 如果一条数据被修改，则需要删除所有包含这条数据的缓存结果。

对象级别的缓存
将你的数据视为对象，就像对待的你应用代码一样。让应用程序将数据从数据库中组合到

类实例或数据结构中：

如果对象的基础数据已经更改了，那么从缓存中删除这个对象。

允许异步处理：workers通过使用最新的缓存对象来组装对象。

建议缓存的内容：

用户会话

完全渲染的web页面

活动流

用户图数据

何时更新缓存

缓存中只能存储有限的数据，所以你需要一个适用于你用例的缓存更新策略。

缓存模式 cache-aside

图：client指向storage，client指向cache

应用从存储器读写，缓存不和存储器直接交互，应用执行以下操作：

在cache中查找记录，如果所需的数据不在缓存中：

从数据中加载所需内容

将查询到的结果缓存到cache中

返回所需内容。

Memcached 通常通过这种方法来使用。

添加到缓存中的数据读取速度很快。 缓存模式也成为延迟加载。只缓存所请求的数据，避免被 没有请求的数据占满 缓存空间。

缓存模式的缺点：

请求的数据如果不在缓存中，就需要3个步骤来获取数据，会导致明显的延迟

如果数据库的数据更新了，会导致缓存中的数据过时。这需要通过设置TTL强制更新缓存或 直写模式来缓解这种情况。

当一个节点出现故障，它会被一个新节点替代，这增加了延迟。

直写模式 write-through

图：client指向cache，cache指向storage

应用使用cache作为主要的数据存储，在缓存上进行读写，而缓存负责和数据库交互。

。。。根据后面的回写模式，，这个直写模式 是同步的，就是 要等 cache和数据库交互完以后，cache才会返回到client

应用向 缓存中 添加/更新 数据

缓存同步地写入到 数据存储

返回所需内容。

由于写操作的存在，所以直写模式整体是很慢的操作，但是读取刚写入的数据很快。

相比读取数据，用户通常更能接受 更新数据时慢。 缓存中的数据不会过时。

直写模式缺点

由于故障或者缩放而创建新节点，这些新节点不会缓存内容，直到这些内容在数据库被更新。 在直写模式中联合使用缓存模式 可以缓解这个问题。

写入的大多数数据可能永远不会被读取，使用TTL可以最小化这种情况的出现。

回写模式write-behind

图：client 写入到 cache，cache添加一个写入事件到queue，cache返回； 另外的线程会处理queue中的事件，来讲事件写入到DB。
。。但是这里怎么都是 写的模式。 直写，回写。。都是写的。。如果是读呢？直写的读还能理解， 回写的读，总不能返回一个future吧？还是说，读写分离了？

在回写模式中，应用执行以下操作
在缓存中增加或更新条目
异步写入数据，提高写入性能

回写模式缺点
缓存可能在其内容成功存储之前丢失数据
实现回写模式 比 实现缓存模式，直写模式 都要更复杂。

刷新refresh-ahead
你可以将缓存 配置成 在到期之前自动刷新最近访问过的内容。
如果缓存可以准确预测将来可能请求哪些数据，那么刷新 可以降低延迟。

刷新的缺点
不能准确预测到未来需要用到的数据，这可能导致性能不如 不使用刷新

缓存的缺点
需要保持缓存和真实数据之间的一致性，如数据库的 cache invalidation.
需要改变应用程序，来增加redis或memcached
无效缓存是一个难题，什么时候更新缓存 是一个复杂的问题。

异步
异步工作流有助于减少那些原本顺序执行的请求的等待时间。它们可以通过提前一些耗时的工作来帮助减少请求的等待时间，比如定期汇总数据。

消息队列 Message queues
消息队列 接收，保留和传递 消息。如果按顺序执行操作太慢的话，你可以使用有以下工作流的消息队列：
应用程序将作业发布到队列，然后通知用户作业状态
一个worker从队列中取出改作业，对其进行处理，然后显示该作业已完成。

不阻塞用户操作，作业在后台被处理。

Redis 是一个令人满意的简单的消息代理，但消息可能会丢失
RabbitMQ 很受欢迎，但是要求你 适应 AMQP协议 并且管理你自己的节点
Amazon SQS 是被托管的，但可能具有高延迟，并且消息可能会被传递2次。

任务队列 task queues
任务队列接收任务及其相关数据，运行它们，然后传递其结果。它们可以支持调度，并可用于在后台运行计算密集型作业。

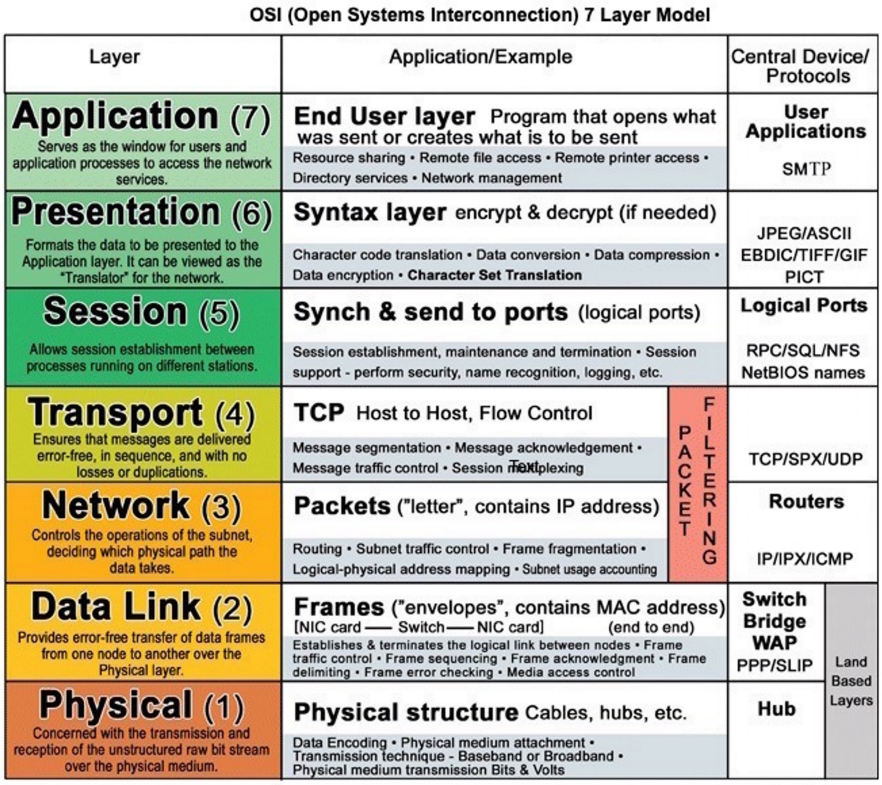
Celery 支持调度，是使用Python开发的。

背压 back pressues
如果队列开始明显增长，那么队列大小可能会超过内存大小，导致高速缓存未命中，磁盘

读取，甚至性能更慢。
背压可以通过限制队列大小来帮助我们，从而为队列中的作业保持高吞吐率和良好的响应时间。
一旦队列填满，客户端将受到服务器忙 或者 503，以便稍后重试。
。。就这？就是队列满，就直接返回 信息。。？

异步的缺点
简单的计算和实时工作流 等用例可能更适用于同步操作，因为引入队列可能会增加延迟和复杂性。

通讯



超文本传输协议(HTTP)

http是一种在客户端和服务器之间 编码和传输数据的 方法。
它是一个请求/响应协议：客户单和服务端 针对相关内容和完成状态信息 的请求 和响应。
http是独立的，允许 请求和响应 流经许多 中间路由器和服务器(这些可能执行 负载均衡，缓存，加密，压缩)

一个级别的http请求由 一个动词(方法) 和一个资源(端点) 组成。下面是常见http动词

动词	描述	幂等	安全性	可缓存
GET	读取资源	T	T	T
POST	创建资源或触发处理数据的进程	F	F	T如果响应包含刷新信息。Only cachable if response contains explicit freshness information
PUT	创建或替换资源	T	N	N

PATCH	部分更新资源	N	N	T通过响应包含刷新信息。Only cachable if response contains explicit freshness information
DELETE	删除资源	T	F	F

幂等：多次执行 不会 产生不同的结果。

http是依赖于 较低级协议(如tcp和udp) 的应用层协议

传输控制协议(TCP)
tcp是通过 ip网络 的面向连接的协议。 使用握手 建立和断开连接。发送的所有数据包保证以原始顺序到达目的地，用以下措施保证数据包 不被损坏：
每个数据包的 序列号 和校验码
确认包和自动重传

如果发送者没有收到正确的响应，它将重新发送数据包。如果多次超时，连接就会断开。
tcp实行流量控制 和拥塞控制。这些确保措施 会导致延迟，而且通常导致传输效率比 udp低。

为了确保高吞吐量，web服务器可以保持大量的tcp连接，从而导致高内存使用(估计是说内存消耗大。。)。
在web服务器线程间拥有大量开放连接可能开销巨大。
连接池有助于 在适当的时候 切换到UDP

TCP对于 需要高可靠性 但时间紧迫 的应用程序很有用。 比如 web服务器，数据库，SMTP，FTP，SSH。

以下情况下优先使用tcp
需要数据完好无损
想对网络吞吐量自动进行最佳评估

用户数据报协议 (UDP)
udp是无连接的。数据报(类似数据包) 只在数据报级别有保证。
数据报可能会无序地到达目的地，也可能丢失。
udp不支持拥塞控制
虽然不如tcp那样有保证，但是udp通常效率更高。

udp可以通过广播将数据报 发送到子网内的所有设备。这对dhcp很有用，因为子网内的设备还没有分配IP地址，而IP对于tcp是必须的。
。。udp在 无ip的 时候。。不，子网它依然是有ip 的，只不过 广播不需要精确到哪个ip，因为无连接。。 连接是需要知道 对方ip 的。

udp可靠性低，适合用于 网络电话，视频聊天，流媒体，实时多人游戏上。

以下情况使用udp
需要低延迟
相对于 数据丢失， 更糟的是 数据延迟
你想实现自己的错误 校正方法。

远程过程调用协议 (RPC)

在rpc中，客户端会去调用另一个地址空间(通常是一个远程服务器)的方法。调用代码看起来就像是调用一个本地方法；
客户端和服务器交互的具体过程被抽象。
远程调用相对于本地调用一般较慢而且可靠性更差。

热门的rpc框架包括 Protobuf，Thrift，Avro

RPC是一个 请求-响应 协议

- 客户端程序　调用客户端存根程序，就想调用本地方法一样，参数会被压入栈中
- 客户端stub程序　将请求过程的id和参数打包进请求信息中
- 客户端通信模块　将信息从客户端发送至服务端
- 服务端通信模块　将接受的包传给服务端存根程序
- 服务端stub程序　将请求解包，根据过程id 调用服务端方法并将参数传递进去。

rpc专注于暴露方法，rpc通常用于处理内部通讯的性能问题，这样你可以手动处理本地调用以更好的适应你的情况。

以下情况时选择本地库(也就是SDK)

- 你知道你的目标平台
- 你想控制如何访问你的逻辑
- 你相对发生在你的库中的错误进行控制
- 性能和终端用户体验是你关系的事

rpc缺点

- rpc客户端和服务实现 捆绑地很紧密
- 一个新的api 必须在每一个操作 或用例中定义
- rpc很难调试。
- 你可能没有办法很方便地去修改现有的技术。如，如果你希望在Squid这样的缓存服务器上确保 RPC被正确缓存 的话，可能需要一些额外的努力了。

表述性状态转移(REST)

rest是一种强制的 客户端/服务端 架构设计模型，客户端基于服务端管理的一系列资源操作。
服务端提供修改或 获取资源的接口。所有的通信必须是无状态和可缓存的。

RESTful 接口有4条规则：

- 标志资源(http的uri)　使用和操作无关的 uri
- 表示的改变(http的工作) 使用 verbs(动词)，headers和body
- 可自我描述的错误信息(http的status code) 使用状态码，不要重复造轮子
- HATEOAS(http中html接口) 你的web服务器应该能够通过浏览器访问。

rest关注于暴露数据，它减少了客户端/服务端 的耦合程度，经常用于公共http api接口设计。
rest使用更通常和更规范化的方法 来通过uri暴露资源。
通过header来表述，并通过get, post, put, delete, patch来进行操作。

因为无状态的特性，rest易于横向扩展和隔离。

rest缺点

由于rest 重点在于 暴露数据，所以当资源不是自然组织的，或者资源结构复杂时，它无法很好的适应。如，返回过去一个小时中与特定事件匹配的更新记录，这种操作就很难表示为路径。使用rest，可能会使用uri路径，查询参数和可能的请求体来实现。

rest 一般依赖于几个动作(get,post,put,delete,patch)，但有时这些没有办法满足你的要求，如，将过期文档移动到归档文件夹。

为了渲染单个页面，获取被嵌套在层级结构中的复杂资源需要客户端，服务器之间多次往返通信。如获得博客内容及其相关评论。对于使用不确定网络环境的移动应用来说，这些多次往返通信是非常麻烦的。

随着时间的推移，更多的字段可能会被添加到api响应中，较旧的客户端会收到所有新的数据字段，即使它们不需要这些字段，这会增加负载大小并引起更大的延迟。

RPC和REST比较

操作	RPC	REST
注册	POST /signup	POST /persons

注销	POST /resign
	{
	"personid": "1234"
	}
		DELETE /persons/1234

读取用户信息	get /readPerson?personid=1234	get /persons/1234
读取用户物品列表	get /readUsersItemsList? persionid=1234	get /persons/1234/items

向用户物品列表添加一项	post /addItemToUsersItemsList
	{
	"personid": "1234",
	"itemid": "456"
	}
		post /perssons/1234/items
		{
		"itemid":"456"
		}

更新一个物品	post /modifyitem
	{
	"itemid": "456",
	"key": "value"
	}
		put /items/456
		{
		"key": "value"
		}

删除一个物品	post /removeItem
	{
	"itemid": "456"
	}
		delete /items/456

安全

是一个很宽泛的话题。一般不需要了解太多，下面是一些安全基础知识：

在运输和等待过程中加密

对所有的用户输入 和 从用户那里得到的参数 进行处理 以防止 XSS 和 SQL注入

使用参数化的查询来防止sql注入
使用最小权限原则

延迟数

Latency Comparison Numbers

L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	10,000	ns	10 us		
Send 1 KB bytes over 1 Gbps network	10,000	ns	10 us		
Read 4 KB randomly from SSD*	150,000	ns	150 us		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	250 us		
Round trip within same datacenter	500,000	ns	500 us		
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000 us	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000	ns	10,000 us	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from 1 Gbps	10,000,000	ns	10,000 us	10 ms	40x memory, 10X SSD
Read 1 MB sequentially from disk	30,000,000	ns	30,000 us	30 ms	120x memory, 30X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150,000 us	150 ms	

Notes

1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns

基于上述数字的指标：

从磁盘以 30 MB/s 的速度顺序读取
以 100 MB/s 从 1 Gbps 的以太网顺序读取
从 SSD 以 1 GB/s 的速度读取
以 4 GB/s 的速度从主存读取
每秒能绕地球 6-7 圈
数据中心内每秒有 2,000 次往返

还有一些设计/真实 架构 的网址，不过感觉有点远。。