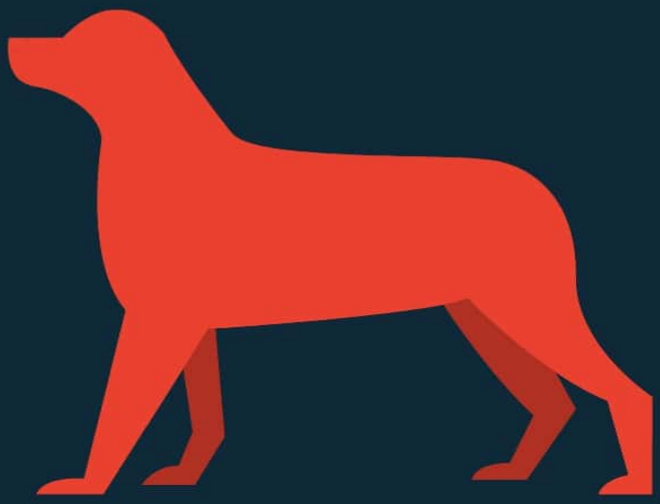
System Design



半栈程序员

等天黑译，主要内容来自于 Alex Xu 的《System Design Interview》

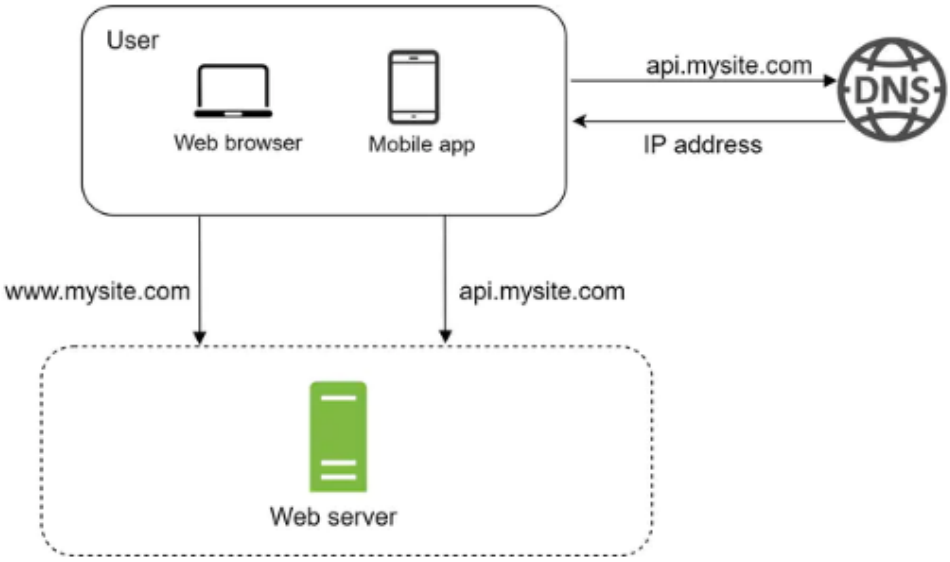
Table of contents

1. 设计一个支持百万用户的系统

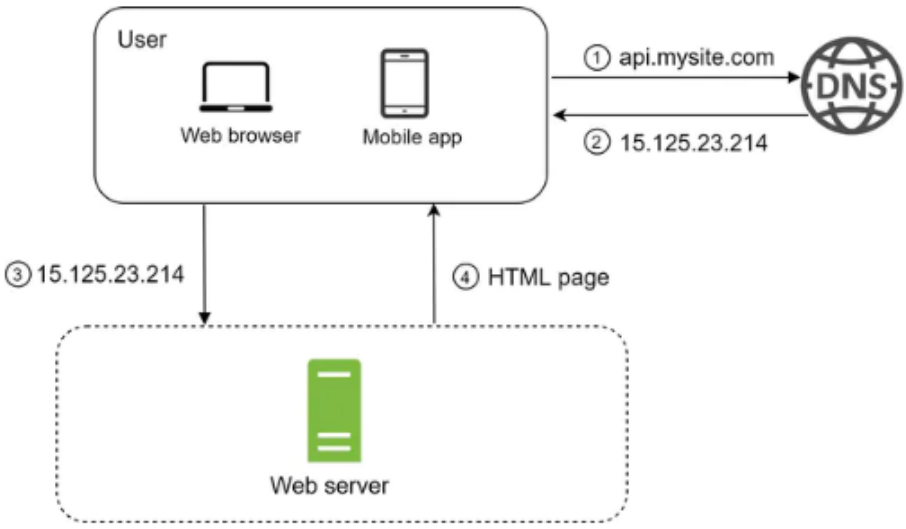
设计一个支持数百万用户的系统是非常有挑战性的, 这是一个需要不断调整和优化的过程, 接下来的内容中，我将构建一个系统，从单个用户开始，到最后支持数百万的用户。

从单个服务开始

千里之行，始于足下，让我们从最简单的单个服务开始。所有的内容都在一台服务器上运行，包括 Web 程序, 数据库，缓存 等等， 如下图



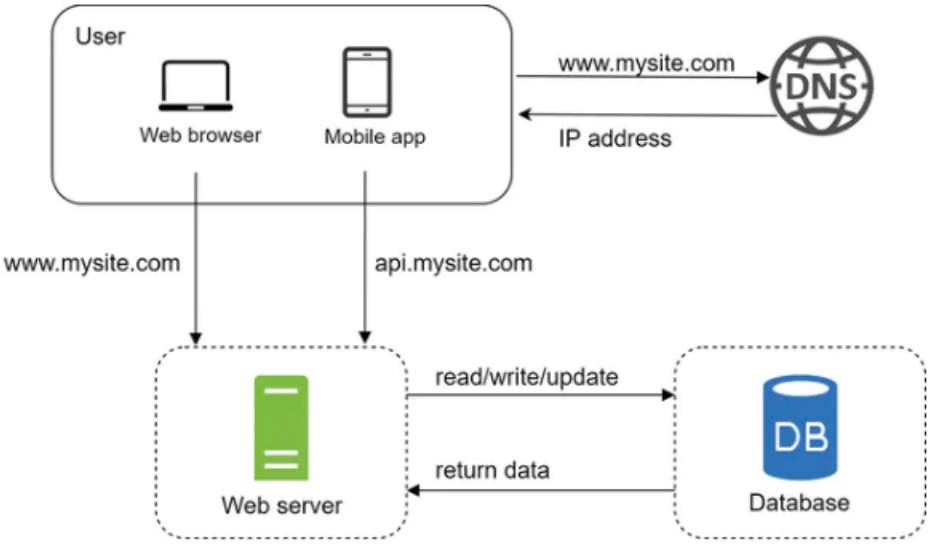
我们看一下它的工作流程。



1. 用户通过域名访问网站, 比如, api.mysite.com, 通常情况下, 域名解析服务 (DNS) 是由第三方提供的付费服务, 而不是我们的服务器所提供的。
2. 返回 IP 地址给浏览器或者移动设备, 比如, 15.125.23.214。
3. 通过 IP 地址, 发送 Http 请求到我们的 Web 服务器。
4. Web 服务器返回 html 或者 json 内容, 浏览器进行渲染。

分离数据库

随着用户量的增长，此时一台服务器已经独木难支，我们需要两台服务器, 一个用于 Web 服务, 一个用于数据库。



应该选择哪种数据库?

您可以选择关系型数据库和非关系型数据库，那它们都有什么特点呢?

关系型数据库也称为关系型数据库管理系统 (RDBMS) 或
SQL 数据库，最常见的有 MySQL、Oracle 、PostgreSQL、Sql Server 等，可以通过 SQL 进行跨表查询。

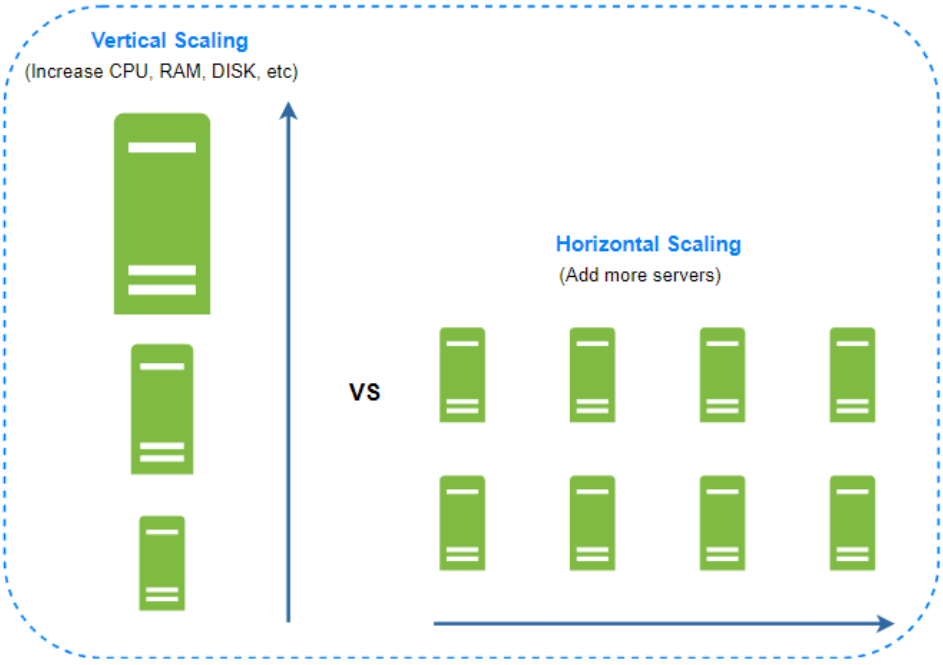
而非关系型数据库也称为 NoSQL 数据库，最常见的有 Redis、 CouchDB，Neo4j、Cassandra、HBase、Amazon DynamoDB 等。它们分为四类：键值(Key-Value)存储数据库、列存储数据库、文档型数据库、图(Graph)数据库。

对于大多数开发人员来说，通常会选择关系型数据库。而非关系型数据库更适合以下几种情况：

- 应用程序需要超低延迟。
- 数据是非结构化的，或者没有任何关系数据。
- 只需要序列化和反序列化数据（JSON、XML、YAML 等）。
- 需要存储海量数据。

垂直缩放 、 水平缩放

垂直缩放，又称为 "纵向扩展" (scale up), 是指升级服务器资源, 比如 CPU, RAM 等。而水平缩放又称为 "横向扩展" (scale out)，是指添加服务器到资源池中。



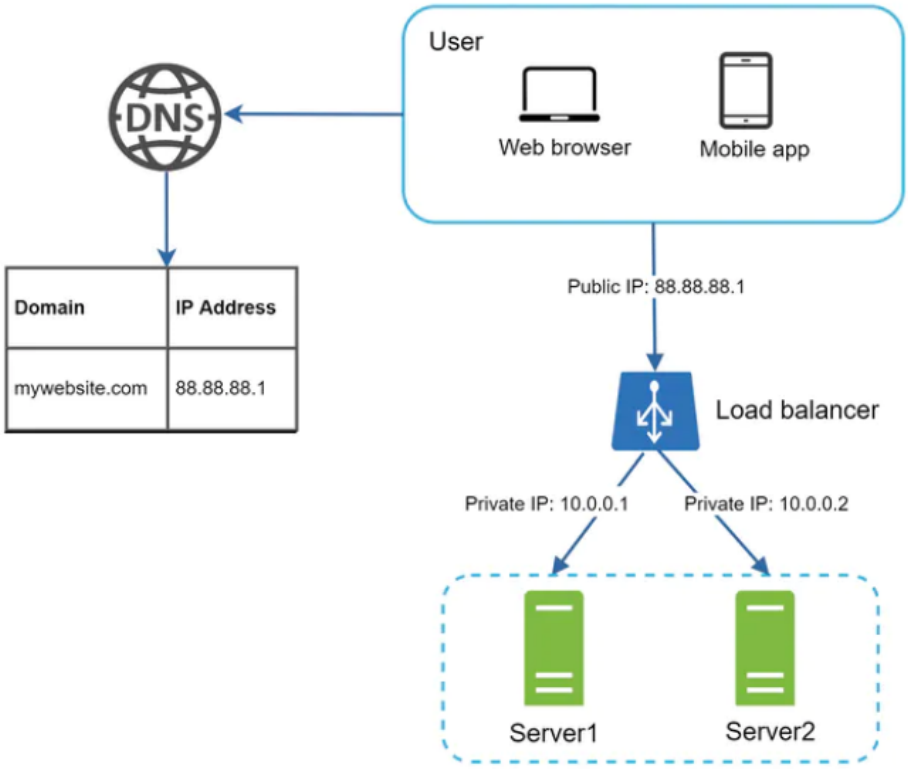
当流量比较少的时候， 选择纵向扩展就足够了，因为它足够简单，不过也有很大的局限性。

- 纵向扩展有硬件限制， 无限制的升级 CPU 和内存是不现实的。
- 纵向扩展没有高可用，如果一台服务器出现故障，网站或者应用就会直接崩溃。

而流量较大的时候，横向扩展是更好的选择，多个服务器也保证了高可用。如何让这些服务器更好的提供服务，我们还需要做负载均衡。

Load balancer

负载均衡器可以平均分配流量给每台服务器，如下



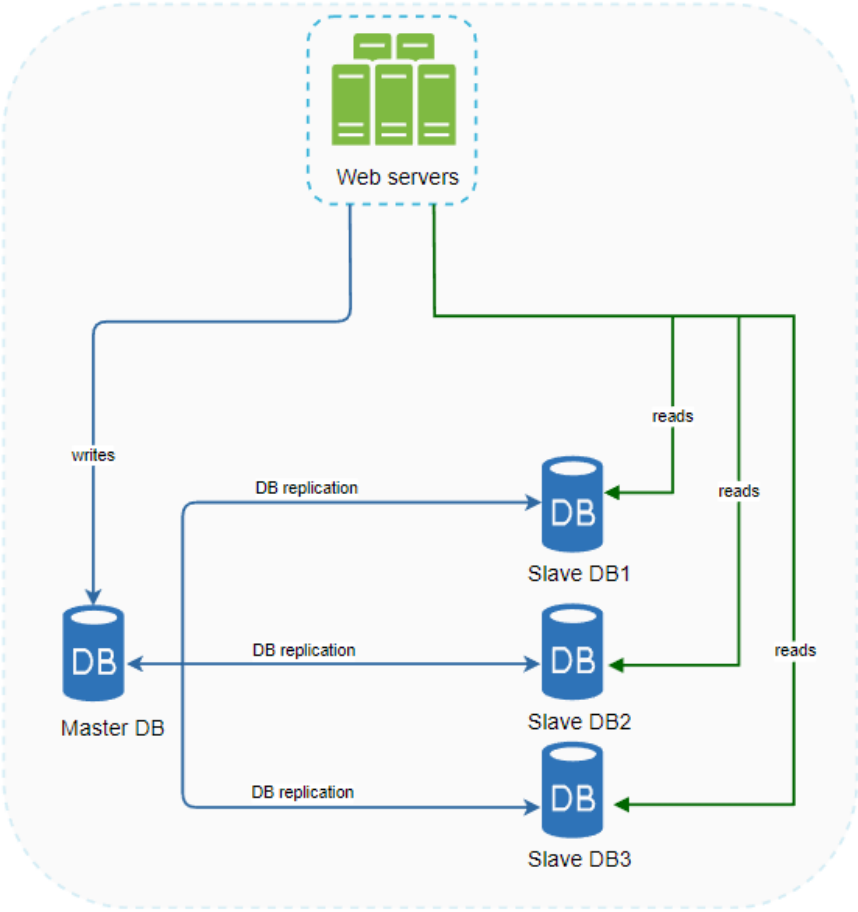
我们水平扩展了 Web 服务，并引入了负载均衡器，来应对快速增长的网站流量， 并提供了高可用的服务。

现在，Web 层看上去不错，但是不要忘了，当前的设计只有一个数据库，并不支持故障转移和冗余。而数据库复制是一种常见的技术，可以解决这个问题。

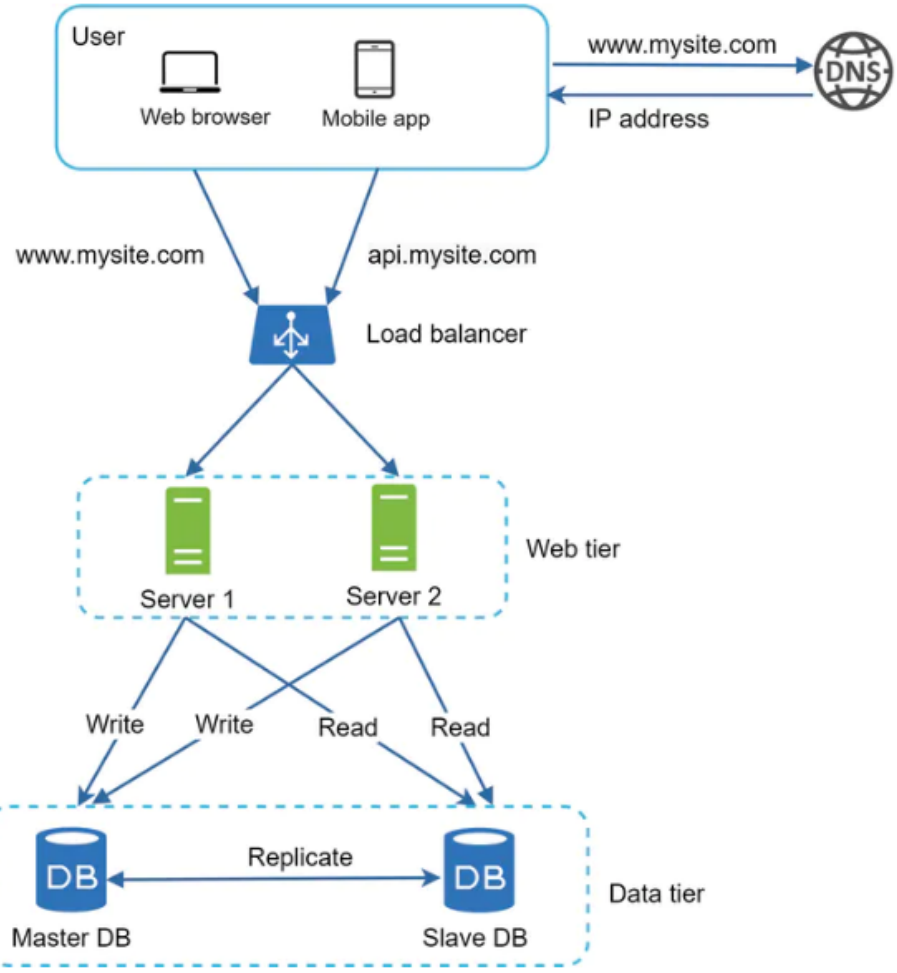
Database replication

数据库复制是把数据复制、传输到另外一个数据库，最终形成一个分布式数据库。用户可以访问到相同的信息，从而提高一致性、可靠性和性能。

通常它们之间是主/从(master/slave) 的关系，一主多从，主节点支持读写操作，而从节点仅支持读取操作，如下



引入了数据库复制， 让我们看看现在网站整体的设计。



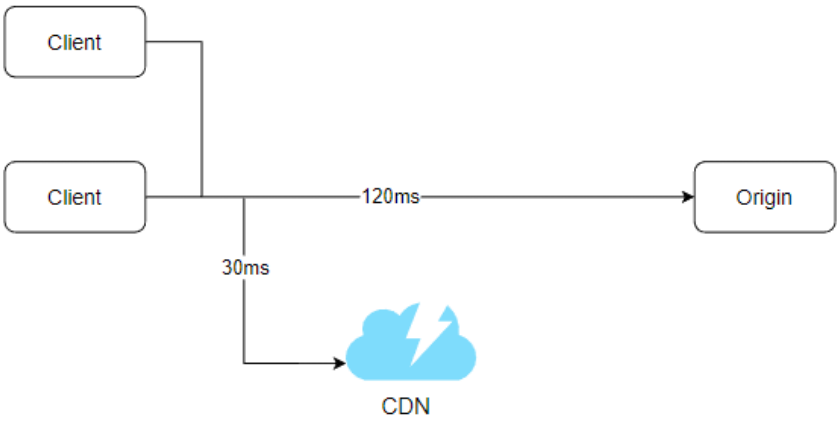
1. 用户从 DNS 获取到 Load balancer 的 IP 地址， 并连接到 Load balancer。
2. Http 请求被路由到服务器1 或者 服务器2。
3. 使用数据库复制，进行读写分离。

现在，web 服务和数据库都已经做了优化，看上去不错!

接下来，还需要提升 web 的加载和响应时间，我们可以使用 CDN 缓存静态资源, 包括 js、css、image 等。

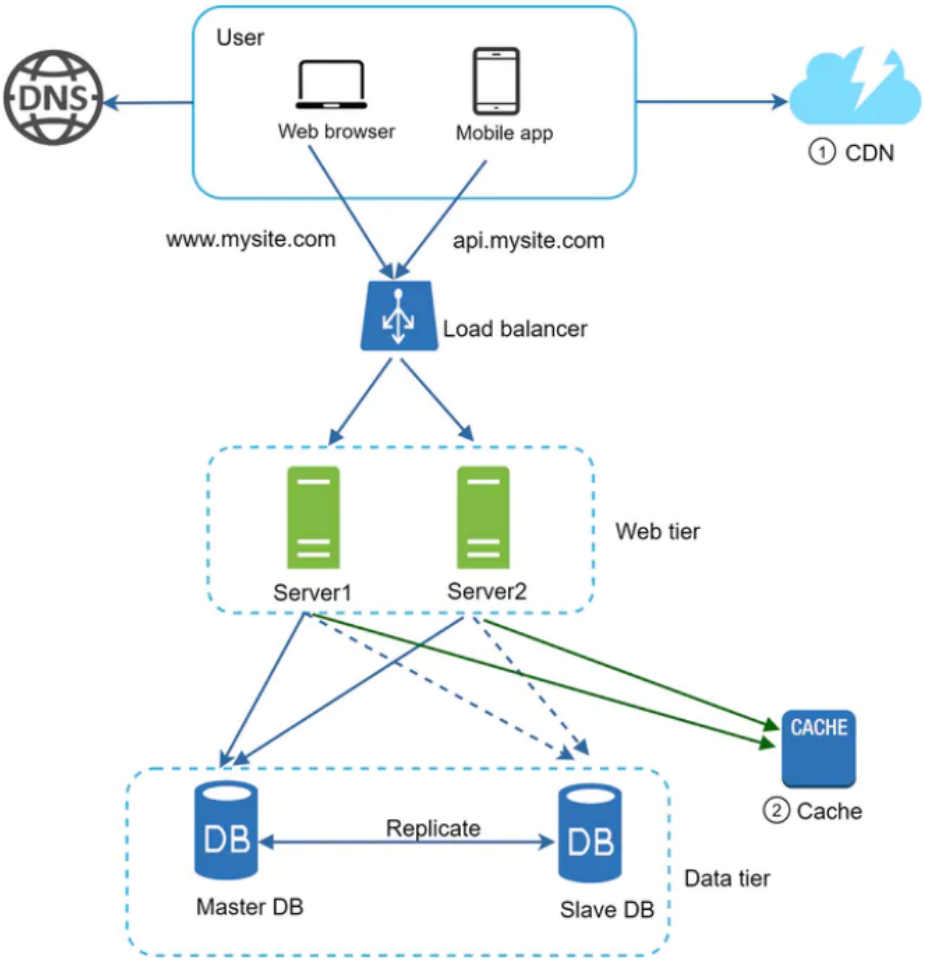
Content delivery network (CDN)

CDN 是一个用于交付静态内容的网络服务，分布在不同的地理位置。当用户访问网站时，距离最近的 CDN 服务器提供静态资源，可以很好的改善网站的加载时间。



另外，对于数据库来说，我们也可以把一些热点数据添加到缓存中，这样可以减轻数据库的压力。

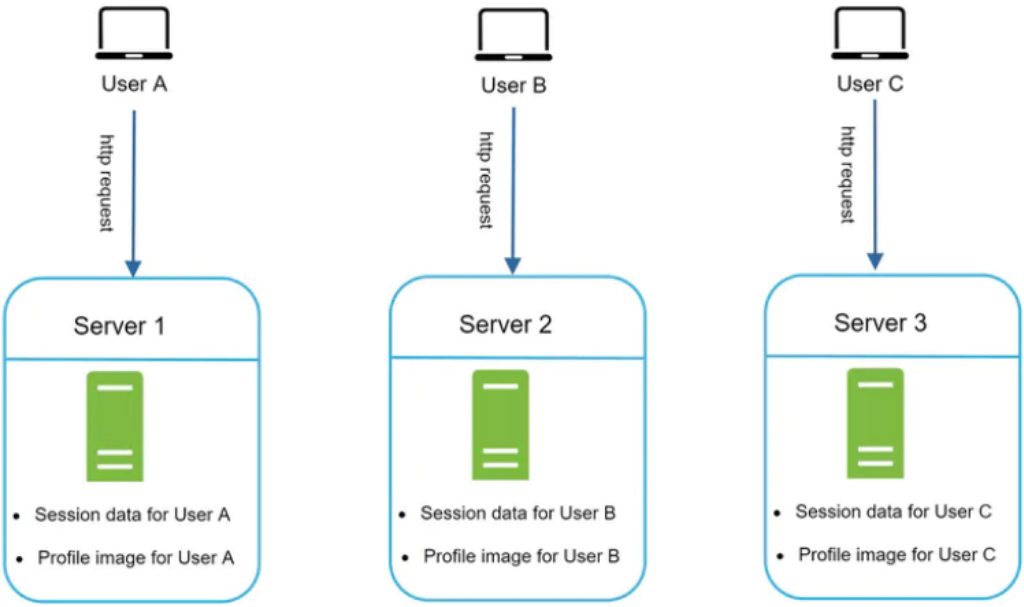
现在，我们的系统加了两层缓存。



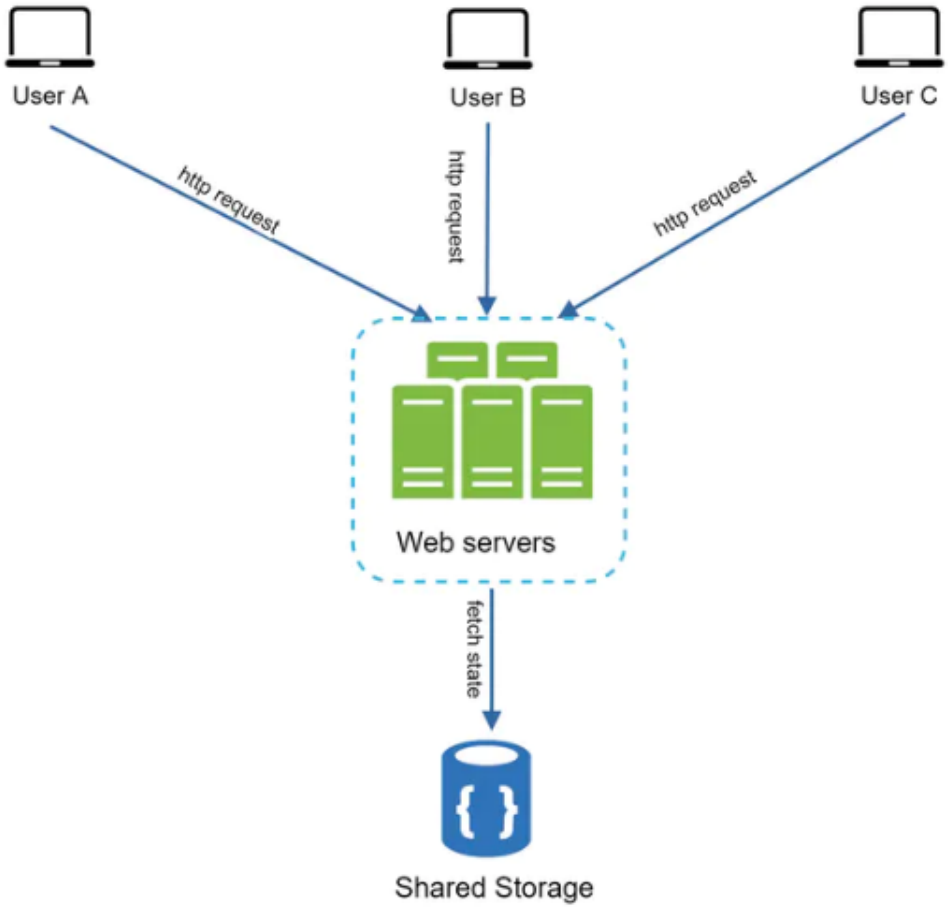
1. 对于静态资源，由 CDN 提供而不是 Web 服务器。
2. 通过缓存数据来减少对数据库的访问。

无状态 Web 层

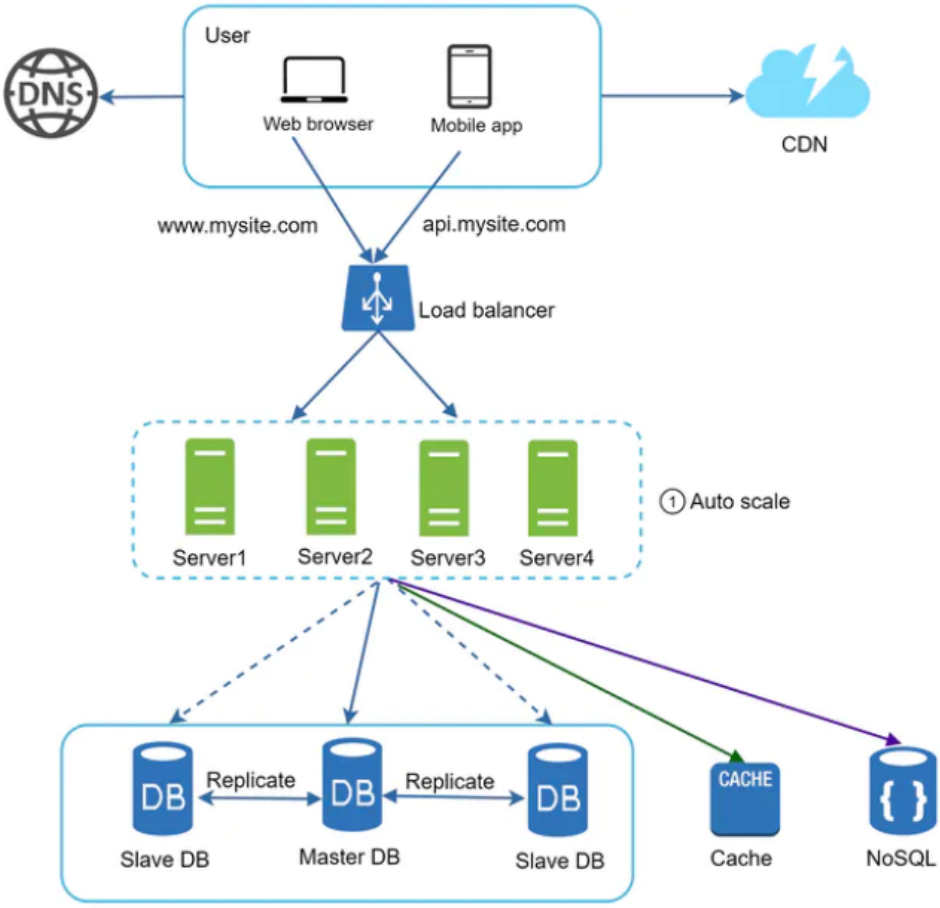
现在我们的 Web 应用是有状态的服务，什么意思呢？假如用户在 Server 1 进行了登陆, 那后续也只能在 Server1 请求资源，因为只有 Server1 才拥有用户的会话信息，每个 Web 服务的状态都是独立的、隔离的。



我们需要把这些状态移出 Web层，通常单独保存在关系型数据库或者 NoSQL, 这样 Web 层就变成了无状态的。



这样做有什么好处呢？在无状态的架构中，来自用户的 Http 请求可以发送到任何 Web 服务器，而状态信息统一保存在单独的共享存储中。无状态系统更简单、更容易扩展。

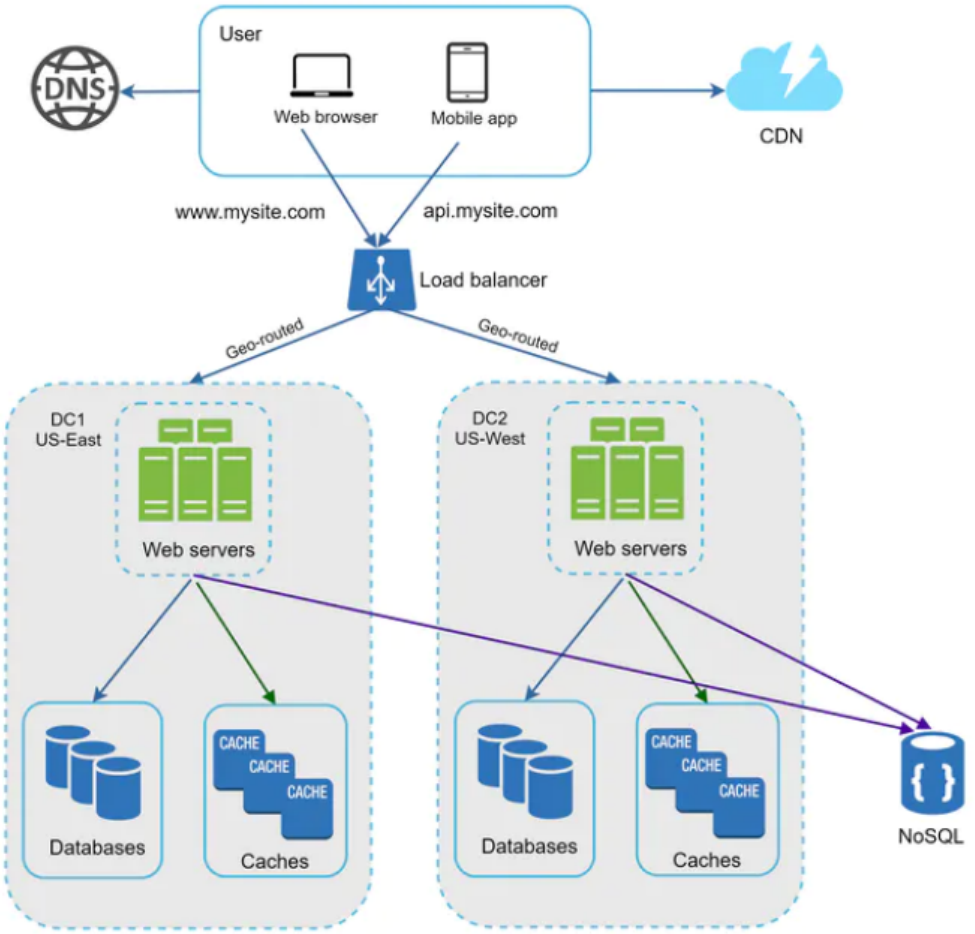


数据中心

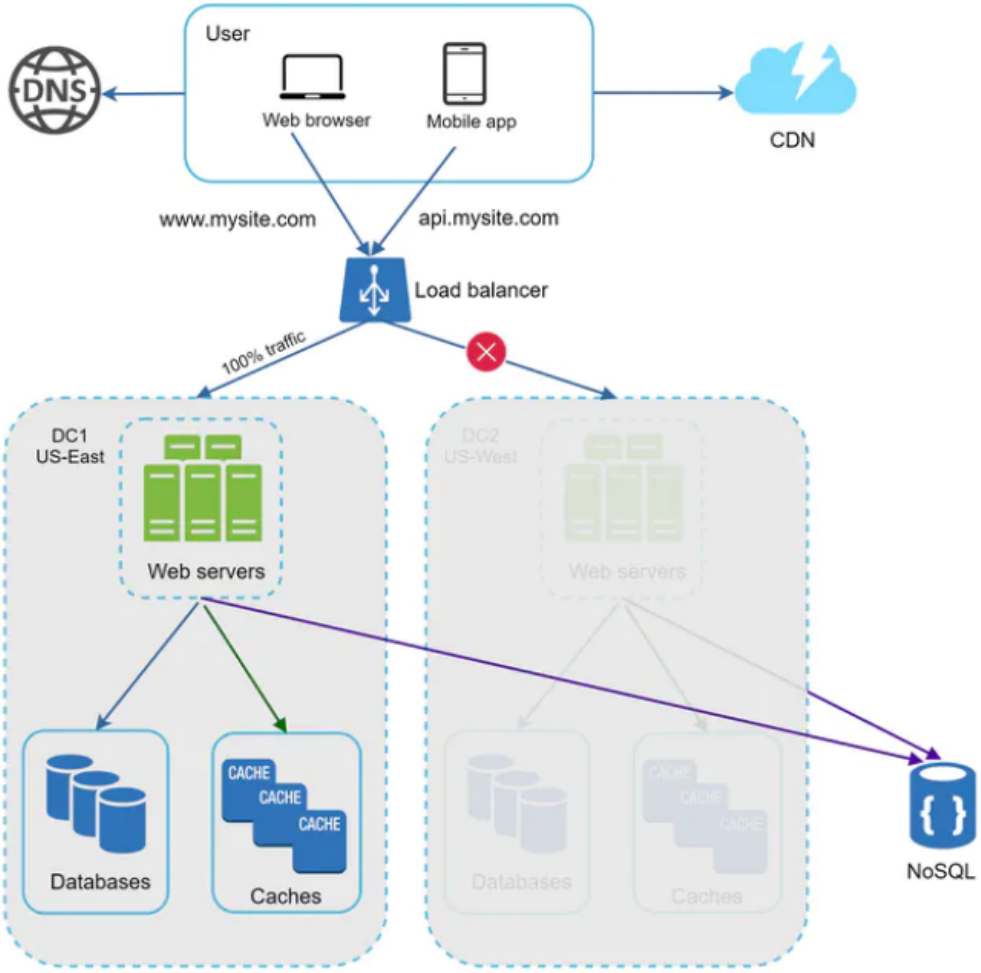
您的网站受到越来越多人的关注，用户也迅速发展，并扩展到全球。

如何为各个地区的用户都提供满意的服务？您可以在不同的地区设置多个数据中心。

如下图，我们分别在东、西两个地区配置了单独的数据中心, DC1、DC2。



看上去不错！但是如何引导用户去不同的数据中心呢？答案是：DNS, 是的，众所周知，DNS 可以把我们网站的域名解析为 IP 地址，而使用 GeoDNS, 可以根据用户请求所在的位置，解析为不同的地区的 IP 地址。把用户引导到离他最近的数据中心，来达到加速的目的。



另外，如果某个数据中心发生重大事故，导致集群故障，我们可以把所有的流量都引导到健康的数据中心，这种架构就是我们常说的 "异地多活"。

Message queue

当需要进行解耦时，引入消息队列通常是优先考虑的, 它支持异步通信，当您有耗时的任务需要处理时，可以通过生产者把消息发送到消息队列，Web 服务可以尽快的响应用户的请求，而消费者可以异步地去处理这些耗时任务。

日志、指标、自动化

当网站的流量越来越大时，就必须要引入监控工具了。

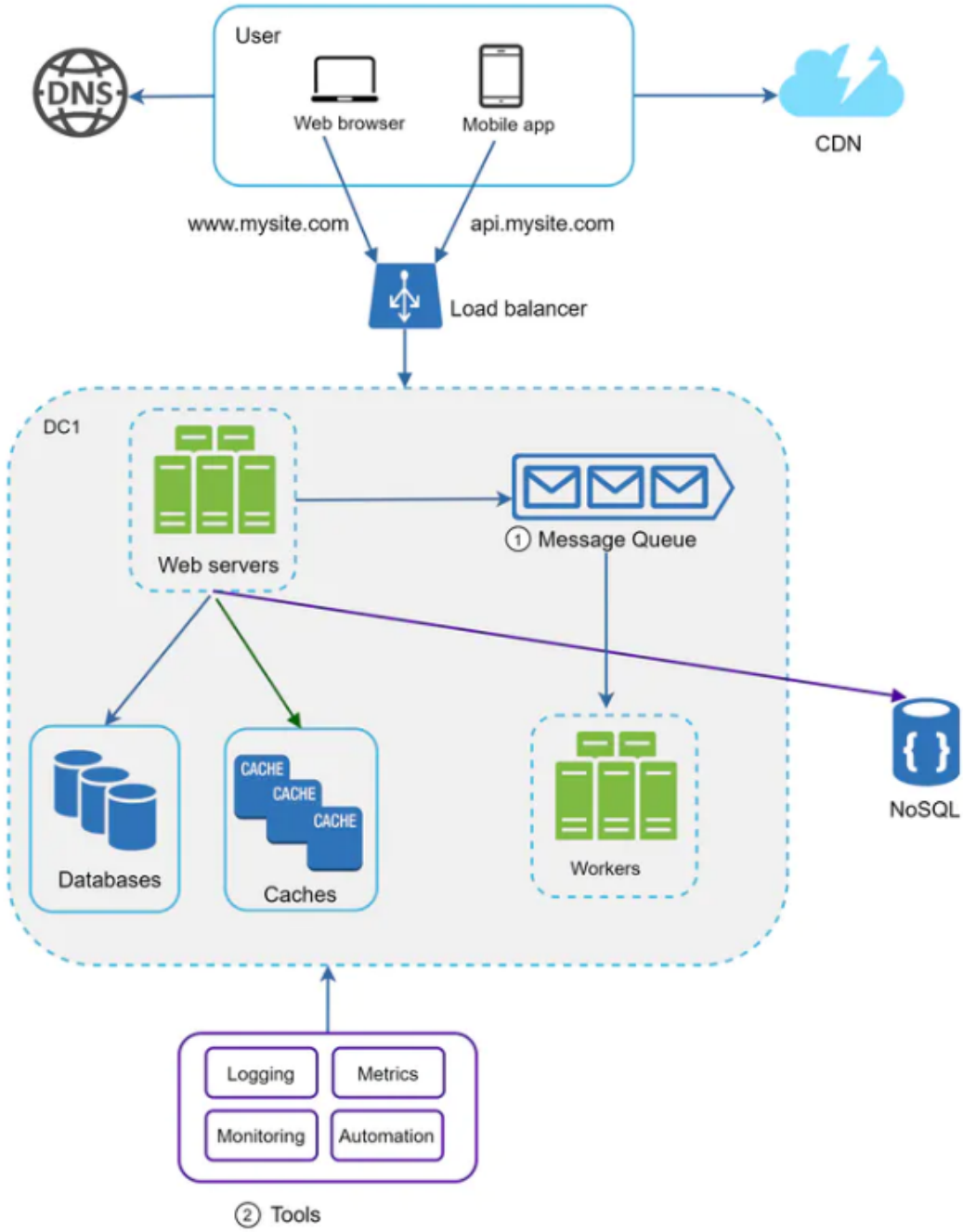
日志：监控错误日志很重要，它可以帮助您发现系统问题。 您可以把日志统一发送到日志中心，这样便于分析和查看。

指标：收集各种各样的指标，可以帮助我们更好的理解业务和系统。

- 系统指标:：CPU、内存、磁盘 I/O，数据库等等。
- 业务指标：每日用户、活跃度等等。

自动化，当系统变得庞大且复杂时，我们需要引入自动化工具，CI/CD 很重要，自动化构建、测试、部署可以极大的提高开发人员的生产力。

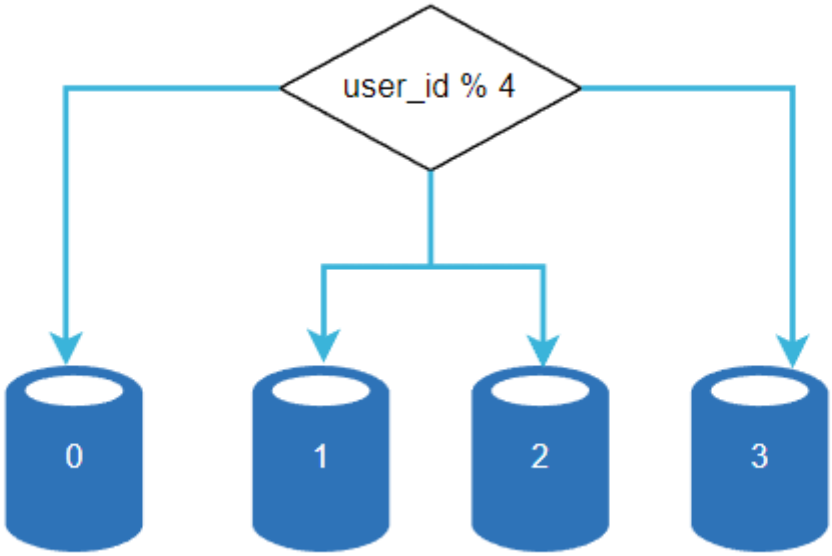
现在，我们的系统引入了消息队列，以及一些监控和自动化工具。



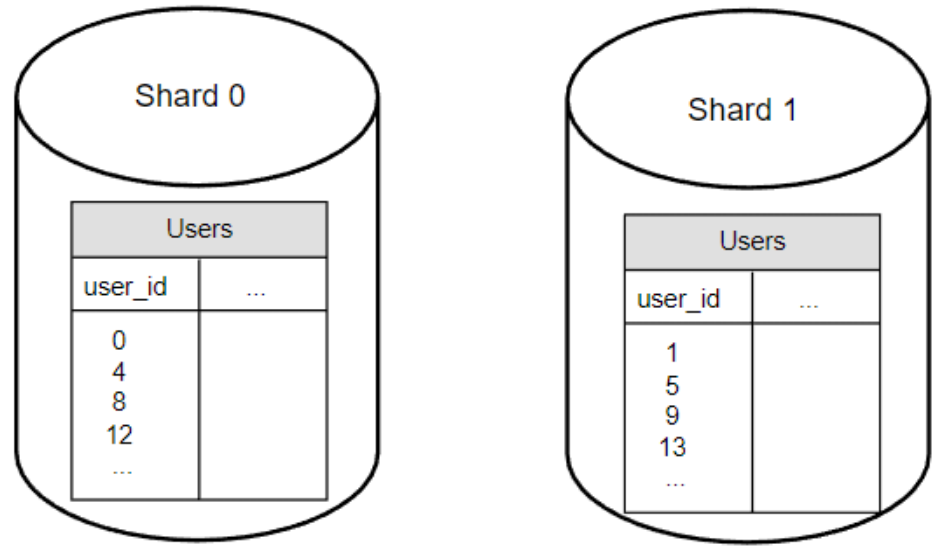
Database Sharding

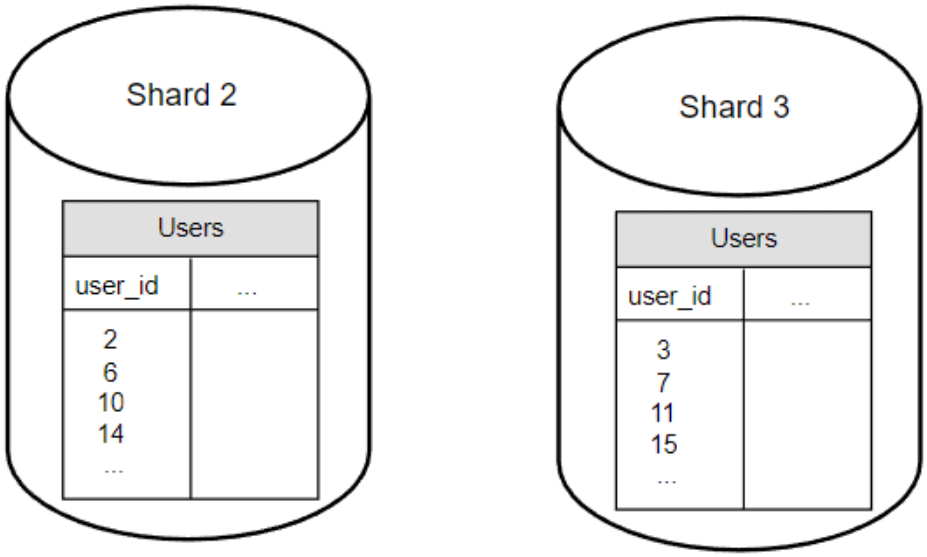
数据库的数据每天都在大步的增长，我们的数据库已经不堪重负了，是时候扩展数据库了，数据库分片是个很好的方案。

在下面的示例中，我们使用了哈希函数来进行分片, 根据不同的 user_id, 把数据平均分配到 4个数据库中。



现在，我们看一下数据库的数据。



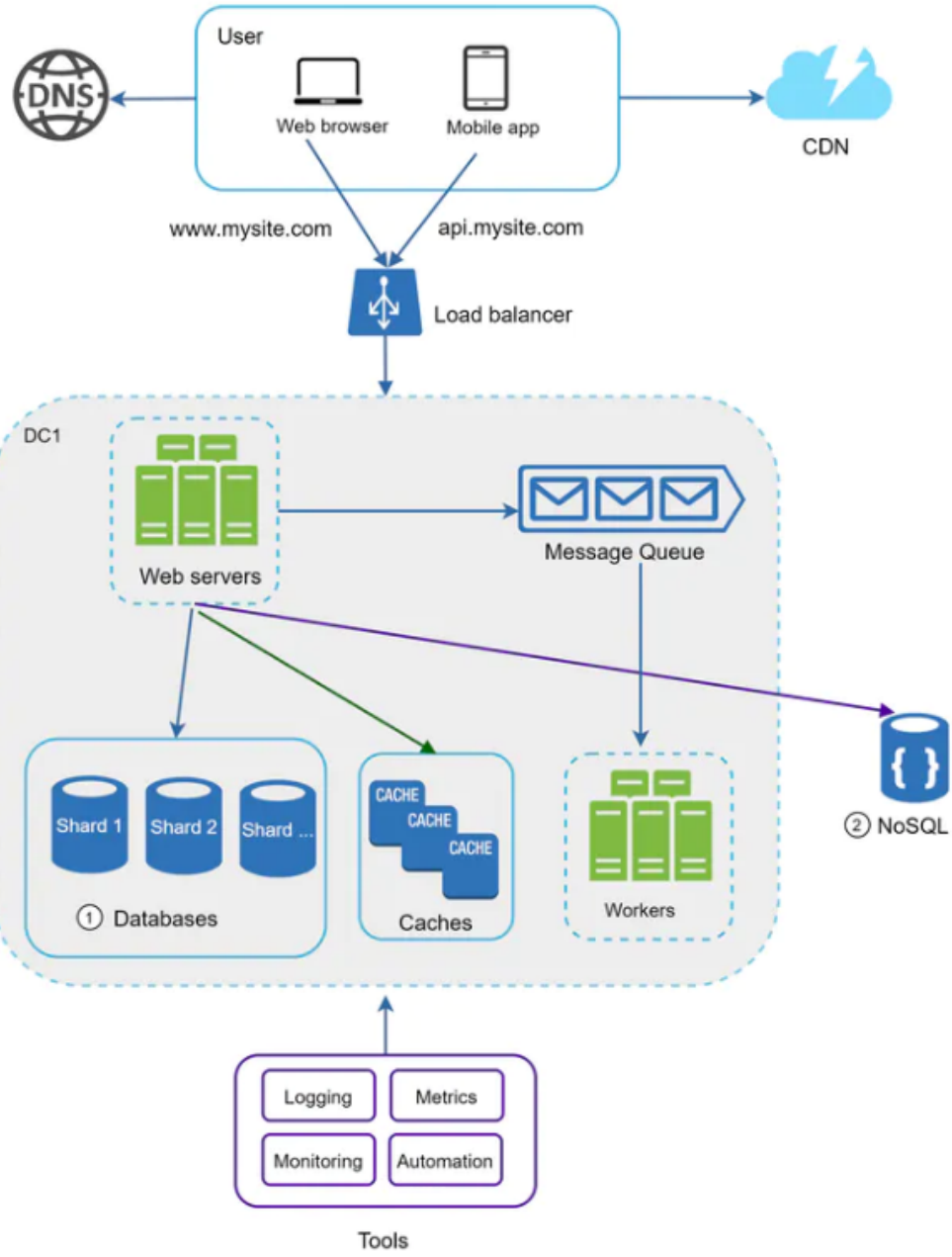


使用数据库分片的方案时，有一个要考虑的重要因素是分片键（sharding key），或者叫分区键，比如上面的 user_id，因为可以通过 sharding key 找到相对应的数据库，另外，我们要选择一个可以均匀分布数据的键。

看起来不错！不过这种方案也给系统带来的复杂性和新的挑战，当数据越来越多，增加了数据库节点之后，我们需要重新进行数据分片。比如 useri_id % 5，此时，为了保证哈希函数的正确路由，我们需要移动数据库大量的数据。

我们可以使用一致性哈希技术，来解决上面的问题，重新分片后，只需要移动一小部分数据即可，当然一致性哈希本文就不做详细的介绍了。

让我们看看最终的系统设计。



总结

构建一个健壮的架构系统，其实是一个迭代的过程，为了支持数百万的用户的架构，我们需要做到以下几点:

- 保证 Web 层无状态
- 尽可能的缓存数据
- 异地多活，配置多个数据中心
- 使用分片扩展数据库
- 监控系统并使用自动化工具

2.设计一个限流组件

限速器 (Rate Limiter) 相信大家都不会陌生，在网络系统中，限速器可以控制客户端发送流量的速度，比如 TCP, QUIC 等协议。而在 HTTP 的世界中， 限速器可以限制客户端在一段时间内发送请求的次数，如果超过设定的阈值，多余的请求就会被丢弃。

生活中也有很多这样的例子，比如

- 用户一分钟最多能发 5 条微博
- 用户一天最多能投 3 次票
- 用户一小时登录超过5次后，需要等待一段时间才能重试。

限速器（Rate Limiter）有很多好处，可以防止一定程度的 Dos 攻击，也可以屏蔽掉一些网络爬虫的请求，避免系统资源被耗尽，导致服务不可用。

设计要求

让我们从一个面试开始吧！

面试官：你好，我想考察一下你的设计能力，如果让你设计一个限速器 (Rate Limiter)，你会怎么做？

面试者： 我们需要什么样的限速器？ 它是在客户端限速还是在服务端限速？

面试官：这个问题很好，没有固定要求，取决于你的设计。

面试者：我想了解一下限速的规则，我们是根据 IP、UserId，或者手机号码进行限速吗？

面试官：这个不固定，限速器应该是灵活的，要能很方便的支持不同的规则。

面试者：如果请求被限制了，需要提示给用户吗？

面试官：需要提示，要给用户提供良好的体验。

面试者：好吧，那系统的规模是多大的？是单机还是分布式场景？

面试官：我们是 TOC 的产品， 系统流量很大，并且是分布式的环境， 所以限速器要支持海量请求。

面试者：（小声嘀咕）你这是造火箭呢？

我们总结一下限速器的设计要求：

- 低延迟，性能要好
- 需要适用于分布式场景。
- 用户的请求受到限制时，需要提示具体的原因。
- 高容错，如果限速器故障，不应该影响整个系统。

限速器应该放在哪里？

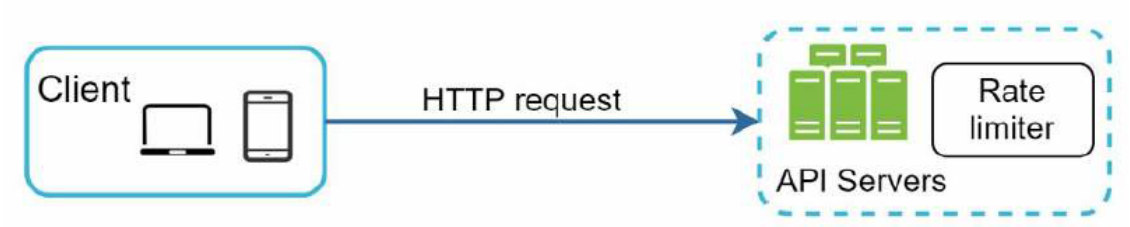
从系统整体的角度上来看，我们的限速器应该放在哪里？通常有三种选择，如下

客户端

是的，我们可以在客户端设置限速器。但是有个问题是，我们都知道在 Web 前端做一些限制实际上是不安全的，同样客户端也是一样的， 限速客户端可以做，但是远远不够。

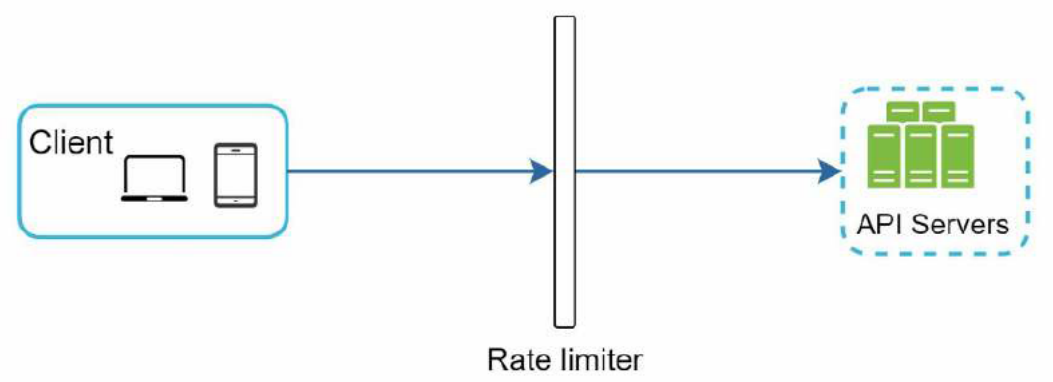
服务端

在服务端设置限速器是很常见且安全的，如下

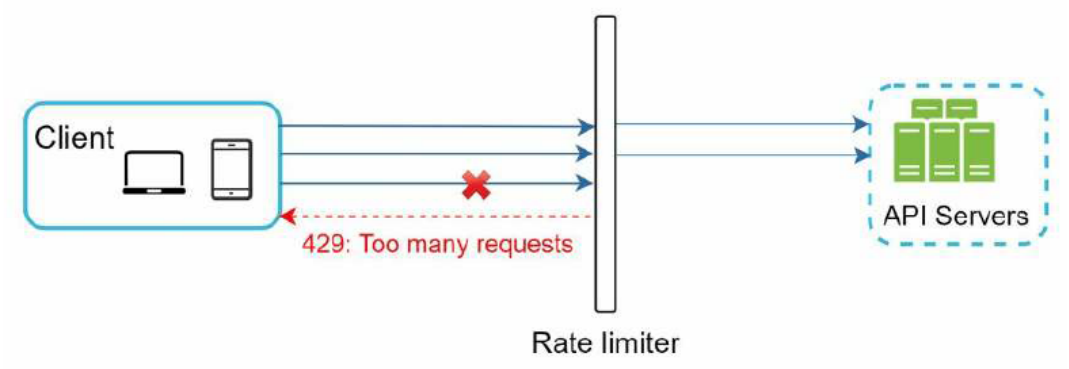


中间件

还有一种做法是，我们可以提供一个单独的限速中间件，如下



假如限速器设置了每秒最大允许2个请求，那么客户端发出的多余的请求就会被拒绝，并返回 HTTP 状态码 429, 表示用户发送了太多的请求。



实际上，很多网关都有限速的实现，包括认证、IP 白名单功能。

限速器应该放在哪里？没有固定的答案，它取决于公司的技术栈，系统规模。

限速算法

实际上，我们可以用算法实现限速器，下面是几种经典的限速算法，每种算法都有自己的优点和缺点，了解这些可以帮助我们选择更适合的算法。

- 令牌桶 （Token bucket）
- 漏桶（Leaking bucket）
- 固定窗口计数器（Fixed window counter）
- 滑动窗口日志（Sliding window log）
- 滑动窗口计数器（Sliding window counter）

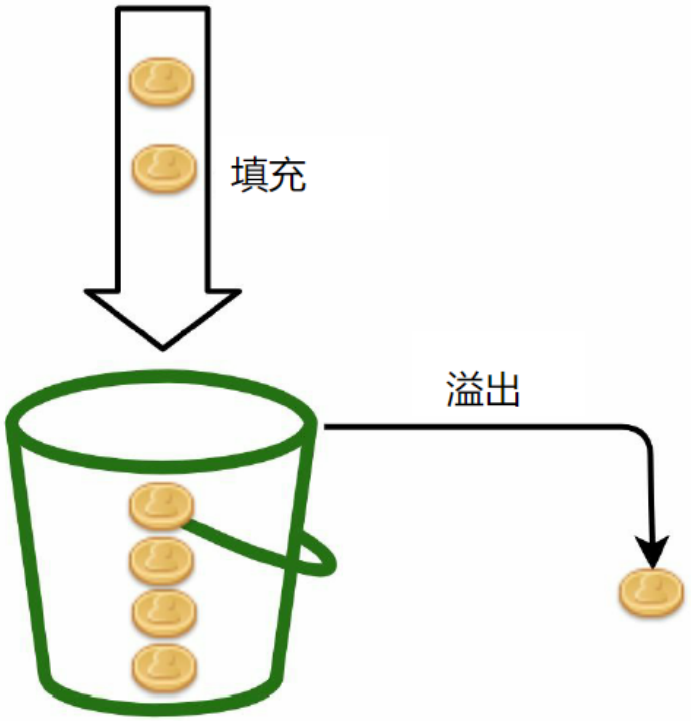
令牌桶算法

令牌桶算法是实现限速使用很广泛的算法，它很简单也很好理解。

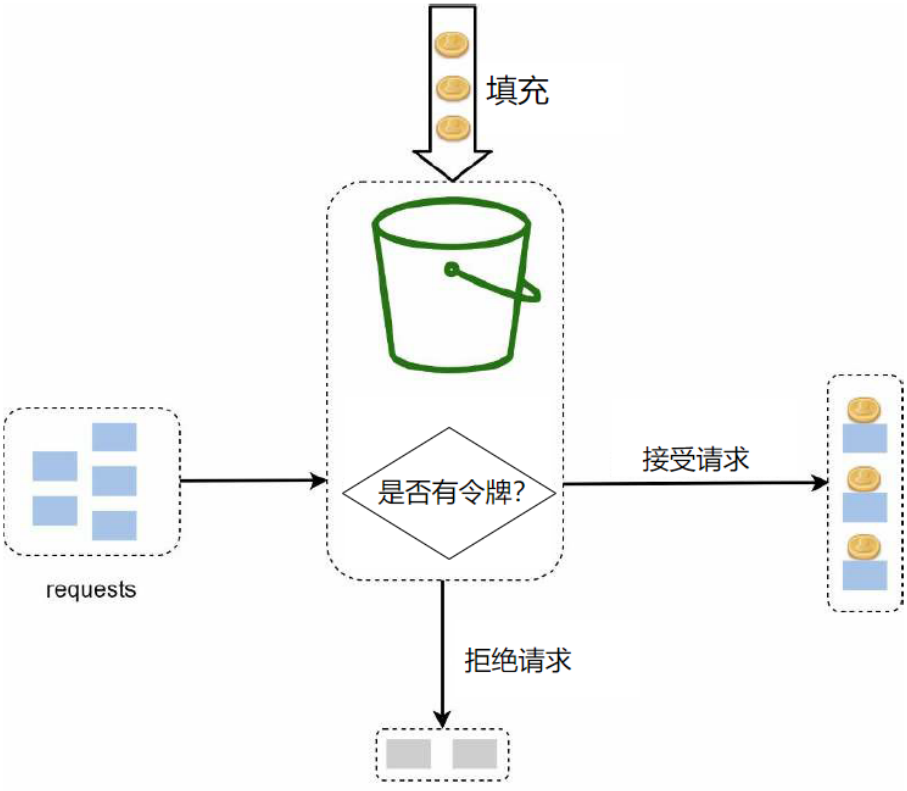
令牌桶是固定容量的容器。

一方面，按照一定的速率，向桶中添加令牌，桶装满后，多余的令牌就会被丢弃。

如下图，桶的容量为4，每次填充2个令牌。



另一方面，一个请求消耗一个令牌，如果桶中没有令牌了，则拒绝请求。直到下个时间段，继续向桶中填充新的令牌。



令牌桶算法有两个重要的参数，分别是桶大小和填充率，另外有时候可能需要多个桶，比如多个 api 限速的规则是不一样的。

令牌桶算法的优点是简单易实现，并且允许短时间的流量并发。缺点是，在应对流量变化时，正确地调整桶大小和填充率，会比较有挑战性。

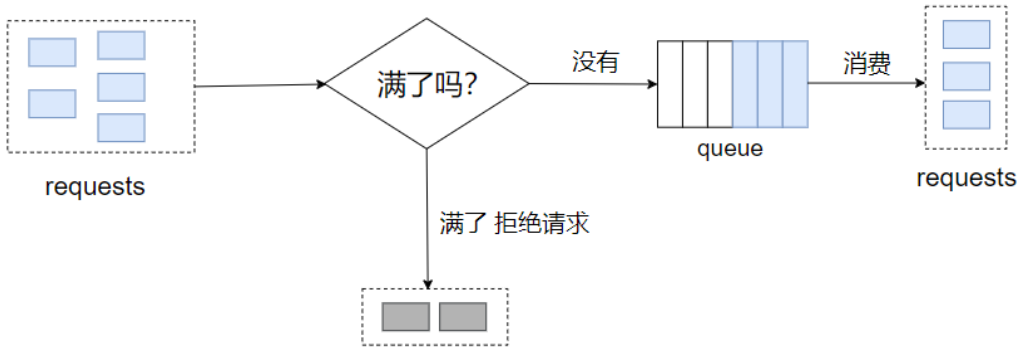
漏桶算法

漏桶算法和令牌桶算法是类似的，同样有一个固定容量的桶。

一方面，当一个请求进来时，会被填充到桶里，如果桶满了，就拒绝这个请求。

另一方面，想象桶下面有一个漏洞，桶里的元素以固定的速率流出。

通常可以用先入先出的队列实现，如下图

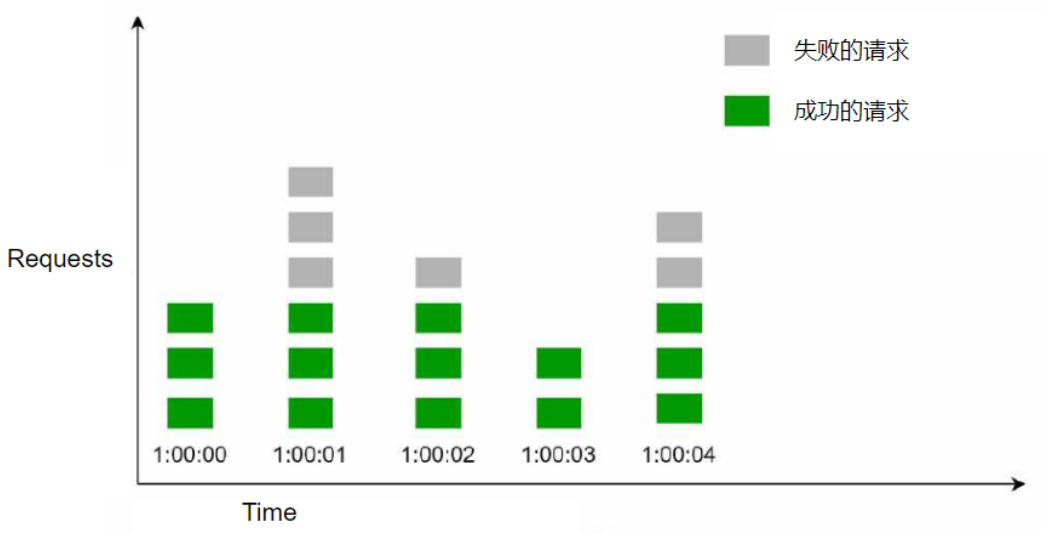


漏桶算法有两个参数，分别是桶大小和流出率，优点是使用队列易实现，缺点是，面对突发流量时，虽然有的请求已经推到队列中了，但是由于消费的速率是固定的，存在效率问题。

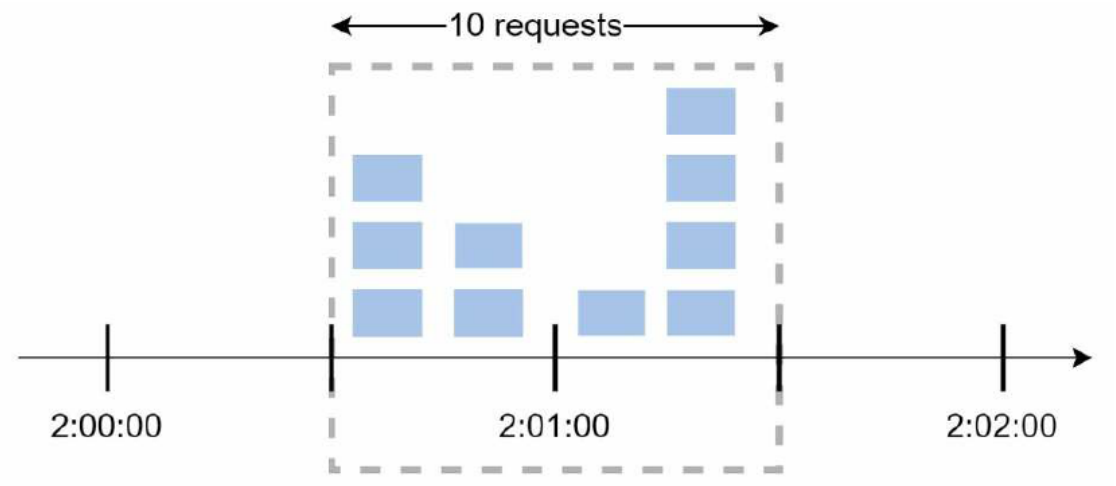
固定窗口计数器算法

固定窗口计数器算法的工作原理是，把时间划分成固定大小的时间窗口，每个窗口分配一个计数器，接收到一个请求，计数器就加一，一旦计数器达到设定的阈值，新的请求就会被丢弃，直到新的时间窗口，新的计数器。

让我们通过下面的例子，来看看它是如何工作的。一个时间窗口是1秒，每秒最多允许3个请求，超出的请求就会被丢弃。



不过这个算法有一个问题是，如果在时间窗口的边缘出现突发流量时，可能会导致通过的请求数超过阈值，什么意思呢？我们看看下面的情况



一个时间窗口是1分钟，每分钟最多允许5个请求。如果前一个时间窗口的后半段有5个请求，后一个时间窗口的前半段有5个请求，也就是 2:00:30 到 2:01:30 的一分钟内，是可以通过10个请求的，这明显超过了我们设置的阈值。

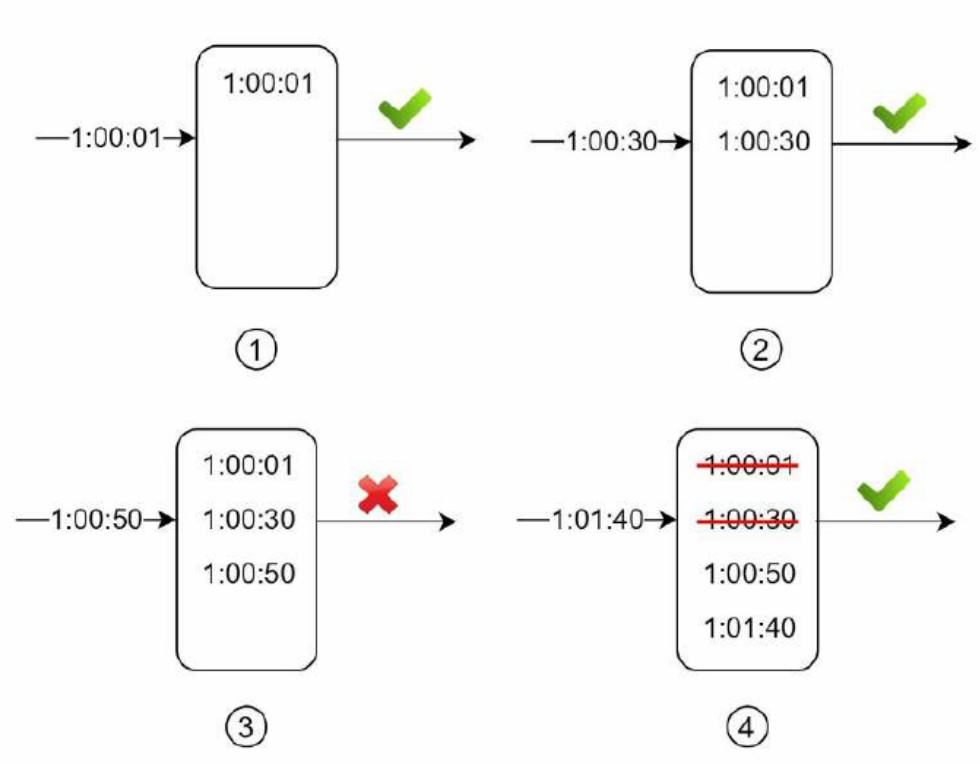
固定窗口计数器的优点是，简单易于理解，缺点是，时间窗口的边缘应对流量高峰时，可能会让通过的请求数超过阈值。

滑动窗口日志算法

我们上面看到了，固定窗口计数器算法有一个问题，在窗口边缘可能会突破限制，而滑动窗口日志算法可以解决这个问题。

它的工作原理是，假如设定1分钟内最多允许2个请求，每个请求都需要记录请求时间，比如保存在 Redis 的 sorted sets 中，保存之后还需要删除掉过时的日志，过时日志是如何定义的？比如某次请求的时间是 1:01:36，那么往前推1分钟，1:00:36 之前的日志都算过时的，需要从集合中删掉。同时，判断集合中的数量是否大于阈值，如果大于2则丢弃请求，如果小于或等于2，则处理这个请求。

让我们看看下面的例子

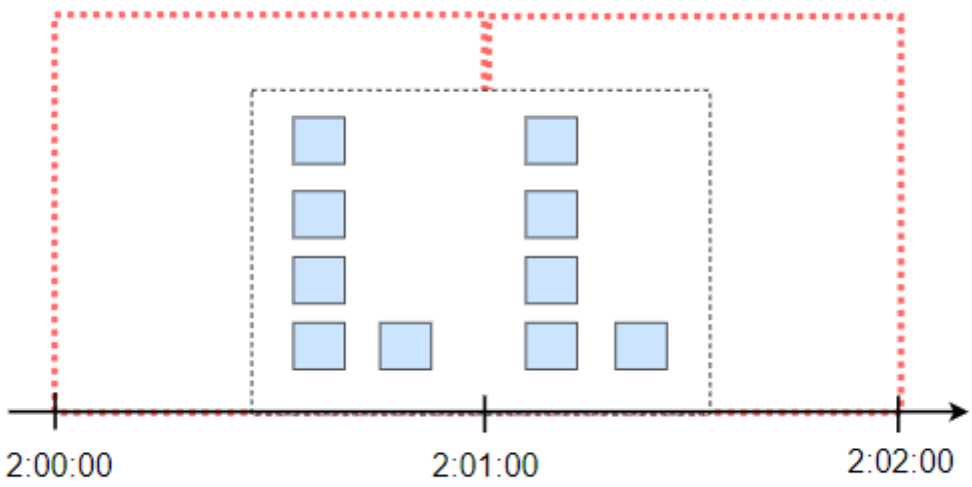


1. 在 1:00:01 来了一个请求，第一步，记录请求时间到日志中，第二步，判断是否有过时的日志, 也就是 0:59:01 之前的日志，明显没有，第三步判断日志中的数量，没有大于2，处理这次请求。
2. 在 1:00:30 来了一个请求，执行上面的三个步骤，最后处理这次请求。
3. 在 1:00:50 的新请求，没有过时的日志，然后发现日志的数量为3，拒绝这次请求。
4. 在 1:01:40 的新请求，清除2条过时的日志，也就是 1:00:40 之前的日志，此时，日志中的数量为2，处理这次请求。

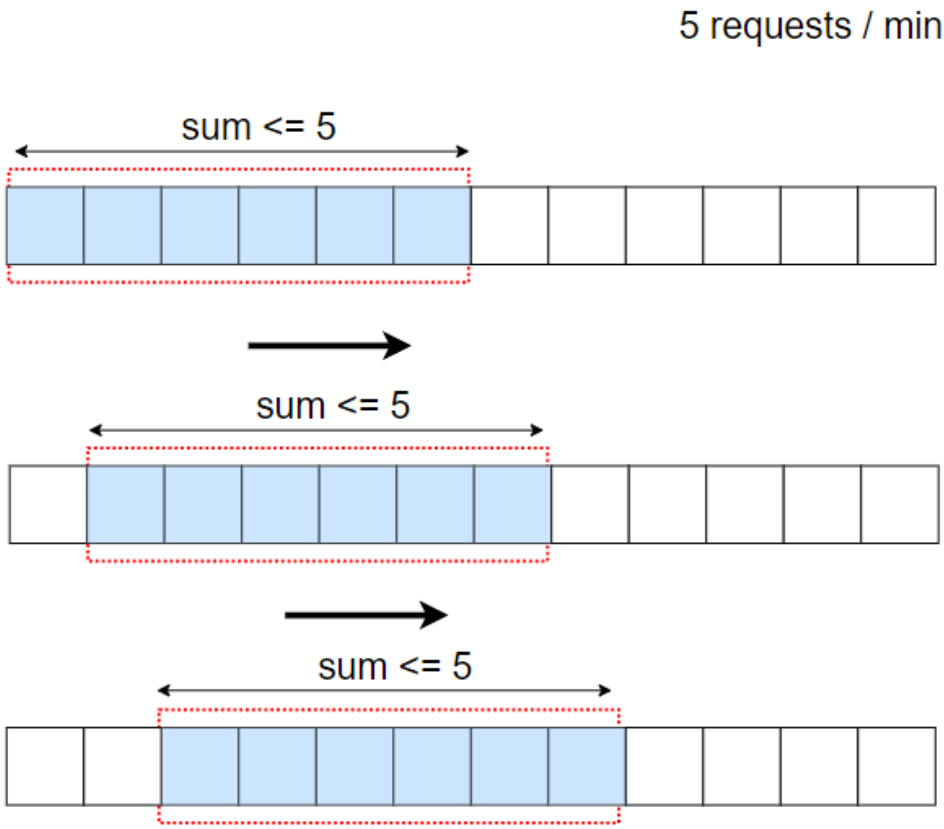
这个算法实现的限速非常准确，但是它可能会消耗较多的内存。

滑动窗口计数器算法

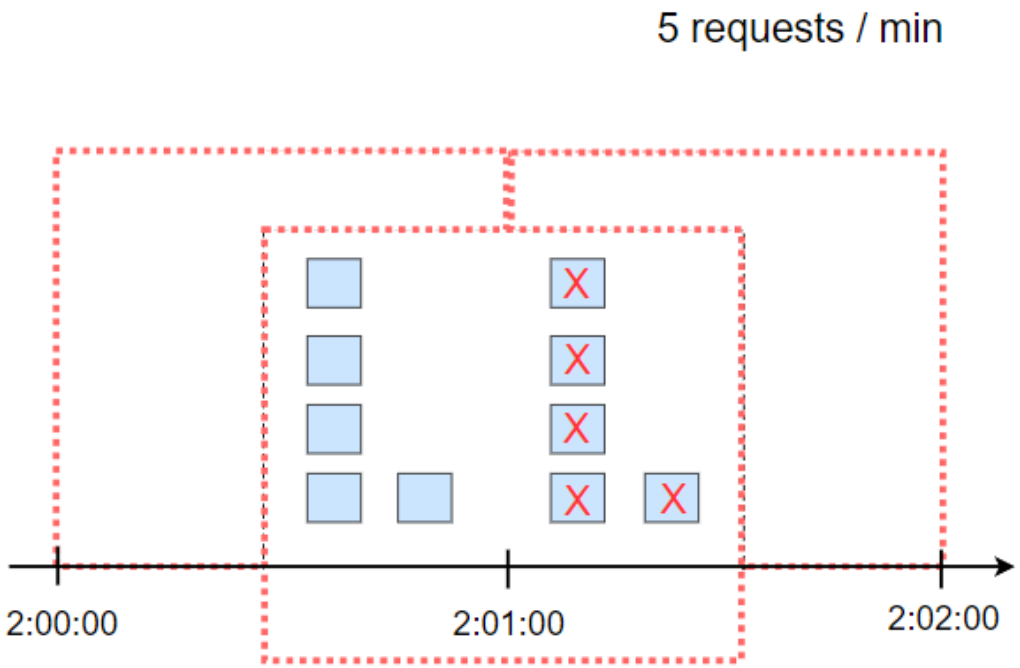
滑动窗口计数器可以说是固定窗口计数器的升级版，上面提到了，固定窗口计数器存在窗口边缘可能会有超出限制的情况，如下



而滑动窗口把固定的窗口又分成了很多小的窗口单位，比如下图，每个固定窗口的大小为1分钟，又拆分成了6份，每次移动一个小的单位，保证总和不超过阈值。



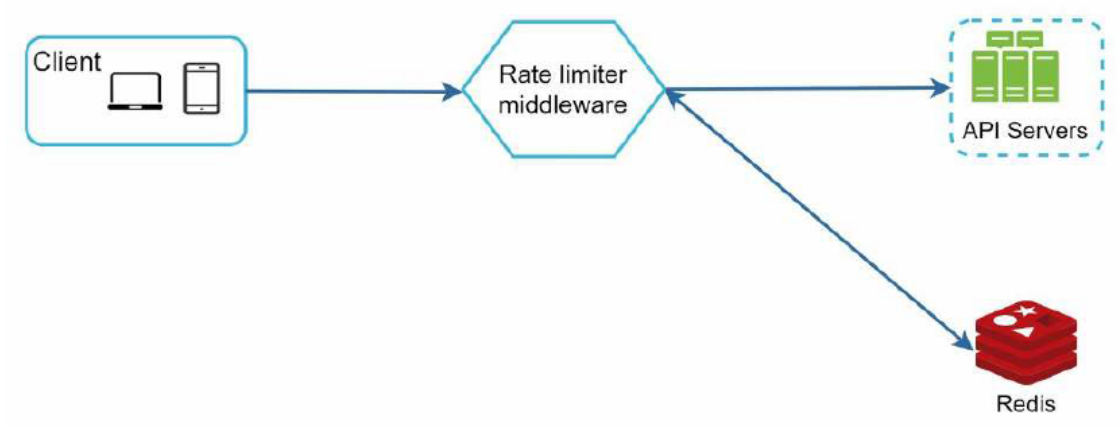
这样就可以避免上面的窗口边缘超出限制的问题。



使用 Redis 实现高效计数器

限速器算法的思想其实很简单，我们需要使用计数器记录用户的请求，如果超过阈值，服务这个请求，否则，拒绝这个请求。

一个很重要的问题是，我们应该把计数器放在哪里？我们知道，磁盘速度比较慢，使用数据库明显是不太现实的方案，想要更快的话，可以使用内存缓存，最常见的就是 Redis，是的，我们可以使用 Redis 实现高效计数器，如下



规则引擎

Lyft 是一个开源的限速组件，可以供我们参考，它通过 Yaml 配置文件实现灵活的限速规则，看下面的示例

```
domain: messaging
descriptors:
  - key: message_type
    Value: marketing
    rate_limit:
      unit: day
      requests_per_unit: 5
```

这个配置表示系统每天只能发送 5 条营销信息。

```yaml
domain: auth
descriptors:
  - key: auth_type
    Value: Login
    rate_limit:
      unit: minute
      requests_per_unit: 5
```

这个配置表示 1分钟的登录次数不能超过 5 次。

可以看到，基于配置文件，声明式的限速规则是非常灵活的，我们可以把配置文件保存到磁盘中。

返回限速信息

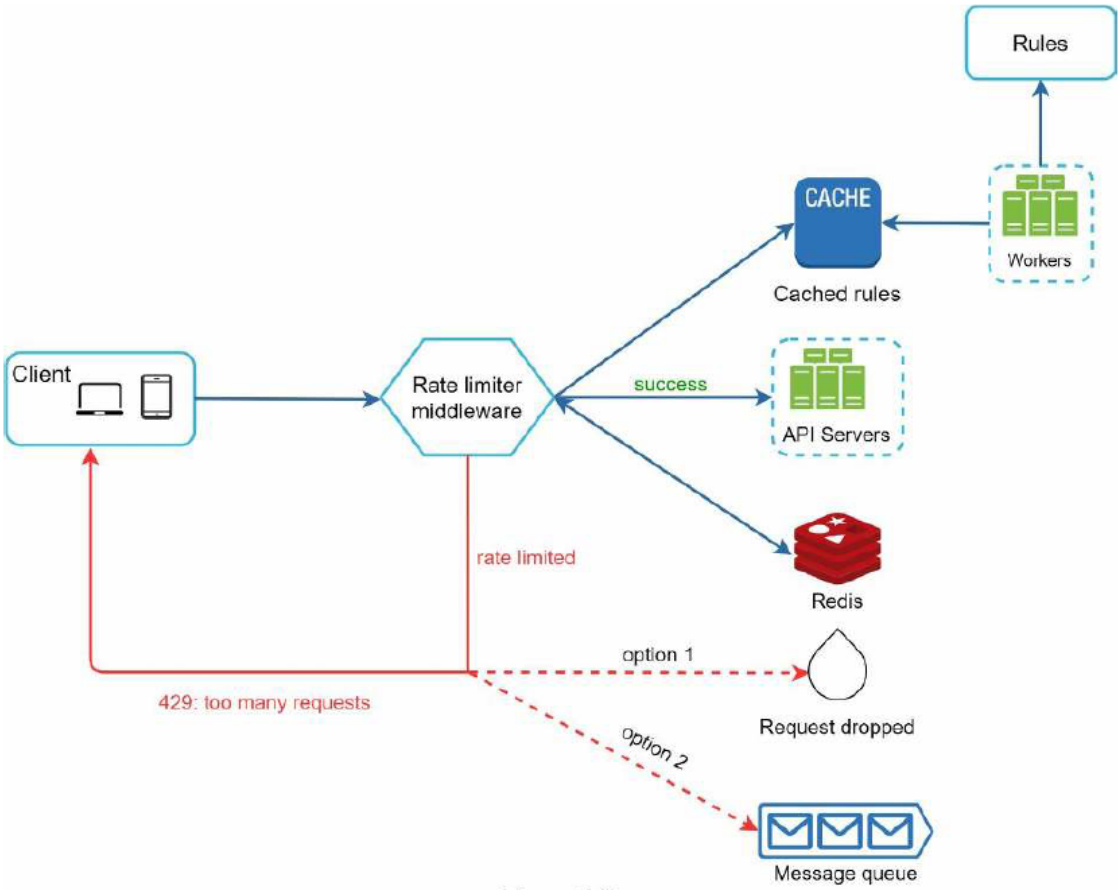
当请求超过限制时，限速器会拒绝掉其他的请求，这样其实不够，为了更好的用户体验，我们需要返回友好的错误信息给用户，并提示。

首先，限速器拒绝请求后，可以返回 HTTP 状态码 429，表示请求过多。

其次，我们可以返回更详细的信息，比如，剩余请求次数、等待时间等。一种很常见的做法时，把这些信息放到 Http 响应的 Header 中返回，如下

- X-Ratelimit-Remaining：表示剩余次数
- X-Ratelimit-Limit：表示客户端每个时间窗口可以进行多少次调用
- X-Ratelimit-Retry-After：表示等待多长时间可以进行重试

看起来不错！让我们看看现在的架构设计



首先，限速规则存储在磁盘上，因为要经常访问，可以添加到缓存中。当客户端向服务器发送请求时，会先发送到限速中间件。限速中间件从缓存中拉取限速规则，同时把请求数据写入到 Redis 的计数器，然后判断是否超出限制。如果没有超出限制，把请求转发给我们的后端服务器。如果超出了限制，第一种方案，丢弃多余的请求，返回 429，第二种方案，把多余的请求推送到消息队列中，后续再进行处理。使用哪种方案，取决于您的实际场景。

分布式环境

构建一个在单服务器运行的限速器是很简单的，但是在分布式环境中，支持多台服务器，那就完全是另外一回事了。

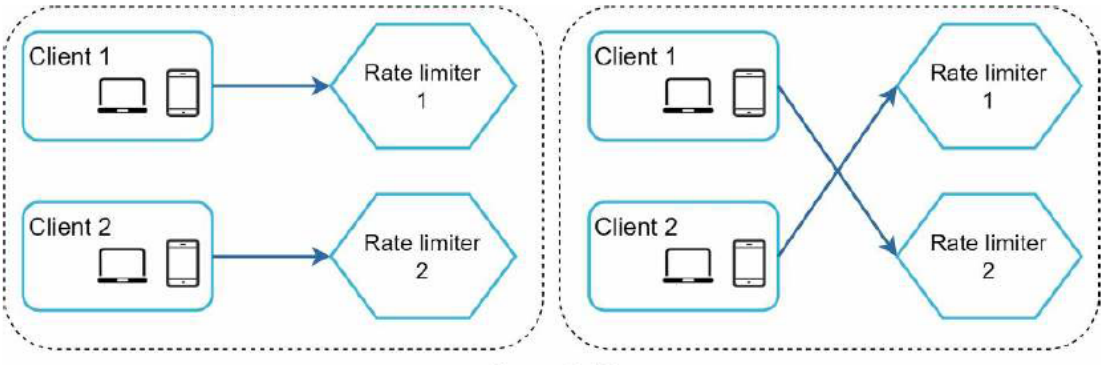
我们主要要考虑两个问题：

- 并发问题
- 数据同步问题

并发问题，我们的限速器的工作原理是，当接收到新的请求时，从 Redis 中读取计数器 counter，然后做加一的操作，在高并发场景中，可能存在多个线程读到了旧的值，比如，两个线程同时读取到 counter 的值为3，然后都把 counter 改成了4，这样是有问题的，其实最终应该是 5。

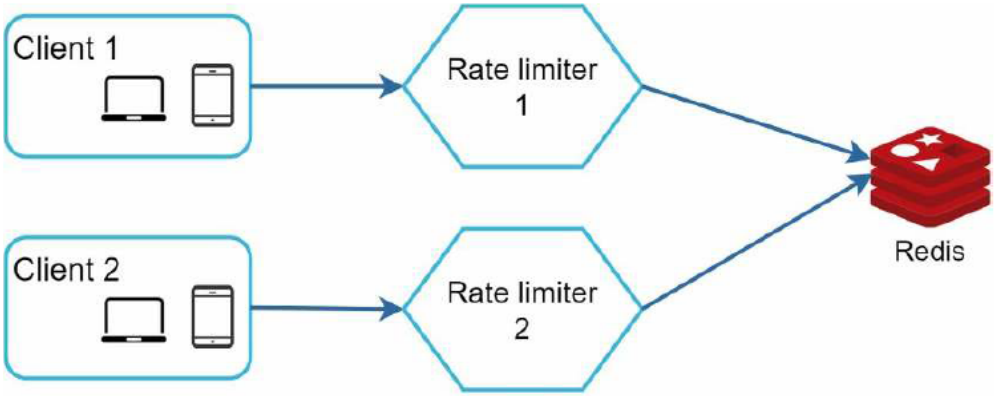
有朋友说，我们可以用锁，但实际上，锁的效率是不高的。解决这个问题通常有两个方案，第一个是使用 Lua 脚本，第二个是使用 Redis 的 sorted sets 数据结构，具体的细节本文不做过多介绍。

数据同步问题，在流量比较大的情况下，一个限速器是难以支撑的，我们需要多个限速器。由于 Web 层通常是无状态的，客户端的请求会随机发送给不同的限速器，如下



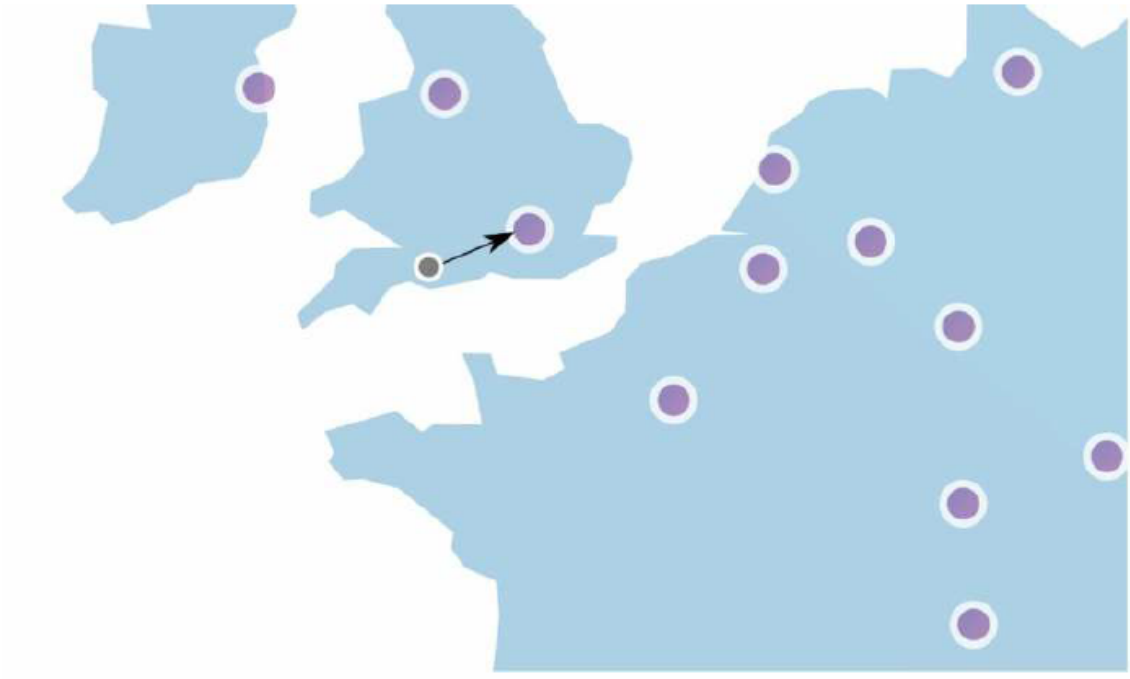
这种情况下，如果没有数据同步，我们的限速器肯定是没办法正常工作的。

我们可以使用像 Redis 这样的集中式数据存储，如下



性能优化

当我们的系统是面向全球用户时，为了让各个地区的用户都能有一个不错的体验，通常会在不同的地区设置多个数据中心。另外，现在很多云服务商在全球各地都有边缘服务器，流量会自动路由到最近的边缘服务器，来减少网络的延迟。



当然，存在多个数据中心时，可能还要考虑到数据的最终一致性。

总结

在本文中，我们讨论了不同的限速算法，以及它们有优缺点，算法包括：

- 令牌桶
- 漏桶
- 固定窗口计数器
- 滑动窗口日志
- 滑动窗口计数器

然后，我们讨论了分布式环境中的系统架构，并发问题和数据同步问题，和灵活配置的限速规则，最后你会发现，实现一个限速器，其实没有那么难！

3.设计一个短链接系统

短链接系统可以把比较长的 URL 网址转换成简短的网址字符串，短链接的优势是方便传播。适合在一些对字符串长度有要求的场景中使用，比如短信，微博等，比如

https://www.cnblogs.com/myshowtime/p/16227260.html

转换成短链接为

https://bit.ly/3z0QtB9

设计要求

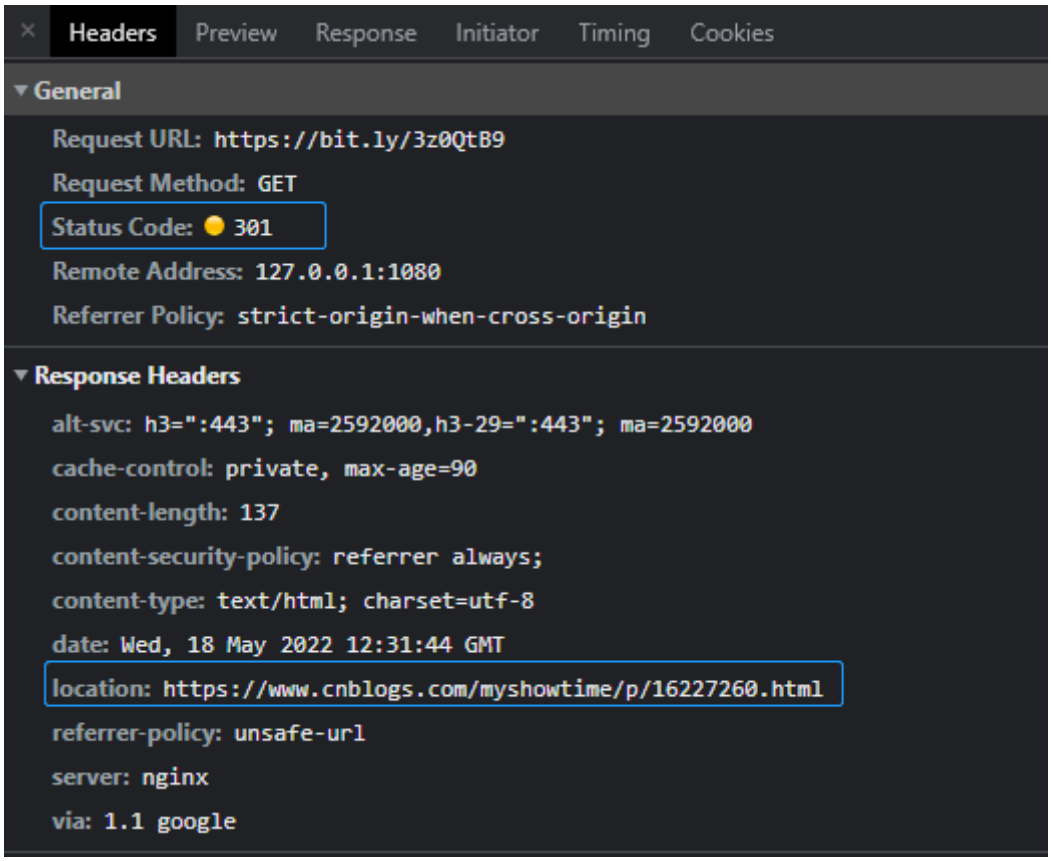
根据面试的要求，你需要设计一个短链接系统， 链接的长度尽量比较短，每天生成 1 亿个URL，服务要运行 10 年。

首先，我们看一下短链接的工作原理。

工作原理

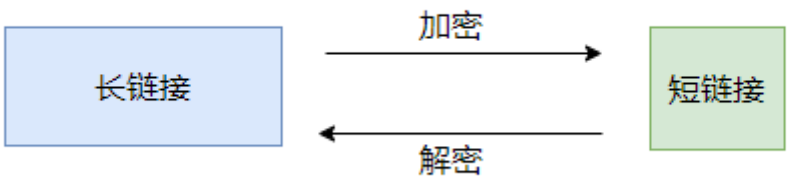
在 Chrome 上输入短链接，会发生什么？

打开开发者工具， 可以看到， 服务器收到请求后，会把短链接转换成长链接，然后返回浏览器，进行 301 重定向，请求到长链接地址。



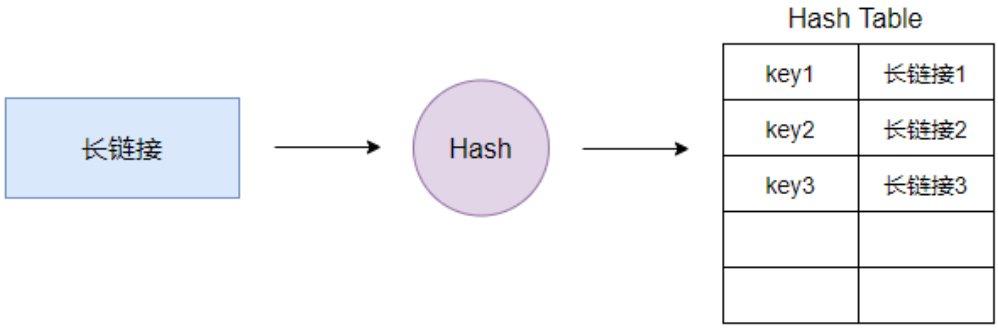
另外一个问题，如何把长链接转换成短链接？

能否使用一些加密算法呢？明显是行不通的，因为字符串加密后会变的更长。



哈希算法

实际上，我们可以使用哈希算法和哈希表实现，如下



长链接经过哈希算法后， 会生成固定长度的哈希值 key，也就是短链接的值，并保存到哈希表中。

使用短链接查询长链接时，只需要查询哈希表即可。

Hash function	Hash Value	Length
CRC32	0972d361	8
MD5	49ba59abbe56e057	16
SHA-1	7c4a8d09ca3762af61e59520943dc26494f8941b	40

上面是常见的哈希算法，最少也要8位。

那我们需要多少位的短链接呢？根据上面的要求，一天生成一个亿的短链接，运行10年，1亿 * 365 * 10 = 3650 亿。

短链接的字符在 [0-9,a-z,A-Z] 之间，总共 62 个不同的字符，可以计算出下面的数据。

N	Max
1	62 ^ 1 = 62
2	62 ^ 2 = 3,844
3	62 ^ 3 = 238,328
4	62 ^ 4 = 14,776,336
5	62 ^ 5 = 916,132,832
6	62 ^ 6 = 56,800,235,584 568 亿
7	62 ^ 7 = 3,521,614,606,208 35216 亿

可以看出，要满足系统要求的话，短链接的长度最少为 7 位。在实际中，很多短链接系统的长度也是 7 位。有兴趣的同学还可以看一下，米勒定律 7±2 法则。

上面的 CRC32 算法，最少也是 8 位。不过我们可以截取前 7 位，最后一位丢弃。但是这样可能会出现哈希冲突的问题，我们可以给长链接递归地拼接一个值，直到不再发现冲突，当然也可以用其他的哈希冲突解决方法。

Base 62 转换

这是另外一种常见的方法，Base 62 字符由大写字母 A-Z、小写字母 a-z 和数字 0-9 组成，总共 62 位，如下

Base62 table [edit]

The Base62 index table:

Decimal	Binary	Base62	Decimal	Binary	Base62	Decimal	Binary	Base62	Decimal	Binary	Base62
0	000000	0	16	010000	G	32	100000	W	48	110000	m
1	000001	1	17	010001	H	33	100001	X	49	110001	n
2	000010	2	18	010010	I	34	100010	Y	50	110010	o
3	000011	3	19	010011	J	35	100011	Z	51	110011	p
4	000100	4	20	010100	K	36	100100	a	52	110100	q
5	000101	5	21	010101	L	37	100101	b	53	110101	r
6	000110	6	22	010110	M	38	100110	c	54	110110	s
7	000111	7	23	010111	N	39	100111	d	55	110111	t
8	001000	8	24	011000	O	40	101000	e	56	111000	u
9	001001	9	25	011001	P	41	101001	f	57	111001	v
10	001010	A	26	011010	Q	42	101010	g	58	111010	w
11	001011	B	27	011011	R	43	101011	h	59	111011	x
12	001100	C	28	011100	S	44	101100	i	60	111100	y
13	001101	D	29	011101	T	45	101101	j	61	111101	z
14	001110	E	30	011110	U	46	101110	k			
15	001111	F	31	011111	V	47	101111	l			

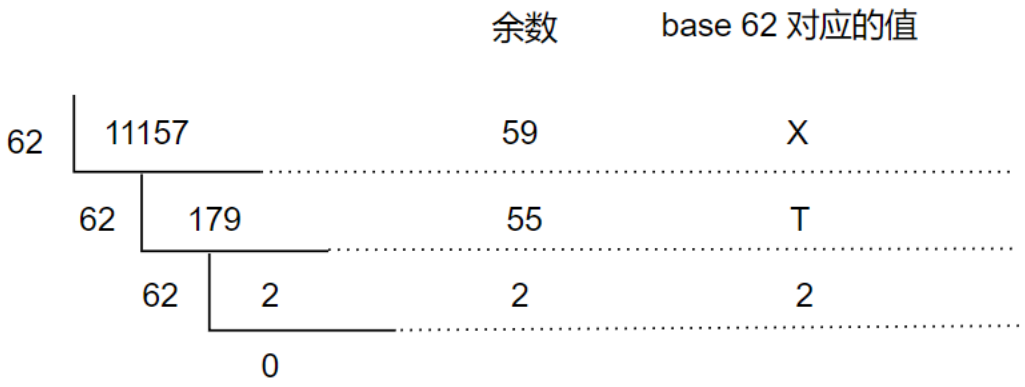
base 62 和 base 64 相比，只不过少了2个字符 + 和 /，大家可以想一下，这里我们为什么不用 base 64。

Base 62 和上面的哈希算法的思路是不一样的，哈希算法是根据长链接计算哈希值，然后保存到哈希表中。而 base 62 需要给每条长链接生成一个唯一的数字 ID，如下

Id	ShortURL	LongURL
1331		https://www.cnblogs.com/myshowtime/p/1634324.html
1332		https://www.cnblogs.com/myshowtime/p/6324324.html
1333		https://www.cnblogs.com/myshowtime/p/565654.html

那么如何计算短链接 ShortURL 呢? 因为 Id 是唯一的 10 进制数字，我们只需要把它转成 62 进制即可，这里和从2进制转换到10进制是一样的。

假如有一个 ID 为 11157，转换的过程如下



最终得到的短链接的值为 https://xxx.com/2TX。

Id	ShortURL	LongURL
1331	LT	https://www.cnblogs.com/myshowtime/p/1634324.html
1332	LU	https://www.cnblogs.com/myshowtime/p/6324324.html
1333	LV	https://www.cnblogs.com/myshowtime/p/565654.html
.....	.....	.....
11157	2TX	https://www.cnblogs.com/myshowtime/p/339435.html

总结

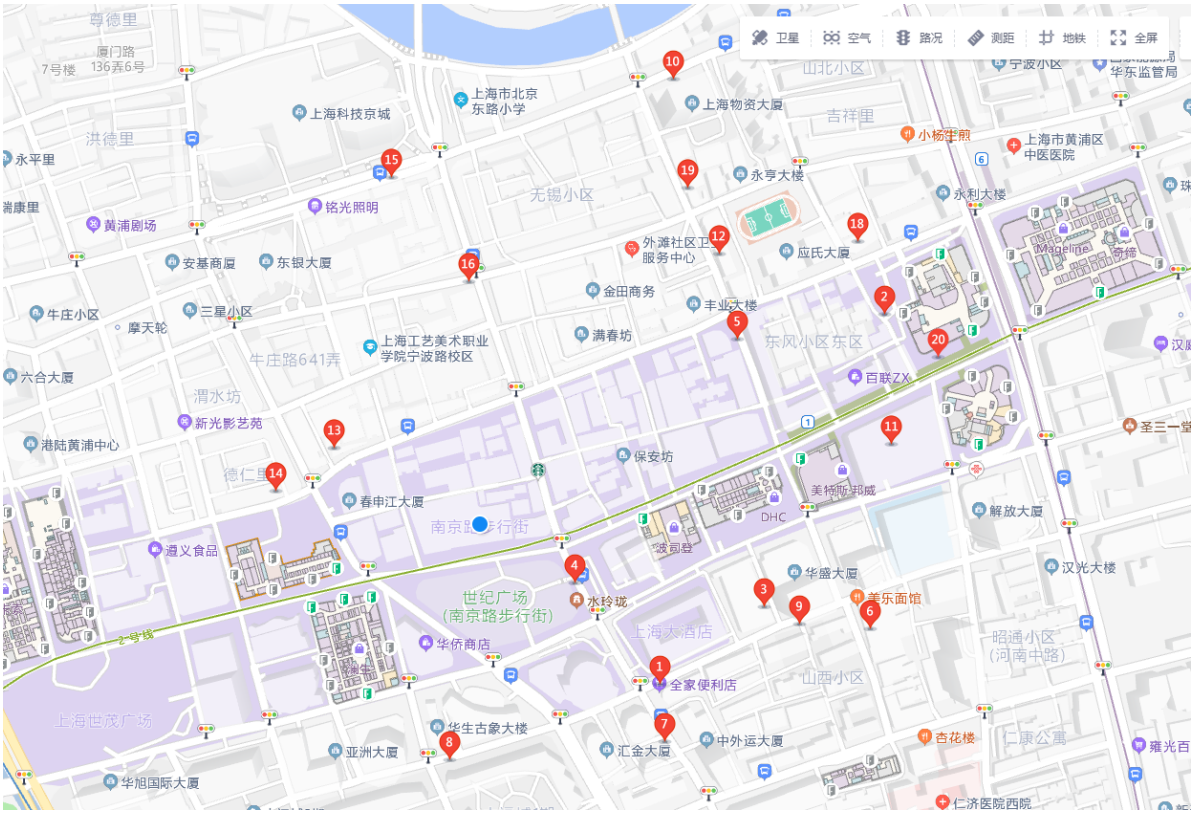
在本文中，介绍了两种实现短链接的方法，分别是哈希算法和 base 62。

哈希算法的特点是，固定的短链接长度，不需要生成唯一ID，可能会出现哈希冲突。

base 62 转换的特点是，长度不固定，取决于 ID 的大小，1000 转换后是 G8, 1000 亿 转换后是 1l9Zo9o。另外还需要生成唯一数字 ID，没有哈希冲突的问题。

4.基于位置的服务

在本文中，我们将设计一个邻近服务，用来发现用户附近的地方，比如餐馆，酒店，商场等。



设计要求

从一个小明去面试的故事开始。

面试官：你好，我想考察一下你的设计能力，如果让你设计一个邻近服务，用来搜索用户附近的商家，你会怎么做？

小明：好的，用户可以指定搜索半径吗？如果搜索范围内没有足够的商家，系统是否支持扩大搜索范围？

面试官：对，用户可以根据需要修改，大概有以下几个选项，0.5km，1km，2km，5km，10km，20km。

小明：嗯，还有其他的系统要求吗？

面试官：另外还需要考虑的是，系统的低延迟，高可用，和可扩展性，以及数据隐私。

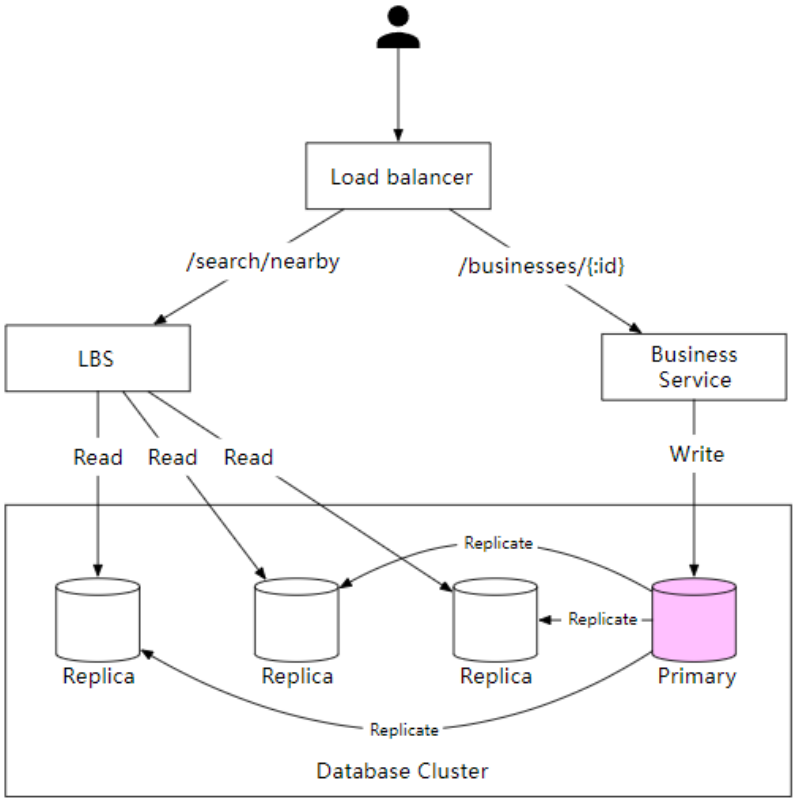
小明：好的，了解了。

总结一下，需要做一个邻近服务，可以根据用户的位置（经度和纬度）以及搜索半径返回附近的商家，半径可以修改。因为用户的位置信息是敏感数据，我们可能需要遵守数据隐私保护法。

高层次设计

高层次设计图如下所示，系统包括两部分：基于位置的服务 （location-based service）LBS 和业务（bussiness）相关的服务。

让我们来看看系统的每个组件。



负载均衡器

负载均衡器可以根据路由把流量分配给多个后端服务。

基于位置的服务 (LBS)

LBS 服务是系统的核心部分，通过位置和半径寻找附近的商家。LBS 具有以下特点：

- 没有写请求，但是有大量的查询
- QPS 很高，尤其是在密集地区的高峰时段。
- 服务是无状态的，支持水平扩展。

Business 服务

商户创建，更新，删除商家信息，以及用户查看商家信息。

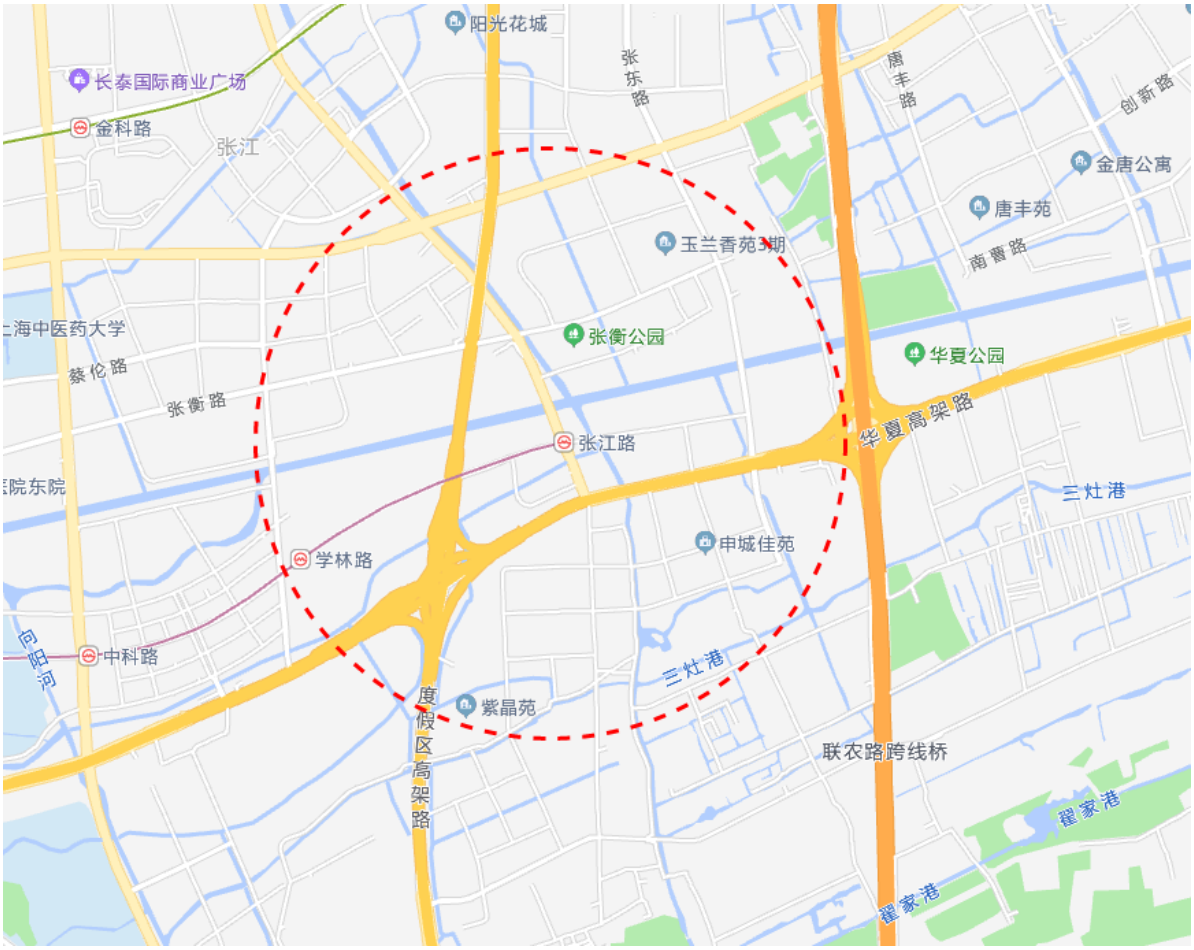
数据库集群

数据库集群可以使用主从配置，提升可用性和性能。数据首先保存到主数据库，然后复制到从库，主数据库处理所有的写入操作，多个从数据库用于读取操作。

接下来，我们具体讨论位置服务 LBS 的实现。

1. 二维搜索

这种方法简单，有效，根据用户的位置和搜索半径画一个圆，然后找到圆圈内的所有商家，如下所示。



商家的纬度用 latitude 表示，经度用 longitude 表示。同样的用户的纬度和经度可以用 user_latitude 和 user_longitude 表示，半径用 radius 表示。

上面的搜索过程可以翻译成下面的伪 SQL 。

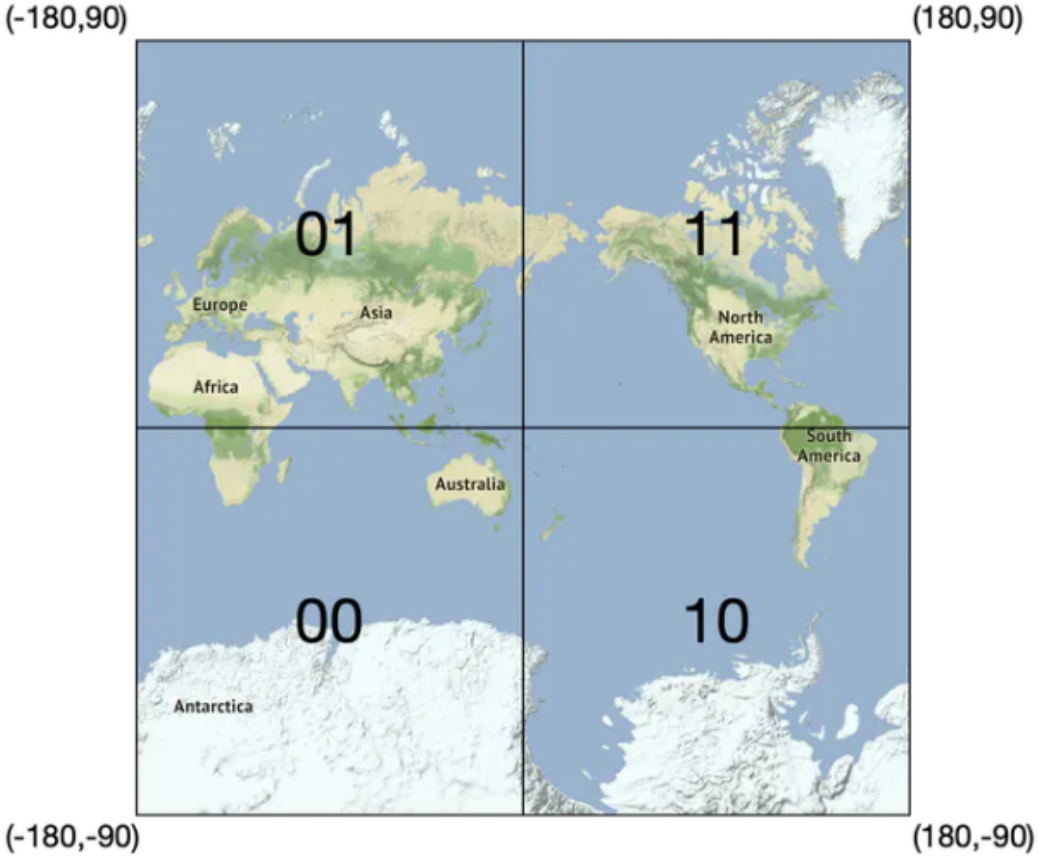
```
SELECT business_id, latitude, longitude,
FROM business
WHERE
latitude >= (@user_latitude - radius) AND latitude < (@user_latitude + radius)
AND
longitude >= (@user_longitude - radius) AND longitude < (@user_longitude +
radius)
```

这种方式可以实现我们的需求，但是实际上效率不高，因为我们需要扫描整个表。虽然我们可以对经纬度创建索引，效率有提升，但是并不够，我们还需要对索引的结果计算取并集。

2. Geohash

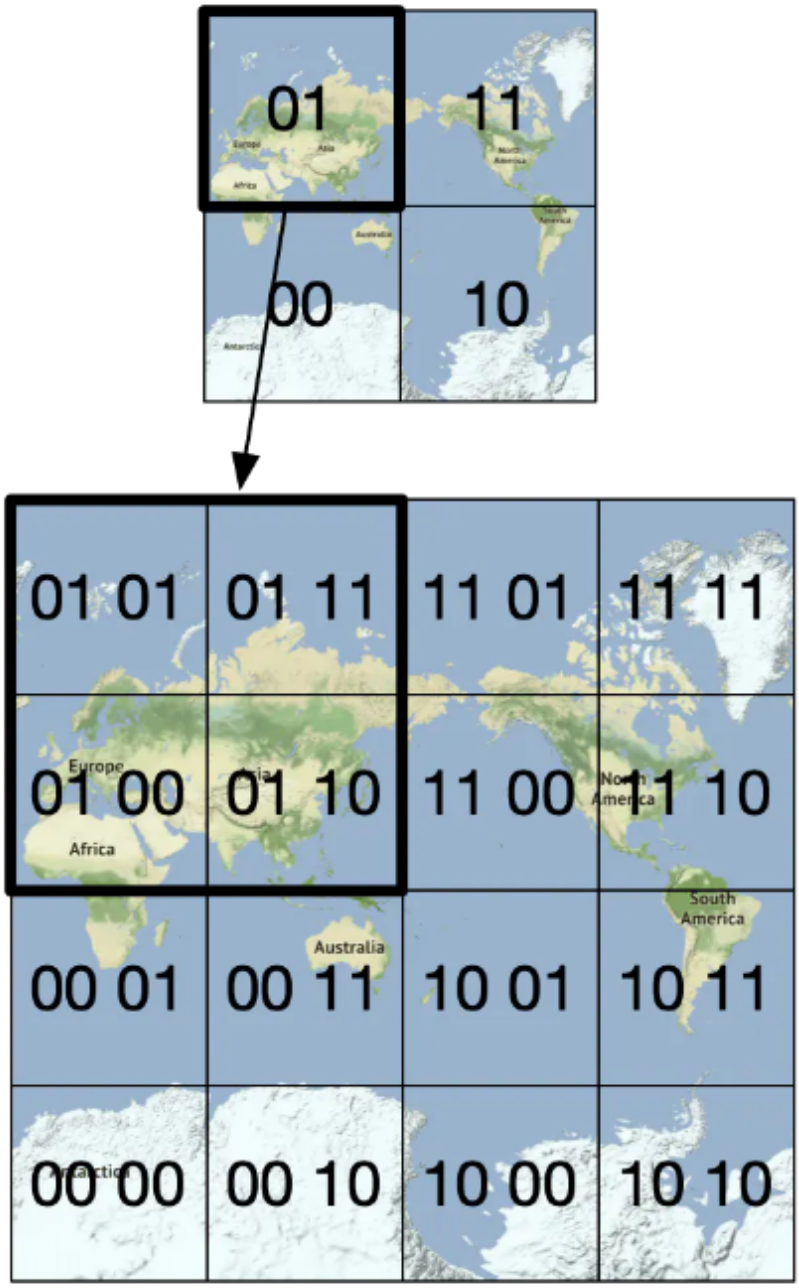
我们上面说了，二维的经度和纬度做索引的效果并不明显。而 Geohash 可以把二维的经度和纬度转换为一维的字符串，通过算法，每增加一位就递归地把世界划分为越来越小的网格，让我们来看看它是如何实现的。

首先，把地球通过本初子午线和赤道分成四个象限，如下



- 纬度范围 [-90, 0] 用 0 表示
- 纬度范围 [0, 90] 用 1 表示
- 经度范围 [-180, 0] 用 0 表示
- 经度范围 [0, 180] 用 1 表示

然后，再把每个网格分成四个小网格。



重复这个过程，直到网格的大小符合我们的需求，Geohash 通常使用 base32 表示。让我们看两个例子。

- Google 总部的 Geohash（长度为 6）：

```
1001 10110 01001 10000 11011 11010 (base32 convert) → 9q9hvu (base32)
```

- Facebook 总部的 Geohash（长度 为 6）：

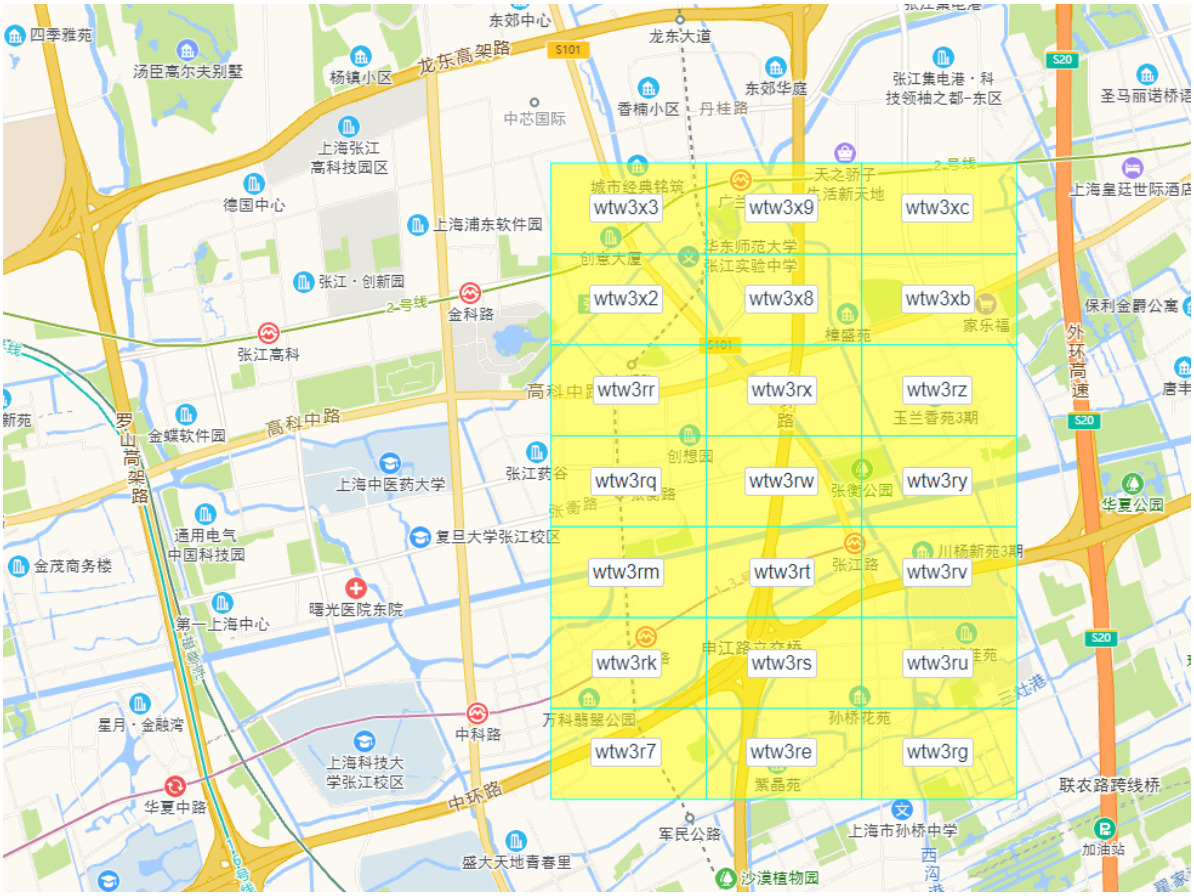
```
1001 10110 01001 10001 10000 10111 (base32 convert) → 9q9jhr (base32)
```

Geohash 有 12 个精度（也称为级别），它可以控制每个网格的大小，字符串越长，拆分的网格就越小，如下

Geohash length	Grid width x height
1	5,009.4km x 4,992.6km (the size of the planet)
2	1,252.3km x 624.1km
3	156.5km x 156km
4	**39.1km x 19.5km**
5	**4.9km x 4.9km**
6	**1.2km x 609.4m**
7	152.9m x 152.4m
8	38.2m x 19m
9	4.8m x 4.8m
10	1.2m x 59.5cm
11	14.9cm x 14.9cm
12	3.7cm x 1.9cm

实际中，按照具体的场景选择合适的 Geohash 精度。

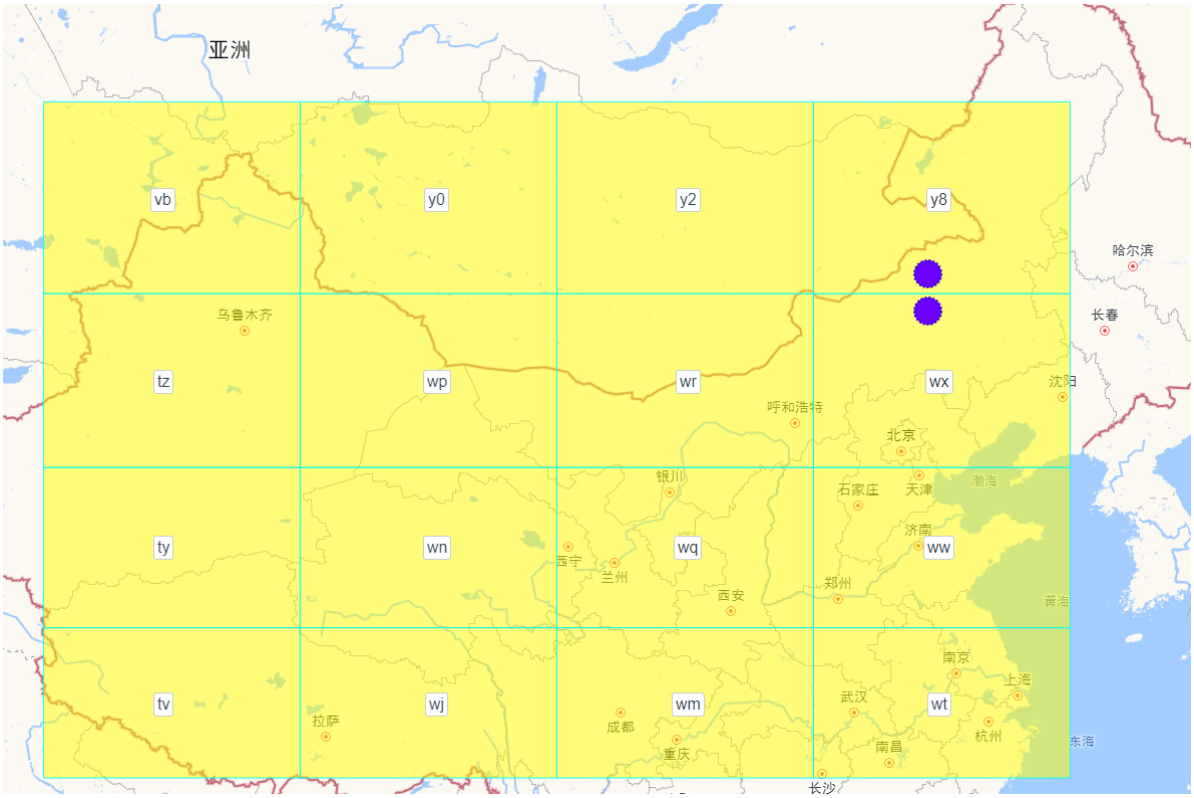
通过这种方式，最终把地图分成了下面一个个小的网格，一个 Geohash 字符串就表示了一个网格，这样查询每个网格内的商家信息，搜索是非常高效的。



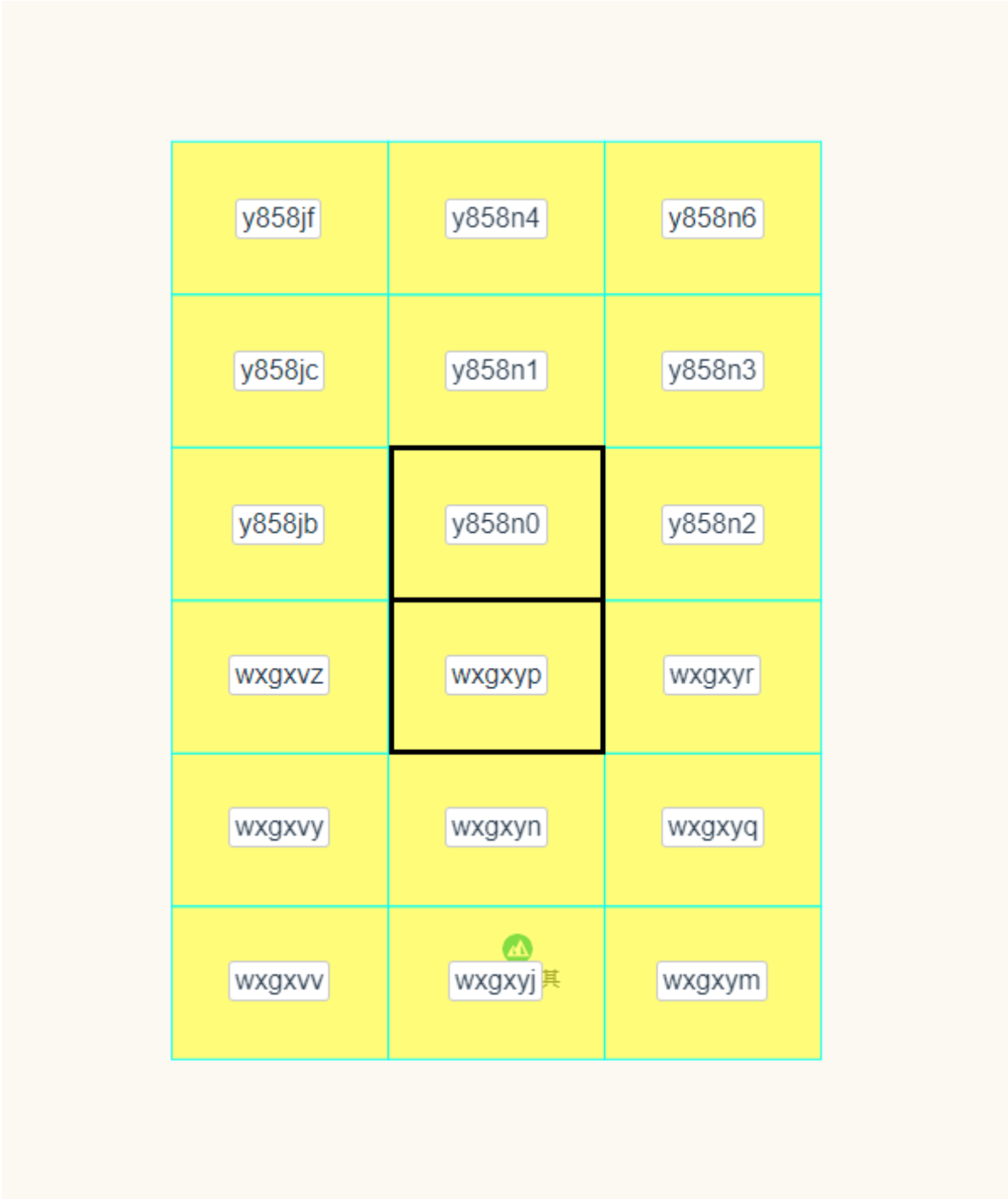
可能你已经发现了一些规律，上图的每个网格中，它们都相同的前缀 `wtw3`。是的，Geohash 的特点是，两个网格的相同前缀部分越长，就表示它们的位置是邻近的。

反过来说，两个相邻的网格，它们的 Geohash 字符串一定是相似的吗？

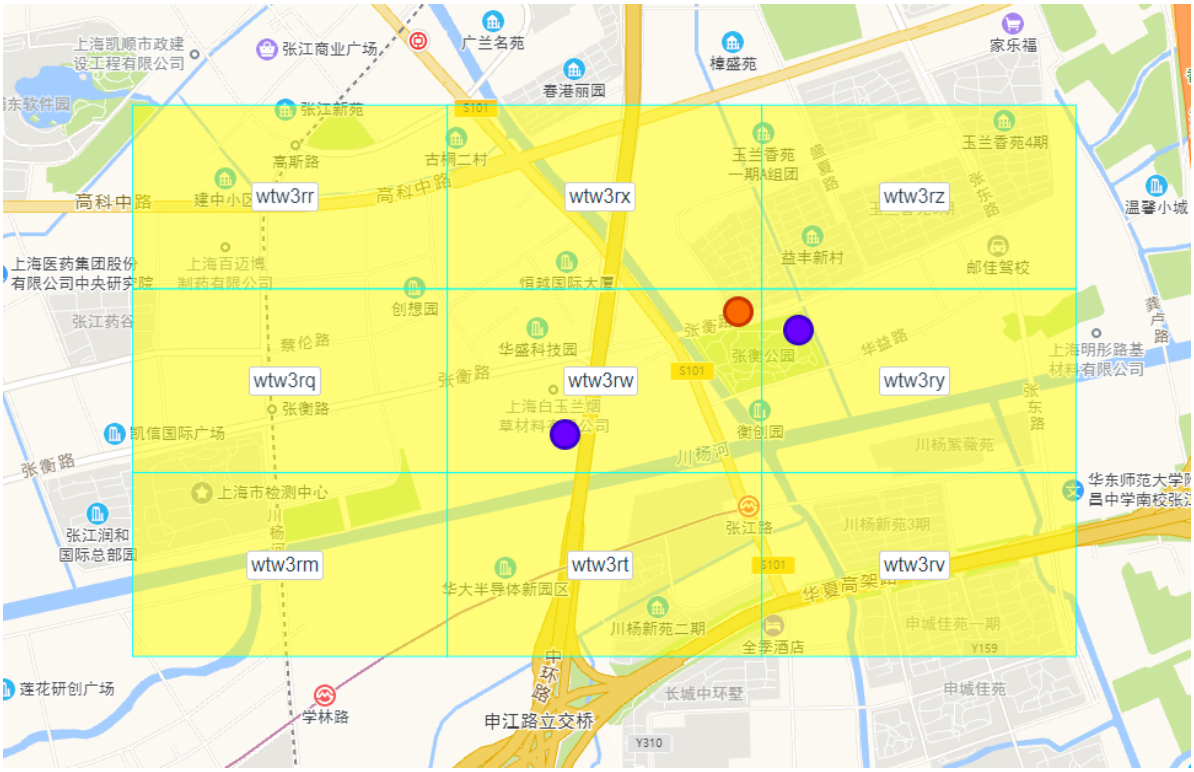
不一定，因为存在 **边界问题**。当两个网格都在边缘时，虽然它们是相邻的，但是 Geohash 的值从第一位就不一样，如下图，两个紫色的点相邻。



下面是一个精度比较高的网格，有些相邻网格的 Geohash 的值是完全不一样的。

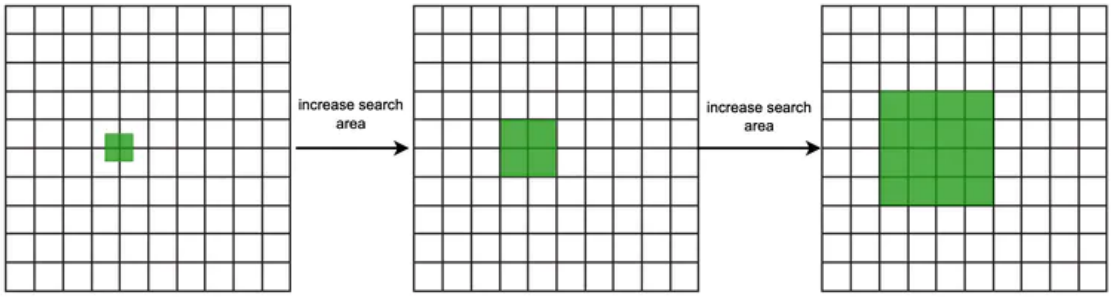


还有一个边界问题是，对于用户（橙色）来说，隔壁网格的商家（紫色）可能比自己网格的商家（紫色）的距离还要近，如下图



所以，在查询附近的商家时，不能只局限于用户所在的网格，要扩大到用户相邻的4个或者9个网格，然后再计算距离，进行筛选，最终找到距离合适的商家。

另外，当在用户在偏远的郊区时，我们可以按照下面的方式，扩大搜索范围，返回足够数量的商家。

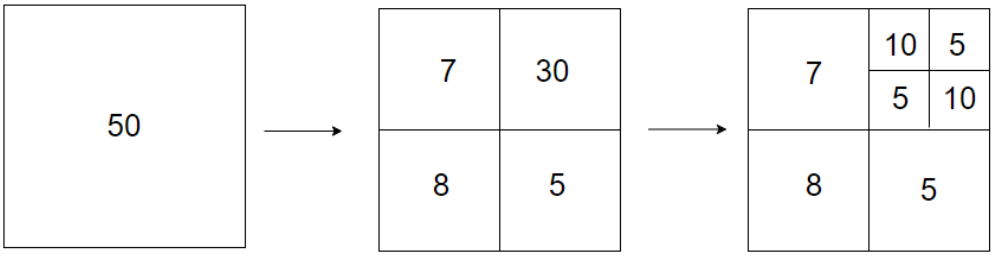


Geohash 的使用非常广泛的，另外 Redis 和 MongoDB 都提供了相应的功能，可以直接使用。

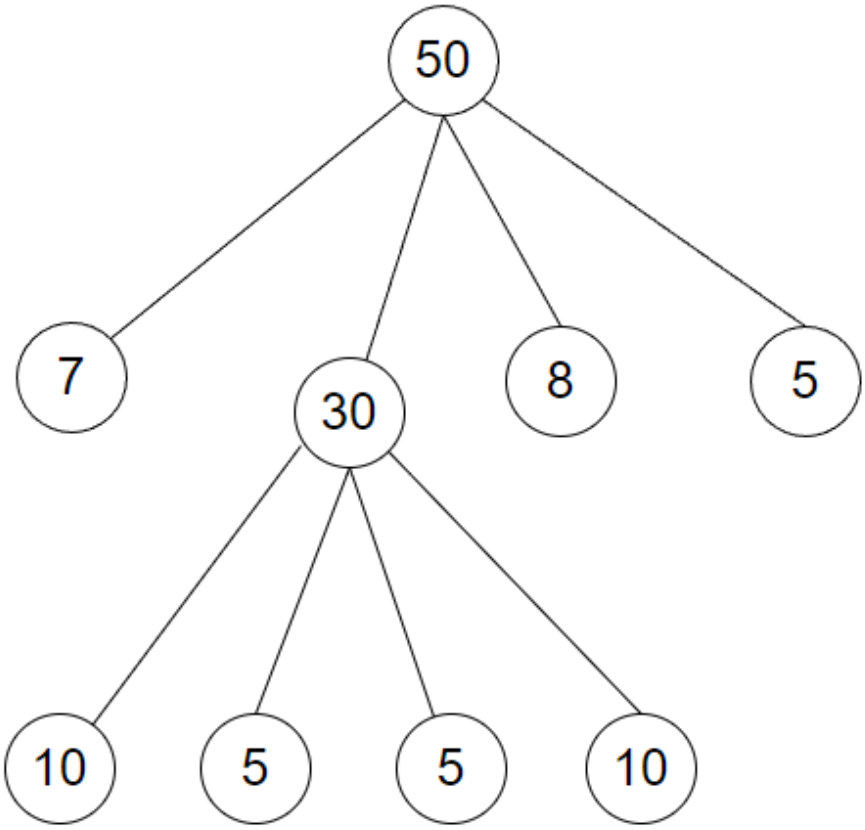
3 . 四叉树

还有一种比较流行的解决方案是四叉树，这种方法可以递归地把二维空间划分为四个象限，直到每个网格的商家数量都符合要求。

如下图，比如确保每个网格的数量不超过10，如果超过，就拆分为四个小的网格。



请注意，**四叉树是一种内存数据结构，它不是数据库解决方案。**它运行在每个LBS 服务上，数据结构是在服务启动时构建的。



接下来，看一下节点都存储了哪些信息?

内部节点

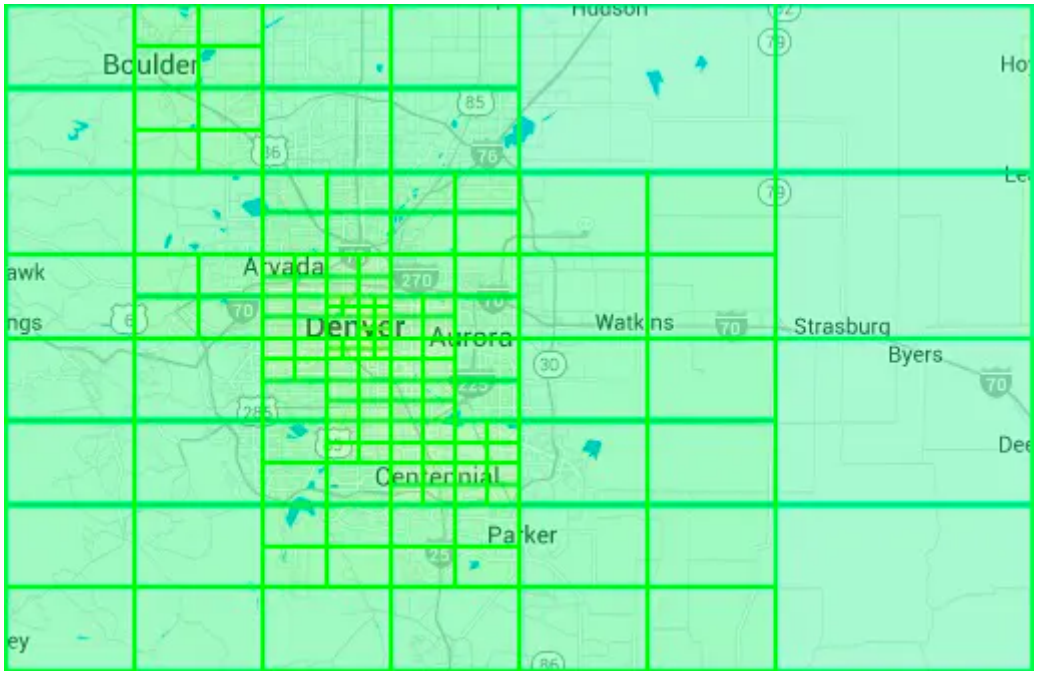
网格的左上角和右下角的坐标，以及指向 4个 子节点的指针。

叶子节点

网格的左上角和右下角的坐标，以及网格内的商家的 ID 数组。

现实世界的四叉树示例

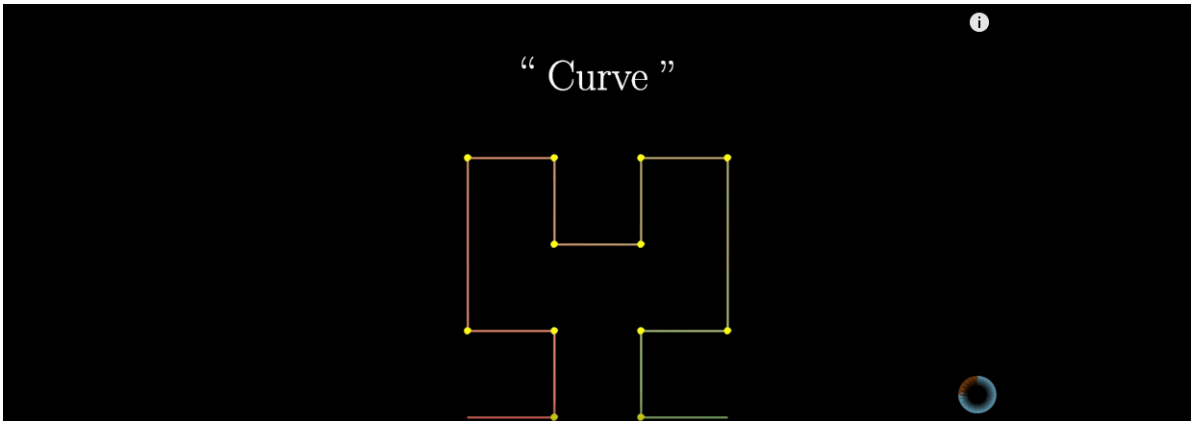
Yext 提供了一张图片 ，显示了其中一个城市构建的四叉树。我们需要更小、更细粒度的网格用在密集区域，而更大的网格用在偏远的郊区。



谷歌 S2 和 希尔伯特曲线

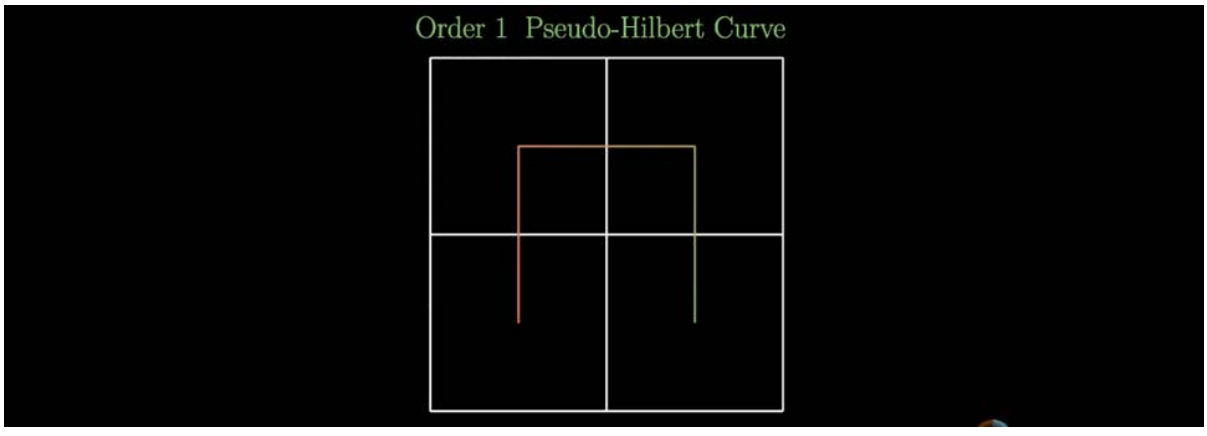
Google S2 库是这个领域的另一个重要参与者，和四叉树类似，它是一种内存解决方案。它基于希尔伯特曲线把球体映射到一维索引。

而 **希尔伯特曲线** 是一种能填充满一个平面正方形的分形曲线（空间填充曲线），由大卫·希尔伯特在1891年提出，如下

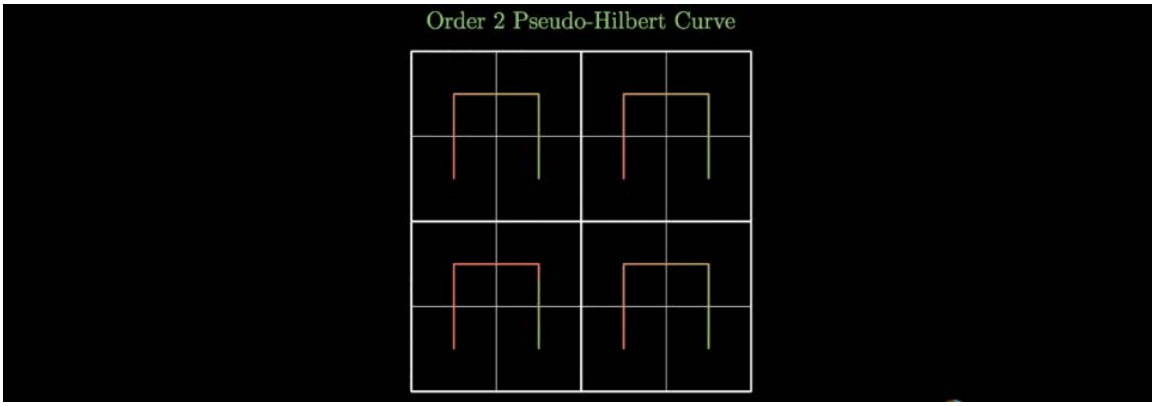


希尔伯特曲线是怎么生成的?

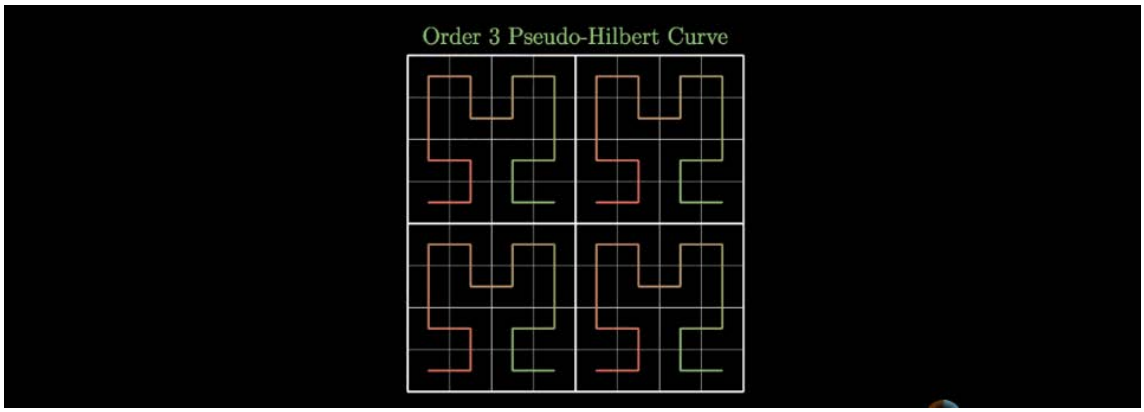
最简单的一阶希尔伯特曲线，先把正方形平均分成四个网格，然后从其中一个网格的正中心开始，按照方向，连接每一个网格。



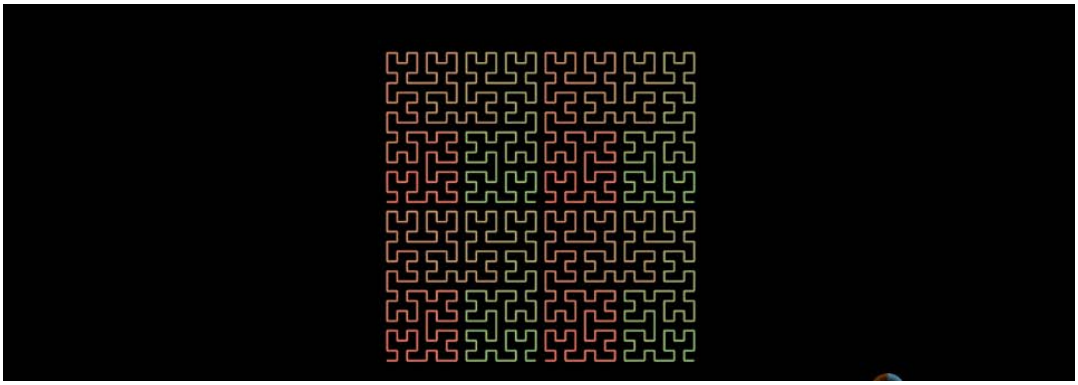
二阶的希尔伯特曲线， 每个网格都先生成一阶希尔伯特曲线， 然后把它们首尾相连。



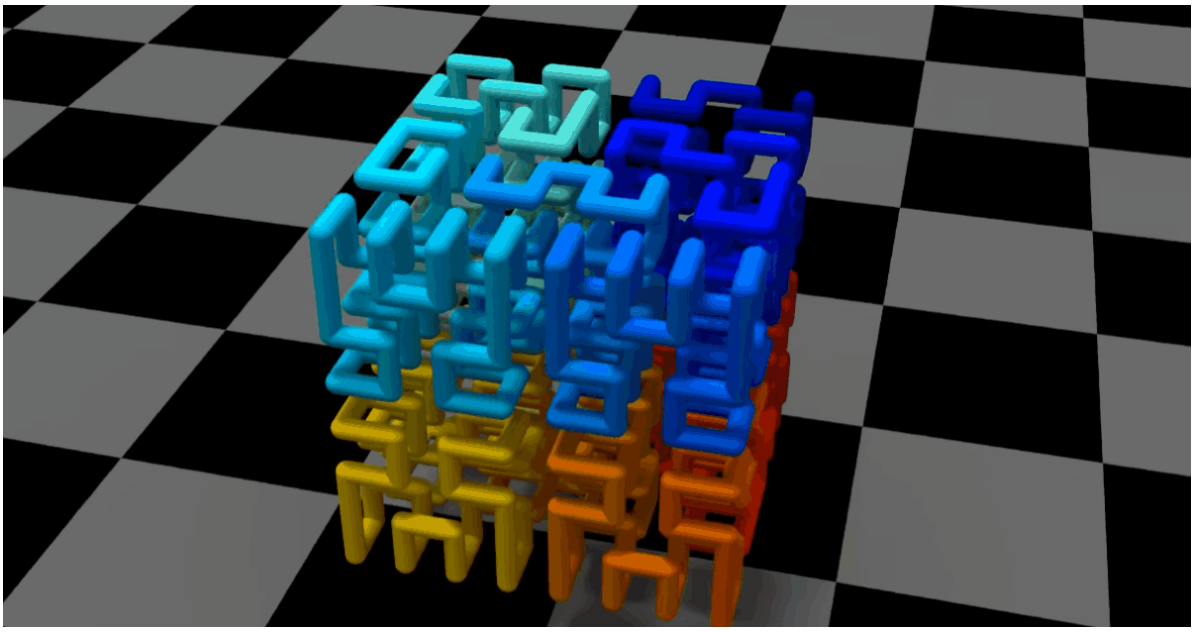
三阶的希尔伯特曲线



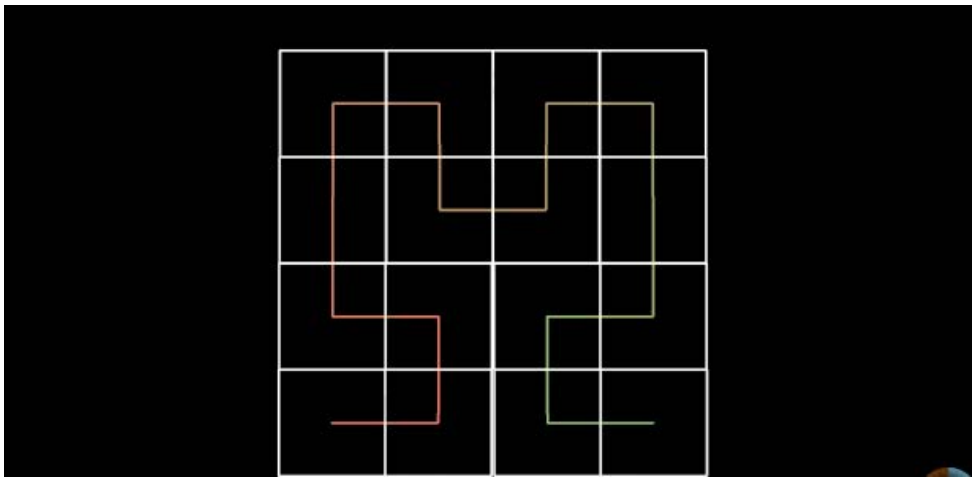
n阶的希尔伯特曲线, 实现一条线连接整个平面。



同样，希尔伯特曲线也可以填充整个三维空间。



希尔伯特曲线的一个重要特点是 **降维**，可以把多维空间转换成一维数组，可以通过动画看看它是如何实现的。

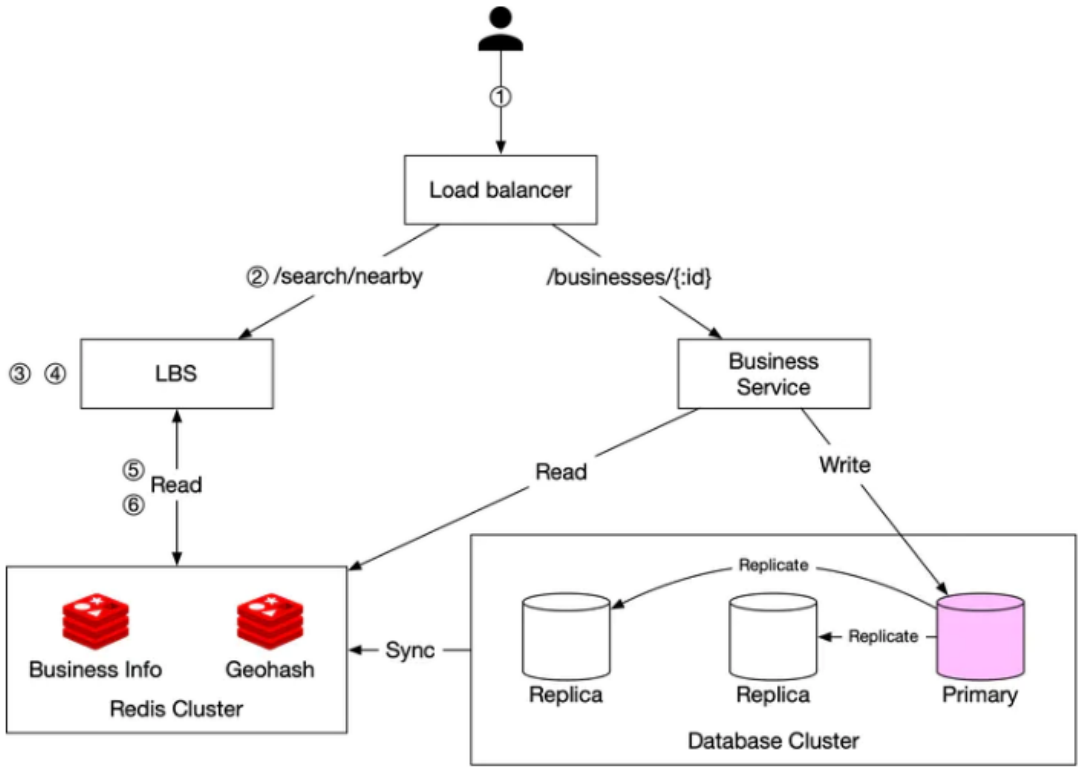


在一维空间上的搜索比在二维空间上的搜索效率高得多了。

多数据中心和高可用

我们可以把 LBS 服务部署到多个区域，不同地区的用户连接到最近的数据中心，这样做可以提升访问速度以及系统的高可用，并根据实际的场景，进行扩展。

最终设计图



1. 用户需要寻找附近 500 米的餐馆。客户端把用户位置（经度和纬度），半径（500m）发送给后端。
2. 负载均衡器把请求转发给 LBS。
3. 基于用户位置和半径信息，LBS 找到与搜索匹配的 geohash 长度。
4. LBS 计算相邻的 Geohash 并将它们添加到列表中。
5. 调用 Redis 服务获取对应的商家 ID。
6. LBS 根据返回的商家列表，计算用户和商家之间的距离，并进行排名，然后返回给客户端。

总结

在本文中，我们设计了一个邻近服务，介绍了4种常见了实现方式，分别是二维搜索，Geohash, 四叉树和 Google S2。它们有各自的优缺点，您可以根据实际的业务场景，选择合适的实现。

Reference

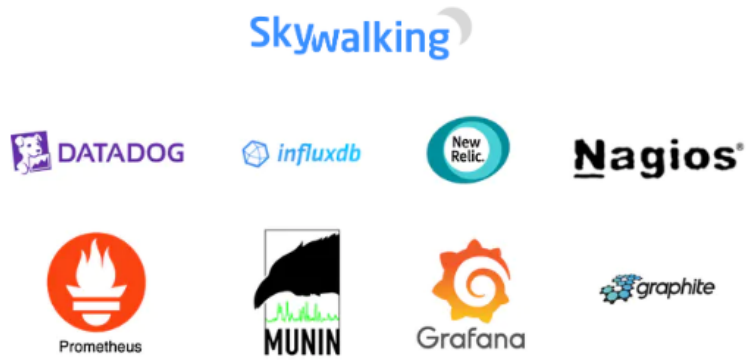
https://halfrost.com/go_spatial_search/#toc-25

https://www.amazon.com/System-Design-Interview-Insiders-Guide/dp/1736049119

5.指标监控和告警系统

在本文中，我们将探讨如何设计一个可扩展的指标监控和告警系统。 一个好的监控和告警系统，对基础设施的可观察性，高可用性，可靠性方面发挥着关键作用。

下图显示了市面上一些流行的指标监控和告警服务。



接下来，我们会设计一个类似的服务，可以供大公司内部使用。

设计要求

从一个小明去面试的故事开始。

面试官：如果让你设计一个指标监控和告警系统，你会怎么做?

小明：好的，这个系统是为公司内部使用的，还是设计像 Datadog 这种 SaaS 服务?

面试官：很好的问题，目前这个系统只是公司内部使用。

小明：我们想收集哪些指标信息?

面试官：包括操作系统的指标信息，中间件的指标，以及运行的应用服务的 qps 这些指标。

小明：我们用这个系统监控的基础设施的规模是多大的?

面试官：1亿日活跃用户，1000个服务器池，每个池 100 台机器。

小明：指标数据要保存多长时间呢?

面试官：我们想保留一年。

小明：好吧，为了较长时间的存储，可以降低指标数据的分辨率吗?

面试官：很好的问题，对于最新的数据，会保存 7 天，7天之后可以降低到1分钟的分辨率，而到 30 天之后，可以按照 1 小时的分辨率做进一步的汇总。

小明：支持的告警渠道有哪些?

面试官：邮件，电 钉钉，企业微信，Http Endpoint。

小明：我们需要收集日志吗？还有是否需要支持分布式系统的链路追踪?

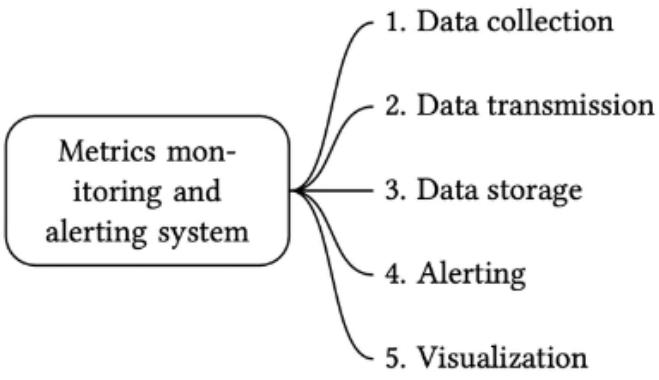
面试官：目前专注于指标，其他的暂时不考虑。

小明：好的，大概都了解了。

总结一下，被监控的基础设施是大规模的，以及需要支持各种维度的指标。另外，整体的系统也有较高的要求，要考虑到可扩展性，低延迟，可靠性和灵活性。

基础知识

一个指标监控和告警系统通常包含五个组件，如下图所示



1. 数据收集：从不同的数据源收集指标数据。
2. 数据传输：把指标数据发送到指标监控系统。
3. 数据存储：存储指标数据。
4. 告警：分析接收到的数据，检测到异常时可以发出告警通知。
5. 可视化：可视化页面，以图形，图表的形式呈现数据。

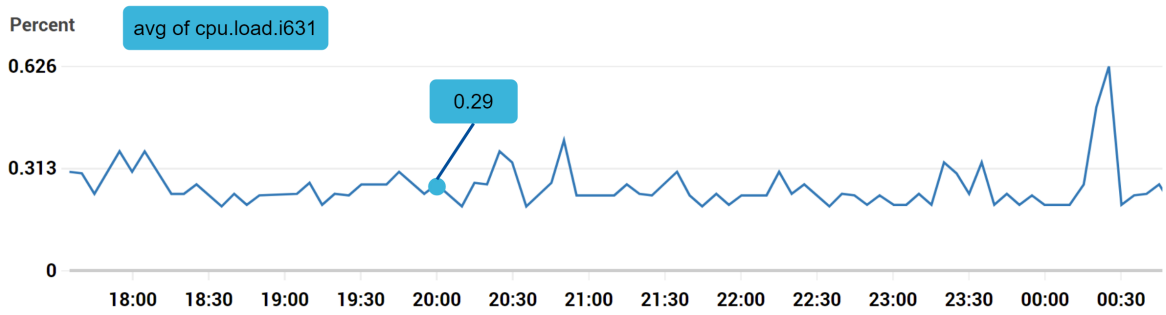
数据模式

指标数据通常会保存为一个时间序列，其中包含一组值及其相关的时间戳。

序列本身可以通过名称进行唯一标识，也可以通过一组标签进行标识。

让我们看两个例子。

示例1：生产服务器 i631 在 20:00 的 CPU 负载是多少?



上图标记的数据点可以用下面的格式表示

metric_name	cpu.load
labels	host:i631,env:prod
timestamp	1613707265
value	0.29

在上面的示例中，时间序列由指标名称，标签（host:i631,env:prod），时间戳以及对应的值构成。

示例2：过去 10 分钟内上海地区所有 Web 服务器的平均 CPU 负载是多少?

从概念上来讲，我们会查询出和下面类似的内容

```
CPU.load host=webserver01,region=shanghai 1613707265 50

CPU.load host=webserver01,region=shanghai 1613707270 62

CPU.load host=webserver02,region=shanghai 1613707275 43
```

我们可以通过上面每行末尾的值计算平均 CPU 负载，上面的数据格式也称为行协议。是市面上很多监控软件比较常用的输入格式，Prometheus 和 OpenTSDB 就是两个例子。

每个时间序列都包含以下内容：

- 指标名称，字符串类型的 metric name 。
- 一个键值对的数组，表示指标的标签，List<key,value>
- 一个包含时间戳和对应值的的数组，List <value, timestamp>

数据存储

数据存储是设计的核心部分，不建议构建自己的存储系统，也不建议使用常规的存储系统（比如 MySQL）来完成这项工作。

理论下，常规数据库可以支持时间序列数据， 但是需要数据库专家级别的调优后，才能满足数据量比较大的场景需求。

具体点说，关系型数据库没有对时间序列数据进行优化，有以下几点原因

- 在滚动时间窗口中计算平均值，需要编写复杂且难以阅读的 SQL。
- 为了支持标签（tag/label）数据，我们需要给每个标签加一个索引。
- 相比之下，关系型数据库在持续的高并发写入操作时表现不佳。

那 NoSQL 怎么样呢？理论上，市面上的少数 NoSQL 数据库可以有效地处理时间序列数据。比如 Cassandra 和 Bigtable 都可以。但是，想要满足高效存储和查询数据的需求，以及构建可扩展的系统，需要深入了解每个 NoSQL 的内部工作原理。

相比之下，专门对时间序列数据优化的时序数据库，更适合这种场景。

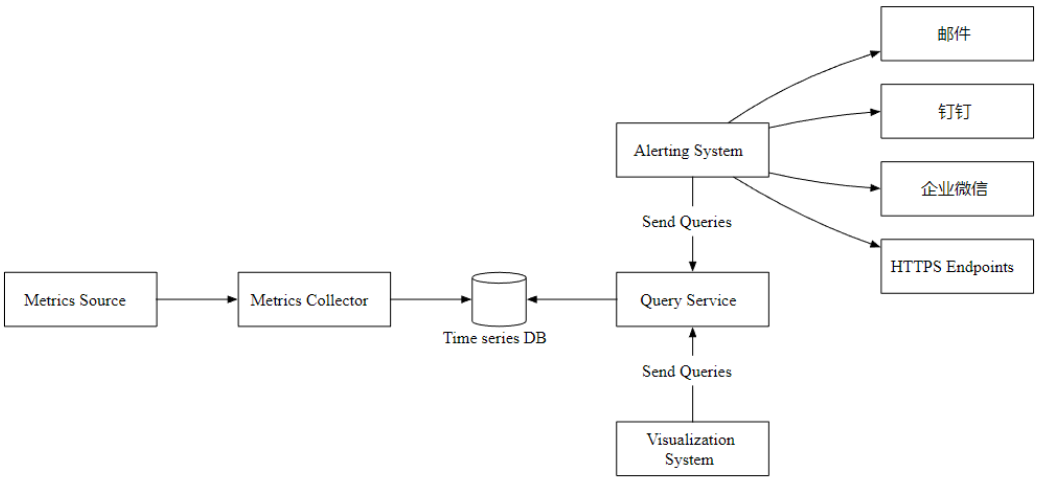
OpenTSDB 是一个分布式时序数据库，但由于它基于 Hadoop 和 HBase，运行 Hadoop/HBase 集群也会带来复杂性。Twitter 使用了 MetricsDB 时序数据库存储指标数据，而亚马逊提供了 Timestream 时序数据库服务。

根据 DB-engines 的报告，两个最流行的时序数据库是 InfluxDB 和 Prometheus，它们可以存储大量时序数据，并支持快速地对这些数据进行实时分析。

如下图所示，8 核 CPU 和 32 GB RAM 的 InfluxDB 每秒可以处理超过 250,000 次写入。

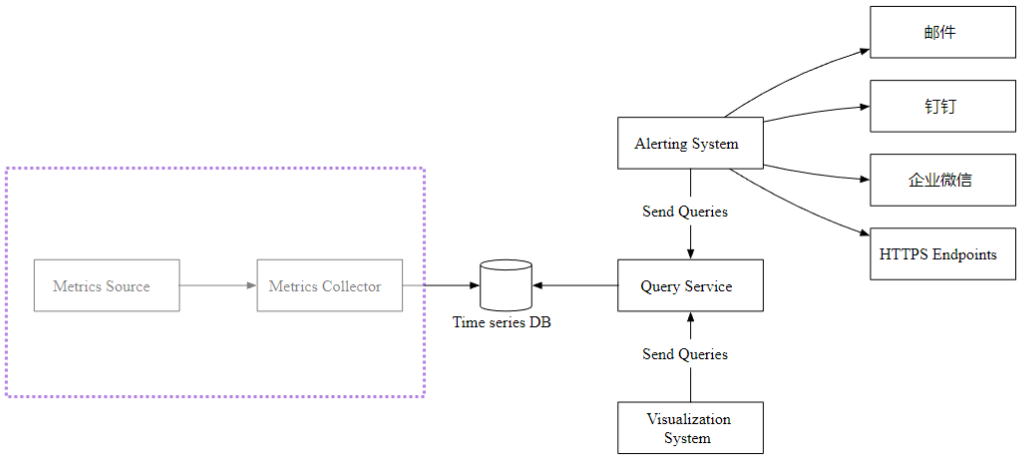
vCPU or CPU	RAM	IOPS	Writes per second	Queries* per second	Unique series
2-4 cores	2-4 GB	500	< 5,000	< 5	< 100,000
4-6 cores	8-32 GB	500-1000	< 250,000	< 25	< 1,000,000
8+ cores	32+ GB	1000+	> 250,000	> 25	> 1,000,000

高层次设计



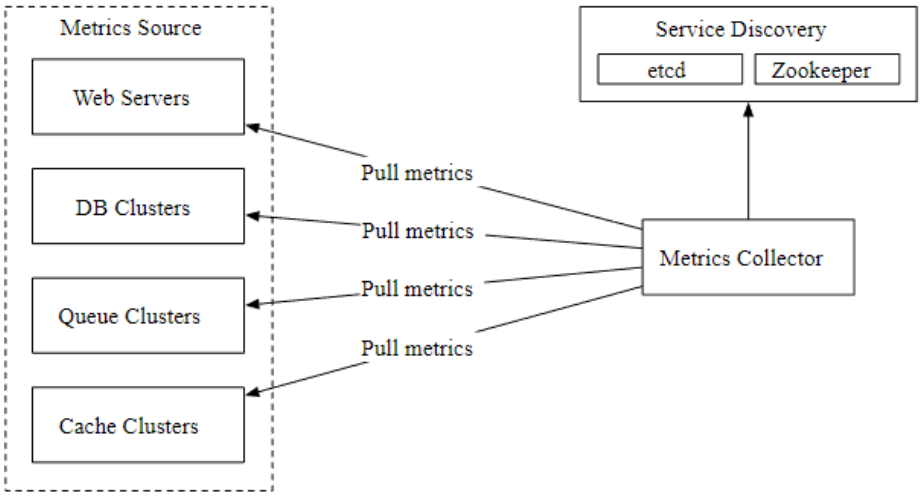
- Metrics Source 指标来源，应用服务，数据库，消息队列等。
- Metrics Collector 指标收集器。
- Time series DB 时序数据库，存储指标数据。
- Query Service 查询服务，向外提供指标查询接口。
- Alerting System 告警系统，检测到异常时，发送告警通知。
- Visualization System 可视化，以图表的形式展示指标。

深入设计



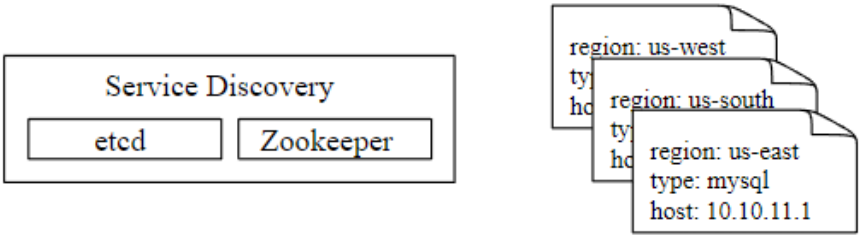
现在，让我们聚焦于数据收集流程。主要有推和拉两种方式。

拉模式

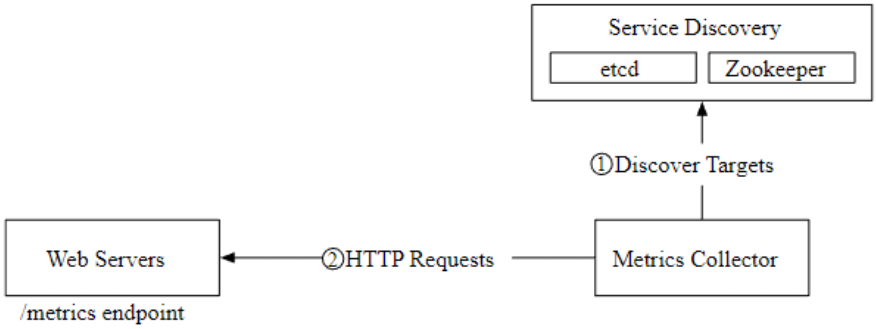


上图显示了使用了拉模式的数据收集，单独设置了数据收集器，定期从运行的应用中拉取指标数据。

这里有一个问题，数据收集器如何知道每个数据源的地址? 一个比较好的方案是引入服务注册发现组件，比如 etcd，ZooKeeper，如下



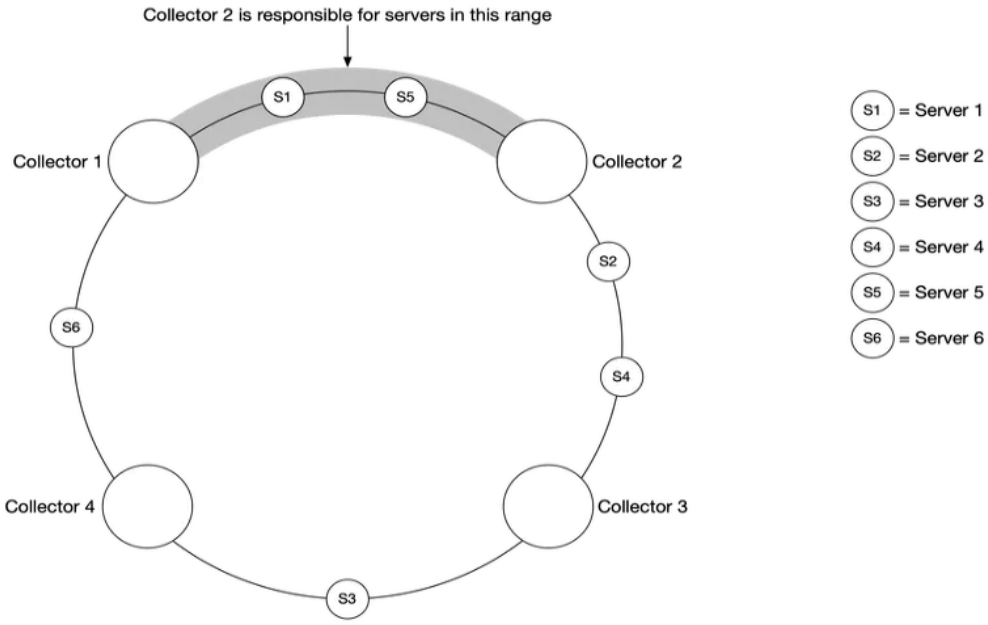
下图展示了我们现在的数据拉取流程。



1. 指标收集器从服务发现组件中获取元数据，包括拉取间隔，IP 地址，超时，重试参数等。
2. 指标收集器通过设定的 HTTP 端点获取指标数据。

在数据量比较大的场景下，单个指标收集器是独木难支的，我们必须使用一组指标收集器。但是多个收集器和多个数据源之间应该如何协调，才能正常工作不发生冲突呢？

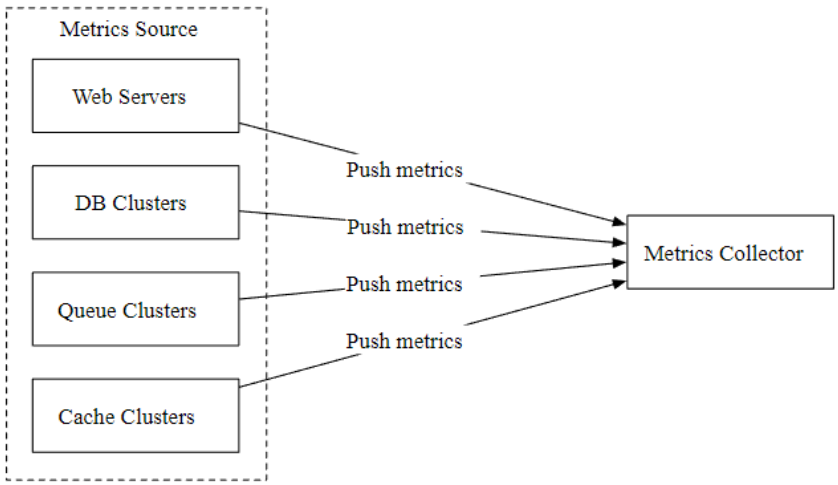
一致性哈希很适合这种场景，我们可以把数据源映射到哈希环上，如下



这样可以保证每个指标收集器都有对应的数据源，相互工作且不会发生冲突。

推模式

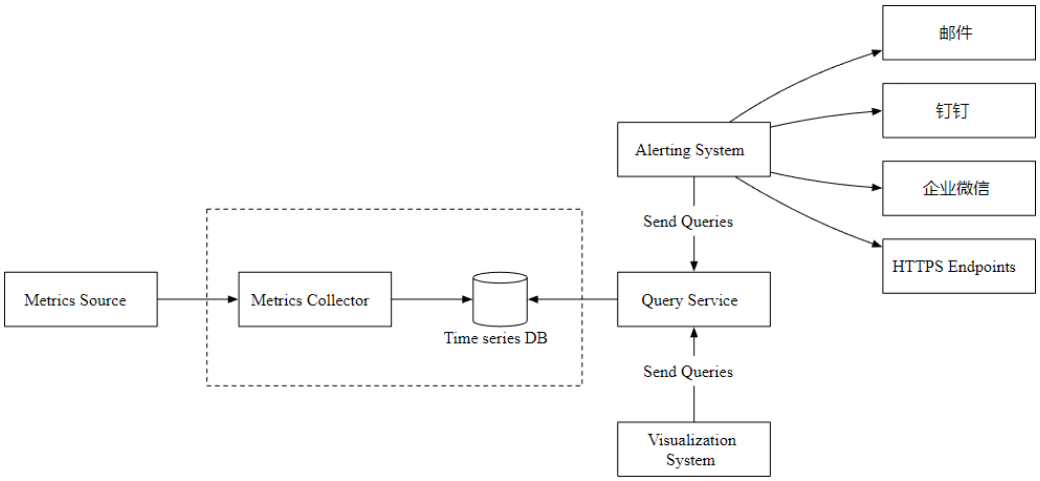
如下图所示，在推模式中，各种指标数据源（Web 应用，数据库，消息队列）直接发送到指标收集器。



在推模式中，需要在每个被监控的服务器上安装收集器代理，它可以收集服务器的指标数据，然后定期的发送给指标收集器。

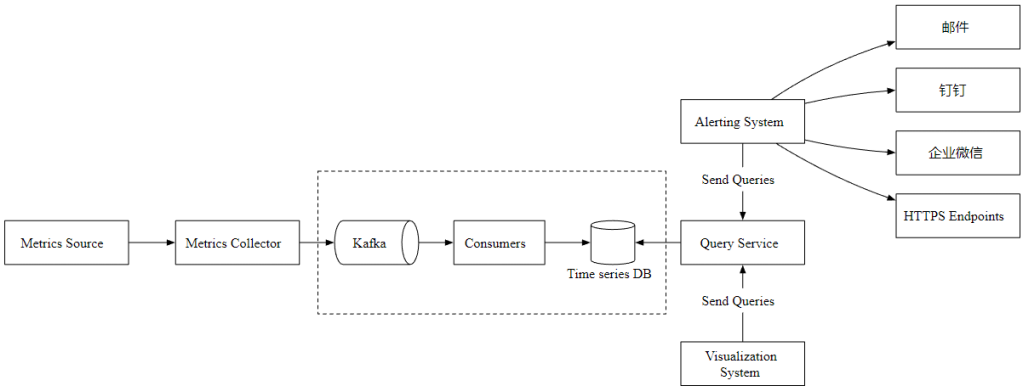
推和拉两种模式哪种更好？没有固定的答案，这两个方案都是可行的，甚至在一些复杂场景中，需要同时支持推和拉。

扩展数据传输



现在，让我们主要关注指标收集器和时序数据库。不管使用推还是拉模式，在需要接收大量数据的场景下，指标收集器通常是一个服务集群。

但是，当时序数据库不可用时，就会存在数据丢失的风险，所以，我们引入了 Kafka 消息队列组件， 如下图



指标收集器把指标数据发送到 Kafka 消息队列，然后消费者或者流处理服务进行数据处理，比如 Apache Storm、Flink 和 Spark，最后再推送到时序数据库。

指标计算

指标在多个地方都可以聚合计算，看看它们都有什么不一样。

- 客户端代理：客户端安装的收集代理只支持简单的聚合逻辑。
- 传输管道：在数据写入时序数据库之前，我们可以用 Flink 流处理服务进行聚合计算，然后只写入汇总后的数据，这样写入量会大大减少。但是由于我们没有存储原始数据，所以丢失了数据精度。
- 查询端：我们可以在查询端对原始数据进行实时聚合查询，但是这样方式查询效率不太高。

时序数据库查询语言

大多数流行的指标监控系统，比如 Prometheus 和 InfluxDB 都不使用 SQL，而是有自己的查询语言。一个主要原因是很难通过 SQL 来查询时序数据, 并且难以阅读，比如下面的SQL 你能看出来在查询什么数据吗?

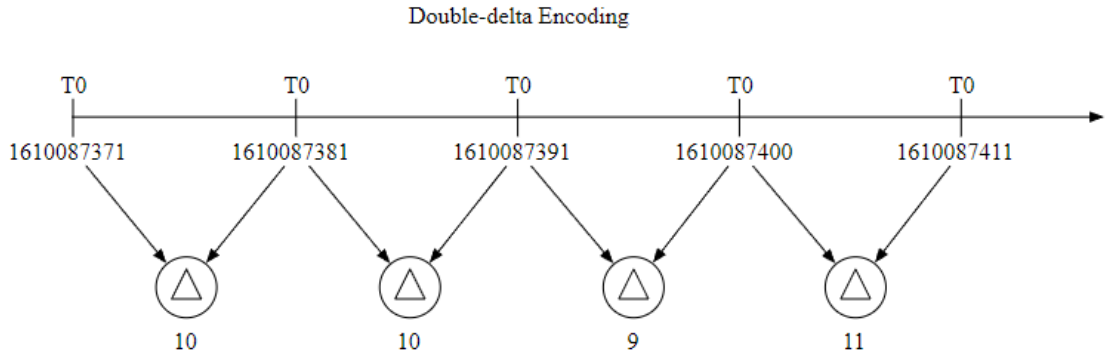
```
select id,
       temp,
       avg(temp) over (partition by group_nr order by time_read) as rolling_avg
from (
  select id,
         temp,
         time_read,
         interval_group,
         id - row_number() over (partition by interval_group order by time_read)
as group_nr
  from (
    select id,
    time_read,
    "epoch"::timestamp + "900 seconds"::interval * (extract(epoch from
time_read)::int4 / 900) as interval_group,
    temp
    from readings
  ) t1
) t2
order by time_read;
```

相比之下， InfluxDB 使用的针对于时序数据的 Flux 查询语言会更简单更好理解，如下

```
from(db:"telegraf")
  |> range(start:-1h)
  |> filter(fn: (r) => r._measurement == "foo")
  |> exponentialMovingAverage(size:-10s)
```

数据编码和压缩

数据编码和压缩可以很大程度上减小数据的大小，特别是在时序数据库中，下面是一个简单的例子。



因为一般数据收集的时间间隔是固定的，所以我们可以把一个基础值和增量一起存储，比如 1610087371, 10, 10, 9, 11 这样，可以占用更少的空间。

下采样

下采样是把高分辨率的数据转换为低分辨率的过程，这样可以减少磁盘使用。由于我们的数据保留期是1年，我们可以对旧数据进行下采样，这是一个例子：

- 7天数据，不进行采样。
- 30天数据，下采样到1分钟的分辨率
- 1年数据，下采样到1小时的分辨率。

我们看另外一个具体的例子，它把 10 秒分辨率的数据聚合为 30 秒分辨率。

原始数据

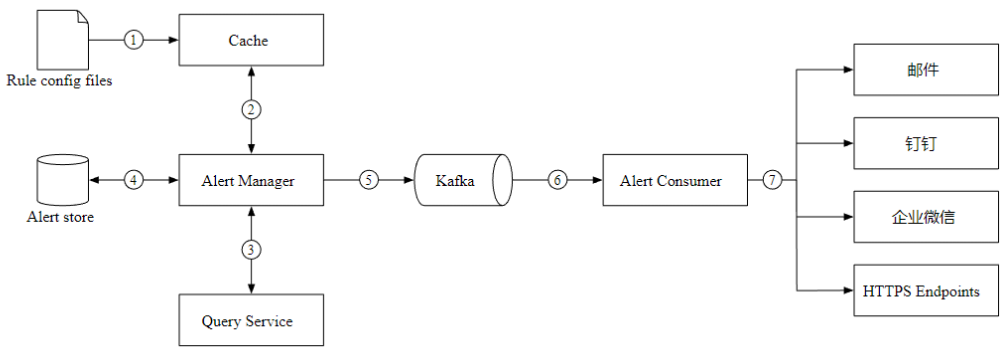
metric	timestamp	hostname	Metric_value
cpu	2022-07-04T19:00:00Z	host-a	10
cpu	2022-07-04T19:00:10Z	host-a	16
cpu	2022-07-04T19:00:20Z	host-a	20
cpu	2022-07-04T19:00:30Z	host-a	30
cpu	2022-07-04T19:00:40Z	host-a	20
cpu	2022-07-04T19:00:50Z	host-a	30

下采样之后

metric	timestamp	hostname	Metric_value (avg)
cpu	2022-07-04T19:00:00Z	host-a	19
cpu	2022-07-04T19:00:30Z	host-a	25

告警服务

让我们看看告警服务的设计图，以及工作流程。



1. 加载 YAML 格式的告警配置文件到缓存。

```
- name: instance_down
  rules:
  # 服务不可用时间超过 5 分钟触发告警.
  - alert: instance_down
    expr: up == 0
    for: 5m
    labels:
      severity: page
```

2. 警报管理器从缓存中读取配置。
3. 根据告警规则，按照设定的时间和条件查询指标，如果超过阈值，则触发告警。
4. Alert Store 保存着所有告警的状态（挂起，触发，已解决）。
5. 符合条件的告警会添加到 Kafka 中。
6. 消费队列，根据告警规则，发送警报信息到不同的通知渠道。

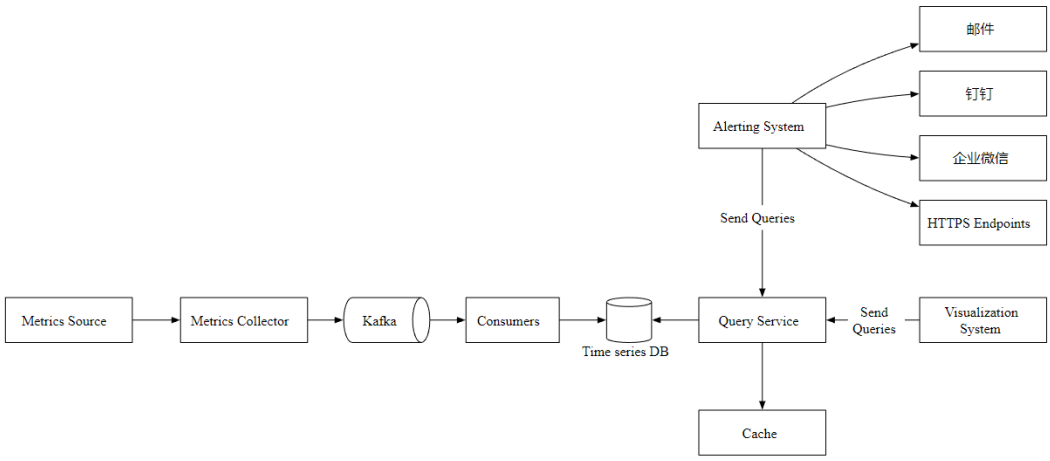
可视化

可视化建立在数据层之上，指标数据可以在指标仪表板上显示，告警信息可以在告警仪表板上显示。下图显示了一些指标，服务器的请求数量、内存/CPU 利用率、页面加载时间、流量和登录信息。

Grafana 可以是一个非常好的可视化系统，我们可以直接拿来使用。

总结

在本文中，我们介绍了指标监控和告警系统的设计。在高层次上，我们讨论了数据收集、时序数据库、告警和可视化，下图是我们最终的设计：



Reference

[0] System Design Interview Volume 2: https://www.amazon.com/System-Design-Interview-Insiders-Guide/dp/1736049119

[1] Datadog: https://www.datadoghq.com/

[2] Splunk: https://www.splunk.com/

[3] Elastic stack: https://www.elastic.co/elastic-stack

[4] Dapper, a Large-Scale Distributed Systems Tracing Infrastructure: https://research.google/pubs/pub36356/

[5] Distributed Systems Tracing with Zipkin: https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.html

[6] Prometheus: https://prometheus.io/docs/introduction/overview/

[7] OpenTSDB - A Distributed, Scalable Monitoring System: http://opentsdb.net/

[8] Data model: : https://prometheus.io/docs/concepts/data_model/

[9] Schema design for time-series data | Cloud Bigtable Documentation https://cloud.google.com/bigtable/docs/schema-design-time-series

[10] MetricsDB: TimeSeries Database for storing metrics at Twitter: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/metricsdb.html

[11] Amazon Timestream: https://aws.amazon.com/timestream/

[12] DB-Engines Ranking of time-series DBMS: https://db-engines.com/en/ranking/time+series+dbms

[13] InfluxDB: https://www.influxdata.com/

[14] etcd: https://etcd.io

[15] Service Discovery with Zookeeper https://cloud.spring.io/spring-cloud-zookeeper/1.2.x/multi/multi_spring-cloud-zookeeper-discovery.html

[16] Amazon CloudWatch: https://aws.amazon.com/cloudwatch/

[17] Graphite: https://graphiteapp.org/

[18] Push vs Pull: http://bit.ly/3aJEPxE

[19] Pull doesn't scale - or does it?: https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/

[20] Monitoring Architecture: https://developer.lightbend.com/guides/monitoring-at-scale/monitoring-architecture/architecture.html

[21] Push vs Pull in Monitoring Systems: https://giedrius.blog/2019/05/11/push-vs-pull-in-monitoring-systems/

[22] Pushgateway: https://github.com/prometheus/pushgateway

[23] Building Applications with Serverless Architectures https://aws.amazon.com/lambda/serverless-architectures-learn-more/

[24] Gorilla: A Fast, Scalable, In-Memory Time Series Database: http://www.vldb.org/pvldb/vol8/p1816-teller.pdf

[25] Why We're Building Flux, a New Data Scripting and Query Language: https://www.influxdata.com/blog/why-were-building-flux-a-new-data-scripting-and-query-language/

[26] InfluxDB storage engine: https://docs.influxdata.com/influxdb/v2.0/reference/internals/storage-engine/

[27] YAML: https://en.wikipedia.org/wiki/YAML

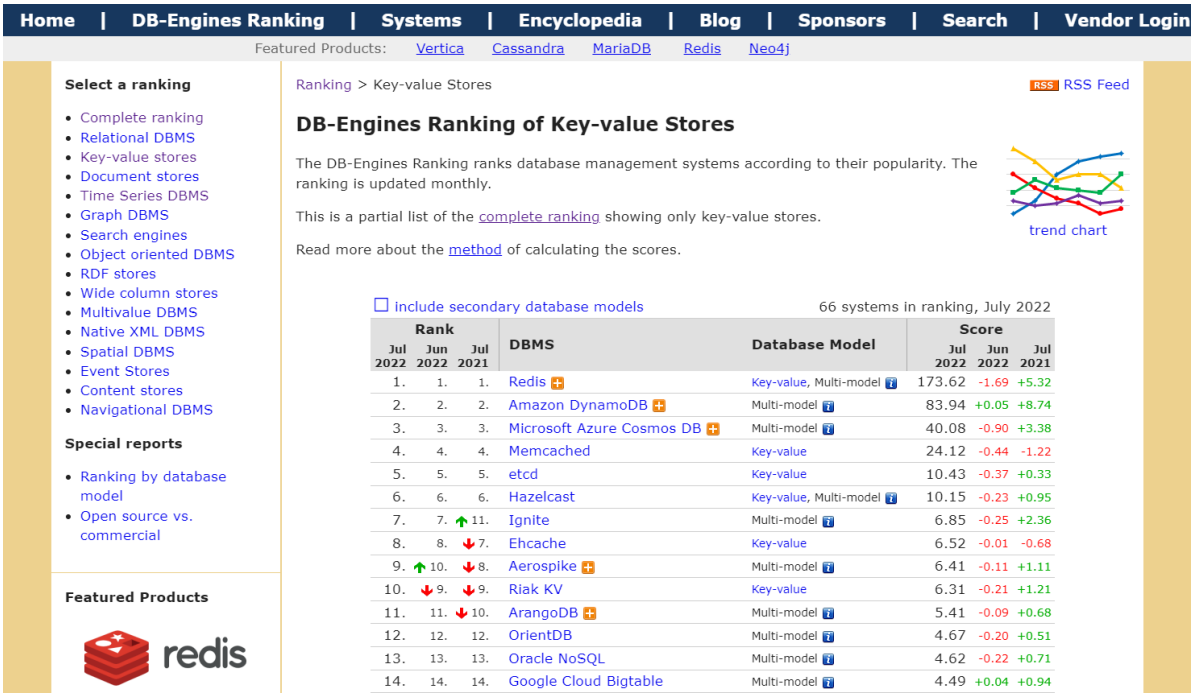
[28] Grafana Demo: https://play.grafana.org/

6.分布式键值数据库

键值存储 (key-value store)，也称为 K/V 存储或键值数据库，这是一种非关系型数据库。每个值都有一个唯一的 key 关联，也就是我们常说的 **键值对**。

常见的键值存储有 Redis, Amazon DynamoDB，Microsoft Azure Cosmos DB，Memcached，etcd 等。

你可以在 DB-Engines 网站上看到键值存储的排行。



Home | DB-Engines Ranking | Systems | Encyclopedia | Blog | Sponsors | Search | Vendor Login

Featured Products: Vertica Cassandra MariaDB Redis Neo4j

Select a ranking

- Complete ranking
- Relational DBMS
- Key-value stores
- Document stores
- Time Series DBMS
- Graph DBMS
- Search engines
- Object oriented DBMS
- RDF stores
- Wide column stores
- Multivalue DBMS
- Native XML DBMS
- Spatial DBMS
- Event Stores
- Content stores
- Navigational DBMS

Special reports

- Ranking by database model
- Open source vs. commercial

Featured Products

redis

Ranking > Key-value Stores

RSS RSS Feed

DB-Engines Ranking of Key-value Stores

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

This is a partial list of the complete ranking showing only key-value stores.

Read more about the method of calculating the scores.

trend chart

include secondary database models — 66 systems in ranking, July 2022

Rank Jul 2022	Rank Jun 2022	Rank Jul 2021	DBMS	Database Model	Score Jul 2022	Score Jun 2022	Score Jul 2021
1.	1.	1.	Redis	Key-value, Multi-model	173.62	-1.69	+5.32
2.	2.	2.	Amazon DynamoDB	Multi-model	83.94	+0.05	+8.74
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	40.08	-0.90	+3.38
4.	4.	4.	Memcached	Key-value	24.12	-0.44	-1.22
5.	5.	5.	etcd	Key-value	10.43	-0.37	+0.33
6.	6.	6.	Hazelcast	Key-value, Multi-model	10.15	-0.23	+0.95
7.	7.	↑ 11.	Ignite	Multi-model	6.85	-0.25	+2.36
8.	8.	↓ 7.	Ehcache	Key-value	6.52	-0.01	-0.68
9.	↑ 10.	↓ 8.	Aerospike	Multi-model	6.41	-0.11	+1.11
10.	↓ 9.	↓ 9.	Riak KV	Key-value	6.31	-0.21	+1.21
11.	11.	↓ 10.	ArangoDB	Multi-model	5.41	-0.09	+0.68
12.	12.	12.	OrientDB	Multi-model	4.67	-0.20	+0.51
13.	13.	13.	Oracle NoSQL	Multi-model	4.62	-0.22	+0.71
14.	14.	14.	Google Cloud Bigtable	Multi-model	4.49	+0.04	+0.94

设计要求

在这个面试的系统设计环节中，我们需要设计一个键值存储， 要满足下面的几个要求

- 每个键值的数据小于 10kB。
- 有存储大数据的能力。
- 高可用，高扩展性，低延迟。

单机版 - 键值存储

对于单个服务器来说，开发一个键值存储相对来说会比较简单，一种简单的做法是，把键值都存储在内存中的哈希表中，这样查询速度非常快。但是，由于内存的限制，把所有的数据放到内存中明显是行不通的。

所以，对于热点数据（经常访问的数据）可以加载到内存中，而其他的数据可以存储在磁盘。但是，当数据量比较大时，单个服务器仍然会很快达到容量瓶颈。

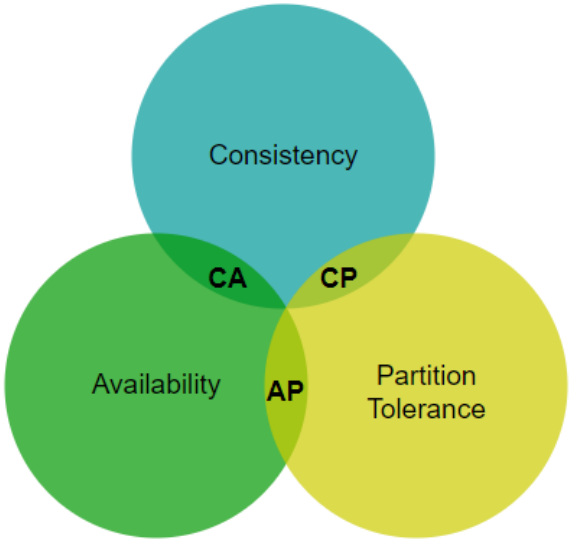
分布式 - 键值存储

分布式键值存储也叫分布式哈希表，把键值分布在多台服务器上。在设计分布式系统时，理解 CAP（一致性，可用性，分区容错性） 定理很重要。

CAP 定理

CAP 定理指出，在分布式系统中，不可能同时满足一致性、可用性和分区容错性。让我们认识一下这三个定义：

- 一致性： 无论连接到哪一个节点，所有的客户端在同一时间都会看到相同的数据。
- 可用性：可用性意味着任何请求数据的客户端都会得到响应，即使某些节点因故障下线。
- 分区容错性：分区表示两个节点之间的网络通信中断。分区容错性意味着，当存在网络分区时，系统仍然可以继续运行。



通常可以用 CAP 的两个特性对键值存储进行分类：

CP（一致性和分区容错性）系统：牺牲可用性的同时支持一致性和分区容错。

AP（可用性和分区容错性）系统：牺牲一致性的同时支持可用性和分区容错。

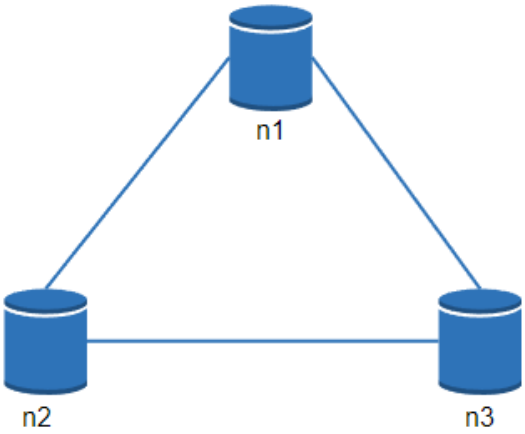
CA（一致性和可用性）系统：牺牲分区容错性的同时支持一致性和可用性。

由于网络故障是不可避免的，所以在分布式系统中，必须容忍网络分区。

让我们看一些具体的例子，在分布式系统中，为了保证高可用，数据通常会在多个系统中进行复制。假设数据在三个节点 n1, n2, n3 进行复制，如下：

理想情况

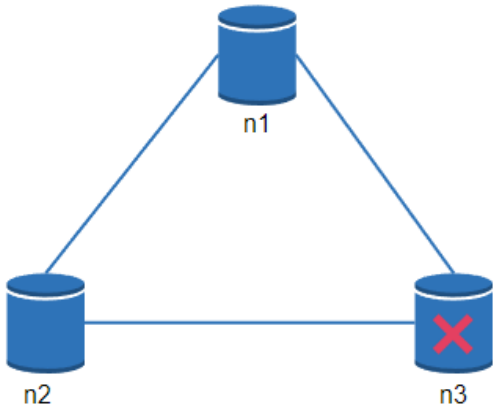
在理想的情况下，网络分区永远不会发生。写入 n1 的数据会自动复制到 n2 和 n3，实现了一致性和可用性。



现实世界的分布式系统

在分布式系统中，网络分区是无法避免的，当发生分区时，我们必须在一致性和可用性之间做出选择。

在下图中，n3 出现了故障，无法和 n1 和 n2 通信，如果客户端把数据写入 n1 或 n2，就没办法复制到 n3，就会出现数据不一致的情况。



如果我们选择一致性优先（CP系统），当 n3 故障时， 就必须阻止所有对 n1 和 n2 的写操作，避免三个节点之间的数据不一致。涉及到钱的系统通常有极高的一致性要求。

如果我们选择可用性优先（AP系统），当 n3 故障时，系统仍然可以正常的写入读取，但是可能会返回旧的数据，当网络分区恢复后，数据再同步到 n3 节点。

选择合适的 CAP 是构建分布式键值存储的重要一环。

核心组件和技术

接下来，我们会讨论构建键值存储的核心组件和技术：

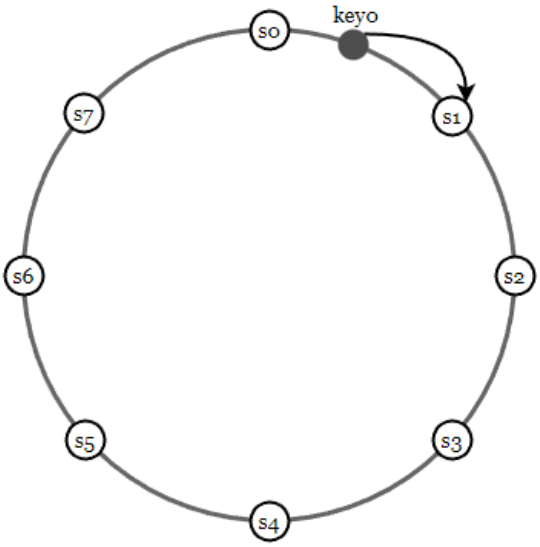
- 数据分区
- 数据复制
- 一致性
- 不一致时的解决方案
- 故障处理
- 系统架构图
- 数据写入和读取流程

数据分区

在数据量比较大场景中，把数据都存放在单个服务器明显是不可行的，我们可以进行数据分区，然后保存到多个服务器中。

需要考虑到的是，多个服务器之间的数据应该是均匀分布的，在添加或者删除节点时，需要移动的数据应该尽量少。

一致性哈希非常适合在这个场景中使用，下面的例子中，8台服务器被映射到哈希环上，然后我们把键值的 key 也通过哈希算法映射到环上，然后找到顺时针方向遇到的第一个服务器，并进行数据存储。



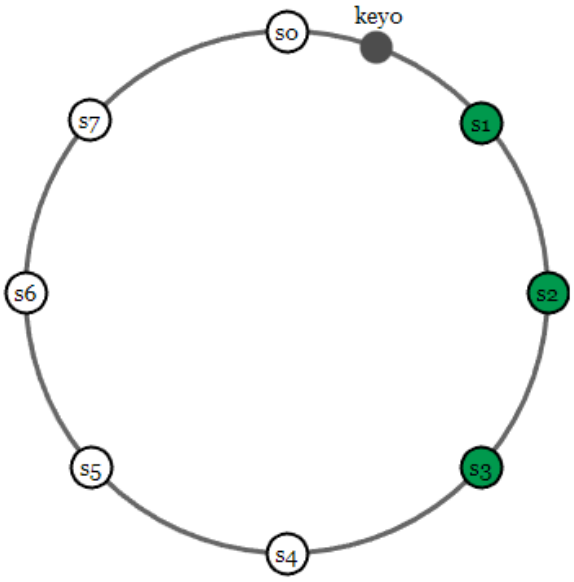
使用一致性哈希，在添加和删除节点时，只需要移动很少的一部分数据。

数据复制

为了实现高可用性和可靠性，一条数据在某个节点写入后，会复制到其他的节点，也就是我们常说的多副本。

那么问题来了，如果我们有 8 个节点，一条数据需要在每个节点上都存储吗?

并不是，副本数和节点数没有直接关系。副本数应该是一个可配置的参数，假如副本数为 3，同样可以借助一致性哈希环，按照顺时针找到 3 个节点，并进行存储，如下



一致性

因为键值数据在多个节点上复制，所以我们必须要考虑到数据一致性问题。

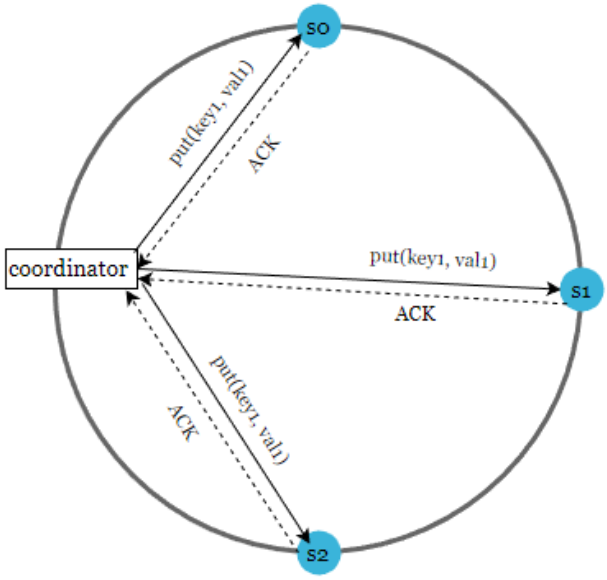
Quorum 共识算法可以保证读写操作的一致性，我们先看一下 Quorum 算法中 NWR 的定义。

N = 副本数，也叫复制因子，在分布式系统中，表示同一条数据有多少个副本。

W = 写一致性级别，表示一个写入操作，需要等待几个节点的写入后才算成功。

R = 读一致性级别，表示读取一个数据时，需要同时读取几个副本数，然后取最新的数据。

如下图，N = 3



注意，W = 1 并不意味着数据只写到一个节点，控制写入几个节点的是 N 副本数。

N = 3 表示，一条数据会写入到 3 个节点，W = 1 表示，只要收到任何节点的第一个写入成功确认消息（ACK）后，就直接返回写入成功。

这里的重点是，对 N、W、R的值进行不同的组合时，会产生不同的一致性效果。

- 当 W + R > N 的时候，通常是 N = 3, W = R = 2，对于客户端来讲，整个系统能保证强一致性，一定能返回更新后的那份数据。
- 当 W + R <= N 的时候，对于客户端来讲，整个系统只能保证最终一致性，所以可能会返回旧数据。

通过 Quorum NWR，可以调节系统一致性的程度。

一致性模型

一致性模型是设计键值存储要考虑的另外一个重要因素，一致性模型定义了数据一致性的程度。

- **强一致性：** 任何一个读取操作都会返回一个最新的数据。
- **弱一致性：** 数据更新之后，读操作可能会返回最新的值，也有可能会返回更新前的值。
- **最终一致性：** 这是弱一致性的另外一种形式。可能当前节点的值是不一致的，但是等待一段时间的数据同步之后，所有节点的值最终会保持一致。

强一致性的通常做法是，当有副本节点因为故障下线时，其他的副本会强制中止写入操作。一致性程度比较高，但是牺牲了系统的高可用。

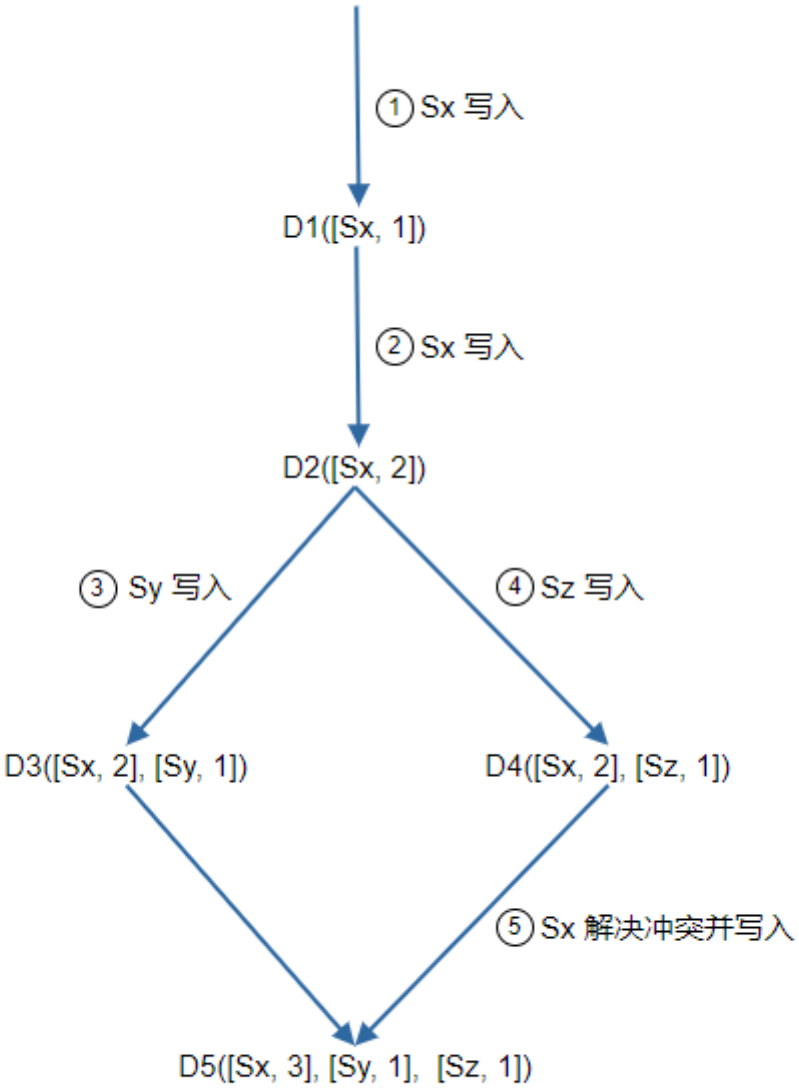
而 Dynamo 和 Cassandra 都采用了最终一致性，这也是键值存储推荐使用的一致性模型，当数据不一致时，客户端读取多个副本的数据，进行协调并返回数据。

不一致的解决方案：版本控制

多副本数据复制提供了高可用性，但是多副本可能会存在数据不一致的问题。

版本控制和向量时钟（vector clock ）是一个很好的解决方案。向量时钟是一组 [server，version] 数据，它通过版本来检查数据是否发生冲突。

假设向量时钟由 D([S1, v1], [S2, v2], ..., [Sn, vn]) 表示，其中 D 是数据项，v1 是版本计数器，下面是一个例子



1. 客户端把数据 D1 写入系统，写入操作由 Sx 处理，服务器 Sx 现在有向量时钟 D1[(Sx, 1)]。
2. 客户端把 D2 写入系统，假如这次还是由 Sx 处理，则版本号累加，现在的向量时钟是 D2([Sx, 2])。
3. 客户端读取 D2 并更新成 D3，假如这次的写入由 Sy 处理， 现在的向量时钟是D3([Sx, 2], [Sy, 1]))。
4. 客户端读取 D2 并更新成 D4，假如这次的写入由 Sz 处理，现在的向量时钟是 D4([Sx, 2], [Sz, 1]))。

5. 客户端读取到 D3 和 D4，检查向量时钟后发现冲突（因为不能判断出两个向量时钟的顺序关系），客户端自己处理解决冲突，然后再次写入。假如写入是 Sx 处理，现在的向量时钟是 D5([Sx, 3], [Sy, 1], [Sz, 1])。

注意，向量时钟只能检测到冲突，如何解决，那就需要客户端读取多个副本值自己处理了。

故障处理

在分布式大型系统中，发生故障是很常见的，接下来，我会介绍常见的故障处理方案。

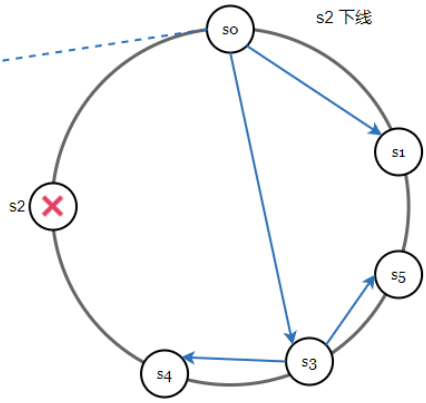
故障检测

一种很常见的方案是使用 Gossip 协议，我们看一下它的工作原理：

- 每个节点维护一个节点成员列表，其中包含成员 ID 和心跳计数器。
- 每个节点周期性地增加它的心跳计数器。
- 每个节点周期性地向一组随机节点发送心跳，这些节点依次传播到另一组节点。
- 一旦节点收到心跳，成员列表就会更新为最新信息。
- 如果在定义的周期内，发现心跳计数器的值比较小，则认为该成员离线。

成员列表

Member ID	Heartbeat counter	Time
0	10232	12:00:01
1	10224	12:00:10
2	**9908**	**11:58:02**
3	10237	12:00:20
4	10234	12:00:34

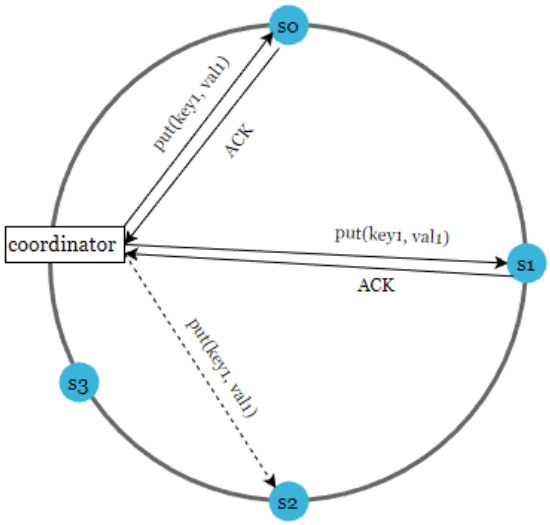


处理临时故障

通过 gossip 协议检测到故障后，为了保证数据一致性，严格的 Quorum 算法会阻止写入操作。而 sloppy quorum 可以在临时故障的情况下，保证系统的可用性。

当网络或者服务器故障导致服务不可用时，会找一个临时的节点进行数据写入，当宕机的节点再次启动后，写入操作会更新到这个节点上，保持数据一致性。

如下图所示，当 s2 不可用时，写入操作暂时由 s3 处理， 在一致性哈希环上顺时针查找到下一个节点就是s3，当 s2 重新上线时，s3 会把数据还给 s2。



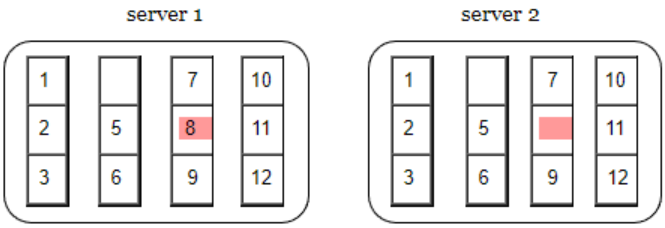
处理长时间故障

数据会在多个节点进行数据复制，假如节点发生故障下线，并且在一段时间后恢复，那么，节点之间的数据如何同步? 全量对比？明显是低效的。我们需要一种高效的方法进行数据对比和验证。

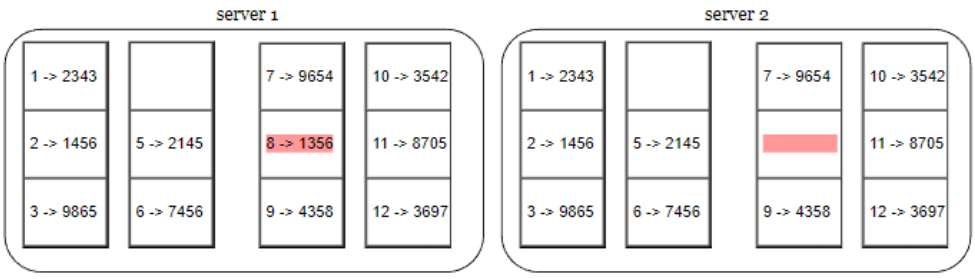
使用 Merkle 树是一个很好的解决方案，Merkle 树也叫做哈希树，这是一种树结构，最下面的叶节点包含数据或哈希值，每个中间节点是它的子节点内容的哈希值，根节点也是由它的子节点内容的哈希值组成。

下面的过程，展示了 Merkle 树是如何构建的。

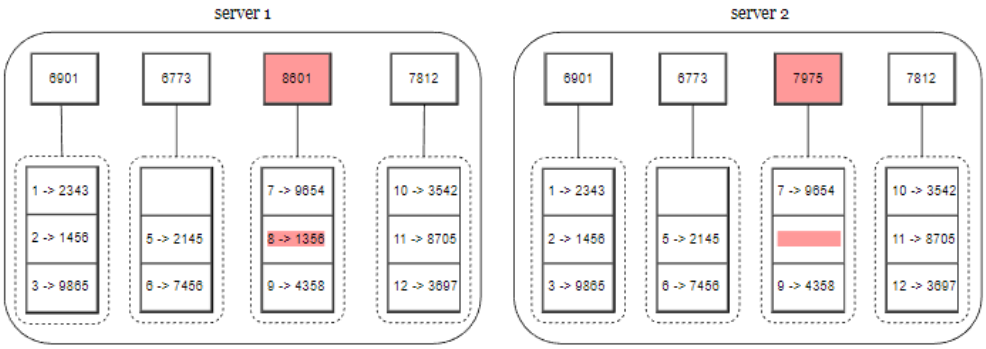
第 1 步，把键值的存储空间划分为多个桶，一个桶可以存放一定数量的键值。



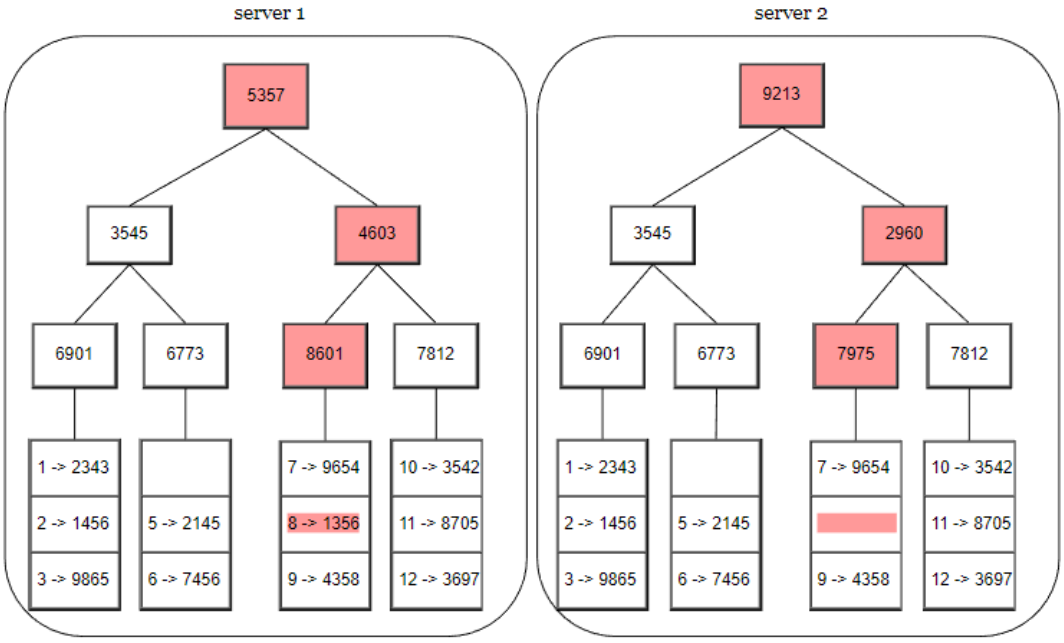
第 2 步，创建桶之后，使用哈希算法计算每个键的哈希值。



第 3 步，根据桶里面的键的哈希值，计算桶的哈希值。



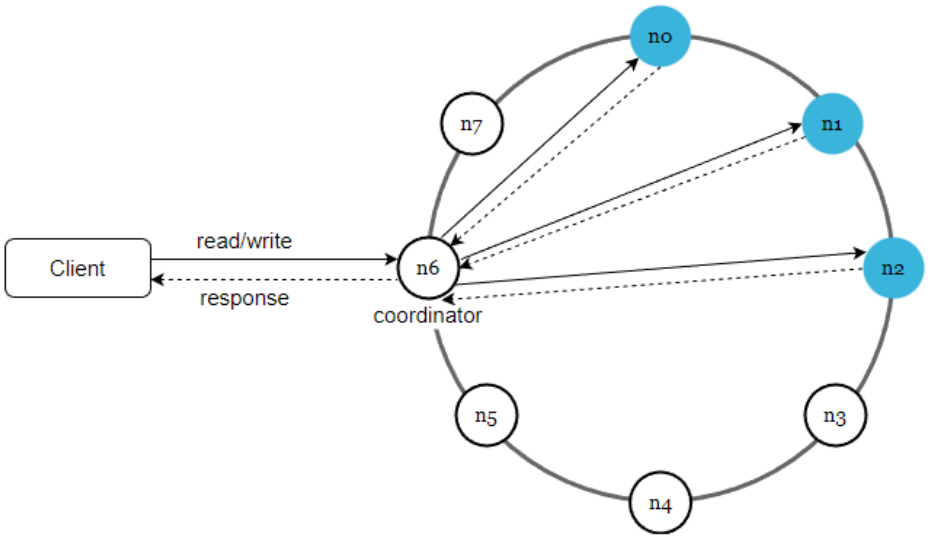
第 4 步，计算子节点的哈希值，并向上构建树，直到根节点结束。



如果要比较两个 Merkle 树，首先要比较根哈希，如果根哈希一致，表示两个节点有相同的数据。如果根哈希不一致，就遍历匹配子节点，这样可以快速找到不一致的数据，并进行数据同步。

系统架构图

我们已经讨论了设计键值存储要考虑到的技术问题，现在让我们关注一下整体的架构图，如下

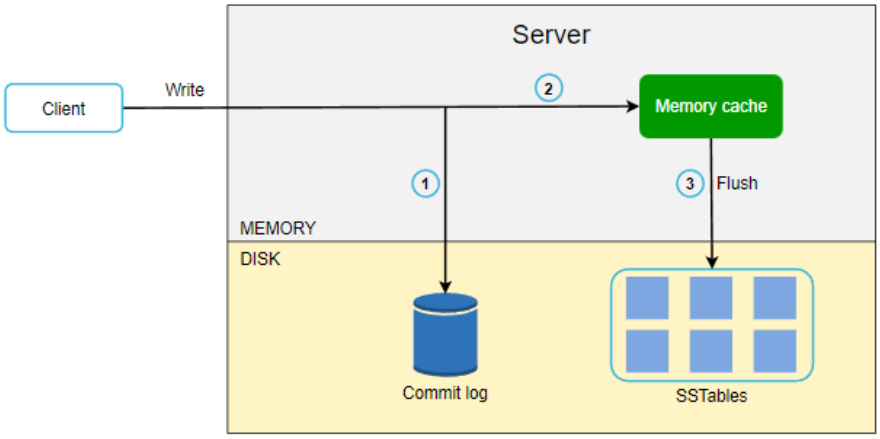


这个架构主要有下面几个特点：

- 客户端通过简单的 API 和键值存储进行通信，get (key) 和 put (key, value)。
- coordinator 协调器充当了客户端和键值存储之间的代理节点。
- 所有节点映射到了一致性哈希环上。
- 数据在多个节点上进行复制。

写入流程

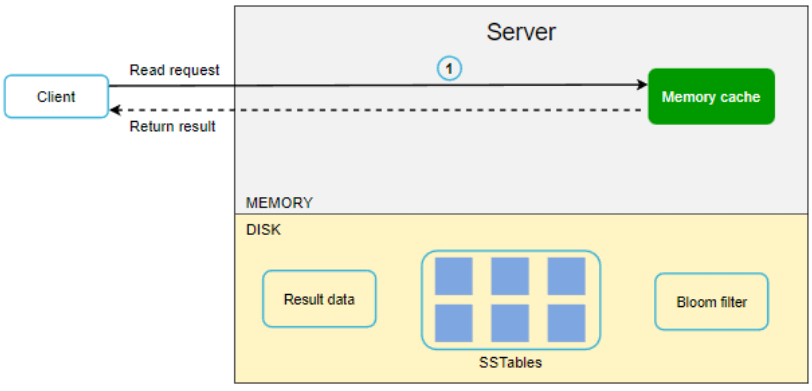
下图展示了数据写入到存储节点的过程，主要基于 Cassandra 的架构设计。



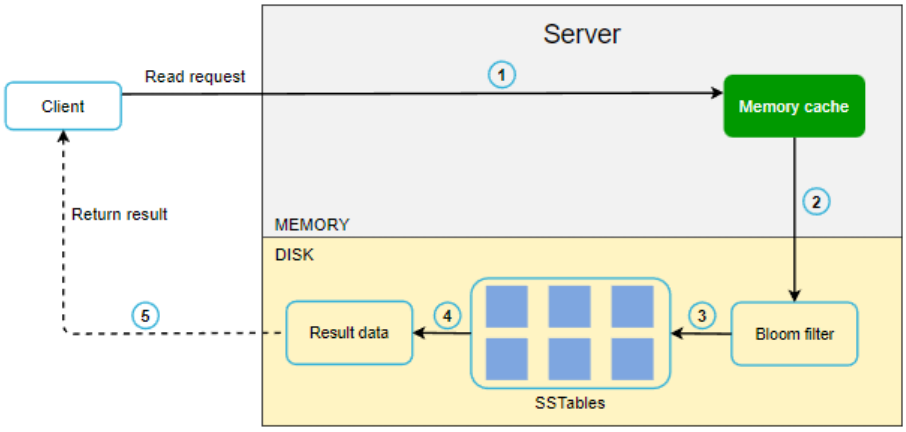
1. 写入请求首先被持久化在提交日志文件中。
2. 然后数据保存在内存缓存中。
3. 当内存已满或者达到阈值时，数据移动到本地磁盘的 SSTable，这是一种高阶数据结构，感兴趣的读者自行查阅资料了解。

读取流程

在进行数据读取时，它首先检查数据是否在内存缓存中，如果是，就把数据返回给客户端，如下图所示：



如果数据不在内存中，就会从磁盘中检索。我们需要一种高效的方法，找到数据在哪个 SSSTable 中，通常可以使用布隆过滤器来解决这个问题。



1. 系统首先检查数据是否在内存缓存中。
2. 如果内存中没有数据，系统会检查布隆过滤器。
3. 布隆过滤器可以快速找出哪些 SSTables 可能包含密钥。
4. SSTables 返回数据集的结果。
5. 结果返回给客户端。

Reference

[0] System Design Interview Volume 2: https://www.amazon.com/System-Design-Interview-Insiders-Guide/dp/1736049119

[1] Amazon DynamoDB: https://aws.amazon.com/dynamodb/

[2] memcached: https://memcached.org/

[3] Redis: https://redis.io/

[4] Dynamo: Amazon's Highly Available Key-value Store: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[5] Cassandra: https://cassandra.apache.org/

[6] Bigtable: A Distributed Storage System for Structured Data: https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

[7] Merkle tree: https://en.wikipedia.org/wiki/Merkle_tree

[8] Cassandra architecture: https://cassandra.apache.org/doc/latest/architecture/

[9] SStable: https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/

[10] Bloom filter https://en.wikipedia.org/wiki/Bloom_filter

7.S3 对象存储

在本文中，我们设计了一个类似于 Amazon Simple Storage Service (S3) 的对象存储服务。S3 是 Amazon Web Services (AWS) 提供的一项服务， 它通过基于 RESTful API 的接口提供对象存储。根据亚马逊的报告，到 2021 年，有超过 100 万亿个对象存储在 S3 中。

在深入设计之前，有必要先回顾一下存储系统和相关的术语。

存储系统

在高层次上，存储系统分类三大类：

- 块存储
- 文件存储
- 对象存储

块存储

块存储最早出现在 1960 年。常见的物理存储设备，比如常说的 HDD 和 SSD 都属于块存储。块存储直接暴露出来卷或者盘，这是最灵活，最通用的存储形式。

块存储不局限于物理连接的存储，也可以通过网络、光纤和 iSCSI 行业标准协议连接到服务器。从概念上讲，网络附加块存储仍然暴露原始块，对于服务器来说，它的工作方式和使用物理连接的块存储是相同的。

文件存储

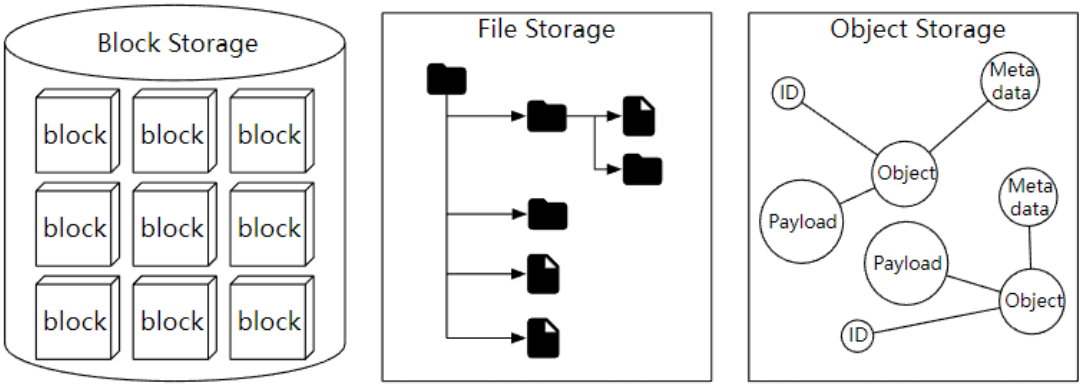
文件存储在块存储的上层，提供了更高级别的抽象，文件存储不需要处理管理块、格式化卷等，所以它处理文件和目录更简单，数据文件存储在分层目录结构。

对象存储

对象存储相对来说比较新，为了高持久性，大规模和低成本而牺牲性能，这是一个非常刻意的权衡。对象存储针对的是相对 “冷” 的数据，主要用于归档和备份。对象存储把所有的数据作为对象存储在平面结构中，没有分层的目录结构。

通常提供了 RESTful API 用来支持数据访问，和其他的存储相比，它是比较慢的，大多云服务商都提供了对象存储的产品，比如 AWS S3, Azure Blob 存储等。

对比



术语

要设计一个类似于 S3 的对象存储，我们需要先了解一些对象存储的核心概念。

- 桶 （Bucket），桶是对象的逻辑容器，存储桶名称是全局唯一的。
- 对象（Object），对象时我们存储在桶中的单个数据，它由对象数据和元数据组成。对象可以是我们存储的任何字节序列，元数据是一组描述对象的键值对。
- 版本控制 （Versioning）， 数据更新时，允许多版本共存。
- 统一资源标识符 (URI)，对象存储提供了 RESTful API 来访问资源，所以每个资源都有一个URI 唯一标识。

- 服务等级协议 (SLA)，SLA 是服务提供商和客户之间的协议。比如 AWS S3 对象存储，提供了 99.9 的可用性，以及夸张的 99.999999999% （11个9） 的数据持久性。

设计要求

在这个面试的系统设计环节中，需要设计一个对象存储，并且要满足下面的几个要求。

- 基础功能，桶管理，对象上传和下载，版本控制。
- 对象数据有可能是大对象（几个 GB），也可能是小对象（几十 kb）。
- 一年需要存储 100 PB 的数据。
- 服务可用性 99.99% （4个9），数据持久性 99.9999 % （6个 9）。
- 需要比较低的存储成本。

对象存储的特点

在开始设计对象存储之前，你需要了解它的下面这些特点。

对象不变性

对象存储和其他两种存储的主要区别是，存储对象是不可变的，允许进行删除或者完全更新，但是不能进行增量修改。

键值存储

我们可以使用 URI 来访问对象数据，对象的 URI 是键，对象的数据是值，如下

```
Request:
GET /bucket1/object1.txt HTTP/1.1

Response:
HTTP/1.1 200 OK
Content-Length: 4567

[4567 bytes of object data]
```

写一次，读多次

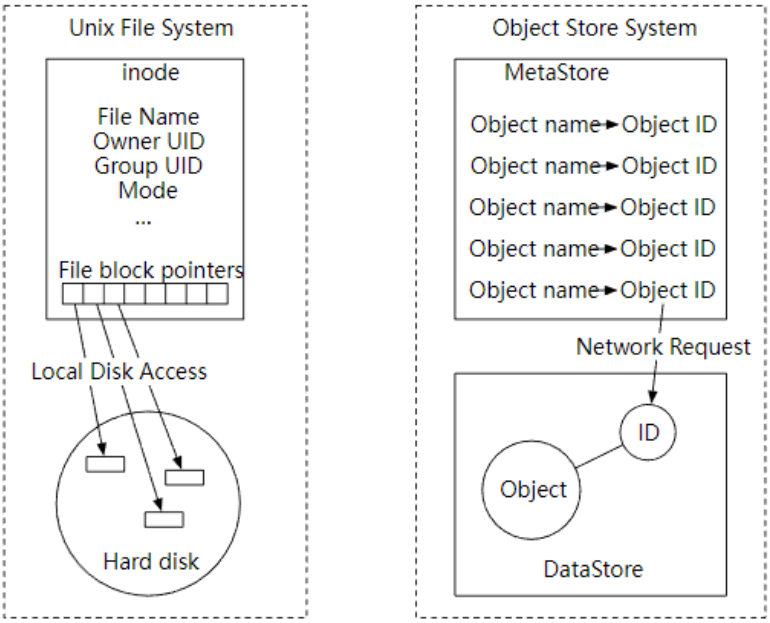
对象数据的访问模式是一次写入，多次读取。根据 LinkedIn 做的研究报告，95 %的请求是读取操作。

【Ambry: LinkedIn’s Scalable Geo-Distributed Object Store】

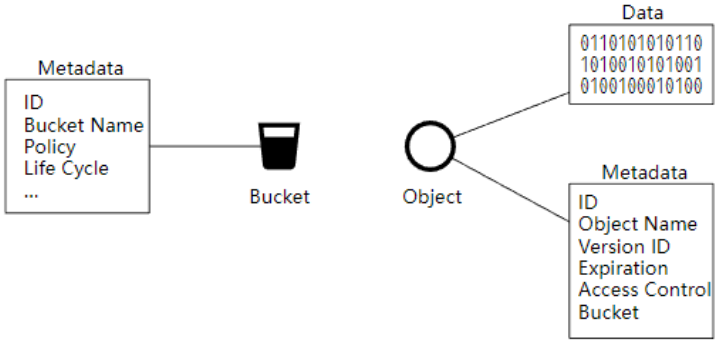
支持小型和大型对象

对象存储的设计理念和 UNIX 文件系统的设计理念非常相似。在 UNIX 中，当我们在本地文件系统中保存文件时，它不会把文件名和文件数据一起保存。那是怎么做的呢？它把文件名存储在 inode 的数据结构中，把文件数据存储在不同的磁盘位置。inode 包含一个文件块指针列表，这些指针指向文件数据的磁盘位置。当我们访问本地文件时，首先会获取 inode 中的元数据。然后我们按照文件块指针来读取磁盘的文件数据。

对象存储的工作方式也是如此，元数据和数据存储分离，如下

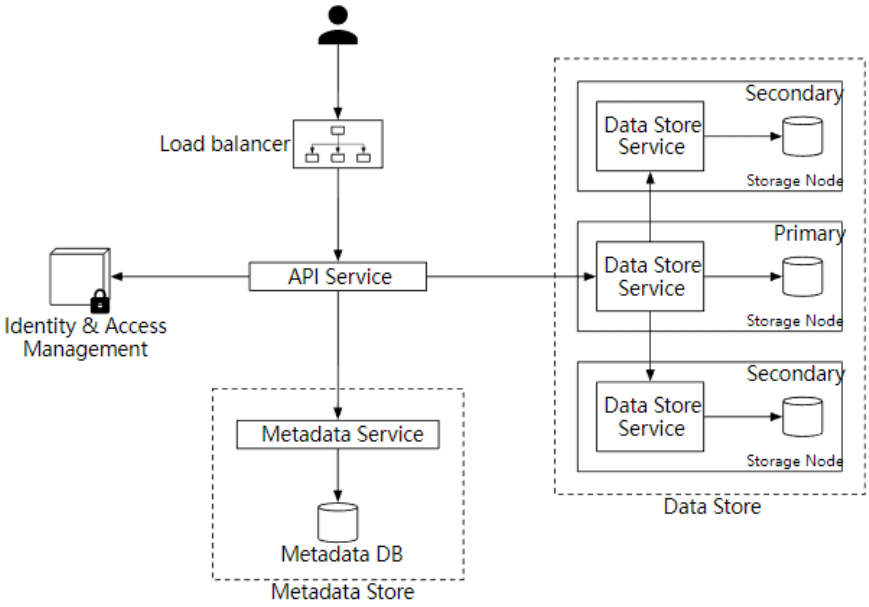


看一看我们的存储桶和对象的设计



整体设计

下图显示了对象存储的整体设计。

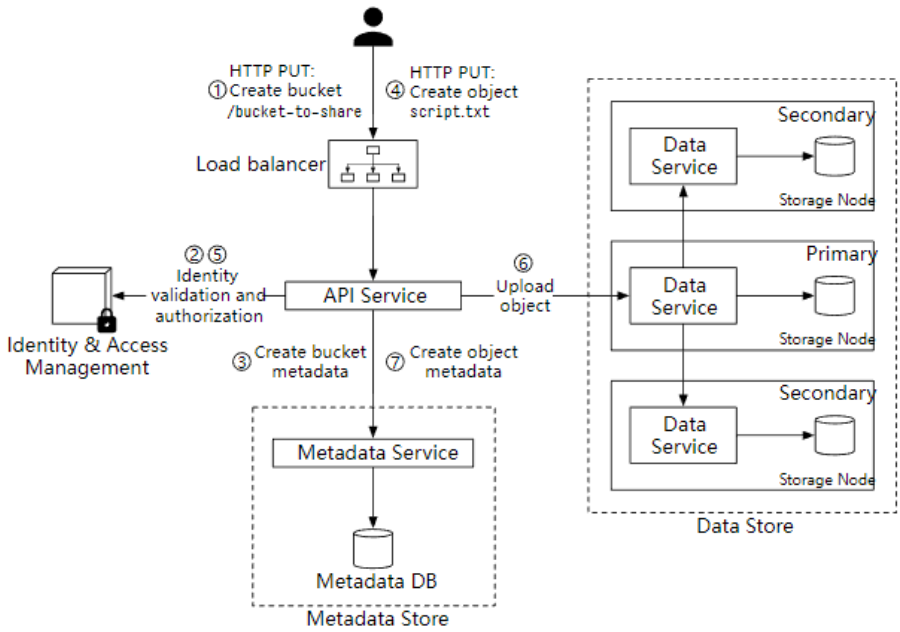


- Load balancer 负载均衡，向多个 API 服务分发 RESTful API 请求。
- API Service，编排身份验证服务，元数据服务和存储服务，它是无状态的，可以很好的支持水平扩展。
- Identity & Access Management （IAM），身份和访问管理，这是处理身份验证、授权和访问控制的服务。
- Data Store 数据存储，存储和检索对象数据，所有和数据有关的操作都是基于对象 ID（UUID）。
- Metadata Service 元数据服务，存储对象的元数据。

接下来我们一起来探索对象存储中的一些重要的工作流程。

- 上传对象
- 下载对象
- 版本控制

上传对象



在上面的流程中，我们首先创建了一个名为 "bucket-to-share" 的存储桶，然后把一个名为 "script.txt" 的文件上传到这个桶。

1. 客户端发送一个创建 “bucket-to-share” 桶的 HTTP PUT 请求，经过负载均衡器转发到 API 服务。
2. API 服务调用 IAM 确保用户已获得授权并且有 Write 权限。
3. API 服务调用元数据服务，创建存储桶，并返回成功给客户端。
4. 客户端发送创建 “script.txt” 对象的 HTTP PUT 请求。
5. API 服务验证用户的身份并确保用户对存储桶具有 Write 权限。
6. API 服务把 HTTP 请求发到到数据存储服务，完成存储后返回对象的 UUID。
7. 调用元数据服务并创建元数据项，格式如下

object_name	object_id	bucket_id
script.txt	239D5866-0052-00F6-014E-C914E61ED42B	82AA1B2E-F599-4590-B5E4-1F51AAE5F7E4

上传数据的 Http 请求示例如下

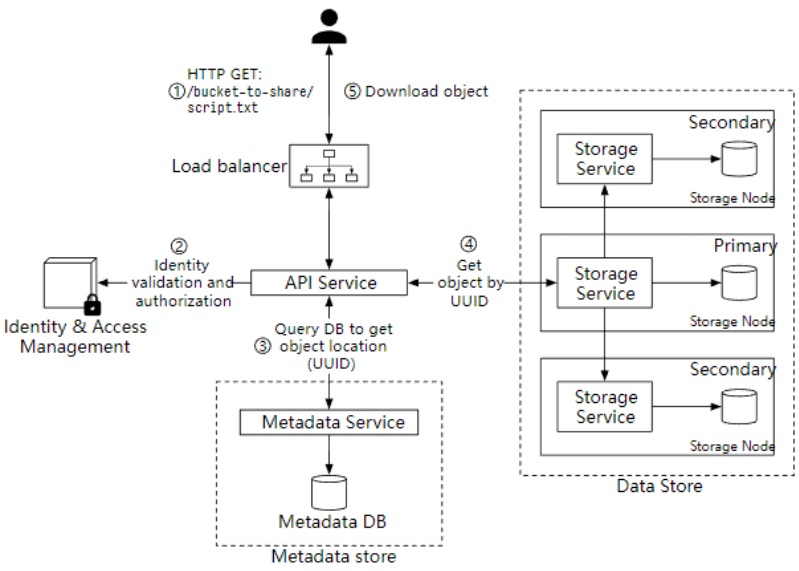
```
PUT /bucket-to-share/script.txt HTTP/1.1
Host: foo.s3example.org
Date: Sun, 12 Sept 2021 17:51:00 GMT
Authorization: authorization string
Content-Type: text/plain
Content-Length: 4567
x-amz-meta-author: Alex

[4567 bytes of object data]
```

下载对象

存储对象可以通过 HTTP GET 请求进行下载，示例如下

```
GET /bucket-to-share/script.txt HTTP/1.1
Host: foo.s3example.org
Date: Sun, 12 Sept 2021 18:30:01 GMT
Authorization: authorization string
```

下载流程图



1. 客户端发送 GET 请求，GET /bucket-to-share/script.txt
2. API 服务查询 IAM 验证用户是否有对应桶的读取权限。
3. 验证后，API 服务会从元数据服务中获取对象的 UUID。
4. 通过 对象的 UUID 从数据存储中获取相应的对象。
5. API 服务返回对象给客户端。

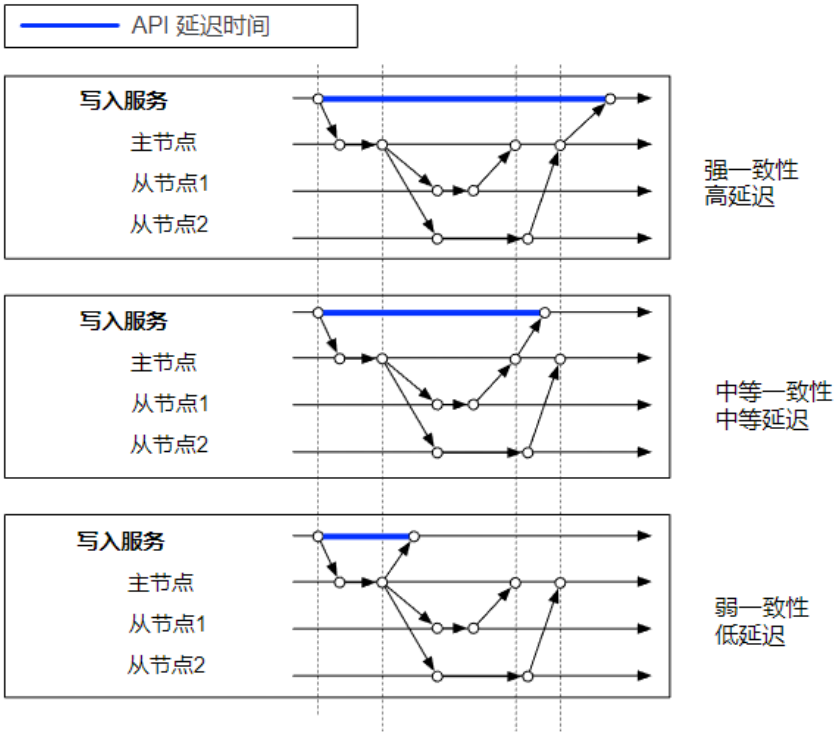
深入设计

接下来，我们会讨论下面几个比较重要的部分。

- 数据一致性
- 元数据
- 版本控制
- 优化大文件的上传
- 垃圾收集 GC

数据一致性

对象数据只存放在单个节点肯定是不行的，为了保证高可用，需要把数据复制到多个节点。这种情况下，我们需要考虑到一致性和性能问题。



保证强一致性就要牺牲性能，如果性能要求比较高时，可以选择弱一致性。鱼和熊掌不可兼得。

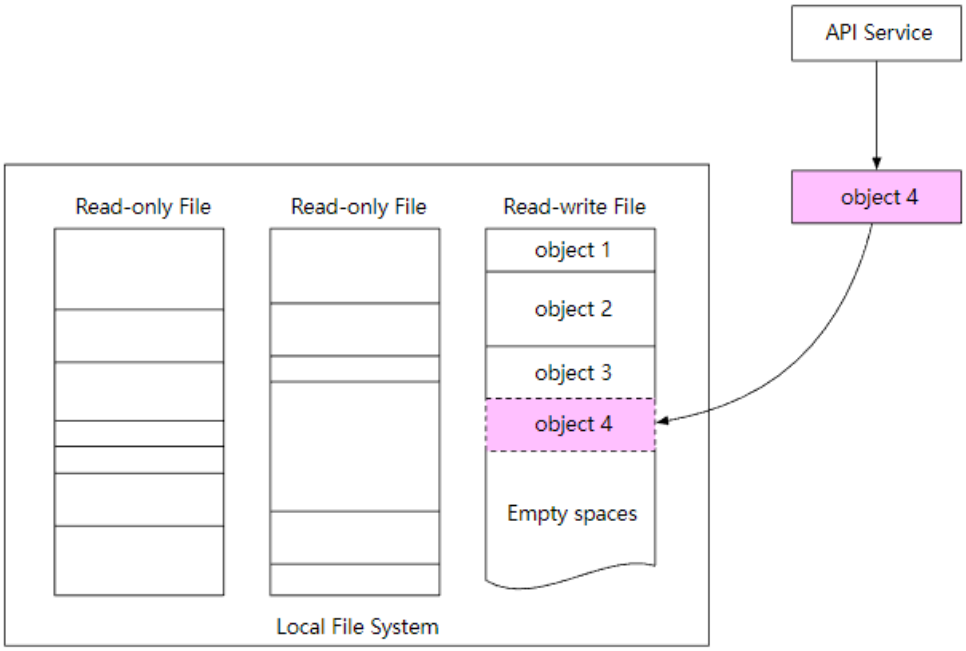
数据存储方式

对于数据存储，一个简单的方式是把每个对象都存储在一个独立的文件中，这样当然是可以的。但是，当有大量的小型文件时，会有下面两个问题。

第一个问题是，会浪费很多数据块。文件系统把文件存储在磁盘块中，磁盘块的大小在卷初始化的时候就固定了，一般是 4 kb。所以，对于小于 4kb 的文件，它也会占满整个磁盘块。如果文件系统中保存了大量的小文件，那就会就会有很多浪费。

第二个问题是，系统的 inode 容量是有限的。文件系统把文件元数据存储在 inode 特殊类型的磁盘块中。对于大多数文件系统，inode 的数量在磁盘初始化时是固定的。所以有大量的文件时，要考虑到 inode 容量满的问题。

为了解决这个问题，我们可以把很多小文件合并到一个更大的文件中。从概念上讲，类似于预写日志（WAL）。当我们保存一个对象时，它被附加到一个现有的文件中。文件大小达到一定值（比如说 1 GB）后，创建一个新的文件来存储对象，下图解释了它的工作流程。



数据持久性

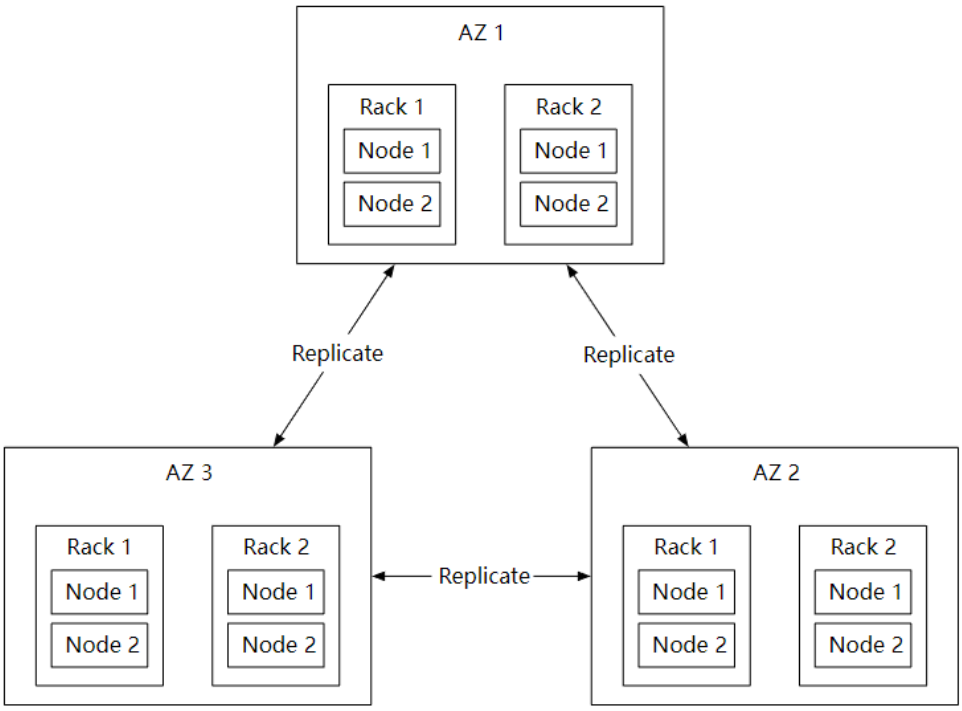
对存储系统来说，数据持久性非常重要，如何设计出一个 6 个 9 (99.9999%) 持久性 的存储系统?

硬件故障和故障域

无论使用哪种存储，硬件故障都是不可避免的。所以为了数据持久性，需要把数据复制到多个硬盘中。

假设硬盘的年故障率是 0.81 %, 当然不同的型号和品牌这些是不一样的，那个我们需要三个数据副本，1-(0.0081)^3=~0.999999, 才可以满足要求。

另外，我们还需要考虑到不同故障域的影响。这样可以在极端情况下，带来更好的可靠性，比如大规模停电，自然灾害等。

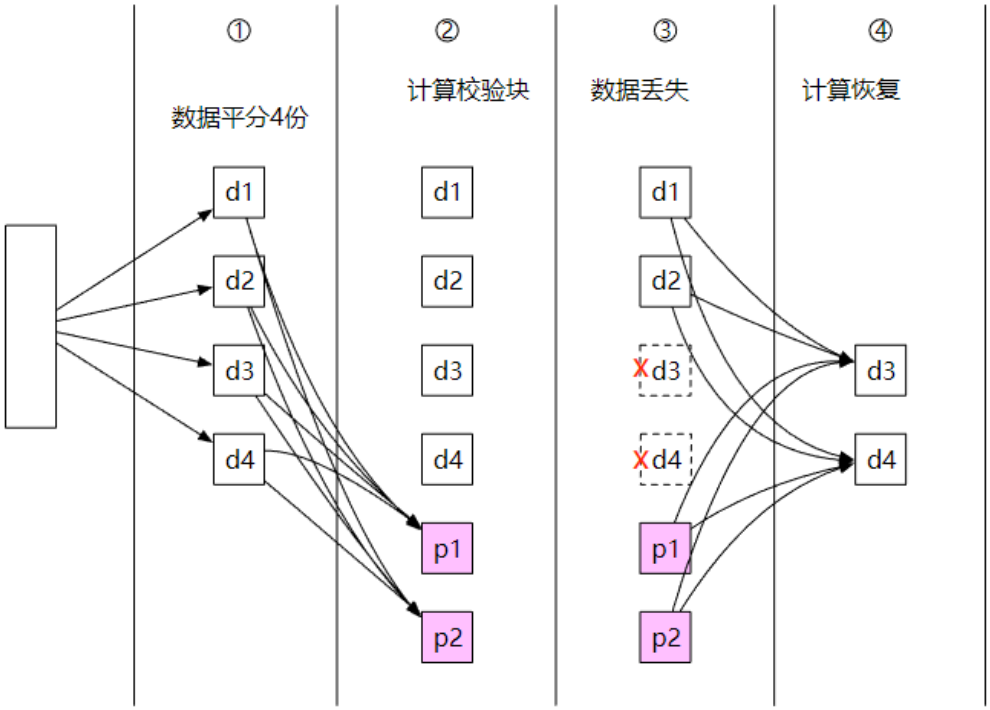


Erasure Coding 纠删码

上面提到，我们用三个完整的数据副本可以提供大概 6 个 9 的数据持久性，但是，这样的成本太高了。

还能不能优化呢？我们可以使用纠删码技术，它的原理其实很简单，假设现在有 a 和 b 两条数据，进行异或 （XOR）运算后得到 c，a ^ b = c，而 b = c ^ a，a = c ^ b，所以这三条数据丢失任意一条数据，都可以通过剩余两条数据计算出丢失数据。

下面是一个 4 +2 纠删码的例子。



1. 数据被分成四个大小均匀的数据块 d1、d2、d3 和 d4。
2. 使用 Reed-Solomon 数学公式计算校验块，比如

```
p1 = d1 + 2*d2 - d3 + 4*d4
p2 = -d1 + 5*d2 + d3 - 3*d4
```

3. 节点崩溃，导致数据 d3 和 d4 丢失。
4. 通过数据公式和现有数据，计算出丢失的数据并恢复。

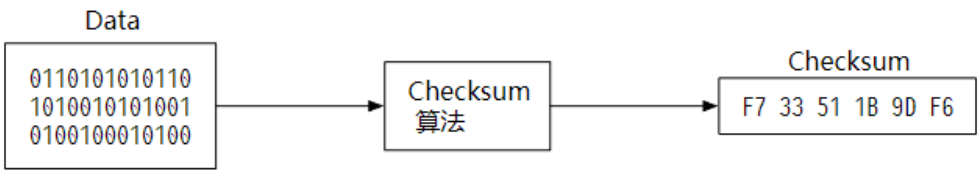
```
d3 = 3*p1 + 4*p2 + d1 - 26*d2
d4 = p1 + p2 - 7*d2
```

和多副本复制相比，纠删码占用的存储空间更少。但是，在进行丢失数据恢复时，它需要先根据现有数据计算出丢失数据，这也消耗了 CPU 资源。

数据完整性校验

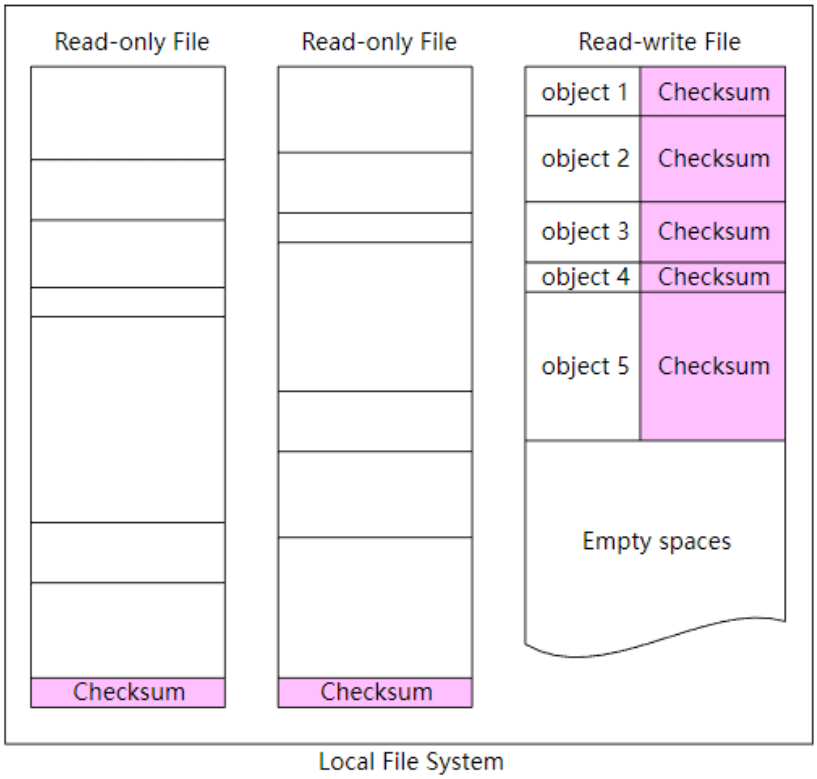
纠删码技术在保证数据持久性的同时，也降低的存储成本。接下来，我们可以继续解决下一个难题：数据损坏。

我们可以给数据通过 Checksum 算法计算出校验和。常见的 checksum 算法有 MD5, SHA1 等。



当需要验证数据时，只需要对比校验和即可，如果不一致，说明文件数据发生了改变。

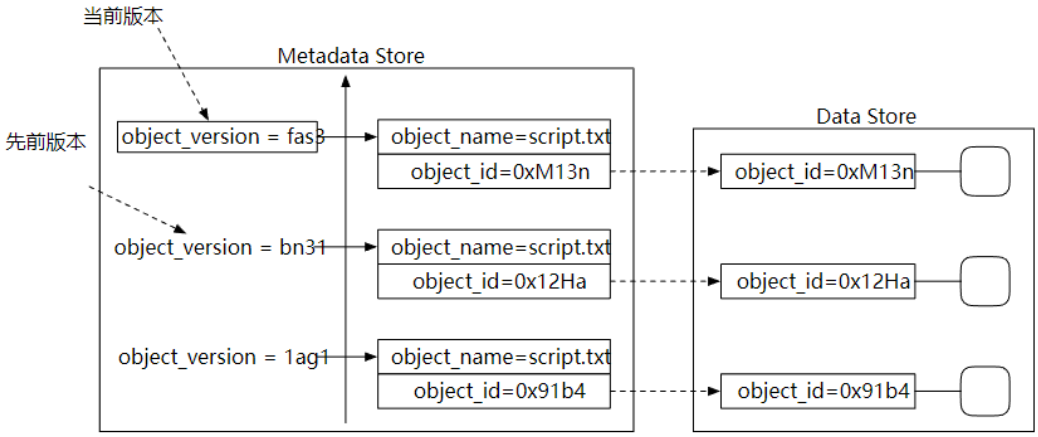
我们同样可以把校验和添加到存储系统中，对于读写文件，每个对象都计算校验和，而对于只读文件，只需要在文件的末尾添加上整个文件的校验和即可。



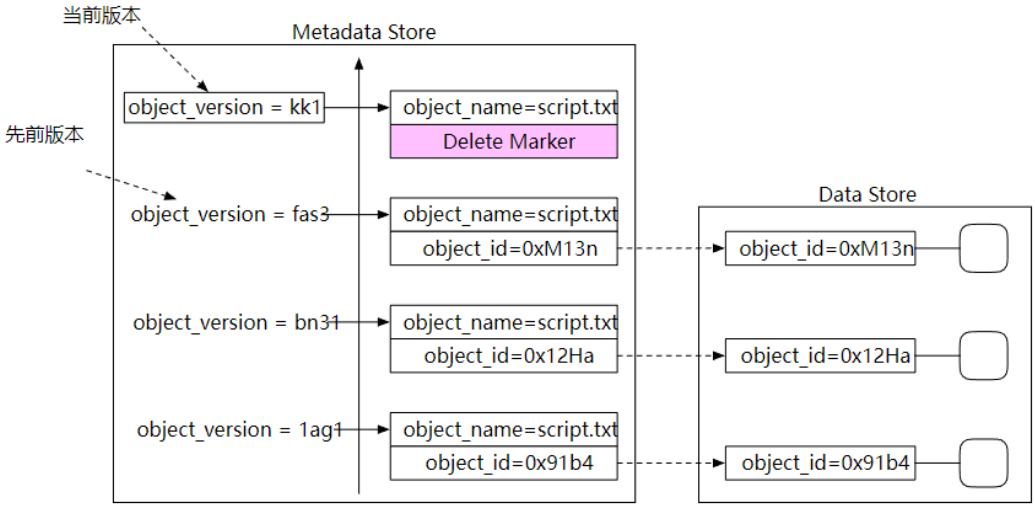
版本控制

版本控制可以让一个对象的多个版本同时保存在存储桶中。这样的好处是，我们可以恢复意外删除或者覆盖的对象。

为了支持版本控制，元数据存储的列表中需要有一个 object_version 的列。上传对象文件时，不是直接覆盖现有的记录，而是插入一个新记录。



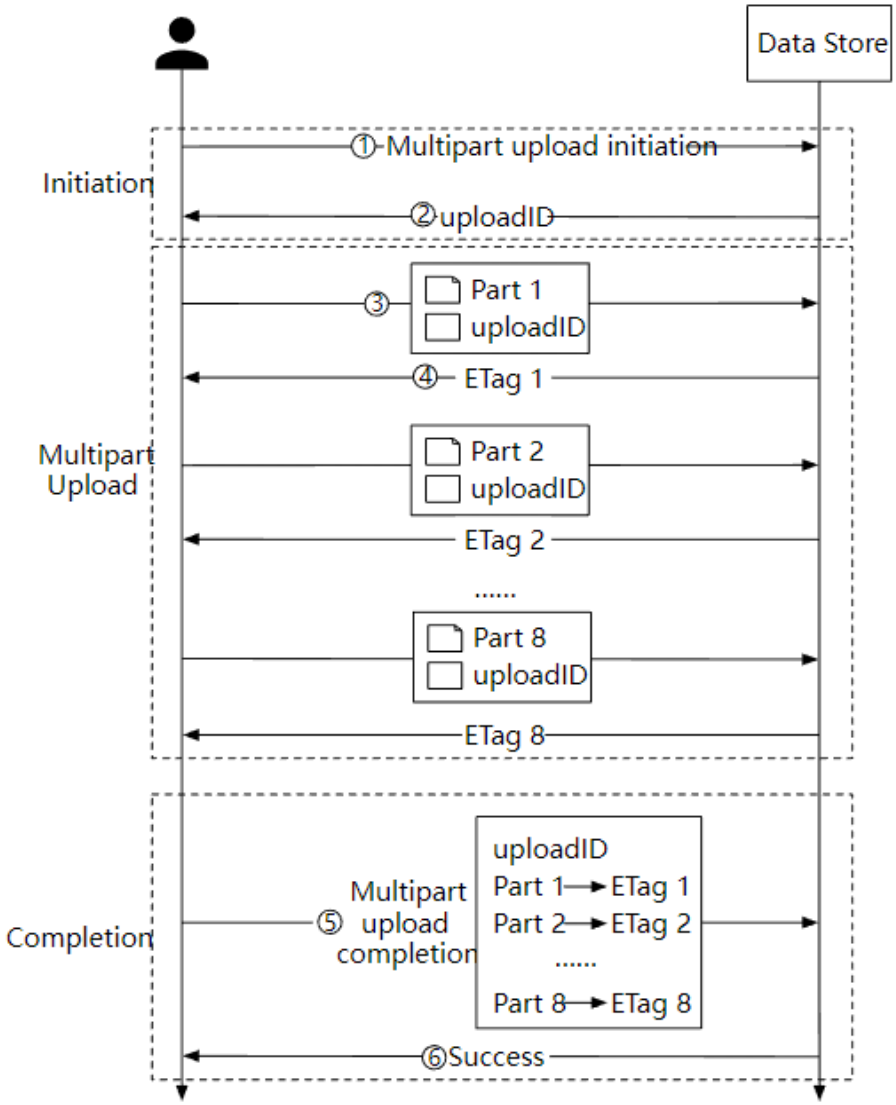
当进行对象删除的时候，不需要删除这条记录，而是添加一个删除标记即可，然后等垃圾收集器自动处理它。



优化大文件上传

对于比较大的对象文件（可能有几个 GB），上传可能需要较长的时间。如果在上传过程中网络连接失败，就要重新进行上传了。

为了解决这个问题，我们可以使用分段上传，上传失败时可以快速恢复。



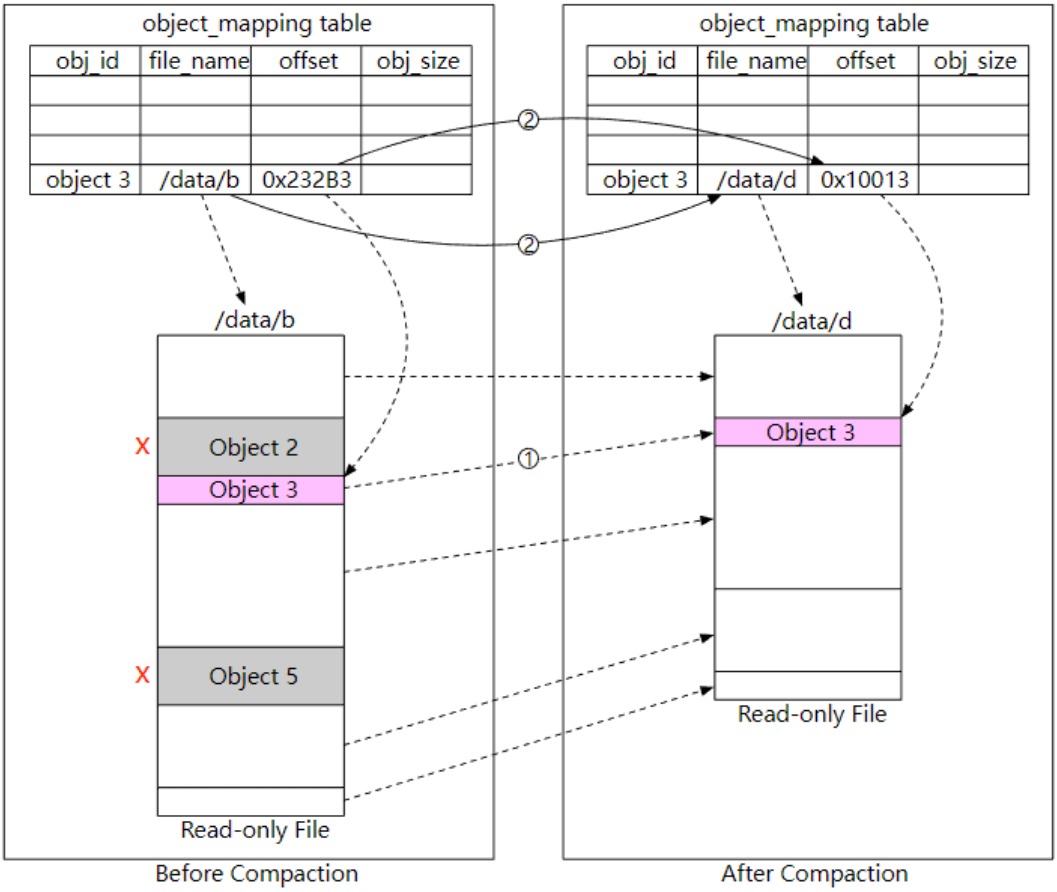
1. 客户端调用对象存储服务发起分段上传请求。
2. 数据存储服务返回一个唯一的 uploadID。
3. 客户端把大文件拆分为小对象并开始上传，假设文件大小是 1.6 GB，每个部分的大小是 200 MB，客户端上传第一部分和 uploadID 。
4. 上传第一部分后，数据存储服务会返回一个 ETag，本质上它是第一部分的 md5 校验和，客户端通过它来判断数据是否发生了更改，如果是则重新上传。
5. 当每个部分都上传成功后，客户端发送一个分段上传成功的请求。
6. 数据存储服务组装小对象为大文件，并返回一个成功消息。

垃圾收集 GC

垃圾收集是自动回收不再使用的存储空间的过程，数据可能变成垃圾的几种方式：

- 延迟删除的对象，对象在删除时标记成已删除，但实际上还没有删除。
- 孤儿数据，比如上传一半的数据。
- 损坏的数据。

对于需要删除的对象，我们使用压缩机制定期清理，下图显示了它的工作流程。



1. 垃圾收集器把对象 “/data/b”复制到一个名为“/data/d”的新文件中。这里会跳过对象 2 和 5，因为它们的删除标志都是 true。
2. 复制完所有的对象后，垃圾收集器会更新 object_mapping 表，指向新的文件地址，然后删除掉旧的文件。

总结

在本文中，介绍了类似于 S3 的对象存储，比较了块存储、文件存储和对象存储之间的区别，设计了对象上传，对象下载，版本控制功能，并讨论了两种提高可靠性和持久性的方法：复制和纠删码，最后介绍了对象存储的垃圾收集的工作流程。

希望这篇设计对象存储的文章对大家有用！

Reference

[0] System Design Interview Volume 2: https://www.amazon.com/System-Design-Interview-Insiders-Guide/dp/1736049119

[1] Fibre channel: https://en.wikipedia.org/wiki/Fibre_Channel

[2] iSCSI: https://en.wikipedia.org/wiki/ISCSI

[3] Server Message Block: https://en.wikipedia.org/wiki/Server_Message_Block

[4] Network File System: https://en.wikipedia.org/wiki/Network_File_System

[5] Amazon S3 Strong Consistency: https://aws.amazon.com/s3/consistency/

[6] Serial Attached SCSI: https://en.wikipedia.org/wiki/Serial_Attached_SCSI

[7] AWS CLI ls command: https://docs.aws.amazon.com/cli/latest/reference/s3/ls.html

[8] Amazon S3 Service Level Agreement: https://aws.amazon.com/s3/sla/

[9] Ambry: LinkedIn's Scalable Geo-Distributed Object Store: https://assured-cloud-computing.illinois.edu/files/2014/03/Ambry-LinkedIns-Scalable-GeoDistributed-Object-Store.pdf

[10] inode: https://en.wikipedia.org/wiki/Inode

[11] Ceph's Rados Gateway: https://docs.ceph.com/en/pacific/radosgw/index.html

[12] grpc: https://grpc.io/

[13] Paxos: https://en.wikipedia.org/wiki/Paxos_(computer_science)

[14] Raft: https://raft.github.io/

[15] Consistent hashing: https://www.toptal.com/big-data/consistent-hashing

[16] RocksDB: https://github.com/facebook/rocksdb

[17] SSTable: https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/

[18] B+ tree: https://en.wikipedia.org/wiki/B%2B_tree

[19] SQLite: https://www.sqlite.org/index.html

[20] Data Durability Calculation: https://www.backblaze.com/blog/cloud-storage-durability/

[21] Rack: https://en.wikipedia.org/wiki/19-inch_rack

[22] Erasure Coding: https://en.wikipedia.org/wiki/Erasure_code

[23] Reed–Solomon error correction: https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction

[24] Erasure Coding Demystified: https://www.youtube.com/watch?v=Q5kVuM7zEUI

[25] Checksum:https://en.wikipedia.org/wiki/Checksum

[26] Md5: https://en.wikipedia.org/wiki/MD5

[27] Sha1: https://en.wikipedia.org/wiki/SHA-1

[28] Hmac: https://en.wikipedia.org/wiki/HMAC

[29] TIMEUUID: https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/timeuuid_functions_r.html