IT瞭望 月刊 No.40 2018年
期刊号第七期 7月5日出版
IT OUTLOOK

战略号刊

FUTURE SCIENCE AND TECHNOLOGY

目录

◎数据库与缓存

◎一致性

◎方法论

◎漫谈

缓存和数据库，应该先操作哪个？

缓存存储，也是数据的冗余。

（1）数据库访问数据，磁盘 IO，慢；

（2）缓存里访问数据，存操作，快；

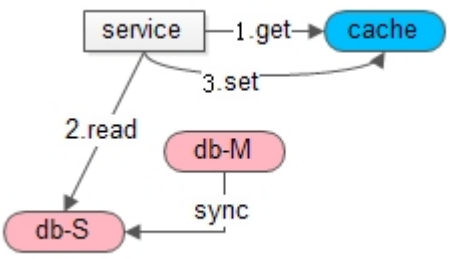
（3）数据库里的热数据，可在缓存冗余一份；

（4）先访问缓存，如果命中，能大大的提升访问速度，降低数据库压力；

以上四点是缓存的核心读加速原理。但是，一旦没有命中缓存，或者一旦涉及写操作，流程会比没有缓存更加复杂，这些是今天要分享的话题。

读操作，如果没有命中缓存，流程是怎么样的？

如下图所示



（1）尝试从缓存 get 数据，结果没有命中；

（2）从数据库获取数据，读从库，读写分离；

（3）把数据 set 到缓存，未来能够命中缓存；

读操作的流程应该没有歧义。

写操作，流程是怎么样的？

写操作，既要操作数据库中的数据，又要操作缓存里的数据。

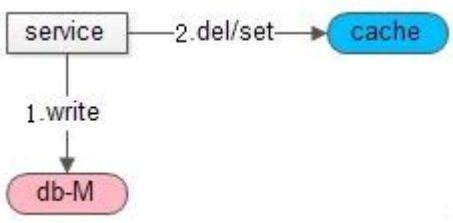
这里，有两个方案：

（1）先操作数据库，再操作缓存；

（2）先操作缓存，再操作数据库；

并且，希望保证两个操作的原子性，要么同时成功，要么同时失败。这演变为一个分布式事务的问题，保证原子性十分困难，很有可能出现一半成功，一半失败，接下来看下，当原子性被破坏的时候，分别会发生什么。

一、先操作数据库，再操作缓存



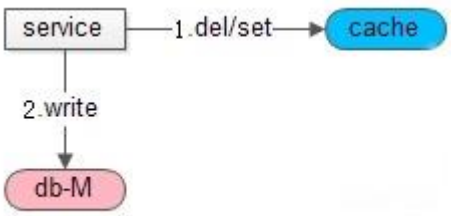
如上图，正常情况下：

（1）先操作数据库，成功；

（2）再操作缓存（delete 或者 set），也成功；

但如果这两个动作原子性被破坏：第一步成功，第二步失败，会导致，数据库里是新数据，而缓存里是旧数据，业务无法接受。

二、先操作缓存，再操作数据库



如上图，正常情况下：

（1）先操作缓存（delete 或者 set），成功；

（2）再操作数据库，也成功；

如果原子性被破坏，会发生什么呢？

这里又分了两种情况：

（1）操作缓存使用 set

（2）操作缓存使用 delete

使用 set 的情况：第一步成功，第二步失败，会导致，缓存里是 set 后的数据，数据库里是之前的数据，数据不一致，业务无法接受。并且，一般来说，数据最终以数据库为准，写缓存成功，其实并不算成功。

使用 delete 的情况：第一步成功，第二步失败，会导致，缓存里没有数据，数据库里是之前的数据，数据没有不一致，对业务无影响。只是下一次读取，会多一次 cache miss。

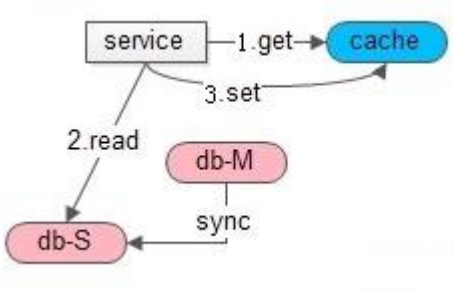
先操作缓存，还是先操作数据库？

（1）读请求，先读缓存，如果没有命中，读数据库，再 set 回缓存

（2）写请求

（2.1）先缓存，再数据库

（2.2）缓存，使用 delete，而不是 set



在缓存读取流程中，如果主从没有同步完成，步骤二读取到一个旧数据，可能导致缓存里 set 一个旧数据，最终导致数据库和缓存数据不一致。

文章来源

架构师之路

微信号 road5858

功能介绍 架构师之路，坚持撰写接地气的架构文章

缓存，是淘汰，还是修改？

允许 cache miss 的场景，不管是 memcache 还是 redis，当被缓存的内容变化时，是改修改缓存，还是淘汰缓存？这是今天将要讨论的话题。

KV 缓存都缓存了一些什么数据？

（1）朴素类型的数据，例如：int

（2）序列化后的对象，例如：User 实体，本质是 binary

（3）文本数据，例如：json 或者 html

淘汰缓存中的这些数据，修改缓存中的这些数据，有什么差别？

（1）淘汰某个 key，操作简单，直接将 key 置为无效，但下一次该 key 的访问会 cache miss

（2）修改某个 key 的内容，逻辑相对复杂，但下一次该 key 的访问仍会 cache hit

可以看到，差异仅仅在于一次 cache miss。

缓存中的 value 数据一般是怎么修改的？

（1）朴素类型的数据，直接 set 修改后的值即可

（2）序列化后的对象：一般需要先 get 数据，反序列化成对象，修改其中的成员，再序列化为 binary，再 set 数据

（3）json 或者 html 数据：一般也需要先 get 文本，parse 成 doom 树对象，修改相关元素，序列化为文本，再 set 数据

结论：对于对象类型，或者文本类型，修改缓存 value 的成本较高，一般选择直接淘汰缓存。

对于朴素类型的数据，究竟应该修改缓存，还是淘汰缓存？

仍然视情况而定。

案例 1：

假设 ,缓存里存了某一个用户 uid=123 的余额是 money=100 元 ,业务场景是，购买了一个商品 pid=456。

分析：如果修改缓存，可能需要：

（1）去 db 查询 pid 的价格是 50 元

（2）去 db 查询活动的折扣是 8 折（商品实际价格是 40 元）

（3）去 db 查询用户的优惠券是 10 元（用户实际要支付 30 元）

（4）从 cache 查询 get 用户的余额是 100 元

（5）计算出剩余余额是 100 - 30 = 70

（6）到 cache 设置 set 用户的余额是 70

为了避免一次 cache miss，需要额外增加若干次 db 与 cache 的交互，得不偿失。

结论：此时，应该淘汰缓存，而不是修改缓存。

案例 2：

假设缓存里存了某一个用户 uid=123 的余额是 money=100 元，业务场景是，需要扣减 30 元。

分析：如果修改缓存，需要：

（1） 从 cache 查询 get 用户的余额是 100 元

（2）计算出剩余余额是 100 - 30 = 70

（3）到 cache 设置 set 用户的余额是 70

为了避免一次 cache miss，需要额外增加若干次 cache 的交互，以及业务的计算，得不偿失。

结论：此时，应该淘汰缓存，而不是修改缓存。

案例 3：

假设 ,缓存里存了某一个用户 uid=123 的余额是 money=100 元 业务场景是，余额要变为 70 元。

分析：如果修改缓存，需要到 cache 设置 set 用户的余额是 70

修改缓存成本很低。

结论：此时，可以选择修改缓存。当然，如果选择淘汰缓存，只会额外增加一次 cache miss，成本也不高。

允许 cache miss 的 KV 缓存写场景：大部分情况，修改 value 成本会高于“增加一次 cache miss”，因此应该淘汰缓存。

session 一致性架构设计实践

一、缘起

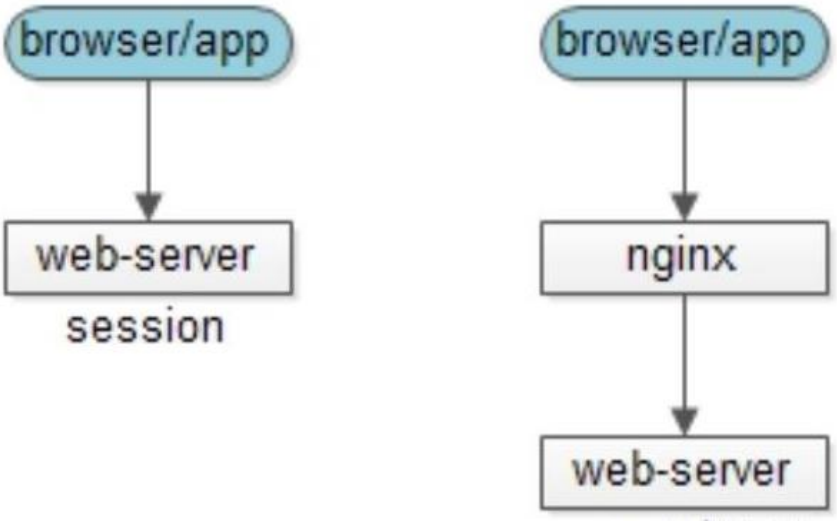
什么是 session？

服务器为每个用户创建一个会话，存储用户的相关信息，以便多次请求能够定位到同一个上下文。

Web 开发中，web-server 可以自动为同一个浏览器的访问用户自动创建 session，提供数据存储功能。最常见的，会把用户的登录信息、用户信息存储在 session 中，以保持登录状态。

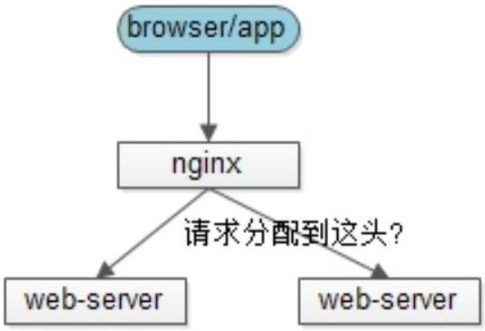
什么是 session 一致性问题？

只要用户不重启浏览器，每次 http 短连接请求，理论上服务端都能定位到 session，保持会话。



当只有一台 web-server 提供服务时，每次 http 短连接请求，都能够正确路由到存储 session 的对应 web-server（废话，因为只有一台）。

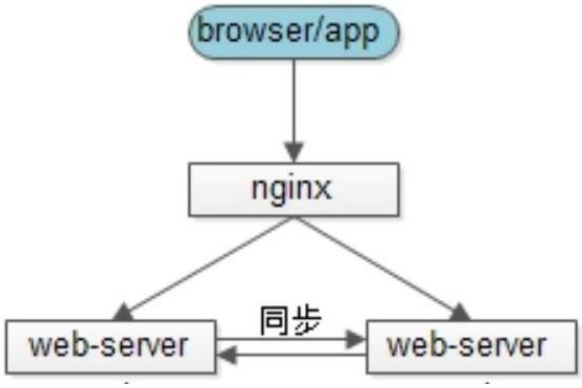
此时的 web-server 是无法保证高可用的，采用“冗余+故障转移”的多台 web-server 来保证高可用时，每次 http 短连接请求就不一定能路由到正确的 session 了。



如上图，假设用户包含登录信息的 session 都记录在第一台 web-server 上，反向代理如果将请求路由到另一台 web-server 上，可能就找不到相关信息，而导致用户需要重新登录。

在 web-server 高可用时，如何保证 session 路由的一致性，是今天将要讨论的问题。

二、session 同步法



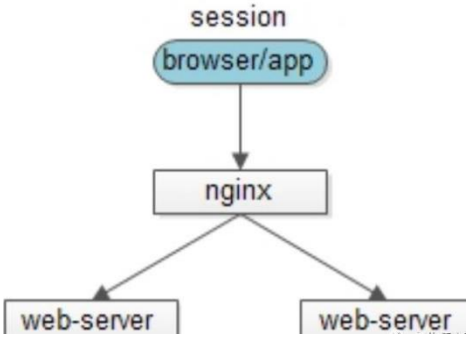
思路：多个 web-server 之间相互同步 session，这样每个 web-server 之间都包含全部的 session

优点：web-server 支持的功能，应用程序不需要修改代码。

不足：

- session 的同步需要数据传输，占内网带宽，有时延
- 所有 web-server 都包含所有 session 数据，数据量受内存限制，无法水平扩展
- 有更多 web-server 时要歇菜

三、客户端存储法



思路：服务端存储所有用户的 session，内存占用较大，可以将 session 存储到浏览器 cookie 中，每个端只要存储一个用户的数据了。

优点：服务端不需要存储。

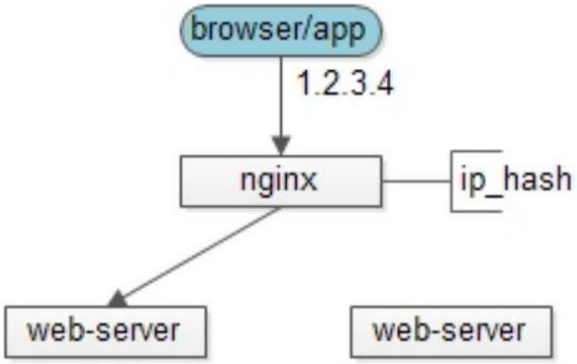
缺点：

- 每次 http 请求都携带 session，占外网带宽
- 数据存储在端上，并在网络传输，存在泄漏、篡改、窃取等安全隐患
- session 存储的数据大小受 cookie 限制

“端存储”的方案虽然不常用，但确实是一种思路。

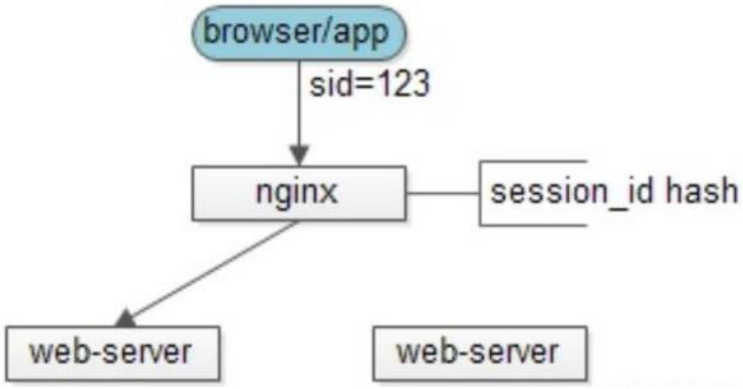
四、反向代理 hash 一致性

思路：web-server 为了保证高可用，有多台冗余，反向代理层能不能做一些事情，让同一个用户的请求保证落在一台 web-server 上呢？



方案一：四层代理 hash

反向代理层使用用户 ip 来做 hash，以保证同一个 ip 的请求落在同一个 web-server 上



方案二：七层代理 hash

反向代理使用 http 协议中的某些业务属性来做 hash ,例如 sid ,city_id ,user_id 等，能够更加灵活的实施 hash 策略，以保证同一个浏览器用户的请求落在同一个 web-server 上

优点：

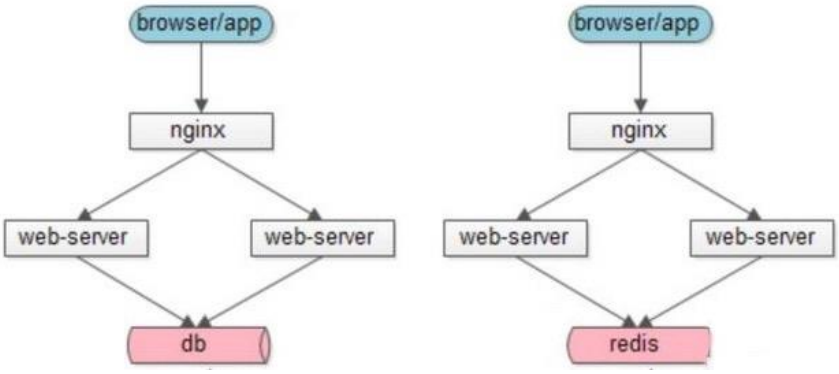
- 只需要改 nginx 配置，不需要修改应用代码
- 负载均衡，只要 hash 属性是均匀的，多台 web-server 的负载是均衡的
- 可以支持 web-server 水平扩展（session 同步法是不行的，受内存限制）

不足：

- 如果 web-server 重启，一部分 session 会丢失，产生业务影响，如部分用户重新登录
- 如果 web-server 水平扩展，rehash 后 session 重新分布，也会有一部分用户路由不到正确的 session

session 一般是有有效期的，所有不足中的两点，可以认为等同于部分 session 失效，一般问题不大。对于四层 hash 还是七层 hash，个人推荐前者：让专业的软件做专业的事情，反向代理就负责转发，尽量不要引入应用层业务属性，除非不得不这么做（例如，有时候多机房多活需要按照业务属性路由到不同机房的 web-server）。

五、后端统一存储



思路：将 session 存储在 web-server 后端的存储层，数据库或者缓存

优点：

- 没有安全隐患
- 可以水平扩展，数据库/缓存水平切分即可
- web-server 重启或者扩容都不会有 session 丢失

不足：增加了一次网络调用，并且需要修改应用代码

对于 db 存储还是 cache，个人推荐后者：session 读取的频率会很高，数据库压力会比较大。如果有 session 高可用需求，cache 可以做高可用，但大部分情况下 session 可以丢失，一般也不需要考虑高可用。

六、总结

保证 session 一致性的架构设计常见方法有：

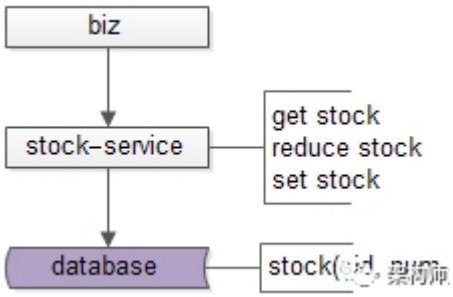
- session 同步法：多台 web-server 相互同步数据
- 客户端存储法：一个用户只存储自己的数据
- 反向代理 hash 一致性：四层 hash 和七层 hash 都可以做，保证一个用户的请求落在一台 web-server 上
- 后端统一存储：web-server 重启和扩容，session 也不会丢失

库存扣多了，到底怎么整

业务复杂、数据量大、并发量大的业务场景下，典型的互联网架构，一般会分为这么几层：

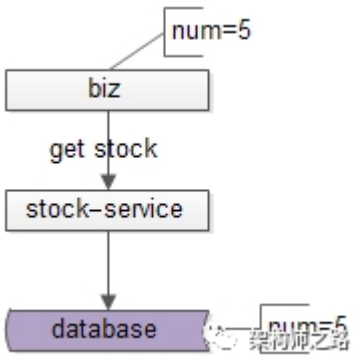
- 调用层，一般是处于端上的 browser 或者 APP
- 站点层，一般是拼装 html 或者 json 返回的 web-server 层
- 服务层，一般是提供 RPC 调用接口的 service 层
- 数据层，提供固化数据存储的 db

对于库存业务，一般有个库存服务，提供库存的查询、扣减、设置等 RPC 接口：



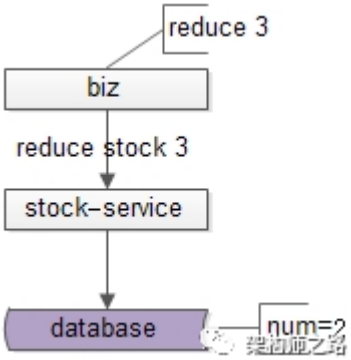
- 库存查询，stock-service 本质上执行的是
  select num from stock where sid=$sid
- 库存扣减，stock-service 本质上执行的是
  update stock set num=num-$reduce where sid=$sid
- 库存设置，stock-service 本质上执行的是
  update stock set num=$num_new where sid=$sid

用户下单前，一般会对库存进行查询，有足够的存量才允许扣减：



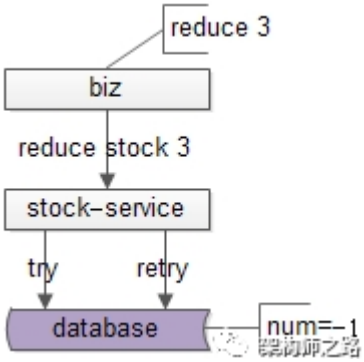
如上图所示，通过查询接口，得到库存是 5。

用户下单时，接着会对库存进行扣减：



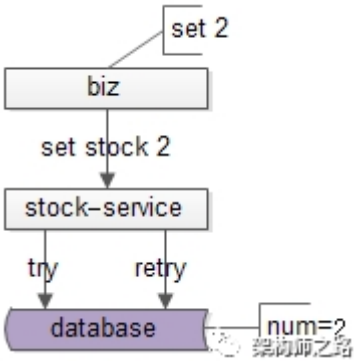
如上图所示，购买 3 单位的商品，通过扣减接口，最终得到库存是 2。

希望设计往往有容错机制，例如“重试”，如果通过扣减接口来修改库存，在重试时，可能会得到错误的数据，导致重复扣减：



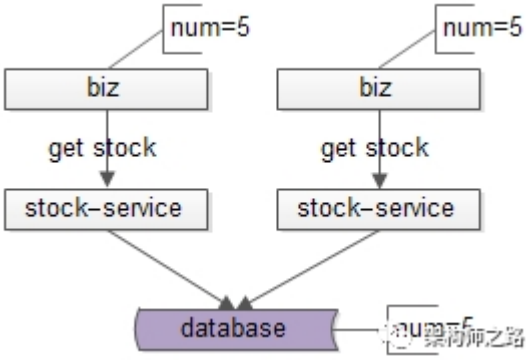
如上图所示，如果数据库层面有重试容错机制，可能导致一次扣减执行两次，最终得到一个负数的错误库存。

重试导致错误的根本原因，是因为“扣减”操作是一个非幂等的操作，不能够重复执行，改成设置操作则不会有这个问题：



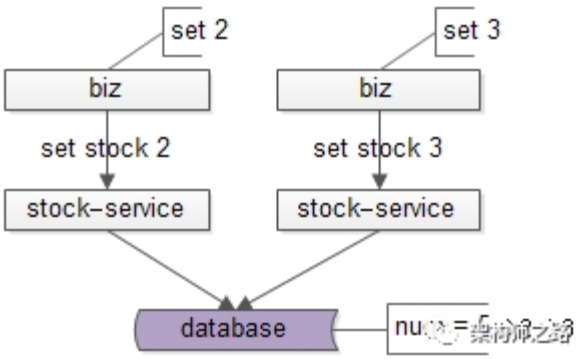
如上图所示，同样是购买 3 单位的商品，通过设置库存操作，即使有重试容错机制，也不会得到错误的库存，设置库存是一个幂等操作。

在并发量很大的情况下，还会有其他的问题：



如上图所示，两个并发的操作，查询库存，都得到了库存是 5。

接下来用户发生了并发的购买动作（秒杀类业务特别容易出现）：



如上图所示：

- 用户 1 购买了 3 个库存，于是库存要设置为 2
- 用户 2 购买了 2 个库存，于是库存要设置为 3
- 这两个设置库存的接口并发执行，库存会先变成 2，再变成 3，导致数据不一致（实　际卖出了 5 件商品，但库存只扣减了 2，最后一次设置库存会覆盖和掩盖前一次并发操作）

其根本原因是，设置操作发生的时候，没有检查库存与查询出来的库存有没有变化，理论上：

- 库存为 5 时，用户 1 的库存设置才能成功
- 库存为 5 时，用户 2 的库存设置才能成功

实际执行的时候：

- 库存为 5，用户 1 的 set stock 2 确实应该成功
- 库存变为 2 了，用户 2 的 set stock 3 应该失败掉

升级修改很容易，将库存设置接口，stock-service 上执行的：

update stock set num=$y where sid=$sid

升级为：

update stock set num=$num_new where sid=$sid and num=$num_old

这正是大家常说的“Compare And Set”（CAS），是一种常见的降低读写锁冲突，保证数据一致性的方法。

总之，在业务复杂，数据量大，并发量大的情况下，库存扣减容易引发数据的不一致，常见的优化方案有两个：

- 调用“设置库存”接口，能够保证数据的幂等性
- 在实现“设置库存”接口时，需要加上原有库存的比较，才允许设置成功，能解决高并发下库存扣减的一致性问题

架构师之路

微信号 road5858

功能介绍 架构师之路，坚持撰写接地气的架构文章

文章来源

细聊分布式 ID 生成方法

一、需求缘起

几乎所有的业务系统，都有生成一个记录标识的需求，例如：

（1）消息标识：message-id

（2）订单标识：order-id

（3）帖子标识：tiezi-id

这个记录标识往往就是数据库中的唯一主键，数据库上会建立聚集索引（cluster index），即在物理存储上以这个字段排序。

这个记录标识上的查询，往往又有分页或者排序的业务需求，例如：

（1）拉取最新的一页消息：selectmessage-id/ order by time/ limit 100

（2）拉取最新的一页订单：selectorder-id/ order by time/ limit 100

（3）拉取最新的一页帖子：selecttiezi-id/ order by time/ limit 100

所以往往要有一个 time 字段，并且在 time 字段上建立普通索引（non-cluster index）。

我们都知道普通索引存储的是实际记录的指针，其访问效率会比聚集索引慢，如果记录标识在生成时能够基本按照时间有序，则可以省去这个 time 字段的索引查询：

select message-id/ (order by message-id)/limit 100

再次强调，能这么做的前提是，message-id 的生成基本是趋势时间递增的。

这就引出了记录标识生成（也就是上文提到的三个 XXX-id）的两大核心需求：

（1）全局唯一

（2）趋势有序

这也是本文要讨论的核心问题：如何高效生成趋势有序的全局唯一 ID。

二、常见方法、不足与优化

【常见方法一：使用数据库的 auto_increment 来生成全局唯一递增 ID】

优点：

（1）简单，使用数据库已有的功能

（2）能够保证唯一性

（3）能够保证递增性

（4）步长固定

缺点：

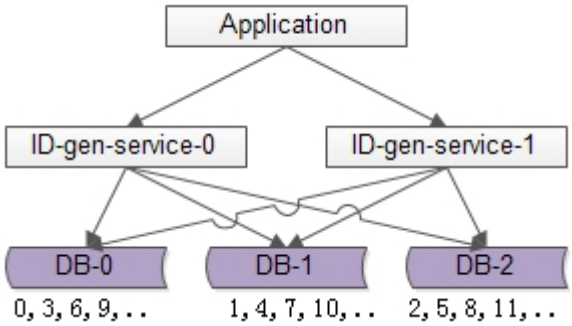
（1）可用性难以保证：数据库常见架构是一主多从+读写分离，生成自增 ID 是写请求，主库挂了就玩不转了

（2）扩展性差，性能有上限：因为写入是单点，数据库主库的写性能决定 ID 的生成性能上限，并且难以扩展

改进方法：

（1）增加主库，避免写入单点

（2）数据水平切分，保证各主库生成的 ID 不重复



如上图所述，由 1 个写库变成 3 个写库，每个写库设置不同的 auto_increment 初始值，以及相同的增长步长，以保证每个数据库生成的 ID 是不同的（上图中库 0 生成 0,3,6,9...，库 1 生成 1,4,7,10，库 2 生成 2,5,8,11...）

改进后的架构保证了可用性，但缺点是：

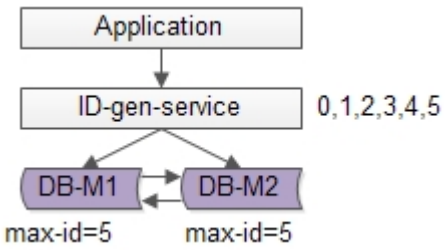
（1）丧失了 ID 生成的“绝对递增性”：先访问库 0 生成 0,3，再访问库 1 生成 1，可能导致在非常短的时间内，ID 生成不是绝对递增的（这个问题不大，我们的目标是趋势递增，不是绝对递增）

（2）数据库的写压力依然很大，每次生成 ID 都要访问数据库

为了解决上述两个问题，引出了第二个常见的方案

【常见方法二：单点批量 ID 生成服务】

分布式系统之所以难，很重要的原因之一是“没有一个全局时钟，难以保证绝对的时序”，要想保证绝对的时序，还是只能使用单点服务，用本地时钟保证“绝对时序”。数据库写压力大，是因为每次生成 ID 都访问了数据库，可以使用批量的方式降低数据库写压力。



如上图所述，数据库使用双 master 保证可用性，数据库中只存储当前 ID 的最大值，例如 0。ID 生成服务假设每次批量拉取 6 个 ID，服务访问数据库，将当前 ID 的最大值修改为 5，这样应用访问 ID 生成服务索要 ID，ID 生成服务不需要每次访问数据库，就能依次派发 0,1,2,3,4,5 这些 ID 了，当 ID 发完后，再将 ID 的最大值修改为 11，就能再次派发 6,7,8,9,10,11 这些 ID 了，于是数据库的压力就降低到原来的 1/6 了。

优点：

（1）保证了 ID 生成的绝对递增有序

（2）大大的降低了数据库的压力，ID 生成可以做到每秒生成几万几十万个

缺点：

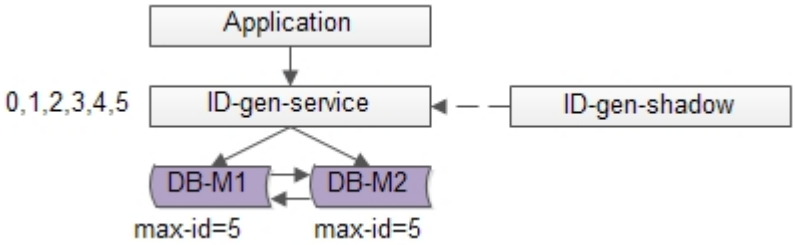
（1）服务仍然是单点

（2）如果服务挂了，服务重启起来之后，继续生成 ID 可能会不连续，中间出现空洞（服务内存是保存着 0,1,2,3,4,5，数据库中 max-id 是 5，分配到 3 时，服务重启了，下次会从 6 开始分配，4 和 5 就成了空洞，不过这个问题也不大）

（3）虽然每秒可以生成几万几十万个 ID，但毕竟还是有性能上限，无法进行水平扩展

改进方法：

单点服务的常用高可用优化方案是“备用服务”，也叫“影子服务”，所以我们能用以下方法优化上述缺点（1）：



如上图，对外提供的服务是主服务，有一个影子服务时刻处于备用状态，当主服务挂了的时候影子服务顶上。这个切换的过程对调用方是透明的，可以自动完成，常用的技术是 vip+keepalived，具体就不在这里展开。

【常见方法三：uuid】

上述方案来生成 ID，虽然性能大增，但由于是单点系统，总还是存在性能上限的。同时，上述两种方案，不管是数据库还是服务来生成 ID，业务方 Application 都需要进行一次远程调用，比较耗时。有没有一种本地生成 ID 的方法，即高性能，又时延低呢？

uuid 是一种常见的方案：string ID =GenUUID();

优点：

（1）本地生成 ID，不需要进行远程调用，时延低

（2）扩展性好，基本可以认为没有性能上限

缺点：

（1）无法保证趋势递增

（2）uuid 过长，往往用字符串表示，作为主键建立索引查询效率低，常见优化方案为“转化为两个 uint64 整数存储”或者“折半存储”（折半后不能保证唯一性）

【常见方法四：取当前毫秒数】

uuid 是一个本地算法，生成性能高，但无法保证趋势递增，且作为字符串 ID 检索效率低，有没有一种能保证递增的本地算法呢？

取当前毫秒数是一种常见方案：uint64 ID = GenTimeMS();

优点：

（1）本地生成 ID，不需要进行远程调用，时延低

（2）生成的 ID 趋势递增

（3）生成的 ID 是整数，建立索引后查询效率高

缺点：

（1）如果并发量超过 1000，会生成重复的 ID

我去，这个缺点要了命了，不能保证 ID 的唯一性。当然，使用微秒可以降低冲突概率，但每秒最多只能生成 1000000 个 ID，再多的话就一定会冲突了，所以使用微秒并不从根本上解决问题。

【常见方法五：类 snowflake 算法】

snowflake 是 twitter 开源的分布式 ID 生成算法，其核心思想是：一个 long 型的 ID，使用其中 41bit 作为毫秒数，10bit 作为机器编号，12bit 作为毫秒内序列号。这个算法单机每秒内理论上最多可以生成 1000*(2^12)，也就是 400W 的 ID，完全能满足业务的需求。

借鉴 snowflake 的思想，结合各公司的业务逻辑和并发量，可以实现自己的分布式 ID 生成算法。

举例，假设某公司 ID 生成器服务的需求如下：

（1）单机高峰并发量小于 1W，预计未来 5 年单机高峰并发量小于 10W

（2）有 2 个机房，预计未来 5 年机房数量小于 4 个

（3）每个机房机器数小于 100 台

（4）目前有 5 个业务线有 ID 生成需求，预计未来业务线数量小于 10 个

（5）...

分析过程如下：

（1）高位取从 2016 年 1 月 1 日到现在的毫秒数（假设系统 ID 生成器服务在这个时间之后上线），假设系统至少运行 10 年，那至少需要 10 年*365 天*24 小时*3600 秒*1000 毫秒=320*10^9，差不多预留 39bit 给毫秒数

（2）每秒的单机高峰并发量小于 10W，即平均每毫秒的单机高峰并发量小于 100，差不多预留 7bit 给每毫秒内序列号

（3）5 年内机房数小于 4 个，预留 2bit 给机房标识

（4）每个机房小于 100 台机器，预留 7bit 给每个机房内的服务器标识

（5）业务线小于 10 个，预留 4bit 给业务线标识

这样设计的 64bit 标识，可以保证：

（1）每个业务线、每个机房、每个机器生成的 ID 都是不同的

（2）同一个机器，每个毫秒内生成的 ID 都是不同的

（3）同一个机器，同一个毫秒内，以序列号区区分保证生成的 ID 是不同的

（4）将毫秒数放在最高位，保证生成的 ID 是趋势递增的

缺点：

（1）由于“没有一个全局时钟”，每台服务器分配的 ID 是绝对递增的，但从全局看，生成的 ID 只是趋势递增的（有些服务器的时间早，有些服务器的时间晚）

最后一个容易忽略的问题：

生成的 ID，例如 message-id/ order-id/ tiezi-id，在数据量大时往往需要分库分表，这些 ID 经常作为取模分库分表的依据，为了分库分表后数据均匀，ID 生成往往有“取模随机性”的需求，所以我们通常把每秒内的序列号放在 ID 的最末位，保证生成的 ID 是随机的。

又如果，我们在跨毫秒时，序列号总是归 0，会使得序列号为 0 的 ID 比较多，导致生成的 ID 取模后不均匀。解决方法是，序列号不是每次都归 0，而是归一个 0 到 9 的随机数，这个地方。

文章来源

架构师之路
微信号 road5858
功能介绍 架构师之路，坚持撰写接地气的架构文章

线程数究竟设多少合理

一、需求缘起

Web-Server 通常有个配置，最大工作线程数，后端服务一般也有个配置，工作线程池的线程数量，这个线程数的配置不同的业务架构师有不同的经验值，有些业务设置为 CPU 核数的 2 倍，有些业务设置为 CPU 核数的 8 倍，有些业务设置为 CPU 核数的 32 倍。“工作线程数”的设置依据是什么，到底设置为多少能够最大化 CPU 性能，是本文要讨论的问题。

二、一些共性认知

在进行进一步深入讨论之前，先以提问的方式就一些共性认知达成一致。

工作线程数是不是设置的越大越好？——肯定不是的

1）一来服务器 CPU 核数有限，同时并发的线程数是有限的，1 核 CPU 设置 10000 个工作线程没有意义

2）线程切换是有开销的，如果线程切换过于频繁，反而会使性能降低

调用 sleep()函数的时候，线程是否一直占用 CPU？

不占用，等待时会把 CPU 让出来，给其他需要 CPU 资源的线程使用。不止调用 sleep()函数，在进行一些阻塞调用，例如网络编程中的阻塞 accept()[等待客户端连接]和阻塞 recv()[等待下游回包]也不占用 CPU 资源

如果 CPU 是单核，设置多线程有意义么，能提高并发性能么？

即使是单核，使用多线程也是有意义的

1）多线程编码可以让我们的服务/代码更加清晰，有些 IO 线程收发包，有些 Worker 线程进行任务处理，有些 Timeout 线程进行超时检测

2）如果有一个任务一直占用 CPU 资源在进行计算，那么此时增加线程并不能增加并发，例如这样的一个代码

```
while(1){ i++; }
```

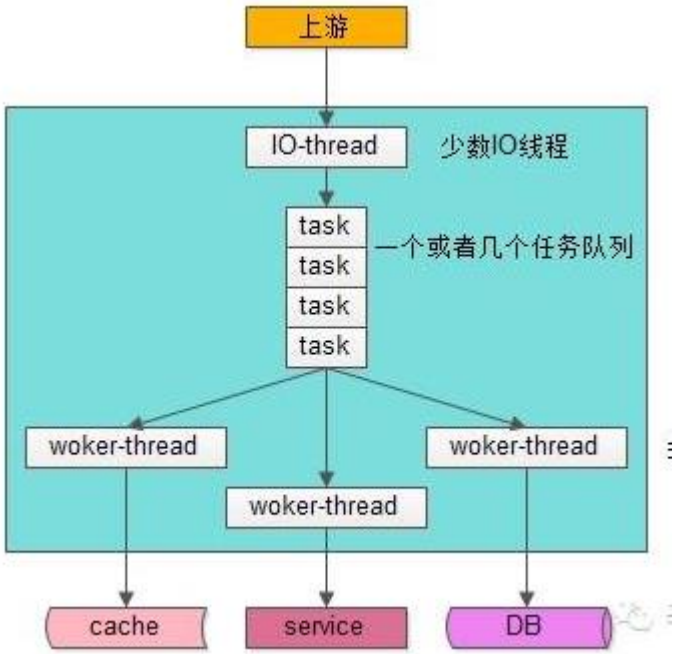
该代码一直不停的占用 CPU 资源进行计算，会使 CPU 占用率达到 100%

3）通常来说，Worker 线程一般不会一直占用 CPU 进行计算，此时即使 CPU 是单核，增加 Worker 线程也能够提高并发，因为这个线程在休息的时候，其他的线程可以继续工作

三、常见服务线程模型

了解常见的服务线程模型，有助于理解服务并发的原理，一般来说互联网常见的服务线程模型有如下两种

IO 线程与工作线程通过队列解耦类模型



如上图，大部分 Web-Server 与服务框架都是使用这样的一种“IO 线程与 Worker 线程通过队列解耦”类线程模型：

1）**有少数几个 IO 线程**监听上游发过来的请求，并进行收发包（生产者）

2）**有一个或者多个任务队列**，作为 IO 线程与 Worker 线程异步解耦的数据传输通道（临界资源）

3）**有多个工作线程**执行正真的任务（消费者）

这个线程模型应用很广，符合大部分场景，这个线程模型的特点是，工作线程内部是同步阻塞执行任务的（回想一下 tomcat 线程中是怎么执行 Java 程序的，dubbo 工作线程中是怎么执行任务的），因此可以通过增加 Worker 线程数来增加并发能力，今天要讨论的重点是“该模型 Worker 线程数设置为多少能达到最大的并发”。

纯异步线程模型

任何地方都没有阻塞，这种线程模型只需要设置很少的线程数就能够做到很高的吞吐量，Lighttpd 有一种单进程单线程模式，并发处理能力很强，就是使用的的这种模型。该模型的缺点是：

1）如果使用单线程模式，难以利用多 CPU 多核的优势

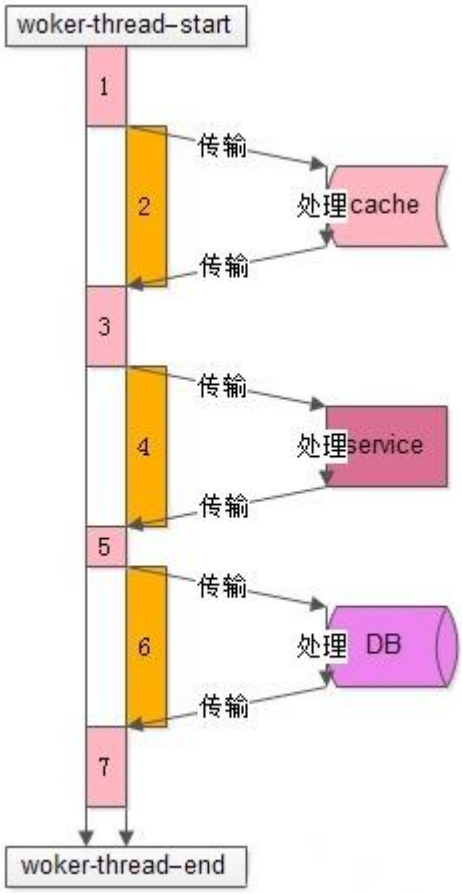
2）程序员更习惯写同步代码，callback 的方式对代码的可读性有冲击，对程序员的要求也更高

3）框架更复杂，往往需要 server 端收发组件，server 端队列，client 端收发组件，client 端队列，上下文管理组件，有限状态机组件，超时管理组件的支持

however，这个模型不是今天讨论的重点。

四、工作线程的工作模式

了解工作线程的工作模式，对量化分析线程数的设置非常有帮助：



上图是一个典型的工作线程的处理过程，从开始处理 start 到结束处理 end，该任务的处理共有 7 个步骤：

1、从工作队列里拿出任务，进行一些本地初始化计算，例如 http 协议分析、参数解析、参数校验等

2、访问 cache 拿一些数据

3、拿到 cache 里的数据后，再进行一些本地计算，这些计算和业务逻辑相关

4、通过 RPC 调用下游 service 再拿一些数据，或者让下游 service 去处理一些相关的任务

5、RPC 调用结束后，再进行一些本地计算，怎么计算和业务逻辑相关

6、访问 DB 进行一些数据操作

7、操作完数据库之后做一些收尾工作，同样这些收尾工作也是本地计算，和业务逻辑相关

分析整个处理的时间轴，会发现以下几点：

1）其中 1，3，5，7 步骤中【上图中粉色时间轴】，线程进行本地业务逻辑计算时需要占用 CPU

2）而 2，4，6 步骤中【上图中橙色时间轴】，访问 cache、service、DB 过程中线程处于一个等待结果的状态，不需要占用 CPU，进一步的分解，这个“等待结果”的时间共分为三部分：

2.1）请求在网络上传输到下游的 cache、service、DB

2.2）下游 cache、service、DB 进行任务处理

2.3）cache、service、DB 将报文在网络上传回工作线程

五、量化分析并合理设置工作线程数

最后一起来回答工作线程数设置为多少合理的问题。通过上面的分析，Worker 线程在执行的过程中，有一部计算时间需要占用 CPU，另一部分等待时间不需要占用 CPU，通过量化分析，例如打日志进行统计，可以统计出整个 Worker 线程执行过程中这两部分时间的比例，例如：

1）时间轴 1，3，5，7【上图中粉色时间轴】的计算执行时间是 100ms

2）时间轴 2，4，6【上图中橙色时间轴】的等待时间也是 100ms

得到的结果是，这个线程计算和等待的时间是 1：1，即有 50%的时间在计算（占用 CPU），50%的时间在等待（不占用 CPU）：

1）假设此时是单核，则设置为 2 个工作线程就可以把 CPU 充分利用起来，让 CPU 跑到 100%

2）假设此时是 N 核，则设置为 2N 个工作现场就可以把 CPU 充分利用起来，让 CPU 跑到 N*100%

结论：N 核服务器，通过执行业务的单线程分析出本地计算时间为 x，等待时间为 y，则工作线程数（线程池线程数）设置为 N*(x+y)/x，能让 CPU 的利用率最大化。

经验：一般来说，非 CPU 密集型的业务（加解密、压缩解压缩、搜索排序等业务是 CPU 密集型的业务），瓶颈都在后端数据库，本地 CPU 计算的时间很少，所以设置几十或者几百个工作线程也都是可能的。

六、结论

N 核服务器，通过执行业务的单线程分析出本地计算时间为 x，等待时间为 y，则工作线程数（线程池线程数）设置为 N*(x+y)/x，能让 CPU 的利用率最大化。

文章来源

架构师之路
微信号 road5858
功能介绍 架构师之路，坚持撰写接地气的架构文章

烂代码传奇

现在你们管叫我烂代码， 实在是委屈我了，想当年我年轻的时候，那可真是人见人爱，花见花开，气质高贵，身段优雅，无数程序员对我着迷。 那个时候你们叫我什么来着？ 好像是优雅代码、漂亮代码吧。但是请注意，虽然被你们称为烂代码，我可是一直在生产环境上运行的代码啊，支持起成千上万的并发访问和计算，所以准确点儿说，我叫遗留代码。我为什么会变成这样， 你们程序员负有不可推卸的责任，如果你看看我的版本管理历史，对，就是 2010 年以前， 还能看到年轻而漂亮的我。但也是从那个时候开始， 多变的需求、进度的压力、人员的素质、混乱的管理， 让我慢慢走向了这条不归路。

我清楚的记得那一天，一个叫小董的程序员奉命开发一个新功能， 要对我这一块代码进行改动， 当时有两个方案，一个是对我重构，继续保持优雅的身段，但是工作量比较大。

另外一个是就地修改，只需要寥寥几行代码即可，代价是需要加上我极为讨厌的类似 if(p instanceof XXX) {……}这样的代码， 在进度的压力下， 小董为了少加班， 没有顶住魔鬼的诱惑，不假思索地选择了后一种方案。我本来想抗议一下， 但是看着他睡眼惺忪的眼睛， 想到他昨晚加班到凌晨 3 点才睡，心就软了。体谅一下码农 ,忍了吧，不就多了一个看起来非常丑陋的 if 分支吗？小董的代码被传到了组长那里做 Code Review, 曾经要求非常严格的组长这时候忙得四脚朝天，根本没时间细看，这段丑陋的代码作为漏网之鱼，轻松进入了版本管理系统的代码仓库。心太软的后果很严重， 这个丑陋的 if 分支如同我漂

亮面孔上的一道伤疤， 开了一个恶劣的先例，时刻在诱惑着程序员们： 大家都不用有心理负担了， 以后想怎么改就怎么改，反正已经开始腐化掉了！三天以后， 张大胖毫不留情的在这里补了一刀，又是一个丑陋的特殊判断！ 不过张大胖还算良心，写了“长达”10 个字的注释，努力地辩解代码的意图，实际上我内心非常清楚 ,这 10 个字完全词不达意 ,把人带沟里的可能是 99%。一周以后，项目进度依然落后， 经理招了一大批新人入职， 终极“杀人王”小李出现了！小李是个典型地使用面向对象语言写面向过程代码的年轻人，刚进团队的时候，他还纳闷说这个项目的代码怎么和自己见到的、自己写的不一样呢？ 这里的代码读起来怎么这么难理解 ？其实他不知道，面向对象的代码由于是针对接口编程， 每当你去找接口实现的时候是比较费劲的， 比起直接的面向过程代码，读起来是个不小的阻碍。然后小李便发现了被改得面目全非的我，熟悉的配方，熟悉的味道， 他欣喜若狂， 举着一把杀猪刀， 在我这里大砍大杀， 我疼得昏倒在地，在被编译器唤醒以后，看到这数不清的伤疤，我吐血三升，再次晕倒。拖着伤痕累累的身体，我和伙伴们又进入编译器， 让它把我们编译成二进制代码，这些二进制代码长得都差不多，也无法看出曾经的丑陋伤疤， 他们乐呵呵的奔向生产系统。编译器看到我们的疑惑，解释说： “你们这些源代码主要是给人看的，所以很在乎自己的外表， 但是二进制代码是让机器执行的，都是指令而已，根本不关心长得如何， 能执行就行。”我心里一阵羡慕：二进制代码的生活真好，单纯而快乐。一年以后，这个系统的源代码已经变得凌乱不堪，极难维护了。 有一回高层领导震怒了： 为什么新加一个小小的功能，你们这里

都得改一个月？客户可等不及啊，加班！可是加班也不行， 代码太难懂了，这简直就是一片黑暗丛林， 程序员通常是有去无回。领导又说了，你们的瀑布开发流程有问题， 现在敏捷软件开发很有效，赶紧转型敏捷。于是敏捷教练来了，看到我们这些烂代码，不由的露出了得意的笑容： “看看， 和我之前估计的一模一样吧，这么烂的代码你们还想迅速响应市场变化？ 一定得重构啊。可是没有人敢重构我们这些遗留代码，张大胖说：“不是不想重构，是不敢啊， 这代码牵一发而动全身， 万一该改错了怎么办？”教练说：“想重构必须得有测试，你们得先写测试用例！”那就写吧， 在教练的指导下，开始轰轰烈烈的单元测试运动，可是刚推进没多久就被迫停止了，因为我们这些遗留代码很难写测试，我们经常依赖数据库，网络，文件系统的数据，按照单元测试的做法，需要把他们用 Mock 技术给模拟出来， 但是想 Mock 的话代码中也得有“接缝”才行。（码农翻身注： 接缝的概念来自于《修改代码的艺术》， 指的是是指程序中的一些特殊的点，在这些点上你无需做任何修改就可以达到改动程序行为的目的，Mock 代码通常可以在这里安插）但是我们这些“烂代码”中哪里有“接缝”？为了提供接缝，就必须在没有测试保护的情况下对遗留代码进行修改， 这就回到了原始的问题，张大胖他们不敢改啊。敏捷教练说张大胖你们得有勇气做啊，张大胖说你真是站着说话不腰疼，万一改错了你又不承担责任都是我们的锅我们必须来背你拍拍屁股就走了.....教练无语，真的走了。 有人提议重新写， 但是也被否决， 原因是业务这么复杂，重写怎么能考虑到这么多细节， 重写出来的代码 Bug 会更多！于是只好保持现状，三年过去了， 开发人员都换了一茬，能够透彻理解系统的人几乎没有了。

经历了这么多事，我也就破罐子破摔了，完全不再在意自己的相貌。有些时候，程序员为了理解代码逻辑， 翻起了代码的早期版本，惊叹于当时的优雅代码，我也就有机会跟着自我陶醉一下，仅此而已了。

文章来源：

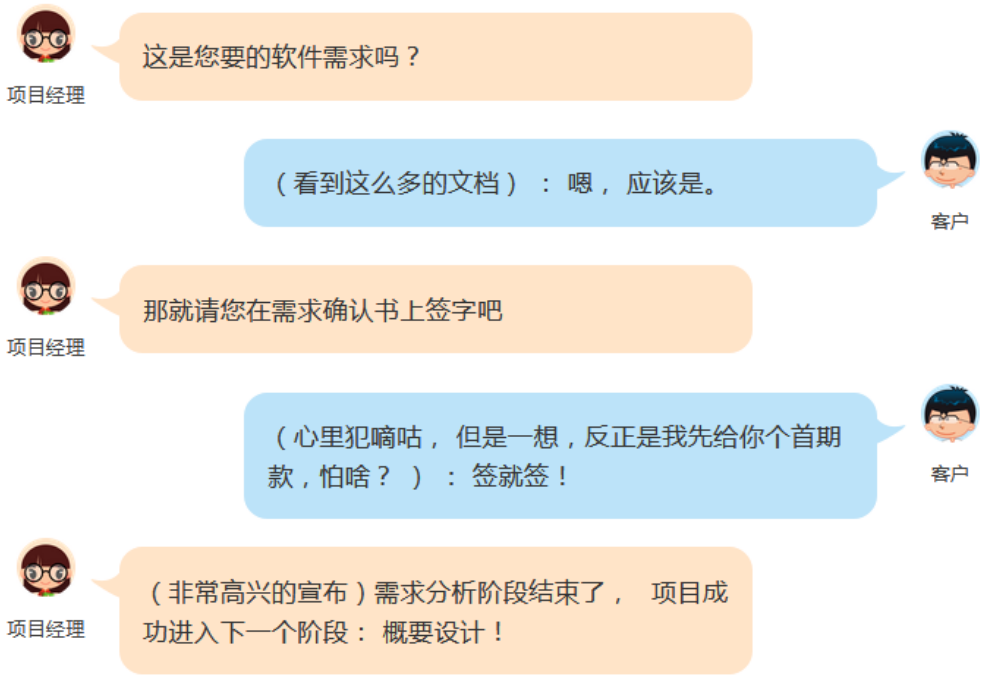
码农翻身
微信号 coderising
功能介绍 工作15年的前IBM架构师分享好玩有趣的编程知识和职场的经验教训， 不容错过。

白话敏捷软件开发

敏捷的意思就是反应迅速，为什么要反应迅速？看看那么多 996 公司就知道了，市场变化越来越快，客户要求越来越高，为了满足用户的需求，人家一个星期发一个版本，我们仨月才能憋出一个来，那还不被打的满地找牙？问题是如何才能反应迅速？先来看一个场景：

一、残酷的现实

软件开发有一大难题就是客户脑子中的需求难于描述出来， 我们通常的应对方法是这样：先花上几个月整理需求， 天天和客户座谈， 画出几百页的流程图， 写出上千页的文档， 最后把客户都快搞晕了。



然后是详细设计，开发，测试，我们强悍的技术团队开始发动，一切都严格按照计划进行，一切看起来都很完美，看来项目马上成功结束了！但是客户的验收测试给了我们当头一棒：这个界面怎么少了一个选项？那个界面怎么不能跳转，那个功能需要给领导一个后门，我的业务规则怎么不能改？每个人都很郁闷，几个月的辛苦开发看来要付诸东流了。

从这个场景中能看出的是，我们从客户那里得到的需求描述和需求文档，其实离客户真正想要的软件还差的很远。在瀑布式的开发模式下，验收测试发现的问题，要想改正代价是非常高昂的。

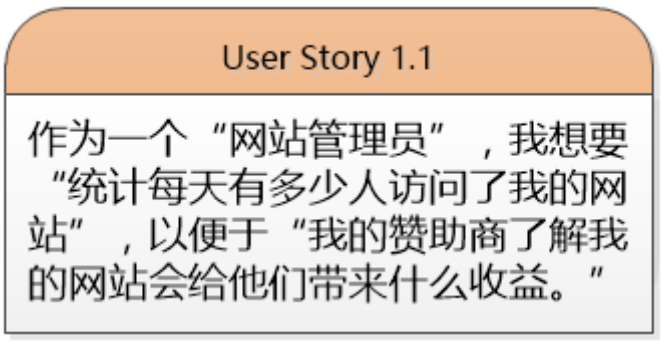
二、改进

一个想法自然而然就浮现出来： 为了避免到最后习惯性崩盘，能不能让客户经常性的做验收测试？让他们经常性的去使用一个可以工作的软件，从而告诉我们那些地方还有欠缺？ 那些地方做错了？这样我们可以迅速的修改，这样我们就会轻松多了。

我们可以把软件开发划分成一个个小的开发周期，例如每个周期就两三周时间，在这两周之内我们完成一个或几个功能，然后就让用户去试用，有问题立刻反馈，在下一个开发周期马上改掉。这样就可以逐步逼近客户的最终目标。这还带来了一个额外的好处，不用花费巨长的时间来分析，整理冗长的需求文档了。听起来很美是不是？但仔细想想这里边的问题很多。

1.抛弃了冗长的需求文档，但还是得描述需求啊

需要发明一个简单的、主要用来促进客户和开发团队沟通的描述形式，这个新的形式叫做用户故事，这里有个用户故事的例子：



这是一个卡片，背面还会记录下针对需求的讨论和验收标准。用户故事主要彰显的是：谁做了什么事，带来什么商业价值。

2、怎么决定每个小开发周期（我们称之为迭代吧）要开发的东西？

用户故事得有估算，得有大小，太大了一个迭代开发不完，还得拆分一下。我们需要对需求按照优先级进行排序，按照优先级从高到低的原则来开发。

3. 不要架构设计了吗？

一上来就按优先级选择需求，直接进入迭代开发，把架构师撂在一边，合适吗？架构工作肯定还是需要的，在正式的迭代周期开始之前需要架构设计，但是和设计出面面俱到的架构设计不同，我们更需要演进式的架构，随着迭代的推进而进化。

4. 那详细设计怎么办？

在每个迭代开始的时候，团队在一起把这些用户故事给拆分成一个个小的任务，这个拆分的过程就相当于详细设计了。对于一些特别复杂的，例如算法，当然可以写文档，帮助大家理解。

5. 由于是迭代式开发，这个迭代周期修改上一个迭代周期的代码在所难免，怎么保证不破坏原有的功能？总不能每次都手工重测一遍吧？这个绝对是一大难点，答案就是自动化的回归测试，包括单元测试和功能测试。开发人员写代码的同时，也要写下自动化的单元测试，　测试人员需要开发自动化的功能测试，这样一旦有了代码的修改，就可以运行它们，检查现有功能有没有被破坏。像持续集成这样的基础设施是必不可少的，每天，每小时，甚至每次代码提交都会触发编译，打包、部署、测试这样的过程。

6. 这么短的开发周期，测试人员怎么测试啊？

开发和测试需要同步进行，当开发在澄清需求的时候，测试需要参与，当开发在编码的时候， 测试人员在编写测试用例，等到一个用户故事开发完，马上就可以投入测试。

7. 看来开发、测试之间需要紧密的协作，它们之间怎么沟通？

肯定是面对面的沟通，有问题就跑到对方的座位那里去问，大家的座位最好在一起，扭头就可以讨论，尽可能减少效率不高的电话、QQ/微信等工具的使用。开发团队每天都开一个 15 分钟左右的站会，展示自己的进展和计划，让进度保持透明，及时暴露问题，解决问题。

8. 客户什么时候可以做验收测试？

随时欢迎，但是我们更倾向于迭代结束以后，这时候功能会稳定下来，我们会给客户做一个演示，告诉他这个迭代完成的工作，邀请他也测试一下软件，给我们反馈。当然客户可能会发现问题，甚至提出新的需求，我们表示欢迎，我们要和客户合作，而不是对抗。除了给客户演示之外，我们自己还会反思一下，看看有那些地方做的好，要继续保持，那些地方做的不好要持续改进。估计你也明白了，这种看起来很美的迭代化开发方法实施起来挺不容易的，如果我们给它起个名字的话，可以叫做：敏捷软件开发。

文章来源：

码农翻身
微信号 coderising
功能介绍 工作15年的前IBM架构师分享好玩有趣的编程知识和职场的经验教训， 不容错过。

ZHINENGKEJI

智能生活

科技引领未来

2018

主办单位：规划管理部
承办单位：服务产品部
地址：虹翔三路36号东航之家B1楼7层

顾问：李福娟、陆体山、张唯、张倩
主编：刘静莉　责编：张智　美编：朱媛
编辑：张希运、蒋纬、徐斌、屈鹏飞
本期作者：李建波等
联系电话：22335716
E-mail：zhangzhi@ceair.com