如何写一个后端的技术方案?

为什么要写方案？

写方案的目的是为了帮助我们想清楚需求，更好的落地需求。

并且记录实现的初衷，**后续方便进行迭代。** 我们经常会说之前的人没留下文档，导致很多时候都是通过代码来猜测功能的意图，导致最后很多代码我们不敢删，一直堆积在那里。

后续接手的人同样会产生疑问，“为什么之前迭代不删掉这部分代码？”

代码是这样一步一步堆积出来的，我们在这个过程中就要不断地往熵减的方向去努力，降低系统的整体复杂度。

再通过方案跨组评审的方式，可以帮助我们思考**通用性和复用点**。

比如有些问题可能在其他组已经遇到了，我们就可以直接借鉴他们成熟的代码。

如果这个过程中能够看到可以抽离服务的内容，那独立成服务进行维护后，健壮性也会上升一个等级，后续需要相似功能的人就不再需要过分担心这部分代码的稳定性。

应用设计的难点会因为系统复杂度的不同而存在巨大的差异。

如果是一个全新的系统，没有历史包袱，这个时候可以根据我们实际情况进行设计。

实现 CURD 的简单功能并不复杂，但是在 QPS 上实现高可用是需要前期进行功能容错设计，**并且要根据实际的项目限制条件，找出最合适的方案而不是技术最优的方案。**

如果是在原有的系统进行功能的增加，并且原有的系统复杂度和耦合度都相对较高，这个时候我们就需要非常小心了。

错综复杂的API依赖，牵一发而动全身，所以需要将变更内容先梳理后看涉及方是否需要更改。

因为是已经在运行的系统，所以上线的方案也会相对来说比较复杂，新增功能的同时也不能影响原有用户的使用。

这个时候就要从多个方面去考虑兼容性，如异步任务是否能正常处理、数据迁移的兼容性和API接口的兼容性。

也要在这个过程中考虑缓存是否会产生影响，如果本机缓存了数据，部署的时候是否会因为没有初始化缓存而导致应用瞬间无法扛住流量。

先通过一张思维导图来快速预览一下整体的内容：

接下具体看看每个点在写方案的过程中，会重点写哪些内容，来帮助我们完成一个具体的项目需求

一.需求背景

1.1 业务背景

需求背景一定是在最开头，要阐释清楚做这个的产品**需求背景、业务痛点**。

例如项目起因、业务方预期。**产品是为了解决什么样的问题才需要引入这个需求的**，如果后续某个产品功能下线后，对应的痛点也不存在了，则对应的需求也无需保留。

不然最后是否要保留需求都不知道，只会让代码越积重难返。

1.2 技术目标

还要了解这个需求所容纳的数据量的大小，如果是否有时效性的限制。

比如一个活动需求，它的时效性仅服务于每年的活动，并且每一年的活动形式可能不相同，所以是一个临时性的需求，但是由于是活动，所以 QPS 有可能会比常规功能高。

这就引出了对应需求的限制，在后续技术方案的设计中会十分关键

* 数据具有时效性和完整性
* 峰值QPS可能会高比常规流量高，这个可以根据产品的用户量给出实际的估算值，比如会有多少人参与活动

二.系统架构设计

2.1 业务架构

通过时序图划分清楚边界以及调用，新增或改动现有接口调用时，重点分析调用场景和链路，可以帮助我们发现性能隐患.

如果是新功能设计，能够帮助我们理清系统边界，明确后续用例应该在哪个系统内进行实现。

2.2 核心状态机

在业务流程比较复杂时，需要引入状态机来标识实体的状态，则需要明确画出状态机转换是是什么动作触发的，对应的动作则是我们是需要实现的用例。

比如支付订单创建后，用户支付后会流转成已支付等待发货，或者时超时支付则订单关闭解锁库存等，都是需要在状态机中体现出来，这也是为了后续实现的时候管理好整体流程。

三.存储方案设计

3.1 数据库设计

主要包含以下三个内容：

1. **核心领域模型的ER图**，这一部分帮我们梳理清楚满足需求所需要的实体内容。
2. 对应数据库表具体的**字段含义以及索引的设计：** 索引的设计对于查询来说十分重要，一个好的索引，可能是亿级别的数据存储下，能够不出现慢查询，甚至不需要进行分库分表，单表就能应付巨大的业务量。
3. **读写分离、分库分表和冷热分离等的设计：** 这部分如果在业务量大的时候需要重点分析，难点主要在于哪些是频繁读写的热数据，通过数据分级的方式，不同的数据存储在不同的介质中，这样不仅提高了程序的性能，也降低了应用整体部署的成本。

3.2 缓存设计

缓存资源占用的预估，如果占用资源过多，那么需要对缓存淘汰策略进行设计。

哪些数据需要进行缓存？为什么这些数据需要缓存？

缓存数据多久算多？缓存的更新策略是怎么样的？

存储需要的内存和成本，怎么进行估算的？

3.3 消息队列

削峰填谷是消息队列最主要的作用，就是将峰值流量平均分配到应用非热门时段去进行处理。

这里的设计最需要区分的点是数据处理的**实时性要求**，可以将实时性分成以下三类

1.实时

比如用户付钱款或者操作订单状态变更，这种都是需要实时去确认结果的，所以都是在API 操作完成返回的时候就已经持久化好了数据结果。

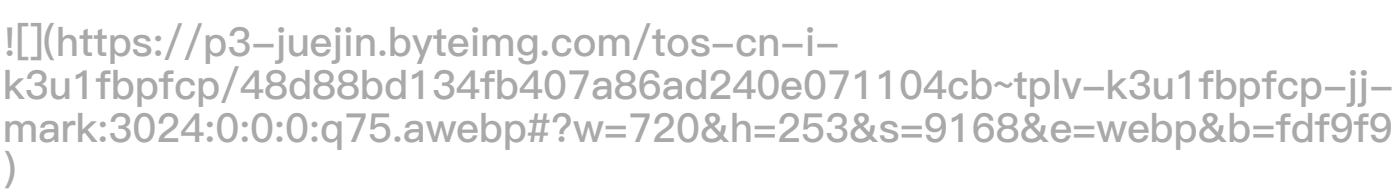
2.近实时

这种操作一般是用户不需要立即看到结果或者无需在页面展示的，比如确定收货后，货款是否立即打到对方的账户上，这个操作牵涉到多个系统，同步调用耗时无法接受，但是也不允许过长时间未处理，所以可以放在消息队列中去进行处理

但是这里处理也要区分，有些队列任务并不需要非常高的时效，那么对应的队列消费者也不需要太多，可以通过队列消费者个数来控制下游消费速度，并且通过队列消息堆积来看是否在可控范围内。

3.离线

这种一般属于数据统计和报表类需求，基本是天级别的数据延迟，



3.4 对象存储

> 对象存储是一种以非结构化格式（称为对象）存储和管理数据的技术

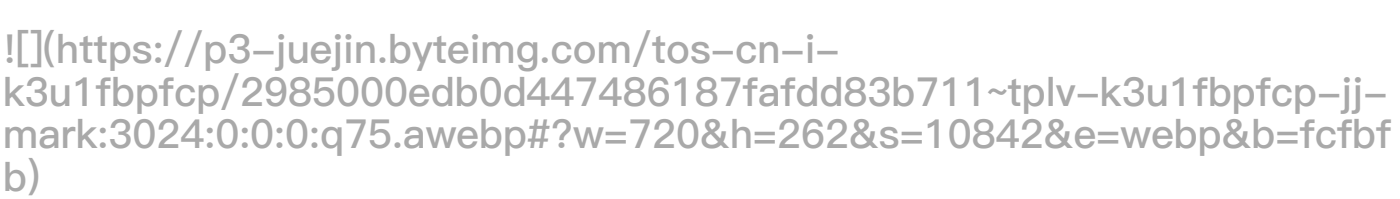
图片、文件等通过对象存储进行资源访问，以什么样的数据存储在库中，如果需要切换对象存储域名，是否方便进行替换。

四.服务应用设计

4.1 异步化

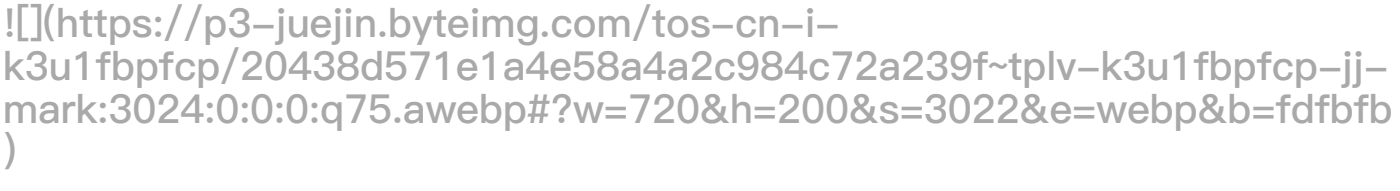
应用服务本身怎么做异步化?

比如服务如果数据操作逻辑没有顺序问题，则可以通过并发操作来完成。如果出现了错误，是要全部失败返回，还是缺失部分信息就可以?



上图是并发了多个相同的处理函数，如果其中有发生错误，则会看是否需要中断所有执行进行返回，避免资源浪费。

并发的好处显而易见，就是让整个**操作的时长等于最长的那一个，而不是所有操作时长相加。**



但是这里也需要注意一个问题，会不会给服务造成过大的TPS和QPS?

如果并发度是8，一个操作增加了并发逻辑之后，最高可能会把QPS 放大八倍，这个时候如果并发的操作都是在不同的微服务上，那也只是流量等比例打过去而已，而如果有三个操作在同一个微服务上，相当于**下游的服务承受了放大三倍的流量**，在大促的时候，这有可能就是压垮系统的一个潜在问题。

所以在并发设计的时候，要**特别去说明哪些要进行并发**，并让相关方评估流量是否能够承受，是否需要给下游的微服务增加物理资源，避免出现功能在测试环境没问题，但是上线后流量增大导致功能异常。

4.2 服务预热

服务预热一般包含

* **应用程序缓存预热**：把一些数据预先加载到内存中。
* **分布式缓存预热**：将数据预先加载到redis。
* **数据库预热**：预先初始化一些数据库数据，避免实时计算。

比如有些产品需求要达到一个列表展示的目的，但是由于数据分库表和分业务团队负责的原因，没办法简单做分页逻辑，则需要增加中间层对数据进行聚合。然后根据不同的性能指标、数据量来选择不同的聚合方案。

4.3 API 设计

这里包含不仅包含了给客户端的前端的API ，如果涉及到微服务之间的调用，还需要提供服务间调用的接口协议，比如 `protobuf` 等

五.服务治理的设计（非功能性设计）

5.1 可靠性设计

这一项属于非功能性设计，即使我们不做这一部分，功能也能够正常上线，并不会影响我们的正常使用。

但是如果应用流量很大的话，一旦出现问题，有这方面的设计就能够快速缩小爆炸的半径。

描述系统在可靠性上的分析和设计，系统在容错性上是如何处理的，**故障恢复、服务降级、熔断和限流等**的处理机制，以及在数据可靠性方面的考虑等。

5.2 扩展性和可维护性

服务也可以拆分为冷热服务，**比如经常更新迭代的和基本不变的基础服务**，这样拆分减少核心链路和非核心链路耦合发布的风险，变更范围越小，出问题概率越小。

如果流量过大是否可以使用 HPA 进行自动扩缩容？这个需要根据实际情况，如果部署拓扑相对简单，比如只依赖自己服务就能够承受住流量，则可以直接进行扩容，但是大部分的时候服务会依赖下游服务一起提供服务，如果某个节点扩容，吞吐量大的话，可能会造成连锁反应，甚至差的情况会导致下游服务无法承受流量增长而宕机。

如果出现一个错误，更倾向的是先缩小爆炸半径，不随意做扩容处理，避免造成雪崩。

我们遇到环境问题，如果不搞清楚根源就开始胡乱修改配置，最后只会引入更多的问题。

这跟我们在某个服务出现瓶颈的时候单点扩容容易导致上下游问题加剧的问题一样，**系统扩容是需要纵观全局**，如**消费速率加快会不会导致数据库无法承受压力等**。

5.3 监控&告警

必须明确需求涉及的 **核心业务指标** ，**围绕其制定监控和告警策略，**具体例子如下：

* 订单数目的监控，如果上线后订单数目增长出现明显掉0或者调到阈值以下，那么需要及时给出告警。
* 错误码是否超过业务正常阈值，如大量接口请求后返回错误码为订单状态错误，这时候需要看是否前置订单状态扭转的时候出现问题
* QPS 环比、同比的监控，如创建订单和支付订单的比值是否远低于原来的值，如果是的话可能是支付场景出现异常，需要及时排查定位。

六.上线方案

6.1 部署方案

如果涉及到多个服务，注意**服务发布的顺序**

比如有异步任务处理的 Job 进程，是否需要先进行部署，部署之后是否能够接

任原来 Pod的任务，旧任务能否兼容。

多个微服务之间是否有相互调用的依赖关系，如果相互依赖，则要规定好本次发布的顺序，切忌出现服务间的循环依赖，**因为不同 Pod的更新我们没有办法控制它们同时变成新版本。**

如果出现循环依赖，则需要看是否是应用层级划分是否需要更细，比如有些内容是否需要继续拆解成更小的微服务或者是异步处理任务，而不是都靠同步操作实现。

如果涉及到数据库变更，要看是否对现在的数据有影响，比如增加索引，是否会有锁表的风险。

表结构改变了之后，尽量不要立刻删除原来的字段，等迭代一段时间稳定之后，再逐步进行删除。

6.2 灰度方案

出现问题了的话需要回滚，回滚的方案是怎么样的，需要回滚哪些内容

数据库的库表是否需要回滚，**如何回滚？是否会产生脏数据，应该如何处理？**

发布的应用哪些需要回滚？**回滚的顺序是怎么样的？是否有副作用？**

这部分回滚需要考虑新旧逻辑是否兼容，**被新应用操作的数据旧应用是否还能继续**

6.3 数据迁移&兼容

如果在一个需求内涉及到将旧数据初始化到新表中，则需要先设计好数据迁移方案，比如上线前是否需要脚本先迁移旧的数据，然后增量数据则通过线上代码改造成双写写入数据库中，避免中间有数据缺失，然后通过数据比对来看新数据是否已经达到可用的标准。

最后在一系列操作完成之后，是否需要将原有双写方案的旧数据操作下线，减

轻数据库压力，整个都需要有详细的计划，如果无法对资源进行回收，也会给成本带来很大的压力。

拓展思考

1.性能的设计，还**没做之前怎么确定性能是否有问题？**

这个问题要从多个方面去考虑，从数据量和业务QPS 、TPS 等去评估。

如果通过业务量的评估，数据在单表的情况下会很快增长到亿级，也并不是马上需要进行分表的，比如通过合适的方法创建索引加数据归档，也是能比较简单的实现性能的提升。

2.设计的时候性能越高越好吗？

性能高意味着架构会跟着变得复杂，需要的人力成本也会增加，设计出一个能满足需求的方案，并且还能够用最低的成本来完成，这是方案设计时需要重点去考虑的。

通过人力成本和维护成本来判断，如果是一次性活动方案，性能要求也不高，设计的时候通过单表的操作即可，无需分库分表或者缓存策略设计，并且一次性需求不太需要考虑代码维护的成本，因为可能活动过去之后就不会再使用相应的代码，不维护自然没有维护的复杂度。

在一个需求被提出的时候要去**找到它的限制条件**，产品想要的都很大，我们要从错综复杂的内容中挖掘到真正合理的需求。

3.怎么设计让一个需求复用起来？中台是否真的可行？

用户体系，监控上报的通用需求是可以的，**但是不要一上来就想的太大，很难去实现，要做的是先让代码能复用起来。** 代码的复用是最简单并且降低研发成本最快的一种方式。

我们如果要让一整个功能变成中台可能很困难，但是我们在设计过程中把一个处理工具变成复用的代码却不是那么困难，当积累的足够多工具包的时候，最终写代码的时候更多的是逻辑的组装。

比如需要给一个对象数组做过滤动作，我们一开始将 `filter` 函数封装好 `input` 和 `output` ，当另外的需求需要过滤操作的时候，我们可以通过拓展方法来实现，到最后这个 `filter` 方法能满足我们90%的场景时，就可以直接当成工具包使用。

复用的好处我也不再赘述，只有真正的复用融入到研发中，我们的研发效率会更高，研发质量也会更好。

去搭建一个中台的时候也需要考虑到整体**系统的易用性。**如果很复杂导致别人无法用起来，这也是达不到我们复用的目的，价值也会趋近于0。

比如应用方是否能够通过 SDK 快速去接入，然后**无需配置或者通过简单的配置就可以立即使用。**

这个迁移到我们写代码中，在调用别人的工具方法时，我们肯定更倾向于传入关键的参数即可，想要计算两个字符串的相似度，入参就是传入需要对比的字符串，虽然背后的相似度计算算法很多，但是我们可以先**默认选择一个**，如果调用方没有传就用默认的计算方法，但是也**保留能够让调用方使用的入参**。

```
// 1.默认参数设置相似度计算的算法
func Default() OptionFunc {
return OptionFunc(func(o *option) {
if o.cmp == nil {
l := similarity.EditDistance{}
o.cmp = l.CompareUtf8
if o.ascii {
o.cmp = l.CompareAscii
}
}
})
}

// 2.调用Option接口设置option，如果没有传则直接用默认值填充
func (o *option) fillOption(opts ...Option) {
// 应用传入的值
for _, opt := range opts {
opt.Apply(o)
}
```

```
// 应用默认值
opt := Default()
opt.Apply(o)
}

// 3.比较两个字符串相似度
func Compare(s1, s2 string, opts ...Option) float64 {
var o option

o.fillOption(opts...)

return compare(s1, s2, &o)
}
```

原文链接: https://juejin.cn/post/7385156591027273754