朱晔的互联网架构实践心得 S1E9：架构评审一百问和设计文档五要素

【下载文本 PDF 进行阅读】

本文我会来说说我认为架构评审中应该看的一些点，以及我写设计文档的一些心得。助你在架构评审中过五关斩六将，助你写出能让人收藏点赞的设计文档。

技术架构评审

架构评审或技术方案评审的价值在于集众人的力量大家一起来分析看看方案里是否有坑，方案上线后是否会遇到不可逾越的重大技术问题，提前尽可能把一些事情先考虑到提出质疑其实对项目的健康发展有很大的好处。很多公司都有架构评审委员会都有架构评审的流程，做业务的兄弟要么看到这个流程往往心惊胆战害怕自己做的方案被毙了咋整，要么就是认为这是一种过场，随便糊弄一点文档发给委员会看一下就算过去了。我做过架构评审委员也做过提交评审方案的业务兄弟，不管哪个角色我都不害怕而是喜欢这一流程，评审这个事情做的好其实可以很享受，大家都是一起学习的过程，不存在谁为难谁。下面说说我觉得架构评审（非代码评审）中看重的需要评审的一些点。

组件选型

- 为什么选 A 不选 B 呢？
- A 不是开源的，出了问题怎么办？
- B 虽然是开源的，但是是 Erlang 写的，公司没人能看懂怎么办？
- C 我看待解决的 Issues 还有很多，有没有去了解过？
- 这个组件在性能方面你是否了解过？
- 开源的免费版本不支持集群怎么办？
- 如果彻底要自己写这个组件有没有可能性？

组件特别是存储组件选型挺重要的，真心建议事先可以有那么几周的时间搭一个高可用的集群，使用接近于真实的数据对组件进行压测（你看之前我博客上的 Mongodb 的压测文章停火的，说明很多人没有这个时间和条件对一些组件进行压测）。眼见为实耳听为虚，自己通过压测对比一下自己得出的数据和公开的数据是否有差异，如果有的话说不定还能发现自己使用上的一些问题。尽量还是选用使用的人多的开源组件吧，出了问题至少 Patch 不会来的太慢。

性能

- 我们需求的 TPS、QPS 和 RT 是多少?
- 整体设计上会做到的 TPS、QPS 和 RT 是多少?
- 随着数据量的增大系统性能会不会出现明显问题?
- 系统哪个环节会是最大的瓶颈?
- 是否打算做压力测试，压力测试方案是怎么样的?
- 怎么提高前端用户的访问流畅性?

对于重要的项目，建议做一下整体压测，没有经过压测得出来的估计的结论往往可能是错的，我们总以为最终会死在最后的存储上，但是很可能早早就死在了程序的低级错误上，比如在一些存储组件的 Client 使用上，如果没把控好最佳实践（把应该作为单例使用 Clinet 每次都去创建一次，默认的线程数小的可怜应该重新配置但是保留了默认值），死的非常难看。

可伸缩性

- 每一个环节是否都是可以横向扩展的?
- 扩容需要怎么做手动还是自动?
- 数据库不能横向扩展怎么办?
- 纵向扩展有多少效果?
- 横向扩展是否是线性的?
- 扩展后是否可以提高响应速度?

灵活性

- 是否有了解过产品层面以后会怎么发展?
- 模块 A 是否能拆分出去独立为其它业务服务?
- 模块 B 是否可以替换为另一种第三方数据源?
- 如果流程有变，需要多大的工作量来适应?
- 业务是否可以做到可配?

可扩展性

- 为什么 A 和 B 都有差不多的逻辑?
- 是否考虑到了 A 业务的实现以后还有 B 的可能性?
- 如果现在有两种策略以后扩展到了八种策略怎么做?
- 以后是否可以把这个业务的 H5 前端适配到 PC?

可靠性

- 是否架构中有单点?
- 故障转移是怎么实现的?
- 集群内部故障转移需要多久?
- MQ 或存储出现问题的时候系统会怎么样?
- MQ 或存储出现问题又恢复了系统是否会自己恢复?
- 是否考虑过异地故障转移的方案?
- 是否考虑过多活的方案?
- 是否有数据丢失的可能性?
- 数据丢失后是否可以恢复?
- 系统彻底挂了对其它业务的影响是什么?
- 系统彻底挂了是否可以有线下的方式走业务?

安全性

- 是否彻底避免 SQL 注入和 XSS?
- 是否做了风控策略?
- 是否有防刷保护机制?
- 数据库拖库了会怎么样?
- 是否有数据泄露的可能性?
- 数据的权限怎么控制的?
- 功能的权限是怎么控制的?
- 是否做了日志审计?
- 受到了 DDOS 攻击怎么办?
- 数据传输是否加密验签?

兼容性

- 老的系统打算怎么办?
- 怎么进行新老系统替换?
- 新老系统能否来回切换?
- 别的系统怎么连接你这套新服务?
- 上下游依赖是否梳理过，影响范围多大?
- 上下游改造的难度怎么样?
- 上下游改造有排期吗?
- 上下游改造的计划和通知时间确定了吗?
- 使用了新的数据源数据怎么迁移?
- 使用了新的技术老项目开发能否适应?

弹性处理

- 这个数据重复消费会怎么样?

- 这个接口重复调用会怎么样？
- 是否考虑了服务降级？哪些业务支持降级？
- 是否考虑了服务熔断？熔断后怎么处理？
- 是否考虑了服务限流？限流后客户端表现怎么样？
- 队列爆仓会怎么样？
- 是否考虑了隔离性？

事务性

- 这段业务由谁保证事务性？
- 数据库事务回滚后会怎么样？
- 服务调用了失败怎么办？
- 队列补偿怎么做的？
- 服务调用补偿怎么做的？
- 数据补偿实现最终一致需要多久？
- 在数据不完整的时候用户会感知到吗？

可测试性

- 测试环境和线上的差异多大？
- 是否支持部署多套隔离的测试环境？
- 是否打算做单元测试，覆盖率目标是多少？
- 测试黑盒白盒工作量的比例是怎么样的？
- 是否支持接口层面的自动化测试？
- 是否有可能做 UI 自动化测试？
- 压测怎么造数据？
- 是否可以在线上做压测？
- 线上压测怎么隔离测试数据？

- 是否有测试白名单功能?

可运维性

- 每一个组件对服务器哪方面的压力会最大?
- 重新搭建整套系统最快需要多少时间?
- 系统是否可以完全基于源代码构建?
- 系统是否有初始化或预热的环节?
- 系统里哪些环节需要人工参与?
- 数据是否需要定期归档处理?
- 会不会有突发的数据量业务量增大?
- 随着时间的推移如果压力保持不变的话系统需要怎么来巡检和维护?
- 怎么在容器里进行部署?

监控

- 业务层面哪些指标需要监控和报警?
- 应用层面系统内部是否有暴露了一些指标作监控和报警?
- 系统层面使用的中间件和存储是否有监控报警?
- 是否所有环节都接入了全链路跟踪?
- 出现报警的时候应该由谁来处理?
- 每一个模块是否有固定的主要和次要负责人?
- 有没有可能系统出了问题无法通过监控指标体现?
- 哪些指标需要上大屏由监控进行 7*24 监控?

看了这么多问题可能会觉得这架构设计是没法做了，其实不同阶段的项目有不同的目标，我们不会在项目起步的时候做 99.99%的可用性支持百万 QPS 的架

构，高效完成项目的业务目标也是架构考虑的因素之一。而且随着项目的发展，随着公司中间件和容器的标准化，很多架构的工作被标准化替代，业务代码需要考虑架构方面伸缩性运维性等等的需求越来越少，慢慢的这些工作都能由架构和运维团队来接。一开始的时候我们可以花一点时间来考虑这些问题，但是不是所有的问题都需要有最终的方案。

技术设计文档

如果你在 Google 搜索架构设计文档、技术设计文档、概要设计文档可以搜索到很多模板，很多公司也会以这些模板作为设计文档的模板来让大家填写。对于大部分所谓的项目只是一个项目中的一个小环节是一个具体的业务逻辑，因此总是可以看到大家写的所谓的技术设计文档只能填满文档的 20%，其余都不知道怎么写。当你不知道文档应该写哪些内容的时候可以这么来考虑问题，就是你这个项目接下去是要卖给别人来用的，你没有机会当面和他把这个项目说清楚，你只能通过文档来把事情写清楚，别人就给你一次机会，如果看不懂你文档表达的意思，不能理解你的项目，你的项目就卖不出去不值钱，这个时候你一定会从各个角度来剖析你的系统想尽一切办法来把事情说清楚，用这个方法逼一下自己的，你的文档会很优秀。接下去我想说一下如果是我的话我看重技术设计的哪些部分以及这些部分的文档的写作方式，这些内容构成了一个必要的（概要）设计文档，算是抛砖引玉。

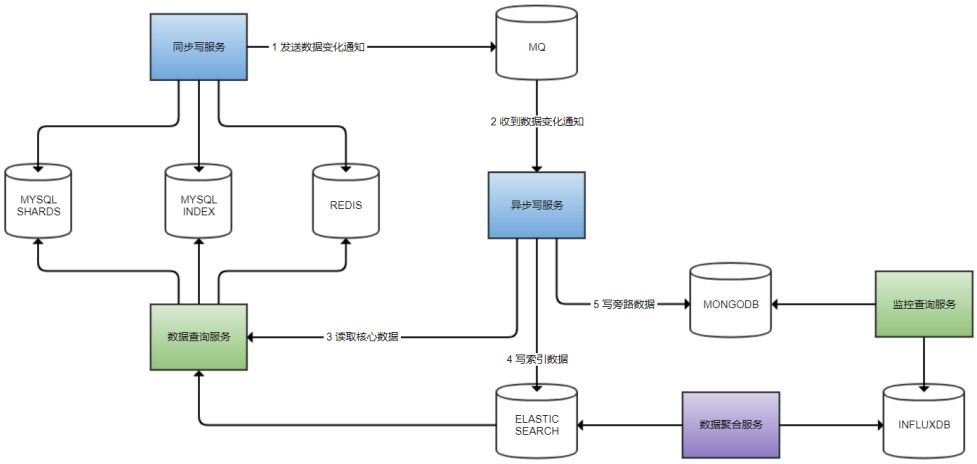
交代背景和需求全貌

在这里，推荐使用脑图在技术角度给出一下自己理解的项目需求的分布。PRD中的产品功能脑图可以和这里技术角度的脑图有差异，在说清楚需求的同时侧重技术层面，体现在：

- 可以不按照需求的功能点进行归类而是按照实际项目归类，把需求列在实际的项目下面
- 可以区分需求是自己干的还是调用外部的接口，可以在图上以不同的形态提现
- 可以画出功能之间的依赖关系，以虚线实线进一步区分依赖的程度
- 可以在图上体现需求的优先级以及需求目前的进展和负责人，这样基本一个图就可以看清楚这个项目需要消耗多少资源需要多久可以结束

看了这个图基本产品需求就可以理解个大概，具体的细节规则可以进一步参考PRD。

系统架构图

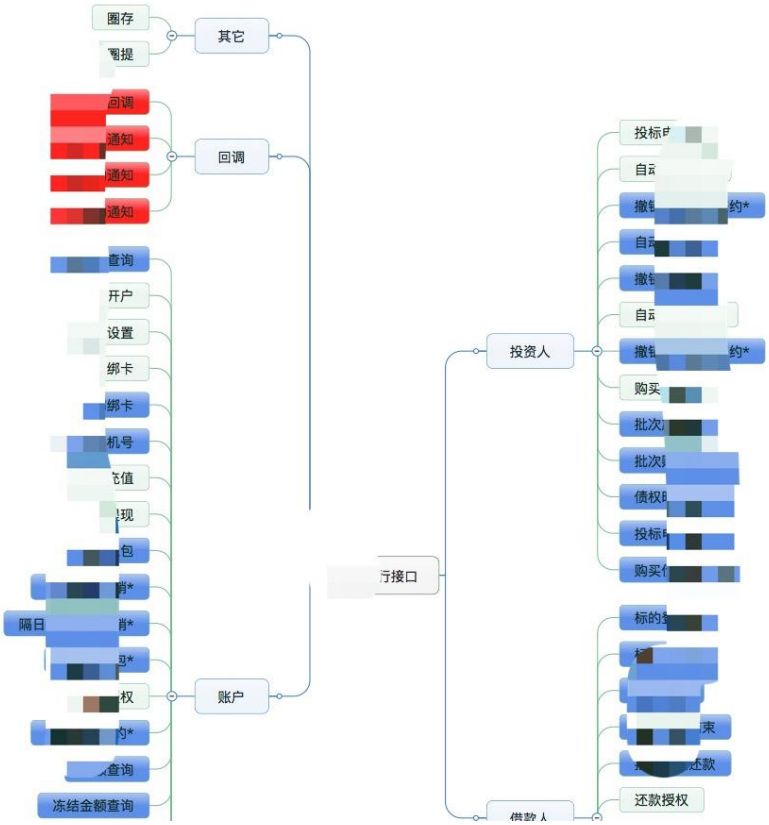


在本系列文章的第二到第五篇中，我都配了一个架构图。架构图需要传递清楚下面的信息：

- 项目由哪些模块、服务、缓存、存储构成，可以以不同的图案和颜色代表不同类型
- 模块之间的依赖关系（当然，也可以从数据的流向角度来画）
- 核心流程的步骤，沿着图上的 1、2、3 基本就可以大概了解核心流程的实现
- 可以用大的框把组件进行分组来描述组件的部署方式（比如相同机器上承载的组件在一个框内）
- 可以以边框的虚实来分类项目内的组件或三方组件，可以以箭头的虚实来标记主要流程次要流程等等

UML 里会有一些具体的分类，什么类图、组件图、部署图等等，我觉得不必拘泥于这些细节，通过线线框框的架构图能把模块和模块之间的关系表述请求，然后再配以一定的文字来说明每一个组件即可。我自己常用的是 gliffy，只要能说清楚，Word 画也可以。根据项目的大小，图上的模块不一定是需要独立部署的进程，模块也可以是项目内部的一个模块或类。对于复杂的项目，要画一个图说清楚很难，可以画一个大的架构图，然后针对每一个模块或流程再细化画不同层次的图。

对外 API 定义



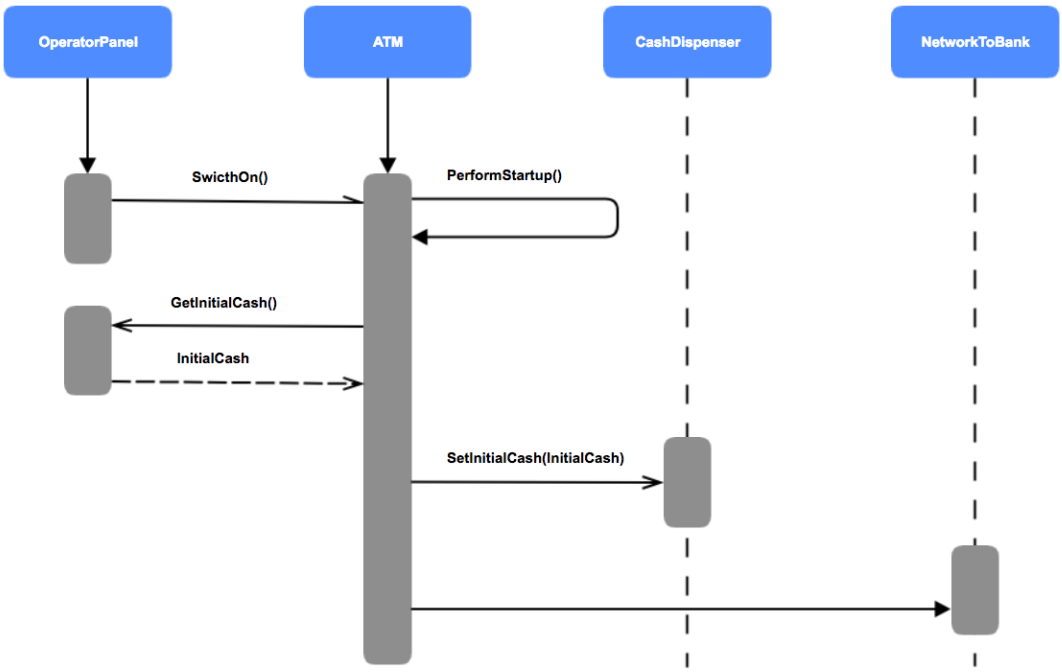
API 的详细定义可以由 Swagger UI、Spring REST Doc、Miredot 等等工具生成，这些生成的接口是按照代码来组织层次关系的，只能体现接口的参数定义不能体现接口的形态等，是没有思想的，不适合用来阅读，只适合用来参考。因此还是建议做一个脑图来总体阐述一下接口的：

- 分类，按业务功能的分类，按受众的分类等等
- 形式（图上不同的颜色），同步还是异步（结果不在响应中，单独的回调返回），单条还是批量，数据接口还是页面调用等等
- 重要性（图上文字后的星号），比如重点标记主流程的接口

图上可以不显示出参数清单，但可以以简单的文字来描述重要参数，比如下单接口：->@用户身份，优惠券 ID*，[{商品 ID,数量}]<-订单号，下单结果（0-失败，1-成功）。（->代表输入，<-代表输出，[]代表数组，@代表隐式参数，*代表可空参数等等）。

之前强调过好多次涉及到和外部交互的 API 是设计中非常重要的一个环节，不仅仅体现了系统对外输出的能力，也体现了系统设计在安全性、复用性、封装等方面的平衡。

交互时序



时序图的表达非常重要，可以表现需求脑图、架构图和 API 脑图无法表现出来的几个方面，清晰展现了某个事情：

- 关键的利益关系方。这个事情由哪几个方面构成，可以是用户、甲方、乙方这么来分，也可以是用户、APP 客户端、服务端这么来分
- 每一方在做什么，依赖的又是什么，整个顺序是怎么样的
- 技术层面这是同步接口、还是回调、还是非技术的线下流程
- 还可以在图上表现出可选逻辑，条件判断逻辑，循环逻辑等等

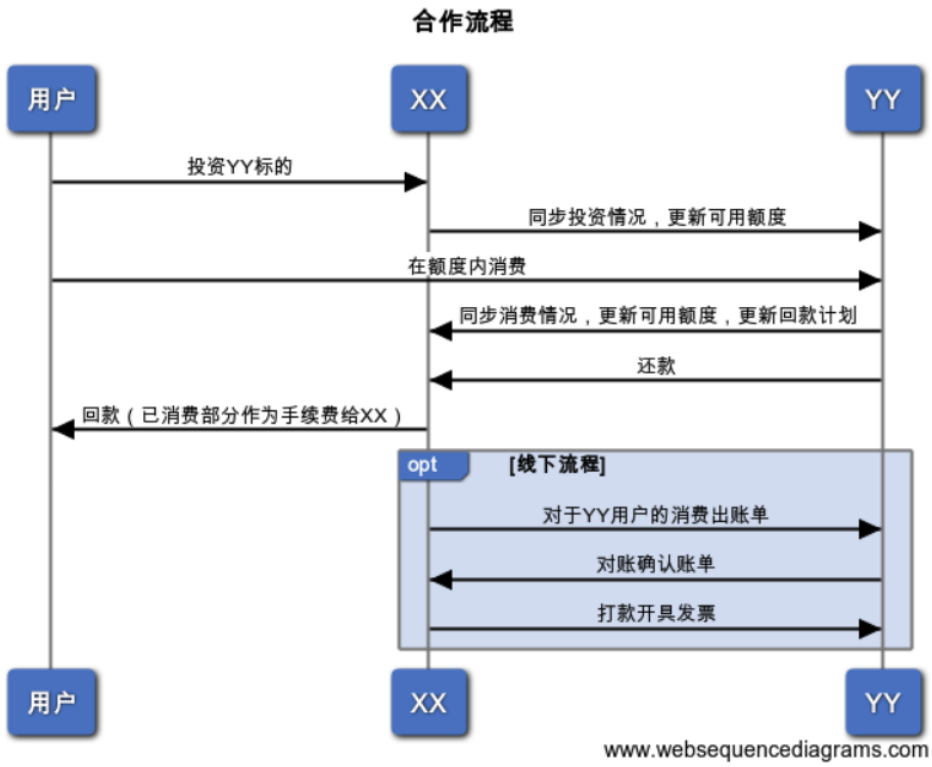
我觉得能在比较高的层面说一下技术（对接）流程即可，不一定要详细到类和类之间的交互，类和类之间的交互阅读代码或直接看全链路调用的图就可以。如果项目有多个合作方多个依赖方，项目流程比较复杂，那么序列图是能把这个事情说清楚的最好的方式。

对于这种时序图，采用传统的工具来画费时费力，推荐下面两个工具（https://www.websequencediagrams.com/和 http://plantuml.com/sequence-diagram），可以在几分钟内生成需要的图。

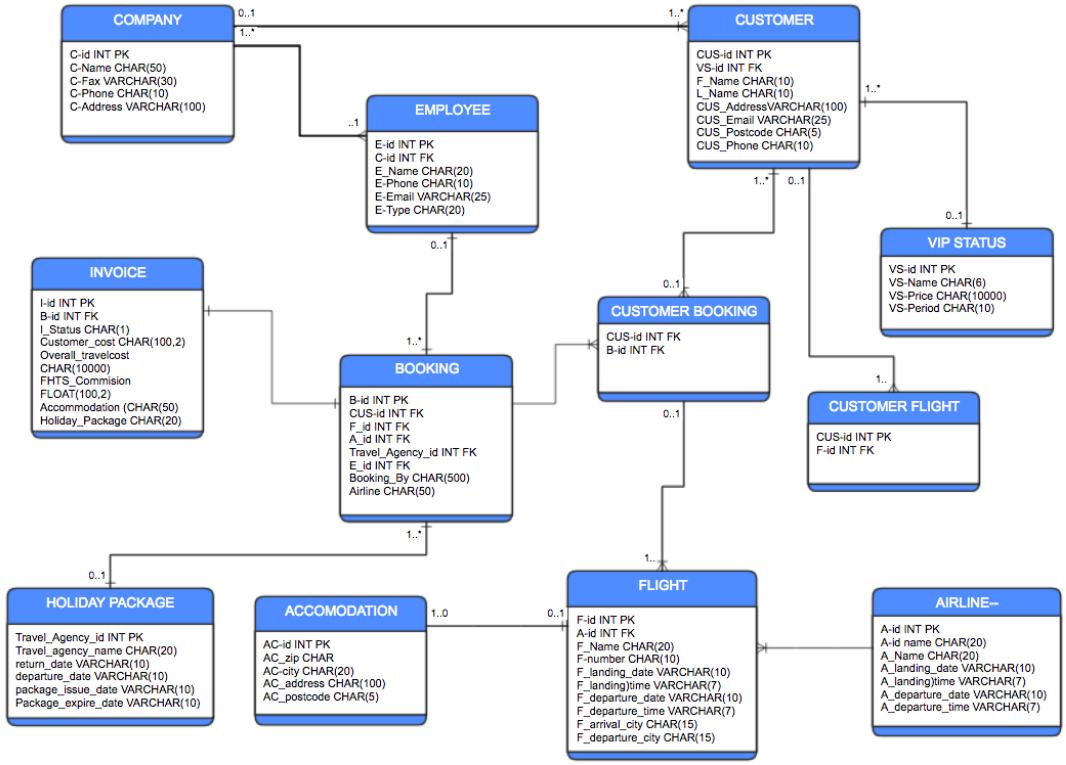
我们输入类似的文字：

```
title 合作流程
用户->XX:投资 YY 标的
XX->YY:同步投资情况，更新可用额度
用户->YY:在额度内消费
YY->XX:同步消费情况，更新可用额度，更新回款计划
YY->XX:还款
XX->用户:回款（已消费部分作为手续费给 XX）
opt 线下流程
XX->YY:对于 YY 用户的消费出账单
YY->XX:对账确认账单
XX->YY:打款开具发票
End
```

网站生成类似的时序图，还可以自由选择自己喜欢的样式：



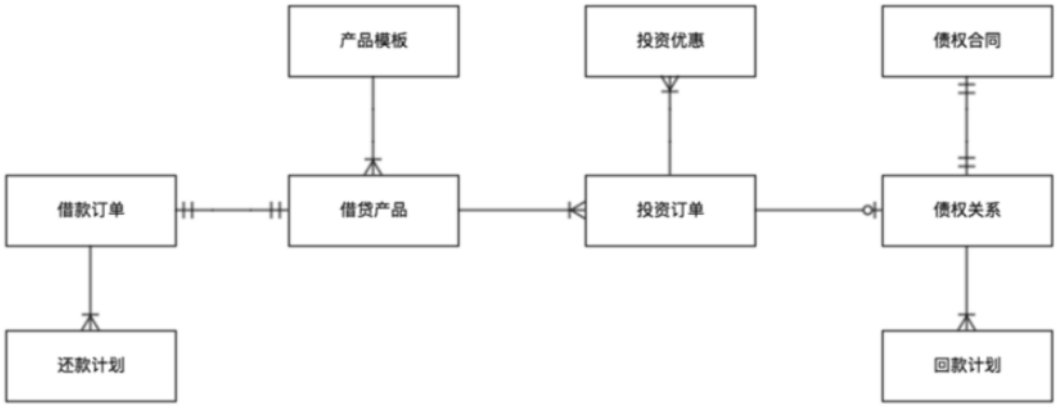
数据库 ER



ER 图就是实体联系图。形式上我们可以在图上表现几个点：

- 实体：哪些表
- 实体上的属性：体现实体之间关系以及实体业务功能的重要字段
- 联系：实体和实体之间的关系，比如一对多，多对一还是多对多之类

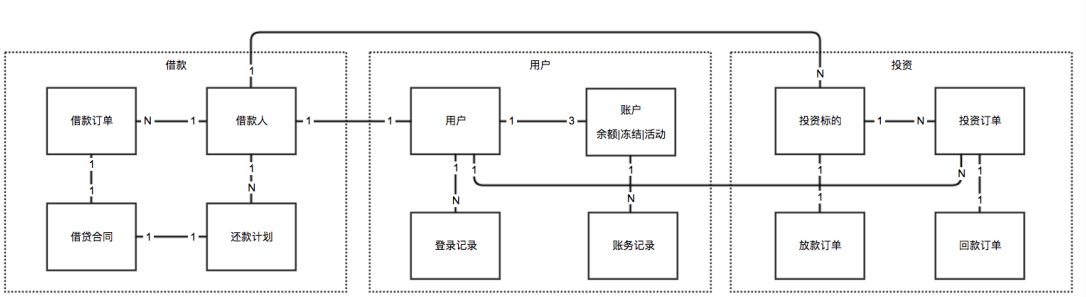
在有的时候我们可以省略属性的类型定义，甚至可以直接省略具体的属性（实体名和 M 对 N 的关系是必须的），把图简化为类似下面的部分：



ER 图的信息量非常大，绝对不是粘贴一下表结构的 DDL 可以替代的，原因是：

- ER 图可以以极小的空间展现很多信息，这样我们可以在图上对业务进行分组，看到全貌
- ER 图展现的是表和表之间的关系，一眼可以看出最重要核心的表是哪些

比如下图，是否一眼就可以看明白一套 P2P 金融业务整个数据结构的架构呢：



（随便画的图，不代表任何意义，请不要以这个图做 P2P 的表结构设计）

所有的图，文字只是一个维度，我们要学会利用边框类型（矩形、圆角矩形），边框样式（虚线，实线），填充色，文字颜色，关联线条粗细颜色样式，框内 ICON 来增加我们要表现信息的维度，最多可以增加到 6 维+，这种能力是文字很难实现的。一图胜过千言，所以我一直认为图是设计文档中非常重要的部分。

其它

之前我说了我们可以以五图的形式（需求脑图、架构图、API 脑图、序列图、数据库 ER 图）把系统大概介绍一个底朝天，说清楚了需求、架构、对外接口、交互流程和数据结构设计这几个事情，业务项目说清楚这些足够了。
对于偏向于中间件（不管是基础中间件还是业务中间件，中台）的项目（而不是业务项目），这里我再补充几个重要的方面，需要在设计文档中有体现：

- 可靠性：是否有单点的组件，非单点的组件如何做故障转移
- 高性能：是否有抗突发性能压力的能力，大概可以满足多少的 TPS 和 QPS，怎么去做来实现高性能

- 可扩展：随着压力上升哪些环节可以做扩展，怎么做扩展
- 安全性：哪些手段防突破，万一突破了后果怎么样
- 兼容性：和遗留系统怎么通讯，怎么做迁移
- .....等等方面

这些点我就不一一展开说了，在第一节说架构评审的时候都有提到过，针对那些问题写一下自己的设计是怎么应对的吧。

这些点我认为可以构成一个合格的设计文档，文档的形式不重要，重要的是可以把业务的技术实现梳理清楚，确保我们在开发之前有一个清晰的思路，在开发上线后，文档也是一个后人熟悉系统的非常重要的手段。你可能会提出疑问说这样的设计文档是不是太粗略了一点，完全没有体现到软件层面设计的细节，没错是这样，但是我一直说的是互联网架构心得，敢问现在互联网项目从 0 开始的大项目 1 到 2 个月上线，大的版本迭代 2 周一次，如果设计的时间是五分之一的话，设计也就是 2 天到一周这样子，我们有多少时间和能力来细化文档呢，如果能把我这里说的五要素都做好，对于互联网项目已经笑死。
还有一点往往会比较可惜，我们或许可以做到在开发之前有一个概要设计文档的产出，但是我们很难做到在系统上线后随着迭代还能继续维护第一版产品上线时那个大而全的文档。随着产品的迭代，我们的技术文档也像 PRD 迭代文档一样只说这次迭代的技术改动的话，这种设计文档因为没有全局观意义不大。
对于这个情况，我觉得对于每一条业务线的产品，我都建议我们至少能维护大而全的下面这些文档的全量版本：

- 完整表结构（顺带标一下归档方案、重要程度）
- 需求全貌
- 对外产品能力输出全貌
- 整体架构图
- 关键业务交互流程（特别是那种很难说清楚的多方结算关系）

定期回顾一下这五个文档，根据最近的需求改改，可能也只需要花费几小时的时间，对于大项目其意义往往是新人的灵魂导师（之前我有画过一个比较复杂系统的架构图，这个架构图我看到有人做了桌面）。