腾讯软件工程师是怎样写设计文档的?

腾讯程序员 InfoQ 2022-08-11 15:51 Posted on 北京

作者 | marinewu
策划 | 闫园园

1、设计文档是什么?

设计文档是软件工程设计中的重要组成部分，是对一个技术问题的解决方案的系统性描述。设计文档的目的，是阐明设计的总体思想和设计中考虑的权衡点。

作为一名软件工程师，我们的工作本质不仅仅是编写程序代码，而是解决真正的问题。因此，相比最终的程序代码，文字形式的设计文档，在早期能够更加简明扼要地传达信息，便于让读者理解问题，找到解决方案。

除了作为系统设计的最初体现，设计文档在软件工程的开发周期中起到下面重要作用：

通过设计文档，我们可以：

- 在可以低成本迭代的时候，尽早发现设计中的问题。
  - 设计左移，代价左移，快速失败（fail fast）。
- 在团队中对设计达成一致。
  - 设计的本质是取舍 (tradeoff)。几乎所有的架构设计决策都会被挑战，原因之一是：读者并非对所有的取舍都知晓，且与作者达成共识。在设计文档中清晰地列出取舍，有利于帮助读者了解（并认可）你的决策思路，减少被挑战的可能。
- 将资深工程师的经验和思想扩展到整个团队，帮助团队成长。
  - 作为作者，可以供资浅工程师学习。

- 作为读者，可以审核资浅工程师的设计并提供建议。

- 形成团队软件设计的一致方式，沉淀团队 / 公司技术积累。
  - 企业的生命力在于知识价值的积累。

2、什么时候需要设计文档?

本身撰写设计文档是需要编写成本的。如果问题的解决方案非常清晰，没有明确的取舍，设计文档中基本都是实现描述，则应该省略设计文档而直接实现。换言之，如果编写设计文档的时间主要消耗在“写”而不是在“思考”上，则这个设计文档可省略。

当你考虑编写一个设计文档时，想一想以下这几点：

- 软件的规模是否较大，值得付出额外的编写评审设计文档的时间来降低失败的风险?
- 高级工程师无法确保 CR 每一份代码，让他们参与设计评审是否回报更高?
- 软件设计决策是模糊的，甚至有争议。有必要围绕设计文档在组织上达成共识?
- 是否需要通过设计文档强调项目交叉问题，如隐私性 (Privacy)、安全性 (Security)、日志记录？是否有必要写一份文档来对有关遗留系统的设计问题提供高层次的分析?

如果以上的问题的答案为“是”，那么设计文档可能是开始你的下一个软件项目的绝佳方法。

3、设计文档要怎么写?

在考虑通过用设计文档解决问题，开始着手准备设计文档前，需要厘清设计文档易混淆的三个概念，它们也是创作设计文档的根基。一旦出现偏差，我们认真撰写的文档很有可能完全不可用，在纠正偏差时也会出现大量工作返工，造成资源的浪费。所以，撰写设计文档前需要搞清楚这些前提！

3.1 撰写设计文档的三个前提

前提一：设计文档的读写比最高

实现代码、系统接口、设计文档，读写比（内容被所有人阅读花费的时间：内容写作花费的时间）是逐步上升的。通常，设计文档供阅读的时间往往远多于写的时间。因此，编写设计文档时就更多考虑读者的体验而非作者的体验。为提升设计可读性的时间非常值得投资。

前提二：设计文档不是文学写作

设计文档的目的是为了沟通设计，而不是为了自我表达。把精力放在如何清晰、简洁地表达，而非放在文采上。

前提三：设计文档为谁而写

首先先了解你的读者是谁? 在良好的文档分享文化下，读者不应该只是你的 TL 以及该设计文档的实施者，你的设计文档实际读者的范围往往大得多。在不确定的时候，经验做法是，假设的读者群体为：公司内部的、有一定工程经验的、但对该系统的上下文只有初步了解的软件工程师。

通常，设计文档的范围越大，假定的受众群也会更大。这意味着受众对目标系统的平均了解程度更低，也就意味着设计文档往往需要：

- 更加详细的背景介绍。
- 更少使用内部术语或缩写。
- 更多阐述设计思路、取舍，更少解释具体实现细节。

其次，考虑读者喜欢如何阅读? 大部分读者不会逐字逐句阅读你的设计文档。大家都很忙，通常只会扫描大体结构，然后阅读（或者跳读）自己感兴趣的部分。读者喜欢“故事”。将内容以故事的结构呈现最容易被接受，即使我们并不是需要讲述一个传统的打怪升级的故事。虽然故事内容各有不同，但大部分故事都遵循一些基本的范式。例如， 约瑟夫·坎伯总结提出全世界大部分神话故事都符合 “英雄之旅” -- “启程、启蒙、归程”三幕 -- 这个模式。

对于设计文档 / 科技论文写作，通用安全的选择是 Writing Science 所介绍的 OCAR 故事结构。

- Opening：开场，背景介绍
- Challenge：挑战，所要解决的问题
- Action：行动，执行的实验 / 设计 /...
- Resolution：结果

设计文档通常遵循特定的组织结构，我们可以将每一个结构对应到 OCAR 的不同部分，以此讲述故事。

最后，辩证看待设计文档可读性。设计文档可读性 vs. 代码可读性都称作可读性，两者有些共通之处：

- 文档着重强调的内容应该是并非显而易见的事项：

设计文档	代码文档
说明背景	说明上下文
强调 Why/权衡/其它方案	强调 Why
注意事项	注意事项
代码可以清晰表达的内容不需要文档	代码可以清晰表达的内容不需要文档

- 没有绝对的正确答案：没有完美的代码，也没有完美的设计文档。
  - 不同的读者对可读性的理解有细微的不同， 可读性是主观的。
  - 在实践中，我们追求让更多（而非所有）读者更顺畅地阅读设计文档。
  - 不要为完美主义付出过重代码。平衡可读性与时间成本。
- 我们的目标是有意识地提高文档写作 / 代码水平。高质量的写作是一种习惯。提高水平的方法有：

- 多读他人优秀的设计文档。
- 评审（Design doc review/Code review）有益。
  - 设计文档评审往往主要关注系统设计合理性，但是可读性方面的评审也有必要。
- 多写作、多修改、多重写。

3.2 设计文档的核心原则

在搞清楚设计文档撰写的三个前提后，就会进入文档的编写阶段。设计文档是非正式的文档，因此他们的内容不会遵循严格的准则，一个首要原则是，针对项目的具体情况可以用相对合理的方式来编写。尽管如此，笔者也参考文献并结合自身经验给出一些建议。

写作风格的三要素

设计文档的写作也是技术写作（Technical Writing），因此同样强调以下三要素：

- 清晰
- 简洁
- 优雅

设计文档的五个要点

系统设计及编写设计文档时需要注意的 5 个要点。

1. **任何架构问题都是取舍。**

在软件设计中，没有任何一个维度有绝对意义上的优劣。每一个设计决定都需要考量很多相违背的因素。例如，可扩展性和效率相背；长期效率和短期收益相背；规模化提升了效率，但降低了灵活性。“高内聚低耦合” 便于迭代，但是会增加短期的开发成本。NoSQL 比 SQL 性能高，但代价是功能的大幅降低。

如果一个设计决策看上去没有任何的取舍，往往是因为取舍还没有被识别。在设计时应从取舍视角切入，寻找不同需求间的平衡。

2. “为什么” 比 “怎么做” 更重要。

设计所解决的问题往往是复杂而模糊的，因此，解决方案往往是不唯一的。对工程设计，方案的论证通常比方案本身更重要。

3. 考虑时间维度

做设计取舍时不能忽略时间维度，只设计某个阶段的终态。设计需要考虑以下方面：

- 可维护性与可扩展性考虑
- 实现路径
- 考虑未来计划

4. 避免过度设计

设计伊始，界定问题的范围。一个良好界定的问题是一个良好设计的必要条件。

不要迷信设计模型、设计模式、XX 驱动设计，这些是工具，而非法则；不要为了制造问题而解决问题；不要通过复杂的设计来体现工作的难度和深度: 一个困难的问题可能会有一个简单的答案；也不要过于担忧设计被迅速淘汰。保留可扩展性，但不要在未知时浪费精力扩展。

5. 总结

最重要的是要知道如何设计，知道自己在设计什么。邓宁·克鲁格效应告诉我们，这未必显然。

I think test-driven design is great. I do that a lot more than I used to do. But you can test all you want and if you don’ t know how to approach the problem, you’ re not going to get a solution.

- Coders at Work, Peter Norvig

3.3 设计文档的最佳实践

遣词

用词要简练、准确、直白。

- 正确使用专业术语。
  - 合理地使用常见术语可以降低沟通成本。
  - 不要过多使用过于小众或自创的术语。
    - 必要时提供对照的英文术语以方便理解。
    - 避免无上下文的缩略词。
- 省略程度副词。
  - 不管作者意图为何，“非常重要”和“重要”在读者看来大同小异。
- 使用数据。
  - 与其说明“该系统的性能提升明显”，不如“该系统的性能提升了 42%”更为可信，也更方便读者做出自己的判断。
- 忌佶屈聱牙。
  - 例如上文，应改为“不要使用过于生僻的词汇，不要过度使用书面语”。
  - 千万不要写文言文。

造句

- 使用短句，不要使用多从句的复杂句式。
  - 读者不是来考 GRE 的。
  - 写文档也不是为了炫耀自己可以驾驭长难句。

“系统形式问题就是下面这样一个问题：怎样把各种不同的对象种类安排在一个系统中，以使较高的对象种类总能从较低的对象种类构造出来，也就是说前者可还原为后者。为了解决这个问题，我们必须从其相互可还原性来研究各种不同的对象种类。为此目的，我们要根据所涉及的对象领域的实际科学知识为每一个要考察的对象寻找其基本事实存在的充分而必要的条件的各种可能性。对此我们可采取下面的办法来进行，即要求这门实际科学给出基本事实的一个（确实而常在的）表征。”

Excerpt From: 【德】鲁道夫·卡尔纳普. “世界的逻辑构造.”

- 简单表达，去掉无意义的修饰，去掉试图缓和语气的从句。
  - 反例：“我们可以看到， TencentDB 在一定程度上可以满足我们对事务支持的需求。
    - ” 修改后：“TencentDB 支持事务”。
  - 反例：“MR 提交信息作为读者查阅修改历史时第一时间看到的信息，其重要性不言而喻。
    - ” 修改后：“读者查阅修改历史时会首先关注 MR 提交信息。”
  - 本段讨论另一个问题，即...
- 语气要冷静。避免过于口语化。
  - 不要加顺口溜
  - 不要使用语气词
  - 不要使用叹号！如果希望强调，使用**粗体**或者*斜体*！也可以使用分级标题！
- 准确。描述客观事实，避免加入主观情绪。

段落

段落应该尽量短。通常，一个段落不要超过 8 个完整的句子。每个段落有且仅有一个清晰的主题。每个段落开头应该是主题句，方便读者快速了解段落大意。段落中的每一句话应该与主题紧密相关；否则，它应该另起段落，或者应该删掉。

注意段落的流动。段落句子应该始于一个读者已经熟悉的概念，将新的内容放在句子结尾。这样，读者可以更连贯地理解。

使用列表

使用 Bullet point 标明无顺序的列表，使用数字序号明确前后顺序。如何正确使用列表不在本文详细展开，会在后续文章介绍（如果有后续的话），也可参见文末的参考文献。

结构

使用模板：使用模板可以作为思考辅助，同时也提供了相对较完整且规范的结构。文末提供了一份设计文档模板以供参考。

使用图表：一图胜千言。合理地使用图表可以极大地降低用户的理解成本。

使用标题：标题要分级、要简短清晰

篇幅

设计文档不要过长。太多内容堆积在一个文档中会让读者丧失兴趣。

对于一个大型项目来说，10 页（~5000 字）左右是一个合适的长度。当超过这个长度时，可以考虑将问题拆分成子问题分别编写设计文档，并在总体设计文档中链接子设计文档。

对于小问题做增量的改进，考虑使用单页文档 (one-pager)。通常这类文档的范围较小，解决问题较简单，目标用户群体仅限于对问题已经有充分了解的内部成员。这时，可以省略背景等内容，而仅使用 目标 -- 方案 两段式论证的结构。

排版

使用统一的字体。用户不会意识到不同的字体代表不同的含义，只会感受到混乱。微软雅黑是安全选择。

不要使用不同颜色来区分内容。不要在文中使用超过三种颜色。可以在标题及分级标题使用标志性的颜色，同时正文使用黑色。

附录：设计文档模板

设计文档没有定式。即使如此，笔者参考谷歌设计文档的结构和格式，并结合实际工作经验加以完善。在此提供一个可供新手参考的设计文档模版，您可以使用此文档模板作为思考的基础。通常，无须事无巨细地填写每一部分，不相关的内容直接略过即可。

目标

“我们要解决什么问题?”

用几句话说明该设计文档的关键目的，让读者能够一眼得知自己是否对该设计文档感兴趣。

如: “本文描述 Spanner 的顶层设计"

继而，使用 Bullet Points 描述该设计试图达到的重要目标，如:

- 可扩展性
- 多版本
- 全球分布
- 同步复制

非目标也可能很重要。非目标并非单纯目标的否定形式，也不是与解决问题无关的其它目标，而是一些可能是读者非预期的、本可作为目标但并没有的目标，如:

- 高可用性
- 高可靠性

如果可能，解释是基于哪些方面的考虑将之作为非目标。如:

- 可维护性:本服务只是过渡方案，预计寿命三个月，待 XX 上线运行后即可下线

设计不是试图达到完美，而是试图达到平衡。显式地声明哪些是目标，哪些是非目标，有助于帮助读者理解下文中设计决策的合理性，同时也有助于日后迭代设计时，检查最初的假设是否仍然成立。

背景

“我们为什么要解决这个问题?”

为设计文档的目标读者提供理解详细设计所需的背景信息。按读者范围来提供背景。见上文关于目标读者的圈定。**设计文档应该是“自足的”（self-contained），**即应该为读者提供足够的背景知识，使其无需进一步的查阅资料即可理解后文的设计。**保持简洁，**通常以几段为宜，每段简要介绍即可。如果需要向读者提供进一步的信息，最好只提供链接。警惕知识的诅咒（知识的诅咒（Curse of knowledge）是一种认知偏差，指人在与他人交流的时候，下意识地假设对方拥有理解交流主题所需要的背景知识）。

背景通常可以包括：

- 需求动机以及可能的例子。如，“（tRPC) 微服务模式正在公司内变得流行，但是缺少一个通用的、封装了常用内部工具及服务接口的微服务框架”。
    - 这是放置需求文档的链接的好地方。
- 此前的版本以及它们的问题。如，“(tRPC) Taf 是之前的应用框架， 有以下特点，.....， 但是有以下局限性及历史遗留问题”。
- 其它已有方案， 如公司内其它方案或开源方案， "tRPC v.s. gRPC v.s. Arvo"
- 相关的项目，如 "tRPC 框架中可能会对接的其它 PCG 系统"

不要在背景中写你的设计，或对问题的解决思路。

总体设计

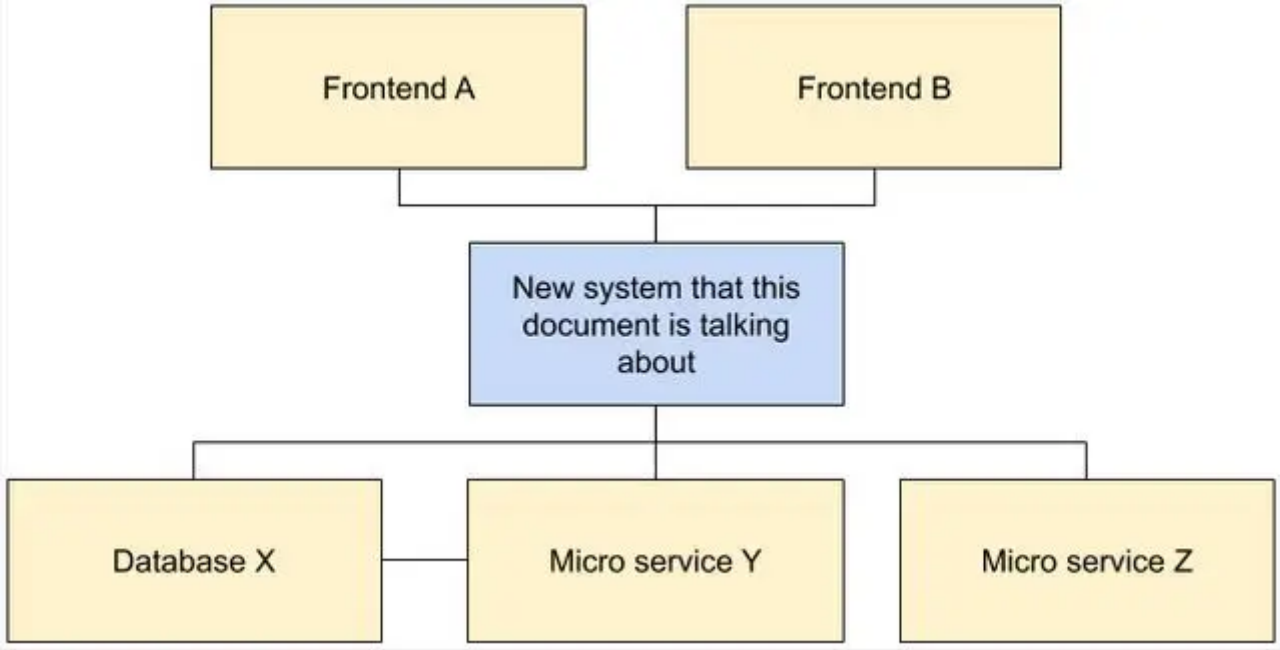
“我们如何解决这个问题？”

用一页描述高层设计。说明系统的主要组成部分，以及一些关键设计决策。应该说明该系统的模块和决策如何满足前文所列出的目标。

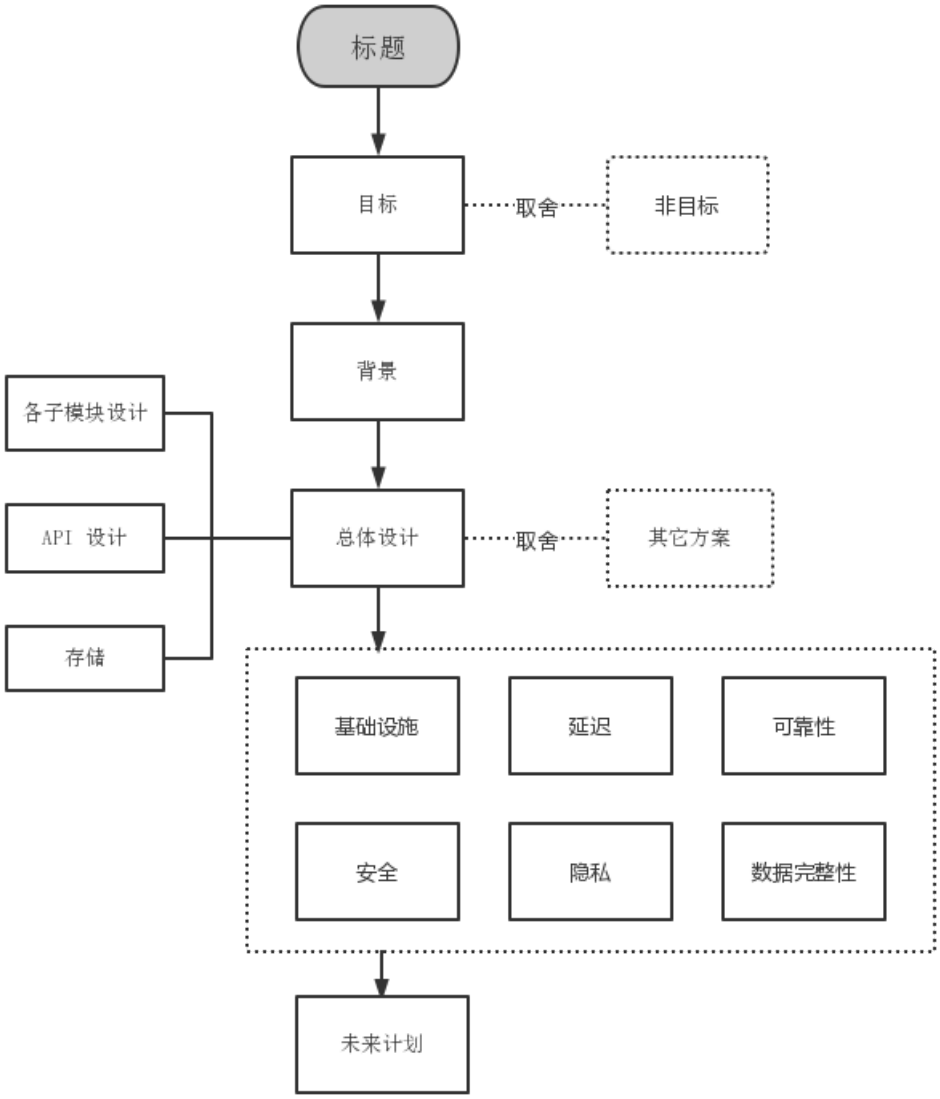
本设计文档的评审人应该能够根据该总体设计理解你的设计思路并做出评价。描述应该对一个新加入的、不在该项目工作的腾讯工程师而言是可以理解的。

推荐使用 系统关系图 描述设计。它可以使读者清晰地了解文中的新系统和已经熟悉的系统间的关系，也可以包含新系统内部概要的组成模块。

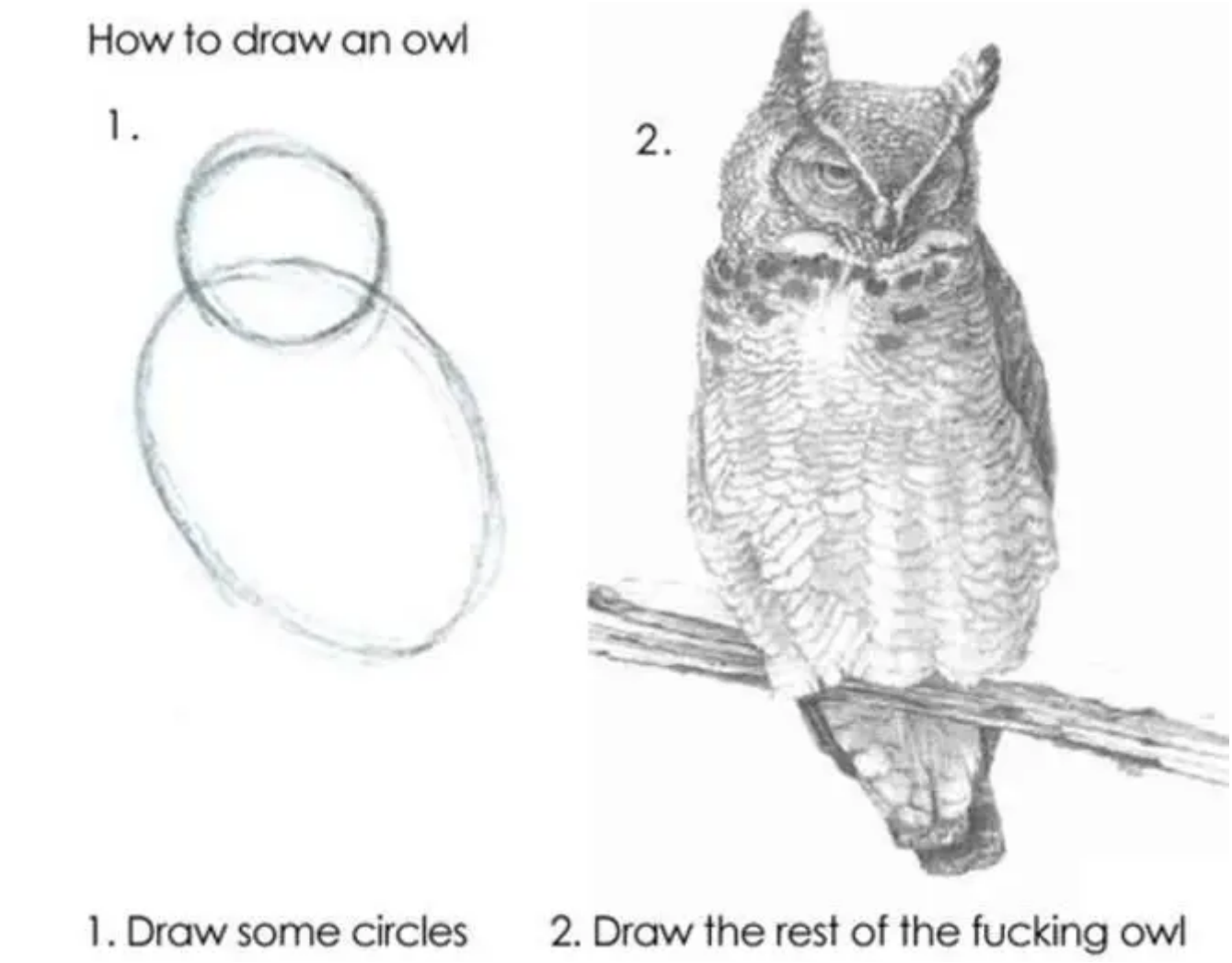
注意：不要只放一个图而不做任何说明，请根据上面小节的要求用文字描述设计思想。


Frontend A
Frontend B
New system that this document is talking about
Database X
Micro service Y
Micro service Z

一个示例系统关系图


标题
目标
取舍
非目标
背景
各子模块设计
API 设计
存储
总体设计
取舍
其它方案
基础设施
延迟
可靠性
安全
隐私
数据完整性
未来计划

自举的文档结构图



可能不太好的顶层设计

不要在这里描述细节，放在下一章节中；不要在这里描述背景，放在上一章节中。

详细设计

在这一节中，除了介绍设计方案的细节，还应该包括在产生最终方案过程中，主要的设计思想及权衡（tradeoff）。这一节的结构和内容因设计对象（系统，API，流程等）的不同可以自由决定，可以划分一些小节来更好地组织内容，尽可能以简洁明了的结构阐明整个设计。

不要过多写实现细节。就像我们不推荐添加只是说明"代码做了什么"的注释，我们也不推荐在设计文档中只说明你具体要怎么实现该系统。否则，为什么不直接实现呢?

以下内容可能是实现细节例子，不适合在设计文档中讨论：

- API 的所有细节

- 存储系统的 Data Schema
- 具体代码或伪代码
- 该系统各模块代码的存放位置、各模块代码的布局
- 该系统使用的编译器版本
- 开发规范

通常可以包含以下内容（注意，小节的命名可以更改为更清晰体现内容的标题）：

各子模块的设计

阐明一些复杂模块内部的细节，可以包含一些模块图、流程图来帮助读者理解。可以借助时序图进行展现，如一次调用在各子模块中的运行过程。每个子模块需要说明自己存在的意义。如无必要，勿添模块。如果没有特殊情况（例如该设计文档是为了描述并实现一个核心算法），不要在系统设计加入代码或者伪代码。

API 接口

如果设计的系统会暴露 API 接口，那么简要地描述一下 API 会帮助读者理解系统的边界。避免将整个接口复制粘贴到文档中，因为在特定编程语言中的接口通常包含一些语言细节而显得冗长，并且有一些细节也会很快变化。着重表现 API 接口跟设计最相关的主要部分即可。

存储

介绍系统依赖的存储设计。该部分内容应该回答以下问题，如果答案并非显而易见：

- 该系统对数据 / 存储有哪些要求?
  - 该系统会如何使用数据?
  - 数据是什么类型的?
  - 数据规模有多大?
  - 读写比是多少？读写频率有多高?

- 对可扩展性是否有要求?
- 对原子性要求是什么?
- 对一致性要求是什么?是否需要支持事务?
- 对可用性要求是什么?对性能的要求是什么?
- ……

- 基于上面的事实,数据库应该如何选型?
  - 选用关系型数据库还是非关系型数据库?是否有合适的中间件可以使用?
  - 如何分片?是否需要分库分表?
  - 是否需要副本?
  - 是否需要异地容灾?
  - 是否需要冷热分离?
  - ……

数据的抽象以及数据间关系的描述至关重要。可以借助 ER 图 (Entity Relationshiop) 的方式展现数据关系。

回答上述问题时,尽可能提供数据,将数据作为答案或作为辅助。不要回答“数据规模很大,读写频繁”,而是回答“预计数据规模为 300T, 3M 日读出, 0.3M 日写入, 巅峰 QPS 为 300”。这样才能为下一步的具体数据库造型提供详细的决策依据,并让读者信服。

注意:在选型时也应包括可能会造成显著影响的非技术因素,如费用。

避免将所有数据定义(data schema)复制粘贴到文档中,因为 data schema 更偏实现细节。

其他方案

“我们为什么不用另一种方式解决问题?”

在介绍了最终方案后，可以有一节介绍一下设计过程中考虑过的其他设计方案（Alternatives Considered）、它们各自的优缺点和权衡点、以及导致选择最终方案的原因等。通常，有经验的读者（尤其是方案的审阅者）会很自然地想到一些其他设计方案，如果这里的介绍描述了没有选择这些方案的原因，就避免读者带着疑问看完整个设计再来询问作者。这一节可以体现设计的严谨性和全面性。

交叉关注点

基础设施

如果基础设施的选用需要特殊考量，则应该列出。

如果该系统的实现需要对基础设施进行增强或变更，也应该在此讨论。

可扩展性

你的系统如何扩展？横向扩展还是纵向扩展？注意数据存储量和流量都可能会需要扩展。

安全 & 隐私

项目通常需要在设计期即确定对安全性的保证，而难以事后补足。不同于其它部分是可选的，安全部分往往是必需的。即使你的系统不需要考虑安全和隐私，也需要显式地在本章说明为何是不必要的。安全性如何保证?

- 系统如何授权、鉴权和审计 (Authorization, Authentication and Auditing, AAA）?
- 是否需要破窗（break-glass）机制?
- 有哪些已知漏洞和潜在的不安全依赖关系?
- 是否应该与专业安全团队讨论安全性设计评审?
- ……

- 数据完整性

如何保证数据完整性（Data Integrity）？如何发现存储数据的损坏或丢失？如何恢复？由数据库保证即可，还是需要额外的安全措施？为了数据完整性，需要对稳定性、性能、可复用性、可维护性造成哪些影响？

延迟

声明延迟的预期目标。描述预期延迟可能造成的影响，以及相关的应对措施。

冗余 & 可靠性

是否需要容灾？是否需要过载保护、有损降级、接口熔断、轻重分离？是否需要备份？备份策略是什么？如何修复？在数据丢失和恢复之间会发生什么？

稳定性

SLA 目标是什么？如果监控？如何保证？

外部依赖

你的外部依赖的可靠性（如 SLA）如何？会对你的系统的可靠性造成何种影响？如果你的外部依赖不可用，会对你的系统造成何种影响？除了服务级的依赖外，不要忘记一些隐含的依赖，如 DNS 服务、时间协议服务、运行集群等。

实现计划

描述时间及人力安排（如里程碑）。这利于相关人员了解预期，调整工作计划。

未来计划

未来可能的计划会方便读者更好地理解该设计以及其定位。

我们确实应该把设计限定在当前问题，但是该设计可能是更高层系统所要解决问题的一部分，或者只是阶段性方案。读者可能会对方案的完整性有所疑问，会质疑到底问题是否得到完整解

决，甚至会质疑该问题在更高层的系统中是否确实值得解决。“背景（过去）-- 当前方案 -- 未来计划” 三者的结合会为读者提供更好的全景图。

附录：参考文献

参考文档

设计文档

- 如何编写软件工程设计文档 感谢 hankzheng，本文部分原文源自此文。
- Design Docs at Google[模板]
- 设计文档模板
- Bzlmod 设计提案
- [示例] 视频后台组件平台 - 总体设计
- [示例] Unionplus K-List 存储设计替代方案
- Unionplus K-List 存储设计替代方案
- chromium 设计文档实例

技术写作

- Google Technical Writing Course
- Microsoft Writing Style Guide

参考书籍

写作 / 表达

- Style, Joseph M. Williams / Joseph Bizup
- 金字塔原理 : 思考、写作和解决问题的逻辑, 巴巴拉·明托
- 写作这回事，斯蒂芬·金
- The elements of style, William Strunk Jr. / E. B. White
- Coders at work，Peter Siebel 采访 Joshua Block 原文：

Joshua Block: "Another is Elements of Style, which isn’ t even a programming book. You should read it for two reasons: The first is that a large part of every software engineer’ s job is writing prose. If you can’ t write precise, coherent, readable specs, nobody is going to be able to use your stuff. So anything that improves your prose style is good. The second reason is that most of the ideas in that book are also applicable to programs."

- 精准表达，高田贵久
- 写作提高一点点，Mary-Kate Mackey
- On writing well, William Zinsser
- Artful Sentences, Virginia Tufte

技术写作 / 文献写作

- 写作是门手艺，刘军强
- How to write a lot, Paul J. Silvia
- Writing for Computer Science, Justin Zobel
- The craft of research, Wayne C. Booth / Gregory G. Colomb / Joseph M. Williams
- 会读才会写， Phillip C. Shon
- Writing Science, Joshua Schimel

故事

前文强调了要讲故事。以下书目阐述了何谓故事、为什么要讲故事及如何讲故事：

- Writing Science, Joshua Schimel
- 金字塔原理 : 思考、写作和解决问题的逻辑, 巴巴拉·明托
- 故事，Robert McKee
- 如何阅读一本文学书， 托马斯·福斯特

架构设计

- Software engineering at Google, Titus Winters / Tom Manshreck / Hyrum K. Wright
- Build Secure and realiable systems, Heather Adkins / Betsy Beyer / Paul Blankinship / Piotr Lewandowski / Ana Oprea / Adam Stubblefield
- The Design of Design, Fredrick P. Brooks Jr.
- Fundamentals of Software Architecture, Mark Richards / Neal Ford

图表

- Storytelling with data，Cole Nussbaumer Knaflic

列表

- https://developers.google.com/tech-writing/one/lists-and-tables

作者介绍：

marinewu： 腾讯 PCG 应用架构平台部 开发效率中心 Tech Lead,有较丰富的 API 设计与评审经验，有及技术文档的写作与评审经验。2021 年加入腾讯，主要负责 DevOps 工具体系开发和优化。

今日好文推荐