**Google如何让几万工程师高效协作？**

以下内容是我离开Google 4个月后的一些思考，也和之前一些来自google的同事有一些讨论，**不代表Google这种方式是比我们现在更好的 (所有的制度都是trade-off)，或者我在make a proposal**，(Google的方式也有很多问题和弊端: Google协作方式的问题(以及如何尝试解决) )只是给大家呈现一个不同的perspective。**这里主要聚焦在google如何能让几万工程师用同样的方式去思考和协作, 也并不包含其他的google culture。**

我加入之后发现了一些不同的地方，大家其实沟通和协作没有那么方便

* 很多事情都是靠口口相传
* 新人上手比较难
* 各个团队做事的认知不同
* 用的stack都不太一样
* 拿到对方的代码会觉得不太认可，协作相对困难。就比较倾向于重新造轮子。
* Google团队之间虽然也有竞争，但是通常大家对于一个项目应该怎么做，代码应该怎么写，这部分的认知是相对很一致的。

这里我总结了一些点，希望和大家讨论。**最根本的原因，是Google早期有一波非常强，非常固执的人 (Jeff,Sanjay,Urs,etc)，制定了规则，并且强力推进大家执行。Google的方法根本上是一种very expensive的方法，但是长期来看，很多事情反而变得容易了很多。**下面我说一些影响很大的点 （**按照重要性排序的**）

* **Committee based and Senior ICs:** 很多事情都是committee based来推行，committee的member几乎是random选择，一定会包括很多非常高级别的IC们 (Senior Individual contributors)。包括招聘，定级，升职，也包括很多技术标准的制定和迭代。这个不仅仅是标准经过讨论这么简单，更重要的是，Committee member通常都是利益不相关的人 (Senior ICs自己不管人，很多人主要的贡献还是自己个人的技术贡献)，这样能保证标准相对objective。关于技术制定和迭代，举两个我自己参与的例子
* 我加入google的时候，参与了GoogleSQL这个项目，这是google要定义一套更好的SQL language，来统一F1, Spanner, BigQuery, Dremel等。这个语言由大概大概5-10个DB领域20年经验的专家组成。每次修改都需要经过专家的同意。
* Google cloud APIs。 这里是一样的，任何一个public facing API的style都需要经过committee同意。
* **Definition of "output" in Perf:** Perf的标准不仅仅只看自己的最直接的业绩，而是对整个公司这个ecosystem的贡献。比如，有很多时候我们系统会越搞越复杂，后面就没法maintain了。这种是很常见的现象。所以这时必须有人要出来**Reduce the chaos(做减法)**。然而rewarding system决定了人们会不会去做这种事。Google一个著名的指标就是SWE hours (Software Engineer hours)。意思就是我做了这个事情后，会在未来节省多少SWE hours? 认可这样的工作一方面会让系统更robust，以后越来越好maintain，同时也会让大家更好协作。比方我看到一个可能是别人的lib或者framework，不完全适合我的使用，但是相对比较接近，这时我不会一味的追求我自己的业绩，去重新造轮子，而会跟这个lib的owner去聊，我如何帮你重构这个lib，能让更多的人使用起来。节省未来的SWE hours。这种事情我在google做过很多次。做这种事会拿到其他团队人给我非常好的peer review，这些对我的perf rating以及promotion也会很有帮助。
* **Hire smart generalists, not specialists:** specialist通常有很多bias towards the "best solution", 这些best solution通常和他们过往的经验匹配，他们也不想去了解新的东西。**这时候协作就很难了，因为每个人的背景都不同，对事情有不同的bias，都倾向于去摘自己熟悉的low hanging fruits**。然而实际上技术上的问题会经常变化，smart generalists会去用最合理的方法论找到最合适的解法。(Note: I am not claiming that the specialists can't be smart generalists)
* **Single repo, single branch**: 这个看起来是一个纯技术的决定，但是实际上对大家的认知拉齐的一个基础。**这样一个造成大家的一个核心共识是，我们在maintain同一个project协作开发**。
* google除了几个非常特殊的项目(e.g. AlphaGo)有权限设定，或者branch (Spanner)，其他项目都是全部被code search index（语义上的）好的。所以所有的dependency都必须在同一个版本上
* 大家写代码之前可以非常容易的看到类似的代码别人怎么写的
* 因为dependency都是在同一个版本上，所以会有一个很大的测试，跑整个代码库。这样每个人都有责任保证所有tests不fail。如果你会造成你的下游test failure，你有责任去帮他fix
* **我们有责任让整个codebase用同一种style去开发。**
* **Readability**: 在google有各个编程语言readability这个制度。google有自己一套style guide和设计理念。如果你没拿到readability之前，你是没法自己checkin代码的。你必须要让有readability的人，给你readability的approval。有一群readability的reviewer专门来帮大家看代码。这些算作是community contribution。大家自发的来做这个事情。具体readability的制度可以单独讨论，这里只想说的是，这是一个很好的办法能让所有人用一样的style去写代码。这里google也研发了各种tooling去保证在compiler的层面，尽量的去维持思维模式的一致性。
* **Chase perfection on APIs**: 各个团队之间很大的一个连接是API，API基本定义了协作的semantics。Google所有比较popular库的API都写的非常clean，经过无数人迭代修改。这些从launch前就被人反复讨论 (**这里面Senior ICs起了很大作用**)，launch后又会根据用户的意见无数次的修改。从来不怕麻烦去跟用户协调如何做migration。这样又是一个拉齐认知的过程。每次的改进都会被贡献到codebase里，这样每次做新的项目，在code search里都会看到之前这些API的定义，以及为什么这样定义和设计。那下一个版本的人就会考虑当时的理由，从而尽量遵守这个style，同时也有自己改进的地方。慢慢全公司的认知一起迭代升级。
* **More Info/Stats/numbers, More transparencies:** 内部有很多工具来量化参考一个项目，来辅助评估一个人的“产出”。比方你可以轻松看到一个人所有checkin的代码，每周check-in code的histogram，运行test的记录，每天submit/review代码的时间热度分布，Interview数量的统计等等。我作为一个manager，这是我在perf calibration committee和 promotion committee中使用的工具之一，可以轻松看到我的团队成员(当然也有其他人)的产出排序 (也有一个聚合的网站做ranking)。这个不能完全决定perf，但是是一个data point。除此之外，工程师很乐于开发各种各样生成stats的工具，来提供更多信息。最重要的是，所有这些信息对所有人公开 (open source, of course)，在很大程度上保证了公正性。这个为什么能提升沟通效率呢？我个人认为是能让我快速的了解一个人的工作，不必什么事都要问别人。
* **Tolerance for "Betrayal":** 硅谷的很多文化我认为很多建立在对“背叛”的宽容上面。我个人认为，只有当一个人的利益跟团队基本一致，才能创造更大的价值。人和人之间的关系是一辈子的，不应该强制一个人为了团队利益留下。所以在google经常可以见到一个人刚来了发现不合适就转组（虽然有规定一年内不能转，但是我从来没见过有director或者VP来block这个）。还有就是刚刚promotion就转组。**如果我们的peer能找到公司内更能发挥他价值的地方，这对公司难道不是更好吗？我们完全没必要阻拦，只需要确保他们想清楚了就好**。另外就是在系统上会非常鼓励这么做，比方有内部job posting的网站，你点了申请后会自动把通知对方团队，而自己的团队不会知道。对方团队可以轻松request你的perf history。让这个过程基本上非常smooth。
* **Build a community:** Google有一个内部的问答系统 (Yaqs)，非常类似stack overflow。可以提问和回答。每个人都有自己的expertises的tag。比方我之前的tag是Spanner/Spanner Transactions/Spanner SQL等。这样相关的问题会route给我。回答完问题之后也有投票系统选出最佳答案。我在公司内部的问题，很多时候都可以直接搜出来答案，只有一小部分问题需要再次提出新的问题来。
* **g3doc, not google docs:** g3doc是一个类似各个团队官网，并非documentation。这里的东西都必须经过peer review，而且有一个刷新的时间 (就是你在这个时间后必须重新自己review更新一遍，确保大家看到的是最新的)。Google docs不能提供团队官网这个概念，而且经常会有比较旧的信息，而g3doc会强制大家隔一段时间review一次，更新细节。另外，g3doc里有很多功能，大部分情况下保证你加的一些tutorial里的commandline可以直接运行。都不需要自己修改研究。

下面我做一下假设对比一下各个方面

* 招聘
* 字节：自己团队定如何招聘，会有bias，会难拉齐标准，会更倾向于招经验符合，但是人不一定raw talent最强的。**会因为团队短期的利益而妥协** (human nature)
* google：面试过程跟自己利益不相关，是完全一个统一的标准。只有在Hiring committee认可后，各个hiring manager才来沟通。
* 新人上手：新人加入团队
* 字节：看到wiki，很多旧的文档，很迷惑，copy一些command，没法运行
* 这时候去问人，很多信息口口相传，也没被记下来。下一个新人还是不知道。
* 看了代码，感觉写的很差，API定义很随意，各种layer violation。无从下手。
* 因为绩效偏向产出，心里可能也比较急，就想快糙猛上手搞。系统可能越来越难maintain
* Google: 看g3doc, copy command一般直接能跑起来sandbox，可以快速自己验证对系统的理解。
* 看一些design doc，一般写的比较详细。
* 有问题去yaqs上搜索，一般都有答案。没有答案的很方便提问。
* 看了很多核心代码第一感觉都是这些API写的真好，很多东西都详细思考过，愿意学这些API的方式去继续开发。
* 协作：考虑这样一个情况，我现在要做一个新项目X
* 字节：搜一下wiki或者飞书，看到一堆文档，各个版本的都有，不知道该看哪个。
* 看了发现里面错很多，很多都是out of date。**只能去问人**。
* 问了人后给我一个pointer，我去接着看文档或者code，发现这想法设计都跟我自己思路很不一样，感觉无从下手去改。
* 最后就倾向于自己重写，或者尽量少有外部团队的dependency
* Google：我会先去moma搜索一下可能用到的dependency的网站，看下大概情况
* 根据网站找到一些design doc读一下
* 然后去code search找一下别人怎么用的这些lib
* 然后根据自己的情况分析，能否直接用。如果能直接用，那我就直接使用了。如果不能，那我就跟lib的owner沟通一下。大部分时候，**我会主动去修改他的代码来符合更scalable的需求**，因为前面一系列的culture，大家做事的方法很接近，那如何来修改的分歧不会很大。而且各个团队在评估一个人的时候不会只以自己的业务为核心去评估，而是更全面的看给整个公司带来了多少价值。