A hundred things i learned working on the react team

author: @dan_abramov

https://twitter.com/dan_abramov/status/1470613731071696896

1. every few years your audience changes. new users don’ t appreciate problems of the past bc never seen them. old users burn out or lose excitement. new users have different reference frame, learned in diff ways, you might be their first programming env. plan accordingly.

2. when you fix a problem you better really really understand the problem you’ re fixing. take a few steps back and reintegrate new knowledge into the design. should it change the design? it’ s like fighting a hydra: solving a problem in the wrong spot spawns 10 new problems.

3. most PRs create more work than solve. even when the project’ s internals are well-documented (which for us was brief), significant contributions are rare. this is not so much because of difficulties writing code but due to months worth of context needed to make decisions.

4. for libraries, make sure your PR descriptions are great (they become squashed commit messages). for apps, include product videos before/after and urls you were testing in PR descriptor. someone (me) will thank you eight years later when diagnosing a bug or planning a change.

5. a couple of trolls, even well-intentioned, can poison an entire discussion space.

6. write tests against public API instead of unit tests for internal implementation modules. then you can fearlessly rewrite the implementation using tests for verification and guidance. you can even keep both implementation versions at the same time and run tests against both.

7. long-running branches suck. merge into main and keep it behind a feature flag. invest time into a good feature flag mechanism that cuts out dead code and/or lets you deploy feature flag changes for % of uses in prod experiments.

8. if a change shows up red in metrics, maybe you made it slower. but maybe you introduced a bug which broke some interaction for %. or maybe you changed how metric is logged. or maybe you fixed a bug and metric was overreported before. metrics are hard.

9. many people want to score a PR in a popular project so that it appears on their gh. somebody will convert if-else to early return. next day somebody else will send a PR to convert it back. it doesn’ t hurt to merge until you miss a mistake in the middle of a “stylistic” change.

10. if you fix something, add a test that fails. if you don’ t add one, you’ re tacitly agreeing that someone else (including you) will break it again.

11. your writing in project blog/docs will be copy pasted verbatim, rephrased, adapted, and then posted on content farms, other people’ s personal blogs and slides. in a way that’ s actually kind of cool because this means your writing worked

12. some projects have wide surface area and decisions can be decentralized. other projects have narrow surface area and decisions need to be centralized. both styles are ok, just don’ t get confused why techniques that work for one type of project may be disastrous for another

13. keeping github clean is nice but for each individual person on the team there’ s almost always something higher-value to do. that’ s ok and you don’ t need to feel bad about it. it’ s much more important that your software works well. if it becomes dire you’ ll know

14. nevertheless it’ s invaluable to do occasional github issue sweep and read/triage every thread. you’ ll get a window into the pain points and confusion that is hard to build up otherwise. that alone is worth the time and effort

15. with some PRs it feels like the person really knows what they’ re doing. reach out to them. they might be your future teammate

16. if you don’ t explain the story about what you’ re doing and why, someone else will do it for you (and you might not like it)

17. be careful with what other projects you recommend and endorse, they might be the very thing that holds the entire ecosystem back in a few years

18. if there is harsh feedback, it is better to hear it from a kind person. unfortunately a kind person will often keep it to themselves, while a troll would shout about it. making it tempting to dismiss. validate harsh feedback with kind people and keep the door open for it

19. don’ t DM with a troll. it’ s a waste of time. don’ t get dragged into twitter feuds. not enough space for nuance. redirect discussion to a technical long-form space. keep DMs for friends and people you trust

20. don’ t throw inside DEV-only code. this makes some branches reachable in production but unreachable in DEV. so you might actually have *worse* problems than crashing in prod, like a privacy compromise, and none of the developers experienced it on their machines due

to throw

21. when you write a tutorial, sit down with someone from your target audience and have them walk through it. they’ ll stumble in five different places than you expected. you’ ll tear your hair out and make the tutorial 5x better

22. some for the nicest, kindest and most amazing people you’ ll meet and hang out with will be the people working on the competing projects

23. when you deprecate an api, don’ t put a tombstone in your slides. this is tempting and might be funny to you as an author but categorically a wrong move

24. big rewrites can work if you know what you’ re doing. make sure you know what you’ re doing though

25. lock down introspection APIs and keep first-class tooling that uses them colocated. this ensures you’ re not locked out of major internal changes by a long tail of tooling that assumes something about internal data structures and has become unmaintained or is incompatible

26. a crunch with a few weeks of overworking can be emotionally fulfilling when the team is in the flow state. a perfect work/life balance while team morale is low and dynamics are thorny can feel like a piece of your soul is flushed down the sink every day

27. innovation is not linear. sometimes you need to make it worse to make it better.

28. expanding on that, when you replace a system, you usually need to go incrementally. otherwise it’ s too much risk for stakeholders to go all-in. but that means you won’ t reap the benefits of *replacing* it until the end. you’ re *adding* so likely regressing. set expectations.

29. let users try a new thing one piece at a time. even if you want them to use it everywhere, let that be *their* idea

30. if you can’ t prove out a migration strategy with a small team on a big codebase, you don’ t have a migration strategy. start again

31. the most important principle in API design is that it composes well. that means that if two people create two things, they need to still work well together combined. it also means you can delete one of them later without breaking the other. or copy and paste it to isolate it.

32. you will see the most bizarre feature requests. don’ t dismiss them outright but ask why they’ re trying to do this. a lot of use cases are actually legit and food for thought. eventually you run out of new ones — you’ ve covered the entire problem space. consider solutions

33. if something doesn’ t make sense, no amount of documentation or explanations will make it make sense.

34. people will copy and paste every code snippet you write. even the ones marked with “don’ t copy paste, this is the bad example, the good one is below”

35. your personal social media account shouldn’ t be a primary source of truth for news in a mature project. people will get FOMO plus it’ s legitimately hard to track. you can repost but keep news in official sources (like GH)

36. even if someone is a great developer or a good friend, it still doesn’ t mean they’ ll necessarily succeed on your team. this can be gut-wrenching for all parties when it doesn’ t work out.

37. if you accept and merge every individually reasonable clarification that someone sends as a PR to your docs, they will become a sorry unreadable mess

38. people *love* to hit the Accept button on someone’ s PR even if they never contributed to the project and never left a single comment on it. it’ s just a thing people do

39. occasionally, it’ s good to take the minified code, run it though Prettier, and quickly read through it. you might spot things that shouldn’ t be there, things that could be minified better, or get ideas on how to reduce the size

40. if you’ re trying to introduce a new workflow (eg different task tracking tool) to the team, it’ s on you to get people to feel good about it and try it out (“i know you prefer keeping tasks in a doc, let me sync that doc for you for a bit and we’ ll see how it goes”)

41. taking ownership over something unowned (but that everybody cares about) is one of the most valuable things you can do on a team. try not to drop it though (i’ m guilty of that)

42. when the code serves a particular purpose that’ s too high-level to be seen from the code, write it down as a comment. long-ass ten line comments are good if they add context to the code (rather than describe what it’ s doing)

43. re-running a test you’ re actively iterating on after a file change shouldn’ t take more than a second

44. write a script to do releases for you. get the CI to create automatic prereleases from main so you can always test the latest main in codesandbox (and other libs can run tests against nightlies). a stable release should patch+promote a CI version, not build locally

45. your teammates can literally move mountains

46. when you work on your messaging, run it by people. use it many times in narrower circles and widen slowly. close the gaps using feedback so that by the time you have final messaging, it’ s the parts that resonate, are easy to understand, are easy to quote, and (!) make sense

47. if your technical vision is sound in theory, and you chip away at the practical obstacles with perseverance, organisational support, pragmatic compromises, and optimism, eventually the theory wins

48. if you have one scary bug after another, write regression tests but also use this as a signal. something in the model might be broken. complex code isn’ t bad, but code built around a flawed model is very bad, complex or not

49. all people are irreplaceable. every iteration of a team is a new team in some sense. in some ways this is poignant, in others this is exciting

50. naming is one of the most important decisions you will be doing. it’ s not just bikeshedding. give it appropriate time and effort. you will mess it up anyway though.

51. you keep hearing complaints about an API. the fix is simple, like flipping a switch. fine!! you change it. suddenly, the other half, previously quiet, comes frustrated. oops. turns out both options were equally disliked. congrats, you’ ve just churned your users for no reason

52. semver is “just” a social contract. yes it’ s important to communicate intent. breaking changes shouldn’ t go into minors/patches. but even fixing a bug is a breaking change for someone relying on the buggy behavior. semver is first and foremost a human communication tool

53. you are not your project. it also seems unwise to pretend that it isn’ t in some way a significant part of your life. not “work hustle 4 da boss” capitalism thing but more like “regardless of power structures i’ ve put a lot into this and i have reasons to care” . that’ s valid

54. most things don’ t stick around long enough. you’ ll now read most of the “announcing X, a new way to Y” with a 2-3 year flash-forward “X is legacy” in mind. be gentle or people will read this as being cynical. also some things *will* stick around, either in a good or a bad way

55. edge cases are your bread and butter. they’ re more common than you think (with enough usage anything happens) and they are a first-class part of your design. you should have satisfactory (if not satisfying) answers to how the system behaves even (especially) when it’ s wonky

56. it’ s better when issues reproduce consistently. but if some bug caused by API misuse in

product code is non-deterministic and happens less often than random typical existing bugs in the product code, it’ s below the threshold of rethinking the design

57. thought experiments are your main tool as an API designer. “imagine a tabbed view…” “imagine a combobox…” “suppose we have a feed and a profile…” is how many breakthroughs begin. you start with a situation, think through ideal behavior — from user and system point of view

58. that clever optimization? probably unnecessary and will cost you a few weeks of bug hunting. oh, and good luck ripping it out

59. introduce some slight inefficiency in a hot path, and clicking every button becomes 10% slower. do that several more times and you’ re 100% slower

60. constraints and edge cases drive your design. constraints exclude 99% of possible designs. edge cases accentuate the remaining choices and help you pick which one sucks the least (or least surprisingly). then if the common case makes sense you’ re golden

62. it’ s unwise to put all effort into optimizing a fixed cost of library code if the cost of user code is unbounded. ignore benchmarks that focus on how fast the 5% of a realistic app runs, and focus on making 95% you don’ t own faster. this doesn’ t justify being sloppy though

63. if you go on a panel with a competing library maintainer and talk about an upcoming feature in your release a few months away, there’ s a high chance they will ship a similar feature next week. that’ s only fair

64. you will build for people who are excited to use your project. everyone’ s jazzed. then you’ ll notice a flood of people who hate it. this is because the people who were excited to use it are now forcing other people to use it (oh no) also now there’ s a job market, congrats

65. your goal isn’ t to make every app use your library. it’ s to make other libraries copy the ideas you value the most. then your work won’ t be lost because it’ s upstreamed to the global programming consciousness. the failure case is good ideas dying with your code

66. look out for inspiration outside of your direct competitors. your direct competitors are looking at you too so at some point there’ s a deadlock for fresh ideas

67. innovation looks like synthesis of existing ideas with a novel twist. it doesn’ t happen often and it’ s easy to miss if you’ re not paying attention

68. people like to optimize something that they know how to optimize and where there’ s obvious progress. there will never be a shortage of people excited about replacing a 5 KB library with a 2 KB library

69. designing is not so much creating as it is uncovering what already should be there. like math or archeology. it’ s exploiting the properties of the system and making them shine

70. “how it scales” matters for technology adoption by small shops too. this is because if it doesn’ t scale to big shops, there’ ll be a competitor that scales *and also* works well enough for small shops. the solution that’ s good enough for most people will wipe out yours

71. invoking “simple vs easy” as an argument in an API discussion works great if you have the design sensibilities of rich hickey. otherwise it’ s just a sparkling HN comment

72. people have opinions on everything. even if you’ re unable to handle the volume or respond individually, it helps to provide a void to shout into so that people feel heard, the discussion is consolidate, and doesn’ t spill out into random platforms. then you can revisit it

73. much of Programming Discourse is shaped by people who aren’ t shipping production code and have lost touch with the reality. get used to it and carry on. oh by the way… you are one of those people now

74. people arguing about applying abstract code patterns on Twitter means you’ re failing. the wise API designer would empty their users’ minds, fill their bellies, weaken their wishes, strengthen their bones…

75. “no” is the correct answer to most feature requests. no matter who they’ re coming from. try your best to help people though and bring the pain points back to the table. this is developer advocacy

76. “try harder” sounds too harsh as a motto and can be misused. I’ d say, “peek around the corner” . it’ s okay to stop when you’ re stuck. but make sure you didn’ t just give up on the last few steps to the solution because you ran out of hope

77. there are few forms of communication more powerful than a tech talk. it’ s like carving out a meme out of stone. use this medium thoughtfully.

78. experimental/research work and content should be branded separately from the stuff people need to know and can use today.

79. eager vs lazy: both suck. eager sucks because you do more blocking work upfront that may not be used. lazy sucks because you start work too late. “eager non-blocking” is best. start optional work asap but you can drop it midway or downprioritize it if it’ s not needed anymore

80. it’ s often more valuable to have someone who “gets it” in partner teams than having

that person on your own team

81. don’ t compromise on core principles and properties of the system for the sake of convenience. convenience helps narrowly, but core principles will keep on giving for years in ways you didn’ t expect. it’ s nice when you can have both though

82. “i don’ t know” and “i don’ t understand this part” and “can you say this again” are phrases you should expect to say (and hear) in productive meetings

83. you need to create space for design discussions that isn’ t crammed into a regular sync meetings. most enlightening discussions I’ ve seen are free-flowing, didn’ t have a predetermined agenda, went into rabbit holes, and needed a few hours to get to a conclusion. write it down!

84. failing is ok but you want some lesson carried away from it. so that the next person who tries has a chance to mitigate the same issue proactively. failing comes with a cost: it’ s harder to justify doing same project again. better to wait out than to try at a wrong time.

85. the best features are vertically integrated. that means that they don’ t just work at a particular level of the stack but pierce through the whole stack and give you some leverage at every level, low and high. they’ re also integrated together: they all compose as you’ d expect.

86. don’ t cut a stable release or deploy the website on Friday unless you’ re doing this intentionally. for example, if there’ s a big codemod and you need to land it while nobody else is pushing conflicts

87. global configuration like React.setOptions() is a bad idea. this is because components on npm written assuming one set of options wouldn’ t work with the other set of options. so global configuration breaks composition.

88. people can handle bad news but people hate surprises

89. the way each person thinks makes a lasting contribution to the team. “What would X do” is not only a useful question that X can answer but also a way to nudge our own thinking for years going forward

90. an autocomplete has one chance to reorder per keystroke. if you got it wrong and you have a better ordering a bit later you must “swallow the sadness” (as per the original author of this wisdom) but never change already displayed items

91. if you have two similar APIs and you’ re worried people will overuse A instead of B (which is actually better), give A a clunky name and make B nice and short

92. ?

93. sometimes your thing sucks but you don’ t know the solution and don’ t have anything that helps move it forward. you have to learn to get comfortable with something badly sucking for years. just make sure a workaround exists

94. re-read every release post imagining that you’ re drunk and your attention span is zero. if the message still lands you’ re good

95. carrying messaging directly from a small team to the community doesn’ t scale if the messaging is complex and nuanced. instead it is helpful to have a buffer of trusted folks who can learn the ins and outs, become experts, and participate in those panels

96. the best task management system for a release is long-ass quip doc (or gh issue) with a freeform checklist. we call these “umbrellas”

97. never assume you know others’ emotional states. if you don’ t ask, you might not hear some critical information until it’ s too late

98. it’ s all about the people. if relationships are frail and people don’ t trust each other, projects fail. technical wiring directly reflects which teams trust each other. poor communication produces broken systems. screw the org chart and go talk to the ICs directly

99. newer people on the team won’ t know the norms if you don’ t tell them. if reviews are slow in general they might think they specifically are being ghosted. be explicit about everything and always follow-up on how they feel and what’ s concerning them

100. it is incredibly rewarding to learn directly from people whose expertise and experience vastly eclipses yours. i am thankful to @sebmarkbage, @sophiebits, @tomocchino and many others who I’ ve learned all of these things from!

1. 每隔几年你的观众就会改变。新用户不喜欢过去的问题，因为他们从未见过。老用户精疲力竭或失去兴奋。新用户有不同的参考框架，以不同的方式学习，你可能是他们的第一个编程环境。相应地计划。

2. 当你解决一个问题时，你最好真正了解你正在解决的问题。退后几步，将新知识重新整合到设计中。它应该改变设计吗？这就像与九头蛇战斗：在错误的地方解决问题会产生 10 个新问题。

3. 大多数 PR 创造的工作多于解决的问题。即使项目的内部结构有据可查（这对我们来说很简短），也很少有重大贡献。这不是因为编写代码的困难，而是因为需要几个月的上下文来做出决定。

4. 对于库，确保你的 PR 描述很好（它们变成了压扁的提交消息）。对于应用程序，包括之

前/之后的产品视频以及您在 PR 描述符中测试的 url。八年后，有人（我）会在诊断错误或计划更改时感谢您。

5. 几个喷子，即使是善意的，也会毒害整个讨论空间。

6. 编写针对公共 API 的测试，而不是针对内部实现模块的单元测试。然后您可以使用测试进行验证和指导，无所畏惧地重写实现。您甚至可以同时保留两个实现版本并针对两者运行测试。

7.长期运行的分支很烂。合并到 main 并将其保留在功能标志后面。花时间在一个好的功能标志机制上，它可以减少死代码和/或让您在生产实验中为百分比的使用部署功能标志更改。

8. 如果更改在指标中显示为红色，则可能是您让它变慢了。但也许你引入了一个错误，它破坏了 % 的一些交互。或者您可能更改了指标的记录方式。或者也许您修复了一个错误并且之前的指标被高估了。指标很难。

9. 很多人想在一个热门项目中获得一个 PR 以便它出现在他们的 gh 上。有人会将 if-else 转换为提前返回。第二天，其他人将发送 PR 将其转换回来。合并不会有什么坏处，除非您在 “风格” 更改过程中错过了一个错误。

10.如果你修复了什么，添加一个失败的测试。如果你不添加一个，你就默认了其他人（包括你）会再次破坏它。

11. 您在项目博客/文档中的写作将被逐字复制粘贴、改写、改编，然后发布到内容农场、其他人的个人博客和幻灯片上。以一种实际上很酷的方式，因为这意味着你的写作有效

12. 一些项目涉及面广，决策可以分散。其他项目的表面积很小，需要集中决策。两种风格都可以，只是不要混淆为什么适用于一种类型的项目的技术对另一种类型的项目可能是灾难性的

13. 保持 github 干净是好的，但对于团队中的每个人来说，几乎总是有更高价值的事情要做。没关系，你不需要为此感到难过。更重要的是您的软件运行良好。如果它变得可怕，你就会知道

14. 尽管如此，偶尔进行 github 问题扫描和读取/分类每个线程是非常宝贵的。你会看到痛点和困惑，否则很难建立起来。仅此一项就值得花时间和精力

15. 使用一些 PR 感觉这个人真的知道他们在做什么。联系他们。他们可能是你未来的队友

16.如果你不解释你在做什么以及为什么做的故事，别人会为你做（你可能不喜欢）

17.小心你推荐和认可的其他项目，它们可能会在几年内阻碍整个生态系统的发展

18.如果有苛刻的反馈，最好从好心人那里听到。不幸的是，一个好心的人往往会把它藏在

心里，而一个巨魔会大喊大叫。让人很想解雇。与善良的人一起验证苛刻的反馈，并为之敞开大门

19. 不要用巨魔 DM。这是浪费时间。不要被拖入推特的不和中。细微差别的空间不足。将讨论重定向到技术长篇空间。为朋友和您信任的人保留 DM

20. 不要把 DEV-only 代码扔进去。这使得某些分支在生产中可以访问，但在 DEV 中无法访问。所以你实际上可能会遇到比在 prod 中崩溃更糟糕的问题，比如隐私泄露，并且没有一个开发人员在他们的机器上遇到它由于抛出

21. 编写教程时，请与目标受众中的某个人坐下来，让他们逐步完成。他们会在五个与您预期不同的地方绊倒。你会撕掉你的头发，让教程变得更好 5 倍

22. 一些你会遇到和一起出去玩的最好、最善良、最了不起的人将是那些在竞争项目中工作的人

23. 当你弃用一个 api 时，不要在你的幻灯片中放置墓碑。这很诱人，作为作者可能对你来说很有趣，但绝对是错误的举动

24. 如果你知道自己在做什么，大的重写就可以奏效。确保你知道你在做什么

25. 锁定内省 API 并保持使用它们的一流工具共存。这可确保您不会被长尾工具锁定在重大内部更改之外，这些工具假设有关内部数据结构的某些内容并且已变得无法维护或不兼容

26.当团队处于心流状态时，几周过度工作的紧缩可能会在情感上得到满足。完美的工作/生活平衡，同时团队士气低落和动态是棘手的感觉就像你的灵魂每天都被冲入水槽

27. 创新不是线性的。有时你需要让它变得更糟才能让它变得更好。

28. 进一步说，当你更换一个系统时，你通常需要逐步进行。否则，利益相关者全押风险太大。但这意味着直到最后你才能获得*替换*它的好处。你正在*添加*很可能会倒退。设定期望。

29.让用户一次一件尝试新事物。即使你想让他们在任何地方使用它，让它成为*他们*的想法

30.如果你不能在大代码库上用一个小团队证明一个迁移策略，你就没有迁移策略。重新开始

31、API 设计最重要的原则就是组合好。这意味着如果两个人创造了两个东西，他们仍然需要很好地结合在一起。这也意味着您可以稍后删除其中一个而不会破坏另一个。或复制并粘贴它以隔离它。

32. 你会看到最离奇的功能请求。不要直接解雇他们，而是要问他们为什么要这样做。许多用例实际上是合法的，值得深思。最终你会用完新的——你已经涵盖了整个问题空间。考虑

解决方案

33. 如果某件事没有意义，再多的文件或解释也没有意义。

34. 人们会复制和粘贴你写的每一个代码片段。甚至那些标有 “不要复制粘贴，这是不好的例子，好的在下面”

35. 在成熟的项目中，您的个人社交媒体帐户不应成为新闻的主要真实来源。人们会得到 FOMO，而且很难追踪。您可以重新发布但将新闻保留在官方来源（如 GH）中

36. 即使某人是一位出色的开发人员或好朋友，也并不意味着他们一定会在您的团队中取得成功。当它不起作用时，这对所有各方来说都是令人痛苦的。

37. 如果你接受并合并某人作为 PR 发送给你的文档的每一个单独合理的澄清，它们将成为令人遗憾的无法阅读的混乱

38. 人们*喜欢*点击某人公关上的 “接受” 按钮，即使他们从未为该项目做出贡献并且从未对其发表过任何评论。这只是人们做的事情

39. 偶尔，把缩小后的代码，通过 Prettier 运行，然后快速通读。你可能会发现不应该存在的东西，可以更好地缩小的东西，或者关于如何减小尺寸的想法

40.如果你想向团队引入一个新的工作流程（例如不同的任务跟踪工具），你有责任让人们对它感觉良好并尝试它（ “我知道你更喜欢将任务保存在文档中，让我为你同步一下那个文档，我们会看看它是怎么回事” ）

41. 对无人拥有的（但每个人都关心的）事物拥有所有权是您在团队中可以做的最有价值的事情之一。尽量不要放弃它（我对此感到内疚）

42.当代码服务于某个特定目的而从代码中看不出来时，把它写下来作为注释。如果将上下文添加到代码中（而不是描述它在做什么），那么长的十行注释是好的

43.在文件更改后重新运行您正在积极迭代的测试不应超过一秒钟

44. 写一个脚本为你做发布。让 CI 从 main 创建自动预发布，这样你就可以始终在 codeandbox 中测试最新的 main（其他库可以针对 nightlies 运行测试）。一个稳定的版本应该修补+提升 CI 版本，而不是在本地构建

45.你的队友真的可以移山

46.当你处理你的消息时，由人来运行。在较窄的圆圈中多次使用它并慢慢扩大。使用反馈缩小差距，以便在您收到最终消息时，会引起共鸣、易于理解、易于引用且 (!) 有意义的部分

47.如果你的技术愿景在理论上是合理的，并且你用毅力、组织支持、务实的妥协和乐观来消除实际障碍，最终理论会获胜

48. 如果你有一个又一个可怕的错误，编写回归测试，但也将其用作信号。模型中的某些内容可能已损坏。复杂的代码还不错，但是围绕有缺陷的模型构建的代码非常糟糕，无论复杂与否

49.所有的人都是不可替代的。一个团队的每一次迭代，在某种意义上都是一个新的团队。在某些方面这是令人心酸的，在其他方面这是令人兴奋的

50. 命名是您将要做的最重要的决定之一。这不仅仅是自行车棚。给它适当的时间和精力。无论如何你都会把它搞砸。

51. 你不断听到关于 API 的抱怨。修复很简单，就像拨动开关一样。美好的！！你改变它。突然，原本安静的另一半变得沮丧。哎呀。事实证明，这两种选择都同样不受欢迎。恭喜，你刚刚无缘无故地流失了你的用户

52. semver “只是”一种社会契约。是的，传达意图很重要。重大更改不应进入未成年人/补丁。但即使修复错误对于依赖错误行为的人来说也是一个重大改变。 semver 首先是一种人类交流工具

53.你不是你的项目。假装它在某种程度上不是你生活的重要组成部分似乎也是不明智的。不是“工作喧嚣 4 大老板”资本主义的事情，而是更像是“无论权力结构如何，我都投入了很多，我有理由关心”。这是有效的

54. 大多数事情都坚持的时间不够长。现在，您将阅读“宣布 X，通往 Y 的新方式”的大部分内容，并记住 2-3 年的“X 是遗产”。要温柔，否则人们会认为这是愤世嫉俗。还有一些事情*会*留下来，无论是好的还是坏的

55. 边缘案例是你的面包和黄油。它们比您想象的更常见（使用足够多，任何事情都会发生），并且它们是您设计的一流部分。你应该对系统的行为有满意（如果不满意）的答案，即使（特别是）当它不稳定时

56. 问题持续重现会更好。但是，如果产品代码中 API 误用导致的某些错误是不确定的，并且比产品代码中随机存在的典型错误发生的频率更低，则低于重新思考设计的门槛

57. 思想实验是你作为 API 设计者的主要工具。 “想象一个选项卡式视图……” “想象一个组合框……” “假设我们有一个提要和一个配置文件……”是多少突破开始的地方。你从一种情况开始，思考理想的行为——从用户和系统的角度来看

58.那巧妙的优化？可能没有必要，并且会花费您数周的 bug 搜寻时间。哦，祝你好运把它撕掉

59. 在热路径中引入一些轻微的低效率，点击每个按钮会变慢 10%。多做几次，你就慢了 100%

60. 约束和边缘情况驱动你的设计。约束排除了 99% 的可能设计。边缘情况突出了剩余的选择，并帮助您选择哪个最差（或最不令人惊讶）。那么如果常见的情况是有道理的，你就是金子

62. 如果用户代码的成本是无限的，那么将所有精力都放在优化库代码的固定成本上是不明智的。忽略关注真实应用程序中 5% 的运行速度的基准，而专注于使您不拥有的 95% 更快。但这并不能成为马虎的理由

63. 如果您与竞争对手的库维护者一起参加一个小组讨论，并讨论几个月后您发布的即将推出的功能，他们很有可能会在下周发布类似的功能。这才是公平的

64. 你将为那些对使用你的项目感到兴奋的人而构建。每个人都很兴奋。然后你会注意到一大群讨厌它的人。这是因为那些对使用它感到兴奋的人现在正在强迫其他人使用它（哦不）现在还有一个就业市场，恭喜

65. 你的目标不是让每个应用都使用你的库。是为了让其他图书馆复制你最看重的想法。那么您的工作就不会丢失，因为它已上升到全球编程意识。失败案例是好主意随着你的代码而消亡

66. 在你的直接竞争对手之外寻找灵感。你的直接竞争对手也在看着你，所以在某些时候，新鲜想法会陷入僵局

67. 创新看起来像是将现有想法与新颖的转折相结合。它不会经常发生，如果你不注意，很容易错过

68. 人们喜欢优化他们知道如何优化以及有明显进步的地方。永远不会缺少对用 2 KB 库替换 5 KB 库感到兴奋的人

69. 设计与其说是创造，不如说是揭示已经存在的东西。比如数学或考古。它正在利用系统的特性并使它们发光

70. “如何扩展” 对于小商店采用技术也很重要。这是因为如果它不能扩展到大商店，就会有一个竞争者可以扩展*并且*对小商店也足够好。对大多数人来说足够好的解决方案会消灭你的

71. 如果你有丰富的设计敏感性，那么在 API 讨论中将 “简单 vs 容易” 作为一个论点会很有效。否则它只是一个闪闪发光的 HN 评论

72.人对一切都有意见。即使您无法处理音量或单独响应，它也有助于提供一个可以大声呼喊的空间，以便人们感到被倾听，讨论是巩固的，并且不会蔓延到随机的平台上。然后你可以重新访问它

73. 大部分编程话语都是由那些没有发布生产代码并且与现实脱节的人塑造的。习惯并继续。

哦，顺便说一句……你现在是那些人中的一员

74. 人们争论在 Twitter 上应用抽象代码模式意味着你失败了。聪明的 API 设计者会清空用户的思想，填饱肚子，弱化他们的愿望，强健他们的骨骼……

75. “否” 是对大多数功能请求的正确答案。不管他们来自谁。尽最大努力帮助人们，并将痛点带回桌面。这是开发者的倡导

76. “努力” 作为座右铭听起来过于苛刻，可能会被滥用。我会说， “在拐角处偷看” 。卡住的时候停下来也没关系。但请确保您不会因为没有希望而放弃解决方案的最后几步

77. 没有比技术谈话更有效的交流形式了。这就像用石头雕刻出一个模因。慎重使用这种媒介。

78. 实验/研究工作和内容应该与人们今天需要知道和可以使用的东西分开标记。

79. 渴望 vs 懒惰 :都很糟糕。急切很糟糕，因为您预先做了更多可能不会使用的阻塞工作。懒惰很糟糕，因为你开始工作太晚了。 “急切的非阻塞” 是最好的。尽快开始可选的工作，但如果不再需要，你可以中途放弃它或降低它的优先级

80. 在合作团队中拥有一个 “明白” 的人通常比在你自己的团队中拥有这个人更有价值

81.不要为了方便而妥协系统的核心原则和属性。便利的帮助很小，但核心原则会以您意想不到的方式持续提供多年。不过，当你可以同时拥有两者时，这很好

82. “我不知道” 和 “我不明白这部分” 和 “你能再说一遍吗” 是你应该在富有成效的会议中说（和听到）的短语

83. 你需要为设计讨论创造空间，而不是挤在定期的同步会议中。我见过的最有启发性的讨论都是自由流动的，没有预定的议程，陷入了困境，需要几个小时才能得出结论。写下来！

84. 失败是可以的，但你想从中吸取教训。以便下一个尝试的人有机会主动缓解相同的问题。失败是有代价的：再次做同一个项目更难。与其在错误的时间尝试，不如等待。

85. 最好的功能是垂直整合。这意味着它们不仅在堆栈的特定级别起作用，而且贯穿整个堆栈，并在每个级别（低和高）上为您提供一些影响力。它们也集成在一起：它们都按照您的预期进行组合。

86. 除非你是故意的，否则周五不要发布稳定版或部署网站。例如，如果有一个很大的 codemod 并且您需要在没有其他人推动冲突的情况下登陆它

87. 像 React.setOptions() 这样的全局配置是个坏主意。这是因为 npm 上的组件假设一组选项不适用于另一组选项。所以全局配置打破了组合。

88. 人们可以处理坏消息，但人们讨厌惊喜

89. 每个人的思维方式都会对团队产生持久的贡献。 “X 会做什么” 不仅是 X 可以回答的有用问题，而且是推动我们未来多年思考的一种方式

90. 自动完成每次击键有一次重新排序的机会。如果你弄错了，稍后你有更好的订购，你必须 “吞下悲伤” （按照这个智慧的原作者）但永远不要改变已经显示的项目

91. 如果你有两个相似的 API 并且你担心人们会过度使用 A 而不是 B（实际上更好），给 A 一个笨重的名字，让 B 漂亮而简短

92. ?

93. 有时你的事情很糟糕，但你不知道解决方案，也没有任何帮助推动它前进的东西。你必须学会适应多年来糟糕透顶的事情。只要确保存在解决方法

94. 重新阅读每个发布的帖子，想象你喝醉了，你的注意力为零。如果消息仍然登陆你很好

95. 如果消息传递复杂且细微，则直接从小团队向社区传递消息不会扩展。相反，拥有一批可信赖的人作为缓冲区是有帮助的，他们可以了解来龙去脉，成为专家并参与这些小组

96. 发布的最佳任务管理系统是带有自由格式清单的长篇俏皮的 quip doc（或 gh 问题）。我们称这些为 “伞”

97.永远不要假设你知道别人的情绪状态。如果你不问，你可能不会听到一些关键信息，直到为时已晚

98.一切以人为本。如果关系脆弱，人们彼此不信任，项目就会失败。技术布线直接反映了哪些团队相互信任。沟通不畅会导致系统崩溃。搞砸组织结构图并直接与 IC 交谈

99.如果你不告诉他们，团队中的新人不会知道规范。如果评论总体上很慢，他们可能会认为他们特别被鬼魅。对每件事都直截了当，并始终跟进他们的感受以及与他们有关的事情

100. 直接向那些专业知识和经验远远超过你的人学习是非常有益的。我很感谢 @sebmarkbage，@sophiebits，@tomocchino 和许多其他我从他们那里学到了所有这些东西的人！