前几天看了《Code Review 程序员的寄望与哀伤》，想到我们团队开展 Code Review 也有2年了，结果还算比较满意，有些经验应该可以和大家一起分享、探讨。

我们为什么要推行Code Review呢？我们当时面临着代码混乱、Bug频出的状况。

当时我觉得要有所改变，希望能提高产品的代码质量，改善开发团队面临的困境。并且我个人在开发上有很多经验，也希望这些知识能够在团队内传播。

各种考虑后，我们最后认为推行Code Review能改善或解决我们面临的很多问题。

这篇文章的目的不是告诉大家怎么在一个团队内推行Code Review，首先因为我个人仅在一家公司内推行过，并没有很多经验。

其次每家公司、每个团队的情况都不太一样，应该根据公司或团队的实际情况选择恰当的方案，并根据成员的反馈来及时调整，推动Code Review的实施。

所以，本文是介绍我们公司是如何实施Code Review的，我们是如何解决我们遇到的问题的，希望我们的经验能给大家带来些帮助。

行文仓促，如有遗漏或错误，欢迎指正。

一、流程和规则

经过简单的对比、试用，我们最后采用了Git Flow+Pull Request（PR）模式来做Code Review。（PR模式详情可参见 Git工作流指南：Pull Request工作流）

Pull Request(PR)简单的说就是你没有权限往一个特定的仓库或分支提交代码，你请求有权限的人把你提交的代码从你的仓库或分支合并到指定的仓库或分支。

由于PR需要有权限的人确认，所以非常适合在这个过程中做Code Review，是否接受或者拒绝就取决于Code Review的结果。

在支持PR模式的软件里，每一个PR都有一个新增代码的对比（diff）界面。

代码审核者可以在线浏览请求合并的新增代码，并针对有疑问的代码行添加评论，通过这种方式来实现Code Review。

评论可以被所有有权限查看仓库的人看到，每个人都可以回复任何人的评论，有点像论坛里某个帖子的讨论。

这种模式是事后审核，也就是代码已经提交到了中心仓库，Review过程中频繁的改动会造成历史签入记录的混乱。

当然Git可以采用更改历史记录来解决这个问题，由于容易误操作，我们一般只在基础类库这类要求比较严格的项目上实施。

我们所了解到的支持PR模式的软件都采用Git作为源代码版本控制工具，所以我们的源代码版本控制工具也迁移到了Git。

由于Git太灵活了，因此诞生了很多的Git流程，用来规范Git的使用。

常见的有集中式工作流、功能分支工作流、Gitflow工作流、Forking工作流、Github工作流。

我们对Git Flow做了些调整，调整后的流程被命名为Baza Flow，定义见后文。

根据Baza Flow，我们大部分仓库只定义了2个主干分支，master和develop。(例外，我们有一个仓库有3个开发小组同时进行开发，定义了4个主干分支，目前还比较顺畅，再多估计主干分支之间的合并就比较繁琐了。)

master对应生产环境代码，所有面向生产环境的发布来源都是master分支的代码。

develop则对应本地测试环境的代码。

绝大多数情况下，QA（测试）只测试develop分支和master分支的代码。

由于开发人员都在一个团队内，所以我们没有采用基于仓库的PR，采用的是基于分支的PR。

我们对主干分支的操作权限做了限制，只有特定的人才能操作，develop分支是项目开发Leader和架构师，master分支是QA。

有权限往主干分支合并的成员会按照约定的规则来执行合并，不会合并没有完成审核的PR。

上面这点其实蛮重要的，所以我们会对有权限合并的人有特别的约定，在什么情况下才能合并代码。（见后文PR的说明）

PR的发起人要主动的推动PR的审核，Leader也会密切关注PR审核的进度，在需要的时候及时介入。

我们配置了CI服务器（什么是CI）只编译特定的分支，通常是develop和master分支。

所有的代码合并到了主干分支之后，都会自动触发编译和本地测试环境的发布，QA无需依赖开发人员编译的代码来测试，也无需自己手工操作这些，保证了开发人员和测试人员的相互独立。

我们本地测试环境的发布包含了数据库和站点的发布，全自动的，发布完成以后就是一个可用的产品，有时间这部分也可以分享一下。

我们还使用了Scrum里面一个很重要的概念：完成定义。

就是我们规定了我们一个任务的完成被定义为：代码编写完成，经过自测，提交的PR经过审核并且合并到主干分支。

也就是说，所有的代码被合并到了主干分支之后任务才算是完成，而被合并到主干分支必须要经过Code Review，这是强制的。

由于我们的托管软件对于Pull Request的限制，我们对Git Flow做了改动，改动的地方有：

1、每一个大功能我们会创建一个单独的feature分支，项目开发人员基于这个单独的feature分支创建自己的任务分支。

比如，对于CS 2项目来说，启动的时候分支的创建是：master -> develop -> feature/v2。

开发人员应该基于这个大特性分支feature/v2来创建自己的任务分支，比如创建XXXX，可以用一个单独的分支feature/v2-xxxx。

完成这个任务以后，立即向上游分支（feature/v2）提交pull request。然后从feature/v2-xxxx 创建自己的下一个任务分支，比如YYYY编辑 feature/v2-yyyy。

请注意，合并到上游分支的功能必须相对独立而且是可用的，分支任务工作量0.5-1个工作日，不宜超过2个工作日，超过2个工作日不向上游合并，需要向团队解释。

代码经过Review以后，可能会进行必要的修改，修改在原分支修改，修改完毕代码合并进上游分支，原分支会定期删除。

项目组成员在收到合并成功的通知后，请自行从上游大特性分支向下合并到自己当前的开发分支。

提交pull request后创建新任务分支的时候务必知会一下相关配合同事（比如前端的同事），让他们在新的分支上继续开发。

2、对于小功能，预计在0.5-1个（不超过2个）工作日工作量的开发任务，直接基于develop分支创建特性分支即可。

3、在各个分支遇到的bug，请基于该分支创建一个Bug分支。

如果在缺陷跟踪管理系统上有对应的项，命名请使用缺陷跟踪管理系统的ID，比如BAZABUG-1354 比如这个Bug的分支命名就是bugfix/BAZABUG-1354。

如果在缺陷跟踪管理系统上没有对应的项，命名请简短的说明修改内容，比如“JX 9df2b01 引用bootstrap css虚拟路径重写，避免出现字体无法找到的问题”，分支命名可以是bugfix/miss-font。

完成修改以后提交并推送到中心仓库然后立即向上游分支提交pull request。

4、发起pull request以后，请将pull request的链接在IM上发给代码审核者，以此通知对方及时进行审核。

二、执行

我们在团队内部提倡质量优先，开发团队不能为了进度牺牲质量，并在团队内部达成了共识。

所以，无论进度有多么紧迫，Code Review的过程都一定会做。

所有的问题一定会被提出，只是会根据进度的紧迫程度，以及问题的大小，改动成本，决定问题是现在解决，还是加一个TODO，并记录在缺陷跟踪管理系统内，以防日后遗忘。

多数情况下，我们都会要求立即解决，哪怕因此造成了发布的推迟。

我们深知，其实多数情况下，现在不解决，日后不知道猴年马月才能解决。

我们在团队内推行Code Review的过程中没有遇到太多阻力。

原因大概有两点，首先管理层方面了解之前遇到的各种问题，也迫切希望能有所改善，所以从一开始就是支持的态度。

其次，绝大部分开发人员觉得在这个过程中能自己能学习到东西，并没有抵触，遇到很好的意见时大家都还是很高兴的。

最后，慢慢的形成了一种氛围，整个团队都会自觉的维护它。

附一张我们审核的对话图，这位童鞋尝试对系统内部散落各地发业务邮件的代码做一个整理，用一套模式来处理，调整了3版才定调，然后修改了很多细节才通过了合并，前后大概用一个多星期时间：

表面上看来Code Review会延缓项目的进度，但是在我们2年多的执行过程中，大多数时候没感觉到有延缓。

原因是，虽然代码合并的周期变长了，但是由于代码质量提高了，导致Bug变少了，由于Bug引起的返工问题也变少了，因此整体的进度其实并没有延缓。

我个人认为对一个成熟的团队其实做Code Review反而会加快整体的项目进度，但是手头上没有统计数据支撑我的观点。（对于软件开发的度量，欢迎有心得的同学告知我）

我们每个分支有权限合并的人都不止一个，这样可以保证有人请假不在的时候，代码仍然可以被其他同事审核通过之后合并。

半年前，我们团队加入了很多新成员，刚加入的新同事对规范、项目、产品的熟悉程度都不高，导致了有一段时间，我们遇到了PR审核周期变长的问题。

加上之前遇到的一些问题，我们总结了一个说明，目的是减轻Code Review对开发人员工作的负担，加快PR审核通过的过程。

说明如下：

Pull Request 的说明

任务完成才能提交PR。

PR应该在一个工作日内被合并或者被拒绝。

PR在有严重问题（包括但不限于架构问题、安全问题、设计问题），太多问题，或者任务无效的情况下会被拒绝。

严禁一个PR里面有多个任务，除非它们是紧密关联的。

PR提交之后只允许针对Review发现问题再次提交代码，除非有充足的理由，严禁在同一个PR中再次提交其它任务的代码。

提交PR时候有一个描述框，内容会自动根据Commit的message合并而成。

切记，如果一次提交的内容包含很多Commit，请不要使用自动生成的描述。

请用简短但是足够说明问题的语言（理想是控制在3句话之内）来描述：

你改动了什么，解决了什么问题，需要代码审查的人留意那些影响比较大的改动。

特别需要留意，如果对基础、公共的组件进行了改动，一定要另起一行特别说明。

审核人员邀请原则：

1. 在创建PR时，Reviewers（审核人）一栏里主要填写“必需审核人”。只有这些人审核都通过，才允许合并。
2. 除了“必需审核人”外，还有一些其它审核人，我们可以在Description里做为“邀请审核嘉宾”@进来。
3. 主干分支间的合并，如Develop => Master，或Master => Develop等，则需要把整个团队（开发+QA）都列为“必需审核人”。

必须审核人的列表由团队决定，可能包括以下人选：

团队Leader

前端架构师（如果有前端代码改动）（可以授权）

后端架构师（如果有后端代码改动）（可以授权）

产品架构师

对此PR解决的问题比较熟悉的（之前一直负责这部分业务的同事）

此PR解决的问题对他影响比较大（比如认领的任务依赖此PR的同事）

其它审核人，包括但不限于：

需要知悉此处代码改动的人但又不必非要其审核通过的同事

可以从这个PR中学习的同事

可以授权指的是，根据约定，Bug修复之类的改动，或者影响较小的改动，前端架构师和后端架构师可以授权团队内的某个资深开发人员，由这个资深开发人员代表他们进行审核。

主干分支之间的合并，大型Feature的合并，前端架构师和后端架构师需要参与。

上述审核人关注的视角不太一样：

团队Leader关注你是否完成了任务，前后端架构师关注是否符合公司统一的架构、风格、质量，产品架构师从整个产品层面来关注这个PR。

熟悉此问题的同事可以更好的保证问题被解决，确保没有引入新问题。

被影响的同事可以及时了解他受到的影响。

团队Leader或者产品架构师如果觉得PR邀请的审核者不足或者过多，必须调整为合适的人员，其它同事可以在评论中建议。

三、收获

我们团队实施Code Review收获不少，总结出来大概有以下几点：

1、短期内迅速提高了代码质量。

原因有几个，大家知道自己的代码会被人审核之后写得会比较认真。

理论上代码质量是由整个团队内最优秀的那个人决定的。

大家也能在Review的过程中学习到其它同事优秀的编码。

2、Bug数量迅速减少。

但是这个我们没有数据统计比较，比较遗憾。
我和QA聊过，他给我的数据是在我们的一个新项目每2周一次的大发布，平均只会发现1~2个Bug。
这点提高了整个团队的幸福感，大家不用经常被火烧眉毛。

3、团队成员对项目的熟悉程度会比较均衡。

新同事通过参与Code Review能很快熟悉团队的规范。
代码不会只有个别人了解、熟悉，Bug谁都能改，新功能谁都能做。
对公司来说避免了人员的风险，对个人来说比较轻松（谁都能来帮你），可以选自己喜欢的任务做。

4、改善团队的氛围
Review的过程中会需要非常多的沟通，多沟通能拉近团队成员的距离。
并且无论级别高低，大家的代码都是要经过Review的，可以在团队内营造一个平等的氛围。
每个成员都可以审查别人的代码，这很容易激发他们的积极性。
亮一下我们的数据:
我们从2014年1月17日开始第一个PR的提交，到2016年7月5日一共发出了6944个PR，其中6171个通过，739个拒绝。日均11.85个PR，最多的一天提了55个PR。
这些PR一共产生了30040个评论，平均每个PR有4.32个评论，最多的一个PR有239个评论。
参与上述PR评论的同事一共有53位，平均每位同事发出了539个评论，最多的用户发出了5311个评论，最少的发了1个（刚推行Code Review就离职的同事）。
需要说明一下，只有简单的问题会通过评论来提出。比较复杂的，比如涉及到架构、安全等方面的问题，其实都会面对面的沟通，因为这样效率更高。

四、总结

虽然有合适的工具支持会更容易实施Code Review，但它本身并不特别依赖具体的工具，所以前文并没有具体指明我们用了什么工具，除了Git。

原因是基于分支的PR流程依赖于大量创建分支，而Git创建一个分支非常的简单，所以PR模式+Git是一个很好的搭配。

我们在切换到Git之前，也做Code Review，采用的是提交代码以后把commit的Id发给相关同事来审查的流程。

审核通过以后会在缺陷跟踪管理系统里面评论，QA同事没见到审核通过的评论就认为任务没有完成，拒绝进行测试。

虽然没有现在这样直接方便，但是也还是做起来了。

PR审核的过程中，新加入的团队成员常见的问题是不符合代码规范之类的，其实是可以通过源代码检查工具来解决的，这部分我们一直在计划中（(╯□╰)），并没有开始实施。

前言

我一直认为Code Review（代码审查）是软件开发中的最佳实践之一，可以有效提高整体代码质量，及时发现代码中可能存在的问题。包括像Google、微软这些公司，Code Review都是基本要求，代码合并之前必须要有人审查通过才行。

然而对于我观察到的大部分软件开发团队来说，认真做Code Review的很少，有的流于形式，有的可能根本就没有Code Review的环节，代码质量只依赖于事后的测试。

也有些团队想做好代码审查，但不知道怎么做比较好。网上关于如何做Code Review的文章已经有很多了，这里我结合自己的一些经验，也总结整理了一下Code Review的最佳实践，希望能对大家做好Code Review有所帮助。

Code Review有什么好处?

很多团队或个人不做Code Review，根源还是不觉得这是一件有意义的事情，不觉得有什么好处。这个问题要从几个角度来看。

首先是团队知识共享的角度

一个开发团队中，水平有高有低，每个人侧重的领域也有不同。怎么让高水平的帮助新人成长？怎么让大家都对自己侧重领域之外的知识保持了解？怎么能有人离职后其他人能快速接手？这些都是团队管理者关心的问题。而代码审查，就是一个很好的知识共享的方式。通过代码审查，高手可以直接指出新手代码中的问题，新手可以马上从高手的反馈中学习到好的实践，得到更快的成长；通过代码审查，前端也可以去学习后端的代码，做功能模块A的可以去了解功能模块B的。可能有些高手觉得给新手代码审查浪费时间，自己也没收获。其实不然，新人成长了，就可以更多的帮高手分担繁重的任务；代码审查中花时间，就少一些帮新人填坑擦屁股的时间；良好的沟通能力、发现问题的能力、帮助其他人成长，都是技术转管理或技术上更上一层楼必不可少的能力，而通过代码审查可以有效的去练习这些方面的能力。

然后是代码质量的角度

现实中的项目总是人手缺进度紧，所以被压缩的往往就是自动化测试和代码审查，结果影响代码质量，欠下技术债务，最后还是要加倍偿还。也有人寄希望于开发后的人工测试，然而对于代码质量来说，很多问题通过测试是测试不出来的，只能通过代码

审查。比如说代码的可读性可维护性，比如代码的结构，比如一些特定条件才触发的死循环、逻辑算法错误，还有一些安全上的漏洞也更容易通过代码审查发现和预防。也有人觉得自己水平高就不需要代码审查了。对于高手来说，让别人审查自己的代码，可以让其他人学习到好的实践；在让其他人审查的同时，在给别人说明自己代码的时候，也等于自己对自己的代码进行了一次审查。这其实就跟我们上学时做数学题一样，真正能拿高分的往往是那些做完后还会认真检查的。

还有团队规范的角度

每个团队都有自己的代码规范，有自己的基于架构设计的开发规范，然而时间一长，就会发现代码中出现很多不遵守代码规范的情况，有很多绕过架构设计的代码。比如难以理解和不规范的命名，比如三层架构里面UI层绕过业务逻辑层直接调用数据访问层代码。

如果这些违反规范的代码被纠正的晚了，后面再要修改就成本很高了，而且团队的规范也会慢慢的形同虚设。通过代码审查，就可以及时的去发现和纠正这些问题，保证团队规范的执行。关于代码审查的好处，还有很多，也不一一列举。还是希望能认识到Code Review和写自动化测试一样，都是属于磨刀不误砍柴工的工作，在上面投入一点点时间，未来会收获代码质量，会节约整体的开发时间。

该怎么做?

现在很多人都已经有意识到Code Review的重要性了，只是苦于不知道如何去实践，不知道怎么样算是好的Code Review实践。

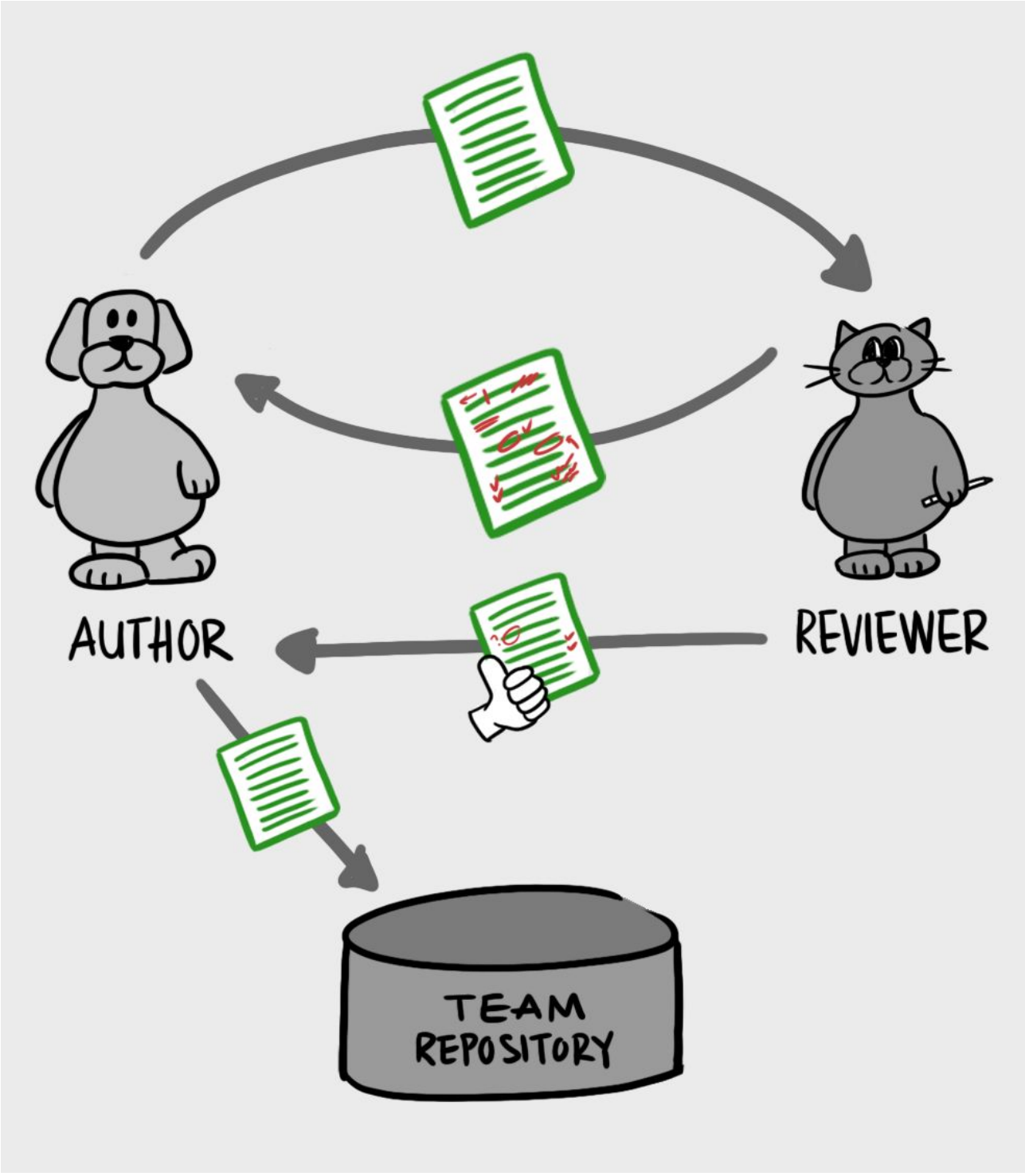
把Code Review作为开发流程的必选项而不是可选项

在很早以前，我就尝试过将代码审查作为代码流程的一部分，但只是一个可选项，没有Code Review也可以把代码合并到master。这样的结果就是想起来才会去做Code Review，去检查的时候已经有了太多的代码变更，审查起来非常困难，另外就算审查出问题，也很难得以修改。

我们现在对代码的审查则是作为开发流程的一个必选项，每次开发新功能或者修复Bug，开一个新的分支，分支要合并到master有两个必要条件：

•所有的自动化测试通过

•有至少一个人Code Review通过，如果是新手的PR，还必须有资深程序员Code Review通过



图片来源：How to Do Code Reviews Like a Human

这样把Code Review作为开发流程的一个必选项后，就很好的保证了代码在合并之前有过Code Review。而且这样合并前要求代码审查的流程，好处也很明显：

•由于每一次合并前都要做代码审查，这样一般一次审查的代码量也不会太大，对于审查者来说压力也不会太大
•如果在Code Review时发现问题，被审查者希望代码能尽快合并，也会积极的对审查出来的问题进行修改，不至于对审查结果太过抵触

如果你觉得Code Review难以推行，不妨先尝试着把Code Review变成你开发流程的一个必选项。

把Code Review变成一种开发文化而不仅仅是一种制度

把Code Review 作为开发流程的必选项后，不代表Code Review这件事就可以执行的很好，因为Code Review 的执行，很大部分程度上依赖于审查者的认真审查，以及被审查者的积极配合，两者缺一不可！如果仅仅只是当作一个流程制度，那么就可能会流于形式。最终结果就是看起来有Code Review，但没有人认真审查，随便看下就通过了，或者发现问题也不愿意修改。真要把Code Review这件事做好，必须让Code Review变成团队的一种文化，开发人员从心底接受这件事，并认真执行这件事。要形成这样的文化，不那么容易，也没有想象的那么难，比如这些方面可以参考：

•首先，得让开发人员认识到Code Review这件事为自己、为团队带来的好处
•然后，得要有几个人做好表率作用，榜样的力量很重要
•还有，对于管理者来说，你激励什么，往往就会得到什么
•最后，像写自动化测试一样，把Code Review要作为开发任务的一部分，给审查者和被审查者都留出专门的时间去做这件事，不能光想着马儿跑得快又舍不得给马儿吃草

如何形成这样的文化，有心的话，还有很多方法可以尝试。只有真正让大家都认同和践行，才可能去做好Code Review这件事。

一些Code Review的经验技巧

在做好Code Review这件事上，还有一些经验技巧可以参考。

选什么工具辅助做CODE REVIEW?

现在很多源代码管理工具都自带Code Review工具，典型的像Github、Gitlab、微软的Azure DevOps，尤其是像Gitlab，还可以自己在本地搭建环境，根据自己的需要灵活配置。

配合什么样的开发流程比较好?

像Github Flow这样基于分支开发的流程是特别适合搭配Code Review的。其实不管什么样的开发流程，关键点在于代码合并到master（主干）之前，要先做Code Review。

真遇到紧急情况，来不及代码审查怎么办?

虽然原则上，必须要Code Review才能合并，但有时候确实会存在一些紧急情况，比如说线上故障补丁，而又没有其他人在线，那么这种情况下，最好是在任务管理系统中，创建一个Ticket，用来后续跟踪，确保后续补上Code Review，并对Code Review结果有后续的代码更新。

先设计再编码

有些新人发现自己的代码提交PR（Pull Request）后，会收到一堆的Code Review意见，必须要做大量的改动。这多半是因为在开始做之前，没有做好设计，做出来后才发现问题很多。建议在做一个新功能之前，写一个简单的设计文档，表达清楚自己的设计思路，找资深的先帮你做一下设计的审查，发现设计上的问题。设计上没问题了，再着手开发，那么到Review的时候，相对问题就会少很多。

代码在提交CODE REVIEW之前，作者要自己先REVIEW和测试一遍

我在做代码审查的时候，有时候会发现一些非常明显的问题，有些甚至自己都没有测试过，就等着别人Code Review和测试帮助发现问题。这种依赖心理无论是对自己还

是对团队都是很不负责任的。一个好的开发人员，代码在提交Code Review之前，肯定是要自己先Review一遍，把该写的自动化测试代码写上，自己把基本的测试用例跑一遍的。我对于团队提交的PR，有个要求就是要在PR的描述中增加截图或者录屏，就是为了通过截图或者录屏，确保提交PR的人自己是先测试过的。这也是一个有效的辅助手段。

PR要小

在做Code Review的时候，如果有大量的文件修改，那么Review起来是很困难的，但如果PR比较小，相对就比较容易Review，也容易发现代码中可能存在的问题。所以在提交PR时，PR要小，如果是比较大的改动，那么最好分批提交，以减轻审查者的压力。

对评论进行分级

在做Code Review时，需要针对审查出有问题的代码行添加评论，如果只是评论，有时候对于被审查者比较难甄别评论所代表的含义，是不是必须要修改。建议可以对Review的评论进行分级，不同级别的结果可以打上不同的Tag，比如说：

•[blocker]: 在评论前面加上一个[blocker]标记，表示这个代码行的问题必须要修改
•[optional]：在评论前面加上一个[optional]标记，表示这个代码行的问题可改可不改
•[question]：在评论前面加上一个[question]标记，表示对这个代码行不理解，有问题需要问，被审查者需要针对问题进行回复澄清

类似这样的分级可以帮助被审查者直观了解Review结果，提高Review效率。评论要友好，避免负面词汇；有说不清楚的问题当面沟通 虽然评论是主要的Code Review沟通方式，但也不要过于依赖，有时候面对面的沟通效率更高，也容易消除误解。另外文明用语，不要用一些负面的词汇。

总结

Code Review是一种非常好的开发实践，如果你还没开始，不妨逐步实践起来；如果已经做了效果不好，不妨对照一下，看有没有把Code Review作为开发流程的必选项而不是可选项？有没有把Code Review变成一种开发文化而不仅仅是一种制度？