# 代码评审（Code Review）

**日期2020-07-31**

## 引言

代码评审（Code Review）已经被广泛的认可为一种非常好的做法，可以使缺陷发现前移，很多团队也都在进行代码评审工作。但是，投入多少高级别资源进行代码评审比较合适？代码评审采用何种方式的投入产出比最高？不同的场景和对象采用何种代码评审方法和策略？这些问题都没有很好的应对方法。

## 编写目的

代码评审主要目的是能够确保代码库整体的健康程度随着时间的推移而不断改善，提高代码和产品的整体质量。

本文档的愿景也是这样，但这并不是编写此文档的目的，当然也不是为了说明代码评审的重要性和怎么做会更好，而是希望通过适当的规范和标准，建立评审和协作流程以提升当前工作的质量，让代码评审成为开发流程的一部分，营造良好的代码评审文化。

## 代码评审指南

在谷歌的工程实战指南，开源了一份非常完善且能够指导代码评审实战的操作性文档，其中很多的地方都可以借鉴过来，在实践运用的时候，可以根据团队的实际情况进行调整和完善。在这会对其中主要的内容做出介绍,详情请参阅：[Google Engineering Practices Documentation](https://github.com/google/eng-practices)。

### 术语

**CL:** "Change List"的缩写，代表已经进入版本控制软件或者正在进行代码评审的变更。其他组织经常称为"change"或者"patch"。

**LGTM:** "Looks Good to Me."的缩写，评审者批准CL时会这么说。

### 评审者指南

#### **评审标准**

通常而言，只要代码对系统有明显的提升且正常工作，即便不完美，评审者也应该倾向于通过这次变更。这是所有代码评审中的最高原则。当然这也有些局限。例如，如果变更里加入了有些评审者在系统里不想要的功能，即便代码设计的很好，评审者也可以拒绝掉。

没有完美无缺的代码，只有越来越好的代码。代码评审者绝不应该要求开发者打磨好CL中的每个细节才予以通过，相反，评审者应该权衡项目进度和他们给出建议的重要性，适当放宽要求。评审者应该追求持续提高，而不是追求完美。那些可以提升整个系统可维护性、可读性和可以理解性的变更不应该因为代码不够完美而被推迟几天甚至几周。

评审者要始终不拘谨于在代码评论里提示可以更好的想法。 但如果不是很重要信息，可以在评论前面加上标识告诉他们可以忽略。

* **指导性**

代码评审有个重要的作用，那就是可以教会开发者关于语言、框架或者通用软件设计原理。在代码评审中留下评论来帮助开发者学习新东西是很值得提倡的，毕竟共享知识也是长期提升系统代码健康度的一部分。但请注意，如果你的评论纯粹是教育性的，并且不是这篇文档中提到的关键标准，请在前面加上“Nit:”标识，或者明确指出不需要在这次变更中解决。

* **原则**

1. 代码不能损害系统的整体质量；
2. 技术和数据高于意见和个人偏好；
3. 权衡项目进度和建议的重要性，适当放宽要求；
4. 紧急情况下，应该更加注重评审速度和代码的正确性(能否解决当前的紧急情况)；
5. 代码只要能明显改进系统和改善系统的整体运行状况（功能完成性、可理解性、可维护性、可测试性···），即便不够完美，也应该批准；
6. 系统里不想要的功能，即便代码设计的很好，也可以拒绝掉；

* **解决代码冲突**

如果评审中有任何冲突，开发人员和评审人员都应该首先根据指南文档中的内容，尝试达成一致意见。

当很难达成一致时，开发者和评审者不应该在评审评论里解决冲突，而是应该召开面对面会议或者找个权威的人来协商。（如果你在评论里协商，确保在评论里记录了讨论结果，以便日后其他人翻阅。）

如果这样都解决不了问题，那解决问题的方式就应该升级了。通常的方式是拉着团队一起讨论、让团队主管来权衡、参考代码维护者的意见，或者让管理层来决定。不要因为开发者和评审者不能达成一致而把变更一直放在那里。

#### **评审关注点**

* **设计**

代码是否设计良好并且适合当前的系统。

* **功能性**

代码功能和业务功能是否一致，对用户和维护者是否友好

* **复杂性**

是否过度设计，代码能不能更简单？ 其他开发者能否快速理解并在未来很容易地使用这段代码？

* **测试**

代码是否有正确和设计良好的自动化测试样例？测试是否正确、合理和有用的。

* **命名**

变量、类名等命名是否清晰明了

* **注释**

注释的可读性，是否清晰有用？（不要写干了什么，而是写为什么这么写）

* **代码风格**

确保遵循编码规范，但不强制

* **文档**

如果变更改变了用户构建、测试、交互或者发布代码相关的逻辑，是否更新了相关文档。

* **安全性**

接口是否安全。

#### **评审速度**

持续优化团队开发新产品的速度，而不是优化单个开发人员编码速度。个人开发速度固然重要，但并没有团队整体的速度那么重要。只有一种情况下，个人速度胜过团队速度：正在处理诸如编码之类的重要工作，开发在专注中被打断后需要很长的时间才能重新恢复到专注状态，这明显得不偿失。

提高代码评审的响应速度（从评审请求提交到通过的时间）。评审人员在不中断自己的工作情况下，在工作间隙合理响应评审请求，尽可能做到在一个工作日内响应。

但是，不要为了提升速度牺牲评审标准和代码质量，从长远来看，这并不会提升团体整体的速度。

#### **如何评论**

* **礼貌**

通常来说当你评审代码时保持礼貌和尊重能使开发人员更加清晰，得到更多帮助。这样是为了保证你的代码评论仅仅针对的是“代码”而不是针对“开发人员”。你不必一直这么去做，但是当你的评论会让开发人员生气或者产生争执时有必要这么去做。比如：

不好的例子: 你为什么会在这里使用线程，这样做难道会有任何好处？

好的例子: 我并没有发现这个并发模块给程序带来了多少帮助，并且还增加了程序的复杂性，因此我认为这段代码最好是用单线程而不是多线程。

* **解释清楚原因**

从上面“好”的例子当中你能发现，这样有助于开发人员理解为什么你写了这些评注。你不一定非得包含这些信息在你的评注里面，但是适当的多解释你的意图或者多给出一些提升代码质量的建议都是非常好的实践。

* **给予指导**

通常来说修复CL是开发人员的职责而不是评审人员的。你不需要向开发人员提供详细的解决方案或者代码。

但是这并不意味着评审员就不应该提供帮助。你需要在指出问题和提供直接指导之间找到平衡。指出问题并且帮助开发人员决策能够帮助开发人员学同事让代码评审变得更加简单。这样也能产生更好的方案，因为开发人员比评审者更加了解代码。

尽管这样，有时候直接给出指导，建议甚至是代码更有帮助。代码评审的主要目的是尽可能得到最好的CL。其次是提高开发人员的技能这样就能减少以后评审的次数。

* **接受解释**

与其要求让开发人员解释一段你看不懂的代码，其实更应该做的是让他们重写代码，让代码更清晰。当一段代码不是太过于复杂的时候通过加一些注释偶尔也是一种不错的做法。

解释仅仅是写在评审工具里的，不利于将来的代码阅读者。只有极少数情况是可行的，比如你对你评审的需求不太熟悉，但是开发人员解释的是大多数人都知道的。

#### **解决冲突**

有时开发者会在Code Review中反驳你，他们可能不同意你的意见，或者抱怨你太严格了。

* **谁是谁非？**

当开发者不同意你的建议时，首先花点思考下他们是否是对的，但通常而言你比他们更熟悉代码，所以可能在某个方面理解更深。他们的争论有意义吗？从代码健康的角度来看他们的反驳有意义吗？如果是，让他们知道他们是对的，然后这个问题就解决了。

然而，开发者不总是对的，这种情况下评审者应当进一步介绍为什么他们的建议是对的。好的解释不仅得体现出对开发人员回复的理解，而且还要说明为什么要这么改。

如果评审者认为他们的建议可以改善代码质量，并且他们认为带来的代码质量改进值得开发者做这些额外的工作，评审者就应该坚持自己的立场。

不积跬步无以至千里，不积小流无以成江河。有时候要让开发者接受，你得花很多时间反复解释，但始终确保该有的礼貌，并让开发者知道在知道他们了什么，即便是你不同意。

* **惹恼开发者**

有时评审者会认为坚持让开发者做出改动，可能会惹恼开发者。有时开发者确实会恼怒，但这种情况通常都很短暂，而且之后他们会感激你帮助他们提高了代码质量。 如果你在代码评审中很有礼貌，开发者根本不会被惹恼，这种担心是多余的。通常，令惹恼开发者的是你写注解的方式，而不是你对代码质量的坚持。

* **稍后解决**

一种常见的反驳原因是开发者希望能尽快完成任务。他们不想一轮又一轮地做代码评审，然后就会说他们会在后续CL中处理这些问题，你只需要通过就行。有些开发者做的很好，他们会立马提交后续CL处理这些问题。然而，经验告诉我们原始CL通过之后拖的时间越久，就越不可能修复。除非开发者在当前CL通过后立马修复，否则他们就不可能修复。这并不是开发者不负责任，而是因为他们有好多工作要做，而修复工作也会因为工作压力而被遗忘。所以最好坚持让开发者\*现在\*就在CL中处理掉这些问题，“留着以后清理”是一种不可取的方式。

如果CL中引入了新的复杂性，提交之前必须清理掉，除非是紧急情况。 如果CL中暴露出一些目前还无法定位的问题，开发者应该记录下这些bug，并将其分配给他们自己，确保这些问题不会被遗忘。他们还可以在代码中加入 TODO 注释，指向已经记录好的 bug。

* **抱怨太严格**

如果你之前代码评审很宽容，然后突然变得严格起来，可能会引起一些开发者的抱怨。不过没关系，加快代码评审的速度通常会让这些抱怨消失。

有时，这些抱怨可能需要几个月的时间才能消除，但最终开发者在看到产出的优质代码时会理解严格代码评审带来的价值。有时候，一旦发生某些事让他们真正看到严格代码评审的价值，抗议最大声的人甚至会成为你最坚定的支持者。

### 开发者指南

#### **如何处理评审者的建议**

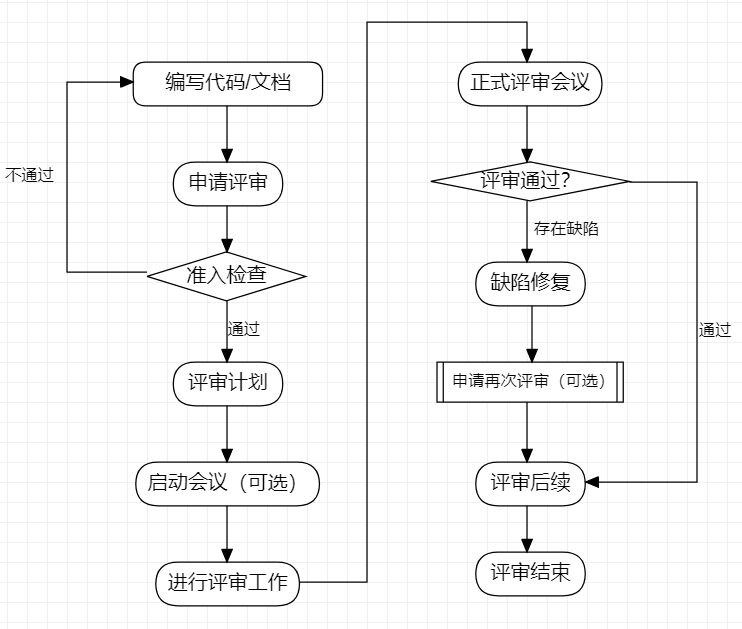
自我反思，换位思考，达成共识。

## 正式评审流程

在此，主要讲述正式评审的流程，非正式评审将会在**[第五章节 评审方式](#_评审方式)**中说明。同时有必要说明下评审中涉及到的角色和职责：

1. **开发者（Developer）**：对代码/文档的质量负最直接责任；
2. **评审主持人（Moderator）**：保证评审过程规范性和一致性；
3. **评审人（Reviewer）**：掌握相关知识，进行平时工作，记录所有发现的缺陷；
4. **质量保证人员（QA）**：一般由测试人员担任，作为第三方对整个过程进行审计，跟踪关键步骤，检查输入和输出项，保证评审过程中发现的缺陷得到有效的修改；

正式评审，一般由开发组长组织代码评审，主要检查代码和设计的一致性，代码对文档标准的遵循及代码的可读性，代码的逻辑表达式正确性，代码结构的合理性等方面。正常的流程如图4-1所示：



**图 4-1**

### 申请评审

确定评审主持人，指定具备一定经验的人员担任，必要时请项目经理协助指定。

准备相关资料文档，帮助评审人了解待评审文档。

### 准入检查

* **文档**

格式检查，是否符合相关模板。

* **代码**

1. 编译通过；
2. 辅助工具检查；
3. 创建标签（Tag）保存被评审版本；

最后将文档/代码及所有相关资料发送给评审主持人。

### 评审计划

1. 确定评审人；
2. 决定是否需要启动会议（可选，一般在评审人对待评审代码/文档内容不熟悉的情况下进行）；
3. 准备文档和评审记录表；
4. 提前1-2天将所有文档交给评审人，并通知其他的相关人员；

### 评审工作

1. 评审人阅读代码/文档，记录反馈意见；
2. 填写评审记录表中的“缺陷记录”，在评审会议前交给评审主持人；
3. 反馈意见类型：1）主要缺陷；2）次要缺陷；3）疑问；

### 正式评审会议

1. 评审主持人在会议开始前汇总所有反馈；
2. 评审会议上，记录评审人的评审工作耗时，并澄清每条反馈信息，支持讨论；

### 评审后续

1. 评审主持人整理评审报告和记录表，统计各项评审信息；
2. 将评审报告和记录表发给所有参会人员；
3. 开发者更新代码/文档；
4. 保持关注评审发现的缺陷，确保这些缺陷不被遗忘，让开发者尽快处理掉这些问题；

## 评审方式

在实际项目中，规模、工作量、进度等估算活动过分依赖个体经验，评审活动无法衡量其有效性，不同的评审方式，效率和成本也不一样。对此，项目组应该根据代码质量要求平衡进度要求，规划迭代中的评审，使评审轻量化、对他人友好，确保所有人都参与进来，将其融合到每个人的日常工作流程中。

### 提交前评审

一旦完成了一段代码的编写和测试，就要准备好提交代码，先把代码提交给团队进行最终评审是一种好的做法。提交前评审可以让每个人都有机会看到所构建的东西、提出问题并找到错误。像Google、Facebook都严格执行提交前评审。具体的流程如下：

1. 程序员在试图提交代码变更到代码库之前，先提交变更申请，变更申请包含了这次变更的内容；
2. 评审人查看变更内容，评估变更，与变更申请人沟通，评估是否通过变更；
3. 如果评审人通过变更，则变更申请人才可以提交代码到代码库；
4. 如果评审人不通过变更，则变更申请人需要根据讨论结果或评审建议做出修改，直到与评审人达成一致，通过评审，才可以提交代码到代码库；

### 静态分析

除了让人们评审我们的代码，使用自动化的工具去检查代码也是个好主意。这些工具可以帮助我们识别出一些常见的源代码bug的来源，发现代码风格的问题，检测重复的代码和没有使用的代码，对一些复杂的指标进行计算以找出需要重构的代码，找出使用了不安全特性的代码，识别出潜在的安全漏洞，发现潜在的内存泄漏。

### 结对编程

结对编程是一种开发技术，让两名程序员在一台计算机前面工作。一人是驾驶员，负责编写代码；另一人是观察员，负责评审代码，并在更高层次上思考程序。这两名程序员定期交换角色。

在实际工作中，并不推荐这种评审方式。尽管结对编程能提高设计和代码质量，但缺点也是显而易见的，首先需要更多的人力和时间，同时并不是每个开发人员都希望自己编码时，有个人在旁边指点的，这样会限制开发人员的思维和创新，以及会影响其专注力，对整个团队长期成长是不利的。

## 评审工具

GitLab Merge Request

Gerrit

Fisheye + Crucible

Phabricator

## 实施评审

代码评审是每一个公司都应该应用的必备实践，但要更好地实施也要遵循若干指南。

首先，为代码的每个部分指定一个所有人是一个好做法。所有人并不是唯一允许修改这部分代码的人，但他们有责任知道这段代码如何工作，并且要保持它运行。这意味着他们要决定可以对代码做什么样的修改、谁可以进行评审、必须强制执行什么编码标准、何时可以部署代码。

其次，需要提前将代码评审指南写下来，营造良好的代码评审文化。这些指南应该包含评审者期望掌握的事项的备忘录，比如代码是否容易阅读、是否遵循团队的编码约定、是否包含测试，这些指南也应该定义实施代码评审的代码。

代码评审并不是炫耀自己的知识，取笑他人的代码或者责怪他人。它是一种学习的手段，能让团队中的所有人变得更加出色，让所有人都感觉（及表现得）像主人一样。为了让所有人都达成共识，代码评审指南应该定义什么代码必须被评审（例如是所有的代码还是实现关键任务的那部分），什么时候代码必须被评审（例如是每次提交前、合并分支前还是在每周的评审会议上），谁的代码必须被评审（例如所有人，不管级别有多高），谁负责给出评审意见（例如所有人，不管级别多低），什么类型的意见是合适的（例如花时间标注代码好的一面，而不仅仅是不好的一面；永远不要侮辱提交者本人；不要害怕承认你不理解某些内容）。

1. 保持小规模的评审。10行代码的修改是很容易评审的，但是1000行代码的评审几乎是不可能的。这就意味着我们应该鼓励开发人员进行小修改，增量式地提交，而这恰好也是减少bug、合并冲突和后期集成问题出现概率的好方法。
2. 提倡代码内部开源。代码供他人重用或使用 在其他人的项目中进行协作，补充，测试或修复代码。