Google Code Review Guidelines

Google积累了很多最佳实践，涉及不同的开发语言、项目，这些文档，将Google工程师多年来积攒的一些最佳实践经验进行了总结并分享给众开发者。学习下这里的经验，我们在进行项目开发、开源协同的过程中，相信也可以从中受益。

Google目前公开的最佳实践相关文档，目前包括：

- Google's Code Review Guidelines，Google代码review指引，包含以下两个系列的内容：
  - The Code Reviewer's Guide
  - The Change Author's Guide

这了涉及到Google内部使用的一些术语，先提下：

- CL: 代表changelist，表示一个提交到VCS的修改，或者等待review的修改，也有组织称之为change或patch；
- LGTM：代表Looks Good to ME，负责代码review的开发者对没有问题的CL进行的评论，表明代码看上去OK；

The Code Reviewer's Guide

从代码reviewer的角度出发，介绍下Google内部积累的一些good practices。

Introduction

Code Review（代码评审）指的是让第三者来阅读作者修改的代码，以发现代码中存在的问题。包括Google在内的很多公司会通能过Code Review的方式来保证代码和产品的质量。

前文已有提及，CR相关内容主要包括如下两个系列：

- The Code Reviewer's Guide
- The Change Author's Guide

这里先介绍下CR过程中应该做什么，或者CR的目标是什么。

What Do Code Reviewers Look For?

Code review应该关注如下方面：

- Design：程序设计、架构设计是否设计合理
- Functionality：代码功能是否符合作者预期，代码行为是否用户友好
- Complexity：实现是否能简化，代码可读性是否良好，接口是否易用
- Tests：是否提供了正确、设计良好的自动化测试、单元测试
- Naming：变量名、类名、方法名等字面量的选择是否清晰、精炼
- Comments：是否编写了清晰的、有用的注释

- Style：代码风格是否符合规范
- Documentation：修改代码的同时，是否同步更新了相关文档

Picking the Best Reviewers

一般，Code review之前，我们应该确定谁才是最好的、最合适的reviewer，这个reviewer应该**“有能力在比较合理的时间内对代码修改是否OK做出透彻、全面的判断”**。通常reviewer应该是编写被修改代码的owner，可能他是相关项目、相关源文件、相关代码行的创建者或者修改者，意味着我们发起Code review时，同一个项目可能需要涉及到多个reviewer进行Code review，让不同的、最合适的reviewer来review CL中涉及到的不同部分。

如果你心目中有一个合适的reviewer人选，但是这个人当前无法review，那么我们至少应该“@”或者“邮件抄送”该reviewer。

In-Person Reviews

如果是结对编程的话，A写的代码B应该有能力进行代码review，那么直接找B进行review就可以了。

也可以进行现场评审（In-Person Reviews），一般是开发者介绍本次CL的主要内容、逻辑，其他reviewer对代码中年可能的问题、疑惑进行提问，本次CL的开发者进行解答，这种方式来发现CL中的问题也是常见的一种方式。较大型、急速上线的项目，这种方式团队内部用的还是比较多的。

How to Do a Code Review

这里总计了一些Code review的建议，主要包括如下一些方面：

- The Standard of Code Review
- What to Look For In a Code Review
- Navigating a CL in Review
- Speed of Code Reviews
- How to Write Code Review Comments
- Handling Pushback in Code Reviews

The Standard of Code Review

代码review的主要目的就是为了保证代码质量、产品质量，另外Google的大部分代码都是内部公开的，一个统一的大仓库，通过代码review的方式来保证未来Google代码仓库的质量，Google设计的代码review工具以及一系列的review规范也都是为了这个目的。

为了实现这个目标，某些方面需要做一些权衡和取舍。

首先，开发者必须能够持续优化。如果开发者从来不对代码做优化，那么最终代码仓库一定会烂掉。如果一个reviewer进行代码review时很难快速投入，如不知道做了哪些变更，那么代码reviewer也会变得很沮丧。这样就不利于整体代码质量的提高。

另外，代码reviewer有责任维护CL中涉及到的修改的质量，要保证代码质量不会出现下降，时间久了也不至于烂尾。有的时候，某些团队可能由于时间有限、赶项目，代码质量就可能会出现一定的下降。

还有，代码reviewer对自己review过的代码拥有owner权限，并要为代码后续出现的问题承担责任。通过这种方式来进一步强化代码质量、一致性、可维护性。

基于上述考虑，Google制定了如下规定来作为Code review的内部标准：

> **In general, reviewers should favor approving a CL once it is in a state where it definitely improves the overall code health of the system being worked on, even if the CL isn't perfect.**
>
> 一般，reviewers对CL进行review的时候，达到approved的条件是，CL本身可能不是完美的，但是它至少应保证不会导致系统整体代码质量的下降。

当然，也有一些限制，例如，如果一个CL添加了一个新特性，reviewer目前不想将其添加到系统中，尽管这个CL设计良好、编码良好，reviewer可能也会拒绝掉。

值得一提的是，没有所谓的perfect code，只有better code。

- 代码reviewers应该要求开发者对CL中每一行代码进行精雕细琢，然后再予以通过，这个要求并不过分。
- 或者，代码reviewers需要权衡下他们建议的"精雕细琢"的必要性和重要性。代码reviewers应该追求代码的持续优化，不能一味地追求完美。对于提升系统可维护性、可读性、可理解性的代码CL，reviewers应该尽快给出答复，不能因为一味追求完美主义将其搁置几天或者几周。

Mentoring

Code review对于教授开发者一些新的东西，如编程语言、框架、软件设计原则等是非常重要的手段。进行代码review的时候添加一些comments有助于帮助开发者之间分享、学习一些新东西。分享知识也是持续改进代码质量的重要一环。

需要注意的是，如果comments内容是纯教育性的、分享性的，不是我们前面提到的强制性的必须应该做出优化的，那么最好在comments内容里面添加前缀"**Nit (Not Important)**"，这样的评论表示当前CL中不一定非要做出对应的优化、修改，只是一个建议、分享。

Principles

- 技术本身、数据至上，以及一些个人偏好
- 代码风格的重要性，力求代码风格的一致，如果没有明确的代码风格，就用之前作者的风格
- 业内的代码风格、个人偏好，需要在二者之间适当平衡，reviewer也应该在代码风格上注意
- 如果没有明确的一些规定，reviewer可以要求CL作者遵循当前代码库中的一些惯用的做法

上述各条，均以不降低系统整体代码质量为度量标准。

Resolving Conflicts

如果在代码review中出现了冲突的意见、观点，首先，开发者、reviewer应尽可能基于之前的代码、现在CL的代码达成一个共识，如果仍然达不成共识，或者很困难，最好能进行面对面沟通，或者将当前CL升级一下，供更多的人员进行讨论。可以考虑将技术Leader、项目经理拉进来一起讨论下。

这种面对面的方式比单纯地通过comments进行讨论要高效、友好地多，如果条件不允许只能通过comments方式进行，那么对于讨论的结果，应进行适当的总结，给出一个结论，方便之后的开发者能够了解之前的讨论结果。

目标就是，不要因为代码reviewer和CL作者之间达不成一致，就长时间将CL搁置。

What to Look For In a Code Review

结合前面提到的一些Code review标准，将Code review中应该关注的点进一步细化，主要以下内容。

Design

Code review过程中最重要的事情就是看CL的整体设计是否合理，如CL中涉及到的各个部分的代码之间的接口、交互、衔接是否合理，是业务代码修改还是库的修改，和系统整体的集成是否合理，现在这个时间点添加这个新特性是否合理等等。

Functionality

CL的功能是否符合开发者预期，开发者期望的这部分修改是否对用户友好，这里的用户包括产品用户（实际使用产品的人员）和开发者（将来可能使用这部分代码的人员，如CL修改的库代码）。

一般，我们希望开发者发起Code review之前，能够对CL进行充分的测试确保功能是符合预期的。但作为reviewer，还是要检查下边界条件的处理是否到位，比如并发中的data race问题，确保代码中不存在“可能”的bug。

reviewer有条件的话，也可以亲自验证下CL，比如CL是面向用户的产品（如UI改变），单纯看代码不能直觉地感受到做的调整，reviewer可以亲自patch这部分代码、编译构建、安装之后来体验下具体的改变。如果不是特别方便的话，也可以找相应的开发者提供一个demo演示下CL中涉及的变化。

另一个非常重要的点是，要检查CL中是否存在某种类型的并发问题，如deadlocks、race conditions等。这些问题也不是运行一下代码就能发现的，往往需要reviewer来细致地考虑下相关的操作，才能判定是否有引入该类问题。

Complexity

CL是否过于复杂，这个需要对CL中的不同层次的内容进行逐一检查，如每行代码是否过于复杂，函数实现是否过于复杂，类实现是否过于复杂等。“过于复杂”意味着，不能被其他开发者快速吸收、理解。也有个笑话，开发者在调用、修改自己编写的代码的时候容易引入引入bug。这些都说明了复杂性的问题所在。

过度设计也是Complexity中的一种，开发人员可能对当前需要解决的问题进行了解决，但是在此基础上进行了过度的、不必要的延伸。reviewers需要小心判断，识别出一个CL中是否存在过度设计的问题。reviewers应鼓励开发人员解决当前已经存在的、明确的问题，避免开发人员做些主观上认为未来可能需要的某些功能。

Tests

CL中应该包含必要的单元测试、集成测试等必要的测试用例，除非这个CL是为了紧急解决线上问题，这种情况下未能及时添加可以原谅。

确保CL中包含的测试用例是正确的、有意义的、有效的，不要为了写测试用例而写测试用例，测试用例是为了辅助验证我们的代码是否符合预期的，因此几乎任何时候，确保测试用例有效都是至关重要的，测试用例也是代码维护工作的一部分。

Naming

命名（变量名、函数名、类名等等）是否合理，一个好的名字应该长度适中，又能够清晰地表达其代表什么。

Comments

开发者是否提供了清晰、易于理解的英文注释。

提供的注释是否是必要的，注释不是笔记，注释应该用来解释某段代码为什么存在，不需要解释这段代码要干什么，这样的注释才有意义。如果不提供该注释将无法理解该代码，请考虑一下设计、编码实现复杂度等是否有问题，是否可以简化。也有一些例外情况，如正则表达式或者复杂的算法，提供注释注明其作用是有价值的。

当前CL之前的注释，也需要检查一下，可能当前CL解决了一个问题，对应的某个TODO可以删除了。

注：这里的注释不同于类注释、模块注释、函数注释，这些注释需要表明其存在的目的、功能、如何使用、有什么行为或副作用。

Style

代码风格问题，Google也提供了一些代码规范，这里的规范涵盖了多种编程语言的规范，请确认CL遵循此规范。

如果某些地方没有明确的代码规范指引，而你有觉得开发者这种写法不太好，你想提出点建议的话，review的时候请在意见里面加个前缀“Nit:”，Not Import Pick，以表明这是个非强制性的建议。不要因为个人偏好问题，阻塞CL的代码review过程。

CL里面不要既做代码逻辑修改，又做大范围格式化的操作，这可能让review、merge、rollback等变得复杂，如果确实有必要做这样的调整，请提供两个CL。第一个用来格式化，第二个在此基础上再进行代码逻辑的修改。反过来也可以，但是不要一次做两件“变动巨大”的事情。

Documentation

如果一个CL改变了用户build、test、interact with、release code的方式，请同步检查下文档是否也要更新，包含READMEs、g3doc pages以及其他一些生成的reference docs。如果CL删除了或者废弃了某些代码，考虑下是否对应的文档内容也有需要删除的。

如果发现缺少某些文档内容，请联系开发者补齐。

Every Line

检查review任务中你分配的每一行代码。对于某些自动化生成的数据文件、代码文件（如*.pb.go）、大型数据结构，你可以快速扫过去，但是如果是开发者自己写的class、function、block of code，这种不能扫一下就过去了，需要仔细检查。

开发者自己写的代码，也有重要的、不重要的之分，某些代码片段需要更仔细地review，比如某些分支判断逻辑，reviewer需要培养这方面的识别重要代码路径的能力。如果目前没有这种识别关键路径的能力，也至少应该确保你看懂了这些代码。

如果review过程中，你发现某些代码实在是费解，严重拖慢了review进度，这种情况下不要犹豫，直接联系代码作者来解释它是干什么的、能否简化实现，然后再review。在Google雇佣了很多优秀的开发者，如果某位看不懂，那么其他人可能也看不懂，随着业务需求变更，将来新加入的开发者更可能看不懂。在Tencent以及其他公司，这种情况也是一样的，reviewer应该直接联系代码作者来解释、优化代码实现，某种意义上这也是为提升整体代码质量做贡献了。

如果你理解了代码逻辑，但是对于某些部分你拿不准，请邀请另一位资历更深的reviewer来review，特别是对于那些安全、并发、可用、国际化等方面复杂的问题。

Context

review过程中查看CL相关代码上文是有帮助的。代码review的时候，代码review工具只会显示几行修改的代码，但是前后与之相关的代码却可能大范围折叠。有时，你不得不查看整个文件内容来确保CL是有意义的。再比如，有时review工具只显示了4行代码，但是这四行代码位于一个几十上百行的方法中，如果你查看了CL的上下文，就会意识到这个函数的实现需要适当拆分成几个小的函数。

将整个系统的代码作为Context来判定CL代码质量也是有必要的，如果这个CL中的代码质量低于系统整体代码的质量，请不要接受这样的代码，这回导致整体代码质量的下降。

Good Things

如果CL中有比较好的做法，请告诉开发者，特别是对于你的修改建议开发者很好地完成的时候。代码review通常是聚焦于可能的错误，但是这个过程也给了我们鼓励、学习good practices的机会。Mentoring机制，我们Tencent也有，去鼓励新人如何做的更好，比告诉他们哪里错了，更有价值。

Summary

简单总结下，Code review的时候也确保如下几点：

- 代码是否设计良好
- 功能对于代码的使用者而言更友好
- UI的改变是有意义的
- 并发处理是安全的
- 不进行过度设计，没有演变到那种不必要的复杂
- 没有去实现一些将来可能需要但是现在不需要的东西
- 提供了有效的测试用例，包括单元测试等
- 测试是有用的，测试是经过精心设计的，不要写一堆没用的测试
- 命名选择都是合理的，如变量名、函数名等
- 注释是有价值的、有用的，注释解释了why而不是what，当然也有例外
- 文档中对代码变更也进行了合理的补充
- 代码遵循制定好的代码规范

确保逐行review分配的代码（代码生成工具生成的可以快速浏览），当然你可以合理分配review时间对部分代码进行重点review，但请确保你看懂了每行代码。注意code context，确保提升整体代码质量，对于review过程中发现的good practices适当鼓励开发者。

Navigating a CL in Review

现在我们知道了What to look for，如果review涉及到多个文件，如何最高效地进行review呢?

- review之前，查看CL的整体表述，确认是否有意义
- 首先，看CL中最有价值的部分，整体设计是否良好
- 然后，再根据合理顺序看CL中剩余的部分

Step One: Take a broad view of the change

查看CL描述，先理解CL是做了什么工作，这个CL是否有意义。如果reviewer觉得这个修改没有意义，并决定拒绝该CL的时候，请选择合适的措辞向开发者解释清楚。

例如，reviewer：“哇看上去你做了很多工作，非常感谢，但是我们未来方向是要移除你这里修改的FooWidget组件，如果你有时间，可以帮忙重构下BarWidget组件吗？”

通能过这种方式，reviewer既向developer or contributor表明了立场、态度，也不失礼貌，保持礼貌是重要的，特别是对于开源项目，即便是你不认同贡献者的CL，也至少应看到他尝试进行付出。保持礼貌的review、讨论、沟通有助于维护开源协同的氛围。

如果你收到了不少CLs，但是这些都不是你想要的，可能就需要从更高角度出发来解决这个问题，比如完善下Contributing文档，告知开发者项目需要什么，哪些方面鼓励优先解决等等。

Step Two: Examine the main parts of the CL

找到CL涉及的最主要修改部分，优先进行review。CL中逻辑的变动可能主要集中在少数几个文件中，找到这些changes优先进行review，这有助于聚焦修改的主要部分，加速整体的review进度。如果CL涉及代码太多，很难识别哪部分是主要部分，那可以先问下开发者哪部分是主要修改，或者让开发者把当前这次CL拆分成多个CLs，然后再发起review。

如果在review CL主要部分的时候，发现设计上明显存在不合理的地方，reviewer应该立即发送review comments给开发者，其他的代码可以不用review了，继续review纯粹是浪费时间。因为既然存在明显的设计问题，等开发者修改之后，剩余的要review的代码可能根本不存在了。

发现有明显设计问题，立即发送design review comments，还有两个好处：

- 开发者可能会发起一个CL之后，会立即在这个CL的基础上继续进行其他的修改工作。如果前面的CL存在明显的设计问题，那么很可能会将这里的问题继续带入到后续的CL中，并继续发起新的Code review。所以尽快发送设计上的review意见一定程度上会避免、减少这类问题的发生，避免后续review重复解决同一类问题。
- 主要的设计变更，比其他小修小补，花费的时间更多，每个开发者排需求都是有deadline的，及时发送review意见有助于在deadline之前，让开发者仍然由机会对设计做出调整，以保证项目进度，又能兼顾整体代码质量。

Step Three: Look through the rest of the CL in an appropriate sequence

一旦确定CL中主体代码没有明显的设计问题，就可以按照合理的顺序review剩下的部分，比如按照逻辑处理的顺序来review，reviewer可以根据逻辑处理中的过程、分支判断来review相关的代码，从而确保不遗漏每一行变更。

通常review完CL的主体部分之后，review剩下的部分就相对简单多了。有时阅读测试用例对于review代码也是有帮助的，比如，通过测试用例你能清楚地知道各个函数参数的含义，如何使用该函数，当然你可以联想到一些边界条件，带着这些问题去review CL主体代码效果也不错。

Speed of Code Reviews

Why Should Code Reviews Be Fast?

Google对于Speed的追求，更期望的是团队能够更快更好地生产一个好的产品的速度，而不是个人开发者编码的速度，当然这并不是说开发者的开发效率就不重要。团队整体推进的速度是非常重要的，它关系到产品迭代

的进度，关系到团队的氛围，关系到每个开发者的感受，设置是他们的日常生活。

在追求速度的过程中，Code review的速度扮演者一个比较重要的角色。如果review速度很慢，可能会发生：

- **团队整体推进的速度被严重拖慢。**

  如果reviewer没有对CL进行快速的响应，可能就会耽误CL作者的其他后续工作，因为它可能要在此CL上开展其他工作。如果这里的修改是解决一个线上bug、重要的features等，如果搁置换一个几天、周、月，这种项目速度是不可接受的。

- **开发者开始抗议或者不重视Code review。**

  如果一个reviewer每隔几天才回复一次CL review意见，但是每次review意见都要求CL作者进行修改，对于CL作者这种体验是非常沮丧的，开发者可能会抱怨这样的reviewer，为什么这么review个代码这么苛刻。但是如果reviewer能够快速、及时地回复review意见，这种抱怨往往会消失了。大家在意的不是CL有没有问题，而是reviewer的怠慢、反应迟缓。所以reviewer要尽快回复review意见。

- **代码整体健康度会受到影响。**

  如果review速度过慢，就会给开发人员、reviewer人员持续带来更多的压力，这意味着开发人员会提交更多不符合标准的CLs，reviewer人员需要回复、拒绝、解释、二次review的工作做会更多。进一步，也会降低开发者代码清理、重构、优化的积极性，导致整体代码质量下降。

How Fast Should Code Reviews Be?

如果当前没有对专注度要求很高的任务的话，reviewer应该立即或者稍后进行review。

一个工作日，这个是Google内部指定的上限，在Code review发起的一个工作日内，reviewer必须进行review。

如果一个CL进行review之后，提出了修改意见，并且CL作者进行了修改，那一天之内可能要进行多轮review，尽量不要拖到第二天，跨天就不太好了。

Speed vs. Interruption

对于review速度和个人工作专注度之间需要做个权衡，如果正在进行某些对专注度要求比较高的任务，如正在写代码，这个时候还是不要让自己的工作中断。研究表明开发者被中断之后，再回到之前的工作，是需要比较长的时间的，为了追求review进度反而拖慢了个体开发进度，反过来也可能会拖慢项目整体进度。

所以建议在当前没有对专注度要求很高的时间进行review，比如你写完一段关键代码之后，或者吃完午饭、午休之后，等等，这个就要根据自己情况选择了。

Fast Responses

我们谈论Code review的速度，其实强调的是CL的review意见的回复速度，而不是CL最终通过的速度。当然，如果review意见回复速度够快，CL作者修改也会更快，CL整体通过速度也会快些。

尽管CL整体通过耗时可能比较久，维持比较快的CL review意见还是很重要，CL作者不会因为review过慢而沮丧。

如果你现在太忙了，实在没有时间对CL进行完整的review，你也可以发送一个回复信息让开发者知道你大约会在什么时间段进行review，好让他心理有底，他也可以继续安排自己接下来的工作。或者你也可以邀请其他合适的review代你进行review。这里并不是说你就要立即放下手里的编码工作立即去回复，你可以选个稍微可以放松难点的时间去回复，比如你刚刚写完一段关键代码，大脑可以稍微休息几分钟的时候。

reviewers花费足够的时间进行review是很重要的，时间充足，reviewer回复“LGTM”才更有底气。CL中每个意见的回复还是要能快就快。

Cross-Time-Zone Reviews

如果review涉及到多地的、不同时区的开发人员，reviewer最好能在CL作者还在公司工作的时候给到回复review意见，如果开发人员已经下班回家了，尽量在隔日开发人员上班前完成review，尽量不要中断、delay他人的工作。

LGTM With Comments

为了加速Code review整体通过进度，有些场景下，即便开发者没有完全完成reviewer的修改意见，reviewer可能也会给通过“LGTM” or “Approval”，这几种情况下是允许的：

- reviewer有信心，开发者会在之后不久完成提及的修改意见
- 剩余的还未修改之处，是些微小的修改，可以后面改，或者不一定要当前开发者来修改

reviewer针对上述两个情况，在LGTM的时候应该注明是哪种情况。LGTM With Comments对于多地、跨时区的开发者而言，是比较有价值的，因为reviewer最终给LGTM或者Approval可能要隔一天的时间呢，这个时候会比较久，如果是LGTM的同时附带上reviewer对开发者的一点小期望，开发者后续再修改、优化，也是可以的，毕竟这两种情况都是无伤大雅的事情。

Large CLs

如果开发者提交的一个CL非常大，以至于你不知道要从哪开始看起，这个时候可以要求开发者将这个大的CL拆分成几个小的CL，CL2 build on CL1， CL3 build on CL2... 然后再进行review。这对reviewer而言是比较有帮助的，对开发者而言可能需要额外的一点工作。

如果换一个CL无法拆分成多个小的CLs，并且你眼下也没有时间去快速地完成整体的review，可以考虑看下整体设计，回复下对整体设计的意见，开发者可以先进行适当的优化。reviewer的目标，就是激励开发者持续地对代码质量进行提升。

Code Review Improvements Over Time

持续地遵循上述的Code review流程，坚持下去，就会发现自己在review代码的时候，处理地越来越好、越来越快。开发者也能够学习到符合代码质量要求的代码应该是什么样子的，后续提交的CLs也会变得越来越合理，花费的reviewer的时间也会越来越少。reviewer也会更快地进行review意见回复，对于整体review进度delay的时间也会越来越少。

但是，但是，但是.....不要为了“快”而降低Code review标准或者破坏整体代码的质量。

Emergencies

也存在一些非常紧急的情况，这些情况下CLs可能必须非常快地通过review过程，这种情况下review时的要求可以适当放低。在应用这里的Emergencies指引之前，先了解下What is An Emergency?，区分什么是紧急场

景，什么不是，避免滥用。

How to Write Code Review Comments

Summary

- 保持友好
- 解释原因
- 权衡“给出可能的方案并指出问题”、“给出可能的方案让开发者自主决策”两种方式
- 鼓励开发者简化代码，鼓励开发者添加注释，而不是解释问题有多复杂

Courtesy

review别人代码的时候，保持礼貌、尊重是非常重要的，至少不要让开发者、贡献者感觉到明显的不适，如何做到这点呢？一个比较好的办法就是在写review意见的时候，只对code本身进行评论，不要对开发者本人进行评论，对于某些人称代词是否有必要出现，也需要斟酌下。

举个例子：

Bad：“%#@!$ 这个场景下多线程并不会有太大的性能提升，为什么**你**要用多线程？”

Good：“多线程并发增加了复杂性，但%#@!$ 场景下收到的性能提升并不明显，最好用单线程模型代替多线程。”

Explain Why

上面展示的这个good example向开发者解释了review不通过的原因，同时也解释了为什么多线程模型在这个场景 %#@!$ 下并不好，开发者就会从中学习，意识到自己的方案存在的问题，并进行优化。

当然不需要每次review的时候都进行大篇幅的解释，但是适当的解释是有必要的。

Giving Guidance

修复一个CL中存在的问题，是CL开发者的责任，而不是reviewer的责任。没有要求reviewer要代替开发者给出一个完整的详细设计，或者亲自帮助其写代码。

但是这并不意味着reviewer可以无视，不去主动帮助他人，特别是在CL开发者确实get不到reviewer的点或者束手无策的时候，reviewer可以选择性的指出问题，并给出一个直接的指引，或者给出几个备选方案让开发者去对比、选择，或者给一个简单的设计让开发者去进一步细化、完善，或者可以写一个简单的demo以供开发者参考。这也是Mentoring精神的一种发扬吧，这可以帮助开发者学习，使得Code review变得更加简单、正向、积极。最终达成的结果也往往更好。

Code review追求的第一目标就是尽可能高质量的CL，第二目标就是提升开发者的技能，这样后续的开发、Code review工作都会变得越来越简单、积极，技术传承也更加温和、有效。

Accepting Explanations

如果reviewer有段代码不太懂，并请开发者进行解释的话，最终的结果往往是开发者需要重写这段不清晰的代码。偶尔，代码中添加注释也是一种类似于“解释”的回应，但是如果代码实现太过复杂，该重写还是要重写，不能用注释解释，套逃脱应该简化、重构的工作。

代码review工具中填写的解释（对reviewer疑问的解释）对将来阅读代码的人的帮助微乎其微，这个很好理解，阅读工程代码时，别人很少会去翻之前的review意见，而要等到翻review意见的时候，意味着这里的代码可能已经需要重写了。

通过注释的方式来解释代码的行为，往往只在很少的几种场景下有效，比如在review一个自己不太熟悉的领域的代码时，如果有注释reviewer更容易理解，但是对于熟知这个领域的人而言，这些注释可能都是些常识，是多余的。

Handling Pushback in Code Reviews

Handling Pushback in Code Reviews

有时候，reviewer的意见开发者会“怼”回来，可能是开发者不同意reviewer的建议，也可能是抱怨reviewer过于苛刻。

Who is Right?

当开发者不同意reviewer的建议时，reviewer应该停下来先想想他们的想法是不是对的。通常，开发者可能比reviewer更加熟悉相关代码，可能在某些方面他们理解的比reviewer更清楚。那么，他们的争论是否有意义？从代码健康度的角度考虑，他们的修改是否有意义？如果确实有意义，让他们知道他们是对的，并且解决掉issue。

然而，开发者也不都是对的。有些情况下，reviewer应该更透彻地解释为什么他们给出的建议时正确的。一个好的解释应该能展示出开发者角度的理解、reviewer角度的理解，以及reviewer的修改建议为什么是有价值的。

某些情况下，reviewer可能要与开发者进行好几轮的沟通、解释，不管怎么样，保持礼貌的态度，应该让开发者感觉到“**你一直在倾听他们在说什么，你只是不同意他们的做法**”。

Upsetting Developers

reviewers有时会觉得自己一直坚持CL代码的优化，开发者本人会不会觉得有点沮丧。有时，开发者可能会，或者刚开始时，开发者可能会，但是当他们觉得reviewer的建议有效确实帮助他提升了代码质量的时候，他就不会觉得沮丧了。

通常，只要在review的时候保持礼貌的沟通，开发者可能根本不会感觉到沮丧，这里的担心可能只是reviewer自己头脑中的一种直觉而已。

Cleaning it Up Later

开发者针对reviewer的建议，有时会用这样的方式，“这次先review通过吧，我后面再优化下”，确实有些开发者会这么做，他们这么做的原因，无非是不想再经过一轮新的耗时的review过程。

reviewer可能会给通过，有些开发者确实会在本次CL通过后，继续写一个新的CL来解决上次reviewer提到的问题，大部分是这样的。但是也有不少开发者事后就忘了这些，并没有提交新的CL cleanup之前遗留的问题。并不是说这些开发者就是不负责任，很可能是因为他们被其他工作填满了，时间被挤占了之后人就会失忆或者“选择性”失忆，最终慢慢地就忘了cleanup。

因此，一个好的做法是，CL通过之后，reviewer应该立即通过开发者、督促开发者cleanup遗留的问题。

General Complaints About Strictness

如果之前Code review都比较松，后面Code review比较严的话，开发者刚开始肯定不适应，他们可能会抱怨多起来，这个时候，提升下Code review的速度有助于减少大家的抱怨。

对于一个团队而言，如果是Code review从松到严，大家抱怨的时间可能持续的比较久，几个月都有可能，但是最终开发者会看到严格执行Code review的好处，抗议声最高的开发者可能会变成最有力的支持者，只要他能感受到严格之上的价值。

Resolving Conflicts

如果在review代码时遵循了上述Code review的建议，但是仍然遇到了一些和开发者之间难以解决的问题，请参考 前面的章节 `The Standard of Code Review` 来浏览下相关的指引和原则，应该有助于解决遇到的问题或冲突。

The Change Author's Guide

从代码CL开发者的角度出发，介绍下Google内部积累的一些good practices。

Writing Good CL Descriptions

CL描述用来记录做了什么改变、为什么做这个改变，它会作为VCS中的提交历史记录下来，之后可能会有更多的开发者阅读到这里的CL描述。

将来，开发者也可能根据一些关键词来搜索这里的CL描述，如果CL相关的关键信息只记录在代码中，在CL描述中不能予以体现的话，那么别人定位你的CL就会异常困难。

First Line

- 需要对修改进行一个精炼的总结
- 描述应该是一个完整的句子
- 描述后跟一个空行

CL描述的首行应该对做的修改进行一个精炼的总结、概括，然后后面跟一个空行。大部分情况下，开发者查看、检索CL描述信息（log信息）是看的这些内容，所以CL描述的首行信息是至关重要的。

通常，首行信息应该是一个完整的句子，例如，一个好的首行表述："**Delete** the FizzBuzz RPC and **replace** it with the new system.", 下面这个则是一个不好的示例："**Deleting** the FizzBuzz RPC and **replacing** it with the new system."

Body is Informative

CL描述的剩余部分应该详细一点，可以包含对要解决问题的描述，以及为什么CL中的实现是比较好的或者是最优的方案。如果该方法有什么不足，也应该提一下。如果有一些bug编号、性能结果、设计文档之类的背景信息，也应该包含进来，方便后来者查看。

即使CLs涉及到的改动不多，有必要的话，也需要在body部分描述下。

Bad CL Descriptions

“Fix bug”不是一个足够充分的CL描述，修的什么bug？为了修bug做了哪些修改？其他类似的不良CL表述包括：

- Fix build
- Add patch
- Moving code from A to B
- Add convennience functions
- kill weired URLs

上面这些是Google代码仓库中捞出来的真实CL描述，作者可能认为他们的描述比较清晰了，但是实际上这样的CL描述没有任何意义，完全没有起到CL描述应有的作用。

Good CL Descriptions

下面是一些比较好的CL描述示例。

Functionality change

> rpc: remove size limit on RPC server message freelist.
>
> Servers like FizzBuzz have very large messages and would benefit from reuse. Make the freelist larger, and add a goroutine that frees the freelist entries slowly over time, so that idle servers eventually release all freelist entries.

这个CL描述的首行信息概述了做的修改，后面body部分又解释了为什么做这个修改，以及自己是怎么做的，有什么好处，还提供了实现相关的一些信息。

Refactoring

> Construct a Task with a TimeKeeper to use its TimeStr and Now methods.
>
> Add a Now method to Task, so the borglet() getter method can be removed (which was only used by OOMCandidate to call borglet’s Now method). This replaces the methods on Borglet that delegate to a TimeKeeper.
>
> Allowing Tasks to supply Now is a step toward eliminating the dependency on Borglet. Eventually, collaborators that depend on getting Now from the Task should be changed to use a TimeKeeper directly, but this has been an accommodation to refactoring in small steps.
>
> Continuing the long-range goal of refactoring the Borglet Hierarchy.

这个CL描述的首行信息指出了CL做了什么，相对于以前的实现做了哪些修改。body部分介绍了实现细节、CL的context，也指出了虽然这个方案没有那么理想，但是也指出了未来的改进方向。同时也解释了为什么需要做这里的重构。

Small CL that needs some context

> Create a Python3 build rule for status.py.
>
> This allows consumers who are already using this as in Python3 to depend on a rule that is next to the original status build rule instead of somewhere in their own tree. It encourages new consumers to use Python3 if they can, instead of Python2, and significantly simplifies some automated build file refactoring tools being worked on currently.

这个CL的首行信息描述了做了什么，body部分解释了为什么要做这里的修改，并且给reviewer提供了充分的context（领域相关知识）信息。

Review the description before submitting the CL

CLs在review过程中可能会再次修改，再次review的时候，CL描述信息可能会与最新的修改不符，在提交CL之前review一下描述信息是否与代码修改仍然一致，这个是有必要的。

Small CLs

Why Write Small CLs?

Small, simple CLs有这样的好处：

- **Review更快。**如果提交一个大的CL，reviewer可能要专门抽30分钟才能完成review过程，这个可能比较困难，但是如果CL拆的都比较小，reviewer每次抽个5分钟完成一次review，多次review，回比前者更快完成。
- **Review更充分。**如果提交一个大的CL，改动东西很多情况下，reviewer需要游走在更多代码中，心理上会倾向于关注更加重要的代码，难免某些代码会被降权，review可能不那么充分。如果CL比较小，很容易就能看懂，也不需要来回翻代码，review的会更充分。
- **不易引入bug。**因为每次改动都比较小，CL作者不容易引入bug，reviewer也更容易发现bug。
- **如果被拒绝，可减少无谓的工作。**CL可能会被拒绝，如果一次做大量修改，被拒绝的话，CL中的所有修改动作可能都要重新来过，不管是合理的、不合理的，相当于做了更多无谓的工作。
- **更易merge。**如果提交一个大的CL，涉及修改比较多，冲突可能也会比较多，那么解决冲突花费的时间会更多，不容易merge。小的CL在merge的时候就简单多了。
- **设计不至于出大问题。**如果CL涉及修改比较少，那么很容易把修改优化到最佳，对于reviewer的建议也更容易完成，如果CL比较大，reviewer意见、建议比较多，就很难一次完成了。
- **后续工作不至于阻塞在review上。**CL作者可能希望基于这个CL做更多工作，如果CL比较小那么review可以很快通过，但是如果CL比较大，那么CL作者将不得不等待更多的时间
- **merge后有问题，更易回退。**merge之后有时候也会意识到merge的代码有问题，如果要回退的话，小的CL更容易回退，如果CL很大，哇，涉及到的代码多，回退就比较痛苦、复杂。

reviewer不会因为某个CL很大，我不review了，然后给你拒绝掉，通常他们会表现的比较礼貌，告知你将这个大的CL拆分成几个小的CLs。当你已经完成了一个CL时，再将其拆分成几个小的CLs，其实这里的工作量可能不少的，当然和reviewer争辩为什么要求拆分成多个CLs花的时间可能也会很长。最省力的做法就是，一开始就坚持写小的CL，多次CLs。

What is Small?

一般，CL一般建议的大小：

- CL包含最少内容，一次只做一件事情，比如新增feature，CL只包含feature的一个部分，而不是所有的部分。也可以与reviewer进行沟通，确定合适的CL大小，不同的reviewer习惯也不一样，对reviewer而言，CL太大太小都是负担。我们的建议是CL一次只做一件事情。
- reviewer需要看到的任何东西都已经包含在了CL中，这些信息可能位于CL代码中、描述中、已经存在的代码中、reviewer之前review过的CL中。
- 当把这个CL合入之后，系统仍然能够工作的很好。
- CL也不小到隐晦难懂的程度。比如新增了一个API，应该同时包含API的使用示例，这样reviewer能够更好地理解API的使用方式，这样也可以避免引入没有用的API。

也没有硬性的规定指明CL多大才算大，一般100行代码是一个比较合理的CL尺寸，1000行有点太大了，主要还是要看reviewer的态度。CL中涉及到的文件的数量也是CL大小的一个度量维度，如果CL中包含了一个文件，文件修改了200行代码，这个感觉还是OK的，但是如果同样是修改了200行代码但是散落在50个文件中，那CL有点太大了。

体会一下，我们写代码的时候是有了解对应的背景的，但是reviewer可能了解比较少或者完全不知道。一个可接受的CL大小，对reviewer而言是很重要的。如果你不确定reviewer期待的CL大小是怎样的，那就尽量写一个比自己预期中的合理大小更小点的，很少有reviewer会嫌弃CL太小。

When are Large CLs OK?

下面这些场景，如果CL比较大也还OK，不算坏：

- 删除整个文件的内容，或者删除大段内容，可以看做是一行修改，因为它不会花费reviewer太多时间review。
- 有时一个大的CL是有自动化refactor工具生成的，而我们信任这些工具，reviewer的工作就是检查并确认这里的修改没问题。这样的CL很大，但是对reviewer而言仍然不会花太多时间。

Splitting by Files

另一种拆分大的CL的方式是，将修改的文件进行适当分组，并将不同分组的文件做成CL并提交不同的reviewer进行review。

例如：你发送了一个CL修改，同时包含了对protocol buffer文件的修改，另一个CL是业务代码修改但是使用了这里修改后的protocol buffer文件。首先我们要先提交proto文件对应的CL，然后再提交引用它的业务代码CL。虽然提交的时候有先后，但是review过程可以是同时进行的。这么做的同时，最好也知会下reviewer另一个CL中的存在以及与当前CL的关系。

Separate Out Refactorings

一般将CL中的重构之类的工作和其中包含的feature开发、bug修改区分开是比较好的做法。例如，将CL中包含的移动、重命名类名之类的操作单独放在一个CL中，这样reviewer能够更容易理解每个CL中的修改。

不过，一个小的变量名修改的也可以包含在另一个feature change或者bugfix相关的CL中。如果一个CL中包含了一些重构之类的修改，让开发者、reviewer判断这部分重构相关的修改是否对当前CL来说太大了，太大了会让review变得更加困难，开发者应该据此作出一些调整。

Keep related test code in the same CL

避免将相关的测试代码单独放在另一个CL中。测试代码验证代码修改的正确性，所以和代码修改相关的测试代码，应该放置在一个相同的CL中，尽管测试代码增加了修改代码的行数。

不相关的测试代码修改，可以放到另一个CL中，类似于 `Separate Out Refactorings`， 包含：

- 验证已经存在的、提交过的代码，当前只是新增、完善测试代码；
- 重构测试代码（如引入了新的helper函数）
- 引入了更大的测试框架（如集成测试）

Don't Break the Build

如果多个CLs互相依赖，需要找一个办法来确保这几个CL每提交一个，系统整体仍然能够正常工作，不能因为提交了一个CL就导致系统构建失败或者工作不正常。比如，可以考虑将几个依赖的小的CL修改合并。

Can't Make it Small Enough

有时会遇到这种情况，看上去CL会无可避免地变得很大，其实，这种情况几乎是不存在的。只要开发者尝试练习去写小的CLs多次code review，习惯了之后开发者总能找到合适的办法来将一个大的修改分解成几个小的修改。

在开发者动手进行一个很大的CL之前，开发者应考虑下是否先来一次只包含重构（只修改设计，不变动功能）的CL然后在此基础上再做修改，这样是不是会更好一点。如果一个CL中既包含重构代码，又包含逻辑变动代码，这个修改就很重。多数情况下，先来一次只包含重构的CL会为后续的修改扫清障碍，后续的修改会更clean。

如果因为某些客观原因，无法做到上述这些，那就先提前向reviewer说明情况，让reviewer做好心理准备，准备好review一个大的CL，review可能花费比较长的时间，请务必小心引入bug、务必添加好测试用例并通过测试，reviewer在这么大的CL中review的效果可能会大打折扣。

How to Handle Reviewer Comments

当开发者针对CL发起Code review时，reviewer一般会通过comments写一些意见、建议，那开发者该如何处理reviewer comments呢?

Don't Take it Personally

Code review的初衷是为了维护代码的质量、产品的质量，当一个reviewer对开发者代码写了些意见时，开发者应该将其看作是来自reviewer的帮助，不要将其看做是对开发者个人、个人能力的人身攻击。

有时，reviewers会变得很沮丧，并且可能将沮丧、不满情绪在评论中体现出来。一个好的reviewer是不会这么做的，但是作为一个好的开发者，应该做好面对这种情况的准备。开发者可以反问下自己，当reviewer的评论到底是在描述一个什么代码问题，然后尝试去修改就可以了。

对于别人的 Review 意见，永远不要用愤怒去回应！ 这违反了 Code Review 或者开源协同的精神，并且这样的负面信息可能会永远留存在 Review 历史中。如果你心里面很愤怒，并且想愤怒地做出回应，请尝试离开你的电脑、键盘，或者先干点别的，直到你内心平静下来再礼貌地做出回应。