目录

Google代码实践的一些感悟

自认为是一枚资深码农，对代码review早已驾轻就熟。读完之后，仍旧受益匪浅。受到原文中“希望其他组织也能从中受益”这句话的感召，我决定将其翻译成中文，托管到Github上，与大家一起分享。

原文标题为工程实践文档，表达更直接点，就是代码review规约。全文分两部分，一部分是针对代码审核人的指南，另一部分是针对代码提交人的指南，两部分文档交相呼应。

这份文档讲述了代码review的标准是什么，详细列举了在代码review过程中常见的问题，如何避免或解决这些问题，同时讲解了这么做的原因以及不这么做可能产生的后果。

文档列举了不少正面与反面的例子，可操作性很强。相信企业文化接近的朋友在读了本文之后，会有很深的认同感。在文档中提到、现实中做得不够好的地方，能很快掌握，并能把它融进自己所在的团队中，提升整个团队的开发效率。企业文化差距比较大的朋友读了之后，即使管理机制与此不兼容，在个人的层面也有很多可以改进的地方。从事非软件开发行业的朋友读了之后，相信也能有所感悟。文中提到的一些软技能，如情绪管理、沟通技巧等，在日常生活与工作中也非常有用。

回顾自己过去十几年review代码与被review代码的经历，有些事项一直都这么做，我知道为什么应该这么做；有些事项一直都这么做，至于为什么这么做我颇有微词；也有事项直到今天才知道原来这样做更好。

荀子在《劝学》中提到：不积跬步，无以至千里。抗战时期“积小胜为大胜”的战略思想，也是同样的道理。在本文中，鼓励 **每次都做一点微小的改进，多次微小的改进比一次大的改进容易得多**。这与Scrum把大User Story拆分成小User Story的观点相吻合。因为Points比较大的User Story，往往因为太大而无从下手，时间上也不可控。在进行Scrum指导时，我常对团队说“我们一个迭代周期只需改进两三个重要的问题，不必一次到位。习惯的力量太强大，需要逐步改进、适应，再改进，直到我们需要改进的措施成为我们的习惯。”

原文提到一条看似要求很低的原则：**在修改时，至少不要恶化代码**。不要恶化代码是底线，一旦发生微小的恶化，破窗效应开始起作用，代码的健康状况会飞流直下。制定过高的标准，容易让人受挫；而合理的标准配以可实操的步骤，更容易达到目标，建立自信心。

我见过这样的主管，一次性提出很多要求，让下属立即（或很快）按照要求实施，而他要求的这些改变与团队当前的环境、习惯、认知存在很大的差距。立即实施的要求看似简单，但缺乏行之有效的措施，忽视了客观环境，忽视了突然改变所付出的代价与造成的负面影响，注定无法长期坚持下去。道理很简单，反面教材在生活中的确很常见，如上海严格的垃圾分类政策。

关于尊重，在彼此尊重的环境里，大家心态比较放松，愿意以积极的心态去避免或解决问题。因为彼此的尊重，大家更愿意通过沟通、讲理的方式去解决问题，而非依职位定高下。在这方面，不少企业还有很大的提升空间。

提到沟通，本文提到了一些沟通与冲突解决的技巧。如，当审核者与开发者有不同观点时，文档对审核者说，“开发者对代码更熟悉，先考虑对方的看法是否是正确的。”对开发者说，“先思考审核者提供的反馈中是否存在有价值的部分。”很有趣，在冲突解决时，都是先审视自身，对方是否是对的。如果每个人都以这种方式处事，相信全人类离世界和平的愿望又接近了一步。

世事并不完美。在代码审核过程中，“违反专业礼仪时间一件很严重的事情。我们可以保证自己遵守，但永远没法保证他人不违反。”当他人违反了专业礼仪，我们该怎么做呢？本文也提出了解决方案。

还有一点很重要，“**不要在代码审核中带着情绪回复评论。**”为人处世也是如此，在情绪激动（或紧张）时，人很难理智地思考，此时做出的决定往往会让自己后悔。不如先采取措施让自己冷静下来，之后再做决定。

以上是我的个人感悟，有兴趣的朋友读读原文吧。

Frank
2019.10.18

Google的工程实践文档

Google有很多优秀的工程实践，这些实践遍布公司内的所有项目，覆盖了几乎所有编程语言。 随着开发项目的增多，我们不断总结经验，把这些最佳实践以文档的形式整理出来。这份文档是我们集体经验的结晶。除了我们之外，相信其他公司、组织或开源项目也能从中受益，现在时机成熟了，我们决定将其公开发布。

这份工程实践文档包含如下内容：

- Google的代码审核指南 (Google's Code Review Guidelines)，它包含两部分：
  - 代码审核者指南 (The Code Reviewer's Guide)
  - 代码提交者指南 (The Change Author's Guide)

术语

在文档中，我们用到了一些 Google 内部术语，为避免误解，我们稍作解释：

- **CL**： 即“changelist”， 中文可以翻译成修改列表，它是提交到版本控制工具中的一次代码修改（即将审核的代码）。有的公司或组织称它为 “改变”(change)或“补丁”(patch)。
- **LGTM**： “Looks Good to Me.” 的缩写，“看起来不错”。 当一个审核者这么说的时候，意味着他会批准这个CL。
- **g3doc**： Google内部的工程文档平台。

License

英文原文来自 Google's Engineering Practices documentation，中文版由 zijinshi 翻译整理。根据中文表达习惯，在原文基础上有少量删改。

版本	日期	说明
1.0	2019.10.07	初版完成
1.1	2019.10.18	修复某些翻译不准确的地方

中文版同时发布于网站：

PDF版本下载：

- Google的工程实践文档

Google代码实践的一些感悟

代码审核指南

介绍

开发者写完代码后，让其他人来检查这些代码，这个过程称为代码审核（译者注：也译为“代码评审”）。

在 Google，我们通过代码审核来保证代码的质量，进而保证产品的质量。

此文档是 Google 代码审核的规范说明流程和规范。

本节对代码审核过程作简要介绍，后面的两篇文档会做更详细的说明：

- 怎样做代码审核 ：针对代码审核者的指南。
- 开发者指南 ：针对 CL 提交者的指南。

代码审核者应该看什么？

代码审核者应该关注以下事项：

- **设计**：代码是否设计良好？这种设计是否适合当前系统？
- **功能**：代码实现的行为与作者的期望是否相符？代码实现的交互界面是否对用户友好？
- **复杂性**：代码可以更简单吗？如果将来有其他开发者使用这段代码，他能很快理解吗？
- **测试**：这段代码是否有正确的、设计良好的自动化测试？
- **命名**：在为变量、类名、方法等命名时，开发者使用的名称是否清晰易懂？
- **注释**：所有的注释是否都一目了然？
- **代码样式**：所有的代码是否都遵循代码样式指南？
- **文档**：开发者是否同时更新了相关文档？

详情可参见文档：怎样做代码审核 。

挑选最好的代码审核者

一般来讲，你一定会找*最好*的代码审核者来帮你审核代码，这个人应该在你期望的时间内有能力对审核工作负责。

如果若干人能对你的代码给出正确的反馈意见，那么最好的审核者就是这些人中最有见地的那个。他可能是你正在修改的代码的原始作者，也可能不是。有时候，一个代码审核者无法覆盖整个 CL，他们只能审核其中一部分，这种情况就需要把 CL 发给多个人（这并不意味着当一个审核者能覆盖所有代码时，就只需要一个审核者），以确保能覆盖所有代码。

如果最理想的代码审核者无法帮你审核，至少应该抄送给他（或者把他加到可选的审核者名单里去）。

面对面审核

如果你正在与一个人结对编程，你的伙伴已经对代码做过细致审核，那么这段代码可以认为是审核通过的。

你还可以与代码审核者进行面对面审核。当有疑问时，审核者提问，开发者回答。

参考

- 怎样做代码审核：针对代码审核者的指南。
- 开发者指南：针对 CL 提交者的指南。

怎样做代码审核

本文档包含若干章节。在代码审核的长期实践中，我们总结出了最佳实践，并在此基础上整理出了这些建议。整篇文档各部分的衔接性并不大，在阅读时，你可以选取自己感兴趣的部分，而不必按顺序阅读全文。当然，如果通读全篇的话，你会发现这篇文档对你非常有用。

- 代码审核的标准 （The Standard of Code Review）
- 代码审核过程中要看些什么？ （What to Look For In a Code Review）
- 代码审核的步骤 （Navigating a CL in Review）
- 代码审核的速度 （Speed of Code Reviews）
- 怎样写代码审核的评论？ （How to Write Code Review Comments）
- 代码评论被拒绝时，应如何处理？ （Handling Pushback in Code Reviews）

本文档是针对代码审核者的指南，至于代码提交者，可以参见这篇文档： 代码提交者指南 （CL Author's Guide）。

代码审核的标准

标准

代码审核的目的是为了保证代码库中的代码质量持续改进，代码审核的工具和流程都是为了实现这个目的而设计。

为了达到目标，我们需要权衡得失。

首先，开发人员必须能在任务上 *取得进展* 。如果从没向代码库提交代码，那么代码库就不会改善。同时，如果审核者让开发者在提交代码时变得很困难，那么开发者不得不花费大量的精力解决审核评论，没有动力在未来的提交中改进代码质量。

另一方面，审核者有责任确保提交者的代码质量。随着时间的推移，代码库的质量不会降低。这有点棘手，冰手三尺非一日之寒，代码库质量的降低是随着每次代码提交的微小降低累积而成的，尤其当团队面临很大的时间压力时，为了完成任务，他们不得不采取一些临时方案。

另外，代码审核者对他们审核的代码有所有权和责任，他们有义务确保代码库是一致的、可维护的，所有这些内容可参见代码审核过程中要看些什么？（What to Look For In a Code Review）这篇文章。

因此，我们希望在代码审核中能遵循这条原则：

一般情况下，如果代码提交者的代码能显著提高代码库的质量，那么审核者就应该批准它，尽管它并不完美。

这是代码评审中所有规则的 *最高原则*。

当然，也有例外。例如，一次提交包含了系统中不应加入的功能，那么审核者就不应批准它，即使它设计得非常完美。

还有一个关键点，那就是世上根本就没有“完美”的代码——只有 *更好* 的代码。审核者不应该要求代码提交者在每个细节都写得很完美。审核者应该做好修改时间与修改重要性之间的平衡。无需追求完美，而应寻求 *持续的改进* 。 倘若一个 CL 能够改进系统的可维护性、可读性，那么它不应该仅仅因为不够完美而延迟数天（甚至数周）才批准提交。

我们应该营造这种氛围：当审核者发现某些事情有更好的方案时，他可以无拘束地提出来。如果这个更好的方案并不是非改不可，可以在注释前加上：“Nit:”，让提交者明白，这段评论只是锦上添花，你可以选择忽略。

注意：在提交代码时不应显著地 *恶化* 代码质量，唯一的例外是 紧急情况。

指导

代码审查还有一项重要的功能：能让开发者学到新知识，可能是编程语言方面的，也可能是框架方面的，或一些常规的软件设计原则。作为审核者，如果你认为某些评论有助于开发者学到新知识，那就毫不犹豫地写下来吧。分享知识是提高代码质量的一种方式。记住，如果你的评论是纯学习相关的，与文档中提及的标准关系不大，那就最好在前面加上“Nit”，否则就意味着开发者必须在 CL 中修正这个问题。

原则

- 以技术因素与数据为准，而非个人喜好。
- 在代码样式上，遵从代码样式指南的权威。任何与样式指南不一致的观点（如空格）都是个人偏好。所有代码都应与其保持一致。如果某项代码样式在文档中并未提及，那就接受作者的样式。
- **任何涉及软件设计的问题，都不应由个人喜好来决定。** 它应取决于基本设计原则，以这些原则为基础进行权衡，而不是简单的个人看法。当有多种可行方案时，如果作者能证明（以数据或公认的软件工程原理为依据）这些方案基本差不多，那就接受作者的选项；否则，应由标准的软件设计原则为准。
- 如果没有可用的规则，那么审核者应该让作者与当前代码库保持一致，至少不会恶化代码系统的质量。

冲突解决

在代码审核中碰到冲突时，首先要做的永远是先尝试让双方（开发者和审核者）在两份文档（开发者指南 和 审核者指南）的基础上达成共识。

当很难达成共识时，审核者与开发者最好开一个面对面的会议（或视频会议），而不要继续通过代码审核评论进行解决。（在开会讨论之后，一定要在评论中记录讨论的结果，以便让以后阅读的人知道前因后果。）

如果仍旧无法解决问题，最好的方式是升级。常见的升级方式比较多，如让更多的人（如团队的其他人）参与讨论，把团队领导卷进来，征询代码维护人员的意见，让工程经理来决定。 **千万不要因为开发者与审核者无法达成一致而让CL停留在阻塞状态。**

下一章：在代码审核过程中要看些什么？

在代码审核过程中要看些什么？

注意：在考虑下面这些要素时，时刻谨记要遵循代码审核标准。

设计

审核一个 CL 最重要的事情就是考虑它的整体设计。CL 中的代码交互是否有意义？这段代码应该放到代码库（codebase）里，还是库（library）里？它能很好地与系统其他部分集成吗？现在加入这个功能是时机正好吗？

功能

这个 CL 所实现的功能与开发者期望开发的功能是一致的吗？开发者的意图是否对代码的“用户”有好处？此处提到的“用户”通常包含最终用户（使用这些开发出来的功能的用户）和开发者（以后可能会“使用”这些代码的开发者）。

绝大多数情况，我们期望开发者在提交 CL 进行审核之前，已经做过充分的测试。但作为审核者，在审核代码时仍要考虑边界情况、并发问题等等。确保消灭那些通过阅读代码就能发现的缺陷。

作为审核者，你 *可以* 根据需要亲自验证 CL 的功能，尤其是当这个 CL 的行为影响用户交互时，如**UI改变**。仅通过阅读代码，你很难理解有哪些改变，对用户有哪些影响。对于这种修改，可以让开发者演示这个功能。当然，如果方便把 CL 的代码集成到你的开发环境，你也可以自己亲自尝试。

在代码审核过程中，对功能的考虑还包含一种重要场景：CL 中包含一些“并行计算”，可能会带来死锁或竞争条件。运行代码一般很难发现这类问题，通常需要（开发者和审核者）仔细考虑，以确保不会引入新的问题。（这也是不要引入并发模型的一个好理由，因为它可能引入死锁或竞争条件，同时也增加了代码评审和代码理解的难度。）

复杂性

是不是 CL 可以不必这么复杂？在 CL 的每个层次上检查——哪一行或哪几行是不是太复杂了？功能是否太复杂了？类（class）是否太复杂了？“太复杂”的定义是**代码阅读者不易快速理解。** 同时意味着**以后其他开发者调用或修改它**

时，很容易引入新的缺陷。

另一种类型的复杂是**过度工程化**（也称为过度设计）。开发者在设计代码时太过于在意它的通用性，或在系统中加入了目前不需要的功能。审核者应该特别警惕过度工程化。鼓励开发者解决 *当前* 应该解决的问题，而不是开发者推测将来 *可能* 需要解决的问题。将来的问题，等碰到的时候，你才能看到它的实际需求和具体情况，到那时再解决也不迟。

测试

同时要求开发者提供 CL 对应的单元测试、集成测试或端到端的测试。测试代码与开发代码应放到同一个 CL 中，除非碰到紧急情况。

确保 CL 中的测试是正确的、明智的、有用的。测试代码并不是用来测试其自身，我们很少为测试代码写测试代码——这就要求我们确保测试代码是正确的。

当代码出问题时，是否测试会运行失败？如果代码改变了，是否会产生误报？是否每个测试都使用了简单有用的断言？不同的测试方式是否做了合适的拆分？

谨记：测试代码也是需要维护的代码。不要因为不会编译打包到最终的产品中，就接受复杂的测试代码。

命名

开发者是否为所有的元素（如类、变量、方法等）选取了一个好名称。一个好名称应该足够长，足以明确地描述它是什么，他能做什么，但是也不要长到难以阅读。

注释

开发者是否使用英文写了清晰的注释？是否所有的注释都是必须的？通常当注释**解释为什么**这些代码应该存在时，它才是必须的，而不是解释这些代码做**什么**。如果代码逻辑不清晰，让人看不懂，那么应该重写，让它变得更简单。当然，也有例外（例如，正则表达式和复杂的算法通常需要注释来说明），但大部分注释应该提供代码本身没有提供的信息，如这么做背后的原因是什么。

有时候也应该看一下这个 CL 相关的历史注释。例如，以前写的TODO，现在可以删掉了；某段代码修改了，其注释也应随之修改。

注意，注释与类、模块、功能的 *文档* 是不同的，这类文档应该描述代码的功能，怎样被调用，以及被调用时它的行为是什么。

代码样式

在Google，我们所有的主要编程语言都要遵循代码样式指南，确保 CL 遵守代码样式指南中的建议。

如果发现某些样式在代码样式指南中并未提及，在注释中加上“Nit”，让开发者知道，这是一个小瑕疵，他可以按照你的建议去做，但这不是必须的。不要因为个人的样式偏好而导致 CL 延迟提交。

作者在提交 CL 时，代码中不应包含较大的样式改变。因为这样很难比较出 CL 中有哪些代码修改，其后的代码合并、回滚会变得更困难，容易产生问题。如果作者想重新格式化文件，应该把代码格式化作为单独的 CL 先提交，之后再提交包含功能的 CL。

文档

如果 CL 修改了编译、测试、交互、发布的方式，那么应检查下相关的文档是否也更新了，如 README 文件、g3doc 页面，或其他所有生成的参考文档。如果 CL 删除或弃用（deprecate）了一些代码，考虑是否也应删除相应的文档。如果没有这些文档，让开发者（ CL 提交者）提供。

每行代码

在评审代码时，仔细检查 *每行* 代码。某些文件，如数据文件、生成的代码或较大的数据结构，可以一扫而过。但是人写的代码，如类、功能或代码块不能一目十行，我们不应假设它是正确的。有些代码得尤其小心——这需要你自己权衡——至少你应该确认你 *理解* 这些代码在做什么。

如果代码很难读懂，那就放慢审核速度，告诉开发者你没读懂代码，让他解释与澄清，之后继续审核。在Google，我们雇佣都是伟大的工程师，你是其中一员。如果你读不懂代码，很有可能其他工程师也不懂。实际上，这么做也是在

帮助以后的工程师，当他读到这段代码时更容易理解代码。所以，让开发者解释清楚。

如果你理解这些代码，但是感觉自己不够资格审核这它，确保找到一个够资格的人来审核，尤其是比较复杂的问题，如安全、并发、可访问性、国际化等等。

上下文

把 CL 放到一个更广的上下文中来看，通常很有用。在审核工具中，我们往往只能看到开发者修改的那部分代码。更多时候从整个文件的角度来读代码才有意义。例如，有时候你只看到添加了几行代码，但从整个文件来看，你发现这4行代码添加到了一个50行的方法中。在增加之后，需要把它拆分成更小的方法。

把 CL 放到系统的上下文中来考虑也很有用。CL 能提升系统的代码健康状况，还是让系统变得更复杂、更难测试？**不要接受恶化系统健康状况的代码。**大多数系统变得很复杂都是由每个细小的复杂累积而成的，在提交每个 CL 时都应避免让代码变得复杂。

好东西

如果在 CL 中看到一些比较好的方面，告诉开发者，尤其是当你在审核代码时添加了评论，他在回复你的评论，尝试向你解释的时候。审核者往往只关注代码中的错误，他们也应该对开发者的优秀实践表示鼓励和感谢。有时候，告诉开发者他们在哪些方面做得很好，比告诉他们在哪些方面做得不足更有价值。

总结

在进行代码评审时，应确保如下几点：

- 代码是否设计良好。
- 功能是否对代码的用户有用。
- 所有的UI改变都是明智的，看起来很不错。
- 所有的并行计算都很安全。
- 代码尽量简单。
- 开发这没有开发现在不需要，但是他们认为将来 *可能* 会用到的功能。
- 代码有合适的单元测试。

- 测试设计良好。
- 开发者是否使用了清晰的命名。
- 注释是否清晰、有用，大多数应该解释 *为什么* ，而不是 *什么* 。
- 代码是否包含合适的文档（通常是 g3doc ）。
- 代码是否遵循代码样式指南。

确保审核了**每行代码**，要查看上下文，确保你**正在提升代码质量**，当开发者的 CL 中包含 **好东西** 时，称赞他们。

下一章：代码审核的步骤

代码审核的步骤

概要

到目前为止，你已经知道在代码审核过程中要看些什么？。一个 CL 往往包含多个文件，怎样做会让代码审核变得高效呢？三步就可以做到：

1. 全面了解 CL。这个 CL 是否有意义？它是否包含好的描述？
2. 综观整个 CL 中最重要的部分。从整体来看，设计是否合理？
3. 以合适的顺序检查CL的其他部分。

第一步：全面了解 CL

阅读 CL 描述，了解CL实现的功能。判断这个修改是否有意义？如果答案是“否”，请立即回复，并解释为什么要取消这个修改。当拒绝的同时，你最好向开发者给出建议，这种情况应该怎么做。

例如，你可能会这么说：“这个 CL 的代码看起来挺不错的，谢谢你！不过，我们正在删除 FooWidget 系统，并用新的系统代替他，目前最好不要修改它（或对它加入新的功能）。建议换种方式，你看重构一下 BarWdidget 类，怎么样？”

在上面的例子里，你拒绝了这个 CL ，并提供了一种可选方案。整个过程，你都 *很有礼貌* 。这种礼貌非常重要，这在告诉对方：尽管不同意你的观点，但我很尊重你。

如果这种情况（开发者提交了你认为不应该这么做的 CL）经常出现，那么你应该考虑一下，是不是应该优化团队的开发流程或外部贡献者（针对某些与外部开发人员共同协作的场景）的发布流程，与开发者先进行充分的沟通确保他已经理解开发的内容，再进行开发。最好在开发者开发一大堆工作之前就说“不”，以避免大量不必要的返工。

第二步：检查 CL 的主体部分

找到包含 CL “主体”部分的文件。通常，如果一个文件包含大量的逻辑修改，那么它就是 CL 的主体部分。先审视这些主体部分有助于为其他部分理出上下文。如果 CL 太大，很难找到主体部分的位置，可以征询开发者的建议，你应该先看哪些部分，并建议他把一个 CL 拆分成多个。

如果发现 CL 中有一些重要的设计问题，立即给出反馈，即使现在还没来得及审核其他部分。实际上，审核其他部分很有可能是浪费时间。只要这个设计问题足够大，在重新设计时，其他代码很有可能会消失或变得无关紧要了。

为什么要立即对这重要设计问题给出反馈呢？有两个原因：

- 开发者经常在发出 CL 之后就立即基于这个 CL 开始新的工作。如果你发现正在审核的 CL 有重要设计问题，那么他正在做的新 CL 还得返工。我们应该及时指出，避免开发者在基于错误的设计下做了太多工作。
- 重要设计错误比小修改花费更多的时间。每个开发者在进行开发工作时都有最后期限；为了在保证代码质量的前提下按时交付，开发者需要尽快重新设计 CL。

第三步：以合适的顺序审视 CL 的其他部分

在确认 CL 没有重要设计问题之后，整理出审视文件的顺序，并确保不会遗漏任何文件。通常，在审视了主要文件之后，最简单的方式就是按照代码审核工具呈现出来的顺序遍历每个文件。有时候，先阅读测试代码更有帮助，因为看了测试代码之后，你就明白这个 CL 的期望行为是什么。

下一章：代码审核的速度

代码审核的速度

什么应该尽快审核代码

在Google，我们优化了团队开发产品的的速度，而不是优化单个开发人员写代码的速度。单个开发人员的开发速度固然重要，但远没有整个团队的开发速度 *重要* 。

当代码审核者很慢的时候，会发生以下几件事：

- **作为整体，团队的进度降低了。** 是的，单个审核者没有对代码审核及时响应，而是在完成其他的工作。如果每个人都这样的话，团队的新功能开发或缺陷修复就会延期，累积下来，延期可能是几天、几周，甚至几个月，团队中每个人都在等待别人审核（或再次审核）自己的代码。
- **开发者开始抗议代码审核流程。** 如果审核者几天反馈一次，每次都要求 CL 重大改变，这样开发者就会变得很沮丧，很困惑。通常，这表达为对审核者太过“严格”的抱怨。如果审核者 *同样* 要求大量的修改（的确有利于改善代码质量的修改），并且每当开发者更新后，审核者 *迅速* 响应，抱怨就会消失。 **大多数关于代码审核流程的抱怨实际上可以通过让流程变得更快来解决。**
- **影响代码质量。** 当审核很慢时，会增加开发者的压力，他会认为自己的代码不尽人意。迟缓的审核也会阻碍代码清理、重构，阻碍已有 CL 的进一步改善。

代码审核应该多迅速？

如果你现在没有进行一项需要集中精力的任务，**你应该在收到审核邀请时，短时间内就开始代码审核。**

在收到审核请求时，**一个工作日是审核响应的最长期限** ，即第二天早上做的第一件事情。

遵循这些规则意味着一个典型的 CL 的几轮审核（如果有必要的话）都会在一天内完成。

速度 vs. 中断

当然也有列外。**如果你正在做一项需要集中精力的任务时，例如写代码时，不要打断自己。** 研究显示，在被打断之后，开发者很长时间才能进入打断前的状态。所以，相比让另外一位开发者稍候，在写代码的时候打断自己，*成本更高* 。

什么时候审核呢？在下一个工作断点之后再审核。这个断点可能是你当前的代码已经完成的时候、午饭后、某个会议结束、从公司的餐厅回来，等等。

快速响应

当我们谈及代码审核的速度时，我们指的是 *响应时间* ，而不是审核 CL 整个流程走下来直到提交代码所花费的时间。整个流程也应该快，但 **更重要的是 *个人的快速响应* ，而不是整个流程快点完成。**

即使整个 *审核* 流程走下来会花费很多时间，但在整个过程中，审核者都在快速响应，这也会很大程度减轻开发者对 “慢” 代码审核的挫败感。

当收到一个 CL 审核的时候，如果你当时太忙没有时间做完整的审核，你仍然可以快速响应，告知开发者你稍后会做审核（但是当时没时间），他可以让他其他能快速响应的人先审核，或者先提供一些初步的反馈。（注意：这并不意味着你可以打断当前的编码工作，还是应该在工作的断点后再审核。）

有一点很重要，当审核者反馈“LGTM”时，意味着他花了足够的时间审核代码，并且认为代码满足代码标准。 然而，每个人还是应该迅速响应。

跨时区的审核

当有时差时，尽量在开发者离开办公室之前给他反馈。如果对方已经下班回家，尽量确保在他在第二天早上来公司之前给出反馈。

LGTM的评论

为了加快代码审核，有一些确定的场景，你应该给出 LGTM/赞同 的反馈，即使开发者仍有一些未处理的反馈（unresolved comments）。这些场景如下（满足任一场景即可）：

- 审核者相信开发者会对所有未处理的反馈做出合适的响应。
- 未处理的反馈无关紧要，开发者 *没必要* 处理。

审核者应该阐明他做出的 LGTM 是哪种场景。

LGTM 特别值得考虑，尤其是当开发者与审核者跨时区时，否则开发者又得等一整天时间才能得到“LGTM，赞成”。

大 CL

如果某位开发者提交一份很大的代码审核，你不大确认自己是否有时间审核它，一种典型的响应是让开发者把一个CL拆分成多个，而不是让审核者一次审核大 CL。这样对审核者比较有用，虽然开发者有些额外的工作要做，这是值得的。

如果一个 CL *不能* 拆分成多个，并且你很难在短时间内审核代码，至少在CL的整体设计上向开发者提出反馈，以便让开发者改进。作为一个审核者，你的目标之一是：尽量不要阻塞开发者，让他能迅速采取下一步行动。当然，前提是不会降低代码质量。

代码审核在不断改善中

在遵循本文中的建议进行代码审核之后，尽管代码审核很严格，你会发现，在运行一段时间后，整个流程会越来越快。开发者学会了健康的代码需要什么，在发送 CL 之前会尽量保证代码质量，因此需要审核的时间会越来越短。审核者学会了快速响应，不会在审核中增加不必要的延时。

但是， **不要为了想象中的速度提升，在代码审核标准或质量上妥协** ——实际上，从长期来，这样做并不会节省时间。

紧急情况

当然，也会有紧急情况，要求审核流程尽快完成，此时代码质量也有适当的弹性空间。但是，请先确保它的确属于紧急情况。如果不确认，先查看一下什么是紧急情况，这篇文章详细讲述了哪些情况属于紧急情况，哪些不是。

下一章: 怎样写代码审核的评论

怎样写代码审核的评论？

概述

- 保持友善。
- 解释原因。
- 给出明确的信息，指出问题所在，让开发者最后做决定。
- 鼓励开发者简化代码，给代码添加注释，而不是向你解释为什么这么复杂。

礼貌

在审核代码时，礼貌和尊重都很重要，与此同时，评论应该描述清晰，有利于开发者改进代码。确保你对代码的评论应该是针对“代码”，而不是针对“开发者”本人。当然，不必总是遵循这条原则，但是当你要说某些可能让人沮丧或引起争议的话时，一定要对事不对人。例如：

不好的说法：“为什么你在这儿使用线程？明显这儿使用并发没有任何好处。”

好的说法：“这儿的并发模型增加了系统的复杂度，我在性能上没有看到好处。因为没有性能提升，这段代码最好还是由多线程改成单线程。”

解释为什么

从上面“好的说法”中，我们看到，它有助于开发者理解 *为什么* 你要写这条评论。当然，不必每次都解释为什么，但某些情形——阐明你的意图、你正在遵循的最佳实践、你在提升代码健康程度——解释原因是有必要的。

提供指导

一般而言，修复 CL 的责任人是开发者，而不是审核者。 你不必为开发者写出详细的设计方案，也不必为他写代码。

但这不代表审核者可以不提供任何帮助。作为审核者，你应该在指出问题所在与提供直接指导之间做好平衡。指出问题，并让开发者自己做决定，这样有助于开发者自我学习，审核者自己也很省时间。这种方式，开发者也更容易找出更好的解决方案，因为相对审核者，开发者对自己的代码更熟悉。

当然，有些时候也可以给出直接的指示、建议或代码。毕竟，代码审核的首要目标是尽可能让 CL 变得更好；次要目标才是提升开发者的技能，以缩短审核时间。

接受解释

如果某段代码你看不懂，让开发者解释，通常结果是他们会**重写代码让它更清晰**。有时候，添加注释也可以，只要它不是用来解释一段过于复杂的代码。

仅仅把解释写到代码审核工具里不利于以后的代码阅读者。 只有几种情况可以这样，如当你正在审核一段你并不是很熟悉的代码时，开发者向你解释的文字，其他开发者都知道，那这种解释就不必写到代码里。

下一章：代码评论被拒绝时，应如何处理？

代码评论被拒绝时，应如何处理？

有时候，面对审核者的评论，开发者可能会拒绝。他们不同意你的建议，或者他们认为你太过于苛刻了。

谁是对的?

当开发者不同意你的建议时，先考虑他们是否正确。开发者往往更熟悉代码，在某些方面他们对代码有更好的见解。他们的论点是否理由充分？站在代码健康的角度，是否应该如此？如果回答“是”，告诉他们，他们是对的，可以忽略这条评论。

但开发者并非总是对的。此时，代码审核者应该进一步解释为何他相信自己的建议是正确的。一个很好地解释通常包含两部分：对开发人员的回复的解释和进一步说明这么改的必要性。

尤其当审核者相信他们的建议能够有效提升代码质量时，他们应该继续坚持这种修改。即使有证据显示这种代码提升需要一些额外的工作，也值得做。 **提升代码质量往往是聚沙成塔的过程。**

有时候，需要经过好几轮的解释与澄清，你的建议才会被接受。确保自始至终都保持礼貌 ，让开发者知道你在 *听* 他说话，只是不 *同意* 他的观点而已。

烦躁的开发者

有时候，审核者相信，如果自己坚持改进，那么开发者可能会心烦。这种事的确偶有发生，但通常持续时间很短，稍后他们会感谢审核者，正是他的坚持让代码质量得以提升。通常，如果在评论中保持礼貌，开发者根本就不会烦躁，审核者不必担忧。烦躁大多是因为写评论的方式没有把焦点放在代码质量上。

稍后再清理

一种常见的拒绝原因是：开发者想尽快完成它（这种心理很常见）。他们不想再进行一轮审核，只想快点把 CL 提交到代码库。所以，他们说他们会在稍后的 CL 中再清理代码，这样你应该对 *当前* CL 评论 LGTM了。有些开发者的确是这么做的，他们随后的确创建了一个新的 CL，用于修复当问题。然而，经

验告诉我们，从开发者提交了原始 CL 之后，时间过得越久，他们清理代码的可能性越小。除非开发者在当前 CL 之后 *立即* 清理代码，否则以后也不会清理。这并不是说开发者不靠谱，而是因为他们有太多的开发工作要做，迫于其他工作的压力已经忘了清理之前提交的代码这件事。因此，我们建议，最好坚持让开发者 *现在* 就开始清理代码，而不是提交到代码库 *之后* 。 我们应该有种理念， 代码退化的常见原因有几个，“稍后再清理” 便是其中之一。

如果 CL 让代码变得复杂，那么它必须在提交之前清理完毕，除非是紧急情况。如果 CL 处处是问题却不立即解决，开发者应该给自己发一个清理代码相关的 bug，把它分配给自己，以避免遗忘。除此之外，在代码中加上 TODO 注释和相关的 bug 编号。

与严格相关的常见抱怨

如果以前你在做代码评审时比较宽松，现在突然变得严格起来，有些开发者可能会抱怨。没关系，通过提升审核代码的速度通常会让抱怨消失。

有时候，让抱怨消失的过程比较漫长，长达数月，但最后，开发者会往往趋于赞同严格审核代码的价值，因为在严格审核的帮助下，他们写出了伟大的代码。一旦大家认识到这种严格带来的价值，甚至最强烈的抗议者可能会变成最强大的拥护者。

冲突解决

作为审核者，如果你已经遵循本文中提到的所有规则，但仍旧与开发者之间产生了难以解决的冲突，可以参考代码审核的标准 ，以其中提到的规则与理论来指导冲突解决。

代码开发者指南

本文包含开发者怎样让代码审核容易通过的最佳实践。在读完本指南后，相信能够让你的审核质量更高，速度更快。作为开发者，可以选择阅读自己感兴趣的部分。当然，我们还是建议你按顺序通读全文，你会发现这篇文档对你非常有用。

- 编写良好的 CL 描述
- 小 CL
- 怎样处理审核者的评论

如果你是代码审核者，可以参考怎样做代码审核。

编写良好的 CL 描述

CL 描述是一项公开的记录，其内容包含修改了 什么 与 为什么 这么修改。 虽然你的 CL 只是在你与审核者之间发生，但它是版本控制历史的一部分，若干年之后，很有可能会有成百上千的人阅读。

以后的开发者可能会根据描述搜到你以前写的 CL 。在没有精确数据的情况下，他可能根据自己的模糊记忆搜索 CL。如果所有的信息都只包含在代码里面，描述中几乎没有相关内容，那么定位到你的 CL 将会花费太多的时间。

第一行

- 简短描述做了什么。
- 完整的句子，祈使句。
- 后面空一行。

CL 描述的 第一行 应该是一句简短的描述，用以说明 *CL做了* 什么 。在第一行后面留一空行。以后有开发者搜索版本控制历史的代码时，这是他们看到的第一行，所以第一行应该提供足够的信息，他们不必阅读代码，也不必阅读整个描述，只需扫一眼便知道 CL 做什么，他们节省时间。

一般而言，CL 描述的第一行是以命令口吻（祈使句）写的一句话。举例说明，我们应该写“ 删除 FizzBuzz RPC，并用写的系统 替换 它。”，而不是写成“ 删除了 FizzBuzz RPC，并 已经 用写的系统 替换 它。” 当然，第一行写成祈使句就可以了，其他内容不必如此。 (译者注：原文中的反面例子是现在进行时。但在中文中现在进行时与祈使句基本一致，不好翻译。此处改成了现在完成时。)

描述内容要提供充分的信息

描述内容应该提供足够的信息。它可能包含一段关于问题的简短描述，为什么这是最好的解决方案。如果有更好的解决方案，也应该提及。如果有的话，相关的背景信息，如 bug 编号、基准结果和相关的设计文档也应包含在内。

即使是小的 CL，也应该包含这些信息。

糟糕的 CL 描述

“修复 bug”是一个很不恰当的描述。哪个 bug ？你做了哪些事情来修复它？通通都没有。类似糟糕的描述还包括：

- “修复编译。”
- “增加补丁。”
- “把代码从 A 移到 B。”
- “第一阶段。”
- “增加方便的功能。”
- “清除死链。”

以上这些都是我们在真实案例中见过的 CL 描述。作者可能认为他们提供了足够的信息，其实它们不符合 CL 的目的描述。

良好的 CL 描述

这是几个良好描述的样例。

功能修改

> RPC：移除 RPC 服务器的消息空闲列表的大小限制。
>
> 服务器（如 FizzBuzz）有大量的消息，可以从重用中受益。使空闲列表更大，并添加一个goroutine，随着时间的推移缓慢释放空闲列表，以便空闲 服务器最终释放所有空闲列表。

前面几句话描述了 CL 做什么的，接下来描述解决了什么问题，为什么这是一个好的解决方案，最后涉及到了一些实现细节。

重构

> 构建一个带 TimeKeeper 的 Task，以便使用它的 TimeStr 和 Now 方法。
>
> 在 Task 中增加一个 Now 方法，然后删掉 borglet() 方法（这个方法仅仅被 OOMCandidate 使用，它调用了 borglet 的 Now 方法）。这样就替换掉 Borglet的方法，把它委托给 TimeKeeper。
>
> 让 Tasks 提供 Now 是减少对 Borglet 的依赖所做的一小步。最终，从 Task 上调用 Now 的方式会替代成直接调用 TimeKeeper，我们会逐步实现。
>
> 继续重构 Borglet 层次的长期目标。

第一行描述了 CL 做什么的，以及过去它是怎么改变的。描述的其他部分谈到了具体实现、CL 的上下文，这种方法并不完美，但在朝着完美的方向前进。而且也解释了 *为什么* 应该这么改。

需要一些上下文的小 CL

> 为 status.py 创建一个 Python3 的编译。
>
> 在原始的编译（Python2）旁创建一个 Python3 的编译，让已经使用过 Python3 编译的用户根据某些规则选择 Python3 还是 Python2，而不是依赖于某个路径。 它鼓励新用户尽可能使用 Python3 而不是 Python2，并大大简化了当前正在使用的某些自动编译文件重构工具。

第一句话描述做了什么，其他部分解释 *为什么* 要这么修改，并向审核者提供了不少额外的上下文信息。

提交 CL 之前审核描述

在审核过程中，CL 可能会发生重大改变（与最初提交审核的 CL 相比）。在提交 CL 之前有必要再审视一遍 CL 描述，确保描述能够正确地反映 CL 做了什么。

下一章: 小 CL

小 CL

为什么应该写小 CL？

小 CL 有如下优点：

- 相对让审核者单独拿出30分钟审核大 CL，不如让他花费几个5分钟审核代码。对审核者而言，后者更容易。
- **审核更彻底。** 发生较大修改时，往往会反复审核、修改，审核者与开发者经常会因为过多的反复而在情绪上受到影响，以致于把精力放在修改上了，却忽略了 CL 中更重要的部分。
- **引入新缺陷的可能性更小。** 如果修改的内容比较少，自然审核人的效率会更高，开发者与审核者都更容易判断是否引入了新的缺陷。
- **如果被拒绝，浪费的时间更少。** 如果开发者花费了很大的精力开发了一个大 CL，直到审核的时候才知道整个开发的方向错了，那么之前的所有时间就全浪费了。
- **更容易合并代码。** 大 CL 在合并代码时会花费很长的时间，在合并时需要花费大量时间，而且在写 CL 期间可能不得不频繁地合并。
- **更易于设计。** 完善小 CL 的设计和修改要容易得多，多次微小的代码质量提高比一次大的设计改变更容易。
- **减少阻塞审核的可能性。** 小 CL 通常是功能独立的部分，你可能正在修改很多代码（多个小 CL），在发送一个 CL 审核时，同时可以继续修改其他的代码，并不会因为当前 CL 的审核没有完成而阻塞。
- **更容易回退。** 一个大 CL 开发的时间比较长，这意味从开发到代码提交这段期间，代码文件的变更会比较多。当回退代码时，情况会变得很复杂，因为所有中间的 CL 很有可能也需要回退。

请注意：**审核者有权因为你的 CL 太大而拒绝它。** 通常，他们会感谢你为代码做出的贡献，但是会要求你把它拆分多个小 CL。一旦写好了代码之后，要把它拆分成小 CL 通常需要花很多时间，当然，你也可能会花费大量时间与审核者争论为什么他应该接受这个大 CL 。与其如此，不如设计之初就保证 CL 尽量小。

如何定义“小”？

一般而言，一个 CL 的大小就应该是 **独立功能的修改**。这意味着：

- 一个 CL 尽量最小化，它只 **做一件事**。通常它只是一个功能的一部分，而不是整个功能。总体而言，CL 太小或太大都不好，二者取其轻的话，太小稍微好点。可以与审核者一起讨论，找出大多比较合适。
- 审核者需要理解 CL 中包含的一切（除了以后可能要开发的功能之外），包括 CL 代码、描述、已存在的代码（或之前已经审核过的相关 CL ）。
- 在 CL 代码提交之后，无论是针对用户，还是针对开发人员，系统应该仍旧运行良好。
- 如果代码难以理解，通常是以为 CL 还不够小。如果新增一个 API，同时应该同一个 CL 中附上这个 API 的使用方法，便于审核者理解如何使用，也方便以后的开发者理解。同时也可能有效防止提交的 API 无人使用。

没有直观的标准判断 CL “太大”应该符合哪些条件。 100行代码通常是一个合理的大小。1000行代码通常太大了，但也不能一概而论，这取决于审核者的判断。修改文件的个数也影响它的“大小”。在一个文件中修改200行可能没问题，如果这200行代码横跨50个文件，通常而言太大了。

记住，当你开始编写代码时，只有你最了解代码，而审核者通常不了解上下文。在你看起来很是一个合适大小的 CL，审核者可能会很困惑。毫无疑问，在写 CL 时，CL 的大小最好比自认为的还要小。审核者通常不会抱怨你的 CL 太小了。

什么时候可以有大 **CL**？

当然，也有一些例外情形，允许 CL 比较大：

- 删除一个文件与修改一行没有太大区别， 因为它不会花费审核者太多时间。
- 有时候，一个大 CL 可能是由可靠的自动代码重构工具生成的，审核者的工作主要是检查它是否做了它应该做的工作。虽然符合以上提到的注意事项（例如合并和测试），这类 CL 也可能比较大。

以文件来拆分

另外一种拆分大 CL 的方法是：对 CL 中涉及的文件进行分组，这就要求不同独立功能的修改需要相应的审核者。

例如：你提交了一个 CL，这个 CL 修改了协议缓冲区，而且另外一个 CL 用到它。因此我们先提交第一个 CL，再提交第二个 CL，并让两个 CL 同时审核。此时，你应分别向两个 CL 的审核者告知另外一个 CL 的内容，以便他们知道上下文。

以代码和配置文件进行拆分。例如，你提交了2个 CL：其中一个 CL 修改了一段代码，另外一个 CL 调用了这段代码或代码的相关配置；当需要代码回滚时，这也比较容易，因为配置或调用文件有时候推送到产品比代码修改相对容易。

单独重构

在修改功能或修复缺陷的 CL 中，不建议把重构也加进来，而是建议把它放到单独的 CL 中。例如，修改类名或把某个类移到其他包内是一个 CL，修复这个类中的某个缺陷是一个 CL，不要把它们合并到一个 CL 中。把它们拆分出来更有利于审核者理解代码的变化。

有些代码清理工作，如修改某个类中的一个变量的名称，可以把它包含在一个功能修改或缺陷修复的 CL 中。那标准是什么呢？这取决开发者与审核者的判断，这种重构是否大到让审核工作变得很困难。

把测试代码包含到对应功能的 CL 中

避免单独提交测试代码。测试代码用以验证代码功能，应该把它与代码提交到相同的 CL 中，虽然它会增大 CL 的代码行数。

然而，独立的测试修改可以放到单独的 CL 中，这与单独重构中的观点比较类似。它包含如下内容：

- 为过去提交的已存在代码创建新的测试代码。
- 重构测试代码（例如，引入 helper 函数）。
- 引入测试框架代码（如，集成测试）。

不要破坏编译

如果同时在审核的有多个 CL，并且这些 CL 之间存在依赖关系，你需要找到一种方式，确保依次提交 CL 时，保证整个系统仍旧运行良好。否则，可能在提交某个 CL 之后，让系统编译错误。此时，你的同事在更新代码后，不得不花时间查看你的 CL 历史并回退代码以确保本地编译没有问题（如果你之后的 CL 提交出了问题，可能会花费更多时间）。

无法将其变小

在某些情形下，好像你没法然 CL 变得更小，这种情况很少发生。如果开发者经常写小 CL，那么他往往都能找到一种把 CL 拆得更小的方法。

如果在写代码之前就估计这个 CL 比较大，此时应该考虑是否先提交一个代码重构的 CL，让已有的代码实现更清晰。或者，与团队其他成员讨论下，看是否有人能帮你指出，怎样在提交小 CL 的前提下实现当前功能。

如果以上所有方法都试过，还是不可行（当然，这种情况比较罕见），那就先与所有的审核者沟通一下，告知他们你将会提交一个大 CL，让他们先有心理准备。出现这种情况时，审核过程往往会比较长，同事需要写大量的测试用例。需要警惕，不要引入新的 bug。

下一章: 如何处理审核者的评论

如何处理审核者的评论

在发出 CL 之后，审核者一般会给出反馈（评论），让你修改代码。下面我们就来详细描述如何处理这些评论。

不要情绪化

代码审核的目标是保证提交到的代码库中代码的质量，进而保证产品的质量。当审核者提出一些批判性的评论时，开发者应该告诉自己，对方在尝试帮助你，保证代码库的质量，帮助 Google，而不是针对你的人身攻击或个人能力的怀疑。

有时候，审核者感到很沮丧，并在评论中表达了这种心情。其实，这不是一种正确的方式，但作为开发者，你应该有足够的心理准备来面对这种情况。问一下自己，“我能从审核者的评论中读出哪些建设性的意见？”假想他们就是以这种建设性的语气对你说的，然后按照这种建议去做。

不要在代码审核中带着情绪回复评论。 在审核代码过程中，违反专业礼仪是件很严重的事情，但我们永远没法确保别人不违反专业礼仪。我们可以保证自己不要违反它。如果你很生气或恼火，以致于无法友好地回复，那就离开电脑一会儿，或者先换一件事情做直到心态平静下来，再礼貌地回复。

一般情况下，如果审核者没有以建设性地口吻提供反馈，反馈的方式不够礼貌，可以私下与他沟通。如果不方便与他私下沟通，也不方便通过是视频电话远程沟通，可以给他单独发一封邮件。在邮件中，以友好的态度向他解释，你很难接受这种反馈方式，期待他能换一种沟通方式。如果他仍旧以一种非建设性的方式回复你，或没有达到预期效果，那就升级到他的主管吧。

修复代码

如果审核者说，他对你的代码中有些内容不理解，你的第一反应是澄清代码本身。如果代码无法澄清，加一段注释用以解释为什么这段代码这样写。如果这段注释放到代码里毫无意义，那就应该放到代码评审工作的评论中作为反馈的解释。

如果审核者无法理解某段代码，很有可能其他的代码阅读者也不懂。在代码审核工具中回复它对未来的代码阅读者没有任何好处。这种情况应该尝试清理代码，或增加一些必要的注释，以帮助他们阅读。

先思考自己是否有改进的空间

写一个 CL 会耗费大量精力。在提交一个 CL 审核时，开发者会往往会产生几乎快要完成的幻觉，自认为无需进一步修改。当审核者回复需要修改某些代码时，开发者容易条件反射地认为反馈不正确，审核者没必要阻碍自己的开发，他应该让这个 CL 审核通过 。 然而，**无论你有多确定** 这点，最好还是先退一步，仔细考虑审核者是否在反馈中提供了有价值的内容，可以帮助提高代码库的质量。你的第一个问题应该永远都是，“审核者说得对吗？”

如果无法回答这个问题，很有可能审核者需要进一步澄清评论。

如果你 *已经* 思考过，并确认你是对的，那就在回复中解释为什么你的方法比较好（相对已有的代码、用户）。通常，审核者也会提供*建议*，他们希望你比较一下哪个更好。你可能知道一些关于用户、代码库或 CL的内容，而这些正是审核者不了解的，那么就把这些写出来，提供更多的上下文。通常，你在技术上可以与审核者达成一致。

冲突解决

在解决冲突的第一步永远都是应该先尝试与审核者达成共识。如果无法达成共识，可以参考审核代码标准，当面临这种情形时，它为我们提供了一些准则。

紧急情况

有时候会有紧急的 CL。紧急情况发生时，必须尽快完成审核流程并提交。

哪些是紧急情况？

紧急 CL 应该是一个 小 修改：一个重要的发布版本需要包含某个功能（无法回滚），修复产品中严重影响用户的缺陷，处理紧迫的法律问题，关闭一个重要的安全漏洞，等等。

处于紧急情况时，我们应该关心整个代码审核流程的速度，而不仅仅是响应的速度。更准确地说，在这种情况下，审核者应该把审核速度与代码的正确性（代码是否解决了紧急问题？）放在首位。并且，当紧急情况发生时，它的审核优先级必须高于其他所有的代码审核。

当紧急情况处理完毕之后，应该回过头来再继续做一次更全面的审核。

哪些不是紧急情况？

需要明确的是，如下情形 *不是* 紧急情况：

- 希望本周完成，而不是下周（除非有一些无法避免的硬期限，如合作伙伴之间的契约）。
- 开发者为这个功能已经开发了很长一段时间，他们希望尽快提交代码。
- 所有的审核者都不在相同的时区，他们现在是半夜或下班时间。
- 现在是周五快下班的时间，如果开发者能再周末离开公司之前提交代码那就太棒了。
- 经理说这个审核必须完成，CL 在今天必须提交因今天是截止日期(软期限，而非硬期限)。
- 回滚一个造成测试失败或编译错误的 CL。

  等等。

什么是硬期限？

所谓硬期限（hard deadline），就是错过了就会有**灾难性的事情发生**。例如：

- 合同规定，你必须在某个特定日期之前提交 CL。
- 如果产品没有在某个特定日期之前发布，很有可能会影响销量，甚至在产品市场导致失败。
- 有些硬件制造上每年只会发布一次产品。如果在截止日期之前没有提交代码给他们，而这些代码又非常重要，很有可能会造成灾难性的后果。

推迟一周发布不是灾难性的。错过某次重要的会议可能是灾难性的，也可能不是。

大多数截止日期都是软期限（soft deadline），并非是不可改变的。这些软期限期望在某个时间节点得到需要的功能。它们很重要，但不应该为了达到目标而破坏代码的健康状况。

如果发布周期较长（好几个月），则很有可能会牺牲代码质量，以期在下一个周期之前实现某个功能。如果这种模式反复出现，那就为项目筑起了难以承受的技术债，这是项目开发中常见的问题。如果开发者经常在临近开发周期结束的时候提交代码，那么团队“必定”只能做表面上的代码审核。此时，团队应该修改开发流程，大型的功能修改应该在周期的早期进行，以确保有足够的时间做好代码审核。