CR实践总结

CR（下文简称CR），即代码审查，是一种通过评审代码以发现并修正错误的实践。它不是一个新概念，但在软件开发中，它的重要性毋庸置疑。

为什么要CR

- **提前发现缺陷**：在CR阶段发现的逻辑错误、业务理解偏差、性能隐患等时有发生，CR可以提前发现问题。
- **提高代码质量**：主要体现在代码健壮性、设计合理性、代码优雅性等方面，持续CR可以提升团队整体代码质量。
- **统一规范和风格**：集团编码规范自不必说，对于代码风格要不要统一，可能会有不同的看法。但代码其实不只是写给自己看的，也是写给其他人看的，代码风格的统一更有助于代码的可读性及继任者的快速上手。
- **防止架构腐烂**：架构的维护者是谁？仅靠架构师是远远不够的，需要所有成员的努力，所谓人人都是架构师。架构防腐最好前置在设计阶段，但CR作为对最终产出代码的检查，也算是最后一道关

键工序。

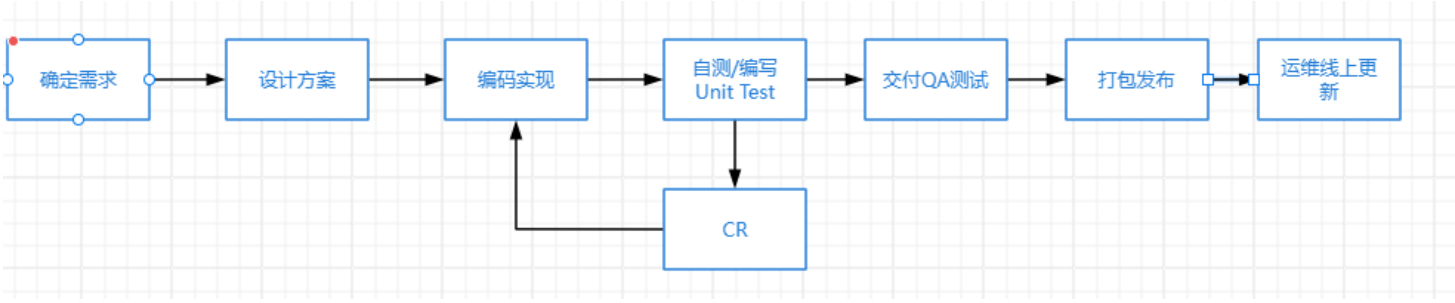
- **知识分享**：每一次CR，都是一次知识的分享，通过参与CR，团队成员可以集百家之所长，融百家之所思。同时，业务逻辑都在代码中，团队CR也是一种新人学习业务的途径。
- **团队共识**：通过多次讨论与交流，逐步达成团队共识，特别是对架构理解和设计原则的认知，在共识的基础上团队也会更有凝聚力，这在较多新人加入时尤为重要。

最佳实践

CR相关的实践指导，可以参考此文档：google CR工程实践。以下列举一些要点：

时间宜早不宜迟

一般软件开发流程图：



从软件工程的角度讲，工作流越往后，消耗的人力越多。CR中发现了问题，修改的成本也越高。并且因为临近发布，为了项目稳定发布，优化代码的阻力变得更大，开发者解决CR问题的意愿也更低。CR最合适的时机在代码自测完之后就开始，此时还没有交付测试，修改只涉及开发者自己。

我们目前大多安排在封板后甚至测试回归时才做CR，就变成发现了问题也只改严重bug。因为如果改动较大，测试需要重做功能验证和回归，成本很高。这方面有待优化。

善用工具

凡是标准规范都是比较机械化的条条框框，可以交给机器去检查（如SonarQube，Copilot等），机器静态扫描效率不仅比人高出一个数量级，而且非常严谨，不容易出错。当前各种AI工具也能提供很多有用的建议。开发者则更多地聚焦于业务逻辑，结构设计，可扩展性等方面，如此可以大幅提升review效率。

控制评审量

commit尽量拆分：小型的changelist，拥有降低评审难度、缩短评审时间、减少引入错误的可能性、易于合并等诸多好处。通常认为将changelist控制在只解决一件事（可以只是feature的一部分），视作合适的大小。我们可以按层进行水平拆分、按功能进行垂直拆分，亦或是结合两者。
一次不要评审过多的代码：一般而言，建议每次评审的代码控制在100~300行，最多不超过500行，每次评审时间不超过1.5小时（报告显示超过这些阈值会导致CR质量及效率降低）。

互相尊重原则

代码作者应该尽可能的为审核人提供配合和方便,提交高质量的代码；有清晰的 commit 历史,让人可以一目了然代码的提交内容；保持开放的心态，将评审当做自我学习和提升的过程。
代码审查者一定要懂得相互尊重，提出建议要懂得换位思考，考虑代码作者的感受，不要用主观的批评或者情绪化的语气指责团队的同事；提出代码改进建议，必须是基于事实，或者明确的代码规范文档，不基于个人喜好，更多地把精力放在代码逻辑，功能设计等问题上。
对于CR中提出的问题，审查者和提交者应及时沟通，对问题达成一致意见。

CR计入开发的工作量

很多团队不做 CR 都有一个共同的原因是觉得浪费时间，结果导致糟糕的代码合并入库，出现线上问题，然后开发人员疲于奔命的去修复线上事故。虽然短期来看功能是快速上线了，但是算上复工的时间，长期来看整体的交付周期还是被拉长了。而且糟糕的生产质量人很容易打击开发人员的持续生产高质量代码的信心。
所以将 CR 计入开发的工作量是重视长期利益的一种做法，也是 CR 能够成功落地的重要前提。从团队管理的角度来说，不计入工作量的事情就不会被重视，不被重视的话那么 CR 最终在团队只会被废弃或者流于应付形式，不能真正发挥作用。

评价和改进

设定一些度量指标，并持续追踪趋势，有助于我们持续不断改进CR过程。以下是一些可以用作度量的指标，例如审查时长、缺陷密度、CR率等。
定期或不定期通过报告/会议等形式，对CR运作状况进行总结，分析问题，解决问题，持续优化，在团队内外对CR的作用建立共识。

CR的CheckList

审查者需要对代码中的坏味道敏感一些。一般来说，这种坏味道代码可能存在以下一些特点：

- 大量复制粘贴的重复代码。
- 扩展旧逻辑导致逻辑链过长，大量的if else。
- 性能较低的代码，不合理的数据结构，层层嵌套的for循环。
- 对合理性缺少严格要求，如多个同步/异步调用的先后顺序要求。
- 并发环境下不做加锁处理，导致race condition。
- 复杂/不合理的继承关系，过度封装。
- 安全性不佳的代码，如不遵守RAII规则的设计，非常规的类型强制转换，不做长度检查就使用memcpy等。
- 缺少设计的流水线代码。
- 过早优化。
- 标新立异，使用奇技淫巧。

在进行CR时，可以参照下面提供的**CheckList**：

代码风格和规范

命名、代码样式、缩进等是否符合规范？(可参考google开源项目代码风格指南，尽量保持新旧代码风格一致)
文件目录结构，包含关系是否合理？

逻辑和功能

代码实现是否满足功能需求？
实现上有没有需求的理解偏差？
代码是否逻辑清晰、易理解？
是否有重复可简化的复杂逻辑，是否符合KISS(Keep It Simple,Stupid)和DRY(Don't Repeat Yoursel)原则？

程序设计

是否分层清晰、模块化合理、高内聚低耦合、遵从基本设计原则（不过度设计，组合大于继承等）？
是否考虑了全局设计和兼容现有业务细节？
代码交互、系统交互是否恰当？
技术、组件的使用是否恰当？

性能和优化

算法是否高效，是否存在明显的性能瓶颈？
是否有适当的缓存机制和优化措施？

代码是否正确管理了资源？如内存、文件句柄等。

程序漏洞

是否正确处理了边界条件和异常情况？
代码是否正确处理了并发和多线程问题？
输入数据是否进行了验证和处理？
是否正确处理了错误和异常？
敏感数据是否进行了加密处理？
是否正确处理了权限和访问控制？
对内存的使用是否安全？

具体项可参考我之前关于开发问题避坑的分享：游戏服务器开发避坑&&内存安全编程实践。

注释和文档

是否提供了合适的注释？(如设计取舍、业务背景、某些tricky的实现，要解释清楚为什么这么写)
系统是否提供了相关文档，如接口文档等？

日志与监控

是否需要日志,日志级别是否合理？
日志是否完整,包含所有有效信息？
敏感数据是否接入监控？

测试

是否有足够的单元测试用例，覆盖主要功能和边界条件？
是否有集成测试，确保各个模块协同工作？
是否便于做自动化测试？

团队和协作

代码提交是否符合团队的提交规范，提交信息是否清晰？
代码合并是否经过充分的测试和审查？
是否及时响应代码审查反馈，修复问题？