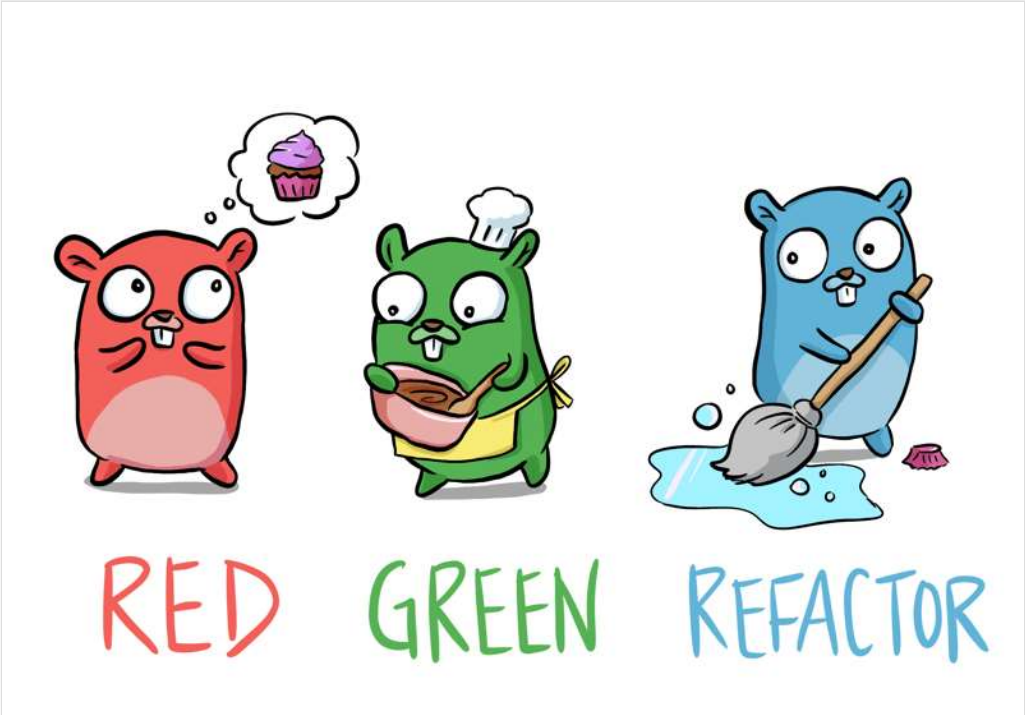
为什么要进行单元测试，以及如何让它为你工作

英文原文 / 翻译 / 951 / 1 / 发布于 1年前



为什么要进行单元测试，以及如何让它为你工作

这里是一个关于我谈论这个话题的视频链接

如果你不喜欢视频，这里有一个冗长的版本。

软件

软件是可以改变的。这就是为什么它被称为「软」件，它的可塑性是比硬件强的。一个优秀的工程师团队应该是一个公司一笔惊人的财富，他们编写可以随着业务发展而不断增值的系统。

那么，为什么我们在这方面做的如此糟糕呢？你听说过多少完全失败的项目？或者成为「遗产」，必须完全重写 (重写通常也会失败！)

软件系统是如何 “失败” 的呢？难道不能在它正确之前进行修改吗？这就是我们的承诺！

很多人选择用 Go 来构建系统，因为它已经做出了许多选择，人们希望这些选择能让它更经得起遗产的考验。

- 和我之前 Scala 生涯相比 我形容它简直会让你有上吊的冲动，Go 只有 25 个关键词和 很多 可以从标准库和一些其他小型库中构建的系统。 愿景是通过 Go 你可以编写代码并在 6 个月内回顾它，它仍然有意义。
- 与大多数替代品相比，测试，基准测试，语义解析和装载方面的工具是一流的。
- 很棒的标准库。

- 严谨的反馈回路让编译非常迅速
- Go 有向后兼容性承诺。看起来 Go 将来会获得泛型和其他功能，但设计师们已经承诺，即使你 5 年前写的 Go 代码仍然会构建。我花了几周的时间将项目从 Scala 2.8 升级到 2.10。

即使拥有所有这些优秀的属性，我们仍然可能会制造出糟糕的系统，因此我们应不论你使用的语言优秀与否，你都应该回顾过去的软件工程并理解其中获得的经验教训。

1974 年，一位名叫 [曼尼・雷曼](https://en.wikipedia.org/wiki/Manny_Lehman...) 的聪明的软件工程师写下了 雷曼软件进化定律。

> 这些定律描述了推动新发展的力量和阻碍进步的力量之间的平衡。

如果我们不希望开发的系统变成遗产，被一遍又一遍的重写，那么这些力量是我们需要着重理解的。

连续变化规律

> 实际生活中被使用的软件系统都必须不断的改变，要不然就会被大环境淘汰

很明显，一个系统 必须 不断的改变，要不然就会变的越来越没用，但是这种情况为什么经常被忽略呢?

因为很多开发团队在指定的日期交付一个项目会得到奖金，然后他们会继续开发下一个项目。如果这个软件是「幸运的」，至少它会以某种形式移交给另一组人来维护它，但是他们一定不会继续迭代它。

人们通常关心的是选择一个框架来帮助他们「快速交付」，而不是关注系统持久性发展。

即使你是一名出色的软件工程师，你仍然会因为不了解自己系统的未来需求而成为受害者。随着业务的变化，你写的出色的代码也会变得不再适用。

雷曼在 70 年代很成功，因为他给了我们另一条值得深思的规律。

复杂性增加的规律

> 随着系统的发展，除非采取措施减少系统复杂性的增加，否则系统的复杂程度会持续增加

他现在要所说的就是：我们不能让软件团队成为纯粹的功能工厂，只是通过将越来越多的功能集中到软件上， 来让系统能够继续长期运行。

随着我们知识领域的变化，我们 **必须** 持续管理系统的复杂性。

重构

软件的开发可以在 许多 方面保持软件的可塑性，例如：

- 开发人员授权
- 通常「好」的代码。关注代码合理的分离，等等
- 沟通能力
- 体系结构
- 可观性
- 可部署性
- 自动化测试
- 闭环

我将重点放在重构上。在开发人员编程的第一天经常听到的话就是「我们需要重构它」。

这句话从和而来？重构与编写代码有什么不同？

我知道我和其他很多人都 认为 我们在进行重构，但我们错了。

马丁·福勒描述了人们是如何犯错的

> 然而「重构」经常被用在不合适的地方。如果有人讨论一个系统在重构时出现了几天故障，你可以肯定他们不是在重构。

那是什么呢？

因式分解

在学校学习数学时，你可能学了因式分解。这里有一个非常简单的例子

计算 `1/2 + 1/4`

为此，将分母 分解，将表达式转换为

`2/4 + 1/4` 你可以把它变成 `3/4` .

我们可以从中吸取一些重要的教训。当我们 分解表达式 时，我们**没有改变表达式的含义**。两者都等于 `3/4` ，但我们通过将 `1/2` 变为 `2/4` 后，我们的工作变得更容易了；它更适合我们的「领域」。

当你重构代码时，你应该在「符合」你当前系统需求的情况下，尝试找到一个方法来使你的代码更容易理解。关键是**你不应该改变代码原有的行为**.

Go 的一个例子

下面这个方法是用特定的 `language` 问候 `name`

```
func Hello(name, language string) string {
    if language == "es" {
        return "Hola, " + name
    }
```

```
    if language == "fr" {

        return "Bonjour, " + name

    }

    // 想象一下更多的语言

    return "Hello, " + name

}
```

如果有几十个 `if` 语句会让人感觉不舒服，而且我们还要重复的使用特定的 `language` 去伴随着 `,` 问候 `name` 。因此，我们来重构代码。

```
func Hello(name, language string) string {
      return fmt.Sprintf(
          "%s, %s",
          greeting(language),
          name,
      )
}

var greetings = map[string]string {
  es: "Hola",
  fr: "Bonjour",
  //等等...
}

func greeting(language string) string {
  greeting, exists := greetings[language]

  if exists {
     return greeting
  }

  return "Hello"
}
```

实际上，这个重构的本质并不重要，重要的是我们没有改变代码的行为。

当重构时，你可以做任何你喜欢的事情，添加接口，新类型，函数，方法等等。唯一的规则是你不能改变代码的行为。

重构代码时不要改变功能

这非常重要。如果你重构时改变功能，你相当于同时在做 两 件事。作为软件工程师，我们应该学习把系统分成不同的文件 / 包 / 功能 / 等等，因为我们知道试图理解一大块东西是困难的。

我们不要一次想很多事情，因为那会使我们犯错误。我目睹了许多重构工作的失败，因为开发人员贪多嚼不烂。

当我在数学课上用笔和纸做因式分解时，我必须手动检查我是否改变了头脑中表达式的意思。当我们重构代码时，尤其是在一个重要的系统上，我们如何知道我们是否改变了功能？

那些选择不编写测试的人通常依赖于手动测试。除非是一个小项目，要不然这将是一个巨大的时间消耗，并且从长远来看不利于系统将来的扩展。

为了安全地重构，您需要单元测试因为它提供了

- 可以在不担心功能改变的情况下重构代码
- 有助于开发人员编写关于系统应该如何运行的文档
- 比手工测试更快更可靠的反馈

Go 的一个例子

我们有一个 `Hello` 方法的单元测试是这样的：

```
func TestHello(t *testing.T) {

    got := Hello("Chris", es)

    want := "Hola, Chris"

    if got != want {

        t.Errorf("got %q want %q", got, want)

    }

}
```

在命令行中，我们可以运行 `go test` ，并且可以立即得到我们的重构工作是否影响了原来程序运行的反馈。实际上，最好学习在编辑器 / IDE 中运行测试。

你想要得到你程序正在运行的状态

- 小的重构
- 运行测试
- 重复

所有这些都在一个非常紧密的反馈回路中，这样你就不会犯错误。

在一个项目中，你所有的关键行为都经过了单元测试，并且在一秒钟内给出反馈，这是一个非常强大的安全网，可以在你需要的时候进行大胆的重构。这有助于我们管理雷曼所描述的

日益增长的复杂性。

单元测试这么的优秀，为什么有时候编写它们会受到阻力呢?

一方面，有人（像我一样）说单元测试对你的系统的长期健康很重要，因为它们确保你可以自信地继续重构。

另一方面，有人说单元测试实际上 阻碍 了重构。

扪心自问，重构时需要多久更改一次测试？这些年来，我参与了许多测试覆盖率非常高的项目，但是工程师们不愿意重构，因为他们认为更改测试是费力的事情。

这与我们的承诺相反！

为什么会这样?

假设你被要求去画一个正方形，我们认为最好的方法是把两个三角形粘在一起。

两个直角三角形构成一个正方形

我们在正方形周围写单元测试以确保两边相等然后我们在三角形周围写一些测试。我们想确保我们的三角形渲染正确所以我们断言这些角之和是 180 度，我们做了两个测试来检查，等等。测试覆盖率非常重要，编写这些测试非常简单，为什么不呢?

几周后，不断变化的规律冲击了我们的系统，一个新的开发人员做出了一些改变。她现在认为，如果用两个矩形来构成正方形会比两个三角形更好。

两个长方形组成一个正方形

他尝试进行这种重构，并从一些失败的测试中得到一些提示。他真的破坏了代码重要的功能吗？她现在必须深入研究这些三角形的测试，并试图理解它内部到底发生了什么。

正方形由三角形组成实际上并不重要，但是**我们的测试错误地提高了实现细节的重要性** 。

测试功能而不是实现细节

当我听到人们抱怨单元测试时，通常是因为测试处于错误的抽象级别。他们都在测试实现细节，过度的观察协作者的代码并进行许多嘲讽。

我相信这个问题是因为他们对单元测试的误解，以及对度量标准 (测试覆盖率) 的追求。

如果我说的只是测试功能，我们不应该只编写系统 / 黑盒测试吗？在验证关键用户历程方面，这类测试确实有很多价值，但它们通常编写成本高，运行速度慢。由于这个原因，它们对 重构 没有太大帮助，因为反馈循环很慢。此外，与单元测试相比，黑盒测试对解决根本问题并没有太大帮助。

那什么 是 正确的抽象级别呢？

编写有效的单元测试是一个设计问题

暂时忘记测试，最好在您的系统中包含自包含的、解耦的「单元」，以您的领域中的关键概念为中心。

我喜欢将这些单元想象成简单的乐高积木，它们具有一致的 API，我可以将这些 API 与其他积木结合起来构建更大的系统。在这些 API 的内部，可能有许多东西 (类型、函数等) 协作，使它们能够按照需要工作。
例如，如果你使用 Go 开发一个银行系统，你应该会有一个「账户」包。它将提供一个不会泄漏实现细节且易于集成的 API。

如果您的单元遵循了这些属性，那么你可以针对它们的公共 API 编写单元测试。根据定义，这些测试只能测试有用的功能。在有这些单元的基础下，只要有需要，我们可以自由地实现重构，并且在大多数情况下测试不会成为我们的阻碍。

这些是单元测试吗?

是的。单元测试是针对我所描述的「单元」的。它们 从不 只针对一个类 / 函数 / 任何东西。

将这些概念结合起来

我们已经讲过了

- 重构
- 单元测试
- 单元设计

我们可以看到的是，这些方面的软件设计是相辅相成的。

重构

- 为我们的单元测试提供信号。如果我们不得不手动检查，那么我们需要更多的测试。如果测试失败，那么我们的测试就处于错误的抽象级别 (或者没有值，应该删除)
- 帮助我们处理单元内部和单元之间的复杂性。

单元测试

- 为重构提供了安全防护。
- 验证并记录我们单元的功能。

(精心设计的) 单元

- 易于编写 有意义 的单元测试。
- 易于重构。

是否有一个流程可以帮助我们不断地重构代码，以管理复杂性并保持系统的可伸缩性?

为什么要测试驱动开发（TDD）

一些人可能会因为雷曼关于如何让软件不断改变的思想，因此过度的设计，浪费大量的时间提前尝试创建「完美的」可扩展系统，结果却一事无成。

在过去糟糕的软件时代，分析师团队将花费 6 个月的时间编写需求文档，架构师团队将花费 6 个月的时间进行设计，几年后整个项目就失败了。

我说过去的日子很糟糕，但现在还是这样!

敏捷开发告诉我们，我们需要迭代地工作，从小处着手并不断改进软件，这样我们就可以快速地得到关于软件设计的反馈，以及软件如何与实际用户一起工作；TDD 强制执行这种方法。

TDD 通过鼓励一种不断重构和迭代交付的开发方式，解决了雷曼所谈论的定律和其他历史上难以学到的教训。

小步骤

- 为小功能写一个小测试
- 检查测试是否失败，并有明显的错误（红色）
- 编写最少的代码使测试通过（绿色）
- 重构
- 重复以上步骤

随着你熟练程度的挺高，对于你来说这会变为一种很自然的工作方式，工作效率也会越来越高

你开始希望你的小测试单元完成整个测试不要花费太多时间，因为如果你看到你的程序一直处于非「绿色」状态，那表明你有可能遇到了一点小麻烦。

有了这些测试反馈，你能轻松保证一些小型应用功能的稳定性。

圆满结束

- 软件的优点在于我与可以根据需要去改变他。随着时间的推移，由于一些不可预知的原因，大多数软件需要根据需求去做出相应的改变；但是一开始不要工作过头，过度的设计，因为未来太难预测。
- 因此为了满足上面的需要，我们需要保持我们的软件的可扩展性。否则当我们的软件需要重构升级时，情况将变得非常糟糕。
- 一个好的单元测试可以帮助您快速、愉悦的地进行项目重构。
- 编写好的单元测试是一个设计问题，你必须用心思考去去构建你的代码，使你的每个单元测试像拼乐高积木一样有趣的整合在一起，顺利完成整个项目的测试。
- 测试驱动开发（TDD Test-Driven Development) 可以帮助并促使你去迭代开发一个精心设计的软件，以他为技术支持，会对你未来的工作有很大的帮助。

原文地址：https://quii.gitbook.io/learn-go-with-te...
译文地址：https://learnku.com/go/t/34095