



下载APP



02 | 拒绝“Hello and Bye”：Go语言的设计哲学是怎么一回事？

2021-10-15 Tony Bai

《Tony Bai · Go语言第一课》

课程介绍 >



讲述：Tony Bai

时长 21:46 大小 19.93M



你好，我是 Tony Bai。

上一讲，我们探讨了“Go 从哪里来，并可能要往哪里去”的问题。根据“绝大多数主流编程语言将在其 15 至 20 年间大步前进”这个依据，我们给出了一个结论：**Go 语言即将进入自己的黄金 5~10 年。**

那么此时此刻，想必你已经跃跃欲试，想要尽快开启 Go 编程之旅。但在正式学习 Go 语法之前，我还是要再来给你**泼泼冷水**，因为这将决定你后续的学习结果，是“从入门到继续”还是“从入门到放弃”。



很多编程语言的初学者在学习初期，可能都会遇到这样的问题：最初兴致勃勃地开始学习一门编程语言，学着学着就发现了很多“别扭”的地方，比如想要的语言特性缺失、语法

风格冷僻与主流语言差异较大、语言的不同版本间无法兼容、语言的语法特性过多导致学习曲线陡峭、语言的工具链支持较差，等等。

其实以上的这些问题，本质上都与语言设计者的设计哲学有关。所谓编程语言的设计哲学，就是指决定这门语言演化进程的高级原则和依据。

设计哲学之于编程语言，就好比一个人的价值观之于这个人的行为。

因为如果你不认同一个人的价值观，那你其实很难与之持续交下去，即所谓道不同不相为谋。类似的，如果你不认同一门编程语言的设计哲学，那么大概率你在后续的语言学习中，就会遇到上面提到的这些问题，而且可能会让你失去继续学习的精神动力。

因此，在真正开始学习 Go 语法和编码之前，我们还需要先来了解一下 Go 语言的设计哲学，等学完这一讲之后，你就能更深刻地认识到自己学习 Go 语言的原因了。

我将 Go 语言的设计哲学总结为五点：简单、显式、组合、并发和面向工程。下面，我们就先从 Go 语言的第一设计哲学“**简单**”开始了解吧。

简单

知名 Go 开发者戴维·切尼（Dave Cheney）曾说过：“大多数编程语言创建伊始都致力于成为一门简单的语言，但最终都只是满足于做一个强大的编程语言”。

而 Go 语言是一个例外。Go 语言的设计者们在语言设计之初，就拒绝了走语言特性融合的道路，选择了“做减法”并致力于打造一门简单的编程语言。

选择了“简单”，就意味着 Go 不会像 C++、Java 那样将其他编程语言的新特性兼蓄并收，所以你在 Go 语言中看不到传统的面向对象的类、构造函数与继承，看不到结构化的异常处理，也看不到本属于函数编程范式的语法元素。

其实，Go 语言也没它看起来那么简单，自身实现起来并不容易，但这些复杂性被 Go 语言的设计者们“隐藏”了，所以 Go 语法层面上呈现了这样的状态：

仅有 25 个关键字，主流编程语言最少；

内置垃圾收集，降低开发人员内存管理的心智负担；

首字母大小写决定可见性，无需通过额外关键字修饰；

变量初始为类型零值，避免以随机值作为初值的问题；

内置数组边界检查，极大减少越界访问带来的安全隐患；

内置并发支持，简化并发程序设计；

内置接口类型，为组合的设计哲学奠定基础；

原生提供完善的工具链，开箱即用；

... ..

看，我说的没错吧，确实挺简单的。当然了，任何的设计都存在着权衡与折中。我们看到 Go 设计者选择的“简单”，其实是站在巨人肩膀上，去除或优化了以往语言中，已经被开发者证明为体验不好或难以驾驭的语法元素和语言机制，并提出了自己的一些创新性的设计。比如，首字母大小写决定可见性、变量初始为类型零值、内置以 go 关键字实现的并发支持等。

Go 这种有些“逆潮流”的“简单哲学”并不是一开始就能得到程序员的理解的，但在真正使用 Go 之后，我们才能真正体会到这种简单所带来的收益：简单意味着可以使用更少的代码实现相同的功能；简单意味着代码具有更好的可读性，而可读性好的代码通常意味着更好的可维护性以及可靠性。

总之，在软件工程化的今天，这些都意味着对生产效率提升的极大促进，我们可以认为**简单的设计哲学是 Go 生产力的源泉**。


显式

好，接下来我们继续来了解学习下 Go 语言的第二大设计哲学：**显式**。

首先，我想先带你来看一段 C 程序，我们一起来看看“隐式”代码的行为特征。

在 C 语言中，下面这段代码可以正常编译并输出正确结果：

```
1 #include <stdio.h>
2
3 int main() {
```

 复制代码

```
4     short int a = 5;
5
6     int b = 8;
7     long c = 0;
8
9     c = a + b;
10    printf("%ld\n", c);
11 }
```

我们看到在上面这段代码中，变量 a、b 和 c 的类型均不相同，C 语言编译器在编译 `c = a + b` 这一行时，会自动将短整型变量 a 和整型变量 b，先转换为 long 类型然后相加，并将所得结果存储在 long 类型变量 c 中。那如果换成 Go 来实现这个计算会怎么样呢？我们先把上面的 C 程序转化成等价的 Go 代码：

[复制代码](#)

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var a int16 = 5
7     var b int = 8
8     var c int64
9
10    c = a + b
11    fmt.Printf("%d\n", c)
12 }
```

如果我们编译这段程序，将得到类似这样的编译器错误：“invalid operation: a + b (mismatched types int16 and int)”。我们能看到 Go 与 C 语言的隐式自动类型转换不同，Go 不允许不同类型的整型变量进行混合计算，它同样也不会对其进行隐式的自动转换。

因此，如果要使这段代码通过编译，我们就需要对变量 a 和 b 进行**显式转型**，就像下面代码段中这样：

[复制代码](#)

```
1 c = int64(a) + int64(b)
2 fmt.Printf("%d\n", c)
```

而这其实就是 Go 语言**显式设计哲学**的一个体现。

在 Go 语言中，不同类型变量是不能在一起进行混合计算的，这是因为 **Go 希望开发人员明确知道自己在做什么**，这与 C 语言的“信任程序员”原则完全不同，因此你需要以显式的方式通过转型统一参与计算各个变量的类型。

除此之外，Go 设计者所崇尚的显式哲学还直接决定了 Go 语言错误处理的形态：Go 语言采用了**显式的基于值比较的错误处理方案**，函数 / 方法中的错误都会通过 return 语句显式地返回，并且通常调用者不能忽略对返回的错误的处理。

这种有悖于“主流语言潮流”的错误处理机制还一度让开发者诟病，社区也提出了多个新错误处理方案，但或多或少都包含隐式的成分，都被 Go 开发团队一一否决了，这也与显式的设计哲学不无关系。

组合

接着，我们来看第三个设计哲学：**组合**。

这个设计哲学和我们各个程序之间的耦合有关，Go 语言不像 C++、Java 等主流面向对象语言，我们在 Go 中是找不到经典的面向对象语法元素、类型体系和继承机制的，Go 推崇的是组合的设计哲学。

在诠释组合之前，我们需要先来了解一下 Go 在语法元素设计时，是如何为“组合”哲学的应用奠定基础的。

在 Go 语言设计层面，Go 设计者为开发者们提供了正交的语法元素，以供后续组合使用，包括：

Go 语言无类型层次体系，各类型之间是相互独立的，没有子类型的概念；

每个类型都可以有自己的方法集合，类型定义与方法实现是正交独立的；

实现某个接口时，无需像 Java 那样采用特定关键字修饰；

包之间是相对独立的，没有子包的概念。

我们可以看到，无论是包、接口还是一个具体的类型定义，Go 语言其实是为我们呈现了这样一幅图景：一座座没有关联的“孤岛”，但每个岛内又都很精彩。那么现在摆在面前的的工作，就是在这些孤岛之间以最适当的方式建立关联，并形成一個整体。而 **Go 选择采用的组合方式，也是最主要的方式。**

Go 语言为支撑组合的设计提供了**类型嵌入**（Type Embedding）。通过类型嵌入，我们可以将已经实现的功能嵌入到新类型中，以快速满足新类型的功能需求，这种方式有些类似经典面向对象语言中的“继承”机制，但在原理上却与面向对象中的继承完全不同，这是一种 Go 设计者们精心设计的“语法糖”。

被嵌入的类型和新类型两者之间没有任何关系，甚至相互完全不知道对方的存在，更没有经典面向对象语言中的那种父类、子类的关系，以及向上、向下转型（Type Casting）。通过新类型实例调用方法时，方法的匹配主要取决于方法名字，而不是类型。这种组合方式，我称之为**垂直组合**，即通过类型嵌入，快速让一个新类型“复用”其他类型已经实现的能力，实现功能的垂直扩展。

你可以看看下面这个 Go 标准库中的一段使用类型嵌入的组合方式的代码段：

[复制代码](#)

```
1 // $GOROOT/src/sync/pool.go
2 type poolLocal struct {
3     private interface{}
4     shared  []interface{}
5     Mutex
6     pad     [128]byte
7 }
```

在代码段中，我们在 poolLocal 这个结构体类型中嵌入了类型 Mutex，这就使得 poolLocal 这个类型具有了互斥同步的能力，我们可以通过 poolLocal 类型的变量，直接调用 Mutex 类型的方法 Lock 或 Unlock。

另外，我们在标准库中还会经常看到类似如下定义接口类型的代码段：

[复制代码](#)

```
1 // $GOROOT/src/io/io.go
2 type ReadWriter interface {
3     Reader
```

```
4     Writer
5 }
```

这里，标准库通过嵌入接口类型的方式来实现接口行为的聚合，组成大接口，这种方式在标准库中尤为常用，并且已经成为了 Go 语言的一种惯用法。

垂直组合本质上是一种“能力继承”，采用嵌入方式定义的新类型继承了嵌入类型的能力。Go 还有一种常见的组合方式，叫**水平组合**。和垂直组合的能力继承不同，水平组合是一种能力委托（Delegate），我们通常使用接口类型来实现水平组合。

Go 语言中的接口是一个创新设计，它只是方法集合，并且它与实现者之间的关系无需通过显式关键字修饰，它让程序内部各部分之间的耦合降至最低，同时它也是连接程序各个部分之间“纽带”。

水平组合的模式有很多，比如一种常见方法就是，通过接受接口类型参数的普通函数进行组合，如以下代码段所示：

```
1 // $GOROOT/src/io/ioutil/ioutil.go
2 func ReadAll(r io.Reader)([]byte, error)
3
4 // $GOROOT/src/io/io.go
5 func Copy(dst Writer, src Reader)(written int64, err error)
```

[复制代码](#)

也就是说，函数 ReadAll 通过 io.Reader 这个接口，将 io.Reader 的实现与 ReadAll 所在的包低耦合地水平组合在一起了，从而达到从任意实现 io.Reader 的数据源读取所有数据的目的。类似的水平组合“模式”还有点缀器、中间件等，这里我就不展开了，在后面讲到接口类型时再详细叙述。

此外，我们还可以将 Go 语言内置的并发能力进行灵活组合以实现，比如，通过 goroutine+channel 的组合，可以实现类似 Unix Pipe 的能力。

总之，组合原则的应用实质上是塑造了 Go 程序的骨架结构。类型嵌入为类型提供了垂直扩展能力，而接口是水平组合的关键，它好比程序肌体上的“关节”，给予连接“关

节”的两个部分各自“自由活动”的能力，而整体上又实现了某种功能。并且，组合也让遵循“简单”原则的 Go 语言，在表现力上丝毫不逊色于其他复杂的主流编程语言。

并发

好，前面我们已经看过 3 个设计哲学了，紧接着我带你看的是第 4 个：**并发**。

“并发”这个设计哲学的出现有它的背景，你也知道 CPU 都是靠提高主频来改进性能的，但是现在这个做法已经遇到了瓶颈。主频提高导致 CPU 的功耗和发热量剧增，反过来制约了 CPU 性能的进一步提高。2007 年开始，处理器厂商的竞争焦点从主频转向了多核。

在这种大背景下，Go 的设计者在决定去创建一门新语言的时候，果断将面向多核、**原生支持并发**作为了新语言的设计原则之一。并且，Go 放弃了传统的基于操作系统线程的并发模型，而采用了**用户层轻量级线程**，Go 将之称为 **goroutine**。

goroutine 占用的资源非常小，Go 运行时默认为每个 goroutine 分配的栈空间仅 2KB。goroutine 调度的切换也不用陷入（trap）操作系统内核层完成，代价很低。因此，一个 Go 程序中可以创建成千上万个并发的 goroutine。而且，所有的 Go 代码都在 goroutine 中执行，哪怕是 go 运行时的代码也不例外。

在提供了开销较低的 goroutine 的同时，Go 还在语言层面内置了辅助并发设计的原语：channel 和 select。开发者可以通过语言内置的 channel 传递消息或实现同步，并通过 select 实现多路 channel 的并发控制。相较于传统复杂的线程并发模型，Go 对并发的原生支持将大大降低开发人员在开发并发程序时的心智负担。

此外，并发的设计哲学不仅仅让 Go 在语法层面提供了并发原语支持，其对 Go 应用程序设计的影响更为重要。并发是一种程序结构设计的方法，它使得并行成为可能。

采用并发方案设计的程序在单核处理器上也是可以正常运行的，也许在单核上的处理性能可能不如非并发方案。但随着处理器核数的增多，并发方案可以自然地提高处理性能。

而且，并发与组合的哲学是一脉相承的，并发是一个更大的组合的概念，它在程序设计的全局层面对程序进行拆解组合，再映射到程序执行层面上：goroutines 各自执行特定的工作，通过 channel+select 将 goroutines 组合连接起来。并发的存在鼓励程序员在程序设计时进行独立计算的分解，而对并发的原生支持让 Go 语言也更适应现代计算环境。

面向工程

最后，我们来看一下 Go 的最后一条设计哲学：面向工程。

Go 语言设计的初衷，就是**面向解决真实世界中 Google 内部大规模软件开发存在的各种问题，为这些问题提供答案**，这些问题包括：程序构建慢、依赖管理失控、代码难于理解、跨语言构建难等。

很多编程语言设计者和他们的粉丝们认为这些问题并不是一门编程语言应该去解决的，但 Go 语言的设计者并不这么看，他们在 Go 语言最初设计阶段就**将解决工程问题作为 Go 的设计原则之一**去考虑 Go 语法、工具链与标准库的设计，这也是 Go 与其他偏学院派、偏研究型的编程语言在设计思路上的一个重大差异。

语法是编程语言的用户接口，它直接影响开发人员对于这门语言的使用体验。在面向工程设计哲学的驱使下，Go 在语法设计细节上做了精心的打磨。比如：

重新设计编译单元和目标文件格式，实现 Go 源码快速构建，让大工程的构建时间缩短到类似动态语言的交互式解释的编译速度；

如果源文件导入它不使用的包，则程序将无法编译。这可以保证任何 Go 程序的依赖树是精确的。这也可以保证在构建程序时不会编译额外的代码，从而最大限度地缩短编译时间；

去除包的循环依赖，循环依赖会在大规模的代码中引发问题，因为它们要求编译器同时处理更大的源文件集，这会减慢增量构建；

包路径是唯一的，而包名不必唯一的。导入路径必须唯一标识要导入的包，而名称只是包的使用者如何引用其内容的约定。“包名称不必是唯一的”这个约定，大大降低了开发人员给包起唯一名字的心智负担；

故意不支持默认函数参数。因为在规模工程中，很多开发者利用默认函数参数机制，向函数添加过多的参数以弥补函数 API 的设计缺陷，这会导致函数拥有太多的参数，降低清晰度和可读性；

增加类型别名（type alias），支持大规模代码库的重构。

在标准库方面，Go 被称为“自带电池”的编程语言。如果说一门编程语言是“自带电池”，则说明这门语言标准库功能丰富，多数功能不需要依赖外部的第三方包或库，Go 语

言恰恰就是这类编程语言。

由于诞生年代较晚，而且目标比较明确，Go 在标准库中提供了各类高质量且性能优良的功能包，其中的net/http、crypto、encoding等包充分迎合了云原生时代的关于API/RPC Web 服务的构建需求，Go 开发者可以直接基于标准库提供的这些包实现一个满足生产要求的 API 服务，从而减少对外部第三方包或库的依赖，降低工程代码依赖管理的复杂性，也降低了开发人员学习第三方库的心理负担。

而且，开发人员在工程过程中肯定是需要使用工具的，Go 语言就提供了足以让所有其它主流语言开发人员羡慕的工具链，工具链涵盖了编译构建、代码格式化、包依赖管理、静态代码检查、测试、文档生成与查看、性能剖析、语言服务器、运行时程序跟踪等方方面面。

这里值得重点介绍的是 **gofmt**，它统一了 Go 语言的代码风格，在其他语言开发者还在为代码风格争论不休的时候，Go 开发者可以更加专注于领域业务中。同时，相同的代码风格让以往困扰开发者的代码阅读、理解和评审工作变得容易了很多，至少 Go 开发者再也不会会有那种因代码风格的不同而产生的陌生感。Go 的这种统一代码风格思路也在开始影响着后续新编程语言的设计，并且一些现有的主流编程语言也在借鉴 Go 的一些设计。

在提供丰富的工具链的同时，Go 在标准库中提供了官方的词法分析器、语法解析器和类型检查器相关包，开发者可以基于这些包快速构建并扩展 Go 工具链。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

在这一讲中，我和你一起了解了 Go 语言的设计哲学：**简单、显式、组合、并发和面向工程**。

简单是指 Go 语言特性始终保持在少且足够的水平，不走语言特性融合的道路，但又不乏生产力。简单是 Go 生产力的源泉，也是 Go 对开发者的最大吸引力；

显式是指任何代码行为都需开发者明确知晓，不存在因“暗箱操作”而导致可维护性降低和不安全的结果；

组合是构建 Go 程序骨架的主要方式，它可以大幅降低程序元素间的耦合，提高程序的可扩展性和灵活性；

并发是 Go 敏锐地把握了 CPU 向多核方向发展这一趋势的结果，可以让开发人员在多核时代更容易写出充分利用系统资源、支持性能随 CPU 核数增加而自然提升的应用程序；

面向工程是 Go 语言在语言设计上的一个重大创新，它将语言要解决的问题域扩展到那些原本并不是由编程语言去解决的领域，从而覆盖了更多开发者在开发过程遇到的“痛点”，为开发者提供了更好的使用体验。

这些设计哲学直接影响了 Go 语言自身的设计。理解这些设计哲学，也能帮助我们理解 Go 语言语法、标准库以及工具链的演化决策过程。

好了，学完这节课之后，你认同 Go 的设计哲学吗？认同的话就继续跟着我学下去吧。

思考题

今天，我还想问下你，你还能举出哪些符合 Go 语言设计哲学的例子吗？欢迎在留言区多多和我分享讨论。

感谢你和我一起学习，也欢迎你把这节课分享给更多对 Go 语言的设计哲学感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 13

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | 前世今生：你不得不了解的Go的历史和现状

精选留言 (7)

 写留言

**自由**

2021-10-15

Tony Bai 老师，你好，例举两个我认为复合 Go 语言设计哲学的例子，我的技术能力捉襟见肘，说的不对地方还希望老师斧正。

一、异常处理

...

展开 ∨

作者回复: 自由。说的很好。很认同你提到的基于“类型别名”的渐进式代码修复(Gradual code repair) 思路。这也是类型别名最初被引入go的初衷 (<https://github.com/golang/proposal/blob/master/design/18130-type-alias.md>)。我觉得它也是go面向工程设计哲学的体现。另外type alias在基于现有实现进行扩展并做出新的封装方面也有“奇效”。



7

**学昊**

2021-10-15

本人老java码农了。进阶的代码设计是设计模式，让代码能更优雅的实现。终极的代码设计是哲学，是代码中表达出的价值观。



2

**罗杰**

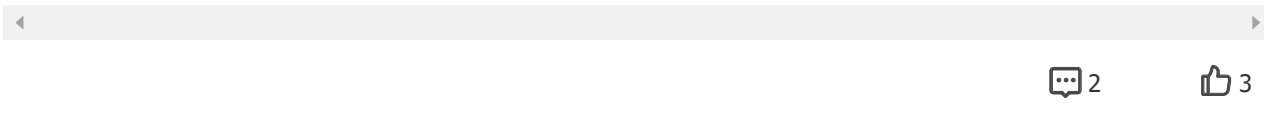
2021-10-15

21 世纪的 C 语言，的确至名归。依然有几个小问题：1. Go 有 GC，我们使用 Go 来开发后端的所有服务，有个 PVP 的服务，需要逐帧计算客户端上报的结果是否正确，此时对于内存的分配就要特别小心，开发起来很不顺畅。是否这种服务的性质不太合适使用 Go 来开发；2. 有人吐槽 Go 核心人员不想做的东西，就是 Less is more，自己想做就是各种哲学，这个问题，老师怎么看？

作者回复: 罗杰出品的问题，都是精品问题^_^。

先来看第一个问题，Go最初设计目标是通用的系统编程语言，但Go选择支持了GC。Go的GC虽然在go团队的努力下，开销越来越小，但开销小，低延迟不代表没有，这就决定了Go在一些对性能极其敏感的领域可能并不是最好的选择。你的问题中也提到了pvp服务，想必你们也是采用了面向服务的架构，这种架构本身就是可以天然适合技术异构的。如果觉得不妥，也别强求，果断换非GC语言，比如c、c++或是rust。如果说非要坚持用go来完成，那么说明你是go的骨灰粉，在解决问题的过程中，你也会完成一次go技能的升华。

第二个问题，Go语言的简单或者说功能特性少，的确来自与less is more的理念。保持一门小语言，让语言更容易学习与理解。同时每个特性都是经过精心打磨与实现，不能再少了。上周我看了rob pike最新一期的talk，他还在说“Go语言中变量声明的方式有些多了”，这也是我在实际编码过程中的体会。如果重新来过，我想rob pike会更彻底的执行less is more，将变量声明方式再减少一种。所以说，特性少不是不想做，而是经过深思熟虑，那个特性的确没必要加入到语言中。



2

3

**费城的二鹏**

2021-10-15

老师讲的很透彻，如数家珍。



3

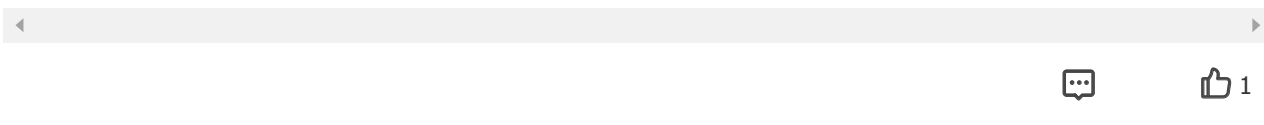
3

**Geek_399042**

2021-10-15

自动加入分号是不是也是简单的设计哲学呢，能让编译器做的事不需要交给开发者。

作者回复: 手动点赞



1

1

**lesserror**

2021-10-15

感谢 Tony Bai 这一篇的分享，很精彩。有以下几点困惑，麻烦有时间回答一下：

1. go1.16.4 版本中的 poolLocal 结构体的实现和本文中的不太一样呢？
2. 水平组合“模式”还有点缀器、中间件等方式后面的文章中会有例子吗？...

展开 ∨

作者回复:

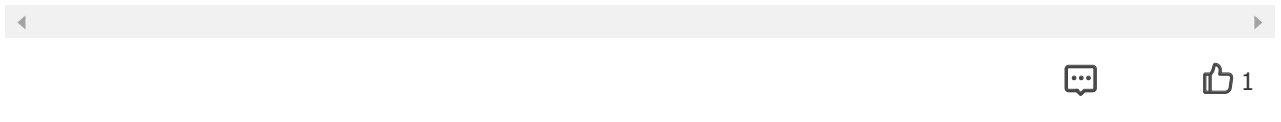
你真非常细致和喜欢思考，提出的问题都很棒！👍

第一个问题，专栏中的poolLocal实现的确是早期的版本，不过也无伤大雅，文中的目的就是为了说明：类型嵌入的组合方式。感谢

你指出。不过这块我就不改了。

第二个问题，在讲解接口类型时，应该会有具体例子。

第三个问题，在后面讲解并发的章节中，我希望我的讲解能让你觉得更通俗易懂^_^。



liaomars

2021-10-15

go的异常处理，使用起来简单，但是不方便，请问老师这是在践行go的简单设计哲学吗？

作者回复: 从go设计者的初衷来看(<https://golang.google.cn/doc/faq#exceptions>)，go没有采用像java那样的结构化异常处理的确是出于对“简单”原则的考虑。

在java中错误处理与真正的“异常”是混杂在Try-catch机制中的，并没有明显的界限，无论是错误还是异常，一旦throw，方法的调用者就得负责处理它。

但在go中，错误处理与真正的异常处理是严格分开的，也就是说不要将panic掺和到错误处理中。

错误处理是常态，go中只有错误是返回给上层的。一旦出现panic，这意味着整个程序处于即将崩溃的状态，返回给上层几乎也是“无济于事”，所以在go中，一个常见的api设计思路是 不要向外部抛出panic (don't panic!)。如果api中存在panic的可能性，那么api自己要负责处理panic，并通过error将状态返回给上层。如果api无法处理panic，那程序就很大可能是要崩溃了，这种panic多是因为程序bug导致的。

