Design by Contract на минималках для Go

Aleksandr Ivanov
alexandr.ivanov@yadro.com

YADRO
БУДУЩЕЕ В НАШИХ РУКАХ

Александр Иванов

Старший инженер-программист, team lead
ЯДРО - ЦЕНТР ТЕХНОЛОГИЙ МОБИЛЬНОЙ СВЯЗИ

- Более 20 лет программирую на C/C++ и Go
- Руковожу талантливыми разработчиками и тестировщиками

Сегодня поговорим о:

01

Контрактное программирование - это не **программирование за деньги :)**

02

Ошибках из-за неправильных параметров передаваемых в функцию

03

Опыт использования аналога assertions из C/C++ в коде на языке Go

04

О пользе Design by Contract и встречающихся антипаттернах

Введение

Контрактное программирование

Предусловия и Постусловия и Ширина контракта

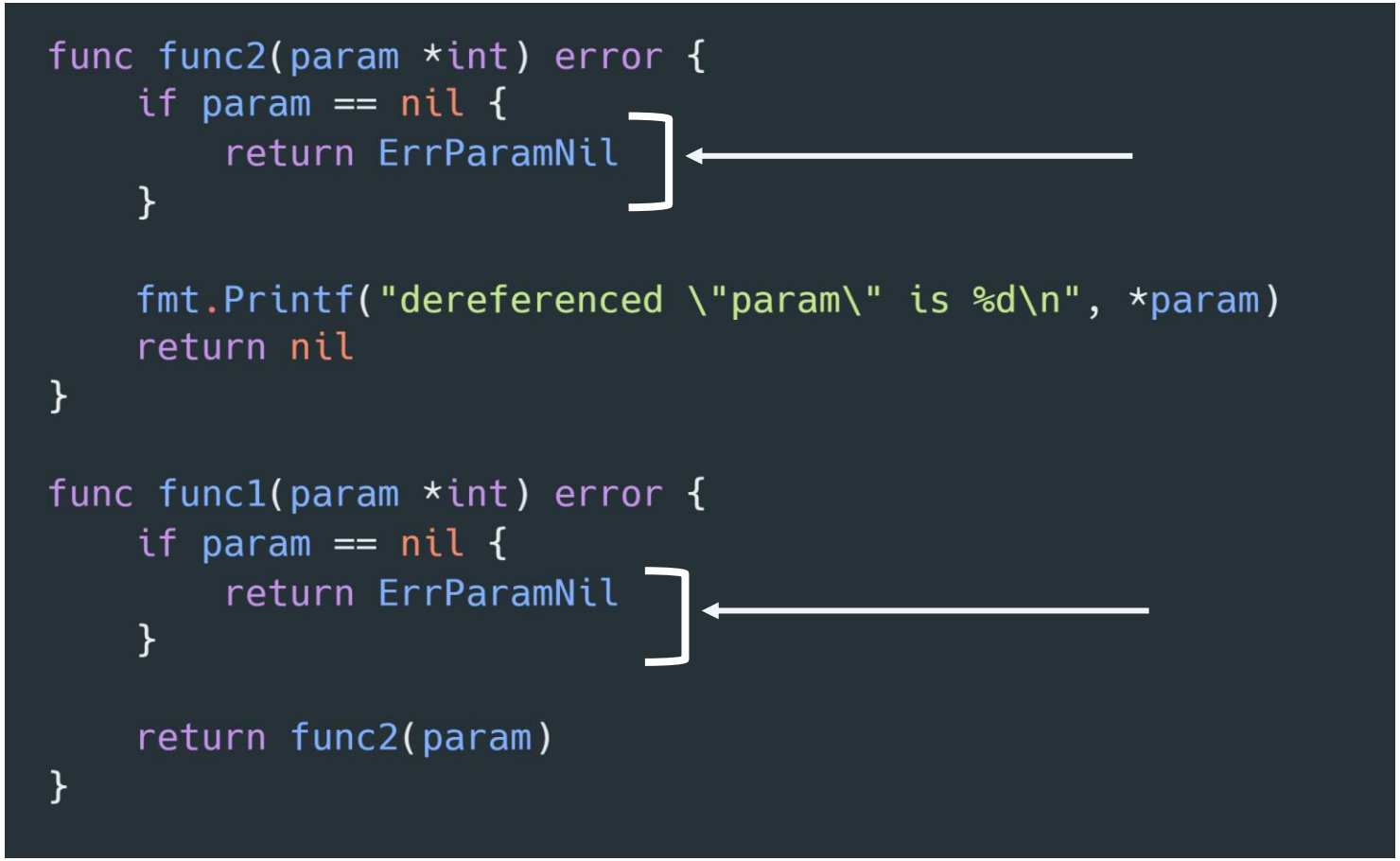
Пишем свои Assert'ы

Применения Assert'ов

Польза Assert'ов

Антипаттерны

Defensive подход



```go
func func2(param *int) error {
    if param == nil {
        return ErrParamNil
    }

    fmt.Printf("dereferenced \"param\" is %d\n", *param)
    return nil
}

func func1(param *int) error {
    if param == nil {
        return ErrParamNil
    }

    return func2(param)
}
```

Код засоряется и присутствует некоторая избыточность проверок:

- В вызывающей func1 и в вызываемой func2 переменная param проверяется на nil
- Если после компиляции не произойдёт inline func2 в func1 проверка будет выполнена дважды

Введение

Контрактное программирование aka DbC

Предусловия и Постусловия и Ширина контракта

Пишем свои Assert'ы

Применения Assert'ов

Польза Assert'ов

Антипаттерны

Контрактное программирование (DbC)


```go
var ErrParamNil = errors.New("\"param\" should not be nil")

func func2(param *int) error {
    fmt.Printf("dereferenced \"param\" is %d\n", *param)
    return nil
}

// Func1 expects param initialized by value other than nil
func Func1(param *int) error {
    return func2(param)
}
```

- Код чище отсутствует избыточность проверок
- Но комментарии – не очень надёжная договорённость

Введение

Контрактное программирование

Предусловия и Постусловия и Ширина контракта

Пишем свои Assert'ы

Применения Assert'ов

Польза Assert'ов

Антипаттерны

Предусловия

Для строгого описания договорённостей о вызове используют:

Предусловия

Нарушение предусловия предупреждает о баге **на стороне вызывающего кода**, который не соблюдает свою часть контракта

```
func funcWithPrecond(param *int)  {
  assertions.Assert(param != nil, "param should not be nil")

  // some code has been skipped here
}
```

Предусловия

Для строгого описания договорённостей о вызове используют:

Постусловия

Нарушение постусловия предупреждает о баге **на стороне вызываемого кода**, который неправильно выполняет свою работу

```
funcShouldKeepInputParamConsistency(&someStruct)
assertion.Assert(
    isConsistent(someStruct),
    "previous call has inconsistently changed struct fields",
)
```

Ширина контракта

Defence

Контракты, защищающие вызываемый код «от дурака», называются широкими контрактами

```go
// Func1 expects param initialized by value other than nil
func Func1(param *int) error {
    if param == nil {
        return ErrParamNil
    }

    return func2(param)
}
```

DbC

Контракты, которые обуславливают всевозможные предусловия, называются узкими контрактами

```go
// func2 expects param initialized by value other than nil
func func2(param *int) error {
    assertion.Assert(param != nil, "func2 expects non nil param")

    fmt.Printf("dereferenced \"param\" is %d\n", *param)
    return nil
}
```

Введение

Контрактное программирование

Предусловия и Постусловия в DbC и ширина контракта

Пишем свои Assert'ы

Применения Assert'ов

Польза Assert'ов

Антипаттерны

Для отладки и тестирования

```go
//go:build enable_assert

// Package assertion provides method for asserting your code
package assertion

import (
    "fmt"
    "log"
)

const panicMessage = "Assertion happened"

// Assert will panic in case when condition is not true
func Assert(condition bool, format ...any) {
    if !condition {
        s := fmt.Sprintf(format[0].(string), format[1:])
        log.Fatal(s)
    }
}
```

Для production

```go
//go:build !enable_assert

// Package assertion provides method for asserting your code
package assertion

// Assert will print error log in case when condition is not true
func Assert(condition bool, format ...any) {
    // do nothing
    return
}
```

- Production вариант отличается от того, который используется для отладки и тестирования
- Он может совсем ничего не делать или только логировать факт ошибки, без остановки работы программы

Введение

Контрактное программирование aka DbC

Предусловия и Постусловия в DbC и ширина контракта

Пишем свои Assert'ы

Применения Assert'ов

Польза Assert'ов

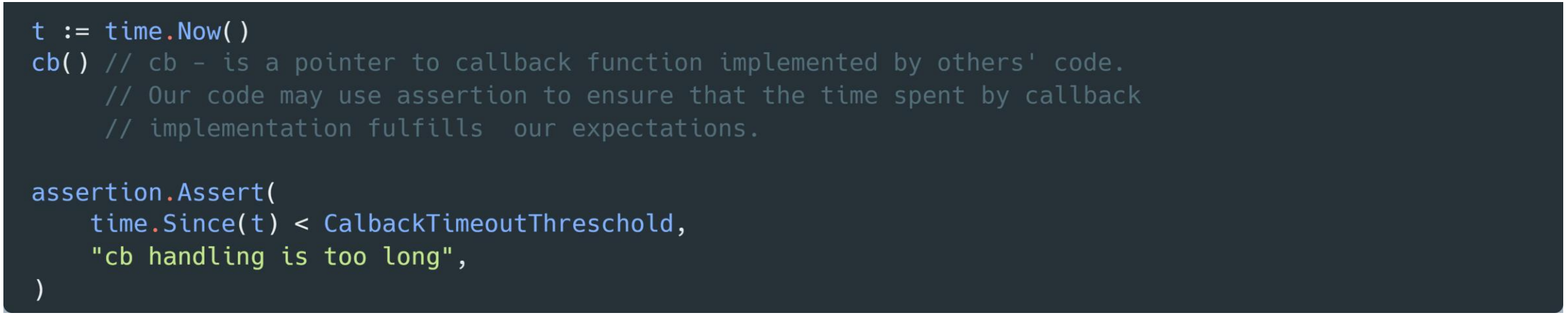
Антипаттерны

Применения Assert'ов: проверка предусловий

```go
// func2 expects param initialized by value other than nil
func func2(param *int) error {
    assertion.Assert(param != nil, "func2 expects non nil param")

    fmt.Printf("dereferenced \"param\" is %d\n", *param)
    return nil
}

// Func1 expects param initialized by value other than nil
func Func1(param *int) error {
    if param == nil {
        return ErrParamNil
    }

    return func2(param)
}
```

Пример: проверяем ожидаемое условие, накладываемых на вызывающую сторону, о том, что param != nil

Применения Assert'ов: проверка постусловий



```
t := time.Now()
cb() // cb - is a pointer to callback function implemented by others' code.
     // Our code may use assertion to ensure that the time spent by callback
     // implementation fulfills  our expectations.

assertion.Assert(
    time.Since(t) < CalbackTimeoutThreschold,
    "cb handling is too long",
)
```

Пример: вызов callback функции должен укладываться в некоторый ожидаемый временной промежуток, иначе тот, кто реализует callback, должен переписать его.

Введение

Контрактное программирование aka DbC

Предусловия и Постусловия в DbC и ширина контракта

Пишем свои Assert'ы

Применения Assert'ов

Польза Assert'ов

Антипаттерны

Польза от использования assertion.Assert

Самодокументированный код

Добавляем текст, который может довольно подробно пояснить почему проверка не прошла

Улучшаем читабельность кода

Избавляемся от 3-х строчные if-then-else и меняем на одну строку с поясняющим Assert

Улучшение при отладке

Возможность добавить call stack в сообщение Assert'а

Польза от использования assertion.Assert

Поддержание консистентными кода и некоторой модели

При использовании генераторов кода из модели, можно проверять, что реализация, зависимая от некоторой версии модели и сама модель, соответствуют друг другу

Улучшение работы с кэшем CPU

Убираем из скомпилированного кода jump'ы и branch', тем самым уменьшаем cache-misses

Введение

Контрактное программирование aka DbC

Предусловия и Постусловия в DbC и ширина контракта

Пишем свои Assert'ы

Применения Assert'ов

Польза Assert'ов

Антипаттерны

Антипаттерны использования assertion.Assert

Assert вместо if err != nil { return err}

Одно из неправильных применений Assert – это замена им проверки, которая действительно должна быть и на которую действительно нужно реализовать реакцию в коде

Антипаттерны использования assertion.Assert

Вычисления при вызове Assert

Ещё одна распространённая и трудно выявляемая ошибка – это выполнение вычислений и присваиваний значений переменным прямо при вызове Assert, которые могут быть упразднены при оптимизации кода компилятором

```
Assert(i++ > 0, "осторожно, не факт, что в релизе i увеличится")
```

```
Assert(call_to_f1(), "осторожно, не факт, что call_to_f1() будет вызвана в релизе")
```

Антипаттерны использования assertion.Assert

Не следует торопиться удалять Assert так как это часть описания контракта

Assert – это часть контракта, непонимание этого может привести к тому, что разработчик захочет просто удалить проверку.

Например: в нынешней реализации (Go 1.23) пакета fmt функция Printf всегда возвращает err равное nil. И практически все игнорируют возвращающееся значение ошибки, тогда как могли бы по-хорошему проверять постусловие assertion.Assert(err == nil) чтобы рано или поздно в последующих версиях начать реагировать на err отличный от nil

Выводы

Defensive vs DbC

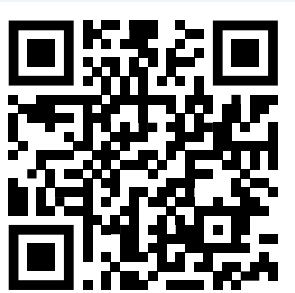
Можно сочетать: часто Defensive подход, то есть **широкие контракты**, для видных извне функций API, а DbC, то есть **узкие контракты**, для внутренней реализации

Unit-tests vs DbC

Можно сочетать: на пути к качеству нужно использовать все способы повышения качества и не стоит пренебрегать любой возможностью

Учитывайте нужды проекта

Можно писать самостоятельно, а можно использовать имеющиеся реализации DbC, например: github.com/drblez/dbc



QA

Москва,
ул. Рочдельская, 15, стр. 13
+7 800 777-06-11

yadro.com