Дубровин Д.М. Эффективность фреймворков для внедрения зависимостей в GOLANG // Международный журнал информационных технологий и энергоэффективности. – 2025. – Т. 10 № 3(53) с. 13–19

Международный журнал информационных технологий и энергоэффективности

Сайт журнала: http://www.openaccessscience.ru/index.php/ijcse/

УДК 004.438

ЭФФЕКТИВНОСТЬ ФРЕЙМВОРКОВ ДЛЯ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ В GOLANG

Дубровин Д.М.
ФГБОУ ВО "МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)", Москва, Россия, (125993, Москва, Волоколамское ш., д. 4), e-mail: daniildubr0vin@yandex.ru

В статье рассматривается паттерн "Внедрение зависимостей" (Dependency injection) и проводится сравнительный анализ различных фреймворков, реализующих этот паттерн в языке программирования Golang. Внедрение зависимостей позволяет управлять зависимостями объектов через внешние компоненты, что способствует улучшению модульности, тестируемости и поддержки кода. Также исследуется производительность этих фреймворков. В ходе анализа, были выявлены ключевые особенности и различия между подходами внедрения зависимостей, основанными на рефлексии и кодогенерации.

Ключевые слова: Внедрение зависимостей, Golang, Wire, Dig, Fx.

THE EFFECTIVENESS OF FRAMEWORKS FOR IMPLEMENTING DEPENDENCIES IN GOLANG

Dubrovin D.M.
MOSCOW AVIATION INSTITUTE (NATIONAL RESEARCH UNIVERSITY), Moscow, Russia, (125993, Moscow, Volokolamskoye shosse, 4), e-mail: daniildubr0vin@yandex.ru

The article examines the "Dependency Injection" (DI) pattern and conducts a comparative analysis of various frameworks implementing this pattern in the Go programming language. Dependency Injection allows managing object dependencies through external components, enhancing modularity, testability, and code maintainability. The performance of these frameworks is also investigated. During the analysis, key features and differences between reflection-based and code generation-based approaches to dependency injection were identified.

Keywords: Dependency injection, Golang, Wire, Dig, Fx.

Введение

Внедрение зависимостей (англ. Dependency Injection, сокр. DI) — это паттерн проектирования, основным принципом которого является предоставление управления жизненным циклом зависимостей объекта другому внешнему компоненту [1]. Применение данного паттерна помогает следовать принципам инверсии зависимостей и единой ответственности SOLID [2]. В DI зависимости объекта предоставляются извне, а не создаются внутри самого объекта. Например, предположим, что для эффективного выполнения своих операций сервису X требуется функция из сервиса Y. Вместо того, чтобы сервис X создавал новый экземпляр сервиса Y внутри себя, рекомендуется использовать DI, чтобы отдельный компонент отвечал за создание экземпляра сервиса Y, а затем внедрял этот экземпляр в сервис X.

Основные преимущества использования DI в разработке программного обеспечения:

- Слабая связанность**:** в DI объекты зависят от абстракций, а не от конкретных реализаций, что обуславливает слабую связанность кода. Благодаря этому упрощается поддержка, тестирование и рефакторинг**.**
- Улучшение тестирования**:** благодаря внедрению зависимостей становится проще заменять реальные объекты тестовыми дублерами во время модульных или интеграционных тестов.
- Модульность и возможность повторного использования**:** DI способствует улучшению структуры, разделяя приложения на более мелкие и автономные модули и компоненты. У каждого компонента есть свои зависимости, что позволяет использовать их в любом контексте.
- Конфигурация во время выполнения**:** управление зависимостями извне дает возможность настраивать и переключать их в зависимости от различных условий выполнения.

Основные недостатки DI:

- Повышенная сложность: использование DI может увеличить сложность понимания кода из-за необходимости определять зависимости и управлять их жизненными циклами. Также DI может привести к увеличению ошибок и сбоев приложения, связанных с неправильными или неразрешёнными зависимостями.
- Снижение производительности: внедрение зависимостей может снизить производительность из-за динамического разрешения зависимостей во время выполнения.
- Увеличение времени адаптации: внедрение DI требует времени на освоение его принципов, что может замедлить начальную разработку.

Несмотря на недостатки данного паттерна, использование Dependency Injection является широко признанной и часто используемой практикой в современной разработке программного обеспечения, особенно в контексте предметно-ориентированного проектирования (Domain-Driven Design) [3].

Практический пример простого внедрения зависимости через функцию в Golang: в следующем фрагменте кода объявлены две структуры JapanPrinter и RusPrinter, которые затем посредством общего интерфейса по очереди внедряются в PrinterService (Рисунок 1). При этом управление зависимостями происходит не внутри PrinterService, а во внешнем коде, то есть в функции main.

```go
// Интерфейс для сервиса Printer
type Printer interface {
    Print() string
}

type JapanPrinter struct{}

func (e JapanPrinter) Print() string {
    return "こんにちは!"
}

type RusPrinter struct{}

func (e RusPrinter) Print() string {
    return "Привет!"
}

// Сервис, который зависит от Printer
type PrinterService struct {
    printer Printer
}

func NewPrintingService(printer Printer) *PrinterService {
    return &PrinterService{printer: printer}
}

func (s *PrinterService) SayHello() {
    fmt.Println(s.printer.Print())
}

func main() {
    // Внедрение зависимости JapanPrinter в PrintingService
    printerService := NewPrintingService(JapanPrinter{})
    printerService.SayHello()

    // Внедрение зависимости RusPrinter в PrintingService
    printerService = NewPrintingService(RusPrinter{})
    printerService.SayHello()
}
```

Рисунок 1 - Практический пример внедрения зависимости в Golang

Сравнительный анализ фреймворков для внедрения зависимостей

На практике под применением паттерна Dependency Injection обычно имеется ввиду IoC-контейнеры, позволяющие автоматизировать управление жизненным циклом объектов и их зависимостями в приложении [4]. Однако, в отличие от других языков программирования в Golang отсутствует встроенная поддержка данного подхода, поэтому разработчики, использующие Golang, должны либо разрабатывать собственную реализацию IoC-контейнеров, либо использовать существующие решения. В данной статье исследованы 5 фреймворков: Dig, Fx, Wire, Di, Do.

Dig – это IoC-контейнер, использующий рефлексию для динамического внедрения зависимостей [5]. Этот инструмент позволяет определять зависимости через интуитивно понятный API и разрешать их на основе графа зависимостей. Dig поддерживает гибкую настройку процесса внедрения, что делает его идеальным выбором для сложных проектов, требующих высокой степени гибкости и настройки.

Кроме Dig, компания Uber разработала Uber FX – фреймворк, основанный на Dig, но с расширенными возможностями [6]. Uber FX упрощает настройку крупных приложений, управляя логированием, внедрением зависимостей и запуском приложений. Однако, несмотря на свои преимущества, оба фреймворка имеют недостатки. Использование рефлексии сильно замедляет их работу, а ошибки в графе зависимостей можно обнаружить только во время выполнения программы, а не на этапе компиляции. Оба фреймворка имеют обширную

документацию, а также поддерживаются активными сообществами пользователей и разработчиков.

Wire – это фреймворк для внедрения зависимостей в Golang, разработанный компанией Google [7]. Вместо рефлексии Wire использует генерацию кода, что позволяет автоматически подключать зависимости без накладных расходов во время выполнения программы. Однако в отличии от других фреймворков Wire ориентирован на внедрение зависимостей на основе инициализации и не поддерживает некоторые функции, например, middleware и interceptors. Кроме того, возможности настройки Wire ограничены из-за его сильной зависимости от генерации кода, что делает его менее функциональным по сравнению с другими фреймворками DI. В Wire нет контейнера для зависимостей, как в других фреймворках. Например, в Dig есть контейнер, который управляет жизненным циклом объектов и позволяет их динамически запрашивать. Это упрощает работу с зависимостями в реальном времени и предоставляет дополнительные возможности для тестирования и управления состоянием. Wire, напротив, генерирует код для разрешения зависимостей, и вся структура зависимостей фиксирована на момент компиляции. Это снижает гибкость, особенно когда необходимо взаимодействовать с компонентами в рантайме. Данный фреймворк активно поддерживается разработчиками.

Оба фреймворка, Do и Di, реализуют паттерн внедрения зависимостей с использованием рефлексии для динамического внедрения зависимостей во время выполнения программы. Существенным недостатком данных проектов является неполная документация и отсутствие примеров использования. Пример создания IoC-контейнера, используя фреймворк Do показан на Рисунке 2.

```go
// создание контейнера DI
injector := do.New()

do.Provide(injector, NewCar)
do.Provide(injector, NewEngine)

//вызов car создаст экземпляр Car services и его зависимость от двигателя
car, err := do.Invoke[*Car](injector)
if err != nil {
    log.Fatal(err.Error())
}

car.Start()

// управление завершением работы программы
injector.ShutdownOnSignals(syscall.SIGTERM, os.Interrupt)
```

Рисунок 2 - Практический пример использования фреймворка Do

Пример использования фреймворка Di показан на Рисунке 3. В данном случае для каждого http запроса создается подконтейнер, который управляет временем жизни зависимостей, таких как соединение с базой данных.

```go
builder, _ := di.NewEnhancedBuilder()
builder.Add(MysqlPoolDef)
builder.Add(MySqlDef)
app, _ := builder.Build()
defer app.Delete()

http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    // Create a request and delete it once it has been handled.
    // Deleting the request will close the connection.
    request, _ := app.SubContainer()
    defer request.Delete()

    handler(w, r, request)
})

http.ListenAndServe(":8080", nil)
```

Рисунок 3 - Практический пример использования фреймворка Di

В Таблице 1 представлено сравнение фреймворков по расписанным выше критериям.

Таблица 1 - Сравнительный анализ фреймворков внедрения зависимостей

	Dig	**Fx**	**Wire**	**Do**	**Di**
Этап внедрения зависимостей	Во время работы программы	Во время работы программы	Во время компиляции	Во время работы программы	Во время работы программы
Гибкость	+	+	-	+	+
Генерация кода	-	-	+	-	-
Наличие документации	+	+	+	-	-
Наличие примеров использования	+	+	+	-	-
Сообщество и поддержка	+	+	+	-	-
Поддержка жизненного цикла	+	+	-	+	+

Анализ производительности фреймворков внедрения зависимостей

Рассмотренные фреймворки были проанализированы по потреблению памяти и времени работы. Измерения производились на вычислительной машине со следующими характеристиками:

- CPU: AMD Ryzen 7 5800H CPU @ 3.20GHz
- RAM: 32 GB
- Версия OS: Windows 11, amd64
- Версия Golang: 1.24.0

Принцип тестирования следующий: для каждого тестируемого фреймворка создается большое количество структур (всего было создано 450), которые зависимы друг от друга, аналогично числам Фибоначчи (Рисунок 4).

```go
type Fib1 struct{}

type Fib2 struct {
    Fib1 *Fib1
}

type Fib3 struct {
    Fib2 *Fib2
    Fib1 *Fib1
}

//     ..... еще 447 структур
```

Рисунок 4 - Структуры, используемые для тестирования фреймворков

Каждый тест запускает внедрение зависимости для выбранной структуры 100 раз, чтобы результаты были более точными и отражали среднюю производительность в условиях повторяющейся нагрузки. Результаты для каждого фреймворка представлены в Таблице 2.

Таблица 2 - Анализ производительности фреймворков

Фреймворк	Время работы	Потребляемая память
Dig	6.03 мс	0.36 мб
Fx	7.45 мс	0.39 мб
Wire	3.04 мс	0.2 мб
Do	12 мс	0.42 мб
Di	9.23 мс	0.45 мб

Проанализировав Таблицу 2, можно сделать вывод, что фреймворки, использующие кодогенерацию во время компиляции для внедрения зависимостей более эффективны.

Заключение

Dependency Injection является важным инструментом для эффективного управления зависимостями в приложениях, улучшая их модульность, тестируемость и облегчая процесс поддержки. В ходе анализа фреймворков DI для Golang, были выявлены ключевые особенности и различия между подходами, основанными на рефлексии и кодогенерации. Фреймворки, использующие генерацию кода, такие как Wire, демонстрируют минимальные накладные расходы, что делает их предпочтительными в контексте производительности. С другой стороны, фреймворки, применяющие рефлексию, такие как Dig или Fx, предоставляют большую гибкость, позволяя динамически изменять зависимости от контекста выполнения. Также важно учитывать, что наличие документации и активного сообщества вокруг фреймворка играет немалую роль в выборе инструмента. Хорошо документированные и поддерживаемые фреймворки упрощают процесс обучения и разработки. В заключение можно отметить, что правильный выбор фреймворка для внедрения зависимостей должен основываться на балансе между производительностью, гибкостью и удобством использования.

Список литературы

1. Dependency Injection. [Электронный ресурс] URL: https://en.wikipedia.org/wiki/Dependency_injection. (дата обращения 17.02.2025).
2. *SOLID* [Электронный ресурс] URL https://ru.wikipedia.org/wiki/SOLID_(программирование). (дата обращения 15.02.2025).
3. Domain-driven design [Электронный ресурс] URL https://en.wikipedia.org/wiki/Domain-driven_design. (дата обращения 16.02.2025).
4. Inversion of control [Электронный ресурс] URL https://en.wikipedia.org/wiki/Inversion_of_control. (дата обращения 17.02.2025).
5. Documentation of Dig [Электронный ресурс] URL https://pkg.go.dev/go.uber.org/dig. (дата обращения 17.02.2025).
6. Documentation of Fx [Электронный ресурс] URL https://pkg.go.dev/go.uber.org/fx. (дата обращения 18.02.2025).
7. Documentation of Wire [Электронный ресурс] URL https://pkg.go.dev/github.com/google/wire. (дата обращения 19.02.2025).

References

1. Dependency Injection. [Electronic resource] URL: https://en.wikipedia.org/wiki/Dependency_injection . (accessed 17.02.2025).
2. SOLID [Electronic resource] URL https://ru.wikipedia.org/wiki/SOLID_ (programming). (accessed 02/15/2025).
3. Domain-driven design [Electronic resource] URL https://en.wikipedia.org/wiki/Domain-driven_design . (accessed 02/16/2025).
4. Inversion of control [Electronic resource] URL https://en.wikipedia.org/wiki/Inversion_of_control . (accessed 17.02.2025).
5. Documentation of Dig [Electronic resource] URL https://pkg.go.dev/go.uber.org/dig . (accessed 17.02.2025).
6. Documentation of Fx [Electronic resource] URL https://pkg.go.dev/go.uber.org/fx . (accessed 02/18/2025).
7. Documentation of Wire [Electronic resource] URL https://pkg.go.dev/github.com/google/wire (accessed 02/19/2025).