Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Отчет по практике»

Студент ИУ7-86Б _____ Д. В. Варин
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсовой работы _____ _____
(Подпись, дата) (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире, где информационные технологии занимают все более важное место, защита данных становится критически важным вопросом.

Резервное копирование является одним из ключевых инструментов для обеспечения надежности и безопасности данных. Оно позволяет восстановить данные в случае их потери или повреждения.

Существует множество методов резервного копирования, отличающихся друг от друга по способу организации резервных копий, использованию избыточной информации и т.д.

В рамках работы предстоит разработать метод резервного копирования с контролируемым размером избыточной информации. Этот метод позволяет контролировать размер такой информации, что повышает эффективность использования ресурсов хранения.

Важным элементом резервного копирования является выбор хранилища для резервных копий. Распределенные файловые системы предоставляют множество преимуществ в этом отношении, в том числе возможность распределения данных по нескольким узлам, обеспечения отказоустойчивости и т.д.

Цель работы — реализовать метод резервного копирования с контролируемым размером избыточной информации в распределенном файловом хранилище.

Для достижения поставленной цели требуется решить следующие задачи:

- выбрать инструменты и технологии, требуемые для реализации метода;
- реализовать новый метод в виде программного комплекса;
- привести пример использования разработанного метода;
- описать методы тестирования.

1 Технологический раздел

1.1 IDEF0-диаграмма разрабатываемого метода

На рисунке ниже представлена диаграмма IDEF0 метода резервного копирования.

Рисунок 1.1 – IDEF0–диаграмма уровня A0

На вход метода подаются:

- место хранения суперблоков - расположение директорий в ФС;
- место хранения блоков файлов - расположение директории в ФС;
- файлы, для которых делается резервная копия.

Выходными данными являются резервные копии файлов (суперблоки).

Механизмы, используемые методом — файловая система (ФС) и хранилища данных (директории для хранения).

1.2 Выбор инструментов и технологий

1.2.1 Язык программирования

В качестве языка программирования для разработки было решено использовать язык Golang [1]. К преимуществам данного языка можно отнести статическую типизацию, компилируемость, большое разнообразие библиотек для документирования, тестирования и создания интерфейсов.

Статическая типизация обеспечивает:

- Надежность и безопасность — статическая типизация позволяет обнаруживать ошибки во время компиляции, такие как несоответствие типов, отсутствие ожидаемых методов и неправильное использование переменных. Это помогает предотвратить множество ошибок времени выполнения и повысить надежность и безопасность программы.

- Читаемость и поддержку кода — статическая типизация делает код более читаемым и понятным, поскольку типы переменных и функций указывают на их предназначение и взаимодействие. Кроме того, статическая типизация облегчает сопровождение кода, поскольку компилятор и инструменты разработчика могут предоставлять более точные подсказки и автодополнение кода.

Golang обладает малым размером исполняемых файлов и низким потреблением памяти. Это особенно важно для метода резервного копирования, где требуется обработка больших объемов данных. Go позволяет эффективно использовать ресурсы, что обеспечивает более быструю и эффективную обработку данных при резервном копировании.

1.2.2 Интерфейс приложения

Приложение будет представлять из себя консольную утилиту (CLI) [2].

Для этого будет использоваться библиотека cobra [3], которая предоставляет:

1. механизмы обработки ошибок и валидации пользовательского ввода;
2. API для создания команд и флагов;
3. предоставляет встроенную поддержку генерации справки по командам и флагам;
4. обеспечивает поддержку для тестирования.

Такой интерфейс имеет ряд преимуществ:

- CLI обеспечивает легкость в автоматизации и скриптовании операций резервного копирования. Пользователь может создавать скрипты или автоматические задачи для выполнения резервного копирования по расписанию (cron [4]) или на основе определенных событий.
- CLI-интерфейсы более гибкие и переносимые, поскольку они не привязаны к конкретной операционной системе или графической библиотеке.

Выбор CLI для метода резервного копирования обеспечивает возможность автоматизации, высокую производительность, гибкость и переносимость.

1.3 Реализация метода

Рассмотрим реализации алгоритмов, описанных в конструкторском разделе.

1.3.1 Создание блоков

Проверяем, если директория пустая, то завершаем алгоритм, иначе - начинаем обработку каждого файла.

Листинг 1.1 – Алгоритм создания блоков. Часть 1.

```
func addFiles(cmd *cobra.Command, args []string) {
    dir := args[0]
    files, err := ioutil.ReadDir(dir)
    if err != nil {
        fmt.Printf("Error reading directory: %v\n", err)
        os.Exit(1)
    }
    if len(files) == 0 {
        fmt.Printf("Backup directory is empty")
        return
    }
    numBlocks := calculateNumBlocks(fileInfo.Size(), len(cfg.
       Servers))
    blockSize := calculateBlockSize(fileInfo.Size(), numBlocks)
    for _, file := range files {
        if file.Mode().IsRegular() {
            addFile(filepath.Join(dir, file.Name()), &cfg,
               numBlocks, blockSize)
        }
    }
}
```

Считываем файл, затем делим его на блоки размера *blockSize*. Размер блока определяется функцией *calculateBlockSize*.

Листинг 1.2 – Алгоритм определения размера блока

```
func calculateBlockSize(fileSize int64, numBlocks int) int64 {
    return int64(RoundUpToNearestMultiple(int(fileSize),
       numBlocks) / numBlocks)
}

func RoundUpToNearestMultiple(n, k int) int {
    return ((n + k - 1) / k) * k
}
```

Для вычисления размера блока используется алгоритм округления вверх до ближайшего кратного.

Находится такое число X, которое больше числа N и кратно ему, но при этом минимально. В данном случае N — размер файла, k — количество серверов.

Такое вычисление размера позволяет гарантировать, что последний блок всегда будет меньше остальных. При создании суперблока это позволяет дописывать нули в конец последнего блока , чтобы размер блоков был одинаковый.

Затем считается чек-сумма, сохраняются блоки и информация о них в конфигурационный файл.

Листинг 1.3 – Алгоритм создания блоков. Часть 2.

```
func addFile(path string, cfg *config.Config, numBlocks int64,
   blockSize int64) {
    filePath := path
    file, err := os.Open(filePath)
    if err != nil {
        fmt.Printf("Error opening file: %v\n", err)
        os.Exit(1)
    }
    defer file.Close()
    fileInfo, err := file.Stat()
```

Листинг 1.4 – Алгоритм создания блоков. Часть 3.

```
    if err != nil {
        fmt.Printf("Error getting file information: %v\n", err)
        os.Exit(1)
    }
    checkSum, err := fileChecksum(file)
    if err != nil {
        fmt.Printf("Error getting file checksum: %v\n", err)
        os.Exit(1)
    }
    blocks := make([]config.FileBlock, 0)

    for i := 0; i < numBlocks; i++ {
        offset := int64(i) * blockSize
        blockSize := blockSize
        if i == numBlocks-1 {
            blockSize = fileInfo.Size() - offset
        }
        checkSum, err := blockChecksum(file, offset, blockSize)
        if err != nil {
            fmt.Printf("Error getting block checksum: %v\n", err)
            os.Exit(1)
        }
        blockPath := filepath.Join(blockPath, fmt.Sprintf("%s.%d
           ", fileName[len(fileName)-1], i))
        fileName := strings.Split(filePath, "/")
        block := config.FileBlock{
            Id:       i,
            Size:     int(blockSize),
            Checksum: checkSum,
            Location: blockPath,
        }
        blocks = append(blocks, block)
        err = os.MkdirAll(filepath.Dir(blockPath), 0755)
        if err != nil {
            fmt.Printf("Error creating directory: %v\n", err)
            os.Exit(1)
        }
```

Листинг 1.5 – Алгоритм создания блоков. Часть 4.

```
        err = saveBlockToFile(file, offset, blockSize, blockPath)
        if err != nil {
            fmt.Printf("Error saving block to file: %v\n", err)
            os.Exit(1)
        }
    }
    fileInf := config.FileInfo{
        Filename:  filepath.Base(filePath),
        Size:      fileInfo.Size(),
        BlockSize: blockSize,
        Checksum:  checkSum,
        Blocks:    blocks,
        Date:      time.Now().Format("2006-01-02 15:04:05"),
    }
    cfg.Files = append(cfg.Files, fileInf)
    err = saveConfig(*cfg)
    if err != nil {
        fmt.Printf("Error saving config file: %v\n", err)
        os.Exit(1)
    }
    fmt.Println("File info and blocks saved successfully.")
}
```

1.3.2 Создание и восстановление суперблока

Для создания суперблока проходим по всем файлам из конфигурационного файла, выбираем сервер для сохранения суперблока, создаем суперблок при помощи XOR блоков, добавляя нули в конец последнего блока, затем сохраняем информацию о суперблоке в конфиг, а сам суперблок на сервер.

Листинг 1.6 – Алгоритм создания и восстановления суперблока. Часть 1.

```
func backup(config *config.Config) error {
    for _, server := range config.Servers {
        err := os.MkdirAll(server.Path, os.ModePerm)
        if err != nil {
            return fmt.Errorf("failed to create directory %s: %v
               ", server.Path, err)
        }
    }
    serverIndex := 0
    for id, file := range cfg.Files {
        xorData := xorBlocks(file.Blocks)
        backupDir := filepath.Join(cfg.Servers[serverIndex].Path)

        err := os.MkdirAll(backupDir, os.ModePerm)
        if err != nil {
            return fmt.Errorf("failed to create backup directory:
                %v", err)
        }
        // Create superblock
        superblock := superBlock{
            ID:     id,
            Blocks: xorData,
        }
        // Save superblock in file
        err = writeSuperblock(backupDir, file.Filename, &
           superblock)
        if err != nil {
            return fmt.Errorf("failed to write superblock for %s:
                %v", file.Filename, err)
        }
```

Листинг 1.7 – Алгоритм создания и восстановления суперблока. Часть 2.

```
        // Save information about superblock to config
        updateBlockLocationInConfig(&cfg, backupDir, file.
           Filename, file.Blocks)
        // Calculate next server by Round-Robin
        serverIndex = (serverIndex + 1) % len(cfg.Servers)
    }
    // Save new config version
    err := saveConfig(cfg)
    if err != nil {
        return fmt.Errorf("failed to save config: %v", err)
    }

    return nil

}
```

Функция для создания суперблока через XOR его блоков. Для последнего блока добавляются нули, выравнивающие размер блока.

Листинг 1.8 – Функция создания суперблока через XOR. Часть 1.

```
func xorBlocks(blocks []config.FileBlock) [][]byte {
    if len(blocks) == 0 {
        return nil
    }
    blocksAll := make([][]byte, 0, len(blocks))
    for _, block := range blocks {
        blockData, err := readBlock(block.Location)
        if err != nil {
            log.Fatal("error read block")
        }
        blocksAll = append(blocksAll, blockData)
    }
    blockSize := len(blocksAll[0])
    xorBlocks := make([][]byte, len(blocks))
    for i, block := range blocksAll {
        xorBlock := make([]byte, blockSize)
        if i == 0 {
            copy(xorBlock, block[:blockSize])
```

Листинг 1.9 – Функция создания суперблока через XOR. Часть 2.

```
        } else {
            if i == len(blocksAll)-1 {
                block = addPadding(block, blockSize)
            }
            for j, b := range block {
                xorBlock[j] = b ^ blocksAll[i-1][j]
            }
        }
        xorBlocks[i] = xorBlock
    }
    return xorBlocks
}

func addPadding(data []byte, blockSize int) []byte {
    padLen := blockSize - len(data)
    fmt.Println("padLen=", padLen)
    padding := bytes.Repeat([]byte{0}, padLen)
    return append(data, padding...)
}
```

1.3.3 Восстановление файла

Для восстановления файла нужна информация о нем и о суперблоки. Такая информация берется из конфигурационного файла, после чего начинается процесс восстановления — выполняется обратный XOR с удаланием избыточных нулей из последнего блока.

Листинг 1.10 – Алгоритм восстановления файла. Часть 1.

```
func restore(config *config.Config) error {
    for _, superblock := range config.SuperBlocks {
        // Read superblock from file
        superblockData, err := readSuperblock(superblock.Location)
        if err != nil {
            return fmt.Errorf("failed to read superblock %s: %v", superblock.Location, err)
        }
        // Get filename from superblock location
        fileName := getLastWordWithoutExtension(superblock.Location)

        // Create restore directory if not exists
        restoreDir := "./restore"
        err = os.MkdirAll(restoreDir, os.ModePerm)
        if err != nil {
            return fmt.Errorf("failed to create restore directory: %v", err)
        }

        // Create destination file
        restorePath := filepath.Join(restoreDir, fileName)
        file, err := os.Create(restorePath)
        if err != nil {
            log.Fatal(fmt.Errorf("failed to create file %q: %v", restorePath, err))
        }
```

Листинг 1.11 – Алгоритм восстановления файла. Часть 2.

```
        prevBlock := make([]byte, superblock.BlockSize)
        for i := 0; i < len(superblock.Blocks); i++ {
            if i == len(superblock.Blocks)-1 {
                //fmt.Println("block  restore ", sb.Blocks[i-1])
                fmt.Println("check padding")
                xorBlock(superblockData.Blocks[i], prevBlock)
                superblockData.Blocks[i] = superblockData.Blocks[
                   i][:superblock.Blocks[len(superblock.Blocks)
                   -1].Size]
            } else {
                log.Println(fmt.Sprintf("block num = %v", i))
                xorBlock(superblockData.Blocks[i], prevBlock)
                prevBlock = superblockData.Blocks[i]
            }
            _, err = file.Write(superblockData.Blocks[i])
            if err != nil {
                file.Close()
                os.Remove(restorePath)
                log.Fatal(fmt.Errorf("failed to write block to
                   file %q: %v", restorePath, err))
            }
        }
    }
    return nil
}
```

1.4 Пример использования разработанного метода

Рассмотрим сценарий использования.

В директории есть несколько файлов:

- 1 текстовый файл;
- 3 картинки;
- 1 видеозапись.

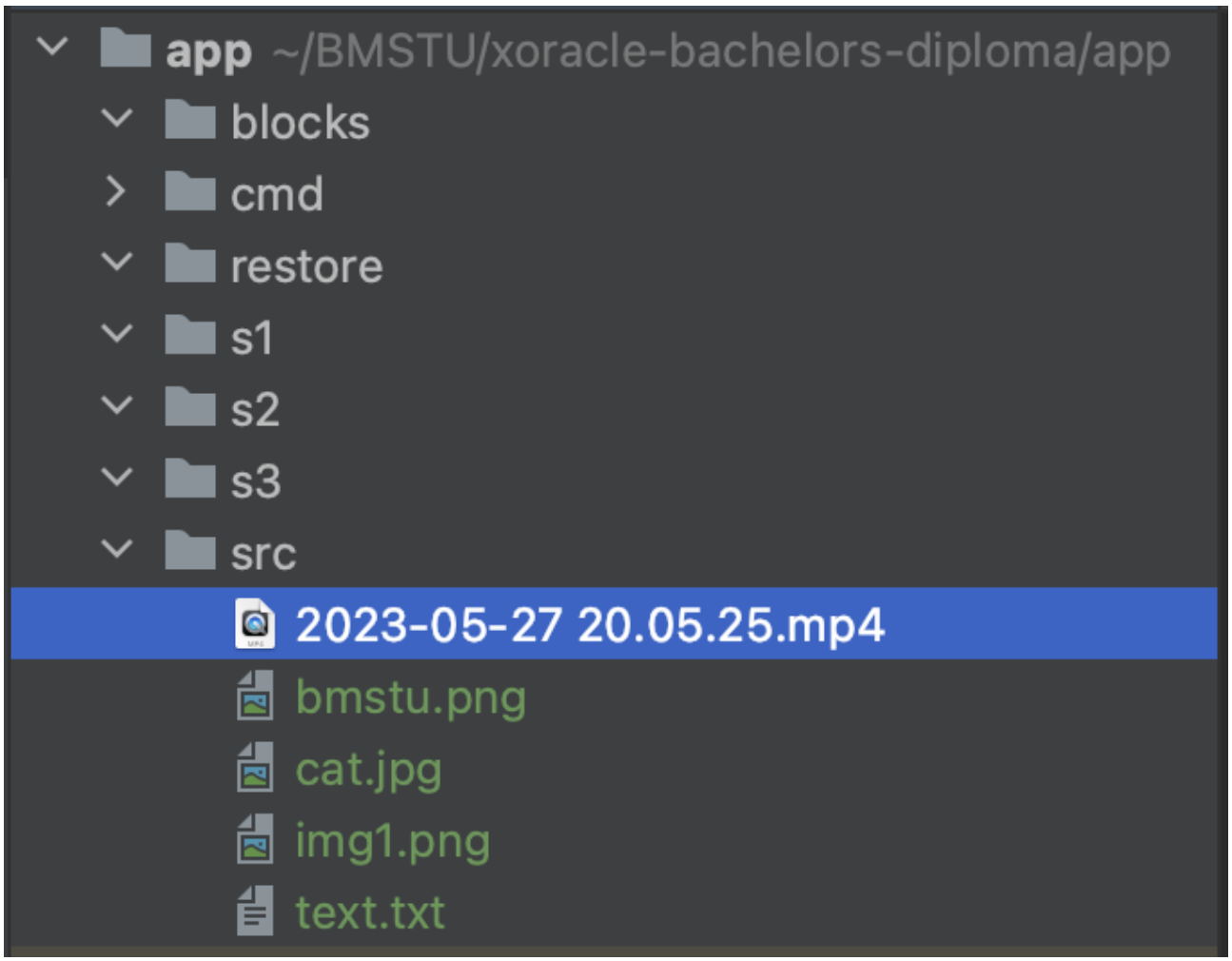

Рисунок 1.2 – Исходные файлы для создания резервной копии

Далее соберем и воспользуемся программой. При запуске без команд выводится справка.

```
➜  app git:(main) ✗ go build ./cmd/backupcli
➜  app git:(main) ✗ ./backupcli
A CLI utility for working with files

Usage:
  backupcli [command]

Available Commands:
  add         Add file info and file blocks
  backup      Perform file backup
  completion  Generate the autocompletion script for the specified shell
  help        Help about any command
  info        Verify file checksums
  restore     Restore file backup to file

Flags:
  -h, --help   help for backupcli

Use "backupcli [command] --help" for more information about a command.
```

Рисунок 1.3 – Сборка программы и вывод справки

Первым шагом нужно разбить файлы на блоки. Для этого вызовем команду *backupcli add –dir=./src*

```
➜  app git:(main) ✗ ./backupcli add --dir=./src
File src/2023-05-27 20.05.25.mp4 blocks and info saved successfully.
File src/bmstu.png blocks and info saved successfully.
File src/cat.jpg blocks and info saved successfully.
File src/img1.png blocks and info saved successfully.
File src/text.txt blocks and info saved successfully.
➜  app git:(main) ✗
```

Рисунок 1.4 – Команда add

В результате выполнения команды в папку *./blocks* были сохранены блоки, на которые были поделены исходные файлы. Каждый файл разбился на 3 блока — по количеству доступных серверов.

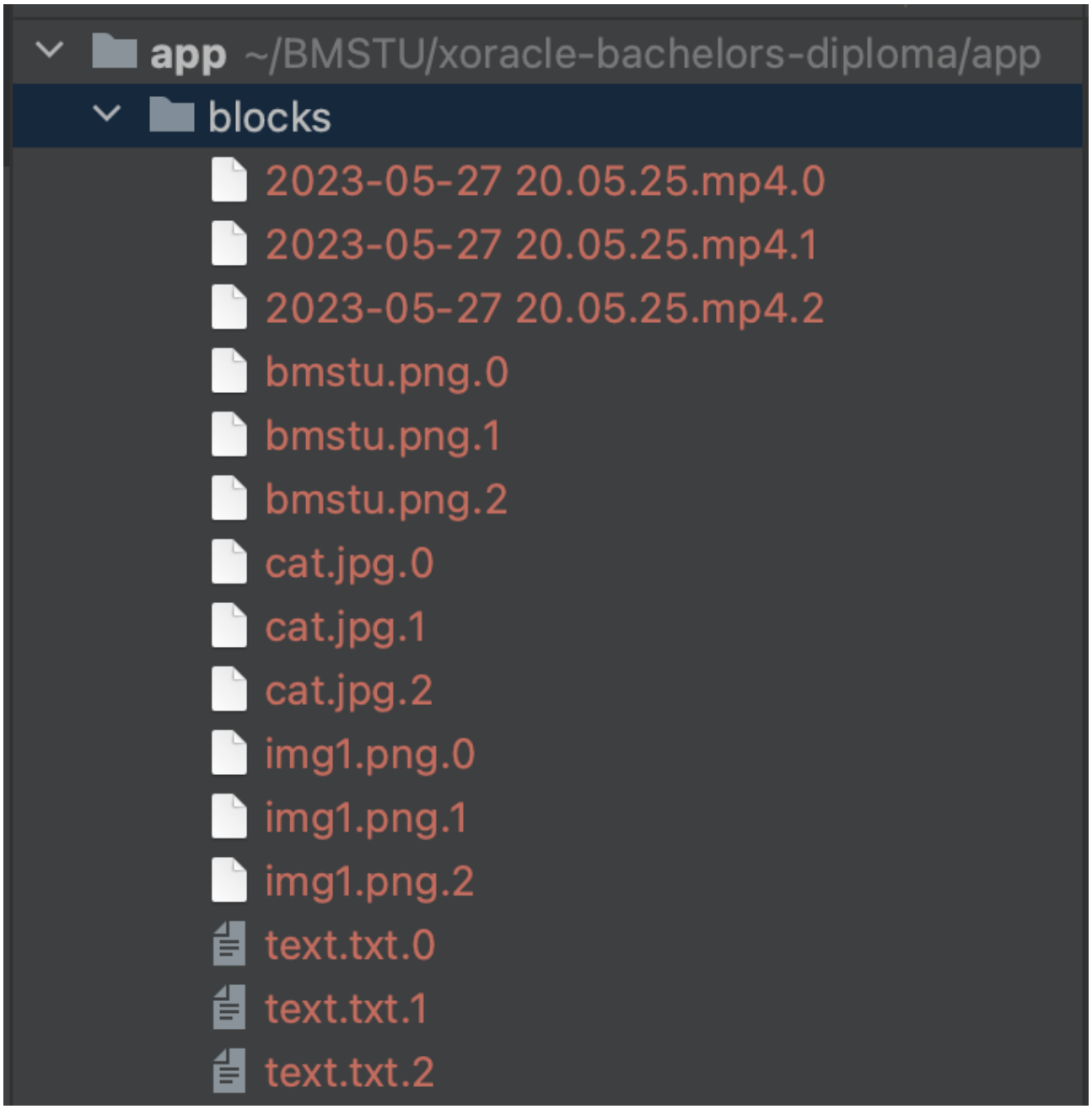

Рисунок 1.5 – Файлы, разбитые на блоки

Далее вызовем команду *backupcli backup*, которая создаст резервные копии для файлов и распределит их по серверам.

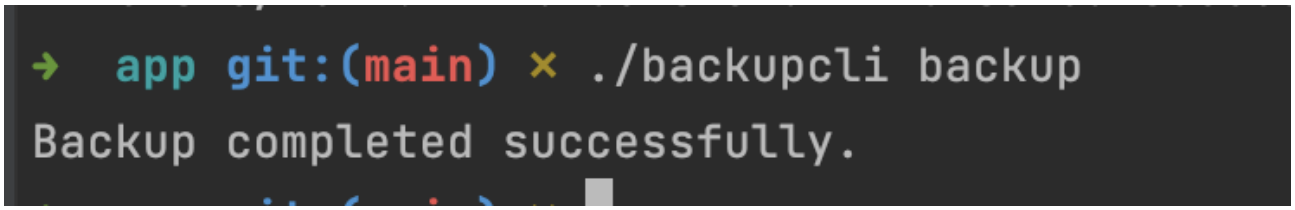

Рисунок 1.6 – Файлы, разбитые на блоки

Получившиеся суперблоки распределяются по серверам по алгоритму Round-Robin.

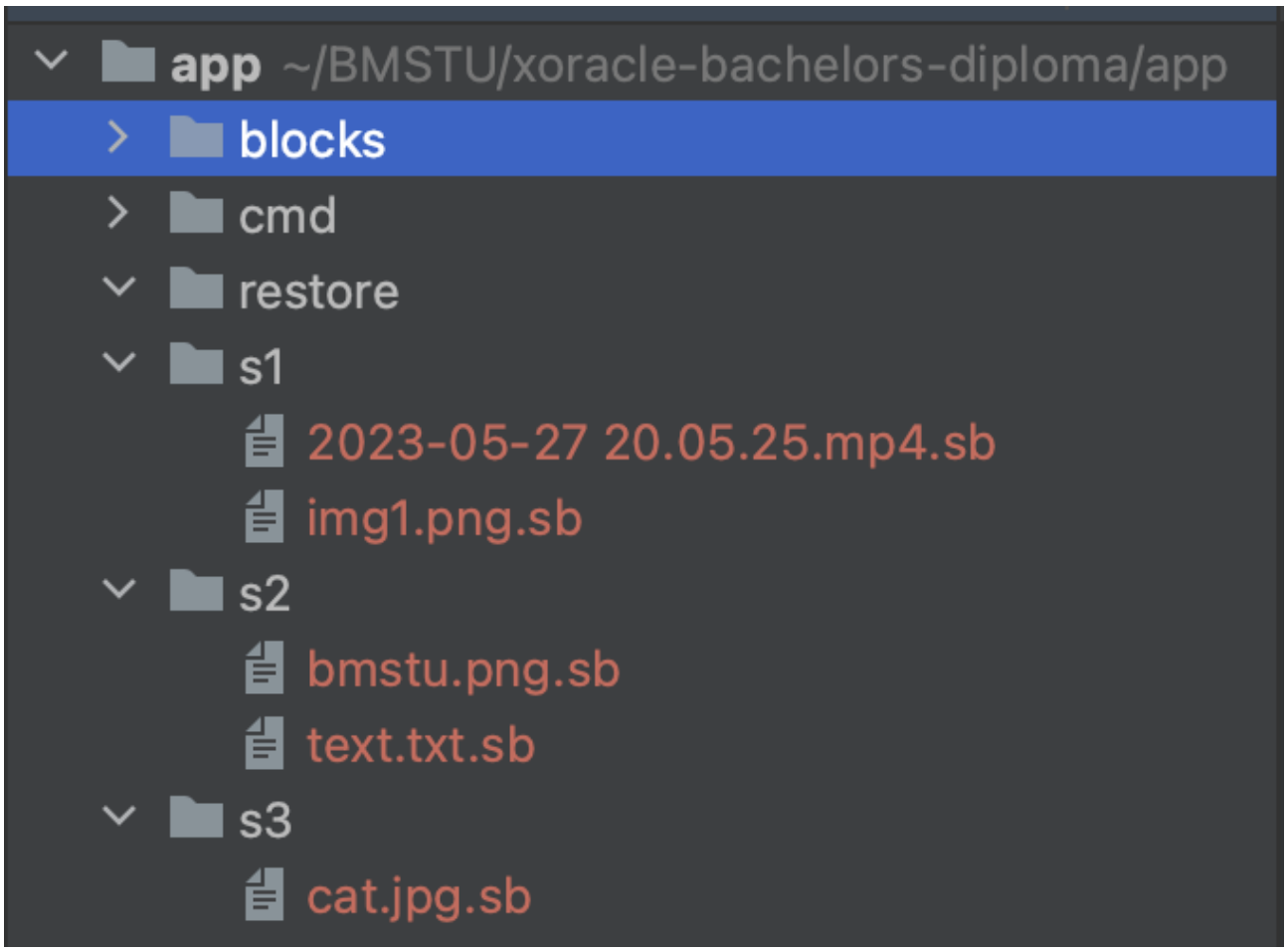

Рисунок 1.7 – Суперблоки, распределенные по серверам

Проверим целостность файлов: являются файлы из директории */src* равными файлам, которые хранятся в виде суперблоков на серверах.

Для этого вызовем команду *backupcli info*, которая посчитает чек-суммы исходных файлов и сравнит с теми, что хранятся в конфигурационном файле.

```
→  app git:(main) × ./backupcli info --dir=./src
Checksum match for file 2023-05-27 20.05.25.mp4
Checksum match for file bmstu.png
Checksum match for file cat.jpg
Checksum match for file img1.png
Checksum match for file text.txt
Check files completed successfully.
```

Рисунок 1.8 – Команда info

Поменяем один из файлов, для удобства сделаем это с текстовым файлом.

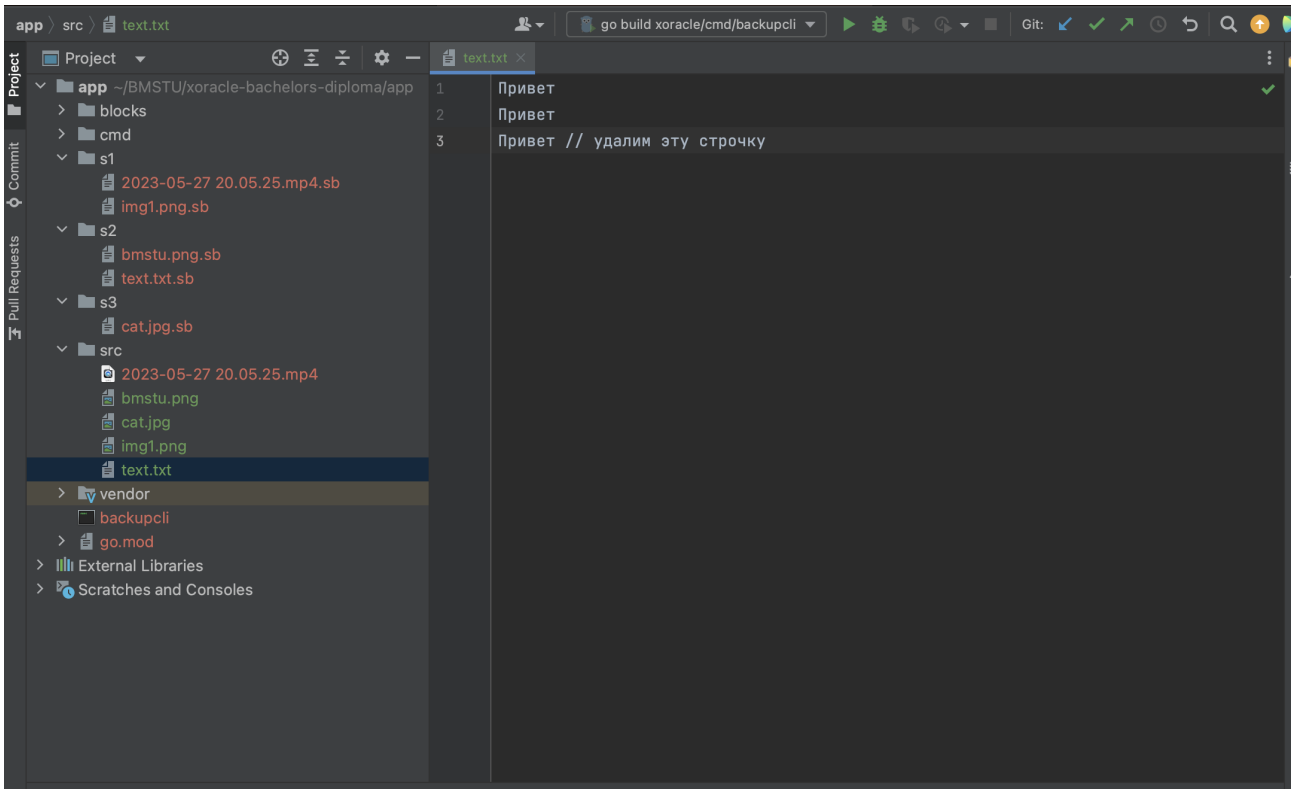

Рисунок 1.9 – Изменение исходного файла

Вызовем *backupcli info* и увидем, что исходный файл изменился, чек-суммы не сходятся.

Рисунок 1.10 – Команда info, файлы не равны

Восстановим этот файл - воспользуемся *backupcli restore*.

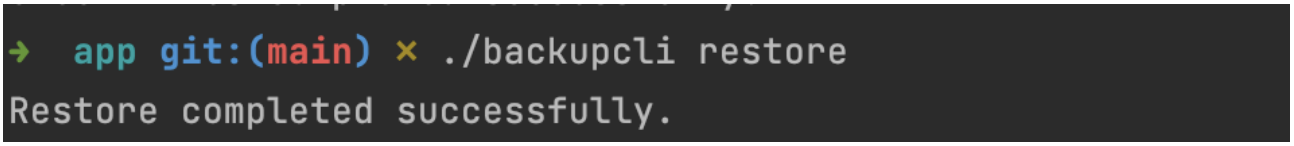

Рисунок 1.11 – Команда restore

Посмотрим на diff исходного файла, который был изменен и восстановленного.

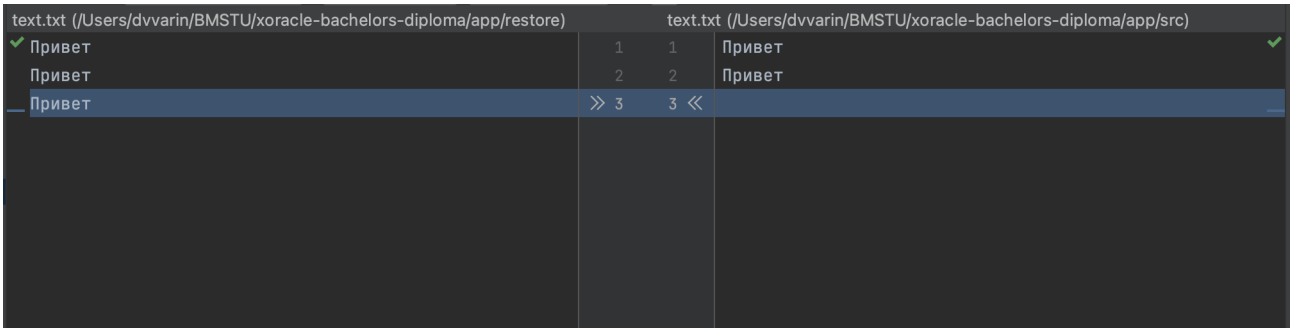

Рисунок 1.12 – Разница в исходном файле и восстановленном

Восстановленный файл отличается от исходного одной строкой, которую мы ранее удалили для проверки восстановления.

Перепишем исходный файл восстановленным в папке *./src* и снова проверим файлы на целостность.

```
➜  app git:(main) ✗ ./backupcli info --dir=./src
Checksum match for file 2023-05-27 20.05.25.mp4
Checksum match for file bmstu.png
Checksum match for file cat.jpg
Checksum match for file img1.png
Checksum match for file text.txt
Check files completed successfully.
```

Рисунок 1.13 – Команда info, файлы равны

1.5 Тестирование

Тестирование метода производилось несколькими способами:

- юнит тестирование - покрыть основные функции тестами;
- ручное тестирование.

Для ручного тестирования использовался следующий сценарий:

1. создать несколько тестовых файлов, которые будут использоваться в процессе резервного копирования;
2. создать резервные копии через команду *backup*;
3. восстановить файлы через команду *restore*;
4. проверить различия файлов (diff) визуально, или с использованием специальных программ, которые позовляют найти отличия в файлах.

Для юнит тестирования будут использоваться встроенные в язык программирования инструменты.

1.6 Выводы

В данном разделе:

- были выбраны средства реализации метода резервного копирования;
- было разработано программное обеспечение, реализующее разработанный метод;
- приведен пример использования разработанного метода;
- осуществлено тестирование метода, для которого использовано два типа тестов — юнит и ручные тесты.

ЗАКЛЮЧЕНИЕ

В рамках данной работы:

- были выбраны инструменты и технологии, требуемые для реализации метода;
- был реализован новый метод в виде программного комплекса;
- приведен пример использования разработанного метода;
- были описан методы тестирования.

Таким образом, цель работы — реализовать метод резервного копирования с контролируемым размером избыточной информации в распределенном файловом хранилище была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. The Go Programming Language. — URL: https://go.dev/ (дата обр. 24.05.2023).
2. Командная строка. — URL: https://help.ubuntu.ru/wiki/%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0 (дата обр. 24.05.2023).
3. A Commander for modern Go CLI interactions. — URL: https://github.com/spf13/cobra (дата обр. 24.05.2023).
4. Crontab command. — URL: https://www.ibm.com/docs/en/aix/7.2?topic=c-crontab-command (дата обр. 24.05.2023).