

目次

第 1 章	シス研というサークルについて	3
1.1	はじめに	3
1.2	シス研とは	3
1.3	この本について	6
1.4	まとめ	6
第 2 章	DiscordBot を作ってみよう!	7
2.1	DiscordBot を作ってみよう	7
2.2	実行環境・使用技術	8
2.3	ローカル環境で Bot が動作するようにする	8
2.4	まとめ	12
第 3 章	リポジトリ作成後に設定しておきたいこと	13
3.1	はじめに	13
3.2	ファイルの設定	14
3.3	インフラ回り	17
3.4	おわりに	19
第 4 章	Gin × Neo4j × Docker で最短経路を返してくれる API サーバを建てる	20
4.1	Gin × Neo4j × Docker で最短経路を返してくれる API サーバを建てる . . .	20
4.2	今回扱うデータの図	21
4.3	ディレクトリ構造	22
4.4	Neo4j に最初にインポートするファイル	23
4.5	Go のソースコード	30
4.6	実行方法	36
4.7	おわりに	36
第 5 章	Next.js 13 触ってみた	37

5.1	はじめに	37
5.2	プロジェクト作成からサーバ起動	37
5.3	Layout と Head	39
5.4	React Server Components	44
5.5	サーバーコンポーネントでデータを取得	46
5.6	ローディング UI 表示	48
5.7	エラーハンドリング	53
5.8	終わりに	56
第 6 章	これは chapter	57
6.1	これは section	57
第 7 章	これは chapter	59
7.1	これは section	59
第 8 章	これは chapter	61
8.1	これは section	61

1

シス研というサークルについて

1.1 はじめに

初めまして！シス研会長の林です！はい、ここでみなさんシス研とはなんぞ？となっていると思うのでまずは自分が水先案内人となりましてシス研とこの本について解説していこうと思います。

1.1.1 どんな話をするのか

シス研ってどんなサークル？どんな活動をしているの？この本はどういったもの？といったものを紹介していきます。それでは、さっそく行ってみましょう！！！

1.2 シス研とは

シス研は正式名称を「システム工学研究会」と言い、愛知工業大学公認の情報系サークルです。歴史は長く、2023年で創立47周年を迎え、かのAppleと同じ年となります！！
シス研ではハッカソン出場をはじめとしたチーム開発、ゲーム作成、インフラの構築、運用などを行っています。

1.2.1 どんな活動をしているの？

シス研の主な活動はチーム開発とインフラ整備です。サークル全体としての開発物などはなく、それぞれがチームを組んでハッカソンに出場したりしています。

インフラ面では、部室に物理サーバを持っており、そこでシス研のホームページ^{*1}や各種サービス^{*2}を公開しています。23年4月現在、大幅な工事を行なっておりごく一部のサービスのみ稼働しています（すみません）。そのほかにもシス研主催のLT会・ハッカソンの開催、STECH様^{*3}と共同でQiita Advent Calendar 2022^{*4}への参加もしています。



▲図1.1 部室の様子

1.2.2 2021～2022年度の活動実績

- 2021 愛工大大学祭 工科展 最優秀賞
- 2022 技育 CAMP vol3 努力賞
- 2022 技育 CAMP vol4 努力賞
- 2022 技育博 参加
- 2022 技育 CAMP vol8 優秀賞

*1 シス研ホームページ <https://set1.ie.aitech.ac.jp>

*2 シス研紹介ページ <https://welcome.sysken.net>

*3 STECH様 HP <https://stech.careerselect.jp>

*4 Qiita Advent Calendar 2022 <https://qiita.com/advent-calendar/2022/stech-ait-advent>

- 2022 技育 CAMP vol9 努力賞
- 2022 愛工大大学祭 工科展 瑞若賞
- 2022 技育展 出展
- 2022 愛工大大学祭 模擬店 最優秀賞
- 2022 Open Hack U 2022 ONLINE 参加
- 2022 技育 CAMP アドバンス 登壇
- 2023 Open Hack U 2022 Spring ONLINE 参加
- 長期休暇中の LT 会、ハッカソン主催
- 各種勉強会の開催

シス研の設備

- ラックマウントサーバ、ネットワーク機器
- デスクトップ PC
- iMac,MacBook
- iPhone,iPad
- Android 端末各種
- Raspberry Pi
- はんだ等の電子工作セット
- その他多数...



▲図1.2 ラックマウントサーバ



▲図1.3 ネットワーク機器



▲図1.4 タブレット端末



▲図1.5 Raspberry Pi

1.3 この本について

この本はシス研のメンバーが経験したこと、取り組んだことのアウトプットを目的としたものです。この本を通じて皆さんにはシス研のメンバーは具体的にどのような活動をしているのか知ってもらいたいと思います。

また、シス研として本を出すのは今回が初めてなのでどうか温かい目で見ていただけると幸いです。

1.4 まとめ

ここまでお話を来てきましたが簡単にでもシス研について知ってもらうことはできたでしょうか？うまく伝えることができていたらとても嬉しいです。

次のページからはメンバーの記事本編になります！シス研初めての本をよろしくお願いします！！

2

技術記事のまとめ

- suda

p.7 『DiscordBot を作ってみよう』

似たような会話をするのが煩わしいので自分っぽい反応をする DiscordBot を作ってみました！

- hihumikan

p.11 『リポジトリ作成後に設定しておきたいこと』

Git 初心者に向けた Git で使えると嬉しいことをまとめました！

- 水谷

p.16 『Gin × Neo4j × Docker で最短経路を返す API サーバを建てる』

グラフ DB である Neo4j と Go(Gin) を用いた経路探索 API と、その API サーバを構築します！！

- Beyond Toyama

p.29 『Next.js 13 触ってみた』

最新の Next.js 13 になったことで追加された機能や変更された部分を紹介しています！

3

DiscordBot を作ってみよう

3.1 はじめに

はじめて、suda です。私は Discord で人とチャットをしている時に同じ会話が頻繁に続き、これ Bot で返事をするようにしたら返事をする手間が省けるし面白いのでは？と思い Bot を作ることにしました。発想がひどいって！？まあでも自分の発想したものを形にすることが面白いことだと思うので今回はそこには目を瞑りましょう…もちろん自分が送ったメッセージに対して Bot に返答させることもできるので、自分だけのオリジナル DiscordBot を作ってみましょう！

3.2 何を作るのか

Discord のサーバで特定のメッセージが来たら、特定のメッセージを返す Discord の Bot を作ります。サンプルプログラムを参照したい方は以下の URL からご覧下さい。^{*1} 例えば自分が「仕事終わった」と言うと Bot が「お疲れ様」と返してくれます。

^{*1} 今回作る DiscordBot のサンプルプログラム https://github.com/sudamichiyo/Discord_Bot_sampleprogram

3.3 実行環境・使用技術

- Python 3.10.8

3.4 ローカル環境で Bot が動作するようにする

まずはローカル環境で Bot が動作するようにしてみます。

3.4.1 Bot の作成・管理をする

初めに、機能などはまだついていない Bot を Discord のポータルサイトから作成します。Discord の Bot の作り方 (メモ) という記事の「1.Discord 上の Bot の作成」を見ながら Bot を作成してみて下さい。^{*2}

3.4.2 ファイルの作成

Bot を実行する Python ファイルを作ります。

```
$ mkdir message_discord_bot  
$ cd message_discord_bot  
$ touch main.py
```

3.4.3 discord.py の準備

ここからはライブラリ (discord.py) のドキュメントを見ながら環境構築をしていきます。^{*3} Python で Discord の API を操作するために必要なライブラリをインストールします。先ほど作成したディレクトリにアクセスして、以下のコマンドで discord.py をインストールします。

```
$ python3 -m pip install -U discord.py
```

^{*2} Discord の Bot の作り方 (メモ)<https://note.com/exteo/n/nf1c37cb26c41>(参照 2023.3.29)

^{*3} discord.py ドキュメント <https://discordpy.readthedocs.io/ja/latest/intro.html#basic-concepts>(参照 2023.3.29)

次に、先ほど作成した main.py を以下のソースコードに書き換えます。

```
1 import discord
2
3 class MyClient(discord.Client):
4     async def on_ready():
5         print(f'Logged on as {self.user}!')
6
7     async def on_message(self, message):
8         print(f'Message from {message.author}
9             : {message.content}')
10
11 intents = discord.Intents.default()
12 intents.message_content = True
13
14 client = MyClient(intents=intents)
15 client.run('my token goes here')
```

ここで、以下のボットに関する 2 つの設定を Discord のポータルサイトから設定してください。

- ポータルサイトの「Bot」からトークンを取得する
- ポータルサイトの「Bot」の「MESSAGE CONTENT INTENT」を有効にする

'my token goes here' は取得した Bot のアクセストークンを書きます。以上の設定が終わったら python3 main.py を実行すると、Bot のサーバが立ち上がります。Bot のいるサーバの任意のチャンネルでメッセージを投稿すると、コマンドライン上に「書いた人」と「メッセージ」がそのまま出力されます。

3.4.4 環境変数の設定

ソースコードに直接トークンを書いてしまうと、GitHub でソースコードをホスティングするときにトークンキーが他の人にバレてしまいます。これを防ぐために.env ファイルを作成して、その中に Discord のアクセストークンを書きます（下記参照）。

```
1 DISCORD_TOKEN='My token goes here'
```

Python の中で.env ファイルに書かれている変数を取得するために dotenv というライブラリを使用します。以下のようにインストールします。

```
$ pip3 install python-dotenv
```

インストール後に main.py に下記のコードを付け加えて下さい。
main.py の import discord と class MyClient の間に以下のコードを追加します。

```
1 import os  
2 from dotenv import load_dotenv  
3 load_dotenv()
```

そして、最後の行を以下のように書き換えて下さい。

```
1 client.run(os.environ['DISCORD_TOKEN'])
```

書き換えたあとに python3 main.py を実行すると、先ほどと同じようにメッセージの受け取りをしてくれるサーバーサイドアプリケーションが立ち上がります。

3.4.5 Bot が特定のワードに反応して、特定のメッセージを返答する機能をつける

プログラムを起動して正常にサーバーサイドアプリケーションがメッセージを受け取れるようになったら、Bot が特定のワードに反応して、特定のメッセージを返答する機能をつけていきます。Bot に機能をつけるには上記のソースコードの 8 行目と 10 行目の間に以下のコードを付け足していきます。

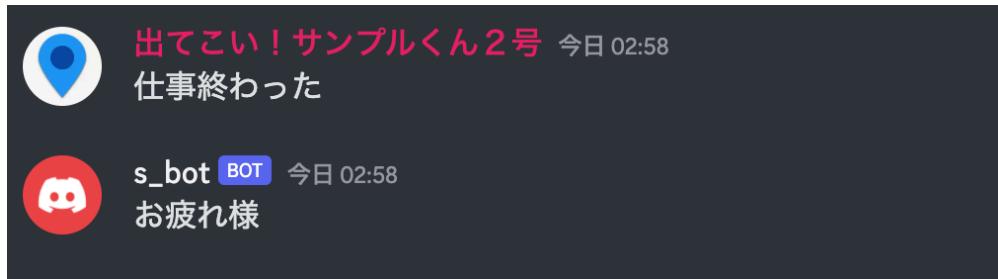
```
1 # メッセージを書いた人が Bot なら処理終了  
2 if message.author.bot:  
3     return  
4 channel = message.channel  
5 if message.content == '仕事終わった':  
6     await channel.send('お疲れ様')
```

付け足したコードの解説をしていきます。

2,3 行目でメッセージを書いた人が Bot なら処理を終了させています。

4 行目でメッセージが投稿されたチャンネルを取得しています。

5 行目の message.content はメッセージの内容で、今回の場合「仕事終わった」というメッセージをチャンネルに投稿すると、メッセージが投稿されたチャンネルに Bot が「お疲れ様」と返答します。



▲ 図 3.1 動作例

3.5 まとめ

今回は Discord の Bot の作り方を説明しました。上記の「仕事終わった」や「お疲れ様」に当たる部分を変えたりして自分好みに改良してみて下さい。

4

リポジトリ作成後に設定しておきたいこと

4.1 はじめに

こんにちは。hihumikan です。

本チャプターでは「リポジトリ作成後に設定しておきたいこと」をご紹介します。私自身が次回、プロジェクトを開始する際に、こういった設定や技術を使うだろうというものをまとめました。

ただし、これら全ての内容をプロジェクトに適用出来るものではないため、ご利用の環境や用途に合わせて利用していただければと思います。

4.1.1 対象読者

対象読者は、Git/GitHub を利用した開発を行う初学者の方を想定しています。

4.2 ファイルの設定

リポジトリ作成後に設定しておきたい「ファイルの設定」についてご紹介します。

4.2.1 .gitignore

.gitignore は、Git による管理から除外したいファイルやディレクトリを指定するための設定ファイルです。

例えば、MacOS の場合、ディレクトリ毎に.DS_Store というファイルが自動的に生成されます。このファイルは、ディレクトリの meta 情報を記録しますが、通常の開発において共有する必要がないため、Git の管理対象外としておくのが望ましいです。

また、.env などの環境変数を利用してプログラムを動かす場合、.env ファイルには、パスワードや秘密鍵などの情報が含まれているのが多いため、これも Git の管理対象外としておくのが望ましいです。

リスト 1.1 のようなテキストファイルを Git の管理下に置くだけで、Git から管理対象外として扱われます。

リスト 1.1 .gitignore

```
1 .DS_Store  
2 .env
```

リポジトリを共有する場合、.gitignore ファイルを置いておくだけで、開発メンバー全員が同じ設定を利用できるため、必要なファイルだけが Git の管理下に置かれるようになります。

4.2.2 .gitattributes

.gitattributes は、特定のファイルに対して Git の挙動を変更するための設定ファイルです。主に、改行コードやファイルの文字コードを指定するために利用されます。

改行コードを指定する理由としては、Windows と MacOS では改行コードが異なる点が挙げられます。Git で管理されているファイルの改行コードが一致しないと、差分が発生してしまい、想定した挙動と異なる動作をする可能性があります。それを防ぐために、.gitattributes に改行コードを明示しておくことで、安全に開発を進めれます。

リスト 1.2 のように、ファイルの拡張子に対して、改行コードを指定が出来ます。

リスト 1.2 .gitattributes

```
1 * text=auto  
2 *.sh text eol=lf
```

これも.gitignore と同様に.gitattributes 共有するだけで、開発メンバー全員が同じ設定を利用できます。

4.2.3 Makefile

Makefile は、コマンドをまとめて実行するための設定ファイルです。

利点として、開発メンバー全員が同じコマンドを実行出来る所にあります。環境構築やテストの実行など、手順が複雑な作業を一人一人が実行した場合、手順の違いによるエラーが発生する可能性があります。それらを防ぐために、Makefile にまとめておくことで、開発メンバー全員が同じコマンドを実行でき、人的ミスを防げます。

Makefile の例としては、リスト 1.3 のようなものです。

リスト 1.3 Makefile

```
1 up: ## API とデータベースを起動
2   docker compose -f docker-compose-db.yml -p db up -d
3   docker compose -f docker-compose-api.yml -p api up -d
4
5 build: ## サービスの構築
6   docker compose -f docker-compose-db.yml -p db build
7   docker compose -f docker-compose-api.yml -p api build
8
9 stop: ## サービスを停止
10  docker compose -f docker-compose-db.yml -p db stop
11  docker compose -f docker-compose-api.yml -p api stop
12
13 kill: ## サービスを強制停止
14  docker compose -f docker-compose-db.yml -p db kill
15  docker compose -f docker-compose-api.yml -p api kill
16
17 down: ## サービスの停止とコンテナの削除
18  docker compose -f docker-compose-db.yml -p db down
19  docker compose -f docker-compose-api.yml -p api down
20
21 restart: ## サービスの再起動
22  docker compose -f docker-compose-db.yml -p db restart
23  docker compose -f docker-compose-api.yml -p api restart
```

これらを実行する場合、シェルに

```
$ make up
```

と入力するだけで、長いコマンドであった 2,3 行目のコマンドが簡単に実行されます。リスト 1.3 の例は簡単なものですぐ、他にも Makefile 内に変数を定義できるため、変更が必要な箇所を変数に置き換えることで、コマンドの変更を容易に行えます。

4.3 インフラ周り

4.3.1 GitHub Actions

GitHub Actions は、GitHub 上で動作する CI/CD ツールです。簡単に言えば、GitHub リポジトリに関連するイベントに応じて、あらかじめ定義しておいたワークフローを仮想マシン上で実行するものです

用途としては、コードの静的解析やテストの実行、デプロイなどが挙げられます。その他にも、Pull Request に対して、テスト内容をコメントしてくれるなどの GitHub 上での機能を利用できます。

利用方法としては、リスト 1.4 のように、.github/workflows ディレクトリを作成し、その中に設定ファイルを作成します。

リスト 1.4 .github/workflows/pr.yml

```
1 on:
2   pull_request:
3     types: [opened]
4   name: Pull Request
5   jobs:
6     assignAuthor:
7       name: Assign author to PR
8       runs-on: ubuntu-latest
9       steps:
10      - name: Assign author to PR
11        uses: technote-space/assign-author@v1
```

上記の例では、Pull Request が作成された際に、Pull Request の作成者を Assignee に設定する設定ファイルの例です。

その他にも、リスト 1.5 のように、ssh 鍵を設定することで、リモートサーバーにアクセスができます。

リスト 1.5 .github/workflows/deploy.yml

```
1 name:CI
2 on:
3   push:
4     branches:
5       - main
6 jobs:
7   deploy:
8     runs-on: ubuntu-latest
9     steps:
10    - uses: actions/checkout@v2
11    - name: Install SSH Key for Deploy
12      uses: appleboy/ssh-action@master
13      with:
14        key: ${{ secrets.SK }}
15        host: ${{secrets.SSH_HOST}}
16        username: ${{secrets.SSH_USERNAME}}
17        port: ${{secrets.SSH_PORT}}
18        script: |
19          git pull
```

この他にも、コードを自動で整形して commit してくれるツールなどの様々なツールが存在します。調べてみると面白いかもしれません。

4.4 おわりに

本チャプターでは「リポジトリ作成後に設定しておきたいこと」をご紹介しました。ご紹介したのは一部分ですが、これらを設定しておくことで、開発の効率化や、開発メンバーのミスを防げます。ぜひ、設定してみてください。

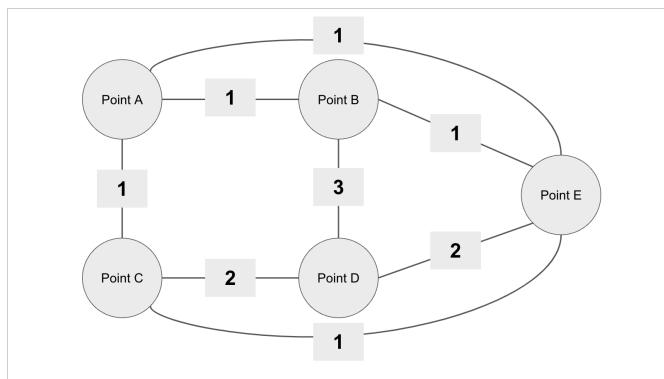
5

Gin × Neo4j × Docker で最短経路を返す API サーバを建てる

5.1 はじめに

このセクションでは、現在最もメジャーなグラフ DB である Neo4j と GitHub でも多くのスターを獲得している Go 言語の Web フレームワーク、Gin を用いて API サーバーを構築していきます。また、実行環境統一のため Docker を用います。

5.2 今回扱うデータの図



5.3 ディレクトリ構造

```
ディレクトリ構造

build
|   └── Docker
|       |   └── go
|       |       └── Dockerfile
|       └── neo4j
|           |   └── Dockerfile
|           └── volumes
|               └── import
|                   |   └── done
|                   └── points.csv
|                   └── route.csv
|               └── script
|                   └── import_data.sh
|   └── docker-compose.yml
└── server
    ├── config
    |   └── config.go
    └── environments
        └── neo4j.yml
    ├── controllers
    |   └── coordinate_controller.go
    ├── db
    |   └── neo4j.go
    ├── go.mod
    ├── go.sum
    ├── main.go
    ├── models
    |   └── coordinate.go
    ├── router
    |   └── router.go
    └── sample.http
```

5.4 Neo4j に最初にインポートするファイル

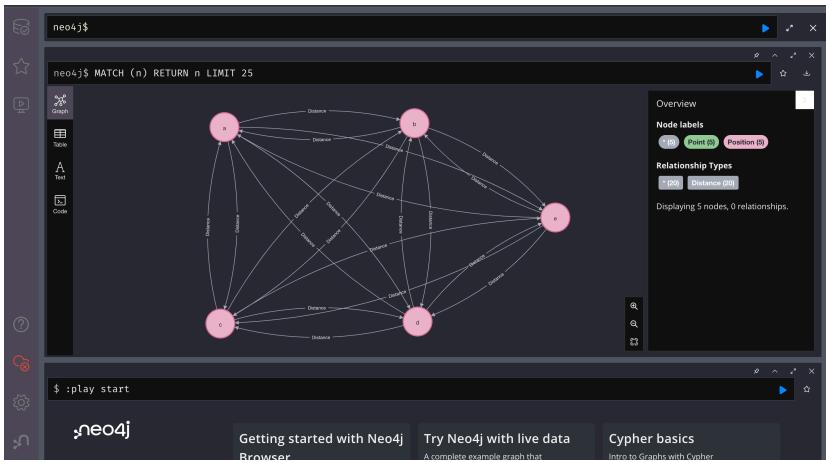
point.csv

```
1 point_id:ID,point_name,:LABEL
2 a,PointA,Point;Position
3 b,PointB,Point;Position
4 c,PointC,Point;Position
5 d,PointD,Point;Position
6 e,PointE,Point;Position
```

route.csv

```
:START_ID,:END_ID,:TYPE,cost:int
b,a,Distance,1
c,b,Distance,3
d,c,Distance,2
e,d,Distance,2
c,a,Distance,1
d,a,Distance,2
e,b,Distance,1
e,a,Distance,1
c,e,Distance,1
d,b,Distance,3
a,b,Distance,1
b,c,Distance,3
c,d,Distance,2
d,e,Distance,2
a,c,Distance,1
a,d,Distance,2
b,e,Distance,1
a,e,Distance,1
e,c,Distance,1
b,d,Distance,3
```

これらを Neo4j にインポートすることで、先ほどの図を表現することができます。



無事できましたね。

docker-compose.yml

```
1 version: "3"
2 services:
3   go:
4     container_name: NEO4JAPI_G0
5     build:
6       context: ./docker/go
7       dockerfile: Dockerfile
8     stdin_open: true
9     tty: true
10    volumes:
11      - ../../server:/server
12    ports:
13      - 8080:8080
14    networks:
15      app_net:
16        ipv4_address: 192.168.0.1
17    depends_on:
18      - "neo4j"
19
20 neo4j:
```

```

21   container_name: NE04JAPI_NE04J
22
23   build:
24     context: ./Docker/neo4j
25     dockerfile: Dockerfile
26
27   restart: always
28
29   ports:
30     - 57474:7474
31     - 57687:7687
32
33   volumes:
34     - ./Docker/neo4j/volumes/data:/data
35     - ./Docker/neo4j/volumes/logs:/logs
36     - ./Docker/neo4j/volumes/conf:/conf
37     - ./Docker/neo4j/volumes/import:/import
38     - ./Docker/neo4j/volumes/script:/script
39
40   environment:
41     - NEO4J_AUTH=neo4j/admin
42     - EXTENSION_SCRIPT=/script/import_data.sh
43
44   networks:
45     app_net:
46       ipv4_address: 192.168.0.2
47
48
49 networks:
50   app_net:
51     driver: bridge
52
53     ipam:
54       driver: default
55
56     config:
57       - subnet: 192.168.0.0/24

```

volumes はデータベースに入るデータ、ログをバインドするための場所を指定しています。environment ではユーザーとパスワード、最初に読み込んでもらうシェルスクリプトの場所を指定します。

go 関連の Dockerfile。

Dockerfile

```
1 # go バージョン
2 FROM golang:1.19.3-alpine
3 # アップデートと git のインストール
4 RUN apk add --update && apk add git
5 # app ディレクトリの作成
6 RUN mkdir /server
7 # ワーキングディレクトリの設定
8 WORKDIR /server
9 # ホストのファイルをコンテナの作業ディレクトリに移行
10 ADD . /server
11 # main.go を実行
12 CMD ["go", "run", "main.go"]
```

Neo4j 関連の Dockerfile。

Dockerfile

```
1 FROM neo4j:4.4.9
```

csv ファイルを読ませるためのシェルスクリプト。

import_data.sh

```
1 #!/bin/bash
2 set -euC
3
4 # EXTENSION_SCRIPT はコンテナが起動するたびにコールされるため、
5 # import 処理が実施済かフラグファイルの有無をチェック
6 if [ -f /import/done ]; then
7     echo "Skip import process"
8     return
9 fi
10
11 # データを全削除
12 echo "delete database started."
13 rm -rf /data/databases
14 rm -rf /data/transactions
15 echo "delete database finished."
16
17 # CSV データのインポート
18 echo "Start the data import process"
19 neo4j-admin import \
20   --nodes=/import/points.csv \
21   --relationships=/import/route.csv
22 echo "Complete the data import process"
23
24 # import 処理の完了フラグファイルの作成
25 echo "Start creating flag file"
26 touch /import/done
27 echo "Complete creating flag file"
```

5.5 Go のソースコード

5.5.1 config ディレクトリ

config.go

```
1 package config
2
3 import (
4     "github.com/spf13/viper"
5 )
6
7 var n *viper.Viper
8
9 func init() {
10    n = viper.New()
11    n.SetConfigType("yaml")
12    n.SetConfigName("neo4j")
13    n.AddConfigPath("config/environments/")
14 }
15
16 func GetNeo4jConfig() *viper.Viper {
17    if err := n.ReadInConfig(); err != nil {
18        return nil
19    }
20    return n
21 }
```

このファイルで neo4j.yaml の環境変数を読み込みます。

neo4j.yml

```
1 neo4j:
2   user: neo4j
3   password: admin
4   uri: neo4j://192.168.176.1:57687
```

Neo4j と接続するための環境変数です。

5.5.2 db ディレクトリ

neo4j.go

```
1 package db
2
3 import (
4     "log"
5
6     "neo4j/api/server/config"
7
8     "github.com/neo4j/neo4j-go-driver/v4/neo4j"
9 )
10
11 func GetDriverAndSession() neo4j.Session {
12     n := config.GetNeo4jConfig()
13     dr, err := neo4j.NewDriver(n.GetString("neo4j.uri"),
14         neo4j.BasicAuth(n.GetString("neo4j.user"), n.
15             GetString("neo4j.password"), ""))
16     if err != nil {
17         log.Fatal(err)
18     }
19     ses := dr.NewSession(neo4j.SessionConfig{AccessMode
20         : neo4j.AccessModeRead})
21     return ses
22 }
```

このファイルで Neo4j と接続し、セッションを返すようにします。

5.5.3 models ディレクトリ

coordinate.go

```
1 package models
2
3 import (
4     "fmt"
5     "log"
6
7     "neo4japi/server/db"
8
9     "github.com/neo4j/neo4j-go-driver/v4/neo4j"
10 )
11
12 type Route struct {
13     Position string `json:"point"`
14 }
15
16 func FindRoute(fr, to string) []*Route {
17     var r []*Route
18     ses := db.GetDriverAndSession()
19     defer ses.Close()
20     cyp := fmt.Sprintf(`
21         MATCH (from:Position {point_name: "%s"}) , (to:
22             Position {point_name: "%s"})
23             path=allShortestPaths ((from)-[distance:
24             Distance*]->(to))
25             WITH
26                 [position in nodes(path) | position.
27                  point_name] as name,
28             REDUCE(totalMinutes = 0, d in distance |
29                   totalMinutes + d.cost) as 所要時
```

```

間

26      RETURN name
27      ORDER BY 所要時間
28      LIMIT 10;
29      ', fr, to)
30
31  _, err := ses.ReadTransaction(func(transaction
neo4j.Transaction) (interface{}, error) {
32      result, err := transaction.Run(cyp, nil)
33      if err != nil {
34          return nil, err
35      }
36      if result.Next() {
37          name, _ := result.Record().Get("name")
38          nameAr := name.([]interface{})
39
40          for i := 0; i < len(nameAr); i++ {
41              r = append(r, &Route{nameAr[i].(string)})
42          }
43      }
44      return nil, result.Err()
45  })
46  if err != nil {
47      log.Fatal(err)
48  }
49  return r
50 }

```

出発地点と目的地を受け取ることで Neo4j に Cypher というクエリ言語を用いて最短経路を導出してもらいます。

5.5.4 controllers ディレクトリ

coordinate_controller.go

```
1 package controllers
2
3 import (
4     "net/http"
5
6     "github.com/gin-gonic/gin"
7     "neo4japi/server/models"
8 )
9
10 func RouteSearch(c *gin.Context) {
11     fr := c.Query("fr")
12     to := c.Query("to")
13     res := models.FindRoute(fr, to)
14     c.JSON(http.StatusOK, res)
15 }
```

GET リクエストで受け取ったパラメータを models で作成した関数に渡します。

5.5.5 router ディレクトリ

```
router.go
```

```
1 package router
2
3 import (
4     "github.com/gin-gonic/gin"
5     "neo4japi/server/controllers"
6 )
7
8 func Init() {
9     r := gin.Default()
10    r.GET("/coordinate", controllers.RouteSearch)
11    r.Run()
12 }
```

ルーティング先を定義します。

5.5.6 実行ファイル

```
coordinate_controller.go
```

```
1 package main
2
3 import (
4     "neo4japi/server/router"
5 )
6
7 func main() {
8     router.Init()
9 }
```

5.6 実行方法

```
docker compose up
```

このコマンドを実行することで Neo4j サーバーと Gin サーバーが立ち上がります。

5.6.1 実行確認

sample.http

```
1 GET http://localhost:8080/coordinate?fr=PointB&to=PointE
```

今回は VSCode の拡張機能である REST Client を用いて実行確認を行います。

ここで、 coordinate?fr=PointB&to=PointE の部分を自分の好きな地点にしてリクエストを送ると、最短経路が返されます。

5.7 おわりに

このチャプターではグラフ DB と Go 言語を用いた API サーバーの建て方を説明しました。Gin、Neo4j は奥が深いので、もし気になった方はぜひ自分で調べて触ってみてください。

6

Next.js 13 触ってみた

6.1 はじめに

10月後半に行われた Next.js Conf 2022 で発表された Next.js 13 を実際に触ってみたのでその内容を書きます。当初は全体を網羅して書く予定だったのですが量が多くかったの少し絞ってます。対象読者としてある程度 React や Next.js を触っている人を対象としています。この記事は半年程前に書いたため現在と異なっている場合があります。執筆時の Next のバージョンは 13.0.5 です。

6.2 プロジェクト作成からサーバ起動

TypeScript と ESLint を入れるか聞かれるので入れる。

```
1 $ npx create-next-app@latest --ts
```

pages ディレクトリを削除。

```
1 $ rm -rf pages
```

app ディレクトリを作る。

```
1 $ mkdir app
```

app ディレクトリは実験段階の機能なので、next.config.js を変更する。

next.config.js

```
1 const nextConfig = {
2   reactStrictMode: true,
3   swcMinify: true,
4   experimental: {
5     appDir: true,
6   },
7 };
```

app/page.tsx を作り、以下のようにする。

app/page.tsx

```
1 export default function Page() {
2   return <h1>Hello, Next.js!</h1>;
3 }
```

ローカルサーバ起動。

```
1 $ npm run dev
```

ブラウザで <http://localhost:3000/> にアクセスすると、Hello, Next.js! が表示される。



▲ 図 6.1 Hello,Next.js

6.3 Layout と Head

サーバを起動すると自動的に app ディレクトリに layout.tsx、head.tsx というファイルが作られていました。

next/head を使って各ページファイルに head を定義していたのが head.tsx に書けるようになったみたいですね。

```
head.tsx
```

```
1  export default function Head() {
2      return (
3          <>
4              <title></title>
5              <meta content="width=device-width,initial-scale=1" name="viewport" />
6              <link rel="icon" href="/favicon.ico" />
7          </>
8      )
9  }
```

ページの共通レイアウトを定義するファイル。

layout.tsx

```
1  export default function RootLayout({  
2      children,  
3  }: {  
4      children: React.ReactNode  
5  }) {  
6      return (  
7          <html>  
8              <head />  
9              <body>{children}</body>  
10         </html>  
11     )  
12 }
```

今まででは_app.tsx などに下記のようにレイアウトを定義していました。

_app.tsx

```
1  <Layout>  
2      <Component {...pageProps} />  
3  </Layout>
```

この方法だとページごとにレイアウトを変えられないです。なのでページごとにレイアウトを変えたい場合は getLayout を用いる必要がありました。Next.js 13 ではレイアウトを変えたいページがあるディレクトリに layout.tsx を置けばページごとにレイアウトを変えられるようになりました。

ここではダッシュボードページのレイアウトの場合を考えます。app ディレクトリの中で dashboard ディレクトリを作成して以下のように layout.tsx を配置するだけです。

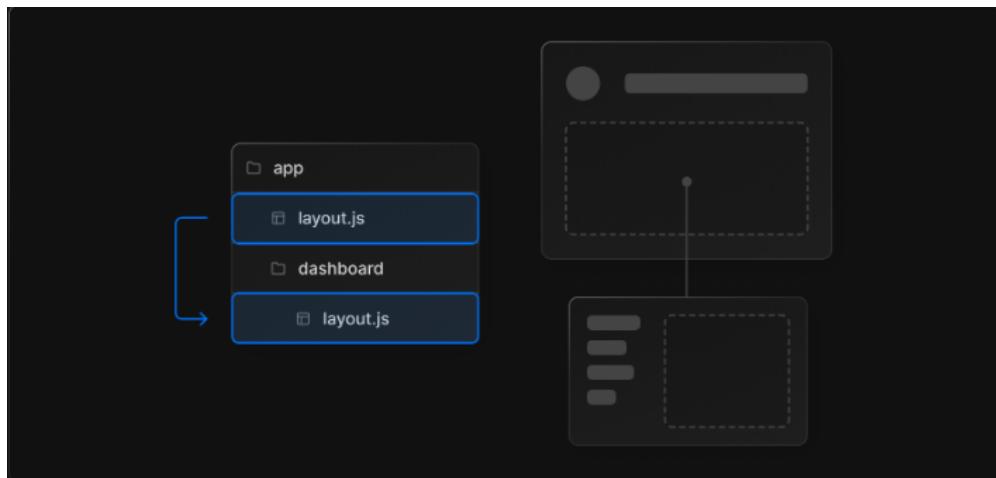
app/dashboard/layout.tsx

```
1  export default function DashboardLayout({  
2      children,
```

```
3     }: {
4   children: React.ReactNode;
5 }) {
6   return <section>{children}</section>;
7 }
```

少し疑問に思ったのが dashboard ディレクトリの page.tsx では RootLayout は呼ばれず DashboardLayout のみが呼ばれるのかと思っていました。しかし、実際には入れ子構造で呼び出されるようです。

公式の画像がわかりやすいので貼っておきます。



▲ 図 6.2 layout.js の適用範囲

次の機能に行く前に dashboard ディレクトリに page.tsx も作っておきます。

app/dashboard/layout.tsx

```
1 export default function DashBoardPage() {
2   return <h1>DashBoard Page</h1>;
3 }
```

違いがわかりやすいように他のファイルも変更します。

app/layout.tsx

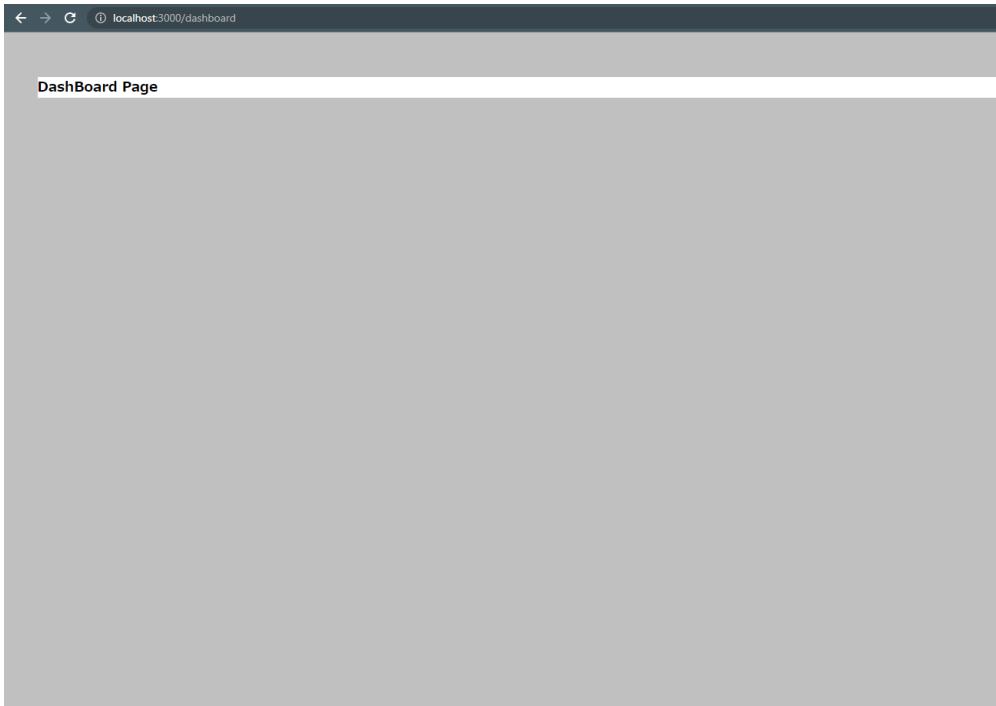
```
1  export default function RootLayout({
2      children,
3  }: {
4      children: React.ReactNode;
5  }) {
6      return (
7          <html>
8              <head />
9              <body
10                 style={{
11                     backgroundColor: '#COCOCO',
12                     padding: '50px',
13                 }}
14             >
15                 {children}
16             </body>
17         </html>
18     );
19 }
```

app/dashboard/layout.tsx

```
1  export default function DashboardLayout({
2      children,
3  }: {
4      children: React.ReactNode;
5  }) {
6      return (
7          <section
8              style={{
```

```
9         backgroundColor: 'white',
10        }
11      >
12      {children}
13    </section>
14  );
15 }
```

ページの見た目が画像のようになってれば OK。



▲ 図 6.3 DashBoard ページ

6.4 React Server Components

React Server Components(RSC) は React18 で追加された機能でクライアントとサーバ側が協調してアプリケーションをレンダリングできる機能です。これによりコンポーネントごとに最適なレンダリング方法を選択できるようになります。例えばデータの取得はサーバ側で行い、ユーザの操作によって変わる部分はクライアント側でレンダリングできます。これも公式ドキュメントの図がわかりやすいので貼っておきます。

What do you need to do?	Server Component	Client Component
Fetch data. Learn more.	✓	⚠
Access backend resources (directly)	✓	✗
Keep sensitive information on the server (access tokens, API keys, etc)	✓	✗
Keep large dependencies on the server / Reduce client-side JavaScript	✓	✗
Add interactivity and event listeners (<code>onClick()</code> , <code>onChange()</code> , etc)	✗	✓
Use State and Lifecycle Effects (<code>useState()</code> , <code>useReducer()</code> , <code>useEffect()</code> , etc)	✗	✓
Use browser-only APIs	✗	✓
Use custom hooks that depend on state, effects, or browser-only APIs	✗	✓
Use <code>React Class components</code>	✗	✓

▲ 図 6.4 Sever Component と Client Components の違い

また SSR との違いとしてクライアント側の JavaScript の量を減らせる点です。SSR の場合ハイドレーション（サーバ側で生成した DOM とクライアントで生成した DOM を合成する）というステップがありページを早く表示できてもクライアント側でも同じ処理が走るため JavaScript の量は同じでした。RSC はサーバ側でレンダリングした後、残りをクライアント側でレンダリングします。これによってクライアント側に送信される JavaScript の量を減らせます。

6.5 サーバーコンポーネントでデータを取得

app ディレクトリ内のコンポーネントはデフォルトだとサーバコンポーネントになっています。以下のコードはサーバサイドで Qiita の記事リストを取得して表示するものです。dashboard/page.tsx を書き換えます。

dashboard/page.tsx

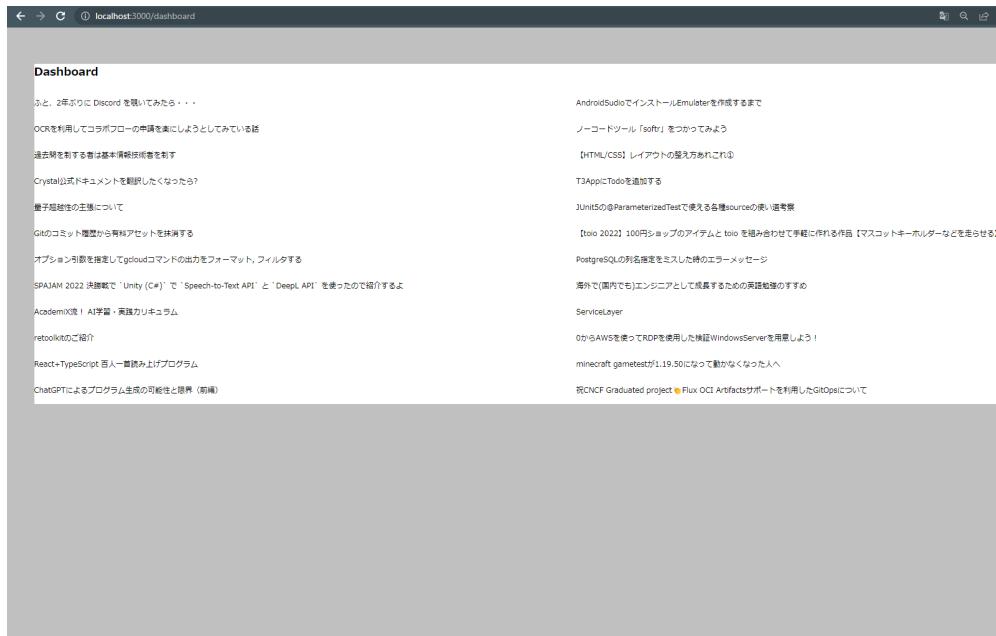
```
1 type Article = {
2   id: number;
3   title: string;
4 };
5
6 async function getArticle(): Promise<Article[]> {
7   const res = await fetch('https://qiita.com/api/v2/items?page=1&
8   per_page=24');
9
10  if (!res.ok) {
11    throw new Error('Failed to fetch data');
12  }
13
14  return res.json();
15
16 export default async function DashBoardPage() {
17   const articles = await getArticle();
18
19   return (
20     <div>
21       <h1>Dashboard</h1>
22       <div
23         style={{
24           display: 'flex',
25           flexDirection: 'column',
26           flexWrap: 'wrap',
27           height: '50vh',
28         }}
29       >
30         {articles?.map((article) => (
```

```

31         <div
32             key={article.id}
33             style={{
34                 display: 'flex',
35                 gap: '10px',
36             }}
37         >
38             <p>{article.title}</p>
39         </div>
40     ))}
41     </div>
42   </div>
43   );
44 }

```

画像のように表示される。



▲ 図 6.5 API からデータ取得後のページ

サイトにカーソルを合わせて右クリックしてページのソースを表示をクリックしてみましょ

う。あらかじめデータが入った状態でサーバから送られてくるのでページソースに記事データが表示されます

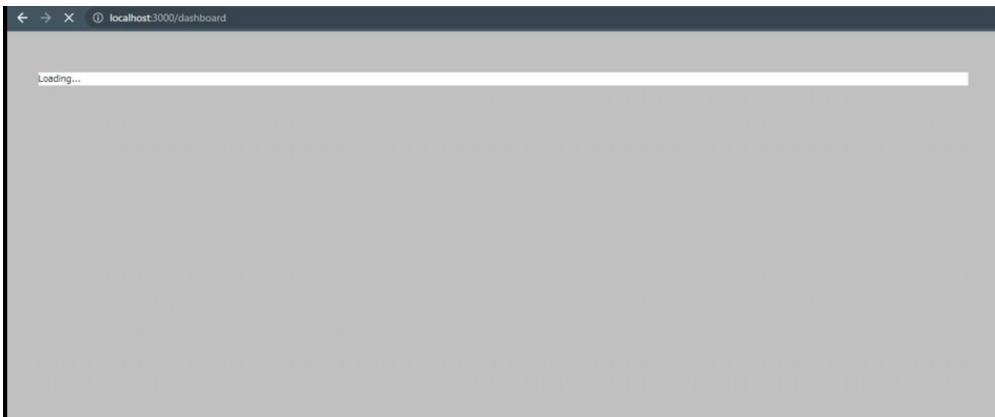
6.6 ローディング UI 表示

次はローディング UI を表示する機能です。dashborad ディレクトリに loading.tsx を作成します。

loading.tsx

```
1  export default function Loading() {  
2      return <p>Loading...</p>;  
3  }
```

データの取得が終わり、page コンポーネントがレンダリングされるまでの間は Loading コンポーネントが表示されます。



▲ 図 6.6 ローディング UI

これは React18 で追加された Suspense という機能が使われています。Suspense について説明するとコンポーネントが表示されるまでの状態を指定できるコンポーネントです。非同期的なコンポーネントの場合レンダリングに時間がかかるためその間に何を表示させるかを Suspense を使うと指定できるようになります。

具体的な使用例を上げます。下のコードはデータフェッチライブラリ React Query を使ったデータ取得と表示のサンプルです。

```
1 import { QueryClient, QueryClientProvider, useQuery } from 'react
-query';
2
3 const queryClient = new QueryClient();
4
5 export default function App() {
6   return (
7     <QueryClientProvider client={queryClient}>
8       <Example />
9     </QueryClientProvider>
10    );
11  }
12 export function Loading() {
13   return <p>Loading...</p>;
14 }
15
16 function Example() {
17   const { isLoading, data } = useQuery('repoData', () =>
18     fetch('https://api.github.com/repos/tannerlinsley/react-query')
19       .then(
20         (res) => res.json()
21       )
22     );
23   if (isLoading) return <Loading />;
24
25   return (
26     <div>
27       <h1>{data.name}</h1>
28       <p>{data.description}</p>
29     </div>
30   );
}
```

```
31 }
```

Suspense を使えば下のコードに置き換えられます。

```
1 import { Suspense } from 'react';
2 import { QueryClient, QueryClientProvider, useQuery } from 'react-
    query';
3
4 const queryClient = new QueryClient();
5
6 export default function App() {
7   return (
8     <QueryClientProvider client={queryClient}>
9       {/* コンポーネントでラップ suspense */}
10      <Suspense fallback={<Loading />}>
11        <Example />
12      </Suspense>
13    </QueryClientProvider>
14  );
15}
16 export function Loading() {
17   return <p>Loading...</p>;
18 }
19
20 function Example() {
21   const { data } = useQuery('repoData', () =>
22     fetch('https://api.github.com/repos/tannerlinsley/react-query')
23       .then(
24         (res) => res.json()
25       )
26     );
27 }
```

```
26 //ローディングプロパティによる表示分岐の削除
27
28 return (
29   <div>
30     <h1>{data.name}</h1>
31     <p>{data.description}</p>
32   </div>
33 );
34 }
```

変更点は Example コンポーネントを Suspense コンポーネントでラップしているのと、Example コンポーネントの中の isLoading プロパティを使った表示切替の部分が消えているところです。Suspense のいいところはデータ取得をするコンポーネントの中で表示を切り替える処理を書く必要がない所です。これによってより宣言的なコードになりました。またコンポーネントの責務の観点から見てもローディング完了時の表示だけでよくシンプルになっています。

Next.js 13 では loading.tsx を置いてあげると Next.js 側がそれを読み取り Page コンポーネントを Suspense コンポーネントでラップしてくれるようです。普通に書くと以下のようにになります。

```
1 <Layout>
2   <Headr/>
3   <SideNav/>
4   <Suspense fallback={<Loading />}>
5     <DashBoardPage />
6   </Suspense>
7 </Layout>
```

ディレクトリ内に loading.tsx を配置すると、自動的にこのコードを記述した動作が実現されます。

6.7 エラーハンドリング

次にエラーハンドリングを見ていきます。dashboard ディレクトリに error.tsx を作成します。

dashboard/error.tsx

```
1 'useClient';
2
3 import { useEffect } from 'react';
4
5 export default function Error({
6   error,
7   reset,
8 }: {
9   error: Error;
10  reset: () => void;
11}) {
12  useEffect(() => {
13    // Log the error to an error reporting service
14    console.error(error);
15  }, [error]);
16
17  return (
18    <div>
19      <p>Something went wrong!</p>
20      <button onClick={() => reset()}>Reset error boundary</button>
21    </div>
22  );
23}
```

公式ドキュメントによると error.tsx はクライアントコンポーネントである必要があり use client; と書けばクライアントコンポーネントして利用するできるようです。loading.tsx と同じようにファイルを置いておけば自動的に Page をネストしてエラーハンドリングをしているようです。

普通に書いた場合

dashboard/error.tsx

```
1 <Layout>
2   <Headr/>
3   <SideNav/>
4   <ErrorBoundary fallback={<Loading />}>
5     <DashBoardPage />
6   </ErrorBoundary>
7 </Layout>
```

ただこれを見てわかる通り page をラップしてるので同階層のコンポーネントのエラーハンドリングはできない感じですね。したいならより上の階層で ErrorBoundary を使う必要がありそうです。

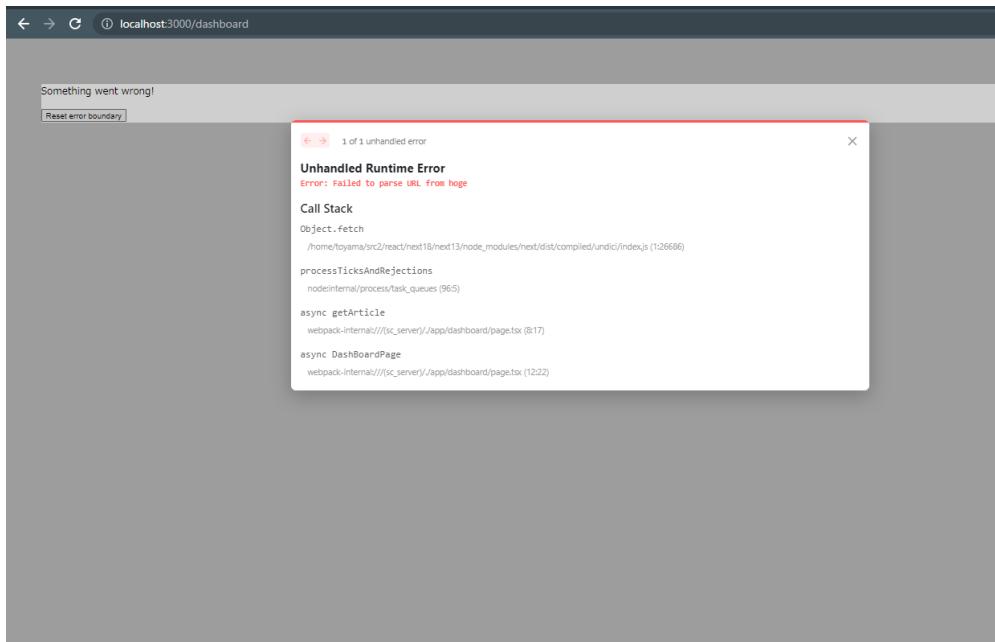
実際にコードを書き換えてエラーを出してみます。存在しない URL 'hoge' を指定し getArticle でしていたエラーハンドリングを削除しています。

dashboard/page.tsx

```
1
2 'useClient';
3
4 import { useEffect } from 'react';
5
6 export default function Error({
7   error,
8   reset,
9 }): {
10   error: Error;
11   reset: () => void;
12 }) {
13   useEffect(() => {
14     // Log the error to an error reporting service
15     console.error(error);
```

```
16     }, [error]);
17
18     return (
19       <div>
20         <p>Something went wrong!</p>
21         <button onClick={() => reset()}>Reset error boundary</button>
22       </div>
23     );
24 }
```

エラーの内容とエラーページが表示されていますね。



▲ 図 6.7 エラーハンドリング画面

6.8 終わりに

感想としては、Next.js 13 の機能は React18 の Suspense や RSC などの機能に合わせたアップデートというのを強く感じました。

7

シス研の日常

シス研は技術一辺倒ではなく、次のような活動をしているメンバーもいます！

- 松土

p.47 『戦闘機に乗ろう』

戦闘機のフライトシュミレータの紹介をしています！



▲ 図 7.1 たこ焼きで IT をそりゅーしょんしている様子

8

戦闘機に乗ろう

8.1 DCS World

8.1.1 はじめに

はじめまして、松土です。いきなりですが、僕はミリオタ^{*1}で、特に戦闘機が好きです。僕は戦闘機に乗りたい！！！。しかし、戦闘機というのは皆さんもご存知の通り、軍用の航空機のため主に軍隊が保有しており、日本では航空自衛隊のみが保有しています。そんな戦闘機に乘ろうと思ったら、航空自衛隊に入隊し、高い倍率の選抜を受けた後に何年もの厳しい訓練を乗り越えないといけません。そこで、一度は耳にしたことであろう、フライトシミュレーターというのがこの世には存在しており、これは一般人が仮想のコクピットに乗り込み、操縦をシミュレートできる素晴らしいものです。本物のパイロットが実機を操縦する前の訓練をするにあたって使用する事もあります。僕はこれをを利用して戦闘機に乗りたい欲を解消することにしました。

8.1.2 DCS World とは

いきなり出てきた DCS World^{*2}とは何か？、これは戦闘機に特化したフライトシミュレーターで、フランス空軍にも採用されている大変優れたものです。現実に存在する戦闘機がいく

*1 ミリタリーオタクの略 所謂軍事が好きな人

*2 Digital Combat Simulator World の略

つかモジュールとしてあり、一般人が家庭でパソコンの中にインストールするだけで、実機を仮想状で操縦できるようになります。プラットフォームは Windows で、2008 年からリリースされており、開発元は本社をロシアのモスクワに置いている Eagle Dynamics 社です。

>>>アプリは無料です<<< ※モジュールは有料です

8.2 動作環境

8.2.1 最小構成

- OS: 64-bit Windows 10 , DirectX11 (version 2.7 以降は Windows 7 非対応かつ Windows 8 が明記落ち、version 2.5.x までは Windows 7/8 も可)
- CPU: Intel Core i3 2.8 GHz / AMD FX
- RAM: 8 GB (重いミッションをプレイする場合: 16 GB)
- HDD 空き容量: 60 GB (※注: 2022 年夏時点の version2.7.16 では 120GB に増加している)
- ビデオカード: NVIDIA GeForce GTX 760 / AMD R9 280X 以上
- ユーザー認証用インターネット接続

8.2.2 推奨構成

- OS: 64-bit Windows 10 , DirectX11 (version 2.7 以降は Windows 8 が明記落ち、version 2.5.x までは Windows 8 も推奨内)
- CPU: Core i5 3GHz 以上 / AMD FX 又は Ryzen
- RAM: 16 GB (重いミッションをプレイする場合: 32 GB)
- HDD 空き容量: 130 GB (SSD 推奨 全モジュール導入には 460GB)
- ビデオカード: NVIDIA GeForce GTX 1070 / AMD Radeon RX VEGA 56 (8GB VRAM) 以上
- ジョイスティック
- ユーザー認証用インターネット接続

8.2.3 VR を使用する場合

推奨構成に上書きして

- CPU: Core i5 3GHz 以上 / AMD FX 又は Ryzen
- RAM: 16 GB (重いミッションをプレイする場合: 32 GB)
- HDD 空き容量: 130 GB (SSD 推奨 全モジュール導入には 460GB)

- ビデオカード: NVIDIA GeForce GTX 1080 / AMD Radeon RX VEGA 64 (8GB VRAM) 以上

8.3 導入

8.3.1 公式版と Steam 版

公式 DCS World のサイトよりダウンロードする場合とゲームプラットフォームで有名な Steam でダウンロードする場合があります。特に両者違いはありませんが、Steam 版は公式版よりも更新が遅いことがあります。

8.3.2 安定版と Open Beta 版

新しいモジュールはまず Open Beta 版のみに対して提供され、何度かの Open Beta アップデートを経て十分にバグを無くしてから安定版への提供となります。

Open Beta の利点

マルチプレイヤーサーバーの多くは Open Beta を使用している (2023/05/07 現在) ためマルチプレイをしたいのなら Open Beta 推奨新しいモジュールをより早く遊ぶことができるレーダーや FLIR など、前バージョンではモジュールへの実装がオミットされていた一部機能や兵装が追加実装されたのを早く体験できる

Open Beta の欠点

Open Beta は当然バグが多く、特定の武器を使うとゲームがクラッシュする現象が起きることもあるゲームがクラッシュまではしなくとも、以前のバージョンでは正常だったはずの特定のミサイルの誘導能力がおかしくなっている、レーダーや照準などの装置の操作や挙動がおかしくなっている、といったバグが新しく増えていることもある。

8.3.3 モジュールの購入

アプリ自体は無料のため、インストール後起動し、プレイすることが可能ですが、初期で乗ることのできる機体は、第二次世界大戦で使用されたプロペラ機の練習機版 TF-51 とソ連^{*3}が開発した攻撃機 Su-25 だけです。どちらも、戦闘機ではない上にカッコ悪い^{*4}です。

注意として、機体内部のボタンやスイッチを一つ一つまで操作できる機体（クリッカブル機）と、キーボードを使い、キーで細かい機体の操作を行う機体（FC3 機）があり、クリッカブル機は基本的に高価ですが、実機と同じく全てのボタンが操作でき、リアルな操作を楽しむ

^{*3} ソビエト連邦の略

^{*4} あくまで個人の感想です

ことができます。

8.4 まとめ

今回は、ミリオタ向けライトシミュレーターとして、DCS World を紹介させていただきました。導入した後の楽しみ方は、実機と同じエンジンスタートを行ってみたり、マルチプレイで友人と飛んでみたり、または戦ってみたり、と様々であり、あなたの思うがままにやりたいことができるるのがこの DCS World の良いところです。もしも、これを読んでいるあなたが DCS World を始めたら、僕と一緒に飛びましょう。

9

あとがき 枢

9.1 本誌創刊に寄せて

物は試し、ということわざがあります。また、言うは易く行なうは難し、ということわざもあります。どちらのことわざにしても、実際にやってみれば良く分かる、という共通の帰結があります。しかし、あまりに簡単に実行できる仕組みを作ってしまうと、しばしば人間は仕組み無しで試すことや仕組み自体を軽視し始め、その仕組みの存在のありがたさを忘れてしまうものなのでしょう。そして、失って初めてその存在のありがたみに気が付く、という言葉は、前述のような体験を経た（やってみた）人からしばしば生まれてくる言葉なのでしょう。では、あまりに簡単に成功する仕組みがある状況下で、次代にその仕組みの意味や価値を伝えるにはどうすればよいのでしょうか。

次代の人々に対して、比較的短期間かつ効果が期待できる方法のひとつに、意図的な損失を生み出す方法があります。失って気が付くのであれば一度失ってみさせれば良い、という考えです。しかし、一度得たものを失うことによつて少なからず不満を抱いてしまうのは、人間の性です。故に、実行者にはその不満の矛先が向けられる可能性があります。また、損失は発展を阻害する要因でもあります。従って、進歩のための損失であるように、損失の度合いには十分に配慮しなければなりません。

一方で、正義は常に正しいとは限らない、という考えがあります。それは、時代や地域や環境によって、文化や思想や理念など、是非を判断する上での前提条件が異なるからです。ある正義の下では正当だとされる仕組みでも、別の正義の下では不当だとされ得るということで

す。しかしながら、正義は人間の行為における動機付けのひとつとされています。昨日までの正義を否定するような今日の正義に直面した時、我々はどのように受け止め考えていけば良いのでしょうか。選択肢として、一切の拒絶をするか、昨日までの自らの行いを否定することによって自己正当化をするか、などが考えられます。それらの中でも、多少なりとも理解を試み受け入れようとする選択が、多角的な視点からより良い考えが期待できるでしょう。

私はシス研に長らく在籍していました。本誌のような試みが10年振りに復活し、製本されることを大変嬉しくありがとうございます。シス研では、ある時は観察者として、またある時は友や助言者として、そして敵として、振舞いました。かつて理想と信念を掲げ、仲間と共にある時代を築いた者のひとりとして、自主的に行ったとはいえ、次代の芽が出るまでの橋渡し役はなかなかに堪えるものでした。自らが信じた正義と次代を担う彼らの正義を常に見比べ、その上で役割を果たす必要があったからです。何かを手に入れようとする若さを生かすためには、何かを失うまいとする老いた私情は捨て去らねばなりません。しかしながら、過去の歴史を知り同じ過ちを繰り返さないようにすることは、未来を思い描くために重要なことであるはずです。ある意味ではそれを正義として、客観的な答えを導くために私は自分自身を自制していたのかもしれません。少なくとも、未来を思い描く先導者にとって、彼ら自身が追従するような相手はいませんから。

これから、シス研は新たな時代を迎えようとしています。最後に、マハトマ・ガンジーが残した2つの名言で締めつつ、シス研の今後の活動にご期待ください。

物事は初めはきまつて少数の人によって、ときにはただ一人で始められるものである。
満足は努力の中にあって、結果にあるものではない。

9.1 奥付け

今回はこの本を手に取っていただきありがとうございました。企画・編集を行いました、牧野です。さて、10年ほど前のシス研では、本という形で技術をアウトプットするという文化があったと先輩から教えてもらいました。今では、Qiita や Zenn のような技術ブログが発達して、物理的なもので残す文化がちょっとずつ廃れてしまったりらしいです。そんな中、私が C101 の冬コミで技術本というものに出会いました、一つ一つ個人で調べたり研究した情報を、本という形で残すことにしていいと思い、今回はこのような本を執筆しました。本は知識としてずっと蓄えられるものですので、私たちが書いた本が誰かの役にたてればと思い、締めさせてもらいます。次回も余裕があれば出したいな。

企画・編集 牧野遙斗

Sysken の技術本 様々な技術を詰め合わせてみました。

発行日	2023 年 5 月 28 日 (日) 技書博 8 & OSC2023 (初版)
サークル	愛知工業大学 システム工学研究会
Instagram ID	@ait.sysken
Twitter ID	@set_official
QiitaOrganization URL	https://qiita.com/organizations/sysken
代表	牧野遙斗
代表者メールアドレス	harutiro2027@icloud.com
企画・編集	牧野遙斗 (Twitter: @minesu1224)
著者	林航平 suda michiyo hihumikan Beyond Toyama 水谷祐生 (Twitter: @yurayura_0303) BlacKnight 松土 (Twitter: @kk22blacknight) shirataki1126
印刷所	しまや出版

※本書の無断複写、複製、データ配信はかたくお断りいたします。