iOS授業7日目資料

課題発表

発表が終わったら拍手しましょう👏

はじめに

はじめに：アンケート結果

最後の授業でやって欲しいことのリクエスト

7 件の回答

TikTokみたいな全画面表示の動画アプリの、動画表示やいいねボタンを作ってみたい(最近のUIの流行りだと思うので)

授業後の独学方法。まだ知らないswiftの世界について。道筋は見えるけどとても難しいもの(抽象的ですいません。)

APIにしたデータベースと接続する方法など・・・

firebaseのDB設計をお願いします！！

これまでの復習とどのファイルがどう言う処理をしているかの解説

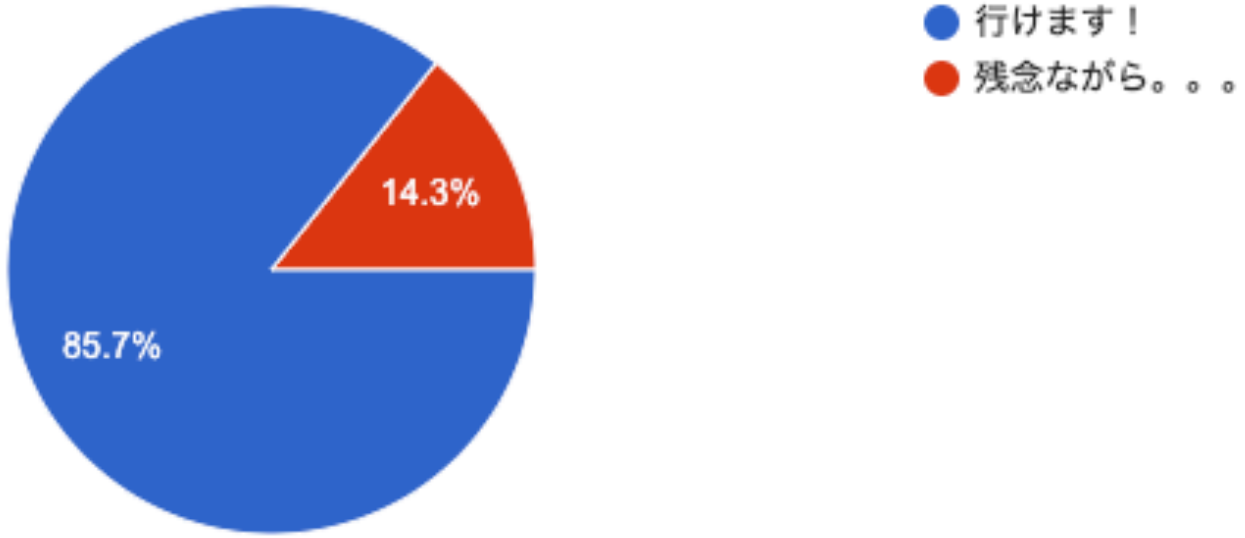
firestoreのリレーション、画像データの扱い

テーブルの一部のデータのフィルターのかけ方がしりたいです。
例えば、講義で作成したポケモンテーブルの中に、写真と名前と、その生息場所を持たせて、
その生息場所で、フィルターをかけて、表示する項目に制限をかける方法を知りたいです

はじめに：アンケート結果

最終日(2/22)に飲み会（打ち上げ）をやりたいのですが来れる方！

7 件の回答



はじめに：今日以降の授業

日付	タイトル	内容
2/8	Cloud Firestore	・Firestoreでデータを保存・取得する
2/15	Firebase Storage	・Firebase Storageに画像を保存する
2/21	よしなになんかやる	なんかやるよ。

はじめに：今日やること

◆1.Firestoreについて

◆2.Firestoreセットアップ

◆3.Firestoreで保存・読み込みをする

◆4.(時間があれば)その他Firestore主要機能

Firestoreについて

Firestoreについて

・Firebaseの主要サービスの１つ、NoSQLデータベースサービス

・正式版リリース以降はRealtime Databaseよりこちらがメイン
（実際、機能もどんどん増えている）

・バックエンドのインターフェースを開発しなくても、
クライアントアプリにSDKを組み込むだけで、簡単にDBとのやりとりができてしまう

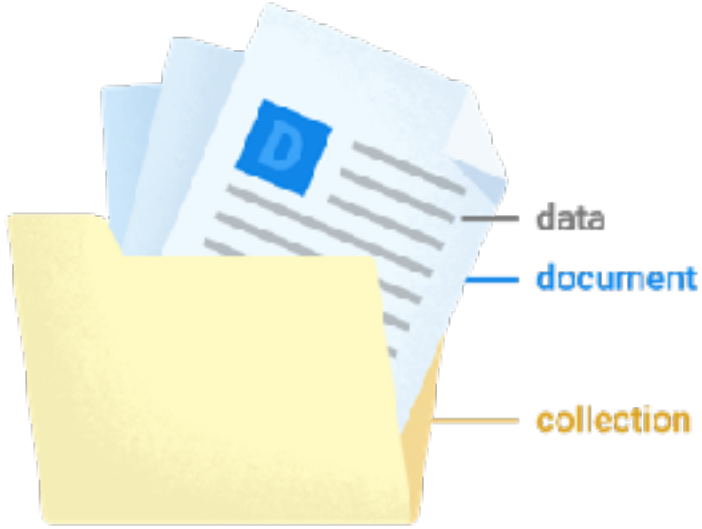
・クエリはRealtime Databaseよりマシではあるが高度なものは苦手

・Firestoreではユーザーが増えて負荷があがっていっても勝手にスケーリングしてくれる

・24時間365日Googleさんが監視してくれてるので安心

・認証含めサービスの根幹部分をFirebaseに全て預けることにはなるので移行は大変

Firestoreのデータについて



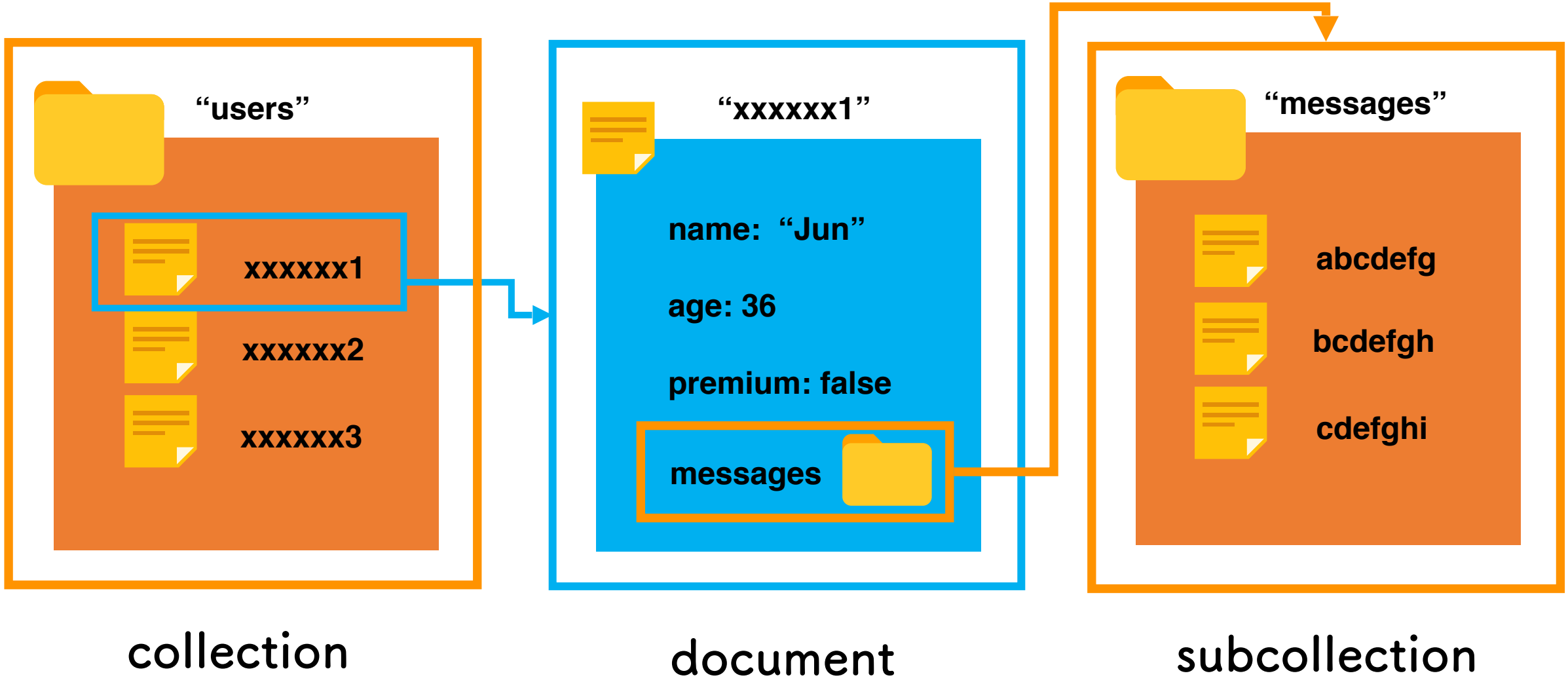
・表でデータを扱うSQLデータベースと違って、
　データ自体を「キー」と「値」のペアで扱う

・「キー」と「値」のデータの束は、
　「ドキュメント」というノート的なものに格納される

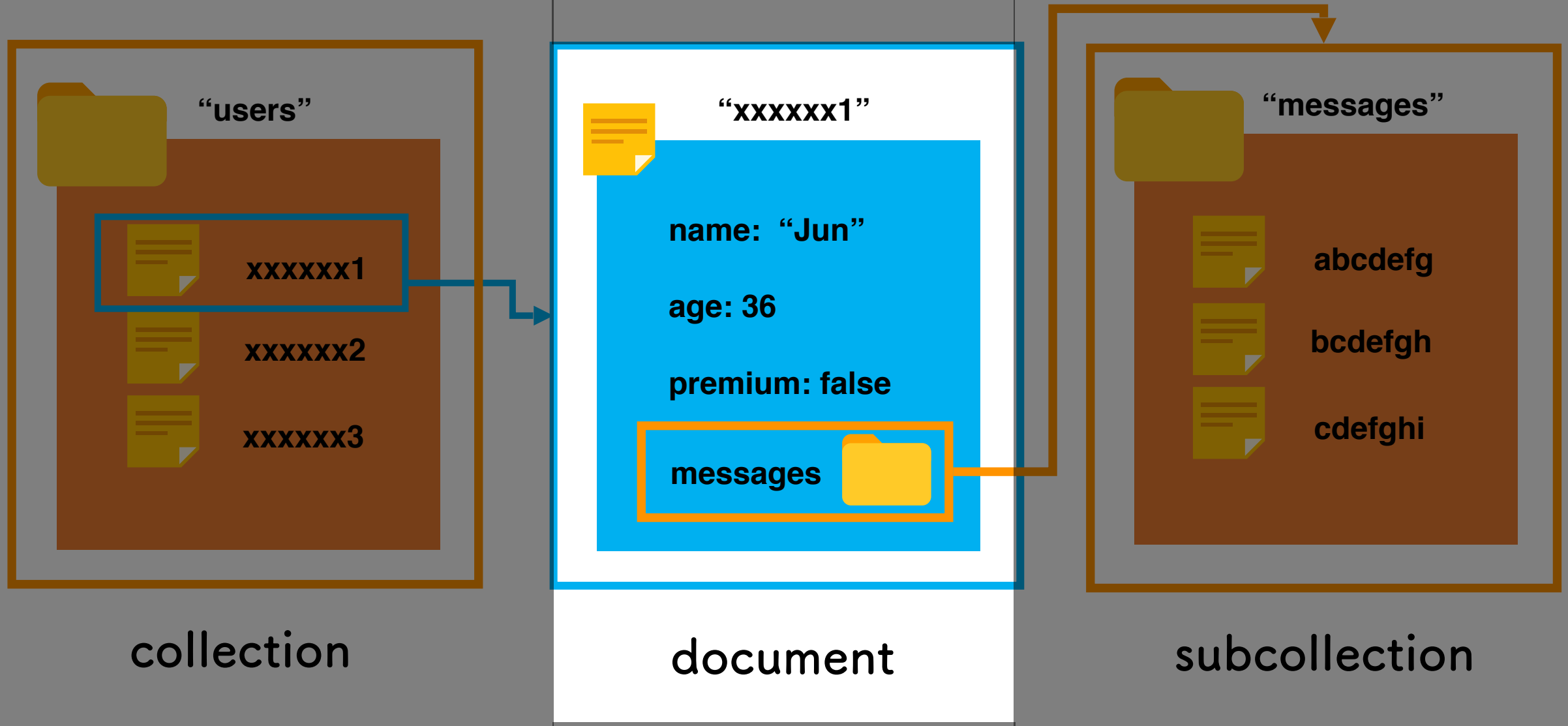
・「ドキュメント」には、数値・文字列・参照・コレクション
　といった様々な値が保存できる

・「ドキュメント」 は「コレクション」というフォルダ的なものに格納される
　「ドキュメント」に入った「コレクション」は「サブコレクション」と呼ぶ
　＝「コレクション」はただの「ドキュメント」の入れ物

Firestoreのデータについて

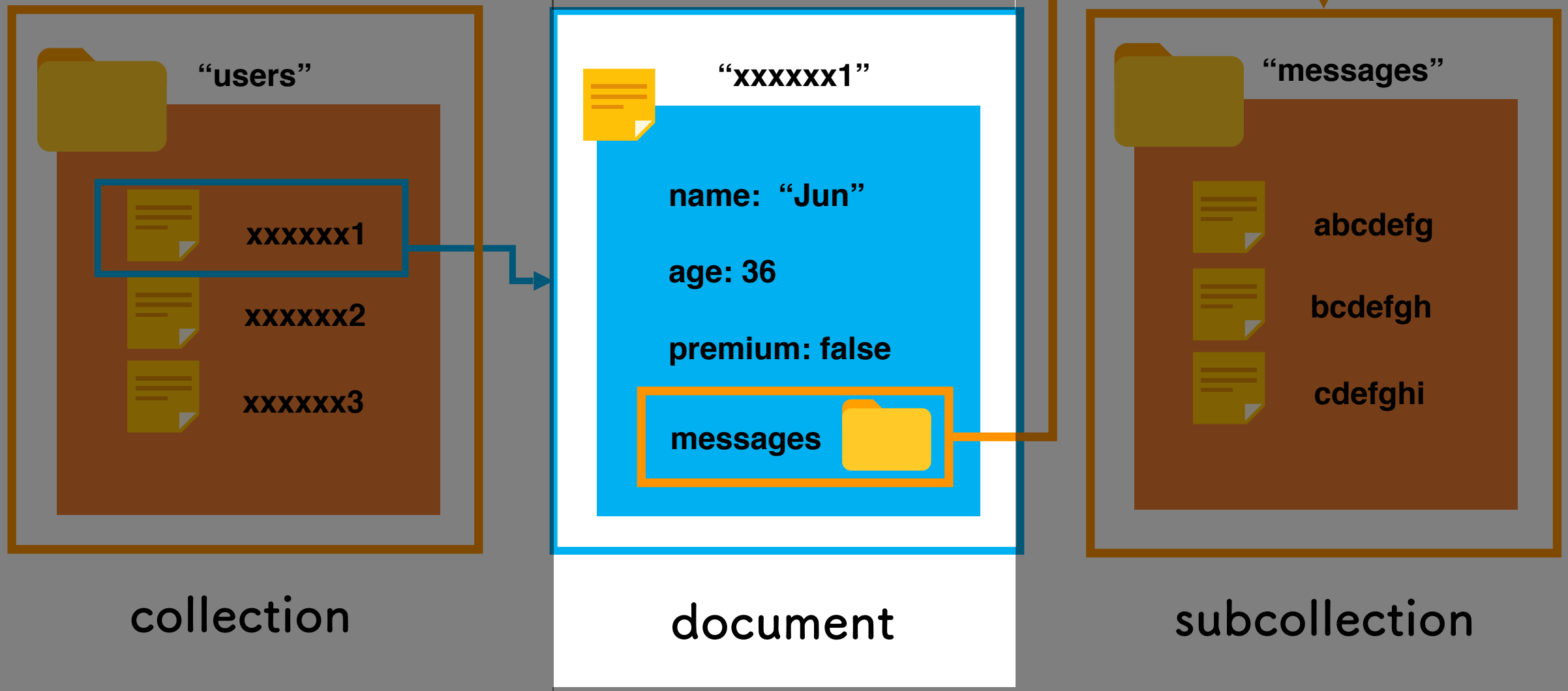


Firestoreのデータを「キー」と「値」のペアで扱う



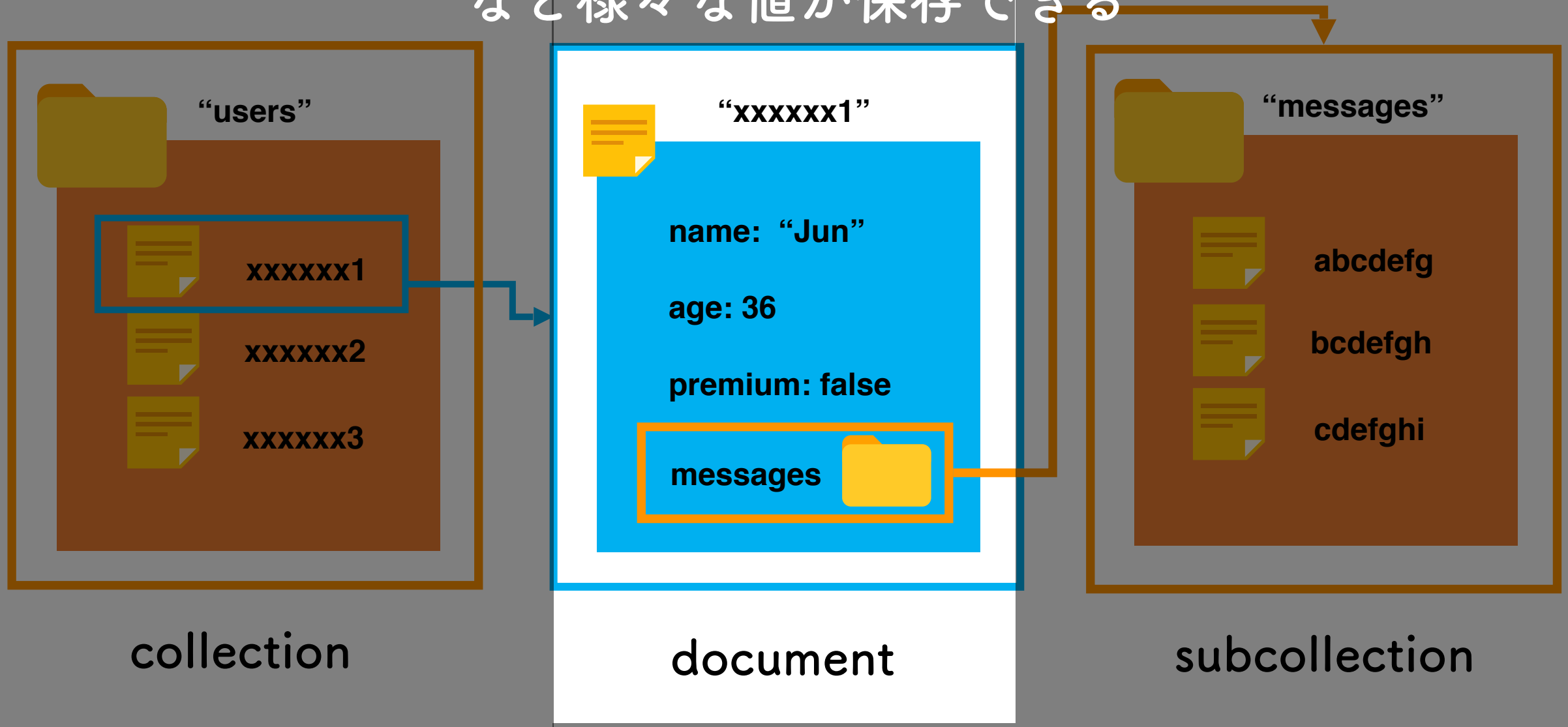
Firestoreのデータについて

「キー」と「値」のデータの束は、
「ドキュメント」というノート的なものに格納される



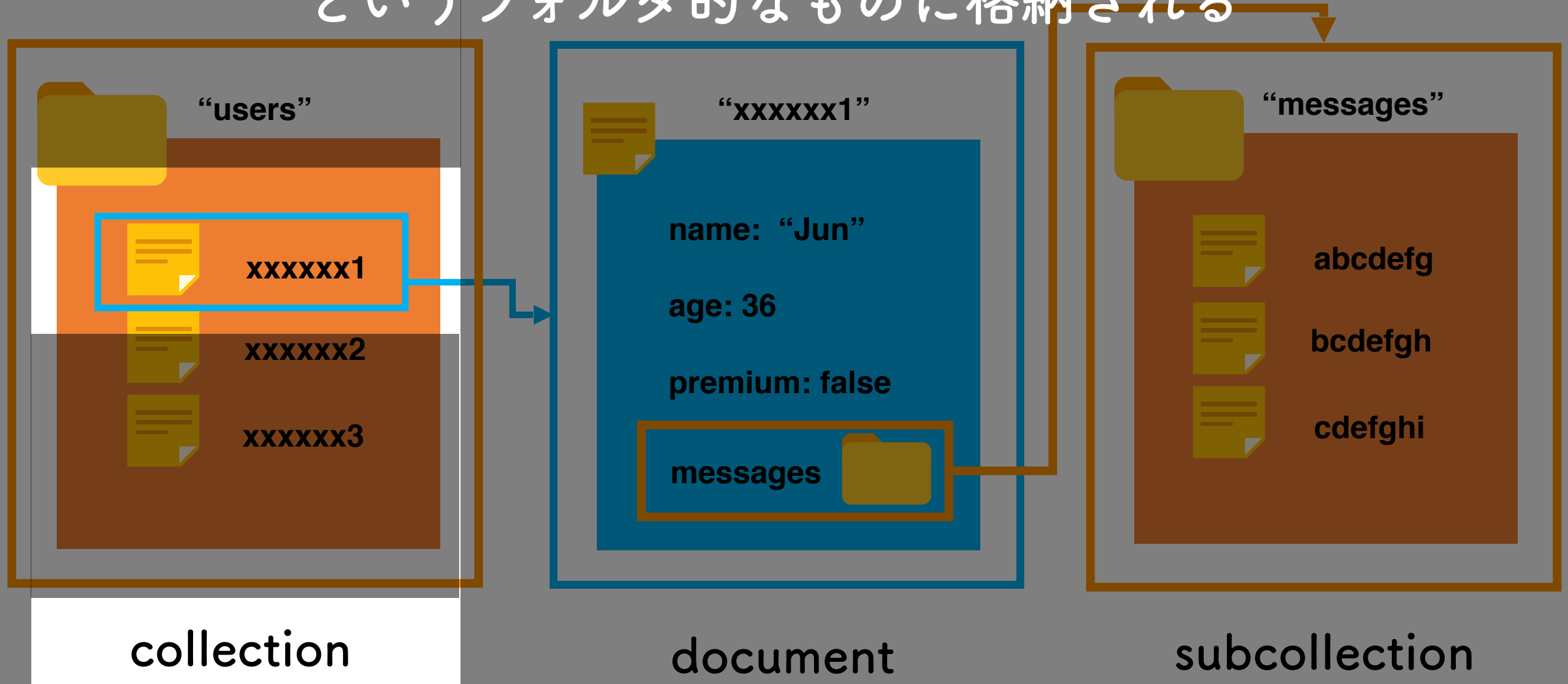
Firestoreのデータについて

「ドキュメント」には数値・文字列・参照・コレクション
など様々な値が保存できる



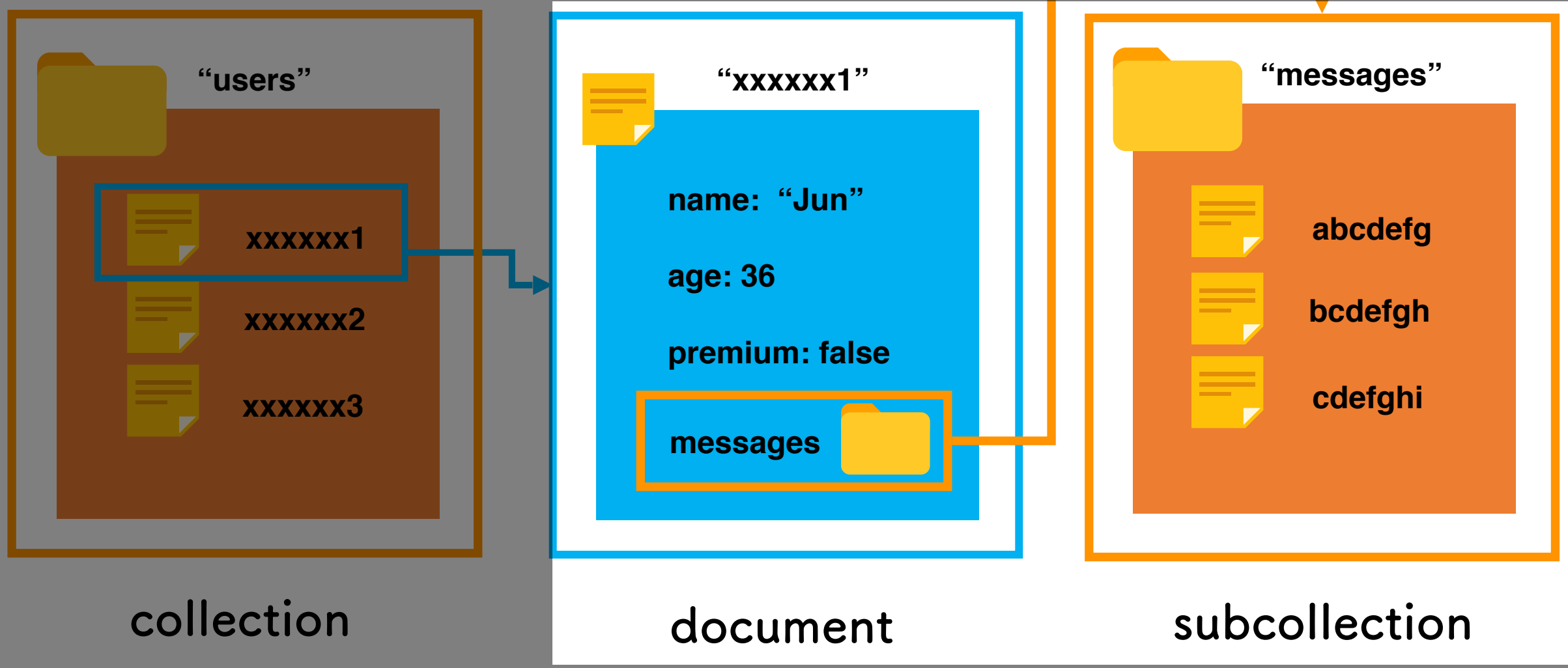
Firestoreのデータについて

「ドキュメント」は「コレクション」というフォルダ的なものに格納される



Firestoreのデータについて

「ドキュメント」に入った「コレクション」は
「サブコレクション」と呼ぶ



補足：サポートしているデータ型

Realtime Database(String/Int&Double/Dictionary/Array)よりも
多くのデータ型をサポートしている

https://firebase.google.com/docs/firestore/manage-data/data-types?hl=ja

[Cloud Firestore データモデル | Firebase](https://firebase.google.com/docs/firestore/data-model?hl=ja)

ここまでのおさらい

こんなことを学びました

①Firestoreについて

②Firestoreのデータについて

Firestoreの導入

【一緒にやってみよう】

Firestoreのセットアップをしよう

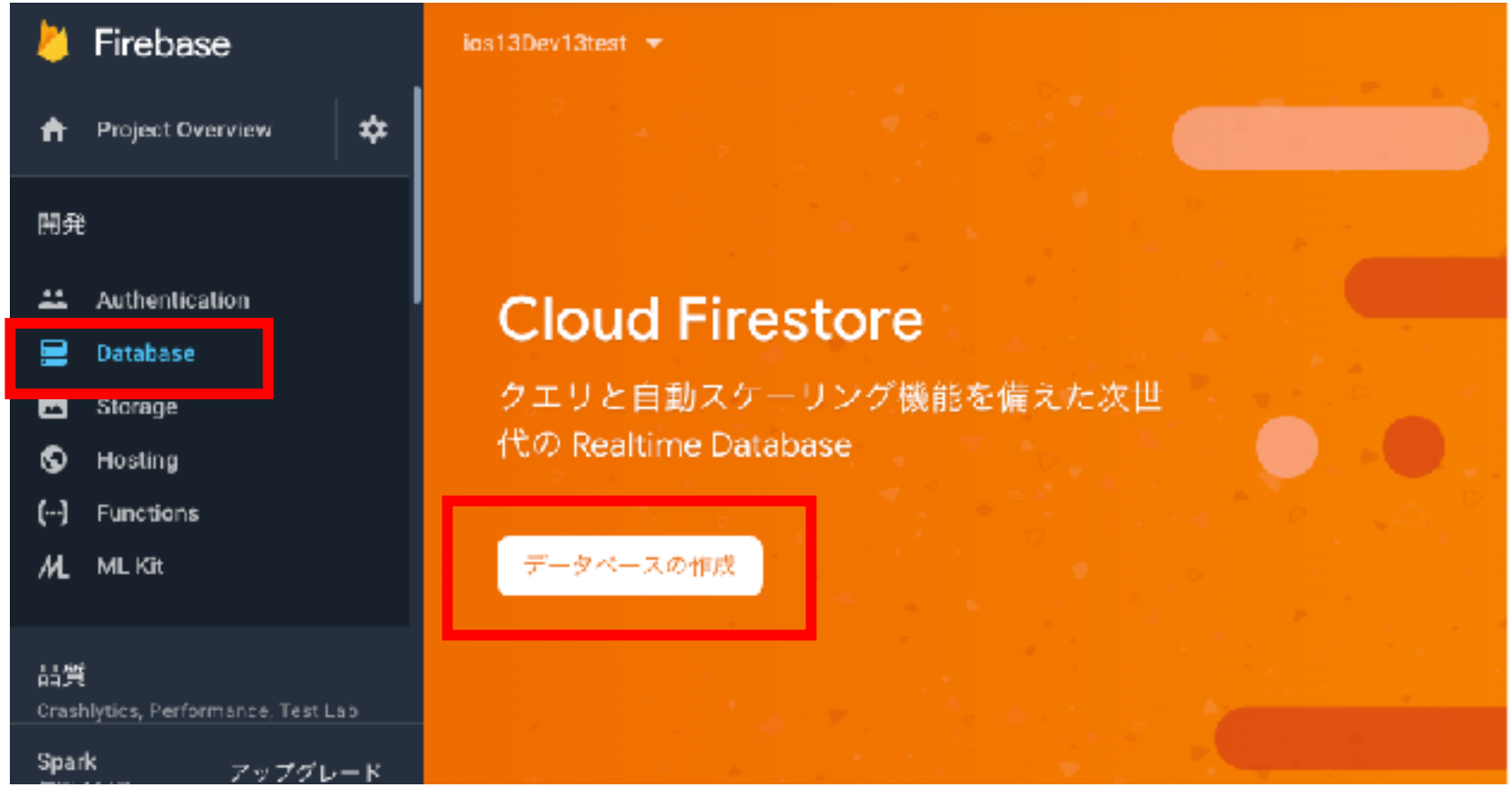
準備

本日配布している
「GsTodo」のプロジェクトを
手元に準備してください

GoogleService-Info.plistは
先週使用した自分のものに置き換えてください

Firestoreセットアップ

コンソールでDatabaseを選択してデータベースを作成



Firestoreセットアップ

テストモードで開始（セキュリティルールはあとでやります）

Firestoreセットアップ

いつものpod install

PodFileに追加して…

```
pod ‘Firebase/Firestore'
pod ‘FirebaseFirestoreSwift'
```

Podコマンド

```
$ pod install
```

ここまでのおさらい

こんなことを学びました

①Firestoreのセットアップ方法
※セキュリティルールは除く

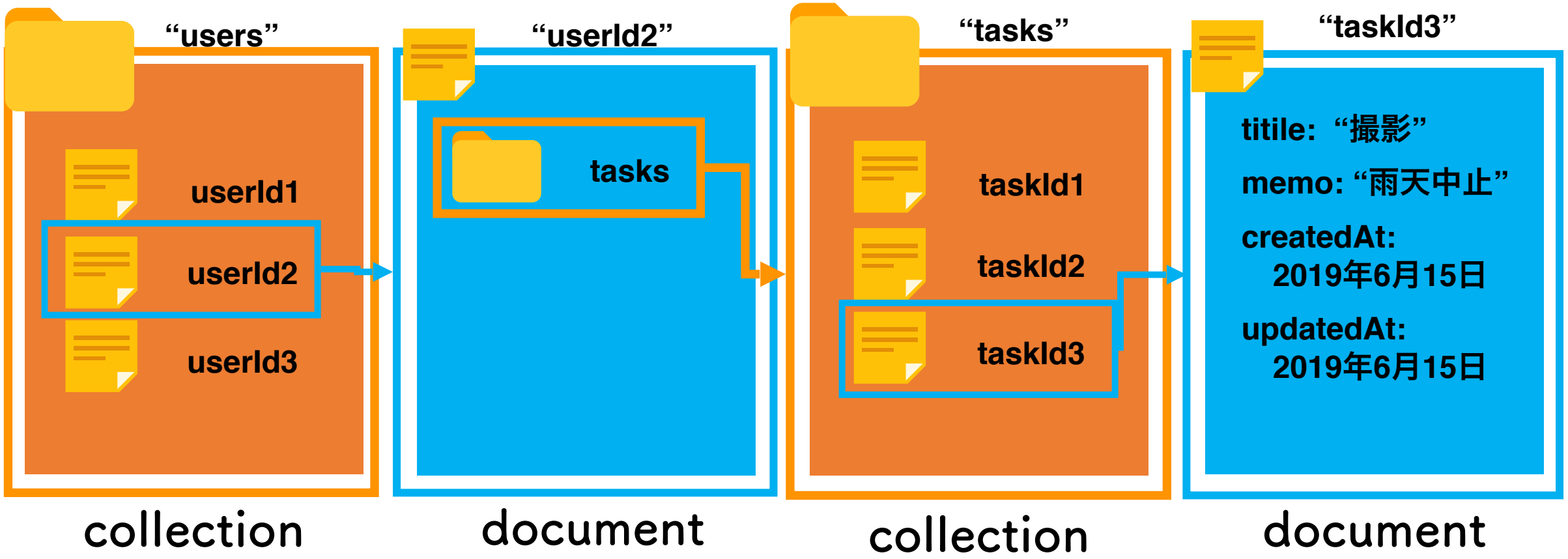
FirestoreでCRUDする

【一緒にやってみよう】

TODOのデータを
Firestoreに「追加」してみよう

※完成PJは配布するので
ついてこれなくなったら
手を止めて見ることに集中！

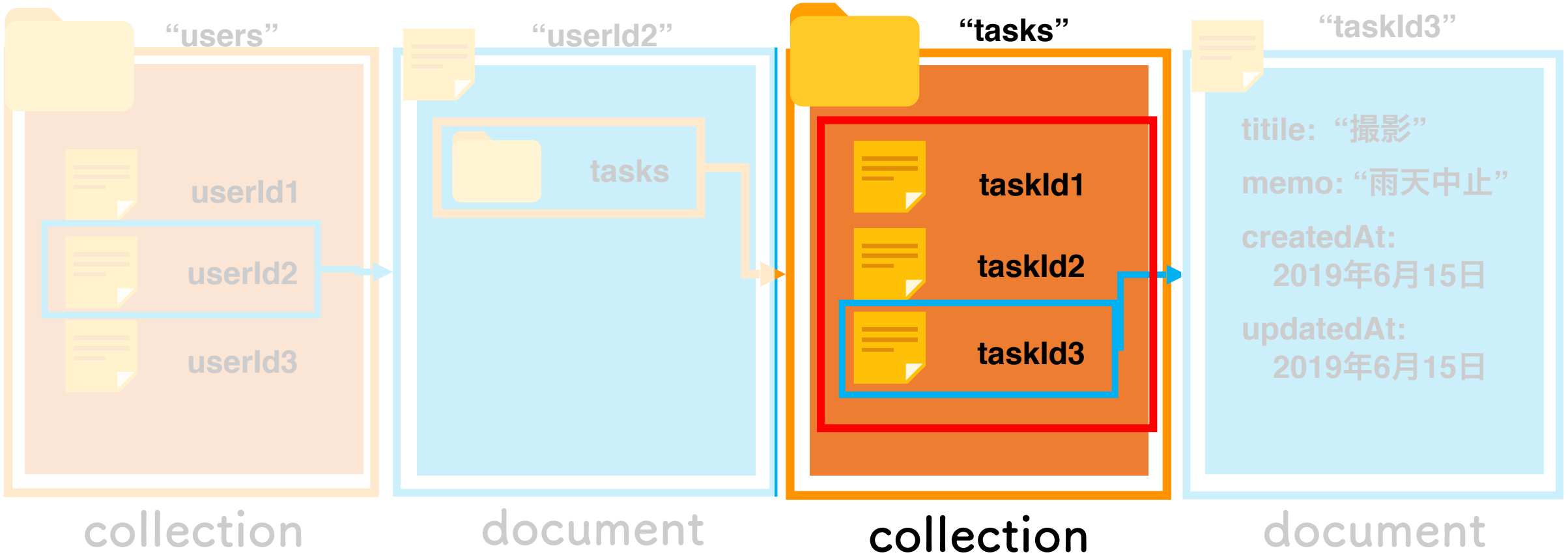
着手する前に今回作成予定のデータ構造をチェック



今のデータだと対応できない部分があります

どこを対応しないといけないか洗い出してみましょう

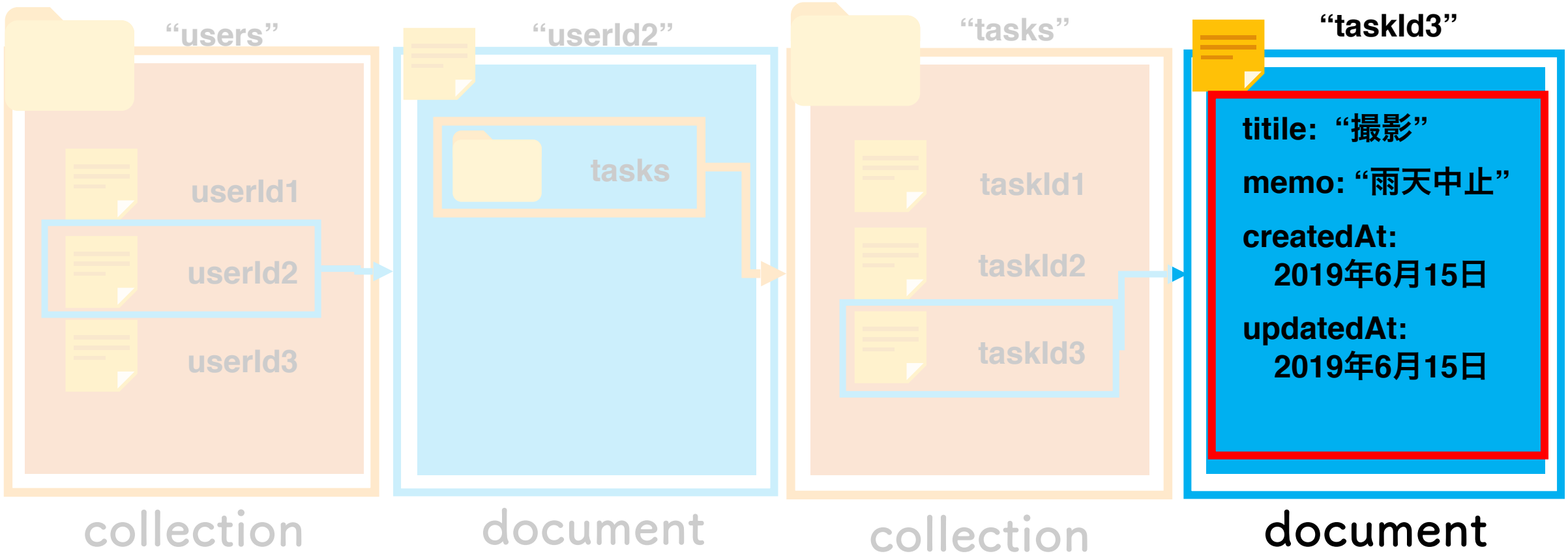
データの変更箇所をチェックしてみよう



このTaskIDというのは

現在存在しないので用意する必要がありそう

データの変更箇所をチェックしてみよう



IDの他に項目増えてるし

データの形式も変える必要がありそう

やること＜前半：Taskモデルに最低限の変更をかける＞

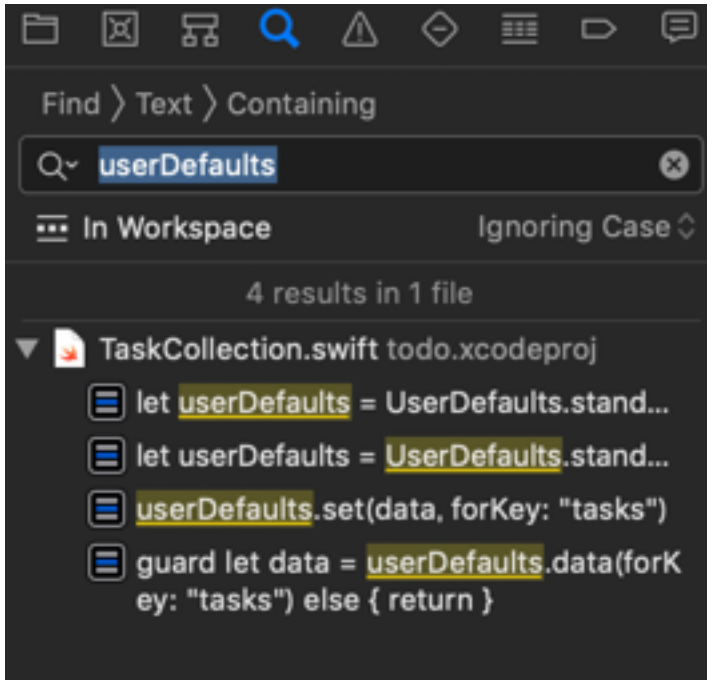
①UserDefaults関連の箇所を削除

②タスクにIDを追加
　タスクに作成日・更新日を追加
　初期化処理を変更

①UserDefaults関連の箇所を削除

TaskCollection

UserDefaults関連のコードはもう使わないので
削除 or コメントアウト



1. TaskCollection内の…

・UserDefaultsの宣言を削除
(コメントアウトでもOK)

Cmd + Shift + F で検索が便利！！

②タスクに作成日・更新日・IDを追加、初期化処理を変更

Task

DB保存時の定石なのでID・作成・更新日時を保存する

今回はFirestoreで取扱いやすいTimestamp型にしてみる

```
import Foundation
import FirebaseFirestore
import FirebaseFirestoreSwift

class Task: Codable {
    var id: String
    var title: String = ""
    var memo: String = ""
    var createdAt: Timestamp
    var updatedAt: Timestamp

 init(id: String) {
    self.id = id
    self.createdAt = Timestamp()
    self.updatedAt = Timestamp()
  }
}
```

プロパティを追加

初期化時に今の日時で

Timestampを作成する

処理を追加

②タスクに作成日・更新日・IDを追加、初期化処理を変更

2. AddVCでエラーをでている箇所を暫定的に修正

AddVC

```
        if let index = selectIndex {
            // Edit
            let editTask = Task(id: "一時的に修正")
            TaskCollection.shared.editTask(task: editTask, index: index)
        } else {
            // Add
            let task = Task(id: "一時的に修正")
            TaskCollection.shared.addTask(task)
        }
```

※一旦エラーが出ないようにしてるだけ

やること＜前半：Taskモデルに最低限の変更をかける＞

①UserDefaults関連の箇所を削除

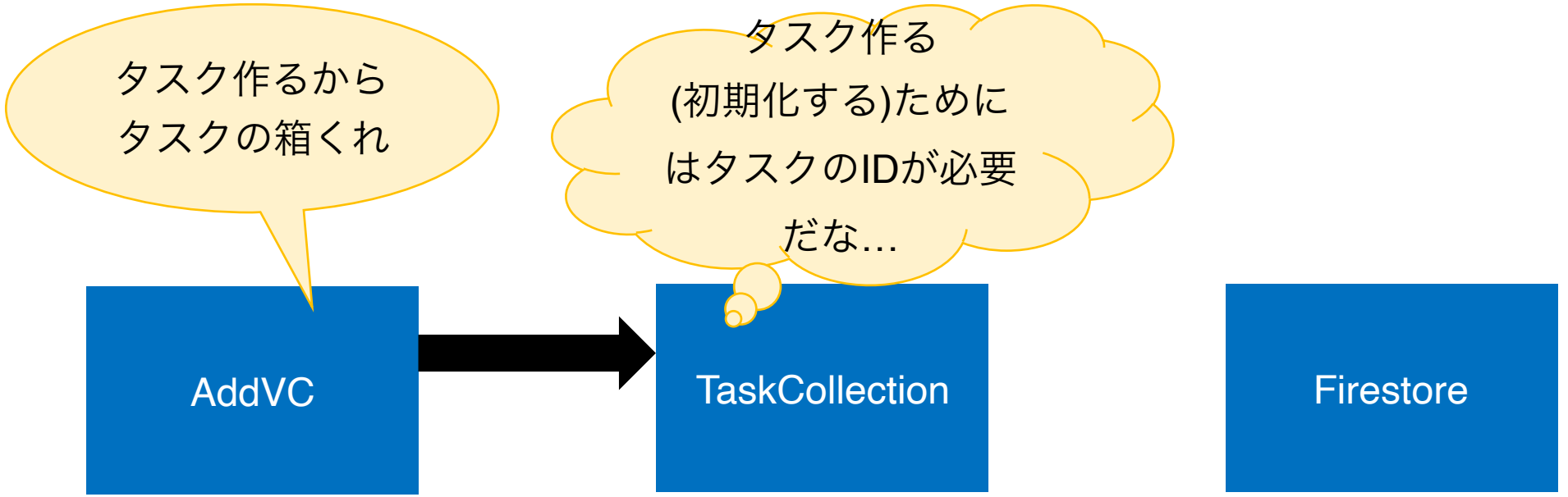
②タスクにIDを追加
　タスクに作成日・更新日を追加
　初期化処理を変更

完了

やること＜後半：処理の流れを作成する＞

①Firestoreとやりとりするファイルを作成
②ユーザーごとにタスクを保存する先のRef（参照）を
　作ってくれる関数を作成
③タスクIDを作る処理を作成
④IDで初期化した空のタスクを作る処理を作成
⑤Firestoreに追加する関数を作る＆Taskの追加修正
⑥AddVCの新規のフローを修正
⑦TaskCollection内のappend関数の処理を修正

続きに着手する前に新規登録の流れをチェック

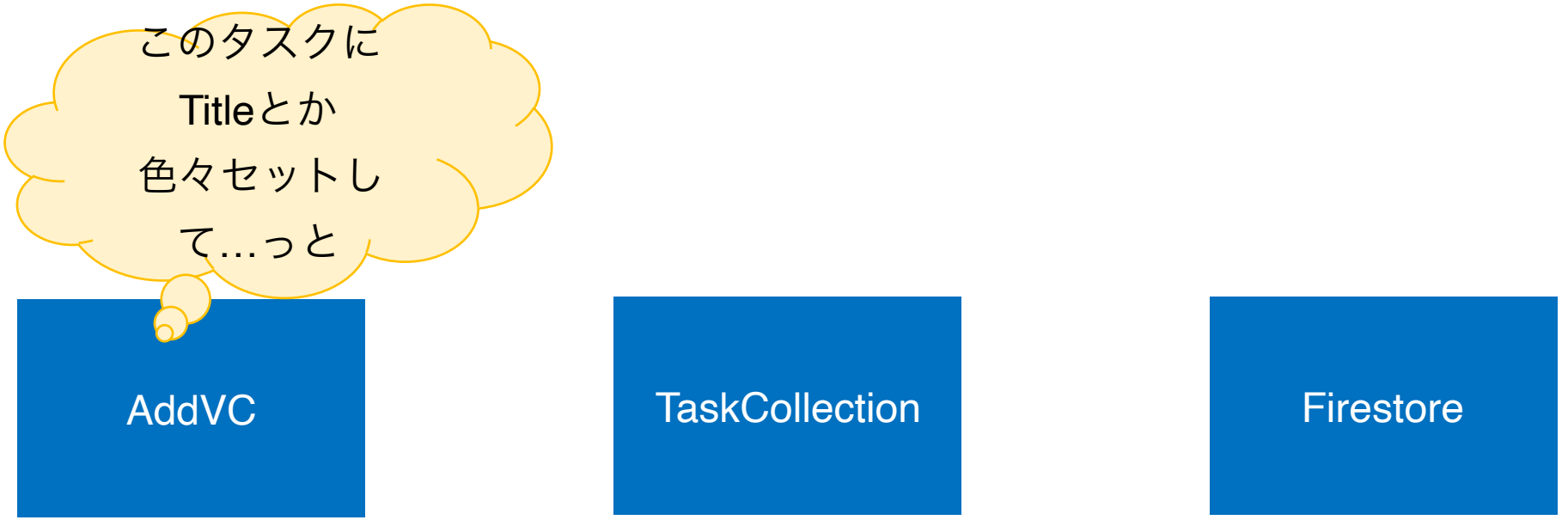


TaskCollectionに
初期化したタスクデータを返す処理が必要そう

続きに着手する前に新規登録の流れをチェック

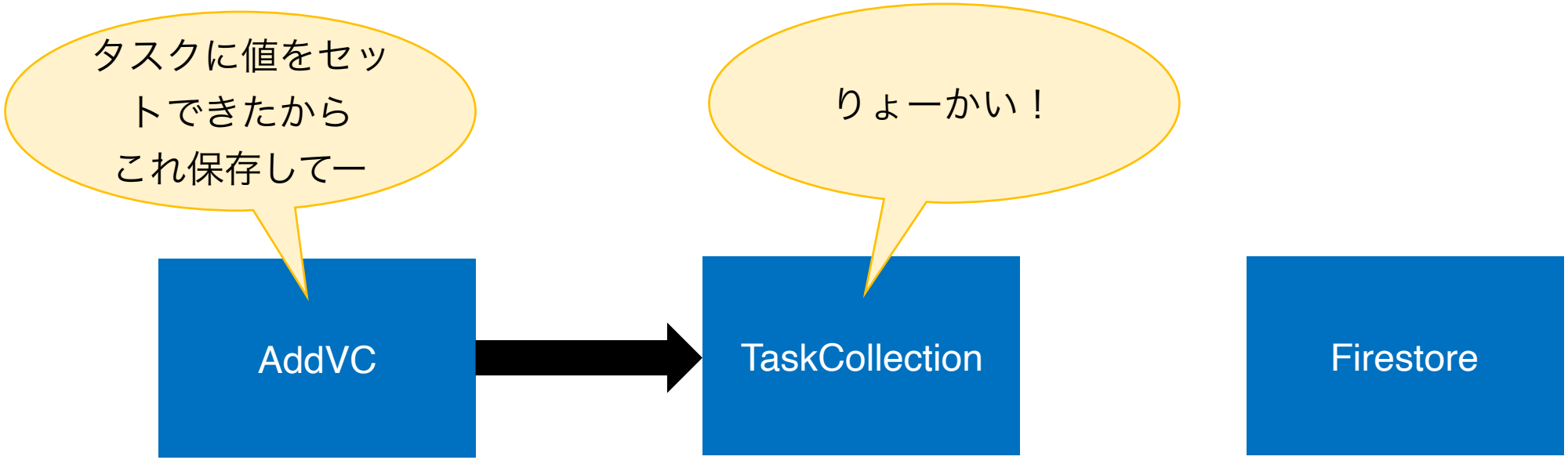
TaskIDを保存できるようにはなっているが
生成したりする仕組みを作ってないので用意する必要がありそう

続きに着手する前に新規登録の流れをチェック



さきほどTaskの項目を色々変えてしまったので
AddVCでタスクのプロパティをセットしている箇所に
色々と変更が必要そうな予感

続きに着手する前に新規登録の流れをチェック



続きに着手する前に新規登録の流れをチェック

今のままだと

ローカルのTaskCollectionが更新されるだけなので

Firestoreに保存する処理を追加しないとまずそうだ

①Firestoreとやりとりするファイルを作成

TaskUseCase

サーバーサイドとのやりとりを
「TaskUseCase」というファイルで行うことにする

1-1. 作成したらとりあえずFirestoreをインポート

```
import FirebaseFirestore
```

1-2. データベースを用意する

```
let db = Firestore.firestore()
```

②ユーザーごとにタスクを保存する先のRef（参照）を作ってくれる関数を作成

TaskUseCase

ユーザーごとの保存先のRefを取得する関数を作成

```
import FirebaseAuth
```

```
private func getCollectionRef () -> CollectionReference {
    guard let uid = Auth.auth().currentUser?.uid else {
        fatalError ("Uidを取得出来ませんでした。")
    }
    return self.db.collection("users").document(uid).collection("tasks")
}
```

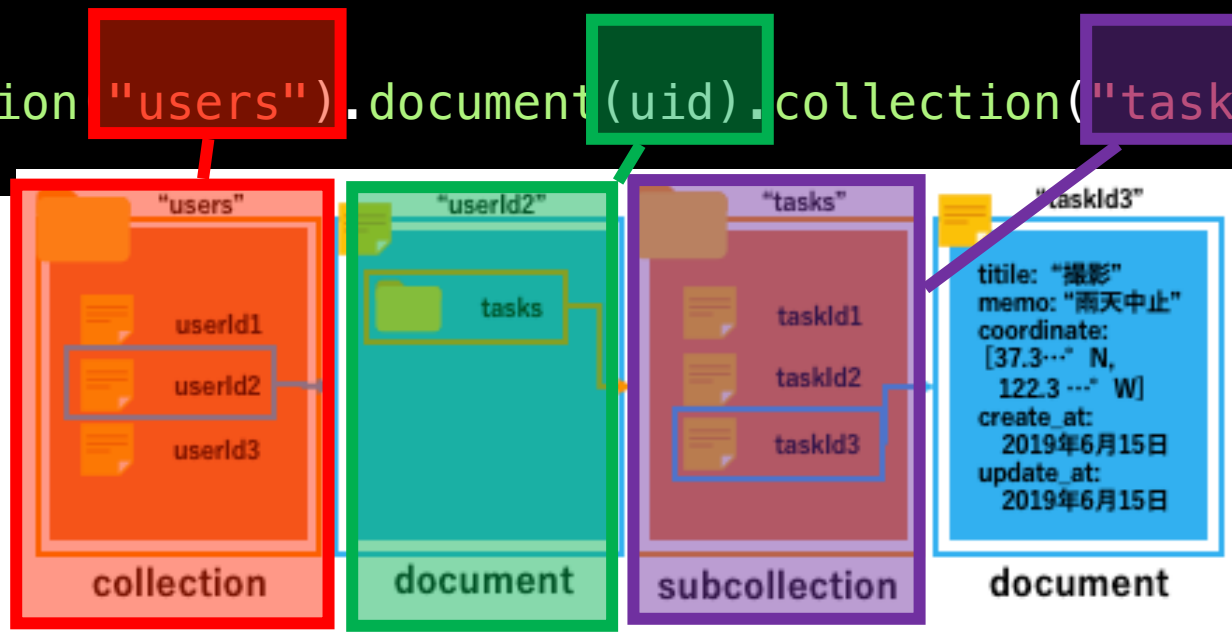
②ユーザーごとにタスクを保存する先のRef（参照）を作ってくれる関数を作成

TaskUseCase

ユーザーごとの保存先のRefを取得する関数を作成

```
import FirebaseAuth
```

```
private func getCollectionRef () -> CollectionReference {
    guard let uid = Auth.auth().currentUser?.uid else {
        fatalError ("Uidを取得出来ませんでした。")
    }
    return self.db.collection("users").document(uid).collection("tasks")
}
```



残りはTaskIDを指定して
データを保存するとこだ😳

参考：UIDを取る処理をAuthUserモデル等で実装する場合 AuthUser

ユーザーのモデルを用意する場合はそこにロジックをもたせてもよい
（こちらの方が責務は分かりやすい）

```
import FirebaseAuth
```

```
class AuthUser {

    static let shared = AuthUser()

    private init(){}

    var authUser: FirebaseAuth.User? {
        get {
            return Auth.auth().currentUser
        }
    }
}
```

③タスクIDを作る処理を作成 TaskUseCase

タスクのIDを作ってくれる処理を作成する

```
func createTaskId() -> String {
    let id = self.getCollectionRef().document().documentID
    print("taskIdは",id)

    return id
}
```

※今回、データベースっぽく、先にユニークIDを作っていますが
Firestoreの「addDocument」などを利用して、
先にIDを作らずに保存時に自動でIDを採番させる方法もあります
https://firebase.google.com/docs/firestore/manage-data/add-data?hl=ja#add_a_document

④IDで初期化した空のタスクを作る処理を作成

TaskCollection

4-1. TaskCollectionからTaskUseCaseを呼び出せるようにする

4-2. 既に作ってあるcreateTaskId関数でIDを生成し、
そのIDでTaskを初期化して生成する関数を作成

```
let taskUseCase: TaskUseCase


private init() {
    taskUseCase = TaskUseCase()
    load()
}


func createTask() -> Task {
    let id = taskUseCase.createTaskId()
    return Task(id: id)
}
```

⑤Firestoreに追加する関数を作る＆Taskの追加修正

TaskUseCase

5-1. Firestoreに新規登録する関数 addTaskを作る（不完全）

```
func addTask(_ task: Task){
    let documentRef = self.getCollectionRef().document(task.id)
    let encodeTask: [String: Any] = [:] //仮に入れてみているだけ(あとで消す)

    documentRef.setData(encodeTask) { (err) in
        if let _err = err {
            print(“データ追加失敗",_err)

        } else {
            print("データ追加成功")

        }
    }
}
```

⑤Firestoreに追加する関数を作る＆Taskの追加修正

TaskUseCase

5-1. Firestoreに新規登録する関数 addTaskを作る（不完全）

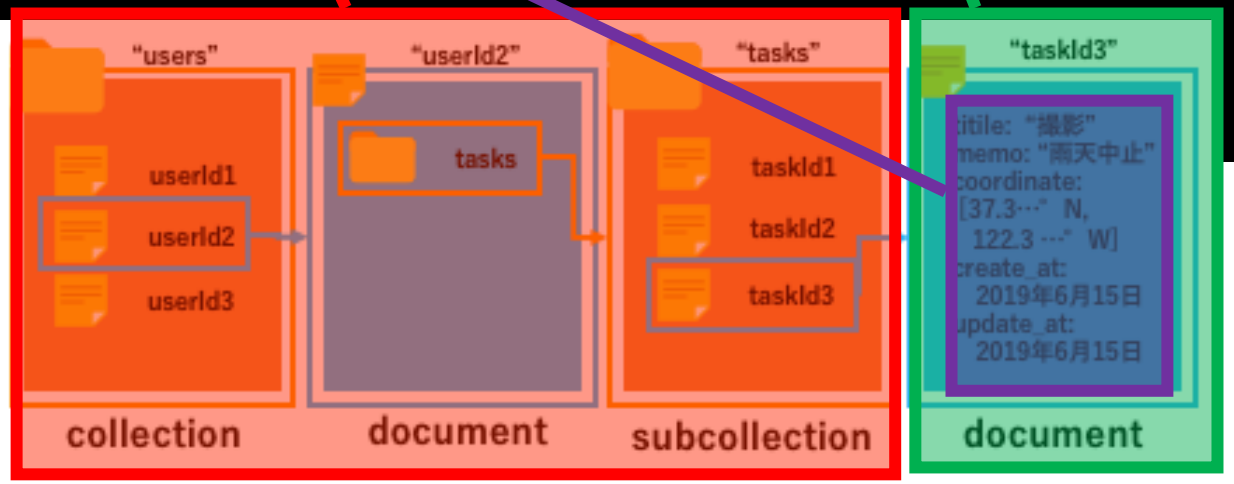
```
func addTask(_ task: Task){
    let documentRef = self.getCollectionRef().document(task.id)
    let encodeTask: [String: Any] = [:] //仮に入れてみているだけ（あとで消す）

    documentRef.setData(encodeTask) { (err) in
        if let _err = err {
            print("データ追加失敗",_err)

        } else {
            print("データ追加成功")

        }
    }
}
```

紫に入れるタスクデータを
Dictionaryで作れれば
保存できそう😳



⑤Firestoreに追加する関数を作る＆Taskの追加修正

Task

5-2. Firestoreへ保存するタスクの値を
辞書型で生成する関数を作成

```
func toValueDict() -> [String: Any] {
    return [
        "title": self.title as Any,
        "memo": self.memo as Any,
        "coordinate": self.coordinate as Any,
        "create_at": self.createAt,
        "update_at": self.updateAt,
    ]
}
```

いらない

⑤Firestoreに追加する関数を作る＆Taskの追加修正

TaskUseCase

5-3. addTask関数内で
タスクのデータをセットするように修正

```
import FirebaseFirestoreSwift
```

```
func addTask(_ task: Task){
    let documentRef = self.getCollectionRef().document(task.id)
    let encodeTask = try! Firestore.Encoder().encode(task)
    documentRef.setData(encodeTask) { (err) in
        if let _err = err {
            print("データ追加失敗",_err)
        } else {
            print("データ追加成功")
        }
    }
}
```

⑥AddVCの新規のフローを修正

AddViewController

新規のときにIDで初期化したタスクを
TaskCollectionからもらうように修正

```
    // ここで Edit か Add  かを判定している
        if let index = selectIndex {
            // Edit
            let editTask = Task(id: “一時的に修正”)
            TaskCollection.shared.editTask(task: editTask, index: index)
        } else {
            // Add
            let task = TaskCollection.shared.createTask()
            task.title = title
            task.memo = memoTextView.text
            TaskCollection.shared.addTask(task)
        }
```

⑦TaskCollection内のappend関数の処理を修正

TaskCollection

ローカルのタスクを追加するだけでなくて
Firestoreも更新するように修正

```
    func addTask(_ task: Task) {
        tasks.append(task)
        taskUseCase.addTask(task)
        save()
    }
```

ここまでできたら、タスクを登録したときに
Firestoreへ反映されるかを確認してみよう

【一緒にやってみよう】

FirestoreのTODOのデータを
「更新」してみよう

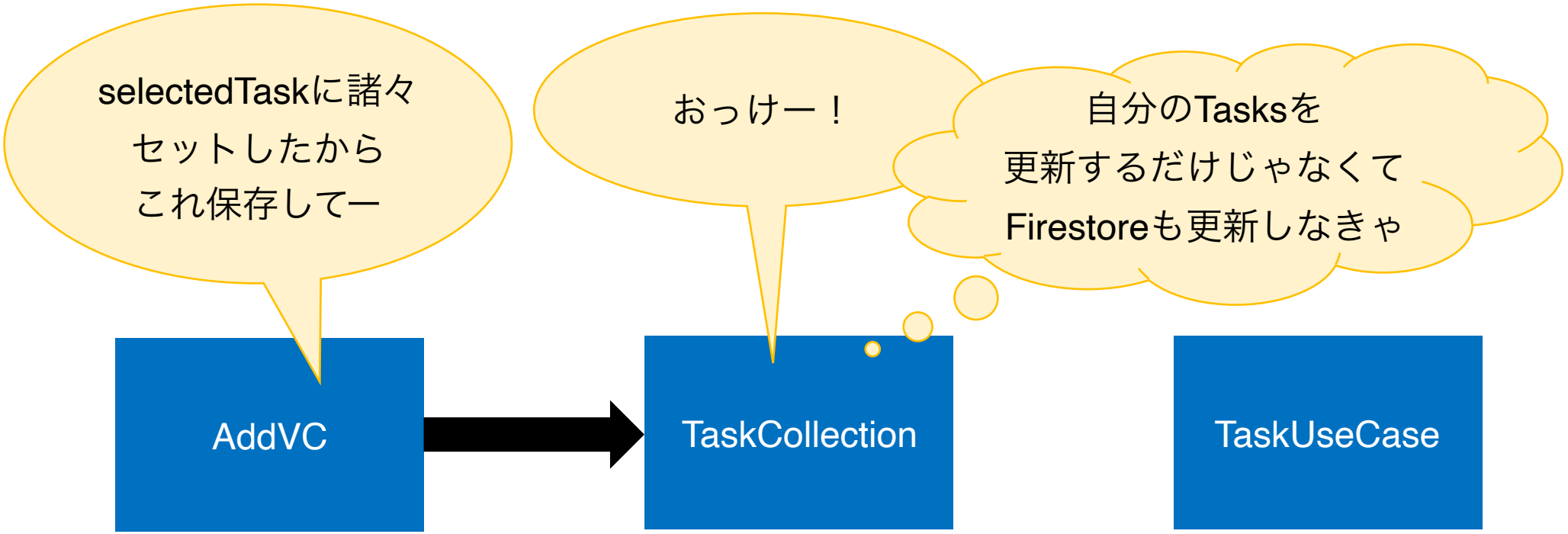
※完成PJは配布するので
ついてこれなくなったら
手を止めて見ることに集中！

やること

①TaskUseCaseに更新処理を作成

②TaskCollectionから更新処理を呼び出す

続きに着手する前に編集保存の流れをチェック



続きに着手する前に編集保存の流れをチェック

今のままだと

ローカルのTaskCollectionが更新されるだけなので

Firestoreも更新する処理を追加しないとまずそうだ

①TaskUseCaseに更新処理を作成

TaskUseCase

```
    func editTask(_ task: Task){
        let documentRef = self.getCollectionRef().document(task.id)
        let encodeTask = try! Firestore.Encoder().encode(task)
        documentRef.updateData(encodeTask) { (err) in
            if let _err = err {
                print("データ修正失敗",_err)
            } else {
                print("データ修正成功")
            }
        }
    }
```

②TaskCollectionから更新処理を呼び出す

TaskCollection

2-1. 関数にタスクの引数を作る

2-2. TaskUseCaseのeditTask関数を呼び出して
引数に編集対象のタスクを入れる

```
func editTask(task: Task, index: Int) {
    tasks[index] = task
    taskUseCase.editTask(task)
    save()
}
```

③AddVCのeditTask を修正

AddViewController

3-1. editTaskにセット

3-2. タスクの更新日時をセットするように修正

```
    // ここで Edit か Add  かを判定している
        if let index = selectIndex {
            // Edit
            let editTask = TaskCollection.shared.getTask(at: index)
            editTask.title = title
            editTask.memo = memoTextView.text
            editTask.updatedAt = Timestamp()
            TaskCollection.shared.editTask(task: editTask, index: index)
        } else {
            // Add
```

ここまでできたら、タスクを登録後

タスクを修正したときに修正が反映されるか

ダッシュボードで確認してみよう

【(時間があれば)一緒にやってみよう】

FirestoreからTODOのデータを
「削除」してみよう

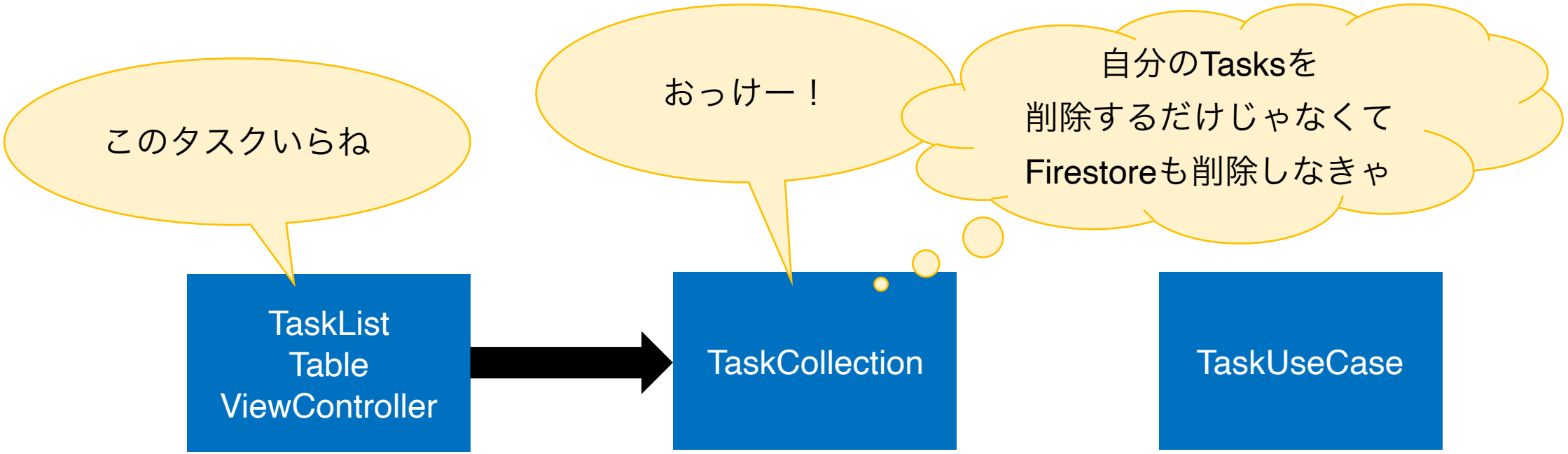
※完成PJは配布するので
ついてこれなくなったら
手を止めて見ることに集中！

やること

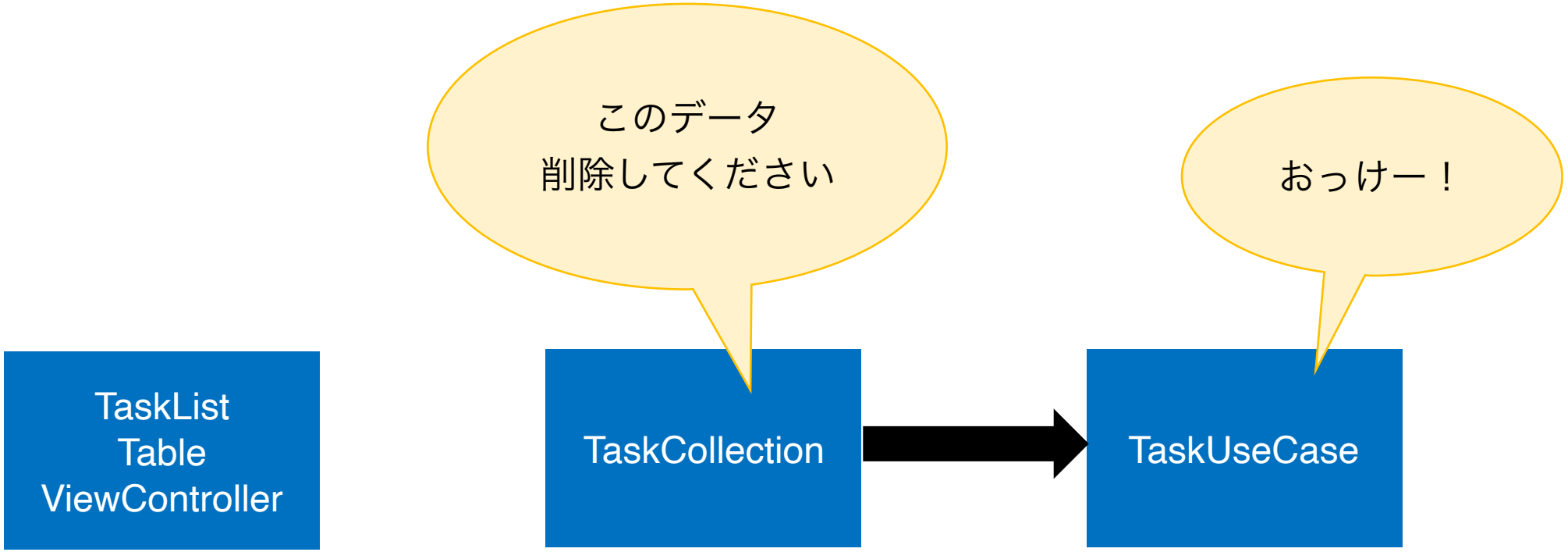
①TaskUseCaseに削除処理を作成

②TaskCollectionから削除処理を呼び出す

続きに着手する前に削除の流れをチェック



続きに着手する前に削除の流れをチェック



①TaskUseCaseに削除処理を作成

TaskUseCase

タスクを削除する処理を作成

```
func removeTask(taskId: String){
        let documentRef = self.getCollectionRef().document(taskId)
        documentRef.delete { (err) in
            if let _err = err {
                print("データ取得",_err)
            } else {
                print("データ削除成功")
            }
        }
    }
```

TaskCollection

TaskCollectionからTaskUseCaseの処理を呼び出す

```
    func removeTask(index: Int) {
        tasks.remove(at: index)
        taskUseCase.removeTask(taskId: tasks[index].id)
        save()
    }
```

ここまでできたら、タスクを登録後
タスクを削除したときに削除が反映されるか
ダッシュボードで確認してみよう

【一緒にやってみよう】

FirestoreからTODOのデータを
「読み込み」してみよう

※完成PJは配布するので
ついてこれなくなったら
手を止めて見ることに集中！

やること

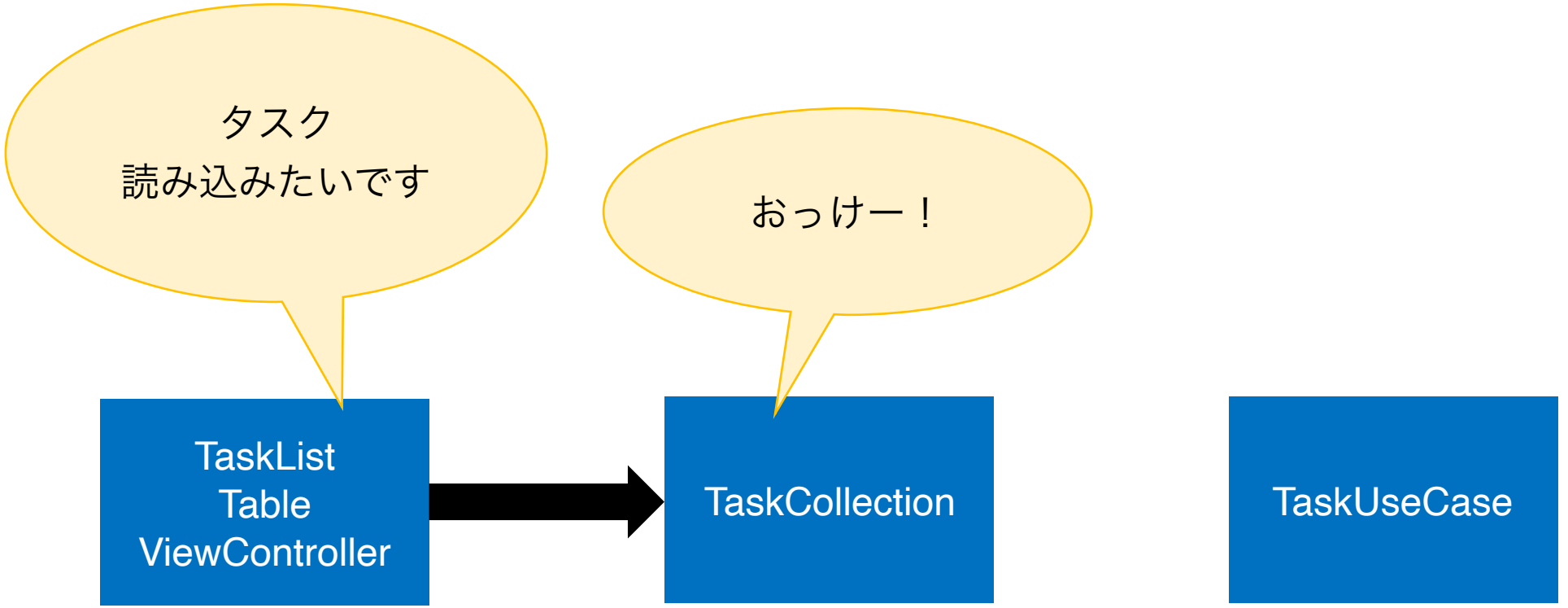
①Taskの初期化処理を修正

②TaskUseCaseに読み込み処理を作成

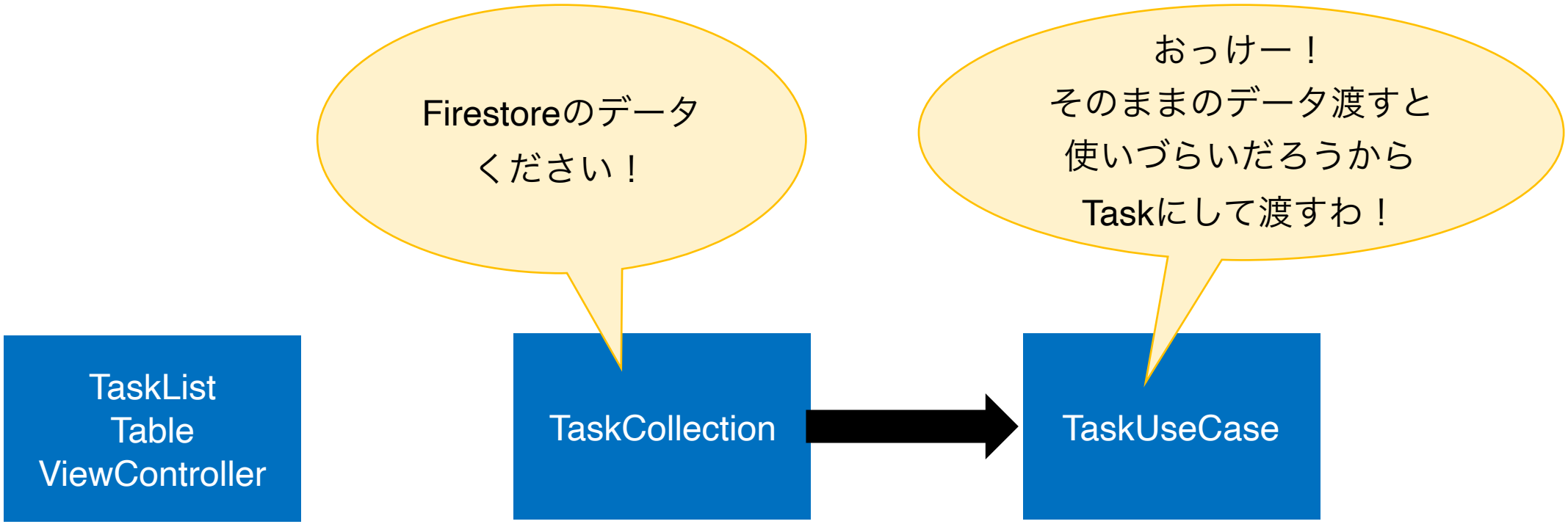
③TaskCollectionから更新処理を呼び出す

④Taskの並び替えをする

続きに着手する前に読み込みの流れをチェック



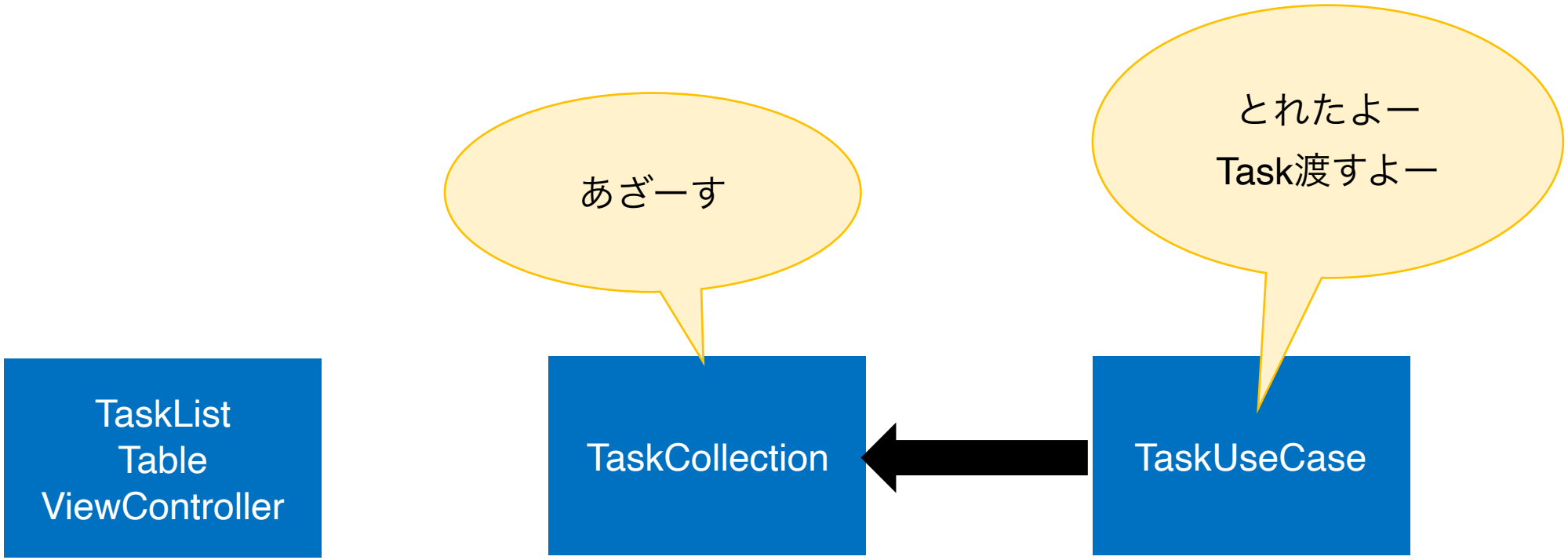
続きに着手する前に読み込みの流れをチェック



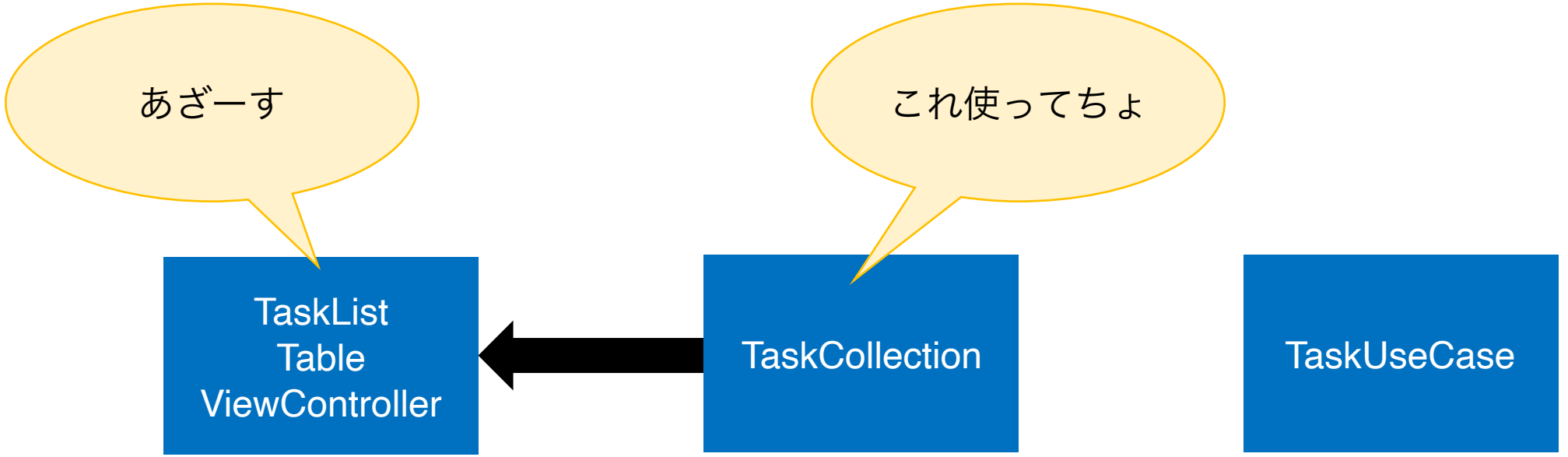
Firestoreから読み込む処理に加えて

それを扱いやすくするためにTaskに変換する処理が必要そう

続きに着手する前に読み込みの流れをチェック



続きに着手する前に読み込みの流れをチェック



②TaskUseCaseに読み込み処理を作成

TaskUseCase

Firestoreから全タスクをフェッチする処理を作成

```
func fetchTaskDocuments(callback: @escaping ([Task]?) -> Void){
    let collectionRef = getCollectionRef()
    collectionRef.getDocuments(source: .default) { (snapshot, err) in
        guard err == nil, let snapshot = snapshot,!snapshot.isEmpty else {
            print("データ取得失敗",err.debugDescription)
            callback(nil)
            return
        }

        print("データ取得成功")
        let tasks = snapshot.documents.compactMap { snapshot in
            return try? Firestore.Decoder().decode(Task.self, from: snapshot.data())
        }
        callback(tasks)
    }
}
```

補足：クロージャーについて

関数として扱える変数みたいなもん

```
//関数
func functionSample() {
    print("こんにちは")
}
functionSample()

//クロージャ
let closureSample: () -> () = {
    print("こんにちは")
}
closureSample()
```

```
//関数
func functionSample(_ p1:Int, _ p2:Int)->Int{
    return p1 + p2
}
print(functionSample(1,3))


//クロージャー
let closureSample: (Int,Int) -> Int = {
    p1,p2 -> Int in
    return p1 + p2
}
print(closureSample(1,3))
```

【swift】イラストで分かる！具体的なClosureの使い方。

https://qiita.com/narukun/items/b1b6ec856aee42767694

補足：@escapingについて

- クロージャがスコープから抜けても
  存在し続けるときに「@escaping」が必要

- 存在し続けるときとは以下のいずれか
  ①クロージャーがプロパティとして保存される
  ②クロージャーが非同期的に実行される(メソッド内ですぐに実行されない)

```
func fetchTaskDocuments(callback: @escaping ([Task]?) -> Void) {
    let collectionRef = getCollectionRef()
    collectionRef.getDocuments(source: .default) { (snapshot, err) in
        guard err == nil,
            let _snapshot = snapshot,
            !_snapshot.isEmpty else {
                callback(nil)
                return
        }
        let taskCollection: [Task] = _snapshot.documents.compactMap{ (snapshot) in
            let id = snapshot.documentID
            let value = snapshot.data()
            return Task(id : id ,value: value)
        }
        callback(taskCollection)
    }
}
```

補足：CompactMapについて

「map」では配列の全てに○○できる

```
let numbers = [1, 2, 3, 4]
let mapped = numbers.map { $0 * 5 }
// [5, 10, 15, 20]
```

「compactMap」ではnilのものを排除しつつmapしてくれる

```
// ★MAPの場合
let mapValues = ["1", "2", "3", "more"]
let mapped = mapValues.map { Int($0) }
// [Optional(1), Optional(2), Optional(3), nil]

// ★COMPACTMAPの場合
let compactValues = ["1", "2", "3", "more"]
let compactMapped = compactValues.compactMap { Int($0) }
// [1, 2, 3]
```

③TaskCollectionから更新処理を呼び出す

TaskCollection

```
    private func load() {
        taskUseCase.fetchTaskDocuments { (tasks) in
            guard let tasks = tasks else {
                self.save()
                return
            }
            self.tasks = tasks
            self.delegate?.loaded()
        }
    }
```

TaskListVC の load を削除（必要なくなったので）

```
        #warning("ロードする")
        TaskCollection.shared.load()
```

④Taskの並び替えをする

TaskCollection

```
    private func save() {
        tasks = tasks.sorted(by: {$0.updatedAt.dateValue() > $1.updatedAt.dateValue()})
        delegate?.saved()
    }

    private func load() {
        taskUseCase.fetchTaskDocuments { (tasks) in
            guard let tasks = tasks else {
                self.save()
                return
            }
            self.tasks = tasks.sorted(by: {$0.updatedAt.dateValue() > $1.updatedAt.dateValue()})
            self.delegate?.loaded()
        }
    }
```

ここまでできたら、

起動してダッシュボードに登録されている

データが表示されるか確認してみよう

ここまでのおさらい

こんなことを学びました

①Firestoreへのデータ追加

②Firestoreのデータ更新

③Firestoreのデータ削除

④Firestoreのデータ読み込み

Firestoreの
セキュリティルールについて知る

Firestoreのセキュリティルールとは？

誰がどこのデータを読み書きできるか？
というルールを設定することで
堅牢なデータベースを構築することができる

ダッシュボード内で簡単なコードを書くことで
セキュリティルールを構築することができる

Cloud Firestore セキュリティ ルールを使ってみる

https://firebase.google.com/docs/firestore/security/get-started?hl=ja

セキュリティルールはどんなことができる？

- Write(Create/Update/Delete)およびRead(Get/List)を設定できる

- 対象のドキュメントをURLのような形で指定することができる

- もし●●だったら、○権限を付与といったようなことができる
  （ex: もし登録ユーザーだったら読み込み権限を付与）

- 基本は全て拒否して、許可する箇所だけ許可するというのが正しいやり方
  （拒否するところだけ定義していくとルールに漏れがでやすい）

- 他のドキュメントからデータを取ってきて、それをルールに組み込んだり、
  内部で関数を作ることなども可能

テストモードのデフォルトのルールを見てみる

```
① service cloud.firestore {
  ② match /databases/{database}/documents {
    ③match /{document=**} {
        //ルート以下の全てのドキュメントは誰でも読み書きOK
      ④allow read, write:
      }
    }
}
```

①サービスを識別している

②データベースのルートを指定している（この辺までは決まったおまじない）

③ワイルドカードを使っていて「全てのドキュメントで…」という意味

④読み込み、書き込みを許可している

→このPJのFirestoreでは全てのユーザーがどこでも読み書きできる　という意味

…こりゃ危険ですよね😳

ToDoアプリ用にルールを設定してみる

```
service cloud.firestore {
  match /databases/{database}/documents {

    allow read, write: if false;

    function isAuthenticated() {
      return request.auth != null;
    }

    match /users/{userId}/tasks/{taskId} {

      function isUserAuthenticated() {
        return request.auth.uid == userId;
      }

      allow read, write: if isAuthenticated() && isUserAuthenticated();
    }
  }
}
```

全て禁止！

ユーザーが認証されているか
チェックする関数

ドキュメントを指定

上のuserIdとリクエスト元の
ユーザIDが同一か
チェックする関数

関数が両方trueだったら
読み込みと書き込みを
許可している

Firestoreのその他の主要機能
クエリ・並び替え・監視について知る

Firestoreのクエリ

whereFieldメソッドを利用すると取得するデータにクエリをかけることができる

```
let query1 = db.collection("cities").whereField("state", isEqualTo: "CA")
let query2 = db.collection("cities").whereField("population", isLessThan: 100000)
let query3 = db.collection("cities").whereField("name", isGreaterThanOrEqualTo: "San Francisco")
let query4 = db.collection("cities").whereField("state", isEqualTo: "CO")
              .whereField("name", isEqualTo: "Denver")
```

Cloud Firestore で単純なクエリと複合クエリを実行する

https://firebase.google.com/docs/firestore/query-data/queries?hl=ja

Firestoreの並び替え

orderbyで並び替え、limitでクエリへの制限をかけることができる

```
let ref = db.collection("cities").order(by: "name", descending: true).limit(to: 3)
```

※ドキュメントをnameフィールドで降順に並べて最後の３件をとっている

Cloud Firestore でのデータの並べ替えと制限

https://firebase.google.com/docs/firestore/query-data/order-limit-data?hl=ja

Firestoreの監視

addSnapshotListenerメソッドを使うことで変更をリッスンすることができる

```
db.collection("cities").document("SF")
    .addSnapshotListener { documentSnapshot, error in
        guard let document = documentSnapshot else {
            print("Error fetching document: \(error!)")
            return
        }
        print("Current data: \(document.data())")
}
```

※documentChangesを使うと追加/変更/削除で処理の切り分けも可能

Cloud Firestore でリアルタイム アップデートを入手する

https://firebase.google.com/docs/firestore/query-data/listen?hl=ja

ここまでのおさらい

こんなことを学びました

①Firestoreのセキュリティルール

②Firestoreのその他の便利機能

【次回までの課題】

自由

※ できればFirebaseでサーバーも含めたアプリ開発
※ 卒業制作のプロトタイプを作り始めてもよいです

※注意点
GoogleService-Info.plistをプッシュしない

【次回の予告】

Cloud Storageを使って
画像データを保存します