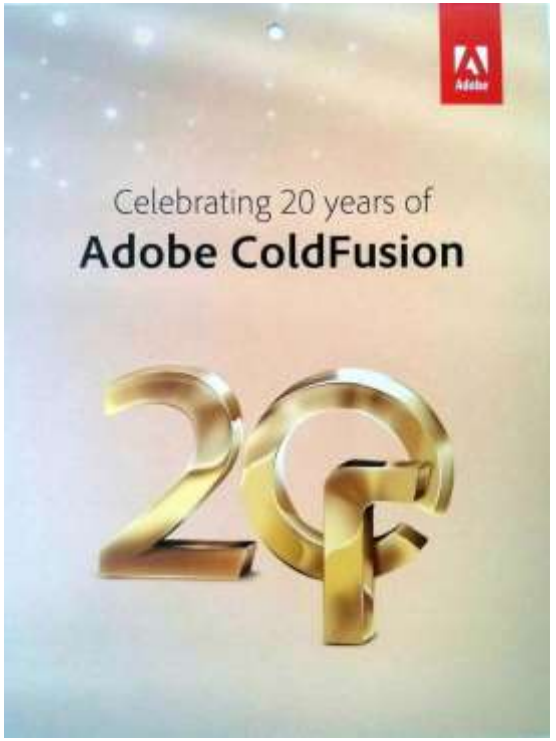
ColdFusion20周年記念
JCFUG ユーザー会

2015.08.07



CFモダン化計画 :
モダンColdFusionコーディング入門

～cfcの使い方編～

CFモダン化計画

- モダン感
  - オブジェクト指向
  - 関数型言語
  - 自動化（テスト、デプロイ、その他）
  - アジャイル/スパイラル
  - 意識高い系開発者
- レガシー感
  - 構造化言語止まり
  - 手動（テスト、デプロイ、全部）
  - ウォーターフォール
  - 土方系開発者

なぜオブジェクト指向にならないのか？

- 設計の問題。
- 既に存在しているソースがそうなっていない。
- cfcを使ってはいるが、ただの関数置き場になっている。
- そもそもWebアーキテクチャと相性が悪い？
- なぜオブジェクト指向にしないといけないのか？

Webでオブジェクト指向の問題点

- RDBMSとオブジェクト指向の話は相性が悪い。
  - オブジェクトにひたすら値を詰める本質的ではない作業。
  - ORMを使っても、大量データのパフォーマンス問題など、簡単には全てが解消しない。
- RDBMSとオブジェクトのギャップ部分を最初から扱うと、労多く報いが少ない結果になりがち。

Webでオブジェクト指向の問題点

- Webはステートレスなので、オブジェクトが毎回破棄されてしまったりと、何かと扱いにくい。
  - 基本的にはどうしようもないことだが、それでも使えるポイントはありそう。

Webでオブジェクト指向の問題点

- 今回は、オブジェクトの良さが活きやすいDB以外の処理について考える。

ソフトウェア開発の問題点

- ポイント
  - ソフトウェア開発において、要求は常に変化する。
    - 要求が変化しないのは死んだソフトウェアだけ。
  - 要求の変化に抗うことはできない。
    - 変化を禁止するのではなく、変化に強くする。

オブジェクト指向以前

- 構造化言語：機能分解でのアプローチ
  - 問題を小さな機能（モジュール）に分解していく。
    - ピラミッドのような階層構造
  - 各モジュールを統括するメインモジュールが必要となり、メインモジュールが複雑化する。
  - 機能そのものに着目
    - 機能やデータを変更すると、その影響が連鎖的に波及し易く、影響範囲が見極めにくい。

オブジェクト指向でのアプローチ

- ソフトウェア開発プロセスにおける3つの観点
  - 概念
    - ソウトウェアではなく、問題領域における概念の分析
    - 「私は何に対して責任があるのか？」
  - 仕様
    - 内部実装ではなく、インターフェース
    - 「私はどのように使用されるのか？」
      - 内部で何をしているかは知らなくてよい。
  - 実装
    - 具体的なソースコード
    - 「私はどのようにして自身の責任を全うするのか？」

オブジェクト指向でのアプローチ

- オブジェクト指向言語：オブジェクトへの分解
  - オブジェクトとは？
    - 要求上のある責任（概念）を担保する実体であり、
    - メソッド/インターフェース（仕様）の集合であり、
    - コード/アルゴリズムとデータ（実装）である。
  - これらの要素が言語機能レベルでマッピングされているプログラム言語
    - システムにおける抽象概念を言語要素として取り込む。
  - オブジェクト（責務を持った実体）に着目

今回の題材

- ユーザーのログイン処理
  - 要求
    - ユーザーという利用者を模した概念があってログインしたりしなかったりする。
    - 一度ログインするとしばらくログインしっぱなしになる。
    - ユーザーがログイン済みかどうかを判別したい。
  - 機能
    - ユーザーがログインする。
    - セッションにUserIDを記憶。
    - 次以降のリクエストでどのユーザーとしてログインしているかをUserIDで判別。

ユーザーのログイン処理

- ユーザーがログインする

```
<cfquery name="qUser">
        SELECT userID
        FROM UserData
        WHERE UserData.loginID = #Form.loginID#
</cfquery>

<cfif qUser.RecordCount eq 1>
        ～
</cfif>
```

- セッションにUserIDを記憶

```
<cfset session.userID = qUser.userID>
```

- 次以降のリクエストでどのユーザーとしてログインしているかをUserIDで判別

```
<cfif StructKeyExists(session, "userID") and IsNumeric(session.userID)>
        <cfquery>
          SELECT userName
          FROM UserData
          WHERE UserData.userID = #session.userID#
        </cfquery>
</cfif>
```

- 何回も使うのでudfにまとめて・・・

ユーザーのログイン処理

- ユーザーがログインする
```
<cfquery name="qUser">
        SELECT userName
        FROM UserData
        WHERE UserData.loginID = #Form.loginID#
</cfquery>

<cfif qUser.RecordCount eq 1>

</cfif>
```
- セッションにUserIDを記憶
```
<cfset session.userID = qUser.userID>
```
- 次以降のリクエストでどのユーザーとしてログインしているかをUserIDで判別
```
<cfif StructKeyExists(session, "userID") and IsNumeric(session.userID)>
        <cfquery>
          SELECT userName
          FROM UserData
          WHERE UserData.userID = #session.userID#
        </cfquery>
</cfif>
```
- 何回も使うのでudfにまとめて・・・

生き急いでいる

ユーザーのログイン処理

- いきなり実装詳細に行かず、一旦整理する。

ユーザーのログイン処理

- 要求
  - ユーザーという利用者を模した概念があってログインしたりしなかったりする。
  - 一度ログインするとしばらくログインしっぱなしになる。
  - ユーザーがログイン済みかどうかを判別したい。
- 概念
  - ユーザーという概念の責任範囲
    - ユーザーIDや名前などを持っている。
    - ログインを行う。
    - ログイン状態が保持される？
- 仕様
  - ユーザーID、名前などを取得可能。
  - ログイン処理を行う機能がある。
  - ログイン状態が保持され、それを判別可能。
- 実装
  - Userオブジェクト
    - getUserID()
    - getUserName()
    - login()
    - isLoggedin()
  - ログイン状態はSessionスコープに保存
    - クラスタ化した場合、Clientスコープに保存という可能性も？

ユーザーのログイン処理

- User.cfc
  - プロパティ
    - userID
    - userName
    - loginStatus
  - getUserID()
    <cfreturn userID>
  - getUserName()
    <cfreturn userName>
  - isLoggedin()
    <cfreturn loginStatus>
  - login(loginID)

```
<cfquery name="qUser">
  SELECT userName
  FROM UserData
  WHERE UserData.loginID = #arguments.loginID#
</cfquery>

<cfif qUser.RecordCount eq 1>
  <cfset userID = qUser.userID>
  <cfset userName = qUser.userName>
  <cfset loginStatus = true>
<cfelse>
  <cfset userID = "">
  <cfset userName = "">
  <cfset loginStatus = false>
</cfif>
```

ユーザーのログイン処理

- ログイン状態の保存をどうするか？
  - スマホアプリ、デスクトップアプリの様にUserオブジェクトがメモリ内に保持されていればいいのだが、次のリクエスト時には消滅してしまう。
  - SessionスコープなどにUserオブジェクトそのものを保存したいところだが・・・
    - メモリ消費の問題やClientスコープには複合型を保存できない等の制約。
  - とりあえずUserIDのみ保存し、データはリクエストごとにDBから自動復帰する方向で考える。

ユーザーのログイン処理

- ログイン状態の保存をどうするか？
  - ユーザーという概念と状態保持の分離

ユーザーと状態保持の分離

- 状態の保存
  - 概念
    - クライアント、サーバ間での一時的な状態保持
  - 仕様
    - キーと値を渡して保存
    - キーを渡して保存した値の参照
  - 実装
    - 内部的にはSessionスコープを使用する。
      （呼び出し側はどこに保存しているかは関知しない）

ユーザーと状態保持の分離

- SessionStorage.cfc
  - setValue(key, value)

```
<cfset session[key] = value>
```

  - getValue(key)

```
<cfif StructKeyExists(session, key)>
  <cfreturn session[key]>
<cfelse>
  <cfreturn “”>
</cfif>
```

ユーザーと状態保持の分離

- User.cfc
  - プロパティの追加
    - storage
  - Init(storage)

```
<cfset storage = arguments.storage>

<cfif storage.getValue(“userID”) neq “”>
    userIDを元にDBからロードしてオブジェクト内に保持
</cfif>
```

ユーザーと状態保持の分離

- User.cfc
  - login(loginID)

```
<cfquery name="qUser">
  SELECT userName
  FROM UserData
  WHERE UserData.loginID = #arguments.loginID#
</cfquery>

<cfif qUser.RecordCount eq 1>
  <cfset userID = qUser.userID>
  <cfset userName = qUser.userName>
  <cfset loginStatus = true>
  <cfset storage.setValue("userID", qUser.userID)>
<cfelse>
  <cfset userID = "">
  <cfset userName = "">
  <cfset loginStatus = false>
  <cfset storage.setValue("userID", "")>
</cfif>
```

呼び出し側

- Application.cfc
  - 初期化

```
<cfset Request.user
  = CreateObject(“component”, “User”).init(
    CreateObject(“component”, “SessionStorage”).init()
)>
```

- index.cfm
  - ログイン処理
    - Request.user.login(Form.loginID)
  - 名前の表示
    - #Request.user.getUserName()#

呼び出し側

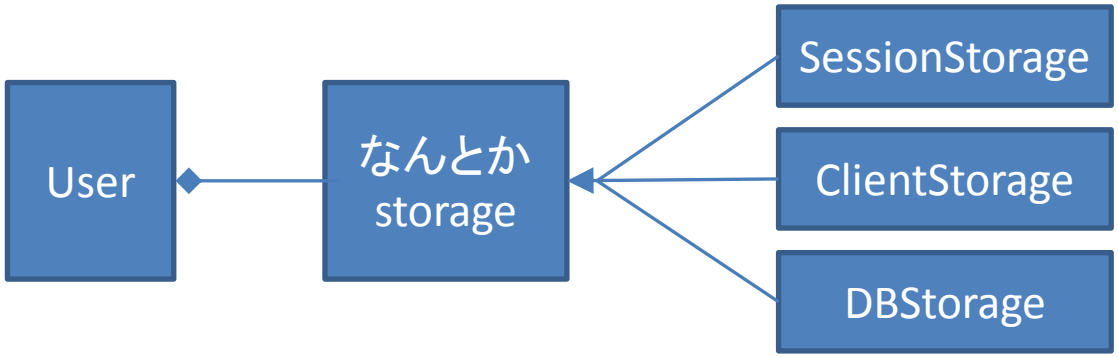
- サーバをクラスタ化して、セッション維持をClientスコープに変更した場合
  - 初期化

```
<cfset Request.user
  = CreateObject(“component”, “User”).init(
    CreateObject(“component”, “ClientStorage”).init()
)>
```

  - Sesssionスコープを使っているという実装の詳細を外部から隠蔽（カプセル化）しているため、他のコードはいじらなくてよい。

呼び出し側

- User：ユーザーという概念の表現
  - ユーザー名の参照やログインなどの機能を提供する。
  - 中で何をしているかは不明でよい。
- xxxStorage：セッション状態の保持
  - どこに保存しているかは不明でよい。
  - インターフェースや継承などで、機能呼び出し用のメソッド仕様を担保したまま、内部の挙動を変えられる。



まとめ

- オブジェクト指向が活きてくるポイント
  1. 概念、仕様、実装の3つの観点を意識して、設計の抽象度を維持する。
  2. オブジェクト同士の責任を分離し、疎結合の状態を保つ。
  3. 抽象度が上がった分、実装内容に隙間が生まれ、変更を差し込む余地が生まれる。
  4. カプセル化により、実装の詳細を封じ込めることで変更の余波を外部に漏らさない。

まとめ

- オブジェクト指向が活きてくるポイント
  1. 概念、仕様、実装の3つの観点を意識して、設計の抽象度を維持する。
  2. オブジェクト単位の責任と分離を組合わせて状態を保つ。
  3. 抽象度が上がった分、実装内容に隙間が生まれ、変更を差し込む余地が生まれる。
  4. カプセル化により、実装の詳細を封じ込めることで変更の余波を外部に漏らさない。

変更に強いソフトウェア

CFモダン化計画 :
モダンColdFusionコーディング入門

～cfcの使い方編～

ありがとうございました

参考文献

- ーデザインパターンとともに学ぶー
  オブジェクト指向のこころ
  - アラン・シャロウェイ (著), ジェームズ・R・トロット (著), 村上 雅章 (翻訳)
  - 丸善出版(2014/3/11) ￥4,104
  - **ISBN-10:** 4621066048
  - **ISBN-13:** 978-4621066041
  - 2005年9月に株式会社ピアソン桐原より出版された同名書籍を再出版したもの。