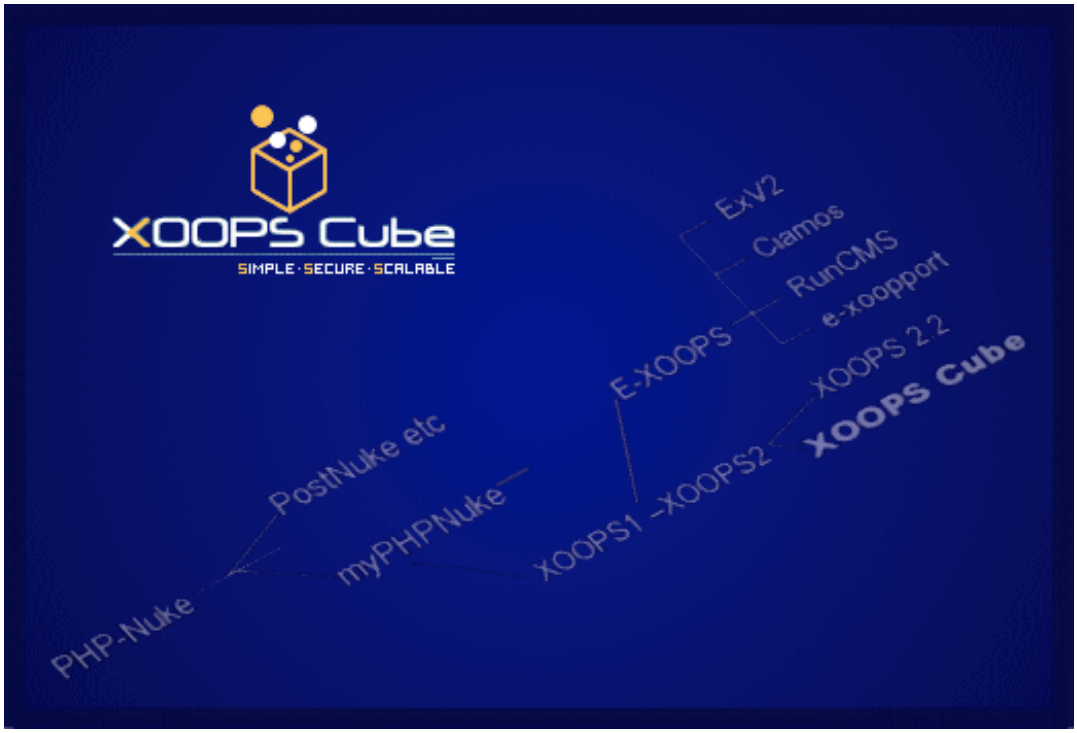
DOCUMENT VERSION 1.5.0

2.1 提案モデル



更新履歴

１１月８日０１：１０

- 「イベントとデリゲート」「イベントの高速化」「仮想サービス」を追加
- 一部のオムニグラフ・ベクトルデータファイルを損失 orz （いや、いいんですけど……）

１１月９日２３：４８ バージョン 1.2.1

- 「特別なイベント」セクションを追加
- 「イベントとデリゲート」セクションに「イベントリプレス」「アンカーデリゲート」の２項目を追加

１１月１０日００：３６　バージョン 1.2.2

- 目次追加　（まさか１クリックで追加できるとは……orz）

１１月１６日０３：３９　バージョン 1.3

- 全体概要から「フィーチャーメカニズム」を削り「仮想サービス」「イベント」を追加
- フィーチャーメカニズム関するセクションを削除
- 「デリゲート」を「仮想デリゲート」に名称変更
- 主なToDoを冒頭に追加
- その他、ドキュメント側に情報があり、コードと食い違っていたものを修正
- 「目標としたもの」セクションを追加

１１月２９日０１：５４　バージョン 1.4

- 「レンダーシステムクラス」セクションを改訂

１２月３０日２２：２４　バージョン 1.5

- 全般的に修正

主なToDo

- ルート/class ディレクトリ配下に追加したXCube名前空間に属するファイルおよびクラスをルート/kernelへ移動させる
- ブロックプロシージャーのアダプタがXCube名前空間のファイルで定義されているので互換モジュールへ移動
- 互換モジュールのうち、ルート/kernelの継承クラスとなるものはモジュール内にkernelディレクトリを作って移動させる
- baseモジュールはlegacyモジュールにファイル名、および空間名変更
- デバッガの定義位置を確認。また生成マネージャが不要。
- sitesettings.ini.phpの位置づけを確認
- PMマネージャー不要
- XCube空間にモジュールがない
- XoopsObjectとActionFormのgetVarの位置づけがどうしても異なる。生値やりとりのためのメソッド名を統一。
- 基底にトークンを入れて既存移植コードをトークン化（最後でいい）
- bbcode などのテキストエンコードを換装式にする方法の検討（一部検討実装済み）
- マニフェストにより各パーツの接合関係を診断するツールの開発

互換モジュール(base)

- Legacy の名前空間に統一が行われていない
- misc / search の代替が未踏項目になっている
- 元 system の管理機能の実装がかなり不足している
- 元コアモジュールと相互依存度が高すぎる
- レンダーシステムの複数起動・用途スイッチに対応していない
- キャッシュマネジメントと通信を行っていない
- 設計上換装方式をとっているにも関わらず、それをルートやコントローラに追加する過程が開発途上の関係でハードコーディングされている
- イベント通知機能互換を実装していない

- コメント機能互換は検討コードはあるが盛り込んでいない（レンダーバッファを用いてテンプレートで描画する方法へ変更）
- ハードコードされたHTMLのレンダーシステムへの委譲が終わっていない（現在作業中のものは設計変更。レンダーバッファを用いたテンプレートレンダリングに統一）
- コンフィグファイルからの起動設定が曖昧
- 一部 class に残ったままのクラスがあり、ものによっては kernel へ移動させなければならない
- base の権限がなければ他モジュールの Help を読み上げられない
- 管理メニューが整理された関係でコマンド検索のヒット率が低下している

ユーザーモジュール

- 旧 kernel に依存しており、移動させる必要がある
- 表側の実装がまだ終わっていない（アバターの変更などはできない）
- register.php は開発中のため、旧版と差し替えた
- 旧 system からユーザー系の機能移行が半分程度しか終わっていない
- このモジュールの権限がゲストに開放されていない場合ログインができない。デリゲートから直接要求に対する処理を実行するなど(misc.phpで使用している、デリゲート内で$controllerをとって実行）して解決するか、別の方法を考える
- メールメッセージのテンプレート化

PMモジュール

- 基本的に完了。ただし旧kernel依存、SQLをどちらが吐くか？などの旧コアに対する依存性の問題をファイルを適切に移動させるなどで対応する必要がある
- 一部デリゲートから直接処理を行っている関係で権限解決の解釈が異なっている操作があると思われる

X2互換テーマエンジン（レガシーレンダーシステム）

- キャッシュマネジメントの実装
- マルチカラム・フリーカラムに対する検討と実装
- HTMLハードコードの委譲待ち（現在のメソッドで委譲を受ける形はいかにもうまくないので、レンダーバッファ方式に変更）

- 独立したレンダーバッファを持つことで、モジュールがテーマ変数をオーバーライトできなくなった問題の解決が必要
- このモジュールの仕様によるプレースホルダが Smarty / class 下にあるのは全体に影響を与えるため、XOOPS固有のものは移動し、インスタンス生成時にセットする方法に変更

バナーモジュール

- 開発する必要がある

インストーラ

- 改良型インストーラをベースに新作する
- その際に必要なモジュールも同時にインストールすること
- ディストリビューションのためにインストーラビルダを開発する

旧コード群

- カーネル系、アダプタ系を除き、このペースでスクラッチを続ける
- 作業後に残ったところから NOTICE 関係の問題を取り除く
- NOTICE 問題を取り除くために作業するのではなく、先に作業をして副作用としてNOTICEの大半が片付くのを待つ

目標としたもの

パーツ・アッセンブル

2.1 提案モデルは「任天堂Revolution」のスピリッツを基本においてます。

「任天堂Revolution」はそれ自体が次世代ゲーム機そのものではなく、バーチャル・コンソールの実行環境となっており、古いソフトを実行することができます。Revolution自体はこれらの「ゲームが実行状態に入ったときに使われるコンソール」の最上位に位置するものとなってます。

2.1 提案モデルはXOOPSの基本的な構造を引き継ぎながらも、最上位で定義する「パーツ」をサイトオーナーが組み合わせ、サイト稼働状態を構築するという概念をXOOPS Cubeとする、という位置づけをとっています。

各パーツはダイナミックリンクライブラリのイメージでやっています。

下位互換性

Cube本体は下位互換性を持ちません。Cube上で「パーツの組み合わせ」の中に、互換モジュールで定義されている実行環境を組み合わせることで XOOPS 2.0.x との互換性を持ち得るという形にしてます。

これによってコモンプロセス(common.php)を高速化した互換用コントローラの高速版のようなものをユーザーが作り、サイトオーナー間で交換するといったことも可能です。従来ここはコアでなければ立ち入ることができなかった領域でしたが、コア以外の開発者に開放されています。

具体的には、フォーク後に、私たちが確立しようといろいろやっている「ディストリビューション」というポジションの方々が手を出す部分になるのではないかと考えています。

実装ノンタッチ

Cubeは（ありもしない）各パーツの接合部を管理し、ゆるやかな制約を課す立場にあり、基本的に実装に対してノンタッチです。下位互換のファイルツリーの関係上、多くのファイルがありますが、ある実行環境に関して必ずロードが必要なライブラリもCubeのファイルツリーには含まれないという考え方をしたいと思っています。

たとえば互換コントローラが phpMailer を必要とするなら、そのファイルの位置とロードはその管理責任下にあって、Cubeとは無関係です。

実際に私は2.1ブランチで、「これぞXOOPS Cube」となる部分（DBレイヤーの換装など）には一切手を出してません。これはユーザーが増えすぎたXOOPSにおいて、唯一実装を巡って非建設的な議論を行う必要が無く、コアチームの神聖不可侵領域もないため、「じゃあそれぞれで作りましょう」と前向きな方向へもっていくといった点でもメリットがあると考えます。

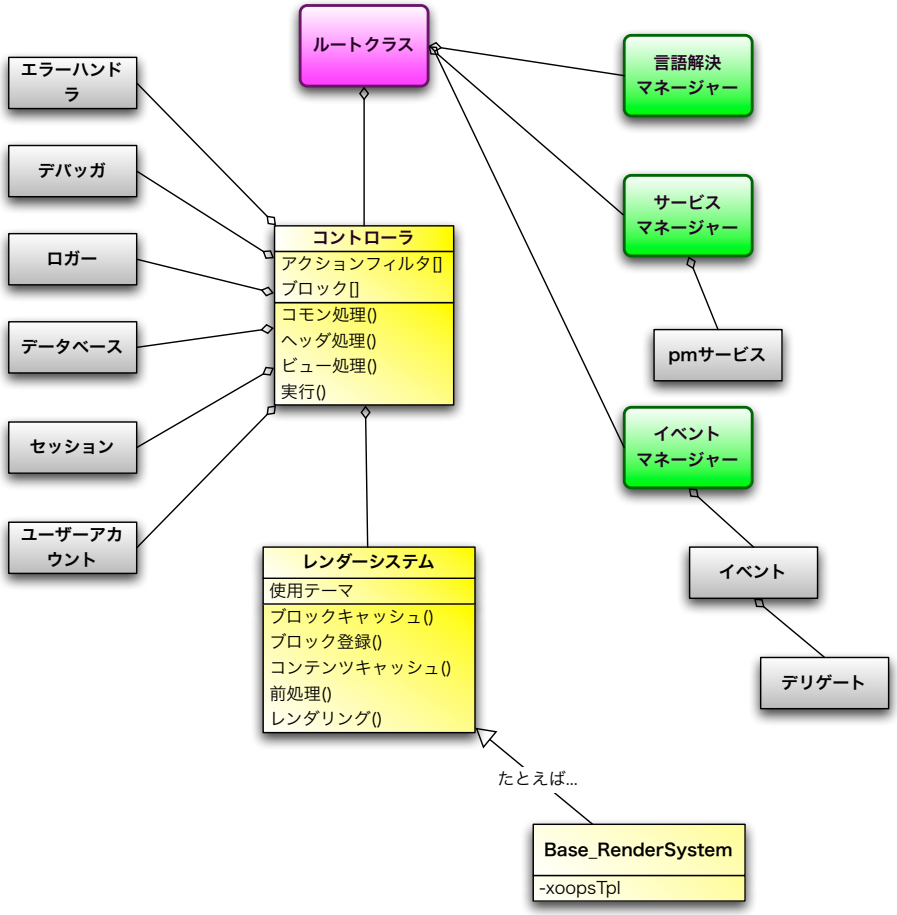
AJAX管理画面とアクセシビリティ管理画面は、同時に存在でき、それは誰が開発しても構わないわけです。

全体構造

仮想コントローラと各システム

試作版2.1の全体の構造を以下に示します。

（この図は古い！　後日、修正します）



- コンセンサスに基づき、2.1では交換性をもっとも重視しました。あるユーザーがXOOPSの一部分を完全拒否したとき、自力でそのサブクラスを開発することにより、機能の交換が可能です。
- 全体の動作は「仮想コントローラ」が行います。仮想コントローラはページコントローラであるXOOPSから部分的に処理をコールされますが、フロントコントローラ制御に交換も可能です。将来的な移行、あるいはカスタマイズに備えています。
- XOOPS Cubeは各パーツの接合部を管理するシステムという位置づけです、それぞれのスーパークラスは事実上の抽象クラスになっています。
- 2.0.xの互換性は継承クラスによって実装されています。パフォーマンスなどXOOPS全体の動作をチューンする際は、ハックではなく、継承クラス（互換パーツ）を作成してセットアップします。

- XOOPS 2.0.x 互換テーマは、XOOPS Cube全体の仕様で互換性を持つのではなく、互換性をもったレンダーシステムが描画することで１００％互換を保ちます。つまり、テーマはCubeの仕様ではなく、各レンダーシステムの仕様となります。
- XOOPS 2.0.x 仕様の中には将来的に廃棄される非推奨仕様が多く含まれます。これらのインスタンスは主にコントローラ内部のプロパティとします。

ページコントローラと仮想コントローラ

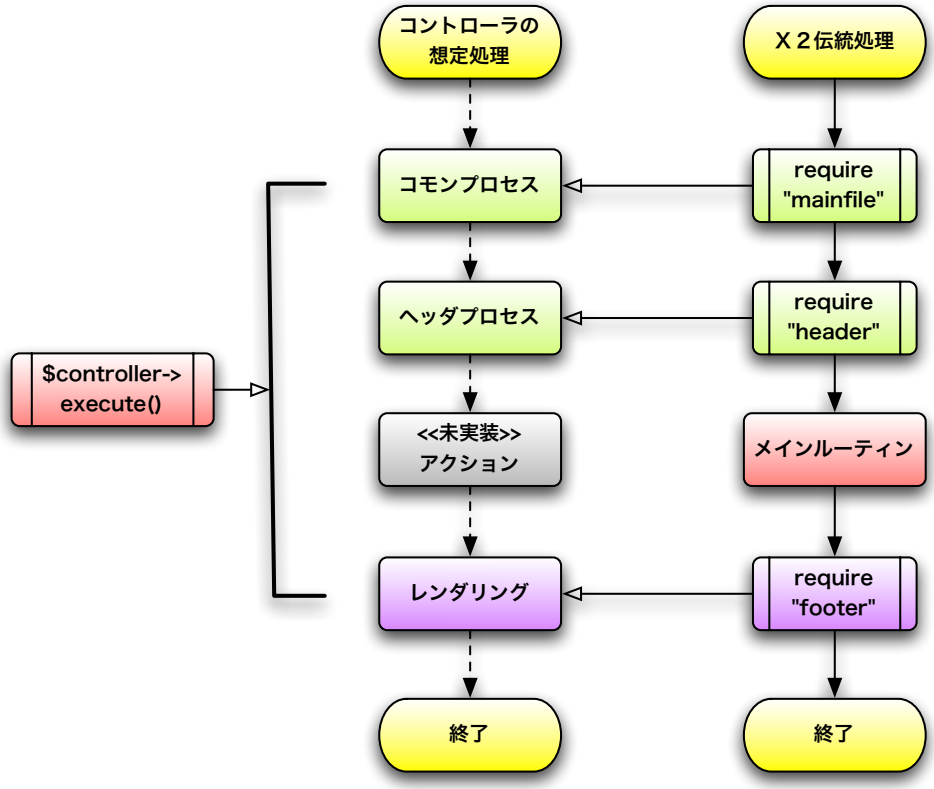
旧XOOPSのモジュールにおける処理は、

1. mainfile.php をインクルード。mainfile.php が common.php を実行して主要な処理が行われる
2. header.php をインクルード。テンプレート初期化、ブロック処理などが行われる
3. モジュールのページコントローラのメインロジックを実行。ob_start()がかかっており、この間の標準出力はテンプレートに取り込まれる。
4. footer.php をインクルード。テーマ読み込み、レンダリング実行。

というプロセスで行われてきました。これをフロントコントローラで置き換えて考えると、3. のプロセスがモデルにあたり、1. 2. は初期化プロセス 4. がレンダリングのプロセスと考えられます。

2.1 で導入した仮想コントローラは、 3. の部分を実装していないフロントコントローラです。そして初期化プロセスと、レンダリングのプロセスは、 common.php,header.php,footer.php などが代弁してコールすることで、旧モジュールの実行時にもコントローラに実装された初期化プロセスとレンダリングプロセスを互換的に実装します。

イメージを以下にしめします。



コントローラの本来の想定は、スタンダードなフロントコントローラ構造です。もし、コントローラに execute() メソッドが実装されていれば、 common.php に相当する executeCommon() 、 header.php に相当する executeHeader() を初期化としてコールし、アクションを処理した後、 executeView() をコールして結果をレンダリングすることでしょう。典型的なフレームワーク的処理です。

しかし、 XOOPS はフロントコントローラではありません。そこで、 2.1 では common.php と header.php 、 footer.php を大幅に変更しています。common.php が呼び出されると、内部で設定されたコントローラを生成してルートオブジェクトに登録し、コントローラの executeCommon() のみをコールします。

次にモジュールの手順に従って header.php がインクルード（実行）されると、シングルトンのルートオブジェクトからコントローラのインスタンスを取得し、 executeHeader() のみをコールします。

モジュールは処理を終えた後、普通 footer.php をコールします。footer.php も同様にコントローラのインスタンスを取得して、executeView() をコールします。

まとめますと、モジュールのページコントローラは、コントローラのアクションの部分を抜いて、頭側とお尻側を従来のXOOPSの必須includeを通じて呼び出すことで、変則的なMVC構造に持ち込んでいることになります。

また、ユーザーは必要に応じて execute メソッドを実装したコントローラを開発し、セッティングすることで、フロントコントローラタイプの XOOPS Cube を動作させることが可能になっています。(実験済み)

コモンプロセス

XOOPS 2.0.x では common.php 内で次のような処理を行っていました。（多少順番が違うかもしれません）

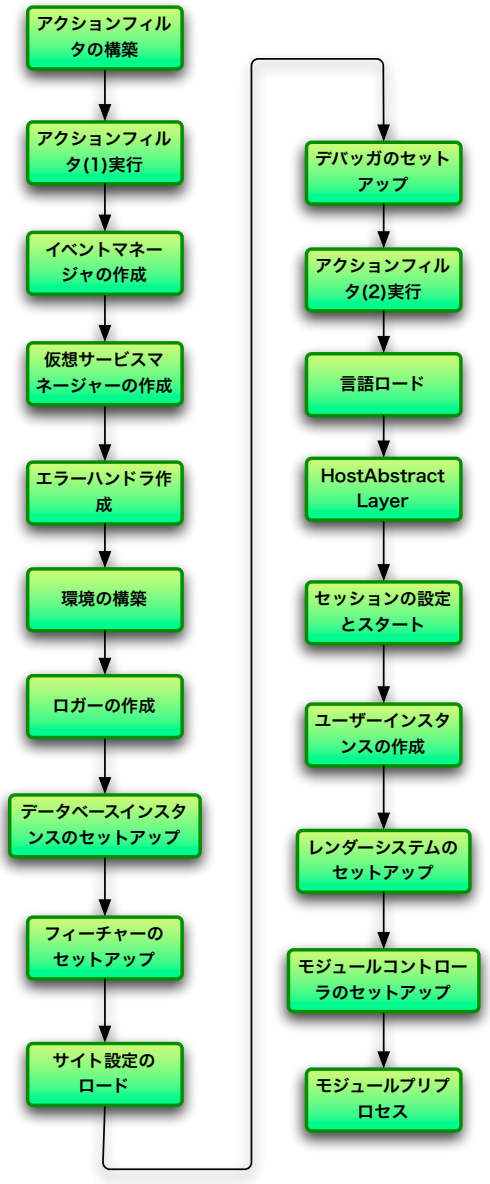
1. 簡易プロテクション処理
2. エラーハンドラの作成
3. 必須ファイル（共通ヘッダー）のインクルード
4. ロガーの作成とスタート
5. データベース接続とデータベースインスタンスの生成
6. サイト全体設定(xoopsConfig)のロード
7. デバッグ処理（PHPエラー設定時にその設定を反映）
8. サイト基本メッセージカタログ（定数ファイル）のロード
9. サイト設定における「拒否ＩＰ」をロードし、ＩＰ拒否を実行する
10. リクエストを解析し、各環境変数を設定（ホストアブストラクトレイヤー）
11. セッションハンドラの登録、セッションスタート
12. セッションよりユーザインスタンスの復元（ユーザーインスタンスの生成）
13. サイトクローズ設定を調べ、ユーザーと比較し、ログイン非対称ならクローズ処理を行い exit()
14. テーマセレクタに関する処理
15. モジュール冒頭処理

common.php には冒頭処理が集中しているため、XOOPS ハックにおいて common.php ハックは高度ながらも必須のテクニックになっていました。 2.1 ではコントローラの processCommon() メソッドがコールされると上記相当処理を行います。ただし、上記相当処理は X2 互換動作をメインテーマとした Base_Controller クラスの仕事であり、基底のコントローラクラスは必ずしも上の処理を行うとは限らないこととします。

2.1 試作版の基底コントローラが processCommon() メソッド内で行っている処理は図のようなものになっています。

processCommon() は、コントローラクラスの各メソッドを呼び出すことで処理を行っています。いわゆる Template pattern であり、自分で自分を組み立てる実装を行っているとも言えます。

2.0.x の common.php ともっとも大きく異なる点は、ブレスト段階で出ていた「common.phpなどにコールバックポイントを用意して、プロテクターなどのモジュールのエントリポイントを作ろう」というアイデアを、アクションフィルタとして実際に実装を試みた点です。



（左の図は古い！　後日修正します）

アクションフィルタは mojavi や Ethna が実装しているアクションフィルタ機能と理屈は同じものです。

ただし、従来「commonハック」だったものを、プラグインやモジュールのように抜き差しができるように……という考え方で組み込んだもののため、自己のセットアップにこの仕組みは使用せず、ユーザーに開放しています。

つまり、ユーザーがアクションフィルタを全部カットしても、動作に支障は出ないというものです。

従来の簡易プロテクション処理、ＩＰ拒否、サイトクローズ、テーマセレクトの４つの機能は、アクションフィルタに移動しています。

後で詳しく説明しますが、アクションフィルタのクラスには２つのメソッドがあり、最初のアクションフィルタ実行では preFilter() メソッドがコールされます。

アクションフィルタ(2)実行では、別のメソッドがコールされますが、このときにはデータベースが使用可能になっているため、ブロックより前のタイミングでいろいろな処理がユーザー側で追加可能です。

オートログインハックのたぐいも、このようなプラグインとして実装できる必要があると思っています。しかし、そのためにはコールバックを発生させるタイミングを少し前後させないといけないかもしれません。

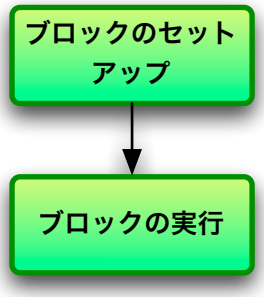
２回目のアクションフィルタ実行が現在のタイミングなのはＩＰ拒否機能を働かせる位置を重視したためです。しかし、この位置での起動はあまりおいしくないかも……

フィルタ同士の優先度の設定はコントローラの実装責任になります。普通、考慮しなくて構わないと思いますが、考慮するコントローラが登場するぶんには一考に構いません……という考え方です。

そのほかのセットアップメソッドでは、インスタンスを生成してコントローラのプロパティか、ルートオブジェクトのプロパティにこれをセットします。ただし XOOPS 2.0.x はグローバル変数に依存しているため、X2互換動作コントローラはセットアップするメソッドの中で、最後に$GLOBALS配列に生成したインスタンスの参照を渡します。

たとえば、データベースインスタンスを生成した後は、$GLOBALS['xoopsDB']にもそのインスタンスをセットします。

ヘッダープロセス



header.php よりコールされる executeHeader() メソッドの処理内容は、テーマやテンプレートに関する処理がレンダーシステムクラスへ委譲されたため、比較的シンプルになっています。

左図のそれぞれの処理もメソッドのコールによって実装されています。現時点ではレンダーシステムは初期化後、コントローラのプロパティにセットします。

2.1試作版のブロックのメカニズムは、アクションフィルタとほぼ同じです。ロジックの書き込まれたクラスのインスタンス（ブロックプロシージャ）を配列に積み、配列をループしながらインスタンスの実行メソッドをコールしていきます。

ただし、アクションフィルタと大きく異なる点として、ブロックはコンテンツ結果を持つコンテンツ出力を目的としたクラスであるという点と、そのコンテンツがキャッシュされるため、キャッシュを管理しているシステムと通信する必要がある点が挙げられます。

そのため、処理の基本的な方針は似ていても、別物として取り扱っています。

もし、コントローラの方針で、ブロックのセットアップを行うメソッドを空に（ブロックプロシージャ配列を構築しない）すれば、ブロックは実行されないことになります。

現時点ではコンテンツキャッシュを管理しているクラスはレンダーシステムのため、ブロック実行時にコントローラが仲介してブロックプロシージャがレンダーシステムに渡されます。レンダーシステムはブロックプロシージャーから必要な情報を受け取ってキャッシュの有無を判定し、必要があればブロックの実行メソッドを呼び出し、レンダーシステムに渡します。

レンダーシステムによってはこの段階で内部バッファにレンダリングを行います。この動作はＸ２互換動作のものです。

ルートクラス

ルートクラスの必要性

ルートクラスのオブジェクトは、XCube_Rootクラスからシングルトンで取得できます。このインスタンスはプロパティにコントローラへの参照を持っており、XCube_Rootクラスのシングルトンを通じて、XOOPSプログラムのあらゆる場面（正確にはcommon.php以降）でグローバル変数を使用せずに各情報を取り出すことが出来ます。

通常のフレームワークでは、必要なクラスに必要な情報（たとえばコントローラへの参照など）が渡されますが、XOOPSは新旧が入り交じるプログラムのため、たとえば旧ブロックメカニズムのコールバック関数内でコントローラのプロパティからロガーを取り出すことは困難です。（ロガーはシングルトンですが）

また換装を推奨しているため、コントローラをシングルトン実装にしても、コントローラの名前空間（クラス名）を特定できません。たとえば Base_Controller を使っている場合は Base_Controller::getSingleton() でインスタンスを得られるかもしれませんが、モジュールより決め打ちすることはできません。

そこで、それ以上継承することを許さないクラスXCube_Rootをシングルトンとし、必要な情報への参照をこのオブジェクトに集めることで、最悪の場合でもルートオブジェクトを取得することで各情報へアクセスできる構造にしました。

common.php における使われ方

```
$root=&XCube_Root::getSingleton();
$xoopsController=&$root->createController( コントローラのあるディレクトリ, コントローラ名 );
$xoopsController->executeCommon();
```

ルートオブジェクトを取得し、クラス名からコントローラのインスタンスを作成し、登録するメソッド createController() をコールします。

ページコントローラのグローバル空間でrequireされるcommon.phpの中で、このような処理を行うことで、xoopsControllerはグローバル変数となりますが、各処理ではこの変数名に依存しないこととします。

（モジュールのページコントローラに対しては適切なグローバル変数を保証します）

コントローラクラスの詳細(前編)

このセクションより、試作版コントローラクラスの詳細について説明します。コントローラは大きく executeCommon() executeHeader() executeView() の３メソッドが呼び出しポイントですので、順番に概要を書いていきたいと思います。

コンストラクタ

コントローラのインスタンス生成はルートオブジェクトのインスタンスメソッドを通じて行います。このときルートオブジェクトのインスタンスがコンストラクタの第一引数に与えられ、コントローラ側でもルートオブジェクトのポインタ（参照）を保持します。

コモンプロセス

最初の概要で説明したとおりの処理順序となっています。

（下のコードは古い！　ソースコードを参照してください）

```
function executeCommon()
{
        $this->_setupFilterChain();
        $this->_processFilter();

        $this->mRoot->setEventManager($this->_createEventManager());
        $this->mRoot->setServiceManager($this->_createServiceManager());

        $this->_setupErrorHandler();

        $this->_setupEnvironment();

        $this->_setupLogger();

        $this->_setupDB();

        $this->mRoot->setFeatureManager($this->_createFeature());

        $this->_setupConfig();

        $this->_setupDebugger();

        $this->_processPreBlockFilter();

        $this->mRoot->setLanguageManager($this->_createLanguageManager());

        $this->_processHostAbstractLayer();

        $this->_setupSession();

        $this->_setupUser();

        $this->_setupModuleController();

        $this->_processModuleController();
}
```

あとでまとめますが、

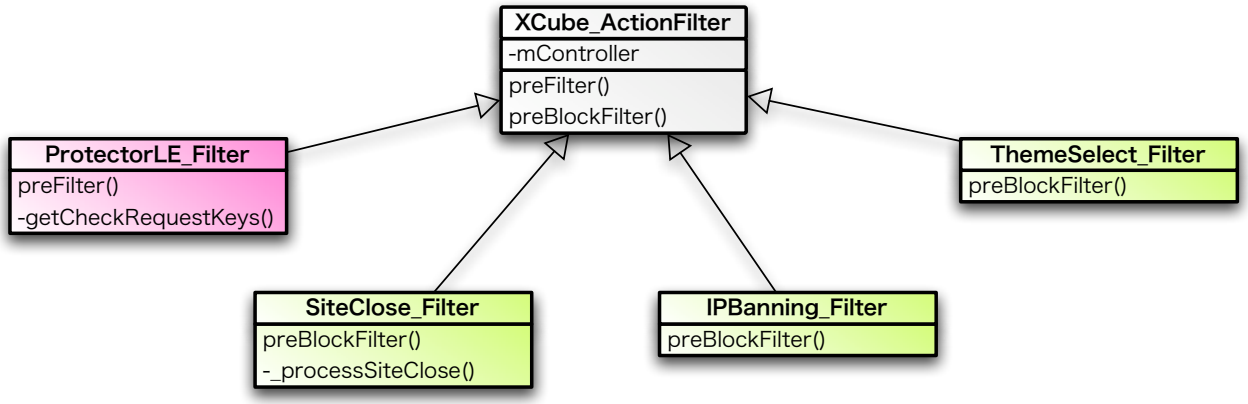
- protected および private なメソッドにはプレフィックスにアンダースコアを用いる
- publicメソッドで実行による副作用を期待するメソッドは execute で開始する
- private / protected メソッドで実行による副作用を期待するメソッド、すなわち「サブルーティン」は process で開始する
- メソッド内で何らかの初期化作業を行い、その結果をそのメソッド内でセットしなければならないメソッドは setup で開始する
- メソッド内で何らかの初期化作業を行い、その結果を return するのみで、その結果がどのようにセットもしくは運営されるかをメソッド側が関知しない場合のメソッドは create で開始する
- プロパティは m で開始する **（例外有り ^^;）**

といった簡単な規則を使っていますので、コードレビューの追いかける参考にしてください。

またPHP4 では instance->getFooInstance()->call() という記述方法ができないため、「C#のスマートプロパティ」と自らに暗示をかけながら instance->mFoo->call() という直接プロパティのインスタンスにアクセスするコードを使っているところが何カ所もあります。ただし二層以上追わないというルールにしています。

- OK ... instance->mFoo->call()
- NG ... instance->mFoo->mNodeFoo->call()

二層以上追う場合は、一度 getFoo() でインスタンスを取得してから処理します。また、実際にはPHPにはスマートプロパティもインデックサもないため、「そこにインスタンスがあるかどうか、ルール上でも保証されない」場合は踏み込まず、プロパティのセットアップの確認をします。



アクションフィルタ (_setupFilterChain(),_processFilter(),_processPreBlockFilter())

アクションフィルタはフレームワークmojaviやEthnaに搭載されているメカニズムであり、古くはアプリケーション初期化TCBとほぼ同等のテクニックです。この機構では、特定のインターフェイス(スーパークラス)を実装したクラスのインスタンスはベクター配列に構築し、実行時にこれらのインスタンスをイテレーションで取り出し、メソッドをコールバックします。

メソッドは preFilter() と preBlockFilter() の二種類があり、preFilterはexecuteCommon() の「ど頭」でコールバックされるメソッド。preBlockFilter() は割と後方でコールバックされるメソッドです。この二つのメソッドの使い分けは上の図がしめすように、「簡易プロテクション」は冒頭の処理が重要なためpreFilter()をオーバーライド。「サイトクローズ」などユーザーインスタンスが構築し終わらなければロジックが実行できないものは preBlockFilter() のほうをオーバーライドします。

アクションフィルタのインスタンス配列は、executeCommon()のコール直後に、コントローラの内部メソッド _setupActionFilter() をコールすることで行っています。

```
function executeCommon()
{
        $this->_setupFilterChain();
        $this->_processFilter();
```

この_setupFilterChain()メソッド内で、どのようにベクター配列を構築するか、という点は、コントローラの責任となります。言わずもがなではありますが、インストールされているフィルタを走査し、有効/無効/優先度を検討しながら配列を構築するものは負荷がかかり、決め打ちになっているものは幾分か高速です。

実際にはPHPの特性を利用して、名前だけを配列に積み、静的メソッドでコールすれば、インスタンス生成がないぶん幾分か高速になるようですが、インスタンスメソッドによる特性も多少失われます。

イベントマネージャーの作成 (_createEventManager())

このメソッドでは、イベントマネージャーのインスタンスを作成し、必要であれば初期化を行って、returnしなければなりません。戻した値は、コントローラの責任において、通常ルートクラスのシングルトンに登録されます。

仮想サービスマネージャーの作成 (_createServiceManager())

このメソッドでは、仮想サービスマネージャーのインスタンスを作成し、必要であれば初期化を行って、returnしなければなりません。戻した値は、コントローラの責任において、通常ルートクラスのシングルトンに登録されます。

エラーハンドラのセットアップ (_setupErrorHandler())

このメソッドではエラーハンドラのインスタンスを生成し、コントローラの mErrorHadnler プロパティにセットしなければなりません。

2.1試作実装では、エラーハンドラは従来のエラーハンドラクラスを使用しています。

エラーハンドラの刷新の可能性を考え、「特定のバージョンに依存した処理はコントローラの責任」ということで、コントローラのプロパティに生成したインスタンスを預けていますが、本来、これはルートオブジェクトのプロパティにセットすべき性質のものかもしれません。

ただし通常のウェブアプリケーション開発文化では、多くのこの手のインスタンスはコントローラのプロパティに預けられているようです。ルートオブジェクトはコントローラが変名する前提条件でシングルトンを引っ張り出すための対策であり、あまりこちらに重要なインスタンスを預けすぎると少しおかしくなってしまうかもしれません。

ただし、コントローラがパラメータで渡されない末端のコード位置から、引き出しやすいのはクラス名固定のシングルトンを持つルートオブジェクトでもあります。なお、ルートオブジェクトは元々マルチスレッドの３Dエンジン制御から拝借したアイデアのため、「ｘｘｘマネージャ」系はルートに預けてあります。（理由がむちゃくちゃですね...）

Xoops 2.0.x との互換性を保持するコントローラ Base_Controller では、プロパティにエラーハンドラのインスタンスをセットしたあと、$GLOBALS[‘xoopsErrorHandler’] に同じ値を代入して、過去のグローバル変数依存コードとの互換性をとっています。

環境セットアップ (_setupEnvironment())

そのコントローラが動作上必要な必要な定数を定義したり、依存しているファイルをロードします。

XOOPS 2.0.x 互換のBase_ControllerであればXOOPS_XXXX系の定数やkernel関係のクラスをロードして、環境を整えます。元common.phpの冒頭に近い場所にあった定数宣言、require関係と等価の処理を行っています。

ここではなく、もっと冒頭で処理するべきかもしれません。特にアクションフィルタをセットアップする_setupActionFilterChain()より後ろでコールされているため、そのことがアクションフィルタ構築に特性を与えることを難しくするため、さらなる検討が必要かと考えます。

ロガーのセットアップ (_setupLogger())

このメソッドは、ロガーを生成し、コントローラのプロパティ mLogger にセットしたあと、ロガーの必要な初期化の実行を実装しなければなりません。

ここでの処理は、エラーハンドラのセットアップとほぼ同じ事情で処理を行っています。処理後、Base_Controolerが$GLOBALS配列を利用して、グローバル変数にロガーをセットする点も同様です。

同様にロガーのインスタンスの預け先がルートオブジェクトではなく、コントローラのプロパティでよいのか、という問題も残りますが、詳しくはエラーハンドラのセットアップで書いた問題提起を参照してください。

データベースインスタンスのセットアップ (_setupDB())

このメソッドは、そのコントローラの思想において、データベースへ接続を行い、データーベースインスタンスをコントローラのプロパティ mDB にセットしなくてはなりません。XCube_Controllerではもちろんこのメソッドは実装されておらず、Base_Controllerでは過去互換のデータベースインスタンスを生成しています。

サイト設定コンフィグ値のセットアップ (_setupConfig())

このメソッドは、コントローラから参照できるサイト全体設定のコンフィグ値をプロパティ mConfig にセットしなければなりません。このような性質のものが必要である、ということがコントローラの特性となると解釈しているため、ルートオブジェクトではなくコントローラのプロパティにセットします。

要するにこれは $xoopsConfig であり Cube の新仕様ではこのような形で使わないかもしれませんし……

XOOPS 2.0.x 互換コントローラである Base_Controller では、ここまでと同様にこの値を$GLOBLAS{'xoopsConfig'] にもコピーして過去互換を保たせています。

デバッガーのセットアップ

（なぜか）base モジュールディレクトリ内で宣言している XoopsDebugger のいずれかのインスタンスを生成し、コントローラのプロパティ mDebugger にセットしなければならないメソッドです。このコードはクリンナップの余地があります。

この「デバッグモード」に関する処理をクラスに委譲した理由は、詳しくはデバッグクラスの説明のところで書きたいと思います。

言語解決マネージャーの作成 (_createLanguageManager())

このメソッドは、言語解決マネージャー（ランゲージマネージャー）のインスタンスを作成して、returnします。返したインスタンスは、呼び出し元の責任で、ルートオブジェクトに登録されます。

言語解決マネージャーは特定のタイミングで送られるメッセージングに従い、多国語対応のメッセージロード処理を行うクラスです。XOOPSでは定数を使用して言語を解決してきましたが、この仕様は2.1試作版の考え方ではXOOPS本体の仕様ではなく、ひとつの言語解決マネージャーの実装です。

たとえば、 po ファイルなどを使う方式に将来的に移行した場合や、ブロック処理の関係上過去方式とブリッジしなければならない場合も、その仕事は言語解決マネージャーの仕事となり、本体仕様から切り離されます。

また、カスタムクラスを作ることで、オリジナルサイトの構築を容易にするかもしれません。ただし、本当に po ファイルやgettextの方式などを実装しようとしたときいまのメッセージング（インターフェイス）で不足がないのかどうかは、かなり不明です。現時点でも、開発が完了しておらず、不十分なのは明らかです。

狙い通りにいけば、現仕様のままであっても、むりやりutf-8仕様に対応したいサイトオーナーがこのクラスと、他いくつかのクラスを組み合わせることでそれを可能にすることができ、同じ希望を持つサイトオーナーが継承クラスでそれを交換し合うことができます。

ただ、この分野において「交換性の恩恵」をきちんと発揮していくには、まだまだブラッシュアップが必要です。現段階では単に委譲しただけです。（しかもそれも終わっていない）

ホストアブストラクトレイヤー (_processHostAbstractLayer())

この処理は、common.phpで PHP_SELF の XSS 問題の際に宿題を残した一連のコードの部分を担当しています。

現時点では宿題を残したまま移植しています。この処理が、仮想コントローラという抽象クラスの中でどのような段階にあたるのか、まだ見極められておらず、XOOPS 2.0.x 互換コントローラである継承クラス Base_Controller の独自メソッドにすることも考えています。

（nobunobuさんの調査を元に全値一致を目指し修正予定）

セッションのセットアップ (_setupSession())

このメソッドでは、最低でも session_start() を実装する必要があります。

コントローラの方針に従ってセッションハンドラを設定する場合は、ここでその処理を行います。Base_Controllerでは、 common.php と同様に、ここでDBセッションハンドラを設定しています。

ユーザーインスタンスのセットアップ (_setupUser())

このメソッドでは、ユーザーインスタンスを生成してコントローラの mUser プロパティにセットする必要があります。その方法はコントローラの責任範囲ですが、互換コントローラは後述する仮想イベント Site.Login を発行して EventArgs を通じて新方式のユーザーインスタンスと旧来のXoopsUserObjectを得ます。

取得後、その値をコントローラのプロパティにセットします。

レンダーシステムのセットアップ (_setupRenderSystem())

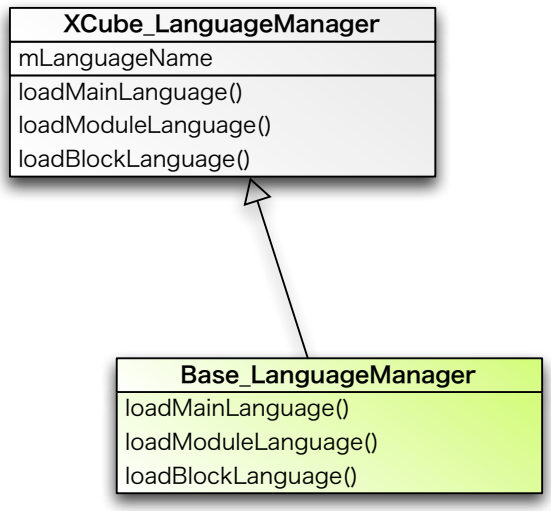
このメソッドでは、描画に使用するレンダーシステムを決定し、インスタンスを生成してコントローラの mRenderSystem プロパティにセットする必要があります。

モジュールコントローラのセットアップと実行 (_setupModuleController(),_processModuleController())

モジュールコントローラは「現在実行のリクエストがモジュールかどうか」「アクセス権はあるか」などの考え方や、実際のチェックをコモンプロセスより委譲されたクラスです。

詳しくはモジュールコントローラクラスの説明をご覧下さい。名称含め、いろいろ議論が必要なクラスだと考えています。

ランゲージマネージャークラス



LanguageManagerクラスは、各ルーティンより「指定された分野の”何らか”のメッセージカタログを”何らか”にロードする」というメソッドのコールを受け、実際にその処理を受け持つクラスです。

このクラスは互換性を考えると明らかにメソッドが不足しています。これからの開発項目です。

言語解決をひとつのクラスとし、構築時に交換できる仕組みにしたメリットのひとつは、後方互換性を保ちつつ、新しいメッセージカタログ方式を検討したり、サイトオーナーが独自に貸すタムシ他言語解決を、ハックという形ではなく、拡張という形で導入できる点だと考えています。

しかし実際には、定数ロード方式をとっているXOOPS 2.0.xと、まだ見ぬかなり将来に検討されるであろう新方式の言語解決方式に共通したスーパークラスを抽出するのは現段階ではかなり不可能であり、まずBase_LanguageManager の実装を終え、どこまでを共通とするか、またそもそもこのアプローチは狙いを達成できるのか？といった点を検討していきたいと思います。

UTF-8対応

定数ロード時に UTF-8 変換をかけて評価することでファイルを EUC にしたまま UTF へ強引に移行させることが可能です。もちろん、それは恒久的な解決方法ではなく、この構造によってハックよりは健全な方法で実装できる解決策のひとつという扱いになればいいと考えています。

（もうひとつの方法としてアクションフィルタをレンダーシステムに採用し、出力前に丸ごと変換してしまうという方法も考えられます）

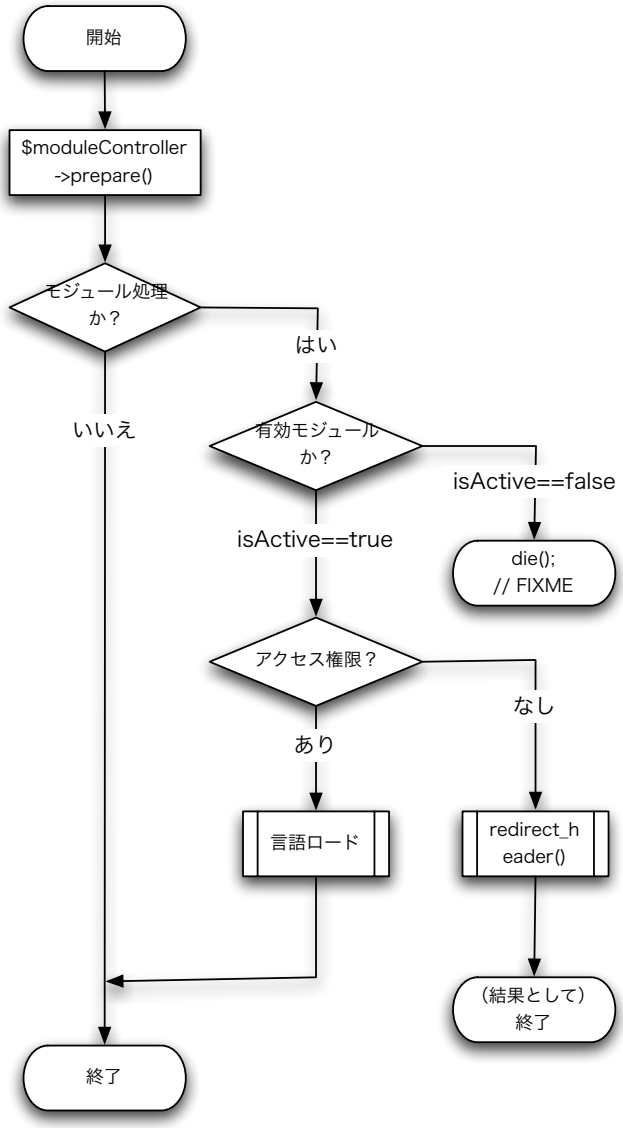
モジュールコントローラクラス

モジュールコントローラ

モジュールコントローラは「モジュール事前処理」の処理を行うために、コントローラと通信するクラスであり、元common.phpの終盤部のコードと相当の機能を持っています。このクラスは、コントローラの処理から委譲を受けたクラスであり、プリプロセッサーと名付けたほうが用途に即しているかもしれません。

2.1の仮想コントローラがカスタムフロントコントローラの実装まで視野に入れているのだとすれば、現在のモジュールコントローラクラスはインターフェイス不足でしょう。

コントローラに内包した方がPHP的ということで、修正するかもしれない。（少なくともCube kernel部でこれをController実装側に強要しないかもしれない）



処理内容

フローチャートは、X2互換処理を前提とした_processModuleController()の現在の処理を図示したものです。

コントローラは現在のリクエストがモジュール処理かどうかをモジュールコントローラに問い合わせます。モジュールコントローラ側では、たとえばリクエストされたファイルパスと同ディレクトリにxoops_version.phpファイルがあれば「モジュール処理」と判断して真を戻すでしょう。

次にモジュールの有効性を確認し、最後にアクセス権限を確認したあと、モジュール用メッセージカタログのロードを行います。これらはすべてモジュールコントローラを通じて行っており、実際にどのような処理になっているのか、コントローラ側では関知しません。

コントローラに直に実装しても構わない性質のものとも言えます。

互換コントローラでの使用例

互換コントローラでは、通常のモジュール実行時には module_read 権限を確認するモジュールコントロー

ラを使用します。しかし、リクエスト位置から現在アクセスが管理画面であると判断した場合はモジュールコントローラを管理画面用のものと変更します。

　管理画面用のモジュールコントローラの判断は、ゲストにはアクセス権限を付与しない、アクセス権限はmodule_admin 権限で判断する、など公開側とロジックが異なります。

コントローラクラスの詳細(中編)

ヘッダープロセス

このセクションでは再びコントローラクラスに戻って executeHeader() メソッドの中身を扱います。

```
XCube_Controoler::executeHeader()
{
	$this->_setupBlock();
	$this->_processBlock();
}

Base_Controller::executeHeader()
{
	parent::executeHeader();
	$this->mRenderSystem->_processStartPage();
}
```

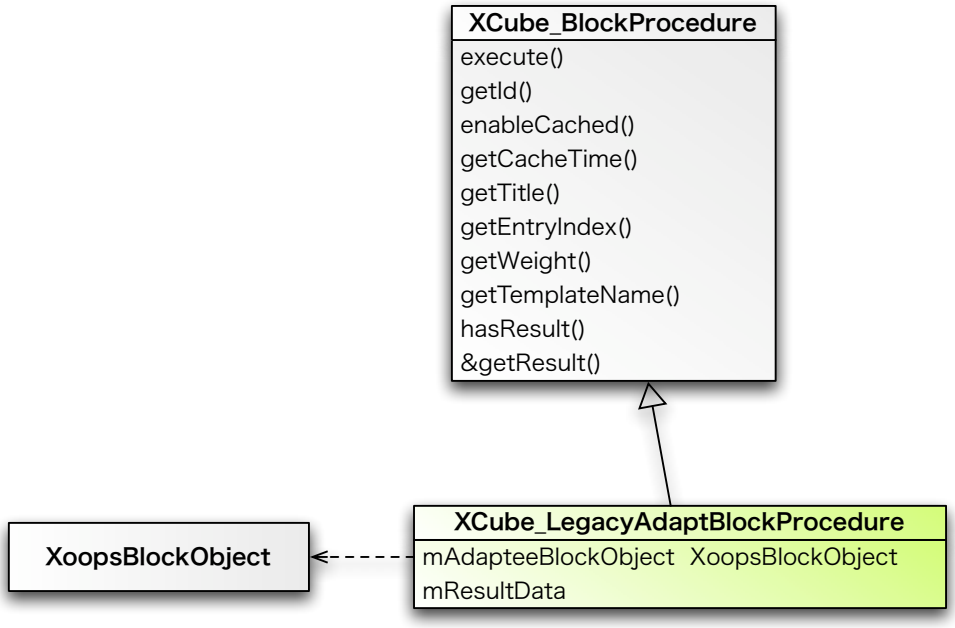
ヘッダープロセスは header.php 相当の処理を行います。header.phpでは、テンプレートを初期化し、権限に応じたブロックをロードして、その処理を行っていました。仮想コントローラクラスにおける executeHeader() も処理的には同等の内容です。

なお、XOOPS 2.0.x では XOOPS1 時代のテーマに対応するために header.php は大きな if で括られていました。

2.1試作版では「テーマの仕様はレンダーシステムの担保」となっています。レンダーシステムは最終的な描画を行うテンプレートエンジンとキャッシュマネジメントシステムで、Base_LegacyRenderSystemが現時点でラッパーでしかないため、XOOPS2.0.xのテーマに対応しています。

XOOPS1のテーマに対応させる場合は、XOOPS1のレンダーエンジンを開発して構築に加える……という考え方になっています。

ブロックのセットアップと実行 (_setupBlock(), _processBlock())



ブロック処理は、アクションフィルタとよく似た処理方式です。アクションフィルタと同様に_setupBlock()メソッドの中で構築を行い、プロパティ mBlockFilter[] をベクター配列とみなして、XCube_BlockProcedureインスタンスを積みます。ブロックとアクションフィルタの違いは、コンテンツがあることと、ブロックキャッシュ、レンダリング方針の決定のためにレンダーシステムに引き渡されて処理を任せる点があることです。

ブロッククラスのサブクラスは、レンダーシステムと適切な情報交換が出来るように各メソッドを実装する必要があります。旧ブロック機構は、アダプターであるXCube_LegacyAdaptBlockProcedureクラスを通じて変換されます。

現在新方式のブロックシステムは実装されていないため[1]、XCube_LegacyAdaptBlockProcedureクラスによる旧ブロックのみが動作しています。新方式のブロックシステムを実装するには、現在のXCube_BlockProcedureクラスの想定は（コンストラクタパラメータ含め）不十分のため、まだまだ検討段階の設計です。

ブロックの実装プロセスは_processBlock()で実装されていますが、本日時点のものはブロックキャッシュを問い合わせていないために、コードが簡素です。いずれにせよ、

1. ブロックキャッシュの管理者（ここではレンダーシステム）にブロックを投げて、ブロックが持つビジネスロジックを実行する必要性を確認
2. ブロックインスタンスのexecute()メソッドをコール
3. ブロックの処理結果がコンテンツ結果を持っているかどうかをhasResult()メソッドで確認

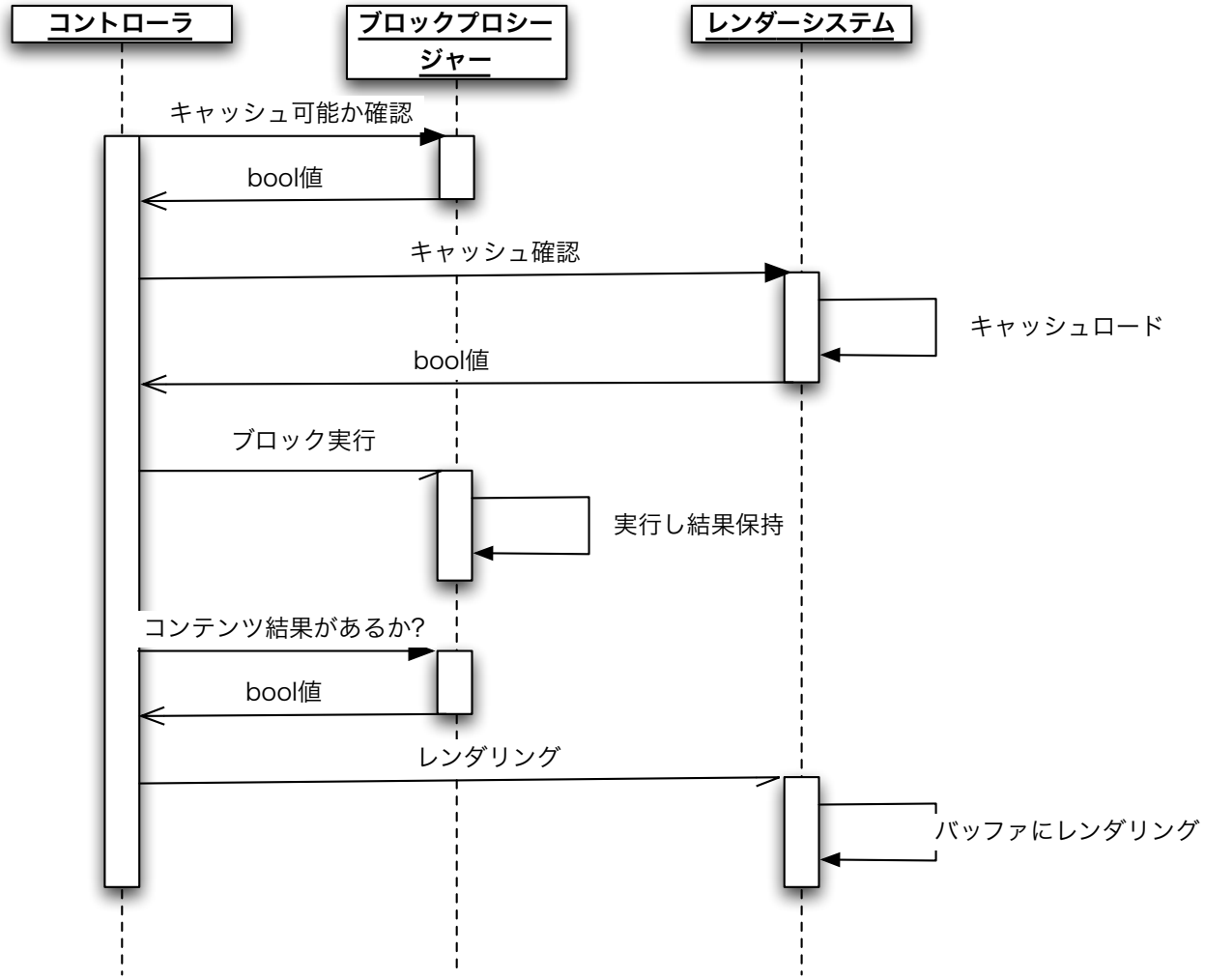
---

[1] Cubeのブロッククラスを直接拡張した唯一の例は、現在管理画面のサイドメニューブロック処理に使われています。

4. コントローラ内で処理をせず、インスタンスごとレンダーシステムのrenderBlock()メソッドに訪問させて処理

といった手順になるでしょう。この段階でレンダリングまで進めるのはX2との互換性のためで、ブロックに関してはトリッキーなテクニックが流行しているため、それらの動作を保証するために処理タイミングを合わせています。本来ならば、executeView()の中で処理したいところではあります。

ブロックキャッシュをふまえたとき、以下のような流れになるかと思います。



これと同様のことが、通常のモジュールのコンテンツに関しても言えます。

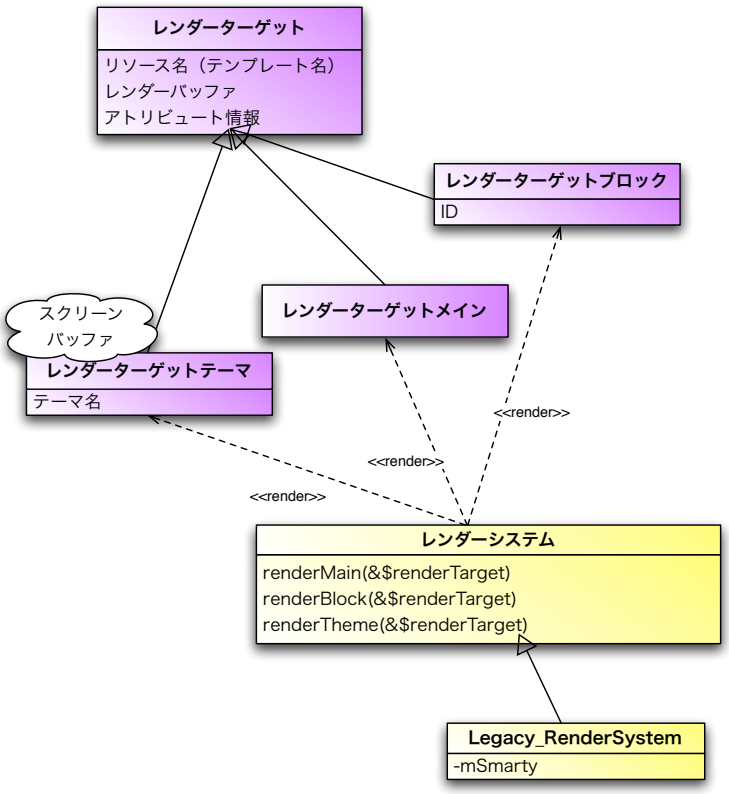
ページスタート前の処理 (_processStartPage())

抽象クラスである XCube_Controller ではなく、XOOPS 2.0.x 互換をテーマとする Base_Controller が executeHeader() の最後に、レンダーシステムのメソッド _processStartPage() をコールします。

ここでは ob_start() を行って、XOOPSの伝統である「header.php～footer.phpの間に標準出力すると」センターブロックのコンテンツとなる、というお手軽な仕様を再現しています。

レンダーシステムクラス

レンダーシステムの概要



レンダーシステムは、リソースの解釈方法と描画方法を知る描画実行システムです（PHPでもCでもよい）。

僕の感覚だとレンダーシステムがしっくり来ているが自己感覚的な命名、**レンダラに改名する**かもしれません。ただしmojaviなどでみられる renderer と異なり、レンダーバッファを分離していて、モジュール開発者は基本的にレンダラに触れない。（そのコントロールはすべてコアに委ねる）

レンダーバッファ

複数のレンダーシステムを持つサイトでは特定のブロックやモジュールは互換レンダーシステムに依存し、新型モジュールは新型レンダーシステムに依存するという状態があり得ます。

Cubeはこの状態を前提とし、レンダラとレンダーバッファを分離した構造を持ちます。

レンダーシステムは、レンダーバッファを渡されて描画を行います（このときの描画対象がレンダーターゲットとなる）。レンダーターゲットから、付属情報を受け取り、リソース情報からリソース位置を解決し、描画を行ってその結果をレンダーターゲットのレンダーバッファに格納します。

レンダーバッファがどのように使われるか、レンダーシステムは関知する必要はありません。レンダーシステムはレンダーバッファに対してのみの描画を実装します。

レンダーターゲットは概念上では、現時点で、通常のレンダーターゲット（メイン）、ブロック、テーマの３種類に分かれており、レンダーシステム側もこの３種類に応対する描画処理を持っていなければなりませんが、これはCubeが最上位で定義する最低限の３種類であり、拡張は自由です。

Legacy仮想互換コントローラは、ページコントローラの処理に入る前に、互換性の関係から一度レンダーシステムのメインレンダーターゲットにLegacy_RenderTargetMainのインスタンスをセットして処理を切り替えます。その後、

このレンダーターゲットへの描画を片付けた後、最終的な処理を行うために Legacy_RenderTargetTheme インスタンスを生成してここまでの処理結果を格納し、レンダーシステムに描画を任せます。

Legacy_RenderTargetThemeのレンダーバッファはメモリではなく、書き込まれた情報を標準出力に流します。

現時点ではレンダーシステムが最終的な描画を行うための結果を格納し続けるコンテナも兼ねていますが、このメカニズムの目的は、ブロックテンプレート、モジュールテンプレート、テーマテンプレートがそれぞれ異なる特定のバージョンのテンプレートエンジンに依存していても、問題なく全体の描画を終えることです。

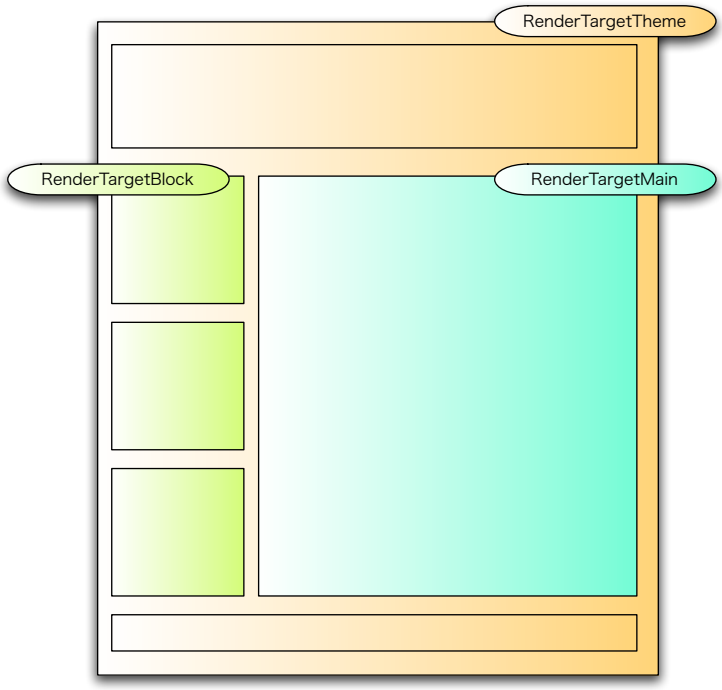
レンダーシステム間の互換性を考えなくても、複数のレンダーシステムを描画に用いて最終出力にまとめることができます。また、特性をつかむことで、メインモジュールをブロックのように制御したり、複数呼び出すことも可能です。

描画担当であるレンダーシステムが、コンテンツキャッシュのマネジメントも行うのは妙で、正直分離したいところです。ここは実際には、レンダーシステムなどと言っても Smarty が中心にあり、現在唯一の実装である Legacy_LegacyRenderSystemクラスが Smarty のラッパーであることが影響しています。

これは修正予定です。

テーマを読み込み、全体に適用するのはレンダーシステムの責任となります。テーマに合わせてレンダーシステムを交換することで、W3C完全対応描画などをこれまでのアサインにとらわれずに行うことができます。

カプセル化の影響



カプセル化という表現は間違いですが、レンダーバッファはそれぞれ独立しており、ブロックからコンテンツ、コンテンツからテーマは見えません。現在は一時的な回避策をとっていますが、コンテンツとテーマのレンダーシステムが異なる状況で正しく動作する仕様を目指す以上、現在の回避策はおかしい。

このことで、モジュール側でテーマのもともとの情報を上書きすることができなくなっており（モジュールのレンダーバッファはテーマのレンダリングに影響しない）、互換性に支障を来すかもしれない。

現在のレンダーシステム

従来と全く同一の仕様(DBテンプレート)を

もち、公開側のレンダリングに使用される Legacy_RenderSystem と、ファイルベースのSmartyを使用し専用のテーマを必要とする管理画面用レンダーシステム Legacy_AdminRenderSystem があります。

Legacy_RenderSystem はそれぞれのレンダーターゲットに対し、

- RenderTargetMain の場合 ... 指定テンプレート名をDBテンプレートとみなして情報取得し、レンダリングして戻す
- RenderTargetBlock の場合 ... 指定テンプレートをDBブロックテンプレートとみなす
- RenderTargetTheme の場合 ... 指定テンプレートをファイルテンプレートとみなし Themes/ 以下で解決を行う

レンダーシステム（テーマエンジン）の必要性

レンダーシステムの基本的な発想は、公式フォーラムでテーマやテンプレートの「唯一仕様」を巡り、建設的とは思えない議論が起こったため、「ウェブデザイン用途においてCubeは唯一仕様を持ってはまずい」と感じたところにあります。

レンダーシステムとここまでの交換許容性の各パーツの大きな違いは、レンダーシステムは最終的には描画テーマの依存性から逆に決定しなければならないという点です。これまでは先にパーツありきでしたが、レンダーシステムの場合はリソースからパーツを決定するタイプかと思います。

このようにレンダーシステムをパーツ化することで、

- 従前スタイルのテーマに対して100%の互換方針を打ち出せる
- 切り替えだけで済むので、次のCube用テーマシステムを探りやすい
- W3Cや完全CSSベースデザインのように現実味がないものも否定せずに並行で進めることが出来る
- 商業関係者には究極的には「御社で好きなレンダーシステムを作ってください」ということができる（そもそも、あまり相手にしたくない……）
- 現在の管理画面の実装のように、ファイルベースSmartyを採用したり、PHPテンプレートのハイブリッドを実装するなど、テンプレートエンジンの存在を隠匿できる。
- モジュール開発者が対応テンプレートを用意していない場合でもレンダーターゲットに対するレンダーシステムの切り替えや、他システム互換のレンダーシステムを用いて柔軟に運営できる（重いでしょうけどね!!^^;;）

といった多くのメリットを得ることが出来ると考えています。名称、設計などいろいろあると思いますが、ぜひテーマエンジンというユニット単位を交換性の対象とすることを検討していただければと思います。

HTMLハードコードを委譲する

XOOPSでは、内部にHTMLレンダリング処理を持つ XoopsForm クラス群や、 xoops_confirm() などのダイアログ的なAPIが HTML をハードコーディングしています。これらの関数を残しておいては、表示系すべてをそのレンダーシステムの思想（W3Cなど）に基づいて変更するという考え方が通りません。

そこで現在は、まだ作業を進めていませんが、これらの関数は残すものの、実際の処理の内容を

1. ロジック内でレンダーターゲットを生成し、レンダーシステムにレンダリングを依頼
2. レンダーシステムがSmarty等からレンダリング
3. ロジック内でレンダーバッファから結果を取り出して return ないしは標準出力へ out

に変更することで委譲を進めたいと思います。

現在の委譲の処理方法は変なので修正します。

標準モジュール開発における補助仕様

標準モジュールについて

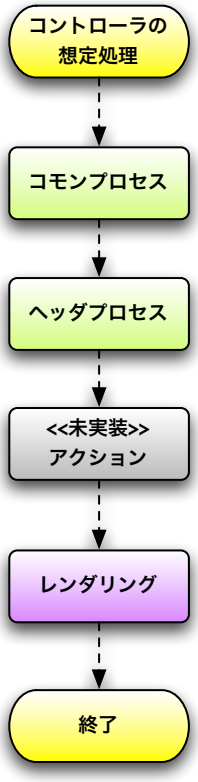
XOOPS 2.1 ブランチでは system を除くすべてのモジュールを削除しました。しかし user モジュール、 pm モジュールをモジュールとした関係上、これらが新しい標準モジュールとなってしまいました。

現在 PHP もフレームワーク開発の世界になっており、フレームワーク開発に慣れきっていると、通常のページコントローラの開発がきつくて仕方がありません。ただし、XOOPSがフレームワークを装備し、それをモジュールに強制するやり方はXOOPS的ではありません（僕たちもその仕様をフィックスするのにどれくらいの議論を費やすのか想像もつかない）。

しかし開発速度のこともあり、こちらの独断で「シンプルなアクションフォーム」と「仮想コントローラの実装されていない_processActionにストラテジを送り込む機構」のみを実装しました。

これらは各モジュールが独自に実装したテンプレートで、フレームワークではありません。フレームワーク的なものはコアは持っていません。これは痛し痒しですが、XOOPSにEthnaを移植する動きがあるように、外部からフレームワークを追加することができるため、2.xベースのうちはXOOPSがフレームワークを持つべきではないと判断しました。

コントローラにアクション処理を実装



前の方で説明しましたとおり、現在のXCube_Controllerは_processAction()における処理がまったく抜けています。フロントコントローラなどの典型的MVC処理を行う際のMに関する配慮がポッカリ抜けているわけで、それを活かしたいマニアなサイトオーナーや開発者がサブクラスでそれを実装して使う、という考え方です。

実際には未実装となっている _processAction() は、プロパティ mActionStrategy にインスタンスがある場合に限り、自分自身のポインタを預けて処理を完全に切り替える形になっています。

つまりサブクラスで _processAction() を埋めるほかに、その処理を受け持つクラスを定義して、インスタンスを登録することでもMのプロセスを実行することが出来るというものです。

edituser.php 内でこの仕組みを使って「楽をしている」コードを以下に示します。

```
$moduleRunner=new UserModuleFrame(new UserModule_EditUserAction());
$root=&XCube_Root::getSingleton();
$controller=&$root->getController();
$controller->setActionStrategy($moduleRunner);
$controller->executeAction();
```

実際にはコントローラがアクションを実装している場合に、それを上書きしないようにロッ

クする仕組みも必要かと思います。

アクションフォーム

こちらも、開発作業の手間の問題から、アクションフォーム XCube_ActionForm を作成しました。これは現在 Cube パッケージの扱いになっているが、 Legacy パッケージへ移動させることが妥当であると考える。

- プロパティとバリデーションの設定をXMLファイルに記述し、makeActionFormツールを使用してPHPクラスの雛形を作ることができます。
- XMLファイルの書式は struts ライクです（現在はさくっと作った状態）。他に ini に対応させようかと思っています。
- magic_quote の設定値を見てリクエストから値をフェッチし、必要に応じて strip をかけ、プログラマに生値が渡るようにします。
- 整数値型、float型、文字列型、複数行文字列（テキスト）型とそれぞれの配列型を指定できます。文字列型、テキスト型はコントロールコードを検出すると攻撃検出と見なして処理を中断します。
- まだバリデーションパターンは作成途中ですが requireif 以外は実装する予定です。

といった機能があります。しかし最低限の機能しかなく、コアシステムからモジュール開発者への「実装脅迫感」をなくすような作りにしたつもりです。

アクションフォーム生成に使用した XML ファイルの原型は、 2.1 ブランチの /etc/actinform/ に入っています。

アクションフォームにはフェッチと検証の程度の操作しかついていませんが、使用側でのアクションフォームの基本的な考え方はexFrame等と同じで、入力時に一旦アクションフォームに値をロードし、検査に合格するまでこれを保持し、保存前にマスターデータオブジェクトをvisitさせてデータを入力させます。

```
$actionForm->update($user);
$userHandler->insert($user);
```

入力画面にはアクションフォームをそのまま渡します。

```
$xoopsTpl->assign("actionForm",$actionForm);
```

従来は XoopsObject::getVar('key','e') が適切なサニタイズをかけていたため、<input>タグの中に入力しても安全でした。この処理プロセスではテンプレートに、生値を保持するアクションフォームを渡しており、プログラマがビジネスロジック内で対テンプレート処理を行わないという考え方になっています。

Smarty plugin の拡張

2.1 ブランチでは、生値をテンプレート上で安全に埋め込むための簡易プレースホルダとなるプラグインを勝手に追加させていただきました。;;**（レンダーシステムの項で書いたようにこれは位置がおかしい。レンダーシステムの拡張であるから、Smartyの拡張をもってそれにあてるべきではない）**

また、従来 XoopsForm を使用しなければ埋め込むことが出来なかった DHtmlTextArea を埋め込める Smarty プラグインや、 TikiWiki のようにテンプレート上で formatTimestamp をかけられる（ビジネスロジック側でUNIXTIMEから変更しておく必要がない）プラグインや、uidとキーを組み合わせて必要な情報を表示するプラグインを追加し、表示のためにビジネスロジック側で煩雑な処理を行ってミスを犯す危険性を少し軽減しています。

簡易プレースホルダは次のように扱います。

```
<{xoops_input class=hogehoge id=name type=text name=name value=$actionForm->getVar('name')}>
<{xoops_textarea name=message rows=5 cols=60 value=$actionForm->getVar('name')}>
```

class, id などは指定しなくても構いません。

XoopsObject クラスにメソッドを追加

アクションフォームとXoopsObjectが加工のない値をやりとりできるよう getProperty メソッドを追加しました。これは getVar('name','e') ではサニタイズ済みの値が戻ってくるためです。

XoopsFormと組み合わせる場合にベストな組み合わせですが、今回の方法ではテンプレートまで生の値を持って行き、ActionForm はロード値と入力値をまったく同じ感覚で扱えるようにしようと考えたため、「神聖なる」基底クラスにメソッドを追加しました。

しかしこれでXoopsForm->getVar('key','e')はサニタイズ値、ActionForm->getVar('key')は生値と、テンプレートでの扱いが難しくなったので、生値をやりとりするためのメソッド名は統一する予定です。

イベントと仮想デリゲート

概要

試作版2.1は自由度の高い構成カスタマイズの上でサイトが稼働します。この仕組みではコア機能にハードコードされた機能が外部へ委譲されているため、「ユーザーがオンラインになった」「ユーザーが削除された」といった「出来事」が発生したとき、それに対する処理を持っているプラグインやモジュールにコールバックが発生する仕組みがあることが好ましいと考えます。

イベントとデリゲートは、関数コールバックの仕組みであり、一部で発生した出来事の顛末をモジュール側で処理したり、従来のノティフィ機能のように使うなど、様々な応用が考えられます。

マルチキャストであり、通常の機能をコールする前にフックを確認するタイプの機能フックではなく、本来の機能もデリゲートで実装したうえで、イベントに登録されたすべてのデリゲートがコールバックされます。

デリゲート自体もマルチキャストにする予定ですが、感覚的にひとつずれたのが痛い。現在イベントマネージャを名乗っているクラスはデリゲートマネージャーであり、マネージャーにはイベントハンドラのクラスが追加されるべきだと思います。ただしデリゲートは象徴的な名称であり、これをもって各コールバックメソッド/関数を取り扱えるのは感覚的にはよいという解釈もあるかもしれない。（僕個人は、もうずれてしまったと考えています）

仮想デリゲート

デリゲートは、コールバックの場所を扱うためのクラスです。PHPには関数ポインタがないため、コールバックに登録するための統一的な手続き方法がありません。デリゲートクラスは、

- 静的メソッド（クラスメソッド）
- インスタンスメソッド（オブジェクトメソッド）
- 静的グローバル関数

の三種類のコールバックを抽象化して、統一的に扱うためのクラスです。

XCube_Deligate クラスが「静的メソッド」と「静的グローバル関数」を、XCube_InstanceDeligateクラスが「インスタンスメソッド」を表すことができます。（PHPは引数の違いによるオーバーロード宣言ができないため、コンストラクタのシグネイチャが異なるものは分けました）

- $delegate=new XCube_Delegate(“login”); / / login() のコールバックをしめす
- $delegate=new XCube_Delegate(“User”,”login”); / / User::login() のコールバックをしめす
- $delegate=new XCube_InstanceDelegate($userManager,”login”); / / $userManager->login()をしめす

デリゲートはそれだけでは単にコールバック先の情報を示すだけですので、意味を持ちません。

なおコールバックを受ける関数・メソッドは以下のシグネイチャを持つ必要があります

function Foo(&$sender,&$eventArgs);

イベント

イベントクラスは、イベントをあらわすクラスです。

イベントクラスは複数のデリゲートをコールバック情報として持つ（マルチキャスト）ことができ、raiseEvent()メソッドがコールされると、自分に登録されているすべてのデリゲートの情報を元にコールバックをかけます。

イベントクラスとデリゲートクラスを組み合わせると、「Hello」イベントによって、「Hello, World」が標準出力されるコードは次のようになります。

```
class Hello {
        function hello_event(&$sender,&$eventArgs) {
                print “Hello,World”;
        }
}

class Test {
        var $mEvent;
        function Test() {
                $this->mEvent=new XCube_Event(“hello”);
                $deligate=new XCube_Delegate(“Hello”,”hello_event”);
                $this->mEvent->add($delegate);
        }
        function main() {
                $args=array();
                $this->mEvent->raiseEvent($this,$args);
        }
}

// メインルーティン
$test=new Test();
$test->main();
```

イベントと仮想デリゲートの関係は以下のようになっています。

- ひとつのイベントは複数の仮想デリゲートを持つことが出来る
- イベントクラスは、高速化のために配列をデリゲート情報として扱うこともできる
- （現時点では）デリゲートはひとつのコールバックしか登録できない。デリゲートの加算はできない。

イベントマネージャ

イベントマネージャは複数のイベントクラスをプロパティに持ち、その作成とイベント発動を代行するクラスです。

イベントとデリゲートはモジュール内で記事が承認されたときの処理などを書くうえでも使用できますが、基本的にはサイト上で発生したイベントの処理をプラグインやモジュールに任せるための仕組みです。そのためのイベントマネージャがコントローラのセットアップ時に作成され、ルートオブジェクトのプロパティに格納されます。このマネージャがグローバルなイベントを管理するイベントマネージャになります。

イベントマネージャを使うとき、呼び出し側はイベントクラスの存在を考えなくても大丈夫です。イベントマネージャは文字列で指定されたイベントインスタンスを持っていなければ、自動的に作成して自分の管理に加えます。

```
$eventManager = new XCube_EventManager();
$eventManager->add("Site.Login",$delegate);                    // Site.Login イベントがなければ作成される
$eventManager->raiseError("Site.Login",$this,$eventArgs);                    // 呼び出し代行
```

情報の受け渡し

XCube_EventManager::raiseEvent() の第三引数、および XCube_Event::raiseEvent()の第二引数が情報としてデリゲートで登録したコールバック関数へ渡されます。そのため、先ほども書きましたとおり、コールバック関数は次のようなシグネイチャを持つ必要があります。

```
function login(&$sender,&$eventArgs){ ... }
```

デリゲートはイベントからのコールバックにおいてマルチキャストであるため、基本的に戻り値を持たず、対象に対して直接操作を行います。しかし$eventArgsが参照パラメータであることを利用して、値を戻すことができます。

イベントリプレス(未実装)

イベントマネージャーはデリゲートの登録を養成されたイベントが自分の管理下にない場合、自動的に XCube_Event オブジェクトを生成します。しかし、その後に正規の処理に入ったコントローラは、そのイベントに対して特別なイベントオブジェクトをマッピングしたいかもしれません。

2.1試作版の仕組みではコモンプロセスの順序が保証されにくいため、イベントマネージャーに対してイベントオブジェクトの交換を申請することができます。このメソッドでは、すでに XCube_Event オブジェクトがマッピングされていた場合、パラメータで渡されたイベントオブジェクトに交換し、さらにそこまでの処理で交換元のイベントオブジェクトに登録された全デリゲートを交換先につけかえます。

アンカーデリゲート

Site.Login イベントなど、デリゲートのコールバックに入る前や後にコントローラが特別な処理を行ったり、デリゲートのシグネイチャを変える場合は、イベントクラスを定義してそのオブジェクトをイベントマネージャーに登録する方法が比較的正当ですが、処理の内容と定義の負荷のバランス次第では、アンカーデリゲートが代替になることがあります。

アンカーデリゲートはシングルキャストのデリゲートで、通常のデリゲートがすべて呼び出された後にコールされます。

アンカーデリゲートをイベントにセットするには setAnchorDelegate() メソッドを使用します。

イベント関連の高速化

デリゲートを配列で記述する

イベントクラスは、add()メソッドにおいてデリゲート（コールバック情報）をデリゲートクラスのオブジェクトだけでなく、配列からも受け付けています。

```
$delegate['class'] = インスタンスかクラス名;
$delegate['function'] = 関数名;
```

イベントクラスは raiseEvent() でコールバックを実行する際、配列のデリゲート情報があると class キーと function キーに登録されている内容の型を調べて、静的メソッド、インスタンスメソッド、グローバル関数のいずれかを判断してコールバックを実行します。

CLASSの値	FUNCTIONの値	判定結果
文字列	文字列	静的メソッド
インスタンス	文字列	インスタンスメソッド
なし	文字列	静的グローバル関数

インスタンスを生成しないぶん、若干高速です。

デリゲート使用時生成

イベントには、リクエスト毎に実行されるイベントもあれば「ユーザー削除イベント」のように、滅多に実行されないイベントもあります。そのために、デリゲートを生成してイベントマネージャーに登録していては負荷がかかります。

イベントマネージャーには XCube_EventProxyRegister というクラスを用意しています。このクラスは、イベントが実際に発生した時に必要なデリゲートのみを生成して、その場にイベントに追加するメカニズムです。イベントマネージャーがこのメカニズムを組み込んでいるため、通常のデリゲート登録と並行して使うことができるほか、デリゲート単位でプロキシを生成する必要がない点がメリットです。

プロキシーレジスターを使うには、XCube_EventProxyRegisterを継承します。

```
class UserEventProxyRegister extends XCube_EventProxyRegister
{
	var $_mList=array("USER_CheckLogin");
	function getEventNameList()
	{
		return $this->_mList;
	}
}
```

getEventNameList メソッドでは、このプロキシーレジスターが遅延登録可能なイベント名を配列で戻すように実装します。実際に指定したイベントが実行されると、イベントマネージャーはプロキシーレジスターの &createDelegate()をコールして、このイベントに関わる仮想デリゲートの return を要求します。

XCube_EventProxyRegister クラスの createDeligate()はメソッド名合成による自クラスのコールバックをすでに実装していますので、それを利用して実装するのが今のところ手っ取り早いです。

```
class UserEventProxyRegister extends XCube_EventProxyRegister
{
	function &createUSER_CheckLoginDelegate()
	{
		// ....
		return $deligate;
	}
}
```

まとめてイベントを定義する場合に役立ちます。テストの範囲内では正常に動作しており、同じイベントを直接デリゲートで登録しても、この代行登録を利用しても、実際にイベントが発生したときはデリゲートの条件をそろえてマルチキャストでコールバックします。

特別なイベント

ユーザー認証など、コントローラがコモンプロセスの中でインスタンスを得る必要があるイベントは特別なパラメータが使われるため、ルールに従ってデリゲートを書きます。

ログイン／ログアウトいずれもデリゲートクラスを通じて登録するため、クラス名・関数名・ファイル配置等に制限はありません。

イベントの仕組みを使っており、複数のデリゲートが呼ばれます。オートログインハックをログインイベントに登録するなどの使い方をします。

Site.Login イベント

イベント送信者	XCube_Controller *
イベント要素	LoginEventArgs *

Login ボタンを押した時のイベントではなく、従来 common.php で処理されていた、$_SESSION['xoopsUserId']よりxoopsUserインスタンスを得ていたアレです。

このイベントでは、専用のEventArgsクラスをパラメータとして受け取ります。

現在のログイン状況をチェックし、XoopsUserインスタンスを作成して EventArgs にチェックします。また新方式のインスタンスもセットできます（現時点では義務ではない）。

自分より前に実行されたデリゲートで認証が終わってないか、eventArgsを通じてチェックしておく必要があります。

```
if($eventArgs->hasXoopsUser()) { ... }
```

Site.CheckLogin イベント

イベント送信者	XCube_Controller *
イベント要素	CheckLoginEventArgs *

Site.Login.Click のどっちにしようか一瞬悩んだのですが……一般的には、ユーザーがログインボタンを押した時に発行されるイベントです。パラメータに専用の CheckLoginEventArgs が渡されます。

自分より以前のデリゲートでチェックを通過している可能性があるため、eventArgs->isSuccess()で成否を確認し、そのうえで実行・中断・無視など適切な処理を行います。

可能であればXoopsUserObject等を作成してeventArgsにセットしますが、セットできなくてもフェータルではない処理が行われるものとします。

Site.CheckLogn.Success

イベント送信者	XCube_Controller *
イベント要素	XoopsUserObject $args[‘xoopsUser’] 新方式ユーザー $args[‘user’]

チェックログインイベントに成功した後で、そのイベントからさらに送信されるイベントです。最終ログイン時刻の変更などのために使われます。たいていのイベントは、このイベントのように高速化のためにEventArgsのための型を持たず、連想配列を使用します。

Site.Logout

イベント送信者	XCube_Controller *
イベント要素	bool $args[‘successFlag’]

ログアウトがクリックされたときに送信される（送信しなければならない）イベントです。自分より以前に処理されたデリゲートの結果はarray[‘successFlag’]に格納されているので、これをチェックして方針を決めます。

ログアウト処理に成功した場合はarray[‘successFlag’]にtrueをセットします。

ログアウト後のは、イベントもしくはコントローラの処理方針に任せて構わないので、成否だけをeventArgsを通じて戻すようにします。

ルート配置の旧ページコントローラ

Cube2.1の正式なパッケージには、index.php以外のコントローラは含まれません（理屈上は;;）。ただしＵＲＬ互換性の関係で多くのコントローラが残されています。

これらのコントローラにアクセスがあると、Legacypage.ページ名.Access イベントを発行して処理を終了します。

つまり、エントリポイントとしての役割しか果たしません。

- Legacypage.Userinfo.Access
- Legacypage.User.Access ... 原作が if switch 型なので内部でイベント発行
- Legacypage.Viewpmsg.Access
- Legacypage.Readpmsg.Access ... これは外部モジュールでアクセスしている可能性あるんだろうか！？
- Legacypage.Register.Access

イベントはマルチキャストが大原則ですが、現在は header location によるリダイレクトで処理されているため、この処理を行うデリゲートがコールされた時点で他のデリゲートはコールされなくなります。

競合(?)を柔軟に解決するために、専用のイベントクラスを定義するかも知れません。

仮想サービス

ロードマップに掲げた換装可計画においては、ユーザーモジュールやＰＭモジュールの入れ替えが容易です。これらのモジュールの持つ機能の中には、他のモジュールのフロントエンドから使われるものがあります。

サイトにどのようなフィーチャーが交換もしくは新規追加され、それがどのようなインターフェイスを持つかは、多のモジュールやコアからは分かりません（分かるようにする＝縛ることも当然できますが）。コール元からは、

- サイト構築状況から、自分がやりたいことを実現できる機能を検索する
- インターフェイス、機能を調べる（ことができる）
- その機能にリクエストを送り、副作用 or 結果を得る

ことができなくてはいけないということになります。当初これをフィーチャーマネージャーという仕組みで実装していましたが、このメカニズムはウェブサービスに似てないかと考え、Shadeでテスト後、仮想サービスクラスを作成して追加しました。

これらの仮想サービスクラス XCube_Service および XCube_ServiceClient は、この一連の手続きをnuSOAP、PEAR::Soapインターフェイスに合わせるための簡単なクラスです。

- 制限バリバリにするか、トリッキーな実装かどちらかになってしまう上記のテーマをSOAP感覚にすることで誤魔化してしまう
- しかし本気でSOAPアダプターを書いて、同様の仕組みで他のウェブサービスに接続したり、仮想サービスをマジウェブサービスにする可能性が匂わなくもない

という後ろ向きな理由と前向きな理由が同居しています。

メリットとデメリット

この形態を取ることで、

- アマゾン書評モジュールが提供するＡＰＩサービスから取得済みアマゾンデータを借り受ける
- コンテンツコンテナモジュールから文章だけを受け取るモジュールをデュプリケータブルにする
- What's new、Waiting情報をサービスとして送信する

など、数年来のモジュールＡＰＩ問題を市民権のある方法で解決できるかも知れません。

問題は速度ですが……イベントマネージャーと同様、サービスを管理・統括するマネージャーに遅延生成の仕組みを入れることで、なんとかフォローできないかと考えます。実際にはサービスは検索の必要があるので、発見時にサービスをあらわすインスタンスを生成できればいいと考えています。

サービス側も当然、WSDLなどを書く必要はなく、基底クラスを継承して必要なメソッドを拡張するだけです。

イベント・仮想デリゲートと仮想サービスの区別

モジュール間の、特定の機能の連携ということではよく似ていますが、イベントはマルチキャスト（複数のデリゲートを登録）できるため、「ユーザーがログインした」ときにメールを送ったり、オンライン情報を更新したりといった複数のロジックをコールすることができます。

また原則的に戻り値を持つべきではありません。

仮想サービスは、あるアクションを行うときに、ひとつのサービスを探し、それをハンドラにする考え方ですので、守備範囲はかなり異なるかと思います。

いずれもルートオブジェクトのマネージャーに属するものが、グローバルイベント＆グローバルサービスになります。

サンプル...

/module/pm/service/LegacyPmService.class.php に定義してあります。サービスマネージャはまだありません。

```
$service=new LegacyPmService();  // ホントはここでサービス検索
$client = new XCube_ServiceClient($service);
$parameters=array("uid"=>$xoopsUser->getVar('uid'));
$pmObjects=&$client->call("getList",$parameters);
if($client->getError()) {...}
```

ＰＭの実装でサービスを使うか否か？

自分でも一瞬迷ったのですが、自分の中での整理をここにメモとして残します。

まず、ＰＭモジュールは、標準版、代替版にかかわらず、ＰＭサービスを提供しなければいけません。これはXoopsMailerがこれを使用してPMを送る必要があるからです。

で、ＰＭ送信機能のフロントエンドpmlite.phpなどは、自らのＰＭサービスを使うべきなんでしょうか？

以下のように考えて整理しました……↓

- もし送信機能がフロントエンド部のみを提供して、バックエンドのＰＭサービスを切り替えることが可能な仕組みにするのであれば、pmlite.phpはサービスを使うべきです。
- しかし実際に「advancedPMモジュールでは添付ファイルが扱えます」とかいうことになると、入力インターフェイスからしてpmlite.phpのものは使えません。

  インターフェイスまで受信するとなると「仮想サービス」の定義が変わってしまいます。

この場合 pmlite.php はそのＰＭモジュールが「"知っている"ＰＭの処理方法」に依存して処理してしまって構わないと考えますが、どうでしょうか？

（他方、ユーザーメニューブロックから「カレントユーザーの未読のPM数」を調べるにはPM仮想サービスを使用します）

テキストモディファイアを巡る検討

BBCODE

現在X2には、非常に使いにくいエンコード・デコード一体型のテキストサニタイザクラスが用意されています。このクラスは決め打ちになっているため、これをWikiエンコードに変更したり、拡張したりということが難しい。また当然HTMLコンバータとサニタイザは分離されるべきです。

これに関しては特定の変換処理を行うクラス（変換フィルタ的？）を数珠繋ぎにしてモディファイアを名乗らせ、コアもしくはレンダーターゲットに登録して、プログラマブルシェーダのエフェクト・フレームワークのようにしてテキストを変換する案を検討し実装しました。（そのため、これは一部がCVSに残っている）

しかし大がかりすぎること、設定制御が難しいことから、引き続き各自の検討項目になっています。