Tutorial de Metastorage

Desenvolvimento de aplicações com banco de dados em PHP usando a ferramenta Metastorage

http://www.manuellemos.net/

Manuel Lemos

mlemos@acm.org

Novembro de 2005

Resumo

- Introdução
- Instalação da ferramenta
- Desenho de modelos de dados
- Como usar o código gerado
- Referências →

Introdução ao Metastorage

- Aceleração do desenvolvimento
- Gera código repetitivo que obdece a padrões
- Armazena e recupera objetos persistentes
- Mapeamento objeto-relacional
- Gera classes de formulários, relatórios, etc.
- Totalmente escrito em PHP
- Invocação pela linha de comandos (shell/DOS) ou interface Web (WebStorage) →

Requisitos

- Compilador da linguagem MetaL
- Plataforma suportada por PHP

  Linux, Microsoft Windows, Solaris, etc..
- Servidor Web para usar o WebStorage

  Apache, Microsoft IIS, etc..
- PHP 4 ou posterior
- Configuração do PHP no arquivo `php.ini`:

```
memory_limit = 32M
safe_mode = Off →
```

Obtenção da ferramenta

- Download de arquivo .tar.gz ou .zip

  http://www.meta-language.net/download.html
- Inclui o compilador MetaL e módulos auxiliares
- Versão em desenvolvimento por CVS

```
cd /caminho/do/htdocs

cvs -d :pserver:cvsread@cvs.meta-language.net:/opt2/ena/metal login →
Senha vazia

cvs -z3 -d :pserver:cvsread@cvs.meta-language.net:/opt2/ena/metal
checkout metal xmlparser readarguments forms metabase metastorage
```

- Cópia da versão em CVS

  http://www.meta-language.net/download.html#snapshots →

Instalação

- Descompactar o arquivo do Metastorage num diretório debaixo da árvore do servidor Web

```
tar zxvf metastorage.tar.gz -C /caminho/do/htdocs

unzip metastorage.zip -d /caminho/do/htdocs
```

- Testar a instalação compilando o componente do projeto de exemplo

```
cd /caminho/do/htdocs/metal/metastorage/projects/cms

php -q -C ../../metastorage cms.component →
```

Criação de componentes

- Um componente é definido num formato XML chamado CPML
- Inclui a definição de classes de objetos persistentes, fábrica, instalação de esquema, formulários e relatórios

```
<?xml version="1.0"?>
  <component>
    <name>cms</name>
    <description>Gerenciamento de conteudo</description>
    <!-- O resto da definição do componente segue aqui -->
  </component> →
```

Criação de classes

- Uma classe define a estrutura de dados dos objetos persistentes
- Inclui a definição de variáveis, regras de validação, relacionamentos e funções para manipular os objetos

```
<class>

  <name>artigo</name>

  <!-- O resto da definição da classe segue aqui -->

</class> →
```

Variáveis de classes

- Uma variável define cada elemento de informação que uma classe armazena
- Inclui o nome da variável, tipo, valor inicial e propriedades específicas de cada tipo

```
<variable>
   <name>body</name>
   <type>text</type>
   <optional>1</optional>
   <initialvalue>Um valor inicial</initialvalue>
   <length>64</length>
   <multiline>1</multiline>
</variable> →
```

OID – Identificadores de objetos

- Cada objeto de cada classe tem um identificador único – *OID* – *Object IDentifier*
- O Metastorage implementa um *OID* como uma variável de tipo inteiro com o nome `id`
- A variável `id` duma classe é implícita, logo não deve de ser incluida na definição da classe
- Na tabela do banco de dados é guardada num campo `auto_increment` e é chave primária →

Variáveis de referência

- Os relacionamentos entre objetos de 1 para 1 são representados por variáveis de referência
- Podem conter referências para objetos de outra classe ou até da mesma
- Uma variável de referência armazena o *OID* do objeto relacionado
- Pode conter uma referência nula quando o objeto relacionado ainda não foi definido →

Relacionamentos de 1 para 1

```
<class>
  <name>homem</name>

  <variable>
    <name>esposa</name>
    <class>mulher</class>
  </variable>
</class>

<class>
  <name>mulher</name>

  <variable>
    <name>marido</name>
    <class>homem</class>
  </variable>
</class> →
```

Relacionamento de 1 para 1 entre objetos da mesma classe

```
<class>

  <name>pessoa</name>

  <variable>
    <name>conjuge</name>
    <class>pessoa</class>
  </variable>

</class> →
```

Relacionamentos externos

- Uma classe pode ter relacionamentos com classes definidas noutros componentes
- Nesse caso cada variável de referência identifica o nome do componente externo
- Cada componente externo referido deve ser declarado no inicio do componente que usa referências →

Componentes externos

```
<component>
  <name>contas</name>
  <file>contas.component</file>
</component>

<class>
  <name>artigo</name>

  <variable>
    <name>autor</name>
    <class>usuario</class>
    <component>contas</component>
  </variable>
</class> →
```

Coleções de objetos

- Os relacionamentos entre um objeto duma classe e um conjunto de objetos de outra classe são representados coleções
- As coleções incluem objetos apenas duma classe, que pode ser outra classe ou até a mesma
- Os objetos contidos numa coleção têm uma variável de referência que armazena o *OID* do objeto que contém a coleção →

Relacionamento de 1 para muitos

```
<class>
   <name>escritor</name>

   <collection>
     <name>artigos</name>
     <class>artigo</class>
     <reference>autor</reference>
   </collection>
 </class>

 <class>
   <name>artigo</name>

   <variable>
     <name>autor</name>
     <class>escritor</class>
   </variable>
 </class> →
```

Coleções mútuas

- Um relacionamento de muitos para muitos entre objetos de duas classes é definido por coleções mútuas
- Cada objeto duma classe contém uma coleção de objetos de outra, e essa contém uma coleção de objetos da primeira
- As definições de coleções mútuas indicam o nome da coleção recíproca na outra classe
- Uma tabela do banco de dados intermediária é usada para guardar os *OIDs* de cada par de objetos relacionados →

Relacionamento de muitos para muitos

```
<class>
   <name>categoria</name>

   <collection>
     <name>artigos</name>
     <class>artigo</class>
     <reference>categorias</reference>
   </collection>
 </class>

 <class>
   <name>artigo</name>

   <collection>
     <name>categorias</name>
     <class>categoria</class>
     <reference>artigos</reference>
   </collection>
 </class> →
```

Funções

Abordagem *JE WIN*: *Just Exactly What I Need*

- **Cada classe pode incluir vários tipos de funções para manipular os seus objetos**
- **O desenvolvedor define que funções cada classe precisa de acordo com as necessidades; o Metastorage gera apenas o código solicitado**
- **Uma função pode ter argumentos passados em tempo de execução**
- **Alguns tipos de função requerem parâmetros adicionais para especificar outros detalhes →**

Funções de classes persistentes

- `addtocollection`

  Adicionar um objeto a uma coleção
- `delete`

  Eliminar o objeto
- `getcollection`

  Obter objetos de uma coleção
- `getreference`

  Obter o objeto definido por uma variável de referência
- `persist`

  Armazenar o objeto
- `removefromcollection`

  Retirar um objeto de uma coleção
- `setreference`

  Definir o objeto de uma variável de referência
- `validate`

  Validar o objeto
- `custom`

  Customização escrita em PHP →

Regras de negócio

- As regras de validação servem para satisfazer requisitos da aplicação também conhecidos por regras de negócio
- O Metastorage pode implementar alguns tipos de validação pré-definidos: `notempty` e `unique`
- As regras de validação são verificadas através de funções do tipo `validate`
- A cada regra de validação definida é associado um código de erro para as aplicações darem o devido tratamento →

Validação

```
<class>
  <name>artigo</name>

  <variable>
    <name>titulo</name>
    <type>text</type>
  </variable>

  <validation>
    <type>unique</type>
    <errorcode>1</errorcode>
    <parameters>
      <variable>titulo</variable>
    </parameters>
  </validation>

  <function>
    <name>validarArtigo</name>
    <type>validate</type>
    <parameters>
      <errorcode>
        <argument>erro</argument>
      </errorcode>
    </parameters>
  </function>
</class> →
```

Classe fábrica

- A classe fábrica é necessária para gerenciar todos os recursos: conexões, transações, objetos em memória, etc..
- A criação de novos objetos ou o acesso a objetos armazenados deve ser feito através de funções da classe fábrica para evitar problemas causados se existissem várias cópias do mesmo objeto em memória
- Cada componente deve definir sempre uma e só uma classe fábrica →

Funções da classe fábrica

- **createobject**
  Criar um novo objeto de uma dada classe
- **getobject**
  Buscar um objeto armazenado de uma dada classe
- **getallobjects**
  Buscar um conjunto de objetos armazenados de uma dada classe
- **starttransaction**
  Iniciar uma transação
- **finishtransaction**
  Terminar uma transação
- **custom**
  Customização escrita em PHP →

```
<factory>
  <function>
    <name>criarArtigo</name>
    <type>createobject</type>
    <parameters>
      <class>artigo</class>
    </parameters>
  </function>

  <function>
    <name>buscarTodosArtigos</name>
    <type>getallobjects</type>
    <parameters>
      <class>artigo</class>
    </parameters>
  </function>

  <function>
    <name>buscarArtigoPorOID</name>
    <type>getobject</type>
    <parameters>
      <class>artigo</class>
      <id>
        <argument>artigo</argument>
      </id>
    </parameters>
  </function>
</factory>
```

OQL – Linguagem de pesquisa de objetos

- Uma função que acessa um ou mais objetos pode ter associada uma condição de pesquisa que os objetos devem satisfazer
- As condições são definidas através duma linguagem de pesquisa de objetos chamada ***OQL*** – ***Object Query Language*** →

Funções de busca de objetos

- As condições em *OQL* podem usar vários tipos de operações e expressões, incluindo usar valores passadas para as funções em tempo de execução como parâmetro
- Condições em *OQL* podem ser associadas através do parâmetro `filter` às funções `getobject` e `getallobjects` da classe fábrica e `getcollection` das classes persistentes →

```
<function>
   <name>buscarArtigoPorTitulo</name>
   <type>getobject</type>

   <argument>
    <name>titulo</name>
    <type>text</type>
   </argument>

   <parameters>

     <class>artigo</class>

     <filter>
         <variable>
           <name>titulo</name>
         </variable>
       <equalto />
         <argument>titulo</argument>
     </filter>

   </parameters>
 </function>
```

Classe de instalação do esquema

- A classe do esquema é responsável pela instalação do esquema do banco de dados
- A classe instala ou atualiza o banco de dados que terá as tabelas para armazenar os objetos persistentes definidos num componente
- Esta classe requer apenas a definição duma função do tipo `installschema` →

```
<schema>
  <function>
    <name>installschema</name>
    <type>installschema</type>
  </function>
</schema>
```

Classes de formulários

- O Metastorage pode gerar classes que implementam casos de uso para executar operações comuns de manipulação de objetos persistentes usando formulários Web
- Os formulários asseguram a satisfação das regras de validação associadas aos objetos
- A apresentação dos formulários é configurada através de temas configuráveis
- Por enquanto, apenas um tipo de caso de uso foi implementado que serve para criar novos objetos de uma dada classe →

Classes de relatórios

- O Metastorage pode gerar classes para processamento maciço de dados de objetos persistentes

  Por exemplo: geração de relatórios, envio de *e-mail* para muitos destinatários, etc..

- As classes de relatórios executam consultas definidas por expressões em *OQL* que podem envolver objetos de várias classes
- As consultas são realizadas por funções que retornam os resultados de forma mais eficiente usando *arrays*, podendo ser ordenados e limitados a uma gama de linhas →

Exemplo de classe de relatório

Exemplo de classe de relatório

Geração do código de um componente

- Depois de criar um componente o programa `metastorage` deve ser usado para gerar o código das classes definidas
- O código pode ser gerado através da linha de comando (shell ou DOS) ou através do interface Web (*WebStorage*)
- O código é gerado num diretório indicado pela opção `-i`, que se for omitida é `install`

```
$ cd /caminho/do/metastorage/projects/cms
$ php -q -C ../../metastorage cms.component →
```

Geração do diagrama de classes em UML

- O Metastorage pode gerar opcionalmente um diagrama das classes geradas em UML
- Para isso gera um arquivo em formato `.dot` que descreve o diagrama no formato *GraphViz*
- O programa *GraphViz* pode transformar o diagrama num gráfico em vários formatos: GIF, JPEG, SVG, Postscript, etc.. →

```
$ cd /caminho/do/metastorage/projects/cms
$ php -q -C ../../metastorage cms.component -g .
$ dot cms.dot -Tgif -o cms.gif →
```

Instalação do banco de dados

- Antes de poder usar o código gerado deve-se instalar o esquema do banco de dados
- A classe do esquema tem algumas variáveis para configurar detalhes da instalação
- Para facilitar a instalação, o Metastorage cria o script `install.php` no diretório `setup`
- O script `global_options.php` define opções comuns para configurar o banco de dados
- Para alterar os valores das opções deve-se mudar o script `local_options.php` →

```
$ cd install/setup
$ php -q install.php →
```

Configuração do banco de dados

Opções definidas em `global_options.php`

- `database_connection`

  String de conexão ao banco de dados em formato do Metabase
- `database_name`

  Nome do banco de dados
- `create_database`

  Determina se o banco de dados será criado na instalação
- `debug`

  Determina se serão geradas mensagens do que é executado

`global_options.php`

```
$global_options=array(
 "database_connection"=>
  "mysql://localhost/cms",
  "database_name"=>"cms",
  "create_database"=>1,
 "debug"=>0
);
```

`local_options.php`

```
$global_options["database_co
  nnection"]=
   "pgsql://localhost/cms";
$global_options["debug"]=1
```

Inicialização do objeto fábrica

- A partir do momento em que o banco de dados for instalado, o ambiente está pronto para implementar aplicações com o código gerado
- Todos scripts de manipulação de objetos persistentes devem começar por criar um objeto da classe fábrica, inicializando variáveis de opções e chamando a função `initialize`
- No final desses scripts deve-se chamar a função `finalize` do objeto fábrica →

Exemplo de uso do objeto fábrica

Exemplo de uso do objeto fábrica

Exemplo de uso de transações

Exemplo de uso de transações

Criação de novos objetos

- Para criar um novo objeto é necessário chamar uma chamar uma função da classe fábrica to tipo `createobject`
- Os dados do objeto retornado podem ser inicializados alterando as respectivas variáveis
- Quando o objeto estiver pronto a ser armazenado deverá ser chamada uma função da sua classe do tipo `persist` →

Exemplo de criação dum novo objeto

Exemplo de criação dum novo objeto

Validação de objetos

- Quando se criam objetos com valores vindos do usuário ou sistemas externos é conveniente validar os valores para evitar inconsistências
- Uma classe com uma função tipo `validate` pode verificar se um objeto foi inicializado com valores inválidos
- Essa função retorna um número inteiro que é um código que serve para distinguir o tipo de erro de validação detectado
- Se a função retornar o código de erro `0`, todas as regras de validação foram satisfeitas →

Exemplo de validação dum objeto

Exemplo de validação dum objeto

Busca de um objeto por *OID*

- Um objeto armazenado pode ser trazido para memória a partir do seu *OID* usando uma função da classe fábrica de tipo `getobject`
- Se não existir um objeto da classe consultada com o *OID* passado para a função ou ocorrer um erro é retornada uma referência nula
- Nesse caso, a variável `error` da classe fábrica deve ser consultada para distinguir se objeto não existe ou se ocorreu um erro →

Exemplo de busca de um objeto por *OID*

Exemplo de busca de um objeto por OID

Busca de todos objetos duma classe

- Para obter todos os objetos de uma dada classe deve ser usada uma função da classe fábrica do tipo `getallobjects`
- Essa função retorna uma referência para um *array* com todos os objetos da classe indicada
- Se não existirem objetos armazenados dessa classe, o *array* será retornado vazio
- Se ocorrer um erro a função retorna uma referência nula →

Exemplo de busca de todos objetos duma classe

Exemplo de busca de todos objetos duma classe

Busca de objetos que satisfazem uma dada condição

- A busca de objetos que satisfazem uma dada condição é feita da mesma forma através de funções da classe fábrica do tipo `getobject` e `getallobjects`, mas usando o parâmetro `filter`
- A única diferença que pode haver é a passagem de parâmetros para a função, que são usados na condição de pesquisa
- De resto, a invocação e o processamento de resultados é idêntico ao caso em que não é usada qualquer condição de pesquisa →

Eliminação de objetos

- A eliminação de um dado objeto em memória deve ser feita através duma função da sua classe de tipo `delete`
- Se o objeto já tiver sido armazenado antes, essa função elimina o objeto do banco de dados
- Esta função também retira o objeto de coleções a que possa pertencer
- A função elimina de memória as referências para o objeto eliminado, porém pode necessitar usar a função `UnSet()` em variáveis que apontam para o objeto, devido ao sistema de *garbage collection* do PHP →

Estabelecimento de relações de 1 para 1

- Uma relação de 1 para 1 entre dois objetos pode ser estabelecida através duma função do tipo `setreference` da classe de um dos objetos
- Essa função recebe como parâmetro uma referência para o outro objeto
- Esse objeto tem de já ter sido armazenado no banco de dados antes de usar essa função
- Se for passada uma referência nula para a função, isso termina um relacionamento que possa ter sido estabelecido antes
- O estabelecimento da relação só é efetivado quando o objeto for armazenado com função do tipo `persist` da sua classe →

Estabelecimento de relações de 1 para muitos ou muitos para muitos

- As relações de 1 para muitos ou de muitos para muitos entre objetos de duas classes são estabelecidas através de funções do tipo `addtocollection` da classe que define a coleção
- Essa função recebe como parâmetro uma referência para o objeto da outra classe a ser incluido na coleção
- No caso de relações de 1 para muitos, esta função é equivalente a uma função do tipo `setreference` da classe do objeto a incluir na coleção, pelo que esse objeto terá de ser guardado a seguir com a sua função de tipo `persist` →

Remoção de relacionamentos entre objetos

- A remoção de objetos envolvidos em relações de 1 para muitos ou de muitos para muitos entre objetos de duas classes é feita através de funções do tipo `removefromcollection` da classe que define a coleção
- Essa função recebe como parâmetro uma referência para o objeto da outra classe a ser removido da coleção
- No caso de relações de 1 para muitos, esta função é equivalente a passar uma referência nula para uma função do tipo `setreference` da classe do objeto a remover da coleção, pelo que esse objeto terá de ser guardado a seguir com a sua função de tipo `persist` →

Busca em coleção de objetos relacionados

- A busca de objeto envolvidos em relacionamentos de 1 para muitos ou muitos para muitos é feita através de uma função do tipo `getcollection` da classe que define a coleção
- Essa função é idêntica a uma função do tipo `getallobjects` da classe fábrica, exceto que restringe a busca aos objetos pertencentes à coleção
- Da mesma forma pode associar expressões em *OQL* para restringir a busca de acordo com critérios de pesquisa →

Fim

Obrigado

Questões?

Referências

- Manual de referência do Metastorage

http://www.meta-language.net/metastorage-documentation.html

- Perguntas frequentes sobre o projeto Metastorage

http://www.meta-language.net/metastorage-faq.html

- Perguntas frequentes sobre como usar Metastorage

http://www.meta-language.net/metastorage-howto.html

- Tutorial do Metastorage

http://www.meta-language.net/metastorage-tutorial.html

- Resolução de problemas comuns do Metastorage

http://www.meta-language.net/metastorage-troubleshootingl.html

- Manual de referência do Metabase

http://www.meta-language.net/metabase.html