Rails ステップアップ講座

2007.11.26

株式会社万葉　大場寧子

y.ohba@everyleaf.com

自己紹介

- 実装大好きプログラマ
- ... → C++ → Java(7年) → Ruby(1.5年)
- Perl CGI, EJB, Servlet/JSP (Struts)

実装大好き

- ベンチャーでマネージャ
- ・・・・orz
- やっぱ実装がしたい

Ruby on Rails との出会い

- 会社をやめたのでさっそく実装を
- Java 環境をホームサーバに・・・作ってもらえません
- 泣く泣く Ruby on Rails に着手

やってみたら

Ruby on Rails サイコー(^0^)/

Award on Rails 2006

- Web家計簿「小槌」
- 大賞
- まつもとゆきひろさん審査員特別賞

http://www.kozuchi.net

それから１年・・・

- Ruby on Rails の仕事100%
- ECサイト
- 医事会計システム
- SNS

現在の仕事

- Ruby on Rails による開発支援
- 会社も作っちゃいました
  - 株式会社万葉
  - http://www.everyleaf.com

Award on rails 2007

- 蔵書管理システム「BookScope」
- http://bookscope.net/
- 久保さんの手伝い

ウェブキャリア賞

＆合宿賞

BookScopeの特徴

- Amazon WebServiceを利用してデータ取得
- バーコードリーダーで登録
- 本棚.org インポート
- フィード配信
- 蔵書の貸し借りを管理できる

本日のテーマ

- Ruby on Rails で実践的にモノを作ろう
- 初級者向け講座/入門本の次ステップへ
- 高度な話題も少し

みなさんの状況は？

- Ruby on Rails をさわったことがある？
- Ruby on Rails を仕事で使ったことがある？

Hello, RoR

- まずこんなことを勉強しますよね
- Scaffold
- データベースのCRUD
- Session, Request

開発を始めると

- とにかくコードを書いて目的を果たす
- あ、これ Rails に機能があったんだ
- あ、これもあったんだ
- 全然 Ruby らしくないコードになってた
- 最初からこれを使えばよかった etc...

ということで

- 結局は、作らないと身につかない
- でも、いいヒントがあれば効率的
  - 日頃よく使うAPI、パターン
  - 先に知っておきたかった Rails機能, Plugin
  - Ruby らしいコードにするコツ

熟練へのキーワード

RESTな設計

愛

Hash と Array

セキュリティ

ActiveRecord

Module による Mix-in

DRY

Ajax

キャッシュ

よくある問題と解決パターン

Plugin

愛のある開発

- ツールを揃える - IDE, SVN, Trac, Wiki, UML
- よい仕様、よい設計、よい実装を追求する
- よい名前をつける
- テストを書く
- 助け合う

出発！

Java経験者のための Rubyのコツ

- 0も真なり
- ||=
- １行で書いてみる
- Hash と Array
- クラスメソッド

Ruby - 0も真なり

```
s = 0
if s
  p "0も真なり"
end
```

- 偽は nil と false だけ
- 0 も真

nil チェック

```
if value != nil
  ...
```

かっこわるい

```
if value
  ...
```

通常これで十分

```
unless value.nil?
  ...
```

厳密なチェック

1行で書いてみる

```
if !id
  raise "no id"
end
```

Java から来ると１行に抵抗感

```
raise "no id" if !id
```

中身が１行なら１行が見やすい

```
raise "no id" unless id
```

異常判定は unless 向き

Hash と Array

- これがなくては始まらない
- APIをよく読んで何ができるか覚えよう

Hash

```
hash = {key => value, key => value, ...}
```

```
value = hash[key]
```

記法に目を慣らそう

```
obj.foo( {key => value, key => value} )
```

```
obj.foo(key => value, key => value)
```

引数のハッシュの{} は省略できる場合も

名前つき引数のように使われる

Array

自分でゴリゴリ走査するのはほとんどが無駄

```
admin= nil
for element in @array
  if element.name == "admin"
    admin = element
    break
  end
end
```

```
admin = @array.detect{|e| e.name == "admin" }
```

クラスに関する特色

- クラスメソッドも継承される
- リフレクションが強力。メソッドの追加、書き換えなどが簡単
- public, protected, private がJavaと微妙に違う
- 名前空間の解決は Module で

||=

```
name ||= "名無し"
```

- 偽だったら代入
- デフォルト値の挿入に便利
- 省略しないで書くと以下のようになる

```
name || name = "名無し"
```

ex) value += 1

private, protected, public

どこから呼ばれるか		インスタンス内			インスタンス外				
		クラス	派生クラス		クラス	派生クラス		他	
			パ内	外		パ内	外	パ内	外
private	Ruby	○	○		×	×		×	
	Java	○	×	×	○	×	×	×	×
protected	Ruby	○	○		○	○		×	
	Java	○	○	○	○	○	×	○	×
なし	Java	○	○	×	○	○	×	○	×
public	Ruby	○	○		○	○		○	
	Java	○	○	○	○	○	○	○	○

private の特徴

```
item.my_private_method
```

```
self.my_private_method
```

private メソッドは

レシーバを使って呼べない

```
my_private_method
```

レシーバなしのメソッド呼び出し

この形ならprivate メソッドも呼べる

Rubyは

- 派生クラスに対してオープン
- インスタンスの内外を区別

RoR での使い方

- 公開メソッド - public
- それ以外 - private
- protected は以下の場合に使う
  - ActiveRecordのコールバックを直接書く時
  - インスタンスを超えた処理で完全公開はしたくないもの

クラスメソッド

```
class Foo
  def self.foo
    p 'foo'
  end
end
```

呼ぶときは

```
> Foo.foo
```

```
> f = Foo.new
> f.class.foo
```

クラスメソッドも派生クラスに受け継がれる

本日のメインテーマ

ActiveRecord

ぜひ覚えてほしいこと

- 単一テーブル継承
- ポリモーフィック関連
- コールバック
- :joins と :include

その他の話題

- 検証
- カウンターキャッシュ
- Acts as List

ActiveRecord とは

- O/R マッピング
  - RDBデータがオブジェクトとして扱える
- ActiveRecord とは
  - “アクティブレコードパターン” の実装
  - クラスとテーブル、オブジェクトとレコードが対応する

シンプルなActiveRecord

1テーブルが1クラスに対応

1レコードが1インスタンスに対応

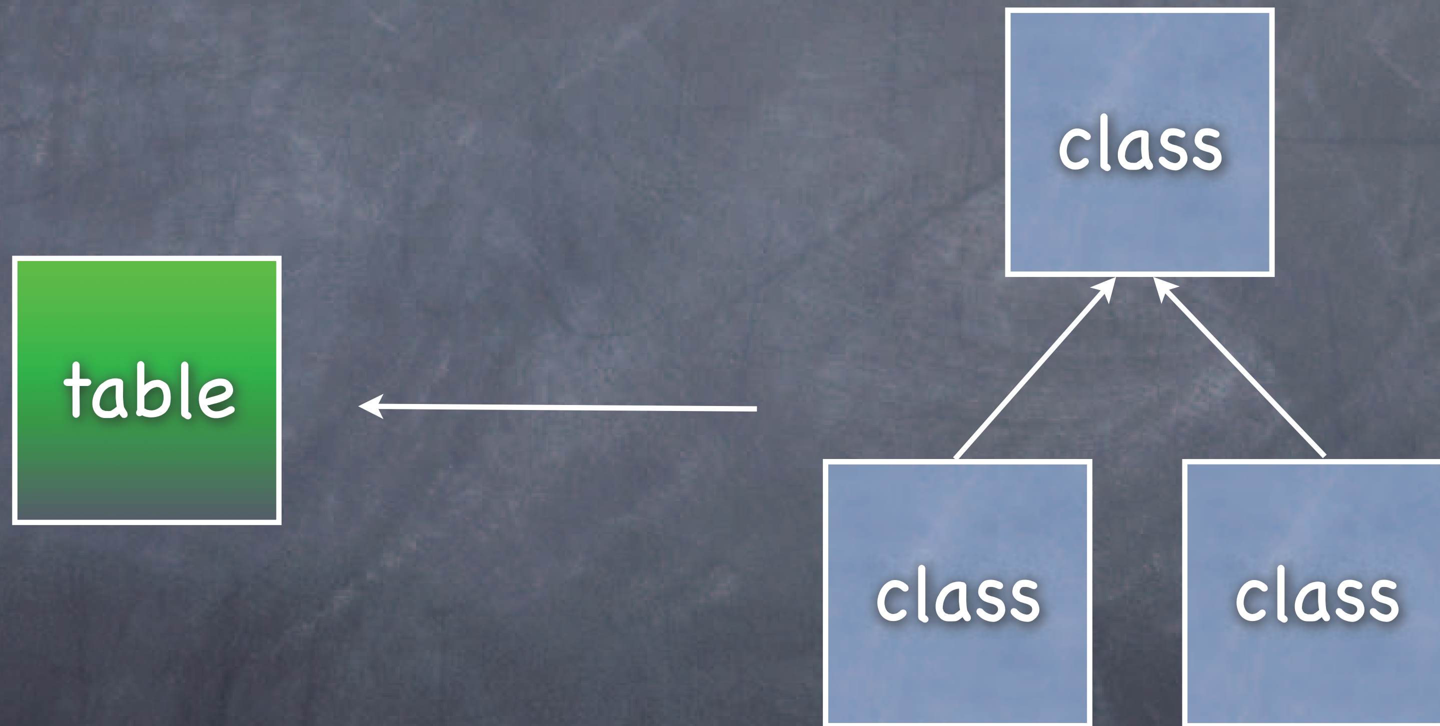
id	1
name	太郎

@user

(id = 1, name = ‘太郎’)

単一テーブル継承

継承関係にある複数のクラスを

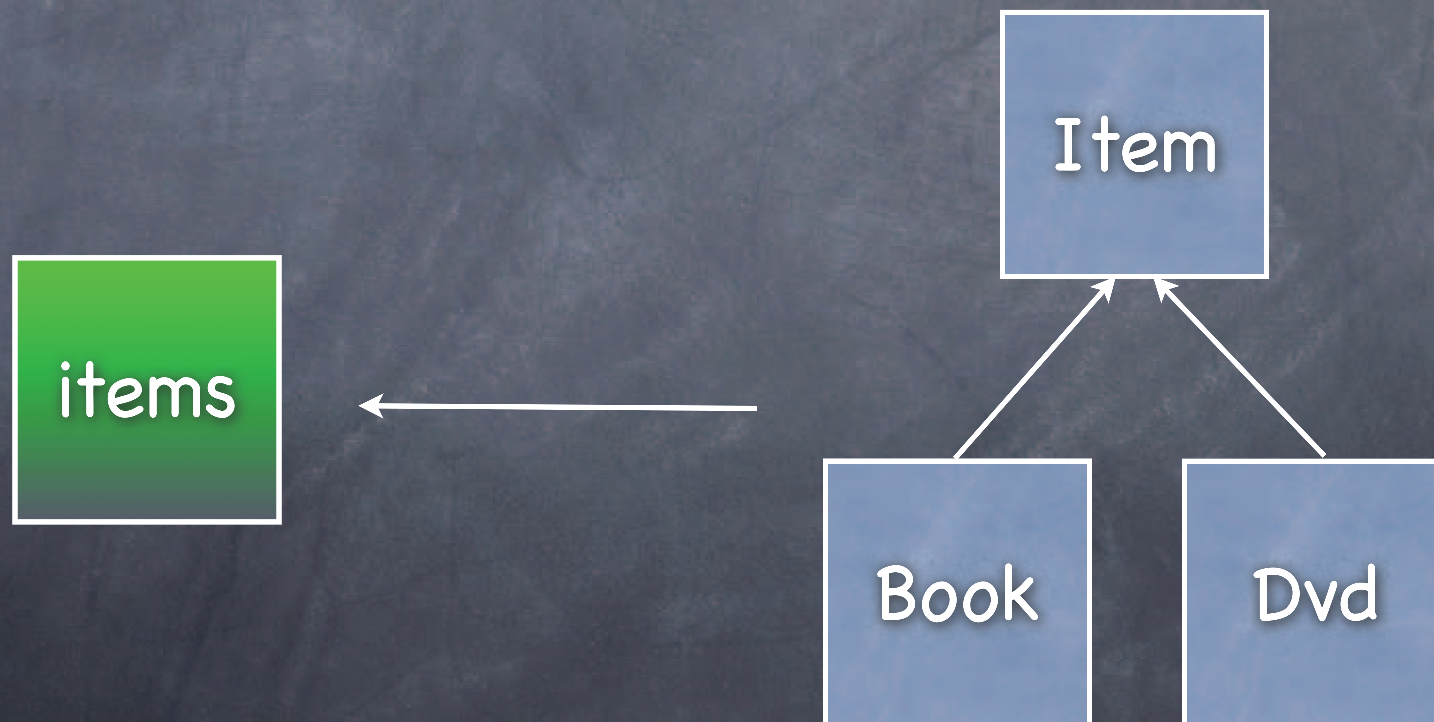


１つのテーブルに対応づける

単一テーブル継承の例(1)

Item, Book, Dvdクラスを

みんな items テーブルに格納



検索時の使い分け例

```
Item.find(:all, :order => `created_at desc')
```

登録順に、DVD, 本、その他商品を混ぜて取得

```
Book.find(:all, :order => `created_at desc')
```

本だけを取得

検索で得たオブジェクトは

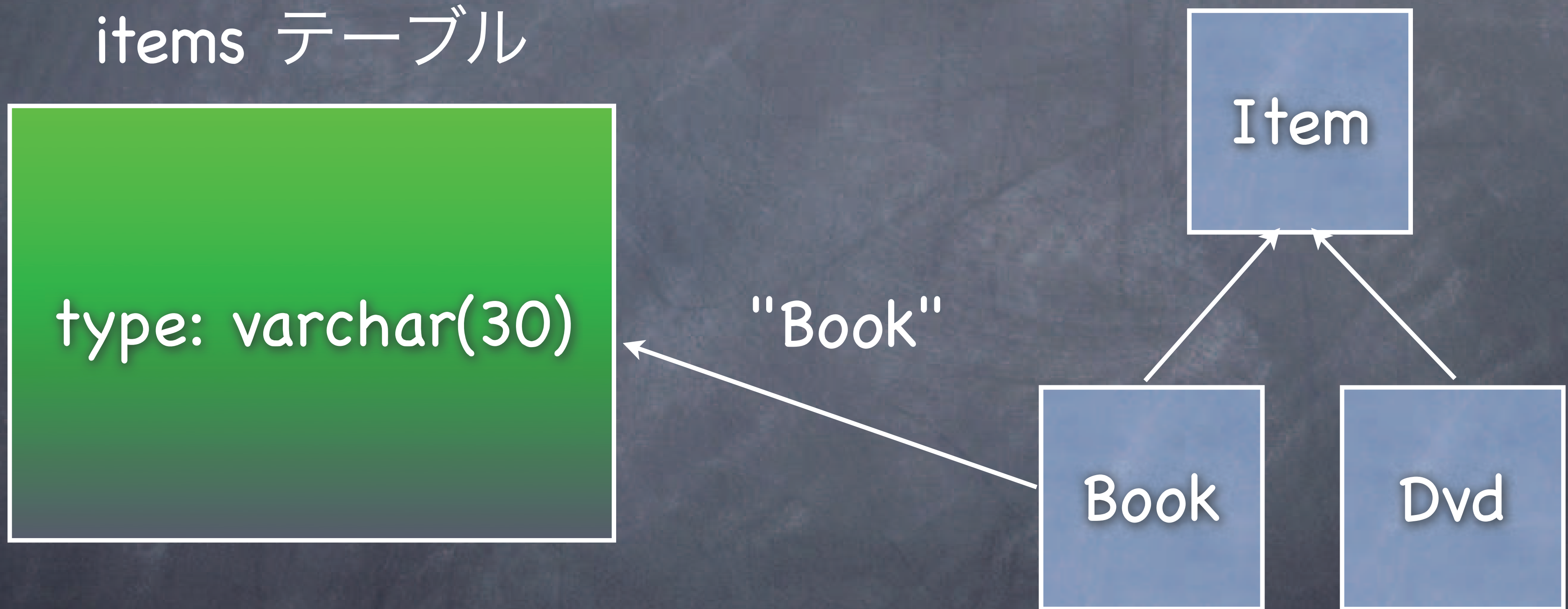
対応するクラスのインスタンスとなる。

微妙な動作の違いを美しく記述できる。

単一テーブル継承の発動条件

- テーブルに文字列を格納できる type カラムがあること
- 同じテーブルに入れたいクラス同士が継承関係でまとまっていること

type カラム

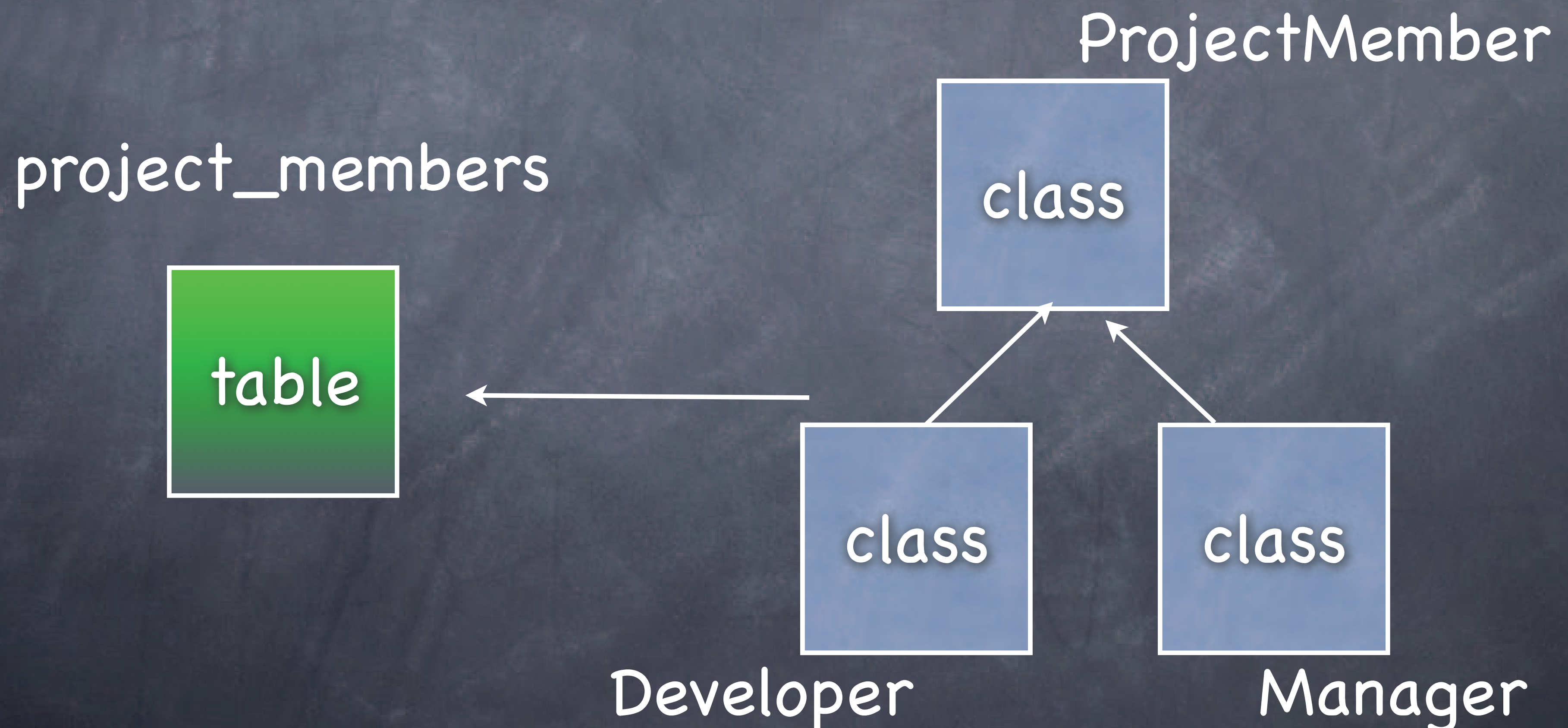


type カラムにクラス名が格納される

単一テーブル継承の例(2)

プロジェクト参加者を

ManagerとDeveloperに分ける



単一テーブル継承の使いどころ

- 似ているけどちょっと違うモデル同士を継承を使って実装したい
- 同じようなカラム構成のテーブルを２つ作りたくないなぁ・
- item_type とか role とか、データの種類を表すカラムがあって、if 分岐が多い

次の話題

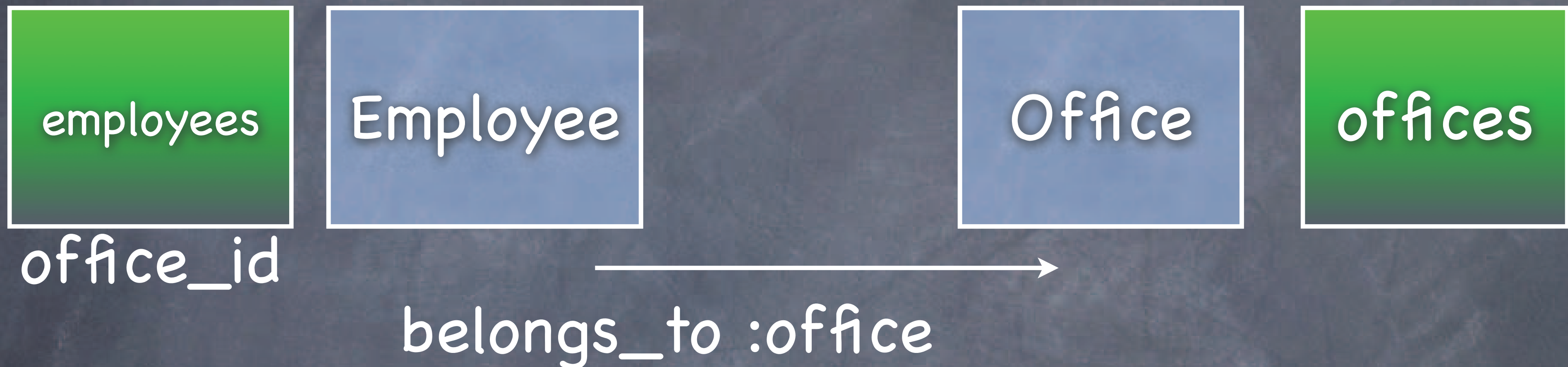
ポリモーフィック関連

Polymorphic Association

関連(Association)

- テーブルとテーブルの関係 = クラスとクラスの関係 (=関連)
- belongs_to
- has_one
- has_many

belongs_to




```
class Employee < ActiveRecord::Base
  belongs_to :office
end
```

相手のキーを持つ ＝ 弱い＝ 相手に belongs_to(従属)

関連の使い方

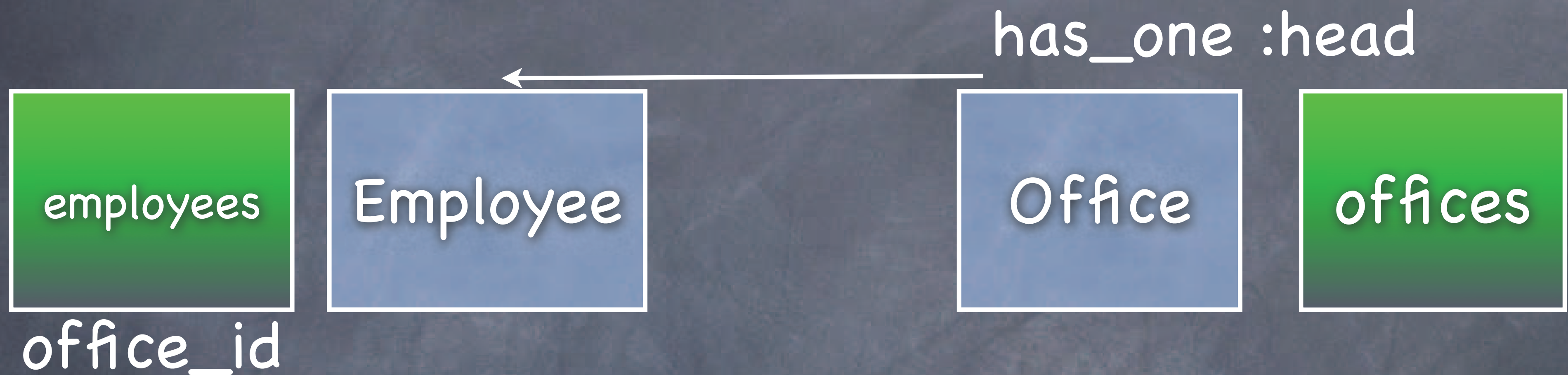
Controller で Employee インスタンスをとっておき

```
@employee = Employee.find(params[:id])
```

View で所属事務所を表示

```
<table>
  <tr>
    <th>事務所</th>
    <td><%= @employee.office.name %></td>
  </tr>
</table>
```

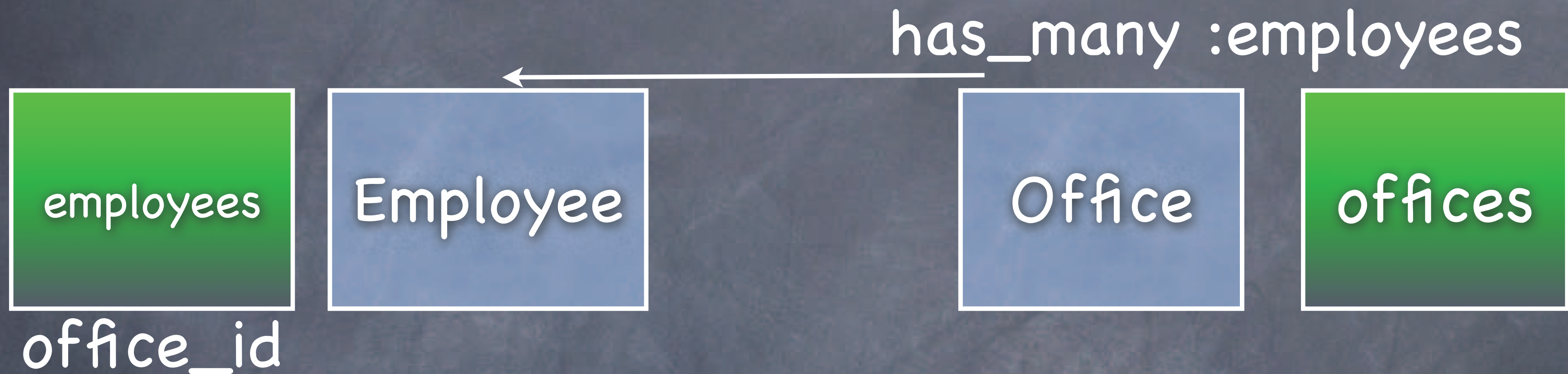
has_one




```
class Office < ActiveRecord::Base
  has_one :head, :class_name => 'Employee',
                 :conditions => "head = 1"
end
```

相手にキーを持たせる ＝ has_xxx (相手は私のもの)

has_many




```
class Office < ActiveRecord::Base
  has_many :employees
end
```

has_one と has_many は

相手を１つと想定するか、複数と想定するかの違い

has_many の使用例

Controller で office インスタンスをとっておき

```
@office = Office.find(params[:id])
```

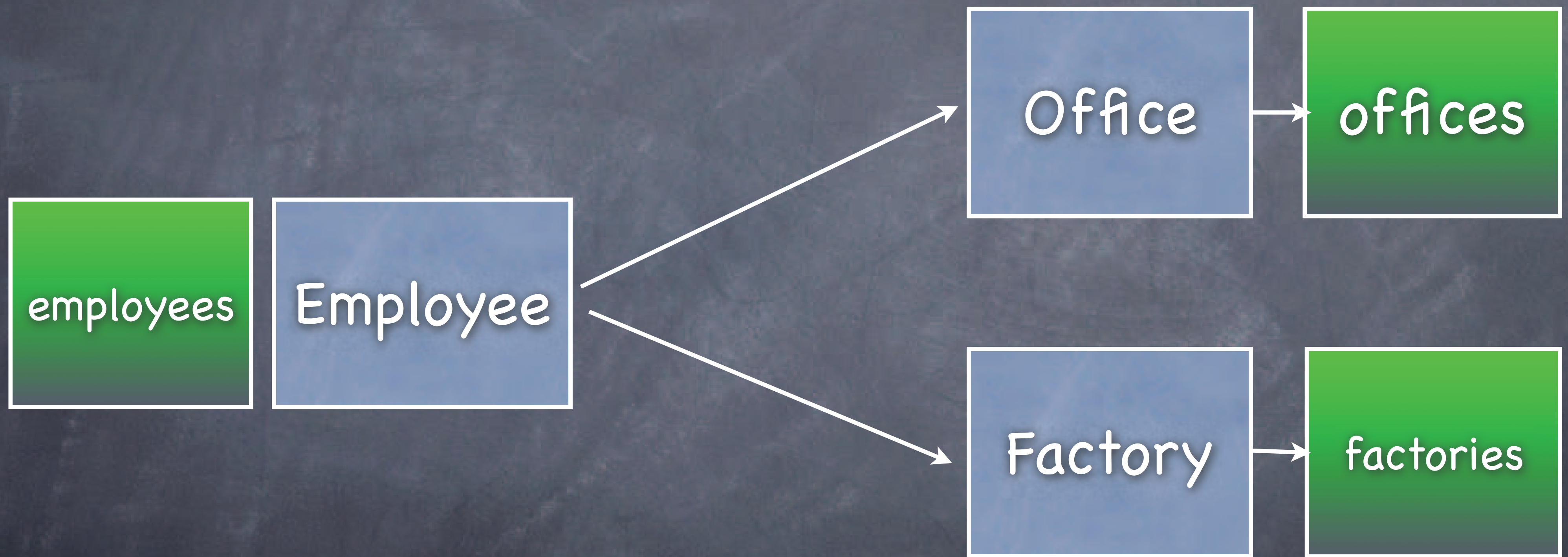
View で従業員を一覧

```
<% for e in @office.employees -%>
  <div><%= e.name %></div>
<% end -> 
```

ポリモーフィック関連

- 複数のテーブルを１つの関連として扱う
- 例
  - 事業拠点には工場と事務所があり、別のテーブルで管理している
  - どちらも事業拠点なので、ビジネスロジック上は似ている点が多い

モデル構造



Employee は、Office か Factory に所属する。

ポリモーフィック関連を使わないと…

```
class Employee < ActiveRecord::Base
  belongs_to :office
  belongs_to :factory
end
```

```
<% if @employee.office %>
  <%= @employee.office.name %>
<% else -%>
  <%= @employee.factory.name %>
<% end -%>
```

ポリモーフィック関連を使ってこうしたい

```
class Employee < ActiveRecord::Base
  belongs_to :workplace
end
```

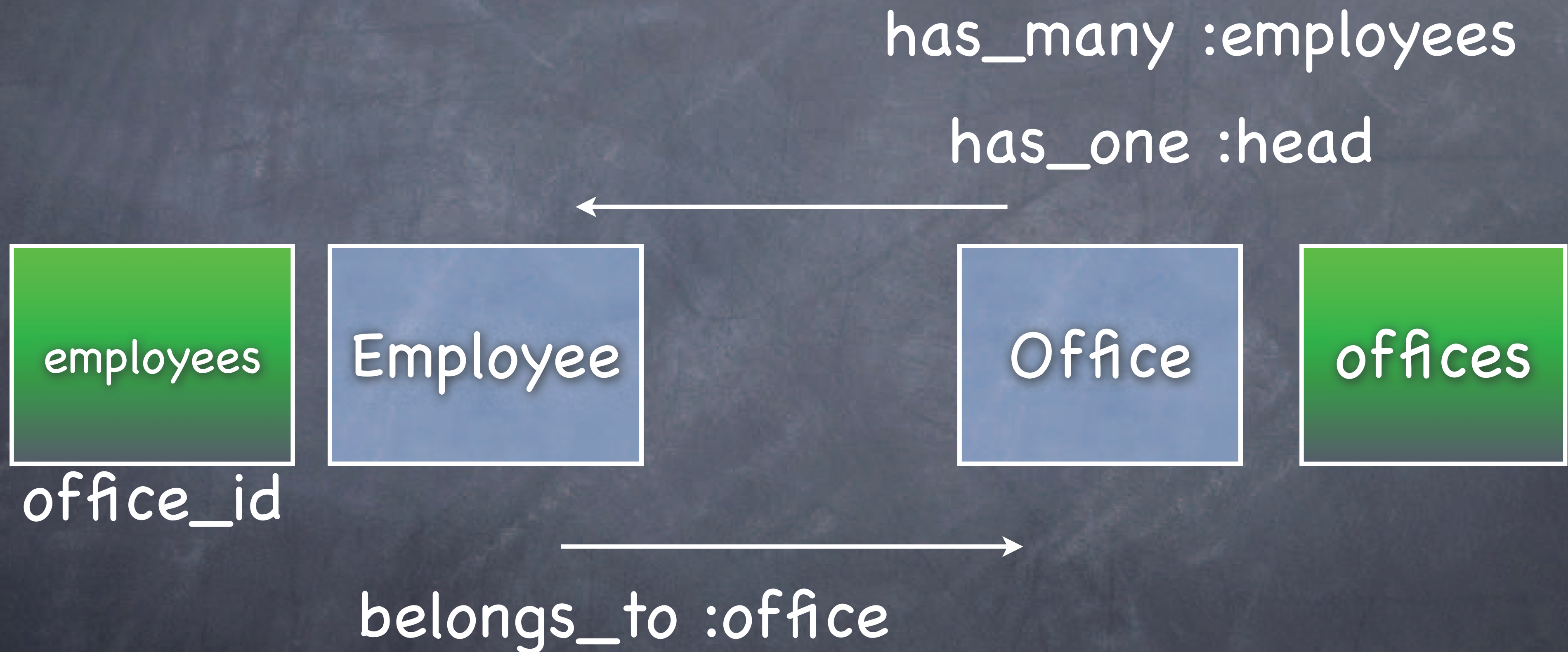
Office も Factory も WorkpPlace の一種だ！

```
<%= @employee.workplace.name %>
```

区別する必要のないところでは同一視

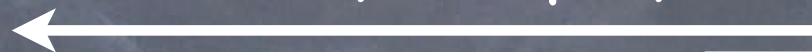
コードがすっきり！

関連のおさらい

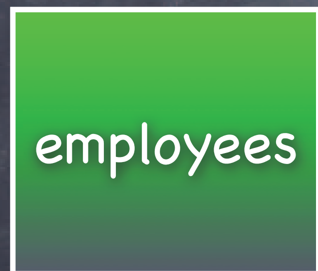


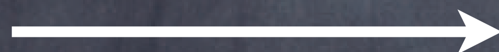
ポリモーフィック関連

has_many :employees, :as => work_place



work_place_id
work_place_type





belongs_to :work_place, :polymorphic => true

基本的アイディア

- 相手のキーを抱える側が、相手のクラス(xxx_typeカラム)も一緒に抱えてあげる
- クラスとキーがあればどこへでも探しに行ける

記述例

```
class Employee < ActiveRecord::Base
  belongs_to :work_place, :polymorphic => true
end
```

```
class Office < ActiveRecord::Base
  has_many :employees, :as => :work_place
end
```

```
class Factory < ActiveRecord::Base
  has_many :employees, :as => :work_place
end
```

work_place_id, work_place_type

- xxx_id, xxx_type の xxx は自由に命名できる
- 通常、xxx_type についての SQL 断片を conditions などに書かなければならないことはない。ActiveRecord がよきにはからってくれる。

ポリモーフィック関連の典型的な使いどころ

- いろいろなオブジェクトに、共通のフィーチャーをつけたい
  - コメントをつける
  - タグをつける
  - 写真をつける

例：コメントをつける

commentable_id
commentable_type

相手は誰でもいい

「商品」に
コメントをつけてみよう

```
class Item < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

これだけで、

```
@item.comments.create(:body => ‘コメントだよ〜’)
```

簡単にコメントデータを追加したりできる

単一テーブル継承と ポリモーフィック関連

- クラス名を(xxx_)typeカラムに入れる
- データベース設計とクラス設計の差を吸収してくれる　→　クラス設計の自由度Up！
- 設計前に理解しておきたい

コールバック

- ActiveRecord モデルインスタンスのライフサイクルにあわせて呼ばれる
- インスタンス構築、Save、検証、削除
- ビジネスロジックを書くときは、コールバックに入れられないかまず考えるとよい

例えばこんな利用例

```
# 親を更新したら子も強制的に更新
def after_update
  children.each {|c| c.save!}
end
```

その他の利用例

- 検証を実行する前に、カタカナをひらがなに変えておきたい　→ before_validation
- モデルを削除するときに、対応するデータファイルも削除したい　→ after_destroy

余談 - 上達の流れ

Controller にコードを書きまくる

Modelに独自メソッドを書く

Rails APIを使うことで解決

コールバックへ移す

正しい解だった

:joins と :include

特定の都道府県に勤めている社員を検索

```
employees = Employee.find(:all,
      :select => "employee.*",
      :conditions => ["offices.prefectue_id = ?",
                                    prefecture_id],
      :joins => "left join offices on
                  employees.office_id = offices.id")
```

:joins はSQL の join 部分

:joinsと:include (2)

社員オブジェクトをとるためのDB検索で

職場オブジェクトも取得しておく

```
class Employee < ActiveRecord::Base
  belongs_to :office
end
```

```
employee = Employee.find(@id,
              :include => :office)
```

検証

- 基本的な検証APIは用意されている
  - validates_presence_of etc...
- 自分でも書ける
- 検証 = errors にエラーを登録する

```
def validate
  errors.add_to_base(“奇数はだめ！”) unless quantity % 2 == 0
end
```

検証を使う

```
if @employee.save
  flash[:notice] = ‘セーブしました。’
  redirect_to :action => ‘index’
else
  render :action => ‘new’
end
```

Controller では、save の返り値で検証結果をチェック

```
<%= error_messages_for ‘employee’ %>
```

View で検証エラー表示

検証についての所感

- 大部分は既存のAPIで足りる
- 既存のAPIで出来ないと思っても、オプションを見ると出来ることがある
- エラーメッセージがデフォルトだと英語！
    - 以前は ActiveHeart Plugin
    - 最近は GetText , Scpecial Generator など

Acts as List

- has_many で定義した子モデルたちに「順序」がある
- 例)
  - 家計簿の同じ日の記入
  - 複数著者がいる場合の並び方

Acts as List を使う

- テーブルに position カラムを用意

```
class Note < ActiveRecord::Base
  has_many :lines, :order => "position"
end
```

```
class Line < ActiveRecord::Base
  belongs_to :note
  acts_as_list :scope => :note
end
```

rails 2.0 では

プラグインへ

新しくセーブするとき、自動的に適切な

position を割り当ててくれる

カウンターキャッシュ

- has_many 関係にある子モデルが何個あるかを、親モデルのカラムに格納しておく
- 編集にくらべて集計結果の表示が頻繁な要件では高速に処理できて便利
- 例)　事務所一覧で、所属社員数を表示したい

カウンターキャッシュを使う

```
class Employee < ActiveRecord::Base
  belongs_to :office, :counter_cache => true
end
```

- offices テーブルに、employees_count というカラムを用意
- デフォルトを0にすること

ActiveRecord編 終了

- API仕様書を何度でもよく読みましょう
- 特にAssociation 系 (has_manyなど)

http://api.rubyonrails.org/

RESTな設計

- Controller 設計はまず URL 設計から
- 記事にコメントをつけるアクションとして以下のどっちがかっこいい？
- entries/new_comment/1
- entries/1/comments/new
- 後者なら entries/1/comments/3 などに発展

RESTな設計は経済的

- 操作対象のモデルクラスごとにController
- 短い統一されたメソッド名
- routes.rb が定義しやすい
- Filter が機能しやすい
  - 親オブジェクト（先の例では@book）の存在チェックを一斉にできて見やすい

rails 2.0

- このあたり（RESTな設計）に関するサポートが入るらしい

よくある問題とヒント

- 認証
- 権限
- 親子モデルをセットにした編集
- 論理削除
- アップロード・画像処理
- ローカライゼーション（多言語対応）

認証

- acts_as_authenticated plugin 他
- before_filter を利用
- 個人的によくやる手法

```
before_filter :load_user
private
def load_user
  @user = User.find(session[:user_id])
end
```

権限

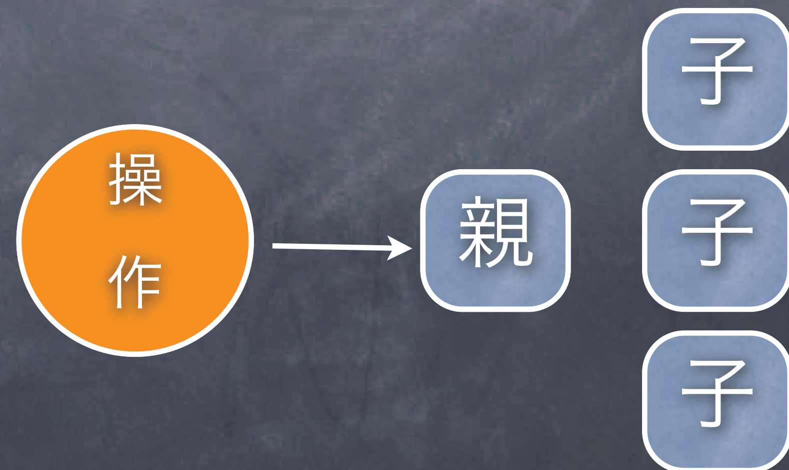
- 今のところ、要件ごとに作ることにしている
- 個人的にはモデルに管理させる方法をよく使う

```
@entries = Entry.secure_find(@user, options)
```

　ユーザー（要求者）を渡し、権限のあるものだけをもらう。
　返ってきたオブジェクトはどんなアクションが許可されているかしっており、違反すると例外。

親子モデルをセットにした編集

- 全部 create するときは親のsaveで保存される
- update のときは子は自動では更新されない
- 子の削除を親更新と同期させたいときは自前でケアが必要
- validates_association
- :dependent => :destroy



論理削除

- acts_as_paranoid プラグイン
- 簡単
- 削除、検索、関連など、各方面を拡張してくれる

画像アップロード

- file_column プラグイン
- RMagickと一緒につかう
- 手軽にアップロードと画像処理ができる
- I/F・モデルの柔軟性に凝った別のプラグインを公開予定だったりします

多言語対応

- GetText (gem で入れるパッケージ)
- Globalize プラグイン

使ったことのある
その他のプラグイン

- acts_as_taggable
  - タグづけ
- SpecialGenerator
  - 高機能な Scaffold

プラグインとのつきあい

- 要件に合うならば使う
- ソースを読む覚悟で使う
- プラグインを読むのは勉強になる
- Rails のバージョンアップに注意
- 多言語化・日本語化しにくい場合も

RubyをDSLとして使う

DSL = ドメイン特化言語

```
has_many :employees
```

- ARのクラスメソッドを呼んでいるとは通常、意識しない
- Rails 専用記法のように理解される
- = DSL として使っている

黒魔術っぽい領域

- 例）String クラスに独自のメソッドが追加されている
  - under_score, pluralize ...
- acts_as_paranoid プラグインをいれたらなぜ論理削除ができるようになるの？
- どうやっているの？

ヒント

- クラスにメソッドを追加する
- Module による Mix-in
- メタプログラミング

クラスにメソッドを足す

- Java ではロードしてからメソッドを足すのは大変
- Ruby だと以下を実行するとできちゃう

```
class Object
  def yeah
    p ‘yeah!’
  end
end
```

Module の Mix-in

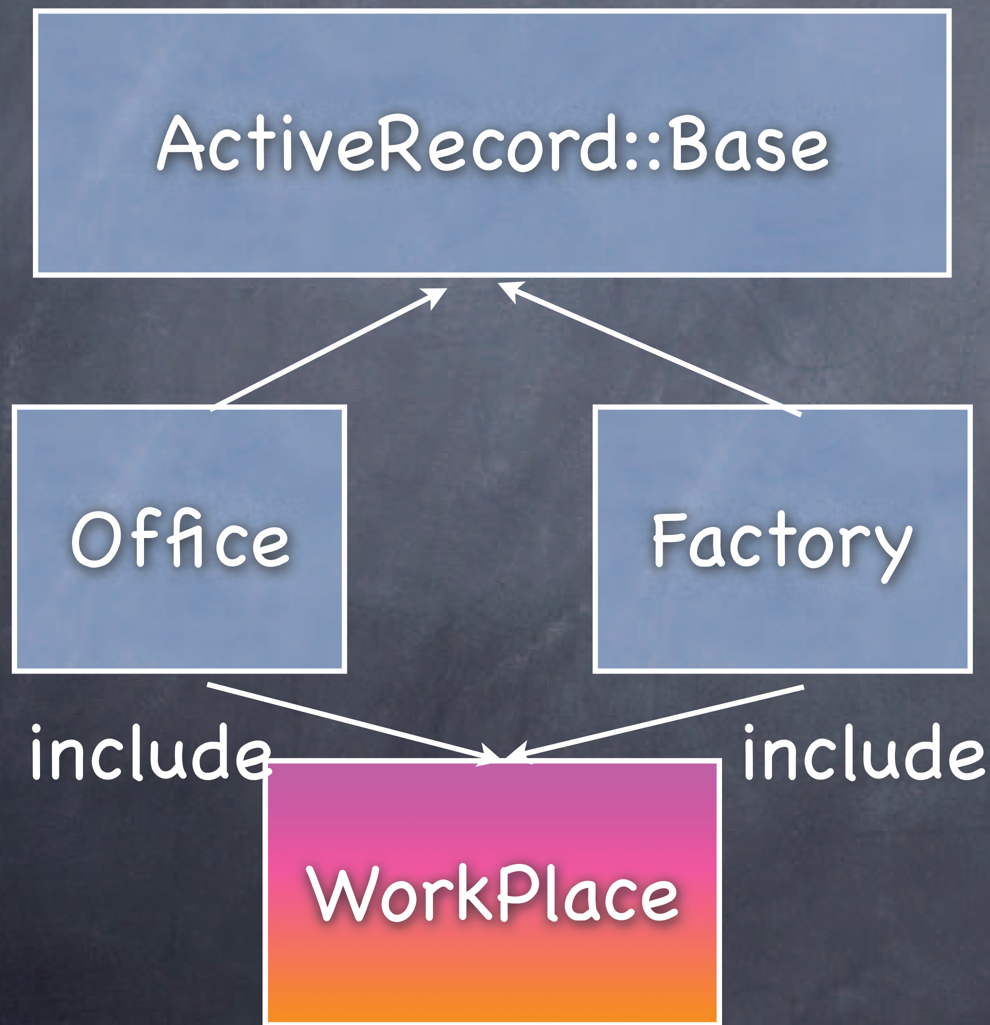
- Module を使ってインスタンスメソッドをクラスに持たせることができる

```
module Human
  def age(date)
    # 誕生日から計算(略)
  end
end
```

include

```
class Employee < ActiveRecord::Base
  include Human
end
```

ポリモーフィック関連と相性がよい



継承がつかえなくても

Moduleで汎化

共通処理を一カ所に記述できる

include のタイミングでクラスメソッドが呼べる

```
module WorkPlace
  # 職場モデルはどれも社員とつながってる
  def self.included(base)
    base.has_many :employees,
                  :as => :work_place
  end
end
```

クラスメソッドも Moduleから渡せる

```
module WorkPlace
  module ClassMethods
    def foo
      p 'foo'
    end
  end
  def self.included(base)
    base.extend(ClassMethods)
  end
end
```

privateメソッドをどうしても呼びたい？

```
obj.__send__(:himitsu)
```

※Ruby1.9では別の方法をとることに

メソッドを動的に足す

```
clazz.__send__(:define_method, :yeah) {
  p ‘yeah!’
}
```

削除、再定義などもできる

やりたい放題 (^o^)/

プラグインの init.rb

- アプリケーション実行前に読み込まれる
- 既存のクラスの拡張に便利
- Module を読み込ませ、既存のクラスに include させ、クラスメソッドを追加
- → DSLっぽく使えるようになる
- 例) acts_as_paranoid

例えば・・・

- 文字列カラムに NULL を許可しているが、
  空文字列は許可したくない

email カラムについては以下のように書けばいい

```
def email=(value)
  value = nil if value.blank?
  self[:email] = value
end
```

対象カラムが多いなら

こんな風に書きたい

```
class User < ActiveRecord::Base
  nullify_blank :email, :address, :tel, :fax, :url
end
```

こんな感じで

vendor/plugins/my_extension/init.rb

```
class ActiveRecord::Base
  def self.nullify_blank(*args)
    args.each do |attr|
      define_method(:"#{attr.to_s}=") do |v|
        v = nil if v.blank?
        self[attr.to_sym] = v
      end
    end
  end
end
```

挑戦しましょう

- Module や Helper を活用してコードの重複をなくす
- DSLっぽく作っていくとかっこいい
- 将来の生産性がどんどんあがっていく
- まずは重複したコードをどうにかしたいという感性・情熱・執念から

ご静聴

ありがとうございました。