第 10 章

バリデーション (1)

> この章から最終章（第 16 章）までの 7 章で、PicoPlanner の予定項目追加・編集フォームにバリデーション機能を加えていきます。バリデーション（validations）とは、モデルオブジェクトの各属性にセットされた値を検証することです。ユーザーがフォームに不正な値を入力した際には、エラーメッセージを加えてフォームを再表示します。

10.1　バリデーションとは

現時点での PicoPlanner には、Web アプリケーションとして不備な点があります。それは、ユーザーが予定項目追加・編集フォームに対して不正な値を入力してもそのまま保存されてしまうということです。

ここでいう「不正な値」というのは、法律的あるいは倫理的な意味合いではありません。Web アプリケーションの目的に合致しない、あるいはデータベースに不整合を生じさせる、といった意味で使っています。英語に直せば "invalid values" となります。例えば、以下のような値は不正です。

- 空の件名
- 開始日時よりも前の終了日時
- 開始日よりも前の終了日

こういった不正な値がデータベースに保存されるのを防止するための仕組みが**バリデーション**（validations）です。「（値の）検証」と呼ぶこともあります。Rails では、モデル定義の中でクラスメソッド `validates` と `validate` を用いて

バリデーションの設定を行います。メソッド名の末尾における “s” の有無に注意してください。

10.2 文字列が空でないことの検証

では、具体例を通じてバリデーションの設定方法を学んでいきましょう。

まず、ターミナルで rails c コマンドを実行し Rails コンソール（『初級②』第 7 章）を開いてください。そして、以下のように式を順に入力してください。

```
> item = PlanItem.new
=> #<PlanItem id: nil, name: nil, ...>
> item.name = ""
=> ""
> item.valid?
=> true
```

モデルオブジェクトの valid? メソッドは、このオブジェクトに対してバリデーションを行い、その結果を真偽値（true または false）で返します。正確に言えば、このオブジェクトが持つすべての属性について個々にバリデーションが行われ、すべての検証がパスすればオブジェクト全体が妥当（valid）であると判定されます。

ここでは件名（name）属性に空文字をセットした PlanItem オブジェクトに対してバリデーションを行い、妥当であるという検証結果（true）が返ってきています。

PicoPlanner の仕様では、空の件名を持つ予定項目を許容すべきではないので、name 属性の値が空でないことを検証するバリデーションを加えましょう。

Rails コンソールはそのままにして、テキストエディタで PlanItem モデルのソースコードを開き、次のように書き換えてください。

app/models/plan_item.rb

```
   :
17    class PlanItem < ApplicationRecord
18      scope :natural_order,
19        -> { order(starts_at: :asc, all_day: :desc, id: :asc) }
20 +
```

```
21 +    validates :name, presence: true
22
23      before_save do
 :
```

クラスメソッド validates は、引数として属性名を表すシンボルを取り、その属性に対するバリデーションを設定します。presence オプションは、バリデーションの種類を示しています。このように記述すると、name 属性が「空」のときに値が不正（invalid）であると判定されるようになります。

Rails では、ある文字列が以下に挙げる文字のみから構成されるときに「空（blank）」であると判定されます。

- 半角スペース（" "）
- 水平タブ（"\t"）
- 垂直タブ（"\v"）
- 改行（"\n"）
- キャリッジリターン（"\r"）
- 改ページ（"\f"）
- NEL（"\u0085"）

つまり、ユーザーがフォームに何も入力せずに送信した場合だけでなく、何個か半角スペースを入力して送信した場合もバリデーションに失敗します。

> 垂直タブ、改ページ、NEL は一般的なキーボードからは入力しにくいので、ブラウザから送信されてくることは考えにくいです。なお、NEL（"\u0085"）は、IBM 系のメインフレーム等で使われている文字セット EBCDIC における改行です。「次行記号」とも呼ばれます。

では、Rails コンソールに戻って試してみましょう。

```
> reload!
Reloading...
```

ソースコードを書き換えたので、まず reload! メソッドを呼び出して Rails コンソールをリセットしました。

```
> item = PlanItem.new
=> #<PlanItem id: nil, name: nil, ...>
> item.name = ""
=> ""
> item.valid?
=> false
```

続いて、さきほどと同じように PlanItem オブジェクトを作り、長さ 0 の空文字をセットしてバリデーションを行いました。false という結果が返ってきていれば OK です。

```
> item.name = " "
=> " "
> item.valid?
=> false
```

半角スペース 1 個からなる文字列を件名（name 属性）にセットすると、バリデーションの結果は false です。私の説明した通りですね。

```
> item.name = "A"
=> "A"
> item.valid?
=> true
```

最後に、文字列 "A" を件名にセットするとバリデーションは成功します。

10.3 HTML5 Form validation errors remover の導入

次に進む前に、本シリーズでの推奨ブラウザである Google Chrome に HTML5 Form validation errors remover というプラグインを導入します。以下、このプラグインを「Errors Remover」と呼ぶことにします。

Chrome で次の URL を開きます。

https://chrome.google.com/webstore/category/extensions

そして、検索ボックスに「HTML5 Form validation」と入力して Enter キーを

押し、検索結果の中から「HTML5 Form validation errors remover」を探して「CHROME に追加」ボタンをクリックし、確認ポップアップウィンドウが開いたら「拡張機能を追加」ボタンをクリックします。すると、Chrome のツールバーにプラグインのアイコンが加わります（図 10.1）。

図 10.1　HTML5 Form validation errors remover のアイコン

Errors Remover は、HTML5 で導入されたフォームバリデーション機能を一時的に無効にするためのプラグインです。PicoPlanner の予定項目追加・変更フォームでは、「説明」欄以外の入力欄に対して `required` 属性が付けられていて、ユーザーがそれらの入力欄を空にしたまま送信ボタンをクリック（タップ）すると、ブラウザ上に入力を促すメッセージが表示されます。この機能を無効にします。

このプラグインを利用することで、ブラウザから `name` パラメータが空であるようなフォームデータが送信された場合の動作確認を行うことができます。

実際に試してみましょう。PicoPlanner のサーバーを起動してから、ブラウザで「予定の追加」ページを開きます。まず、何も記入せずに「追加」ボタンをクリックすると、「件名」の入力欄上に「このフィールドを入力してください。」というメッセージが出ます（図 10.2）。

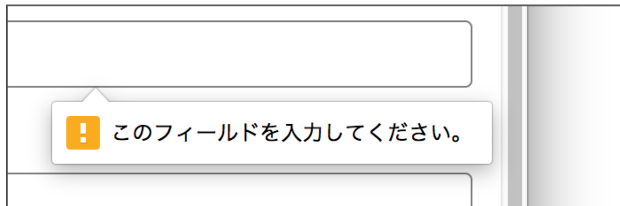

図 10.2　HTML5 フォームバリデーションのエラーメッセージ

次に、Errors Remover プラグインのアイコンをクリックしてから、「追加」ボタンをクリックします。すると予定項目の一覧ページに遷移して「予定を追加しました。」というフラッシュメッセージも出力されますが、実際には追加されていません。バリデーションが失敗したので、データベースへの書き込みが行われなかったのです。

■コラム: HTML5 フォームバリデーションの限界

最新の Google Chrome や Firefox を用いて開発を行っていると気づきにくいのですが、`input` 要素や `textarea` 要素に `required` 属性を付けてあっても、ブラウザがそれらの要素に対応するパラメータに空の文字列をセットして Rails アプリケーションに送ってくる可能性は十分にあります。

まず、ユーザーが HTML5 未対応のブラウザを使っている場合が考えられます。また、プログラミング知識のあるユーザーが Ruby とか Python でスクリプトを作ってアクセスするケースもあります。もちろん、私たちと同様に HTML5 Form validation errors remover のようなプラグインを使う人もいます。

HTML5 フォームバリデーションは、あくまでブラウザでのユーザビリティを高めるための機能であって、簡単に迂回できます。ブラウザでバリデーションしているからといって、サーバー側（Rails アプリケーション側）でバリデーションしなくてもよい、というわけではありません。

10.4 plan_items#create アクションの書き換え

それでは、plan_items コントローラ側の変更に進みます。予定項目の追加を行っている create アクションのコードをご覧ください。

```
def create
  PlanItem.create(plan_item_params)

  redirect_to :plan_items, notice: '予定を追加しました。'
end
```

モデルクラスの create メソッドはフォームから送られてきたパラメータを引数に取り、データベーステーブルに新たなレコードを挿入します。実は、挿入処理を行う前に暗黙のうちにバリデーションを行っていて、バリデーションが失敗した場合は、データベースへの書き込みをスキップします。

さて、この章の前半で学んだ valid? メソッドを用いると create アクションは次のように書き換えられます。

app/controllers/plan_items_controller.rb

```
   :
45      def create
46 -      PlanItem.create(plan_item_params)
46 +      @plan_item = PlanItem.new(plan_item_params)
47 +      if @plan_item.valid?
48 +        @plan_item.save!
49 +      end
50        redirect_to :plan_items, notice: '予定を追加しました。'
51      end
   :
```

save! メソッドについては、『初級②』第 5 章で学びました。モデルオブジェクトが保持している値をデータベースに書き込むメソッドです。このメソッドも create メソッドと同様に暗黙のうちにバリデーションを行いますが、create メソッドとは異なりバリデーションが失敗した場合は、例外を発生させます。ここでは、46 行目でバリデーションが成功していることが保証されているので、例外が発生する可能性はありません。

続いて、create アクションを次のように書き換えてください。

app/controllers/plan_items_controller.rb

```
   :
45      def create
46        @plan_item = PlanItem.new(plan_item_params)
47        if @plan_item.valid?
48          @plan_item.save!
49 +        redirect_to :plan_items, notice: '予定を追加しました。'
50        end
50 -      redirect_to :plan_items, notice: '予定を追加しました。'
51      end
   :
```

redirect_to メソッドを if 節の中に入れました。そして、さらに次の通り書き換えます。

app/controllers/plan_items_controller.rb

```
   :
45      def create
46        @plan_item = PlanItem.new(plan_item_params)
47        if @plan_item.valid?
48          @plan_item.save!
49          redirect_to :plan_items, notice: '予定を追加しました。'
50 +      else
51 +        render action: 'new'
52        end
53      end
   :
```

新たに else 節を加え、その中で render メソッドを用いて予定項目追加フォームを再び表示しています。

以上の変更の結果、PicoPlanner はバリデーションの成否を反映した自然な振る舞いをするようになります。改めてブラウザで予定項目追加フォームを開き、Errors Remover プラグインを有効にした上で何も入力せずに「追加」ボタンをクリックすると、同じフォームが再表示されます。ボタンが反応していないようにも見えますが、Rails サーバーのログを見れば Rails サーバーにフォームデータが送信されていることがわかります。他方、件名に「A」と記入して「追加」ボ

タンをクリックすると、予定が追加されます。

実は、 plan_items#create アクションはもう少し簡潔に書くことができます。次のように書き換えてください。

app/controllers/plan_items_controller.rb

```
   :
45     def create
46       @plan_item = PlanItem.new(plan_item_params)
47 -     if @plan_item.valid?
47 +     if @plan_item.save
48 -       @plan_item.save!
48         redirect_to :plan_items, notice: '予定を追加しました。'
49       else
50         render action: 'new'
51       end
52     end
   :
```

モデルオブジェクトの save メソッド（感嘆符がない点に注意）は、save! メソッドと同様にバリデーションを行ってからデータベースへの書き込みを行いますが、save! メソッドとは異なりバリデーションが失敗しても例外を発生させません。その代わりに戻り値として false を返します。

10.5　スタイルシート errors.scss の追加

さて、予定項目追加フォームの件名を空にしたまま「追加」ボタンをクリックしてフォームが再表示されたところで、Chrome のデベロッパーツールを用いて件名入力欄の HTML フラグメントを調べると次のようになっています。

```
<div class="form-group">
  <div class="field_with_errors">
    <label for="plan_item_name">件名</label>
  </div>
  <div class="field_with_errors">
    <input class="form-control" required="required"
      type="text" value="" name="plan_item[name]" id="plan_item_name">
  </div>
</div>
```

元のフォームにはなかった div 要素によって label 要素と input 要素が囲まれています。この div 要素は、バリデーションが失敗したパラメータを示すために Rails が付け加えたものです。その class 属性には field_with_errors という値がセットされています。

この事実を利用して、エラーの発生した入力欄に色を付けてみましょう。app/assets/stylesheets ディレクトリに新規ファイル errors.scss を次のような内容で作成してください。

app/assets/stylesheets/errors.scss (New)

```
div.field_with_errors {
  input.form-control[type="text"],
  textarea.form-control {
    border-color: red;
    background-color: pink;
  }
  label {
    color: red;
  }
}
```

そして、もう一度、件名を空にしたまま「追加」ボタンをクリックすると図 10.3 のように、「件名」ラベルの文字とテキスト入力欄の枠線が赤くなり、テキスト入力欄の内側はピンク色になります。

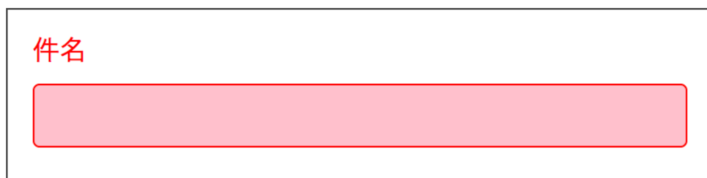

図 10.3　エラーの発生した入力欄に色を付ける

10.6　plan_items#update アクションの書き換え

最後に、plan_items#update アクションを書き換えて、この章を終えましょう。

app/controllers/plan_items_controller.rb

```
 :
54      def update
55 -       plan_item = PlanItem.find(params[:id])
55 +       @plan_item = PlanItem.find(params[:id])
56 -       plan_item.update_attributes(plan_item_params)
56 +       @plan_item.assign_attributes(plan_item_params)
57
58         redirect_to :plan_items, notice: '予定を更新しました。'
59      end
 :
```

まず、プライベート変数 plan_item をインスタンス変数@plan_item で置き換えています。バリデーションが失敗したとき、フォームを再表示するために必要な変更です。

次に、update_attributes メソッドを assign_attributes メソッドで置き換えています。このふたつのメソッドの機能はよく似ています。どちらも引数に与えられたパラメータを用いて PlanItem オブジェクトの各属性の値をセットします。ただし、前者の update_attributes メソッドは、さらにデータベースへの書き込みを行います。

続いて、plan_items#update アクションを次のように書き換えます。

app/controllers/plan_items_controller.rb

```
 :
54      def update
55        @plan_item = PlanItem.find(params[:id])
56        @plan_item.assign_attributes(plan_item_params)
57 -
58 -      redirect_to :plan_items, notice: '予定を更新しました。'
57 +      if @plan_item.save
58 +        redirect_to :plan_items, notice: '予定を更新しました。'
59 +      else
60 +        render action: 'edit'
61 +      end
62      end
 :
```

`if ... else ... end` による条件分岐を加えました。式 `@plan_item.save` は、`@plan_item` のバリデーションが成功すればデータベースを更新して `PlanItem` オブジェクトを返します。これは条件式では真とみなされるので、58 行目の `redirect_to` メソッドが呼びだされます。バリデーションが失敗した場合、`save` メソッドは `false` を返しますので、60 行目の `render` メソッドが呼び出されます。

動作確認のため、ブラウザで適当な予定項目の変更フォームを表示し、Errors Remover プラグインを有効にした上で件名欄を空にしてから「更新」ボタンをクリックしてみてください。フォームが再表示されて、件名欄が赤色で表示されっれば OK です。件名欄に半角スペースだけを入力した場合も同様のエラー表示になります。最後に、件名欄に「A」とだけ入力して「更新」ボタンをクリックし、予定項目を更新できることも確認してください。

第 11 章

Bootstrap フォームバリデーション

> この章では、Bootstrap フォームバリデーションを利用してエラー発生時のフォームのビジュアルデザインを変更する方法について学びます。また、モデルオブジェクトでバリデーションが失敗したときに作られる `Errors` オブジェクトについて学びます。このオブジェクトはどの属性でどのような種類のエラーが発生したのかを保持しており、フォームのカスタマイズをする際に重要な役割を果たします。

11.1 事前準備

前章（第 10 章）では、バリデーションが失敗したフォームの入力欄に色を付けて目立たせるため、`errors.scss` という SCSS ファイルを作成し、Rails が自動的に `label` 要素と `input` 要素の周りに配置する `div` 要素を利用しました。

しかし、本シリーズでは Bootstrap をベースとするビジュアルデザインを採用しているので、Bootstrap が提供しているスタイルシートを利用してエラー発生時のフォームのビジュアルデザインを変更することにします。

まず、前章で作った `errors.scss` を削除します。

```
$ rm app/assets/stylesheets/errors.scss
```

そして、`config/initializers` ディレクトリに新規ファイル `action_view.rb` を次の内容で作成します。

config/initializers/action_view.rb (New)

```
1  ActionView::Base.field_error_proc = -> (html_tag, instance) { html_tag }
```

このファイルを作成することにより、Rails はバリデーションが失敗した入力欄の label 要素や input 要素を div 要素で囲まなくなります。なお、config/initializers ディレクトリの内容を変更したときは、Rails サーバーの再起動が必要です。

> config/initializers ディレクトリの action_view.rb では、バリデーションが失敗した入力欄の label 要素や input 要素をどのように書き換えるかを設定しています。記号 -> についてはは第 7 章で触れました。無名関数を定義するための記号です。本来は、中括弧の内側（つまり無名関数の本体）で HTML フラグメントを変換して返すコードを記述しますが、ここでは変換せずにそのまま返しています。その結果、バリデーションが失敗しても入力欄が div 要素で囲まれなくなります。

11.2 Bootstrap フォームバリデーション

Bootstrap がフォームバリデーションのために用意しているスタイルシートの基本的な使用例は次の通りです。

```
<div class="form-group has-danger">
  <label class="form-control-label" for="user_name">名前</label>
  <input type="text" class="form-control" id="user_name">
</div>
```

個々の入力欄を囲む div 要素に form-group クラスを指定するのは、Bootstrap での約束事です（『初級③』第 1 章）。これに has-danger クラスを加えると、この div 要素の中のラベルとテキスト入力欄の枠線が赤く表示されるようになります。ただし、label 要素には form-control-label クラスを指定する必要があります。

上記の HTML フラグメントは図 11.1 のようにブラウザで表示されます。モノクロディスプレイや書籍で見ている方には分かりにくいですが、「名前」とい

うラベルとテキスト入力欄の枠線が赤く表示されています。

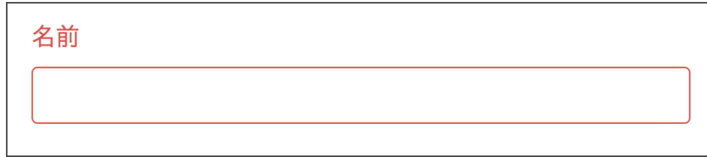

図 11.1　Bootstrap フォームバリデーションの使用例

では、PicoPlanner に Bootstrap フォームバリデーションのスタイルシートを適用してみましょう。app/views/plan_items ディレクトリの _fields.html.erb を次のように書き換えてください。

app/views/plan_items/_fields.html.erb

```
1 - <div class='form-group'>
1 + <div class='form-group has-danger'>
2 -   <%= f.label :name, '件名' %>
2 +   <%= f.label :name, '件名', class: 'form-control-label' %>
3     <%= f.text_field :name, class: 'form-control', required: true %>
4   </div>
:
```

ブラウザで予定項目の追加・編集フォームを開くと、図 11.2 のようにラベルと枠線が赤色で表示されます。

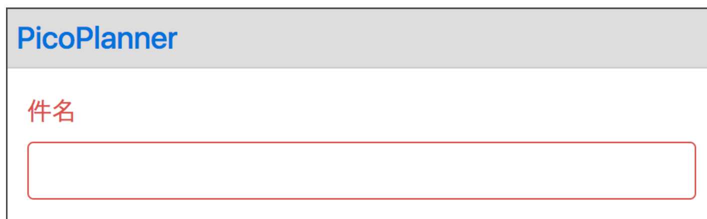

図 11.2　Bootstrap フォームバリデーションを適用

Bootstrap はさらに面白いスタイルシートを用意しています。同じファイルを次のように書き換えてください。

app/views/plan_items/_fields.html.erb

```
1     <div class='form-group has-danger'>
2       <%= f.label :name, '件名', class: 'form-control-label' %>
3 -     <%= f.text_field :name, class: 'form-control', required: true %>
3 +     <%= f.text_field :name,
4 +       class: 'form-control form-control-danger', required: true %>
5     </div>
:
```

input 要素の class 属性に form-control-danger を加えると、図 11.3 のようにテキスト入力欄の右端に赤色のバツ印が表示されます。こうすれば、バリデーションが失敗したことがより明瞭になります。

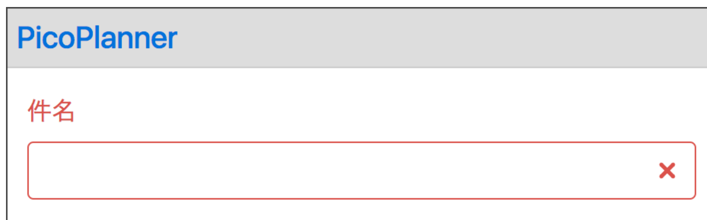

図 11.3　テキスト入力欄の右端に赤色のバツ印を表示

11.3　Errors オブジェクト

さて、現時点での予定項目追加・編集フォームは、バリデーションの結果にかかわらず「件名」入力欄がエラー発生時のスタイル（ラベルや枠線が赤色）で表示されています。バリデーションの結果に応じて入力欄を囲む div 要素の class 属性を切り替える必要があります。

件名（name 属性）でバリデーションが失敗したかどうかは、次の式で調べることができます。

```
f.object.errors.include?(:name)
```

変数 f はフォームビルダーです。それの object メソッドは、フォームで編集

対象となっているモデルオブジェクトを返します。そして、モデルオブジェクトの errors メソッドは、ActiveModel::Errors クラスのインスタンスを返します。

このオブジェクトはどの属性でどのような種類のエラーが発生したのかを保持しています。今後は、これを **Errors** オブジェクトと呼ぶことにします。

11.4 ヘルパーメソッド content_tag

前節で学んだ知識を用いれば、div 要素の class 属性を動的に切り替えることが可能になります。_fields.html.erb の 1 行目にある has-danger を

```
<%= 'has-danger' if f.object.errors.include?(:name) %>
```

で置き換えればいいのです。

しかし、実際にやってみると、ERB テンプレートのコードが少々不格好なものになります。

```
<div class='form-group
  <%= 'has-danger' if f.object.errors.include?(:name) %>'>
  <%= f.label :name, '件名', class: 'form-control-label' %>
  <%= f.text_field :name,
    class: 'form-control form-control-danger', required: true %>
</div>
```

HTML タグの内側で ERB のタグ `<%= ... %>` を用いているため、角括弧（`< ... >`）とシングルクオート（`' ... '`）が入れ子になってしまいます。ソースコードが読みにくくなるので、なるべく避けた方がよいと筆者は考えます。

こんなときに役立つのが、ヘルパーメソッド content_tag です。次の例をご覧ください。

```
<%= content_tag :p do %>
Hello, world!
<% end %>
```

これは、次の HTML フラグメントを生成します。

```
<p>
Hello, world!
</p>
```

ヘルパーメソッド content_tag は第 1 引数に指定されたシンボルと同じ名前の HTML 要素を生成します。ブロックの戻り値が要素の中身となります。

ヘルパーメソッド content_tag の第 2 引数に指定されたハッシュは HTML 要素の属性として使われます。例えば、

```
<%= content_tag :p, class: 'message' do %>
Hello, world!
<% end %>
```

は、次の HTML フラグメントを生成します。

```
<p class='message'>
Hello, world!
</p>
```

ヘルパーメソッド content_tag を利用すると、_fields.html.erb は次のように書き換えることができます。

app/views/plan_items/_fields.html.erb

```
1 - <div class='form-group has-danger'>
1 + <%= content_tag :div, class: 'form-group has-danger' do %>
2     <%= f.label :name, '件名', class: 'form-control-label' %>
3     <%= f.text_field :name,
4       class: 'form-control form-control-danger', required: true %>
5 - </div>
5 + <% end %>
:
```

ただし、この段階では div 要素の class 属性は固定されています。これを動的に切り替えるには、まず ERB テンプレートの冒頭に 4 行のコードを加えます。

app/views/plan_items/_fields.html.erb

```
1 + <%
2 +   html_classes = %w(form-group)
3 +   html_classes << 'has-danger' if f.object.errors.include?(:name)
4 + %>
5   <%= content_tag :div, class: 'form-group has-danger' do %>
:
```

2行目で、変数 html_classes に "form-group" という文字列を要素として持つ配列をセットしています。%w(...) は文字列の配列を作る記号です。件名（name 属性）でバリデーションが失敗している場合には、この配列に 'has-danger' を加えます。

　そして、5 行目を次のように書き換えます。

app/views/plan_items/_fields.html.erb

```
1   <%
2     html_classes = %w(form-group)
3     html_classes << 'has-danger' if f.object.errors.include?(:name)
4   %>
5 - <%= content_tag :div, class: 'form-group has-danger' do %>
5 + <%= content_tag :div, class: html_classes.join(' ') do %>
:
```

文字列の配列を join メソッドで連結して HTML 要素の class 属性に指定しています。同じようなテクニックを、本巻の第 4 章でも使用しましたね。

　動作確認をしましょう。ブラウザで予定項目の追加フォームを開き、件名を空にしたまま送信して、図 11.4 のように件名フィールドのラベルおよび入力欄の枠が赤色で表示され、入力欄の右端にバツ印が表示されれば OK です。

PicoPlanner

件名

説明

終日

開始日時

2017-07-11

10:00

終了日時

2017-07-11

11:00

追加

図 11.4　件名フィールドでバリデーションが失敗した場合の表示