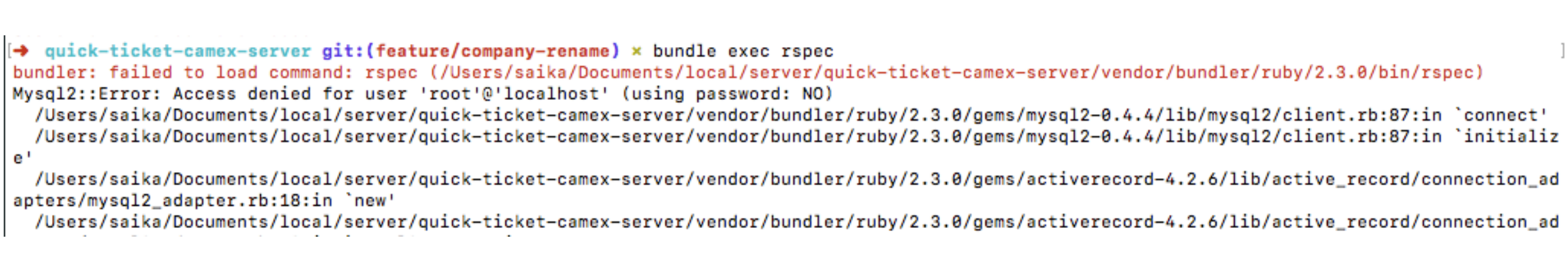
環境設定

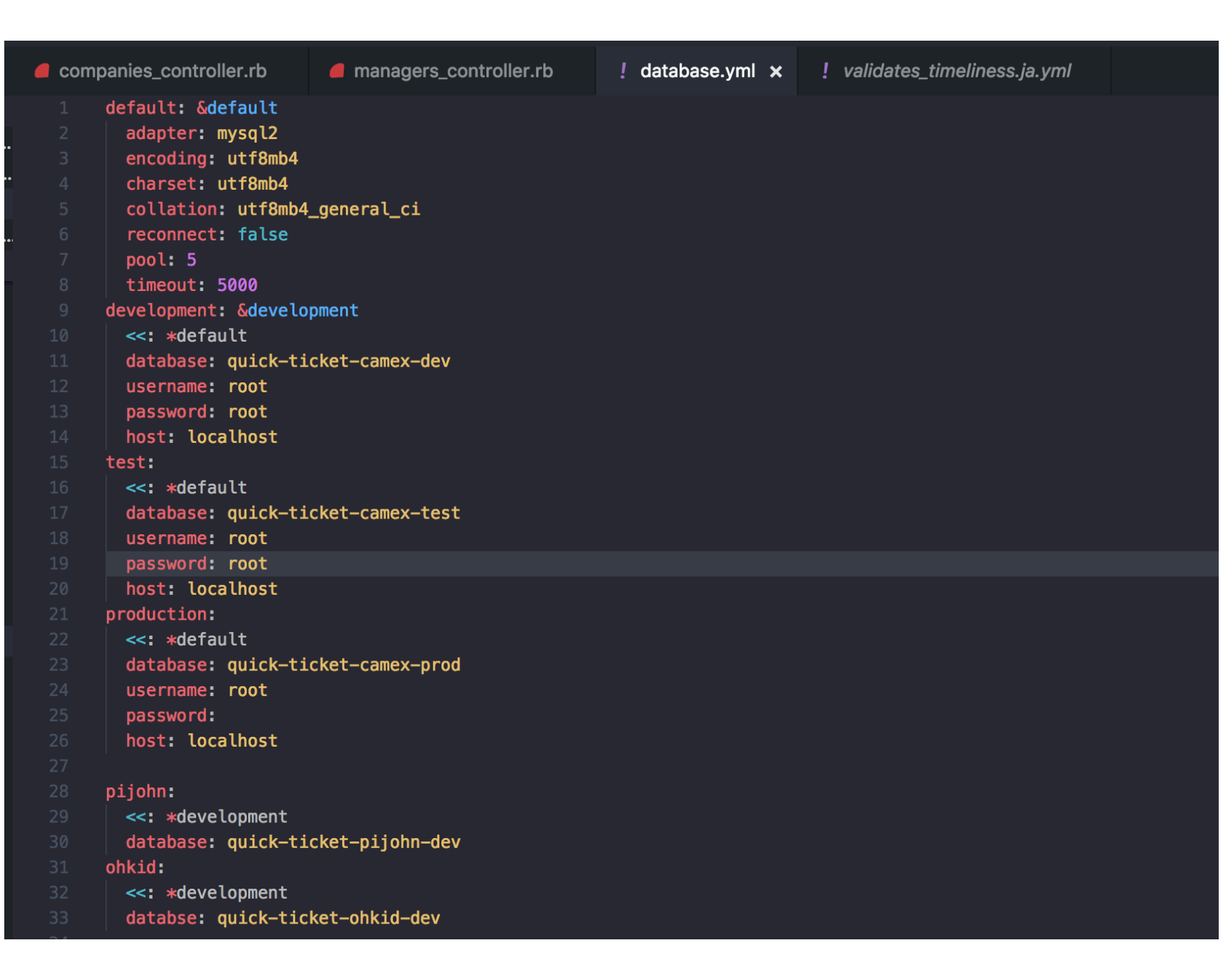
https://qiita.com/hitomi_/items/fc9e9a3aa68039d68776

想要執行公司之前的RSpec的文件 老是報錯：



```
[➜  quick-ticket-camex-server git:(feature/company-rename) ✗ bundle exec rspec
bundler: failed to load command: rspec (/Users/saika/Documents/local/server/quick-ticket-camex-server/vendor/bundler/ruby/2.3.0/bin/rspec)
Mysql2::Error: Access denied for user 'root'@'localhost' (using password: NO)
  /Users/saika/Documents/local/server/quick-ticket-camex-server/vendor/bundler/ruby/2.3.0/gems/mysql2-0.4.4/lib/mysql2/client.rb:87:in `connect'
  /Users/saika/Documents/local/server/quick-ticket-camex-server/vendor/bundler/ruby/2.3.0/gems/mysql2-0.4.4/lib/mysql2/client.rb:87:in `initializ
e'
  /Users/saika/Documents/local/server/quick-ticket-camex-server/vendor/bundler/ruby/2.3.0/gems/activerecord-4.2.6/lib/active_record/connection_ad
apters/mysql2_adapter.rb:18:in `new'
  /Users/saika/Documents/local/server/quick-ticket-camex-server/vendor/bundler/ruby/2.3.0/gems/activerecord-4.2.6/lib/active_record/connection_ad
```

想了想應該是yml環境設定裡面沒有寫密碼的原因：



```
1  default: &default
2    adapter: mysql2
3    encoding: utf8mb4
4    charset: utf8mb4
5    collation: utf8mb4_general_ci
6    reconnect: false
7    pool: 5
8    timeout: 5000
9  development: &development
10   <<: *default
11   database: quick-ticket-camex-dev
12   username: root
13   password: root
14   host: localhost
15 test:
16   <<: *default
17   database: quick-ticket-camex-test
18   username: root
19   password: root
20   host: localhost
21 production:
22   <<: *default
23   database: quick-ticket-camex-prod
24   username: root
25   password:
26   host: localhost
27
28 pijohn:
29   <<: *development
30   database: quick-ticket-pijohn-dev
31 ohkid:
32   <<: *development
33   databse: quick-ticket-ohkid-dev
```

在19行加了test的password就可以運行了。

```
[➜  quick-ticket-camex-server git:(feature/company-rename) ✗ rspec spec/requests/ohkid/api/v1/companies/update_spec.rb
....

Finished in 0.53541 seconds (files took 5.97 seconds to load)
4 examples, 0 failures

Coverage report generated for RSpec to /Users/saika/Documents/local/server/quick-ticket-camex-server/coverage. 67339 / 142960 LOC (47.1%) covered
.
➜  quick-ticket-camex-server git:(feature/company-rename) ✗ ▌
```

構文

https://qiita.com/jnchito/items/42193d066bd61c740612

include_context:
let:
shard_examples_for :
context :
it_behaves_like :

---

describe / it / expect の役割を理解する

一番単純なRSpecのテストはこんな記述になります。

```
describe '四則演算' do
  it '1 + 1 は 2 になること' do
    expect(1 + 1).to eq 2
  end
end
```

`describe` はテストのグループ化を宣言します。ここでは「四則演算に関するテストを書くよー」と宣言しています。

`it` はテストを example という単位にまとめる役割をします。
`it do ... end` の中のエクスペクテーション（期待値と実際の値の比較）がすべてパスすれば、その example はパスしたことになります。

`expect(X).to eq Y` で記述するのがエクスペクテーションです。
expect には「期待する」という意味があるので、 `expect(X).to eq Y` は「XがYに等しくなることを期待する」と読めます。
よって、 `expect(1 + 1).to eq 2` は「1 + 1 が 2 になることを期待する」テストになります。

`describe` の中には複数の example（`it do ... end`）が書けます。

```
describe '四則演算' do
  it '1 + 1 は 2 になること' do
    expect(1 + 1).to eq 2
  end
  it '10 - 1 は 9 になること' do
    expect(10 - 1).to eq 9
  end
end
```

テストを実行すると、以下のような結果が表示されます。

```
2 examples, 0 failures, 2 passed

Finished in 0.002592 seconds

Process finished with exit code 0
```

context と before でもう少し便利に

ここではこんなクラスをテストします。

```
class User
  def initialize(name:, age:)
    @name = name
    @age = age
  end
  def greet
    if @age <= 12
      "ぼくは#{@name}だよ。"
    else
      "僕は#{@name}です。"
    end
  end
end
```

「初めの一歩」で学んだ知識を使うと次のようなテストが書けます。

```
describe User do
  describe '#greet' do
    it '12歳以下の場合、ひらがなで答えること' do
      user = User.new(name: 'たろう', age: 12)
      expect(user.greet).to eq 'ぼくはたろうだよ。'
    end
    it '13歳以上の場合、漢字で答えること' do
      user = User.new(name: 'たろう', age: 13)
      expect(user.greet).to eq '僕はたろうです。'
    end
  end
end
```

`describe` には `describe User` のように、文字列ではなくクラスを渡すこともできます。
また、「インスタンスメソッドの greet メソッドをテストしますよ」という意味で `describe '#greet'` のように書くこともよくあります。

context で条件別にグループ化する

RSpecには describe 以外にも context という機能でテストをグループ化することもできます。
どちらも機能的には同じですが、 context は条件を分けたりするときに使うことが多いです。
ちなみに、contextは日本語で「文脈」や「状況」の意味になります。

ここでは「12歳以下の場合」と「13歳以上の場合」という二つの条件にグループ分けしてみました。

```
describe User do
  describe '#greet' do
    context '12歳以下の場合' do
      it 'ひらがなで答えること' do
        user = User.new(name: 'たろう', age: 12)
        expect(user.greet).to eq 'ぼくはたろうだよ。'
      end
    end
    context '13歳以上の場合' do
      it '漢字で答えること' do
        user = User.new(name: 'たろう', age: 13)
        expect(user.greet).to eq '僕はたろうです。'
      end
    end
  end
end
```

`describe` と同様、 `context` で適切にグループ化すると、「この context ブロックはこういう条件の場合をテストしてるんだな」と読み手がテストコードを理解しやすくなります。

before で共通の前準備をする

`before do ... end` で囲まれた部分は example の実行前に毎回呼ばれます。`before` ブロックの中では、テストを実行する前の共通処理やデータのセットアップ等を行うことが多いです。
少々強引ですが、サンプルコードでは `name: 'たろう'` が重複しているので、DRYにしてみましょう。

```
describe User do
  describe '#greet' do
    before do
      @params = { name: 'たろう' }
    end
    context '12歳以下の場合' do
      it 'ひらがなで答えること' do
        user = User.new(@params.merge(age: 12))
        expect(user.greet).to eq 'ぼくはたろうだよ。'
      end
    end
    context '13歳以上の場合' do
      it '漢字で答えること' do
        user = User.new(@params.merge(age: 13))
        expect(user.greet).to eq '僕はたろうです。'
      end
    end
  end
end
```

上の例を見るとわかるように、 ローカル変数ではなく、インスタンス変数にデータをセットしています。
これは `before` ブロックと `it` ブロックの中では変数のスコープが異なるためです。

ネストした describe や context の中で before を使う

`before` は `describe` や `context` ごとに用意することができます。
`describe` や `context` がネストしている場合は、親子関係に応じて `before` が順番に呼ばれます。
先ほどのコード例を次のように変えてみましょう。

```
describe User do
  describe '#greet' do
    before do
      @params = { name: 'たろう' }
    end
    context '12歳以下の場合' do
      before do
        @params.merge!(age: 12)
      end
      it 'ひらがなで答えること' do
        user = User.new(@params)
        expect(user.greet).to eq 'ぼくはたろうだよ。'
      end
    end
    context '13歳以上の場合' do
      before do
        @params.merge!(age: 13)
      end
      it '漢字で答えること' do
        user = User.new(@params)
        expect(user.greet).to eq '僕はたろうです。'
      end
    end
  end
end
```

インスタンス変数のかわりに let を使う

先ほど出てきたコード例では、インスタンス変数の @params を使っていました。
しかし、RSpecではこのインスタンス変数を let という機能で置き換えることができます。
実際に let で置き換えたコードを見てみましょう。

```
describe User do
  describe '#greet' do
    let(:params) { { name: 'たろう' } }
    context '12歳以下の場合' do
      before do
        params.merge!(age: 12)
      end
      it 'ひらがなで答えること' do
        user = User.new(params)
        expect(user.greet).to eq 'ぼくはたろうだよ。'
      end
    end
    context '13歳以上の場合' do
      before do
        params.merge!(age: 13)
      end
      it '漢字で答えること' do
        user = User.new(params)
        expect(user.greet).to eq '僕はたろうです。'
      end
    end
  end
end
```

`let(:foo) { ... }` のように書くと、 `{ ... }` の中の値が `foo` として参照できる、というのが `let`の基本的な使い方です。ただ、上の例では `{ { name: 'たろう' } }` と、 `{ }` が2回出てくるのでややこしくなってます。外側の `{ }` はRubyのブロックで、 内側の `{ }` はハッシュリテラルです。
わかりづらい方は、こんなふうに書くとイメージが付きやすいかもしれません。

```
# let(:params) { { name: 'たろう' } } と同じ意味のコード
let(:params) do
  hash = {}
  hash[:name] = 'たろう'
  hash
end
```

user を let にする

インスタンス変数だけでなく、ローカル変数を `let` で置き換えるのもアリです。
`user` も `let` で置き換えてみましょう。

```
describe User do
  describe '#greet' do
    let(:user) { User.new(params) }
    let(:params) { { name: 'たろう' } }
    context '12歳以下の場合' do
      before do
        params.merge!(age: 12)
      end
      it 'ひらがなで答えること' do
        expect(user.greet).to eq 'ぼくはたろうだよ。'
      end
    end
    context '13歳以上の場合' do
      before do
        params.merge!(age: 13)
      end
      it '漢字で答えること' do
        expect(user.greet).to eq '僕はたろうです。'
      end
    end
  end
end
```

重複していた `user = User.new(params)` の部分を共通化することができました。

let のメリットを活かして age も let で置き換える

上のコードでは `before` ブロックの中で `params.merge!(age: 12)` みたいなコードを書いているのがあまりクールじゃありません。
どうせなら、ここも `let` で置き換えてスッキリさせましょう。

```
describe User do
  describe '#greet' do
    let(:user) { User.new(params) }
    let(:params) { { name: 'たろう', age: age } }
    context '12歳以下の場合' do
      let(:age) { 12 }
      it 'ひらがなで答えること' do
        expect(user.greet).to eq 'ぼくはたろうだよ。'
      end
    end
    context '13歳以上の場合' do
      let(:age) { 13 }
      it '漢字で答えること' do
        expect(user.greet).to eq '僕はたろうです。'
      end
    end
  end
end
```

`let` は「`before` + インスタンス変数」を使うときとは異なり、 **遅延評価される** という特徴があります。

すなわち、 `let` は必要になる瞬間まで呼び出されません。

上のコード例だと、こんな順番で呼び出されます。

1. `expect(user.greet).to` が呼ばれる => `user` って何だ？
2. `let(:user) { User.new(params) }` が呼ばれる => `params` って何だ？
3. `let(:params) { { name: 'たろう', age: age } }` が呼ばれる => `age` って何だ？
4. `let(:age) { 12 }` （または13） が呼ばれる
5. 結果として `expect(User.new(name: 'たろう', age: 12).greet).to` を呼んだことになる

subject を使ってテスト対象のオブジェクトを1箇所にまとめる

テスト対象のオブジェクト（またはメソッドの実行結果）が明確に一つに決まっている場合は、 `subject` という機能を使ってテストコードをDRYにすることができます。
たとえば、先ほどのコード例ではどちらのexampleも `user.greet` の実行結果をテストしています。
そこで、 `user.greet` を `subject` に引き上げて、 example の中から消してみましょう。

```
describe User do
  describe '#greet' do
    let(:user) { User.new(params) }
    let(:params) { { name: 'たろう', age: age } }
    subject { user.greet }
    context '12歳以下の場合' do
      let(:age) { 12 }
      it 'ひらがなで答えること' do
        is_expected.to eq 'ぼくはたろうだよ。'
      end
    end
    context '13歳以上の場合' do
      let(:age) { 13 }
      it '漢字で答えること' do
        is_expected.to eq '僕はたろうです。'
      end
    end
  end
end
```

`subject { user.greet }` を宣言したので、今まで `expect(user.greet).to eq 'ぼくはたろうだよ。'`と書いていた部分が is_expected.to eq 'ぼくはたろうだよ' に変わりました。

さらに、 `it` に渡す文字列（'ひらがなで答えること' など）を省略してみます。

```
describe User do
  describe '#greet' do
    let(:user) { User.new(params) }
    let(:params) { { name: 'たろう', age: age } }
    subject { user.greet }
    context '12歳以下の場合' do
      let(:age) { 12 }
      it { is_expected.to eq 'ぼくはたろうだよ。' }
    end
    context '13歳以上の場合' do
      let(:age) { 13 }
      it { is_expected.to eq '僕はたろうです。' }
    end
  end
end
```

`it { is_expected.to eq 'ぼくはたろうだよ。' }` は "it is expected to eq 'ぼくはたろうだよ。'" と、自然な英文っぽく読むことができます。

ちなみに subject は日本語で「主語」や「対象」という意味があります。
こんな書き方はできませんが、 `user.greet { is_expected.to eq 'ぼくはたろうだよ。' }` と考えると、 「subject = テストの主語」と解釈することもできそうです。

RSpecの高度な機能

example の再利用: shared_examples と it_behaves_like

上で作ったテストコードに、もうちょっとテストパターンを増やしてみましょう。
12歳と13歳だけでなく、もっと小さい子ども（0歳）や、もっと大きな大人（100歳）にもあいさつしてもらいます。

```
describe User do
  describe '#greet' do
    let(:user) { User.new(name: 'たろう', age: age) }
    subject { user.greet }

    context '0歳の場合' do
      let(:age) { 0 }
      it { is_expected.to eq 'ぼくはたろうだよ。' }
    end
    context '12歳の場合' do
      let(:age) { 12 }
      it { is_expected.to eq 'ぼくはたろうだよ。' }
    end

    context '13歳の場合' do
      let(:age) { 13 }
      it { is_expected.to eq '僕はたろうです。' }
    end
    context '100歳の場合' do
      let(:age) { 100 }
      it { is_expected.to eq '僕はたろうです。' }
    end
  end
end
```

見てもらえばわかると思いますが、同じexampleが2回ずつ登場しています。
こういう場合は、 `shared_examples` と `it_behaves_like` という機能を使うと、example を再利用することができます。

```
describe User do
  describe '#greet' do
    let(:user) { User.new(name: 'たろう', age: age) }
    subject { user.greet }

    shared_examples '子どものあいさつ' do
      it { is_expected.to eq 'ぼくはたろうだよ。' }
    end
    context '0歳の場合' do
      let(:age) { 0 }
      it_behaves_like '子どものあいさつ'
    end
    context '12歳の場合' do
      let(:age) { 12 }
      it_behaves_like '子どものあいさつ'
    end

    shared_examples '大人のあいさつ' do
      it { is_expected.to eq '僕はたろうです。' }
    end
    context '13歳の場合' do
      let(:age) { 13 }
      it_behaves_like '大人のあいさつ'
    end
    context '100歳の場合' do
      let(:age) { 100 }
      it_behaves_like '大人のあいさつ'
    end
  end
end
```

`shared_examples 'foo' do ... end` で再利用したいexampleを定義し、 `it_behaves_like 'foo'`で定義したexampleを呼び出すイメージです。
ちなみに `shared_examples` と `it_behaves_like` は日本語にするとそれぞれ、「共有されているexample」と「〜のように振る舞うこと」というふうに訳せます。

context の再利用: shared_context と include_context

`User` クラスに新しいメソッド、 `child?` を追加してみます。

```
class User
  def initialize(name:, age:)
    @name = name
    @age = age
  end
  def greet
    if child?
      "ぼくは#{@name}だよ。"
    else
      "僕は#{@name}です。"
    end
  end
  def child?
    @age <= 12
  end
end
```

せっかくなので `greet` メソッドだけでなく、 `child?` メソッドもテストしておきましょう。

```
describe User do
  describe '#greet' do
    let(:user) { User.new(name: 'たろう', age: age) }
    subject { user.greet }
    context '12歳以下の場合' do
      let(:age) { 12 }
      it { is_expected.to eq 'ぼくはたろうだよ。' }
    end
    context '13歳以上の場合' do
      let(:age) { 13 }
      it { is_expected.to eq '僕はたろうです。' }
    end
  end

  describe '#child?' do
    let(:user) { User.new(name: 'たろう', age: age) }
    subject { user.child? }
    context '12歳以下の場合' do
      let(:age) { 12 }
      it { is_expected.to eq true }
    end
    context '13歳以上の場合' do
      let(:age) { 13 }
      it { is_expected.to eq false }
    end
  end
end
```

テストコードを書いてみると、どちらのテストも「12歳以下の場合」と「13歳以上の場合」で、同じ context が登場しています。
こういう場合は `shared_context` と `include_context` を使うと、 context を再利用することができます。

```
describe User do
  let(:user) { User.new(name: 'たろう', age: age) }
  shared_context '12歳の場合' do
    let(:age) { 12 }
  end
  shared_context '13歳の場合' do
    let(:age) { 13 }
  end

  describe '#greet' do
    subject { user.greet }
    context '12歳以下の場合' do
      include_context '12歳の場合'
      it { is_expected.to eq 'ぼくはたろうだよ。' }
    end
    context '13歳以上の場合' do
      include_context '13歳の場合'
      it { is_expected.to eq '僕はたろうです。' }
    end
  end

  describe '#child?' do
    subject { user.child? }
    context '12歳以下の場合' do
      include_context '12歳の場合'
      it { is_expected.to eq true }
    end
    context '13歳以上の場合' do
      include_context '13歳の場合'
      it { is_expected.to eq false }
    end
  end
end
```