Course - Laravel Framework

Collection

Class Illuminate\Support\Collection sẽ cung cấp một wrapper có các mục được bố trí tuần tự, thuận tiện để làm việc với các mảng dữ liệu.

Tags: laravel collection, laravel

Giới thiệu

Lớp `Illuminate\Support\Collection` cung cấp một wrapper trong đó chứa các mục được bố trí tuần tự, thuận tiện để làm việc với các mảng dữ liệu. Ví dụ, hãy xem đoạn mã sau. Chúng ta sẽ sử dụng hàm `collect` để tạo một phiên bản bộ sưu tập mới từ mảng, chạy hàm `strtoupper` trên từng phần tử và sau đó xóa tất cả các phần tử trống:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
  return strtoupper($name);
})->reject(function ($name) {
  return empty($name);
});
```

Như bạn có thể thấy, Collection cho phép bạn gọi một chuỗi các phương thức của nó để thực hiện việc bố trí và thu gọn mảng bên trong một cách nhanh chóng. Nói chung, các tập hợp là immutable, có nghĩa là mọi phương thức `Collection` đều trả về một đối tượng `Collection` hoàn toàn mới.

Tạo collection

Như đã đề cập ở trên, hàm `collect` trả về một đối tượng mới `Illuminate\Support\Collection` cho mảng đã cho. Vì vậy, việc tạo một collection sẽ đơn giản như sau:

```
$collection = collect([1, 2, 3]);
```

> Kết quả của các truy vấn Eloquent luôn được trả về dưới dạng các đối tượng `Collection`.

Mở rộng Collection

Các collection là "macroable", cho phép bạn thêm các phương thức bổ sung vào class `Collection` tại thời điểm chạy. Phương thức `macro` của class `Illuminate\Support\Collection` chấp nhận một hàm nặc danh sẽ được thực thi khi macro của bạn được gọi. Hàm nặc danh của `macro` có thể truy cập các phương thức khác của collection thông qua biến `$this`, giống như thể nó là một phương thức thực của class collection. Ví dụ, đoạn mã sau thêm phương thức `toUpper` vào class `Collection`:

```php
use Illuminate\Support\Collection;
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
  return $this->map(function ($value) {
    return Str::upper($value);
  });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

Thông thường, bạn nên khai báo macro thu thập trong phương thức `boot` của service provider.

Đối số vĩ mô

Nếu cần, bạn có thể xác định các macro chấp nhận các đối số bổ sung:

```php
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\Lang;

Collection::macro('toLocale', function ($locale) {
  return $this->map(function ($value) use ($locale) {
    return Lang::get($value, [], $locale);
  });
});

$collection = collect(['first', 'second']);

$translated = $collection->toLocale('es');
```

Các phương thức sẵn có

Đối với phần lớn tài liệu collection còn lại, chúng ta sẽ thảo luận về từng phương thức có sẵn trên class `Collection`. Hãy nhớ rằng, tất cả các phương thức này có thể được gọi xâu chuỗi để xử lý nhanh mảng bên trong. Hơn nữa, hầu hết mọi phương thức đều trả về một đối tượng `Collection` mới, cho phép bạn giữ lại bản sao gốc của bộ sưu tập khi cần thiết:

Phương thức `all`

Phương thức `all` trả về mảng cơ bản được bao bọc bởi collection:

```
collect([1, 2, 3])->all();

// [1, 2, 3]
```

Phương thức `average`

Bí danh của phương thức `avg`.

Phương thức `avg`

Phương thức `avg` trả về giá trị trung bình của một khóa nhất định:

```
$average = collect([
  ['foo' => 10],
  ['foo' => 10],
  ['foo' => 20],
  ['foo' => 40]
])->avg('foo');

// 20

$average = collect([1, 1, 2, 4])->avg();

// 2
```

Phương thức `chunk`

Phương thức `chunk` chia collection thành nhiều collection nhỏ hơn với kích thước nhất định:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->all();

// [[1, 2, 3, 4], [5, 6, 7]]
```

Phương pháp này đặc biệt hữu ích trong các template khi làm việc với layout lưới như Bootstrap. Ví dụ: hãy tưởng tượng bạn có một collection các Eloquent model mà bạn muốn hiển thị trong một lưới:

```
@foreach ($products->chunk(3) as $chunk)
  <div class="row">
    @foreach ($chunk as $product)
      <div class="col-xs-4">{{ $product->name }}</div>
    @endforeach
  </div>
@endforeach
```

Phương thức `chunkWhile`

Phương thức `chunkWhile` chia collection thành nhiều collection nhỏ hơn dựa trên đánh giá của callback đã cho. Biến `$chunk` được truyền vào hàm nặc danh có thể được sử dụng để kiểm tra phần tử trước đó:

```
$collection = collect(str_split('AABBCCCD'));

$chunks = $collection->chunkWhile(function ($value, $key, $chunk) {
  return $value === $chunk->last();
});
```

```
$chunks->all();

// [['A', 'A'], ['B', 'B'], ['C', 'C', 'C'], ['D']]
```

Phương thức `collapse`

Phương thức `collapse` thu gọn collection các mảng thành một collection phẳng, và duy nhất:

```
$collection = collect([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Phương thức `collect`

Phương thức `collect` trả về một đối tượng `Collection` mới với các mục hiện có trong collection:

```
$collectionA = collect([1, 2, 3]);

$collectionB = $collectionA->collect();

$collectionB->all();

// [1, 2, 3]
```

Phương thức `collect` này chủ yếu hữu ích để chuyển đổi các collection lười thành các đối tượng `Collection` tiêu chuẩn:

```
$lazyCollection = LazyCollection::make(function () {
  yield 1;
  yield 2;
  yield 3;
});

$collection = $lazyCollection->collect();

get_class($collection);

// 'Illuminate\Support\Collection'

$collection->all();

// [1, 2, 3]
```

> Phương thức `collect` này đặc biệt hữu ích khi bạn có một đối tượng `Enumerable` và một đối tượng collection tiêu chuẩn. Vì `collect()` là một phần của contract `Enumerable` nên bạn có thể yên tâm sử dụng nó để lấy một đối tượng Collection.

Phương thức `combine`

Phương thức combine sẽ kết hợp các giá trị của collection, dưới dạng các khóa, với các giá trị của mảng hoặc collection khác.

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

Phương thức `concat`

Phương thức concat sẽ nối các giá trị mảng hoặc collection đã cho vào phần cuối của tập hợp khác:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

Phương thức `contains`

Phương thức `contains` xác định xem collection có chứa một mục nhất định nào đó hay không. Bạn có thể truyền một hàm nặc danh vào phương thức `contains` để xác định xem một phần tử có tồn tại trong tập hợp khớp với mục kiểm tra thật sự đã cho hay không:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function ($value, $key) {
  return $value > 5;
});

// false
```

Ngoài ra, bạn có thể truyền một chuỗi vào phương thức `contains` để xác định xem collection có chứa một giá trị mục nhất định hay không:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true
```

```php
$collection->contains('New York');

// false
```

Bạn cũng có thể truyền một cặp khóa/giá trị vào phương thức `contains`, điều này sẽ xác định xem cặp đã cho có tồn tại trong collection hay không:

```php
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

Phương thức `contains` sử dụng so sánh "lỏng lẻo" khi kiểm tra giá trị mục, nghĩa là một chuỗi có giá trị nguyên sẽ được coi là bằng một số nguyên có cùng giá trị. Sử dụng phương thức `containsStrict` để lọc bằng cách so sánh "nghiêm ngặt".

Đối với nghịch đảo của `contains`, hãy xem phương thức `doesntContain`.

Phương thức `containsStrict`

Phương thức này có cùng cách thức hoạt động với phương thức `contains`; tuy nhiên, tất cả các giá trị được so sánh bằng cách sử dụng so sánh "nghiêm ngặt".

> Hoạt động của phương thức này được sửa đổi khi sử dụng Eloquent `Collections`.

Phương thức `count`

Phương thức `count` trả về tổng số mục trong collection:

```php
$collection = collect([1, 2, 3, 4]);

$collection->count();
```

```
// 4
```

Phương thức `countBy`

Phương thức `countBy` đếm số lần xuất hiện của các giá trị trong bộ sưu tập. Mặc định, phương thức sẽ đếm số lần xuất hiện của tất cả phần tử, nhưng nó cũng cho phép bạn đếm các "loại" phần tử nhất định trong collection:

```
$collection = collect([1, 2, 2, 2, 3]);

$counted = $collection->countBy();

$counted->all();

// [1 => 1, 2 => 3, 3 => 1]
```

Bạn truyền một hàm nặc danh vào phương thức `countBy` để đếm tất cả các mục theo một giá trị được điều chỉnh:

```
$collection = collect(['alice@gmail.com', 'bob@yahoo.com', 'carlos@gmail.com']);

$counted = $collection->countBy(function ($email) {
  return substr(strrchr($email, "@"), 1);
});

$counted->all();

// ['gmail.com' => 2, 'yahoo.com' => 1]
```

Phương thức `crossJoin`

Phương thức `crossJoin` kết hợp chéo các giá trị của collection giữa các mảng hoặc collection đã cho, trả về tích Descartes với tất cả các hoán vị có thể có:

```
$collection = collect([1, 2]);
```

```php
$matrix = $collection->crossJoin(['a', 'b']);

$matrix->all();

/*
  [
    [1, 'a'],
    [1, 'b'],
    [2, 'a'],
    [2, 'b'],
  ]
*/

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/*
  [
    [1, 'a', 'I'],
    [1, 'a', 'II'],
    [1, 'b', 'I'],
    [1, 'b', 'II'],
    [2, 'a', 'I'],
    [2, 'a', 'II'],
    [2, 'b', 'I'],
    [2, 'b', 'II'],
  ]
*/
```

Phương thức `dd`

Phương thức `dd` kết xuất các mục của collection và kết thúc thực thi tập lệnh:

```php
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
  Collection {
    #items: array:2 [
      0 => "John Doe"
      1 => "Jane Doe"
    ]
  }
*/
```

Nếu bạn không muốn ngừng thực thi tập lệnh, hãy sử dụng phương thức `dump` để thay thế.

Phương thức `diff`

Phương thức diff so sánh collection với một collection khác hoặc một mảng PHP thuần dựa trên các giá trị của nó. Phương thức này sẽ trả về các giá trị trong collection ban đầu không có trong collection đã cho:

```php
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

> Hoạt động của phương thức này được sửa đổi khi sử dụng Eloquent `Collections`.

Phương thức `diffAssoc`

Phương thức `diffAssoc` so sánh collection với một collection khác hoặc một mảng PHP thuần túy dựa trên các khóa và giá trị của nó. Phương thức này sẽ trả về các cặp khóa/giá trị trong collection ban đầu không có trong collection đã cho:

```php
$collection = collect([
  'color' => 'orange',
  'type' => 'fruit',
  'remain' => 6,
]);

$diff = $collection->diffAssoc([
  'color' => 'yellow',
  'type' => 'fruit',
  'remain' => 3,
  'used' => 6,
]);

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

Phương thức `diffKeys`

Phương thức `diffKeys` so sánh collection với một collection khác hoặc một mảng PHP thuần dựa trên các khóa của nó. Phương thức này sẽ trả về các cặp khóa/giá trị trong collection ban đầu nếu không có trong collection đã cho:

```php
$collection = collect([
  'one' => 10,
  'two' => 20,
  'three' => 30,
  'four' => 40,
  'five' => 50,
]);

$diff = $collection->diffKeys([
  'two' => 2,
  'four' => 4,
  'six' => 6,
  'eight' => 8,
```

```
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

Phương thức `doesntContain`

Phương thức doesntContain xác định liệu collection không chứa một mục nào đó. Bạn có thể truyền một hàm cho phương thức doesntContain để xác định xem một phần tử không tồn tại trong collection khớp với một phép kiểm tra so sánh đã cho hay không:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->doesntContain(function ($value, $key) {
  return $value < 5;
});

// false
```

Ngoài ra, bạn có thể truyền một chuỗi câu vào phương thức `doesntContain` để xác định xem collection có chứa giá trị mục nhất định hay không:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->doesntContain('Table');

// true

$collection->doesntContain('Desk');

// false
```

Bạn cũng có thể truyền một cặp khóa/giá trị vào phương thức `doesntContain`, điều này sẽ xác định xem cặp đã cho không tồn tại trong collection hay ngược lại:

```php
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 100],
]);

$collection->doesntContain('product', 'Bookcase');

// true
```

Phương thức `doesntContain` sử dụng so sánh "lỏng lẻo" khi kiểm tra giá trị mục, nghĩa là một chuỗi có giá trị nguyên sẽ được coi là bằng một số nguyên có cùng giá trị.

Phương thức `dump`

Phương thức `dump` loại bỏ các mục của collection:

```php
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/*
  Collection {
    #items: array:2 [
      0 => "John Doe"
      1 => "Jane Doe"
    ]
  }
*/
```

Nếu bạn muốn dừng thực thi tập lệnh sau khi kết xuất collection, thì hãy sử dụng phương thức `dd` thay thế.

Phương thức `duplicates`

Phương thức `duplicates` lấy và trả về các giá trị trùng lặp từ collection:

```php
$collection = collect(['a', 'b', 'a', 'c', 'b']);

$collection->duplicates();

// [2 => 'a', 4 => 'b']
```

Nếu collection chứa các mảng hoặc đối tượng, bạn có thể truyền chuỗi câu đại diện cho khóa của các thuộc tính mà bạn muốn kiểm tra các giá trị trùng lặp:

```php
$employees = collect([
  ['email' => 'abigail@example.com', 'position' => 'Developer'],
  ['email' => 'james@example.com', 'position' => 'Designer'],
  ['email' => 'victoria@example.com', 'position' => 'Developer'],
]);
$employees->duplicates('position');
// [2 => 'Developer']
```

Phương thức `duplicatesStrict`

Phương thức này có cùng cách thức hoạt động với phương thức `duplicates`; tuy nhiên, tất cả các giá trị được so sánh bằng cách sử dụng so sánh "nghiêm ngặt".

Phương thức `each`

Phương thức `each` sẽ lặp lại các mục trong collection và chuyển từng mục đến hàm nặc danh đã cho:

```php
$collection->each(function ($item, $key) {
  //
});
```

Nếu bạn muốn ngừng lặp lại các mục, bạn có thể trả về false khi đóng hàm:

```php
$collection->each(function ($item, $key) {
  if (/* condition */) {
```

```
    return false;
  }
});
```

Phương thức `eachSpread`

Phương thức `eachSpread` sẽ lặp lại các mục của collection, truyền từng giá trị mục lồng nhau vào callback đã cho:

```
$collection = collect([['John Doe', 35], ['Jane Doe', 33]]);

$collection->eachSpread(function ($name, $age) {
  //
});
```

Bạn có thể ngừng lặp lại các mục bằng cách trả lại `false` khi đóng hàm lặp:

```
$collection->eachSpread(function ($name, $age) {
  return false;
});
```

Phương thức `every`

Phương thức `every` có thể được sử dụng để xác minh rằng tất cả các phần tử của collection đều vượt qua một bài kiểm tra so sánh đã cho nào đó:

```
collect([1, 2, 3, 4])->every(function ($value, $key) {
  return $value > 2;
});

// false
```

Nếu collection trống, phương thức `every` sẽ trả về true:

```php
$collection = collect([]);

$collection->every(function ($value, $key) {
  return $value > 2;
});

// true
```

Phương thức `except`

Phương thức `except` trả về tất cả các mục trong bộ sưu tập ngoại trừ những mục có khóa khớp với các chuỗi câu so sánh đã được chỉ định:

```php
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

Nếu muốn nghịch đảo của `except`, hãy xem phương thức `only`.

> Hoạt động của phương thức này được sửa đổi khi sử dụng Eloquent Collections.

Phương thức `filter`

Phương thức `filter` sẽ lọc collection bằng cách sử dụng callback đã cho, chỉ giữ lại những mục đã vượt qua kiểm tra xác thực đã cho:

```php
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {
  return $value > 2;
});
```

```
$filtered->all();

// [3, 4]
```

Nếu không có callback nào được cung cấp, tất cả các mục nhập của bộ sưu tập mà tương đương với `false` sẽ bị xóa:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);

$collection->filter()->all();

// [1, 2, 3]
```

Đối với nghịch đảo của `filter`, hãy xem phương thức `reject`.

Phương thức `first`

Phương thức `first` trả về phần tử đầu tiên trong collection vượt qua một bài kiểm tra xác thực nhất định:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {
  return $value > 2;
});

// 3
```

Bạn cũng có thể gọi phương thức `first` không có đối số để lấy phần tử đầu tiên trong collection. Nếu bộ sưu tập trống, `null` được trả lại:

```
collect([1, 2, 3, 4])->first();

// 1
```

Phương thức `firstWhere`

Phương thức `firstWhere` trả về phần tử đầu tiên trong collection với cặp khóa/giá trị đã cho:

```
$collection = collect([
  ['name' => 'Regena', 'age' => null],
  ['name' => 'Linda', 'age' => 14],
  ['name' => 'Diego', 'age' => 23],
  ['name' => 'Linda', 'age' => 84],
]);

$collection->firstWhere('name', 'Linda');

// ['name' => 'Linda', 'age' => 14]
```

Bạn cũng có thể gọi phương thức `firstWhere` bằng toán tử so sánh:

```
$collection->firstWhere('age', '>=', 18);

// ['name' => 'Diego', 'age' => 23]
```

Giống như phương thức `where` , bạn có thể truyền một đối số cho phương thức `firstWhere`. Trong trường hợp này, phương thức `firstWhere` sẽ trả về mục đầu tiên mà giá trị của khóa mục đã cho là "true":

```
$collection->firstWhere('age');

// ['name' => 'Linda', 'age' => 14]
```

Phương thức `flatMap`

Phương thức `flatMap` lặp qua bộ sưu tập và chuyển từng giá trị cho hàm đã cho. Hàm nặc danh có thể tự do sửa đổi vật phẩm và trả lại, do đó tạo thành một collection vật phẩm sửa đổi mới. Sau đó, mảng được làm phẳng một cấp:

```
$collection = collect([
```

```
  ['name' => 'Sally'],
  ['school' => 'Arkansas'],
  ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
  return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

Phương thức `flatten`

Phương thức `flatten` làm phẳng một collection nhiều chiều thành một chiều duy nhất:

```
$collection = collect([
  'name' => 'taylor',
  'languages' => [
    'php', 'javascript'
  ]
]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

Nếu cần, bạn có thể chuyển cho phương thức `flatten` một đối số "độ sâu":

```
$collection = collect([
  'Apple' => [
    [
      'name' => 'iPhone 6S',
```

```
      'brand' => 'Apple'
    ],
  ],
  'Samsung' => [
    [
      'name' => 'Galaxy S7',
      'brand' => 'Samsung'
    ],
  ],
]);

$products = $collection->flatten(1);

$products->values()->all();

/*
  [
    ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
  ]
*/
```

Trong ví dụ này, việc gọi `flatten` mà không cung cấp độ sâu cũng sẽ làm phẳng các mảng lồng nhau, dẫn đến `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Cung cấp độ sâu cho phép bạn chỉ định số lượng các mảng lồng nhau sẽ được làm phẳng.

Phương thức `flip`

Phương thức `flip` hoán đổi các khóa của collection với các giá trị tương ứng của chúng:

```php
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

Phương thức `forget`

Phương thức `forget` xóa một mục khỏi bộ sưu tập bằng khóa của nó:

```php
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

> **Chú ý:** Không giống như hầu hết các phương pháp thu thập khác, phương thức `forget` không trả về một tập hợp được sửa đổi mới; nó sửa đổi bộ sưu tập mà nó được gọi.

Phương thức `forPage`

Phương thức `forPage` sẽ trả về một collection mới chứa các mục sẽ có mặt trên một số trang nhất định. Phương thức chấp nhận số trang làm đối số đầu tiên và số mục hiển thị trên mỗi trang làm đối số thứ hai:

```php
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();
```

```
// [4, 5, 6]
```

Phương thức `get`

Phương thức `get` trả về mục tại một khóa nhất định. Nếu khóa không tồn tại, `null` được trả về:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

Bạn có thể tùy chọn truyền một giá trị mặc định làm đối số thứ hai:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('age', 34);

// 34
```

Bạn thậm chí có thể truyền một callback làm giá trị mặc định của phương thức. Kết quả của callback sẽ được trả về nếu khóa được chỉ định không tồn tại:

```
$collection->get('email', function () {
  return 'taylor@example.com';
});

// taylor@example.com
```

Phương thức `groupBy`

Phương thức `groupBy` sẽ nhóm các mục của collection theo một khóa nhất định:

```php
$collection = collect([
  ['account_id' => 'account-x10', 'product' => 'Chair'],
  ['account_id' => 'account-x10', 'product' => 'Bookcase'],
  ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->all();

/*
  [
    'account-x10' => [
      ['account_id' => 'account-x10', 'product' => 'Chair'],
      ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'account-x11' => [
      ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
  ]
*/
```

Thay vì truyền một chuỗi `key`, bạn có thể truyền một callback. Callback này sẽ trả về giá trị bạn muốn nhóm:

```php
$grouped = $collection->groupBy(function ($item, $key) {
  return substr($item['account_id'], -3);
});

$grouped->all();

/*
  [
    'x10' => [
      ['account_id' => 'account-x10', 'product' => 'Chair'],
      ['account_id' => 'account-x10', 'product' => 'Bookcase'],
```

```
    ],
    'x11' => [
      ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
  ]
*/
```

Nhiều tiêu chí nhóm có thể được truyền dưới dạng một mảng. Mỗi phần tử mảng sẽ được áp dụng cho cấp độ tương ứng trong một mảng đa chiều:

```
$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);

$result = $data->groupBy(['skill', function ($item) {
  return $item['roles'];
}], $preserveKeys = true);

/*
[
  1 => [
    'Role_1' => [
      10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
      20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    ],
    'Role_2' => [
      20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    ],
    'Role_3' => [
      10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    ],
  ],
  2 => [
    'Role_1' => [
```

```
      30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    ],
    'Role_2' => [
      40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
    ],
  ],
];
*/
```

Phương thức `has`

Phương thức `has` xác định xem một khóa nào đó có tồn tại trong collection hay không:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);

$collection->has('product');

// true

$collection->has(['product', 'amount']);

// true

$collection->has(['amount', 'price']);

// false
```

Phương thức `implode`

Phương thức `implode` kết hợp các mục trong một collection. Các đối số của nó phụ thuộc vào loại mục trong collection. Nếu collection chứa các mảng hoặc đối tượng, bạn phải truyền khóa của các thuộc tính mà bạn muốn tham gia và chuỗi "keo - dấu phân biệt" mà bạn muốn đặt giữa các giá trị:

```
$collection = collect([
  ['account_id' => 1, 'product' => 'Desk'],
```

```
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

Nếu collection chứa các chuỗi hoặc giá trị số đơn giản, bạn nên truyền "keo - dấu phân biệt" làm đối số duy nhất cho phương thức:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

Phương thức `intersect`

Phương thức `intersect` loại bỏ bất kỳ giá trị nào khỏi collection ban đầu không có trong array, collection hoặc collection đã cho. Collection kết quả sẽ lưu giữ các khóa của bộ sưu tập ban đầu:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

> Hoạt động của phương thức này được sửa đổi khi sử dụng Eloquent Collections.

Phương thức `intersectByKeys`

Phương thức `intersectByKeys` sẽ xóa bất kỳ khóa nào và các giá trị tương ứng của chúng khỏi collection ban đầu không có trong collecction đã cho có thể là array hoặc collection:

```php
$collection = collect([
  'serial' => 'UX301', 'type' => 'screen', 'year' => 2009,
]);

$intersect = $collection->intersectByKeys([
  'reference' => 'UX404', 'type' => 'tab', 'year' => 2011,
]);

$intersect->all();

// ['type' => 'screen', 'year' => 2009]
```

Phương thức `isEmpty`

Phương thức `isEmpty` trả về `true` nếu bộ sưu tập trống; nếu ngược lại, `false` được trả lại:

```php
collect([])->isEmpty();

// true
```

Phương thức `isNotEmpty`

Phương thức `isNotEmpty` trả về `true` nếu collection không trống; nếu ngược lại, `false` được trả lại:

```php
collect([])->isNotEmpty();

// false
```

Phương thức `join`

Phương thức `join` nối các giá trị của collection bằng một chuỗi. Sử dụng đối số thứ hai của phương thức này, bạn cũng có thể chỉ định cách phần tử cuối cùng sẽ được nối vào chuỗi:

```
collect(['a', 'b', 'c'])->join(', '); // 'a, b, c'
collect(['a', 'b', 'c'])->join(', ', ', and '); // 'a, b, and c'
collect(['a', 'b'])->join(', ', ' and '); // 'a and b'
collect(['a'])->join(', ', ' and '); // 'a'
collect([])->join(', ', ' and '); // ''
```

Phương thức keyBy

Phương thức keyBy sẽ khóa collection bằng khóa đã cho. Nếu nhiều mục có cùng một khóa, chỉ cái cuối cùng sẽ xuất hiện trong collection mới:

```
$collection = collect([
  ['product_id' => 'prod-100', 'name' => 'Desk'],
  ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
  [
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
  ]
*/
```

Bạn cũng có thể truyền một callback đến phương thức. callback sẽ trả về giá trị để khóa collection bằng cách sau:

```
$keyed = $collection->keyBy(function ($item) {
  return strtoupper($item['product_id']);
});

$keyed->all();
```

```
/*
  [
    'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
  ]
*/
```

Phương thức `keys`

Phương thức `keys` sẽ trả về tất cả các khóa của collection:

```
$collection = collect([
  'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
  'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

Phương thức `last`

Phương thức `last` trả về phần tử cuối cùng trong collection khi vượt qua một bài kiểm tra xác thực đã cho:

```
collect([1, 2, 3, 4])->last(function ($value, $key) {
  return $value < 3;
});

// 2
```

Bạn cũng có thể gọi phương thức `last` không có đối số để lấy phần tử cuối cùng trong

collection. Nếu collection trống, `null` được trả lại:

```
collect([1, 2, 3, 4])->last();

// 4
```

Phương thức `macro`

Phương thức tĩnh `macro` cho phép bạn thêm các phương thức vào class `Collection` tại thời điểm chạy. Tham khảo tài liệu về mở rộng collection ở bên dưới để biết thêm thông tin.

Phương thức `make`

Phương thức tĩnh `make` sẽ tạo một đối tượng collection mới. Xem phần Tạo Collection.

Phương thức `map`

Phương thức `map` lặp qua collection và truyền từng giá trị cho callback đã cho. Callback có thể tự do sửa đổi mục và trả lại, do đó tạo thành một collection các mục đã sửa đổi mới:

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
  return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

> **Chú ý:** Giống như hầu hết các phương thức collection khác, `map` sẽ trả về một đối tượng tập hợp mới; nó không sửa đổi collection mà nó được gọi. Nếu bạn muốn chuyển đổi bộ sưu tập ban đầu, hãy sử dụng phương thức transform.

Phương thức `mapInfo`

Phương thức `mapInto` lặp qua collection, tạo một đối tượng mới của class đã cho bằng cách truyền giá trị vào constructor:

```
class Currency
{
  /**
   * Create a new currency instance.
   *
   * @param  string  $code
   * @return void
   */
  function __construct(string $code)
  {
    $this->code = $code;
  }
}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all();

// [Currency('USD'), Currency('EUR'), Currency('GBP')]
```

Phương thức `mapSpread`

Phương thức mapSpread sẽ lặp lại các mục của collection, truyền từng giá trị mục lồng nhau vào một hàm đã cho. Hàm này có thể tự do sửa đổi mục và trả lại, do đó tạo thành một collection mới gồm các mục đã sửa đổi:

```
$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function ($even, $odd) {
```

```
  return $even + $odd;
});

$sequence->all();

// [1, 5, 9, 13, 17]
```

Phương thức `mapToGroups`

Phương thức `mapToGroups` sẽ nhóm các mục của collection theo hàm nặc danh đã cho. Hàm này sẽ trả về một mảng hỗn hợp chứa một cặp khóa/giá trị duy nhất, do đó tạo thành một collection mới các giá trị được nhóm lại:

```
$collection = collect([
  [
    'name' => 'John Doe',
    'department' => 'Sales',
  ],
  [
    'name' => 'Jane Doe',
    'department' => 'Sales',
  ],
  [
    'name' => 'Johnny Doe',
    'department' => 'Marketing',
  ]
]);

$grouped = $collection->mapToGroups(function ($item, $key) {
  return [$item['department'] => $item['name']];
});

$grouped->all();

/*
  [
    'Sales' => ['John Doe', 'Jane Doe'],
```

```
    'Marketing' => ['Johnny Doe'],
  ]
*/

$grouped->get('Sales')->all();

// ['John Doe', 'Jane Doe']
```

Phương thức `mapWithKeys`

Phương thức `mapWithKeys` sẽ lặp qua collection và chuyển từng giá trị cho callback đã cho. Callback phải trả về một mảng hỗn hợp chứa một cặp khóa/giá trị:

```
$collection = collect([
  [
    'name' => 'John',
    'department' => 'Sales',
    'email' => 'john@example.com',
  ],
  [
    'name' => 'Jane',
    'department' => 'Marketing',
    'email' => 'jane@example.com',
  ]
]);

$keyed = $collection->mapWithKeys(function ($item, $key) {
  return [$item['email'] => $item['name']];
});

$keyed->all();

/*
  [
    'john@example.com' => 'John',
    'jane@example.com' => 'Jane',
  ]
```

```
*/
```

Phương thức `max`

Phương thức `max` sẽ trả về giá trị lớn nhất của một khóa nhất định:

```
$max = collect([
  ['foo' => 10],
  ['foo' => 20]
])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

Phương thức `median`

Phương thức `median` sẽ trả về giá trị trung bình của một khóa nhất định:

```
$median = collect([
  ['foo' => 10],
  ['foo' => 10],
  ['foo' => 20],
  ['foo' => 40]
])->median('foo');

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5
```

Phương thức `merge`

Phương thức `merge` này hợp nhất mảng hoặc collection đã cho với collection ban đầu. Nếu khóa chuỗi trong các mục đã cho khớp với khóa chuỗi trong collection ban đầu, giá trị của các mục đã cho sẽ ghi đè giá trị trong collection ban đầu:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

Nếu khóa của các mục đã cho là số, các giá trị sẽ được thêm vào cuối collection:

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

Phương thức `mergeRecursive`

Phương thức `mergeRecursive` sẽ hợp nhất mảng hoặc collection đã cho theo cách đệ quy với collection ban đầu. Nếu một khóa chuỗi trong các mục đã cho khớp với khóa chuỗi trong collection ban đầu, thì các giá trị cho các khóa này sẽ được hợp nhất với nhau thành một mảng và điều này được thực hiện theo cách đệ quy:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->mergeRecursive([
  'product_id' => 2,
  'price' => 200,
  'discount' => false
]);
```

```php
$merged->all();

// ['product_id' => [1, 2], 'price' => [100, 200], 'discount' => false]
```

Phương thức `min`

Phương thức `min` sẽ trả về giá trị nhỏ nhất của một khóa nhất định:

```php
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

Phương thức `mode`

Phương thức `mode` sẽ trả về giá trị mode của một khóa nhất định:

```php
$mode = collect([
  ['foo' => 10],
  ['foo' => 10],
  ['foo' => 20],
  ['foo' => 40]
])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]

$mode = collect([1, 1, 2, 2])->mode();
```

```
// [1, 2]
```

Phương thức `nth`

Phương thức `nth` tạo một collection mới bao gồm mọi phần tử thứ n:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

Bạn có thể tùy chọn truyền một khoảng bù bắt đầu làm đối số thứ hai:

```
$collection->nth(4, 1);

// ['b', 'f']
```

Phương thức `only`

Phương thức `only` sẽ trả về các mục trong collection với các khóa được chỉ định:

```
$collection = collect([
  'product_id' => 1,
  'name' => 'Desk',
  'price' => 100,
  'discount' => false
]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

Đối với nghịch đảo của `only`, hãy xem phương pháp `except`.

> Hoạt động của phương thức này được sửa đổi khi sử dụng Eloquent Collections.

Phương thức `pad`

Phương thức `pad` sẽ điền vào mảng với giá trị đã cho cho đến khi mảng đạt đến kích thước được chỉ định. Phương thức này hoạt động giống như hàm `array_pad` PHP.

Để đệm sang trái, bạn nên chỉ định kích thước âm. Không có khoảng đệm sẽ diễn ra nếu giá trị tuyệt đối của kích thước đã cho nhỏ hơn hoặc bằng độ dài của mảng:

```
$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all();

// ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all();

// [0, 0, 'A', 'B', 'C']
```

Phương thức `partition`

Phương thức `partition` sẽ có thể được kết hợp với cấu trúc mảng PHP để tách các phần tử vượt qua bài kiểm tra xác thực đã cho với những phần tử không vượt qua:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

[$underThree, $equalOrAboveThree] = $collection->partition(function ($i) {
  return $i < 3;
});
```

```php
$underThree->all();

// [1, 2]

$equalOrAboveThree->all();

// [3, 4, 5, 6]
```

Phương thức `pipe`

Phương thức `pipe` sẽ truyền collection vào một hàm đã cho và kết quả trả về của hàm sẽ được thực thi:

```php
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
  return $collection->sum();
});

// 6
```

Phương thức `pipeInto`

Phương thức `pipeInto` tạo một đối tượng mới của class đã cho và truyền collection vào constructor:

```php
class ResourceCollection
{
  /**
   * The Collection instance.
   */
  public $collection;

  /**
   * Create a new ResourceCollection instance.
   *
```

```
     * @param  Collection  $collection
     * @return void
     */
    public function __construct(Collection $collection)
    {
        $this->collection = $collection;
    }
}

$collection = collect([1, 2, 3]);

$resource = $collection->pipeInto(ResourceCollection::class);

$resource->collection->all();

// [1, 2, 3]
```

Phương thức `pipeThrough`

Phương thức `pipeThrough` sẽ truyền collection đến một mảng các hàm nặc danh đã cho và trả về kết quả của các hàm được thực thi:

```
$collection = collect([1, 2, 3]);

$result = $collection->pipeThrough([
    function ($collection) {
        return $collection->merge([4, 5]);
    },
    function ($collection) {
        return $collection->sum();
    },
]);

// 15
```

Phương thức `pluck`

Phương thức `pluck` sẽ truy xuất tất cả các giá trị cho một khóa nhất định:

```
$collection = collect([
  ['product_id' => 'prod-100', 'name' => 'Desk'],
  ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

Bạn cũng có thể chỉ định collection báo kết quả theo khóa key:

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

Phương thức `pluck` sẽ cũng hỗ trợ truy xuất các giá trị lồng nhau bằng cách sử dụng ký hiệu "dấu chấm":

```
$collection = collect([
  [
    'speakers' => [
      'first_day' => ['Rosa', 'Judith'],
      'second_day' => ['Angela', 'Kathleen'],
    ],
  ],
]);

$plucked = $collection->pluck('speakers.first_day');

$plucked->all();
```

```
// ['Rosa', 'Judith']
```

Nếu các khóa trùng lặp tồn tại, phần tử phù hợp cuối cùng sẽ được chèn vào collection đã lấy ra:

```
collection = collect([
  ['brand' => 'Tesla',  'color' => 'red'],
  ['brand' => 'Pagani', 'color' => 'white'],
  ['brand' => 'Tesla',  'color' => 'black'],
  ['brand' => 'Pagani', 'color' => 'orange'],
]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all();

// ['Tesla' => 'black', 'Pagani' => 'orange']
```

Phương thức pop

Phương thức pop loại bỏ và trả về mục cuối cùng từ collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

Bạn có thể truyền một số nguyên vào phương thức pop để loại bỏ và trả về nhiều mục từ phần cuối của một collection:

```php
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop(3);

// collect([5, 4, 3])

$collection->all();

// [1, 2]
```

Phương thức `prepend`

Phương pháp `prepend` thêm một mục vào đầu collection:

```php
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]
```

Bạn cũng có thể truyền đối số thứ hai để chỉ định khóa của mục được thêm trước:

```php
$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]
```

Phương thức `pull`

Phương thức `pull` sẽ xóa và trả về một mục khỏi collection bằng khóa của nó:

```php
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

// ['product_id' => 'prod-100']
```

Phương thức `push`

Phương thức `push` sẽ nối một mục vào cuối collection:

```php
$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

Phương thức `put`

Phương thức `put` đặt khóa và giá trị đã cho trong collection:

```php
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

Phương thức `random`

Phương thức `random` trả về một mục ngẫu nhiên từ collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

Bạn có thể truyền một số nguyên vào phương thức `random` để chỉ định số lượng mục bạn muốn lấy ngẫu nhiên. Một collection các vật phẩm luôn được trả lại khi vượt qua số lượng vật phẩm bạn muốn nhận một cách minh bạch:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

Nếu cá thể collection có ít mục hơn yêu cầu, phương thức `random` sẽ ném một `InvalidArgumentException`.

Phương thức `range`

Phương thức `range` sẽ trả về một collection chứa các số nguyên giữa phạm vi được chỉ định:

```
$collection = collect()->range(3, 6);

$collection->all();

// [3, 4, 5, 6]
```

Phương thức `reduce`

Phương thức `reduce` sẽ giảm collection xuống một giá trị duy nhất, truyền kết quả của mỗi lần lặp vào lần lặp tiếp theo:

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
  return $carry + $item;
});

// 6
```

Giá trị cho biến `$carry` lần lặp đầu tiên là `null`; tuy nhiên, bạn có thể chỉ định giá trị ban đầu của nó bằng cách truyền đối số thứ hai vào `reduce`:

```
$collection->reduce(function ($carry, $item) {
  return $carry + $item;
}, 4);

// 10
```

Phương thức `reduce` này cũng chuyển các khóa mảng trong collection kết hợp tới callback đã cho:

```
$collection = collect([
  'usd' => 1400,
  'gbp' => 1200,
  'eur' => 1000,
]);

$ratio = [
  'usd' => 1,
  'gbp' => 1.37,
  'eur' => 1.22,
];

$collection->reduce(function ($carry, $value, $key) use ($ratio) {
```

```
  return $carry + ($value * $ratio[$key]);
});

// 4264
```

Phương thức `reduceSpread`

Phương thức `reduceSpread` sẽ giảm collection thành một mảng giá trị, truyền kết quả của mỗi lần lặp vào lần lặp tiếp theo. Phương pháp này tương tự như phương thức `reduce`; tuy nhiên, nó có thể chấp nhận nhiều giá trị ban đầu:

```
[$creditsRemaining, $batch] = Image::where('status', 'unprocessed')
  ->get()
  ->reduceSpread(function ($creditsRemaining, $batch, $image) {
    if ($creditsRemaining >= $image->creditsRequired()) {
      $batch->push($image);
      $creditsRemaining -= $image->creditsRequired();
    }
    return [$creditsRemaining, $batch];
  }, $creditsAvailable, collect());
```

Phương thức `reject`

Phương thức `reject` lọc collection bằng cách sử dụng hàm nặc danh đã cho. Hàm này sẽ trả trở lại true nếu mục cần được xóa khỏi collection kết quả:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($value, $key) {
  return $value > 2;
});

$filtered->all();

// [1, 2]
```

Đối với nghịch đảo của phương thức `reject`, hãy xem phương thức `filter`.

Phương thức `replace`

Phương thức `replace` này hoạt động tương tự như `merge`; tuy nhiên, ngoài việc ghi đè các mục phù hợp có khóa chuỗi, phương thức `replace` cũng sẽ ghi đè các mục trong collection có khóa số phù hợp:

```
$collection = collect(['Taylor', 'Abigail', 'James']);

$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);

$replaced->all();

// ['Taylor', 'Victoria', 'James', 'Finn']
```

Phương thức `replaceRecursive`

Phương thức này hoạt động giống như `replace`, nhưng nó sẽ lặp lại thành các mảng và áp dụng cùng một quy trình thay thế cho các giá trị bên trong:

```
$collection = collect([
  'Taylor',
  'Abigail',
  [
    'James',
    'Victoria',
    'Finn'
  ]
]);

$replaced = $collection->replaceRecursive([
  'Charlie',
  2 => [1 => 'King']
]);
```

```php
$replaced->all();

// ['Charlie', 'Abigail', ['James', 'King', 'Finn']]
```

Phương thức `reverse`

Phương pháp `reverse` đảo ngược thứ tự của các mục trong collection, giữ nguyên các khóa ban đầu:

```php
$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();

/*
  [
    4 => 'e',
    3 => 'd',
    2 => 'c',
    1 => 'b',
    0 => 'a',
  ]
*/
```

Phương thức `search`

Phương thức `search` tìm kiếm collection cho giá trị đã cho và trả về khóa của nó nếu được tìm thấy. Nếu hàng không được tìm thấy, `false` được trả lại:

```php
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

Việc tìm kiếm được thực hiện bằng cách so sánh "lỏng lẻo", nghĩa là một chuỗi có giá trị nguyên sẽ được coi là bằng một số nguyên có cùng giá trị. Để sử dụng so sánh "nghiêm ngặt", hãy truyền true làm đối số thứ hai cho phương thức:

```
collect([2, 4, 6, 8])->search('4', $strict = true);

// false
```

Ngoài ra, bạn có thể cung cấp hàm của riêng mình để tìm kiếm mục đầu tiên vượt qua bài kiểm tra xác thực đã cho:

```
collect([2, 4, 6, 8])->search(function ($item, $key) {
  return $item > 5;
});

// 2
```

Phương thức `shift`

Phương thức `shift` loại bỏ và trả về mục đầu tiên khỏi collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

Bạn có thể truyền một số nguyên cho phương thức `shift` để xóa và trả lại nhiều mục từ đầu một collection:

```
$collection = collect([1, 2, 3, 4, 5]);
```

```php
$collection->shift(3);

// collect([1, 2, 3])

$collection->all();

// [4, 5]
```

Phương thức `shuffle`

Phương thức `shuffle` trộn ngẫu nhiên các mục trong collection:

```php
$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] - (generated randomly)
```

Phương thức `sliding`

Phương thức sliding sẽ trả về một collection các khối mới đại diện cho chế độ xem "cửa sổ trượt" của các mục trong collection:

```php
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(2);

$chunks->toArray();

// [[1, 2], [2, 3], [3, 4], [4, 5]]
```

Điều này đặc biệt hữu ích khi kết hợp với phương thức `eachSpread`:

```php
$transactions->sliding(2)->eachSpread(function ($previous, $current) {
  $current->total = $previous->total + $current->amount;
});
```

Bạn có thể tùy ý truyền giá trị "bước" thứ hai, giá trị này xác định khoảng cách giữa mục đầu tiên của mỗi đoạn:

```php
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(3, step: 2);

$chunks->toArray();

// [[1, 2, 3], [3, 4, 5]]
```

Phương thức `skip`

Phương thức skip sẽ trả về một collection mới, với số phần tử đã cho bị xóa khỏi đầu collection:

```php
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$collection = $collection->skip(4);

$collection->all();

// [5, 6, 7, 8, 9, 10]
```

Phương thức `skipUtil`

Phương thức `skipUntil` bỏ qua các mục từ collection cho đến khi callback đã cho trả về true và sau đó trả về các mục còn lại trong collection dưới dạng một đối tượng collection mới:

```php
$collection = collect([1, 2, 3, 4]);
```

```php
$subset = $collection->skipUntil(function ($item) {
  return $item >= 3;
});

$subset->all();

// [3, 4]
```

Bạn cũng có thể truyền một giá trị đơn giản cho phương thức `skipUntil` để bỏ qua tất cả các mục cho đến khi tìm thấy giá trị đã cho:

```php
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipUntil(3);

$subset->all();

// [3, 4]
```

> **Chú ý:** Nếu không tìm thấy giá trị đã cho hoặc callback không bao giờ trả về true, thì phương thức `skipUntil` sẽ trả về một collection rỗng.

Phương thức `skipWhite`

Phương thức `skipWhile` sẽ bỏ qua các mục từ collection trong khi callback đã cho trả về true và sau đó trả về các mục còn lại trong collection dưới dạng một collection mới:

```php
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipWhile(function ($item) {
  return $item <= 3;
});

$subset->all();
```

```
// [4]
```

> **Chú ý:** Nếu callback không bao giờ trả về `false`, thì phương thức `skipWhile` sẽ trả về một collection rỗng.

Phương thức `slice`

Phương thức `slice` sẽ trả về một phần của collection bắt đầu từ chỉ mục đã cho:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

Nếu bạn muốn giới hạn kích thước của tập hợp con được trả về, hãy truyền kích thước mong muốn làm đối số thứ hai cho phương thức:

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

Slice trả về sẽ giữ các khóa theo mặc định. Nếu bạn không muốn giữ lại các khóa ban đầu, bạn có thể sử dụng phương thức `values` để lập chỉ mục lại chúng.

Phương thức `sole`

Phương thức sole sẽ trả về phần tử đầu tiên trong collection vượt qua một bài kiểm tra xác thực đã cho, nhưng chỉ khi kiểm tra xác thực khớp chính xác một phần tử:

```
collect([1, 2, 3, 4])->sole(function ($value, $key) {
```

```
  return $value === 2;
});

// 2
```

Bạn cũng có thể truyền một cặp khóa/giá trị cho phương thức sole, phương thức này sẽ trả về phần tử đầu tiên trong collection khớp với cặp đã cho, nhưng chỉ khi nó khớp đúng một phần tử:

```
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 100],
]);

$collection->sole('product', 'Chair');

// ['product' => 'Chair', 'price' => 100]
```

Ngoài ra, bạn cũng có thể gọi phương thức `sole` mà không có đối số để lấy phần tử đầu tiên trong bộ sưu tập nếu chỉ có một phần tử:

```
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
]);

$collection->sole();

// ['product' => 'Desk', 'price' => 200]
```

Nếu không có phần tử nào trong bộ sưu tập được phương thức `sole` trả về, một `\Illuminate\Collections\ItemNotFoundException` sẽ được ném ra. Nếu có nhiều hơn một phần tử cần được trả về, một phần tử `\Illuminate\Collections\MultipleItemsFoundException` sẽ được ném ra.

Phương thức `some`

Bí danh cho phương thức `contains`.

Phương thức `sort`

Phương thức `sort` sẽ sắp xếp bộ sưu tập. Tập hợp đã sắp xếp giữ các khóa mảng ban đầu, vì vậy trong ví dụ sau, chúng tôi sẽ sử dụng phương thức `values` để đặt lại các khóa thành các chỉ mục được đánh số liên tiếp:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all();

// [1, 2, 3, 4, 5]
```

Nếu nhu cầu sắp xếp của bạn nâng cao hơn, bạn có thể truyền một callback tới `sort` bằng thuật toán của riêng mình. Tham khảo tài liệu PHP `uasort`, đó là những gì mà phương thức `sort` của collection sử dụng trong nội bộ.

> Nếu bạn cần sắp xếp một collection các mảng hoặc đối tượng lồng nhau, hãy xem các phương thức `sortBy` và `sortByDesc`.

Phương thức `sortBy`

Phương thức `sortBy` sẽ sắp xếp collection theo khóa đã cho. Collection đã sắp xếp giữ các khóa mảng ban đầu, vì vậy trong ví dụ sau, chúng ta sẽ sử dụng phương thức `values` để đặt lại các khóa thành các chỉ mục được đánh số liên tiếp:

```
$collection = collect([
  ['name' => 'Desk', 'price' => 200],
  ['name' => 'Chair', 'price' => 100],
  ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');
```

```php
$sorted->values()->all();

/*
  [
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
    ['name' => 'Desk', 'price' => 200],
  ]
*/
```

Phương thức `sortBy` chấp nhận các cờ sắp xếp làm đối số thứ hai của nó:

```php
$collection = collect([
  ['title' => 'Item 1'],
  ['title' => 'Item 12'],
  ['title' => 'Item 3'],
]);

$sorted = $collection->sortBy('title', SORT_NATURAL);

$sorted->values()->all();

/*
  [
    ['title' => 'Item 1'],
    ['title' => 'Item 3'],
    ['title' => 'Item 12'],
  ]
*/
```

Ngoài ra, bạn có thể truyền hàm kiểm tra của riêng mình để xác định cách sắp xếp các giá trị của collection:

```php
$collection = collect([
  ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
  ['name' => 'Chair', 'colors' => ['Black']],
```

```
  ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$sorted = $collection->sortBy(function ($product, $key) {
  return count($product['colors']);
});

$sorted->values()->all();

/*
  [
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
  ]
*/
```

Nếu bạn muốn sắp xếp collection của mình theo nhiều thuộc tính, bạn có thể truyền một mảng các thao tác sắp xếp cho phương thức `sortBy`. Mỗi thao tác sắp xếp phải là một mảng bao gồm thuộc tính mà bạn muốn sắp xếp và hướng của sắp xếp mong muốn:

```
$collection = collect([
  ['name' => 'Taylor Otwell', 'age' => 34],
  ['name' => 'Abigail Otwell', 'age' => 30],
  ['name' => 'Taylor Otwell', 'age' => 36],
  ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
  ['name', 'asc'],
  ['age', 'desc'],
]);

$sorted->values()->all();

/*
  [
```

```
    ['name' => 'Abigail Otwell', 'age' => 32],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Taylor Otwell', 'age' => 34],
  ]
*/
```

Khi sắp xếp một collection theo nhiều thuộc tính, bạn cũng có thể cung cấp các hàm từng thao tác sắp xếp:

```
$collection = collect([
  ['name' => 'Taylor Otwell', 'age' => 34],
  ['name' => 'Abigail Otwell', 'age' => 30],
  ['name' => 'Taylor Otwell', 'age' => 36],
  ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
  fn ($a, $b) => $a['name'] <=> $b['name'],
  fn ($a, $b) => $b['age'] <=> $a['age'],
]);

$sorted->values()->all();

/*
  [
    ['name' => 'Abigail Otwell', 'age' => 32],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Taylor Otwell', 'age' => 34],
  ]
*/
```

Phương thức `sortByDesc`

Phương thức này có cùng cách thức với phương thức `sortBy`, nhưng sẽ sắp xếp collection

theo thứ tự ngược lại.

Phương thức `sortDesc`

Phương thức này sẽ sắp xếp collection theo thứ tự ngược lại với phương thức `sort`:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sortDesc();

$sorted->values()->all();

// [5, 4, 3, 2, 1]
```

Không giống như `sort`, bạn có thể không truyền một hàm vào phương thức `sortDesc`. Thay vào đó, bạn nên sử dụng phương thức `sort` và đảo ngược so sánh của mình.

Phương thức `sortKeys`

Phương thức `sortKeys` sẽ sắp xếp collection theo các khóa của mảng kết hợp bên dưới:

```
$collection = collect([
  'id' => 22345,
  'first' => 'John',
  'last' => 'Doe',
]);

$sorted = $collection->sortKeys();

$sorted->all();

/*
  [
    'first' => 'John',
    'id' => 22345,
    'last' => 'Doe',
  ]
```

```
*/
```

Phương thức `sortKeysDesc`

Phương thức này có cùng cách thức với phương thức `sortKeys`, nhưng sẽ sắp xếp collection theo thứ tự ngược lại.

Phương thức `sortKeysUsing`

Phương thức `sortKeysUsing` sẽ sắp xếp collection theo các khóa của mảng kết hợp cơ bản bằng cách sử dụng một callback:

```
$collection = collect([
  'ID' => 22345,
  'first' => 'John',
  'last' => 'Doe',
]);

$sorted = $collection->sortKeysUsing('strnatcasecmp');

$sorted->all();

/*
  [
    'first' => 'John',
    'ID' => 22345,
    'last' => 'Doe',
  ]
*/
```

Callback phải là một hàm so sánh trả về một số nguyên nhỏ hơn, bằng hoặc lớn hơn 0. Để biết thêm thông tin, hãy tham khảo tài liệu PHP `uksort`, đây là hàm PHP mà phương thức `sortKeysUsing` sử dụng nội bộ bên trong đó.

Phương thức `splice`

Phương thức `splice` sẽ loại bỏ và trả về một phần của các mục bắt đầu từ chỉ mục được

chỉ định:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

Bạn có thể truyền đối số thứ hai để giới hạn kích thước của collection kết quả:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 4, 5]
```

Ngoài ra, bạn có thể truyền đối số thứ ba chứa các mục mới để thay thế các mục bị xóa khỏi collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();
```

```
// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

Phương thức `split`

Phương thức `split` chia một collection thành một số nhóm nhất định:

```
$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->all();

// [[1, 2], [3, 4], [5]]
```

Phương thức `splitIn`

Phương thức `splitIn` chia nhỏ collection thành một số nhóm nhất định, điền đầy đủ các nhóm không phải đầu cuối trước khi phân bổ phần còn lại cho nhóm cuối cùng:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$groups = $collection->splitIn(3);

$groups->all();

// [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

Phương thức `sum`

Phương thức `sum` trả về tổng của tất cả các mục trong collection:

```php
collect([1, 2, 3, 4, 5])->sum();

// 15
```

Nếu collection chứa các mảng hoặc đối tượng lồng nhau, bạn nên chuyển một khóa sẽ được sử dụng để xác định các giá trị cần tính tổng:

```php
$collection = collect([
  ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
  ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

Ngoài ra, bạn có thể truyền hàm của riêng mình để xác định giá trị nào của bộ sưu tập để tính tổng:

```php
$collection = collect([
  ['name' => 'Chair', 'colors' => ['Black']],
  ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
  ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function ($product) {
  return count($product['colors']);
});

// 6
```

Phương thức `take`

Phương thức `take` này trả về một collection mới với số lượng mục được chỉ định:

```php
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

Bạn cũng có thể truyền một số nguyên âm để lấy số lượng mục được chỉ định từ cuối collection:

```php
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

Phương thức `takeUntil`

Phương thức `takeUntil` sẽ trả về các mục trong collection cho đến khi callback đã cho trả về true:

```php
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(function ($item) {
  return $item >= 3;
});

$subset->all();

// [1, 2]
```

Bạn cũng có thể truyền một giá trị đơn giản cho phương thức `takeUntil` để lấy các mục

cho đến khi tìm thấy giá trị đã cho:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(3);

$subset->all();

// [1, 2]
```

> **Chú ý:** Nếu không tìm thấy giá trị đã cho hoặc callback không bao giờ trả về true, thì phương thức `takeUntil` sẽ trả về tất cả các mục trong collection.

Phương thức `takeWhile`

Phương thức `takeWhile` trả về các mục trong collection cho đến khi callback đã cho trả về `false`:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeWhile(function ($item) {
  return $item < 3;
});

$subset->all();

// [1, 2]
```

> **Chú ý:** Nếu callback không bao giờ trả về `false`, thì phương thức `takeWhile` sẽ trả về tất cả các mục trong collection.

Phương thức `tap`

Phương thức `tap` sẽ truyền collection tới callback đã cho, cho phép bạn "khai thác" vào collection tại một điểm cụ thể và thực hiện điều gì đó với các mục trong khi không ảnh hưởng đến chính collection. Collection sau đó được trả về theo phương thức `tap`:

```php
collect([2, 4, 3, 1, 5])
  ->sort()
  ->tap(function ($collection) {
    Log::debug('Values after sorting', $collection->values()->all());
  })
  ->shift();

// 1
```

Phương thức `times`

Phương thức tĩnh `times` sẽ tạo ra một collection mới bằng cách gọi một hàm đã cho với một số lần được chỉ định:

```php
$collection = Collection::times(10, function ($number) {
  return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

Phương thức `toArray`

Phương thức `toArray` chuyển đổi collection thành một mảng PHP thuần túy. Nếu giá trị của collection là mô hình Eloquent, thì các model này cũng sẽ được chuyển đổi thành mảng:

```php
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
  [
    ['name' => 'Desk', 'price' => 200],
  ]
```

```
*/
```

> **Chú ý:** `toArray` cũng chuyển đổi tất cả các đối tượng lồng nhau của collection là một đối tượng của một mảng `Arrayable`. Nếu bạn muốn lấy mảng thô làm cơ sở cho collection, hãy sử dụng phương thức `all` này để thay thế.

Phương thức `toJson`

Phương thức `toJson` chuyển đổi collection thành một chuỗi được tuần tự hóa JSON:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name":"Desk", "price":200}'
```

Phương thức `transform`

Phương thức `transform` sẽ lặp qua collection và callback đã cho với từng mục trong collection. Các mục trong collection sẽ được thay thế bằng các giá trị được trả về bởi callback:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {
  return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```

> **Chú ý:** Không giống như hầu hết các phương thức collection khác, transform sẽ sửa đổi chính collection đó. Nếu bạn muốn tạo một collection mới, hãy sử dụng phương thức `map` này.

Phương thức `undot`

Phương thức `undot` mở rộng collection đơn chiều sử dụng ký hiệu "dấu chấm" thành collection đa chiều:

```
$person = collect([
  'name.first_name' => 'Marie',
  'name.last_name' => 'Valentine',
  'address.line_1' => '2992 Eagle Drive',
  'address.line_2' => '',
  'address.suburb' => 'Detroit',
  'address.state' => 'MI',
  'address.postcode' => '48219'
])

$person = $person->undot();

$person->toArray();

/*
  [
    "name" => [
      "first_name" => "Marie",
      "last_name" => "Valentine",
    ],
    "address" => [
      "line_1" => "2992 Eagle Drive",
      "line_2" => "",
      "suburb" => "Detroit",
      "state" => "MI",
      "postcode" => "48219",
    ],
  ]
*/
```

Phương thức `union`

Phương thức `union` thêm mảng đã cho vào collection. Nếu mảng đã cho chứa các khóa đã có trong collection ban đầu, thì các giá trị của collection ban đầu sẽ được ưu tiên:

```
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['d']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

Phương thức `unique`

Phương thức `unique` trả về tất cả các mục duy nhất trong collection. Collection được trả về giữ các khóa mảng ban đầu, vì vậy trong ví dụ sau, chúng ta sẽ sử dụng phương pháp `values` để đặt lại các khóa thành các chỉ mục được đánh số liên tiếp:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

Khi xử lý các mảng hoặc đối tượng lồng nhau, bạn có thể chỉ định khóa được sử dụng để xác định tính thống nhất:

```
$collection = collect([
  ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
  ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
  ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
  ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
  ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);
```

```php
$unique = $collection->unique('brand');

$unique->values()->all();

/*
  [
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
  ]
*/
```

Cuối cùng, bạn cũng có thể truyền hàm của riêng mình cho phương thức `unique` để chỉ định giá trị nào sẽ xác định tính thống nhất của một mục:

```php
$unique = $collection->unique(function ($item) {
  return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
  [
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
  ]
*/
```

Phương thức `unique` sử dụng so sánh "lỏng lẻo" khi kiểm tra giá trị mục, nghĩa là một chuỗi có giá trị nguyên sẽ được coi là bằng một số nguyên có cùng giá trị. Sử dụng phương thức `uniqueStrict` để lọc bằng cách so sánh "nghiêm ngặt".

> Hoạt động của phương thức này được sửa đổi khi sử dụng Eloquent Collections.

Phương thức `uniqueStrict`

Phương thức này có cùng cách thức với phương thức `unique`; tuy nhiên, tất cả các giá trị được so sánh bằng cách sử dụng so sánh "nghiêm ngặt".

Phương thức `unless`

Phương thức `unless` sẽ thực hiện callback đã cho trừ khi đối số đầu tiên được cung cấp cho phương thức đánh giá là true:

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {
  return $collection->push(4);
});

$collection->unless(false, function ($collection) {
  return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

Một callback thứ hai có thể được chuyển cho phương thức `unless`. Callback thứ hai sẽ được thực thi khi đối số đầu tiên được cung cấp cho phương thức `unless` đánh giá là true:

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {
  return $collection->push(4);
}, function ($collection) {
  return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

Đối với nghịch đảo của `unless`, hãy xem phương thức `when`.

Phương thức `unlessEmpty`

Bí danh cho phương thức `whenNotEmpty`.

Phương thức `unlessNotEmpty`

Bí danh cho phương thức `whenEmpty`.

Phương thức `unwrap`

Phương thức tĩnh `unwrap` trả về các mục cơ bản của collection từ giá trị đã cho khi áp dụng:

```
Collection::unwrap(collect('John Doe'));

// ['John Doe']

Collection::unwrap(['John Doe']);

// ['John Doe']

Collection::unwrap('John Doe');

// 'John Doe'
```

Phương thức `values`

Phương thức `values` sẽ trả về một collection mới với các khóa được đặt lại thành các số nguyên liên tiếp:

```
$collection = collect([
  10 => ['product' => 'Desk', 'price' => 200],
  11 => ['product' => 'Desk', 'price' => 200],
]);
```

```php
$values = $collection->values();

$values->all();

/*
  [
    0 => ['product' => 'Desk', 'price' => 200],
    1 => ['product' => 'Desk', 'price' => 200],
  ]
*/
```

Phương thức `when`

Phương thức `when` sẽ thực thi callback đã cho khi đối số đầu tiên được cung cấp cho phương thức đánh giá là true:

```php
$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {
  return $collection->push(4);
});

$collection->when(false, function ($collection) {
  return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 4]
```

Một callback thứ hai có thể được truyền cho phương thức `when`. Callback thứ hai sẽ được thực thi khi đối số đầu tiên được cung cấp cho phương thức `when` đánh giá là `false`:

```php
$collection = collect([1, 2, 3]);
```

```php
$collection->when(false, function ($collection) {
  return $collection->push(4);
}, function ($collection) {
  return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

Đối với nghịch đảo của `when`, hãy xem phương pháp `unless`.

Phương thức `whenEmpty`

Phương thức `whenEmpty` sẽ thực hiện callback đã cho khi collection trống:

```php
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function ($collection) {
  return $collection->push('Adam');
});

$collection->all();

// ['Michael', 'Tom']


$collection = collect();

$collection->whenEmpty(function ($collection) {
  return $collection->push('Adam');
});

$collection->all();

// ['Adam']
```

Callback thứ hai có thể được truyền tới phương thức `whenEmpty` sẽ được thực thi khi collection không trống:

```
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function ($collection) {
  return $collection->push('Adam');
}, function ($collection) {
  return $collection->push('Taylor');
});

$collection->all();

// ['Michael', 'Tom', 'Taylor']
```

Đối với nghịch đảo của `whenEmpty`, hãy xem phương thức `whenNotEmpty`.

Phương thức `whenNotEmpty`

Phương thức `whenNotEmpty` sẽ thực hiện callback đã cho khi bộ sưu tập không trống:

```
$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function ($collection) {
  return $collection->push('adam');
});

$collection->all();

// ['michael', 'tom', 'adam']


$collection = collect();

$collection->whenNotEmpty(function ($collection) {
  return $collection->push('adam');
});
```

```php
$collection->all();

// []
```

Một callback thứ hai có thể được truyền vào phương thức `whenNotEmpty` sẽ được thực thi khi collection trống:

```php
$collection = collect();

$collection->whenNotEmpty(function ($collection) {
  return $collection->push('adam');
}, function ($collection) {
  return $collection->push('taylor');
});

$collection->all();

// ['taylor']
```

Đối với nghịch đảo của `whenNotEmpty`, hãy xem phương thức `whenEmpty`.

Phương thức `where`

Phương thức `where` sẽ lọc collection theo một cặp khóa/giá trị nhất định:

```php
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 100],
  ['product' => 'Bookcase', 'price' => 150],
  ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();
```

```
/*
  [
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
  ]
*/
```

Phương thức `where` sử dụng so sánh "lỏng lẻo" khi kiểm tra giá trị mục, nghĩa là một chuỗi có giá trị nguyên sẽ được coi là bằng một số nguyên có cùng giá trị. Sử dụng phương thức `whereStrict` để lọc bằng cách so sánh "nghiêm ngặt".

Theo tùy chọn, bạn có thể truyền một toán tử so sánh làm tham số thứ hai.

```
$collection = collect([
  ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
  ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
  ['name' => 'Sue', 'deleted_at' => null],
]);

$filtered = $collection->where('deleted_at', '!=', null);

$filtered->all();

/*
  [
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
  ]
*/
```

Phương thức `whereStrict`

Phương thức này có cùng cách thức hoạt động với phương thức `where`; tuy nhiên, tất cả các giá trị được so sánh bằng cách sử dụng so sánh "nghiêm ngặt".

Phương thức `whereBetween`

Phương thức `whereBetween` sẽ lọc collection bằng cách xác định xem một giá trị mục được chỉ định có nằm trong một phạm vi nhất định hay không:

```
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 80],
  ['product' => 'Bookcase', 'price' => 150],
  ['product' => 'Pencil', 'price' => 30],
  ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereBetween('price', [100, 200]);

$filtered->all();

/*
  [
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
  ]
*/
```

Phương thức `whereIn`

Phương thức `whereIn` sẽ loại bỏ các phần tử khỏi collection không có giá trị mục cụ thể được chứa trong mảng đã cho:

```
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 100],
  ['product' => 'Bookcase', 'price' => 150],
  ['product' => 'Door', 'price' => 100],
]);
```

```php
$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
  [
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Bookcase', 'price' => 150],
  ]
*/
```

Phương thức `whereIn` sử dụng so sánh "lỏng lẻo" khi kiểm tra giá trị mục, nghĩa là một chuỗi có giá trị nguyên sẽ được coi là bằng một số nguyên có cùng giá trị. Sử dụng phương thức `whereInStrict` để lọc bằng cách so sánh "nghiêm ngặt".

Phương thức `whereInStrict`

Phương thức này có cùng cách thức hoạt động với phương thức `whereIn`; tuy nhiên, tất cả các giá trị được so sánh bằng cách sử dụng so sánh "nghiêm ngặt".

Phương thức `whereInstanceOf`

Phương thức `whereInstanceOf` sẽ lọc collection theo một loại lớp nhất định:

```php
use App\Models\User;
use App\Models\Post;

$collection = collect([
  new User,
  new User,
  new Post,
]);

$filtered = $collection->whereInstanceOf(User::class);

$filtered->all();
```

```
// [App\Models\User, App\Models\User]
```

Phương thức `whereNotBetween`

Phương thức `whereNotBetween` sẽ lọc collection bằng cách xác định xem một giá trị được chỉ định có nằm ngoài một phạm vi nhất định hay không:

```
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 80],
  ['product' => 'Bookcase', 'price' => 150],
  ['product' => 'Pencil', 'price' => 30],
  ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/*
  [
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Pencil', 'price' => 30],
  ]
*/
```

Phương thức `whereNotIn`

Phương thức `whereNotIn` sẽ loại bỏ các phần tử khỏi collection có giá trị mục được chỉ định được chứa trong mảng đã cho:

```
$collection = collect([
  ['product' => 'Desk', 'price' => 200],
  ['product' => 'Chair', 'price' => 100],
  ['product' => 'Bookcase', 'price' => 150],
  ['product' => 'Door', 'price' => 100],
```

```
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/*
  [
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
  ]
*/
```

Phương thức `whereNotIn` sử dụng so sánh "lỏng lẻo" khi kiểm tra giá trị mục, nghĩa là một chuỗi có giá trị nguyên sẽ được coi là bằng một số nguyên có cùng giá trị. Sử dụng phương thức `whereNotInStrict` để lọc bằng cách so sánh "nghiêm ngặt".

Phương thức `whereNotInStrict`

Phương thức này có cùng cách thức hoạt động với phương thức `whereNotIn`; tuy nhiên, tất cả các giá trị được so sánh bằng cách sử dụng so sánh "nghiêm ngặt".

Phương thức `whereNotNull`

Phương thức `whereNotNull` sẽ trả về các mục từ collection mà khóa key đã cho không phải là `null`:

```
$collection = collect([
  ['name' => 'Desk'],
  ['name' => null],
  ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNotNull('name');

$filtered->all();
```

```
/*
  [
    ['name' => 'Desk'],
    ['name' => 'Bookcase'],
  ]
*/
```

Phương thức `whereNull`

Phương thức `whereNull` sẽ trả về các mục từ collection có khóa đã cho là `null`:

```
$collection = collect([
  ['name' => 'Desk'],
  ['name' => null],
  ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNull('name');

$filtered->all();

/*
  [
    ['name' => null],
  ]
*/
```

Phương thức `wrap`

Phương thức tĩnh `wrap` sẽ bao bọc giá trị đã cho trong một collection khi có thể:

```
use Illuminate\Support\Collection;

$collection = Collection::wrap('John Doe');
```

```php
$collection->all();

// ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all();

// ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all();

// ['John Doe']
```

Phương thức `zip`

Phương thức `zip` này hợp nhất các giá trị của mảng đã cho với các giá trị của tập hợp ban đầu tại chỉ mục tương ứng của chúng:

```php
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Higher order messages

Collection cũng hỗ trợ kỹ thuật "higher order messages", được gọi tắt cho việc thực thi các hành động phổ biến trên các collection. Các phương thức collection có cung cấp higher order message gồm có: `average`, `avg`, `contains`, `each`, `every`, `filter`, `first`, `flatMap`, `groupBy`, `keyBy`, `map`, `max`, `min`, `partition`, `reject`, `skipUntil`, `skipWhile`, `some`, `sortBy`, `sortByDesc`, `sum`, `takeUntil`, `takeWhile`, và `unique`.

Mỗi higher order messages có thể được truy cập như một thuộc tính động trên một đối tượng collection. Ví dụ: hãy sử dụng từng higher order message để gọi một phương thức trên mỗi đối tượng trong một collection:

```
use App\Models\User;

$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Tương tự như vậy, chúng tôi có thể sử dụng `sum` higher order message để thu thập tổng số "phiếu bầu" cho collection người dùng:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Collection Lười

Giới thiệu

> **Chú ý:** Trước khi tìm hiểu thêm về collection lười của Laravel, hãy dành chút thời gian để tự làm quen với PHP generator.

Để bổ sung cho class `Collection` vốn đã mạnh mẽ, class này sử dụng các chương trình tạo `LazyCollection` của PHP để cho phép bạn làm việc với các tập dữ liệu rất lớn trong khi vẫn giữ mức sử dụng bộ nhớ thấp.

Ví dụ: hãy tưởng tượng ứng dụng của bạn cần xử lý tập tin nhật ký nhiều gigabyte trong khi tận dụng các phương thức thu thập của Laravel để phân tích cú pháp nhật ký. Thay vì đọc toàn bộ tập tin vào bộ nhớ cùng một lúc, collection lười sẽ có thể được sử dụng để chỉ giữ một phần nhỏ của tập tin trong bộ nhớ tại một thời điểm nhất định:

```
use App\Models\LogEntry;
use Illuminate\Support\LazyCollection;
```

```
LazyCollection::make(function () {
  $handle = fopen('log.txt', 'r');

  while (($line = fgets($handle)) !== false) {
    yield $line;
  }
})->chunk(4)->map(function ($lines) {
  return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
  // Process the log entry...
});
```

Hoặc, hãy tưởng tượng bạn cần lặp lại 10.000 model Eloquent. Khi sử dụng collection Laravel truyền thống, tất cả 10.000 model Eloquent phải được tải vào bộ nhớ cùng một lúc:

```
use App\Models\User;

$users = User::all()->filter(function ($user) {
  return $user->id > 500;
});
```

Tuy nhiên, phương thức `cursor` của trình tạo truy vấn sẽ trả về một đối tượng `LazyCollection`. Điều này cho phép bạn vẫn chỉ chạy một truy vấn duy nhất đối với cơ sở dữ liệu nhưng cũng chỉ giữ một model Eloquent được tải trong bộ nhớ tại một thời điểm. Trong ví dụ này, lệnh callback của `filter` sẽ không được thực thi cho đến khi chúng tôi thực sự lặp lại từng người dùng riêng lẻ, cho phép giảm đáng kể mức sử dụng bộ nhớ:

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
  return $user->id > 500;
});

foreach ($users as $user) {
  echo $user->id;
}
```

Tạo collection lười

Để tạo một đối tượng collection lười, bạn nên truyền một hàm của PHP generator vào phương thức `make` của collection:

```
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
  $handle = fopen('log.txt', 'r');

  while (($line = fgets($handle)) !== false) {
    yield $line;
  }
});
```

Hợp đồng Enumerable

Hầu như tất cả các phương thức có sẵn trên class `Collection` cũng có sẵn trên class `LazyCollection`. Cả hai class này đều thực thi contract `Illuminate\Support\Enumerable`, sẽ gồm các phương thức sau:

- `all`
- `average`
- `avg`
- `chunk`
- `chunkWhile`
- `collapse`
- `collect`
- `combine`
- `concat`
- `contains`
- `containsStrict`
- `count`
- `countBy`
- `crossJoin`
- `dd`
- `diff`
- `diffAssoc`
- `diffKeys`
- `dump`

- duplicates
- duplicatesStrict
- each
- eachSpread
- every
- except
- filter
- first
- firstWhere
- flatMap
- flatten
- flip
- forPage
- get
- groupBy
- has
- implode
- intersect
- intersectByKeys
- isEmpty
- isNotEmpty
- join
- keyBy
- keys
- last
- macro
- make
- map
- mapInto
- mapSpread
- mapToGroups
- mapWithKeys
- max
- median
- merge
- mergeRecursive
- min
- mode
- nth
- only
- pad
- partition

- pipe
- pluck
- random
- reduce
- reject
- replace
- replaceRecursive
- reverse
- search
- shuffle
- skip
- slice
- some
- sort
- sortBy
- sortByDesc
- sortKeys
- sortKeysDesc
- split
- sum
- take
- tap
- times
- toArray
- toJson
- union
- unique
- uniqueStrict
- unless
- unlessEmpty
- unlessNotEmpty
- unwrap
- values
- when
- whenEmpty
- whenNotEmpty
- where
- whereStrict
- whereBetween
- whereIn
- whereInStrict
- whereInstanceOf

- `whereNotBetween`
- `whereNotIn`
- `whereNotInStrict`
- `wrap`
- `zip`

> **Chú ý:** Các phương thức làm thay đổi tập hợp (chẳng hạn như, `shift`, `pop`, `prepend`, v.v.) không có sẵn trên lớp `LazyCollection`.

Các phương thức Collection lười

Ngoài các phương thức được xác định trong contract `Enumerable`, class `LazyCollection` còn chứa các phương thức sau:

Phương thức `takeUntilTimeout`

Phương thức `takeUntilTimeout` sẽ trả về một collection lười mới sẽ liệt kê các giá trị cho đến thời điểm được chỉ định. Sau thời gian đó, collection sẽ ngừng liệt kê:

```
$lazyCollection = LazyCollection::times(INF)
  ->takeUntilTimeout(now()->addMinute());

$lazyCollection->each(function ($number) {
  dump($number);

  sleep(1);
});

// 1
// 2
// ...
// 58
// 59
```

Để minh họa việc sử dụng phương pháp này, hãy tưởng tượng một ứng dụng gửi hóa đơn từ cơ sở dữ liệu bằng cách sử dụng con trỏ. Bạn có thể xác định một tác vụ đã lên lịch chạy 15 phút một lần và chỉ xử lý hóa đơn trong tối đa 14 phút:

```php
use App\Models\Invoice;
use Illuminate\Support\Carbon;

Invoice::pending()->cursor()
  ->takeUntilTimeout(
    Carbon::createFromTimestamp(LARAVEL_START)->add(14, 'minutes')
  )
  ->each(fn ($invoice) => $invoice->submit());
```

Phương thức `tapEach`

Mặc dù phương thức `each` gọi callback đã cho cho từng mục trong collection ngay lập tức, nhưng phương thức `tapEach` chỉ gọi callback đã cho vì các mục đang được kéo ra khỏi danh sách lần lượt:

```php
// Nothing has been dumped so far...
$lazyCollection = LazyCollection::times(INF)->tapEach(function ($value) {
  dump($value);
});

// Three items are dumped...
$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

Phương thức `remember`

Phương thức `remember` sẽ trả về một collection lười mới sẽ nhớ bất kỳ giá trị nào đã được liệt kê và sẽ không truy xuất lại chúng trong các lần liệt kê collection tiếp theo:

```php
// No query has been executed yet...
$users = User::cursor()->remember();
```

```php
// The query is executed...
// The first 5 users are hydrated from the database...
$users->take(5)->all();

// First 5 users come from the collection's cache...
// The rest are hydrated from the database...
$users->take(20)->all();
```