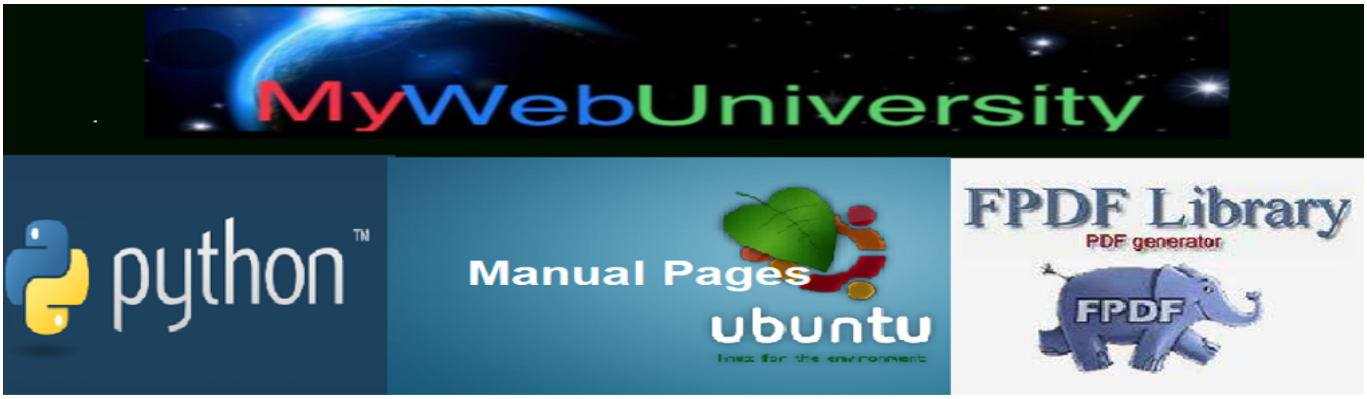

Rocky Enterprise Linux 9.2 Manual Pages on command 'Hash::MultiValue.3pm'

$ man Hash::MultiValue.3pm

Hash::MultiValue(3pm) User Contributed Perl Documentation Hash::MultiValue(3pm)

NAME

Hash::MultiValue - Store multiple values per key

SYNOPSIS

```
use Hash::MultiValue;
my $hash = Hash::MultiValue->new(
    foo => 'a',
    foo => 'b',
    bar => 'baz',
);
# $hash is an object, but can be used as a hashref and DWIMs!
my $foo = $hash->{foo};         # 'b' (the last entry)
my $foo = $hash->get('foo');    # 'b' (always, regardless of context)
my @foo = $hash->get_all('foo'); # ('a', 'b')
keys %$hash; # ('foo', 'bar')    not guaranteed to be ordered
$hash->keys; # ('foo', 'foo', 'bar') guaranteed to be ordered
```

DESCRIPTION

Hash::MultiValue is an object (and a plain hash reference) that may contain multiple values per key, inspired by MultiDict of WebOb.

RATIONALE

In a typical web application, the request parameters (a.k.a CGI parameters) can be single value or multi values. Using CGI.pm style "param" is one way to deal with this problem (and it is good, as long as you're aware of its list context gotcha), but there's another

approach to convert parameters into a hash reference, like Catalyst's "$c->req->parameters" does, and it sucks.

Why? Because the value could be just a scalar if there is one value and an array ref if there are multiple, depending on user input rather than how you code it. So your code should always be like this to be defensive:

```
my $p = $c->req->parameters;
my @maybe_multi = ref $p->{m} eq 'ARRAY' ? @{$p->{m}} : ($p->{m});
my $must_single = ref $p->{m} eq 'ARRAY' ? $p->{m}->[0] : $p->{m};
```

Otherwise you'll get a random runtime exception of Can't use string as an ARRAY ref or get stringified array ARRAY(0xXXXXXXXXX) as a string, depending on user input and that is miserable and insecure.

This module provides a solution to this by making it behave like a single value hash reference, but also has an API to get multiple values on demand, explicitly.

HOW THIS WORKS

The object returned by "new" is a blessed hash reference that contains the last entry of the same key if there are multiple values, but it also keeps the original pair state in the object tracker (a.k.a inside out objects) and allows you to access the original pairs and multiple values via the method calls, such as "get_all" or "flatten".

This module does not use "tie" or overload and is quite fast.

Yes, there is Tie::Hash::MultiValue and this module tries to solve exactly the same problem, but using a different implementation.

UPDATING CONTENTS

When you update the content of the hash, DO NOT UPDATE using the hash reference interface: this won't write through to the tracking object.

```
my $hash = Hash::MultiValue->new(...);
# WRONG
$hash->{foo} = 'bar';
delete $hash->{foo};
# Correct
$hash->add(foo => 'bar');
$hash->remove('foo');
```

See below for the list of updating methods.

METHODS

new

$hash = Hash::MultiValue->new(@pairs);

Creates a new object that can be treated as a plain hash reference as well.

get

$value = $hash->get($key);

$value = $hash->{$key};

Returns a single value for the given $key. If there are multiple values, the last one (not first one) is returned. See below for why.

Note that this always returns the single element as a scalar, regardless of its context, unlike CGI.pm's "param" method etc.

get_one

$value = $hash->get_one($key);

Returns a single value for the given $key. This method croaks if there is no value or multiple values associated with the key, so you should wrap it with eval or modules like Try::Tiny.

get_all

@values = $hash->get_all($key);

Returns a list of values for the given $key. This method always returns a list regardless of its context. If there is no value attached, the result will be an empty list.

keys

@keys = $hash->keys;

Returns a list of all keys, including duplicates (see the example in the "SYNOPSIS"). If you want only unique keys, use "keys %$hash", as normal.

values

@values = $hash->values;

Returns a list of all values, in the same order as "$hash->keys".

set

$hash->set($key [, $value ...]);

Changes the stored value(s) of the given $key. This removes or adds pairs as necessary to store the new list but otherwise preserves order of existing pairs. "$hash->{$key}" is updated to point to the last value.

add

```
$hash->add($key, $value [, $value ... ]);
```

Appends a new value to the given $key. This updates the value of "$hash->{$key}" as well so it always points to the last value.

remove

```
$hash->remove($key);
```

Removes a key and associated values for the given $key.

clear

```
$hash->clear;
```

Clears the hash to be an empty hash reference.

flatten

```
@pairs = $hash->flatten;
```

Gets pairs of keys and values. This should be exactly the same pairs which are given to "new" method unless you updated the data.

each

```
$hash->each($code);
# e.g.
$hash->each(sub { print "$_[0] = $_[1]\n" });
```

Calls $code once for each "($key, $value)" pair. This is a more convenient alternative to calling "flatten" and then iterating over it two items at a time.

Inside $code, $_ contains the current iteration through the loop, starting at 0. For example:

```
$hash = Hash::MultiValue->new(a => 1, b => 2, c => 3, a => 4);
$hash->each(sub { print "$_: $_[0] = $_[1]\n" });
# 0: a = 1
# 1: b = 2
# 2: c = 3
# 3: a = 4
```

Be careful not to change @_ inside your coderef! It will update the tracking object but not the plain hash. In the future, this limitation may be removed.

clone

```
$new = $hash->clone;
```

Creates a new Hash::MultiValue object that represents the same data, but obviously not sharing the reference. It's identical to:

```
    $new = Hash::MultiValue->new($hash->flatten);
```

as_hashref

```
    $copy = $hash->as_hashref;
```

Creates a new plain (unblessed) hash reference where a value is a single scalar. It's identical to:

```
    $copy = +{%$hash};
```

as_hashref_mixed, mixed

```
    $mixed = $hash->as_hashref_mixed;
    $mixed = $hash->mixed;
```

Creates a new plain (unblessed) hash reference where the value is a single scalar, or an array ref when there are multiple values for a same key. Handy to create a hash reference that is often used in web application frameworks request objects such as Catalyst. Ths method does exactly the opposite of "from_mixed".

as_hashref_multi, multi

```
    $multi = $hash->as_hashref_multi;
    $multi = $hash->multi;
```

Creates a new plain (unblessed) hash reference where values are all array references, regardless of there are single or multiple values for a same key.

from_mixed

```
    $hash = Hash::MultiValue->from_mixed({
        foo => [ 'a', 'b' ],
        bar => 'c',
    });
```

Creates a new object out of a hash reference where the value is single or an array ref depending on the number of elements. Handy to convert from those request objects used in web frameworks such as Catalyst. This method does exactly the opposite of "as_hashref_mixed".

WHY LAST NOT FIRST?

You might wonder why this module uses the last value of the same key instead of first. There's no strong reasoning on this decision since one is as arbitrary as the other, but this is more consistent to what Perl does:

```
  sub x {
      return ('a', 'b', 'c');
```

```
}
my $x = x(); # $x = 'c'
my %a = ( a => 1 );
my %b = ( a => 2 );
my %m = (%a, %b); # $m{a} = 2
```

When perl gets a list in a scalar context it gets the last entry. Also if you merge hashes having a same key, the last one wins.

NOTES ON ref

If you pass this MultiValue hash object to some upstream functions that you can't control and does things like:

```
if (ref $args eq 'HASH') {
    ...
}
```

because this is a blessed hash reference it doesn't match and would fail. To avoid that you should call "as_hashref" to get a finalized (= non-blessed) hash reference.

You can also use UNIVERSAL::ref to make it work magically:

```
use UNIVERSAL::ref;    # before loading Hash::MultiValue
use Hash::MultiValue;
```

and then all "ref" calls to Hash::MultiValue objects will return HASH.

THREAD SAFETY

Prior to version 0.09, this module wasn't safe in a threaded environment, including win32 fork() emulation. Versions newer than 0.09 is considered thread safe.

AUTHOR

Tatsuhiko Miyagawa <miyagawa@bulknews.net>

Aristotle Pagaltzis

Hans Dieter Pearcey

Thanks to Michael Peters for the suggestion to use inside-out objects instead of tie.

LICENSE

SEE ALSO

- <http://pythonpaste.org/webob/#multidict>
- Tie::Hash::MultiValue