NAME

Digest - Modules that calculate message digests

SYNOPSIS

```
$md5  = Digest->new("MD5");
$sha1 = Digest->new("SHA-1");
$sha256 = Digest->new("SHA-256");
$sha384 = Digest->new("SHA-384");
$sha512 = Digest->new("SHA-512");

$hmac = Digest->HMAC_MD5($key);
```

DESCRIPTION

The `Digest::` modules calculate digests, also called "fingerprints" or "hashes", of some data, called a message. The digest is (usually) some small/fixed size string. The actual size of the digest depend of the algorithm used. The message is simply a sequence of arbitrary bytes or bits.

An important property of the digest algorithms is that the digest is *likely* to change if the message change in some way. Another property is that digest functions are one-way functions, that is it should be *hard* to find a message that correspond to some given digest. Algorithms differ in how "likely" and how "hard", as well as how efficient they are to compute.

Note that the properties of the algorithms change over time, as the algorithms are analyzed and machines grow faster. If your application for instance depends on it being "impossible" to generate the same digest for a different message it is wise to make it easy to plug in stronger algorithms as the one used grow weaker. Using the interface documented here should make it easy to change algorithms later.

All `Digest::` modules provide the same programming interface. A functional interface for simple use, as well as an object oriented interface that can handle messages of arbitrary length and which can read files directly.

The digest can be delivered in three formats:

binary

: This is the most compact form, but it is not well suited for printing or embedding in places that can't handle arbitrary data.

hex

: A twice as long string of lowercase hexadecimal digits.

base64

: A string of portable printable characters. This is the base64 encoded representation of the digest with any trailing padding removed. The string will be about 30% longer than the binary version. *MIME::Base64* tells you more about this encoding.

The functional interface is simply importable functions with the same name as the algorithm. The functions take the message as argument and return the digest. Example:

```
use Digest::MD5 qw(md5);
$digest = md5($message);
```

There are also versions of the functions with "_hex" or "_base64" appended to the name, which returns the digest in the indicated form.

OO INTERFACE

The following methods are available for all `Digest::` modules:

$ctx = Digest->XXX($arg,...)

$ctx = Digest->new(XXX => $arg,...)

$ctx = Digest::XXX->new($arg,...)

The constructor returns some object that encapsulate the state of the message-digest algorithm. You can add data to the object and finally ask for the digest. The "XXX" should of course be replaced by the proper name of the digest algorithm you want to use.

The two first forms are simply syntactic sugar which automatically load the right module on first use. The second form allow you to use algorithm names which contains letters which are not legal perl identifiers, e.g. "SHA-1". If no implementation for the given algorithm can be found, then an exception is raised.

If new() is called as an instance method (i.e. $ctx->new) it will just reset the state the object to the state of a newly created object. No new object is created in this case, and the return value is the reference to the object (i.e. $ctx).

$other_ctx = $ctx->clone

The clone method creates a copy of the digest state object and returns a reference to the copy.

$ctx->reset

This is just an alias for $ctx->new.

$ctx->add($data)

$ctx->add($chunk1, $chunk2, ...)

The string value of the $data provided as argument is appended to the message we calculate the digest for. The return value is the $ctx object itself.

If more arguments are provided then they are all appended to the message, thus all these lines will have the same effect on the state of the $ctx object:

```
$ctx->add("a"); $ctx->add("b"); $ctx->add("c");
$ctx->add("a")->add("b")->add("c");
$ctx->add("a", "b", "c");
$ctx->add("abc");
```

Most algorithms are only defined for strings of bytes and this method might therefore croak if the provided arguments contain chars with ordinal number above 255.

$ctx->addfile($io_handle)

The $io_handle is read until EOF and the content is appended to the message we calculate the digest for. The return value is the $ctx object itself.

The addfile() method will croak() if it fails reading data for some reason. If it croaks it is unpredictable what the state of the $ctx object will be in. The addfile() method might have been able to read the file partially before it failed. It is probably wise to discard or reset the $ctx object if this occurs.

In most cases you want to make sure that the $io_handle is in "binmode" before you pass it as argument to the addfile() method.

$ctx->add_bits($data, $nbits)

$ctx->add_bits($bitstring)

The add_bits() method is an alternative to add() that allow partial bytes to be appended to the message. Most users should just ignore this method as partial bytes is very unlikely to be of

any practical use.

The two argument form of add_bits() will add the first $nbits bits from $data. For the last potentially partial byte only the high order `$nbits % 8` bits are used. If $nbits is greater than `length($data) * 8`, then this method would do the same as `$ctx->add($data)`.

The one argument form of add_bits() takes a $bitstring of "1" and "0" chars as argument. It's a shorthand for `$ctx->add_bits(pack("B*", $bitstring), length($bitstring))`.

The return value is the $ctx object itself.

This example shows two calls that should have the same effect:

```
$ctx->add_bits("111100001010");
$ctx->add_bits("\xF0\xA0", 12);
```

Most digest algorithms are byte based and for these it is not possible to add bits that are not a multiple of 8, and the add_bits() method will croak if you try.

$ctx->digest

Return the binary digest for the message.

Note that the `digest` operation is effectively a destructive, read-once operation. Once it has been performed, the $ctx object is automatically `reset` and can be used to calculate another digest value. Call $ctx->clone->digest if you want to calculate the digest without resetting the digest state.

$ctx->hexdigest

Same as $ctx->digest, but will return the digest in hexadecimal form.

$ctx->b64digest

Same as $ctx->digest, but will return the digest as a base64 encoded string.

Digest speed

This table should give some indication on the relative speed of different algorithms. It is sorted by throughput based on a benchmark done with of some implementations of this API:

```
 Algorithm      Size    Implementation                  MB/s

 MD4             128    Digest::MD4 v1.3              165.0
 MD5             128    Digest::MD5 v2.33              98.8
 SHA-256         256    Digest::SHA2 v1.1.0            66.7
 SHA-1           160    Digest::SHA v4.3.1             58.9
 SHA-1           160    Digest::SHA1 v2.10             48.8
 SHA-256         256    Digest::SHA v4.3.1             41.3
 Haval-256       256    Digest::Haval256 v1.0.4        39.8
 SHA-384         384    Digest::SHA2 v1.1.0            19.6
 SHA-512         512    Digest::SHA2 v1.1.0            19.3
 SHA-384         384    Digest::SHA v4.3.1             19.2
 SHA-512         512    Digest::SHA v4.3.1             19.2
 Whirlpool       512    Digest::Whirlpool v1.0.2       13.0
 MD2             128    Digest::MD2 v2.03               9.5

 Adler-32         32    Digest::Adler32 v0.03           1.3
 CRC-16           16    Digest::CRC v0.05               1.1
 CRC-32           32    Digest::CRC v0.05               1.1
 MD5             128    Digest::Perl::MD5 v1.5          1.0
 CRC-CCITT        16    Digest::CRC v0.05               0.8
```

These numbers was achieved Apr 2004 with ActivePerl-5.8.3 running under Linux on a P4 2.8 GHz CPU. The last 5 entries differ by being pure perl implementations of the algorithms, which explains why they are so slow.

SEE ALSO

Digest::Adler32, *Digest::CRC*, *Digest::Haval256*, *Digest::HMAC*, *Digest::MD2*, *Digest::MD4*, *Digest::MD5*, *Digest::SHA*, *Digest::SHA1*, *Digest::SHA2*, *Digest::Whirlpool*

New digest implementations should consider subclassing from *Digest::base*.

MIME::Base64

http://en.wikipedia.org/wiki/Cryptographic_hash_function

AUTHOR

Gisle Aas <gisle@aas.no>

The `Digest::` interface is based on the interface originally developed by Neil Winton for his `MD5` module.