How to generate a random string of a fixed length in golang?

Paul's solution provides a simple, general solution.

The question asks for the *"the fastest and simplest way"*. Let's address this. We'll arrive at our final, fastest code in an iterative manner. Benchmarking each iteration can be found at the end of the answer.

All the solutions and the benchmarking code can be found on the `Go Playground`. The code on the Playground is a test file, not an executable. You have to save it into a file named `XX_test.go` and run it with `go test -bench .`.

I. Improvements

1. Genesis (Runes)

As a reminder, the original, general solution we're improving is this:

```
func init() {
 rand.Seed(time.Now().UnixNano())
}
var letterRunes =
[]rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
func RandStringRunes(n int) string {
 b := make([]rune, n)
 for i := range b {
 b[i] = letterRunes[rand.Intn(len(letterRunes))]
 }
 return string(b)
}
```

2. Bytes

If the characters to choose from and assemble the random string contains only the uppercase and lowercase letters of the English alphabet, we can work with bytes only because the English alphabet letters map to bytes 1-to-1 in the UTF-8 encoding (which is how Go stores strings).

So instead of:

```
var letters =
[]rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

we can use:

```
var letters =
[]bytes("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

Or even better:

```
const letters =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Now this is already a big improvement: we could achieve it to be a `const` (there are `string` constants but there are no slice constants). As an extra gain, the expression `len(letters)` will also be a `const`! (The expression `len(s)` is constant if `s` is a string constant.)

And at what cost? Nothing at all. `string`s can be indexed which indexes its bytes, perfect, exactly what we want.

Our next destination looks like this:

```
const letterBytes = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
func RandStringBytes(n int) string {
 b := make([]byte, n)
 for i := range b {
 b[i] = letterBytes[rand.Intn(len(letterBytes))]
 }
 return string(b)
}
```

3. Remainder

This is much slower compared to `rand.Int63()` which produces a random number with 63 random bits.

So we could simply call `rand.Int63()` and use the remainder after dividing by `len(letterBytes)`:

```
func RandStringBytesRmndr(n int) string {
 b := make([]byte, n)
 for i := range b {
 b[i] = letterBytes[rand.Int63() %
int64(len(letterBytes))]
 }
 return string(b)
}
```

This works and is significantly faster, the disadvantage is that the probability of all the letters will not be exactly the same (assuming `rand.Int63()` produces all 63-bit numbers with equal probability). Although the distortion is extremely small as the number of letters `52` is much-much smaller than `1<<63 - 1`, so in practice this is perfectly fine.

To make this understand easier: let's say you want a random number in the range of `0..5`. Using 3 random bits, this would produce the numbers `0..1` with double probability than from the range `2..5`. Using 5 random bits, numbers in range `0..1` would occur with `6/32` probability and numbers in range `2..5` with `5/32` probability which is now closer to the desired. Increasing the number of bits makes this less significant, when reaching 63 bits, it is negligible.

4. Masking

Building on the previous solution, we can maintain the equal distribution of letters by using only as many of the lowest bits of the random number as many is required to represent the number of letters. So for example if we have 52 letters, it requires 6 bits to represent it: `52 = 110100b`. So we will only use the lowest 6 bits of the number returned by `rand.Int63()`. And to maintain equal distribution of letters, we only "accept" the number if it falls in the range `0..len(letterBytes)-1`. If the lowest bits are greater, we discard it and query a new random number.

Note that the chance of the lowest bits to be greater than or equal to `len(letterBytes)` is less than `0.5` in general (`0.25` on average), which means that even if this would be the case, repeating this "rare" case decreases the chance of not finding a good number. After `n` repetition, the chance that we sill don't have a good index is much less than `pow(0.5, n)`, and this is just an upper estimation. In case of 52 letters the chance that the 6 lowest bits are not good is only `(64-52)/64 = 0.19`; which means for example that chances to not have a good number after 10 repetition is `1e-8`.

So here is the solution:

```
const letterBytes = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
const (
 letterIdxBits = 6 // 6 bits to represent a letter index
 letterIdxMask = 1<<letterIdxBits - 1 // All 1-bits, as many as
letterIdxBits
)
func RandStringBytesMask(n int) string {
 b := make([]byte, n)
 for i := 0; i < n; {
 if idx := int(rand.Int63() & letterIdxMask); idx < len(letterBytes) {
 b[i] = letterBytes[idx]
 i++
 }
 }
 return string(b)
}
```

5. Masking Improved

The previous solution only uses the lowest 6 bits of the 63 random bits returned by `rand.Int63()`. This is a waste as getting the random bits is the slowest part of our algorithm.

If we have 52 letters, that means 6 bits code a letter index. So 63 random bits can designate `63/6 = 10` different letter indices. Let's use all those 10:

```
const letterBytes = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
const (
 letterIdxBits = 6 // 6 bits to represent a letter index
 letterIdxMask = 1<<letterIdxBits - 1 // All 1-bits, as many as
letterIdxBits
 letterIdxMax = 63 / letterIdxBits // # of letter indices fitting in 63
bits
)
func RandStringBytesMaskImpr(n int) string {
 b := make([]byte, n)
 // A rand.Int63() generates 63 random bits, enough for letterIdxMax
letters!
 for i, cache, remain := n-1, rand.Int63(), letterIdxMax; i >= 0; {
 if remain == 0 {
 cache, remain = rand.Int63(), letterIdxMax
 }
 if idx := int(cache & letterIdxMask); idx < len(letterBytes) {
 b[i] = letterBytes[idx]
 i--
 }
 cache >>= letterIdxBits
 remain--
 }
 return string(b)
}
```

6. Source

The **Masking Improved** is pretty good, not much we could improve on it. We could, but not worth the complexity.

Now let's find something else to improve. **The source of random numbers.**

There is a `crypto/rand` package which provides a `Read(b []byte)` function, so we could use that to get as many bytes with a single call as many we need. This wouldn't help in terms of performance as `crypto/rand` implements a cryptographically secure pseudorandom number generator so it's much slower.

So let's stick to the `math/rand` package. The `rand.Rand` uses a `rand.Source` as the source of random bits. `rand.Source` is an interface which specifies a `Int63() int64` method: exactly and the only thing we needed and used in our latest solution.

So we don't really need a `rand.Rand` (either explicit or the global, shared one of the `rand` package), a `rand.Source` is perfectly enough for us:

```
var src = rand.NewSource(time.Now().UnixNano())
func RandStringBytesMaskImprSrc(n int) string {
 b := make([]byte, n)
 // A src.Int63() generates 63 random bits, enough for letterIdxMax
characters!
 for i, cache, remain := n-1, src.Int63(), letterIdxMax; i >= 0; {
 if remain == 0 {
 cache, remain = src.Int63(), letterIdxMax
 }
 if idx := int(cache & letterIdxMask); idx < len(letterBytes) {
 b[i] = letterBytes[idx]
 i--
 }
 cache >>= letterIdxBits
 remain--
 }
 return string(b)
}
```

Also note that this last solution doesn't require you to initialize (seed) the global `Rand` of the `math/rand` package as that is not used (and our `rand.Source` is properly initialized / seeded).

II. Benchmark

All right, let's benchmark the different solutions.

```
BenchmarkRunes 1000000 1703 ns/op
BenchmarkBytes 1000000 1328 ns/op
BenchmarkBytesRmndr 1000000 1012 ns/op
BenchmarkBytesMask 1000000 1214 ns/op
BenchmarkBytesMaskImpr 5000000 395 ns/op
BenchmarkBytesMaskImprSrc 5000000 303
ns/op
```

Just by switching from runes to bytes, we immediately have **22%** performance gain.

Getting rid of `rand.Intn()` and using `rand.Int63()` instead gives another **24%** boost.

Masking (and repeating in case of big indices) slows down a little (due to repetition calls): **-20%**...

But when we make use of all (or most) of the 63 random bits (10 indices from one `rand.Int63()` call): that speeds up **3.4 times**.

And finally if we settle with a `rand.Source` instead of `rand.Rand`, we again gain **23%.**

Comparing the final to the initial solution: `RandStringBytesMaskImprSrc()` is **5.6 times faster** than `RandStringRunes()`.