Zapr :zap:

A logr implementation using Zap.

Usage

```go
import (
    "fmt"

    "go.uber.org/zap"
    "github.com/go-logr/logr"
    "github.com/go-logr/zapr"
)

func main() {
    var log logr.Logger

    zapLog, err := zap.NewDevelopment()
    if err != nil {
        panic(fmt.Sprintf("who watches the watchmen (%v)?", err))
    }
    log = zapr.NewLogger(zapLog)

    log.Info("Logr in action!", "the answer", 42)
}
```

Increasing Verbosity

Zap uses semantically named levels for logging (`DebugLevel` , `InfoLevel` , `WarningLevel` , ...). Logr uses arbitrary numeric levels. By default logr's `V(0)` is zap's `InfoLevel` and `V(1)` is zap's `DebugLevel` (which is numerically -1). Zap does not have named levels that are more verbose than `DebugLevel` , but it's possible to fake it.

As of zap v1.19.0 you can do something like the following in your setup code:

```go
    zc := zap.NewProductionConfig()
    zc.Level = zap.NewAtomicLevelAt(zapcore.Level(-2))
    z, err := zc.Build()
    if err != nil {
        // ...
    }
    log := zapr.NewLogger(z)
```

Zap's levels get more verbose as the number gets smaller and more important and the number gets larger (`DebugLevel` is -1, `InfoLevel` is 0, `WarnLevel` is 1, and so on).

The `-2` in the above snippet means that `log.V(2).Info()` calls will be active. `-3` would enable `log.V(3).Info()` , etc. Note that zap's levels are `int8` which means the most verbose level you can give it is

-128. The zapr implementation will cap `V()` levels greater than 127 to 127, so setting the zap level to -128 really means "activate all logs".

Implementation Details

For the most part, concepts in Zap correspond directly with those in logr.

Unlike Zap, all fields *must* be in the form of sugared fields -- it's illegal to pass a strongly-typed Zap field in a key position to any of the logging methods (`Log` , `Error`).