Trace

This package provides an interface for recording the latency of operations and logging details about all operations where the latency exceeds a limit.

Usage

To create a trace:

```
func doSomething() {
    opTrace := trace.New("operation", Field{Key: "fieldKey1", Value: "fieldValue1"})
    defer opTrace.LogIfLong(100 * time.Millisecond)
    // do something
}
```

To split an trace into multiple steps:

```
func doSomething() {
    opTrace := trace.New("operation")
    defer opTrace.LogIfLong(100 * time.Millisecond)
    // do step 1
    opTrace.Step("step1", Field{Key: "stepFieldKey1", Value: "stepFieldValue1"})
    // do step 2
    opTrace.Step("step2")
}
```

To nest traces:

```
func doSomething() {
    rootTrace := trace.New("rootOperation")
    defer rootTrace.LogIfLong(100 * time.Millisecond)

    func() {
        nestedTrace := rootTrace.Nest("nested", Field{Key: "nestedFieldKey1", Value:
"nestedFieldValue1"})
        defer nestedTrace.LogIfLong(50 * time.Millisecond)
        // do nested operation
    }()
}
```

Traces can also be logged unconditionally or introspected:

```
opTrace.TotalTime() // Duration since the Trace was created
opTrace.Log() // unconditionally log the trace
```

Using context.Context to nest traces

`context.Context` can be used to manage nested traces. Create traces by calling `trace.GetTraceFromContext(ctx).Nest`. This is safe even if there is no parent trace already in the context

because `(*(Trace)nil).Nest()` returns a top level trace.

```
func doSomething(ctx context.Context) {
    opTrace := trace.FromContext(ctx).Nest("operation") // create a trace, possibly
nested
    ctx = trace.ContextWithTrace(ctx, opTrace) // make this trace the parent trace
of the context
    defer opTrace.LogIfLong(50 * time.Millisecond)

    doSomethingElse(ctx)
}
```