Error handling framework

Usage instructions

The new error handling framework should be used in place of all calls to `fmt.Errorf()` or `Errors.new()`. Using this framework will provide error codes to check against as well as the option to generate a callstack that will be appended to the error message when logging.error is set to debug in `peer/core.yaml`.

Using the framework is simple and will only require an easy tweak to your code. First, you'll need to import github.com/hyperledger/fabric/core/errors in any code that uses this framework.

Let's take the following as an example:

```
fmt.Errorf("Error trying to connect to local peer: %s", err.Error())
```

For this error, we will simply call the constructor for `Error` and pass a component name, error message name, followed by the error message and any arguments to format into the message (note the component and message names are case insensitive and will be converted to uppercase):

```
err = errors.Error("Peer", "ConnectionError", "Error trying to connect to local peer: %s", err.Error())
```

For more critical errors for which a callstack would be beneficial, we can create the error as follows:

```
err = errors.ErrorWithCallstack("Peer", "ConnectionError", "Error trying to connect to local peer: %s", err.
Error())
```

If you are working with a team, try to agree upon a consistent component name to use. We may, in the future, add constants to allow searching for currently defined components for those using an editor with code completion capabilities; we are avoiding this for now to avoid merge conflict issues.

Setting stack trace to display for any CallStackError

The display of the stack trace with an error is tied to the logging level for the "error" module, which is initialized using `logging.error` in `core.yaml`. It can also be set dynamically for code running on the peer via CLI using "`peer logging setlevel error <log-level>`". The default level of "warning" will not display the stack trace; setting it to "debug" will display the stack trace for all errors that use this error handling framework.

General design

Module `core/errors/errors.go` implements an error handling framework (package errors) supporting multiple languages.

1. The module contains a general error interface, `CallStackError`:
    a. that implements Golang's error interface so it can be handled as a simple error
    b. that has a `GetStack()` function which returns the actual call stack as a string
    c. that has hierarchical errors with a component and a reason code
2. Call stacks are by default not captured by errors. To enable call stack capturing use `ErrorWithCallstack`

Functions

From `CallStackError` interface (what can I do with a `CallStackError`?):

1. `GetStack() string` - Gets the call stack (note that only errors instantiated with `ErrorWithCallstack` record the stack!)
2. `GetErrorCode() string` - Gets the error code as a pre-formatted string ("Utility-UtilityUnknownError").
3. `GetComponentCode() string` - Gets the component code which describes the originating module/component.
4. `GetReasonCode() string` - Gets the reason code which describes the lower level reason of the error.

To instantiate an error implementing `CallStackError` interface (how do I get a `CallStackError`?):

1. `Error(componentcode string, reasoncode string, message string, args...) CallStackError`
2. `ErrorWithCallstack(componentcode string, reasoncode string, message string, args...) CallStackError` **will also record the call stack**

Examples

Small examples are included in `errors_test.go`. It is also being used in `peer/clilogging` and `peer/common`.

Error handling best practices

A good introduction about errors and some potential ideas for error-handling in Go:

- https://blog.golang.org/errors-are-values
- https://blog.golang.org/defer-panic-and-recover
- https://github.com/mitchellh/panicwrap
- https://golang.org/src/runtime/traceback.go - how golang implements traceback

Directives for error handling in Fabric

1. if you are inclined to add a comment like `// this should never happen` => then you panic e.g.: fatal system error
2. if it is some sort of best effort thing you are doing, you log the error and ignore it
3. if you are servicing a user request, you log the error and you return it
4. do not stack (`Errorf("I am stacking errors: %s", original_error)`), but pass the original error
5. A panic should be handled within the same layer by throwing an internal error code/start a recovery process and should not be allowed propagate to other packages

Framework directives

1. K.I.S.S.
2. HyperLedgerError interface
    a. Error (from Error interface)
    b. GetCallStack (to get the call stack)
    c. GetErrorCode (**DO WE NEED THIS?** {HELP NEEDED})
3. Some example code

Open questions

How to give examples?

1. divide & conquer: subsystem experts do their parts of the code
2. some examples needed:
    a. random 10% of the code? {which? HELP NEEDED}
    b. a specific 'area' (e.g. subfolder) where there are many 'bad error examples' {which? HELP NEEDED}

- GOTRACEBACK=single is the default value for golang traceback information. This environment variable specifies level of detailed output generated when a Go program fails due to a panic or some runtime condition. The possible values for GOTRACEBACK are All,crash,system. It would be good to decide on what level of detail we might need to show for a traceback in Hyperledger by default. Along these lines do we need to have a configurable environment variable for the hyperledger specific stack trace which signifies the level of detail of the stack trace information?