21-日志处理（下）：手把手教你从0编写一个日志包

你好，我是孔令飞。

上一讲我介绍了如何设计日志包，今天是实战环节，我会手把手教你从0编写一个日志包。

在实际开发中，我们可以选择一些优秀的开源日志包，不加修改直接拿来使用。但更多时候，是基于一个或某几个优秀的开源日志包进行二次开发。想要开发或者二次开发一个日志包，就要掌握日志包的实现方式。那么这一讲中，我来带你从0到1，实现一个具备基本功能的日志包，让你从中一窥日志包的实现原理和实现方法。

在开始实战之前，我们先来看下目前业界有哪些优秀的开源日志包。

有哪些优秀的开源日志包？

在Go项目开发中，我们可以通过修改一些优秀的开源日志包，来实现项目的日志包。Go生态中有很多优秀的开源日志包，例如标准库log包、glog、logrus、zap、seelog、zerolog、log15、apex/log、go-logging等。其中，用得比较多的是标准库log包、glog、logrus和zap。

为了使你了解开源日志包的现状，接下来我会简单介绍下这几个常用的日志包。至于它们的具体使用方法，你可以参考我整理的一篇文章：优秀开源日志包使用教程。

标准库log包

标准库log包的功能非常简单，只提供了Print、Panic和Fatal三类函数用于日志输出。因为是标准库自带的，所以不需要我们下载安装，使用起来非常方便。

标准库log包只有不到400行的代码量，如果你想研究如何实现一个日志包，阅读标准库log包是一个不错的开始。Go的标准库大量使用了log包，例如`net/http`、`net/rpc`等。

glog

glog是Google推出的日志包，跟标准库log包一样，它是一个轻量级的日志包，使用起来简单方便。但glog比标准库log包提供了更多的功能，它具有如下特性：

- 支持4种日志级别：Info、Warning、Error、Fatal。
- 支持命令行选项，例如`-alsologtostderr`、`-log_backtrace_at`、`-log_dir`、`-logtostderr`、`-v`等，每个参数实现某种功能。
- 支持根据文件大小切割日志文件。
- 支持日志按级别分类输出。
- 支持V level。V level特性可以使开发者自定义日志级别。
- 支持vmodule。vmodule可以使开发者对不同的文件使用不同的日志级别。
- 支持traceLocation。traceLocation可以打印出指定位置的栈信息。

Kubernetes项目就使用了基于glog封装的klog，作为其日志库。

logrus

logrus是目前GitHub上star数量最多的日志包，它的优点是功能强大、性能高效、高度灵活，还提供了自定义插件的功能。很多优秀的开源项目，例如Docker、Prometheus等，都使用了logrus。除了具有日志的基本功能外，logrus还具有如下特性：

- 支持常用的日志级别。logrus支持Debug、Info、Warn、Error、Fatal和Panic这些日志级别。
- 可扩展。logrus的Hook机制允许使用者通过Hook的方式，将日志分发到任意地方，例如本地文件、标准输出、Elasticsearch、Logstash、Kafka等。
- 支持自定义日志格式。logrus内置了JSONFormatter和TextFormatter两种格式。除此之外，logrus还允许使用者通过实现Formatter接口，来自定义日志格式。
- 结构化日志记录。logrus的Field机制允许使用者自定义日志字段，而不是通过冗长的消息来记录日志。
- 预设日志字段。logrus的Default Fields机制，可以给一部分或者全部日志统一添加共同的日志字段，例如给某次HTTP请求的所有日志添加X-Request-ID字段。
- Fatal handlers。logrus允许注册一个或多个handler，当产生Fatal级别的日志时调用。当我们的程序需要优雅关闭时，这个特性会非常有用。

zap

zap是uber开源的日志包，以高性能著称，很多公司的日志包都是基于zap改造而来。除了具有日志基本的功能之外，zap还具有很多强大的特性：

- 支持常用的日志级别，例如：Debug、Info、Warn、Error、DPanic、Panic、Fatal。
- 性能非常高。zap具有非常高的性能，适合对性能要求比较高的场景。
- 支持针对特定的日志级别，输出调用堆栈。
- 像logrus一样，zap也支持结构化的目录日志、预设日志字段，也因为支持Hook而具有可扩展性。

开源日志包选择

上面我介绍了很多日志包，每种日志包使用的场景不同，你可以根据自己的需求，结合日志包的特性进行选择：

- **标准库log包：** 标准库log包不支持日志级别、日志分割、日志格式等功能，所以在大型项目中很少直接使用，通常用于一些短小的程序，比如用于生成JWT Token的main.go文件中。标准库日志包也很适合一些简短的代码，用于快速调试和验证。
- **glog：** glog实现了日志包的基本功能，非常适合一些对日志功能要求不多的小型项目。
- **logrus：** logrus功能强大，不仅实现了日志包的基本功能，还有很多高级特性，适合一些大型项目，尤其是需要结构化日志记录的项目。
- **zap：** zap提供了很强大的日志功能，性能高，内存分配次数少，适合对日志性能要求很高的项目。另外，zap包中的子包zapcore，提供了很多底层的日志接口，适合用来做二次封装。

举个我自己选择日志包来进行二次开发的例子：我在做容器云平台开发时，发现Kubernetes源码中大量使用了glog，这时就需要日志包能够兼容glog。于是，我基于zap和zapcore封装了github.com/marmotedu/iam/pkg/log日志包，这个日志包可以很好地兼容glog。

在实际项目开发中，你可以根据项目需要，从上面几个日志包中进行选择，直接使用，但更多时候，你还需要基于这些包来进行定制开发。为了使你更深入地掌握日志包的设计和开发，接下来，我会从0到1带你开发一个日志包。

从0编写一个日志包

接下来，我会向你展示如何快速编写一个具备基本功能的日志包，让你通过这个简短的日志包实现掌握日志包的核心设计思路。该日志包主要实现以下几个功能：

- 支持自定义配置。
- 支持文件名和行号。
- 支持日志级别 Debug、Info、Warn、Error、Panic、Fatal。
- 支持输出到本地文件和标准输出。
- 支持JSON和TEXT格式的日志输出，支持自定义日志格式。
- 支持选项模式。

日志包名称为`cuslog`，示例项目完整代码存放在 cuslog。

具体实现分为以下四个步骤：

1. 定义：定义日志级别和日志选项。
2. 创建：创建Logger及各级别日志打印方法。
3. 写入：将日志输出到支持的输出中。
4. 自定义：自定义日志输出格式。

定义日志级别和日志选项

一个基本的日志包，首先需要定义好日志级别和日志选项。本示例将定义代码保存在options.go文件中。

可以通过如下方式定义日志级别：

```
type Level uint8

const (
    DebugLevel Level = iota
    InfoLevel
    WarnLevel
    ErrorLevel
    PanicLevel
    FatalLevel
)

var LevelNameMapping = map[Level]string{
    DebugLevel: "DEBUG",
    InfoLevel:  "INFO",
    WarnLevel:  "WARN",
    ErrorLevel: "ERROR",
    PanicLevel: "PANIC",
    FatalLevel: "FATAL",
}
```

在日志输出时，要通过对比开关级别和输出级别的大小，来决定是否输出，所以日志级别Level要定义成方便比较的数值类型。几乎所有的日志包都是用常量计数器iota来定义日志级别。

另外，因为要在日志输出中，输出可读的日志级别（例如输出INFO而不是1），所以需要有Level到Level Name的映射LevelNameMapping，LevelNameMapping会在格式化时用到。

接下来看定义日志选项。日志需要是可配置的，方便开发者根据不同的环境设置不同的日志行为，比较常见的配置选项为：

- 日志级别。
- 输出位置，例如标准输出或者文件。
- 输出格式，例如JSON或者Text。
- 是否开启文件名和行号。

本示例的日志选项定义如下：

```
type options struct {
    output        io.Writer
    level         Level
    stdLevel      Level
    formatter     Formatter
    disableCaller bool
}
```

为了灵活地设置日志的选项，你可以通过选项模式，来对日志选项进行设置：

```
type Option func(*options)

func initOptions(opts ...Option) (o *options) {
    o = &options{}
    for _, opt := range opts {
        opt(o)
    }

    if o.output == nil {
        o.output = os.Stderr
    }

    if o.formatter == nil {
        o.formatter = &TextFormatter{}
    }

    return
}

func WithLevel(level Level) Option {
    return func(o *options) {
```

```
        o.level = level
    }
}
...
func SetOptions(opts ...Option) {
    std.SetOptions(opts...)
}

func (l *logger) SetOptions(opts ...Option) {
    l.mu.Lock()
    defer l.mu.Unlock()

    for _, opt := range opts {
        opt(l.opt)
    }
}
```

具有选项模式的日志包，可通过以下方式，来动态地修改日志的选项：

```
cuslog.SetOptions(cuslog.WithLevel(cuslog.DebugLevel))
```

你可以根据需要，对每一个日志选项创建设置函数 `WithXXXX` 。这个示例日志包支持如下选项设置函数：

- WithOutput（output io.Writer）：设置输出位置。
- WithLevel（level Level）：设置输出级别。
- WithFormatter（formatter Formatter）：设置输出格式。
- WithDisableCaller（caller bool）：设置是否打印文件名和行号。

创建Logger及各级别日志打印方法

为了打印日志，我们需要根据日志配置，创建一个Logger，然后通过调用Logger的日志打印方法，完成各级别日志的输出。本示例将创建代码保存在logger.go文件中。

可以通过如下方式创建Logger：

```
var std = New()

type logger struct {
    opt       *options
    mu        sync.Mutex
    entryPool *sync.Pool
}

func New(opts ...Option) *logger {
    logger := &logger{opt: initOptions(opts...)}
    logger.entryPool = &sync.Pool{New: func() interface{} { return entry(logger) }}
    return logger
}
```

上述代码中，定义了一个Logger，并实现了创建Logger的New函数。日志包都会有一个默认的全局Logger，本示例通过 `var std = New()` 创建了一个全局的默认Logger。cuslog.Debug、cuslog.Info和cuslog.Warnf等函数，则是通过调用std Logger所提供的方法来打印日志的。

定义了一个Logger之后，还需要给该Logger添加最核心的日志打印方法，要提供所有支持级别的日志打印方法。

如果日志级别是Xyz，则通常需要提供两类方法，分别是非格式化方法`Xyz(args ...interface{})`和格式化方法`Xyzf(format string, args ...interface{})`，例如：

```
func (l *logger) Debug(args ...interface{}) {
    l.entry().write(DebugLevel, FmtEmptySeparate, args...)
}
func (l *logger) Debugf(format string, args ...interface{}) {
    l.entry().write(DebugLevel, format, args...)
}
```

本示例实现了如下方法：Debug、Debugf、Info、Infof、Warn、Warnf、Error、Errorf、Panic、Panicf、Fatal、Fatalf。更详细的实现，你可以参考 cuslog/logger.go。

这里要注意，Panic、Panicf要调用panic()函数，Fatal、Fatalf函数要调用 `os.Exit(1)` 函数。

将日志输出到支持的输出中

调用日志打印函数之后，还需要将这些日志输出到支持的输出中，所以需要实现write函数，它的写入逻辑保存在entry.go文件中。实现方式如下：

```
type Entry struct {
    logger *logger
    Buffer *bytes.Buffer
    Map    map[string]interface{}
    Level  Level
    Time   time.Time
    File   string
    Line   int
    Func   string
    Format string
    Args   []interface{}
}

func (e *Entry) write(level Level, format string, args ...interface{}) {
    if e.logger.opt.level > level {
        return
    }
    e.Time = time.Now()
    e.Level = level
    e.Format = format
    e.Args = args
    if !e.logger.opt.disableCaller {
        if pc, file, line, ok := runtime.Caller(2); !ok {
```

```
            e.File = "???"
            e.Func = "???"
        } else {
            e.File, e.Line, e.Func = file, line, runtime.FuncForPC(pc).Name()
            e.Func = e.Func[strings.LastIndex(e.Func, "/")+1:]
        }
    }
    e.format()
    e.writer()
    e.release()
}

func (e *Entry) format() {
    _ = e.logger.opt.formatter.Format(e)
}

func (e *Entry) writer() {
    e.logger.mu.Lock()
    _, _ = e.logger.opt.output.Write(e.Buffer.Bytes())
    e.logger.mu.Unlock()
}

func (e *Entry) release() {
    e.Args, e.Line, e.File, e.Format, e.Func = nil, 0, "", "", ""
    e.Buffer.Reset()
    e.logger.entryPool.Put(e)
}
```

上述代码，首先定义了一个Entry结构体类型，该类型用来保存所有的日志信息，即日志配置和日志内容。写入逻辑都是围绕Entry类型的实例来完成的。

用Entry的write方法来完成日志的写入，在write方法中，会首先判断日志的输出级别和开关级别，如果输出级别小于开关级别，则直接返回，不做任何记录。

在write中，还会判断是否需要记录文件名和行号，如果需要则调用 `runtime.Caller()` 来获取文件名和行号，调用 `runtime.Caller()` 时，要注意传入正确的栈深度。

write函数中调用 `e.format` 来格式化日志，调用 `e.writer` 来写入日志，在创建Logger传入的日志配置中，指定了输出位置 `output io.Writer` ，output类型为 `io.Writer` ，示例如下：

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

io.Writer实现了Write方法可供写入，所以只需要调用`e.logger.opt.output.Write(e.Buffer.Bytes())`即可将日志写入到指定的位置中。最后，会调用release()方法来清空缓存和对象池。至此，我们就完成了日志的记录和写入。

自定义日志输出格式

cuslog包支持自定义输出格式，并且内置了JSON和Text格式的Formatter。Formatter接口定义为：

```
type Formatter interface {
    Format(entry *Entry) error
}
```

cuslog内置的Formatter有两个：JSON和TEXT。

测试日志包

cuslog日志包开发完成之后，可以编写测试代码，调用cuslog包来测试cuslog包，代码如下：

```
package main

import (
    "log"
    "os"

    "github.com/marmotedu/gopractise-demo/log/cuslog"
)

func main() {
    cuslog.Info("std log")
    cuslog.SetOptions(cuslog.WithLevel(cuslog.DebugLevel))
    cuslog.Debug("change std log to debug level")
    cuslog.SetOptions(cuslog.WithFormatter(&cuslog.JsonFormatter{IgnoreBasicFields: false}))
    cuslog.Debug("log in json format")
    cuslog.Info("another log in json format")

    // 输出到文件
    fd, err := os.OpenFile("test.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        log.Fatalln("create file test.log failed")
    }
    defer fd.Close()

    l := cuslog.New(cuslog.WithLevel(cuslog.InfoLevel),
        cuslog.WithOutput(fd),
        cuslog.WithFormatter(&cuslog.JsonFormatter{IgnoreBasicFields: false}),
    )
    l.Info("custom log with json formatter")
}
```

将上述代码保存在main.go文件中，运行：

```
$ go run example.go
2020-12-04T10:32:12+08:00 INFO example.go:11 std log
2020-12-04T10:32:12+08:00 DEBUG example.go:13 change std log to debug level
{"file":"/home/colin/workspace/golang/src/github.com/marmotedu/gopractise-demo/log/cuslog/example/example.g
{"level":"INFO","time":"2020-12-04T10:32:12+08:00","file":"/home/colin/workspace/golang/src/github.com/marm
```

到这里日志包就开发完成了，完整包见 log/cuslog。

IAM项目日志包设计

这一讲的最后，我们再来看下我们的IAM项目中，日志包是怎么设计的。

先来看一下IAM项目log包的存放位置：pkg/log。放在这个位置，主要有两个原因：第一个，log包属于IAM项目，有定制开发的内容；第二个，log包功能完备、成熟，外部项目也可以使用。

该log包是基于 go.uber.org/zap 包封装而来的，根据需要添加了更丰富的功能。接下来，我们通过log包的Options，来看下log包所实现的功能：

```
type Options struct {
    OutputPaths       []string `json:"output-paths"       mapstructure:"output-paths"`
    ErrorOutputPaths  []string `json:"error-output-paths" mapstructure:"error-output-paths"`
    Level             string   `json:"level"              mapstructure:"level"`
    Format            string   `json:"format"             mapstructure:"format"`
    DisableCaller     bool     `json:"disable-caller"     mapstructure:"disable-caller"`
    DisableStacktrace bool     `json:"disable-stacktrace" mapstructure:"disable-stacktrace"`
    EnableColor       bool     `json:"enable-color"       mapstructure:"enable-color"`
    Development       bool     `json:"development"        mapstructure:"development"`
    Name              string   `json:"name"               mapstructure:"name"`
}
```

Options各配置项含义如下：

- development：是否是开发模式。如果是开发模式，会对DPanicLevel进行堆栈跟踪。
- name：Logger的名字。
- disable-caller：是否开启 caller，如果开启会在日志中显示调用日志所在的文件、函数和行号。
- disable-stacktrace：是否在Panic及以上级别禁止打印堆栈信息。
- enable-color：是否开启颜色输出，true，是；false，否。
- level：日志级别，优先级从低到高依次为：Debug, Info, Warn, Error, Dpanic, Panic, Fatal。
- format：支持的日志输出格式，目前支持Console和JSON两种。Console其实就是Text格式。
- output-paths：支持输出到多个输出，用逗号分开。支持输出到标准输出（stdout）和文件。
- error-output-paths：zap内部(非业务)错误日志输出路径，多个输出，用逗号分开。

log包的Options结构体支持以下3个方法：

- Build方法。Build方法可以根据Options构建一个全局的Logger。
- AddFlags方法。AddFlags方法可以将Options的各个字段追加到传入的pflag.FlagSet变量中。
- String方法。String方法可以将Options的值以JSON格式字符串返回。

log包实现了以下3种日志记录方法：

```
log.Info("This is a info message", log.Int32("int_key", 10))
log.Infof("This is a formatted %s message", "info")
log.Infow("Message printed with Infow", "X-Request-ID", "fbf54504-64da-4088-9b86-67824a7fb508")
```

`Info` 使用指定的key/value记录日志。`Infof` 格式化记录日志。 `Infow` 也是使用指定的key/value记录日志，跟 `Info` 的区别是：使用 `Info` 需要指定值的类型，通过指定值的日志类型，日志库底层不需要进行反射操作，所以使用 `Info` 记录日志性能最高。

log包支持非常丰富的类型，具体你可以参考 types.go。

上述日志输出为：

```
2021-07-06 14:02:07.070 INFO This is a info message {"int_key": 10}
2021-07-06 14:02:07.071 INFO This is a formatted info message
2021-07-06 14:02:07.071 INFO Message printed with Infow {"X-Request-ID": "fbf54504-64da-4088-9b86-67824a7fb
```

log包为每种级别的日志都提供了3种日志记录方式，举个例子：假设日志格式为 `Xyz` ，则分别提供了 `Xyz(msg string, fields ...Field)` ，`Xyzf(format string, v ...interface{})` ，`Xyzw(msg string, keysAndValues ...interface{})` 3种日志记录方法。

另外，log包相较于一般的日志包，还提供了众多记录日志的方法。

第一个方法， log包支持V Level，可以通过整型数值来灵活指定日志级别，数值越大，优先级越低。例如：

```
// V level使用
log.V(1).Info("This is a V level message")
log.V(1).Infof("This is a %s V level message", "formatted")
log.V(1).Infow("This is a V level message with fields", "X-Request-ID", "7a7b9f24-4cae-4b2a-9464-69088b45b9
```

这里要注意，Log.V只支持 `Info` 、`Infof` 、`Infow`三种日志记录方法。

第二个方法， log包支持WithValues函数，例如：

```
// WithValues使用
lv := log.WithValues("X-Request-ID", "7a7b9f24-4cae-4b2a-9464-69088b45b904")
lv.Infow("Info message printed with [WithValues] logger")
lv.Infow("Debug message printed with [WithValues] logger")
```

上述日志输出如下：

```
2021-07-06 14:15:28.555 INFO Info message printed with [WithValues] logger {"X-Request-ID": "7a7b9f24-4cae-
2021-07-06 14:15:28.556 INFO Debug message printed with [WithValues] logger {"X-Request-ID": "7a7b9f24-4cae
```

`WithValues` 可以返回一个携带指定key-value的Logger，供后面使用。

第三个方法， log包提供 `WithContext` 和 `FromContext` 用来将指定的Logger添加到某个Context中，以及从某个Context中获取Logger，例如：

```
// Context使用
ctx := lv.WithContext(context.Background())
lc := log.FromContext(ctx)
lc.Info("Message printed with [WithContext] logger")
```

`WithContext`和`FromContext`非常适合用在以`context.Context`传递的函数中，例如：

```
func main() {

    ...

    // WithValues使用
    lv := log.WithValues("X-Request-ID", "7a7b9f24-4cae-4b2a-9464-69088b45b904")

    // Context使用
    lv.Infof("Start to call pirntString")
    ctx := lv.WithContext(context.Background())
    pirntString(ctx, "World")
}

func pirntString(ctx context.Context, str string) {
    lc := log.FromContext(ctx)
    lc.Infof("Hello %s", str)
}
```

上述代码输出如下：

```
2021-07-06 14:38:02.050 INFO Start to call pirntString {"X-Request-ID": "7a7b9f24-4cae-4b2a-9464-69088b45b9
2021-07-06 14:38:02.050 INFO Hello World {"X-Request-ID": "7a7b9f24-4cae-4b2a-9464-69088b45b904"}
```

将Logger添加到Context中，并通过Context在不同函数间传递，可以使key-value在不同函数间传递。例如上述代码中， `X-Request-ID` 在main函数、printString函数中的日志输出中均有记录，从而实现了一种调用链的效果。

第四个方法， 可以很方便地从Context中提取出指定的key-value，作为上下文添加到日志输出中，例如internal/apiserver/api/v1/user/create.go文件中的日志调用：

```
log.L(c).Info("user create function called.")
```

通过调用 `Log.L()` 函数，实现如下：

```
// L method output with specified context value.
func L(ctx context.Context) *zapLogger {
    return std.L(ctx)
}

func (l *zapLogger) L(ctx context.Context) *zapLogger {
    lg := l.clone()

    requestID, _ := ctx.Value(KeyRequestID).(string)
    username, _ := ctx.Value(KeyUsername).(string)
    lg.zapLogger = lg.zapLogger.With(zap.String(KeyRequestID, requestID), zap.String(KeyUsername, username)

    return lg
}
```

`L()` 方法会从传入的Context中提取出 `requestID` 和 `username` ，追加到Logger中，并返回Logger。这时候调用该Logger的Info、Infof、Infow等方法记录日志，输出的日志中均包含 `requestID` 和 `username` 字段，例如：

```
2021-07-06 14:46:00.743 INFO    apiserver       secret/create.go:23     create secret function called.  {"r
```

通过将Context在函数间传递，很容易就能实现调用链效果，例如：

```
// Create add new secret key pairs to the storage.
func (s *SecretHandler) Create(c *gin.Context) {
    log.L(c).Info("create secret function called.")

    ...

    sec, err := s.store.Secrets().List(c, username, metav1.ListOptions{
    Offset: pointer.ToInt64(0),
    Limit:  pointer.ToInt64(-1),
    })

    ...

    if err := s.srv.Secrets().Create(c, &r, metav1.CreateOptions{}); err != nil {
    core.WriteResponse(c, err, nil)
```

```
    return
}
```

上述代码输出为：

```
2021-07-06 14:46:00.743 INFO    apiserver       secret/create.go:23     create secret function called.  {"r
2021-07-06 14:46:00.744 INFO    apiserver       secret/create.go:23     list secret from storage.  {"reques
2021-07-06 14:46:00.745 INFO    apiserver       secret/create.go:23     insert secret to storage.  {"reques
```

这里要注意， `log.L` 函数默认会从Context中取 `requestID` 和 `username` 键，这跟IAM项目有耦合度，但这不影响log包供第三方项目使用。这也是我建议你自己封装日志包的原因。

总结

开发一个日志包，我们很多时候需要基于一些业界优秀的开源日志包进行二次开发。当前很多项目的日志包都是基于zap日志包来封装的，如果你有封装的需要，我建议你优先选择zap日志包。

这一讲中，我先给你介绍了标准库log包、glog、logrus和zap这四种常用的日志包，然后向你展现了开发一个日志包的四个步骤，步骤如下：

1. 定义日志级别和日志选项。
2. 创建Logger及各级别日志打印方法。
3. 将日志输出到支持的输出中。
4. 自定义日志输出格式。

最后，我介绍了IAM项目封装的log包的设计和使用方式。log包基于 `go.uber.org/zap`封装，并提供了以下强大特性：

- log包支持V Level，可以灵活的通过整型数值来指定日志级别。
- log包支持 `WithValues` 函数， `WithValues` 可以返回一个携带指定key-value对的Logger，供后面使用。
- log包提供 `WithContext` 和 `FromContext` 用来将指定的Logger添加到某个Context中和从某个Context中获取Logger。
- log包提供了 `Log.L()` 函数，可以很方便的从Context中提取出指定的key-value对，作为上下文添加到日志输出中。

课后练习

1. 尝试实现一个新的Formatter，可以使不同日志级别以不同颜色输出（例如：Error级别的日志输出中 `Error` 字符串用红色字体输出， `Info` 字符串用白色字体输出）。
2. 尝试将runtime.Caller(2)函数调用中的 2 改成 1 ，看看日志输出是否跟修改前有差异，如果有差异，思考差异产生的原因。

欢迎你在留言区与我交流讨论，我们下一讲见。

精选留言：

- nio 2021-07-13 17:16:57
  IAM 项目 log 包的性能比较大概是什么样子呢

  作者回复2021-07-15 00:32:40
  性能跟github.com/pkg/log性能接近一致。github.com/pkg/log这个包很多生产环境在用，所以iam的log包应用在生产环境完全没问题。

  你要感兴趣，可以跟其它包对比下，比如：logrus，zap，glog等。也欢迎在留言区分享对比结果。

- helloworld 2021-07-13 01:45:46
  log.Int32("int_key", 10)还有V Level这两处没有get到是干啥用的

  作者回复2021-07-13 10:20:13
  log.Int32直接指定了字段类型，log不需要再做反射，这种疾苦方式可以提高性能。

  V Level可以允许指定任意优先级的日志级别。你可以参考glog的用法来理解V level。

  有时候日志包预定义的日志级别可能不够用，这时候可以试试V Level