golang日志库，你选对了吗?

1 背景

一般在进行业务开发时都需要进行日志打印输出，这个时候如果本身公司内部没有现成的日志库可用，那么就需要进行日志库的技术选型。在选型之前，需要梳理一般对于日志库的需求：

(1)支持日志级别设置

(2)支持hook机制

(3)支持日志格式输出

(4)支持自定义field机制

(5)支持打印行号和调用方法，文件

(6)支持多目的地输出

(7)支持日志切割

(8)支持并发安全

2 业界已有的开源成熟日志库

目前业界比较有名气的日志库主要有logrus，zap，zerolog。在分析这些开源日志库之前，先说下golang本身自带的log库有哪些能力和不足。

2.1 golang自带log包

log包，本身也就300多行代码，相比fmt.Println而已，在日志中添加了输出时间，调用者信息等，支持日志打印和转存功能，并且还支持高并发操作。

```
type Logger struct {
  mu   sync.Mutex // 使用互斥锁，可以支持高并发
  prefix string   // 日志打印前缀
  flag  int    // properties
  out  io.Writer // destination for output
  buf  []byte   // for accumulating text to write
}
```

log包支持简单的自定义日志格式，在创建logger对象的时候传入

```
 const (
  Ldate     = 1 << iota   // the date in the local time zone: 2009/01/23
  Ltime             // the time in the local time zone: 01:23:23
  Lmicroseconds          // microsecond resolution: 01:23:23.123123. assumes Ltime.
  Llongfile           // full file name and line number: /a/b/c/d.go:23
  Lshortfile          // final file name element and line number: d.go:23. overrides
Llongfile
```

```
  LUTC             // if Ldate or Ltime is set, use UTC rather than the local time zone
  Lmsgprefix          // move the "prefix" from the beginning of the line to before the
message
  LstdFlags   = Ldate | Ltime // initial values for the standard logger
)

// 构造函数，创建Logger对象
func New(out io.Writer, prefix string, flag int) *Logger {
  return &Logger{out: out, prefix: prefix, flag: flag}
}
```

log包对外只支持print，printf，println，fatal，fatalf，fatalln，panic，panicf，panicln方法。

```
// Printf calls l.Output to print to the logger.
// Arguments are handled in the manner of fmt.Printf.
func (l *Logger) Printf(format string, v ...interface{}) {
  l.Output(2, fmt.Sprintf(format, v...))
}

// Print calls l.Output to print to the logger.
// Arguments are handled in the manner of fmt.Print.
func (l *Logger) Print(v ...interface{}) { l.Output(2, fmt.Sprint(v...)) }

// Println calls l.Output to print to the logger.
// Arguments are handled in the manner of fmt.Println.
func (l *Logger) Println(v ...interface{}) { l.Output(2, fmt.Sprintln(v...)) }

// Fatal is equivalent to l.Print() followed by a call to os.Exit(1).
func (l *Logger) Fatal(v ...interface{}) {
  l.Output(2, fmt.Sprint(v...))
  os.Exit(1)
}

// Fatalf is equivalent to l.Printf() followed by a call to os.Exit(1).
func (l *Logger) Fatalf(format string, v ...interface{}) {
  l.Output(2, fmt.Sprintf(format, v...))
  os.Exit(1)
}

// Fatalln is equivalent to l.Println() followed by a call to os.Exit(1).
func (l *Logger) Fatalln(v ...interface{}) {
  l.Output(2, fmt.Sprintln(v...))
  os.Exit(1)
}

// Panic is equivalent to l.Print() followed by a call to panic().
func (l *Logger) Panic(v ...interface{}) {
  s := fmt.Sprint(v...)
```

```
  l.Output(2, s)
  panic(s)
}

// Panicf is equivalent to l.Printf() followed by a call to panic().
func (l *Logger) Panicf(format string, v ...interface{}) {
  s := fmt.Sprintf(format, v...)
  l.Output(2, s)
  panic(s)
}

// Panicln is equivalent to l.Println() followed by a call to panic().
func (l *Logger) Panicln(v ...interface{}) {
  s := fmt.Sprintln(v...)
  l.Output(2, s)
  panic(s)
}
```

优缺点分析：

优点：golang本身自带，有默认全局logger提供(一般对于sdk开发的话可以借鉴，考虑是否需要提供全局默认的实例)，支持并发安全打印，相比fmt支持将打印输出到日志文件中。

缺点：功能过于简单，不支持日志级别设置，日志格式自定义，hook机制，自定义field输出，日志分发等功能。

2.2 logrus

针对golang本身log包的不足，golang生态自然就催生了很多第三方的日志库，而logrus就是其中的一款第三方日志库，它的功能强大，性能高效，而且具有高度灵活性，提供了自定义插件的功能。主要具有以下特性：

(1)完全兼容golang标准库日志模块：logrus拥有六种日志级别：debug、info、warn、error、fatal和panic，这是golang标准库日志模块的API的超集。如果项目使用标准库日志模块，完全可以以最低的代价迁移到logrus上。

(2)可扩展的Hook机制：允许使用者通过hook的方式将日志分发到任意地方，如本地文件系统、标准输出、logstash、elasticsearch或者mq等，或者通过hook定义日志内容和格式等。

(3)可选的日志输出格式：logrus内置了两种日志格式，`JSONFormatter`和`TextFormatter`，如果这两个格式不满足需求，可以自己动手实现接口`Formatter`，来定义自己的日志格式。

(4)Field机制：logrus鼓励通过Field机制进行精细化的、结构化的日志记录，而不是通过冗长的消息来记录日志。

(5)logrus是一个可插拔的、结构化的日志框架。

logrus的具体使用参考：https://blog.csdn.net/sinat_38068807/article/details/106941878

logrus优缺点分析：

优点：功能强大，有默认全局实例，需求基本都可以满足，代码设计比较简洁，只有2000多行。

缺点：官方无日志切割方案，需要和第三方库配合，对于多目的地输出，需要开发人员自己实现hook机制，

日志打印性能方面相对欠缺。

logrus本身可以借鉴的关键代码设计：

(1)采用sync.pool提升性能，对象复用，减少gc

```
type Logger struct {
  // 日志输出目的地
  Out io.Writer
  // hook机制，每个日志级别对于一个hook切片
  Hooks LevelHooks
  // 日志输出格式，支持扩展
  Formatter Formatter
  // 是否打印输出日志调用函数和行号信息
  ReportCaller bool
  // 日志级别
  Level Level
  // 并发互斥锁
  mu MutexWrap
  // 对于entry的获取和回收机制采用sync.pool
  entryPool sync.Pool
  // Function to exit the application, defaults to `os.Exit()`
  ExitFunc exitFunc
  // The buffer pool used to format the log. If it is nil, the default global
  // buffer pool will be used.
  BufferPool BufferPool
}
func (logger *Logger) newEntry() *Entry {
  // 创建时优先从pool中获取
  entry, ok := logger.entryPool.Get().(*Entry)
  if ok {
    return entry
  }
  return NewEntry(logger)
}

func (logger *Logger) releaseEntry(entry *Entry) {
  entry.Data = map[string]interface{}{}
  // 使用完后，reset掉数据信息再放回pool中
  logger.entryPool.Put(entry)
}
```

(2)hook机制，提供高扩展性

```
type Hook interface {
  Levels() []Level
  Fire(*Entry) error
}

// Internal type for storing the hooks on a logger instance.
type LevelHooks map[Level][]Hook

// hook函数添加
```

```
func (hooks LevelHooks) Add(hook Hook) {
  for _, level := range hook.Levels() {
    hooks[level] = append(hooks[level], hook)
  }
}

// hook函数执行
func (hooks LevelHooks) Fire(level Level, entry *Entry) error {
  for _, hook := range hooks[level] {
    if err := hook.Fire(entry); err != nil {
      return err
    }
  }

  return nil
}
```

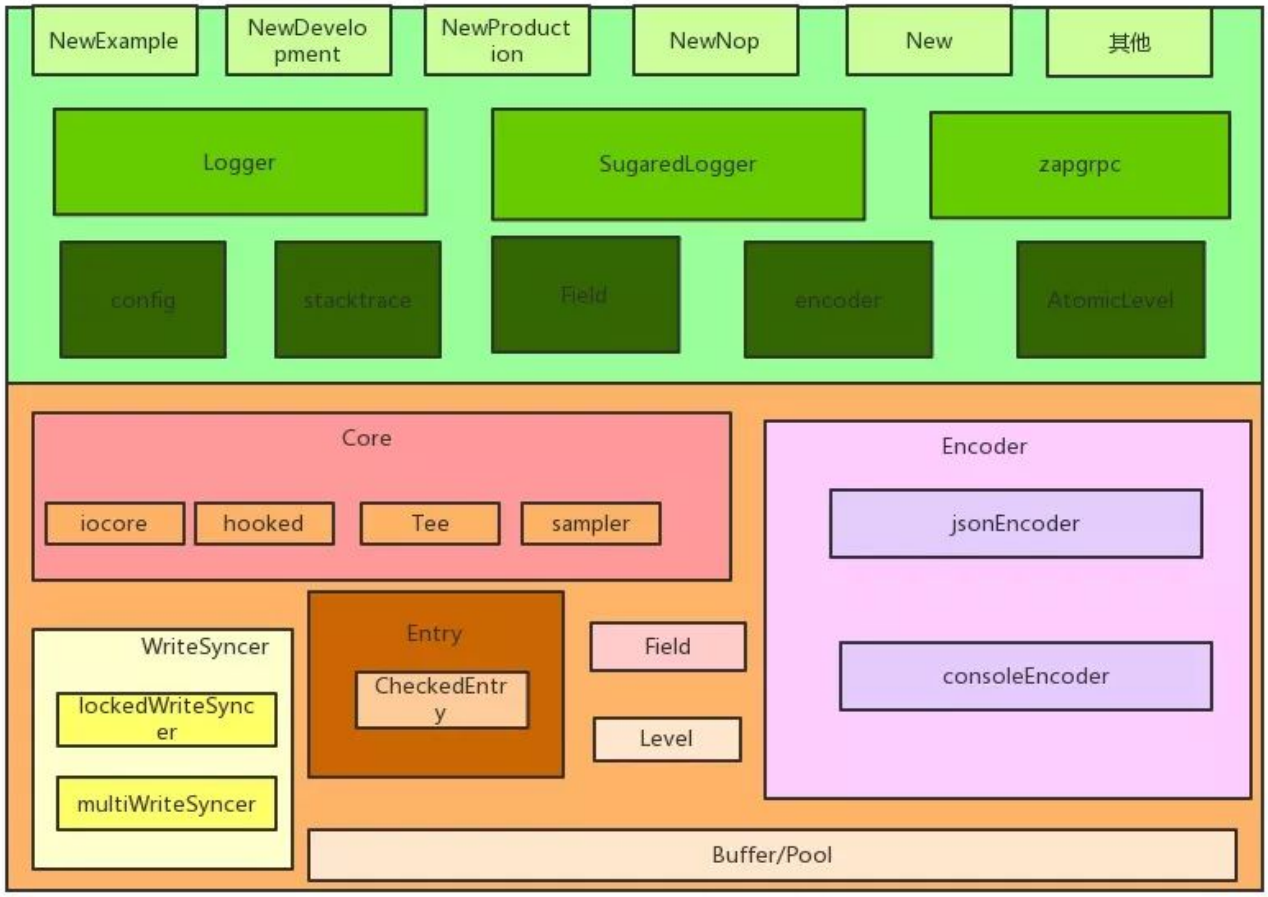
(3)全局实例设计

```
// exporter.go
var (
  // std is the name of the standard logger in stdlib `log`
  std = New()
)

func StandardLogger() *Logger {
  return std
}

// logger.go
func New() *Logger {
  return &Logger{
    Out:          os.Stderr,
    Formatter:    new(TextFormatter),
    Hooks:        make(LevelHooks),
    Level:        InfoLevel,
    ExitFunc:     os.Exit,
    ReportCaller: false,
  }
}
```

2.3 zap

zap 是uber开源的高性能日志库，跟 logrus 类似，提倡采用结构化的日志格式，而不是将所有消息放到消息体中。一般来说，日志有两个概念：字段和消息，其中字段用来结构化输出错误相关的上下文环境，而消息简明扼要的阐述错误本身。

zap设计



Logger：使用较为繁琐，只能使用结构化输出，但是性能更好；

SugaredLogger：可以使用 Printf 来输出日志，性能较 Logger 相比差 40% 左右；

zapgrpc：用做 grpc 的日志输出；

在设计上 Logger 可以很方便的转化为 SugaredLogger 和 zapgrpc。这几个 Logger 需要传入一个 Core 接口的实现类才能创建。

Core 接口：zap 也提供了多种实现的选择：NewNopCore 、ioCore、multiCore 、hook。

最常用的是 ioCore、multiCore ，从名字便可看出来 multiCore 是可以包含多个 ioCore 的一种配置，比方说可以让 Error 日志输出一种日志格式以及设置一个日志输出目的地，让 Info 日志以另一种日志格式输出到别的地方。

在上面也说了，对于 Core 的实现类 ioCore 来说它需要传入三个对象：输入数据的编码器 Encoder、日志数据的目的地 WriteSyncer，以及日志级别 LevelEnabler。

Encoder 接口：zap 提供了 consoleEncoder、jsonEncoder 的实现，分别提供了 console 格式与 JSON 格式日志输出，这些 Encoder 都有自己的序列化实现，这样可以更快的格式化代码；

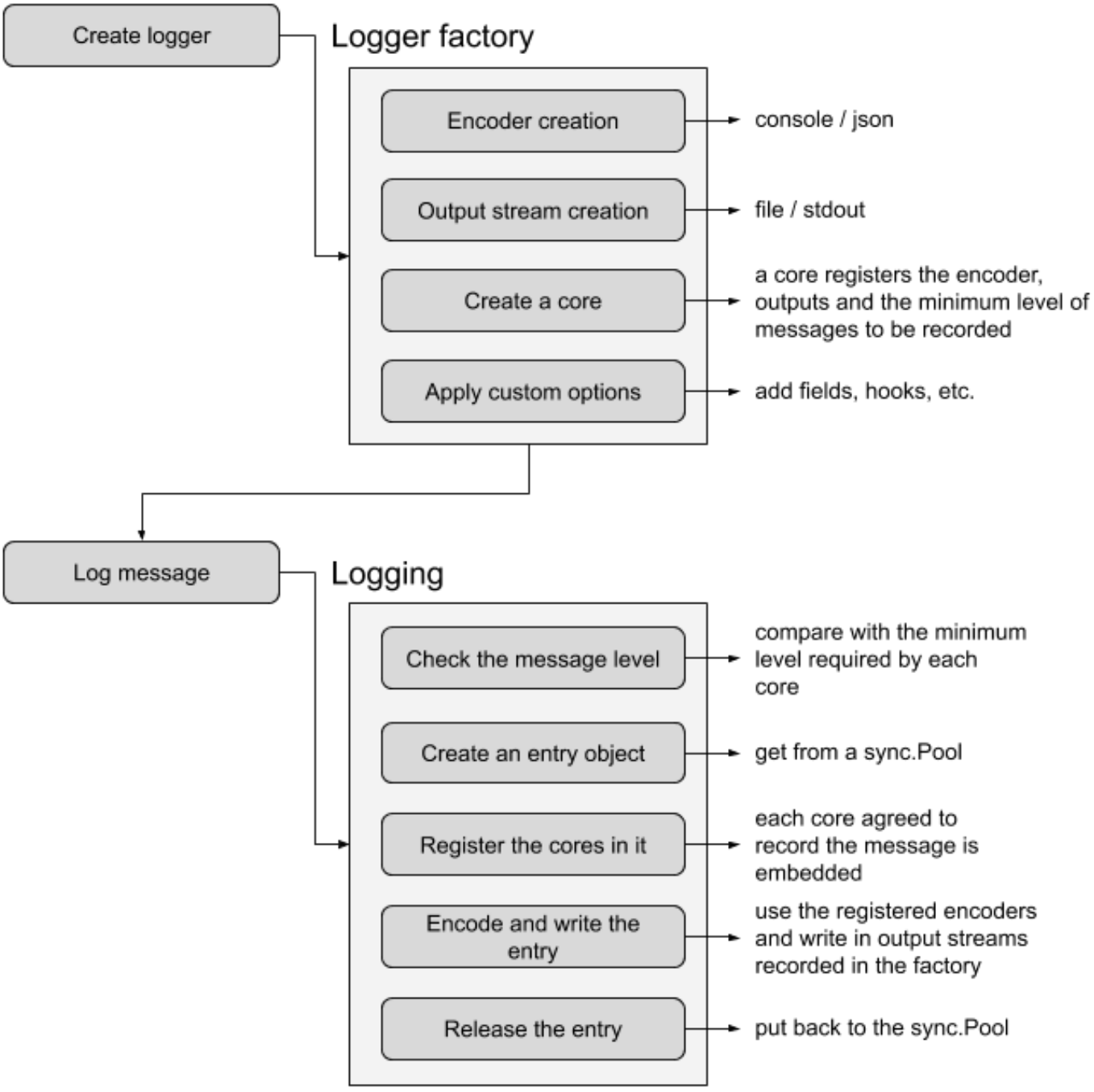
EncoderConfig：上面所说的 Encoder 还可以根据 EncoderConfig 的配置允许使用者灵活的配置日志的输出格式，从日志消息的键名、日志等级名称，到时间格式输出的定义，方法名的定义都可以通过它灵活配置。

WriteSyncer 接口：zap 提供了 writerWrapper 的单日志输出实现，以及可以将日志输出到多个地方的 multiWriteSyncer 实现；

Entry：配置说完了，到了日志数据的封装。首先日志数据会封装成一个 Entry，包含了日志名、日志时间、日志等级，以及日志数据等信息，没有 Field 信息，然后经验 Core 的 Check 方法对日志等级校验通过之后会生成一个 CheckedEntry 实例。

CheckedEntry 包含了日志数据所有信息，包括上面提到的 Entry、调用栈信息等。

zap的简单实用：https://studygolang.com/articles/17394



zap优缺点分析

优点：功能强大，性能比较强，需求基本都可以满足。

缺点：官方无日志切割方案，需要和第三方库配合。

zap高性能秘诀

(1)使用sync.pool提升性能

zap 通过 sync.Pool 提供的对象池，复用了大量可以复用的对象，zap 在实例化 CheckedEntry 、Buffer、Encoder 等对象的时候，会直接从对象池中获取，而不是直接实例化一个新的，这样复用对象可以降低 GC 的压力，减少内存分配。

```
// entry.go
var (
  _cePool = sync.Pool{New: func() interface{} {
    // Pre-allocate some space for cores.
    return &CheckedEntry{
      cores: make([]Core, 4),
    }
  }}
)

func getCheckedEntry() *CheckedEntry {
  ce := _cePool.Get().(*CheckedEntry)
  ce.reset()
  return ce
}

func putCheckedEntry(ce *CheckedEntry) {
  if ce == nil {
    return
  }
  _cePool.Put(ce)
}
func (ce *CheckedEntry) Write(fields ...Field) {
  if ce == nil {
    return
  }

  if ce.dirty {
    if ce.ErrorOutput != nil {
      // Make a best effort to detect unsafe re-use of this CheckedEntry.
      // If the entry is dirty, log an internal error; because the
      // CheckedEntry is being used after it was returned to the pool,
      // the message may be an amalgamation from multiple call sites.
      fmt.Fprintf(ce.ErrorOutput, "%v Unsafe CheckedEntry re-use near Entry %+v.\n",
ce.Time, ce.Entry)
      ce.ErrorOutput.Sync()
    }
    return
  }
  ce.dirty = true
```

```
  var err error
  for i := range ce.cores {
    err = multierr.Append(err, ce.cores[i].Write(ce.Entry, fields))
  }
  if ce.ErrorOutput != nil {
    if err != nil {
      fmt.Fprintf(ce.ErrorOutput, "%v write error: %v\n", ce.Time, err)
      ce.ErrorOutput.Sync()
    }
  }

  should, msg := ce.should, ce.Message
  putCheckedEntry(ce)

  switch should {
  case WriteThenPanic:
    panic(msg)
  case WriteThenFatal:
    exit.Exit()
  case WriteThenGoexit:
    runtime.Goexit()
  }
}

// AddCore adds a Core that has agreed to log this CheckedEntry. It's intended to be
// used by Core.Check implementations, and is safe to call on nil CheckedEntry
// references.
func (ce *CheckedEntry) AddCore(ent Entry, core Core) *CheckedEntry {
  if ce == nil {
    // 从pool中获取
    ce = getCheckedEntry()
    ce.Entry = ent
  }
  ce.cores = append(ce.cores, core)
  return ce
}

// Should sets this CheckedEntry's CheckWriteAction, which controls whether a
// Core will panic or fatal after writing this log entry. Like AddCore, it's
// safe to call on nil CheckedEntry references.
func (ce *CheckedEntry) Should(ent Entry, should CheckWriteAction) *CheckedEntry {
  if ce == nil {
    ce = getCheckedEntry()
    ce.Entry = ent
  }
  ce.should = should
  return ce
}
```

(2)避免反射，避免使用 `interface{}` ，引入强类型参数，防止装箱拆箱的性能损耗

`fmt.Sprintf`效率实际上是很低的，通过查看fmt.Sprintf源码， 可以看出效率低有两个原因：

1. `fmt.Sprintf` 接受的类型是 interface{}，内部使用了反射；
2. `fmt.Sprintf` 的用途是格式化字符串，需要去解析格式串，比如 `%s`、 `%d`之类的，增加了解析的耗时。

但是在 zap 中，使用的是内建的 Encoder，它会通过内部的 Buffer 以 byte 的形式来拼接日志数据，减少反射所带来性能损失；以及 zap 是使用的结构化的日志，所以没有 `%s`、 `%d`之类的标识符需要解析，也是一个性能提升点。

```
func (c *ioCore) Write(ent Entry, fields []Field) error {
  buf, err := c.enc.EncodeEntry(ent, fields)
  if err != nil {
    return err
  }
  _, err = c.out.Write(buf.Bytes())
  buf.Free()
  if err != nil {
    return err
  }
  if ent.Level > ErrorLevel {
    // Since we may be crashing the program, sync the output. Ignore Sync
    // errors, pending a clean solution to issue #370.
    c.Sync()
  }
  return nil
}

// field.go
func addFields(enc ObjectEncoder, fields []Field) {
  for i := range fields {
    fields[i].AddTo(enc)
  }
}
type Field struct {
  Key       string
  Type      FieldType
  Integer   int64
  String    string
  Interface interface{}
}
// 避免使用反射
func (f Field) AddTo(enc ObjectEncoder) {
  var err error

  switch f.Type {
  case ArrayMarshalerType:
    err = enc.AddArray(f.Key, f.Interface.(ArrayMarshaler))
  case ObjectMarshalerType:
```

```
    err = enc.AddObject(f.Key, f.Interface.(ObjectMarshaler))
  case InlineMarshalerType:
    err = f.Interface.(ObjectMarshaler).MarshalLogObject(enc)
  case BinaryType:
    enc.AddBinary(f.Key, f.Interface.([]byte))
  case BoolType:
    enc.AddBool(f.Key, f.Integer == 1)
  case ByteStringType:
    enc.AddByteString(f.Key, f.Interface.([]byte))
  case Complex128Type:
    enc.AddComplex128(f.Key, f.Interface.(complex128))
  case Complex64Type:
    enc.AddComplex64(f.Key, f.Interface.(complex64))
  case DurationType:
    enc.AddDuration(f.Key, time.Duration(f.Integer))
  case Float64Type:
    enc.AddFloat64(f.Key, math.Float64frombits(uint64(f.Integer)))
  case Float32Type:
    enc.AddFloat32(f.Key, math.Float32frombits(uint32(f.Integer)))
  case Int64Type:
    enc.AddInt64(f.Key, f.Integer)
  case Int32Type:
    enc.AddInt32(f.Key, int32(f.Integer))
  case Int16Type:
    enc.AddInt16(f.Key, int16(f.Integer))
  case Int8Type:
    enc.AddInt8(f.Key, int8(f.Integer))
  case StringType:
    enc.AddString(f.Key, f.String)
  case TimeType:
    if f.Interface != nil {
      enc.AddTime(f.Key, time.Unix(0, f.Integer).In(f.Interface.(*time.Location)))
    } else {
      // Fall back to UTC if location is nil.
      enc.AddTime(f.Key, time.Unix(0, f.Integer))
    }
  case TimeFullType:
    enc.AddTime(f.Key, f.Interface.(time.Time))
  case Uint64Type:
    enc.AddUint64(f.Key, uint64(f.Integer))
  case Uint32Type:
    enc.AddUint32(f.Key, uint32(f.Integer))
  case Uint16Type:
    enc.AddUint16(f.Key, uint16(f.Integer))
  case Uint8Type:
    enc.AddUint8(f.Key, uint8(f.Integer))
  case UintptrType:
    enc.AddUintptr(f.Key, uintptr(f.Integer))
  case ReflectType:
```

```
    err = enc.AddReflected(f.Key, f.Interface)
  case NamespaceType:
    enc.OpenNamespace(f.Key)
  case StringerType:
    err = encodeStringer(f.Key, f.Interface, enc)
  case ErrorType:
    err = encodeError(f.Key, f.Interface.(error), enc)
  case SkipType:
    break
  default:
    panic(fmt.Sprintf("unknown field type: %v", f))
  }

  if err != nil {
    enc.AddString(fmt.Sprintf("%sError", f.Key), err.Error())
  }
}
```

但是在使用sugar logger时，依然使用的fmt.sprintf

```
func (s *SugaredLogger) log(lvl zapcore.Level, template string, fmtArgs []interface{},
context []interface{}) {
  // If logging at this level is completely disabled, skip the overhead of
  // string formatting.
  if lvl < DPanicLevel && !s.base.Core().Enabled(lvl) {
    return
  }
  // 其底层使用的也是反射能力
  msg := getMessage(template, fmtArgs)
  if ce := s.base.Check(lvl, msg); ce != nil {
    ce.Write(s.sweetenFields(context)...)
  }
}

// getMessage format with Sprint, Sprintf, or neither.
func getMessage(template string, fmtArgs []interface{}) string {
  if len(fmtArgs) == 0 {
    return template
  }

  if template != "" {
    return fmt.Sprintf(template, fmtArgs...)
  }

  if len(fmtArgs) == 1 {
    if str, ok := fmtArgs[0].(string); ok {
      return str
    }
  }
```

```
  return fmt.Sprint(fmtArgs...)
}
```

2.4 zerolog

zerolog 包提供了一个专门用于 JSON 输出的简单快速的 Logger。zerolog 的 API 旨在为开发者提供出色的体验和令人惊叹的性能。其独特的链式 API 允许通过避免内存分配和反射来写入 JSON日志。

在整体设计上，zerolog和zap差不多保持一致，主要区别将entry改成event，在使用使用采用链式调用方式。

zerolog使用案例：https://studygolang.com/articles/25383

优缺点：

优点：性能优越，提供全局实例，采用链式调用，对于开发者的体验比较友好。

缺点：社区活跃度，生态上面相比logrus，zap会差点，同时也需要配合使用第三库进行日志切割。

3 对比分析

3.1 功能对比分析

	logrus	**zap**	**zerolog**
支持日志级别设置	支持	支持	支持
支持hook机制	支持	支持	支持
支持日志格式输出	支持	支持	支持
支持自定义field机制	支持	支持	支持
支持打印行号和调用方法，文件	支持	支持	支持
支持多目的地输出	支持	支持	支持
支持日志切割	支持，需第三方配合	支持，需第三方配合	支持，需第三方配合
支持并发安全	支持	支持	支持

3.2 性能对比分析

Log a message and 10 fields:

Package	Time	Time % to zap	Objects Allocated
⚡ zap	862 ns/op	+0%	5 allocs/op
⚡ zap (sugared)	1250 ns/op	+45%	11 allocs/op
zerolog	4021 ns/op	+366%	76 allocs/op
go-kit	4542 ns/op	+427%	105 allocs/op
apex/log	26785 ns/op	+3007%	115 allocs/op
logrus	29501 ns/op	+3322%	125 allocs/op
log15	29906 ns/op	+3369%	122 allocs/op

Log a message with a logger that already has 10 fields of context:

Package	Time	Time % to zap	Objects Allocated
⚡ zap	126 ns/op	+0%	0 allocs/op
⚡ zap (sugared)	187 ns/op	+48%	2 allocs/op
zerolog	88 ns/op	-30%	0 allocs/op
go-kit	5087 ns/op	+3937%	103 allocs/op
log15	18548 ns/op	+14621%	73 allocs/op
apex/log	26012 ns/op	+20544%	104 allocs/op
logrus	27236 ns/op	+21516%	113 allocs/op

Log a static string, without any context or `printf`-style templating:

Package	Time	Time % to zap	Objects Allocated
⚡ zap	118 ns/op	+0%	0 allocs/op
⚡ zap (sugared)	191 ns/op	+62%	2 allocs/op
zerolog	93 ns/op	-21%	0 allocs/op
go-kit	280 ns/op	+137%	11 allocs/op
standard library	499 ns/op	+323%	2 allocs/op
apex/log	1990 ns/op	+1586%	10 allocs/op
logrus	3129 ns/op	+2552%	24 allocs/op
log15	3887 ns/op	+3194%	23 allocs/op

3.3 可靠性对比分析

暂无

3.4 业务连续性对比分析

	logrus	zap	zerolog
github stars	18.2k	13.1k	4.9k
contributors	250	88	83
最近一次提交时间	2021-4-20	2021-7-6	2021-7-15
最近一次版本发布时间	2021-3-9	2021-6-29	2017-6-23
资料文档	详细	详细	一般
社区活跃度	活跃	活跃	一般
fork	2k	988	297
License	MIT	MIT	MIT

4 结论

综合从功能需求，性能维度，业务连续性维度分析，选择使用zap作为日志库上比较好的。

5 参考

[1] https://www.cnblogs.com/luozhiyun/p/14885034.html

[2] https://github.com/rs/zerolog

[3] https://github.com/uber-go/zap

[4] https://github.com/sirupsen/logrus