附录 A

使应用程序可观察

在本附录中，我将提供一些关于实现技术的指南，以使应用程序行为可通过遥测数据进行观察。这些数据通常分为日志(log)、指标(metric)和跟踪(trace)。所有这三种数据都可以帮助我们了解应用程序正在做什么，并帮助我们了解其在给定时间点的内部状态。首先，我们将介绍从应用程序发出的遥测数据的类别，讨论用于实现它们的 Go 包。此后将列举将它们集成到应用程序的示例。

日志、指标和跟踪

到目前为止，我们在本书中使用了标准库日志包中定义的函数，例如 Printf()和 Fatal()来记录来自应用程序的消息。这种日志记录技术很容易实现，并且比根本没有任何日志记录要好。可以使用日志中的文本在日志系统中搜索日志，但当想要在日志中搜索特定数据时，这变得不那么有用了。一个非常常见的操作是搜索具有与请求相关的特定上下文或元数据的日志，例如，特定命令、特定 HTTP 路径或 gRPC 方法的所有日志。如果使用我们迄今为止学习的日志记录技术，执行此类搜索既昂贵又低效。因此，我们必须找到一种从应用程序发出日志的方法，以使每个日志行符合特定的结构——单独标记的字段包含日志数据以及作为元数据的上下文信息。有多种方法可以实现这种日志记录机制，它们通常都称为结构化日志记录。日志行不是自由格式的文本，而是由数据字段组成，通常作为键值

对或 JSON 格式的字符串。

在代码清单 6.2 中，我们定义了一个日志中间件(loggingMiddleware())来发出日志行，如下所示：

```
config.Logger.Printf(
      "protocol=%s path=%s method=%s duration=%f status=%d",
      r.Proto, r.URL.Path, r.Method,
      time.Now().Sub(startTime).Seconds(),
      customRw.code,
)
```

此应用程序发出的日志行由键值对组成，以空格分隔，例如 protocol=HTTP path=/api duration=0.05 status=200。这比简单地记录一行(比如收到/api/search 的 HTTP 请求)有所改进。大多数日志系统都内置了对这种格式的日志行的解析和索引的支持，因此每个键值对都可以单独搜索。在 Stripe 的一篇题为 Canonical Log Lines(https://stripe.com/blog/canonical-log-lines)的博文发表之后，这种格式在社区中被称为 logfmt。使用 Printf()或任何 log 包的函数构造上述日志行很麻烦。因此，可使用第三方包 https://github.com/go-logfmt/logfmt 构建一个格式化为键值对的日志行。这个包只实现了编码器和解码器，因此我们仍将使用标准库的日志功能来记录格式化的日志行。另一个第三方包 https://github.com/apex/log 支持以 logfmt 样式发出日志，我们可以使用它来代替标准库的 log 包。

实现结构化日志记录的另一种方法是使用将日志行作为 JSON 编码字符串发出的库。在这种格式中，前面的示例日志行如下所示：{"protocol":"HTTP", "path":"api","duration": 0.05, "status": 200}。这可能是最流行的结构化日志格式，并且有一些选项可以实现这种格式。最早的包之一是 https://github.com/sirupsen/logrus，它实现了一个与标准库的 log.Logger 类型实现的完全兼容的 API。近年来，已经开发了其他软件包，如 https://github.com/uber-go/zap 和 https://github.com/rs/zerolog，它们提供了更多功能和更好的性能。在下一节中，你将看到如何将 github.com/rs/zerolog 包集成到应用程序中。我们选择这个包而不是 zap，是因为它实现了一个更简单的 API。

接下来，我们将讨论如何从应用程序中导出指标。

指标是我们从应用程序计算和发布的数字，用于量化应用程序的各种行为。此类行为的示例包括从命令行应用程序执行命令所花费的时间、HTTP 请求或 gRPC 方法调用的延迟测量以及执行数据库操作所花费的时间。

指标通常是三个关键类别之一：计数器、仪表或直方图。计数器指标类型用于其值为整数且单调递增的度量——例如，应用程序在其生命周期内服务的请求数。计量指标用于可能随时间增加或减少的度量，它可以采用整数或浮点值——例如应用程序的内存使用量或应用程序每秒服务的请求数。直方图指标用于记录

观察结果，例如请求的延迟。与计量指标相比，直方图度量通常用于通过将指标值分组到桶中并启用诸如任意百分位值的计算进行分析。

一旦应用程序计算了指标，它们要么被发送到外部监控系统(推送模型)，要么监控系统将从应用程序中读取数据(拉取模型)。一旦数据存储在监控系统中，我们就可以查询它们，执行各种统计操作，并配置告警。

从历史上看，应用程序作者必须使用特定于供应商的库来使监控数据可用于监控系统。近年来，OpenTelemetry 项目(https://opentelemetry.io/)的开发使应用程序作者能够以供应商中立的方式导出指标，包括商业供应商。指标在哪里结束以及如何与应用程序的关注点分离。当我们更改监控系统时，应用程序代码保持不变。话虽如此，在撰写本书时，OpenTelemetry Go 社区(https://github.com/open-telemetry/opentelemetry-go)已决定降低指标支持的开发优先级，以专注于跟踪支持。因此，即使当前可以支持，我们也将避免使用它。相反，我们将使用 https://github.com/DataDog/datadog-go 以 statsd(https://github.com/ statsd/statsd)开源监控解决方案定义的格式直接从应用程序导出指标。即使你的组织没有直接使用 statsd，也很有可能正在使用的监控解决方案支持读取 statsd 指标格式。

接下来，我们将讨论如何从应用程序中导出跟踪。

traces 是跟踪系统中事务(transaction)的遥测数据。当一个请求进来时，作为处理该请求的一部分，通常会发生不止一个动作。跟踪的生命周期(trace's lifetime)与应用程序中事务的生命周期相同。在事务处理过程中发生的每个动作或事件都将开始一个跨度(span)。因此，跟踪由一个或多个跨度组成，可能跨越系统边界——例如，多个服务和数据库。与事务相关的所有跨度都将共享一个跟踪标识符，因此通过使用跟踪系统，可直观地分析作为事务的一部分执行的各种操作的延迟和成功/错误值。指标告诉我们事务很慢，而跟踪可提供更详细的信息来了解为什么它很慢。

例如，考虑我们在第 11 章中实现的包服务器。上传包是一个包含两个不同操作的事务：将包上传到对象存储服务并将包元数据更新到关系数据库。这些操作中的每一个都会发出一个跟踪，其中包含有关操作的详细信息以及操作所花费的时间。由于两个跟踪共享事务标识符，我们可以看到事务的总体延迟以及每个组成操作的延迟。这在面向服务的体系结构中最有用，在该体系结构中，我们在单个事务中发生多个服务调用。跟踪数据的存储和分析需要专门的系统，过去我们必须使用供应商特定的库来向它们发送跟踪数据。然而，OpenTelemetry 项目的 Go 库使得以供应商中立的方式实现跟踪成为可能。在撰写本书时，该项目(https://github.com/open-telemetry/opentelemetry-go)发布了 1.0.0-RC2 版本。我们将使用这个版本的库，并且只使用与导出跟踪相关的功能。

在下一节中，你将了解一些用于修改我们在书中编写的应用程序的模式，以便它们发出遥测数据。

发出遥测数据

我创建了一个自定义命令行客户端 pkgcli，用于与第 11 章中实现的包服务器交互。我还修改了包服务器以与 gRPC 服务器通信，以验证上传者的详细信息。你可以在本书源代码库的 appendix-a 目录中找到所有相关代码和说明。日志、指标和跟踪的存储需要专门的系统，并且有许多开源和商业解决方案。以下示例仅演示从应用程序发出相关数据，而不是这些数据的可视化和分析。你可以在文件 appendix-a/README.md 中找到运行演示命令行应用程序和服务器的完整说明。

命令行应用

对于命令行应用程序，我们将在执行任何命令之前配置日志记录、初始化网络客户端以在初始化期间导出指标和跟踪。然后，我们将这些初始化配置提供给应用程序的其余部分，以便在执行任何命令期间启用记录任何消息、发布指标或导出跟踪。命令行应用程序示例 pkgcli 的代码位于 appendix-a/command-line-app 目录中。它使用 flag 包，并应用第 2 章中讨论的子命令体系结构来创建具有两个子命令的应用程序——注册和查询。第一个子命令允许用户将包上传到包服务器，第二个命令允许从服务器查询包信息。我们创建了一个配置包，在里面有一个结构 PkgCliConfig 来封装初始化的日志配置、指标和跟踪客户端：

```
type PkgCliConfig struct {
      Logger zerolog.Logger
      Metrics telemetry.MetricReporter
      Tracer telemetry.TraceReporter
}
```

我们创建了一个遥测包，其中包含三个函数：InitLogging()、InitMetrics()和 InitTracing()，它们返回初始化的 zerolog.Logger、telemetry.MetricReporter 和 telemetry.TraceReporter 对象。最后两个自定义类型被定义为分别封装用于发布指标和导出跟踪的客户端。telemetry 包中的 logging.go 文件定义了 InitLogging()函数，如下所示：

```
package telemetry

import (
       "io"
       "github.com/rs/zerolog"
)

func InitLogging(
       w io.Writer, version string, logLevel int,
```

```
) zerolog.Logger {

        rLogger := zerolog.New(w)
        versionedL := rLogger.With().Str("version", version)
        timestampedL := versionedL.Timestamp().Logger()
        levelledL := timestampedL.Level(zerolog.Level(logLevel))

        return levelledL
}
```

我们从 github.com/rs/zerolog 包中调用 zerolog.New()函数来创建一个根记录器，一个将输出写入器(writer)设置为 w 的 zerolog.Logger 对象。然后调用 With()方法将日志上下文添加到根记录器。使用 Str()方法添加一个上下文，从根记录器 rLogger 创建一个子记录器，该方法将一个键(版本)添加到所有日志消息，其值设置为版本中的指定字符串。这将导致来自应用程序的所有日志在此字段中包含应用程序的版本。

接下来添加另一个日志记录上下文以向日志添加时间戳并创建另一个子记录器 timestampedL。最后通过调用 Level()方法添加分级日志记录的逻辑，并返回创建的 zerolog.Logger 对象。配置的记录器仅在其级别等于或大于 logLevel 中的值指示的配置级别时才会记录消息。logLevel 的值必须是介于-1 和 5 之间的整数(包括两个值)。将级别设置为-1 会记录所有消息，将级别设置为 5 会仅记录 panic 消息。

telemetry 包中的 metrics.go 文件定义了 InitMetrics()函数，如下所示：

```
package telemetry
import (
        "fmt"

        "github.com/DataDog/datadog-go/statsd"
)

type MetricReporter struct {
        statsd *statsd.Client
}

func InitMetrics(statsdAddr string) (MetricReporter, error) {
        var m MetricReporter
        var err error
        m.statsd, err = statsd.New(statsdAddr)
        if err != nil {
                return m, err
        }
        return m, nil
```

```
}
```

我们选择 github.com/DataDog/datadog-go/statsd 包，因为它维护得很好。通过调用 statsd.New()函数创建客户端，传递 statsd 服务器的地址，并将创建的客户端分配给 MetricReporter 对象的 statsd 字段。再次值得注意的是，一旦 OpenTelemetry 的指标支持准备就绪，将不会在应用程序中使用供应商特定的库。

定义了一个类型 DurationMetric 来封装一个命令执行持续时间的单一度量：

```
type DurationMetric struct {
      Cmd         string
      DurationMs float64
      Success     bool
}
```

然后定义一个方法 ReportDuration()，它将推送一个包含命令运行时间的直方图指标。持续时间以秒为单位。指标将添加两个标签，这将允许对指标进行分组和聚合。我们将 Cmd 中指定的已执行命令添加为一个标签，另一个标签指示该命令是否成功执行(如 Success 字段)。该方法的定义如下：

```
func (m MetricReporter) ReportDuration(metric DurationMetric) {
       metricName := "cmd.duration"
       m.statsd.Histogram(
              metricName,
              metric.DurationMs,
              []string{
                     fmt.Sprintf("cmd:%s", metric.Cmd),
                     fmt.Sprintf("success:%v", metric.Success),
              },
              1, //sample rate (0-none, 1 - all)
       )
}
```

可以定义类似的方法来推送其他指标类型。

InitTracing()方法在 telemetry 包内的 trace.go 文件中定义。在 telemetry 包中设置应用程序范围的跟踪配置，如下所示：

```
package telemetry

import (
      "context"

      "go.opentelemetry.io/otel"
      "go.opentelemetry.io/otel/trace"
      "go.opentelemetry.io/otel/exporters/jaeger"
```

```
        // TODO 导入其他 otel 包
)

type TraceReporter struct {
        Client trace.Tracer
        Ctx context.Context
}

func InitTracing(
        jaegerAddr string,
) (TraceReporter, *sdktrace.TracerProvider, error) {

        // (1) 设置跟踪(trace)导出器

        // (2) 设置跨度(span)处理器

        // (3) 设置跟踪器(tracer)提供商

        // (4) 创建传播器(propagator)

        // (5) 创建和配置跟踪器(Tracer)

        // (6) 返回一个 TraceReporter 类型的值

}
```

有六个关键步骤，如下所述。我们需要一个可以将跟踪导出的目的地，以便可以查看和查询它们。我们将使用 Jaeger(www.jaegertracing.io)，一个开源分布式跟踪系统，因此使用由 go.opentelemetry.io/otel/exporters/jaeger 包实现的 Jaeger 导出器。

以下代码片段创建跟踪导出器：

```
traceExporter, err := jaeger.New(
          jaeger.WithCollectorEndpoint(
                jaeger.WithEndpoint(jaegerAddr),
          ),
)
```

请注意，我们可以改用由 go.opentelemetry.io/otel/exporters/otlp 包实现的 OpenTelemetry 收集器导出器，并使应用程序对正在使用的分布式跟踪系统完全中立。但为了简单起见，我直接使用了 Jaeger 导出器。接下来设置跨度处理器来监督处理应用程序发出的跨度数据，并使用之前创建的跟踪导出器对其进行配置：

```
bsp := sdktrace.NewSimpleSpanProcessor(traceExporter)
```

对于生产级应用，建议配置不同的跨度处理器，创建的 Batch 跨度处理器如下：

```
 bsp := sdktrace.NewBatchSpanProcessor(traceExporter)
```

下一步是使用上述跨度处理器创建跟踪器提供程序：

```
tp := sdktrace.NewTracerProvider(
              sdktrace.WithSpanProcessor(bsp),
              sdktrace.WithResource(
                     resource.NewWithAttributes(
                            semconv.SchemaURL,
                            semconv.ServiceNameKey.String(
                                   "PkgServer-Cli",
                            ),
                     ),
              ),
       )
```

我们创建了名为 sdktrace.NewTraceProvider()的跟踪器提供程序，它具有两个参数。第一个是上一步中创建的跨度处理器。第二个参数标识创建由 OpenTelemetry 文档“资源语义约定”所描述的跟踪的应用程序。在这里，我们将产生这些跟踪的服务名称设置为 PkgServer-Cli。一旦像之前一样创建了跟踪器提供程序，就可以使用以下代码将其配置为当前应用程序的全局跟踪器提供程序：

```
otel.SetTracerProvider(tp)
```

接下来为跟踪设置全局传播器，这是当前应用程序的跟踪标识符传输到其他服务的方式：

```
propagator := propagation.NewCompositeTextMapPropagator(
      propagation.Baggage{},
      propagation.TraceContext{},
)
otel.SetTextMapPropagator(propagator)
The final steps are carried out by the code snippet below:
v1, err := baggage.NewMember("version", "version")
bag, err := baggage.New(v1)
tr.Client = otel.Tracer("")
ctx := context.Background()
tr.Ctx = baggage.ContextWithBaggage(ctx, bag)
return tr, tp, nil
```

通过 HTTP 与包服务器通信时，我们将使用 go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp 包提供的特殊配置的 HTTP 客户端：

```
// pkgregister/pkgregister.go
func RegisterPackage(
        ctx context.Context, cliConfig *config.PkgCliConfig,
```

```
            url string, data PkgData,
    ) (*PkgRegisterResult, error) {
          // 其他代码已删除

          r, err := http.NewRequestWithContext(
                ctx, http.MethodPost, url+"/api/packages",
                reader,
          )
          if err != nil {
                return nil, err
          }
          r.Header.Set("Content-Type", contentType)
          authToken := os.Getenv("X_AUTH_TOKEN")
          if len(authToken) != 0 {
                r.Header.Set("X-Auth-Token", authToken)
          }

          client := http.Client{
                Transport:
otelhttp.NewTransport(http.DefaultTransport),
          }
          resp, err := client.Do(r)

          // 处理响应
    }
```

当我们使用工具化的 HTTP 客户端时，跨度会在使用此客户端执行的 HTTP 请求响应事务期间自动发出。

在应用程序的 main()函数中，遥测配置的初始化如下所示：

```
    func main() {
          var tp *sdktrace.TracerProvider

          cliConfig.Logger = telemetry.InitLogging(
                os.Stdout, version, c.logLevel,
          )
          cliConfig.Metrics, err =
          telemetry.InitMetrics(c.statsdAddr)
          if err != nil {
                cliConfig.Logger.Fatal().Str("error",
                err.Error()).Msg(
                      "Error initializing metrics system",
                )
          }

          cliConfig.Tracer, tp, err = telemetry.InitTracing(
```

```
                c.jaegerAddr+"/api/traces", version,
        )
        if err != nil {
                cliConfig.Logger.Fatal().Str("error",
                err.Error()).Msg(
                        "Error initializing tracing system",
                )
        }
        defer func() {
                tp.ForceFlush(context.Background())
                tp.Shutdown(context.Background())
        }()

        err = handleSubCommand(cliConfig, os.Stdout, subCmdArgs)
        if err != nil {
                cliConfig.Logger.Fatal().Str("error",
                err.Error()).Msg(
                        "Error executing sub-command",
                )
        }
}
```

你可以看到如何使用初始化的记录器来记录结构化消息：

```
cliConfig.Logger.Fatal().Str("error", err.Error()).Msg(
              "Error initializing metrics system",
        )
```

日志消息将显示如下：

```
{"level":"fatal","version":"0.1","error":"lookup 127.0.0.: no
such host","time":"2021-09-11T12:22:23+10:00","message":
"Error initializing metrics system"}
```

在 cmd 包的 HandleRegister()函数中，你将找到分级日志记录的示例：

```
cliConfig.Logger.Info().Msg("Uploading package...")
cliConfig.Logger.Debug().Str("package_name", c.name).
         Str("package_version", c.version).
         Str("server_url", c.serverUrl)
```

为了报告命令运行的持续时间，我们实现了以下模式：

```
c.Logger = c.Logger.With().Str("command", "register").Logger()
tStart := time.Now()
defer func() {
        duration := time.Since(startTime).Seconds()
        c.Metrics.ReportDuration(
                telemetry.DurationMetric{
```

```
                    Cmd:       "pkgcli.register",
                    DurationMs: duration,
                    Success:    err == nil,
                },
        )
    }()
    err = cmd.HandleRegister(&c, w, args[1:])
```

在调用特定的子命令处理程序函数之前，我们更新记录器以创建一个新的上下文，这样所有日志都将有一个额外的字段 command 设置为 register，该字段将标识与 register 子命令关联的日志消息。

还启动了一个计时器，然后在一个延迟函数中，我们使用 ReportDuration()方法来报告命令运行的持续时间。

在 cmd 包中定义的 HandleRegister()函数中，我们在向包服务器发出 HTTP 请求之前创建一个跨度，如下所示：

```
ctx, span := cliConfig.Tracer.Client.Start(
        cliConfig.Tracer.Ctx,
        "pkgquery.register",
)
defer span.End()
```

接下来，让我们看看包服务器的工具化版本。

HTTP 应用

对于 HTTP 服务器应用程序，将配置日志记录，初始化网络客户端以在服务器启动期间导出指标和跟踪。然后将这些初始化配置提供给应用程序的其余部分，以便在执行任何命令期间启用记录任何消息、发布指标或导出跟踪。

修改后的包服务器的代码在 appendix-a/http-app 目录下。它建立在我们在第 11 章中实现的包服务器版本之上。

在 config 包中定义了一个结构 AppConfig 来封装应用的配置，包括遥测配置：

```
type AppConfig struct {
        PackageBucket *blob.Bucket
        Db        *sql.DB
        UsersSvc  users.UsersClient

        // 遥测
        Logger   zerolog.Logger
        Metrics  telemetry.MetricReporter
        Trace    trace.Tracer
        TraceCtx context.Context
        Span     trace.Span
```

```
        SpanCtx  context.Context
}
```

telemetry 包定义了初始化文件 logging.go、metrics.go 和 trace.go 中所有遥测配置的方法。InitLogging()和 InitMetrics()方法将与之前为命令行应用程序定义的相同。

middleware 包包含用于发送遥测数据的中间件的定义。日志中间件定义如下：

```
func LoggingMiddleware(
       c *config.AppConfig, h http.Handler,
) http.Handler {
       return http.HandlerFunc(func(
              w http.ResponseWriter, r *http.Request,
        ) {
              c.Logger.Printf("Got request-headers:%#v\n",
              r.Header)
              startTime := time.Now()
              h.ServeHTTP(w, r)
              c.Logger.Info().Str(
                     "protocol",
                     r.Proto,
              ).Str(
                     "path",
                     r.URL.Path,
              ).Str(
                     "method",
                     r.Method,
              ).Float64(
                     "duration",
                     time.Since(startTime).Seconds(),
              ).Send()
       })
}
```

你可在前面的代码片段中看到结构化日志记录的示例。我们通过逐步添加日志的键值字段并调用 Send()方法来构造日志行(内部是 zerolog.Event 值)，这会导致发出日志。上面定义的 LoggingMiddleware()函数发出的请求日志行将如下所示：

```
{"level":"info","version":"0.1","protocol":"HTTP/1.1",
"path":"/api/packages","method":"POST","duration":0.038707083,
"time":"2021-09-12T08:39:05+10:00"}
```

我们定义了另一个中间件来推动请求处理延迟：

```
func MetricMiddleware(c *config.AppConfig, h http.Handler)
```

```
http.Handler {
        return http.HandlerFunc(func(
                w http.ResponseWriter, r *http.Request,
        ) {
                startTime := time.Now()
                h.ServeHTTP(w, r)
                duration := time.Since(startTime).Seconds()
                c.Metrics.ReportLatency(
                        telemetry.DurationMetric{
                                DurationMs: duration,
                                Path:      r.URL.Path,
                                Method:    r.Method,
                        },
                )

        })
}
```

InitTracing()方法看起来略有不同：

```
func InitTracing(jaegerAddr string) error {
        traceExporter, err := jaeger.New(
                jaeger.WithCollectorEndpoint(
                        jaeger.WithEndpoint(jaegerAddr+"/api/traces"),
                ),
        )
        if err != nil {
                return err
        }
        bsp := sdktrace.NewSimpleSpanProcessor(traceExporter)

        tp := sdktrace.NewTracerProvider(
                sdktrace.WithSpanProcessor(bsp),
                sdktrace.WithResource(
                        resource.NewWithAttributes(
                                semconv.SchemaURL,
                                semconv.ServiceNameKey.String(
                                        "PkgServer",
                                ),
                        ),
                ),
        )
        otel.SetTracerProvider(tp)
        propagator := propagation.NewCompositeTextMapPropagator(
                propagation.Baggage{},
                propagation.TraceContext{},
        )
```

```
        otel.SetTextMapPropagator(propagator)
        return nil
}
```

我们配置全局跟踪提供程序，但不创建跟踪。与执行命令后退出的命令行应用程序不同，服务器将在其生命周期内服务多个请求。因此，我们使用专用中间件为每个请求创建跟踪。

TracingMiddleware()为每个新请求创建一个跟踪，它在中间件包中定义，如下所示：

```
func TracingMiddleware(
        c *config.AppConfig, h http.Handler,
) http.Handler {
        return http.HandlerFunc(func(
                w http.ResponseWriter, r *http.Request,
        ) {
                c.Trace = otel.Tracer("")
                tc := propagation.TraceContext{}
                incomingCtx := tc.Extract(
                        r.Context(),
                        propagation.HeaderCarrier(r.Header),
                )
                c.TraceCtx = incomingCtx

                ctx, span := c.Trace.Start(c.TraceCtx, r.URL.Path)
                c.Span = span
                c.SpanCtx = ctx
                defer c.Span.End()

                h.ServeHTTP(w, r)
        })
}
```

从请求中提取传入的上下文，然后使用该上下文启动一个新的跨度，并将跨度名称设置为正在处理的当前请求路径。在处理请求之后和从这个中间件返回之前结束跨度。值得注意的是，尽管 go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp 包定义了一个用于跟踪 HTTP 服务器的中间件，但这里定义了自己的中间件，因为这将允许我们创建自己的跨度。例如，在 storage 包的 UpdateDb() 函数中，会在开始数据库事务之前创建一个跨度，并在事务提交或回滚后结束它：

```
func UpdateDb(
        ctx context.Context,
        config *config.AppConfig,
        row types.PkgRow,
) error {
```

```
        conn, err := config.Db.Conn(ctx)
        if err != nil {
               return err
        }
        defer func() {
               err = conn.Close()
               if err != nil {
                      config.Logger.Debug().Msg(err.Error())
               }
        }()

        _, spanTx := config.Trace.Start(
               config.SpanCtx, "sql:transaction",
        )
        defer spanTx.End()

        tx, err := conn.BeginTx(ctx, nil)
        if err != nil {
               return err
        }

        // 剩余代码
}
```

将来可能提供 database/sql 包的自动工具化版本，那时我们将不必为数据库操作编写自己的手动跟踪代码。

在创建 gRPC 客户端以与 Users 服务通信时，我们利用自动工具化的库。在 server.go 中，你会发现以下函数：

```
func setupGrpcConn(addr string) (*grpc.ClientConn, error) {
       return grpc.DialContext(
             context.Background(),
             addr,
             grpc.WithInsecure(),
             grpc.WithUnaryInterceptor(
                    otelgrpc.UnaryClientInterceptor(),
             ),
             grpc.WithStreamInterceptor(
                    otelgrpc.StreamClientInterceptor(),
             ),
       )
}
```

go.opentelemetry.io/contrib/instrumentation/google.golang.org/grpc/otelgrpc 包定义了客户端和服务器端拦截器，以将 gRPC 应用程序与 OpenTelemetry 集成。

接下来，让我们看看 gRPC 服务器的插桩版本。

gRPC 应用

与 HTTP 服务器应用程序一样，我们将配置日志记录，初始化网络客户端以在服务器启动期间导出指标和跟踪。工具化的 gRPC 服务器的代码位于 appendix-a/grpc-server 目录中。它是我们在第 8 章中创建的 Users 服务的一个版本，它定义了一个单一的一元 RPC 方法 GetUser()。该服务的实现在文件 usersServiceHandler.go 中。还可以通过在 userService 结构中添加额外的字段来查看跨服务处理程序共享数据的示例，该结构是 Users 服务的实现：

```
type userService struct {
	users.UnimplementedUsersServer
	config config.AppConfig
}

func (s *userService) GetUser(
	ctx context.Context,
	in *users.UserGetRequest,
) (*users.UserGetReply, error) {
	s.config.Logger.Printf(
		"Received request for user verification: %s\n",
		in.Auth,
	)
	u := users.User{
		Id: rand.Int31n(4) + 1,
	}
	return &users.UserGetReply{User: &u}, nil
}
```

grpc.Server 对象在 main()函数中创建，如下所示：

```
s := grpc.NewServer(
	grpc.ChainUnaryInterceptor(
		interceptors.MetricUnaryInterceptor(&config),
		interceptors.LoggingUnaryInterceptor(&config),
		otelgrpc.UnaryServerInterceptor(),
	),
	grpc.ChainStreamInterceptor(
		interceptors.MetricStreamInterceptor(&config),
		interceptors.LoggingStreamInterceptor(&config),
		otelgrpc.StreamServerInterceptor(),
	),
)
```

我们已经在拦截器包中定义了指标和日志拦截器。还注册了由 go.opentelemetry.io/contrib/instrumentation/google.golang.org/grpc/otelgrpc 包定义的拦截器，它为我们

提供了 gRPC 服务处理程序调用的自动工具化。telemetry 包初始化日志记录、指标和跟踪配置。你会发现初始化代码与上一节中用于 HTTP 服务器的代码相同。

使用上述针对命令行应用程序、HTTP 服务器和 gRPC 服务器的工具，当我们发出上传包的请求时，将能看到从每个应用程序发出的日志、指标和跟踪。当然，跟踪还有一个额外的优势，即在整个事务中通过三个系统传播时以可视方式显示它。

小结

在本附录中，我们首先概述了日志、指标和跟踪。然后学习了检测命令行应用程序、HTTP 客户端和服务器以及 gRPC 应用程序的模式。我们使用 github.com/rs/zerolog 实现了结构化和分级的日志记录。之后，学会了使用 github.com/DataDog/datadog-go 从应用程序中以 statsd 格式导出度量值。最后，学会了使用 github.com/open-telemetry/opentelemetry-go 导出跟踪以关联跨系统边界的事务。

附录 B

部署应用程序

在本附录中，我们将讨论管理应用程序的配置、分发和部署的策略。遵循的特定策略通常取决于部署应用程序的基础架构。我的目标绝不是详尽无遗。相反，我只寻求为你提供通用指南。

管理配置

我们一直在使用命令行标志和环境变量来指定应用程序中的各种配置数据。配置数据是应用程序执行功能所需的信息。但是，用户不需要指定它们。例如，在附录 A 中，我们使用标志来指定指标(metrics)服务器的地址。但是，指标服务器的地址不会随着用户的输入而变化，例如要上传的包名称或版本。事实上，在大多数情况下，应用程序的用户很高兴不需要指定它，除非要专门覆盖它。

同样，我们使用环境变量来指定应用程序中的非敏感和敏感配置数据。例如，第 11 章中介绍的数据库密码被指定为环境变量。命令行标志以及环境变量很容易理解，并且不需要额外的库来支持应用程序。但是，随着应用程序的增长和配置数据的增加，你会发现希望使用其他方式来配置应用程序，甚至组合使用多种方法，例如使用命令行标志、环境变量和文件。例如，对非敏感数据使用配置文件是一种直接的方法。这需要编写额外的代码来读取配置文件并使它们可用于应用

程序。对于密码等敏感数据，可以继续使用环境变量。我们将看到一个使用以YAML 数据格式编写的配置文件来指定附录 A 中编写的包命令行客户端的配置的示例。我们将使用第三方包 https://pkg.go.dev/go.uber.org/config，它支持读取 YAML 格式的数据文件，包括组合多个 YAML 数据源以及读取环境变量。为命令行客户端指定了四个关键的配置数据——日志级别、指标服务器的地址、Jaeger(分布式跟踪服务器)的地址以及要使用的身份验证令牌。可在 YAML 格式的文件中指定这些关键信息，如下所示：

```
---
server:
 auth_token: ${X_AUTH_TOKEN}
telemetry:
 log_level: ${LOG_LEVEL:1}
 jaeger_addr: http://127.0.0.1:14268
 statsd_addr: 127.0.0.1:9125
```

该文件由两个顶级对象组成：server 和 telemetry。server 对象包含一个密钥 auth_token，它将通过环境变量 X_AUTH_TOKEN 指定，因为它被视为敏感数据。telemetry 对象定义了三个键：log_level、jaeger_addr 和 statsd_addr，分别包含日志级别以及 Jaeger 和 statsd 服务器的地址。log_level 的值将设置为 LOG_LEVEL 环境变量定义的值，如果指定，则默认为 1。日志级别的含义由 github.com/rs/zerolog 包确定，我们在附录 A 中使用过。

以下是在应用程序中读取上述数据的方法。将首先定义三种结构类型，配置文件将被反序列化为：

```
type serverCfg struct {
       AuthToken string `yaml:"auth_token"`
}

type telemetryCfg struct {
       LogLevel   int    `yaml:"log_level"`
       StatsdAddr string `yaml:"statsd_addr"`
       JaegerAddr string `yaml:"jaeger_addr"`
}

type pkgCliInput struct {
       Server    serverCfg
       Telemetry telemetryCfg
}
```

第一个结构类型 serverCfg 对应于 YAML 配置中的 server 对象。第二个结构体类型 telemetryCfg 对应于 YAML 配置中的 telemetry 对象，第三个结构体类型

pkgCliInput 对应于完整的 YAML 配置。结构标签`yaml: "auth_token"`用于指示相应的键名，因为它们将出现在 YAML 文件中。

接下来使用 go.uber.org/config 包读取包含 YAML 格式数据的配置文件：

```
import uberconfig "go.uber.org/config"

provider, err := uberconfig.NewYAML(
        uberconfig.File(configFilePath),
        uberconfig.Expand(os.LookupEnv),
)
```

NewYAML()函数接收一个或多个数据源。这里指定两个来源。第一个是通过 configFilePath 变量指定路径的文件。第二个是 Expand()函数，它接收具有此签名作为参数的函数：func(string)(string, bool)。这里指定标准库的 os.LookupEnv 函数作为调用 Expand()函数时的参数。结果是将环境变量和配置文件结合起来，为应用程序创建了一个合并的配置。值得注意的是，解析对象值的能力，例如 ${X_AUTH_TOKEN}和 ${LOG_LEVEL:1}，是由 go.uber.org/config 包的 Expand()函数实现的。

如果 NewYAML()函数调用返回 nil 错误，就可以读取数据了。以下代码片段将读取数据并尝试将其反序列化为 pkgCliInput 类型的对象：

```
c := pkgCliInput{}
if err := provider.Get(uberconfig.Root).Populate(&c); err != nil {
       return nil, err
}
```

如果前面的操作成功，就可以对读取的数据执行任何验证，例如：

```
if c.Telemetry.LogLevel < -1 || c.Telemetry.LogLevel > 5 {
       return nil, errors.New("invalid log level")
}
```

可在本书源代码库的 appendix-b/command-line-app 目录中找到使用上述逻辑读取应用程序配置的 pkgcli 的修改版本。特别是在 main.go 文件中，你会找到一个实现上述逻辑的函数 readConfig()和一个示例 config.yml 文件。

如果你想使用不同的文件格式以及其他方式来读取配置数据，github.com/spf13/viper 包值得研究。此外，如果你希望与云提供商的配置和密钥管理服务集成，Go Cloud Development Kit 项目提供的支持也值得研究。查看 runtimevar(https://gocloud.dev/howto/runtimevar/)和 secrets 的文档(https://gocloud.dev/howto/secrets/)。

分发应用程序

分发 Go 应用程序通常意味着分发编译的二进制文件，而不管分发机制如何。默认情况下，运行 go build，应用二进制格式对应于编译环境的操作系统和架构。Go 编译工具识别环境变量 GOOS 和 GOARCH 来编译特定操作系统和架构的二进制格式。因此，如果我们正在编译一个供其他人运行的应用程序，将需要为每个操作系统和硬件组合分发二进制文件。可通过指定 GOOS 和 GOARCH 环境变量来做到这一点。目前可以识别各种组合，例如 linux 和 arm64(适用于在 64 位 ARM 架构上运行的 Linux)、windows，以及 amd64(适用于在 AMD 或 Intel 64 位处理器上运行的 Windows)。如果要从 macOS 或 Linux 系统构建 Windows AMD64 二进制文件，请运行 go build 命令，如下所示：

```
$ GOOS=windows GOARCH=amd64 go build -o application.exe
```

编译的 application.exe 现在可以复制到另一台运行 Microsoft Windows 64 位操作系统的计算机上，并从那里运行。

除了二进制文件和配置文件之外，我们可能还需要为 Web 应用程序分发其他文件，例如模板或静态资产。我们可以使用标准库的嵌入包，而不是手动复制这些文件，从而使分发复杂化，它允许将文件嵌入编译的应用程序中。例如，考虑以下代码片段：

```
import _ "embed"
//go:embed templates/main.go.tmpl
var tmplMainGo []byte
```

编译包含此代码片段的应用程序时，变量 tmplMainGo(一个字节切片)将包含文件 templates/main.go.tmpl 的内容。因此，当运行应用程序时，该文件不需要存在，因为它已嵌入应用程序中。当然，这会导致应用程序可执行文件的大小增加，因此请注意你嵌入的文件。

一旦编译了应用程序，分发机制是另一个必须解决的问题。近年来，通过 Docker 容器实现的容器镜像变得流行起来，使得分发变得非常方便。构建 Docker 容器镜像的第一步是创建一个 Dockerfile，它是编译应用程序的一系列指令，然后将编译的应用程序复制到操作系统镜像(Linux 或 Windows)中。以下 Dockerfile 将构建一个包含命令行应用程序 pkgcli 和配置文件的映像：

```
FROM golang:1.16 as build
WORKDIR /go/src/app
COPY . .
RUN go get -d -v ./...
RUN go build
```

```
FROM golang:1.16
RUN useradd --create-home application
WORKDIR /home/application
COPY --from=build /go/src/app/pkgcli .
COPY config.yml .
USER application
ENTRYPOINT ["./pkgcli"]
```

在文件的第一个块中编译应用程序。在第二个块中创建一个包含已编译应用程序和 config.yml 文件的新镜像。有多种策略可以确保生成的最终镜像更小。除了使用 golang:1.16 作为基础镜像，我们可以使用特殊的暂存基础镜像或 https://github.com/GoogleContainerTools/distroless 项目提供的镜像之一。最后将镜像的 ENTRYPOINT 设置为 pkgcli，这是编译的应用程序二进制文件的名称。要构建镜像，请将 Dockerfile 保存在要编译的应用程序目录的根目录中，然后按如下方式运行 Docker 构建：

```
$ docker build -t practicalgo/pkgcli .
```

此命令将构建一个名为 practicego/pkgcli 的 Docker 镜像。构建镜像后，使用 docker push 命令将最终镜像推送到容器镜像注册表中。然后，任何想要使用它的人都可以从注册表中提取镜像并按如下方式运行它：

```
$ docker run -v /data/packages:/packages \
      -e X_AUTH_TOKEN=token-123 -ti practicalgo/pkgcli register \
      -name "test" -version 0.7 -path packages/file.tar.gz \
      http://127.0.0.1:8080
```

你可以看到如何使用 docker run 命令的-e 标志指定环境变量。使用-v 标志来指定卷挂载；也就是说，我们正在从容器内的主机系统挂载一个目录。在这里，我们正在挂载包含要上传的包的目录。你可在本书源代码存储库的 appendix-b/command-line-app 目录中找到包含 Dockerfile 和应用程序的完整示例。通常，会使用 Docker 镜像来分发服务器应用程序，但如果命令行应用程序想要分发默认配置文件，那么 Docker 镜像是一种便捷方式。

部署服务器端应用

将 HTTP 或 gRPC 服务器部署到公共网络或供其他人使用的内部网络时，我们应该运行应用程序的多个实例。可以直接在虚拟机上运行服务器，也可以将其作为容器运行，可能需要借助定制的解决方案或 Nomad 或 Kubernetes 等编排系统。

然后，我们应该配置一个负载平衡器来接收请求并将其转发到应用程序。因

此，充分理解负载平衡器和应用程序通信非常重要。

负载均衡器如何知道应用程序实例已准备好接收新请求呢？ 运行状况检查是负载均衡器检查应用程序实例运行状况的常用方法。因此，通常在应用程序中定义一个专用的 HTTP 端点或 gRPC 方法，负载均衡器可以定期向其发出请求以了解应用程序的健康状况。其他现代软件(例如服务网格)也扮演着与传统负载均衡器相同的角色，也会探测应用程序的健康状况，并将其作为是否转发到应用程序实例的流量决策中的考虑因素。如果应用程序实例的运行状况检查失败，它将停止接收新请求，并可自动将其删除并使用各种基础架构功能创建一个新请求。通常定义两类健康检查：健康检查和深度检查。第一次检查的成功响应仅确认应用程序本身能够响应检查。第二类检查更复杂，因为它将确保应用程序的依赖项(例如另一个服务或数据库)也可以访问，从而排除网络故障或凭据错误等问题。深度检查应该不那么频繁地运行，也许只在启动时运行，因为它要捕获的那些情况很可能在一开始就被捕获。

此外，本书各个章节中详细讨论过的超时配置必须引起足够的重视。必须特别注意云提供商的负载均衡器或你正在使用的其他类似软件的超时配置，因为它们决定了在终止连接之前等待响应的时间。当你使用长连接(例如在 gRPC 流通信期间)时，这一点也值得关注。

在第 7 章和第 10 章中，我们讨论了如何使用 HTTP 和 gRPC 应用程序的 TLS 证书来加密客户端和服务器之间的通信。在负载均衡器后面运行应用程序实例时，通常会在负载均衡器上终止来自客户端的 TLS 连接，因此负载均衡器和应用程序之间的通信是未加密的。这很常见，因为这样做很简单，并且存在一种错误的安全感，因为在大多数情况下，此流量都在组织的专用网络中。但值得强调的是，不建议这样做，我们必须希望能保护所有网络通信。我之前提到的一类软件(服务网格)在这方面通常很有用，因为它们可以自动确保加密的网络通信，而不需要应用程序做任何额外的工作。

小结

在本附录中，我们回顾了部署 Go 应用程序时的三个关键问题：管理配置、分发应用程序本身以及部署服务器应用程序。需要采取的具体步骤取决于部署应用程序的基础架构，但希望所讨论的策略能成为你进一步研究的良好起点。

附录 C

配置 Go 开发环境

安装 Go

本书中的代码适用于 Go 1.16 及以上版本。请按照 https://go.dev/learn/为你的操作系统安装最新版本的 Go 编译器。它通常涉及下载和运行 Windows 或 macOS 的图形安装过程。对于 Linux，发行版的包存储库可能已经包含最新版本，这意味着你也可以使用包管理器安装 Go 编译器。

安装完成后不需要进一步的配置就可以运行你在整本书中编写的程序。通过从终端程序运行命令 go version 来验证你是否已正确设置所有内容。你应该会看到输出告诉你安装了哪个 Go 版本以及操作系统和架构。例如，在我的 MacBook Air (M1)上看到以下内容：

```
$ go version
go version go1.16.4 darwin/arm64
```

如果可以看到如上所示的输出，则可以继续执行后续步骤。

选择一个编辑器

如果还没有喜欢的 Go 编辑器/集成开发环境 (IDE)，我推荐 Visual Studio Code

(https://code.visualstudio.com/download)。如果你是 Vim 用户，我推荐 vim-go 扩展(https://github.com/fatih/vim-go)。

安装 Protocol Buffer 工具链

对于本书的某些章节，你需要安装用于 Go 的 Protocol Buffer(protobuf)和 gRPC 工具。你将安装三个独立的程序：protobuf 编译器 protoc、Go protobuf 插件 protoc-gen-go 和 gRPC 插件 protoc-gen-go-grpc。

Linux 和 macOS

请针对 Linux 或 macOS 执行以下步骤来安装编译器。

(1) 从 https://github.com/protocolbuffers/protobuf/releases 下载与你的操作系统和架构对应的最新版本(本书撰写时为 3.16.0)文件。例如，对于 x86_64 系统上的 Linux，下载名为 protoc-3.16.0-linux-x86_64.zip 的文件。对于 macOS，下载名为 protoc-3.16.3-osx-x86_64.zip 的文件。

(2) 接下来，使用 unzip 命令提取文件内容并将它们复制到$HOME/.local 目录：$ unzip protoc-3.16.3-linux-x86_64.zip -d $HOME/.local。

(3) 最后，将$HOME/.local/bin 目录添加到$PATH 环境变量中：$ export PATH="$PATH:$HOME/.local/bin"(例如 Bash shell 下的$HOME/.bashrc 和 Z shell 下的.zshrc)。

完成上述步骤后，打开一个新的终端窗口，然后运行命令 protoc --version：

```
$ protoc --version
libprotoc 3.16.0
```

如果你看到与上面类似的输出，则你已准备好继续下一步。要为 Go 安装 protobuf 插件 protoc-gen-go(版本 v1.26)，请从终端窗口运行以下命令：

```
$ go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.26
```

运行以下命令安装 Go 的 gRPC 插件 protoc-gen-go-grpc(release v1.1)：

```
$ go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.1
```

然后将以下内容添加到 shell 的初始化文件($HOME/.bashrc 或$HOME/.zshrc)中：

```
$ export PATH="$PATH:$(go env GOPATH)/bin"
```

打开一个新的终端窗口，然后运行以下命令：

```
$ protoc-gen-go --version
```

```
protoc-gen-go v1.26.0
$ protoc-gen-go-grpc --version
protoc-gen-go-grpc 1.1.0
```

如果你看到类似上面的输出，则工具已成功安装。

Windows

要安装协议缓冲区(Protocol Buffer)编译器，请执行以下步骤(**注意**，你需要以管理员身份打开 Windows PowerShell 窗口才能执行这些步骤)。

(1) 从 https://github.com/protocolbuffers/protobuf/releases 下载与你的架构对应的最新版本(本书撰写时为 3.16.0)文件，查找名为 protoc-3.16.0-win64.zip 的文件。

(2) 然后创建一个目录用于存储编译器。例如，在 C:\Program Files 目录下执行：PS C:\> mkdir 'C:\Program Files\protoc-3.16.0'。

(3) 接下来，将下载的.zip 文件解压缩到该目录中。运行命令：PS C:\> Expand-Archive.\protoc-3.16.0-win64\DestinationPath 'C:\Program Files\protoc-3.16.0 '。

(4) 最后更新 Path 环境变量，添加上述路径： PS C:\> [Environment]:: SetEnvironmentVariable("Path", $env:Path + ";C:\Program Files\ protoc-3.16.0\bin", "Machine")。

打开一个新的 PowerShell 窗口，然后运行命令 protoc --version：

```
$ protoc --version
libprotoc 3.16.0
```

如果你看到与上述类似的输出，则已准备好继续下一步。

要为 Go 安装 protobuf 编译器 protoc-gen-go(版本 v1.26)，请从终端窗口运行以下命令：

```
C:\> go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.26
```

要安装 Go 的 gRPC 插件 protoc-gen-go-grpc(release v1.1)，请运行以下命令：

```
C:\> go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.1
```

打开一个新的 Windows PowerShell 窗口，并运行以下命令：

```
$ protoc-gen-go --version
protoc-gen-go v1.26.0
$ protoc-gen-go-grpc --version
protoc-gen-go-grpc 1.1.0
```

如果你看到类似上面的输出，则工具已成功安装。

安装 Docker Desktop

在本书的最后一章中，你需要在软件容器中运行应用程序。我们推荐使 Docker Desktop(https://www.docker.com/get-started)。对于 macOS 和 Windows 系统，请从上述网站下载与你的操作系统和架构对应的安装程序，然后按照说明完成安装。

对于 Linux，安装步骤会因你使用的发行版而异。有关特定发行版的详细步骤，请参阅 https://docs.docker.com/engine/install/#server。我还建议，为了便于使用(不推荐用于生产环境)，将 docker 安装配置为允许非 root 用户在不使用 sudo 的情况下运行容器。

按照特定操作系统的安装步骤操作后，运行以下命令从 Docker Hub 下载 docker 镜像，并运行它以确保安装已成功完成：

```
    $ docker run hello-world
    Unable to find image 'hello-world: latest' locally
    latest: Pulling from library/hello-world
    109db8fad215: Pull complete
    Digest:
sha256:0fe98d7debd9049c50b597ef1f85b7c1e8cc81f59c8d623fcb2250e8bec85
b38
    Status: Downloaded newer image for hello-world: latest
    Hello from Docker!
    This message shows that your installation appears to be
    working correctly.
    ..
```

这样就完成了本书所需软件的安装。接下来，我们将快速介绍整本书中使用的一些约定。