服务器优雅退出

作业题目

假设我们现在有一个 Web 服务。这个 Web 服务会监听两个端口：8080和8081。其中 8080 是用于监听正常的业务请求，它会被暴露在外部网络中；而 8081 用于监听我们开发者的内部管理请求，只在内部使用。

同时为了性能，我们在该服务中使用了本地缓存，并且采用了 write-back 的缓存模式。这个缓存模式要求，缓存在 key 过期的时候才将新值持久化到数据库中。这意味着在应用关闭的时候，我们必须将所有的 key 对应的数据都刷新到数据库中，否则会存在数据丢失的风险。

为了给用户更好的体验，我们希望你设计一个优雅退出的步骤，它需要完成：

- 监听系统信号，当收到 ctrl + C 的时候，应用要立刻拒绝新的请求
- 应用需要等待已经接收的请求被正常处理完成
- 应用关闭 8080 和 8081 两个服务器
- 我们能够注册一个退出的回调，在该回调内将缓存中的数据回写到数据库中

假设在我们的系统中有了几个基本结构和定义：

```
// 代表应用本身
type App struct {
  servers []*Server
}

// 代表一个 HTTP 服务器，一个实例监听一个端口
type Server struct {
  //
}
```

下面是更加详细的设计文档。

> 该设计文档是站在一个设计这样一个功能角度而撰写的，而不仅仅是为了作业，所以里面会有更加多的细节和要考虑的点

背景

优雅退出，是指应用主动监听关闭信号，在收到关闭信号如 ctrl + C 的时候，能够主动停下来。优雅退出和一般的直接退出比起来，优雅退出需要确保正在处理的请求能够正常结束，同时能够释放掉应用所使用的资源。

名词解释

名词	含义	注释
应用	这里指一个已经部署到了特定某个机器上的服务实例。 应用和实例，在这里会有相同的含义	
server	监听了某个端口，提供了对外接口的某个类型的实例	一般来说，应用和 server 是一对多的关系。即我们会在一个应用内启动多个 server，分别监听不同的端口。 举例来说，在 web 服务里面，我们会启动两个 server，一个 server 正常为用户提供服务；另外一个 server 则是暴露了管理接口，提供给开发人员使用

需求分析

场景分析

功能性需求

非功能性需求

设计

优雅退出过程：

在这个流程里面，我们将执行回调的时机放在关闭 server 之后，应用释放资源之前，是因为考虑到开发者执行回调的时候，可能依旧需要应用的资源；与此同时，我们又不希望开发者继续利用 server，所以放在这两个步骤之间。

同时我们在回调设计上，也没有引入优先级，或者排序的特性。**我们会并发调用回调，开发者自己需要确保这些回调之间没有依赖。**

详细设计

拒绝新请求

要拒绝新请求，在 `http.Server` 里面可以考虑封装 `ServeMux` ，在每一次处理请求之前，先检查一下当下需不需要拒绝新请求。

```
// serverMux 既可以看做是装饰器模式，也可以看做委托模式
type serverMux struct {
   reject bool
   *http.ServeMux
}
func (s *serverMux) ServeHTTP(w http.ResponseWriter, r *http.Request) {
   if s.reject {
      w.WriteHeader(http.StatusServiceUnavailable)
      _, _ = w.Write([]byte("服务已关闭"))
      return
   }
   s.ServeMux.ServeHTTP(w, r)
}
```

而后，当开始优雅退出的时候，将 `reject` 标记位设置为 `true`

> 为了减轻作业难度，这个部分我们已经写好了，你只需要阅读代码，可以稍微感受一下我们这里为什么采用装饰器模式

如果此时收到新请求，那么将会返回 503 响应。

等待已有请求执行完毕

等待已有请求执行完毕有两种思路：

- 等待一段固定时间
- 实时维护当前正在执行的请求数量

我们这里采用比较简单的方案一，只会固定等待一算时间，而开发者可以配置等待时间。

自定义选项和注册回调

在一些步骤中，开发者希望自定义一些参数，比如说等待时间。此外，我们还需要允许开发者注册自己的退出回调，这些需求可以通过 `Option` 设计模式来解决，例如注册回调：

```
type Option func(*App)

type ShutdownCallback func(ctx context.Context)

func WithShutdownCallbacks(cbs ...ShutdownCallback) Option {
   return func(app *App) {
      app.cbs = cbs
   }
}
type App struct {
    cbs []ShutdownCallback
}
```

回调本身是需要是需要考虑超时问题，既我们不希望开发者的回调长时间运行。同时我们也希望开发者明确意识到超时这个问题，所以我们采用了 context 的方案，开发者需要自己处理超时问题。

注册的自定义回调将会被并发调用。

监听系统信号

在 Go 里面监听系统信号是一件比较简单的事情。主要是依赖于

```
c := make(chan os.Signal, 1)
signal.Notify(c, signals)
select {
  case <-c:
  // 监听到了关闭信号
}
```

在不同系统下，需要监听的信号 `signals` 是不一样的。因此要可以利用 Go 编译器的特性，为不同的平台定义 `signals` 的值

- Mac OS（定义在以 `_darwin.go` 为结尾的文件中）

```
os.Interrupt, os.Kill, syscall.SIGKILL, syscall.SIGSTOP,
syscall.SIGHUP, syscall.SIGINT, syscall.SIGQUIT, syscall.SIGILL, syscall
syscall.SIGABRT, syscall.SIGSYS, syscall.SIGTERM,
```

- Windows（定义在以 `_windows.go` 为结尾的文件中）

```
os.Interrupt, os.Kill, syscall.SIGKILL,
syscall.SIGHUP, syscall.SIGINT, syscall.SIGQUIT, syscall.SIGILL, syscall
syscall.SIGABRT, syscall.SIGTERM,
```

- Linux （定义在以 `_linux.go` 为结尾的文件中）

```
os.Interrupt, os.Kill, syscall.SIGKILL, syscall.SIGSTOP,
syscall.SIGHUP, syscall.SIGINT, syscall.SIGQUIT, syscall.SIGILL, syscall
syscall.SIGABRT, syscall.SIGSYS, syscall.SIGTERM,
```

强制退出

采用两次监听的策略。在第一次监听到退出信号的信号的时候，启动优雅退出。之后需要做两件事：

- 再次监听系统退出信号，再次收到信号之后就强制退出
- 启动超时计时器，超时则强制退出

```
c := make(chan os.Signal, 1)
signal.Notify(c, signals)
select {
  case <-c:
  // 监听到了关闭信号
      shutdown() // 优雅退出
      go func() {
          select {
            case <- c:
            os.Exit(1) // 再次监听退出信号
            case time.AfterFunc(timeout, func(){
              // 超时控制
              os.Exit(1)
            })
          }
      }
}
```

例子

```
func main() {
   s1 := service.NewServer("business", "localhost:8080")
   s1.Handle("/", http.HandlerFunc(func(writer http.ResponseWriter, requ
      _, _ = writer.Write([]byte("hello"))
   }))
   s2 := service.NewServer("admin", "localhost:8081")
   app := service.NewApp([]*service.Server{s1, s2}, service.WithShutdown
   app.StartAndServe()
}
func StoreCacheToDBCallback(ctx context.Context) {
    // 需要处理 ctx 超时
}
```

测试

N/A

> 暂时不需要考虑测试，因为这个功能比较难自动化测试。

参考资料

业界方案

Kratos

代码在kratos/app.go at main · go-kratos/kratos (github.com)
Kratos 采用了类似的方案，但是 Kratos 做得更加精细化一点。它的核心代码是：

```
// Run executes all OnStart hooks registered with the application's Life
func (a *App) Run() error {
   instance, err := a.buildInstance()
   if err != nil {
      return err
   }
   ctx := NewContext(a.ctx, a)
   eg, ctx := errgroup.WithContext(ctx)
   wg := sync.WaitGroup{}
   for _, srv := range a.opts.servers {
      srv := srv
      eg.Go(func() error {
         <-ctx.Done() // wait for stop signal
         sctx, cancel := context.WithTimeout(context.Background(), a.opt
         defer cancel()
         return srv.Stop(sctx)
      })
      wg.Add(1)
      eg.Go(func() error {
         wg.Done()
         return srv.Start(ctx)
      })
   }
   wg.Wait()
   if a.opts.registrar != nil {
      rctx, rcancel := context.WithTimeout(a.opts.ctx, a.opts.registrarT
      defer rcancel()
      if err := a.opts.registrar.Register(rctx, instance); err != nil {
         return err
      }
      a.lk.Lock()
      a.instance = instance
      a.lk.Unlock()
   }
   c := make(chan os.Signal, 1)
   signal.Notify(c, a.opts.sigs...)
   eg.Go(func() error {
      for {
         select {
         case <-ctx.Done():
            return ctx.Err()
         case <-c:
            err := a.Stop()
            if err != nil {
               a.opts.logger.Errorf("failed to stop app: %v", err)
               return err
            }
         }
```

```
        }
    })
    if err := eg.Wait(); err != nil && !errors.Is(err, context.Canceled)
        return err
    }
    return nil
}
```

要点在于：

- 在启动 server 的时候同步启动了一个 goroutine 用于等待退出信号
- 退出信号使用 context.Done 来传递。因为 Kratos 同时考虑了启动失败之后，要通知其它启动成功的 server 退出。我们的设计里面没有考虑启动失败的问题；
- 监听信号：Kratos 允许用户主动指定监听的信号。但是对于一般的开发者来说，他们并不太清楚这些信号应该监听什么，以及在不同平台上怎么处理，所以我们的方案直接设计了在不同平台上的信号量。代价则是，缺乏可配置性；

Dubbo-go

代码在dubbo-go/graceful_shutdown_config.go at master · apache/dubbo-go (github.com)

Dubbo-go 的优雅退出从机制上来说类似于我们的方案，但是在具体退出的步骤上要更加复杂：

- 首先要从注册中心将本服务取消注册。这是微服务和 Web 服务首要的不同点，微服务服务端在退出之前，一定要先将自己从注册中心摘下来。如果 Web 服务前面有一个网关，那么 Web 服务也可以考虑主动通知网关，后续流量不再转发给自己
- 等待一段时间，这段时间还需要继续接收新请求。这与我们的方案也不同，因为Dubbo-go作为一个微服务框架，在取消注册之后，要等一段时间客户端才知道这个服务下线了，因此这一小段时间内的新请求，还需要继续接收
- 拒绝新请求，同时等待已有的请求结束。Dubbo-go 维持了活跃请求计数，所以它每十秒轮询一次，确定是不是所有请求都已经结束。如果都已经结束，或者超时，则步入下一个阶段
- 摧毁所有的协议实例。Dubbo-go是支持多协议的，所以在所有请求结束之后，就可以销毁这些协议实例了。绝大多数微服务框架没有这个步骤，因为它们只支持单一协议，如果 gRPC
- 执行开发者注册的回调。回调在这里并不是并发调用，而是挨个执行；

Dubbo-go 也是分平台定义了要监听的信号。