



下载APP



## 29 | 控制流（下）：iam-apiserver服务核心功能实现讲解

2021-07-31 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 32:44 大小 29.98M



你好，我是孔令飞。

🔗 **上一讲**，我介绍了 iam-apiserver 是如何构建 Web 服务的。这一讲，我们再来看下 iam-apiserver 中的核心功能实现。在对这些核心功能的讲解中，我会向你传达我的程序设计思路。

iam-apiserver 中包含了很多优秀的设计思想和实现，这些点可能比较零碎，但我觉得很值得分享给你。我将这些关键代码设计分为 3 类，分别是应用框架相关的特性、编程规范相关的特性和其他特性。接下来，我们就来详细看看这些设计点，以及它们背后的设计思想。



### 应用框架相关的特性

应用框架相关的特性包括三个，分别是优雅关停、健康检查和插件化加载中间件。

## 优雅关停

在讲优雅关停之前，先来看看不优雅的停止服务方式是什么样的。

当我们需要重启服务时，首先需要停止服务，这时可以通过两种方式来停止我们的服务：

在 Linux 终端键入 `Ctrl + C`（其实是发送 `SIGINT` 信号）。

发送 `SIGTERM` 信号，例如 `kill -9` 或者 `systemctl stop` 等。

当我们使用以上两种方式停止服务时，都会产生下面两个问题：

有些请求正在处理，如果服务端直接退出，会造成客户端连接中断，请求失败。

我们的程序可能需要做一些清理工作，比如等待进程内任务队列的任务执行完成，或者拒绝接受新的消息等。

这些问题都会对业务造成影响，所以我们需要一种优雅的方式来关停我们的应用。在 Go 开发中，通常通过拦截 `SIGINT` 和 `SIGTERM` 信号，来实现优雅关停。当收到这两个信号时，应用进程会做一些清理工作，然后结束阻塞状态，继续执行余下的代码，最后自然退出进程。

先来看一个简单的优雅关停的示例代码：

```
1 package main
2
3 import (
4     "context"
5     "log"
6     "net/http"
7     "os"
8     "os/signal"
9     "time"
10
11     "github.com/gin-gonic/gin"
12 )
13
14 func main() {
```

 复制代码

```
15     router := gin.Default()
16     router.GET("/", func(c *gin.Context) {
17         time.Sleep(5 * time.Second)
18         c.String(http.StatusOK, "Welcome Gin Server")
19     })
20
21     srv := &http.Server{
22         Addr:    ":8080",
23         Handler: router,
24     }
25
26     go func() {
27         // 将服务在 goroutine 中启动
28         if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClo
29             log.Fatalf("listen: %s\n", err)
30     }
31 }()
32
33 quit := make(chan os.Signal)
34 signal.Notify(quit, os.Interrupt)
35 <-quit // 阻塞等待接收 channel 数据
36 log.Println("Shutdown Server ...")
37
38 ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second) //
39 defer cancel()
40 if err := srv.Shutdown(ctx); err != nil { // 调用 net/http 包提供的优雅关闭函
41     log.Fatal("Server Shutdown:", err)
42 }
43 log.Println("Server exiting")
44 }
```

上面的代码实现优雅关停的思路如下：

1. 将 HTTP 服务放在 goroutine 中运行，程序不阻塞，继续执行。
2. 创建一个无缓冲的 channel quit，调用 signal.Notify(quit, os.Interrupt)。通过 signal.Notify 函数调用，可以将进程收到的 os.Interrupt ( SIGINT ) 信号，发送给 channel quit。
3. <-quit 阻塞当前 goroutine（也就是 main 函数所在的 goroutine），等待从 channel quit 接收关停信号。通过以上步骤，我们成功启动了 HTTP 服务，并且 main 函数阻塞，防止启动 HTTP 服务的 goroutine 退出。当我们键入 Ctrl + C 时，进程会收到 SIGINT 信号，并将该信号发送到 channel quit 中，这时候 <-quit 收到了 channel 另一端传来的数据，结束阻塞状态，程序继续执行。这里，<-quit 唯一目的是阻塞当前的 goroutine，所以对收到的数据直接丢弃。

4. 打印退出消息，提示准备退出当前服务。
5. 调用 net/http 包提供的 Shutdown 方法，Shutdown 方法会在指定的时间内处理完现有请求，并返回。
6. 最后，程序执行完 `log.Println("Server exiting")` 代码后，退出 main 函数。

**iam-apiserver 也实现了优雅关停，优雅关停思路跟上面的代码类似。**具体可以分为三个步骤，流程如下：

**第一步**，创建 channel 用来接收 `os.Interrupt ( SIGINT )` 和 `syscall.SIGTERM ( SIGKILL )` 信号。

代码见 [internal/pkg/server/signal.go](#)。

 复制代码

```
1 var onlyOneSignalHandler = make(chan struct{})
2
3 var shutdownHandler chan os.Signal
4
5 func SetupSignalHandler() <-chan struct{} {
6     close(onlyOneSignalHandler) // panics when called twice
7
8     shutdownHandler = make(chan os.Signal, 2)
9
10    stop := make(chan struct{})
11
12    signal.Notify(shutdownHandler, shutdownSignals...)
13
14    go func() {
15        <-shutdownHandler
16        close(stop)
17        <-shutdownHandler
18        os.Exit(1) // second signal. Exit directly.
19    }()
20
21    return stop
22 }
```

SetupSignalHandler 函数中，通过 `close(onlyOneSignalHandler)` 来确保 iam-apiserver 组件的代码只调用一次 SetupSignalHandler 函数。否则，可能会因为信号传给了不同的 shutdownHandler，而造成信号丢失。

SetupSignalHandler 函数还实现了一个功能：收到一次 SIGINT/ SIGTERM 信号，程序优雅关闭。收到两次 SIGINT/ SIGTERM 信号，程序强制关闭。实现代码如下：

[复制代码](#)

```
1 go func() {
2     <-shutdownHandler
3     close(stop)
4     <-shutdownHandler
5     os.Exit(1) // second signal. Exit directly.
6 }()
```

这里要注意：signal.Notify(c chan<- os.Signal, sig ...os.Signal)函数不会为了向 c 发送信息而阻塞。也就是说，如果发送时 c 阻塞了，signal 包会直接丢弃信号。为了不丢失信号，我们创建了有缓冲的 channel shutdownHandler。

最后，SetupSignalHandler 函数会返回 stop，后面的代码可以通过关闭 stop 来结束代码的阻塞状态。

**第二步**，将 channel stop 传递给启动 HTTP (S)、gRPC 服务的函数，在函数中以 goroutine 的方式启动 HTTP (S)、gRPC 服务，然后执行 <-stop 阻塞 goroutine。

**第三步**，当 iam-apiserver 进程收到 SIGINT/SIGTERM 信号后，关闭 stop channel，继续执行 <-stop 后的代码，在后面的代码中，我们可以执行一些清理逻辑，或者调用 google.golang.org/grpc 和 net/http 包提供的优雅关停函数 GracefulStop 和 Shutdown。例如下面这个代码（位于 [internal/apiserver/grpc.go](#) 文件中）：

[复制代码](#)


```
1 func (s *grpcAPIServer) Run(stopCh <-chan struct{}) {
2     listen, err := net.Listen("tcp", s.address)
3     if err != nil {
4         log.Fatalf("failed to listen: %s", err.Error())
5     }
6
7     log.Infof("Start grpc server at %s", s.address)
8
9     go func() {
10         if err := s.Serve(listen); err != nil {
11             log.Fatalf("failed to start grpc server: %s", err.Error())
12         }
13     }()
```

```
13     }()  
14  
15     <-stopCh  
16  
17     log.Infof("Grpc server on %s stopped", s.address)  
18     s.GracefulStop()  
19 }
```

除了上面说的方法，iam-apiserver 还通过

[github.com/marmotedu/iam/pkg/shutdown](https://github.com/marmotedu/iam/pkg/shutdown) 包，实现了另外一种优雅关停方法，这个方法更加友好、更加灵活。实现代码见 [PrepareRun](#) 函数。

[github.com/marmotedu/iam/pkg/shutdown](https://github.com/marmotedu/iam/pkg/shutdown) 包的使用方法如下：

 复制代码


```
1 package main  
2 import (  
3     "fmt"  
4     "time"  
5     "github.com/marmotedu/iam/pkg/shutdown"  
6     "github.com/marmotedu/iam/pkg/shutdown/shutdownmanagers/posixsignal"  
7 )  
8 func main() {  
9     // initialize shutdown  
10    gs := shutdown.New()  
11    // add posix shutdown manager  
12    gs.AddShutdownManager(posixsignal.NewPosixSignalManager())  
13    // add your tasks that implement ShutdownCallback  
14    gs.AddShutdownCallback(shutdown.ShutdownFunc(func(string) error {  
15        fmt.Println("Shutdown callback start")  
16        time.Sleep(time.Second)  
17        fmt.Println("Shutdown callback finished")  
18        return nil  
19    }))  
20    // start shutdown managers  
21    if err := gs.Start(); err != nil {  
22        fmt.Println("Start:", err)  
23        return  
24    }  
25    // do other stuff  
26    time.Sleep(time.Hour)  
27 }
```

上面的代码中，通过 `gs := shutdown.New()` 创建 `shutdown` 实例；通过 `AddShutdownManager` 方法添加监听的信号；通过 `AddShutdownCallback` 方法设置监听到指定信号时，需要执行的回调函数。这些回调函数可以执行一些清理工作。最后，通过 `Start` 方法启动 `shutdown` 实例。

## 健康检查

通常，我们会根据进程是否存在来判定 `iam-apiserver` 是否健康，例如执行 `ps -ef|grep iam-apiserver`。在实际开发中，我发现有时候服务进程仍然存在，但是 HTTP 服务却不能接收和处理请求，所以更加靠谱的检查方法是，直接请求 `iam-apiserver` 的健康检查接口。

我们可以在启动 `iam-apiserver` 进程后，手动调用 `iam-apiserver` 健康检查接口进行检查。但还有更方便的方法：启动服务后自动调用健康检查接口。这个方法的具体实现，你可以查看 `GenericAPIServer` 提供的 `ping` 方法。在 `ping` 方法中，你需要注意函数中的如下代码：

 复制代码

```
1 url := fmt.Sprintf("http://%s/healthz", s.InsecureServingInfo.Address)
2 if strings.Contains(s.InsecureServingInfo.Address, "0.0.0.0") {
3     url = fmt.Sprintf("http://127.0.0.1:%s/healthz", strings.Split(s.InsecureS
4 }
```

当 HTTP 服务监听在所有网卡时，请求 IP 为 `127.0.0.1`；当 HTTP 服务监听在指定网卡时，我们需要请求该网卡的 IP 地址。

## 插件化加载中间件

`iam-apiserver` 支持插件化地加载 `Gin` 中间件，通过这种插件机制，我们可以根据需要选择中间件。

那么，为什么要将中间件做成一种插件化的机制呢？一方面，每个中间件都完成某种功能，这些功能不是所有情况下都需要的；另一方面，中间件是追加在 HTTP 请求链路上的一个处理函数，会影响 API 接口的性能。为了保证 API 接口的性能，我们也需要选择性地加载中间件。



例如，在测试环境中为了方便 Debug，可以选择加载 dump 中间件。dump 中间件可以打印请求包和返回包信息，这些信息可以协助我们 Debug。但是在现网环境中，我们不需要 dump 中间件来协助 Debug，而且如果加载了 dump 中间件，请求时会打印大量的请求信息，严重影响 API 接口的性能。这时候，我们就期望中间件能够按需加载。

iam-apiserver 通过 [🔗 InstallMiddlewares](#) 函数来安装 Gin 中间件，函数代码如下：

[📄 复制代码](#)

```
1 func (s *GenericAPIServer) InstallMiddlewares() {
2     // necessary middlewares
3     s.Use(middleware.RequestID())
4     s.Use(middleware.Context())
5
6     // install custom middlewares
7     for _, m := range s.middlewares {
8         mw, ok := middleware.Middlewares[m]
9         if !ok {
10             log.Warnf("can not find middleware: %s", m)
11
12             continue
13         }
14
15         log.Infof("install middleware: %s", m)
16         s.Use(mw)
17     }
18 }
```

可以看到，安装中间件时，我们不仅安装了一些必备的中间件，还安装了一些可配置的中间件。

上述代码安装了两个默认的中间件：[🔗 RequestID](#) 和 [🔗 Context](#)。

RequestID 中间件，主要用来在 HTTP 请求头和返回头中设置 X-Request-ID Header。如果 HTTP 请求头中没有 X-Request-ID HTTP 头，则创建 64 位的 UUID，如果有就复用。UUID 是调用 [github.com/satori/go.uuid](https://github.com/satori/go.uuid) 包提供的 `NewV4().String()` 方法来生成的：

[📄 复制代码](#)

```
1 rid = uuid.NewV4().String()
```



另外，这里有个 Go 常量的设计规范需要你注意：常量要跟该常量相关的功能包放在一起，不要将一个项目的常量都集中放在 `const` 这类包中。例如，[requestid.go](#) 文件中，我们定义了 `XRequestIDKey = "X-Request-ID"` 常量，其他地方如果需要使用 `XRequestIDKey`，只需要引入 `XRequestIDKey` 所在的包，并使用即可。

Context 中间件，用来在 `gin.Context` 中设置 `requestID` 和 `username` 键，在打印日志时，将 `gin.Context` 类型的变量传递给 `log.L()` 函数，`log.L()` 函数会在日志输出中输出 `requestID` 和 `username` 域：

 复制代码

```
1 2021-07-09 13:33:21.362 DEBUG    apiserver      v1/user.go:106  get 2 users fr
```

`requestID` 和 `username` 字段可以方便我们后期过滤并查看日志。

除了默认的中间件，`iam-apiserver` 还支持一些可配置的中间件，我们可以通过配置 `iam-apiserver` 配置文件中的 [server.middlewares](#) 配置项，来配置这些中间件。

可配置以下中间件：

`recovery`：捕获任何 `panic`，并恢复。


`secure`：添加一些安全和资源访问相关的 HTTP 头。

`nocache`：禁止客户端缓存 HTTP 请求的返回结果。

`cors`：HTTP 请求跨域中间件。

`dump`：打印出 HTTP 请求包和返回包的内容，方便 `debug`。注意，生产环境禁止加载该中间件。

当然，你还可以根据需要，添加更多的中间件。方法很简单，只需要编写中间件，并将中间件添加到一个 `map[string]gin.HandlerFunc` 类型的变量中即可：

 复制代码

```
1 func defaultMiddlewares() map[string]gin.HandlerFunc {
2     return map[string]gin.HandlerFunc{
3         "recovery": gin.Recovery(),
```

```
4      "secure":    Secure,
5      "options":   Options,
6      "nocache":   NoCache,
7      "cors":      Cors(),
8      "requestid": RequestID(),
9      "logger":    Logger(),
10     "dump":      gindump.Dump(),
11   }
12 }
```

上述代码位于 [internal/pkg/middleware/middleware.go](#) 文件中。

## 编程规范相关的特性

编程规范相关的特性有四个，分别是 API 版本、统一的资源元数据、统一的返回、并发处理模板。

### API 版本

RESTful API 为了方便以后扩展，都需要支持 API 版本。在 [12 讲](#) 中，我们介绍了 API 版本号的 3 种标识方法，iam-apiserver 选择了将 API 版本号放在 URL 中，例如/v1/secrets。放在 URL 中的好处是很直观，看 API 路径就知道版本号。另外，API 的路径也可以很好地跟控制层、业务层、模型层的代码路径相映射。例如，密钥资源相关的代码存放位置如下：

 复制代码

```
1 internal/apiserver/controller/v1/secret/ # 控制层代码存放位置
2 internal/apiserver/service/v1/secret.go # 业务层代码存放位置
3 github.com/marmotedu/api/apiserver/v1/secret.go # 模型层代码存放位置
```

关于代码存放路径，我还有一些地方想跟你分享。对于 Secret 资源，通常我们需要提供 CRUD 接口。

C : Create ( 创建 Secret )。

R : Get ( 获取详情 )、List ( 获取 Secret 资源列表 )。

U : Update ( 更新 Secret )。

D : Delete ( 删除指定的 Secret )、DeleteCollection ( 批量删除 Secret )。

每个接口相互独立，为了减少更新 A 接口代码时因为误操作影响到 B 接口代码的情况，这里建议 CRUD 接口每个接口一个文件，从物理上将不同接口的代码隔离开。这种接口还可以方便我们查找 A 接口的代码所在位置。例如，Secret 控制层相关代码的存放方式如下：

[复制代码](#)

```
1 $ ls internal/apiserver/controller/v1/secret/  
2 create.go delete_collection.go delete.go doc.go get.go list.go secret.go
```

业务层和模型层的代码也可以这么组织。iam-apiserver 中，因为 Secret 的业务层和模型层代码比较少，所以我放在了 internal/apiserver/service/v1/secret.go 和 github.com/marmotedu/api/apiserver/v1/secret.go 文件中。如果后期 Secret 业务代码增多，我们也可以修改成下面这种方式：

[复制代码](#)

```
1 $ ls internal/apiserver/service/v1/secret/  
2 create.go delete_collection.go delete.go doc.go get.go list.go secret.g
```

这里再说个题外话：/v1/secret/和/secret/v1/这两种目录组织方式都可以，你选择一个自己喜欢的就行。

当我们需要升级 API 版本时，相关代码可以直接放在 v2 目录下，例如：

[复制代码](#)

```
1 internal/apiserver/controller/v2/secret/ # v2 版本控制层代码存放位置  
2 internal/apiserver/service/v2/secret.go # v2 版本业务层代码存放位置  
3 github.com/marmotedu/api/apiserver/v2/secret.go # v2 版本模型层代码存放位置
```

这样既能够跟 v1 版本的代码物理隔离开，互不影响，又方便查找 v2 版本的代码。

## 统一的资源元数据

iam-apiserver 设计的一大亮点是，像 Kubernetes REST 资源一样，支持统一的资源元数据。

iam-apiserver 中所有的资源都是 REST 资源，iam-apiserver 将 REST 资源的属性也进一步规范化了，这里的规范化是指所有的 REST 资源均支持两种属性：

公共属性。

资源自有的属性。

例如，Secret 资源的定义方式如下：

[复制代码](#)

```
1 type Secret struct {
2     // May add TypeMeta in the future.
3     // metav1.TypeMeta `json:",inline"`
4
5     // Standard object's metadata.
6     metav1.ObjectMeta `json:"metadata,omitempty"`
7     Username           string `json:"username"          gorm:"column:username"
8     SecretID           string `json:"secretID"           gorm:"column:secretID"
9     SecretKey          string `json:"secretKey"          gorm:"column:secretKey"
10
11     // Required: true
12     Expires            int64  `json:"expires"           gorm:"column:expires"   validate:
13     Description string `json:"description" gorm:"column:description" validate:
14 }
```

资源自有的属性，会因资源不同而不同。这里，我们来重点看一下公共属性

[ObjectMeta](#)，它的定义如下：

[复制代码](#)

```
1 type ObjectMeta struct {
2     ID uint64 `json:"id,omitempty" gorm:"primary_key;AUTO_INCREMENT;column:id"`
3     InstanceID string `json:"instanceID,omitempty" gorm:"unique;column:instanceID"`
4     Name string `json:"name,omitempty" gorm:"column:name;type:varchar(64);not null"`
5     Extend Extend `json:"extend,omitempty" gorm:"- validate:"omitempty"`
6     ExtendShadow string `json:"-" gorm:"column:extendShadow validate:"omitempty"`
7     CreatedAt time.Time `json:"createdAt,omitempty" gorm:"column:createdAt"`
8     UpdatedAt time.Time `json:"updatedAt,omitempty" gorm:"column:updatedAt"`
9 }
```

接下来，我来详细介绍公共属性中每个字段的含义及作用。

## 1. ID


这里的 ID，映射为 MariaDB 数据库中的 id 字段。id 字段在一些应用中，会作为资源的唯一标识。但 iam-apiserver 中没有使用 ID 作为资源的唯一标识，而是使用了 InstanceID。iam-apiserver 中 ID 唯一的作用是跟数据库 id 字段进行映射，代码中并没有使用到 ID。

## 2. InstanceID

InstanceID 是资源的唯一标识，格式为<resource identifier>-xxxxxx。其中，<resource identifier>是资源的英文标识符号，xxxxxx是随机字符串。字符集合为abcdefghijklmnopqrstuvwxyz1234567890，长度 >=6，例如 secret-yj8m30、user-j4lz3g、policy-3v18jq。

腾讯云、阿里云、华为云也都是采用这种格式的字符串作为资源唯一标识的。

InstanceID 的生成和更新都是自动化的，通过 gorm 提供的 AfterCreate Hooks 在记录插入数据库之后，生成并更新到数据库的 instanceID字段：

 复制代码

```
1 func (s *Secret) AfterCreate(tx *gorm.DB) (err error) {
2     s.InstanceID = idutil.GetInstanceID(s.ID, "secret-")
3
4     return tx.Save(s).Error
5 }
```

上面的代码，在 Secret 记录插入到 iam 数据库的 secret 表之后，调用 idutil.GetInstanceID生成 InstanceID，并通过 tx.Save(s)更新到数据库 secret 表的 instanceID字段。

因为通常情况下，应用中的 REST 资源只会保存到数据库中的一张表里，这样就能保证应用中每个资源的数据库 ID 是唯一的。所以 GetInstanceID(uid uint64, prefix string) string函数使用 github.com/speps/go-hashids包提供的方法，对这个数据库 ID 进行哈希，最终得到一个数据库级别的唯一的字符串（例如：3v18jq），并根据传入的 prefix，得到资源的 InstanceID。

使用这种方式生成资源的唯一标识，有下面这两个优点：

数据库级别唯一。

InstanceID 是长度可控的字符串，长度最小是 6 个字符，但会根据表中的记录个数动态变长。根据我的测试，2176782336 条记录以内生成的 InstanceID 长度都在 6 个字符以内。长度可控的另外一个好处是方便记忆和传播。

这里需要你注意：如果同一个资源分别存放在不同的表中，那在使用这种方式时，生成的 InstanceID 可能相同，不过概率很小，几乎为零。这时候，我们就需要使用分布式 ID 生成技术。这又是另外一个话题了，这里不再扩展讲解。

在实际的开发中，不少开发者会使用数据库数字段 ID（例如 121）和 36/64 位的 UUID（例如 20cd59d4-08c6-4e86-a9d4-a0e51c420a04）来作为资源的唯一标识。相较于这两种资源标识方式，使用<resource identifier>-xxxxxx这种标识方式具有以下优点：

看标识名就知道是什么类型的资源，例如：secret-yj8m30说明该资源是 secret 类型的资源。在实际的排障过程中，能够有效减少误操作。

长度可控，占用数据库空间小。iam-apiserver 的资源标识长度基本可以认为是 12 个字符（secret/policy 是 6 个字符，再加 6 位随机字符）。

如果使用 121 这类数值作为资源唯一标识，相当于间接向友商透漏系统的规模，是一定要禁止的。

另外，还有一些系统如 Kubernetes 中，使用资源名作为资源唯一标识。这种方式有个弊端，就是当系统中同类资源太多时，创建资源很容易重名，你自己想要的名字往往填不了，所以 iam-apiserver 不采用这种设计方式。

我们使用 instanceID 来作为资源的唯一标识，在代码中，就经常需要根据 instanceID 来查询资源。所以，在数据库中要设置该字段为唯一索引，一方面可以防止 instanceID 不唯一，另一方面也能加快查询速度。

### 3. Name


Name 即资源的名字，我们可以通过名字很容易地辨别一个资源。

#### 4. Extend、ExtendShadow

Extend 和 ExtendShadow 是 iam-apiserver 设计的又一大亮点。


在实际开发中，我们经常会遇到这个问题：随着业务发展，某个资源需要增加一些属性，这时，我们可能会选择在数据库中新增一个数据库字段。但是，随着业务系统的演进，数据库中的字段越来越多，我们的 Code 也要做适配，最后就会越来越难维护。

我们还可能遇到这种情况：我们将上面说的字段保存在数据库中叫 meta 的字段中，数据库中 meta 字段的数据格式是{"disable":true,"tag":"colin"}。但是，我们如果想在代码中使用这些字段，需要 Unmarshal 到一个结构体中，例如：

 复制代码

```
1 metaData := `{"disable":true,"tag":"colin"}`
2 meta := make(map[string]interface{})
3 if err := json.Unmarshal([]byte(metaData), &meta); err != nil {
4     return err
5 }
```

再存入数据中时，又要 Marshal 成 JSON 格式的字符串，例如：

 复制代码

```
1 meta := map[string]interface{}{"disable": true, "tag": "colin"}
2 data, err := json.Marshal(meta)
3 if err != nil {
4     return err
5 }
```

你可以看到，这种 Unmarshal 和 Marshal 操作有点繁琐。

因为每个资源都可能需要用到扩展字段，那么有没有一种通用的解决方案呢？iam-apiserver 就通过 Extend 和 ExtendShadow 解决了这个问题。



Extend 是 Extend 类型的字段，Extend 类型其实是 `map[string]interface{}` 的类型别名。在程序中，我们可以很方便地引用 Extend 包含的属性，也就是 map 的 key。Extend 字段在保存到数据库中时，会自动 Marshal 成字符串，保存在 ExtendShadow 字段中。

ExtendShadow 是 Extend 在数据库中的影子。同样，当从数据库查询数据时，ExtendShadow 的值会自动 Unmarshal 到 Extend 类型的变量中，供程序使用。

具体实现方式如下：

1. 借助 gorm 提供的 BeforeCreate、BeforeUpdate Hooks，在插入记录、更新记录时，将 Extend 的值转换成字符串，保存在 ExtendShadow 字段中，并最终保存在数据库的 ExtendShadow 字段中。
2. 借助 gorm 提供的 AfterFind Hooks，在查询数据后，将 ExtendShadow 的值 Unmarshal 到 Extend 字段中，之后程序就可以通过 Extend 字段来使用其中的属性。

## 5. CreatedAt

资源的创建时间。每个资源在创建时，我们都应该记录资源的创建时间，可以帮助后期进行排障、分析等。

## 6. UpdatedAt

资源的更新时间。每个资源在更新时，我们都应该记录资源的更新时间。资源更新时，该字段由 gorm 自动更新。

可以看到，ObjectMeta 结构体包含了很多字段，每个字段都完成了很酷的功能。那么，如果把 ObjectMeta 作为所有资源的公共属性，这些资源就会自带这些能力。

当然，有些开发者可能会说，User 资源其实是不需要 `user-xxxxxx` 这种资源标识的，所以 InstanceID 这个字段其实是无用的字段。但是在我看来，和功能冗余相比，功能规范化、不重复造轮子，以及 ObjectMeta 的其他功能更加重要。所以，也建议所有的 REST 资源都使用统一的资源元数据。

## 统一的返回

在 [🔗 18 讲](#) 中，我们介绍过 API 的接口返回格式应该是统一的。要想返回一个固定格式的消息，最好的方式就是使用同一个返回函数。因为 API 接口都是通过同一个函数来返回的，其返回格式自然是统一的。

IAM 项目通过 [🔗 github.com/marmotedu/component-base/pkg/core](https://github.com/marmotedu/component-base/pkg/core) 包提供的 [🔗 WriteResponse](#) 函数来返回结果。WriteResponse 函数定义如下：

[📄 复制代码](#)

```
1 func WriteResponse(c *gin.Context, err error, data interface{}) {
2     if err != nil {
3         log.Errorf("%#+v", err)
4         coder := errors.ParseCoder(err)
5         c.JSON(coder.HTTPStatus(), ErrResponse{
6             Code:      coder.Code(),
7             Message:   coder.String(),
8             Reference: coder.Reference(),
9         })
10
11     return
12 }
13
14 c.JSON(http.StatusOK, data)
15 }
```

可以看到，WriteResponse 函数会判断 err 是否为 nil。如果不为 nil，则将 err 解析为 [github.com/marmotedu/errors](https://github.com/marmotedu/errors) 包中定义的 Coder 类型的错误，并调用 Coder 接口提供的 Code()、String()、Reference() 方法，获取该错误的业务码、对外展示的错误信息和排障文档。如果 err 为 nil，则调用 c.JSON 返回 JSON 格式的数据。

## 并发处理模板

在 Go 项目开发中，经常会遇到这样一种场景：查询列表接口时，查询出了多条记录，但是需要针对每一条记录做一些其他逻辑处理。因为是多条记录，比如 100 条，处理每条记录延时如果为 X 毫秒，串行处理完 100 条记录，整体延时就是  $100 * X$  毫秒。如果 X 比较大，那整体处理完的延时是非常高的，会严重影响 API 接口的性能。

这时候，我们自然就会想到利用 CPU 的多核能力，并发来处理这 100 条记录。这种场景我们在实际开发中经常遇到，有必要抽象成一个并发处理模板，这样以后在查询时，就可以使用这个模板了。

例如，iam-apiserver 中，查询用户列表接口 [List](#)，还需要返回每个用户所拥有的策略个数。这就用到了并发处理。这里，我试着将其抽象成一个模板，模板如下：

[复制代码](#)

```
1 func (u *userService) List(ctx context.Context, opts metav1.ListOptions) (*v1.
2   users, err := u.store.Users().List(ctx, opts)
3   if err != nil {
4       log.L(ctx).Errorf("list users from storage failed: %s", err.Error())
5
6       return nil, errors.WithCode(code.ErrDatabase, err.Error())
7   }
8
9   wg := sync.WaitGroup{}
10  errChan := make(chan error, 1)
11  finished := make(chan bool, 1)
12
13  var m sync.Map
14
15  // Improve query efficiency in parallel
16  for _, user := range users.Items {
17      wg.Add(1)
18
19      go func(user *v1.User) {
20          defer wg.Done()
21
22          // some cost time process
23          policies, err := u.store.Policies().List(ctx, user.Name, metav1.ListOpti
24          if err != nil {
25              errChan <- errors.WithCode(code.ErrDatabase, err.Error())
26
27              return
28          }
29
30          m.Store(user.ID, &v1.User{
31              ...
32              Phone:      user.Phone,
33              TotalPolicy: policies.TotalCount,
34          })
35      }(user)
36  }
37
38  go func() {
39      wg.Wait()
40      close(finished)
```

```
41  }()  
42  
43  select {  
44  case <-finished:  
45  case err := <-errChan:  
46      return nil, err  
47  }  
48  
49  // infos := make([]*v1.User, 0)  
50  infos := make([]*v1.User, 0, len(users.Items))  
51  for _, user := range users.Items {  
52      info, _ := m.Load(user.ID)  
53      infos = append(infos, info.(*v1.User))  
54  }  
55  
56  log.L(ctx).Debugf("get %d users from backend storage.", len(infos))  
57  
58  return &v1.UserList{ListMeta: users.ListMeta, Items: infos}, nil  
59 }
```

在上面的并发模板中，我实现了并发处理查询结果中的三个功能：

**第一个功能，goroutine 报错即返回。**goroutine 中代码段报错时，会将错误信息写入 errChan 中。我们通过 List 函数中的 select 语句，实现只要有一个 goroutine 发生错误，即返回：

 复制代码

```
1 select {  
2 case <-finished:  
3 case err := <-errChan:  
4     return nil, err  
5 }
```

**第二个功能，保持查询顺序。**我们从数据库查询出的列表是有顺序的，比如默认按数据库 ID 字段升序排列，或者我们指定的其他排序方法。在并发处理中，这些顺序会被打断。但为了确保最终返回的结果跟我们预期的排序效果一样，在并发模板中，我们还需要保证最终返回结果跟查询结果保持一致的排序。

上面的模板中，我们将处理后的记录保存在 map 中，map 的 key 为数据库 ID。并且，在最后按照查询的 ID 顺序，依次从 map 中取出 ID 的记录，例如：

 复制代码

```
1     var m sync.Map
2     for _, user := range users.Items {
3         ...
4         go func(user *v1.User) {
5             ...
6             m.Store(user.ID, &v1.User{})
7         }(user)
8     }
9     ...
10    infos := make([]*v1.User, 0, len(users.Items))
11    for _, user := range users.Items {
12        info, _ := m.Load(user.ID)
13        infos = append(infos, info.(*v1.User))
14    }
```

通过上面这种方式，可以确保最终返回的结果跟从数据库中查询的结果保持一致的排序。

**第三个功能，并发安全。** Go 语言中的 map 不是并发安全的，要想实现并发安全，需要自己实现（如加锁），或者使用 sync.Map。上面的模板使用了 sync.Map。

当然了，如果期望 List 接口能在期望时间内返回，还可以添加超时机制，例如：

 复制代码

```
1     select {
2     case <-finished:
3     case err := <-errChan:
4         return nil, err
5     case <-time.After(time.Duration(30 * time.Second)):
6         return nil, fmt.Errorf("list users timeout after 30 seconds")
7     }
8 }
```

goroutine 虽然很轻量，但还是会消耗资源，如果我们需要处理几百上千的并发，就需要用协程池来复用协程，达到节省资源的目的。有很多优秀的协程包可供我们直接使用，比如 [🔗 ants](#)、[🔗 tunny](#) 等。

## 其他特性

除了上面那两大类，这里我还想给你介绍下关键代码设计中的其他特性，包括插件化选择 JSON 库、调用链实现、数据一致性。

## 插件化选择 JSON 库

Golang 提供的标准 JSON 解析库 `encoding/json`，在开发高性能、高并发的网络服务时会产生性能问题。所以很多开发者在实际的开发中，往往会选用第三方的高性能 JSON 解析库，例如 [jsoniter](#)、[easyjson](#)、[jsonparser](#) 等。

我见过的很多开发者选择了 `jsoniter`，也有一些开发者使用了 `easyjson`。`jsoniter` 的性能略高于 `encoding/json`。但随着 go 版本的迭代，`encoding/json` 库的性能也越来越高，`jsoniter` 的性能优势也越来越有限。所以，IAM 项目使用了 `jsoniter` 库，并准备随时切回 `encoding/json` 库。

为了方便切换不同的 JSON 包，`iam-apiserver` 采用了一种插件化的机制来使用不同的 JSON 包。具体是通过使用 go 的标签编译选择运行的解析库来实现的。

标签编译就是在源代码里添加标注，通常称之为编译标签（build tag）。编译标签通过注释的方式在靠近源代码文件顶部的地方添加。go build 在构建一个包的时候，会读取这个包里的每个源文件并且分析编译便签，这些标签决定了这个源文件是否参与本次编译。例如：

 复制代码

```
1 // +build jsoniter
2
3 package json
4
5 import jsoniter "github.com/json-iterator/go"
```

`+build jsoniter`就是编译标签。这里要注意，一个源文件可以有多个编译标签，多个编译标签之间是逻辑“与”的关系；一个编译标签可以包括由空格分割的多个标签，这些标签是逻辑“或”的关系。例如：

 复制代码

```
1 // +build linux darwin
2 // +build 386
```

这里要注意，编译标签和包的声明之间应该使用空行隔开，否则编译标签会被当作包声明的注释，而不是编译标签。

那具体来说，我们是如何实现插件化选择 JSON 库的呢？

首先，我自定义了一个 [github.com/marmotedu/component-base/pkg/json](https://github.com/marmotedu/component-base/pkg/json) json 包，来适配 encoding/json 和 json-iterator。github.com/marmotedu/component-base/pkg/json 包中有两个文件：

**json.go**：映射了 encoding/json 包的 Marshal、Unmarshal、MarshalIndent、NewDecoder、NewEncoder 方法。


**jsoniter.go**：映射了 github.com/json-iterator/go 包的 Marshal、Unmarshal、MarshalIndent、NewDecoder、NewEncoder。

json.go 和 jsoniter.go 通过编译标签，让 Go 编译器在构建代码时选择使用哪一个 json 文件。

接着，通过在执行 go build 时指定 `-tags` 参数，来选择编译哪个 json 文件。

json/json.go、json/jsoniter.go 这两个 Go 文件的顶部，都有一行注释：

```
1 // +build !jsoniter
2
3 // +build jsoniter
```

 复制代码

`// +build !jsoniter`表示，tags 不是 jsoniter 的时候编译这个 Go 文件。`// +build jsoniter`表示，tags 是 jsoniter 的时候编译这个 Go 文件。也就是说，这两种条件是互斥的，只有当 tags=jsoniter 的时候，才会使用 json-iterator，其他情况使用 encoding/json。

例如，如果我们想使用包，可以这么编译项目：



```
1 $ go build -tags=jsoniter
```

在实际开发中，**我们需要根据场景来选择合适的 JSON 库**。这里我给你一些建议。

### 场景一：结构体序列化和反序列化场景

在这个场景中，我个人首推的是官方的 JSON 库。可能你会比较意外，那我就来说说我的理由：

首先，虽然 easyjson 的性能压倒了其他所有开源项目，但它有一个最大的缺陷，那就是需要额外使用工具来生成这段代码，而对额外工具的版本控制就增加了运维成本。当然，如果你的团队已经能够很好地处理 protobuf 了，也是可以用同样的思路来管理 easyjson 的。

其次，虽然 Go 1.8 之前，官方 JSON 库的性能总是被大家吐槽，但现在（1.16.3）官方 JSON 库的性能已不可同日而语。此外，作为使用最为广泛，而且没有之一的 JSON 库，官方库的 bug 是最少的，兼容性也是最好的

最后，jsoniter 的性能虽然依然优于官方，但没有达到逆天的程度。如果你追求的是极致的性能，那么你应该选择 easyjson 而不是 jsoniter。jsoniter 近年已经不活跃了，比如说，我前段时间提了一个 issue 没人回复，于是就上去看了下 issue 列表，发现居然还遗留着一些 2018 年的 issue。

### 场景二：非结构化数据的序列化和反序列化场景

这个场景下，我们要分高数据利用率和低数据利用率两种情况来看。你可能对数据利用率的高低没啥概念，那我举个例子：JSON 数据的正文中，如果说超过四分之一的数据都是业务需要关注和处理的，那就算是高数据利用率。

在高数据利用率的情况下，我推荐使用 jsonvalue。

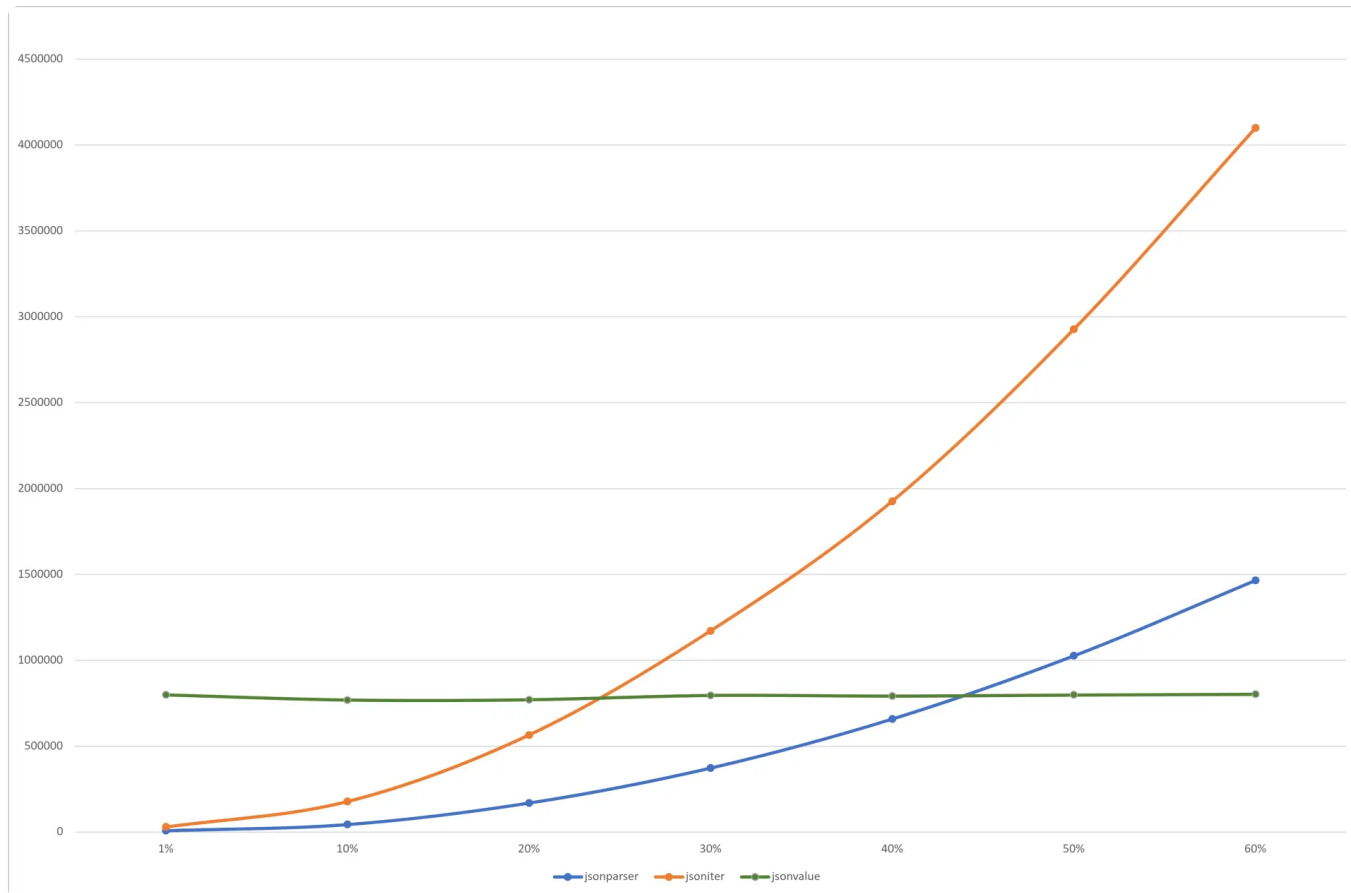
至于低数据利用率的情况，还可以根据 **JSON 数据是否需要重新序列化**，分成两种情况。

如果无需重新序列化，这个时候选择 jsonparser 就行了，因为它的性能实在是耀眼。

如果需要重新序列化，这种情况下你有两种选择：如果对性能要求相对较低，可以使用 jsonvalue；如果对性能的要求高，并且只需要往二进制序列中插入一条数据，那么可以采用 jsoniter 的 Set 方法。

实际操作中，超大 JSON 数据量，并且同时需要重新序列化的情况非常少，往往是在代理服务器、网关、overlay 中继服务等，同时又需要往原数据中注入额外信息的时候。换句话说，jsoniter 的适用场景比较有限。

下面是从 10% 到 60% 数据覆盖率下，不同库的操作效率对比（纵坐标单位： $\mu\text{s/op}$ ）：



可以看到，当 jsoniter 的数据利用率达到 25% 时，和 jsonvalue、jsonparser 相比就已经没有任何优势；至于 jsonvalue，由于对数据做了一次性的全解析，因此解析后的数据存取耗时极少，因此在不同数据覆盖率下的耗时都很稳定。

## 调用链实现

调用链对查日志、排障帮助非常大。所以，在 iam-apiserver 中也实现了调用链，通过 requestID 来串联整个调用链。

具体是通过以下两步来实现的：

第一步，将 `ctx context.Context` 类型的变量作为函数的第一个参数，在函数调用时传递。

第二步，不同函数中，通过 `log.L(ctx context.Context)` 来记录日志。

在请求到来时，请求会通过 [Context](#) 中间件处理：

[复制代码](#)

```
1 func Context() gin.HandlerFunc {
2     return func(c *gin.Context) {
3         c.Set(log.KeyRequestID, c.GetString(XRequestIDKey))
4         c.Set(log.KeyUsername, c.GetString(UsernameKey))
5         c.Next()
6     }
7 }
```

在 Context 中间件中，会在 `gin.Context` 类型的变量中设置 `log.KeyRequestID` 键，其值为 36 位的 UUID。UUID 通过 [RequestID](#) 中间件来生成，并设置在 gin 请求的 Context 中。

RequestID 中间件在 Context 中间件之前被加载，所以在 Context 中间件被执行时，能够获取到 RequestID 生成的 UUID。


`log.L(ctx context.Context)` 函数在记录日志时，会从头 `ctx` 中获取到 `log.KeyRequestID`，并作为一个附加字段随日志打印。

通过以上方式，我们最终可以形成 iam-apiserver 的请求调用链，日志示例如下：

[复制代码](#)


```
1 2021-07-19 19:41:33.472 INFO    apiserver    apiserver/auth.go:205    user
2 2021-07-19 19:41:33.472 INFO    apiserver    policy/create.go:22     create
```

另外，ctx context.Context作为函数 / 方法的第一个参数，还有一个好处是方便后期扩展。例如，如果我们有以下调用关系：

 复制代码


```
1 package main
2
3 import "fmt"
4
5 func B(name, address string) string {
6     return fmt.Sprintf("name: %s, address: %s", name, address)
7 }
8
9 func A() string {
10     return B("colin", "sz")
11 }
12
13 func main() {
14     fmt.Println(A())
15 }
```

上面的代码最终调用 B函数打印出用户名及其地址。如果随着业务的发展，希望 A 调用 B 时，传入用户的电话，B 中打印出用户的电话号码。这时候，我们可能会考虑给 B 函数增加一个电话号参数，例如：

 复制代码

```
1 func B(name, address, phone string) string {
2     return fmt.Sprintf("name: %s, address: %s, phone: %s", name, address)
3 }
```

如果我们后面还要增加年龄、性别等属性呢？按这种方式不断增加 B 函数的参数，不仅麻烦，而且还要改动所有调用 B 的函数，工作量也很大。这时候，可以考虑通过 ctx context.Context 来传递这些扩展参数，实现如下：

 复制代码

```
1 package main
2
3 import (
4     "context"
```

```
5     "fmt"
6 )
7
8 func B(ctx context.Context, name, address string) string {
9     return fmt.Sprintf("name: %s, address: %s, phone: %v", name, address, ctx.
10 }
11
12 func A() string {
13     ctx := context.WithValue(context.TODO(), "phone", "1812884xxxx")
14     return B(ctx, "colin", "sz")
15 }
16
17 func main() {
18     fmt.Println(A())
19 }
```

这样，我们下次需要新增参数的话，只需要调用 context 的 WithValue 方法：

```
1 ctx = context.WithValue(ctx, "sex", "male")
```

[复制代码](#)

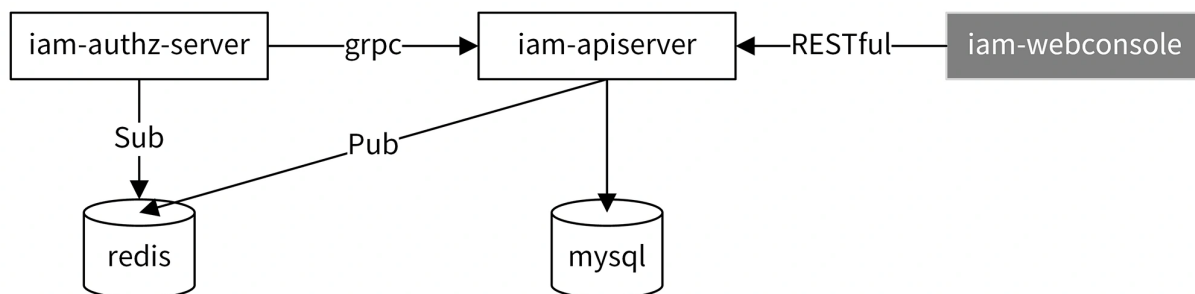
在 B 函数中，通过 context.Context 类型的变量提供的 Value 方法，从 context 中获取 sex key 即可：

```
1 return fmt.Sprintf("name: %s, address: %s, phone: %v, sex: %v", name, address,
```

[复制代码](#)

## 数据一致性

为了提高 iam-authz-server 的响应性能，我将密钥和授权策略信息缓存在 iam-authz-server 部署机器的内存中。同时，为了实现高可用，我们需要保证 iam-authz-server 启动的实例个数至少为两个。这时候，我们会面临数据一致性的问题：所有 iam-authz-server 缓存的数据要一致，并且跟 iam-apiserver 数据库中保存的一致。iam-apiserver 通过如下方式来实现数据一致性：



具体流程如下：

第一步，iam-authz-server 启动时，会通过 grpc 调用 iam-apiserver 的 GetSecrets 和 GetPolicies 接口，获取所有的密钥和授权策略信息。

第二步，当我们通过控制台调用 iam-apiserver 密钥 / 授权策略的写接口（POST、PUT、DELETE）时，会向 Redis 的 iam.cluster.notifications 通道发送 SecretChanged/PolicyChanged 消息。

第三步，iam-authz-server 会订阅 iam.cluster.notifications 通道，当监听到有 SecretChanged/PolicyChanged 消息时，会请求 iam-apiserver 拉取所有的密钥 / 授权策略。

通过 Redis 的 Sub/Pub 机制，保证每个 iam-authz-server 节点的缓存数据跟 iam-apiserver 数据库中保存的数据一致。所有节点都调用 iam-apiserver 的同一个接口来拉取数据，通过这种方式保证所有 iam-authz-server 节点的数据是一致的。

## 总结

今天，我和你分享了 iam-apiserver 的一些关键功能实现，并介绍了我的设计思路。这里我再简要梳理下。

为了保证进程关停时，HTTP 请求执行完后再断开连接，进程中的任务正常完成，iam-apiserver 实现了优雅关停功能。

为了避免进程存在，但服务没成功启动的异常场景，iam-apiserver 实现了健康检查机制。

Gin 中间件可通过配置文件配置，从而实现按需加载的特性。

为了能够直接辨别出 API 的版本，iam-apiserver 将 API 的版本标识放在 URL 路径中，例如 /v1/secrets。

为了能够最大化地共享功能代码，iam-apiserver 抽象出了统一的元数据，每个 REST 资源都具有这些元数据。

因为 API 接口都是通过同一个函数来返回的，其返回格式自然是统一的。

因为程序中经常需要处理并发逻辑，iam-apiserver 抽象出了一个通用的并发模板。

为了方便根据需要切换 JSON 库，我们实现了插件化选择 JSON 库的功能。

为了实现调用链功能，iam-apiserver 不同函数之间通过 `ctx context.Context` 来传递 RequestID。

iam-apiserver 通过 Redis 的 Sub/Pub 机制来保证数据一致性。

## 课后练习

1. 思考一下，在你的项目开发中，使用过哪些更好的并发处理方式，欢迎你在留言区分享。
2. 试着给 iam-apiserver 增加一个新的、可配置的 Gin 中间件，用来实现 API 限流的效果。

欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta订阅后你可得 **24** 元现金奖励

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [28 | 控制流（上）：通过iam-apiserver设计，看Web服务的构建](#)



更多课程推荐

# 说透区块链

拨开迷雾，还原区块链真相

赵铭

区块链服务平台资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言 (1)

 写留言



nio

2021-07-31

很不错，有启发，边学边用！

展开 ▾



👍 1