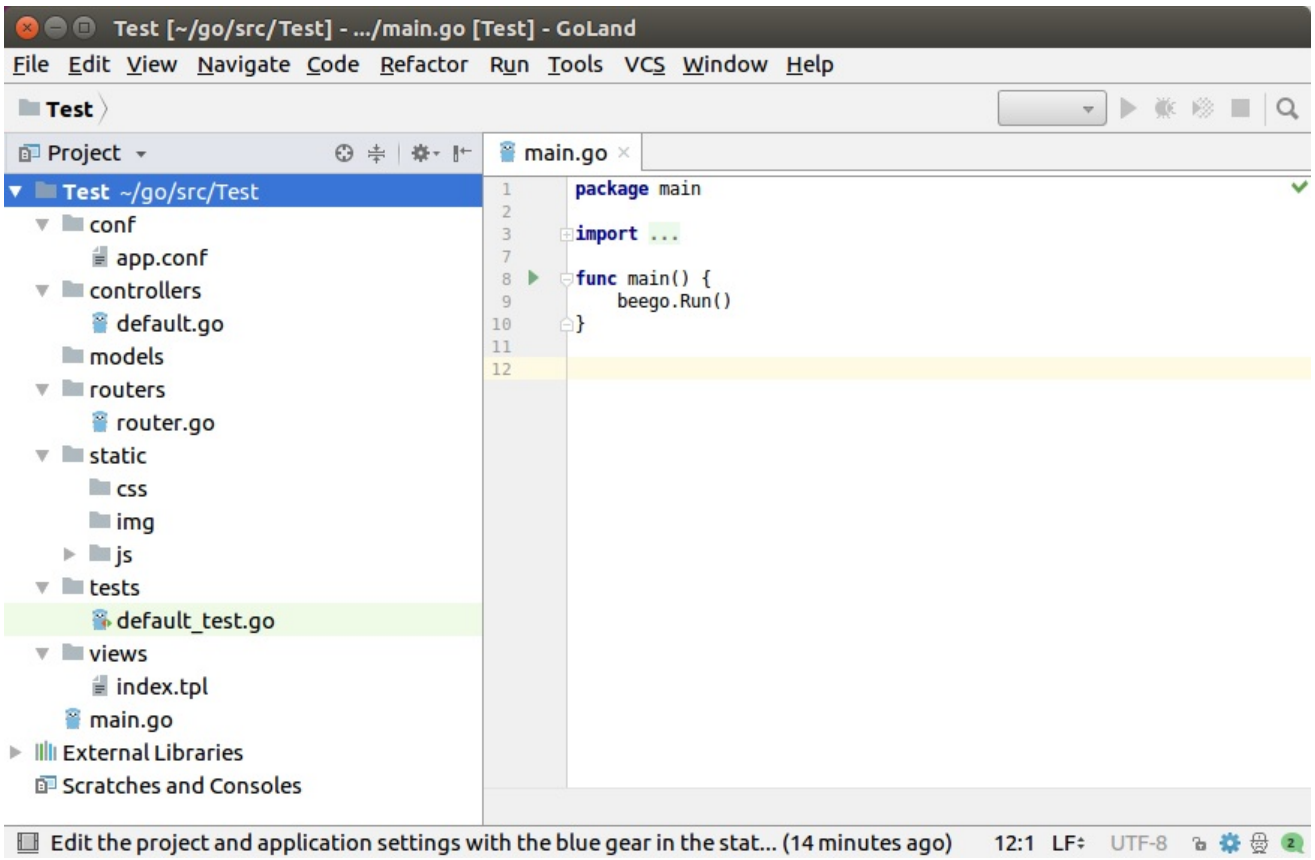
项目开发

创建一个Go Web应用程序

为了快速的创建一个示例的应用程序，我们使用beego 的bee工具。命令如下：

```
bee new Test
```



依赖管理

使用govendor进行项目依赖管理，该工具将项目依赖的外部包拷贝到项目下的 vendor 目录下，并通过 vendor.json 文件来记录依赖包的版本，方便用户使用相对稳定的依赖。 对于 govendor 来说，依赖包主要有以下多种类型:

状态	缩写状态	含义
+local	l	本地包，即项目自身的包组织
+external	e	外部包，即被 $GOPATH 管理，但不在 vendor 目录下
+vendor	v	已被 govendor 管理，即在 vendor 目录下
+std	s	标准库中的包
+unused	u	未使用的包，即包在 vendor 目录下，但项目并没有用到
+missing	m	代码引用了依赖包，但该包并没有找到
+program	p	主程序包，意味着可以编译为执行文件
+outside		外部包和缺失的包
+all		所有的包

- 安装

```
go get -u github.com/kardianos/govendor
```

命令行执行 `govendor`，查看安装结果。

> 注意：
>
> 需要把 `$GOPATH/bin/` 加到 `PATH` 中。

- 子命令

```
init     创建 vendor 文件夹和 vendor.json 文件

add      从 $GOPATH 中添加依赖包，会加到 vendor.json
remove   从 vendor 文件夹删除依赖
update   从 $GOPATH 升级依赖包
list     列出已经存在的依赖包

fetch    从远端库增加新的，或者更新 vendor 文件中的依赖包
sync     本地存在 vendor.json 时候拉去依赖包，匹配所记录的版本

status   列出本地丢失的、过期的和修改的package
get      类似 go get，但是会把依赖包拷贝到 vendor 目录
```

- 测试
  - Fabric-SDK-Go

```
go get -u github.com/hyperledger/fabric-sdk-go
```

- 初始化

```
govendor init
```

- 添加依赖包到项目

```
govendor add +external
```

> 注意：
> 由于项目中没有使用fabric-sdk-go，所以引入进来的只有项目中用到的第三方依赖

- 添加fabric-sdk-go依赖

```
_ "github.com/hyperledger/fabric-sdk-go"
```

- 再次执行添加依赖包到项目

- vendor.json作用

项目在传输时可以仅传递该文件，使用`govendor sync`命令可以下载依赖信息

项目代码

- 项目启动入口

```
// main.go 文件
// import中引入routers
func main() {
    beego.Run()
}
```

- 项目访问路由

```
// routers目录下的routers.go文件
// import中引入controller
func init() {
    ……
    beego.Router("/blockchain/:bank/userdetails",
&transactioncontroller.TransactionController{}, "post:FeedAccountInfo")
    beego.Router("/blockchain/createaccount", &transactioncontroller.TransactionController{},
"post:CreateType2Account")
}
```

- 控制器处理

```
// 在controllers的transaction文件夹中，transaction.go
func (tc *TransactionController) FeedAccountInfo() {
    beego.Debug("upload account info")
```

```
    ……
    // css为models.ChainCodeSpec
    ccs, stdError = models.Initialize(channelID, bankName, chainCodeID, userId)
    ……
    _, err = ccs.ChaincodeInvoke("creditAccountInfo", chaincodeArgs)
}
func (tc *TransactionController) CreateType2Account() {
    beego.Debug("create type2 account")
    ……
    // css为models.ChainCodeSpec
    ccs, stdError = models.Initialize(channelID, createAccount.T2AcntInfo.BankName,
chainCodeName, userId)
    ……
    accountHashReturned, stdError := ccs.ChaincodeQuery("authAccount", chaincodeArgs)
}
```

- 模型处理

在model中会引入fabric-sdk-go，完成与区块链服务的交互

```
// models文件夹中channel.go文件
package models

import (
    ……
    "github.com/hyperledger/fabric-sdk-go/api/apitxn"
    "github.com/hyperledger/fabric-sdk-go/def/fabapi"
)

type ChainCodeSpec struct {
    channelClient apitxn.ChannelClient
    userID string
    chainCodeID string
}

// Initialize reads the configuration file and sets up the client, chain and event hub
func Initialize(channelID, bank, chainCodeId, userId string)  (*ChainCodeSpec,error) {
    var configFile string
    if bank == "xxx1" {
        configFile =  beego.AppConfig.String("CORE_XXX1_CONFIG_FILE")
    } ……

    chnlClient, err := getChannelClient(channelID, configFile, userId)
    ……
    return &ChainCodeSpec{channelClient: chnlClient, chainCodeID: chainCodeId, userID: userId},
nil
}

func getChannelClient(channelID, configFile, userId string) (apitxn.ChannelClient, error) {

    var chClient apitxn.ChannelClient
    // Create SDK setup for the integration tests
    sdkOptions := fabapi.Options{
```

```
        ConfigFile: configFile,
    }

    sdk, err := fabapi.NewSDK(sdkOptions)
    ……

    chClient, err = sdk.NewChannelClient(channelID, userId)
    ……

    return chClient, err
}

//cf *peerChaincode.ChaincodeCmdFactory
func (ccs *ChainCodeSpec) ChaincodeInvoke(fcn string,chaincodeArgs [][]byte) (responsePayload
[]byte, err error) {
    _, err = ccs.channelClient.ExecuteTx(apitxn.ExecuteTxRequest{ChaincodeID: ccs.chainCodeID,
Fcn: fcn, Args: chaincodeArgs})
    ……
    return nil, nil
}

func (ccs *ChainCodeSpec) ChaincodeQuery(fcn string,chaincodeArgs [][]byte) (responsePayload
[]byte, err error) {

    value, err := ccs.channelClient.Query(apitxn.QueryRequest{ChaincodeID: ccs.chainCodeID, Fcn:
fcn, Args: chaincodeArgs})
    ……
    return value, err
}
……
```

Yaml

使用Docker部署Go Web应用程序

目标

- 了解Docker如何帮您开发Go和部署应用程序
- 知道如何为Go应用程序创建团队统一的Docker容器

创建Dockerfile

最终生成的Dockerfile内容如下：

```
# 使用Go的官方映像作为基础映像。这个映像是Go 1.6预安装的。该映像的$GOPATH值已被设置为/go。所有安装在/go/src的程序包都能通过go命令访问。
FROM golang:1.6

# 安装beego程序包和bee工具。beego程序包将在应用程序内部使用，bee工具将用于在开发过程中实时重载代码。
RUN go get github.com/astaxie/beego && go get github.com/beego/bee

# 通过开发计算机上容器的8080端口暴露该应用程序。最后一行，
EXPOSE 8080

# 使用bee命令开始对我们的应用程序进行实时重载。
CMD ["bee", "run"]
```

构建镜像

创建好Docker文件之后，可运行下列命令创建映像：

```
docker build -t go-web-demo .
```

执行上述命令可创建一个名为go-web-demo的镜像。我们可以把这个镜像发布到镜像仓库供开发团队中所有的成员来使用，这样就可以保证开发环境的一致性。

要查看您系统中的映像列表，请运行下列命令：

```
docker images
```

运行容器

准备好go-web-demo之后，可以使用下列命令启动一个容器：

```
docker run -it --name my-go-web-demo -p 8080:8080  -v /app/go/src/go-web-demo:/go/src/go-web-demo -w /go/src/go-web-demo go-web-demo
```

接下来，我们对执行的命令做一些简单的解释：

- docker run命令可用于通过镜像运行容器
- -it标记使用交互式模式启动该容器
- --name my-go-web-demo 将容器命名为my-go-web-demo
- -p 8080:8080将容器8080端口映射到主机8080端口上，最终我们可以通过主机的8080端口访问容器里的内容
- -v /app/go/src/go-web-demo:/go/src/go-web-demo，使用volume将/app/go/src/go-web-demo从计算机映射至容器的/go/src/go-web-demo目录
- -w /go/src/go-web-demo 设置容器的工作目录
- go-web-demo 指定了容器使用的镜像名称

当容器启动以后，我们可以通过访问http://localhost:8080来验证容器是否运行正常。