后端基础知识 · 讲稿

前言

大家好，我是今天后端部分的主讲人。在接下来的二十多分钟里，我会带大家快速过一遍后端开发的核心脉络——从 API 路由到容器化部署。

来看看今天的目录：我们一共六个模块。API 路由、中间件、数据模型设计、缓存策略、开发规范与 CI/CD、容器化部署。

好，我们直接开始。

模块一：API 路由

前后端交互模型

先建立一个最基础的直觉：前后端到底是怎么交互的？

一句话：客户端发起 HTTP 请求，后端处理完业务逻辑，返回数据——通常是 JSON 格式。

你点一下浏览器里的按钮，或者 App 刷新页面，背后发生的事情大致是这样：请求从浏览器出发，向后端服务器发送HTTP或者HTTPS request；服务器收到请求后，第一件事就是**路由匹配**——服务器根据 URL 和方法判断这个请求该分发给哪个函数(我们称之为handler)处理业务逻辑。处理完，将处理结果以HTTP/HTTPS response的格式返回前端。

HTTP 语义

那路由是怎么匹配的呢？核心就两样东西：HTTP Method 和 URL。这两个都是HTTP协议在请求中定义的字段。

左边是 Method：GET 查询、POST 创建、PUT 全量更新、PATCH 部分更新、DELETE 删除。这个表大家以后会天天看，先混个脸熟就行。

而状态码则是HTTP协议回复中定义的字段

右边是状态码：2 开头成功、3 开头重定向、4 开头客户端的问题、5 开头服务端的问题。比如 404 就是你要的资源不存在，500 就是服务端自己崩了。

简单记：Method 表达你想干什么，状态码表达这件事干没干成。各司其职。

HTTP 请求参数

HTTP请求不仅有方法和URL字段，也有各种参数字段、请求体等。

三种基本形态：第一，路径参数—— `/users/123` ，用 URL 里的变量定位具体资源。第二，查询参数—— `/users?page=1&size=20` ，问号后面带的键值对，用来过滤、排序、分页。第三，请求体——POST、PUT、PATCH 的时候，大段 JSON 数据放在 body 里传。

右边是一个完整的 POST 请求示例：向 `/api/v1/orders` 发一个创建订单的请求，body 里带了用户 ID 和商品列表。

> 这一部分内容可参考RFC文档或直接询问大模型 https://www.rfc-editor.org/info/rfc2616

RESTful 设计原则

有了这些内容，我们就可以来设计服务器处理各种https请求的API接口了

- 第一个原则：**资源导向**。URL 里放的是"东西"——用户、订单、商品——而不是动作(quiz: URL的英文全称是什么？)。动词用 HTTP Method 来表达。
  看这几个例子： `GET /users/123` 查用户，对的； `POST /createUser` 创建用户，错的——`POST /users` 就够了。 `DELETE /users/123` 删用户，对的； `GET /getUser?uid=123` ——URL 里写动词是不规范的。
- 第二个原则：**无状态**。每个请求必须携带服务端需要的全部信息，服务端不保存"上次跟这个客户端聊到哪了"。
  这样做有什么好处？服务器可以水平扩展。请求打到哪台机器，哪台就能处理。代价就是状态得自己想办法管理——要么存在客户端，要么数据库之类的外部存储。

路由分组与版本管理

最后说一下版本管理。为什么要版本？你用手机上 App，老版本还在跑，服务端的 API 不能随便改——改了老客户端就挂了。所以用 `/api/v1` 、 `/api/v2` 来区分版本，新老并存。

路由分组就是把同一资源的路由放在一起管理。比如用 Gin 框架，把 `/api/v1` 下面所有用户相关的路由——增删改查——全部归到一个 Group 里，代码清晰，好维护。

模块二：中间件

洋葱模型

到一个新模块：中间件。

中间件是什么？就是一个函数，在请求真正到达 handler 之前和之后执行。注意"之前"和"之后"这两个词都很重要。

看这个时序图：请求进来，先经过安全中间件，再经过日志中间件，再经过鉴权中间件，最后到达 handler。处理完，响应沿着原路返回——鉴权、日志、安全，一层一层往外走。

这就叫洋葱模型：进去的时候处理请求，出来的时候处理响应。每一层是一个洋葱皮。

JWT 认证鉴权

中间件最典型的应用就是鉴权。我们来聊 JWT。

流程是这样的：用户登录，服务端验证用户名密码正确之后，签发一个 JWT 令牌。这个令牌里包含了用户信息和一个防篡改的签名。客户端拿到 JWT 后，以后每次请求都在 header 里带上。服务端的中间件看到这个 JWT，验证签名，通过之后把用户信息注入到上下文里，后面的 handler 直接就能拿到"当前是谁在请求"。

JWT 的结构很简单：Header、Payload、Signature，三段用点分隔，每段 Base64 编码。Header 说这是什么算法、什么类型；Payload 放用户信息和过期时间；Signature 是防篡改的签名——有人改了 Payload，签名就对不上了。

JWT 最大的优势是**无状态**。服务端不用存 session，验证完就可以忘掉——这对**Restful API设计非常友好**。

实例：鉴权中间件

看一眼代码。左边是一个 Auth 中间件的 Go 实现。

它接收一个 `next http.Handler`，返回一个新的 Handler。在新 Handler 里，从请求头取 `Authorization`，如果为空就返回 401 并且 `return`——注意这个 return，意味着**不再调用 `next.ServeHTTP`**，请求在这里就被拦截了。如果 token 不为空，才调用 `next.ServeHTTP(w, r)`，继续往下一层走。

右边展示了怎么用：一行嵌套 `Auth(HelloHandler)` 就搭好了洋葱。这个嵌套就是洋葱模型在代码里的体现。

执行路径也很直观：Auth 进入，验证 token。失败就直接 401，后面的 Handler 碰都不碰。通过就执行 Handler 写响应。最后 Auth 退出。

这就是责任链模式的核心思想：**每个中间件只做一件事，通过嵌套组合起来，配合路由分组，实现高效代码复用与请求管线管理**。（PS：可以对整个路由分组套中间件）

模块三：数据模型设计

场景引入

下一个模块：数据模型设计。

为什么要讲这个？代码写得再漂亮，表设计一塌糊涂，上线之后性能差、改不动、全是 Bug。数据库设计不是 DBA 的事，是每个后端都要会的。

怎么做？三步法：

- 第一，识别实体——系统里有哪些东西？
- 第二，梳理关系——这些东西之间怎么关联？
- 第三，建表——落实到数据库。

设计实操：简易商城

拿一个简易商城来练手。

识别实体：四个东西。User，有 ID、名字、邮箱、创建时间。Product，有 ID、名字、价格、库存。Order，有 ID、关联的用户 ID、总金额、状态、创建时间。OrderItem，有 ID、关联的订单 ID、关联的商品 ID、数量和单价。

梳理关系：User 和 Order 是一对多——一个用户可以下很多订单，所以在 Order 表里加 `user_id` 外键。Order 和 OrderItem 是一对多——一个订单包含多个商品项，所以在 OrderItem 表里加 `order_id` 外键。Product 和 OrderItem 是一对多——一个商品可以被多个订单购买，所以在 OrderItem 表里加 `product_id` 外键。

四张表，三条外键关系，一个简易商城的骨架就有了。

设计要点总结

设计完表结构，还有两个要点要记住。

第一，范式到 3NF 就够了。1NF 就是列不能再拆分，比如不要把"地址"存成一个字段，拆成省市区。2NF 就是非主键列必须完全依赖主键。3NF 就是非主键列之间不能有依赖。日常开发做到 3NF，读多写少的场景可以适当冗余一些字段，避免老是 JOIN。

第二，索引。什么列该建索引？WHERE 条件里常出现的列、外键列、排序和分组的列。但要记住，索引不是越多越好——它会拖慢写入，占用额外存储。建索引之前想清楚：查询真的慢了吗？这个列真的常用来查吗？

> 其实一般不太用考虑索引，普通开发直接用mysql等开源数据库即可。但**查表性能是有必要仔细考虑**的——直接查主键和查字符串匹配(甚至逐条解码json字段)的开销肯定是不同的。

模块四：缓存策略

缓存目标与 Redis 概览

缓存是干什么的？两个字：快和省。快，就是降低延迟——内存读写比磁盘快几个数量级。省，就是减轻数据库压力——热点数据不要每次都去查库。

Redis 是目前最主流的内存缓存。它不只是个 key-value 存储，它提供了五种核心数据结构。

String 最简单：缓存单个值、做分布式锁、做计数器。Hash 用来存对象，比多个 String 更省内存。List 可以做消息队列、存最新列表。Set 做去重、查共同好友。ZSet 有序集合，做排行榜、延迟队列。

这五种结构覆盖了后端开发里绝大多数的数据场景。你不需要一次全记住，但要知道：每种结构对应的典型场景是什么——将来遇到需求时，至少知道自己该往哪个方向查。

模块五：开发规范与 CI/CD

Git 与远程仓库

开发规范这一块，先讲 Git。

为什么每个项目都要用 Git？三点：回溯历史——改坏了可以回滚；多人协作——不会互相覆盖代码；代码审查——通过 Pull Request 让别人检查你的代码再合并。

基本工作流：clone 仓库、修改代码、commit 提交、push 推送、发起 PR。团队成员 review 通过后，merge 到主分支。就这六步。

平台推荐 GitHub 和 GitLab，都是行业标准。

项目结构与 CI/CD

- 左边是一个典型的 Go 项目结构。 `cmd` 放入口， `internal` 下按职责分：handler 处理路由，middleware 实现中间件，service 写业务逻辑，repository 访问数据库，model 定义数据模型。 `config` 独立出来管配置。
  有个重要原则：**密码、密钥这些敏感信息决不能进代码仓库，用环境变量管理并注入到镜像构建过程，镜像内也不以文件形式保存这些环境变量**。
- 右边是 CI/CD 流水线：代码推送触发，然后自动 Lint 检查代码风格，跑测试，构建产物打 Docker 镜像，推送镜像到仓库，最后自动部署上线。
  这一整套流程的核心思想是：代码提交后，**机器自动帮你检查、测试、打包、上线**。出了问题是在开发阶段就被机器拦住的，不是等到生产环境才被用户发现的。

模块六：容器化部署

Docker 核心理念

最后一个技术模块：容器化部署。Docker 的 slogan 就是"一次构建，到处运行"。

> 基础镜像，是由一些企业或者开源开发者发布的；我们可以在基础镜像上通过Dockfile构建自己的镜像，譬如安装新的软件、把自己的代码拷贝进镜像；
> 当然到此为止镜像还只是文件数据，我们将其启动后，就形成了活动的容器。

三个核心特点。

- 第一，共享宿主机内核。容器本质是宿主机上的进程，没有自己的操作系统。跟虚拟机的区别很重要：硬边界——内核版本、系统调用、GPU 驱动这些东西必须跟宿主机一致，你改不了。但用户态的所有东西——glibc、Python 运行时、软件包——随便你怎么配。因为共享内核，容器启动是毫秒级的，资源开销比虚拟机小得多。
- 第二，镜像分层构建。每条 Dockerfile 指令生成一个只读层，一层一层摞起来。容器层在最上面，是可写的。一个 Dockerfile 把整个运行环境固化了——再也不会"我电脑上能跑，服务器上不行"。
- 第三，写时复制。容器要修改镜像层的文件时，先把文件复制到自己的容器层再改，原始的镜像层不动。这样多个容器共享同一份基础镜像，却互不干扰。

> 这部分的原理确实不太好特别简单地表述。总之理解Docker提供了能够**自由配置并固定运行环境**的方法

Dockerfile 示例

```
# 仅供理解，实际编写请咨询LLM
# 第一阶段：构建
FROM golang:1.21-alpine AS builder
WORKDIR /app
COPY . .
RUN go build -o server ./cmd/main.go

# 第二阶段：运行
FROM alpine:3.19
COPY --from=builder /app/server .
EXPOSE 8080
CMD ["./server"]
```

看一个多阶段构建的 Dockerfile。第一阶段用 Go 的编译镜像，把源码编译成二进制。第二阶段切换到 Alpine——一个极简 Linux，只把编译好的二进制拷过来，指定端口，启动。

为什么要多阶段？单阶段的话，最终镜像里带着 Go 编译器、源码、中间产物——能到 1GB。多阶段之后，只保留一个二进制加上最小的运行时，镜像体积降到大概 15MB。

实用提醒

两个实用提醒。第一，Docker Compose——一个 YAML 文件同时定义应用、数据库、缓存三个服务，一条命令启动整个开发环境。第二，国内网络问题——Docker Hub 官方源直连很慢，一定要在 `daemon.json` 里配置镜像加速地址。

收尾

好，六个模块全部讲完了。我们快速回顾一下：API 路由是后端的骨架，中间件是请求管线上的逻辑单元，数据模型设计决定了你的系统能走多远，缓存让你的服务快起来，Git 和 CI/CD 让你的代码不只是跑在自己电脑上，容器化让你一次构建到处运行。

在 AI 时代理解它们反而更重要——因为 Agent 能帮你写代码，但不能替你理解为什么系统这样设计、出问题时该从哪里排查。开发者至少要能够理解并评估AI提出的方案，而不是AI说什么就是什么。

以上。