GORM 框架自测题

概述

这是一份 GORM 框架的自测试卷，覆盖核心机制 → 查询技巧 → 工程实践全链路知识点。通过选择题、填空题、代码补全、主观题四种题型帮助你检验对 GORM 的理解程度，找到知识盲区后回到对应笔记重新学习。

正文

一、选择题（每题 3 分，共 30 分）

每题只有一个正确答案。

Q1.【软删除的设计】

为什么 GORM 的软删除使用 `DeletedAt` 时间戳而不是 `IsDeleted BOOLEAN`？以下原因**最全面**的是：

A. BOOLEAN 无法区分"未删除"和"误删恢复"两种场景

B. 时间戳记录精确的删除时刻便于审计；NULL 表示未删除语义更清晰

C. BOOLEAN 在 MySQL 中占 1 byte，时间戳占 8 bytes，浪费存储空间

D. BOOLEAN 会导致唯一约束检查变慢

Q1 答案 >

B — 时间戳提供审计信息 + NULL 语义清晰

解析：

`gorm.DeletedAt`（本质是 `time.Time`）相比 BOOLEAN 的优势在于：

1. **精确时间**：记录了确切的删除时刻，方便后续审计和问题排查
2. **NULL = 未删除**：Go 中零值 `time.Time{}` 不等于 nil，GORM 用 `nil` 指针表示"未删除"——这与 BOOLEAN 的 `false = 正常` 相比，语义更不容易混淆
3. **时间排序**：可以按删除时间排序，做定时清理策略时更方便

为什么不是其他选项：

- ❌ A：BOOLEAN 也能通过业务逻辑区分恢复，这不是核心原因
- ❌ C：这是缺点而非设计理由——软删除本身就比物理删除多占空间
- ❌ D：BOOLEAN 和时间戳在唯一约束上的行为差异取决于数据库实现，不是 GORM 的考量

💡 **拓展：** `DeletedAt` 的类型是 `*time.Time`（指针），这样就能用 `nil` 表示未删除。如果用的是非指针 `time.Time`，零值也会被误判为已删除。

Q2.【Preload vs Joins】

关于 `Preload` 和 `Joins` 的区别，以下说法**正确**的是：

A. `Preload` 只执行一条 SQL，`Joins` 执行多条 SQL

B. `Preload` 默认对关联数据加软删除过滤，`Joins` 不会自动加

C. `Joins` 返回的数据会自动填充嵌套 struct，`Preload` 需要手动映射

D. `Joins` 会产生行膨胀（笛卡尔积），`Preload` 不会产生

Q2 答案 >

D — Joins 会产生行膨胀，Preload 不会

解析：

特性	Preload	Joins
SQL 数量	N+1 条（先查主表，再批量 IN 查关联）	通常 1 条复杂 JOIN SQL
行重复	❌ 不会产生	✅ 一对多关系会产生重复行（一个用户有多个订单会返回多行 User）
关联过滤	容易（WHERE 作为第二参数）	需写在 JOIN 子句中，控制不够灵活
内存占用	适中	JOIN 膨胀时较高

逐项分析：

- ❌ A：反了——Preload 是多条 SQL，Joins 是一条
- ❌ B：两者都会自动加软删除过滤，GORM 内部都对关联查询附加 `deleted_at IS NULL`
- ❌ C：两者都能自动填充嵌套 struct，不需要手动映射
- ✅ D：正确！LEFT JOIN 时若 OneToMany 关系会返回多条相同主记录的行（行膨胀），而 Preload 分两次查询完全避免了这个问题

💡 **经验法则：** "优先 Preload，必要时 Joins"——除非确定关联数据量极小，否则 Preload 更安全可控。

Q3.【First vs Take vs Last】

三个方法都要求带条件，关于它们的行为差异，以下说法**正确**的是：

A. `First` 和 `Take` 都会按主键排序后取第一条

B. `Take` 是无序的，适合随机抽奖或推荐场景

C. `Last` 不按主键排序，而是按创建时间倒序

D. 三者不带 Where 都能正常工作

Q3 答案 >

B — Take 无序，适合随机场景

解析：

方法	排序方式	结果确定性	是否需要 Where
`First`	主键升序（ORDER BY PK ASC LIMIT 1）	✅ 确定性	✅ 必须
`Take`	**不指定任何排序**	❌ 任意一条	✅ 必须
`Last`	主键降序（ORDER BY PK DESC LIMIT 1）	✅ 确定性	✅ 必须

逐项分析：

- ❌ A：`Take` 不带排序，数据库返回哪条就是哪条——这恰恰是关键区别
- ✅ B：正确。`Take` 等同于 `SELECT * FROM table LIMIT 1`，没有 ORDER BY，所以每次拿到的可能不同，正好适合"随机"语义
- ❌ C：`Last` 是按主键倒序，不是创建时间

- ❌ D：文档明确写了三者都需要至少一个 Where 条件

💡 **实际选择：** 要"最新的记录"用 `First`；要"随便来一条"用 `Take`；要"最后一条"用 `Last`。

Q4.【Updates 的 zero value 陷阱】

以下代码的输出结果是？

```
user := User{Name: "Alice", Age: 25}
db.Model(&user).Updates(User{Name: "", Status: "active"})
```

A. UPDATE SET name='', status='active' —— Name 被更新为空字符串

B. UPDATE SET status='active' —— Name 跳过了，因为空串是零值

C. UPDATE SET name='', status='active' —— struct 方式的 Updates 不会跳过零值

D. 报错，因为不能传入零值

Q4 答案 >

B — struct 方式跳过了零值字段

解析：

GORM 的 `Updates` 有两种传参方式，处理方式截然不同：

方式	零值处理	生成的 SQL
`Updates(struct)`	**跳过零值字段**（Zero Value Skipped）	`UPDATE users SET status='active' WHERE ...`
`Updates(map[string]any)`	**写入所有 key**（包含零值）	`UPDATE users SET name='', status='active' WHERE ...`

在上面的例子中，`Name: ""` 是 string 类型的零值，所以 `Updates(User{Name: "", Status: "active"})` 只会生成 `SET status='active'`。

真实踩坑案例： 用户想把昵称清空，调用了 `Updates(User{Nickname: ""})`，结果什么 SQL 都没生成——数据完全没有变化！

💡 **解决方案：** 需要更新零值时用 map 方式：`Updates(map[string]any{"name": ""})`。

Q5.【事务回调的错误返回值】

```go
err := db.Transaction(func(tx *gorm.DB) error {
    var account Account
    if err := tx.First(&account, 1).Error; err != nil {
        return err
    }
    account.Balance -= 100
    tx.Save(&account)
    return nil
})
```

如果 `tx.First` 找不到账户（返回 `ErrRecordNotFound`），会发生什么？

A. 错误被静默吞掉，继续执行 Save

B. 函数 return err → GORM 自动 Rollback，最终 `err == ErrRecordNotFound`

C. 函数 panic，因为 error 没有包装

D. GORM 自动把错误转换为 HTTP 404 响应

Q5 答案 >

B — Transaction() 回调返回 error 自动回滚

解析：

`db.Transaction()` 是 GORM 提供的便捷事务 API，它的核心机制是：

1. **内部自动 Begin**：进入回调前开启事务
2. **return nil → Commit**：回调返回 nil 说明一切正常，GORM 自动提交
3. **return non-nil error → Rollback**：回调返回任何非 nil error，GORM 自动回滚
4. **panic → Rollback**：回调内 panic，GORM 捕获后也回滚

在这个例子中：

- `tx.First` 找不到记录 → 返回 `ErrRecordNotFound`
- `return err` → GORM 检测到非 nil → 自动 `Rollback()`
- 外层 `err` 拿到的是包装后的 `ErrRecordNotFound`

为什么要这样设计？ 避免了手动写 `Begin/Rollback/Commit` 的大量样板代码，且天然防止"事务泄漏"——出了闭包就拿不到 tx 实例了。

💡 **限制：** 回调内不能再调用 `Transaction()`，否则 panic——因为此时已经处于事务中了。

Q6.【WHERE vs HAVING】

以下 GORM 查询语句的错误原因是：

```
db.Model(&Order{}).Where("COUNT(*) > 5").Group("user_id").Find(&result)
```

A. GORM 不支持 GROUP BY

B. WHERE 不能在 GROUP BY 之前执行聚合函数

C. Find 不能和 Group 连用

D. COUNT 必须配合 Select 才能使用

Q6 答案 >

B — WHERE 在分组前执行，无法使用聚合函数

解析：

SQL 的标准执行顺序是：

> FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY

关键理解： `WHERE` 发生在 `GROUP BY` 之前——此时还没有分组，`COUNT(*)` 毫无意义。SQL 引擎会在解析 WHERE 时就报语法错误。

正确写法： 把聚合条件放到 `HAVING` 中：

```
db.Model(&Order{}).
    Group("user_id").
    Having("COUNT(*) > ?", 5).        // ✅ HAVING 才能用聚合函数
    Find(&result)
```

如果想先用 WHERE 缩小范围再做聚合：

```
db.Model(&Order{}).
    Where("created_at > ?", cutoff).   // 先用 WHERE 过滤单行
    Group("user_id").
    Having("COUNT(*) > ?", 5).         // 再用 HAVING 过滤聚合结果
    Find(&result)
```

💡 **记忆口诀：** "WHERE 筛行，HAVING 筛组"。

Q7.【钩子与操作的关系】

如果一个模型包含 `DeletedAt` 字段并嵌入了 `gorm.Model`，当调用 `db.Delete(&user)`（软删除）时，哪个钩子**会被调用**？

A. `BeforeDelete` 和 `AfterDelete`

B. `BeforeUpdate` 和 `AfterUpdate`

C. `BeforeDelete` 和 `AfterUpdate`

D. 没有任何钩子被调用

Q7 答案 >

B — BeforeUpdate 和 AfterUpdate

解析：

这是软删除最常见的误区之一。虽然名字叫 `Delete()`，但在软删除模式下它发送的是 **UPDATE 语句**：

```
UPDATE users SET deleted_at = NOW() WHERE id = ?
```

因此触发的是 UPDATE 钩子链：

```
BeforeDelete（❌ 不会被调用）
  ↓
BeforeUpdate  （✅ 被调用）→ SQL: UPDATE ... SET deleted_at=...
  ↓
AfterUpdate   （✅ 被调用）
  ↓
AfterDelete   （❌ 不会被调用）
```

如果想触发 BeforeDelete / AfterDelete： 需要用 `Unscoped().Delete()` 执行真正的 DELETE 语句。

💡 **工程影响：** 如果你在 `BeforeDelete` 里做了级联清理（如删除 Redis Session），软删除时这些逻辑**不会执行**——需要额外处理。

Q8.【DryRun vs Debug】

关于 DryRun 模式，以下说法**错误**的是：

A. DryRun 会构建完整 SQL 但**不执行**，零网络开销

B. DryRun 的参数占位符会保持为 `?`，不会展开为实际值

C. `Debug()` 方法会修改全局 Logger 配置，影响后续所有查询

D. 可以用 DryRun 在 CI Pipeline 中校验 struct tag 变更是否合法

Q8 答案 >

C — Debug() 不影响全局配置

解析：

特性	DryRun	Debug()
是否执行 SQL	❌ 不执行	✅ 执行
性能开销	零（无网络 IO）	正常查询开销
参数占位符	保持 `?`	展开为实际值
全局影响	无	无——返回新 `*gorm.DB` 实例

逐项分析：

- ✅ A：正确。DryRun 只做 SQL 构建，不走网络
- ✅ B：正确。预编译语句的参数化就是安全的体现
- ❌ C：**错误！** `Debug()` 返回一个新的 `*gorm.DB` 实例，原 DB 不受影响。这是函数式编程的思想
- ✅ D：正确。可以在 CI 中跑 DryRun 验证迁移脚本不会引入破坏性变更

💡 **实战用法：** `db.Session(&gorm.Session{DryRun: true}).First(&user, 1)` 可以用于审计复杂的链式调用生成的最终 SQL。

Q9.【Preload 与 N+1 问题】

以下代码中提到了 **N+1 问题**，请问这里的 N 指的是：

```
var users []User
db.Preload("Orders").Find(&users)
```

A. 用户的总数

B. Orders 的总行数

C. 预处理阶段执行的次数

D. 数据库连接数

Q9 答案 >

A — 用户总数

解析：

Preload 的工作方式是"两步走"：

```
SQL 1: SELECT * FROM users;                          ← 获取 N 个用户
SQL 2: SELECT * FROM orders WHERE user_id IN (1,2,...); ← 一次性批量查询所有关联订单
```

这里的"N+1"名称来源于另一种**懒加载**（手动循环查关联）的模式：

```go
var users []User
db.Find(&users)                    // SQL 1: 查用户
for _, u := range users {
    db.Where("user_id = ?", u.ID).Find(&u.Orders)  // SQL 2~N+1: 每个用户一条 SQL
}
```

Preload 把 N+1 优化成了**恰好 2 条**（无论用户多少），因为第二条 SQL 用的是 `IN (...)` 批量查询。

💡 **延伸：** Preload 的深度是层级维度——`Preload("Orders.Products")` 变成 3 条 SQL（Users + Orders + Products），每层各一次批量 IN 查询。

Q10.【连接池配置】

生产环境中连接池的配置，以下做法**正确**的是：

A. `SetMaxOpenConns(0)` 不设上限，让 Go 自己决定最大连接数

B. `SetConnMaxLifetime(8 * time.Hour)` 匹配 MySQL 默认的 wait_timeout

C. `SetMaxIdleConns(50)` 设为 MaxOpenConns 的一半以应对突发流量

D. `SetConnMaxLifetime(5 * time.Minute)` 小于数据库侧的 wait_timeout

Q10 答案 >

D — ConnMaxLifetime 必须小于数据库 wait_timeout

解析：

参数	常见错误	正确做法
`MaxOpenConns`	设为 0（无上限）	显式设置，不超过 MySQL max_connections
`MaxIdleConns`	设得过大浪费资源	CPU 核数或 10~20，约为 MaxOpenConns 的 10%~25%
`ConnMaxLifetime`	设为 8h 匹配 MySQL 默认	**设为 5~10 分钟**，远小于 MySQL 的 8h wait_timeout
`ConnMaxIdleTime`	不设置	设为 5 分钟，减少闲置连接占用

关键原理： MySQL 默认 `wait_timeout = 8h`——空闲连接超过 8 秒务端主动断开。如果 Go 的连接池不知道这一点（即 `ConnMaxLifetime >= 8h`），它会从池中取出已经断开的旧连接，导致 `server has gone away` 错误。

正确的参数关系： `IdleTime < Lifetime << wait_timeout`

💡 **经验公式：** `MaxOpenConns = CPU核心数 × 2 + 磁盘数`。例如 4 核 2 盘 → 10（I/O 密集型可放宽到 20~50）。

二、填空题（每题 3 分，共 24 分）

根据知识填写空缺的代码或概念。

Q11.【建表与表名推导】

```go
type Article struct {
    ID    uint
    Title string
}

// GORM 推断的表名是 _________
db.AutoMigrate(&Article{})

// 显式指定表名的方法是实现 _________ 方法
```

Q11 答案 >

`articles` ； `TableName() string`

解析：

GORM 的默认命名策略是将 struct 名转为蛇形复数形式：

- `User` → `users`
- `Article` → `articles`

- `OrderItem` → `order_items`

`TableName()` 是 model 级别的约定，GORM 会优先使用其返回值作为表名。用值接收者 `(Article)` 而非指针 `(Article)` 是因为 GORM 内部先尝试值接收者，找不到再试指针接收者——值接收者兼容性更好。

💡 **注意：** `db.Table("custom_name")` 只在当前链式调用中生效，不会修改模型的 `TableName()` 返回值。

Q12.【批量插入参数含义】

```
users := make([]User, 350)
db.CreateInBatches(&users, 100).Error
```

上面代码总共执行 _ 批 INSERT，每批最多 _ 条记录。

Q12 答案 >

4 ； 100

解析：

`CreateInBatches(slice, batchSize)` 将 slice 拆分为多个批次：

```
350 ÷ 100 = 3.5 → 向上取整为 4 批
第 1 批：100 条 → INSERT INTO users VALUES (...), (...), ...
第 2 批：100 条
```

```
第 3 批：100 条
第 4 批：50 条   ← 剩余的不足一批的数据单独执行
```

如果 `batchSize = 200`：350 ÷ 200 = 1.75 → 2 批（200 + 150）

💡 **底层机制：** GORM 内部的 `Create` 也会自动拆批（约 256 条/批），但不可自定义。需要精确控制时请用 `CreateInBatches`。

Q13.【关联查询的外键位置】

```go
type User struct {
    ID   uint
    Name string
}

type Order struct {
    ID     uint
    UserID uint
    User   User `gorm:"_________"`  // 空1：声明关联类型
}
```

`Order` **属于** `User`，应使用的关联类型是 ____（填关联类型名）；该关联的外键位于 ____（填"当前方"或"被关联方"）的表中。

Q13 答案 >

`belongsTo` ； **当前方**

解析：

GORM 四种关联的核心区别在于**外键在哪张表上**：

关联类型	描述	外键位置	示例
HasOne	一对一	被关联方	User → Profile
HasMany	一对多	被关联方	User → Orders
BelongsTo	多对一	**当前方**	Order → User
ManyToMany	多对多	中间表	Order ↔ Product

`BelongsTo` 表示"属于"关系——订单属于某个用户。外键 `UserID` 定义在 `Order` 表（当前方），而不是 `User` 表。

💡 **判断口诀：** "谁 belongs to 谁，外键就在谁这边"——Order belongs to User，外键在 Order 表。

Q14.【错误判断方式】

GORM 的错误应该通过 ___ 来判断是否等于 `gorm.ErrRecordNotFound`，而不能直接用 `==` 比较。

Q14 答案 >

```
errors.Is(err, gorm.ErrRecordNotFound)
```

解析：

GORM 的哨兵错误（如 `ErrRecordNotFound`）内部会用 `fmt.Errorf("%w", ...)` 进行包装，形成错误链。标准库的 `errors.Is()` 能逐层识别哨兵错误，而直接 `==` 比较在复杂场景中可能漏判。

```go
result := db.First(&user, 1)
if errors.Is(result.Error, gorm.ErrRecordNotFound) {
    // 正确处理
}
```

💡 **最佳实践：** 统一使用 `errors.Is()` 做哨兵错误判断，`errors.As()` 做具体错误类型的类型断言。

Q15.【Upsert 冲突策略】

MySQL 中 ON DUPLICATE KEY UPDATE 的等效 GORM 写法：

```go
db.Clauses(clause.OnConflict{
    Columns:   []clause.Column{{Name: "email"}},
    DoUpdates: clause._________("last_login"),  // 空1：填入方法名
}).Create(&users)
```

Q15 答案 >

`AssignmentColumns`

解析：

`clause.OnConflict` 支持多种冲突策略：

方法	作用	生成 SQL
`DoNothing`	冲突时不做任何事	`ON CONFLICT DO NOTHING`
`AssignmentColumns(["col"])`	冲突时更新指定列	`ON CONFLICT DO UPDATE SET col = excluded.col`
`Assignments(gorm.Expr(...))`	自定义表达式	`ON CONFLICT DO UPDATE SET col = expr`

对于 PostgreSQL，同样的写法生成 `ON CONFLICT (email) DO UPDATE SET last_login = excluded.last_login;`。

💡 **进阶：** 冲突时递增计数器：

```
DoUpdates: clause.Assignments(gorm.Expr("count = count + 1"))
```

Q16.【游标分页参数计算】

传统 OFFSET 分页中，第 5 页、每页 20 条数据的 Offset 值为 ____；对应的计算公式是 ____（用变量表达）。

Q16 答案 >

80 ； `(page - 1) * pageSize`

解析：

页码	每页数	Offset	结果
第 1 页	20	(1-1)×20 = 0	第 1-20 条

页码	每页数	Offset	结果
第 3 页	20	(3-1)×20 = **40**	第 41-60 条
第 5 页	20	(5-1)×20 = **80**	第 81-100 条

```
offset := (page - 1) * pageSize
db.Limit(pageSize).Offset(offset).Find(&items)
```

💡 **深度翻页问题：** OFFSET 80000 时数据库仍需扫描并跳过前面 80000 行。百万级数据下应改用游标分页（Keyset Pagination）。

Q17.【批量更新字段控制】

```
// 只更新白名单中的字段（忽略 email）
db.Model(&User{}).
    Where("status = ?", "trial").
    ________("status", "plan").          // 空1：白名单控制
    Updates(map[string]any{
        "status": "active",
        "plan":   "pro",
        "email":  "should_not_change@test.com",
    })
```

Q17 答案 >

`Select`

解析：

方法	类型	效果
`Select("col1", "col2")`	**白名单**	只更新指定的字段
`Omit("col1")`	**黑名单**	除了指定字段外的其他字段全部更新

```
// 黑名单方式（排除 password）
db.Model(&User{}).
    Omit("password", "secret_key").
    Updates(map[string]any{
        "name":     "new name",
        "password": "new pass", // 被忽略
    })
```

两者可以组合：`Select("a", "b").Omit("b")` = 只更新 a。

💡 **注意：** `Select` 和 `Omit` 只对 `Updates` 有效，对 `Save` 和 `UpdateColumn` 等全量写入方法无效。

Q18.【PrepareStmt 缓存机制】

开启 PrepareStmt 缓存后，GORM 使用 ____ 存储已编译的 SQL 语句。这对 ____ 类查询有显著的性能提升。

Q18 答案 >

`sync.Map` ； **固定模式 / 重复执行**（或"热点查询"）

解析：

`PrepareStmt: true` 的工作原理：

1. 第一次执行 `SELECT * FROM users WHERE id = ?` → GORM 预编译 SQL，存入 `sync.Map`
2. 第二次执行相同的 SQL → 直接从缓存取预编译语句，跳过解析和计划编译

适用场景： 固定的查询模式（如按 ID 查询用户、按状态查询列表）

不适用场景： 大量参数不同的动态 SQL → 缓存命中率低，徒增内存消耗

```
db, _ := gorm.Open(mysql.Open(dsn), &gorm.Config{
    PrepareStmt: true,
})
```

💡 **重要提醒：** PrepareStmt 的缓存是**独立于连接池**的，但它仍然依赖底层连接。高并发下仍需合理配置连接池大小。

三、代码补全（每题 6 分，共 30 分）

补充代码中空缺的部分。有些题目有多个空。

Q19.【级联创建与关联追加】

```go
// 创建用户的同时创建 Profile 并关联商品
user := User{
    Name: "Alice",
    Profile: Profile{AvatarURL: "https://example.com/avatar.png"},
    Orders: []Order{
        {Amount: 99.9, Products: []Product{{Name: "Book", Price: 29}}},
    },
}
if err := db._________(&user).Error; err != nil { // 空1：方法名
    log.Fatal(err)
}

// 给已有订单添加商品（追加，不覆盖）
product := Product{Name: "New Book", Price: 39}
db.Model(&order).Association("Products")._________(&product)      // 空2：方法名
```

Q19 答案 >

`Create` ; `Append`

解析：

空 1： `db.Create(&user)` 会自动级联写入关联数据——先生成 INSERT INTO users，再 INSERT INTO profiles，最后 INSERT INTO orders。这是 GORM 的"递归保存"特性。

空 2： Association API 提供了五个核心操作：

方法	行为	适用场景
`Append`	追加（新增不删除旧的）	累积添加
`Replace`	替换全部（先删旧再加新）	重新赋值
`Delete`	删除指定关联	移除单项
`Clear`	清空所有关联	批量移除
`Count`	统计关联数量	计数

💡 **Append 的 SQL 效果：** 只是向中间表 INSERT 新记录，不会影响已有的关联关系。

Q20.【事务中的锁行读取】

```go
func TransferMoney(fromID, toID uint, amount float64) error {
    return db.Transaction(func(tx *gorm.DB) error {
        // 转出账户需要悲观锁，防止并发扣款超余额
        var fromAccount Account
        if err := tx._______("gorm:query_option", "FOR UPDATE").
            First(&fromAccount, fromID).Error; err != nil { // 空1：设置查询选项的方法
```

```
            return fmt.Errorf("查询转出账户失败: %w", err)
        }

        if fromAccount.Balance < amount {
            return errors.New("余额不足")
        }

        fromAccount.Balance -= amount
        if err := tx._________(&fromAccount).Error; err != nil { // 空2：更新账户的方法
            return fmt.Errorf("扣款失败: %w", err)
        }

        var toAccount Account
        if err := tx.First(&toAccount, toID).Error; err != nil {
            return fmt.Errorf("查询入账账户失败: %w", err)
        }
        toAccount.Balance += amount
        if err := tx.Save(&toAccount).Error; err != nil {
            return fmt.Errorf("入账失败: %w", err)
        }

        return nil // 返回 nil → 自动 Commit
    })
}
```

Q20 答案 >

`Set` ； `Save`

解析：

空 1： `tx.Set("key", value)` 用于在查询中携带额外的 session 级选项。`"gorm:query_option"` 是一个内置键名，GORM 会将其原样拼接到 SQL 末尾——`SELECT ... FOR UPDATE` 就是在数据库层面加排他锁，防止其他事务同时读取并修改同一行。

空 2： `Save` 是全量更新，无视零值全部写回。这里用 `Save` 是因为需要确保 `Balance` 字段的最新值被持久化。

```
// FOR UPDATE 的效果对比：
// ❌ 无锁：两个并发请求同时读到 Balance=100，都扣 60，结果 Balance=-20（超卖）
// ✅ 有锁：第二个请求必须等待第一个事务结束后才能读，保证原子性
```

💡 **延伸：** PostgreSQL 同样使用 `SELECT ... FOR UPDATE`；SQLite 默认串行执行事务，不需要显式加锁。

Q21.【条件预加载 + 排序】

```go
// 加载用户及其订单，但只加载金额大于 100 的订单，并按金额降序排列
db.Preload("Orders", func(db *gorm.DB) *gorm.DB {
    return db._______("amount > ?", 100).
              _________("amount DESC")
}).Find(&users)
```

Q21 答案 >

`Where` ； `Order`

解析：

`Preload` 的第二个参数可以是函数形式的条件构造器：

```
db.Preload("AssociationName", func(db *gorm.DB) *gorm.DB {
    return db.Where("condition").Order("field")
}).Find(&parent)
```

这个函数的签名固定为 `func(*gorm.DB) *gorm.DB`——接收 DB 实例，返回修饰后的 DB 实例。可以在里面自由链式调用 `Where`、`Order`、`Limit`、`Select` 等方法。

💡 **注意：** 函数形式的条件只适用于 Preload，不适用于 Joins。Joins 的条件需要写在 `Joins("Table", "condition")` 的第二个参数中。

Q22.【JSON 自定义类型的 Scanner】

```
type JSONMap map[string]interface{}

// Scan 方法负责从数据库读出数据（DB → Go）
func (j *JSONMap) Scan(value interface{}) error {
    if value == nil {
```

```
        *j = nil
        return nil
    }
    str, ok := value._________                          // 空1：类型断言
    if !ok {
        str = []byte(fmt.Sprintf("%v", value))
    }
    return json.Unmarshal(str, j)                       // 空2：反序列化的目标
}
```

Q22 答案 >

`([]byte)` ； `j` （指针接收者本身）

解析：

`sql.Scanner` 接口定义了 `Scan(value interface{}) error` 方法，由 GORM 在查询后调用，将数据库读出的原始值转换为 Go 类型。

```
type Scanner interface {
    Scan(value interface{}) error
}
```

流程分解：

1. `value` 是从数据库来的原始字节数组 `[]byte`

2. 类型断言 `value.([]byte)` 提取字节切片

3. `json.Unmarshal(str, j)` 将 JSON 反序列化到 `*JSONMap`（指针接收者，这样才能修改外部结构体）

💡 **为什么用指针接收者？** `Scan` 需要将反序列化后的内容**写回**目标对象。如果用值接收者 `(j JSONMap)`，修改的只是副本，原始结构体不会被改变。同理 `Value()` 用值接收者 `(j JSONMap)` 是因为只需读不需写。

Q23.【事务内的条件检查与日志输出】

```
func ExecTx(db *gorm.DB, fn func(tx *gorm.DB) error) error {
    tx := db.Begin()
    defer func() {
        if r := recover(); r != nil {
            tx._________                                    // 空1：出错时回滚
            panic(r)
        }
    }()

    if err := fn(tx); err != nil {
        tx.Rollback()
        return err
    }
    return tx._________.Error                               // 空2：提交事务的方法
}
```

Q23 答案 >

`Rollback()` ； `Commit()`

解析：

这是一个通用的事务封装函数 `ExecTx`，消除了每个业务函数里重复的 Begin/Rollback 样板代码。

```
// 使用方式
err := ExecTx(db, func(tx *gorm.DB) error {
    // 业务逻辑...
    return nil // 成功 → ExecTx 自动 Commit
}) // 失败 → ExecTx 自动 Rollback
```

关键点：

1. `defer` 只在 panic 分支中执行 `Rollback()`，正常路径不调用——避免 Commit 后再 Rollback 报错
2. `tx.Commit()` 返回 `*gorm.DB`（和大多数 GORM 方法一样），`.Error` 获取最终结果

💡 **对比：** 简单场景直接用 `db.Transaction(callback)` 即可，代码更简洁。复杂业务（如跨包传递 tx）才需要 `ExecTx` 这样的封装。

四、主观题（每题 4 分，共 16 分）

简要回答即可，不需要长篇大论。关键是说出你的理解。

Q24.【何时不该用事务?】

以下场景中哪些**不需要**用事务？请给出理由。

① 单条 `INSERT` 记录

② 转账：扣 A 加 B

③ COUNT / SUM 统计查询

④ 下单：减库存 + 创建订单 + 生成流水

Q24 参考答案 >

① **不需要**——单条 INSERT 本身就是原子操作，数据库层面保证了要么全成功要么全失败，无需额外的事务包裹。

② **必须用事务**——涉及两笔以上的资金变动，如果没有事务，可能出现"A 扣了钱但 B 没到账"的数据不一致。

③ **不需要**——只读查询不涉及数据修改，不存在一致性问题。可以使用普通查询 + 缓存来减轻数据库压力。

④ **必须用事务**——三步操作具有业务原子性要求：如果减库存成功但创建订单失败，会导致库存少了但没有对应订单。

总结： 涉及多步数据写入且要求原子性的场景才需要事务。单条写入、只读查询都可以免去事务开销。

Q25.【软删除 + 唯一索引冲突】

某电商系统中 `Product.SKU` 设置了 `uniqueIndex`。现有记录 SKU="ABC-123" 已被软删除。此时还能再次创建 SKU="ABC-123" 的新产品吗？MySQL 和 PostgreSQL 的行为是否一致？说明原因和你的解决方案。

Q25 参考答案 >

不一致！

- MySQL（InnoDB）：✅ 可以插入。InnoDB 在处理软删除行的唯一约束时有历史缺陷——已软删除的行不参与唯一约束检查，所以允许两条 SKU 相同的记录存在（一条被删，一条未删）。但这意味着可能出现"同一 SKU 多条有效记录"的数据不一致。
- PostgreSQL：❌ 不能插入。PG 的正确行为是唯一约束包含软删除行，会报 duplicate key 错误。

解决方案：

1. **方案一（PG）：** 用部分唯一索引 `CREATE UNIQUE INDEX ... WHERE deleted_at IS NULL`，只约束未删除行
2. **方案二（通用）：** 加版本字段 `SlugVersion uint`，保证同一 SKU 不会同时出现在两条未删除记录中
3. **方案三：** 应用层校验——插入前先检查是否有未删除的同 SKU 记录

💡 **核心教训：** 不要依赖单一数据库的行为一致性，应在应用层做好校验。

Q26.【BeforeUpdate Changed 的最佳实践】

```go
func (u *User) BeforeUpdate(tx *gorm.DB) error {
    if tx.Statement.Changed("Password") {
        hashed, _ := bcrypt.GenerateFromPassword([]byte(u.Password), bcrypt.DefaultCost)
        u.Password = string(hashed)
    }
```

```
    return nil
}
```

为什么要用 `Changed("Password")` 判断，而不直接在 `BeforeUpdate` 里无条件哈希？这样做有什么优势？

Q26 参考答案 >

核心原因是**避免重复哈希**。

如果无条件哈希，每次 UPDATE 操作（即使改的是名字或邮箱）都会重新对密码做 bcrypt 哈希。bcrypt 故意很慢（计算密集型），频繁调用会带来不必要的性能开销。

此外，如果用户修改了其他字段但未修改密码，数据库执行的是 `Updates(struct)`——`Password` 字段不在 struct 中参与更新，但前面的哈希操作仍然白白浪费了 CPU。

`Changed("Password")` 的作用：只有当 `Updates(map)` 的 map 中确实包含了 `password` 键时才执行哈希。这样既保证了安全性（密码改了就加密），又避免了不必要的计算。

💡 **注意：** `Changed` 只对 `Updates` 方法有效，`Save` 是全量写入不会追踪变化。

Q27.【AutoMigrate 的局限性与替代方案】

为什么生产环境不推荐仅靠 `AutoMigrate` 管理数据库迁移？请至少列举三条原因，并简述推荐的替代方案。

Q27 参考答案 >

三条原因：

1. **不可回滚：** AutoMigrate 只能"加"不能"减"。当代码回退到旧版本时，它无法撤销已添加的列或索引——只会跳过已存在的部分。版本化迁移方案通过 Down 脚本可以轻松应对回退场景。
2. **不可预测：** GORM 内部决定迁移的执行顺序和具体 ALTER 语句，不同数据库方言的行为可能不一致。版本化迁移用手写 SQL，完全可控。
3. **团队协作困难：** 每个人改了 model 就 auto migrate，容易产生冲突和不一致。版本化迁移通过文件合并可以更好地管理多人协作。

推荐替代方案： 本地开发用 `AutoMigrate + DryRun` 快速迭代，生产环境用专门的迁移工具（如 Goose 或 golang-migrate），手写版本化的 Up/Down SQL 脚本。这样既享受了开发便利，又保证了生产环境的可控性和可回滚能力。

评分参考

题目类型	满分	权重
选择题（Q1-Q10）	30 分	30%
填空题（Q11-Q18）	24 分	24%
代码补全（Q19-Q23）	30 分	30%
主观题（Q24-Q27）	16 分	16%
总计	**100 分**	**100%**

及格线： ≥70 分

优秀线： ≥85 分