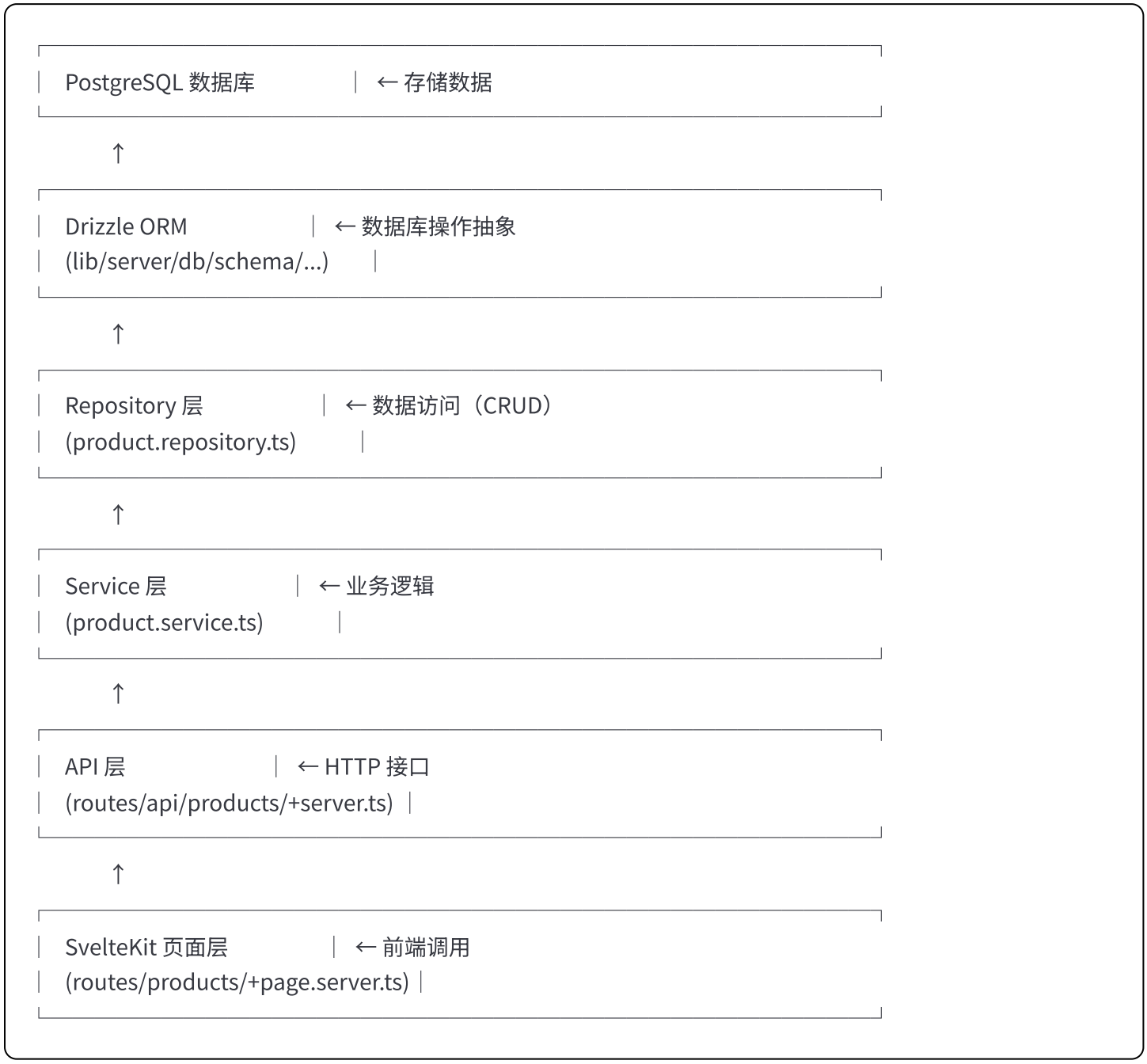
商品后端逻辑完整解析

🎯 核心架构理解

整体分层（从下到上）



📊 数据流转过程

场景1：用户访问首页查看商品列表

Step 1: 用户访问 `/`

```typescript
```

```
// 浏览器
GET http://localhost:5173/
```

Step 2: SvelteKit 执行页面服务端逻辑

typescript

```typescript
// src/routes/+page.server.ts
export const load: PageServerLoad = async ({ fetch, url }) => {
 // 1. 从 URL 获取查询参数
 const search = url.searchParams.get('search') || '';
 const categoryId = url.searchParams.get('categoryId') || '';
 const page = url.searchParams.get('page') || '1';

 // 2. 构建 API 请求参数
 const params = new URLSearchParams({
  published: 'true',    // 只显示已发布的
  inStock: 'true',      // 只显示有库存的
  sortBy: 'created_desc', // 按创建时间倒序
  page,
  pageSize: '20'
 });

 if (search) params.append('search', search);
 if (categoryId) params.append('categoryId', categoryId);

 // 3. 调用内部 API（注意：这是服务端到服务端的调用）
 const productsRes = await fetch(`/api/products?${params}`);

 // 4. 解析响应
 const { products, pagination } = await productsRes.json();

 // 5. 返回给前端页面
 return {
  products,    // 商品列表
  pagination,  // 分页信息
  filters: { search, categoryId, page: parseInt(page) }
 };
};
```

关键点理解：

- ✅ `fetch` 在服务端执行，不会暴露给浏览器
- ✅ 这是 **SSR（服务端渲染）**，数据在服务端就准备好了
- ✅ 返回的数据会通过 `data` prop 传给 `+page.svelte`

Step 3: API 层处理请求

typescript

```typescript
// src/routes/api/products/+server.ts
export const GET: RequestHandler = async ({ url }) => {
 try {
  // 1. 解析查询参数
  const filter = {
   categoryId: url.searchParams.get('categoryId') || undefined,
   status: url.searchParams.get('status') as any,
   isPublished: url.searchParams.get('published') === 'true' ? true : undefined,
   search: url.searchParams.get('search') || undefined,
   page: parseInt(url.searchParams.get('page') || '1'),
   pageSize: parseInt(url.searchParams.get('pageSize') || '20')
  };

  // 2. 调用 Service 层
  const products = await productService.getProducts(filter);

  // 3. 返回 JSON 响应
  return json({
   products,
   pagination: {
    page: filter.page,
    pageSize: filter.pageSize,
    total: products.length
   }
  });
 } catch (err) {
  // 错误处理
  return json({ error: 'Internal server error' }, { status: 500 });
 }
};
```

关键点理解：

- ✅ API 层只负责：接收请求 → 调用 Service → 返回响应
- ✅ **不包含任何业务逻辑**
- ✅ 统一的错误处理

Step 4: Service 层执行业务逻辑

typescript

```
// src/lib/server/product/product.service.ts
export class ProductService {
  async getProducts(filter: ProductFilter) {
    // 直接调用 Repository 层
    return await productRepository.findMany(filter);
  }
}
```

关键点理解：

- ✅ Service 层是业务逻辑层
- ✅ 对于简单的查询，直接转发给 Repository
- ✅ 对于复杂操作（如创建商品），会有额外的业务逻辑

Step 5: Repository 层查询数据库

typescript

```ts
// src/lib/server/product/product.repository.ts
export class ProductRepository {
 async findMany(filter: ProductFilter) {
  // 1. 构建查询条件
  const conditions = [];

  if (filter.categoryId) {
   conditions.push(eq(products.categoryId, filter.categoryId));
  }

  if (filter.isPublished !== undefined) {
   conditions.push(eq(products.isPublished, filter.isPublished));
  }

  if (filter.search) {
   conditions.push(
    or(
     ilike(products.name, `%${filter.search}%`),
     ilike(products.description, `%${filter.search}%`)
    )!
   );
  }

  // 2. 执行查询（使用 Drizzle ORM）
  let query = db
   .select({
    id: products.id,
    name: products.name,
    slug: products.slug,
    price: products.price,
    stock: products.stock,
    // ... 其他字段
   })
   .from(products)
   .leftJoin(categories, eq(products.categoryId, categories.id))
   .leftJoin(users, eq(products.sellerId, users.id))
   .where(conditions.length > 0 ? and(...conditions) : undefined);

  // 3. 排序
  query = query.orderBy(desc(products.createdAt));

  // 4. 分页
  query = query.limit(filter.pageSize).offset((filter.page - 1) * filter.pageSize);

  // 5. 执行并返回结果
  return await query;
```

```
  }
}
```

关键点理解：

- ✅ Repository 只负责数据访问，不包含业务逻辑
- ✅ 使用 Drizzle ORM 构建类型安全的 SQL
- ✅ 返回的是原始数据库数据

Step 6: 数据返回到前端

svelte

```
<!-- src/routes/+page.svelte -->
<script lang="ts">
  import type { PageData } from './$types';

  let { data } = $props();

  // data.products 就是从 +page.server.ts 返回的数据
  let products = $derived(data.products);
</script>

<!-- 渲染商品列表 -->
{#each products as product (product.id)}
  <article>
    <h3>{product.name}</h3>
    <p>¥{product.price}</p>
  </article>
{/each}
```

完整流程总结：

```
用户访问 /
  ↓
+page.server.ts.load() 执行
  ↓
fetch('/api/products?published=true')
  ↓
API 层接收请求
  ↓
productService.getProducts(filter)
  ↓
productRepository.findMany(filter)
  ↓
Drizzle 执行 SQL 查询
  ↓
PostgreSQL 返回数据
  ↓
层层返回到 +page.svelte
  ↓
渲染 HTML
  ↓
发送给用户浏览器
```

📝 场景2：商家创建商品

Step 1: 商家访问创建页面

```typescript
// src/routes/products/create/+page.server.ts
export const load: PageServerLoad = async ({ locals, fetch }) => {
 // 1. 检查权限
 if (!locals.user?.permissions.includes('product.write')) {
  throw redirect(302, '/auth/login');
 }

 // 2. 加载分类列表（供下拉选择）
 const categoriesRes = await fetch('/api/categories');
 const { categories } = await categoriesRes.json();

 return {
  categories // 传给表单使用
 };
};
```

关键点：

- ✅ 页面加载前先检查权限
- ✅ 加载表单需要的数据（分类列表）

Step 2: 商家提交表单

```svelte
<!-- src/routes/products/create/+page.svelte -->
<form method="POST">
  <input type="text" name="name" required />
  <input type="text" name="slug" required />
  <input type="number" name="price" required />
  <input type="number" name="stock" required />
  <button type="submit">创建商品</button>
</form>
```

关键点：

- ✅ `method="POST"` 会触发同路径的 action
- ✅ 表单数据会自动提交到服务端

Step 3: Action 处理表单

```typescript
```

```ts
// src/routes/products/create/+page.server.ts
export const actions: Actions = {
 default: async ({ request, fetch, locals }) => {
  // 1. 再次检查权限（防止绕过）
  if (!locals.user?.permissions.includes('product.write')) {
   return fail(403, { error: '无权限' });
  }

  // 2. 解析表单数据
  const formData = await request.formData();

  const productData = {
   name: formData.get('name')?.toString(),
   slug: formData.get('slug')?.toString(),
   price: parseFloat(formData.get('price')?.toString() || '0'),
   stock: parseInt(formData.get('stock')?.toString() || '0'),
   // ... 其他字段
  };

  // 3. 基础验证
  if (!productData.name || !productData.slug || !productData.price) {
   return fail(400, { error: '请填写必填字段' });
  }

  // 4. 调用 API 创建商品
  const response = await fetch('/api/products', {
   method: 'POST',
   headers: { 'Content-Type': 'application/json' },
   body: JSON.stringify(productData)
  });

  // 5. 处理结果
  if (!response.ok) {
   const error = await response.json();
   return fail(response.status, { error: error.error || '创建失败' });
  }

  const { product } = await response.json();

  // 6. 成功后重定向到商品详情页
  throw redirect(302, `/products/${product.slug}`);
 }
};
```

关键点理解：

- ✅ Action 在服务端执行
- ✅ 验证 → 调用 API → 返回结果或重定向
- ✅ 错误返回给表单，成功则重定向

Step 4: API 创建商品

typescript

```typescript
// src/routes/api/products/+server.ts
export const POST: RequestHandler = async ({ request, locals }) => {
 try {
  // 1. 权限检查
  requirePermission(locals.user, 'product.write');

  // 2. 解析请求体
  const data = await request.json();

  // 3. 基础验证
  if (!data.name || !data.slug || !data.price || data.stock === undefined) {
   return json({ error: 'Missing required fields' }, { status: 400 });
  }

  // 4. 调用 Service 创建
  const product = await productService.createProduct(locals.user!.id, {
   categoryId: data.categoryId,
   name: data.name,
   slug: data.slug,
   description: data.description,
   price: Number(data.price),
   stock: Number(data.stock),
   // ... 其他字段
  });

  // 5. 返回创建的商品
  return json({ product }, { status: 201 });
 } catch (err) {
  if (err instanceof ProductError) {
   return json({ error: err.message }, { status: err.statusCode });
  }
  return json({ error: 'Internal server error' }, { status: 500 });
 }
};
```

关键点：

- ✅ 权限检查使用 guard 函数
- ✅ 验证 → Service → 返回
- ✅ 统一错误处理

Step 5: Service 执行业务逻辑

typescript

```typescript
// src/lib/server/product/product.service.ts
export class ProductService {
 async createProduct(sellerId: string, data: CreateProductData) {
  // 1. 验证分类（如果有）
  if (data.categoryId) {
   const category = await categoryRepository.findById(data.categoryId);
   if (!category) {
    throw new ProductError('Category not found', 'INVALID_DATA', 400);
   }
  }

  // 2. 检查 slug 唯一性
  const existingProduct = await productRepository.findBySlug(data.slug);
  if (existingProduct) {
   throw ProductError.SlugExists;
  }

  // 3. 创建商品
  const product = await productRepository.create({
   ...data,
   sellerId,
   price: data.price.toString(), // 转为字符串（数据库存储格式）
   // ... 其他转换
  });

  return product;
 }
}
```

关键点理解：

- ✅ **业务逻辑在这里**：验证分类、检查唯一性
- ✅ 数据格式转换（如 number → string）
- ✅ 抛出业务异常

Step 6: Repository 插入数据库

typescript

```typescript
// src/lib/server/product/product.repository.ts
export class ProductRepository {
 async create(data: CreateProductData & { sellerId: string }) {
  // 使用 Drizzle 插入数据
  const [product] = await db.insert(products).values(data).returning();
  return product;
 }

 async findBySlug(slug: string) {
  const [product] = await db
   .select()
   .from(products)
   .where(eq(products.slug, slug))
   .limit(1);
  return product;
 }
}
```

关键点：

- ✅ 纯数据操作，无业务逻辑
- ✅ `.returning()` 返回插入的数据
- ✅ Drizzle 自动生成类型安全的 SQL

完整创建流程：

```
商家填写表单并提交
  ↓
+page.server.ts.actions.default() 执行
  ↓
解析表单 + 基础验证
  ↓
fetch('/api/products', { method: 'POST', body: {...} })
  ↓
API 层权限检查
  ↓
productService.createProduct(sellerId, data)
  ↓
Service 层业务逻辑：
 - 验证分类
 - 检查 slug 唯一性
 - 调用 Repository
  ↓
productRepository.create(data)
  ↓
Drizzle 执行 INSERT SQL
  ↓
PostgreSQL 插入数据并返回
  ↓
层层返回到 Action
  ↓
Action 重定向到商品详情页
  ↓
用户看到新创建的商品
```

🔐 权限检查的多层防护

第 1 层：Layout 检查（页面级）

typescript

```typescript
// src/routes/seller/+layout.server.ts
export const load = async ({ locals }) => {
  if (!locals.user?.permissions.includes('product.write')) {
    throw redirect(302, '/auth/login');
  }
  return { user: locals.user };
};
```

作用：

- ✅ 访问 `/seller/*` 下的任何页面都会先检查
- ✅ 无权限直接重定向，不会加载页面

第 2 层：Action 检查（表单提交）

```typescript
// +page.server.ts
export const actions = {
  default: async ({ locals }) => {
    if (!locals.user?.permissions.includes('product.write')) {
      return fail(403, { error: '无权限' });
    }
    // 继续处理
  }
};
```

作用：

- ✅ 防止用户绕过前端直接提交表单
- ✅ 服务端再次验证权限

第 3 层：API 检查（接口调用）

```typescript
// src/routes/api/products/+server.ts
export const POST = async ({ locals }) => {
  requirePermission(locals.user, 'product.write');
  // 继续处理
};
```

作用：

- ✅ 防止直接调用 API
- ✅ 即使是内部调用也会检查权限

第 4 层：Service 检查（所有权）

```typescript
```

```
// product.service.ts
async updateProduct(productId: string, sellerId: string, data) {
  const product = await productRepository.findById(productId);

  // 检查是否是商品所有者
  if (product.sellerId !== sellerId) {
    throw ProductError.Forbidden;
  }

  // 继续更新
}
```

作用：

- ✅ 即使有 `product.write` 权限
- ✅ 也只能操作**自己的**商品
- ✅ 管理员例外（有 `product.manage` 权限）

多层防护示意图：

```
用户请求
  ↓
【Layout 检查】有没有访问该页面的权限?
  ↓ 有
【Action 检查】有没有提交表单的权限?
  ↓ 有
【API 检查】有没有调用接口的权限?
  ↓ 有
【Service 检查】是不是商品所有者?
  ↓ 是
执行操作
```

🔄 数据格式转换理解

为什么需要转换?

数据库存储的格式和前端使用的格式不一定相同：

```
typescript
```

```
// 前端提交的数据
{
  price: 99.99,          // JavaScript number
  stock: 10,             // JavaScript number
  attributes: { color: '红色' }  // JavaScript object
}

// 数据库存储的格式
{
  price: "99.99",        // PostgreSQL DECIMAL → string
  stock: 10,             // PostgreSQL INTEGER → number
  attributes: {"color":"红色"} // PostgreSQL JSONB → object
}

// 返回给前端的格式
{
  price: "99.99",        // 保持字符串，前端显示时转换
  stock: 10,             // 数字类型
  attributes: { color: '红色' }  // 对象类型
}
```

在哪里转换?

Service 层（写入时）

typescript

```
async createProduct(sellerId: string, data: CreateProductData) {
  const product = await productRepository.create({
    ...data,
    sellerId,
    price: data.price.toString(),  // number → string
    compareAtPrice: data.compareAtPrice?.toString(),
    cost: data.cost?.toString()
  });
  return product;
}
```

前端（显示时）

svelte

```
<script lang="ts">
  let { data } = $props();
  let product = $derived(data.product);

  // price 是字符串 "99.99"，直接显示
  let displayPrice = $derived(product.price);

  // 如果需要计算，转为数字
  let numericPrice = $derived(parseFloat(product.price));
</script>

<p>¥{displayPrice}</p>
<p>折扣后: ¥{(numericPrice * 0.9).toFixed(2)}</p>
```

💡 关键设计理解

1. 为什么分这么多层？

单层代码（不推荐）：

```typescript
// 所有逻辑混在一起
export const POST = async ({ request, locals }) => {
  const data = await request.json();

  // 权限检查
  if (!locals.user) throw error(401);

  // 业务逻辑
  if (!data.name) throw error(400);
  const existing = await db.select()...
  if (existing) throw error(409);

  // 数据操作
  const product = await db.insert()...

  return json({ product });
};
```

问题：

- ❌ 难以测试
- ❌ 难以复用

- ❌ 职责不清晰
- ❌ 难以维护

分层代码（推荐）：

```typescript
// API 层：只负责 HTTP
export const POST = async ({ request, locals }) => {
  requirePermission(locals.user, 'product.write');
  const data = await request.json();
  const product = await productService.createProduct(locals.user.id, data);
  return json({ product });
};

// Service 层：只负责业务逻辑
async createProduct(sellerId, data) {
  this.validateData(data);
  await this.checkUniqueness(data.slug);
  return await productRepository.create({ ...data, sellerId });
}

// Repository 层：只负责数据访问
async create(data) {
  return await db.insert(products).values(data).returning();
}
```

优点：

- ✅ 每层职责清晰
- ✅ 易于测试（可以 mock）
- ✅ 易于复用（Service 可以被多个 API 调用）
- ✅ 易于维护

2. 为什么前端还要调用 API？

```typescript
// +page.server.ts
const response = await fetch('/api/products', {...});
```

疑问：前端都在服务端了，为什么不直接调用 Service？

```typescript
```

```
// ❌ 不推荐
const product = await productService.createProduct(...);

// ✅ 推荐
const response = await fetch('/api/products', {...});
```

原因：

1. **统一接口**：API 可以被前端和其他服务调用
2. **权限统一**：API 层统一处理权限
3. **验证统一**：API 层统一验证
4. **错误统一**：API 层统一错误处理
5. **易于测试**：可以直接测试 API 接口

3. locals.user 是怎么来的?

```typescript
// src/hooks.server.ts
export const handle = async ({ event, resolve }) => {
 const sessionId = event.cookies.get('session');

 if (sessionId) {
  // 从 session 获取用户信息（包含角色和权限）
  event.locals.user = await getSessionUser(sessionId);
 }

 return resolve(event);
};
```

流程：

用户登录成功
↓
创建 session 并设置 cookie
↓
用户下次请求时浏览器自动携带 cookie
↓
hooks.server.ts 从 cookie 中获取 session
↓
通过 session 查询用户（包含权限）
↓
存储到 event.locals.user
↓
所有 +page.server.ts 都可以访问 locals.user

📚 总结

核心流程理解

1. **查询商品：**

页面 → API → Service → Repository → 数据库 → 返回

2. **创建商品：**

表单 → Action → API → Service(验证) → Repository → 数据库 → 返回/重定向

3. **权限控制：**

Layout检查 → Action检查 → API检查 → Service检查所有权

关键设计原则

- ✅ **分层明确**：API/Service/Repository 各司其职
- ✅ **权限多层**：Layout/Action/API/Service 多重防护
- ✅ **数据转换**：Service 层统一处理
- ✅ **错误统一**：每层都有错误处理
- ✅ **类型安全**：TypeScript + Drizzle 保证类型

这套设计可以让你：

- 轻松添加新功能
- 轻松修改现有功能
- 代码易于理解和维护
- 安全性有保障