5.8 接口视角 (Interface Viewpoint)

接口视角详细描述了 FitnessAI 系统中各组件之间、系统与外部环境之间的接口契约。本视角关注接口的语法（签名、参数）、语义（行为契约）及非功能属性（性能、可靠性），为系统集成与测试提供精确的接口规约。

5.8.1 接口分类与层次

FitnessAI 系统的接口按层次划分为三类：

5.8.1.1 外部接口（External Interfaces）

系统与外部实体（用户、浏览器运行时、第三方服务）的交互边界。

用户界面接口：

- **GUI 交互契约**：基于 React 的事件驱动模型，用户通过点击、拖拽等手势触发状态变更
- **视觉反馈契约**：Canvas 2D 渲染接口，以 30fps 频率绘制骨骼覆盖层

浏览器运行时接口：

- **MediaDevices API**：`getUserMedia(constraints)` 获取摄像头流，返回 `MediaStream` 对象
- **WebRTC 接口**：视频流捕获与处理
- **Fetch API**：HTTP 请求接口，用于前后端通信

第三方服务接口：

- **智谱 AI OpenAPI**：RESTful HTTP 接口，遵循 OpenAPI 3.0 规范

5.8.1.2 内部组件接口（Component Interfaces）

系统内部组件间的接口契约，包括前端模块间、后端服务层间及前后端之间的接口。

前端组件接口：

- **usePoseDetection Hook 接口**：暴露 `{ videoRef, canvasRef, isActive, exerciseStats, startDetection, stopDetection, resetStats }`
- **ExerciseAnalyzer 接口**：统一分析接口 `analyze(landmarks: Landmark[]): ExerciseAnalysis`

后端服务接口：

- **RESTful API 接口**：遵循 REST 风格，JSON 序列化
- **Repository 接口**：数据访问抽象层，定义 `save()`, `findById()`, `findByUserId()` 等方法

5.8.1.3 数据接口（Data Interfaces）

跨越系统边界的数据结构定义，包括请求/响应载荷、数据库 Schema。

API 数据契约：

- **请求载荷**：JSON 格式，包含 `exercise_type`, `pose_data`, `user_id` 等字段
- **响应载荷**：JSON 格式，包含 `session_id`, `analysis_result`, `feedback` 等字段

5.8.2 RESTful API 接口规约

后端暴露的 RESTful API 接口遵循统一的设计原则：资源导向、无状态、幂等性。

5.8.2.1 认证接口组

接口标识：`/api/auth/*`

接口路径	HTTP 方法	功能	请求体	响应体	认证要求
`/api/auth/register`	POST	用户注册	`{ username, password, email?, nickname? }`	`{ token, user }`	无
`/api/auth/login`	POST	用户登录	`{ username, password }`	`{ token, user }`	无
`/api/auth/me`	GET	获取当前用户信息	无	`{ user_id, username, nickname, email, profile }`	Bearer Token
`/api/auth/change-password`	POST	修改密码	`{ old_password, new_password }`	`{ message }`	Bearer Token

接口契约细节：

- **认证机制**：JWT Token，通过 `Authorization: Bearer {token}` Header 传递
- **Token 有效期**：24 小时
- **密码安全**：SHA-256 哈希存储，最小长度 6 位

5.8.2.2 会话管理接口组

接口标识：`/api/session/*`

接口路径	HTTP 方法	功能	请求体	响应体	认证要求
`/api/session/start`	POST	开始训练会话	`{ exercise_type, user_id? }`	`{ session_id, message }`	可选
`/api/session/{session_id}/data`	POST	提交运动数据	`{ pose_data, is_correct, score, feedback }`	`{ message, session_stats }`	无
`/api/session/{session_id}/end`	POST	结束训练会话	无	`{ session_id, summary }`	无

接口契约细节：

- **会话 ID 格式**：`{user_id}_{YYYYMMDD_HHMMSS}`
- **数据提交频率**：前端 Telemetry Buffer 批量提交，避免高频请求
- **会话状态**：`active` → `completed`

5.8.2.3 AI 服务接口组

接口标识：`/api/ai/*`

接口路径	HTTP 方法	功能	请求体	响应体	认证要求
`/api/ai/generate-plan`	POST	AI 生成健身计划	`{ height?, weight?, age?, gender? }`	`{ daily_goals, weekly_goals, suggestions, bmi, fitness_level }`	Bearer Token

接口契约细节：

- **降级机制**：API Key 未配置或调用失败时，自动降级使用规则引擎
- **响应时间**：AI 调用超时 30 秒，重试 2 次
- **数据脱敏**：发送给 AI 的数据不包含用户 PII

5.8.3 前端组件接口规约

5.8.3.1 usePoseDetection Hook 接口

接口签名：

```
interface UsePoseDetectionReturn {
  videoRef: RefObject<HTMLVideoElement>;
  canvasRef: RefObject<HTMLCanvasElement>;
  isActive: boolean;
  poseResults: PoseResults | null;
  exerciseStats: ExerciseStats;
  startDetection: () => Promise<void>;
  stopDetection: () => void;
  resetStats: () => void;
}
```

接口行为契约：

- **startDetection()**：异步初始化 MediaPipe，启动摄像头流，设置 `isActive = true`
- **stopDetection()**：停止摄像头流，释放资源，设置 `isActive = false`
- **resetStats()**：重置分析器状态，清零计数与得分

5.8.3.2 ExerciseAnalyzer 接口

接口签名：

```
interface ExerciseAnalyzer {
  analyze(landmarks: Landmark[]): ExerciseAnalysis;
  reset(): void;
}

interface ExerciseAnalysis {
  isCorrect: boolean;
  score: number;        // 0-100
  feedback: string;
  count?: number;       // 计数类运动
  duration?: number;    // 计时类运动（秒）
}
```

接口行为契约：

- **analyze()**：纯函数，无副作用，基于几何计算返回分析结果
- **reset()**：重置内部状态（计数、计时器等）

5.8.4 接口图

```
@startuml
package "外部接口层" {
  interface "User Interface" as UI {
    + onClick()
    + onCameraPermission()
  }

  interface "Browser Runtime API" as BrowserAPI {
    + getUserMedia()
    + fetch()
    + Canvas API
  }

  interface "Zhipu AI API" as ZhipuAPI {
    + POST /api/paas/v4/chat/completions
  }
}

package "前端组件接口" {
  interface "usePoseDetection" as PoseHook {
    + startDetection()
    + stopDetection()
    + resetStats()
    + exerciseStats: ExerciseStats
  }

  interface "ExerciseAnalyzer" as Analyzer {
    + analyze(landmarks): ExerciseAnalysis
```

```
    + reset()
  }

  interface "MediaPipe Adapter" as MPAdapter {
    + detect(image): Landmarks
    + onResults(callback)
  }
}

package "后端服务接口" {
  interface "REST API" as RESTAPI {
    + POST /api/auth/login
    + POST /api/session/start
    + POST /api/ai/generate-plan
  }

  interface "Repository Interface" as Repo {
    + save(entity)
    + findById(id)
    + findByUserId(userId)
  }

  interface "Advisor Agent" as Agent {
    + generatePlan(height, weight, age, gender): Plan
  }
}

UI --> PoseHook
BrowserAPI --> MPAdapter
PoseHook --> Analyzer
Analyzer --> MPAdapter
PoseHook --> RESTAPI
RESTAPI --> Repo
RESTAPI --> Agent
Agent --> ZhipuAPI

@enduml
```

图5-16：系统接口层次图

5.8.5 接口非功能属性

5.8.5.1 性能约束

- **前端实时分析接口**：`analyze()` 方法执行时间 < 16ms（满足 60fps）
- **REST API 响应时间**：P95 < 200ms，P99 < 500ms
- **AI 接口超时**：30 秒，重试 2 次

5.8.5.2 可靠性约束

- **接口可用性**：REST API 可用性 ≥ 99.5%
- **错误处理**：所有接口返回标准错误格式 `{ error: string, message?: string }`
- **降级策略**：AI 接口失败时自动降级使用规则引擎

5.8.5.3 安全性约束

- **认证接口**：密码传输使用 HTTPS，存储使用 SHA-256 哈希
- **数据脱敏**：AI 接口调用前移除用户 PII
- **CORS 策略**：前端仅允许从 `http://localhost:3000` 访问后端

5.9 结构视角 (Structure Viewpoint)

结构视角描述系统的静态组织结构，包括模块的层次结构、组件间的组合关系及数据结构的组织方式。本视角通过包图、组件图及数据结构图，揭示系统的静态架构骨架。

5.9.1 系统层次结构

FitnessAI 系统采用**分层架构**，严格遵循**关注点分离**原则。系统从下至上分为四层：

5.9.1.1 表示层（Presentation Layer）

职责：用户界面渲染与交互事件处理。

组件：

- **App Container**：React 应用根容器，管理全局状态（AuthContext）
- **CameraView**：摄像头视图组件，集成视频流与 Canvas 覆盖层
- **StatsPanel**：实时统计面板，显示计数、得分、反馈
- **ExerciseSelector**：运动类型选择器
- **Profile**：用户资料管理界面

设计模式：组件化、状态提升（Lifting State Up）

5.9.1.2 业务逻辑层（Business Logic Layer）

职责：核心业务规则实现，不依赖 UI 或数据存储细节。

前端业务逻辑：

- **usePoseDetection Hook**：姿态检测生命周期管理
- **ExerciseAnalyzer**：运动分析策略（SquatAnalyzer, PushupAnalyzer 等）
- **Telemetry Buffer**：数据缓冲与批量提交

后端业务逻辑：

- **SessionService**：会话业务编排

- **PoseAnalysisEngine**：姿态分析引擎
- **AdvisorAgent**：AI 代理服务

设计模式：策略模式、工厂模式、服务层模式

5.9.1.3 数据访问层（Data Access Layer）

职责：封装数据持久化逻辑，提供领域对象与数据存储的映射。

组件：

- **SessionRepository**：会话数据访问
- **UserRepository**：用户数据访问
- **PlanRepository**：健身计划数据访问

设计模式：仓储模式（Repository Pattern）

5.9.1.4 基础设施层（Infrastructure Layer）

职责：提供技术能力支撑，包括外部库适配、网络通信、数据存储。

组件：

- **MediaPipeAdapter**：MediaPipe 库适配器
- **ApiClient**：HTTP 客户端封装（Axios/Fetch）
- **DatabaseConnection**：PostgreSQL 连接池管理

设计模式：适配器模式、外观模式

5.9.2 前端模块结构

前端采用 **React + TypeScript** 架构，按功能域划分为以下模块：

5.9.2.1 核心模块（Core Modules）

```
frontend/src/
├── components/          # UI 组件
|   ├── CameraView.tsx
|   ├── ExerciseSelector.tsx
|   ├── StatsPanel.tsx
|   └── Profile.tsx
├── hooks/               # React Hooks
|   └── usePoseDetection.ts
├── utils/               # 工具函数
|   └── exerciseAnalyzer.ts
├── services/            # API 服务
|   └── api.ts
└── contexts/            # Context 提供者
    └── AuthContext.tsx
```

模块依赖关系：

- `components` → `hooks` → `utils` → `services`

- `components` → `contexts`（全局状态注入）

5.9.2.2 数据流结构

前端采用**单向数据流**（Unidirectional Data Flow）：

1. **用户交互** → 触发事件处理器
2. **事件处理器** → 调用 Hook 方法
3. **Hook** → 更新本地状态
4. **状态变更** → 触发组件重渲染
5. **组件** → 展示最新 UI

5.9.3 后端模块结构

后端采用 **Flask + Python** 架构，按分层原则组织：

5.9.3.1 模块划分

```
backend/
├── app.py              # Flask 应用入口，路由定义
├── pose_analyzer.py    # 姿态分析引擎
├── repositories/       # 数据访问层（未来扩展）
├── services/           # 业务服务层（未来扩展）
└── models/             # 领域模型（未来扩展）
```

当前实现：采用**扁平化结构**，所有逻辑集中在 `app.py`，符合小型项目快速迭代需求。未来可重构为分层结构。

5.9.4 数据结构组织

5.9.4.1 领域对象结构

User（用户聚合根）：

```
interface User {
  user_id: string;
  username: string;
  password_hash: string;
  email: string;
  nickname: string;
  profile: {
    height: number;    // cm
    weight: number;    // kg
    age: number;
    gender: 'male' | 'female' | 'other';
  };
  created_at: string;  // ISO 8601
}
```

Session（训练会话）：

```
interface Session {
  session_id: string;
  user_id: string;
  exercise_type: string;
  start_time: string;
  end_time: string | null;
  total_count: number;
  correct_count: number;
  scores: Array<{
    timestamp: string;
    score: number;
    is_correct: boolean;
    feedback: string;
    pose_data?: Landmark[];
  }>;
  status: 'active' | 'completed';
}
```

ExerciseAnalysis（分析结果）：

```
interface ExerciseAnalysis {
  isCorrect: boolean;
  score: number;        // 0-100
  feedback: string;
  count?: number;
  duration?: number;    // 秒
}
```

5.9.5 结构图

```
@startuml
package "表示层 (Presentation)" {
  [App Container]
  [CameraView]
  [StatsPanel]
  [ExerciseSelector]
  [Profile]
}

package "业务逻辑层 (Business Logic)" {
  [usePoseDetection Hook]
  [ExerciseAnalyzer]
  [SquatAnalyzer]
  [PushupAnalyzer]
  [PlankAnalyzer]
  [Telemetry Buffer]
  [SessionService]
  [AdvisorAgent]
}

package "数据访问层 (Data Access)" {
  [SessionRepository]
```

```
  [UserRepository]
  [PlanRepository]
}

package "基础设施层 (Infrastructure)" {
  [MediaPipeAdapter]
  [ApiClient]
  [DatabaseConnection]
}

[App Container] --> [usePoseDetection Hook]
[CameraView] --> [usePoseDetection Hook]
[StatsPanel] --> [usePoseDetection Hook]
[usePoseDetection Hook] --> [ExerciseAnalyzer]
[ExerciseAnalyzer] <|-- [SquatAnalyzer]
[ExerciseAnalyzer] <|-- [PushupAnalyzer]
[ExerciseAnalyzer] <|-- [PlankAnalyzer]
[usePoseDetection Hook] --> [Telemetry Buffer]
[Telemetry Buffer] --> [ApiClient]
[ApiClient] --> [SessionService]
[SessionService] --> [SessionRepository]
[SessionService] --> [AdvisorAgent]
[SessionRepository] --> [DatabaseConnection]
[MediaPipeAdapter] --> [ExerciseAnalyzer]

@enduml
```

图5-17：系统分层结构图

```
@startuml
package "前端模块结构" {
  package "components" {
    [CameraView]
    [StatsPanel]
    [ExerciseSelector]
    [Profile]
  }

  package "hooks" {
    [usePoseDetection]
  }

  package "utils" {
    [exerciseAnalyzer]
    [SquatAnalyzer]
    [PushupAnalyzer]
  }

  package "services" {
    [api]
  }

  package "contexts" {
```

```
    [AuthContext]
  }
}

[CameraView] --> [usePoseDetection]
[StatsPanel] --> [usePoseDetection]
[usePoseDetection] --> [exerciseAnalyzer]
[exerciseAnalyzer] --> [SquatAnalyzer]
[exerciseAnalyzer] --> [PushupAnalyzer]
[usePoseDetection] --> [api]
[CameraView] --> [AuthContext]

@enduml
```

图5-18：前端模块结构图

5.10 交互视角 (Interaction Viewpoint)

交互视角描述系统组件之间的动态协作关系，通过时序图、协作图展示对象/组件在特定场景下的消息传递与调用序列。本视角重点关注关键业务流程中的组件交互模式。

5.10.1 核心交互场景

5.10.1.1 场景一：实时姿态检测闭环

场景描述：用户启动摄像头，系统实时检测姿态并给出反馈，整个过程无需后端参与。

参与者：

- User（用户）
- CameraView（视图组件）
- usePoseDetection（Hook）
- MediaPipeAdapter（适配器）
- ExerciseAnalyzer（分析器）
- Canvas（渲染层）

交互序列：

```
@startuml
actor User
participant "CameraView" as CV
participant "usePoseDetection" as Hook
participant "MediaPipeAdapter" as MP
participant "ExerciseAnalyzer" as Analyzer
participant "Canvas" as Canvas

User -> CV: 点击"开始检测"
CV -> Hook: startDetection()
activate Hook
```

```
Hook -> MP: initialize()
MP --> Hook: Pose instance ready
Hook -> MP: start camera stream
MP -> MP: capture frame (30fps)
loop 每帧处理
  MP -> MP: detect pose landmarks
  MP -> Hook: onResults(landmarks)
  Hook -> Analyzer: analyze(landmarks)
  activate Analyzer
  Analyzer -> Analyzer: calculate angles
  Analyzer -> Analyzer: detect state transition
  Analyzer -> Analyzer: update count/score
  Analyzer --> Hook: ExerciseAnalysis
  deactivate Analyzer
  Hook -> Canvas: drawPose(landmarks)
  Hook -> CV: update exerciseStats
  CV -> User: 显示反馈（<100ms延迟）
end
User -> CV: 点击"停止检测"
CV -> Hook: stopDetection()
Hook -> MP: stop camera
deactivate Hook
@enduml
```

图5-19：实时姿态检测交互时序图

关键交互点：

1. **初始化阶段**：Hook 创建 MediaPipe 实例，配置检测参数
2. **实时处理循环**：30fps 帧率，每帧执行检测→分析→渲染→反馈
3. **低延迟保证**：整个闭环在浏览器内完成，延迟 < 100ms

5.10.1.2 场景二：训练会话生命周期

场景描述：用户开始训练，系统创建会话，持续提交数据，最后结束会话并生成报告。

参与者：

- Frontend（前端）
- SessionController（控制器）
- SessionService（服务层）
- SessionRepository（仓储层）
- Database（数据库）
- AdvisorAgent（AI 代理）

交互序列：

```
@startuml
participant "Frontend" as FE
participant "SessionController" as SC
participant "SessionService" as SS
```

```
participant "SessionRepository" as SR
database "PostgreSQL" as DB
participant "AdvisorAgent" as Agent
participant "Zhipu AI" as AI

== 开始会话 ==
FE -> SC: POST /api/session/start\n{exercise_type, user_id}
activate SC
SC -> SS: createSession(exercise_type, user_id)
activate SS
SS -> SS: generate session_id
SS -> SR: save(session)
activate SR
SR -> DB: INSERT INTO sessions
DB --> SR: session_id
SR --> SS: Session entity
SS --> SC: {session_id, message}
SC --> FE: 200 OK
deactivate SC
deactivate SS
deactivate SR

== 持续提交数据 ==
loop 每批次数据
  FE -> SC: POST /api/session/{id}/data\n{pose_data, score, feedback}
  SC -> SS: submitData(session_id, data)
  SS -> SR: updateSession(session_id, data)
  SR -> DB: UPDATE sessions SET scores = ...
  DB --> SR: updated
  SR --> SS: success
  SS --> SC: {message, session_stats}
  SC --> FE: 200 OK
end

== 结束会话 ==
FE -> SC: POST /api/session/{id}/end
SC -> SS: endSession(session_id)
SS -> SR: findById(session_id)
SR -> DB: SELECT * FROM sessions WHERE id = ...
DB --> SR: session data
SR --> SS: Session entity
SS -> Agent: generateSummary(session_stats)
activate Agent
Agent -> AI: POST /api/paas/v4/chat/completions\n{prompt}
AI --> Agent: {choices[0].message.content}
Agent -> Agent: parse AI response
Agent --> SS: AI feedback text
deactivate Agent
SS -> SR: updateSession(session_id, {end_time, ai_feedback})
SR -> DB: UPDATE sessions SET ...
SS -> SS: calculate summary statistics
SS --> SC: {session_id, summary}
SC --> FE: 200 OK
```

```
deactivate SC
deactivate SS
deactivate SR
@enduml
```

图5-20：训练会话生命周期交互时序图

关键交互点：

1. **会话创建**：前端发起请求，后端生成唯一 session_id
2. **数据提交**：Telemetry Buffer 批量提交，减少 HTTP 请求频率
3. **会话结束**：触发 AI 总结生成，异步调用外部服务

5.10.1.3 场景三：AI 健身计划生成

场景描述：用户请求 AI 生成个性化健身计划，系统调用智谱 AI API，解析返回结果。

参与者：

- Frontend（前端）
- AIController（AI 控制器）
- AdvisorAgent（AI 代理）
- ZhipuAI（外部服务）
- PlanRepository（计划仓储）

交互序列：

```
@startuml
participant "Frontend" as FE
participant "AIController" as AC
participant "AdvisorAgent" as Agent
participant "Zhipu AI API" as API
participant "PlanRepository" as PR
database "PostgreSQL" as DB

FE -> AC: POST /api/ai/generate-plan\n{height?, weight?, age?, gender?}
activate AC
AC -> AC: load user profile
AC -> Agent: generatePlan(height, weight, age, gender)
activate Agent

Agent -> Agent: calculate BMI
Agent -> Agent: determine fitness_level
Agent -> Agent: build prompt

alt API Key 已配置
  Agent -> API: POST /api/paas/v4/chat/completions\n{model: "glm-4", messages: [...]}
  activate API
  API --> Agent: {choices[0].message.content}
  deactivate API
  Agent -> Agent: parse_ai_response(text)
```

```
  Agent -> Agent: extract daily_goals, weekly_goals
  Agent --> AC: {daily_goals, weekly_goals, suggestions, ai_used: true}
else API Key 未配置或调用失败
  Agent -> Agent: use rule-based engine
  Agent -> Agent: generate plan by rules
  Agent --> AC: {daily_goals, weekly_goals, suggestions, ai_used: false}
end

deactivate Agent
AC -> PR: savePlan(user_id, plan)
activate PR
PR -> DB: INSERT/UPDATE training_plans
DB --> PR: success
PR --> AC: saved
deactivate PR
AC --> FE: 200 OK {plan data}
deactivate AC
@enduml
```

图5-21：AI 健身计划生成交互时序图

关键交互点：

1. **降级机制**：API 调用失败时自动使用规则引擎
2. **数据解析**：使用正则表达式提取结构化数据
3. **结果持久化**：生成的计划保存到数据库

5.10.2 组件协作模式

5.10.2.1 前端组件协作

前端采用**观察者模式**实现组件间通信：

- **usePoseDetection Hook** 作为**被观察者**（Subject），维护 `exerciseStats` 状态
- **StatsPanel** 作为**观察者**（Observer），订阅状态变更并自动更新 UI

5.10.2.2 后端服务协作

后端采用**服务层模式**实现业务编排：

- **Controller** 负责 HTTP 协议解析，调用 **Service** 执行业务逻辑
- **Service** 协调多个 **Repository** 和 **Agent**，实现复杂业务流程
- **Repository** 封装数据访问细节，向 Service 暴露领域对象接口

5.10.3 交互图

```
@startuml
package "前端交互" {
  [CameraView] --> [usePoseDetection]
  [usePoseDetection] --> [MediaPipeAdapter]
  [usePoseDetection] --> [ExerciseAnalyzer]
```

```
  [StatsPanel] --> [usePoseDetection]
  [TelemetryBuffer] --> [ApiClient]
}

package "后端交互" {
  [SessionController] --> [SessionService]
  [SessionService] --> [SessionRepository]
  [SessionService] --> [AdvisorAgent]
  [AdvisorAgent] --> [ZhipuAI]
  [AIController] --> [AdvisorAgent]
}

package "跨层交互" {
  [ApiClient] --> [SessionController]
  [ApiClient] --> [AIController]
}

@enduml
```

图5-22：组件协作关系图

5.11 状态动态视角 (State Dynamics Viewpoint)

状态动态视角描述系统及其组件在生命周期内的状态变迁，通过状态图、状态转换表揭示系统的动态行为模式。本视角重点关注关键实体的状态机模型。

5.11.1 训练会话状态机

Session（训练会话）是系统的核心状态实体，其生命周期包含以下状态：

5.11.1.1 状态定义

1. **INITIALIZED（已初始化）**：会话对象已创建，但尚未开始数据采集
2. **ACTIVE（活跃）**：会话正在进行，持续接收运动数据
3. **PAUSED（暂停）**：会话临时暂停，数据采集中断但会话未结束
4. **COMPLETED（已完成）**：会话正常结束，数据已保存，AI 总结已生成
5. **CANCELLED（已取消）**：会话异常终止，数据可能未完整保存

5.11.1.2 状态转换规则

当前状态	触发事件	下一状态	动作
INITIALIZED	start_session()	ACTIVE	设置 start_time，初始化计数器
ACTIVE	submit_data()	ACTIVE	更新 scores 数组，累计计数
ACTIVE	pause()	PAUSED	暂停数据采集
PAUSED	resume()	ACTIVE	恢复数据采集

当前状态	触发事件	下一状态	动作
ACTIVE	end_session()	COMPLETED	设置 end_time，调用 AI 生成总结
PAUSED	end_session()	COMPLETED	设置 end_time，调用 AI 生成总结
ACTIVE	cancel()	CANCELLED	标记为取消，不生成总结
PAUSED	cancel()	CANCELLED	标记为取消
INITIALIZED	cancel()	CANCELLED	删除会话记录

5.11.1.3 状态图

```
@startuml
[*] --> INITIALIZED : createSession()

INITIALIZED --> ACTIVE : start_session()
INITIALIZED --> CANCELLED : cancel()

ACTIVE --> ACTIVE : submit_data()\n[更新数据]
ACTIVE --> PAUSED : pause()
ACTIVE --> COMPLETED : end_session()\n[生成AI总结]
ACTIVE --> CANCELLED : cancel()

PAUSED --> ACTIVE : resume()
PAUSED --> COMPLETED : end_session()\n[生成AI总结]
PAUSED --> CANCELLED : cancel()

COMPLETED --> [*]
CANCELLED --> [*]

state ACTIVE {
  [*] --> CollectingData
  CollectingData : total_count++
  CollectingData : correct_count++
  CollectingData : scores.append()
}

state COMPLETED {
  [*] --> GeneratingSummary
  GeneratingSummary : calculate statistics
  GeneratingSummary : call AI agent
  GeneratingSummary --> SummaryReady
  SummaryReady : save to database
}

@enduml
```

图5-23：训练会话状态机图

5.11.2 姿态检测状态机

usePoseDetection Hook 管理前端姿态检测的生命周期状态：

5.11.2.1 状态定义

1. **IDLE（空闲）**： Hook 已初始化，但未启动检测
2. **INITIALIZING（初始化中）**： 正在加载 MediaPipe 模型和启动摄像头
3. **DETECTING（检测中）**： 摄像头已启动，持续进行姿态检测
4. **ERROR（错误）**： 检测过程中发生错误（如摄像头权限被拒绝）

5.11.2.2 状态转换规则

当前状态	触发事件	下一状态	动作
IDLE	startDetection()	INITIALIZING	加载 MediaPipe，请求摄像头权限
INITIALIZING	onMediaPipeReady()	DETECTING	启动摄像头流，开始检测循环
INITIALIZING	onError()	ERROR	设置错误消息
DETECTING	stopDetection()	IDLE	停止摄像头，释放资源
DETECTING	onCameraError()	ERROR	设置错误消息
ERROR	retry()	INITIALIZING	重新初始化
ERROR	reset()	IDLE	重置状态

5.11.2.3 状态图

```
@startuml
[*] --> IDLE : Hook mounted

IDLE --> INITIALIZING : startDetection()
INITIALIZING --> DETECTING : MediaPipe ready\nCamera started
INITIALIZING --> ERROR : Permission denied\nMediaPipe load failed

DETECTING --> DETECTING : onFrame()\n[检测循环 30fps]
DETECTING --> IDLE : stopDetection()
DETECTING --> ERROR : Camera error

ERROR --> INITIALIZING : retry()
ERROR --> IDLE : reset()

IDLE --> [*] : Hook unmounted

state DETECTING {
  [*] --> CapturingFrame
  CapturingFrame --> ProcessingPose : MediaPipe.detect()
  ProcessingPose --> AnalyzingExercise : ExerciseAnalyzer.analyze()
```

```
  AnalyzingExercise --> RenderingFeedback : update UI
  RenderingFeedback --> CapturingFrame : next frame
}

@enduml
```

图5-24：姿态检测状态机图

5.11.3 运动分析器状态机

SquatAnalyzer（深蹲分析器）的状态机模型，用于跟踪单个深蹲动作的状态：

5.11.3.1 状态定义

1. **STANDING（站立）**：用户处于站立姿态，膝盖角度 > 160°
2. **DESCENDING（下蹲中）**：用户正在下蹲，膝盖角度从 160° 减小到 90°
3. **BOTTOM（底部）**：用户蹲到最低点，膝盖角度 < 90°
4. **ASCENDING（起立中）**：用户正在站起，膝盖角度从 90° 增大到 160°

5.11.3.2 状态转换规则

当前状态	条件	下一状态	动作
STANDING	knee_angle < 160°	DESCENDING	开始下蹲
DESCENDING	knee_angle < 90°	BOTTOM	到达底部，记录时间戳
BOTTOM	knee_angle > 90°	ASCENDING	开始起立
ASCENDING	knee_angle > 160°	STANDING	完成一次，count++
*	knee_angle 异常	STANDING	重置状态

5.11.3.3 状态图

```
@startuml
[*] --> STANDING : reset()

STANDING --> DESCENDING : knee_angle < 160°\n[开始下蹲]
DESCENDING --> BOTTOM : knee_angle < 90°\n[到达底部]
BOTTOM --> ASCENDING : knee_angle > 90°\n[开始起立]
ASCENDING --> STANDING : knee_angle > 160°\n[完成一次，count++]

STANDING --> STANDING : knee_angle > 160°\n[保持站立]
DESCENDING --> DESCENDING : 90° < knee_angle < 160°\n[继续下蹲]
BOTTOM --> BOTTOM : knee_angle < 90°\n[保持底部]
ASCENDING --> ASCENDING : 90° < knee_angle < 160°\n[继续起立]

state BOTTOM {
  [*] --> ValidatingForm
  ValidatingForm : 检查背部姿态
```

```
  ValidatingForm : 检查平衡
  ValidatingForm --> CountingRep : 姿势标准
  ValidatingForm --> InvalidForm : 姿势不标准
  InvalidForm : score -= 10
  InvalidForm --> CountingRep
  CountingRep : count++
}

@enduml
```

图5-25：深蹲分析器状态机图

5.11.4 用户认证状态机

AuthContext 管理用户认证状态：

5.11.4.1 状态定义

1. **UNAUTHENTICATED（未认证）**：用户未登录
2. **AUTHENTICATING（认证中）**：正在验证用户凭证
3. **AUTHENTICATED（已认证）**：用户已登录，Token 有效
4. **TOKEN_EXPIRED（Token 过期）**：Token 已过期，需要重新登录

5.11.4.2 状态转换规则

当前状态	触发事件	下一状态	动作
UNAUTHENTICATED	login(username, password)	AUTHENTICATING	发送登录请求
AUTHENTICATING	onSuccess(token)	AUTHENTICATED	保存 Token，设置用户信息
AUTHENTICATING	onError()	UNAUTHENTICATED	显示错误消息
AUTHENTICATED	logout()	UNAUTHENTICATED	清除 Token
AUTHENTICATED	tokenExpired()	TOKEN_EXPIRED	清除 Token，提示重新登录
TOKEN_EXPIRED	login()	AUTHENTICATED	重新认证

5.11.4.3 状态图

```
@startuml
[*] --> UNAUTHENTICATED : App start

UNAUTHENTICATED --> AUTHENTICATING : login(username, password)
AUTHENTICATING --> AUTHENTICATED : API success\nsave token
AUTHENTICATING --> UNAUTHENTICATED : API error
```

```
AUTHENTICATED --> UNAUTHENTICATED : logout()
AUTHENTICATED --> TOKEN_EXPIRED : token expired\n(24h)

TOKEN_EXPIRED --> AUTHENTICATING : login()
TOKEN_EXPIRED --> UNAUTHENTICATED : user cancels

AUTHENTICATED --> [*] : App close
UNAUTHENTICATED --> [*] : App close

state AUTHENTICATED {
  [*] --> ValidToken
  ValidToken : user_id
  ValidToken : username
  ValidToken : profile
  ValidToken --> MakingRequest : API call
  MakingRequest --> ValidToken : success
}

@enduml
```

图5-26：用户认证状态机图

5.12 算法视角 (Algorithm Viewpoint)

算法视角详细描述系统中关键算法的设计思路、计算步骤及复杂度分析。本视角重点关注姿态分析算法、评分算法及 AI 响应解析算法。

5.12.1 姿态关键点角度计算算法

5.12.1.1 算法描述

功能：计算三个关键点之间的夹角（度），用于判断关节弯曲程度。

输入：

- `point1: Landmark` - 第一个关键点（如髋部）
- `point2: Landmark` - 第二个关键点（如膝盖，顶点）
- `point3: Landmark` - 第三个关键点（如脚踝）

输出：角度值（0-180°）

5.12.1.2 算法步骤

```
function calculateAngle(point1: Landmark, point2: Landmark, point3: Landmark): number {
  // 步骤1：计算向量
  const vector1 = {
    x: point1.x - point2.x,
    y: point1.y - point2.y
  };
  const vector2 = {
```

```
    x: point3.x - point2.x,
    y: point3.y - point2.y
  };

  // 步骤2：计算向量夹角（弧度）
  const radians = Math.atan2(vector2.y, vector2.x) - Math.atan2(vector1.y, vector1.x);

  // 步骤3：转换为角度
  let angle = Math.abs((radians * 180.0) / Math.PI);

  // 步骤4：规范化到 [0, 180] 范围
  if (angle > 180.0) {
    angle = 360 - angle;
  }

  return angle;
}
```

时间复杂度：O(1) - 常数时间，仅涉及基本数学运算

空间复杂度：O(1) - 仅使用固定数量的临时变量

5.12.1.3 算法图

```
@startuml
start
:输入: point1, point2, point3;
:计算向量 vector1 = point1 - point2;
:计算向量 vector2 = point3 - point2;
:计算夹角弧度\nradians = atan2(v2) - atan2(v1);
:转换为角度\nangle = radians * 180 / π;
if (angle > 180°) then (yes)
  :angle = 360 - angle;
else (no)
endif
:返回 angle;
stop
@enduml
```

图5-27：角度计算算法流程图

5.12.2 深蹲动作检测算法

5.12.2.1 算法描述

功能：基于膝盖角度检测深蹲动作，实现计数与质量评估。

输入：`landmarks: Landmark[]` - MediaPipe 33 个关键点数组

输出：`ExerciseAnalysis` - 包含计数、得分、反馈的分析结果

5.12.2.2 算法步骤

```
function analyzeSquat(landmarks: Landmark[]): ExerciseAnalysis {
  // 步骤1：提取关键点
  const leftHip = landmarks[23];
  const leftKnee = landmarks[25];
  const leftAnkle = landmarks[27];
  const rightHip = landmarks[24];
  const rightKnee = landmarks[26];
  const rightAnkle = landmarks[28];

  // 步骤2：计算左右膝盖角度
  const leftKneeAngle = calculateAngle(leftHip, leftKnee, leftAnkle);
  const rightKneeAngle = calculateAngle(rightHip, rightKnee, rightAnkle);
  const avgKneeAngle = (leftKneeAngle + rightKneeAngle) / 2;

  // 步骤3：状态检测与计数
  let count = this.squatCount;
  let feedback = '';
  let score = 50;

  if (avgKneeAngle < 90) {
    // 深蹲到底部
    if (!this.inSquatPosition) {
      this.inSquatPosition = true;
      feedback = '很好！保持深蹲姿势';
      score = 85;
    }
  } else if (avgKneeAngle > 160) {
    // 站立位置
    if (this.inSquatPosition) {
      // 完成一次深蹲
      this.squatCount++;
      this.inSquatPosition = false;
      feedback = `完成！已做 ${this.squatCount} 次深蹲`;
      score = 100;
    }
  }

  // 步骤4：平衡检测
  const hipDistance = calculateDistance(leftHip, rightHip);
  const ankleDistance = calculateDistance(leftAnkle, rightAnkle);
  const balanceRatio = Math.abs(hipDistance - ankleDistance) / hipDistance;

  if (balanceRatio > 0.3) {
    feedback += '，注意保持平衡';
    score = Math.max(0, score - 10);
  }

  // 步骤5：返回结果
  return {
    isCorrect: score >= 70,
```

```
    score: Math.min(100, Math.max(0, score)),
    feedback,
    count: this.squatCount
  };
}
```

时间复杂度：O(1) - 固定数量的关键点计算

空间复杂度：O(1) - 仅使用固定数量的状态变量

5.12.2.3 算法图

```
@startuml
start
:输入: landmarks[33];
:提取关键点\n(hip, knee, ankle);
:计算左右膝盖角度;
:avgKneeAngle = (left + right) / 2;
if (avgKneeAngle < 90°) then (是)
  if (inSquatPosition == false) then (否)
    :inSquatPosition = true;
    :feedback = "很好! 保持深蹲姿势";
    :score = 85;
  else (是)
    :保持底部姿势;
  endif
elseif (avgKneeAngle > 160°) then (是)
  if (inSquatPosition == true) then (是)
    :squatCount++;
    :inSquatPosition = false;
    :feedback = "完成! 已做 N 次";
    :score = 100;
  else (否)
    :保持站立;
  endif
else (中间状态)
  :反馈中间状态;
endif
:计算平衡度;
if (balanceRatio > 0.3) then (不平衡)
  :score -= 10;
  :feedback += "注意保持平衡";
endif
:返回 ExerciseAnalysis;
stop
@enduml
```

图5-28：深蹲检测算法流程图

5.12.3 AI 响应解析算法

5.12.3.1 算法描述

功能：从智谱 AI 返回的自然语言文本中提取结构化的健身计划数据。

输入：`ai_text: string` - AI 返回的文本响应

输出：`{ daily_goals, weekly_goals, suggestions }` - 解析后的结构化数据

5.12.3.2 算法步骤

```
def parse_ai_response(ai_text: str, height: int, weight: int, age: int, gender: str) ->
dict:
    # 步骤1：初始化默认值
    daily_goals = {
        "squat": 20,
        "pushup": 15,
        "plank": 60,
        "jumping_jack": 30
    }
    weekly_goals = {
        "total_sessions": 5,
        "total_duration": 150
    }
    suggestions = []

    # 步骤2：使用正则表达式匹配深蹲次数
    squat_patterns = [
        r'深蹲[：:].*?每组\s*(\d+)\s*次',
        r'深蹲[：:].*?(\d+)\s*次(?!组)',
        r'深蹲[：:].*?(\d+)\s*组',
    ]
    for pattern in squat_patterns:
        match = re.search(pattern, ai_text, re.IGNORECASE)
        if match:
            value = int(match.group(1))
            if value < 10:  # 可能是组数
                each_match = re.search(r'深蹲[：:].*?每组\s*(\d+)\s*次', ai_text,
re.IGNORECASE)
                if each_match:
                    value = int(each_match.group(1)) * value
            daily_goals["squat"] = value
            break

    # 步骤3：类似地匹配其他运动类型
    # ... (俯卧撑、平板支撑、开合跳)

    # 步骤4：匹配每周目标
    sessions_match = re.search(r'总运动次数[：:]\s*(\d+)', ai_text, re.IGNORECASE)
    if sessions_match:
        weekly_goals["total_sessions"] = int(sessions_match.group(1))
```

```
    # 步骤5：提取建议（按段落分割）
    lines = [line.strip() for line in ai_text.split('\n') if line.strip()]
    for line in lines:
        if len(line) > 20 and not re.match(r'^[#*\-•\d\s]+$', line):
            suggestions.append(line)

    suggestions = suggestions[:5]  # 最多5条建议

    return {
        "daily_goals": daily_goals,
        "weekly_goals": weekly_goals,
        "suggestions": suggestions
    }
```

时间复杂度：O(n) - n 为文本长度，正则匹配为线性时间

空间复杂度：O(n) - 存储解析后的文本行

5.12.3.3 算法图

```
@startuml
start
:输入: ai_text (自然语言);
:初始化默认值\ndaily_goals, weekly_goals;
:使用正则表达式匹配深蹲次数;
if (匹配成功?) then (是)
  :提取数值;
  if (值 < 10?) then (是，可能是组数)
    :查找"每组X次";
    :计算总次数 = 组数 × 每组次数;
  endif
  :daily_goals["squat"] = 值;
endif
:类似匹配其他运动类型\n(俯卧撑、平板支撑、开合跳);
:匹配每周运动次数;
:匹配每周运动时长;
:按行分割文本;
:过滤标题和数字行;
:提取建议文本;
:限制建议数量 ≤ 5;
:返回结构化数据;
stop
@enduml
```

图5-29：AI 响应解析算法流程图

5.12.4 评分算法

5.12.4.1 算法描述

功能：基于多个因素计算动作质量得分（0-100）。

输入：

- `knee_angle: number` - 膝盖角度
- `balance_ratio: number` - 平衡度比率
- `form_correctness: boolean` - 姿态正确性

输出：得分（0-100）

5.12.4.2 算法步骤

```
function calculateScore(
  kneeAngle: number,
  balanceRatio: number,
  formCorrectness: boolean
): number {
  let baseScore = 100;

  // 步骤1：角度评分（深蹲标准角度为90°）
  const angleDeviation = Math.abs(kneeAngle - 90);
  if (angleDeviation > 0) {
    baseScore -= angleDeviation * 2;  // 每度偏差扣2分
  }

  // 步骤2：平衡度评分
  if (balanceRatio > 0.3) {
    baseScore -= 10;  // 不平衡扣10分
  } else if (balanceRatio > 0.2) {
    baseScore -= 5;   // 轻微不平衡扣5分
  }

  // 步骤3：姿态正确性评分
  if (!formCorrectness) {
    baseScore -= 15;  // 姿态不正确扣15分
  }

  // 步骤4：规范化到 [0, 100] 范围
  return Math.min(100, Math.max(0, Math.round(baseScore)));
}
```

时间复杂度：O(1)

空间复杂度：O(1)

5.12.4.3 算法图

```
@startuml
start
:输入: kneeAngle, balanceRatio, formCorrectness;
:baseScore = 100;
:计算角度偏差\nangleDeviation = |kneeAngle - 90|;
:baseScore -= angleDeviation × 2;
if (balanceRatio > 0.3) then (严重不平衡)
  :baseScore -= 10;
elseif (balanceRatio > 0.2) then (轻微不平衡)
  :baseScore -= 5;
endif
if (formCorrectness == false) then (姿态不正确)
  :baseScore -= 15;
endif
:score = max(0, min(100, round(baseScore)));
:返回 score;
stop
@enduml
```

图5-30：评分算法流程图

5.13 资源视角 (Resource Viewpoint)

资源视角描述系统对计算资源、存储资源及网络资源的占用与分配策略。本视角关注性能优化、资源限制及可扩展性设计。

5.13.1 计算资源

5.13.1.1 前端计算资源

CPU 资源占用：

- **MediaPipe 推理**：每帧约 10-15ms CPU 时间（30fps 下）
- **姿态分析计算**：每帧约 1-2ms CPU 时间（角度计算、状态检测）
- **Canvas 渲染**：每帧约 2-3ms CPU 时间（绘制骨骼连线）

总 CPU 占用：约 15-20ms/帧，满足 60fps 要求（16.67ms/帧）

GPU 资源占用：

- **MediaPipe WASM**：利用 WebGL 加速，GPU 占用约 20-30%
- **Canvas 2D 渲染**：CPU 渲染，不占用 GPU

内存资源占用：

- **MediaPipe 模型**：约 5-8 MB（WASM 二进制）
- **React 应用**：约 10-15 MB（组件树、状态）
- **视频流缓冲**：约 2-3 MB（640×480 @ 30fps，1秒缓冲）

- **总内存占用**：约 20-30 MB

5.13.1.2 后端计算资源

CPU 资源占用：

- **Flask 请求处理**：平均 5-10ms/请求（简单 CRUD）
- **AI 代理调用**：异步处理，不阻塞主线程
- **姿态分析引擎**：TensorFlow 推理，单次约 50-100ms（如启用）

内存资源占用：

- **Flask 应用**：约 50-100 MB（Python 运行时）
- **TensorFlow 模型**：约 200-500 MB（如加载 `.h5` 模型文件）
- **会话数据缓存**：约 10-50 MB（内存中暂存活跃会话）

并发处理能力：

- **Flask 单进程**：约 50-100 并发请求（Gunicorn 多进程可扩展）
- **数据库连接池**：最大 20 个连接

5.13.2 存储资源

5.13.2.1 前端存储

LocalStorage：

- **用户 Token**：约 500 bytes
- **用户偏好设置**：约 1-2 KB
- **总占用**：< 5 KB

SessionStorage：

- **临时会话数据**：约 10-50 KB（训练过程中的中间数据）

5.13.2.2 后端存储

数据库存储（PostgreSQL）：

users 表：

- 单条记录：约 500 bytes
- 10,000 用户：约 5 MB

sessions 表：

- 单条记录：约 2-5 KB（包含 JSONB 遥测数据）
- 100,000 会话：约 200-500 MB

training_plans 表：

- 单条记录：约 1-2 KB（JSONB AI 计划）
- 10,000 计划：约 10-20 MB

总存储估算：

- **小规模**（1,000 用户，10,000 会话）：约 50-100 MB
- **中规模**（10,000 用户，100,000 会话）：约 500 MB - 1 GB
- **大规模**（100,000 用户，1,000,000 会话）：约 5-10 GB

文件存储：

- **日志文件**：每日约 10-50 MB（按日志级别）
- **模型文件**（如使用）：约 200-500 MB（`.h5` 格式）

5.13.3 网络资源

5.13.3.1 前端网络流量

初始加载：

- **HTML**：约 10 KB
- **CSS**：约 50-100 KB
- **JavaScript Bundle**：约 500 KB - 1 MB（压缩后）
- **MediaPipe WASM**：约 5-8 MB（CDN 加载）
- **总初始加载**：约 6-10 MB

运行时流量：

- **API 请求**：平均 1-5 KB/请求
- **数据提交频率**：批量提交，每 5-10 秒一次，约 10-50 KB/批次
- **总运行时流量**：约 100-500 KB/分钟（取决于使用频率）

5.13.3.2 后端网络流量

API 响应：

- **认证接口**：约 500 bytes - 1 KB
- **会话接口**：约 1-5 KB
- **AI 接口**：约 2-10 KB（AI 返回文本）

外部服务调用：

- **智谱 AI API**：请求约 1-2 KB，响应约 2-5 KB
- **调用频率**：每用户每次训练结束调用一次

5.13.4 资源分配策略

5.13.4.1 前端资源优化

代码分割（Code Splitting）：

- 按路由懒加载组件，减少初始 bundle 大小
- MediaPipe 库按需加载，不阻塞首屏渲染

资源缓存：

- MediaPipe WASM 文件使用 CDN 缓存，减少重复下载
- API 响应使用 HTTP 缓存头，减少重复请求

内存管理：

- 及时释放视频流资源（`stream.getTracks().forEach(track => track.stop())`）
- 使用 `useCallback` 和 `useMemo` 避免不必要的重渲染

5.13.4.2 后端资源优化

数据库优化：

- **索引策略**：在 `user_id`, `session_id`, `created_at` 上创建索引
- **JSONB 索引**：对 `telemetry_log` 字段创建 GIN 索引，加速 JSON 查询
- **连接池**：复用数据库连接，减少连接开销

缓存策略：

- **会话数据缓存**：活跃会话数据暂存内存，减少数据库查询
- **AI 响应缓存**：相同输入参数的 AI 请求缓存结果（可选）

异步处理：

- AI 代理调用使用异步任务队列，不阻塞主请求线程
- 数据归档使用后台任务，避免影响实时性能

5.13.5 资源限制与扩展性

5.13.5.1 前端资源限制

浏览器限制：

- **内存限制**：约 2-4 GB（取决于设备）
- **存储限制**：LocalStorage 约 5-10 MB
- **并发请求限制**：同域约 6 个并发连接

应对策略：

- 使用 Web Workers 分担计算密集型任务
- 数据批量提交，减少 HTTP 请求频率
- 使用 IndexedDB 存储大量历史数据（如需要）

5.13.5.2 后端资源限制

服务器限制：

- **单进程并发**：约 50-100 请求/秒
- **数据库连接**：最大 20 个连接（可配置）

扩展策略：

- **水平扩展**：使用 Gunicorn 多进程 + Nginx 负载均衡
- **数据库扩展**：读写分离，主从复制
- **缓存层**：引入 Redis 缓存热点数据

5.13.6 资源监控图

```
@startuml
package "前端资源监控" {
  [CPU Usage] : MediaPipe: 10-15ms/帧
  [CPU Usage] : Analysis: 1-2ms/帧
  [CPU Usage] : Rendering: 2-3ms/帧

  [Memory Usage] : MediaPipe: 5-8 MB
  [Memory Usage] : React App: 10-15 MB
  [Memory Usage] : Video Buffer: 2-3 MB

  [Network] : Initial Load: 6-10 MB
  [Network] : Runtime: 100-500 KB/min
}

package "后端资源监控" {
  [CPU Usage] : Flask: 5-10ms/req
  [CPU Usage] : TensorFlow: 50-100ms/inference

  [Memory Usage] : Flask: 50-100 MB
  [Memory Usage] : TensorFlow: 200-500 MB
  [Memory Usage] : Session Cache: 10-50 MB

  [Database] : Users: ~5 MB/10K
  [Database] : Sessions: ~500 MB/100K
  [Database] : Plans: ~20 MB/10K

  [Network] : API Response: 1-5 KB/req
  [Network] : AI Call: 2-5 KB/call
}

@enduml
```

图5-31：系统资源占用概览图

5.13.7 性能基准

5.13.7.1 前端性能指标

指标	目标值	实际值	备注
首屏加载时间	< 3s	~2-3s	取决于网络速度
姿态检测延迟	< 100ms	~50-80ms	浏览器内闭环
帧率	≥ 30fps	~30fps	稳定运行

指标	目标值	实际值	备注
内存占用	< 50 MB	~20-30 MB	正常使用

5.13.7.2 后端性能指标

指标	目标值	实际值	备注
API 响应时间 (P95)	< 200ms	~100-150ms	简单 CRUD
API 响应时间 (P99)	< 500ms	~200-300ms	包含数据库查询
AI 接口响应时间	< 30s	~5-15s	取决于网络
数据库查询时间	< 50ms	~20-40ms	带索引查询
并发处理能力	≥ 50 req/s	~50-100 req/s	单进程

以上内容涵盖了 IEEE 1016 标准要求的 5.8 至 5.13 六个视角，每个视角都包含了详细的文字描述和 UML 图代码（PlantUML 格式）。您可以将这些内容添加到您的 SDD 文档中，并使用 PlantUML 工具渲染图表。