

- **日历应用产品报告**
  - **一、产品功能介绍**
    - 1.1 核心功能
      - 主要功能模块：
    - 1.2 技术特性
  - **二、程序概要设计**
    - 2.1 系统架构概述
    - 2.2 模块划分
      - 2.2.1 表示层（Presentation Layer）
      - 2.2.2 状态管理层（State Management Layer）
      - 2.2.3 领域层（Domain Layer）
      - 2.2.4 数据层（Data Layer）
    - 2.3 数据流设计
    - 2.4 关键设计模式
  - **三、软件架构图**
    - 3.1 整体架构图
    - 3.2 数据模型关系图
    - 3.3 状态流转图
    - 3.4 服务交互时序图
  - **四、技术亮点及其实现原理**
    - 4.1 响应式状态管理（Riverpod）
    - 4.2 本地持久化存储（Hive）
    - 4.3 时区感知的本地通知系统
    - 4.4 RFC5545 标准兼容的事件模型
    - 4.5 高性能时间轴渲染（日视图）
    - 4.6 Material 3 设计系统集成
    - 4.7 国际化与本地化
    - 4.8 模块化架构设计
  - **五、技术栈总结**
    - 5.1 核心依赖
    - 5.2 开发工具
    - 5.3 平台支持
  - **六、总结**

# 日历应用产品报告

# 一、产品功能介绍

## 1.1 核心功能

**日历应用（Calendar App）** 是一款基于 Flutter 开发的轻量级日程管理应用，提供完整日程创建、查看、编辑和提醒功能。

**主要功能模块：**

### 1. 多视图日历展示

- **日视图**：精确到小时/分钟的时间轴视图，支持全天事件和定时事件分类显示
- **周视图**：一周七天的事件概览
- **月视图**：整月日历网格视图，快速浏览月度安排

### 2. 日程事件管理

- 创建、编辑、删除日程事件
- 支持事件标题、描述、地点、颜色标识
- 支持全天事件和定时事件
- 事件时间范围设置（开始时间、结束时间）
- 事件唯一标识（UUID）管理

### 3. 提醒通知系统

- 本地通知提醒（支持提前时间设置）
- 时区感知的精确提醒调度
- 通知内容包含事件标题、时间、地点、描述
- 支持多个提醒设置（每个事件可配置多个提醒）

### 4. 数据持久化

- 基于 Hive 的本地数据库存储
- 事件数据自动保存和恢复
- 支持应用重启后数据恢复

### 5. 国际化支持

- 中文（zh\_CN）日期格式显示
- 农历日期显示

- 周次计算与显示
- 节气、节日标识

## 6. 用户体验优化

- Material 3 设计规范
- 流畅的视图切换动画
- 响应式布局设计
- 测试通知功能（便于验证通知系统）

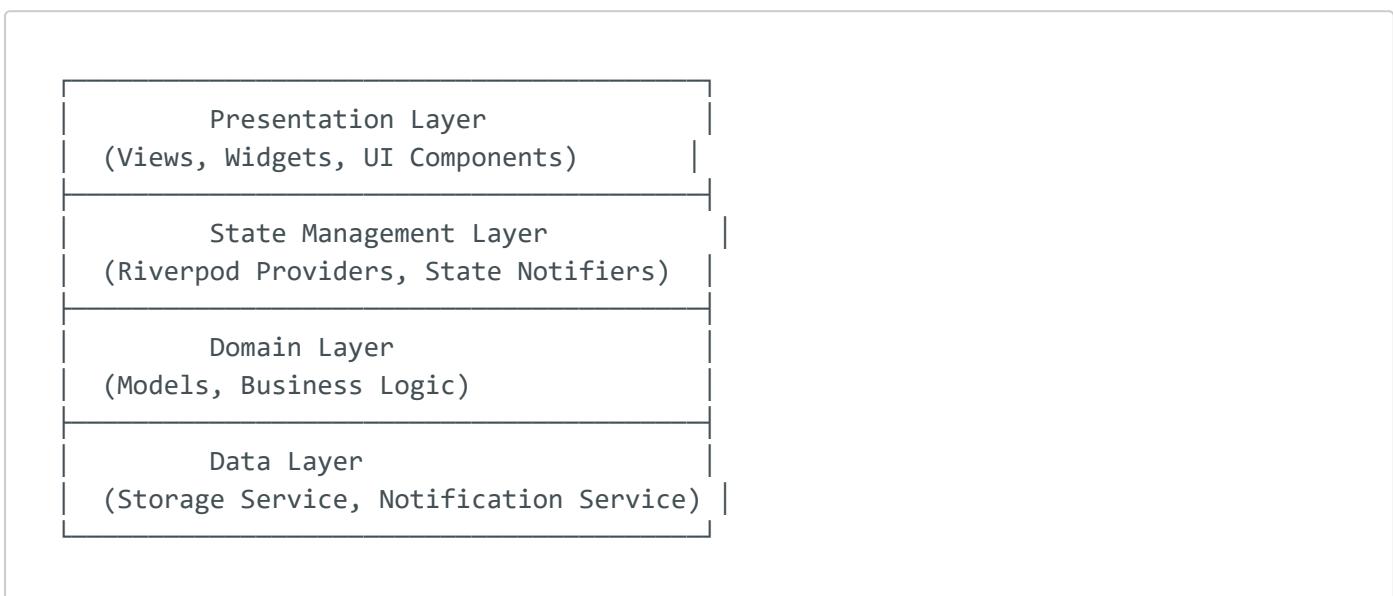
## 1.2 技术特性

- **跨平台支持：**Android、iOS、Web、Windows、macOS、Linux
- **离线优先：**所有数据本地存储，无需网络连接
- **高性能：**基于 Riverpod 的状态管理，最小化重建范围
- **可扩展性：**模块化架构，易于扩展新功能

## 二、程序概要设计

### 2.1 系统架构概述

本应用采用分层架构设计，遵循关注点分离和依赖倒置原则，主要分为以下层次：



### 2.2 模块划分

## 2.2.1 表示层 (Presentation Layer)

**职责：**负责用户界面展示和用户交互处理

- **Views (视图组件)**

- **DayView**: 日视图，显示单日时间轴
- **WeekView**: 周视图，显示一周事件
- **MonthView**: 月视图，显示月度日历
- **EventFormView**: 事件创建/编辑表单

- **Widgets (可复用组件)**

- **EventCard**: 事件卡片组件

## 2.2.2 状态管理层 (State Management Layer)

**职责：**管理应用状态，协调数据流

- **Providers**

- **calendarProvider**: 日历状态管理（当前视图、选中日期、事件列表）
- **eventProvider**: 事件列表状态管理（预留，用于扩展）
- **storageServiceProvider**: 存储服务提供者

- **State Notifiers**

- **CalendarNotifier**: 日历状态变更逻辑
  - 日期选择、视图切换
  - 事件增删改查
  - 事件与存储服务的同步

## 2.2.3 领域层 (Domain Layer)

**职责：**定义业务模型和业务规则

- **Models (数据模型)**

- **CalendarEvent**: 日历事件模型
  - 符合 RFC5545 标准的基本要求
  - 支持 UUID、时间、地点、描述、颜色、提醒设置
  - 支持重复规则 (RRULE)
  - 支持时区 (TZID)

- **ReminderSetting**: 提醒设置模型
  - 提前时间、提醒类型、时区
- **CalendarState**: 日历状态模型
  - 选中日期、当前视图、事件映射

## 2.2.4 数据层 (Data Layer)

职责：数据持久化和外部服务集成

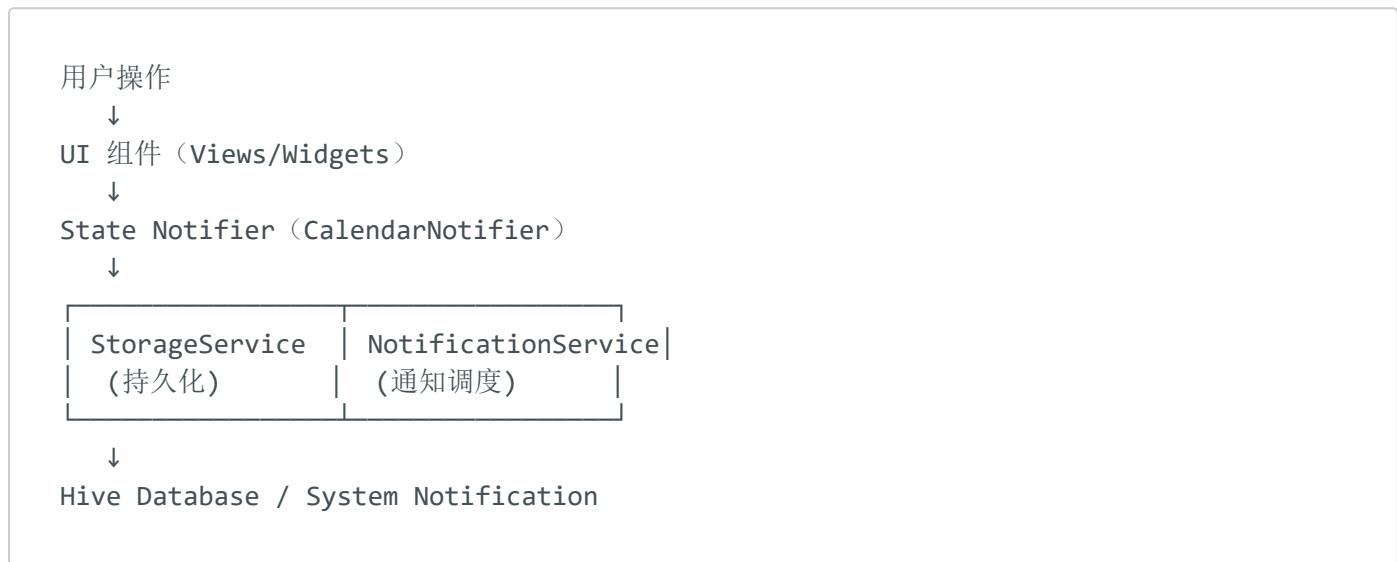
- **Services (服务)**

- **StorageService**: 本地存储服务
  - 基于 Hive 的键值存储
  - 事件 CRUD 操作
- **NotificationService**: 通知服务
  - 本地通知调度与管理
  - 时区感知的提醒触发
  - 通知权限管理

- **Utils (工具类)**

- **LunarUtils**: 农历日期计算工具

## 2.3 数据流设计



## 2.4 关键设计模式

### 1. Repository Pattern (仓储模式)

- `StorageService` 封装数据访问逻辑，便于替换存储实现

## 2. Provider Pattern (提供者模式)

- 使用 Riverpod 实现依赖注入和状态管理

## 3. State Notifier Pattern (状态通知者模式)

- `CalendarNotifier` 管理不可变状态，通过 `copyWith` 更新

## 4. Singleton Pattern (单例模式)

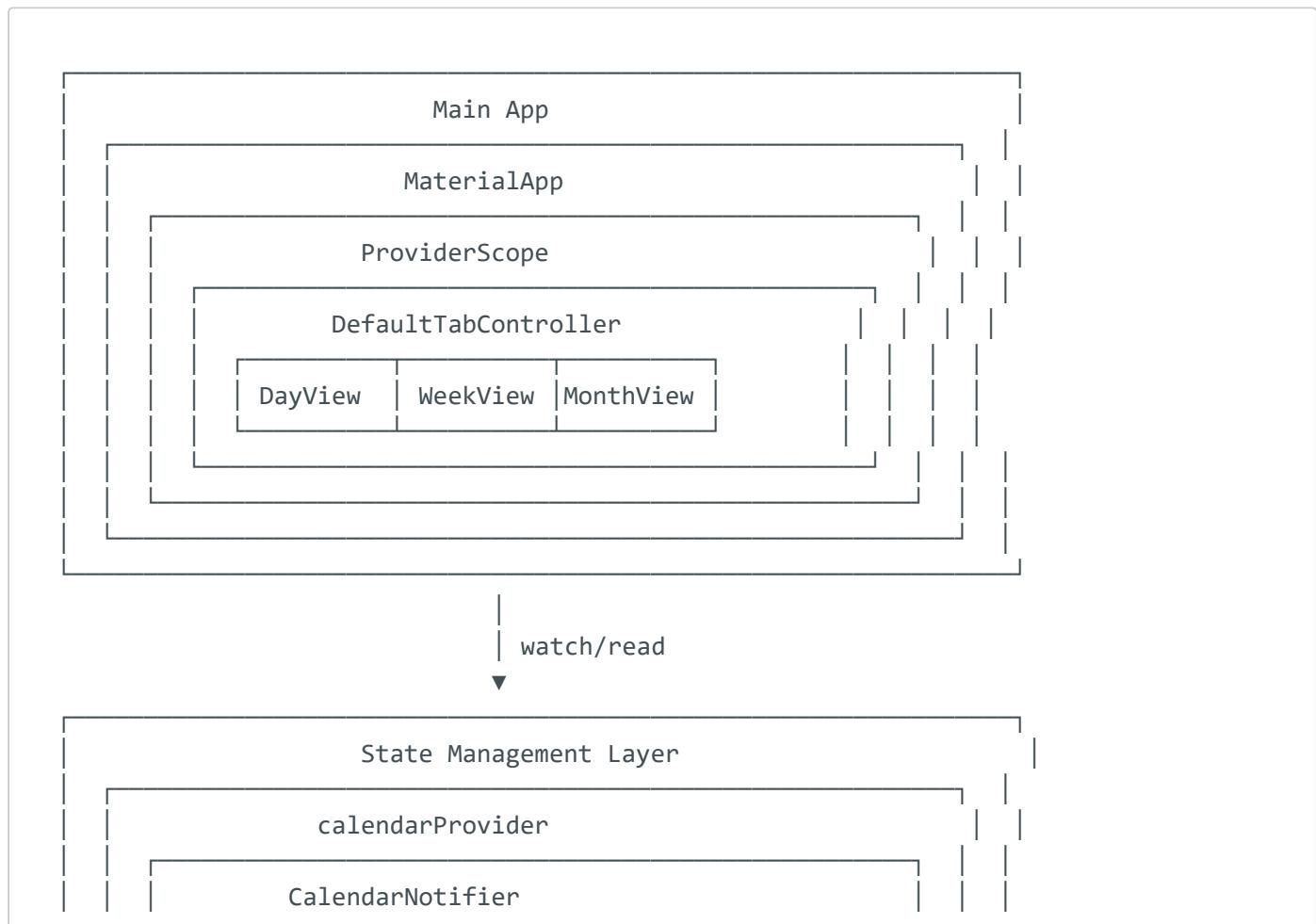
- `NotificationService` 使用单例确保全局唯一实例

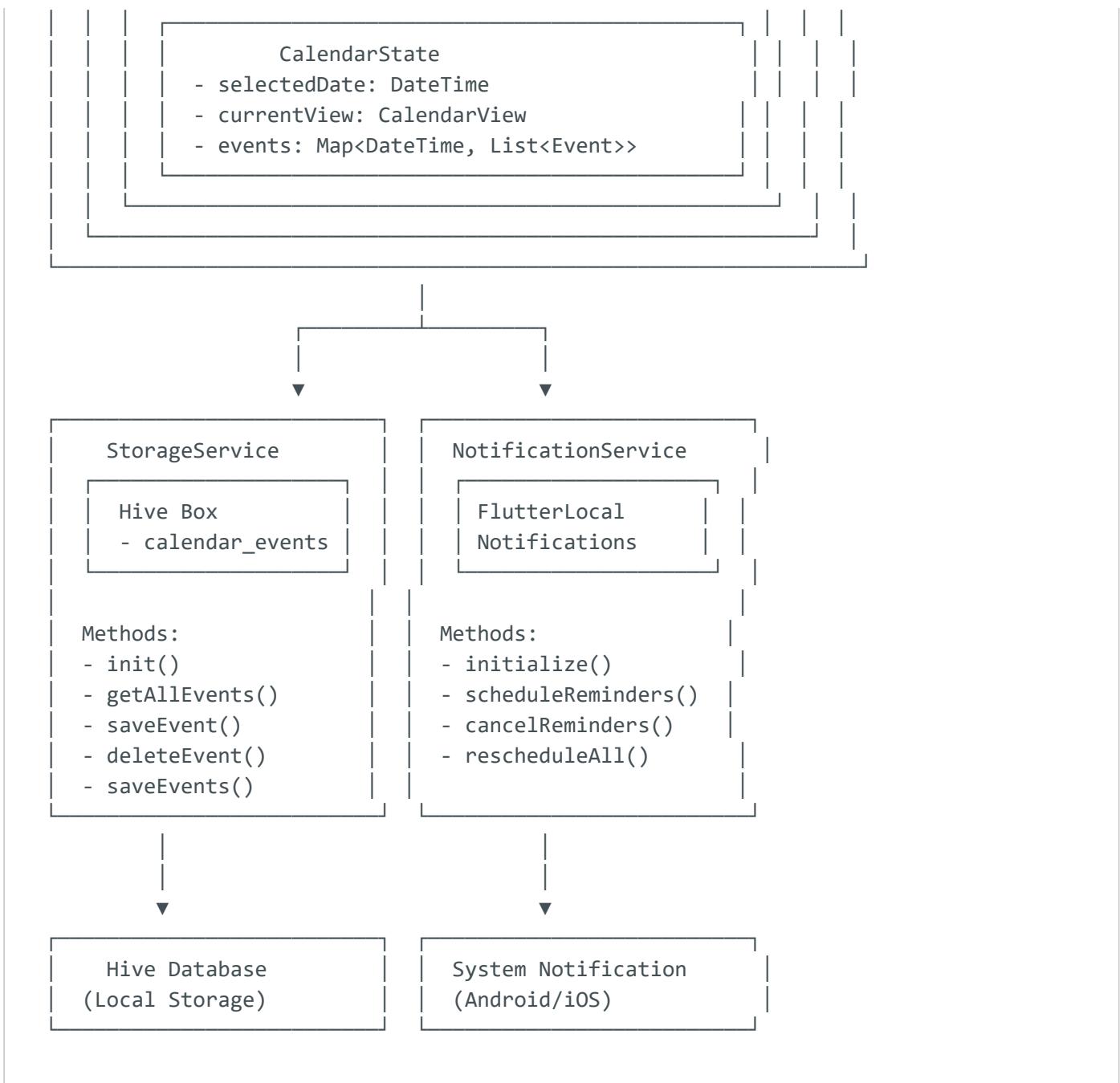
## 5. Adapter Pattern (适配器模式)

- Hive TypeAdapter 实现模型与存储格式的转换

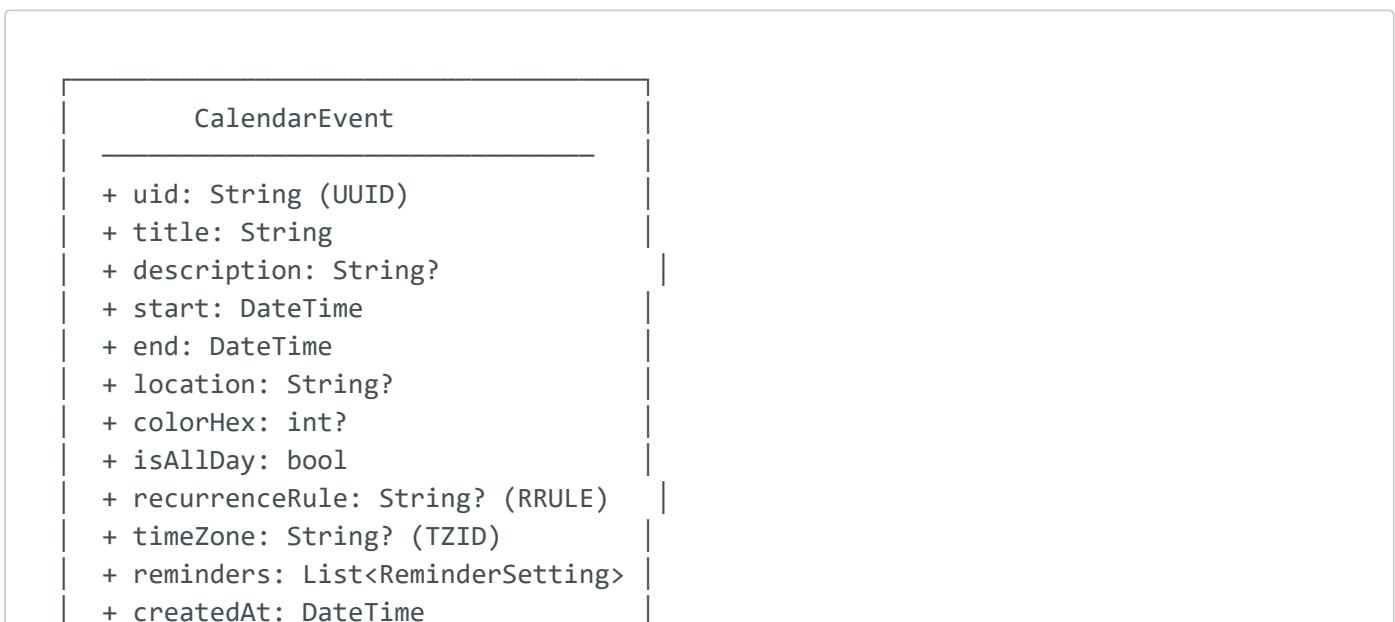
# 三、软件架构图

## 3.1 整体架构图





## 3.2 数据模型关系图



```
+ updatedAt: DateTime  
+  
+ isOnDate(date): bool  
+ overlapsWith(start, end): bool  
+ duration: Duration
```

contains

### ReminderSetting

```
+ id: String (UUID)  
+ beforeTime: Duration  
+ type: ReminderType  
+ timeZoneId: String?
```

## 3.3 状态流转图

User Interaction  
(Tap, Swipe, Form Submit)

UI Component  
(DayView, WeekView, EventFormView)

ref.read/write

### CalendarNotifier

```
+ selectDate(date)  
+ switchView(view)  
+ addEvent(event)  
+ updateEvent(event)  
+ removeEvent(uid)
```

state = state.copyWith(...)

CalendarState  
(Immutable State)

notifyListeners (Riverpod)

## 3.4 服务交互时序图

```
用户创建事件
|
↓
EventFormView
|
| submit
|
↓
CalendarNotifier.addEvent()
|
→ StorageService.saveEvent()
|
|
↓
Hive.put(event.uid, event)
|
|
↓
[Persisted]
|
→ NotificationService.scheduleReminders()
|
|
↓
FlutterLocalNotifications.zonedSchedule()
|
|
↓
[Scheduled Notification]
```

```
用户查看事件
|
↓
DayView/WeekView/MonthView
|
| ref.watch(calendarProvider)
|
↓
CalendarNotifier.eventsFor(date)
|
| (from CalendarState.events)
|
↓
[Display Events]
```

## 四、技术亮点及其实现原理

## 4.1 响应式状态管理 (Riverpod)

技术亮点：使用 Riverpod 实现声明式状态管理，实现 UI 与状态的自动同步。

实现原理：

- **Provider 定义**: 通过 `StateNotifierProvider` 创建状态提供者
- **不可变状态**: `CalendarState` 使用 `copyWith` 方法创建新状态，确保状态不可变
- **自动重建**: `Consumer` 组件监听 Provider 变化，自动重建依赖的 Widget
- **性能优化**: Riverpod 使用细粒度依赖追踪，只重建必要的 Widget

代码示例：

```
// Provider 定义
final calendarProvider = StateNotifierProvider<CalendarNotifier, CalendarState>
((ref) {
  final storageService = ref.watch(storageServiceProvider);
  return CalendarNotifier(storageService);
});

// 状态更新（不可变）
void addEvent(CalendarEvent event) {
  final updated = Map<DateTime, List<CalendarEvent>>.from(state.events);
  // ... 更新逻辑
  state = state.copyWith(events: updated); // 创建新状态
}

// UI 监听
Consumer(
  builder: (context, ref, _) {
    final calendar = ref.watch(calendarProvider); // 自动订阅
    // UI 构建
  },
)
```

优势：

- **类型安全**: 编译时检查状态类型
- **可测试性**: Provider 可独立测试
- **可组合性**: 多个 Provider 可组合使用

## 4.2 本地持久化存储 (Hive)

技术亮点：使用 Hive 实现高性能的本地键值存储，支持复杂对象序列化。

## 实现原理：

- **TypeAdapter 模式**: 为每个模型实现 **TypeAdapter**, 定义序列化/反序列化逻辑
- **二进制存储**: Hive 使用二进制格式存储, 读写速度快
- **类型安全**: 通过泛型确保存储类型一致性
- **延迟加载**: Box 按需打开, 减少启动时间

## 代码示例：

```
// 适配器实现
class CalendarEventAdapter extends TypeAdapter<CalendarEvent> {
    @override
    CalendarEvent read(BinaryReader reader) {
        final uid = reader.readString();
        final title = reader.readString();
        // ... 读取所有字段
        return CalendarEvent(uid: uid, title: title, ...);
    }

    @override
    void write(BinaryWriter writer, CalendarEvent obj) {
        writer.writeString(obj.uid);
        writer.writeString(obj.title);
        // ... 写入所有字段
    }
}

// 存储服务
class StorageService {
    Box<CalendarEvent>? _box;

    Future<void> init() async {
        _box = await Hive.openBox<CalendarEvent>('calendar_events');
    }

    Future<void> saveEvent(CalendarEvent event) async {
        await _box!.put(event.uid, event); // 键值存储
    }
}
```

## 优势：

- 性能：比 SQLite 快 2-3 倍
- 零依赖：纯 Dart 实现，无需原生代码
- 类型安全：编译时类型检查

## 4.3 时区感知的本地通知系统

**技术亮点：**实现跨时区的精确提醒调度，支持设备时区变化。

**实现原理：**

- **时区库集成：**使用 `timezone` 包加载时区数据
- **ZonedSchedule：**使用 `zonedSchedule` API 调度时区感知通知
- **精确模式：**Android 使用 `exactAllowWhileIdle` 模式确保精确触发
- **通知 ID 管理：**使用事件 UID 和提醒 ID 组合生成唯一通知 ID

**代码示例：**

```
// 时区初始化
tz.initializeTimeZones();

// 调度通知
await _notificationsPlugin.zonedSchedule(
  notificationId,
  event.title,
  _buildNotificationBody(event),
  tz.TZDateTime.from(triggerTime, tz.local), // 时区转换
  notificationDetails,
  androidScheduleMode: AndroidScheduleMode.exactAllowWhileIdle, // 精确模式
  matchDateTimeComponents: DateTimeComponents.time, // 时间匹配
  payload: event.uid, // 用于取消通知
);
```

**关键特性：**

- **时区转换：**自动处理设备时区变化
- **精确触发：**Android 12+ 支持精确提醒（需权限）
- **批量管理：**支持批量取消和重新调度
- **过期过滤：**自动跳过已过期事件的提醒

**权限处理：**

```
// Android 13+ 权限请求
await androidImplementation.requestNotificationsPermission();
await androidImplementation.requestExactAlarmsPermission(); // 精确提醒权限
```

## 4.4 RFC5545 标准兼容的事件模型

**技术亮点：**事件模型设计符合 RFC5545 (iCalendar) 标准，便于未来扩展和互操作。

## 实现原理：

- **UID 字段**: 使用 UUID v4 作为事件唯一标识（对应 RFC5545 UID）
- **RRULE 支持**: 预留重复规则字段，支持标准重复模式
- **TZID 支持**: 支持时区标识符，确保跨时区事件正确显示
- **VALARM 兼容**: 提醒设置模型对应 RFC5545 VALARM 组件

## 代码示例：

```
class CalendarEvent {  
    /// 唯一标识符 (UUID, 对应 RFC5545 UID)  
    final String uid;  
  
    /// 重复规则 (RRULE, 符合 RFC5545 标准)  
    /// 例如: "FREQ=DAILY;INTERVAL=1" 表示每天重复  
    final String? recurrenceRule;  
  
    /// 事件时间的时区标识 (对应 RFC5545 TZID)  
    final String? timeZone;  
  
    /// RFC5545 语义兼容的别名  
    String get id => uid;  
}
```

## 优势：

- **标准化**: 符合国际标准，便于与其他日历系统互操作
- **可扩展性**: 预留字段支持未来功能扩展
- **兼容性**: 可导出为标准 iCalendar 格式

## 4.5 高性能时间轴渲染（日视图）

**技术亮点：** 实现流畅的 24 小时时轴滚动，支持事件精确定位和同步滚动。

## 实现原理：

- **双 ScrollController 同步**: 时间标签和内容区域使用独立的 ScrollController，通过监听器实现同步滚动
- **防抖机制**: 使用 `_isSyncing` 标志防止循环同步
- **Stack 布局**: 使用 `Stack + Positioned` 实现事件块的精确定位
- **像素级计算**: 每分钟对应 1 像素，每小时 60 像素，实现精确的时间-位置映射

## 代码示例：

```
// 双控制器同步
void _syncFromContent() {
    if (_isSyncing) return; // 防止循环
    _isSyncing = true;
    _timeLabelController.jumpTo(_contentController.offset.clamp(...));
    _isSyncing = false;
}

// 事件定位计算
double _getEventTopPosition(DateTime startTime, DateTime dayStart) {
    final diff = startTime.difference(dayStart);
    return diff.inMinutes.toDouble(); // 分钟转像素
}

// Stack 布局定位
Positioned(
    top: topPixels,
    height: heightPixels,
    child: EventCard(event: event),
)
```

## 性能优化：

- **ListView.builder**: 时间标签使用 ListView.builder 实现虚拟滚动
- **最小高度限制**: 事件块最小高度 20 像素，确保可点击性
- **按需渲染**: 只渲染可见区域的事件

## 4.6 Material 3 设计系统集成

**技术亮点：**全面采用 Material 3 设计规范，提供现代化的 UI 体验。

## 实现原理：

- **ColorScheme**: 使用 ColorScheme.fromSeed 生成主题色彩
- **Material 3 Widgets**: 使用最新的 Material 3 组件（如 FloatingActionButton）
- **动态颜色**: 支持基于种子颜色的动态主题生成
- **圆角与阴影**: 使用 Material 3 的圆角半径和阴影规范

## 代码示例：

```
MaterialApp(
    theme: ThemeData(
```

```
        colorScheme: ColorScheme.fromSeed(  
            seedColor: const Color(0xFF1A73E8), // 种子颜色  
        useMaterial3: true, // 启用 Material 3  
)
```

## 设计特点：

- **一致性**: 遵循 Material 3 设计语言
- **可访问性**: 支持无障碍功能
- **响应式**: 适配不同屏幕尺寸

## 4.7 国际化与本地化

**技术亮点**: 完整的中文本地化支持，包括日期格式、农历显示、周次计算。

### 实现原理：

- **intl 包**: 使用 `intl` 包提供国际化支持
- **日期格式化**: 使用 `DateFormat` 进行本地化日期格式化
- **农历工具**: 自定义 `LunarUtils` 计算农历日期
- **周次算法**: 实现 ISO 8601 周次计算算法

### 代码示例：

```
// 初始化本地化  
await initializeDateFormatting('zh_CN', null);  
  
// 日期格式化  
DateFormat('yyyy年MM月dd日', 'zh_CN').format(date);  
DateFormat('EEEE', 'zh_CN').format(date); // 星期  
  
// 周次计算 (ISO 8601)  
int _getWeekNumber(DateTime date) {  
    final year = date.year;  
    final jan4 = DateTime(year, 1, 4);  
    final jan4weekday = jan4.weekday;  
    final firstWeekStart = jan4.subtract(Duration(days: jan4weekday - 1));  
    // ... 计算逻辑  
}
```

## 特性：

- **农历显示**: 显示农历日期、节气、节日
- **周次计算**: 符合 ISO 8601 标准的周次计算
- **日期格式**: 符合中文习惯的日期格式

## 4.8 模块化架构设计

**技术亮点**: 采用清晰的分层架构，实现高内聚、低耦合的代码组织。

**实现原理**:

- **目录结构**: 按功能模块组织代码 (models, providers, services, views, widgets)
- **依赖注入**: 通过 Riverpod Provider 实现依赖注入
- **接口抽象**: 服务层使用接口抽象，便于替换实现
- **单一职责**: 每个类/模块只负责一个功能

**目录结构**:

```
lib/
├── main.dart          # 应用入口
└── models/            # 数据模型
    ├── calendar_event.dart
    ├── calendar_state.dart
    └── reminder_settings.dart
└── providers/         # 状态管理
    ├── calendar_provider.dart
    └── event_provider.dart
└── services/          # 业务服务
    ├── storage_service.dart
    └── notification_service.dart
└── views/              # 视图组件
    ├── day_view.dart
    ├── week_view.dart
    ├── month_view.dart
    └── event_form_view.dart
└── widgets/            # 可复用组件
    └── event_card.dart
└── utils/              # 工具类
    └── lunar_utils.dart
```

**优势**:

- **可维护性**: 代码结构清晰，易于理解和修改
- **可测试性**: 各模块可独立测试
- **可扩展性**: 新功能可轻松添加到对应模块

# 五、技术栈总结

## 5.1 核心依赖

包名	版本	用途
flutter_riverpod	^2.5.1	状态管理
hive / hive_flutter	^2.2.3 / ^1.1.0	本地存储
flutter_local_notifications	^19.5.0	本地通知
timezone	^0.10.1	时区处理
intl	^0.20.2	国际化
table_calendar	^3.2.0	日历组件
uuid	^4.5.2	UUID 生成

## 5.2 开发工具

- **Flutter SDK:** 3.9.2+
- **Dart SDK:** 3.9.2+
- **Linter:** flutter\_lints ^5.0.0

## 5.3 平台支持

- Android
- iOS
- Web
- Windows
- macOS
- Linux

# 六、总结

本日历应用通过采用现代化的 Flutter 技术栈和清晰的分层架构，实现了功能完整、性能优异、用户体验良好的日程管理应用。主要技术亮点包括：

1. **响应式状态管理**: Riverpod 实现声明式 UI 更新
2. **高性能存储**: Hive 提供快速的本地数据持久化
3. **精确提醒系统**: 时区感知的本地通知调度
4. **标准化模型**: RFC5545 兼容的事件模型设计
5. **流畅交互**: 高性能时间轴渲染和同步滚动
6. **现代 UI**: Material 3 设计系统集成
7. **国际化支持**: 完整的中文本地化
8. **模块化架构**: 清晰的分层设计，易于维护和扩展

该应用为后续功能扩展（如云同步、重复事件、事件分享等）奠定了坚实的技术基础。