Emacs Lisp 游戏编程入门

王思成

Feb 05, 2026

Emacs Lisp 游戏编程拥有独特的魅力，它源于 Emacs 作为「终极编辑器」的无限扩展性。Emacs 不仅仅是一个文本编辑器，更是一个运行 Lisp 方言的完整运行时环境，这使得开发者能够将游戏逻辑无缝嵌入日常工作流中。选择 Emacs Lisp 开发游戏的原因在于其轻量级特性：无需复杂的构建管道，只需几行代码即可启动一个交互式游戏；其交互性极强，通过 `ielm` 或直接评估缓冲区内容，你可以实时调试游戏状态；此外，可视化调试工具如 edebug 让复杂逻辑一目了然；Emacs 社区还提供了丰富的游戏源码资源，从经典的 dunnet 文本冒险到现代 Tetris 实现，应有尽有。

本文面向 Emacs 用户、Lisp 爱好者和游戏开发入门者。如果你已经能熟练使用 Emacs 的基本命令，并理解 Lisp 的列表、函数和闭包概念，就可以跟随本文逐步构建完整游戏。文章从环境搭建开始，逐步深入基础概念、核心组件，然后通过 Snake 和 Tetris 两个实战项目展示完整实现，最后探讨高级主题和发布流程。通过这些内容，你将掌握在 Emacs 中创建互动娱乐的艺术。

文章结构清晰：先准备开发环境，然后讲解游戏基础概念，接着构建核心组件，通过两个项目实战演练高级技巧，并以发布和扩展建议收尾。先决条件包括基础 Emacs 使用经验和基本 Lisp 知识，如 `(car lst)` 和 `(defun ...)` 的用法。

1　环境准备

安装和配置 Emacs 是第一步。推荐使用 Emacs 27 或更高版本，这些版本内置了现代化的包管理器和性能优化。启动 Emacs 后，确保 `package.el` 已启用，通过在 `init.el` 中添加 `(require 'package)` 并配置 MELPA 仓库，即可安装扩展。`use-package` 是高效管理包的首选，它简化了依赖加载和配置，例如 `(use-package dash :ensure t)` 即可自动安装并加载 `dash.el` 函数式工具集。

必需的 Emacs Lisp 库包括内置的 `cl-lib`，提供通用 Lisp 函数如 `cl-loop` 和 `cl-find`；`subr-x` 从 Emacs 25 开始内置，用于字符串处理如 `string-trim`；`dash.el` 来自 MELPA，提供链式操作如 `->`；可选的 `emacs-game` 框架可在 GitHub 获取，用于快速搭建游戏骨架。这些库通过 `(require ' 库名)` 加载，确保在游戏代码前调用。

测试环境的最佳方式是编写一个“Hello World”游戏片段。考虑以下代码，它在专用缓冲区中显示问候并响应按键：

```
(defun hello-game ()
  "第一个␣Emacs␣Lisp␣游戏片段。"
  (interactive)
  (let ((buffer (get-buffer-create "*Hello␣Game*")))
    (switch-to-buffer buffer)
```

```
  (erase-buffer)
  (insert "欢迎来到␣Emacs␣游戏世界！按任意键继续，按␣q␣退出。\n")
  (let ((event (read-event)))
    (when (not (eq event ?q))
      (insert (format "你按了␣%s！" event))
      (hello-game)))))  ; 递归调用实现循环
```

这段代码首先创建名为 *Hello Game* 的缓冲区并切换到它，然后擦除内容并插入欢迎消息。read-event 阻塞等待用户输入事件，当输入不是 q 时，格式化显示按键并递归调用自身，形成简单循环。调用 (hello-game) 即可启动，体验 Emacs 缓冲区作为游戏画布的即时反馈。这种测试验证了环境就绪。

开发工具推荐包括 ielm（交互式评估模式，通过 M-x ielm 启动，用于逐行测试表达式）、调试模式（M-x debug-on-error 捕获运行时错误）和 edebug（用于函数级步进调试，例如 (edebug-defun my-function) 后 M-x edebug-my-function）。这些工具让游戏开发如鱼得水。

2 Emacs Lisp 游戏基础概念

Emacs 缓冲区天然充当游戏画布，它本质上是一个文本网格系统，每行由换行符分隔，每列由字符位置定义。通过插入字符、设置文本属性或使用覆盖（overlay），你可以实现动态渲染。例如，文本属性 (face 'highlight) 可为特定区域添加高亮，而覆盖允许在不修改底层文本的情况下叠加视觉效果。

游戏循环是核心，通常采用 while 循环形式。以下是典型实现：

```
(defun game-loop (state)
  "基础游戏循环：更新、渲染、输入。"
  (while (not (game-over-p state))
    (setq state (update-state state))
    (render state)
    (accept-input state)))
```

这里 state 是游戏状态（如玩家位置、分数），game-over-p 检查结束条件如碰撞。update-state 处理逻辑更新，如移动物体；render 重绘缓冲区；accept-input 读取用户事件并修改状态。setq 更新状态变量，确保循环中使用最新值。这种结构简单高效，适合终端式游戏。

输入处理支持事件驱动和轮询两种模式。事件驱动使用 read-event 阻塞等待，适合回合制游戏；轮询通过定时器定期检查键盘状态，适用于实时游戏。时间控制依赖 run-with-timer，例如 (run-with-timer 0.1 nil #'game-tick state) 每 0.1 秒调用一次游戏刻（tick），实现帧率管理。帧率通过调整间隔控制，例如 60 FPS 对应约 16ms 间隔，但需注意 Emacs 的单线程性质，避免阻塞 UI。

3 核心游戏组件

游戏状态管理采用 alist 或 plist 结构，便于扩展。例如，Snake 游戏状态可表示为 ((pos (5 5)) (dir right) (score 0))，通过 (assoc 'pos state) 访问位置，(plist-get state :dir) 处理 plist。自定义 struct 使用 cl-defstruct，如 (cl-defstruct game-state pos dir score)，提供访问器如 game-state-pos。状态序列化通过 prin1-to-string 转为字符串保存至文件，反序列化用 read 加载，支持存档

功能。

渲染系统从纯文本起步，向高级方法演进。纯文本简单兼容，Unicode 块字符如 ■ 提升视觉；覆盖支持动画效果；图像通过 image.el 显示 PNG。考虑一个简单渲染函数：

```
(defun render (state)
  "渲染游戏状态到缓冲区。"
  (with-current-buffer (get-buffer-create "*Game*")
    (erase-buffer)
    (let ((pos (cdr (assoc 'pos state))))
      (goto-char (point-min))
      (insert "游戏画布␣\n")
      (dotimes (row 20)
        (dotimes (col 40)
          (if (equal (list row col) pos)
              (insert "■") ; 玩家位置
            (insert "␣")))
        (insert "\n")))))
```

此函数切换到游戏缓冲区，擦除旧内容，在 20x40 网格中定位玩家 (row col) 并插入块字符 ■，其余填充空格。dotimes 实现嵌套循环模拟网格，goto-char 和 insert 操作缓冲区内容。这种网格渲染适用于 Roguelike 游戏。

输入处理使用 read-key 或 read-event。键盘事件如 (let ((key (read-key))) (pcase key (?w (update-dir 'up)) ...)) 通过 pcase 模式匹配处理方向键；鼠标支持 read-event 捕获点击坐标；自定义绑定通过 (local-set-key (kbd C-c C-g) #'game-toggle) 在游戏缓冲区设置快捷键。碰撞检测实现网格 AABB（轴对齐包围盒）：对于位置 (x1 y1) 和 (x2 y2)，若 and (<= x1 x2) (<= y1 y2) (>= x1 x2) (>= y1 y2) 则碰撞。简单物理模拟添加速度和摩擦，例如 new-pos = (list (+ x (* vx dt)) (+ y (* vy dt)))，dt 为时间步长。

4 实战项目 1：Snake（贪吃蛇）

Snake 游戏的核心循环是蛇移动、吃食物生长、碰撞检测结束。评分基于食物数量，结束条件包括撞墙或自撞。游戏设计文档简述：20x20 网格，蛇初始长度 3，食物随机生成。

逐步实现从状态定义开始：

```
(defvar snake-game-buffer nil)
(defvar snake-state '((snake ((1 1) (1 2) (1 3))) (dir right) (food (10 10)) (score
    ↪ 0)))

(defun init-snake ()
  "初始化蛇游戏。"
  (setq snake-game-buffer (get-buffer-create "*Snake*"))
  (switch-to-buffer snake-game-buffer)
```

```
  (snake-render snake-state)
  (local-set-key (kbd "q") #'snake-quit)
  (local-set-key (kbd "<up>") (lambda () (interactive) (snake-turn 'up)))
  ;; 类似绑定 down, left, right
  (run-with-timer 0.2 0.2 #'snake-update))
```

snake-state 使用 alist：snake 是链表表示蛇身，从头到尾；dir 为当前方向；food 为食物位置；score 计分。init-snake 创建缓冲区，渲染初始状态，绑定方向键和退出键 q，启动 0.2 秒间隔的定时器驱动更新。方向绑定使用 lambda 捕获 interactive 标记，确保菜单可见。

渲染函数如下：

```
(defun snake-render (state)
  "渲染蛇游戏。"
  (with-current-buffer snake-game-buffer
    (erase-buffer)
    (insert (format "分数␣:␣%d\n" (cdr (assoc 'score state))))
    (dotimes (row 21)
      (dotimes (col 25)
        (let ((pos (list row col)))
          (cond ((member pos (cdr (assoc 'snake state)))
                 (insert "■"))
                ((equal pos (cdr (assoc 'food state)))
                 (insert "●"))
                (t (insert "␣")))))
      (insert "\n"))))
```

类似前述网格渲染，此处检查位置是否在蛇身链表中（member），或匹配食物则插入圆点 ●，其余空格。format 显示分数。

更新逻辑核心：

```
(defun snake-update ()
  "蛇游戏更新刻。"
  (let* ((state snake-state)
         (snake (cdr (assoc 'snake state)))
         (head (car snake))
         (dir (cdr (assoc 'dir state)))
         (new-head (pcase dir
                     ('up (list (1- (car head)) (cadr head)))
                     ('down (list (1+ (car head)) (cadr head)))
                     ('left (list (car head) (1- (cadr head))))
                     ('right (list (car head) (1+ (cadr head))))))
         (new-snake (cons new-head snake))
```

```
      (score (cdr (assoc 'score state))))
  (when (or (< (car new-head) 1) (> (car new-head) 20)
            (< (cadr new-head) 1) (> (cadr new-head) 24)
            (member new-head (cdr snake)))
    (snake-game-over))
  (when (equal new-head (cdr (assoc 'food state)))
    (setq score (1+ score))
    (setq new-snake (cons new-head snake)) ; 不移除尾巴，生长
    (setq state (snake-new-food state)))
  (setq snake-state (list (cons 'snake new-snake)
                          (cons 'dir dir)
                          (cons 'food (cdr (assoc 'food state)))
                          (cons 'score score)))
  (snake-render snake-state)))
```

计算新头位置基于方向，pcase 匹配计算坐标偏移。新蛇为 (cons new-head old-snake)。边界检查若超出 1-20 行或 1-24 列，或新头撞上蛇身（除尾），则游戏结束。吃食物时分数增 1，不移除尾巴实现生长，并生成新食物。最终更新全局 snake-state 并重绘。蛇身链表自动管理长度，此实现捕捉了贪吃蛇精髓。

关键技术包括蛇身链表：头插入新位置，正常移动时需移除尾巴（此处简化，未显式移除以示生长逻辑）；食物随机生成用 (list (+ 1 (random 20)) (+ 1 (random 24)))；边界反弹扩展可修改新头计算为折返。调用 (init-snake) 启动完整游戏。

5 实战项目 2：Tetris（俄罗斯方块）

Tetris 设计围绕 Tetrominoes：七种方块形状，如 I 形 [[1,1,1,1]]、O 形等。核心循环包括落块、旋转、行消除、难度递增。

方块形状定义为旋转状态列表，每个状态是 4x4 矩阵偏移。核心算法从旋转开始：使用变换矩阵计算新位置。例如，逆时针旋转公式为 $x' = x\cos\theta - y\sin\theta$， $y' = x\sin\theta + y\cos\theta$，$\theta = 90^\circ$ 时简化为 $(x', y') = (y, -x)$。

渲染优化使用 Unicode 方块和颜色：

```
(defface tetris-block '((t :background "blue" :foreground "white"))
  "Tetris 方块样式。")

(defun tetris-render (state)
  "渲染俄罗斯方块。"
  (with-current-buffer "*Tetris*"
    (erase-buffer)
    (let ((board (cdr (assoc 'board state)))
          (piece (cdr (assoc 'current-piece state)))
          (pos (cdr (assoc 'pos state))))
      (dotimes (row 22)
```

```
      (dotimes (col 12)
       (let ((cell (aref (aref board row) col)))
         (if (or (/= cell 0)
                 (tetris-piece-at-p piece pos row col))
             (progn
               (put-text-property (point) (1+ (point))
                                  'face 'tetris-block)
               (insert "■ "))
           (insert "□ "))))))
      (insert "\n")))))
```

board 是 22x12 数组（vector of vector），0 表示空。tetris-piece-at-p 检查当前方块是否覆盖 (row col)。put-text-property 为块设置面（face），实现彩色渲染。空格用 □ 填充。
行消除扫描完整行：

```
(defun tetris-clear-lines (board)
  "清除满行并下移。"
  (let ((new-board (make-vector 22 (make-vector 12 0))))
    (let ((write-row 0))
      (dotimes (read-row 22)
        (let ((full (cl-every (lambda (x) (/= x 0)) (aref board read-row))))
          (if full
              (cl-incf (cdr (assoc 'lines-cleared state))) ; 更新分数
            (aset new-board write-row (copy-sequence (aref board read-row)))
            (cl-incf write-row)))))
    (list 'board new-board)))
```

cl-every 检查行是否全非零，若满行则跳过，下移其余行至 new-board。分数基于清除行数递增。
落块预测用模拟下移检查碰撞，难度通过缩短定时器间隔实现。此项目整合了旋转矩阵、数组操作和属性渲染，完整代码约 200 行。

6 高级主题

声音支持通过 play-sound API，例如 (play-sound eat.wav)，需预置音频文件，支持 MIDI/OGG 格式嵌入资源目录。
多人游戏利用 Emacs 服务器模式 (server-start)，通过 emacsclient 连接多实例；网络集成简单 WebSocket 使用 websocket.el 包，发送 JSON 序列化状态。
性能优化聚焦渲染局部重绘：维护脏矩形列表，仅更新变化区域，避免全擦除；垃圾回收用 (garbage-collect) 在低负载时手动触发；缓冲区通过 (bury-buffer) 隐藏非活跃游戏。
跨平台打包用 make 生成独立 tarball，easy-install.el 简化用户安装。测试框架推荐 buttercup，编写断言如 (it should move snake (expect new-head :to-equal expected))。

7 游戏发布和社区分享

MELPA 打包需 `package-lint` 检查，创建 `.el` 和 `.pkg.el`，提交至 MELPA 仓库。GitHub Pages 可托管在线 Demo，通过 `js-emacs` 模拟 Emacs 环境。
Emacs Lisp 游戏社区资源丰富，如 GitHub 上的 `tetris.el` 提供旋转优化，内置 `dunnet` 展示文本冒险，itch.io 的 Emacs Game Jam 鼓励参赛。常见问题包括定时器泄漏（用 `cancel-timer` 清理）和缓冲区焦点丢失（用 `select-window` 修复）。

8 扩展阅读和项目挑战

进阶项目从 Roguelike 开始，焦点程序生成地图和视野锥（FOV）算法如阴影投射；Platformer 需物理引擎模拟跳跃，重力 $a = -9.8$ dt²；Puzzle 使用 A* 状态搜索；FPS 挑战 3D 投影和射线追踪。
推荐书籍包括《Mastering Emacs》详解扩展、《Land of Lisp》趣味游戏章节，以及 Emacs Lisp 游戏源码合集。

9 结论

通过本文，你已掌握 Emacs Lisp 游戏开发的完整链路，从缓冲区画布到复杂模拟。Emacs Lisp 游戏体现了 Lisp 的简洁哲学：代码即数据，调试即交互。这不仅是技术实践，更是重塑生产力的艺术。现在，动手开发你的第一款游戏，加入社区分享成果！

10 附录

完整代码仓库见 GitHub emacs-game-examples。常用函数速查：`read-event` 输入、`run-with-timer` 循环、`overlay-put` 动态效果。故障排除：定时器不触发检查 `timer-list`，渲染卡顿启用 `garbage-collect-at-exit`。鸣谢 Emacs 社区贡献者。